Operating System

⇒ OS is an Interface between user & H/W.

* User view

1] Easy to use → GUI
2] Min interruption while using the System.

* System view

1] use of command line to perform operations like copy etc in unix like systems.

* Evolution of OS :

1] Batch O.S → Scheduling a set of similar task on the system for execution.

2] Multiprogramming OS → Process Leave CPU as per its choice → only 1 resource.

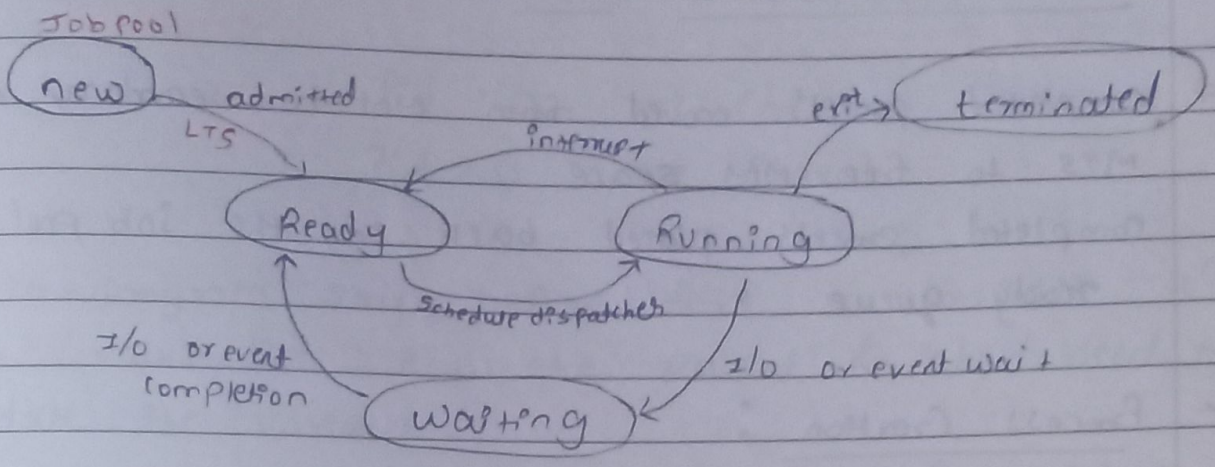
3] Multi tasking OS → Logical extension of multiprogramming (only 1 CPU resource)
→ process execute for a quantime time

4] Multi processing OS → Multi processor.

