LINK:
https://whimsical.com/operating-system-cheatsheet-by-love-babbar-S9tuWBCSQfzoBRF5EDNinQ

A. Introduction to OS

What is OS ?

- Interface between user and hardware.
- Responsible for the execution of all the processes, Resource Allocation, CPU management, File Management and many other tasks.

What does an Operating system do?

1. Process Management
2. Process Synchronization
3. Memory Management
4. CPU Scheduling
5. File Management
6. Security

Types of OS ?

1. **Batch OS –** A set of similar jobs are stored in the main memory for execution. A job gets assigned to the CPU, only when the execution of the previous job completes.
   Disadvantage: Starvation
2. **Multiprogramming OS –** Multi-programming increases CPU utilisation by organising jobs (code and data) so that the CPU always has one to execute. The idea is to keep multiple jobs in main memory. If one job gets occupied with IO, CPU can be assigned to other job.
3. **Multitasking OS -** Multitasking is the ability of an OS to execute more than one task simultaneously on a CPU machine. These

multiple tasks share common resources (like CPU and memory). In multi-tasking systems, the CPU executes multiple jobs by switching among them typically using a small time quantum, and the switches occur so quickly that the users feel like interact with each executing task at the same time

Sr.no	**Multiprogramming**	**Multi-tasking**
1.	Both of these concepts are for single CPU.	Both of these concepts are for single CPU.
2.	Concept of Context Switching is used.	Concept of Context Switching and Time Sharing is used.
3.	In multiprogrammed system, the operating system simply switches to, and executes, another job when current job needs to wait.	The processor is typically used in time sharing mode. Switching happens when either allowed time expires or where there other reason for current process needs to wait (example process needs to do IO).
4.	Multi-programming increases CPU utilization by organising jobs .	In multi-tasking also increases CPU utilization, it also increases responsiveness.
5.	The idea is to reduce the CPU idle time for as long as possible.	The idea is to further extend the CPU Utilization concept by increasing responsiveness Time Sharing.

4. **Time Sharing OS -** Time sharing OS allows the user to perform more than one task at a time, each task getting the same amount of time to execute. Hence, the name time sharing OS. Multiple jobs are running at the CPU time and also, they use the CPU simultaneously.
5. **Real Time OS -** Tasks ought to be accomplished inside a definite deadline. The time period in operation systems not solely need correct results however conjointly the timely results, which implies

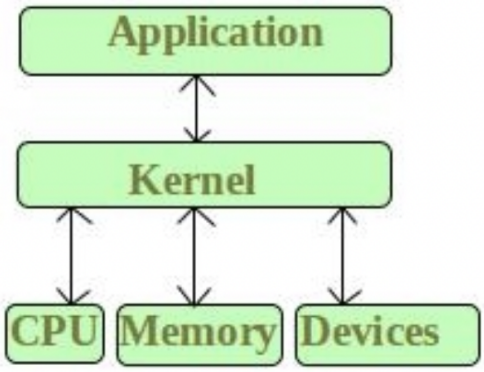
beside the correctness of the results it should be created in an exceedingly sure deadline otherwise the system can fail.

S.NO	Time Sharing Operating System	Real-Time Operating System
1.	In time sharing operating system, quick response is emphasized for a request.	While in real time operating system, computation tasks are emphasized before its nominative point.
2.	In this operating system **Switching** method/function is available.	While in this operating system **Switching** method/function is not available.
3.	In this operating system any modification in the program can be possible.	While in this modification does not take place.
4.	In this OS, computer resources are shared to the external.	But in this OS, computer resources are not shared to the external.
5.	It deals with more than processes or applications simultaneously.	Whereas it deals with only one process or application at a time.
6.	In this OS, the response is provided to the user within a second.	While in real time OS, the response is provided to the user within time constraint.
7.	In time sharing system, high priority tasks can be preempted by lower priority tasks, making it impossible to guarantee a response time for your critical applications.	Real time operating systems, give users the ability to prioritize tasks so that the most critical task can always take control of the process when needed.

KERNEL

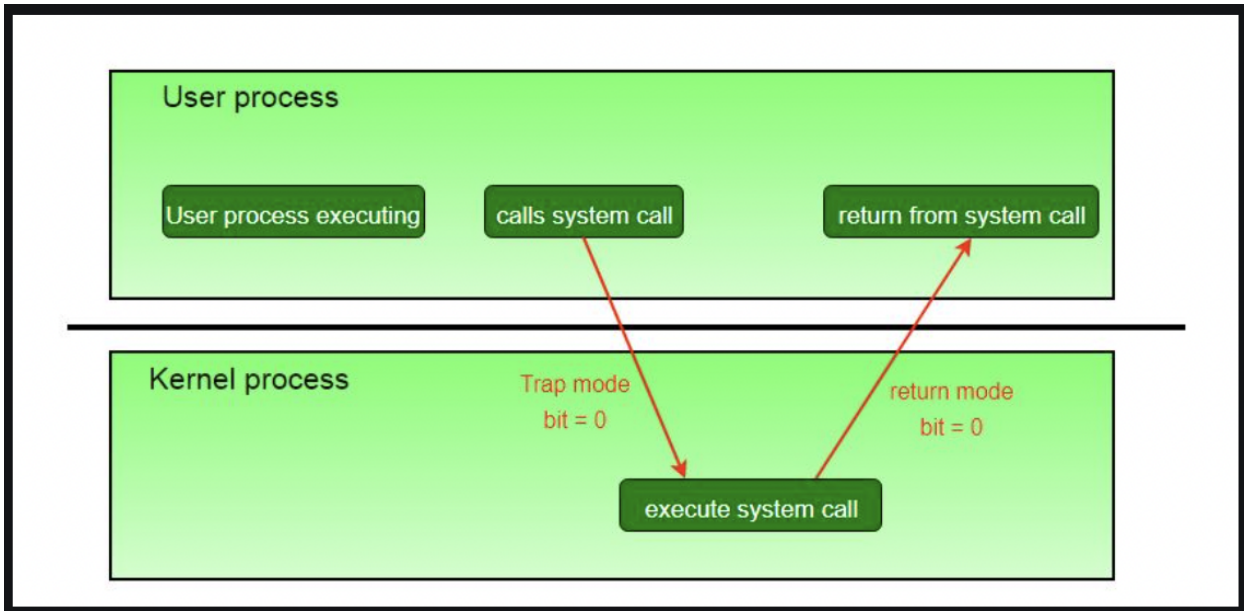
- Acts as a bridge between the software and hardware of the computer.
- Important part of an OS that manages system resources.

- It is one of the first program which is loaded on start-up after the bootloader.
- Also responsible for offering secure access to the machine's hardware for various programs.



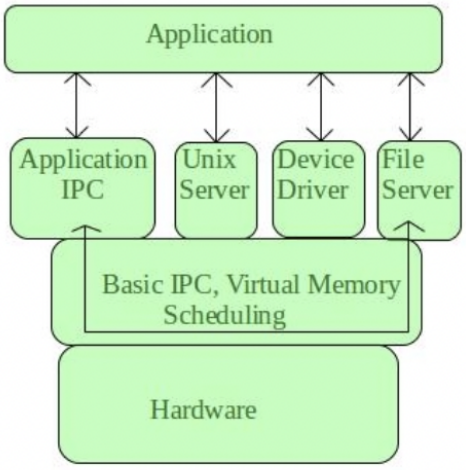
User Mode and Kernel Mode

The CPU can execute certain instructions only when it is in kernel mode. These instructions are called **privilege instruction**



MICROKERNEL

- The user services and kernel services are implemented in different address spaces. The user services are kept in user address space, and kernel services are kept under kernel address space
- Reduces the size of kernel and size of an operating system as well.



Microkernel Architecture –
In this architecture, only the most important services are inside the kernel and the rest of the OS services are present inside the system application program. The microkernel is solely responsible for the most important services of the operating system they are named as follows:

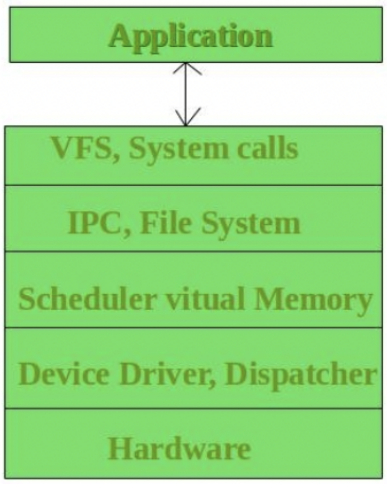
- Inter process-Communication
- Memory Management
- CPU-Scheduling

Advantages of Microkernel –

- The architecture of this kernel is small and isolated hence it can function better.
- Expansion of the system is easier, it is simply added to the system application without disturbing the kernel.

MONOLITHIC KERNEL

- Manages system resources between application and hardware
- User services and kernel services are implemented under the same address space.
- It increases the size of the kernel, thus increases the size of the operating system as well.



Monolithic Kernel

Advantages of Monolithic Kernel –

- One of the major advantages of having a monolithic kernel is that it provides CPU scheduling, memory management, file management, and other operating system functions through system calls.
- The other one is that it is a single large process running entirely in a single address space.

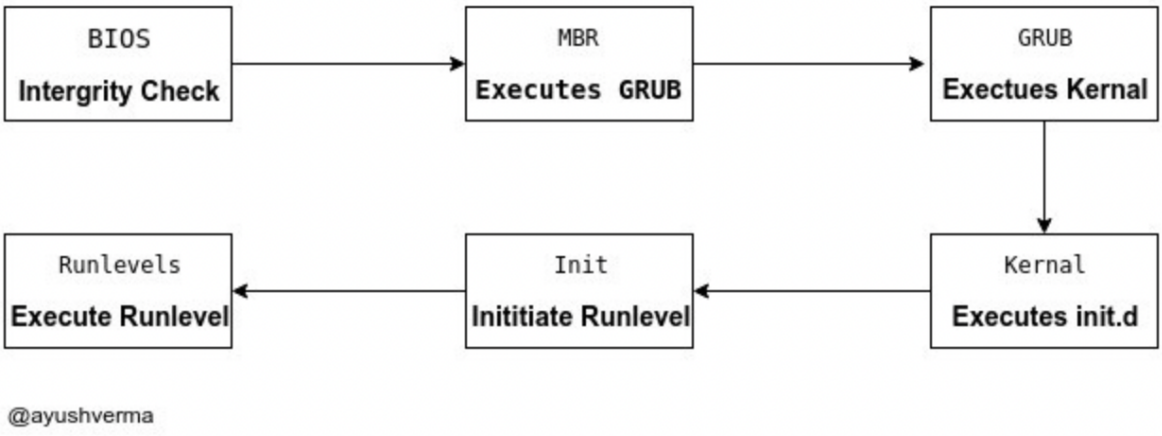
Disadvantages of Monolithic Kernel –

- One of the major disadvantages of a monolithic kernel is that if any one service fails it leads to an entire system failure.
- If the user has to add any new service. The user needs to modify the entire operating system.

Microkernel	Monolithic kernel
In microkernel user services and kernel, services are kept in separate address space.	In monolithic kernel, both user services and kernel services are kept in the same address space.
OS is complex to design.	OS is easy to design and implement.
Microkernel are smaller in size.	Monolithic kernel is larger than microkernel.
Easier to add new functionalities.	Difficult to add new functionalities.
To design a microkernel, more code is required.	Less code when compared to microkernel
Failure of one component does not effect the working of micro kernel.	Failure of one component in monolithic kernel leads to failure of entire system.
Execution speed is low.	Execution speed is high.
It is easy to extend Microkernel.	It is not easy to extend monolithic kernel.
Example : Mac OS X.	**Example :** Microsoft Windows 95.

BOOTING

- what happens in the background from the time you press the Power button until the Linux login prompt appears? - LeetCode Discuss
- What Exactly happens when you press the power button on PC ? | by Ayush Verma | Medium
-



- Booting is the process of loading and starting of an Operating System
- The BIOS (stored in the ROM) initiates the booting process by doing two things
    - Performs a series of activities or functionality tests on programs stored in ROM, called on POST (Power-on Self Test) that checks to see whether peripherals in the system are in perfect order or not.
    - Load the Master Boot Record (MBR) from a device which is stored in the first sector of the hard disk.
- MBR identifies how and where the system's operating system (OS) is located in order to be booted (loaded) into the computer's main storage or random access memory (RAM).
- MBR contains the OS Boot Loader also called the Bootstrap Loader
    - Typically 2 stage boot loading is used
- Boot Loader loads the full OS into RAM
    - The OS starts running
- OS performs its own initialization
    - OS starts a shell that displays login prompt

RAM

- It is also called read-write memory or the main memory or the primary memory.
- The programs and data that the CPU requires during the execution of a program are stored in this memory.

- It is a volatile memory as the data is lost when the power is turned off.
- Types - SRAM, DRAM

ROM

- Stores crucial information essential to operate the system, like the program essential to boot the computer.
- It is not volatile.
- Always retains its data.
- Used in calculators and peripheral devices.
- Types - PROM, EPROM, EEPROM

RAM	ROM
1. Temporary Storage.	1. Permanent storage.
2. Store data in MBs.	2. Store data in GBs.
3. Volatile.	3. Non-volatile.
4.Used in normal operations.	4. Used for startup process of computer.
5. Writing data is faster.	5. Writing data is slower.

32 BIT V/S 64 BIT OS

[Difference between 32-bit and 64-bit operating systems - GeeksforGeeks]

B. Process Management

Process

- A process is a program in execution.
- A **program counter (PC)** is a CPU register in the computer processor which has the address of the next instruction to be executed from memory.
- Each process is represented by a Process Control Block (PCB).
- When a program is loaded as a process it is allocated a section of virtual memory which forms its usable address space.

