Write a C program to simulate FCFS CPU Scheduling algorithm.

Example:

PROCESS	BURST TIME
P1	25
P2	4
P3	3

The processes arrive in the order **P1, P2, P3** and are served as per the **FCFS algorithm**. The Gantt chart is as shown:

The waiting time for **P1** is **0 milliseconds**, for **P2** it is **25 milliseconds** and **29 milliseconds** for **P3**. Thus, average waiting time is **(0+25+29)/3 = 18 milliseconds**.

Algorithm:

```
1-  Input the processes along with their burst time (bt).
2-  Find waiting time (wt) for all processes.
3-  As first process that comes need not to wait so
    waiting time for process 1 will be 0 i.e. wt[0] = 0.
4-  Find waiting time for all other processes i.e. for
     process i ->
       wt[i] = bt[i-1] + wt[i-1] .
5-  Find turnaround time = waiting_time + burst_time
    for all processes.
6-  Find average waiting time = total_waiting_time / no_of_processes.
7-  Similarly, find average turnaround time =
                 total_turn_around_time / no_of_processes.
```

Source Code:

```
#include<stdio.h>
int main()
{
    int bt[10]={0},at[10]={0},tat[10]={0},wt[10]={0},ct[10]={0};
    int n,sum=0;
    float totalTAT=0,totalWT=0;
    printf("Enter number of processes      ");
    scanf("%d",&n);
    printf("Enter arrival time and burst time for each process\n\n");
    for(int i=0;i<n;i++)
    {
        printf("Arrival time of process[%d]    ",i+1);
        scanf("%d",&at[i]);
        printf("Burst time of process[%d]      ",i+1);
        scanf("%d",&bt[i]);
        printf("\n");
    }
    //calculate completion time of processes
    for(int j=0;j<n;j++)
    {
        sum+=bt[j];
        ct[j]+=sum;
```

```
	}
	//calculate turnaround time and waiting times
	for(int k=0;k<n;k++)
	{
		tat[k]=ct[k]-at[k];
		totalTAT+=tat[k];
	}
	for(int k=0;k<n;k++)
	{
		wt[k]=tat[k]-bt[k];
		totalWT+=wt[k];
	}
	printf("Solution: \n\n");
	printf("P#\t AT\t BT\t CT\t TAT\t WT\t\n\n");
	for(int i=0;i<n;i++)
	{
		printf("P%d\t %d\t %d\t %d\t %d\t %d\n",i+1,at[i],bt[i],ct[i],tat[i],wt[i]);
	}
	printf("\n\nAverage Turnaround Time = %f\n",totalTAT/n);
	printf("Average WT = %f\n\n",totalWT/n);
	return 0;
}
```

Output:

```
Arrival time of process[3]      0
Burst time of process[3]        3

Solution:

P#          AT          BT          CT          TAT         WT

P1          0           24          24          24          0
P2          0           3           27          27          24
P3          0           3           30          30          27


Average Turnaround Time = 27.000000
Average WT = 17.000000
```

Advantages:

- It is easy to understand and implement.

Dis-advantages:

- It is a **Non-Pre-emptive scheduling algorithm**: Once a process has been allocated the CPU, it will not release the CPU until it finishes executing. Thus, it is not suitable for modern systems which work on the principle of time sharing.
- The Average Waiting Time is high.
- It results in **CONVOY EFFECT** i.e., many processes which require CPU for short duration have to wait for a bigger process to finish thus resulting in low resource utilization.

Future Enhancement (Link to Next Program):

- To overcome the above dis-advantages we apply SJF, Round Robin, Priority CPU Scheduling algorithms.

Write a C program to simulate SJF CPU Scheduling algorithm.

Example:

```
Process ID      Arrival Time    Burst Time
1               2               3
2               0               4
3               4               2
4               5               4
Final Result...
Process ID      Arrival Time    Burst Time      Waiting Time    Turnaround
Time
2               0               4               0               4
3               4               2               0               2
1               2               3               4               7
4               5               4               4               8
```

Algorithm:

```
1- Traverse until all process gets completely
   executed.
   a) Find process with minimum remaining time at
     every single time lap.
   b) Reduce its time by 1.
   c) Check if its remaining time becomes 0
   d) Increment the counter of process completion.
   e) Completion time of current process =
     current_time +1;
   e) Calculate waiting time for each completed
     process.
   wt[i]= Completion time - arrival_time-burst_time
   f)Increment time lap by one.
```

2- Find turnaround time (waiting_time+burst_time).

