Chapter One: Introduction

Overview of Operating Systems (OS)

What is Operating System?

Operating system is the core software component of your computer. It performs many functions and is, in very basic terms, an interface between your computer and the outside world. A computer is a modern system consists of one or more processor, some main memory, disks, printers, a keyboard, display and other I/O systems. This is a complex system. Thus writing programs that keeps track of all these components and use them correctly is a mandatory. For these reason, computers are provided with a layer of software called the O***perating Systems (OS***).The job of OS is to manage all these devices and provide user program with a simpler interface to the hardware.

An operating system is the actual software that controls the allocation and use of a computer's hardware. It keeps components working in unity, acting as a communicator between the user, the computer's hardware and software.

Operating system is system software that controls the execution of programs and that provides services such as resource allocation, scheduling, I/O control and data/file management. It hides the complexity of how hardware components work.

Operating System interprets commands and instructions. It also coordinates compilers, assemblers, utility programs, and other software to the various user of the computer system. OS provides easy communication between the computer system and the computer operator (human). It also establishes data security and integrity.

Basic Functions of operating system

Operating systems perform basic tasks, such as recognizing input from the keyboard, sending output to the display screen, keeping track of files and directories on the disk, and controlling peripheral devices such as disk drives and printers.

For large systems, the operating system has even greater responsibilities and powers. It makes sure that different program and users running at the same time do not interfere with each other. The operating system is also responsible for security, ensuring that unauthorized users do not access the system.

Today most operating systems perform the following important functions:

1. **Process Management**:

That is, assignment of processor to different tasks/processes being performed by the computer system. This allows two or more processes to be executed at a time. Here the operating system must decide if it can run the different processes on single processor at a time, using multitasking concept.

The operating system needs to allocate enough of the processor's time to each process and application so that they can run as efficiently as possible. This is particularly important for multitasking. When the user has multiple applications and processes running, it is up to the operating system to ensure that they have enough resources to run properly.

2**. Memory management**:

That is, allocation of main memory and other storage areas to the system programs as well as user programs and data. This involves allocating, and often to create a virtual memory for program. Paging which means organizing data so that the program data is loaded into pages of memory. Another method of managing memory is swapping. This involves swapping the content of memory to disk storage.

The operating system needs to ensure that each process has enough memory to execute the process, while also ensuring that one process does not use the memory allocated to another process. This must also be done in the most efficient manner. A computer has four general types of memory. In order of speed, they are: high-speed cache, main memory, secondary memory, and disk storage. The operating system must balance the needs of each process with the different types of memory available.

3. **Input/output management or Device Management**:

That is, co-ordination and assignment of the different output and input device while one or more programs are being executed. Most computers have additional hardware, such as printers and scanners, connected to them. These devices require drivers, or special programs that translate the electrical signals sent from the operating system or application program to the hardware device. The operating system manages the input to and output from the computer.

4. **File Management**:

That is, the storage of file on various storage devices. It also allows all files to be easily changed and modified through the use of text editors or some other files manipulation. An Operating System can create and maintain a file System, where users can create, delete and move files around a structured file system.

At the simplest level, an operating system does two things:

1. It manages the hardware and software resources of the system.

2. It provides a stable, consistent way for applications, for users, to deal with the hardware without having to know all the details of the hardware.

Types of Operating Systems

There are several types of operating systems, with Windows, Linux and Macintosh group being the most widely used. Here is an overview on each system:

Windows: Windows is the popular Microsoft brand preferred by most personal users. Although Windows has made steps in regard to security, it has a reputation for being one of the most vulnerable systems.

Unix/Linux: The UNIX operating system is well known for its stability. UNIX is often used more as a server than a workstation. Linux was based on the UNIX system, with the source code being a part of GNU open-source project. Both systems are very secure yet far more complex than Windows.

Macintosh: Recent versions of the Macintosh operating system, including the Mac OS X, follow the secure architecture of UNIX. Systems developed by Apple are efficient and easy to use, but can only function on Apple branded hardware.

Within the broad family of operating systems, there are generally four types, categorized based on the types of computers they control and the sort of applications they support. The categories are:

- **Real-time operating system** (RTOS) - Real-time operating systems are used to control machinery, scientific instruments and industrial systems. An RTOS typically has very little user-interface capability, and no end-user utilities, since the system will be a "sealed box" when delivered for use. A very important part of an RTOS is managing the resources of the computer so that a particular operation executes in precisely the same amount of time, every time it occurs.
- **Single-user, single task** - As the name implies, this operating system is designed to manage the computer so that one user can effectively do one thing at a time. DOS is single-task operating system.
- **Single-user, multi-tasking** - This is the type of operating system most people use on their desktop and laptop computers today. Microsoft's Windows and Apple's MacOS X platforms are both examples of operating systems that will let a single user have several programs in operation at the same time. For example, it's entirely possible for a Windows user to be writing a note in a word processor while downloading a file from the Internet while printing the text of an e-mail message.
- **Multi-user** - A multi-user operating system allows many different users to take advantage of the computer's resources simultaneously. The operating system must make sure that the requirements of the various users are balanced, and that each of the programs they are using has sufficient and separate resources so that a problem with one

user doesn't affect the entire community of users. UNIX and Mainframe operating systems, such as *MVS*, are examples of multi-user operating systems.

When OS start its work?

Although operating systems differ, many of their basic functions are similar. Most users today have a computer with a hard disk drive. When the computer is turned on, the operating system will be loaded from the hard drive into the computer's memory, thus making it available for use. The process of loading the operating system into memory is called bootstrapping, or booting the system. The word booting is used because, figuratively speaking, the operating system pulls itself up by its own bootstraps.

When you turn on the power to a computer, the first program that runs is usually a set of instructions kept in the computer's read-only memory (ROM). This code examines the system hardware to make sure everything is functioning properly. This power-on self test (POST) checks the CPU, memory, and basic input-output systems (BIOS) for errors and stores the result in a special memory location. Once the POST has successfully completed, the software loaded in ROM (sometimes called the BIOS or firmware) will begin to activate the computer's disk drives. In most modern computers, when the computer activates the hard disk drive, it finds the first piece of the operating system: the bootstrap loader.

The bootstrap loader is a small program that has a single function: It loads the operating system into memory and allows it to begin operation. In the most basic form, the bootstrap loader sets up the small driver programs that interface with and control the various hardware subsystems of the computer. It sets up the divisions of memory that hold the operating system, user information and applications. Then it turns control of the computer over to the operating system.

Chapter Two: Process Management

Process Description

Modern computers work in a multitasking environment in which they execute multiple programs simultaneously. These programs cooperate with each other and share the same resource, such as memory and cpu. An operating system manages all processes to facilitate proper utilization of these resources.

Important concept: decomposition. Given hard problem, chop it up into several simpler problems that can be solved separately. A big size program should be broken down into **processes** to be executed.

What is a process?

- A running piece of code or a program in execution.
- "An execution program in the context of a particular process state."
- A process includes code and particular data values
- Process state is everything that can affect, or be affected by,
- Only one thing (one state) happens at a time within a process.
- The term process, used somewhat interchangeably with 'task' or 'job'.

Process States:

A process goes through a series of discrete process states.

- **New State:** The process being created.
- **Ready State:** The new process that is waiting to be assigned to the processor.
- **Running State:** A process is said to be running if it actually using the CPU at that particular instant.

- **Blocked (or waiting) State:** A process is said to be blocked if it is waiting for some event to happen such that as an I/O request before it can proceed.
- **Terminated state:** The process has finished execution.

How processes are created?

Creating a process from scratch (e.g., the Windows/NT uses **CreateProcess()**):

- Load code and data into memory.
- Create (empty) call stack.
- Create and initialize process control block.
- Make process known to dispatcher/ scheduler.

Forking: want to make a copy of existing process (e.g., UNIX uses **fork ( )** function).

- Make sure process to be copied is not running and has all state saved.
- Make a copy of code, data, and stack.
- Copy PCB of source into new process.
- Make process known to dispatcher/scheduler.