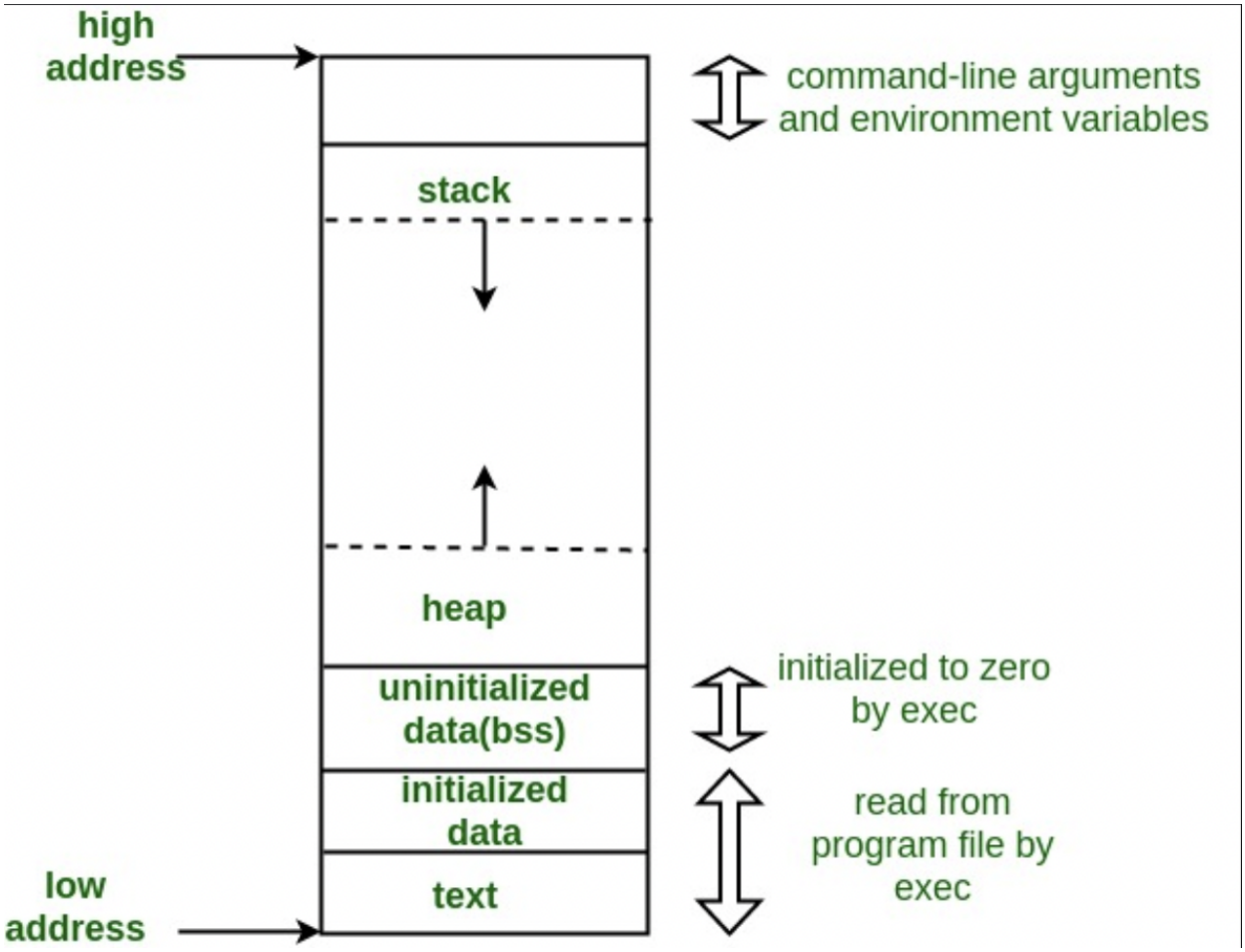
Memory Map of Process:

User Space:

- **Code/Text**
  - The program instructions to be executed
  - A text segment may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it.
- **Data**
  - Contains the global variable.
- **Stack**
  - Store state in case of function calls, memory allocation of arrays
  - LIFO structure
  - Typically located in the higher parts of memory
- **Heap**
  - dynamic memory allocation usually takes place.
  - managed by malloc, realloc, and free

Kernel Space

- **State**
- **Kernel Stack**



IMPORTANT POINTS

- A process will commonly have at least two LIFO stacks, including a user stack for user mode and a kernel stack for kernel mode.
- Kernel Maintains some information(current state of process) about each one of the processes.
- State (info about different registers and their content) of the process is not stored in user space because of risks of being tampered by running error or bad code.
- Every process has an identifier (PID).

Process Control Block

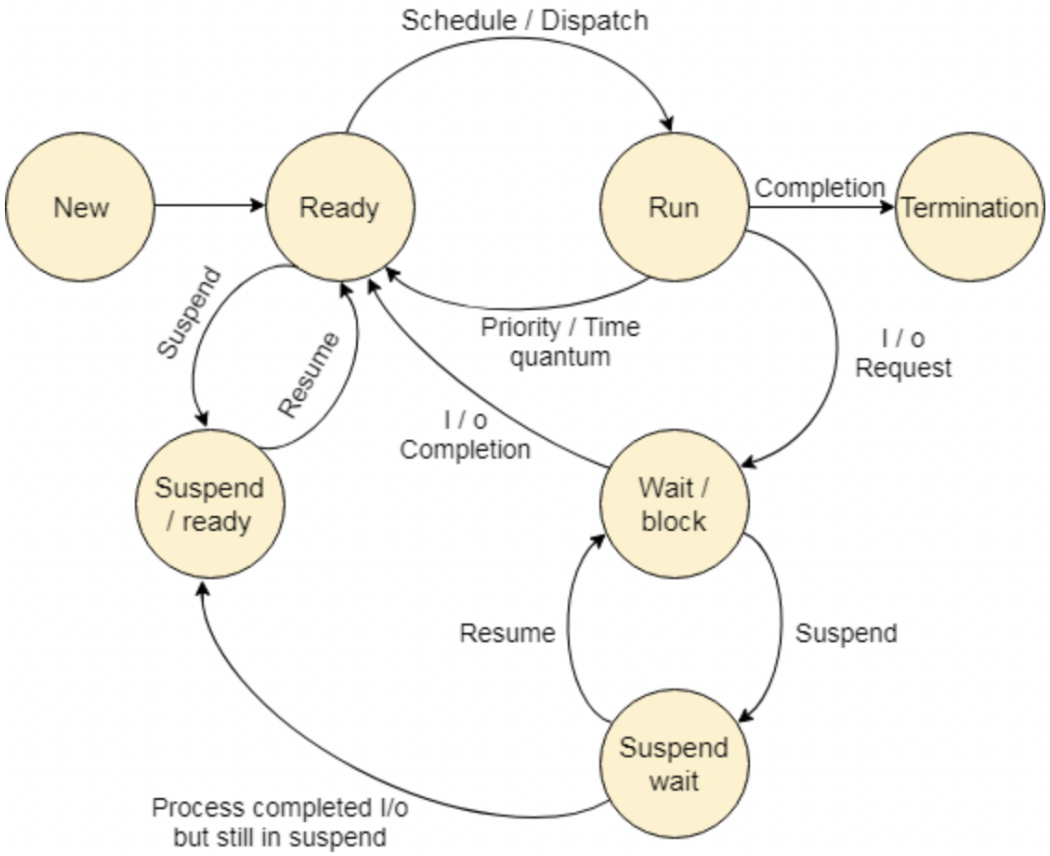
- Kept in a memory area that is protected from the normal user access.

- Some of the operating systems place the PCB at the beginning of the kernel stack for the process as it is a safe location
- It contains Information about:
    - Process state
    - Values of program counter and other registers
    - CPU scheduling information - priority, pointer to scheduling queue, etc.
    - Accounting information - process id, CPU- and real- time used, time limits, etc.
    - I/O status information - list of i/o devices allocated, list of open files, etc.

Program vs Process

Sr.No.	Program	Process
1.	Program contains a set of instructions designed to complete a specific task.	Process is an instance of an executing program.
2.	Program is a passive entity as it resides in the secondary memory.	Process is a active entity as it is created during execution and loaded into the main memory.
3.	Program exists at a single place and continues to exist until it is deleted.	Process exists for a limited span of time as it gets terminated after the completion of task.
4.	Program is a static entity.	Process is a dynamic entity.
5.	Program does not have any resource requirement, it only requires memory space for storing the instructions.	Process has a high resource requirement, it needs resources like CPU, memory address, I/O during its lifetime.
6.	Program does not have any control block.	Process has its own control block called Process Control Block.
7.	Program has two logical components: code and data.	In addition to program data, a process also requires additional information required for the management and execution.
8.	Program does not change itself.	Many processes may execute a single program. There program code may be the same but program data may be different. these are never same

Process State Diagram



- **Process Scheduling -** selecting one process for execution out of all the ready processes.
- The objective of **multiprogramming** is to have some process running at all times so as to maximize CPU utilization.
- The objective of **multitasking** is to switch the CPU among the processes so frequently that the user can interact with each process while it is running.
- The ready queue is implemented as a linked list of PCBs with a header containing pointers to the first and the last PCBs.

Long term Scheduler

- Selects processes from those submitted by the user and loads them into the memory.
- Controls the degree of multiprogramming which is represented by the number of processes in the memory.

- Long-term scheduler should select a proper mix of CPU-bound processes and i/o-bound processes.
- Invoked less frequently.
- Acts from New state to Ready state

Short term Scheduler

- Selects one of the processes in the memory and allocates the CPU to it.
- Invoked frequently and should be very fast.
- From Ready to running State

Medium term Scheduler

- Removes processes from the main memory and from the competition for the CPU
- Reducing the degree of multiprogramming
- From Wait to suspended wait or from ready to suspended ready

Operation on Processes

- Creation
- Scheduling
- Execution
- Deletion

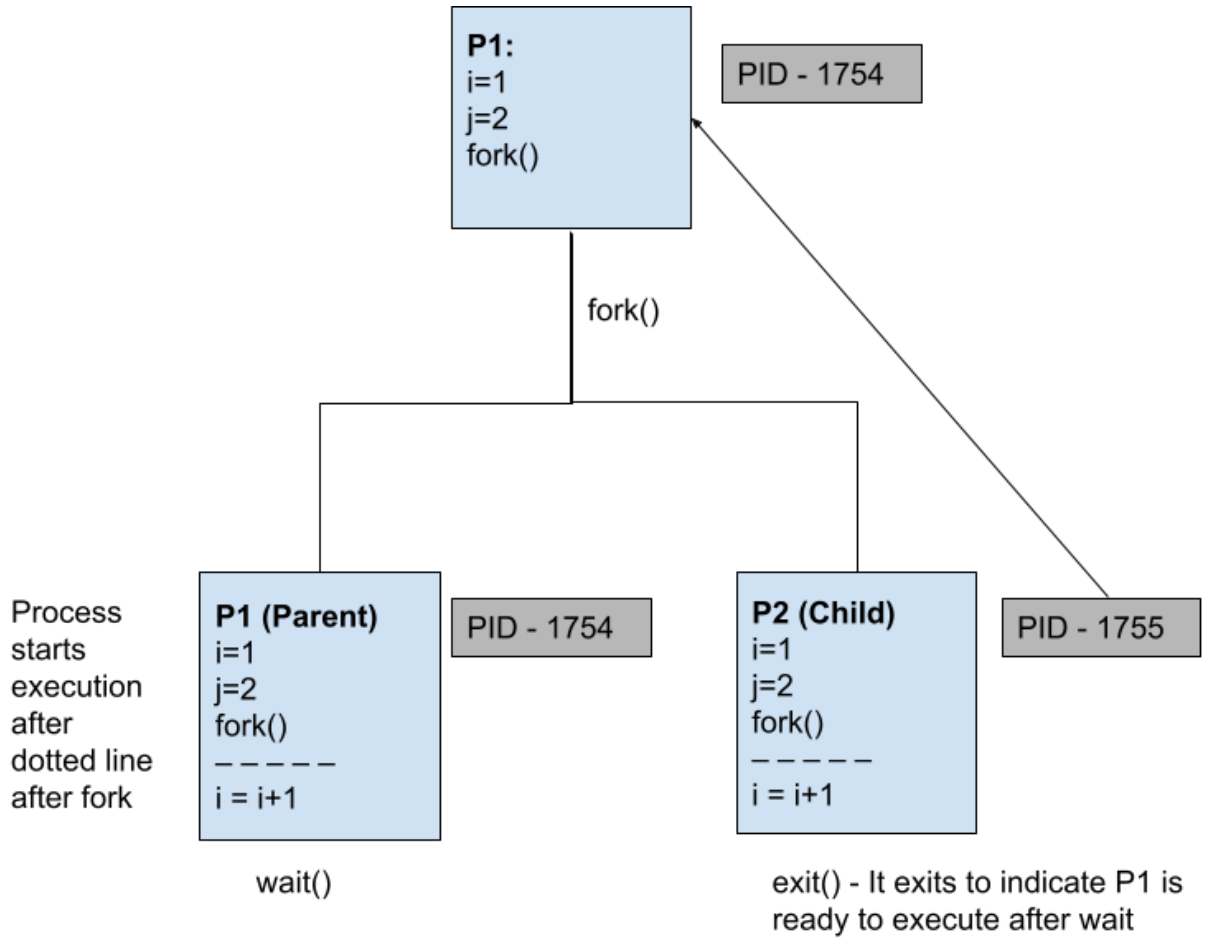
Context Switching

- Switch between processes.
- Switching the CPU to another process requires saving the state of the current process and reloading the state of another process.
- States are saved into and reloaded from PCBs.
- Pure overhead

Process Creation

- One process creating another process.
- Processes are called parent process and child process, respectively.
- Each process has a unique id PID
- Process may obtain resources either from its parent or from the operating system directly
- Process may be a duplicate of its parent process (same code and data) or may have a new program loaded into it.

- Process are direct or indirect descendent of init process
- Every process is initialized by another process
- Every process are separate entity i.e. They have different virtual map of each
- The kernel is the first process to be created, is the ancestor of all other processes and is at the root of the process tree.



- When a process forks itself it creates a new process is
    - If fork is successful
        - Returns PID of child to the parent process
        - Returns value 0 to child process
    - Is fork is unsuccessful
        - -1 is returned to the parent process.
        - A child process is not created.
- If we want child process to complete before parent we use wait system call.

- Child and parent process run in a time shared manner

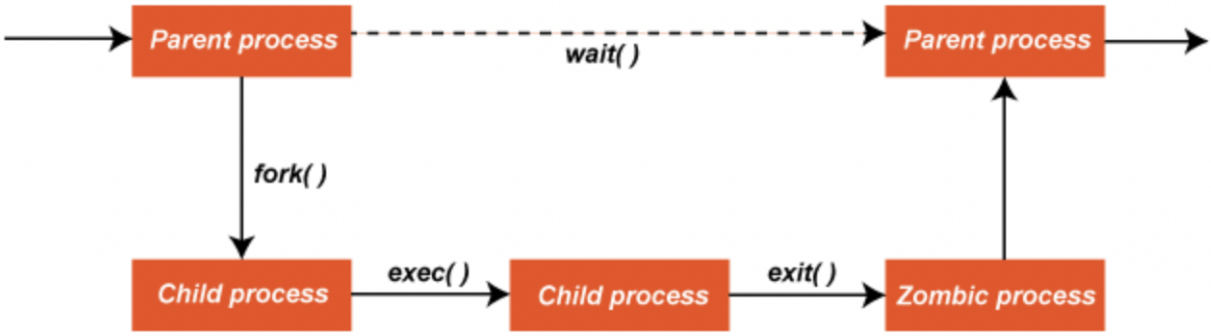
```
int main()
{
  int pid;
  pid = fork();

  if (pid == 0)
  {
     // CHILD PROCESS
    printf ("Child : I am the child process\n");
    printf ("Child : Child's PID: %d\n", getpid());
    printf ("Child : Parent's PID: %d\n", getppid());

     // Indicated the child process is complete
     exit(0);
  }
  else
  {
     // PARENT PROCESS
    printf ("Parent : I am the parent process\n");
    printf ("Parent : Parent's PID: %d\n", getpid());
    printf ("Parent : Child's PID: %d\n", pid);

     // Commands the parent to wait until all the child process are
complete
     wait();
  }
}
```

```
Child : I am the child process
Child : Child's PID: 4706
Child : Parent's PID: 4705
Parent : I am the Parent process
Parent : Parent's PID: 4705
Parent : Child's PID: 4706
```

Process Termination + Orphan/Zombie Process

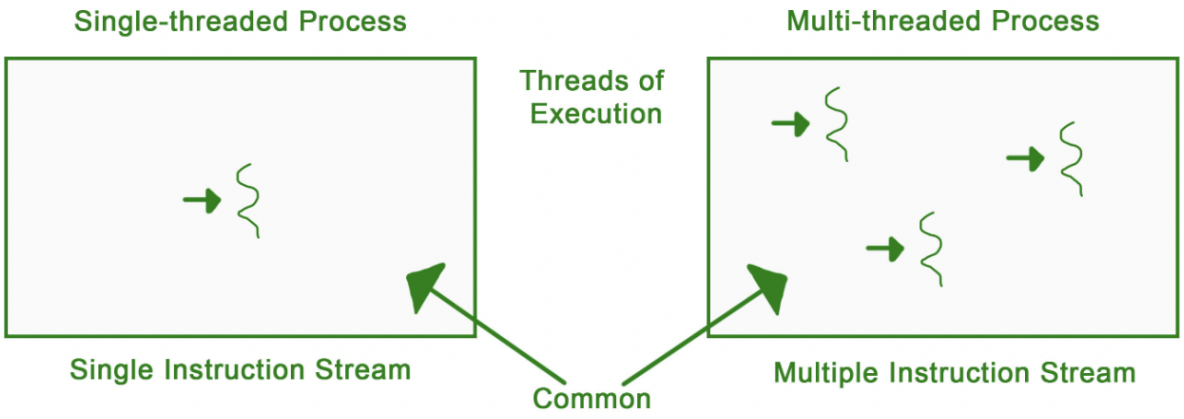
- Marks the deletion of the PCB of the process.
- **Orphan Process**

    - A process whose parent process no longer exists i.e. either finished or terminated without waiting for its child process to terminate is called an orphan process.
    - Kernel allocates init as the parent of child
- **Zombie Process**
    - Child terminates without sending signal to parent and parent remains in waiting state.
    - A process which has finished the execution but still has entry in the process table to report to its parent process is known as a zombie process
    - Once the exit status is read via the wait system call, the zombie's entry is removed from the process table and said to be "reaped".
    - A child process always first becomes a zombie before being removed from the process table.



