INDEX

NAME: N. PRIYANKA STD.: _____ SEC.: _____ ROLL NO.: _____ SUB.: OS LAB

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
01	08/05/24	LAB-1		
02	15/05/24	LAB-2		
0.1	08/05/24	FCFS		
2	08/05/24	SJF		
3	08/05/24	SJF (preemptive)		
4	15/05/24	Priority scheduling		
5	15/05/24	Round Robbin scheduling		
6	5/06/24	Multi-level queue		
7	5/06/24	a) Rate - monotonic		
8	5/06/24	b) Earliest deadline first		
9	5/06/24	c) Proportional scheduling		
10	12/06/24	Producer - consumer		
11	19/06/24	Dining Philosopher		
12	19/06/24	Deadlock avoidence		
13	19/06/24	Deadlock detection		10/7 completed
14	3/07/24	Memory allocation		
15	10/07/24	Page replacement		

8/05/24 LAB - 1

(1) Write a C program to simulate the following non-preemptive CPU scheduling algorithm to find turn around time and waiting time
(i) FCFS

```
#Include <stdio.h>
# define MAX 10

void fcfs (int n, int at[], int bt[])
{
    int ct [MAX];
    int tat[MAX];
    int wt[MAX];

    int total_wt =0;
    int total_tat =0;
    int current_time = 0;

    for(int i =0; i<n ; i++)
    {
        ct[i] = -1;
    }

    for (int i = 0 ; i<n; i ++)
    {
        if ( current_time <at[i])
            current_time = at[i];
        else{
            ct[i] = current_time + bt[i];
            current_time = ct[i];
        }
    }
```

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

printf("\n Process \t Arrival time \t Burst time \t
    completion time \t Turn around Time \t
    Waiting time\n");
for (int i=0; i<n; i++)
{
    printf("%d \t %d \t %d \t %d \t %d \t %d \t%d\n",
        i+1, at[i], bt[i], ct[i], tat[i], wt
}

printf("Average waiting time: %f", total_wt/n);
printf("Average ~~total~~ turn around time: %f",
    total_tat/n);
}
int main()
{
    int n, i;
    printf("Enter the number of process:");
    scanf("%d", &n);
    int at[n], bt[n];

    printf("Enter Arrival time:");
    for (int i=0; i<n; i++)
        scanf("%d", at[i]);
```

```
printf(" Enter the burst time : \n");
for(int i=0; i<n; i++)
 {
        scanf(" %d", & bt[i]);
 }
fcfs (n, at, bt);
}
```

Output:

Enter the number of processes : 4
Enter the arrival time:
0
1
5
6
Enter the Burst time:
2
2
3
4

Process	Arrival time	Burst time	Ct	tat	wt
1	0	2	2	2	0
2	1	2	4	3	1
3	5	3	8	3	0
4	6	4	12	6	2

Average waiting time = 0.75
average turn around time = 03.50

SJF (Shortest Job First)
non- Preemptive.

```
#include <stdio.h>
#define MAX 10

void sjf( int n, int at[], int bt[])
{
    int ct[MAX];
    int tat[MAX];
    int wt[MAX];

    int total_wt = 0;
    int total_bt = 0;

  for (int i = 0; i<n;i++)
   {
      ct[i] = -1;
   }
 for (int i=0; i<n;i++)
   {
     for(int j = 0; j<n ;j++)
       {
          if ( bt[j] > bt[j+1])
            {
              int temp = bt[j];
              bt[j] = bt[j+1];
              bt[j+1] = ~~bt~~ temp;

              int temp_at = at[j];
              at[j] = at[j+1];
              at[j+1] = temp-at;
            }
       }
   }
```

```
for(int i=0; i<n; i++)
{
    if(current_time<at[i])
    {
        current_time = at[i];
    }
    ct[i] = current_time + bt[i];
    current_time = ct[i];
}

for(int i=0; i<n; i++)
{
    tat[i] = ct[i] - at[i];
    total_tat = total_tat + tat[i];
}

for(int i=0; i<n; i++)
{
    wt[i] = tat[i] - bt[i];
    total_wt = total_wt + wt[i];
}

printf("\n Process \t Arrival time \t Burst time \t
    Completion Time \t Turn around time \t
    waiting time \n");

for(int i=0; i<n; i++)
{
    printf("%d \t %d \t %d \t %d \t %d \t %d \t",
        i+1, at[i], bt[i], ct[i], tat[i],
        wt[i]);
}

printf("\n Average turn around time :
    %.2f", total_tat/n);
printf("\n Average waiting time : %.2f",
    total_wt/n);
}
```

```
int main()
{
    int n;
    printf(" Enter the number of processes:");
    scanf("%d", &n);

    int at[n];
    int bt[n];

    printf("Enter the arrival time:\n");
    for(int i=0; i<n; i++)
      {
          scanf("%d", &at[i]);
      }
    printf(" Enter the Burst time :\n");
     for(int i=0; i<n; i++)
     {
          scanf("%d", &bt[i]);
     }
     sij(n, at, bt);

}
```

output:

```
Enter the number of proces:4
Enter the arrival time:
0
0
0
0
Enter the Burst time:
6
8
```

A

3

Process	Arrival time	Burst time	Ct	tAt	Wt
1	0	3	3	3	0
2	0	6	9	9	3
3	0	7	16	16	9
4	0	8	24	24	16

Average waiting time: 7

Average turnaround time: 13.

4
3

Process	Arrival time	Burst time	CT	TAT	WT
1	0	2	3	3	0
2	0	6	9	9	3
3	0	7	16	16	9
4	0	8	24	24	16

Average waiting time : 7
Average turnaround time : 13.

SJF preemptive:

```
#include <stdio.h>
#define MAX 10

int find_min (int arr[], int n)
{
    int min = arr[0];
    int index = 0;
    for (int i=0; i<n; i++)
    {
        if (arr[i] < min)
        {
            min = arr[i];
            index = i;
        }
    }
    return index;
}

void sjf_preemptive( int n, int at[], int bt[])
{
    int ct[MAX] = {0};
    int tat[MAX] = {0};
    int wt[MAX] = {0};
    int rt[MAX];

    int total_wt = 0;
    int total_tat = 0;

    for( int i =0; i<n; i++)
    {
        rt[i] = bt[i];
    }
```

```
int current_time = 0;
int completed process = 0;

while ( completed_process < n)
{
    int available_process[MAX];
    int available_count = 0;

    for( int i = 0; i<n; i++)
    {
        if ( at[i] <= current_time && rt[i] > 0)
        {
            available_process[available_count] = i;
            available_count ++;
        }
    }
    if( available_count == 0)
    {
        current_time ++;
        ~~continue~~
    }

int sjf_index = available process [find min
                (rt, available_count)];

