Delta

CEC14 - Operating System

- Operating system → basic electronic circuit → application
  - ↳ software that manages hardware & software
- basic electronic circuit → fetch instruction & trigger some electronic circuit to execute them
- manages hardware using stored program concept and operating system

1940's (1946) ⇒ Von Neumann proposed stored program concept

1950's (1958) ⇒ Jack Kilby created chip (2 components) co-founder of Texas Instruments

1960's → Moorey decided to club both & make a microprocessor (founder of intel)
4004 was created
First full fledged microprocessor → 8085
then 8086 was made.

Then first OS was made Disk OS (DOS).

Now if we had Disk, Memory, chips etc. etc
so a Control Program ~~was~~ (OS) was decided to be written.

directory → file → Memory } manage them
disk ⇓
OS.

Scanned by CamScanner

Delta

Bill gates proposed GUI based OS → Windows 3.0

Dos → Commands but windows → graphic based

Now Graphic card also came → manage them also
Microsoft was created.

(OS → disk to memory)

Texas Instruments → chips
Intel → Microprocessor.
Microsoft → Operating Systems.

from 2 components on chips to billions of components

Hardware → complex (powerful) Microprocessor → complex

OS → match up (software has to catch up hardware to manage it).

pipeline was created (No of stages ⇒ No of instructions)
386.

486 → More than 1 pipeline.
↳ pentium. (20 pipeline)

Windows - 3.3 X.
Windows NT → allowed multiprocessing.

now → process scheduling, memory managing
↑ ↑
because of so many tasks at one time. so than 1 memory program in 1 memory

Scanned by CamScanner

Delta

1998 → windows 98 (memory on chip)
2000 → Windows 2000 (previous different chips)

Now multi pipeline + multiple processor

One chip → more than 1 processor (2 processor)

OS → A program which is executing (inside electronic circuit) → process

Now more complicated → because program in which process? then in which pipeline?

Now software architecture evolved → object Oriented, api's etc.

Now OS has to be written by high level language DOS → machine language.

Kernal → core portion of operating System. now decided to create micro-kernal.

Now networking & internet also came into picture.

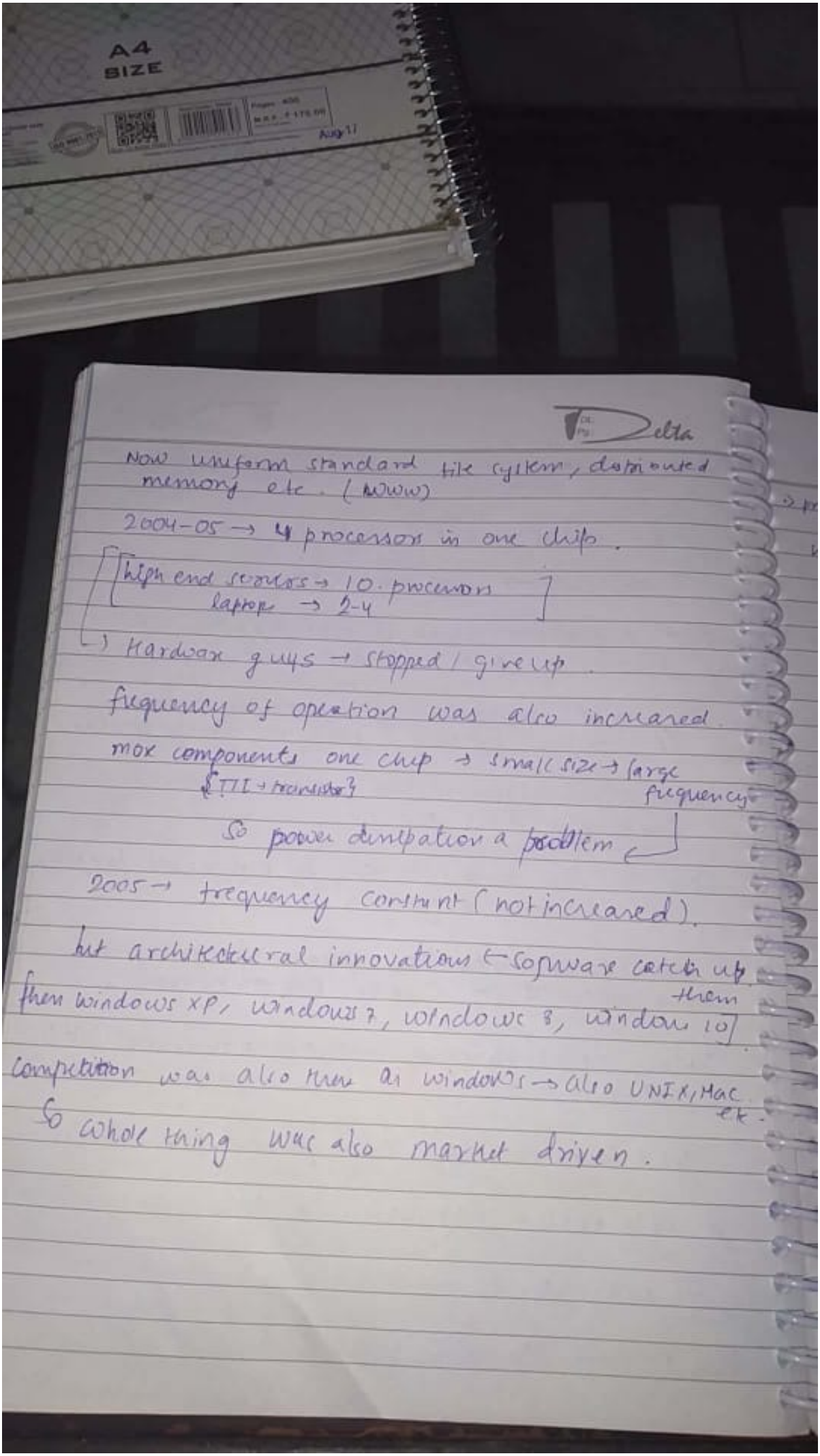
1961 → first message was passed b/w computer

p. networking parallel to hardware & software

OS matched to networking

Then in 2000s Tim came into picture → www world wide web.

Scanned by CamScanner



Delta

Now uniform standard file system, distributed memory etc. (WWW)

2004-05 → 4 processor in one chip.

[High end servers → 10. processors
laptop → 2-4]

↳ Hardware guys → stopped / give up.

frequency of operation was also increased.

max components one chip → small size → large frequency
[TTI → transistor]

So power dissipation a problem

2005 → frequency constant (not increased).

but architectural innovations ← software catch up. them

then windows XP, windows 7, windows 8, window 10]

Competition was also there as windows → also UNIX, Mac etc.

So whole thing was also market driven.

Scanned by CamScanner

Delta

- process → program in execution

when we double click :

from disk ⟶ main memory
(electro mechanical)
↑
instructions fetched from there

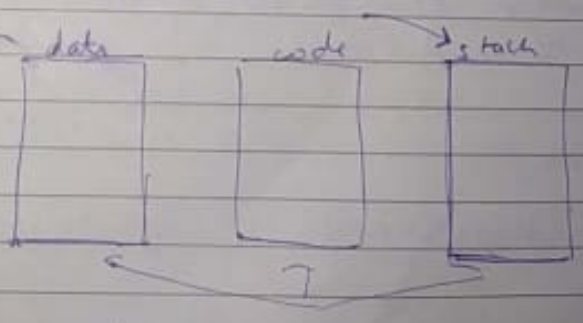
more than 1 CPU
in 1 CPU → many pipelines
(multiple instructions can be run).

many programs running at same time

there should be decider who will decide which program will go in memory etc
that decider ⟹ Operating System

- each process → Process id, status → for current execution
PC, & General purpose registers
(priority) B C H L D E ---- },
SP (Stack Pointer)

there are base register/segment register for each segment.



memory add
↑
in Cache (general).

for next instructions.

Scanned by CamScanner

A4 SIZE

Delta

all these things are stored as os needs it
because there when new process is started
then some process has to be kicked off so
its status would be needed
↳ stored in memory / special cache. (look aside buffer)
every process has Process control Block (PCB) [stored in memory]

frequencies of computer → 10^9 gigahertz → 10^9 Hz
So in 1 second → 10^9 process
nano seconds
at split of 1 nanosecond ⇒ 1 process per CPU unit for compute

So delays are in microseconds → we don't realise it.

Sometimes → delays in seconds ⇒ when?

- PCB (process control Block) ⇒ every active process has.
not stored in memory → cache (look aside buffer)
(delayed (fast)

then CPU → nanoseconds
Memory → microseconds. [slow]

one

When new process is started ⇒ one PCB → out, new PCB → in } context switch (minimal possible time)
↓
Overhead for OS.

Task Manager → one process ⇒ CPU clocks, memory (time) resources used by process for accounting purposes.

Called House keeping.

Scanned by CamScanner

Delta

House keeping → open files, devices, network port open

program → enter → Ready → Run

OS stores status → ready in the PCB

Queue (highest priority front) process will run

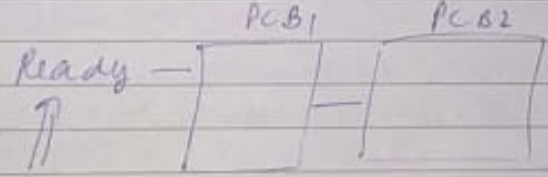
Long term scheduler → from secondary memory → main memory.

When double clicked

Short term scheduler (dispatcher) → only one PCB. Ready → Run (in kernal)

(There can be multiple queues based on schedules.)

Queue ⇒ because there could be more than one process in ready status (could be priority queue)



All progs are ready in memory but one in CPU.

→ Response time → time for difference between response (when CPU gives) & when we request

↳ time from submission to first response

→ turnout time → time from submission to completion.

Scanned by CamScanner

response time
turnaround time

Delta

Scheduler
- one procss complete → then 2nd procss
- round robin → take 2nd procss for some time
  - every procss is given some time then context switch takes place
  - time-out (and procss runs after some time on 1st procss is turned out)

for real life feel

Response time ↓, turnaround time ↑

all procss in round robin is premptible (can be interrupted).

only a small part of OS is non-premptible not all (only OS will Run).

non-premptible part of OS is called Kernel.

previously → monolithic kernel (all heavy functions) ⇒ only OS Running & problems

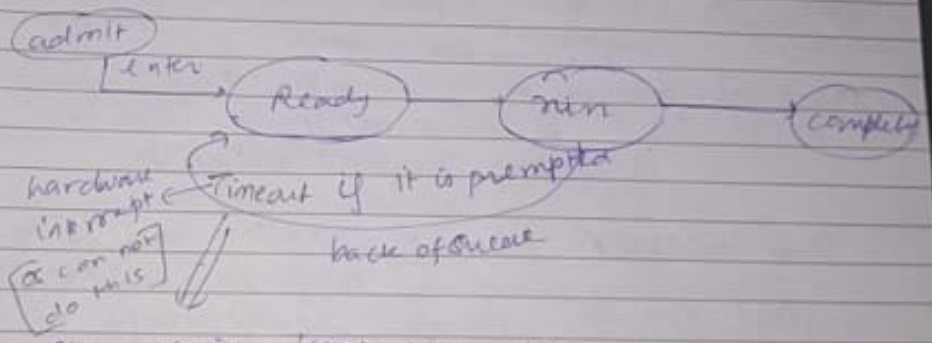
now → micro kernel
- (only few functions)
- based on Object-oriented approach
- focus in modules
- → procss context switch (most imp)
- → procss scheduling
- → thread scheduling → kernel thread scheduling
- → exception handling
- → interrupt handling
- → basic memory address management

Scanned by CamScanner

Delta

- OS-functions: file management
  device management
  Memory management
  Process scheduling
  → many task to run together etc.

