Operating System

Date / /

An O.S is software layer between hardware & user program. It manages resources, does file system management, process management, security & protection.

Types

- Batch OS: Jobs are grouped into batches & OS executes those batches.
- Multiprogramming OS: Keeps several jobs in main memory & allows more efficient utilization by context switching when a program is waiting for I/O.
- MultiTasking / Timesharing OS: CPU executes multiple jobs by frequently switching among them.
- Multiprocessing OS: 2 or more CPU's within a single computer. True parallelism.
- Real time OS: Special purpose OS which has well defined time constraints.
- Distributed OS: Manages a group of distinct computers & makes them appear to the user as single cohesive system.

Structure of OS

- Simple Structure (Monolith Kernel): All OS components are integrated into single large executable. Ex- Linux, Mac OS

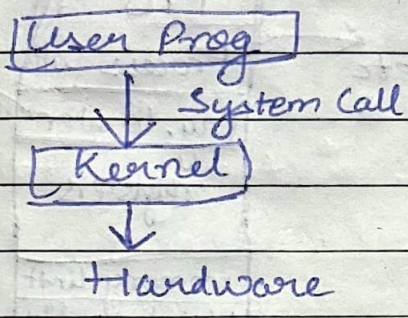
Date

IPC
CPU/scheduling
Memory manag.

- Microkernel : Only essential functions are included in the Kernel. Ex - Minix 3
- Layered Structure : OS is divided into layers. Ex - IBM's OS/2

OS = Kernel + User Interface → Shell, GUI's
(core components)

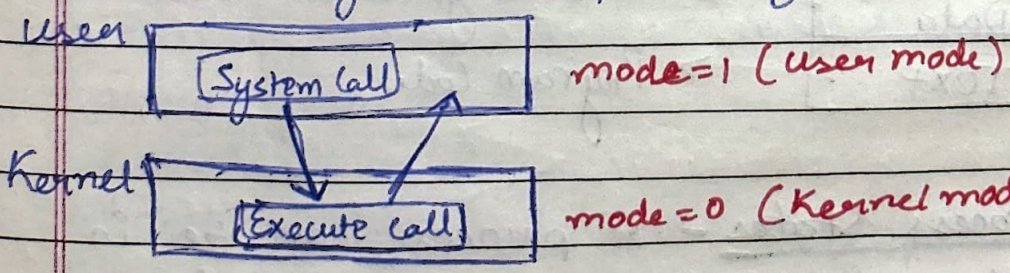
System Call - Provides the means for a user program to ask the OS to perform tasks reserved for OS on user program's behalf.



Types of System calls →
- Process Control - end, abort, load, execute, allocate memory
- File Management - Create, delete, open, read, write
- Device Management - Request device, logically attach or detach

etc..

Mode - Refers to privilege level of a process.

