Date
Page

Operating System - I

App
OS
Device

~~Devices to talk~~ to apps needs an OS in b/w. (Middle level comp.)

code → compile → object code → Process (Main Mem) → Graphics card → Monitor

- Abstracts the hardware no need for us to configure the device.
- It does Resource Mgmt.

Processor

power to run and execute instructions. compared by clock speed.

5 GHz → 5×10^9 instructions per sec.

RAM >>> HD.
↓ Magnetic Disk
capacitor & Transistor

Cache >>>
100KB - 8MB.
↑ used with cache

- SRAM → Static → Transistor
  DRAM → Dynamic → capacitor
  ↓ used in RAM. ↓ leak current and are slow.

- Multi-Programming:

if a process waits for I/O processor does a context switch to a new process.

How does processor know?
Status changes from I/p to Ready.

• Multi-Processing

Same scenario as before but now a new processor.

So, Multiprogramming + Processor (Multi)

• MultiTasking

You are processing 1 process at a single unit of time but when you look at over a span of time You'll find a bunch of process giving a view of simultaneous execution of processes. → This is Multitasking (Time Slicing takes place)

•all these require Job Scheduling

(A) Print A-Z (B) Print 1-100

So in Multitasking → Since Time is sliced O/P → ABC 123 DEFGH..79

• Process is always run in RAM

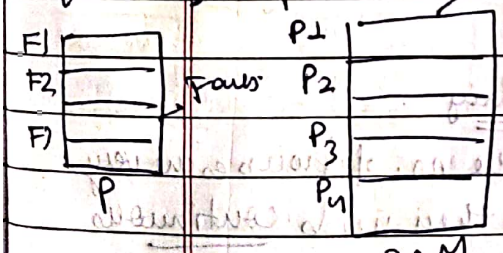
• Programs reside in HD.

#Fragmentation

You have available space but not contiguous amount of space → Ram is split into fragments of free memory.

How to solve?

Divide the process into equal sized frames. → Page



Frame Size = Page Size

Date
Page

So now we can assign frames non-contiguously.

to keep track of frames and page we have -

Processor Page Table.

# Frame	# Page

→ For Process A

each Process will have it's own Process table

All this task is done by Kernel (OS in Admin Mode) (Creating Frames assigning Pages etc)

Windows is susceptible to viruses because it's a private OS and has to wait for their own dev to fix and release security updates.

However Linux is Open source.

So whenever a virus comes the dev automatically patches them.

So in general virus are created for windows so that they would pay them to get the virus removed.

. no need to load all the frames at once. You can load some subset into RAM.

#Virtual Mem & Demand Paging

When is demand paging used →

• Either your process is too big to fit inside the RAM

Or

• There are only few pages that can accomodate some of the frames. → leads to demand Pg

The remaining frames are stored in secondary Memory (Virtual).

#f	#p
1	2
2	5
3	
4	

3 is not present
So, 2 is sent to swap and 3 is given page 5 then after 2 is completed 3 is sent to swap and 4 is given page 5.

Whenever page fault occurs needs to swap from swap space. Since we are accessing the HD. it will be slow. So try to go with a cache in b/w. When cache is used it's called TLB When loop is broken down to frames.

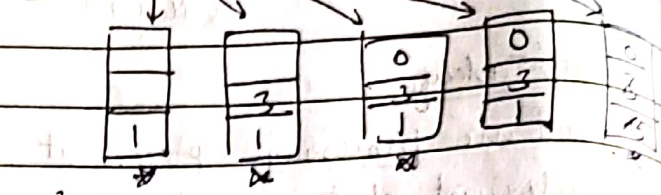
say 1 and 2 then you will be constantly swapping 1 & 2.

- Why we are swapping and not fetching?
  - During execution there may be a case if the frame in RAM gets changed so if we would have only fetched then the updated frame would have been lost.

- Cache stores the Most Recently used frames and swaps out the LRU into the HD.

Page Replacement Algorithms

Order: 1 3 0 3 5 6 3



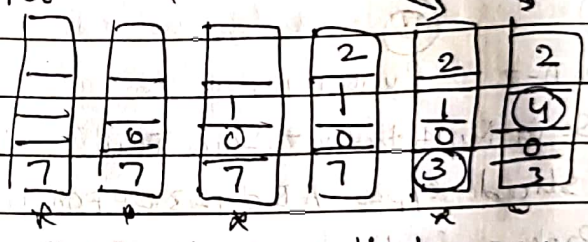
✱ → page fault

if FIFO used then 5 will replace 1.

if LRU used then 1 would be replaced

For LRU

7 0 1 2 0 3 0 4 2 3 0 3



Replace the page that was used most time ago.