~~rt[shortest~~
rt[sjf_index] --;
current_time ++;

if ( rt[sjf_index] == 0)
{
    completed_process++;
    ct[ sjf_index ] = current_time;
    tat[ sjf_index ] = ct[sjf_index] - at[sjf_index];
```

```
        wt[sjf_index] = tat[sjf_index] - bt[sjf_index];
        total_wt += wt[sjf_index];
        total_tat += tat[sjf_index];
      }
    }

    printf("\n Process \t Arrival time \t Burst time \t completion
        time \t Turn around time \t waiting time \t\n");
    for(int i=0;i<n;i++)
    {
        printf("%d \t %d \t %d \t %d \t %d \t %d \t\n",
            i+1, at[i], bt[i], ct[i], tat[i], wt[i]);
    }

    printf("\n Average waiting time : %.2f", (float)
            total_wt/n);
    printf("\n Average turn around time : %.2f", (float
            total_tat/n);
}

int main()
{
    int n;
    printf("Enter the number of process");
    scanf("%d", &n);

    int at[MAX];
    int bt[MAX];

    printf("Enter the arival time:\n");
    for(int i=0; i<n;i++)
    {
        scanf("%d", &at[i]);
    }
```

```
printf(" Enter the burst time:\n");
for(int i=0; i<n; i++)
{
    scanf(" %d", &bt[i]);
}

sjf_preemptive(n, at, bt);

}
```

output:

Enter the number of processes: 5
Enter the arrival time:
2
1
4
0
2
Enter the Burst time
1
5
1
4
2

Process	Arrival time	Burst time	CT	TAT	WT
1	2	1	3	1	0
2	1	5	7	6	1
3	4	1	8	4	3
4	0	4	11	11	7
5	2	2	13	11	9

Average waiting time: 4.00
Average turn around time: 6.60

15/05/24

LAB ->

Priority scheduling (Non-preemptive)

```
#include <stdio.h>
#define MAX 10

void priority (int n, int at[], int bt[], int p[])
{
    int ct[MAX] = {0};
    int tat[MAX] = {0};
    int wt[MAX] = {0};

    int total_wt = 0;
    int total_tat = 0;

    int bt_copy[MAX];
    for (int i=0; i<n; i++)
    {
        bt_copy[i] = bt[i];
    }

for (int i = 0; i<n; i++)
{
    for (int j=0; j<n; j++)
    {
        if (p[i] < p[j])
        {
            int temp = at[i];
            at[i] = at[j];
            at[j] = temp;

            temp = bt[i];
            bt[i] = bt[j];
```

```
            bt[j] = temp;

            temp = p[i];
            p[i] = p[j];
            p[j] = temp;
        }
    }
}

    ct[0] = at[0] + bt[0];
    tat[0] = ct[0] - at[0];
    wt[0] = tat[0] - bt_copy[0];

    total_wt = total_wt + wt[0];
    total_tat = total_tat + tat[0];

    for(int i=0; i<n; i++)
    {
        ct[i] = ct[i-1] + bt[i];
        tat[i] = ct[i] - at[i];
        wt[i] = tat[i] - bt_copy[i];

        total_wt += wt[i];
        total_tat += tat[i];

    }

    printf("\n Process \t Arrival time \t Burst time \t Priority
        \t completion time \t Turnaround time \t waiting
        time \t\n");
    for(int i=0; i<n; i++)
    {
        printf("%d %t %d \t %d \t %d \t %d \t %d \t %d \t %d \t\n",
            i+1, at[i], p[i], ct[i], tat[i], wt[i]);
    }
```

```
    printf(" Average waiting time :", (float) total_wt/n);
    printf(" Average turnaround time : %.2f ",
                (float) total_tat/n);

}

int main()
{
    int n;
    printf(" Enter the number of process ");
    scanf(" %d ", &n);

    int at[MAX], bt[MAX], p[MAX];

    printf("Enter the Arrival time:");
    for(int i=0; i<n; i++)
      {
          scanf(" %d", &at[i]);
      }
    printf(" Enter the Burst time: ");
    for(int i=0; i<n; i++)
        scanf(" %d", &bt[i]);

    printf(" enter the priority ");
    for(int i=0; i<n; i++)
        scanf(" %d", &p[i]);

    priority(n, at, bt, p);

    return 0;
}
```

output:

Enter the number of process: 4
Enter the arrival time
0
1
2
4
Enter the burst time:
5
4
2
1
Enter the priority:
10
20
30
40

Process	Arrival time	Burst time	Priority	TAT	WT	CT
1	4	5	40	1	4	5
2	2	4	30	5	1	7
3	1	3	20	10	8	11
4	0	1	10	16	1	16

Average waiting time: 5.00
Average turn around time: 8.00.

ROUND ROBIN Scheduling

```
#include <stdio.h>

#define MAX 10
#define QUANTUM 2

void round_robin (int n, int at[], int bt[])
{
    int ct[MAX] = {0};
    int tat[MAX] = {0};
    int wt[MAX] = {0};

    int total_wt = 0;
    int total_tat = 0;

    int bt_copy[MAX];
    for(int i = 0; i < n; i++)
    {
        bt_copy[i] = bt[i];
    }
    int time = 0;
    int i = 0;

    while(1)
    {
        int done = 1;
        for(int i = 0; i < n; i++)
        {
            if (bt_copy[i] > 0)
            {
                done = 0;
                if (bt_copy[i] > QUANTUM)
                {
```

```
            time += QUANTUM;
        }
        else
        {
            time += bt_copy[i];
            bt_copy[i] = 0;
            ct[i] = time;
            tat[i] = ct[i] - at[i];
            wt[i] = tat[i] - bt[i];
            total_wt += wt[i];
            total_tat += tat[i];
        }
    }
}
if (done)
    break;
}

printf("\n Process \t Arrival time \t Burst time \t Completion
    Time \t Turn around time \t waiting time\n");
for (int i=0; i<n; i++)
{
    printf("%d \t %d \t %d \t %d \t %d \t %d \t",
        i+1, at[i], bt[i], ct[i], tat[i], wt[i]);
}

printf("Average waiting time : %.2f", (float)
    total_wt/n);
printf("Average turn around time %.2f", (float)
    total_tat/n);
}
int main()
{
    int n;
```

