Page No:
Date:

What is Operating System ?

① Intermediatory - Acts as the intermediatory b/w user and hardware

② Resource manager / allocator - Operating system controls and coordinates the use of system resources among various application programs in an unbiased fashion

③ Platform - OS provides the platforms on which other application programs can be installed, provides the environment within which programs are executed

- Application software performs specific tasks for the user
- System software operates and controls computer system and provides a platform to run application software

Operating system functions -

(i) Access to computer hardware
(ii) Interface b/w user and computer hardware
(iii) Resource management (AKa, arbitration) (memory, device, file, security, process etc)
(iv) Hides the underlying complexity of the hardware (AKa, abstraction)
(v) Facilitates execution of application programs by providing isolation and protection.

OS goals -

- Maximum CPU utilization
- less process starvation
- Higher priority job execution

Major components of operating systems

① Kernel
- Central component : Manages the system's resources and communication b/w hardware and software.

② Process management
- Process scheduler : Determines execution of process

- Process control block (PCB): contains process details such as process ID, priority, status, etc.
- Concurrency control: Manages simultaneous execution.

③ Memory management
- Physical memory management: - Manages RAM allocation
- Virtual memory management - Stimulates additional memory using disk space.
- Memory allocation - Assigns memory to different processes

④ File system management
- File handling - Manages the creation, deletion and access of files and directories.
- File control block - Stores file attributes and control information
- Disk scheduling - Organises the order of reading or writing to disk

⑤ Device management
- Device drivers: Interface b/w hardware and the OS
- I/O Controllers: Manage data transfer to and from peripheral devices.

⑥ Security and Access control
- Authentication: Verifies user credentials
- Authorization: Controls access permissions to files / directories
- Encryption: Ensures data confidentiality and integrity

⑦ User interface
- Command line Interface (CLI): text based user interaction
- Graphical User Interface (GUI): Visual, user friendly interaction with the OS.

⑧ Networking
- Network protocols: Rules for communication b/w devices on a network

• Network Interface : Manages connection b/w computer and network

Single Process OS

Only one process executes at a time from ready queue

Batch processing system

① Firstly, user prepares his job using punch cards
② Then, he submits the job to the computer operators
③ Operator collects the jobs from different users and sort the jobs into batches with similar needs
④ Then operator submits the batches to the processor one by one
⑤ All the jobs of one batch are executed together

- Priorities cannot be set, if a job comes with some higher priority.
- May lead to starvation (A batch may take more time to complete)
- CPU may become idle in case of i/o operations

Multiprogramming

It increases CPU utilization by keeping multiple jobs (code and data) in the memory so that the CPU always has to execute in case some jobs get busy with i/o.

- Single CPU
- Context switching for processes
- Switch happens when current process goes to wait state.
- CPU idle time reduced.

Page No:
Date:

- **Multitasking** - It is a logical extension of multiprogramming
  - Single CPU
  - Able to run more than one task simultaneously
  - Context switching and time sharing used.
  - Increases responsiveness
  - CPU idle time is reduced further.

- **Multiprocessing OS** - More than 1 CPU in a single computer
  - Increases reliability, 1 CPU fails others can work
  - Better throughput
  - lesser process starvation (if 1 CPU is working on some process, other can be executed on other CPU)

- **Distributed OS** - OS manages many bunch of resources, >=1 CPUs, >=1 memory, >=1 I/Os etc
  - Loosely connected autonomous, interconnected computer nodes
  - Collection of independent, networked, communicating and physically separate computational nodes

- **Real time operating system** - A real time operating system is a special purpose operating system which has well defined fixed time constraints. Processing must be done within the defined time limit or system will fail.

  Valued more for how quickly or how predictably it can respond, without buffer delays than for the amount of work it can perform in a given amount of time.

  Ex: Air traffic control systems, ROBOTS etc.

Page No:
Date:

Program: A program is an executable file which contains certain set of instructions written to complete the specific job or operation on your computer

- Its a compiled code, ready to be executed
- Stored in disk

Process: Program under execution. Resides in computers primary memory (RAM).

Thread:
- Single sequence stream within a process
- An independent path of execution in a process
- Light weight process
- Used to achieve parallelism by dividing a process into which are independent path of execution
- Ex: Multiple tabs in a browser
  Text editor

Multi tasking	Multi threading
• The execution of more than one task simultaneously is called multi tasking.	• Process divided into several different ~~threads~~ sub tasks called threads which has its own path of execution.
• Concept of more than 1 processes being context switched	• Concept of more than 1 thread Threads are context switched.
• No. of CPU = 1	• No. of CPU >= 1
• Isolation and memory protection exists. OS must allocate separate memory and resources to each program that CPU is executing.	• No isolation and memory protection, resources are shared among threads of that process. OS allocates memory to a process; multiple threads of that process share same memory & resources allocated to the process.

Page No:
Date:

Thread Scheduling: Threads are scheduled for execution based on their priority. Even though threads are executing within the runtime, all threads are assigned processor time slices by the OS.

Thread Context Switching	Process context switching
• OS saves current state of thread and switches to another thread of same process.	• OS saves current state of process and switches to another process by restoring its state.
• Doesn't include switching of memory address space	• Includes switching of memory address space
• Fast switching	• Slow switching
• CPU cache state is preserved	• CPU's cache state is flushed

Thread Mapping

Threads are of two types:

Threads
- User level
- Kernel level

The CPU executes a user program in two modes
i) User mode ii) Kernel mode

User mode executes most of the instructions.
Few of the restricted instruction can't run in user mode they run on Kernel mode

CPU can't run I/O operations in user mode, they run on Kernel mode.

Page No:
Date:

There are two types of stack → User mode stack
→ Kernel mode stack

The threads which are created inside a single process is the user level thread.
User level thread are created, managed by the user through library functions.

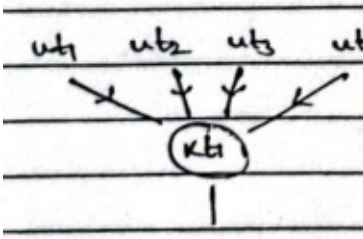
If one of the thread is blocked the remaining thread still also be blocked. Happens in user level thread.
All these user level threads belong to the same address space.
Switching b/w user level threads is faster.

Thread management is easier in user level thread

Switching b/w two kernel level threads can be performed but switching is a single thread can't be performed.

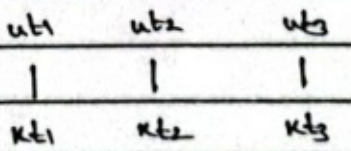
Kernel can create limited threads but user can create any no. of threads.

Mapping → One-one
→ Many-One



(many to one)

It is important because os have limited Kernel but so many user level threads

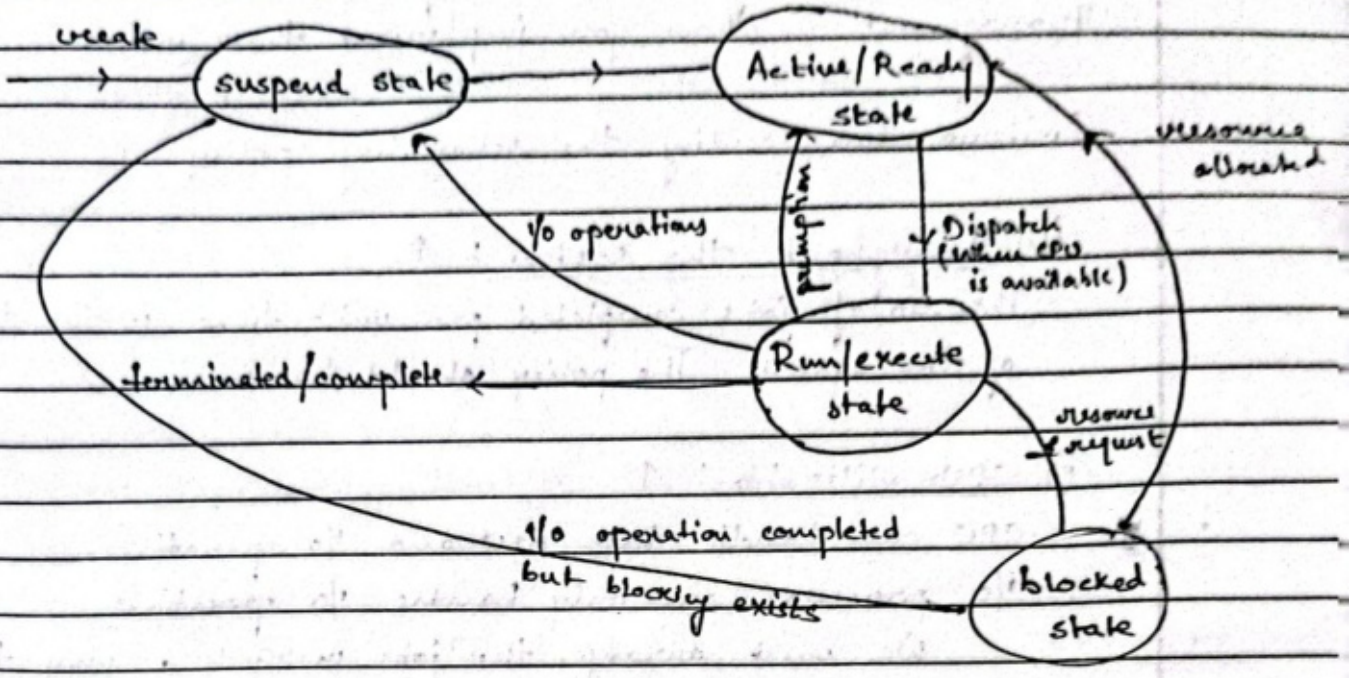


(one to one)

Kernel level threads need more resources so its not that useful.

Page No:
Date:

Process state transition diagram (Snail diagram)



Resource Allocation

- Static
- Dynamic (Min resources to start execution)

Blocked state only appears during dynamic allocation.

A process is in the execution state and requires a resource called X

$P_{ex} \longrightarrow X$

Another process is in the suspended state but have a resource called X

$P_{sup} \longrightarrow X$

- **Policy and management**

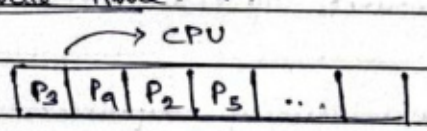
Policy is nothing but a set of rules
Management is how you implement those rules

Criteria for deciding the scheduling policy:

(i) **Throughput the system** : ↑
The no. of jobs completed per unit time is the throughput of the system. The policy should be increased.

ii) **CPU utilization**: ↑
CPU can handle both ALU and I/O operations.
I/O processor can only handle I/O operations
We must arrange the jobs in such a way that CPU must be utilised more.

iii) **Response time**: ↓



Response time is a time a process spends in the active state for the first time being responded by the CPU for execution.

iv) **Wait time** : ↓
Total time a process waits in the active state until execution.

When a process executes in first attempt
response time = wait time.

v) **CPU burst time** : ↓
The time CPU takes to execute a process

vi) **Turn around time** : ↓
Total time for ~~execution~~ completion of executions

Page No:
Date:

Turn around time = CPU burst time + Wait time

* priority should be given → wait time

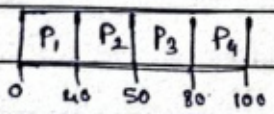
• Average wait time

Avg wait time = $\frac{\text{Sum of wait time of processes}}{\text{No. of processes}}$

First come first service

Process that comes first in the active state will go first for execution.

Process	Arrival time	CPU burst time	Wait time	CPU time (max)
P_1	0	40	0	0+40+10+
P_2	0	10	40	30+20
P_3	0	30	50	
P_4	0	20	80	

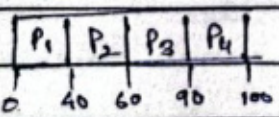


Avg wait time = $\frac{0+40+50+80}{4}$ = 42.5 μsec

FCFS is non preemtive that means it can't be called back before completion.

Process	Arrival time	CPU burst time	Wait time	CPU time (max)
P_1	0	40	0	0+40+20+30
P_2	30	10	60	+10
P_3	20	30	40	
P_4	10	20	30	

Gantt chart



Avg wait time = $\frac{0+60+40+30}{4}$

= 32.5 μsec

Page No:
Date:

Disadvantages of Avg wait time

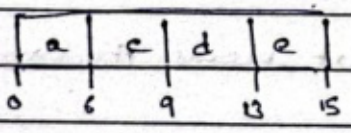
(i) The average wait time would be more.

(ii) If a longer job arrives first all the other small jobs would be waiting for a large amount of time period and as a result the avg wait time will be high.

Shortest job first:

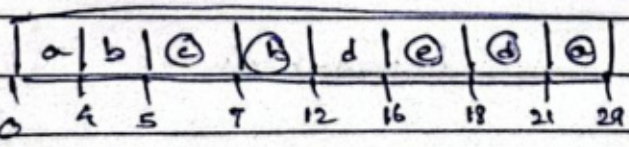
- Non preemptive → Shortest job first
- Preemptive (forcefully taking CPU out of the process without continuing its execution) → Shortest job remaining

Process	at (4)	bt (3)	wt (1)	CPU time (2)
a	0	6	0	6 + 3 + 4
b	4	14		
c	5	3	1	
d	8	4		
e	10	2		



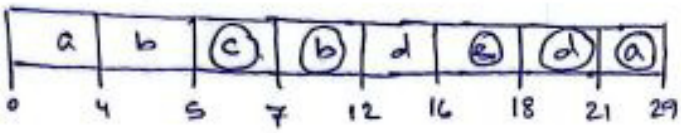
Preemptive

Process	at (4)	bt (3)	wt (1)	CPU time (2)
a	~~0~~ 4	~~6~~ 8	0+17	0+4+1+2+5
b	~~4~~ 5	~~14~~ 10	0+2	+1+2+3+8
c	5	~~3~~ 0	0	
d	~~8~~ 16	~~4~~ ~~2~~ 0	4+2	
e	16	~~2~~ 0	0	



Preemptive

Process	a.t ④	b.t ③	w.t ①	② CPU time
a	0 4	12 8	0+17	0+4+1+2+5
b	4 5	6 ~~5~~ 0	0+2	+4+2+3+8
c	5	2 0	0	
d	8 16	7 ~~3~~ 0	4+2	
e	16	~~2~~ 0	0	



i) Total wait time

ii) Avg wait time

iii) Turn around time

iv) Order of completion of processes

v) Sequence of execution.

vi) Frequency of a process in the runstate.
↳ (How many times a process arrives in the execution state)

vii) ~~When~~ How many context switching are there?
↳ (If a process comes to the active state during its execution)

Disadvantages:

i) If too many small processes arrive the large processes will have to wait for too long. A long long waiting for a process is too high, the phenomena is called ~~start~~ starving.

