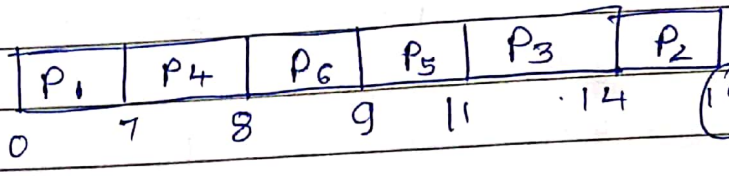
~~SRTF:~~ Make sure, don't keep idle.

Corrected:

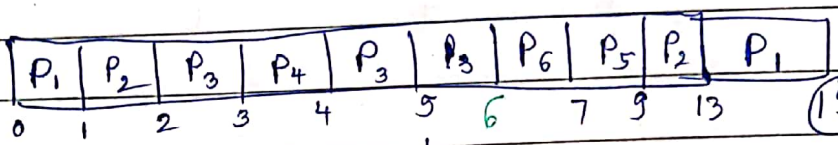
SJF



Same

SRTF:

(Preem



Don't do more context switch if it is equal / already there.

T1

4/2

* Priority Scheduling:

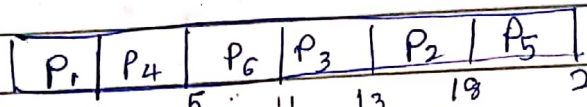
Preemptive (AT + priority.)

Non preemptive.

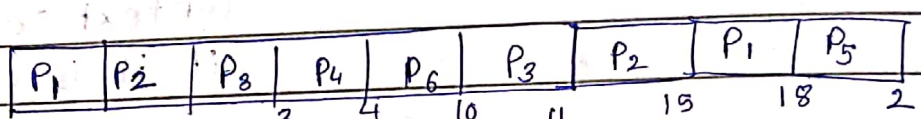
next page

P No.	AT	BT	priority.
1	0	4	4
2	1	5	5
3	2	2	6
4	3	1	8
5	3	3	2
6	4	6	7

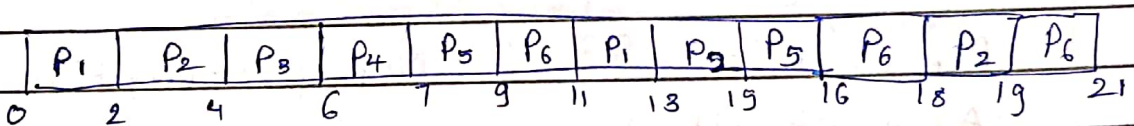
Non Preemp.



Preemp.

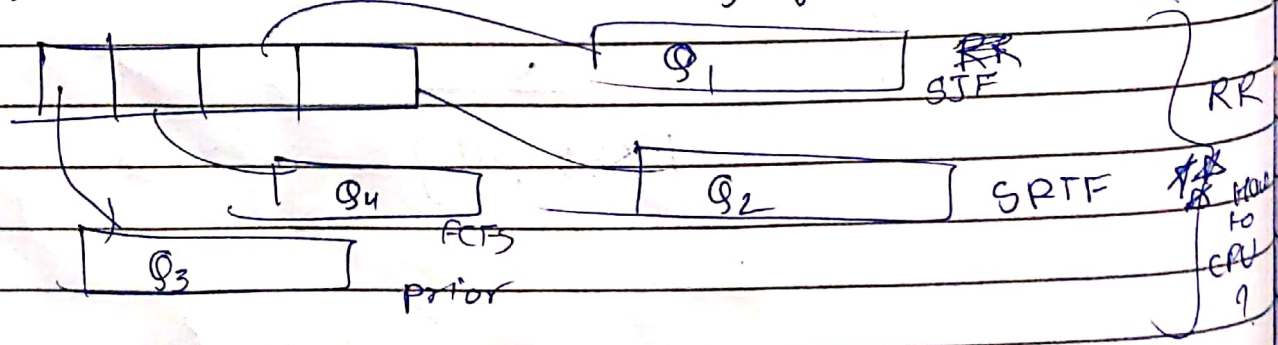


* Round Robin:
AT + Time quantum / time slice.
Same as above



* Multilevel queue scheduling:

Ready queue is partitioned into different queues. Diff. queue for every type of process



Here, every queue is scheduled using different algorithm. But 'starvation' !! lower priority queues have to wait ncl.

Multilevel feed back queue:

Similar, just that, if there is priority considered Q_1 can take more time of CPU. If at all some process in the queue Q_2 is waiting for long, then it can be scheduled in another queue. Time slice per queue is variable.

Aging: High priority is given to a process of lower priority.

Generally

SJF = optimal algorithm.

Bcoz it calculates ~~min~~ avg. waiting time.

min. avg waiting

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

T_{n+1} → Predicted CPU burst.

t_n → n^{th} cpu burst.

T_n → Past CPU Burst.

$$0 \leq \alpha \leq 1.$$

If $\alpha = 0$, $\boxed{T_{n+1} = T_n}$

$\alpha = 1$, $\boxed{T_{n+1} = t_n}$.

Consider a system with SJF with exponential avg. technique. what could be the next CPU burst of proc which has earliest completed burst of 5, 8, 3 & 5. Now for an initial value of $T_1 = 10$, $\alpha = 1/2$.

Calculate

t_i 5 8 3 5

t_2

$T_{n+1} = \alpha t_n + (1-\alpha) T_1$ $T_2 =$

$T_{1+1} = 0.5 * 5 + (1-0.5) * 10$

$T_2 = 2.5 + 5$

$= 7.5$

t_1 3
t_2 5
5
3

$T_3 = (0.5 * 8) + (1-0.5) * 7.5$

$= 4 + 3.75 = 7.75$

$T_4 = (0.5 \times 3) + (1-0.5)(7.75)$

$= 1.5 + 5.375$

$T_5 = \frac{(5 + 5.375)}{2} = 5.1875$.

6/2

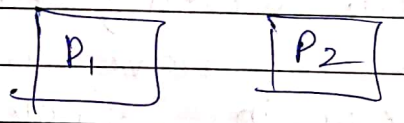
* Threads:
  Single threading
  Multi threading

Suppose for completing job
1 man → 10 days
∴ 10 men ⇒ 1 day.

1 man = $\frac{1}{10}$ th work in one day.

