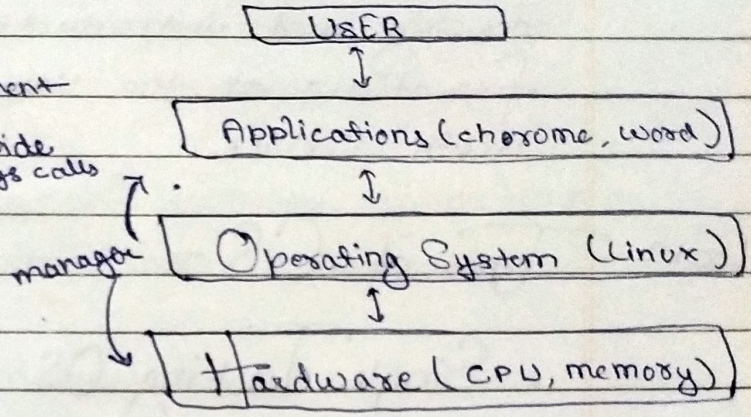
Date __/__/__

Introduction to Operating Systems

Services :-
Resource management
Abstraction - Provide sys calls
Protection



Theory :-

An operating system is a software that manages the computer hardware. It acts as an intermediary between the user of a computer and computer hardware.

The purpose of an operating system is to provide an environment in which a user can execute programs in a convenient & efficient manner. It records that the hardware must provide appropriate mechanisms to ensure the correct operation of the computer system and to prevent user programs from interfering with the proper operation of the system.

- Key functions of an OS

- Convienience - An OS makes a Computer more Convenient
- Efficiency - An OS allows the computer system resources to be used in an efficient manner.
- Ability to Evolve - An OS should be constructed in

Date __/__/__

such a way as to permit the effective development, testing and introduction of new system functions at the same time without interfering with service..

Lec 1.2 Types of OS -

- Single Tasking OS (MS-DOS) / inefficient
  Can run only one program at a particular time.

Runs Concurrently

- Multiprogramming OS (An multiple process to exists in mem & running them in interleaved fashion).
  A computer running more than one program at a time. (like firefox & Excel simultaneously)
- Multitasking OS (more responsive than multiprogramming) → extension of multiprogramming
  Tasks sharing a common resource (1 CPU). (Runs process in time slot manner)

- Multiprocessing OS (supports multiple processor) (distribute process to processors).
  A computer using more than 1 CPU at a time.

☆ • Multithreading OS (More Responsiveness) (Interleaving within Process)
  This is an extension of multitasking.
  (Mutiple threads running in a process in an interleaved fashion).

- Multiuser OS
  This refers to the system where multiple users sitting on different computers can access a single ~~core~~ OS resource.

Date

Threads - smallest unit of execution that can assigned to a CPU. A process can have single or multiple threads.

- Switching threads won't takes much time as switching processes take.

Lec 1.3 Multithreading Introduction

Multitasking - diff process coexist at a time
Multithreading - multiple thing within a process or task (Downloading and ~~browsi~~ surfing internet in browser).

Most famous example for multithreading :-

1. Word processor : Typing, formatting and spell checking
2. Web Server : Apache HTTP server uses thread pool.
3. IDEs : Modern IDEs do compiler check while you're writing the code.
4. Games : Modern games, multiple obj. are implemented at diff. threads.

- Advantages of multithreading
  1. Parallelism and improve performance.
  2. Responsiveness
  3. Better Resource utilization.

Disadvantages of multithreading

1. Difficulty in writing, Testing and debugging code.
2. Can lead to Deadlock and Race conditions.

Example of Race condition

Thread 1	Thread 2
Read x x++ Write x	Read x x++ write x

1st seq. of execution
initial x = 5
x = 7

Thread 1	Thread 2
Read x x++ Write x	Read x x++ write x

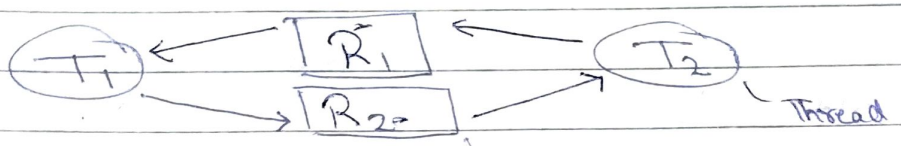
IInd seq. of execution
initial x = 5
x = 6

. It requires experience to write multithreaded programs

- A process is a program in execution. and it can have multiple functions.

1 Deadlock

R_1 and R_2 are non sharable resources.



Non sharable resources
Only one thread can use at a time.

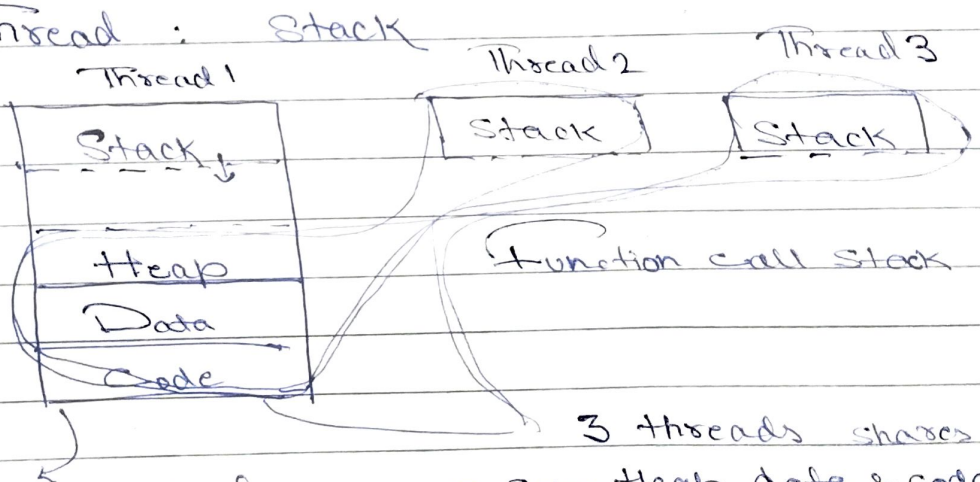
T_1 holds R_1
T_2 holds R_2
T_1 won't ~~o~~ waits for R_2 to release &
T_2 " " R_1 for ".

lec 1.4

Process Vs. Thread

Process - Code, data files, Heap, stack, etc like word processor & music player are 2 diff. processes.

Thread : Stack



- Here we have 1 process and 3 threads running on a single processor

Single thread of execution

3 threads shares the same heap, data & code.

- Parallel execution - threads on diff. processor
- Concurrent execution - on same processor or scheduled on a single processor.

* Stack are basically diff function calls of a process

Note → To communicate betw processes we need OS intervention for some but in threads it might not be the case.

- Threads are very lightweight, faster to create & terminate.
- Multiple threads in a process
  - share same address space
  - Easier to communicate (share same mapping phy addr)
  - Context switching is easier (specially when kernel is not involved and process must be in user space.)
- Threads are lightweight.

Lec. 1 & 5

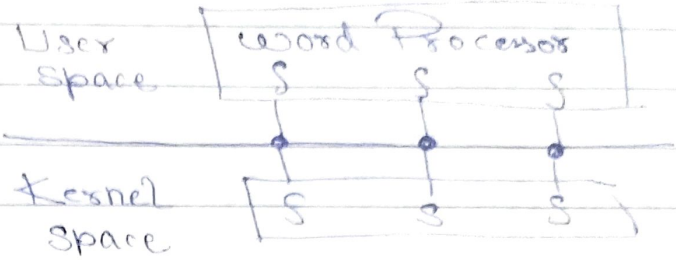
User Threads V.s. Kernel Threads

	User Managed Threads	Kernel Managed Threads
Management:	In user space	In kernel space
Context Switching:	Fast (within same process switching)	Slow (it incurs kernel involvement)
Blocking:	One thread might block all other threads	A thread blocks itself only. Doesn't effect other threads
Multicore or Multiprocessor:	Cannot take advantage of multicore system. Only concurrent execution on single processors.	Take full advantage of multicore system.
Creation / Termination	Fast	Slow (makes a system call, need to involve kernel)

Mapping of user threads to Kernel threads

#MOST common

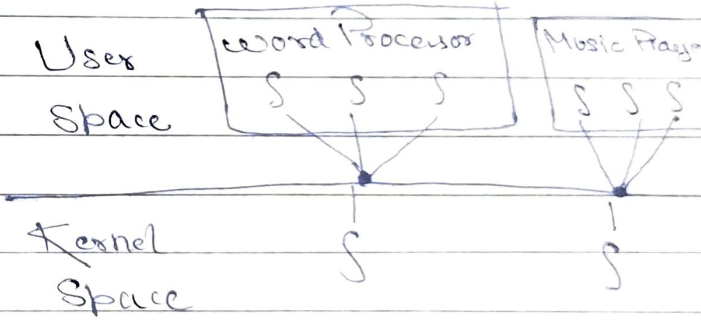
(i) One to One



- mapping of kernel thread to user thread.

- Context switching is slow
- Purely kernel based thread.
- Can take advantage of multithreading
- Blocking one thread doesn't block all the threads.

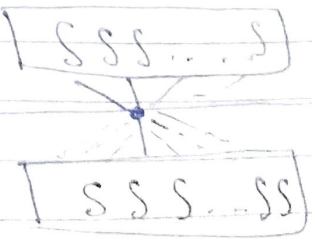
(ii) Many to One.



- multiple user threads mapped to a single kernel thread.

- Context Switching is fast
- Purely user based thread
- Cannot take much advantage of multithreading.
- Block one thread (kernel) blocks all the other thread (SS).

(iii) Many to many



× Module 1 Ends ×

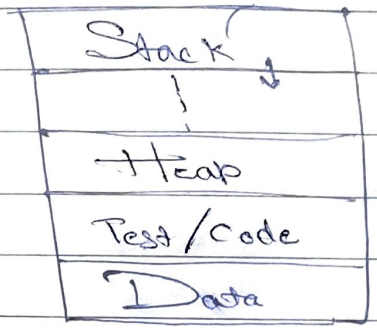
process is loaded into Ram to make it in ready state.