A process in operating system is represented by a data structure known as **a Process Control Block (PCB)** or process descriptor. Process information has to keep track in PCB. For each process, PCB holds information related to that process.

The PCB contains important information about the specific process including:

- The current state of the process i.e., whether it is ready, running, waiting, or whatever.
- Unique identification of the process in order to track "which is which" information.
- A pointer to parent process.
- Similarly, a pointer to child process (if it exists).
- The priority of process (a part of CPU scheduling information).
- Pointers to locate memory of processes.
- A register save area.

- The process it is running on.

The PCB is a certain **store** that allows the operating systems to locate key information about a process. Thus, the PCB is the data structure that defines a process to the operating systems.

Process table: collection of all process control blocks for all processes.

How can several processes share one CPU? OS must make sure that processes do not interfere with each other. This means

- Making sure each gets a chance to run inside the cpu (fair scheduling).
- Making sure they do not modify each other's state (protection).

Context Switching

Most modern computer systems allow more than one process to be executed simultaneously. This is called ***Multitasking*** **systems**.

Context switching means switching the cpu to different processes. It requires saving of the state of the current process into the PCB and load the saved PCB of the previous or new process.

How does cpu switch between different processes?

Dispatcher (also called Short Term **Scheduler**): inner-most portion of the OS that runs processes without interference: Scheduler supports and facilitates the following activities

- Run process for a while
- Save its state
- Load state of another process
- Run the new process and after some time it reload the suspended /previous process.

OS needs some agent to facilitate the state saving and restoring code.

- All machines provide some special hardware support for saving and restoring state:

Process Scheduling:

When more than one process is running, the operating system must decide which one should first run. The part of the operating system concerned with this decision is called the ***scheduler***, and algorithm it uses is called the ***scheduling algorithm***.

In multitasking and uniprocessor system scheduling is needed because more than one process is in ready and waiting state. A certain scheduling algorithm is used to get all the processes to run correctly. The processes should be in such a manner that no processes must be made to wait for a long time.

Goals of scheduling (objectives):

What the scheduler try to achieve?

General Goals

Fairness:

Fairness is important under all circumstances. A scheduler makes sure that each **process gets its fair share of the CPU** and no process can suffer indefinite postponement/delay. Not giving equivalent or equal time is not fair.

Efficiency:

Scheduler should **keep the system (or in particular CPU) busy 100%** of the time when possible. If the CPU and all the Input/Output devices can be kept running all the time, more work gets done per second than if some components are idle.

Response Time:

A scheduler should minimize the response time for interactive user.

Turnaround:

A scheduler should **minimize** the time batch users must wait for an output of executing process. It is **Cpu time + Wait time**

Throughput:

A scheduler should **maximize** the number of jobs processed per unit time.

Preemptive Vs Non preemptive Scheduling:

The Scheduling algorithms can be divided into two categories with respect to how they deal with interrupts.

Non preemptive Scheduling:

A scheduling discipline is non preemptive if once a process has been given the CPU, the CPU cannot be taken away from that process until the assigned process completes its execution.

Following are some characteristics of non preemptive scheduling

1. In non preemptive system, short jobs are made to wait by longer jobs..
2. In non preemptive system, response times are more predictable, maximum, because incoming high priority jobs cannot displace waiting jobs.
3. In non preemptive scheduling, a scheduler executes jobs in the following two situations.
    a. When a process switches from running state to the waiting state.
    b. When a process terminates.

Preemptive Scheduling:

The strategy of allowing processes that are logically running to be temporarily suspended is called Preemptive Scheduling and it is contrast to the "run to completion" method.

Scheduling Algorithms

CPU Scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU.

The following are some scheduling algorithms:

- First Come First Served (FCFS) Scheduling.
- Shortest Job First (SJF) Scheduling.
- Shortest Remaining Time (SRT) Scheduling.
- Round Robin Scheduling.
- Priority Scheduling.

First-Come-First-Served (FCFS) Scheduling

Other names of this algorithm are:

- First-In-First-Out (FIFO)
- Run-to-Completion
- Run-Until-Done

First-Come-First-Served algorithm is the simplest scheduling algorithm. Processes are dispatched according to their arrival time on the ready queue. Being a non preemptive discipline, once a process gets a CPU, it runs to completion. The FCFS scheduling is fair in the formal sense or human sense of fairness but it is unfair in the sense that long jobs make short jobs wait a long period.

FCFS scheme is not useful in scheduling interactive users because it cannot guarantee good response time. The code for FCFS scheduling is simple to write and understand. One of the major drawbacks of this scheme is that the average time is often quite long.

Example:

Shortest-Job-First (SJF) Scheduling

Other name of this algorithm is Shortest-Process-Next (SPN).

Shortest-Job-First (SJF) is a non-preemptive discipline in which waiting job (or process) with the smallest estimated run-time-to-completion is run first. In other words, when CPU is available, it is assigned to the process that has smallest next CPU burst.

The SJF algorithm favors for short jobs (or processes) at the expense of longer ones. Since the SJF scheduling algorithm gives the minimum average time for a given set of processes, it is probably optimal.

Example:

Shortest-Remaining-Time (SRT) Scheduling

If a new process arrives with a shorter next cpu time than what is left of the currently executing process, then the new process get the cpu.

- The SRT is the preemtive counterpart of SJF and useful in time-sharing environment.
- In SRT scheduling, the process with the smallest estimated left run-time to completion is run next, including new arrivals.
- In SJF scheme, once a job begins executing, it run to completion.
- In SRT scheme, a running process may be preempted/ interrupted by a new arrival process with shortest estimated remaining/left run-time.
- The algorithm SRT has higher overhead than its counterpart SJF.
- The SRT must keep track of the elapsed time of the running process and must handle occasional preemptions.
- In this scheme, arrival of small processes will run almost immediately. However, longer jobs have even longer mean waiting time.

Example:

Round Robin Scheduling

One of the simplest, fairest and most widely used algorithms is round robin (RR).

In the round robin scheduling, processes are dispatched in a FIFO manner but are given a limited amount of CPU time called a time-slice or a quantum.

If a process does not complete before its CPU-time expires, the CPU is preempted/interrupted and given to the next process waiting in a queue. The preempted process is then placed at the back of the ready list.

Round Robin Scheduling is preemptive (at the end of time-slice) therefore it is effective in time-sharing environments in which the system needs to guarantee reasonable response times for interactive users.

The only interesting issue with round robin scheme is the length of the quantum. Setting the quantum too short causes too many context switches and lower the CPU efficiency. On the other hand, setting the quantum too long may cause poor response time and approximates FCFS. So the slice time should not be too short and too long.

Example:

Priority Scheduling

The basic idea is straightforward: each process is assigned a priority, and priority is allowed to run. Equal-Priority processes are scheduled in FCFS order or in RR. The shortest-Job-First (SJF) algorithm is a special case of general priority scheduling algorithm.

SJF algorithm is also a priority algorithm where the priority is the inverse of the (predicted) next CPU burst. That is, the longer the CPU burst, the lower the priority and vice versa

Example:

Inter process Communication (IPC)

In the case of uniprocessor systems, processes interact on the bases of the degree to which they are aware of each other's existence.

1. Processes unaware of each other: The processes are not working together; the OS needs to be concerned about competition for resources (cpu or memory).
2. Processes directly aware each other: Processes that are designed to work jointly on the same program or application. These processes exhibit cooperation but competition for shared variables (global variables).

One of the benefits of multitasking is that several processes can be made to cooperate in order to achieve their ends. To do this, they must do one of the following.

Communicate: Inter-process communication (IPC) involves sending information from one process to another, since the output of one process may be an input for the other process.

Share data: A segment of memory must be available to both processes.

Waiting: Some processes wait for other processes to give a signal before continuing. All these are issues of synchronization/communication.

