① FCFS

Criteria : Arrival Time

Mode : Non-preemptive [FCFS cannot be implemented in preemptive mode]

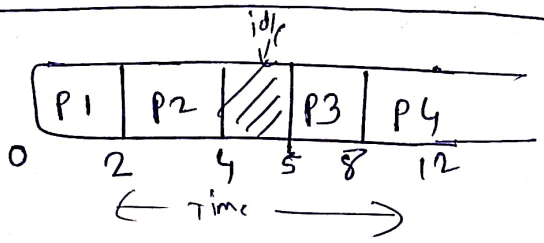
e.g ①

Process No	Arrival time	Burst time	Completion Time	TAT	WT	RT
P1	0	2	2	2	0	0-0=0
P2	1	2	4	3	1	2-1=1
P3	5	3	8	3	0	5-5=0
P4	6	4	12	6	2	8-6=2

Avg: $\frac{14}{4}$ $\frac{3}{4}$

$= 3.5$ $= 0.75$

Gantt chart



TAT = CT - AT

WT = TAT - BT

Response time

RT = Time at which the process got the CPU first time - A.T

```
int time = 0;
for(int i=0; i<n; i++) {
    if(time < at[i]) {
        time = at[i];
    }
    time += bt[i];
    ct[i] = time;
    TAT[i] = ct[i] - at[i];
    wt[i] = tat[i] - bt[i];
}
```

DOCTOR STAMP & SIGNATURE

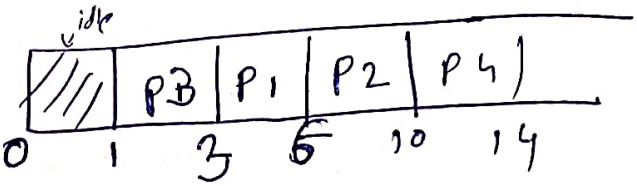
② SJF

Criteria : Burst Time

Mode : "non-preemptive"

Process NO	A.T	B.T	Compl.T	T.AT	W.T	R.T
P1	1	3	6	5	2	3-1 = 2
P2	2	4	10	8	4	6-2 = 4
P3	1	2	3	2	0	1-1 = 0
P4	4	4	14	10	6	10-4 = 6
				25/4 = 6.25	12/4 = 3	

Gantt chart



BT 3 4

Ready Queue P1, P3

P2, P4

```
boolean done[] = new boolean[n]; // track finished process

int time = 0, completed = 0;
while (completed < n){
        int idx = -1;
        int minBT = Integer.MAX_VALUE;

// find process with minimum burst time among arrived ones
        for(int i = 0 ; i<n ; i++ ){
            if( !done[i] && at[i] <= time && bt[i] < minBT ){
                minBT = bt[i];
                idx = i;
            }
        }
        if(idx == -1) {
            time++;
        } else {  // execute the chosen process
            time += bt[idx];
            ct[idx] = time;
            done[idx] = true;
            completed++;
        }
```

DOCTOR STAMP & SIGNATURE

(3) SRTF

Criteria : Burst time

Mode : preemptive

Next Visit Date / /

Process No	A.T	B.T	Com.T	T.A.T	W.T	R.T [CPU First time - A.T]
P1	0	8, 3, 2, 1	9	9	4	0 - 0 = 0
P2	1	3, 1	4	3	0	1 - 1 = 0
P3	2	4	13	11	7	9 - 2 = 7
P4	4	1	5	1 (24/4 = 6)	0 (11/4 = 2.75)	4 - 4 = 0

7/4 = 1.75

Gantt chart

P1	P2	P2	P2	P4	P1	P1	P1	P1	P3
0	1	2	3	4	5	6	7	8	9	13

3 → R.Q = P1, P2, P3

P1, P2, P3 → chk B.T whose lower

4 → P1, P3, P4

5 → P1, P3 → chk B.T → same no.
chk A.T

6 → P1, P3 → chk B.T → P1

7, 8 →

9 → P3 → only

```
rt[i] = bt[i]  // initially RT = BT
int time = 0, completed = 0;
while (completed < n) {
    int idx = -1;
    int minRT = Integer.MAX_VALUE;
// find process with minimum Remaining time among arrived ones
    for (int i=0; i<n; i++) {
        if (at[i] <= time && rt[i] > 0 && rt[i] < minRT) {
            minRT = rt[i];
            idx = i;
        }
    }
    if (idx == -1) {
        time++;
    } else {
        rt[idx]--;   // execute for 1 unit of time
        time++;
        if (rt[idx] == 0) {   // process finished
            completed++;
            ct[idx] = time;
        }
    }
}
```

