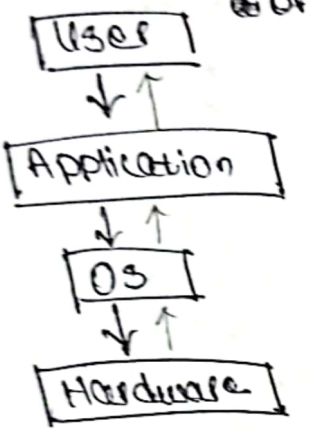
LECTURE 1

SUNDAY DATE: 28/05/2023

INTRODUCTION [LECTURE 1]

OS: A program that acts as an intermediary between user of a computer & the computer hardware



Major Goals of OS:—

- execute user programs
- make computer system convinient to use.
- use the computer hardware in an efficient manner.
- manages and allocates all resources
- controls the execution of user programs & operations of I/O devices.

System software: OS and all utility programs that manage computer resources at low level eg compilers, loaders, linkers, debuggers.

Application software: programs desing for an end user.
eg. word processors, database systems, spreadsheet programs.

Timeline of OS → (See slide)

LECTURE 2

TUESDAY DATE: 30/05/23

Components of a computer system:

* Prompt goes from user to compiler.

System programs → compilers

Application programs → database system

Kernel — is in RAM.
— central module of OS
— Kernel starts and initialise the other.
— Kernel is not of limited amount.
— The smaller in size the kernel, the more it can store.
— Kernel must be in protective layer and cannot overwrite.

Bootstrap program — starts ~~the~~ and initialises the kernel.
— it is in other device (ROM or EEPROM) and not in RAM. As we need to install the OS again & again if bootstrap is stored in RAM.

Storage structure

→ main memory — RAM
— volatile (if it's turned off, all the programs are turned off)

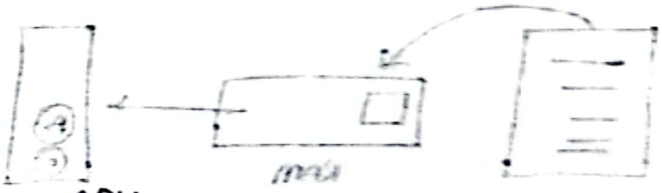
→ secondary memory — non volatile
— needs to be initialised ~~this~~ by RAM.

* Every program always takes instructions from RAM

Storage Device Hierarchy

registers
↑ ↓
cache
↑ ↓
solid-state disk
↑ ↓
magnetic disk
↑ ↓
optical disk
↑ ↓
magnetic tapes

speed ↑
storage ↓
size ↓
price ↑



OS Architecture

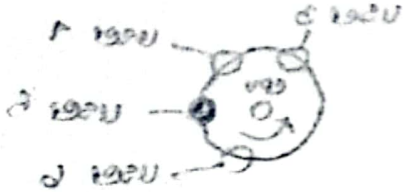
(1) Single-Processor System → One CPU

- processors are queued before being operated by CPU.
- Parallel execution is not possible

(2) Multiple-Processor System → 64 CPU

- Parallel execution can be done.
- Execution time is less than single processor system

Bus communication system



OS operations

Interrupt based operations — notification is send to OS which helps OS to decide what to do.

— Hardware generated notifications.

Trap — Software generated notifications

— eg exception.

Dual mode operation:

*Single mode operation :—

User process executing → calls system call → execute system call

programmer writes codes — make triggers in hardware

*All these operations are done in Kernel. Here, programmers can update and overwrite in Kernel.

*Dual mode operation:—

User process

User process executing → calls system call

Kernel

trap
mode bit = 0

execute system call

return
mode bit = 1

return from system call

User mode
(mode bit = 1)

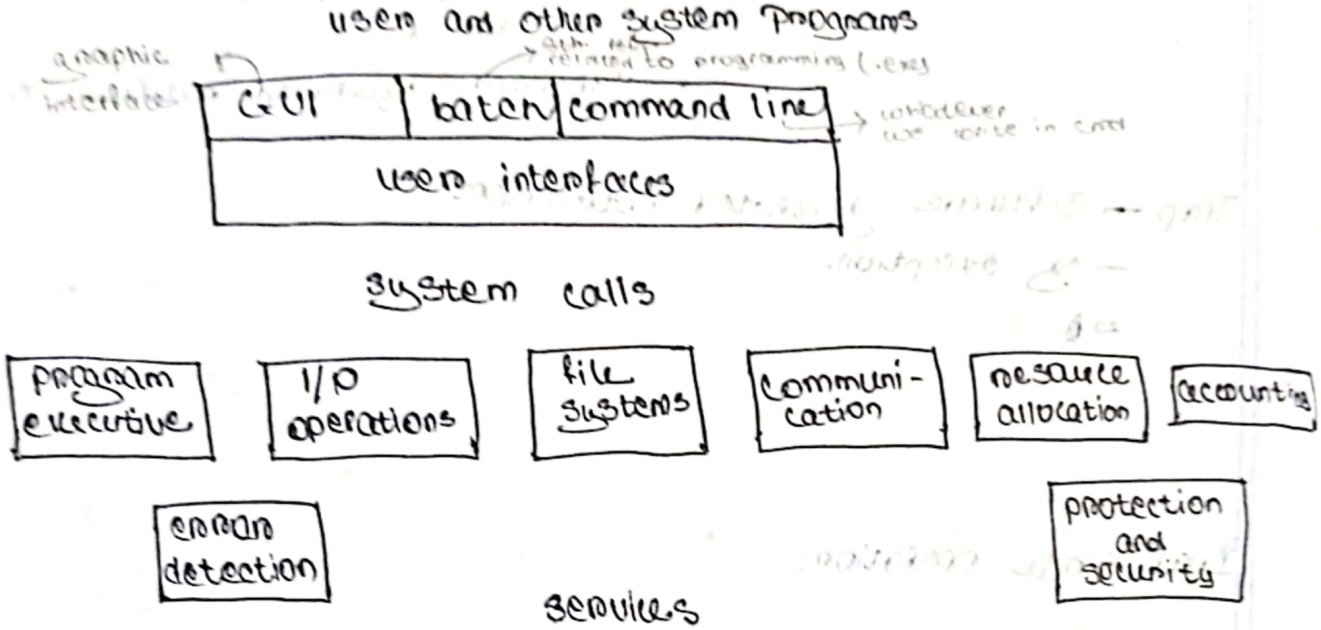
Kernel mode
(mode bit = 0)

LECTURE 3

SUNDAY

DATE: 01/06/23

OS SERVICES



SYSTEM CALLS

- creates connection between user and kernel.
- A function in user mode calls the system calls.
- Through the system calls, there is changes in the kernel and there is a change in the hardware.
- The result in the hardware then returns back to the user through the system calls.

System call (SYS) function
AX SYS1, function AX2.

If the content of one file is copied and pasted in another file:-

Input two file name → Open the input file, and create the output file → Read from input file, and write into output file → Close the file

System calls:

- Acquire input file name
  - write prompt to screen
  - accept input
- Acquire output file name
  - write prompt to screen
  - accept input

Open the input file; Open( )
- if file doesn't exist, abort

Create output file; Create( )
- if file exists, abort

Loop
- Read from input file; Read( )
- write to output file; write( )

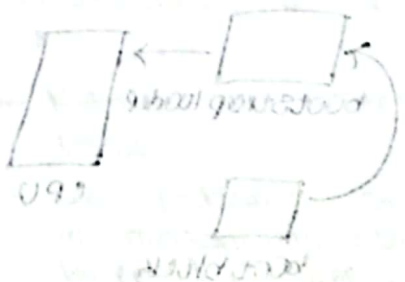
Until read fails

Close output file; close( )
write completion message to screen.
Terminate normally.

