

Applied Operating System

Chapter 1: Operating Systems Types and Structure

Prepared By:
Amit K. Shrivastava
Asst. Professor
Nepal College Of Information Technology

1.1 Introduction and History of Operating Systems

- An operating system acts as an intermediary between the user of a computer and the computer hardware. The purpose of an Operating system is to provide an environment in which a user can execute programs in a convenient and efficient manner.
- An operating system is software that manages the computer hardware. The hardware must provide appropriate mechanisms to ensure the correct operation of the computer system and to prevent user programs from interfering with the proper operation of the system.

History of Operating System:

- Operating systems have been evolving through the years. Since operating systems have historically been closely tied to the architecture of the computers on which they run, we will look at successive generations of computers to see what their operating systems were like.

The First Generation (1945- 55) Vacuum Tubes and Plug boards

- First of all calculating engines is built and mechanical relays were used but were very slow, with cycle times measured in seconds. Relays were later replaced by vacuum tubes. All programming was done in absolute machine language, often by wiring up plugboards to control the machine's basic functions. By the early 1950s, the routine had improved somewhat with the introduction of punched cards. It was now possible to write programs on cards and read them in instead of using plugboards;

History of Operating System(contd..):

The Second Generation (1955-65) Transistors and Batch Systems

▪_The introduction of the transistor in the mid-1950s changed the picture radically. These machines, now called mainframes, and to run a job a programmer would first write the program on paper (in FORTRAN or possibly even in assembly language), then punch it on cards. But it was time consuming and costly. The solution generally adopted was the batch system. The idea behind it was to collect a tray full of jobs in the input room and then read them onto a magnetic tape using a small (relatively) inexpensive computer, such as the IBM 1401.

The Third Generation (1965-1980) ICs and Multiprogramming

▪_The 360 developed by IBM was the first major computer line to use (small-scale) Integrated Circuits (ICs), thus providing a major price/ performance advantage over the second-generation machines, which were built up from individual transistors. And the most important advantage was multiprogramming. Here, the memory was partitioned into several pieces, with a different job in each partition, while one job was waiting for the I/O to complete, another job could be using the CPU.

History of Operating System(contd..):

The Fourth Generation (1980Present) Personal Computers

With the development of LSI (Large Scale Integration) circuits chips containing thousands of transistors on a square centimeter of silicon, the age of personal computer dawned. First kildall wrote a disk base operating system called CP/M(Control Program for Microcomputers) for Intel in 1974, then in early 1980's DOS(Disk Operating System) was invented and after that Microsoft revised it and renamed MS-DOS(Microsoft Disk Operating System). All these operating systems were all based users typing in commands from the keyboard after that GUI(Graphical User Interface) was invented, complete with windows, icons, menus, and mouse. After that different version of windows and Unix came in to light.

Operating System Types:

- Batch Systems
- Time-Sharing Systems
- Personal-Computer Systems
- Parallel Systems
- Real Time Systems
- Distributed Systems

Batch Systems

- Jobs with similar needs are batched together and run through the computer as a group by an operator or automatic job sequencer. Performance is increased by attempting to keep CPU and I/O devices busy at all times through buffering, off-line operation, spooling, and multiprogramming. Batch is good for executing large jobs that need little interaction; it can be submitted and picked up later. The problems with Batch Systems are following.
 - Lack of interaction between the user and job.
 - Difficult to provide the desired priority.

Time-Sharing Systems

- This systems uses CPU scheduling and multiprogramming to provide economical interactive use of a system. The CPU switches rapidly from one user to another. Instead of having a job defined by spooled card images, each program reads its next control card from the terminal, and output is normally printed immediately to the screen. Advantages of Timesharing operating systems are - Provide advantage of quick response, Avoids duplication of software, Reduces CPU idle time.

Personal-Computer Systems

■ A Personal Computer(PC) is a small, relatively inexpensive computer designed for an individual user. All are based on the microprocessor technology that enables manufacturers to put an entire CPU on one chip. At home, the most popular use for personal computers is for playing games. Businesses use personal computers for word-processing, accounting, desktop publishing, and for running spreadsheet and database management applications. The goals of these operating systems is not only maximizing CPU and peripheral utilization, but also maximizing user convenience and responsiveness.

Parallel Systems

■ Parallel operating systems are used to interface multiple networked computers to complete tasks in parallel. The architecture of the software is often a UNIX-based platform, which allows it to coordinate distributed loads between multiple computers in a network. Parallel operating systems are able to use software to manage all of the different resources of the computers running in parallel, such as memory, caches, storage space, and processing power. Parallel operating systems also allow a user to directly interface with all of the computers in the network. Its one advantage is increased throughput.

Real-Time Systems

▪Often used in a dedicated application, this system reads information from sensors and must respond within a fixed amount of time to ensure correct performance. In this Response Time is already fixed. Means time to Display the Results after Possessing has fixed by the Processor or CPU. Real Time System is used at those Places in which we Requires higher and Timely Response.

Distributed Systems

▪This system distributes computation among several physical processors. The processors do not share memory or a clock. Instead, each processor has its own local memory. They communicate with each other through various communication lines, such as a high-speed bus or telephone line. The advantages of distributed systems are- With resource sharing facility user at one site may be able to use the resources available at another, Speedup the exchange of data with one another via electronic mail, If one site fails in a distributed system, the remaining sites can potentially continue operating, Better service to the customers, Reduction of the load on the host computer.

❖ Operating System as Resource Manager:

- Operating system is collection of software which is close to hardware. We can view operating system as a resource – hardware and software collector. A system has many hardware and software that may be required to solve the problem, cpu time, memory space, file storage space, i/o device etc. the operating system acts as manager of these resources.
- Modern computer consists of process, memories, times, disks, network, printer and wide varieties of other devices. The task of the operating system is to provide for an orderly and controlled allocation of the process, memories and I/O devices among the various programs completing for them.
- An operating system is a control program, a control program manages the execution of user program to prevent errors and improve use of computer. It is especially concerned with the operation and control of I/O devices. When a computer has multiple users the operating system manages and protects the memory I/O devices. The operating system keeps in trace that who is using which resource to grant resource required amount for usage and to mediate conflicting required different programs and users.

❖ **Operating System as Extended Machine:**

- The program that hides the truth about the hardware from the programmer and presents a nice, simple view of named files that can be read and written is, of course, the operating system. Just as the operating system shields the programmer from the disk hardware and presents a simple file-oriented interface, it also conceals a lot of unpleasant business concerning interrupts, timers, memory management, and other low-level features. In each case, the abstraction offered by the operating system is simpler and easier to use than that offered by the underlying hardware.
- In this view, the function of the operating system is to present the user with the equivalent of an **extended machine** or **virtual machine** that is easier to program than the underlying hardware. To summarize it in a nutshell, the operating system provides a variety of services that programs can obtain using special instructions called system calls.

Operating-System Structures

System Components: Common System Components are

- Process Management
- Main Memory Management
- Secondary-Storage Management
- I/O System Management
- File Management
- Protection System
- Networking
- Command-Interpreter System

Process Management

- A process is a program in execution. A process needs certain resources, including CPU time, memory, files, and I/O devices, to accomplish its task.
- The operating system is responsible for the following activities in connection with process management.
 - Process creation and deletion.
 - process suspension and resumption.
 - Provision of mechanisms for:
 - * process synchronization
 - * process communication

Main-Memory Management

- Memory is a large array of words or bytes, each with its own address. It is a repository of quickly accessible data shared by the CPU and I/O devices.
- Main memory is a volatile storage device. It loses its contents in the case of system failure.
- The operating system is responsible for the following activities in connections with memory management:
 - Keep track of which parts of memory are currently being used and by whom.
 - Decide which processes to load when memory space becomes available.
 - Allocate and deallocate memory space as needed.

Secondary-Storage Management

- Since main memory (primary storage) is volatile and too small to accommodate all data and programs permanently, the computer system must provide secondary storage to back up main memory.
- Most modern computer systems use disks as the principle on-line storage medium, for both programs and data.
- The operating system is responsible for the following activities in connection with disk management:
 - Free space management
 - Storage allocation
 - Disk scheduling

I/O System Management

- The I/O system consists of:
 - A buffer-caching system
 - A general device-driver interface
 - Drivers for specific hardware devices

File Management

- A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data.
- The operating system is responsible for the following activities in connections with file management:
 - File creation and deletion.
 - Directory creation and deletion.
 - Support of primitives for manipulating files and directories.
 - Mapping files onto secondary storage.
 - File backup on stable (nonvolatile) storage media.

Protection System

- *Protection refers to a mechanism for controlling access by programs, processes, or users to both system and user resources.*
- The protection mechanism must:
 - distinguish between authorized and unauthorized usage.
 - specify the controls to be imposed.
 - provide a means of enforcement.

Networking (Distributed Systems)

- A *distributed system* is a collection processors that do not share memory or a clock. Each processor has its own local memory.
- The processors in the system are connected through a communication network.
- A distributed system provides user access to various system resources.
- Access to a shared resource allows:
 - Computation speed-up
 - Increased data availability
 - Enhanced reliability

Command-Interpreter System

- Many commands are given to the operating system by control statements which deal with:
 - process creation and management
 - I/O handling
 - secondary-storage management
 - main-memory management
 - file-system access
 - protection
 - networking

Operating System Services

- Program execution – system capability to load a program into memory and to run it.
- I/O operations – since user programs cannot execute I/O operations directly, the operating system must provide some means to perform I/O.
- File-system manipulation – program capability to read, write, create, and delete files.
- Communications – exchange of information between processes executing either on the same computer or on different systems tied together by a network. Implemented via *shared memory* or *message passing*.
- Error detection – ensure correct computing by detecting errors in the CPU and memory hardware, in I/O devices, or in user.

Additional Operating System Functions

Additional functions exist not for helping the user, but rather for ensuring efficient system operations.

- Resource allocation – allocating resources to multiple users or multiple jobs running at the same time.
- Accounting – keep track of and record which users use how much and what kinds of computer resources for account billing or for accumulating usage statistics.
- Protection – ensuring that all access to system resources is controlled.

System Calls

- System calls provide the interface between a running program and the operating system.
 - Generally available as assembly-language instructions.
 - Languages defined to replace assembly language for systems programming allow system calls to be made directly.
- Three general methods are used to pass parameters between a running program and the operating system.
 - Pass parameters in registers.
 - Store the parameters in a table in memory, and the table address is passed as a parameter in a register.
 - Push (store) the parameters onto the stack by the program, and pop off the stack by operating system.

System Calls(contd..)

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

- **Process control**

- end, abort
- load, execute
- create process, terminate process
- get process attributes, set process attributes
- wait for time
- wait event, signal event
- allocate and free memory

- **File management**

- create file, delete file
- open, close
- read, write, reposition
- get file attributes, set file attributes

System Calls(contd..)

- **Device management**

- request device, release device
- read, write, reposition
- get device attributes, set device attributes
- logically attach or detach devices

- **Information maintenance**

- get time or date, set time or date
- get system data, set system data
- get process, file, or device attributes
- set process, file, or device attributes

- **Communications**

- create, delete communication connection
- send, receive messages
- transfer status information
- attach or detach remote devices

System Programs

- System programs provide a convenient environment for program development and execution. They can be divided into:
 - File manipulation
 - Status information
 - File modification
 - Programming language support
 - Program loading and execution
 - Communications
 - Application programs
- Most users' view of the operation system is defined by system programs, not the actual system calls.

System Programs

- System programs provide a convenient environment for program development and execution. They can be divided into:
 - File manipulation
 - Status information
 - File modification
 - Programming language support
 - Program loading and execution
 - Communications
 - Application programs
- Most users' view of the operation system is defined by system programs, not the actual system calls.

System Structure – Simple Approach

- MS-DOS – written to provide the most functionality in the least space
 - not divided into modules
 - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated.

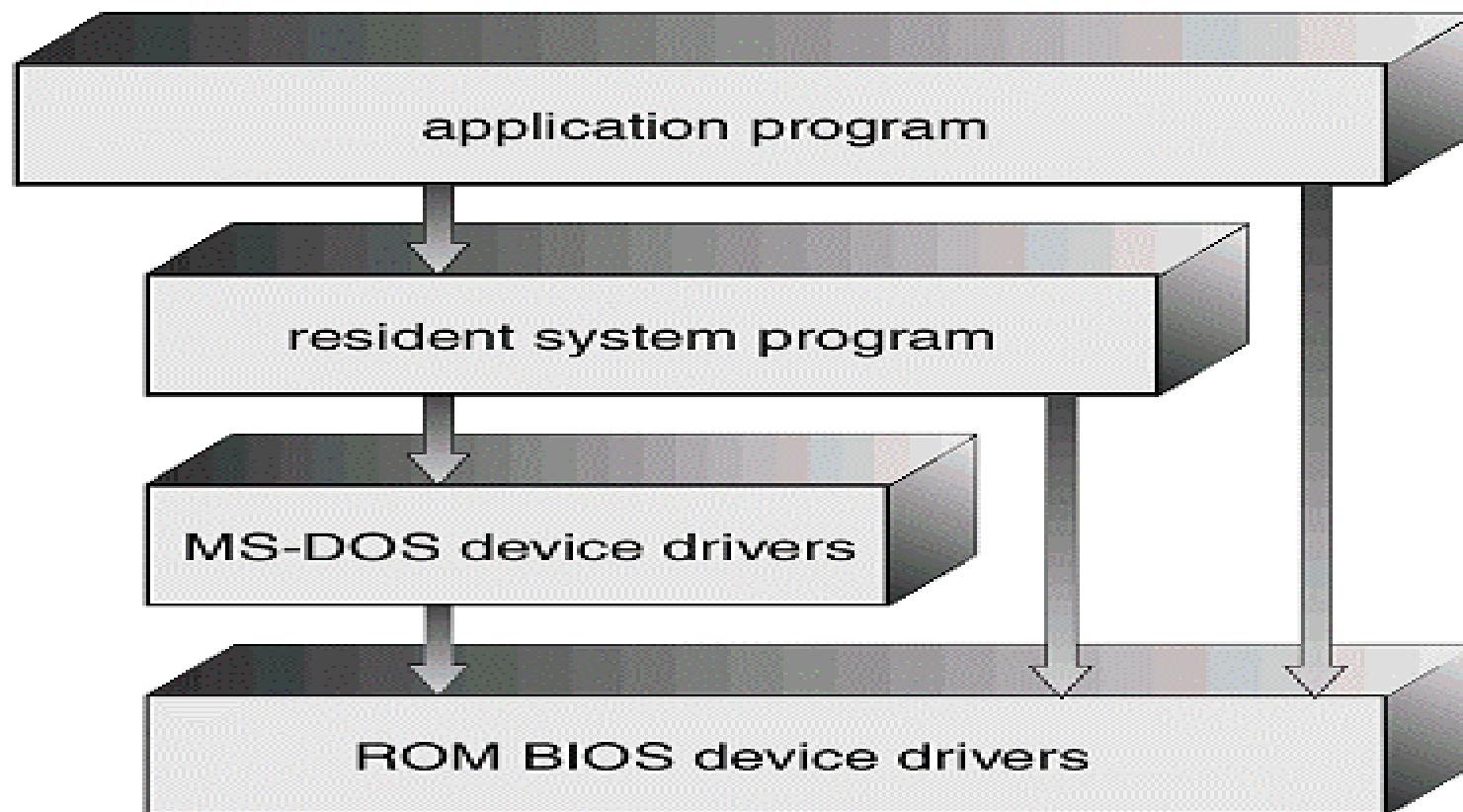
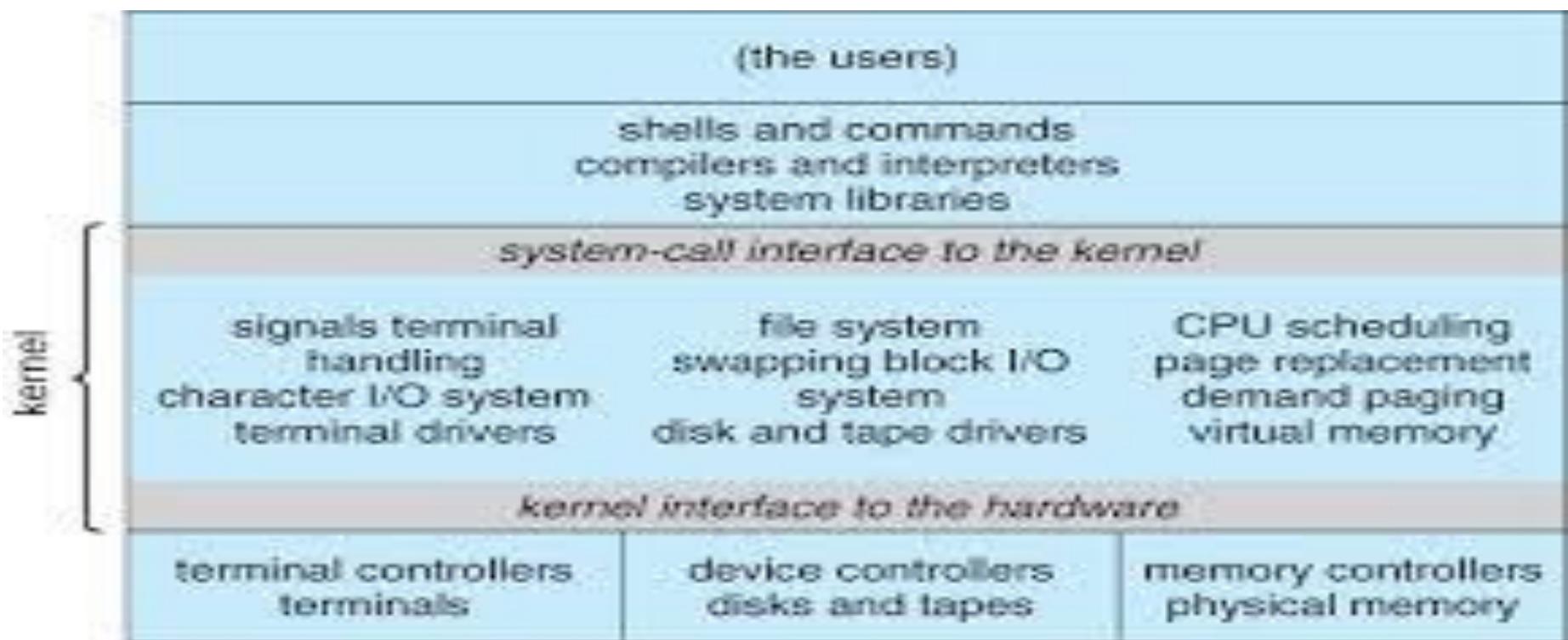


Fig: MS-DOS Layer Structure

System Structure – Simple Approach (Cont.)

- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts.
 - Systems programs
 - The kernel
 - * Consists of everything below the system-call interface and above the physical hardware.
 - * Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level.



System Structure – Layered Approach

- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers.

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

Structure of the THE operating system.

Layered Systems(contd..)

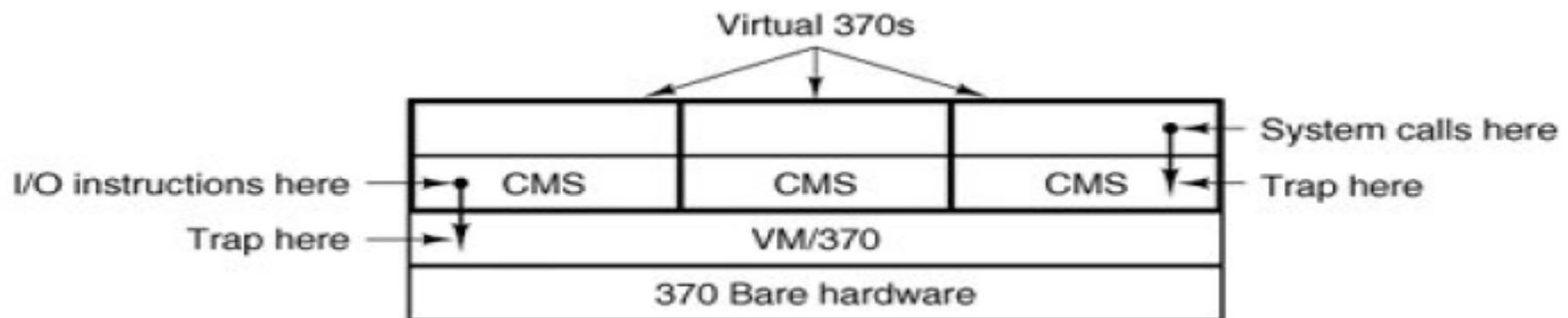
- A further generalization of the layering concept was present in the MULTICS system. Instead of layers, MULTICS was organized as a series of concentric rings, with the inner ones being more privileged than the outer ones. When a procedure in an outer ring wanted to call a procedure in an inner ring, it had to make the equivalent of a system call, that is, a TRAP instruction whose parameters were carefully checked for validity before allowing the call to proceed.

Virtual Machines:

- A *virtual machine* takes the *layered approach to its logical conclusion*. It treats hardware and the operating system kernel as though they were all hardware.
- A virtual machine provides an interface *identical to the underlying bare hardware*.
- The operating system creates the illusion of multiple processes, each executing on its own processor with its own (virtual) memory.
- The resources of the physical computer are shared to create the virtual machines.
 - CPU scheduling can create the appearance that users have their own processor.
 - Spooling and a file system can provide virtual card readers and virtual line printers.
 - A normal user time-sharing terminal serves as the virtual machine operator's console.

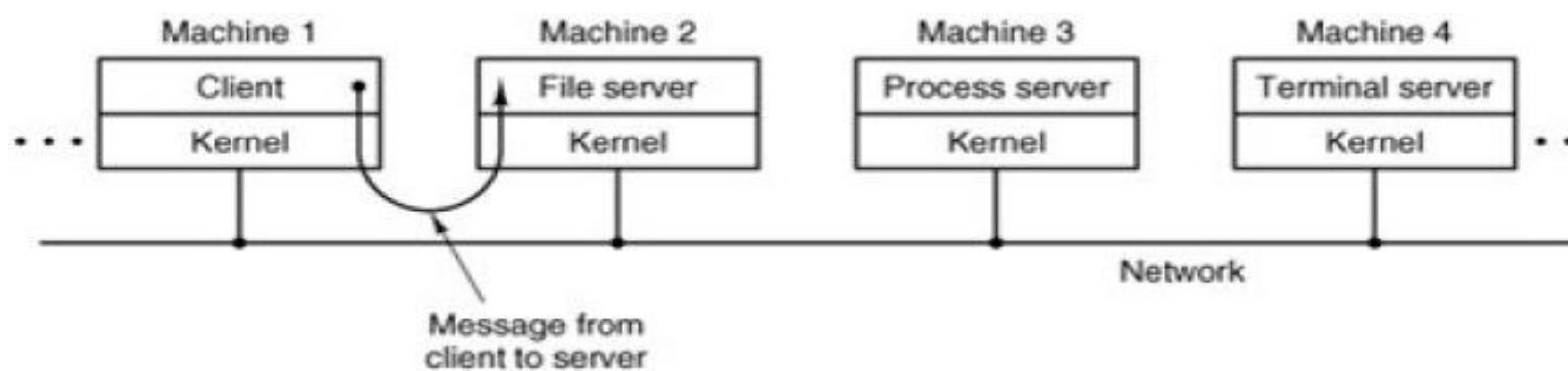
Advantages/Disadvantages of Virtual Machines

- The virtual-machine concept provides complete protection of system resources since each virtual machine is isolated from all other virtual machines. This isolation, however, permits no direct sharing of resources.
- A virtual-machine system is a perfect vehicle for operating systems research and development. System development is done on the virtual machine, instead of on a physical machine and so does not disrupt normal system operation.
- The virtual machine concept is difficult to implement due to the effort required to provide an *exact duplicate to the underlying machine*.



1.3.4. Client-Server Model:

- This concept is based on two classes of processes, the servers, each of which provides some service, and the clients, which use these services. This model is known as client-server model. Communication between clients and servers is often by message passing. To obtain a service, a client process constructs a message saying what it wants and sends it to the appropriate service. The service then does the work and sends back the answer. An obvious generalization of this idea is to have the clients and servers run on different computers, connected by a local or wide-area network.



System Design Goals

- User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast.
- System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient.

Mechanisms and Policies

- Mechanisms determine how to do something, policies decide what will be done.
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later.

System Implementation

- Traditionally written in assembly language, operating systems can now be written in higher-level languages.
- Code written in a high-level language:
 - can be written faster.
 - is more compact.
 - is easier to understand and debug.
- An operating system is far easier to *port* (*move to some other hardware*) if it is written in a high-level language.

System Generation (SYSGEN)

- Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site.
- SYSGEN program obtains information concerning the specific configuration of the hardware system.
- *Booting – starting a computer by loading the kernel.*
- *Bootstrap program – code stored in ROM that is able to locate the kernel, load it into memory, and start its execution.*

Applied Operating System

Chapter 2: Processes/Threads Management

Prepared By:
Amit K. Shrivastava
Asst. Professor
Nepal College Of Information Technology

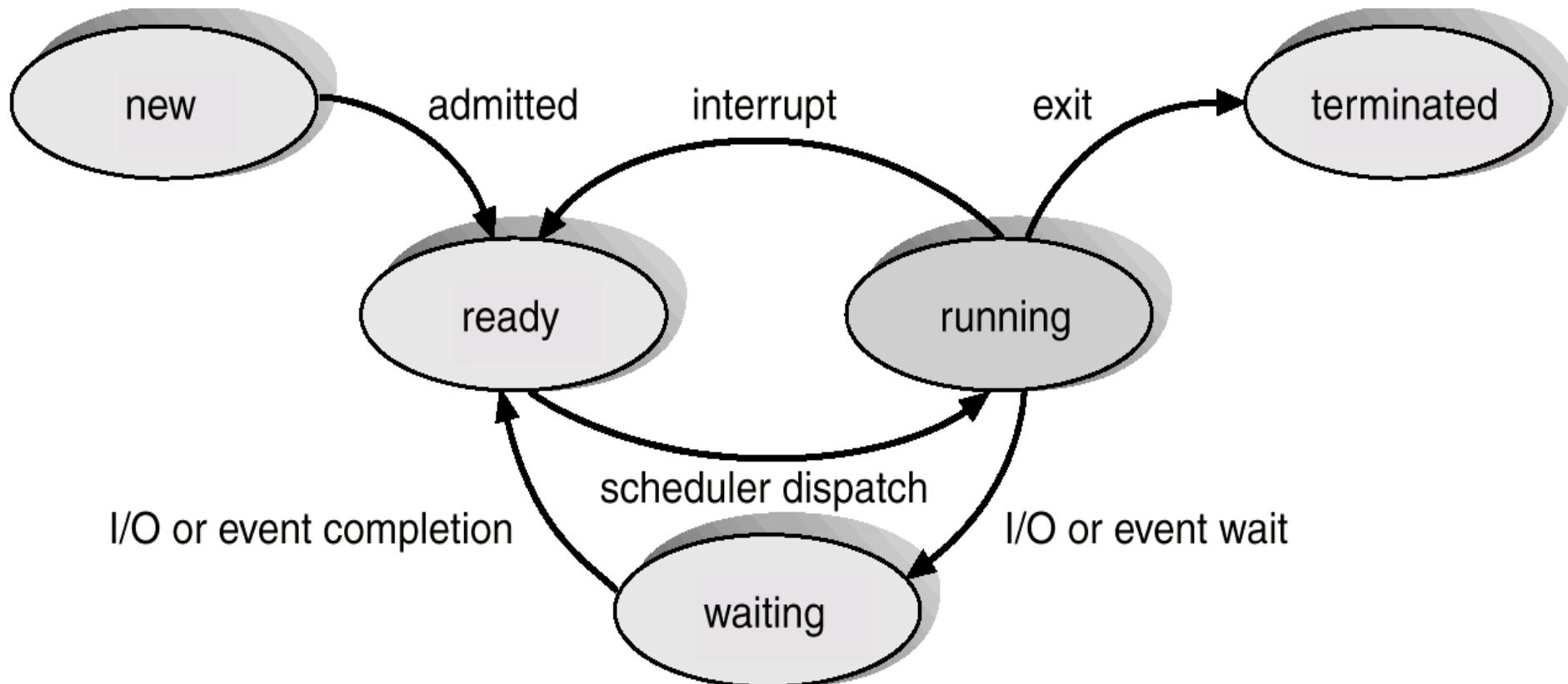
Process Concept

- An operating system executes a variety of programs:
 - ◆ Batch system – jobs
 - ◆ Time-shared systems – user programs or tasks
- The terms job and process are used almost interchangeably.
- Process – a program in execution; process execution must progress in sequential fashion.
- A process includes:
 - ◆ program counter- specifying next instruction to be executed.
 - ◆ stack- containing temporary data such as return address.
 - ◆ data section- containing global variables.

Process State

- As a process executes, it changes state
 - ◆ new: The process is being created.
 - ◆ running: Instructions are being executed.
 - ◆ waiting: The process is waiting for some event to occur.
 - ◆ ready: The process is waiting to be assigned to a process.
 - ◆ terminated: The process has finished execution.

Diagram of Process State

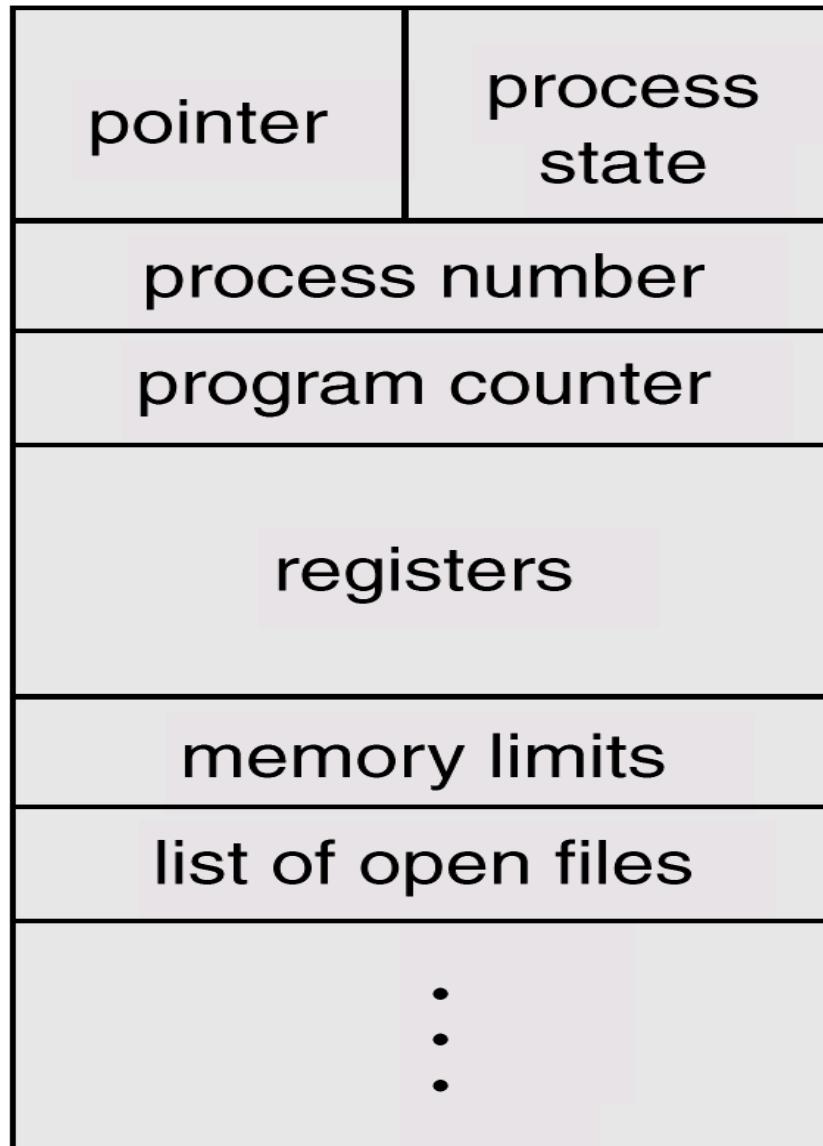


Process Control Block (PCB)

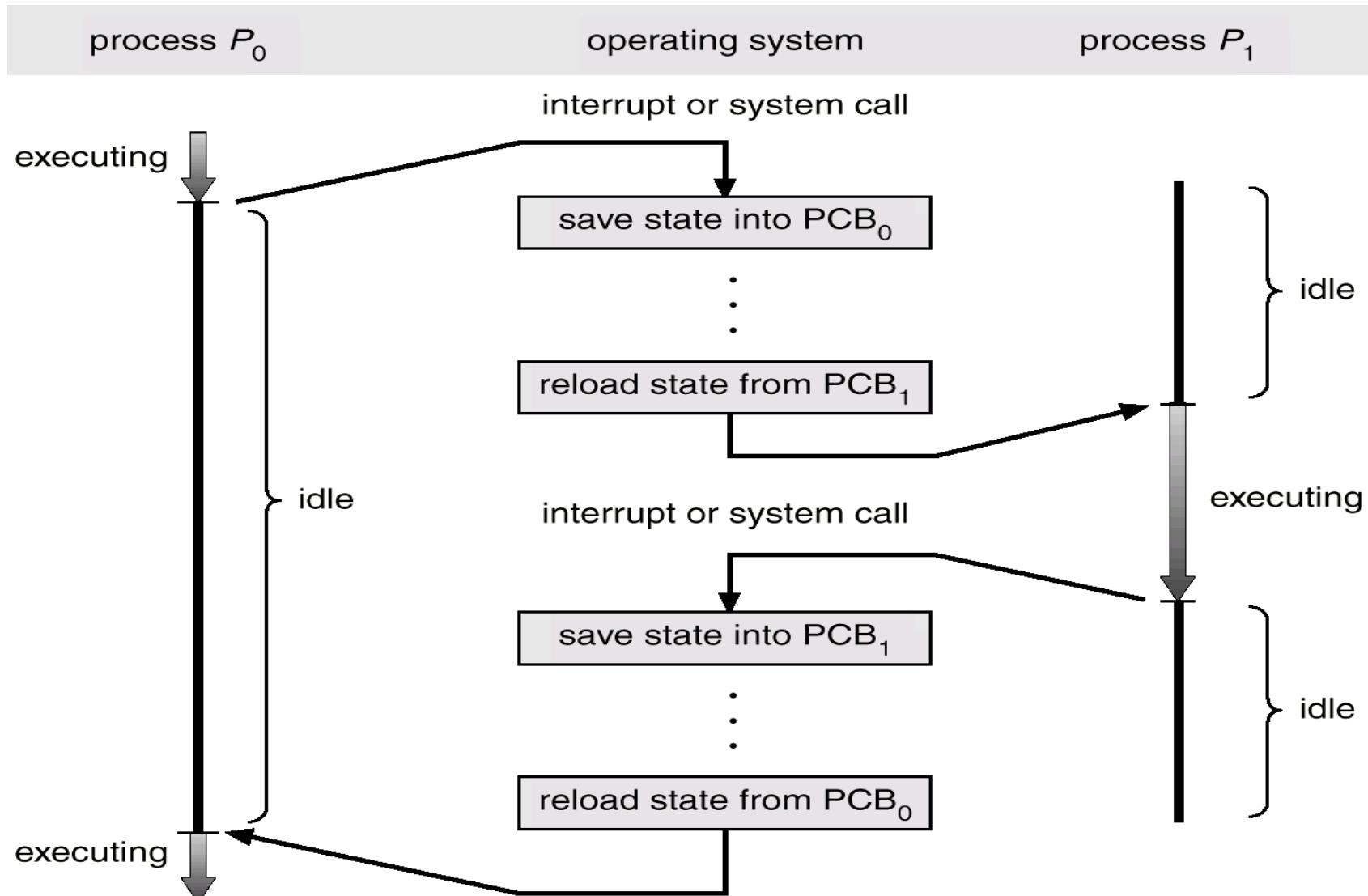
Information associated with each process.

- Process state- new, ready, ...
- Program counter- indicates the address of the next instruction to be executed for this program.
- CPU registers- includes accumulators, stack pointers, ...
- CPU scheduling information- includes process priority, pointers to scheduling queues.
- Memory-management information- includes the value of base and limit registers (protection) ...
- Accounting information- includes amount of CPU and real time used, account numbers, process numbers, ...
- I/O status information- includes list of I/O devices allocated to this process, a list of open files, ...

Process Control Block (PCB)



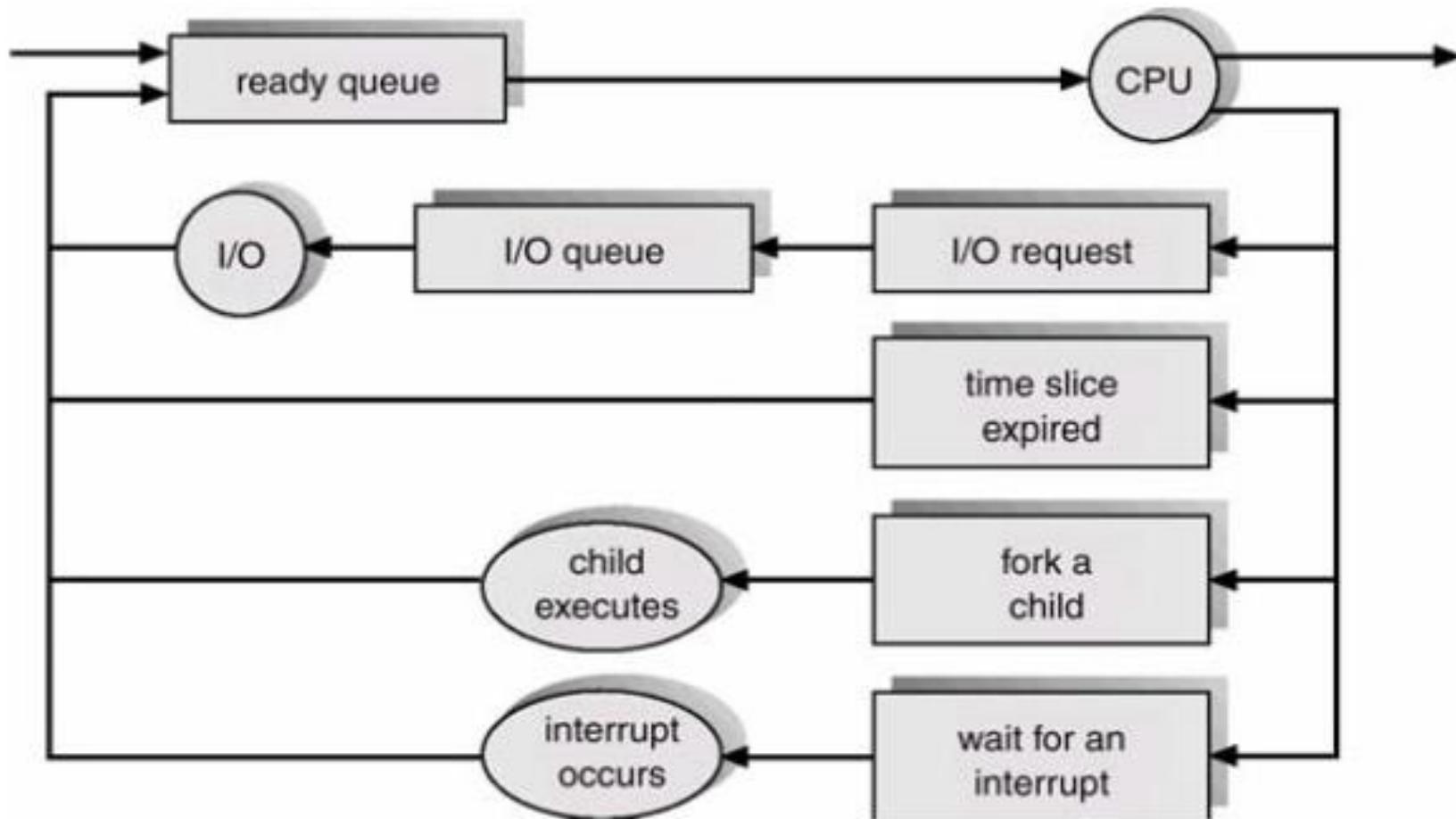
CPU Switch From Process to Process



Process Scheduling Queues

- Objective of multiprogramming is to have some process running at all time to maximize CPU utilization.
- Objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.
- For a uniprocessor 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.
- Job queue – set of all processes in the system.
- Ready queue – set of all processes residing in main memory, ready and waiting to execute. Ready queue is stored as linked list. A Ready Queue Header will contain pointers to the first and last PCBs in the list. Each PCB has a pointer field that points to the next process in the Ready Queue.
- Device queues – set of processes waiting for an I/O device. Each device has its own device queue.

Representation of Process Scheduling



Schedulers

- Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue (i.e, selects processes from pool (disk) and loads them into memory for execution).
- Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU (i.e, selects from among the processes that are ready to execute, and allocates the CPU to one of them) .
- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast).
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow).
- The long-term scheduler controls the degree of multiprogramming (the number of processes in memory).
- Medium-term scheduler – to remove processes from memory and reduce the degree of multiprogramming (the process is swapped out and swapped in by the medium-term scheduler).

Schedulers (Cont.)

- Processes can be described as either:
 - *I/O-bound process* – spends more time doing I/O than computations, many short CPU bursts.
 - *CPU-bound process* – spends more time doing computations; few very long CPU bursts.
- If all processes are I/O bound, the ready queue will almost always be empty and the short-scheduler will have little to do.
- If all processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused, and the system will be unbalanced.
- To get best performance the system should have a combination of CPU and I/O bound processes.

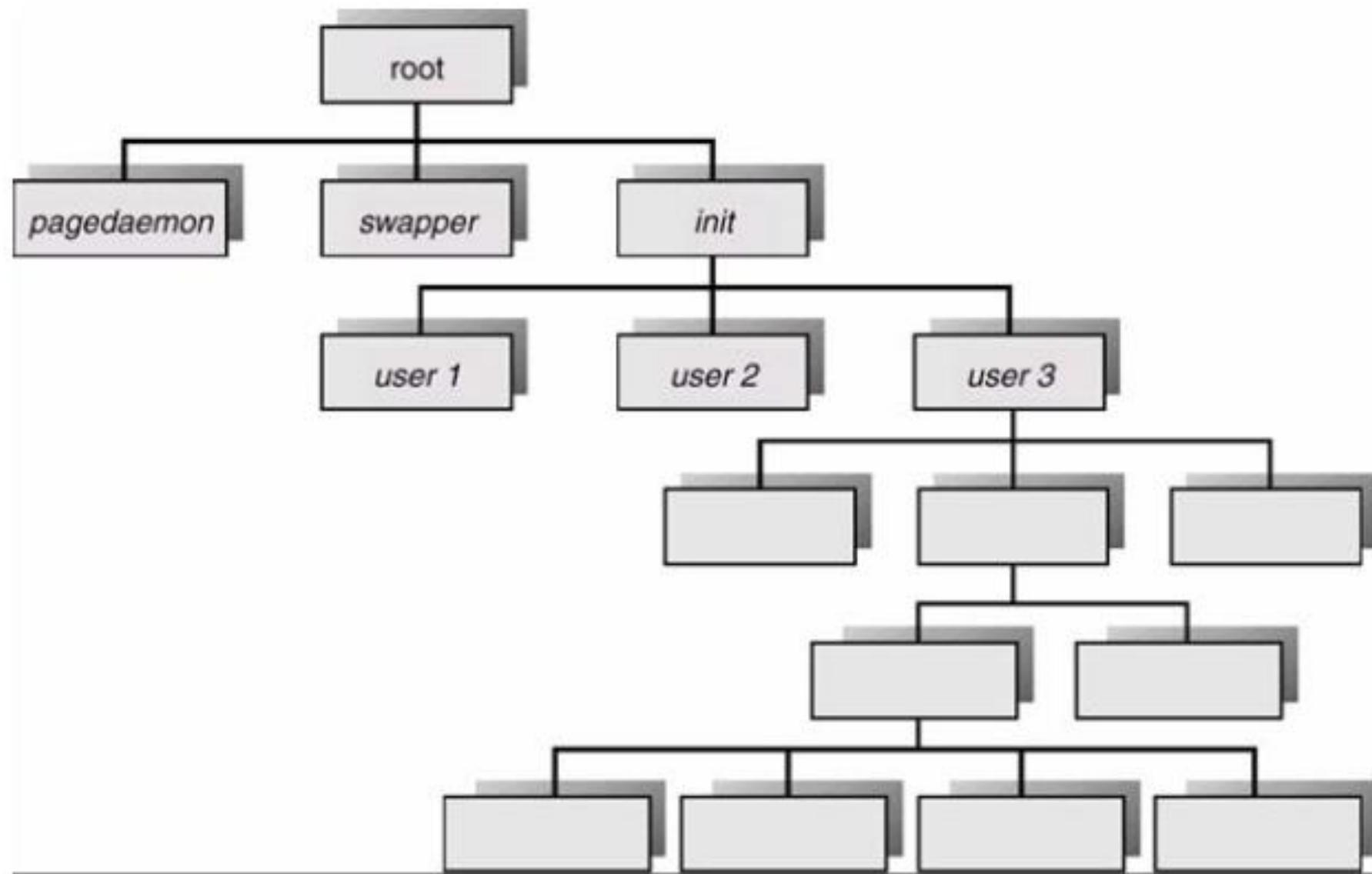
Context Switch

- Context Switch - When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process.
- Context-Switch Time is overhead; the system does no useful work while switching.
- Context-Switch Time depends on hardware support.
- Context-Switch Speed varies from machine to machine depending on memory speed, number of registers copied. The speed ranges from 1 to 1000 microsecond.

Operations in Process: Process Creation

- A process may create several new processes, via a create-process system call, during execution.
- Parent process creates children processes, which, in turn create other processes, forming a tree of processes.
- Resource sharing, such as CPU time, memory, files, I/O devices ...
 - Parent and children share all resources.
 - Children share subset of parent's resources.
 - Parent and child share no resources.
- When a process creates a new process, two possibilities exist in terms of execution:
 - Parent and children execute concurrently.
 - Parent waits until children terminate.
- There are also two possibilities in terms of the address space of the new process:
 - Child duplicate of parent.
 - Child has a program loaded into it.
- UNIX examples
 - **fork** system call creates new process
 - **execve** system call used after a **fork** to replace the process' memory space with a new program.