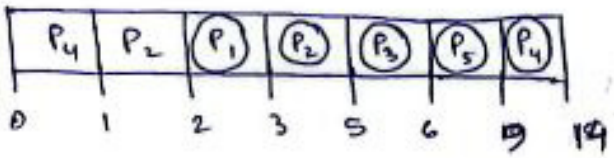
ii) To eliminate starving we use two policies.

a) Round Robin

b) Highest Response Ratio Next

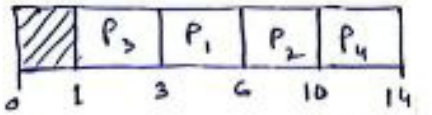
Preemtive

Process	A.t	B.t	Wait time	CPU time
P_1	2	~~1~~ 0	0	
P_2	1	~~3~~ ~~2~~ 0	0+2	
P_3	4	~~1~~ 0	1	
P_4	0	~~6~~ 5	0+9	0+1+1+1+2+1+3+5
P_5	2	3	0+4	



Non-preemtive

Process	A.t	B.t	C.T	Turn around time	W.t	Response time
P_1	1	3	6	5	2	2
P_2	2	4	10	8	4	4
P_3	1	2	3	2	0	0
P_4	4	4	14	10	6	6



Gantt Chart

(TAT) Turn around time = Completion time − Arrival time

Wait time = Turn around time − Burst time

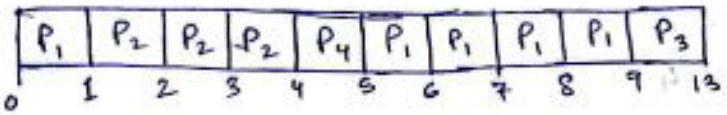
Response time is same as wait time for non-preemtive.

$$\text{Avg TAT} = \frac{5+8+2+10}{4} = 6.25$$

$$\text{Avg wait time} = \frac{2+4+0+6}{4} = 3$$

Shortest Job First for preemtive is called 'Shortest Remaining Time First' (SRTF)

Process	A.T	B.T	C.T	TAT	WT	RT
P_1	0	~~8 4 3 2 1~~ 0	9	9	4	(0-0)=0
P_2	1	~~8 7 6~~ 0	4	3	0	(1-1)=0
P_3	2	~~4~~ 0	13	11	7	(9-2)=7
P_4	4	~~1~~ 0	5	1	0	(4-4)=0



Gantt Chart

TAT = CT - AT

WT = TAT - BT

RT = First CPU time - AT

Criteria = Burst time

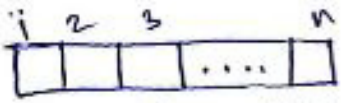
$$\text{Avg TAT} = \frac{9+3+11+1}{4} = 6$$

$$\text{Avg wait time} = \frac{4+0+7+0}{4} = 2.75$$

$$\text{Avg response time} = \frac{0+0+7+0}{4} = 1.75$$

All the processes completed at 13 sec.

Round Robin



AQ (Active State Queue)

q → Each process can be executed in q time

What could be the maximum response time of a proce

For (n-1) no of processes, the total response time

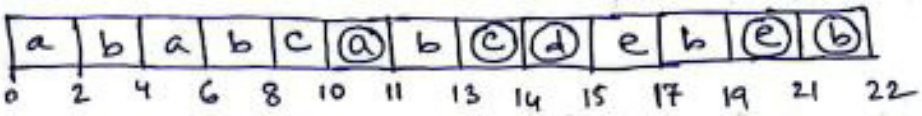
$$(n-1) \times q \rightarrow \max$$

Total maximum time = $(n-1) \times q + q = nq$

Passes #	④ a.t	③ CPU b.t	① w.t	② CPU time
a	0	5	0+2+4	0+2+2+2+2+2+1+2+1+1+2+2+2+
b	1	8	1+2+3+4+2	
c	5	10	3+3	
d	10	1	4	
e	11	7	4+2	

time quantam = 2 unit

Purely Preemtive



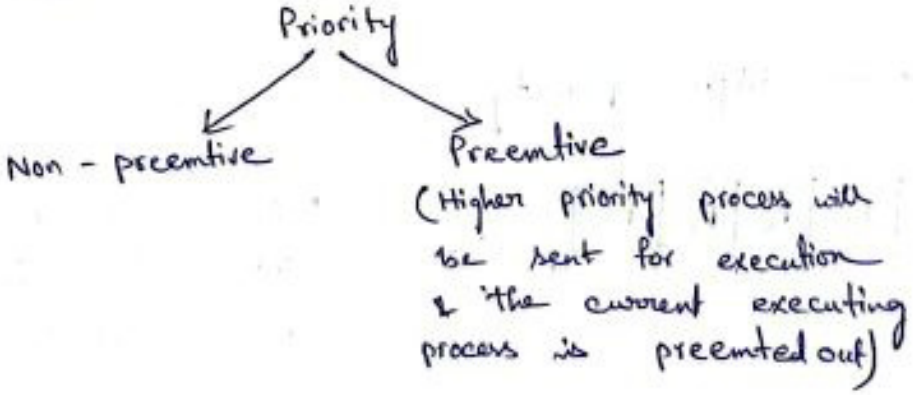
Disadvantages

i) Processes which are Larger are entering more than single time.

ii) Round robin scheduling policy will slow down the execution due to context switching.

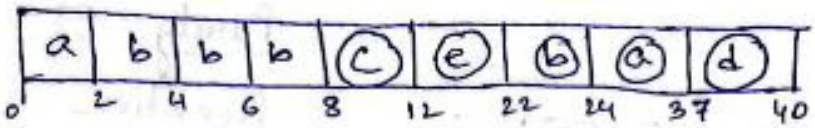
Priority Scheduling

In a real time os priority scheduling is used means which priority is higher will send first.



Non Preemtive

Process #	a.t ④	③ b.t	Priority	w.t ①	② CPU time
a	0 2	~~15~~ 13 0	4	0+22	0+2+2+2+2+4 +10+2+13'+3
b	2 8	8 ~~x~~ 0	3	0+14	
c	8	4 0	1	0	
d	11	3 0	5	26	
e	12	10 0	2	0	

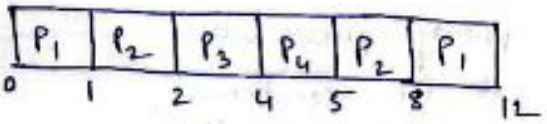


Lower digit indicates higher priority.

Preemtive

Priority	Process	a.t	b.t	c.t	TAT	wt	rt
10	P_1	0	8 ~~x~~ 0	12	12	7	0
20	P_2	1	4 ~~3~~ 0	8	7	3	0
30	P_3	2	~~2~~ 0	4	2	0	0
40	P_4	4	~~1~~ 0	5	1	0	0

Higher no ⟶ Higher priority



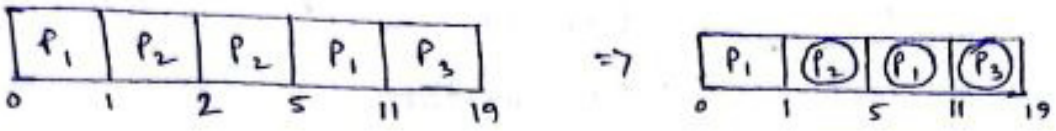
TAT = CT − AT

wt = TAT − BT

RT = First Time CPU − AT

Preemtive SJF → SRTF

Process	a.t	b.t	c.t	TAT	w.t	rt
P_1	0	~~7~~ 6	11	11	4	0
P_2	1	~~4~~ ~~3~~ 0	5	4	0	0
P_3	2	8	19	17	9	9

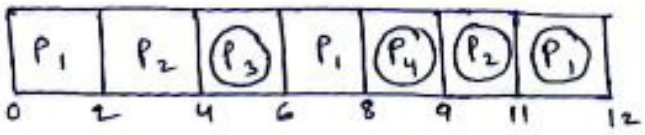


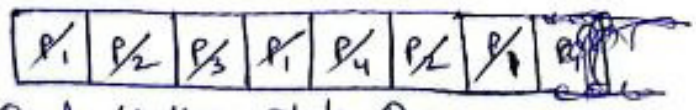
Round Robin

Time Quantum = 2

Process	a.t	b.t	c.t	TAT	w.t	rt
P_1	0	~~8~~ ~~3~~ 1	12	12	7	0
P_2	1	~~4~~ ~~2~~ 0	11	10	6	1
P_3	2	~~2~~ 0	6	4	2	2
P_4	4	~~1~~ 0	9	5	4	4

CPU





Ready / Active State Queue

Process Scheduling Algorithms :

1) First Come First Service → Preemtive
                             → Non-Preemtive

2) Shortest Job First → Preemtive (Shortest Remaining Time Job - SRTJ)
                      → Non-Preemtive (SJF)

3) ~~Rotate~~ Round Robin → Preemtive Only (bcz of TQ)

4) Priority Scheduling → Preemtive
                       → Non-Preemtive

Page No:
Date:

Process Scheduling Algorithms

(i) First come first service → Preemptive / Non preemptive

(ii) Shortest job first → Preemptive (Shortest Remaining Time Job - SRTJ) / Non preemptive (SJF)

(iii) Round Robin → Preemptive only (bez of TQ)

(iv) Priority Scheduling → Preemptive / Non preemptive

(v) Highest response ratio next → Non preemptive

In priority scheduling, starvation occurs to lower priority processes. Aging technique is used to avoid starvation occurs due to priority

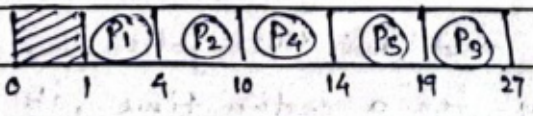
Highest Response Ratio Next (HRRN) - (Priority with aging)

- Non preemptive

- Process with highest response ratio will be sent first

$$1 + \frac{\text{wait time}}{\text{CPU burst time}} = \text{Priority of a process (Response ratio)}$$

Process	AT	BT	CT	TAT	WT	RT
P_1	1	3	4	3	0	
P_2	3	6	10	7	1	
P_3	5	8	27	22	14	
P_4	7	4	14	7	3	
P_5	8	5	19	11	6	



at 10

Response ratio for $P_3 = 1 + (10-5)/8 = 13/8 = 1.62$

$P_4 = 1 + (10-7)/4 = 7/4 = 1.75$

$P_5 = 1 + (10-8)/6 = 7/5 = 1.4$

Page No:
Date:

Implementations

(i) First Come First Serve : Will have a circular queue where new process will be added at the end and processes are ready for execution are extracted from begining.

(ii) Shortest Job First : Will have a sorting algo with 'nlogn' time complexity. After completion of each process or after certain amount of time, will have to run the sorting algo for repeatative nos. So the total running time will be huge.

So SJF is implemented using heap where processes can come (insertion) and go (deletion) in logn time (much more faster than 'nlogn' sort algo)

(iii) Round Robin: Will have a counter that will count downwards according to the time quantum given. When it comes to 0 the currently executing process is sent to the back of the circular queue and a new process is sent for execution. Again the counter will start counting

- Multilevel Poriority Queue : Real Scheduler

Real time and system jobs are very important and smaller. We can apply first come first serve policy.

Because the job sizes are unequal / uneven we go for Round Robin.

