

**Noorul Islam College of Engineering**  
Department of Information Technology  
II M.Sc Information Technology  
Semester IV

**Operating Systems (BCS246)**

Part B Questions and Answers

**Unit I**

**1. Explain about mainframe systems?**

These systems were the first used to tackle many commercial and scientific applications.

**Batch systems**

In early systems the user did not interact directly with the computer systems. Rather, the user prepared a job – which consisted of the program, the data, and some control information about the nature of the job – and submitted it to the computer operator. The job was originally in the form of punch cards.

The operating system task was to transfer control automatically from one job to the next. The operating system was always resident in memory. To speed up processing, operators **batched** together jobs and ran them through the computers. The output from each job would be sent back to the programmer.

In this execution environment, the CPU is often idle, because the speeds of the mechanical I/O devices are intrinsically slower than are those of electronic devices. A fast card reader read 1200 cards per minute.

The introduction of disk technology allowed the operating system to keep all jobs on a disk. With direct access to several jobs, the operating system could perform **job scheduling**, to use resources and perform tasks efficiently.

**Multiprogrammed systems**

A single user cannot, in general, keep either the CPU or the I/O devices busy at all times. Multiprogramming increases CPU utilization by organizing jobs so that the CPU always has one to execute.

**Procedure**

The operating system keeps several jobs in memory simultaneously. It picks and begins to execute one of the jobs in the memory. In multiprogramming system, the operating

system simply switches to, and executes, another job. When a job needs to wait, the CPU is switched to another job, and so on. Eventually, the first job finishes waiting and gets the CPU back. As long as one job needs to execute, the CPU is never idle.

In multiprogrammed operating system all the jobs that enter the system are kept in the job pool. If several jobs are ready to be brought into memory, and there is not enough room for all of them, then the system must choose among them. Making this decision is **job scheduling**. Having several programs in memory at the same time, the system must choose among them. Making this decision is **CPU scheduling**.

## **Time-sharing Systems**

**Time sharing (or multitasking)** is a logical extension of multiprogramming. The CPU executes multiple jobs switching among them, but the switches occurs so frequently that the users can interact with each program while it is running.

An **interactive (or hands-on)** computer system provides direct communication between the user and the system.

A time-shared operating system allows many users to share the computer simultaneously. As the system switches rapidly from one user to the next, each user is given an impression that the entire system is dedicated to one user.

Time-shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer. A program loaded into memory and executing is commonly referred to as a **process**.

Time-sharing operating systems are even more complex than multiprogrammed operating system. To obtain a reasonable response time, jobs may have to be swapped in and out of main memory to the disk. A common method for achieving this goal is **virtual memory**. It is a technique that allows the execution of a job that may have not be completely in memory. The main advantage of the virtual-memory scheme is that programs can be larger than **physical memory**.

Time-sharing systems must also provide a file system that resides on a collection of disks. Time-sharing system provides a mechanism for concurrent execution. To ensure orderly execution, the system must provide mechanism for job synchronization and communication.

## **2. Explain about multiprocessor systems?**

Multiprocessor systems have more than one processor in close communication, sharing the computer bus, the clock and sometimes memory and peripheral devices.

Advantage of multiprocessor systems

1. Increased throughput. By increasing the number of processors, we hope to get more work done in less time. The contention (argument) for shared resources lowers the expected gain from additional processors.
2. Economy of scale. Multiprocessor systems can save more money than multiple single-processor systems, because they can share peripherals, mass storage, and power supplies.
3. Increased reliability. The functions can be distributed properly among several processors, and then the failure of one processor will not halt the system, only slow it down. This ability to continue providing service proportional to the level of surviving hardware is called **graceful degradation**. System designed for graceful degradation are also called **fault tolerant**.

The most common multi-processor systems now use **symmetric multiprocessing (SMP)**, in which each processor runs an identical copy of the operating system, and these copies communicate with one another as needed. Some systems are **asymmetric multiprocessing**, in which each processor is assigned a specific task. The master processor controls the system, the other processor looks the master.

SMP means that all processor are peers; no master-slave relationship exists between processors. Each processor concurrently runs a copy of the operating system. An example is the Encor's version of UNIX for the Multimax computer. This form of multiprocessor systems will allow processes and resources – such as memory – to be shared dynamically among the various processors.

The difference between symmetric and asymmetric multiprocessing may be the result of either hardware or software.

As microprocessors become less expensive and more powerful, additional operating-system functions are off-loaded to slave processors (or **back ends**).

### 3. Briefly discuss on computing environments?

Overview of how different operating systems are used in a variety of computing environment settings.

#### **Traditional computing**

Web technologies are stretching the boundaries of traditional computing. Companies implement **portals** which provide web accessibility to their internal servers. **Network computers** are essentially terminals that understand web based computing. Handheld computers can also connect to **wireless networks** to use the company's web portal.

Most users had single computers with modem connection to the office, the Internet, or both. Network allows more access to more data at a company or from Web. Some homes even have **firewalls** to protect these home environments from security breaches.

## Web based computing

The Web has become more access by a wider variety of devices. PCs are still the most prevalent access devices, with workstations, handheld PDAs and even cell phones also providing access.

Devices that were networked now have faster network connectivity, either by improved networking technology, optimal network implementation code or both. The implementation of web-based computing has given rise to new categories of devices, such as **load balancers** which distribute network connections among pool or similar servers. Operating system like Windows 2000, this can act as web servers as well as clients.

## Embedded computing

Embedded computers run embedded real-time operating system. These devices are found everywhere from car engines and manufacturing robots to VCRs and microwave ovens. They have lacking advanced features such as virtual memory, and even disks. They have little or no user interface.

In the use of embedded systems entire houses can be computerized, so that a central computer – either general-purpose computer or an embedded system – can control heating and lighting, alarm systems and even coffee makers.