```
printf("Enter the number of process:");
scanf("%d", &n);

int at[MAX], bt[MAX];

printf("Enter arrival time:\n");
for(int i=0; i<n; i++)
 {
    scanf("%d", &at[i]);
 }

printf("Enter burst time:\n");
for(int i=0; i<n; i++)
{
    scanf("%d", &bt[i]);

}

round_robin(n, at, bt);

}
```

output:

```
Enter the number of process: 5
Enter arrival time
0
1
2
3
4
Enter the burst time
5
3
```

1

2

3

process	Arrival time	Burst time	CT	TAT	WT
1	0	5	14	14	9
2	1	3	12	11	8
3	2	1	5	3	2
4	3	2	7	4	2
5	4	3	13	9	6

Average waiting time : 5.40

Average turnaround time = 8.20.

LAB-3

MULTI LEVEL QUEUE:

```
#include <stdio.h>

void sort (int proc_id[], int at[], int bt[], int n)
{
    int min, temp;
    for (int i=0; j<n-1; i++)
    {
        for(int j = i+1; j<n; j++)
        {
            if(at[j] < at[i])
            {
                temp = at[i];
                at[i] = at[j];
                at[j] = temp;

                temp = bt[i];
                bt[i] = bt[j];
                bt[j] = temp;

                temp = proc_id[i];
                proc_id[i] = proc_id[j];
                proc_id[j] = temp;
            }
        }
    }
}

void simuletFCFS (int proc_id[], int at[], int bt[],
                  int n, int start_time)
{
    int c = start_time, ct[n], tat[n], wt[n];
```

```
    double tat = 0.0,
double ttot = 0.0;

    for(int i = 0; i<n; i++)
    {
        if (c>= at[i])
            ct = bt[i];
        else
            c = at[i] + bt[i];
            ct[i] = c;
    }

    for (int i = 0 ; i<n; i++)
        tat[i] = ct[i] - at[i];
    for(int i=0; i<n ; i++)
        wt[i] = tat[i] - bt[i];
    printf (" pid \t AT\t BT\t CT \tTAT\t WT \n");
    for (int i=0;i <n;i++)
    {
        printf("%d \t %d\t %d\t %d\t %d\t %d\t %d\t",
            proc_id[i], at[i], bt[i], ct[i], tat[i],
            wt[i];
        ttat += tat[i];
        twt += wt[i];
    }
    printf(" Average Turnaround Time : %.2f ms\n",
            tat/n);
    printf(" Average Wailting Time : %.2 f \n", twt\n");
}

void main()
{
    int n;
    printf("Enter the number of processes:");
```

```
scanf("%d", &n);

int proc_id[n], at[n], bt[n], type[n];
int sys_proc_id[n], sys_at[n], sys_bt[n];
user_proc_id[n], user_at[n], user_bt[n];
int sys_count=0;
int user_count=0;

for(int i=0;i<n;i++)
{
proc_id[i] = i+1;
printf("Enter arrival time, burst time and
        type");
printf(" 10 for system, 1 for user) for
        process %d", i+1);
scanf("%d %d %d", &at[i], &bt[i],
        &type[i]);

if( type[i] == 0)
{
    sys_proc_id[sys_count] = proc_id[i];
    sys_at[sys_count] = at[i];
    sys_bt[Bsys_count] = bt[i];
    sys_count++;
}
else{
        user_proc_id[user_count] = proc_id[i];
        user_at[user_count] = at[i];
        user_bt[user_count] = bt[i];
        user_count++;
    }
}

sort(sys_proc_id, sys_at, sys_bt, sys_count);
sort(user_proc_id, user_at, user_bt, user_count);
```

```
printf("System Process scheduling:\n");
simulateFCFS(sys_procid, sys_at, sys_bt,
    sys_count, 0);

int system_end_time = 0;
if (sys_count > 0)
{
    system_end_time = 0;
    if (system_end_time = sys_at[sys_count-1]
                + sys_bt[sys_count-1];
    for (int i = 0; i < sys_count-1; i++)
    {
        if (sys_at[i+1] > system_end_time)
        {
            system_end_time = sys_at[i+1];
        }
        system_end_time += sys_bt[i];
    }
}
printf("\n User Process scheduling:\n");
simulateFCFS(user_proc_id, user_at,
        user_bt, user_count, system_end_
        time);
}
```

OUTPUT:

```
Enter number of processes: 5
Enter arrival time, burst time and type(0 for system,
1 for user) for process 1: 0 2 0
            for process 2: 1 3 1
            for process 3: 2 1 0
            for process 4: 3 4 1
            for process 5: 4 2 1
```

System process scheduling

PID	AT	BT	CT	TAT	WT
1	0	2	2	2	0
3	2	1	3	1	0

Average turn around time : 1.50
Average waiting time : 0.0ms

User processing scheduling:

PID	AT	BT	CT	TAT	WT
2	1	3	8	7	4
4	3	4	12	9	5
5	4	2	14	10	8

Average turn around time : 8.67 ms
Average waiting time : 5.67 ms.

15/6/21

P-4 RATE MONOTONIC SCHEDULING

a)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void sort (int proc[], int b[], int bt[], int n[])
{
        int temp = 0;
        for (int i=0; i<n; i++)
        {
            for (int j = i; i<n; i++)
            {
                if (bt[j] < bt[i])
                {
                    temp = bt[i];
                    bt[i] = bt[j];
                    bt[j] = temp;
                    temp = b[j];
                    b[j] = b[i];
                    b[i] = temp;
                    temp = proc[i];
                    proc[i] = proc[j];
                    proc[j] = temp;
                }
            }
        }
}

int gcd (int a, int b)
{
    int r;
    while (b>0)
    {
        r = a%b;
```

```
        a = b;
        b = r;
    }
    return a;
}

int lcmul(int p[], int n)
{
    int lcm = p[0];
    for(int i = 1; i<n; i++)
    {
        lcm = (lcm * p[i]) / gcd(lcm, p[i]);
    }
    return lcm;
}

void main()
{
    int n;
    printf("Enter the number of processes:");
    scanf("%d", &n);
    int proc[n], b[n], bt[n], rem[n];
    printf("Enter the Burst Time:\n");
    for(int i=0; i<n; i++)
    {
        scanf("%d", &b[i]);
        rem[i] = b[i];
    }
    printf("Enter the time periods:\n");
    for(int i=0; i<n; i++)
        scanf("%d", &bt[i]);
    for(int i=0; i<n; i++)
        proc[i] = i+1;
```

```
sort (proc, b, pt, n)

//LCM
    int l = lcmul(pt, n);
    printf("LCM = %d\n", l);
printf("\n Rate Monotonic scheduling:\n");
printf("PID\t Burst time\t Period\t");
for(int i=0; i<n; i++)
    printf("%d\t %d\t %d\n", proc[i], b[i], pt[i]);

double sum = 0.0;
for(int i=0; i<n; i++)
{
    sum += (double) b[i] / pt[i];
}
double rhs = n * (pow(2.0, (1.0/n)) - 1.0);
printf("\n%.1f <= %.2f => %s\n", sum,
        rhs, (sum<=rhs) ? "true" : "false");
if(sum > rhs)
exit(0);

printf("scheduling occurs for %d\n", l);

int time=0, prev=0, x=0;
while(time<l)
{
    int f=0;
    for(int i=0; i<n; i++)
    {
        if(time % pt[i] == 0)
        rem[i] = b[i];
        if(rem[i] > 0)
        {
            if(prev != proc[i])
            {
```

```
            printf(" %d onwards : Process %d running", time, proc[i]);
            prev = proc[i];
        }
        rem[i]--;
        f = 1;
        break;
        x = 0;
    }
}
if(!f)
{
    if(x != 1)
    {
        printf(" %d ms CPU is idle", time);
        x = 1;
    }
}
time++;
}
}
```