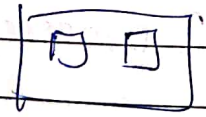
10 men ⇒ work ✓ in 1 day.
ie. parallization.
can increase speed.

multiproce

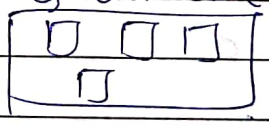


But for cores,
multi proc. in single chip.

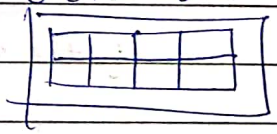
Dual core



Quad core

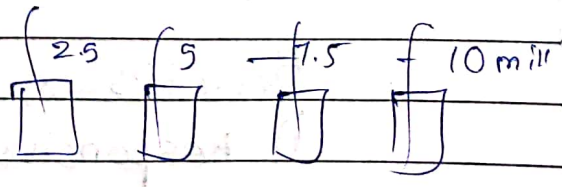


Octa core



Single

14

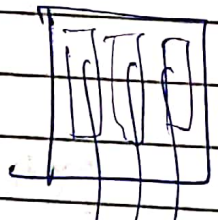


4 processes
or threads
simultaneously.

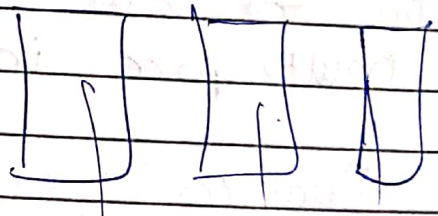
∴ 4 forks.

IPC required.
More system calls. => ∴ more overhead.
More management
Every process has own memory

So... instead of many processes,
there is only single process,
only one fork
same resources alloc. and memory.
Context switch.
Communication easier.
∴ So same process
Thing is that: use a certain library,
while writing program.
Thus, we have concurrency / multiple
thread in single process



3 threads in a process.



3 processes

Separate stack for a separate thread.

* Benefits of multithreading
i) Responsiveness
ii) Economy
iii) Scalability
iv) Resource sharing.

For this, we have PThread library.

1). int pthread_create (pthread_t *thread, ...)
↑ invoke this fuction for creating thread.
4 arguments

i. pthread_t *thread (pointer → create identifier) like p-id !
ii. const pthread_attr_t *attr.
What are the properties assigned to the thread.
iii. void * (*start routine)(void *)
function invoked for thread to start executing.
iv. void *arg :
Arguments for the function.

for exiting thread.

2) void pthread_exit (void *retval)
Status of return value to be given to next thread if any, it will destroy thread.

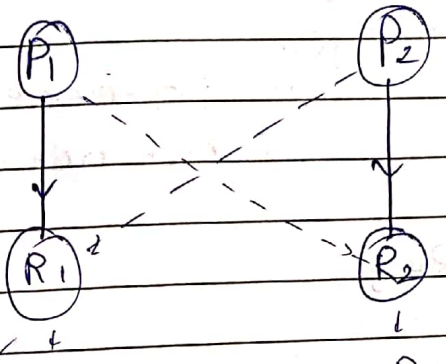
3). join:
int pthread_join (pthread_t thread, void **retval).
t_1 calls t_2, then t_1 waits till t_2 is completed and t_2 returns value to t_1.

```
#include <pthread.h>
#include <stdio.h>

unsigned long sum[4];

void * thread_fn (void * arg) {
long id = (long) arg
int start = id * 2500000
int i = 0
while (i < 2500000)
{
    sum[id] += (i * start)
    i++;
}
return NULL
}

int main() {

pthread_t t1, t2, t3, t4;

Pthread_create (&t1, NULL, thread_fn (void * 0));

Pthread_join (t1, NULL);
Pthread_join (t2, NULL);
  //      (t3, NULL);
  //      (t4, NULL);

print (sum[0] + sum[1] + sum[2] + sum[3]);
```

Basically,
understand what is and how to divide the program.

Priority
Round robin ✗ Not for T-1.

11/2.
* Deadlock:



Oops.. then this is a deadlock condition !!

5 instances of Resource type printers.
for eg:
R = { R1, R2, R3, ... Rn }
P = { P1, P2 ..... Pn }

$R_j \rightarrow W_j$
Director graph : Resource Allocation Graph (RAG).
V : { P, R }
↓ Proc. ↓ Res.

Edges = i) Request edge
ii) Assignment edge

Pi requesting for Rj ∴ Pi → Rj
is a request edge.
Assigned : Rj → Pi.

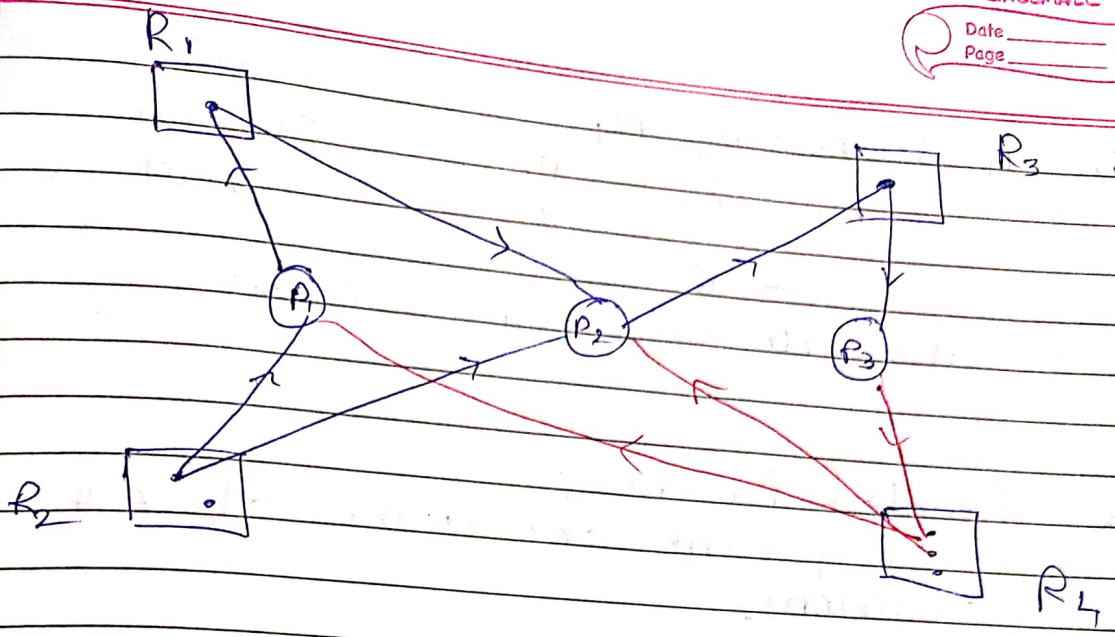
Necessary and sufficient condition for deadlock to occur.
i) single instance → cycle. — neck suff.
ii) ~~Multiple instance~~ → cycle
↳ its necessary but not suff.

