OPERATING SYSTEMS

(BCS303)

LAB PROGRAMS

Integrated Lab Programs:

Sl. No.	Programs	COs
1.	Implement the process system calls fork (), exec () and wait ()	CO1
2.	a) Implement FCFS (First Come First Serve) scheduling algorithm b) Implement SJF (Shortest Job First) scheduling algorithm.	CO1
3.	Implement Priority Scheduling algorithm.	CO1
4.	Implement Round Robin Scheduling algorithm.	CO1
5.	Write a program to implement Producer-Consumer Problem using semaphores.	CO1
6.	Implement Memory Allocation methods for Fixed partition using First Fit, Worst Fit, Best Fit	CO2
7.	Write a program to implement FIFO page replacement algorithm.	CO3
8.	Write a program to implement LRU page replacement algorithm.	CO3
9.	Write a program to detect Deadlock.	CO4
10.	Write a program to implement Banker’s algorithm for Deadlock avoidance.	CO4

1. Implement the process system calls fork (), exec () and wait ()

fork() system call

- Fork system call use for creates a new process, which is called ***child process***, which runs concurrently with the process (which process called system call fork) and this process is called ***parent process***.
- After a new child process created, both processes will execute the next instruction following the fork() system call.
- A child process uses the same pc(program counter), same CPU registers, same open files use in the parent process.
- It takes no parameters and returns an integer value. Below are different values returned by fork().

Negative Value: the creation of a child process was unsuccessful. ***Zero***: Returned to the newly created child process.
Positive value: Returned to parent or caller. The value contains the process ID of a newly created child process.

program:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
   // make two process which run same
  // program after this instruction
  fork();
   printf("Hello world!\n");
  return 0;
}
```

Output:

```
Hello world!
Hello world!
```

Wait System Call in C

A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After the child process terminates, the parent continues its execution after wait system call instruction.

The child process may terminate due to any of these:It calls exit();

- It returns (an int) from main
- It receives a signal (from the OS or another process) whose default action is to terminate.
- If any process has more than one child process, then after calling wait(), the parent process has to be in a wait state if no child terminates.
- If only one child process is terminated, then return a wait() returns the process ID of the terminated child process.
- If more than one child processes are terminated than wait() reap any arbitrarily child and return a process ID of that child process.
- When wait() returns they also define exit status (which tells our, a process why terminated) via a pointer, If status are not **NULL**.

If any process has no child process then wait() returns immediately "-1".

Program:

```
// C program to demonstrate working of wait()
#include<stdio.h>
#include<sys/wait.h>
#include<unistd.h>
int main()
{
   if (fork()== 0)
      printf("HC: hello from child\n");
   else
   {
      printf("HP: hello from parent\n");
      wait(NULL);
      printf("CT: child has terminated\n");
   }
     printf("Bye\n");
   return 0;
}
```

Output: (Depend on Environment)

```
HC: hello from child
HP: hello from parent
```

```
CT: child has terminated
     (or)
HP: hello from parent
HC: hello from child
CT: child has terminated     // this sentence does
                              // not print before HC
                              // because of wait.
```

<u>exec() system call:</u>

- The exec family of functions replaces the currently running process with a new process.
- It can be used to run a C program by using another C program.
- It comes under the header file **unistd.h.**

Program:

```
// parent.c: the parent program
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>

int main (int argc, char **argv)
{
   int i = 0;
   long sum;
   int pid;
   int status, ret;

   printf ("Parent: Hello, World!\n");

   pid = fork ();

   if (pid == 0) {

      // I am the child

      execvp ("./child", NULL);
   }

   // I am the parent

   printf ("Parent: Waiting for Child to complete.\n");

   if ((ret = waitpid (pid, &status, 0)) == -1)
       printf ("parent:error\n");

   if (ret == pid)
      printf ("Parent: Child process waited for.\n");
}
```

Output:

```
gcc parent.c -o parent   //compiling parent.c
./parent                 // running parent.c
```

2 a) Implement FCFS (First Come First Serve) scheduling algorithm

FCFS Scheduling algorithm.

We are given with the n number of processes i.e. P1, P2, P3, ..., Pn and their corresponding burst times. The task is to find the average waiting time and average turnaround time using **FCFS CPU Scheduling algorithm**.

What is Waiting Time and Turnaround Time?

- Turnaround Time is the time interval between the submission of a process and its completion.
  Turnaround Time = completion of a process – submission of a process
- Waiting Time is the difference between turnaround time and burst time
  Waiting Time = turnaround time – burst time

What is FCFS Scheduling?

First Come, First Served (FCFS) also known as First In, First Out (FIFO) is the CPU scheduling algorithm in which the CPU is allocated to the processes in the order they are queued in the ready queue.

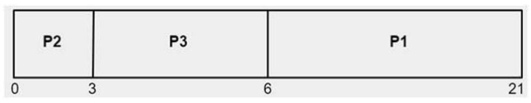
FCFS follows non-preemptive scheduling which mean once the CPU is allocated to a process it does not leave the CPU until the process will not get terminated or may get halted due to some I/O interrupt.

Example

Let’s say, there are four processes arriving in the sequence as P2, P3, P1 with their corresponding execution time as shown in the table below. Also, taking their arrival time to be 0.

Process	Order of arrival	Execution time in msec
P1	3	15
P2	1	3
P3	2	3

Gantt chart showing the waiting time of processes P1, P2 and P3 in the system



As shown above,

The waiting time of process P2 is 0

The waiting time of process P3 is 3

The waiting time of process P1 is 6

Average time = (0 + 3 + 6) / 3 = 3 msec.

As we have taken arrival time to be 0 therefore turn around time and completion time will be same.

Example

```
Input-:  processes = P1, P2, P3
        Burst time = 5, 8, 12
Output-:
Processes  Burst     Waiting     Turn around
1          5         0           5
2          8         5           13
3          12        13          25
Average Waiting time = 6.000000
Average turn around time = 14.333333
```

Algorithm

```
Start                                                    Step 1-> In function int
waitingtime(int proc[], int n, int burst_time[], int wait_time[])
   Set wait_time[0] = 0
   Loop For i = 1 and i < n and i++
      Set wait_time[i] = burst_time[i-1] + wait_time[i-1]
   End For
Step 2-> In function int turnaroundtime( int proc[], int n, int burst_time[], int wait_time[], int
tat[])
   Loop For  i = 0 and i < n and i++
      Set tat[i] = burst_time[i] + wait_time[i]
   End For
Step 3-> In function int avgtime( int proc[], int n, int burst_time[])
   Declare and initialize wait_time[n], tat[n], total_wt = 0, total_tat = 0;
   Call waitingtime(proc, n, burst_time, wait_time)
   Call turnaroundtime(proc, n, burst_time, wait_time, tat)
   Loop For  i=0 and i<n and i++
      Set total_wt = total_wt + wait_time[i]
      Set total_tat = total_tat + tat[i]
      Print process number, burstime wait time and turnaround time
   End For
   Print "Average waiting time =i.e. total_wt / n
   Print "Average turn around time = i.e. total_tat / n
Step 4-> In int main()
   Declare the input int proc[] = { 1, 2, 3}
   Declare and initialize n = sizeof proc / sizeof proc[0]
   Declare and initialize burst_time[] = {10, 5, 8}
   Call avgtime(proc, n, burst_time)
Stop
```

Example

```
#include <stdio.h>
// Function to find the waiting time for all processes
int waitingtime(int proc[], int n,
int burst_time[], int wait_time[]) {
   // waiting time for first process is 0
   wait_time[0] = 0;
   // calculating waiting time
   for (int i = 1; i < n ; i++ )
   wait_time[i] = burst_time[i-1] + wait_time[i-1] ;
   return 0;
}
// Function to calculate turn around time
int turnaroundtime( int proc[], int n,
 int burst_time[], int wait_time[], int tat[]) {
   // calculating turnaround time by adding
   // burst_time[i] + wait_time[i]
   int i;
   for ( i = 0; i < n ; i++)
   tat[i] = burst_time[i] + wait_time[i];
   return 0;
}
//Function to calculate average time
int avgtime( int proc[], int n, int burst_time[]) {
   int wait_time[n], tat[n], total_wt = 0, total_tat = 0;
   int i;
   //Function to find waiting time of all processes
   waitingtime(proc, n, burst_time, wait_time);
   //Function to find turn around time for all processes
   turnaroundtime(proc, n, burst_time, wait_time, tat);
   //Display processes along with all details
   printf("Processes  Burst   Waiting Turn around
");
   // Calculate total waiting time and total turn
   // around time
   for ( i=0; i<n; i++) {
      total_wt = total_wt + wait_time[i];
      total_tat = total_tat + tat[i];
      printf(" %d\t  %d\t\t %d \t%d
", i+1, burst_time[i], wait_time[i], tat[i]);
   }
   printf("Average waiting time = %f
", (float)total_wt / (float)n);
   printf("Average turn around time = %f
", (float)total_tat / (float)n);
   return 0;
}
// main function
int main() {
   //process id's
   int proc[] = { 1, 2, 3};
   int n = sizeof proc / sizeof proc[0];
   //Burst time of all processes
   int burst_time[] = {5, 8, 12};
   avgtime(proc, n, burst_time);
   return 0;
}
```

Output

