

Lecture 1

Operating System

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

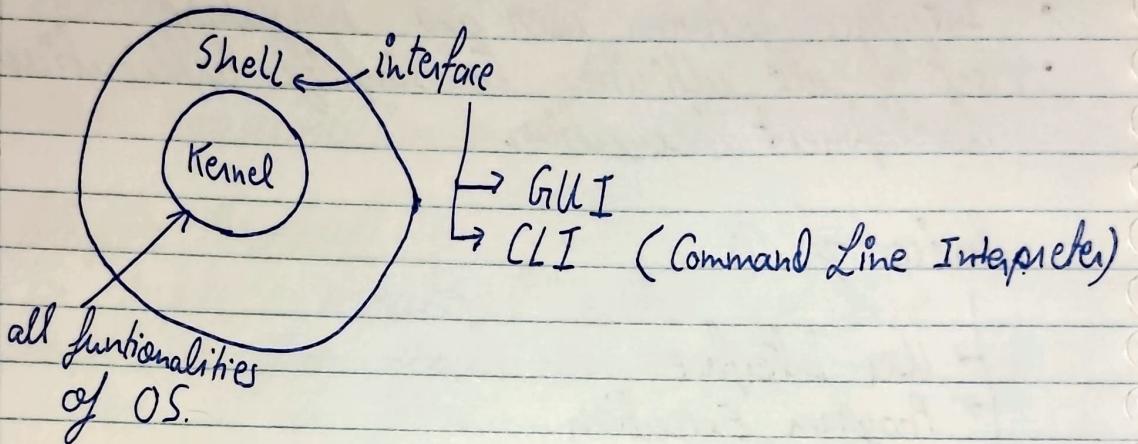
Services of OS

- 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 besides itself.

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

- So not good CPU ~~utilization~~ utilization.

Multi-programming OS

OS allows multiple processes to reside in main memory.

- Better CPU utilization than uni-programming
- Degree of multiprogramming = $\frac{\text{no. of running programs in memory}}{\text{except the OS}}$

→ 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 terminates

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.

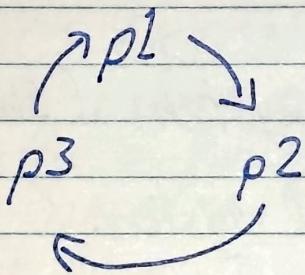
Uniprogramming OS cannot be preemptive.

Multi-tasking OS

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

Processes are run ~~in a queue~~ for a
limited time on CPU before they
are briefly taken out and
replaced with another process.

So if we have three processes
 p_1 , p_2 and p_3



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

Since CPU is very fast to use
it appears that processes
are in parallel.

This is called time sharing OS as
well.

Multi-User OS

This OS allows multiple user to access single system simultaneously.

Eg → Linux / BSD.

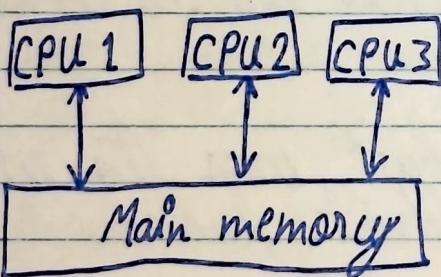
Multi-Processing OS

An OS used in systems with multiple CPUs.

Types

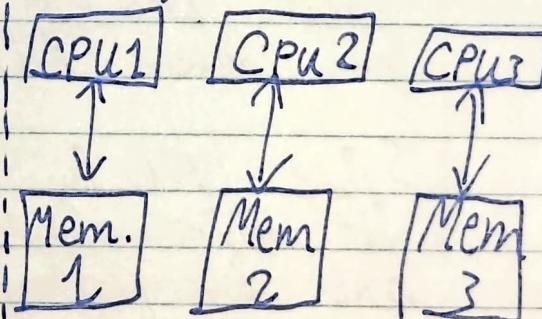
- Tightly coupled system (Shared memory system)
- Loosely coupled system (Distributed system)

Tightly coupled



All CPUs use same memory

Loosely coupled



Every CPU has separate 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 bridge).
- 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

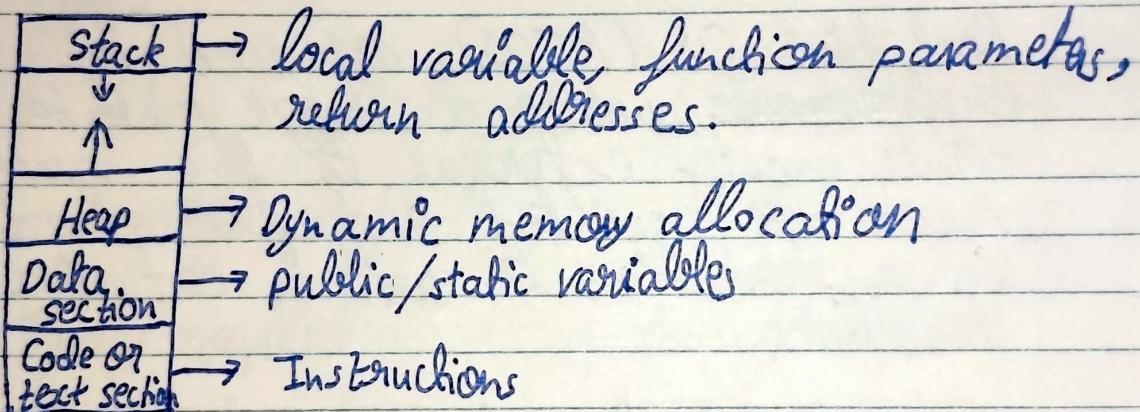
Process

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 is a locus of control.

Representation of a process



Every process has similar structure in main memory.

Operations on a process

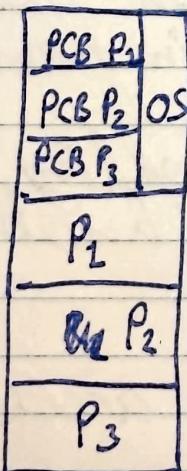
- Create (Resource allocation)
- Schedule, Run
- Wait / Block
- Suspend / Resume
- Terminate (Resource deallocation).

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 all 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 ~~various~~ 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.

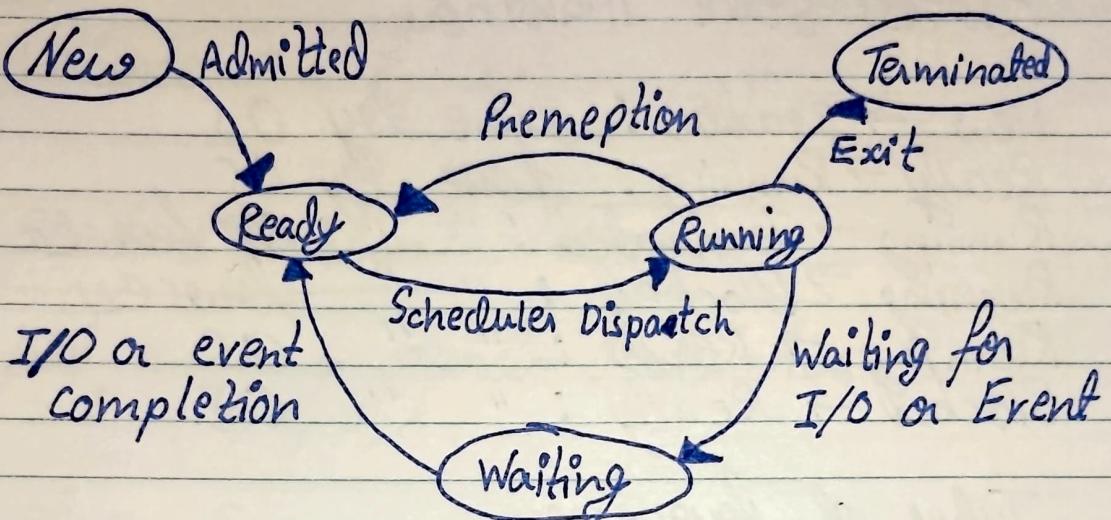
So context switching is stopping a process and running another process.

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 happens 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 : " " ~~by process~~

Running → Blocked : " " ~~processes~~

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 is 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.
- ⇒ Queues store PCBs of process.

Types of Schedulers

- └ Long term scheduler
- └ 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 called swapping and is done by mid term scheduler.