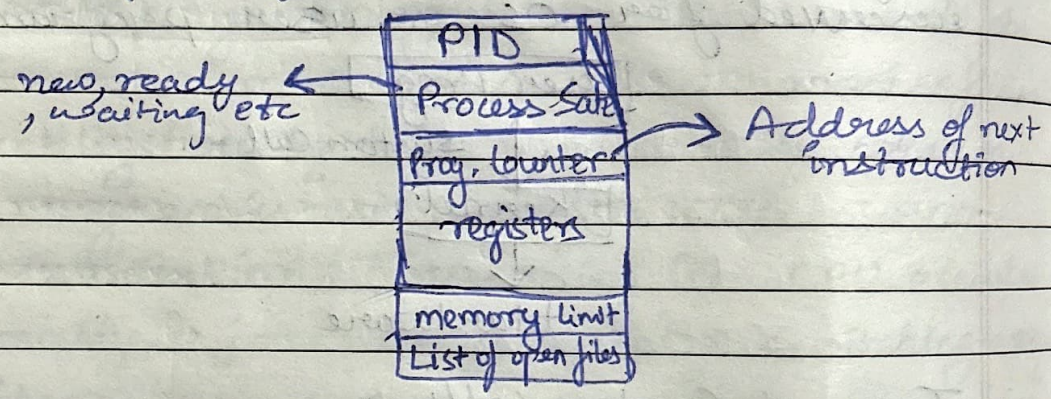


Date / /

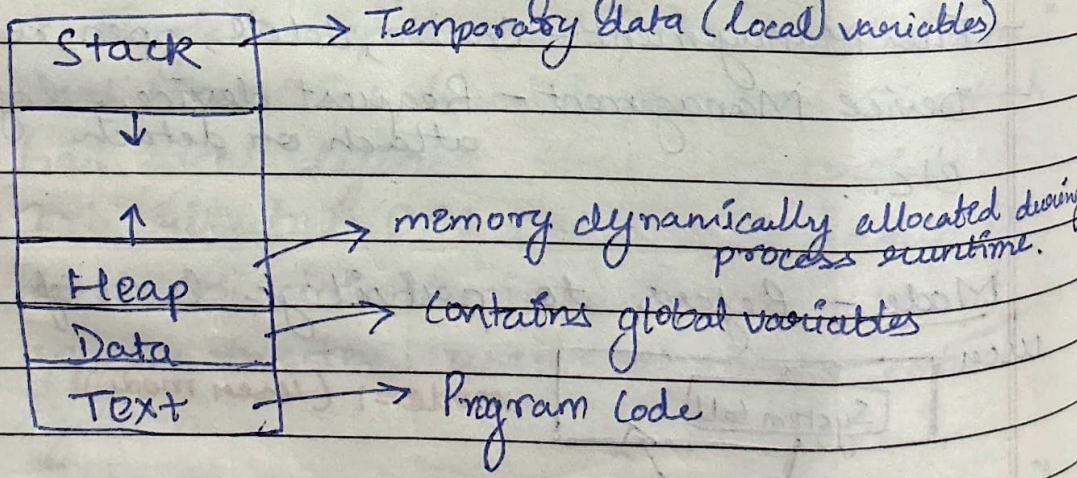
Process Management

Program in execution is process.
When a executable file is loaded into main memory & when its PCB is created.

PCB - Data structure maintained by OS to keep track of state of process.

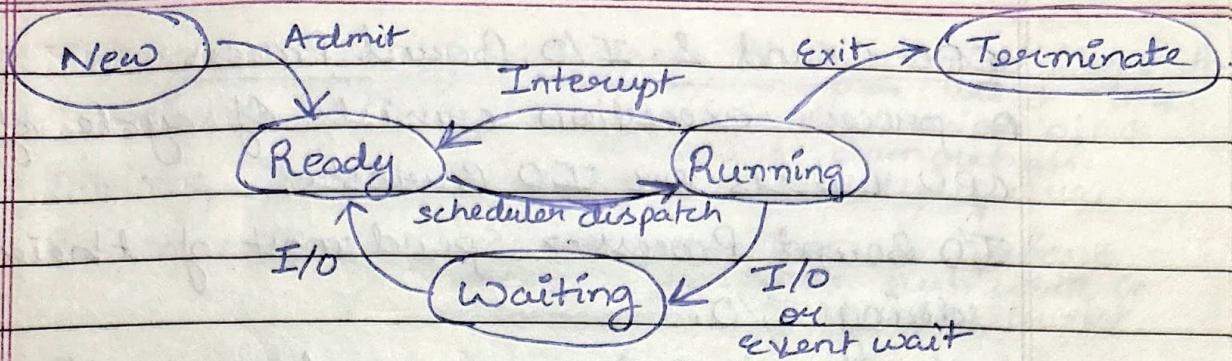


A process consists of following section →



Process States - A process can be in different states during its lifetime

Date / /



Schedulers - Component responsible for determining which process should be executed & next on CPU.

Types

→ Short Term: Picks next process to run from ready queue. It is invoked frequently to ensure efficient CPU utilization.

→ Medium Term: Swaps process b/w main memory & secondary storage. Used to manage memory efficiently & increase degree of multiprogramming.

→ Long Term: Decides which process from job queue should be admitted to ready queue.

Dispatchers - Module that gives control of the CPU to the process selected by short term scheduler.

Key functions:

Context Switch & Hardware support (like interrupts & mode switching)

Date / /

CPU Bound & I/O Bound Process

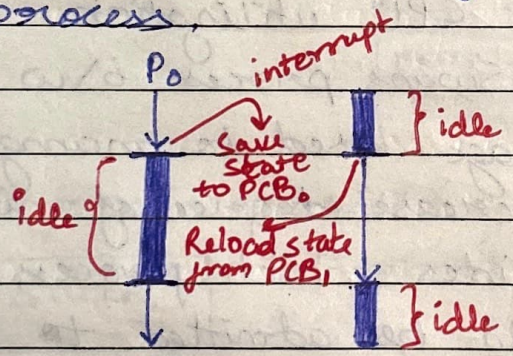
A process execution consists of cycle of CPU bursts or I/O Bursts.

I/O Bound Process → Spend most of their time doing I/O.

CPU Bound Process → Spend more time on CPU.

Context Switch

Switching from 1 process to another requires to save the state of currently running process & restoring state of another process.



CPU Scheduling Algorithms

Process of determining which process in ready queue is allocated to the CPU

Types of scheduling → Non-preemptive
→ Preemptive : Forcefully moved from running state.

Scheduling Criteria →

- CPU utilization : Percent of time CPU is busy

Date / /

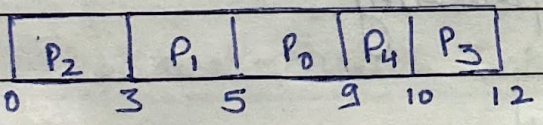
- Throughput: No. of processes that complete execution per unit time.
- Turnaround Time: Time b\w submission of a process & completion.
- Waiting Time: Time spent in ready queue.
- Response Time: Time from when a process becomes ready to run until it recieves its 1st CPU burst.
- Overhead: Computational cost of scheduling algorithm itself.

FCFS (First Come First Serve)

non-preemptive in nature

P No.	AT	BT	CT	TAT	WT
P_0	2	4	9	9-2=7	7-4=3
P_1	1	2	5	4	2
P_2	0	3	3	3	0
P_3	4	2	12	8	6
P_4	3	1	10	7	6

Gantt chart



Note → Smaller process have to wait for CPU because of larger process — Convoy Effect.

