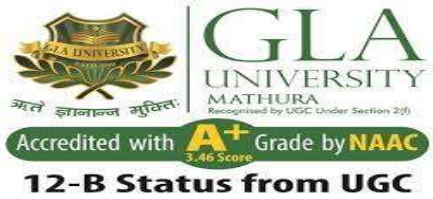
Introduction of Operating System

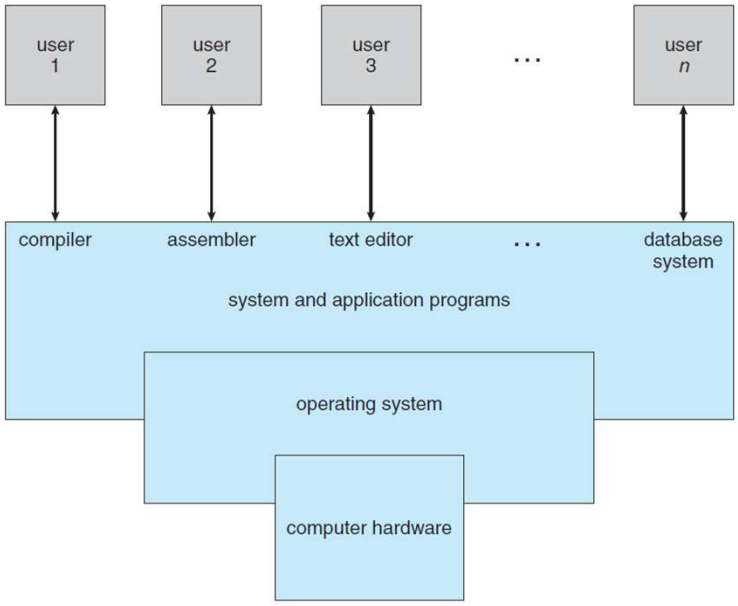


Presented by:
Dr. Premnarayan Arya
Assistant Professor
Department of CEA, GLA University, Mathura

What is Operating System?

- Operating System provides an **interface** between user and hardware.

Abstract view of the components of a computer system



Functions of Operating System

- **Processor Management:** An operating system manages the processor's work by allocating various jobs to it and ensuring that each process receives enough time from the processor to function properly.
- **Memory Management:** An operating system manages the allocation and deallocation of the memory to various processes and ensures that the other process does not consume the memory allocated to one process.

Functions of OS cont…

- **Device Management:** There are various input and output devices. An OS controls the working of these input-output devices. It receives the requests from these devices, performs a specific task, and communicates back to the requesting process.
- **File Management:** An operating system keeps track of information regarding the creation, deletion, transfer, copy, and storage of files in an organized way. It also maintains the integrity of the data stored in these files, including the file directory structure, by protecting against unauthorized access.

Functions of OS cont…

- **Security:** The operating system provides various techniques which assure the integrity and confidentiality of user data. Following security measures are used to protect user data:
  - Protection against unauthorized access through login.
  - Protection against intrusion by keeping Firewall active.
  - Protecting the system memory against malicious access.
  - Displaying messages related to system vulnerabilities.

Functions of OS cont…

- **Error Detection:** From time to time, the operating system checks the system for any external threat or malicious software activity. It also checks the hardware for any type of damage. This process displays several alerts to the user so that the appropriate action can be taken against any damage caused to the system.
- **Job Scheduling:** In a multitasking OS where multiple programs run simultaneously, the operating system determines which applications should run in which order and how time should be allocated to each application.

Features of the Operating System

- Provides a platform for running applications
- Handles memory management and CPU scheduling
- Provides file system abstraction
- Provides networking support
- Provides security features
- Provides user interface
- Provides utilities and system services
- Supports application development

Thank You

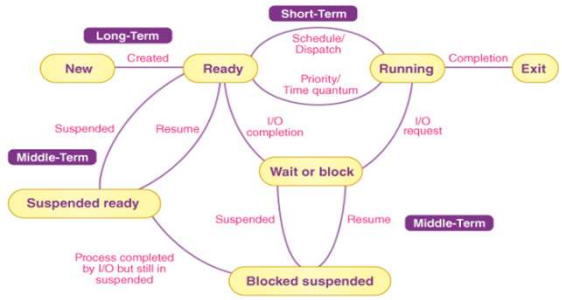
Operating System

Presented by:
Dr. Premnarayan Arya
Assistant Professor
Department of CEA, GLA University, Mathura

Contents

- CPU Scheduling
- Important Terminologies in CPU Scheduling
- CPU Scheduling Criteria
- Types of CPU scheduling Algorithms
- Performance Criteria

Diagram of Process Schedulers



CPU Scheduling

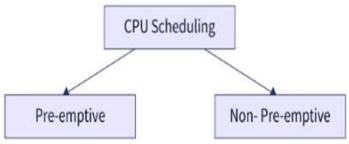
- **In the Uniprogrammming systems** like MS DOS, when a process waits for I/O operation, the CPU remains idle.
- This is an overhead since it wastes the time and causes the problem of starvation.

CPU Scheduling

- In **Multiprogramming systems**, the CPU does not remain idle during the waiting time of the Process and it starts executing of other processes.
- The Operating system **schedules** the processes to the CPU for maximum utilization.
- This procedure is called **CPU scheduling**. The Operating System uses various scheduling algorithm to schedule the processes.

CPU Scheduling

- There are mainly two types of CPU scheduling:

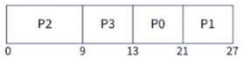


Non-Preemptive Scheduling

- The new processes are executed only after the current process has completed its execution.
- The process holds the resources of the CPU (CPU time) till its state changes to terminated.
- If a process is currently being executed by the CPU, it is not interrupted till it is completed. Once the process has completed its execution, the processer picks the next process from the ready queue.

Example of Non-Preemptive Scheduling

Process	Arrival Time	CPU Burst Time (in millisecond)
P0	2	8
P1	3	6
P2	0	9
P3	1	4

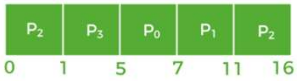


Preemptive Scheduling

- Some processes could have a **higher priority** and hence must be executed before the processes that have a lower priority.
- In preemptive scheduling, the CPU resource is allocated to a process for only a limited period of time and then those resources are taken back and assigned to another process.

Example of Preemptive Scheduling

Process	Arrival Time	CPU Burst time (in millisec.)
P0	3	2
P1	2	4
P2	0	6
P3	1	4



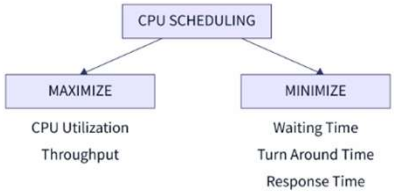
Important Terminologies in CPU Scheduling

- **Arrival time:** Arrival time (AT) is the time at which a process arrives at the ready queue.
- **Burst Time:** It is the time required by the CPU to complete the execution of a process, It is also sometimes called the execution time or running time.
- **Completion Time:** The time when a process completes its execution. It is not to be confused with burst time.

Important CPU Scheduling Terminologies

- **Turn-Around Time:** TAT is the difference between completion time and arrival time (Completion time - arrival time).
- **Waiting Time:** WT of a process is the difference between turn around time and burst time (TAT - BT), i.e. the amount of time a process waits for getting CPU resources in the ready queue.
- **Response Time:** RT of a process is the time after which any process gets CPU resources allocated after entering the ready queue.

CPU Scheduling Criteria



CPU Scheduling Criteria

- **Arrival time:** Arrival time is the point of time at which a process enters the ready queue.
- **Throughput:** It is the total number of processes that are completed or executed per unit of time or, it is the total work done in a unit of time by the CPU. An algorithm must work to maximize throughput.

CPU Scheduling Criteria

- **Completion Time:** Completion time is the point of time at which a process completes its execution on the CPU and takes exit from the system. It is also called as **exit time**.

CPU Scheduling Criteria

- **Burst time :** Burst time is the amount of time required by a process for executing on CPU.
- It is also called as **execution time** or **running time**.
- Burst time of a process can not be known in advance before executing the process.
- It can be known only after the process has executed.

CPU Scheduling Criteria

- **Turn Around Time:** Turn Around time is the total amount of time spent by a process in the system. When present in the system, a process is either waiting in the ready queue for getting the CPU or it is executing on the CPU.

Turn Around time = Burst time + Waiting time

Or

Turn Around time = Completion time – Arrival time

CPU Scheduling Criteria

- **Waiting Time:** Waiting time is the amount of time spent by a process waiting in the ready queue for getting the CPU.

Waiting time = Turn Around time – Burst time

CPU Scheduling Criteria

- **Response Time:** Response time is the amount of time after which a process gets the CPU for the first time after entering the ready queue.

Response Time = Time at which process first gets the CPU – Arrival time

Types of CPU scheduling Algorithms

- **First-Come, First-Served (FCFS):** The processes are executed in the order in which they arrive.
- **Shortest Job First (SJF):** This algorithm schedules processes based on their expected CPU burst time. The process have shortest time is executed first.
- **Priority Scheduling:** This algorithm assigns a priority value to each process, the high priority processes are executed first.

Types of CPU scheduling Algorithms

- **Round Robin (RR):** This algorithm allocates a fixed time slice (time quantum) to each process in a cyclic order. If a process has not completed its execution within its allotted time slice, it is preempted, and the next process in the queue is executed.
- **Multilevel Queue Scheduling:** This algorithm divides the processes into separate queues based on their characteristics. Each queue has its own scheduling algorithm, and processes are moved between queues based on their priority or other criteria.

Types of CPU scheduling Algorithms

- **Multilevel Feedback Queue Scheduling:** This algorithm is an extension of multilevel queue scheduling that allows processes to move between queues based on their behavior. For example, a process that uses a lot of CPU time may be moved to a lower-priority queue to make room for other processes.

Performance Criteria

- CPU scheduling is the task performed by the CPU that decides the way and order in which processes should be executed. There are two types of CPU scheduling - Preemptive, and non-preemptive. The criteria the CPU takes into consideration while "scheduling" these processes are - CPU utilization, throughput, turnaround time, waiting time, and response time.

Multi-processor Scheduling

- A Multi-processor is a system that has more than one processor but shares the same memory, bus, and input/output devices.
- In multi-processor scheduling, more than one processors(CPUs) share the load to handle the execution of processes smoothly.
- The scheduling process of a multi-processor is more complex than that of a single processor system.

Multi-processor Scheduling

- The CPUs may be of the same kind (homogeneous) or different (heterogeneous).
- The multiple CPUs in the system share a common bus, memory, and other I/O devices.
- There are two approaches to multi-processor scheduling - symmetric and asymmetric Multi-processor scheduling.

Thank You

Operating System

Presented by:
Dr. Premnarayan Arya
Assistant Professor
Department of CEA, GLA University, Mathura

Contents

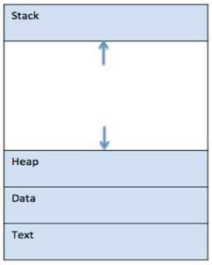
- Process
- Program
- Difference b/w Program & Process
- Process States
- Process Schedulers
- PCB

Process

- A process is a **sequence of instructions** executed in a predefined order. Any program that is executed is called a **process.** Process changes **state** when it executes and can be either new, ready, running, waiting or terminated.

Process Cont…

- When a program is loaded into the memory and it becomes a process, it can be divided into four sections, stack, heap, data and text.



Process Cont…

- **Stack:** The Stack contains the temporary data such as method/function parameters, return address and local variables.
- **Heap:** This is dynamically allocated memory to a process during its run time.
- **Text:** This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.
- **Data:** This section contains the global and static variables.

Program

- A program is a piece of code which may be a single line or millions of lines.

```
#include <stdio.h>
int main()
{
   printf("GLAU");
   return 0;
}
```

Difference between Program and Process

Features	Program	Process
Definition	A program is a set of instructions that are used to perform a certain task.	A process is a program being executed.
Lifespan	A program is stored in the secondary memory and has a higher lifespan.	A process is active until the program is executed and has a short lifespan.

Difference between Program and Process

Features	Program	Process
Nature	A program will be inactive unless it is being used hence it is considered as a passive entity	A process gets activated when a program is executed therefore it is an active entity
Overhead (processing time required by system software)	A program does not have a significant overhead.	A process has considerable overhead.

Difference between Program and Process

Features	Program	Process
Computation Time	A program does not have computational time and cost.	A process consumes significant computational time to execute.
Resource usage	A program does not have resource requirements. A program only needs memory space to store all the instructions.	A process needs high resources as compared to a program, it requires CPU, disk, Input/Output, memory address, etc.

Process States or Life Cycle

- **New (Create):** The process is created but not yet created. It is the program that is present in secondary memory that will be picked up by OS to create the process.

Process States Cont…

- **Ready:** After the creation of a process, the process enters the ready state i.e. the process is loaded into the main memory. The process here is ready to run and is waiting to get the CPU time for its execution. Processes are ready for execution by the CPU are maintained in a queue called ready queue.

Process States Cont…

- **Run:** The process is chosen from the ready queue by the CPU for execution and the instructions within the process are executed by any one of the available CPU cores.

Process States Cont…

- **Blocked or Wait:** Whenever the process requests access to I/O or needs input from the user or needs access to a critical region (the lock for which is already acquired) it enters the blocked or waits for the state. The process continues to wait in the main memory and does not require CPU. Once the I/O operation is completed the process goes to the ready state.

Process States Cont…

- **Terminated or Completed:** Process is killed as well as **PCB** is deleted. The resources allocated to the process will be released or deallocated.

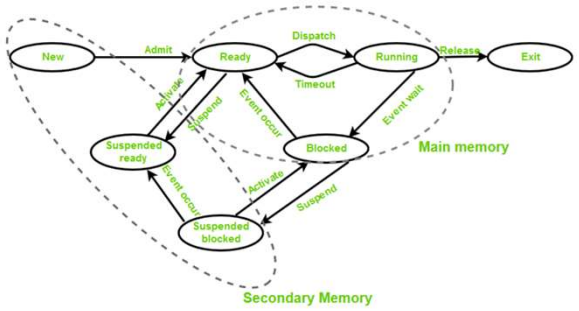
Process States Cont…

- **Suspend Ready:** Process that was initially in the ready state but was swapped out of main memory and placed onto external storage by the scheduler is said to be in suspend ready state. The process will transition back to a ready state whenever the process is again brought onto the main memory.

Process States Cont…

- **Suspend wait or suspend blocked:** Similar to suspend ready but uses the process which was performing I/O operation and lack of main memory caused them to move to secondary memory. When work is finished it may go to suspend ready.

Process States Diagram



Types of Schedulers

- Long-Term Scheduler or Job Scheduler
- Short-Term Scheduler or CPU Scheduler
- Medium-Term Scheduler

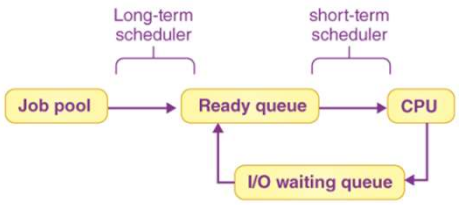
Long-Term Scheduler or Job Scheduler

- The primary objective of long-term scheduler is to **maintain a good degree of multiprogramming**.
- Long-term scheduler is also known as **Job Scheduler**.
- It selects a balanced mix of I/O bound and CPU bound processes from the secondary memory (new state).
- Then, it loads the selected processes into the main memory (ready state) for execution.

Short-Term Scheduler or CPU Scheduler

- The primary objective of short-term scheduler is to **increase the system performance**.
- Short-term scheduler is also known as **CPU Scheduler**.
- It decides which process to execute next from the ready queue.
- After short-term scheduler decides the process, **Dispatcher** assigns the decided process to the CPU for execution.

Short-Term Scheduler or CPU Scheduler



Medium-Term Scheduler

- The primary objective of medium-term scheduler is to perform **swapping**.
- Medium-term scheduler swaps-out the processes from main memory to secondary memory to free up the main memory when required.
- Thus, medium-term scheduler reduces the degree of multiprogramming.
- After some time when main memory becomes available, medium-term scheduler swaps-in the swapped-out process to the main memory and its execution is resumed from where it left off.
- Swapping may also be required to improve the process mix.

Medium-Term Scheduler

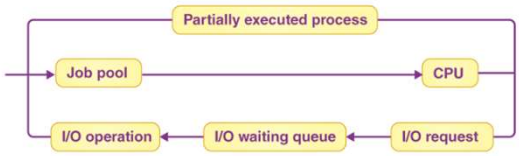
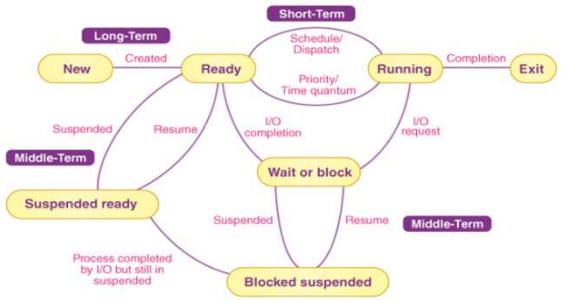


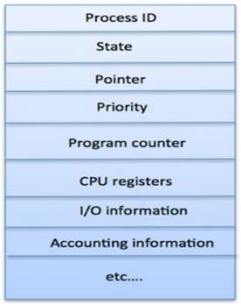
Diagram of Process Schedulers



Process Control Block (PCB)

- When the process is created by the operating system it creates a data structure to store the information of that process. This is known as Process Control Block (PCB). Process Control block (PCB) is a data structure that stores information of a process.
- The architecture of a PCB is completely dependent on Operating System and may contain different information in different operating systems.

Diagram of PCB



PCB Cont…

- **Process State:** The current state of the process i.e., whether it is ready, running, waiting, or whatever.
- **Process privileges:** This is required to allow/disallow access to system resources.
- **Process ID:** Unique identification for each of the process in the operating system.

PCB Cont…

- **Pointer:** A pointer to parent process.
- **Program Counter:** Program Counter is a pointer to the address of the next instruction to be executed for this process.
- **CPU registers:** Various CPU registers where process need to be stored for execution for running state.
- **CPU Scheduling Information:** Process priority and other scheduling information which is required to schedule the process.

PCB Cont…

- **Memory management information:** This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.
- **Accounting information:** This includes the amount of CPU used for process execution, time limits, execution ID etc.
- **IO status information:** This includes a list of I/O devices allocated to the process.

Multiprogramming

We have many processes ready to run. There are two types of multiprogramming:

1. **Preemption:** Process is forcefully removed from CPU. Pre-emotion is also called time sharing or multitasking.
2. **Non-preemption:** Processes are not removed until they complete the execution. Once control is given to the CPU for a process execution, till the CPU releases the control by itself, control cannot be taken back forcibly from the CPU.

Degree of Multiprogramming

- Multiple processes may be present in the **ready state** which are all ready for execution.
- Degree of multiprogramming is the **maximum number of processes that can be present in the ready state**.
- Long-term scheduler **controls** the degree of multiprogramming.
- Medium-term scheduler **reduces** the degree of multiprogramming.

Thank You

Operating System

Presented by:
Dr. Premnarayan Arya
Assistant Professor
Department of CEA, GLA University, Mathura

Contents

FCFS scheduling Algorithms

- The process which **arrives first** in the ready queue is firstly assigned to the CPU.
- In case of a tie, process with smaller process id is executed first.
- It is non-preemptive algorithm.
- Real life example of FCFS scheduling is **buying tickets at ticket counter**.

Advantages of FCFS scheduling Algorithms

- It is simple and easy to understand.
- It can be easily implemented using queue data structure.
- It does not lead to starvation.

Disadvantages of FCFS scheduling Algorithms

- It does not consider the priority or burst time of the processes.
- It suffers from **convoy effect**.

Convoy Effect

- Convoy Effect is a situation where many processes, who need to use a resource for short time are blocked by one process holding that resource for a long time.
- The higher burst time process arrived before the smaller burst time process. Then, the smaller processes have to wait for a long time for longer process to release the CPU.

Problem -1 based on FCFS scheduling Algorithms

- Consider the set of 5 processes whose arrival time and burst time are given below:

Process Id	Arrival time	Burst time
P1	3	4
P2	5	3
P3	0	2
P4	5	1
P5	4	3

- If the CPU scheduling policy is FCFS, calculate the average waiting time and average turn around time.

Solution

- **Gantt Chart-**

- Here, black box represents the idle time of the CPU.

Solution

Now calculate,

- Turn Around time = Exit time – Arrival time
- Waiting time = Turn Around time – Burst time

Process Id	Exit time	Turn Around time	Waiting time
P1	7	7 – 3 = 4	4 – 4 = 0
P2	13	13 – 5 = 8	8 – 3 = 5
P3	2	2 – 0 = 2	2 – 2 = 0
P4	14	14 – 5 = 9	9 – 1 = 8
P5	10	10 – 4 = 6	6 – 3 = 3

Now,

Average Turn Around time = (4 + 8 + 2 + 9 + 6) / 5 = 29 / 5 = 5.8 unit

Average waiting time = (0 + 5 + 0 + 8 + 3) / 5 = 16 / 5 = 3.2 unit

Problem -2 based on FCFS scheduling Algorithms

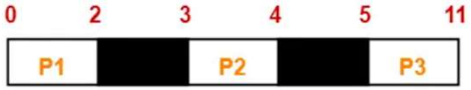
- Consider the set of 3 processes whose arrival time and burst time are given below;

Process Id	Arrival time	Burst time
P1	0	2
P2	3	1
P3	5	6

- If the CPU scheduling policy is FCFS, calculate the average waiting time and average turn around time.

Solution

- **Gantt Chart-**



- Here, black box represents the idle time of the CPU.

Solution

Now calculate-

- Turn Around time = Exit time – Arrival time
- Waiting time = Turn Around time – Burst time

Process Id	Exit time	Turn Around time	Waiting time
P1	2	2 – 0 = 2	2 – 2 = 0
P2	4	4 – 3 = 1	1 – 1 = 0
P3	11	11- 5 = 6	6 – 6 = 0

Now,

Average Turn Around time = (2 + 1 + 6) / 3 = 9 / 3 = 3 unit

Average waiting time = (0 + 0 + 0) / 3 = 0 / 3 = 0 unit

Problem -3 based on FCFS scheduling Algorithms

- Consider the set of 6 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time
P1	0	3
P2	1	2
P3	2	1
P4	3	4
P5	4	5
P6	5	2

- If the CPU scheduling policy is FCFS and there is 1 unit of overhead in scheduling the processes, find the efficiency of the algorithm.

Solution

- **Gantt Chart-**

- Here, δ denotes the context switching overhead.

Solution

Now,

- Useless time / Wasted time = 6 x δ = 6 x 1 = 6 unit
- Total time = 23 unit
- Useful time = 23 unit – 6 unit = 17 unit

Efficiency (η) = Useful time / Total Time
= 17 unit / 23 unit
= 0.7391
= 73.91%

Shortest Job First(SJF)/Shortest Remaining Time First(SRTF) Scheduling

- Among all available processes, CPU is assigned to the process having smallest burst time. In case of a tie, then it followed FCFS Scheduling.

SJF/SRTF Scheduling

- SJF Scheduling can be used in both preemptive and non-preemptive mode.
- Preemptive mode of Shortest Job First is called as **Shortest Remaining Time First (SRTF)**.

Advantages of SJF/SRTF Scheduling

- SRTF is optimal and guarantees the minimum average waiting time.
- It provides a standard for other algorithms since no other algorithm performs better than it.

Disadvantages of SJF/SRTF Scheduling

- It can not be implemented practically since burst time of the processes can not be known in advance.
- It leads to starvation for processes with larger burst time.
- Priorities can not be set for the processes.
- Processes with larger burst time have poor response time.

Problem -1 based on SJF scheduling

- Consider the set of 5 processes whose arrival time and burst time are given below;

Process Id	Arrival time	Burst time
P1	3	1
P2	1	4
P3	4	2
P4	0	6
P5	2	3

- If the CPU scheduling policy is SJF **non-preemptive**, calculate the average waiting time and average turn around time.