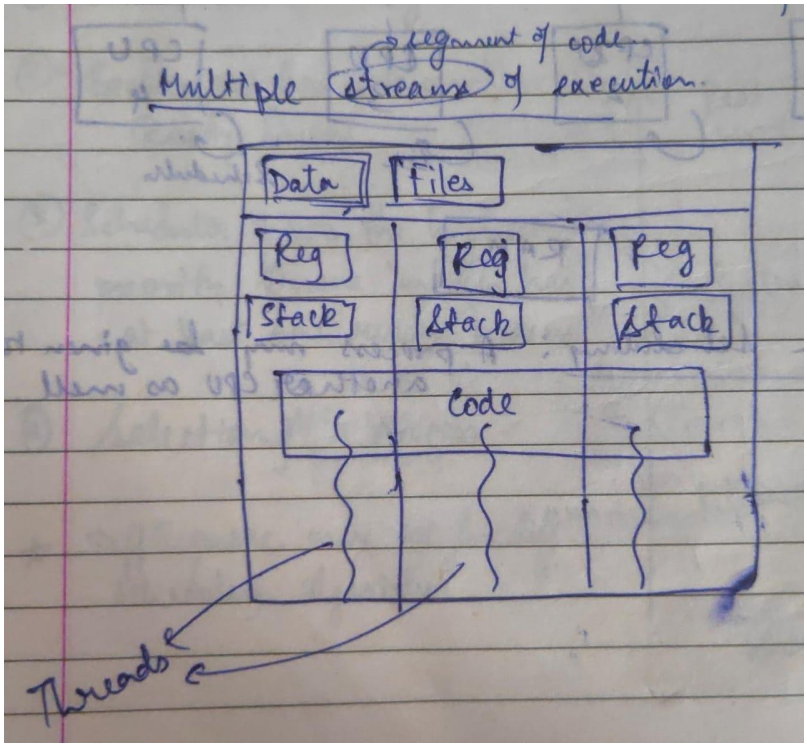
Threads

- Component of a process.
- Smallest sequence of instructions that can be managed independently by a scheduler.
- Multiple threads can exist within the same process, executing concurrently and share resources such as memory.
- The child process has a copy of the address space of the parent, but it does not inherit any of its threads.
- Threads are not independent of one another like processes are, and as a result threads share with other threads their code section, data section, and OS resources (like open files and signals). But, like process, a thread has its own program counter (PC), register set, and stack space.



Single Thread and Multi Thread Process

Below image is an example of multi-threaded process



Types of thread

- User level thread
    - Managed by user level libraries
    - Faster context switches
    - If one user level thread blocks then entire process blocks because kernel has no idea about the multiple threads being used in the process.

Ex - p_thread library is used to create thread by user application

- Kernel Level thread
    - Managed by OS system calls
    - Slower context switches since it involves OS system call
    - If one kernel level thread blocks no effect to others since all the threads are being controlled by kernel and it has all the information

Process vs Thread

Process	Thread
A process is an instance of a program that is being executed or processed.	Thread is a segment of a process or a lightweight process that is managed by the scheduler independently.
Processes are independent of each other and hence don't share a memory or other resources.	Threads are interdependent and share memory.
Each process is treated as a new process by the operating system.	The operating system takes all the user-level threads as a single process.
If one process gets blocked by the operating system, then the other process can continue the execution.	If any user-level thread gets blocked, all of its peer threads also get blocked because OS takes all of them as a single process.
Context switching between two processes takes much time as they are heavy compared to thread.	Context switching between the threads is fast because they are very lightweight.
The data segment and code segment of each process are independent of the other.	Threads share data segment and code segment with their peer threads; hence are the same for other threads also.
The operating system takes more time to terminate a process.	Threads can be terminated in very little time.
New process creation is more time taking as each new process takes all the resources.	A thread needs less time for creation.

Multi-threading advantages

- Responsiveness
- faster execution
- better resource utilization
- easy communication
- parallelization

Non-preemptive Scheduling

- A process keeps the CPU until it terminates or switches to the waiting state.
- Ex - FCFS, SJF, LJF, Multilevel Queue

Preemptive Scheduling

- A process can be forced to leave the CPU and switch to the ready queue.
- Ex - SRJF, LRJF, Round Robin, Priority Based

Dispatcher

- Module of the operating system that gives control of the CPU to the process selected by the CPU scheduler.
- A dispatcher performs various tasks, including
    - Context switching
    - Setting up user registers
    - Memory mapping

When dispatching, the process changes from the ready state to the running state.

- The time consumed by the Dispatcher is known as dispatch latency.
- Dispatch latency is a pure overhead.

Scheduling Criteria

- ↑ **CPU utilization**
- ↑ **Throughput -** number of processes completed per unit time
- ↓ **Turnaround time -** time from submission to completion (time spent in different queues + time spent in CPU + time spent in different i/o

devices)
- ↓ **Waiting time -** time spent in ready queue (only)
- ↓ **Response time -** time from submission to first response

FCFS (First Come First Serve)

- **Advantages:**
  - Simple and Easy to implement
- **Disadvantages:**
  - Higher Average Waiting Time => Poor Performance
  - Can lead to Convoy Effect or Starvation (short process stuck behind long process) => under utilization of processor & OS

SJF (Shortest Job First)

- **Advantages**:
  - Maximum throughput
  - Minimum average waiting and turnaround time
- **Disadvantages:**
  - Difficult to predict Burst Time hence not implementable
  - May lead to starvation

SRTF (Shortest Remaining Time First)

- **Advantages:**
  - It makes processing of jobs faster
- **Disadvantages:**
  - More context Switches than SJF and consumes CPU time

Round Robin

- **Advantages**:
  - Fair allocation of CPU to all jobs
  - Does not lead to convoy or starvation
- **Disadvantages:**
  - Deciding perfect Time Quantum is difficult
  - Higher Time Quantum => Higher response time => becomes FCFS
  - Lower Time Quantum => Higher context switching => lower CPU efficiency

Priority Scheduling

- CPU is allocated to the process with the highest priority
- Scheduling can be Preemptive and Non-preemptive.
- Priority assigned can be Static or Dynamic.
- **Aging:** gradually increase the priority of a process waiting for a long time
- **Priority inversion:** a low-priority process gets the priority of a high-priority process waiting for it

Multilevel Queue

- Ready Queue is divided into separate queues for each class of processes.
- Processes are permanently assigned to a queue on entry to the system and processes are not allowed to move between queues.
- **Advantages**
    - The processes are permanently assigned to the queue, so it has advantage of low scheduling overhead.
- **Disadvantages**
    - Some processes may starve for CPU if some higher priority queues are never becoming empty.

Multilevel Feedback Queue

- **Advantages**:
    - It is more flexible.
    - It allows different processes to move between different queues.
    - It prevents starvation by moving a process that waits too long for the lower priority queue to the higher priority queue.
- **Disadvantages**
    - It produces more CPU overheads.
    - It is the most complex algorithm.

Comparison Between all Algorithms

Algorithm	Allocation is	Complexity	Average waiting time (AWT)	Preemption	Starvation	Performance

FCFS	According to the arrival time of the processes, the CPU is allocated.	Not complex	Large.	No	No	Slow performance
SJF	Based on the lowest CPU burst time (BT).	More complex than FCFS	Smaller than FCFS	No	Yes	Minimum Average Waiting Time
LJFS	Based on the highest CPU burst time (BT)	More complex than FCFS	Depending on some measures e.g., arrival time, process size, etc.	No	Yes	Big turn-around time
LRTF	Same as LJFS the allocation of the CPU is based on the highest CPU burst time (BT). But it is preemptive	More complex than FCFS	Depending on some measures e.g., arrival time, process size, etc.	Yes	Yes	The preference is given to the longer jobs
SRTF	Same as SJF the allocation of the CPU is based on the lowest CPU burst time (BT). But it is preemptive.	More complex than FCFS	Depending on some measures e.g., arrival time, process size, etc	Yes	Yes	The preference is given to the short jobs
RR	According to the order of the process arrives with fixed time quantum (TQ)	The complexity depends on TQ size	Large as compared to SJF and Priority scheduling.	No	No	Each process has given a fairly fixed time

Priority Pre-emptive	According to the priority. The bigger priority task executes first	This type is less complex	Smaller than FCFS	Yes	Yes	Well performance but contain a starvation problem
Priority non-preemptive	According to the priority. with monitoring the new incoming higher priority jobs	This type is less complex than Priority preemptive	preemptive Smaller than FCFS	No	Yes	Most beneficial with batch systems
MLQ	According to the process that resides in the bigger queue priority	More complex than the priority scheduling algorithms	Smaller than FCFS	No	Yes	Good performance but contain a starvation problem
MFLQ	According to the process of a bigger priority queue.	It is the most Complex but its complexity rate depends on the TQ size	Smaller than all scheduling types in many cases	No	No	Good performance

C. Process Synchronization

Race Condition

- Several process **access a shared data/data structure** and outcome depends on the order of execution of processes.
- To avoid we need to ensure only one process is able to manipulate data at a given time using process synchronization.

P1 {
…
Count++; ⬅ Critical Section
…
}
	P2 {
	…
	Count - -;
	…
	}

R1 ← count ⬅ P1
– – – – – – –
R2 ← count ⬅ P2
R2 ← R2 - 1
Count ← R2
– – – – – – –
R1 ← R1 + 1 ⬅ P1
Count ← R1

Critical Section

- Section of code that try to access **shared resources**
- Can lead to Race Condition if multiple processes access that resource at the same time

Requirements

Mutual Exclusion

- No more than one process must be in critical section at a time

Progress

- When no process is in the critical section & some process requests to enter CS, it should be allowed immediately
- There are only a subset of processes that request for entry in to CS.
- The request to access CS is taken among the subset of processes that can access CS & should not be dictated by any other process not belonging to the subset

Bounded Wait (No Starvation)

- There is an upper bound on the number of times a process can access CS

Lock Mechanism

- Software based solution
- Locks are shares variable with two possible values 0/1
- Can be used for more than two processes.
- Lock = 0 implies critical section is vacant (initial value) and Lock = 1 implies critical section occupied.

```
P1 {
        ….
        1 while(lock != 0) ;   ⬅ Entry Section
        2 Lock = 1
```

```
        3 CS
        4 Lock = 0          ⬅ Exit Section
        ….
}
```

- **No Mutual Exclusion**

Lock = 1
P1 : 1 | 2, 3 |
P2 : 1, 2, 3 |

Test Set Lock Mechanism

- Hardware based solution
- It provides **Mutual exclusion and Progress** but **not bounded wait**
- It is based on **atomic instruction** => it cannot be stopped in between => context switch cannot happen in between

```
int test_ans_set(int *LOCK){
        Int prev = *LOCK;
        *LOCK = 1;
        Return prev;
}

P1 {
        …
        while(test_and_set(&LOCK) == 1) ;
        CS
        LOCK = 0;
        …
}
```

- It can lead to a state where each of the process neither executes nor completes in case of **Priority Inversion.**
  This is different from deadlock since they are not in blocked state. One is in ready state and the other is in running state, but neither of the two is being executed.
- **Spinlock** is a locking system mechanism. It allows a thread to acquire it to simply wait in loop until the lock is available i.e. a thread waits in a loop or spin until the lock is available. Spinlock is held for a short period of time.

Peterson Solution

- Software based solution
- Implemented at user mode
- Only for **2 processes**
- Has two shared variables

    - boolean flag[i] :Initialized to FALSE, initially no one is interested in entering the critical section
    - int turn : The process whose turn is to enter the critical section.
- A deadlock can never happen because the process which first sets the turn variable will enter in the critical section for sure.