A Tree of Processes On A Typical UNIX System



Process Termination

- Process executes last statement and asks the operating system to delete it by using the **exit** system call.
 - Output data from child to parent via **wait** system call.
 - Process' resources are deallocated by operating system.
- Parent may terminate execution of children processes via **abort** system call for a variety of reasons, such as:
 - Child has exceeded allocated resources.
 - Task assigned to child is no longer required.
 - Parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

Cooperating Processes

- Independent process cannot affect or be affected by the execution of another process.
- Cooperating process can affect or be affected by the execution of another process

Advantages of process cooperation:

- Information sharing: Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to these types of resources.
- Computation speedup: If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Such a speedup can be achieved only if the computer has multiple processing elements (such as CPUSor I/O channels).

- Modularity: We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads,
- Convenience: Even an individual user may have many tasks on which to work at one time. For instance, a user may be editing, printing, and compiling in parallel.

IPC(Inter-Process Communication):

- Processes frequently need to communicate with other processes i.e they need to exchange data and information. Thus there is need a need for communication between processes, preferably in a well-structured way not using interrupts. This communication mechanism between processes are known as Inter-Process Communication.
- There are two fundamental models of interprocess communication: shared memory and message passing.
- In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.
- In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.

IPC(Inter-Process Communication) contd...:

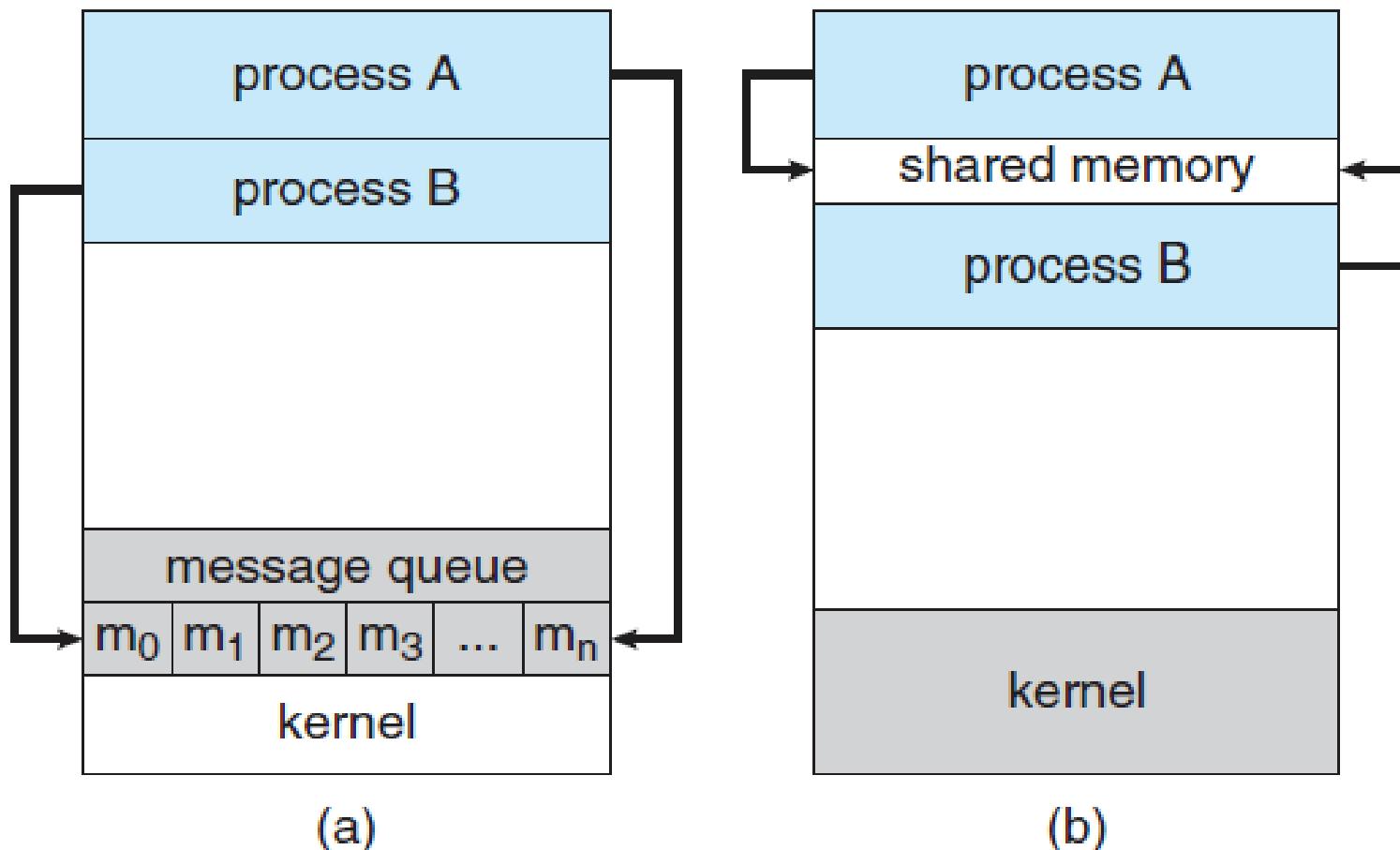


Figure 3.12 Communications models. (a) Message passing. (b) Shared memory.

Threads

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

Advantages of Threads

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

1. Responsiveness: Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. For instance, a multithreaded web browser could still allow user interaction in one thread while an image is being loaded in another thread.

2. Resource sharing: By default, threads share the memory and the resources of the process to which they belong. The benefit of code sharing is that it allows an application to have several different threads of activity all within the same address space.

Advantages of Threads(contd..)

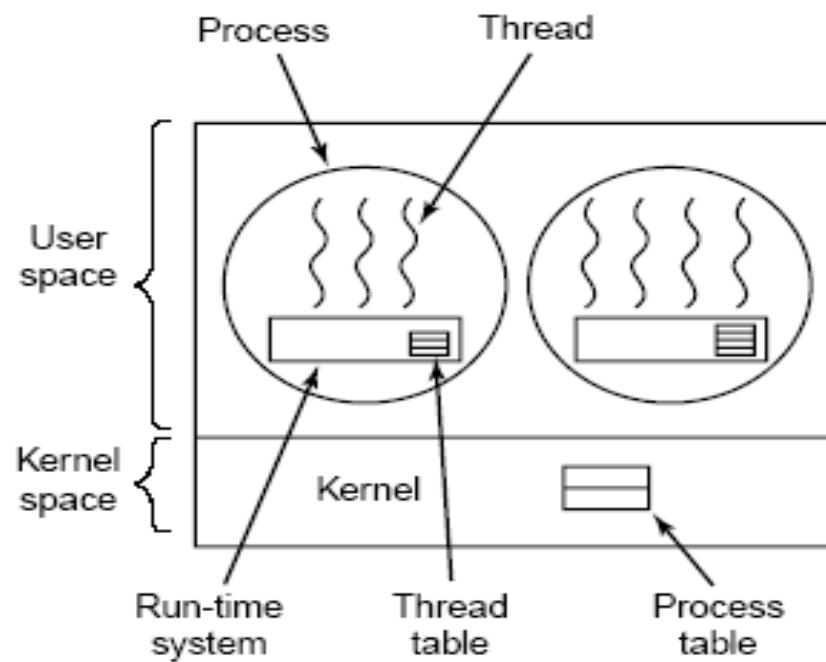
- 3. Economy:** Allocating memory and resources for process creation is costly. Alternatively, because threads share resources of the process to which they belong, it is more economical to create and context switch threads.
- 4. Utilization of multiprocessor architectures:** The benefits of multithreading can be greatly increased in a multiprocessor architecture, where each thread may be running in parallel on a different processor. A single-threaded process can only run on one CPU, no matter how many are available. Multithreading on a multi-CPU machine increases concurrency.

User Space Threads

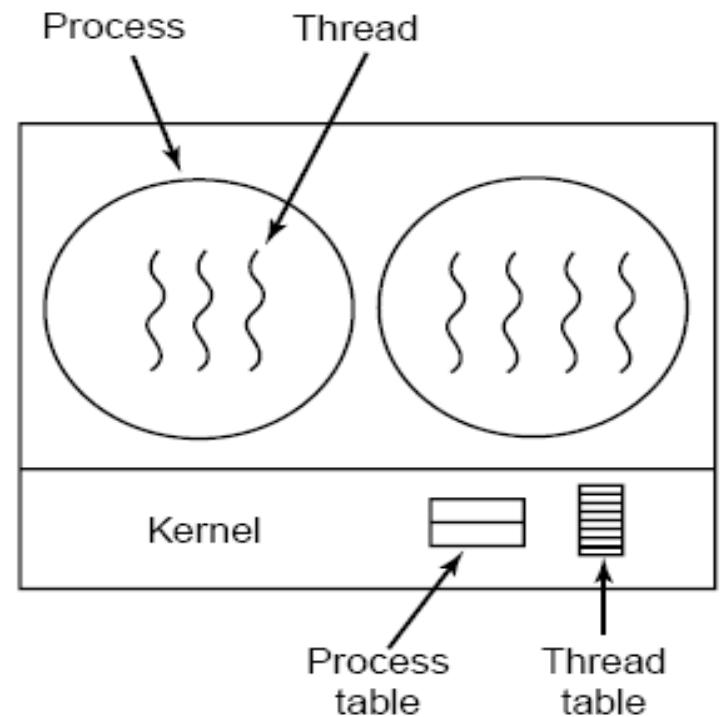
- User threads are supported above the kernel and are implemented by a thread library at the user level.
- The library provides support for thread creation, scheduling, and management with no support from the kernel.
- Because the kernel is unaware of user-level threads, all thread creation and scheduling are done in user space without the need for kernel intervention. Therefore, user-level threads are generally fast to create and manage.
- Its drawback is that if the kernel is single-threaded, then any user-level thread performing a blocking system call will cause the entire process to block, even if other threads are available to run within the application.

Kernel Space Threads

- › Kernel threads are supported directly by the operating system. The kernel performs thread creation, scheduling, and management in kernel space.
- › Because thread management is done by the operating system, kernel threads are generally slower to create and manage than are user threads.
- › However, since the kernel is managing the threads, if a thread performs a blocking system call, the kernel can schedule another thread in the application for execution.
- › Also, in a multiprocessor environment, the kernel can schedule threads on different processors.



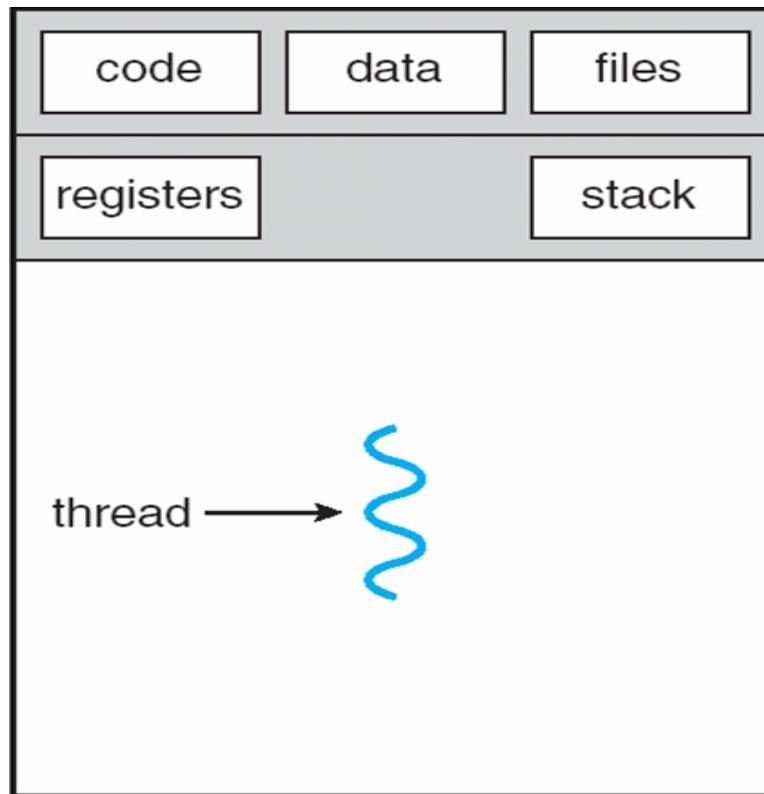
User space threads



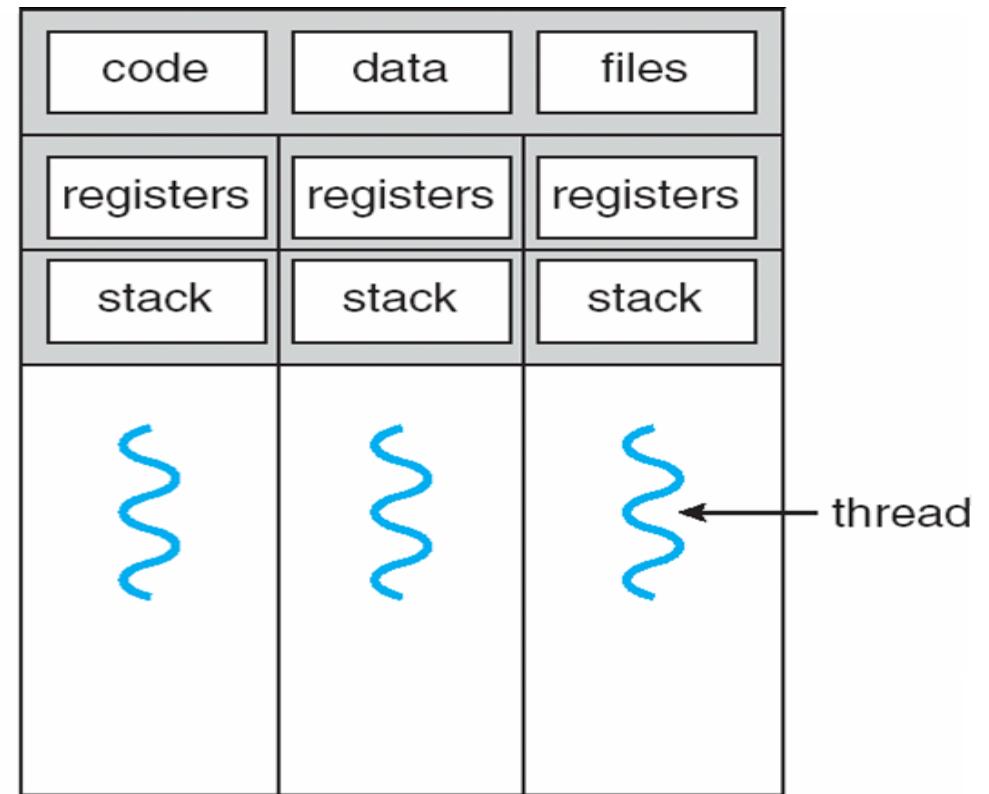
Kernel space threads

Multithreading

- Multithreading is the ability of an operating System to support multiple threads of execution within a single process.
 - Multiple threads run in the same address space, share the same memory areas
 - The creation of a thread only creates a new thread control structure, not a separate process image



single-threaded process



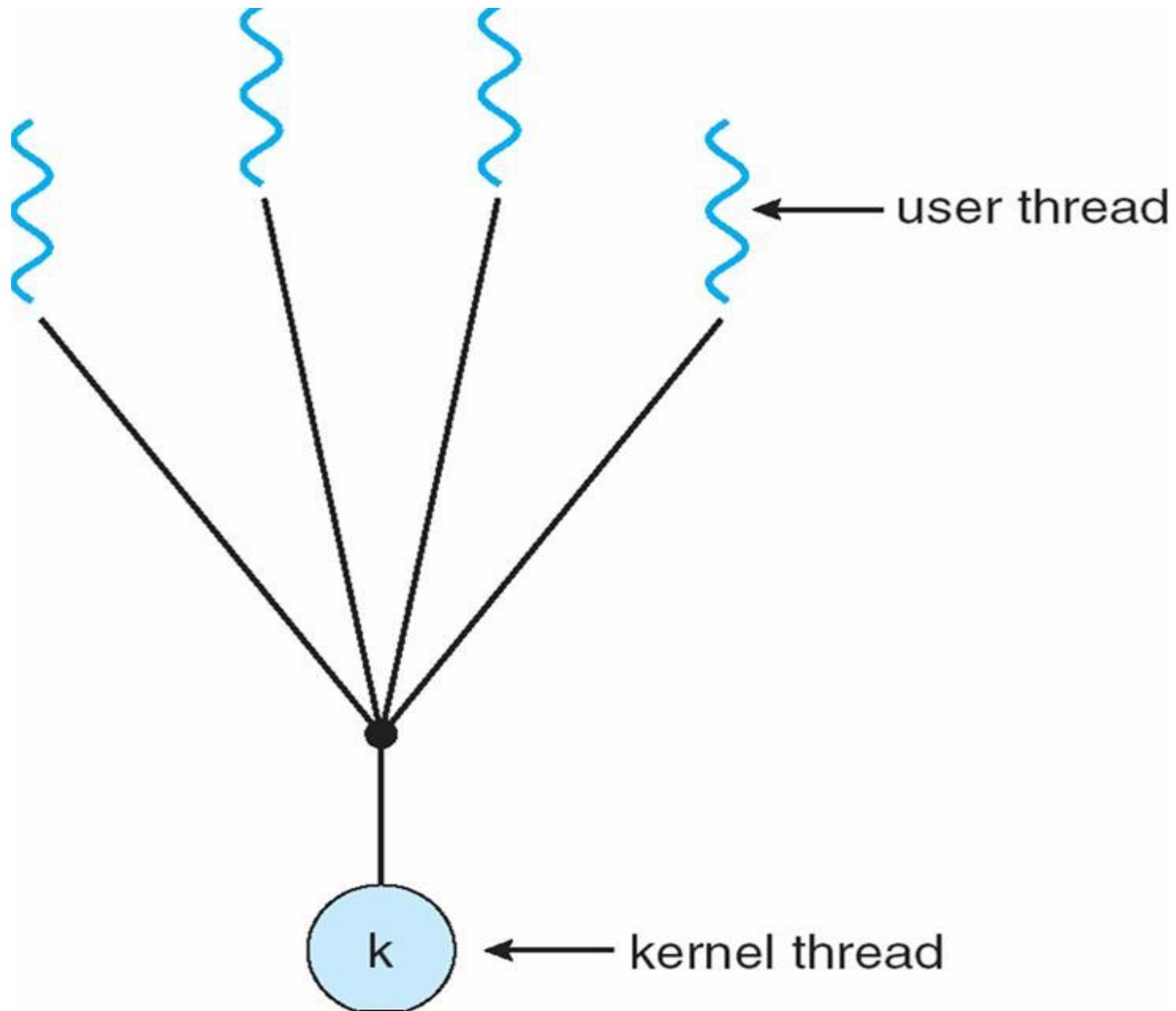
multithreaded process

Multithreading Models

Many systems provide support for both user and kernel threads, resulting in different multithreading models.

Many-to-One Model:

- All user-level threads of one process mapped to a single kernel-level thread
- Thread management in user space
 - Efficient
 - Application can run its own scheduler implementation
- One thread can access the kernel at a time
 - Limited concurrency, limited parallelism
- Examples
 - “Green threads” (e.g. Solaris)
 - Gnu Portable Threads



Many-to-one model

Multithreading Models(contd...)

One-to-One Model:

- The one-to-one model maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.
- It also allows multiple threads to run in parallel on multiprocessors.
- The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread. Because the overhead of most implementations of this model restrict the number of threads supported by the system
- Examples:
 - Linux, along with the family of windows operating systems.

Multithreading Models(contd...)

Many-to-Many Model:

- The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine.
- The many-to-many model suffers from neither of these shortcomings of above two model. Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.
- Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.

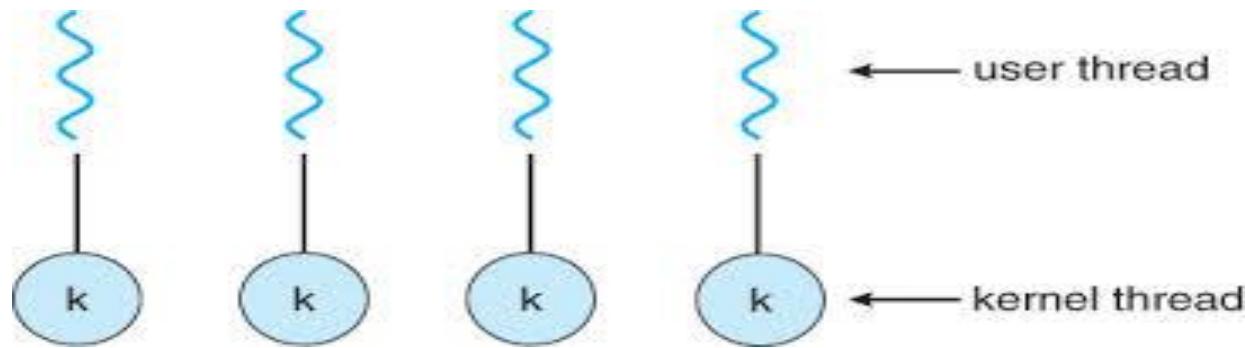


Fig: One-to One model

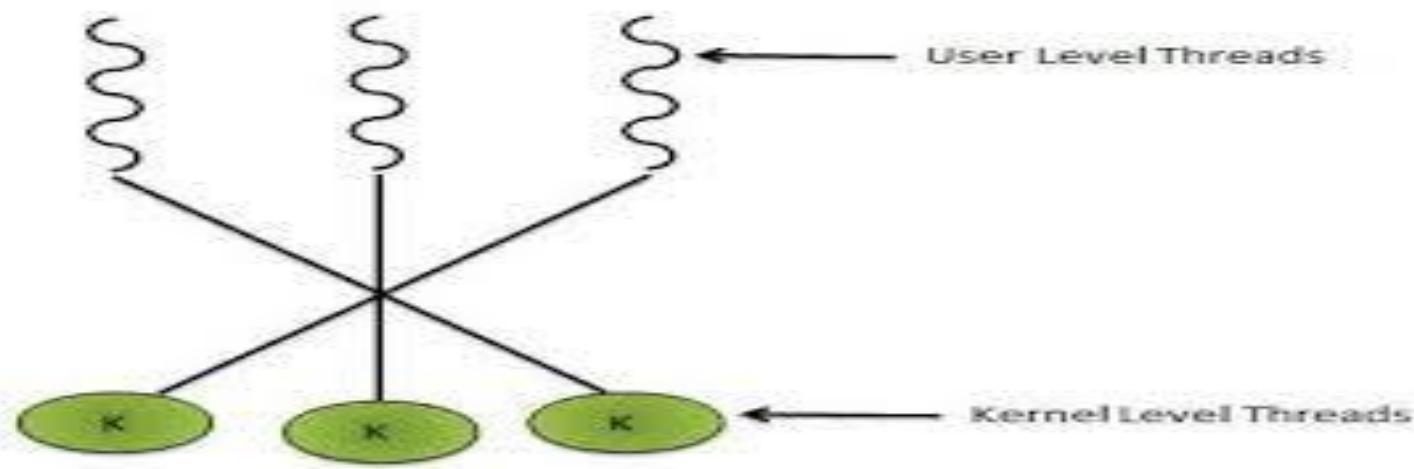


Fig: Many-to Many
model

Differences Between Processes and Threads:

- A process is a program in execution, whereas a thread is a path of execution within a process.
- Processes are generally used to execute large, ‘heavyweight’ jobs such as running different applications, while threads are used to carry out much smaller or ‘lightweight’ jobs such as auto saving a document in a program, downloading files, etc. Whenever we double-click an executable file such as Paint, for instance, the CPU starts a process and the process starts its primary thread.
- Each process runs in a separate address space in the CPU. But threads, on the other hand, may share address space with other threads within the same process. This sharing of space facilitates communication between them. Therefore, Switching between threads is much simpler and faster than switching between processes.
- Threads can directly communicate with other threads of its process; processes must use interprocess communication to communicate with sibling processes.
- Threads also have a great degree of control over other threads of the same process. Processes only have control over child processes.

Processor Scheduling:

- When a computer is multiprogrammed, it frequently has multiple processes competing for the CPU at the same time. When more than one process is in the ready state and there is only one CPU available, a choice has to be made which process to run next. The part of the operating system that makes the choice is called the scheduler, and the algorithm it uses is called the scheduling algorithm and the mechanism is called scheduling.

Scheduling Criteria :

The criteria include the following:

- CPU utilization: The CPU should keep as busy as possible. CPU utilization may 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 used system).
- Throughput: If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes completed per time unit, called throughput.
- 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. IT is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
- Waiting Time: Waiting time is the sum of the periods spent waiting in the ready queue.
- Response Time: Response time is the time it takes to start responding.

Preemptive Scheduling and Non Preemptive Scheduling

- **Non preemptive:** In this case, once a process is in the running state, It continues to execute until (a) it terminates or (b) blocks itself to wait for I/O or to request some operating system service.
- **Preemptive:** The currently running process may be interrupted and moved to the ready state by the operating system. The decision to preempt may be performed when a new process arrives or periodically based on a clock interrupt OR when a new process switches from the waiting state to the ready state(for example, at completion of I/O)

Scheduling Technique

- **First-Come-First-Served(First-In-First-Out) Scheduling:** 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.

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



First-Come, First-Served (FCFS) Scheduling

<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1, P_2, P_3
The Gantt Chart for the schedule is:



- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$



FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order

$$P_2, P_3, P_1.$$

- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case.
- *Convoy effect* short process behind long process

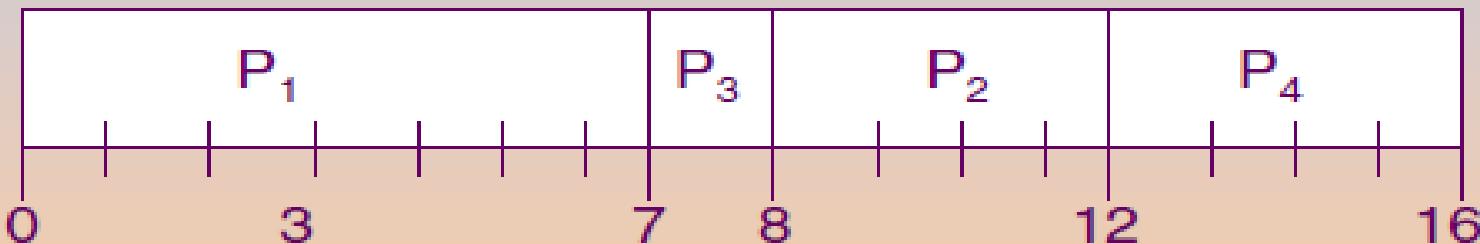
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.
- Two schemes:
 - ◆ nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst.
 - ◆ preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is known as the Shortest-Remaining-Time-First (SRTF).
- SJF is optimal – gives minimum average waiting time for a given set of processes.

Example of Non-Preemptive SJF

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

SJF (non-preemptive)

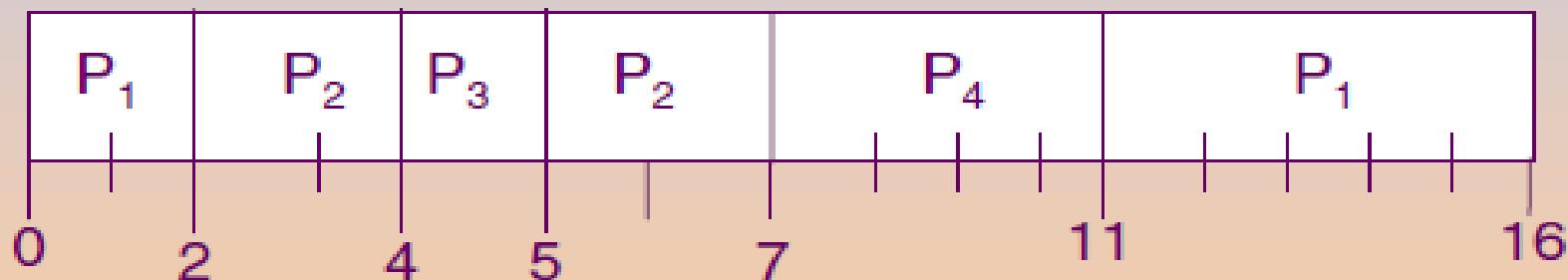


Average waiting time = $(0 + 6 + 3 + 7)/4 - 4$

Example of Preemptive SJF

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)



- Average waiting time = $(9 + 1 + 0 + 2)/4 - 3$

Priority Scheduling

➤ A priority number (integer) is associated with each process
The CPU is allocated to the process with the highest priority (smallest integer \equiv highest priority).

◆ Preemptive

◆ nonpreemptive

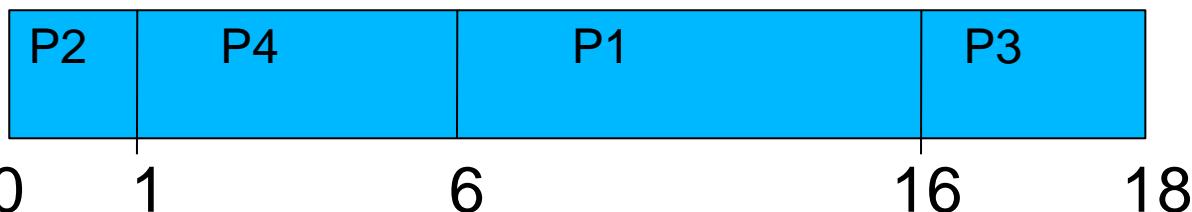
➤ SJF is a priority scheduling where priority is the predicted next CPU burst time.

Problem \equiv Starvation – low priority processes may never execute.

Solution \equiv Aging – as time progresses increase the priority of the process.

Consider following set of process arrived at time 0

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
P1	10	3
P2	1	1
P3	2	4
P4	5	2



Average Waiting Time=8.2 ms

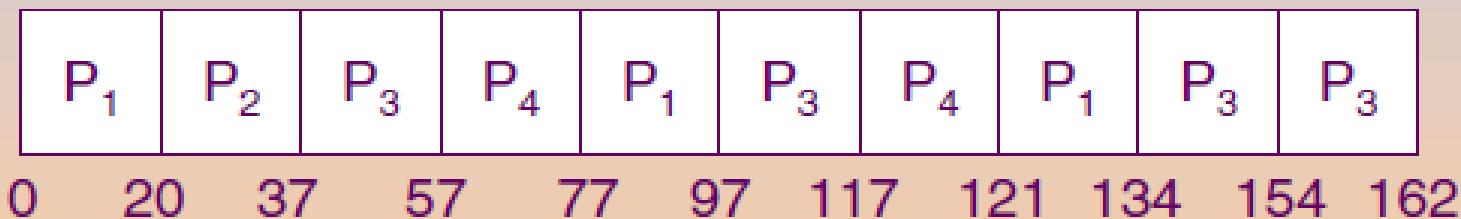
Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are n processes in the ready queue and the time quantum is q , then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units.
- Performance
 - ◆ q large \Rightarrow FIFO
 - ◆ q small \Rightarrow q must be large with respect to context switch, otherwise overhead is too high.

Example of RR with Time Quantum = 20

<u>Process</u>	<u>Burst Time</u>
P_1	53
P_2	17
P_3	68
P_4	24

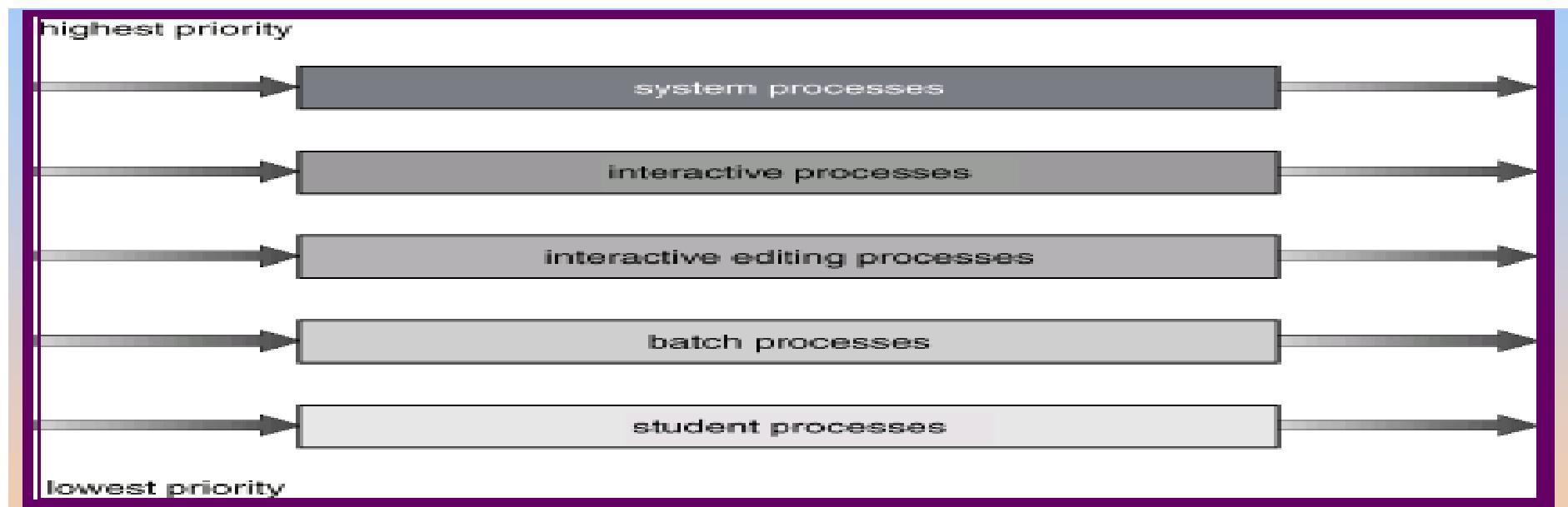
- The Gantt chart is:



- Typically, higher average turnaround than SJF, but better response.

Multilevel Queue Scheduling:

- Ready queue is partitioned into separate queues:
foreground (interactive), background (batch)
- Each queue has its own scheduling algorithm,
foreground – RR
background – FCFS
- Scheduling must be done between the queues.
 - ◆ Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
 - ◆ Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR.20% to background in FCFS



Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way.
- Multilevel-feedback-queue scheduler defined by the following parameters:
 - ◆ number of queues
 - ◆ scheduling algorithms for each queue
 - ◆ method used to determine when to upgrade a process
 - ◆ method used to determine when to demote a process
 - ◆ method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

- Three queues:

- Q_0 – time quantum 8 milliseconds
- Q_1 – time quantum 16 milliseconds
- Q_2 – FCFS

- Scheduling

- A new job enters queue Q_0 , which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1 .
- At Q_1 , job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2 .



Highest-Response-Ratio-Next(HRN) Scheduling

➤ HRN is a nonpreemptive scheduling discipline in which the priority of each job is a function not only of the job's service time but also of the amount of the time the job has been waiting for service. Once a job gets the CPU, it runs to completion. Dynamic priorities in HRN are calculated according to the formula

$$\text{priority} = \frac{\text{time waiting} + \text{Service}}{\text{Service Time}}$$

➤ Because the service time appears in the denominator, shorter job will get preference. But because time waiting appears in the numerator, longer jobs that have been waiting will also be given favorable sum.

Process Synchronization

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills **all** the buffers. We can do so by having an integer **count** that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer and Consumer Problem

- To illustrate the concept of cooperating processes, let us consider the producer-consumer problem, which is a common paradigm for cooperating processes.
- A producer process produces information that is consumed by a consumer process. For example, a print program produces characters that are consumed by the printer driver. A compiler may produce assembly code, which is consumed by an assembler.
- To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.
- A producer can produce one item while the consumer is consuming another item.
- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced. In this situation, the consumer must wait until an item is produced.

- Unbounded-buffer places no practical limit on the size of the buffer-
The consumer may have to wait for new items, but the producer can always produce new items.
- Bounded-buffer assumes that there is a fixed buffer size.In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.
- The buffer may either be provided by the operating system through the use of an interprocess-communication (IPC) facility or by explicitly coded by the application programmer with the use of shared memory.
- Trouble arises when the producer wants to put a new item in the buffer, but it is already full. The solution is for the producer to go to sleep, to be awakened when the consumer has removed one or more items. Similarly, if the consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep until the producer puts something in the buffer and wakes it up.

```

#define N 100
int count = 0;

void producer(void)
{
    int item;
    while (TRUE) {
        item = produce_item();
        if (count == N) sleep();
        insert_item(item);
        count = count + 1;
        if (count == 1) wakeup(consumer);
    }
}

void consumer(void)
{
    int item;
    while (TRUE) {
        if (count == 0) sleep();
        item = remove_item();
        count = count - 1;
        if (count == N - 1) wakeup(producer);
        consume_item(item);
    }
}

```

/* number of slots in the buffer */
 /* number of items in the buffer */

/* repeat forever */
 /* generate next item */
 /* if buffer is full, go to sleep */
 /* put item in buffer */
 /* increment count of items in buffer */
 /* was buffer empty? */

/* repeat forever */
 /* if buffer is empty, go to sleep */
 /* take item out of buffer */
 /* decrement count of items in buffer */
 /* was buffer full? */
 /* print item */

Figure 2-27. The producer-consumer problem with a fatal race condition.

Race Condition:

- In some operating systems, processes that are working together may share some common storage that each one can read and write.
- Situations like this, where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called **race conditions**.

Critical Regions:

- Sometimes a process has to access shared memory or files, or do other critical things that can lead to races. That part of the program where the shared memory is accessed is called the **critical region** or **critical section**.
- If we could arrange matters such that no two processes were ever in their critical regions at the same time, we could avoid race conditions.

Critical Regions(contd..):

- Although this requirement avoids race conditions, this is not sufficient for having parallel processes cooperate correctly and efficiently using shared data. We need four conditions to hold to have a good solution:
 1. No two processes may be simultaneously inside their critical regions.
 2. No assumptions may be made about speeds or number of CPUs.
 3. No process running outside its critical region may block other processes.
 4. No process should have to wait forever to enter its critical region.
- The behavior that we want is shown in Fig. of next slide.

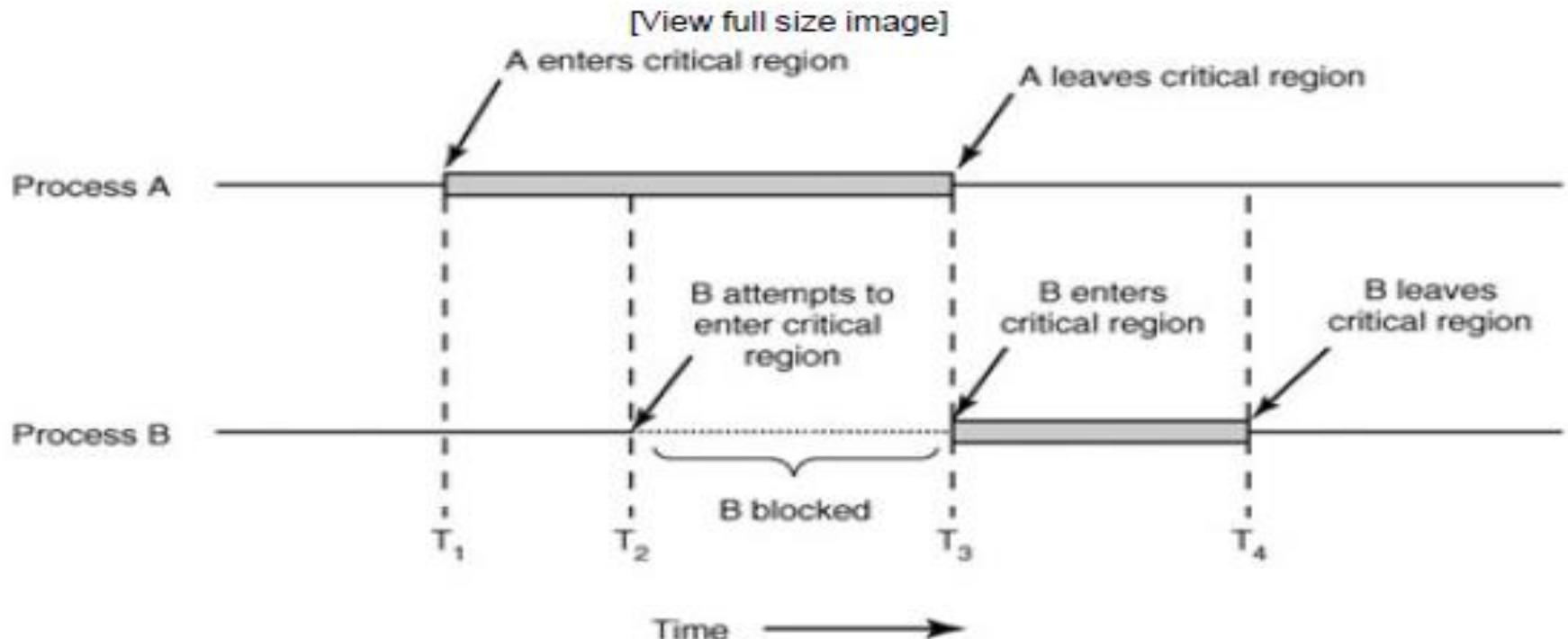


Fig: Mutual Exclusion using Critical Regions

Here process A enters its critical region at time T_1 . A little later, at time T_2 process B attempts to enter its critical region but fails because another process is already in its critical region and we allow only one at a time. Consequently, B is temporarily suspended until time T_3 when A leaves its critical region, allowing B to enter immediately. Eventually B leaves (at T_4) and we are back to the original situation with no processes in their critical regions.

The Critical-Section Problem

- n processes all competing to use some shared data .
- Each process has a code segment, called critical section, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Solution to Critical-Section Problem

1. **Mutual Exclusion.** If process P_i 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 there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
3. **Bounded Waiting.** A bound must exist 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.
 - Assume that each process executes at a nonzero speed
 - No assumption concerning relative speed of the n processes.

Two-Tasks Solutions

Peterson's Algorithm:

Two Process Solution. It is a much simpler way to achieve mutual exclusion.

```
#define FALSE 0
#define TRUE 1
#define N    2           /* number of processes */

int turn;                  /* whose turn is it? */
int interested[N];         /* all values initially 0 (FALSE) */
void enter_region(int process) /* process is 0 or 1 */
{
    int other;              /* number of the other process */
    other = 1 - process;    /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process;          /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */;
}

void leave_region(int process) /* process: who is leaving */
{
    interested[process] = FALSE; /* indicate departure from critical region */
}
```

Peterson's Algorithm Explanation

- Before using the shared variables (i.e., before entering its critical region), each process calls `enter_region` with its own process number, 0 or 1, as the parameter. This call will cause it to wait, if need be, until it is safe to enter. After it has finished with the shared variables, the process calls `leave_region` to indicate that it is done and to allow the other process to enter, if it so desires.
 - Let us see how this solution works. Initially, neither process is in its critical region. Now process 0 calls `enter_region`. It indicates its interest by setting its array element and sets `turn` to 0. Since process 1 is not interested, `enter_region` returns immediately. If process 1 now calls `enter_region`, it will hang there until `interested[0]` goes to FALSE, an event that only happens when process 0 calls `leave_region` to exit the critical region.
 - Now consider the case that both processes call `enter_region` almost simultaneously. Both will store their process number in `turn`. Whichever store is done last is the one that counts; the first one is lost. Suppose that process 1 stores last, so `turn` is 1. When both processes come to the while statement, process 0 executes it zero times and enters its critical region. Process 1 loops and does not enter its critical region.

Synchronization Hardware

Many systems provide hardware support for critical section code

TSL(Test and Set Lock):

➤ TSL is a hardware solution to the mutual exclusion problem. The key to success here is to have a single hardware instruction that reads a variable, stores its value in a save area, and sets the variable to a certain value. This instruction, often called test and set, once initiated will complete all of these functions without interruption. The instruction is like

TSL REGISTER, LOCK

➤ (Test and Set Lock) that works as follows. It reads the contents of the memory word lock into register RX and then stores a nonzero value at the memory address lock. The operations of reading the word and storing into it are guaranteed to be indivisible—no other processor can access the memory word until the instruction is finished. The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.

➤ To use the TSL instruction, we will use a shared variable, lock, to coordinate access to shared memory. When lock is 0, any process may set it to 1 using the TSL instruction and then read or write the shared memory. When it is done, the process sets lock back to 0 using an ordinary move instruction.

TSL(contd..)

```
do {  
    acquire lock  
    Critical section  
    Release lock  
    remainder section  
} while(TRUE);  
fig: Solution to the critical-section problem using locks
```

enter_region:

```
TSL REGiSTER,LOCK      / copy lock to register and set lock to 1  
CMP REGiSTER,#0         / was lock zero?  
JNE enter_region        / if it was nonzero, lock was set, so loop  
RET                     / return to caller; critical region entered
```

leave_region:

```
MOVE LOCK,#0            / store a 0 in lock  
RET                     / return to caller
```

Figure : Entering and leaving a critical region using the TSL instruction.

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore S – integer variable
- Two standard operations modify S : `wait()` and `signal()`
 - Originally called `P()` and `V()`
- Less complicated
- Can only be accessed via two indivisible (atomic) operations
 - `wait (S) {`
 `while S <= 0`
 `; // no-op`
 `S--;`
 `}`
 - `signal (S) {`
 `S++;`
 `}`

Semaphore as General Synchronization Tool

- | Counting semaphore – integer value can range over an unrestricted domain
- | Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
 - Also known as mutex locks
- | Can implement a counting semaphore S as a binary semaphore
- | Provides mutual exclusion
 - Semaphore S ; // initialized to 1
 - `wait (S);`
 Critical Section
 `signal (S);`

Producer Consumer Problem Using Semaphore

Producer Process

The code that defines the producer process is given below:

```
do {  
    .  
    . PRODUCE ITEM  
    .  
    wait(empty);  
    wait(mutex);  
    .  
    . PUT ITEM IN BUFFER  
    .  
    signal(mutex);  
    signal(full);  
}  
} while(1);
```

- In the above code, mutex, empty and full are semaphores. Here mutex is initialized to 1, empty is initialized to n (maximum size of the buffer) and full is initialized to 0.
- The mutex semaphore ensures mutual exclusion. The empty and full semaphores count the number of empty And full spaces in the buffer.
- After the item is produced, wait operation is carried out on empty. This indicates that the empty space in the buffer has decreased by 1. Then wait operation is carried out on mutex so that consumer process cannot interfere.
- After the item is put in the buffer, signal operation is carried out on mutex and full. The former indicates that consumer process can now act and the latter shows that the buffer is full by 1.

