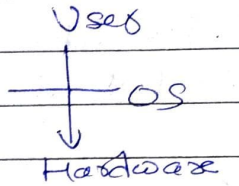
Kernel: It is a computer program at the core of computer's OS and has complete control over everything in the system. Portion of OS, always resident in memory.

AMAS Page No. Date

Operating System

Operating is a system software. It acts as an interface between the applications and hardwares of a computer



Need of operating system
→ User-process synchronizations
→ Safety and Security
→ Abstraction → primary goal of os is to provide convenience.
→ ~~Role~~

* Throughput: No. of tasks executed per unit time.
Linux has higher throughput.

* Functionality of OS
1. Resource manager: Multiple users accessing resources parallelely. The OS decides how much to be allocated to a particular user for the smooth functioning of system.

2. Process manager: Execution of different processes by using CPU (CPU scheduling)

3 Storage Management: (Hard disk)
→ Through file systems
Efficiently storing data.

4. Memory Management (Primary → RAM)
→ Allocation and deallocation of 1° memory to process in execution done by OS

5. Security and Privacy
→ Security between processes
→ Authentication

Types of OS

1. Batch 2. Multi-programmed
3. Multitasking 4. Real time OS
5. Distributed 6. ~~Networked~~ 7. Embedded
6. Clustered

1. Batch OS: Similar kinds of jobs are executed together.
CPU remains idle if the current process of the batch goes for I/O.

2. Multi-programmed: (Non-preemptive)
CPU get allocated to other processes when current process demands I/o time.

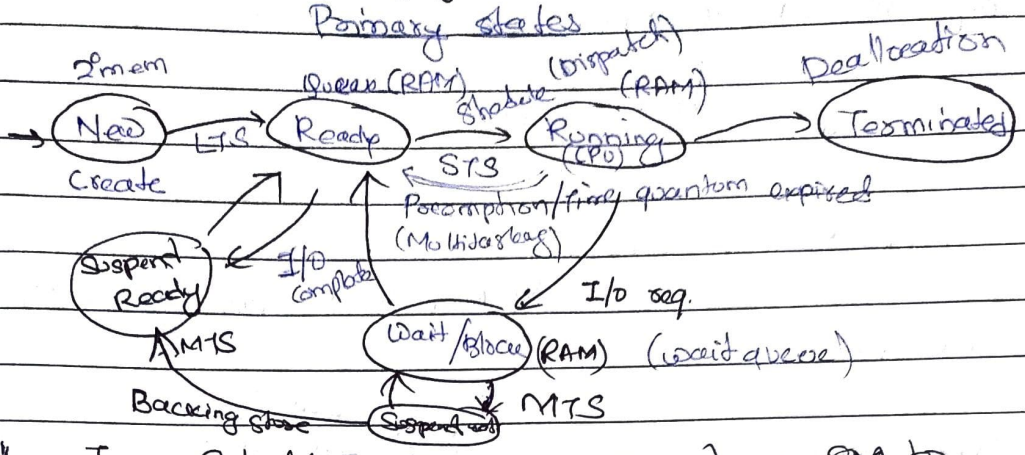
3. Multi-tasking/Time sharing OS. (Preemptive)
→ Response time is improved as CPU gets allocated to each process for a stipulated amt of time
→ Idleness is also reduced as CPU is always having a process to execute with it.

4. Real-time OS
→ Hard: Delays are not tolerable, process has to complete its execution at sharp deadline
→ eg Missile launch

→ Soft → Process has to complete it's execution within a range of time.

5. Distributed: loosely coupled machines; geographically ~~dist~~ separated.
6. Clustered: Multiple devices connected to a local machine to form a cluster (Fault tolerance)
7. Embedded: Specific functionality

Process States / Lifecycle



* Long Term Scheduler: Bring processes from SM to Ready Queue, as max no. as possible

* Short Term Scheduler: Ready queue to Running vice-versa

* Medium Term Scheduler: Wait queue

Additional states: Suspend wait (when wait queue is full)
: Suspend Ready (Ready queue full and a high priority task comes in)

* If a process continuously tries to go to wait queue but it is full then it is send to suspend ready, this is called backing store.

Important Linux commands

① chmod ugo=r note filename
(read permissions all 3)

	r	w	x	r	w	x	r	w	x
	u			g			o		

change mode
u (user) r w x
g (group)
o (others)
read, write, execute

② chmod ugo+rwx note in octal
↳ chmod 666 note

r	w	x
4	2	1

initially 0

lseek (moves read/write headers (pointers)

eg lseek(n, 10, SEEK_CUR);

n → file desc; 10 → to where; SEEK_CUR → from where

lseek(n, 5, SEEK_SET)

↳ Move to 5th pos (file pointer)

SEEK_END.

Date: 11/07/21

System call : when process in user mode requires access to a resource via requests kernel via system call. Can directly access hardwares and make things happen.

e.g write() syscall writes on monitor screen (open())

(1) File related (2) Device (3) Information (getpid) (getpc())

(4) Process Control (fork) (5) Communication (pipe) (interprocess)

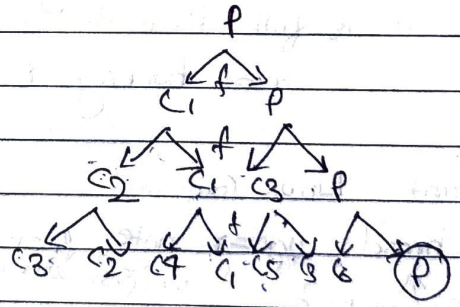
: fork() : To create a child process of a process

Returns 0 → child

+1 → +ve parent

-1 → child not created

e.g
```
main(){
fork();
fork();
fork();
printf("Hello");
}
```



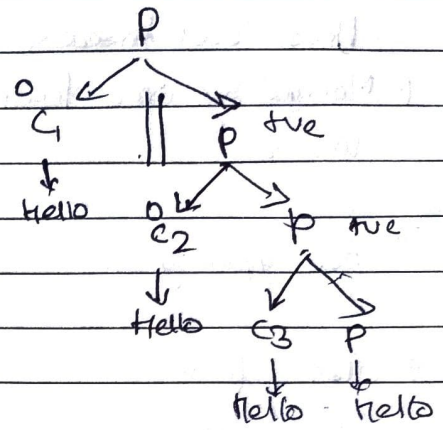
2^3 or (2^n) calls → 7 child + P.

n = no. of ~~child process~~ forks.

No of c. process = $2^n - 1$

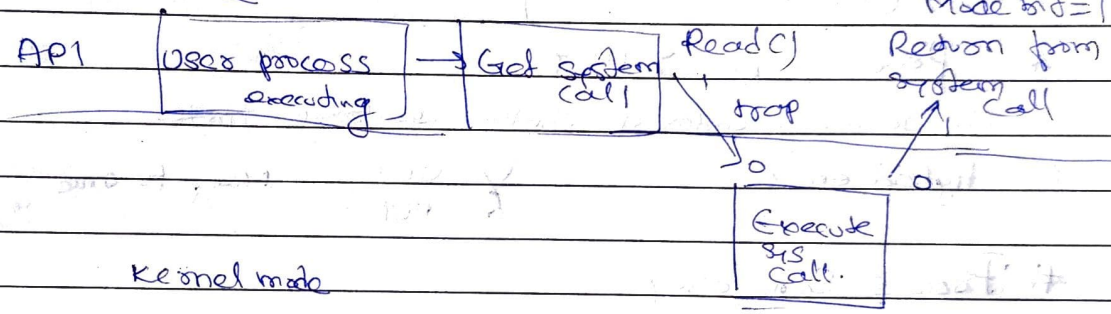
2^{n-1} ← parent

```
main(){
    if (fork() && fork())
        fork();
    printf("Hello")
}
```



User vs kernel mode

↔ Processor switches b/w these 2 modes



* Process vs Threads in OS

	Process	Threads (user level)
1.	System calls involved in Process	1. No system call involved.
2.	OS treats diff process differently	2. User level threads treated as single task for OS
3.	Diff process have diff copies of data, files, code	3. Shares same copy of code and data.
4.	Context switching is slower	4. Context switching is faster
5.	Blocking a process won't block another process	5. Blocking a thread will block entire process
6.	Independent.	6. Interdependent

Stack
Registers
Code
Data

Date: 12/07/12

KLT creation is almost similar to process creation.