2. Process Management

2.1 Process - A program in an execution

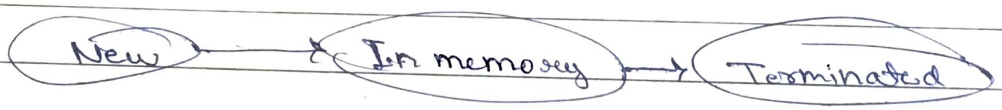
manage function calls

```
Program                    Compiler        Binary
int main(){        ------------------>     Code
  cout << "Hello";
}
```



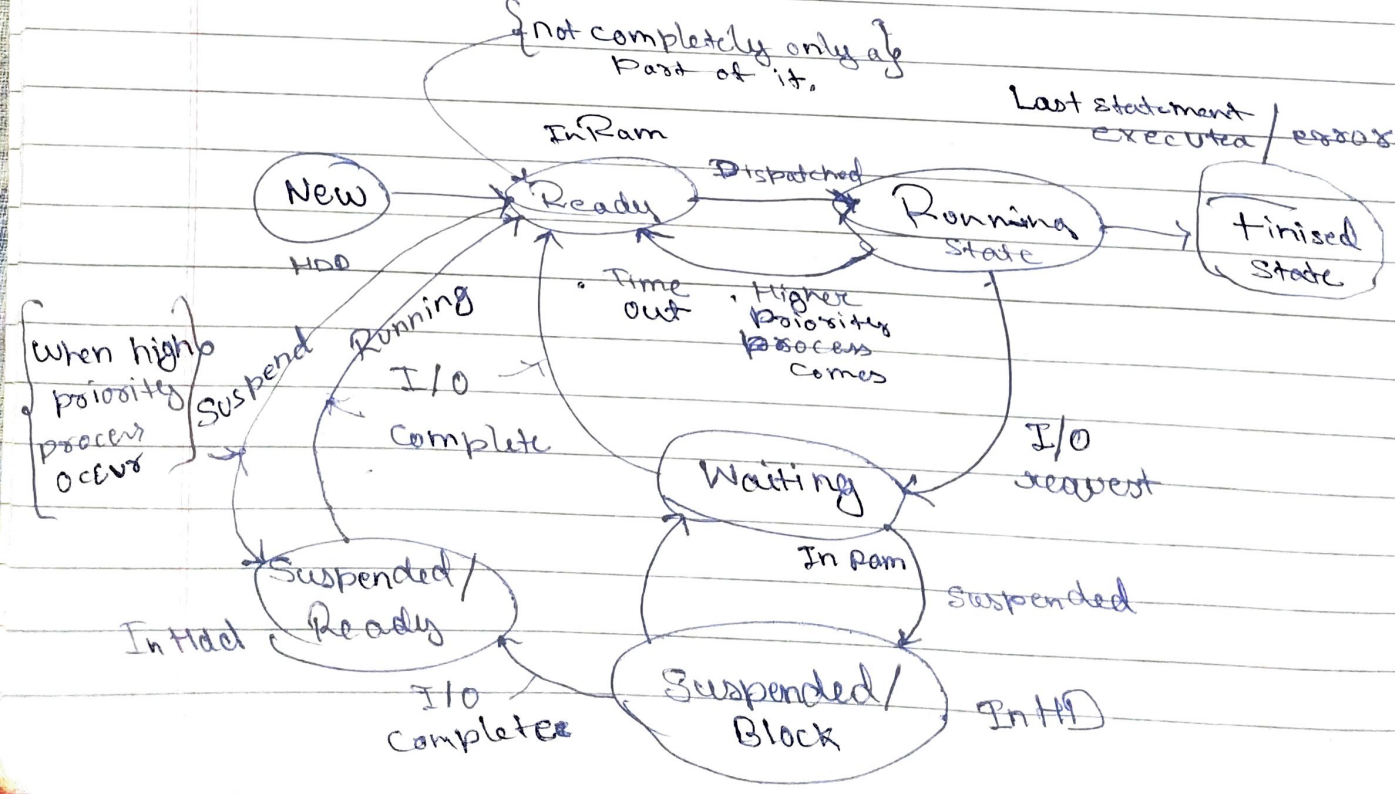
2.2 — Process States

(i) Single Tasking System (MS-Dos) 3-State model



(ii) Multiprogramming System (5-State Model)

Multiple processes can exits in user space at the same time other than the OS.
(Ram)



—— MULTI Processing ——

* Processors are free most of the time.

Time sharing is used to allocate processer to diff processes. These processes can be stopped, so that other processes can run because their time slot is expired and resumed when the other process ends.

Date

- To store suspended process data in hdd →
  - (i) Linux - Swap partation, swap space, dedicated space
  - (ii) Windows - page.file.sys

Note
- CPU never interacts with harddisk. It only interacts with RAM.
- It's the responsibility of OS to assign process to processors

Lec 2.3

Process Control Block

- An operating system must store all the required information about the process, so it can resume it from the same point. All that data is stored in a Operating system in a data structure i.e called Process control Block.

- Most Central Data structure of an OS.
- All OS modules i.e I/O module, memory & process management module use PCB. (process control block).
- Stored in most secure place so that it can't be corrupted.

1. Process ID - A unique identifier assigned to a process by OS.

2. Process State - Ready, running states, finished.

3. CPU Register - Running process uses CPU registers.
   for eg. Program Counter

- Program counter is an important CPU register which tells the next instruction to be executed.
- Program counter is also be saved by the OS in order to resume process from the same point.

4. Accounts Information — Stores the data of how CPU time it consumes so far.
5. I/O information - States of I/O
6. CPU Scheduling info - Priority of process.
7. Memory information. - what all mem. block are allocated to this process.

Lec 2.4 • Process Schedulers

1. Long Term schedulers
   - which brings processes from disk to ram. (or ready state).
   - Controls the degree of multiprogramming
   - ~~Should be~~

IO bound process - uses I/O most of time
CPU " " - uses CPU " " "

* Should bring good mix of IO bound & CPU bound processes.

Date

2. Short term schedulers (Decides which process to acquire processor) running
   - Brings process from ready state to ~~ready~~ state.
   - Assign process to a CPU.
   - Runs actively by OS.
   - Calls Dispatcher (Process control switch).

3. Medium term Scheduler
   (Swapping out & Swapping in)
   - Move process ~~to~~ from ~~ready~~ waiting to suspend block and ready to suspend ready state.
   - Ram to hdd (process swapout & in).

[2.5] Background for Scheduling Algorithms.
(for short term schedulers)

1) Different queues
   - Ready queue
   - Job queue
   - I/O Queue

2) Short term scheduler & Dispatcher (from Ready Queue)

Short term scheduler pick jobs to be done and Dispatcher do the context switch and put the process at to Begin state.

These Scheduling algorithm are for only Short term Scheduler which are in ready queue.

- When the short term scheduler are called -

(a) When some other process move from running to waiting state. The short term scheduler is called to pick some other process quickly and CPU doesn't got idle.

(Premptive Scheduling Algorithm)

(b) When some process moves from running to ready state. because of Time out or some other high priority tasks comes in. (Time bound)

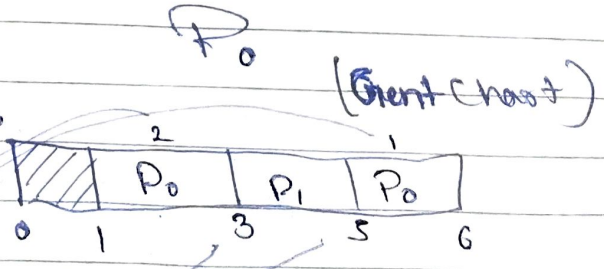
(c) When a new/existing process moves to ready state. (High priority task).

(d) When a process terminates.

- Premptive scheduling algorithm which used by short term scheduling algorithm in which a running process stopped and goes to ready (it terminated by processor) and waiting (stopped itself).

- Different times in Scheduling Algorithms

Points on timeframe
- Arrival Time - 1
- Completion Time - 6



Diff. betw. 2 points.
- Burst Time - 3
- Turn around time - 5 (6-1)
- Waiting time - 2
- Response time - 0

(Completion time - Arrival time)

* Burst time - the time cpu was acquired by a process.

Date

waiting time = turn around - burst time = 5 - 3 = 2
(Time when process is in waiting state)

- Goals of a Scheduling algorithm

(i) max cpu utilization
(ii) max throughput (No. of process per unit time)
(iii) Min Turnaround time (Time diff. between process completion and arrival time)

(iv) Min waiting time.
(v) Min response time
(vi) CPU Allocation. (No starvation)

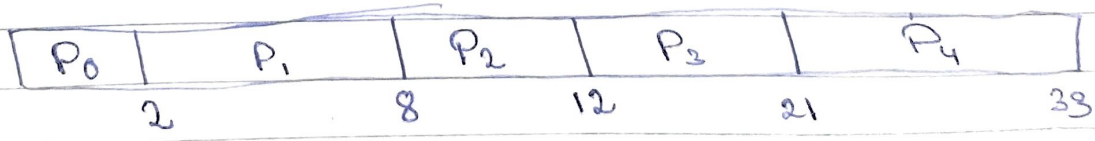
Lec 2.6

FCFS Scheduling (Non-premtive)

(once cpu acquired by a process it won't be taken back from it).

Process	Arrival time	Burst time
P_0	0	2
P_1	1	6
P_2	2	4
P_3	3	9
P_4	4	12

* Process are in the ready queue in FCFS
* Once a process start, it will finish itself



* Turn around time - How much time does every process spent in your system.
Diff. bet^n completion time & start time is turn around time.

Comp. - Arrival | TAT - Burst

Process	Arrival Time	Burst Time	Completion time	Turn around time	waiting time
P_0	0	2	2	2-0 = 2	2-2 = 0
P_1	1	6	8	8-1 = 7	7-6 = 1
P_2	2	4	12	12-2 = 10	10-4 = 6
P_3	3	9	21	21-3 = 18	18-9 = 9
P_4	4	12	33	33-4 = 29	29-12 = 17

$$\text{Avg. waiting time} = \frac{(0+1+6+9+17)}{5}$$

$$= 33/5$$

$$\text{Avg. T.A.T} = \frac{(2+7+10+18+29)}{5}$$

$$= 74/5$$

- Points ~~to remberi~~ remember for FCFS

→ It's easy and simple to implement
→ Non-premtive
→ Can Cause Convoy effect

e.g. if 1 CPU bound process takes CPU multiple I/O bound process have to wait which takes less CPU time.

When CPU bound process end it could create large i/o queues. This is called Convoy effect.

→ Avg. waiting time might not be good. if there comes a cpu bound process before i/o processes.

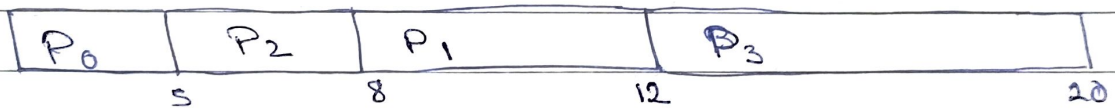
Lec 2.7 Date Shortest Job first algorithm (both Premtive & non-premtive)

Non-premtive SJF

- Consider the Jobs in ascending order of their cpu burst time.