SYSTEM CALL INTERFACE

- System call is initiated from user mode.
- If the system call has a number in the table, then the system call is initiated in the Kernel.
- The status of the system call is then returned.

(Function)	Line no. where function
open( )	128
close( )	250
create( )	300
⋮	⋮



SYSTEM SOFTWARE

Helps in :-
file management
Status information
file modification
programming - language support

- Provides convinent environment for program development and execution.

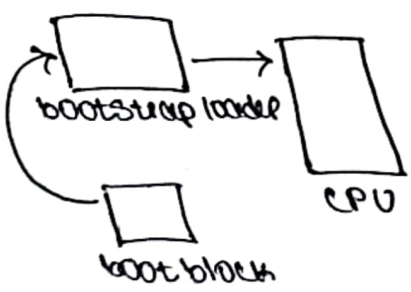
~~3Fs~~

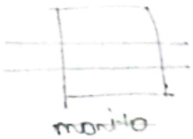
SYSTEM BOOT

- bootstrap program.
- It is in ROM or hard drive.

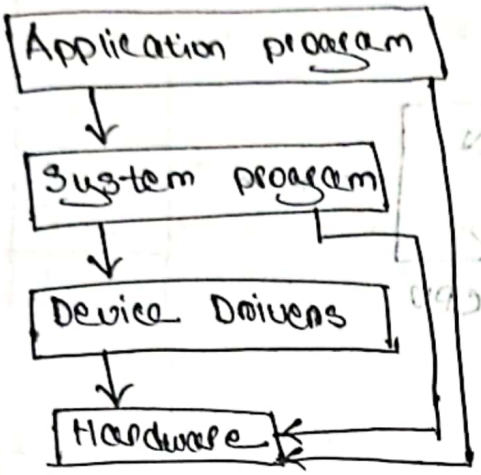
(1) Bootstrap loader finds the kernel and initialises it.
(2) Boot block

* Boot block is small in size so can be loaded faster than bootstrap
* Boot block is in fixed position which is loaded by ROM. Loads bootstrap loader from disk.

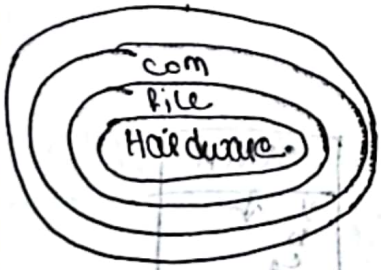




Process
Deadlock
Memory management



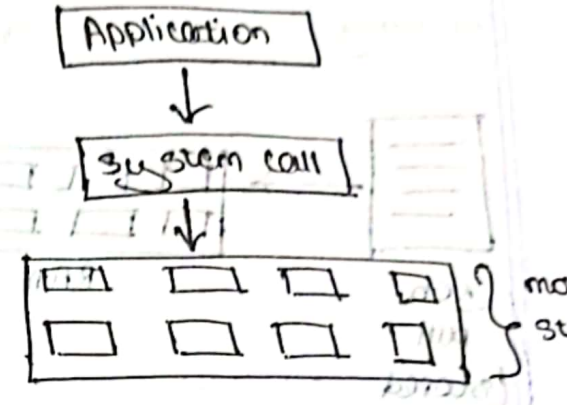
Here, using system program to change hardware, one can make changes to OS which is exploiting the OS



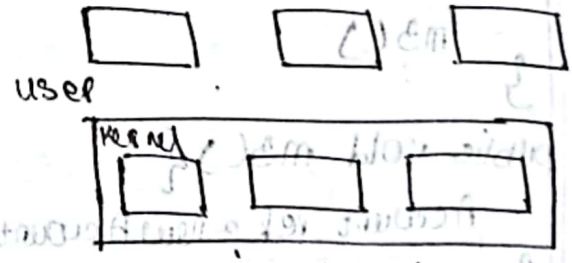
উপরে থেকে নিচে আসতে সময় নিচে থেকে উপরে আসতে সময় লাগবে না.

layered structure

- indirect loop from application to hardware slows down operations.
- Due to seperate, error in one layer can be handled seperately.
- ~~Higher level~~ each layer can use services of its lower layers.



- Difficulty in debuging & implementation
- ~~storing~~ more storage is used in kernel



micro kernel

* some services are stored in user and others are stored in kernel.
- ~~Kernel~~ micro kernel is used.
- the problem of space in monolythic structure is resolved here.

LECTURE 1

TUESDAY

DATE: 06/06/23

PROCESS [LECTURE 2]

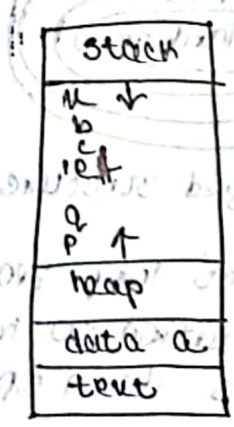
job pull (stored in hard disc/ROM)

process → program is in execution (active entity)

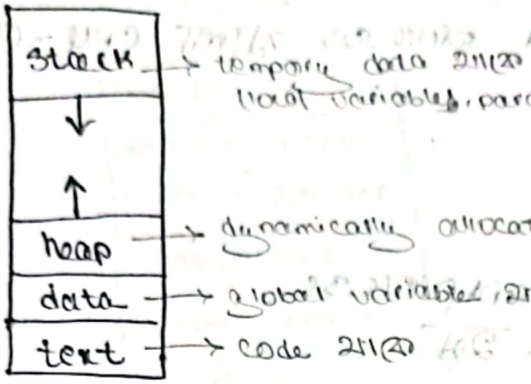
program → text section of a process (passive entity)

```
a = 200
public void m1(){
    int x = 20;
    m2(10);
}
public void m2(int b){
    boolean c;
    m3();
}
public void m3(){
    Account ref = new Account();
}
Account{
    int p = 0;
    int q = 0;
}
```

→ System class



* stack → variables up (grows down → towards)
* heap → variables down (grows up → towards)

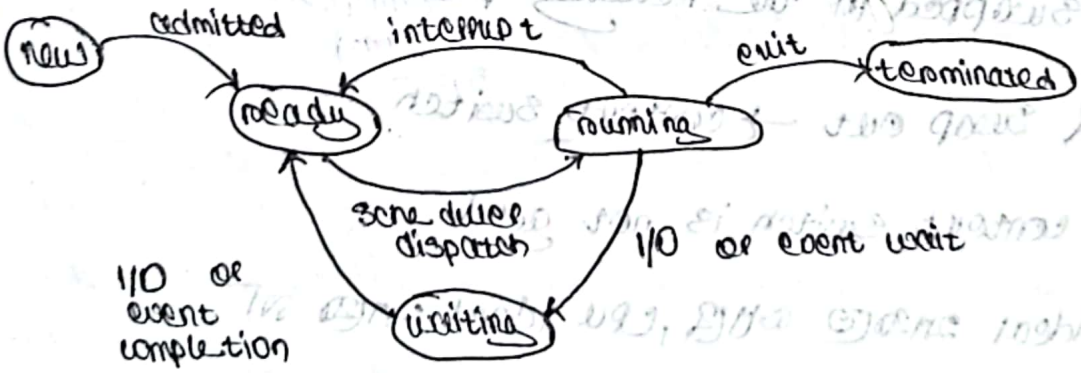


program → any code is a program

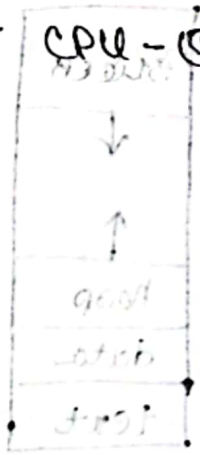
process → when a program is in RAM, it is process

PID → After the program is in RAM, it is assigned with a unique ID called PID.