Solution

- **Gantt Chart-**

Now, we know-

- Turn Around time = Exit time – Arrival time
- Waiting time = Turn Around time – Burst time

Solution

Process Id	Exit time	Turn Around time	Waiting time
P1	7	7 – 3 = 4	4 – 1 = 3
P2	16	16 – 1 = 15	15 – 4 = 11
P3	9	9 – 4 = 5	5 – 2 = 3
P4	6	6 – 0 = 6	6 – 6 = 0
P5	12	12 – 2 = 10	10 – 3 = 7

Now,

- Average Turn Around time = (4 + 15 + 5 + 6 + 10) / 5 = 40 / 5 = 8 unit
- Average waiting time = (3 + 11 + 3 + 0 + 7) / 5 = 24 / 5 = 4.8 unit

Problem -2 based on SJF/SRTF scheduling

- Consider the set of 5 processes whose arrival time and burst time are given below;

Process Id	Arrival time	Burst time
P1	3	1
P2	1	4
P3	4	2
P4	0	6
P5	2	3

- If the CPU scheduling policy is SJF preemptive, calculate the average waiting time and average turn around time.

Solution

- **Gantt Chart-**

Now, we know-

- Turn Around time = Exit time – Arrival time
- Waiting time = Turn Around time – Burst time

Solution

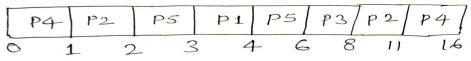
Process Id	Exit time	Turn Around time	Waiting time
P1	4	4 – 3 = 1	1 – 1 = 0
P2	6	6 – 1 = 5	5 – 4 = 1
P3	8	8 – 4 = 4	4 – 2 = 2
P4	16	16 – 0 = 16	16 – 6 = 10
P5	11	11 – 2 = 9	9 – 3 = 6

Now,

- Average Turn Around time = (1 + 5 + 4 + 16 + 9) / 5 = 35 / 5 = 7 unit
- Average waiting time = (0 + 1 + 2 + 10 + 6) / 5 = 19 / 5 = 3.8 unit

Solution 2

Gantt chart 2:



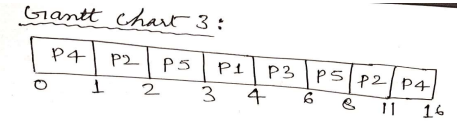
Process Id	CT	TAT	WT
P1	4	1	0
P2	11	10	6
P3	8	4	2
P4	16	16	10
P5	6	4	1

Avg TAT = $\frac{35}{5}$ = 7

Avg WT = $\frac{19}{5}$ = 3.8

Solution 3

Gantt chart 3:



Process Id	CT	TAT	WT
P1	4	1	0
P2	11	10	6
P3	6	2	0
P4	16	16	10
P5	8	6	3

- Avg TAT = $\frac{35}{5}$ = 7
- Avg WT = $\frac{19}{5}$ = 3.8

Problem - 3 based on SRTF scheduling

- Consider the set of 5 processes whose arrival time and burst time are given below;

Process Id	Arrival time	Burst time
P1	0	20
P2	15	25
P3	30	10
P4	45	15

- If the CPU scheduling policy is SRTF, calculate the waiting time of process P2.

Solution

- **Gantt Chart-**

Now, we know-

- Turn Around time = Exit time – Arrival time
- Waiting time = Turn Around time – Burst time

Solution

Now, we know-

- Turn Around time = Exit time – Arrival time
- Waiting time = Turn Around time – Burst time

Thus,

- Turn Around Time of process P2 = 55 – 15 = 40 unit
- Waiting time of process P2 = 40 – 25 = 15 unit

Priority Scheduling

- Out of all the available processes, CPU is assigned to the process having the highest priority. In case of a tie, it is solved by **FCFS Scheduling**.
- In the Shortest Job First scheduling algorithm, the priority of a process is generally the inverse of the CPU burst time, i.e. the larger the burst time the lower is the priority of that process.
- Priority Scheduling can be used in both preemptive and non-preemptive mode.

Advantages of Priority Scheduling

- It considers the priority of the processes and allows the important processes to run first.
- Priority scheduling in preemptive mode is best suited for real time operating system.

Disadvantages of Priority Scheduling

- Processes with lesser priority may starve for CPU.
- There is no idea of response time and waiting time.

Problem - 1 based on Priority scheduling

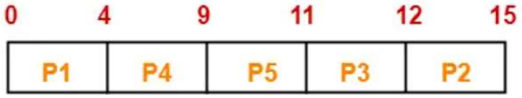
- Consider the set of 5 processes whose arrival time and burst time are given below;

Process Id	Arrival time	Burst time	Priority
P1	0	4	2
P2	1	3	3
P3	2	1	4
P4	3	5	5
P5	4	2	5

- If the CPU scheduling policy is **priority** **non-preemptive**, calculate the average waiting time and average turn around time. (Higher number represents higher priority)

Solution

- **Gantt Chart-**



Now, we know-

- Turn Around time = Exit time – Arrival time
- Waiting time = Turn Around time – Burst time

Solution

Process Id	Exit time	Turn Around time	Waiting time
P1	4	4 – 0 = 4	4 – 4 = 0
P2	15	15 – 1 = 14	14 – 3 = 11
P3	12	12 – 2 = 10	10 – 1 = 9
P4	9	9 – 3 = 6	6 – 5 = 1
P5	11	11 – 4 = 7	7 – 2 = 5

- Average Turn Around time = (4 + 14 + 10 + 6 + 7) / 5 = 41 / 5 = 8.2 unit
- Average waiting time = (0 + 11 + 9 + 1 + 5) / 5 = 26 / 5 = 5.2 unit

Problem - 2 based on Priority scheduling

- Consider the set of 5 processes whose arrival time and burst time are given below;

Process Id	Arrival time	Burst time	Priority
P1	0	4	2
P2	1	3	3
P3	2	1	4
P4	3	5	5
P5	4	2	5

- If the CPU scheduling policy is **priority preemptive**, calculate the average waiting time and average turn around time. (Higher number represents higher priority)

Solution

- **Gantt Chart-**

Now, we know-

- Turn Around time = Exit time – Arrival time
- Waiting time = Turn Around time – Burst time

Solution

Process Id	Exit time	Turn Around time	Waiting time
P1	15	15 – 0 = 15	15 – 4 = 11
P2	12	12 – 1 = 11	11 – 3 = 8
P3	3	3 – 2 = 1	1 – 1 = 0
P4	8	8 – 3 = 5	5 – 5 = 0
P5	10	10 – 4 = 6	6 – 2 = 4

- Average Turn Around time = (15 + 11 + 1 + 5 + 6) / 5 = 38 / 5 = 7.6 unit
- Average waiting time = (11 + 8 + 0 + 0 + 4) / 5 = 23 / 5 = 4.6 unit

Round Robin Scheduling

- CPU is assigned to the process on the basis of FCFS for a fixed amount of time.
- This fixed amount of time is called as **time quantum** or **time slice**. After the time quantum expires, the running process is preempted and sent to the ready queue.
- Then, the processor is assigned to the next arrived process. It is always preemptive in nature.

Advantages of Round Robin Scheduling

- It gives the best performance in terms of average response time.
- It is best suited for time sharing system, client server architecture and interactive system.

Disadvantages of Round Robin Scheduling

- It leads to starvation for processes with larger burst time as they have to repeat the cycle many times.
- Its performance heavily depends on time quantum.
- Priorities can not be set for the processes.

Problem - 1 based on Round Robin Scheduling

- Consider the set of 5 processes whose arrival time and burst time are given below;

Process Id	Arrival time	Burst time
P1	0	5
P2	1	3
P3	2	1
P4	3	2
P5	4	3

- If the CPU scheduling policy is Round Robin with **time quantum = 2 unit,** calculate the average waiting time and average turn around time.

Solution

- **Gantt Chart-**

Now, we know-

- Turn Around time = Exit time – Arrival time
- Waiting time = Turn Around time – Burst time

Solution

Process Id	Exit time	Turn Around time	Waiting time
P1	13	13 – 0 = 13	13 – 5 = 8
P2	12	12 – 1 = 11	11 – 3 = 8
P3	5	5 – 2 = 3	3 – 1 = 2
P4	9	9 – 3 = 6	6 – 2 = 4
P5	14	14 – 4 = 10	10 – 3 = 7

- Average Turn Around time = (13 + 11 + 3 + 6 + 10) / 5 = 43 / 5 = 8.6 unit
- Average waiting time = (8 + 8 + 2 + 4 + 7) / 5 = 29 / 5 = 5.8 unit

Problem - 2 based on Round Robin Scheduling

- Consider the set of 6 processes whose arrival time and burst time are given below;

Process Id	Arrival time	Burst time
P1	5	5
P2	4	6
P3	3	7
P4	1	9
P5	2	2
P6	6	3

- If the CPU scheduling policy is RR **with time quantum = 3,** calculate the average waiting time and average turn around time.

Solution

- **Gantt Chart-**

Now, we know-
- Turn Around time = Exit time – Arrival time
- Waiting time = Turn Around time – Burst time

Solution

Process Id	Exit time	Turn Around time	Waiting time
P1	32	32 – 5 = 27	27 – 5 = 22
P2	27	27 – 4 = 23	23 – 6 = 17
P3	33	33 – 3 = 30	30 – 7 = 23
P4	30	30 – 1 = 29	29 – 9 = 20
P5	6	6 – 2 = 4	4 – 2 = 2
P6	21	21 – 6 = 15	15 – 3 = 12

- Average Turn Around time = (27 + 23 + 30 + 29 + 4 + 15) / 6 = 128 / 6 = 21.33 unit
- Average waiting time = (22 + 17 + 23 + 20 + 2 + 12) / 6 = 96 / 6 = 16 unit

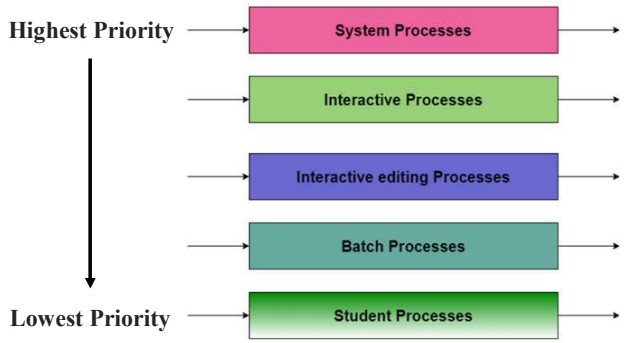
Multilevel Queue Scheduling Algorithm

- A multi-level queue scheduling algorithm divides the ready queue into several separate queues.
- The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm.

MQSA Cont…

- **For example,** separate queues might be used for foreground (interactive) and background (batch) processes.
- The foreground queue might be scheduled by the Round Robin algorithm, while the background queue is scheduled by an FCFS algorithm.

MQSA Cont…



MQSA Cont…

- **System Process**: The Operating system has own process to run as System Process.
- **Interactive Process**: The Interactive Process is a process in which there should be the same kind of interaction (basically an online game).

MQSA Cont…

- **Batch Processes**: Batch processing is basically a technique in the Operating system that collects the programs and data together in the form of the **batch** before the **processing** starts.
- **Student Process**: The system process always gets the highest priority while the student processes always get the lowest priority.

MQSA Cont…

- Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process will be preempted.

MQSA Cont…

- In this case, if there are no processes on the higher priority queue only then the processes on the low priority queues will run. For **Example:** Once processes on the system queue, the Interactive queue, and Interactive editing queue become empty, only then the processes on the batch queue will run.

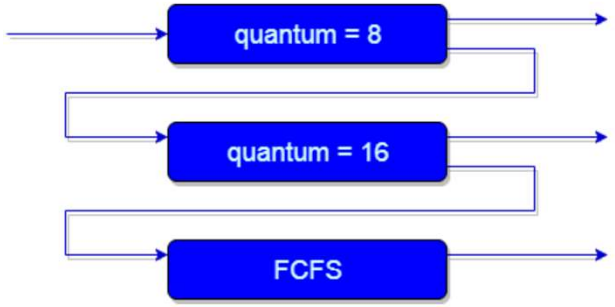
Multilevel Feedback Queue Scheduling

- In a multilevel queue-scheduling algorithm, processes are permanently assigned to a queue in the system.
- Processes do not move between queues.
- It allows a process to move between queues.
- The idea is to separate processes with different CPU-burst characteristics.

Multilevel Feedback Queue Scheduling

- If a process uses too much CPU time, it will be moved to a lower-priority queue.
- Similarly, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue.
- This Algorithm prevents from starvation problem.

MFQS Cont…



MFQS Cont…

- First of all, Suppose that queues 1 and 2 follow round robin with time quantum 8 and 16 respectively and queue 3 follows FCFS. One of the implementations of Multilevel Feedback Queue Scheduling is as follows:

1. If any process starts executing then firstly it enters queue 1.
2. In queue 1, the process executes for 8 unit and if it completes in these 8 units or it gives CPU for I/O operation in these 8 units unit than the priority of this process does not change, and if for some reasons it again comes in the ready queue than it again starts its execution in the Queue 1.
3. If a process that is in queue 1 does not complete in 8 units then its priority gets reduced and it gets shifted to queue 2.
4. Above points 2 and 3 are also true for processes in queue 2 but the time quantum is 16 units. Generally, if any process does not complete in a given time quantum then it gets shifted to the lower priority queue.
5. After that in the last queue, all processes are scheduled in an FCFS manner.
6. It is important to note that a process that is in a lower priority queue can only execute only when the higher priority queues are empty.
7. Any running process in the lower priority queue can be interrupted by a process arriving in the higher priority queue.

Advantages of Multilevel Feedback Queue Scheduling

- This is a flexible Scheduling Algorithm, allows different processes to move between different queues.
- A process that waits long time in the lower priority queue may be moved to the higher priority queue, which helps to preventing starvation problem.

Disadvantages of Multilevel Feedback Queue Scheduling

- This is complex algorithm.
- The processes are moving around different queues which leads more CPU overheads.

Thank You

Operating System

Presented by:
Dr. Premnarayan Arya
Assistant Professor
Department of CEA, GLA University, Mathura

Contents

Types of Process

There are two type of process in the synchronization;

- **Independent Process:** Two processes are said to be independent if the execution of one process does not affect the execution of another process.

Types of Process Cont…

- **Cooperative Process:** Two processes are said to be cooperative if the execution of one process affects the execution of another process. These processes need to be synchronized so that the order of execution can be guaranteed.
- This process share variable, memory, code, resources (CPU, printer, scanner etc.)

Interprocess Communication

- Processes executing concurrently in the operating system may be either independent processes or cooperating processes.
- A process is ***independent*** if it cannot affect or affected by other processes executing in the system. A process that does not share data with any other process is **independent**.
- A process is ***cooperating*** if it can affect or affected by other processes executing in the system. A process that share variable, code, memory, resource with other processes is a cooperating process.

Critical section

- Critical section is a part of the program where shared resources are accessed by the process, such as common variables and files, and perform write operations on them.
- Since, processes execute concurrently, any process can be interrupted mid-execution.
- Critical section is a section of the program where a process access the shared resources during its execution.

Example of Critical section

If multiple processes execute concurrently without any synchronization.

- Two processes P_1 and P_2 are executing concurrently.
- Both the processes share a common variable named "count" having initial value = 5.
- Process P1 tries to increment the value of X.
- Process P2 tries to decrement the value of X.

Example of Critical section cont…

int count = 5

Process 1	**Process 1**
(1) int x=count;	(1) int y=count;
(2) x++;	(2) y--;
(3) sleep(2)	(3) sleep(2)
(4) count = x;	(4) count = y;

Example of Critical section cont…

- Case 1: P1, P2, P1, P2
- **Ans: 4**
- Case 2: P2, P1, P2, P1
- **Ans: 6**

Race Condition

Race condition is a **situation** where;

- The final output produced, it depends on the **execution order of instructions** of different processes. The processes **compete** to each other.

Race Condition

- When more than one process is either running the same code or modifying the same memory or any shared data, there is a **risk** that the result or value of the shared data may be **incorrect** because all processes try to access and modify this shared resource.
- Thus, all the processes race to say that my result is correct. This condition is called the race condition.

Problem based on Process Synchronization

The following two functions **P1** and **P2** that share a variable B with an initial value of **2** execute concurrently;

```
P1( )
{
C = B - 1 ;
B = 2 x C ;
}
```

```
P2( )
{
D = 2 x B ;
B = D - 1 ;
}
```

Solution

Case-1:

- The execution order of the instructions may be-

$P_1(1), P_1(2), P_2(1), P_2(2)$

- In this case,

Final value of B = 3

Solution

Case-2:

- The execution order of the instructions may be-

$P_2(1), P_2(2), P_1(1), P_1(2)$

- In this case,

Final value of B = 4

Solution

Case-3:

- The execution order of the instructions may be-

$P_1(1), P_2(1), P_2(2), P_1(2)$

- In this case,

Final value of B = 2

Solution

Case-4:

- The execution order of the instructions may be-

$P_2(1), P_1(1), P_1(2), P_2(2)$

- In this case,

Final value of B = 3

Solution

Case-5:

- The execution order of the instructions may be-

$P_1(1), P_2(1), P_1(2), P_2(2)$

- In this case,

Final value of B = 3

Solution

Case-6:

- The execution order of the instructions may be-

$P_2(1), P_1(1), P_2(2), P_1(2)$

- In this case,

Final value of B = 2

Critical Section

Critical Section

- Critical section is a **part of the program** where shared resources are accessed by the process.
- A part of code that can only be **accessed** by a single process at a time is known as a critical section.
- Many programs want to access and **change** a single shared data, only one process will be allowed to change at a time.
- The other processes will be **wait** until the data is free to be used.

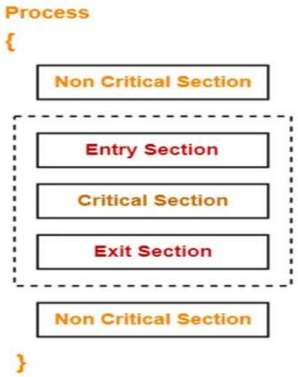
Critical Section Problem

- If multiple processes **access** the critical section concurrently, then results produced might be **inconsistent**.
- This problem is called as **critical section problem**.

Synchronization Mechanisms

- Synchronization mechanisms allow the processes to access critical section in a synchronized manner to avoid the inconsistent results.
- For every critical section in the program, a synchronization mechanism adds;
  - An entry section before the critical section
  - An exit section after the critical section

Synchronization Mechanisms



Entry Section

- It **acts** as a **gateway** for a process to enter inside the critical section.
- It ensures that **only one process is present** inside the critical section at any time.
- It **does not allow any other process** to enter inside the critical section if one process is already present inside it.

Exit Section

- It acts as an **exit gate** for a process to leave the critical section.
- When a process takes exit from the critical section, some **changes are made** so that other processes can enter inside the critical section.

Criteria For Synchronization Mechanisms

Any synchronization mechanism proposed to handle the critical section problem should meet the following criteria;

1. Mutual Exclusion
2. Progress
3. Bounded Wait
4. No assumption related to H/w and speed

1. Mutual Exclusion

- The processes access the critical section in a **mutual exclusive manner.**
- Only **one process** can present inside the critical section at a time.
- No other process can enter in the critical section until the process already present inside it completes.

2. Progress

- An entry of a process inside the critical section is not dependent on the entry of another process inside the critical section.
- A process can **freely** enter inside the critical section if there is no other process present inside it.
- A process enters the critical section only if it wants to enter.
- A process is not forced to enter inside the critical section if it does not want to enter.

3. Bounded Wait

- The wait of a process to enter the critical section is bounded.
- A process can enter in critical section before its wait over.

Critical Section

There are four rules for synchronization mechanism:

1. Mutual Exclusion
2. Progress
3. Bounded wait
4. No assumption related to H/w and speed

Critical Section

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (TRUE);
```

Solutions for the Critical Section Problem

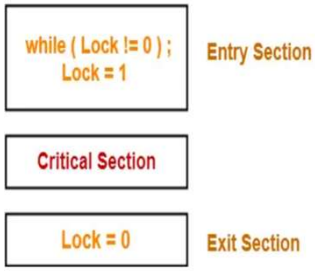
Lock Variable

Lock Variable

- Lock variable is a **synchronization** mechanism.
- It uses a lock variable to provide the synchronization among the processes executing **concurrently**.
- It can be used for any number of processes.
- However, it completely **fails** to provide the synchronization.

Lock Variable Cont…

- Initially, lock value is set to 0.
- Lock value = 0 means the critical section is **currently vacant** and no process present inside CS.
- Lock value = 1 means the critical section is **currently occupied** and a process is present inside CS.



Case - 1

- Process P_0 arrives.
- It executes the lock!=0 (lock= = 1) instruction.
- Since lock value is set to 0, so it returns value 0 to the while loop.
- The while loop condition breaks.
- It sets the lock value to 1 and enters the critical section.
- Now, even if process P_0 gets preempted in the middle, no other process can enter the critical section.
- Any other process can enter only after process P_0 completes and sets the lock value to 0.

Case - 2

- Another process P_1 arrives.
- It executes the lock!=0 (lock= = 1) instruction.
- Since lock value is set to 1, so it returns value 1 to the while loop.
- The returned value 1 does not break the while loop condition.
- The process P_1 is trapped inside an infinite while loop.
- The while loop keeps the process P_1 busy until the lock value becomes 0 and its condition breaks.

Case - 3

- Process P_0 arrives.
- It executes the lock!=0 (lock= = 1) instruction.
- Since lock value is set to 0, so it returns value 0 to the while loop.
- The while loop condition breaks.
- Now, process P_0 gets preempted before it sets the lock value to 1.

Case – 3 Cont…

- Another process P_1 arrives.
- It executes the lock!=0 (lock= = 1) instruction.
- Since lock value is still 0, so it returns value 0 to the while loop.
- The while loop condition breaks.
- It sets the lock value to 1 and enters the critical section.
- Now, process P_1 gets preempted in the middle of the critical section.

Case – 3 Cont…

- Process P_0 gets scheduled again.
- It resumes its execution.
- Before preemption, it had already failed the while loop condition.
- Now, it begins execution from the next instruction.
- It sets the lock value to 1 (which is already 1) and enters the critical section.
- Thus, both the processes get to present inside the critical section at the same time.

Drawback of Lock Variable

- This mechanism completely fails to provide the synchronization among the processes.
- No mutual exclusion guarantee.
- If there are n processes, then all of them may be present inside the critical section at the same time.
- This happens when each process gets preempted immediately after breaking the while loop condition.

Test and Set Lock

Test and Set Lock

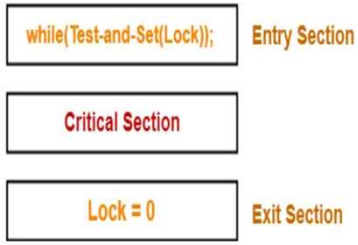
- It is an instruction that returns the old value of a memory location and sets the memory location value to 1 as a single atomic operation.
- If one process is currently executing a test-and-set, no other process is allowed to begin another test-and-set until the first process test-and-set is finished.

Test and Set Lock Cont…

- Test and Set Lock (TSL) is a synchronization mechanism.
- It uses a test and set instruction to provide the synchronization among the processes executing concurrently.

Test and Set Lock Cont...

- Lock value = 0 means the critical section is currently **vacant** and no process is present inside CS.
- Lock value = 1 means the critical section is currently **occupied** and a process is present inside CS.



Test and Set Lock Cont...