Process	Arrival time	Burst time	Completion time	Turn Around time	waiting time
P_0	0	5	5	5	0
P_1	1	4	12	11	7
P_2	2	3	8	6	3
P_3	3	8	20	17	9

Gent Chart –



$$\text{Avg. Waiting time} = \frac{0+7+3+9}{4} = 19/4$$

$$\text{Avg. TAT} = \frac{5+11+6+17}{4} = 39/4$$

Lec 2.8 Premtive SJF OR Shortest remaing time first

Premtive SJF/SRTF

- Whenever a new job arrives, we compare it's burst time with the remaining time of currently running process. If the ~~the~~ burst time of newly arrived process is less, then we prempt the current process and cpu is acquired by new process.

If there's no new process incoming (acc. to the arrival time) then processes runs in ascending order of their burst time.

Date / /

Process	Arrival time	Burst Time	Completion time	Turn Around time	waiting time
P_0	0	8 ~~7~~	17	17	9
P_1	1	4	5	4	0
P_2	2	9	26	24	15
P_3	3	5	10	7	2

Grent Chart —

Avg. TAT = $\frac{17+4+24+7}{4} = \frac{52}{4} = 13$

(Time in ready & running state)

Avg waiting time = $\frac{9+0+15+2}{4} = \frac{26}{2} = 13/2$

(Time in ready queue)

- Points to remember in Premptive ~~S~~ Shortest Job First and Shortest Remaining time first.

1. Minimum average waiting time among all Scheduling algorithm.
2. May cause high waiting and response time for CPU bound jobs.
3. It is impractical algorithm.

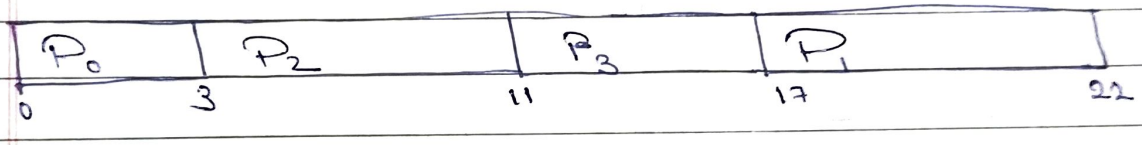
   (We doesn't know the burst time of every process in advance).

- SRTF algo assumes that the OS knows all the burst time of all processes in advance. (in ready queue).

Lec-2.9 Date Priority Scheduling algorithm. (Preemptive/ Non preemptive)

— Non-preemtive PSA

Process	Arrival time	Priority	Burst time	Completion time	TAT	waiting time
P_0	0	5	3	3	3	0
P_1	1	3	5	22	21	16
P_2	2	15	8	11	9	1
P_3	3	12	6	17	14	8



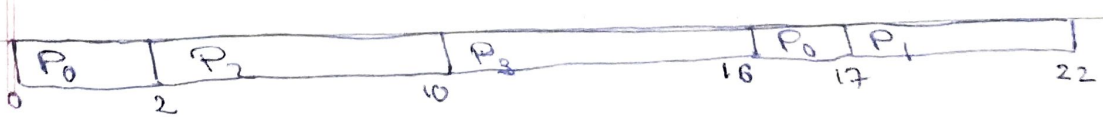
$$\text{Avg T.A.T} = \frac{3+21+9+14}{4} = 47/4$$

$$\text{Avg waiting time} = \frac{0+16+1+8}{4} = 25/4$$

— In this algo, processes are scheduled by given priority. As it's non preemptive once a process acquires cpu, it doesn't let it go until the process finishes. ~~and next~~

— Premptive PSA

Process	Arrival time	Priority	Burst time	Completion time	TAT	Waiting time
P_0	0	5	3	17	17	14
P_1	1	3	5	22	21	17
P_2	2	15	8	10	8	0
P_3	3	12	6	16	13	7



Response time

P_0 — 0 - 0 = 0

P_1 — 17 - 1 = 16

P_2 — 2 - 2 = 0

P_3 — 10 - 3 = 7

There's a problem of Starvation in priority scheduling algorithm. If a low priority process comes first and higher ~~process~~ priority processes keep coming. Then the lower priority process never gets the CPU.

Solution - Aging

- If a process waits for a long time in ready queue, increase it's priority.

— How priority is discussed

- Deadline priority assignment (Static)
- Waiting time priority assignment (Dynamic)

— Round Robin Scheduling algorithm

- You maintain your ready queue as a circular queue.
- We maintain a time quantum. (say 2 units) and keep running processes in circular manner. (Premptive)

→ if a process take less time than time quantum it certn be executed and if it takes more than 2. it will be given 2 units then prempted.

Date

Process	Arrival Time	Burst time
P_0	0	3,
P_1	1	1
P_2	1	3

$P_0 \rightarrow P_1 \rightarrow P_2$

- Avg waiting time can be higher, but good response time.
- Sensitive to Time quantam
  - smaller : Context Switch Overload
  - Large : Become FCFS

	Completion time	TAT	waiting time
P_0	6	6	3
P_1	3	2	1
P_2	9	8	3

Avg TAT = 16/3

Avg. waiting = 7/3

Respons time —

P_0 — 0 − 0 = 0

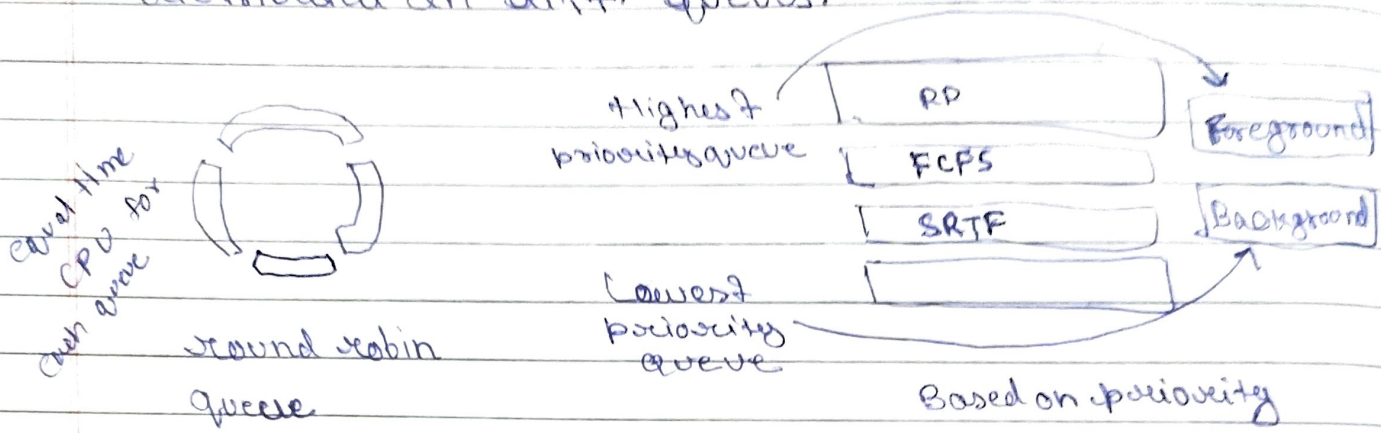
P_1 — 2 − 1 = 1

P_2 — 3 − 1 = 2

* Best part of this algorithm is each process gets response quickly.

Multilevel Queue Scheduling Algorithm

- Flexibility of applying diff. algo. with process distributed in diff. queues.



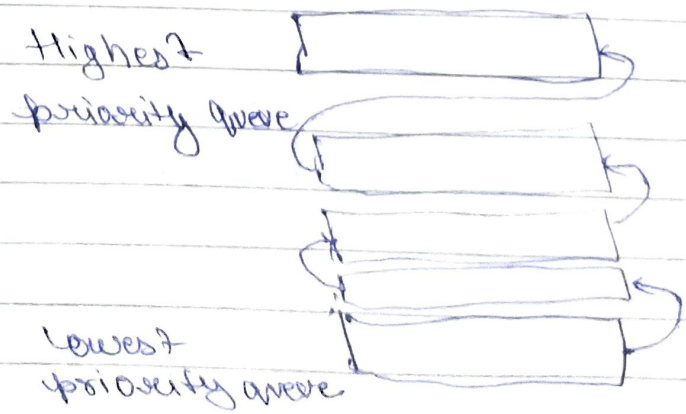
- Disadvantage - The lowest ~~priority~~ process queue experience starvation.

Multilevel Queue Scheduling algorithm with feedback

[MOST USED ALGO]

- I/O bound process has given priority.
- Processes are allowed to move across the queues. Like initially it's a cpu bound ~~procs~~ ~~give~~, moves to low priority but with time it will be I/O bound, so, it moves across queue & goes to a higher priority queue.

- Complex to implement
- Most flexible



3. Process Synchronization

Process — [Independent
Co-operative
» ps | grab "chrome" | wc

- Inter-process communication on one computer

(i) Happens in both a single processor system or a multicore processor system.

(ii) Concurrent execution – In round robin, one process gets prempted by another [P3][P2][P1] or when a high priority process gets in.

(iii) Processor communicate with each other by shared memory or global variable.

For e.g

P1

```
Void producer(){
{
while(true){
while(count == size)
;
buff[in] = produceItem();
in = (in+1) % SIZE;
count++;
}
```

Global variable

```
Int SIZE=0
char Buff[size];
int count=0
int in, out;
```

P2

```
void Consumer(){
while(true){
while(count==0){
;
}
consumeItem(buff[out]);
out = (out+1) % SIZE;
count--;
}
```

exits infinitely if the buff is full

Date

- There will a case or condition ~~when~~ of inconsistency when the Process P_1 gets ~~prempted~~ right when it has to load mem. from register and P_2 will get CPU before P_1.

P_1

```
Void producer(){
    .
    .
    .
    Count++;
}
```

P_2

```
void Consumer(){
    .
    .
    .
    Count--;
```

Count = 8	
reg = Count	
reg = reg+1	✕ gets prempted
Count = reg	

Count = 8	
reg = Count	
reg = reg-1	
Count = reg	

Count = 9

Count = 7

→ Inconsistency in Count value;

Race Condition

When the output depends upon the secquence of execution in a concurrent or multiprocessor or multithreading environment.

for e.g

```
f1(){
    .
    .
    .
    if(x==2)
        y = x+5;
    .
    .
    .
}
```

✕ gets prempted

```
f2(){
    .
    .
    x++;        // x=3
    .
    .
}
```

y = 8

★ but it should be 7.

- f1() gets prempted after if (x==2) f2() executed and f1() gets the CPU back.