P = { P1, P2, P3 }
R = { R1, R2, R3, R4 }.

E = { P1 → R1, P2 → R3, R1 → P2, R2 → P2
R2 → P1, R3 → P3 }.

○ = Processes.
[•] = • = resource instance (number).
□ = resource box.

No. of dots = no. of instances
If 2, ∴ 2 printers are there.



If red...
then we get cycles!!
Cycles...
i) $P_1 \rightarrow R_1 \rightarrow R_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_4 \rightarrow P_1$.
ii) $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_4 \rightarrow P_2$.

* How to avoid/prevent deadlocks?
Necessary conditions for deadlock: ~~prevention~~.

i) Mutual Exclusion:
R allocated to P_i, unless & until P_i completes, it won't release that resource

ii) Holds & wait:
P_i Holding R_1 and P_i is waiting for R_2, if this condition is there then there will be deadlock.

iii) No preemption.
P_i is ziddi, ∴ P_i won't leave R_2, ∴ deadlock will happen.

iv. P_1 is holding R_1 ...

$P_1 \rightarrow R_2 \rightarrow P_2$ Circular Wait

$\leftarrow R_1$

this can also cause deadlock.

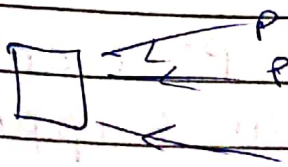
* Deadlock Prevention:
  To prevent one of the above 4 conditions.

* Deadlock Avoidance:
  Mantain system in safe state, allocate in such a way, that proces gets R_i as needed in a proper sequence Execute in proper order or fashion

* Deadlock Recovery:
  "Agar hoga toh dekhenge...."
  Nice 100

* Prevention Methods:

i) Mutual Exclusion:
  Sharable, But writing is a problem, reading is fine
  Data inconsistency
  No way to avoid it.

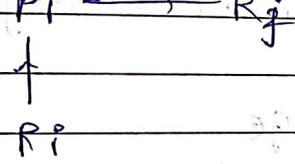


ii) Hold & wait:
It has some problems: (after avoiding)
a) Low resource utilization
b) Starvation.

iii) No preemption:
To avoid it, P_1/P_2 should preempt all its resources

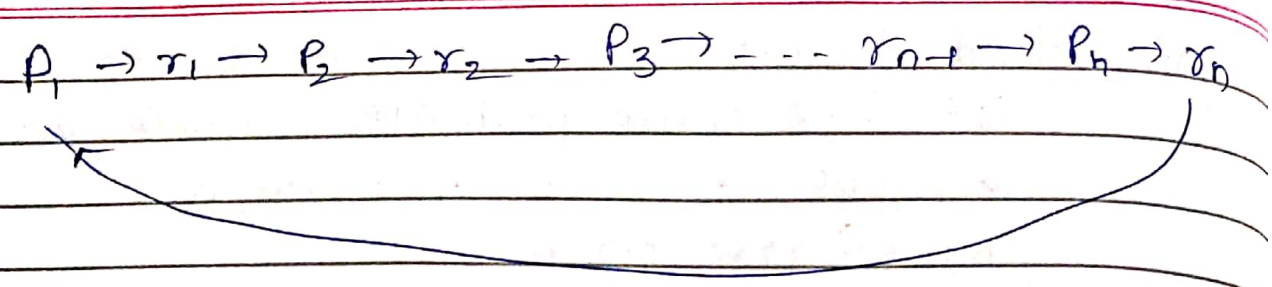
P_1 or P_2 should release the resource depends on how many resources its waiting for. here P_1 will release.

iv) Circular waiting:



whether $F(R_j) > F(R_i)$.
$F(R) \rightarrow N$ (assign a number).

13/2. Every resource is given some numeric value, based on some function
If P_i is requesting to R_j and is allocated R_i already, then R_j will be allocated only if
$F(r_i) < F(r_j)$
But if $F(r_j) < F(r_i)$, then no allocation, so as to avoid deadlock.



$F(r_1) > F(r_n)$
$F(r_2) > F(r_1)$
⋮
$F(r_n) > F(r_{n-1})$

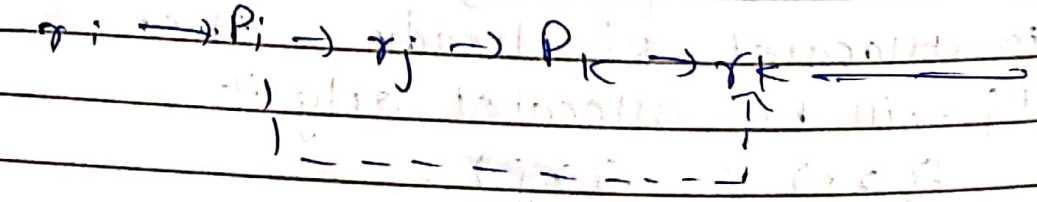
$F(r_1) > F(r_1)$

* (1) Deadlock Avoidance:

i) Safe State.
Single instance = Res. Allocation Graph (RAG)
Multiple instance = Banker's algorithm.

→ = assignment edge
---→ = claim edge
⇠ = request edge.

For single instance, no cycle
claim edge → will become assignment edge only if no cycle.



Thus, we can just avoid deadlock.
Not for multiple instance, bcoz,

cycle is not necessary in that case.

For multiple instance,
Banker's algorithm, is used.

i). Available [j] = k
ii). Need [i, j]
iii). Max [i, j].
iv). Allocation [i, j].

n process, m resources.
resources available array [j] = k
∴ k instances of available [j].

Need [i, j], ⇒ Pi is requesting for Rj.
Max [i, j] ⇒ maximum demand of process Pi which requires max k resources.
Allocation [i, j] ⇒ How many out of k are available and allocated to process Pi.

Max = need + allocation
(kind of...)

algorithm next page →

m n

Work & finish

i. work = available
& finish[i] = false ∀ p : 1, 2, ... n.

ii. Find i such that
finish[i] = false
need <= work.
if no such i, go to (4)

iii. work = work + allocation
finish[i] = true
go to step 2

iv. finish[i] = True

⇒ Safe State.