## 4. What are the operations of processes?

Process creation and termination.

### Process creation

A process may create several new processes. The creating process is called a **parent** process, where as the new process is called the **children** processes. When a processes creates a sub processes, the parent may have to partition its resources among its children.

When a process creates a new processes, two possibilities exists in terms of execution.

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

In regarding the address space

1. The child process is a duplicate of the parent process.
2. The child process has a program loaded into it.

In UNIX, each process is identified by its **process identifier**, which is a unique integer. A new process is created by the **fork system call**. The new process consists of a

copy of the address space of the original process. The **execvp** system call is used to replace the processes memory space with a new program.

Fig: 4.8 - Page 106 – The C program forking a separate process.

### Process termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit system call. At that time the process may return data to its parent processes. All the resources are deallocated by the operating system.

A parent may terminate the execution of one of its children's for a variety of reasons.

- ❖ The child has exceeded its usage of some of the resources that it has been allocated.
- ❖ The task assigned to the child is no longer required.
- ❖ The parent is exiting. On such systems, if processes terminates, then all its children must also be terminated. This phenomenon, referred to as **cascading termination**.

In UNIX we can terminate processes by using **exit** system call.

### 5. Explain about interprocess communication?

The cooperating processes communicate with each other via an interprocess communication facility. IPC provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. It is particularly useful in a distributed environment.

#### Message passing system

The function of message system is to allow processes to communicate with one another without the need to resort (option) to shared data. Communication among the user processes is accomplished through the passing of messages. An IPC facility provides at least two operations: send and receive.

If processes P and Q want to communicate, they must send messages to and receive messages from each other; a **communication link** must exist between them.

Methods for logically implementing links

- Direct or indirect communication.
- Symmetric or asymmetric communication.
- Automatic or explicit buffering.
- Sent by copy or send by reference.
- Fixed-sized or variable-sized messages.

## Naming

To refer the processes involved in communication.

### Direct communication

With **direct communication** each process that wants to communicate must explicitly name the recipient or sender of the communication. The send and receive primitives are

- Send (P,message) – Send a message to process P.
- Receive (Q,message) – Receive a message from process Q.

A communication link has the following properties.

- ❖ A link is established automatically between every pair of processes that want to communicate.
- ❖ A link is associated with exactly two processes.
- ❖ Exactly one link exists between each pair of processes.

Another scheme in communication employs asymmetry in addressing. Only the sender names the recipient; the recipient is not required to name the sender. The send and receive primitives in this scheme are

- Send (P,message) – Send a message to process P.
- Receive (id,message) – Receive a message from any process; the variable id is the name of the processes with which the communication has taken place.

### Indirect communication

With **indirect communication**, the messages are sent to and received from **mailboxes**, or **ports**. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification. Two processes can communicate only if they share a mailbox. The primitives are

- Send (A,message) – Send a message to mailbox A.
- Receive (A,message) – Receive a message from mailbox A.

A communication link has the following properties.

- ❖ A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- ❖ A link may be associated with more than two processes.
- ❖ Each link corresponding to one mailbox.

A mailbox may be owned either by processes or by the operating system. If the mailbox is owned by the process then we distinguish between the owner (receive message) and the user (send message).

Mailbox owned by the operating system is independent and is not attached to any particular process. The OS allows the process to do the following

- Create a new mailbox
- Send and receive messages through the mailbox.
- Delete a mailbox.

The process that creates a new mailbox is that mailbox's owner by default.

### **Synchronization**

In communication between processes messages passed may be blocking or **nonblocking** – also known as **synchronous** and **asynchronous**.

- Blocking send : The sending process is blocked until the message is received.
- Nonblocking send: The sending process sends the message and resumes operation.
- Blocking receive: The receiver blocks until a message is available.
- Nonblocking receive: The receiver retrieves either a valid message or a null.

### **Buffering**

Messages exchanged by communication processes reside in a temporary queue. Such a queue can be implemented in three ways.

- **Zero capacity:** The queue length is 0. In this case, the sender must block until the recipient receives the message.
- **Bounded capacity:** The queue has finite length n, at most n message can reside in it. If the link is full, the sender must block until space is available in the queue.
- **Unbounded capacity:** The queue has potentially infinite length. The sender never blocks.

## **Unit II**

### **6. What are the multithreading models?**

There are three common types of threading implementations.

#### **Many-to-One model**

The many-to-one model maps many user-level threads to one kernel thread. Thread management is done in user space. One thread can access the kernel at a time. **Green threads** – a thread library available for Solaris 2.

### **One-to-One model**

It maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model. It allows multiple threads to run in parallel on multiprocessors.

Drawback is creating a user thread requires creating the corresponding kernel thread. Windows NT, Windows 2000, and OS/2 implement the one-to-one model.

### **Many-to-Many model**

It multiplexes many user-level threads to a smaller or equal number of kernel threads. It allows the developer to create as many user threads as possible.

The shortcomings of this model are

1. Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.
2. When a thread performs a blocking system call, the kernel can schedule another thread for execution.

Solaris 2, IRIX, HP-UX, and Tru64 UNIX support this model.

## **7. Explain about threading issues?**

### **The fork and exec system calls**

In a multithreaded program of some UNIX systems have chosen to have two versions of fork, one that duplicates all threads and another that duplicates only the thread that invoked the fork system call. If a thread invokes the exec system call, the program specified in the parameter to exec will replace the entire process.

### **Cancellation**

**Thread cancellation** is the task of terminating a thread before it has completed. A thread that is to be cancelled is often referred to as the **target thread**. Cancellation of a target thread may occur in two scenarios.

1. Asynchronous cancellation: One thread immediately terminates the target thread.
2. Deferred cancellation: The target thread can periodically check if it should terminate.