PROCESS STATE DIAGRAM



CPU scheduler: কোন sequence এ process গুলো CPU-তে execute হবে।



Job/process/task — same

fork() — duplicates a child process
— child process create হয়

CPU bound processor — spends more time in computation using processors than I/O

I/O bound process — spend more time in I/O than CPU.

Long term scheduler must be selected wisely.

Medium term scheduler: partially execute হয়ে ready queue তে গিয়ে যায়।

Partially executed swapped out processes: partially complete process is swapped out and then saved in PCB and swapped in the ready queue.

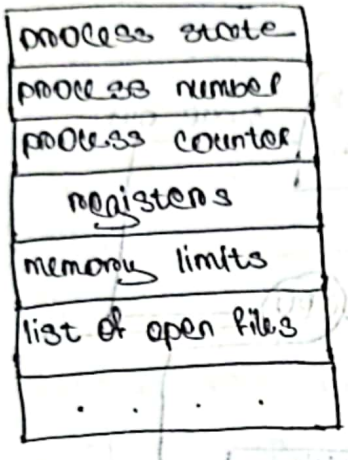
Swap in & swap out → context switch

*Frequent context switch is not good.

*process 'ideal' থাকতে পারে, CPU 'ideal' থাকতে পারে না

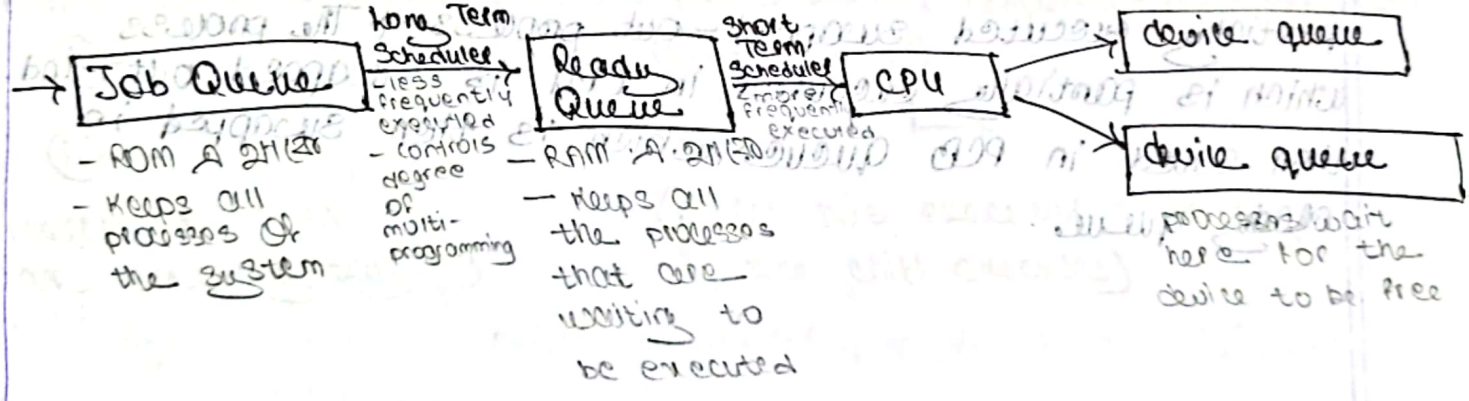
*PCB is in RAM.

REPRESENTATION OF PROCESSES IN OS



PCB: Process Controller

SCHEDULING QUEUE

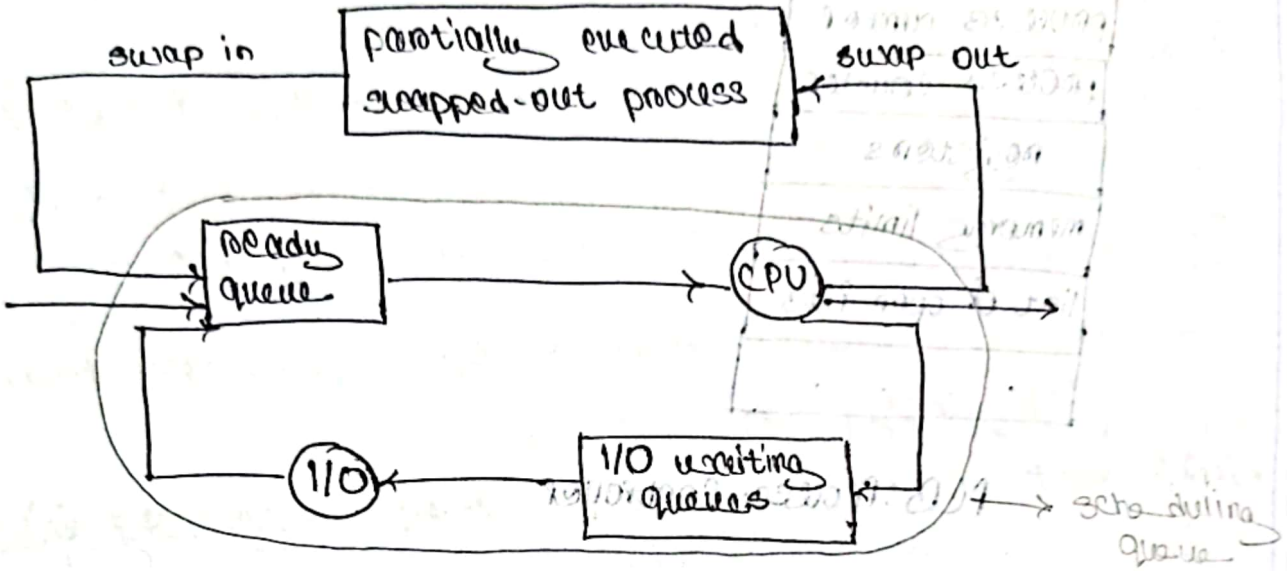


- ROM & 2ndary
- Keeps all process of the system

- RAM & 2ndary
- Keeps all the processes that are waiting to be executed

processes wait here for the device to be free

MEDIUM TERM SCHEDULER



partially executed swapped-out process → The process which is partially executed in CPU is swapped out and then saved in PCB queue, which is then swapped in ready queue.

CONTEXT SWITCH

When an interrupt occurs, the system needs to save the current context (state) of the process running on the CPU.

Context Switch: (1) Storing currently executed process content

(2) Restoring the next process content to execute

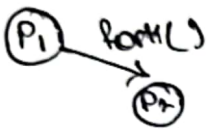
LECTURE 8

SUNDAY

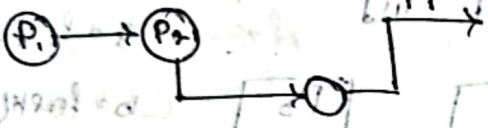
DATE: 11/06/23

PROCESS CREATION

* When there is a big task or is busy, parent process can, create child process.
* Parent & child processes have same code.