5] Real time OS

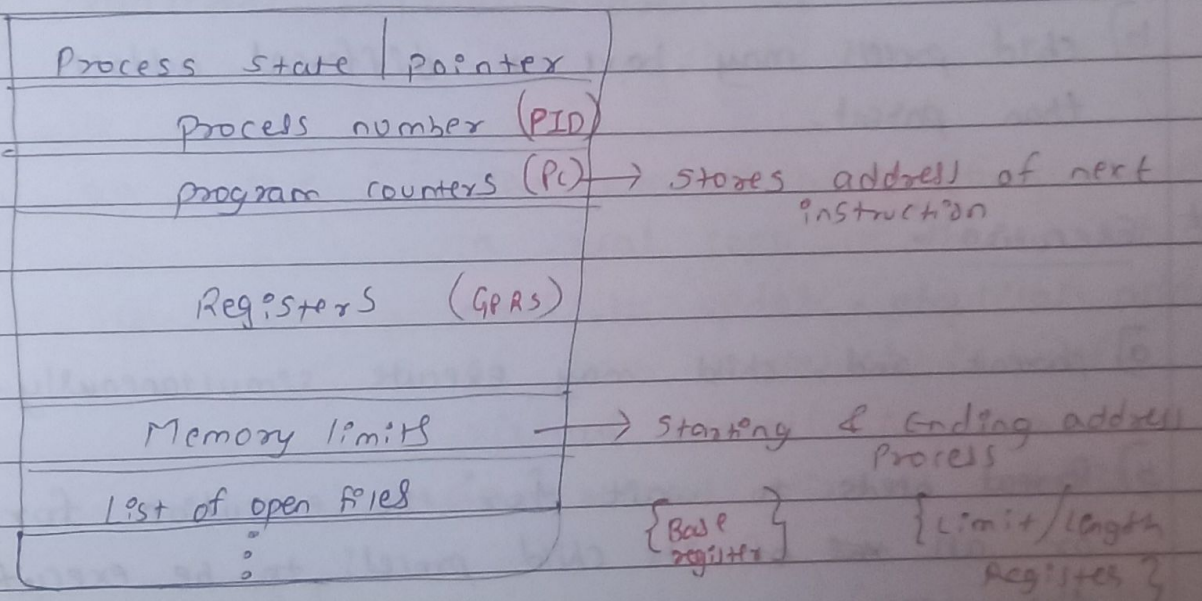
6] Distributed OS

* Process Model :



LTS → [Job pool → Ready Queue]
[Degree of multiprogramming]

* Process Control Block :



- process state
- Program counter
- CPU registers
- CPU Scheduling information
- Memory management information
- Accounting info
- I/O status info

* NOTE :

↳ Medium Term Scheduler (MTS)

⇒ Completed process moved from MM to secondary memory by MTS to free MM space
completed process placed back in the job pool from ready queue

* Process Creation :

* Address space :

a) child process may share the same address space as parent process, happens when child is replica of parent

b) child process may have a different address space than parent.

Execution :

a) parent and child may execute simultaneously.

b) Parent needs to wait for the execution for some or all ~~of~~ of its child process to be executed & then it can resume.

* Fork :

if Pid < 0 → error

Pid = 0 → child

Pid > 0 → Parent

In a program say before fork a variable 'a' has value 20, now the parent & child need to update the variable.
parent needs to update it to a+5 & child to a-5
So parent will read a = 20, update it & create a local copy of it. This is copy on write.
similarly child will read a = 20, update it & create a local copy of it
This variable 'a' has the same virtual address but different physical address & different value.

imp

* Copy on write : The process after fork creates a local copy of the variable it needs to update. physical address of both variables is diff. but virtual address is same.
&a → returns virtual address

a)
```
If (fork() == 0){
    a = a + 5 ;
    Printf("%d %d \n", a, &a);
}
else {
    a = a - 5 ;
    Printf("%d %d \n", a, &a);
}
```

child { a = a + 5 ; Printf("%d %d \n", a, &a); }

parent { a = a - 5 ; Printf("%d %d \n", a, &a); }

Parent : u = a-5
v = &a

child : x = a+5
y = &a

u = x+10 v = y

* Execlp System call :

→ The fork system call created new process & it is called by parent process.

→ Child process calls execlp system call
→ Whenever a process calls the execlp system call that process completely replaced by the new program & the new program starts executing in the main function.

→ execlp loads new program, which could be the parent, now parent can resume its execution

* Exit System call :

→ we can terminate a process.

* Wait System call :

→ A process is waiting due to some other processes's execution.

* Signal System call :

→ When a process resumes its execution.

* Bounded Buffer problem :

```
repeat
    produce an item

    while in+1 mod n = out do no-op ;
    buffer[i] = nextp ;
    in = in+1 mod n ;
until false ;
```

The consumer process has a local variable nextc, in which the item to be consumed is stored

```
repeat
    while in = out do no-op ;
    nextc = buffer[out] ;
    out = out+1 mod n ;
until false ;
```

* Components of process Synchronization
  - → Mutual Exclusion
  - → Progress
  - → Bounded waiting.

* Turn variable :

```
repeat
    while turn ≠ i do no-op ;
        CS
    turn = j ;
until false ;
```

ME ✓
Progress ✗
BW ✓

* Flag variable :

```
repeat
    flag[i] = true ;
    while flag[j] do no-op ;
          (CS)
    flag[i] = false ;
until false
```

ME ✓
Progress ✗
BW ?

