# Lecture 1

## Operating System

- Software abstracting hardware
- Interface between user and hardware
- set of utilities to simplify application development / execution.

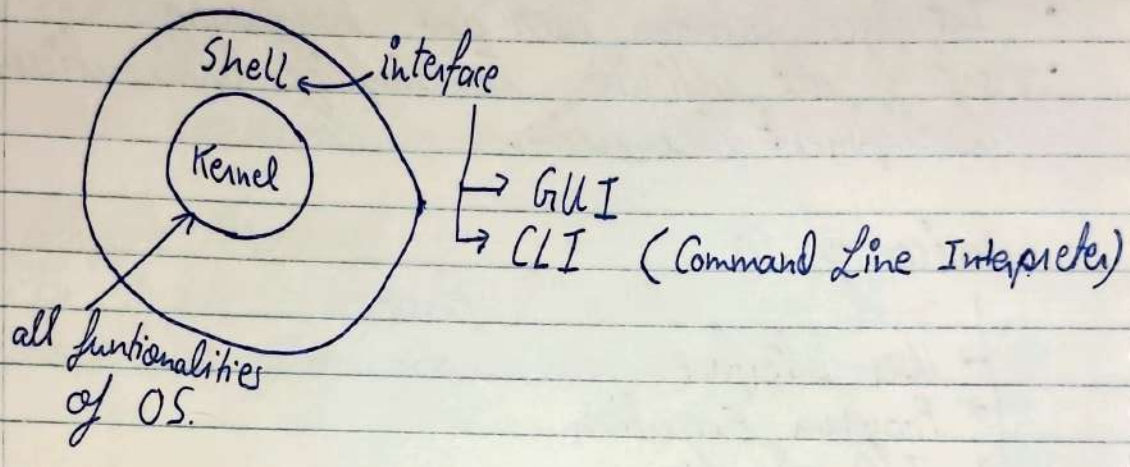
### Services of OS

- User Interface
- Program Execution
- I/O operation
- File-System Manipulation
- Communication (Inter-process communication)
- Error detection
- Resource allocation
- Accounting
- Protection & Security.

### Goals of OS

- Convenience (User-friendly)
- Efficiency
- Portability
- Reliability
- Scalability
- Robustness

## Parts of OS




## System calls

A way for programs to interact with OS.

## Dual mode of operation

- User mode (mode bit = 1)
- Kernel/System/Supervisor Mode (mode bit = 0)

We use dual mode for protection in OS.

# Lecture 2 (OS)

Types of OS

- Uni-programming OS
- Multiprogramming OS
- Multi Tasking OS
- Multi User OS
- Multi Processing OS
- Embedded OS
- Real-Time OS
- Hand-held device OS

## Uni-programming OS

OS only allows one process to reside in main memory beside itself.

Since single process cannot keep I/O and CPU engaged at the same time, this can lead to time where CPU is not utilized at all.

→ So not good CPU utilization.

## Multi-programming OS

OS allows multiple processes to reside in main memory.

→ Better CPU utilization than uni-programming
→ Degree of multiprogramming = no. of running programs in memory (except the OS)

→ As the degree of multiprog. increases, CPU utilization also increases upto a certain limit.

Multiprogramming OS has 2 types of process scheduling
- Preemptive scheduling
- Non-Preemptive scheduling

~~Preemptive ⇒~~

Non-Preemptive ⇒ ~~A process runs until~~
- ~~Process ter~~

Non-Preemptive ⇒ A process runs on CPU until
- Either process terminates or goes for I/O operation

The process cannot be taken out of CPU by OS.

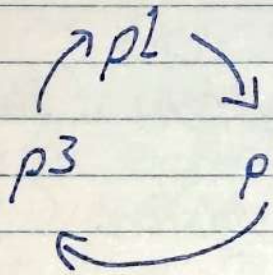
Preemptive ⇒ Process can be forcefully taken out of ~~system~~ CPU.

Uniprogramming OS cannot be preemptive.

## Multi-tasking OS

Extension of multiprogramming OS, processes execute in round-robin fashion.

Processes are run ~~in a router~~ for a limited time on CPU before they are forcibly taken out and replaced with another process.
So if we have three processes P1, P2 and P3




This is called round-robin method. Each process is given limited time on the CPU.

Since CPU is very fast ~~user~~ to user it ~~also~~ appears that process are in parallel.

This is called time sharing OS as well.

## Multi-User OS

This OS allows multiple users to access single system simultaneously.

Eg → Linux / BSD.

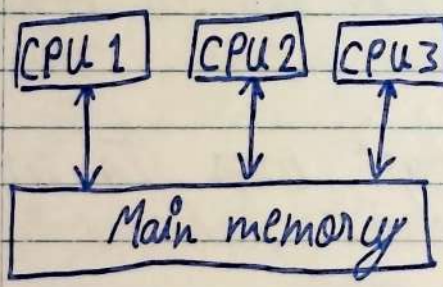
## Multi-Processing OS

An OS used in systems with multiple CPUs.

Types

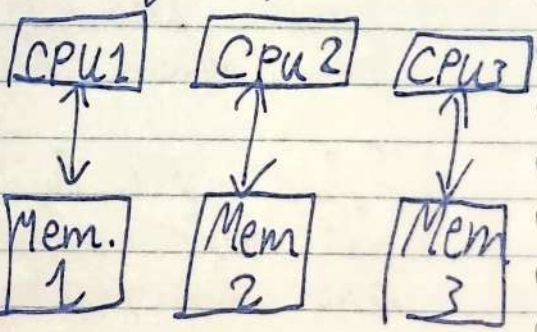
- Tightly coupled system (Shared memory system)
- Loosely coupled system (Distributed system).

Tightly coupled




All CPUs use same memory

Loosely coupled.




Every CPU has seperate memory.

## Embedded OS

An OS for embedded systems.

→ Designed for a specific purpose, to increase functionality and reliability for achieving a specific task, in embedded system (Eg, smart fridge).

→ User interaction is minimal with OS.

## Real-time OS

Real time OS (RTOS) are used in environments where a large number of events, mostly external to the system, must be accepted and processed in a short time or within certain deadlines.

Eg, OS used for adjusting path of a rocket.

→ Every process in RTOS has a deadline.

Types

- Hard RTOS (strict about deadline)
- Soft RTOS (some relaxation in deadline).

## Handheld device OS

Used in handheld devices (Eg, android and iOS).

# Lecture 3

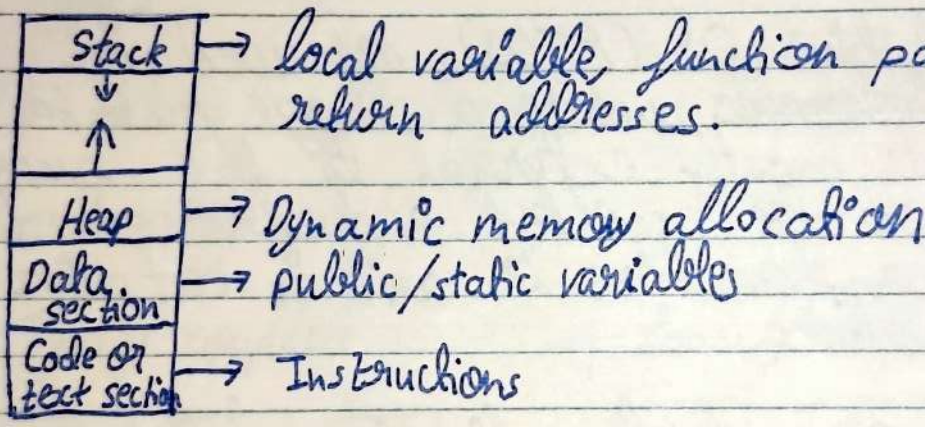
## Process

A program under execution is a process.

program + runtime activity = process

In terms of CPU, a process is a schedulable/dispatchable unit.
For OS, it a locus of control.

## Representation of a process




Every process has similar structure in main memory.

## Operations on a process

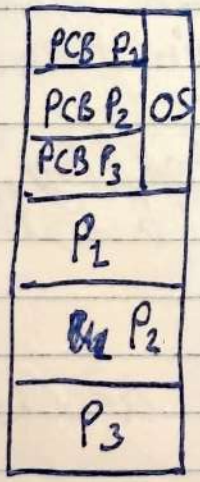
- Create (Resource allocation)
- Schedule, Run
- Wait / Block
- Suspend / Resume
- Terminate (Resource deallocation).

## Attributes of a Process

- PID (Process ID)
- PC (Program Counter)
- GPR (General Purpose Registers)
- List of devices
- Type
- Size
- Memory limits
- Priority
- State
- List of files

All attributes are stored in a structure called PCB (Process Control Block). Also known as process descriptor.

PCBs are kept in OS process structure in memory. That is, PCB is stored in the memory taken by OS process in main memory.




Main Memory

→ The GPR attribute and PC attributes

store GPR and PC value when process is suspended / blocked.

The value of PCB attributes is collectively called context of the process.

When switching processes, we simply switch the context of the processes. This operation is called context switch.

Context switch
- context save → Save the context from CPU to PCB in main memory.
- context load → load cpu registers and other flags/ attributes from PCB stored in main memory.

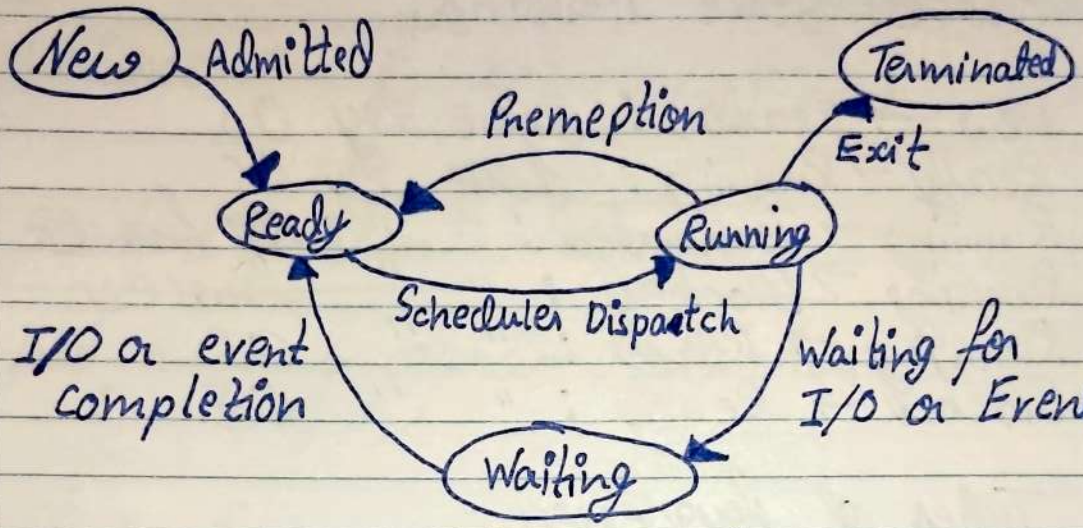
So context switching is stopping a process and running another process.

Note : other processes cannot access the PCBs.

⇒ Context switching is done by dispatcher in OS.

# Lecture - 4

## Process States




Waiting is sometimes also called Blocked state.

⇒ Resource allocation happens when process is admitted and deallocation happen when process exits.

⇒ Process can only transition from running to exit and running to waiting. All other transitions are controlled by OS.

States

- New : All installed processes are known to be in new state
- Ready : All process waiting to be run on CPU are in ready state
- Running : A process running on CPU
- Terminated : A complete process has its state as terminated

↳ Blocked/Waiting : Processes waiting for I/O or event.

Process State Transitions

New → Ready : done by OS
Ready → Running : done by OS
Running → Terminated : " " process
Running → Blocked : " " process
Running → Ready : " " OS
Blocked → Ready : " " OS.

Types of processes

- CPU Bound
- I/O Bound

CPU Bound : If the process is intensive in terms of CPU operations.
I/O Bound : If the process in intensive in terms of I/O operations.

A process which is terminated but has not relinquished its resources is called zombie process.

Main Memory

- Kernel space ⇒ Space for core of the OS, can't be accessed by users.
- User space ⇒ Space for user processes.

# Lecture - 5

## Process Scheduling

We need scheduling for better resource utilization

### Scheduling Queues

- Job Queue
- Ready Queue
- Device Queue

⇒ Job Queue is in main memory in the OS process. Ready & Device queue are also in the OS process.
⇒ All processes in new state are kept in job queue.
⇒ All processes in the ready state are kept in ready queue.
⇒ Queue in which process waits for I/O device. Every I/O device has its own queue.

⇒ Ques store PCBs of process.

### Types of Schedulers

- Long term scheduling
- Short term scheduler
- Mid term Scheduler.

⇒ Long term scheduler transitions a process from new state to ready state. (or in other words moves from job queue to ready queue). Long term scheduler is initiated by either (i) user or (ii) OS.

It also does resource allocation.

⇒ Short term scheduler selects one of all the ready processes to run.

⇒ If there is not enough memory remaining, the mid term scheduler will move process from main memory to secondary memory (swap space). This ~~process is~~ operation is called swap out.

When user wants to again use the process we do a swap in.

The process of swap in and swap out is calling swapping and is done by mid term scheduler.