For batch jobs we use first come first serve
No scheduling policy is used for NULL jobs
If a job isn't responding for a certain time, its priority will be increased.
After executing for certain time, if the job isn't completed the priority gets decreased

Page No:
Date:

System Calls

When a user uses an application or an API basically the user is in User mode. But if we need to use the functionalities of OS we need to swith to the Kernel mode

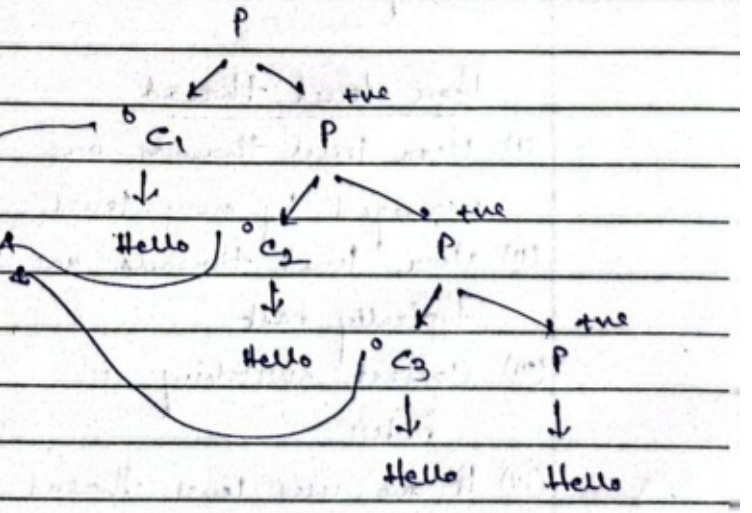
System call is the way of switching from user mode to Kernel mode to access the functionalities of OS. User mode doesn't have the direct access to the hardware thats why we need to switch to the Kernel mode to access via OS

System Calls
- File related => Open(), Read(), Write(), Close(), Create()
- 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
- Communication => pipe(), create/delete connections, Shmget()

Fork: Suppose there is a program that can be referred as a parent program. If we use 'fork' inside the parent program it will create a child program (basically clone of parent program)

Fork → 0 (child)
Fork → 1 (parent)

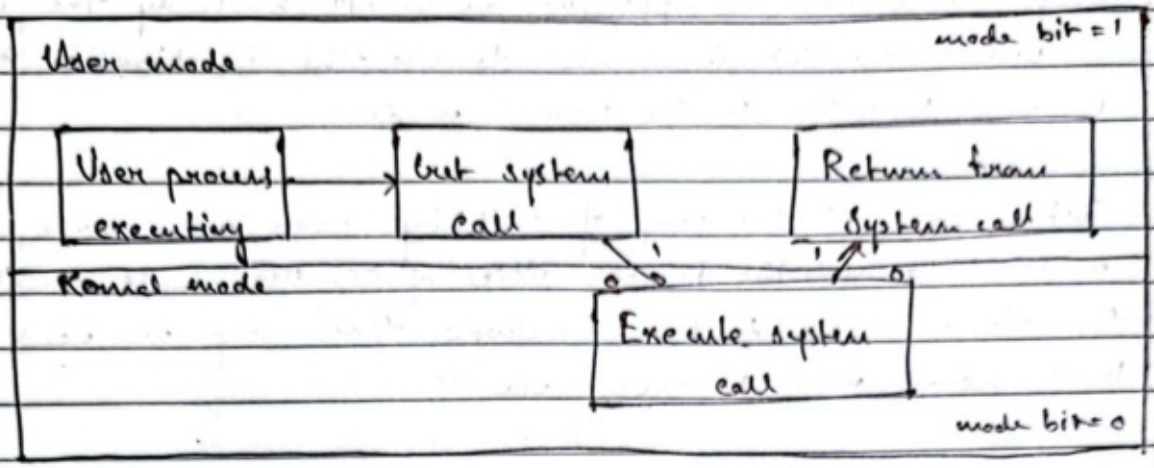
```
#include <stdio.h>
# include <unistd.h>
int main()
{
if (fork() && fork())
  fork();
printf("Hello");
return 0;
}
```



No. of times Hello will be printed

Page No:
Date:

User mode v/s Kernel mode



Process	Threads
(i) System called involved in process	(i) There is no system call involved
(ii) OS treats different process differently	(ii) All user level threads treated as single task for OS.
(iii) Different process have different copies of data, code, files.	(iii) Threads share same copy of code and data
(iv) Context switching is slower	(iv) Context switching is faster
(v) Blocking a process will not block another.	(v) Block entire process
(vi) Independent	(vi) Interdependent

User level thread	Kernel level thread
(i) User level threads are managed by user level	(i) Kernel level threads are managed by OS
(ii) User level threads are typically fast	(ii) Kernel level threads are slower than user level
(iii) Context switching is faster	(iii) Context switching is slower
(iv) If one user level thread performs blocking operation entire process gets blocked	(iv) If one kernel level thread is blocked, no effect on other kernels

Page No:
Date:

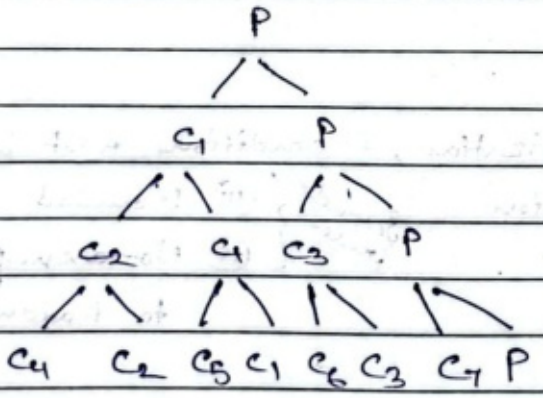
Context Switching Time

Process > Kernel level thread > User level thread

(i) PCB holds information about a process
(ii) The exe code is loaded into main memory
(iii) The stack and heap area is also assigned

(i) fork() is a system call. (Create()) This system call creates a child process which is a replica of the parent.
(ii) fork() returns 0 to the child process and process id of the child process to the parent.
(iii) The execution to the child process starts next to the fork() command.
(iv) In UNIX OS every hardware device is considered as a file
(v) Every process will have a process ID (in PCB) → starts from 1
(vi) Parent returns 0 to the child

P_id for parent ⇒ P_id returned by child
P_id for child ⇒ 0 returned by parent



child = 7
Total = 8

No. of childs = $2^n - 1$

```
int a, Pid
for i=1 to 3
do {
  fork();
}
```

≡ fork() fork()

Process Synchronization

Process Synchronization is the coordination of execution of multiple processes in a multi process system to ensure that they access shared resources in a controlled and predictable manner. It aims to resolve the problems of race conditions and other synchronization issues in a concurrent system.

Process Synchronization

- **Independent** → The execution of one process doesn't affect the execution of other processes.
- **Cooperative** → A process that can affect or be affected by other processes executing in the system.

Critical Section Problem

A critical section is a code segment that can be accessed by only one process at a time. It contains shared variables that need to be synchronized to maintain the consistency of data variables.

To achieve synchronization, 4 conditions must be satisfied:

- Primary rules:
  - (I) Mutual exclusion
  - (II) Progress
- Secondary rules:
  - (III) Bounded wait
  - (IV) No assumption related to hardware & speed

- **Mutual exclusion**: If process P_i is executing in its critical section, then no other processes can be executing in their critical section.

- **Progress**: If no process is executing in its critical section and some processes wish to enter their critical section, then only those processes that aren't executing in their remainder sections can participate in deciding which

will enter the critical section next and this selection cannot be postponed indefinitely.

- Bounded waiting : There exists a bound, an 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.

Use of critical sections in a program can cause a number of issues including :

(i) Deadlock
(ii) Starvation
(iii) Overhead

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

```
do {
    flag = 1;
    while (flag);      // entry section
                       // critical section
    if (flag)
                       // remainder section
} while (true);
```

Page No:
Date:

Classic S/W soln: Peterson's solution

- Restricted to two processes
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted (not true for modern processors)
- The two threads share two variables:
  - int turn;
  - Boolean flag[2];
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process Pi is ready

Algorithm for process Pi

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);
        CRITICAL SECTION
    flag[i] = FALSE;
        REMAINDER SECTION
} while (TRUE);
```

① Mutual exclusion because only way thread enter critical section when flag[j] == FALSE or turn == TRUE
② Only way to enter section is by flipping flag[] inside loop
③ turn = j allows the other thread to make progress

Solution using TestAndSet

- Defination of TestAndSet

```
boolean TestAndSet (boolean *target) {
        boolean rv = *target;
        *target = TRUE;
        return rv;
        }
```

- Shared boolean variable lock., initialized to false
- Solution

```
do {
    while (TestAndSet (&lock))
        ; /* do nothing
        // critical section
    lock = FALSE;
        // remainder section
    } while (TRUE);
```

Solution using Swap

- Defination of Swap

```
void swap (boolean *a, boolean *b) {
    boolean temp = *a;
    *a = *b;
    *b = temp;
    }
```

- Shared boolean variable lock initialized to FALSE ; Each process has a local Boolean variable Key
- Solution :

```
do {
    Key = TRUE;
    while (Key == TRUE)
        Swap (&lock, &Key);
            // critical section
    Lock = FALSE;
            // remainder section
    } while (TRUE);
```

Solution with TestAndSet and bounded wait

- boolean waiting [n]; boolean lock; initialized to false

Pi can enter critical section iff waiting [i] == false or key == false

```
do {
    waiting [i] = TRUE;
    Key = TRUE;
    while (waiting [i] && Key)
        Key = TestAndSet (&lock);
    waiting [i] = FALSE;
        // critical section
    j = (i+1) % n;
    while ((j != i) && !waiting [j])
        j = (j+1) % n;
    if (j == i)
        lock = FALSE;
    else
        waiting [j] = FALSE;
        // remainder section
} while (TRUE);
```

Semaphore Synchronization ~~Techniq~~ Primitive

- Test And Set are hard to program for end users
- Introduce a simple function called semaphore
    - Semaphore is an integer S
    - Only legal operations on S are:
        - wait() [atomic] - if S > 0, decrement S else loop
        - Signal () [atomic] - increment S

```
• wait(S){
    while S <= 0
        ; // no-op
        S--;
    }
• signal (S) {
        S++;
    }
```

- Counting (S: is unrestricted), binary (mutex lock) (S: 0, 1)

Page No:
Date:

Semaphore

Binary Semaphore
This is also known as a mutex lock. It can only have two values - 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problems with multiple processes

Counting Semaphore
Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

Advantages of Process Synchronization
- Ensures data consistency and integrity
- Avoids race conditions
- Prevents inconsistent data due to concurrent access
- Supports efficient and effective use of shared resources

Disadvantages of process synchronization
- Adds overhead to the system
- Can lead to performance degradation.
- Increase in complexity of the system.
- Can cause deadlocks if not implemented properly.

Mutex Locks
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first acquire() a lock then release() the lock.
  - Boolean variable indicating if lock is available or not
- Calls to acquire() and release() must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires busy waiting
  - This lock therefore is called Spinlock

```
• acquire () {
      while (!available)
          ; /* busy wait */
          available = false;
          }
• release () {
          available = true;
          }
• do {
      acquire lock()
            // critical section
      release lock()
            // remainder section
  } while (true);
```

Semaphore implementation with no Busy waiting

```
typedef struct {
    int value;
    struct process * list;
    } semaphore;

wait (semaphore *S) {
    S-> value--;
    if (S-> value < 0) {
        add this process to S-> list;
        block();
        }
    }

signal (semaphore *S) {
    S-> value++;
    if (S-> value <= 0) {
        remove a process P from S-> list;
        wakeup (P);
        }
    }
```

Page No:
Date:

Producer Consumer Problem

Suppose we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer counter that keeps track of the number of full buffers. Initially counter is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {
    /* produce an item in next produced */
    while (counter == BUFFER_SIZE);
        /* do nothing */
    buffer [in] = next_produced;
    in = (in+1) % BUFFER_SIZE;
    counter ++;
}
```

Consumer

```
while (true) {
    while (counter == 0)
        ; /* do nothing */
    next_consumed = buffer [out];
    out = (out+1) % BUFFER_SIZE;
        counter --;
    /* consume the item in next consumed */
}
```

Page No:
Date:

Reader - Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers : Only read the dataset ; don't perform any updates
  - Writers : Can both read and write
- Problem - allow multiple readers to read at the same time.
  - Only one single writer can access the shared data at the shared time.
- Several variations of how readers and writers are considered all involve some form of priorities
- Shared data - Data set
  - Semaphore rw_mutex initialized to 1
  - Semaphore mutex initialized to 1
  - Integer read_count initialized to 0

Writer :

```
do {
    wait (rw_mutex);
    ...
    /* writing is performed */
    ...
    signal (rw_mutex);
} while (true);
```

Writer :

```
do {
    wait (mutex);
    read_count ++;
    if (read_count == 1)
        wait (rw_mutex);
    signal (mutex);
    ...
    /* reading is performed */
    ...
    wait (mutex);
    read_count --;
    if (read_count == 0)
        signal (rw_mutex);
    signal (mutex);
} while (true);
```

Reader - Writer Problem Variations

First variation : no reader kept waiting unless writer has permission to use shared object
Second variation : once writer is ready, it performs the write ASAP
Both may have starvation leading to even more variations
Problem is solved on some systems by kernel providing reader-writer locks

DINING PHILOSOPHERS PROBLEM

- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbours, occasionally try to pick 2 chopsticks (one at a time) to eat from bowl.
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

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

Deadlock Handling

- Allow atmost 4 philosophers to be sitting simultaneously at table
- Allow a philosopher to pick up the forks only if both are available (picking must be done in critical section)
- Use an assymetric solution -- an odd numbered philosophers picks up first the left chopstick and then right chopstick. Even numbered philosopher picks up first the right chopstick and then left chopstick

Advantages of semaphores

- Simple and effective mechanism for process synchronization
- Supports coordination b/w multiple processes
- Can be used to implement critical sections in program
- Used to avoid race conditions

Disadvantages of semaphores

- It can lead to performance degradation due to overhead associated with wait and signal operations.
- Can result in deadlock if used incorrectly
- It can cause performance issues in a program if not used properly.
- It can be prone to race conditions and other synchronization problems if not properly used

Monitors

Monitors are used for process synchronization. Monitors are defined as construct of programming language, which helps in controlling shared data access

* Characteristics

① Inside monitor, we can execute one process at a time
② A group of procedures and condition variables that are merged together in special type of module.
③ Offer high level of synchronization

Syntax

```
Monitor demo // Name of the monitor
{
variables ;
condition variables ;
procedure p1 {.... }
procedure p2 {.... }
}
```

Monitor	Semaphore
(i) We can use condition variables only in monitor	(i) In semaphore, we can use condition variables anywhere in the program but we cannot use condition variables in a semaphore.
(ii) Wait always block the caller	(ii) Wait doesn't always block the caller.
(iii) Monitors comprised of shared variables and procedures which operate the shared variable.	(iii) The semaphore's values means the number of shared resources that are present in the system.
(iv) Condition variables present in monitor	(iv) Condition variables not present in semaphore

Dining Philosophers problem using Monitor

"Five philosophers sit around a circular table. Each philosopher spends his life alternatively thinking and eating. In the center of a table is a large plate of spaghetti. A philosopher needs two forks to ~~help~~ eat a helping of spaghetti. Unfortunately, as philosophy is not as well paid as computing, the philosophers can only afford five forks. One fork is placed between each pair of philosopher and they agree that each will only use the fork to his immediate right and left."