OUTPUT:

```
Enter the number of processes: 3
Enter CPU Burst time:
3
2
2
Enter the time period:
20
5
10
LCM : 20
```

Rate monotonic scheduling

PID	Burst	Period
2	2	5
3	2	10
1	3	20

$0.7000 <= 0.79763 \Rightarrow$ True.

Scheduling occurs for 20ms.

0ms : Process 2 is running
2ms : Process 3 is running
4ms : Process 1 is running
5ms : Process 2 is running
7ms : Process 1 is running
8 onwards: CPU is idle
10 ms : Process 2 running.

Q.4 EARLIEST DEADLINE FIRST:

b)

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

void sort(int proc[], int d[], int b[], int pt[], int n)
{
    int temp=0;
    for(int i=0; i<n; i++)
    {
        for(int j=i; j<n; j++)
        {
            if(d[j]<d[i])
            {
                temp = d[j];
                d[j] = d[i];
                d[i] = temp;
                temp = pt[i];
                pt[i] = pt[j];
                pt[j] = temp;
                temp = b[j];
                b[j] = b[i];
                b[i] = temp;

                temp = proc[i];
                proc[i] = proc[j];
                proc[j] = temp;
            }
        }
    }
}
int gcd(int a, int b)
{
```

```
    int r;
    while( b>0)
    {
        r = a % b;
        a = b;
        b = r;
    }
    return a;
}
int lcmul (int p[], int n)
{
    int lcm = p[0];
    for(int i = 1; i<n ;i++)
    {
        lcm = (lcm * p[i])/ gcd (lcm, p[i]);
    }
    return lcm;
}

void main ()
{
    int n;
    print ("Enter the number of processes : ");
    Scanf(" %d", &n);
    int proc[n], b[n], pt[n], d[n], rem[n];
    print f("Enter the CPU burst time");
    for (int i=0; i<n ;i++)
    {
        scanf(" %d", &b[i]);
        rem[i] = b[i];
    }
    print ("Enter the deadlines: \n");
    for(int i =0 ; i<n ;i++)
```

```
scanf("%d", &d[i]);
printf("Enter the time period");
for(int i=0; i<n; i++)
scanf("%d", &pt[i]);
for(int i=0; i<n; i++)
    proc[i] = i+1;

sort(proc, d, b, pt, n);
int l = lcmul(pt, n);

printf("Earliest Deadline scheduling");
printf("\n PID\t Burst\t Deadline\t Period\t");
for(int i=0; i<n; i++){
printf("%d \t %d \t %d \t %d \t %d\n",
        proc[i], b[i], d[i], pt[i]);

printf("scheduling occurs for %d \n", l);

int time=0; prev=0; z=0;
int nextDeadlines[n];
for(int i=0; i<n; i++)
{
    nextDeadlines[i] = d[i];
    rem[i] = b[i];
}
while (time<l)
{
    for(int i=0; i<n; i++)
    {
        if (time%pt[i] == 0 && time != 0)
        {
            nextDeadline[i] = time + d[i];
            rem[i] = b[i];
        }
    }
}
```

```
int minDeadline = 1+1;
int taskToExecute = -1;
for(int i=0; i<n; i++)
{
    if (rem[i]>0 && ~~nextDeadline~~ NextDeadline[i]
            < minDeadline)
        {
            minDeadline = nextDeadlines[i]<
              minDeadline)
            {
                min deadline = nextDeadline[i];
                taskToExecute = i;
            }
        }
    if(taskToExecute != -1){
    printf("%d : Task %d is running \n",
            time, proc[TaskToExecute]);
          rem[taskToExecute]--;
        }
        else {
            printf("%d ms: CPU is idle \n", time);
        }
        time++
    }
}.
```

OUTPUT:

```
Enter the number of process : 3
Enter CPU Burst time:
 3
 2
 2
Enter the deadines:
```

7
4
8
Enter the time period:
20
5
10

Earliest Deadline scheduling:

PID	Burst	Deadlines	Period
2	2	4	5
1	3	7	20
3	2	8	10

Scheduling occurs for 20ms.

5/6

0ms : Task 2 is running
1 ms : Task 2 is running
2 ms : Task 1 is running
3ms : Task 1 is running
4ms : Task 1 is running
5ms : Task 3 is running
6ms : Task 3 is running
7ms : Task 2 is running
8ms : Task 2 is running
9ms : CPU is idle
10ms : Task 2 is running
11ms : Task 2 is running
12ms : Task 3 is running
13ms : Task 3 is running
14ms : CPU is idle
15ms : Task 2 is running
16ms : Task 2 is running
17ms : CPU is idle
18ms : CPU is idle
19ms : CPU is idle

05/06/24

P 11 (B) PROPORTIONAL SCHEDULING:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX_TASKS 10
#define MAX_TICKETS 100
#define MAX_UNIT_DURATION_MS 100

struct Task
{
    int tid;
    int tickets;
};

void schedule(struct Task tasks[], int num_tasks,
              int *time_span_ms)
{
    int total_tickets = 0;
    for (int i = 0; i < num_tasks; i++)
    {
        total_tickets += tasks[i].tickets;
    }
    srand(time(NULL));

    int current_time = 0;
    int completed_tasks = 0;

    printf("Process Scheduling");
    while(completed_tasks < num_tasks)
    {
        int winning_tasks < num_tasks)
    {
```

```
int winning_ticket = rand() % total_tickets;
int cumulative_tickets = 0;

for(int i = 0; i < num_tasks; i++)
{
    cumulative_tickets += tasks[i].tickets;

    if(winning_ticket < cumulative_tickets)
    {
        printf("Time %d - %d: Task %d is
            running \n", current_time + 1,
            tasks[i].tid);
        current_time++;
        break;
    }
}

completed_tasks++;
}

* time_span_ms = current_time * TIME_UNIT_DURATION

int main()
{
    struct Task tasks[MAX_TASKS];
    int num_tasks;
    int time_span_ms;

    printf("Enter the number of tasks");
    scanf("%d", &n);

    if(num_tasks <= 0 || num_tasks > MAX_TASK)
    {
        printf("Invalid number of Tasks");
        return;
    }
```

```
        printf("Enter the number of tickets for each
                task : \n");
        task[i].id = i+1;
        printf(" Task %d tickets ", task[i].id);
        scanf("%d", &tasks[i].tickets);
    }
    printf("\n Running Tasks : \n");
    Schedule (tasks, num_tasks, & time_span_ms);

    printf("\n Time span of the Gantt Chart : %d
            milliseconds \n", time_span_ms);
    return 0;
}
```