SJF (Shortest Job First) & SRTF (Shortest Remaining Time First)

SJF → non-pre-emptive

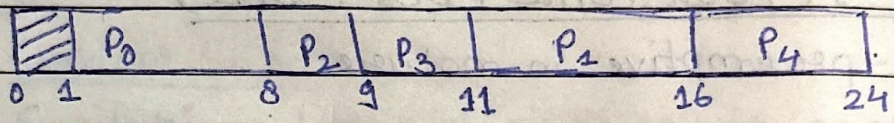
SRTF → preemptive

Provide better avg response time compared to FCFS but the processes with longer CPU burst time go into starvation.

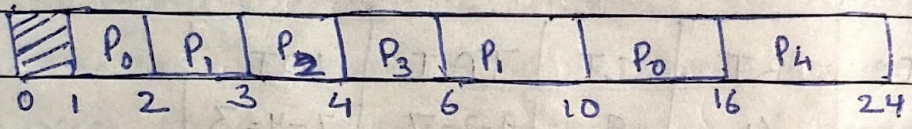
Date / /

P_{No}	AT	BT
P_0	1	7
P_1	2	5
P_2	3	1
P_3	4	2
P_4	5	8

SJF



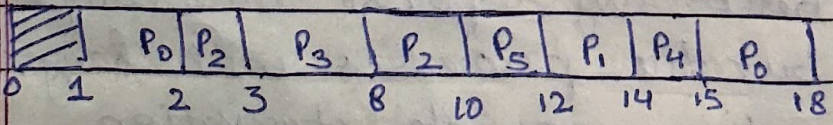
SRTF



Priority Scheduling

A priority is associated with each process, at any instance CPU is allocated to process with highest priority.

P_{No}	AT	BT	Priority
P_0	1	4	4
P_1	2	2	5
P_2	2	3	7
P_3	3	5	8(H)
P_4	3	1	5
P_5	4	2	6



Note → If W.T of a process increases, you increase its priority to avoid starvation — Ageing

Date / /

Round Robin

Designed for time-sharing systems. Each process can hold CPU for particular time quantum (q).

q = 2

P No	AT	BT
P_0	0	4
P_1	1	5
P_2	2	2
P_3	3	1
P_4	4	6
P_5	6	3

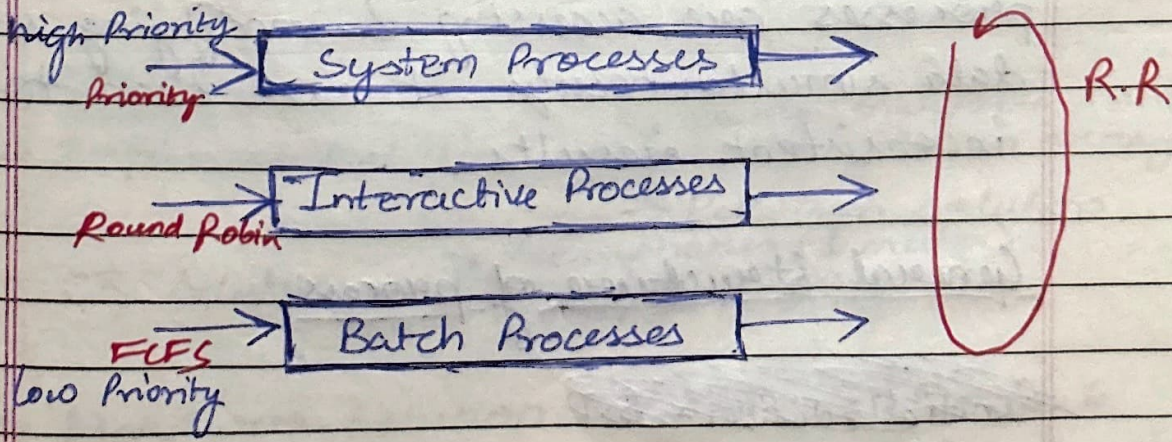
RQ → $P_0 P_1 P_2 P_0 P_3 P_4 P_1 P_5 P_4 P_1 P_5 P_4$

P_0	P_1	P_2	P_0	P_3	P_4	P_1	P_5	P_4	P_1	P_5	P_4

0 2 4 6 8 9 11 13 15 17 18 19 21

Multi Level-Queue Scheduling

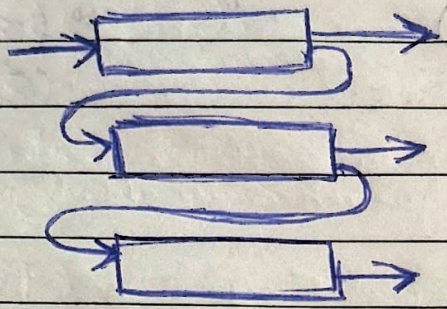
CPU Scheduling algorithm that divides processes into multiple queues based on the characteristics & requirements. Each Queue uses diff sch algo.



Date / /

Multi-level Feedback Queue Scheduling

Processes can move b/w queues based on their behaviour.



Process Synchronization

In a multiprogramming environment a good number of processes compete for limited resources. Concurrent access to shared data at some point may result in data inconsistency.

```
P() {
  read(i);
  i = i + 1;
  write(i);
}
```

P_1 : $i = 10$, $i = 10 + 1 = 11$ → context switched before write

P_2 : $i = 10$, $i = 10 + 1 = 11$, write 11

P_1 : write 11

Now value of $i = 11$ instead of being 12.