Since processes frequently need to communicate with other processes therefore, there is a need for a well-structured communication. And the interaction or communication between two or more processes is termed as **Interprocess communication (IPC).**Information sharing between processes or IPC achieved through shared variable ,by passing through shared file and by using OS services provided for this purpose like OS signals(system calls like interrupts).

When processes co-operate each other's there is a problem of how to synchronize cooperating processes. For example, suppose two processes modify the same file. If both processes tried to write simultaneously the result would be a senseless mixture. We must have a way of synchronizing processes, so that even concurrent processes must stand in line to access shared data *serially* without any interference or competition or racing.

Race Conditions

In operating systems, processes that are working together share some common storage (main memory, cpu, file etc.) that each process can read and write. When 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***. Concurrently executing processes that share data need to synchronize their operations and processing in order to avoid race (competition) condition on shared data or shared resource. Only one process at a time should be allowed to examine and update the shared variable.

Critical Section

The key to preventing trouble involving shared storage is find some way to prohibit more than one process from reading and writing the shared data simultaneously. That part of the program where the shared memory is accessed is called the ***Critical Section***. To avoid race conditions and flawed/unsound results, one must identify codes in *Critical Sections* in each process.

Here, the important point is that when one process is executing shared modifiable data in its critical section, ***no other process is to be allowed*** to execute in its critical section. Thus, the execution of critical sections by the processes is mutually exclusive in time.

How to avoid race conditions?

Mutual Exclusion:

A way of making sure that if one process is using a shared modifiable data, the other processes will be excluded from doing the same thing.

Formally, while one process executes the shared variable, all other processes desiring to do so at the same time should be kept waiting; when that process has finished executing the shared variable, one of the processes waiting to do so should be allowed to proceed. In this fashion, each process executing the shared data (variables) excludes all others from doing so simultaneously. ***This is called Mutual Exclusion;*** *this approach can avoid race condition.*

Note that mutual exclusion needs to be enforced only when processes access shared modifiable data - when processes are performing operations that do not conflict with one another they should be allowed to proceed concurrently.

Methods of Achieving Mutual Exclusion

In order to achieving mutual exclusion one should develop a pre-protocol (or entry protocol) and a post-protocol (or exist protocol) to keep two or more processes from being in their critical sections at the same time. When one process is updating shared modifiable data in its critical section, no other process should allow entering in its critical section.

The following are methods that used to avoid race conditions:

Disabling Interrupts:

Each process disables all interrupts just after entering in its critical section and re-enables all interrupts just before leaving critical section. With interrupts turned off the CPU could not be switched to other process. Hence, no other process will enter its critical and mutual exclusion achieved.

Example: double deposite=80,000;//deposite is global variable shared by both processes A and B

// Process A	// Process B
disable_interrupt();	disable_interrupt();
Cal_interest()	withdraw()
{	{
float interest=deposite*0.1;	float withdrawamount=20,000;
P_B → deposite=deposite+interest;	deposite=deposite-withdrawamount;
}	}
enable_interrupt();	enable_interrupt();

Lock Variable

In this solution, we consider a single, shared, (lock) variable, initially 0. When a process wants to enter in its critical section, it first tests the lock. If lock is 0, the process first sets it to 1 and then enters the critical section. If the lock is already 1, the process just waits until (lock) variable becomes 0. Thus, lock= 0 means that no process in its critical section, and lock= 1 means some process uses the critical section.

Example: boolean testset(int lock)

```
{   If(lock==0)
  {   lock=1;return true;
     else return false;
   }
```

}

Using Systems calls 'sleep' and 'wakeup'

Basically, what above mentioned solution do is: when a process wants to enter in its critical section, it checks to see if the entry is allowed? If it is not, the process goes into tight loop and waits (i.e., start busy waiting) until it is allowed to enter. This approach waste CPU-time.

Now look at some interprocess communication primitives is the pair of sleep-wakeup.

- Sleep ( )
  - It is a system call that causes the caller to block, that is, be suspended/sleep until some other process wakes it up.
- Wakeup ( )
  - It is a system call that wakes up the process.

The Bounded Buffer Producers and Consumers example: an example how sleep-wakeup system calls are used.

The bounded buffer producers and consumers assume that there is a fixed buffer size i.e., a finite numbers of slots is available.

To suspend the producers when the buffer is full, to suspend the consumers when the buffer is empty, and to make sure that only one process at a time manipulates a buffer so there are no race conditions or lost updates.

Two processes share a common, fixed-size (bounded) buffer. The producer puts information into the buffer and the consumer takes information out.

Trouble arises when

1. The producer wants to put a new data in the buffer, but buffer is already full.
   Solution: Producer goes to sleep and to be awakened when the consumer has removed data.
2. The consumer wants to remove data the buffer but buffer is already empty.
   Solution: Consumer goes to sleep until the producer puts some data in buffer and wakes consumer up.

Semaphores

A semaphore is a protected variable whose value can be accessed and altered only by the operations **down ( )** and **up ( )** and initialization operation called 'Semaphoiinitislize'.

The down (or wait or sleep) operation on semaphores S, written as down(S) or wait (S) operates as follows:

down(S): **IF** S > 0
THEN S := S - 1
ELSE (wait on S)

The up (or signal or wakeup) operation on semaphore S, written as up(S) or signal (S), operates as follows:

up(S): **IF** (one or more process are waiting on S)
THEN (let one of these processes proceed)
ELSE S := S +1

Operations down and up are done as single, indivisible, atomic action. It is guaranteed that once a semaphore operation has stared, no other process can access the semaphore until operation has completed. Mutual exclusion on the semaphore, S, is enforced within **down(S)** and **up(S)**.

If several processes attempt a down(S) simultaneously, only one process will be allowed to proceed. The other processes will be kept waiting, but the implementation of down and up guarantees that processes will not suffer indefinite postponement/delay.

Monitor

Some languages have special class-environments for dealing with mutual exclusion. Such an environment is called a monitor.

> A monitor is a language component which removes some of the pain from synchronization. Only one process can be `inside' a monitor at a time.
>
> A procedure or function defined under the umbrella of a monitor can only access those shared memory locations declared within that monitor and vice-versa.

Chapter III: Memory Management

3.1. Introduction

- Memory is one of the most important resources of the computer system that is used to store data and programs temporarily.
- Part of the operating system that manages memory is called the Memory Manager (MM).
- The main functions of the Memory Manager (MM) are the following:
    - Keeping track of which part of memory are in use and which parts are free
    - Allocating and deallocating memory to processes
    - Managing swapping between memory and disk when memory is not big enough to hold all the processes
- A good memory manager has the following attributes
    - It allows all processes to run
    - Its memory (space used for the management activity) overhead must be reasonable
    - Its time overhead (time required for the management activity) is reasonable

Memory Management Schemes

- There are a number of memory management schemes, ranging from very simple to highly sophisticated ones. They can be broadly classified into two:
    - Mono-programming – Only one process will be in the memory in addition to the operating system
    - Multiprogramming - It allows multiple programs to run in parallel running in parallel. Divides the memory among concurrently running processes and the operating system.

3.2 Memory Partitioning

- In order to run multiple programs in parallel a memory should be partitioned.
- There are two partitioning options – fixed partitioning and Dynamic partitioning
    - Fixed partitioning – partitioning is done before the processes comes to memory
    - Dynamic partitioning – partitioning is done when processes request memory space

3.2.1. Fixed partitioning

- It is the simplest and oldest partitioning technique
- It divides the memory into fixed number of equal or unequal sized partitions
- The partitioning can be done by the OS or the operator when the system is started
- Variations of fixed partitioning
    - Equal sized fixed partitions
    - Unequal sized fixed partitions

A. <u>Equal sized fixed partitions</u>

- The memory is partitioned into equal-sized partitions
- Any process whose size is less than or equal to the partition size can be loaded into any available partition

- If all partitions are occupied with processes that are not ready to run, then one of these processes is swapped-out and a new process is loaded or a suspended process is swapped in and loaded
- Problems:
    - A program may be too big to fit into a partition
    - Inefficient use of memory due to **internal fragmentation**. Internal fragmentation is the phenomenon, in which there is wasted space internal to a partition due to the fact that the process loaded is smaller than the partition.