OUTPUT:

```
Enter the number of tasks : 3
Enter the number of tickets for each task :
Task 1  tickets : 10
Task 2  tickets : 20
Task 3  tickets : 30

Running tasks:
Process scheduling
Time 0-1 : Task 3 is running
Time 1-2 : Task 3 is running
Time 2-3 : Task 1 is running

Time span of gantt chart : 300 milliseconds.
```

12/06/24

Program - 5

PRODUCER CONSUMER PROBLEM USING SEMAPHORE

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

int mutex = 1, full=0 , empty = 5, x = 0;

int main()
{
 int n;
void producer();
void consumer();
int wait (int);
int signal (int);

printf(" 1.Producer \n 2. Consumer \n 3. Exit ");
while(1)
{
        printf("\nEnter your choice");
        scanf("%d", &n);
        switch(n);
        {
            case 1:
                if ((mutex == 1) && (empty != 0))
                producer();
                else
                printf(" Buffer is full");
                break;
            case 2:
                if ((mutex == 1) && ( full != 0))
                  consumer();
                else
```

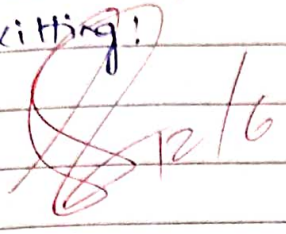
```
            printf("Buffer is empty");
            break;
        case 3:
            exit(0);
            break;
    }
  }
  return 0;
}

int wait (int s) {
        return(--s);
}

int signal (int s)
{
        return(++s);
}

void producer ()
{
        mutex = wait (mutex);
        full = signal (full);
        empty = wait( empty);
        x++;
        printf("\n Producer produces the item %d", x);
        mutex = signal (mutex);
}

void consumer ()
{
    mutex = wait(mutex);
    full = wait( full);
    empty = signal (empty);
```

```
        printf("\n In consumer consumes item %d", x);
        x--;
        mutex = signal(mutex);
}
```

Output:

```
1. Producer
2. consumer
3. Exit
Enter your choice : 1

Producer produces the item 1
Enter your choice : 1

Producer produces the item 2
Enter your choice : 1

Producer produces the item 3
Enter your choice : 1

Producer produces the item 4
Enter your choice : 2

consumer consumes item 4
Enter your choice : 2
6
consumes consumes item 3
Enter your choice : 2

consumer consumes item 2
Enter your choice : 2

consumer consumes item 1
Enter your choice : 2
```

```
Buffer is empty!
Enter your choice : 1

Producer produces item 1
Enter your choice : 3
exiting!
```



19/06/24

program - 6

→ Write a C-program to simulate the concept of Dining-Philosophers problem

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

#define MAX_PHILOSOPHERS 10

typedef enum { THINKING, HUNGRY, EATING } state_t;

state_t states[MAX_PHILOSOPHERS];
int num_philosophers;
int num_hungry;
int hungry_philosophers[MAX_PHILOSOPHERS];
int forks[MAX_PHILOSOPHERS];

void print_state() {
    printf("\n");
    for (int i = 0; i < num_philosophers; i++) {
        if (states[i] == THINKING) printf("P %d is thinking
            \n", i+1);
        else if (states[i] == HUNGRY) printf("P %d is
            waiting\n", i+1);
        else if (states[i] == EATING) printf("P %d is
            eating", i+1);
    }
}

int can_eat(int philosopher_id) {
    int left_fork = philosopher_id;
    int right_fork = (philosopher_id+1) % num_
        philosophers;
```

```
    if (forks[left_fork] == 0 && forks[right_fork] == 0){
        forks[left_fork] = forks[right_fork] = 1;
        return 1;
    }
    return 0;
}
void simulate (int allow_two){
  int eating_count = 0;
  for(int i=0; i<num_hungry; ++i){
    int philosopher_id = hungry_philosophers[i];
    if (state[philosophers_id] == HUNGRY){
      if (can_eat(philosopher_id)){
          state[philosopher_id] = EATING;
          eating_count++;
        printf("P%d is granted to eat\n", philosopher_id+1);
        if(!allow)
        if (!allow_two && eating_count == 1) break;
        if (allow_two && eating_count==2) break;
      }
    }
  }
}

for(int i=0; i<num_hungry; ++i){
    int philosopher_id = hungry_philosophers[i];
    if (state[philosopher_id] == EATING){
        int left_fork = philosopher_id;
        int right_fork = (philosoper_id+1) % num_
                philosophers;
        forks[left_fork] = forks[right_fork] = 0;
        state[philosophers_id] = THINKING;
    }
  }
}
```

```
int main () {
  printf("Enter the total number of philosophers
          (max %d): ", max_philosophers);
  scanf("%d", num_philosophers);

  if (num_philospers < 2 || num_philosophers >
          MAX_PHILOSOPHERS);
  scanf("Invalid number of philosopers. Exiting \n");
    exit;
}

printf("How many are hungry:");
scanf("%d", num_hungry");

for (int i=0; i< num_hungry; i++) {
    printf("Enter the philosopher %d position:" i+1);
    int position;
    scanf("%d", &position);
    hungry_philosophers[i] = position-1;
    state[hungry_philosophers[i] = HUNGRY;
}

for(int I=0; i< num_philosophers; i++) {
     forks[i]=0;
}
int choice;
do {
     int _state();
     printf("\n1. one can eat at a time\n");
     printf("2. Two can eat at a time\n");
     printf("3. Exit\n");
     printf("Enter your choice");
     scanf("%d", &choice);
    switch(choice){
```

```
case 1:
    simulate(0);
    break;
case 2:
    simulate(1);
    break;
case 3:
    printf("exiting");
default:
    printf("Invalid choice");
    break;
}
} while (choice != 3);
return 0;
}
```