## Signal handling

A signal is used in UNIX systems to notify a process that a particular event has occurred.

1. A signal is generated by the occurrence of a particular event.
2. A generated signal is delivered to a process.
3. Once delivered, the signal must be handled.

Every signal may be handled by one of two possible handlers

1. A default signal handler.
2. A user-defined signal handler.

Every signal has a **default signal handler** that is run by the kernel when handling the signal. This default action may be overridden by **user-defined signal handler** function.

Delivering signals is more complicated in multithreaded programs, as a process may have several threads. In general the following options exist

1. Deliver the signal to the thread to which the signal applies.
2. Deliver the signal to every thread in the process.
3. Deliver the signal to certain threads in the process.
4. Assign a specific thread to receive all signals for the process.

Solaris 2 implements the fourth option. Although Windows 2000 does not explicitly provide support for signals, they can be emulated using **asynchronous procedure calls (APCs)**. The APC facility allows a user thread to specify a function that is to be called when the user thread receives notification of a particular event.

## Thread pools

Creating a separate thread is superior to create a separate process. Multithreaded server has potential problems.

- a) The amount of time required to create the thread prior to serving the request.
- b) If we allow all concurrent requests to be serviced in a new thread, we have not placed a bound on the number of threads concurrently active in the system.

Unlimited threads could exhaust system resources, such as CPU time or memory. One solution to this issue is to use **thread pools**.

The benefits of thread pools are

1. It is usually faster to service a request with an existing thread than waiting to create a thread.

2. A thread pool limits the number of threads that exists at any one point.

### **Thread-specific data**

Threads belonging to a process share the data of the process. Indeed, this sharing of data provides one of the benefits of multithreaded programming. However, each thread might need its own copy of certain data in some circumstances. We will call such data **thread-specific data**.

### **8. What are the various scheduling criteria?**

Many criteria have been suggested for comparing CPU-scheduling algorithms. The criteria include.

- CPU utilization: CPU utilization ranges from 0 to 100 percent. In real system, it should range from 40 to 90 percent.
- Throughput: One measure of work is the number of processes completed per time unit, called **throughput**.
- Turnaround time: The interval from the time of submission of a process to the time of completion is the **turnaround time**. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing in the CPU and doing I/O.
- Waiting time: The CPU scheduling algorithm affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.
- Response time: The measure of the time from the submission of a request until the first response is produced. This measure is called **response time**. It is the amount of time it takes to start responding.

The scheduling criteria are to maximize CPU utilization and throughput, and to minimize turnaround time, waiting time, and response time.

### **9. Explain shortest job first scheduling?**

#### **Shortest-job-first scheduling**

This algorithm associates with each process the length of the next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If two processes have the same length, FCFS scheduling is used to break the tie. The appropriate term is shortest next CPU burst.

The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes. The difficult with the SJF algorithm is knowing the length of the next CPU request.

Let  $t_n$  be the length of the  $n$ th CPU burst, and let  $t_{n+1}$  be our predicted value for the next CPU burst.

#### 10. Briefly discuss on realtime scheduling?

It is divided into two types. **Hard real-time** systems are required to complete a critical task within a guaranteed amount of time. The scheduler then either admits the process, guaranteeing that the process will complete on time, or rejects the request as impossible. This is known as **resource reservation**. Therefore, hard real-time systems are composed of special-purpose software running on hardware dedicated to their critical process.

Software real-time computing is less restrictive. It requires that critical processes receive priority over less fortunate ones.

There are several ways to achieve preemptible to keep dispatch latency low. One is to insert **preemption points** in long-duration system calls that check to see whether a high priority process needs to run. Another method for dealing with preemption is to make the entire kernel preemptible.

In real-time scheduling, the high-priority process would be waiting for a lower-priority one to finish. This situation is known as **priority inversion**. A chain of processes could all be accessing resources that the high-process needs. This problem can be solved via the **priority-inheritance protocol**, in which all these processes inherit the high priority until they are done with the resource in question.

The **conflict phase** of dispatch latency has two components

1. Preemption of any process running in the kernel.
2. Release by low-priority processes resources needed by the high-priority process.

### Unit III

#### 11. Explain about deadlock prevention?

In order for the occurrence of deadlock, the four conditions such as mutual exclusion, hold and wait, no preemption and circular wait must occur. By ensuring that one of these conditions cannot hold, we can prevent the occurrence of deadlock.

##### **Mutual exclusion**

It must hold for nonsharable resources. For example a printer cannot be simultaneously shared by several processes. Sharable resources do not require mutually exclusive access, and thus cannot be involved in deadlock. Read-only files are a good example of a sharable resource.

##### **Hold and wait**

To ensure that hold and wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources. The protocols used are

1. Each process is to be request and be allocated all its resources before it begins execution.
2. Allows a process to request some resources only when the process has none (no other resources).

Example : A process copies data from a tape drive to a disk file, sorts the disk file and then prints the result.

These protocols have two main disadvantages.

1. **Resource utilization** may be low, since many of the resources may be allocated but unused for a long period.
2. **Starvation** (need) is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other processes.

### No preemption

There is no preemption of resources that have already been allocated. The protocols used are

1. 1. If a process is holding some resources and requests another resource that cannot be immediately allocated to it, then all resources currently being held are preempted. In other words, these resources are implicitly released. The process will be restarted only when it can regain its old resources, as well as the new one that it is requesting.
2. If a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not available, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process. If both conditions are not satisfied, the requesting process must wait.

### Circular wait

One way to ensure that that this condition never holds is to impose a total ordering of all resource types, and to require that each process requests resources in an increasing ordering of enumeration. The protocols used to prevent deadlock are

1. Each process can request resources only in an increasing order of enumeration.
2. Whenever a process requests an instance of resource type  $R_j$ , it has released any resources  $R_i$  such that  $F(R_i) \geq F(R_j)$

If these two protocols are used, then the circular-wait condition cannot hold.