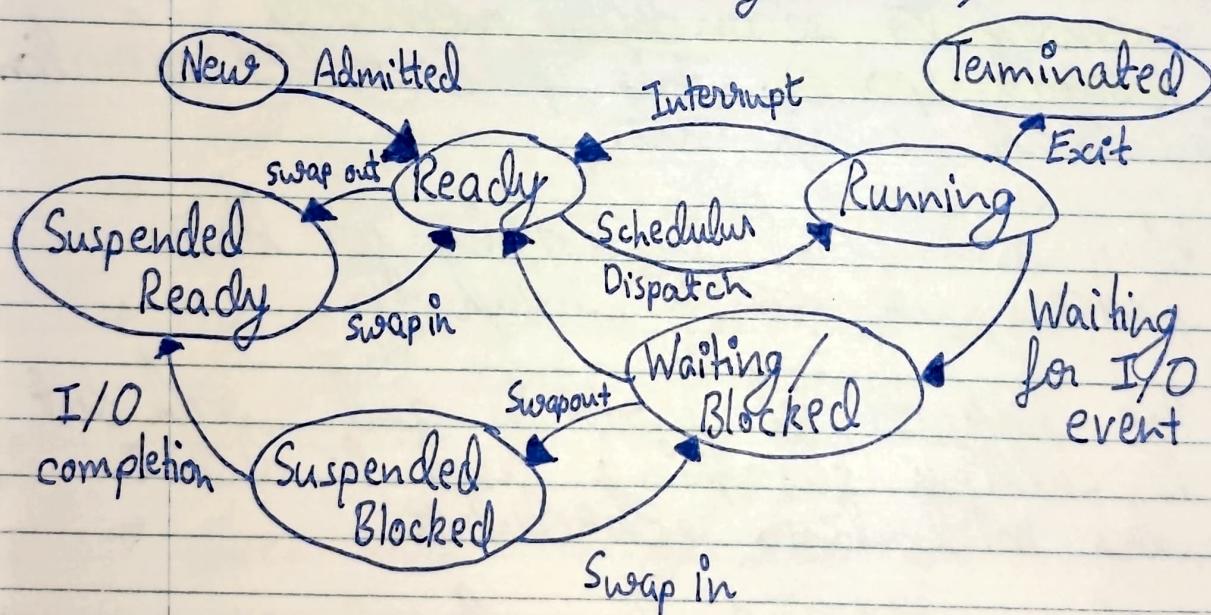
If swapping is done based on priority, it is called polling. So swap in/out is also called poll in/out.

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,



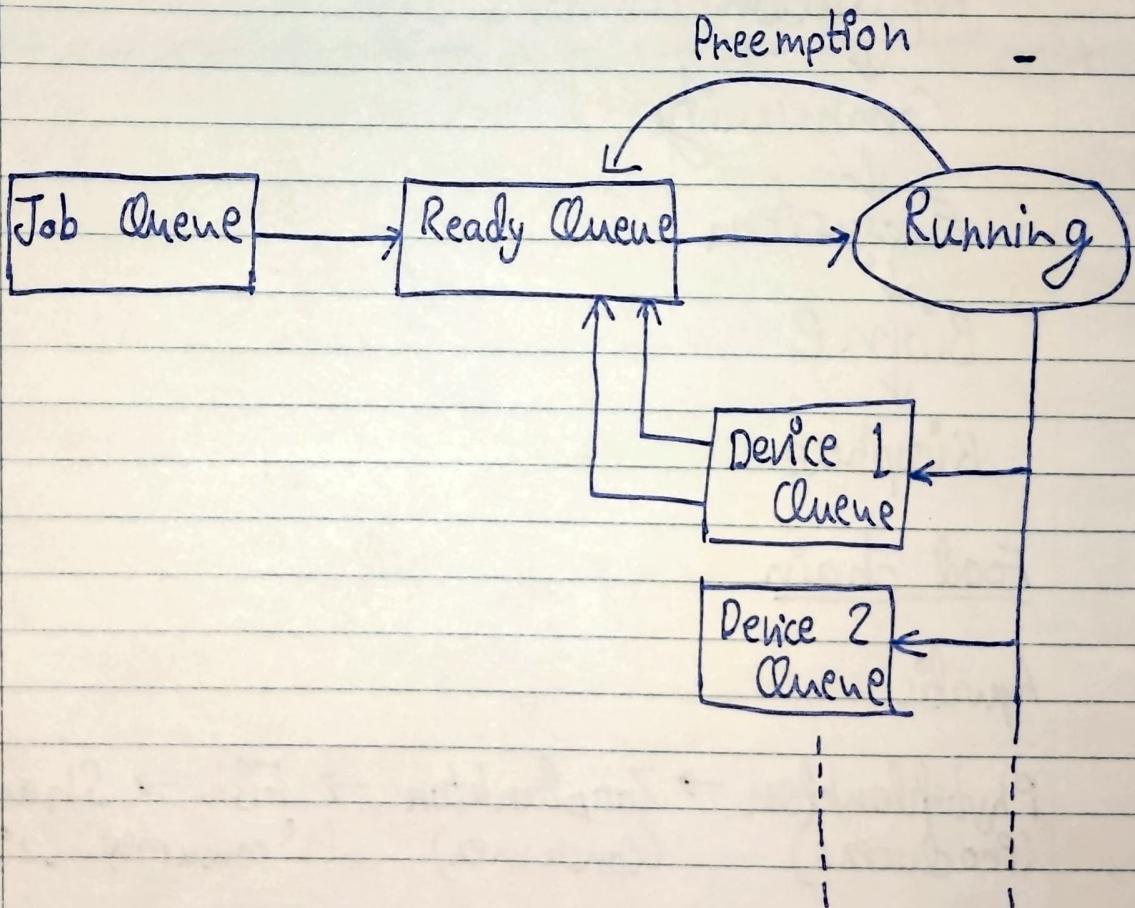
- ⇒ Long term scheduler controls max degree of multiprogramming and can ~~not~~ increase it.
- ⇒ Mid term scheduler can decrease degree of multiprogramming.

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 \Rightarrow Time at which process arrives in the system.

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

CT \Rightarrow The time at which process completes the execution.

TAT \Rightarrow Amount of time from arrival to completion.

$$\text{TAT} = \text{CT} - \text{AT}$$

WT \Rightarrow Total time spent waiting in the ready queue

$$\text{WT} = \text{TAT} - \text{BT}$$

Other scheduling Times

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

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

Scheduling Length \Rightarrow Time taken for scheduling the process.

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

Throughput \Rightarrow Number of processes executed per unit of time.

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

CPU Scheduling

- Preemptive
- Non-Preemptive

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

First Come First Serve

Criteria \Rightarrow Arrival Time (AT)

- Tie-breaker : smaller process id first.

Type \Rightarrow Non-preemptive.

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

Eg, for FCFS

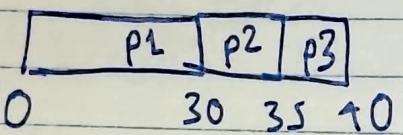
Given in Ques

To be calculated

Process	Arrival Time	Burst Time	CT	TAT	WT
p ₁	0	30	30	30	0
p ₂	0	5	35	35	30
p ₃	0	5	40	45	35

Work

i) Drawing Gantt chart



ii) Use Gantt chart to get end 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

$$\begin{aligned} L &= \max(CT) - \min(AT) \\ &= 40 - 0 \\ &= 40 \end{aligned}$$

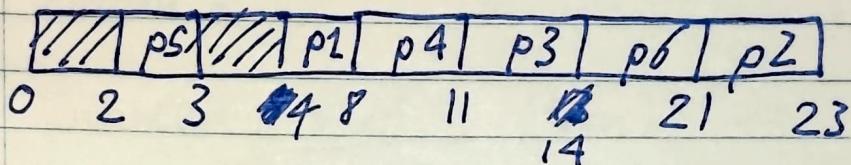
$$vii) \text{Throughput} = \frac{n}{L} = \frac{3}{40}$$

Note : If we have multiple processes in FCFS and to choose from in FCFS, we always go for smaller 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	WF
p1	4	4	8	4	
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)

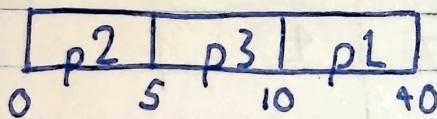
Given in Ques

To be calculated

Eg,

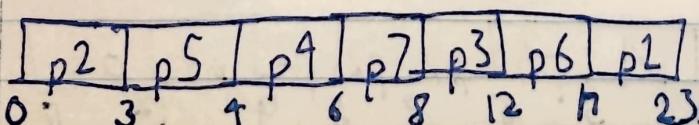
Process	AT	BT	CT	TAT	WT
p1	0	30	40	40	10
p2	0	5	5	5	0
p3	0	5	10	10	5

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	2	4	1	0
p6	4	5	17	13	8
p7	6	2	8	2	0



Shortest Remaining Time First

This is sometimes also referred to as preemptive SJF.

We compare ~~the~~ 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-BT =
				TAT	WT
p ₁	0	6	17	17	11
p ₂	1	4	9	8	4
p ₃	2	1	3	1	0
p ₄	3	2	5	2	0
p ₅	4	1	6	2	1
p ₆	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

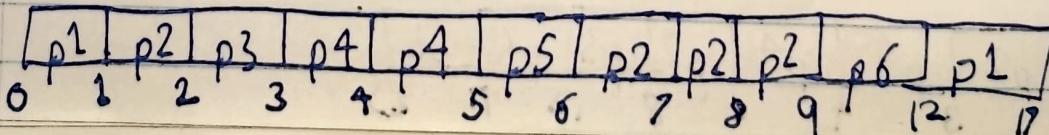
p₂ - 4 - 3 - 2 - 1 - 0 X

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 time of measurement called response time.