OUTPUT:

```
Enter total number of philosophers (max 10) : 5
How many are hungry : 3
Enter philosopher 1 position : 2
Enter philosopher 2 position : 4
Enter philosopher 3 position : 5

P1 is thinking
P2 is waiting
P3 is thinking
P4 is waiting
P5 is waiting

1. One can eat at a time  2. Two can eat at a time
3. Exit.
Enter your choice : 2
P2 is granted to eat
P4 is granted to eat
```

P1 is thinking
P2 is thinking
P3 is thinking
P4 is thinking
P5 is waiting

~~1. one~~

19/06/24

Program-7

⇒ Write a c -program to simulate Bankers Algorithm for the purpose of deadlock avoidance.

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

# define MAX_PROCESSES 10
# define MAX_RESOURCES 10

bool isSafeState (int processes, int resources, int available[],
  int max[][MAX_RESOURCES], int allocation[][MAX RESOURCES],
  int need[][MAX_RESOURSCES], int SafeSequence[]);
Void calculateNeed (int processes, int resources, int max[]
  [MAX_RESOURCES], int allocation[][MAX_RESOURCES], int need[]
  [MAX_RESOURCES]);

int main() {
      int processes, resources;
      int available [MAX_RESOURCES];
      int max [MAX_PROCESSES][MAX_RESOURCES];
      int allocations [MAX_PROCESSES][MAX_RESOURCES];
      int need [MAX_PROCESSES][MAX_RESOURCES];
      int SafeSequence [MAX_PROCESSES];

      printf("Enter the number of processes:");
      Scanf("%d", &processes);

      printf("Enter the number of resources:");
      Scanf("%d", & resources);

      printf(" Enter the available resources");
      for (int i=0; i< no resources; i++) {
          Scanf("%d", &available[i]);
      }
```

```
printf("Enter the maximum resource matrix:\n");
for (int i=0 ; i< processes ;i++) {
    printf("Enter the details for P %d\n", i);
    for (int j =0 ; j < resources ;j++) {
    scanf ("%d", &max[i][j]);
    }
}

printf("Enter the allocation resource matrix:\n");
for (int i=0; i< processes; i++){
    for (i-t j = 0 ;j< resources ;j++) {
        scanf("%d", &allocation[i][j]);
    }
}

calculateNeed(processes, resources, max, allocation,
    need);
if (issafeSate(processes, resources, available, max, allocati
    need, safesequence)){
    printf("system is in safe state\n");
    printf("The safe sequence is --(");
    for(int i=0; i< processes ;i++){
        printf("P%d", safesequence[i]);
    }
    printf(")\n");
} else {
    printf("system is not in safesbate\n");
}

printf("\n Processes \t Allocation \t\t Max \t Need \n");
for (int i=0; i< processes ; i++){
    printf("P%d \t", i);
    for (int j=0 ; j< resources ; j++) {
        printf("%d", allocation[i][j]);
    }
printf("\t");
```

```
        for (int j=0; j< resources; j++){
            printf(" %d", max[i][j]);
        }
        printf("\n");
        for(int j =0; j< resources; j++){
            printf(" %d", need[i][j]);
        }
        printf("\n");
    }
    return 0;
}

void CalculateNeed (int processes, int resources, int MAX[]
    [MAX_RESOURCES], int allocation[][MAX_RESOURCES], int
    need[][MAX_RESOURCES]){
        for (int i=0; i<processes ;i++){
            for(int i=0; j< resources; j++){
                need[i][j] = max[i][j] - allocation[i][j];
            }
        }
    }

bool isSafeSate( int processes, int resources, int available[]
    [MAX_RESOURCES], int allocation[][MAX_RESOURCES],
    int need[][MAX_RESOURCES], int safeSequence[]){
        int work[MAX_RESOURCES];
        bool finish[MAX_PROCESSES] = {false};
        int count=0;

    for (int i=0; i< resources; i++){
        work[i] = available[i];
    }
        while (count< processes){
        bool found = false;
```

```
for(int j=0 ;j< processes ;j++) {
    if (!finish[j]){
        bool canAllocate =true;
        for(int j =0; j< resources ; j++) {
            if (need [i][j] > work[j]) {
            canAllocate = false;
            break;
        }
    }

if (canAllocate) {
    for(int j=0; j <resources ; j++) {
        work[j] += allocation[i][j];
    }
    finish[i] = true;
    safeSequence [count++] = i;
    found =true;
    printf("P %d is visited (", i);
    for(int j=0 ;j< resources ; j++) {
        printf("%d", work[j]);
    }
    printf(")\n");
}
}
}

    if (!found) {
    for(int j=0; j< processes ;j++)
        {    if (!finish[j]) {
                return false;
        }
    }
    return false;
}
}
} return true;
```

OUTPUT:

```
Enter the number of processes : 5
Enter the number of resource : 3
Enter the available resource:
3 3 2
Enter the maximum resource matrix:
Enter the details for P0
 7 5 3
Enter the details for P1
3 2 2
Enter the details for P2
9 0 2
Enter the details for P3
2 2 2
Enter the details for P4
4 3 3
Enter the allocation resource matrix:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
P1 is visited ( 5 3 2)
P3 is visited (7 4 3)
P4 is visited (7 5 5)
P2 is visited (10 5 7
System is in safestate
The safe sequence is - (P1 P3 P4 P0 P2)
```

Process	Allocation	Max	Need
P0	0 1 0	7 5 3	7 4 3
P1	2 0 0	3 2 2	1 2 2
P2	3 0 2	9 0 2	6 0 0
P3	2 1 1	2 2 2	0 1 1
P4	0 0 2	4 3 3	4 3 1

3/07/24

PROGRAM - 8

→ Write a C program to simulate deadlock detection