The function F should be defined according to the normal order of usage of the resources in a system.

## 12. Briefly discuss on Bankers algorithm?

The resource-allocation graph algorithm is not applicable to a resource-allocation system with multiple instances for each resource type. So we go for bankers algorithm but is less efficient than resource allocation graph scheme.

When a new process enters it must declare the maximum number of instances of each resource type that it may need. When a user request a set of resources, the system must determine whether the allocation of these resource will leave the system in a safe state. If it will the resources are allocated; otherwise the process must wait until some other process releases enough resources.

Let n be the number of processes in the system and m be the number of resource type. The data structure is

- Available: A vector of length m indicates the number of available resources of each type. If  $Available[j] = k$ , there are k instances of resource type  $R_j$  available.
- Max: An  $n \times m$  matrix defines maximum demand of each process. If  $Max[i,j] = k$ , then process  $P_i$  may request at most k instances of resource type  $R_j$ .
- Allocation: A matrix defines the number of resources of each type currently allocated to each process. If  $Allocation[i,j] = k$ , then process  $P_i$  is currently allocated k instances of resource type  $R_j$ .
- Need: a matrix indicates the remaining resource need of each process. If  $Need[i,j] = k$ , then process  $P_i$  may need k more instances of resource type  $R_j$  to complete its task.  
 $Need[i,j] = Max[i,j] - Allocation[i,j]$

To simplify the presentation of the banker's algorithm, a notation is established. Let X and Y be the vectors of length n. We say that  $X \leq Y$  if and only if  $X[i] \leq Y[i]$  for all  $i = 1, 2, \dots, n$ .

### Safety algorithm

This algorithm is for finding out whether or not a system is in a safe state.

1. Let Work and Finish be vectors of length m and n, respectively. Initialize  $Work := Available$  and  $Finish[i] := false$  for  $i = 1, 2, \dots, n$ .
2. Find an i such that both
  - a.  $Finish[i] = false$
  - b.  $Need_i \leq Work$
3.  $Work := Work + Allocation_i$   
 $Finish[i] := true$   
Go to step 2.
4. If  $Finish[i] = true$  for all i, then the system is in a safe state.

### 13. Explain about deadlock detection?

In deadlock situation, the system must provide

- An algorithm that examines the state of the system to determine whether a deadlock has occurred.
- An algorithm to recover from the deadlock.

#### Single instance of each resource type

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph.

An edge from  $P_i$  to  $P_j$  in a wait-for graph implies that process  $P_i$  is waiting for process  $P_j$  to release a resource that  $P_i$  needs. An edge  $P_i \rightarrow P_j$  exists in a wait-for graph if and only if the corresponding resource-allocation graph contains two edges  $P_i \rightarrow R_q$  and  $R_q \rightarrow P_j$  for some resource  $R_q$ .

A deadlock exists in the system if and only if the wait-for graph contains a cycle.

#### Several instance of a resource

The algorithm for this employs several time-varying data structures. They are

- ❖ Available: A vector of length  $m$  indicates the number of available resources of each type.
- ❖ Allocation: A matrix defines the number of resources of each type currently allocated to each process.
- ❖ Request: An  $n \times m$  matrix indicates the current request of each process.

The algorithm is as follows

1. Let  $Work$  and  $Finish$  be vectors of length  $m$  and  $n$ , respectively. Initialize  $Work := Available$ . For  $i = 1, 2, \dots, n$ , if  $Allocation_i \neq 0$ , then  $Finish[i] := false$ . Otherwise,  $Finish[i] := true$ .
2. Find an index  $i$  such that both
  - a.  $Finish[i] = false$
  - b.  $Request_i \leq Work$

If no such  $i$  exists, go to step 4.

3.  $Work := Work + Allocation_i$   
 $Finish[i] := true$   
Go to step 2.

4. If  $Finish[i] = false$  for some  $i$ ,  $1 \leq i \leq n$ , then the system is in a deadlock state. Moreover if  $Finish[i] = false$ , then process  $P_i$  is deadlocked.

#### 14. Explain about paging?

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

Recent designs have implemented paging by closely integrating the hardware and other operating system.

#### Basic method

Physical memory is broken into fixed-sized blocks called **frames**. Logical memory is also broken into blocks of the same size called **pages**. When a process is to be executed, its pages are loaded into any available memory frames.

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

The page size is defined by the hardware. The size of page is typically a power of 2, varying between 512 bytes and 16 MB per page. The logical address is as follows

Page number	Page offset
P	D
m-n	n

m-n denotes the high order bits of a logical address. n low order bits.

#### Example

If pages are 2,048 bytes, a process of 72,766 bytes would need 35 pages plus 1,086 bytes. Some CPUs and kernels even support multiple page sizes. For instance, Solaris uses 8 KB and 4MB page sizes, depending on the data stored by the pages.

The operating system is aware of the allocation details of physical memory: which frames are allocated, which frames are available, how many total frames there are and so on. This information is generally kept in a data structure called a **frame table**.

If a user makes a system call and provides to produce the correct physical address. The operating system maintains a copy of the page table for each processes, just as it maintains a copy of instruction counter and register contents. This copy is used to translate logical addresses to physical address. Paging therefore increases the context-switch time.

## Hardware support

The hardware implementation of the page table can be done in several ways. The simplest case, the page table is implemented as a set of dedicated **registers**. These registers should be built with very high-speed logic to make the paging-address translation efficient.

The usage of registers for the page table is satisfactory if the page table is reasonably small. Most computers must allow page table to be very large. For these machines the page table is kept in main memory, and a **page table base register (PTBR)** points to the page table. The problem with this approach is the time required to access a user memory location.