- It would seem to that by increasing the frame size in FIFO → Faults will decrease. But this doesn't happen again Belady Anomaly

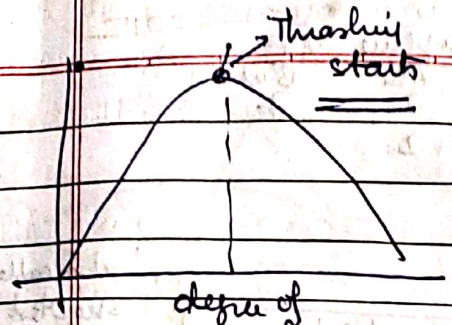
Optimal Algorithm → predicts future pages
  - But not practical.

Second chance Algorithm.

Thrashing

When the no. of processes are very high → then needs continuous swapping with the HDD. which further makes utilisation of Resources weak

also high degree of Multiprogramming leads to Thrashing.



Test submit →

Name → Testname → Task 1 → Task 2

LeetCode Weekly

0 0 0 0 0

10 -10

20 -20

25

10

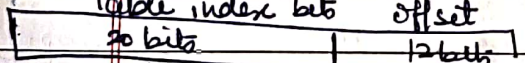
1

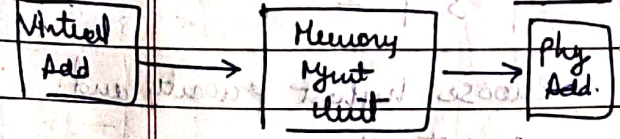
2 3

4 5 6 7

Operating Systems - 2

continue demand Paging





MMU maps the Virtual to physical

Date
Page

Tab 8th

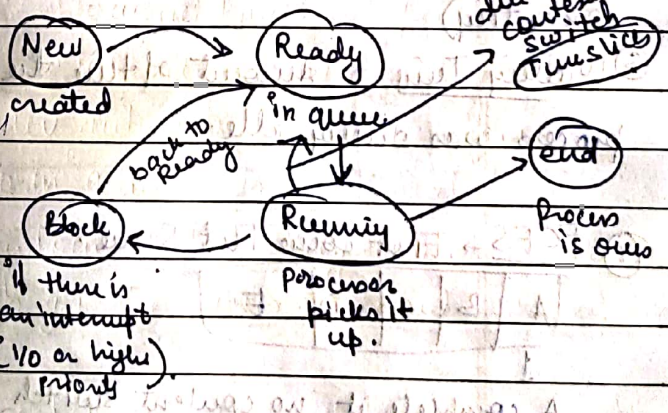
Virtual address is created at Runtime ??
↳ where it is stored ??

Since Process Page Table require 4MB contiguous memory but if we don't find it. We can break it down into parts and do similar like how we broke into frames.
This is called 2-level demand Paging or Multi-level demand Paging ?
↳ Read about this ??

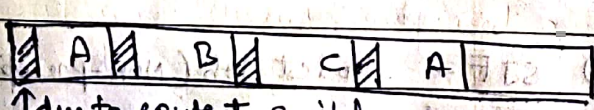
- Job Scheduling
  1. Non-preemptive → no interrupt is taking consideration
  2. Preemptive → interrupts are cared

Process States

5 process states



Types 1. Long Scheduler
2. Medium Scheduler
3. Short Scheduler
} look upon it



↑ due to context switch this time is taken by loading and deloading a process

1) CPU Utilisation
2) Throughput
3) Turn Around Time
4) Response Time
5) Waiting Time.

Date
Page

① CPU util → if there is no context switch CPU util decreases (Sitting Idle)

② Throughput → amount of process executed over a given interval of time (i.e that's why we give priority to shorter jobs so that throughput increases).

③ TAT → the moment it starts processing till the moment it's completely executed.

④ Response Time → The amount of time the process changes from ready to running (Time to take it out from Ready Queue and start executing)

⑤ Waiting Time → Amount of time the process was sitting idle.

① FCFS → First come First Serve.

A	B	C	D	E

A complete it no context switch (Non-preemptive).
then B the C ...
• each process gets turn to execute
Waiting time may be more.

