GATE

Date: 5/8/2020 Page

Operating System

[redacted]

OS → interface between user and hardware
→ Resource allocation
→ managing memory, process files, security
→ user convenience
→ efficiency

→ Batch OS
→ multitasking OS
→ multiprogramming OS
→ multiprocessing OS
→ real time OS

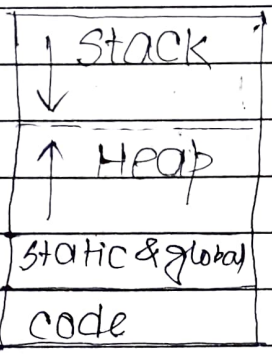
Time

1. CPU time
2. I/O time

Process:

Program in execution

secondary memory
→ primary memory



Attributes of a process:

1. Process id
2. Process counter → next instruction to be executed
3. state - Ready, running
4. Priority
5. General purpose registers
6. List of open files
7. List of open devices
8. Protection

→ Process control block (PCB)
Also known as context

State of a process:

~~1. New - created~~
~~2.~~

1. New - created. Secondary → Primary
2. Ready - " . Primary.
3. Run - one process (1 CPU)
4. Block / wait - IO, main memory
5. Termination / completion - delete PCB
6. Suspend ready - secondary memory
7. Suspend wait / block - " "

multiprogramming : multiple processes "ready"
→ with preemption → without preemption

multitasking / time sharing | Forcefully remove during running.

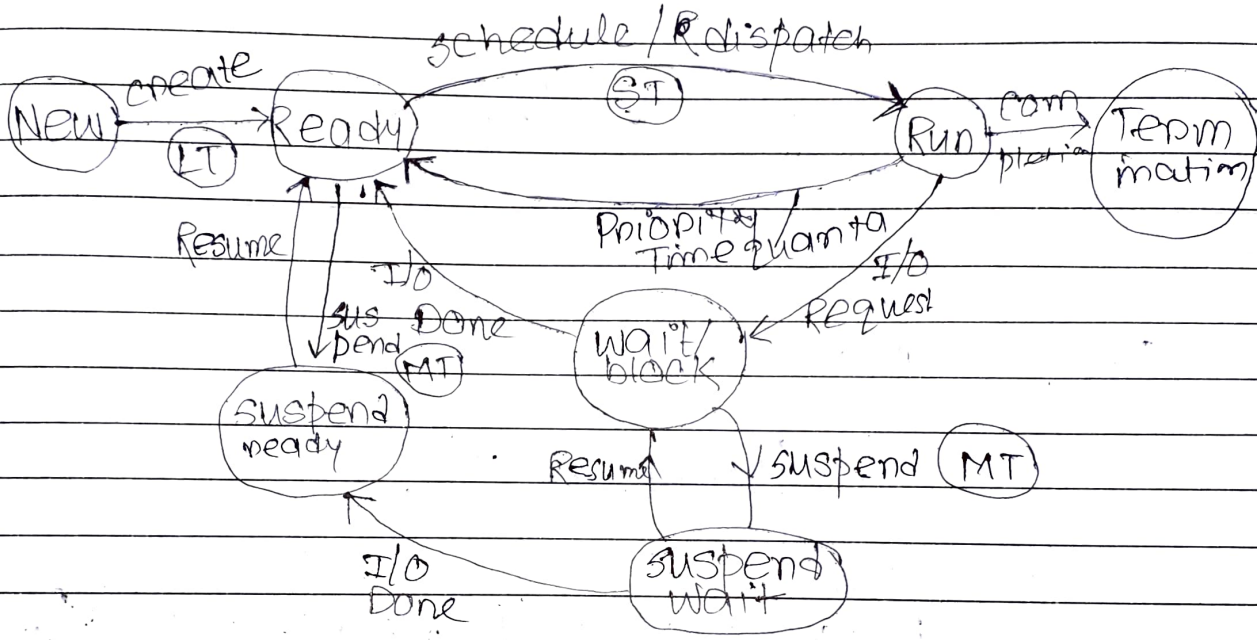
Date: / / Page

Process

(I) creation
(II) scheduling
(III) execution
(IV) Kill / delete

Req. CPU Time > Req. IO Time

(III) CPU bound — IO else bound



many programs can be in Ready state
⇒ multiprogramming

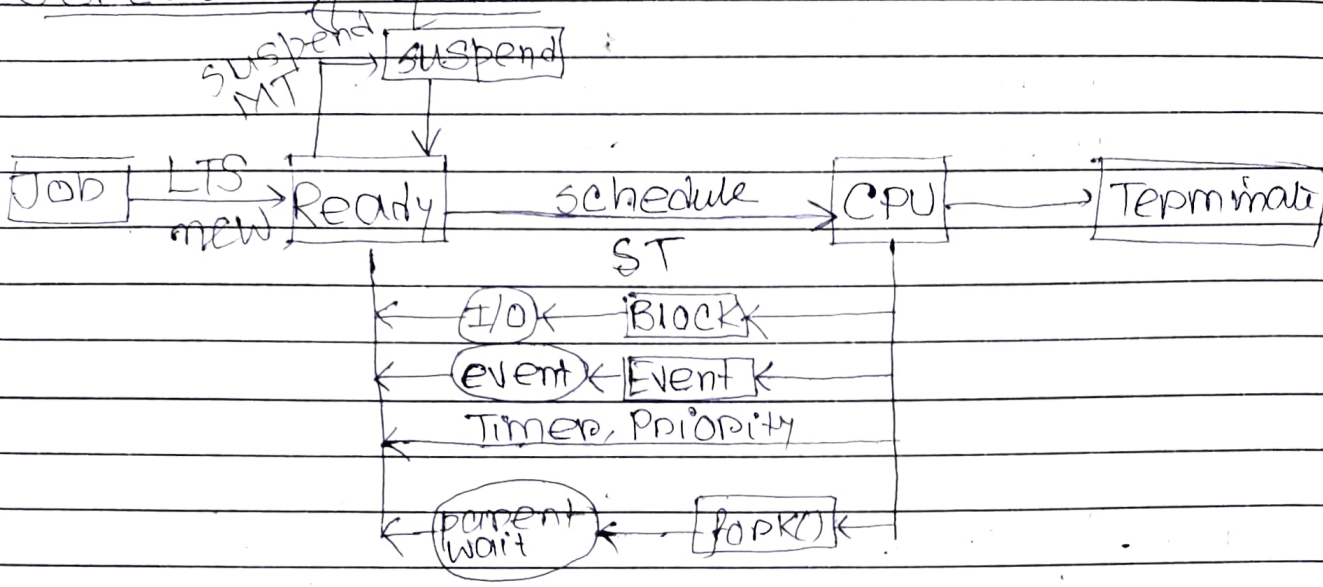
pre emption allowed
⇒ multitasking

degree of multiprogramming = max no of processes allowed to be in Ready state

Scheduler
(1) Long term (LT)
(2) Short term / dispatcher
(3) Medium term (swapping)

Date: / / Page

Scheduling queue:



⇒ queue

Time

ignoring I/O

① Arrival time: The time at which process comes to ready state

② Burst time: CPU time

③ Completion time: the time at which a process finishes

④ Turn around time: (CT − AT)

⑤ Wait time: wait for CPU allocation

⑥ Response time: First time a process get CPU allocated

$$CT - AT = (WT + BT)$$

Operating System

Process scheduling:

CPU scheduling:

Ready → Run

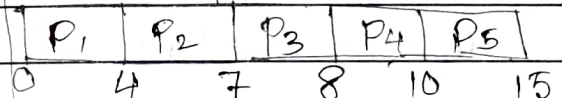
Run → Termination
→ wait
→ Ready
CPU free

Queue of Ready processes
↓
Run

First Come First Serve: (FCFS)

(Assuming no I/O time)
criteria: arrival time
mode : non preemptive

Pno	AT	BT	CT	TAT	WT	RT
1	0	4				
2	1	3				
3	2	1				
4	3	2				
5	4	5				



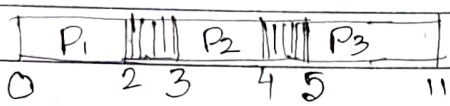
(Grant chart)

TAT ⇒ time spent in the system

Disadv: convoy effect/
starvation : average we wait time becomes high

Date: / / Page

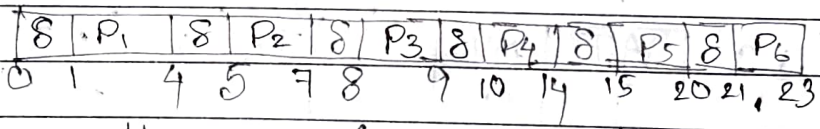
PNO	AT	BT	CT	TAT	WT
1	0	2	2	2	0
2	3	1	4	1	0
3	5	6	11	6	0



Data structure: queue
Time complexity: O(n) → no of processes

PNO	AT	BT	CT
1	0	3	
2	1	2	
3	2	1	
4	3	4	
5	4	5	
6	5	2	

context switching time = 1 unit (δ)



$\left(\frac{6}{23}\right) \times 100\%$ inefficiency (overhead)

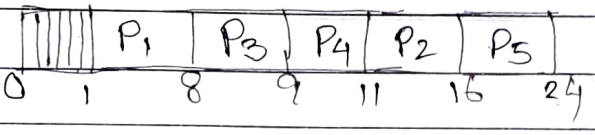
$\eta = \left(1 - \frac{6}{23}\right) \times 100\%$

Date: / / Page

Shortest Job ~~###~~ First (SJF):

Criteria : Burst time
mode : non preemptive

PNO	AT	BT	CT	TAT	WT
1	1	7	8	7	0
2	2	5	16	14	9
3	3	1	9	6	5
4	4	2	11	7	5
5	5	8	24	19	11



Problem : (i) starvation
(ii) Burst time unknown
→ Min Heap (Priority queue)
O(n log n)

Adv : (1) max throughput at any moment during execution ; jobs get done as soon as possible
(2) Min avg WT & TAT

→ ~~we~~ BT needs to be predicted

Date: / / Page

BT prediction

Static

(i) Process size
(ii) " type
- OS
- user
  - interactive
  - for eg foreground / background

$P_{old} \Rightarrow 200\,KB \Rightarrow 20$ unit
$P_{new} \Rightarrow 201\,KB \Rightarrow 20$ unit

Dynamic

(i) simple avg
(ii) Exp avg

Simple avg:

$$T_{n+1} = \frac{1}{n}\sum_{i=1}^{n} t_i$$

$t_i \Rightarrow$ Actual burst time
$T_i \Rightarrow$ Predicted burst time

Exponential Averaging (Aging)

$$T_{n+1} = \alpha t_n + (1-\alpha) T_n$$

α → smoothening factor

$\alpha = 1$ dynamic (t_n)
$\alpha = 0$ static (T_i)

Operating System

Shortest Remaining Time First (SRTF)

criteria : Burst time
mode : preemptive

PNO	AT	BT	CT	TAT	WT	RT
1	0	~~7~~6	19	19	12	0
2	1	~~5~~40	13	12	7	1
3	2	~~3~~20	6	4	1	2
4	3	~~1~~0	4	1	0	3
5	4	~~2~~0	9	5	3	7
6	5	~~1~~0	7	2	1	6

P_1	P_2	P_3	P_4	P_3	P_6	P_5	P_2	P_1
0	1	2	3	4	6	7	9	13	19

→ no starvation → (BT prediction)

Round Robin Algorithm :

criteria : Time Quanta
mode : preemptive

PNO	AT	BT	CT	TAT	WT
1	0	~~4~~20	8	8	4
2	1	~~5~~31	18	17	12
3	2	20	6	4	2
4	3	10	9	6	5
5	4	~~6~~42	21	17	11
6	6	~~3~~1	19	13	10

TQ=2

P_1	P_2	P_3	P_1	P_4	P_5	P_2	P_6	P_5	P_2	P_6	P_5
0	2	4	6	8	9	11	13	15	17	18	19	21

Queue ~~P_1 P_2 P_3 P_1 P_4 P_5 P_2 P_6 P_5 P_2 P_6~~ P_5

TQ ↓ context switching ↑

TQ ↑ → starvation
TQ ↓ → High context switching

TQ = 2

PNO	AT	BT	CT	TAT	WT	RT
1	5	~~5~~ 2	32	27	22	(5-5)
2	4	~~8~~ 8	27	23	17	(9-4)
3	3	~~7~~ 10	36	33	26	(6-3)
4	1	~~16~~ 8	30	29	20	(1-1)
5	2	20	6	4	2	(4-2)
6	6	30	21	15	12	(18-6)

Queue: ~~P4 P5 P3 P6 P4 P1 P6 P3 P2 P4 P1 P3~~

Gnant :

P4	P5	P3	P6	P4	P1	P6	P3	P2	P4	P1	P3
0 1	4	6	9	12	15	18	21	24	27	30	32 36

n processes
s context switching time
q time quantum

$P_1 P_2 \ldots P_n P_1$ ≤ t

$ns + (n-1)q \leq t$

$q \leq \frac{t - ns}{n-1}$

Date: 5/8/2020 Page

Longest Remaining Time First :