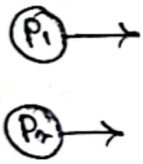


* Every process has unique process id

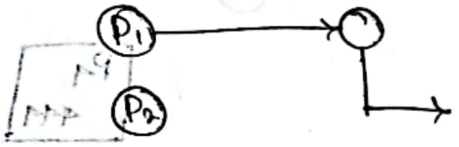


(i) Child process is created by parent.
(ii) Child process is executed first, & parent waits

(iii) After termination of child, parent process completes task & terminates



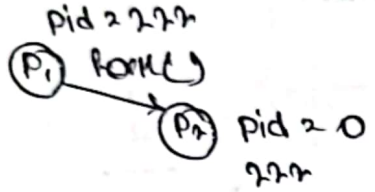
(parent & child are executing together)

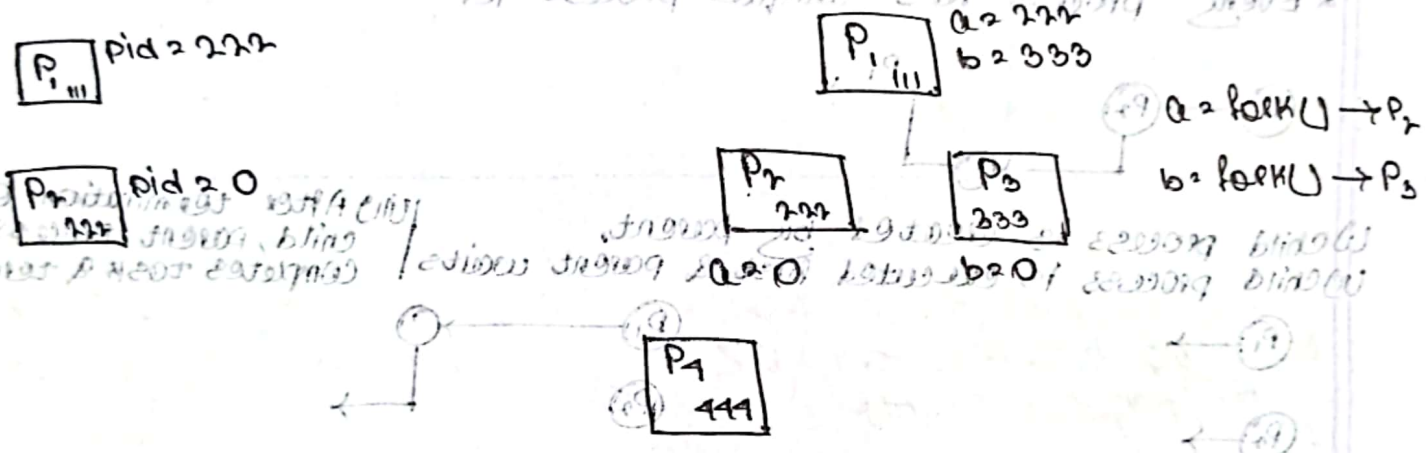


(parent has executed completely & then child executes)

fork() : create a child process

* fork() use ~~a~~ identifier variable use ~~as~~ (pid)



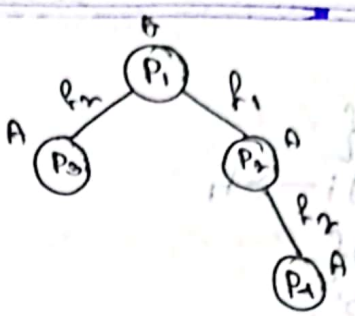


exec() : execute/replace. Either parent or child will be in CPU.

wait() : when wait is called parent process will wait until child process completes execution and is terminated. The parent process kills the child process.

* If wait() is not ~~created~~ called, then the child process holds on resources in the OS. ~~and does not kno~~ And, parent process can terminate b4 the child process leaving behind a zombie process.

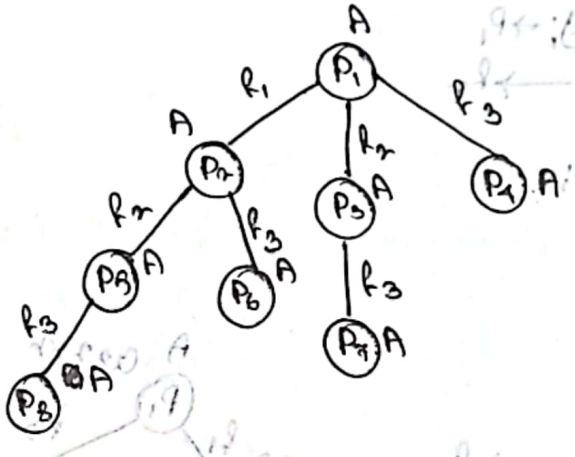
(1)
```
int main() {
    fork(); → f1
    fork(); → f2
    printf("A\n");
}
```



* child execute করবে তাই 2টি 2টি করে 4টি line (2²=4)

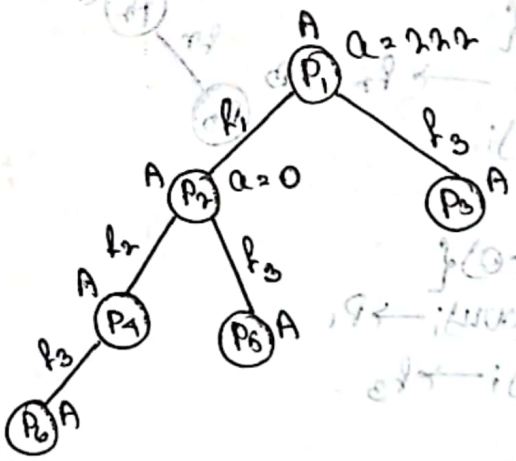
AAAA

(2)
```
int main() {
    fork(); → f1
    fork(); → f2
    fork(); → f3
    printf("A\n");
}
```



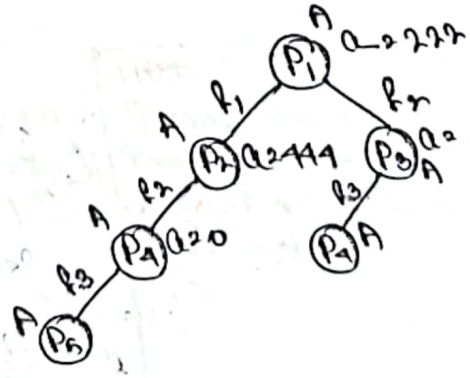
AAAAAAAA

(3)
```
int main() {
    a = fork(); → f1
    if (a == 0){
fork();} → f2
    fork(); → f3
    printf("A\n");
}
```



AAAAAA

(4)
```
int main() {
    fork();
    a = fork();
    if (a == 0) fork();
fork fork();
    printf("A\n");
```

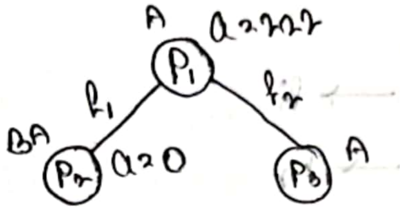


AAAAAA

LECTURE 6

TUESDAY DATE: 13/06/23

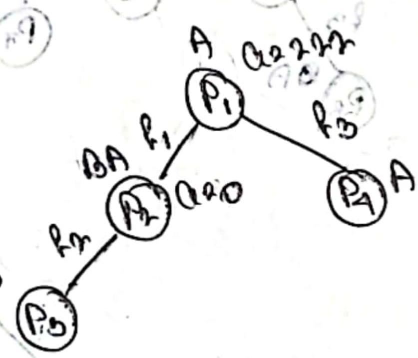
```
int main() {
    a = fork(); → f1
    if (a==0){
        print('B')
    }
    else if (a>0) {
        wait(NULL); → P1
        fork(); → f2
    }
    print('A');
}
```



