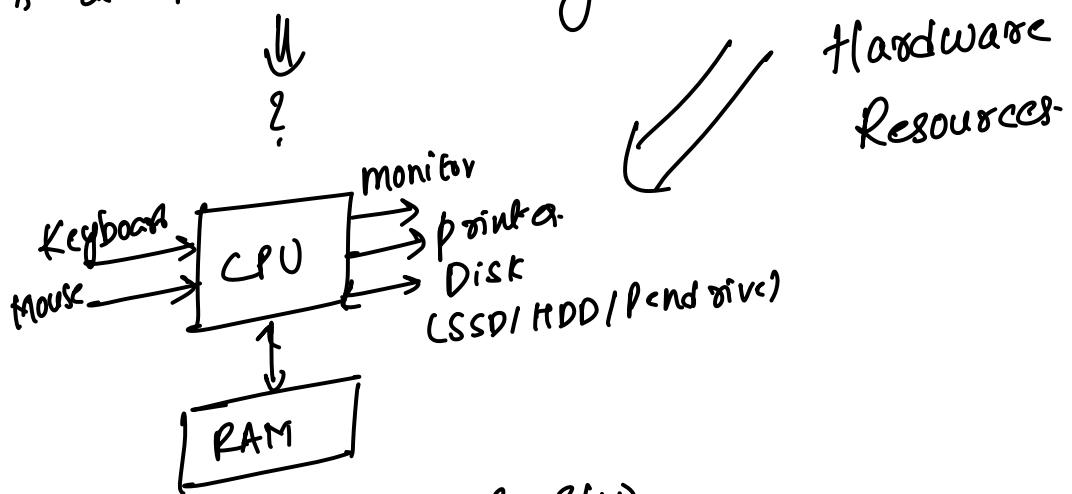


Operating
Systems

P. Sailesh
Kumar

Operating Systems

1) It is a Resource manager.



Resources can be ff(w & S/W)

Make the life easy for S/W engineer

Make the life easy for S/W engineer

(like how pointer works internally)

(like how engineer doesn't need to know)

the S/W engineer doesn't need to know)

* OS can be broadly considered as a resource manager.

* OS can be broadly considered as a resource manager.

Eg Browser runs on our S/W.

Eg Search a file, OS provides this utility, basically

Eg it talks to hardware-

* OS is a piece of software (Most OS's are implemented in C/C++)

A program can be broken into multiple processes

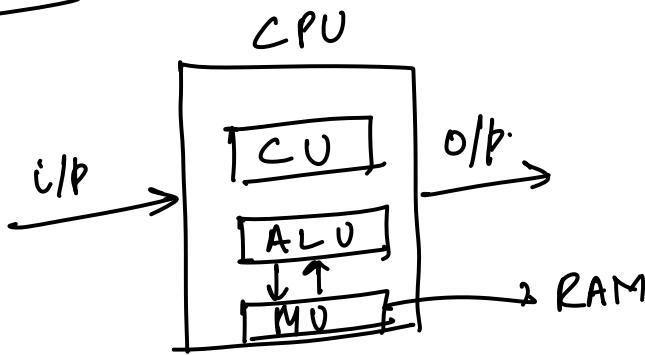
each process uses CPU's resources.

We can see list of processes in our task manager (Windows)

Important topics

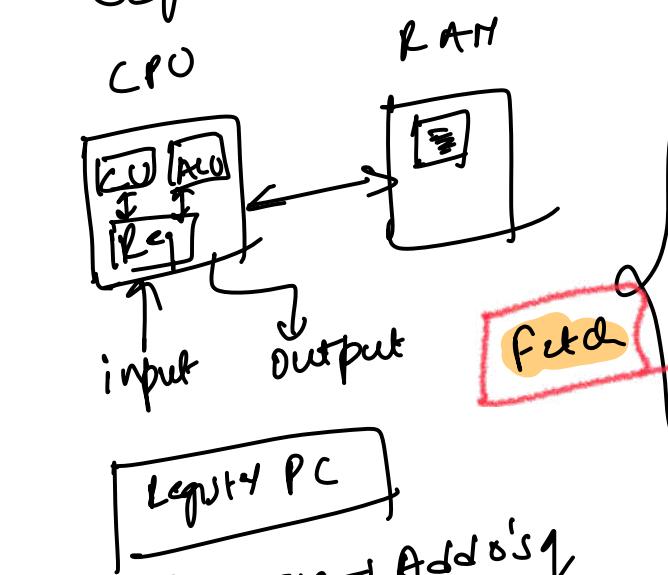
- 1) Process management
- 2) Memory management.
- 3) File Systems & Device management.
- 4) Protection & Security Mechanisms

Let's understand Architecture first



Any program we want to execute, it is stored in RAM.