Source Code:

```
#include<stdio.h>

void main()
{
	int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,pos,temp;
	float avg_wt,avg_tat;
	printf("Enter number of process:");
	scanf("%d",&n);
	printf("\nEnter Burst Time:\n");
	for(i=0;i<n;i++)
	{
		 printf("p%d:",i+1);
		scanf("%d",&bt[i]);
		p[i]=i+1;		//contains process number
	 }

   //sorting burst time in ascending order using selection sort
	for(i=0;i<n;i++)
	{
		pos=i;
		for(j=i+1;j<n;j++)
		{
			if(bt[j]<bt[pos])
			pos=j;
		}

		temp=bt[i];
		bt[i]=bt[pos];
		bt[pos]=temp;
		temp=p[i];
		p[i]=p[pos];
		p[pos]=temp;
	}

	wt[0]=0;		//waiting time for first process will be zero
```

```
        //calculate waiting time
        for(i=1;i<n;i++)
        {
                wt[i]=0;
                for(j=0;j<i;j++)
                wt[i]+=bt[j];
                total+=wt[i];
        }

        avg_wt=(float)total/n;      //average waiting time
        total=0;

        printf("\nProcess\t    Burst Time    \tWaiting Time\tTurnaround Time");
        for(i=0;i<n;i++)
        {
                tat[i]=bt[i]+wt[i];     //calculate turnaround time
                total+=tat[i];
                printf("\np%d\t\t  %d\t\t    %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);
        }

        avg_tat=(float)total/n;     //average turnaround time
        printf("\n\nAverage Waiting Time=%f",avg_wt);
        printf("\nAverage Turnaround Time=%f\n",avg_tat);
}
```

Output:

```
Enter number of process:4

Enter Burst Time:
p1:4
p2:8
p3:3
p4:7

Process      Burst Time          Waiting Time    Turnaround Time
p3                3                  0                   3
p1                4                  3                   7
p4                7                  7                   14
p2                8                  14                  22

Average Waiting Time=6.000000
Average Turnaround Time=11.500000

Process returned 35 (0x23)   execution time : 5.567 s
Press any key to continue.
```

Advantages:

- Short processes are handled very quickly.
- The system also requires very little overhead since it only makes a decision when a process completes or a new process is added.
- When a new process is added the algorithm only needs to compare the currently executing process with the new process, ignoring all other processes currently waiting to execute.

Disadvantage:

- Like shortest job first, it has the potential for process starvation.
- Long processes may be held off indefinitely if short processes are continually added.

Write a C program to simulate Round Robin CPU Scheduling algorithm.

Example:

Time Quantum = 2

Process	Arrival Time	Burst Time
P1	0	9
P2	1	5
P3	2	3
P4	3	4

Process	Arrival Time	Burst Time (x)	Turnaround Time(t)	Normalized Turnaround Time(t/x)	Waiting Time
P1	0	9	21	2.34	12
P2	1	5	17	3.4	12
P3	2	3	11	3.67	8
P4	3	4	12	3	8

Average turnaround time=15.25

Average Waiting time=10

Algorithm:

```
1- Create an array rem_bt[] to keep track of remaining
   burst time of processes. This array is initially a
   copy of bt[] (burst times array)
2- Create another array wt[] to store waiting times
   of processes. Initialize this array as 0.
3- Initialize time : t = 0
4- Keep traversing the all processes while all processes
   are not done. Do following for i'th process if it is
   not done yet.
    a- If rem_bt[i] > quantum
       (i)  t = t + quantum
       (ii) bt_rem[i] -= quantum;
    c- Else // Last cycle for this process
       (i)  t = t + bt_rem[i];
       (ii) wt[i] = t - bt[i]
       (ii) bt_rem[i] = 0; // This process is over
```