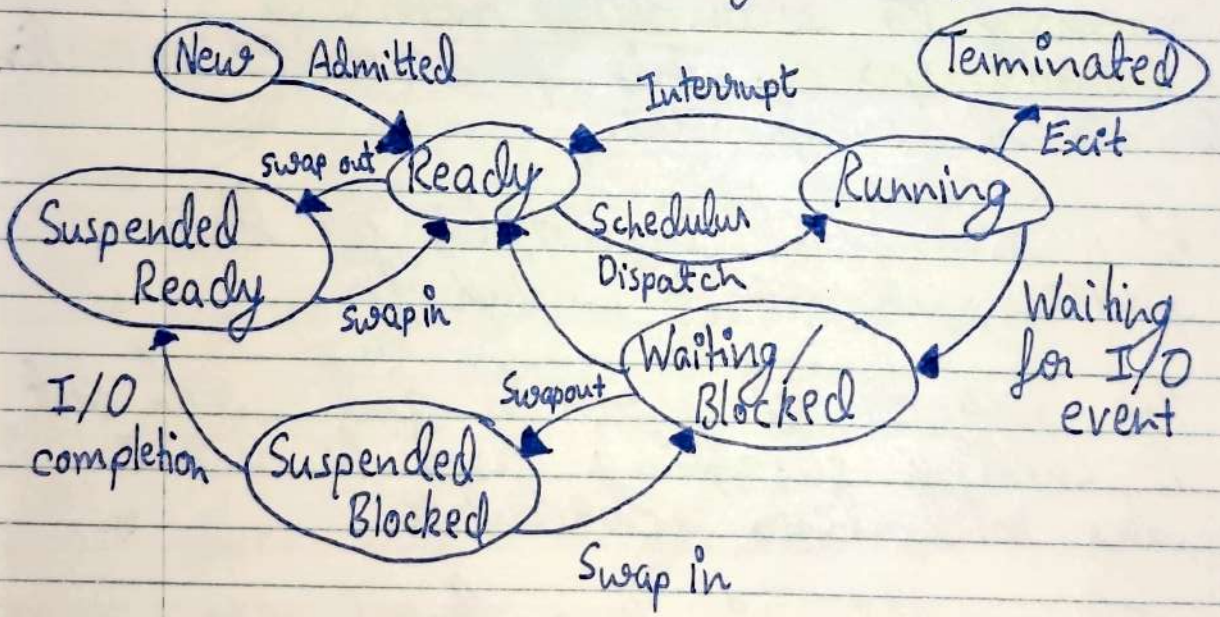
If swapping is done based on priority, it is called rolling. So swap in/out is also called roll in/out.

When we swap out a process, it is called suspended process.

A process can only be suspended from waiting or ready state.

* When suspended from waiting/blocked state, the state is called suspended blocked.
* When suspended from ready state, the state is called suspended ready.

So complete state diagram is,



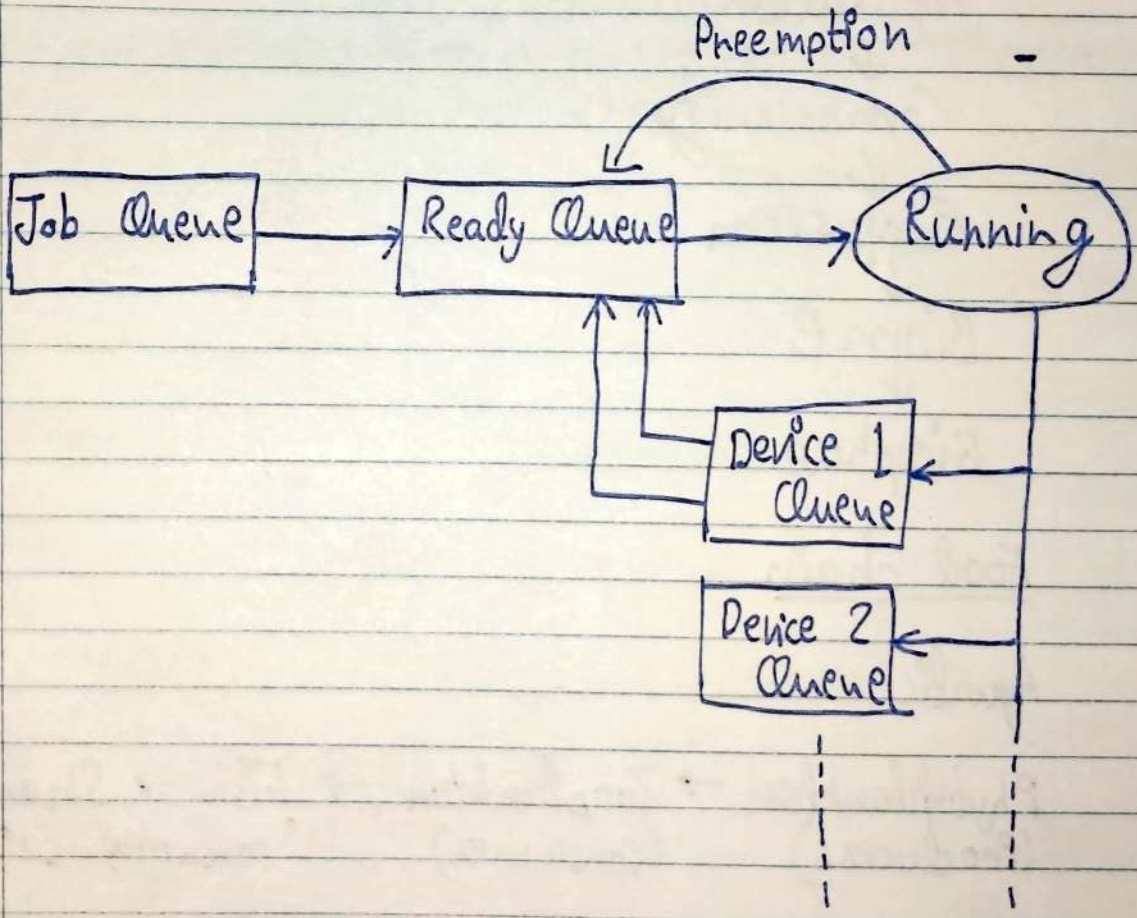

⇒ Long term scheduler controls max degree of multiprogramming and can ~~decrease~~ increase it.
⇒ Mid term scheduler can decrease degree of multiprogramming

# CPU Scheduling

Done by short term scheduler.
Make a selection for one of the processes from ready state to run on CPU.

Goals

- Minimize Wait time and Turn around time
- Maximize CPU utilization (Throughput)
- Fairness in selecting processes.



# Lecture-6

## Scheduling Times (Short term Scheduler)

- Arrival Time (AT)
- Burst Time (BT)
- Completion Time (CT)
- Turnaround Time (TAT)
- Waiting Time (WT)

AT ⇒ Time at which process arrives in the system.

BT ⇒ The amount of time for which process runs on CPU.

CT ⇒ The time at which process completes the execution

TAT ⇒ Amount of time from arrival to completion.

$TAT = CT - AT$

WT ⇒ Total time spent waiting in the ready queue

$WT = TAT - BT$

## Other scheduling Times

- Response Time (RT)
- ~~Deadline~~ Deadline (D) (Not in syllabus)
- Scheduling Length (L)
- Throughput.

RT ⇒ Amount of time from arrival till first time process gets CPU.

Scheduling Length ⇒ Time taken for scheduling the process.

$\mathcal{L} = \max(CT) - \min(AT)$

Throughput ⇒ Number of processes executed per unit of time.

$\text{Throughput} = \frac{\text{no. of processes executed}}{\mathcal{L}}$

CPU Scheduling
- Preemptive
- Non-Preemptive

Note :- For all the algorithms in syllabus, we assume no I/O operation.

## First Come First Serve

Criteria ⇒ Arrival Time (AT)
- Tie-breaker : smaller process id first.

Type ⇒ Non-preemptive.

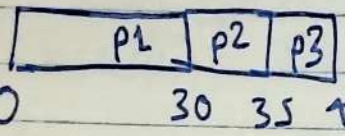
We use a timeline chart, that starts at zero, to study scheduling algorithms. This is called Gantt Chart.

Eg, for FCFS

Given in Ques | To be calculated

| Process | Arrival Time | Burst Time | CT | CT-AT / TAT | TAT-BT / WT |
|---|---|---|---|---|---|
| p1 | 0 | 30 | 30 | 30 | 0 |
| p2 | 0 | 5 | 35 | 35 | 30 |
| p3 | 0 | 5 | 40 | 45 | 35 |

i) Drawing Gantt chart




ii) Use Gantt chart to get completion time

iii) Use formulae for TAT & WT

iv) Get avg TAT and avg WT

v) In non-preemptive algorithms, response time of process = waiting time of process

vi) Scheduling Length

$L = \max(CT) - \min(AT)$

$= 40 - 0$

$= 40$

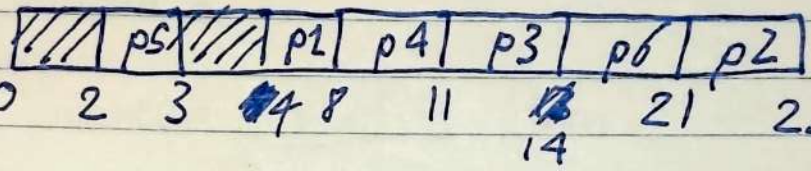
vii) Throughput $= \frac{n}{L} = \frac{3}{40}$

Note: If we have multiple processes in FCFS and to choose from in FCFS, we always go for smallest arrival time, only if the smallest arrival time is of 2 processes, we compare PID.

## Another Example, using FCFS

| Process | Arrival time | BT | CT | TAT | WT |
|---|---|---|---|---|---|
| p1 | 4 | 4 | 8 | 4 | 0 |
| p2 | 8 | 2 | 23 | 15 | 13 |
| p3 | 6 | 3 | 14 | 8 | 5 |
| p4 | 5 | 3 | 11 | 3 | 3 |
| p5 | 2 | 1 | 3 | 1 | 0 |
| p6 | 7 | 7 | 21 | 14 | 7 |

## Making Gantt Chart




CT using Gantt Chart
TAT = CT - AT
WT = TAT - BT

## Convoy Effect

In a non-preemptive system, scheduling a process with big burst time will cause waiting times of all processes to increase. The next algorithms try to avoid this effect.

# Lecture - 7

## Shortest Job First

Criteria : Smallest Burst Time process first
- Tie breaker : FCFS

Type : Non-preemptive and preemptive

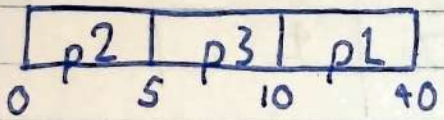
### Shortest Job First (Non-preemptive)

Given in Ques | To be calculated

Eg.

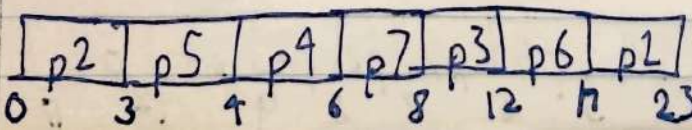
| Process | AT | BT | CT | TAT | WT |
|---|---|---|---|---|---|
| p1 | 0 | 30 | 40 | 40 | 10 |
| p2 | 0 | 5 | 5 | 5 | 0 |
| p3 | 0 | 5 | 10 | 10 | 5 |

Gantt Chart




Another Eg,

| Process | AT | BT | CT | TAT | WT |
|---|---|---|---|---|---|
| p1 | 0 | 6 | 23 | 23 | 17 |
| p2 | 0 | 3 | 3 | 3 | 0 |
| p3 | 1 | 4 | 12 | 11 | 7 |
| p4 | 2 | 2 | 6 | 4 | 2 |
| p5 | 3 | 1 | 4 | 1 | 0 |
| p6 | 4 | 5 | 17 | 13 | 8 |
| p7 | 6 | 2 | 8 | 2 | 0 |



## Shortest Remaining Time First

This is sometimes also refered to as preemptive SJF.

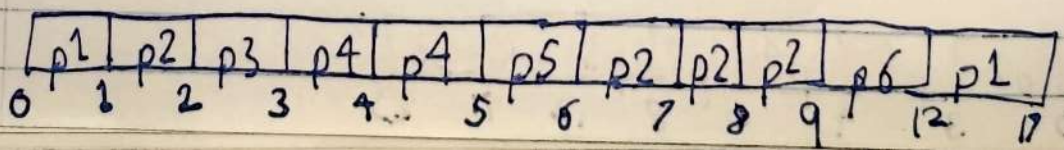
We compare the remaining burst time of current process and compare it with BT of arriving processes.

The smaller one will then run next. Hence this is preemptive.

| Process | AT | BT | CT | CT-AT= TAT | TAT-BT= WT |
|---|---|---|---|---|---|
| p1 | 0 | 6 | 17 | 17 | 11 |
| p2 | 1 | 4 | 9 | 8 | 4 |
| p3 | 2 | 1 | 3 | 1 | 0 |
| p4 | 3 | 2 | 5 | 2 | 0 |
| p5 | 4 | 1 | 6 | 2 | 1 |
| p6 | 5 | 3 | 12 | 7 | 4 |

It is recommended to have rough work for this algorithm, which updates BT every clock as written below,

p1 - 6 - 5 - 0 X
p2 - 4 - 3 - 2 - 1 - 0 X
p3 - 1 - 0 X
p4 - 2 - 1 - 0 X
p5 - 1 - 0 X
p6 - 3 - 0 X



Now, we have a rough gantt chart, It can be easily changed to formal gantt chart.

Also in non-preemptive algorithms we have new ~~kind~~ line of measurment called response time.

Response Time (RT) = Time at which process first gets the CPU − Arrival Time

So for the table in preemptive algos, example.

| process | AT | BT | CT | TAT | WT | RT |
|---|---|---|---|---|---|---|
| p1 | 0 | 6 | 17 | 17 | 11 | 0 |
| p2 | 1 | 4 | 9 | 8 | 4 | 0 |
| p3 | 2 | 1 | 3 | 1 | 0 | 0 |
| p4 | 3 | 2 | 5 | 2 | 0 | 1 |
| p5 | 4 | 1 | 6 | 2 | 1 | 1 |
| p6 | 5 | 3 | 12 | 7 | 4 | 4 |

Note: The formal Gantt chart ~~has~~ does not have same process in adjacent blocks

In ~~case~~ Gantt chat:

No of context switches = No. of process blocks − 1

# Lecture-8

Problems with SJF and SRTF

- Starvation (indefinite waiting of a process)
- No fairness
- Practical implementation not possible.

## Highest Response Ratio Next

The objective is to not only favour short jobs ~~best~~ but decrease WT of longer jobs.