AMAS | Page No. | Date

Both share code & Data section

User level threads Vs Kernel level threads

User level threads	Kernel level threads
1. Managed by user level libraries	1. Managed directly by OS.
2. Are typically fast	2. Slower than user level
3. Less faster	3. Context switching is slower
4. If one ULT perform blocking than entire process gets blocked	4. Blocking of 1 KLT doesn't affects other.

* Process > KLT > ULT

In decreasing order of context switching time

Hybrid env: ULT, KLT — Many to one

Process Scheduling

Selecting a process from ready queue and keeping it in run state. based on criterias

- Preemptive: SRTF, RR, LRTF, Priority
- Non-preemptive: FCFS, SJF, LJF, HRRN, Multilevel queue, ~~Priority~~

Preemption ⇒ improves responsiveness

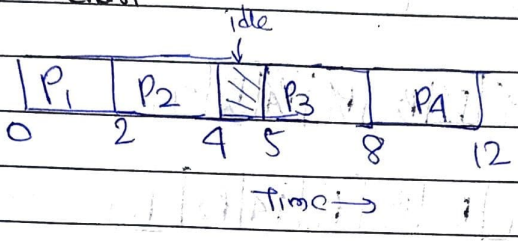
1. Arrival Time: Pt. of time at which a process enters ready queue.
2. Burst time: Time req. to get executed. (duration)
3. Completion time: Process has completed its execution
4. Turn around time: Completion time − Arrival time
5. Waiting time: TAT − Burst time
   or Burst + waiting time = TAT (total time) — useful
6. Response time: Pt of time at which process got CPU for first time − (Arrival time)

AMAS | Page No.
Date / /

FCFS (Criteria: AT, Mode: Non-preemptive)

P No.	AT	BT	CT	TAT	WT	RT
P1	0	2	2	2	0	0
P2	1	2	4	3	1	1
P3	5	3	8	3	0	0
P4	6	4	12	6	2	2

Gantt chart



for non-preemptive WT=RT

SJF (Criteria: BT, Mode: Non-preemptive)

PNo	AT	BT	CT	TAT	WT	RT
P1	1	3	6	5	2	2
P2	2	4	10	8	4	4
P3	1	2	3	2	0	0
P4	4	4	14	10	6	6

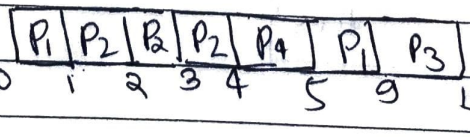
G.C.

SRTF (Criteria: BT, Mode: Preemptive)

PNo	AT	BT	CT	TAT	WT	RT
P1	0	9	9	9	4	0
P2	1	4	4	3	0	0
P3	2	4	13	11	7	7
P4	4	1	5	1	0	0

RQ
P1 P2 P1 P4 P3

G.C



Round Robin (criteria: Time quantum, Mode: Preemptive)

⇒ Sequence of process in ready queue

TQ=2

P.No	AT	BT	CT	TAT	WT	RT
P_1	0	~~5~~ 3, 1	12	12	7	0
P_2	1	~~4~~ 2	11	10	6	1
P_3	2	~~2~~ 0	6	4	2	2
P_4	4	~~1~~	9	5	4	4

G.C Ready Queue | ~~P_1~~ | ~~P_2~~ | ~~P_3~~ | P_1 | P_4 | ~~P_2~~ | P_1 |

G/C Run Queue | P_1 | P_2 | P_3 | P_1 | P_4 | P_2 | P_1 |

0 2 4 6 8 9 11 12

No. of context switches = 6

Priority Scheduling (criteria: Priority, Mode: Preemptive)

$P_1 \geq P_2$

Priority	P.No	AT	BT	CT	TAT	WT	RT
10	P_1	0	~~5~~ 4	12	12	7	
20	P_2	1	~~4~~ 3	8	7	3	
30	P_3	2	2	4	2	0	
40	P_4	4	1	5	1	0	

P_1
$P_1 P_2$
$P_1 P_2 P_3$
$P_1 P_2$ ~~P_3~~ P_4

P_1	P_2	P_3	P_4	P_2	P_1

0 1 2 4 5 8 12

Date: 13/07/21

Multilevel Queue Scheduling

Highest Priority — System Process (interrupts) — RR

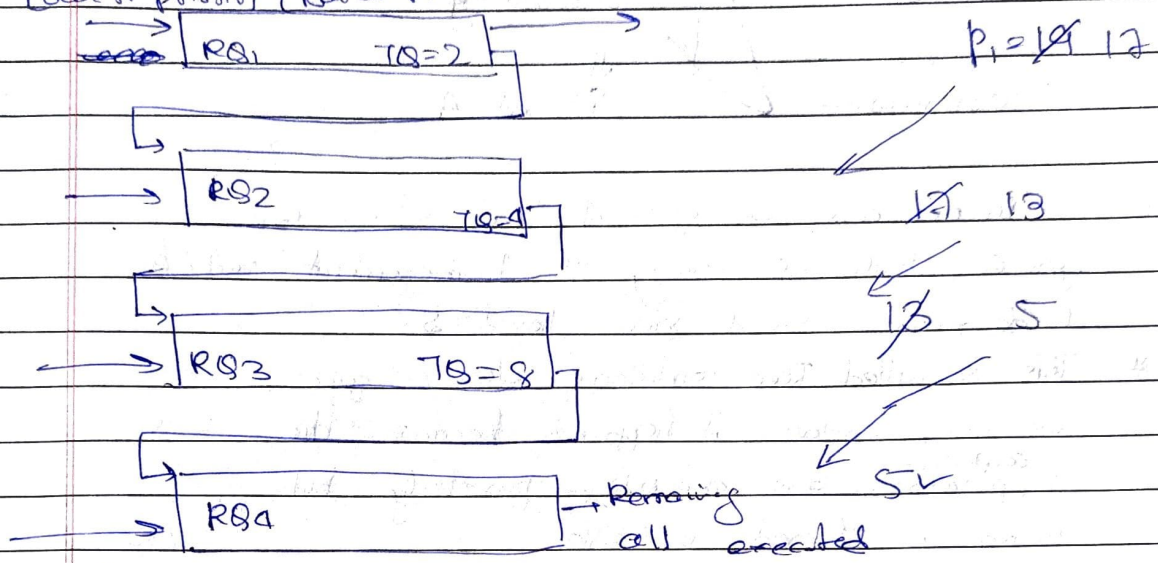
Medium — Interactive Process (Applications) — SJF

Batch Process (Background processes) — FCFS

→ CPU

Multilevel feedback queue

Because lower queues may face starvation if more and more processes are entering higher queues.