B. <u>Unequal sized fixed partitioning</u>

- Memory is partitioned into unequal-sized fixed partitions
- In this case there are two memory allocation schemes:
    - Using multiple queues
    - Using a single queue

i. Using multiple queues

- Each partition has its own queue
- A process is assigned to the smallest partition within which it will fit and it can only be loaded into the partition it belongs
- **Advantage**: Minimize internal fragmentation
- **Disadvantage**: Some partitions may remain unused even though processes are waiting in some other queues

ii. Using single queue

- The smallest available partition that can hold the process is selected
- Whenever a partition becomes free, the process closest to the front of the queue that fits in it could be loaded into the empty partition.
- Since it can cause internal fragmentation, another technique suggests to search from the queue the one that best fits to the now empty space.
- **Advantage**: Partitions are used all the time
- **Disadvantage**: When best fit is searched, it will not give small jobs best service, as they are interactive. A possible remedy for this problem is not to allow small processes not to be skipped more than k – times.

Summary

Advantage of fixed portioning

- It is simple to implement and requires minimal management overhead

Disadvantages of fixed partitioning

- Inefficient use of memory due to internal fragmentation
- Limits number of active processes
- A program may be too big to fit in any of the partitions.

3.2.2. Dynamic partitioning

- Partitions are created dynamically during the allocation of memory
- The size and number of partitions vary throughout of the operation of the system
- Processes will be allocated exactly as much memory they require

- Dynamic partitioning leads to a situation in which there are a lot of small useless holes in memory. This phenomenon is called **external fragmentation**.
- Solution to external fragmentation:
    - Compaction – combining the multiple useless holes in to one big hole
        - The problem of memory compaction is that it requires a lot of CPU time. For instance, on a 32M machine that can copy 16B/ms, it takes 2 sec to compact all of the memory.
- A process has three major sections - *stack, data* and *code*
- With fixed partitions, the space given to process is usually larger than it asked for and hence the process can expand if it needs
- With dynamic partitions since a process is allocated exactly as much memory it requires, it creates an overhead of moving and swapping processes whenever the process grows dynamically.
- It is expected that most processes will grow as they run, so it is a good idea to allocate a little extra memory whenever a process is loaded into the memory.
- If a process runs out of the extra space allocated to it, either
    - It will have to be moved to a hole with enough space or
    - Swapped out of memory until a large enough hole can be created or
    - Kill the process

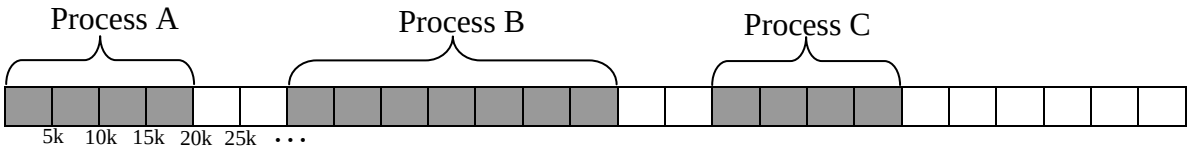
Memory usage management

- To keep free and allocated memory areas, it is possible to use an array data structure
- There are two ways to keep track of memory usage – bitmap and linked list

i. Bitmaps

- The memory is divided into fixed size allocation units
- Each allocation unit is represented with a bit in the bit map. If the bit is 0, it means the allocation unit is free and if it is 1, it means it is occupied

Example: Look at the following portion of a memory. Let the size of the allocation units is 5k as shown in the diagram.



- The corresponding bitmap for the above memory can be as follows:

11110011
11111001
11100000
...

- The size of the allocation unit is an important design issue
    - The smaller the allocation unit, the larger the bitmap size, which will result to memory and CPU overheads. Searching will take time.

    Example: Assume an allocation unit of 4bytes and assume the computer has 800KB of memory

    Number of bits required = 800KB/4B = 200Kbits

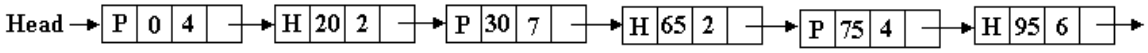
    Size of bitmap = 200Kbits/8 = 25KB

    Percentage of the memory used for the bitmap = (25/800)*100% ≈ 3%
- The larger the allocation unit, the larger the internal fragmentation. This is because a partition must be a multiple of allocation units where the minimum partition size is 1 allocation unit.

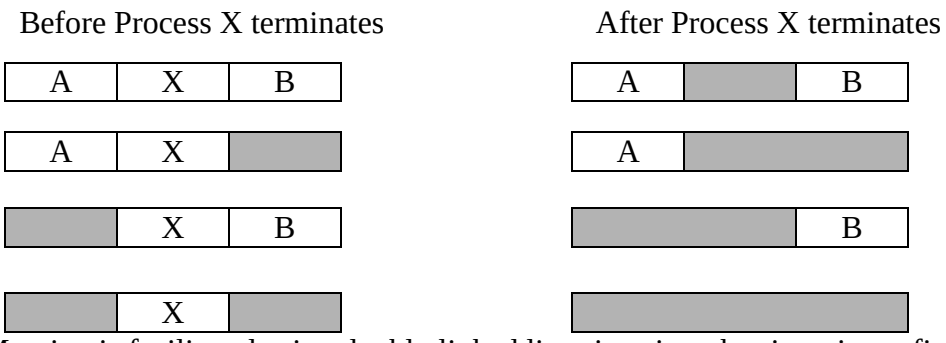
ii. Linked Lists

- A linked list of allocated and free memory segments is used
- Each segment is either a process or a hole between two processes and contains a number of allocation units
- Each entry in the list consists of
    - Segment type: P/H (1bit)
    - The address at which it starts
    - The length of the segment
    - A pointer to the next entry

Example: The memory in the bitmap example can be represented using linked list as follows:



- The list is kept sorted by address. Sorting this way has the advantage of easily updating when processes terminates or is swapped out.
- A terminating process normally has two neighbors, except when it is at the very beginning or end of the list. The neighbors can be either process or holes, leading to four combinations as shown below:



- Merging is facilitated using doubly linked list, since it makes it easier to find the previous entry and to see if a merge is possible.

Memory Allocation algorithms (Placement algorithms)

- When a new process is created or swapped in, the OS must decide which free bock to allocate.
- The following are some of the placement algorithms
  - **i. First-Fit**
    - It scans the memory from the beginning and chooses the first available block that is large enough
    - It is the simplest and fastest algorithm
    - Disadvantage: Searching from the beginning always may result in slow performance
  - **ii. Next Fit**
    - It starts scanning the memory from the location of the last placement and chooses the next available block that is large enough
    - **Disadvantage**: It may cause the last section of memory to be quickly fragmented
  - **iii. Best Fit**
    - It searches the entire list of available holes and chooses the block that is closest in size to the request
    - **Disadvantage**: External fragmentation and slower because it scans the whole list
  - **iv. Worst Fit**
    - It searches for the largest available hole, so that the hole broken off will be big enough to be useful
    - **Disadvantage**: It is as slow as best fit
  - **v. Quick Fit**
    - It maintains separate list for some of the more common size requests
    - **Advantage**: Finding a hole of the required size is extremely fast
    - **Disadvantage**: When a process terminates or is swapped out, finding its neighbors to see if a merge is possible is expensive

Summary

Advantage

- Dynamic memory allocation provides a better memory utilization

Disadvantage

- Management is more complex
- Includes memory compaction overhead

3.2.2. Relocation and Protection

- Multiprogramming introduces two essential problems that must be solved – **relocation** and **protection**

A. Protection

- Programs in other processes should not be able to reference memory locations in a process for reading or writing purposes without permission.
- A user program shouldn't be able to address a memory area which is not in its partition
    - A user process must not access any portion of the OS
    - Usually a program in one process can not branch to an instruction in another process
    - Without permission, a program in one process cannot access the data area of another process.