↳ N philosophers seated around a circular table
- There is one fork b/w each philosopher
- To eat, a philosopher ~~must pick up their~~ must pick up their two nearest forks.
- A philosopher must pick up one fork at a time, not both at ones.

Page No:
Date:

Possibility of Deadlock

If philosophers take one fork at a time, taking a fork from the left and then one from the right, there is a danger of deadlock. This possibility of deadlock means that any solution to the problem must include some provision for preventing or otherwise dealing with deadlocks.

Possibility of Starvation

If philosophers take two forks at a time, there is a possibility of starvation. Philosophers P_2 and P_5 and P1 & P_3 can alternate in a way that starves out philosopher P_4. This possibility of starvation means that any solution to the problem must include some provision for preventing starvation.

Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
  enum (THINKING, HUNGRY, EATING) state [5];
  condition self [5];

  void pickup (int i) {
        state [i] = HUNGRY;
        test (i);
        if (state [i] != EATING) self [i].wait;
  }

  void pushdown (int i) {
        state [i] = THINKING;
              // test left and right neighbours
        test ((i+4) %5);
        test ((i+1) %5);
  }
```

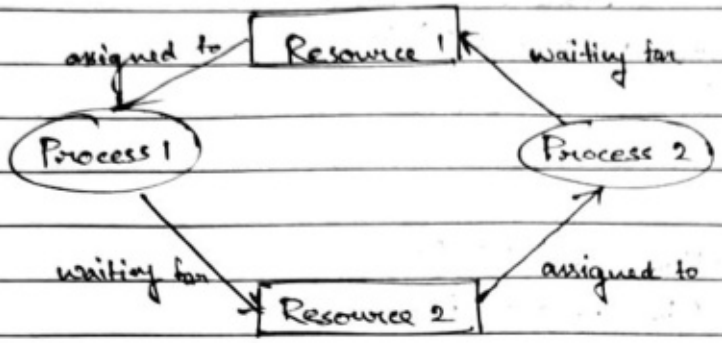
```
void test (int i) {
    if ((state[(i+4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i+1) % 5] != EATING)) {
            state[i] = EATING;
        self[i].signal();
        }
    }