BAAA
$P_2 P_2 P_1 P_3$

*P_1 wait() খেয়েছে so P_2 শেষ হওয়া পর্যন্ত wait করবে

```
int main() {
    a = fork(); → f1
    if (a==0){
        fork(); → f2
        print('B');
    }
    else if (a>0){
        wait(NULL); → P1
        fork(); → f3
    }
    print('A');
}
```

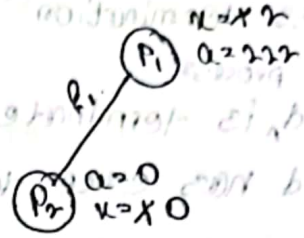


BAABAA
↓ ↓ ↓ ↓
P_2 P_1 P_3 P_4

* এখানে P_3 আগে P_1 আগে process হবে তবে OS নিজেও জানে না. That's why code run দিলে different output আসে,

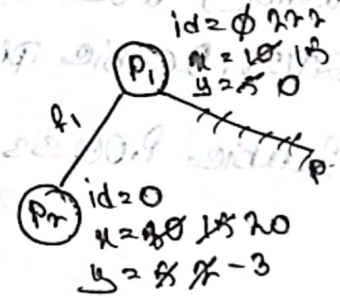
* if (fork()) use করলে child create করবে check করবে। —
0 → child create হয়েছে
-1 → child create হয়নি

```
int main(){
  int x = 1
  a = fork();
  if (a == 0){
    x = x-1;
    printf("value of x: %d", x);
  }
  else if (a > 0){
    wait(NULL);
    x = x+1;
    printf("value of x is: %d", x);
  }
}
```



value of x is 0
value of x is 2

```
int main() {
  int id;
  static int x = 10;
  int y = 5;
  id = fork();
  if (id < 0) {
    printf("fork failed");
  }
  else if (id == 0){
    printf("child started\n");
    x = x+5
    y = y-3
    printf("values of x: %d
      & y: %d\n", x, y);
    printf("child finished\n");
  }
  else{
    wait(NULL);
    printf("parent started\n");
    printf("parent finished\n");
  }
}
```



Output
Parent started
Parent finished
values of x: 15 & y: 0
Terminating

Output:—
child started
value of x: 15 &
y: 2
child finished
values of x: 20 &
y: -3
Terminating

Process termination

A child process is terminated by parent process if—

(1) Child has exceeded the usage of resources.

(2) Task assigned to child is no longer needed.

(3) Parent is exiting (cascading termination)

* wait() use করলে child এর process ~~শেষ~~ শেষ না হওয়া পর্যন্ত parent wait করবে, parent can kill child. child যে resource use করেছে তা ছেড়ে দিবে।

* wait() use না করলে child resource নিয়ে বসে থাকে, zombie process হয়।

* Zombie process end হয় when parent terminates.

Inter Process Communication

Message passing
- ↳ Google hangout, gmail
- ↳ physical distance বেশি
- ↳ very large data send করা যায় না
- ↳ message queue তে message send and receive হয়।

Shared memory
- ↳ Google drive
- ↳ Physical distance কম
- ↳ 2 process এর মাঝে buffer থাকে
- ↳ create → buffer → variable update
- ↳ Data shared betn producer & consumer overlaps [producer – consumer problem]

LECTURE 7

SUNDAY

DATE: 18/10/23

CPU SCHEDULING [LECTURE 3]

P_1, P_2, P_3, P_4, P_5

$P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_5$

$P_5 \rightarrow P_3 \rightarrow P_4 \rightarrow P_1 \rightarrow P_2$

$P_2 \rightarrow P_4 \rightarrow P_1 \rightarrow P_5 \rightarrow P_3$

⋮

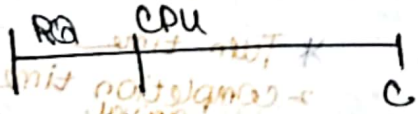
CPU Utilization: CPU ideal থাকবে না, কাজ করতে থাকবে
CPU Utilization ↑

Throughput: Process executed per unit of time.
Throughput ↑

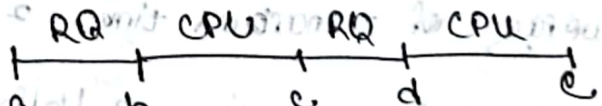
Waiting time: CPU তে আসবার আগে RAM এ অপেক্ষা করে
Waiting time ↓

Turnaround time: একটা process কে সম্পূর্ণ CPU তে কাজ করে, তার
~~wait time~~ to till termination পর্যন্ত time কে বলে
Turnaround time ↓

Response time: RAM থেকে CPU, to first time পর্যন্ত time. [first waiting time]
Response time ↓



Waiting time — (b-a)
turnaround time — (c-a)
Response time — (b-a)



Waiting time — (b-a) + (d-c)
turnaround time — (e-a)
Response time — (b-a)

2 Types of processes:

preemptive process: process is partially executed
eg. Greedy
has context switch.

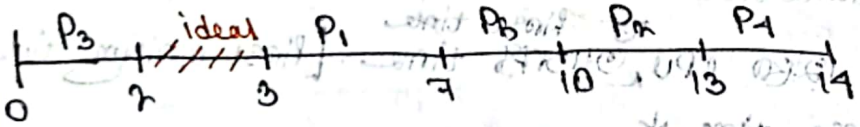
non-preemptive process: process is cannot be partially executed.
eg. Knapsack

* which process is better depends on scenario.

FCFS → First Come, First Serve

	Arrival Time	Burst Time [Time taken to execute]	Turn around Time	wait Time
P_1	3	4	4	0
P_2	6	3	8	5
P_3	0	2	2	0
P_4	6	1	9	8
P_5	4	3	6	3
		13	29	16

Example of non-preemptive process [Arrival time consider zero]



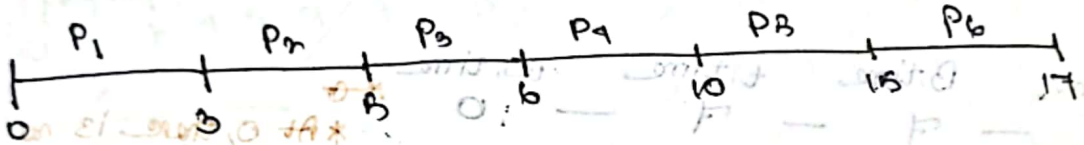
Total time = Σ Burst time + Ideal Time

average of turnaround time = 29/5 = 5.8

average of wait time = 16/5 = 3.2

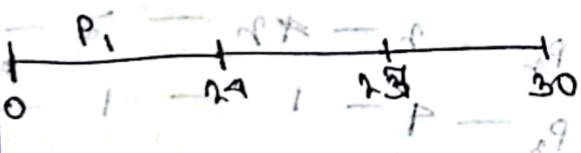
* Turn time = completion time − arrival time
* Average of turnaround time & waiting time
* Wait Time = start time − arrival time

	A time	B time	ti time	w.time
P_1	0	3	3	0
P_2	1	2	4	2
P_3	2	1	4	3
P_4	3	4	7	3
P_5	4	5	11	6
P_6	5	2	13	10

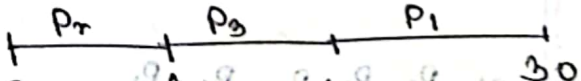


* Weakness of this algorithm: No partial execution is done.

	b.time	A.time	w.time
P_1	24	0	0
P_2	3	~~0~~	~~24~~
P_3	3	0	27
			17



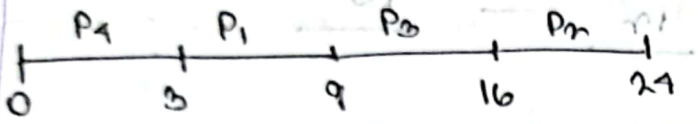
	b.time	A.time	w.time
P_1	24	3	6
P_2	3	1	0
P_3	3	2	3
			17/3



Convoy effect: ~~If~~ a big process ~~is comes first~~ and many more small processes ~~are after it~~, come together ~~it is called~~ then average waiting time increases causing convoy effect.