A B C ⇒ 10, 5, 7 instances

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3 ✓
P_1	2 0 0	3 2 2	5 3 2 *	1 2 2 ✓
P_2	3 0 2	9 0 2	7 4 3	6 0 0 ✓
P_3	2 1 1	2 2 2	~~7 5 3~~ 7 4 5	0 1 1 ✓
P_4	0 0 2	4 3 3	7 5 5	4 3 1 ✓
	7 2 5		10 5 7 ↙	

after completion all resources should come.

Need = Max - allocation

* = work = work + allocation

< P_1, P_3, P_0, P_2, P_4 >

* Resource request algorithm:

$Req_i[j] = k$
if $req_i \leq need_i$
if $req_i \leq available$
available = available - request.
allocation = allocation + request
$need_i$ = $need_i$ - request.

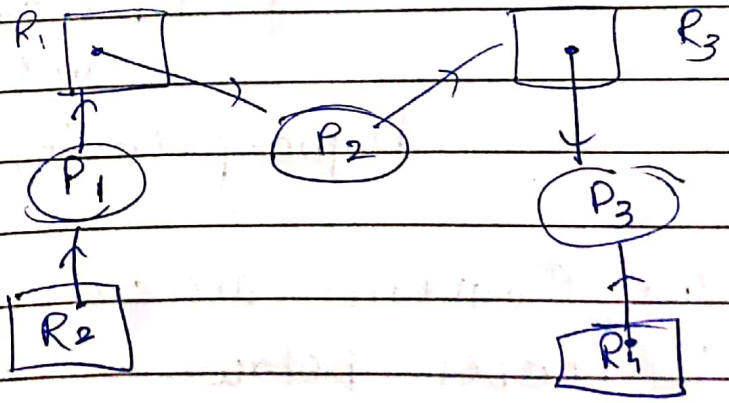
↳ Addition request of P_i for R_j having k instances.

Check on left example.
Same sequence

	Alloc	Max	Avai	Need
P_0	010	753	230	743
P_1	302			
P_2	302			
P_3	211			
P_4	002			

* Recovery:
Here we have to detect a deadlock,

eg:



Check for cycle.
If cycle ⇒ then deadlock
If no cycle ⇒ no problem.

After deadlock detection, we have Deadlock Recovery.

Process Termination
Terminate all processes directly
not recommended
we need to use Banker's again and again.

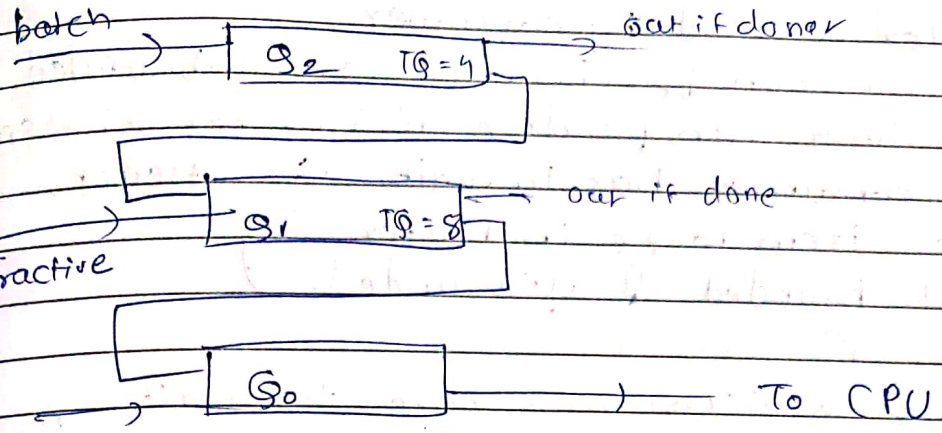
Resource Prempton
Preempt resource from process
i) Victim ?
ii) Rollback ?
iii) Starvation ?

* **Revision**
(Just felt it is important
Multilevel Feedback Queue Scheduling:

In order to overcome 'starvation' of multilevel

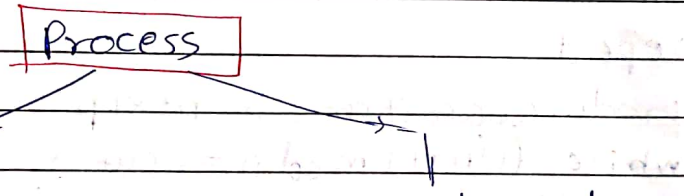
Feedback.. lower priority queue → Time Quantum done → Upgrade → higher priority queue

Time Quantum successively increases below.



16/2

* Process Synchronization:



Co-operating
Producer will produce some item and consumed.
Have shared variables.
eg: compiler (Pro)
↓
(assembly)
↓
(Pro) assembler (Cons)
↓
(object)
↓
(Cons) Loader
Buffer

Buffer is used by consumer.
Buffer can be bounded or unbounded.
Producer can produce 'n' number of items but consumer has to wait for atleast one item to be in buffer.
Check bounded & unbounded.

Prod. & consumer have to wait for one another otherwise producer can't produce.

* Producer

- repeat
  produce an item in nextp
  while ((in+1) mod n == out)
    do nop;

  buffer[in] = nextp
  in = (in + 1 mod n);
- until false
  ↳ (*)

in = next free place for production
out = filled position to take from
(Circular array)

* Consumer.

- repeat
  while (in == out)
    do nop;
  nextc = buffer[out].
  out = (out + 1) mod n
  Consume
- until false

for prod.

(A)

```
while (counter == n)
        do nop;
counter = counter + 1
until false.
```

} to checks if buffer is full or not.

Producer can consumer must synchronize with each other. counter is shared variable, but for that to work, prod & consumer must execute separately.

(B)

```
while (counter == 0)
        do nop;
```

} ← C

```
Counter = Counter - 1
- - -
Consume
until false
}
```

Counter = counter + 1
register = counter
register = register + 1
counter = reg1.

GALVIN ||7

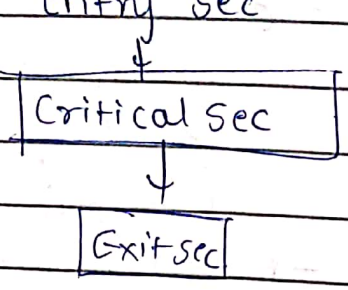
So programs having shared variables must be executed in such a fashion, such that, there is no inconsistency.

Whenever we need to use common code, then we make / call it a critical section.

* Critical Section:
  < many proc exec common code >

  $P_0, \ldots P_{n-1}$

Entry Condition:
  Pi should check, if Crit. sect. is empty, if empty, then enter and block others, otherwise wait.
Only one Pi can exec Crit. sect.