Peterson's solution (2 processes only) –
Shared variables:
int turn;
Boolean flag[2];
Code for P_i:

```
do {   flag[i] = true;
       turn = j;
       while (flag[j] && turn == j)
         ;
       critical section
       flag[i] = FALSE;
       remainder section
    } while (true);
```

do {
…
flag[0] = true;
turn = 1;
while (flag[1] && turn == 1);
critical section
flag[0] = FALSE;
…
} while (true);
	do {
	…
	flag[1] = true;
	turn = 0;
	while (flag[0] && turn == 0);
	critical section
	flag[1] = FALSE;
	…
	} while (true);

Semaphore

- Integer variable that apart from initialization, is accessed only through two atomic operations called wait() and signal().
  Wait(S) – Signal(S)
  Down(S) – Up(S)
  P(S) – V(S)

COUNTING SEMAPHORE
- Initialised to N
- Value can go from -infinte to +infinite
- Uses a count that helps task to be acquired or released numerous times

<table>
<tr><td colspan="2">

- Represents the number of available units of resources
- Used to control access to a resource that has multiple instances

</td></tr>
<tr><td>

```
Down(Semaphore S){
   S.value - -;
   if(S.value < 0){
      Put process PCB in suspended list, sleep()
   }
   Else return ;
}
```

</td><td>

```
Up(Semaphore S){
   S.value++;
   if(S.value <= 0){
      // Putting process in ready queue
      Select a process and wake up()
   }
}

/Now a process from the suspend list
can come in
ready queue and again try to request
for the CS
```

</td></tr>
</table>

<table>
<tr><td colspan="2">

BINARY SEMAPHORE

- initialised to 1
- also known as mutex lock
- used to implement solution of critical section problem with multiple processes

</td></tr>
<tr><td>

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

</td><td>

```
Up(Semaphore S){
   if(Suspended list is empty) S.value = 1;
   Else {
      Select a process and wake up()
   }
}
```

</td></tr>
</table>

- If S=0 initially, none process will be able to access critical section and hence deadlock will occur
- **Counting Semaphore has no mutual exclusion whereas Binary Semaphore has Mutual exclusion.**
- One of the biggest limitations of a semaphore is priority inversion.
- **MORE INFO:** https://www.guru99.com/semaphore-in-operating-system.html#what-is-semaphore

Mutex

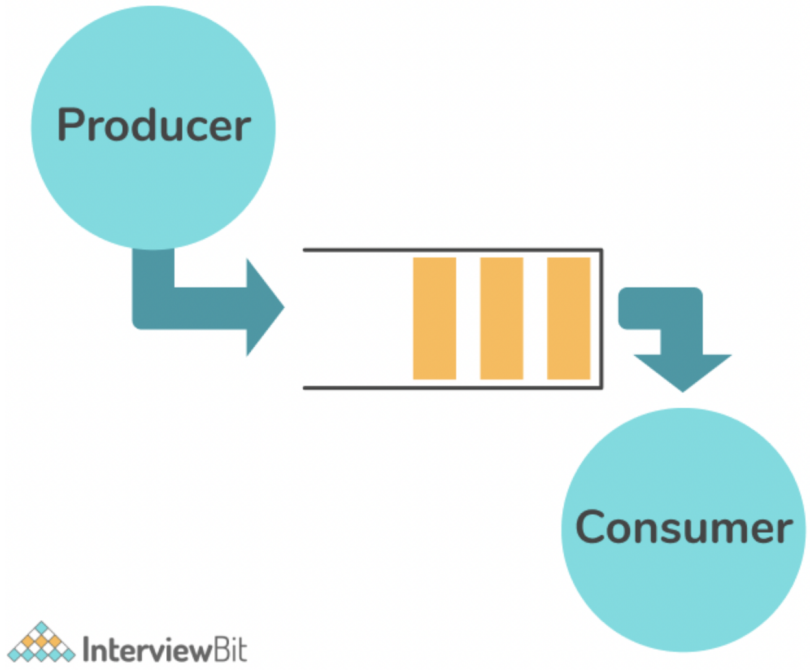
- Mutex is a mutual exclusion object that synchronizes access to a resource.
- The mutex locking mechanism ensures only one thread can acquire the mutex and enter the critical section
- This thread only releases the mutex when it exits in the critical section.

Semaphore vs Mutex

Terms	Mutex	Semaphore
Definition	The mutex is a locking mechanism, as to acquire a resource, a process needs to lock the mutex object, and while releasing a resource process has to unlock the mutex object.	Semaphore is a signalling mechanism as wait() and signal() operations performed on the semaphore variable indicate whether a process is acquiring or releasing the resource.
Existence	A mutex is an object.	Semaphore is an integer variable.
Function	Mutex allows multiple program threads to access a single resource but not simultaneously.	Semaphore allows multiple program threads to access a finite instance of resources.
Ownership	Mutex object lock is released only by the process that has acquired the lock on the mutex object.	Semaphore value can be changed by any process acquiring or releasing the resource by performing wait() and signal() operation.
Categorize	Mutex is not categorized further.	The semaphore can be categorized into counting semaphore and binary semaphore.
Operation	The mutex object is locked or unlocked by the process of requesting or releasing the resource.	Semaphore value is modified using wait() and signal() operation apart from initialization.
Resources Occupied	If a mutex object is already locked, then the process desiring to acquire resource waits and get queued by the system till the resource is released and the mutex object gets unlocked.	Suppose the process acquires all the resources, and no resource is free. In that case, the process desiring to acquire resource performs wait() operation on semaphore variable and blocks itself till the count of semaphore become greater than 0.

Producer-Consumer Problem

- Producer-Consumer Problem is also known as **bounded buffer problem.**



Points to note:

- The producer should produce data only when the buffer is not full. In case it is found that the buffer is full, the producer is not allowed to store any data into the memory buffer.
- Data can only be consumed by the consumer if and only if the memory buffer is not empty. In case it is found that the buffer is empty, the consumer is not allowed to use any data from the memory buffer.
- Accessing memory buffer should not be allowed to producer and consumer at the same time

Example:

- "in" used in a producer code represent the next empty buffer
- "out" used in consumer code represent first filled buffer
- count keeps the count number of elements in the buffer

```
Producer(){
        int itemp;
        while(true){
            produce_item(itemp);
            while(count == n) ;
            Buffer[in] = itemp;
            in = (in+1)%n;
            count = count+1; **
        }
}

**
```

```
Consumer(){
        int itemc;
        while(true){
            while(count == 0) ;
            itemc = Buffer[out];
            out = (out+1)%n;
            count = count-1; **
            process_item(itemc);
        }
}

**
```

```
Load Rp, m[count]
INCR Rp
- - - -
Store m[count], Rp
```

```
Load Rc, m[count]
DECR Rc
- - - -
Store m[count], Rc
```

- In above code Race condition will occur if interrupt occurs at dotted line

Solution using Semaphore

Three semaphores are used:

1. full - count number of full slots (Counting Semaphore)
2. empty - count number of empty slots (Counting Semaphore)
3. Mutex - guard the buffer (Binary Semaphore)

```
#define N 10
typedef int semaphore
semaphore mutex = 1;
semaphore empty=N, full=0;
```

```
void producer(){
   int item;
   while(1){
       item = produce_item();
       P(empty);
       P(mutex);
       insert_item();
       V(mutex);
       V(full); // sending wakeup()
   }
}
```

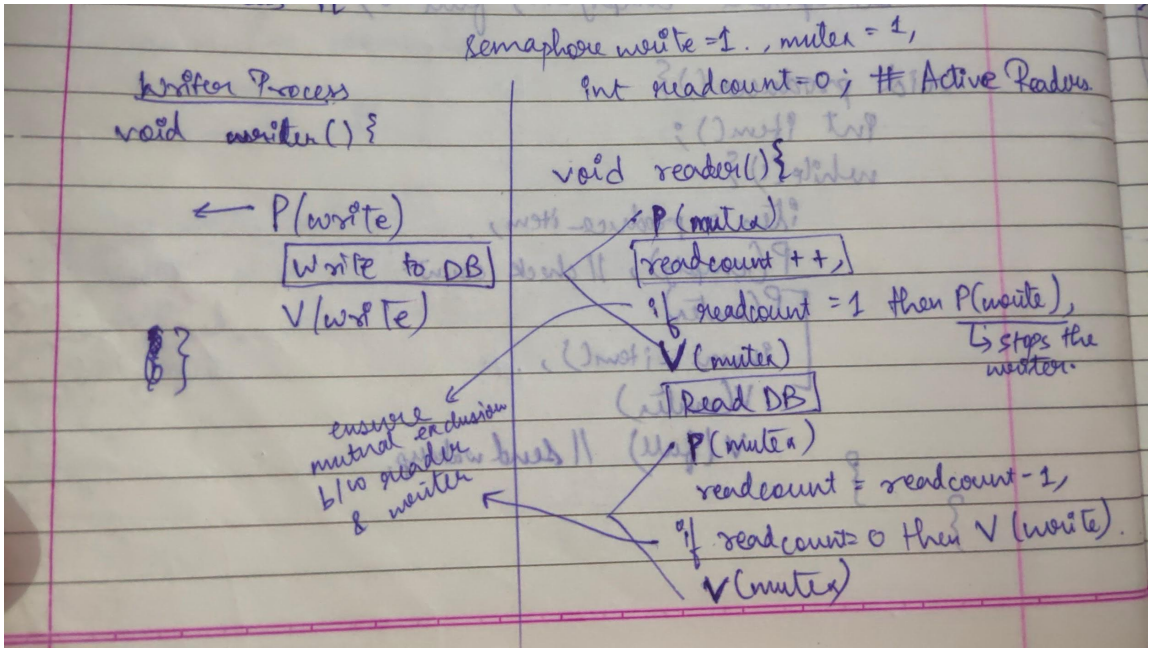
```
void consumer(){
   int item;
   while(1){
       P(full);
       P(mutex);
       item = consume_item();
       V(mutex);
       V(empty);
       consume_item();
   }
}
```

Reader-Writer Problem

- Model to access database

Conditions:

- Multiple reader can access a database at a time
- A writer cannot access the database if at least there is one reader
- If a writer is accessing a database no reader or writer is allowed to access it

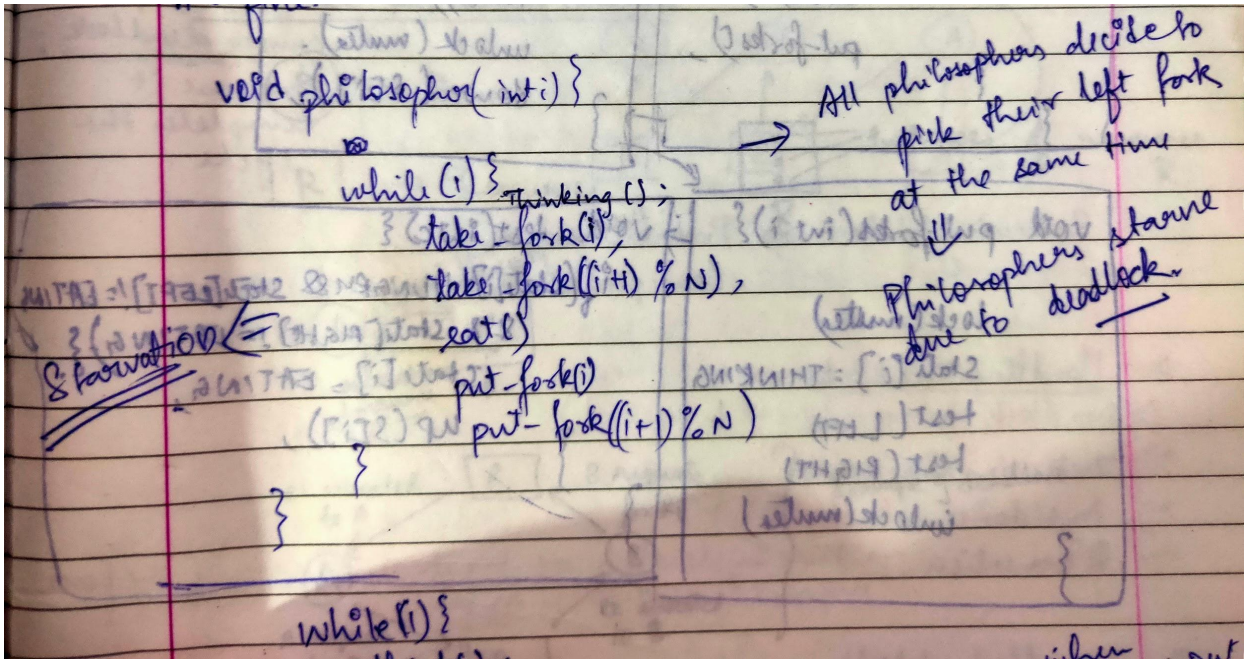


- If writer access the DB first then it will Down() the value of write from 1 to 0. Now if a reader comes them readcount will increase to 1 and in the if statement of reader the reader will go to suspended list because of P().

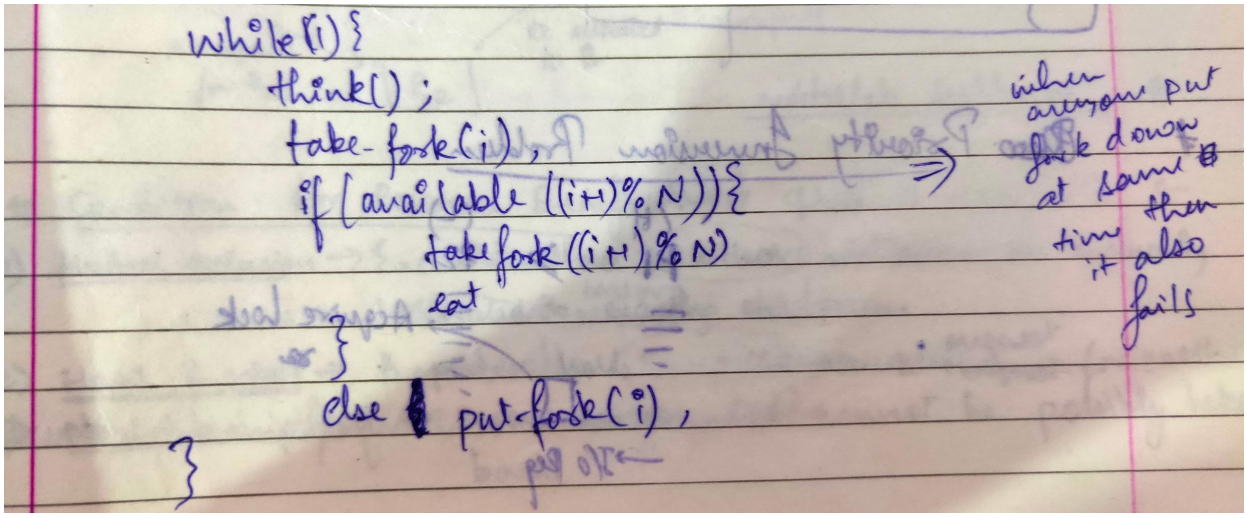
Dining Philosophers Problem

- K philosophers seated around a circular table with one chopstick between each pair of philosophers.
- A philosopher may eat if he can pick up the two chopsticks adjacent to him.
- One chopstick may be picked up by any one of its adjacent followers but not both.
- Each philosopher can either Think() or Eat()

Implementation 1:

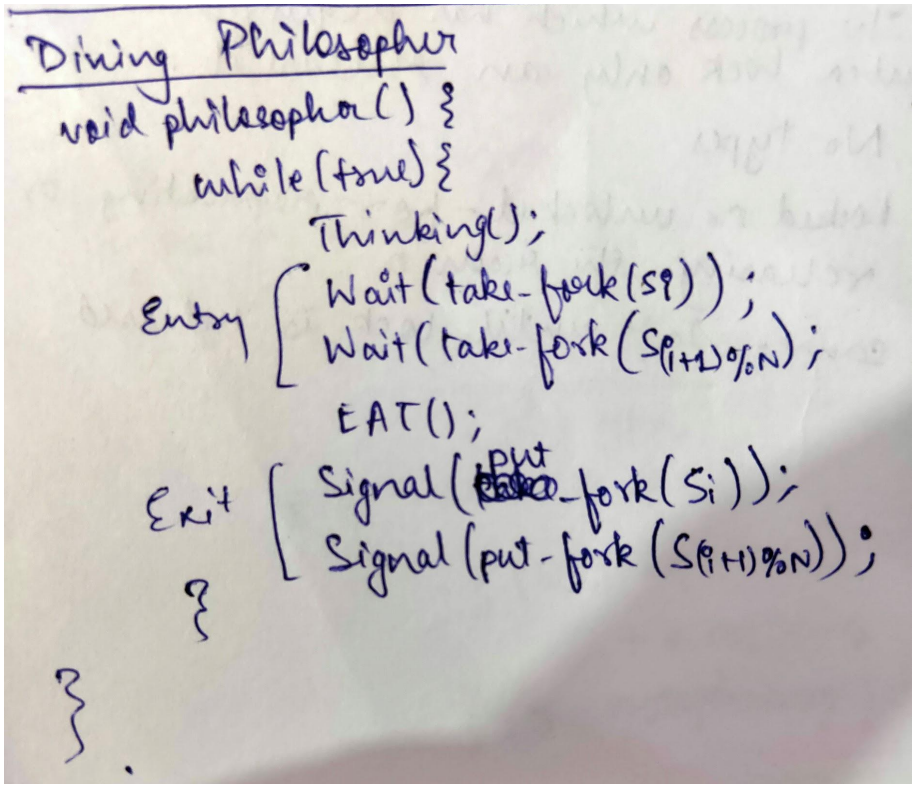


Implementation 2:



Solution using Semaphores

- Use N-Semaphore initialised to 0



- If all decide to take the left fork first then deadlock or starvation might occur. It can be avoided by
    - Allow only N-1 philosophers to pick the left fork first and Nth pick right fork
    - Alternate choice of first chopstick

D. Deadlock

- Deadlock is a situation in which two or more processes are waiting indefinitely because the resources they have requested for are being held by one another.