② SJF → Shortest Job (Non-preempt).

A	B	C	D
5	7	1	12

Sort by their burst time.
FCFS in sorted order.

Preemptive SJF

	Arr	Burst
A	0	7
B	2	
C		
D		

A	A	B	A
0	1	2	

Context switch happened.

also called Shortest Remaining Time First

But practically it's hard to implement SJF because beforehand we can't predict the Burst Time of a Process.
• A may lead to starve because shorter process keep coming.
SJF is not used anywhere.
↳ gives Minimum Average Waiting Time

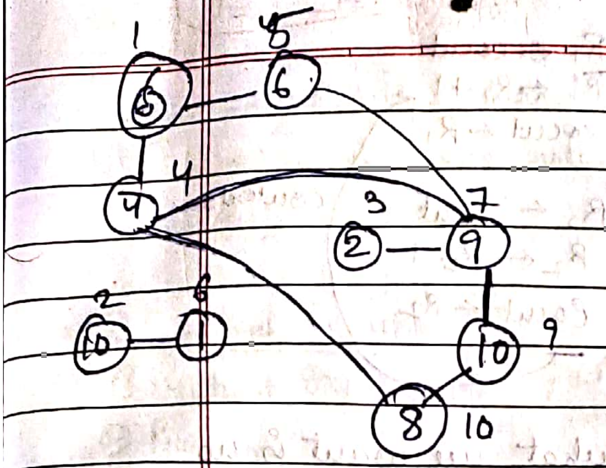
• Round Robin Scheduling.
give each process a quanta time
• Fair
• No starvation
• Better throughput
*choosing a better quanta is imp

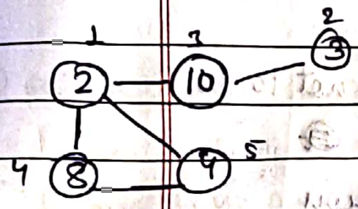
• Priority Scheduling :→

	Burst	Pro
A	5	3
B	2	1 →
C	3	2

• Choose highest priority and execute them.
• starvation

4+3+2+3+3+3





$x_3 \to B$

no 2 cons → x_1

no 2 cons → x_2

$x_1\ x_2\ x_1\ x_2\ x_1$ . . .

$x_2\ x_1\ x_2$ . . .

A (2)

$x_1 . x_3 . x_2$ — "only $x_1 x_2$ → (2)

if (x_2) cons.

$x_1\ x_2\ x_1\ 1$

$x_2\ x_1\ x_2\ 1$

x_3 — — → 2

4

$x_1\ x_2\ x_3$

$x_1\ x_3\ x_2$

x_3 — —

$x_2\ x_1\ x_3$

$x_2\ x_3\ x_1$

$x_3\ x_2\ x_1$

$x_3\ x_2\ x_1$

1 → 4 pp

1 1 → 3   2 2 → 7   9

2 1 → 6   4   3 2 → 16   96   3 3 → 17   23

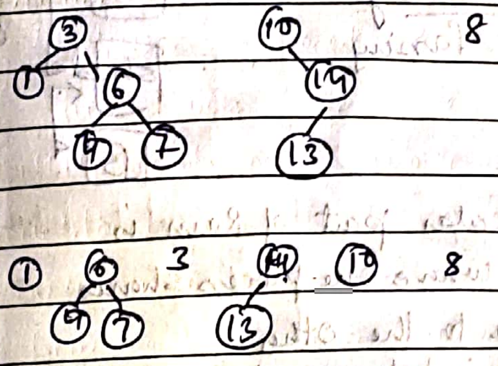
3 1 → 10   4   4 2 → 32   24) 8   4 3 → 40   48) 40

4 1 → 14   4   5 2 → 56   32) 8   5 3 → 88   88

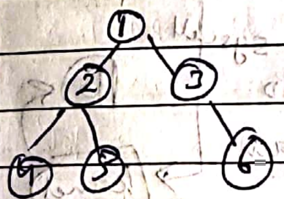
5 1 → 18   4   6 2 → 88   40) 8   6 3 → 176) 40

6 1 → 22   7 2 → 128   7 3 → 320   128

Q) Max diff in Node & Ancestor.