A solution to this problem is to use a special, small, fast-lookup hardware cache, called **translation look-aside buffer (TLB)**. The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key and a value. The TLB contains 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.

If the page number is not in the TLB, a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory.

The percentage of times that a particular page number is found in the TLB is called **hit ratio**. To find **effective memory-access time**, we must weight each case by its probability.

## Protection

Memory protection is accomplished by protection bits that are associated with each frame. Normally, these bits are kept in the page table. One bit can define a page to be read-or write or read-only.

One more bit is generally attached to each entry in the page table: a **valid-invalid bit**. When the bit is set to “valid”, this value indicates that the associated page is in the process logical-address space, and is thus a legal page. If the bit is set to “invalid”, this value indicates that the page is not the process logical-address space.

Some systems provide hardware in the form of a **page-table length register (PTLR)** to indicate the size of the page table.

### 15. Briefly discuss on the structure of page table?

#### 1. Hierarchical paging

One way is to use a two-level paging algorithm, in which the page table itself is also paged. A logical address is divided into a page number and a page offset. The page number is further divided into a 10-bit page number and a 10-bit page offset. Thus a logical address is

Page number		Page offset
P2	P2	D
10	10	12

The address translation method for this architecture works from the outer page table inwards, this scheme is also known as **forward-mapped page table**. The Pentium II uses this architecture.

## 2. Hashed page tables

A common approach for handling address spaces larger than 32 bits is to use a **hashed page table**. Each entry in the hash table contains a linked list of elements that hash to the same location. Each element consist of three fields

- a) The virtual page number
- b) The value of the mapped page frame
- c) A pointer to the next element in the linked list.

The algorithm works as follows: The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared to field (a) in the first element in the linked list. If there is a match, the corresponding page frame is used to form the desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.

A variation to this scheme that is favorable for 64-bit address spaces has been proposed. Clustered page tables are similar to hashed page tables except that each entry in the hash table refers to several pages rather than single page.

## 3. Inverted page table

In page table the page table has one entry for each page that the process is using. The operating system must translate this reference into a physical memory address. One of the drawbacks of this method is that each page table may consist of millions of entries.

To solve this problem, we can use an **inverted page table**. An inverted page table has one entry for each real page of memory. Each entry consists of the virtual address of the page stored in that real memory location; with information about the process that owns that page.

Each virtual address in the system consists of a triple

< process-id, page-number, offset >

Each inverted page-table entry is a pair <process-id, page-number> where the process-id assumes the role of the address-space identifier. Although this scheme decreases the amount of

memory needed to store each page table, it increases the amount of time needed to search the table when a page reference occurs.

## Unit IV

### 16. Explain about demand paging?

A demand paging system is similar to a paging system with swapping. Processes reside on secondary memory. When we want to execute a process, we swap it into memory. Rather than swapping the entire processes into memory, we use a **laze swapper**. A lazy swapper never swaps a page into memory unless that page will be needed.

#### Basic concepts

When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those necessary pages into memory.

The valid-invalid bit scheme can be used to distinguish between pages that are in memory and those pages that are on the disk. Access to a page marked invalid causes a **page-default trap**. This trap is the result of the operating system's failure to bring the desired page into memory.

The procedure for handling the page fault is as follows

1. We check the internal table to determine whether the reference was valid or invalid.
2. If the reference was invalid, we terminate the process.
3. We find a free frame.
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, modify the internal table to indicate that the page is now in memory.
6. We restart the instruction.

The hardware to support demand paging is as follows

- Page table: This table has the ability to mark an entry invalid though a valid-invalid bit or special value of protection bit.
- Secondary memory: This memory holds those pages that are not present in main memory. It is a high-speed disk. It is known as swap device, and the section of disk used for this purpose is known as **swap space**.

The major difficulty occurs when one instruction may modify several different locations. This problem can be solved in two different ways.

In one solution, the microcode computes and attempts to access both ends of both blocks. If a page fault going to occur, it will happen at this step before anything is modified.

The other solution uses temporary registers to hold the values of overwritten locations. If there is a page fault, all the old values are written back into memory before the trap occurs.

A similar architectural problem occurs in machine that uses special addressing modes, including auto decrement and auto increment. These addressing modes use a register as a pointer and automatically decrement or increment the register as indicated.

One solution is to create a new special status register to record the register number and amount modified for any register that is changed during the execution of an instruction.

### **Performance of demand paging**

Demand paging can have a significant effect on the performance of a computer system. As long as we have no page faults, the effective access time is equal to the memory access time. The memory access time is denoted by **ma**, now ranges from 10 to 200 nanoseconds.

Let  $p$  be the probability of page fault. We could expect  $p$  to be close to zero; that is there will be only a few page faults. The **effective access time** is then

$$\text{Effective access time} = (1 - p) * \text{ma} + p * \text{page fault time.}$$

A page fault causes the following sequence to occur.

1. Trap (shut in) to the operating system.
2. Save the process registers and process state.
3. Determine that the interrupt was a page fault.
4. Check that the page reference was legal and determine the location of the page on the disk.
5. Issue a read from the disk to a free frame
6. While waiting, allocate the CPU to some other user.
7. Interrupt fro the disk.
8. Save the registers and process state for the other user.
9. Determine that the interrupt was from disk.
10. Correct the page table and other tables to show that the desired page is now in memory.
11. Wait for the CPU to be allocated to this process again.
12. Restore the registers, process state, and new page table, then resume the interrupted instruction.

We are faced with three major components of the page-fault service time.