Race condition - When multiple threads or processes are accessing & modifying shared data simultaneously it can lead to inconsistent results.

General Structure of process

~~Initial Section - EP~~

Date / /

```
P()
{
  while(T)
  { Initial Section
    Entry Section
    CRITICAL SECTION
    Exit Section
    Remainder Section
  }
}
```

→ segment of code that access a shared resource.

Criteria to solve critical section problem

→ Mutual Exclusion - At most, 1 process can be executing in critical section at any time.

→ Progress - If no process is executing in its critical section & there are processes that wish to enter critical section, then one of these processes must be permitted to enter C.S within a finite amount of time.

→ Bounded Wating - There exists a bound on no. of times that other processes are allowed to enter C.S, this prevents a process from being indefinitely delayed.

Note → It is mandatory to satisfy mutual exclusion & progress.

Solutions

- 2-process Solution ⟷ using variable turn, " boolean array flag
  → Peterson solution
- OS Solution → Semaphores (& mutex)
  → Message Passing
- Hardware Solution → Test & set Lock
  → Disable interrupt

Date / /

Using turn variable →

P_0

```
while(1){
  while(turn!=0);
  <C.S>
  turn=1;
  <Remainder Section>
}
```

P_1

```
while(1){
  while(turn!=1);
  <C.S>
  turn=0;
  <Remainder Section>
}
```

Mutual Exclusion is satisfied, but Progress is not because it is strict alternation, didn't ask process if it wants to go or not.

Peterson's Solution →

P_0

```
while(1){
  flag[0]=T;
  turn=1;
  while(turn==1 && flag[1]==T);
  <C.S>
  flag[0]=F;
  <Remainder Section>
}
```

P_1

```
while(1){
  flag[1]=T;
  turn=0;
  while(turn==0 &&
        flag[0]==T);
  <C.S>
  flag[1]=F;
  <Remainder Section>
}
```

P_0 starts
turn = ~~0~~ 1
flag = [~~F~~ F]
P_0 enters C.S

P_1 wants to enter too
turn = ~~1~~ 0
flag [T T]
P_1 busy waits until P_0 comes out of C.S

This solution ensure Mutual Exclusion, Progress & Bounded wait.

Date / /

Semaphores

Synchronization mechanism used in operating system to control access to shared resources. They are essentially integer variables that can be accessed only through atomic operations: wait (P) and signal (V) Important Start with S=1;

```
Wait(S){
  while(s<=0);
  s--;
}
```

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

```
Pi(){
While(T){
Initial Section.
wait(S)
<C.S>
signal(S)
<Remainder Sec>
}
}
```

Mutual Exclusion ✓
Progress ✓
Bounded wait ✗

Classical Problems on Syncronization

→ Producer - Consumer Problem

2 processes Producer produces info & puts into buffer which is consumed by consumer. Both can produce & consume only 1 item at a time.

Solution using 3 semaphores :
S=1 // critical section
E=n // n empty cells
F=0 // 0 filled cells

Overflow cond" of Buffer ⇒ wait(E) — as E=0 it will keep waiting, makes sense because empty cells are 0!

Underflow cond" of Buffer ⇒ wait(F)

Semaphore S takes care of buffer

Date / /

```
Producer(){
while(T){
  // Produce a item
  wait(E) // overflow
  wait(S)
    // Add item to Buffer
  signal(S)
  signal(F)
}
}
```

```
Consumer(){
while(T){
  wait(F)
  wait(S)
    // Pick item from buffer
  signal(S)
  signal(E)
  // consume a item
}
}
```

→ Readers - Writers Problem

Multiple readers can simultaneously read a shared data item, but only 1 writer can modify at a time. Problem is to ensure readers dont interfere with writers & writers don't interfere with each other.

Mutex
Writ
Readcount
tracks no. of readers
controls access to readcount

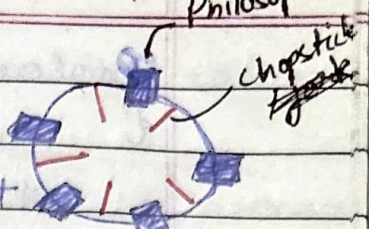
```
Writer(){
wait(writ)
  <C.S> // write
signal(writ)
}
```

```
Reader(){
wait(mutex)
readcount++;
if(readcount==1)
  wait(writ)
signal(mutex)
  <C.S> // read
wait(mutex)
readcount--;
if(readcount==0)
  signal(writ)
signal(mutex)
}
```

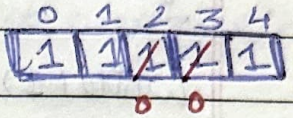
1'st reader acquires writ to prevent any writer from coming but readers are allowed.

Date / /

→ Dining Philosopher Problem



5 philosopher, each with a chopstick to their left & right
Each philosopher must acquire both chopstick.




```
void Philosopher(void){
  while(T){
    Thinking();
    wait(chopstick[i]); // acquire right
    wait(chopstick[(i+1)%5]); // acquire left
    Eat();
    signal(chopstick[i]);
    signal(chopstick[(i+1)%5]);
  }
}
```

Note → But this solution suffers from deadlock, imagine context switch ~~right~~ after acquiring right chopstick.