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

So for the table in preemptive algo, 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 does not have same process in adjacent blocks

In eos Gantt chart:

No of context = No. of processes blocks - 1
switches

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 :** ^{Highest} Response Ratio
 - Tie-breaker : Burst Time
- **Type :** Non-preemptive

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

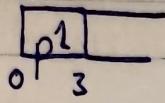
W = Wait Time till that time

S = Service / Burst Time

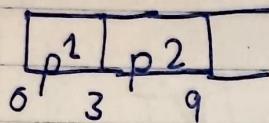
Example,

Process	Arrival Time	BT
p ¹	0	3
p ²	2	6
p ³	4	4
p ⁴	6	5
p ⁵	8	2

At time 0,



At time 3,



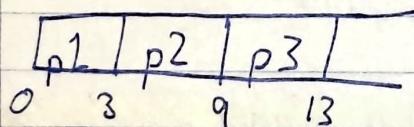
At time 9,

We have p3, p4, p5

$$RR(p_3) = \frac{W+S}{S} = \frac{5+4}{4} = 2.25$$

$$RR(p_4) = \frac{W+S}{S} = \frac{3+5}{5} = 1.3$$

$$RR(p_5) = \cancel{\frac{W+S}{S}} = \frac{1+2}{2} = 1.5$$

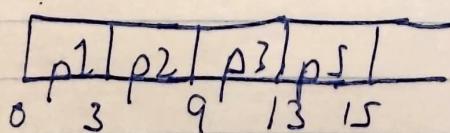


At time 13,

We have p4, p5

$$RR(p_4) = \frac{7+5}{5} = 2.4$$

$$RR(p_5) = \cancel{\frac{5+2}{2}} = 3.5$$



p₄ is the last process,

p ₁	p ₂	p ₃	p ₅	p ₄
0	3	9	13	15

So the table

Process	AT	BT	CT	TAT	WT
p ₁	0	3	3	3	0
p ₂	2	6	9	7	1
p ₃	4	4	13	9	5
p ₄	6	5	20	14	8
p ₅	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

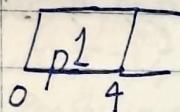
- Static → fixed (default)
- Dynamic → may increase or decrease

Eg. (for non-preemptive)

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

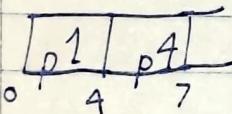
Tip: Draw a ready queue for easier tracking of processes

At time 0,



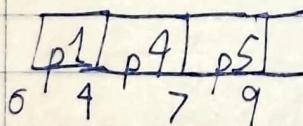
At time 4,

Ready Queue $\Rightarrow p_5, p_4, p_3, p_2$

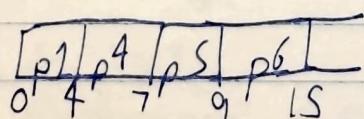


At time 7,

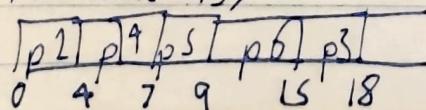
Ready Queue $\Rightarrow p_5, p_3, p_2, p_6$



At time 9,



At time 15,



dat process.

	p1	p4	p5	p6	p3	p2
	0	4	7	9	15	18
						20

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	13	10	4

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

Correct a gantt chart.

	p1	p4	p5	p6	p3	p2
	0	4	5	7	13	16
						18

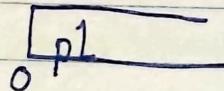
Process	AT	BT	CT	TAT	WT
p1	0	4	4	4	0
p2	1	2	18	17	15
p3	2	3	16	14	11
p4	3	1	5	2	2
p5	4	2	7	3	1
p6	5	6	13	8	2

Eg. for preemptive

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

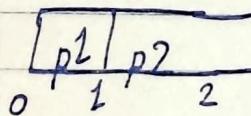
At t=0

Ready queue ; p1



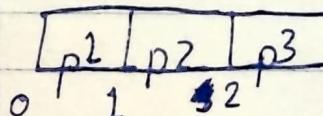
At t=1

Ready queue = p1, p2



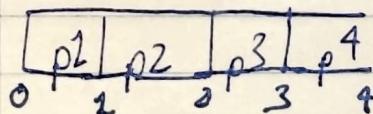
At t=2

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 - 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

p_1	p_2	p_3	p_4	p_5
0	1	2	3	9

At $t=5$

Ready queue; p_1, p_2, p_3, p_5, p_6

p_1	p_2	p_3	p_4	p_5
0	1	2	3	4

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

p_1	p_2	p_3	p_4	p_5	p_6
0	1	2	3	4	6

At $t=12$,

p_1	p_2	p_3	p_4	p_5	p_6	p_3
0	1	2	3	9	6	12

At $t=14$,

p_1	p_2	p_3	p_4	p_5	p_6	p_3	p_2
0	1	2	3	4	6	12	14

At $t=15$,

p_1	p_2	p_3	p_4	p_5	p_6	p_3	p_2	p_1
0	1	2	3	4	6	12	14	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	9	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 ~~the~~ 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 : $AT + Q$ ($Q := \text{Time Quantum}$)
 - Tie-breaker : Smaller PID
- Type : Preemptive.

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

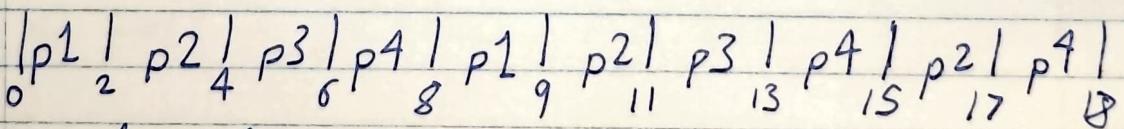
Eg,

Process	AT	BT
p1	0	3
p2	0	6
p3	0	4
p4	0	5
p1		$Q = 2$

BT Tracker:

p1 3 - 1 - X
 p2 6 - 4 - 2 - X
 p3 9 - 2 - X
 p4 5 - 3 - 1 - X

Gantt Chart



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

Table,

	AT	BT	CT	TAT	WT	RT
p ¹	0	3	9	9	6	0
p ²	0	6	17	17	11	2
p ³	0	4	13	13	9	4
p ⁴	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
p ¹	0	4
p ²	1	5
p ³	2	6
p ⁴	3	3
p ⁵	4	2
p ⁶	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 the During the cycle the process at start of ready queue will run.

<u>At start of cycle no:</u>	<u>Ready Queue</u>
0	p1
2	p2 p3 p1
4	p1 p2 p3 p1 p4 p5 p2
6	p1 p4 p5 p2 p6 p3

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

Burst time block

p2 4 - 2 - X

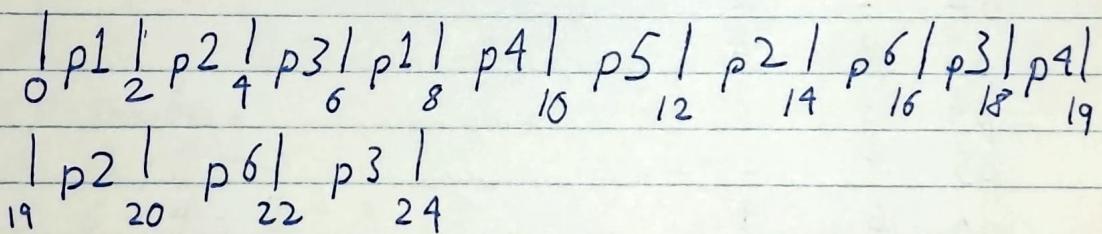
p2 5 - 3 - 1 - X

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

p4 3 - ~~1~~ 1 - X

p5 2 - X

p6 4 - 2 - X



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 large Q, will make system less interactive.

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

Advantages of Round Robin

- F Farness
- E Interactivity for time-sharing system
- E 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.

$$\text{CPU utilization} = 1$$

If there are I/O operations.

n := No. of processes running

p := average percentage of time processes are waiting for I/O

$$\text{CPU utilization} = 1 - p^n$$

Lecture 10 is over on scheduling

Lecture 11

Multi-level 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

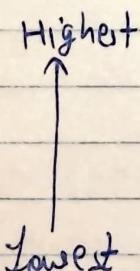
Background Processes Queue

Scheduling Among Queues

- Fixed Priority preemptive scheduling method
Time slicing

Fixed Priority

From our above queues let's consider priority as



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 PU. 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

Multi-level Feedback Queue (MLFQ) Scheduling

To avoid starvation and make multi-level 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
- * Convey Effect

SJF

Advantage

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

- Disadvantages
- * No practical implementation (BT cannot be pre-determined).
 - * No preemption.
 - * Longer processes 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 interactivity.
- * BT not required (Practical algorithm).

Disadvantages

- * Higher average waiting time and TAT.
- * Can degrade to FCFS. (& 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

code	data	files
registers		stack
thread → {		

single threaded process

code	data	files
registers	registers	registers
stack	stack	stack
s	s	s

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

Multithreading in user process.

Created without kernel intervention

Context switch is very fast.

If one thread is blocked, OS blocks entire OS

Generic and can run on any OS.

Faster to create and manage

Kernel Thread

Multithreading in kernel process.