1 4 7 6 13 13 14 10 8



1 2 3

4 5 6

5 7 9 / 2 2 2 = 2 + 3 + 4

4 6 5 / 2 3 1 / 6 9 6 / 3 4 3

1 2 3 / 4 6 5

1 2 4 / 5 6 3 / 6 8 7 / 2 2 2 / 3 4 3

OS-2 continue

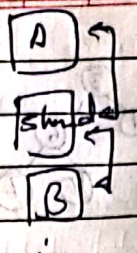
IPC → Interprocess Communication

- Convoy effect → due to a large process other have to wait for a longer time ∴ SJF shows least convoy effect.

Scanned with CamScanner

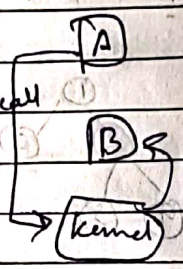
• If two process wants to interact with each other. It must be mutual from both processes.

1) Shared Memory
2) Message Passing

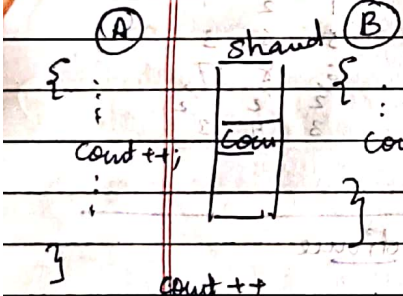


① • a particular part of Ram is shared where one process shares the data to the other.
Everything take place in RAM.

② A sends data to the Kernel via a syscall then Kernel sends it to the desired process again by syscall.
(Talking to the kernel).

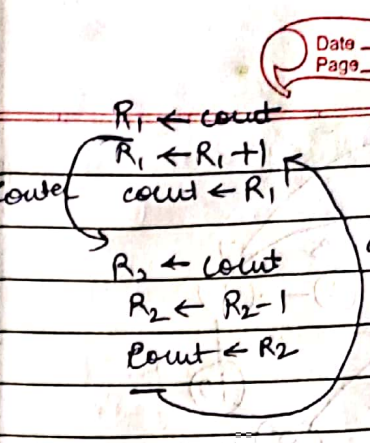


■ Synchronization :



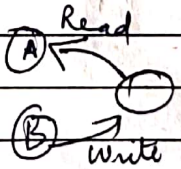
count++
⇒ R1 ← count
R1 ← R1(+)
count ← R1
→ (A).

if both A and B had been processed sequentially :
count = 0 (if only one)
• however say there was a context switch after a time slice and before completing A. B is completed then A.
then count can be random.

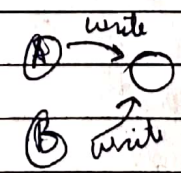


So what we want is no Context switch until A/B is finished →
• You can't ask not to do context switch.
• Atomic operation
set of instructions which when interrupted should be redone from the beginning;
(Either do it all or do again from start)

• Race conditions
if two processes are competing for same shared mem, whoever reaches 1st get to update.
WR Race WW Race.



1) A read first got 0 but B wrote 1 so, A thinks 0 but altered.



• A writes 1 then B writes 2. A thinks 1 but actually it's 2.

Take out Race conditions by applying concepts.

one turn variable
↓
turn

- locking concept.
  lock the shared Memory
  then do your job.

Critical Section → The shared memory where every process is racing for.

Critical Section →

1. Mutual Exclusion
2. Progress
3. Bounded Wait

} Must hold for CS to execute

1. only 1 process can enter CS at a time.
2. if there is no one inside CS, then the process which wants to enter CS must be allowed to do so.
3. any process which is waiting to enter inside CS must wait for a finite time. (It mustn't wait forever).

```
A                 B
{                 {
lock()            lock()
count++  } → CS   count--;
unlock()          unlock();
}                 }
```

Context switch will happen, it can't be stopped. But when B starts the lock will not allow B to use count as A has already placed a lock.
Now ans = 0 everytime.

```
(A)                          (B)
while(1){                    while (1){
  while (turn==2)              while(turn==1)
  {                            {
  }                            }
  //CS                         //CS
  turn=2                       turn=1
}                            }
```

this inf loop does allow A to change turn because B is currently process

this inf loop doesn't allow B to change turn because A is in CS

1. Mutual Excl ✓
2. Progress X → A must complete itself to let B enter into CS. but what if A doesn't want to complete, Progress is not guaranteed.
   A → B, B depends on A

How to remove the dependency?

a = false
b = false.

```
(A)                          (B)
while (1){                   while (1){
  while (b==true)              while(a== true)
  {                            {
  }                            }
  a=true                       b=true
  //CS                         //CS
  a=false                      b=false
}                            }
```

context.