To prevent deadlock →

- Allow a philosopher to pick chopstick only if both are available. (becomes part of <CS>)
- Odd philosopher picks left chopstick first while even " " right " ".

Hardware Based Solution for C.S

Test & Set

```
if(lock==0)
{
  lock=1
  <C.S>
  lock=0
}
```

P_1 is here, but context switched & P_2 enters, lock's value is still 0.

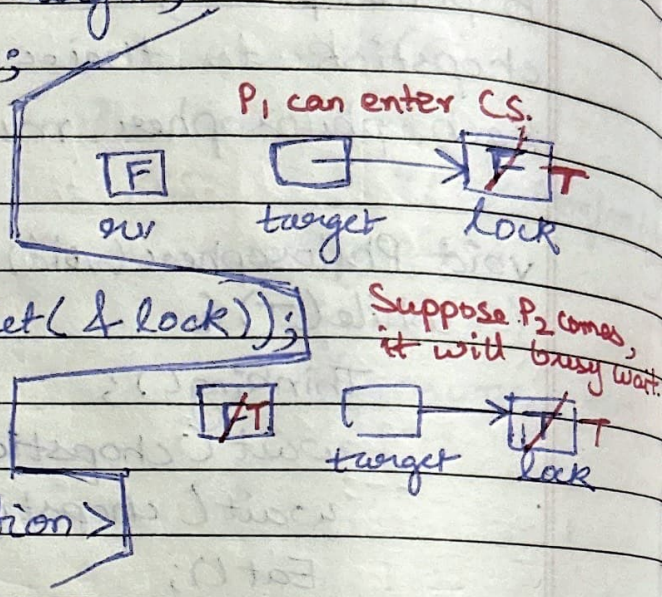
P_2 enters C.S, after some time P_1 will continue change lock's value from 1 to 1 & enter C.S

P_1 enters too

Date / /

```
Boolean testAndSet(Boolean *target)
{   Boolean rv = *target;
    *target = true;
    return rv;
}

while(1){
    while(testAndSet(&lock));
    <C.S>
    lock = false;
    <Remainder Section>
}
```



Note → CISC computers execute it ~~automati~~ atomically

Deadlock

Deadlock in O.S occurs when 2 or more processes are waiting for each other to release resources, resulting in situation where none of them can proceed.

Necessary conditions for deadlock:

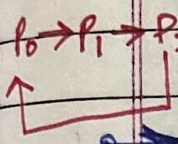
- → Mutual Exclusion: At least 1 resource must be non-shareable, only 1 process can use it at a time.
- → Hold and Wait: A process must be holding at least 1 resource while waiting for another.
- → No preemption: Resources that are being

Date / /

held by a process cannot be forcibly taken away.

→ Circular Wait : A circular chain of process must exist, where each process is waiting for a resource held by next process in chain.

Circular wait is most imp. condⁿ



Deadlock Handling Methods

Prevention, Avoidance, Detection, Ignorance

→ Prevention : Ensure that atleast 1 condⁿ out of 4 is violated to prevent deadlock.

Mutual Exclusion → Can't be violated

Hold & Wait → Process allowed to run iff all resource are acquired.

No Preemption → Pre-empt the desired resource based on priority.

Circular Wait → Resource ordering, assign a unique numerical value to each resource. Process must req. resource in increasing order of numerical values, thus if it's holding higher val. resource it will have to release it.

→ Avoidance : Allows system to dynamically examine resource-allocation state to ensure that a circular wait condition never occurs.

BANKER'S ALGORITHM —

P_1, P_2, P_3 → Processes

A, B, C → Resources

Max Availability

A	B	C
10	5	7

Initial State

Amount of each resource req. by Process

Already allocated amount

Max - Allocated

Process	Max(A,B,C)	Allocation(A,B,C)	Need (A,B,C)
P_1	(7,5,3)	(0,1,0)	(7,4,3)
P_2	(3,2,2)	(2,0,0)	(1,2,2)
P_3	(9,0,2)	(3,0,2)	(6,0,0)

Max Avail

A	B	C
10	5	7

Available

A	B	C
5	4	5

At this state, I can satisfy P_2's demands $(1,2,2) < (5,4,5) \rightarrow (4,2,3)$

After running this will release the resource so new Available $\Rightarrow (4,2,3) + (3,2,2)$

A	B	C
~~7~~	~~6~~	5

Now Even P_1 can can be satisfied as well as P_3

$P_2 \rightarrow P_3 \rightarrow P_1$ → Safe Sequence

Note → If there is no safe sequence there is probability of deadlock.

→ Detection & Recovery : Allows a system to operate normally & periodically check for deadlock (cycle detection in wait-for graph)

Once a deadlock is detected —

- ⊙ Process Termination
- ⊙ Resource Pre-emption

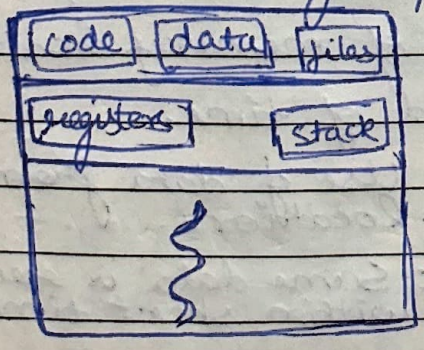
Date / /

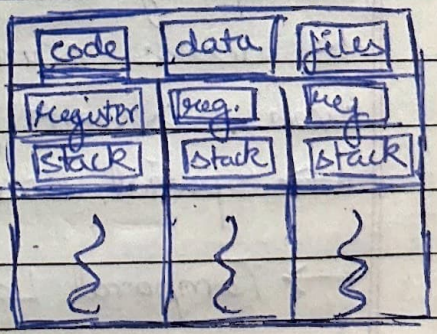
→ Ignorance : Assume that deadlock's are rare so leaving systems on their own. Also called Ostrich algo.