PNo	AT	BT	CT	TAT
1	0	2 1	12	12
2	0	4 3 2 1	13	13
3	0	8 4 3 2 1	14	14

P3	P2	P3	P2	P3	P1	P2	P3	P1	P2	P3
0	4	5	6	7	8	9	10	11	12	13 14

Highest Response Ratio Next (HRRN)

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

W → wait time till now

S → burst time

Priority { ① short job
② long wait time

mode: non preemptive

PNO	AT	BT	CT	TAT	WT
0	0	3	3	3	0
1	2	6	9	7	1
2	4	4	13	9	5
3	6	5	15	9	4
4	8	2	20	12	10

P0	P1	P2	P4	P3
0	3	9	13	15 20

$RR_2 = 5/4$

$RR_3 = 3/5$ $\quad RR_3 = 7/5 = 1.4$

$RR_4 = 1/2$ $\quad RR_4 = 5/2 = 2.5$

Date: 5/8/2020 Page

Priority Scheduling:

→ Static : Fixed priority
→ Dynamic : Priority changes with tim

→ Pre emptive
→ Non preemptive

PNO	Prio	AT	BT	CT	TAT
1	(L) 2	0	~~4~~ 3 0	25	25
2	4	1	~~2~~ 1 0	22	21
3	6	2	~~3~~ 2 0	21	19
4	10	3	~~5~~ 3 0	12	9
5	8	4	1 0	19	15
6	(H) 12	5	~~4~~ 0	9	4
7	9	6	~~6~~ 0	18	12

P1	P2	P3	P4	P6	P4	P7	P5	P3	P2	P1
0	1	2	3	5	9	12	18	19	21	22	25

(CPU + IO) time:

SRTF

PNO	AT	BT	IO	BT	CT
1	0	~~3~~ 1	2	2	5	15
2	0	~~2~~ 0	4	1	~~3~~ 1 0	7
3	2	1 0	3	2	~~3~~ 2 0	9
4	5	~~2~~ 1	2	1	~~3~~ 2	13

P2	P3	P1	P4	P2	P3	P4	P1		P4	P1
0	2	3	5	6	7	8	10	11	12	13	15

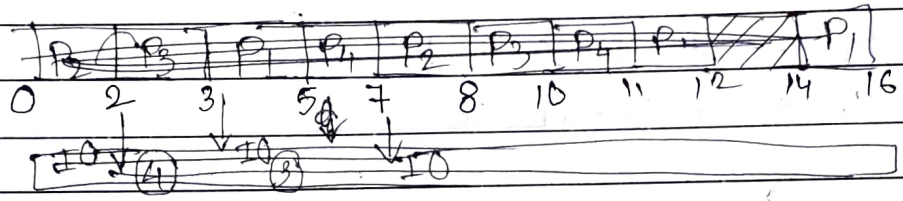
P2 → IO (4)
P3 → IO (3)
IO (2)
IO (2)

Operating System:

CPU + IO time

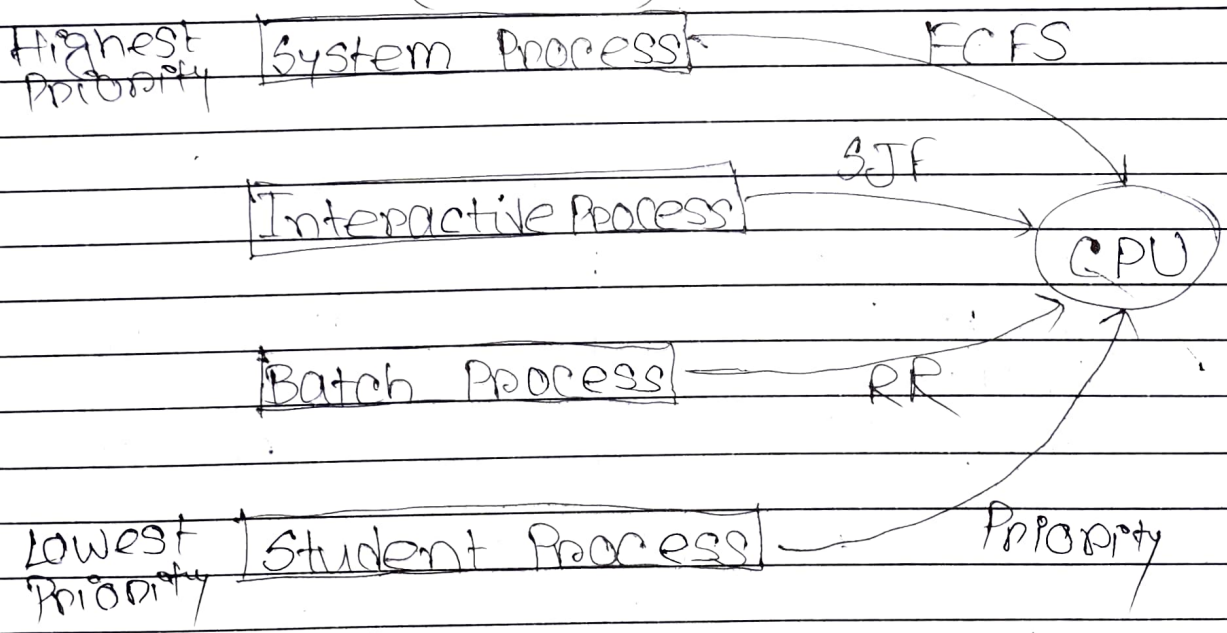
SRTF

PNO	AT	(BT	IO	BT)	
1	0	1	2	2	3
2	0	0	4	0	1
3	2	0	3	2	2
4	5	0	2	1	1



Multi Level Queue scheduling:

(Feedback)



After a wait time, promote a process to higher level of queue.

(Feedback)

Process Synchronization:

Interprocess communication:

Shared memory	↳ critical section (that access shared memory)

need:
preemptive scheduling + shared memory

Graham's Law: If something could go wrong, it would go wrong.

```
P1                          P2
{                           {
  count++                     count--
}                           }
```

count [4]

① mov R0, count	① mov R1, count
② Inc R0	② Dec R1
③ mov count, R0	③ mov count, R1

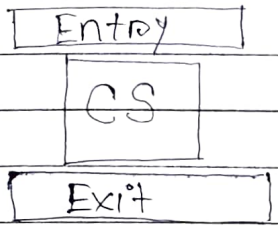
~~1~~ ~~2~~ P1: 1, 2 P2: 1 2 3 P1: 3

count [5]

Race condition: output depends on the order of execution

Date: 6/8/20 Page

solution: only one process allowed to access the critical section



Synchronization mechanism:

Requirement:

Primary { ① Mutual Exclusion — 2 entry not allowed
② Progress — 1 does not want. This must not stop 2

Secondary { ③ Bounded waiting — avoid starvation
④ Portability — architecture neutrality

(Requirement)

→ with busy waiting
→ without busy waiting
↓
infinite loop
↳ CPU wastage

Lock variable:

1. user mode
2. busy waiting
3. multiple processes

entry section
1. while (lock != 0);
2. lock = 1
3. CS

exit section
4. lock = 0

Problem: Mutual exclusion not guranteed
→ Preemption after line 1 2 3

1. Load R0, lock
2. Cmp R0, #0
3. JNZ 1
4. Store lock, #1
   CS
5. Store lock, #0

P_1 : 1	P_2 : 1 2 3 4 CS	P_1 : 2, 3, 4 CS

Operating System

① Load R0, lock	atomic → no preemption
② Store R0, #0	
③ [redacted] CMP R0, #0	
④ JNZ ①	

Mutual exclusion violated only if preemption occurred after ①

TSL (Test Set Lock)

① TSL R0, LOCK
② CMP R0, #0
③ JNZ ①

① Mutual exclusion ✓
② Progress ✓
③ Bounded waiting ✗
④ Portability ✗

Another problem:
Priority inversion problem:

① Higher priority process arrives
② Process preempted from CS

→ Now no process can enter CS
→ Deadlock / Spin lock

Disabling Interrupt: (Kernel mode)

Preemption is done via interrupt.
If interrupt disabled, no preemption

Disable interrupt
CS
Enable interrupt

1. Mutual Exclusion ✓
2. Progress ✓
3. Bounded waiting ✗
4. Portability ✗

Strict Alteration approach / turn variable:

1. user mode
2. busy waiting
3. 2 process solution

P_0	P_1
while (turn != 0) ;	while (turn != 1) ;
CS	CS
turn = 1;	turn = 0;

turn

1. Mutual exclusion ✓
2. Progress ✗
3. Bounded waiting ✓
4. Portability ✓

Date: 6/8/2020 Page

Interested variable:

Intr[0] = F
Intr[1] = T

P0

```
Intr[0] = T;
while (Intr[1] == T);
```

CS

```
Intr[0] = F;
```

P1

```
Intr[1] = T;
while (Intr[0] == F);
```

CS

```
Intr[1] = F;
```

1. Mutual Exclusion ✓
2. Progress ✓
3. Bounded waiting ✗
4. Portability ✓

Dead lock

$I_0 = F$
$I_1 = F$

P0	P1	P0	P1	P0
$I_0 = T$	$I_1 = T$	$I_0 = F$	loop	loop
CS	loop	$I_0 = T$ loop		

(Deadlock)

Date: 5/8/2020 Page

Peterson's method:

1. user mode
2. busy waiting
3. 2 process solution (P0) (P1)

1. Interested } both variable
2. Turn }

(P0) & (P1)

```
void entry_section (int process)
{
    int other;
    other = 1 - process;
    interested[process] = True;
    turn = process;
    while (interested[other] == True &&
           turn == process);
}

void exit_section (int process)
{
    interested[process] = False;
}
```

Date: 22/08/2020 Page

Operating System

Busy waiting :→ Infinite loop in entry section. CPU resource wasta
→ Priority inversion problem

Sleep and wake up :

producer consumer problem
read write problem

count = 0 allocated
N = 100 buffer size

```
void producer (void)
{
    int item;
    while (true)
    {                                   ← (preemption)
        if (count==N) sleep();
        produce_item();
        count++;
        if (count==0)
            wakeup (consumer);
    }
}
void consumer (void)
{
    int item;
    while (true){
        if (count==0) sleep();
        remove_item();
        count--;
        if (count==N-1)
            wakeup (producer);
    }
}
```

Problem:

→ Preemption problem.
Deadlock

solution

set flag [while] wake up ~~symbol~~ call

Semaphore:

Record the wake up calls (count)

OS support needed
OS provides atomicity to
read
modify
store
↳ semaphore variable

① counting semaphore
② binary semaphore (mutual exclusion)

```
struct semaphore
{
    int value;
    Queue type L; (blocked processes)
}
```

Entry	Exit
Down	Up
~~Decrement~~ P	V
Wait	Signal

Date: / / Page

```
Down (Semaphore S)
{
    S.value = S.value - 1;
    if (S.value < 0)
    {
        put PCB in L
        sleep()
    }
}

Up (Semaphore S)
{
    S.value = S.value + 1;
    if (S.value ≤ 0)
    {
        select a process from L
        wakeup();
    }
}
```

(L → queue) (Bounded waiting guaranteed)

Binary Semaphore: (Mutex)

Mutual exclusion

Date: / / Page

Dining Philosophers Problem:

mutex m[0] ..... m[4]
process P[0] P[4]

Pi
wait (m[i]) . wait (m[(i+1) mod 4])
CS
release (m[i]) . release (m[(i+1) mod 4])

0, 1, 2, 3

P0 : P(m0) ; P(m1)
P1 : P(m1) ; P(m2)
P2 : P(m2) ; P(m3)
P3 : P(m3) ; P(m0)

(preemption) → deadlock

P3 : P(m0) P(m3) (avoids deadlock)

→ (out of order)

m1 P0 m0

m0 m1
P3 P1
m3 m2

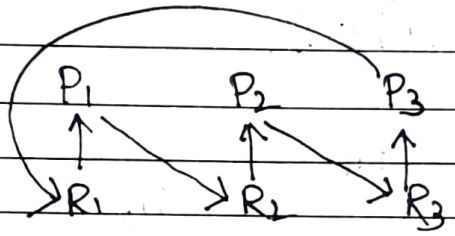
m2 P2 m3

Date: 22/08/2020 Page 25

Operating System

Deadlock: A set of processes are said to be in deadlock if they wait for happenning of an event caused by others in the same set

→ Starvation - long waiting
→ Deadlock - infinite waiting



Necessary conditions for deadlock:

1. Mutual exclusion
2. Hold and wait
3. No preemption
4. Circular wait

Resources: 1. single instance
2. multi instance

→ Max required resources for deadlock
→ Min " " to avoid deadlock

Gate 93 (n) process ← (3)

R=6

$\frac{6}{2} = (3)$

	P_1	P_2	P_3
Deadlock	↑	↑	↑
	2	2	2

A	B	C	
3	4	6	
2	3	5	(10)

minimum (11) no deadlock

Gate 2005

P_1 P_2 ..... P_n ... (m)

$\sum (s_i - 1) < m$

$\sum s_i \leq m + n$

2006 (x_i) y_i

$\sum x_i =$

$x_p + x_q \geq \min (x_p, x_q)\ y_k$

—— X ——

Deadlock Handling:

1. Deadlock ignorance
2. Deadlock prevention
3. Deadlock avoidance
4. Deadlock detection and recovery

① Deadlock ignorance:

most popular
ostrich method
both windows & unix

② Deadlock prevention:

disable one of the necessary conditions

③ Deadlock avoidance:

safe / unsafe

④ Deadlock detection and recovery:

checkpoints

Date 23/8/2020 Page

Deadlock prevention:

For deadlock to occur the must conditions

(i) mutual exclusion
(ii) hold and wait
(iii) circular wait
(iv) no preemption

If any of the necessary conditions is violated then deadlock does not occur.