- (per process). State diagram.

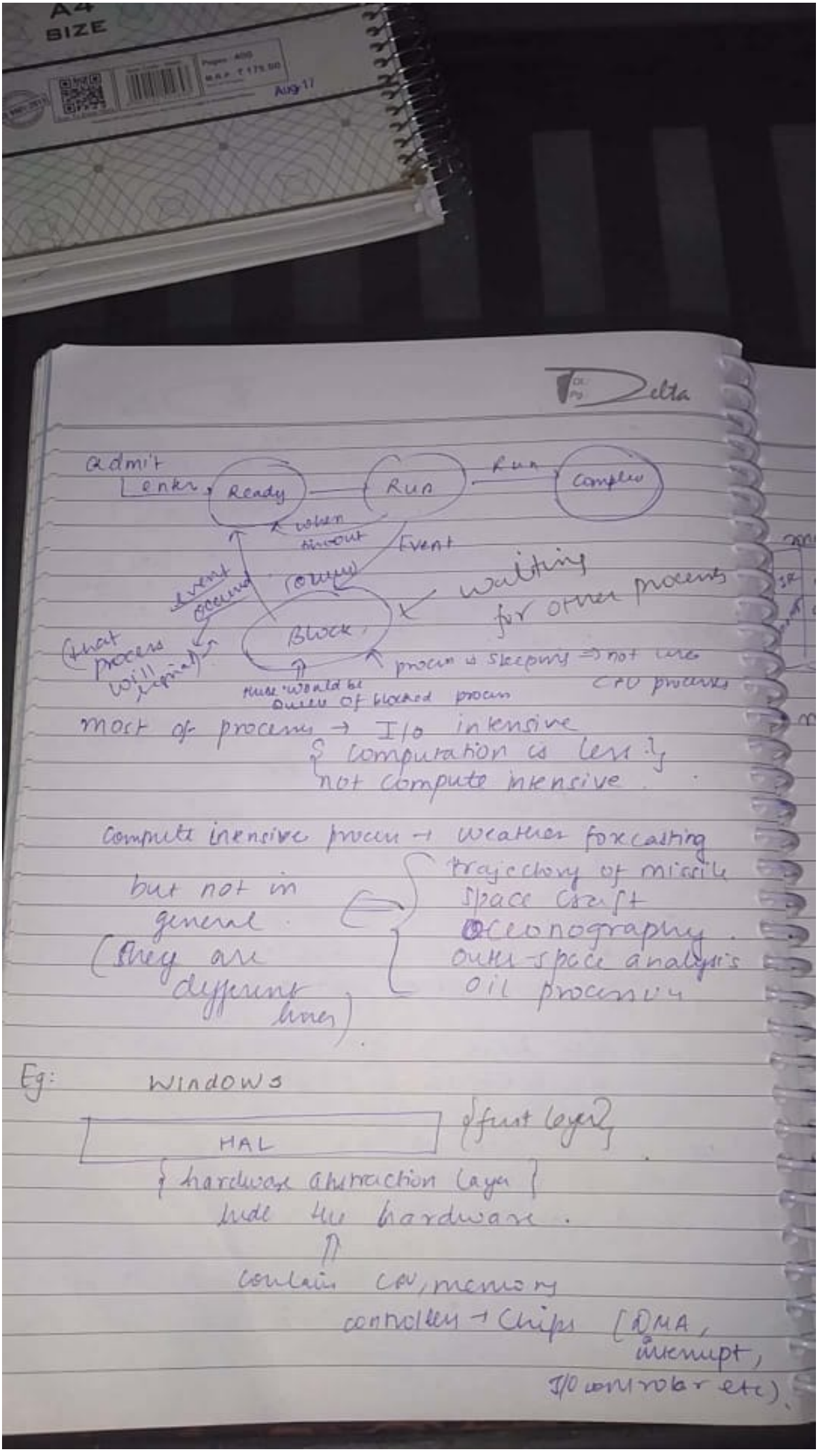


Other things that can happen
- wait for I/o device
- one other processes.

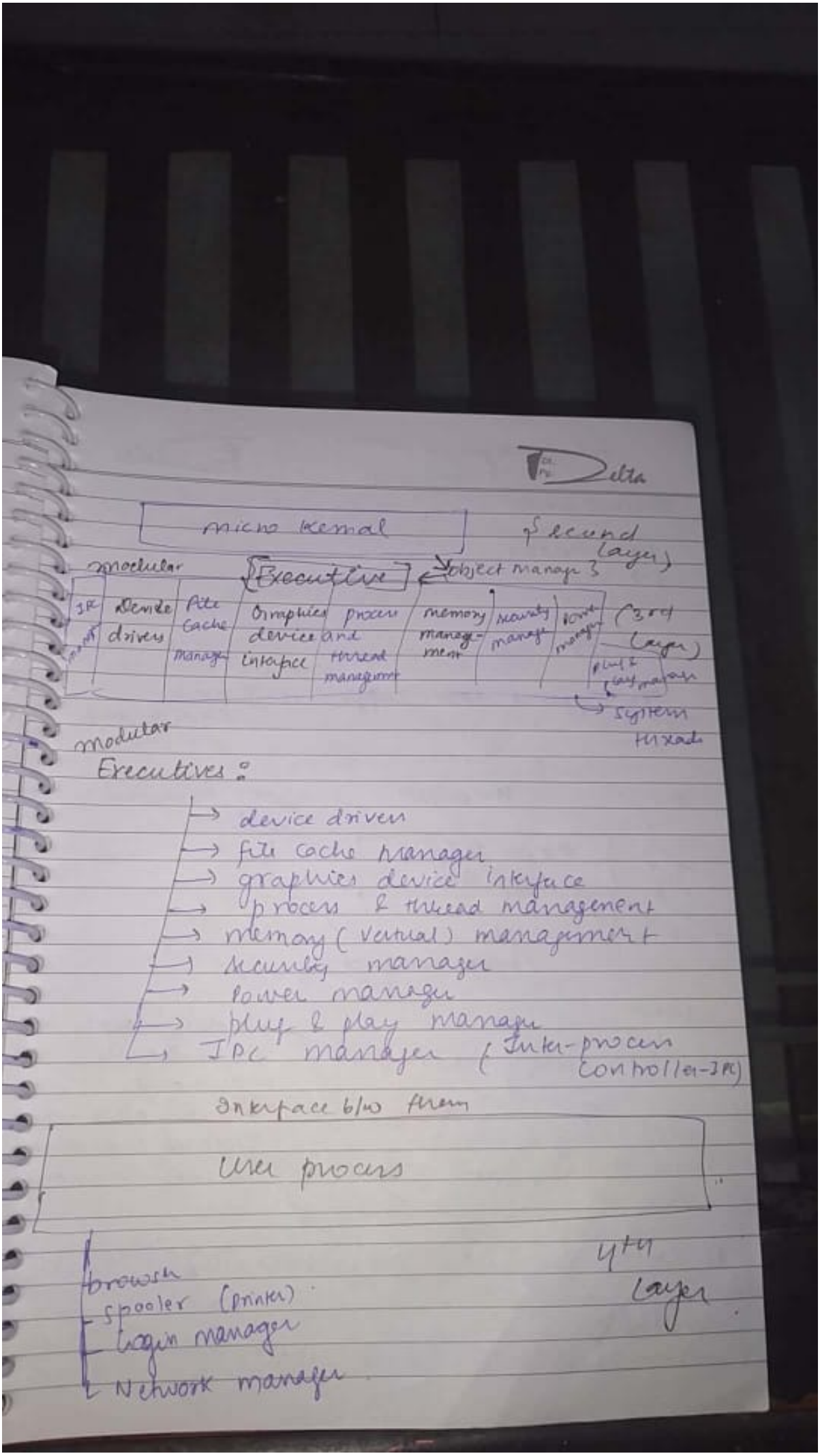
There can have to be Interprocess communication (IPC).

one process depends on many other processes and have to wait for them
If waiting for event
printer to get my print
Other process to do something.

Scanned by CamScanner



Scanned by CamScanner

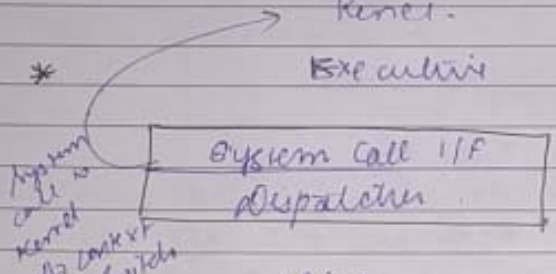


Scanned by CamScanner

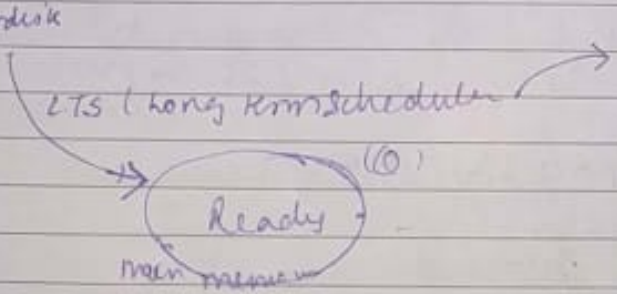
operating systems same // different Environments

- Windows
- Dos
- Linux
- OS/2
- POSIX

same process can support different environment



User
(browser, spooler etc).



but can also all processes can come in main memory

IPC → LPC (Local Process Controller)

→ RPC

(Remote Process Controller).

Scanned by CamScanner

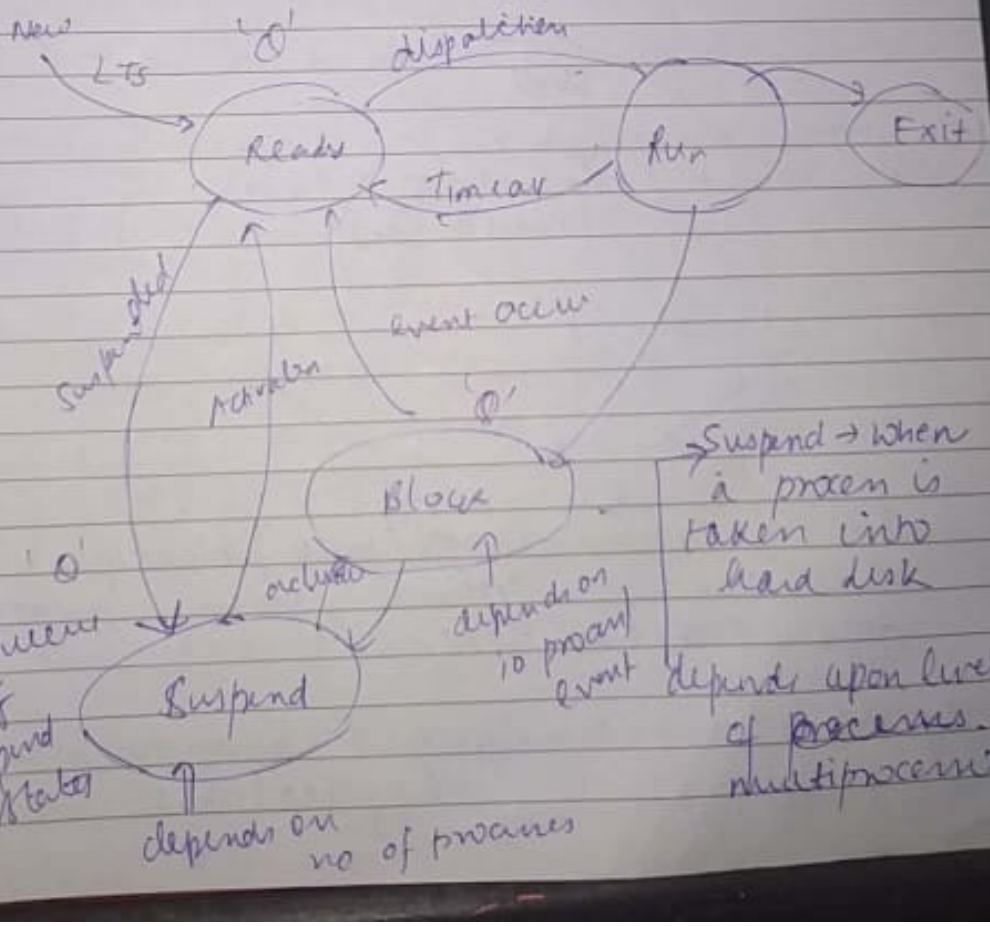
Delta

LTS can not bring all process in memory at the same time => Memory management comes into picture.