    initialization_code() {
        for (int i=0; i<5; i++)
        state[i] = THINKING;
        }
}
```

Page No:
Date:

DEADLOCK

A deadlock is a situation where a set of processes are blocked because each process is holding a resource and for another resource acquired by some other process.



- System has finite number of resources to be distributed among number of competing processes.
- Resource types $R_1, R_2, \ldots R_m$. Example of resources CPU cycles, memory space I/O cycles
- Resources may be partitioned into some number of identical resources called instances. Thus each resource R_i has W_i instances.
- Each process utilizes a resource as follows: request, use and release

- Deadlock can arise if the following four conditions hold simultaneously (Neccessary Condition)

① Mutual Exclusion - Two or more resources are non-shareable (only one process can use at a time)

② Hold and wait - A process is holding atleast one resource and waiting for resources

③ No preemption - A resource cannot be taken from a process unless the process releases the resource.

④ Circular Wait - A set of processes waiting for each other in circular form.

Methods for handling deadlock

① Deadlock prevention and avoidance

Prevention - The idea is to not let the system into a deadlock state. Prevention is done by negating one of the above mentioned neccessary conditions for deadlock. Prevention can be done in four different ways:

- Eliminate mutual exclusion
- Solve hold and wait
- Allow preemption
- Circular wait solution

Avoidance - Avoidance is kind of futuristic. By using the strategy of 'Avoidance', we have to make an assumption. We need to ensure that all information about resources that the process will need is known to us before the execution of the process. We use Banker's algorithm (which in turn is a gift from Dijkstra) to avoid deadlock

⑧ IN PREVENTION AND AVOIDANCE, WE GET CORRECTNESS OF DATA BUT PERFORMANCE DECREASES

② Deadlock detection and recovery

It consists of two phases

- In first phase, we examine the state of the process and check whether there is a deadlock or not in the system.
- If found deadlock in the first phase then we apply the algorithm for the recovery of deadlock.

Page No:
Date:

RECOVERY FROM DEADLOCK

① Manual Intervention - When deadlock is detected, one option is to inform the operator and let them handle the situation manually. This approach allows human judgement and decision making. It may be time consuming and not feasible in large scale operations.

② Automatic Recovery - This approach allows the system to recover from deadlock automatically. This method involves breaking the deadlock cycle by either aborting processes or preempting resources.

③ Deadlock Ignorance

If a deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX takes. We use Ostrich algorithm for deadlock ignorance.

Page No:
Date:

Bankers Algorithm

It is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for the predetermined maximum possible amounts of all resources, then makes an "S-State" check to test for possible activities, before deciding whether allocation should be allowed to continue

- Advantages

(i) It contains various resources that meet requirements of each process

(ii) This algorithm has max resource attribute that represents indicates each process can hold the max. no. of resource in the system.

- Disadvantages

(i) It requires a fixed no. of processes and no additional process can be started in the system while executing the system.

(ii) Each process has to know and state their maximum resource requirement in advance for the system.

The following data structures are used to implement the Banker's algorithm.

(i) Available

- It is the 1D Array of size m indicating the number of available resource of each type
- Available [j] = K means that there are K instances of resource type Rj

(ii) Max

- It is the 2D Array of size 'n * m' that defines the maximum demand of each process in the system
- Max [i, j] = K means process Pi may request atmost 'K' instances of resource type Rj

(iii) Allocation
- It is the 2D array of size n * m that defines the number of resources of each type currently allocated to each process
- Allocation [i,j] = k means process P_i is currently allocated 'k' instances of resource type R_j

(iv) Need
- It is a 2D Array of size 'n * m' that indicates the remaining resource need of each process
- Need [i,j] = k means process P_i currently needs 'k' instances of resource types R_j
- Need [i,j] = Max [i,j] - Allocation [i,j]

Safety Algorithm

It is the safety algorithm used to check whether or not a system is in a safe state or follows the safe sequence in a bankers algorithm:

(i) There are two vectors Work and Finish of length m and n in a safety algorithm
Initialize : Work = Available
Finish [i] = false ; for i = 0, 1, 2, 3, 4 ... n-1

(ii) Check the availability status for each type of resources[i] such as
Need [i] <= Work
Finish [i] == false
* If the i doesn't exist, then go for step 4

(iii) Work = Work + Allocation (i) // to get new resource allocation
Finish [i] = true
* Go to step 2 to check the status of resource availability for the next process

(iv) If finish [i] = true ; it means system safe for all process

Page No:
Date:

Resource Request Algorithm

① If Request$_i$ <= Need
Go to step (2); otherwise, raise an error condition, since the process has executed its maximum claim.

② If Request$_i$ <= Available
Go to step (3); otherwise P_i must wait, since the resources aren't available

③ Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:
Available = Available - Request$_i$
Allocation$_i$ = Allocation$_i$ + Request$_i$
Need$_i$ = Need$_i$ - Request$_i$

Q Consider a system that contains five processes P_1, P_2, P_3, P_4, P_5 and three resource types A, B, C. Following are resource types: A has 10, B has 5 and resource type C has 7 instances.

Process	Allocations			Max			Available		
	A	B	C	A	B	C	A	B	C
P_1	0	1	0	7	5	3	3	3	2
P_2	2	0	0	3	2	2			
P_3	3	0	2	9	0	2			
P_4	2	1	1	2	2	2			
P_5	0	0	2	4	3	3			

(a) What is the reference of the need matrix?
(b) Determine if the system is safe or not?
(c) What will happen if the resource request (1,0,0) for process P_1 can the system accept this request immediately?

Page No:
Date:

(a) Need [i] = Max [i] - Allocation [i]

Need for P_1 : (7,5,3) - (0,1,0) = 7,4,3

Need for P_2 : (3,2,2) - (2,0,0) = 1,2,2

Need for P_3 : (9,0,2) - (3,0,2) = 6,0,0

Need for P_4 : (2,2,2) - (2,1,1) = 0,1,1

Need for P_5 : (4,3,3) - (0,0,2) = 4,3,1

Process	Need		
	A	B	C
P_1	7	4	3
P_2	1	2	2
P_3	6	0	0
P_4	0	1	1
P_5	4	3	1

(b) Available resource of A, B and C are 3, 3 and 2

Step 1 : For process P_1

Need <= Available

7,4,3 <= 3,3,2 condition is false

So we examine another process P_2

Step 2 : For process P_2

Need <= Available

1,2,2 <= 3,3,2 condition is true

New available = available + allocation

(3,3,2) + (2,0,0) => 5,3,2

Step 3 : For process P_3

Need <= Available

(6,0,0) <= (5,3,2) condition is false

Step 4 : For process P_4

Need <= Available

0,1,1 <= 5,3,2 condition is true

New available = available + allocation

(5,3,2) + (2,1,1) => 7,4,3

Page No:
Date:

Step 5: For ~~sta~~ process P_5
Need <= Available
4,3,1 <= 7,4,3 condition is true
New available = Available + Allocation
(7,4,3) + (0,0,2) => (7,4,5)

** Now we examine each type of resource request for processes P_1 and P_3

Step 6: For process P_1
Need <= Available
(7,4,3) <= (7,4,5) condition is true
New available = Available + Allocation
(7,4,5) + (0,1,0) ⇒ (7,5,5)

Step 7: For process P_3
Need <= Available
(6,0,0) = Available + Allocation
(7,5,5) + (3,0,2) ⇒ (10,5,7)

P_2, P_4, P_5, P_1, P_3

(c) For granting the request (1,0,2), first we have to check that Request <= Available that is (1,0,2) <= (3,3,2) since the condition is true. So the process P_1 gets the request immediately.

Page No:
Date:

Memory Management

- What is memory ?

Computer memory can be defined as a collection of some data represented in the binary format

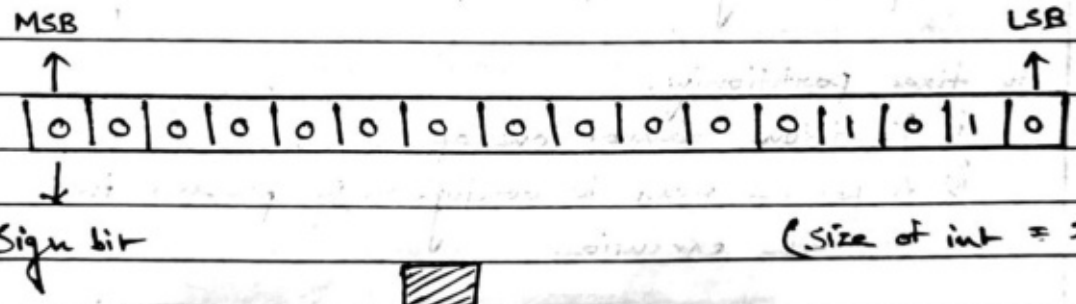
A computer device that is capable to store any information or data temporally or permanantly is called storage device.

- How data is being stored in a computer system ?

Machine understands only binary language i.e. 0 and 1. Computer converts every data into binary language first and then stores into the memory.

int x = 10

computer converts it into binary language and then store into the memory block.



The binary representation of 10 is 1010. Here we are considering a 32 bit system, so the size of int is 2 bytes i.e. 16 bit. 1 memory block stores 1 bit. If we are using signed integer then the MSB in the memory array is a signed bit.

Signed bit value 0 represents positive integer while 1 represents negative integer. Range varies from -32768 to +32767

Page No:
Date:

- **Need for multiprogramming**

CPU can directly access the main memory, registers and cache of the system. Program always executes in the main memory. Size of main memory affects degree of multi programming. If the size of main memory is larger than CPU can load more processes in main memory at some time and therefore will increase degree of multi programming as well as CPU utilization.

- **FIXED PARTITIONING**

The earliest and one of the simplest technique which can be used to load more than one process into the main memory is called fixed partitioning:

In this technique the main memory is divided into partitions of equal or different sizes. The OS always resides in 1st partition and other partitions can be used to store user processes. The memory is assigned to the processes in contiguous way.

In fixed partitioning,

i) Partitions cannot overlap

ii) A process must be contiguously present in the partition for the execution.

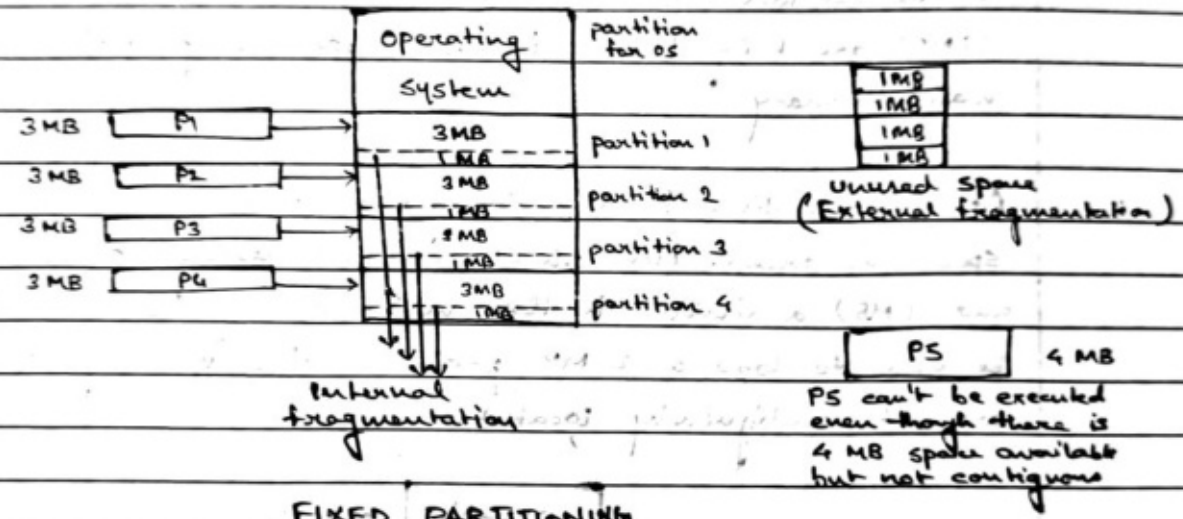
*** Disadvantages**

① **Internal fragmentation**: If the size of the process is lesser than the total size of the partition then some size of the partition get wasted and remain unused. This is wastage of memory and called internal fragmentation.

② **External fragmentation**: The total unused space of various partitions cannot be used to load the processes even though there is space available but not in contiguous form.

③ Limitation on the size of the process : If the process size is ~~larger~~ larger than the size of maximum sized partition then that process cannot be loaded into the memory. Therefore, a limitation can be imposed on the process size that is it cannot be larger than the size of larger partition

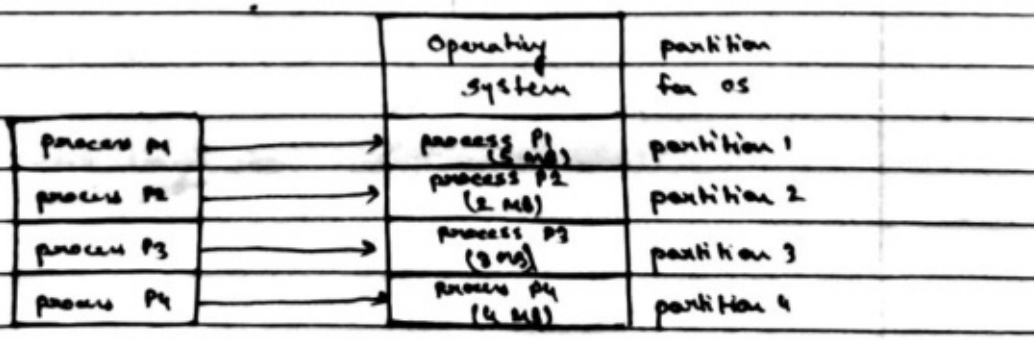
④ Degree of multiprogramming is less : It is fixed and very low due to the fact that the size of the partition cannot be varied according to the size of process



FIXED PARTITIONING
(contiguous memory allocation)

DYNAMIC PARTITIONING

It tries to overcome the problems of fixed partitioning. In this technique the partition size is not declared initially but is declared at the time of process loading.



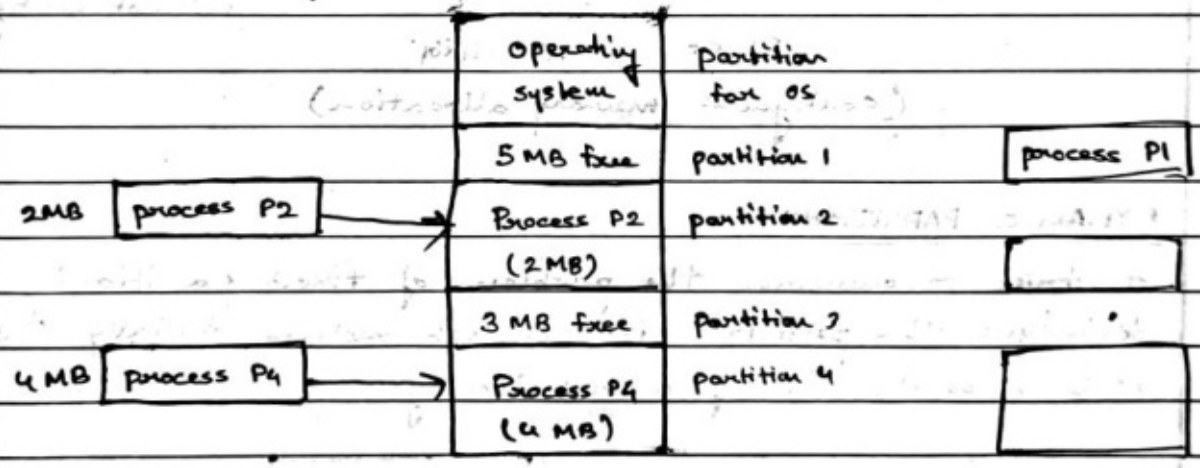
- Advantages of dynamic partitioning over fixed partitioning
  1. No internal fragmentation
  2. No limitation on the size of process
  3. Degree of multiprogramming is dynamic

- External fragmentation

  Absence of internal fragmentation doesn't mean that there won't be any external fragmentation