(i) Mutual exclusion:

multiple process can share a resource

Spooling: printer

memory associated with printer which stores the assigned jobs

Problem: (i) spool memory limited
(ii) every resource is not sharable

So violating mutual exclusion is not practical.

Date: 23/8/2020

Operating System

Deadlock prevention

② **Hold and wait:**

A process is allocated all the required resources at the beginning of the execution.

Now this can be implemented practically only if a process declares its resources initially. Otherwise this is not practical.

Another problem
① low efficiency
② starvation

③ **No preemption:**

Problem: inconsistency

④ **Circular wait:**

Assign numbers to processes and resources. A process cannot request lower numbered resource.

→ Practically implementable.

Condition	Violation	Practical
Mutual exclusion	Spooling	✗
Hold and wait	Request all resources initially	✗
No Preemption	Take resources away	✗
Circular wait	Assign numbers	✓

Deadlock Avoidance :

Request for any resource will be granted if only the resulting ~~resource~~ state does ~~do~~ not cause deadlock.

→ Safe state
→ Banker's algorithm

→ maximum number of resources that a process require.

Assigned

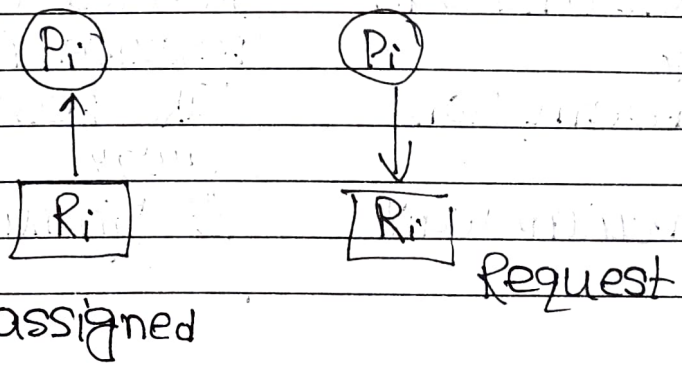
Process	R1	R2	R3	R4
A	3	0	2	2
B	0	0	1	1
C	1	1	1	0
D	2	1	4	0

Needed

Process	R1	R2	R3	R4
A	1	1	0	0
B	0	1	1	2
C	1	2	1	0
D	2	1	1	2

T = (7 6 8 4) (Total)
P = (6 2 8 3)
F = (1 4 0 1)

Resource allocation graph: (RAG)



Date: / / Page

→ Single instance resources:
cycle ⇒ deadlock

→ ~~mu~~ multi instance resource:
cycle: necessary but not sufficient

Allocation matrix:
Request matrix:

③ Deadlock detection and recovery:

OS periodically checks for ~~dead~~ deadlocks

Detection

- Single instanced → Detect cycle
- Multiple instanced → Safety algorithm (Bankers' algo/ Allocation matrix)

Recovery:

Resource → Preempt: choosing a resource to ~~pr~~ preempt is tough

Date: / / Page

Kill a process:

Choosing a process to kill is tough.

Kill the process with least amount of ~~we~~ work done.

Kill all process:

Inefficient

Rollback to a safe state:

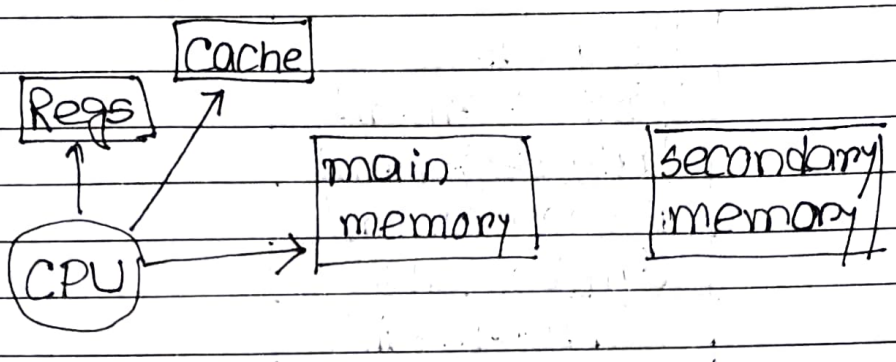
Keep check points.
Roll back to last safe state.

GATE

Date: 23/08/2020 Page

: Operating System :

Memory management :



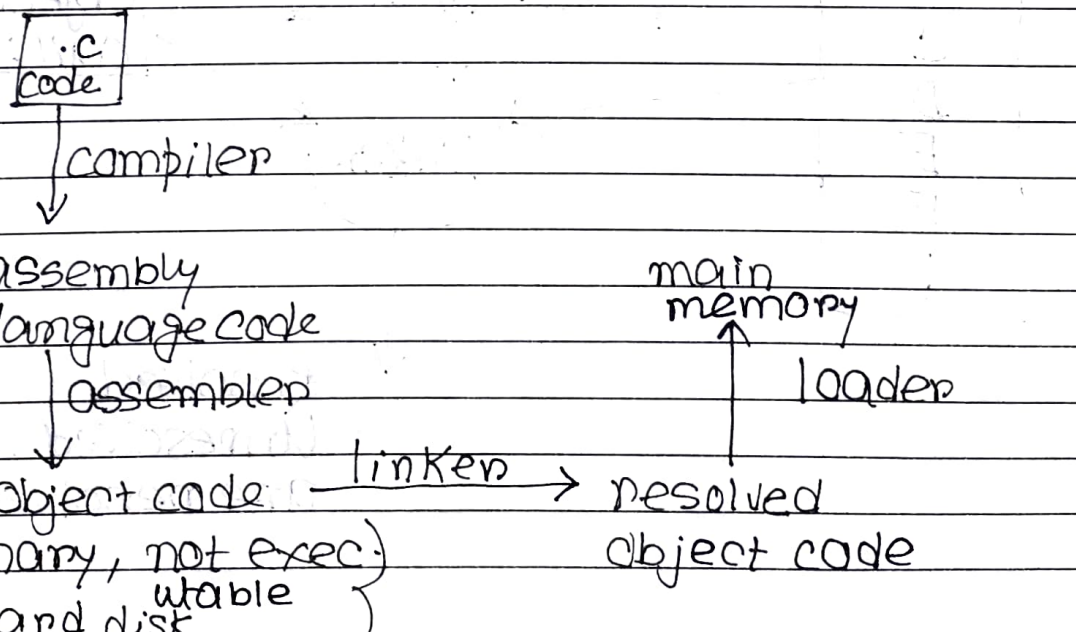
more main memory → higher degree of multiprogramming

higher more processes can be loaded → Higher CPU utilization

need for memory manager :

1. multiprogramming
2. security

Process Creation :



Object code

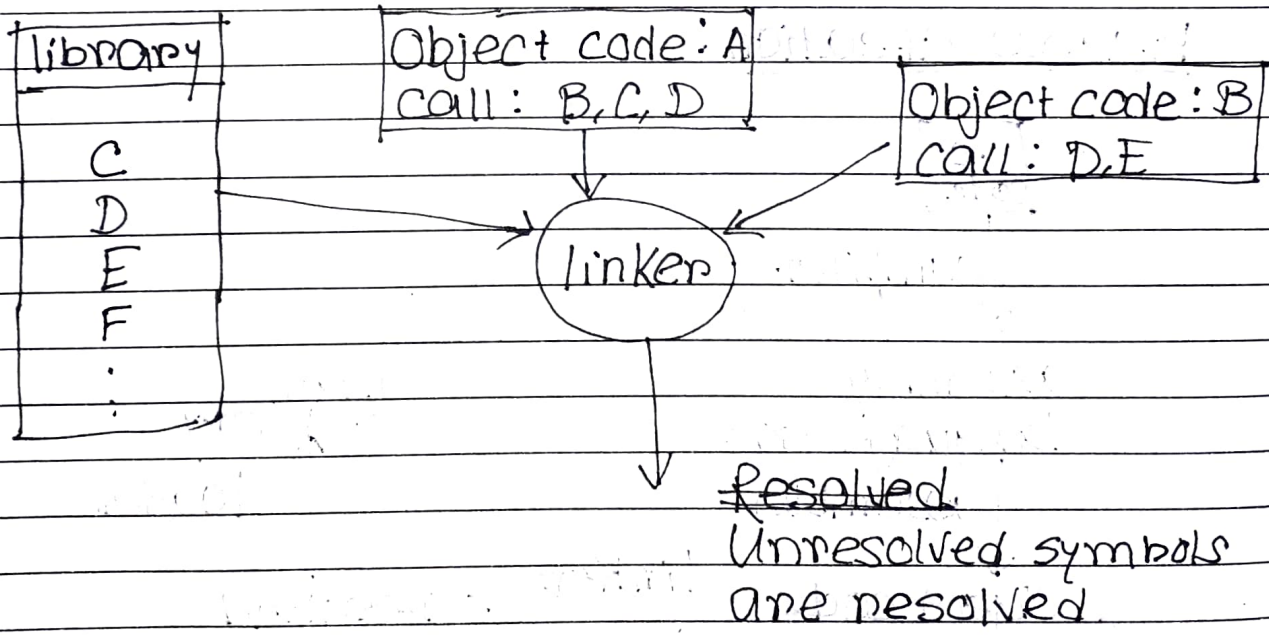
Header
Text segment
Data segment
Relocation info
Symbol table
Debugging info

Relocatable address (w.r.t. 0)
↓ (Relocation)
Absolute address (main w.r.t. the address memory where it is loaded)

Symbol table

mini( )	100
printf( )	— → unresolved

(external/ function / internal variable name)



Date: / / Page

searching for external references
(i) same directory
(ii) system library
(iii) ~~sh~~ user library

Linker (2) Phase:

Phase I: (i) segment table : segments to be loaded
(ii)
(ii) symbol table : symbols to be resolved

Phase II : Resolve unresolved references

Starting address: 0
(linker does not know absolute address)

→ Static linking
→ dynamic linking — leave unresolved
(shared library) (stub/glue code)

Ex: printf → resolved later during execution with the help of OS.
: OS searches for printf in memory (main)

Adv: (i) version change
(ii) low memory

Disadv: Page fault
If no other program is using printf, then it is not present in the main memory. So it needs to be loaded

Date: / / Page

If compiler knows the absolute (main me
address where the process will
be ~~to~~ loaded → absolute address
is used then
compile time
binding

If linker knows the ~~add~~ main
main memory address where the
process will be ~~to~~ loaded →
absolute address
link time
binding

Static linking : OS support not needed

Loader:

loader knows the address of main memory where the process will be loaded.

loader must be present in main memo

linked object code

↓ loader

Absolute address code

① Program loading: Hard disk → main memory

② Relocation: absolute address

③ Load time depends on program size

Compile time: symbolic names → relocatable address

Run time:

① Dynamic linking (DLL)
- OS support required to search for the linked library.