```
while (test and set(&lock));
critical section
lock = false;

boolean test and set(boolean *target)
{
boolean r = *target;
*target = true;
return r;
}
```

Case - 1

- Process P_0 arrives.
- It executes the test-and-set(Lock) instruction.
- Since lock value is set to 0, so it returns value 0 to the while loop and sets the lock value to 1.
- The returned value 0 breaks the while loop condition.
- Process P_0 enters the critical section and executes.
- Now, even if process P_0 gets preempted in the middle, no other process can enter the critical section.
- Any other process can enter only after process P_0 completes and sets the lock value to 0.

Case - 2

- Another process P_1 arrives.
- It executes the test-and-set(Lock) instruction.
- Since lock value is now 1, so it returns value 1 to the while loop and sets the lock value to 1.
- The returned value 1 does not break the while loop condition.
- The process P_1 is trapped inside an infinite while loop.
- The while loop keeps the process P_1 busy until the lock value becomes 0 and its condition breaks.

Case - 3

- Process P_0 comes out of the critical section and sets the lock value to 0.
- The while loop condition breaks.
- Now, process P_1 waiting for the critical section enters the critical section.
- Now, even if process P_1 gets preempted in the middle, no other process can enter the critical section.
- Any other process can enter only after process P_1 completes and sets the lock value to 0.

Characteristics Test and Set Lock

- It ensures mutual exclusion.
- It is deadlock free.
- It does not guarantee bounded waiting and may cause starvation.
- It suffers from spin lock.
- It is a busy waiting solution which keeps the CPU busy when the process is actually waiting.

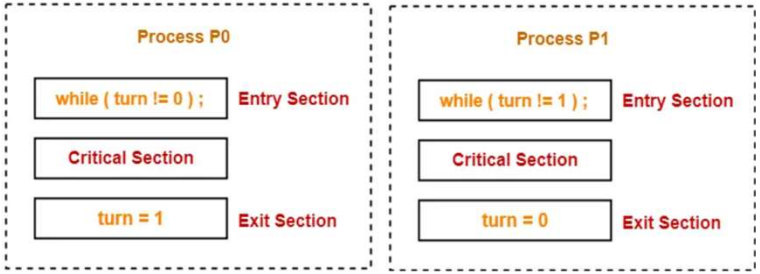
Turn Variable

Turn Variable

- Turn variable provides synchronization between **only two processes.**
- It is also known as **strict alteration**.
- Processes have to **compulsorily** enter the critical section **alternately** whether they want or not.
- This is because if one process does not enter the critical section, then other process will never get a **chance** to execute again.

Turn Variable

- Turn value = 0 means it is the turn of process P_0 to enter the critical section.
- Turn value = 1 means it is the turn of process P_1 to enter the critical section.



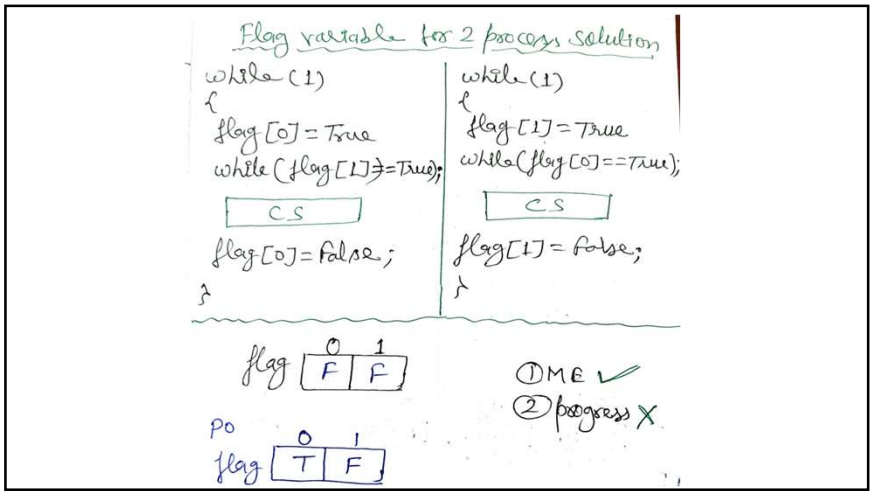
Case - 1

- Process P_0 arrives.
- It executes the turn!=0 instruction.
- Since turn value is set to 0, so it returns value 0 to the while loop.
- The while loop condition breaks.
- Process P_0 enters the critical section and executes.
- Now, even if process P_0 gets preempted in the middle, process P_1 can not enter the critical section.
- Process P_1 can not enter unless process P_0 completes and sets the turn value to 1.

Case - 2

- Process P_1 arrives.
- It executes the turn!=1 instruction.
- Since turn value is set to 0, so it returns value 1 to the while loop.
- The returned value 1 does not break the while loop condition.
- The process P_1 is trapped inside an infinite while loop.
- The while loop keeps the process P_1 busy until the turn value becomes 1 and its condition breaks.

Flag Variable

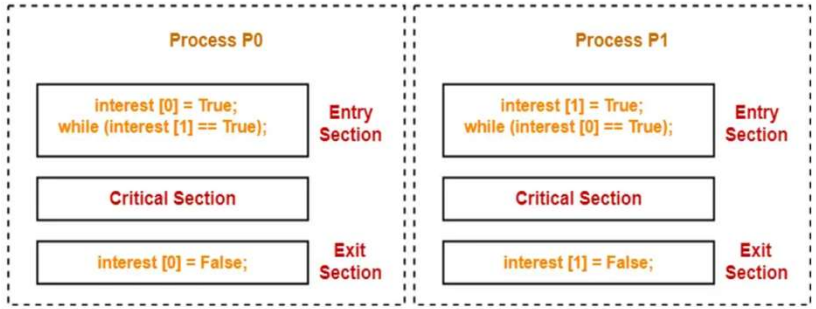


Interest Variable

Interest Variable

- Interest variable is a synchronization mechanism that provides synchronization among two processes.
- It uses an interest variable to provide the synchronization.

Interest Variable



Interest Variable

- Initially, interest [0] and interest [1] are set to False.
- Interest value [0] = False means that process P_0 is not interested to enter the critical section.
- Interest value [0] = True means that process P_0 is interested to enter the critical section.
- Interest value [1] = False means that process P_1 is not interested to enter the critical section.
- Interest value [1] = True means that process P_1 is interested to enter the critical section.

Scene-1

- Process P_0 arrives.
- It sets interest[0] = True.
- Now, it executes while loop condition- interest [1] == True.
- Since interest [1] is initialized to False, so it returns value 0 to the while loop.
- The while loop condition breaks.
- Process P_0 enters the critical section and executes.
- Now, even if process P_0 gets preempted in the middle, process P_1 can not enter the critical section.
- Process P_1 can not enter unless process P_0 completes and sets the interest [0] = False.

Scene-2

- Process P_1 arrives.
- It sets interest[1] = True.
- Now, it executes while loop condition- interest [0] == True.
- Process P_0 has already shown its interest by setting interest [0] = True.
- So, it returns value 1 to the while loop.
- The while loop condition satisfies.
- The process P_1 is trapped inside an infinite while loop.
- The while loop keeps the process P_1 busy until the interest [0] value becomes False and its condition breaks.

Peterson's Solutions

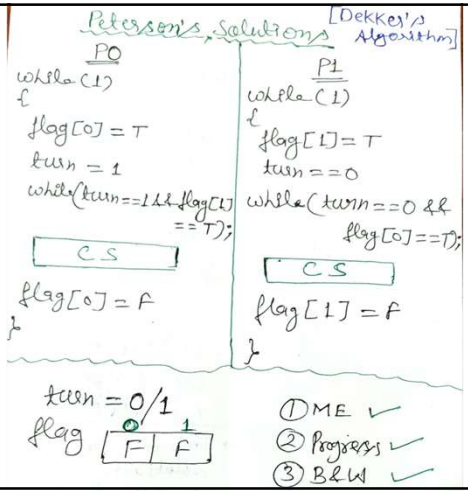
Peterson's Solutions

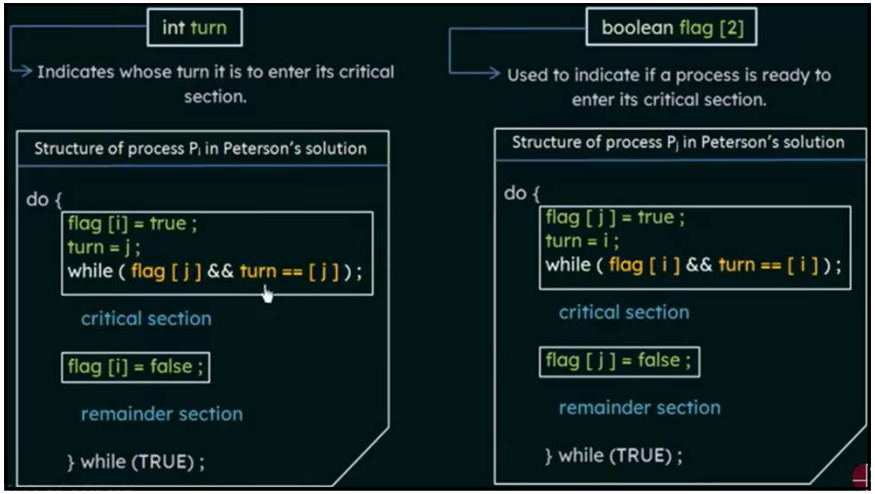
- Peterson's Solution is a classical **software-based solution** of the critical section problem.
- Peterson's solution was developed by a computer scientist **Peterson**.

Peterson's Solutions Cont…

In Peterson's solution, we have two shared variables:

1. **boolean flag[0]:** Initialized to FALSE, initially no process inside the critical section.
2. **int turn:** The process whose turn is to enter the critical section.





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

1. Mutual exclusion:

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

2. Progress:

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

3. Bounded waiting:

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

Drawback of Peterson's solution

- It involves busy waiting.(In the Peterson's solution, the code statement- "while(flag[j] && turn == j);" is responsible for this. Busy waiting is not favored because it wastes CPU cycles that could be used to perform other tasks.)
- It is **limited to 2 processes**.
- Peterson's solution cannot be used in modern CPU architectures.

Semaphore

Semaphore

- Semaphore is an **integer variable** that used to solve the critical section problem by using two atomic operations, **wait** and **signal** that are used for **process synchronization**.
- Semaphore is used in mutual exclusive manner by various concurrent **cooperative processes** in order to achieve **synchronization**.

Semaphore Cont…

- **Wait/down:** The wait/down operation **decrements** the value of its argument **S**, if it is positive.
- If **S** is negative or zero, then no operation would be performed.

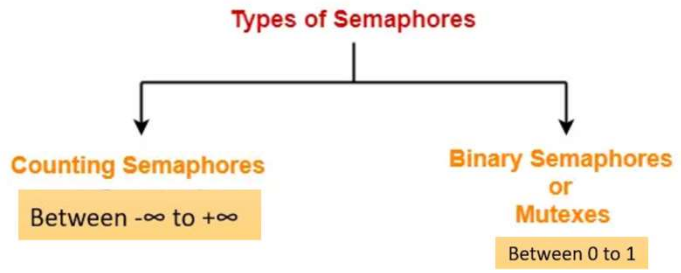
```
wait(S)
{
   while (S<=0);

   S--;
}
```

Semaphore Cont…

- **Signal/up:** The signal/up operation **increments** the value of its argument **S**.

```
signal(S)
{
   S++;
}
```

Types of Semaphore



Counting Semaphores

- Counting semaphores have integer value, to manage a pool of resources that can be accessed by multiple processes.
- It used to **coordinate the resource access**, where the semaphore count is the number of available resources.
- If the **resources** are added, semaphore count automatically incremented and if the resources are removed, the count is decremented.

Entry Code

```
Down (Semaphore S)
{
   S.value = S.value – 1;
   if (S.value < 0)
    {
      Put process (PCB) in
      suspended list, sleep( );
    }
      else
           return;
 }
```

Exit Code

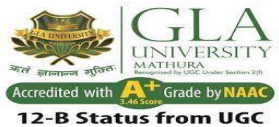
```
Up (Semaphore S)
{
   S.value = S.value + 1;
   if (S.value ≤ 0)
    {
      Select a process from
       suspended list, and
      wake up( );
    }
 }
```

Question

1. If current value of semaphore is 10, then 6 P operation and 4 V operations are performed. What will be the final value of Semaphore?
2. If current value semaphore is 17, then 5 P operation, 3 V operation and 1 P operations are performed. What will be the final value of Semaphore?

Thank You

Operating System



Presented by:
Dr. Premnarayan Arya
Assistant Professor
Department of CEA, GLA University, Mathura

Producer-Consumer Problem

Producer-Consumer Problem

- The **Producer-Consumer problem** is a classic synchronization problem in operating systems.
- There is a **fixed-size buffer** and a Producer process, and a Consumer process.
- The **Producer** process creates an item and adds it to the shared buffer. The **Consumer** process takes items out of the shared buffer and "consumes" them.

Producer-Consumer Problem

Some conditions must be followed by the Producer and the Consumer processes to have consistent data synchronization:

1. Producer Process should not produce any data when the shared buffer is full.
2. Consumer Process should not consume any data when the shared buffer is empty.
3. The access to the shared buffer should be mutually exclusive i.e at a time only one process and make changes to it.

Producer Code

```
int count = 0;
void producer( void )
{
 int itemP;
 while(1)
 {
  Produce_item(item P)
  while( count == n);
  buffer[ in ] = item P;
  in = (in + 1)mod n
  count = count + 1;
 }
}
```

Load Rp, m[count]
Increment Rp
Store m[count], Rp

Consumer Code

```
int count;
void consumer(void)
{
 int itemC;
 while( 1 )
 {
  while( count == 0 );
  itemC = buffer[ out ];
  out = ( out + 1 ) mod n;
  count = count -1;
 }
}
```

Load Rc, m[count]
Decrement Rc
Store m[countl. Rc

Understand Producer Code

Explanation of code, first, understand the few terms used in the above code:

1. "**in**" used in a producer code represent the next **empty buffer**
2. "**out**" used in consumer code represent first **filled buffer**
3. count keeps the count number of elements in the buffer
4. count is further divided into 3 lines code represented in the **block** in both the producer and consumer code.

Cont…

Producer code block:

- Rp is a register which keeps the value of m[count]
- Rp is incremented (As element has been added to buffer)
- An Incremented value of Rp is stored back to m[count]

Consumer code block:

- Rc is a register which keeps the value of m[count]
- Rc is decremented (As element has been removed out of buffer)
- The decremented value of Rc is stored back to m[count].

Buffer

```
  (pointer to the next empty space)→in    = 5
(pointer to the first filled space)→out   = 0
     (number of elements in Buffer)→count = 5
                   (Size of Buffer)→n     = 8
```

0	A
1	B
2	C
3	D
4	E
5	
6	
7	

- Buffer has total 8 spaces out of which the first 5 are filled, in = 5 (pointing next empty position) and out = 0 (pointing first filled position).
- Let's start with **the producer** who wanted to produce an element "F", according to code it will enter into the producer() function, while(1) will always be true, itemP = F will be tried to insert into the buffer, before that while(count == n); will evaluate to be False.

Cont…

0	A
1	B
2	C
3	D
4	E
5	F
6	
7	

- Buffer[in] = itemP → Buffer[5] = F. (F is inserted now)
- in = (in + 1) mod n → (5 + 1)mod 8→ 6, therefore in = 6; (next empty buffer)
- After insertion of F, Buffer looks like this
- Where **out = 0**, but **in = 6**

Since count = count + 1; is divided into three parts:

- Load Rp, m[count] → will copy count value which is 5 to register Rp.
- Increment Rp → will increment Rp to 6.
- Suppose just after Increment and before the execution of third line (store m[count], Rp) **Context Switch** occurs and code jumps to **consumer code**

Understand Consumer Code

- Now starting **consumer** who wanted to consume the first element "A", according to code it will enter into the consumer() function, while(1) will always be true, while(count == 0); will evaluate to be False (since the count is still 5, which is not equal to 0.
- itemC = Buffer[out]→ itemC = A (since out is 0)
- out = (out + 1) mod n → (0 + 1)mod 8→ 1, therefore out = 1(first filled position)
- A is removed now
- After removal of A, Buffer look like this
- Where **out = 1**, and **in = 6**

0	
1	B
2	C
3	D
4	E
5	F
6	
7	

Cont…

- Since count = count - 1; is divided into three parts:
- Load Rc, m[count] → will copy count value which is 5 to register Rp.
- Decrement Rc → will decrement Rc to 4.
- store m[count], Rc → count = 4.
- **Now the current value of count is 4**
- Suppose after this **Context Switch** occurs back to the leftover part of producer code. . .
- Since context switch at producer code was occurred after Increment and before the execution of the third line (store m[count], Rp)
- So we resume from here since Rp holds 6 as incremented value
- Hence store m[count], Rp → count = 6
- **Now the current value of count is 6, which is wrong as Buffer has only 5 elements, this condition is known as Race Condition and Problem is Producer-Consumer Problem.**

Producer-Consumer Problem

Solution

Producer Code

```
int count = 0;
void producer( void )
{
 int itemP;
 while(1)
 {
  Produce_item(item P)
  while( count == n);
  buffer[ in ] = item P;
  in = (in + 1)mod n
  count = count + 1;
 }
}
```

→

```
Load Rp, m[count]
Increment Rp
Store m[count], Rp
```

Consumer Code

```
int count;
void consumer(void)
{
 int itemC;
 while( 1 )
 {
  while( count == 0 );
  itemC = buffer[ out ];
  out = ( out + 1 ) mod n;
  count = count -1;
 }
}
```

→

```
Load Rc, m[count]
Decrement Rc
Store m[countl. Rc
```

Counting Semaphore - full = 0 (no of filled slots)
empty = N (no of empty slots)
Binary Semaphore S = 1

Producer_item (**item p**)
down (empty);
down (S);

buffer[in] = item p;
in = (in + 1)mod n;

up (S);
up (full);

Buffer N = 8

0	A
1	B
2	C
3	
4	
5	
6	
7	

Empty = 5
Full = 3
S = 1

Consumer
down (full);
down (S);

Item C = buffer [out];
out = (out + 1)mod n;

up (S);
up (empty);

Readers-Writers Problem

Readers-Writers Problem

- The readers-writers problem is a classic synchronization problem, where multiple processes require access to a shared resource.
- There are two types of processes **reader** and **writer**. Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource.

Readers-Writers Problem Cont…

- When a **writer** is writing data to the resource, no other process can access the resource.

Cases of Reader-Writer Problem

Case 1: Two processes cannot be allowed to **write** into shared data parallelly thus they must wait to get access to write into it.

Case 2: Even if one process is writing on data and the other is reading then also they cannot be allowed to have the access to shared resources for the reason that a reader will be reading an incomplete value in this case.

Cases of Reader-Writer Problem Cont…

Case 3: The other scenario where one process is reading from the data and another writing on the shared resource, it cannot be allowed. Because the writer updates some data that is not available to the reader.

Case 4: In this case that both processes are **reading** the data then sharing of resources among both the processes will be **allowed.**

Readers-Writers Problem

Cases	Process 1	Process 2	Allowed / Not Allowed
Case 1	Reading	Writing	Problem
Case 2	Writing	Reading	Problem
Case 3	Writing	Writing	Problem
Case 4	Reading	Reading	No Problem

Readers-Writers Problem Solution

Readers-Writers Problem Solution

- One possible solution to the reader-writer problem is to use a **mutex lock** and a **semaphore**.
- The mutex lock ensures mutual exclusion while updating a variable that keeps track of the number of processes performing the read operation.
- The semaphore ensures that no writer can access the critical section while a reader is accessing it.

Readers-Writers Problem Solution Cont…

- The solution works by allowing multiple reader processes to access the critical section simultaneously, but only one writer process can access it at a time.
- The writer process waits for the semaphore to become available and then writes to the resource while all reader processes wait.

Readers-Writers Problem Solution Cont…

- After the writer process releases the semaphore, all reader processes can access the resource simultaneously.

Reader's Code

```
int rc = 0
semaphore mutex=1;
semaphore db=1;
void Reader(void)
{
 while(true)
 {
 down(mutex);
 rc = rc+1;
 if (rc= =1) then down (db);
 up(mutex);
      DB
down(mutex)
rc=rc-1;
if(rc==0) then up(db);
up(mutex)
process data
}
}
```

Writer's Code

```
void write (void)
{
  while (true)
  {
   down (db);
      DB
   up (db);
  }
}
```

The Dining Philosophers Problem

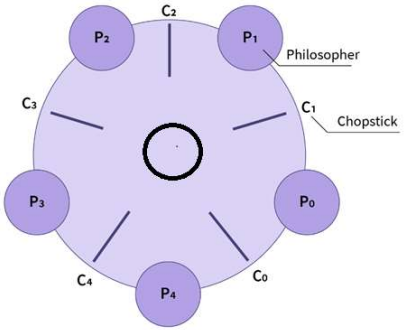
The Dining Philosophers Problem

- The dining philosopher's problem is the **classical problem** of synchronization.
- There are **five philosophers sharing** a **circular table** and their job is to **think and eat** alternatively.
- There is a **bowl of noodles** and **five chopsticks** for each philosophers.
- A philosopher needs both their **left and right chopstick** to eat.

The DPP Cont…

- Each philosopher will pick up **first left side chopsticks** then **right side chopsticks**.
- A hungry philosopher may only eat if there are **both chopsticks** available.
- Otherwise a philosopher **puts down** their chopstick and begin thinking again.

The DPP Cont…




```
Void Philosopher
 {
  while(1)
  {
    take_chopstick[i];
    take_chopstick[ (i+1) % 5] ;
    EATING THE NOODLE
    put_chopstick[i] );
    put_chopstick[ (i+1) % 5] ;
    THINKING…
  }
}
```

The DPP Cont…

- Suppose Philosopher P0 wants to eat, it will enter in Philosopher() function, and execute **take_chopstick[i];** by doing this it holds **C0 chopstick** after that it execute **take_chopstick[(i+1) % 5];** by doing this it holds **C1 chopstick**(since i =0, therefore (0 + 1) % 5 = 1).
- Similarly suppose now Philosopher P1 wants to eat, it will enter in Philosopher() function, and execute **take_chopstick[i];** by doing this it holds **C1 chopstick** after that it execute **take_chopstick[(i+1) % 5];** by doing this it holds **C2 chopstick**(since i =1, therefore (1 + 1) % 5 = 2).

The DPP Cont…

- But Practically Chopstick C1 is not available as it has already been taken by philosopher P0, hence the above code generates problems and produces race condition.

The Dining Philosophers Problem Solution

Dining Philosopher Problem Solution Code using semaphore operations wait and signal