↓
Sequence of instructions.



PC → Program → Address of current instruction.

MAR → Memory address register

MDR → Memory data register

CR → Current instruction register

Instruction cycle

- 1) Address in PC is copied to MAR
- 2) Increment PC
- 3) Inst. found at MAR is copied to MDR
- 4) Inst. in MDR is copied to CR
- 5) CU takes over now & decodes the CR
- 6) CU sends signal to ALU

Execute

RAM
 $(02 : ADD C1) O((21))$
 $\quad\quad\quad (14 \quad 13)$

Take val stored at 120
 (21)
add them & store back
at 120

Interrupts:

main()

{
int i;

int j = 3 + 2;

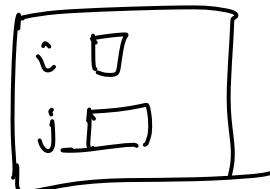
j = j * 2;

scanf("1-d", &i); // keyboard
if p

i = i / j;

printf("1-d", i);

y



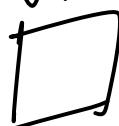
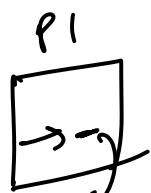
Now CPU is
waiting for keyboard
& hence it is a
interrupt

CPU is in idle state

Print to
monitor

Uni VS Multi-programming

Uni \rightarrow one program executed at a time
huge wastage of resources



Interrupted

CPU is
idle now

P2 is
not
executed

Multi
multiple programs executed
simultaneously.

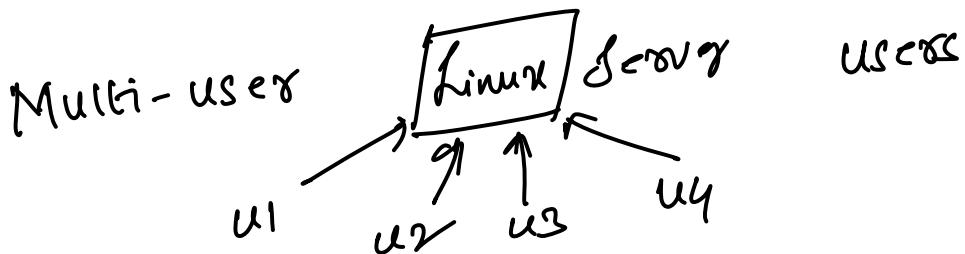
We want P2 to
load in RAM
when P1 is not
in execution

Multi-programming vs Multi-Tasking

(Unix Terminology)

(Windows)
Terminology

At the root both
are same:



Types of multi-programming

1) Preemptive
(forcefully removing)

→ Force a process
to give away control
of resources in CPU

Eg win10, Linux, Android

OS can interrupt the
current program from
running at any given
point of time to let another
program use the CPU

2) Non-preemptive

process has to giveaway control
itself.

↳ completion

↳ I/O event

Eg win3.0 & win3.1

Non-preemptive OS lets a program
run, until it finishes or gives
up voluntarily.