B. Relocation

- Location of a program in main memory is unpredictable due to loading, swapping and compaction

3.3 Virtual Memory

- A process may be larger than the available main memory
- In the early days **overlays** were used to solve this problem
    - The programmer will divide the program into modules and the main program is responsible to for switching the modules in and out of memory as needed.
    - Drawback: The programmer must know how much memory is available and it wastes the programmer time.
- In virtual memory the OS keeps those parts of the program currently in use in main memory and the rest on the disk. Piece of programs are swapped between disk and memory as needed.
- If a process encounters a logical address that is not in main memory, it generates an interrupt indicating a memory access fault and the OS puts the process in blocking state until it brings the piece that contains the logical address that caused the access fault.
- Program generated addresses are called virtual addresses and the set of such addresses form the **virtual address space**.
- The virtual address will go to the **Memory Management Unit (MMU)** that maps the virtual addresses into physical addresses.
- There are two virtual memory implementation techniques
    - Paging
    - Segmentation

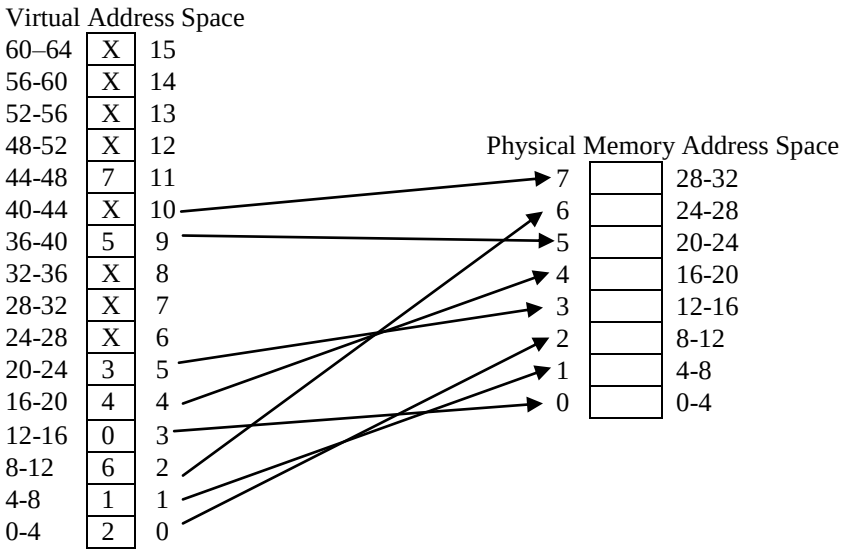
3.4. Paging and Segmentation

1. Paging

- The virtual address space is divided into units called **pages** and the memory is divided into units called **page frames**.
    - Pages and page frames should have the same size
    - Transfer between memory and disks are always in units of pages
    - The operation of dividing a process into pages is done by the compiler or MM

Example: Assume a 16-bit virtual address and each process is divided into 4K pages and the physical memory size is 32K

Virtual Address Space



Virtual Address = (4 bits page number, 12-bits offset)

Virtual Address Space=2^{16}=64K

Maximum Number of pages = 2^4=16pages

Example:

1. When the program tries to access address 0, for example, using the instruction - MOVE REG, 0
   The virtual address 0 is sent to MMU. The MMU sees that this virtual address falls in page 0 (0-4095), which according to its mapping is page frame 2 (8192 to 12287). It thus transforms the address to 8192 and outputs address 8192 onto the bus.
2. In the same way an instruction MOVE REG, 8196 is effectively transformed in to MOVE REG, 24580

Page Tables

- The OS maintains page table for mapping virtual addresses to physical addresses
- Page tables are stored in memory
- The OS also maintains all free-frames list of all frames in main memory that are currently unoccupied
- Each page table entry contains: **present-bit**, **modified-bit**, and **frame number**

Present-bit: Whether the page is present in main memory or not
Modified-bit: whether the contents of the corresponding page have been altered since the page was last loaded into main memory. If it is not altered, then it is not necessary to write the page out when it is to replace the page in the frame it occupies
Frame number: Indicates the corresponding page frame number in memory if its present-bit is 1, otherwise it is null.
Other Bits: Other control bits like protection and sharing bits

- If an instruction in a process tries to access an address in unmapped page
    - The MMU causes the CPU to generate an interrupt, called **page fault**
    - The OS puts the interrupted process in a blocking state and takes control
    - The OS chooses a page (page frame) to remove from memory and updates its contents back to the disk if it is modified since its last load
    - The OS issues a disk I/O read request to fetch the page just referenced into the page frame just freed and the OS can dispatch another process to run while the disk I/O is performed.
    - Once the desired page has been loaded into main memory, and I/O interrupt is issued, giving control back to the OS, which places the affected process back into a ready state.
    - When the process is dispatched, the trapped instruction is restarted

Page Size and mapping Speed

- There are two issues
    - The smaller the page size, the less number of internal fragmentation which optimizes the use of main memory
    - The smaller the page size, the greater the number of pages and thus the larger the page table
    - Every virtual memory can cause tow physical memory access: one to fetch the appropriate page table entry and one to fetch the desired data.

Page Replacement Algorithms (PRA)

Optimal PRA

- It says replace the page that will not be referenced soon
- Drawback – impossible to know whether the page will be referenced or not in the future

Not Recently Used PRA

- It says replace the page that is not referenced in the near past

First In First Out PRA

- Remove the oldest page

Least Recently Used PRA

- Remove pages that have less used in the past
- It assumes that pages that have been heavily used in the last few instructions will probably be heavily used again in the future

3.4.1. Segmentation

- The virtual space is divided into units called segments by the programmer
- Segments can have unequal and dynamic size
- Each segment has its own virtual address space
- Each segment consists of a linear sequence of addresses, form 0 to some maximum
- A process is loaded in partially (Only some of its segments)
- The OS maintains a segment table for each process and also a free-segment list
- It simplifies the handling of growing data structures
- When segmentation is used there is no internal fragmentation

Chapter 4: Input/output Management

4.1. Introduction

I/O is important for communication between the user and computer. The following are functions of the operating system as I/O manager:

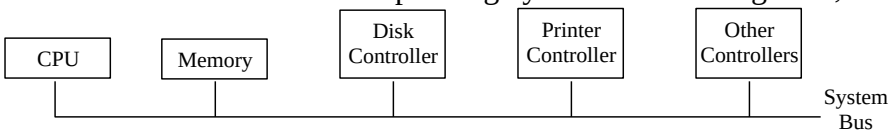
- Control all the computer I/O devices
- Issue commands to I/O devices, catch interrupts and handle errors
- Provide an interface between the device and the rest of the system

4.2 Principles of I/O hardware

I/O devices basics:

- I/O devices concerns with the way data are handled by the I/O device. There are two types of I/O devices – **Blocked devices** (such as disks) and **character devices** (such as printers, mouse and NIC)
    - **Block devices**: work on data of fixed size blocks with each block having a unique address, hence, are randomly accessible. Example: disks
    - **Character devices**: process stream of characters without any block structure. Example: printers, network interfaces, mice, keyboards.
- I/O devices have two major components – **Electronic Component** (Adapter/device controller) and the **Mechanical Component** (the moving parts)

A controller has an interface into which a cable that connects its device is plugged in. Each controller has a few registers for communication with the operating system. Besides registers, some devices have data



buffer on which programs and the operating system can read/write data. For the CPU to communicate with device controllers, each device must have a unique ID.

4.3 Principles of I/O software

- Layered technique is used
- Goals and issues of I/O software:
    1. **Device Independence**: It should be possible to write programs that can read files on a floppy disk, on hard disk, or on a CD-ROM, without having to modify the program for each different device types. It is up to the operating system to take care of the problems caused by the fact that these devices are really different.
    2. **Uniform Naming:** the name of a file or a device should simply be a string or an integer and not dependant on the device in any way.
    3. **Error handling:** In general, errors should be handled as close to the hardware as possible (at the device controller level). Many errors are transient, such as read errors caused by specks of dust on the read head, and will go away if the operations are repeated.