Fork & Threads

Fork - The fork() system call is used to create a new process by duplicating existing process. This new process is called child process.
The child process gets a unique PID & its own memory space, but initially it shares same code and data as parent.
Used where processes need to run independently.

Thread - Smallest unit of execution within a process. They share same memory space & resources of process they belong to, and can run concurrently.
Used to perform multiple tasks simultaneously within a single process.

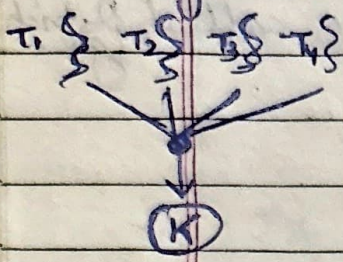




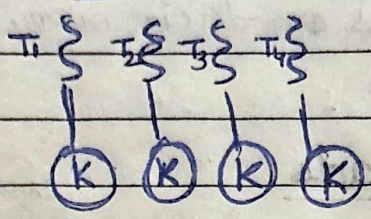
Kernel did'nt even come to know about thread while forking req. Kernel involvement.

Date / /

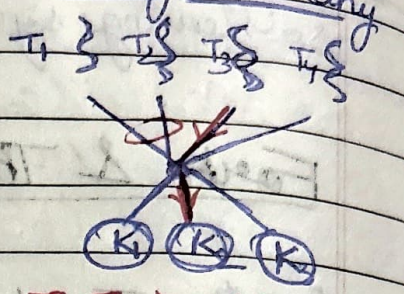
Many to One



One to One



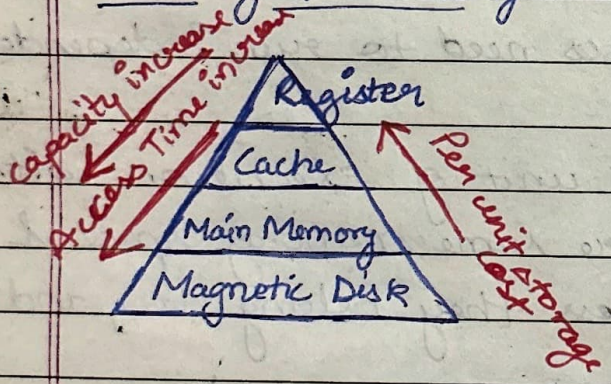
Many to Many

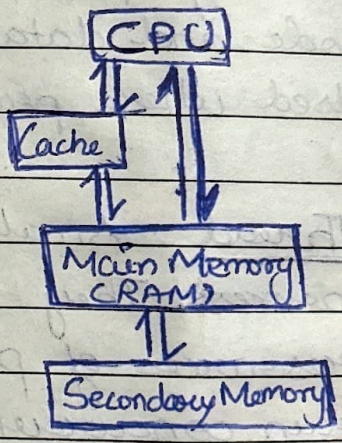


T_1, T_2 handeled by K_1
T_3 by K_2
T_4 by K_3

Memory Management

Memory Hierarchy





Locality of Reference

Principle of locality states that computer tend to access same set of memory locations over a short period of time.

→ Spatial Locality : Use of data from nearby locations.

→ Temporal Locality : Same data is reused within short time.

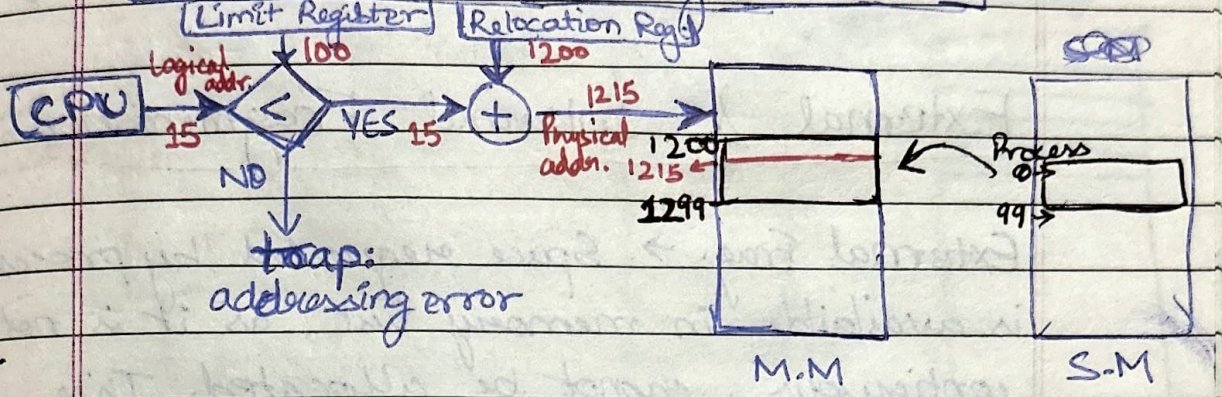
Memory Allocation Policies

→ **Contiguous Allocation :** When a process is required to be executed it must be