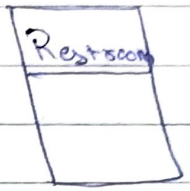
- Race condition also happens when a process is prempted while modifying a shared mem. or global variable and another process use that mem and produce inconsistent result.

Lec 3.2 — Goals of Synchronization mechanism

Mandatory

(i) Mutual ~~execution~~ exclusion
Only One process should be allowed in Critical Session.

(ii) Progress
There should a continous progess.



for e.g Only people who wants to use the restroom should compete for it.

(iii) Bounded waiting (Fair)
Must get equal fair to be in the Critical Session.

(iv) Performance
Locking mechanism. It shouldn't take much time for a process to be in Critical Session. A Locking mechanism can be implemented by both software & hardware.

Lec 3.3 — Overview of synchronized mechanism

- Disabling interrupts
If a process tells the OS that, I'm going into Critical session, so don't interrupt. This may solves the race condition but There are 2 following problems:-

(i) Can only work on Single processor
Fails for multiprocessor.

(ii) Causes Security issues.
(allowing a user-process to disable interrupts)

Date

- **Locks (or Mutex)** (futex in linux)
  - Basic building block for implementing any Synchronization mechanism.
  - Takes the lock ~~exect~~ for critical session, executs and releases the lock.
  - Can be implemented by
    - Software
      - Peterson's Implementation for 2 process
      - Bakery Algo for multiple process
    - Hardware (most Common & fast) ✓✓
      - Test & lock instructions
- **Semaphore** (basically a variable)
  - Higher level mechanism than basic locks
  - Two operations ↳ wait() & signal() (these should be atomic)
  - We use locks for atomicity of wait & signal.
  - Java has semaphore class for thread synchronization.

for C++:
sem_init()
sem_wait()
sem_post()

- **Monitors**
  - Mainly used for thread Synchronization ~~mechanism~~
  - A software sol. managed by JVM - implemented in Java.
  - Monitors are at higher lvl than semaphores

- Instead of putting the code for entry & exit session after every critical session, we put all the shared variable in a class and the methods that modifies the shared variable in this class. We declare these methods as synchronised methods so, that only one process or one thread could only run at a time and individual objects of this class are synchronized.

• Application of Synchronization

As a developer we use thread synchronization a lot. Multithreading is very common. All above mentioned ways are important to achieve that.

- Classical questions on Synchronization

(i) Producer & Consumer
(ii) Dining philosopher
(iii) bounded buffer problem
(iv) Reader writer problem.

Lec 3.4

Lock Mechanism for process Synchronization

P1

```
void deposit(int x){
while(true lock){;}
lock = true;
amount++ = amount;x;
lock = false;
}
```

```
int amount = 100;
bool lock = false;
```

P2

```
void withdraw(int x){
while(lock){;}
lock = true;
amount -= x;
lock = false;
}
```

- There's still a chance of failure as both the process can enter the Critical session, violating the first & mandatory principle of mutual exclusion.

Solution supported by hardware (TSL Lock)
Test & set

Lock is acquired atomically.

P1

```
int amount = 100
bool lock = false
```

P2

```
void deposit(int x)
{
while(test_and_set(lock)){;}
  amount = amount + x;
  lock = false;
}
```

```
void withdraw(int x){
while(test_and_set(lock)){;}
  amount = amount - x;
  lock = false;
}
```

```
bool test_and_set(bool *ptr){
    bool old = *ptr;
    *ptr = true;
    return old;
}
```
ATOMIC

Lec 3.5 Critical Section

P1

```
void deposit(int x){
    NON C.S.
    Entry Section
    balance = balance + x;   <-- critical section
    Exit Section
    NON C.S.
}
```

P2

```
void withdraw(int x){
    NON C.S.
    Entry Section
    balance = balance - x;   <-- critical section
    Exit Section
    NON C.S.
}
```

- You don't need a critical section if you've only read functions. If a write / modify function appear, critical section appears too.

Lec. 3.6

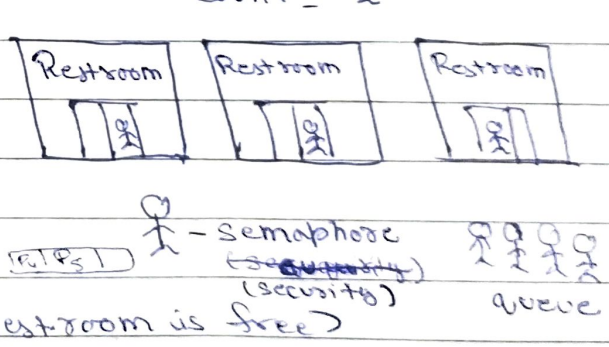
Semaphores

- We use semaphores to overcome the bounded waiting problem. There's no proper queue outside the lock, so, a process can acquire locks again & again resulting other processes not getting the cpu ever.

- Semaphore is basically a count variable

for e.g using restrooms

```
Struct Sem{
    int count;
    Queue q;
}
```



⇒ if count goes (< 0) then the person is added to the queue.

- Before acquiring the resource, we call the wait() process & after using the resource or critical section we call signal().
  count--
  count++;

- If the restrooms are acquired (resource acquired) the semaphore (security guard) tells the people (process) to go to sleep (not run in a continuous loop waiting for the

Date / /

resource to become free). So, whenever restroom (resource) is free signal() will wake you up so they can use the restroom.

- Semaphore store the pointer to PCB of the waiting ~~perso~~ process to call (signal()) later when resource become free (or process exits critical session).

- If count is -ve (it shows no. of process in queue) and when it's +ve it shows no. of free resources.

– Pseudocode for wait() & signal()

```
Struct Sem{
    int count;
    Queue q;
}
Sem S (3, empty);
```

(Counting Semaphore)

```
void wait(){
    S.count --;
    if (S.count<0){
    ① Add the caller process
      P to q.
    ② Sleep P.
}
```

```
void signal(){
    S.count ++;
    if (S.count <=0)
    ① Remove a process P
      from q.
    ② WakeUp(P).
}
```

– Original Implementation by Dijkstra

```
S=3
P(){
    while(S==0)        — wait()
      S=S-1;
}
V(){ S=S+1 }           — signal()
```

} Legacy implementation

Date

This original implementation by Dijkstra have some problems :—

(a) Busy waiting issue
wasting CPU cycles by waiting process. They just keep running & not doing anything.

(b) Bounded waiting issue
A process enters the wait and doesn't gets the CPU as other processes keeps coming in and the process gets prempted continously
That's why we need a queue.

Lec 3.7 Binary Semaphore

It has only 2 values true & false and it can be also used as lock or mutex.

```
Struct BinSem {
    bool val;
    Queue q;
};
```

BinSem S(true, empty); — Global

* Calls just before C.S.

```
void wait() {
    if (S.val == 1)
        S.val = 0;
    else
        ① Put this process P in q.
        ② Sleep (P).
}
```

* Atomicity is req.

* Calls when C.S. is over

```
Void signal() {
    if (q is empty)
        S.val = 1
    else
        ① Pick a process P from queue.
        ② WakeUp(P)
}
```

Date

- ~~Kernel~~ Kernel must implement Semaphores
- Locks used for atomicity
- Semaphores provided by OS.

Lec 3.8

Monitors (better than semaphores)

* Jvm uses monitors to manage threads
- A class consisting shared variable and fonctions to operate on them and these functions are synchronized function.
- There's no keyword "monitor" in java only the concept of monitor implemented using class & synchronized functions.
- Higher level synchronized mechanism

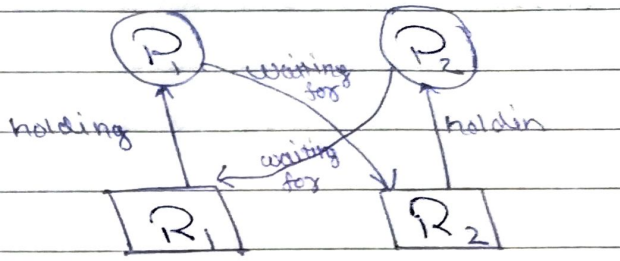
```
Class AccountUpdate(){

    private int bal;
    void synchronized deposit (int x){
        bal = bal + x;
    }
    void synchronized withdraw (int x){
        bal = bal - x;
    }
}
```

- If your class have only one synchronized function then it's a monitor.
- Only one thread can access it at a time.

4. Deadlock

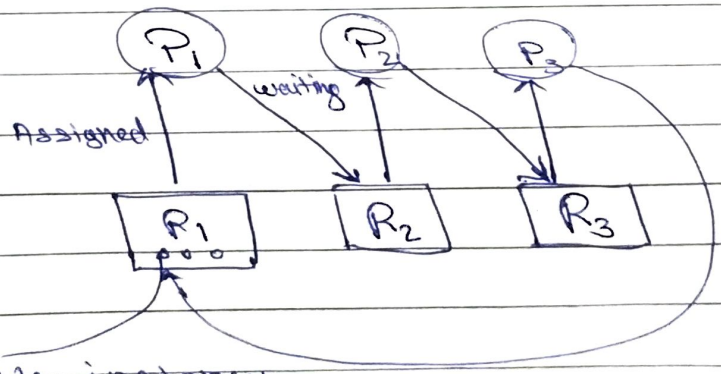
4.1 None out of the 2 processes P_1 & P_2 can be completed.



* Deadlock appears when process hold one ~~share~~ non-shareable resource & waiting for another non-shareable resource.

There are the following 4 necessary condition for deadlock to appear:—

(i) Mutual exclusion
(ii) Hold & wait
(iii) No resource premption
(iv) Circular wait



Resource allocation graph

Multiple instances of a resource.

Date

4.2 Deadlock Handling methods

(1) Deadlock Prevention
Prevent the one of four conditions to happen.

(2) Deadlock Avoidance
Check the incoming req. for deadlock & carefully grants the req. (Banker's algo)

(3) Detection & recovery
Periodically runs a job to check for deadlock.

(4) Ignore the deadlock (most popular)
Ignore the deadlock and let user deal with it.

for linux - deadlock - error EDEADLK

4.3 ⇒ Deadlock Prevention

Eliminate one of 4 of the following :—

(1) Mutual exclusion

- mutal exclusion cannot be eliminated practically because some resources are non-Sharable.

- Spooling
The idea behind spooling is when a process asks printer to print, it gives command and doesn't wait for it. The command/request are collected into a Job queue for printer.

* Spooling isn't a full proof solution for eradicate deadlock. It helps to some extent.

Date / /

② Hold & wait

There are 2 ways to eliminate hold & wait

→ The process needs to tell in advance the resource it'll be use (IMPRACTICAL).