```
Processes  Burst     Waitin g    Turn around
1          5        0          5
2          8        5          13
3          12       13          25
Average Waiting time = 6.000000
Average turn around time = 14.333333
```

2. Implement SJF (Shortest Job First) scheduling algorithm. CO1

Shortest job first(SJF) is a scheduling algorithm, that is used to schedule processes in an operating system. It is a very important topic in Scheduling when compared to round-robin and FCFS Scheduling. In this article, we will discuss the Shortest Job First Scheduling in the following order:

- Types of SJF
- Non-Preemptive SJF
- Code for Non-Preemptive SJF Scheduling
- Code for Pre-emptive SJF Scheduling

There are two types of SJF

- Pre-emptive SJF
- Non-Preemptive SJF

These algorithms schedule processes in the order in which the shortest job is done first. It has a minimum average waiting time.

There are 3 factors to consider while solving SJF, they are

1. BURST Time
2. Average waiting time
3. Average turnaround time

Non-Preemptive Shortest Job First

Here is an example

Processes Id	Burst Time	Waiting Time	Turn Ar
4	3	0	
1	6	3	
3	7	9	
2	8	16	

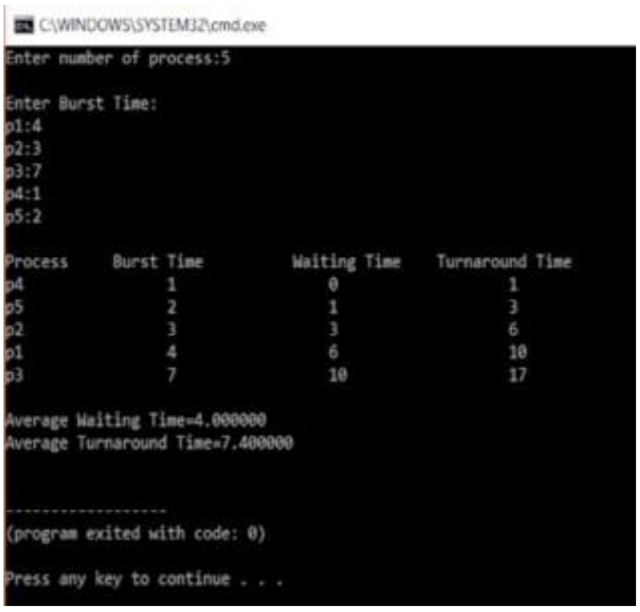
Average waiting time = **7**

Average turnaround time = **13**

T.A.T= waiting time + burst time

Code for Shortest Job First Scheduling

```
#include<stdio.h>
int 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;
  }

 //sorting of burst times
  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;


  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;
  total=0;

  printf("nProcesst    Burst Time    tWaiting TimetTurnaround Time");
  for(i=0;i<n;i++)
  {
      tat[i]=bt[i]+wt[i];
      total+=tat[i];
      printf("np%dtt  %dtt    %dttt%d",p[i],bt[i],wt[i],tat[i]);
  }

  avg_tat=(float)total/n;
  printf("nnAverage Waiting Time=%f",avg_wt);
  printf("nAverage Turnaround Time=%fn",avg_tat);
}
```
Output:



```
C:\WINDOWS\SYSTEM32\cmd.exe
Enter number of process:5

Enter Burst Time:
p1:4
p2:3
p3:7
p4:1
p5:2

Process     Burst Time          Waiting Time    Turnaround Time
p4             1                   0                1
p5             2                   1                3
p2             3                   3                6
p1             4                   6                10
p3             7                   10               17

Average Waiting Time=4.000000
Average Turnaround Time=7.400000


------------------
(program exited with code: 0)