- Criteria : ~~Shortest~~ Highest Response Ratio
  - Tie-breaker : Burst Time
- Type : Non-preemptive

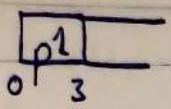
Response Ratio (RR) = $\frac{W+S}{S}$

W = Wait Time till that time
S = Service/Burst Time

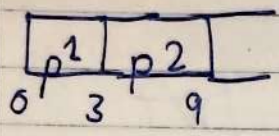
Example,

| Process | Arrival Time | BT |
|---|---|---|
| $p^2$ | 0 | 3 |
| p2 | 2 | 6 |
| p3 | 4 | 4 |
| p4 | 6 | 5 |
| pS | 8 | 2 |

At time 0,




At time 3,

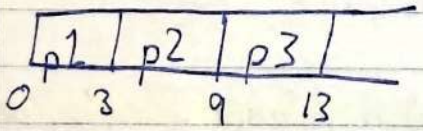



At time 9,
We have p3, p4, p5

$RR(p3) = \frac{W+S}{S} = \frac{5+4}{4} = 2.25$

$RR(p4) = \frac{W+S}{S} = \frac{3+5}{5} = 1.3$

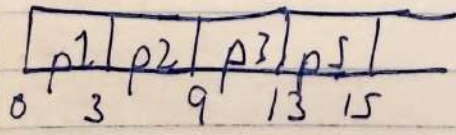
$RR(p5) = \frac{W+S}{S} = \frac{1+2}{2} = 1.5$



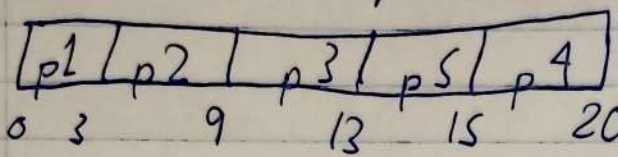

At time 13,
We have p4, p5

$RR(P4) = \frac{7+5}{5} = 2.4$

$RR(P5) = \frac{5+2}{2} = 3.5$



p4 is the last process,




So the table

| Process | AT | BT | CT | TAT | WT |
|---|---|---|---|---|---|
| p1 | 0 | 3 | 3 | 3 | 0 |
| p2 | 2 | 6 | 9 | 7 | 1 |
| p3 | 4 | 4 | 13 | 9 | 5 |
| p4 | 6 | 5 | 20 | 14 | 9 |
| p5 | 8 | 2 | 15 | 7 | 5 |

## Priority Based Scheduling

- Criteria : Highest Priority
  - Tie breaker : FCFS
- Type : Preemptive / Non-preemptive

Note : whether lower number is a higher priority or bigger number is a higher priority will always be in question, no one is default.

Priority can be

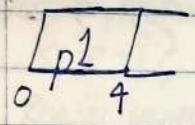
- Static → fixed (default)
- Dynamic → may increase or decrease

Eg. (for non-preemptive)

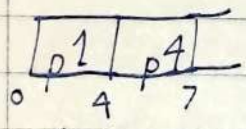
| Process | AT | BT | Priority |
|---|---|---|---|
| p1 | 0 | 4 | 4 |
| p2 | 1 | 2 | 5 |
| p3 | 2 | 3 | 6 |
| p4 | 3 | 1 | 10 (Highest) ← Given Highest Priority |
| p5 | 4 | 2 | 9 |
| p6 | 5 | 6 | 7 |

Tip: Draw a ready queue for easier tracking of processes

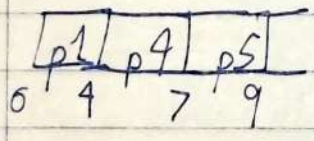
At time 0,



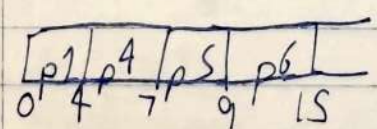

At time 4,
Ready Queue ⇒ p5, p4, p3, p2



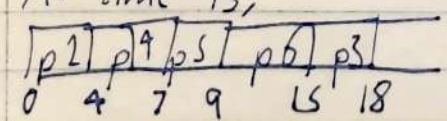

At time 7,
Ready Queue ⇒ p5, p3, p2, p6



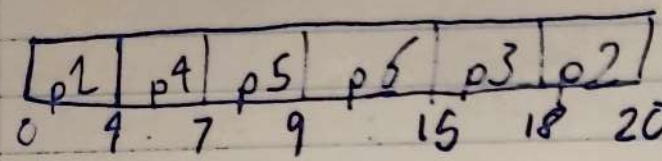

At time 9,




At time 15,



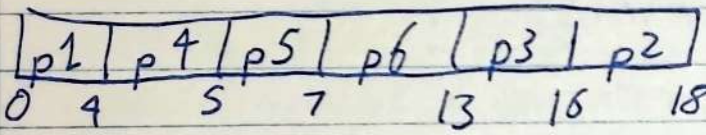
last process.




| Process | AT | BT | CT | TAT | WT |
|---|---|---|---|---|---|
| p1 | 0 | 4 | 4 | 4 | 0 |
| p2 | 1 | 2 | 20 | 19 | 17 |
| p3 | 2 | 3 | 18 | 16 | 13 |
| p4 | 3 | 1 | 7 | 4 | 3 |
| p5 | 4 | 2 | 9 | 5 | 3 |
| p6 | 5 | 6 | 15 | 10 | 4 |

OOPS! Got wrong times, process scheduling ~~is correct!~~ is correct!

Correct a gantt chart.



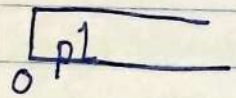

| Process | AT | BT | CT | TAT | WT |
|---|---|---|---|---|---|
| p1 | 0 | 4 | 4 | 4 | 0 |
| p2 | 1 | 2 | 18 | 17 | 15 |
| p3 | 2 | 3 | 16 | 14 | 11 |
| p4 | 3 | 1 | 5 | 2 | 1 |
| p5 | 4 | 2 | 7 | 3 | 1 |
| p6 | 5 | 6 | 13 | 8 | 2 |

# Eg. for preemptive

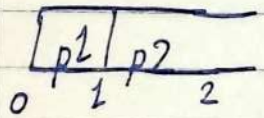
| Process | AT | BT | Priority |
|---|---|---|---|
| p1 | 0 | 4 | 4 |
| p2 | 1 | 2 | 5 |
| p3 | 2 | 3 | 6 |
| p4 | 3 | 1 | 10 (Highest) |
| p5 | 4 | 2 | 9 |
| p6 | 5 | 6 | 7 |

Note: Recommed to keep track of remaining BT and ready queue,

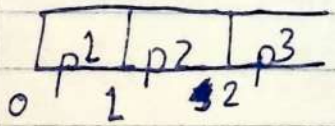
At $t=0$,
Ready queue ; p1



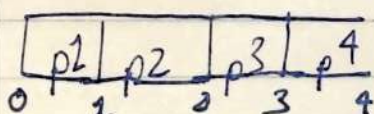

At $t=1$,
Ready queue = p1, p2




At $t=2$,
Ready queue = p1, p2, p3



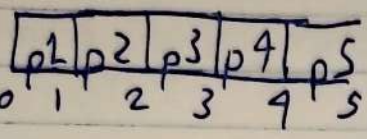

At $t=3$,
Ready queue, p1, p2, p3, p4



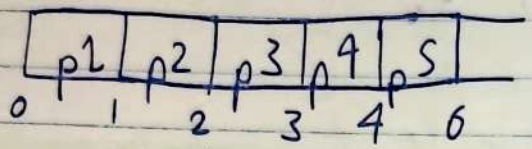

Burst Time tracking

| | |
|---|---|
| p1 | 4 - 3 - X |
| p2 | 2 - 1 - X |
| p3 | 3 - 2 - X |
| p4 | 1 - X |
| p5 | 2 - 1 - X |
| p6 | 6 - X |

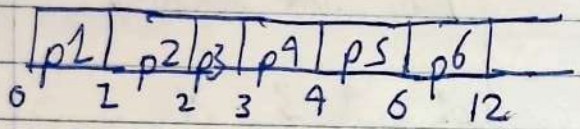
At $t=4$,
Ready queue; $p_1, p_2, p_3, p_5$



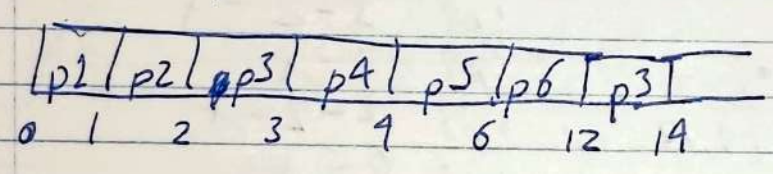

At $t=5$
Ready queue; $p_1, p_2, p_3, p_5, p_6$



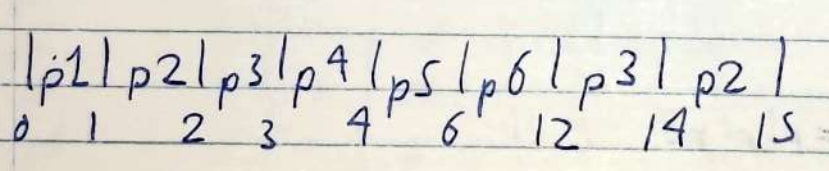

Now that all processes are in ready queue we can schedule strictly according to priority



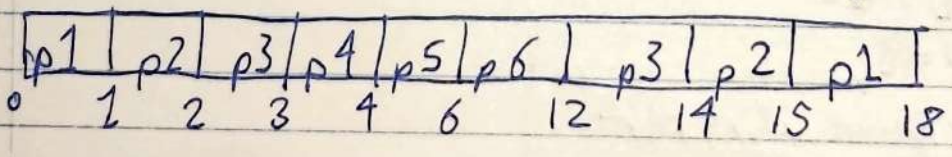

At $t=12$,




At $t=14$,




At $t=15$,



Table,

| Process | AT | BT | CT | TAT | WT | RT |
|---|---|---|---|---|---|---|
| $p^1$ | 0 | 4 | 18 | 18 | 4 | 0 |
| $p^2$ | 1 | 2 | 15 | 14 | 12 | 0 |
| $p^3$ | 2 | 3 | 14 | 12 | 9 | 0 |
| $p^4$ | 3 | 1 | 4 | 1 | 0 | 0 |
| $p^5$ | 4 | 2 | 6 | 2 | 0 | 0 |
| $p^6$ | 5 | 6 | 12 | 7 | 1 | 1 |

---

This algorithm also suffers from starvation, as, if high priority process keep arriving the low priority process may wait for indefinite time.

Solution of ~~Hgh~~ Starvation is Aging.

⇒ Aging is increase the priority of waiting processes after predefined time intervals.

⇒ Aging is possible only with dynamic priority.

# Lecture - 9

## Round-Robin

- Criteria : AT + Q (Q := Time Quantum)
  - Tie-breaker : Smaller PID
- Type : Preemptive.

Provides more interactiveness to our system and fairness to process scheduling.
Basically context switch every Q time units.

Eg,

| Process | AT | BT |
|---|---|---|
| p1 | 0 | 3 |
| p2 | 0 | 6 |
| p3 | 0 | 4 |
| p4 | 0 | 5 |

Q = 2

BT Tracker:

p1 3 - 1 - X
p2 6 - 4 - 2 - X
p3 4 - 2 - X
p4 5 - 3 - 1 - X

Gantt Chart

| p1 | p2 | p3 | p4 | p1 | p2 | p3 | p4 | p2 | p4 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 4 | 6 | 8 | 9 | 11 | 13 | 15 | 17 | 18 |

No. of context switches = no. of Gantt chart cells - 1
= 9

Table,

| | AT | BT | CT | TAT | WT | RT |
|---|---|---|---|---|---|---|
| p1 | 0 | 3 | 9 | 9 | 6 | 0 |
| p2 | 0 | 6 | 17 | 17 | 11 | 2 |
| p3 | 0 | 4 | 13 | 13 | 9 | 4 |
| p4 | 0 | 5 | 18 | 18 | 13 | 6 |

Note: Round robin sacrifices WT having small WT for better fairness and interactive system.

Eg, round robin with different arrival times

Arriving processes are placed at the end of ready queue.

| | AT | BT |
|---|---|---|
| p1 | 0 | 4 |
| p2 | 1 | 5 |
| p3 | 2 | 6 |
| p4 | 3 | 3 |
| p5 | 4 | 2 |
| p6 | 5 | 4 |

Q=2

We will keep track of the ready queue at the start of corresponding cycles.
Like all other preemptive algorithm, we also keep track of remaining burst times.

~~At th~~ During the cycle the process at start of ready queue will run.

| At start of cycle no: | Ready Queue |
|---|---|
| 0 | p1 |
| 2 | p2 p3 p1 |
| 4 | ~~p2 p3~~ p3 p1 p4 p5 p2 |
| 6 | p1 p4 p5 p2 p6 p3 |

After 6th cycle, all process have arrived so no longer need to update the above table and use the last pattern we get.

Burst time track

p1 4 - 2 - X

p2 5 - 3 - 1 - X

p3 6 - ~~2~~ 4 - 2 - X

p4 3 - ~~1~~ 1 - X

p5 2 - X

p6 4 - 2 - X

| p1 | p2 | p3 | p1 | p4 | p5 | p2 | p6 | p3 | p4 | p2 | p6 | p3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 19 | 20 | 22 | 24 |

## How to choose Quantum Value

If Q is very very small, then CPU efficiency will greatly degrade. This is because CPU will spend more time context switching rather than executing processes.

If Q is small, the system will be interactive.

Having longer Q, will make system less interactive.

Having Quantum very-very large will degrade Round Robin to FCFS.

Advantages of Round Robin

- Fairness
- Interactiveness for time-sharing system
- It is not required to know BT of processes.

## CPU utilization

The ratio of time under which CPU is being utilized.

If there are no I/O operations.
CPU utilization = 1

If there are I/O operations.

n := No. of processes running
p := average percentage of time processes are waiting for I/O

CPU utilization = $1 - p^n$

Lecture 10 is ques. on scheduling

## Lecture 11

## Multilevel Queue (MLQ) Scheduling

We will segregate ~~for~~ different types of processes into different queues and each queue can have its own scheduling algorithm implementation.
Queues refer to Ready Queues in this topic.
So we can have queues like

System Processes Queue

Foreground Processes Queue

Background Processes Queue

Scheduling Among Queues

- Fixed Priority preemptive scheduling method
- Time slicing

Fixed Priority

From our above queues let's consider priority as

Highest ↑ Lowest

System Process Queue
Foreground Process Queue
Background Process Queue.

Only if system processes queue is empty, a process from foreground process can run.

Similarly, only if no foreground process in queue, we can run a process from background queue.