RX

Patient Name

Age

Address

Date / /

(4) R.R

Criteria : Time Quantum

Mode : preemptive

Process NO	A.T	B T	Compl Time	T.A.T	W.T	R.T
P1	0	~~3~~ 1	12	12	7	0-0=0
P2	1	~~4~~ 0	11	10	6	2-1=1
P3	2	~~2~~ 0	6	4	2	4-2=2
P4	4	1	9	5	4	8-4=4

Given TQ = 2

Ready Queue: ~~P1~~ | P2 | ~~P3~~ | ~~P1~~ | ~~P4~~ | P2 | P1 (0)

P1 | P2 | P3 | P1 | P4 | P2

Running Queue / Gantt chart: P1 | P2 | P3 | P1 | P4 | P2 | P1 (0, 2, 4, 6, 8, 9, 11, 12)

P1 | P2 | P3 | P4 | P2 | P1 (0, 2, 4, 6, 8, 9, 12)

Saving a Running process & Taking saved, load new process

```
Queue<Integer> q = new LinkedList<>();
int time = 0, completed = 0;
boolean inQueue[] = new boolean[n];

//Initially add first arrived process
for(int i=0; i<n; i++){
    if(at[i] <= time && !inQueue[i]){
        q.add(i);
        inQueue[i] = true;
    }
}
while(completed<n){
    if(q.isEmpty()){
        time++;
    }
    for(int i=0; i<n; i++){
        if(at[i] <= time && !inQueue[i]){
            q.add(i);
            inQueue[i] = true;
        }
    }
    continue;
```

```
    int idx = q.poll();
//reverse process for time Quan
    int exectime = min(tq, rt[idx]);
    rt[idx] -= exectime;
    time += exectime
```

Patient Name

Age Gender Height Weight

Address

Next Visit Date / /

DOCTOR STAMP & SIGNATURE

(5) priority Scheduling

Criteria : priority

Mode : preemptive

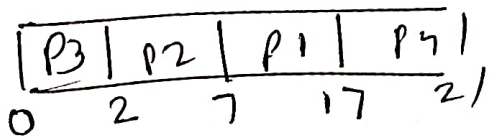
Process No	A.T	B.T	C.T	T.A.T	W.T	Priority
P1	0	4	12	12	8	10
P2	1	3	8	7	3	20
P3	2	2	4	2	0	30
P4	4	1	5	1	0	40

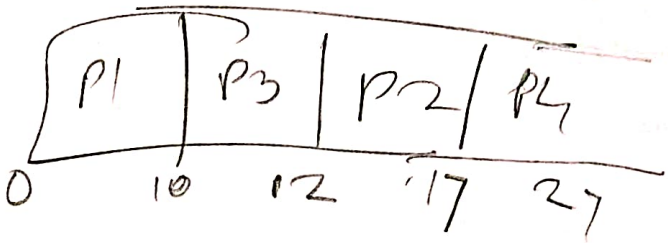
Higher the No, Higher the priority


```
int time = 0, completed = 0;
while(completed < n){
    int idx = -1;
    int bestPr = Integer.MIN_VALUE;
    // find process with highest priority
    for(int i=0; i<n; i++){
        if(at[i] <= time && pr[i] > bestPr && rt[i] > 0){
            bestPr = pr[i];
            idx = i;
        }
    }
    if(idx == -1){
        time++;
        continue;
    }
    rt[idx]--;
    if(rt[idx] == 0){
        completed++;
        int finish = time + 1;
        tat[idx] = finish - at[idx];
        wt[idx] = tat[idx] - bt[idx];
    }
    time++;
}
```