Swap it out to hard disk

OS has to think hard disk as an extension to main memory

Virtual Memory => When hard disk is percieved as main memory.
↑ electronic
↑ time taken more (electro-mechanical)



Scanned by CamScanner

A4 SIZE

Delta

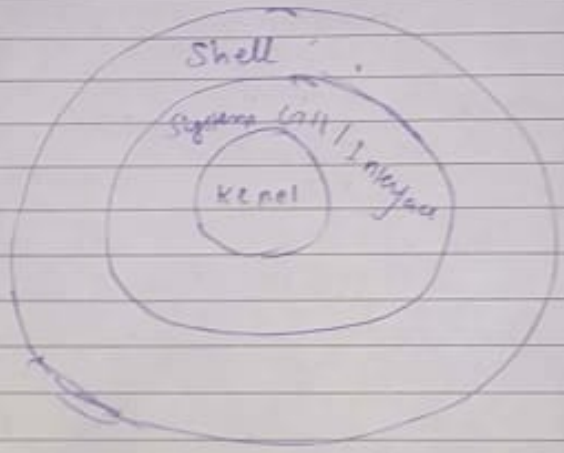
Suspend → not only transferring PCB but entire program (much longer)
Context Switching → transfer PCB to special cache

Suspend, timeout ⇒ overhead ∴ = much time wasted required

So paging, segmentation helps in it.
not whole program but its part in suspend

UNIX architecture

represented in circular chart



System Call / Interface → The calls / interrupts that interrupts program, when kernel does context switching.

Shell → UNIX commands & libraries & user entry of user into OS. applications

Scanned by CamScanner

Delta

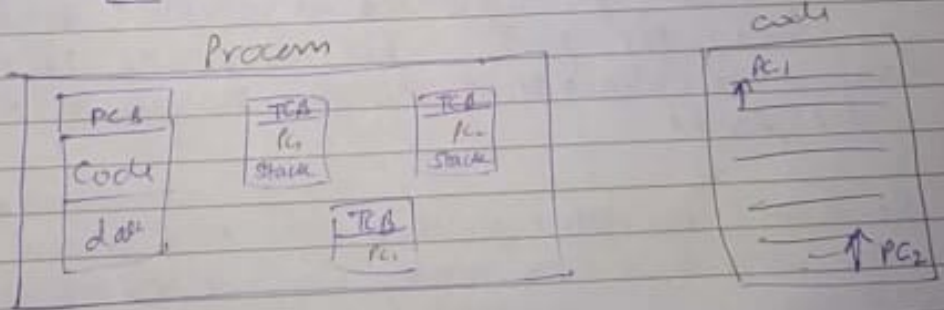
| → pipeline
Output of first ⇒ input of next.
>>, << → redirect
(from one stream to another)

⇒ so many processes → one process depends on many other → Cost of context switching → slow.

So we replace it by **thread** (light weight process) → it is a process but do not have ownership of process resources (IO channel, files opened, devices).

One resources can be given only to one process

Thread has PC, SP, TCB (Thread control block), (shared code, shared data, individual data)

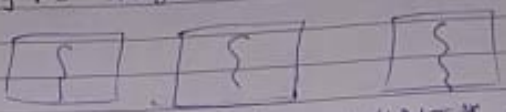


Copy/Paste → sharing the data.

Stack → for individual data.

Scanned by CamScanner

UNIX initially :- only one thread per process

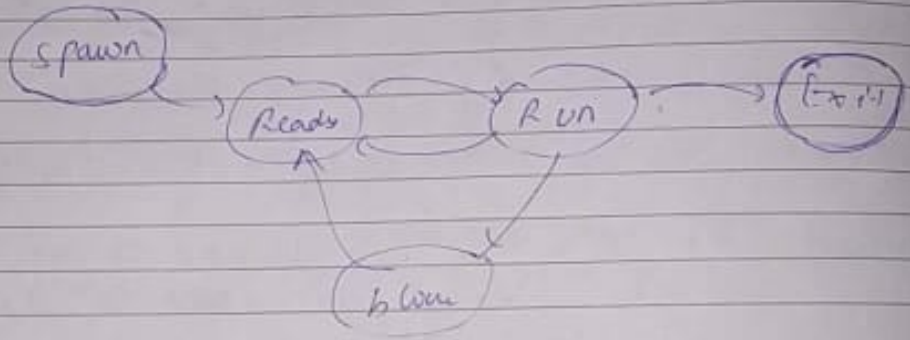


after but library pthread → UNIX also had multithreading

thread spawn other thread

Windows, Mac, os2 → were multithreaded

UNIX ⇒ fork ⇒ spawn new threads.



same state ⇒ differences ⇒ double click is entry for process, but threads spawn/ Create other threads ⇒ within program it happens.

Kernel → multiple process → each process ⇒ multiple threads handled

User threads ⇒ by library.

OS do not handle.

no ownership of resources ⇒ context switch is less.

Scanned by CamScanner

Delta

time reduced in context switching
only registers are to be stored as data/code are shared.
If a thread switching → managed by library ⇒ not OS

now library manages → OS/kernel not in picture (one process saved?
ownership not there → open files (not needed)
time out for process is same
but multi-threading ⇒ asynchronous processes (advantage)
at one thread in foreground others in background.

Inter-thread communication is easier than inter-process communication

eg word pad → one thread is saving periodically your work

Scheduling of threads can be done by user

Disadvantage

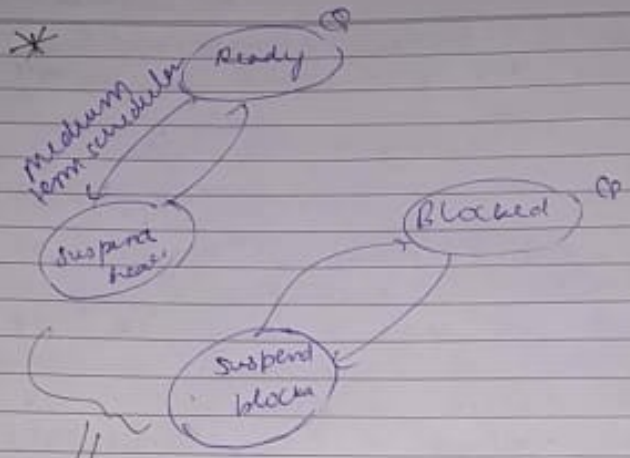
a) When process gets blocked → all threads are blocked. (thread blocked → process not blocked)

b) process gets same amount of time

Scanned by CamScanner

A4 SIZE

Delta



So that it can give return back. So two states.

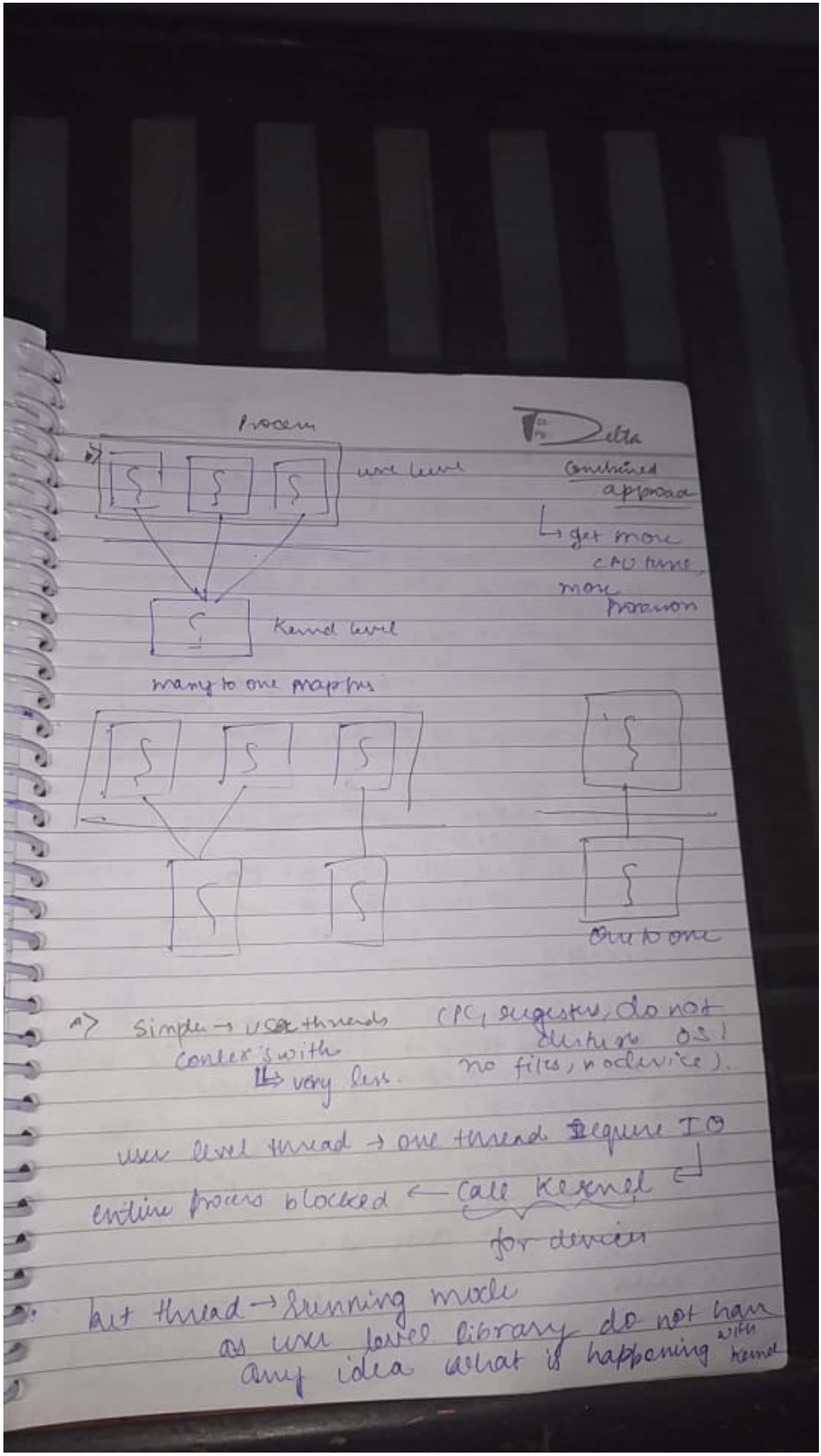
•) Kernel threads has very less efficiency as it is a different program only & a separate context switch.