Process Synchronization (creates problem in parallel execution)

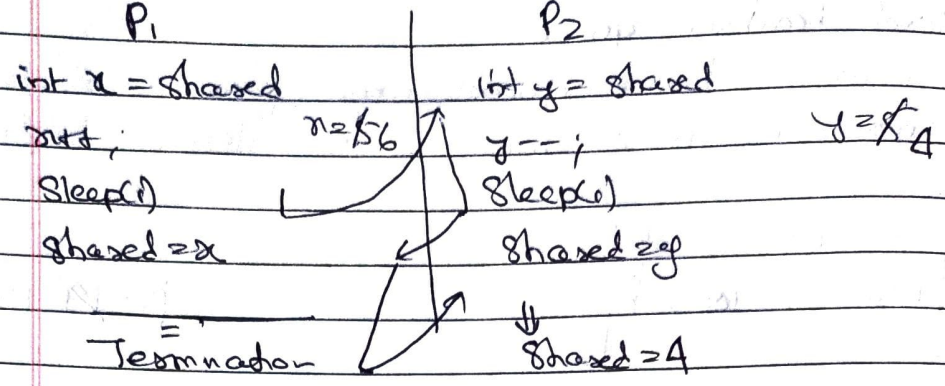
(i) Cooperative process
They share → Variable
Memory
Code
Resources → CPU, Pointer, Scanner

→ Execution of 1 process effects other process's execution.

(ii) Independent Process
No reliability on each other

sleep() for implementing preemption

Int shared = 4



In one case we got 4 and other we got 6, both are wrong, as 1 increment and 1 decrement should give shared = 5.

This is called Race condition, always gives us wrong answers, it happens because the 2 cooperative processes are executing parallely but using a shared variable.

Producer - Consumer Problem

```
int count = 0;          (shared variable)
void producer() {
    int itemp;
    while(true) {
        produceritem(itemp);
        while (count == n);
        Buffer[in] = itemp;
        in = (in+1) mod n;
        count = count + 1;
    }
}
```

Buffer full ←

```
1. load Rp, m[count]
2. INCR Rp
3. Store m[count], Rp
```

n = 8

Buffer [0..n-1]

0 1 2 3 4 5 6 7

count = 0 1

in = 0 1

out =

Case 1 Simple.

P_1

```
Void Consumer () {
    int itemC;
    while (true) {
        while (count==0) ;   // consume Buffer empty
        itemC = Buffer (out);
        out = (out+1) mod n
        count = count - 1;
        Process_item (itemC);
    }
}
```

load
Decr
Store

out = ϕ1
count = 0
itemC = x_1

Case II:

Suppose P2 gets preempt after instruction 2: while producing x_4

0	~~x_1~~	count = ~~4~~
1	x_2	in = ~~4~~
2	x_3	
3	x_4	count = 3

→ Consumer process starts

itemC x_1 out = ϕ1

count = 3

Producer I_1 I_2 (count=3) Consumer I_1 I_2 Producer I_3 (T) I_3 (count=4) (R=2) (count=2 (but it should have been 3))

⇒ Race condition arises giving us wrong value of count

Date: 14/07/21

Printer spooler problem.

Spooler Directory

0	f1.doc
1	f2.doc
2	f3.doc
3	~~f4.doc~~ f5.doc
4	

I1: Load R_i, m[in]
I2. Store SD[R_i], "File Nm"
I3 INCR R_i
I4 Store m[in], R_i

e.g P_1 f4.doc
R_1 0 1
IN 0 1

Suppose 2 processes are there (cooperative)
P_1 → f4.doc, P_2 → f5.doc

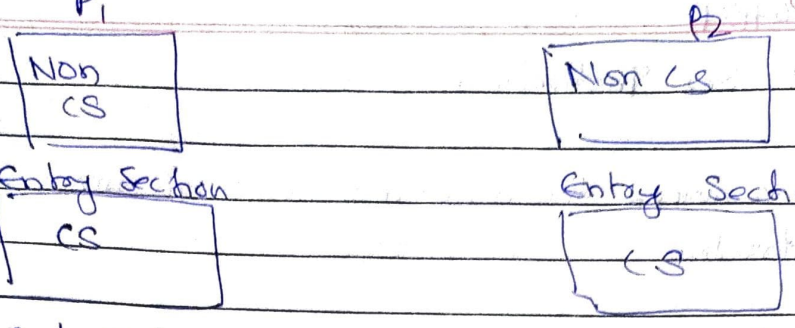
In 3 ~~4~~ 4
R_1 3 4
R_2 3 4

let P_1 comes first:
$P_1 I_1 I_2 I_3$ | $P_2 I_1 I_2 I_3$ | I_4 P_1
Preempt P

Due to asynchronization, loss of data happened in Spooler directory at in=3; data from P_2 overrided the already existing data in SD(3) from P_1, due to Race condition.

Critical Section: It is the part of program where shared resources are accessed by various cooperative processes.
(place where these shared variables/resources are placed)

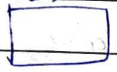
If one process is executing CS and another process tries to execute CS, race condition occurs



Synchronization Mechanism has 4 conditions

(i) Mutual exclusion : At a time, only one process can enter the critical section

Primary conditions

(ii) Progresse 

P_1 wants to enter but P_2 doesn't let it enter, and P_2 itself doesn't execute CS code, or vice-versa then no progress is there. This should be eliminated.

(iii) Bounded wait : There shouldn't be indefinite wait for any process. Starvation shouldn't occur.

(iv) No assumptions related to H/w & speed : Solutions should be independant of H/w, processor, speed, ect. An universal Soln should be there.

Lock Variable in OS

```
do{
    acquire lock
    CS
    release lock
}
```

```
1. while(Lock==1);
2. Lock=1
3. CS
4. Lock=0
```

1, 2 : Entry Code
4 : Release

* Execute in User Mode
* Multiprocess soln
* No mutual exclusion guarantee

Lock=0 → Vacant
1 → Occupied

Lock=1 Now P2 can't enter

Case (1) P1 P2
Case (2) L=0

Preemption

P1 1 * P2 1 2 3 2 3

Two processes inside CS at the same time

Date: 15/07/21

Test and Set Instruction

It has combined the entry codes into an atomic code/instruction

ie

```
while (test_and_set(&lock));   ← entry
CS
lock = false;                  exit
```

```
boolean test_and_set (boolean *target) {
    boolean r = *target;
    *target = true        // set lock to true.
    return r;             // previous value of lock.
}
```

ME ✓ Progress ✓

Turn variable (strict alteration)

→ 2 process soln

→ User mode

turn=0; ⇒ $P_0 \Rightarrow P_1$ If turn=1 ⇒ $P_1 \Rightarrow P_0$

P_0	P_1
NON-CS	Non-CS
Entry while (turn!=0);	while(turn!=1);
CS	CS
Exit turn=1;	turn=0;

ME ✓ Progress ✓

Bounded wait ✓

Semaphore

Prevents race condition.