Press any key to continue . . .
```

In the above program, we calculate the **average waiting** and **average turn around times** of the jobs. We first ask the user to enter the number of processes and store it in n. We then accept the burst times from the user. It is stored in the **bt array**.

After this, the burst times are sorted in the next section so the shortest one can be executed first. Here selection sort is used to sort the array of burst time **bt**.

Waiting time of the first element is zero, the remaining waiting time is calculated by using two for loop that runs from 1 to in that controls the outer loop and the inner loop is controlled by another for loop that runs from j=0 to j<i. Inside the loop, the waiting time is calculated by adding the burst time to the waiting time.

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

Total is the addition of all the waiting time together. The average waiting time is calculated:

avg_wt=(float)total/n;

and it is printed.

Next, the turnaround time is calculated by adding the burst time and the waiting time

```
for(i=0;i<n;i++)
{
  tat[i]=bt[i]+wt[i];
  total+=tat[i];
  printf("np%dtt  %dtt    %dttt%d",p[i],bt[i],wt[i],tat[i]);
}
```

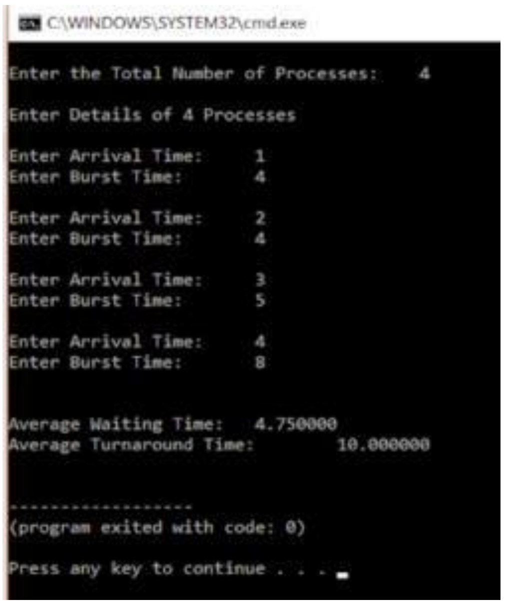
Again, here the for loop is used. And the total variable here holds the total turnaround time. After this the average turnaround time is calculated. This is how Non-Preemptive scheduling takes place

Code for Pre-emptive SJF Scheduling

```
#include <stdio.h>

int main()
{
      int arrival_time[10], burst_time[10], temp[10];
      int i, smallest, count = 0, time, limit;
      double wait_time = 0, turnaround_time = 0, end;
      float average_waiting_time, average_turnaround_time;
      printf("nEnter the Total Number of Processes:t");
      scanf("%d", &limit);
      printf("nEnter Details of %d Processesn", limit);
      for(i = 0; i < limit; i++)
      {
            printf("nEnter Arrival Time:t");
            scanf("%d", &arrival_time[i]);
```

```
        printf("Enter Burst Time:t");
        scanf("%d", &burst_time[i]);
        temp[i] = burst_time[i];
    }
    burst_time[9] = 9999;
    for(time = 0; count != limit; time++)
    {
        smallest = 9;
        for(i = 0; i < limit; i++)
        {
            if(arrival_time[i] <= time && burst_time[i] < burst_time[smallest] && burst_time[i] > 0)
            {
                smallest = i;
            }
        }
        burst_time[smallest]--;
        if(burst_time[smallest] == 0)
        {
            count++;
            end = time + 1;
            wait_time = wait_time + end - arrival_time[smallest] - temp[smallest];
            turnaround_time = turnaround_time + end - arrival_time[smallest];
        }
    }
    average_waiting_time = wait_time / limit;
    average_turnaround_time = turnaround_time / limit;
    printf("nnAverage Waiting Time:t%lfn", average_waiting_time);
    printf("Average Turnaround Time:t%lfn", average_turnaround_time);
    return 0;
}
```

Output:



```
C:\WINDOWS\SYSTEM32\cmd.exe

Enter the Total Number of Processes:    4

Enter Details of 4 Processes

Enter Arrival Time:     1
Enter Burst Time:       4

Enter Arrival Time:     2
Enter Burst Time:       4

Enter Arrival Time:     3
Enter Burst Time:       5

Enter Arrival Time:     4
Enter Burst Time:       8


Average Waiting Time:   4.750000
Average Turnaround Time:        10.000000


------------------
(program exited with code: 0)