Pre-emptive multi-programming
is
Today's world

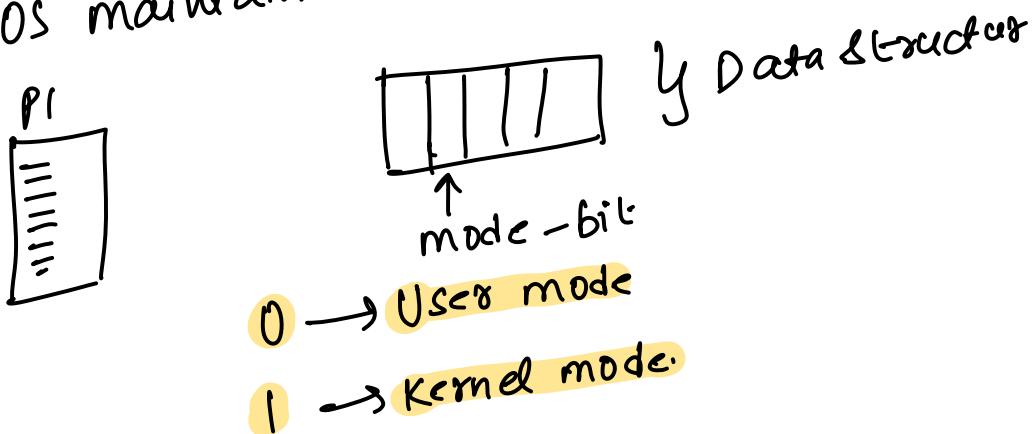
Modes of execution in CPU

i) User mode

ii) Kernel mode

- core of the operating system
- controls & manages most of the resources
- Non-preemptive (No one can alter it)
- Atomic execution (Everything is executed)

for every program/process there is a data structure
that OS maintains called as PSW (Process Status Word)



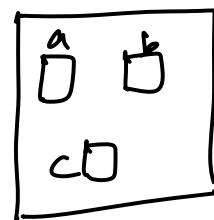
```
main()
{
```

```
    int a,b,c; // user mode
```

```
    c = a+b;
```

fork(c); → System call

```
    printf("Hello");
```

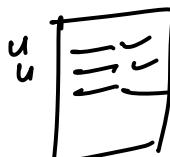


Sys-call

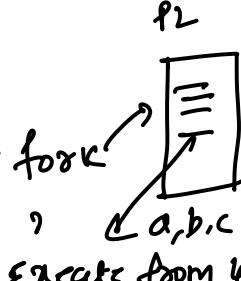
calls
the
OS



Y



a,b,c



a,b,c

execute from 4th line

create new process

copy instructions
&
create new process

`fork → sys-call → OS`

(Kernel mode)

→ duplicate

Execute point in P2 & go back to P1

P1 → Parent process.

P2 → Child process.

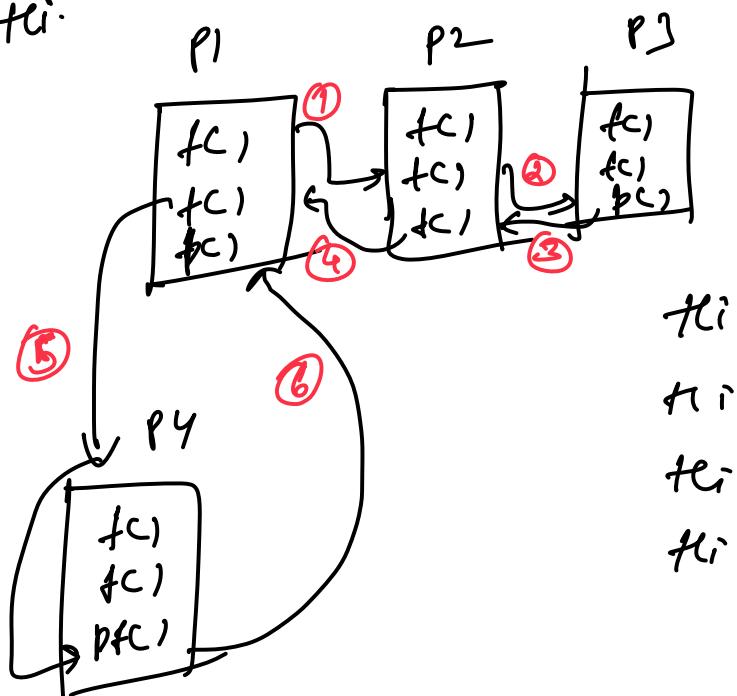
- * fork copies the content of parent process & duplicates it in child process
- * In child process, the line point to tci is executed (line after fork)
- * It comes back to parent process, points to tci so $tcli$.

Eg 2.

```

main()
{
    fork();
    fork();
    ptc("tli");
}

```



$2 \text{ fork} + 2^1 + 2^2$

$n \rightarrow \text{fork} \Rightarrow 2^n \text{ processes}$

Just illustrate it manually.

Eg:

The following C program is executed on a Unix/Linux system :

```
#include<unistd.h>
int main()
{
    int i;
    for(i=0; i<10; i++)
        if(i%2 == 0)
            fork();
    return 0;
}
```

The total number of child processes created is $2^5 - 1 = 31$.