→ It is an integer variable, which is used in mutual exclusion manner by various concurrent cooperative process in order to achieve synchronization

(i) Counting Semaphore : $(-\infty$ to $+\infty)$

Operations in entry code and exit code

P(), Down, wait ← entry code

V(), Up, signal, Post, Release ← exit

⇒ Entry code

```
Down (Semaphore S) {
    S.value = S.value - 1
    if (S.value < 0) {
        Put process (PCB) in
        Suspended list, sleep();
    }
    else
        return;
}
```

S

[2] 1 0 -1

P1 P2 P3 P4

✓ ✓ ✓ X

S

[-4] ⇒ 4 processes in wait state

Suspended

P1 P2 P3

Entry Section

CS P1 P2 P3

Done

Exit code (up)

wait

P5

⇒ Exit code

```
up (Semaphore S) {
    S.value = S.value + 1;
    if (S.value <= 0) {
        Select a process from
        Suspended list;
        wake up(); // keep in ready
                   queue a process from
                   Suspended list.
    }
}
```

S

[-2] → -1

S = 1

(ii) Binary Semaphore → 0, 1

```
Down (Semaphore S) {
    if (S.value == 1)
        S.value = 0
    else
        Block this process
        and place in Suspend
        list, sleep();
}
```

S = 1 ⇒ 0 (Success)

S = 0 ✗ (unSuccess)

```
up (semaphore S) {
    if (Suspend list is empty)
        S.value = 1
    else
        Select a process from
        Suspended list,
        and wake up()
}
```

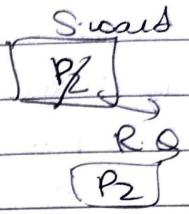
if S is empty → 1

S = 1 + (wake a process)

S = 1 (if 0 no process enters CS)

P_1	P_2
PI_1 Down(S);	QI_1 Down(S);
PI_2 CS	QI_2 CS
PI_3 UP(S)	QI_3 UP(S)

S = 0 ;



Example:

Solution of Producer - Consumer Problem:

Counting Semaphore → full = 0 (Initially)
empty = N

Binary Semaphore → S = 1

```
Produce Item (item p):
  1. down(empty);
  2. down(S)
     Buffer[in] = itemp;
     in = (in+1) mod n;
  3. up(S)
  4. up(full);

Consumer Item ()
  1. down (full)
  2. down (S)
     itemc = Buffer[out];
     out = (out+1) mod n;
  3. up(S)
  4. up(empty);
```

N=8

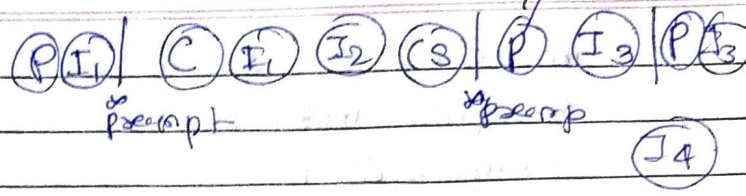
0	a
1	be
2	c
3	d
4	
5	
6	
7	

2

empty = 5 4 3

full = 3 2 1

S = 1 0 1



The process that executes down(S) first enters critical section and the other can't.

Date: 15/07/21

Reader-writer Problem : Two types of users in a single database

R-W , R-R , W-W ⇒ problem R-R → No prob.

rC → no. of readers in Buffer

Case I : R-W problem ; when the first Reader enters CS; No writer can enter

Case II : W-R : when a writer enters CS (DB), no one can enter

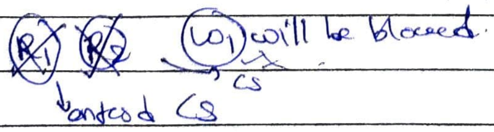
```
int rc=0;
Semaphore mutex=1;
Semaphore db=1;

Void Reader() {
    While (true) {
        1. down (mutex);
        2. rc=rc+1
        3. if(rc==1) down (db);     // only when first reader arrives
        4. up(mutex);
        [CS / DB]
        5. down (mutex);
        6. rc=rc-1;
        7. if(rc==0) up (db);       // the only reader comes out of CS
        8. up(mutex);
        Process_data
    }
}

void Writer() {
    while (true) {
        down (db);
        [CS / DB]
        up(db);
    }
}
```

(entry: lines 1–4; exit: lines 5–8)

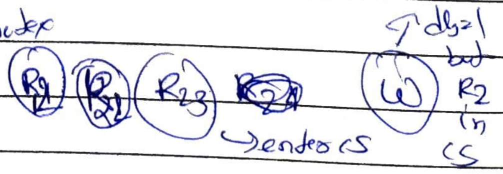
* This part has to be covered by 1st reader then only 2nd reader is allowed

e.g. mutex [1] 0, 1, 0, 1, 0, 1, 0, 1 db [1] 0, 1, 0, 1 rc [0] 1, 2, 1, 0

R1 R2 W1



Suppose a Reader comes w/o mutex



rc: 0, 1, 2

db: 1, 0 — W and R2 both inside CS

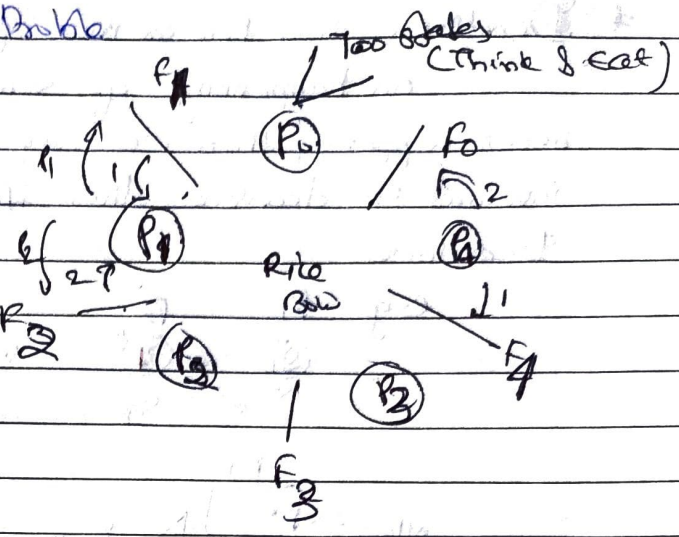
db=1 but R2 in CS

~~race condition~~

No M/E

Date: 16/07/24

Dining Philosopher's Problem



Case I:
P_0 comes to f_1 eat for f_1
P_1 comes to f_2 (eat) f_1 f_2

Case II:
P_0 comes to f_0 | P_1 comes to f_1 f_2

Develops a race condition

We will use Array of Binary Semaphores

#Philosophers = Size of array

S_0 S_1 S_2 S_3 S_4

Case III:
P_0 upto wait(S_0) | P_2 upto (S_1)

It leads to deadlock situation

```
void Philosopher(){
  while(true){
    Thinking();
Entry{ wait(take_fork(Si));
    wait(take_fork(Si+1)%N);
    EAT();
Exit{ signal(Put_fork(i));
    Signal(Put_fork(i+1)%N);
  }
}  // N philosophers
```

and

```
wait(take_fork((Si+1)%N));
wait(take_fork(Si));
```

Remove Deadlock

(1) Remove one philosopher

(2) Out of 5
4 take S_i, S_{i+1}
one faces S_{i+1}, S_i (then no deadlock will be there)

P_0 (S_0) S_1
P_1 (S_1) S_2
P_2 (S_2) S_3
P_3 (S_3) S_4
P_4 S_0 S_4

AMAS / Page No.
Date / /

Dining Philosopher's problem
classic synchronization problem
w/o using semaphores.
→ Race conditions will develop as philosophers willing to eat and dependent, will compete to pick up the forks.

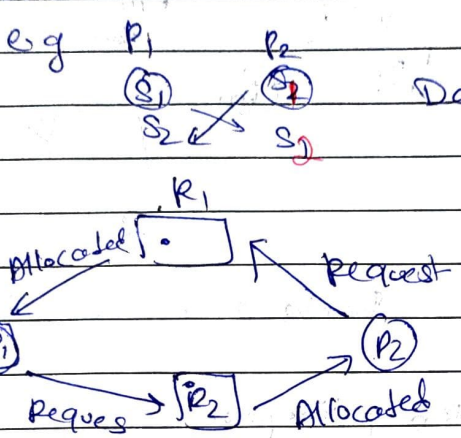
After using an array of Semaphore (representing each philosopher) it is ensured that after one fork is taken, it is not available for others due to wait() operation (1→0)

Difficulty:
The above soln makes sure that no two neighbour philosophers can eat simultaneously.
But in case all the phil. pick up 1 forks. It will lead to a deadlock situation.

```
Void philosopher(){
  Thinking;
  wait(chops[i]);
  wait(chops[(i+1)%5]);
  Eat;
  Signal(chops[i]);
  Signal(chops[(i+1)%5]);
  Thinking
}
```

This can be overcomed by following methods:
(i) Allow only four philosophers on the table and 5 forks
(ii) 5 forks and 5 philosophers
(iii) Odd no. philosopher will pick up left chopstick first and then right. while even no. philosopher pick right followed by left.
(iv) 4 philosophers will follow one order, eg. left → right and the 5th philosopher does reverse right → left.
(v) A philosopher will only pick chopsticks if both available.