* Turn + flag variable : (Peterson algorithm)

Flag has highest priority

```
repeat
    flag[i] = true ;
    turn = j
    while (flag[j] and turn = j) do no-op ;
          (CS)
    flag[i] = false ;
until false ;
```

ME ✓
Progress ✓
BW ✓

Flag[i]	Flag[j]	Turn	CS
F	F	i	-
F	T	i	j
T	F	i	i
T	T	i	i
F	F	j	-
F	T	j	j
T	F	j	i
T	T	j	j

* Bakery algorithm:

```
repeat
    choosing[i] = true;
    number[i] = (max[number(0], number(1), ... number(n-1]) + 1;
    choosing[i] = false;

    for j = 0 to n-1
        do begin
            while choosing[j] do no-op;
            while number[j] ≠ 0 and
                (number[j], j) < (number[i], i) do no-op;
        end;

    CS

    number[i] = 0;
until false;
```

→ ME ✓
→ Progress ✓
→ BW ✓

* Test & set:

```
function Test and set (var target : boolean):
        begin
            test and set = target → LOCK
            target = true;
            return test and set;
        end

repeat
    while Test and set (lock) do no-op;
        CS
    lock = false
```

* Swap :

```
repeat
    key = true;
    repeat
        swap(lock, key);
    until key = false

    [CS]

    lock = false;
until false;
```

lock → global var
key → local var

ME ✓
Progress ✓
BW ✗

* Test and set with bounded wait :

```
var : j = (0 ... n-1);
      key : boolean;

repeat
    waiting[i] = true;
    key = true;
    while waiting[i] and key do key = Testandset(lock);
    waiting[i] = false;
        [CS]
    j = j+1 mod n;
    while (j ≠ i) and (not waiting[j]) do j = j+1 mod n;
    if j = i then lock = false
             else waiting[j] = false;
until false
```

Ideal assumption :	ME ✓
lock = false	Bounded wait ✓
key = false	

* Semaphores :

Atomic operation : piece of code which cannot be interrupted.

Binary semaphore :

⇒ value can be 0 or 1.

* Producer Consumer problem :

Producer : mutex = 1 ; empty = n ; full = 0

```
repeat
    wait (empty)
    wait (mutex)
    [CS]
    signal (mutex)
    signal (full)
until false
```

Consumer :

```
repeat
    wait (full)
    wait (mutex)
    [CS]
    signal (mutex)
    signal (empty)
until false
```

* Reader - writer process

```
Var mutex , wrt : semaphore ;
    readcount : integer ;

Reader :  wait (mutex)
          readcount = readcount + 1 ;
          if (readcount == 1) then wait (wrt) ;
          signal (mutex) ;

                  CS

          wait (mutex) ;
              readcount = readcount - 1
              if (readcount == 0) then signal(wrt) ;
          signal (mutex) ;

Writer :  wait (wrt)
                  CS
          signal (wrt)
```

Writer may go to starvation due to other readers

* Dining philosopher ;

repeat
wait (chopstick [i]);
wait (chopstick [i+1 mod 5]);

CS

signal (chopstick [i]);
signal (chopstic [i+1 mod 5]);
until false;

=> Problem of deadlock

=> Solutions : i) allow atmost four philosopher to be sitting simultaneously.

ii) Allow a philosopher to up the chopsticks only if both of them are available.

iii) use a asymmetric solution, that is an odd philosopher picks up her left chopstick and then her right chopstick.

Lect-08

BOSS
Page No.
Date: / /

* Monitor :

Mutex : Binary semaphore initialized to '1'.
[Required to enter into monitor - guarantees atmost 1 Procel active]

Next : Binary semaphore initialized to '0'.
[used when a procels wakes up another procels & Block itself w/o any condition]

condition_sem (x_sem): Binary semaphore for a particular condition X & is initialized to 0.