As soon as the process arrives in a higher priority queue and if a process from lower priority queue is running in CPU. It will be immediately preempted to allow process of higher priority queue to run. (regardless of scheduling algorithms being used).

This method can cause starvation.

Time Slicing

We will give fraction of CPU execution time to different queues.

So we can split it as, for example,

$Q_1$ has 60% time
$Q_2$ has 30% time
$Q_3$ has 10% time

# Multilevel Feedback Queue (MLFQ) Scheduling

To avoid starvation and make multilevel queues more flexible, we use MLFQ.

We will move processes in high priority queue to lower priority queues if they take too long to complete execution.

We move processes in low priority queues up if they have been in waiting for too long.

## Advantages and Disadvantages of Algorithms

### FCFS

Advantages
- Easy to implement
- No complex logic
- No starvation

Disadvantages
- No option for preemption
- Convoy Effect

### SJF

Advantages
- Minimum average waiting time for non-preemptive scheduling
- Better throughput

Disadvantages
* No practical implementation (BT cannot be predetermined).
* No preemption
* Longer process suffer from starvation

SRTF

Advantages
* Minimum average waiting time
* Better throughput

Disadvantages
* Not practical
* Starvation

Priority Based

Advantages:
* Better response for real time solutions

Disadvantages:
* Lower priority process suffer from starvation.

Round Robin

Advantages
* No starvation
* Better interachiveness
* BT not required (Practical algorithm)

Disadvantages
* Higher average waiting time and TAT time.
* Can degrade to FCFS. (Q too big)

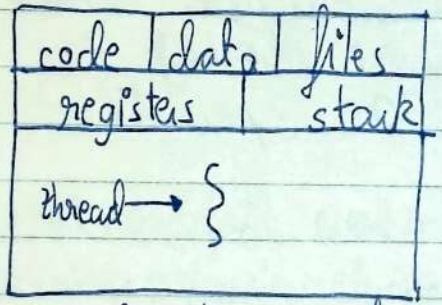
Do CPU Scheduling marathon session

## Lecture - 12

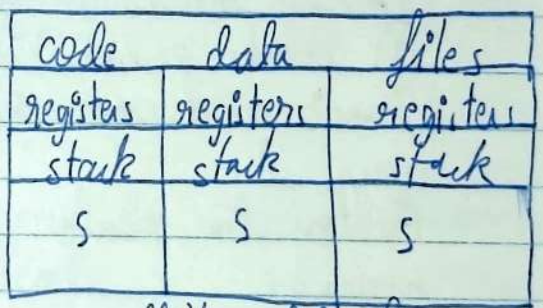
## Thread

Lightweight process. If we want to use same process for various inputs, rather than creating new instance of processes, we use threads. The thread will share resources with process and increase performance through parallelism.

| Shared Among Threads | Unique For Each Thread |
|---|---|
| Code Section | Thread ID |
| Data Section | Register Set |
| OS Resources | Stack |
| Open files & Signals | Program Counter. |




single threaded process




multithreaded process

## Advantages of Multithreading

1. Responsiveness
2. Faster Context Switches
3. Resource Sharing
4. Economy
5. Communication
6. Utilization of Multiprocessor Architecture. (Modern CPUs have multiple threads running in parallel).

Types of Thread

- User Level Thread (Managed by process)
- Kernel Level Thread (Managed by OS)

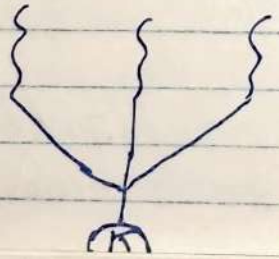
Operating System is not aware of user level threads

| User Threads | Kernel Thread |
|---|---|
| Multithreading in user process. | Multithreading in kernel process. |
| Created without kernel intervention | Kernel itself is multithreaded |
| Context switch is very fast. | Context switch is slow. |
| If one thread is blocked, OS blocks entire OS | Individual thread can be blocked |
| Generic and can run on any OS. | OS Specific |
| Faster to create and manage | slow to create and manage. |

## Multithreading Models

(Based on how user level thread can communicate with kernel thread)

1. Many-to-one

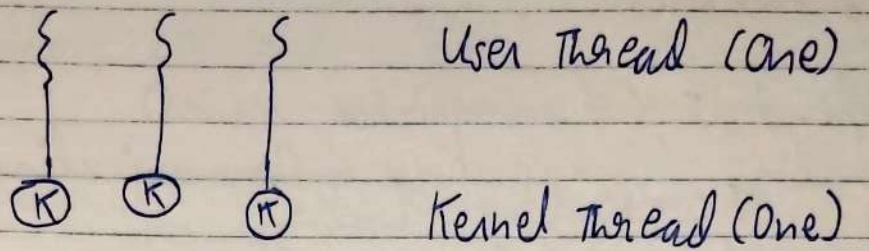


User level Threads (Many)

Kernel Thread (One)

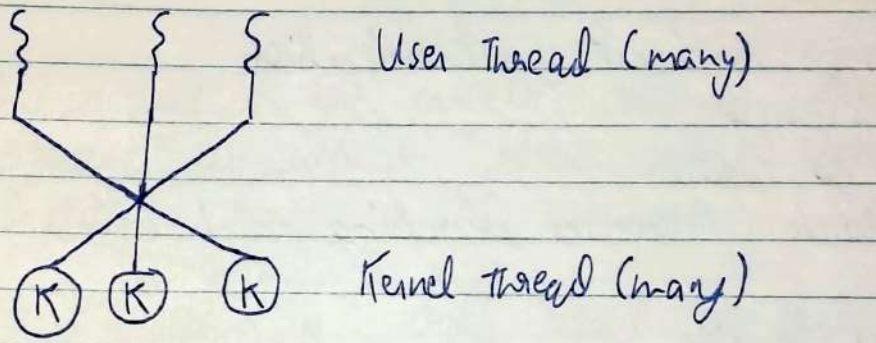
Multiple user threads can communicate with a single kernel thread

2. One-to-One




Single user thread can communicate only with single kernel thread.

3. Many-to-many




Multiple user threads can be associated with multiple kernel threads

# Lecture-13

Types of Processes
- Independent (No synchronization required)
- Cooperating / Coordinating / Communicating

Synchronization ⇒ Rules such that two processes can use shared resources.

## Need of Synchronization

- Problems without synchronization
  - Inconsistency
  - Loss of data
  - Deadlock (Processes blocking each other)

## Critical Section

The critical section is a code segment where the shared variables can be accessed. Hence, segment of code which requires synchronization.

Rest is sometimes reffered to as remainder section.

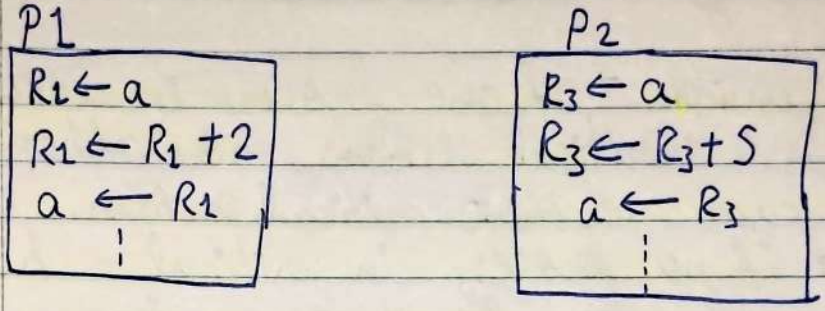
## Race Condition

A race condition is an undesirable situation. It occurs when the final result of concurrent / cooperating processes depends

on sequence in which they complete their execution.

Suppose the given below

$\boxed{a = 5}$ shared resource




final result in (a) will change based on sequence in which these ~~comp~~ communicating processes run.

If p1 and p2 both read a before the other is done updating value, the value will change. If p1 finishes first then also (a) will be effected. This makes value of (a) indeterminable.

# Lecture-14

## Solution for critical section problem (Software)

We have to use synchronization to solve this problem.

- Requirments of Critical section problem solution
  - i. Mutual Exclusion ⇒ If one process is using critical section, no other process can use that critical section.
  - ii. Progress ⇒ If no process in critical section and atleast one process wants to enter critical section, then it should be allowed.
  - iii. Bounded Waiting ⇒ We have to reduce waiting time. So we will not allow the same process to enter the C.S. if other processes are waiting. So bounded waiting means that our processes should not be in indefinite ~~entess~~ or endless waiting

Bounded means within bounds, i.e, not endless waiting.

We divide it as,

- ☆ Entry section
- ☆ Critical section
- ☆ Exit section

will be structure of our solutions

## Solution 1: Using Lock

Boolean lock = false;

```
while (true) {
    while (lock);      } Entry
    lock = true;       } section
    // Critical Section
        :
    lock = false;      } Exit Section.
    // Remainder Section
}
```

P2 will have same code (as shown below) If P1 enters first

```
P1                                  P2
while(true) {                       while(true) {
  while(lock);                        while(lock);
  lock = true;   ---locks execution---> lock = true;
  // CS            further             // CS
  lock = false;                       lock = false;
  // RS                                // RS
}                                   }
```

Unlock execution for P2 (or any other waiting process)

Does not satisfy mutual exclusion. For our above example, if process P1 is preempted after while(lock) loop in entry section, but before it can set lock = ~~test~~ true; then both P1 and P2 can enter C.S.

It does satisfy progress.

## Solution 2: Using Turn

```
int turn = 1;

P1                                  | P2
                                    |
while(true) {                       | while(true) {
    while(turn != 1);               |     while(turn != 2);
    // CS                           |     // CS
    turn = 2;                       |     turn = 1;
    // RS                           |     // RS
}                                   | }
```

Satisfies mutual exclusion

~~Satisfies progress~~

Does not satisfy progress (If P2 enters first, it cannot execute).

~~Has Bounded~~

Processes run in strict order.

## Solution 3: Peterson's Solution

Suppose we have n processes
$P_0, P_1, P_2 \dots P_{n-1}$

Then we have shared resources
Boolean flag[n];
int turn = 0;

So for processes Pi and Pj, that communicate with each other,

```
Pi:
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn==j);   } Entry section
    //CS
    flag[i] = false;              } Exit Section.
    //RS
}
```

And

```
Pj:
while (true) {
    flag[j] = true;
    turn = i;
    while (flag[i] && turn==i);
    //CS
    flag[j] = false;
    //RS
}
```

Peterson's can also be used for n processes, but, This solution is for two processes (n process one does not seem to be in GATE).

- Mutual Exclusion is satisfied
- Progress is satisfied
- Bounded waiting is satisfied

~~P Block elements~~

## Lecture - 15

## Synchronization Hardware

We have ~~as~~ synchronization hardware for hardware solution of critical section problem.

Hardware solution ~~for~~ provides following two instructions support by CPU to provide synchronization

1. Test And Set ()
2. Swap ()

These are privileged instructions. Can only be run in kernel mode and not in user mode.

### Test And Set ()

This function is implemented as

~~bool TestAndSet(bool flag);~~
bool TestAndSet (bool *lock);

It will return current value of lock and will set lock to true.

```
bool TestAndSet(bool *trg){
    bool rv = *trg;
    *trg = true;
    return rv;
}
```

This not a software implementable, but a visualization of process

Since, it is hardware instruction, it cannot be preempted and is atomic.

So in our process,

```
while(true){
    while(TestAndSet(&Lock));
    //CS
    Lock = False;
}
```

initially, Lock = False
When we use while loop (or any other infinite loop) to block a process, it is in CPU but is continuously in execution ~~and~~ of the loop and not actually suspended, hence it is called busy waiting.

## Swap()

This function swaps values of two bools.

```
void Swap(bool *a, bool *b);
```

The function looks as follows (Note: It is only a visualization, actual implementation is hardware).

```
void Swap(bool *a, bool *b){
        bool temp = *a;
        *a = *b;
        *b = temp;
}
```

and our process,

```
while(true){
    Key = True;
    while(key == true)
        Swap(&Lock, &Key);
    // CS
    Lock = false;
    // RS
}
```

Initially, key = nil ; lock = false.

In preemptive system, this does not satisfy mutual exclusion. If one process is preempted after setting Key to true, then two processes can enter critical section.

# Lecture-16

## Synchronization Tools

- Semaphore
- Monitor.

## Semaphore

A synchronization tool which can be accessed using two functions only:

- wait() / P() / Degrade() / lower() / down()
- signal() / V() / Upgrade() / up()

## Types of Semaphore

- Binary Semaphore (Semaphore is either 0 or 1)
- Counting Semaphore (any integer)

Binary Semaphore ⇒ mainly used for solution to critical access problems with multiple processes

Counting Semaphore ⇒ mainly used to control access to a resource that has multiple instances.

## Critical Section solution with semaphore

Initially S = 1.

Initially S=1

Then our process

```
while(true){
    wait(S);
    //CS
    signal(S);
    //RS
}
```

wait() decrements S and stops other processes if $S \leq 0$
signal() increments S and resumes waiting processes.

Both wait() and signal() are atomic. Therefore they run without being interrupted.

wait():
stop process if $S \leq 0$
decrement S

signal():
increment S
resume/allow new processes.

Atomic function if interrupted will have no lasting effect and everything is reset to before function call.

So if atomic functions are interrupted. they have no effect on the processes.

## Characteristics of semaphores

1. Used to provide mutual exclusion
2. Used to control access to resources.
3. Semaphores are machine-independent.
4. Solutions using semaphore may cause deadlock.
5. " " " starvation
6. " " " priority inversion.
7. Semaphore can have busy waiting

Priority inversion : A low priority process may in enter C.S. first and blocks a high priority process is called priority inversion.

Note: Initial value of S for solution can be different based on implementation of P() and V()

So eg, if P() stops processes if $S<0$; then initial $S = 0$;

Lecture 17 is mainly questions on semaphore.

Date / /
Page

# Lecture - 17

## Possible implementation for P() and V()

Following has busy waiting.

```
void wait(int S) {
    while(S<=0);
    S--;
}

void signal(int s) {
    S++;
}
```

**Race Condition**: If concurrent processes are working on a shared variable, then the process that ends last is the one who's effects will last (if we consider that the concurrent processes read from disk before another could writeback.)