4. **Transfer:** There are two types of transfer modes – **Synchronous (Blocking)** and **Asynchronous (interrupt –driven)**. In the case of synchronous transfer the program requesting I/O transfer will be suspended until the transfer is completed. In the case of Asynchronous transfer the CPU starts the transfer and goes off to do something until the interrupt that shows the completion of the transfer arrives.

Layers of I/O software

- The following are the I/O software layers
  - Interrupt handler (bottom)
  - Device driver
  - Device independent OS software
  - User-level software (top)

1. **Interrupt Handler**
   - Interrupts should be hidden
   - The best way to hide them is to have every process starting an I/O operation block until the I/O has completed and the interrupt occurs.
   - When the interrupt happens, the interrupt procedure does whatever it has to do in order to unblock the process that started it.

2. **Device Driver**
   - All device – dependent code goes in the device driver
   - Each device driver handles one device type, or at most, one class of closely related devices.
   - The device driver issues the commands and check that they are carried out properly.
   - Thus, the disk driver is the only part of the OS that knows how many registers that disk controller has and what they are used for.
   - In general terms, the job of a device driver is to accept requests from the device-independent software above it and sees to it that the request is executed.

3. **Device Independent I/O Software**
   - It is large fraction of I/O software
   - Functions of the device-independent I/O software

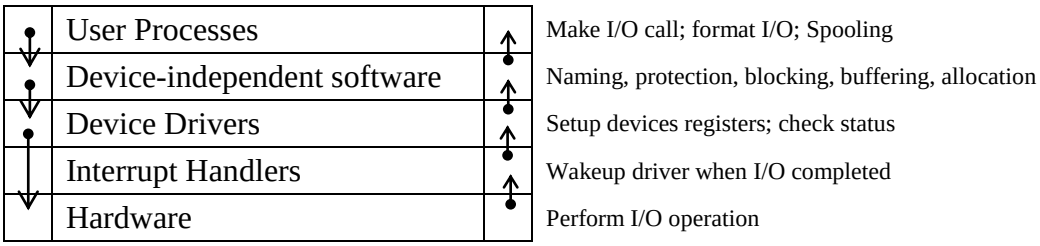
1. **Uniform interfacing for device drivers** – perform I/O function common to all drives
2. **Device Naming** – responsible for mapping symbolic devices names onto the proper driver

3. **Device protection** – prevent users from accessing devices that they are not entitled to access
4. **Buffering**: if a user process write half a block, the OS will normally keep the data in buffer until the rest of the data are written. Keyboard inputs that arrive before it is needed also require buffering.
5. **Storage allocation on block devices:** when a file is created and filled with data, new disk blocks have to be allocated to the file. To perform this allocation, the OS needs a list of free blocks and used some algorithms for allocation
6. **Allocating and releasing dedicated devices**: It is up to the OS to examine requests for devices usage and accept or reject them.
7. **Error reporting:** Errors handling, by and large, is done by drivers. Most errors are device dependent. After the driver tries to read the block a certain number of times, it gives up and informs the device-independent software. It then reports to the caller.

4. User Space I/O Software

- A small portion of the I/O software is outside the OS
- System calls, including the I/O system calls, are normally made by library procedures
- Formatting of input and output is done by library procedures

The following figure shows the I/O system layers and the main functions of each layer



4.4 Disk Arm Scheduling

Disk:

- All real disks are organized into cylinders, each one containing as many tracks as there, are heads stacked vertically
- Each of the tracks then will be divided into sectors (equal number of sectors or different number of sectors)
- In the case of equal number of sectors
  - The data density as closer to the center (hub) is high
  - The speed increases as the read/write moves to the outer tracks
- Modern large hard drives have more sectors per track on outer tracks e.g. IDE drives

- Many controllers can read or write on one drive while seeking on one or more other drives, but floppy disk controller cannot do that.

Disk Access Time

The time required to read or write a disk block is determined mainly by the following three parameters:

- Seek time (the time to move the arm to the proper cylinder)
- Rotational delay (the time to rotate the disk and position the required sector under the read/write head)
- Actual data transfer time

For most disks seek time is the most dominant time. Reducing the seek time can improve system performance. Many disk controller drivers maintain a table of requests indexed by cylinder number. Under these circumstances, some possible disk scheduling algorithms to reduce the average seek time are:

First-Come, First-Served (FCFS): the disk requests are served in the order they arrive. The response time is poor.

Shortest Seek First (SSF): the next cylinder to be accessed is the nearest to the current one. This algorithm minimizes average seek time but cylinders at the extreme locations suffers long waits since most requests tend to cylinders near the middle. Thus, minimal response time compromises fairness. The following figure illustrates shortest seek first disk scheduling algorithm.

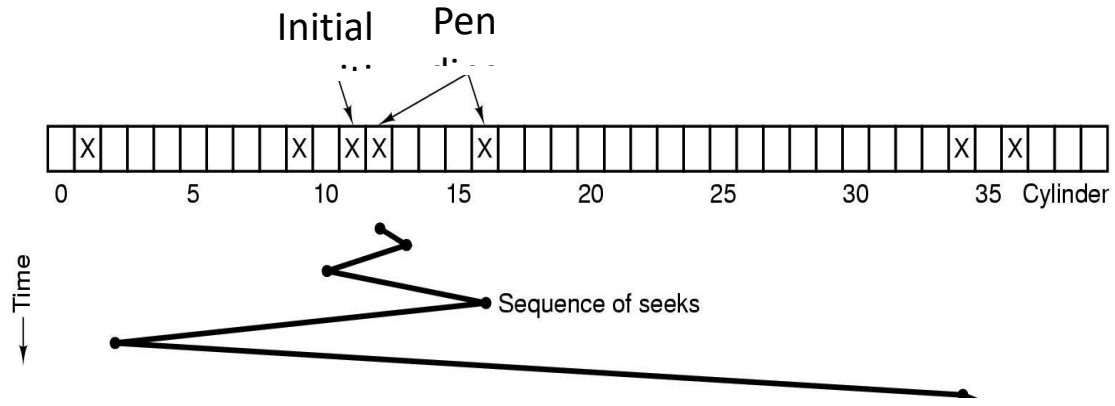


Figure: Shortest seek first (SSF) disk scheduling algorithm

Elevator Algorithm: the next cylinder to be accessed is determined by the direction of search. If the current direction is toward the outer cylinders, the next cylinder is one with the next higher number. If the direction is downward, the next cylinder is one with the next smaller number. The search direction is reversed when there are no more requests in the current direction. In general, the average seek time of the elevator algorithm is higher than that of SSF, but has better fairness. The following figure shows the elevator disk scheduling algorithm assuming that the search direction is toward the outer cylinders.