SJF

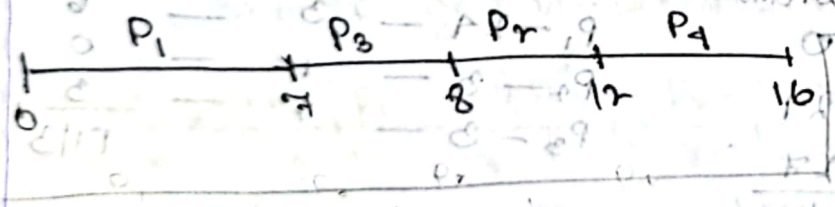
	A.time	B.time	t.time	w.time
P_1	0	6	9	3
P_2	0	8	24	16
P_3	0	7	16	9
P_4	0	3	3	0

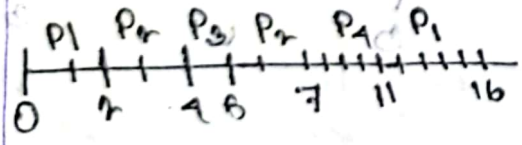


* SJF — min average waiting time হয়
* এখানে B.time কম, তাকে আগে নিব

	A.time	B.time	t.time	w.time
P_1	0	7	7	0
P_2	2	4	10	6
P_3	4	1	4	3
P_4	5	4	11	7

* At 0, there is no process in RQ. So, the process at 0 is the first one to be in CPU. irrespective of their burst time





~~P1 P2 P3 P4~~

SRTF = Preemptive SJF

* Arrival time + Burst time কম

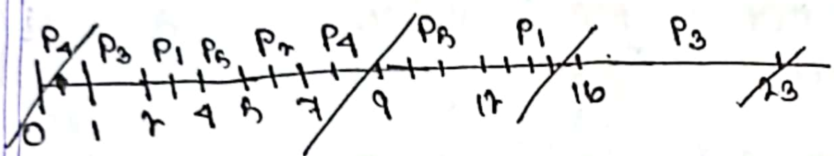
SRTF shortest

	A.time	B.time	t.time	w.time
P_1	0	7	16	9
P_2	2	4	7	1
P_3	4	1	5	0
P_4	5	4	11	2

Example of preemptive process

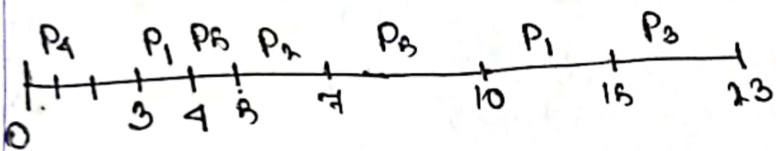
	A.time	B.time	t.time	w.time
P_1	2	~~5~~ 3	13	1 + (10 − 4) = 7
P_2	3	~~2~~ 2	2	0
P_3	1	8	22	14
P_4	0	~~5 2 0~~ 3	3	0
P_5	4	~~4~~ 3	6	0 + (7 − 5) = 2

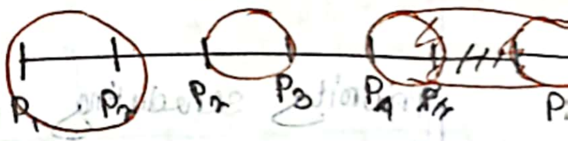
* SRTF + priority
FCFS — Non-P
SJF → Non-P
SRTF — P



~~P4 P3 P1 P5 P2~~

* process এর queue এ রাখো
* when all process are in queue, we stop preemptive way of processing & use only B.time to proceed further.





~~P4~~ P3 P1 P5 ~~P2~~

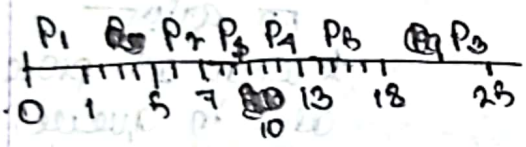
Number of context switch: 3

Context switch criteria
- ~~CPU থেকে বের হলে হবে~~
- CPU থেকে একটা process আসলে হবে and আরেকটা process বের হলে হবে।

LECTURE 8 | TUESDAY | DATE: 20/06/23

SRTF

	A.time	B.time	t.time	w.time
P_1	0	8	10	0
P_2	5	2	7	0
P_3	1	3	14	11
P_4	6	3	7	4
P_5	8	5	10	5

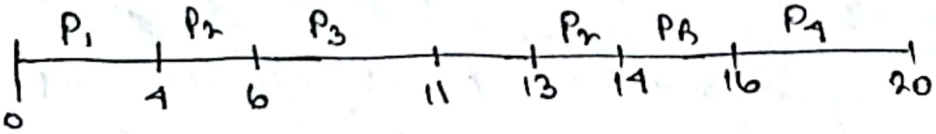


$P_1 P_3 P_2 P_4 P_5$

Priority scheduling

[Lower the number higher the priority scheduling is]

	Priority	A.time	B.time	t.time	w.time
P_1	1	0	4	4	0
P_2	2	0	1	14	4+7
P_3	1	6	7	7	0
P_4	3	11	4	9	5
P_5	2	12	2	4	2



$P_1 P_2 P_3 P_4 P_5$

* priority
* first 2 priority high.
* same priority 2 2[nd] arrival time consider 2nd.
* same arrival time 2nd burst time consider 2nd.

ROUND ROBIN

↳ always preemptive

↳ একটি process এর burst time অনেক বেশি হলে, তাকে forcefully context switch করায়

q = 3 → 3 sec পর forcefully context switch হয়

↳ quantum no. [CPU তে process max time পায়]

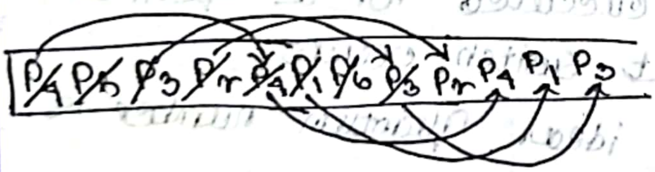
	A.time	B.time	t.time	w.time
P_1	3	~~0~~ ~~5~~ 2	27	10 + 12
P_2	4	~~6~~ 3	23	5 + 12
P_3	3	~~7~~ ~~4~~ 1	30	3 + 12 + 8
P_4	1	~~9~~ ~~6~~ 3	29	0 + 8 + 12
P_5	2	2	4	2
P_6	6	3	18	12

* যার arrival time যেই আগে ready queue তে যায়, max Q sec CPU তে পায়, then তার queue তে আসলে queue এর end এ চলে যায়

* Q sec > burst time হলে q sec পর্যন্ত process CPU পায়

* Q sec < burst time process terminates

idle	P_4	P_5	P_3	P_2	P_4	P_1	P_6	P_3	P_2	P_4	P_1	P_3
0–1	1–4	4–6	6–9	9–12	12–15	15–18	18–21	21–24	24–27	27–30	30–32	32–33



context switch: 11

[Online] LECTURE 9

SUNDAY DATE: 25/06/23

Priority scheduling

Starvation – কোন process এর low priority এর জন্য execution এর chance পাচ্ছে না

Effect of starvation:

waiting time ↑

average waiting time ↑

total time ↑

Solution:

Ageing: process এর priority বাড়তে থাকে, process এর priority number 1 time decrease হয়ে to execute that process. Priority number will decrease when the threshold time is crossed.