Kernel itself is multithreaded

Context switch is slow.

Individual thread can be blocked

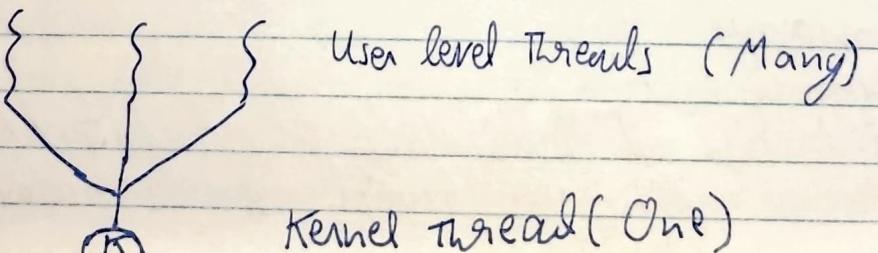
OS specific

slow to create and manage

Multi-threading Models

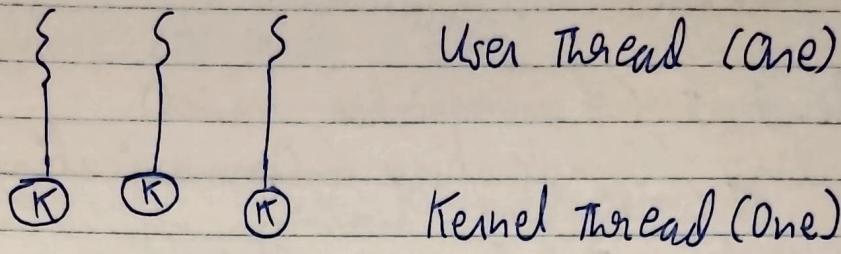
(Based on how user level thread can communicate with kernel thread)

1. Many-to-one



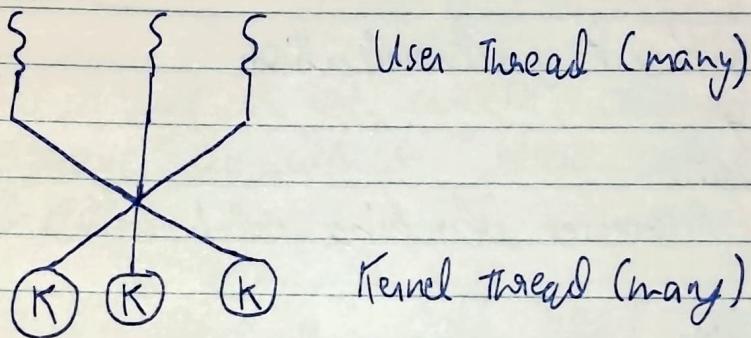
Multiple user threads can communicate with a single kernel thread

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 \Rightarrow 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 referred 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 code

$a = 5$ shared resource

P1

```
R1 ← a  
R1 ← R1 + 2  
a ← R1  
⋮
```

P2

```
R3 ← a  
R3 ← R3 + 5  
a ← R3  
⋮
```

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.

- Requirements of Critical section problem solution
 - i • Mutual Exclusion \Rightarrow If one process is using critical section, no other process can use that critical section.
 - ii • Progress \Rightarrow If no process in critical section and at least one process wants to enter critical section, then it should be allowed.
 - iii • Bounded Waiting \Rightarrow 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

We divide it as,

- * Entry section
- * Critical section
- * Exit section

~~Entered~~ or
endless
waiting

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

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

}

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

P1

while(true) {
 while(lock);
 lock = true;
 // CS
 lock = false;
 // RS

3

P2

while(true) {
 while(lock);
 lock = false;
 // CS
 lock = true;
 // RS

3

unlock execution
for P2 (or any other
waiting process)

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

It does satisfy progress.

Solution 2 : Using Turn
int turn = 1;

P1

while(true) {

 while(turn ~~!=~~ != 1);

 // CS

 turn = 2;

 // RS

}

P2

while(true) {

 while(turn != 2);

 // CS

 turn = 1;

 // 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 P_i and P_j , that communicate with each other,

P_i^o :

while(true) {

flag[i] = true;

turn = j;

while(flag[j] && turn == j); } } Entry section

// CS

flag[i] = false; } Exit Section.

// RS

3

And

P_j^o :

while(true) {

flag[j] = true;

turn = i;

while(flag[i] && turn == i); } } Entry section

// CS

flag[j] = false; } } Exit Section.

// RS

3

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 ~~a~~ synchronization hardware for hardware solution of critical section problem.

Hardware solution ~~pro~~ 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.



1 1 2020
6369

Date / /

Page



bool TestAndSet (bool *tbg) {

* bool r = * tbg;
* tbg = true;
return r;

2

This is not a software implementation, but a visualization of process

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

So in our process,

while (true) {

 while (TestAndSet (&Lock));

 //CS

 Lock = False;

3

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 levels.

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;

3

and our process,

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

3

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 \Rightarrow mainly used for solution to critical access problems with multiple processes

Counting Semaphore \Rightarrow 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
```

}

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

Both $\text{wait}()$ and $\text{signal}()$ are atomic.
Therefore, they run without being interrupted.

$\text{wait}()$:

stop process if $S \leq 0$
decrement S

$\text{signal}()$:

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. " " "
6. " " " starvation priority
7. inversion.
Semaphore can have busy waiting

Priority inversion : A low priority process may 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

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 whose 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

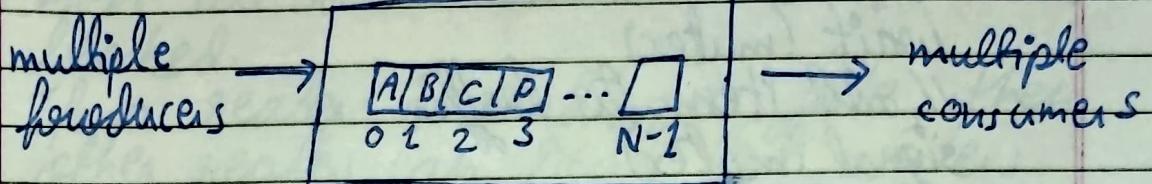
P_1	P_2
$R_1 \leftarrow x$	$R_2 \leftarrow x$
$\cancel{x} R_1 \leftarrow R_1 + 2$	$R_2 \leftarrow R_2 \cancel{-} 2$
$x \leftarrow R_1$	$x \leftarrow R_2$

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

Bounded Buffer with capacity N



Producers 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:

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



Producer () :

Producer () {

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

2

Consumer () :

Consumer () {

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

3

Initialization

mutex = 1

full = 0 // no slots occupied

empty = number of slots // all slots empty



Lecture -19/20

Reader-Writer 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)

Reader () :

reader() {

~~wait (mutex);~~

readcount ++;

if (readcount == 1)

~~wait (wert);~~

~~signal (mutex);~~

~~// perform read operation~~

~~readcount --;~~

~~if (readcount == 0)~~

~~signal (wert);~~

wait (mutex);

readcount --;

if (readcount == 0)

signal (wert);

signal (mutex);

3

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

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

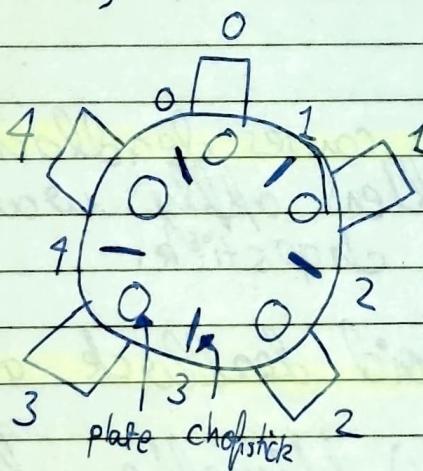
Once no more readers are in execution,
we signal next to allow writer
processes.

Dining Philosophers Problem

Suppose there are 5 philosophers, each
~~they can do~~ 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 i th philosopher will
use ~~odd~~ chopsticks (i and $(i+1) \% n$).

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

Solution

We will use binary semaphores in an array to solve this problem.
So for our example,

Binary semaphore array $\text{chopstick}[5] = \{1, 1, 1, 1, 1\}$

And our solution is, \uparrow
(Initial)

4

```

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 1 000
0000

Date / /
Page



I will run \Rightarrow

wait(chopstick[(i+1)%n])

wait(chopstick[i])

// eat

signal(chopstick[(i+1)%n])

signal(chopstick[i])

others will run

; wait(chopstick[i]);

; wait(chopstick[(i+1)%n]);

// eat

; signal(chopstick[(i+1)%n]);

; 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 \Rightarrow if one process is using resource, another cannot.
2. Hold & Wait \Rightarrow As seen in philosopher problem, then deadlock occurs if all processes