1. Mutual Excl X
   both can execute CS together. Murphy's Law →
   "if there's something wrong then it will go wrong sometime or the other."

want_a = false
want_b = false

(A)	(B)

while (1) {
want_a = true;
while (want_b == true)
{
}
//CS
want_a = false
}
	while (1) {
	want_b = true
	while (want_a == true)
	{
	}
	//CS
	want_b = false
	}.

again no progress.
The above code can go into infinite loop
So both wants CS, but no one is getting.

add a turn variable which will fix all the problems.

(A)	(B)

want_a = true;
turn = 2
while (want_b == true && turn == 2)
{
}
//CS
want_a = false
	want_b = true
	turn = 1
	while (want_a == true && turn == 1)
	{
	}
	//CS
	want_b = false;

turn variable tells that we are favouring 1/2 process to enter.

Peterson Algorithm

MUTEX = Mutual Exclusion (1)

mex → shared variable.

(A)	(B)
lock (mex) //CS unlock (Mex)	lock (mex) //CS unlock (Mex).

lock() {	unlock() {

if (mex == 0)
mex = 1
else while(1){
if (mex == 0)
break
}
if some p is using CS.
others it will be in infinite loop.
	mex = 0;
	}.

optimization → instead of while loop use sleep()

lock() {	unlock()
if (mex == 0) mex = 1 else sleep().	mex = 0 then unlock() }.

}

all the sleeping fn are woken up and pushed to queue.

- Mutex can be unlocked by only that process which locked it. unlike semaphores.

- smallest Kth distance pair (1)

Leet → Dungeon Game →

l h mid
0 25 12

l h m
0 201 100

l h m
0 26 13
0 13 6

-2	-3	3	2
-5	-10	1	5
10	30	-5	6

-14 -24

4 → 1 → 4 → 5 →

[3 5]

l h m
0 4 2
3 4 3
4 4 4

1 -3 3 -3
0 6 -2 0 4
-3 10 -3 -3

goin bottom up manner and remember never to drop below 1. (Dungeon Game)

(Q) Find minimum in Rotated array in case of Repeated elements → ??

2) in worst case O(lgn) methods will take O(N) time. my simple O(N) leetcode solⁿ beats 100%.

question

• question pallete
• tries & submit

0 1 2 3 4 5
1 2 3 4 2 10

sum = 1 3 6 d = 3
sum = 5 7 d = 3
sum = 5 15 d = 3
sum = 12 8 6

l r
0 0

sum = 0
1 → 0 1
3 → 0 2
6 → 0 3
d = 3
5 → 1 3
9 → 1 4
d = 3
7 → 2 4
d = 2
4 → 3 4
6 → 3 5
d = 2
2 → 4 5
12 → 4 6
d = 2
10 → 5 6
d = 1
66

OS-3

Operating System - 3

if lock is not used, then race conditions would occur.

(A)
```
{
lock(mem);
count++;
unlock(mem);
:
}
```

(B)
```
{
lock(mem);
count--;
unlock(mem);
:
}
```

Atomic process / lock & unlock

```
lock(mem){
if (mem==0)
mem=1
else{ while(mem==1)
{
}
} mem=1
```

```
unlock(mem)
{ mem=0;
}
```

Scanned with CamScanner

• Instead of infinite loop make the process sleep so the processor doesn't waste on infloop.
Sleep → block state

```
lock(mem)
{ if (mem==0)
    mem=1
  else { sleep();
         mem=1;
       }
}
```

```
unlock(mem)
{ mem=0;
  wakeup();
}
```

all process that were slept are woken and sent to ready queue.

infloop → busy waiting
↳ the processor is busy but doing an unproductive job

Mutex have ownership. The process which has locked can only unlock it.

Semaphores

1) Binary Sema
2) Counting Sema.

Sema is more powerful. ↳ works on signalling
Sema has not ownership.

lock → mex 0→1
unlock → mem 1→0

• atomic operations are very small and fast. So that lock back time is very less.