```
Void Philosopher
 {
  while(1)
  {
   Wait (take_chopstick[i]);
   Wait (take_chopstick[ (i+1) % 5]);
   EATING THE NOODLE
   Singnal(put_chopstick[i]);
   Signal(put_chopstick[ (i+1) % 5]);
   THINKING…
  }
}
```

The DPPS Cont…

- Let value of i = 0 (initial value), Suppose Philosopher P0 wants to eat, it will enter in Philosopher() function, and execute **Wait(take_chopstick[i]);** by doing this it holds **C0 chopstick** and reduces semaphore C0 to 0**,** after that it execute **Wait(take_chopstick[(i+1) % 5]);** by doing this it holds **C1 chopstick**(since i =0, therefore (0 + 1) % 5 = 1) and reduces semaphore C1 to 0.

The DPPS Cont…

- Similarly, suppose now Philosopher P1 wants to eat, it will enter in Philosopher() function, and execute **Wait(take_chopstick[i]);** by doing this it will try to hold **C1 chopstick** but will not be able to do that**,** since the value of semaphore C1 has already been set to 0 by philosopher P0, therefore it will enter into an infinite loop because of which philosopher P1 will not be able to pick chopstick C1 whereas if Philosopher P2 wants to eat, it will enter in Philosopher() function, and execute **Wait(take_chopstick[i]);** by doing this it holds **C2 chopstick** and reduces semaphore C2 to 0, after that, it executes **Wait(take_chopstick[(i+1) % 5]);** by doing this it holds **C3 chopstick**(since i =2, therefore (2 + 1) % 5 = 3) and reduces semaphore C3 to 0.

The DPPS Cont…

- Hence the above code is providing a solution to the dining philosopher problem, A philosopher can only eat if both immediate left and right chopsticks of the philosopher are available else philosopher needs to wait. Also at one go two independent philosophers can eat simultaneously (i.e., philosopher **P0 and P2, P1 and P3 & P2 and P4** can eat simultaneously as all are the independent processes and they are following the above constraint of dining philosopher problem)

Thank You

Multithreaded System

Presented by:
Dr. Premnarayan Arya
Assistant Professor
Department of CEA, GLA University, Mathura

Contents

Thread

- A thread is a basic unit of CPU utilization, consisting of a program counter, stack, and a set of registers, (and a thread ID.)
- A thread is a lightweight unit of execution within a process. A process is an instance of a program that is being executed, and a thread is a subset of the process that can run concurrently with other threads within the same process.

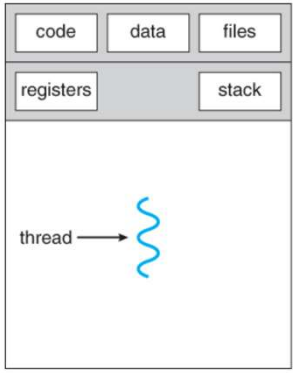
Thread

- Threads share resources with other threads in the same process, such as memory, file handles, and network connections, which makes them more efficient than processes. Because threads are lighter weight than processes, they can be created and destroyed more quickly, and they can switch between tasks more rapidly.

Components of A Thread

- Thread ID
- Program counter
- Stack
- Register set
- Thread priority
- Thread state
- Thread-safety
- Synchronization

Thread Diagram

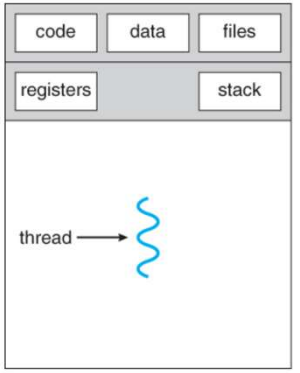


Difference Between Process and Thread

- The **threads** within the same process run in a **shared memory** space, while **processes** run in **separate memory** spaces.
- **Threads** are not **independent** of one another like **processes**, threads share other threads **code, data, files**, and **resources** (like open files and signals).
- Like a process, a thread has its own program counter (PC), register, and stack.

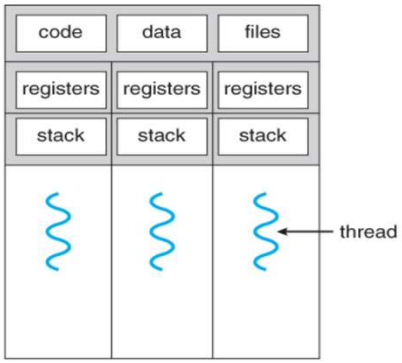
Single Threaded Process

- A process have a single thread of control, there is one program counter, and one sequence of instructions that can be carried out any given time.



Multi-Threaded Process

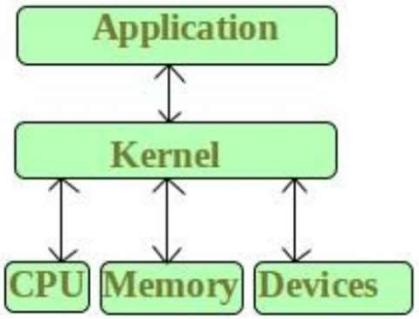
- Multi-threaded applications have multiple threads within a single process, each having own program counter, stack and set of registers, but sharing common code, data, and certain structures such as open files.



What is Kernel?

- Kernel is the core part of an operating system that **manages** system resources.
- It also acts as a **bridge** between the applications and hardware of the computer.
- The first programs **loaded** on start-up (after the Bootloader).
- The Kernel is also responsible for offering **secure** access to the machine's hardware for various programs.
- It also **decides** when and how long a certain application uses specific hardware.

Diagram of Kernel



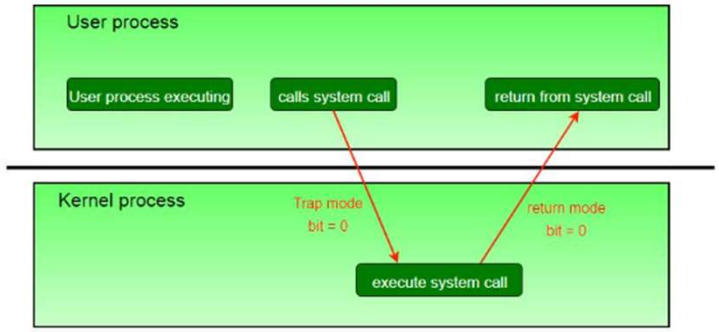
Kernel Mode

- The CPU can **execute certain instructions** only when it is in kernel mode.
- The operating system **puts the CPU** in kernel mode when it is executing in the kernel so, that kernel can execute some special operation.
- For example, instruction for **managing memory protection**.

User Mode

- The operating system puts the CPU in user mode when a **user program is in execution** so, that the user program cannot interface with the operating system program.
- User-level instruction does not require special **privilege**.
- For Example are ADD, PUSH, POP, DELET E etc.

Diagram of User Mode & Kernel Mode



System Calls

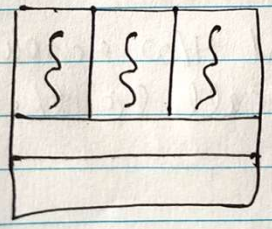
If the user is using any application or API or writing any program then we generally writes it in user mode, But if I want to access any functionality of O.S. then I have to go in Kernel mode. So because our access is on the user mode bcoz we are accessing the user, so to go in Kernel mode we use system call.

→ File related ⇒ open(), read(), write(), close() create file etc.
→ Device related ⇒ Read, write, reposition, IOCTL, FCNTL.
→ Information – get pid, attributes, get system time and data
→ Process control – load, execute, abort, fork, wait signal allocate etc.
→ Communication – Pipe(), create/delete connections, shmget().

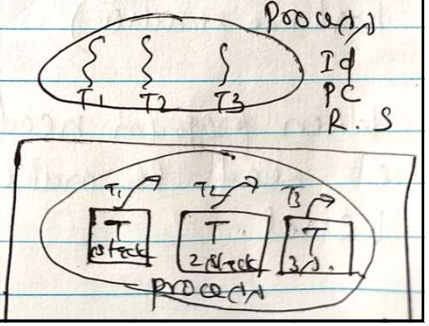
• Difference b/w user level threads & kernel level thread :-

ULT	KLT
① ULT are managed user level library	① KLT are managed by O.S. (use sys. call)
② ULT are typically fast	② KLT are slower than ULT.
③ context switching is faster	③ CS is slower
	Process CS > KLT > ULT

④ If one ULT perform blocking operation then entire process get blocked.



④ If one kernel LT blocked, No effect on others.



Thank You

Operating System

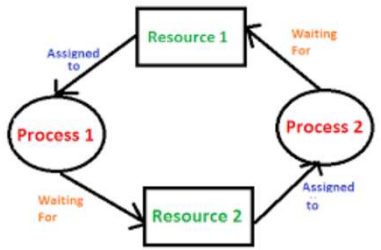
Presented by:
Dr. Premnarayan Arya
Assistant Professor
Department of CEA, GLA University, Mathura

Contents

Deadlock

Deadlock

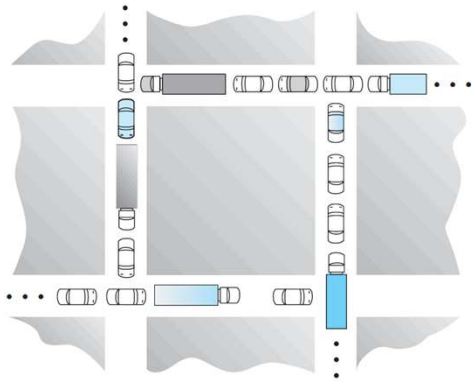
- A **deadlock** is a situation, when two or more processes need some **resource** to complete their execution that is **held** by the other process.



Deadlock

- In the above diagram, the process 1 has resource 1 and needs to acquire resource 2.
- Similarly process 2 has resource 2 and needs to acquire resource 1.
- Process 1 and process 2 are in deadlock as each of them needs the other resource to complete their execution but neither of them is ready to hand over their resources.

Traffic Deadlock



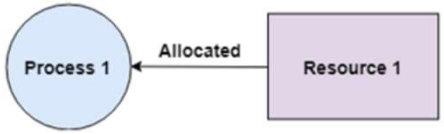
Necessary conditions for Deadlocks OR Coffman Conditions

Necessary conditions for Deadlocks

1. Mutual exclusion
2. No preemption
3. Hold & wait
4. Circular wait

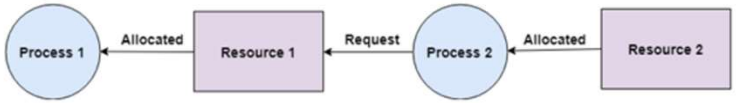
Coffman Conditions

1. **Mutual Exclusion:** At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.



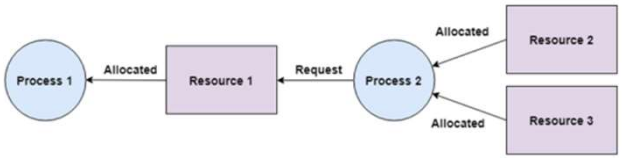
Coffman Conditions Cont…

2. **No Preemption:** Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task. For example, P2 cannot preempt R1 from P1. It will only be released when P1 hand over it voluntarily after its execution is complete.



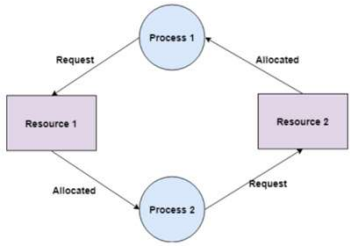
Coffman Conditions Cont…

3. **Hold and Wait:** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes. For example, P2 holds R2 and R3 and is requesting the R1 which is held by P1.



Coffman Conditions Cont…

4. **Circular Wait:** All the processes must be waiting for the resources in a cyclic manner so that the last process is waiting for the resource which is being held by the first process. For example, P1 is allocated R2 and it is requesting R1. Similarly, P2 is allocated R1 and it is requesting R2.



Resource-Allocation Graph

Resource-Allocation Graph

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

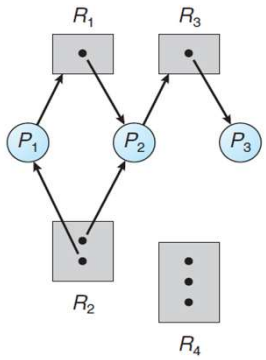
RAG Cont…

- A directed edge from process *Pi* to resource type *Rj* is denoted by ***Pi→Rj***; it signifies that process *Pi* has requested an instance of resource type *Rj* and is currently waiting for that resource.
- A directed edge from resource type *Rj* to process *Pi* is denoted by ***Rj→Pi***; it signifies that an instance of resource type *Rj* has been allocated to process *Pi* . A directed edge ***Pi→Rj*** is called a **request edge**; a directed edge ***Rj→Pi*** is called an **assignment edge**.

RAG Cont…

The sets *P, R,* and *E*:

- *P* = {*P*1, *P*2, *P*3}
- *R* = {*R*1, *R*2, *R*3, *R*4}
- *E* = {*P*1→*R*1, *P*2→*R*3, *R*1→*P*2, *R*2→*P*2, *R*2→*P*1, *R*3→*P*3}



RAG Cont…

Resource instances:

- One instance of resource type $R1$
- Two instances of resource type $R2$
- One instance of resource type $R3$
- Three instances of resource type $R4$

RAG Cont…

Process states:

- Process $P1$ is holding an instance of resource type $R2$ and is waiting for an instance of resource type $R1$.
- Process $P2$ is holding an instance of $R1$ and an instance of $R2$ and is waiting for an instance of $R3$.
- Process $P3$ is holding an instance of $R3$.

Resource-Allocation Graph with Deadlock

Resource-Allocation Graph with Deadlock

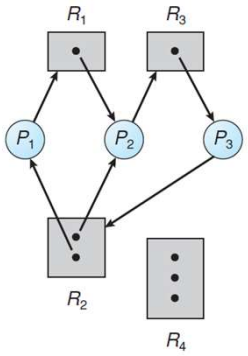
- Suppose that process $P3$ requests an instance of resource type $R2$. Since no resource instance is currently available, we add a request edge $P3 \rightarrow R2$ to the graph. At this point, two minimal cycles exist in the system:

$$P1 \rightarrow R1 \rightarrow P2 \rightarrow R3 \rightarrow P3 \rightarrow R2 \rightarrow P1$$

$$P2 \rightarrow R3 \rightarrow P3 \rightarrow R2 \rightarrow P2$$

- Processes $P1$, $P2$, and $P3$ are deadlocked. Process $P2$ is waiting for the resource $R3$, which is held by process $P3$. Process $P3$ is waiting for either process $P1$ or process $P2$ to release resource $R2$. In addition, process $P1$ is waiting for process $P2$ to release resource $R1$.

Resource-Allocation Graph with Deadlock



$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

Resource-Allocation Graph with a cycle but no deadlock

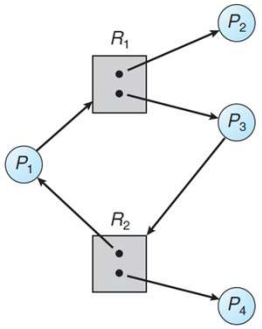
Resource-Allocation Graph with a cycle but no deadlock

- In this example, there is a cycle:

  $P1 \rightarrow R1 \rightarrow P3 \rightarrow R2 \rightarrow P1$

- However, there is no deadlock. Because, the process $P4$ may release its instance of resource type $R2$. That resource can then be allocated to $P3$, breaking the cycle.
- **If a RAG does not have a cycle, then the system is *not* in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state.**

Resource-Allocation Graph with a cycle but no deadlock



$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

Methods for Handling Deadlocks

Methods for Handling Deadlocks

- Deadlock ignorance (Ostrich method)
- Deadlock prevention
- Deadlock Avoidance (Banker's Algorithm)
- Deadlock Detection & Recovery

Deadlock ignorance (Ostrich method)

- **Simplicity:** Ignoring the possibility of deadlock can make the design and implementation of the operating system simpler and less complex.
- **Performance:** Avoiding deadlock detection and recovery mechanisms can improve the performance of the system, as these mechanisms can consume significant system resources.

Deadlock Prevention

- The possibility of deadlock is excluded before making requests, by eliminating one of the necessary conditions for deadlock.
- **Example:** Only allowing traffic from one direction, will exclude the possibility of blocking the road.

Deadlock Avoidance (Banker's Algorithm)

- The OS runs an algorithm on requests to check for a safe state. Any request that may result in a deadlock is not granted.
- **Example:** Checking each car and not allowing any car that can block the road. If there is already traffic on the road, then a car coming from the opposite direction can cause blockage.

Deadlock Detection & Recovery

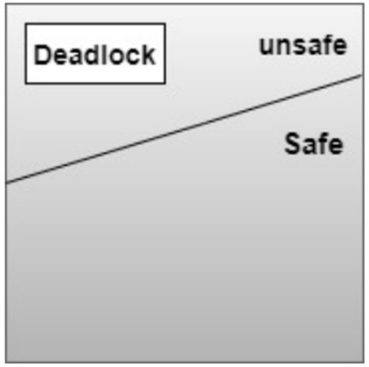
- OS detects deadlock by regularly checking the system state, and recovers to a safe state using recovery techniques.
- **Example:** Unblocking the road by backing cars from one side. Deadlock prevention and deadlock avoidance are carried out before deadlock occurs.

Banker's Algorithm

Safe State and Unsafe State

- A state is **safe** if the system can allocate resources to each process (up to its maximum requirement) in some order and still avoid a deadlock.
- Formally, a system is in a safe state only, if there exists a **safe sequence**.
- **So a safe state is not a deadlocked state** and **a deadlocked state is an unsafe state.**

Safe State and Unsafe State cont…



Deadlock Avoidance (Banker's Algorithm)

- The Bankers Algorithm in OS is **a deadlock avoidance algorithm used to avoid deadlocks and to ensure safe execution of processes**.
- The algorithm maintains a matrix of **allocated resources** and **maximum need** for each process and checks if the system is in a safe state before allowing a process to request additional resources.

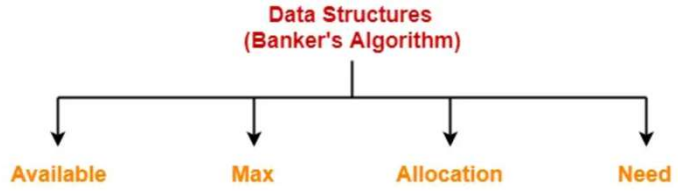
Banker's Algorithm Cont…

- **Banker's Algorithm** is used majorly in the banking system to avoid deadlock. It helps you to identify whether a loan will be given or not.
- This algorithm is used to test for safely simulating the allocation for determining the maximum amount available for all resources. It also checks for all the possible activities before determining whether allocation should be continued or not.

Banker's Algorithm Cont…

- Banker's Algorithm is a deadlock avoidance strategy.
- It is called so because it is used in banking systems to decide whether a loan can be granted or not.