Conditions for deadlock

- Mutual exclusion - Each resource is either available or currently allocated to exactly one process
- Hold and wait - A process is holding some resources and waiting for other resources
- No preemption - Resources granted to a process cannot be preempted
- Circular wait - Chain of two or more process each waiting for resource held by another process

Methods for handling deadlocks

- ## Deadlock Ignorance (Ostrich Method)
  - Deadlocks are infrequent and handling is expensive
- ## Deadlock Detection and Recovery
- ## Deadlock Prevention
  1. Mutual exclusion
     - Mutual exclusion can be prevented using sharable resources, like read-only files.
  2. Hold and wait
     - Ensure that when a process requests for resources it is not holding some other resources.
     - Release current resources before requesting other resources.
     - Disadvantages – low resource utilization, starvation of processes requiring several resources.
  3. No preemption
     - If a process requests for some resources and they cannot be allocated right now, then the resources that the process is holding are preempted.
  4. Circular wait
     - Arrange the resource types as R1, R2, R3 ... Rm.
     - Protocol 1: Request resources in increasing order.
     - Protocol 2: If a process requests for Ri, then it must release Rj for all j>i.
     - All instances of a resource type should be allocated together.
     - Disadvantages – low resource utilization, reduced throughput.

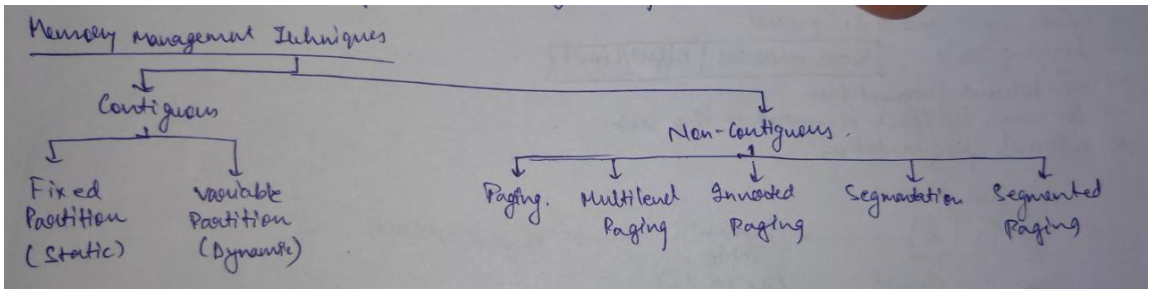
- ## Deadlock Avoidance (Banker Algorithm)
  - **Safe State -** A state if there is some scheduling order in which every process can run to completion even if all of them suddenly requests their maximum number of resources
  - **Unsafe state** could lead to deadlock
  - https://www.youtube.com/watch?v=7gMLNiEz3nw

Notes

- If there is a cycle, then there **may be** a deadlock (necessary but not sufficient condition).
- If there is no cycle, then there is not deadlock.
- If each resource type has one instance, then a cycle implies that there is a deadlock (necessary and sufficient condition).
- If each resource type in a cycle has one instance, then there is a deadlock (necessary and sufficient condition).
- Time complexity of cycle detection algorithm = O(n2)

E. Memory Management

- Required to increase degree of multiprogramming in RAM



Static Memory Allocation

• Divide memory into fixed sized block • Size of each partition may or may not be same • Allocate one block to each process	• Internal Fragmentation • External Fragmentation • Limit in process size • Limit in degree of multiprogramming

Dynamic Memory Allocation

• Allocate available space to a process • No internal Fragmentation • No limit on process size • No limit on number of process	• External Fragmentation ○ **Solution: Compaction** • Allocation/Deallocation becomes complex in presence of holes

- First Fit: Allocate first hole that is big enough
- Next Fit:Same as first fit but start searching always from last allocated hole
- Best Fit: Allocate smallest hole that is big enough
- Worst Fit: Allocate the largest hole

- Best fit and Worst fit are slower

Paging

- Virtual Address is split into a number of pages of finite size
- Page size of fixed
- Page size = Frame size

Page Table

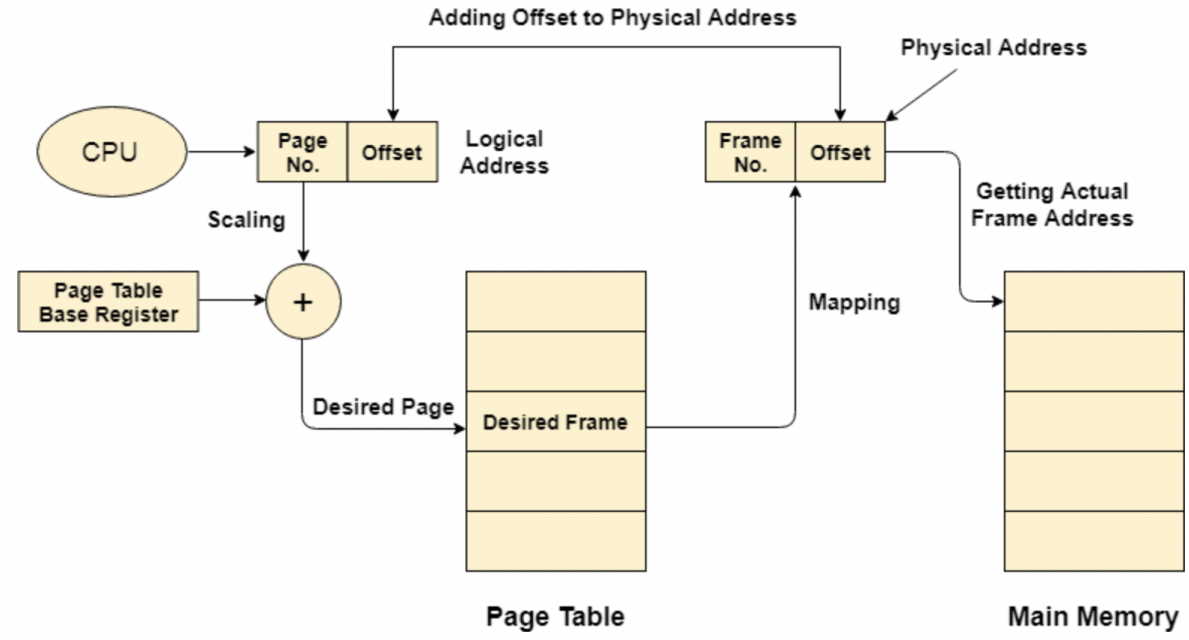
- Maps **Logical Address (generated by CPU)** to **Physical Address (address in Primary Memory)**
- Mapping is maintained by **Memory Management Unit (MMU)**

Logical Address

Page no.	Offset

Physical Address

Frame no.	Offset



Page Table Entry

Frame Number	Valid / Invalid (I/O)	Protection (RWX)	Reference (0/1)	Caching	Dirty / Modified bit

← Mandatory field——————><---------- Optional ——————————————————————————>

- Each process is allocated a page table.
- The page table is stored in the memory
- A page table base register (PTBR) points to the page table.
- Changing the page table requires just changing the value of the register, thus reducing the context switch time.

Advantage/Disadvantages

Advantages:

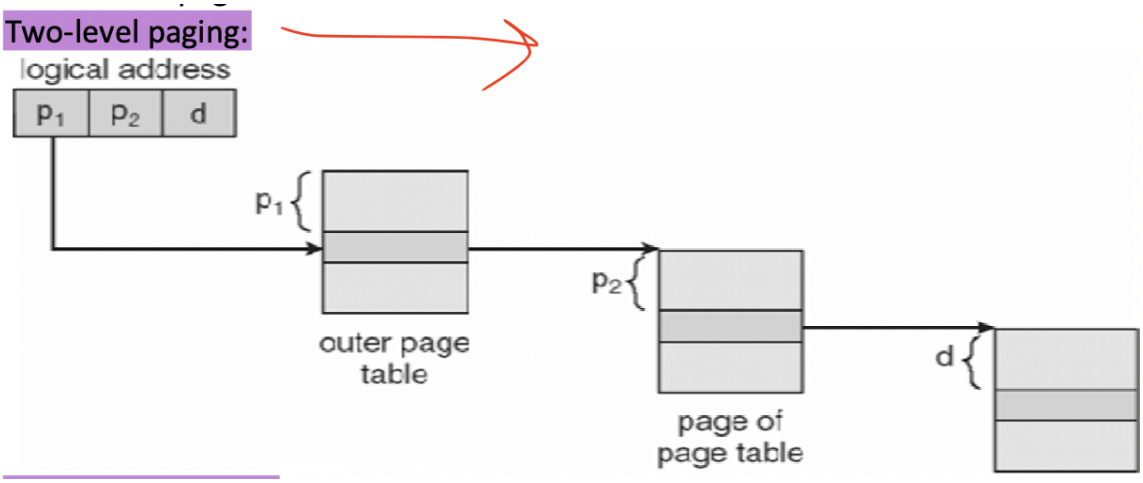
- Size of process independent of page size
- No external fragmentation

Disadvantage:

- Internal fragmentation - Paging uses constant-size blocks of memory, and thus minimizes external fragmentation at the expense of internal, if the memory allocated is less than a page.

Multilevel Paging

- When the Size of page table is less than the size of one Frame then we need not worry because we can directly put the page table in a frame of the main memory .Thus we can directly access the page table.
- But if the size of the page table is larger than the size of Frame. Then the page table in return is to be divided into several pages and these pages of the page table are to be stored in the main memory. Thus, a Outer Page Table comes into the picture.
- This Outer Page Table would contain the address of the Frames which contain the pages of Inner Page Table (i.e., Page Table one pages) in the main memory.
- It reduces the size of page table in physical memory



Inverted Paging (Slower)

- An alternate approach is to use the Inverted Page Table structure that consists of one-page table entry for every frame of the main memory. So the number of page table entries in the Inverted Page Table reduces to the number of frames in physical memory and a single page table is used to represent the paging information of all the processes.
- The overhead of storing an individual page table for every process gets eliminated and only fixed portion of memory is required
- Indexing is done with respect to frame number instead of logical page number

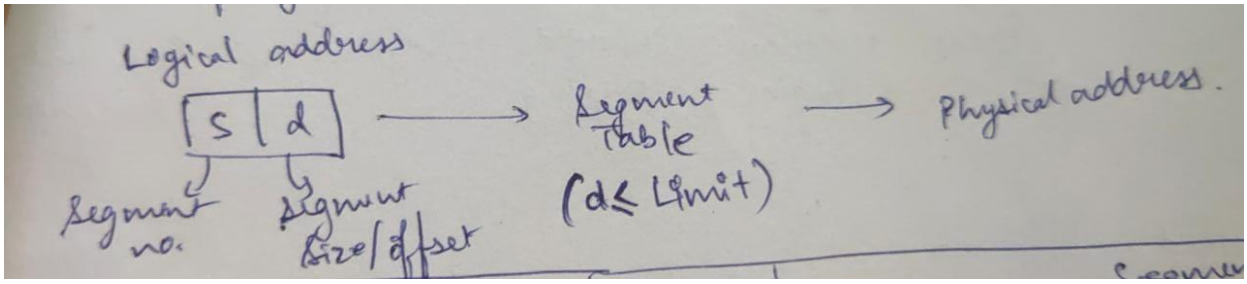
Segmentation

- Program is divided into segments not necessity of same size
- No internal fragmentation
- External fragmentation is possible
- Consumes less space compared to Page Table

Segment Table

Segment Number	Offset

- A process may have segments each storing a part of code or data. A typical program has the following segments –
  - code (CS)
  - data (DS)
  - stack (SS)
  - extra (ES)

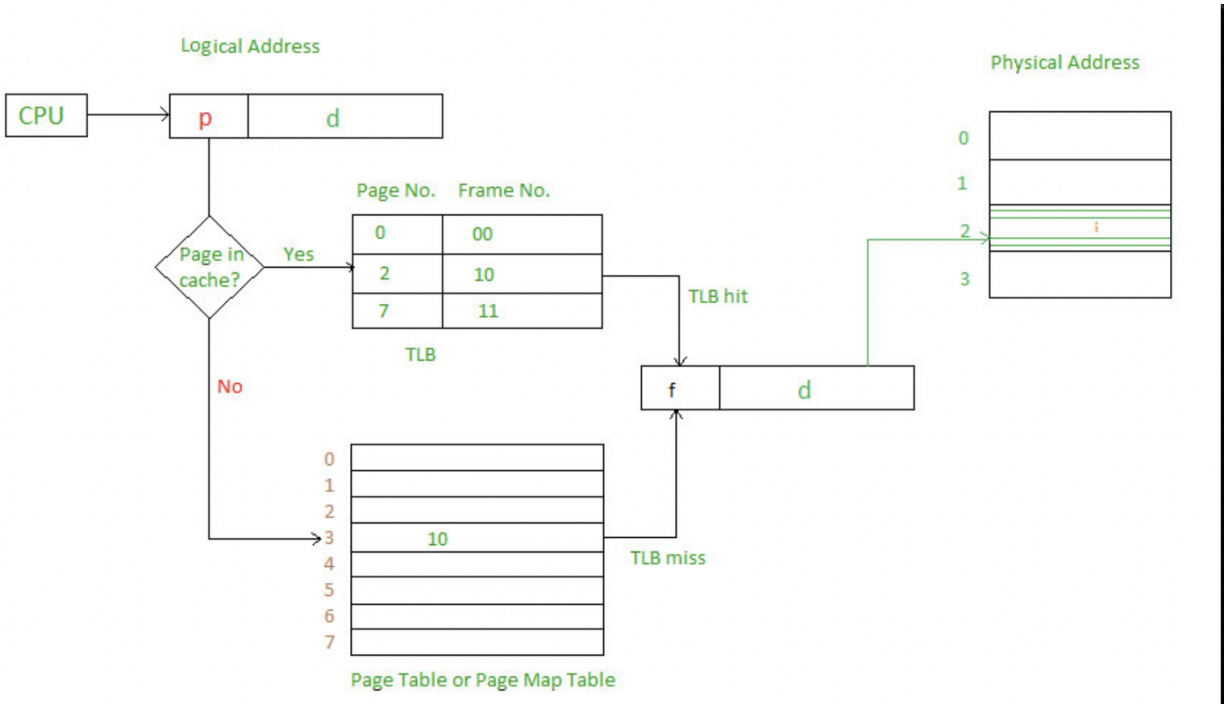


Paging vs Segmentation

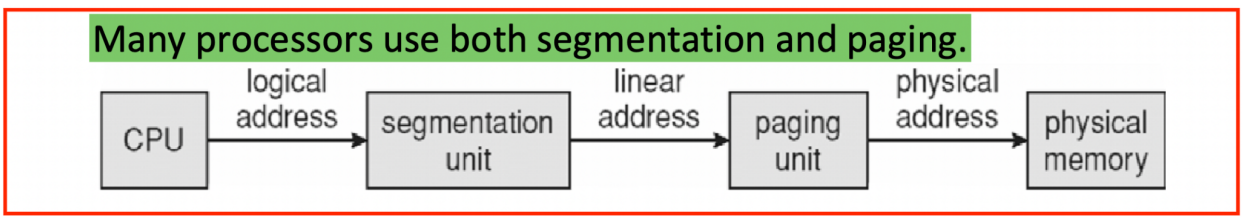
Paging	Segmentation
• Program divided into fixed page size • Page size determined by hardware • Faster • Internal Fragmentation possible • Logical Address split into page number & Page offset • Invisible to user	• Program divided into variable size • Section size determined by user • Slower • External Fragmentation possible • Logical Address split into section number & section offset • Visible to user