Consumer Process

The code that defines the consumer process is given below:

```
do {  
    wait(full);  
    wait(mutex);  
    . . .  
    . REMOVE ITEM FROM BUFFER  
    .  
    signal(mutex);  
    signal(empty);  
    . . .  
    . CONSUME ITEM  
    .  
} while(1);
```

- The wait operation is carried out on full. This indicates that items in the buffer have decreased by 1. Then wait operation is carried out on mutex so that producer process cannot interfere.
- Then the item is removed from buffer. After that, signal operation is carried out on mutex and empty. The former indicates that consumer process can now act and the latter shows that the empty space in the buffer has increased by 1.

Monitors

- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.
- The representation of a monitor type consists of declarations of variables whose values define the state of an instance of the type, as well as the bodies of procedures or functions that implement operations on the type.

```
monitor monitor-name
```

```
{
```

```
shared variable declarations
```

```
procedure body P1 (...) {
```

```
...  
}
```

```
procedure body P2 (...) {
```

```
...  
}
```

```
procedure body Pn (...) {
```

```
...  
}
```

```
{
```

```
initialization code
```

```
}
```

```
}
```

Monitors(contd..)

- To allow a process to wait within the monitor, a **condition** variable must be declared, as

condition x, y;

- Condition variable can only be used with the operations **wait** and **signal**.

The operation

x.wait();

means that the process invoking this operation is suspended until another process invokes

x.signal();

The **x.signal** operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect.

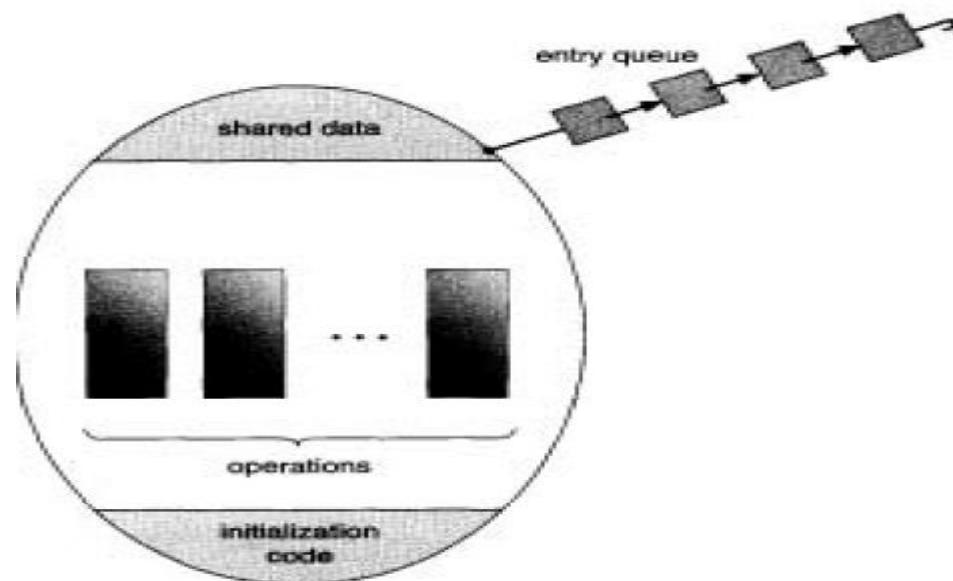


Figure 7.20 Schematic view of a monitor.

Classical Synchronization

There are number of different synchronization problem. Some of them are:

The Dining-Philosophers Problem:

Statement: Consider five philosophers who spend their lives thinking and eating. The philosophers share a common circular table surrounded by five chairs, each belonging to one philosopher. In the centre of the table there is a bowl of rice, and the table is laid with five single chopsticks. When a philosopher gets hungry she tries to pick up her left and right chopstick, one at a time, in either order. If successful in acquiring two chopsticks, she eats for a while, then puts down the chopsticks and continues to think. The key question is: Can you write a program for each philosopher that does what it is supposed to do and never gets stuck?



fig:The situation of the dining philosophers

Solution to the Dining-Philosophers Problem:

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore; she releases her chopsticks by executing the signal() operation on the appropriate semaphores. Thus , the shared data are

semaphore chopstick [5];

The structure of Philosopher i is shown below:

```
do {  
    wait ( chopstick[i] );  
    wait ( chopStick[ (i + 1) % 5] );  
    . . .  
    // eat  
    . . .  
    signal ( chopstick[i] );  
    signal (chopstick[ (i + 1) % 5] );  
    . . .  
    // think  
    . . .  
} while (true);
```

fig: the structure of philosopher i.

But if this solution simply applied deadlock will occur.

Solution to the Dining-Philosophers Problem(contd...):

Several possible remedies to the deadlock problem are listed. Here is a solution to the dining-philosophers problem that ensures freedom from deadlocks.

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this she must pick them up in a critical section).
- Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

Readers Writer Problems:

- The dining philosophers problem is useful for modeling processes that are competing for exclusive access to a limited number of resources, such as I/O devices. Another famous problem is the readers and writers problem which models access to a database (Courtois et al., 1971). Imagine, for example, an airline reservation system, with many competing processes wishing to read and write it. It is acceptable to have multiple processes reading the database at the same time, but if one process is updating (writing) the database, no other process may have access to the database, not even a reader. The question is how do you program the readers and the writers? One solution is shown below.

```
typedef int semaphore; /* use your imagination */
semaphore mutex = 1; /* controls access to 'rc' */
semaphore db = 1; /* controls access to the database */
int rc = 0; /* # of processes reading or wanting to */
```

Readers Writer problems(contd..)

```
void reader(void)
{
    while (TRUE) { /* repeat forever */
        down(&mutex); /* get exclusive access to 'rc' */
        rc = rc + 1; /* one reader more now */
        if (rc == 1) down(&db); /* if this is the first reader ... */
        up(&mutex); /* release exclusive access to 'rc' */
        read_data_base(); /* access the data */
        down(&mutex); /* get exclusive access to 'rc' */
        rc = rc - 1; /* one reader fewer now */
        if (rc == 0) up(&db); /* if this is the last reader ... */
        up(&mutex); /* release exclusive access to 'rc' */
        use_data_read(); /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) { /* repeat forever */
        think_up_data(); /* noncritical region */
        down(&db); /* get exclusive access */
        write_data_base(); /* update the data */
        up(&db); /* release exclusive access */
    }
}
```

fig: A solution to the readers writer problem

Readers Writer problems(contd..)

- In this solution, the first reader to get access to the data base does a down on the semaphore *db* . Subsequent readers merely have to increment a counter, *rc* . As *readers leave, they decrement* the counter and the last one out does an up on the semaphore, allowing a blocked writer, if there is one, to get in.
- Suppose that while a reader is using the data base, another reader comes along. Since having two readers at the same time is not a problem, the second reader is admitted. A third and subsequent readers can also be admitted if they come along.
- Now suppose that a writer comes along. The writer cannot be admitted to the data base, since writers must have exclusive access, so the writer is suspended. Later, additional readers show up. As long as at least one reader is still active, subsequent readers are admitted. As a consequence of this strategy, as long as there is a steady supply of readers, they will all get in as soon as they arrive. The writer will be kept suspended until no reader is present. If a new reader arrives, say, every 2 seconds, and each reader takes 5 seconds to do its work, the writer will never get in.
- To prevent this situation, the program could be written slightly differently: When a reader arrives and a writer is waiting, the reader is suspended behind the writer instead of being admitted immediately. In this way, a writer has to wait for readers that were active when it arrived to finish but does not have to wait for readers that came along after it. The disadvantage of this solution is that it achieves less concurrency and thus lower performance.

The Sleeping Barber Problem

- Another classical IPC problem takes place in a barber shop. The barber shop has one barber, one barber chair, and n chairs for waiting customers, if any, to sit on. If there are no customers present, the barber sits down in the barber chair and falls asleep, as illustrated in [Figure 2-35]. When a customer arrives, he has to wake up the sleeping barber. If additional customers arrive while the barber is cutting a customer's hair, they either sit down (if there are empty chairs) or leave the shop (if all chairs are full). The problem is to program the barber and the customers without getting into race conditions. This problem is similar to various queueing situations, such as a multiperson helpdesk with a computerized call waiting system for holding a limited number of incoming calls.
- This solution uses three semaphores, *customers*, which counts waiting customers (excluding the customer in the barber chair, who is not waiting), *barbers*, the number of barbers (0 or 1) who are idle, waiting for customers, and *mutex*, which is used for mutual exclusion. We also need a variable, *waiting*, which also counts the waiting customers. The reason for having *waiting* is that there is no way to read the current value of a semaphore. In this solution, a customer entering the shop has to count the number of waiting customers. If it is less than the number of chairs, he stays; otherwise, he leaves.
- Our solution is shown [below]. When the barber shows up for work in the morning, he executes the procedure *barber*, *causing him to block on the semaphore customers* because it is initially 0. The barber then goes to sleep, as shown in [Figure 2.35]. He stays asleep until the first customer shows up.
- When a customer arrives, he executes *customer*, *starting by acquiring mutex to enter a critical region*. If another customer enters shortly thereafter, the second one will no be able to do anything until the first one has released *mutex*. *The customer then checks to see if the number of waiting customers is less than the number of chairs*. If not, he releases *mutex and leaves without a haircut*.

The sleeping Barber Preoblem(contd...)

- If there is an available chair, the customer increments the integer variable, *waiting*. Then he does an *Up on the semaphore customers*, thus *waking up the barber*. At this point, the customer and the barber are both awake. When the customer releases *mutex*, the barber grabs it, does some housekeeping, and begins the haircut.
- When the haircut is over, the customer exits the procedure and leaves the shop. Unlike our earlier examples, there is no loop for the customer because each one gets only one haircut. The barber loops, however, to try to get the next customer. If one is present, a haircut is given. If not, the barber goes to sleep.



```

#define CHAIRS 5 /* # chairs for waiting customers */
typedef int semaphore; /* use your imagination */
semaphore customers = 0; /* # of customers waiting for service */
semaphore barbers = 0; /* # of barbers waiting for customers */
semaphore mutex = 1; /* for mutual exclusion */
int waiting = 0; /* customer are waiting (not being cut) */
void barber(void)
{
while (TRUE) {
down(&customers); /* go to sleep if # of customers is 0 */
down(&mutex); /* acquire access to "waiting" */
waiting = waiting - 1; /* decrement count of waiting customers */
up(&barbers); /* one barber is now ready to cut hair */
up(&mutex); /* release 'waiting' */
cut_hair(); /* cut hair (outside critical region */
}
}

void customer(void)
{
down(&mutex); /* enter critical region */
if (waiting < CHAIRS) { /* if there are no free chairs, leave */
waiting = waiting + 1; /* increment count of waiting customers */
up(&customers); /* wake up barber if necessary */
up(&mutex); /* release access to 'waiting' */
down(&barbers); /* go to sleep if # of free barbers is 0 */
get_haircut(); /* be seated and be served */
} else {
up(&mutex); /* shop is full; do not wait */
}
}

```

fig:A solution to the Sleeping Barber Problem

OS Synchronization

Here we discuss synchronization mechanisms provided by the **Solaris** and **Windows** Operating systems.

Solaris Synchronization:

- ❑ Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing.
- ❑ Uses **adaptive mutexes** for efficiency when protecting data from short code segments
 - Starts as a standard semaphore spin-lock
 - If lock held, and by a thread running on another CPU, spins
 - If lock held by non-run-state thread, block and sleep waiting for signal of lock being released
- ❑ Uses **condition variables**
- ❑ Uses **readers-writers locks** when longer sections of code need access to data
- ❑ Uses **turnstiles** to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
 - Turnstiles are per-lock-holding-thread, not per-object
- ❑ Priority-inheritance per-turnstile gives the running thread the highest of the priorities of the threads in its

Windows XP Synchronization:

- ❑ Uses interrupt masks to protect access to global resources on uniprocessor systems
- ❑ Uses spinlocks on multiprocessor systems
- ❑ Also provides dispatcher objects which may act as either mutexes and semaphores
- ❑ Dispatcher objects may also provide events
 - An event acts much like a condition variable

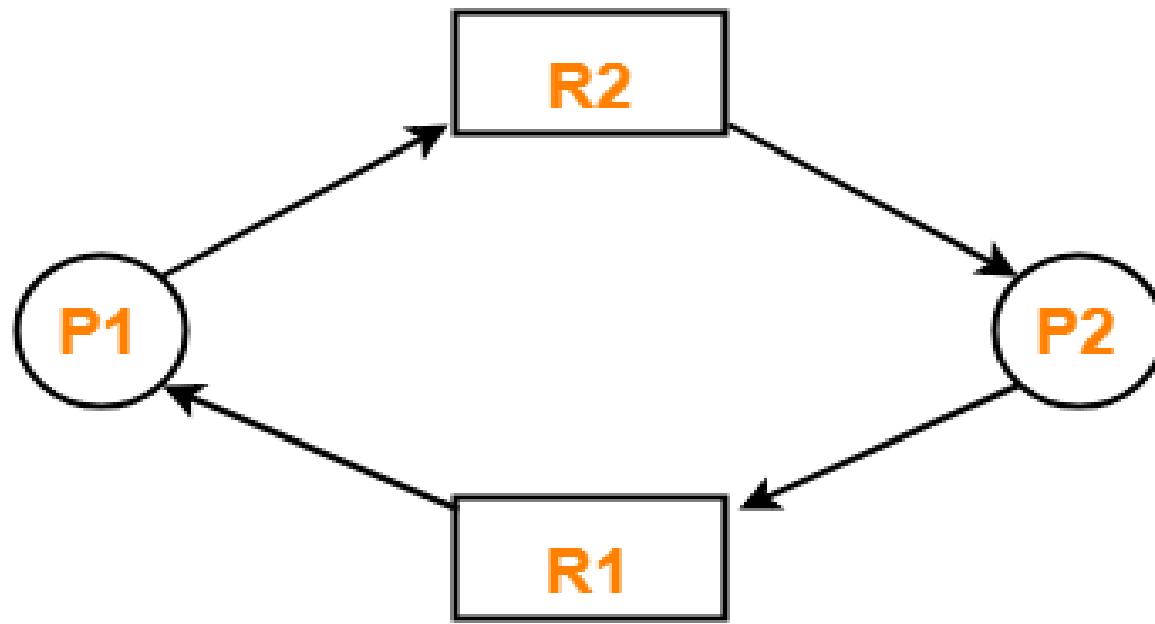
Deadlocks

Introduction:

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

OR

- › A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- › Example
 - ◆ System has 2 tape drives.
 - ◆ P1 and P2 each hold one tape drive and each needs another one.



Example of a deadlock

Bridge Crossing Example

- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

Preemptable and Nonpreemptable Resources

- A **Preemptable** resource is one that can be taken away from the process owning it with no ill effects. Such types of resources are reusable.
 - Memory, printers, tape drives are example of Preemptable resources.
- A **Nonpreemptable** resource, in contrast, is one that cannot be taken away from its current owner without causing the computation to fail.
 - If a process has begun to burn a CD-ROM, suddenly taking the CD recorder away from it and giving it to another process will result in a garbled CD.

Conditions for Deadlock

Deadlock can arise if four conditions hold simultaneously.

- Mutual exclusion: only one process at a time can use a resource.
- Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes.
- No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- Circular wait: there exists a set $\{P_0, P_1, \dots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1 , P_1 is waiting for a resource that is held by P_2 , ..., P_{n-1} is waiting for a resource that is held by P_n , and P_n is waiting for a resource that is held by P_0 .

Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state.
- Allow the system to enter a deadlock state and then Recover.
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

Deadlock Prevention

As we know, for a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

- ♦ **Mutual Exclusion-** The mutual-exclusion condition must hold for nonsharable resources. For example, a printer cannot be simultaneously shared by several processes. Sharable resources, on the other hand, do not require mutually exclusive access, and thus cannot be involved in a deadlock
- ♦ **Hold and Wait –** must guarantee that whenever a process requests a resource, it does not hold any other resources.
 - ◆ Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.
 - ◆ Disadvantages: Low resource utilization; starvation possible.

Deadlock Prevention (Cont.)

- **No Preemption –**
 - ◆ If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
 - ◆ Preempted resources are added to the list of resources for which the process is waiting.
 - ◆ Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- **Circular Wait –** impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

Deadlock Avoidance

Possible side effects of preventing deadlocks by Deadlock-prevention algorithms method, however, are low device utilization and reduced system throughput.

An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested

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

Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Detection-Algorithm Usage

- ◆ When, and how often, to invoke depends on:
 - ◆ How often a deadlock is likely to occur?
 - ◆ How many processes will need to be rolled back?
 - ✓ one for each disjoint cycle
- ◆ If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.
- In which order should we choose to abort?
 - ◆ Priority of the process.
 - ◆ How long process has computed, and how much longer to completion.
 - ◆ Resources the process has used.
 - ◆ Resources process needs to complete.
 - ◆ How many processes will need to be terminated.
 - ◆ Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

If preemption is required to deal with deadlocks, then three issues need to be addressed:

- Selecting a victim – Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost.
- Rollback – return to some safe state, restart process for that state.
- Starvation – same process may always be picked as victim, include number of rollback in cost factor.

Issues Related to Deadlock:

- ◆ **Two Phase Locking**

- ◆ Although both avoidance and prevention are not terribly promising in the general case.
- ◆ As an example, in many database systems, an operation that occurs frequently is requesting locks on several records and then updating all the locked records. When multiple processes are running at the same time, there, there is a real danger of deadlock.
- ◆ The Approach often used is called two-phase locking.
- ◆ In the first phase, the process tries to lock all the records it needs, one at a time. If it succeeds, it begins the second phase, performing its updates and releasing the locks.

Issues Related to Deadlock(contd..):

- ◆ **Non Resource Deadlocks**

- ◆ This type of deadlock can occur in communication systems, in which two or more processes communicate by sending messages.
- ◆ A common arrangement is that process A sends a request message to process B, and then blocks until B sends back a reply message.
- ◆ Suppose that the request message gets lost.
- ◆ A is blocked waiting for the reply. B is blocked waiting for a request asking it to do something. We have a deadlock. Infact, there are no resources at all in the sight.

- ◆ **Starvation**

- ◆ In a dynamic system, requests for resources happen all the time. Some policy is needed to make a decision about who gets which resource when. This policy, although seemingly reasonable, may lead to some processes never getting service even though they are not deadlocked.
- ◆ This is known as starvation

Process scheduling examples:

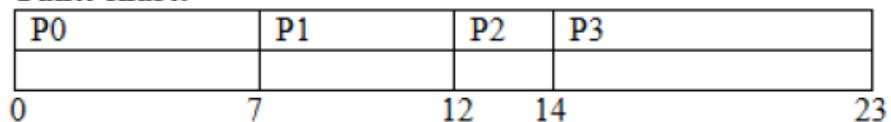
Q1: Find the average waiting time and average turnaround time for executing the following processes using FCFS (first-come first-service) scheduling?

Process	Burst time
P0	7
P1	5
P2	2
P3	9

Answer:

The first step of the solution is founding the Gantt chart.

Gantt chart:



The waiting time of P0 = 0

The waiting time of P1 = 7

The waiting time of P2 = 12

The waiting time of P3 = 14

The average waiting time = $(0 + 7 + 12 + 14) / 4 = 33/4 = 8.25$

The turnaround time of p0 = 7

The turnaround time of p1 = 12

The turnaround time of p2 = 14

The turnaround time of p3 = 23

The average turnaround time = $(7+12+14+23)/4 = 56/4 = 14$

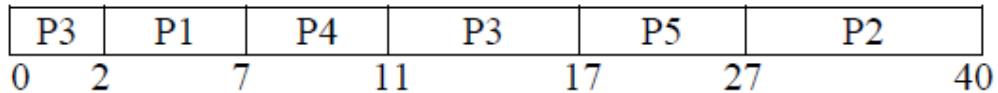
Q2: Find the average waiting time (A.W.T) and average turnaround time (A.T.A.T) for executing the following process using (1) Preemptive short-job first (2) Non-preemptive short-job first?

Process	P1	P2	P3	P4	P5
Burst time	5	13	8	4	10
Arrival time	2	3	0	5	1

Answer

(1) Using preemptive short-job first

Gantt chart



$$W.T. \text{ of } P1 = 2 - 2 = 0$$

$$T.A.T. \text{ of } P1 = 7 - 2 = 5$$

$$W.T. \text{ of } P2 = 27 - 3 = 24$$

$$T.A.T. \text{ of } P2 = 40 - 3 = 37$$

$$W.T. \text{ of } P3 = 0 + (11 - 2) = 9$$

$$T.A.T. \text{ of } P3 = 17 - 0 = 17$$

$$W.T. \text{ of } P4 = 7 - 5 = 2$$

$$T.A.T. \text{ of } P4 = 11 - 5 = 6$$

$$W.T. \text{ of } P5 = 17 - 1 = 16$$

$$T.A.T. \text{ of } P5 = 27 - 1 = 26$$

$$A.W.T. = (0 + 24 + 9 + 2 + 16) / 5 = 51 / 5 = 10.2$$

$$A.T.A.T. = (5 + 37 + 17 + 6 + 26) / 5 = 91 / 5 = 18.2$$

2) Using non-preemptive short-job first

Gantt chart

P3	P4	P1	P5	P2
0	8	12	17	27

$$\text{W.T. of } P1 = 12 - 2 = 10$$

$$\text{T.A.T. of } P1 = 17 - 2 = 15$$

$$\text{W.T. of } P2 = 27 - 3 = 24$$

$$\text{T.A.T. of } P2 = 40 - 3 = 37$$

$$\text{W.T. of } P3 = 0$$

$$\text{T.A.T. of } P3 = 8 - 0 = 8$$

$$\text{W.T. of } P4 = 8 - 5 = 3$$

$$\text{T.A.T. of } P4 = 12 - 5 = 7$$

$$\text{W.T. of } P5 = 17 - 1 = 16$$

$$\text{T.A.T. of } P5 = 27 - 1 = 26$$

$$\text{A.W.T.} = (10+24+0+3+16)/5 = 53/5 = 10.6$$

$$\text{A.T.A.T} = (15+37+8+7+26)/5 = 93/5 = 18.6$$

Q3: Find the average waiting time and turnaround time for executing the following process using priority scheduling algorithm?

Process	P1	P2	P3	P4	P5
Burst time	5	13	8	6	12
Priority	1	3	0	4	2

Answer

Gantt chart

P3	P1	P5	P2	P4
0	8	13	25	38

$$\text{W.T. of } P1 = 8 \quad \text{T.A.T. of } P1 = 13$$

$$\text{W.T. of } P2 = 25 \quad \text{T.A.T. of } P2 = 38$$

$$\text{W.T. of } P3 = 0 \quad \text{T.A.T. of } P3 = 8$$

$$\text{W.T. of } P4 = 38 \quad \text{T.A.T. of } P4 = 44$$

$$\text{W.T. of } P5 = 13 \quad \text{T.A.T. of } P5 = 25$$

$$\text{A.W.T.} = (8+25+0+38+13)/5 = 84/5 = 16.8$$

$$\text{A.T.A.T} = (13+38+8+44+25)/5 = 128/5 = 25.6$$

Q4: Find the average waiting time (A.W.T.) and the average turnaround time (A.T.A.T.) for executing the following processes using round-robin algorithm, where time quantum is 5?

Process	P1	P2	P3	P4	P5
Burst time	11	4	14	9	21
Arrival time	5	0	0	1	2

Answer

$$\begin{aligned}
 P2 &= \cancel{A} \quad 0 \\
 P3 &= \cancel{14} \quad \cancel{A} \quad 0 \\
 P4 &= \cancel{9} \quad \cancel{A} \quad 0 \\
 P5 &= \cancel{21} \quad 16 \quad 11 \quad \cancel{6} \quad \cancel{A} \quad 0 \\
 P1 &= \cancel{5} \quad \cancel{6} \quad \cancel{A} \quad 0
 \end{aligned}$$

Gantt chart

P2	P3	P4	P5	P1	P3	P4	P5	P1	P3	P5	P1	P5	P5
0	4	9	14	19	24	29	33	38	43	47	52	53	58

$$\text{W.T. of P1} = (19-5) + (38-24) + (52-43) = 14 + 14 + 9 = 37$$

$$\text{W.T. of P2} = 0$$

$$\text{W.T. of P3} = (4-0) + (24-9) + (43-29) = 4 + 15 + 14 = 33$$

$$\text{W.T. of P4} = (9-1) + (29-14) = 8 + 15 = 23$$

$$\text{W.T. of P5} = (14-2) + (33-19) + (47-38) + (53-52) = 12 + 14 + 9 + 1 = 36$$

$$\text{A.W.T.} = (37 + 0 + 33 + 23 + 36) / 5 = 129 / 5 = 25.8$$

$$\text{T.A.T of P1} = (53-5) = 48$$

$$\text{T.A.T of P2} = (4-0) = 4$$

$$\text{T.A.T of P3} = (47 - 0) = 47$$

$$\text{T.A.T of P4} = (33-1) = 32$$

$$\text{T.A.T of P5} = (59-2) = 57$$

$$\text{A.T.A.T.} = (48 + 4 + 47 + 32 + 57) / 5 = 188 / 5 = 37.6$$

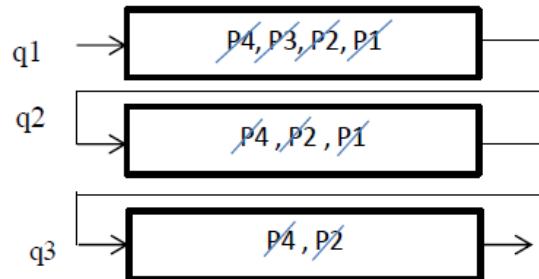
Q25: Consider a multilevel feedback queue scheduling (MLFBQ) with three queues q1, q2, and q3. q1 and q2 use round-robin algorithm with time quantum (TQ) = 5, and 4 respectively. q3 use first-come first-service algorithm. Find the average waiting time (A.W.T) and average turnaround time (A.T.A.T) for executing the following process?

Processes	P1	P2	P3	P4
Burst time	8	22	4	12

Answer

In MLFBQ scheduling algorithm, the process move between queues. If a process uses too much CPU time, it will be moved to a lower-priority queue.

	q1	q2	q3
P1:	8	3	-
P2:	22	17	13
P3:	4	-	-
P4:	12	7	3



Gantt chart

P1	P2	P3	P4	P1	P2	P2	P4	P2	P4
0	5	10	14	19	22	26	30	43	46

$$\text{W.T. of } P1 = 0 + (19-5) = 14$$

$$\text{T.A.T. of } P1 = 22$$

$$\text{W.T. of } P2 = 5 + (22-10) + (30-26) = 5+12+4 = 21$$

$$\text{T.A.T. of } P2 = 43$$

$$\text{W.T. of } P3 = 10$$

$$\text{T.A.T. of } P3 = 14$$

$$\text{W.T. of } P4 = 14 + (26-19) + (43-30) = 14+7+13 = 34$$

$$\text{T.A.T. of } P4 = 46$$

$$\text{A.W.T.} = (14 + 21 + 10 + 34)/4 = 79/4 = 19.75$$

$$\text{A.T.A.T} = (22 + 43 + 14 + 46) = 125/4 = 31.25$$

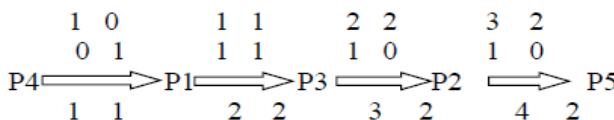
Deadlock Avoidance examples(using Banker's Algorithm)

Q26: Suppose we have two resources, A, and B. A has 6 instances and B has 3 instances. Can the system execute the following processes without deadlock occurring?

Process	Allocate		Maximum need	
	A	B	A	B
P1	1	1	2	2
P2	1	0	4	2
P3	1	0	3	2
P4	0	1	1	1
P5	2	1	6	3

Answer

Available: A= 1; B= 0



Process	Need	
	A	B
P1	1	1
P2	3	2
P3	2	2
P4	1	0
P5	4	2

We can execute the processes in the sequence <P4, P1, P3, P2, P5> without deadlock.

Q27: Consider we have five processes P0, P1, . . . P5 and three resources A, B, and C. Is the executing the following processes in the safe state?

Process	Allocation			Maximum need			Available		
	A	B	C	A	B	C	A	B	C
P0	1	2	0	2	2	2	0	1	0
P1	1	0	0	1	1	0			
P2	1	1	1	1	4	3			
P3	0	1	1	1	1	1			
P4	0	0	1	1	2	2			
P5	1	0	0	1	5	1			

Answer

We find the resources that need for each process

Process	Need		
	P0	P1	P2
P0	1	0	2
P1	0	1	0
P2	0	3	2
P3	1	0	0
P4	1	2	1
P5	0	5	1

$\begin{array}{ccc} 0 & 1 & 0 \\ 1 & 0 & 0 \\ \hline 1 & 1 & 0 \end{array}$	$\begin{array}{ccc} 1 & 1 & 0 \\ 0 & 1 & 1 \\ \hline 1 & 2 & 1 \end{array}$	$\begin{array}{ccc} 1 & 2 & 1 \\ 0 & 0 & 1 \\ \hline 1 & 2 & 2 \end{array}$	$\begin{array}{ccc} 1 & 2 & 2 \\ 1 & 2 & 0 \\ \hline 2 & 4 & 2 \end{array}$	$\begin{array}{ccc} 2 & 4 & 2 \\ 1 & 1 & 1 \\ \hline 3 & 5 & 3 \end{array}$
P1	p3	p4	p0	p2

The process in safe state if they are executed in the sequence<P1, P3, P4, P0, P2, P5>

Q28: Suppose we have five processes and three resources, A, B, and C. A has 2 instances, B has 5 instances and C has 4 instances. Can the system execute the following processes without deadlock occurring, where we have the following?

Process	Maximum need			Allocation		
	A	B	C	A	B	C
P1	1	2	3	0	1	1
P2	2	2	0	0	1	0
P3	0	1	1	0	0	1
P4	3	5	3	1	2	1
P5	1	1	2	1	0	1

Applied Operating System

Chapter 3: Memory Management

Prepared By:

Amit K. Shrivastava

Asst. Professor

Nepal College Of Information Technology

Concept

- . Memory management is the functionality of an operating system which handles or manages primary memory. Memory management keeps track of each and every memory location either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.
- Address binding of instructions and data to memory addresses can happen at three different stages
 - Compile time -- When it is known at compile time where the process will reside, compile time binding is used to generate the absolute code.
 - Load time -- When it is not known at compile time where the process will reside in memory, then the compiler generates re-locatable code.
 - Execution time -- If the process can be moved during its execution from one memory segment to another, then binding must be delayed to be done at run time

Logical vs. Physical Address Space

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

- 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.

.The run-time mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device. MMU uses following mechanism to convert virtual address to physical address.

.The value in the base register is added to every address generated by a user process which is treated as offset at the time it is sent to memory. For example, if the base register value is 10000, then an attempt by the user to use address location 100 will be dynamically reallocated to location 10100.

.The user program deals with virtual addresses; it never sees the real physical addresses.

Swapping

A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.

Backing store – fast disk large enough to accommodate copies of all memory images for all users; 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.

Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped. Let us assume that the user process is of size 100KB and the backing store is a standard hard disk with transfer rate of 1 MB per second. The actual transfer of the 100K process to or from memory will take

$$100\text{KB} / 1000\text{KB per second} = 1/10 \text{ second} = 100 \text{ milliseconds}$$

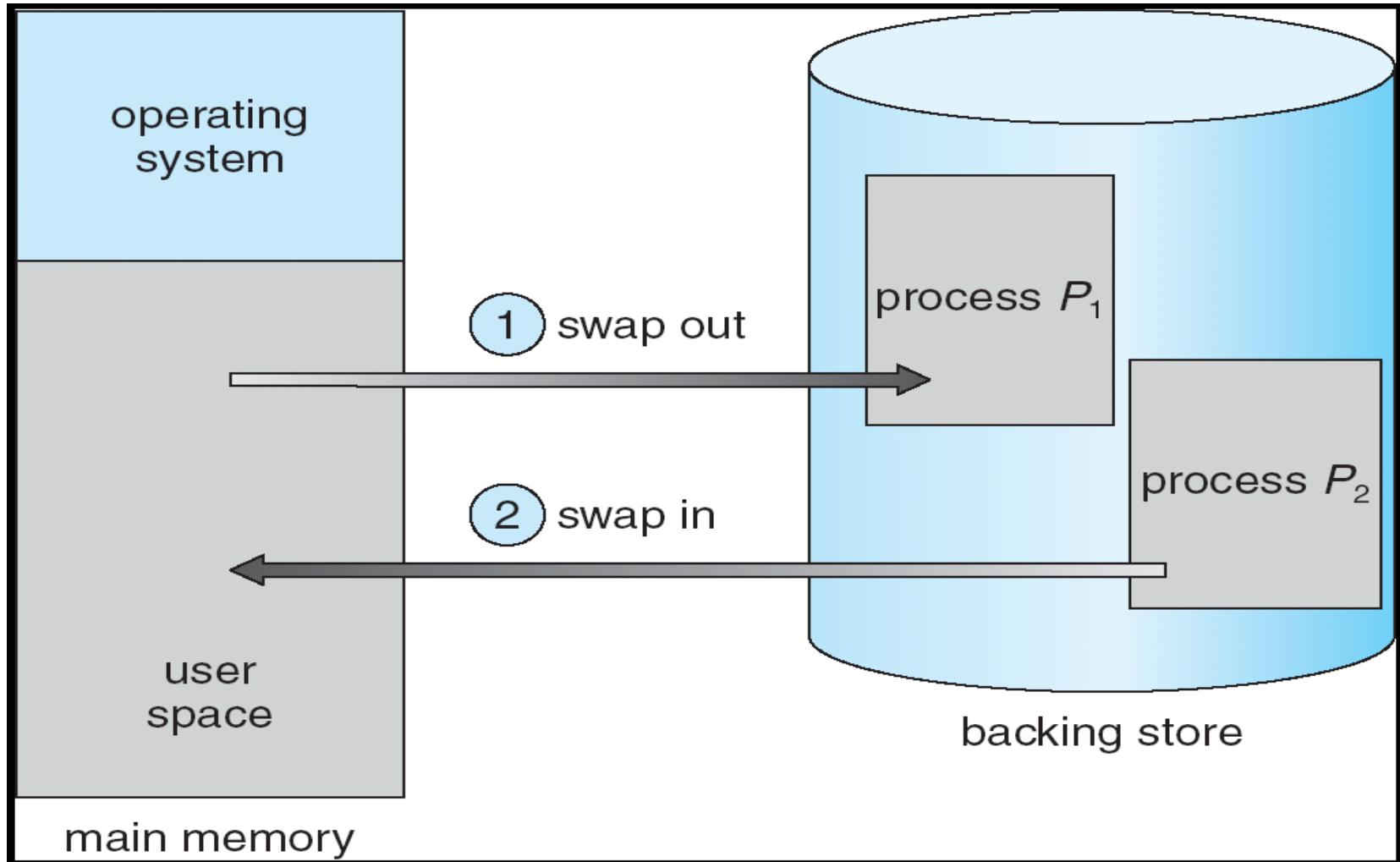


Fig: Schematic View of Swapping

Storage Hierarchy:

In 1960 storage hierarchy is extended by one more level, i.e the cache which is a high-speed storage that is much faster than main storage

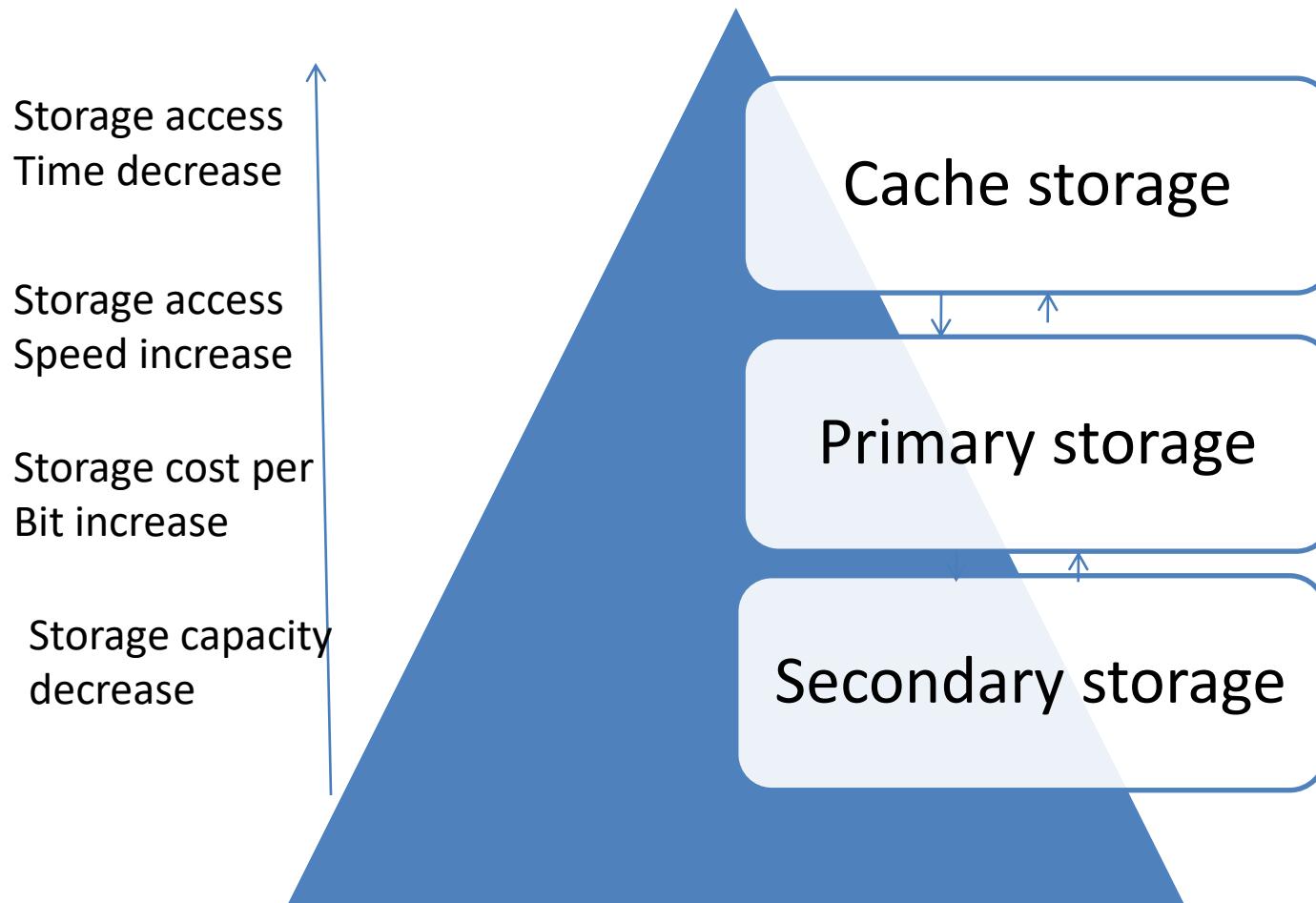


Fig: Hierarchical storage organization

Storage Management Strategies:

- Storage Management strategies are generated for obtaining the best possible use of the main storage. They are divided into following categories:
 - 1. Fetch Strategies
 - a) Demand
 - b) Anticipatory
 - 2. Placement Strategies
 - 3. Replacement Strategies

- Fetch Strategies are concerned with when to obtain the next piece of program or data for transfer to main storage from secondary storage.
- Placement Strategies are concerned with determining where in main storage to place an incoming program. E.g. first-fit, best-fit, and worst-fit
- Replacement Strategies are concerned with determining which piece of program or data to displace to make room for incoming programs.

Contiguous Memory Allocation

- . In this type of allocation, relocation-register scheme is used to protect user processes from each other, and from changing operating-system code and data. Relocation register contains value of smallest physical address whereas limit register contains range of logical addresses. Each logical address must be less than the limit register.
- . **Multiple-Partition allocation:** In this type of allocation, main memory is divided into a number of fixed-sized partitions where each partition should contain only one process. When a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.

Fragmentation:

- **External Fragmentation** – total memory space exists **to** satisfy a request, but it is not contiguous.
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.
- Reduce external fragmentation by compaction
 - ◆ Shuffle memory contents to place all free memory together in one large block.
 - ◆ Compaction is possible *only if relocation is dynamic, and is done at execution time.*
 - ◆ I/O problem
 - ✓ Latch job in memory while it is involved in I/O.
 - ✓ Do I/O only into OS buffers.

Fixed Partition Multiprogramming

In a multiprogramming environment, several programs reside in primary memory at a time and the CPU passes its control rapidly between these programs. One way to support multiprogramming is to divide the main memory into several partitions each of which is allocated to a single process. Depending upon how and when partitions are created, there may be two types of memory partitioning: (1) Static(fixed) and (2) Dynamic(variable). Static partitioning implies that the division of memory into number of partitions and its size is made in the beginning (during the system generation process) and remain fixed thereafter. The basic approach here is to divide memory into several fixed size partitions where each partition will accommodate only one program for execution. The number of programs (i.e. degree of multiprogramming) residing in memory will be bound by the number of partitions. When a program terminates, that partition is free for another program waiting in a queue.

These jobs
run only in
partition 1.

These jobs
run only in
partition 2.

These jobs
run only in
partition 3.

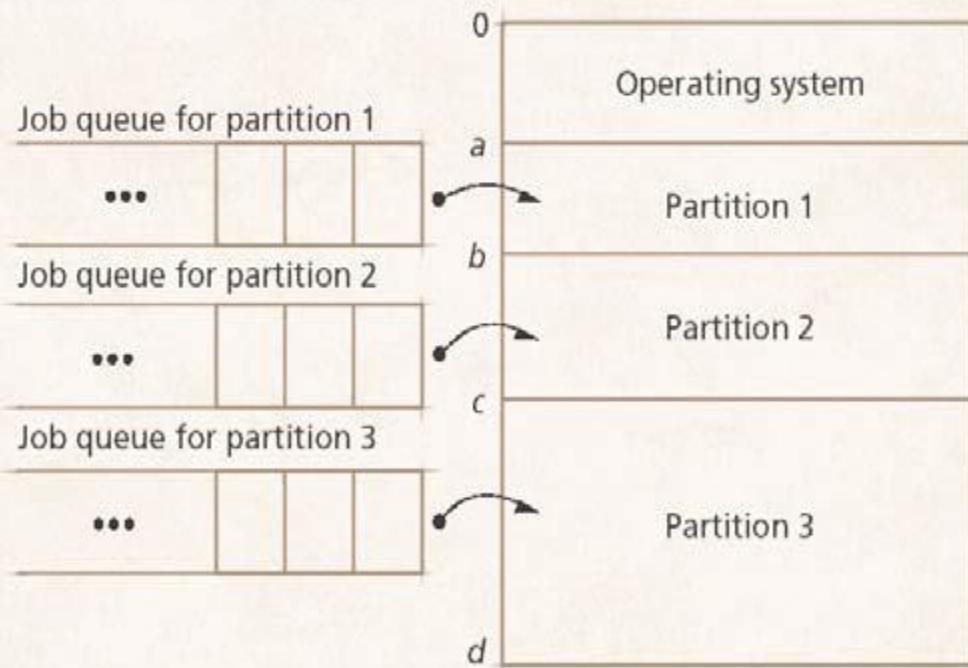


Figure : Fixed-partition multiprogramming with absolute translation and loading.

Drawbacks to fixed partitions

- Early implementations used absolute addresses
 - If the requested partition was full, code could not load
 - Later resolved by relocating compilers

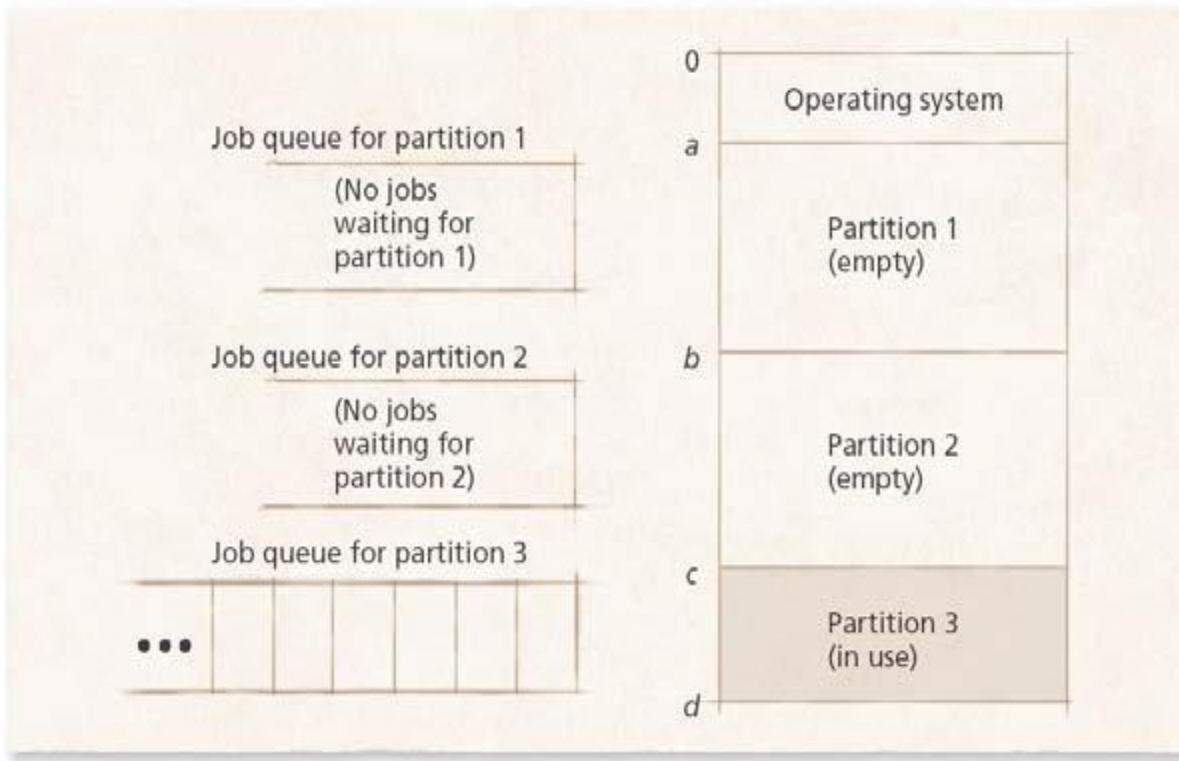


Figure : Memory waste under fixed-partition multiprogramming with absolute translation and loading.