- Whenever a new process is created, it specifies the maximum number of instances of each resource type that it exactly needs.

Banker's Algorithm Cont…



Banker's Algorithm Problem No. - 1

Total resources A=10, B=5, C=7

Process	Allocation			Max Need		
	A	B	C	A	B	C
P1	0	1	0	7	5	3
P2	2	0	0	3	2	2
P3	3	0	2	9	0	2
P4	2	1	1	4	2	2
P5	0	0	2	5	3	3
	7	2	5			

Solution

Total resources A=10, B=5, C=7

Remaining need = Max need - Allocation

Process	Allocation			Max Need			Remaining need			Current Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P1	0	1	0	7	5	3	7	4	3	3	3	2
P2	2	0	0	3	2	2	1	2	2			
P3	3	0	2	9	0	2	6	0	0			
P4	2	1	1	4	2	2	2	1	1			
P5	0	0	2	5	3	3	5	3	1			
	7	2	5									

Safe sequence P2 -> P4 -> P5 -> P1 -> P3

Banker's Algorithm Problem No. - 2

Total resources X=8, Y=4, Z=6

Process	Allocation			Max Need		
	X	Y	Z	X	Y	Z
P0	1	0	1	4	3	1
P1	1	1	2	2	1	4
P2	1	0	3	1	3	3
P3	2	0	0	5	4	1
	5	1	6			

Banker's Algorithm Problem No. - 2

Total resources X=8, Y=4, Z=6

Remaining need = Max need - Allocation

Process	Allocation			Max Need			Remaining need			Current Availability		
	X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z
P0	1	0	1	4	3	1	3	3	0	3	3	0
P1	1	1	2	2	1	4	1	0	2			
P2	1	0	3	1	3	3	0	3	0			
P3	2	0	0	5	4	1	3	4	1			
	5	1	6									

Safe sequence P0 -> P2 -> P1 -> P3

Banker's Algorithm Problem No. - 3

- A system is having 3 processes, each process required 2 units of resource 'R'. What is the minimum no of units of 'R' such that no deadlock will occurs_____?

a) 3

b) 5

c) 6

d) 4

Problem No. – 4

- A **single processor system** has **three resource** types **X, Y and Z**, which are shared by 3 processes. **There are 5 units of each resource type**. Consider the following scenario, where the **column alloc** denotes the number of units of each resource type allocated to each process, and the **column request** denotes the number of units of each resource type requested by a process in order to complete execution. **Which of these processes will finish LAST?**

1. P0 **2. P1** **3. P2**

4. None of the above since the system is in a deadlock

Problem - 4

	Alloc			Request		
	X	Y	Z	X	Y	Z
P0	1	2	1	1	0	3
P1	2	0	1	0	1	2
P2	2	2	1	1	2	0

Solution

According to question;

Total resource = [X Y Z] = [5 5 5]

Total _Allocation = [X Y Z] = [5 4 3]

Now,

Current Availability = Total resource – Total_Allocation

= [5 5 5] – [5 4 3]

= [0 1 2]

Step-01:

- With the instances available currently, only the requirement of the **process P1 can be satisfied**. So, process P1 is allocated the requested resources. It completes its execution and then free up the instances of resources held by it.
- **The Current Availability = [0 1 2] + [2 0 1]**

= [2 1 3]

Step-02:

- With the instances available currently, only the requirement of the process P0 can be satisfied. So, process P0 is allocated the requested resources. It completes its execution and then free up the instances of resources held by it.
- **The Current Availability = [2 1 3] + [1 2 1]**

= [3 3 4]

Step-03:

- With the instances available currently, the requirement of the **process P2 can be satisfied**. So, process P2 is allocated the requested resources. It completes its execution and then free up the instances of resources held by it.
- **The Current Availability = [3 3 4] + [2 2 1]**

= [5 5 5]

- The safe sequence of processes can be executed as P1, P0, P2.
- **The Option (C) is correct.**

Problem No. – 5

- An operating system uses the banker's algorithm for deadlock avoidance when managing the allocation of **3 resource types X, Y and Z** to 3 **processes P0, P1 and P2**. The table given below presents the current system state. Here, the **Allocation matrix** shows the **current number of resources** of each type allocated to each process and the **Max matrix** shows the **maximum number of resources** of each type required by each process during its execution.

Problem – 5

	Allocation			Max		
	X	Y	Z	X	Y	Z
P0	0	0	1	8	4	3
P1	3	2	0	6	2	0
P2	2	1	1	3	3	3

Problem - 5

- There are **3 units of type X, 2 units of type Y and 2 units of type Z still available. The system is currently in safe state.** Consider the following independent requests for additional resources in the current state-

- **REQ1: P0 requests 0 units of X, 0 units of Y and 2 units of Z**
- **REQ2: P1 requests 2 units of X, 0 units of Y and 0 units of Z**

- Which of the following is TRUE?
  1. Only REQ1 can be permitted
  2. Only REQ2 can be permitted
  3. Both REQ1 and REQ2 can be permitted
  4. Neither REQ1 nor REQ2 can be permitted

Solution

- According to question, Available = [X Y Z] = [3 2 2]
- **Now, Need = Max – Allocation**

	Allocation			Max			Need		
	X	Y	Z	X	Y	Z	X	Y	Z
P0	0	0	1	8	4	3	8	4	2
P1	3	2	0	6	2	0	3	0	0
P2	2	1	1	3	3	3	1	2	2

Checking Whether REQ1 Can Be Entertained-

Need of P0 = [0 0 2]

Available = [3 2 2]

- Clearly, With the instances available currently, the requirement of REQ1 can be satisfied. So, banker's algorithm assumes that the request REQ1 is entertained. It then modifies its data structures as-

	Allocation			Max			Need		
	X	Y	Z	X	Y	Z	X	Y	Z
P0	0	0	3	8	4	3	8	4	0
P1	3	2	0	6	2	0	3	0	0
P2	2	1	1	3	3	3	1	2	2

- **Current Availability = [3 2 2] – [0 0 2]**

= [3 2 0]

- Now, it follows the safety algorithm to check whether this resulting state is a safe state or not.
- If it is a safe state, then REQ1 can be permitted otherwise not.

Step-01:

- With the instances available currently, only the requirement of the **process P1** can be satisfied. So, **process P1 is allocated the requested resources.** It completes its execution and then free up the instances of resources held by it.
- Then, **Current Availability = [3 2 0] + [3 2 0]**

= [6 4 0]

- Now, It is not possible to entertain any process. The system has entered the deadlock state which is an unsafe state. Thus, REQ1 will not be permitted.

Checking Whether REQ2 Can Be Entertained-

Need of P1 = [2 0 0]

Available = [3 2 2]

- Clearly, With the instances available currently, the requirement of REQ1 can be satisfied. So, banker's algorithm assumes the request REQ2 is entertained. It then modifies its data structures as-

	Allocation			Max			Need		
	X	Y	Z	X	Y	Z	X	Y	Z
P0	0	0	1	8	4	3	8	4	2
P1	5	2	0	6	2	0	1	0	0
P2	2	1	1	3	3	3	1	2	2

- **Current Availability = [3 2 2] – [2 0 0]**

  = [1 2 2]

- Now, it follows the safety algorithm to check whether this resulting state is a safe state or not.
- If it is a safe state, then REQ2 can be permitted otherwise not.

Step-01:

- With the instances available currently, only the requirement of the **process P1** can be satisfied. So, **process P1** is allocated the requested resources. It completes its execution and then free up the instances of resources held by it.
- Then, **Current Availability = [1 2 2] + [5 2 0]**

  = [6 4 2]

Step-02:

- With the instances available currently, only the requirement of the **process P2** can be satisfied. So, **process P2** is allocated the requested resources. It completes its execution and then free up the instances of resources held by it.
- Then, **Current Availability = [6 4 2] + [2 1 1]**

  = [8 5 3]

Step-03:

- With the instances available currently, the requirement of the **process P0** can be satisfied. So, process P0 is allocated the requested resources. It completes its execution and then free up the instances of resources held by it.
- Then, **Current Availability = [8 5 3] + [0 0 1]**

  = [8 5 4]

- Thus, There exists a safe sequence P1, P2, P0 in which all the processes can be executed. So, the system is in a safe state.
- **Thus, REQ2 can be permitted. Thus, Correct Option is (B).**

Problem No. – 6

- A system has **4 processes** and **5 allocatable resource**. The current allocation and maximum needs are as follows-

	Allocated					Maximum				
A	1	0	2	1	1	1	1	2	1	3
B	2	0	1	1	0	2	2	2	1	0
C	1	1	0	1	1	2	1	3	1	1
D	1	1	1	1	0	1	1	2	2	0

- If Available = [0 0 X 1 1], what is the smallest value of x for which this is a safe state?

Solution

- Let us calculate the additional instances of each resource type needed by each process.
- **Need = Maximum–Allocation**

	Need				
A	0	1	0	0	2
B	0	2	1	0	0
C	1	0	3	0	0
D	0	0	1	1	0

	Allocated					Maximum					Need				
A	1	0	2	1	1	1	1	2	1	3	0	1	0	0	2
B	2	0	1	1	0	2	2	2	1	0	0	2	1	0	0
C	1	1	0	1	1	2	1	3	1	1	1	0	3	0	0
D	1	1	1	1	0	1	1	2	2	0	0	0	1	1	0

Case-01: For X = 0

- If X = 0, then, **Current Availability = [0 0 0 1 1]**
- With the instances available currently, the requirement of any process can not be satisfied. So, for X = 0, system remains in a deadlock which is an unsafe state.

Case-02: For X = 1

- If X = 1, then **Current Availability = [0 0 1 1 1]**

Step-01:

- With the instances available currently, only the requirement of the **process D can be satisfied**. So, process D is allocated the requested resources. It completes its execution and then free up the instances of resources held by it.
- Then, **Current Availability = [0 0 1 1 1] + [1 1 1 1 0]**

= [1 1 2 2 1]

- With the instances available currently, the requirement of any process can not be satisfied. **So, for X = 1, system remains in a deadlock which is an unsafe state.**

Case 02: For X = 2

- If X = 2, then, **Current Availability = [0 0 2 1 1]**

Step-01:

- With the instances available currently, only the requirement of the **process D can be satisfied**. So, process D is allocated the requested resources. It completes its execution and then free up the instances of resources held by it.
- Then, **Current Availability = [0 0 2 1 1] + [1 1 1 1 0]**

= [1 1 3 2 1]

Step-02:

- With the instances available currently, only the requirement of the **process C can be satisfied**. So, process C is allocated the requested resources. It completes its execution and then free up the instances of resources held by it.
- Then, **Current Availability = [1 1 3 2 1] + [1 1 0 1 1]**

= [2 2 3 3 2]

Step-03:

- With the instances available currently, the requirement of both the **processes A and B can be satisfied**. So, processes A and B are allocated the requested resources one by one. They complete their execution and then free up the instances of resources held by it.

Current Availability = [2 2 3 3 2] + [1 0 2 1 1] + [2 0 1 1 0]

= [5 2 6 5 3]

- Thus, There exists a safe sequence in which all the processes can be executed. **So, the system is in a safe state.**
- **The minimum value of X=2 that ensures system is in safe state.**

Thank You

Operating System

Presented by:
Dr. Premnarayan Arya
Assistant Professor
Department of CEA, GLA University, Mathura

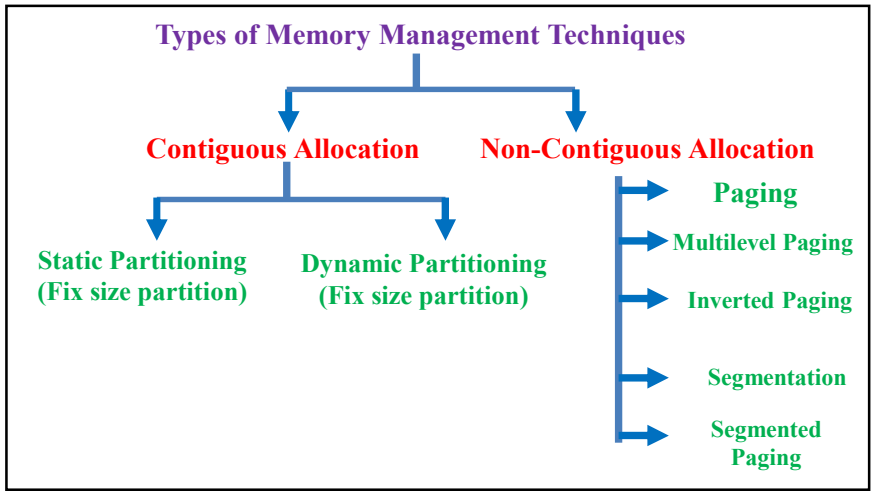
Contents

Memory Management

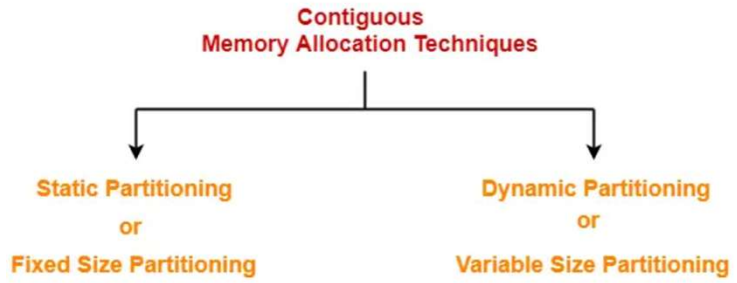
- Memory management is a technique of controlling and managing the functionality of RAM.
- It is used for achieving better concurrency, system performance, and memory utilization.
- Memory management moves processes from primary memory to secondary memory and vice versa.
- It also keeps track of available memory, memory allocation, and unallocated.

Contiguous Memory Allocation

- In contiguous memory allocation each process is contained in a single contiguous block of memory.
- Memory is divided into several fixed size partitions. Each partition contains exactly one process.
- When a partition is free, a process is selected from the input queue and loaded into it.
- The free blocks of memory are known as *holes*. The set of holes is searched to determine which hole is best to allocate.



Contiguous Memory Allocation

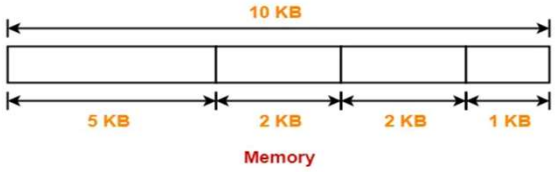


Static Partitioning
or
Fixed Size Partitioning

Static Partitioning

- Static partitioning is a **fixed size** partitioning (memory blocks).
- The main memory (RAM/Physical Memory) is **pre-divided** into fixed size partitions.
- The size of each partition is fixed and can not be changed.
- Only one process is allowed to store in each partition.

Static Partitioning



- These partitions are allocated to the processes as they arrive.
- The partition allocated to the arrived process depends on the algorithm followed.

Advantages of Static Partition

- It is simple and easy to implement.
- It supports multiprogramming since multiple processes can be stored inside the main memory.
- Only one memory access is required which reduces the access time.

Disadvantages of Static Partition

- It suffers from both internal fragmentation and external fragmentation.
- It utilizes memory inefficiently.
- The degree of multiprogramming is limited equal to number of partitions.
- There is a limitation on the size of process since processes with size greater than the size of largest partition can't be stored and executed.

Fragmentation

- Fragmentation refers to **an unwanted problem that occurs in the memory management (Operating System).**
- **A process is unloaded and loaded from memory, and the free memory space gets fragmented**.
- The processes can not be assigned to the memory blocks because of their small size.

External Fragmentation

- In such situation processes are loaded and removed from the memory.
- As a result of this, free holes exists to satisfy a request but is non contiguous i.e. the memory is fragmented into large no. of small holes. This phenomenon is known as **External Fragmentation.**

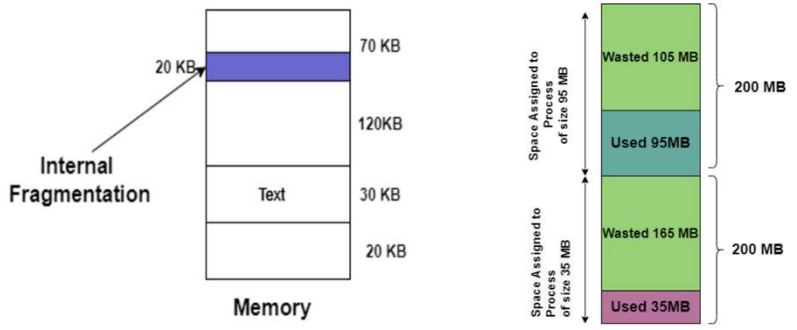
Internal Fragmentation

- The physical memory is fragmented into fixed size blocks and memory is allocated in unit of block sizes.
- The memory allocated to a space may be slightly larger than the requested memory.
- The difference between allocated and required memory is known as **Internal fragmentation** i.e. the memory that is internal to a partition but is of no use.

Internal Fragmentation Cont…

- It occurs when the **space is left inside the partition** after allocating the partition to a process.
- This space is called as **internally fragmented space**.
- This space **can not be allocated** to any other process because **static partitioning** allows to store only one process in each partition.