0 9
0, 2, 4, 6, 8 } 5 times
▶ fork(10)
 2^5 processes

$$2^5 - 1 = 31$$

Process Management

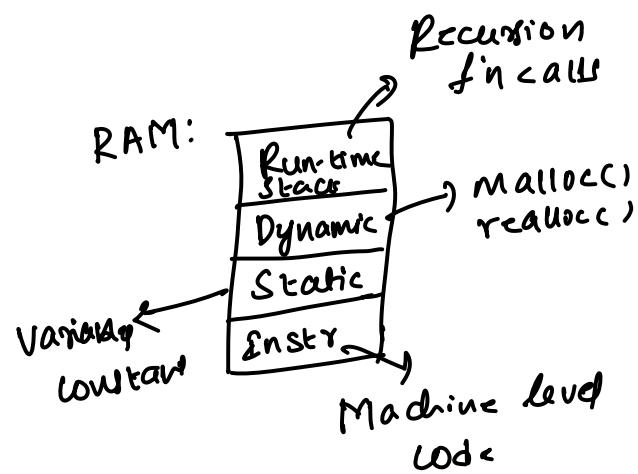
Program under execution in RAM with resources allocated to it (CPU/RAM/IO)

- * Single program can create multiple processes.
- * Unit of execution.

Process in Memory

Static \rightarrow Load time

Dynamic \rightarrow Run-time



Operations on Process

- 1) Create \rightarrow Allocate resources (Allocating space in RAM by CPU's)
- 2) Schedule \rightarrow CPU's, I/O (which should run first)
- 3) Block \rightarrow I/O call (Maybe like interrupt like input from keyboard due to System call's)

- 4) Suspend → whatever process is not important
remove from RAM and keep it in
Disk (Swap out from RAM to disk
temporarily)
- 5) Resume → Starts re-executing it again
- 6) Terminate → Process is completed.
(Resources/Memory everything
is de-allocated)

Attributes

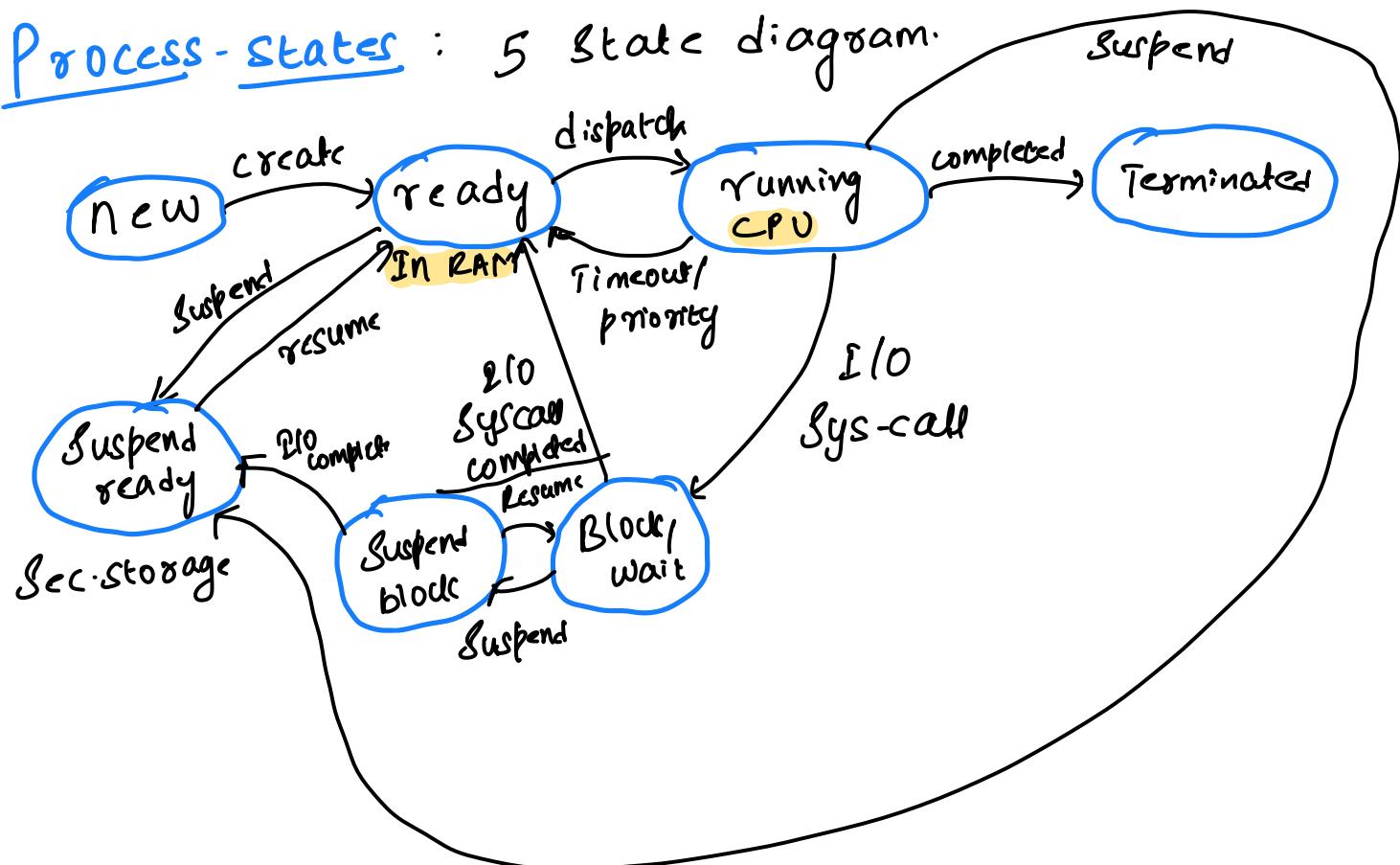
- 1) Process identification: PID
- 2) CPU-related : Program Counter
(Helps the CPU what needs to be
executed next),
CPU, priority states
- 3) Memory related : → memory limits of a process
page-tables
- 4) File-related: → file id's, file's open file pointers
- 5) I/O-related → Keyboard, open-devices
- 6) Protection related → mode PSW
 - 0 → User mode
 - 1 → kernel mode