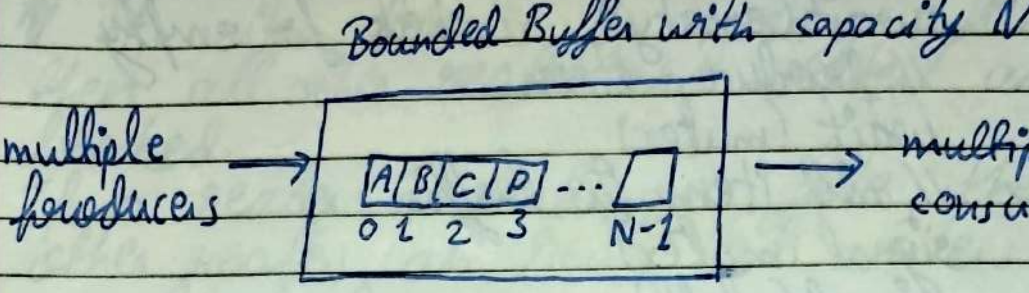
Eg; If $P_1$ and $P_2$ are

| $P_1$ | $P_2$ |
|---|---|
| $R_1 \leftarrow x$ | $R_2 \leftarrow x$ |
| $R_1 \leftarrow R_1 + 2$ | $R_2 \leftarrow R_2 - 2$ |
| $x \leftarrow R_1$ | $x \leftarrow R_2$ |

If $P_1$ reads $x$, and then $P_2$ reads $x$; the value of $x$ depends on which process ends first.

# Lecture-18

## Bounded Buffer Problem




Producer will produce item and put it in bounded buffer. (only if there is space in buffer)

Consumer will take the item from buffer (only if bounded buffer is not empty).

Both producer and consumer cannot access buffer at the same time.

### Solution

Variables:

- Mutex: Binary semaphore for mutual exclusion between producer and consumer.
- Full: Counting semaphore to denote no. of occupied slots in buffer
- Empty: Counting semaphore to denote no. of empty slots.

Date / /
Page

## Producer ():

```
Producer () {
    wait(empty)  // check for empty
    // produce item
    wait (mutex)
    // add item to buffer
    signal (mutex)
    signal (full)
}
```

## Consumer ():

```
consumer () {
    wait (full) // check for item
    wait (mutex)
    // remove item from buffer
    signal (mutex)
    // consume the item
    signal (empty)
}
```

## Initialization

```
mutex = 1
full = 0          // no slots occupied
empty = number of slots  // all slots empty
```

# Lecture -19/20

## Readers-Writers Problem

- If writer process is accessing a file, then all other reader and writers are blocked.
- If a reader is reading a file, then other readers can read but writers are blocked.

## Solution

### Variables

- mutex : Binary semaphore to provide mutual exclusion
- wrt : Binary semaphore to restrict readers and writers if writing is going on.
- readcount : Integer variable to denote no. of readers.

### Initialization

```
mutex = 1
wrt = 1
readcount = 0
```

### Writer():

```
writer() {
    wait(wrt)
    // perform writing operation
    signal(wrt)
}
```

Date / /
Page

## Reader ():

```
reader() {
    wait(mutex);
    readcount++;
    if(readcount == 1)
        wait(wrt);
    signal(mutex);
    // perform read operation
    wait(mutex);
    readcount--;
    if(readcount == 0)
        signal(wrt);
    signal(mutex);
}
```

Note: Since read count is a shared variable, we use mutex semaphore to protect critical sections where readcount is used.

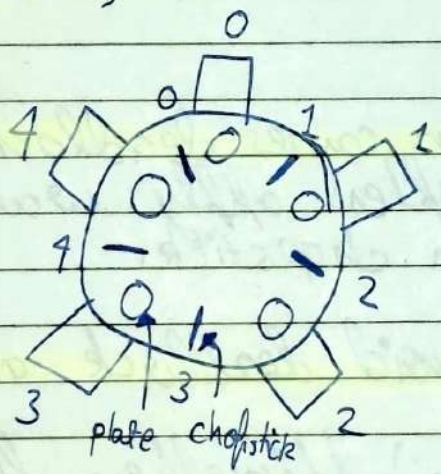
If the first reader starts execution then we use wrt to stop writers from entering their operation.

Once no more readers are in execution, we signal wrt to allow writer processes.

## Dining Philosophers Problem

Suppose there are 5 philosophers, each ~~they/can~~ can do 2 operation at a time either eat or think.

They are eating noodles and each philosopher has one chopstick. But they require 2 chopstick to eat.
Let's number the chopsticks and philosophers as below,




Let's suppose we have n philosophers, then the ith philosopher will use ~~child~~ chopsticks (i) and [(i+1)%n].

So in our example, ith philosopher will use (i) and ((i+1)%5) chopstick.

Date / /
Page

## Solution

We will use binary semaphores in an array to solve this problem.
So for our example,
Binary semaphore array chopstick[5] = {1,1,1,1,1};
(Initial)

And our solution is,

```
{
  wait(chopstick[i]);
  wait(chopstick[(i+1)%5];
  // eat operation
  signal(chopstick[i]);
  signal(chopstick[(i+1)%5];
}
```

Note: This solution causes deadlock if all philosopher's apply wait on their own chopstick.

Some ways to avoid deadlock are as follows:

1) Allow only (k-1) philosophers for execution at a time.
2) A philosopher can apply wait on chopstick if both are available.
3) One philosopher should pick the left chopstick first and then right; while all others pick the right first and then left. (or vice versa)

1 will run ⇒ | others will run

```
wait(chopstick[(i+1)%n]      | wait(chopstick[i]);
wait(chopstick[i])           | wait(chopstick[(i+1)%n];
// eat                       | // eat
signal(chopstick[(i+1)%n])   | signal(chopstick[i];
signal(chopstick[i])         | signal(chopstick[(i+1)%n]);
```

# Lecture 21

Resources

- Hardware ( I/O devices, RAM, CPU)
- Software ( files)

3 operations on resources by a process

- Request (request is done to OS)
- Use (when OS allocates resource to process)
- Release (when use completed, process releases the resource)

## Deadlock

If two or more processes are waiting for an event which is never going to occur, it causes deadlock.

Deadlock only occurs when all following conditions are satisfied

1. Mutual Exclusion ⇒ if one process is using resource, another cannot.
2. Hold & Wait ⇒ As seen in dines philosopher problem, then deadlock occurs if all processes

hold one resource and are in indefinite waiting for another.

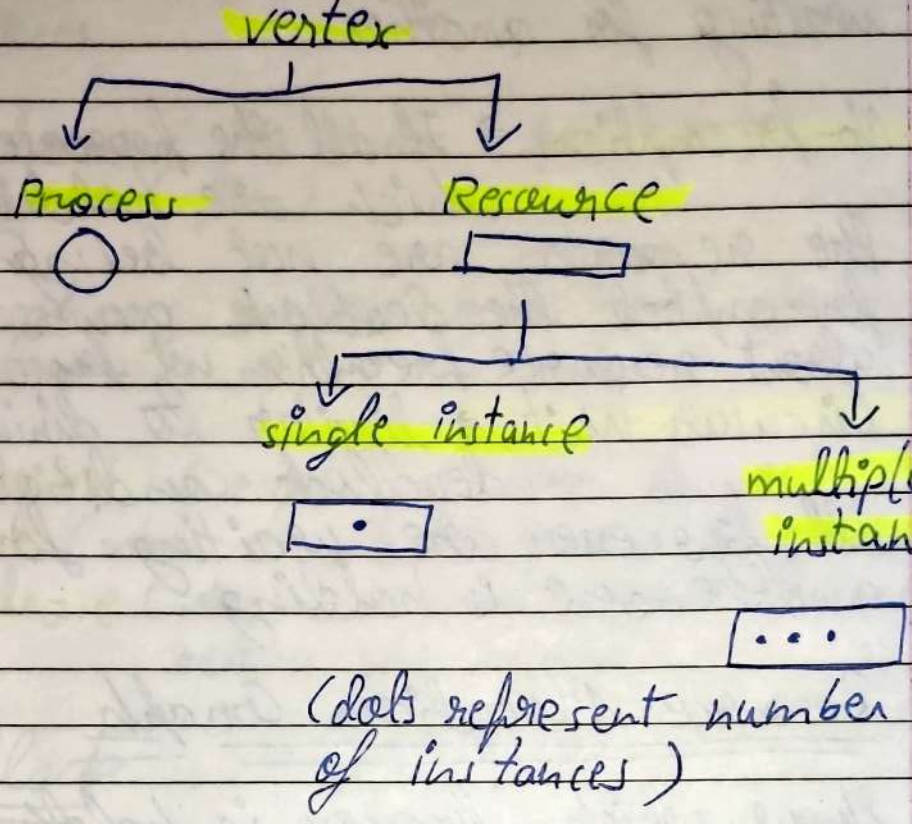
3. No-preemption ⇒ If all the processes which are holding the resources are not being pre-empted the deadlock occurs. (Talking about resource preemption not process preemption)
4. Circular Wait ⇒ Similar to dining philosophers deadlock condition where all processes are waiting for resource another one is holding.

## Resource Allocation Graph

Shows which process is holding which resource and waiting for which resource.

vertex
- Resources → having single instance
  - having multiple instances of same resource. Eg. the cores on a CPU can all perform same functions and hence are called multiple instances of the resource. or if there are multiple I/O of same type like printers.
- Processes

# Symbols of vertexes

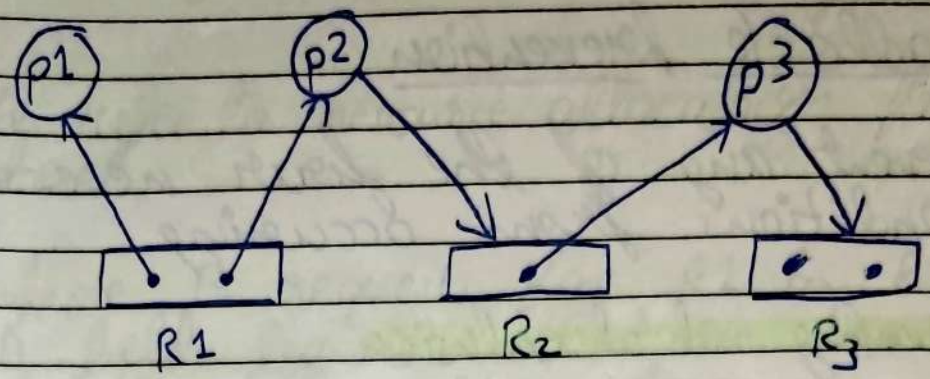



## Edges

- **Request** (from process to resource)
- **Allocation** (from instance to process)

Edges are directed in resource allocation graph.

Example,




## Recovery from deadlock

1. Make sure deadlock never occur
   - Prevent the system from deadlock
   - Avoid deadlock
2. Allow deadlock, detect and recover
3. Pretend there is no any deadlock.

We will study three major portion

- deadlock prevention
- deadlock avoidance
- deadlock detection

# Lecture -22

## Deadlock prevention

Prevent any of the four necessary conditions from occuring.

1. Preventing mutual exclusion
   - Have enough resources for all processes
   - Make all processes independent

   Neither of above 2 are practical.

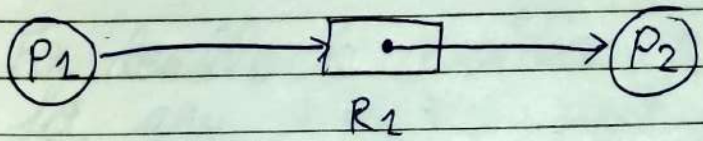
2. Preventing Hold & Wait
   - If all required resources are available the acquire all or else wait for all. This method is prone to starvation.
   - If process is trying to acquire a resource which is not available, while holding some other resources then process will release the allocated process

   If a process holds resources and not using them, then it is poor utilization of resources.

3. Preventing No-freemption (Allowing preemption)

We refer to resource allocation freemption here.

Suppose $P_1$ requests for R1 & R1 is held by P2. But P2 is in waiting for other processes / resource




In this condition OS may preempt $R_1$ from $P_2$ and give it to $P_1$.

4. Preventing circular wait

We do the following to avoid circular wait:

i) Give unique numbers to all the resources. Process will be holding a resource $R_i$

ii) Any process while holding a resource $R_i$, can be requested for $R_j$ only if $j > i$.

iii) If a process is holding a resource $R_i$ and wants another resource $R_j$ when $j < i$. Then process will ~~have to release $R_i$~~ not be allowed to do so.

Basically we assign priority number to each resource. A process can't request for lesser priority resource than it is working on.

This eliminates circular wait.

Date / /
Page

# Lecture - 23

## Avoid Deadlock

The OS tries to keep system in safe state

Safe State ⇒ Condition with no possibility of deadlock.

Unsafe State ⇒ State of system in which deadlock is possible.

For deadlock avoidance, the request for any resource will be granted if the resulting state is not an unsafe state.

## Banker's Algorithm

It is a resource allocation and deadlock avoidance algorithm that tests for safety.

Suppose a scenario, we have m processes and n types of resource (each can have multiple instances)

## Banker's safety algorithm

We use following data structures for this algorithm.

Input structures:

i) Allocation matrix

A matrix of size m X n.
It stores which resources how many instances of each resource have been allocated to each process

Eg,

| | $R_1$ | $R_2$ | $R_3$ | $R_4$ |
|---|---|---|---|---|
| $P_1$ | 0 | 0 | 1 | 2 |
| $P_2$ | 1 | 0 | 0 | 0 |
| $P_3$ | 1 | 3 | 5 | 4 |
| $P_4$ | 0 | 6 | 3 | 4 |
| $P_5$ | 0 | 0 | 1 | 4 |

ii) Max matrix

A matrix of size m X n.
It stores the total instances of each resource needed by process to complete it's execution
Eg,

| | $R_1$ | $R_2$ | $R_3$ | $R_4$ |
|---|---|---|---|---|
| $P_1$ | 0 | 0 | 1 | 2 |
| $P_2$ | 1 | 7 | 5 | 0 |
| $P_3$ | 2 | 3 | 5 | 6 |
| $P_4$ | 0 | 6 | 5 | 2 |
| $P_5$ | 0 | 6 | 5 | 6 |

iii) Available array

An array of size n.
Stores how many instances of each resource are currently free/available.
Eg,

| $R_1$ | $R_2$ | $R_3$ | $R_4$ |
|---|---|---|---|
| 1 | 5 | 2 | 0 |

During the algorithm we use two more data structes.

i) Need matrix
stores how many resources that need to be allocated now in order for process to complete execution

Need = Max - Allocation

ii) Finish array
Used to keep track of processes that are finished. Typically, at start all values in this array are False.

| $P_1$ | $P_2$ | $P_3$ | $P_4$ |
|---|---|---|---|
| F | F | F | F |

## The Algorithm

Note: Sometimes we initially make an extra variable work = available, we do this to avoid changing available itself, but it is not necessary. Though it is recommended because we don't want to allocate before checking.