Source Code:

```
#include<stdio.h>

int main()
{

        int count,j,n,time,remain,flag=0,time_quantum;
        int wait_time=0,turnaround_time=0,at[10],bt[10],rt[10];
        printf("Enter Total Process:\t ");
        scanf("%d",&n);
        remain=n;
        for(count=0;count<n;count++)
        {
                printf("Enter Arrival Time and Burst Time for Process Process Number %d
                :",count+1);
                scanf("%d",&at[count]);
                scanf("%d",&bt[count]);
                rt[count]=bt[count];
        }
        printf("Enter Time Quantum:\t");
        scanf("%d",&time_quantum);
        printf("\n\nProcess\t|Turnaround Time|Waiting Time\n\n");
```

```
        for(time=0,count=0;remain!=0;)
        {
                if(rt[count]<=time_quantum && rt[count]>0)
                {
                        time+=rt[count];
                        rt[count]=0;
                        flag=1;
                }
        else if(rt[count]>0)
                {
                        rt[count]-=time_quantum;
                        time+=time_quantum;
                }
        if(rt[count]==0 && flag==1)
        {
                remain--;
                printf("P[%d]\t|\t%d\t|\t%d\n",count+1,time-at[count],time-at[count]-bt[count]);
                wait_time+=time-at[count]-bt[count];
                turnaround_time+=time-at[count];
                flag=0;
        }
        if(count==n-1)
                count=0;
        else if(at[count+1]<=time)
                count++;
        else
                count=0;
        }
         printf("\nAverage Waiting Time= %f\n",wait_time*1.0/n);
        printf("Avg Turnaround Time = %f",turnaround_time*1.0/n);

        return 0;
}
```

Output:

```
Average Waiting Time= 5.250000
Avg Turnaround Time = 9.500000tusharsoni@tusharsoni-Lenovo-G50-70:~/Desktop$ ./a
.out
Enter Total Process:     4
Enter Arrival Time and Burst Time for Process Process Number 1 :0
9
Enter Arrival Time and Burst Time for Process Process Number 2 :1
5
Enter Arrival Time and Burst Time for Process Process Number 3 :2
3
Enter Arrival Time and Burst Time for Process Process Number 4 :3
4
Enter Time Quantum:     5


Process |Turnaround time|waiting time

P[2]    |       9       |       4
P[3]    |       11      |       8
P[4]    |       14      |       10
P[1]    |       21      |       12

Average Waiting Time= 8.500000
Avg Turnaround Time = 13.750000tusharsoni@tusharsoni-Lenovo-G50-70:~/Desktop$
```

Advantages

- It is simple.
- It is easy to implement
- It deals with all process without any priority.
- In this, all jobs get easily allocated to CPU.
- Like first come first serve scheduling, in this no problem of convoy effect or starvation is there.
- Round robin scheduling does not depend upon burst time. So it can be easily implementable on the system.

Disadvantages

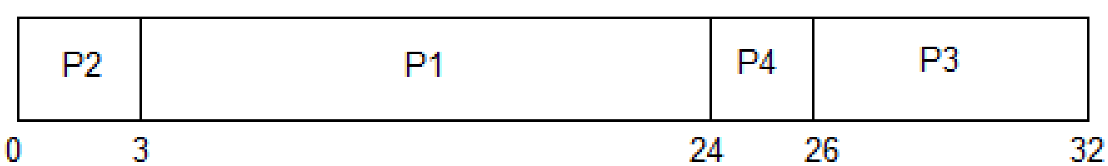
- Since round robin scheduling depends upon time quantum. So deciding a perfect time quantum for scheduling is a very difficult task.
- If the time quantum is higher, then the response time of the system will also be higher.
- If the time quantum is lower, then there is higher context switching overhead.

Write a C program to simulate Priority CPU Scheduling algorithm.