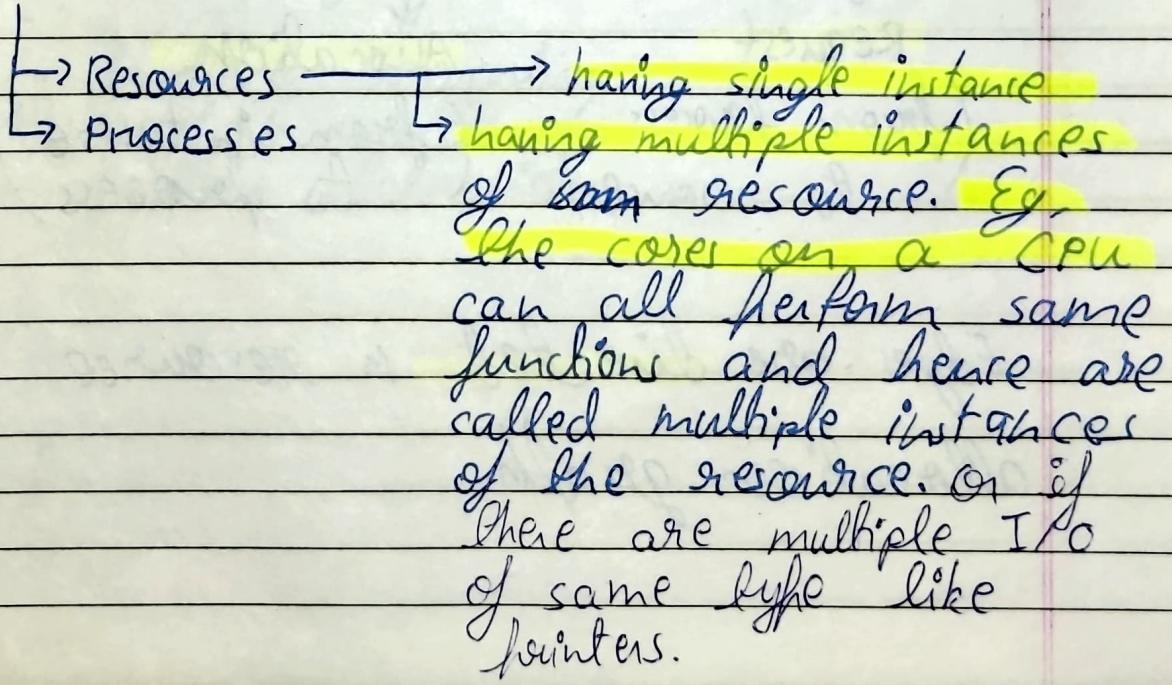
hold one resource and are in indefinite waiting for another.

3. **No-preemption** \Rightarrow 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** \Rightarrow similar to dining philosopher deadlock condition where all processes are waiting for resource another one is holding.

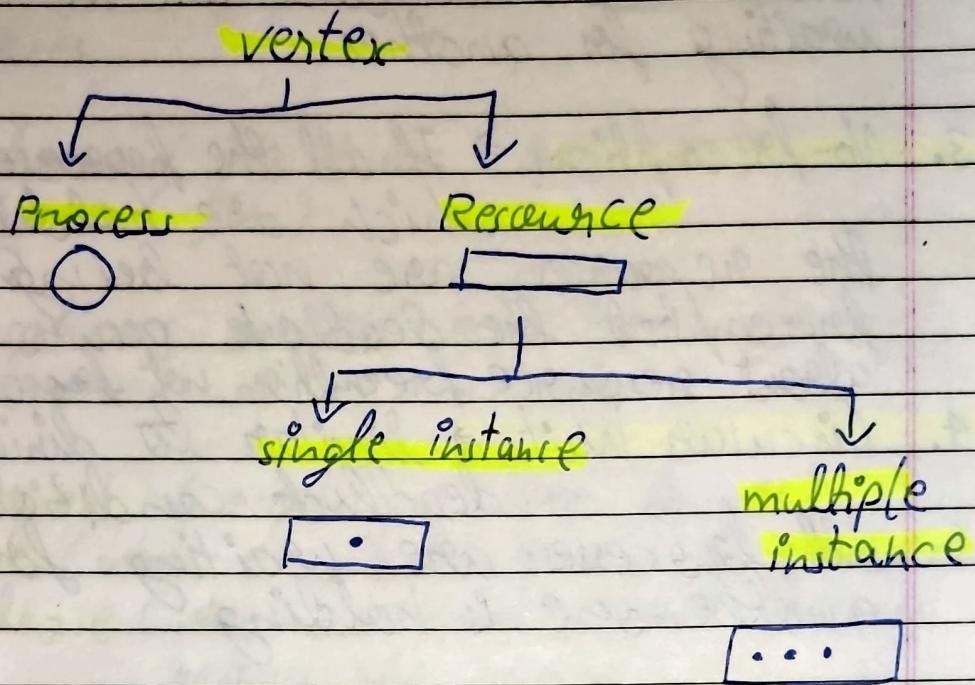
Resource Allocation Graph

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

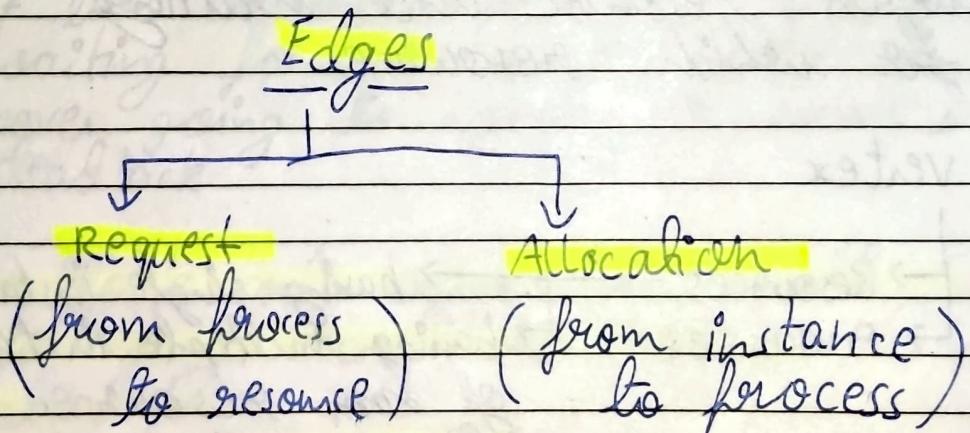
vertex



Symbols of vertices

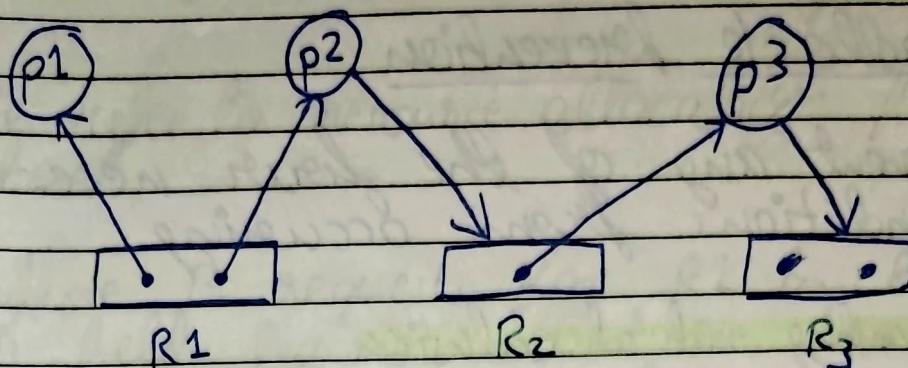


(dots represent number of instances)



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 portions

- deadlock prevention
- deadlock avoidance
- deadlock detection



Lecture - 22

Deadlock prevention

Prevent any of the four necessary conditions from occurring.

1. Preventing mutual exclusion

- Have enough resources for all processes
- Make all processes independent

Neither of above 2 are practical.

2. Preventing Hold & Wait

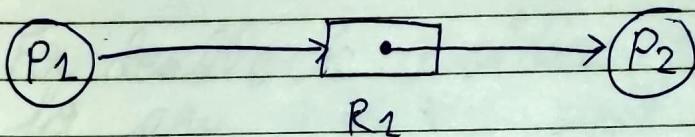
- If all required resources are available the acquire all or else wait for all. This method is prone to starvation.
- If process is trying to acquire a resource which is not available, while holding some other resources then process will release ~~all~~ the allocated process

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

3. Preventing No-preemption (Allowing preemption)

We refer to resource allocation preemption here.

Suppose P_1 requests for R_1 & R_1 is held by P_2 . But P_2 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.

Lecture - 23

Avoid Deadlock

The OS tries to keep system in safe state

~~Safe State~~ **Safe State** \Rightarrow Condition with no possibility of deadlock.

Unsafe State \Rightarrow 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 $m \times n$.

It stores which resources have 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 $m \times n$.

It stores the total instances of each resource needed by process to complete its 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



(ii) 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 structures.

(iii) Need matrix

Stores how many resources that need to be allocated now in order for process to complete execution.

$$\text{Need} = \text{Max} - \text{Allocation}$$

(iv) 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



11 AM

MARCH

Date / /

Page



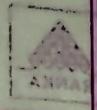
The Algorithm

Note: Sometimes we initially make an extra variable $\text{work} = \text{available}$; 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; $\text{Work} = \text{Available}$ (optional)
 $\text{Finish}[0 \dots m] = \text{False}$ and calculate Need
2. * Find a process; such that :
 - a) $\text{Finish}[i] = \text{false}$ and
 - b) $\text{Need}[i] \leq \text{Work}$
(Get i^{th} row)

If no such process, go to step 4.
3. For this process; found
 $\text{Work} = \text{Work} + \text{Allocation}[i]$
(Get i^{th} row)
 $\text{Finish}[i] = \text{true}$
go to step 2
4. If For all $\text{Finish}[i] = \text{True}$
then system in safe state
Else
System in unsafe state.