* Deadlock : If two or more processes are waiting on happening of some event, which never happens, then we say these processes are involved in deadlock then that state is called Deadlock.

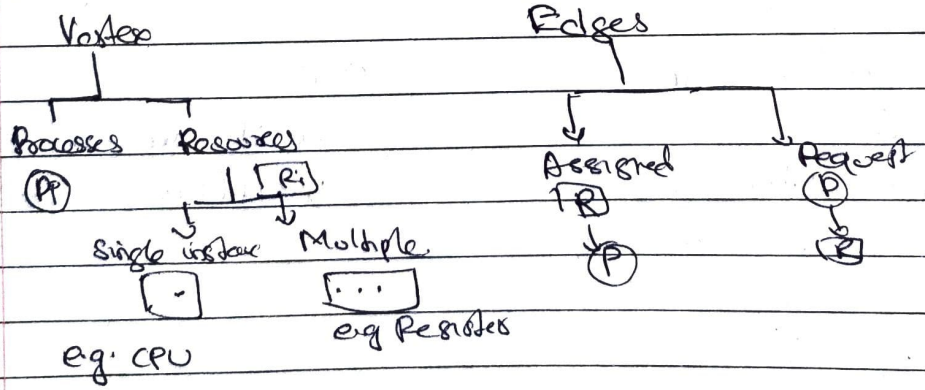


Conditions of Deadlock: (Necessary)

1. Mutual exclusion (Resources must be used one by one)
2. No preemption (Forcefully removal of resource not allowed)
3. Hold and wait (A process must hold & and wait for)
4. Circular wait (in Resource allocation graph)

R → P (Allocated)
P → R (Request)

* Resource Allocation Graph



* Finite waiting and progress ⇒ Starvation
Infinite waiting and no progress ⇒ Deadlock.

* If RAG has circular wait and every resource is of single instance ⇒ Deadlock
Vice - versa ⇒ No deadlock.

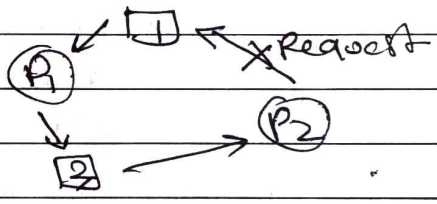
AMAS | Page No.
Date / /

SI && CW ⇒ True
SI && NCW ⇒ False

Various methods to resolve deadlocks.
(1) Deadlock ignorance (ostrich method) [Restart]
(2) Deadlock prevention
(3) Deadlock Avoidance [Banker's algo]
(4) Deadlock detection & Recovery.

(2) Try to remove any one of the four necessary conditions of deadlock. (ME, No preempt ....)
→ No Mutual exclusion (allow multiple processes to enter a particular resource)
→ Preemption
→ !(Hold & wait) = Allocate all req resources initially to a process so that it don't has to wait for any.
→ Removing Circular wait:
Number the resources

1	Printer
2	Scanner
3	CPU
4	Register



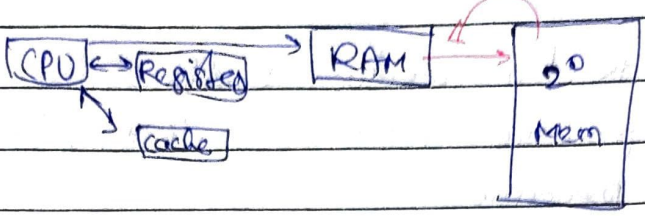
→ A process can only request in increasing order.
i.e. To request a resource R_j, a process must release all R_i s.t. $i \geq j$ for R_1, P_2 must release R_3

Diamond nig problem

Date: 11/08/21

Memory Management

Goal: Efficient utilization of memory.



Increasing size of RAM, Registers, cache will increase cost of system. Hence, they have limited size. And thus we need to manage.

Degree of

Multiprogramming: Multiple or as many as possible programs should be brought into RAM from 2nd mem. for better utilization of CPU.

K → I/O time

(1-K) → CPU time

And n = no. of process

Total I/O time = K^n

CPU utilization = $(1 - K^n)$

the more the value of n, the more is CPU utilization as $0 < K < 1$.

For this we need to increase the size of RAM.

Mem. Management Techniques

(1) Contiguous → Fixed partitioning (Static)
→ Variable Partitioning (Dynamic)

(2) Non-contiguous → Paging
→ Multilevel paging
→ Inverted paging
→ Segmentation
→ Segmentation paging

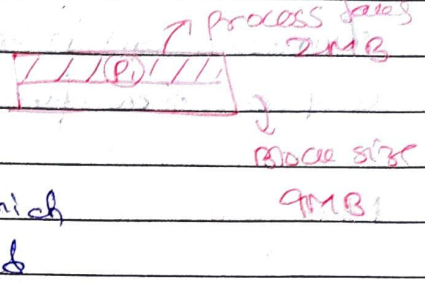
AMAS | Page No.
Date / /

fixed partition (Static)

→ No. of partition fixed.
→ size of each partitions may or may not be same
→ ∵ Contiguous allocation, spanning is not allowed.
i.e. only those processes whose size <= partition size is allowed in RAM.

Demerits:

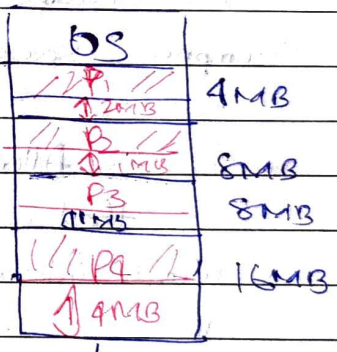
1. # The wasted amount of memory in partitions, which after allocation to a process which can't be used anymore is called internal fragmentation.



2. # There is a limit in Process size (<= max(partition size))

3. # Degree of multiprogramming (= no. of total partitions)

4. # If a new process of size 5MB comes in, although we have 9MB available memory, but this process can't be allocated with this memory, as only one process can reside inside one partition, this is known as external fragmentation.

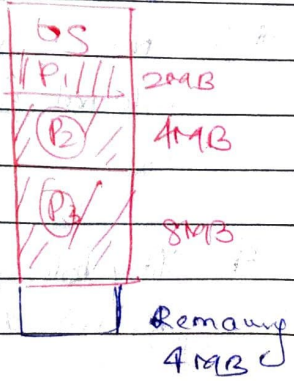


Variable partition (Dynamic): Whenever a process comes in RAM, only then space is allocated to that process based on it's requirement. Partitions are done at runtime.

→ No internal fragmentation
→ Degree of multiprogramming is not compromised
→ No limit in process size

Disadvantages:

→ Suppose if (P2) goes out, a hole of 4MB is created. Now, if a new process P4 comes in of 5MB. Although, we have total 8MB, but we can't allocate it. It suffers from external fragmentation (∵ mem is not contiguous)



We can remove ext. frag using 'Compaction', i.e. shift all allocated space together and rest together. But this has overhead and hence non-desirable.

Allocation and deallocation is a little bit complex than fixed partitioning.