1. Service the page-fault interrupts.
2. Read in the page.
3. Restart the processes.

If we take an average page-fault service time of 25 milliseconds and a memory-access time of 100 nanoseconds, then the effective access time in nanoseconds is

$$\begin{aligned} \text{Effective access time} &= (1-p) \times (100) + p (25 \text{ milliseconds}) \\ &= (1-p) \times 100 + p \times 25,000,000 \\ &= \quad 100 \quad + \quad 24,999,900 \quad \times \quad p. \end{aligned}$$

### 17. Briefly discuss on page replacement?

Page replacement approach is stated as follows. If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space, and changing the page table to indicate that the page is no longer in memory. We now use the freed frame. We modify the page-fault service routine

1. Find the location of the desired page on the disk.
2. Find a free frame:
  - a. If there is a free frame, use it
  - b. If there is no free frame, uses a page-replacement algorithm to select a victim frame.
  - c. Write the victim page to the disk; change the page and frame tables accordingly.
3. Read the desired page into the free frame; change the page and frame tables.
4. Restart the user process.

If no frames are free, two page transfers are required. This situation effectively doubles the page-fault service time and increase the effective access time. We can reduce this overhead by using a modify bit ( or dirty bit). The modify bit for a page is set by the hardware whenever any word or byte in the page is written into indicating that the page has been modified.

If the bit is set, we know that the page has been modified since it was read in from the disk. In this case we must write that page to the disk. If the modify bit is not set, however, the page has not been modified since it was read into memory.

Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, an enormous virtual memory can be provided for programmers on a smaller physical memory. With demand paging the size of the logical address space is no longer constrained by physical memory.

We must solve two major problems to implement demand paging: we must develop a **frame-allocation algorithm and a page-replacement algorithm.**

We evaluate an algorithm by running it on a particular string of memory reference and computing the number of page faults. The string of memory references is called a **reference string.**

### FIFO page replacement

A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. Insert page at the last of the queue.

The FIFO page-replacement algorithm is easy to understand and program. However its performance is not always good. Even if we select for replacement a page that is in active use, everything still works correctly. After we page out an active page to bring in a new one, a fault occurs almost immediately to retrieve the active page. Some other pages will be needed to be replaced to bring the active page back into memory. Thus, a bad replacement choice increases the page-fault rate and slows process execution.

When plotting on a graph the page faults versus the number of available frames. We notice that the number of faults for four frames is greater than the number of faults for three frames. This most unexpected result is known as **Belady's anomaly**. That is for some page-replacement algorithm, the page fault rate may increase as the number of allocated frames increases.

## **18. Discuss on LRU approximation page replacement?**

This algorithm uses reference bit. The reference bit is set by the hardware, whenever that page is referenced.

### 1. Additional-reference-bits algorithm

We can keep an 8-bit byte for each page in a table in memory. The operating system shifts the reference bits right 1 bit , discarding the low-order bit. These 8 bits sift register contain the history of page used for last eight time periods. A page with a history value of 11000100 has been used more recently than has one with 01110111. The page with the lowest number is the LRU page, and it can be replaced.

### 2. Second-Chance algorithm

when a page has been selected, 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. Thus a page that given the second chance will not be replaced until all other pages are replaced.

One way to implement the second-chance algorithm is as a circular queue. A pointer indicates which page is replaced next. When a frame is needed, the pointer advances until it finds a replaced next. Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position.

#### 4. Enhanced second-chance algorithm.

When we enhance the second-chance algorithm by considering both the reference bit and the modify bit as an order pair. We have the following four possible classes.

- a) (0,0) neither recently used nor modified – best page to replace.
- b) (0,1) not recently used but modified – not quite as good.
- c) (1,0) recently used but clean – it probably will be used again soon.
- d) (1,1) recently used and modified – it probably will be used again soon, and the page will be need to be written out to disk before it can be replaced.

This algorithm is used in the Macintosh virtual-memory-management scheme.

#### 19. Explain the various file types and structures?

A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts – a name and an extension, usually separated by a period character. The file with a .com, .exe, or .bat extension can be executed. Assembler expects source files to have an .asm 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, txt, 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

The UNIX system uses a crude **magic number** stored at the beginning of some files to indicate roughly the type of the file – executable program , batch file (or shell script) , postscript file, and so on.

### **File structure**

Certain files must conform to a required structure that is understood by the operating system. The operating system may require that an executable file have a specific structure so that it can determine where in memory to load the file and what location of the first instruction is.

Some operating system imposes a minimal number of file structures. This approach has been adopted in UNIX, MS-DOS and others. UNIX considers each file to be a sequence of 8-bit bytes; no interpretation of these bits is made by the operating system.

The Macintosh operating system supports minimal number of file structures. It expects files to contain two parts: a **resource fork** and a **data fork**. The resource fork contains information of interest to the user. Example it holds the labels of any buttons displayed by the program. A foreign user may want to modify these buttons. The data fork contains program code or data.

### **Internal file structure**

All disk I/O is performed in units of one block, and all blocks are the same size. It is unlikely that the physical record size will exactly match the length of the desired logical record. Logical record may even vary in length. Packing a number of logical records into physical blocks is a common solution to this problem.

The logical record size, physical block size, and packing technique determine how many logical records are in each physical block. Because disk space is always allocated in blocks, some portion of the last block of each file is generally wasted. The wasted byte allocated to keep everything in units of blocks is **internal fragmentation**. All file system suffer from internal fragmentation; the larger the block size, the greater the internal fragmentation.

## **20. Explain about directory structure?**

Some systems store millions of files on terabytes of disk. To manage all these data, we need to organize them. This organization is usually done in two parts. First, disks are split into one or more partitions, also known as minidisks in the IBM world or volumes in the PC and Macintosh. Partitions can be thought of as virtual disks. Partitions can also store multiple operating systems, allowing a system to boot and run more than one.