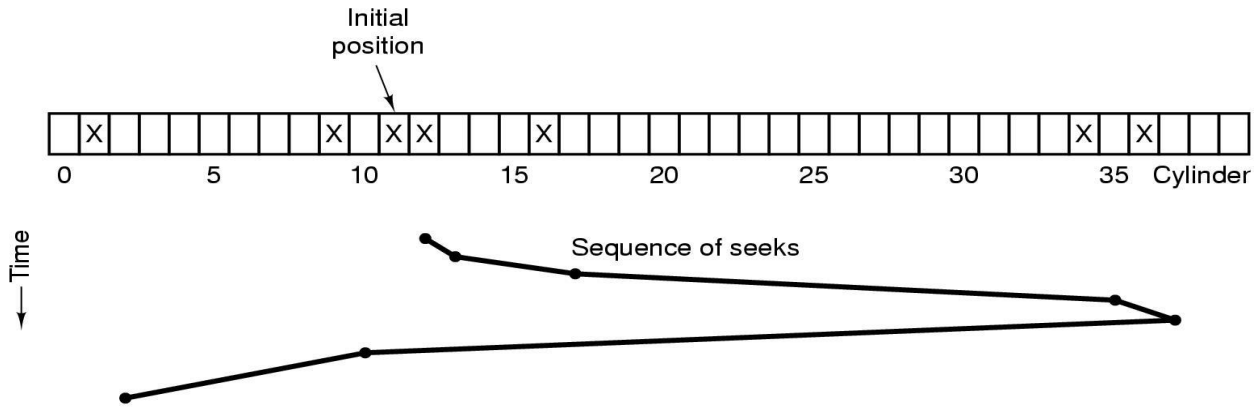


Figure: The elevator algorithm disk scheduling algorithm

Chapter 5
File System Management

Overview of File System

A file is a collection of related information recorded on the secondary storage. For example, file containing student information, file containing employee information, files containing C source code and so on. A file is thus the smallest allotment of logical secondary storage, that is any information to be stored on the secondary storage need to be written on to a file and the file is to be stored. Information in files could be program code or data in numeric, alphanumeric, alphabetic or binary form either formatted or in free form. A file is therefore a collection of records if it is a data file or a collection of bits / bytes / lines if it is code.

You used several files to store information on your computer as the information in them can later be retrieved and used. It is important that the files are managed properly so that they can be located and accessed easily without consuming time.

To help you to organize and manage the files and directories on your computer, an os uses an application called a file manager. A file manager enables you to create, delete, copy, move, rename and view files and create and manage directories (folders). The way in which an os organizes, names, protects, accesses, and uses files is called a file management system.

Files are required for the following reasons.

- store data permanently (even after the termination of the process that creates them)
- store large data that is beyond the size of a process's address space
- enable multiple processes to access data simultaneously as in the case of databases

Part of the operating system that deals with files is known as the file system. Users and designers of file systems have different views of files.

- ✓ From the users' point of view, the most important issue is the interface (naming, protection, and operations allowed.)
- ✓ From the designers' point of view, the choice between the use of linked lists versus bit maps and the number of tracks per block has more importance.

Files and Directories – Interface

Files

File Naming

Name is used to identify a file with an abstract way. A file name consists of strings. The maximum allowed length of the string and case sensitivity of names vary from one operating system to the other. Most operating systems have some characters, called file extension, as part of the file name following a period (.). In some operating systems (such as Microsoft Windows), the file extension is used to associate the file with a program while in others, it has no special use. Some characters, e.g. /, \ >, have special meaning in the file system so that they cannot be used as part of the file name.

File Access

- *Sequential access*: the bytes or records of the file are read from beginning to end sequentially
- *Random access*: the bytes or records of the file can be read out of order

File Attributes

A file has a number of attributes. The available attributes of files differ from one operating system to the other. The most common attributes are:

- Attributes for ownership, access permission, and password
- Flags (bits) for archive, hidden, system, read only, temporary, etc.
- File size
- Time of creation, last access, and last modification

File Operations

In order to provide the basic store and retrieval operations of a file and other properties of files outlined above, operating systems have some common file operations. Some of them are: Create, delete, open, close, read, write, append, seek, get attributes, set attributes, and rename.

Directories

To keep track of files, file systems normally have directories (folders). In most operating systems directories themselves are files.

Single Level Directory Systems

There is only one directory (the root directory) in the system. The advantage of this scheme is its implementation simplicity and ease of locating files. The main disadvantage is the need to

provide distinct name for each file. This kind of directory organization was used in early operating systems.

Two Level Directory Systems
In a multi user system, it is more convenient to have a directory for each user under the root directory and a system directory to store system level files. If only a file name is provided, it refers to a file in the users' own directory, but if a file in another user's directory is needed, the user's directory name has to be supplied along with the file name.

Hierarchical Directory Systems
To give users the ability to organize their files arbitrarily, it is important to have a file system with a general tree structure. The user is allowed to create directory structures of arbitrary levels deep.

Directory Operations
The type of operations on directories varies from one operating system to the other. The most common operations are: create, delete, opendir, closedir, and rename.

Files and Directories – Implementation

Implementing Files

The most important issue of file implementation is to keep track of which disk blocks go to which file. There are several alternative ways of allocating disk space for files.

Contiguous Allocation
A file is stored in consecutive disk blocks, as depicted in Figure 5.1.

Advantage:

- *Simple to implement*: what we keep track of is the first block of a file and the number of blocks used by the file.
- *High performance*: when accessing a file, the only delay is while searching the first block of the file.

Disadvantage:

- The file size must be specified at the time of file creation.
- The disk would eventually have many holes (unused blocks) as files are deleted. Thus the disk may be full while there are holes at many locations which are small to contain a file of specified length.

Linked List Allocation
Files are implemented as a linked list of disk blocks. The first word of a block is used to point to the next block and the rest of the block is used to store data. Figure 5.2 shows implementation of file allocation as a linked list of disk blocks.

Advantage:

- Disk usage is effective; there is no fear of losing disk blocks to holes.
- It is enough to store only the address of the first block of a file in the directory.

Disadvantage:

- Random access of a file is slow since file access always begins at the first block.
- The amount of data stored in a block is not a power of two as some space in each block is allocated for the pointer to the next block. Since many programs read/write data with size of a power of two, it may be necessary to read two consecutive blocks and concatenate the data.

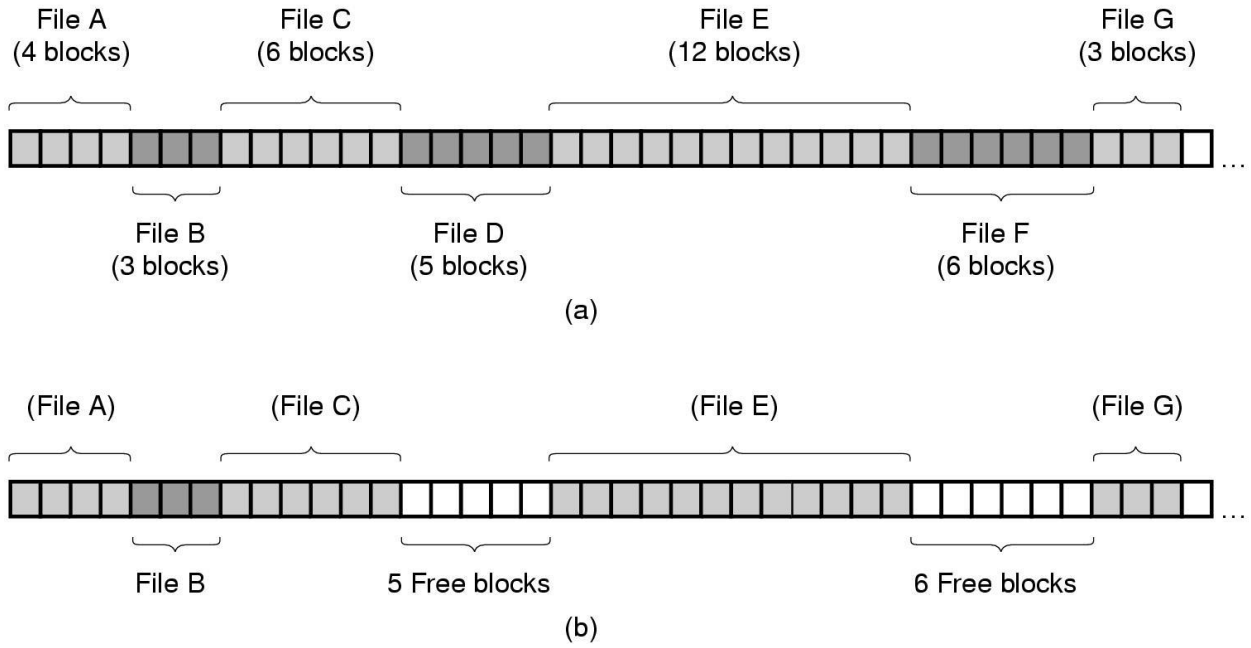


Figure 5.1 (a) Contiguous allocation of disk space for seven files (b) state of disk space after files D and F have been removed