* Necessary Cond:

1. Mutual Exclusion:
   Only one proc in Critical section
   Not more than one.

2. Progress:
   'n' process mei, if 5 are interested, only those 5 are allowed to decide who can enter critical section

3. Bounded wait.
   No forever waiting. ∴ Pi only limited time for each Pi in crit section

2 process Solution:

Algo 1: turn = shared variable, boolean type.

```
repeat
    while turn ≠ i do nop;      → entry
                                  ↳ busy WAIT
        CS;
    turn = j;                   → exit
    remainder sec;
until false
```

turn = boolean,
2 processes = Process i & Process j
(turn = i) & (turn = j).

Same code for process j.
But, it only guarantees Mutual Exclusion, but may not be any progress.

20/2.

Algo 2(a)

P_0	P_1
{	{
while (flag[1]);	while (flag[0]);
flag[0] = T;	flag[1] = T;
CS;	CS
flag[0] = F;	flag[1] = F;
}	

Algo 2(b)

```
P0                              P1
{                               {
  flag[0] = T                     flag[1] = T
  while (flag[1]);  <- busy wait ->  while (flag[0])
  CS                              CS
  flag[0] = F                     flag[1] = F
}                               }
```

In algo 2,
we use a flag instead of single boolean variable.

But, we can't acheive mutual exclusion, so this is not the correct solution (2(a)).
2(b) solves the problem of ME, but, 'progress' is not acheived when both are interested at same time leads to deadlock.

So now,
we have Peterson's algorithm.

It has flag and turn.
flag  F = not interested
T = interested.
turn = boolean to show whose turn.

* Peterson's algorithm:

P0

```
while (1)
{
    flag[0] = T
    turn = 1
    while (turn == 1 && flag[1] == T);
    CS
    flag[0] = F;
}
```

P1

```
while (1)
{   flag[1] = T
    turn = 0
    while (turn == 0 && flag[0] == T);
    CS
    flag[1] = F.
}
```

Mutual Exclusion ✓

Progress

- one Pi can exec. CS multiple ✓
- Context switch properly ✓
- Interested can help to decide. ✓

So claim interest and give other person turn. Hence, no wonder Peterson is a gentleman :P)

<NOT generalizing anything... forget it>.

For multiple processes,
we have Bakery Algorithm.

25/2

Bakery algorithm:
(Simplified)

⇒ lock (i) {
door way !! ↑ should not be non atomic

num[i] = Max (num[0], num[1], ... num[N-1]) + 1

For (P = 0; P < N; ++P)

0 = not interested.

while (num[P] != 0 && num[P] < num[i]);

Entry ↑ ↑ least non zero has priority

↑ wait if you don't have least non zero value.

}

CS

~~write~~

ME ✓

unlock (i) {

num[i] = 0; → after that set 0, to show that you are not interested

}.

Num = array of int, num[i] has index value of ith process.
num[i] = 0 => Pi not interested
Num[i] = non zero priority token.
So I, num[i] allocation should be Atomic
ONLY one Pi should get the token.
Or else no Mutual exclusion !!

Original Bakery algo

```
lock(i)
    choosing[i] = T. *
    num[i] = Max(num[0], num[1] ... num[N-1]) + 1.
    choosing[i] = F; *
    for (p = 0; p < n; ++p) {
        while (choosing[p]);  *
        while (num[p] != 0 &&
               (num[p], p) < (num[i], i));
    }
CS
unlock(i)
{
    num[i] = 0;
}
```

choosing = T will be set if it can choose to update num value.

Now it is fine if not atomic.

to ensure process is not at doorway.

favour process with smaller i value.

$(a,b) < (c,d)$ ⇓ $(a < c)$ or $(a == c \ \&\& \ b < d)$

choosing[i] = boolean array. denotes if P[i] has choosing power or not. ⇓ to update num. or not.

~~If~~ 'choosing' is not considered

choosing | | | | | |

num | 1 | 2 | 2 | 3 | 0 |

~~can't~~ acheive mutual if not atomic.

For progress:

i) Only non zero nums. wale are considered : only interested can ~~ex~~ decide in finite time

ii) No ~~mutual exclu~~ problem if one proc. wants to execute multiple times.

iii) Context switching happening fine ✓.

26/2

* Software Solution.

P1

```
while(1)
{
    while( lock != 0);
    lock = 1;
    CS
    lock = 0
}
```

lock = 0 → means Critical sec is free

lock = 1 ⇒ P1 occupied Critical sec

If at this pt., Context switched to P2, then it can also enter Critical section

∴ No Mutual Exclusion !

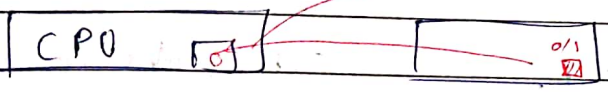
P2

```
while (1)
{
    while(lock != 0);
    lock = 1;
    CS
    lock = 0;
}
```

* Hardware Solution:
(Test & Set)

```
int test-set (int *L) {
    int prev = *L;
    *L = 1;
    return prev;
}
```

together. So will ensure atomicity (Hw like) H/W oriented.




```
while (1) {
    while (test-set (&lock) == 1);
    [CS]
    lock = 0;
}
```

This is the hardware solution.

* Exchange
(See in Galvin).

* Semaphore:
→ Solved critical section problem.
→ Order - sequencing
→ Resource management.

Types :

Binary ⇓ Using 0 & 1.

Counting ⇓ can acheive resource management. eg: 5 printers and f=5

wait (S) → like an entry section. } wait for someone's 'ishara' ENTRY

```
{
    while (S<=0);
    S=S-1;
}
```

CS.

signal (S) } do an 'ishara' ki my work is done, you (someone else) can enter C.S. EXIT

```
{
    S=S+1
}
```

Mutual Exclusion ✓
Progress ✓
But bounded wait ? x.
Everytime only one P is taking a chance to enter.

* Order:
Sequencing and scheduling and/or priority assigning to processes.

eg: P_1 P_2 P_3
↑ ↑ ↑
area radius cost

wait (S_1)
P_1
signal (S_2)
P_2
signal (S_1)

} Binary semaphore is used.

wait (S_1) : P_2 : wait (S_2)
P_1 : signal (S_1) : P_3
signal (S_2) : :

* Resource management:

If you have 5 instances of a resource set $S = 5$ and every process can execute concurrently
Counting semaphore is used.