Process Control Block (PCB)

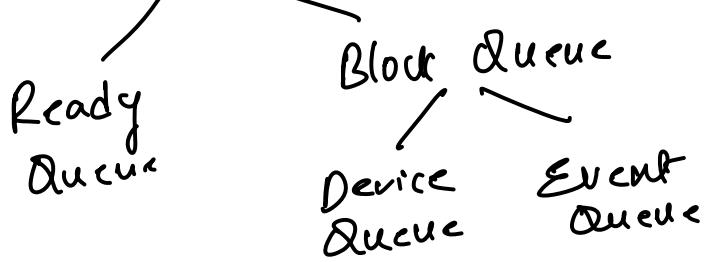
It is a data-structure which has all the above attributes

Every process has a PCB

Every process has a unique PCB.



Scheduling Queue



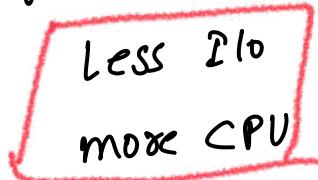
Suspend Queue & Job Queue
 Input Queue
 ↓
 all processes in Suspend state
 waiting to be loaded in memory

I/O Bound vs CPU Bound processes



Disk read
 ↳ 2GB file
 copy to another location on disk

1) Requires lots of CPU
 e.g. lot of mathematical & logical operations.



what is dispatcher & context switching

* Gives control of CPU to a process selected by the scheduler

i) completion of a process \rightarrow PCB is destroyed.



- wait for STS (new process)
- PC, GPR...
(Load at PCB of new process)
- Load the registers into CPU registers, give control to the control Unit & start execution

② Requires I/O

- Save the state of PCB (current state) in main memory
- Move the corresponding PCB to block queue
- wait for STS & load for new process
- context switching \rightarrow switching from one process to other process.

③ Preemption
STS forces preemption (priority higher)

Again save current state of PCB
Load the new process

Remove
process

Add New
process

y context
switching

m - processes

n - CPU's

what is lower bound & upper bound

	min	max
Ready :	0	m
Running :	0	n - running
Blocked :	0	m - blocked

CPU Scheduling Timings

(There are multiple scheduling algorithms)