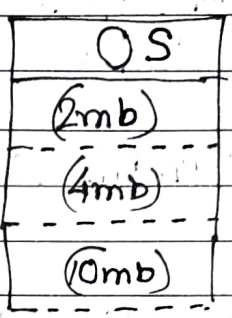
A process is not allowed to access beyond its allocated ~~spa~~ space.

q 2001 }
q 2003 }

Memory management techniques:

→ contiguous memory ~~allacti~~ allocation

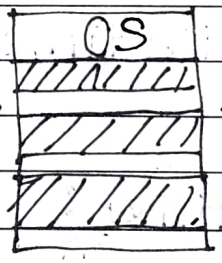
Fixed partitioning:
partitions can be of different sizes



① Internal fragmentation:
space wastage

② Maximum ~~prog~~ process size limited:
unless virtual memory used

③ External fragmentation:



Total free: 3mb
↳ A new process: 3mb
↳ It cannot be allocated.
(contiguous)

④ Degree of multiprogramming is limited

Date: 24/08/2020 Page

Relocation & protection

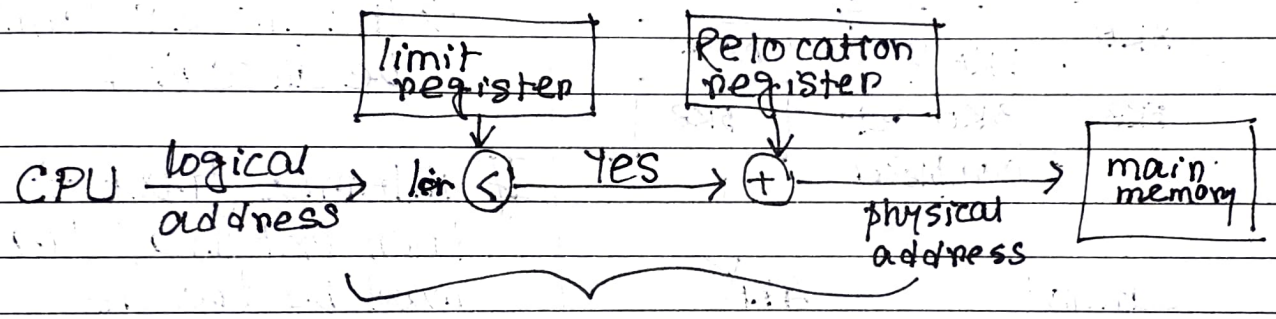
Relocation by loader → software approach

problem: preemption. when their process is loaded again at in main memory, conflict happens

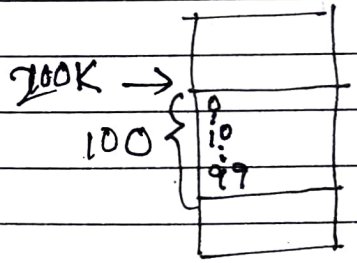
Solution: Run time binding (instead of load time binding)

CPU → logical address → memory manager → physical address → main memory

it assumes the starting address as 0



memory manager (contiguous allocation → partitioning)



limit register: 100 (protection)

relocation register: 200K (relocation)

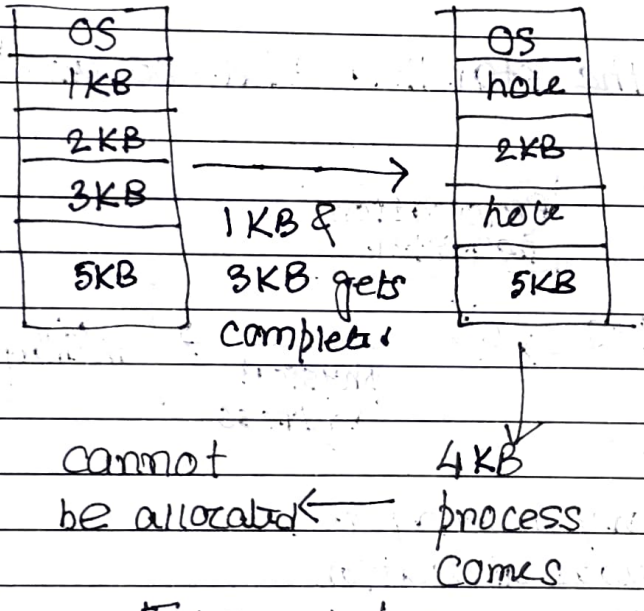
swap out & swap in: relocation register value changes

Date: 26/8/2020

swap out & swap in: limit register & relocation rg must be reserved (~~my comm~~ must not be given to any other process)

Dynamic partitioning: According to the requirement of the processes

→ no internal fragmentation.



(External frag:
→ a process comes
→ memory available (not contiguous)
→ the process cannot be allocated

If internal frag → then external frag must be present

External fragmentation (Reason: contiguous)

Date: 26/8/2020

Operating System

Contiguous memory allocation:

Dynamic partitioning:

External fragmentation:
solution: compaction

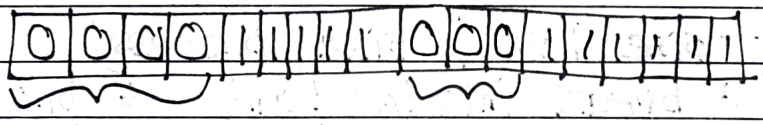
compaction/:
defragmentation — pull the free spaces and put them together.

problem → all the processes need to be copied (relocated)

Keeping track of free & occupied spaces:

Bitmap:
allocation unit (defined)
one bit → one allocation unit
hole: 0
occupied: 1

allocation unit size ↓ bitmap size ↑

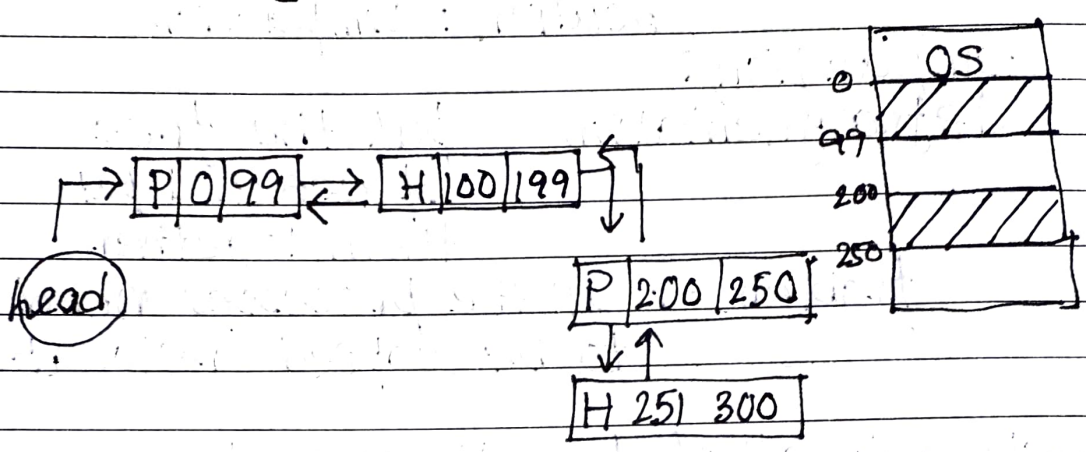


→ compaction requires searching (linear) the whole bit map

Date: / / Page

Linked list:

node | id : process(p) / hole(h)
start :
end :



① maintain the nodes in increasing order of starting address
② double linked list

—— x ——

Process allocation in a hole : both fixed & dynamic partitioning

① First fit :
Search from the ~~begin~~ head till the first hole big enough to hold the requested process

size (hole) > size (process)
↳ break into two part → process
→ new hole

→ fast
→ malloc uses first fit

Date: / / Page

(2) Next fit: instead of searching from the head, search from the point left earlier.

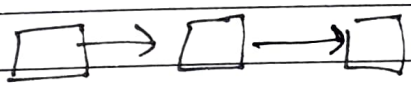
(experimentally next fit is not better than first fit, so not popular.

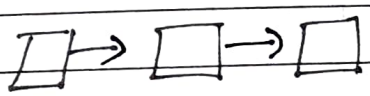
(3) Best fit:
→ Smallest possible hole
→ search the entire list
↳ time high
problem: many small holes get created, so which cannot be allocated to any other process. So space wastage

(4) Worst fit:
→ Largest hole
→ search the entire list

(5) Quick fit:
→ Most frequently used sizes
→ Linked list of that

Say 2mb, 4mb most frequently used

 all holes of 2mb

 all holes of 4mb

Problem: freeing a process creates is expensive

Date: 26/8/2020 Page

800

200	270	260	270

~~740~~ 540

~~460~~ ~~440~~ / 3

153	200	153	260	154

~~153~~ ~~460~~ ~~200~~ ~~260~~ 50 (460) 540 KB free / 3

180

~~270~~

180	200	180	260	180

181 → denied

best fit

fixed partitioning

4K 8K 20K 2K

Job	J1	J2	J3	J4	J5	J6	J7
	2K	~~10~~ 14K	3K	6K	10K	20K	2K
Execution time	4	10	2	1	1	8	6

2K

4K	8K	20K	2K

t=0 J3 J4 J2 J1

t=10 J5

t=4 J6

t=12 (J2) (11+8) = 19

Date: 26/8/2020

Operating System

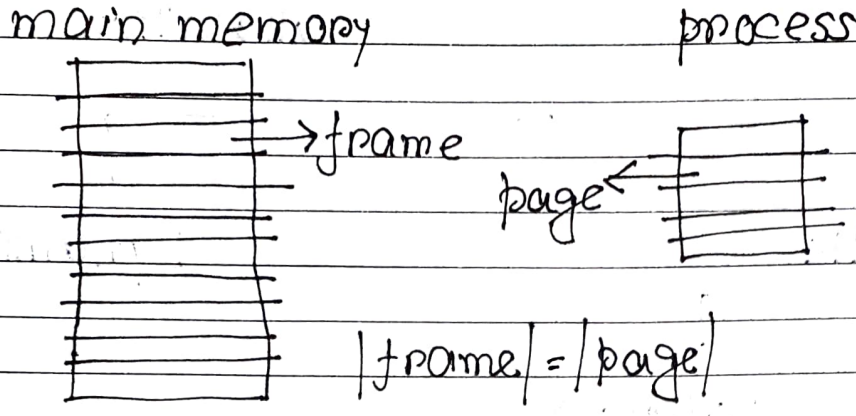
Memory management :

- Partitioning (contiguous allocation)
  - Static
    - Equal sized
    - Different sized
      - ↓
      - maximum process size is limited by the maximum ~~all~~ allocated partition size
      - ↓ solution
      - overlays → done by user
      - → load only the required parts of a process
      - ↓
      - similar to virtual memory → done by OS
  - Dynamic

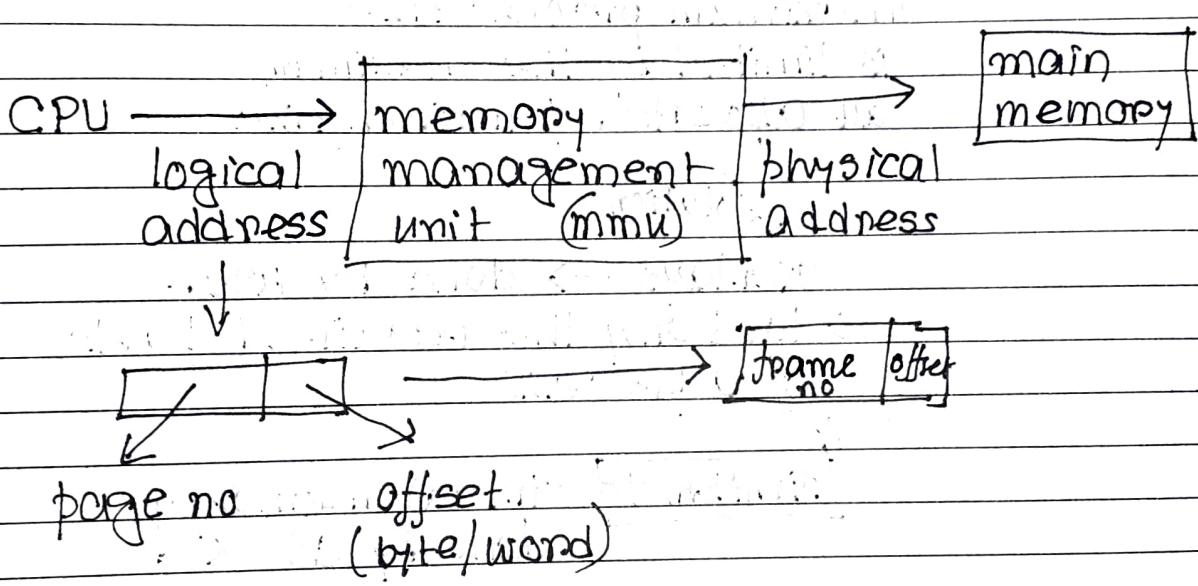
Date: / / Page

Paging:

non conteguous memory allocation



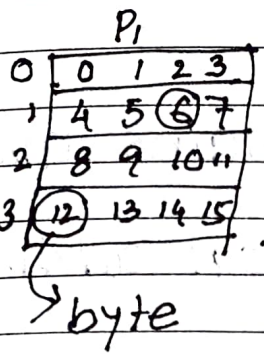
→ no more external fragmentation

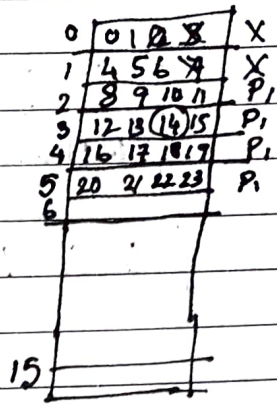


Date: / / Page

process = 16B
page = 4B

main memory = 64B
frame size = 4B

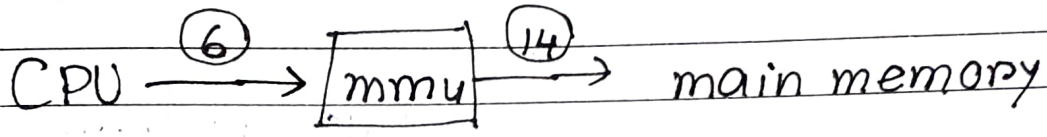




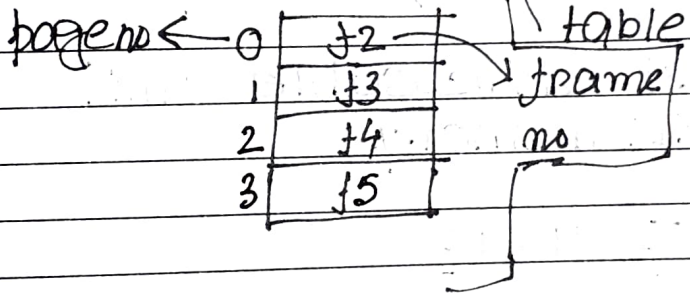
~~6~~

6

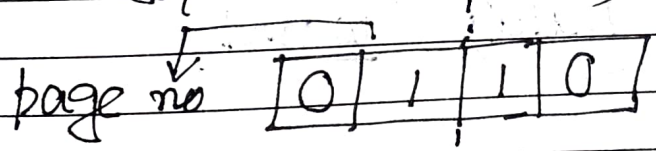
14



page table (every table has a page table)