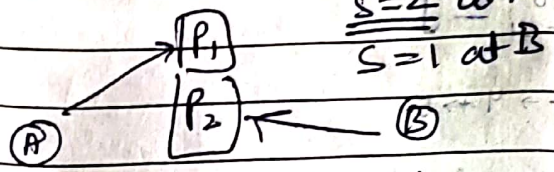
lock() → wait() → P()
unlock() → signal() → V()

```
wait(int *s)
{ while (*s <= 0)
  { }
  *s--;
}
```

```
signal(int *s)
{ *s++
}
```

If s is neg the don't do anything

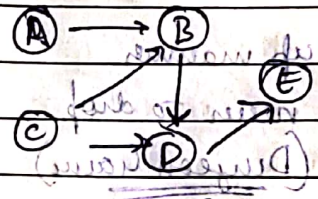
② Counting Semaphores
when there are multiple Shared Resources
Eg. Two Prin



S=2 at A
S=1 at B

wait:
C → S=0 at (
so wait()

How can perform a simulation of Topological Sort using Semaphores?



context switch will keep taking place.

So we will do signal from each dependent process.
Total Sema used would be no of edges → depends on

A	B	C	D	E
Sig(S1)	wait(S1) wait(S2)	Sig(S2) Sig(S3)	wait(S3) wait(S4) Sig(S4)	wait(S3)

wait infinite

if there would have been a cycle then it would lead to Deadlock.

Slight mistake ?? where
write values of S1, S2 and S3 you'll find S3 will wait inf.

So to fix this issue, just signal all those semaphores for which you were waiting.

Modified

A	B	C	D	E
sig(S_1)	wait(S_1) wait(S_2) : sig(S_3) sig(S_1) sig(S_2)	sig(S_2)	wait(S_3) wait(S_2) : sig(S_4) sig(S_3) sig(S_2)	wait(S_4) : sig(S_4)

Producer Consumer Problem

counting Semaphore to keep track of full Buffer and empty buffer
↓ Producer wait   ↓ Consumer wait

count → shared → mutex
full, empty, mem

```
produce(){
while(1){
 item ← produced
 if(count==N)
   sleep(full)
 lock(mem)
 count++;
 unlock(mem);
 if(count==1)
   wakeup(empty);
}

consume(){
 if(count==0)
   sleep(empty)
 lock(mem)
 count--;
 unlock(mem);
 if(count==N-1)
   wakeup(full)
}.
```

→ make these atomic

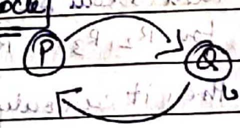
- if producer sleeps the full → consumer wakes it. and vice versa.

can we face problem if Context switch? → make the marked parts as atomic all

We can do this using Semaphores.

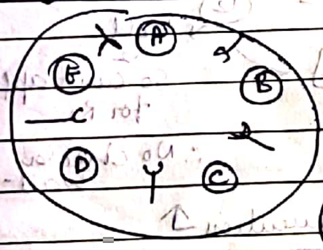
```
producer{
while(1)
 item ← produce
 wait(empty);
 lock(mem);
 count++;
 unlock(mem);
 signal(full);
}

consumer(){
while(1){
 wait(full)
 lock(mutex)
 count--;
 unlock(mem);
 signal(empty);
}.
```

Dead Lock



↳ Stuck. Nothing can be done

- Dining Philosopher

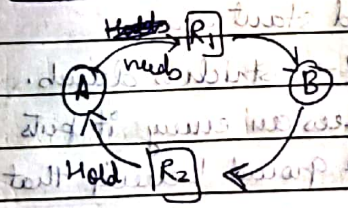


① Think
② Eat.

2 forks needed to eat

- in a greedy state → deadlock

- Resource Allocation graphs



→ deadlock no one can progress

Condition for Deadlock

① Mutual Exclusion → Only 1 process can use a R_1
② Hold & Wait → P holding R_1 and needs R_2.
③ No-preemption
↳ You can't take away a Resource from a Process.
④ Circular Wait → ① ⇄ ②

So to prevent deadlock avoid any one of the following →

1) Can't Remove Mutual Exclusion because you can't allow multiple P. to hold same R.

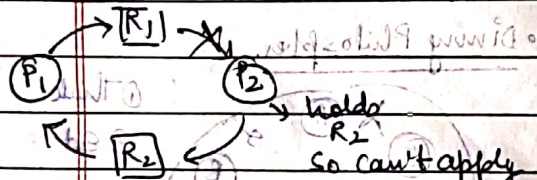
2) Hold & wait → eg: if a process needs R, then execute that P. by first giving R to it.

3) · include preemption and forcefully give the holded Resource to another P.

4) to avoid circular wait do a hierarchial ordering.