Second, each partition contains information about files within it. This information is kept in entries in a **device directory** or **volume table of contents**. The device directory records information such as name, location, size, and type- for all files on that partition.

The directory can be viewed as a symbol table that translates file names into their directory entries. Operations that are to be performed on a directory are

- Search for a file: We are able to find all files whose names match a particular pattern.
- Create a file: New files need to be created and added to the directory.
- Delete a file: We want to remove it from the directory.
- List a directory: List the files along with the content of the directory entry.
- Rename a file: The name of the file is changed.
- Traverse the file system: We may wish to access every directory, and every file within a directory structure.

### **Single-level Directory**

All files are constrained in the same directory, which is easy to support and understood.

One limitation is when the number of files increases or when the system has more than one user. We must use a unique name.

### **Two-level directory**

In the two-level directory structure each user has own **user file directory** (UFD). When a user jobs starts or a user logs in, the system's **master file directory** (MFD) is searched. The MFD is indexed by user name or account number, and each entry points to the UFD for the user.

Although the two-level directory structure solves the name-collision problem, still have disadvantages. This structure isolates one user form another. A two-level directory can be thought of as a tree, or at least an inverted tree, of height 2.

A special case of file name situation occurs in regard to the system files such as loaders, assemblers, compilers, utility routines, libraries and so on. Copying all the system files would waste an enormous of space.

The standard solution is to complicate the search procedure slightly. A special user directory is defined to contain the system files. Whenever a file name is given to be loaded, the operating system first searches the local UFD. IF the file is found it is used. If it is not found, the system automatically searches the special user directory that contains the file system files. The sequence of directories searched when a file is named is called the **search path**.

### **Tree-structured directories**

We can generalize the directory structure to a tree of arbitrary height. This allows the user to create their own sub directories and to organize their files accordingly. The tree has a

root directory. Every file in the system has a unique path name. A path name is the path from the root, through all the subdirectories, to a specified file.

Each user has a current directory. The current directory should contain most of the files that are of current interest to the user. To change directories, a system call is provided that takes a directory name as a parameter and uses it to redefine the current directory.

Path name can be of two types: absolute path names or relative path names. An **absolute path name** begins at the root and allows a path down to the specified file, giving the directory name on the path. A **relative path name** defines a path from the current directory.

If a directory is empty, its entry in its containing directory can simply be deleted. Thus to delete a directory, the user must first delete all the files in that directory.

Another approach taken by the UNIX rm command is, when a request is made to delete a directory, that entire directory's files and subdirectories are also to be deleted. With a tree structure directory system, users can access, in addition to their files, the files of other users.

A path to a file in a tree-structured directory can be longer than that in a two-level directory. To allow users to access programs without having to remember these long paths, the Macintosh operating system maintains a file called the Desktop file, containing the name and location of all executable programs.

### **Acyclic-graph directories**

An **acyclic graph** allows directories to have shared subdirectories and files. The same file or subdirectory may be in two different directories. A shared file is not the same as two copies of the file. With the shared file, only one actual file exists, so any changes made by one person are immediately visible to other.

Shared files and subdirectories can be implemented in several ways. In UNIX shared files are implemented by create a new directory entry called a link. A **link** is effectively a pointer to another file or subdirectory.

Another common approach to implementing shared files is simply to duplicate all information about them in both sharing directories.

Several problems are a file may now have multiple absolute path names. Distinct file names may refer to the same file. To traverse the entire file – this problem becomes significant. Another problem involves deletion. When can the space of shared file is reallocated and reused.

In some situation where link is used, the deletion of a link doses not need to affect the original file. Another approach to deletion is to preserve the file until all references to it are

deleted. When a link or directory entry is deleted, we remove its entry on the list. The file is deleted when its file-reference list is empty.

The UNIX operating system uses this approach for nonsymbolic links (or hard links), keeping a reference count in the file information block.

### **General graph directory**

One serious problem with using an acyclic-graph structure is ensuring that there are no cycles. When we add links to an existing tree-structured directory, the tree structure is destroyed, resulting in a simple graph structure.

If cycles are allowed to exist in the directory, we likewise want to avoid searching any component twice, for reasons of correctness as well as performance. When cycle exists, the reference count may be nonzero, even when it is no longer possible to refer to a directory or file. This anomaly results from the possibility of self-referencing in directory structure. In this case we generally need to use a garbage collection. Garbage collection involves traversing the entire file system marking everything that can be accessed.

## **Unit V**

### **21. Explain about file system structure?**

File system structure have two characteristics they are

1. They can be rewritten in place.
2. They can access directly any given block of information on the disk.

To provide an efficient and convenient access to the disk, the operating system imposes one or more file systems to allow the data to be stored, located and retrieved easily. A file system poses two quite different design problems. The first problem is defining how the file system should look to the user. The second problem is creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.

The file system is composed of many different levels. The structure is

Application programs

Logical file system

File-organization module

Basic file system

## I/O control

### Devices

The lowest level, the I/O control consists of **device drivers** and interrupts handlers to transfer information between the main memory and the disk systems. A device driver can be thought of as a translator. Its input consists of high-level commands such as “retrieve block 123”.

The **basic file system** needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk.

The **file-organization module** knows about files and their logical blocks, as well as physical blocks. It also includes a free-space manager, which tracks unallocated blocks and provides these blocks to the file-organization module when requested.

The **logical file system** manages metadata information. Metadata includes all of the file-system structure, excluding actual data. It maintains file structure via file control blocks. A **file control block** (FCB) contains information about the file, including ownership, permission, and location of the file contents. The logical file system is also responsible for protection and security.

Most operating systems support more than one file system. Windows NT supports disk file-system formats of FAT, FAT32 and NTFS as well as CD-ROM, DVD and floppy-disk file system formats.