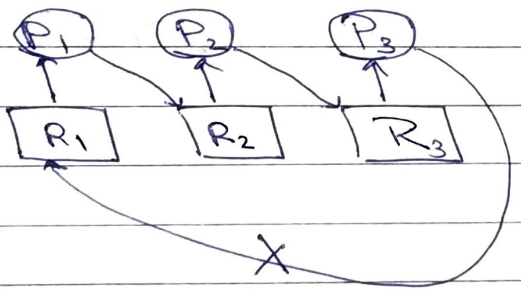
→ A process if asks for a resource, it has to free all the resources it's holding.

③ No preemption (IMPRACTICAL)

Resource can be inbetween in use of a process. If the resorce is taken away in the middle of this, the changes were undone or incomplete.

④ Circular wait

We no. the resources and make a rule a process can only acquire resource in it's increasing no.



This edge is not allowed.

4.4

Deadlocks Avoidance

Banker's Algorithm

developed by Dijkstra

Consider this scenerio and a request from P_3 for <1,0>. Should OS grants this request?

	Allocated		Max	
	R_0	R_1	R_0	R_1
P_0	0	1	7	5
P_1	2	0	3	2
P_2	3	0	9	0
P_3	1	1	2	2
P_4	0	0	4	3

Total = <10, 5>
R_0 R_1

Date ___/___/_____

Check 1 Sum up the req, and given value.

P_3 req $<1,0>$
P_3 having $<1,1>$

Total $<2,1>$ < Max of P_3 $<2,2>$

Passed

Check 2 Compute Available matrix

~~Available < 0 $P_0 + P_1 + P_2$~~
Available < $P_0R_0 + P_1R_0 + P_2R_0 + P_3R_0$ $P_0R_1 + P_1R_1 + P_2R_1 + P_3R_1$>
$<0+2+3+1, 1+0+0+1>$

used
~~Available~~ $<6,2>$ ⇒ ∞
Available = Total − used
= $<10,5> - <6,2>$
= $<4,3>$ > Total of P_3

Passed

check 3 OS assume the give P_3 $<1,0>$ is allocated, then it checks if we allocate this resources the what'll happen.

- To check the new created state after allocation is a safe state or not it generate safe sequence.

→ If safe seq. is generated, then its safe state the process is granted permission to run else if not generated ~~it rejects~~ OS doesn't allow the process to run.

Date ___/___/_____

$P_3 < 1, 0 >$

new State

Safe seq. = Permutation of processes

$P_2\ P_3\ P_1\ P_0\ P_4$ –

if all process can run serial wise then, it'll be the safe seq.

	Allocated		Max	
	R_0	R_1	R_0	R_1
P_0	0	1	7	5
P_1	2	0	3	2
P_2	3	0	9	0
P_3	2	1	2	2
P_4	0	0	4	3

~~Total~~ $< 10, 5 >$

$R_0\ R_1$

Available $< 3, 3 >$

∴ This state is safe

	Allocated		Max		Need	
	R_0	R_1	R_0	R_1	R_0	R_1
P_0	0	1	7	5	7	4
P_1	2	0	3	2	1	2
P_2	3	0	9	0	6	0
P_3	2	1	2	2	0	1
P_4	0	0	4	3	4	3

~~Total~~ $< 10, 5 >$

Available $< 3, 3 >$

Algo :–

```
Safe_seq = {};
while (All Pi are not added to the safe_seq){
    (a) Find Pi such that
            need i <= available
    (b) if (no such i exists)
            return false;
        else (we found an i){
            available += allocated i;
            Add Pi to the Safe_seq;
        }
}
return true;
```

Date

available < 5, 3 >
P_1

available < 7, 4 >
$P_1 P_3$

available < 7, 4 >
$P_1 P_3 P_4$

available < 7, 5 >
$P_1 P_3 P_4 P_0$

available < 10, 5 >
$P_1 P_3 P_4 P_0 P_2$

There's a safe sequence exists so, the proposed process can have the permission.

- Disadvantages

  - It assumes that every process knows in advance it's maximum need.
  - It also assumes that once a process is done it releases all the resources.
  - It's more theoritical than practical
  - If the system is in unsafe state, it doesn't mean that deadlock will happen.

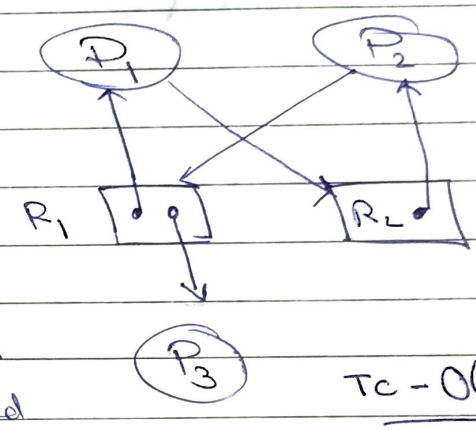
    But for the safe side, if the safe sequence doesn't get generated, we didn't allow the process to run.

Lec 4.5 Date 1 Deadlock detection & recovery

- Detection

If the resource allocation graph has a cycle and each ~~process~~ resource have single instances. i.e there's a deadlock.

→ Here in this resource allocation graph, there's no deadlock. Once P_3 done releases one instance of R_1. R_2 resumes and finished R_2 released P_1 continues and finished.



Tc - O(V+e)

- Banker's algorithm for deadlock detection four multiple instances ~~O(V+e)~~

Same as deadlock prevention algo, with slight changes.

(i) We do not consider those process whose allocation is 0.0.0. They're not the process who are holding some resources than waiting for some resources.

(ii) The process which cannot be added to the safe sequence is the process in the deadlock.

Deadlock recovery

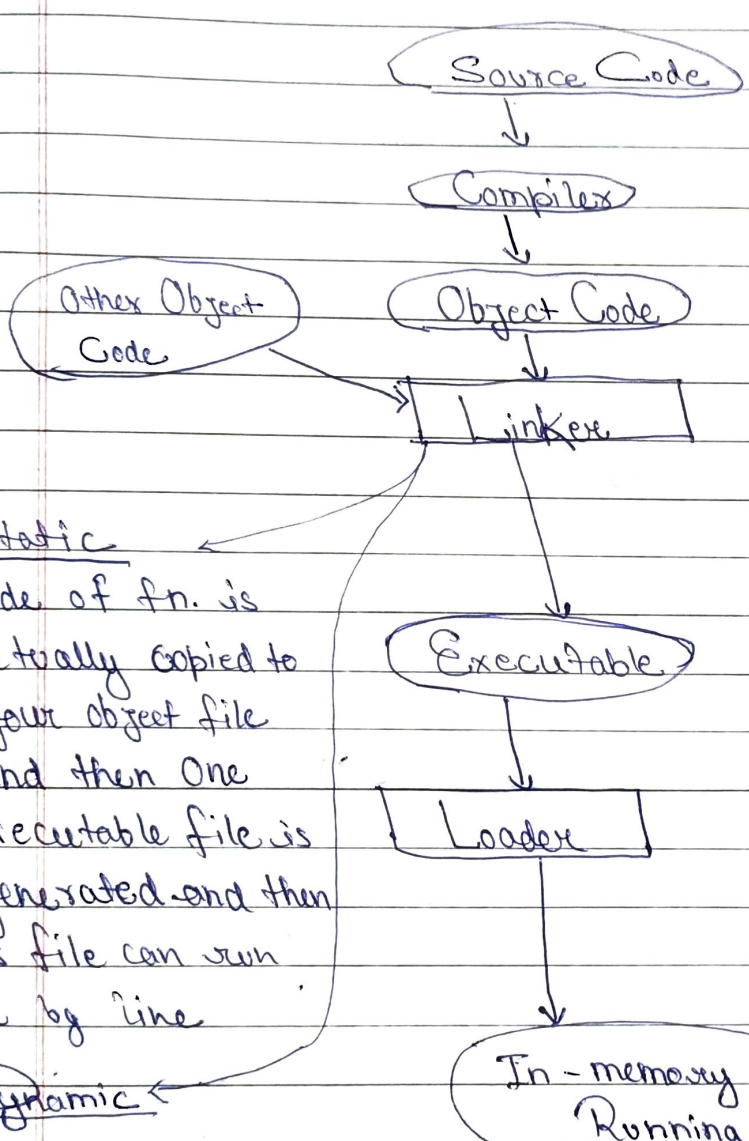
1) Kill process(s)
2) Preempt resources (Can cause starvation)
   (Store the states and roll back process to ~~part~~ previous stored state).

- Kill process
  (i) Kill all process it will be out of deadlock for sure.
  (ii) Kill one process and check deadlock.

  - Criteria for picking up a process to kill
    - Priority of this process
    - no. of resources this process is holding
    - CPU time already taken

5. Memory management & Virtual memory.

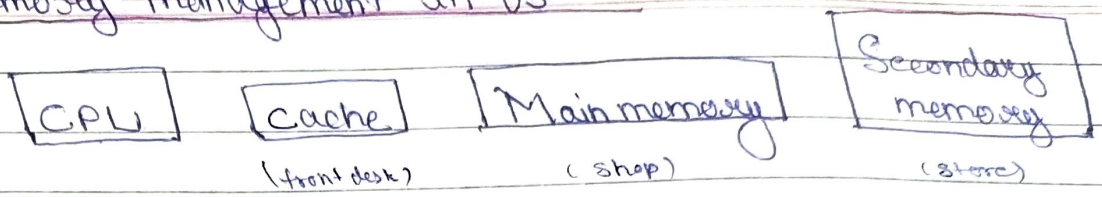
- How are programs compile & run?



* A process isn't allowed to look through the memory. So, in dynamic linking, help of OS is required. OS does the searching job.