1. Initialize ; Work = Available (optional)
   Finish [0....n] = False and calculate Need
2. Find a process; such that :
   a) Finish [i] = false and
   b) Need [i] $\leq$ Work
   (Get ith row)

   If no such process, go to step 4.

3. For this process found
   Work = Work + Allocation [i]
   (Get ith row)

   Finish [i] = true
   go to step 2

4. If For all Finish [i] = True
   then system in safe state
   Else
   System in unsafe state.

Hdr; compare rows of need with available finding a process whose all numbers are <= to equivalent in available; and add that row of allocation to available.
Repeat until all process finish

Eg, Suppose given

| Process | Allocation | | | | Max | | | | Available | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | D | A | B | C | D | A | B | C | D |
| $P_1$ | 0 | 0 | 1 | 2 | 0 | 0 | 1 | 2 | 1 | 5 | 2 | 0 |
| $P_2$ | 1 | 0 | 0 | 0 | 1 | 7 | 5 | 0 | | | | |
| $P_3$ | 1 | 3 | 5 | 4 | 2 | 3 | 5 | 6 | | | | |
| $P_4$ | 0 | 6 | 3 | 4 | 0 | 6 | 5 | 2 | | | | |
| $P_5$ | 0 | 0 | 1 | 4 | 0 | 6 | 5 | 6 | | | | |

Finding Need

| | A | B | C | D |
|---|---|---|---|---|
| $p^1$ | 0 | 0 | 0 | 0 |
| $p^2$ | 0 | 7 | 5 | 0 |
| $p^3$ | 1 | 0 | 0 | 2 |
| $p^4$ | 0 | 0 | 2 | 2 |
| $p^5$ | 0 | 6 | 4 | 2 |

First after $p^1$
Available = 1 5 2 0

Second after $p^3$
Available = 2 8 2 2

First after $p^1$
available += allocation[p1]
available = 1 5 3 2

Second after $p^2$
available = 2 8 3 2

Date / /
Page

First after $p^2$
available += allocated [p1]
available = 1 5 3 2

Second after p3
available += allocated [p3]
available = 2 8 8 6

Third after p2
available += allocated [p2]
available = 3 8 8 6

Fourth after p4
available = 3 14 11 10

Fifth after p5
available = 3 14 12 14

All process complete, hence system in safe state with safe sequence

p1, p3, p2, p4, p5

Safe sequence → a sequence to successfully complete all processes without a deadlock.

A single scenario can have multiple safe sequences.

# Lecture 24

## Resource Request Algorithm (Banker's Algo.)

This is a simple algorithm that builds on the previous safety algorithm.

This one is used during resource allocation when a process sends a Request.

A request can be seen as an array of the size equal to number of resources.

The algorithm:

1. If $Request_{P_i} \leq Need_{P_i}$

   Goto step 2

   Else

   Error since process has exceeded its maximum claim.

2. If $Request_{P_i} \leq$ Available

   Goto step 3

   Else

   $P_i$ must wait, since resource not available

3. Have system pretend that, allocated the requested resources to $P_i$ in a

virtual environment, modify the following

$Available = Available - Request_{Pi}$

$Allocation_{Pi} = Allocation_{Pi} + Request_{Pi}$

$Need_{Pi} = Need_{Pi} - Request_{Pi}$

Then use safety algorithm to test is system will be in safe or unsafe state

4. If in safe state
   Allocate resources to process
   Else
   Decline request and don't allocate resources

Date / /
Page

# Lecture-25

## Relation between no. of instances and deadlock

Suppose we have a condition

| Processes | Max |
|---|---|
| p1 | k |
| p2 | k |
| p3 | k |
| ⋮ | ⋮ |
| pn | k |

The maximum number of instances such that deadlock can still occur can be calculated if we suppose given k-1 to all processes and have no instance left.

| Processes | Max | Allocation |
|---|---|---|
| p1 | k | k-1 |
| p2 | k | k-1 |
| p3 | k | k-1 |
| ⋮ | ⋮ | ⋮ |
| pn | k | k-1 |

This is a deadlock situation. So maximum can be calculated by adding allocation column.

Therefore,

For a resource the

max instances that can cause deadlock = $n(k-1)$

min instances to avoid deadlock = $n(k-1)+1$

Note : If all processes don't have equal max, then process steps in the previous page is applied

Here, $n$ = no. of processes
$k$ = max instances need by each

To find max instance if all processes don't need same amount of instance

Suppose we are asked max instances in which deadlock can occur for given :

| Process | Max |
|---|---|
| p1 | 5 |
| p2 | 4 |
| p3 | 6 |
| p4 | 2 |

We subtract 1 from max column and allocate that many resources

| Process | Max | Allocation |
|---|---|---|
| p1 | 5 | 4 |
| p2 | 4 | 3 |
| p3 | 6 | 5 |
| p4 | 2 | 1 |
| | | $\Sigma = 13$ |

So now max instances that may cause deadlock is summation of allocation

Therefore 13.

And minimum instances is one more than that

Therefore, min instances to avoid deadlock = 14

---

Condition for no deadlock

resource instances $\geq n(k-1)+1$

$n \Rightarrow$ no. of processes
$k \Rightarrow$ maximum instances needed by each process

# Lecture - 26

## Deadlock detection & Recovery

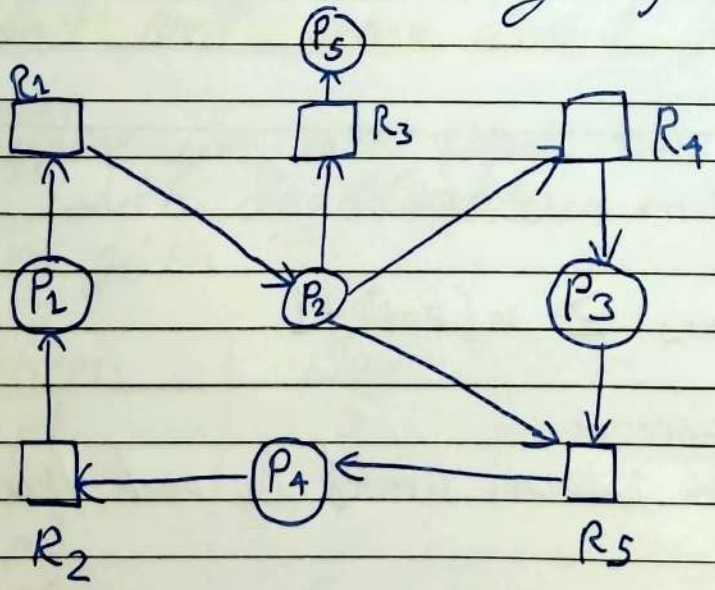
Deadlock detection :

1. All resources have single instance.
   We use wait-for-graph
2. Resources have multiple instances.
   We use a detection algorithm
   (Banker's Safety Algorithm)

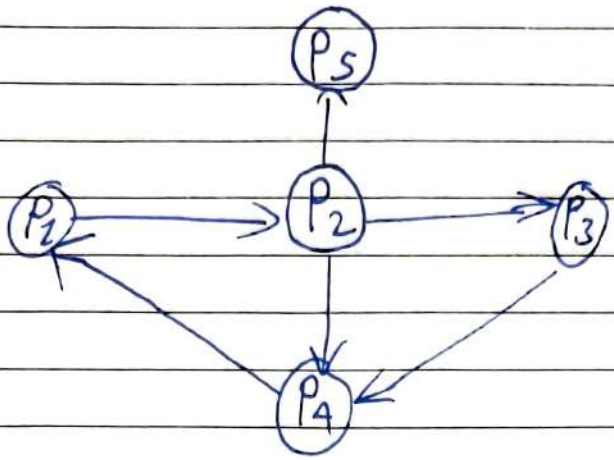
### Wait For Graph

Created using resource allocation graph.

To make wait-for-graph
a) Remove resources nodes
b) Join edges from $P_i$ to $P_j$ if $P_i$ is waiting for a resource held by $P_j$.




Resource allocation graph




Corresponding wait-for-graph

If a cycle exists in wait-for-graph, then deadlock
Else no deadlock.

⇒ Cycle does not need to cover each node of process for deadlock
⇒ Processes which are a part of any cycle are in deadlock.
⇒ Processes which are not in any cycle are not in deadlock and can complete execution.

If there are multiple instances

No cycle means no deadlock.

Cycle means there may or may not be deadlock (indeterminable)

Ff

For multiple instance resources. We check for deadlock using Banker's safety algorithm

## Detection-Algorithm Usage

We can do detection algorithm in two main ways.

1. Do deadlock detection after every resource allocation. Algorithm is used very frequently in this case and negatively affects performance.
2. Do deadlock detection only after some time (like drop in CPU throughput or a set interval of cycles). This is more performant.

## Recovery from deadlock

1. Inform system operator (user) and allow him/her to take manual steps
2. Terminate one or more processes involved in the deadlock
3. Preempt resources from processes.

Date / /
Page

# Process Termination

1. Terminate all processes involved in the deadlock. It is not recommended because too much progress is lost.

2. Terminate process one by one until deadlock is broken. This solution is more time costly but better because less progress is lost. compared to previous solution.

Factors deciding which process to terminate next in one by one method

i) Process priorities
ii) How close process is to finish and how long it has been running
iii) How many & what resource process is holding
iv) How many more resources process needs to complete execution
v) How many need to be terminate in conjunction
vi) Whether process is interactive or batch.

# Resource Preemption

Issues to address when preempting resources :

i) Selecting victim
We need to reduce victims of the preemption and try to resolve deadlock with least no. of preemptions

ii) Rollback
Before preemption the resource, we need to rollback the progress of the process and the state of resource, so that other process and later the same process can work on that resource predictibly and avoid race condition

iii) Starvation
If a single process's resources are continuously preempted, it can cause starvation.
We can change victims to avoid this.

# Lecture-27

## System Call

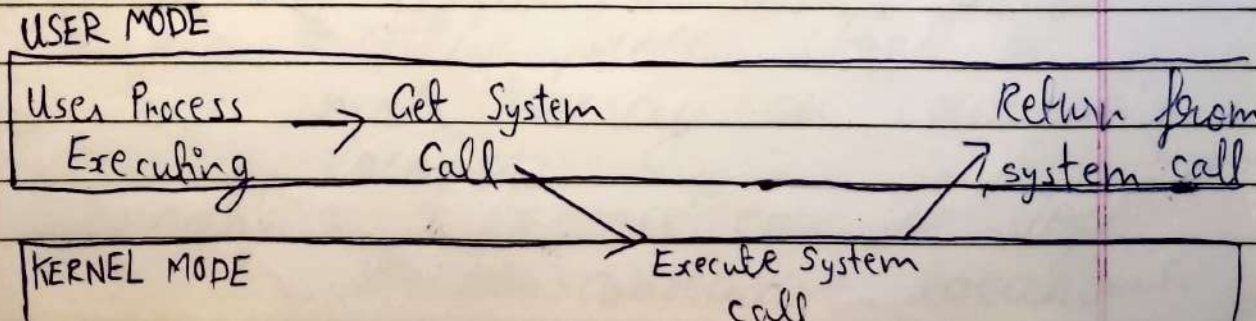
Programmabic way in which a computer program requests a service from the kernel.

## How system call works

1.) Request of process from user space
2.) Interrupt request for kernel
3.) Control transfer to kernel
4.) Kernel checks if operation can be done
5.) Kernel performs operation
6.) Kernel returns output to user processes

## Services by System Calls

1. Process creation and management
2. Main memory management
3. File, Directory and File system management
4. Device handling (I/O)
5. Protection (get/set permissions)
6. Networking



# fork()

Fork system call is used for creating a new process, which is called child process.
child & parent processes then run concurrently

fork() takes no parameters and returns an integer value.

- Negative value : creation of child process was unsuccessful
- Zero : returned to the child process
- Positive value : The value of child process ID (PID). returned to parent process

Note : child process will continue execution after the fork call that created it (not from start)

If we have n fork()'s in a row, then we will have $2^n$ total processes. (since each fork will double the number of processes)

# Lecture - 28

## Memory Management

A module of OS which is for memory management does the following

i) Memory allocation
ii) Memory deallocation
iii) Memory protection (process can only access memory allocated to it)

## Goals of memory management

i) Maximum utilization of memory space (Wastage of memory is called memory fragmentation) (we want min. fragmentation)
ii) Ability to run larger programs with limited space. (using virtual memory concept)

## Memory Management Techniques (MMT)

- Contiguous
- Non-contiguous

Contiguous ⇒ entire process stored in main memory on consecutive locations.

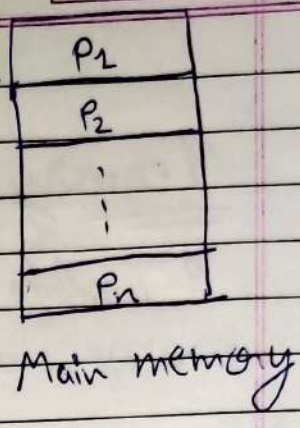
Non-contiguous ⇒ process can be stored on non-consecutive locations.

Contiguous

↳ Fixed partition
↳ Variable partition

Non-contiguous

↳ Paging
↳ Segmentation




## Fixed Partition Contiguous MMT

Main memory is divided into fixed no. of partitions (of variable size) and each partition can be used to accomodate maximum one process.

The process are allocated a partition according to a Partition Allocation Policy.

### Partition allocation policy types

i) First fit ⇒ Process is allocated the first empty partition in which it can fit. (Linear search for partition)

ii) Best fit ⇒ Process is allocated to the smallest partition that

Note: Max degree of multiprogramming is equal to no. of partitions in fixed partition contiguous MMT.