Next_count : At any point of time no. of procels waiting for next.

x-count : No. of procels waiting for x-condition

monitor :

T_1	T_2	T_3	T_4	T_5	T_6
W(A)					
	R(A)				
		R(A)			
		W(B)			
R(B)					
			R(B)		
			W(c)		
	R(c)				
				R(c)	
					W(D)
	R(D)				
				R(D)	

```
wait (mutex);
...
body of F;
...
if next-count > 0
    then signal (next)
    else signal (mutex);
```

mutex = 1
A_count = 1
B_count = 0
C_count = 1
D_count = 0
next_count = 0

A_sem = 0
B_sem = 0
C_sem = 0
D_sem = 0
next = 0

Mutual exclusion with a monitor is ensured.
We can now describe how condition variables are implemented. For each condition x, we introduce a x_sem and an integer variable x_count, both initialized to '0'

x.wait can be implemented as

```
x_count = x_count + 1;
if next_count > 0
    then signal (next)
    else signal (mutex)
wait (x_sem);
x_count = x_count - 1;
```

If read operation then check condition for waiting.
If write operation then do not wait check code is waiting for this operation.

x.signal can be implemented as;

```
if x_count > 0
    then begin
        next_count = next_count + 1
        signal (x_sem);
        wait (next)
        next_count = next_count - 1
    End;
```

* Deadlock :

1] Mutual exclusion
2] Hold and wait
3] NO preemption
4] Circular wait

} Necessary conditions.

Multiple procels & multiple instance of single type of resource.

No. of procels → n

~~# procels~~

1 Resource Type(R) → m instance of R

Assumption : Each procels requires atleast 1 instance of R.

$$1 \leq \text{max Need} \leq m$$

Eg : Each procels max. need = k instance of R what should be min. no of instance of 'R' to ensure that the system is deadlock free ?

⇒ No. of procels = n

$$\text{Total no. of resource} = n(k-1) + 1$$

* Multiple procels multiple Resource [Each resource has single instance]

Construct Resource Allocation Graph

Type of Edge

1) Allocation Edge : i^{th} Resource allocated to i^{th} procell then we have an edge from R_i to P_i

2) Requelt Edge : if i^{th} procels Requelts for j^{th} resource then we have an edge from P_i to P_j

$P_i \longrightarrow P_j$

* Deadlock prevention :

=> In deadlock prevention we try to prevent deadlock by ensuring that atleast one of these condition cannot hold

* Deadlock avoidance :

- Bankers's algorithm
- Resource requelt algorithm

* Safety algorithm :

m → no. of Resource
n → no. of Process

⇒ The algorithm to find whether the system is in safe state or not

1. Let work and finish be vectors of length 'm' and 'n' resp. $O(m+n)$
   Initialize work = Available and finish[i] = false
   $O(m)$ $O(n)$

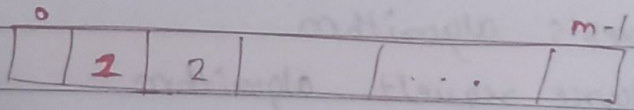
2. Find an i such that both $\left.\begin{matrix} m(n+n-1+n-2+\dots 1) \\ = m \times n^2 \\ = O(m.n^2) \end{matrix}\right\}$
   a. Finish [i] = false
   b. Need i ≤ Work
   if no such i exists, go to step 4.

3. Work = Work + allocation; $\left.\begin{matrix} \\ \\ \end{matrix}\right\} O(n.m)$
   Finish[i] = true
   goto to step 2

$O(n)$ 4. If finish[i] = true for all i, then the system is in a safe state

Total complexity = $O(m.n^2)$

NOTE : i) work is a single Dimensional array.



only 1 and 2 are available

ii) for step 2 : we compare m resources with 1st process till n^{th} process and then we will have (n-1) process.

we found in the last comparision that it is not allowed.