P_1 P_2 P_3 P_4 P_7 P_6
5 4 3 2 1 0
↓ won't be able to
( ) await executions it will
( ) go to sleep or give break!!

27/2

* 3 concurrent process

X
y
z
} → access & update shared variable

Uses 4 semaphores
a, b, c & d.

X uses (a, b, c)
Y -//- (b, c, d)
Z -//- (c, d, a).

Which of the foll. is deadlock free?

	X	Y	Z
A)	P(a) P(b) P(c)	P(b) P(c) P(d)	P(c) P(d) P(a)
B)	P(b) P(a) P(c)	P(b) P(c) P(d)	P(a) P(c) P(d)
C)	P(b) P(a) P(c)	P(c) P(b) P(d)	P(a) P(c) P(d)
D)	P(a) P(b) P(c)	P(c) P(b) P(d)	P(c) P(d) P(a)

→ (B) is ✓

(A), (C), (D) has cyclic wait for semaphore variable ∴ deadlocked.

P( ), wait( ), down( ) ✓
V( ), signal( ), up( ) ✓

* Classification Synchronization problem (using semaphores)
  i). Producer & consumer
  ii). Reader writer
  iii). Dining philosopher

(i). valid producer ( )
{
  while (T)
  {
    produce ( ).
    wait (E)
    wait (S)
    append ( )
    signal (S)
    Signal (F)
  }
}

Critical

(ii). valid consumer ( )
{
  wait (F)
  wait (S)
  consume ( )
  Signal (S)
  signal (E)
  use ( )
}

Data structure
Semaphore (Sem)
Sem S = 1
Sem E = n
Sem F = 0

Producer must check overflow condition
Consumer must check empty condition.

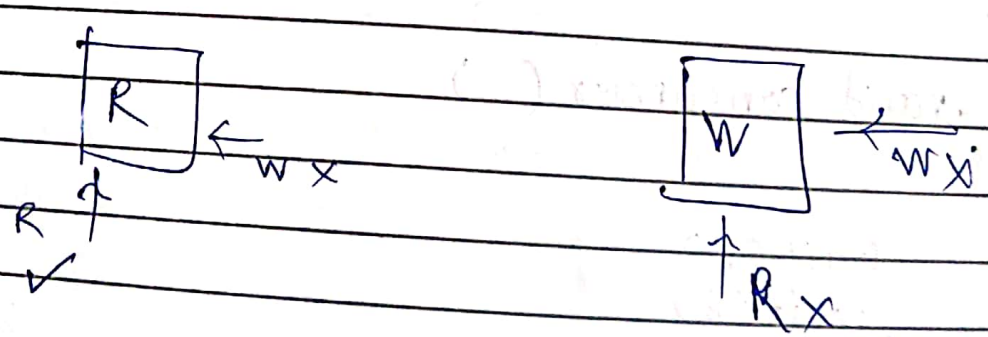
E = Empty = n = how much can be filled?
F = Full = 0 = initially none filled
C = S1 S2 ...
F = 0 1 2 3 ...

* Reader writer Problem:

```
Writer {
    wait (wrt)
    [CS]
    signal (wrt).
}
```

Semaphore write : 1 0

Multiple writers ✗ WW ✗
Multiple readers RR ✓
RW ✗



Now.

Semaphore for readers.
(Multiple readers...).

```
wait (mutex)
    rc++
    if (rc == 1)
        wait (wrt)
    signal (mutex)
        read operator
    wait (mutex)
        rc --
    if (rc == 0)
        signal (wrt).
    signal (mutex)
```

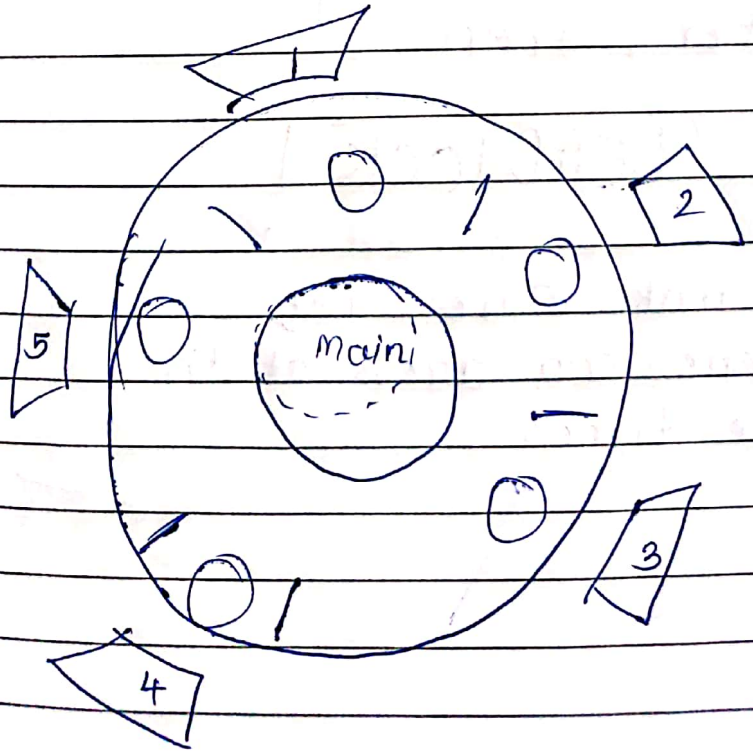
1 = available
0 = occupied

entry → CS

exit

Semaphore : mutex = 1
rc = 0 ... multiple allowed.
wrt = 1.

* Dining philosopher problem:



"Its your kind of a silent and organized party"

exp: sharing chopsticks.

Initially philosophers, will be thinking.

1) #define N 5

```
void p (int i) {
    while (T) {
        think()
        takefork (Ri)
        takefork (Li)
        eat()
        putfork (Li)
        putfork (Ri)
    }
}
```

Can work independently
But not simultaneously.

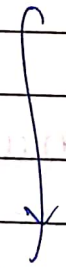
If everyone pick right fork together, then

'DEADLOCK!!'

So make sure, not everyone can access at the same time.

2)

```
void philosopher (int i) {
    while (T) {
        think();
        take fork (Ri)
        if (available (Li)) {
            takefork (Li)
            eat()
            putfork (Ri)
            putfork (Li)
        }
        else {
            putfork (Ri)
            sleep (T)
        }
    }
}
```



for some random time.

That reduces the problem to certain extent, but does not guarantee a system free of starvation.
∴ not correct solution.