## 22. Briefly explain about file system implementation?

Several on-disk and in-memory structures are used to implement a file system. On-disk, the file system may contain information about how to boot an operating system stored there, the total number of blocks, the number and the location of free blocks, the directory structure, and individual files.

The on-disk structures include

- A **boot control block** can contain the information needed by the system to boot an operating system from that partition. If the disk does not contain an operating system, this block can be empty. It is typically the first block of a partition. In UFS this is called **boots block**. In NTFS it is the **partition boot sector**.
- A partition control block contains partitions details, such as the number of blocks in the partition, size of the blocks, free-blocks and free-block pointers and free FCB count and FCB pointers. In UFS this is called a **super block**. In NTFS it is the Master File Table.
- A directory structure is used to organize the files.
- An FCB contains many of the file’s details, including file permissions, ownership, size, and location of the data blocks. In UFS this is called the inode.

The in-memory information is used for both file-system management and performance improvement via caching. The structure can include

- An in-memory partition table containing information about each mounted partition.
- An in-memory directory structure that holds the directory information of recently accessed directories.
- The **system-wide open-file table** contains a copy of the FCB of each open file, as well as other information.
- The **per-process open-file table** contains a pointer to the appropriate entry in the system-wide open-file table as well as other information.

To create a new file, an application program calls the logical file system. The logical file system knows the format of the directory structures. To create a new file, it allocates a new FCB, reads the appropriate directory into memory, updates it with the new file name and FCB, and writes it back to the disk. A typical file control block is as shown below

File permissions
File dates ( create, access, write)
File owner, group, ACL
File size
File data blocks

The open call passes a file name to the file system. Parts of the directory structure are cached in memory to speed directory operations. Next an entry is made in the per-process open-file table, with a pointer in the system-wide open-file table and some other fields. The open call returns a pointer to the appropriate entry in the per-process file system table. All the file operations are performed via this pointer. The name given to the index varies. UNIX system refers to it as a **file descriptor**. Windows 2000 refers to it as a **file handle**.

When a process closes the file, the per-processes table entry is removed, and the system-wide entry's open count is decremented.

Using the caching aspects, all information about the open file, except for its actual data blocks is in memory.

### Partitions and mounting

A disk can be sliced into multiple partitions, or a partition can span multiple disks. Each partition can be either "raw", containing no file system, or "cooked" containing a file system. **Raw disk** is used where no file system is appropriate. Raw disk can also hold information needed by disk RAID systems, such as bit maps indicating which blocks are mirrored and which have changed and needed to be mirrored.

Boot information can be stored in a separate partition. It has its own format. Execution of the image starts at a predefined location, such as the first byte. This boot image contain more than the instructions for how to boot a specific operating system. PCs and other systems can be **dual-booted**. Multiple operating systems can be installed on such system.

The **root partition**, which contains the operating-system kernel and potentially other system files, is mounted at boot time. In successful mount operation, operating system verifies that the device contains a valid file system. Finally the operating system notes in its in-memory **mount table** structure that a file system is mounted, and the type of the file system.

On UNIX file systems can be mounted at any directory. This is implemented by setting a flag in the in-memory copy of the inode for that directory.

### **Virtual file systems**

The file-system implementation consist of three major layers. The first layer is the file-system interface, based on the open, read, write and close calls and file descriptors.

The second layer is called the **Virtual File System (VFS)**. It serves two important functions.

1. It separates file-system-generic operations from their implementation by defining a clean VFS interface.
2. The VFS is based on a file-representation structure called a vnode, which contains a numerical designator for a network-wide unique file. This network-wide uniqueness is required for support of network file systems.

### **23. Explain about file allocation methods?**

The main problem in direct-access nature is how to allocate space to these files so that disk space if utilized effectively and files can be accessed quickly. Three major methods are used; they are contiguous, linked and indexed.

#### **Contiguous allocation**

The contiguous-allocation method requires each file to occupy a set of contiguous blocks of the disk. Disk addresses define a linear ordering on the disk. Thus the number of disks seeds required for accessing contiguously allocated files is minimal. The IBM VM/CMS operating system uses contiguous allocation because it provides such good performance.

Contiguous allocation of a file is defined by the disk address and length of the first block. The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file.

Contiguous allocation has some problems. One difficulty is finding space for a new file. The contiguous disk-space-allocation problem can be seen to be a particular application of the general **dynamic storage-allocation** problem. First fit and best fit are the most common strategies used to select a free hole from the set of available holes.

These algorithms suffer from the problem of **external fragmentation**.

### **Linked allocation**

With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file.

### **Indexed allocation**

Indexed allocation bringing all the pointers together into one location : the index block. Each file has its own index block, which is an array of disk-block addresses. The *i*th entry in the index block points to the *i*th block of the file. The directory contains the address of the index block.

## **24. Briefly discuss on I/O hardware?**

A controller is a collection of electronics that can operate a port, a bus, or a device. A serial-port controller is a simple controller. It is a single chip in the computer that controls the signals the signal on the wires of a serial port.

The device control registers are mapped into the address space of the processor. The CPU executes i/o requests using the standard data transfer instructions to read and write the device-control registers.

An I/O port consist of

- a. Status register
- b. Control register
- c. Data-in register
- d. Data-out register

### **Polling**

### **Interrupts**

The hardware mechanism that enables a device to notify the CPU is called an interrupt. 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 instruction pointer, and jumps to the interrupt-handler routine at a fixed address in memory.

### **Direct memory access**

Many computers avoid burdening the main CPU with programmed I/O by offloading some of this work to a special-purpose processor called a direct memory address controller.

### **25. Explain about disk scheduling?**

- FCFS Scheduling
- SSTF scheduling
- SCAN scheduling
- C-SCAN scheduling
- LOOK Scheduling