Memory allocation techniques

1. # First-fit: Allocate the first hole that is big enough

2. # Next-fit: Same as first fit, but start search always from last allocated hole.
→ We have a pointer pointing to the last allocated hole.
→ Improves searching time.

3. # Best fit: Allocate the smallest hole that is big enough. Which gives min internal fragmentation.

4. # worst fit: Allocate the largest hole available.

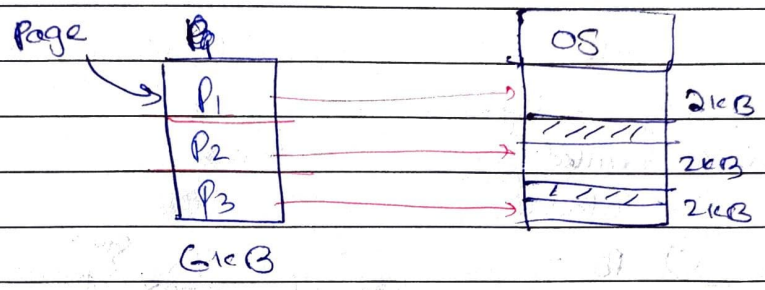
	Advantages	Disadvantages
# 1.	Simple and fast	1. Leads to hole creation
# 2	faster using pointer	2. "
# 3.	No or very low internal fragmentation	3. Because of tiny hole creation, they are non-reusable → Needs to search the entire list
4.	Tiny hole problem is not there.	4. Search time very large

All the above algorithms works for contiguous memory allocations

Non-contiguous memory Allocation

→ processes can span b/w different RAM holes.

→ External fragmentation is removed.



Divide P1 into 3 parts of 2KB each. And allocate each hole to each partition

Holes are changing dynamically and the cost of partitioning a process according to these dynamic holes is high.

Each partition is known as "Page". This partition is done in a memory itself.

RAM memory is divided into "Frames".

Page Size (process) = Frame Size (memory), So that pages fits optimally in frames of the primary

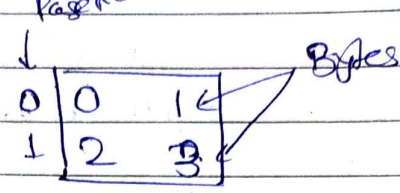
Paging is division of process according to frame size (of main) in 2nd main to reduce external fragmentation.

Page Size = 2B

Process Size = 4B

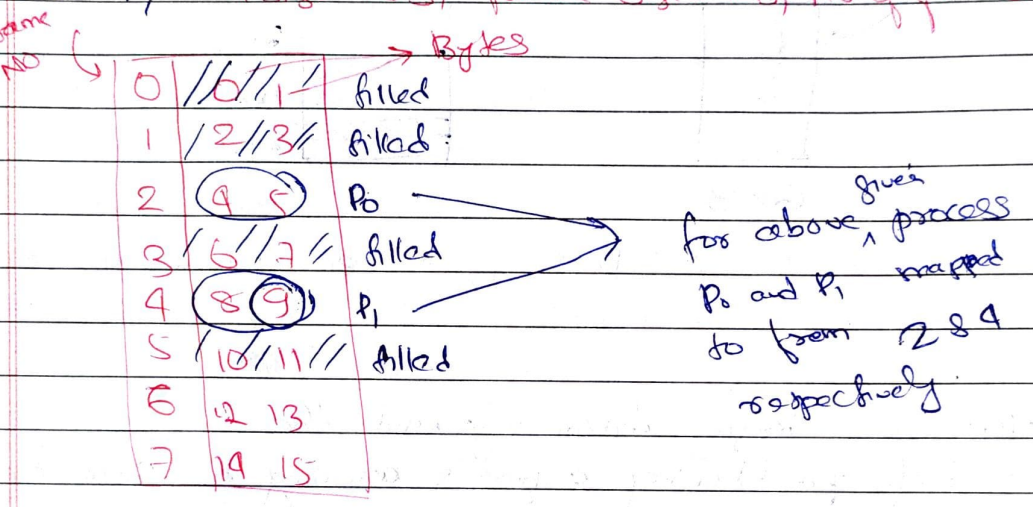
No. of pages = $\frac{4}{2}$ = 2

Representation



This makes our memory "Byte addressable".

In RAM, MSize = 16B, frame Size = 2B, No. of frames = 8



Now, let's CPU ask for 3rd byte from this process
(But 3 is not absolute address)

We need a technique to map

3 indexed → (3) from CPU to (9) (in RAM) ↳ abs address

This is achieved through page table.

→ Each process has page no. 0, 1, 2 . . .

→ So, every process has its own page table.

Page table of P

0	f2
1	f4

↓ frame no, whose corresponding page no. is stored.

AMAS | Page No.
Date / /

CPU always works on logical address

LA | Page No. | Page offset |

Pg No → 1 bit (∵ No of pages = 2)

Page offset → 1 bit (∵ page size = 2b)

So, for 3rd byte.

LA | 1 | 1 |

Converting LA (3) to Abs address (physical) (9)

PA | 100 | 1 |

3 bits → frame no (frame size in bd (2 → 1 bit))

1 bit → frame offset

No. of frames 8 (3 bits) (This comes from page table)

e.g CPU wants 1st byte

1 ⇒ | 0 | 1 | → Pg offset

↓ Pg no

Page table

0	f2
1	f4

PA

0 → 2 ⇒ | 010 | 1 | = 5th byte in RAM.

∴ offset will be same as (frame size = page size)

Ex. LAS = 4GB = $2^2 \times 2^{30}$ B = 2^{32} Bytes ⇒ 32 bytes = LA (size)

Page size = 4KB = $2^2 \times 2^{10}$ B = 2^{12} Bytes ⇒ 12 bytes (for Page offset)

| 20 | 12 |

32

No. of pages = 2^{20}

No of frames = 2^{14}

PAS = 64MB

= $2^6 \times 2^{20}$ B

= 2^{26}

PA

| 14 | 12 |

26

No of entries in page table = No. of pages

Size of page table = (No of pages) × (No. of bits reqd to represent a frame bits) (May)

Page table entries

→ It is used to convert LA to PA.

Page present or not

Enable/disable

based on requirement

Frame No.	Valid(1)/ Invalid(0)	Protection (RWX)	Reference (0/1)	caching	Dirty

Mandatory field

Optional fields

Read write exec

Page reference (0/1)

Modify (if yes (1) else No(0))

Multilevel paging

→ page table is also kept in one of the frames of RAM

→ If size of page table exceeds frame size, we divide into multiple page tables.

e.g PAS = 256MB = $2^8 \times 2^{20} = 2^{28}$ bits

LAS = 4GB

FSize = 4KB = $2^2 \times 2^{10}$

Page table entry = 2B

PA ← 28 →

16	

↔ 12 → frame offset

Page Table

0	fnos
:	
n	

No of frame no. size = 16 bits

Total no. of pages = 2^{20}

LA ← 32 →

20	

→ Page offset

↔ 12 →

bits for No. of pages

$4GB = 2^2 \times 2^{30}$

Page table size = $2^{20} \times 2B$

= 2MB >> 4K (frame size)

∴ Page table is divided so that it can fit under frames

Here, No. of partitions in page table = $\frac{2MB}{4KB} = \frac{2^{21}}{2^{12}} = 2^9 = 512$.

No. of entries in outer table = $2^9 \times 2B = 1KB < 4KB$

Outer table → inner table → frame no.