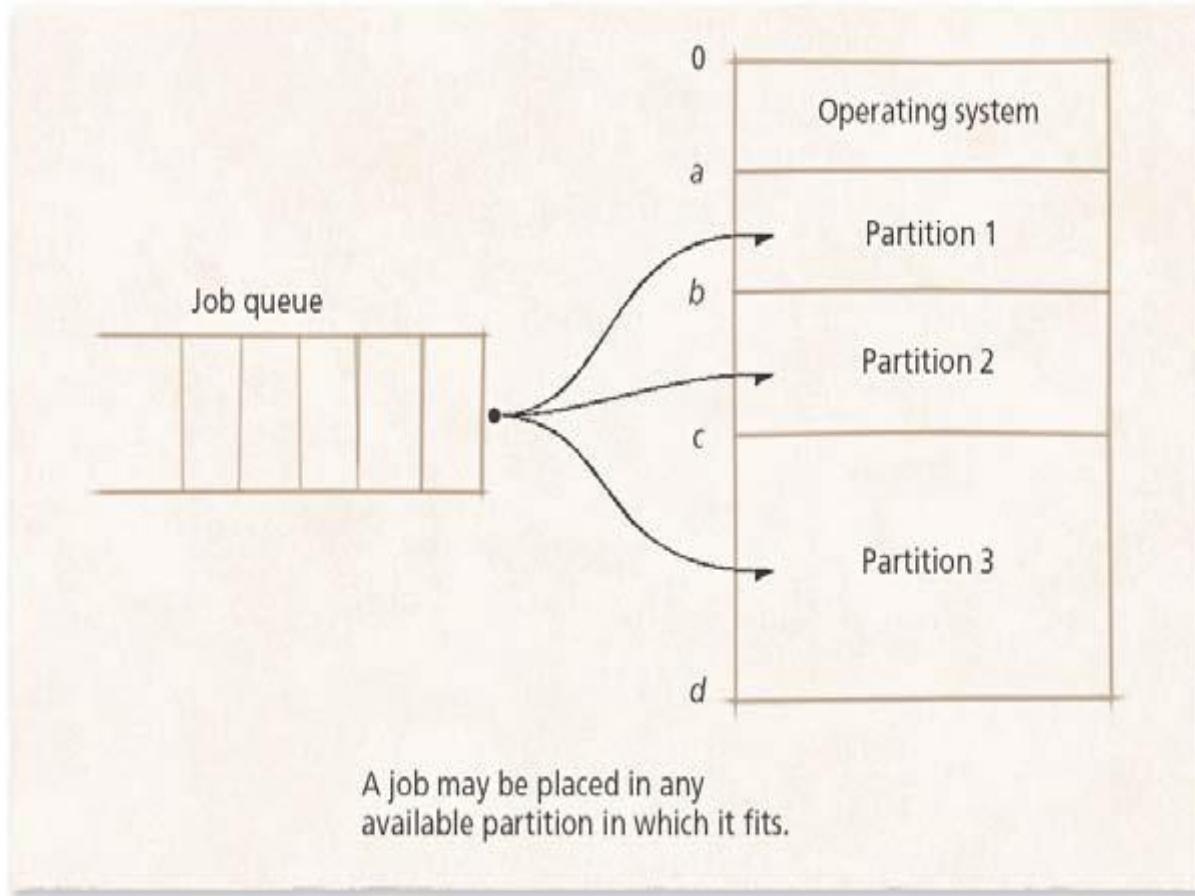


Figure : Fixed-partition multiprogramming with relocatable translation and load

Variable-Partition Multiprogramming

System designers found fixed partitions too restrictive

- Internal fragmentation
- Potential for processes to be too big to fit anywhere
- Variable partitions designed as replacement

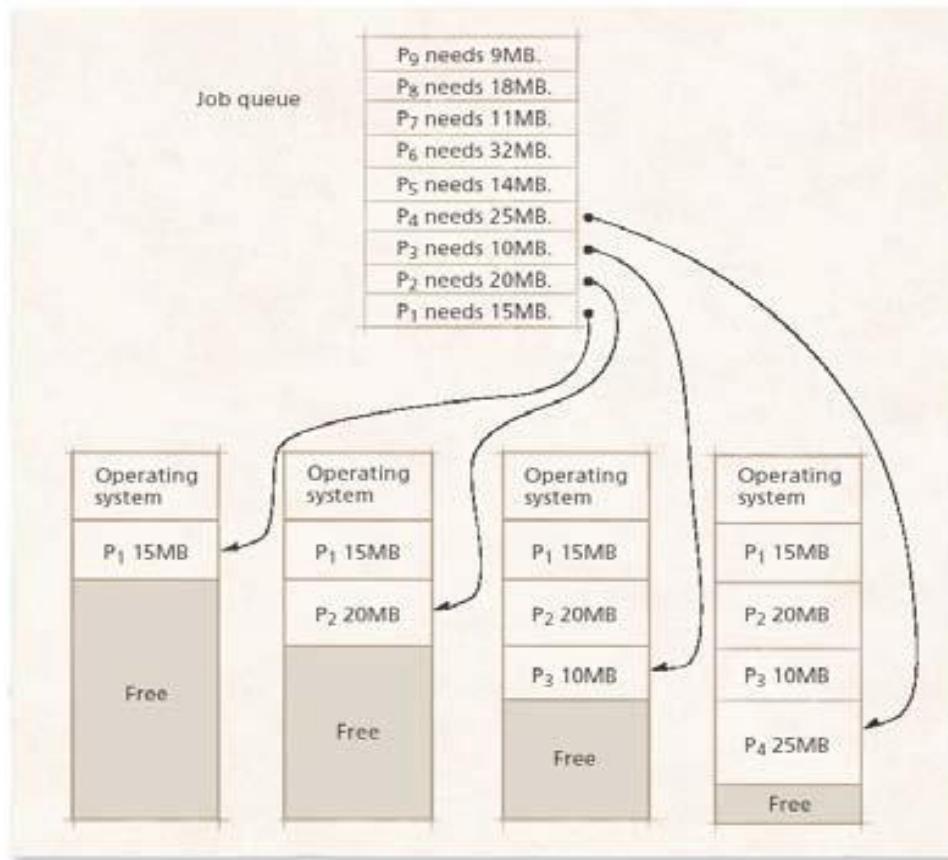


Figure : Initial partition assignments in variable-partition programming.

Variable-Partition Characteristics

Jobs placed where they fit

- No space wasted initially
- Internal fragmentation impossible
 - Partitions are exactly the size they need to be
- External fragmentation can occur when processes removed
 - Leave holes too small for new processes
 - Eventually no holes large enough for new processes

Variable-Partition Characteristics

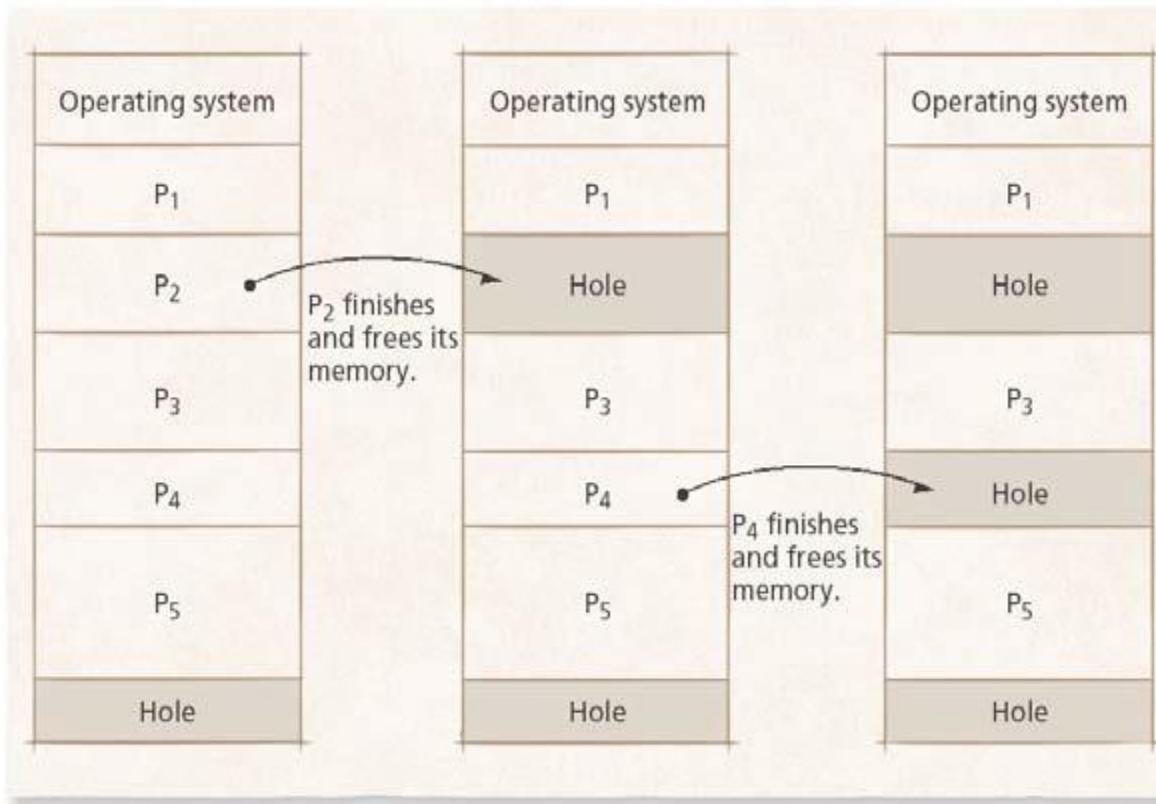


Figure : Memory “holes” in variable-partition multiprogramming.

Variable-Partition Characteristics

- Several ways to combat external fragmentation
 - Coalescing
 - Combine adjacent free blocks into one large block
 - Often not enough to reclaim significant amount of memory
 - Compaction
 - Sometimes called garbage collection (not to be confused with GC in object-oriented languages)
 - Rearranges memory into a single contiguous block of free space and a single contiguous block of occupied space
 - Makes all free space available
 - Significant overhead

Variable-Partition Characteristics

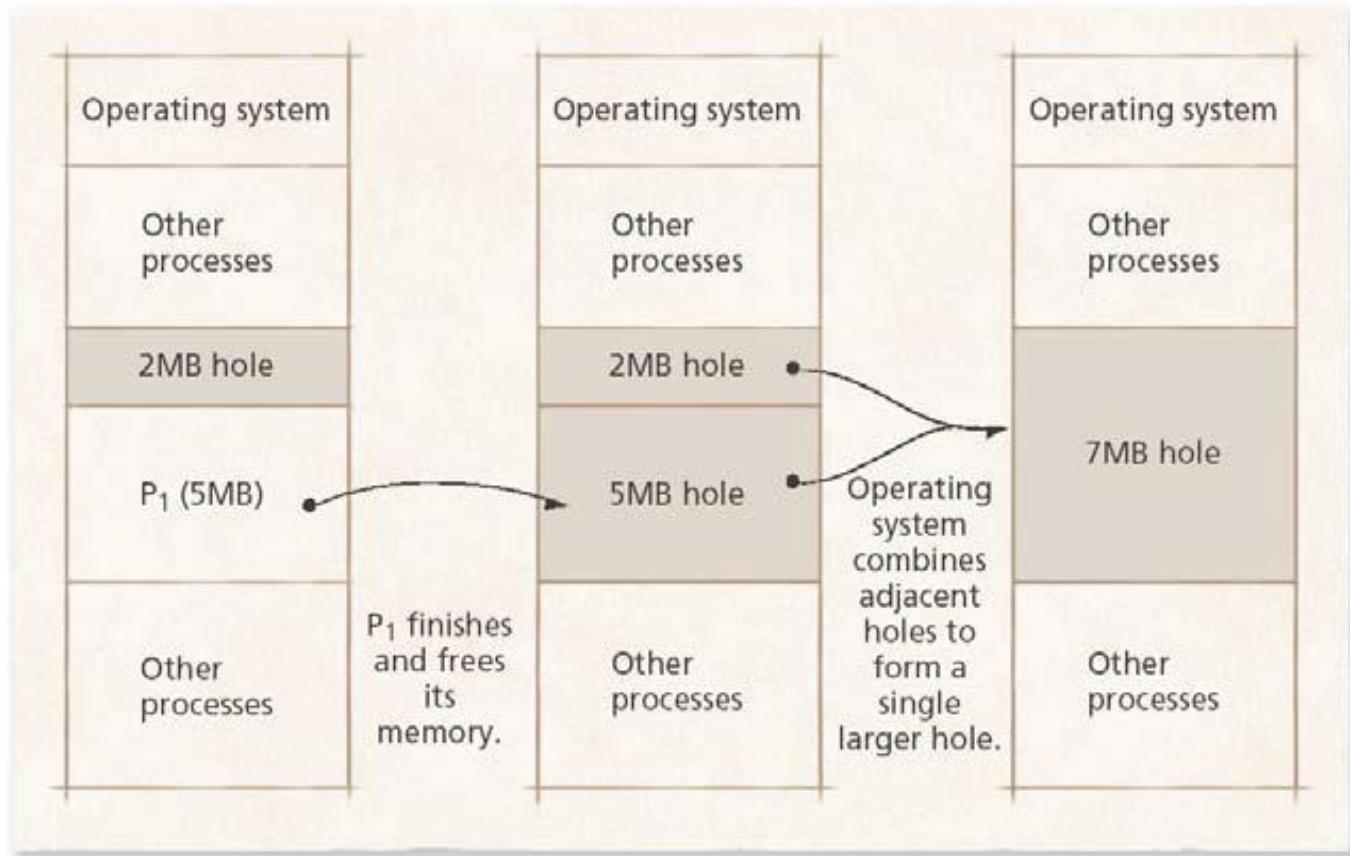


Figure : Coalescing memory “holes” in variable-partition multiprogramming

Variable-Partition Characteristics

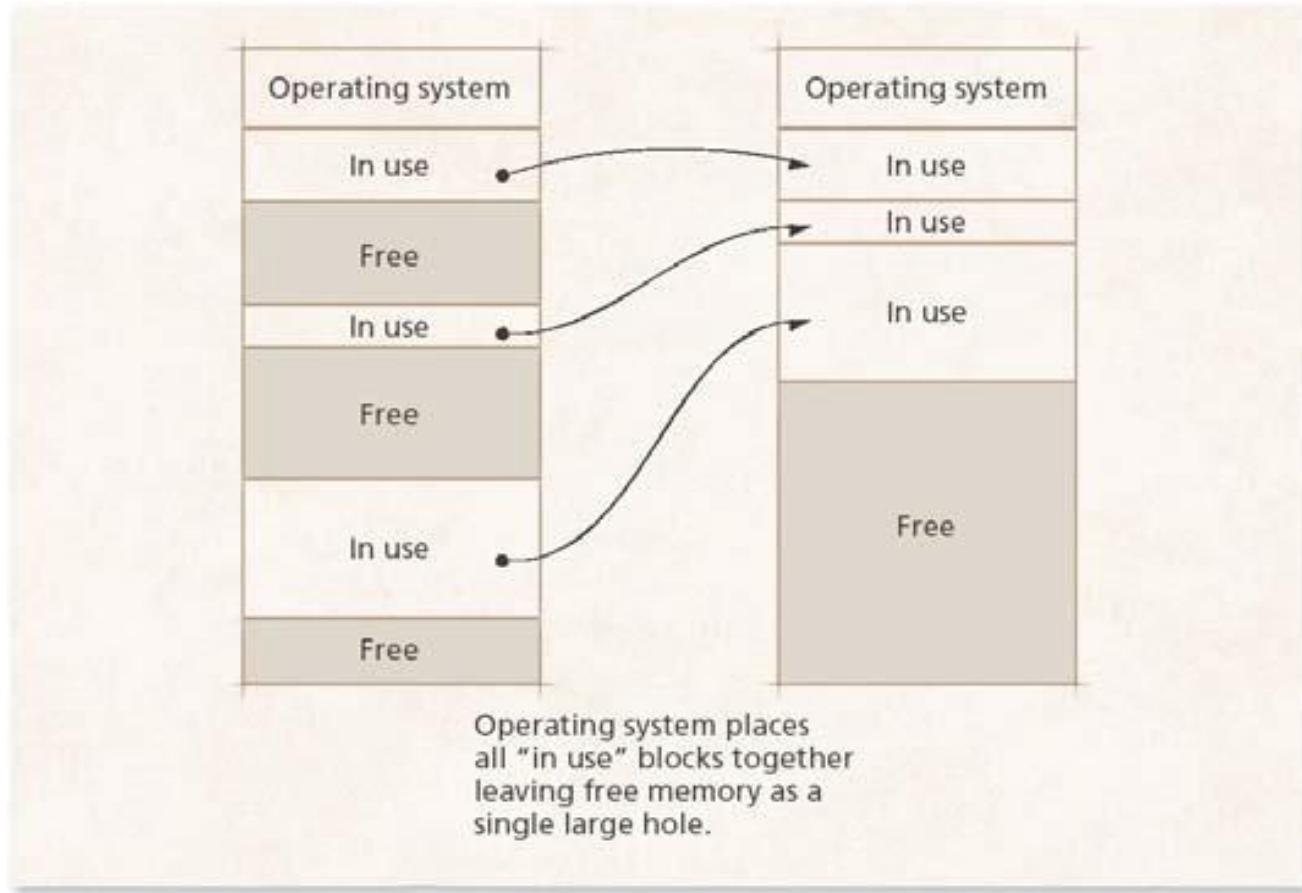


Figure : Memory compaction in variable-partition multiprogramming

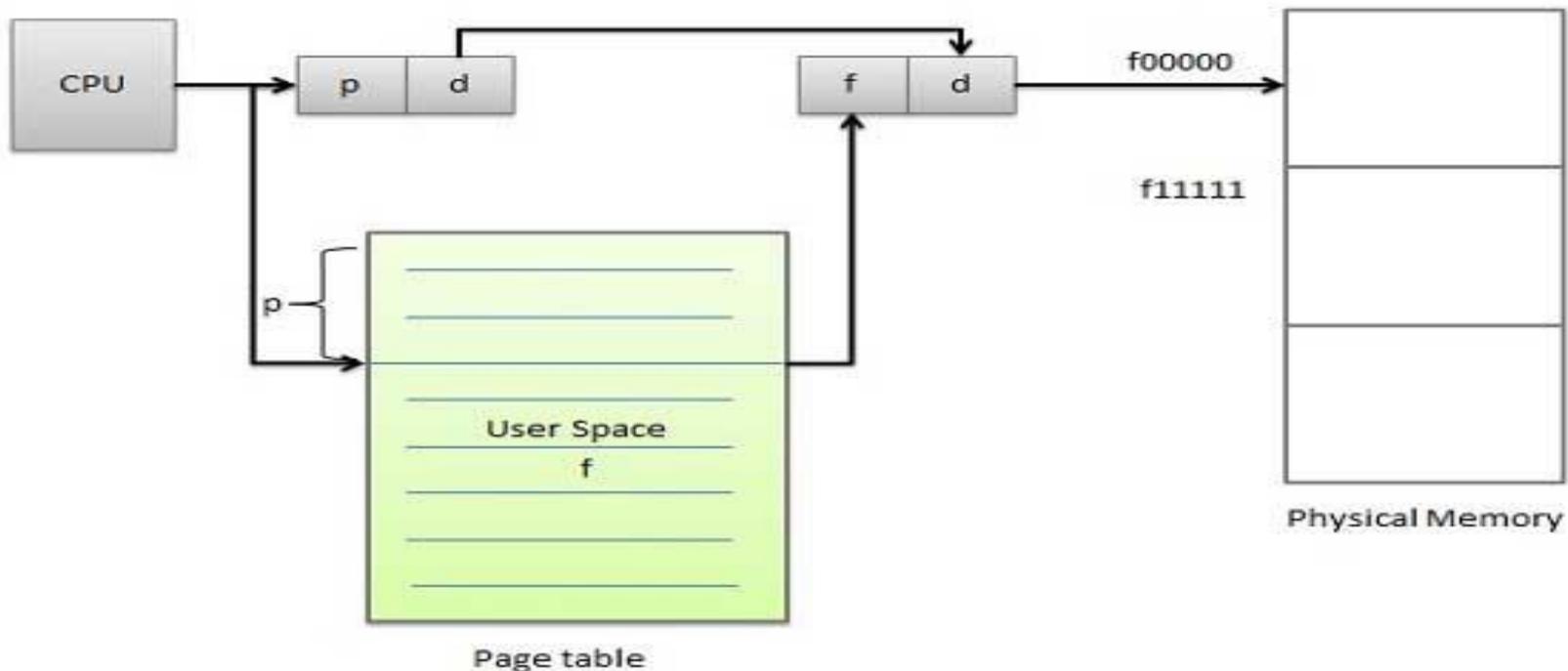
Paging:

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available.
 - Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8192 bytes).
 - Divide logical memory into blocks of same size called **pages**.
 - Keep track of all free frames.
 - To run a program of size n pages, *need to find n free frames* and load program.
 - Set up a page table to translate logical to physical addresses.
 - Internal fragmentation

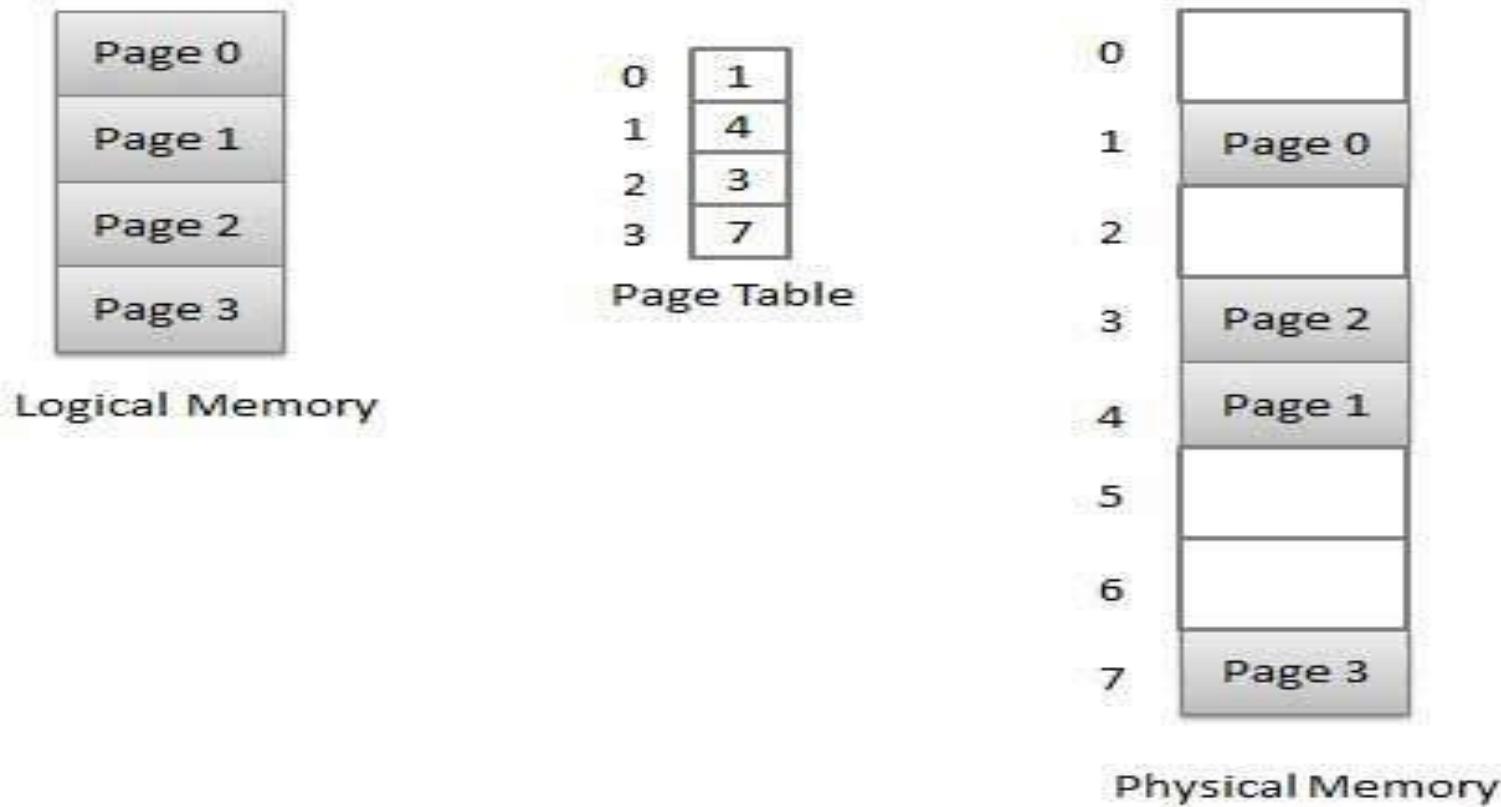
Paging(contd..)

Address generated by CPU is divided into

- Page number (p) -- page number is used as an index into a page table which contains base address of each page in physical memory.
- Page offset (d) -- page offset is combined with base address to define the physical memory address.



Following figure show the paging table architecture



Page Table Structure

- Hierarchical Paging
 - Hashed Page Tables
 - Inverted Page Tables

Hierarchical Page Tables

- Break up the logical address space into multiple page tables.
- A simple technique is a two-level page table.

Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
 - ◆ a page number consisting of 20 bits.
 - ◆ a page offset consisting of 12 bits.
- Since the page table is paged, the page number is further divided into:
 - ◆ a 10-bit page number.
 - ◆ a 10-bit page offset.
- Thus, a logical address is as follows:

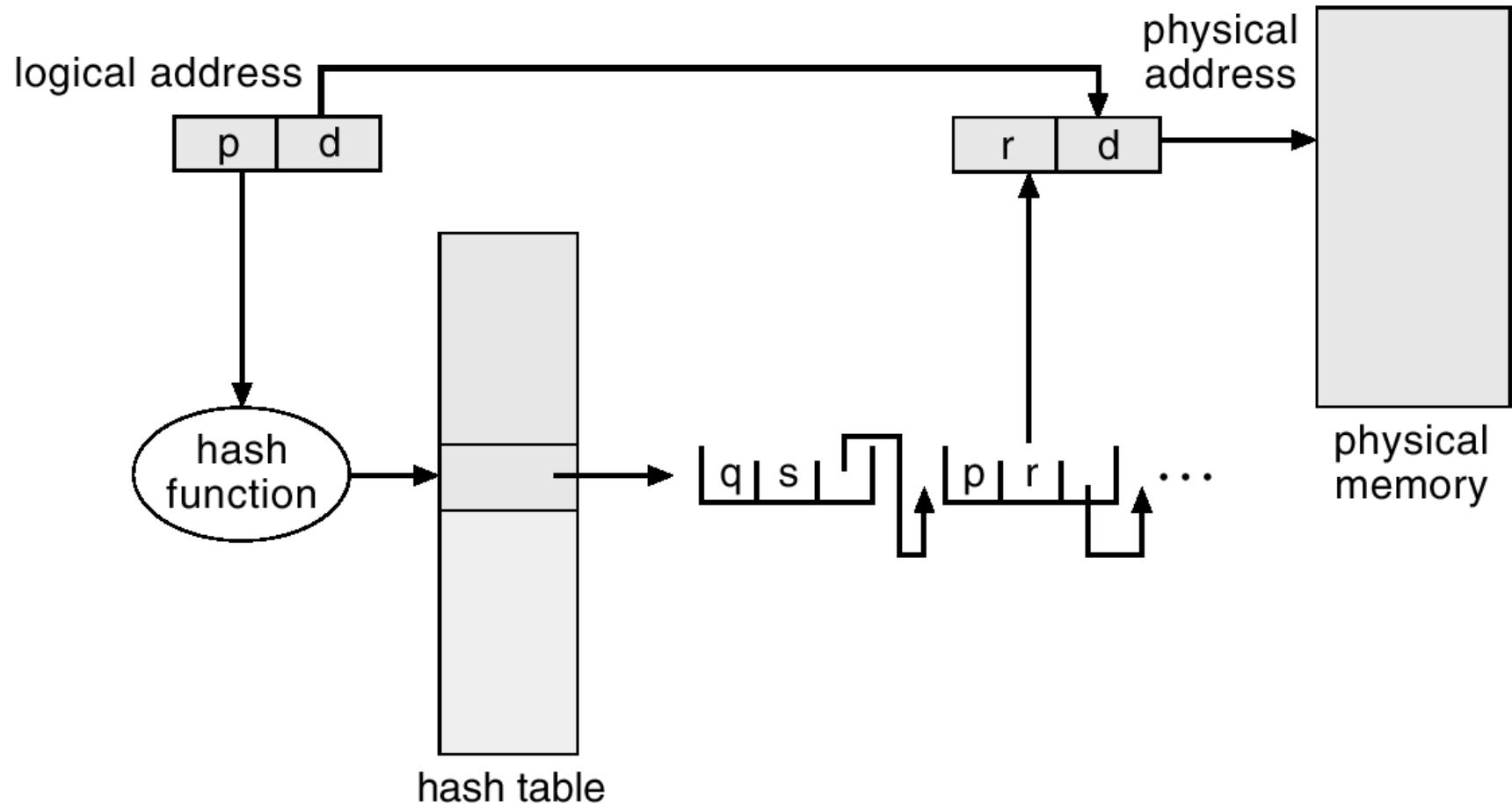
page number	page offset
p_1	p_2
10	10 12

- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the outer page table

Hashed Page Tables

- Common in address spaces > 32 bits.
 - The virtual page number is hashed into a page table. This page table contains a chain of elements hashing to the same location.
 - Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.

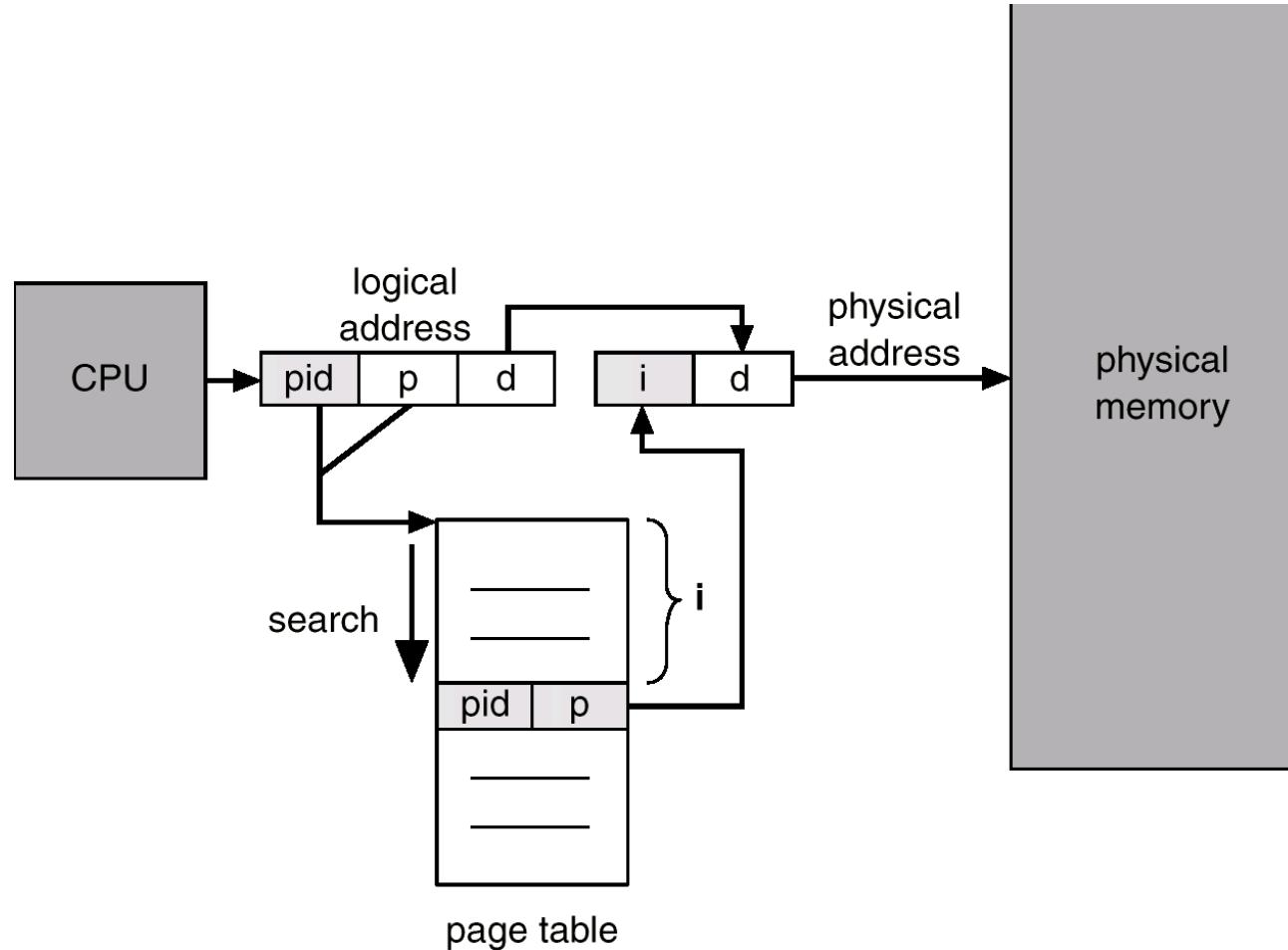
Hashed Page Table



Inverted Page Table

- One entry for each real page of memory.
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.
- Use hash table to limit the search to one — or at most a few — page-table entries.

Inverted Page Table Architecture



TLB(Translation Look aside Buffers)

A translation look aside buffer(TLB) is a memory cache that stores recent translations of virtual memory to physical addresses for faster retrieval.

Each entry in the TLB consists of two parts: a key (or tag) and a value.

When the associative memory is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value field is returned. The search is fast; a TLB lookup in modern hardware is part of the instruction pipeline, essentially adding no performance penalty. To be able to execute the search within a pipeline step, however, the TLB must be kept small. It is typically between 32 and 1,024 entries in size. The TLB is used with page tables in the following way. The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, its page number is presented to the TLB. If the page number is found, its frame number is immediately available and is used to access memory. As just mentioned, these steps are executed as part of the instruction pipeline within the CPU, adding no performance penalty compared with a system that does not implement paging.

TLB(Translation Look aside Buffers(contd...))

- If the page number is not in the TLB (known as a **TLB miss**), a **memory** reference to the page table must be made. Depending on the CPU, this may be done automatically in hardware or via an interrupt to the operating system. When the frame number is obtained, we can use it to access memory (Figure in next slide). In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference. If the TLB is already full of entries, an existing entry must be selected for replacement.

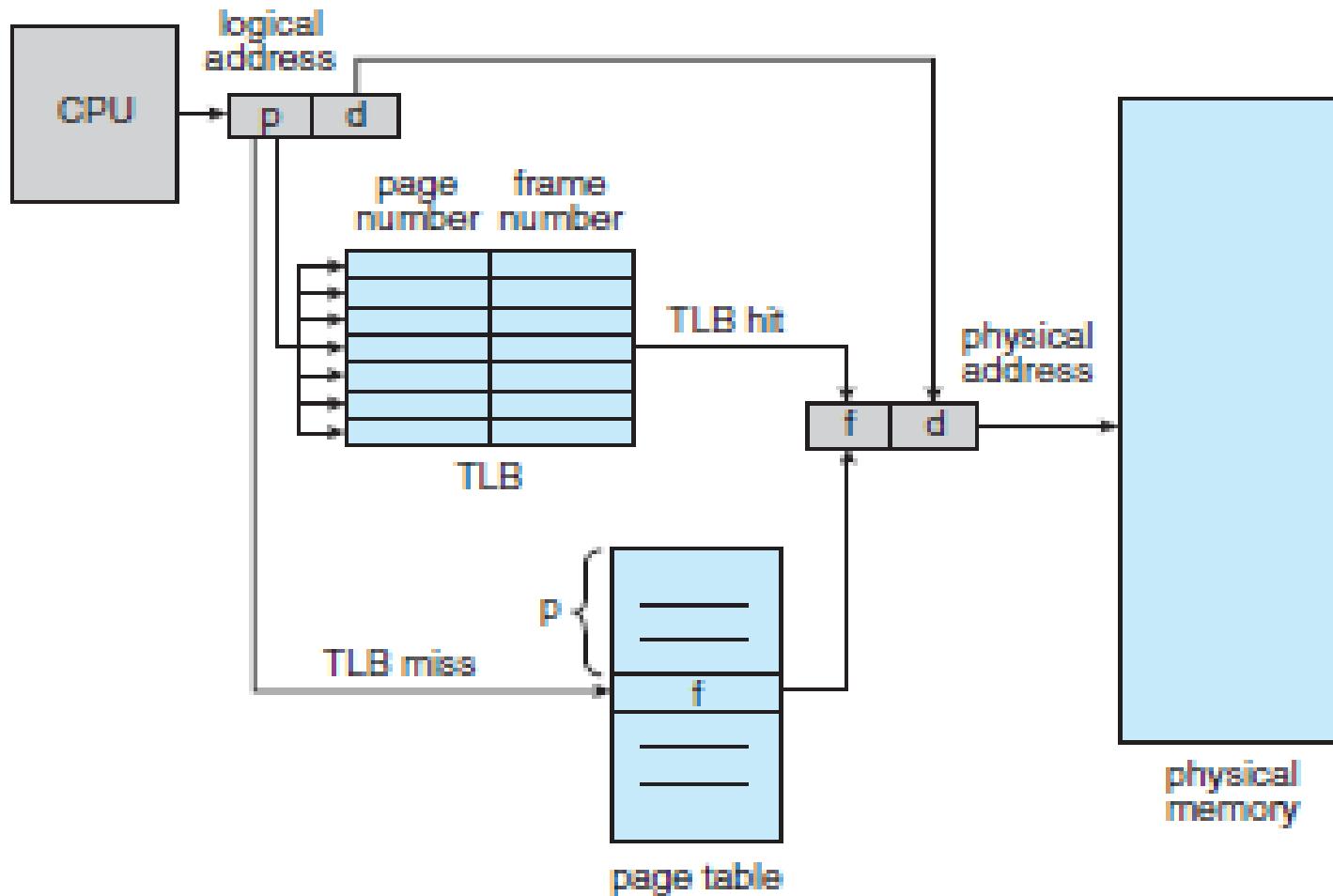


Fig: Paging hardware with TLB

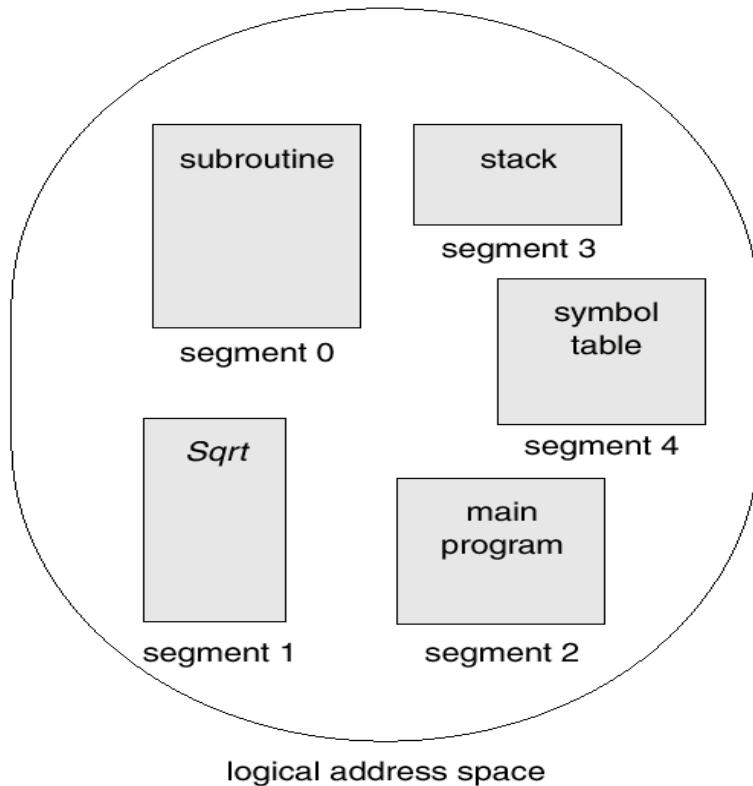
Segmentation

- **Segmentation is a 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

Segmentation Architecture

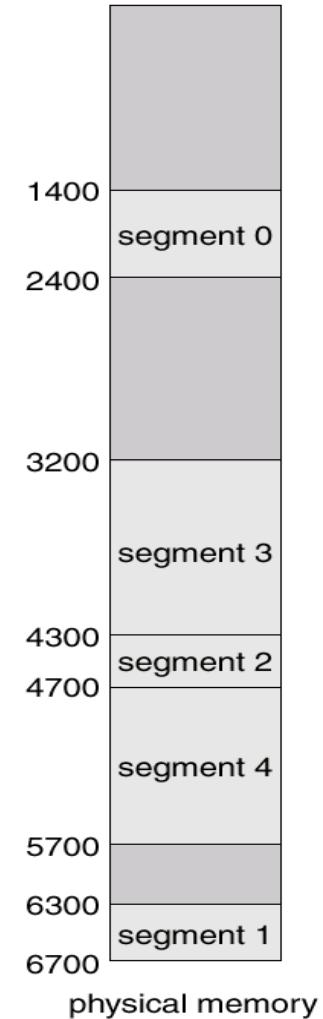
- Logical address consists of a two tuple:
 \langle segment-number, offset \rangle ,
- *Segment table – maps two-dimensional physical addresses; each table entry has:*
 - ◆ base – contains the starting physical address where the segments reside in memory.
 - ◆ *limit – specifies the length of the segment.*
- *Segment-table base register (STBR) points to the segment table's location in memory.*
- *Segment-table length register (STLR) indicates number of segments used by a program;*
segment number s is legal if s < STLR.