Transition Lookaside Buffer (TLB)

- Special cache used to keep track of recently used transaction
- Contains Page table entries that have been most recently used



- **TLB Hit**
  - CPU generates logical address
  - Check in TLB (Hit)
  - Corresponding frame number is retrieved, which tells where in the main memory page lies
- TLB Miss
  - CPU generates logical address
  - Check in TLB (Miss)
  - Now page number is matched to page table residing in main memory
  - Corresponding frame number is retrieved
  - TLB updated with new Page Table entry
- Effective Memory Access Time (EMAT) =
  h(c+m) + (1-h)(c+2m)
  h - Hit Ratio
  c - TLB Access Time
  m - Memory Access Time



F. Virtual Memory Management

- Virtual memory is a memory management technique where secondary memory can be used as if it were a part of the main memory
- Virtual memory is a technique that allows the execution of processes that are not completely in memory.
- This allows execution of programs larger than the memory.
- This allows higher degree of multiprogramming.
- A part of disk is logically treated as part of process space (VM)
- **Swap space** is the area on a hard disk which is part of the Virtual Memory of your machine, which is a combination of accessible physical memory (RAM) and the swap space. Swap space temporarily holds memory pages that are inactive.
- The two ways computers handle virtual memory are through **paging** and **segmenting**.

How does virtual memory work?

Virtual memory uses both the computer's software and hardware to work. It transfers processes between the computer's RAM and hard disk by copying any files from the computer's RAM that aren't currently in use and moving them to the hard disk. By moving unused files to the hard disk, a computer frees up space in its RAM to perform current tasks, such as opening a new application. If the computer later needs to use its RAM for a more urgent task, it can again swap files to make the most of the available RAM.

RAM is a limited resource stored on chips that are built into the computer's CPU. Installing more RAM chips can be expensive, so virtual memory allows the computer to move files between systems as needed to optimize its use of the available RAM.

Virtual memory vs. physical memory

The two most significant differences between virtual memory and physical memory are speed and cost. Computers that rely on physical memory tend to be faster than computers relying on virtual memory. However, increasing a computer's physical memory capacity is more expensive than implementing a virtual memory system. For these reasons, most computers use their physical memory system—their RAM—to maintain the speed of their processing before using their virtual memory system. The computer only uses its virtual memory when it runs out of RAM for storage.

However, users also have the option to expand their RAM. Installing more RAM can resolve computer delays caused by frequent memory swaps. The amount of RAM a computer has depends on how much the user or manufacturer installs. Comparatively, the size of the computer's hard drive determines its virtual memory capacity. When choosing a computer, you may prefer one with more RAM if you're someone who runs many applications at a time. If you work with only one or two computer applications most of the time, you may not need as much RAM.

Advantage/Disadvantage

Advantages of Virtual Memory

- The degree of Multiprogramming will be increased.
- User can run large application with less real RAM.
- There is no need to buy more memory RAMs.

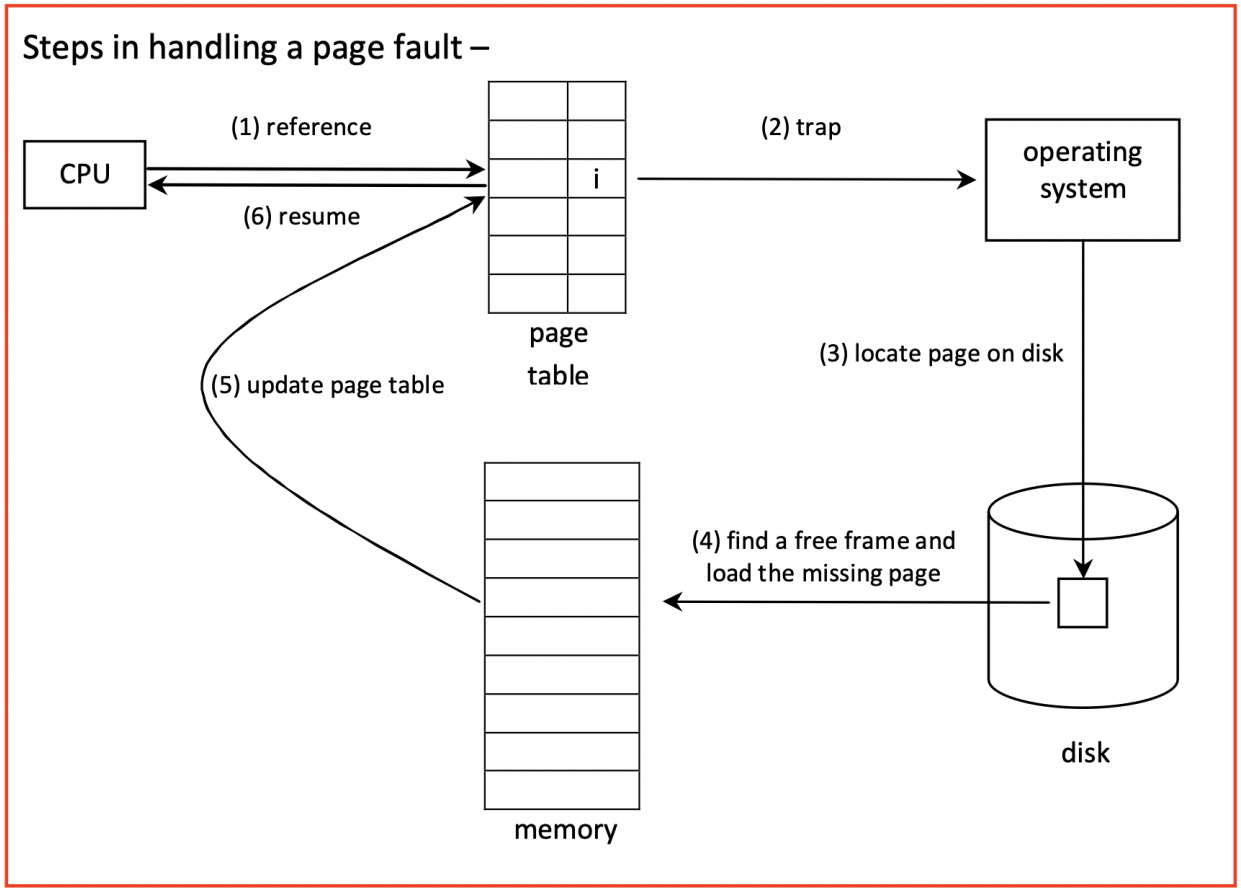
Disadvantages of Virtual Memory

- The system becomes slower since swapping takes time.
- It takes more time in switching between applications.
- The user will have the lesser hard disk space for its use.

Demand Paging

- Demand paging is an approach to implement virtual memory.
- Pages are loaded into the memory only when the CPU wants to access them.
- **Pure demand paging** is an extreme case of demand paging where the execution of a process is started with none of its pages in memory.
- Pages that the CPU does not want to access are not loaded.
- We use a program called **lazy swapper**; also known as pager.

- If the CPU wants to access a memory-resident page, then the execution continues normally.
- If the CPU wants to access a page that is not in memory, then a **page fault trap occurs.**
- **Trap** highest priority non-maskable external-hardware interrupt)
- Effective access time = (1-p) x memory access time + p x page fault service time
  p - Page Fault Ratio
  Typically, $0 < p < 1$
- In case of Trap control is transferred to OS to perform the **page fault service routine**

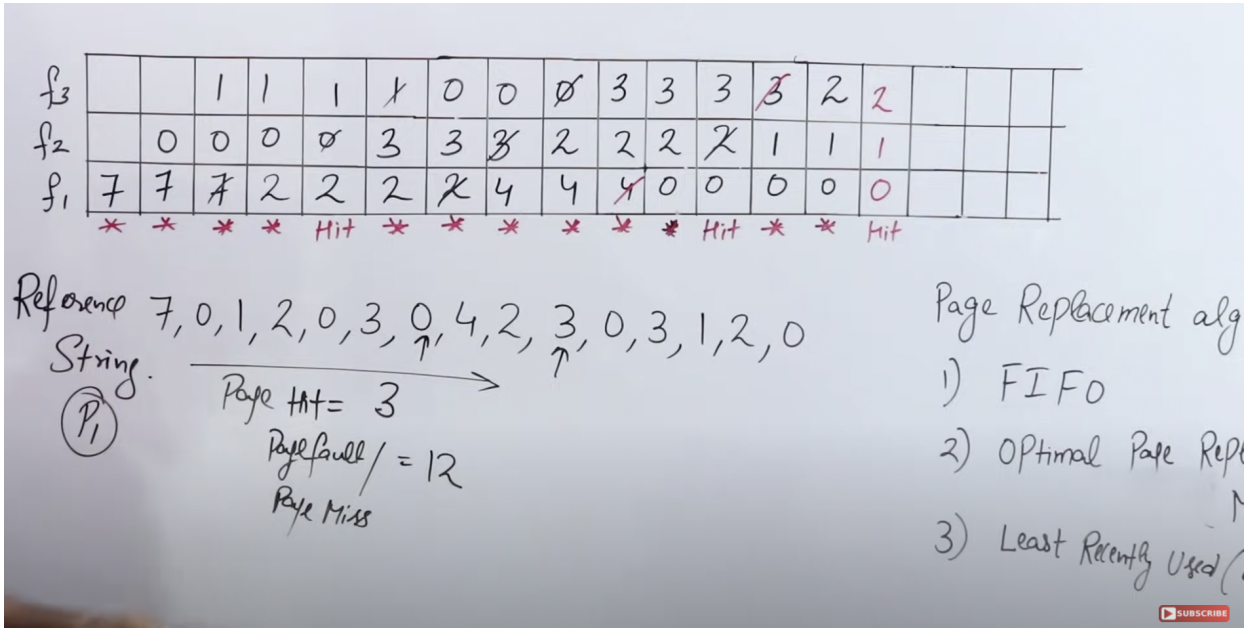


Page Replacement

- Process of replacing one page by another in the memory when there is no free frame.

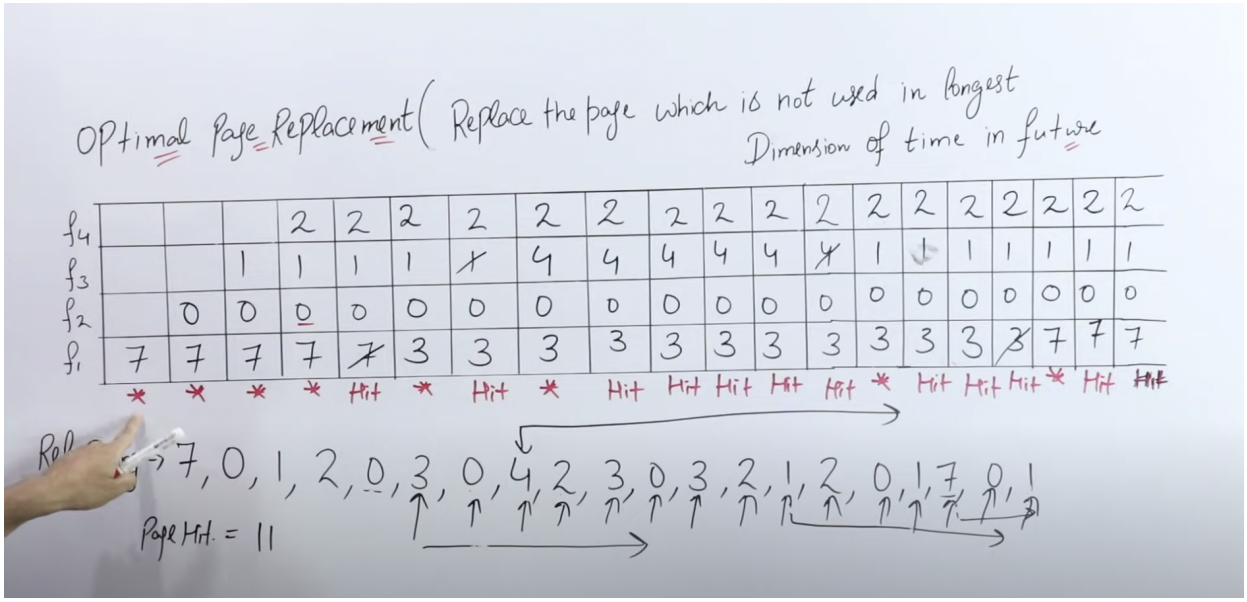
FIFO

- Page which was brought first will be replaced
- No need to note time, only a queue is required
- simple, but performance not good
- shows **Belady's anomaly**



Optimal Page Replacement Algorithm

- Remove page which will not be referred for the longest period of time on future
- Lowest page fault rate, no Belady's anomaly
- Needs future knowledge of reference string, hence, impossible to implement



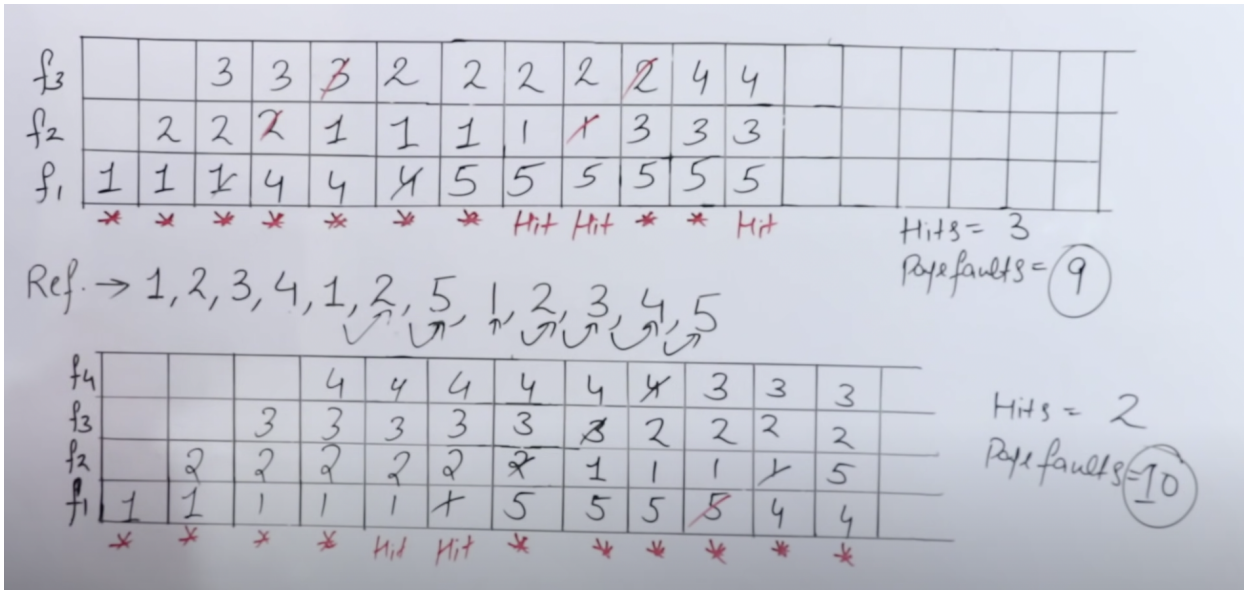
LRU

- Replace the least-recently used page
- No Belady's anomaly
- Implemented using counters or a stack

A **stack algorithm** does not suffer from Belady's anomaly.
A page replacement algorithm is a stack algorithm if we can show that the pages in memory for n frames is always a subset of pages in memory that will be there in memory for (n+1) frames.