Memory management in OS

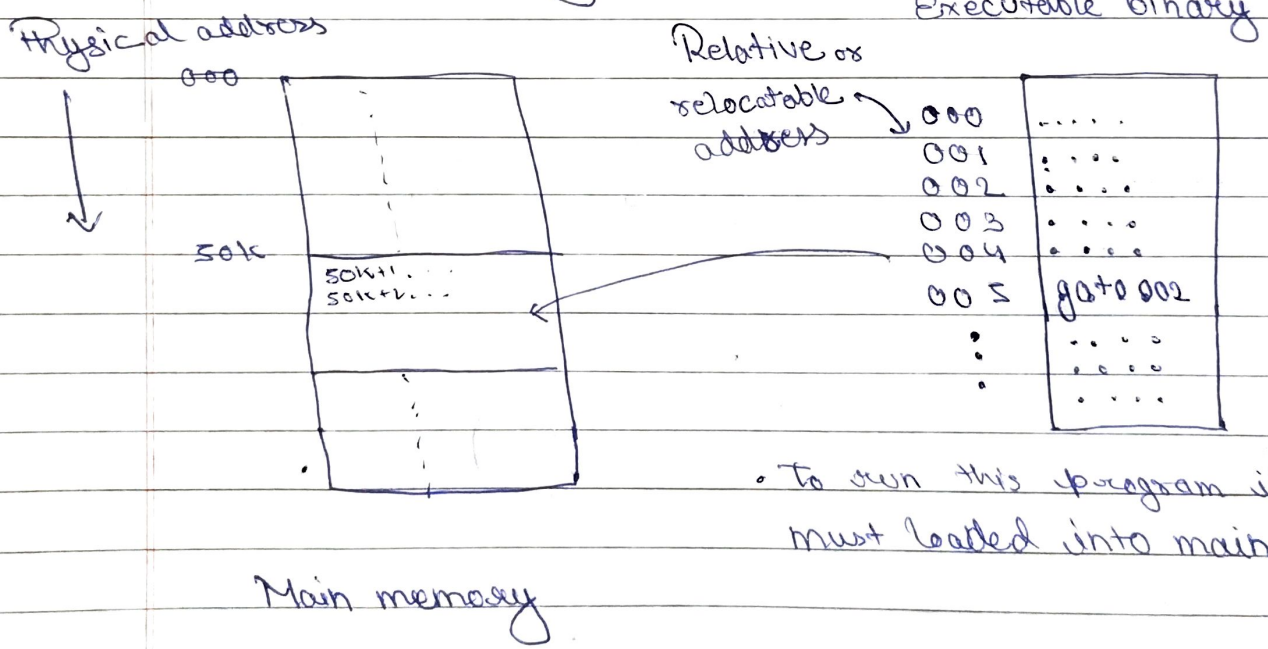
Lec-5.2



- memory is available to the CPU in ascending order of their access time.

- An ideal memory have
  - Access time ↓
  - Capacity ↑
  - Cost ↓

Lec-5.3 Address Binding



- To run this program it must loaded into main

- Address binding is mapping of relocatable address to physical address.

Date __/__/____

- Address binding can happen at different stages

- **Compile time binding**

  If your compiler knows where this program is being loaded in the main memory.

  for eg. .com files in MS-DOS used compile time address binding.

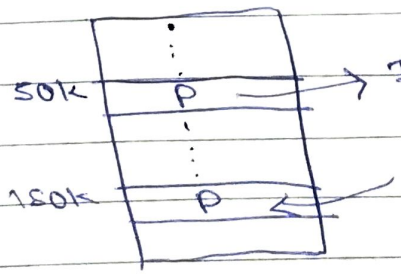
  Actual physical address is mentioned in the program in first place.

- **Load time binding**

  The loader actually consists the program with ~~reference~~ relative address and load it to the main memory with some base address and all the relative address were recompute with the reference of base address of main memory.

- **Problem with load time binding**

  Once the address is allocated to a process in main memory, it can't be changed. If this process starts doing I/O, we've to move this program to hdd. Process can't be moved.

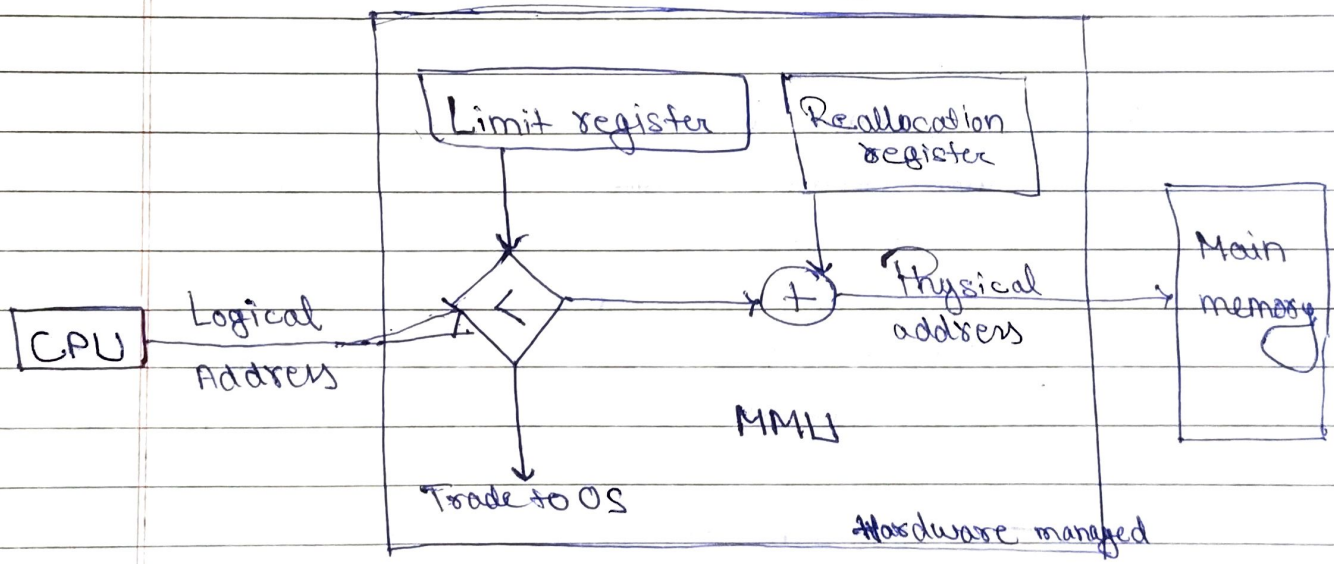


∴ Load time binding fails

Date __/__/__

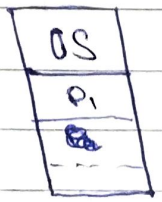
- Run Time binding (Happen in all modern OS.)
  - CPU generates logical address and later converted to physical addresses during runtime.
  - Hardware support is required for the run time binding.
  - Memory Management Unit (MMU) converts the logical address to physical addresses.
  - Your process can move to diff. location even while running.

Lec-5.4



Lec-5.5 Evolution of memory management

1. Single tasking system

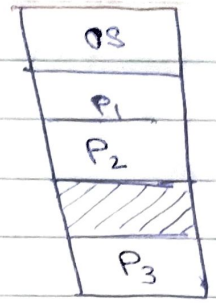


Multiple process should be present in CPU to utilise it better.

Date

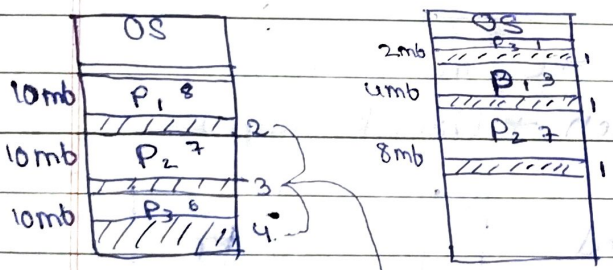
2. Multitasking System

Mutiple process can exists in memory at the same time.



→ Memory allocation in Multitasking

- Static
  - Equal Size
  - Unequal Size
- Dynamic
  - (i) Contiguous
  - (ii) Non-Contiguous
    - (a) Paging
    - (b) Segmentation
    - (c) Paging with Segmentation.



* External fragmentation

- Internal fragmentation – When more memory was allocated (i.e in equal size allocation) than needed.
- Internal fragmentation cause external fragmentation
- External fragmentation appers when a new process can't be loaded in the main memory but total unused memory >= process requirement

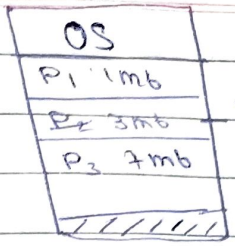
→ Static memory allocation always suffer from internal as well as external fragmentation.

—x— Contiguous memory allocation —x—

Date ___/___/___

Here, if a process with req. 5mb comes can't get loaded into the memory.

P_1 1mb
P_2 3mb
P_3 7mb



→ There's no internal fragmentation but external fragmentation still exists in dynamic allocation.

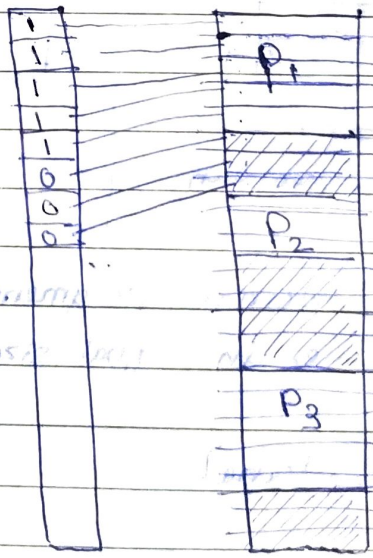
→ Solution — Compaction or defragmentation (Compaction is costly and is not feasible)

→ Degree of multiprogramming is not prefixed.

Lec 5.6 → When process leave the main memory, it create holes. (Solution → Use Bitmap)

* 1 represent occupied
  0 represent free space.

So, one can keep order of holes by traversing the Bitmap.



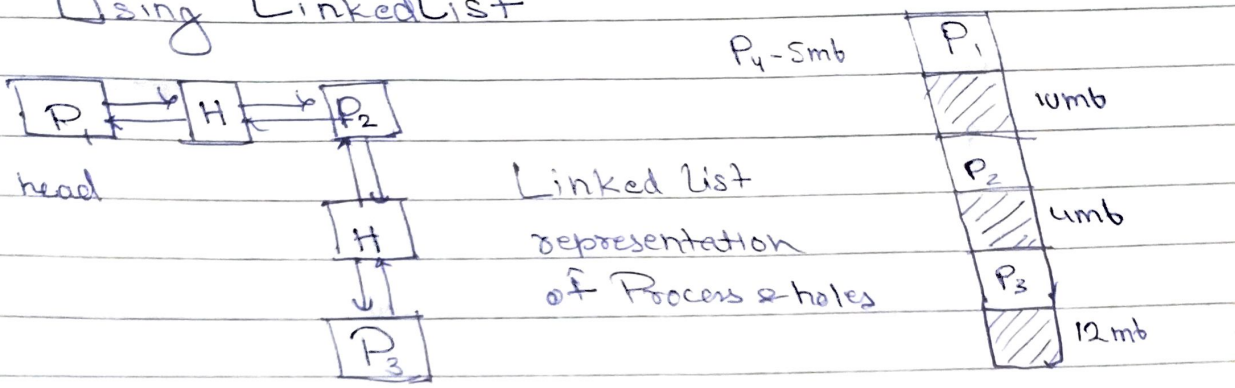
Problems :—
- Requires a lot of space.

Note — Bitmaps aren't used to solve the holes problem.