Example of Segmentation



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

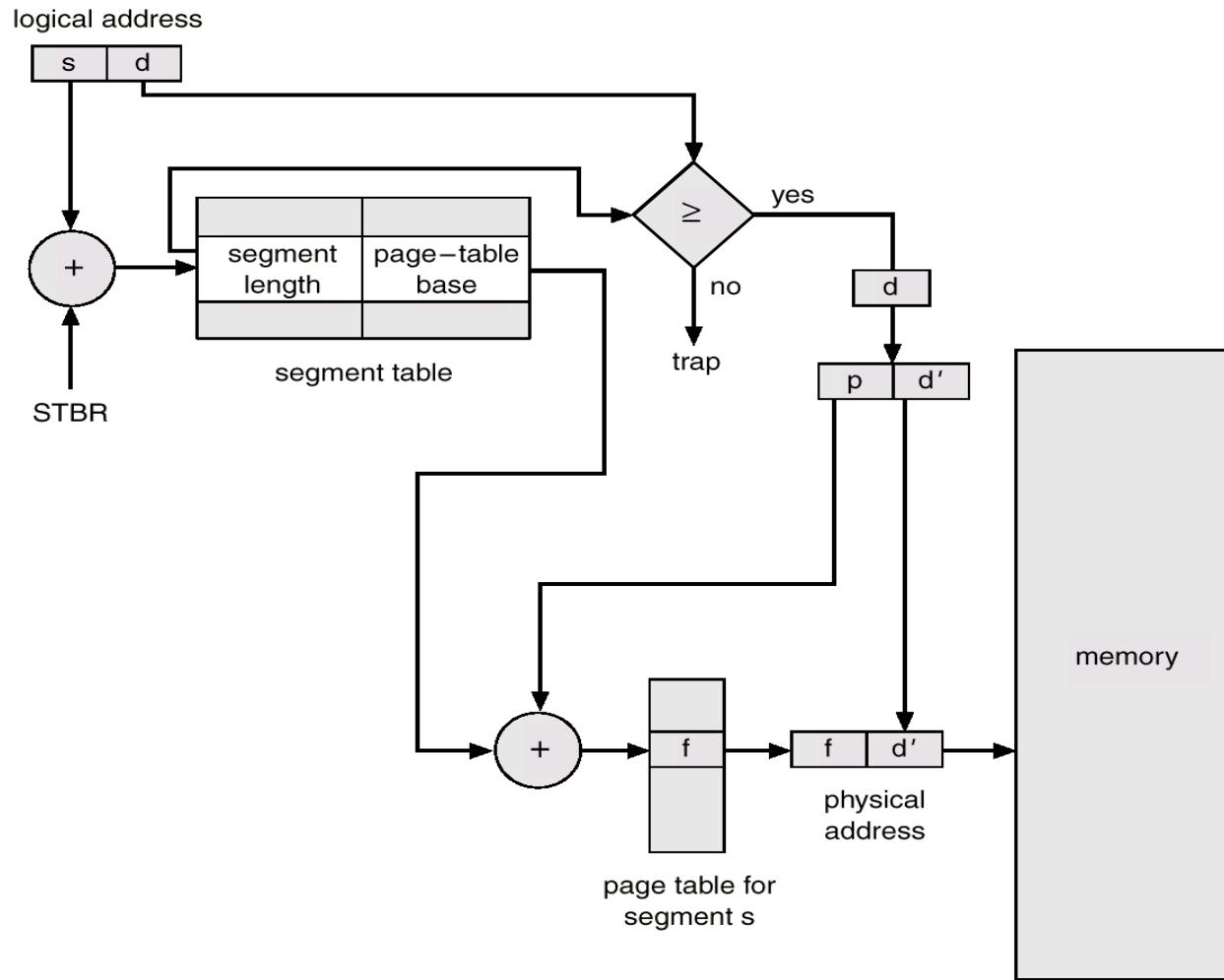
segment table



Segmentation with Paging – MULTICS

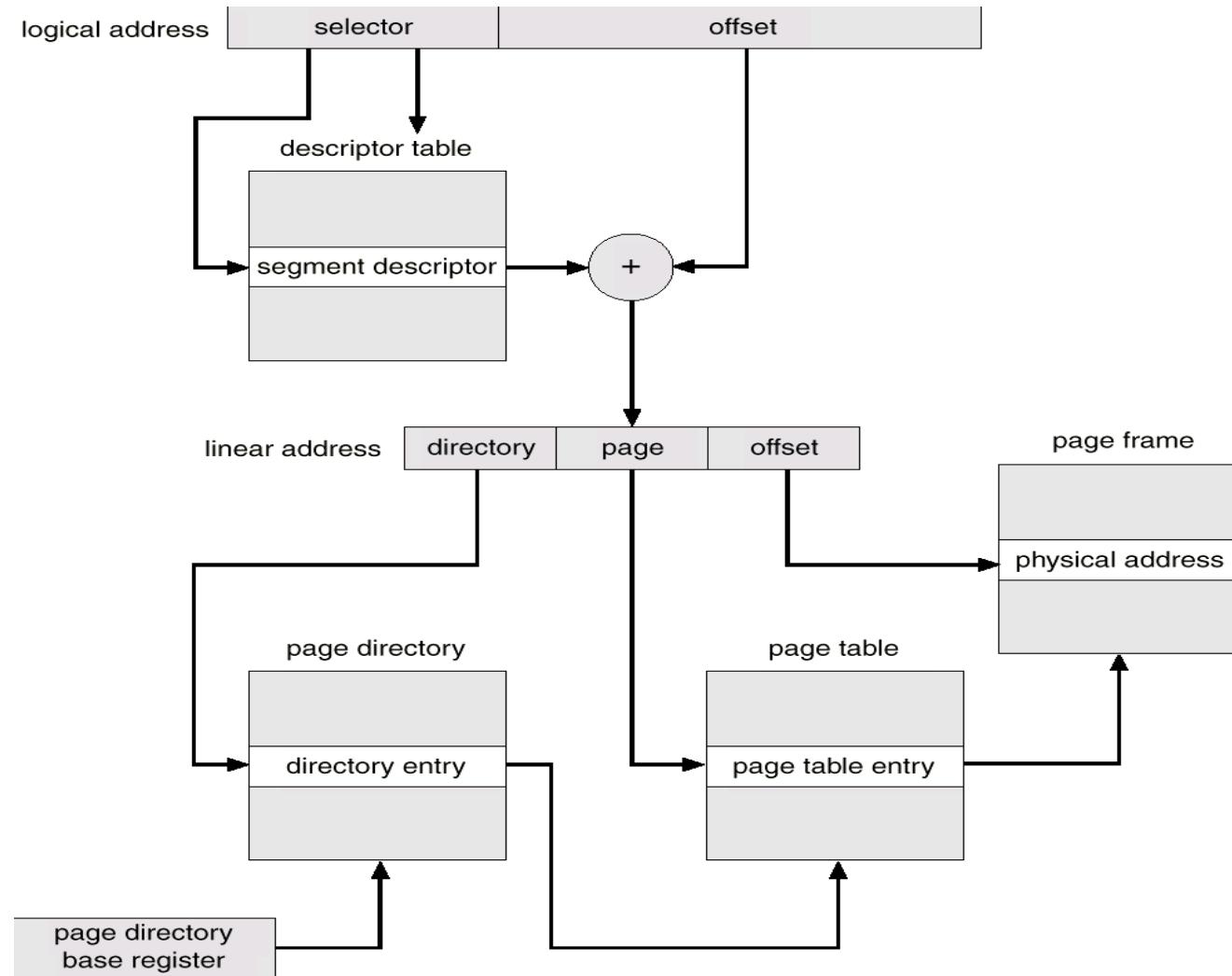
- The MULTICS system solved problems of external fragmentation and lengthy search times by paging the segments.
- Solution differs from pure segmentation in that the segment-table entry contains not the base address of the segment, but rather the base address of a *page table* for this segment.

MULTICS Address Translation Scheme



Segmentation with Paging – Intel 386

As shown in the following diagram, the Intel 386 uses segmentation with paging for memory management with a two-level paging scheme.



Relocation and Protection

- Relocation

In systems with virtual memory, programs in memory must be able to reside in different parts of the memory at different times. This is because when the program is swapped back into memory after being swapped out for a while it can not always be placed in the same location. The virtual memory management unit must also deal with concurrency. Memory management in the operating system should therefore be able to relocate programs in memory and handle memory references and addresses in the code of the program so that they always point to the right location in memory.

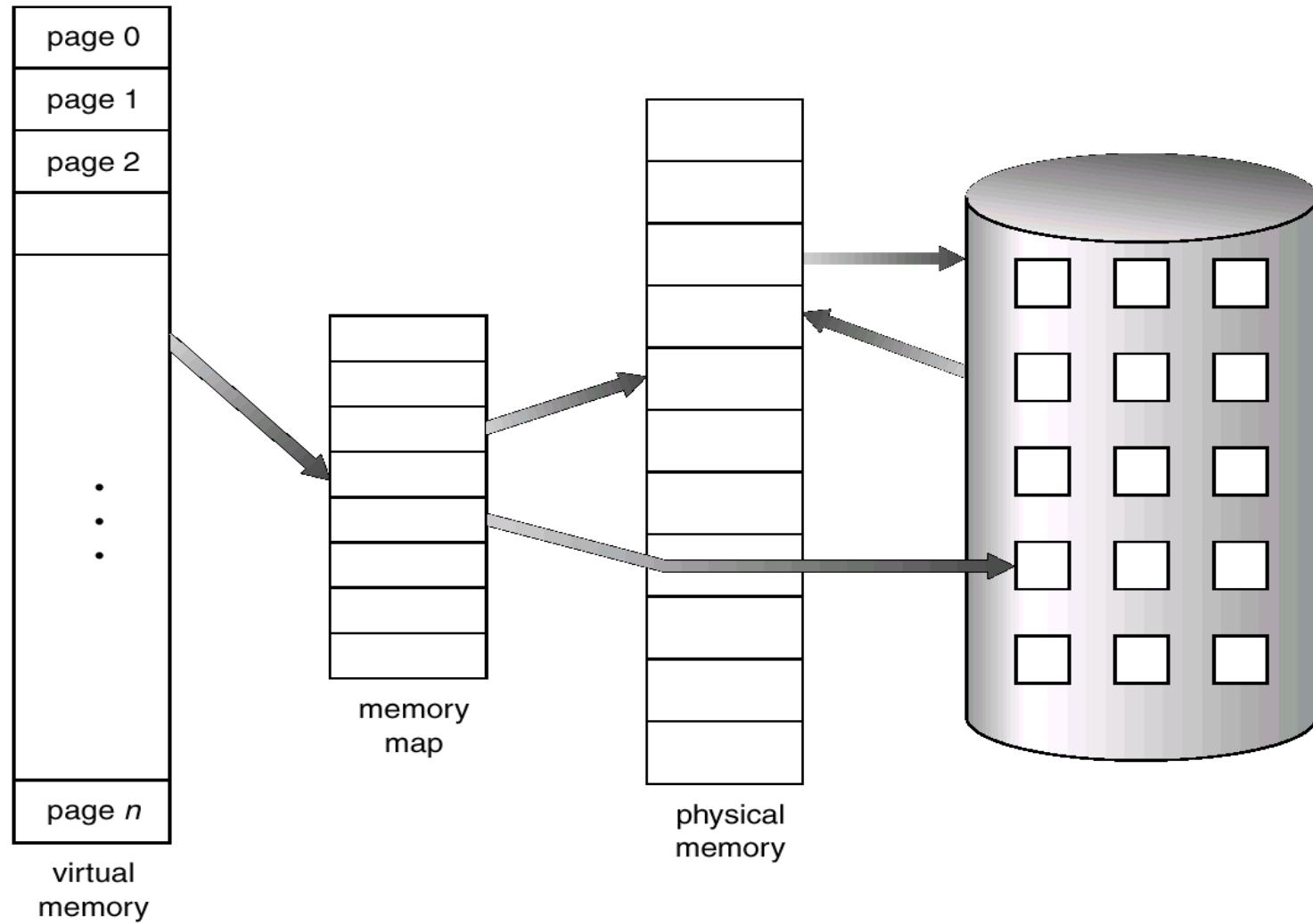
- Protection

Processes should not be able to reference the memory for another process without permission. This is called memory protection, and prevents malicious or malfunctioning code in one program from interfering with the operation of other running programs.

Virtual Memory

- Introduction:
 - Solves problem of limited memory space
 - Creates the illusion that more memory exists than is available in system
 - Two types of addresses in virtual memory systems
 - Virtual addresses
 - Referenced by processes
 - Physical addresses
 - Describes locations in main memory
 - Memory management unit (MMU)
 - Translates virtual addresses to physical address

Virtual Memory That is Larger Than Physical Memory



Virtual Memory: Basic Concepts

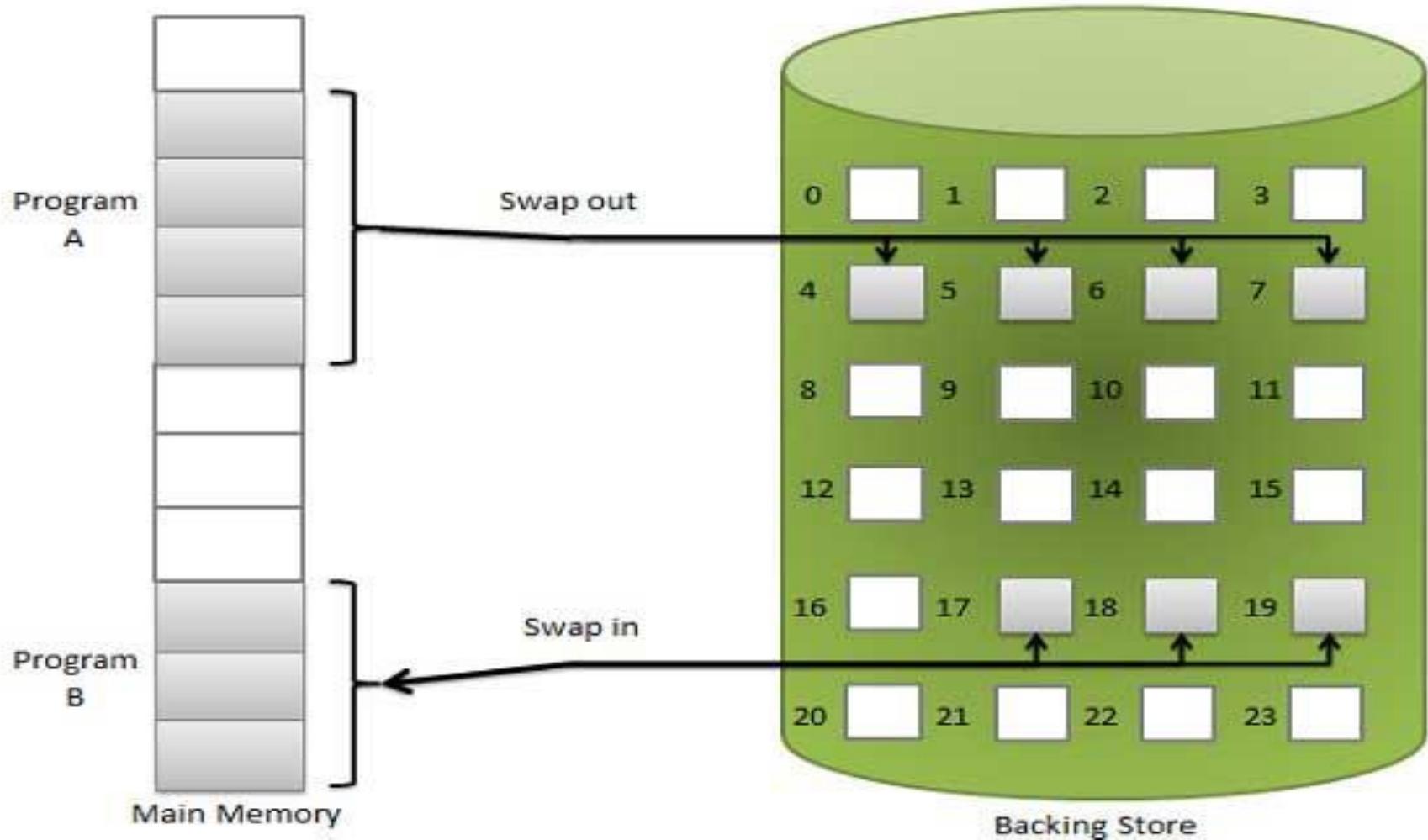
- Virtual address space, V
 - Range of virtual addresses that a process may reference
- Real address space, R
 - Range of physical addresses available on a particular computer system
- Dynamic address translation (DAT) mechanism
 - Converts virtual addresses to physical addresses during program execution
- $|V|$ is often much greater than $|R|$
 - OS must store parts of V for each process outside of main memory
- Two-level storage
 - OS shuttles portions of V between main memory (and caches) and secondary storage

Demand Paging

A demand paging system is quite similar to a paging system with swapping. When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however, we use a lazy swapper called pager.

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

Hardware support is required to distinguish between those pages that are in memory and those pages that are on the disk using the valid-invalid bit scheme. Where valid and invalid pages can be checked by checking the bit. Marking a page will have no effect if the process never attempts to access the page. While the process executes and accesses pages that are memory resident, execution proceeds normally

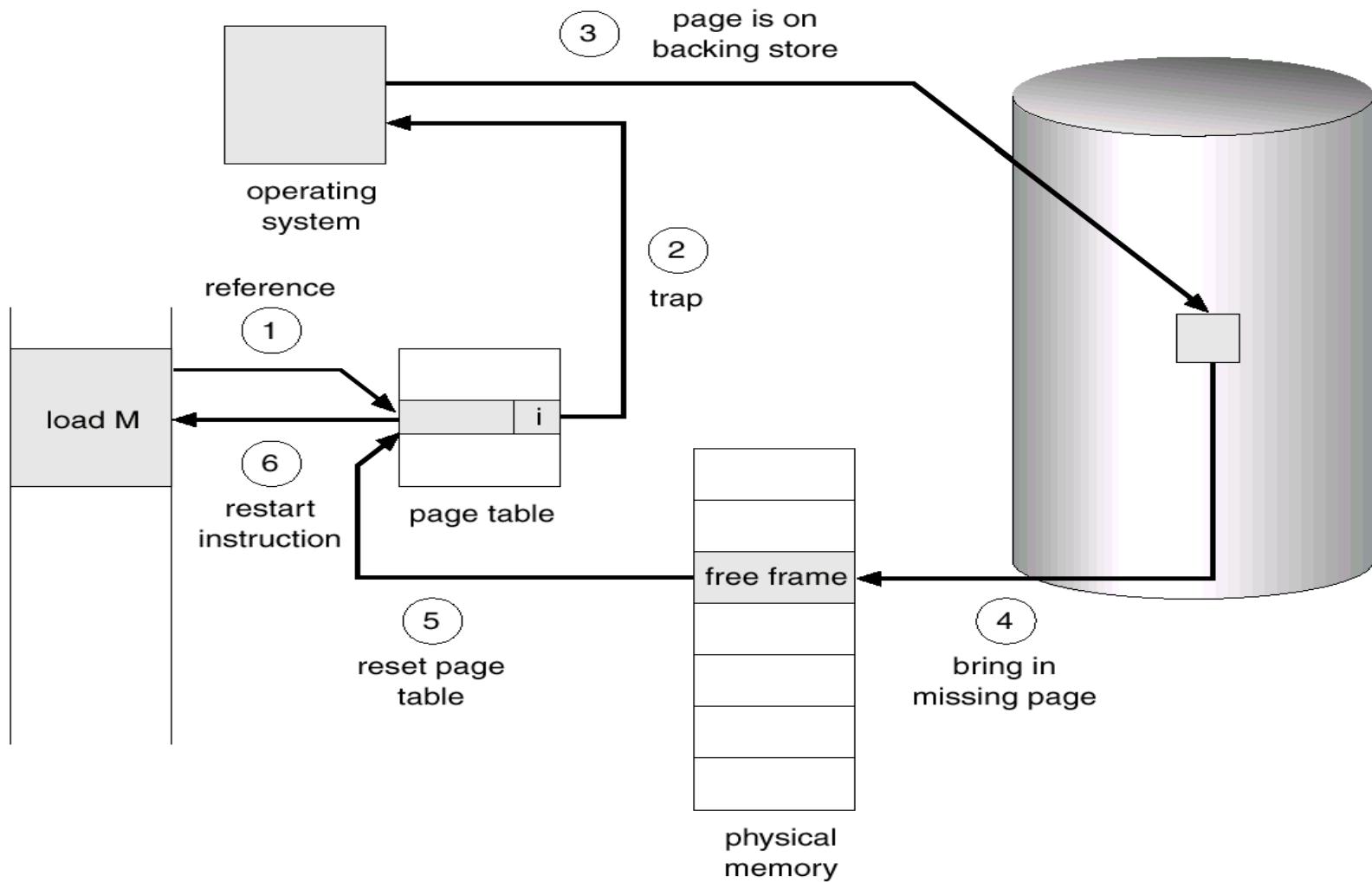


Transfer of a Paged Memory to Contiguous Disk Space

Page Fault

- If there is ever a reference to a page, first reference will trap to OS → page fault
- OS looks at another table to decide:
 - ◆ Invalid reference → abort.
 - ◆ Just not in memory.
- Get empty frame.
- Swap page into frame.
- Reset tables, validation bit = 1.
- Restart instruction: Least Recently Used
 - ◆ block move
 - ◆ auto increment/decrement location

Steps in Handling a Page Fault



Steps in Handling Page Fault:

- The procedure for handling this page fault is straightforward (Figure in previous slide):
 1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.
 2. If the reference was invalid, we terminate the process. If it was valid but we have not yet brought in that page, we now page it in.
 3. We find a free frame (by taking one from the free-frame list, for example).
 4. We schedule a disk operation to read the desired page into the newly allocated frame.
 5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
 6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

What happens if there is no free frame?

- Page replacement – find some page in memory, but not really in use, swap it out.
 - ◆ algorithm
 - ◆ performance – want an algorithm which will result in minimum number of page faults.
- Same page may be brought into memory several times.

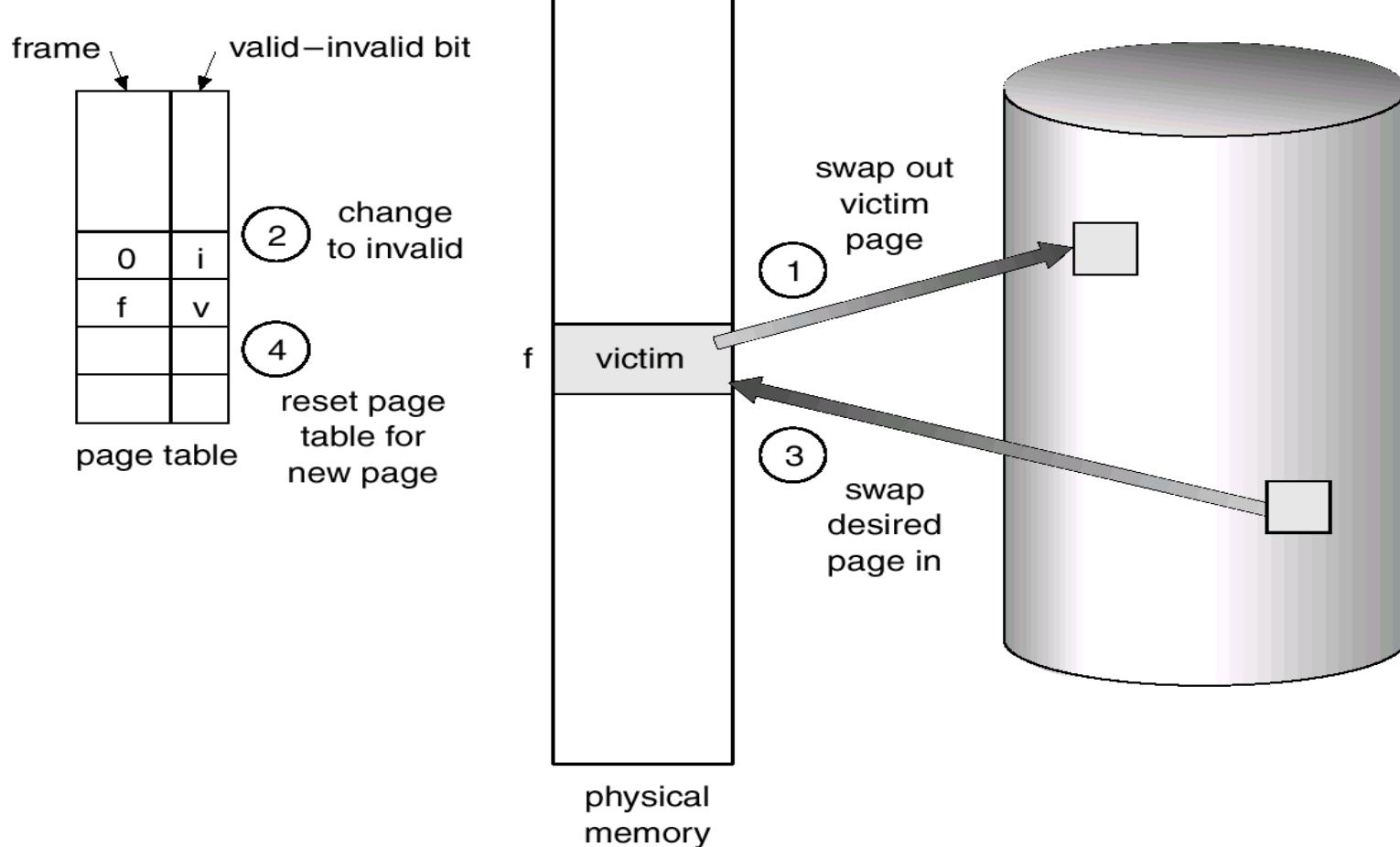
Page Replacement

- Prevent over-allocation of memory by modifying page fault service routine to include page replacement.
 - Use *modify (dirty) bit to reduce overhead of page* transfers – only modified pages are written to disk.
 - Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory.

Basic Page Replacement

- Find the location of the desired page on disk.
- Find a free frame:
 - If there is a free frame, use it.
 - If there is no free frame, use a page replacement algorithm to select a *victim frame*.
- Read the desired page into the (newly) free frame
- . Update the page and frame tables.
- Restart the process.

Page Replacement



Page Replacement Algorithms

- Want lowest page-fault rate.
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string.
- In all our examples, the reference string is
7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

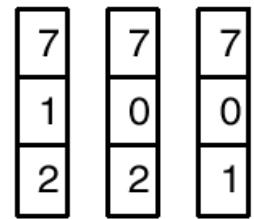
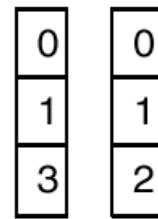
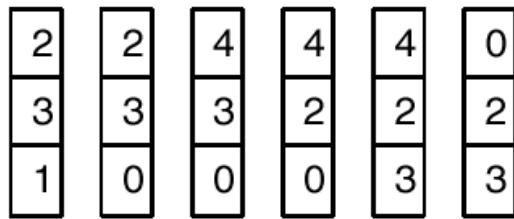
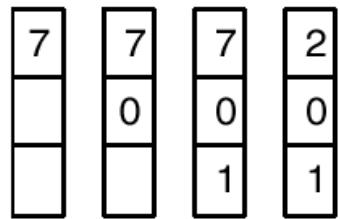
First-In-First-Out (FIFO) Page Replacement

- FIFO page replacement
 - Replace page that has been in the system the longest
 - Likely to replace heavily used pages
 - Can be implemented with relatively low overhead
 - Impractical for most systems

FIFO Page Replacement

reference string

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



page frames

3 frames are used (3 pages can be in memory at a time per process)

FIFO Page Replacement

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

FIFO Anomaly

- Belady's (or FIFO) Anomaly
 - Certain page reference patterns actually cause more page faults when number of page frames allocated to a process is increased

FIFO Anomaly

Page reference	Result	FIFO page replacement with three pages available			FIFO page replacement with four pages available			
A	Fault	A	-	-	Fault	-	-	-
B	Fault	B	A	-	Fault	A	-	-
C	Fault	C	B	A	Fault	B	A	-
D	Fault	D	C	B	Fault	C	B	A
A	Fault	A	D	C	No fault	D	C	B
B	Fault	B	A	D	No fault	D	C	B
E	Fault	E	B	A	Fault	E	D	C
A	No fault	E	B	A	Fault	A	E	D
B	No fault	E	B	A	Fault	B	A	E
C	Fault	C	E	B	Fault	C	B	A
D	Fault	D	C	E	Fault	D	C	B
E	No fault	D	C	E	Fault	E	D	C

Three "no faults"

Two "no faults"

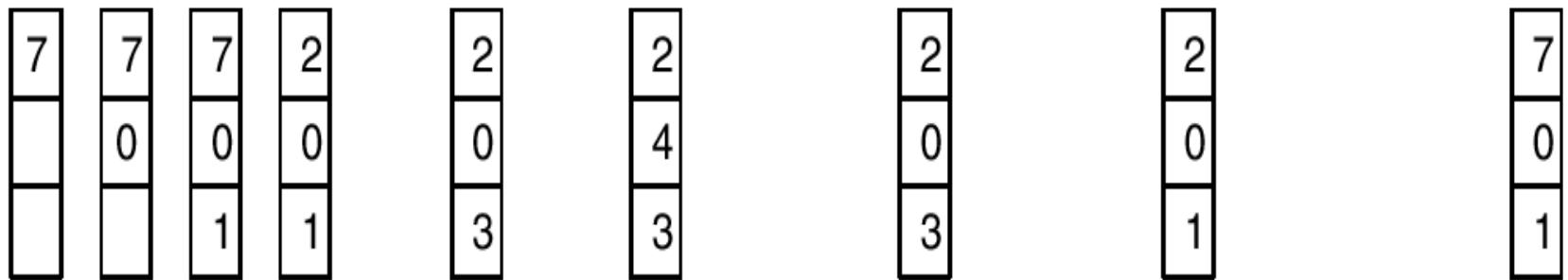
Optimal Page Replacement

- Replace page that will not be used for longest period of time.
- Use of this page replacement algorithm guarantees the lowest possible page rate for a fixed number of frames.
- How do you know this?
- Used for measuring how well your algorithm performs.

Optimal Page Replacement

reference string

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



page frames

Optimal Page Replacement

- For example, on our sample reference string, the optimal page-replacement algorithm would yield nine page faults, as shown in Figure of previous slide. The first three references cause faults that fill the three empty frames. The reference to page 2 replaces page 7, because 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than a FIFO algorithm, which had 15 faults.
- Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

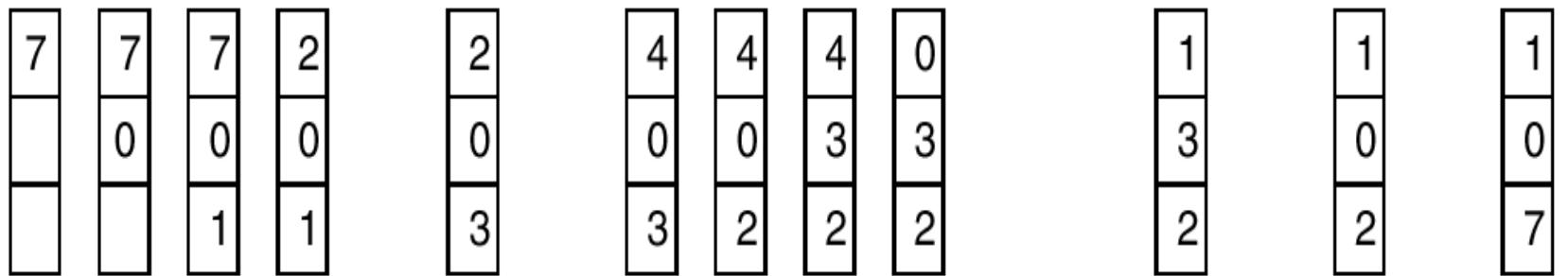
Least Recently Used (LRU) Algorithm

- LRU page replacement
 - Exploits temporal locality by replacing the page that has spent the longest time in memory without being referenced
 - Can provide better performance than FIFO
 - Increased system overhead
 - LRU can perform poorly if the least-recently used page is the next page to be referenced by a program that is iterating inside a loop that references several pages

LRU Page Replacement

reference string

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



page frames

LRU Page Replacement

- The result of applying LRU replacement to our example reference string is shown in Figure of previous slide. The LRU algorithm produces 12 faults. Notice that the first five faults are the same as the optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. The most recently used page is page 0, and just before that page **3 was used. Thus, the LRU algorithm replaces page 2**, not knowing that page 2 is about to be used. When it then faults for page 2, the LRU algorithm replaces page **3 since, of the three pages in memory {0,3,4}, page 3 is the least recently used. Despite these problems, LRU replacement with 12 faults is still much better than FIFO replacement with 15.**

LRU Algorithm (Cont.)

- Counter implementation
 - ◆ Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter.
 - ◆ When a page needs to be changed, look at the counters to determine which are to change.
- Stack implementation – keep a stack of page numbers in a double link form:
- ◆ Page referenced:
 - ✓ move it to the top. So the most recently used page is always at the top of the stack and the least recently used page is always at the bottom.
 - ◆ No search for replacement

Not Recently Used Page Replacement Algorithm

- Approximates LRU with little overhead by using referenced bit and modified bit to determine which page has not been used recently and can be replaced quickly
 - Can be implemented on machines that lack hardware referenced bit and/or modified bit

<i>Group</i>	<i>Referenced</i>	<i>Modified</i>	<i>Description</i>
Group 1	0	0	Best choice to replace
Group 2	0	1	[Seems unrealistic]
Group 3	1	0	
Group 4	1	1	Worst choice to replace

Figure : Page types under NRU.

Second-Chance Page Replacement Algorithm

- The basic algorithm of second-chance replacement is a FIFO replacement algorithm. When a page has been selected, however, we inspect its reference bit. If the value is 0, we proceed to replace this page. If the reference bit is set to 1, however, we give that page a second chance and move on to select the next FIFO page. When a page gets a second chance, its reference bit is cleared and its arrival time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages are replaced (or given second chances). In addition, if a page is used often enough to keep its reference bit set, it will never be replaced.

- **Clock page replacement**

- Similar to second chance, but arranges the pages in circular list instead of linear list

Working Set Page Replacement Algorithm

- The set of pages that a process is currently using is known as its working set. If the entire working set is in memory, the process will run without causing many faults until it moves into another execution phase. If the available memory is too small to hold the entire working set, the process will cause many page faults and run slowly.
- To implement the working set model, it is necessary for the operating system to keep track of which pages are in working set. Having this information also immediately leads to a possible page replacement algorithm: when a page fault occurs, find a page not in the working set and evict it.

Allocation Of Frames

.There were two important tasks in virtual memory management: a page-replacement strategy and a frame-allocation strategy.

Minimum Number of Frames

.The absolute minimum number of frames that a process must be allocated is dependent on system architecture, and corresponds to the worst-case scenario of the number of pages that could be touched by a single (machine) instruction. If an instruction (and its operands) spans a page boundary, then multiple pages could be needed just for the instruction fetch.

Memory references in an instruction touch more pages, and if those memory locations can span page boundaries, then multiple pages could be needed for operand access also.

.The worst case involves indirect addressing, particularly where multiple levels of indirect addressing are allowed. Left unchecked, a pointer to a pointer to a pointer to a pointer to a . . . could theoretically touch every page in the virtual address space in a single machine instruction, requiring every virtual page be loaded into physical memory simultaneously. For this reason architectures place a limit (say 16) on the number of levels of indirection allowed in an instruction, which is enforced with a counter initialized to the limit and decremented with every level of indirection in an instruction - If the counter reaches zero, then an "excessive indirection" trap occurs . This example would still require a minimum frame allocation of 17 per process.

Allocation of frames(contd..)

Allocation Algorithms

.Equal Allocation - If there are m frames available and n processes to share them, each process gets m / n frames, and the leftovers are kept in a free-frame buffer pool.

.Proportional Allocation - Allocate the frames proportionally to the size of the Process relative to the total size of all processes. So if the size of process i is S_i , and S is the sum of all S_i , then the allocation for process P_i is $a_i = m * S_i / S$. Variations on proportional allocation could consider priority of process rather than just their size.

Global versus Local Allocation

.One big question is whether frame allocation (page replacement) occurs on a local or global level.

.With local replacement, the number of pages allocated to a process is fixed, and page replacement occurs only amongst the pages allocated to this process.

With global replacement, any page may be a potential victim, whether it currently belongs to the process seeking a free frame or not.

.Local page replacement allows processes to better control their own page fault rates, and leads to more consistent performance of a given process over different system load levels. Global page replacement is overall more efficient, and is the more commonly used approach.

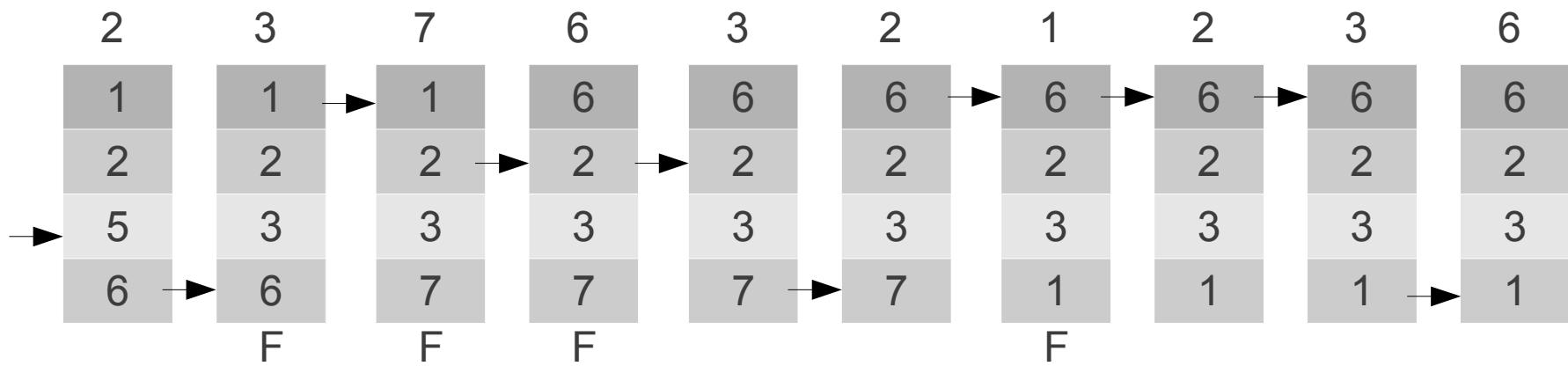
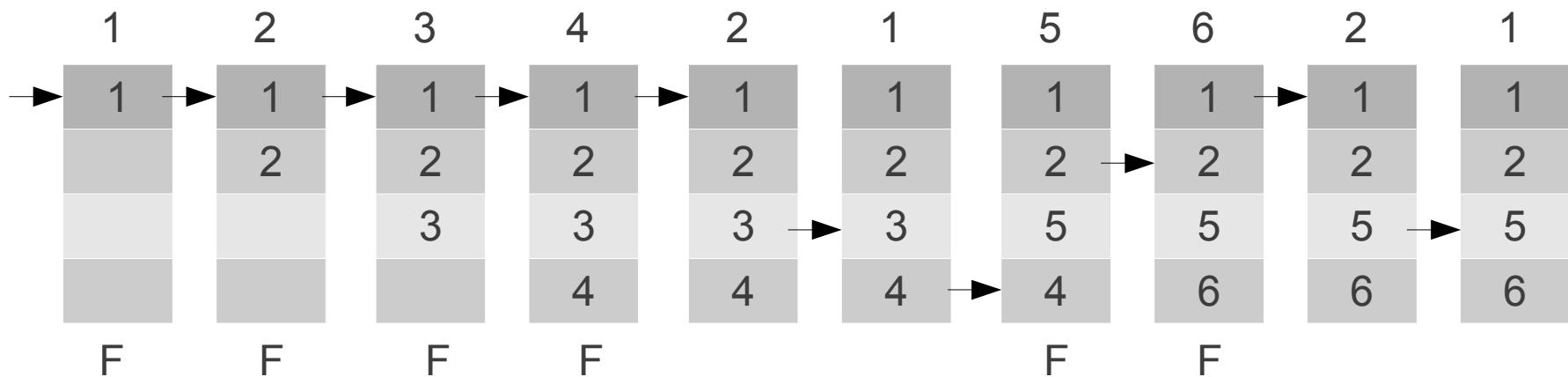
Thrashing

- If the process does not have sufficient number of frames for the pages in active use, it will quickly page fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again. The process continues to fault, replacing pages for which it then faults and brings back in right away.
- This high paging activity is called **thrashing**. A process is **thrashing if it is** spending more time paging than executing.

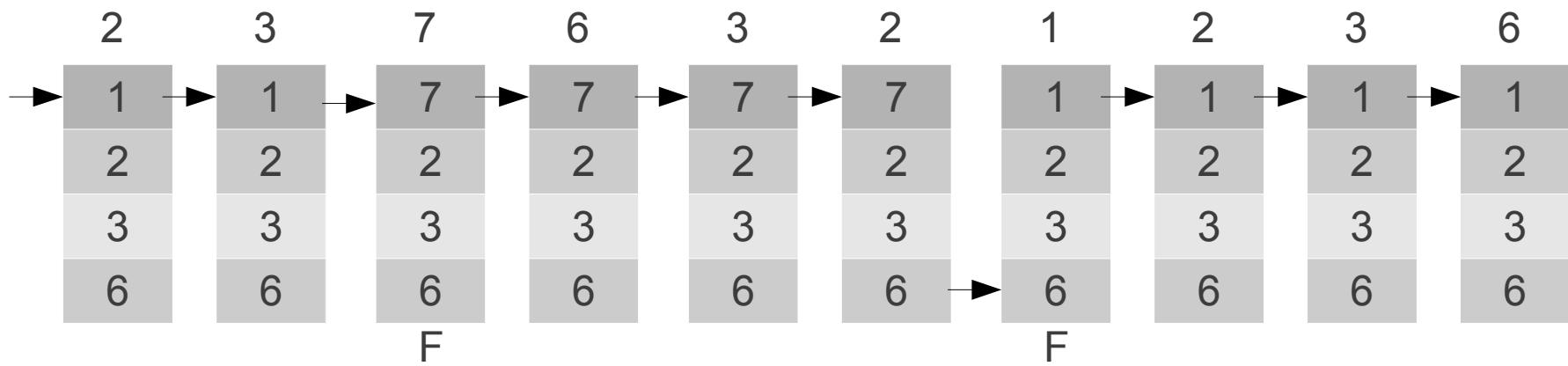
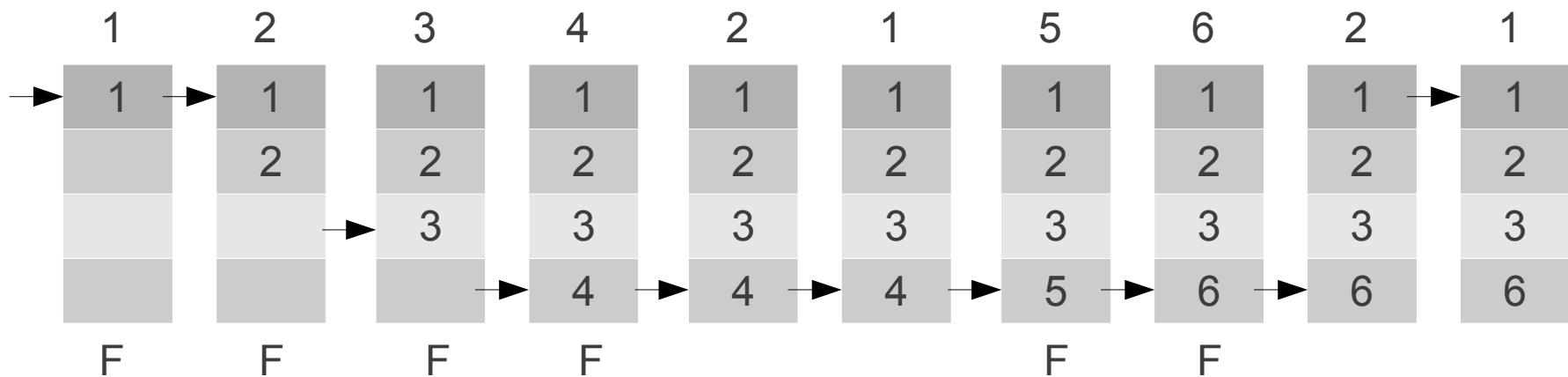
Question – Page replacement algorithm

- Given page reference string:
 - 1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6
- Compare the number of page faults for LRU, FIFO and Optimal page replacement algorithm

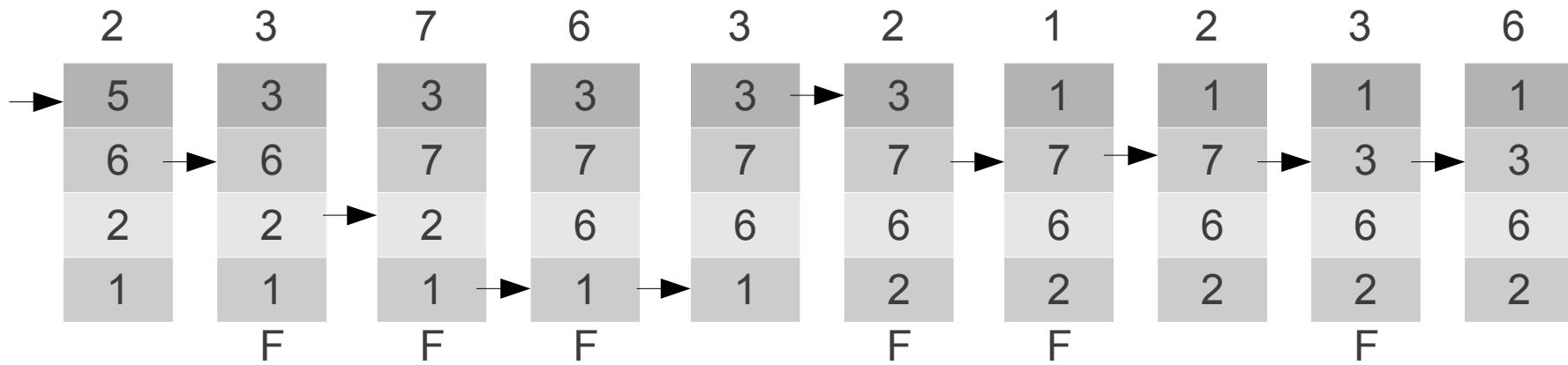
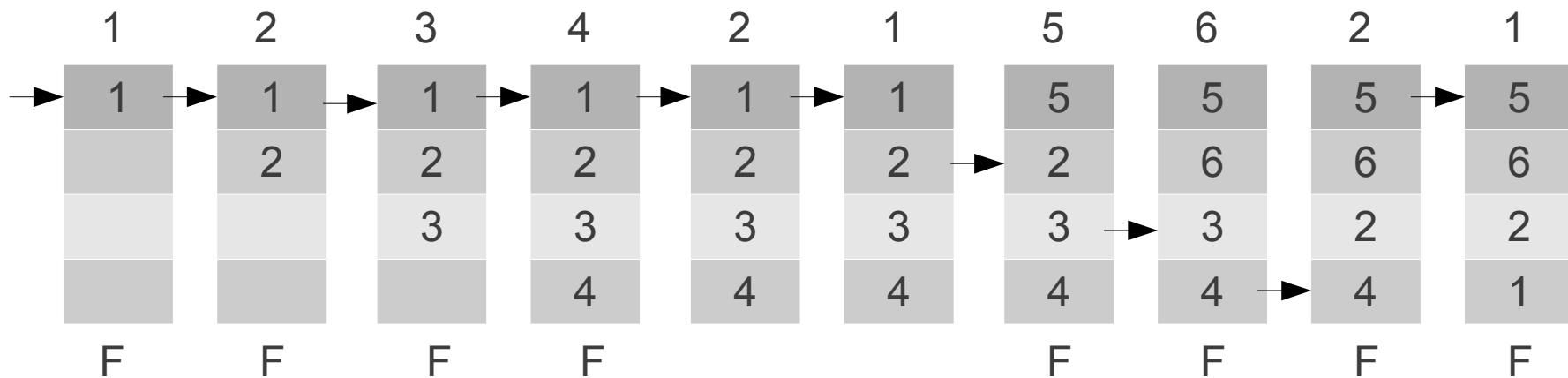
Question : 4 Frames, LRU



Question : 4 Frames, OPT

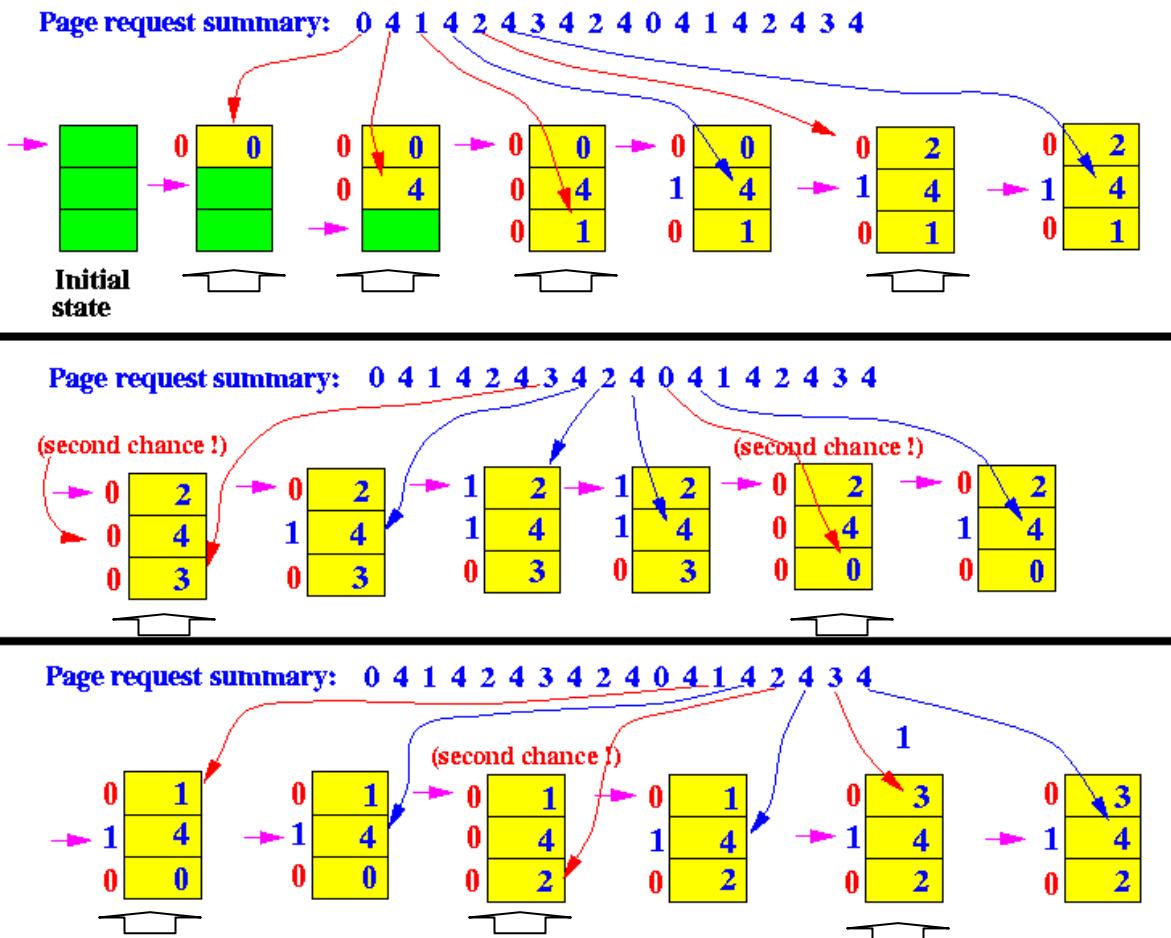


Question : 4 Frames, FIFO



Second Chance page Replacement Algorithm:

The following figure shows the behavior of the program in paging using the Second Chance page replacement policy:



Total Page fault=9

Fault rate/ratio= 9\18=0.50

Applied Operating System

Chapter 4: I/O Management

Prepared By:
Amit K. Shrivastava
Asst. Professor
Nepal College Of Information Technology

4.1 I/O Sub- Systems

4.1.1 Concepts

- Management of I/O devices is a very important part of the operating system - so important and so varied that entire I/O subsystems are devoted to its operation. (Consider the range of devices on a modern computer, from mice, keyboards, disk drives, display adapters, USB devices, network connections, audio I/O, printers, special devices for the handicapped, and many special-purpose peripherals.)
- I/O Subsystems must contend with two trends: (1) The gravitation towards standard interfaces for a wide range of devices, making it easier to add newly developed devices to existing systems, and (2) the development of entirely new types of devices, for which the existing standard interfaces are not always easy to apply.
- Device drivers are modules that can be plugged into an OS to handle a particular device or category of similar devices.