priority non-preemptive

Proc	AT	BT	priority
P1	0	10	2
P2	2	5	1
P3	3	2	0
P4	5	20	3






```
int time = 0, completed = 0;
while (completed < n) {
    int idx = -1;
    int bestPr = Integer.MAX_VALUE;

    find process with highest priority (lower value) among arrived and
    for (int i=0; i<n; i++) {
        if (!done[i] && at[i] <= time && pr[i] < bestPr) {
            bestPr = pr[i];
            idx = i;
        }
    }
    if (idx == -1) {
        time++;
    } else {
        time += bt[idx];
        ct[idx] = time
        .
        .
        done[idx] = true;
        completed++;
    }
}
```

Banker's Algo → Deadlock Avoidance

Safe → Deadlock No

Total A=10, B=5, C=7

Process	Allocation A	B	C	Max Need A	B	C	Available A	B	C	Remaining Need → Max Need - A A	B	C	
P1	0	1	0	7	5	3	3	3	2	7	4	3	P1 X
							2	0	0	Date			
P2	2	0	0	3	2	2	5	3	2	1	2	2	P2
							2	1	1				
P3	3	0	2	9	0	2	7	4	3	6	0	0	
							0	0	2				P3 X
P4	2	1	1	2	2	2	7	4	5	2	1	1	P4
							0	1	0				
P5	0	0	2	5	3	3	7	5	5	5	3	1	P5
							3	0	2				
	7	2	5				10	5	7				