Let us consider three processes P1 (1MB) and P2 (3 MB) and P3 (1MB) are being loaded in the respective partitions of the main memory

After some time P1 and P3 got completed and their assigned space is freed. Now there are two unused portions (1MB and 1MB) available in the main memory but they cannot be used to load a 2 MB process in the memory since they are not contiguously located.



Page No:
Date:

Algorithms for partition allocation
- First fit algorithm
- Best fit algorithm
- Worst fit algorithm

- **FIRST FIT ALGORITHM**
  - This algorithm starts scanning the partitions serially from starting
  - When an empty partition that is big enough to store the process is found, it is allocated to the process
  - Obviously the partition size has to be greater than or at least equal to the process size.

- **BEST FIT ALGORITHM**
  - This algorithm first scans all the empty partitions
  - It then allocates the smallest size partition to the process

- **WORST FIT ALGORITHM**
  - This algorithm first scans all the empty partitions
  - It then allocates the largest size to the process

* For static partitioning
  - Best fit algorithm works best
  - This is because space left after allocation inside the partition is of very small size
  - Thus internal fragmentation is least

* For static partitioning
  - Worst fit algorithm works worst
  - This is because space left after the allocation inside the partition is of very large size
  - Thus internal fragmentation is maximum

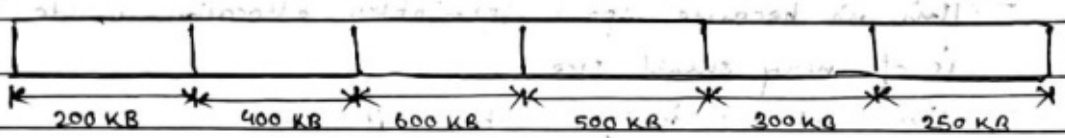
Page No:
Date:

* for dynamic partitioning
  - Worst fit algorithm works best
  - This is because space left after allocation inside partition is of large size
  - There is high probability that this space might suit the requirement of arriving processes

* for dynamic partitioning
  - 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

Q. 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
- First fit algorithm
- Best fit algorithm
- Worst fit algorithm



main memory

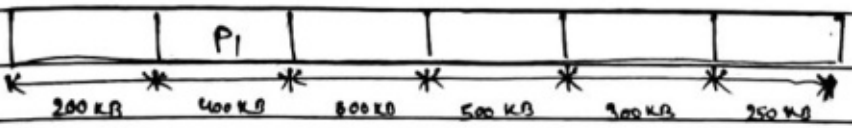
- FIRST FIT ALGORITHM

P1 = 357 KB

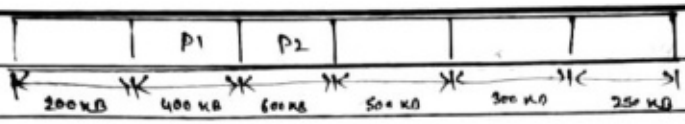
P2 = 210 KB

P3 = 468 KB

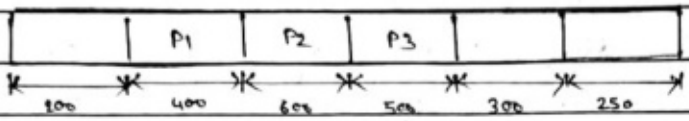
P4 = 491 KB



main memory



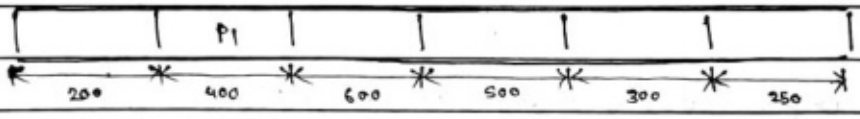
main memory



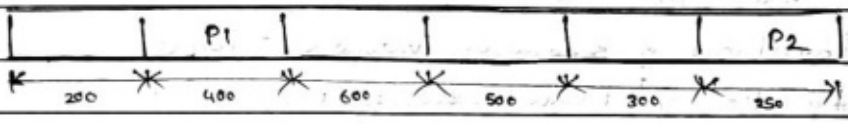
main memory

• P4 can't be allocated the memory. This is bcoz no partition of size greater than or equal to size of process P4 is available

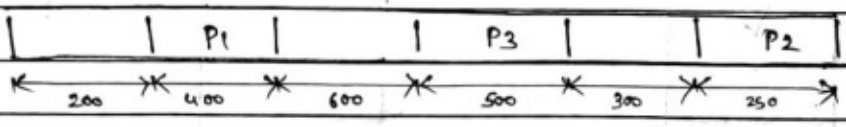
• BEST FIT ALGORITHM



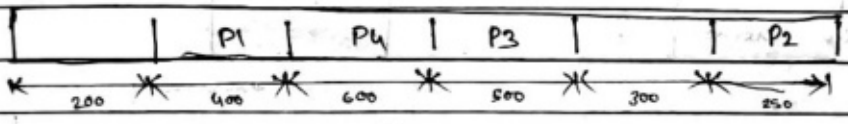
main memory



main memory

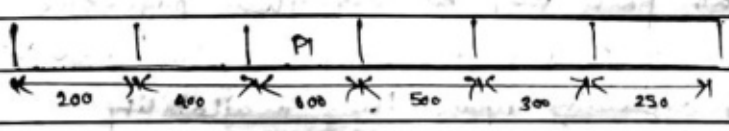


main memory

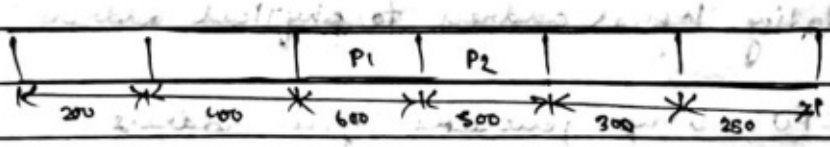


main memory

• WORST FIT ALGORITHM



main memory



main memory

P3 and P4 can't be allocated the memory. This is bcoz no partition of size greater than or equal to the size of process P3 and process P4 is available.

Page No:
Date:

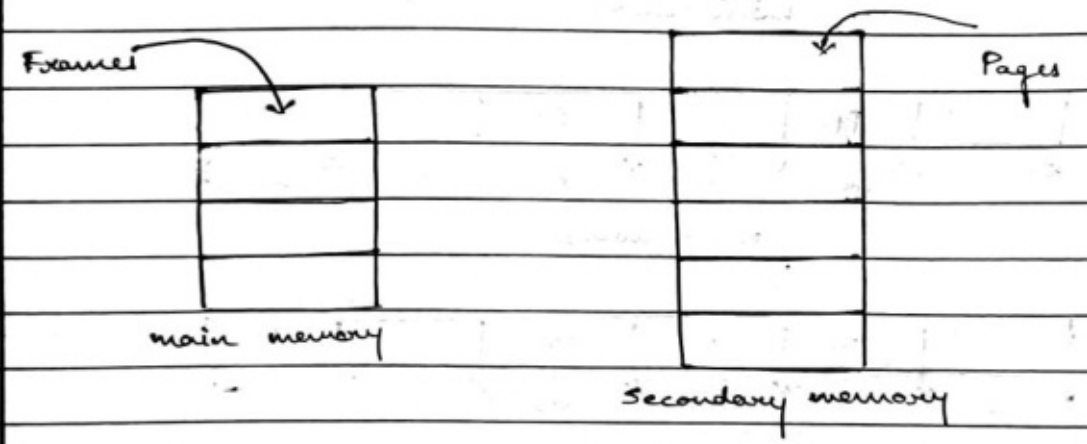
Non contiguous memory allocation

- It is a memory allocation technique
- It allows to store parts of a single process in a non contiguous fashion
- Thus different parts of the same process can be stored at different places in main memory

Techniques → Paging
→ Segmentation

Paging

- Paging is a fixed size partitioning scheme
- Secondary and main memory are divided into equal fixed size partitions
- Partitions of secondary memory : Pages
- Partitions of main memory : Frames



- Each process is divided into parts where size of each part is same as page size
- Size of last part may be less than page size
- The pages of process are stored in the frames of main memory depending upon their availability

Translating logical address to physical address :

- CPU always generates logical address
- Physical address required to access main memory

Page No:
Date:

CPU generates logical address consisting of two parts:
(i) Page Number
(ii) Page Offset

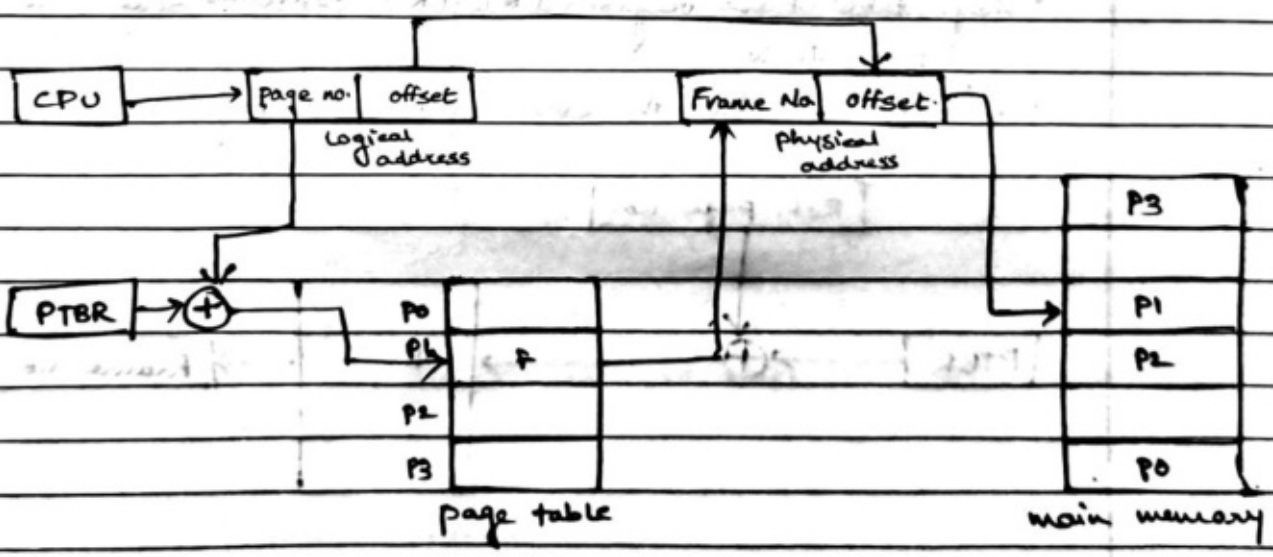
* Page Number: Specifies the specific page of the process from which CPU wants to read the data.

* Page Offset: Specifies the specific word on the page that CPU wants to read.

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 main memory.

Frame number combined with 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.



Page No:
Date:

- **Advantages of Paging**
  - It allows to store parts of a single process in a non-contiguous manner
  - It solves the problem of external fragmentation.

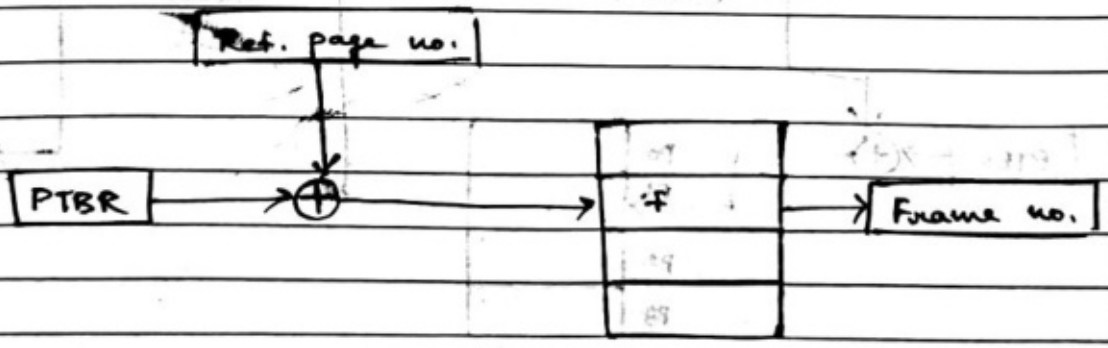
- **Disadvantages of Paging**
  - It suffers from internal fragmentation
  - There is an overhead of maintaining a page table for each process.
  - The time taken to ~~~~ fetch the instruction increases since now two memory accesses are required.

Page Table

- Page table is a data structure
- It maps the page number referenced by the CPU to the frame number where page is stored.

Characteristics

- Page table stored in main memory
- No. of entries in a page table = No. of pages in which the process is divided
- Page table Base register (PTBR) contains the base address of page table
- Each process has its own independent page table



Obtaining frame number from page table

- Page table base register (PTBR) provides the base address of the page table.
- The base address of the page table is added with the page number referenced by the CPU
- It gives the entry of the page table containing the frame number where the referenced page is stored.

compulsory | optional field

Frame No.	Present/Absent	Protection	Reference	Caching	Dirty

① Frame Number

Frame Number specifies the frame where the page is stored in the main memory. Number of bits in frame number depends on the number of frames in the main memory

② Present / Absent bit

Sometimes called valid / invalid bit. It specifies whether that page is present in the main memory or not. If page is not present in main memory, then this bit is set to 0 otherwise set to 1.

③ Protection bit

Sometimes called read / write bit. It is concerned with the page protection. Specifies permission to perform read and write operation on the page.

If only read operation allowed → 0

If both read / write allowed → 1

④ Reference bit

Reference bit specifies whether the page has been referenced in the last clock cycle or not

⑤ Caching Enabled / Disabled

This bit enables / disables the caching of page. Whenever freshness in the data is required, then caching is disabled using this bit

Caching disabled → 0 otherwise → 1

Page No:
Date:

⑥ Dirty bit

Sometimes called as dirty bit. It specifies whether the page has been modified or not. If page has been modified then this bit is set to 1 otherwise 0

FOR MAIN MEMORY

- Physical Address Space = Size of main memory
- Size of main memory = Total no. of frames × Page size
- Frame size = Page size
- No. of frames in main memory = 2^x, then no. of bits in frame number = x bits
- If page size = 2^x bytes, then no. of bits in page offset = x bits
- If size of main memory = 2^x bytes, then no. of bits in physical address = x bits

FOR PROCESS

- Virtual address space = Size of process
- No. of pages the process is divided = Process size / Page size
- If process size = 2^x bytes, then number of bits in virtual address space = x bits

FOR PAGE TABLE

- Size of page table = No. of entries in page table × Page table entry size
- No. of entries in pages table = No. of pages the process is divided
- Page table entry size = No. of bits in frame number + No. of bits used for optional fields (if any)

* If given address consists of 'n' bits, then using n bits, 2^n locations are possible

* Size of memory = 2^n × size of one location

Page No:
Date:

* If the memory is byte addressable, then size of one location = 1 byte. Thus size of memory = 2^n bytes

* If the memory is word addressable where 1 word = m bytes, then size of one location = m bytes. Thus size of memory = $2^n \times m$ bytes

Q. Calculate the size of memory if its address consists of 22 bits and the memory is 2 bit addressable

No. of locations possible with 22 bits = 2^{22} locations

Size of one location = 2 bytes

Size of memory = $2^{22} \times 2$ bytes

= 2^{23} bytes = 8 MB

Q. Calculate the no. of bits required in the address for memory having size of 16 GB. Assume the memory is 4 byte addressable

let n number of bits are required

Size of memory = $2^n \times 4$ bytes

Given size of memory = 16 GB

$\therefore 2^n \times 4 = 16\ G = 2^{34}$

$\Rightarrow 2^n \times 2^2 = 2^{34}$

$\Rightarrow \boxed{n = 32 \text{ bits}}$

Q. Consider a system with byte addressable memory, 32 bit logical addresses, 4 kilobyte page size and page table entries of 4 bytes each. The size of the page table in the system in megabytes is ?

No. of bits in logical address = 32 bits

Page size = 4 KB

Page table entry size = 4 bytes

No. of bits in logical address = 32 bits

Process size = 2^{32} B = 4 GB

No. of pages the process is divided = Process size / Page size

= 4GB / 4 KB

= 2^{20} pages

No. of entries in page table = 2^{20} entries

Page No:
Date:

Page table size = No. of entries in page table × Page table entry size

$= 2^{20} \times 4$ bytes

$= 2^{22}$ bytes

$= 4$ MB

Q. Consider a machine with 64 MB physical memory and a 32 bit virtual address space. If the page size is 4 KB. What is the approximate size of page table ?

Size of main memory = 64 MB

No. of bits in virtual address space = 32 bit

Page size = 4 KB

Size of main memory = 64 MB = 2^{26} B

∴ No. of bits in physical address = 26 bit

No. of frames in main memory = Size of main memory / Frame size

$= 64$ MB / 4 KB

$= 2^{26}$ B / 2^{12} B

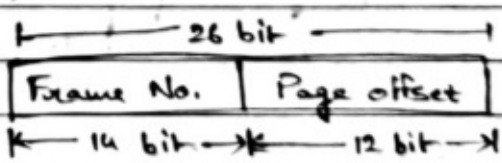
$= 2^{14}$

∴ No. of bits in frame number = 14 bits

Page size = 4 KB = 2^{12} B

∴ No. of bits in page offset = 12 bits

Physical Address :



Process size = 2^{32} B = 4GB

No. of pages the process is divided = Process size / Page size

$= 4$GB / 4 KB

$= 2^{20}$ pages

∴ No. of entries in page table = 2^{20} entries

Page No:
Date:

Page table size = No. of entries in page table × Page table entry size
= No. of entries in page table × No. of bits in frame number
= 2^{20} × 14 bits
= 2^{20} × 16 bytes (approx. 14 bits ≈ 16 bits)
= 2^{20} × 2 bytes
= 2 MB

Q In a virtual memory system, size of virtual address is 32 bit, size of physical address is 30 bit, page size is 4 KB and the size of each page table entry is 32 bit. The main memory is byte addressable. What is the max. number of bits that can be used for storing protection and other information in each page table entry?

No. of bits in virtual address = 32 bit
No. of bits in physical address = 30 bit
Page size = 4 KB
Page table entry size = 32 bits

Size of main memory = 2^{30} B = 1 GB

No. of frames in main memory = Size of main memory / Frame size
= 1 GB / 4 KB
= 2^{30} B / 2^{12} B
= 2^{18}

∴ No. of bits in frame number = 2^{18} bits

Max. number of bits that can be used for storing protection and other information = Page table entry size − No. of bits in (FN)
= 32 bits − 18 bits
= 14 bits

Page No:
Date:

Q Consider a single level paging scheme. The virtual address space is 4 MB and the page size is 4 KB. What is the maximum page table entry size possible such that the entire page table fits well in one page?