P_1 holds R_1 → then it can only apply for $R_2, R_3 \ldots R_N$.

P_2 holds R_2 → then it can only apply for $R_3 R_4 \ldots R_N$.



∴ No circular wait.

Deadlock Prevention ↑

• Deadlock avoidance

basically normal systems follows Ostrich Algo → Restarts the system and start again.

Why ostrich ? → Ostrich is dumb. whenever it sees an enemy it puts head in the ground believing that if it can't see the enemy, the enemy won't see it either. Hahaha!

But important systems like Banking, Aeroplane etc should never ever allow deadlock to occur.

Date
Page

• Detection of Deadlock

make a Resource Allocation graph and find whether there exists a cycle or not.

• Banker's Algorithm

↳ deadlock Avoidance Algo

try not to enter the unsafe one. try to skip it. Read this

 # Deadlock Recovery

1) Preemption → snatch the Resou
2) Kill the process → as name say
3) Roll Back → ↳ keep saving the state of the process in HD and whenever a deadlock occurs, load that past where no deadlock and then do some changes so that same timeline doesn't repeat.

 # Reader & Writer Problem

only 1 writer can write
↳ use mutex.

and multiple Readers can Read → use Semaphore.

also, you can't write if someone is Reading.

Read More.

• Remove duplicate letters – Hard
Greedy.

c b a c d c b c .

c → 4 3 2
b → 2 1
a → 1 0
d → 1

| b | d | c | a | (7 2 5) 10 8

10 32 mid
l h 21

GUI | Terminal

Services

OS

Hardware

• when you click a folder then GUI takes control and calls a service who (show contents). Services do a specific job. GUI → service → OS → Hardware (Harddisk).

• You want to play songs before going to sleep and then after 30 min the PC should shutdown.

• Everything is File even dir's

• the commands does one thing well. cat → shows the content of file

• ls → list • ls -l long list (l → flag)

• touch 1.txt → creates the file.

• rm -rf → force all everything inside

echo "a ____" > todo.text

Task → segregate files on year they are created?

a basic script

for i in 'ls'; do echo $i; done

var = "2018-05-01"

echo $var | cut -d '-' -f 1

-d is delimiter, back ticks

years = $(for i in 'ls'; do echo $i | cut -d '-' -f 1; done);

LINUX, GIT and CLOUD

Product Engineer

'Product Hauge + Software Eng' = Product Eng

• Work with ambiguous problem statement. after applying the correct algorithms, measures the change in performance. Never blames the user. Instead finds what can be done better from product's side.

• Backend Engineer

• works with well defined problems.
• works on optimising systems

1) Week 0 → Tools + Framework

• Git • Linux and Bash
• AWS • Flask • React

2) Week 1 → Team → Project → API → Front End

③ Week 2 → Consume Frontend → Deploy

④ Week 3 → Improvements

for year in `echo $years`;
do
    mkdir $year;
done

bash ... .sh

for img in `ls *.png`;
do
    year = `echo $img | cut -d '-' -f 1`
    mv $img $year/
done

Task 2 → sleep after playing music

Task 3 → download all wallpapers from a site.

1:33 resume
1:41

- git init
- git add .
git log -n 5 → who commited. what. with version

if multiple devs work → clone code to other's system and they cannot push.

Fork → copy to another remote

github feature

How to pull?

- git reset --hard
    ↳ Version No.
    ↳ roll back to a previous version

Advanced GIT and MVC

Date
Page

Task: Calculator in python
- add - subtract - mult - div
Two guys are implementing

How to merge these two files
① copy paste → humanen
    ↳ resides in laptop
② Drive
③ docs → have version control
    ↳ collaborative

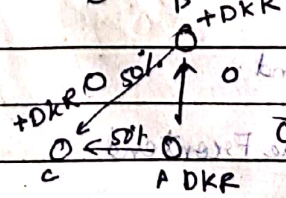
- if you are not sure about your code and needs reviewing by others.
create a branch and push in that branch. In this case the master branch remains stable.

git checkout

merge conflict??
↳ collaborative tool
+ behavioral.
It encourages TeamWork

- Video + Flask
- Hosting
- 8 Questions (Med + Hard)
- create a to-do app in Flask.

Peer to Peer Model
↳ many clients/server



BitTorrent → Torrents system

View ← todo_view → gets a list → print in HTML2
Model ← get_todos_by_name → returns todos for each
Controller ← todos → calls the above function accordingly