If User threads → 10 µs
Kernel threads → 1000 µs

•) Two threads can not have different processes as only OS can do it → Disadvantage of user threads

Kernel threads → scheduling across different processes, more to CPU time etc → but context switch is slow.

Scanned by CamScanner



A) Simple → user threads context switch ↳ very less. (PC, registers, do not disturb OS! no files, no device).

user level thread → one thread require I/O entire process blocked ← call kernel ↵ for device

bit thread → Running mode as user level library do not have any idea what is happening with kernel

Scanned by CamScanner

Delta

- process → 2 threads
  - one thread running
  - other thread ready
  - } user level libraries

require IO → have to go to kernel → as kernel is there so entire process gets blocked.
- but thread1 → run nii, & as user level library do not have any idea what is happening with kernel
- thread2 → ready

after IO → again to ready (process)

→ then when it will come back to run state
- thread1 → will get IO
- continue thread on user level.

Schedulers

- (w) wait time → in ready Queue
- (e) execution time → time spent till now
- (s) → time needed by processor (CPU time).

service time

three schedulers → longterm, short term, medium term

response time, turn around.

in IO based (less in)

batch mode processing as in (less in). real time systems

Scanned by CamScanner

CPU utilization → time CPU is in use

Delta

In real time systems → tasks in dead line
(3 categories) is there

→ soft real time system (deadline can be {Computers} stretched)

→ hard real time system (deadline is imp)
{ embedded system + surgery, aircraft)
deadlines can not be missed
Scheduler gurantees that deadlines are met

↳ (edf) (~~earliest earliest~~
Scheduler { Earliest deadline first
who gurantee Rate monotonic scheduled (RMS)

→ firm real time system → b/w soft & hard.
y-m at a given time + out of m deadlines → k deadlines should be met
{ conveyor belt, rover to Mars)

Another performance parameter

↳ throughput → transaction oriented system

no of transactions/ processes ~~end~~ per unit time

Eg → railway reservation system

TPS → transactions per second.

→ Scheduler → schedules ~~processes~~ processes pass based on different performance parameter e, w/s, response time, turn around, through put, energy consumption reduction etc.

Delta

avg energy consume consumption reduction → minimization of energy consumption of device
→ solar panels, mobile in sleep mode when battery goes down or not using

a) if reduce voltage, all process will take more time
as at low voltage switching takes more time (electronics)

=> another parameter → predictible manner. an Scheduler ensures that the completion time is predictible
execution of same program multiple times → run in same time.

Execution time varies → due to multiprocessing
but we need to reduce the standard deviation.

First Cum First serve Scheduler (FCFS)

Process which is in ready Queue for longest time will be Scheduled first
while waiting time is maximum (Wmax)

Non preemptive algo → Other can not enter till first is completed (it is dispatched)
STS (short term) → dispatcher

Scanned by CamScanner

Process	(t arrival)	service time	(t completion)	Turnaround time	normalized turnaround time (Delta)
	t	S	t_c	T_r	T_r/S
T_1	1	1	2	1	1
T_2	2	100	102	100	1
T_3	3	2	104	101	50.5
T_4	4	200	304	300	1.5

FCFS → prefer longer tasks over scheduler shorter tasks
- if many shorter tasks turn around time suffers.
- favours longer tasks, penalizes shorter tasks.
- unfair to IO oriented tasks, as compared to processor oriented jobs
  - as any task needs IO → it will go to blocked state → then at tail of ready → wait for all the long tasks → response time high

Solution → new blocked queue
- two → ready → first time ready
- block → second blocked (higher priority)

Virtual FCFS
- When it tasks go is dispatched it is first taken from blocked queue.

Scanned by CamScanner

A4 SIZE

Delta

Long term

degree of multiprogramming (no of process in main memory)

Medium term

Swap out some process from the main memory

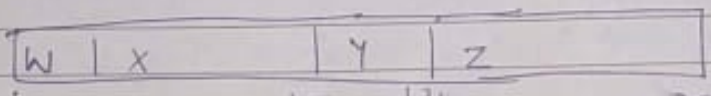
short term

assign the CPU to one of there processes (also known as dispatcher)

FCFS

Process	AT	Burst/Service time	StartTime	FinishTime
W	0	1	0	1
X	1	100	1	101
Y	2	1	101	102
Z	3	100	102	202

GANTT CHART



(TAT/S)

TAT (Turn Around Time)	Normalized Turn around time
1	1
100	1
100	100
199	1,99
average 100	26

Scanned by CamScanner

p shortest jobs suffer.

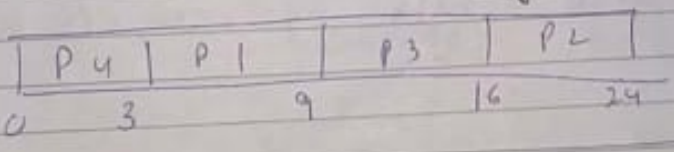
Convoy EFFECTS: FCFS prefers processor bound processes over IO bound processes. When large no of IO bound processes are waiting for 1 large processor bound process to get over leads to inefficient processor & processor utilization.

Shortest Job first Algorithm (SJF):

Optimal in sense → it will give lowest waiting time.

Process	Service time / Burst time	Waiting time
P1	6	3
P2	8	16
P3	7	9
P4	3	0

avg - 7.



⇓

assuming all processes arrive at same time

When ~~when~~

Example of non-preemptive SJF

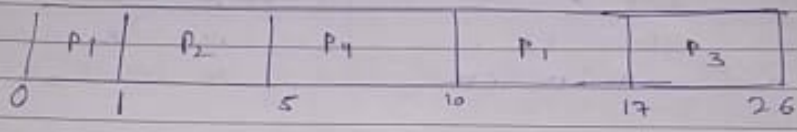
when at different arrival time →

Scanned by CamScanner

Shortest Remaining time :

Process	AT	BT (Burst time)	Waiting time
P_1	0	8	10-1 → 9
P_2	1	4	0
P_3	2	9	15
P_4	3	5	2

$avg \rightarrow 6.5$



At every point we have to check for shortest remaining burst time for every ~~proc~~ process currently in ready Queue and execute it.

Example of preemtive SJF also known as shortest Remaining time Next

Determining the length of next CPU Burst.

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$$

t_n = actual length of n^{th} CPU burst

τ_{n+1} = predicted next CPU burst

$0 \leq \alpha \leq 1$

Scanned by CamScanner

Delta

if $\alpha = 0$

$\tau_{n+1} = \tau_n$

if $\alpha = 1$

$\tau_{n+1} = t_n$

↳ adjusted according to giving process no (t_n) actual nth CPU burst or com previous ~~history~~ calculated history (τ_n) of burst time

Priority scheduling

Priority more → scheduled first

may be fixed on any parameter

if on time taken by priority → becomes shortest Job first (SJF).

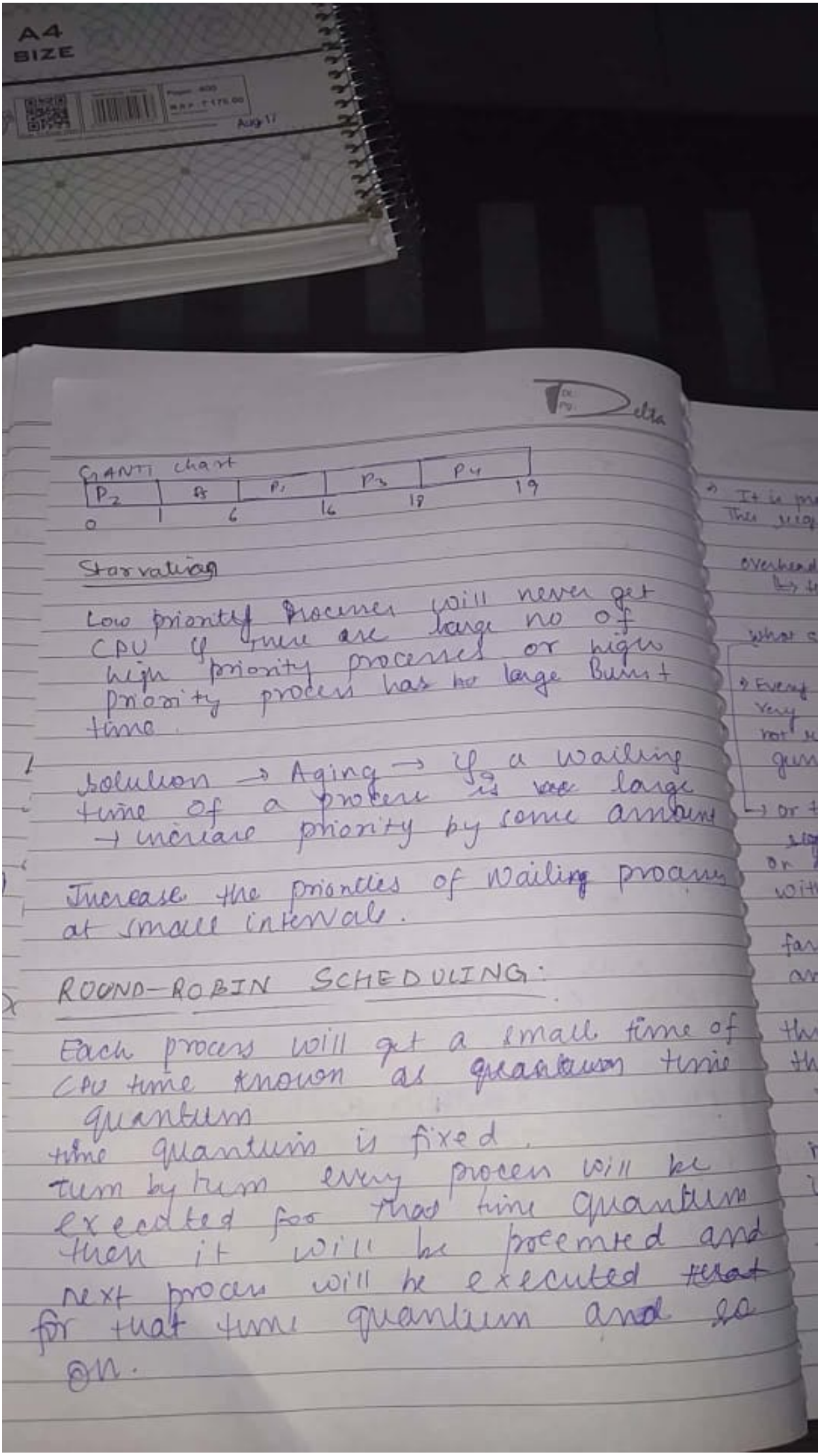
if same priority of two process → process arrived first will be executed first

1 → highest

Process	BT	Priority	Waiting time
P_1	10	3	6
P_2	1	1	0
P_3	2	4	16
P_4	1	5	18
P_5	5	2	1

avg 8.2

Scanned by CamScanner



GANTT chart

P_2	P_3	P_1	P_3	P_4

0 1 6 16 18 19

Starvation

Low priority processes will never get CPU if there are large no of high priority processes or high priority process has so large Burst time.