Date / /

loaded in main memory. In contiguous, it must be loaded to main memory completely & should be stored in contiguous fashion

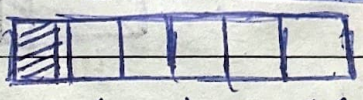
Address Translation in Contiguous Allocation



Space Allocation Method in Contiguous Allocation

1. Variable Size Partitioning

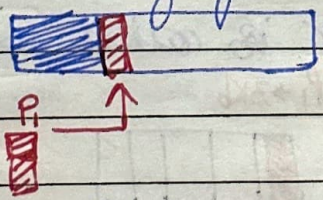
2. Fixed Size Partitioning



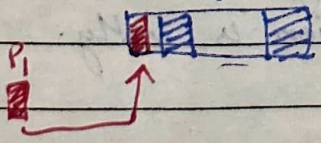
Div memory into fixed size partitions, process takes whole partition, rest of the space is wasted.

Policies

→ First Fit Policy: Search memory from base & allocate first partition which is capable enough for the process.

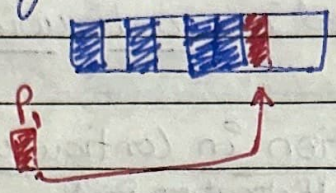


→ Best Fit Policy: Search entire memory & allocate smallest partition which is capable.
Note → "Best" for Fixed size



Date / /

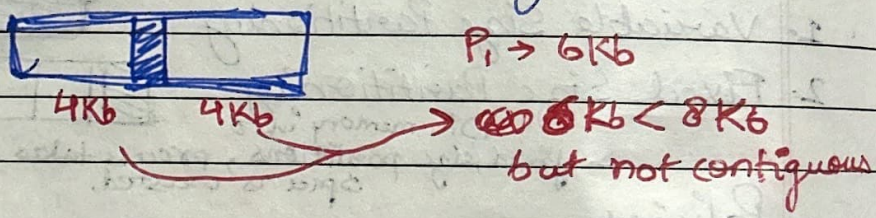
→ Worst Fit Policy: Searches entire memory & allocates largest partition possible.



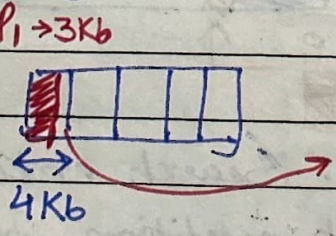
Note → "Worst" for Fixed size but best in variable size partitioning.

External & Internal Fragmentation

External Frag. → Space requested by process is availaible in memory but, as it is not contiguous, cannot be allocated. This wastage is External Fragmentation.



Internal Frag. → Happens in Fixed size partition, when process is allocated space but as it is fixed, there can be some leftover space, that space wasted is Intern Frag.



Note → To solve external fragmentation, we can regularly defragment but this solution is costly.

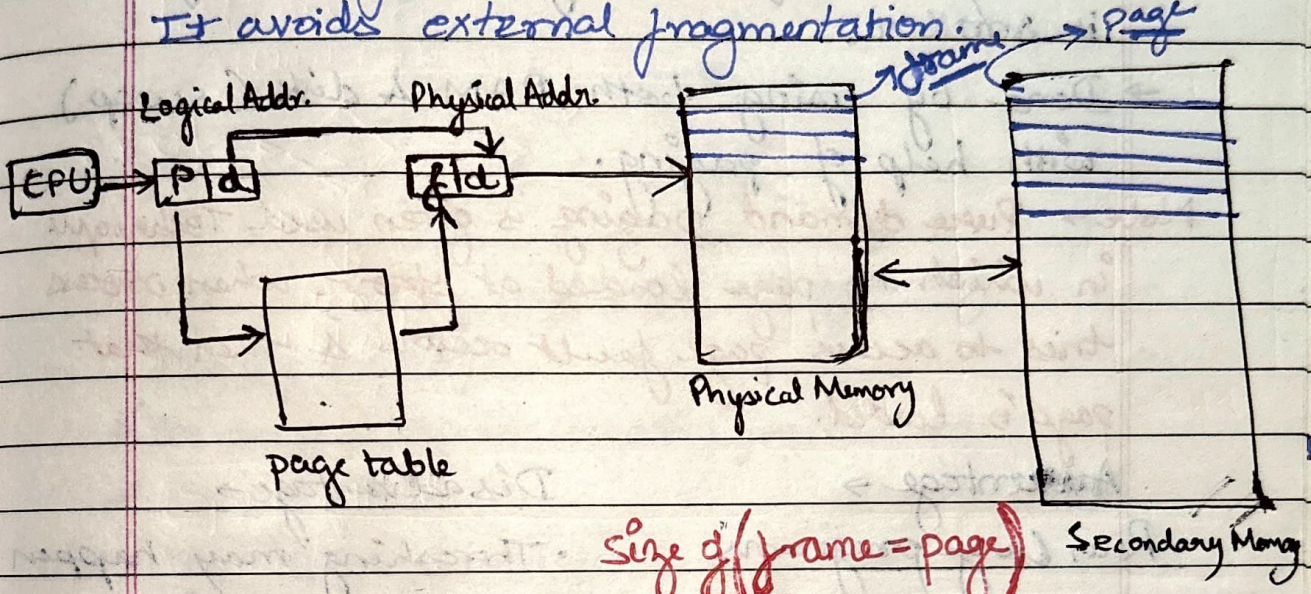
Date / /

(Paging)

→ Non Contiguous Memory Allocation :

Memory management scheme that permits the physical address space of a process to be non-contiguous.