Example:

PROCESS	BURST TIME	PRIORITY
P1	21	2
P2	3	1
P3	6	4
P4	2	3

The GANTT chart for following processes based on Priority scheduling will be,



The average waiting time will be, (0 + 3 + 24 + 26)/4 = 13.25 ms

Algorithm:

```
1- First input the processes with their burst time
   and priority.
2- Sort the processes, burst time and priority
   according to the priority.
3- Now simply apply FCFS algorithm.
```

Source Code:

```c
#include<stdio.h>

int main()
{
	int bt[20],p[20],wt[20],tat[20],pr[20],i,j,n,total=0,pos,temp,avg_wt,avg_tat;
	printf("Enter Total Number of Process:");
	scanf("%d",&n);

	printf("\nEnter Burst Time and Priority\n");
	for(i=0;i<n;i++)
	{
		printf("\nP[%d]\n",i+1);
		printf("Burst Time:");
		scanf("%d",&bt[i]);
		printf("Priority:");
		scanf("%d",&pr[i]);
		p[i]=i+1;	//contains process number
	}

   //sorting burst time, priority and process number in ascending order using selection sort
	for(i=0;i<n;i++)
	{
		pos=i;
		for(j=i+1;j<n;j++)
		{
			 if(pr[j]<pr[pos])
			pos=j;
		}

		temp=pr[i];
```

```
            pr[i]=pr[pos];
            pr[pos]=temp;

            temp=bt[i];
            bt[i]=bt[pos];
            bt[pos]=temp;

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

    wt[0]=0;        //waiting time for first process is zero

    //calculate waiting time
    for(i=1;i<n;i++)
    {
            wt[i]=0;
            for(j=0;j<i;j++)
                    wt[i]+=bt[j];

                    total+=wt[i];
    }

    avg_wt=total/n;     //average waiting time
    total=0;

    printf("\nProcess\t    Burst Time    \tWaiting Time\tTurnaround Time");
    for(i=0;i<n;i++)
    {
            tat[i]=bt[i]+wt[i];     //calculate turnaround time
            total+=tat[i];
            printf("\nP[%d]\t\t  %d\t\t    %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);
    }

    avg_tat=total/n;     //average turnaround time
    printf("\n\nAverage Waiting Time=%d",avg_wt);
    printf("\nAverage Turnaround Time=%d\n",avg_tat);

    return 0;
}
```

Output:

```
Enter Total Number of Process:4

Enter Burst Time and Priority

P[1]
Burst Time:6
Priority:3

P[2]
Burst Time:2
Priority:2

P[3]
Burst Time:14
Priority:1

P[4]
Burst Time:6
Priority:4

Process      Burst Time            Waiting Time     Turnaround Time
P[3]               14                   0                   14
P[2]               2                    14                  16
P[1]               6                    16                  22
P[4]               6                    22                  28

Average Waiting Time=13
Average Turnaround Time=20
```

Advantages:

- The priority of process is selected on the basis of memory requirement, user preference or the requirement of time.
- Processes are executed on the basis of priority. So high priority does not need to wait for long which saves time.
- It is easy to use.
- It is a user friendly algorithm.
- Simple to understand.
- it has reasonable support for priority.

Dis-advantages:

- The major disadvantage of priority scheduling is the process of indefinite blocking or starvation. This problem appears when a process is ready to be executed but it has to wait for the long time for execution by CPU because other high priority processes are executed by the CPU.
- The problem of starvation can be solved by aging. Aging is a technique in which the system gradually increases the priority of those processes which are waiting in the system from a long time for their execution.
- In case if we have the processes which have the same priority, then we have to make use of FCFS scheduling algorithm.
- If the system gets crashes eventually, then all the processes having low priority which are not finished yet, also get lost.

Write programs using the I/O system calls of UNIX/LINUX operating system

```
// C program to illustrate open system call
	#include<stdio.h>
	#include<fcntl.h>
	#include<errno.h>
	externinterrno;
	intmain()
	{
		// if file does not have in directory
		// then file foo.txt is created.
		intfd = open("foo.txt", O_RDONLY | O_CREAT);

		printf("fd = %d/n", fd);

		if(fd ==-1)
		{
		// print which type of error have in a code
		printf("Error Number % d\n", errno);

		// print program detail "Success or failure"
		perror("Program");
		}
		return0;
	}
```