```
#include <stdio.h>

void main()
{
    int n, m, i, j;
    printf("Enter the number of process and
        number of types of resources :\n");
    scanf("%d%d", &n, &m);
    int max[n][m], need[n][m], all[n][m], ava[m],
    flag = 1, finish[n], dead[n], c=0;
    printf("Enter the maximum number of each
      type of resources needed by each process:\n");
    for(int i=0; i<n; i++)
      {
        for(j=0; j<m; j++)
        {
            scanf("%d", max[i][j]);
        }
    }
    printf("Enter the allocated number of each type
        of resources needed by each process:\n");
    for(int i=0; i<n; i++) {
        for(j=0; j<m; j++) {
            scanf("%d", &all[i][j]);
        }
    }
    printf("Enter the available number of each type of
        rource:\n");
    for(j=0; j<m; j++){
        scanf("%d", &ava[j]);
}
```

```
        for(i=0; i<n; i++){
            for(j=0; j<m; j++){
                need[i][j] = max[i][j] - all[i][j];
            }
        }
        for(i=0; i<n; i++){
            finish[i] = 0;
        }
        while(flag){
            flag=0;
            for(int i=0; j<n; i++){
                c=0;
                for(j=0; j<m; j++)
                {
                    if(finish[i]==0 && need[i][j]<ava[j]){
                        c++;
                        if(c==m){
                            for(j=0; j<m; j++){
                                ava[j] += all[i][j];
                                finish[i] = 1;
                                flag = 1;
                            }
                            if(finish[i]=1){
                                i=n;
                            }
                        }
                    }
                }
            }
        }
        j=0;
        flag=0;
        for(i=0; i<n; i++){
            if(finish[i]==0){
```

```
            dead[j] = i;
            j++;
            flag = 1;
        }
    }
    if(flag == 1)
    {
        printf("Deadlock has occured:\n");
        printf("The deadlock process are:\n");
        for(i=0;i<n;i++) {
            printf("P%d", dead[i]);
        }
    }
    else
        printf("No deadlock has occured\n");
}
```

OUTPUT:

Enter the number of processes and number of types of resources:
5 3
Enter the maximum number of each type of resource needed by each process:
7 5 3
3 2 2
9 0 2
2 2 4
4 3 3

Enter the allocated number of each type of resource needed by each process:
0 1 0
2 0 0

3 0 2

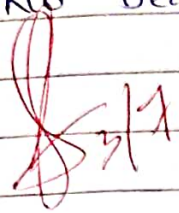
2 1 1

0 0 2

Enter the available number of each type of resource:

3 3 2

No Deadlock has occured.



03/07/24

PROGRAM-9

→ Write a C program to simulate the following contiguous memory allocation techniques
(a) Worst fit
(b) Best - fit
(c) First fit.

```
#include <stdio.h>

Struct Block {
    int block_no;
    int block_size;
    int is_free;
};

Struct File {
    int file_no;
    int file_size;
};

void firstFit( struct Block blocks[], int n_blocks,
    struct File files[], int n_files) {
    printf(" Memory management scheme - First Fit\n");
    printf(" File no : \t File size \t Block no \t Block size
        \t Fragment \n");

    for (int i=0; i<n_files ; i++) {
        for (int j =0; j< n_blocks ; j++) {
            if (blocks[j].is_free && blocks[j].block_size>=
                files[i].file_size) {
                blocks[j].is_free = 0;
                printf("%d \t\t %d \t\t %d \t\t %d \t\t %d\n",
                files[i].file_no, files[i].file_size,
                blocks[j].block_no, blocks[j].block_size
```

```
            blocks[j].blocksize - files[i].file_size);
            break;
        }
    }
}
}

void worstFit(struct Block blocks[], int n_blocks, struct
File files[], int n_files){
    printf("Mememory management scheme - worst Fit\n");
    printf("File No\t File Size\t Block no\t Block size\t
        Fragment:\n");

    for (int i=0; i<n_files; i++){
        int worst_fit_block = -1;
        int max_fragment = -1;
        for (int j=0; j<n_blocks; j++){
            if (blocks[j].is_free && blocks[j].block_size
            => files[i].file_size){
                int fragment = blocks[j].block_size - files[i].
                file_size;
                if(fragment > max_fragment)
                {   max_fragment = fragment;
                    worst_fit_block = j;
                }
            }
        }
        if (worst_fit_block != -1){
            blocks[worst_fit_block].is_free = 0;
            printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", files[i].
            file_no, files[i].file_size, blocks[worst_fit_
            block].block_no, blocks[worst_fit_block].block_
            size, max_fragment);
        }
    }
}
```

```
void bestFit(struct Block blocks[], int n_blocks,
struct File files[], int n_files) {
    printf("Memory management scheme - Best fit\n");

    for (int i=0; i< n_files ; i++) {
        int best_fit_block = -1;
        int min_fragment = 10000;
        for (int j=0 ;j< n_blocks ;j++) {
            if ( blocks[j].is_free && blocks[j].block_size
                >= files[i].file_size ) {
                int fragment = blocks[j].block_size -
                files[i].file_size;
                if (fragment < min_fragment) {
                    min_fragment = fragment;
                    best_fit_block = j;
                }
            }
        }

        if (best_fit_block != -1) {
            blocks[best_fit_block].is_free = 0;
            printf("%d\t\t%d\t\t%d\t\t%d\n", files[i].file_no
            files[i].file_size, blocks[best_fit_block].block_
            blocks[best_fit_block].block_size, min_
                fragment);
        }
    }
}

int main() {
    int n_blocks, n_files;
    printf("Enter the number of blocks:");
    scanf("%d", &n_blocks);
    printf("Enter the number of files:");
    scanf("%d", &n_files);
```

```
struct FileBlock blocks[n_blocks];
for(int i=0; i<n_blocks; i++){
    blocks[i].block_no = i+1;
    printf("Enter the size of block %d: ", i+1);
    scanf("%d", &blocks[i].block_size);
    blocks[i].is_free = 1;
}

struct File files[n_files];
for(int i=0; i<n_files; i++){
    files[i].file_no = i+1;
    printf("Enter the size of file %d: ", i+1);
    scanf("%d", &files[i].file_size);
}

firstFit(blocks, n_blocks, files, n_files);
printf("\n");

for(int i=0; i<n_blocks; i++){
    blocks[i].is_free = 1;
}

worstFit(blocks, n_blocks, files, n_files);
printf("\n");

for(int i=0; i<n_blocks; i++){
    blocks[i].is_free = 1;
}

bestFit(blocks, n_blocks, files, n_files);

return 0;
}
```

OUTPUT:

```
Enter the number of blocks : 3
Enter the number of files : 2
Enter the size of block 1 : 5
Enter the size of block 2 : 2
Enter the size of block 3 : 7
Enter the size of file 1 : 1
Enter the size of file 2 : 4
```

Memory management scheme - First Fit

File_no:	File_size:	Block_No:	Block_size:	Fra
1	1	1	5	4
2	4	3	7	3

Memory management scheme - Worst Fit

File_NO:	File_size	Block_no:	Block_Size:	Fragm
1	1	3	7	6
2	4	1	5	1

Memory management scheme - Best Fit

File NO:	File size:	Block_no:	Block_size:	Fragm
1	1	2	2	1
2	4	1	5	1

10/07/24

PROGRAM-10

Write a c program to simulate page replacement algorithms

(a) FIFO

(b) LRU

(c) Optimal

```
#include <stdio.h>
#include <limits.h>
#include <stdlib.h>

void fifo(int pages[], int n, int capacity){
    int frame[capacity], index=0, page_faults=0;
    for(int i=0; i<capacity; i++)
        frame[i]=-1;
    for(int i=0; i<n; i++){
        int found=0;
        for(int j=0; j<capacity; j++){
            if(frame[j]==pages[i]){
                found=1;
                break;
            }
        }
        if(!found){
            frame[index]=pages[i];
            index=(index+1)%capacity;
            page_faults++;
        }
        print_frames(frame, capacity, found?0:page_faults);
    }
    printf("Total Page faults using FIFO: %d\n\n", page_faults);
```

```
void print_frames (int frame[], int capacity, int
        page_faults) {
    for (int i=0; i< capacity ; i++) {
        if (frame[i] == -1) {
            printf("- ");
        }
        else {
            printf("%d ", frame[i]);
    }
    if (page_faults > 0)
        printf(" PF NO. %d", page_faults);
    printf("\n");
}

void lru( int pages[], int n, int capacity){
    int frame [capacity], counter [capacity], time=0,
        page_faults =0;
    for(int i = 0; i < capacity ;i++) {
        frame [i] = -1;
        counter [i] = 0;
}

printf (" LRU Page replacement processes : \n");
    for (int i=0; i<n ; i++) {
        int found =0;
        for (int j = 0; j < capacity ; j++) {
            if ( frame[j] == pages[i]) {
                found =1;
                counter [j] = time++;
                break;
            }
        }
        if (! found) {
            int min = INT_MAX , min_index = -1;
            for (int j =0; j< capacity ; j++) {
```

```
        if (counter[j] < min) {
            min = counter[j];
            min_index = j;
        }
    }
    frame[min_index] = pages[i];
    counter[min_index] = time++;
    page_faults++;
}
print_frames(frame, capacity, found ? 0 : page_faults);
}
printf("Total Page Faults using LRU : %d\n\n", page_faults);
}

void optimal(int pages[], int n, int capacity) {
    int frame[capacity], page_faults = 0;
    for(int i = 0; i < capacity; i++)
        frame[i] = -1;

    printf("Optimal Page Replacement Process :\n");
    for(int i = 0; i < n; i++) {
        int found = 0;
        for(int j = 0; j < capacity; j++) {
            if (frame[j] == pages[i]) {
                found = 1;
                break;
            }
        }
        if (!found) {
            int farthest = i + 1, index = -1;
            for(int j = 0; j < capacity; j++) {
                int k;
                for(k = i + 1; k < n; k++) {
                    if (frame[j] == pages[k])
                        break;
                }
```

```
            if (k > farthest) {
                farthest = k;
                index = j;
            }
        }
        if (index == -1) {
            for (int j = 0; j < capacity; j++) {
                if (frame[j] == -1) {
                    index = j;
                    break;
                }
            }
        }
        frame[index] = pages[i];
        page_faults++;
    }
    print_frames(frame, capacity, found ? 0 : page_faul
    }
    printf("Total Page faults using optimal : %d\n\n",
                page_faults);
}

int main() {
    int n, capacity;
    printf("Enter the number of pages:");
    scanf("%d", &n);
    int *pages = (int*) malloc(n * sizeof(int));
    printf("Enter the pages : ");
    for(int i = 0; i < n ; i++)
        scanf("%d", &pages[i]);
    printf("Enter the frame capacity:");
    scanf("%d", &capacity);

    printf("\n Pages:");
```

```
    for(int i=0; i<n ;i++)
        printf("%d", pages[i]);
    printf("\n\n");

    fifo(pages, n, capacity);
    lru(pages, n, capacity);
    optimal(pages, n, capacity);

    free(pages)
    return 0;
}
```

OUTPUT:

```
Enter the number of pages: 20
Enter the pages: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1
Enter the frame capacity: 3.

Pages: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

FIFO Page Replacement Process:
7 - - PF No. 1
7 0 - PF No. 2
7 0 1 PF No. 3
2 0 1 PF No. 4
2 0 1
2 3 1 PF No. 5
2 3 0 PF No. 6
4 3 0 PF No. 7
4 2 0 PF No. 8
4 2 3 PF No. 9
0 2 3 PF No. 10
0 2 3
0 2 3
```

0 1 3 PF NO. 11
0 1 2 PF NO. 12
0 1 2
0 1 2
7 1 2 PF NO. 13
7 0 2 PF NO. 14
7 0 1 PF NO. 15.

Total Page Faults using FIFO : 15

LRU Page replacement process:

7 - - PF NO. 1
0 - - PF NO. 2
0 1 - PF NO. 3
0 1 2 PF NO. 4
0 1 2
0 3 2 PF. NO 5
0 3 2
0 3 4 PF NO. 6
0 2 4 PF NO. 7
3 2 4 PF NO. 8
3 2 0 PF NO. 9
3 2 0
3 2 0
3 2 1 PF NO. 10
3 2 1
0 2 1 PF NO. 11
0 2 1
0 7 1 PF NO. 12
0 7 1
0 7 1

Total Page Faults using LRU = 12

optimal page replacement Process:

7 - - PF NO. 1
7 0 - PF NO. 2
7 0 1 PF NO. 3
2 0 1 PF NO. 4
2 0 1
2 0 3 PF NO. 5
2 0 3
2 4 3 PF NO. 6
2 4 3
2 4 3
2 0 3 PF NO. 7
2 0 3
2 0 3
2 0 1 PF NO. 8
2 0 1
2 0 1
2 0 1
7 0 1 PF NO. 9
7 0 1
7 0 1

Total Page Faults using optimal : 9

10/7
complete.