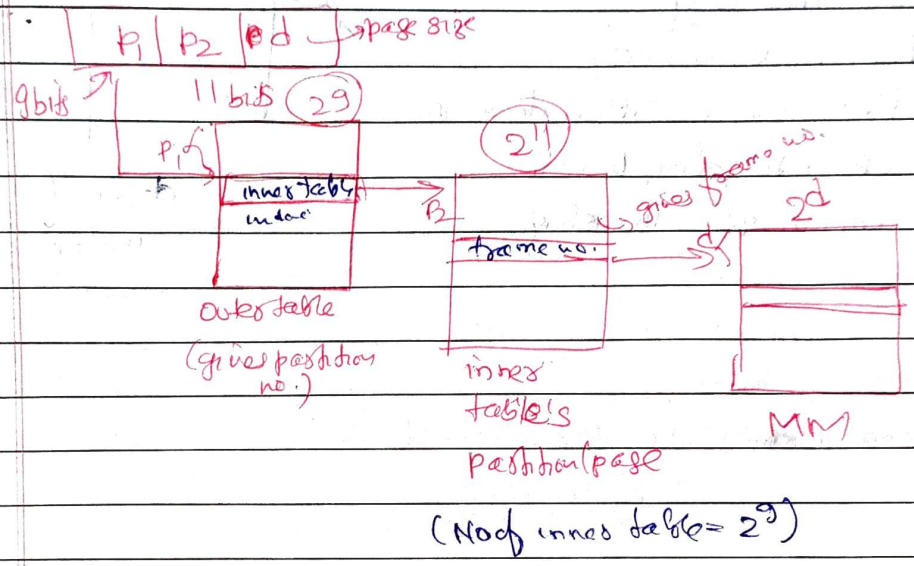
LA

20	12

Each partition of inner table contains $\frac{4KB}{2B} = 2^{11}$ records

Address translational scheme

logical address



(No of inner table = 2^9)

Inverted paging

→ for each process in MM, page table is also stored in MM, even if one page is required by CPU.

→ The above problem is solved using inverted paging.

→ This creates a Global page table.

Structure

frame No.	Page No.	Page ID
0	P0	P_1
1	P_1	$P_{2^{11}}$
2	P0	P_2
⋮	P_2	P_3
1024	.	

if No of frames in MM is =

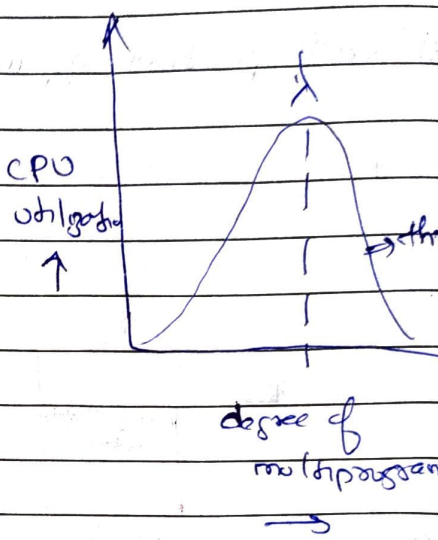
→ High searching time.

AMAS | Page No.
Date / /

Thrashing

→ To increase the degree of multiprogramming one of page from each process is bought.

→ If for each/most of the processes page faults occurs and the CPU get busy in Page fault service, there is a sudden decrease in CPU utilization, which is known as "Thrashing"



Thrashing can be removed by
1) MM size increase
2) Long term scheduler speed should be decreased.

Segmentation Vs Paging

→ In paging, we divide the process without knowing the actual content of it.

→ The problem is that if a fundamental operations code has been divided into two or more pages and either of them is not available in MM (page fault) operation would be executed fully.

→ Segmentation helps in this regard by smartly dividing the process. So that fundamental operation or code related to a single program/function is kept as single segment.

Hence, segments are of variable lengths.

Mem Manag. Unit helps to convert LA to PA.

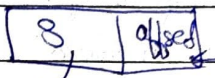
Segment Table

Seg No →

	Base Address	Size
0		
1		
2		
3	1500	300

1500	OS
1800	S3
	S2

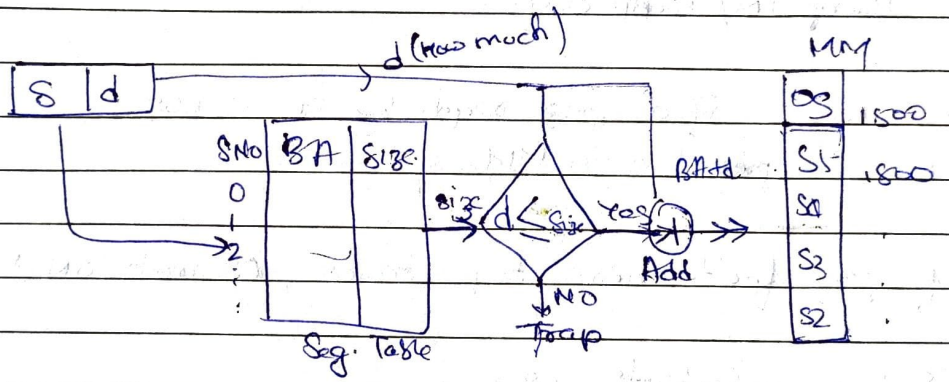
LA



Segment size

Segment No. (How much too travel to access)

Schemas



Advantage:

→ Appropriate from user point of view (related data together)

Overlay : We can accomodate processes having size even greater than MM.

→ We use this concept when MM.size() < Process.size().

⇒ Overlays is generally used in embedded systems.

As, there is no drives in OS which automates it. User itself has to decide which functions are to be kept together which are not.

But embedded systems performs specific tasks and hence overlays can be employed.

Further, the result/area of divided parts of the process should be independent.

AMAS | Page No.
Date / /

Virtual Memory

→ It provides an illusion to the user that a process having size > than MM size can also be executed.

→ It also provides illusion for bringing more and more processes into MM, although it's size is limited

#

We only bring a part or some of the pages into MM instead of entire process into the MM. Whenever a new page is needed, it is swapped with some already existing (non required) page (if no space available) in RAM.
Swap IN / Swap OUT.

Page fault : If a page read by CPU is not present in the MM.

→ If page fault occurs, trap occurs, OS mode ON (takes control)

→ OS first authenticate user
→ Searches for the page from LAS
Brings that page into MM and PMT is updated in page table

Then give control back to user

Effective Mem. access time
'p' prob that page fault occurs

EMAT = p(Page fault service time) + (1-p)(MMAT)
(in msec) ↳ access.
+(MMAT) (in nseconds)

Date: 12/08
Translatio
CPU
→ Page table is
→ And if m
is requ
for increas
(faster the
→ We will
EMAT =
(we can't
limited
TLB st
EMAT =
CPU

Date: 12/08/21

Translational Lookaside Buffer

CPU → (PT) → (MM) Time = 2x
(PT: x, MM: x)

→ Page table is also stored in RAM, so this also consumes time
→ And if multilevel paging is there, even more time is reqd.