Date / /
Page

is free and can fit the process.

iii) Worst fit ⇒ The process is allocated to partition which is the largest in size.

iv) Next fit ⇒ Similar to first fit, but the search continues after the partition which was previously used for allocation. If we reach the end then go back to start.

** The extra space is allocated to process than what is required, this wastage of space is called internal fragmentation in fixed contiguous MMT.
Internal fragmentation is very probable in fixed partition contiguous MMT.

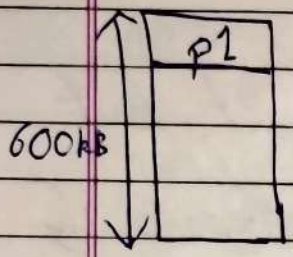
## Variable Partition Contiguous MMT

Main memory is not divided into partitions initially.
When a new process arrives, a new partition is created of same size as process size and process is allocated to this partition.

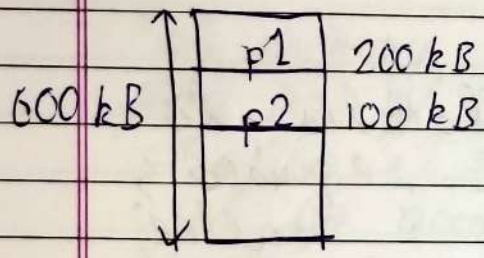
Hence no internal fragmentation.

Suppose we have a main memory of size 600 kB.

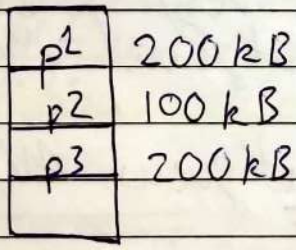
First we allocate a process of 200 kB



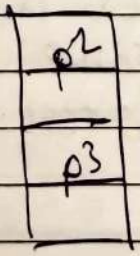

Then we allocate a process of 100 kB




Then we allocate a process of 200 kB




Now p2 completes execution



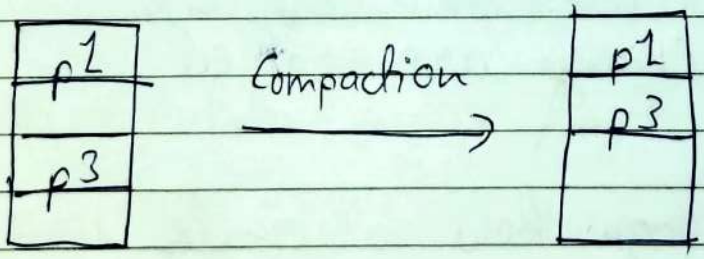
Now suppose, a process of size 150 kB needs to be allocated.

Even though we have enough free space in memory, it is not continuous. Thus it can't be allocated. This wastage of space is called external fragmentation.

## Solution of external fragmentation

Compaction is solution. We collect all allocated processes into one side of memory, so we can utilize the entire free space.

So in our example,




Compaction is expensive (time consuming)

# Lecture-29

## Non-contiguous MMT

Process is scattered in memory, not allocated at one area

Techniques

- Paging : Scattered in same size of memory areas
- Segmentation : Scattered in variable sizes of memory areas.

## Paging

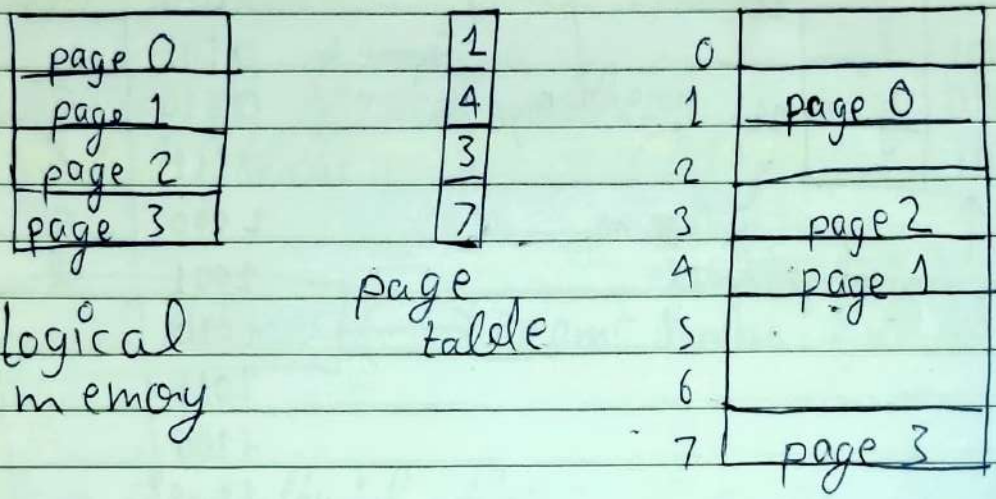
We divide process into equal sizes, these divisons are called pages.
We then divide main memory into parts of the same size as a page size. These divisons of main memory are called frames.

* Each frame can hold a single page at a time.
* The pages can be randomly distributed across the frames.

We use a page table to keep

track of the locations of the pages.

The page table is a data structure that stores frame numbers and the indexes represent page numbers




☆ Each entry of page table contains frame number and some extra bits.

No. of entries = no. of pages.

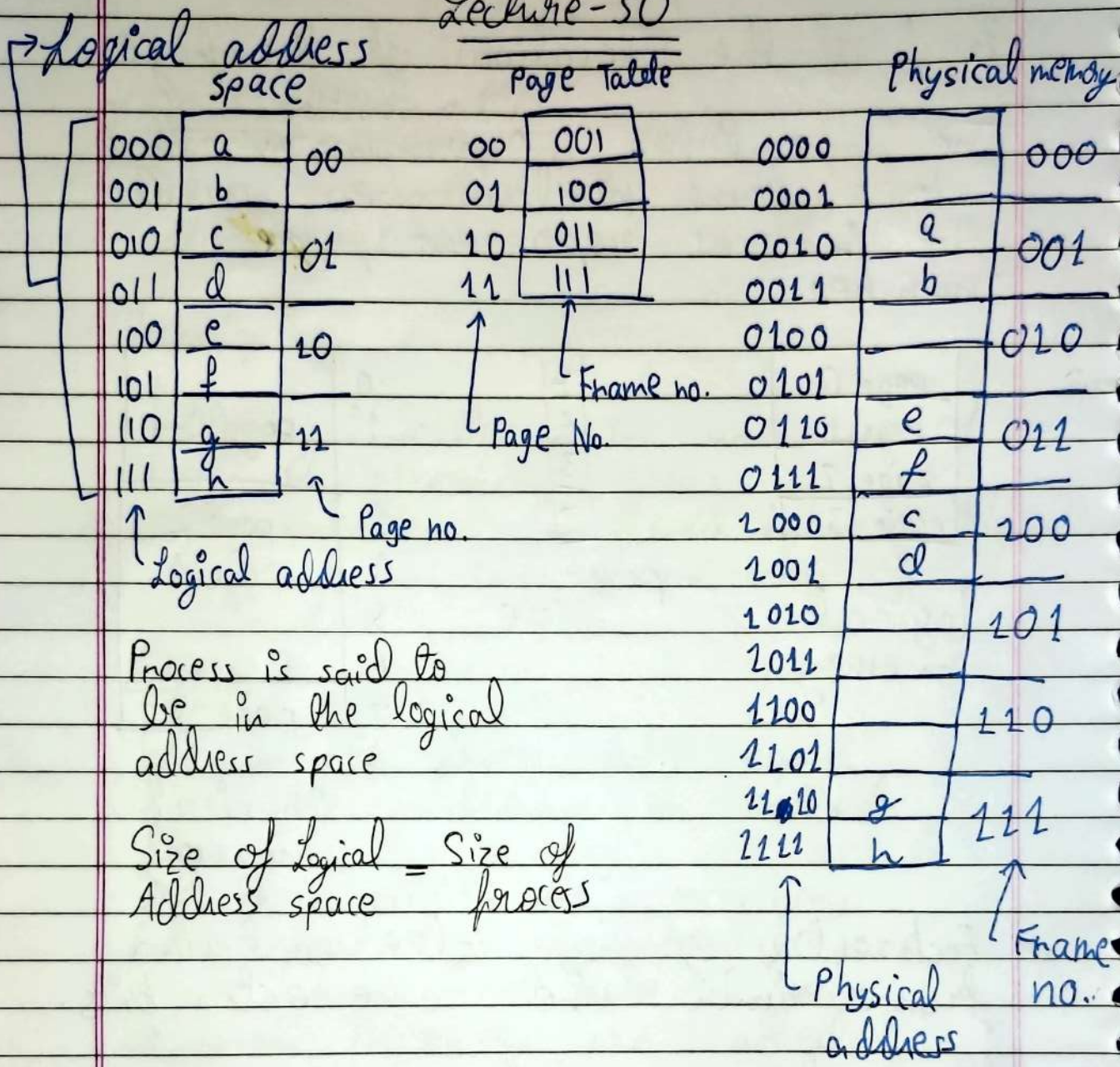
☆ Each process has its own page table

Page Table Size = no. of entries * 1 entry size

So to get some content in this kind of system main memory will be accessed twice

i) First for page table ii) Second for the page

# Lecture-30




Process is said to be in the logical address space

Size of Logical Address space = Size of process

The CPU / process only generate and work on the logical addresses.

Logical Address ⟶ | Page no. | Byte no. |

If memory byte addressable

Bits for (page no.) = $\log_2$ (no. of pages)

Bits for (byte no.) (line no.) = $\log_2$ (page size in byte)

Collection of logical address is called logical address space (LAS).

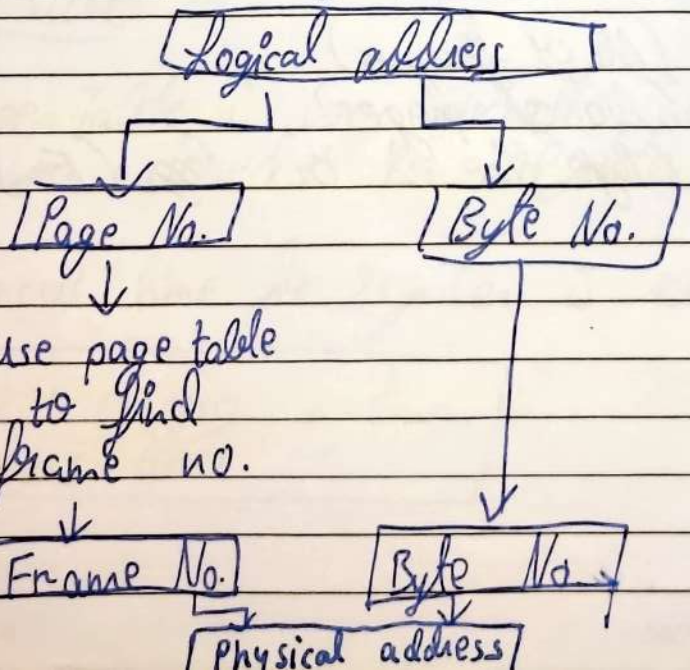
Collection of physical addresses is similarly Physical Address Space (PAS).

Physical Address → Frame Number | Byte No.

Byte no. size in physical address = Byte no. size in logical address.

Bits for frame no. = $\log_2$ (No. of frames)

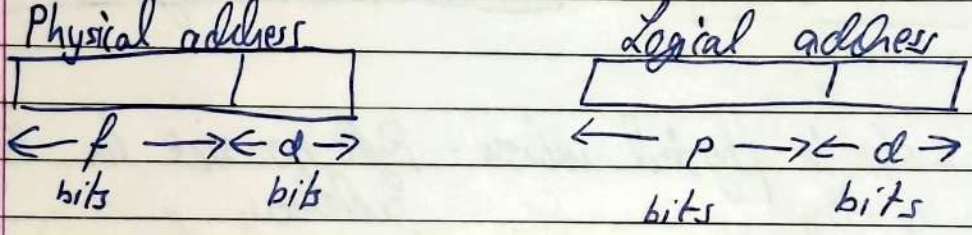
Logical address to physical address



Since Page Table is also stored in the main memory, we need a way to store its location.

We use a special register called Page Table Base Register (PTBR) which holds starting physical address of page table.

⇒ Page Table is not stored in a contiguous way. (if page size < page table size we use multilevel paging)




d bits ⇒ displacement bits; f bits ⇒ frame no. bits
p bits ⇒ page no. bits

$f = \log_2$ (No. of frames)
$p = \log_2$ (No. of pages)
$d = \log_2$ (Page size) or $\log_2$ (Frame Size)

Lecture-31 is questions of paging

Date / /
Page

## Lecture-32

### Time Required in Paging

i) Translate logical addr. to physical addr.
ii) Access main memory for data.

Since the most amount of time is needed to access page table from main memory.

Effective memory access time = Page table access time + content access time from main memory
(Total access time)

If Page Table is stored in Main Memory (By default)

Effective memory access time = 2 × Main Memory Access time = $2t_{mm}$

### Special Case

If Page Table is very small, it is sometimes stored in registers.

Since access time for register is ~~zero~~ negligible

Effective memory access time = $t_{mm}$

So if we are using paging, the data access time is increased by 2 times.

## Performance Improvement in Paging

TLB is used to improve performance of paging.

TLB ⇒ Translation Lookaside Buffer

TLB is a memory hardware that is used to reduce the time taken to acces a memory location.

It stores frequently accesed page table entries so that CPU can get physical address without having to access page table itself. from main memory.

The concept is similar to cache memory.

So first CPU seaches for entry in TLB, if it can't find in TLB then only it accesses the page table.

Similar to cache memory the ratio for which we get TLB hit of all access is TLB Hit Ratio and the ratio of what we miss to all access is TLB Miss Ratio.

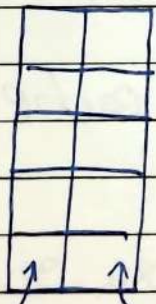
And similar to cache,

$$T_{effective} = \text{Hit ratio} \times (t_{TLB} + t_{mm}) + \text{Miss ratio} \times (t_{TLB} + 2 \times t_{mm})$$

If we further solve the above by putting
Hit ratio = 1 - Miss ratio.

$$T_{effective} = t_{TLB} + t_{mm} + \text{Miss ratio} \times t_{mm}$$

TLB is searched using page number in logical address.