process size = 16B (Byte addressable) 4 bits



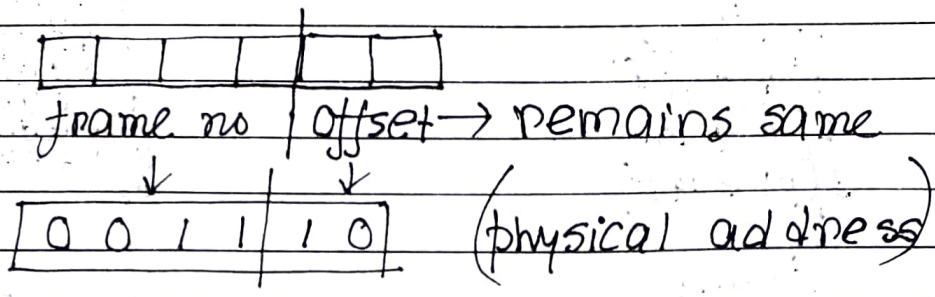
page size = 4B

2 bit to denote page

page no → (page table) → frame no

Date: / / Page

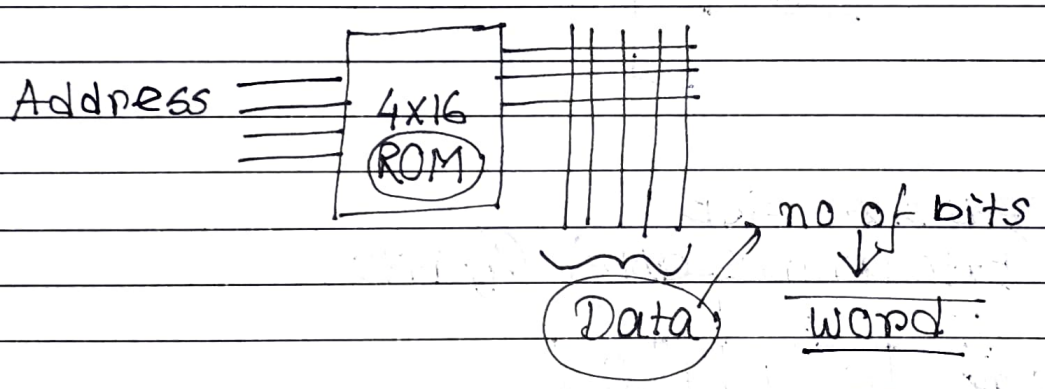
main memory 64 B
byte addressable
(2^6)



Every process has its own page table.

2^{10} = K	Kilo
2^{20} = M	mega
2^{30} = G	giga
2^{40} = T	Tera

word : smallest addressable unit in the memory



Date: 26/8/2020

Operating System

Memory management:

Physical address space = size of main memory

= 128 KB

= $2^7 \times 2^{10}$ B

= 2^{17} B

= $\frac{2^{17}}{2^2}$ words

= 2^{15} words

Word = 4B = 2^2 B

Physical address = 15 bits

Logical address space = size of a process

virtual memory : (process size > main memory size)

↓

only the required part is put in main memory

Also, virtual memory can be used in all processes to increase the degree of multiprogramming

∅ page size = frame size = P words

= $\log_2 P$ bits

(page) offset

↓

no of bits required to identify each word of a page uniquely.

PAS = main memory size = M words
LAS = process size = L words
Page size = P words

$PA = \lceil \log_2 M \rceil = m \quad LA = \lceil \log_2 L \rceil = l$

Page offset = no of bits required to uniquely identify a word with in a page
$= \lceil \log_2 P \rceil = p$

PAS = 128 MB = 2^{27} B
LAS = 1 MB = 2^{20} B
PS = 4 KB = 2^{12} B
1 word = 1 B

Page table:

page in the process = 2^{l-p}

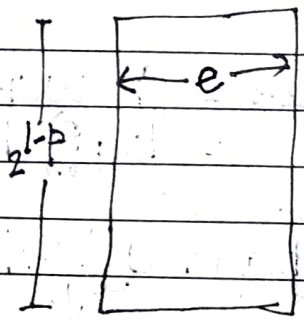
page no page offset

main memory:

frames = $\left(\frac{2^m}{2^p}\right) = 2^{m-p}$

PA

frame no



page table

page table entry size
$= (2^{l-p} \times e)$
$= (2^{l-p}) \times (m-p)$
(normally $e > m-p$) → to save other informations

# entries in page table = # pages in the process
e = each entry size
$= m-p$

Date: 27/8/2020

Multi level paging:

(Byte addressable) LA = 22 bits LAS = 2^{22} B
PS = 4 KB = 2^{12} B
offset = 12

#pages = 2^{22-12} = 1K
page table entry size = 4 B

PTS = (4 KB)

If (Page Table Size > Page size)
Page Table itself needs to be divided into pages

Byte addressable LA = 32 bits LAS = 2^{32} B
PS = 4KB = 2^{12} B
#pages = 2^{20}

PA = 44 bits (main memory = 2^{44} B)
#frame = 2^{32}

page table: (PT2)
#entries = 2^{20}
entry size = 32 bits (minimum)
= 4 B

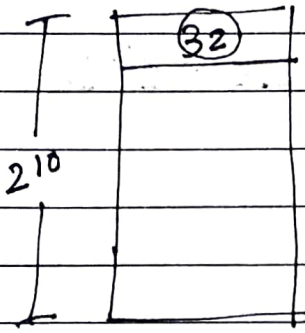
PTS = 2^{22} B
= 4 MB → {cannot fit into one frame in the main memory}

needs to be divided into pages

#pages = $\left(\frac{4MB}{4KB}\right)$ = (1K) (2nd level paging)

Date: 27/8/2020

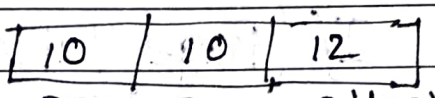
2nd level page table (PT1)



entry size
= 32 bit (minimum)
(since it ~~must~~ denotes a frame)

Page Table Base address (Register)

CPU → LA (32 bits)



Operating System

Virtual memory :

Illusion that

main memory = secondary memory

Example:

LA	Page size	PTE
48	16 KB	4B

(Byte Addressable)

LA = 48 bits

Page size = 16 KB = $(2^4 \times 2^{10})$ B

$= 2^{14}$ B

PT Level 1

Page $= 2^{48-14} = 2^{34}$

PTE = 4B

PTS = $(2^{34} \times 4)$B = 2^{36} B

PT Level 2

page $= 2^{34-14} = 2^{20}$

PTE = 4B

PTS = $(2^{20} \times 2^2) = 2^{24}$ B

PT Level 3

page $= 2^{24-14} = 2^{10}$

PTE $= 2^2$

PTS $= 2^{12}$ B

8	14	14	14

1 page table entry = 4B

Page size $= 2^{14}$ B

12		14

entry in 1 page $= 2^{12}$

10	12	12	14

PT3 PT2 PT1 Offset

S = 2^{34}

1 = 2^{14} * 2

LA = 64 B
Page size = 1 ~~mB~~ MB = 2^{20} B
PTE = 4B

8.

PT1
- #pages = 2^{44}
- PTE = 2^2
- PTS = 2^{46} B

PT2
- #pages = 2^{26}
- PTE = 2^2
- PTS = 2^{28}

PT3
- #pages = 2^8
- PTE = 2^2
- PTS = 2^{10}

8	18	18	20

Page size = 2^{20}
PTE = 2^2
# Entry = 2^{18}

LAS = 4 MB = $(2^2 \times 2^{20})$ B = 2^{22} B
PS = 4 KB = $(2^2 \times 2^{10})$ = 2^{12} B
PTE = 2^x
# pages = $2^{8\,10}$
$2^{10} 2^8 \times 2^x = 2^{12}$
$x = 4$
$2^4 = 16$ B

Fill in single level

$2^2 = 4$

Date: 27/8/2020 Page

single level paging

Page size = 4KB = 2^{12} B

PTE = 4B = 2^2 B

PTS = 4KB = 2^{12} B

page ÷ # entry = 2^{10}

LA = (x)

$$\#page = \frac{2^x}{2^{12}} = 2^{10}$$

LA = $\boxed{x = 22}$ $\boxed{LAS = 2^{22} B = 4 MB}$

Two level paging

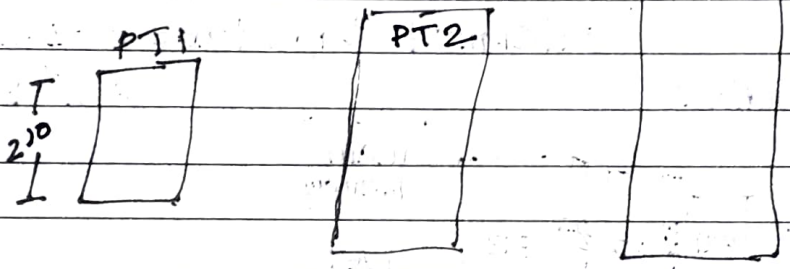
Page size = 4KB = 2^{12} B

PTE = 4B = 2^2

Outer PTS = 4KB = 2^{12} B

page

entry/page = 2^{10}



PTS

$$\frac{PT2\ size}{page\ size} = 2^{10}$$

PT2 size = 2^{10+12}

= 2^{22}

entry = 2^{20}

$$\frac{LAS}{2^{12}} = 2^{20}$$

$LAS = 2^{32}$

$\boxed{LAS = 4\ GB}$

Date: 28/08/2020 Page

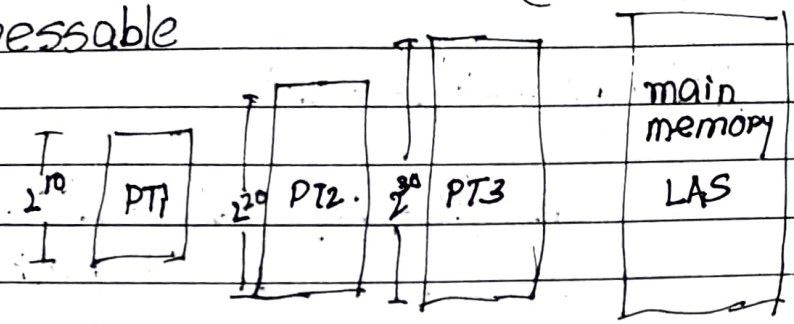
3 level paging

Byte addressable

Page size = 4KB = 2^{12} B

PTE = 4B = 2^2 B

outermost page table size = 4 KB = 2^{12} B



PT2 = 2^{22} B PT3 = 2^{32} B LAS = 2^{42} = 4 TB

—x—

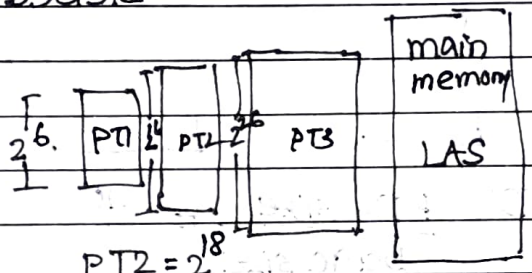
3 level paging

Byte addressable

page size = 4KB = 2^{12} B

PTE = 2^2 B

outermost page table = 256 B = 2^8 B



PT2 = 2^{18}

PT3 = 2^{16+12} = 2^{28}

LAS = 2^{26+12} = 2^{38}

8 GB

2^8 GB = 256 GB

—x—

Operating System

2001 main

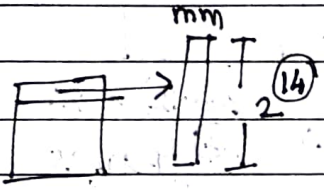
~~memory~~ AS = 64 MB
= $2^6 \times 2^{20}$ B
= 2^{26} B
#frame = 2^{14}
(14)

VAS = 32 bit
PS = 4KB
= 2^{12} B
~~#entry~~ =
#page = 2^{20}

PTE = (14)

PTS = $2^{20} \times 2^{14}$
= 2^{34}
= 8 GB

14×2^{20} bits
14 mb



$\frac{14 \times 2^{20}}{8 \times 2^3}$ B ≈ (2 MB)

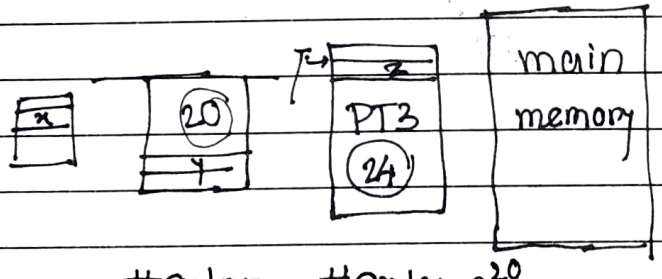
Initially virtual memory was used as only when process size > main memory size.