4.1.2 Application I/O Interface

- User application access to a wide variety of different devices is accomplished through layering, and through encapsulating all of the device-specific code into **device drivers**, while application layers are presented with a common interface for all (or at least large general categories of) devices.

Application I/O Interface(contd...)

- Device-driver layer hides differences among I/O controllers from kernel. New devices talking already-implemented protocols need no extra work. Each OS has its own I/O subsystem structures and device driver frameworks.
- Devices vary in many dimensions
 - Character-stream or block: A character-stream device transfers bytes one by one, whereas a block device transfers a block of bytes as a unit.
 - Sequential or random-access: A sequential device transfers data in a fixed order that is determined by the device, whereas the user of random-access device can instruct the device to seek to any of the available data storage locations.
 - Synchronous or asynchronous: A synchronous device is one that performs data transfers with predictable response time. An asynchronous device exhibits irregular response time
 - Sharable or dedicated: A shareable device can be used concurrently by several processes or threads; a dedicated device cannot be.
 - Speed of operation: Device speeds range from few bytes to few gigabytes per second.
 - Read-write, read only, or write only: Some devices perform both input and output, whereas others support only one data direction.

software

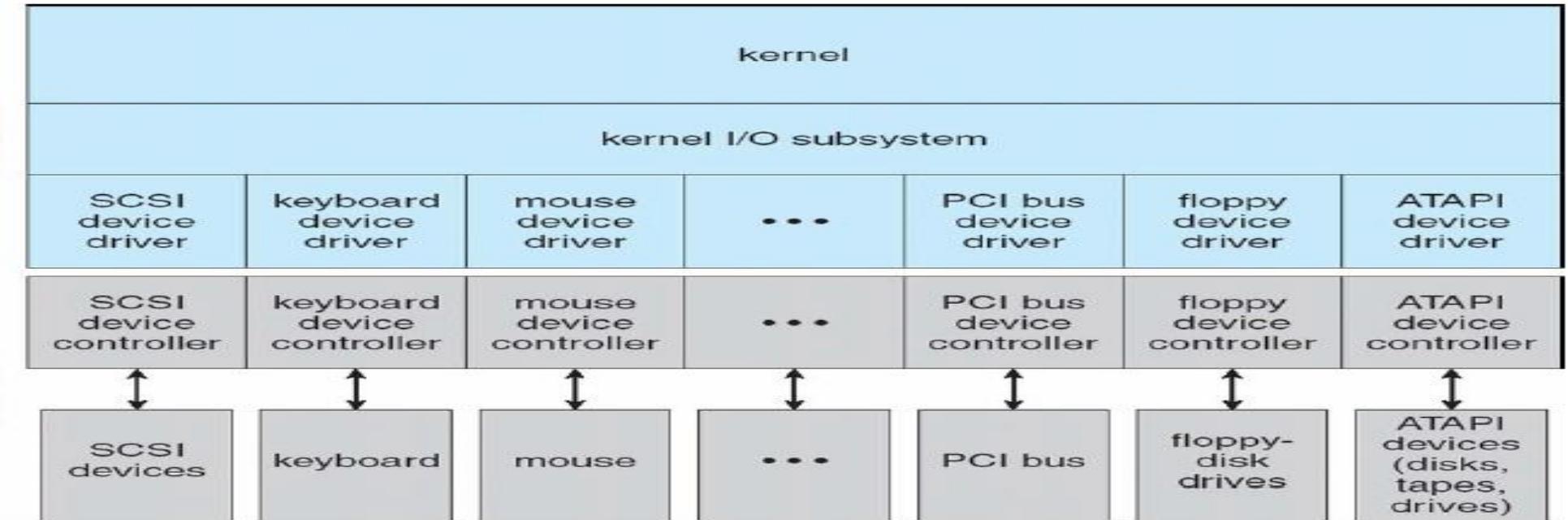


Fig: A kernel I/O structure

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read-write	CD-ROM graphics controller disk

Fig: Characteristics of I/O devices

Characteristics of I/O Devices (Cont.)

- Most devices can be characterized as either block I/O, character I/O, memory mapped file access, or network sockets. A few devices are special, such as time-of-day clock and the system timer.
- Most OSes also have an escape, or back door, which allows applications to send commands directly to device drivers if needed. In UNIX this is the **ioctl()** system call (for I/O Control).

Block and Character Devices

- Block devices include disk drives
 - Commands include read, write, seek
 - **Raw I/O, direct I/O, or file-system access**
 - Memory-mapped file access possible
 - File mapped to virtual memory and clusters brought via demand paging
 - DMA
- Character devices include keyboards, mice, serial ports
 - Commands include get(), put()
 - Libraries layered on top allow line editing

Network Devices

- Varying enough from block and character to have own interface
- Unix and Windows NT/9x/2000 include socket interface
 - Separates network protocol from network operation
 - Includes select() functionality
- Approaches vary widely (pipes, FIFOs, streams, queues, mailboxes)

Clocks and Timers

- Provide current time, elapsed time, timer
- Normal resolution about 1/60 second
- Some systems provide higher-resolution timers
- Programmable interval timer used for timings, periodic interrupts
- *ioctl()* (on UNIX) covers odd aspects of I/O such as clocks and timers

Blocking and Nonblocking I/O

- With **blocking I/O** a process is moved to the wait queue when an I/O request is made, and moved back to the ready queue when the request completes, allowing other processes to run in the meantime. It is easy to use and understand but insufficient for some needs.
- With **non-blocking I/O** the I/O request returns immediately, whether the requested I/O operation has (completely) occurred or not. This allows the process to check for available data without getting hung completely if it is not there.
- Asynchronous - process runs while I/O executes
 - Difficult to use
 - I/O subsystem signals process when I/O completed

Ways To Input/Output:

There are three fundamental ways to do input/output

- 1.Programmed I/O
- 2.Interrupt-Driven
- 3.DMA(Direct Memory Access)

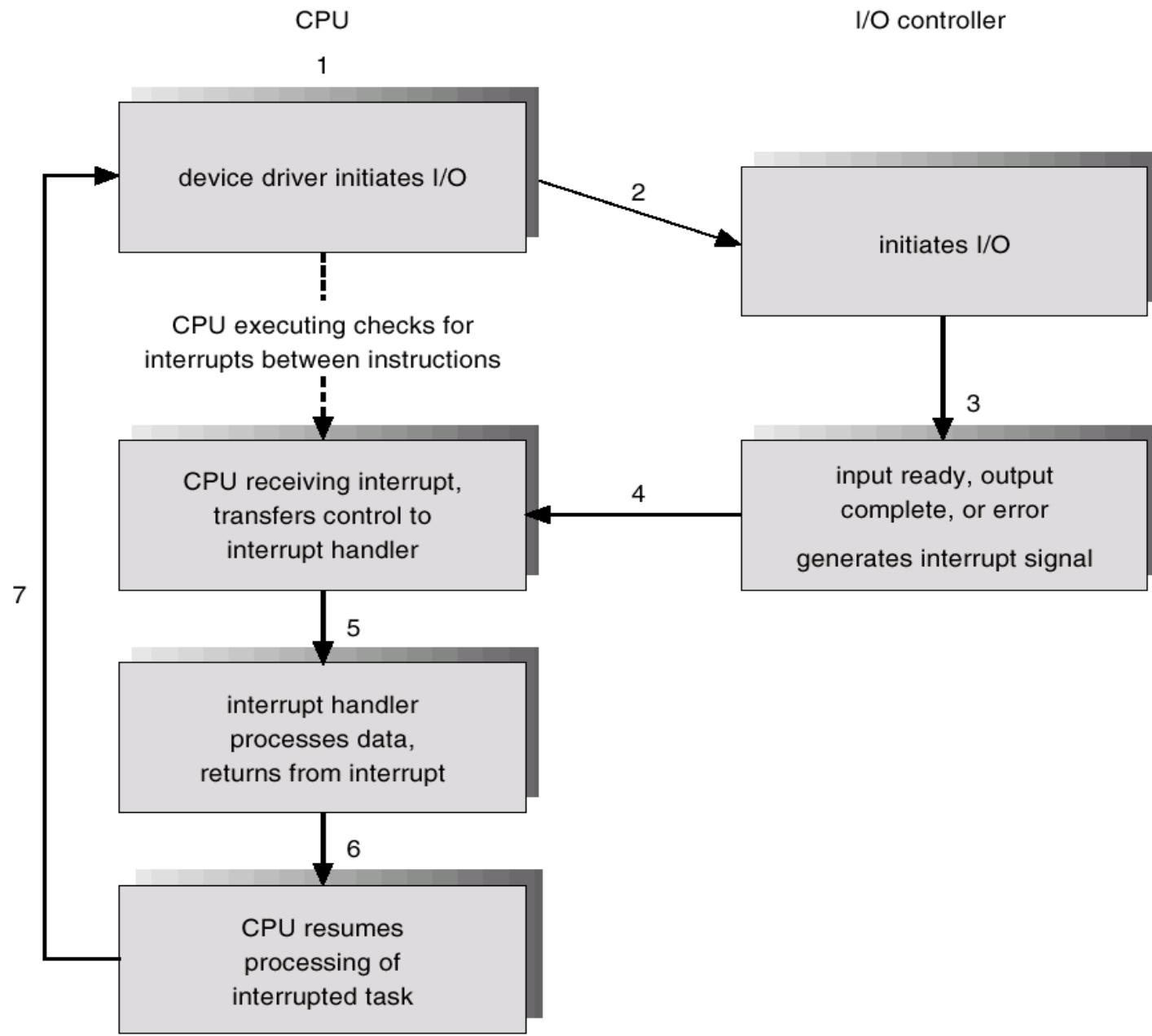
Programmed I/O:

The simplest form of I/O is to have the CPU do all the work. This method is called **programmed I/O**. The actions followed by operating system are summarized in the following manner. First the data are copied to the kernel. Then the operating system enters a tight loop outputting the characters one at a time. The essential aspect of programmed I/O is that after outputting a character, the CPU continuously polls the device to see if it is ready to accept another one. This behavior is often called **polling or busy waiting**.

Interrupt - Driven I/O

- The basic interrupt mechanism works as follows. The CPU hardware has a wire called the interrupt-request line that the CPU senses after executing every instruction. When the CPU detects that a controller has asserted a signal on the interrupt request line, the CPU saves a small amount of state, such as the current value of the instruction pointer, and jumps to the interrupt-handler routine at a fixed address in memory. The interrupt handler determines the cause of the interrupt, performs the necessary processing, and executes a return from interrupt instruction to return the CPU to the execution state prior to the interrupt. We say that the device controller *raises an interrupt by asserting* a signal on the interrupt request line, the CPU *catches the interrupt and dispatches* to the interrupt handler, and the handler *clears the interrupt by servicing the device*.

Interrupt-Driven I/O Cycle

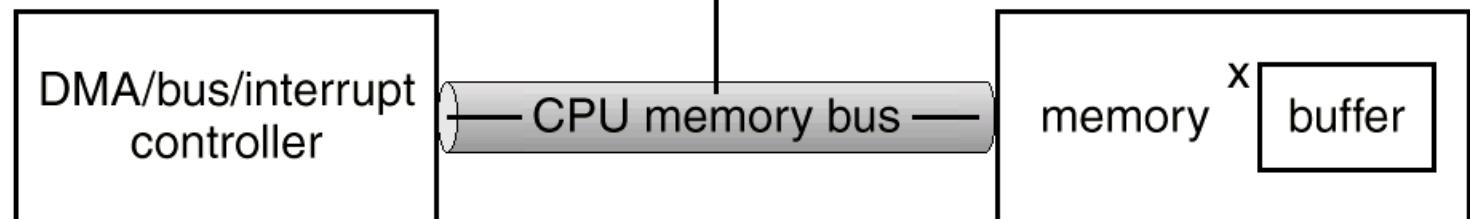


DMA(Direct Memory Access)

- Many computers avoid burdening the main CPU with **PIO** by offloading some of this work to a special-purpose processor called a directmemory-access (**DMA**) controller. To initiate a DMA transfer, the host writes a DMA command block into memory. This block contains a pointer to the source of a transfer, a pointer to the destination of the transfer, and a count of the number of bytes to be transferred.
- The CPU writes the address of this command block to the DMA controller, then goes on with other work. The DMA controller proceeds to operate the memory bus directly, placing addresses on the bus to perform transfers without the help of the main CPU. A simple DMA controller is a standard component in PCs, and bus-mastering I/O boards for the PC usually contain their own high-speed DMA hardware.

5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0
6. when C = 0, DMA interrupts CPU to signal transfer completion

1. device driver is told to transfer disk data to buffer at address X
2. device driver tells disk controller to transfer C bytes from disk to buffer at address X



3. disk controller initiates DMA transfer
4. disk controller sends each byte to DMA controller

4.1.3 Kernel I/O Subsystem

▪ Kernel provide many services related to I/O. The services that we describe are I/O scheduling, buffering, caching, spooling, device reservation, and error handling.

▪ **Scheduling**

- Some I/O request ordering via per-device queue
- Some OSs try fairness
- Some implement Quality Of Service (i.e. IPQOS)

▪ **Buffering** - store data in memory while transferring between devices

- To cope with device speed mismatch
- To cope with device transfer size mismatch
- To maintain “copy semantics”
- Double buffering – two copies of the data
 - Kernel and user
 - Varying sizes
 - Full / being processed and not-full / being used
 - Copy-on-write can be used for efficiency in some cases

▪ **Caching** - faster device holding copy of data

- Always just a copy
- Key to performance
- Sometimes combined with buffering

▪ **Spooling** - hold output for a device

- If device can serve only one request at a time
- i.e., Printing

Kernel I/O Subsystem(contd..)

- **Device reservation** - provides exclusive access to a device
 - System calls for allocation and de-allocation
 - Watch out for deadlock

- **Error Handling** - OS can recover from disk read, device unavailable, transient write failures
 - Retry a read or write, for example
 - Some systems more advanced – Solaris FMA, AIX
 - Track error frequencies, stop using device with increasing frequency of retry-able errors

- Most return an error number or code when I/O request fails
- System error logs hold problem reports

I/O Protection

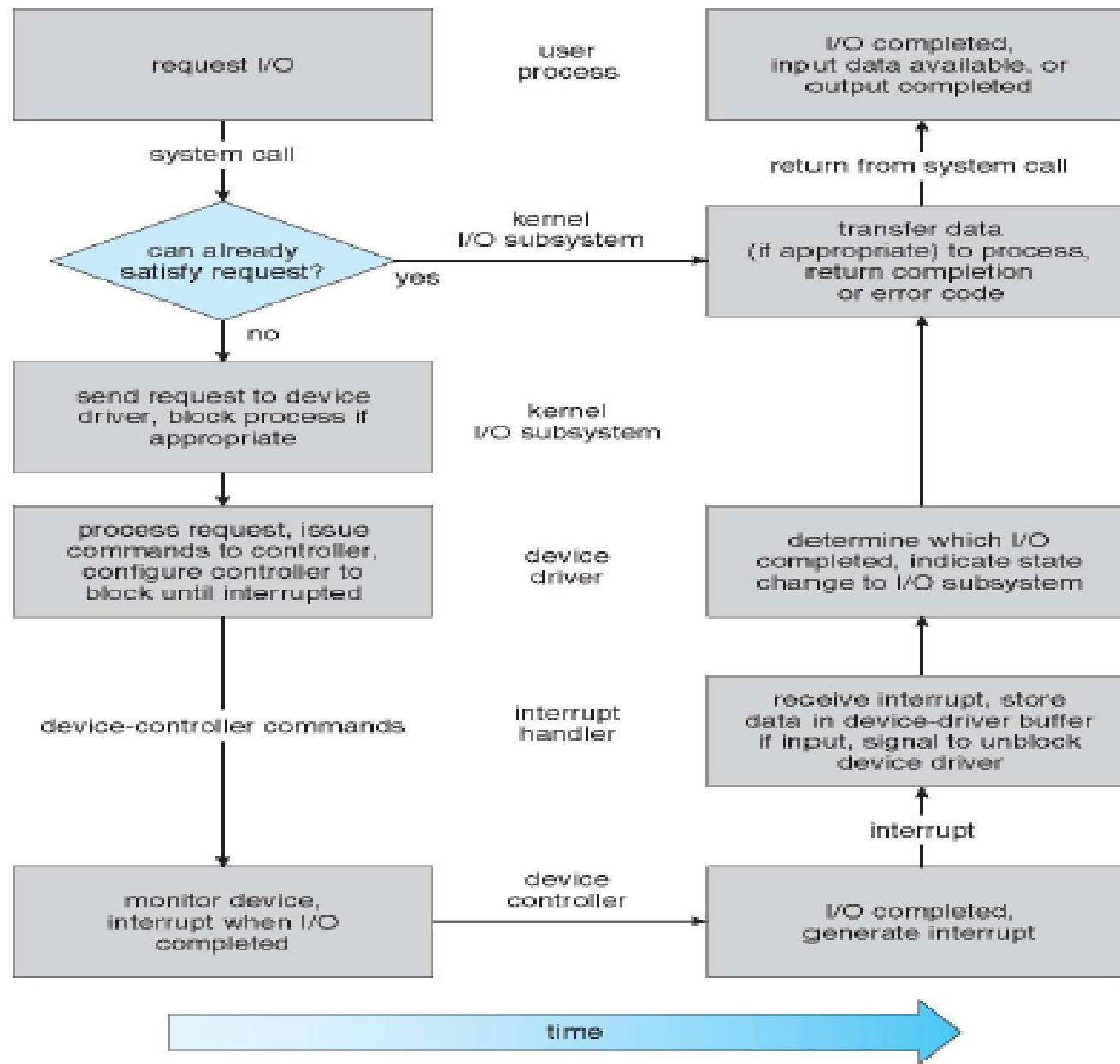
User process may accidentally or purposefully attempt to disrupt normal operation via illegal I/O instructions

- All I/O instructions defined to be privileged
- I/O must be performed via system calls
 - Memory-mapped and I/O port memory locations must be protected too

4.1.4 I/O Requests Handling

- Users request data using file names, which must ultimately be mapped to specific blocks of data from a specific device managed by a specific device driver.
- Consider reading a file from disk for a process
 - Determine device holding file
 - Translate name to device representation
 - Physically read data from disk into buffer
 - Make data available to requesting process
 - Return control to process

Life Cycle of An I/O Request



4.1.5 Performance

- The I/O system is a major factor in overall system performance, and can place heavy loads on other major components of the system (interrupt handling, process switching, memory access, bus contention, and CPU load for device drivers just to name a few.)
- Interrupt handling can be relatively expensive (slow), which causes programmed I/O to be faster than interrupt-driven I/O when the time spent busy waiting is not excessive.
- Network traffic can also put a heavy load on the system. Consider for example the sequence of events that occur when a single character is typed in a telnet session, as shown in figure 1.
- Several principles can be employed to increase the overall efficiency of I/O processing:
 - Reduce the number of context switches.
 - Reduce the number of times data must be copied.
 - Reduce interrupt frequency, using large transfers, buffering, and polling where appropriate.
 - Increase concurrency using DMA.
 - Move processing primitives into hardware, allowing their operation to be concurrent with CPU and bus operations.
 - Balance CPU, memory, bus, and I/O operations, so a bottleneck in one does not idle all the others.

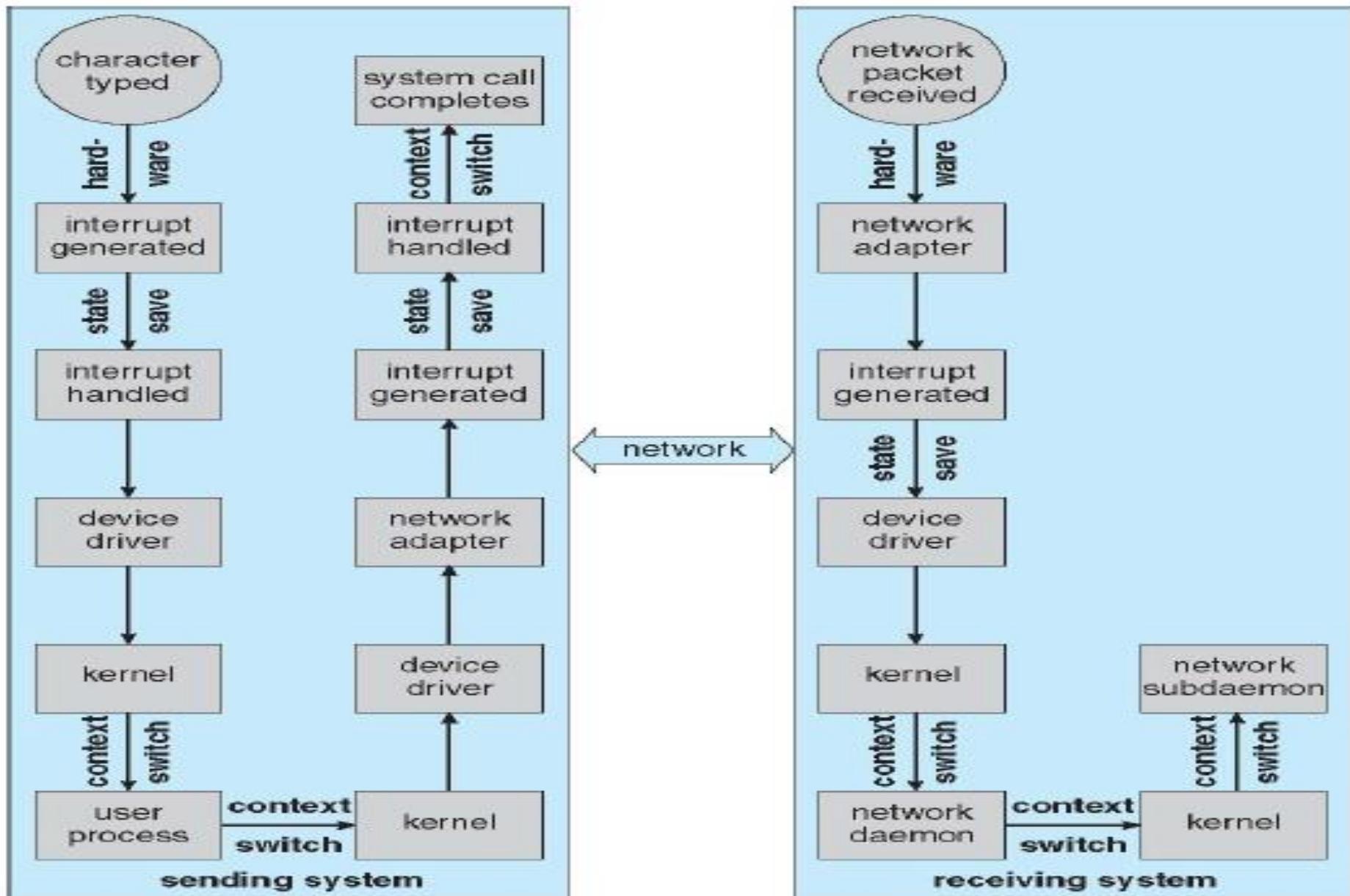


Figure1: Intercomputer Communication

4.2 Mass-Storage Device

- A mass storage device (MSD) is any storage device that makes it possible to store and port large amounts of data across computers, servers and within an IT environment. MSDs are portable storage media that provide a storage interface that can be both internal and external to the computer.

4.2.1 Disk Structure :

- Disk drives are addressed as large 1-dimensional arrays of logical blocks, where the logical block is the smallest unit of transfer.
- The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially
 - Sector 0 is the first sector of the first track on the outermost Cylinder.
 - Mapping proceeds in order through that track, then the rest Of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.
 - Logical to physical address should be easy
 - Except for bad sectors
 - Non-constant # of sectors per track via constant angular velocity

4.2.2 Disk Scheduling

- The operating system is responsible for using hardware efficiently — for the disk drives, this means having a fast access time and disk bandwidth.
- Access time has two major components
 - "*Seek time is the time for the disk are to move the heads to the cylinder containing the desired sector.*"
 - "*Rotational latency is the additional time waiting for the disk to rotate the desired sector to the disk head.*"
- Minimize seek time
- Seek time \approx seek distance
- Disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.

Disk Arm Scheduling Algorithm

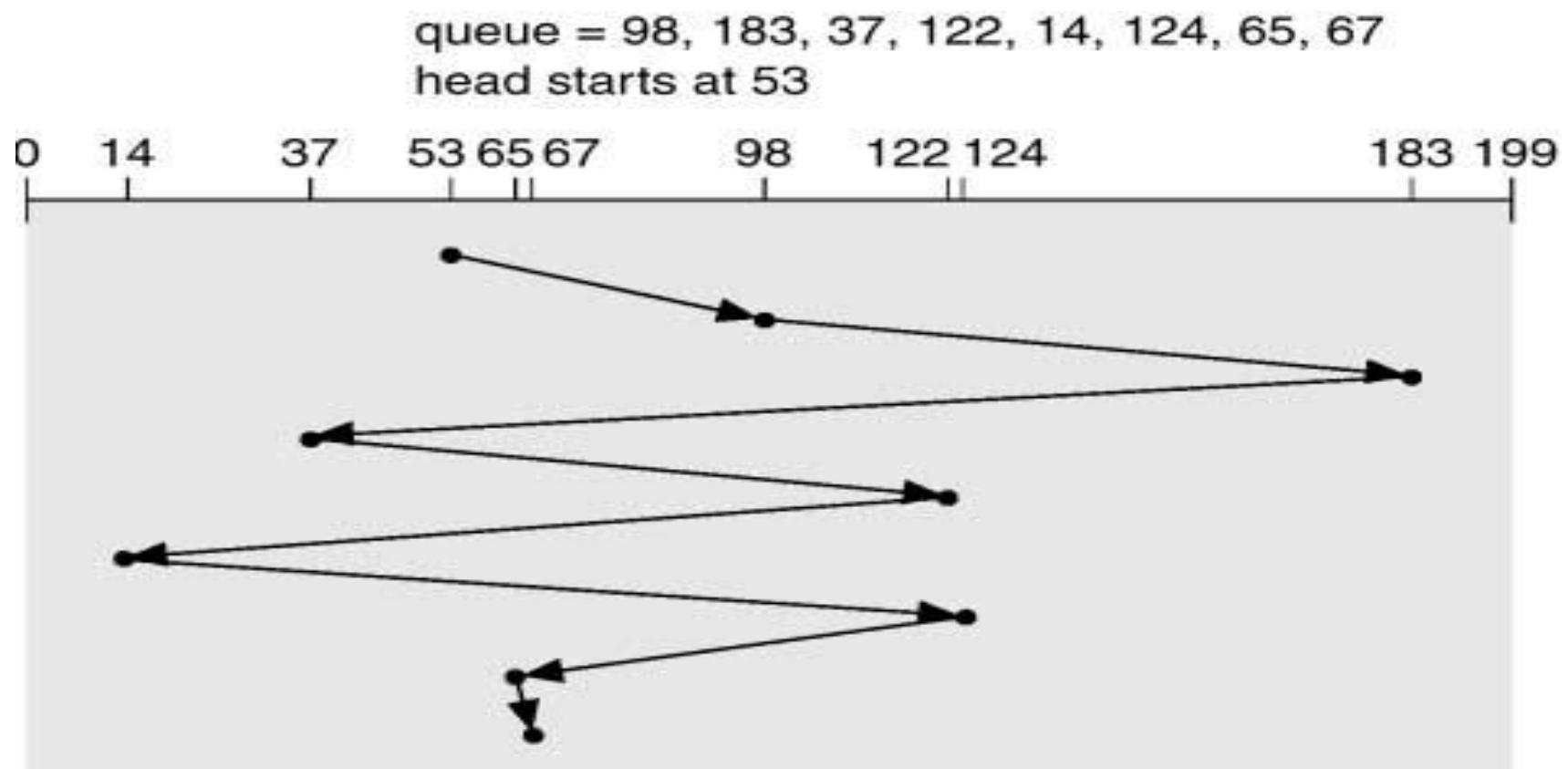
- Several algorithms exist to schedule the servicing of disk I/O requests.
- We illustrate them with a request queue (0-199).

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

Head pointer 53

FCFS

Illustration shows total head movement of 640 cylinders

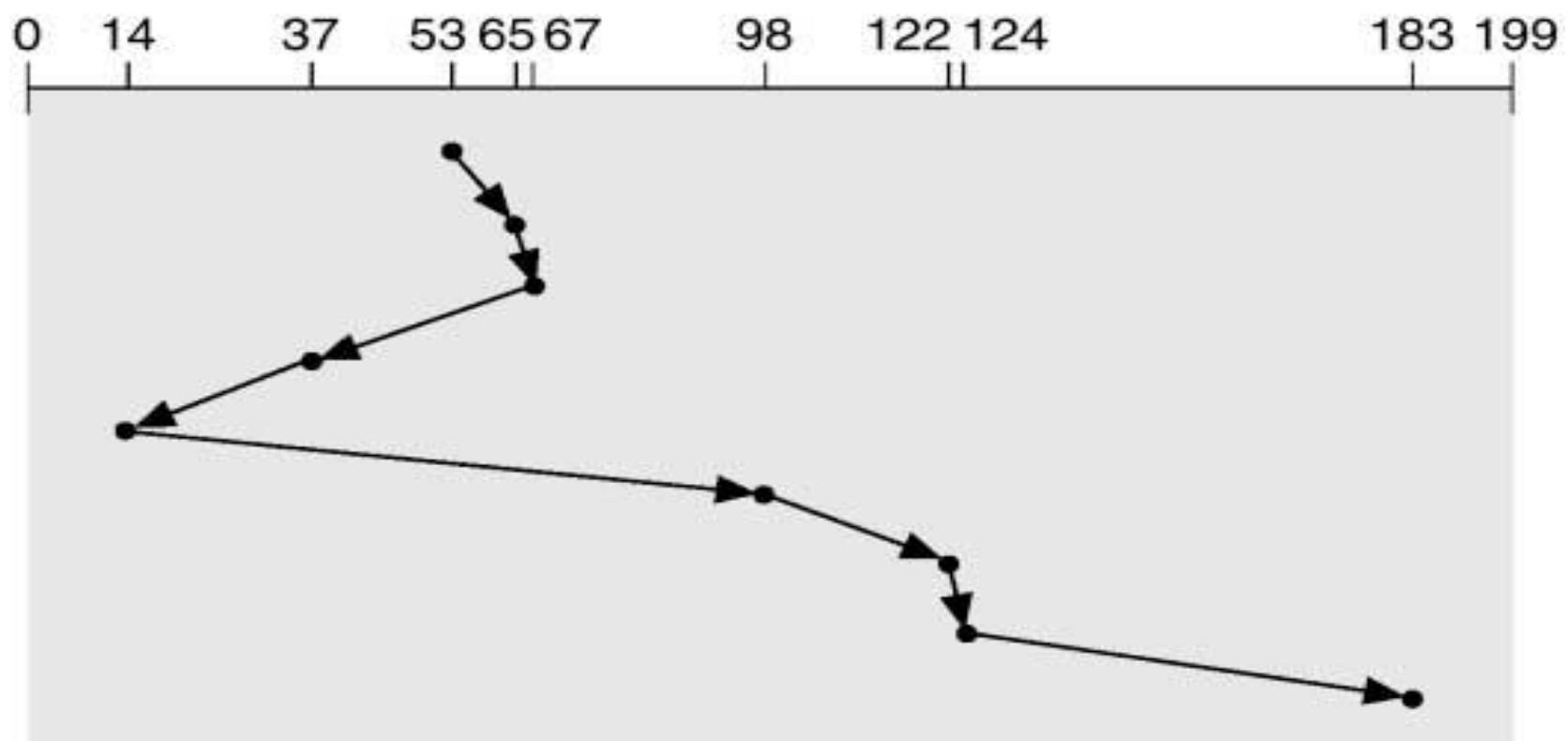


Shortest Seek First(SSF)

- Selects the request with the minimum seek time from the current head position.
- SSF scheduling is a form of SJF scheduling; may cause starvation of some requests.
- Illustration shows total head movement of 236 cylinders.

SSF (Cont.)

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53

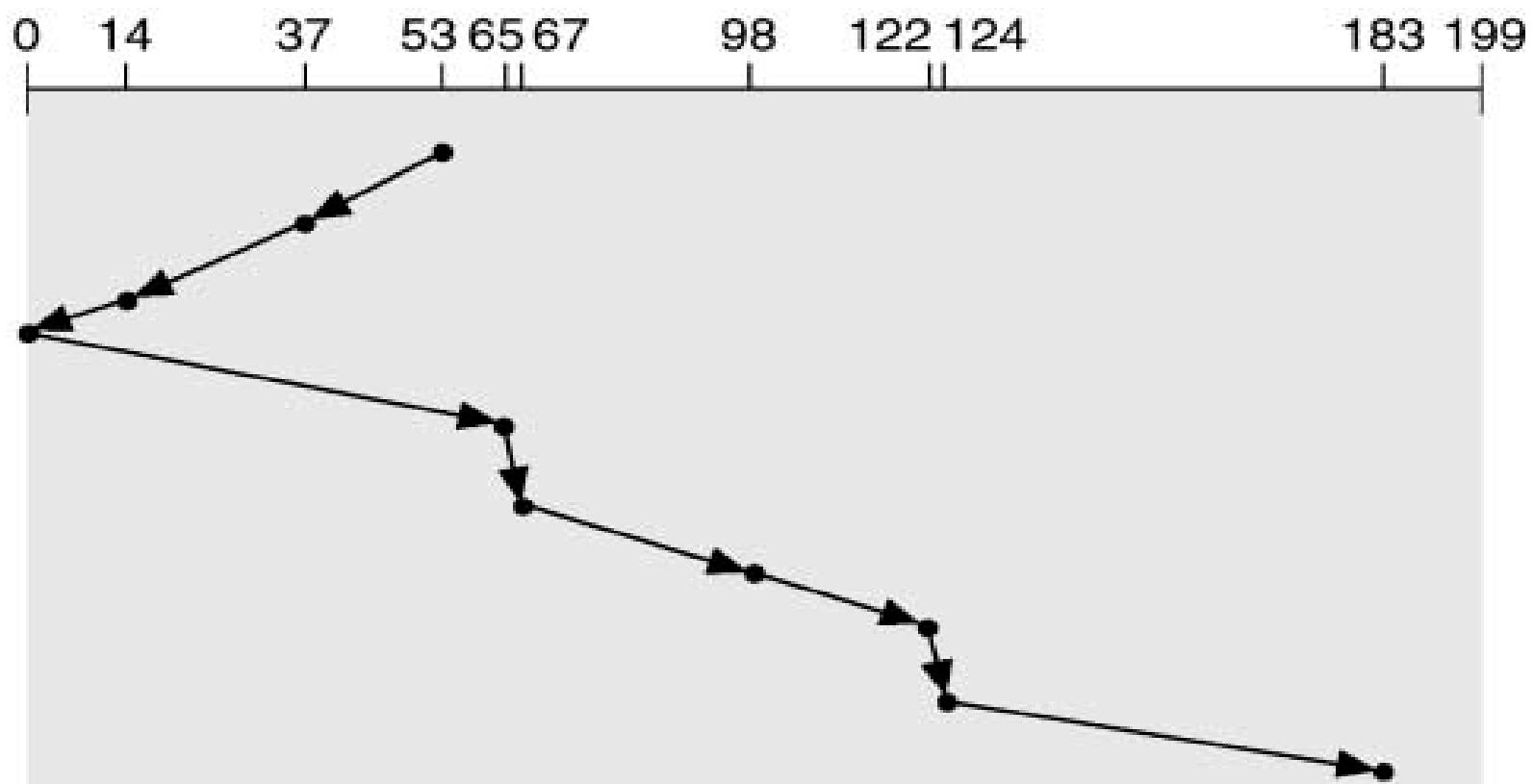


SCAN

- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.
- Sometimes called the *elevator algorithm*.
- Illustration shows total head movement of 208 cylinders.

SCAN (Cont.)

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53

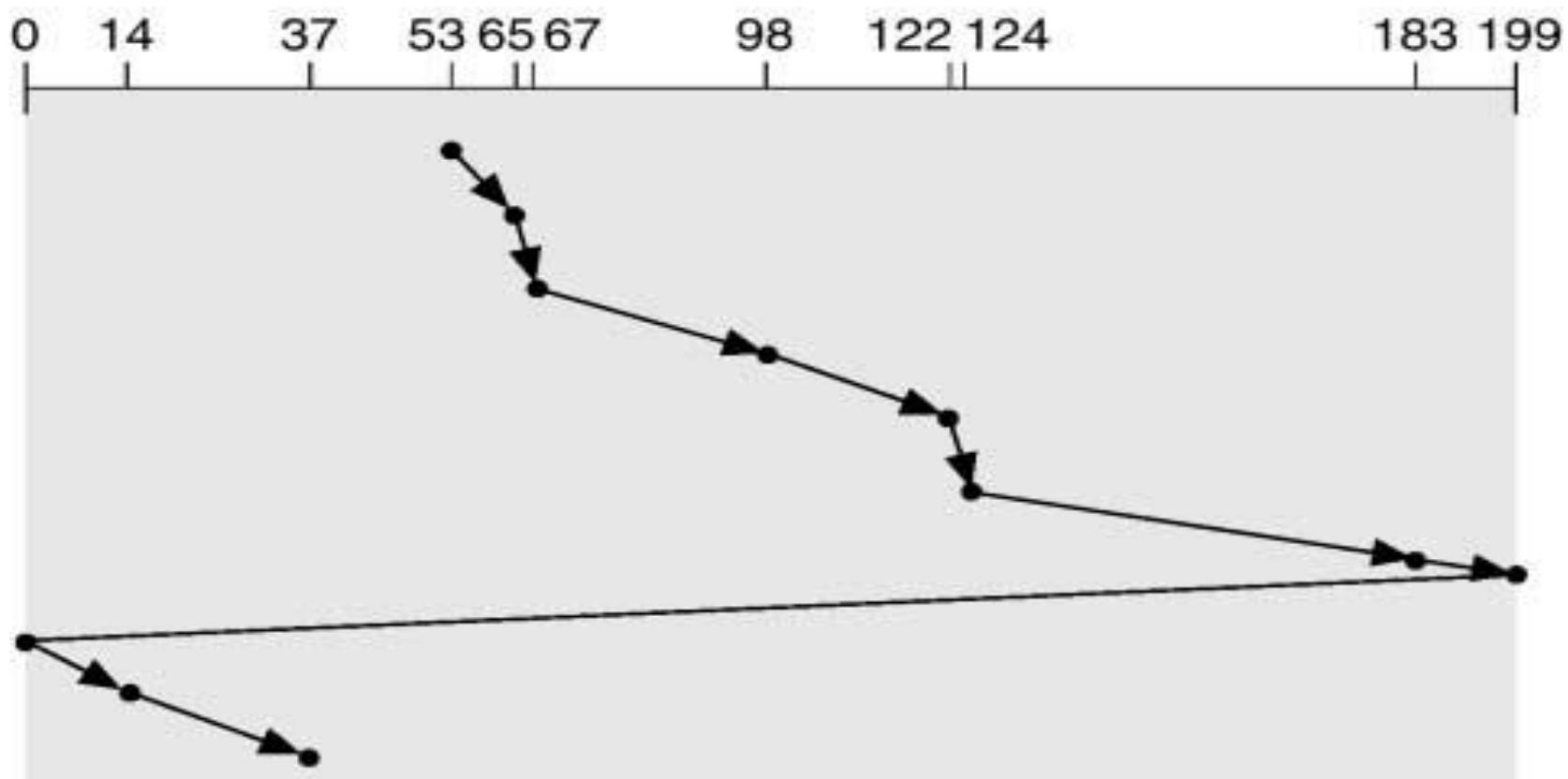


C-SCAN

- Provides a more uniform wait time than SCAN.
- The head moves from one end of the disk to the other, servicing requests as it goes. When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip.
- Treats the cylinders as a circular list that wraps around from the last cylinder to the first one.

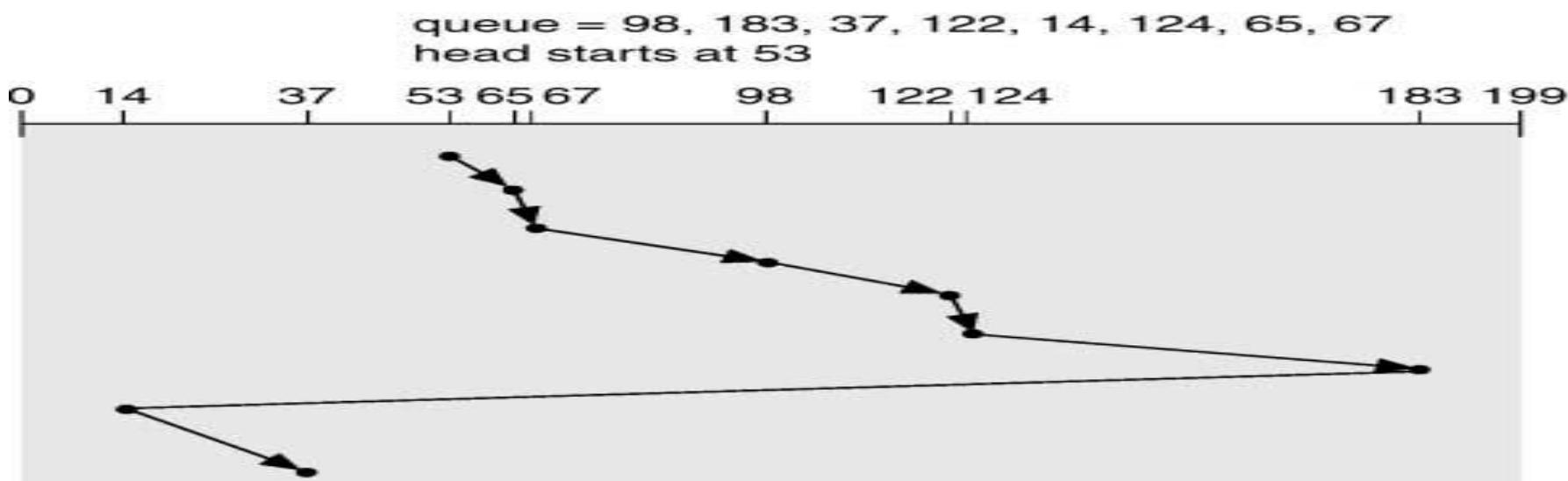
C-SCAN (Cont.)