3) #define N 5
void philosopher (int i) {

while (T) {
think()
lock (mutex)
take fork (Ri)
take fork (Li)
eat()
put fork (Li)
put fork (Ri)
unlock (mutex)
}
}

Here only one philosopher can eat
Rest (who have a chance to eat),
will also be blocked!!

4) Using Semaphore

States: Thinking (initial), hungry, eating.
N semaphores S[0] --- S[N]

void philosopher (int i) {
while (T) {
think()
take fork (i)
eat()
put_fork (i)
}
}

```
void take_fork (int i) {
        lock (mutex)
        state[i] = hungry
        test (i)
        unlock (mutex)
        down (s[i])
}
void test (int i) {

  if ( state[i] == Hungry &&
       state[left] != eating &&
       state[right] != eating ) {

          state[i] = eating;
          up (s[i]);
    }
}
void put_fork (int i) {
     lock (mutex);
     state[i] = thinking;
     test (left)
     test (right)
     unlock (mutex);
}
```

This is the correct solution.

* OS : Post T-1

Scheduling : Priority, RR
Multilevel, Multilevel feedback queue
Threads, multithreading, pthread library
Deadlocks:
a) Prevention : Those 4 conditions
b) Avoidance : Safe state, RAG, Banker's algo
c) Recovery : Res pre & Proc termination

Process Synchronization
Producer - Consumer problem
Critical Section Problem
a) ME, Progr & Bounded Wait.
b) 2 process solution.
c) Peterson's algorithm
Bakery Algorithm (Original & Simplified)
H/W Test and Set Solution.
Semaphore.
Synchro. problems using Semaphore:
a) Producer - Consumer.
b) Reader Writer problem.
c) Dining Philosopher.

Memory management:
Unit upto Contiguous Memory Allocation.

18/3

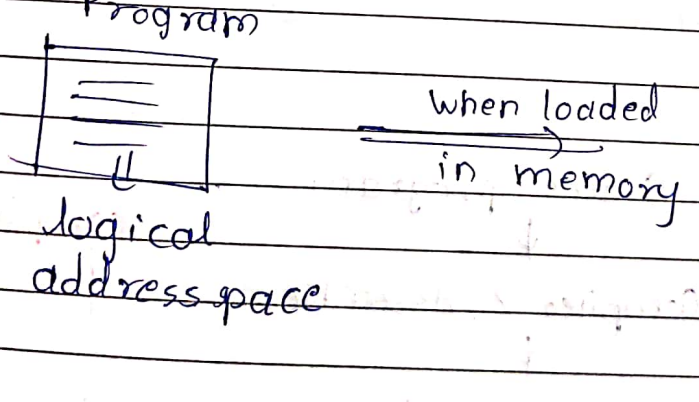
* Memory Management *

Addressing.
Sharing of memory.
Protection.

Address Binding:
- Compile time
- Load time
- Run time.

Address Binding: The process of binding or mapping logical (virtual) address to physical address.

Storing of code and data.
Generally,

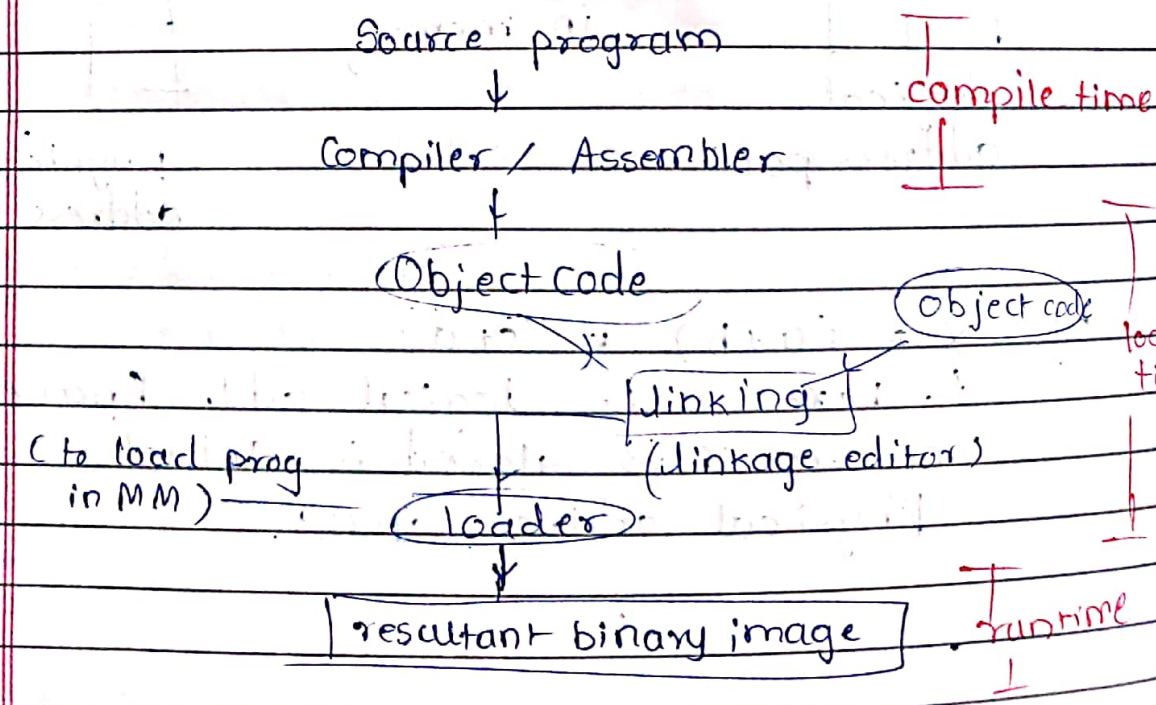


eg: $X = (a+b) * (c+d)$
↳ here X is logical addr space
wherever X is stored is actual physical address space.

* Compile Time:
If during compilation, it is known where the program starts in which physical address (reside in memory). Then 'absolute code' will be generated.

* Load Time:
If not known, where the process will reside at compile time, then load time binding takes place. 'Relocatable code' is generated.

* Run time:
~~Source prog. is there;~~

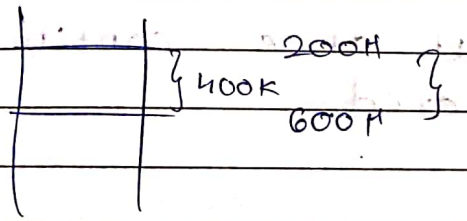


* Types of loading:

Static
All modules/prog and routines are loaded in compile time. Everything required in MM is in it all at once.