Flow : compare rows of need with available
 till finding a process whose all numbers are
 \leq to equivalent in available; and add
 that row of allocation to available.
 Repeat until all process finish



Eg, Suppose given

Process	Allocation				Marc				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	6
p ³	1	0	0	2
p ⁴	0	0	2	2
p ⁵	0	6	4	2

~~First after p¹~~
~~Available = 1 5 3 2~~

~~Second after p³~~

~~Available = 2 8 2 2~~

~~First after p¹~~

~~available + allocation [p¹]~~

~~available = 1 5 3 2~~

~~Second after p²~~

~~available = 2 8 3 2~~

First after p₂
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 \Rightarrow 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 ~~is not~~ sends a request.

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

The algorithm :

1. If $\text{Request}_{P_i} \leq \text{Need}_{P_i}$

Goto step 2

Else

Error since process has exceeded its maximum claim.

2. If $\text{Request}_{P_i} \leq \text{Available}$

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

$$\text{Available}_{\text{p}_i} = \text{Available} - \text{Request}_{\text{p}_i}$$

$$\text{Allocation}_{\text{p}_i} = \text{Allocation}_{\text{p}_i} + \text{Request}_{\text{p}_i}$$

$$\text{Need}_{\text{p}_i} = \text{Need}_{\text{p}_i} - \text{Request}_{\text{p}_i}$$

Then use safety algorithm to test if 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



Lecture - 25

Relation between no. of instances and deadlock

Suppose we have a condition

Processes	Max
p ₁	k
p ₂	k
p ₃	k
⋮	⋮
p _n	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
p ₁	k	k-1
p ₂	k	k-1
p ₃	k	k-1
⋮	⋮	⋮
p _n	k	k-1

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



Therefore,

For a resource with 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 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
p ₁	5
p ₂	4
p ₃	6
p ₄	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

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

$n \Rightarrow$ no. of processes

$k \Rightarrow$ maximum instances needed by each process

Lecture - 26

Deadlock detection & Recovery

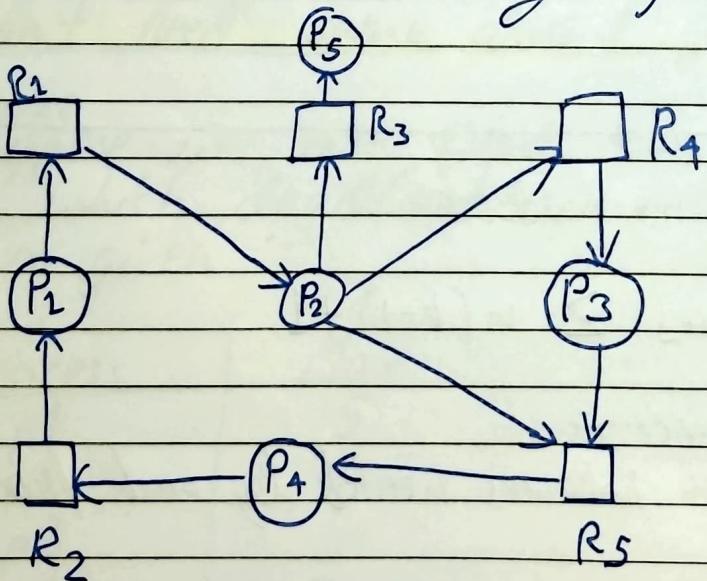
Deadlock detection :

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

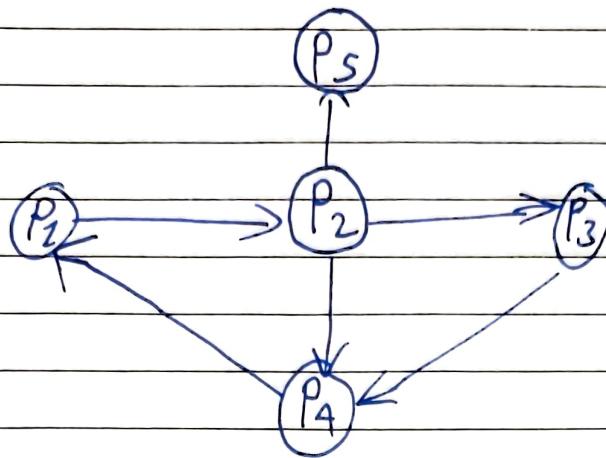
Wait For Graph

Created using resource allocation graph.

To make wait-for graph
Remove resources nodes
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 (indeterminate)

11

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 clue (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. Prevent resources from processes

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 progress is not 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 is what resource process is holding
- iv) How many more resources process needs to complete execution
- v) How many need to be terminated 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 predictably 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

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

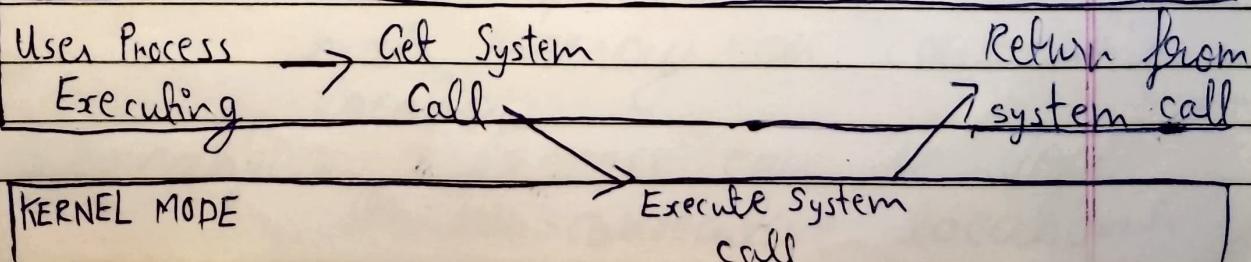
How system call works

- 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

USER MODE



KERNEL MODE

Execute System call



fork()

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

fork 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

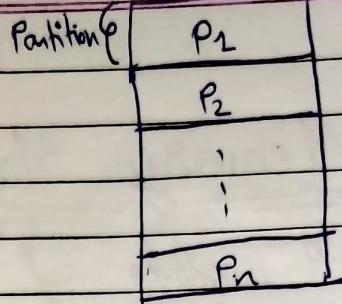
Contiguous \Rightarrow entire process stored in main memory on consecutive locations.

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



Contiguous

- ↳ Fixed partition
- ↳ Variable partition



Main memory

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 \Rightarrow Process is allocated the first empty partition in which it can fit. (Linear search for partition)
- ii) Best fit \Rightarrow 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 \Rightarrow The process is allocated to partition which is the largest in size.
- iv) Next fit \Rightarrow 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 favorable in fixed partition contiguous MMT.

Variable Partition Contiguous MMT

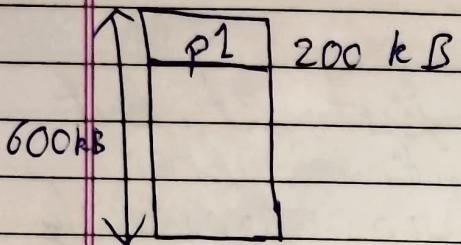
Main memory is not divided into partitions initially.

When a new process arrives, a new partition is created of same size as process size and process is allocated to this partition.

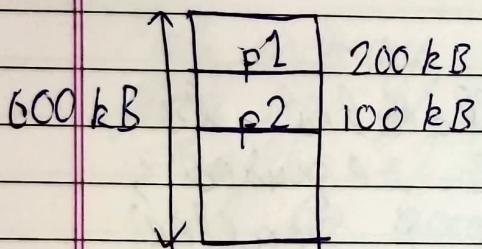
Hence no internal fragmentation.

Suppose we have a main memory of size 600 kB.

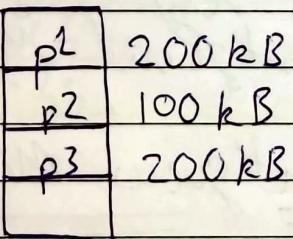
First we allocate a process of 200 kB



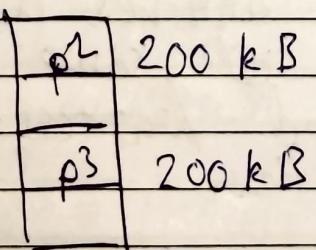
Then we allocate a process of 100 kB



Then we allocate a process of 200 kB



Now p2 completes execution



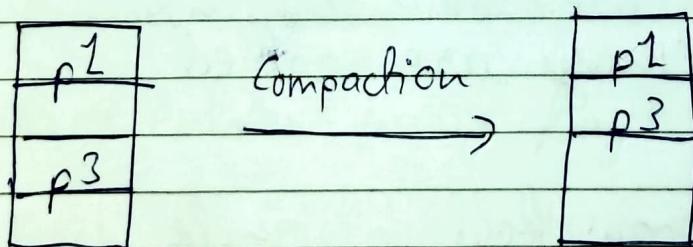
Now suppose, a process of size 150 kB needs to be allocated.