P2 → P4 → P5 → P1 → P3

No Deadlock occurred

Patient Name

Age Gender

Address

Height Weight

Rx

Next Visit Date / /

DOCTOR STAMP & SIGNATURE

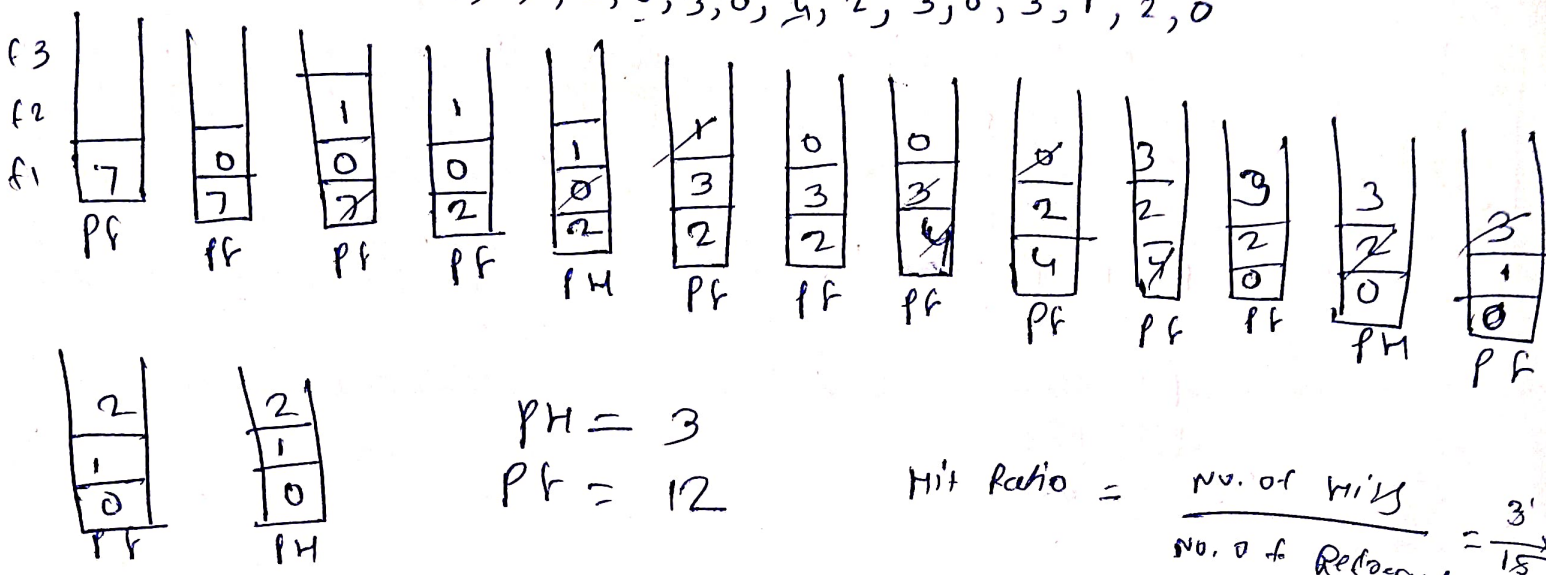
Page Replacement

i] FIFO

Page reference ~~1, 3, 0, 3, 5, 6, 3~~

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

PF = 3



PH = 3

Pf = 12

Hit Ratio = $\frac{\text{No. of Hits}}{\text{No. of References}} = \frac{3}{15}$

20%.

Fault Ration = $\frac{12}{15} \times 100$

= 80%.

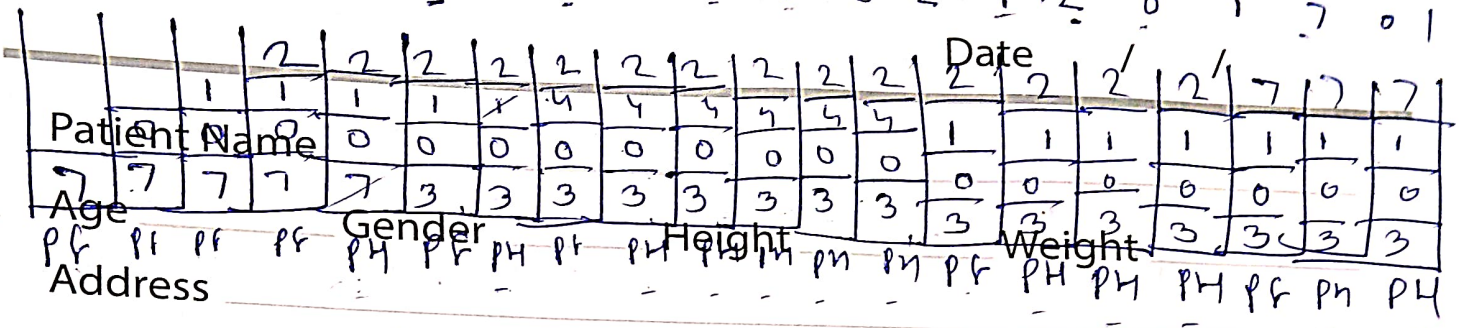
```
int [] pages = {0,1,2};
int capacity = 3;
Queue<Integer> queue = new LinkedList<>();
HashSet<Integer> set = new HashSet<>();

int pagefaults = 0;

for (int page : pages) {
    if (!set.contains(page)) {
        if (set.size() == capacity) {
            int removed = queue.poll();
            set.remove(removed);
        }
        else {
            set.add(page);
            queue.add(page);
            pagefaults++;
        }
    }
}
```