# Lecture - 33

## TLB Mapping

It is the pattern by which we bring entries from page table to TLB

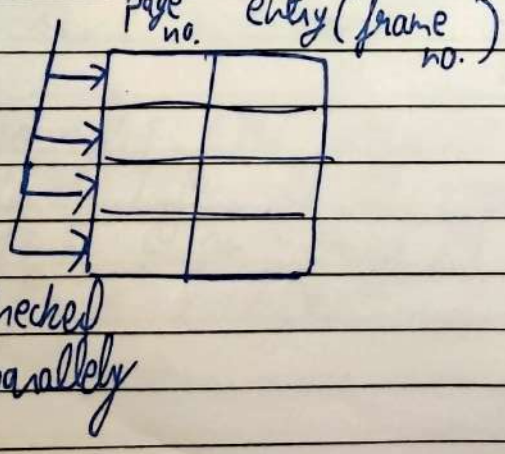
Types

- Direct Mapping
- Set Associative Mapping
- Fully Associative Mapping

## Fully Associative Mapping

It uses content addressable memory. So the page numbers are stored in form of content which are checked parallely.

Similar to Fully Associative cache mapping in concept.



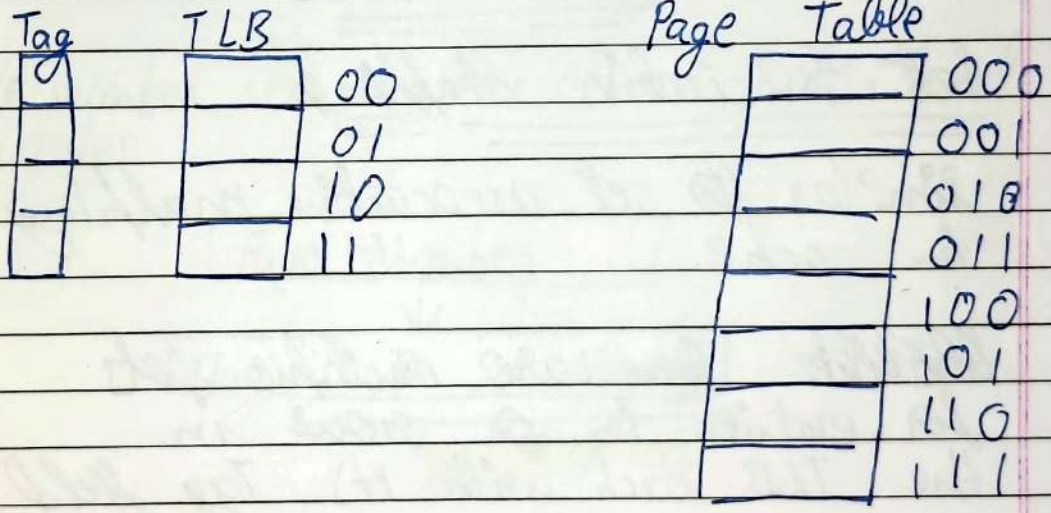
# Direct Mapping

Unlike in cache mapping, we map single entries, rather than blocks of memory. Therefore in direct mapping for paging,

Page Number

| Tag | TLB No. |
|---|---|

Tag identifies which of the possible entries is currently mapped to the TLB. For Eg,

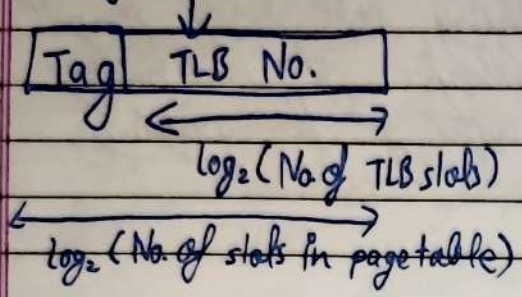



If we use mapping 010

0 is Tag bit & 10 is TLB No.

TLB No. Bits = $\log_2$ (No. of entries TLB can store)
Page No. Bits = $\log_2$ (No. of entries Page Table can store)

Tag Bits = $\log_2$ (No. of possible entries which can be mapped from Page Table)




The collection of all Tags in TLB is called Tag directory. Similar to Tag directory in cache.

## Lecture-34

## Set Associative Mapping

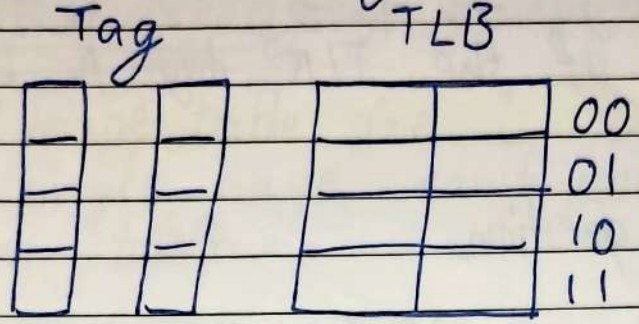
Similar to set associative mapping in cache.

There are multiple slots for entries in a row in the TLB each with it's Tag field

The slots in a single row are collectively called a set.

The number of slots in each set is called the associativity.

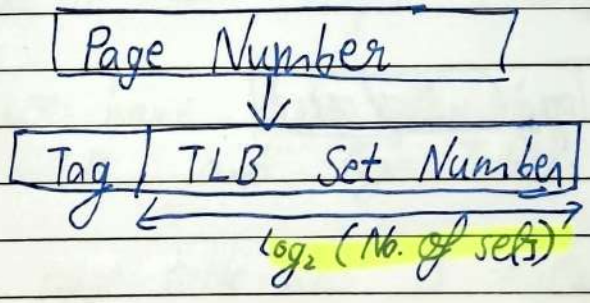
So for two way set associanity



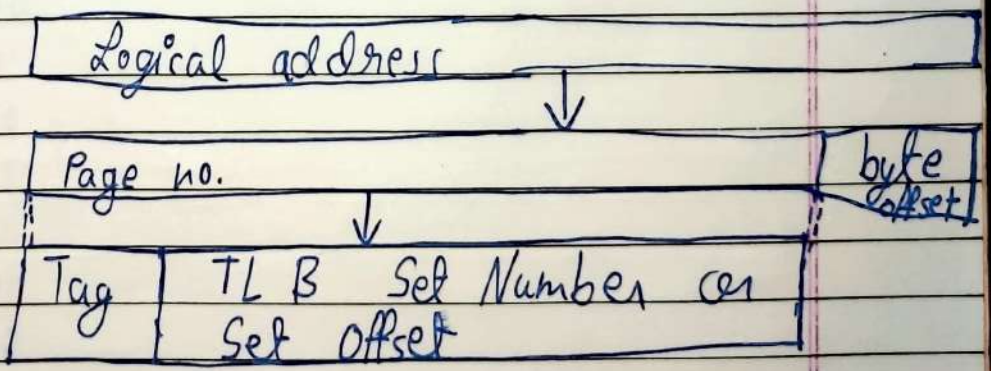

Associanity is 2, so 2 columns per row in TLB.

So 2 entries can be stored at once unlike in direct mapping.

⇒ We number sets rather than individual entries




So,



Eg, A computer has 40 bit virtual address, page size is 8 KB and a 128 entry size TLB organized into 32 sets. The length of the TLB tag in bits is ?

To solve this question,

Virtual address

| page no. | byte offset |
|---|---|

40

byte offset = $\log_2$ (page size)

(in bits) = $\log_2$ ($2^{13}$ B)

= 13

Page no.

| Tag | set offset |
|---|---|

27

Set offset = $\log_2$ (no. of sets)

= 5

Tag = 27 - 5

= 22 bits

# Lecture-35

## Multilevel Paging

If page table size is > page size.
A single page can't contain the page table.

Hence, the page table is also distributed across memory in pages.

This is when we use multilevel paging.

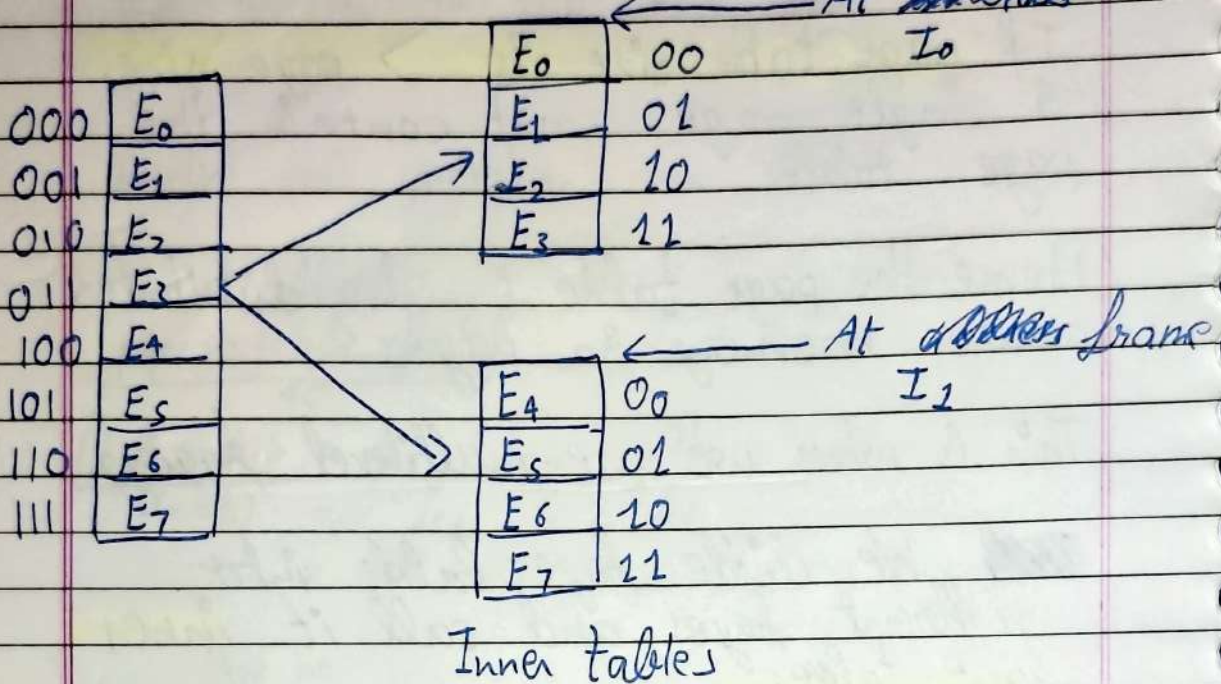
We divide page table into different pages and call it inner page tables

Let's take a simple example,

Suppose we have page size 4B, entry size is 1B and Page Table size is 8B.
So our page table has 8 entries

| | |
|---|---|
| $E_0$ | 000 |
| $E_1$ | 001 |
| $E_2$ | 010 |
| $E_3$ | 011 |
| $E_4$ | 100 |
| $E_5$ | 101 |
| $E_6$ | 110 |
| $E_7$ | 111 |

We divide our page table to page sizes and call the acquired tables inner table

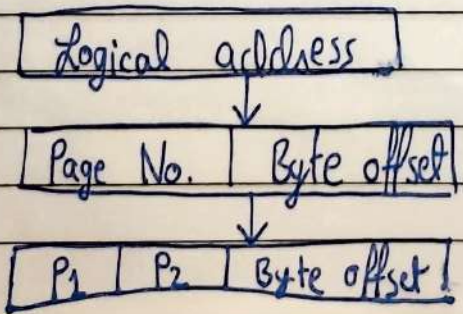



Inner tables

Now we use another page table to track these inner tables in memory called outer page table. It has address (frame no.) to inner tables
For our example

| | |
|---|---|
| $I_0$ | 0 |
| $I_1$ | 1 |

Now when we get a logical address



$P_1 \Rightarrow$ Outer Page Table Entry number
$P_2 \Rightarrow$ Inner Page Table Entry number

Length of $P_1$ in bits = $\log_2$ (No. of inner tables)
Length of $P_2$ in bits = $\log_2$ (No. of entries per inner table)

The PTBR stores starting physical address of outer table.

## Multiple levels

Suppose we have a scenario,

No. of pages = $2^{20}$
Page size = 2kB = $2^{11}$ B
Entry size = 4B = $2^2$ B

Logical address

| Page no. | Byte offset |
|---|---|
| ← 20 → | ← 11 → |

No of entries in one page = $\frac{2kB}{4B} = 2^9$

So having only 2 levels is not enough since max length is 9 for page table entry length. So we have 3 levels.

| $P_1$ | $P_2$ | $P_3$ | Byte Off. |
|---|---|---|---|
| 2 | 9 | 9 | 11 |

Hi, Dr

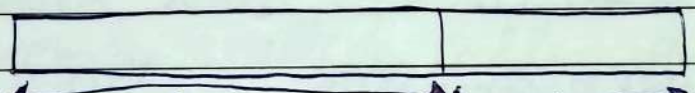
For a given logical address, in multilevel paging or paging in general.

1) Find number of page table entries that a single page had hold. Let this number be $\alpha$.

$$\alpha = \log\left(\frac{\text{Size of a page}}{\text{Size of a page table entry}}\right)$$

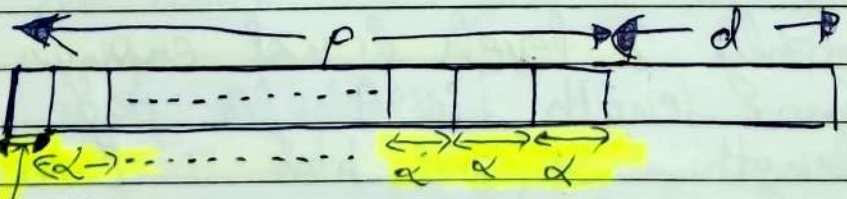
2) Divide the logical address in 2 parts

Logical Address



$\log_2$(no. of pages) = $p$ | $\log_2$(Page size) = $d$

3) Now further divide $p$ by $\alpha$ from right to left




remaining bits

The number of partitions of p is the level of multi paging.

The memory access time now becomes,

$$T_{effective} = \text{Hit ratio} \times (t_{TLB} + t_{MM}) + \text{Miss ratio} \times \left(t_{TLB} + (\text{level of paging} + 1) \times t_{MM}\right)$$