- Internal Fragmentation occurs only in static partitioning.

Internal Fragmentation Cont…

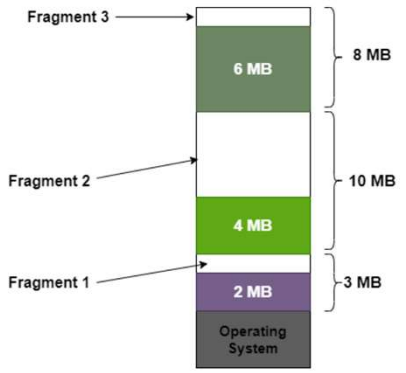


There is some wastage memory space inside the partition.

External Fragmentation

- It occurs when the **total amount of empty space** required to store the process is available in the main memory.
- But because the **space is not contiguous**, so the **process can not be stored.**
- This problem is known as **External Fragmentation.**

Diagram of External Fragmentation



Example of External Fragmentation

- Suppose, we want to allocate 8 MB size process and we have 9 MB space, but the memory that is fragments are not contiguous.

Dynamic Partitioning or Variable Size Partitioning

Dynamic Partitioning

- Dynamic partitioning is a variable size partitioning scheme.
- It performs the allocation dynamically.
- When a process arrives, a partition of size equal to the size of process is created.
- Then, that partition is allocated to the process.

Dynamic Partitioning

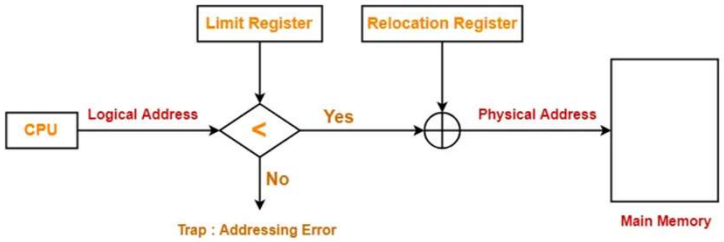
- Partition allocation algorithms are used to decide which hole should be allocated to the arrived process.
- The processes arrive and leave the main memory.
- As a result, holes of different size are created in the main memory.
- These holes are allocated to the processes that arrive in future.

Dynamic Partitioning

For dynamic partitioning

1. Worst Fit Algorithm works best. This is because space left after allocation inside the partition is of large size. There is a high probability that this space might suit the requirement of arriving processes.
2. Best Fit Algorithm works worst. This is because space left after allocation inside the partition is of very small size. There is a low probability that this space might suit the requirement of arriving processes.

Translating Logical Address into Physical Address



Advantages of Dynamic Partitioning

- It does not suffer from internal fragmentation.
- Degree of multiprogramming is dynamic.
- There is no limitation on the size of processes.

Disadvantages of Dynamic Partitioning

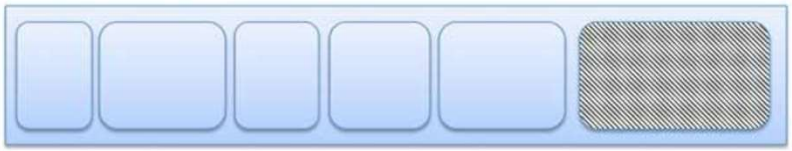
- It suffers from external fragmentation.
- Allocation and deallocation of memory is complex.

Compaction

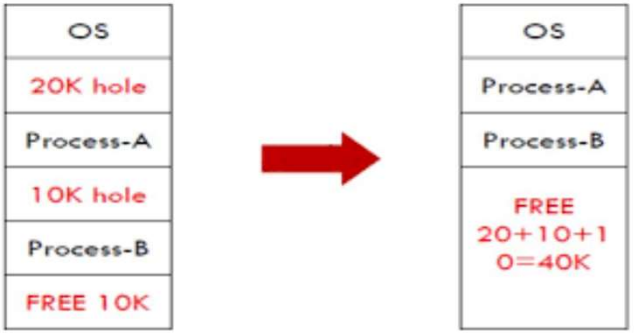
- **Compaction** is a technique to solve the External Fragmentation problem.
- In Compaction, the free spaces are collected from different-different blocks in the large memory chunk to make some space for allocate other processes.

Compaction cont…

- Compaction is not always possible. Suppose if relocation is static and done at the load time then in that case compaction cannot be done. Compaction is only possible if the relocation is dynamic and done at the execution time.



Example of Compaction



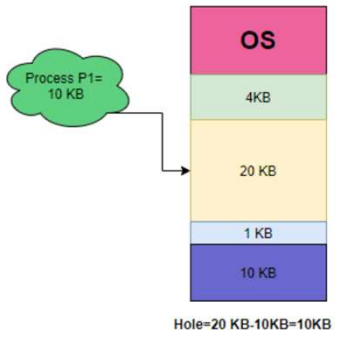
Partition Allocation Algorithms

Partition Allocation Algorithms

1. First Fit Algorithm
2. Best Fit Algorithm
3. Worst Fit Algorithm

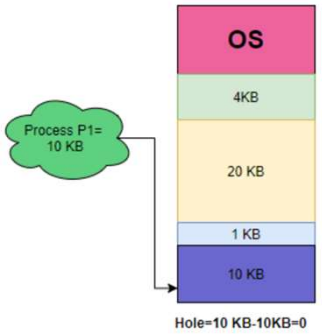
(1) First Fit Algorithm

- Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.



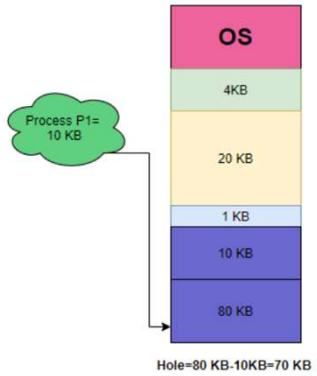
(2) Best Fit Algorithm

- Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.



(3) Worst Fit Algorithm

- Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.



Important Points

For static partitioning,

1. Best Fit Algorithm works best because space left after the allocation inside the partition is of very small size. Thus, internal fragmentation is least.
2. Worst Fit Algorithm works worst because space left after the allocation inside the partition is of very large size. Thus, internal fragmentation is maximum.

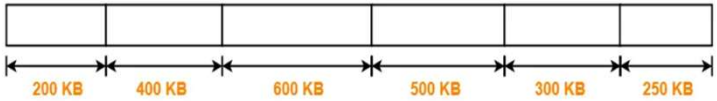
Problem 1

- Consider six memory partitions of size 200 KB, 400 KB, 600 KB, 500 KB, 300 KB and 250 KB. These partitions need to be allocated to four processes of sizes 357 KB, 210 KB, 468 KB and 491 KB in that order. Perform the allocation of processes using-

1. First Fit Algorithm
2. Best Fit Algorithm
3. Worst Fit Algorithm

Solution

- According to question, The main memory has been divided into fixed size partitions as-



The given processes are-

- Process P1 = 357 KB, Process P2 = 210 KB
- Process P3 = 468 KB, Process P4 = 491 KB

Solution Cont…

Allocation Using First Fit Algorithm:

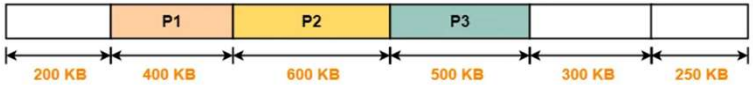
- Algorithm starts scanning the partitions serially.
- When a partition big enough to store the process is found, it allocates that partition to the process.

Step 1:

Solution Cont…

Step 2:

Step-03:



Solution Cont…

Step-04:

- Process P4 can not be allocated the memory.
- This is because no partition of size greater than or equal to the size of process P4 is available.

Solution Cont…

Allocation Using Best Fit Algorithm:

- Algorithm first scans all the partitions.
- It then allocates the partition of smallest size that can store the process.

Step 1:

Solution Cont…

Step-02:

Step-03:

Solution Cont…

Step-04:

Solution Cont…

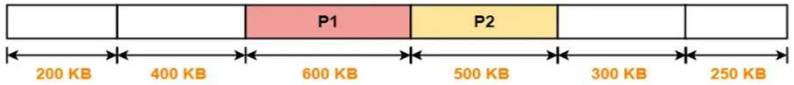
Allocation Using Worst Fit Algorithm:

- Algorithm first scans all the partitions.
- It then allocates the partition of largest size to the process.

Step-01:

Solution Cont…

Step-02:

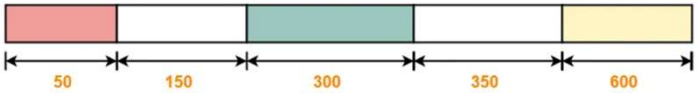


Step-03:

- Process P3 and Process P4 can not be allocated the memory.
- This is because no partition of size greater than or equal to the size of process P3 and process P4 is available.

Problem 2

- Consider the following heap (figure) in which blank regions are not in use and hatched regions are in use:



The sequence of requests for blocks of size 300, 25, 125, 50 can be satisfied if we use:

1. Either first fit or best fit policy (any one)
2. First fit but not best fit policy
3. Best fit but not first fit policy
4. None of the above

Solution

The allocation follows variable size partitioning scheme.

The given processes are-

- Process P1 = 300 units
- Process P2 = 25 units
- Process P3 = 125 units
- Process P4 = 50 units

Solution Cont…

Allocation Using First Fit Algorithm:

Step-01:

Step-02:

Solution Cont…

Step-03:

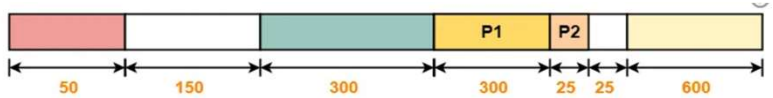
Step-04:

Solution Cont…

Allocation Using Best Fit Algorithm:

Step-01:

Step-02:



Solution Cont…

Step-03:

Step-04:

- Process P4 can not be allocated the memory.
- This is because no partition of size greater than or equal to the size of process P4 is available.

Thus,

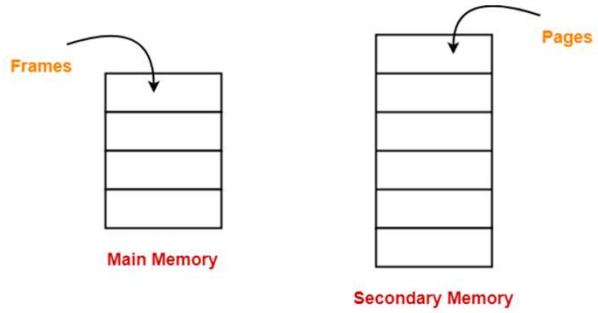
- Only first fit allocation policy succeeds in allocating memory to all the processes.
- Option (B) is correct.

Paging

Paging

- Paging is a fixed size partitioning scheme. In paging, the secondary memory and main memory are divided into equal fixed size partitions.
- The partitions of secondary memory are called as **pages**. The partitions of main memory are called as **frames**.
- The **page size** should be equal to **frame size**.

Paging cont…



Paging cont…

- The pages of process are stored in the frames of main memory depending upon their availability.
- For example: a process is divided into 4 pages P_0, P_1, P_2 and P_3. Depending upon the availability, these pages may be stored in the main memory frames in a non-contiguous manner.

Translating Logical Address into Physical Address

- CPU always generates a logical address.
- A physical address is needed to access the main memory.

- How to translate logical address into physical address?

Step 1

Step 1

CPU generates a logical address consisting of two parts-

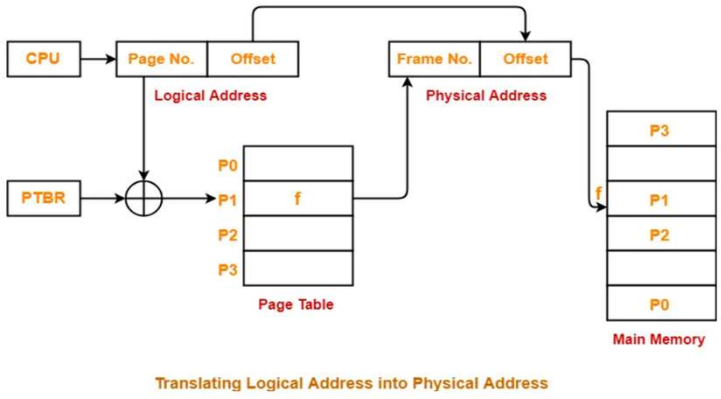
1. Page Number: Page Number specifies the specific page of the process from which CPU wants to read the data.
2. Page Offset: Page Offset specifies the specific word on the page that CPU wants to read.

Step 2

- For the page number generated by the CPU, **Page Table** provides the corresponding frame number (base address of the frame) where that page is stored in the main memory.

Step 3

- The frame number combined with the page offset forms the required physical address.
- Frame number specifies the specific frame where the required page is stored.
- Page Offset specifies the specific word that has to be read from that page.



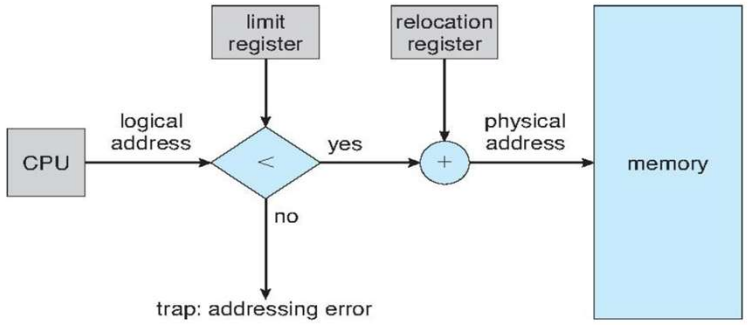
Translating Logical Address into Physical Address

Case-01: Generated Address >= Limit

- If address is found to be greater than or equal to the limit, a trap is generated. This helps to prevent unauthorized access.

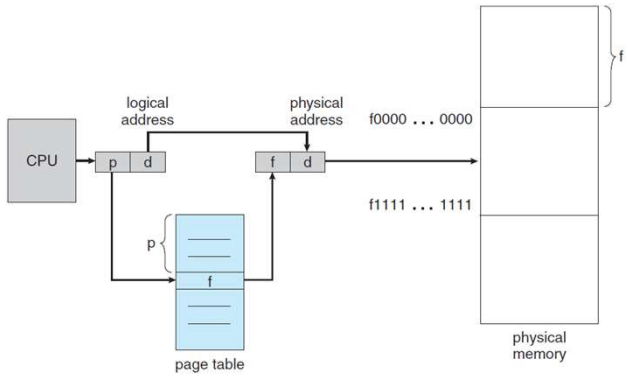
Case-02: Generated Address < Limit

- The address must always lie in the range [0, limit-1].
- If address is found to be smaller than the limit, then the request is treated as a valid request. Then, generated address is added with the base address of the process. The result obtained after addition is the address of the memory location storing the required word.



Relocation register contains the value of the smallest physical address;
the limit register contains the range of logical addresses.

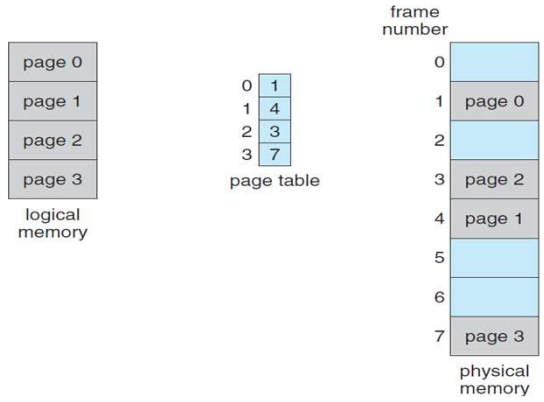
Paging hardware



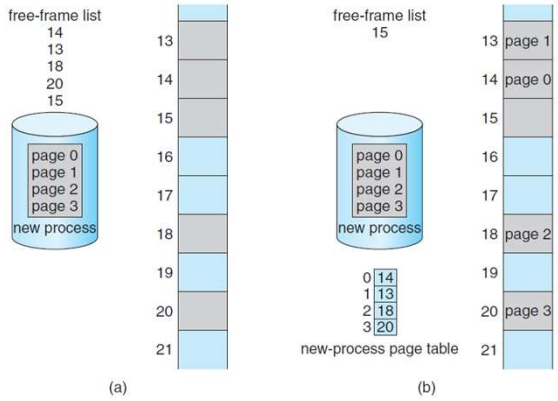
Paging hardware cont…

- 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.

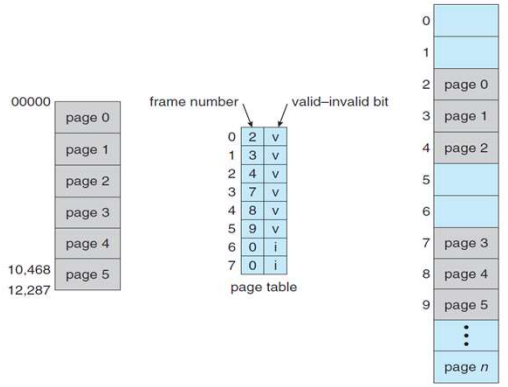
Paging model of logical and physical memory



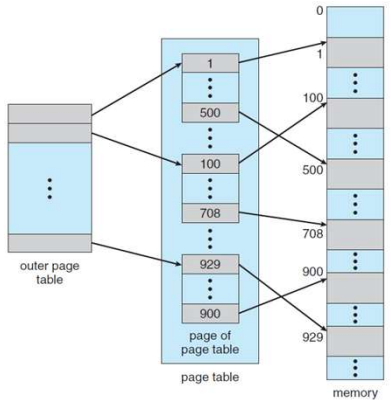
Free frames (a) before allocation and (b) after allocation.



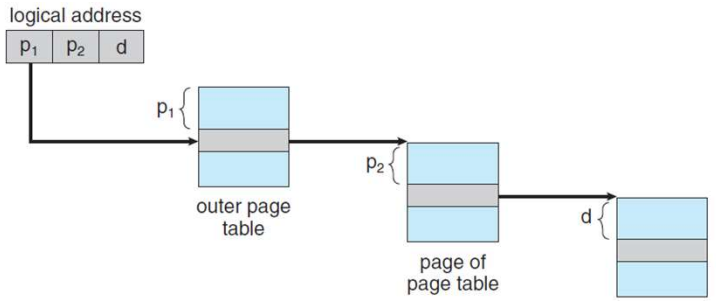
Valid (v) or invalid (i) bit in a page table



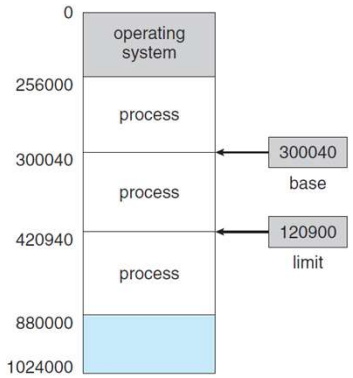
A two-level page-table scheme



Address translation for a two-level 32-bit paging architecture

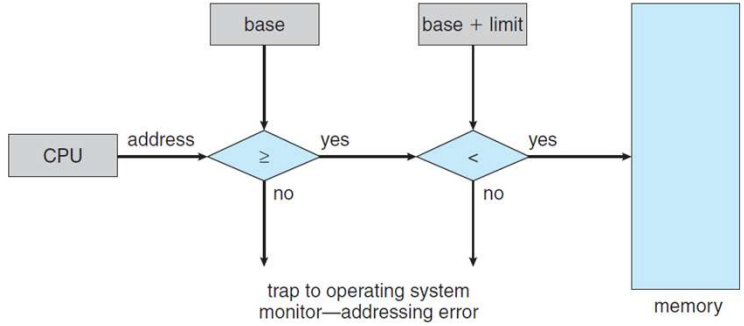


A base and a limit register define a logical address space

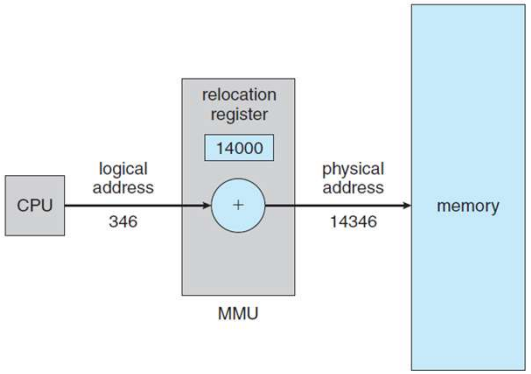


- if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).

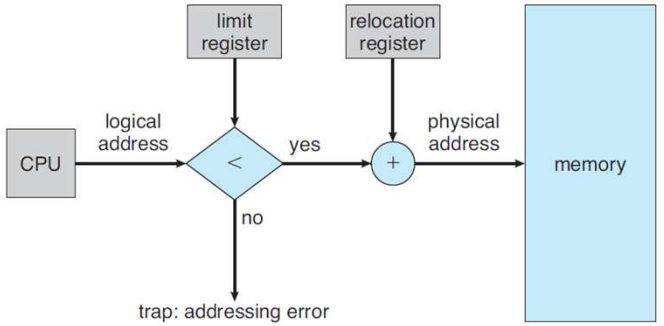
Hardware address protection with base and limit registers



Dynamic relocation using a relocation register



Hardware support for relocation and limit registers



Thank You

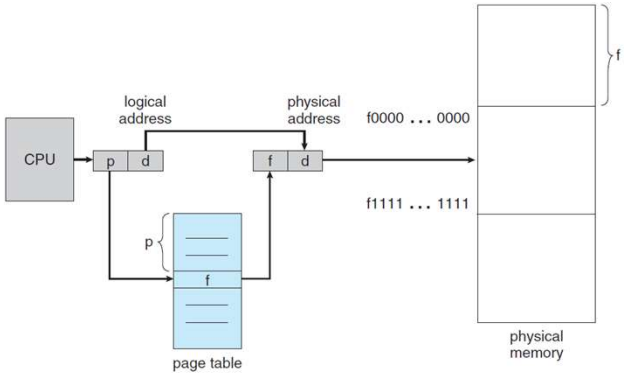
Operating System

Presented by:
Dr. Premnarayan Arya
Assistant Professor
Department of CEA, GLA University, Mathura

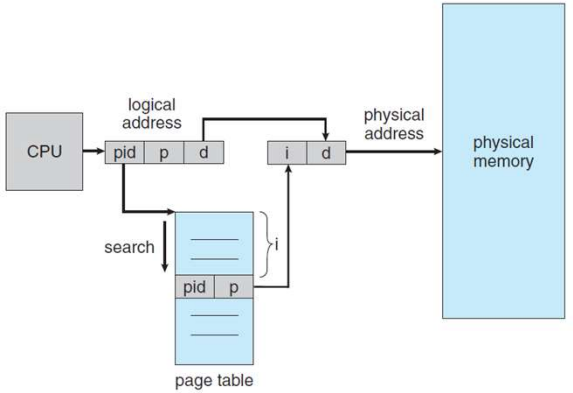
Contents

- Paging
- Inverted Paging
- Segmentation
- Thrashing
- Translation Lookaside Buffer (TLB)
- Swapping

Paging



Inverted Paging



Segmentation

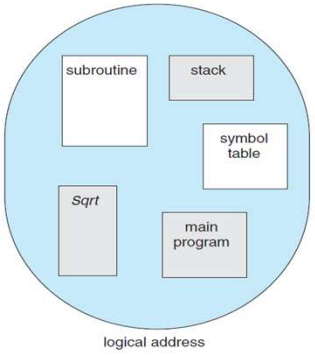
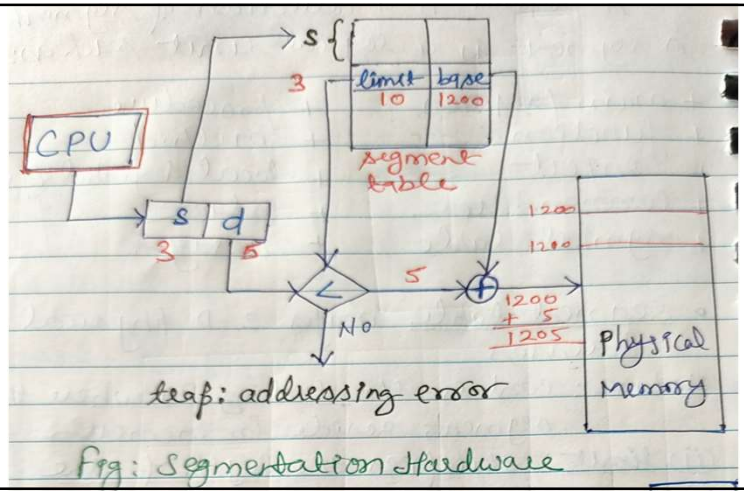
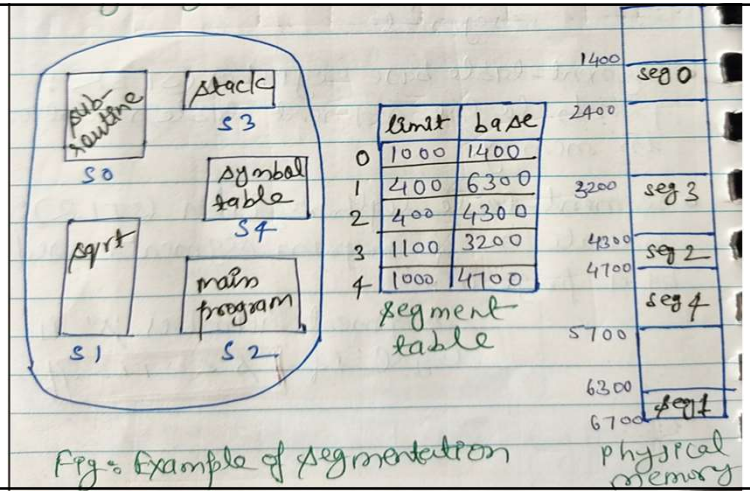
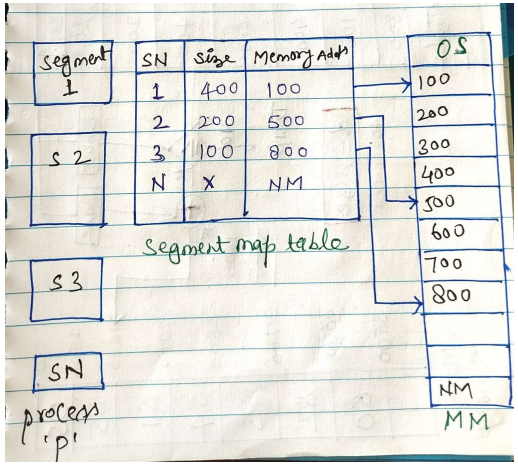
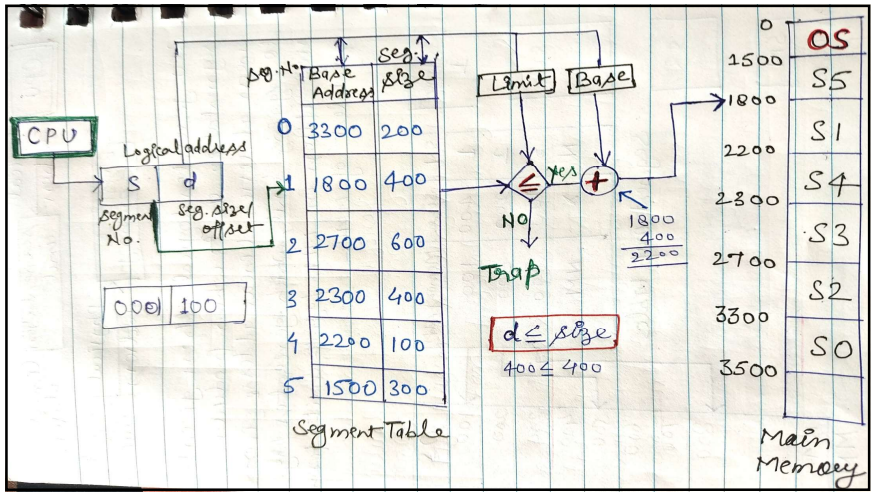


Figure 8.7 Programmer's view of a program.









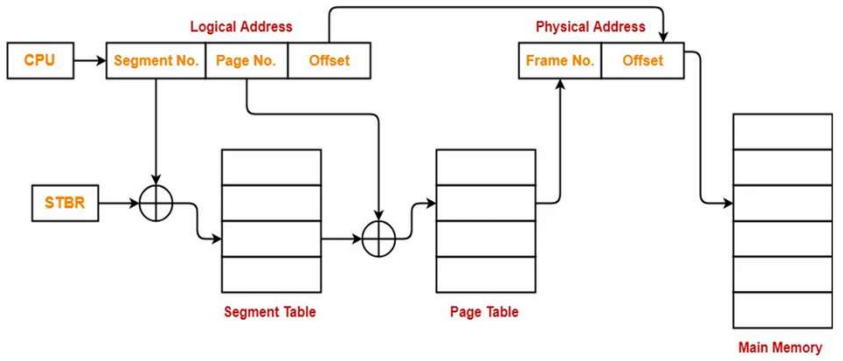
Segmented Paging

- A process divides into **equal** size partitions called as pages. **A** process divides into **unequal** size partitions called as segments. But, in segmentation there is external fragmentation problem. So that remove this problem we can use segmented paging technique.
- Segmented paging is a technique that implements with the combination of segmentation and paging.
- Process is divided into segments then each segment further divided into pages. These pages are then stored in the frames of main memory.

Segmented Paging Cont…

- A page table exists for each segment that keeps track of the frames storing the pages of that segment. Each page table occupies one frame in the main memory.
- Number of entries in the page table of a segment = Number of pages that segment is divided. A segment table exists that keeps track of the frames storing the page tables of segments.
- Number of entries in the segment table of a process = Number of segments that process is divided. The base address of the segment table is stored in the segment table base register.

Segmented Paging Cont…



Thrashing

- **Thrashing** is a situation when the system is spending more time to servicing the page faults, but the actual processing done is very slow (negligible).

Causes of thrashing:

1. High degree of multiprogramming.
2. Lack of frames.
3. Page replacement policy.

Diagram of Thrashing

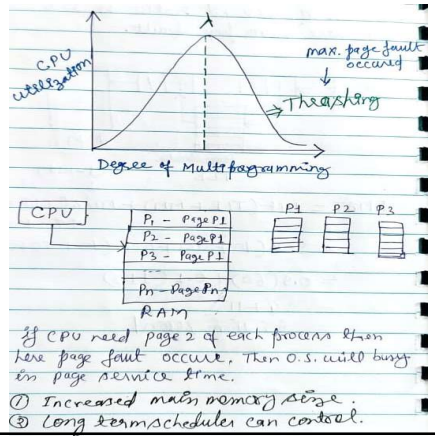
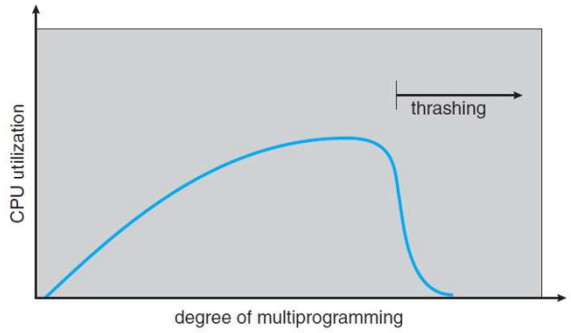


Diagram of Thrashing



Translation Lookaside Buffer (TLB)

- TLB is a solution that tries to reduce the effective access time.
- The access time of TLB is very less as compared to the main memory.
- It is used to reduce the time taken to access memory location.

Diagram of Translation Lookaside Buffer

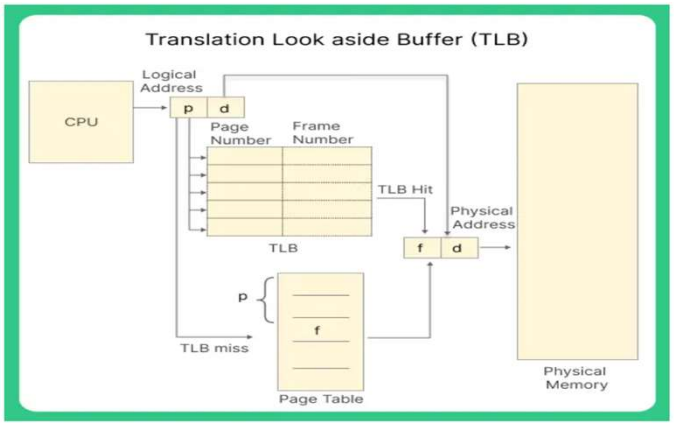
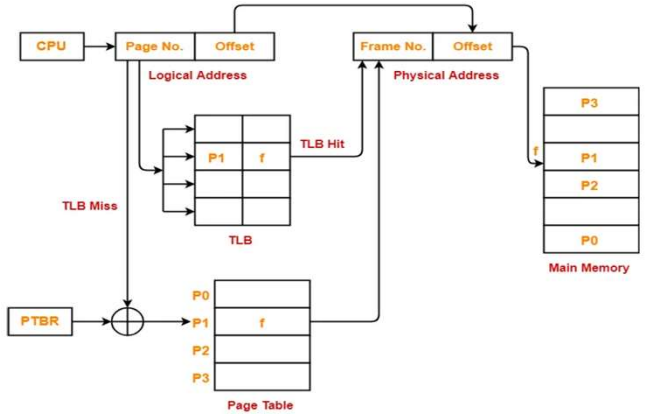
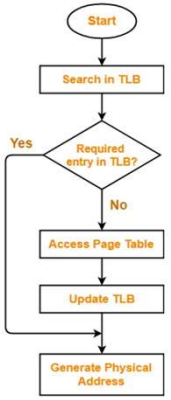


Diagram of Translation Lookaside Buffer



Flowchart



Translation Lookaside Buffer

- Unlike page table, there exists only one TLB in the system. So, whenever context switching occurs, the entire content of TLB is flushed and deleted. TLB is then again updated with the currently running process.
- When a new process gets scheduled: Initially, TLB is empty. So, TLB misses are frequent. With every access from the page table, TLB is updated. After some time, TLB hits increases and TLB misses reduces.
- The time taken to update TLB after getting the frame number from the page table is negligible. Also, TLB is updated in parallel while fetching the word from the main memory.

Effective Memory Access Time

EMAT=

Hit ration of TLB *{Access time of TLB + Access time of main memory}

+

Miss ration of TLB *{Access time of TLB + Access time of Page Table

+Access time of main memory}

EMAT

EMAT = h*(c+m) + (1-h)*(c+2m)

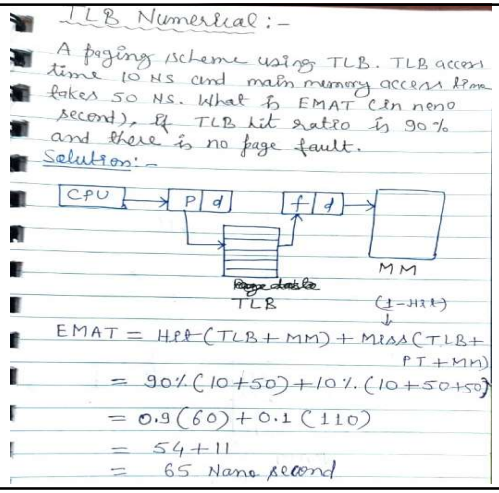
where,

- h = hit ratio of TLB, c = TLB access time, m = Memory access time

EMAT = Hit(TLB+MM) + Miss(TLB+PT+MM)

where,

- TLB-Translation lookaside buffer, MM-Main Memory, PT-Page Table



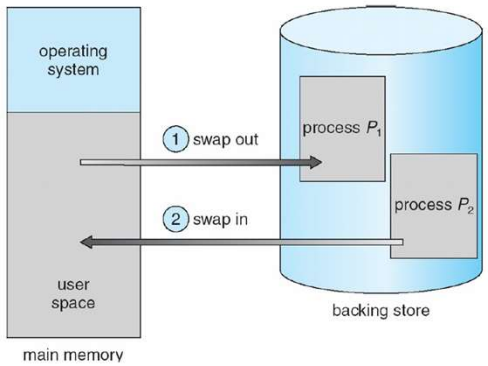
TLB Numerical - 2