ii] Optimal Page Replacement

Ref. String : 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

P.F = 4

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



Patient Name

Date

Age

Gender

Height

Weight

Address

PF = 8 $\frac{8}{20} \times 100 = 0.4 \times 100 = 40$

PH = 12 $= \frac{12}{20} \times 100 = 0.6 \times 100 = 60$

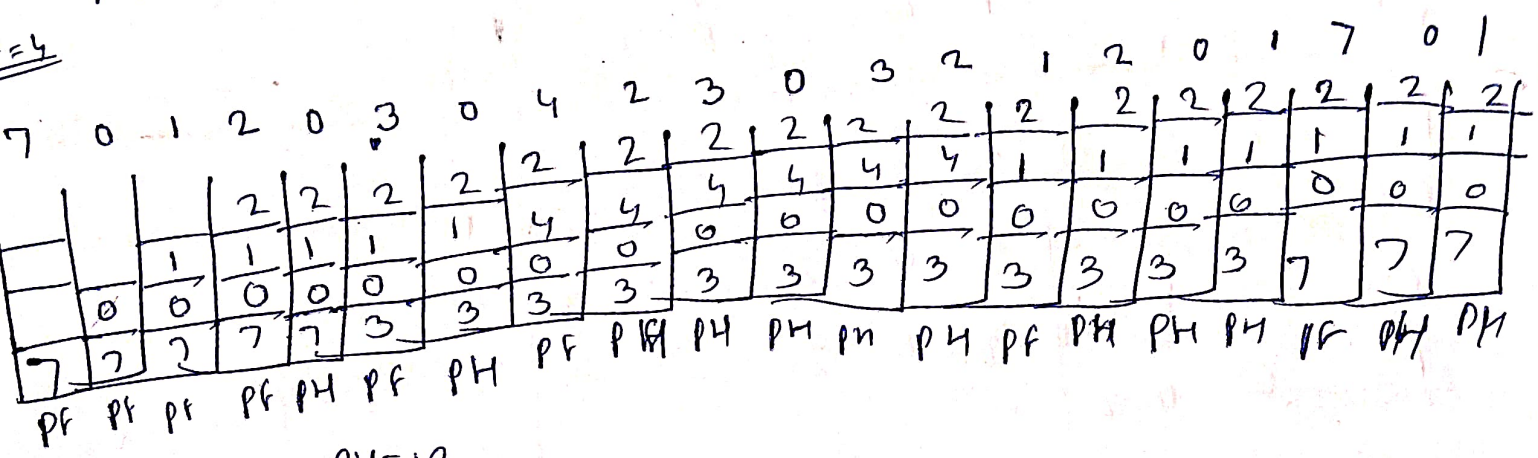
Rx

Next Visit Date / /

DOCTOR STAMP & SIGNATURE

] LRV
- Replace the least recently used page in past

=4



PH=12
PF=8

Semaphore — [~~Us~~ Method, used to prevent race condition]

→ Managing concurrent process by using Integer value variable

2 Types → Binary (0 or 1)
→ Counting (−∞ to +∞)

Used for Synchronization of processes

P(S) ~~wait, sleep, down~~

— Synchronization

P1, P2, P3 Need to Entry code

CS — critical section / Code segment

Exit Code section

entry code & exit code various operations

1) P( ) , Down , wait

2) V( ) up , signal , post , release

P4, P5 enter Block state

entry section

1) down (semaphore S){

S.value = S.value − 1;

if (S.value < 0){

PW . proc (P1A) in Suspended list sleeping;

}

else

return;

}

2) up (Semaphore S){

S.value = S.value + 1;

if (S.value ≤ 0){

Select a process from suspended list P4

wake up ();

}

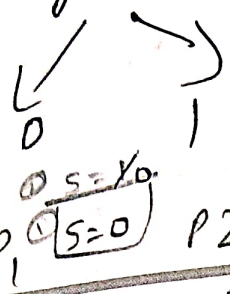
}

Now P4 Again try

Now P5 Again try

PF=4

Binary Semphore



Date / /

Down(S) Down(S)

Patient Name Gender Height Weight

Age

Address

P1 → Block
P1 → X0

P2 → Block
P2 → Block

Down(semaphor S) {

if(S.value == 1) {

S.value = 0;

} else {

Block this proces & Place in suspend list, sleep();

}

}

up(semaphor S) {

if(suspend list is empty) {

S.value = 1;

} else {

select a proces from suspend list & wake up();

}

}

Next Visit Date / /

DOCTOR STAMP & SIGNATURE

Mutex -> Mutual exclusion lock is used to prevent two threads from accessing the same resources at the same time

e.g. Two threads trying to update same bank account,

Date:

Patient Name: producer should not produce if buffer is full / Consumer should not consume if buffer is empty

Age: Both must not access buffer at the same time Sex: Height: Weight:

Mutex -> prevents simultaneous access

Complains Of:

Semaphore -> Rx used for synchronization

(1) Mutex only ensured exclusive access

(2) Semaphore ensured proper ordering

On Examination:

producer cannot insert if buffer is full

Consumer cannot remove if buffer is empty

This ordering is not possible using only mutex

so semaphore required

Investigation:

Doctor Stamp & Signature