It avoids external fragmentation.



Size of (frame = page)

→ Logical Address consist of p (page no.) & d (instruction offset)

→ p is used to look frame no (f) in Page table

→ Combining frame no. (f) with instruction offset we get physical address.

→ Every process has its own page table & will contain all entries that are in secondary memory

→ Page table is also stored in main memory.

(Segmentation)

Note > Better approach for non-contiguous memory allocation is paging.

Another way of dividing a process, istead of fixed size pages, we divide based on logical parts of program (code, stack, heap, data)

The problem - because of this variable sizes again external fragmentation may occur.

Date / /

Virtual Memory

→ Memory management technique that gives the process a illusion of having a large contiguous block of memory even if actual memory (RAM) is smaller.

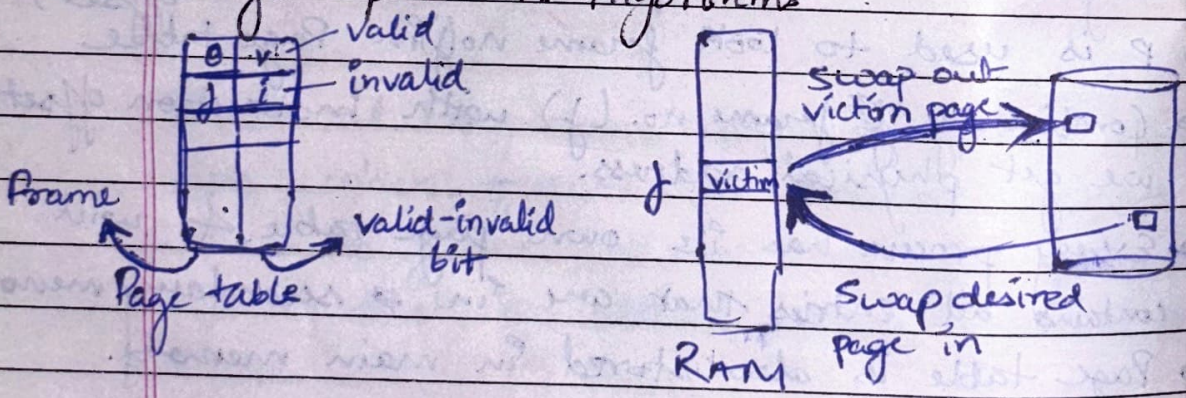
→ Done by using both RAM & disk (swap) with help of paging.

Note → Pure demand paging is often used. Technique in which no page loaded at start, when process tries to access page fault occurs & then that page is loaded.

Advantage →
- Run large programs

Disadvantage →
- Thrashing may happen

Page Replacement Algorithms



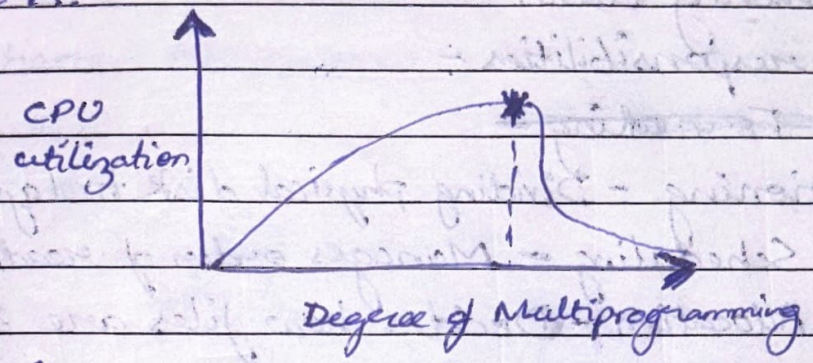
① FIFO - Remove oldest page
② LRU - Remove least recently used page
③ LFU - Remove page that was used least no. of times

Note → Belady's Anamoly : Page fault increase with no. of frames. May happen in FIFO

Date / /

Thrashing

When a system spends more time swapping pages in and out of memory than doing actual work.



Why it happens?

→ Too many processes competing for memory

Working Set Strategy

→ Working set of a process is pages it is actively using during specific time window.

→ Working set strategy states only run a process if their working set fits in memory.

Ex→ Process A's working set = 5 pages
" B's " " = 7 "
" C's " " = 6 "

RAM can hold = 12 pages

So A+B = 5+7 = 12, suspend C

Date / /

Disk Management

Has OS manages storage devices - writing, storing & retrieving data.

Key responsibilities :-

- ~~Disk Formating~~
- Partitioning - Dividing physical disk to logical sections.
- Disk Scheduling - Manages order of read/write requests
- File Allocation - Decides how files are stored
- Space Management - Tracking which disk blocks are free.