- A paging scheme uses a Translation Lookaside buffer (TLB). The effective memory access takes 160ns and a main memory access takes 100 ns. What is the TLB access time (in ns) if the TLB hit ratio is60% and there is no page fault?
  a) 54
  b) 60
  c) 20
  d) 75

Swapping

- A process must be in memory to be executed. A process, however, can be **swapped** temporarily out of memory to a **backing store** and then brought back into memory for continued execution. Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming in a system.

Diagram of Swapping



Thank You

Operating System

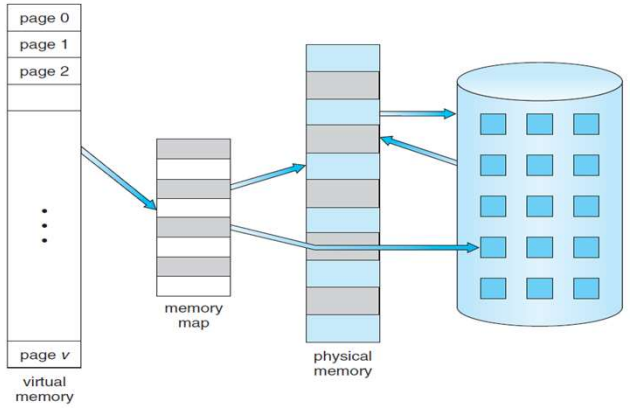
Presented by:
Dr. Premnarayan Arya
Assistant Professor
Department of CEA, GLA University, Mathura

Contents

Virtual Memory

- Virtual memory is a memory management technique that allows the computer to execute big size process than the small size main memory (RAM).
- We can also say that virtual memory in Operating System makes an illusion of extra memory than available physical memory.

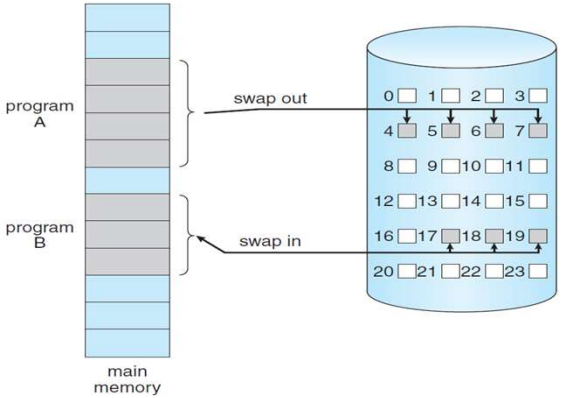
Diagram showing virtual memory that is larger than physical memory



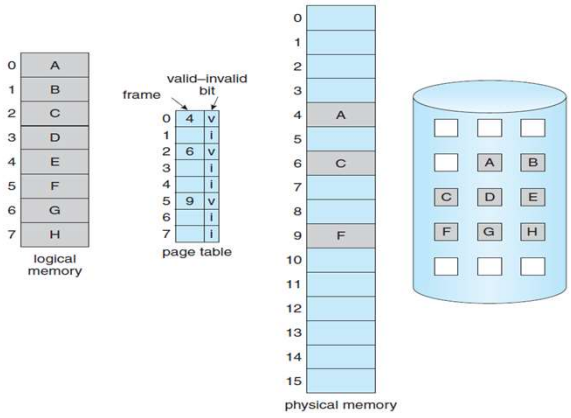
Demand Paging

- Demand paging is a technique in which pages are **loaded** from the secondary memory (hard disk) to the main memory only when they are **needed**.
- This technique allows the operating system to save memory space by keeping only those pages in main memory that are currently required by the program

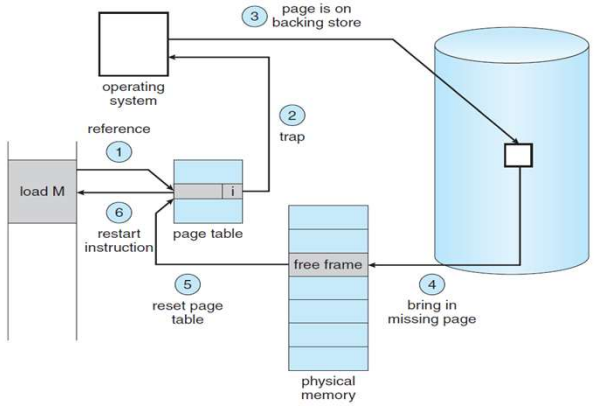
Transfer of a paged memory to contiguous disk space



Page table when some pages are not in main memory



Steps in handling a page fault



What is Page replacement?

- Page replacement happens when a **requested page** is not available in the main memory, this situation is known as **page fault.**
- Operating System **replace** existing pages with the newly needed page.
- Different **page replacement algorithms** suggest different ways to decide which page to replace.

What is Page replacement?

- A **page fault** occurs when a page referenced by the CPU is not found in the main memory.
- The **required page** has to be **brought** from the secondary memory into the main memory.
- A page has to be **replaced** if all the frames of main memory are already occupied.

Page replacement Algorithms

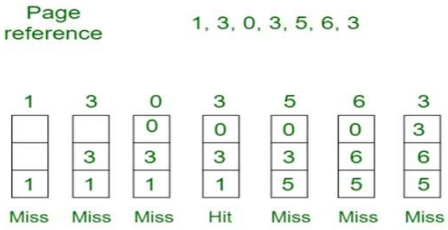
1. First In First Out (FIFO)
2. Optimal Page replacement:
3. Least Recently Used

1. First In First Out (FIFO) Algorithm

- FIFO algorithm mainly **replaces** the oldest page that has been present in the main memory for the longest time.
- This algorithm is implemented by **keeping the record** of all the pages in the queue.
- As new pages are requested and are swapped in, they are added to the tail of a queue and the page which is at the head becomes the victim.

Example of FIFO Algorithm

- Consider **page reference string 1, 3, 0, 3, 5, 6, 3** with **3 page frames**. Find the number of page faults.



Belady's anomaly

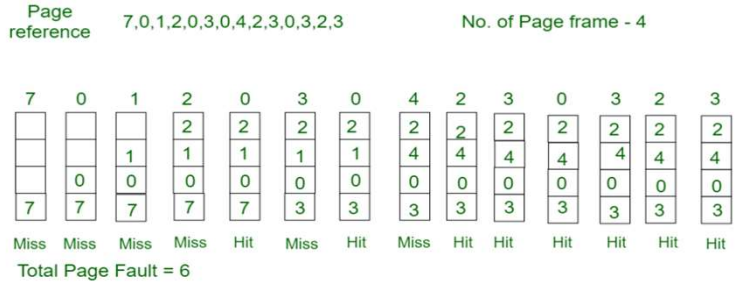
- **Belady's anomaly** proves that it is possible to have more page faults when **increasing the number of page frames** while using the First in First Out (FIFO) page replacement algorithm.
- For example, if we consider **reference strings 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4**, and **3 slots**, we get 9 total page faults, but if we **increase slots to 4**, we get 10-page faults.

2. Optimal Page replacement Algorithm

- In this algorithm, pages are **replaced** which would not be used for the longest duration of time in the **future**.

Example of Optimal Page replacement Algorithm

- Consider the page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3 with 4 page frame. Find number of page fault.

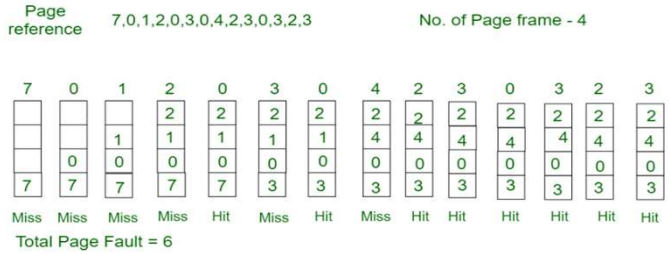


3. Least Recently Used Algorithm

- In LRU, page will be replaced which is least recently used.
- The page that has not been used for the longest time in the main memory will be selected for replacement.

Example of LRU Algorithm

- Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3 with 4 page frames. Find number of page faults.



Here LRU has same number of page fault as optimal but it may differ according to question.

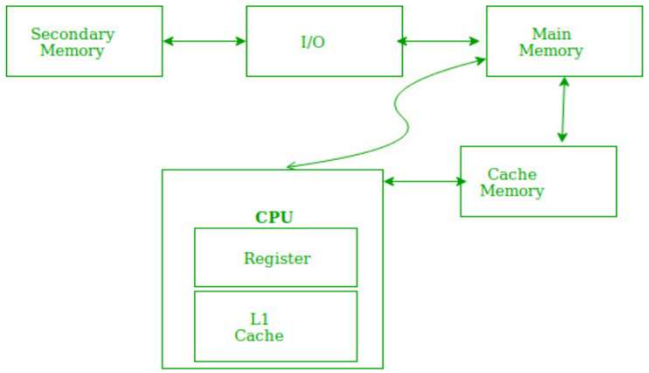
Important Points

- The Ratio of **Page Hit** and **Page Fault** = **8** : **12**
  = 2 : 3
  = 0.66
- The **Page Hit** Percentage = 8 *100 / 20 = 40%
- The **Page Fault** Perc. = 100 - Page Hit Perc. = 100 - 40 = 60%

Locality of Reference

- Locality of reference is a principle in computer program and computer architecture that describes the tendency of a computer program to access the **same set of memory locations repeatedly** over a short period of time.
- The property of Locality of Reference is mainly shown by **loops** and **subroutine** calls in a program.

Locality of Reference



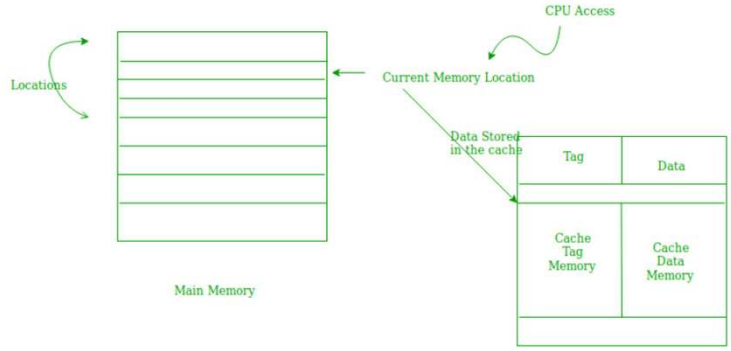
Types of Locality of Reference

(1) Temporal locality

Temporal locality means **current data or instruction** that is being fetched may be needed soon.

So we should store that data or instruction in the **cache memory** so that we can avoid again searching in main memory for the same data.

Diagram of Temporal locality

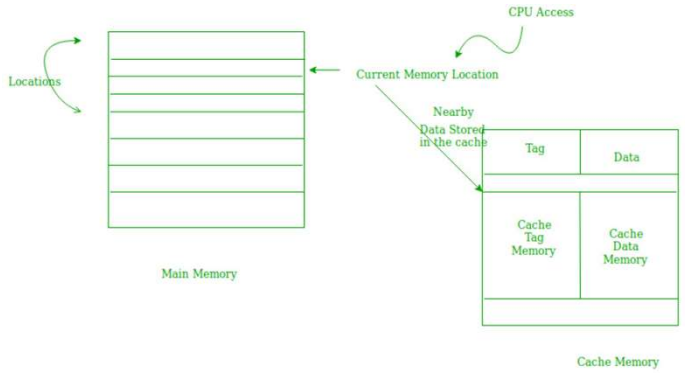


Types of Locality of Reference

(2) Spatial locality

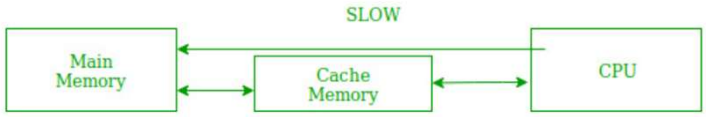
Spatial locality means **instruction or data** near to the current memory location that is being fetched, may be needed soon in the near future. This is slightly different from the temporal locality. Here we are talking about nearly located memory locations while in temporal locality we were talking about the actual memory location that was being fetched.

Diagram of Partial locality

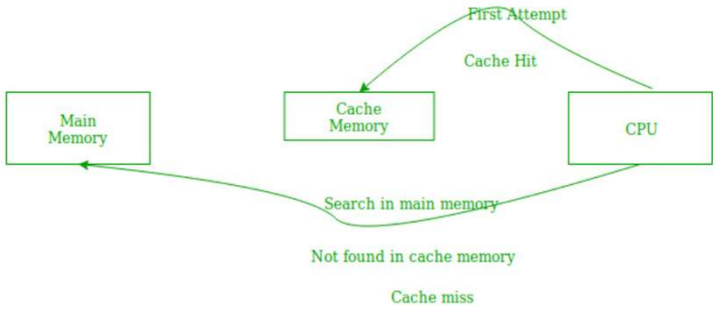


Cache Performance

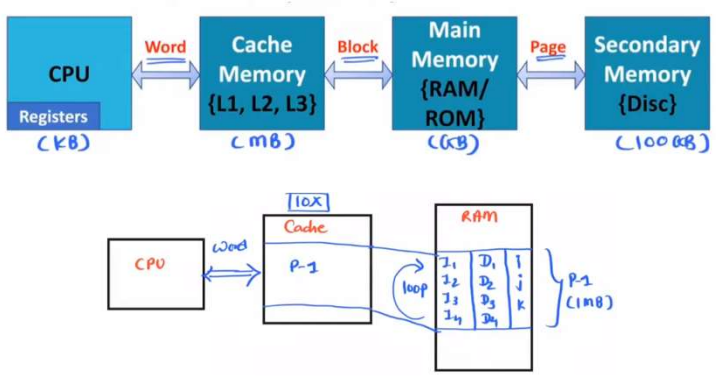
- Total CPU Reference = Hit + Miss
- Hit Ratio(h) = Hit / (Hit+Miss)
- Miss Ratio = 1 - Hit Ratio(h)



Cache Performance



Cache Performance



Thank You

Operating System

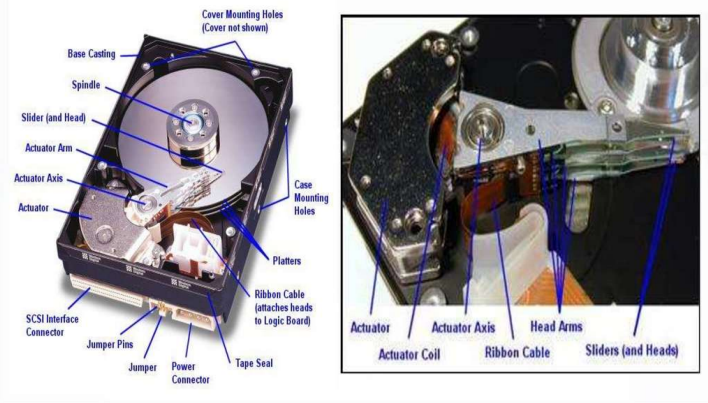
Presented by:
Dr. Premnarayan Arya
Assistant Professor
Department of CEA, GLA University, Mathura

Contents

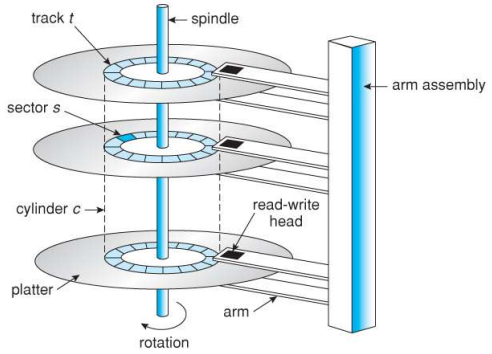
Hard Disk Architecture

- **Hard disks** are secondary storage devices we can use to store data. Most modern computers use hard disks to store large amounts of data.
- The components of a hard disk:
  - Spindle
  - Platters
  - Read/write heads
  - Tracks
  - Sectors

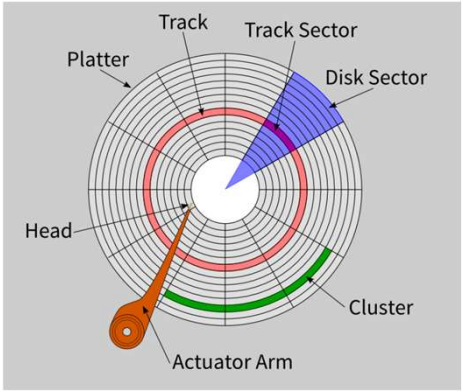
ARCHITECTURE OF THE HARD DISK



Internal Architecture of Hard Disk



Track & Sector



Disk Access Time

- A process needs two type of time, CPU time and IO time. For I/O, it requests the Operating system to access the disk.
- Read-Write(R-W) head moves over the rotating hard disk. The R-W head that performs all the read and writes operations on the disk and hence, the position of the R-W head is a major concern.
- To perform a read or write operation on a memory location, we need to place the R-W head over that position.

Disk Access Time

1. **Seek time –** The time taken by the R-W head to reach the desired track from its current position.
2. **Rotation Time:** Time taken for one full rotation (360 degree).
3. **Rotational latency –** Time taken to reach to desired sector (half of rotation time).
4. **Data transfer time –** Time is taken to transfer the required amount of data. It depends upon the rotational speed.
5. **Average Access time** = seek time + Average Rotational latency + data transfer time

Disk Scheduling Algorithm

A process needs two type of time, **CPU time** and **I/O time.** For I/O, it requests the Operating system to access the disk.

1. FCFS scheduling algorithm
2. SSTF (shortest seek time first) algorithm
3. SCAN scheduling
4. C-SCAN scheduling
5. LOOK Scheduling
6. C-LOOK scheduling

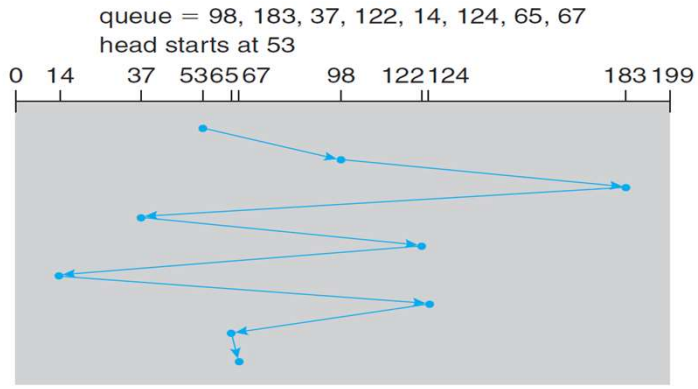
FCFS Disk Scheduling Algorithm

- **First Come First Serve** is the simplest Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue.

Example:

- A disk contains 200 tracks (0-199), requests queue contains track numbers **98, 183, 37, 122, 14, 124, 65, 67,** And current position of Read/Write head is **53.**

FCFS Disk Scheduling Algorithm



FCFS Disk Scheduling Algorithm

Number of cylinders moved by the R/W head

=(98-53)+(183-98)+(183-37)+(122-37)+(122-14)+

(124-14)+(124-65)+(67-65)

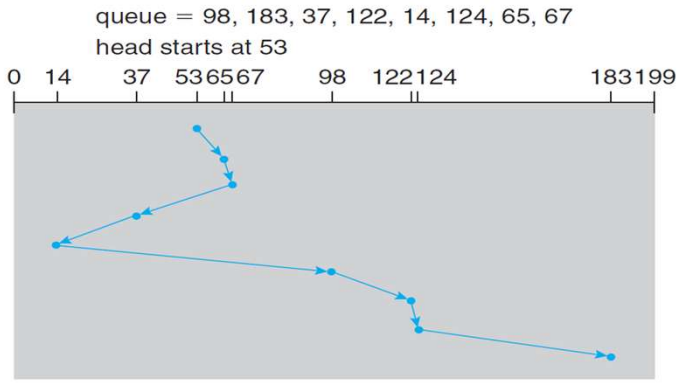
= 45+85+146+85+108+110+59+2

= **640 cylinders**

SSTF Disk Scheduling Algorithm

- Select the request with the minimum seek time from the current head position.
- It is also known as Shortest Seek Distance First (SSDF), it is easier to compute distance.
- It **reduces** the **total seek time** as compared to FCFS.
- It allows the head to move to the closest track in the service queue.

SSTF Disk Scheduling Algorithm



SSTF Disk Scheduling Algorithm

Number of cylinders moved by the R/W head

=(65-53)+(67-65)+(67-37)+(37-14)+(98-14)+(122-98)+(124-122)

+(183-124)

= 12+2+30+23+84+24+2+59 = **236 cylinders**

OR

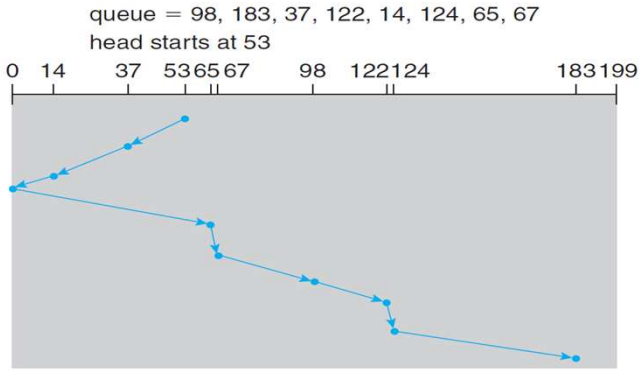
=(67-53)+(67-14)+(183-14)

= 236 cylinders

Scan Disk Scheduling Algorithm

- It is also called as **Elevator Algorithm.**
- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.
- It moves in both directions until end.
- Tend to say more at the ends so more fair to the extreme cylinder request.

Scan Disk Scheduling Algorithm



Scan Disk Scheduling Algorithm

Number of cylinders moved by the head

=(53-0)+(183-0)

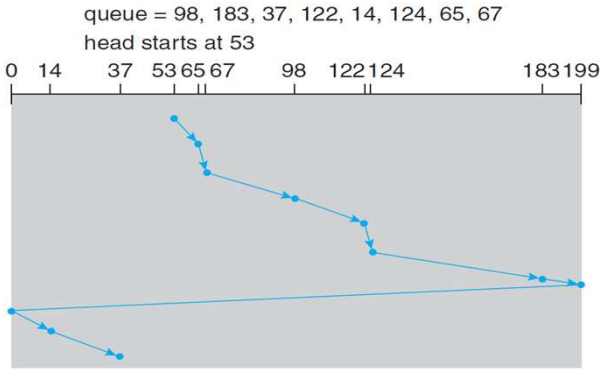
= 53+183

= **236 cylinders**

C-Scan Disk Scheduling Algorithm

- **Circular SCAN scheduling** algorithm, the arm of the disk moves in a particular direction **servicing requests** until it reaches the last cylinder,
- Then, it jumps to the last cylinder of the opposite direction **without servicing** any request then it turns back and start moving in that direction servicing the remaining requests.

C-Scan Disk Scheduling Algorithm



C-Scan Disk Scheduling Algorithm

Number of cylinders moved by the R/W head

=(199-53)+(199-0)+(37-0)

= 146+199+37

= **382 cylinders**

Look Disk Scheduling Algorithm

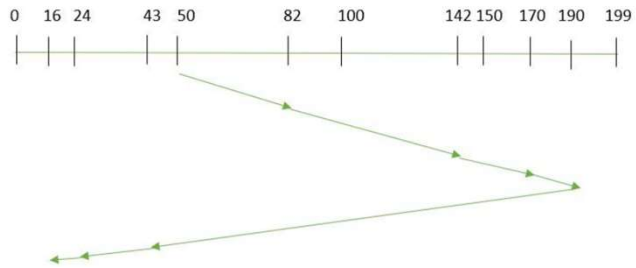
- Look disk scheduling work same as scan algorithm but instead of going till last track, the **R/W head go till last request** and then change the direction.
- **The arm only goes as far as the last request in each direction.**

Look Disk Scheduling Algorithm

- LOOK Algorithm is similar to the SCAN disk scheduling algorithm except for the difference that the disk arm in spite of going to the end of the disk goes only to the last request to be serviced in front of the head and then reverses its direction from there only.
- Thus it prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

Example of Look Disk Scheduling Algorithm

Queue: 82,170,43,140,24,16,190, the Read/Write head position is at 50, the disk arm should move **"towards the larger value".**



Example of Look Disk Scheduling Algorithm

Number of cylinders moved by the R/W head

= (190-50) + (190-16)

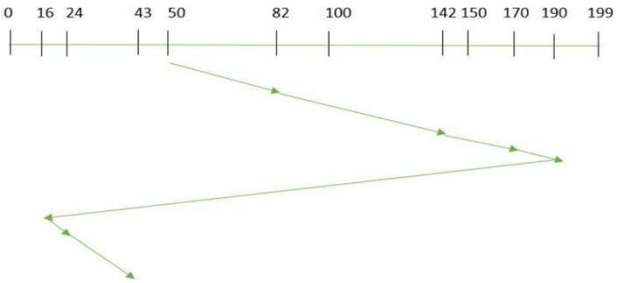
= 314 cylinders

C-Look Disk Scheduling Algorithm

- **C-Look takes the advantage of both C-Scan and Look algorithm.**
- The arm only goes as far as the last request (not till last track) in each direction, then reverses direction immediately, without service any request.
- Thus, it also prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

Example of C-Look Disk Scheduling Algorithm

Queue: 82,170,43,140,24,16,190, the Read/Write head position is at 50, the disk arm should move **"towards the larger value".**



Example of C-Look Disk Scheduling Algorithm

Number of cylinders moved by the R/W head

= (190-50) + (190-16) + (43-16)

= 341 cylinders

Elevator Algorithm

- Algorithms based on the common elevator principle.
- Four combinations of Elevator algorithms:

.Service in both directions or in only one direction –
.Go until last cylinder or until last I/O request –

Direction \ Go until	Go until the last cylinder	Go until the last request
Service both directions	**Scan**	**Look**
Service in only one direction	**C-Scan**	**C-Look**

Numerical

Important Formula

Disk Access time = Seek time + Rotational latency + data transfer time

OR

Total Transfer Time = Seek time + Average Rotational latency + data transfer time

Transfer Time = (File size/track size) * time taken to complete one rotation

Disk capacity = surfaces * tracks * sectors * bytes per sector

Consider a hard disk with 4 surfaces, 64 tracks, 128 sectors, 256 bytes per sector.

Ques 1: What is the capacity of the hard disk?

Solution:

Disk capacity = surfaces * tracks * sectors * bytes per sector

Disk capacity = 4 * 64 * 128 * 256

= 8388608 bytes = 8388608/1024 = 8192 KB

= 8192/1024

= 8 MB

Consider a hard disk with 4 surfaces, 64 tracks, 128 sectors, 256 bytes/sector.

Ques 2: The disk is rotating at 3600 RPM, what is the data transfer rate?

Solution: 60 sec = 3600 rotations, 1 sec = 60 rotations

Data transfer rate = no. of rotations per second * sector capacity * no. of surfaces (since 1 R-W head is used for each surface)

Data transfer rate = 60 * 128 * 256 * 4

Data transfer rate = 7.5 MB/sec

Consider a hard disk with 4 surfaces, 64 tracks, 128 sectors, 256 bytes/sector.

Ques 3: The disk is rotating at 3600 RPM, what is the average access time?

Solution: As seek time, controller time and the amount of data to be transferred is not given in the question, we assume all three as 0.

Average Access time = Average rotational latency (delay)
Rotational latency per 60 sec = 3600 rotations, then 1 sec = 60 rotations
Rotational latency = (1/60) sec = 16.67 msec.
Average Rotational latency = (16.67)/2
= 8.33 msec.
Average Access time = 8.33 msec.

Thank You

Operating System Structure

Faculty Name:
Dr. Premnarayan Arya
Assistant Professor
Department of CEA, GLA University, Mathura

Contents

File System

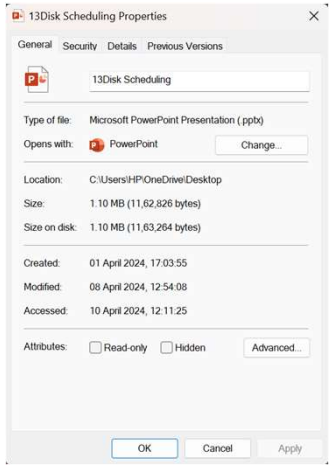