Now a days, virtual memory is used in all cases to increase the degree of multiprogramming.

2008 PA = 36 bits LA = 32 bits

PS = 4KB = 2^{12} B
PTE = 4B = 2^2 B

2	9	9	12
PT1	PT2	PT3	offset



36 − 12 = (24)

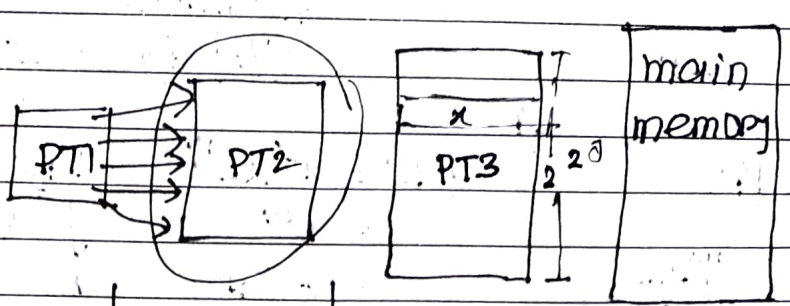
#entry = #entry = 2^{20}
size = $2^{20} \times$
$2^{20} \times 24$

Date: / / Page

PA = 36 bits
LA = 32 bits

PS = 4 KB = 2^{12} B
PTE = 4 B = 2^2 B

2	9	9	12
	24		



(36-12) = 24

#pages = 2^{20} 2^{24} frames

x ≥ 24

PTS = x. 2^{20}

x = 24

(9) 2^9

(1 page) = 2^{12} B
1 entry = x
1 page $\frac{2^{12}}{x}$

$\frac{2^{12}}{x} = 2^9$

$x = 2^3 = 8$

PT2
#pages = 2^2 = 4

PT3
#pages 2^9 X

PT2 PTS = $2^{24} \times 2^{20}$
#pages:

$\frac{2^{10} \times 2^2}{2^{12}}$ (1)

(20) $\frac{2^4 \times 2^{20}}{2^2 \times 2^{20}}$

$\frac{}{2^{12}}$

2^{10}

Date: 28/08/2020 Page

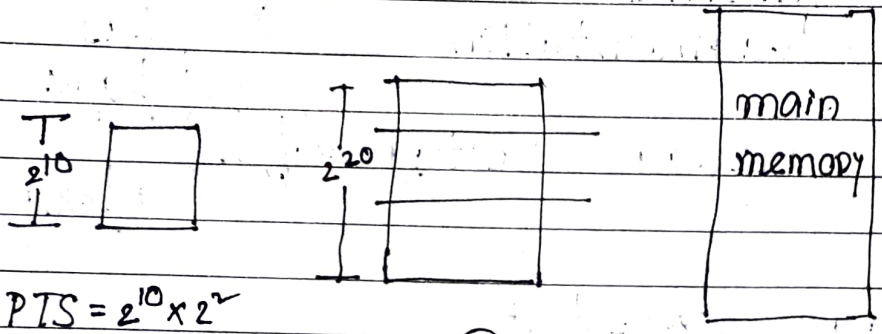
VA = 32 PA = 32

Byte addressable

Two level paging

PS = 4KB = 2^{12} B

PTE = 2^2 B



PTS = $2^{10} \times 2^2$

$= 2^{12}$

4x8 − 20 = 12

PTS = $2^{20} \times 2^2$

$= 2^{22}$

4x8 − 20 = 12

2^{32}

#frames = 2^{20}

confusing

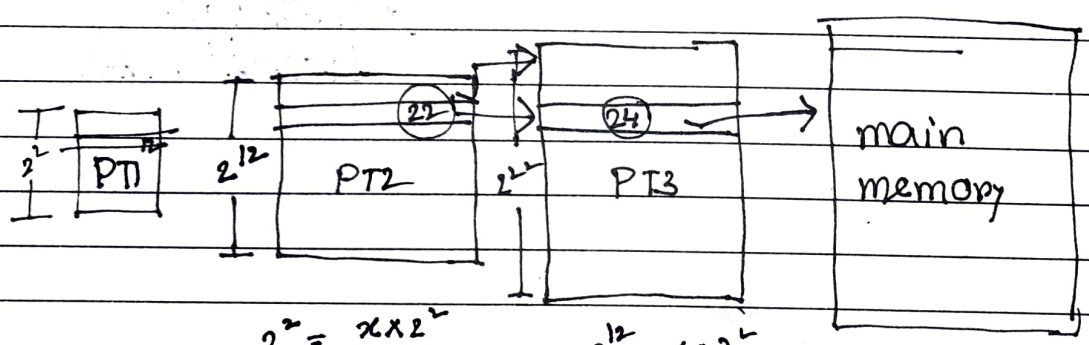
1 page = 2^{10} entries

LA

20	12

10	10	12

22, 24



$2^2 = \frac{x \times 2^2}{2^{12}}$

$2^{12} = \frac{x \times 2^2}{2^{12}}$

$\frac{2^{32} \times 2^2}{2^{12}}$ 2^{22}

36 − 12

2^{24}

#frame = 2^{24}

2	9	9	12

Date: 28/08/2020 Page

Optimal page size:

PS ↑ → PTS ↓ → overhead ↓

→ internal fragmentation → Last page

average wastage = (p/2) → page size

page size = p B

PTE = e B

avg VAS = S B → last page of process

avg overhead = wastage $\left(\frac{p}{2}\right) + \left(\frac{S}{p} \times e\right)$ → page table

$$O = \frac{p}{2} + \frac{se}{p}$$

$$\frac{dO}{dp} = \frac{1}{2} - \frac{se}{p^2}$$

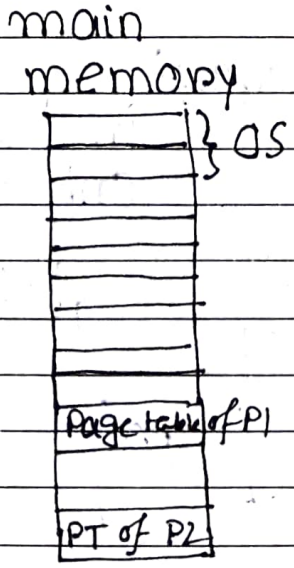
$$\boxed{p = \sqrt{2se}}$$ → PTE

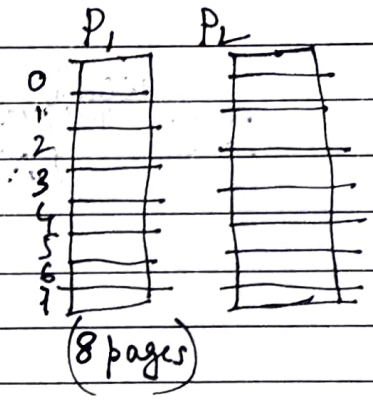
↓ average process size

Date: 28/08/2020 Page

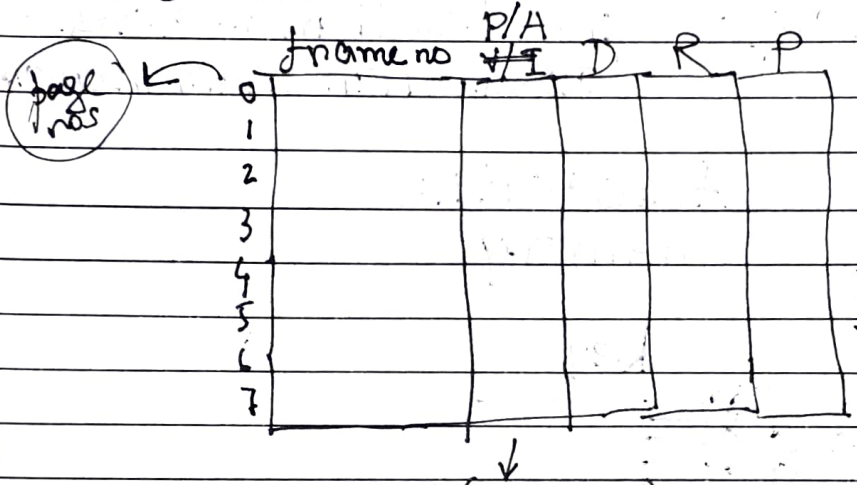
Operating System

Virtual memory :





Page table of P1



Present/absent bit (~~valid bit~~) :: 1 → present in main mem
0 → not present in main m

Dirty bit / modified :: After being loaded in main memory, modified or not.

Reference : in last clock cycle, is it page is referred or no

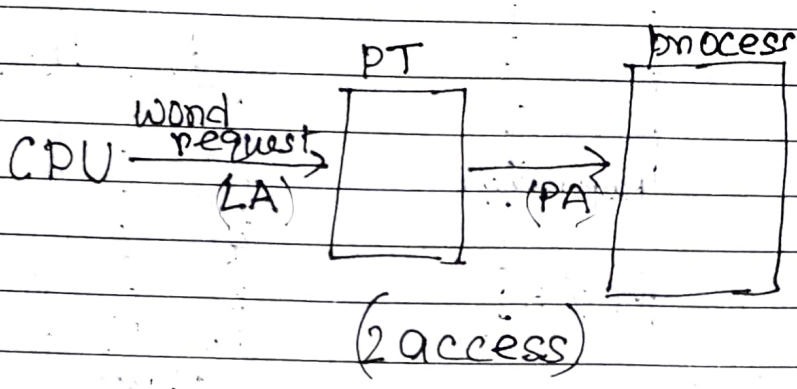
Protection : permission to modify read/ write/ execute

violated
Trap

Date: 28/08/2020

Page Fault: → stop the process
→ read the page from secondary

Locality of reference:
... in page table but only these pages

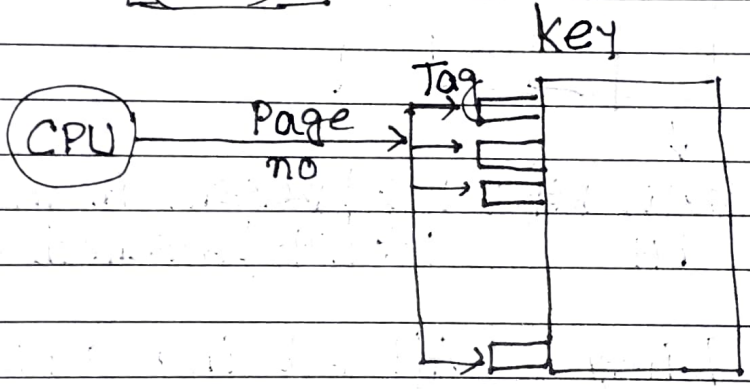


(2 access)

cache memory: cheaper than registers
faster than main memor

Translation Look aside Buffer (TLB)

Associative memory



TLB holds the frequently used (page table entries)

Date: 28/8/2020

cache/TLB hit : p
cache/TLB miss : (1-p)

$$\underbrace{p(t+m)}_{\text{hit}} + \underbrace{(1-p)(t+m+m)}_{\text{miss}}$$

- t → cache access time
- m → main memory access time
- m → page table access (main mem)
- → single level paging

$$p(t+m) + (1-p)(t+km+m)$$

- k → k level page table

Ex :

—x—

TLBA MA TLBH PT level

20 ns 100ns 80% 1 $0.8(20+100)+0.2(20+100+100)$

(no page fault) $= (0.8 \times 120) + (0.2 \times 220)$

$= 96 + 44 = 140$

" " " 2 $0.8(20+100) + 0.2(20+2\cdot100+100)$

$= (0.8 \times 120) + (0.2 \times 320)$

(no page fault) $= 96 + 64$

$= 160$

—x—

Date: 29/08/2020 Page

Demand paging:

Do not load any page in main memory until it is required.