Non-Contiguous memory allocation

Date

- Using LinkedList

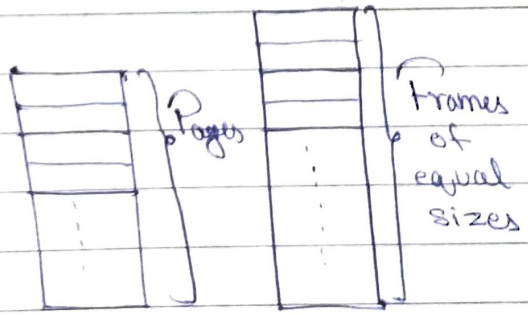


- First Fit - keep traversing from beginning until we find hole.
- Best Fit - Can cause small sized hole
- Next Fit - Start searching from last point (or search).
- Worst Fit - Always allocate the biggest slot.

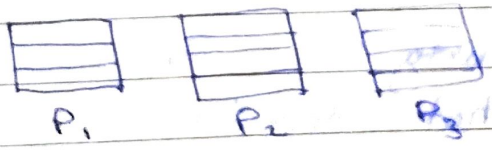
5.7

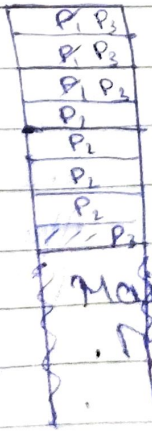
Paging in memory management

- Page size is always equal to frame size.



- Any process can be allocated in any no. of distinct frames available at any locations as long as these frames are available.



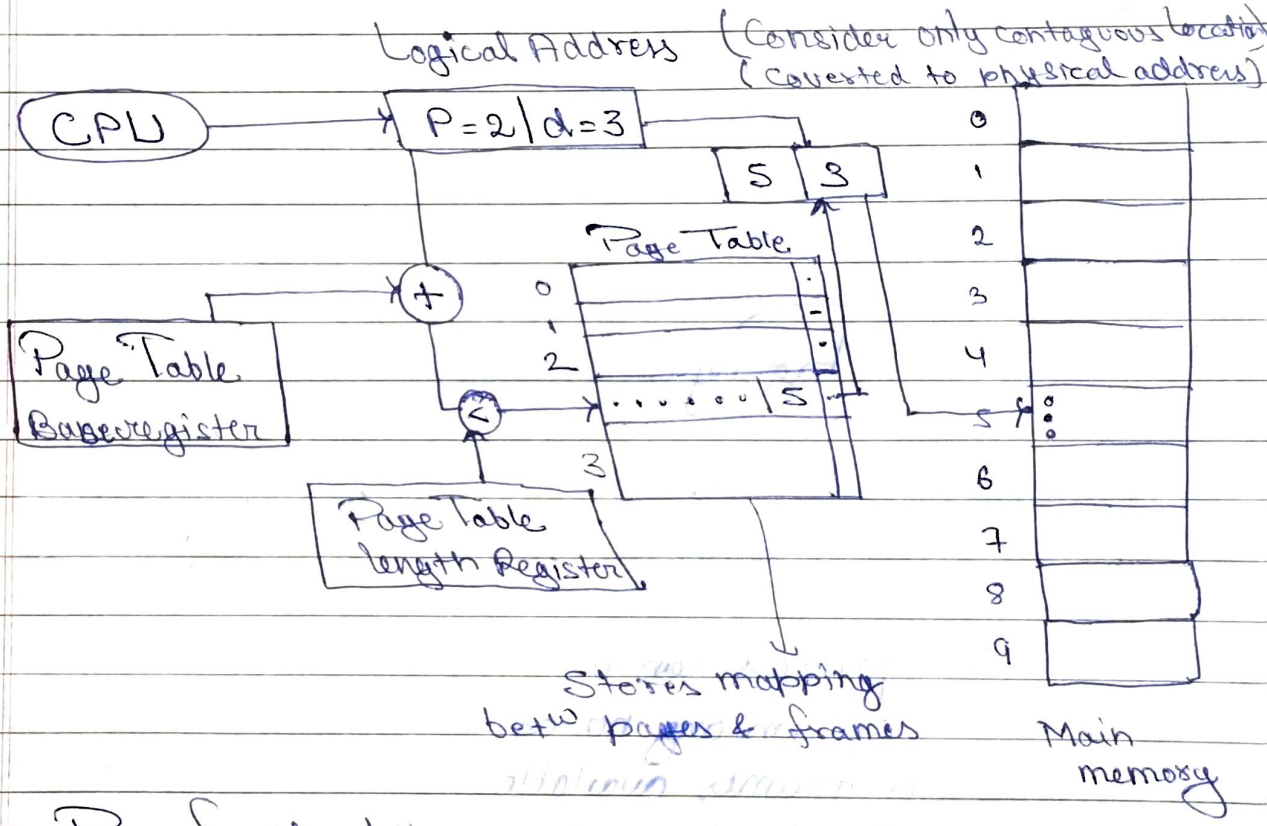


- You're not wasting any memory. So, fragmentation problem is solved.

Date ___/___/___

Frames are something which exists in main memory which can accomodate pages of processes.

- Run time binding is used in paging.
- Logical addresses are contiguous ~~for. allocated~~ location.



- Page fault : When a page is in page table but not in main memory. (in hdd but not in ram)
- Modified bit or dirty bit : An entry in the page table that item tells you whether the data is modified or not.
  (Swapped out or written on the harddisk for doing some I/O)
- Cacheable entry in Page table.

Lec 50

Virtual Memory & related Concepts

— Ondemand paging

- Degree of multiprocessing is higher.
- Chances of page fault increases.
- Only load the required part when something is required load that page from hdd.
- Use the concept of locality of reference. (load some pages which is accessed and it's nearby pages).

— TLB (Transitional look aside Buffer)

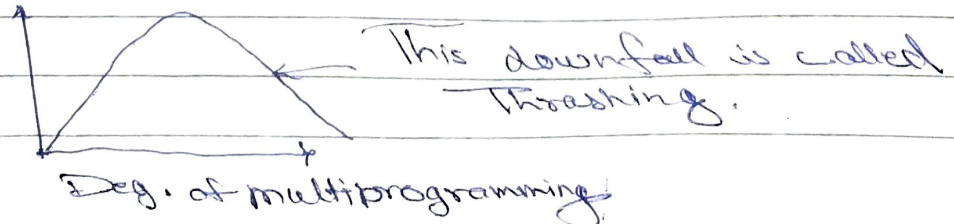
- Associated cache present inside your CPU.
- It's purpose is to optimise the page table lookup as because it happens frequently.
- Hit ratio of TLB is very high.

— Demand Paging

- You only keep the working set of a process in main memory and in pure demand paging environment we keep the minimal part of the process in memory.

— Thrashing

When you allow many process to run in the memory by allowing only a small part of the process to load at a time. This increases the degree of multiprogramming but also the chances of page fault occurence.



Page replacement Algorithms

To replace a page which is already in memory with a new demanded page, we'll use page replacement algorithm.

* Page replacement can be local or global

- Local Strategy for page replacement (most famous)
  A process have a fixed working set or fixed memory allocated in ram. It when a page fault occur, you need to replace one of process's pages to bring up the new page.

- Global Strategy for page replacement
  - You can replace pages of other process to bring page of this process.
  - Overall performance increases by individual processes may suffer.

+ There are the following 3 page replacement algorithm :

(i) First in First Out

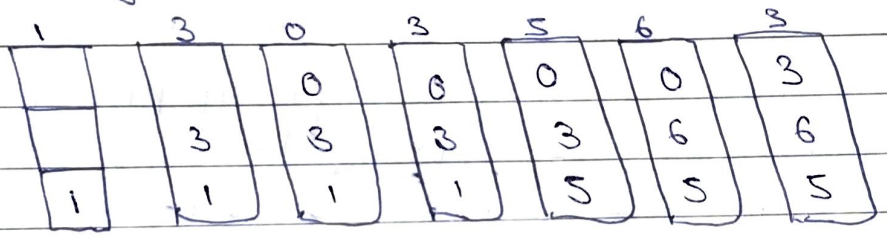
It suffers from (belody's anamoly).
- It states if there're more frames for a process, then the possibility of page faults also increases in some case.

Page fault → A page fault happens when a running program accesses a memory page that is mapped into virtual address space, but not loaded in physical mem.

Date

- This is the simplest algorithm for page replacement. In this algo, the OS keeps a track of all the pages in memory in a queue, the oldest page is at the front of the queue. When a page needs to be replaced, page in the front of the queue is selected for removal.

for. eg. Page reference 1,3,0,3,5,6,3



Miss Miss Miss Hit Miss Miss Miss

Total page fault = 6

(ii) Optimal Page replacement (Not practical coz OS can't know future reference)

In this algo, pages are replaced which would not be used for the longest duration of time in future.

for. eg. Page reference 7,0,1,2,0,3,0,4,2,3,0

No. of page frame = 4 3,2,3

Total page faults - 6

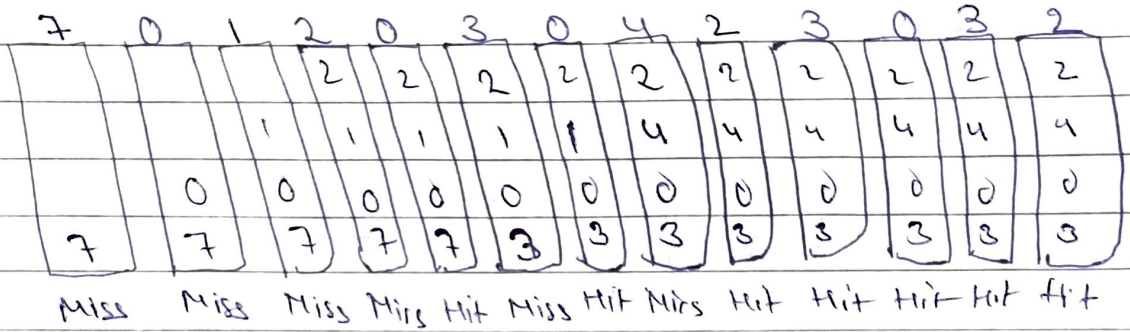
Date ___/___/___

(iii) Least recently used

In this algo, page will be replaced with which is least recently used.

for eg. Page reference 7,0,1,2,0,3,0,4,2,3,0,3,2

Page frame → 4



Total page faults - 6

Lec 5.11

Segmentation in OS (memory management)

(Alternate of paging)

- Adv of paging
  - We can implement a process in non-contiguous manner in memory
  - Helps to implement Virtual memory.

Main idea ~~behind~~ behind segmentation

- Brought the related item ~~in memory~~ together in main memory.
- Segemention, we divide items acc. to User's view.

X64 arch - doesn't implement Segmentation
- Works on Pure Paging.

x86 - Code Segment
Stack "
data "
extra "

- Non-Contiguous memory allocation with segmentation.