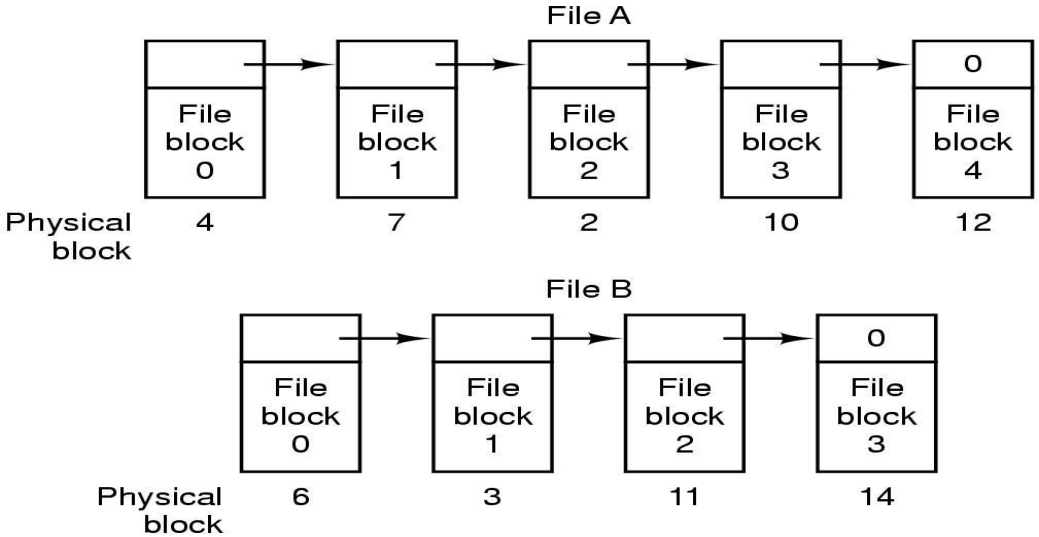


Figure 5.2 Implementation of two files (File A and File B) with linked list of disk blocks

Linked List Allocation Using a Table in Memory

The disadvantages of linked list allocation can be eliminated by moving the block pointers from the blocks to a table, called file allocation table (FAT), and storing the table in memory. Figure 5.3 illustrates a FAT. Still storing the address of the first block is sufficient. The chain of blocks is maintained as entries of the table with a special value to indicate the end of the chain. Random access becomes faster. The main disadvantage is much memory is required to store the table. Moreover, the size of the table grows linearly with disk size.

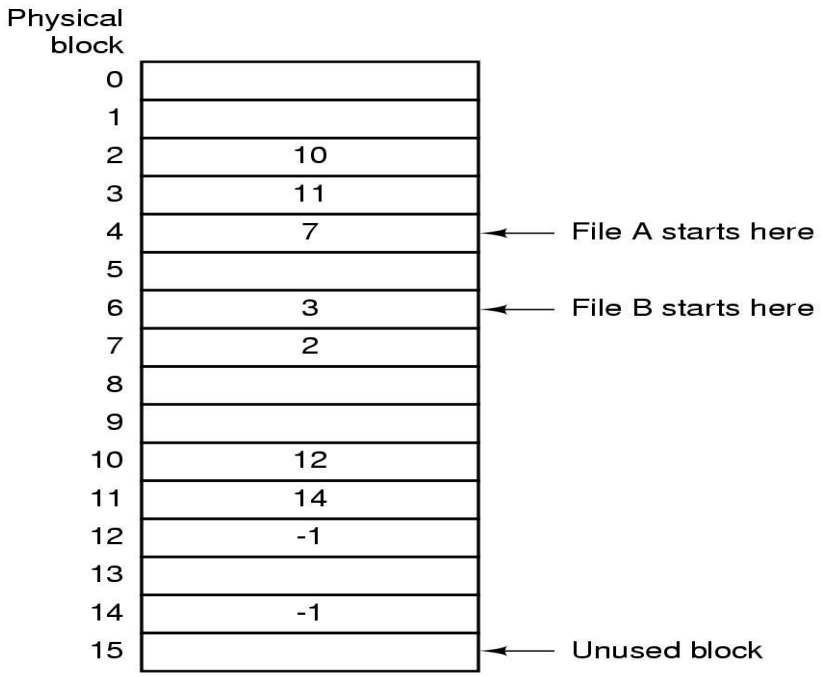


Figure 5.3 Implementation of a file allocation table in memory for the files in Figure 4.2

I-Nodes

An i-node (index node) stores the addresses of disk blocks and attributes of a file. The advantage of i-nodes is that only the i-nodes of opened files are needed to be in memory. If the size of a file is too large to be stored in a disk space pointed by a single i-node, it is possible to use some of the last i-node entries to point to a block with more disk block addresses as shown in Figure 5.4.

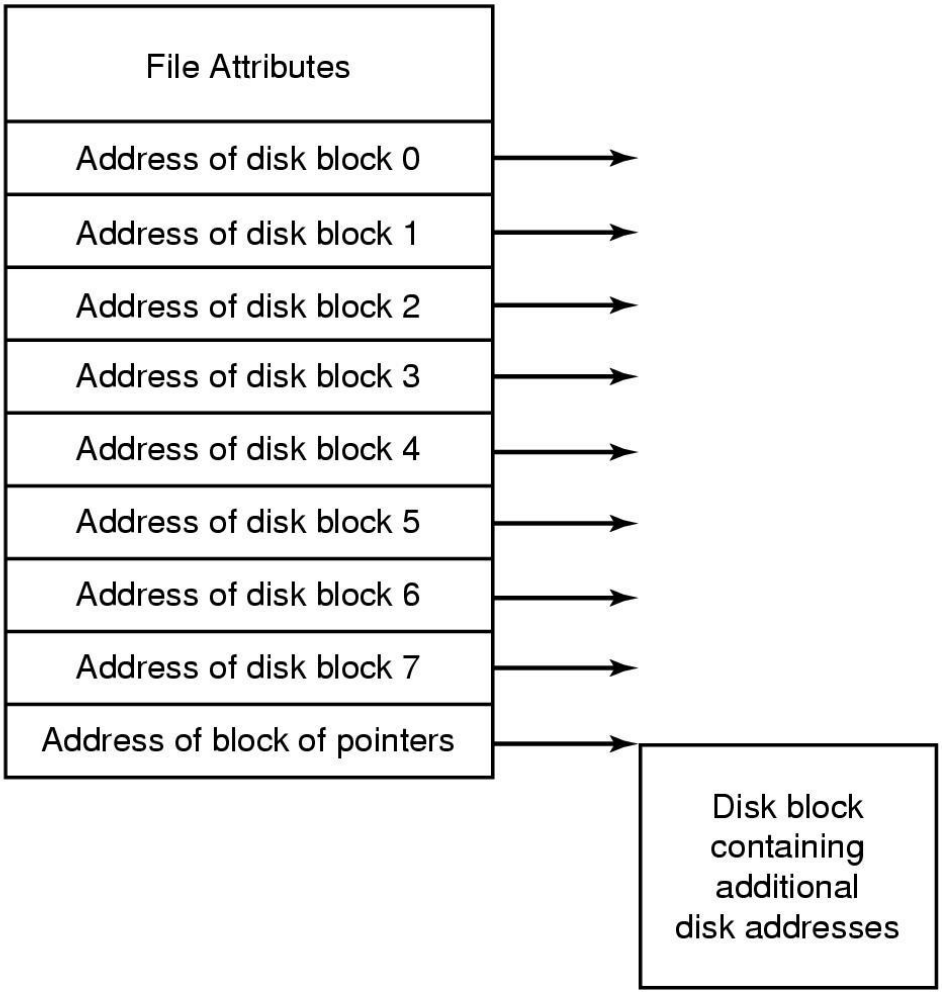


Figure 5.4 An example of an i-node for a file