(dynamic method)

eg. 105 এর মধ্যে execute না হলে priority decrease হয়ে, তাও না হলে আবার decrease হয়ে।

Round Robin (RR)

* In RR, processes are executed for a specific quantum time and then context switch occurs.
* We don't know what's an ideal quantum number is.
* We do context switching so that other processes have the chance for execution.
* Context switch will be costly if frequent context switching is done as resources are loaded & unloaded.
* Lowest turnaround time gives the optimum quantum time.
* 80% of CPU bursts should be shorter than quantum.

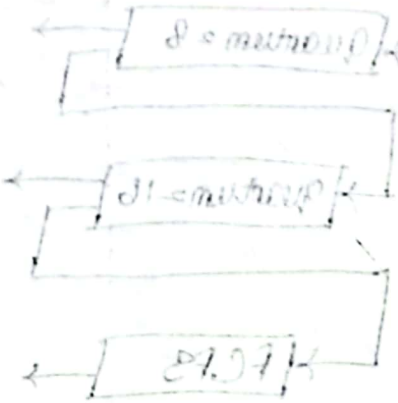
Multilevel Queue : combination of different scheduling used for different processes

2 Types of processes :-

- foreground (interactive) processes — user এর সাথে সরাসরি interactive
  - eg gaming
  - maximum amount of responsive হতে হবে
  - Round Robin Algorithm

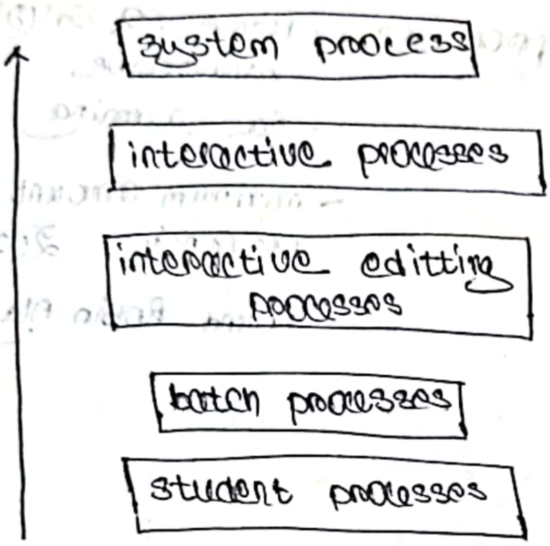
- background (batch) process — user এর সাথে interaction নাই, background এ চলবে
  - eg file download
  - FCFS algorithm

* Priority of foreground > background, so 80% of the foreground process will be executed in CPU and 20% of the background process will be executed. → RR



Exam: Different scenario is given and multiple scheduling is implemented, best scheduling is asked to find and short theory questions related to the scenario.

Multilevel Queue Scheduling



The higher the process, the higher is their priority

* priority 1 দিয়ে শুরু করা যায় না, OS PC limitation থাকে.

* upper the processes, higher the priority. so RR or priority scheduling should be used.

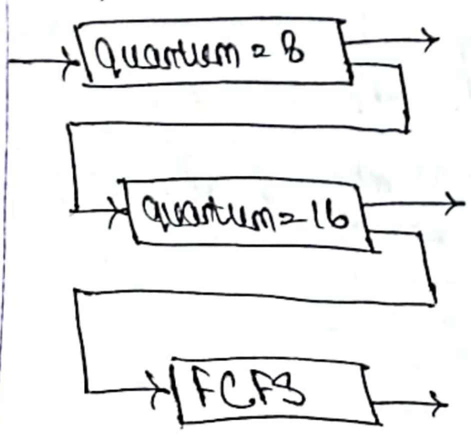
* for lower processes, we can be flexible. or use FCFS

Multilevel Feedback Queue Scheduling

* একটা process কে different time এ different queue দিয়ে different CPU scheduling দিয়ে execute করা যায়।

* একটা process এর execution time/burst time বেশি হলে, then তার priority কমিয়ে দিব (priority number বাড়িয়ে দিব) → starvation avoid করা যায়

Example:



* একটা process RR দিয়ে quantum=8 এবং quantum=16 এ execute হয়নি, তার মানে ওর burst time > 24; so, তাকে FCFS দিয়ে সোজা execute করাই দিব।
This is done by lowering the priority. ~~& incr~~

THREADS [LECTURE 4]

* CPU তে multiple core থাকলে, process parallely execute করা যায়, এটা Thread কিন্তু করা যায়।

* একটি process অনেক বড় হলে, যেমন, multiple thread use করে parallely execute করে কমাতে পারি and executing time কমে যায়।

* Child process v/s thread? Child process এ parent process এর duplicate করে and thus resources are wasted, whereas thread এ resources share হয় and has individual data [thread ID, Program counter, register, stack] and thus resource wastage is reduced. → in terms of space difference.

child process আগে execute হয় and then parent process execute হয় if wait() is given, whereas in thread multiple thread can execute parallely. → in terms of execution time

* একটি process এ multiple thread এ convert করতে পারি, but একটি thread এ multiple process এ convert করা যায় না।

Benefits:-

(1) Responsiveness: If multiple threads are opened in a browser and any one of them is blocked / not working, then the other threads are not affected.

(2) Resource sharing: Resources are shared between threads, rather than being duplicated.

(3) Economy: If we can execute parallelly, PC responses much more faster.

Example:

Concurrent execution on single-core system:

single core	t_1	t_2	t_3	t_4	t_1	t_2	t_3	t_4	...

* process ও task একই কথা

Parallelism on multi-core system:

core 1	t_1	t_3	t_1	t_3	t_1	...
core 2	t_2	t_4	t_2	t_4	t_2	...

* A program can select whether process will be single or multi threaded.

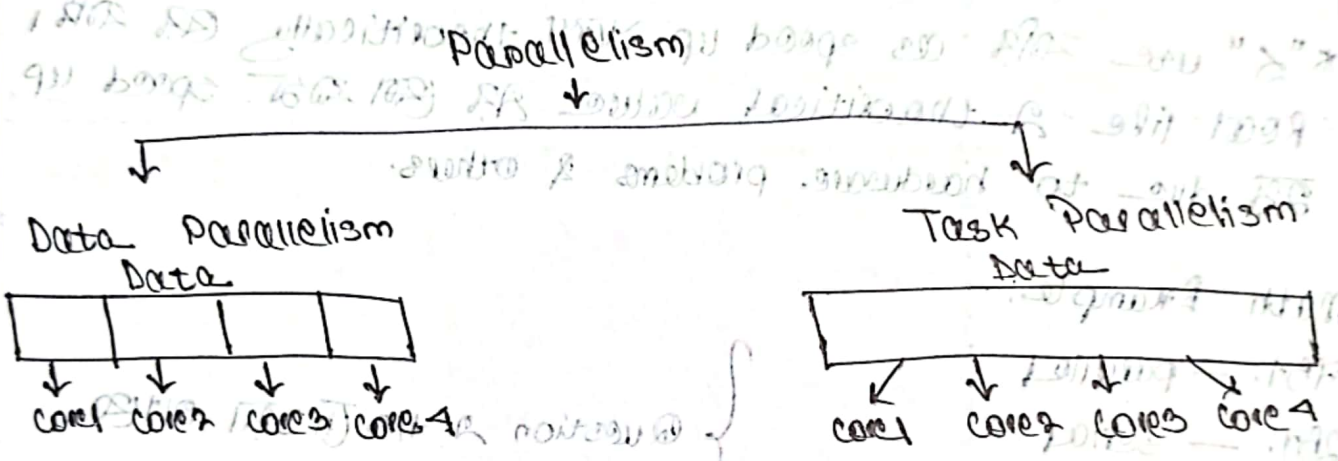
* For multi-core নিলে আমরা same task কে half time এ complete করতে পারি।

(4) Scalability: A single threaded process can only run on one CPU, no matter how many may be available, whereas execution of multi-threaded process may be split into different processors.