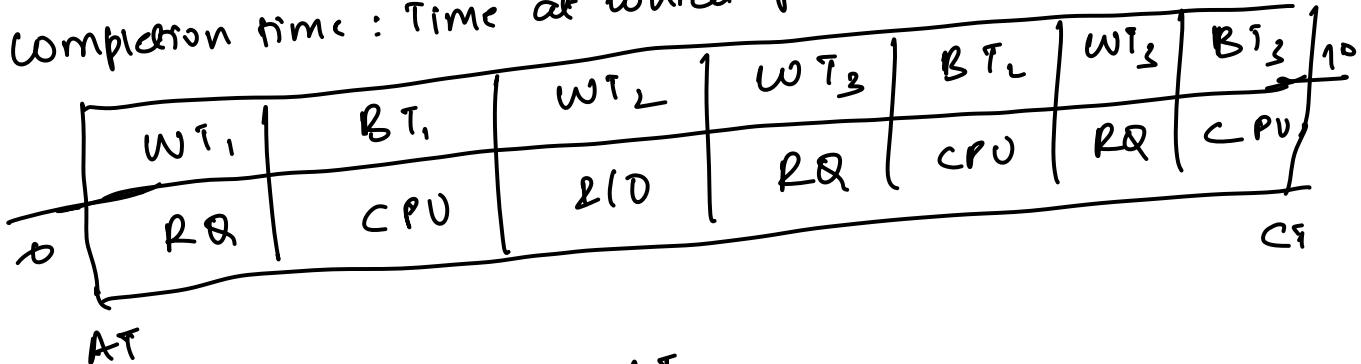
1) Throughput : No of process / Unit time (completed)

2) Arrival time (AT) : Process time, enqueued in the Ready Queue

3) Wait time (WT) \leftarrow CPU wait time
 \leftarrow RQ wait time

4) Service time / Burst time : Time in process - running state
(BT)

5) Completion time : Time at which process is completed



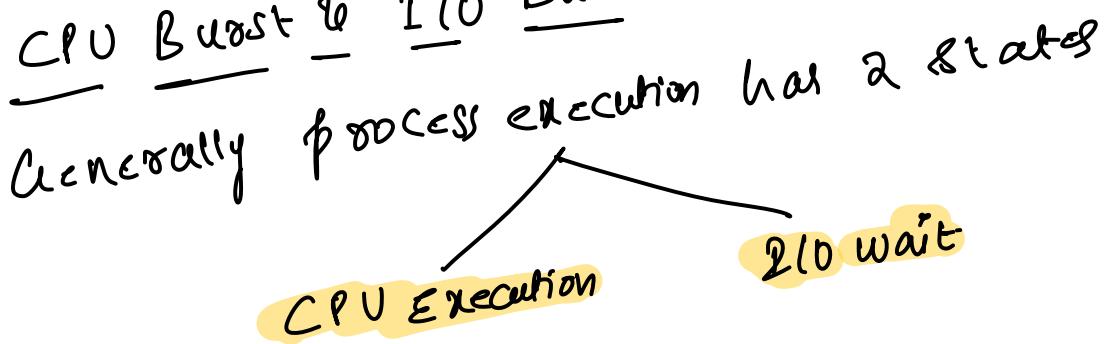
Turn-Around time : CT - AT

CPU Scheduling

By switching the CPU among different processes, the OS can make the computer more productive.

- * We don't want CPU to sit idle
- * Several processes are kept in memory
- * During this time when one process has to wait, the OS takes away the CPU and it gives the CPU to another process.
- * Which process should get the CPU? which has to wait?
- * The OS has to decide which process gets the CPU. It is done through different algorithms.

CPU Burst & I/O Burst



CPU Burst: time under CPU execution

I/O Burst: time under I/O wait

CPU scheduler selects the process that will get the CPU to execution. (Select from ready queue)

Dispatcher: It will give the process to CPU, which was selected by the scheduler.

- * Process switching from running to wait
 - * Running to ready state (Interrupt) (Because N/O occurred to go to wait-state)
 - * waiting to ready state (I/O)
 - It was running initially
 - P/I occurred
 - went to waiting
 - I/O complete
 - comes from waiting to ready
 - * Process terminated
- 2, 3 \rightarrow Preemption
(Before completion
CPU is taken away)

Scheduling Criteria:

- 1) CPU utilization \rightarrow Ideally 40 to 90%
Keeping CPU as busy as possible
- 2) Throughput: No. of processes completed per Unit-time
(By CPU)
- 3) Turn-Around time : waiting to get into memory
Waiting in ready Queue
CPU executing & doing I/O
Complete time
(In terms of processes)
- 4) Waiting time: Time spent in ready Queue.