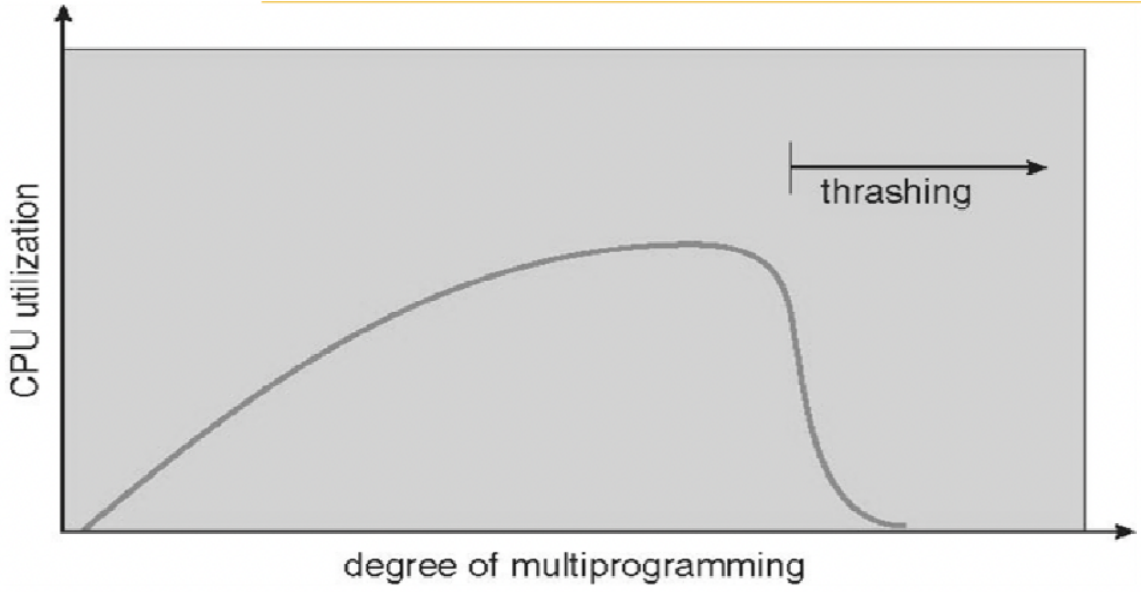
Belady Anomaly

- Phenomenon in which increasing the number of page frames results in an increase in the number of page faults for certain memory access patterns.
- This phenomenon is commonly experienced when using the first-in first-out (FIFO) page replacement algorithm.



Thrashing

- Situation when a system is **spending more time in servicing page faults than executing.**
- Since all the pages in memory are in active use, the page that will be replaced will be needed again. This will bring down the CPU utilization and the system will load more processes in the memory in order to increase the degree of multiprogramming.
- This will trigger a chain reaction of page faults.
- **This situation when the CPU utilization diminishes due to high paging activity is called thrashing.**

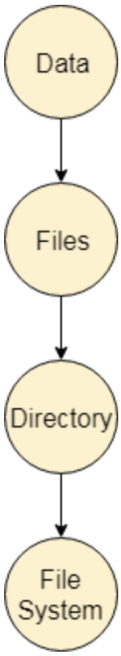


Avoiding Thrashing

- Increase Main Memory
- Use Long Term Scheduler
- Change the scheduling policy
- Working set Model - based on locality of reference

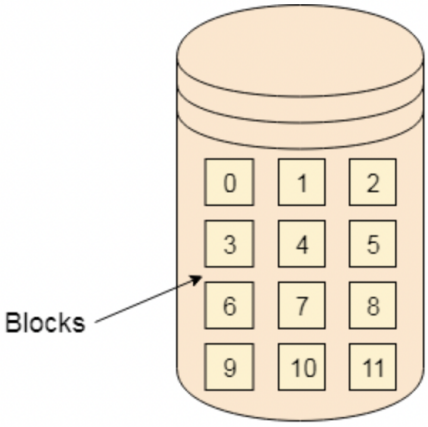
G. Storage Management

- **File** is a named collection of related information that is recorded on secondary storage.
- A file is the smallest allotment of logical secondary storage.
- Data cannot be written to secondary storage unless they are within a file. Files store both code and data.
- File formats – magic number.



- **File System:** To provide efficient and convenient access to the disk, **the operating system imposes a file system to allow data to be stored, located and retrieved easily.**
- **Directory** is a logical construct representing a set of files and subdirectories.
- **File Allocation Methods:**
    - Contiguous
    - Linked
    - Indexed

Contiguous Allocation



Hard Disk

File Name	Start	Length	Allocated Blocks
abc.text	0	3	0,1,2
video.mp4	4	2	4,5
jtp.docx	9	3	9,10,11

Directory

Contiguous Allocation

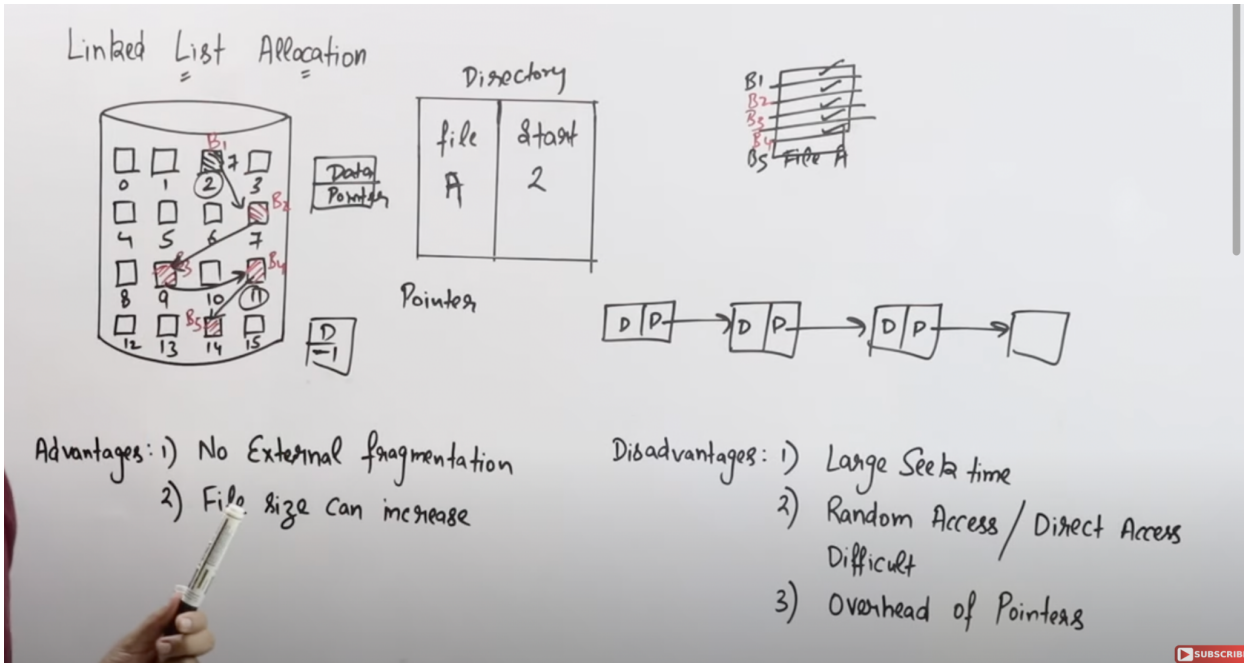
Advantages:

- Simple and Easy
- We will get Excellent read performance.
- Supports Random Access into files.

Disadvantages:

- Internal and External Fragmentation
- Difficult to grow file

Linked List Allocation

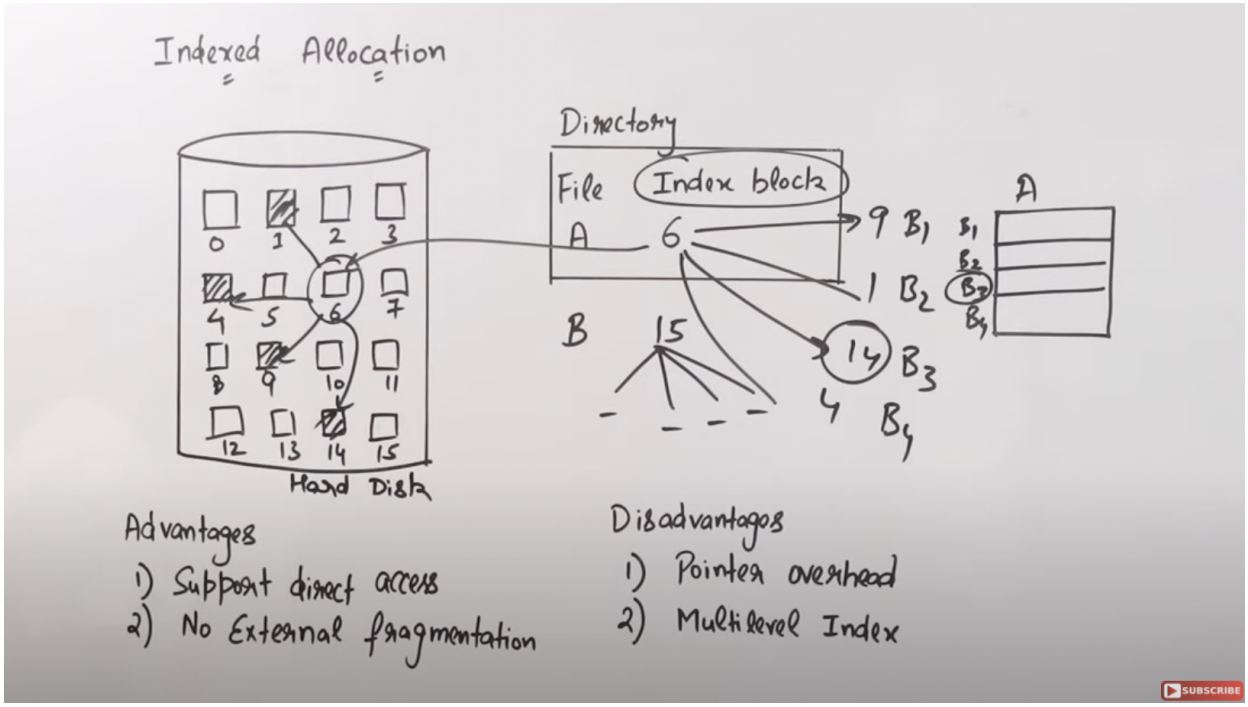


File Allocation Table (FAT)

- Used to facilitate direct/random access of files in Linked Allocation
- The table has one entry for each disk block and is indexed by block number
- FAT needs to be cached in order to reduce head seeks

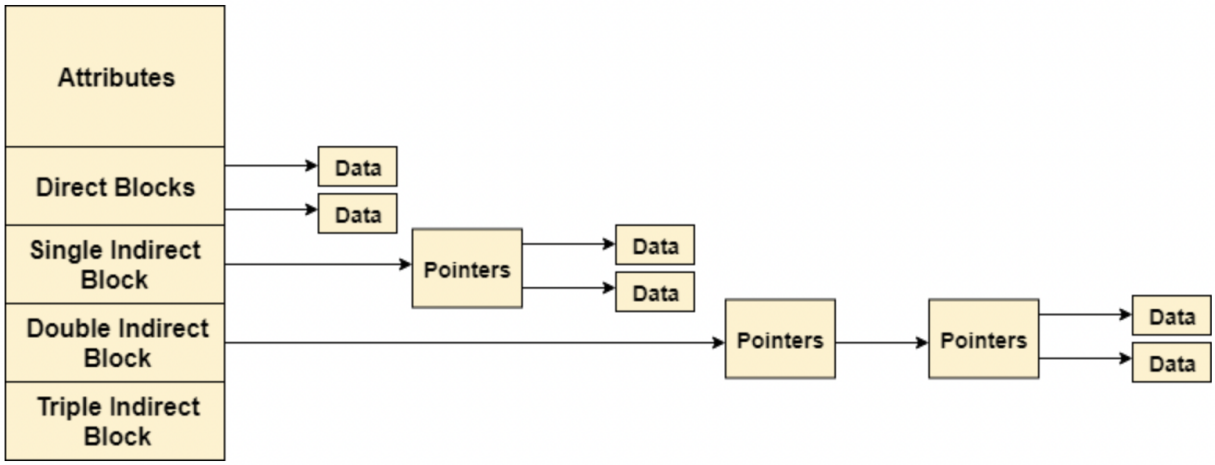
Indexed Allocation

- Use disk blocks as **index block** which don't hold data but holds pointer to disk blocks that hold the data
- Supports direct access
- No external fragmentation
- Easy to grow files



Inode

- In Unix based operating system each file is indexed by an Inode.
- It uses Indexed Allocation and can be said to be real life implementation of Indexed Allocation in UNIX systems