* Resource request Algorithm :

1. If Request$_i$ ≤ Need$_i$ goto step 2 — O(m)

2. If Request$_i$ ≤ Available goto step 3 — O(m)
otherwise, P_i must wait since the resources are not available — O(m)

3. Available = Available − Request$_i$; — O(m)
Allocation$_i$ = Allocation$_i$ + Request$_i$; — O(m)
Need$_i$ = Need$_i$ − Request$_i$; — O(m)

If the resulting resource-allocation state is safe, then the transaction is completed and process P_i is allocated its resources. However, if the new state is unsafe, the P_i must wait for Request$_i$ and the old resource allocation state is restored.

Lect - 14

Page No.
Date : / /

* Process scheduling :

1] FCFS :

=> Advantage : i) It is a fair policy
ii) Does not suffer with starvation

=> Disadvantage : i) Suffer with convoy effect
If a larger process is allocate first then all other smaller process will have to wait for longer duration

2] SJF :

=> Advantage : i) Avg. waiting time is min for Non-preemptive process.

=> Disadvantage : i) It suffers with starvation
ii) very difficult to predict execution time.

τ_{n+1} → next predicted burst time
τ_n → current predicted burst time
t_n → current actual time
α → 0 to 1

$$\tau_{n+1} = \alpha \tau_n + (1-\alpha) t_n$$

3] SRTF :

↳ suffers from starvation.

4) static / Dynamic priority algorithm :

→ Suffers from starvation

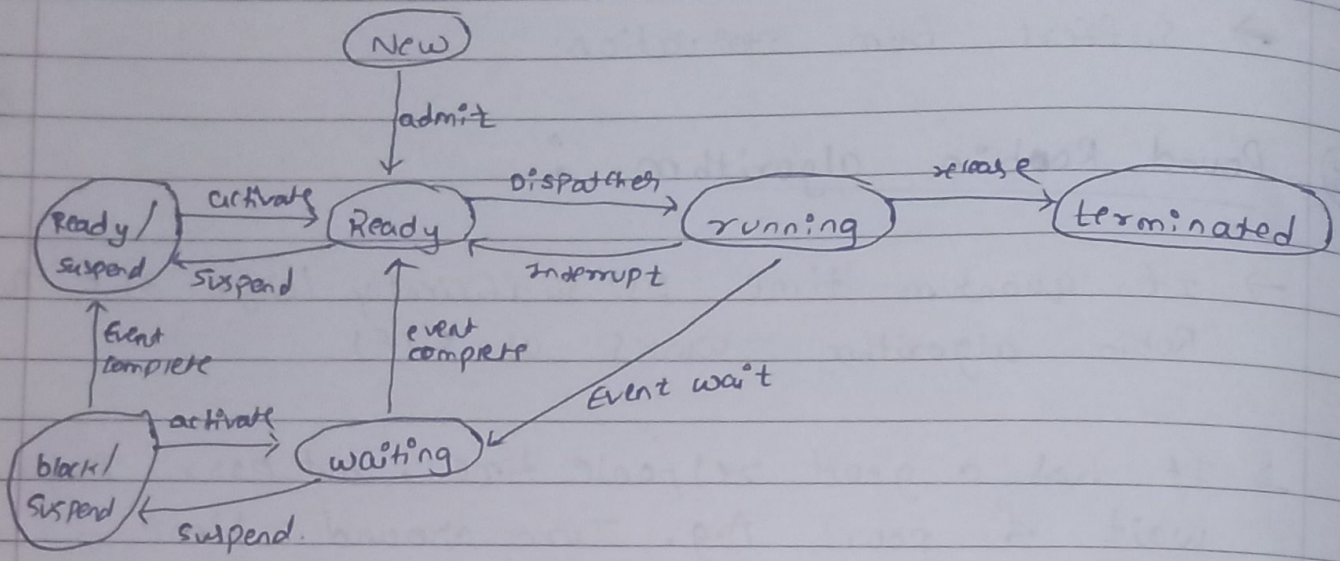
5) Round Robin algorithm :