No. of pages the process is divided = Process size / Page size
= 4 MB / 4 KB
= 2^{10} pages

Let page table entry size = B bytes
∴ Page table size = 2^{10} × B bytes
For page table to fit well in one page, we must have,
Page table size <= page size
2^{10} × B bytes <= 4 KB
2^{10} × B <= 2^{12}
B <= 4
∴ Max. page table entry size possible = 4 bytes

Q Consider a single level paging scheme. The virtual address is 16 GB and page table entry size is 4 bytes. What is the minimum page size possible such that entire page table fits well in one page?

Let page size = B bytes
No. of pages the process is divided
= Process size / Page size
= 16 GB / B bytes
= 2^{34} / B

Page table size = No. of pages the process is divided × Page table entry size
= (2^{34} / B) × 4 Bytes
= (2^{36} / B) bytes
According to condition, for page table to fit well in one page,
page table size <= page size
(2^{36} / B) <= B bytes
B >= 2^{18}
Min. page size possible = 2^{18} bytes = 256 KB

TRANSLATION LOOKASIDE BUFFER

- Translation lookaside Buffer (TLB) is a solution that tries to reduce the effective access time.
- Being a hardware, the access time of TLB is very less as compared to the main memory.

■ It consists of two columns:

1) Page Number
2) Frame Number

In a paging scheme using TLB, the logical address generated by the CPU is translated into the physical address using following steps:

(Step 1) CPU generates a logical address consisting of two parts

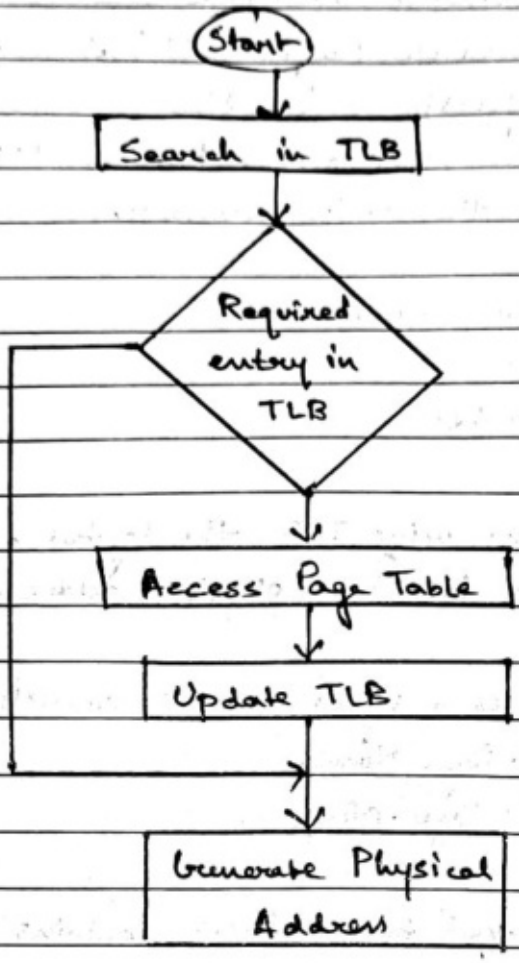
1. Page Number
2. Page Offset

(Step 2) TLB is checked to see if it contains an entry for the referenced page number. The referenced page number is compared with the TLB entries all at once.

(i) If there is a TLB hit
- TLB entry is used to get the corresponding frame number for referenced page number

(ii) If there is a TLB miss
- Page table is used to get the corresponding frame number for the referenced page number. Then TLB is updated with the page number and frame number for future reference.

(Step 3) After frame number is obtained, it is combined with the page offset to generate the physical address. Then physical address is used to read the required word from the main memory.



- 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 again updated with currently running process

* When a new process gets scheduled
  - Initially TLB is empty. So TLB misses are frequent
  - With every access from page table, TLB is updated
  - After sometime, TLB hit increases and TLB miss reduces

* 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.

• Advantages of using TLB
- Reduces the effective access time
- Only one memory access is required when TLB hit occurs

• Disadvantages of using TLB
- After some time of running the process, when TLB hit increases and process starts to run smoothly, context switching occurs
- Entire content of TLB is flushed. Then TLB is updated with currently running process

Effective Access Time

Hit ratio of TLB × {Access time of TLB + Access time of main memory}
+
Miss ratio of TLB × {Access time of TLB + 2R Access time of MM}

Multilevel Paging (Hierarchical Paging)

Multilevel paging is a paging scheme where there exists a hierarchy of page tables

• Need: The need for multilevel paging arises when -
  • Size of page table is greater than frame size
  • Page table can't be stored in single frame in main memory

Consider a system using multilevel paging scheme. The page size is 1 GB. The memory is byte addressable and virtual address is 72 bits long. The page table entry size is 4 bytes. Find:
• How many levels of page table will be required?
• Give the divided physical address and virtual address

Given: Virtual address = 72 bits
Page size = 1 GB
Page table entry size = 4 bytes

• No. of bits in frame number
Page table entry size = 4 bytes = 32 bits
∴ No. of bits in frame size = 32 bits

Page No:
Date:

Number of ~~bits~~ frames in main memory = 2^{32} frames

Size of main memory = Total no. of frames * Frame size
= $2^{32} \times 1$ GB
= 2^{62} B

No. of bits in physical address = 62 bits

Page size = 1GB = 2^{30} B
No. of bits in page offset = 30 bits

No. of pages of process = Process size / Page size
= 2^{72} B / 1 GB
= 2^{72} B / 2^{30} B = 2^{42} pages

Inner page table size = No. of pages of process * Page table entry size
= $2^{42} \times 4$ bytes
= 2^{44} bytes

No. of pages the inner page table is divided
= Inner page table size / page size
= 2^{44} B / 1 GB
= 2^{44} B / 2^{30} B
= 2^{14} pages

No. of page table entries in one page of inner page table
= Page size / Page table entry size
= 1 GB / 4 B
= 2^{30} B / 2^{2} B
= 2^{28} entries

No. of bits required to search a particular entry in one page of inner page table = 28 bit

Page No:
Date:

Outer table page size = No. of entries in outer page table × Page table entry size

= No. of pages the inner table is divided × Page table entry size

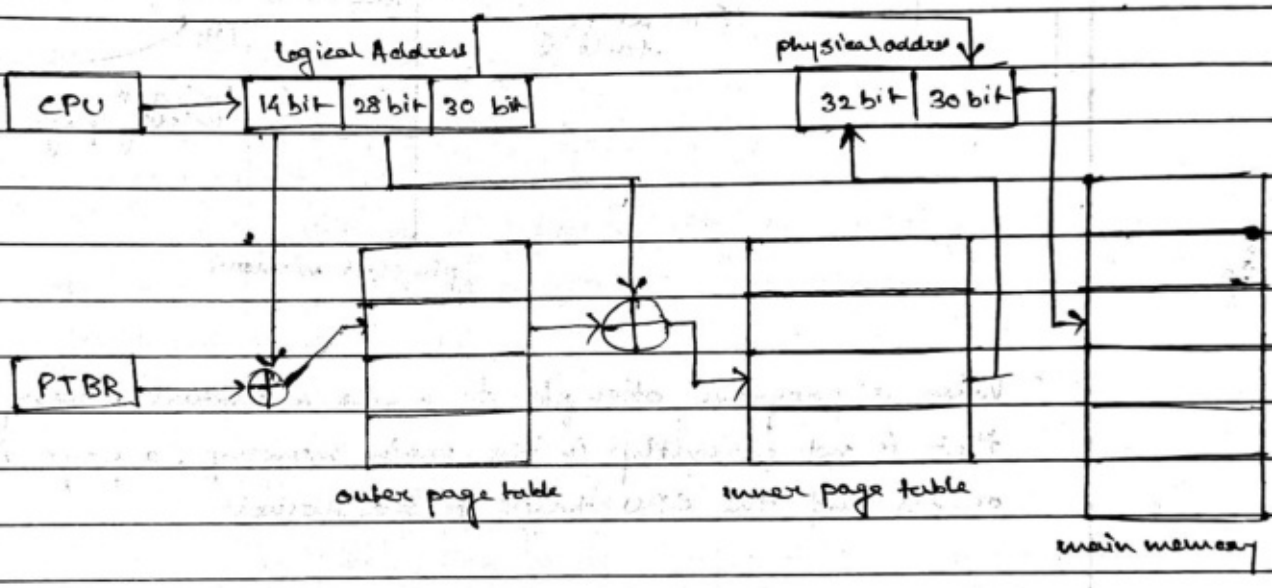
= 2^{14} × 4 bytes

= 2^{16} bytes

= 64 KB

Outer page table contains 2^{14} entries

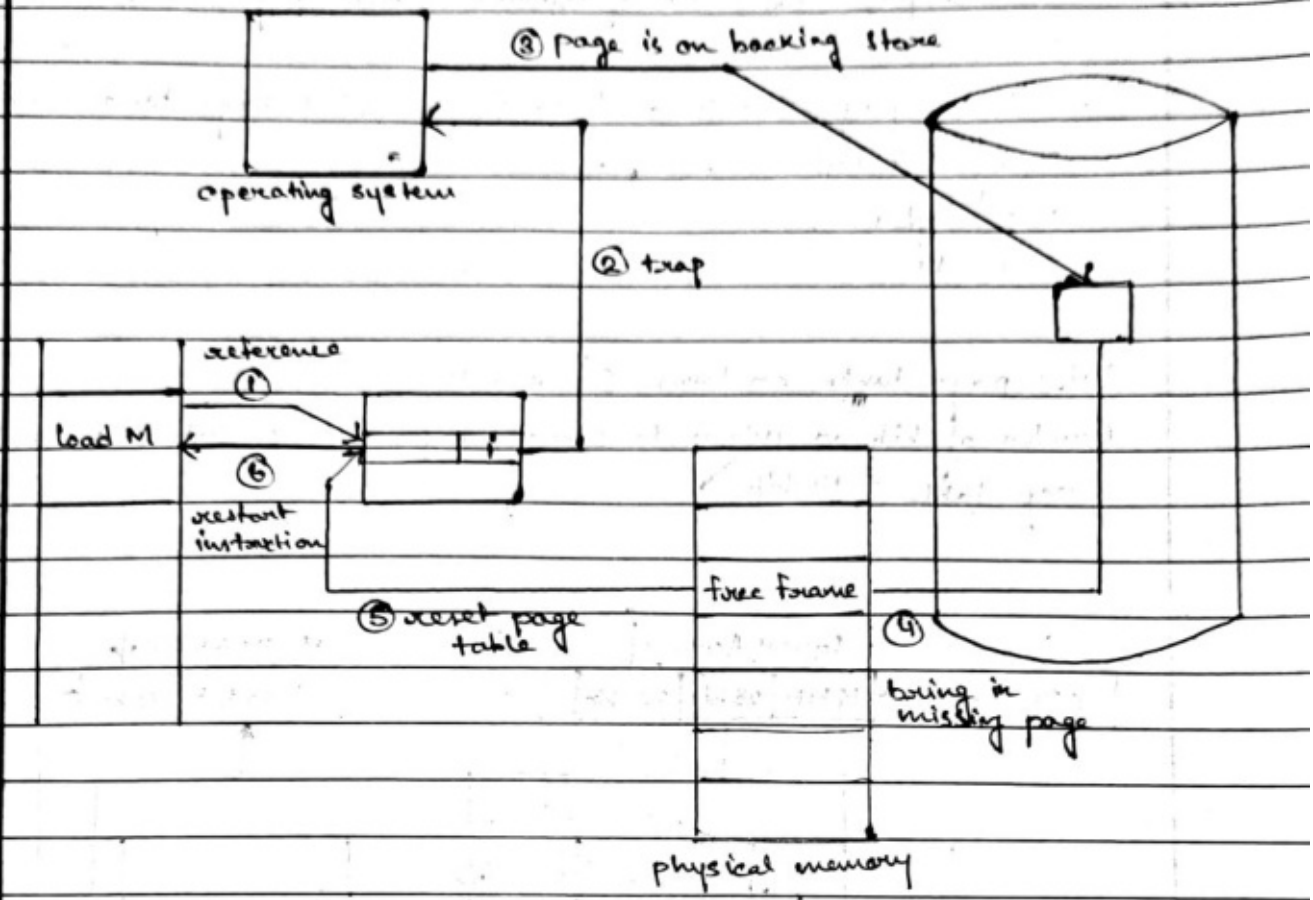
Number of bits required to search a particular entry in outer page table = 14 bits



Page Fault

- When a page referenced by the CPU is not found in the main memory, it is called as page fault.
- When a page fault occurs, the required page has to be fetched from secondary memory into main memory.

A page fault occurs when a program attempts to access data or code that is in its address space, but is not currently in the system RAM.



- When a program attempts to access a virtual memory address that is not currently in the main memory, a page fault occurs and the CPU traps to the kernel

- The operating system saves the current state of the program including the program counter (PC) and CPU registers on the stack

- Operating system determines which virtual memory page caused the fault and checks if the address is valid and accessible

- If the address is valid, the operating system tries to find a free physical memory frame to load the requested page

- If no free frames are available, the page replacement algorithm is used to select a page to be swapped out to disk.

- If the selected page is modified (dirty), it is written to disk
- The OS reads the requested page from disk into the newly available physical memory frame.
- Page tables are updated to map the virtual address to the new physical frame.
- The programs state including CPU registers and PC is restored to the point before page fault occured.
- The program resumes execution with requested memory page now loaded in main memory.

Page replacement

It is the process of swapping out an existing page from the frame of the main memory and replacing it with the required page

Page replacement is required when -

- All the frames of main memory are already occupied
- Thus, a page has to be replaced to create a room for the required page.

Page replacement algorithms

It helps to decide which page must be swapped out from the main memory to create a room for incoming page.

- FIFO
- LIFO
- LRU.
- Optimal
- Random

Page No:
Date:

- FIFO Page replacement algorithm
  - Works on the principle of FIRST IN FIRST OUT
  - Replaces the oldest page that has been present in the main memory for longest time
  - Implemented by keeping track of all pages in the queue.

- LIFO Page replacement algorithms
  - Works on the principle of LAST IN FIRST OUT
  - Replaces the newest page that arrived at last in the main memory
  - Implemented by keeping track of all pages in the stack.

- LRU Page Replacement algorithms.
  - Works on the principle of LEAST RECENTLY USED
  - It replaces the page that hasn't been reffred by the CPU for the longest time.

- Optimal Page Replacement Algorithm
  - This algorithm replaces the page that won't be reffered by the CPU in future for the longest time.
  - Practically impossible to implement this algorithm.
  - This is because pages that won't be used in future for the longest time can't be predicted.
  - However, it is best known algorithm and gives the least number of page fault.
  - Hence it is used as a performance measure criterion for other algorithms.

Q A system uses 3 page frames for storing process pages in main memory. It uses FIFO page replacement policy. Assume that all the page frames are initially empty. What is the total no. of page faults that will occur while processing the page reference string given below -

4, 7, 6, 1, 7, 6, 1, 2, 7, 2

Also calculate the hit and miss ratio.

Page No:
Date:

Total no. of references = 10

4	7	6	1	7	6	1	2	7	2
		6	6	6	6	6	4	7	7
	7	7	7	7	7	7	2	2	2
4	4	4	1	1	1	1	1	1	1

7 2 1 6 7 4

Queue

No. of page faults occurred = 6

Total no. of page hits = Total no. of references - No. of page faults

= 10 - 6

= 4

Hit ratio = Total no. of page hits / Total no. of references

= 4/10

= 0.4 or 40 %

Miss ratio = 1 - Hit ratio = 1 - 0.4 = 0.6 or 60 %

- Most recently used (MRU) Page Replacement Algorithm

In this algorithm, page will be replaced which has been most recently used. Belady's anomaly can occur in this algorithm.

Page refs: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3 No. of page frame = 4

7	0	1	2	0	3	0	4	2	3	0	3	2	3
			2	2	2	2	2	2	3	0	3	2	3
		1	1	1	1	1	1	1	1	1	1	1	1
	0	0	0	0	3	0	4	4	4	4	4	4	4
7	7	7	7	7	7	7	7	7	7	7	7	7	7
miss	miss	miss	miss	Hit	miss	miss	miss	Hit	miss	miss	miss	miss	miss

Total page fault = 12

Page No:
Date:

Initially all slots are empty, so when 7 0 12 are allocated to the empty slots → 4 page faults

0 is already there → 0 page fault

When 3 comes it will take place of 0 because it is most recently used → 1 page fault

When 0 comes it will take place of 3 → 1 page fault
When 4 comes it will take place of 0 → 1 page fault

2 is already in memory → 0 page fault

When 3 comes it will take place of 2 → 1 page fault
When 0 comes it will take place of 3 → 1 page fault
When 3 comes it will take place of 0 → 1 page fault
When 2 comes it will take place of 3 → 1 page fault
When 3 comes it will take place of 2 → 1 page fault

Thrashing

Thrashing in OS is a phenomenon that occurs in computer operating systems when the system spends an excessive amount of time swapping data b/w physical memory (RAM) and virtual memory (disk storage) due to high memory demand and low available resources.

Thrashing can occur when there are too many processes running on a system and do not have enough physical memory to accomodate them all

- Symptoms
  - High CPU utilization
  - Increased Disk Activity
  - High page fault rate
  - Slow response time

- Algorithms during thrashing
  - Global Page Replacement
  - Local Page Replacement

- Causes of thrashing
  - High degree of multiprogramming
  - Lack of frames
  - Page replacement policy
  - Insufficient physical memory
  - Inefficient memory management
  - Poorly designed applications

- Techniques to prevent thrashing
  - Increase the amount of physical memory
  - Reduce the degree of multiprogramming
  - Use an effective page replacement policy
  - Optimise applications
  - Monitor the system's resource usage
  - Use a system monitoring tool

Page No:
Date:

File Management

A computer file is defined as the medium used for saving and managing data in the computer system.

File System: File system is a method in operating system uses to store, organise and manages files and directories on a storage device