Multi-core Programming

* Multi-core programming is done by programmers & they need to focus on:-

(1) Dividing activities
(2) Balance
(3) Data splitting
(4) Data dependency
(5) Testing & debugging



→ One data is divided.
→ The divided datas are assigned to different CPU.
→ 4 CPU = 1/4 execution time
→ একটা বড় data কে একা করতে পারি।
→ Different CPU does same operation

→ A data is assigned to different CPU.
→ add, sub, mul এই test করতে পারি।
core1 শুধু add, core2 শুধু sub } same data কে করি
→ Different CPU does unique operations.

Exam: Scenario দেখা হবে, we have to choose which parallelism to use.
Example of Data & Task Parallelism.

Amdahl's Law

* সব code কে parallelism এ execute করা যায় না,
~~e.g. variable কে parallel~~

* code এ যা parallelism এ করা যায়, তাকে parallel এ করা যায় and বাকি গুলো serial এ করি।

$$\text{speed} \leq \frac{1}{S + \left(\frac{1-S}{N}\right)}$$

S → serial
N → No. of core

* "≤" use করি as speed up গুলো theoritically বের করি। Real life এ theoritical value এর চেয়ে কম speed up হয় due to hardware problems & others.

Math Example:-

75% — parallel
25% — serial
moving from 1 to 2 core

} Question এ এই দেওয়া থাকে।

[1 core থেকে 2 core নিয়ে গেছি 180 upgrade করেছি]

N = 1

$$\text{speed up} = \frac{1}{S + \frac{1-S}{N}} = \frac{1}{0.25 + \frac{1-0.25}{1}} = 1$$

N = 2

$$\text{speed up} = \frac{1}{S + \frac{1-S}{N}} = \frac{1}{0.25 + \frac{1-0.25}{2}} = 1.6$$

[বি] Exam: math আসবে.

LECTURE 10

TUESDAY DATE: 04/07/23

Parallel → 80%.

Core → 9 – 8

$n_1 = 9$, speed $= \dfrac{1}{(1-0.8) + \frac{0.8}{9}} \Rightarrow 3.46$

$n_2 = 8$, speed $= \dfrac{1}{(1-0.8) + \frac{0.8}{8}} \Rightarrow 3.33$

* কত times বাড়ে/কমে সেটা বের করা যায় either by finding speed of n_2 – speed of n_1, or $\dfrac{\text{speed of } n_2}{\text{speed of } n_1}$

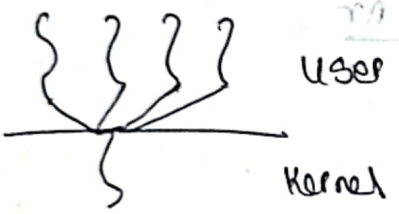
Thread are of 2 types:

User Thread → programmer create करते.

Kernel Thread → OS द्वारा thread ~~create~~ allocate केले जाते parallelism हे kernel level वर होते

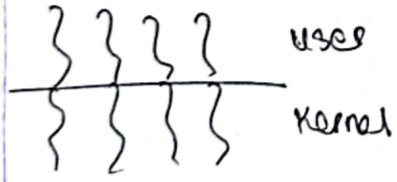
* प्रत्येक user thread ला kernel thread चा असतो.

① Many to One model



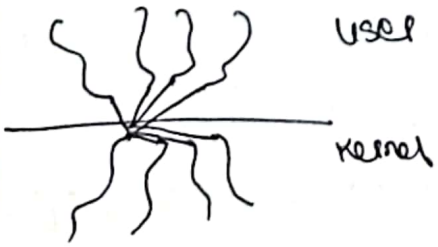
* parallelism cannot be achieved
* If one of the thread in user is blocked by the thread in kernel, then the remaining 3 threads in user are also blocked.

② One to One model



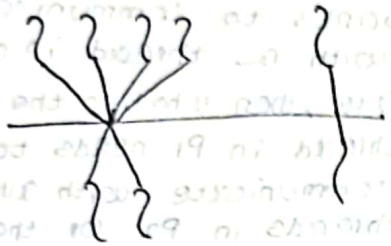
* The problems in many to one model is resolved here.
* Enough amount of kernel thread cannot be provided.

③ many to many model



* The problem of many to one is resolved.
* Multi threads in user are connected so there is a chance of queing. Thus, the model slows down.

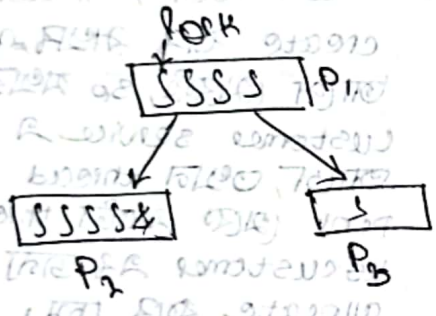
④ Combination of many to many + one to one [Two Level model]



* One to one এর priority বেশি

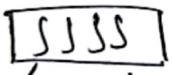
Threading Issues

(1) fork() and exec() system calls:



* fork() call করলে যদি exec() call থাকে তাহলে P_3 create হবে না else P_2 duplicate হবে P_1 তে create হবে

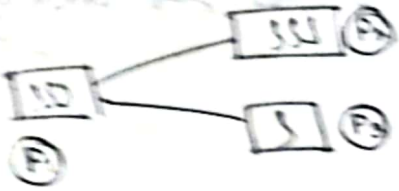
(2) Thread cancellation



Asynchronous
[cancel call থাকলে thread টি cancel হয়ে যাবে at once]

Deferred
[cancel call থাকলে test complete করে, thread cancel হতে পারে]

(3) Signal Handling :



* If a signal thread from one core ~~wants~~ P1 wants to communicate with a thread in other core, then whether the thread in P1 needs to communicate with all threads in P2 or the specific thread in P3 depends on scenario.

(4) Thread Pool

* Thread Pool → Thread create করে রাখা, যেন আগে থেকেই, so যখন customer service এ আসে, তখন thread pool থেকে একটা thread ~~to~~ customer এর জন্য allocate করে দেয়। এই thread এর কাজ শেষ হলে আবার queue তে return করে।

(5) Thread Specific Data

* Thread can share ~~the~~ data.
* The data which is private to specific thread, it is thread specific data.

① Intro + Structure — Theory
[Short Question → Reason, Scenario Based]

② Process — Theory + Tracing
[Tracing — mandatory, short Question (2|3)]

③ Thread — Theory + math
[Short Question, Amdahl's Law (2)]

④ CPU scheduling — Algorithm + Theory
[Algorithm — 2 Different Algo on, 2 Different Algo on same scenario + compare, Theory] (1|2)

Total : 25 → Algorithm + math + Theory → 80%.
→ Theory → 20%.