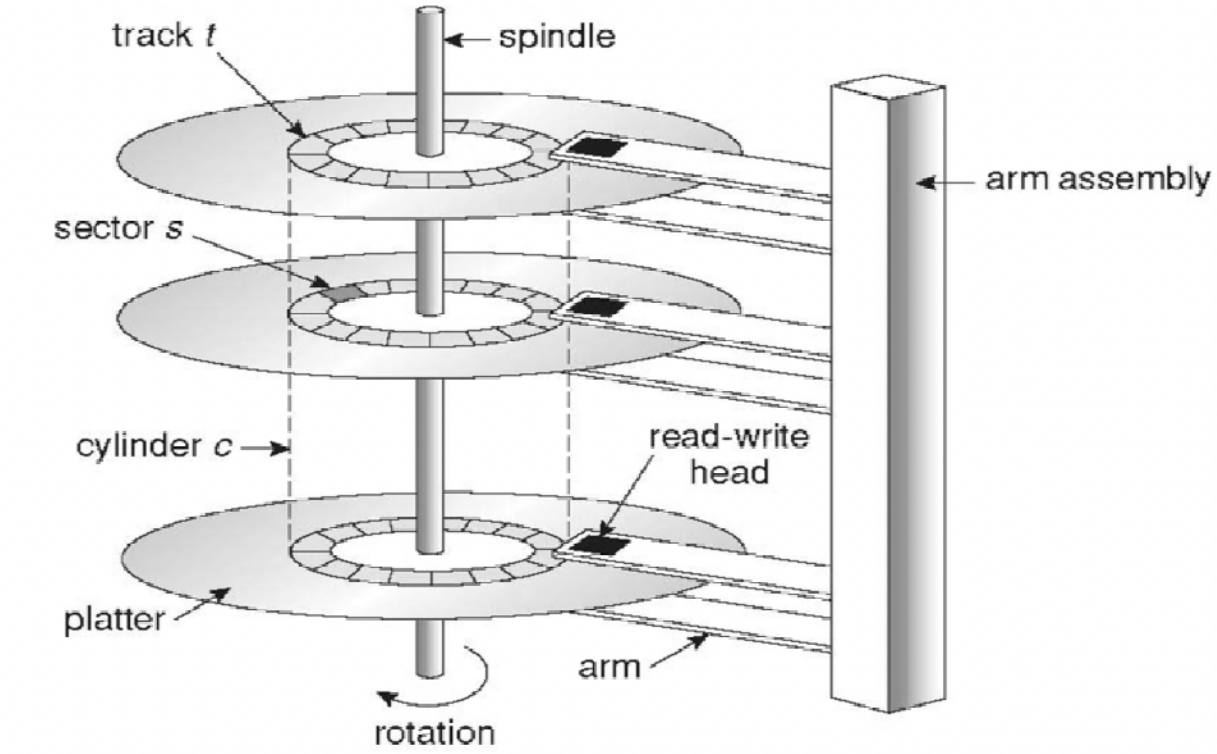


- Attributes (permissions, time stamp, ownership details, etc) of the file
- A number of direct blocks which contains the pointers to first 12 blocks of the file.
- A single indirect pointer which points to an index block. **If the file cannot be indexed entirely by the direct blocks then the single indirect pointer is used.**
- A double indirect pointer which points to a disk block that is a collection of the pointers to the disk blocks which are index blocks. Double index pointer is used if the file is too big to be indexed entirely by the direct blocks as well as the single indirect pointer.

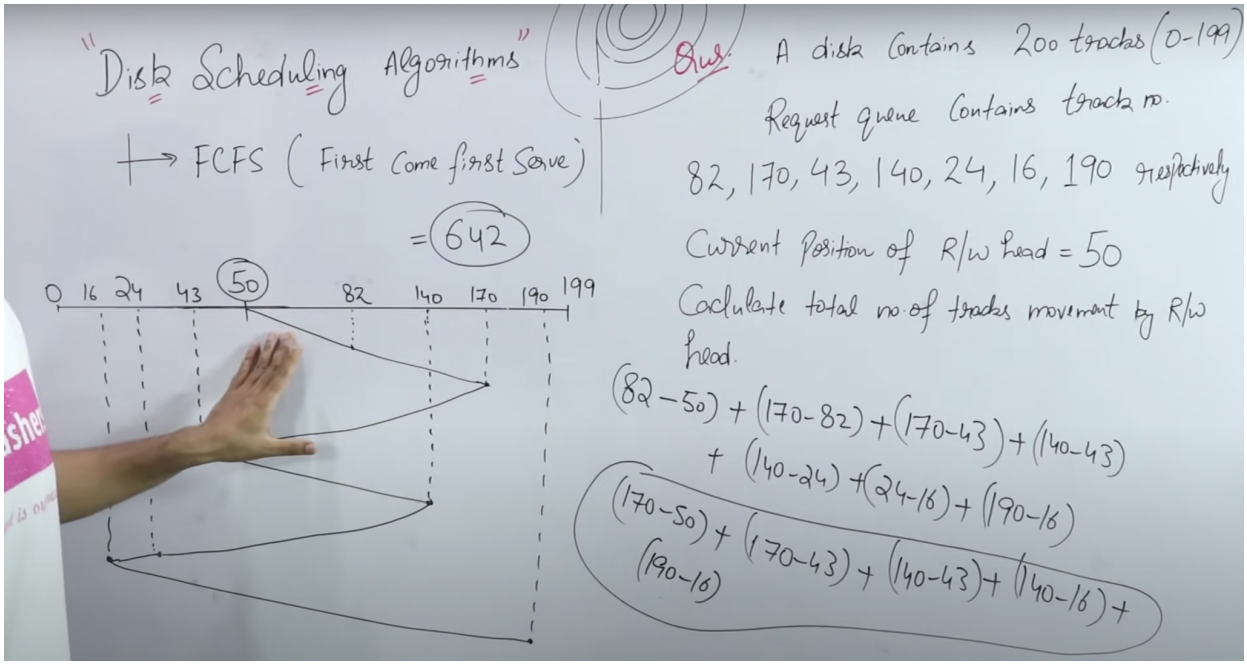
- A triple index pointer that points to a disk block that is a collection of pointers. Each of the pointers is separately pointing to a disk block which also contains a collection of pointers which are separately pointing to an index block that contains the pointers to the file blocks.

Disk Scheduling

- **Seek Time:** Time taken by read-write head to reach desired track
- **Rotation Time:** Time taken for one full rotation(360 degree)
- **Rotational Latency:** Time taken to reach the desired sector (half of rotation time)
- **Transfer time:** (Data to be transferred / Transfer rate)
- **Transfer Rate**: Rate at which data is transferred between the bus and the disk
  = No. of head * Capacity of one track * No. of rotation in one sec
- **Positioning time (random access time):** seek time + rotational latency.
- Performance of a disk depends on transfer rate and positioning time.

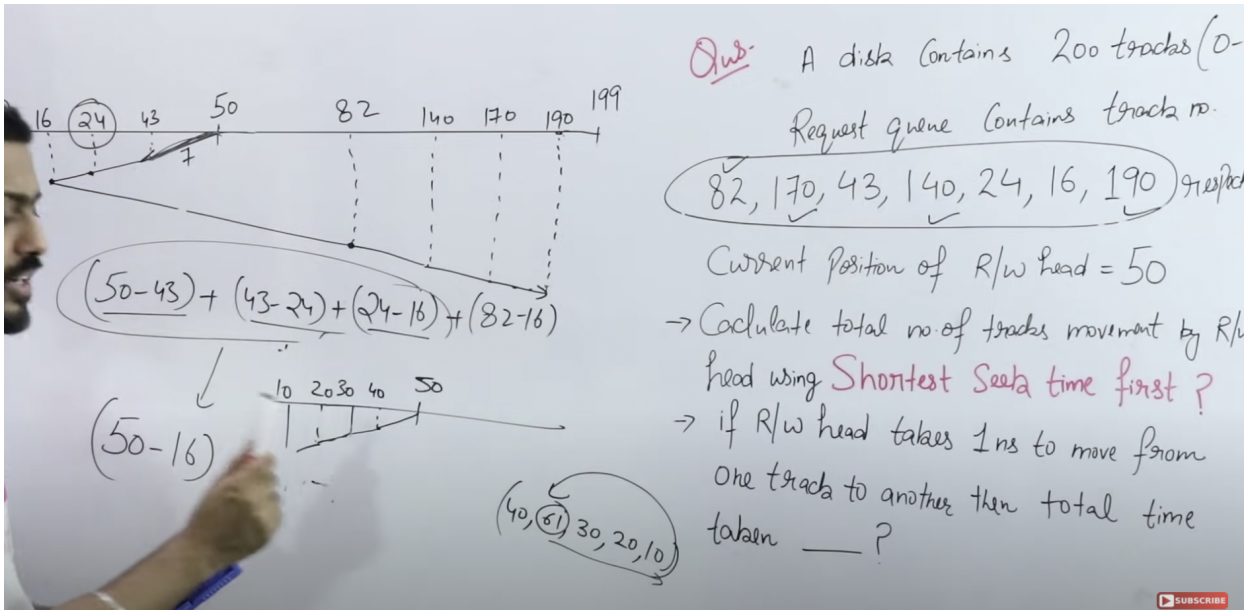


FCFS - First Come First Serve



SSTF - Shortest Seek Time First

- May lead to Starvation
- Overhead to calculate nearest value to head

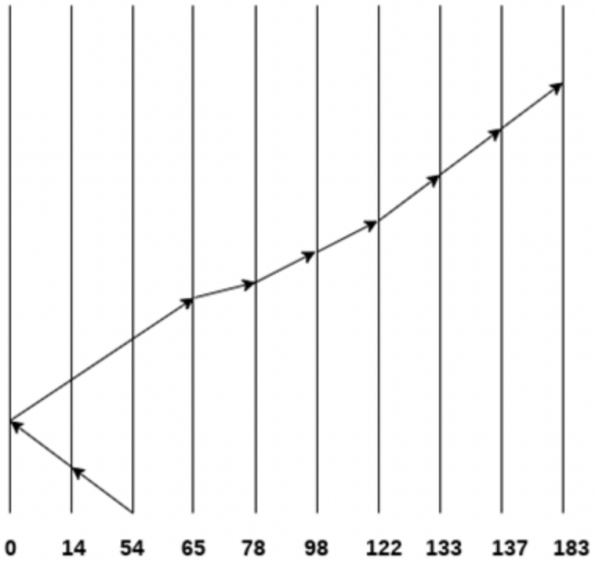


SCAN

- Know head position and direction

98, 137, 122, 183, 14, 133, 65, 78

Head pointer starting at 54 and moving in left direction. Find the number of head movements in cylinders using SCAN scheduling.

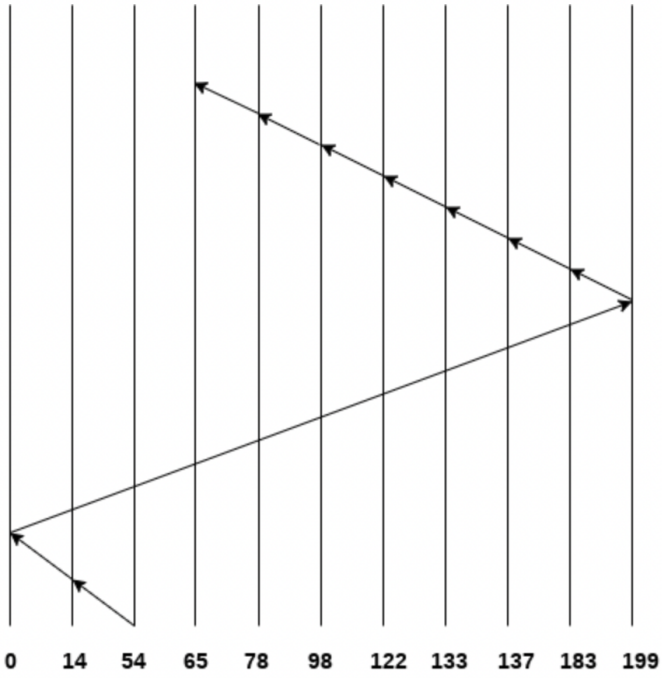


Number of Cylinders = 40 + 14 + 65 + 13 + 20 + 24 + 11 + 4 + 46 = 237

CSCAN

98, 137, 122, 183, 14, 133, 65, 78

Head pointer starting at 54 and moving in left direction. Find the number of head movements in cylinders using C-SCAN scheduling.

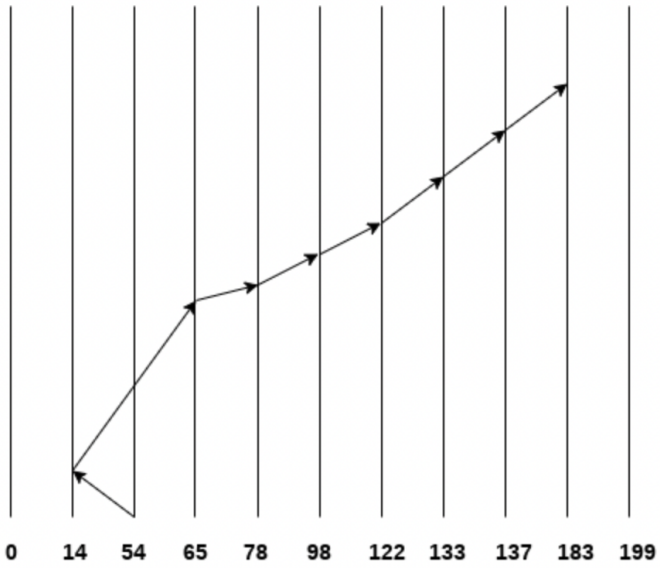


No. of cylinders crossed = 40 + 14 + 199 + 16 + 46 + 4 + 11 + 24 + 20 + 13 = 387

LOOK

98, 137, 122, 183, 14, 133, 65, 78

Head pointer starting at 54 and moving in left direction. Find the number of head movements in cylinders using LOOK scheduling.

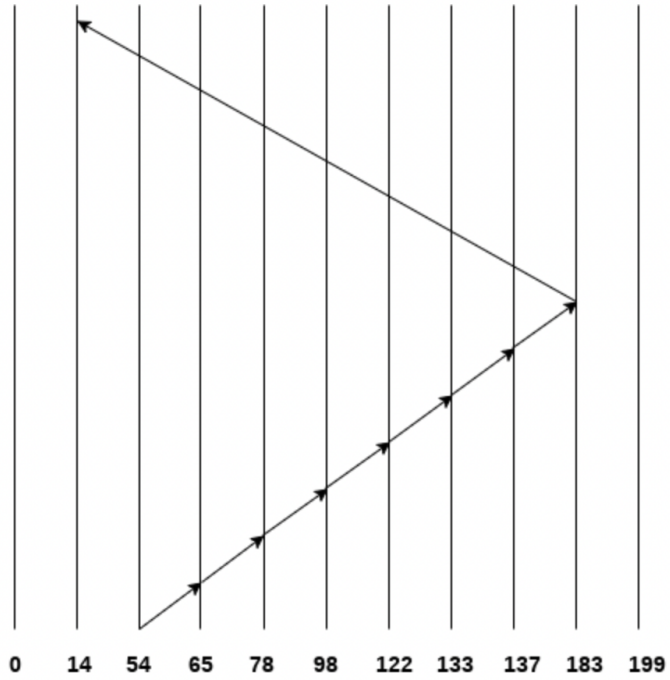


Number of cylinders crossed = 40 + 51 + 13 + +20 + 24 + 11 + 4 + 46 = 209

CLOOK

98, 137, 122, 183, 14, 133, 65, 78

Head pointer starting at 54 and moving in left direction. Find the number of head movements in cylinders using C LOOK scheduling.



Number of cylinders crossed = 11 + 13 + 20 + 24 + 11 + 4 + 46 + 169 = 298