Output:

```
fd = 3
```

```
// C program to illustrate close system Call
          #include<stdio.h>
          #include <fcntl.h>
          intmain()
          {
                    intfd1 = open("foo.txt", O_RDONLY);
                     if(fd1 < 0)
                    {
                              perror("c1");
                              exit(1);
                    }
                    printf("opened the fd = % d\n", fd1);

                    // Using close system Call
                    if(close(fd1) < 0)
                    {
                              perror("c1");
                              exit(1);
                    }
                    printf("closed the fd.\n");
          }
```

Output:

```
opened the fd = 3

closed the fd.
```

```
// C program to illustrate read system Call
        #include<stdio.h>
        #include <fcntl.h>
        intmain()
        {
                intfd, sz;
                char*c = (char*) calloc(100, sizeof(char));

                fd = open("foo.txt", O_RDONLY);
                 if(fd < 0) { perror("r1"); exit(1); }

                sz = read(fd, c, 10);
                printf("called read(% d, c, 10).  returned that"" %d bytes  were read.\n", fd, sz);
                c[sz] = '\0';
                printf("Those bytes are as follows: % s\n", c);
        }
```

Output:

```
called read(3, c, 10).  returned that 10 bytes  were read.

Those bytes are as follows: 0 0 0 foo.
```

```
// C program to illustrate write system Call
        #include<stdio.h>
        #include <fcntl.h>
        main()
        {
                intsz;

                intfd = open("foo.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
                if(fd < 0)
                {
                        perror("r1");
                        exit(1);
                }

                sz = write(fd, "hello geeks\n", strlen("hello geeks\n"));

                printf("called write(% d, \"hello geeks\\n\", %d)."" It returned %d\n", fd,
strlen("hello geeks\n"), sz);

                close(fd);
        }
```

Output:

```
called write(3, "hello geeks\n", 12).  it returned 11
```

Write a C program to simulate Bankers Algorithm for Deadlock Avoidance and Prevention.

Safety Algorithm
The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1. Let Work and Finish be vectors of length 'm' and 'n' respectively.
   Initialize: Work= Available
   Finish [i]=false; for i=1,2,……,n

2. Find an i such that both
   a) Finish [i]=false
   b) Need_i<=work

   if no such i exists goto step (4)
3. Work=Work + Allocation_i
   Finish[i]= true
   goto step(2)

4. If Finish[i]=true for all i,
   then the system is in safe state.

Safe sequence is the sequence in which the processes can be safely executed.

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
		int Max[10][10], need[10][10], alloc[10][10], avail[10], completed[10],
safeSequence[10];
		int p, r, i, j, process, count;
		 count = 0;

		printf("Enter the no of processes : ");
		scanf("%d", &p);

		for(i = 0; i< p; i++)
				completed[i] = 0;

		printf("\n\nEnter the no of resources : ");
```

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

    printf("\n\nEnter the Max Matrix for each process : ");
    for(i = 0; i < p; i++)
    {
            printf("\nFor process %d : ", i + 1);
            for(j = 0; j < r; j++)
                    scanf("%d", &Max[i][j]);
    }
    printf("\n\nEnter the allocation for each process : ");
    for(i = 0; i < p; i++)
    {
            printf("\nFor process %d : ",i + 1);
            for(j = 0; j < r; j++)
            scanf("%d", &alloc[i][j]);
    }

    printf("\n\nEnter the Available Resources : ");
    for(i = 0; i < r; i++)
            scanf("%d", &avail[i]);

    for(i = 0; i < p; i++)

            for(j = 0; j < r; j++)
                    need[i][j] = Max[i][j] - alloc[i][j];
    do
    {
            printf("\n Max matrix:\tAllocation matrix:\n");

            for(i = 0; i < p; i++)
            {
            for( j = 0; j < r; j++)
            printf("%d ", Max[i][j]);
            printf("\t\t");
            for( j = 0; j < r; j++)
                    printf("%d ", alloc[i][j]);
            printf("\n");
            }

  process = -1;
    for(i = 0; i < p; i++)
    {
             if(completed[i] == 0)//if not completed
```

```
            {
                        process = i ;
            for(j = 0; j < r; j++)
            {
                        if(avail[j] < need[i][j])
                        {
                                    process = -1;
                                    break;
                        }
            }
             }
             if(process != -1)
                break;
          }
if(process != -1)
          {
            printf("\nProcess %d runs to completion!", process + 1);
            safeSequence[count] = process + 1;
            count++;
            for(j = 0; j < r; j++)
            {
                        avail[j] += alloc[process][j];
                        alloc[process][j] = 0;
                        Max[process][j] = 0;
                        completed[process] = 1;
            }
            }
       }
       while(count != p && process != -1);
            if(count == p)
            {
                        printf("\nThe system is in a safe state!!\n");
                        printf("Safe Sequence : < ");
                        for( i = 0; i < p; i++)
                        printf("%d ", safeSequence[i]);
                        printf(">\n");
            }
       else
             printf("\nThe system is in an unsafe state!!");

}
```