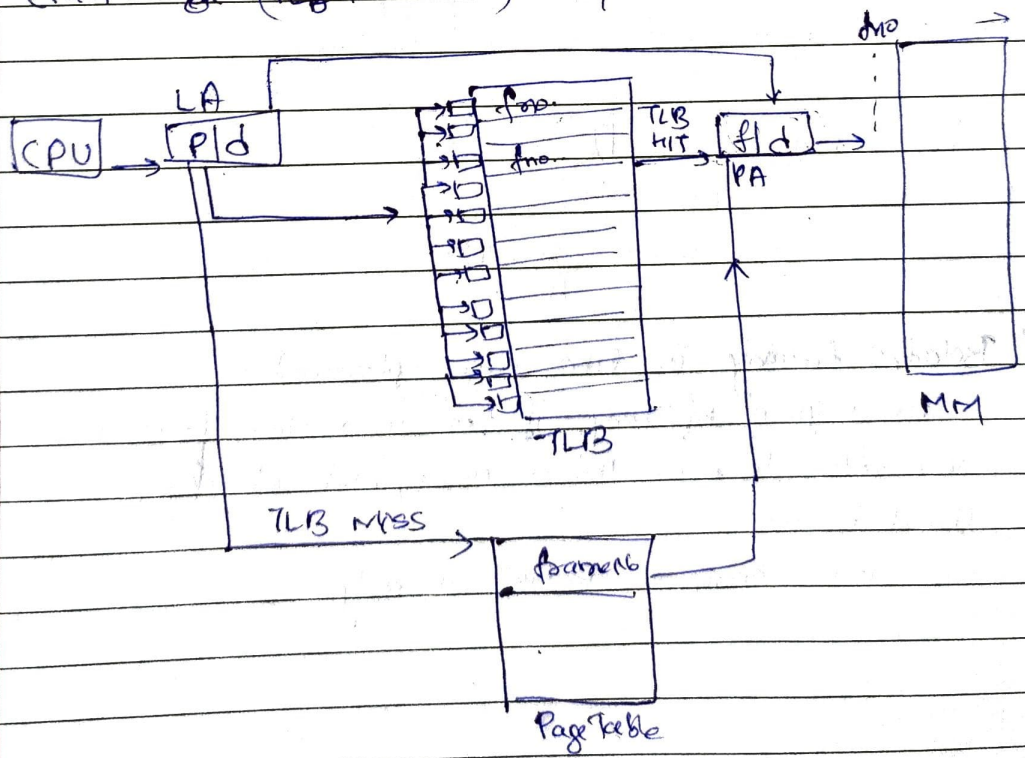
for increasing this speed, we use cache memory or TLB (faster than RAM).

→ We will store page table entries in TLB.

EMAT = (TLB + x) , if TLB hit
(we can't store all page entries in TLB, ∵ memory is limited).

TLB stores frame No. corresponding to a given page No.

EMAT = (TLB + x + x) , miss



AMAS | Page No.
Date / /

Page Replacement algorithms :
→ To swap in / swap pages from RAM and Sec Mem

3 algorithms are:
(1) FIFO
(2) Optimal page Replacement
(3) Least Recently used

(1) FIFO

e.g.
Reference String → 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 1, 2, 0

Time →

F3			1	1	1	1	0	0	0	3	3	3	3	2	2
F2		0	0	0	0	3	3	3	2	2	2	2	1	1	1
F1	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0
	*	*	*	*	H	*	H	*	*	*	*	H	*	*	H

For, page no 2; 7 will be replaced with 2
∵ It came first

No. of page faults = 12
Hits = 3

Hit ratio = $\frac{3}{15} = 1:5 = 20\%$

Belady's Anamoly in FIFO page replacement.
For a given ref string, if we increase frame nos. and still page faults do not decrease, while normally it should decrease.

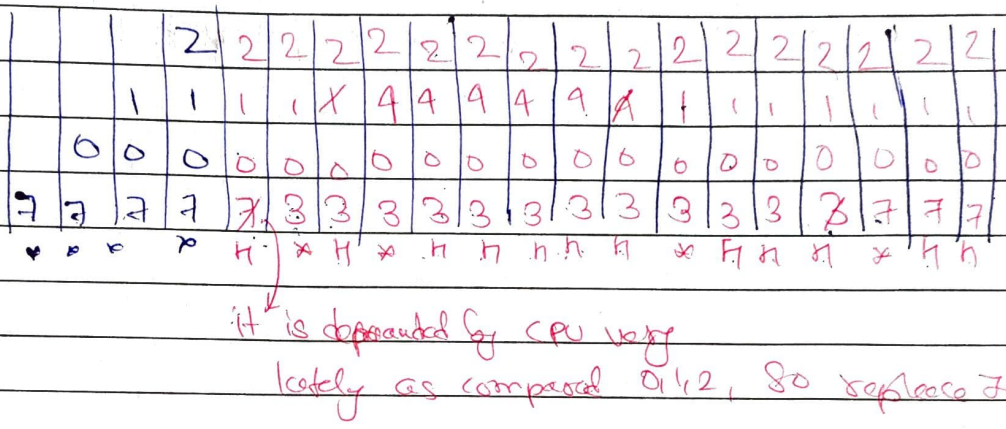
This is known as Belladi's anamoly.

Optimal page Replacement

Replace the page which is not used in longest dimension of time in future. Farthest from current demand's

Ref. string:

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



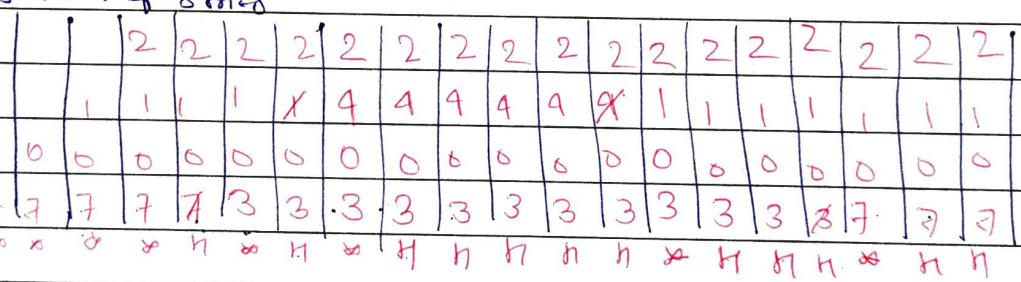
it is deprecated by CPU very lately as compared 0 & 2, so replace 7

Hits = 12

faults = 8

Least Recently used : Remove/swap the least recently used page in past.

Same Ref string



Hits = 12

Faults = 8

→ Speed slower than FIFO, as it searches in past

AMAS | Page No.
Date / /

Most Recently used

Remove the most recently used pages in past.

7, 6, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

f4			2	2	2	2	2	2	3	0	3	2	2	2	0	0	0	0	0
f3		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f2	0	0	0	0	3	0	4	4	4	4	4	4	4	4	4	4	4	4	4
f1	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
	x		x	x	x	H	x	x	x	H	M	x	x	x	F	H	M	F	H	H	H

0 3

hits = 8

faults = 12

(2)

Atomic access

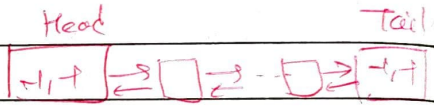
Mutex Vs Semaphore

Both of them are Synchronization primitives often Mutexes are confused for binary semaphores, as their implementation is similar; but they are not.

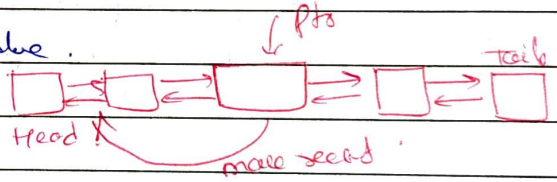
A Mutex is a "locking mechanism" used to synchronize access to a resource. Only one task (can be thread/process) can acquire this lock, and only the owner can release the lock later on.

While semaphore is a "signalling mechanism" ("I am done, you can carry on", kind of signals)

LRU implementation



get(key)
1. find if not in map, return -1
2. if there, find ptr corresponding to key from map
3. moveRecent(ptr).
4. return ptr->value.



put(key, value){
1. find key in map, if found update value and moveRecent
2. If not found (i) create node/dc
(ii) Add to map
(iii) Add to starting of dll
→ if mapsize exceeds capacity
(iv) Remove tail->prev->key from map (LRU)
and tail->prev from dll using Remove)

Unlikely
(1) Recent(ptr)
(2) add(ptr)
(3) Remove(ptr)