- **File access methods**

When a file is used, information is read and accessed into the computer memory and there are several ways to access this information of the file.

There are three ways to access file into the computer system

(i) Sequential Access
(ii) Direct Access
(iii) Index Sequential Access

- **Sequential Access**

It is the simplest access method. Information in the file is processed in order, one record after the other. This mode of access is by far the most common; for example, editor and compiler usually access the file in this system.

- When we use read command, it move ahead pointer by one.
- When we use write command, it will allocate memory and move pointer to end of file
- Such a method is reasonable for tape.

- **Direct Access**

It is also known as the relative access method. A fixed length logical read that allows the program to read and write the record rapidly. The direct access is based on the disk model of a file since disk allows random access to any file block. The file is viewed as a numbered sequence of block/record.

- Index Sequential Method

It is the other method of accessing a file that is built on top of the sequential access method. These method constructs the index for the file. The index, like the index in the back of a book, contains pointer to the various blocks. To find a record in the file, we first search the index and then by the help of pointer, we access the file directly

- Relative Record Access

It is a file access method used in operating system where records are accessed relative to the current position of the file pointer. In this method, records are located based on the position relative to the current record, rather than by a specific address or key value

- Content Addressable Access

It is a file access method used in operating system that allows records or blocks to be accessed based on their content rather than their address. In this method, a hash function is used to calculate the unique key for each record or block and the system can access any record or block by specifying its key

- File Directory

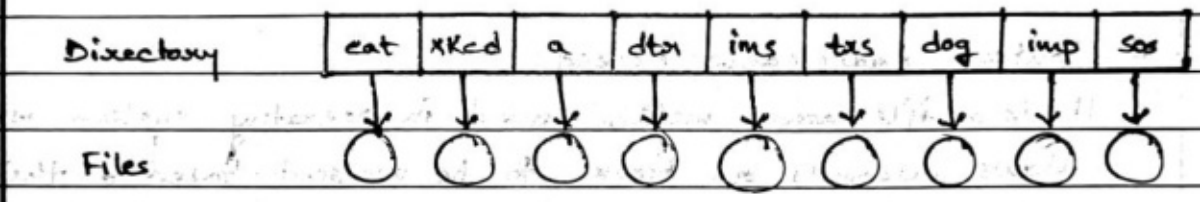
The collection of files is a file directory. The directory contains information about the files including attributes, location and ownership. The directory is itself a file accessible by various file management routines

- Following information contained in device directory
  Name, Type, Address, Current length,
  Maximum length, Date last accessed,
  Date last updated, Owner id,
  Protection information

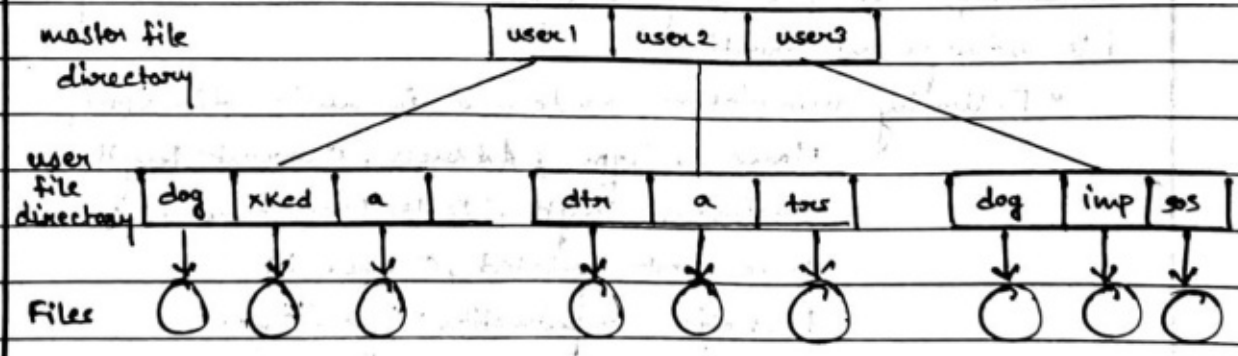
Page No:
Date:

- Operations performed in directory :
  - Search for a file
  - Create a file
  - Delete a file
  - Lists a directory
  - Rename a file
  - Traverse the file system

- **Single Level Directory**
  In this a single directory is maintained for all users
  - Naming problem - User cannot have same name for two files
  - Grouping problem - Users cannot group files according to their needs

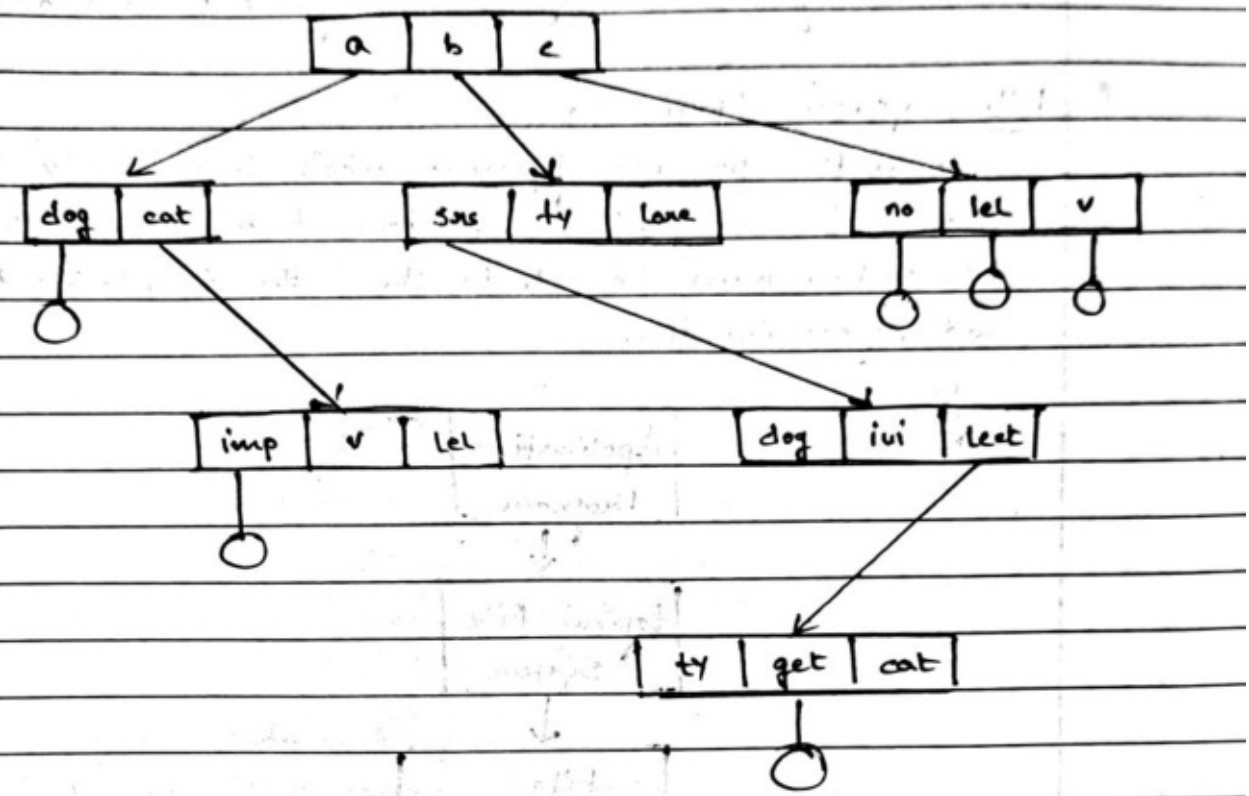


- **Two Level Directory**
  In this separate directories for each user is maintained
  - Path name - Due to two levels there is a path name for every file to locate that file
  - Now we can have same file name for different users
  - Searching is efficient in this model



- Tree structured directory

The directory is maintained in the form of a tree. Searching is efficient also there is grouping capability. We have absolute or relative path name for a file.



- File System Mounting.

Mounting is the process in which the operating system adds the directories and files from a storage devices to the user's computer file system. The file system is attached to an empty directory. By adding so the system user can access the data that is available inside the storage device through the system file manager.

Terminologies

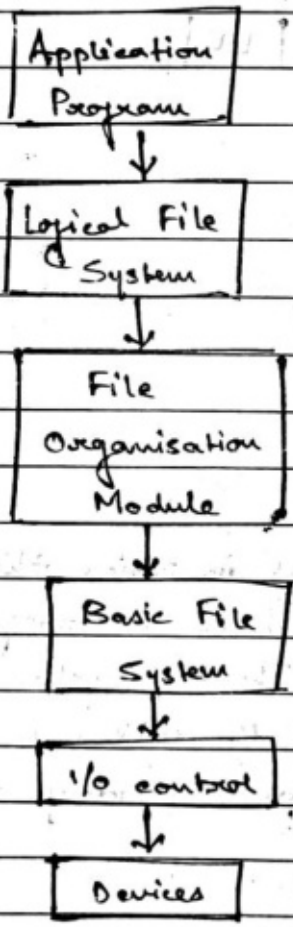
- File system - It is the method used by the OS to manage data storage in a storage device. So the user can access and organise the directories and files in an efficient manner

- Device name : It is a name / identifier given to a storage partition

- Mount point : It is an empty directory in which we are adding the file system during the process of mounting

File System Structure

File system provide efficient access to the disk by allowing data to be stored, located, retrieved in convinient way. A file system must be able to store the file, locate the file and reject the file.



When an application program asks for a file, the first request is directed to the logical file system. The logical file system contains the meta data of the file and directory structure. If the application program doesn't have the required permission of the file then this layer will throw an error. Logical file systems also verify the path to the file.

- Generally files are divided into various logical blocks. Files are stored in the hard disk and to be retrieved from hard disk. Hard disk is divided into various tracks and sectors. Therefore in order to store and retrieve the files, the logical blocks need to be mapped to physical blocks. This organisation is done by file organisation module. It is also responsible for free resource management.

- Once file organisation module decided which physical block the application program needs, it passes the information to basic file system. The basic file system is responsible for issuing the commands to I/O control in order to fetch those blocks.

- I/O controls contains the codes by using which it can access hard disk. These codes are known as device drivers. I/O controls are also responsible for handling interrupts.

File System Implementation

It is the process of designing, developing and implementing the software components that manages the organisation, allocation and access to files on a storage device in an OS

It plays a critical role in ensuring the reliability, performance and security of the OS's file ~~system~~ storage capabilities. Without an effective file system implementation, an OS can't effectively manage the storage of data on a storage device, resulting in data loss, corruption and inefficiency.

- File System Structure
  - DISK LAYER AND PARTITIONING

It refers to how a physical disk is divided into logical partitioning which can be accessed by the OS as separate entities. The disk is divided into one or more partitions which can be formatted with a file system. Disk partitioning involves creating partitions on the disk while disk formatting involves creating a file system on partition.

The partitioning process is typically done when the disk is first installed and the formatting process is typically done when a partition is created.

- FILE SYSTEM ORGANISATION

It refers to how files and directories are stored on the disk. A file system is responsible for managing files and directories and providing a way for users and applications to access and modify them.

- FILE ALLOCATION METHODS

It refers to how file data is stored on the disk. There are several different file allocation methods including contiguous blocks, linked allocation and indexed allocation

Contiguous allocation stores files in contiguous blocks on the disk ~~store~~ while linked allocation uses pointers to link blocks of data together. Indexed allocation uses an index to keep track of where each file block is stored in the disk.

- DIRECTORY STRUCTURE

It refers to how directories are organised and managed on the disk. Directories are used to organise files and other directories into a hierarchy which can be navigated by users and applications

- Implementation Issues

- Disk Space Management
- Consistency checking and error recovery
- File locking and concurrency control
- Performance optimization

- File System Operations
  - File creation and deletion
    File creation involves allocating space on the disk for a new file and setting up its attributes and permissions. File deletion involves removing the file from the disk and releasing the space it occupies.

  - File open and close
    File open involves establishing a connection b/w the file and a process/application that wishes to access it. File close involves terminating that connection and freeing up any resources used by the process/application.

  - File read and write
    File read involves retrieving data from a file and transferring it to a process/applications. File write involves sending data from a process/application to a file. These operations can be performed at various levels of granularity such as bytes, blocks or sectors

  - File seek and position
    File seek involves moving the current position of the file pointer to the specific byte or block within the file. File position refers to the current location of the file pointer within the file. These operations are used for random access and manipulation of specific portions of a file.

  - File attributes and permissions
    File attributes refer to metadata associated with a file such as its name, size and creation/modification dates. File permissions refer to the access control settings that determine who can read, write, execute or modify a file. These settings can be set for individual users or groups and can be used to restrict access to sensitive data/programs

Page No:
Date:

FILE ALLOCATION METHODS

(i) Continuous Allocation

A single continuous set of blocks is allocated to a file at the time of file creation. Thus, this is a pre allocation strategy using variable size portions. The file allocation table needs just a single entry for each file, showing the starting block and length of the file. This method is best from the POV of individual sequential file. Multiple blocks can be read in at a time to improve I/O performance for sequential processing.

(ii) Linked Allocation

Files are stored in scattered blocks linked together by pointers within the blocks. This allows for dynamic file growth but can waste space due to internal fragmentation.

(iii) Indexed Allocation

Indexed allocation fixes fragmentation issues of other methods. It uses an index to locate scattered file blocks, allowing both fast access and file growth. This makes it a popular choice for file systems

FREE SPACE MANAGEMENT

A file system is responsible to allocate the free blocks to the file therefore it has to keep a track of all the free blocks present in the disk. There are mainly two main approaches -

① Bit Vector - In this approach, the free space list is implemented as bit map vector. It contains the number of bits where each bit represent each blocks.

If the block is empty then the bit is 1 otherwise it is 0. Initially all the blocks are empty therefore each bit in the bit map vector contains 1

As the space allocation proceeds, the file system starts allocating blocks to the files and setting the respective bit to 0.

② Linked list - It is another approach for free space management. This approach suggests linking together all the free blocks and keeping a pointer in the cache which points to the first free block.

Therefore, all the free blocks on the blocks on the disks will be linked together with a pointer. Whenever a block gets allocated its previous free block will be linked to its next free block.