Page fault:

present bit : 0
↳ not present in main memory
↓ user process
Trap (hardware interrupt)
↓ context switch
OS
↓ read request
Read from secondary mem
↓ context switch
OS gives up control

Thrasing: page fault > 50% (normally)

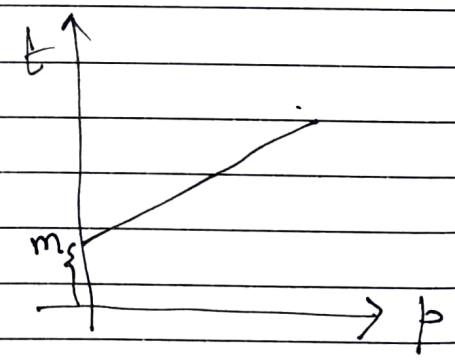
Page fault time/ service time:

Ex: $p(f+m) + (1-p)(m)$

$\frac{1}{10^6} \cdot 10 + \frac{9}{10^6} \cdot 20 \cdot 10^6$

$\frac{1}{10^6} \cdot 10 \cdot 10^6$

$p(f+m) + (1-p)m$
$= pf + pm + m - pm$



$t = pf + m$

$\frac{1}{10^6} 10 \times 10^6 + 20$
$= \boxed{30 \text{ ns}}$

Date: 29/08/2020

Operating System

Ex :

page fault time = 100
replace dirty page = 300
main mem = 1

(p) page fault rate
(p) (page fault → dirty) rate

avg access time = $p(p.300 + (1-p)100) + m$
(400p)

$= p(400p + 100 - 100p) + m$
$= 300p^2 + 100p + m = 1$

$200p^2 + 100p + 1 = 3$
$200p^2 + 100p - 2 = 0$

$300p^2 + 100p + 1 = 3$
$300p^2 + 100p - 2 = 0$

$10000 + 4.2.200$
11600

$10000 + 4.2.300$
12400

~~$\sqrt{116} < 110$~~
$\sqrt{b^2 - 4ac} < 110$

$p = \frac{-100 \pm \sqrt{b^2 - 4ac}}{2.200.(-2)} > \frac{-100 \mp 110}{-}$ $< \frac{-100 + 110}{}$

$\frac{11}{40}$ $\frac{220}{800}$

(2a)

$\frac{-100 - 110}{}$

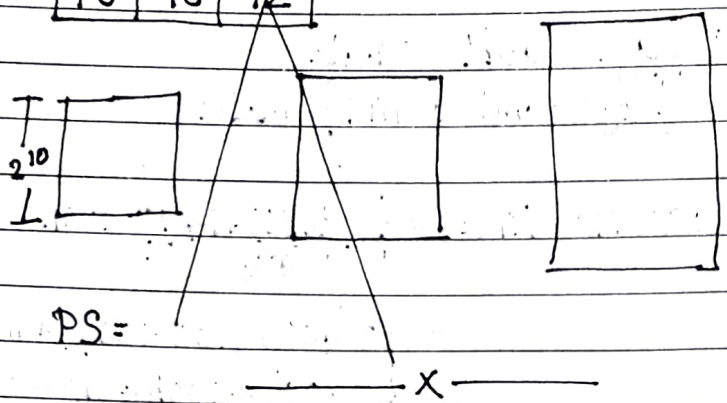
$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

Answering

—— x ——

(b) VA(LA) = PA = 32 bits
Byte addressable
2 level paging
PTE = 4B = 2^2B

10	10	12



PS =

——— x ———

Inverted page table:

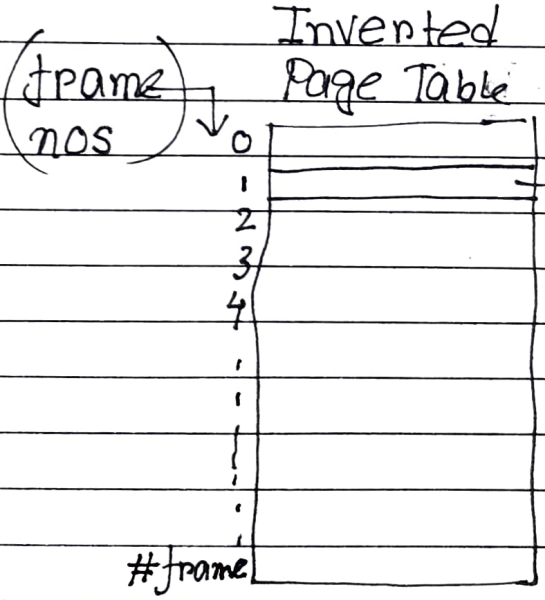
During any time, maximum total number of pages (from all processes) is = frame no.

↓

One page table ~~entry~~ for each of the processes

↳ At one page table takes at least one page (frame) in main memory

↳ Space wastage



→ ① PID + ② Page no + Page offset

→ All the entries must be searched for one access. Time requirement high.

Date: 29/08/2020

Virtual memory implementation:

→ frame allocation	→ page replacement
how many frames to be allocated in main memory for this process	which page to be replaced

minimum requirement:

one instruction (may):
1. 1 code page
2. 2 data page
3. 1 stack page

max: size of the process

→ Equal allocation:
(static allocation)
30 frames available
3 process
(30/3) = 10 page (frame) for each

→ Weighted allocation:
depending on the size of the processes

→ Dynamic allocation:
→ Shrink & grow during execution
→ Priority based

Date: 29/08/2020 Page

Page replacement algorithm:

→ local: replace only from the frames allocated for this process

→ global: considers all the frames

used in practice

Demand paging:

Reference string: CPU generated request contains page number in order

Optimal page replacement algorithm:

Replace the page which wont be referenced in the longest future (which is impossible to predict precisely in practice)

LRU:

FIFO:

Date: 03/9/2026 Page

Operating System

4 7 6 1 7 6 1 2 7 2

LRU

4	1
7	2
6	7

#page fault = 6

FIFO

4	1
7	2
6	7

4	7	6	1	2	7

#page fault = 6

1 2 3 4 2 1 5 3 2 4 6

Optimal

1	3
2	6
3	4

#page fault = 7

LRU

1	4 5 4
2	3 6
3	1 2

#page fault = 10

FIFO

1	4 5 6
2	1 2
3	5 4

1	2	3	4	1	5	3	2	4	6

#page fault = 10

Belady's Anomaly:

Optimal replacement: increasing the allocated no of frames decreases the number of page faults

In case of FIFO it ~~does~~ may not occur. → Belady's anomaly

Stack algorithm:

Property — Reason for Belady's Anomaly.

Pages present with n frame $\subseteq$ Pages present with (n+1) frames

Stack property

If a replacement algorithm follows stack property, it is called stack algorithm.

Stack algorithms do not show Belady's anomaly.

FIFO does not follow stack property.

Optimal, LRU → stack algorithm.

Loop:

1 2 3 4 5 6 1 2 3 4 5 6 1 2 3 4 5 6

Optimal

~~1~~	3
2	4
~~3~~	~~4~~ 5
4	~~5~~ 6

most recently used pages are getting replaced

#fault = 10

Date: / / Page

1 2 3 4 5 6 7 8 9 9 8 7 6 5 4 3 2 1

Optimal

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

least recently used pages are getting replaced.

Page Replacement algorithm implement:

Referred bit: if referred in this clock cycle then set as 1. At the end of clock cycle make alles as 0.

Modified bit:

This clock cycle ←	R	M → overall life time since fetche
↓	0	0
	0	1 → modified in prev cycle
	1	0
	1	1 → replacement cost high due to M=1

Date: / / Page

Working set algorithm:

It assumes that the nearest future is close approximation of recent past.

Dynamic frame allocation.

A window is taken

1 2 3 1 2 4 1 4 2 1 5 1 2 4 2 1

~~Δmax =~~

(Window size = 4) → Predetermined for a system

Requirement

↓

All processes use this value only

W = {1}
W = {1,2}
W = {1,2,3}
W = {1,2,3}
W = {1,2,3}
W = {1,2,3,4}
W = {1,2,4}
W = {1,2,4}
W = {1,2,4}
W = {1,2,4}
W = {1,2,4,5}
W = {1,2,5}
W =

If page faults are increasing, increase window size by (1).

Avg frame requirement: $\frac{\sum w_i}{n}$

Date: 03/9/2025 Page

Operating System

File System :

Secondary memory

→ creating
→ writing
→ reading
→ repositioning
→ deleting — remove data + metadata
→ truncating — remove data only accessible

Logical block ⟷ smallest unit of
Physical block secondary memory

File System
↓
Directory
↓
Files — metadata
- name (.extension) : human readable
- Type : video/audio/ txt
- Location
- Size
- Protection : permissions
- Time & date :
- Identifier

Read/Write pointer → Last read/modified

Accessing a file:

(I) File pointer
(II) File open count
(III) Disk location of the file
(IV) Access rights

File table:

For each process
(I) File pointer
(II) Access rights

Open file table:

Global
(I) Disk location of file
(II) File open count

name	location	open count

When open count = 0, delele ← that entry from the table.

How many processes has have opened that file?

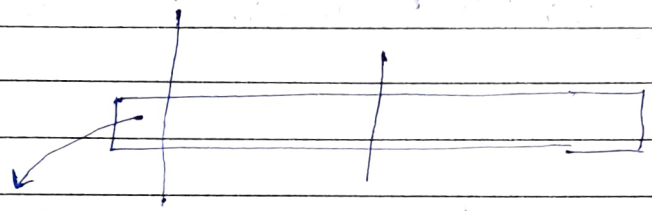
Date: 03/09/2020

Block size = 512 B (normally)

Access methods:

1. Sequential access
2. Direct access / Random access
3. Indexed access → sorted on some field

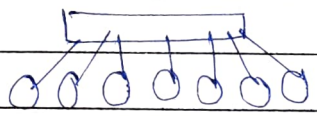
Partitioning: ⟶ multiple file system / partition / volume / minidisk / multiple OS



Directory

Different directory structures

1. Single level

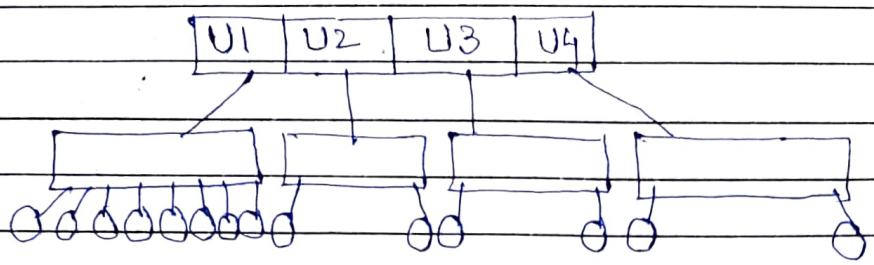


Adv: 1. implementation
2. creating & searching

Disadv: 1. Naming conflict
2. Security problem

(ii) Two level directory:

→ master file directory (MFD)
→ contains info about the users



- PWD:

Adv : (i) Name conflict resolved
(ii) Security problem resolved

Disadv:
(i) System files
→ copy ~~all~~ in all user directory
→ one special directory shared by all users containing system files

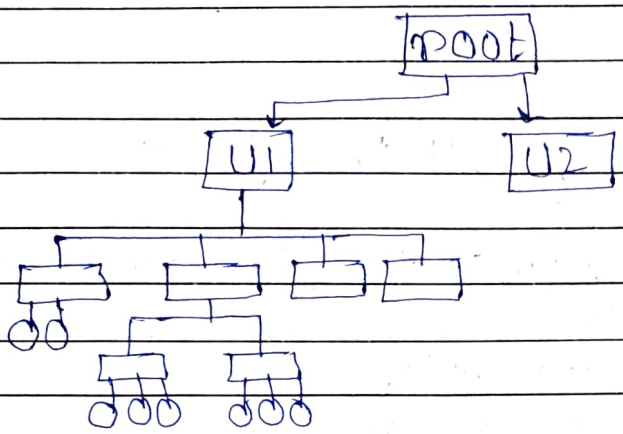
Date: 03/09/2020 Page

Operating System

File system

(iii) Tree structured directory:

Extension of two level directory



Path:

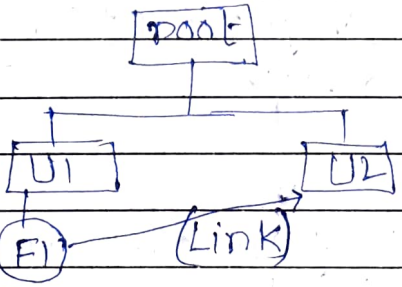
(i) absolute path name

(ii) relative path name w.r.t. pwd (present working directory)

r-w-x permission
→ files can be shared b/w users

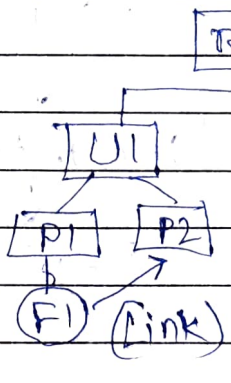
(iv) Acyclic graph structured directory:

Adv: File sharing and directory



Disadv: File deletion
(i) → dangling pointer

(ii)



Date: 03/09/2020 Page

Counter of links

Do not delete a file until link counter = 0 ⇒ all the links to the file are removed

File system:

managing all the files

→ storage and access to file contents
→ file structure
→ disk space allocation
→ freed space recovery
→ track data location
→ interfacing other parts of OS to secondary storage

File system structure:

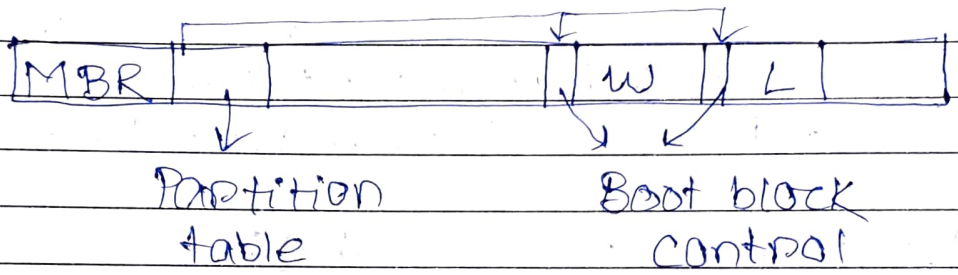
Application (process) ask/request file operation
↓
Logical file system: metadata, directory path structure
↓
File organization module : logical block ⇒ physical block
↓
Basic file system : I/O contrd. buffers
↓
I/O control : device drivers, interrupt handlers
↓
Devices