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53



C-LOOK

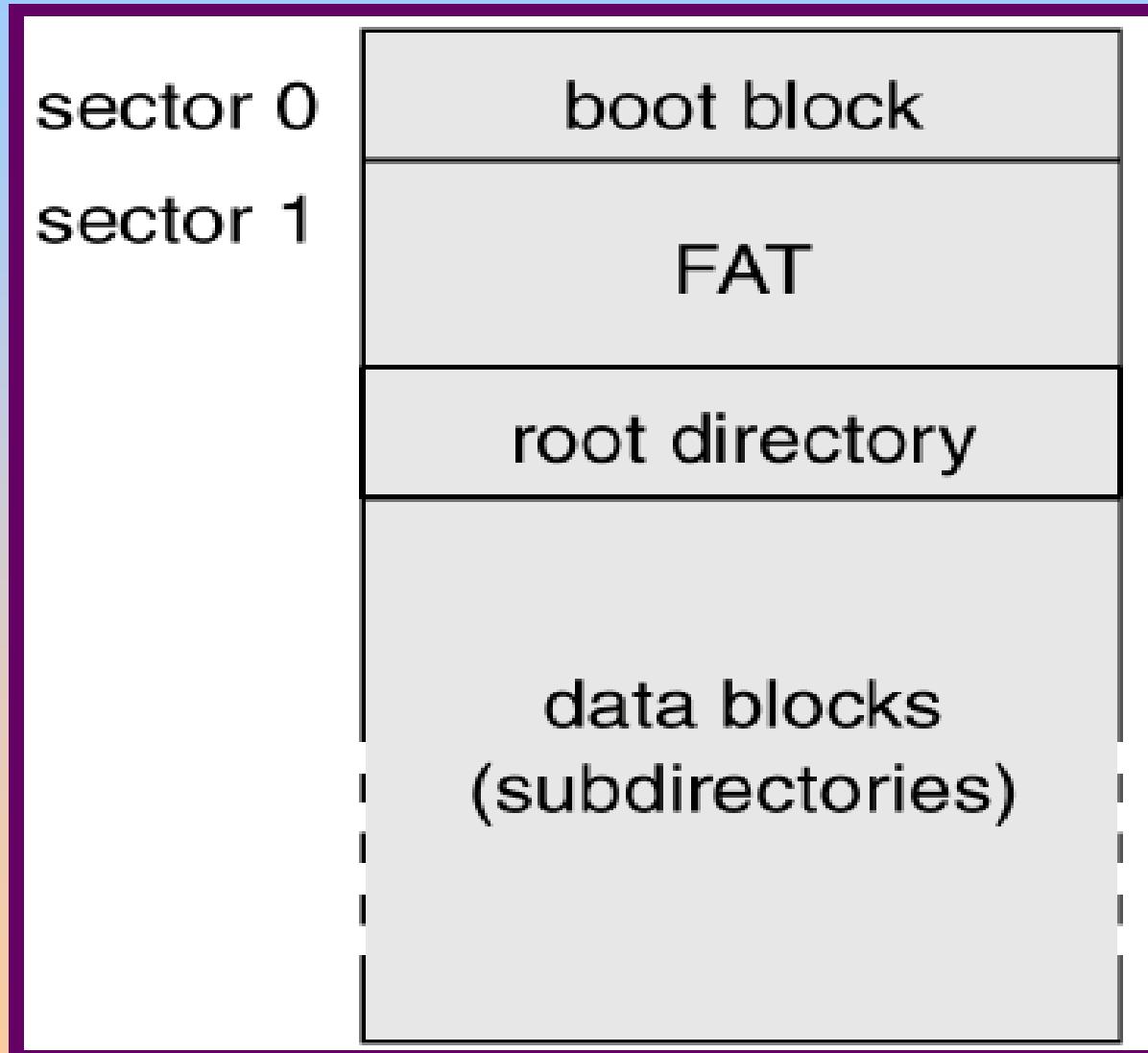
- Version of C-SCAN
- Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk.



4.2.3 Disk Management

- **Low-level formatting**, or **physical formatting** — Dividing a disk into sectors that the disk controller can read and write
 - Each sector can hold header information, plus data, plus error correction code (ECC)
 - Usually 512 bytes of data but can be selectable
- To use a disk to hold files, the operating system still needs to record its own data structures on the disk.
 - Partition the disk into one or more groups of cylinders, each treated as a logical disk
 - Logical formatting or “making a file system”
 - To increase efficiency most file systems group blocks into clusters
 - Disk I/O done in blocks
 - File I/O done in clusters
- Boot block initializes system
 - The bootstrap is stored in ROM
 - Bootstrap loader program stored in boot blocks of boot partition
- Methods such as sector sparing used to handle bad blocks

MS-DOS Disk Layout



Bad Blocks:

- The disk with defected sector is called as bad block.
- Depending on the disk and controller in use, these blocks are handled in a variety of ways;

Method 1: “Handled manually”

- If blocks go bad during normal operation, a special program must be run manually to search the bad blocks and to lock them away as before. Data that resided on the bad Blocks usually are lost.

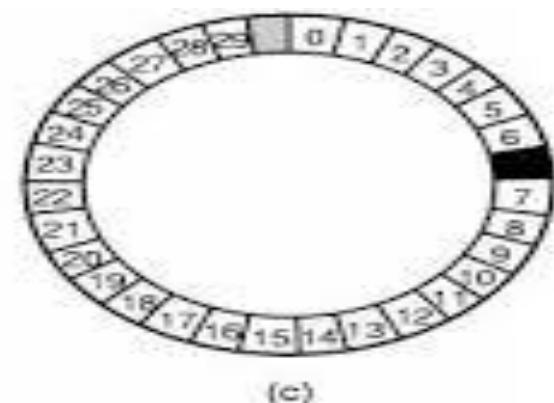
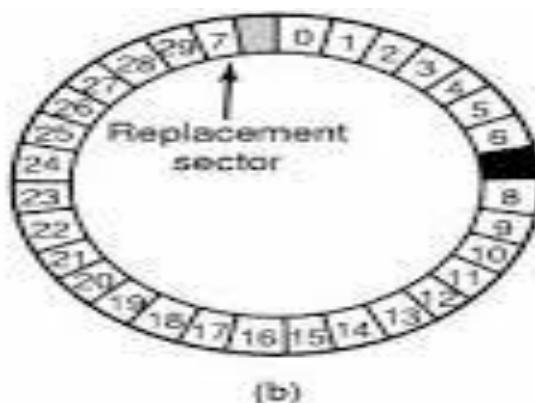
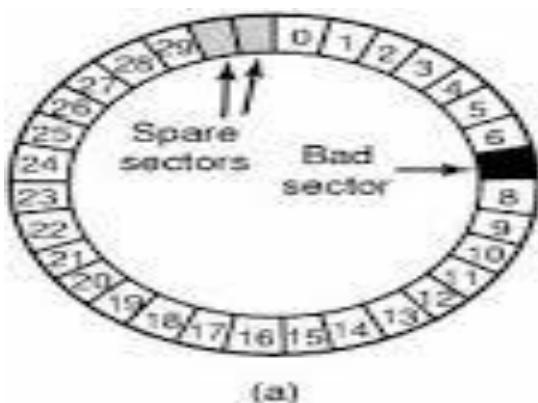
Method 2: “sector sparing or forwarding”

- The controller maintains a list of bad blocks on the disk. Then the controller can be told to replace each bad sector logically with one of the spare sectors. This scheme is known as sector sparing or forwarding.
- A typical bad-sector transaction might be as follows:
 - ✓ The operating system tries to read logical block 7.
 - ✓ The controller calculates the ECC and finds that the sector is bad.
 - ✓ It reports this finding to the operating system.
 - ✓ The next time that the system is rebooted, a special command is run to tell the controller to replace the bad sector with a spare.
 - ✓ After that, whenever the system requests logical block 7, the request is translated into the replacement sector's address by the controller.

Bad Block handling(contd...)

Method 3: “sector slipping”

- For an example, suppose that logical block 17 becomes defective, and the first available spare follows sector 202. Then, sector slipping would remap all the sectors from 17 to 202, moving them all down one spot. That is, sector 202 would be copied into the spare, then sector 201 into 202, and then 200 into 201, and so on, until sector 18 is copied into sector 19. Slipping the sectors in this way frees up the space of sector 18, so sector 17 can be mapped to it.



Interleaving:

- Interleaving is a process or methodology to make a system more efficient, fast and reliable by arranging data in a noncontiguous manner. There are many uses for interleaving at the system level, including:
 - Storage: As hard disks and other storage devices are used to store user and system data, there is always a need to arrange the stored data in an appropriate way.
 - Error Correction: Errors in data communication and memory can be corrected through interleaving.

Interleaving is also known as sector interleave.

- When used to describe disk drives, it refers to the way *sectors* on a disk are organized. In one-to-one interleaving, the sectors are placed sequentially around each track. In two-to-one interleaving, sectors are staggered so that consecutively numbered sectors are separated by an intervening sector.
- The purpose of interleaving is to make the disk drive more efficient. The disk drive can access only one sector at a time, and the disk is constantly spinning beneath the **read/write head**. This means that by the time the drive is ready to access the next sector, the disk may have already spun beyond it. If a data file spans more than one sector and if the sectors are arranged sequentially, the drive will need to wait a full rotation to access the next chunk of the file. If instead the sectors are staggered, the disk will be perfectly positioned to access sequential sectors.

4.2.4 Swap-Space Management

- Swap-space — Virtual memory uses disk space as an extension of main memory.
 - Less common now due to memory capacity increases
- Swap-space can be carved out of the normal file system, or, more commonly, it can be in a separate disk partition (raw)
- Swap-space management
 - 4.3BSD allocates swap space when process starts; holds text segment (the program) and data segment
 - Kernel uses swap maps to track swap-space use
 - Solaris 2 allocates swap space only when a dirty page is forced out of physical memory, not when the virtual memory page is first created
 - File data written to swap space until write to file system requested
 - Other dirty pages go to swap space due to no other home
 - Text segment pages thrown out and reread from the file system as needed
- What if a system runs out of swap space?
- Some systems allow multiple swap spaces

Swap-Space Management: An Example

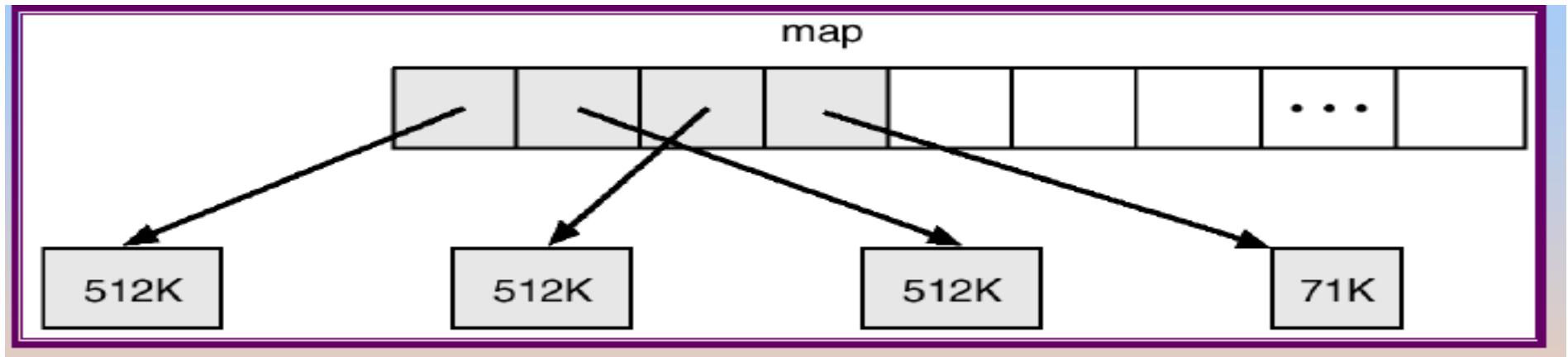


Fig1.1: 4.3BSD text-segment swap map.

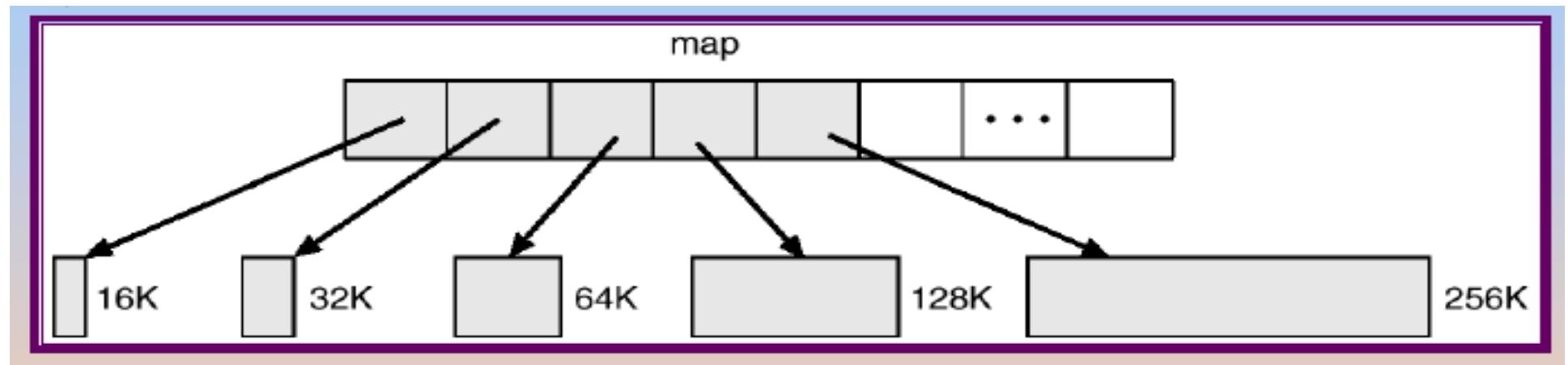


Fig 1.2: 4.3BSD data-segment swap map

- In 4.3 BSD, swap space is allocated to a process when the process is started. Enough space is set aside to hold the program, known as the text pages or the text segment, and the data segment of the process. Preallocating all the needed space in this way generally prevents a process from running out of swap space while it executes. When a process starts, its text is paged in from the file system. These pages are written out to swap when necessary, and are read back in from there, so the file system is consulted only once for each text page. Pages from the data segment are read in from the file system, or are created (if they are uninitialized), and are written to swap space and paged back in as needed. One optimization (for instance, when two users run the same editor) is that processes with identical text pages share these pages, both in physical memory and in swap space.
- Two per-process swap maps are used by the kernel to track swap-space use. The text segment is a fixed size, so its swap space is allocated in 512 KB chunks, except for the final chunk, which holds the remainder of the pages, in 1 KB increments(fig 1.1)
- The data-segment swap map is more complicated, because the data segment can grow over time. The map is of fixed size, but contains swap addresses for blocks of varying size. Given index i , a block pointed to by swap-map entry i is of size $2^i \times 16$ KB, to a maximum of 2 MB. This data structure is shown in Figure 1.2. (The block size minimum and maximum are variable, and can be changed at system reboot.) When a process tries to grow its data segment beyond the final allocated block in its swap area, the operating system allocates another block, twice as large as the previous one. This scheme results in small processes using only small blocks. It also minimizes fragmentation. The blocks of large processes can be found quickly, and the swap map remains small.

4.2.5 Stable-Storage Implementation

- Write-ahead log scheme requires stable storage
- To implement stable storage:
 - Replicate information on more than one nonvolatile storage media with independent failure modes
 - Update information in a controlled manner to ensure that we can recover the stable data after any failure during data transfer or recovery.

A disk write results in one of three outcomes:

1. Successful completion: The data were written correctly on disk.
2. Partial failure: A failure occurred in the midst of transfer, so only some of the sectors were written with the new data, and the sector being written during the failure may have been corrupted.
3. Total failure: The failure occurred before the disk write started, so the previous data values on the disk remain intact.

Stable-Storage Implementation(contd....)

- Whenever a failure occurs during writing of a block, the system needs to detect it and invoke a recovery procedure to restore the block to a consistent state. To do that, the system must maintain two physical blocks for each logical block. An output operation is executed as follows:
 1. Write the information onto the first physical block.
 2. When the first write completes successfully, write the same information onto the second physical block.
 3. Declare the operation complete only after the second write completes successfully.
- During recovery from a failure, each pair of physical blocks is examined. If both are the same and no detectable error exists, then no further action is necessary. If one block contains a detectable error then we replace its contents with the value of the other block. If neither block contains a detectable error, but the blocks differ in content, then we replace the content of the first block with that of the second. This recovery procedure ensures that a write to stable storage either succeeds completely or results in no change.

4.3.6 Tertiary Storage Devices

- Low cost is the defining characteristic of tertiary storage
- Generally, tertiary storage is built using removable media
- Common examples of removable media are floppy disks and CD-ROMs; other types are available

Removable Disks

- Floppy disk — thin flexible disk coated with magnetic material, enclosed in a protective plastic case
 - Most floppies hold about 1 MB; similar technology is used for removable disks that hold more than 1 GB
 - Removable magnetic disks can be nearly as fast as hard disks, but they are at a greater risk of damage from exposure
- A magneto-optic disk records data on a rigid platter coated with magnetic material
 - Laser heat is used to amplify a large, weak magnetic field to record a bit
 - Laser light is also used to read data (Kerr effect)
 - The magneto-optic head flies much farther from the disk surface than a magnetic disk head, and the magnetic material is covered with a protective layer of plastic or glass; resistant to head crashes
- Optical disks do not use magnetism; they employ special materials that are altered by laser light

WORM Disks

- The data on read-write disks can be modified over and over
- WORM (“Write Once, Read Many Times”) disks can be written only once
- Thin aluminum film sandwiched between two glass or plastic platters
- To write a bit, the drive uses a laser light to burn a small hole through the aluminum; information can be destroyed by not altered
- Very durable and reliable
- Read-only disks, such as CD-ROM and DVD, come from the factory with the data pre-recorded

Tapes

- Compared to a disk, a tape is less expensive and holds more data, but random access is much slower.
- Tape is an economical medium for purposes that do not require fast random access, e.g., backup copies of disk data, holding huge volumes of data.
- Large tape installations typically use robotic tape changers that move tapes between tape drives and storage slots in a tape library
 - stacker – library that holds a few tapes
 - silo – library that holds thousands of tapes
- A disk-resident file can be archived to tape for low cost storage; the computer can stage it back into disk storage for active use.

Operating System Support

- Major OS jobs are to manage physical devices and to present a virtual machine abstraction to applications
- For hard disks, the OS provides two abstraction:
 - Raw device – an array of data blocks
 - File system – the OS queues and schedules the interleaved requests from several applications

Application Interface

- Most OSs handle removable disks almost exactly like fixed disks — a new cartridge is formatted and an empty file system is generated on the disk
- Tapes are presented as a raw storage medium, i.e., and application does not open a file on the tape, it opens the whole tape drive as a raw device
- Usually the tape drive is reserved for the exclusive use of that application
- Since the OS does not provide file system services, the application must decide how to use the array of blocks
- Since every application makes up its own rules for how to organize a tape, a tape full of data can generally only be used by the program that created it

Tape Drives

- The basic operations for a tape drive differ from those of a disk drive.
- ***locate*** positions the tape to a specific logical block, not an entire track (corresponds to ***seek***).
- The ***read position*** operation returns the logical block number where the tape head is.
- The ***space*** operation enables relative motion.
- Tape drives are “append-only” devices; updating a block in the middle of the tape also effectively erases everything beyond that block.
- An EOT mark is placed after a block that is written.

File Naming

- The issue of naming files on removable media is especially difficult when we want to write data on a removable cartridge on one computer, and then use the cartridge in another computer.
- Contemporary OSs generally leave the name space problem unsolved for removable media, and depend on applications and users to figure out how to access and interpret the data.
- Some kinds of removable media (e.g., CDs) are so well standardized that all computers use them the same way.

Hierarchical Storage Management (HSM)

- A hierarchical storage system extends the storage hierarchy beyond primary memory and secondary storage to incorporate tertiary storage — usually implemented as a jukebox of tapes or removable disks.
- Usually incorporate tertiary storage by extending the file system
 - Small and frequently used files remain on disk
 - Large, old, inactive files are archived to the jukebox
- HSM is usually found in supercomputing centers and other large installations that have enormous volumes of data.

4.3.7 I/O in UNIX

There are two main categories of I/O units in UNIX, block devices and character devices. In addition there are sockets that are used for network communication.

Block devices

- Devices that addresses blocks of a fixed size, usually disk memories.
- Data blocks are buffered in the buffer cache.
- Block devices are usually called via the file system, but are also available as special files (for example /dev/hde1).

Character devices

- Terminals and printers, but also everything else (except sockets) that do not use the block buffer cache.
- There are for example /dev/mem that is an interface to physical memory.

UNIX I/O System

- Device drivers are called via a switch table. There is one switch table for block devices and one for character devices. A hardware device is identified by its type (block or character) and a device number.
- Device numbers consist of two parts: major device number and minor device number. Major device number is used as an index in the switch table to locate the correct device driver.
- Minor device number is forwarded to the device driver and used to select correct subunit. (For example correct file system partition if the disk is divided in several partitions)

RAID

- Disk drives have continued to get smaller and cheaper, so it is now economically feasible to attach a large number of disks to a computer system. Having a large number of disks in a system presents opportunities for improving the rate at which data can be read or written, if the disks are operated in parallel. Furthermore, this setup offers the potential for improving the reliability of data storage, because redundant information can be stored on multiple disks. Thus, failure of one disk does not lead to loss of data. A variety of disk-organization techniques, collectively called redundant arrays of inexpensive disks (RAID), are commonly used to address the performance and reliability issues.

RAID LEVELS

RAID level 0 – Striping:

- In a RAID 0 system data are split up into blocks that get written across all the drives in the array. By using multiple disks (at least 2) at the same time, this offers superior I/O performance. This performance can be enhanced further by using multiple controllers, ideally one controller per disk.

Advantages:

- RAID 0 offers great performance, both in read and write operations. There is no overhead caused by parity controls.
- All storage capacity is used, there is no overhead.
- The technology is easy to implement.

Disadvantages:

- RAID 0 is not fault-tolerant. If one drive fails, all data in the RAID 0 array are lost. It should not be used for mission-critical systems.

RAID LEVELS

RAID level 1 – Mirroring:

- Data are stored twice by writing them to both the data drive (or set of data drives) and a mirror drive (or set of drives). If a drive fails, the controller uses either the data drive or the mirror drive for data recovery and continues operation. You need at least 2 drives for a RAID 1 array.

Advantages

- RAID 1 offers excellent read speed and a write-speed that is comparable to that of a single drive.
- In case a drive fails, data do not have to be rebuild, they just have to be copied to the replacement drive.
- RAID 1 is a very simple technology.

Disadvantages

- The main disadvantage is that the effective storage capacity is only half of the total drive capacity because all data get written twice.

RAID LEVELS

RAID level 5:

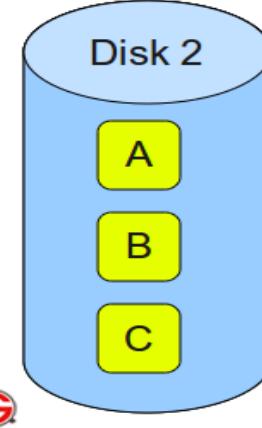
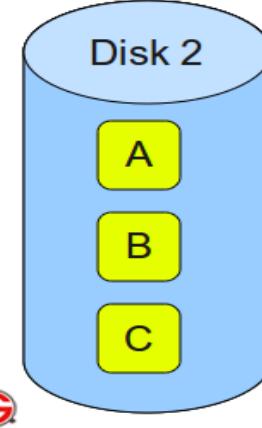
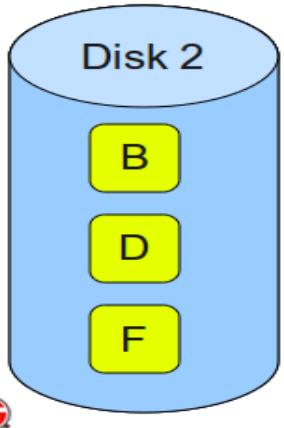
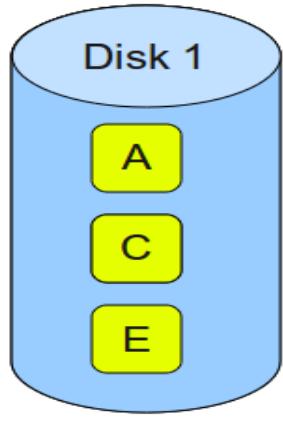
- RAID 5 is the most common secure RAID level. It requires at least 3 drives but can work with up to 16. Data blocks are striped across the drives and on one drive a parity checksum of all the block data is written. The parity data are not written to a fixed drive, they are spread across all drives, as the drawing below shows. Using the parity data, the computer can recalculate the data of one of the other data blocks, should those data no longer be available. That means a RAID 5 array can withstand a single drive failure without losing data or access to data.

Advantages:

- Read data transactions are very fast while write data transactions are somewhat slower (due to the parity that has to be calculated).
- If a drive fails, you still have access to all data, even while the failed drive is being replaced and the storage controller rebuilds the data on the new drive.

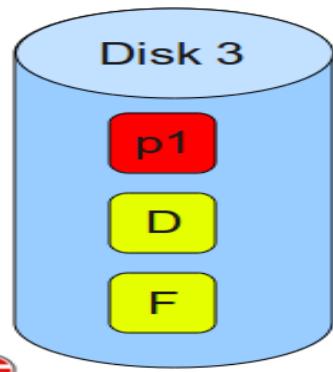
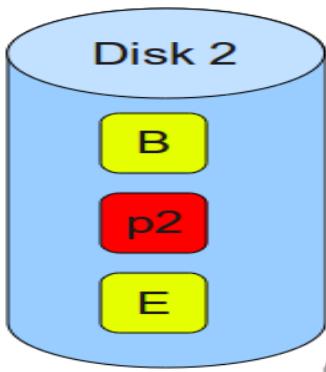
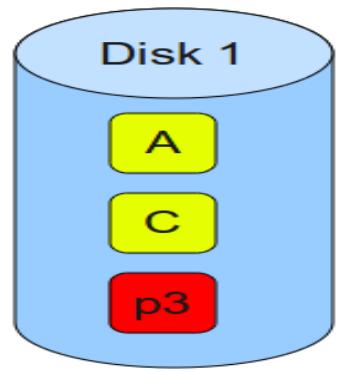
Disadvantages:

- Drive failures have an effect on throughput, although this is still acceptable.



RAID 0 – Blocks Striped. No Mirror. No Parity.

RAID 1 – Blocks Mirrored. No Stripe. No parity.



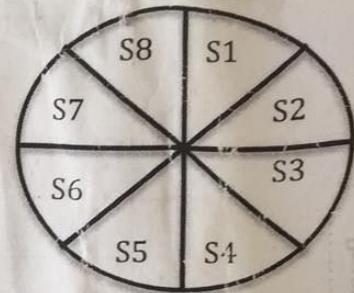
RAID 5 – Blocks Striped. Distributed Parity.

Q. A disk has **8 sectors** per track and spins at **600 rpm**. It takes the controller **10 ms** from the end of one I/O operation before it can issue a subsequent one. How long does it take to read all 8 sectors using the following interleaving systems?

- (a) No interleaving
- (b) Single interleaving
- (c) Double interleaving

Ans:

- Interleaving is when sectors are arranged in alternate order.
- (a) When there is **no interleaving** then sectors would be like this
 $S1 \rightarrow S2 \rightarrow S3 \rightarrow S4 \rightarrow S5 \rightarrow S6 \rightarrow S7 \rightarrow S8$
- We know,
 In 60 sec disk revolves 600 rotations. [600 RPM]
 In 1 sec disk revolves $600/60 = 10$ rotation.



For 10 rotations it requires 1000 ms [1 sec = 1000 ms]

For 1 rotation it requires $1000/10 = 100$ ms

Total time spent on each sector = $100/8 = 12.5$ ms [1/8th of the track = $100/8$]

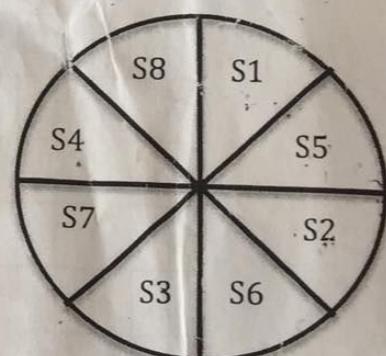
The time it takes to load another request is 10ms that is less than 12.5ms

In first request it read sector **S1** then while the disk is spinning the controller went to fetch another request. And **10 msec** are waste out of **12.5 msec** that means more than half of the **S2** had already passed therefore the heads to rotate again to fetch **S2**. So will be the case of all **8 sectors** meaning it will take **8 revolutions** while each revolution takes **100 msec** and that would be **800 msec**.

- (b) When there is **Single Interleaving** then the sectors would be arranged in the order

$S1 \rightarrow S5 \rightarrow S2 \rightarrow S6 \rightarrow S3 \rightarrow S7 \rightarrow S4 \rightarrow S8$

therefore in 1 revolution 4 sectors will be read and
 in 2 revolutions $4 \times 2 = 8$ sectors will be read



for 1 rotation it takes 100 msec

therefore for 2 rotations it takes $100 \times 2 = 200$ msec

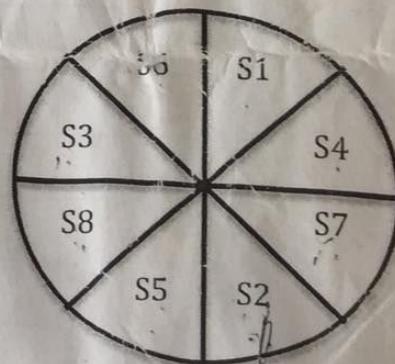
- (c) When there is **Double Interleaving** then the sectors would be arranged in

$S1 \rightarrow S4 \rightarrow S7 \rightarrow S2 \rightarrow S5 \rightarrow S8 \rightarrow S3 \rightarrow S6$

therefore only 3 sectors will be read in 1 revolution.

for 1 rotations it takes 100 msec

for 3 rotations it takes $100 \times 3 = 300$ msec.



but $360/8$ is 45 degrees, i.e. only $3/4$ of the disk rev is required in the 3rd revolution

so we need to multiply $100 \times 3/4$ to get the actual time needed to read the last sector **S8**. Since it comprises only $3/4$.
 Upto 2nd rotations it takes 200 msec.

For 3rd rotation it takes $3/4 \times 100 = 75$ msec

So total time = $200 + 75 = 275$ msec

Applied Operating System

Chapter 5: File Systems

Prepared By:

Amit K. Shrivastava

Asst. Professor

Nepal College Of Information Technology

Files

- File: a named collection of data that may be manipulated as a unit by operations such as:
 - Open, Close, Create, Destroy, Copy, Rename, List
- Individual data items within a file may be manipulated by operations like:
 - Read
 - Write
 - Update
 - Insert
 - Delete
- File characteristics include:
 - Location
 - Accessibility
 - Type
 - Volatility
- Files can consist of one or more records

File Naming

- Probably the most important characteristic of any abstraction mechanism is the way the objects being managed are named. When a process creates a file, it gives the file a name. When the process terminates, the file continues to exist and can be accessed by other processes using its name.
- The exact rules for file naming vary somewhat from system to system, but all current operating systems allow strings of one to eight letters as legal file names. Frequently digits and special characters are also permitted. *Many file systems support names as long as 255 characters.*
- Some file systems distinguish between upper- and lower-case letters, whereas others do not. UNIX (including all its variants) falls in the first category; MS-DOS falls in the second.

File Structure

- None -sequence of words, bytes
- Simple record structure
 - Lines
 - Fixed length
 - Variable length
- Complex Structures
 - Formatted document
 - Relocatable load file
- Can simulate last two with first method by inserting appropriate control characters
- Who decides:
 - Operating system
 - Program

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 systems that support different types of files.
- **Location.** This information is a pointer to a device and to the location of the file on that device.
- **Size.** The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
- **Protection.** Access-control information determines who can do reading, writing, executing, and so on.
- **Time, date, and user identification.** This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.
- Information about files are kept in the directory structure, which is maintained on the disk

File Operations

- **Creating a file:** Two steps are necessary to create a file. First, space in the file system must be found for the file. Second, an entry for the new file must be made in the directory.
- **Writing a file:** To write a file, we make a system call specifying both the name of the file and the information to be written to the file.
- **Reading a file:** To read from a file, we use a system call that specifies the name of the file and where (in memory) the next block of the file should be put.
- **Repositioning within a file:** The directory is searched for the appropriate entry, and the current-file-position is set to a given value. Repositioning within a file does not need to involve any actual I/O. This file operation is also known as a *file seek*.

File Operations(contd..)

- **Deleting a file:** To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files, and erase the directory entry.
- **Truncating a file:** The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged-except for file length-but lets the file be reset to length zero and its file space released.

File Types - Name, Extension

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

File Descriptor

- A file descriptor or file control block is a control block containing information the system needs to manage a file. It is a highly system-dependent structure. A typical file descriptor might include
 - Symbolic file name, location of file in secondary storage, file organization, access control data, type, disposition(permanent vs. temporary), creation date and time, destroy date, date and time last modified, access activity counts(number of reads, for example)
- Ordinarily, file descriptor are maintained on secondary storage. They are brought to primary storage when a file is opened and is controlled by the operating system

File Access Methods

- **Sequential Access:** 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.

read next

write next

Reset

no read after last write

(rewrite)

File Access Methods(contd..)

- **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 method 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.

read n

write n

position to n

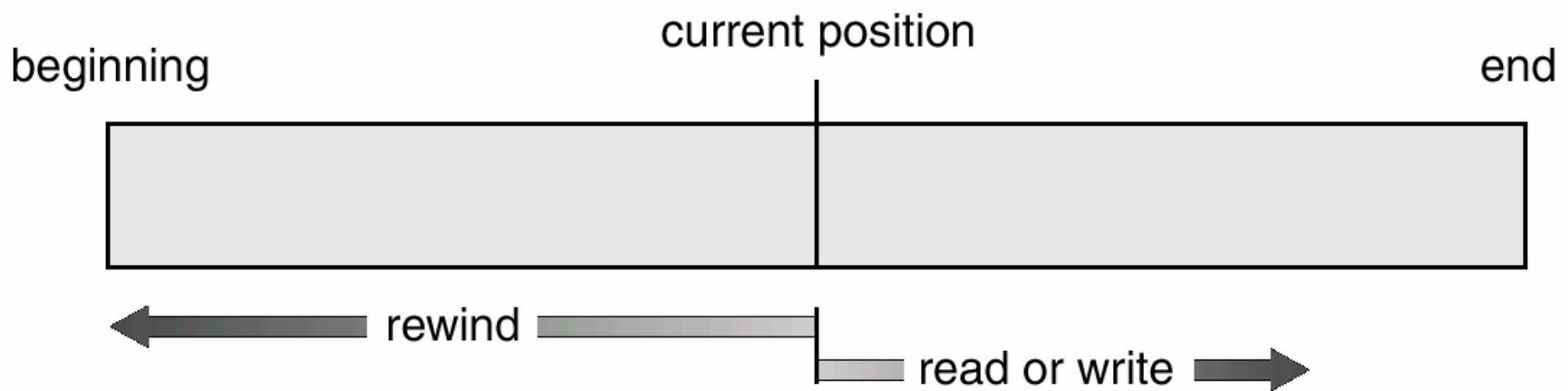
 read next

 write next

rewrite n

n= relative block number

Sequential-access File

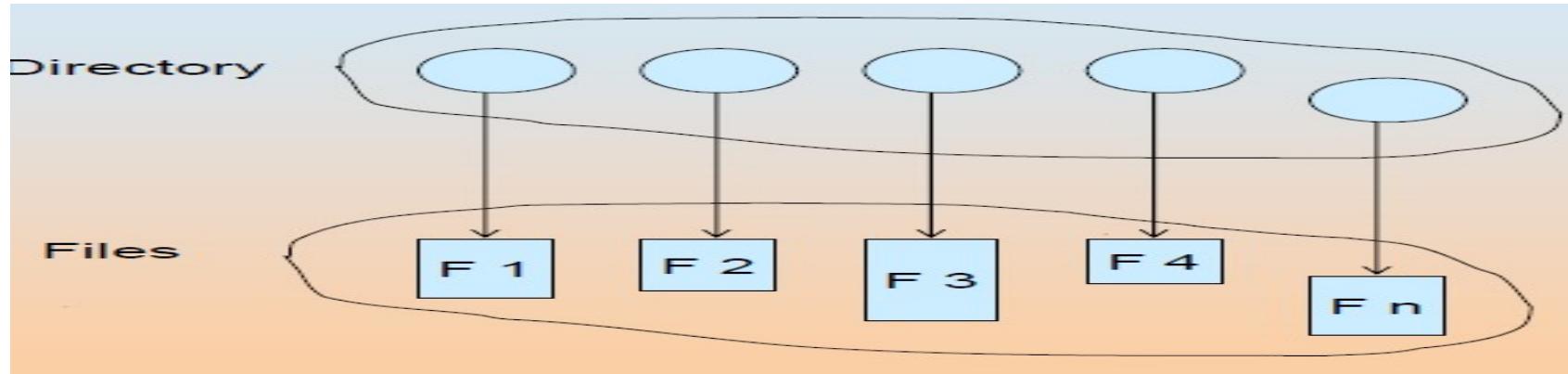


Simulation of Sequential Access on a Direct access File

sequential access	implementation for direct access
<i>reset</i>	$cp = 0;$
<i>read next</i>	<i>read cp;</i> $cp = cp+1;$
<i>write next</i>	<i>write cp;</i> $cp = cp+1;$

Directory Structure

- Directories:
 - Files containing the names and locations of other files in the file system, to organize and quickly locate files
- Directory entry stores information such as:
 - File name
 - Location
 - Size
 - Type
 - Accessed
 - Modified and creation times



Both the directory structure and the files reside on disk
Backups of these two structures are kept on tapes

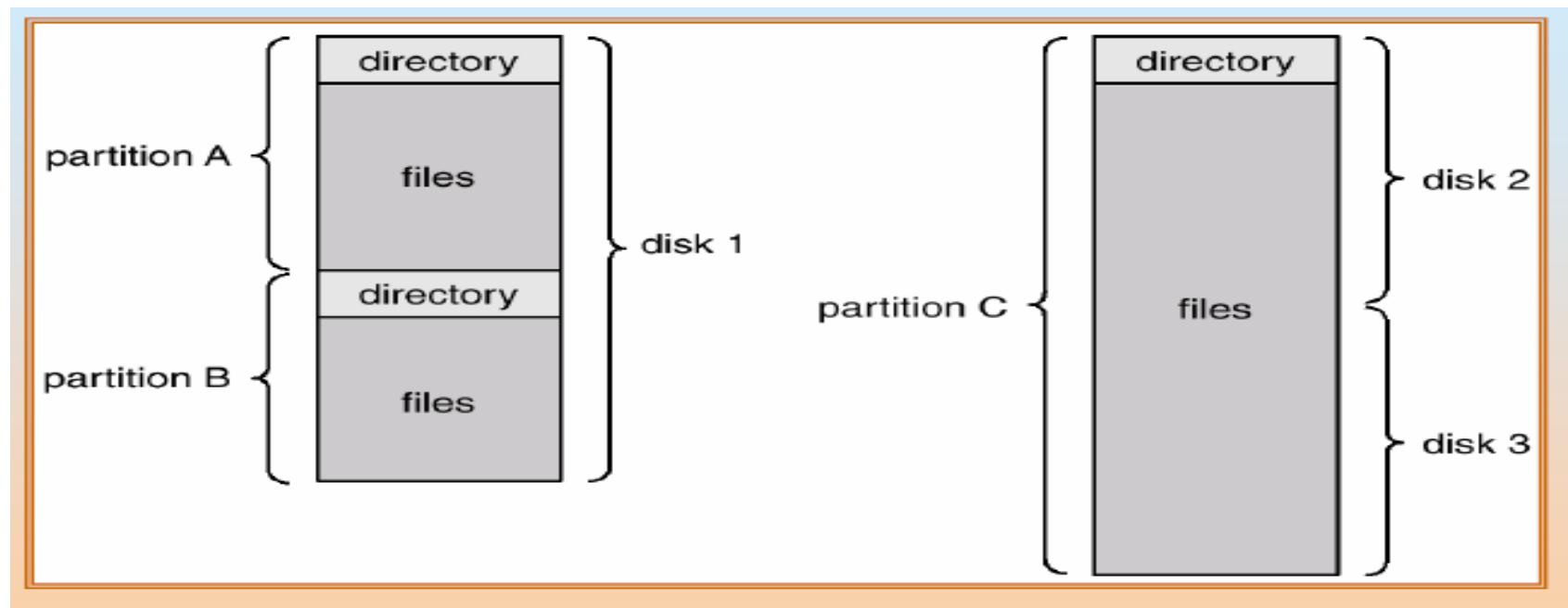


Fig: A typical file- system organization

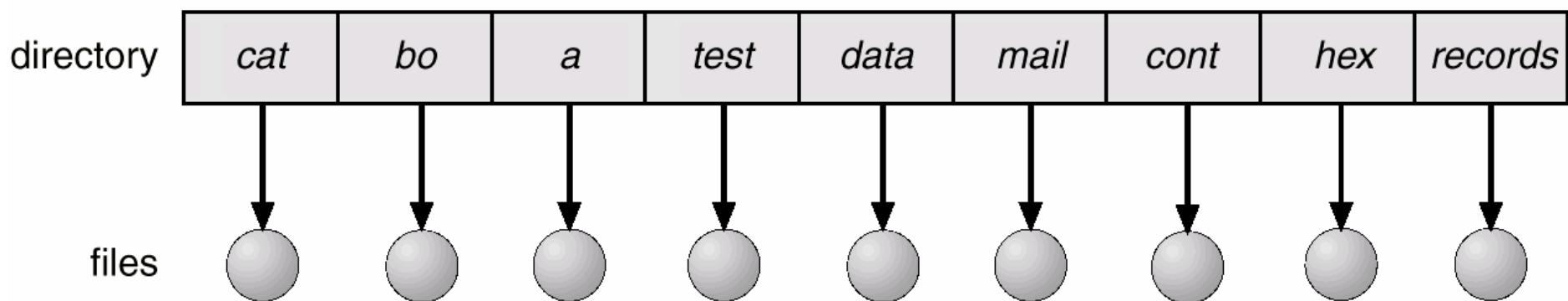
Directory Operations

- **Create a file:** New files need to be created and added to the directory.
- **Delete a file:** When a file is no longer needed, we want to remove it from the directory.
- **List a directory:** We need to be able to list the files in a directory, and the contents of the directory entry for each file in the list.
- **Rename a file:** Because the name of a file represents its contents to its users, the name must be changeable when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.
- **Traverse the file system:** We may wish to access every directory, and every file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals.

Most common schemes for defining the logical structure of a directory are:

Single-level (or flat) directory:

- Simplest file system organization
- Stores all of its files using one directory
- No two files can have the same name
- File system must perform a linear search of the directory contents to locate each file, which can lead to poor performance



Hierarchical Directory

- A root indicates where on the storage device the root directory begins
- The root directory points to the various directories, each of which contains an entry for each of its files
- File names need be unique only within a given user directory
- The name of a file is usually formed as the pathname from the root directory to the file

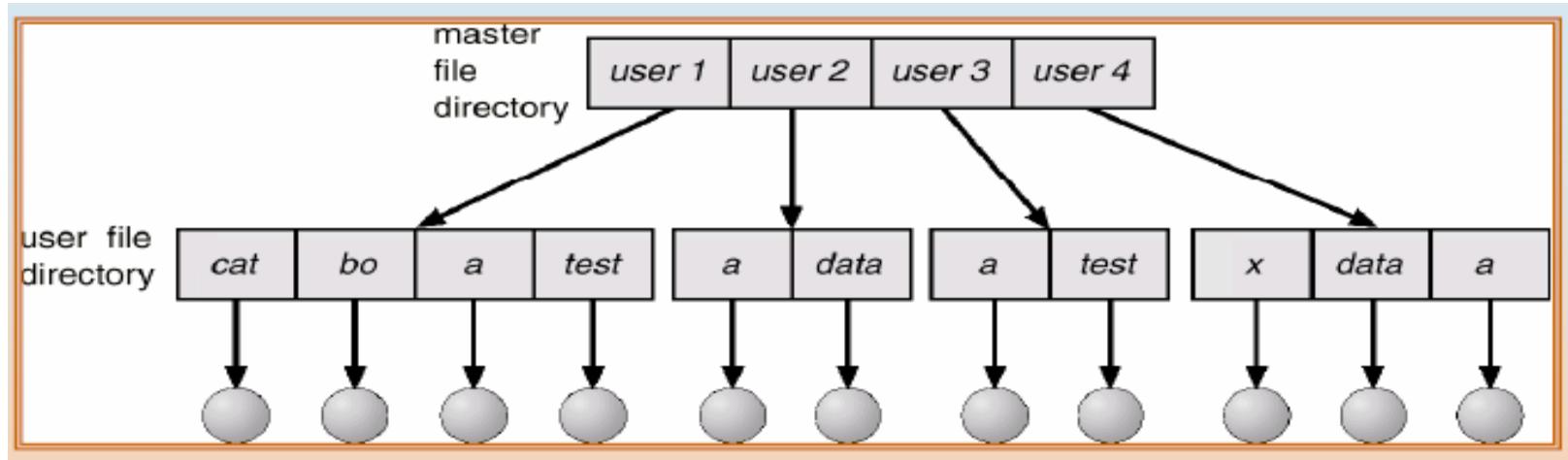


Figure : Two-level directory structure.