Output:

```
Enter the no of processes : 5

Enter the no of resources : 3

Enter the MaxMatrixfor each process :
For process 1 : 7
5
3

For process 2 : 3
2
2

For process 3 : 7
0
2

For process 4 : 2
2
2

For process 5 : 4
3
3
Enter the allocation for each process :
For process 1 : 0
1
0

For process 2 : 2
0
0

For process 3 : 3
0
2

For process 4 : 2
1
1
```

```
For process 5 : 0
0
2
Enter the AvailableResources : 3
3
2

Max matrix:    Allocation matrix:
753010
322200
702302
222211
433002

Process2 runs to completion!
Max matrix:    Allocation matrix:
753010
000000
702302
222211
433002

Process3 runs to completion!
Max matrix:    Allocation matrix:
753010
000000
000000
222211
433002
Process4 runs to completion!
Max matrix:    Allocation matrix:
753010
000000
000000
000000
433002

Process1 runs to completion!
Max matrix:    Allocation matrix:
000000
000000
000000
000000
```

```
433002

Process5 runs to completion!
The system is in a safe state!!
SafeSequence : <23415>
```

Advantages:

- Allows mutual-exclusion, hold-and-wait, and no pre-emption conditions
- System guarantees that processes will be allocated resources within finite time.

Dis-advantages:

- It requires the number of processes to be fixed; no additional processes can start while it is executing.
- It requires that the number of resources remain fixed; no resource may go down for any reason without the possibility of deadlock occurring.
- It allows all requests to be granted in finite time, but one year is a finite amount of time.
- Similarly, all of the processes guarantee that the resources loaned to them will be repaid in a finite amount of time. While this prevents absolute starvation, some pretty hungry processes might develop.
- All processes must know and state their maximum resource need in advance.

Write a C program to implement the Producer – Consumer problem using semaphores using UNIX/LINUX system calls

Producer pseudo-code

```
voidproducer()
{
while(T)
     {
        produce()
        wait(E)
        wait(S)
        append()
        signal(S)
        signal(F)
    }
}
```

Producer pseudo-code

```
voidconsumer()
{
while(T)
     {
        wait(F)
        wait(S)
        take()
        signal(S)
        signal(E)
        use()
    }
}
```

Source Code:

```
#include<stdio.h>
int mutex=1,full=0,empty=3,x=0;
main()
{
	int n;
	void producer();
	void consumer();
	int wait(int);
	int signal(int);
	 printf("\n 1.Producer  \n 2.Consumer \n 3.Exit");
	while(1)
	{
		printf("\n Enter 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;
		}
	}
}
```

```
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 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

Buffer is full

Enter your choice:2

Consumer 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:3

Write C programs to simulate the Paging techniques of memory management

Source Code:

```
#include<stdio.h>
#define MAX 50
int main()
{
        int page[MAX],i,n,f,ps,off,pno;
        int choice=0;
        printf("\nEnter the no of  peges in memory: ");
        scanf("%d",&n);
        printf("\nEnter page size: ");
        scanf("%d",&ps);
        printf("\nEnter no of frames: ");
        scanf("%d",&f);
        for(i=0;i<n;i++)
                page[i]=-1;
        printf("\nEnter the page table\n");
        printf("(Enter frame no as -1 if that page is not present in any frame)\n\n");
        printf("\npageno\tframeno\n-------\t-------");
        for(i=0;i<n;i++)
        {
                printf("\n\n%d\t\t",i);
                scanf("%d",&page[i]);
        }
do
        {
                printf("\n\nEnter the logical address(i.e,page no & offset):");
                scanf("%d%d",&pno,&off);
```