Date: 04/09/2020 Page

Partitioning of hard disk : partition / volume / mini disk

BIOS : present in ROM.
- Basic Input/output
- System

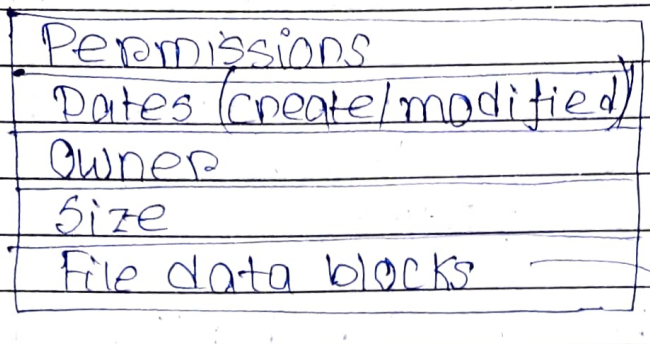
→ Fetches the 0th block from the hard disk. → MBR (master boot record)
- contains partition table
- shows the option of diff OS



Boot Control Block : Per volume/partition
Info about that OS
also known as
- boot block (Unix)
- partition boot sector (ntfs)

Volume Control Block : Next to boot control block
How many blocks in that partition, size of each block
- Super block (Unix)
- master file table (NTFS)

File control block:



Data structures related to File System

Main memory:

performance improvement (similar to cache)

- Loaded at mount time
- Deleted at unmount time

(i) **In memory mount table:** info about each mounted volume

(ii) **In memory directory structure cache:** recently accessed directory info

(iii) **System wide open file table:** FCB of all opened ~~blocks~~ files

(iv) **Per process open file tables:** For each process. It contains pointers to system wide open file table

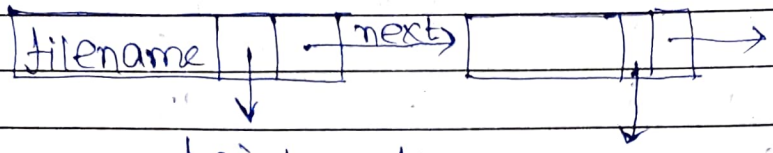
Date: 04/09/2020 Page

Operating System

File system:

Directory implementation:

① Linked list:



Problem:
- Always need to search the whole table

Create/
delete/access

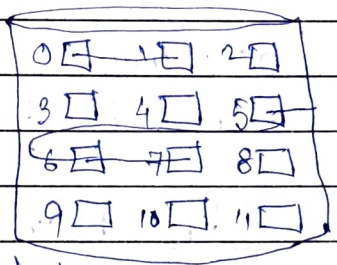
② Hash table: Hashing the name/path along with list (linked list) of files

Allocation methods:

Allocate space to files

① Contiguous allocation:

file	start	length
f1	0	2
f2	5	3



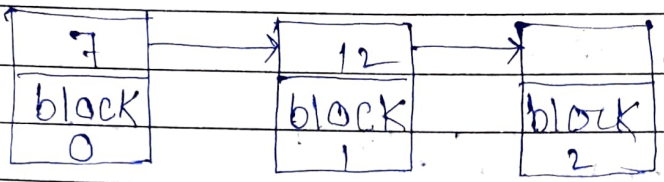
Adv: → Reading performance high
→ Simple implementation

Disadv: → Fragmentation (both internal & external)
→ Append data

Date: 04/09/2020 Page

(2) Linked list allocation:

fileA

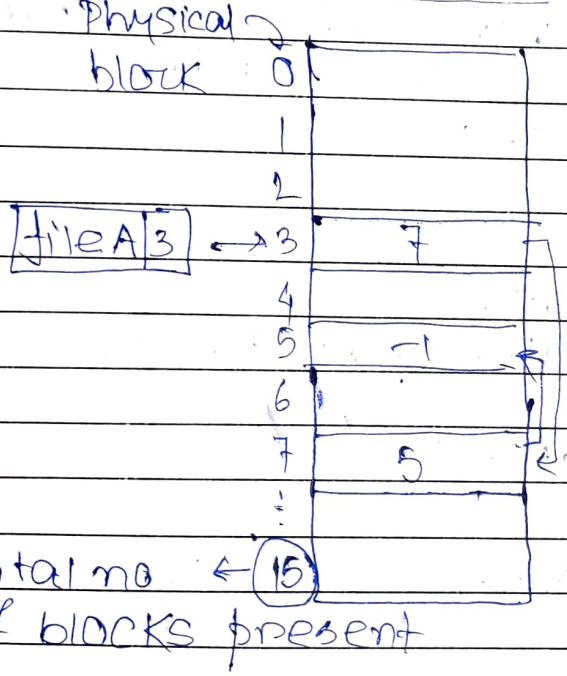


Physical block: 4 7 12

Adv: No external fragmentation

Disadv: → Space wastage due to pointers
→ Random access slow.

(3) File Allocation Table:



Present in main memory (in memory data structure)

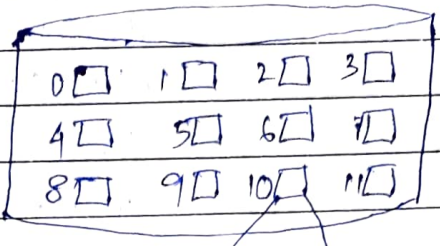
- Adv: Random access

Disadv: Large size of FAT

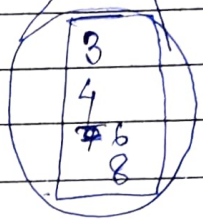
(4) Indexed allocation:

↳ inode / index block
↳ for a file ~~att~~, all the physical blocks it covers

file	block in which inode present
file1	10



file1 is contained in 3, 4, 6, 8



Problem: index block too large ← huge file
↓
cannot be contained in one block

Soln:

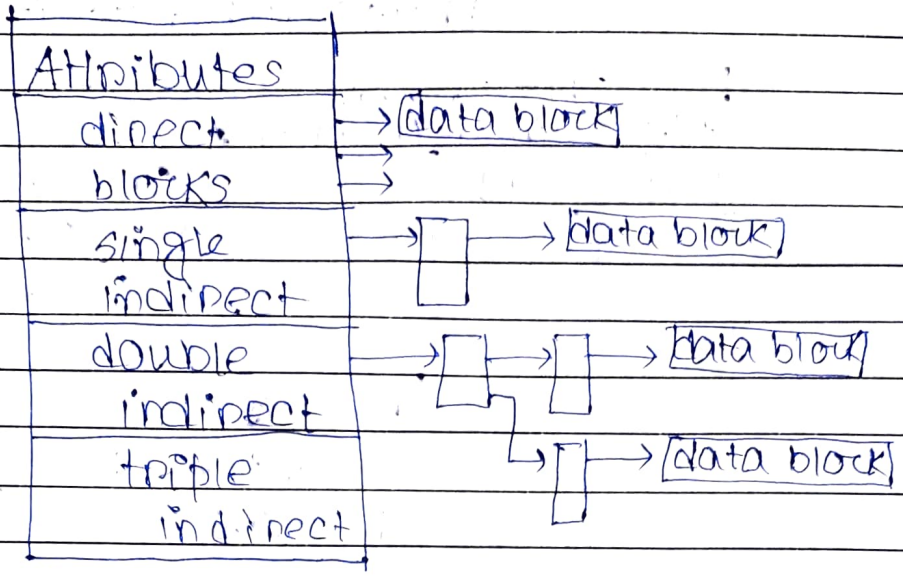
(i) Linked list of multiple index blocks

(ii) Multilevel indexing: B Tree / B+ Tree

(iii) Hybrid of (i) & (ii): most popular in unix

Unix inode implementation:

(Linked list + multilevel indexing)



Free Space Management:

① Bit Vector: one bit for each block

② Linked list: free list

Address of first free block stored in some specified place

Date: 04/09/2020 Page

Operating System

File management

Secondary memory

Disk Scheduling:

Since hard disk is the slowest, so must be scheduled efficiently

① Seek time: move the read write head to the appropriate cylinder on the track

① FCFS: simplest technique
→ performs operations in the order of requested
→ no starvation

② SSTF: Shortest seek time first regardless direction
→ starvation
→ switching direction slows down
→ not the most optimal

③ SCAN: (Elevator algorithm)
→ Go from outside to inside then inside to outside servicing requests
→ Goes till 0.

C SCAN: circular scan

→ Move inward till 0
→ start from the outermost
→ move inward

④ LOOK:

→ ~~Outw~~ Go inward till the lowest request, not till 0

C-LOOK:

→ Go ~~inw~~ outward till the highest req
→ Start from the lowest req & go outward

System calls:

→ function call
↳ present in OS

System functions (unix)	write()
	read()
	open()
	close()

① can be called directly in a C program
② can be called via library function
calls → printf() calls → write()

context switching

user ⟶ OS

① parameter passing
→ using registers
Problem: large size/number of parameters

→ saving at an(allocated) memory
→ stack
↳ store all the parameters in a stack and call the OS function
When all stack values (parameters) are used up, free up the space

System calls needed to be provided to user functions:

⇒ Process control:
- → end, abort → finish before completin
- → load, execute
- → create / terminate process
- → get / set process attributes
- → wait for some time / sleep
- → wait / signal (semaphore)
- → allocate and free memory

(using these OS functions a process deals with another process)

⇒ File manipulation:

A process deals with files using the OS functions:

- → create / delete
- → open / close
- → read / write / repositioning
- → get / set file attributes

⇒ Device manipulation:

- → request device, release device
- → read, write, reposition
  scanner printer hard disk
- → get / set attributes
- → attach / detach devices

Operating System

⇒ **Information maintenance:**

→ Get/set time or date
→ get/set system data
→ get/set process, file, device data

⇒ **Communication:**

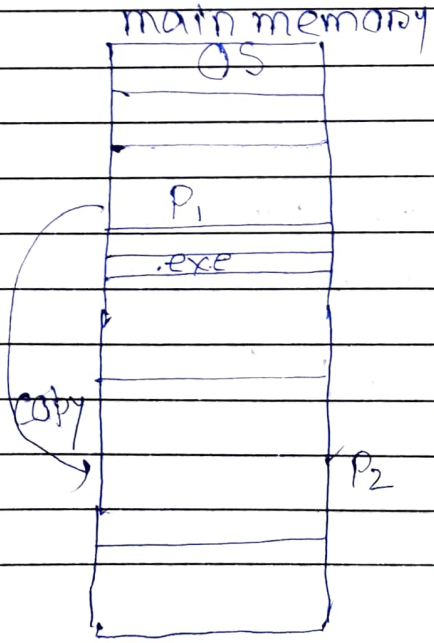
→ create/delete communication connection
→ send/receive messages
→ tranfer status information
→ attach/detach remote devices

—— x ——

FORK / EXECVE

parent P_1

P_2 child



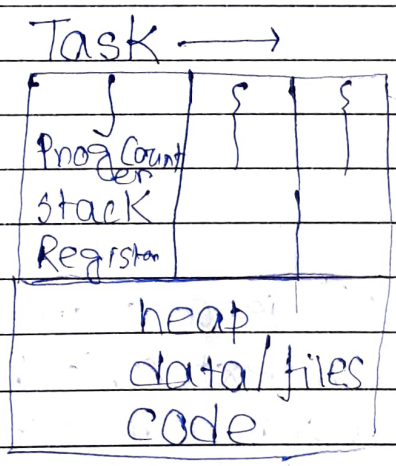
Fork() is a system call
So, context switching (usermode → kernel mode) needed → overhead

```
int i = fork();
if (i==0)
{
    //child
}
if (i!=0)          (pid of child)
{
    //parent
}
```

execve

Replace a ~~PCB~~ process with another

Threads:



Thread: sequence of execution | states are same as a process: new, ready, running

user creates a new thread (user level threads)

Date: 06/09/2020 Page

Since threads are implemented at user level, OS is unaware of its existence.

→ one thread asks for I/O, whole process (all the threads) gets ~~stopped~~ blocked

↓

This is solved by asking the OS to create threads : Kernel level threads. (Problem: context switching)

Another problem of user level threads:

Unfair scheduling

P_1 : 100 threads
P_2 : 1 thread

For user level thread: $P_1\ P_2\ P_1\ P_2\ P_1\ P_2$ (same time) (Round Robin)

↓

Kernel level thread

$P_1\ P_2\ P_1\ P_2\ P_1\ P_2$

(100x) (x) (x) time quanta

Kernel level threads are less expensive than creating another process (fork)

Date: 06/09/2020 Page

Hybrid threads: (used by Solaris)