→ If quantum time is sufficiently large the Round Robin algorithm works as FCFS

→ It has a good response time, but poor Avg. wait & poor Avg. Turn around time

* Multilevel queue	Multilevel feedback queue
1) Ready Queue is partitioned in several smaller queue	1) Ready Queue is partitioned in several smaller queue
2) Process are permanent in the queue	2) Process are not permanent in the queue they can move from one queue to another.
3) Process are divided on basis of memory size & priority.	3) Process are divided on CPU burst characteristics
4) Process are divided in foreground & Background process	4) Process are classified as high & low priority process
5) It has starvation issue.	5) It prevents starvation

* Extended (seven) state process model :



* Necessary and sufficient condition for scheduling for periodic process.

⇒ $\sum_{i=1}^{n} \frac{\text{CPU Burst}}{\text{Period}} \leq 1$ } Condition for EDF [Earliest Deadline first]

* Rate Monotonic scheduling (RMS) is an optimal fixed-priority policy where the higher the frequency (1/period) of a task, the higher its priority.

* Multi-thread processes shares :

⇒ Heap Memory
⇒ Global variables

* Multi-thread process does not share :

⇒ Stack Memory
⇒ Registers.

* Previliged instruction :

⇒ Any instruction that can run only in kernel mode is known as previledged instruction.

⇒ If any attempt is made to execute the previlidged instruction then H/w interupt will occur.

1) I/o instruction
2) Halt instruction
3) turn off interupt
4) set the timer
5) Context switching
6) Clear memory
7) Remove prolcss from memory
8) Modify entire in device status Table

* Thread management is handled by thread library in user space.

* In non-preemptive scheduler, process state changes from running to waiting, and after I/o completion it moves to Ready state and waits for CPU.

* In priority scheduling, I/O bound procell should be given higher priority than CPU bound processes in order to make more efficient use of CPU.

↳ CPU bound process will use CPU for short amount of time and perform I/o operation in that time CPU can be utilized.

* If A and B are threads of same process, A can always read B's stack.

↳ Threads in the same process share the same address space, meaning they can access each other's stacks, even though they have their own seperate stack regions within the shared space.

* After a fork system call, if a parent terminates before its child process then the child process becomes an orphan.

↳ Orphan process are adopted by init process in Unix like OS.

* The Return value of the fork is the child's PID in the parent process

* NO. of process created is NO. of reaves in fork system call.

* The Thread ID (TID) is managed by TCB rather in the PCB. Each thread within a process has a unique identifier Stored in its TCB.

* Livelock occurs when two or more processes continously change their states in response to each other's actions, without making progress. process are not blocked in livelock.

* Compaction :

⇒ Compaction is the process in which the free spaces are collected in the large memory chunk to make some space available in memory.

↳ Helps to remove external fragmentation.

* File allocation Method :

1) Contiguous Allocation :

⇒ A single continuos set of block is allocated to a file at time of allocation.
It allows random access of block.

Disadvantages : 1] External fragmentation will occur.
2] comparction will be necessary to free disk
3] With preallocation it is necessary to declare file size at time of allocation

2] Linked Allocation [Non-contigous]

⇒ Allocation is an individual block basis, each block contains pointer to next block in the chain, blocks need not be continous, NO external fragmentation size of file can increase with availability of free blocks.

Disadvantages : 1] Internal fragmentation can occur in the last block.
2] overhead of maintaining a pointer in every block.
3] If the pointer of any disk is lost file will be truncated.
4] only sequenced access is allowed.

3] Indexed allocation :

=> It addressed many problems of contigous & linked allocations

- FAT contains a seperate 1-level index for each file, the index has
  - one entry for each block of file
  - Allocation may be based on fixed or variable sized block
  - Allocation by fixed blocks eliminates External fragmentation & variable sized blocks improves locality of reference
- It supports both sequential & direct access