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 divisions are called pages.

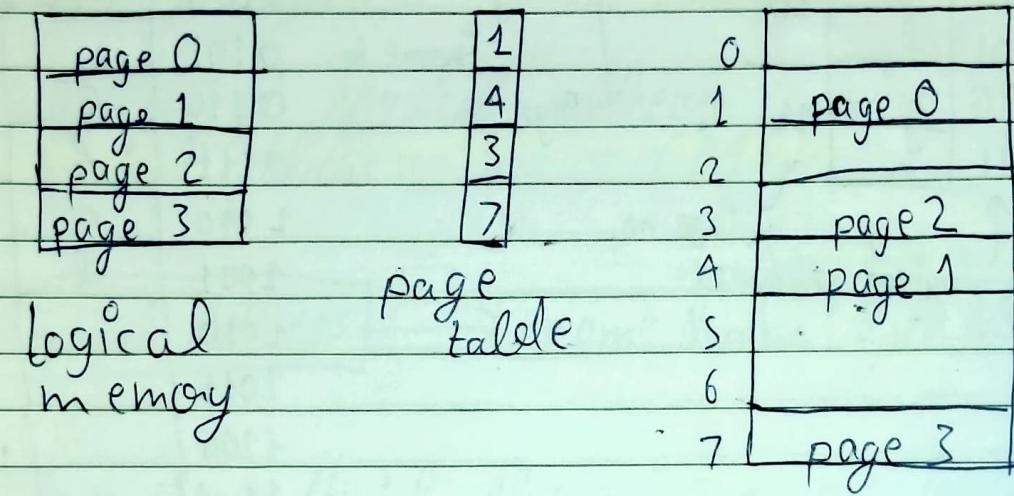
We then divide main memory into parts of the same size as a page size. These divisions of main memory are called frames.

- * 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} ~~an array~~ that stores frame ~~no~~ numbers and the indexes represent page number



* 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 \times 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 ~~data~~

Lecture - 30

→ logical address space

Page Table

Physical memory

000	a	00
001	b	
010	c	01
011	d	
100	e	10
101	f	
110	g	11
111	h	

00	001
01	100
10	011
11	111

0000	000
0001	a
0010	001
0011	b
0100	010

0101	011
0110	e
0111	f
1000	022
1001	c

1010	101
2011	
1100	110
1101	
2110	g

2111	h
111	

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.

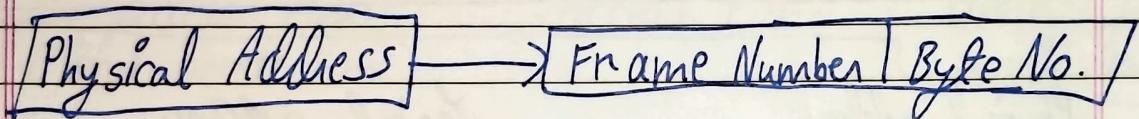
If
memory
byte addressable

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).

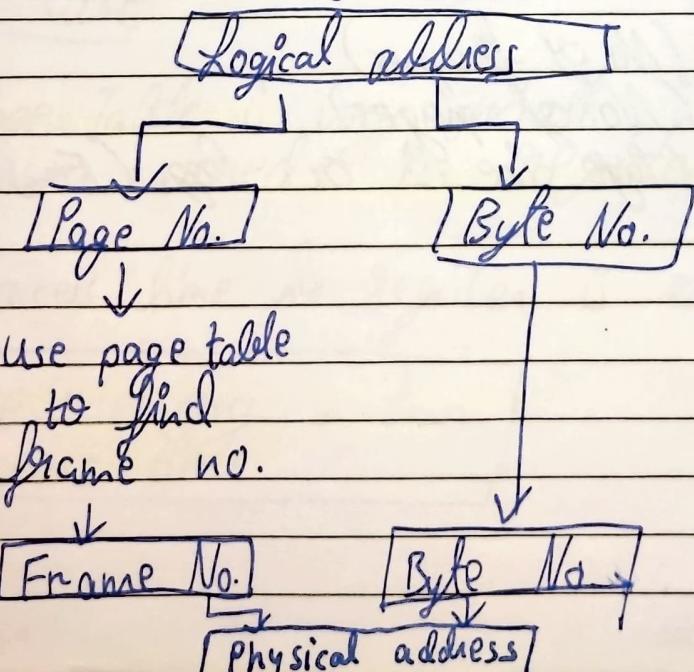
Collection of physical addresses is similarly Physical Address Space (PAS).



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

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

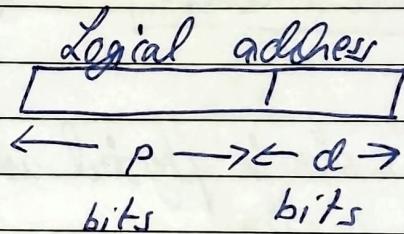
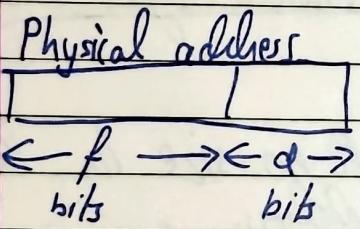
logical address to physical address



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 stored in a contiguous way. (if page size < page table size we use multi-level paging)



d bits \Rightarrow displacement bits; f bits \Rightarrow frame no. bits
p bits \Rightarrow page no. bits

$$f = \log_2 (\text{No. of frames})$$

$$p = \log_2 (\text{No. of pages})$$

$$d = \log_2 (\text{Page size}) \text{ or } \log_2 (\text{Frame Size})$$

Lecture-32

Time Required in Paging

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

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

$$\frac{\text{Effective memory access time}}{\text{Total access time}} = \frac{\text{Page table access time}}{\text{Access time}} + \frac{\text{content access time from main memory}}{\text{Access time}}$$

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

$$\frac{\text{Effective memory access time}}{\text{Access time}} = 2 \times \frac{\text{Main Memory Access time}}{\text{Access time}} = 2 t_{mm}$$

Special Case

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

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

$$\frac{\text{Effective memory access time}}{\text{Access time}} = t_{mm}$$

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 \Rightarrow Translation Lookaside Buffer

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

It stores frequently accessed 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 searches 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 ~~is~~ to all access is TLB Miss Ratio.

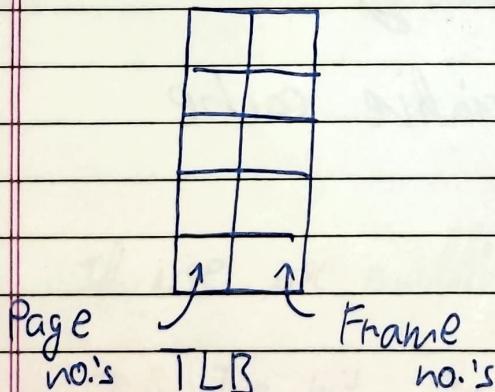
And similar to cache,

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

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

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

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.

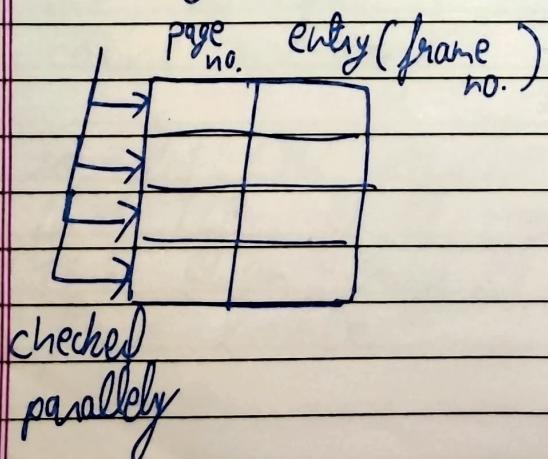
Types

- └ Direct Mapping
- └ Set Associative Mapping
- └ Fully Associative Mapping

Fully Associative Mapping

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

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,

Tag	TLB	Page Table
	00	000
	01	001
	10	010
	11	011
		100
		101
		110
		111

If we are mapping 010

0 is Tag bit & 10 is TLB No.

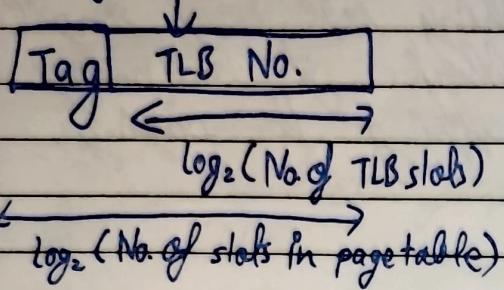
TLB No. Bits = \log_2 (No. of entries TLB can store)

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)



Page Number



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.