Solution → Aging → If a waiting time of a process is very large → increase priority by some amount

Increase the priorities of waiting process at small intervals.

ROUND-ROBIN SCHEDULING:

Each process will get a small time of CPU time known as quantum time quantum

time quantum is fixed.

Turn by turn every process will be executed for that time quantum then it will be preempted and next process will be executed for that time quantum and so on.

Scanned by CamScanner

Delta

⇒ It is preemptive form of First-cum-First serve. This requires external clocks to preamt processes.

overhead of context is there
↳ timer interrupt after some time.

what should be time-slice?

⇒ Every process will have less response time
very good for time-sharing system. like laptops.
not real time scheduling algorithm → no
guarantee that deadlines will met.

↳ or time Quantum??
slightly higher than average service time.
or average → any all process will complete
within one context switch.

fair treatment → response time of shorter
and longer jobs same

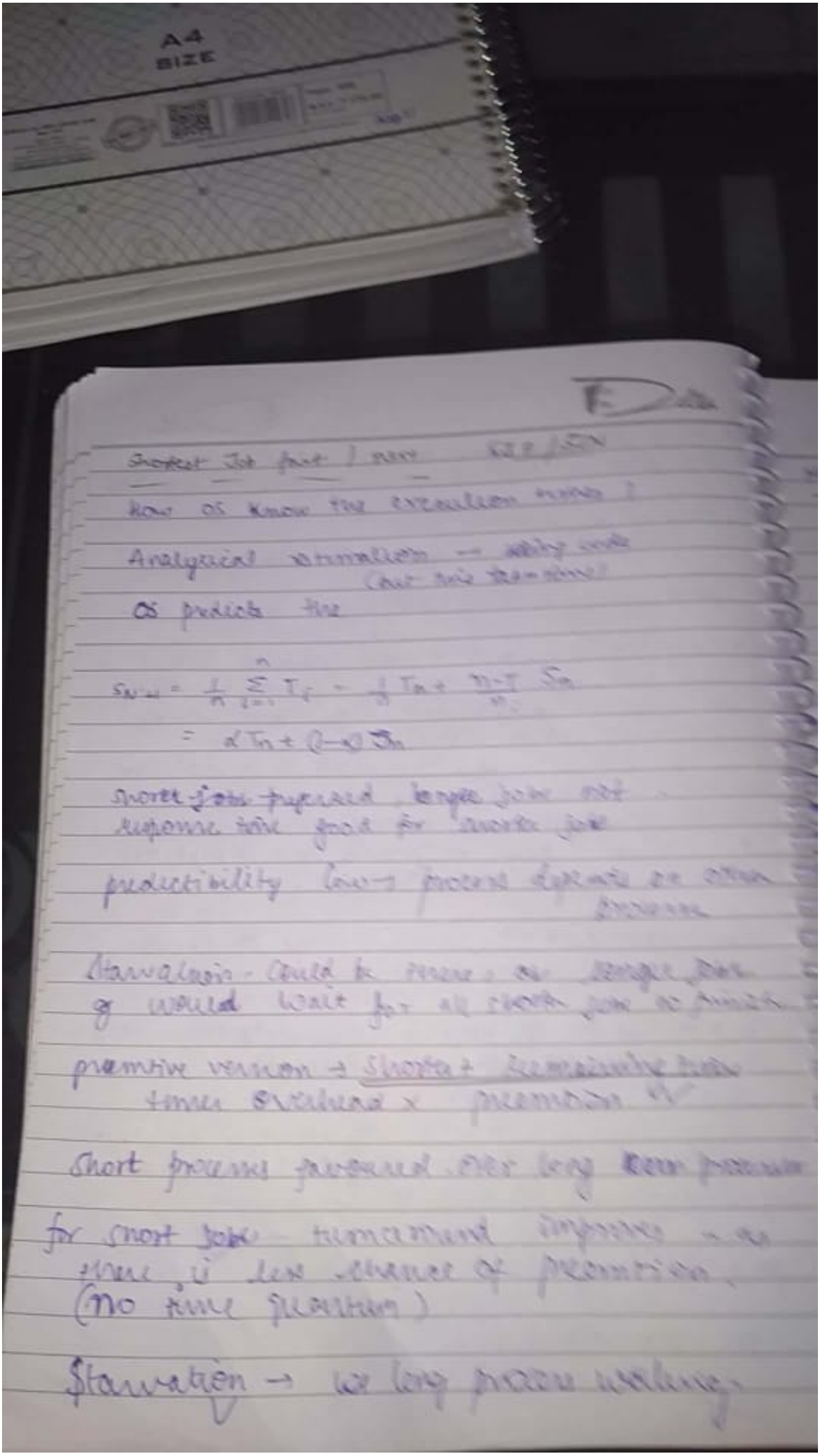
throughput → if time quantum is less than
throughput will suffer (as more context
switch).

not fair treatment to IO / processor bound
if seperate ~~for~~ queue — ✓

~~pred~~ quite predictible

no starvation

Scanned by CamScanner



Scanned by CamScanner

Delta

Multilevel Feedback Queue

First put in ready Q (highest priority)

Once completed time quantum / or preempted put in 1st feedback Q (lower priority) and so on.

turn around time → high.

Should be starvation for longer process

→ give high priority time quantum to lower priority Queues

→ or increase the priority of process waiting for much time.

Mutual Exclusion

if two processes give command to printer → there could be mix.
→ similar for IO ports, buffers.
If one is writing in buffer → other should not.
There must be exclusive access of resource for some process.
(process synchronization).
OS arbitrates manages it.

Scanned by CamScanner

Delta

Some problems of Mutex

① Deadlock (occurs in shared memory communication)

P_1 using R_1, P_2 using R_2
P_1 wants R_2, P_2 wants R_1
there would be deadlock.

Any process that is not mutually exclusive can be preempted.

② Starvation

One process may be waiting for resource and not getting it.
There is no progress of one process.

- Part of code that manages resources is called Critical Section (CS)

P1

```
void P1
{ while (true)
    { /* precode */
      enter CS.
```

Scanned by CamScanner

/CS/
exit CS
/* remaining code */
}
}

Delta
} small but access some resource

Similar for process P2.

One IO port / buffer common b/w P_1 & P_2

b/w enter CS & exit CS → IO port is used

↗ must be controlled.

↗ short (not very long)

Only one process can enter CS, others are waiting but cannot enter.

Simplest method → disable interrupts

↗ Short term scheduler stops + OS stops.

↑ by timer, SJF (preamtive when shortest job come).

↑ could be dangerous.

may not work for multi processors as interrupts could be disabled for only one processor not another.

↗ (that could be shared buffer)

{emergency could come}.

Scanned by CamScanner

OS can also come into picture.

T: Delta

Special hardware instruction →
instruction that can not be preempted. So when it is running nothing could interrupt it.

Test and Set Instruction (TSL) {for swap & exchange}

```
boolean testSet (int i)          ← per function
{
    if (i==0)
    {
        i=1
        return true;
    }
    else
    return false
}
```

Definition of instruction.

← System can not be interrupted during this instruction

Value of i cannot be changed in between.

{Critical section is preemptive}

```
Const int n;    // n number of processes
int bolt;

Void P(int i)    → process
{
    while (true)
        while (!testSet(bolt))
            /* do nothing - wait */
    else /* enter critical section */
        bolt = 0;
    /* remainig code */
}
```

Scanned by CamScanner

```
void main ()
{
    bolt = 0;
    parbegin (P1, P2 - - - - - - - );
    // begin all processes parallely.
}
```

all processes in ready queue

One process → execute ⇒ bolt = 0
also make bolt = 1 ← while condition true CS will execute.

next process will come and as bolt is 1 so function will return false → will not do nothing. (busy-wait).
to CS of this process will execute only when bolt = 0 → will become 0 when CS of first ~~one~~ process completes.
Serialisation is happening
(mutex is spin lock).

Starvation can happen as there is no selection process at entry to CS, it is arbitrary.

NO deadlock → as serialization is happening
deadlock can happen when priority is there.

Scanned by CamScanner

A4 SIZE

Neha

P1 (low priority) → enters CS → time Quantum finish
P2 (high priority) → can not enter CS as bolt is not Open

P1 will now get no chance to open the bolt and P2 will not get chance to enter CS
⇒ Deadlock occurs.

Higher level (Software) level solution needed.

Semaphores → P (wait) } indivisible instructions
→ V (signal) } no interrupt can come
enabled by hardware but not software solutions

two primitive

counting (count: of resources) CS are to one
binary → only one can do P or V

```
Struct Csemaphore {          // counting semaphore
    int count;
    queue type queue;
};
```

process that want resource execute here

```
void Cswait(Csemaphore S)
{
    S.count--;
    if (S.count < 0)
    {
    → place process in S.queue
    → block the process
    }
}
```

(waiting on semaphore)

Semaphore is data structure with some operations like a class but not class. It is an abstract datatype.

Semaphore is synchronisation primitive.
(In last example test-set instruction was synchronisation primitive)
processes in parallel use CS → they need to be serialize.

Scanned by CamScanner

Delta

queuetype → no' can be any discipline
if FIFO ⇒ it is called strong semaphore
and process are totally serialised.
weak semaphor → any process next.
queue → (mechanism) all will get chance.
→ no starvation.

```
void csemSignal ( csemaphore s).
{
    s.count++;
    if (s.count <= 0).
    {
        remove a process P
            from s.queue
        and place P on the
            ready queue
    }
}
```

process who want to return resource will execute it.

at a time, only one signal must execute them.
So they also needs to be in critical section.
so test and set op. is needed here.

s ← count
  ← queue
  ← flag (for operating system)

accessible to all. bolt of test & set.

Scanned by CamScanner

```
semwait (S)
{
  while ( ! testset (S.flag)
      /* do nothing */

  S.count--;
  if (S.count < 0)
    {
      place in S.queue;
      block the process
      set S.flag = 0;
    }
}

semsignal (S)
{
  while ( ! testset (S.flag)
      /* do nothing */

  S.count++;
  if (S.count <= 0)
    {
      remove process from the queue.
      place p on Ready Queue.
      set S.flag = 0.
    }
}
```

Scanned by CamScanner

used for big data structure
↓ where a part can be utilized
small

Delta

Binary semaphores → (Lock / unlock mechanism)
. It can be 0 or 1 only.

```
struct bsemaphore {                    1 → unlock
                                       0 → lock
    int value;                  // value can be 0 or 1
    queue type queue;
}

Void bsemwait (bsemaphore s)
{
    if (s.value == 1)          // resource available
        s.value = 0;           → block for next
    else
    {
        place the process s.queue
        block the process;
    }
}

void bsemsignal ( bsemaphore s)
{
    if ( s.queue is empty())
        s.value = 1;
    else
    {   s.value = 1;
        remove a process p from s.queue
        place p on ready queue;
    }
}
```

Scanned by CamScanner

Delta

Mutual exclusion using semaphores:

```
Const int n;          // n processes

semaphore s;          // binary software
{s.value = 1;}
{s.queue = null}
      ↓↓
Semaphore s = 1;

Void P1 (int i)
{
    /* pre code */

while (true)          // till not successfull try.
{

sem wait(s);          // next process block
                      only one process
                      can access resource
}
   /* execute the
        critical section */

   sem signal(S);
   break;
}
   /* remaining code */
}
```

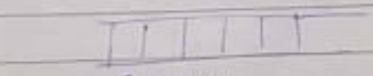
after getting loaded → ag and again coming at in ready Queue → it will again execute the instruction it was blocked.