- Computers can **store information** on various storage media, such as magnetic disks, magnetic tapes, optical disks etc.
- A **text file** is a sequence of characters organized into lines (and possibly pages).
- A **source file** is a sequence of functions, each of which is further organized as declarations followed by executable statements.
- An **executable file** is a series of code sections that the loader can bring into memory and execute.

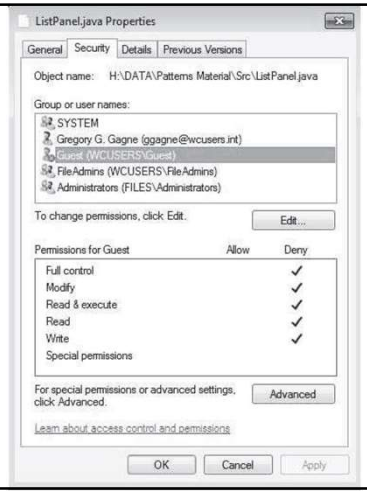
File Attributes

- **Name**: The file name is the only information kept in human readable form.
- **Identifier**: This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
- **Type**: This information is needed for systems that support different types of files.
- **Location**: This information is a pointer to a device and to the location of the file on that device.

File Attributes cont…

- **Size**: The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
- **Protection**: Access-control information determines who can do reading, writing, executing, and so on.
- **Time, date, and user identification**: This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.





File Operations

- **Creating a file:** Two steps are necessary to create a file. First, space in the file system must be found for the file. Second, an entry for the new file must be made in the directory.
- **Writing a file:** To write a file, we make a **system call** specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the file's location. The system must keep a **write pointer** to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.

File Operations cont…

- **Reading a file:** To read from a file, we use a system call that specifies the name of the file and where (in memory) the next block of the file should be put. Again, the directory is searched for the associated entry, and the system needs to keep a **read pointer** to the location in the file where the next read is to take place.
- **Repositioning within a file:** The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value. Repositioning within a file need not involve any actual I/O. This file operation is also known as a file **seek**.

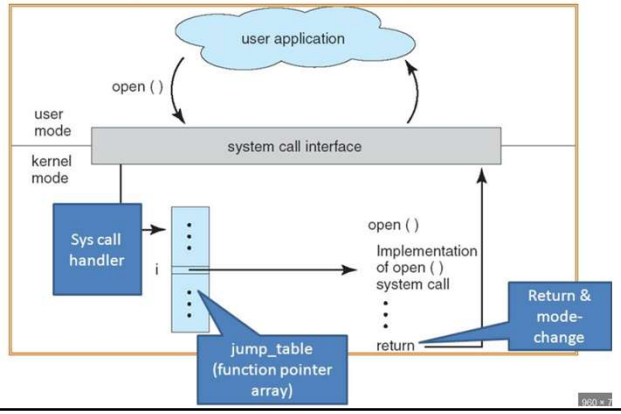
File Operations cont…

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

System Call

- A system call is **an interface** between a program running in user space and the operating system (OS).
- A system call is **a request from computer software to an operating system's kernel**.
- The Application Program Interface (API) connects the operating system's functions to user programs.
- It acts as a link between the operating system and a process, allowing user-level programs to request operating system services.

System Call



UNIX SYSTEM CALLS	DESCRIPTION	WINDOWS API CALLS	DESCRIPTION
Process Control			
`fork()`	Create a new process.	`CreateProcess()`	Create a new process.
`exit()`	Terminate the current process.	`ExitProcess()`	Terminate the current process.
`wait()`	Make a process wait until its child processes terminate.	`WaitForSingleObject()`	Wait for a process or thread to terminate.
`exec()`	Execute a new program in a process.	`CreateProcess()` or `ShellExecute()`	Execute a new program in a new process.
`getpid()`	Get the unique process ID.	`GetCurrentProcessId()`	Get the unique process ID.

File Management			
`open()`	Open a file (or device).	`CreateFile()`	Open or create a file or device.
`close()`	Close an open file (or device).	`CloseHandle()`	Close an open object handle.
`read()`	Read from a file (or device).	`ReadFile()`	Read data from a file or input device.
`write()`	Write to a file (or device).	`WriteFile()`	Write data to a file or output device.
`lseek()`	Change the read/write location in a file.	`SetFilePointer()`	Set the position of the file pointer.
`unlink()`	Delete a file.	`DeleteFile()`	Delete an existing file.
`rename()`	Rename a file.	`MoveFile()`	Move or rename a file.

Directory Management			
`mkdir()`	Create a new directory.	`CreateDirectory()`	Create a new directory.
`rmdir()`	Remove a directory.	`RemoveDirectory()`	Remove an existing directory.
`chdir()`	Change the current directory.	`SetCurrentDirectory()`	Change the current directory.
`stat()`	Get file status.	`GetFileAttributesEx()`	Get extended file attributes.
`fstat()`	Get status of an open file.	`GetFileInformationByHandle()`	Get file information using a file handle.
`link()`	Create a link to a file.	`CreateHardLink()`	Create a hard link to an existing file.
`symlink()`	Get the status of an open file.	`CreateSymbolicLink()`	Create a symbolic link.

Device Management			
`brk()` or `sbrk()`	Increase/decrease the program's data space.	`VirtualAlloc()` or `VirtualFree()`	Reserve, commit, or free a region of memory.
`mmap()`	Map files or devices into memory.	`MapViewOfFile()`	Map a file into the application's address space.

Information Maintenance			
`time()`	Get the current time.	`GetSystemTime()`	Get the current system time.
`alarm()`	Get the status of an open file.	`SetWaitableTimer()`	Set a timer object.
`getuid()`	Set an alarm clock for the delivery of a signal.	`GetUserName()` or `LookupAccountName()`	Get the username or ID.
`getgid()`	Get the group ID.	`GetTokenInformation()`	Get the group information of a security token.

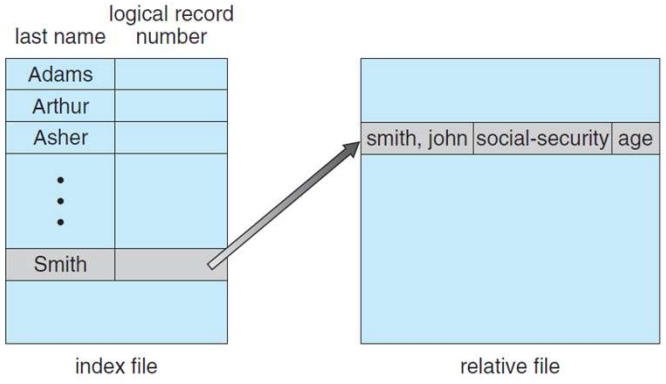
Communication Calls			
`socket()`	Create a new socket.	`socket()`	Create a new socket.
`bind()`	Bind a socket to a network address.	`bind()`	Bind a socket to a network address.
`listen()`	Bind a socket to a network address.	`listen()`	Listen for connections on a socket.
`accept()`	Accept a new connection on a socket.	`accept()`	Accept a new connection on a socket.
`connect()`	Initiate a connection on a socket.	`connect()`	Initiate a connection on a socket.
`send()` or `recv()`	Send and receive data on a socket.	`send()` or `recv()`	Send and receive data on a socket.

Security and Access Control			
`chmod()` or `umask()`	Change the permissions/mode of a file.	`SetFileAttributes()` or `SetSecurityInfo()`	Change the file attributes or security info.
`chown()`	Change the owner and group of a file.	`SetSecurityInfo()`	Set the security information.

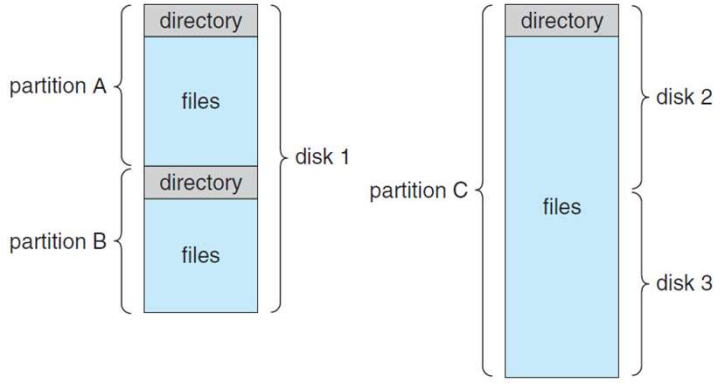
File Types

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

Example of index and relative files



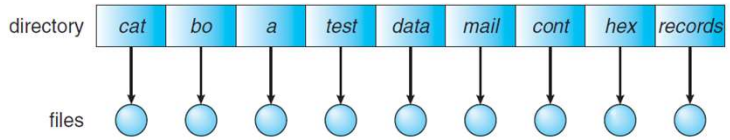
A typical file-system organization



Directory

Single-Level Directory

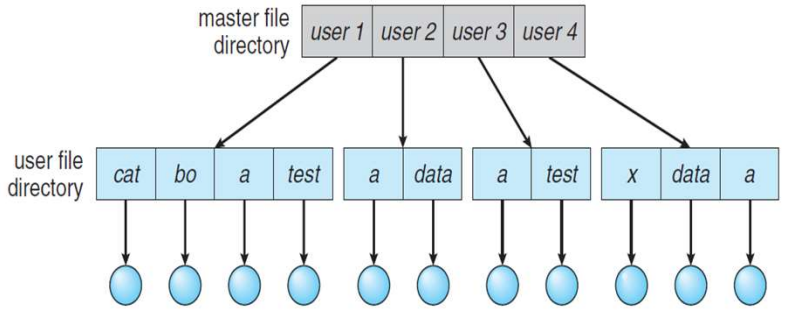
- The simplest directory structure is the single-level directory. All files are contained in the same directory. It has major limitations, when the number of files increases or when the system has more than one user. Since all files are in the same directory, they must have unique names.



Two-Level Directory

- As we have seen, a single-level directory often leads to **confusion** of file names among different users.
- The standard solution is to create a separate directory for each user. In the two-level directory structure, each user has his own **user file directory** (**UFD**).
- The UFDs have similar structures, but each lists only the files of a single user. When a user job starts or a user logs in, the system's **master file directory** (**MFD**) is searched.

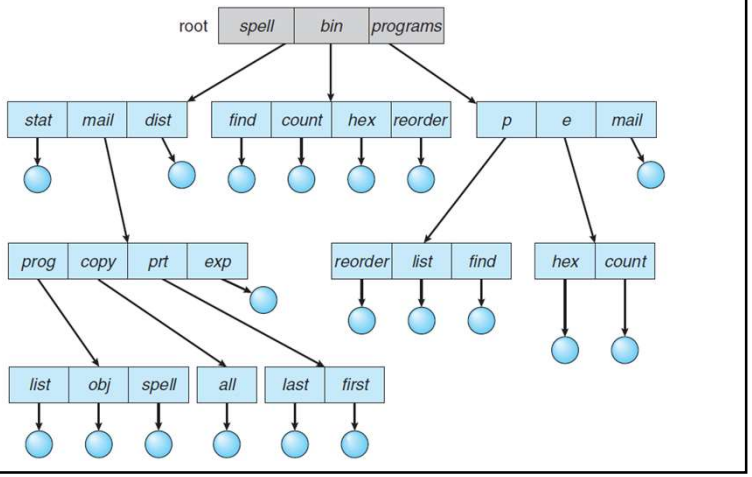
Two-Level Directory



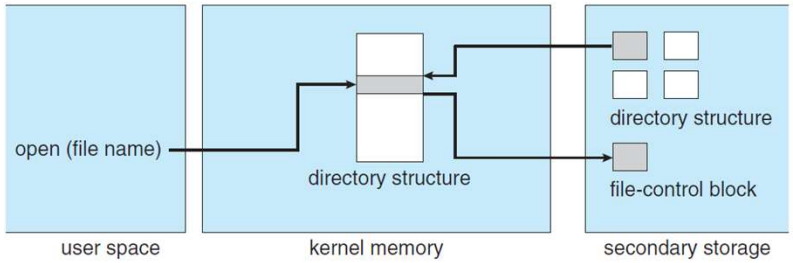
Feature	Single-Level Directory	Two-Level Directory
User Isolation	No user-specific directories. All users share the same directory space.	Each user has their own private directory.
Organization	All files are stored in one directory, making it less organized.	Files can be organized under user-specific directories, allowing for better file management.
Search Efficiency	Can be less efficient as all files are in a single directory, requiring more time to find a specific file.	More efficient due to fewer files in each user-specific directory.
Access Control	Hard to implement user-specific access controls because all files reside in the same directory.	Easier to implement user-specific access controls, enhancing security.
Complexity	Simpler to implement but can become cluttered and difficult to manage with many files.	Slightly more complex due to the need for user management, but offers better organization.

Tree-Level Directory

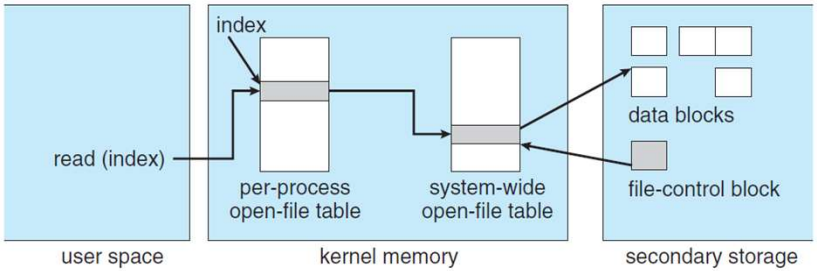
- We have seen how to view a two-level directory as a two-level tree, the natural **generalization** is to extend the directory structure to a tree of arbitrary height.
- This **generalization** allows users to create their own **subdirectories** and to organize their files accordingly.
- A **tree** is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name.



In-memory file-system structures: (a) File open



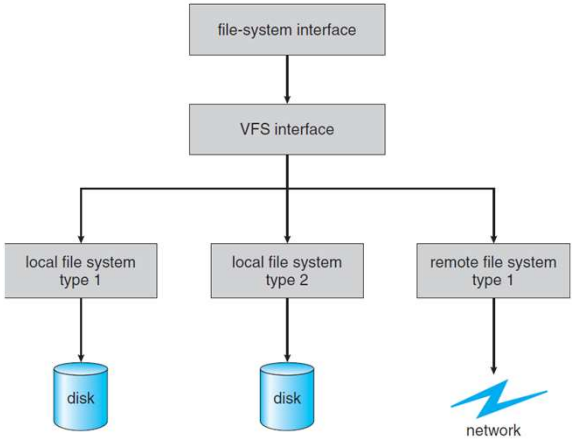
In-memory file-system structures: (b) File read.



Virtual File Systems

The **four main object types** defined by the Linux VFS architecture are:

- The **inode object**, which represents an individual file.
- The **file object**, which represents an open file.
- The **superblock object**, which represents an entire file system.
- The **dentry object**, which represents an individual directory entry.



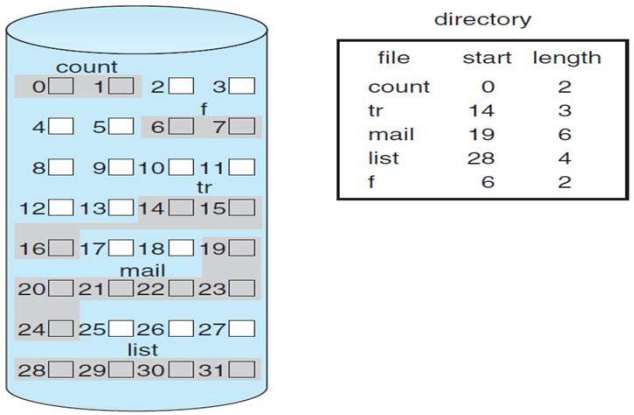
File Allocation Methods

Allocation Methods

1. Contiguous Allocation
2. Linked Allocation
3. Indexed Allocation

Contiguous Allocation

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



directory

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

Fig: Contiguous allocation of disk space

Linked Allocation

- **Linked allocation** solves all problems of contiguous allocation. With linked allocation, each file is a linked list of disk blocks.
- The disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file.
- **For example**, a file of five blocks might start at block 9 and continue at block 16, then block 1, then block 10, and finally block 25. Each block contains a pointer to the next block.
- These pointers are not made available to the user. Thus, if each block is 512 bytes in size, and a disk address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes.

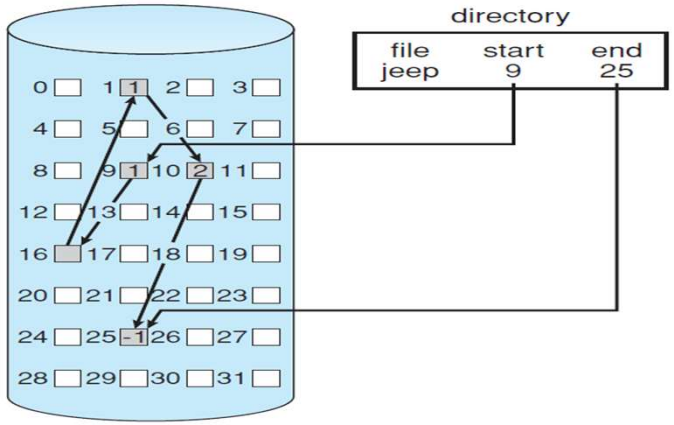


Fig: Linked allocation of disk space

Indexed Allocation

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

Indexed Allocation

- Each file has its own index block, which is an array of disk-block addresses. The ***ith* entry** in the index block points to the ***ith* block** of the file. The directory contains the address of the index block. To find and read the ***ith* block**, we use the pointer in the ***ith* index-block** entry. This scheme is similar to the paging scheme. When the file is created, all pointers in the index block are set to null. When the ***ith* block** is first written, a block is obtained from the free-space manager, and its address is put in the ***i*th index-block** entry.

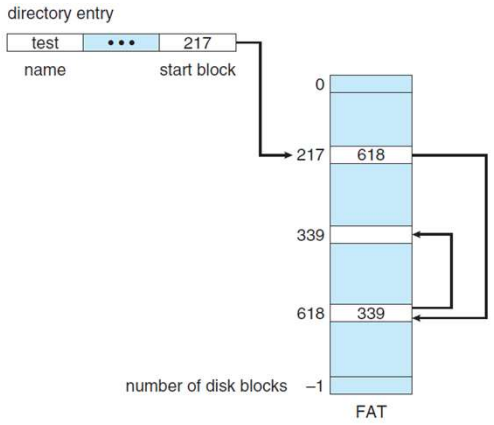


Fig: File allocation table

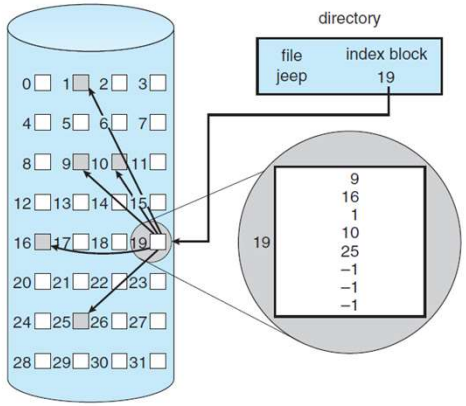
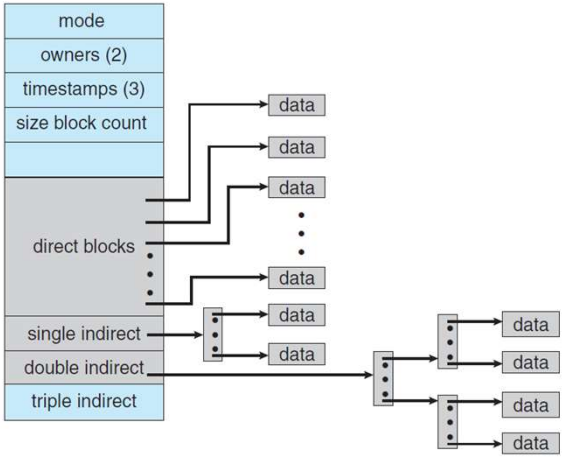


Fig: Indexed allocation of disk space

Multilevel index

- A variant of linked representation uses a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks.
- To access a block, the operating system uses the first-level index to find a second-level index block and then uses that block to find the desired data block.
- This approach could be continued to a third or fourth level, depending on the desired maximum file size.



Free-Space Management

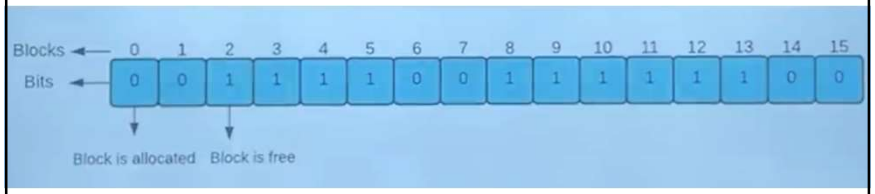
- Since disk space is limited, we need to reuse the space from deleted files for new files, if possible. To keep track of free disk space, the system maintains a **free-space list**. The free-space list records all free disk blocks-those not allocated to some file or directory.
- To create a file, we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

Bit Vector

- Frequently, the free-space list is implemented as a **bit map** or **bit vector**. Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0.
- For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bit map would be

  001111001111110001100000011100000 ...
- The main advantage of this approach is its relative simplicity and its efficiency in finding the first free block or *n* consecutive free blocks on the disk.

Bit Vector



Bit Vector

- One technique for finding the first free block on a system that uses a bit-vector to allocate disk space is to sequentially check each word in the bit map to see whether that value is not 0, since a 0-valued word contains only 0 bits and represents a set of allocated blocks. The first non-0 word is scanned for the first 1 bit, which is the location of the first free block.
- **The calculation of the block number is (number of bits per word) × (number of 0-value words) + offset of first 1 bit.**

Bit Vector

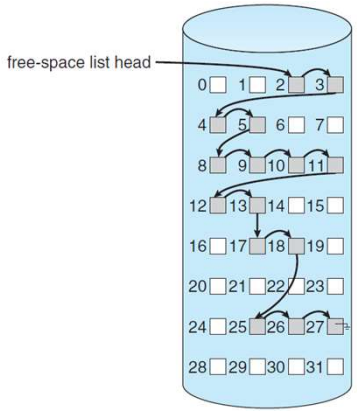
- Unfortunately, bit vectors are inefficient unless the entire vector is kept in main memory. Keeping it in main memory is possible for smaller disks but not necessarily for larger ones.
- A 1.3-GB disk with 512-byte blocks would need a bit map of over 332 KB to track its free blocks.
- A 1-TB disk with 4-KB blocks requires 256 MB to store its bit map. Given that disk size constantly increases, the problem with bit vectors will continue to escalate as well.

Linked List

- Another approach to free-space management is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory.
- This first block contains a pointer to the next free disk block, and so on. In earlier example the blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 were free and the rest of the blocks were allocated.

Linked List

- In this situation, we would keep a pointer to block 2 as the first free block.
- Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, and so on. This scheme is not efficient; to traverse the list, we must read each block, which requires substantial I/O time.



Linked free-space list on disk

Thanks