```
                    if(page[pno]==-1)
                              printf("\n\nThe required page is not available in any of frames");
                    else
                              printf("\n\nPhysical address(i.e,frame no &
offset):%d,%d",page[pno],off);
                    printf("\nDo you want to continue(1/0)?:");
                    scanf("%d",&choice);
          }while(choice==1);
          return 1;
}
```

Output:

```
Enter the no of  peges in memory: 4

Enter page size: 10

Enter no of frames: 10

Enter the page table
(Enter frame no as -1 if that page is not present in any frame)


pageno   frameno
------- -------

0                -1


1                8


2                -1


3                6


Enter the logical address(i.e,page no & offset):2 200


The required page is not available in any of frames
Do you want to continue(1/0)?:1


Enter the logical address(i.e,page no & offset):1 500


Physical address(i.e,frame no & offset):8,500
Do you want to continue(1/0)?:
```

Advantages:

- On the programmer level, paging is a transparent function and does not require intervention.
- No external fragmentation.
- No internal fragmentation on updated OS's.
- Frames do not have to be contiguous.

Dis-advantages:

- Paging causes internal fragmentation on older systems.
- Longer memory lookup times than segmentation; remedy with TLB memory caches.

Write C programs to simulate the Segmentation techniques of memory management

Source Code:

```
#include<stdio.h>
#include<conio.h>
struct list
{
        int seg;
        int base;
        int limit;
        struct list *next;
} *p;
void insert(struct list *q,int base,int limit,int seg)
{
       if(p==NULL)
        {
                p=malloc(sizeof(Struct list));
                p->limit=limit;
                p->base=base;
                p->seg=seg;
                p->next=NULL;
        }
        else
        {
                while(q->next!=NULL)
                {
                        Q=q->next;
                        printf("yes")
                }
```

```
            q->next=malloc(sizeof(Struct list));
            q->next ->limit=limit;
            q->next ->base=base;
            q->next ->seg=seg;
            q->next ->next=NULL;
            }
}
int find(struct list *q,int seg)
{
        while(q->seg!=seg)
            {
                    q=q->next;
            }
            return q->limit;
}
int search(struct list *q,int seg)
{
            while(q->seg!=seg)
            {
                    q=q->next;
            }
            return q->base;
}
main()
{
            p=NULL;
            int seg,offset,limit,base,c,s,physical;
            printf("Enter segment table/n");
            printf("Enter -1 as segment value for termination\n");
```

```
do
{
        printf("Enter segment number");
        scanf("%d",&seg);
        if(seg!=-1)
        {
                printf("Enter base value:");
                scanf("%d",&base);
                printf("Enter value for limit:");
                scanf("%d",&limit);
                insert(p,base,lmit,seg);
        }
}
while(seg!=-1)
printf("Enter offset:");
scanf("%d",&offset);
printf("Enter bsegmentation number:");
scanf("%d",&seg);
c=find(p,seg);
s=search(p,seg);
if(offset<c)
{
        physical=s+offset;
        printf("Address in physical memory %d\n",physical);
}
else
{
        printf("error");
```

```
        }
    }
```

Output:

Enter segment table

Enter -1 as segmentation value for termination

Enter segment number:1

Enter base value:2000

Enter value for limit:100

Enter segment number:2

Enter base value:2500

Enter value for limit:100

Enter segmentation number:-1

Enter offset:90

Enter segment number:2

Address in physical memory 2590

Advantages:

- No internal fragmentation
- Average Segment Size is larger than the actual page size.
- Less overhead
- It is easier to relocate segments than entire address space.
- The segment table is of lesser size as compare to the page table in paging.

Dis-advantages:

- It can have external fragmentation.
- It is difficult to allocate contiguous memory to variable sized partition.
- Costly memory management algorithms.