Scanned by CamScanner

```
void P(int i) ──→ process will keep running.
{
  while(true) ──→ will run till external signal kill it.
  {
    /* precode */
    SemWait(S)
      /* CS */
    SemSignal(S)
      /* remaining code */
  }
}
```

Say an interrupt I0 comes → so P will run again & if another I0 comes it will go to precode → as P is running infinitely.

Producer Consumer Problem



Producer → put data
Consumer → take it

When producer uses it → consumer should not access it, when consumer uses it, producer should not access it.
Consumer should not access empty Queue & producer should not access an full Queue.
for infinite buffer → can not be full but can be empty.

Scanned by CamScanner

```
void producer()
{
  while(true)
  {
    /* produce an item v */
    b[in] = v;          // buffer b
    in++;               // in is pointing to empty slot
  }
}
void consumer()
{
  while(true)
  {
    while (in <= out)
      /*do nothing */

    w = b[out];
    out++;              // first in first out.
    /* consume w */
  }
}
```

This code will not work as both are accessing buffer and may access it at same time.
So we need mutex.

Test & Set X: (not directly used)
Semaphore ✓

int n=0 // ~~no of processes (procs) in buffer~~ number of items in buffer

bsem S = 1 (resource is available)
bsem a = 0; (buffer is empty)

When buffer is in used none can access it.

S → Semaphore for resource whether buffer is available to use or not

Consumer can not consume till bsem a = 1

a → whether buffer is empty or not
If empty → 0 (can not be used)

```
Void producer()
{
    while(true)
    {
        produce()

        bSemwait(S);
        append;
        n++;
        if(n==1)
            bSemsignal(a);
        bSemSignal(S);

    }
}
```

Critical section

Scanned by CamScanner

```
void consumer() { bsemwait(a); // to check for first time.
  while(true) {
    int m;
    bSemWait(S);
    take();
    m--;
    m=n;
    bSemSignal(S);
    consume();
    if(m==0)
      bsemwait(a);
  }}
```

critical section

Bounded buffer Producer/consumer.

```
Const int n=1000;
bSemaphore a=1                    // Exclusive acc
bSemaphore b=0;                   // full
CSemaphore e = size of buffer     // empty.

void producer()
{
  while(true)
  { produce();
    semwait(e);    // check whether full or not
    semwait(a);    // exclusive acc
    append()
    semSignal(a);
    semSignal(b);
  }
}
```

CS → append()

Scanned by CamScanner

```
void consumer()
{
    while(true)
    {
        semwait(b);
        semwait(a);
CS  →   take();
        semsignal(a);
        semsignal(e);
        consume();
    }
}.
```

Semaphores for two purpose → Mutex & process synchronization

main will spawn two process → producer & consumer

semaphores are still low level and prone to mistakes → so people came with the idea of Monitor. monitor is like a class having its own data and procedures.

data → private, procedures → public

at a time one object/entity can enter monitor i.e call the procedures (like CS) one process come and take monitor → monitor blocks (can not be used again). when process has to wait for something, it gets blocked so before getting blocked it should leave monitor and again resume from there only.

Scanned by CamScanner

Delta

Condition variables are special variable used inside monitor and can only be accessed by its procedures.

two procedures → cwait(c) ← process will suspend execution in condition
csignal(c) ← one of the process will resume
(not semaphore wait/signal)

Usually has mutex only also →
cwait(c, m), csignal(c, m)

many processes can wait on condition.

Producer-Consumer Problem using monitor

exclusive access is guaranteed as no other process can enter monitor when one is already there
We need to declare condition variables for full and empty conditions.

monitor bounded buffer;
int nextin, nextout, // need not necessary to be in monitor as one is accessed by producer and one by consumer but if there are multiple consumers/producers it would need exclusive addition.
char buffer[N];
int count; (no of sources).

Cond notfull, notempty; → condition variables
(each condition variable has its queue).

Scanned by CamScanner

Delta

```
void append (char x)
{
    if (count == N)
        cwait(notfull);     // wait will suspend the process, let blocked & save the monitor, next element in queue will come

    buffer[nextin] == x;

    nextin = [nextin+1] % N
                // Circular buffer

    count++;

    csignal (notempty);     // the process which was waiting at notempty Queue will resume otherwise nothing will happen. unlike semaphores → where some computation was taking place.
}

void take (char &x)
{
    if(count == 0)
        cwait(notempty);

    x = buffer[nextout]

    nextOut = [nextout + 1] % N

    count--;
    csignal(notfull);
}
```

Scanned by CamScanner

Scheduler → generally → round robin with multi level queue (improve response time)

# initialization
{
nextin = 0; nextout = 0; count = 0;
}

message passing → don't need exclusive access
overhead ↓
synchronisation primitives ×

within a computer may be faster

but in distributed environment → slower
as prepare message pkt
header / trailer
& wait for acknowledgement

For process of synchronisation → (Inter process communication)
1) Shared Memory (can also be on different
2) message passing computers i.e.
shared a resource
is shared b/w 2 or
more computers rather
than just one)
(distributed shared
memory)

→ no need
of semaphore,
monitor
send & receive
paradigm

OS → has to prevent deadlocks / or
remove it if occurs.

avoid deadlock → serialise process ✗✗. not good

Scanned by CamScanner

special purpose computer → control program
general purpose computer → OS.

Delta

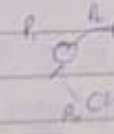
balance performance with unwanted things.
⇈
Challange for OS / developers.

Prevent deadlock → no parallel processing
but it is not efficient.

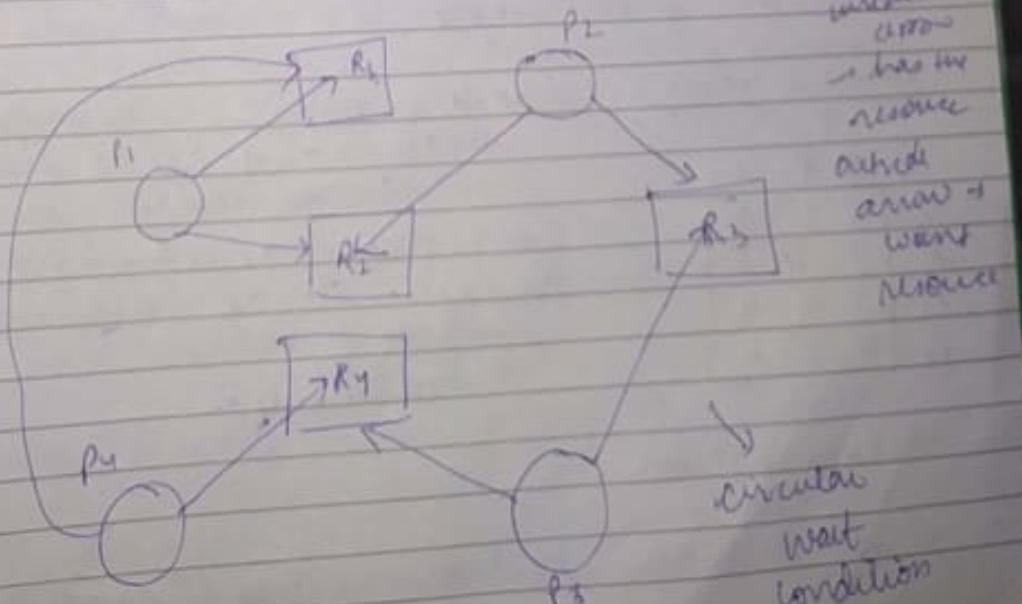
Deadlock Management

Deadlock can happen due to mutual exclusion, hold and wait (process has a resource & needs another resource, it will keep the previous resource with it [nested (S)], no preemption (process will be prempted only on its time out) (cpu are synchronous).

Deadlock will happen in the case of circular wait



P_1 dependent on P_2, P_2 on P_3 & so on.



Scanned by CamScanner

Delta

handle deadlock → forcefully preemption by OS (kills one) & take the resource from one process (not very efficient).

Linear ordering of resource type.

$R_{T_1} < R_{T_2}$ — If one process is given of R_{T_1} then next resource would be of only of higher priority.

$j > i$

P_1 can go from i to j but P_2 can not go from j to i so deadlock is prevented.

AVOID DEADLOCK

predict beforehand that deadlock will happen & give & give resource accordingly so that deadlock do not happen.

Claim C Matrix (should be known before which process needs what how much resource (not always practical because no of resource needed is not known beforehand

	R_1	R_2	R_3
P_1	3	2	2
P_2	6	1	3
P_3	3	4	4
P_4	4	2	2

Allocation Matrix

OS knows this (it knows how many are allocated)

Scanned by CamScanner

Delta

	R_1	R_2	R_3
P_1	1	0	0
P_2	6	1	2
P_3	2	1	1
P_4	0	0	2

(C − A) required

	R_1	R_2	R_3
P_1	2	2	2
P_2	0	0	1
P_3	1	0	3
P_4	4	2	0

R_1	R_2	R_3
9	3	6

} resource vector (R-vector) how many of each type of resources.

R_1	R_2	R_3
0	1	1

} available vector (how many are available (Total − used allocation))

$$R_j = V_j + \sum_{i=1}^{n} A_{ij} \quad \forall j$$

(n → no of processes)

$\{A_{ij} \le C_{ij} \le R_j\}$ for all i & j

- Don't start a process which will lead to deadlock
- Don't grant an incremental resource request if it will leads to deadlock.

approach 1 → $R_j \ge (n+1, j) + \sum_{i=1}^{n} C_{ij}$ { available resources (if it are more than the needed resources)

not effective → not necessary to give all the resources at the beginning

and also some process may release a resource → so it must be utilized.

(it means all the process will get all the resources in beginning)

Scanned by CamScanner

Operating System

Delta

. (Bankers algorithm)

OS will see that if any process will can get all its requirements and after getting → it will return all its resources vector ↑

	R_1	R_2	R_3	
P_1	2	2	2	✗
P_2	0	0	1	✓ → now return resources.
P_3	1	0	3	✗
P_4	4	2	0	✗

→ allocation of every data → 0.

✓ R_1 R_2 P_3

[6 2 3]

now pupil P_1 & return resources & so on.

OS decided safe it is a safe state

P_2, P_1, P_3, P_4

Correct Algorithm deadlock will not happen

Eg 2

Claim same R → same

allocation

	R_1	R_2	P_3
P_1	2	0	1
P_2	5	1	1
P_3	2	1	1
P_4	0	0	2

C − A

	R_1	P_2	P_3
P_1	1	2	1
P_2	1	0	2
P_3	3	0	3
P_4	4	2	0

✓ R_1 P_2 P_3

[0 . [ ]]

Scanned by CamScanner

Delta

P_1 needs incremental resource $R_1 \to 1, R_3 \to 1$
* OS will not give it
as a situation will occur that all process needs R_3 but availability → 0 So deadlock will happen → so will not grant P_1 the resources.

Its a overhead as :
1) no of resources not known before one (needed)
2) lots of computation (find safe state)

So OS generally don't use bankers algorithm
instead just preempt to break deadlock.

Prevention

4 reasons

1) Mutex: some situations critical section not required like if processes only reading from disk
Distributed → consistent cache
* compromise b/w consistency & performance
deadlock → all 4 necessary
1, 2, 3 → not sufficient

2) Hold & Wait: starting want all resources (not true also) → like in multithreaded.