Dynamic
All routines & prog. are loaded whenever required in runtime. In MM, stuff taken as per needed.

* Relocation register:
(One of CPU register)
Base register ⇒ Base addr. ⇒ 200h
Limit register ⇒ Size of process.
eg: 400K



Logical address
⇓
Check with limit
+ ... < limit
⇓
If okk, then add with base register

else, not in limit, then trap in OS

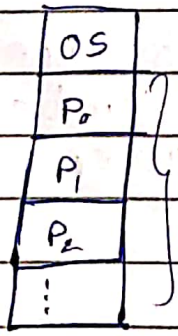
Limit + Reloc register = Physical address of instruction

* Allocation strategy:

- Contiguous Allocation
  - fixed partition
  - variable dynamic partitioning
- Non contiguous Allocation
  - Paging
  - segm Paging
  - Segmentation

Contiguous: One after another; which one available next, fill up next process from that onwards and $ all at once, continuous, not scattered.

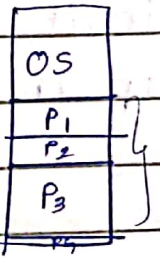
- Fixed



No. of process bound to number of partitions. If one partition has empty space, still next partition of next process.

- Variable

Here partition size is variable.



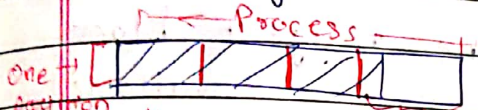
variable size), dynamic as per needed to use max space.

Fixed:

Adv: No other overhead, fixed size!

Disadv: Internal Fragmentation issue !!

Also, there is wastage of memory.

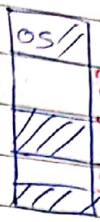


Internal frag. has space but can't be allocated !!

Internal frag

In same partition

Variable:



- Now if 225K process comes, even though there is space, we can't allocate 250K space

This is external fragmentation

* Strategies to allocate spaces: (for processes) (in fixed and variable).

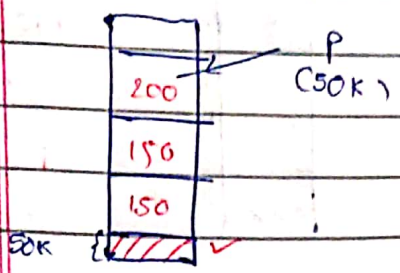
i) First fit.
ii) Best fit.
iii) Worst fit.
iv) Next fit.

1) First fit

Start from starting location whatever you see first, put process there (which can be accomodated.

Proc. can split in 2+ blocks.

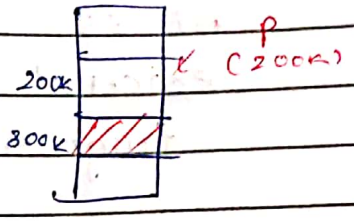
2) Best fit:



Best one or the smallest possible block is allocated to the process.

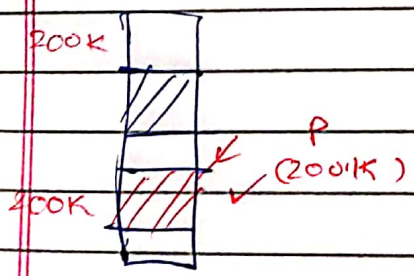
3) Worst Fit:

Give the biggest block to the process available



4) Next Fit:

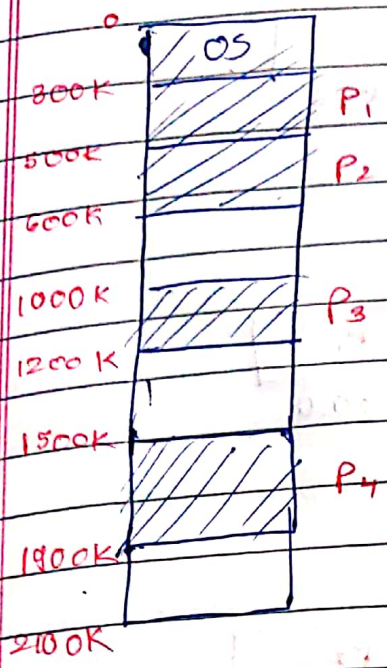
Similar to fit. (first fit). It will not start from first. It will search from current point where it is from after previous allocation



Internal fragmentation:
Cannot be solved (Bcoz size is fixed, can't change)

External fragmentation:
can be solved by 'Compaction'

Compaction:



So total free space
= 400K + 300K + 200K
= 900K.

So now P_4 can be moved to (600K to 1000K) wala slot then we get combined 900K slot free by 'compact'ing allocated space.

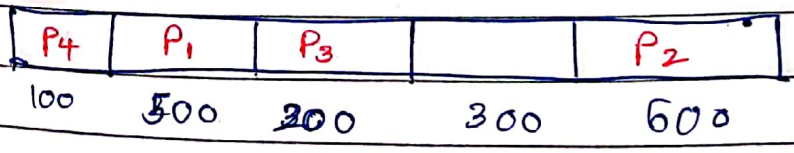
Solve:

Given a memory partition of 100Kb, 500Kb, 200Kb, 300Kb and 600Kb, (partitions in order).

Process size: P_1, P_2, P_3, P_4
212KB 417KB 112KB 426KB.

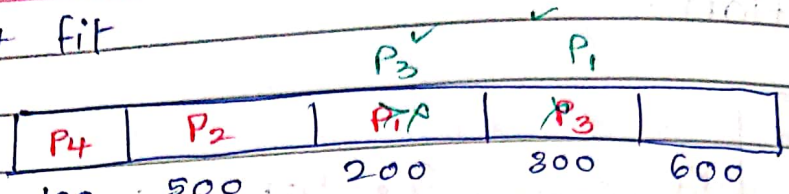
Allocate by:

i) First fit

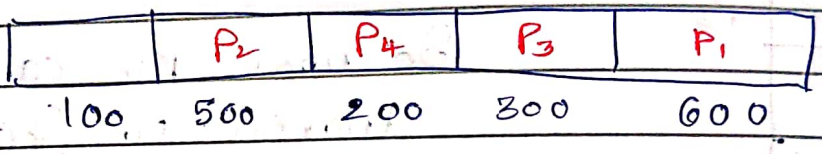


If $P_4 = 426$, then cannot be allocated.

ii) Best fit



iii) Worst fit



iv) Next fit