~~Because~~ There are multiple slots for entries in a row in the TLB each with its Tag field

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

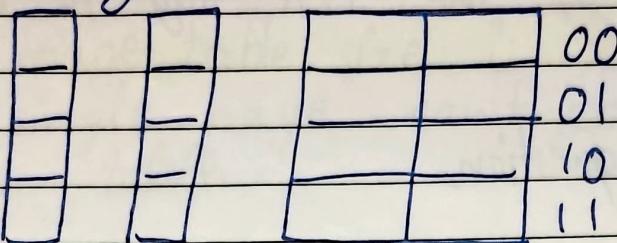
The number of slots in each set is called the associativity.



so for two way set associativity

Tag

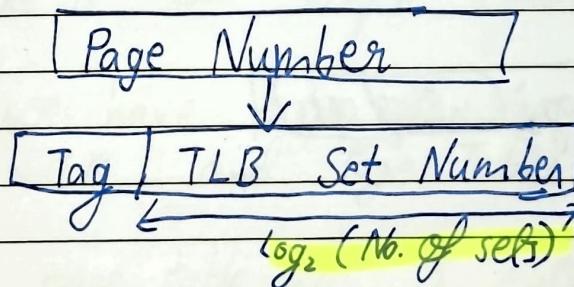
TLB



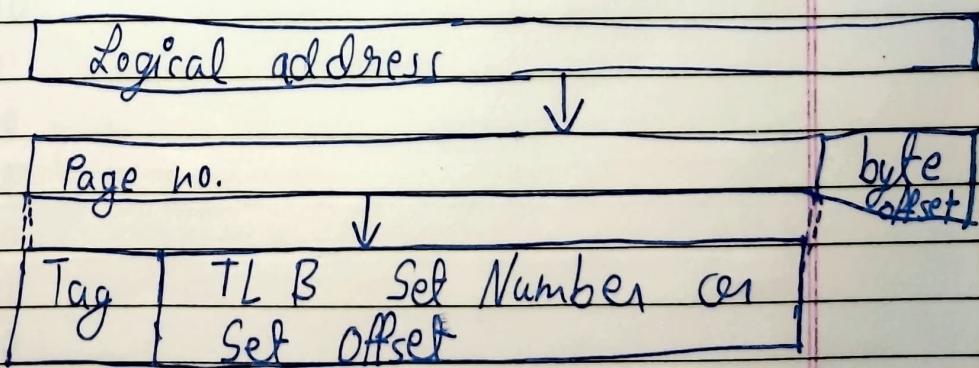
Associativity is 2, so 2 columns per row in TLB.

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

⇒ We number sets rather than individual entries



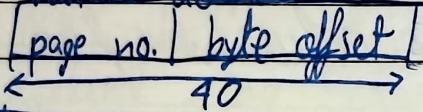
So,



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 ?

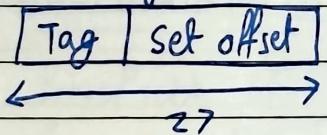
To solve this question,

Virtual address



$$\begin{aligned} \text{byte offset} &= \log_2 (\text{page size}) \\ (\text{in bits}) &= \log_2 (2^{13} B) \\ &= 13 \end{aligned}$$

Page no.



$$\begin{aligned} \text{Set offset} &= \log_2 (\text{no. of sets}) \\ &= 5 \end{aligned}$$

$$\begin{aligned} \text{Tag} &= 27 - 5 \\ &= 22 \text{ bits} \end{aligned}$$

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.

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

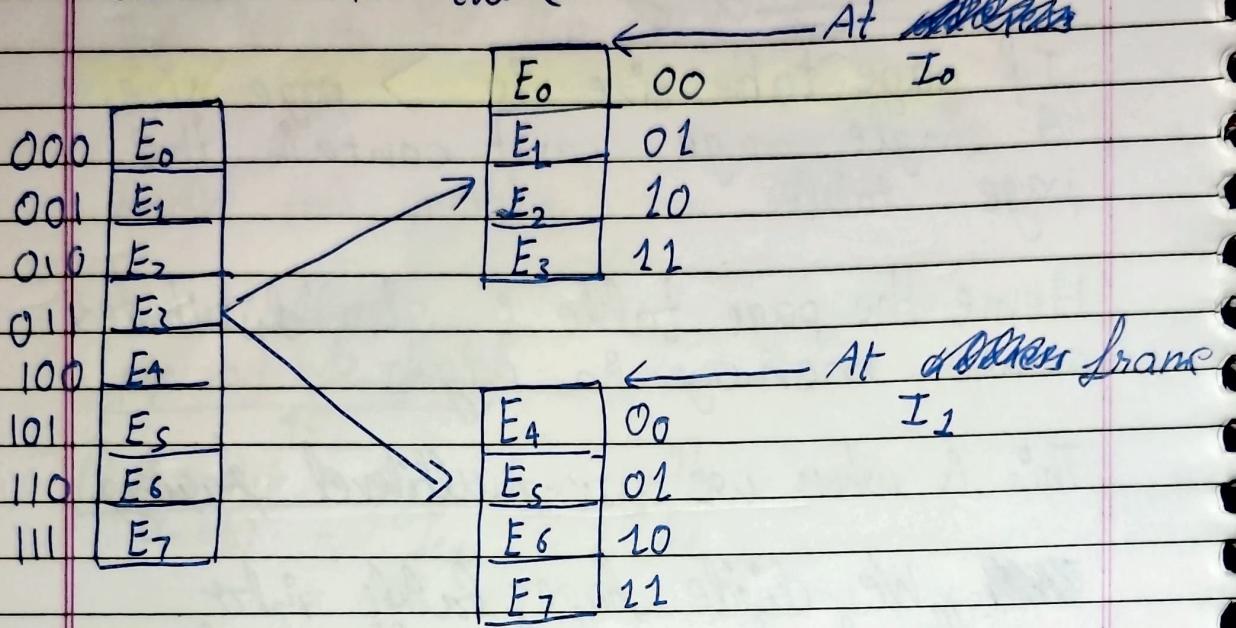
Let's take a ~~very~~ simple example,

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 acquire of tables inner table

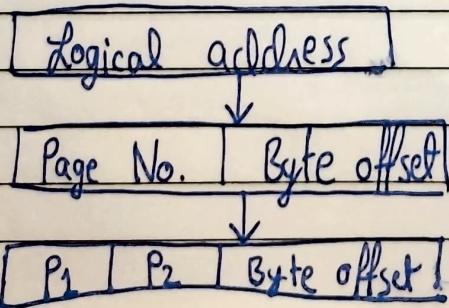


Inner tables

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

size of P_1 in bits = $\log_2 (\text{No. of inner tables})$

length size of P_2 in bits = $\log_2 (\text{No. of entries per inner table})$

The PTBR stores starting physical address of outer table.

Multiple levels

Suppose we have a scenario,

$$\text{No. of pages} = 2^{20}$$

$$\text{Page size} = 2 \text{ kB} = 2^{12} \text{ B}$$

$$\text{Entry size} = 4 \text{ B} = 2^2 \text{ B}$$

Logical address

Page no.	Byte off.
$\leftarrow 20 \rightarrow$	$\leftarrow 11 \rightarrow$

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

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

Fl. Dr.

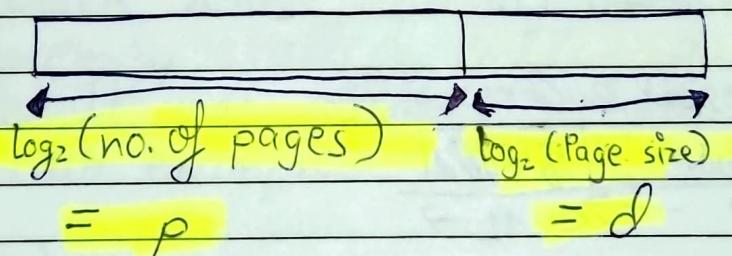
For a given logical address, in multilevel paging or paging in general.

- Find number of page table entries that a single page had hold. Let this number be α .

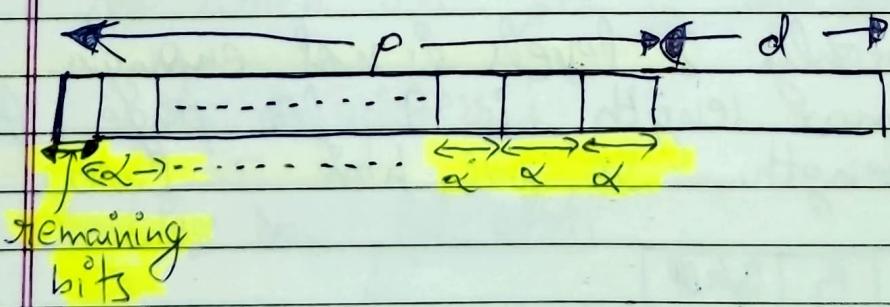
$$\alpha = \frac{\log_{\text{Size of a page}}}{\log_{\text{Size of a page table entry}}}$$

- Divide the logical address in 2 parts

Logical Address



- Now further divide p by α from right to left



The number of fractions of p is the level of multipaging.

~~the access~~

The memory access time now becomes,

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

$$+ \text{Miss ratio} \times (t_{\text{TLB}} + (\text{level of paging} + 1) \times t_{\text{MM}})$$