3) No preemption: OS will be required to give different priorities to all. when preemp → resources lose.
Starvation & save context of the process

Scanned by CamScanner

Delta

(4) Circular wait → linear ordering

Advantage avoidance → less restrictive than prevention
↳ efficiency more
disadvantage of avoidance
- know beforehand resources
- processes must be independent of each other (if all have CS)
- continuously keep track

Or prevent / avoid, just detect.
when process requests resources → check deadlock at that time.

Detect deadlock

V matrix W=V (availability matrix).
A matrix (resources allocated to the process).
Qij (Request matrix) [claim].

mark all the process that are not deadlocked.

Initially all are unmarked.

mark all the process which has all zeros (Zero resources allocated → no deadlock)

find an index i such that Q<W.
ie $Q_{ik} \leq W_k \ \forall k$

If there is any such row mark that process.

$W_k = W_k + A_{ik} \ \forall k$ (return resources)

repeatedly do this process.

Scanned by CamScanner

If no such process could be found, all unmarked processes are deadlocked

[After one process is finished another process could be granted resources:] → again check

Can not gurrantee that deadlock will not occur

No sequence is checked → all process are checked in bulk

Whether or not deadlock will occur or not depends upon order of processes. → here no order is checked

(B matrix (request matrix) at any given point of time)

	R_1	R_2	R_3	R_4	R_5
P_1	0	1	0	0	1
P_2	0	0	1	0	1
P_3	0	0	0	0	1
P_4	1	0	1	0	1

(A matrix) Allocation matrix

	R_1	R_2	R_3	R_4	R_5
P_1	1	0	1	1	0
P_2	1	1	0	0	0
P_3	0	0	0	0	0
P_4	0	0	0	0	0

Totals

$R = [2\ 1\ 1\ 2\ 1]$ $\quad W = Y$ $\quad$ available

$[0\ 0\ 0\ 0\ 1]$

mark P_4 → Zero resources allocated [allocation matrix]

Scanned by CamScanner

mark P_3 → $Q_3 <= W$

W [00011]

B : P3, P4 → removed

now request of P_1 and P_2 can not be fulfilled as $Q_1, Q_2 \nleq W$

They are unmarked ⇒ deadlock occurs.

OS aborts P_1 and P_2] Simple way to get out of deadlock.

↳ or can successively abort process and check for deadlock till deadlock doesn't occur.

↳ in which order ? (many ways)

→ abort the process 'first' who has taken less processor time

→ OR process that has given least output.

→ OR abort the process whose maximum processor time is remaining * completion required

→ least priority process

→ Abort the process who has taken least resources

General purpose Compiler → generally (detection occurs (not prevention/avoidance).

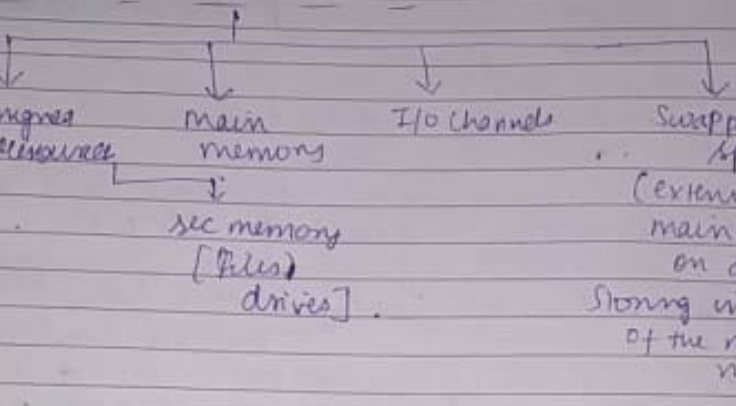
Integrated Methods

Many Prevention/avoidance/detection can be mixed.

Scanned by CamScanner

Delta

Group resources in classes



OS can decide strategies for all the classes.

We could use prevention method to prevent deadlocks among groups. → linear ordering could be given for ~~resource assignment~~ easily

But there could be deadlocks within a class

main memory → "preemption" → because memory could be given easily for some time

assigned resources → avoidance → can not be preempted but printer → can not be stopped in between

In prevention → current state needed to be saved. [registers, file pointers, IPC, process communication, multiple threads].

There are checkpoints → states are saved at checkpoints when it restarts → rolls back to previous checkpoint

Scanned by CamScanner

5 philosophers dining problem

Delta

Lot of transactions -

after 1000 transactions → checkpoint (save state)

If premption occurs at 4900th transaction

till 4000th will be saved

IO-channels → have priority → so linear ordering could be used.

5 philosophers dining problem

5 philosophers
5 forks.

mutual exclusion → one fork can be given to only one philosopher

hold and wait → take one fork and keep it till do not get 2nd fork

No premption → not stopping hungry philosopher to eat

Circular wait → all are hungry.

⇒ hence deadlock will occur.

Scanned by CamScanner

semaphore → not int
but Binary also.

Solution

1) allow only 4 to pick forks.
Atleast one will get 2 forks.

2) let them eat with only one fork.
→ using less resources

3) get extra forks. → atleast one more
→ minimum number of sources required to get out of deadlock.

Program 1 → Deadlock can occur

Semaphore fork[5]=1. // all forks are shared resources → so 5 semaphores

```
Void philosopher(int i)
{
    while(true)
    {
        think();
        wait(fork[i]);
        wait(fork[(i+1)%5]);
        eat();
        signal(fork[i]);
        signal(fork[(i+1)%5]);
    }
}
```

} order does not matter here

} order do not matter here but in some case can be.

Scanned by CamScanner

```
main ()
{
    parbegin ( philosopher(0), philosopher(1)
                                    philosopher(4));
}
```

deadlock does not occur

Program 2

counting semaphore

```
Semaphore room[5] = 4;        // 4 can enter room
Semaphore fork[5] = 1;

Void philosopher (int i)
{
    while (true)
    {
        think ();
        wait (room);
        wait (fork[i]);
        wait (fork[(i+1)%5]);
        eat ();
        signal (fork[i]);
        signal (fork[(i+1)%5]);
        signal (room);
    }
}
```

This order is important as philosopher first need to take room then forks & leave forks first then room.

Scanned by CamScanner

prevention → acquire the forks till he get 2nd one,
⇓
both two forks simultaneously
↳ no hold & wait

Delta

Program 3 → <u>using monitors</u>

monitor

```
cond      ForkReady[5];
boolean   fork[5] = true;        // for status of
                                    fork.

void getfork( int pid)
{
    if ( ! fork[pid])
        cwait (forkReady[pid]);       ] hold & wait → deadlock will occur.
    fork[pid] = false;
    if ( ! fork[ (pid+1)%5])
        cwait ( forkReady[(pid+1)%5])
    fork[pid (pid+1)%5] = false;
}
void releaseFork( int pid)
{
    fork[pid] = true;
if (Q is not empty) → csignal ( forkReady[pid]);      // signal only when there is some one in Queue.
    fork[(pid+1)%5] = true;
if (Q is not empty) → csignal ( forkReady [(pid+1)%5] );
```

holding on left fork till he not get right back

```
philosopher(int i)
{
    think();
    getFork(i);
    eat();
    releaseFork(i);
}

main()
{
    parbegin(philosopher(0) - - - philosopher(4));
}

#

Void releaseFork(int pid)
{
    fork[pid] = true;
    if (forkReady[pid] {a} is not empty)
        Csignal(forkReady[pid]);
    forkReady fork[(pid+1)%5] = true;
    if [forkReady[(pid+1)%5] {a}
        is not empty)

        Csignal(forkReady[(pid+1)%5]);
}
```

Delta

Scanned by CamScanner

CEC 14 : Operating System

Challenges of process management by OS:

(I)
1) allocate memory → address space (data and instruction)
2) whenever there is a context switch → it has to store all the registers, flags needed to in the memory
3) File pointer is needed to be stored : File pointer is the address of directory entry (where all the details of file are stored → in directory etc)
4) How store how many CPU cycles taken, I/P, O/P time

OS has to allocate memory and store in Process Control Block (PCB) and keep updating it.

If there is multiprocessing →

(II) Scheduling needs to be done
- → to Optimise the performance parameters (throughput, turnaround time etc)
- → Power consumption
- → Predictability

(III) Concurrency / parallel processing:

i) shared resources.
(first shared resource is main memory).

Scanned by CamScanner

Mutual Exclusion is required to manage shared resource

ii) Inter-process communication (IPC) [Producer/consumer]
If proces are cooperating ~~resource~~ with each other than memory etc are shared otherwise devices are shared always IPC requires mutual exclusion and other things also like buffer full, empty etc.

iii) Deadlock → avoidance, ~~dest~~ detectance, prevention [objective regarding which strategy is to be followed]

IV Multithreading → OS deals with it or User deals with it (User libraries)
It is part of process management (thread management).

V A process can spawn other processes.
fork/join → spawn new process from existing process
any user process is forked by OS.
Zombie State → a process which is orphaned (no return from main process)
So OS needs to take care of these processes also.
These are some of the functions of OS.

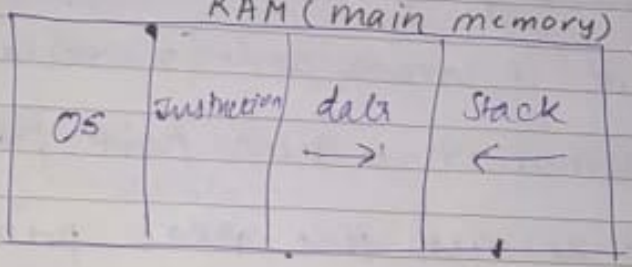
VI If there is only one process - no problem
but many processes → OS has to do work.

Scanned by CamScanner

Memory management

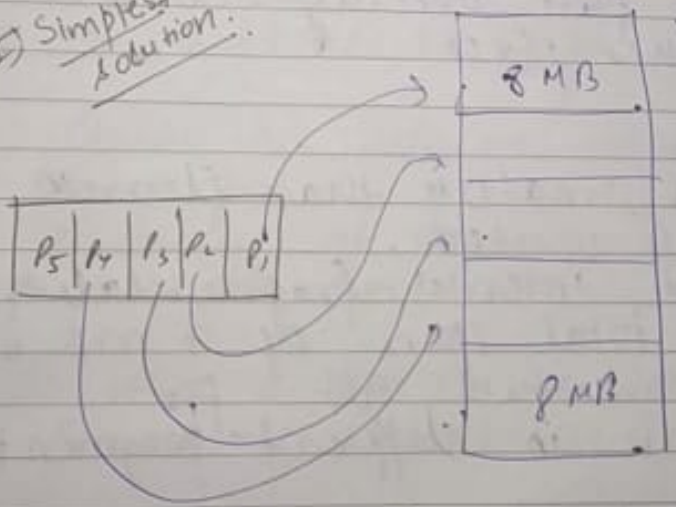
Switch on → PC → 0000 or PC → FFFF
Should be some instruction at 00 or FF
which jumps to sub routine where OS is loaded.
→ Some instruction is already in the main memory.

Space for data, instruction, stack.



- multiprocessing

1) Simplest solution:



Fixed sized partition & Fixed no of partions

memory is divided into chunks of equal size and each prog get one chunk.

allocation: find an empty slot allocate it
If no empty slot → swapping scheduling.