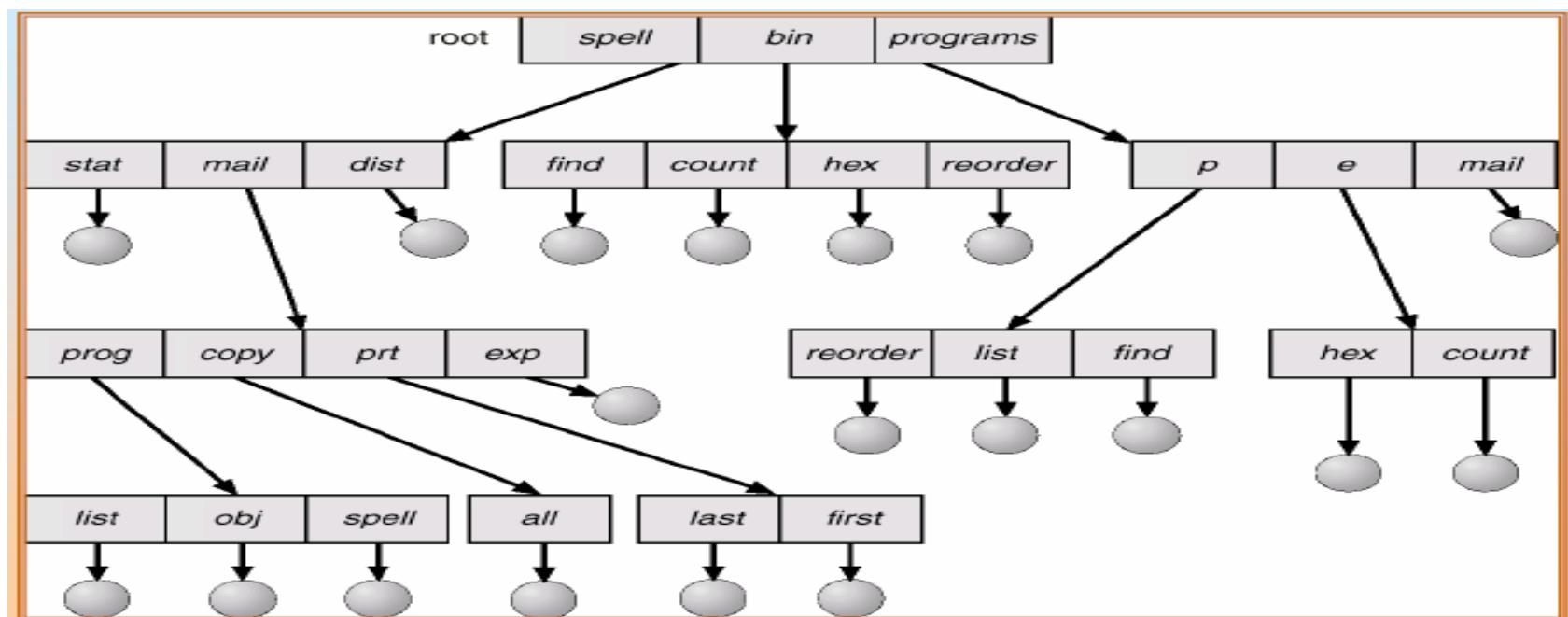


Fig: Tree structure directory

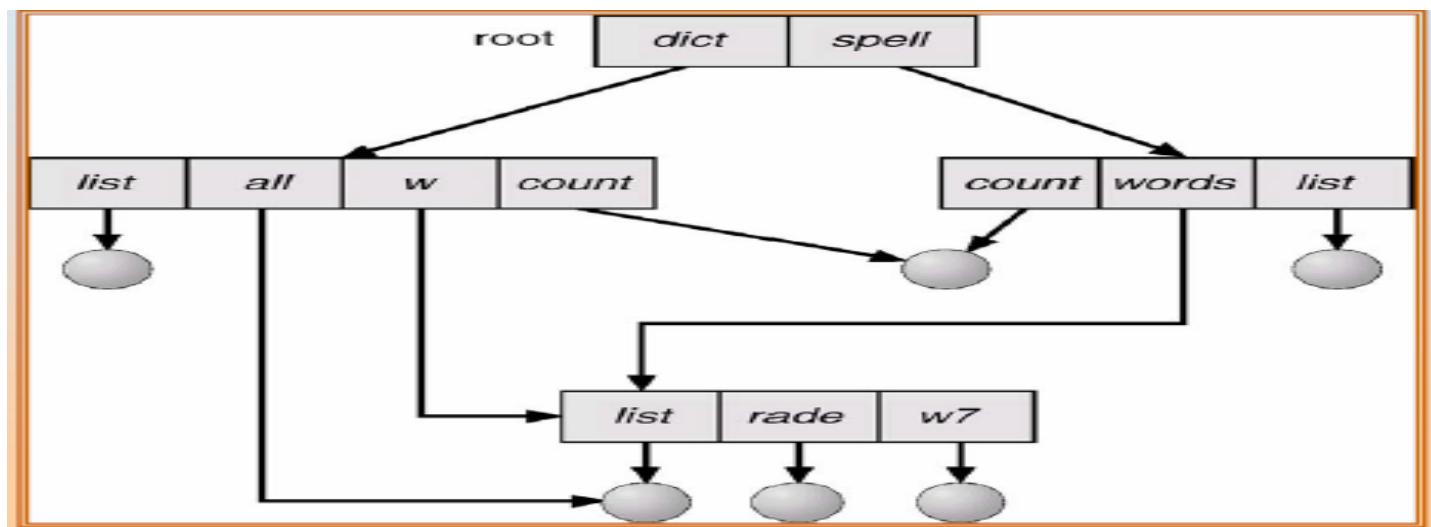


Fig: Acyclic-graph directory structure

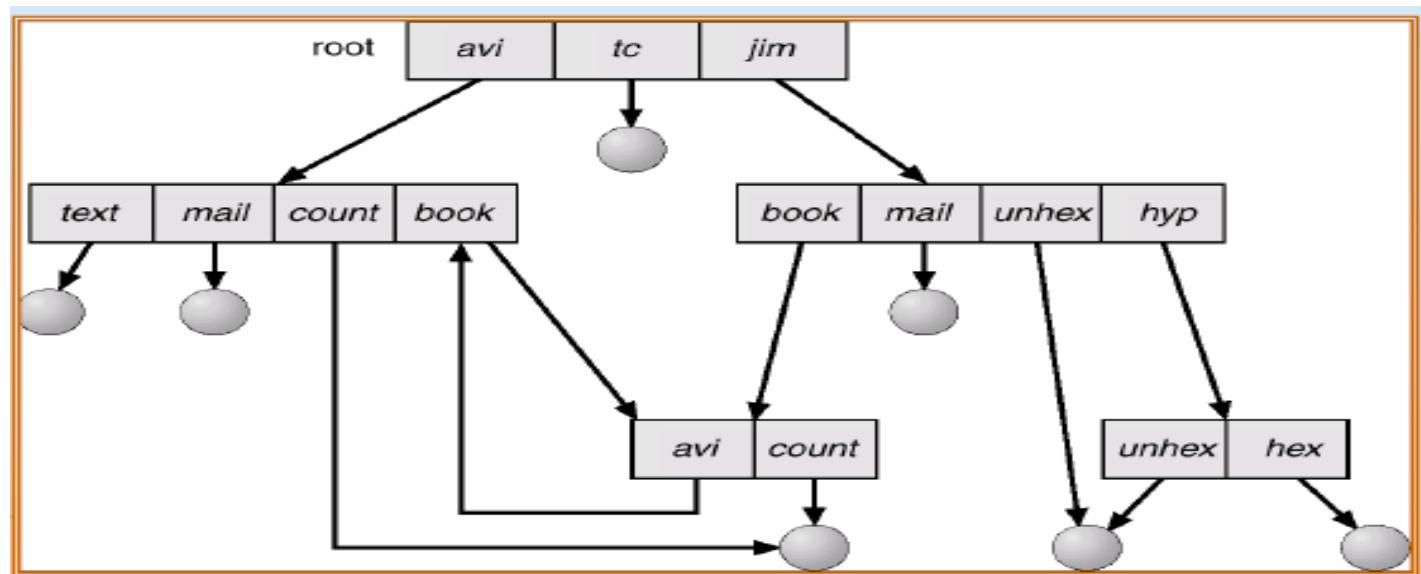


Fig: General graph directory

Protection

- When information is kept in a computer system, one major concern is **protection**, or guarding against improper access.
- File owner/creator should be able to control:
 - what can be done
 - by whom
- Types of access
 - Read: Read from the file.
 - Write: Write or rewrite the file.
 - Execute: Load the file into memory and execute it.
 - Append: Write new information at the end of the file.
 - Delete: Delete the file
 - List: List the name and attributes of the file.

Access Lists and Groups

- Mode of access: read, write, execute
- Three classes of users

RWX

a) **owner access** 7 \Rightarrow 1 1 1

RWX

b) **group access** 6 \Rightarrow 1 1 0

RWX

c) **public access** 1 \Rightarrow 0 0 1

ACL(Access Control List)

- The technique which consists of associating with each object an (ordered) list containing all the domains that may access the object, and how. This list is called the **Access Control List or ACL** and is illustrated in Fig. 1.
- 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.
- Here we see three processes, each belonging to a different domain. *A, B, and C, and three files F1, F2, and F3. For simplicity, we will assume that each domain corresponds to exactly one user, in this case, users A, B, and C. Often in the security literature, the users are called subjects or principals, to contrast them with the things owned, the objects, such as files.*

ACL(contd...)

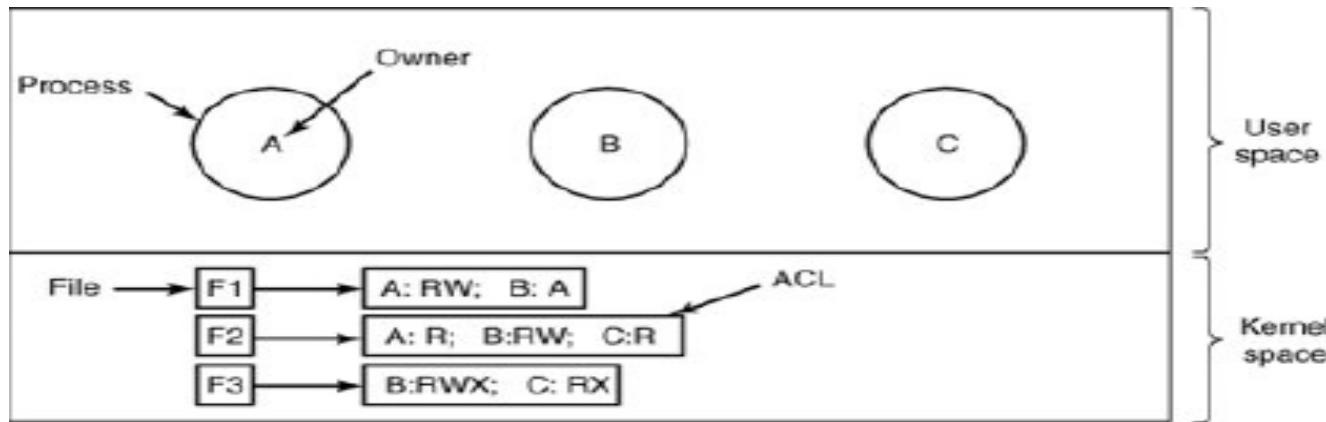


Figure 1. Use of access control lists to manage file access.

- Each file has an ACL associated with it. File *F1* has two entries in its ACL (separated by a semicolon). The first entry says that any process owned by user *A* may read and write the file. The second entry says that any process owned by user *B* may read the file. All other accesses by these users and all accesses by other users are forbidden. Note that the rights are granted by user, not by process. As far as the protection system goes, any process owned by user *A* can read and write file *F1*. It does not matter if there is one such process or 100 of them. It is the owner, not the process ID, that matters. File *F2* has three entries in its ACL: *A*, *B*, and *C* can all read the file, and in addition *B* can also write it. No other accesses are allowed. File *F3* is apparently an executable program, since *B* and *C* can both read and execute it. *B* can also write it.

File-System Structure

- The file system structure is the most basic level of organization in an operating system. The way an operating system interacts with its users, applications, and security model nearly always depends on how the operating system organizes files on storage devices. Providing a common file system structure ensures users and programs can access and write files.
- File systems break files down into two logical categories:
 - Shareable vs. unsharable files
 - Variable vs. static files
- Shareable files can be accessed locally and by remote hosts; unsharable files are only available locally.
- Variable files, such as documents, can be changed at any time; static files, such as binaries, do not change without an action from the system administrator.

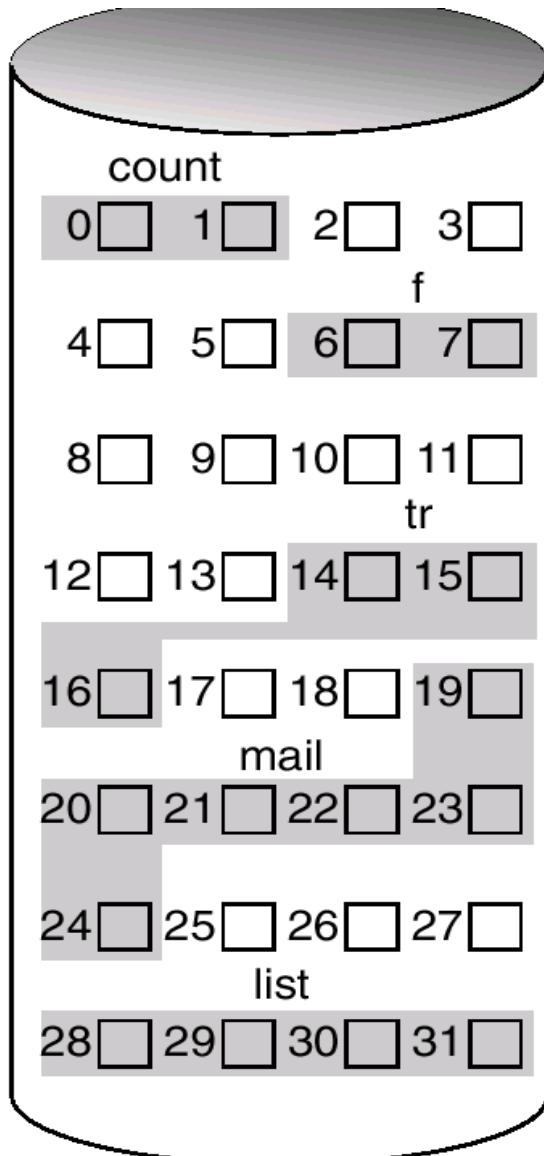
Methods of Allocation

- File allocation
 - Problem of allocating and freeing space on secondary storage is somewhat like that experienced in primary storage allocation under variable-partition multiprogramming
- An allocation method refers to how disk blocks are allocated for files:
- Contiguous allocation
- Linked allocation
- Indexed allocation

Contiguous Allocation

- The **contiguous-allocation** method requires each file to occupy a set of contiguous blocks on the disk. Disk addresses define a linear ordering on the disk. With this ordering, assuming that only one job is accessing the disk, accessing block $b + 1$ after block b normally requires no head movement. When head movement is needed (from the last sector of one cylinder to the first sector of the next cylinder), it is only one track. Thus, the number of disk seeks required for accessing contiguously allocated files is minimal, as is seek time when a seek is finally needed. It access randomly and is wasteful of sapce.

Contiguous Allocation of Disk Space

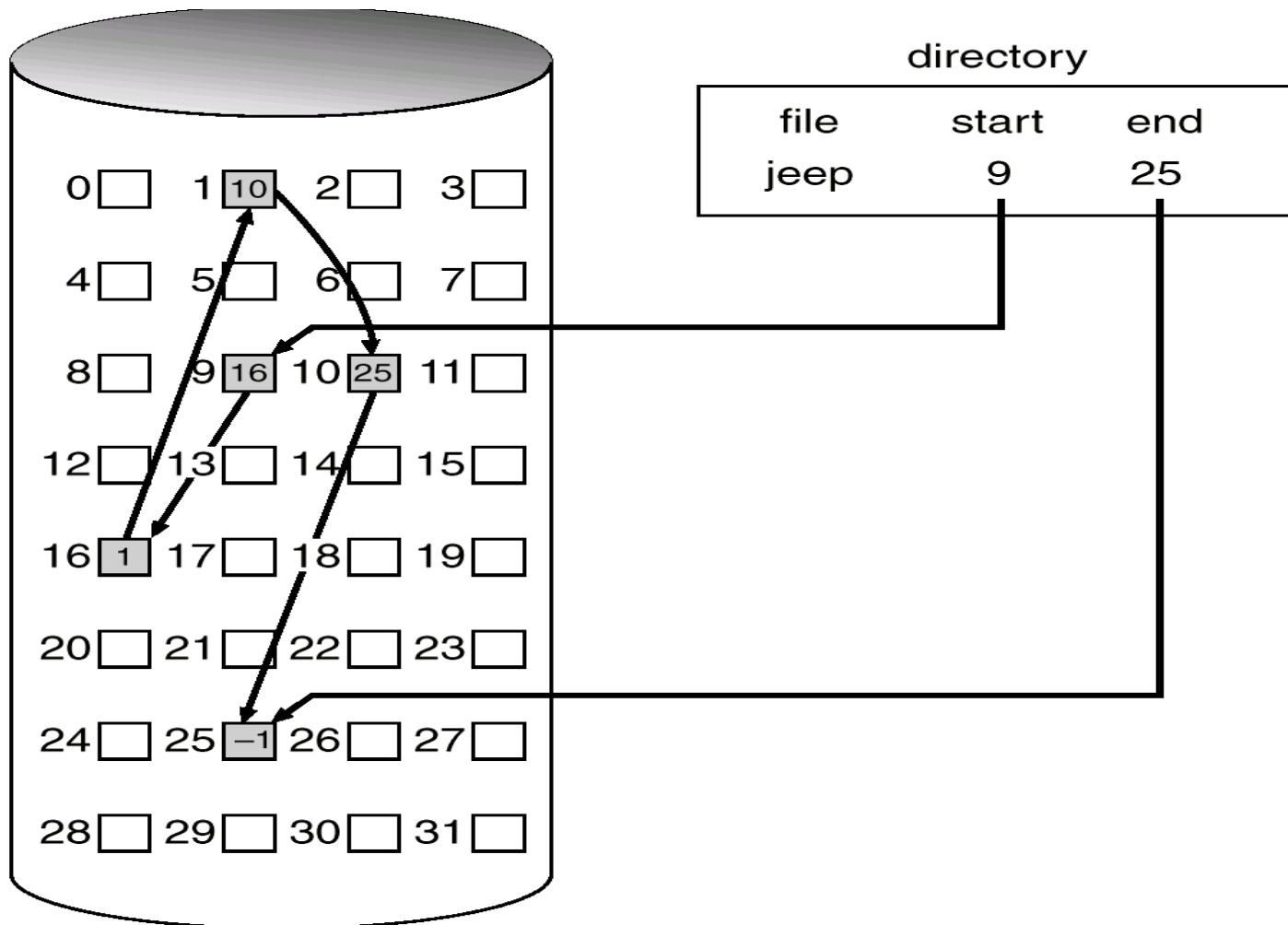


directory		
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Linked List Allocation

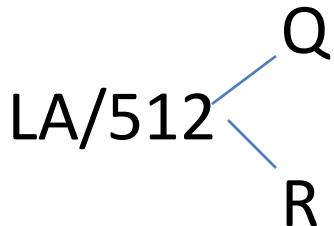
- With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. For example, a file of five blocks might start at block 9, continue at block 16, then block 1, block 10, and finally block 25 (Figure 12.6). Each block contains a pointer to the next block. These pointers are not made available to the user. It does not access randomly and provide free space management system so there is no waste of space.

Linked Allocation



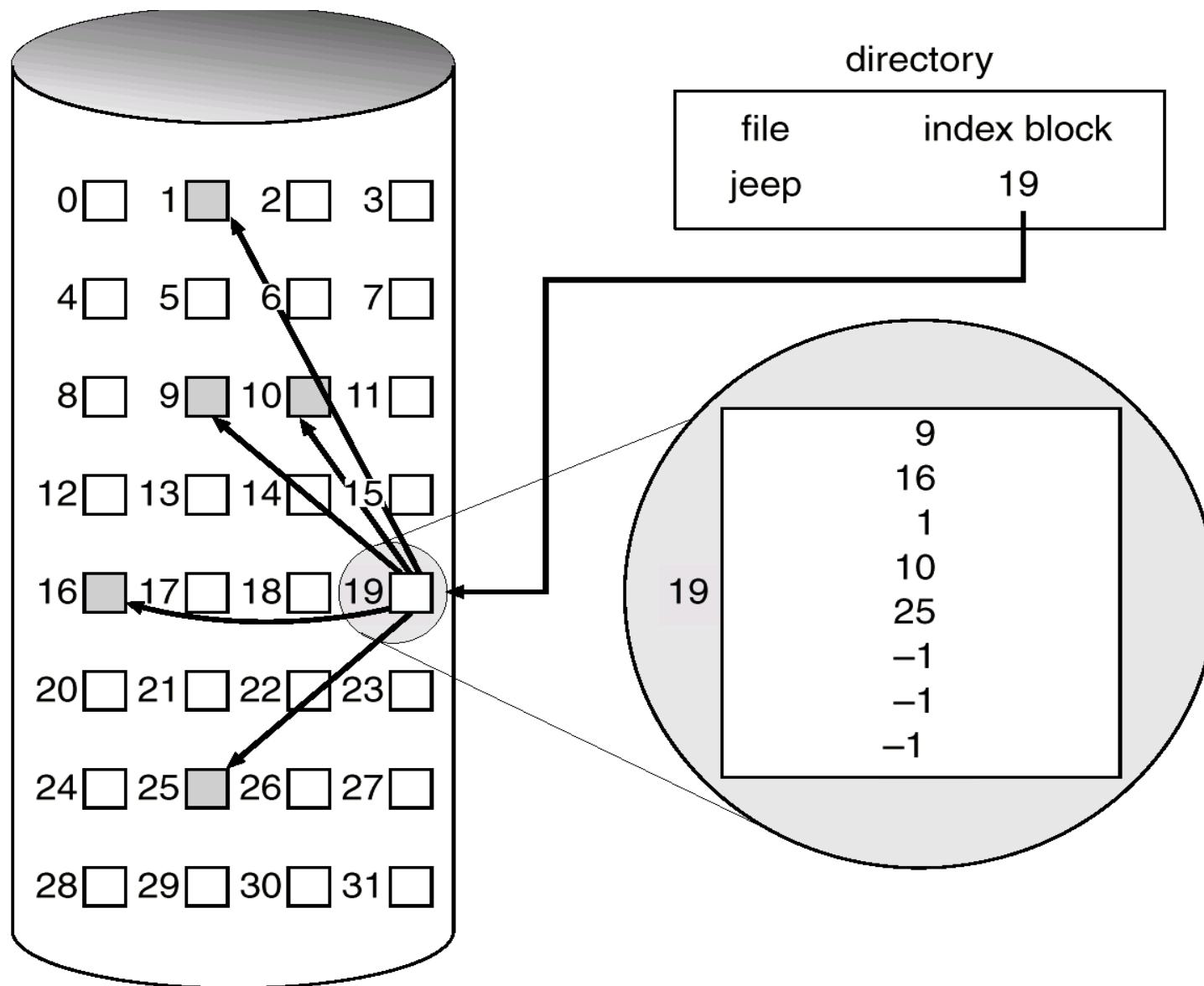
Index Allocation

- Brings all pointers together into the *index block*.
- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block.
- Mapping from logical to physical in a file of maximum size of 256K words and block size of 512 words. We need only 1 block for index table.



- Q = displacement into index table
- R = displacement into block

Example of Indexed Allocation



Free-Space Management

- Since there is only a limited amount of disk space, it is necessary to reuse the space from deleted files for new files, if possible.
- To keep track of free disk space, the system maintains a free-space list. The free-space list records all disk blocks that are free - those not allocated to some file or directory. To create a file, the free-space list has to be searched for the required amount of space, and allocate that space to a new file. This space is then removed from the free-space list. When a file is deleted, its disk space is added to the free-space list.

Bit-Vector

- The free-space list is implemented as a bit map or bit vector. Each block is represented by a 1 bit. If the block is free, the bit is 0; if the block is allocated, the bit is 1.
- For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free, and the rest of the blocks are allocated. The free-space bit map would be:
11000011000000111001111110001111...
- The main advantage of this approach is that it is relatively simple and efficient to find n consecutive free blocks on the disk.

Free-Space Management(contd..)

Linked List

- Another approach is to link all the free disk blocks together, keeping a pointer to the first free block. This block contains a pointer to the next free disk block, and so on. In the previous example, a pointer could be kept to block 2, as the first free block. Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, and so on. This scheme is not efficient; to traverse the list, each block must be read, which requires substantial I/O time.

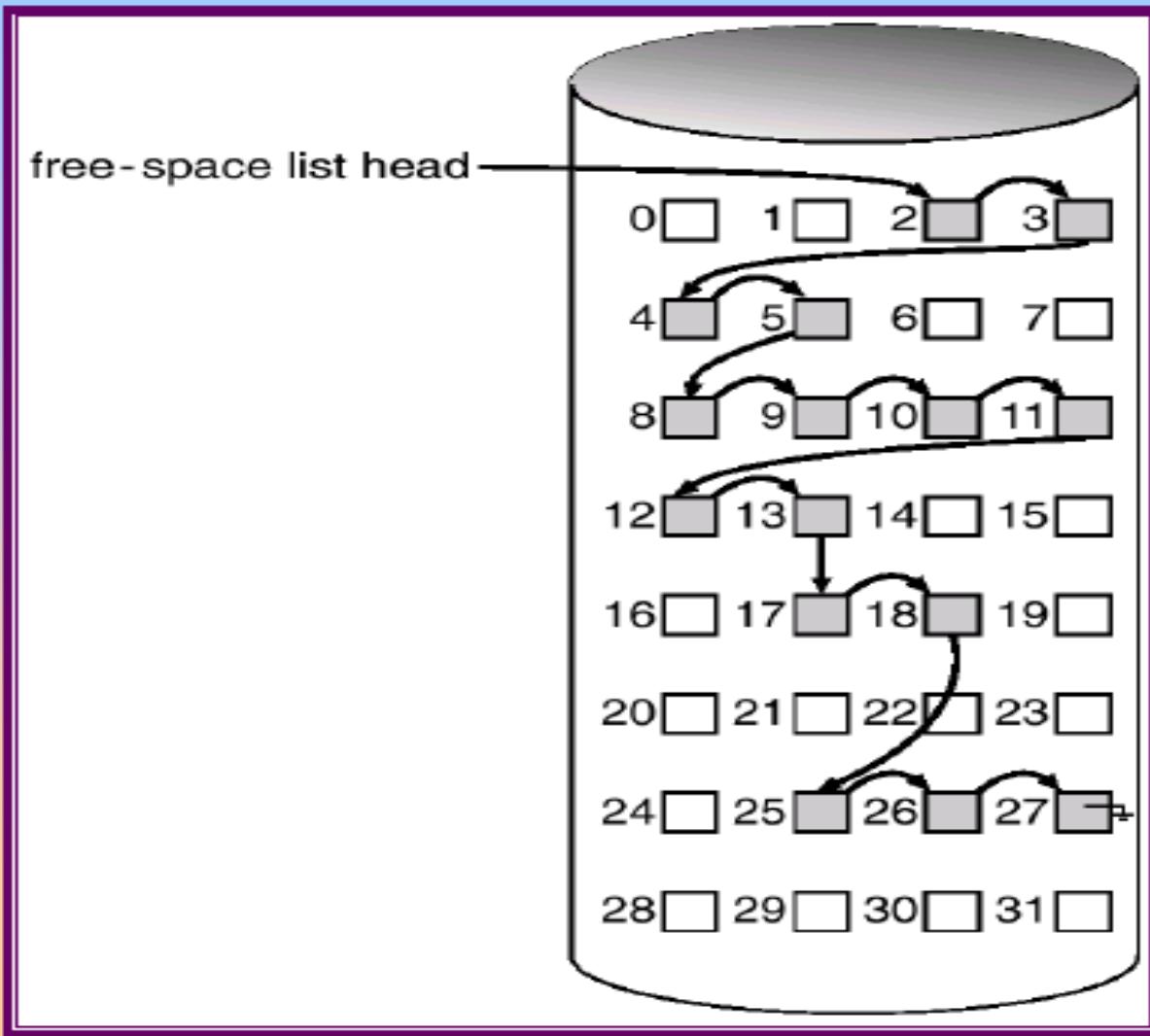
Grouping

- A modification of the free-list approach is to store the addresses of n free blocks in the first free block. The first n-1 of these are actually free. The last one is the disk address of another block containing addresses of another n free blocks. The importance of this implementation is that addresses of a large number of free blocks can be found quickly.

Counting

- Another approach is to take advantage of the fact that, generally, several contiguous blocks may be allocated or freed simultaneously, particularly when contiguous allocation is used. Thus, rather than keeping a list of free disk addresses, the address of the first free block is kept and the number n of free contiguous blocks that follow the first block. Each entry in the free-space list then consists of a disk address and a count. Although each entry requires more space than would a simple disk address, the overall list will be shorter, as long as the count is generally greater than 1.

Linked Free Space List on Disk



Directory Implementation

- Directories need to be fast to search, insert, and delete, with a minimum of wasted disk space.

Linear List

- A linear list is the simplest and easiest directory structure to set up, but it does have some drawbacks.
- Finding a file (or verifying one does not already exist upon creation) requires a linear search.
- Deletions can be done by moving all entries, flagging an entry as deleted, or by moving the last entry into the newly vacant position.
- Sorting the list makes searches faster, at the expense of more complex insertions and deletions.
- A linked list makes insertions and deletions into a sorted list easier, with overhead for the links.
- More complex data structures, such as B-trees, could also be considered.

Hash Table

- A hash table can also be used to speed up searches.
- Hash tables are generally implemented ***in addition to*** a linear or other structure.
- Only good if entries are fixed size, or use chained-overflow method.

Recovery

Consistency Checking

- compares data in directory structure with data blocks on disk, and tries to fix inconsistencies . Can be slow and sometimes fails
- The allocation and free-space management algorithms dictate what types of problems the checker can find, and how successful it will be in fixing those problems. For instance, if linked allocation is used and there is a link from any block to its next block, then the entire file can be reconstructed from the data blocks, and the directory structure can be recreated.

Backup and Restore

- In order to recover lost data in the event of a disk crash, it is important to conduct backups regularly. Files should be copied to some removable medium, such as magnetic tapes, floppy disk or optical disk.
- Recovery from the loss of an individual file, or of an entire disk, may then be a matter of **restoring** the data from backup.
- To minimize the copying needed, we can use information from each file's directory entry.

Nepal College Of Information Technology

Balkumari, lalitpur

Applied Operating System Assignment Solution

Prepared By: Amit Kr. Shrivastava

Q.1 List five services provided by an operating system that are designed to make it more convenient for users to use the computer system. In what cases it would be impossible for user-level programs to provide these services? Explain.

Answer: **Program execution.** The operating system loads the contents (or sections) of a file into memory and begins its execution. A user-level program could not be trusted to properly allocate CPU time.

- **I/O operations.** Disks, tapes, serial lines, and other devices must be communicated with at a very low level. The user need only specify the device and the operation to perform on it, while the system converts that request into device- or controller-specific commands. User-level programs cannot be trusted to only access devices they should have access to and to only access them when they are otherwise unused.

- **File-system manipulation.** There are many details in file creation, deletion, allocation, and naming that users should not have to perform. Blocks of disk space are used by files and must be tracked. Deleting a file requires removing the name file information and freeing the allocated blocks. Protections must also be checked to assure proper file access. User programs could neither ensure adherence to protection methods nor be trusted to allocate only free blocks and deallocate blocks on file deletion.

- **Communications.** Message passing between systems requires messages be turned into packets of information, sent to the network controller, transmitted across a communications medium, and reassembled by the destination system. Packet ordering and data correction must take place. Again, user programs might not coordinate access to the network device, or they might receive packets destined for other processes.

- **Error detection.** Error detection occurs at both the hardware and software levels. At the hardware level, all data transfers must be inspected to ensure that data have not been corrupted in transit. All data on media must be checked to be sure they have not changed since they were written to the media. At the software level, media

must be checked for data consistency; for instance, do the number of allocated and unallocated blocks of storage match the total number on the device. There, errors are frequently process-independent (for instance, the corruption of data on a disk), so there must be a global program (the operating system) that handles all

types of errors. Also, by having errors processed by the operating system, processes need not contain code to catch and correct all the errors possible on a system.

Q.2 What are the advantages and disadvantages of using the same systemcall interface for manipulating both files and devices?

Answer: Each device can be accessed as though it was a file in the file system. Since most of the kernel deals with devices through this file interface, it is relatively easy to add a new device driver by implementing the hardware-specific code to support this abstract file interface. Therefore, this benefits the development of both user program code, which can be written to access devices and files in the same manner, and device driver code, which can be written to support a well-defined API. The disadvantage with using the same interface is that it might be difficult to capture the functionality of certain devices within the context of the file access API, thereby either resulting in a loss of functionality or a loss of performance. Some of this could be overcome by the use of ioctl operation that provides a general purpose interface for processes to invoke operations on devices.

Q.3 Describe the actions taken by a kernel to context-switch between processes.

Answer: In general, the operating system must save the state of the currently running process and restore the state of the process scheduled to be run next. Saving the state of a process typically includes the values of all the CPU registers in addition to memory allocation. Context switches must also perform many architecture-specific operations, including flushing data and instruction caches.

Q.4 Provide two programming examples in which multithreading does not provide better performance than a single-threaded solution.

Answer: (1) Any kind of sequential program is not a good candidate to be threaded. An example of this is a program that calculates an individual tax return. (2) Another example is a "shell" program such as the C-shell or Korn shell. Such a program must closely monitor its own working space such as open files, environment variables, and current working directory.

Q.5 Describe the actions taken by a thread library to context switch between user-level threads.

Answer: Context switching between user threads is quite similar to switching between kernel threads, although it is dependent on the threads library and how it maps user threads to kernel threads. In general, context switching between user threads involves taking a user thread of its LWP and replacing it with another thread. This act typically involves saving and restoring the state of the registers.

Q.6 Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single-threaded solution on a single-processor system?

Answer: When a kernel thread suffers a page fault, another kernel thread can be switched in to use the interleaving time in a useful manner. A single-threaded process, on the other hand, will not be capable of performing useful work when a page fault takes place. Therefore, in scenarios where a program might suffer from frequent page faults or has to wait for other system events, a multi-threaded solution would perform better even on a single-processor system.

Q.7 Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single-processor system?

Answer: A multithreaded system comprising of multiple user-level threads cannot make use of the different processors in a multiprocessor system simultaneously. The operating system sees only a single process and will not schedule the different threads of the process on separate processors. Consequently, there is no performance benefit associated with executing multiple user-level threads on a multiprocessor system.

Q.8 Discuss how the following pairs of scheduling criteria conflict in certain settings.

- a. CPU utilization and response time
- b. Average turnaround time and maximum waiting time
- c. I/O device utilization and CPU utilization

Answer:

- CPU utilization and response time: CPU utilization is increased if the overheads associated with context switching is minimized. The context switching overheads could be lowered by performing context switches infrequently. This could however result in increasing the response time for processes.

- Average turnaround time and maximum waiting time: Average turnaround time is minimized by executing the shortest tasks first. Such a scheduling policy could however starve long-running tasks and thereby increase their waiting time.
- I/O device utilization and CPU utilization: CPU utilization is maximized by running long-running CPU-bound tasks without performing context switches. I/O device utilization is maximized by scheduling I/O-bound jobs as soon as they become ready to run, thereby incurring the overheads of context switches.

Q.9 What is the meaning of the term busy waiting? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer.

Answer: *Busy waiting* means that a process is waiting for a condition to be satisfied in a tight loop without relinquishing the processor. Alternatively, a process could wait by relinquishing the processor, and block on a condition and wait to be awakened at some appropriate time in the future. Busy waiting can be avoided but incurs the overhead associated with putting a process to sleep and having to wake it up when the appropriate program state is reached.

Q.10 Consider the deadlock situation that could occur in the dining-philosophers problem when the philosophers obtain the chopsticks one at a time. Discuss how the four necessary conditions for deadlock indeed hold in this setting. Discuss how deadlocks could be avoided by eliminating any one of the four conditions.

Answer: Deadlock is possible because the four necessary conditions hold in the following manner: 1) mutual exclusion is required for chopsticks, 2) the philosophers tend to hold onto the chopstick in hand while they wait for the other chopstick, 3) there is no preemption of chopsticks in the sense that a chopstick allocated to a philosopher cannot be forcibly taken away, and 4) there is a possibility of circular wait. Deadlocks could be avoided by overcoming the conditions in the following manner: 1) allow simultaneous sharing of chopsticks, 2) have the philosophers relinquish the first chopstick if they are unable to obtain the other chopstick, 3) allow for chopsticks to be forcibly taken away if a philosopher has had a chopstick for a long period of time, and 4) enforce a numbering of the chopsticks and always obtain the lower numbered chopstick before obtaining the higher numbered one.

Q.11 Under what circumstances would a user be better off using a timesharing

system rather than a PC or single-user workstation?

Answer: When there are few other users, the task is large, and the hardware is fast, time-sharing makes sense. The full power of the system can be brought to bear on the user's problem. The problem can be solved faster than on a personal computer. Another case occurs when lots of other users need resources at the same time.

A personal computer is best when the job is small enough to be executed reasonably on it and when performance is sufficient to execute the program to the user's satisfaction.

Q.12 Answer: a. The four necessary conditions for a deadlock are (1) mutual exclusion; (2) hold-and-wait; (3) no preemption; and (4) circular wait.

The mutual exclusion condition holds as only one car can occupy a space in the roadway. Hold-and-wait occurs where a car holds onto their place in the roadway while they wait to advance in the roadway. A car cannot be removed (i.e. preempted) from its position in the roadway. Lastly, there is indeed a circular wait as each car is waiting for a subsequent car to advance. The circular wait condition is also easily observed from the graphic.

b. A simple rule that would avoid this traffic deadlock is that a car may not advance into an intersection if it is clear they will not be able to immediately clear the intersection.

Q.13 Answer: Follow the note.

Nepal College Of Information Technology
Balkumari, lalitpur

Applied Operating System Assignment Sheet -2 Solution
Prepared By: Amit Kr. Shrivastava

Q.1 Explain the difference between internal and external fragmentation.

Answer: **Internal Fragmentation** is the area in a region or a page that is not used by the job occupying that region or page. This space is unavailable for use by the system until that job is finished and the page or region is released. Internal Fragmentation occurs when allotted memory blocks are of fixed size i.e when paging is employed.

When there are small and non-contiguous memory blocks which cannot be assigned to any process, the problem is termed as **External Fragmentation**. External Fragmentation occurs when allotted memory blocks are of varying size i.e when segmentation is employed. External fragmentation can be reduced by compaction or shuffle memory contents to place all free memory together in one large block. To make compaction feasible, relocation should be dynamic.

Q.2 Given five memory partitions of 100 KB, 500 KB, 200 KB, 300 KB, and 600 KB (in order), how would each of the first-fit, best-fit, and worst-fit algorithms place processes of 212 KB, 417 KB, 112 KB, and 426 KB (in order)? Which algorithm makes the most efficient use of memory?

Answer:

- a. First-fit:
 - b. 212K is put in 500K partition
 - c. 417K is put in 600K partition
 - d. 112K is put in 288K partition (new partition 288K = 500K - 212K)
 - e. 426K must wait
- f. Best-fit:
 - g. 212K is put in 300K partition
 - h. 417K is put in 500K partition
 - i. 112K is put in 200K partition
 - j. 426K is put in 600K partition
- k. Worst-fit:
 - l. 212K is put in 600K partition
 - m. 417K is put in 500K partition
 - n. 112K is put in 388K partition
 - o. 426K must wait

In this example, Best-fit turns out to be the best.

Q.3 Why are segmentation and paging sometimes combined into one scheme?

Answer: Segmentation and paging are often combined in order to improve upon each other. Segmented paging is helpful when the page table becomes very large. A large contiguous section of the page table that is unused can be collapsed into a single segment table entry with a page table address of zero. Paged segmentation handles the case of

having very long segments that require a lot of time for allocation. By paging the segments, we reduce wasted memory due to external fragmentation as well as simplify the allocation.

Q.4 Consider the following segment table:

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

What are the physical addresses for the following logical addresses?

- a. 0,430
- b. 1,10
- c. 2,500
- d. 3,400
- e. 4,112

Answer:

- a. $219 + 430 = 649$
- b. $2300 + 10 = 2310$
- c. illegal reference, trap to operating system
- d. $1327 + 400 = 1727$
- e. illegal reference, trap to operating system

Q.5 What is the purpose of paging the page tables?

Answer: In certain situations the page tables could become large enough that by paging the page tables, one could simplify the memory allocation problem (by ensuring that everything is allocated as fixed-size pages as opposed to variable-sized chunks) and also enable the swapping of portions of page table that are not currently used.

Q.6 What is the minimum number of page faults for an optimal page replacement strategy for the following reference string with four page frames? Now repeat this problem for LRU.

1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2.

Answer: 11 page faults(for optimal)

13 page faults(for LRU)

Q.7 What is the cause of thrashing? How does the system detect thrashing? Once it detects thrashing, what can the system do to eliminate this problem?

Answer: Thrashing is caused by underallocation of the minimum number of pages required by a process, forcing it to continuously page fault. The system can detect thrashing by evaluating the level of CPU utilization as compared to the level of multiprogramming. It can be eliminated by reducing the level of multiprogramming.

Q.8 A computer has four page frames. The time of loading, time of last access, and the **R** and **M** bits for each page are as shown below (the times are in clock ticks):

Page	Loaded	Last ref.	R	M
0	126	280	1	0

1	230	265	0	1
2	140	270	0	0
3	110	285	1	1

- (a) Which page will NRU replace?
- (b) Which page will FIFO replace?
- (c) Which page will LRU replace?
- (d) Which page will second chance replace?

Answer: NRU removes page 2. FIFO removes page 3. LRU removes page 1. Second chance removes page 2.

Q.9 Consider a file system that uses a modified contiguous-allocation scheme with support for extents. A file is a collection of extents, with each extent corresponding to a contiguous set of blocks. A key issue in such systems is the degree of variability in the size of the extents. What are the advantages and disadvantages of the following schemes?

- a. All extents are of the same size, and the size is predetermined.
- b. Extents can be of any size and are allocated dynamically.
- c. Extents can be of a few fixed sizes, and these sizes are predetermined.

Answer: If all extents are of the same size, and the size is predetermined, then it simplifies the block allocation scheme. A simple bit map or free list for extents would suffice. If the extents can be of any size and are allocated dynamically, then more complex allocation schemes are required. It might be difficult to find an extent of the appropriate size and there might be external fragmentation. One could use the Buddy system allocator discussed in the previous chapters to design an appropriate allocator. When the extents can be of a few fixed sizes, and these sizes are predetermined, one would have to maintain a separate bitmap or free list for each possible size. This scheme is of intermediate complexity and of intermediate flexibility in comparison to the earlier schemes.

Q.10 Fragmentation on a storage device could be eliminated by recompaction of the information. Typical disk devices do not have relocation or base registers (such as are used when memory is to be compacted), so how can we relocate files? Give three reasons why recompacting and relocation of files often are avoided.

Answer: Relocation of files on secondary storage involves considerable overhead—data blocks would have to be read into main memory and written back out to their new locations. Furthermore, relocation registers apply only to *sequential* files, and many disk files are not sequential. For this same reason, many new files will not require contiguous disk space; even sequential files can be allocated noncontiguous blocks if links between logically sequential blocks are maintained by the disk system.

Q.11 Suppose that a disk drive has 5000 cylinders, numbered 0 to 4999. The drive is currently serving a request at cylinder 143, and the previous request was at cylinder 125. The queue of pending requests, in FIFO order, is

86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130

Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests, for each of the following disk-scheduling algorithms?

- a. FCFS
- b. SSTF
- c. SCAN
- d. C-SCAN

Answer:

- a. The FCFS schedule is 143, 86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130. The total seek distance is 7081.
- b. The SSTF schedule is 143, 130, 86, 913, 948, 1022, 1470, 1509, 1750, 1774. The total seek distance is 1745.
- c. The SCAN schedule is 143, 913, 948, 1022, 1470, 1509, 1750, 1774, 4999, 130, 86. The total seek distance is 9769.
- e. The C-SCAN schedule is 143, 913, 948, 1022, 1470, 1509, 1750, 1774, 4999, 86, 130. The total seek distance is 4986.

Q.12 What are the advantages and disadvantages of supporting memory mapped I/O to device control registers?

Answer: The advantage of supporting memory-mapped I/O to device control registers is that it eliminates the need for special I/O instructions from the instruction set and therefore also does not require the enforcement of protection rules that prevent user programs from executing these I/O instructions. The disadvantage is that the resulting flexibility needs to be handled with care; the memory translation units need to ensure that the memory addresses associated with the device control registers are not accessible by user programs in order to ensure protection.

Q.13 Explain the difference between deadlock, livelock and starvation.

Answer: A deadlock occurs when a set of processes are blocked waiting for an event that only some other process in the set can cause. On the other hand, processes in a livelock are not blocked. Instead, they continue to execute checking for a condition to become true that will never become true. Thus, in addition to the resources they are holding, processes in livelock continue to consume precious CPU time. Finally, starvation of a process occurs because of the presence of other processes as well as a stream of new incoming processes that end up with higher priority than the process being starved. Unlike deadlock or livelock, starvation can terminate on its own, e.g. when existing processes with higher priority terminate and no new processes with higher priority arrive.

Q.14 Differentiate between segmentation and paging.

Answer:

No.	Key	Paging	Segmentation
1	Memory Size	In Paging, a process address space is broken into fixed sized blocks called pages.	In Segmentation, a process address space is broken in varying sized blocks called sections.
2	Accountability	Operating System divides	Compiler is responsible to

No.	Key	Paging	Segmentation
		the memory into pages.	calculate the segment size, the virtual address and actual address.
3	Size	Page size is determined by hardware.	Section size is determined by the user.
4	Speed	Paging technique is faster in terms of memory access.	Segmentation is slower than paging.
5	Fragmentation	Paging can cause internal fragmentation as some pages may go underutilized.	Segmentation can cause external fragmentation as some memory block may not be used at all.
6	Logical Address	During paging, a logical address is divided into page number and page offset.	During segmentation, a logical address is divided into section number and section offset.

Q.15 Explain Producer Consumer Problem using semaphore.

Answer: Refer Lecture note.