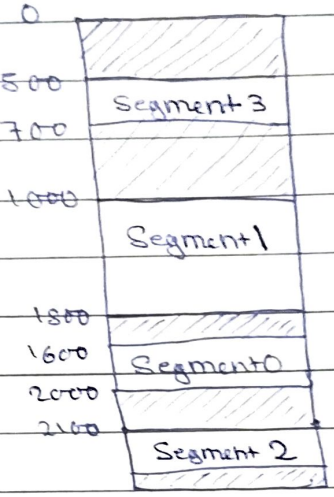
Logical view

Segment 0
Segment 1
Segment 2
Segment 3

	Base	Limit
0	1600	400
1	1000	500
2	2100	400
3	500	200

Segment Table

Physical View



Simple Segmentation:
All segments are present.

Virtual memory Segmentation:
All segments needs not to be present.

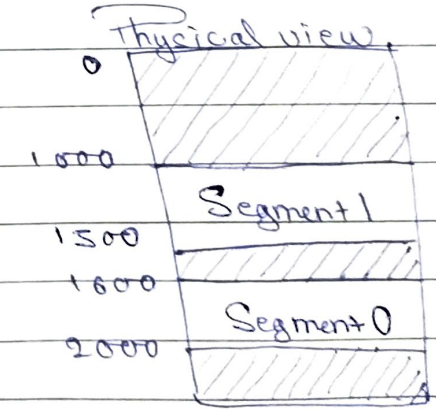
Logical

Segment 0
Segment 1
Segment 2
Segment 3

	Base	Limit	Absent/Present
0	1600	400	1
1	1000	1500	1
2			0
3			0

Segment table

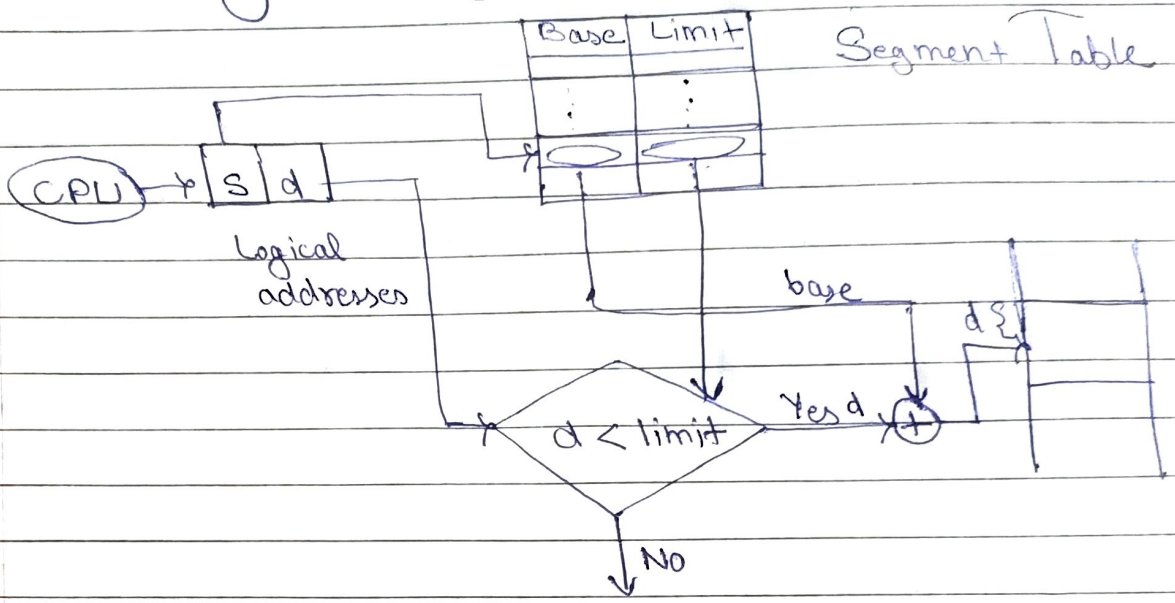
Physical view



- Non-Contiguous allocation
- Implementation of virtual memory
- Segments doesn't have to be of same sizes.

Segments - Collection of similar items.
Variable size segment of the process/User View

Working of Segmentation

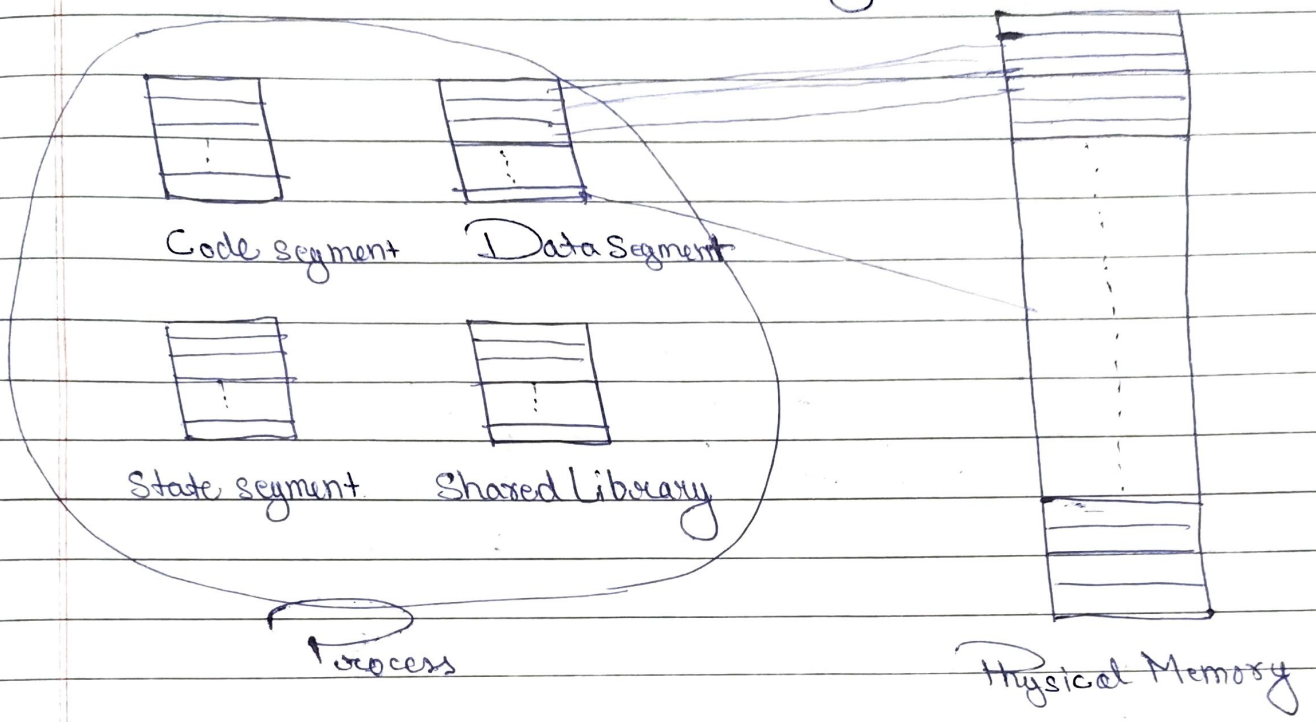


Pure Segmentation
- Simple
- Virtual

Advantages over paging
- No internal fragmentation
- User view (no page fault).
- Segment table in smaller than Page Table. Therefore easier to implement sharing & protection.

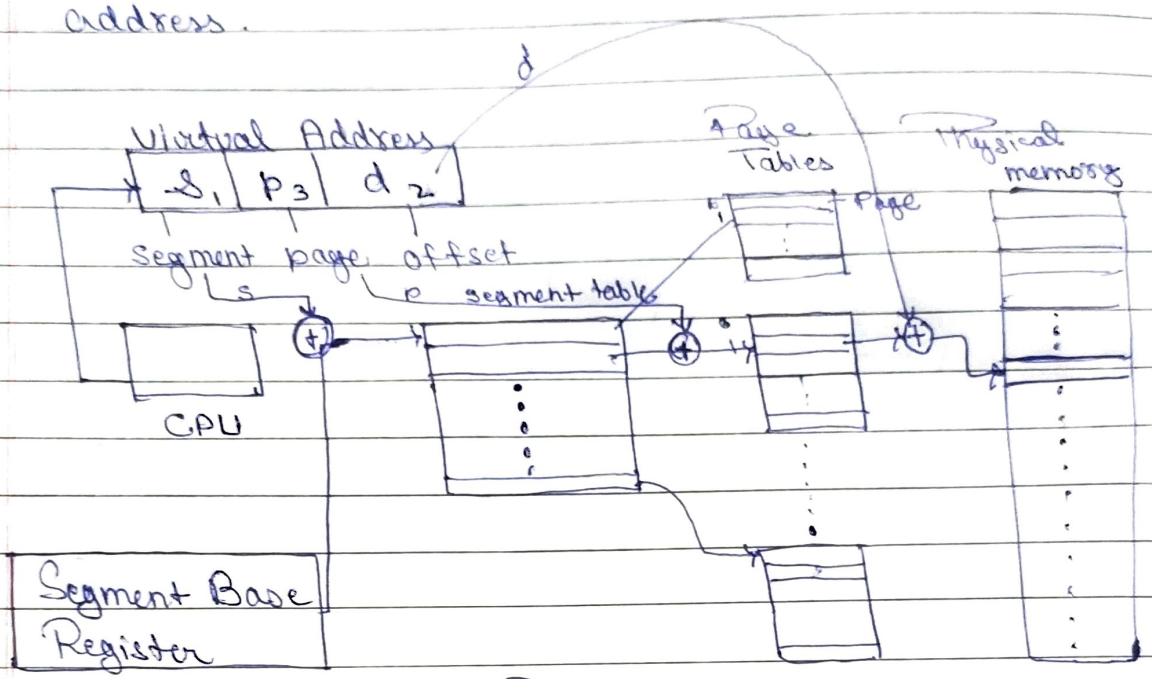
Disadvantages → External fragmentation

Segmentation with paging



- Process are divided into multiple segment and each segment have pages.
- Each segment is going to have their own Page tables and the process is going to have a segment table.
- Logical address is going to have the segment no. page no. and have offset within a page.
- Paging uses memory efficiently and segmentation gives you the user views of the process.
- When the context switching happens i.e from one process to another, a register in the CPU (Segment Base register) gives you the base no. address of segment table. and from segmtion to page table further to page frame and

with the help of offset, we get the physical address.



Single Process

~~Ex: ...~~

Disadvantages :—

- It creates 2 lookups. One for segment table and one for page table.
- Pure segmentation is rarely used.