Static partitions → allocation is simple

Disadvantages

- So if there is a program whose size is greater than the size of chunk
- responsibility of user to determine the overlays
  - ↳ divide program into mutually exclusive parts.

If 16 MB ~~proben~~ program

- → Divide it into 2 parts (2 overlays)
- → give command to run first part
- → after that run the next part.

If there is a jump instruction from 2nd overlay to one first overlay → context switching takes place.

- If program is smaller than Chunk size memory is wasted. This is called 'internal - fragmentation'. There may be total space of 7 MB but 7 MB program would not fit as that space is in different fragments.
- Choice of fragment/Chunk size (to reduce the problems).

Scanned by CamScanner

⇒ fixed no of partitions → limiting the degree of multiprogram → fixed no of process in active state (ready / running)

Memory is needed for process in ready or running state.
Not needed in suspended / blocked state

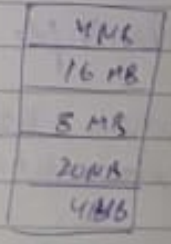
Advantages

Simple and fast.

(II) Second Solution.

Divide into chunks but size should be variable
(Variable size static partioning).

- any process which is less than or equal to 4MB (eg) can be allocated there.



Queue for different sized memory
a queue of 4MB space, another queue for 16 MB space etc.

So when a process is waiting in 4 MB queue → scheduling comes into way.

inefficient → a 4MB process waiting → 16 MB space empty.

Scanned by CamScanner

Different queues → handling internal fragment
↳ reduces performance parameters

Single queue → 4MB space available →
allocate them otherwise any other

* Readers | Writers Problem

File | resource | buffer → to be read by readers, written by writers
Not direct IPC → [IPC with shared resources]

1) Two writers can not write simultaneously

2) when writer is writing, reading should not take place

3) Multiple readers can read it.

4) writer has to wait for all the readers to read
→ starvation for writer.

After writer comes → no more readers should be allowed.

Program:-

Semaphore wsem = 1; (no body can come with writer)
int Rcount; (no of readers)
↑
control variable
need a semaphore to update it (mutex)
Can not be simultaneously updated

Scanned by CamScanner

```
semaphor rCountsem=1;

Void reader()
 {
    while (true)
    {
      semwait (rCountsem);
        rCount++;
      semsignal (rCountsem);

      If (rCount == 1)
         semWait (wSem);

      semSignal (rCountsem);

      Readunit();

      semWait (rCountsem);
        rCount--;

      if (rCount== 0)
        semSignal (wSem);

      semSignal (rCountSem);
    }
 }

Void  write ()
 {
    while (true)
     {
```

Scanned by CamScanner

```
        semWait (wsem);
        writeunit();
        semSignal (wsem);
    }
}
Void main()
{
    Rcount=0;
    parbegin(reader, writer);
}
```

// starvation problem is there in above code.

without starvation:

```
INT Rcount, Wcount, /a word
semaphore wsem=1; RcountSem=1; WcountSem=1;
    rSem=1; manyReadersLock=1;
    wcount                    no of readers allowed
                              to compete with writers
Void reader()
{
    while(true)                    // whether to go in
    {                              only reader or
                                   or normal Q.
        semWait(manyReadersLock);
        semWait(rSem);             // whether all writers
                                   have written

        semWait(rCountSem);
        rCount++;
```

Scanned by CamScanner

```
    if (rcount == 1)
        semWait (wsem);

    semSignal (rcountsem);

    semSignal (rsem);
    semSignal (manyReadersLock);

    ReadUnit();                    -> in CS for only 1st Reader of wsem.

    semWait (rcountSem);
     RCount--;
     if (Rcount == 0)
        semSignal(wsem);
     semSignal(rcount sem);
  }
}

void writer()
  {
     while (true)
     {
        semWait (wcountsem);
          wcount++;

        if (wcount == 1)
          semWait (rsem);

        semSignal (wcountsem);

        semWait (wsem);
```

Scanned by CamScanner

```
WriteUnit();
semSignal(wSem);

semWait(wCountSem);
    wCount--;

if(wCount==0)
    semSignal(rSem);

semSignal(wCountSem);
}
}
```

Memory Management (dynamic partitioning).

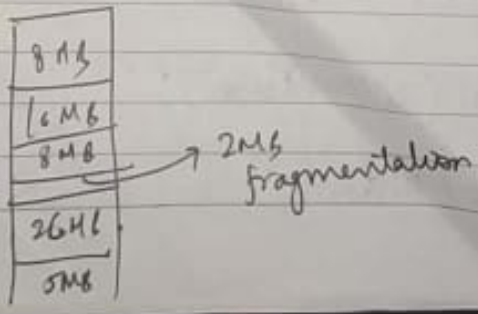
OS should know the maximum size of program running

Size grow → Overlay.

To small to hold any practical program (fragmentation).

a new program of 8 MB come

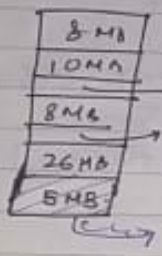
→ 3rd Program will go from blocked to block suspended & 8 MB will be filled then



Scanned by CamScanner

No IO of P_3 is finished so it comes in suspended ready

Now P_2 is removed → (swap to disk) suspended
& P_5 is filled there

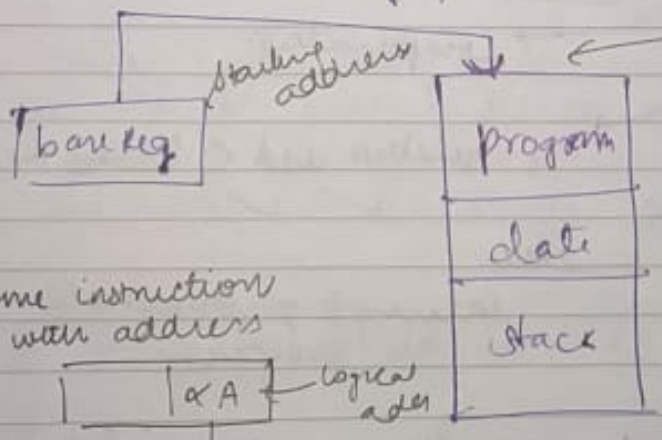


disk has role in earlier OS.

all addresses in executable are logical addr
0 Zero somewhere → start acc to zero
of operand

as program is not in same partition always.

Every program needs a ~~b~~ register to hold
base register → base register.
address of operand = base addr + logical addr



OS will give the starting address
⇒ address of partition in which program is there

relative → also physical address at this point of time

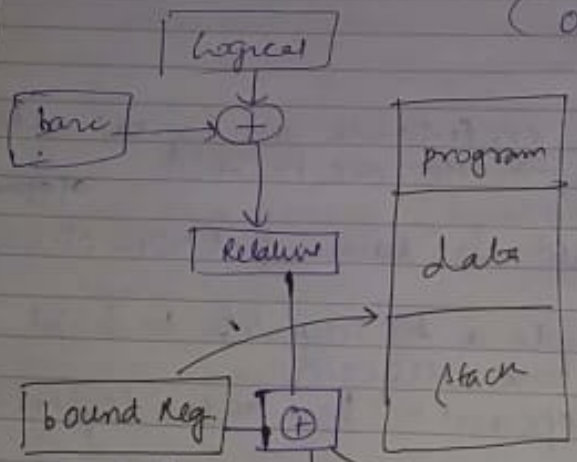
change due to dynamic programming.

Scanned by CamScanner

Program should know the size of programs & under no circumstance user can not exceed the limit

Program in 8MB partition can go only upto 8MB & cannot exceed it.

Not allowed → as it will distroy other programs. (OS is also there)



comparator.

if relative add < bound Reg

else interrupt & terminate the program.

Relocation → ability to give change physical address of it relocation

→ Hardware is required for relocation

Scanned by CamScanner

in static partitioning → (Equal size & variable size)
↳ Relocation is not required.
(as program after coming from blocked state goes to same partition.
in dynamic partitioning → Relocation is required.

disadvantages

- Program is to be relocated every time
- external fragmentation → here some space created b/w partitions.

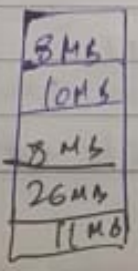
  not internal fragmentation (space within partition)

* Compaction (defragmentation)
  ↳ Move all the programs so that space b/w partitions is not there

for main memory → of program running → OS does compaction (relocation can be done)

time consuming process
OS do it only when a new program enters, space not there but fragments are there → so single partition to create space

for disk compaction → user runs the program to do it ↑ OS do not do it itself.

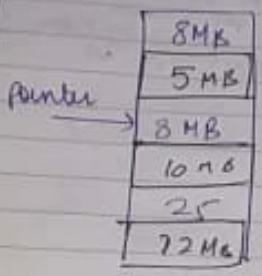


→ same
→ relocated.

Scanned by CamScanner

do something which takes least time.

* where to fit new process?



4MB program

3 algos

→ Best fit
leaves less partitioning
leaves smallest fragment
4MB will go to 5MB
(only 1MB span)

→ first fit
scan from starting &
fit in the first space
5MB here

best fit is worst as it leaves very small fragments which cannot be used
→ so compaction is required

→ next fit
first ~~program~~ fragment after the pointer
→ 10MB here

first fit will leave algorithm fragments on the top while next fit will leave fragments on the bottom.

* There can be worst fit (which leaves largest fragment) ⇒ can be useful as it leaves enough space for next program.

if even after compaction no space?
replace a blocked process
which one? → less time, etc etc.
(algos)

Scanned by CamScanner

new program come → where to place? (Best fit etc algos)

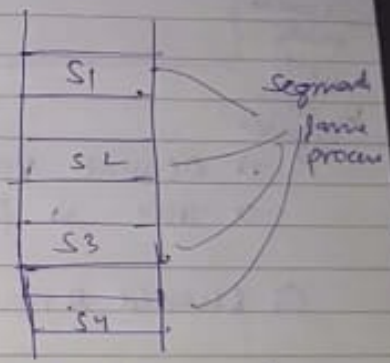
replacement (when a blocked is to be replaced) → which one to replace → (run more complex algos)

* Segmentation
* assembler → segments need to be created for data, program, stack.

segmented memory allocation

segments may be of different sizes

multiple segments in program → diff can be in diff parts of memory



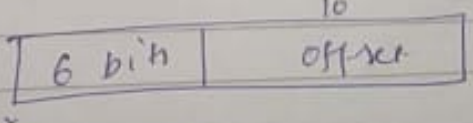
disadvantage → program has to specify size of the segment

need multiple base registers program knows programmer knows size → internal fragmentation ⇒ no

external fragmentation → less but will be there

will need more registers → but only 1, there

Entire table is to be stored → called process segment table



→ for every segment

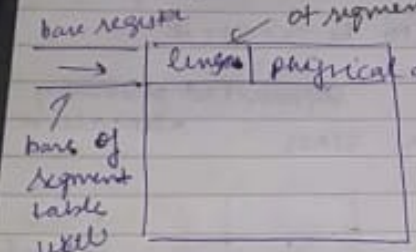
6 → segment (0-63)

Can be in cache or main memory

Scanned by CamScanner

associative cache → best.

- external fragmentation ↓
- modular program help
- dynamic linking



load which segment is required..

no need to load entire program

⇒ OS will vary base register
⇒ This is pure segmentation

OS should know → no of ~~program~~ segments & their sizes

segment → address, comparator, bound, base

it will see the segment which → go to segment table see the physical addr

physical + offset → valid
if > length → interrupt.

Scanned by CamScanner