Press any key to continue . . .
```

The only difference in preemptive and non-preemptive is that when two burst times are same the algorithm evaluates them on first come first serve basis. Hence there is an arrival time variable.

With this, we come to an end of this Shortest Job Scheduling in C article. I hope you got an idea of how this scheduling works.

3. Implement Priority Scheduling algorithm.

Process Scheduling is one of the important methods used by process schedulers in operating systems. A **process** is the logical representation of the work that must be done by the system. Process scheduling involves the methods used by the schedulers to determine the order of execution of a process.

Introduction

Process scheduling is a method used in selecting processes in a specific order from the total processes available to be executed. The queue in which all the processes ready to be executed are present is called the **job queue**. The queue in which the process selected in a specific order by the process scheduling algorithm is present is called the **ready queue**.

There are different algorithms to determine the order by which the process will be selected. This order will have an impact on the performance and speed of the operating systems. These different kinds of algorithms are being used in process scheduling to increase the efficiency of process execution. The priority scheduling algorithm is one of the scheduling algorithms designed to select the process from the ready queue into the job queue based on the priority assigned to the process.

What is Priority Scheduling Algorithm?

The priority scheduling algorithm follows a method by which a priority is set to the processes available for execution, and the process is selected based on the descending order of priority into the ready queue for execution by the CPU. Several factors can be used to determine the priority value of a process. The factors include the time taken to complete execution, memory spaces required by the process, etc.

There are two types of priority scheduling algorithms. They are,

- Priority Preemptive Scheduling Algorithm
- Priority Non-preemptive Scheduling Algorithm

In the **Preemptive** algorithm, during the execution of a process with high priority, if there is an arrival of another process with a priority higher than the process under execution, then the process currently under execution is stopped, and the new process is allowed to execute.

In the **Non-preemptive** algorithm, the process with the highest priority is allowed to execute in the CPU, and if there is an arrival of another process with a priority higher than the process under execution, then the new process will have to wait until the execution of the current process is completed.

Note: The time required for a process to complete execution is called the **burst time** of the process, and the time required for the process to arrive in the ready queue is called the **arrival time**.

Examples of Priority Scheduling Algorithm

Let us consider the following example to understand the priority scheduling algorithm clearly.

Consider there are 3 processes A, B, and C with the burst times 5,6 and 7, respectively. The priority of the three processes is 2,1 and 3, respectively. We can consider all the processes arrive on the ready queue at time 0. All the information is expressed in the table below,

Process	**Burst Time**	**Priority**	**Arrival Time**
A	5	2	0
B	6	1	0
C	7	3	0

According to the non-preemptive scheduling algorithm, the process with the highest priority will be executed first, followed by the process with low priority. Therefore, the order of execution of the process will be,

```
C ----> A -----> B
```

We generally use a gaunt chart to express this output. The following images give the Gantt chart of the problem,

We can see from the diagram that process A will have to wait until the execution of process C is completed to start execution.

Let us now consider the same problem, with the arrival time of processes A, B, and C being 1,3,2 respectively.

Process	**Burst Time**	**Priority**	**Arrival Time**
A	5	2	1
B	6	1	3
C	7	3	2

When a preemptive priority scheduling algorithm is used, process A is allowed to execute at a time of 1 second, and when process C enters the ready queue at a time of 2 seconds, process A is stopped, and process C is allowed to run. This can be useful in cases where a low- priority process may occupy a large amount of time and does not allow other higher-priority processes to execute. The order of execution will be,

```
A ---> C ---> A ---> B
```

Process A will be executed in two phases, the first phase will execute for a time of 1 second, and the second phase will execute for the remaining time of 4 seconds. Since process B has the lowest priority, process B is executed as the last process.

Characteristics of Priority Scheduling Algorithm

The characteristics of the priority scheduling algorithm in C are,

- Every process is associated with a priority number, and processes are executed according to that priority number.
- The priority scheduling algorithm is useful for performing batch processes.
- If there are two processes with the same priority, then the first process to reach the ready queue will be allowed to execute.
- In the case of preemptive priority scheduling, if a process with a higher priority arrives during the execution of a low-priority task, the low-priority task is paused, and the new task is allowed to be executed.
- The priority that can be modified is called dynamic priority, and which can't be modified is called static priority.

Note: The batch process refers to the programs which are executed repetitively in frequent time intervals by the operating systems.

Program to Implement Priority Scheduling Algorithm

The following code shows the implementation of the Priority Scheduling program in C:

```c
#include<stdio.h>
 // structure representing a structure
struct priority_scheduling {

  // name of the process
  char process_name;

  // time required for execution
  int burst_time;

  // waiting time of a process
  int waiting_time;

  // total time of execution
  int turn_around_time;

  // priority of the process
  int priority;
};

int main() {

  // total number of processes
  int number_of_process;

  // total waiting and turnaround time
  int total = 0;

  // temporary structure for swapping
  struct priority_scheduling temp_process;

  // ASCII numbers are used to represent the name of the process
  int ASCII_number = 65;

  // swapping position
  int position;

  // average waiting time of the process
  float average_waiting_time;

  // average turnaround time of the process
  float average_turnaround_time;

  printf("Enter the total number of Processes: ");
  // get the total number of the process as input
  scanf("%d", & number_of_process);
```

```
// initializing the structure array
struct priority_scheduling process[number_of_process];

printf("\nPlease Enter the  Burst Time and Priority of each process:\n");

// get burst time and priority of all process
for (int i = 0; i < number_of_process; i++) {

  // assign names consecutively using ASCII number
  process[i].process_name = (char) ASCII_number;

  printf("\nEnter the details of the process %c \n", process[i].process_name);
  printf("Enter the burst time: ");
  scanf("%d", & process[i].burst_time);

  printf("Enter the priority: ");
  scanf("%d", & process[i].priority);

  // increment the ASCII number to get the next alphabet
  ASCII_number++;

}

// swap process according to high priority
for (int i = 0; i < number_of_process; i++) {

  position = i;

  for (int j = i + 1; j < number_of_process; j++) {

    // check if priority is higher for swapping
    if (process[j].priority > process[position].priority)
      position = j;
  }
  // swapping of lower priority process with the higher priority process
  temp_process = process[i];
  process[i] = process[position];
  process[position] = temp_process;
}
// First process will not have to wait and hence has a waiting time of 0
process[0].waiting_time = 0;

for (int i = 1; i < number_of_process; i++) {
  process[i].waiting_time = 0;
  for (int j = 0; j < i; j++) {
    // calculate waiting time
    process[i].waiting_time += process[j].burst_time;
  }
```

```
    // calculate total waiting time
    total += process[i].waiting_time;
  }

  // calculate average waiting time
  average_waiting_time = (float) total / (float) number_of_process;

  // assigning total as 0 for next calculations
  total = 0;

  printf("\n\nProcess_name \t Burst Time \t Waiting Time \t  Turnaround Time\n");
  printf("------------------------------------------------------------------\n");

  for (int i = 0; i < number_of_process; i++) {

    // calculating the turnaround time of the processes
    process[i].turn_around_time = process[i].burst_time + process[i].waiting_time;

    // calculating the total turnaround time.
    total += process[i].turn_around_time;

    // printing all the values
    printf("\t    %c  \t\t    %d  \t\t  %d  \t\t  %d", process[i].process_name, process[i].burst_time,
process[i].waiting_time, process[i].turn_around_time);
    printf("\n-----------------------------------------------------------------\n");
  }

  // calculating the average turn_around time
  average_turnaround_time = (float) total / (float) number_of_process;

  // average waiting time
  printf("\n\n Average Waiting Time : %f", average_waiting_time);

  // average turnaround time
  printf("\n Average Turnaround Time: %f\n", average_turnaround_time);

  return 0;
}
```

In the above example, we can assume that every process arrives in the ready queue at the same time(0).

- A process in the priority scheduling program in c is represented with a structure called struct priority_scheduling.
- **Waiting time** represents the time the process has to wait to enter into a state of execution. **Turn around time** represents the total time required for a process to complete execution.

- The process is ordered based on priority by swapping the lower priority process with a higher priority process using the temp_proces variable.
- The waiting time is calculated by adding the process's burst time and the previous process's waiting time. The waiting time of the first process is always 0.
- The turnaround time is calculated by adding the burst time and waiting time of the process.
- The average turnaround time and average waiting time are calculated by dividing the total waiting and average time by the total number of processes.
- These average times can be used to estimate the efficiency of the algorithm.

Output

The output for the priority scheduling program in C is shown below,

```
Enter the total number of Processes :3
Please Enter the  Burst Time and Priority of each process:

Enter the details of the process A
Enter the burst time: 5
Enter the priority: 2
Enter the details of the process B
Enter the burst time: 6
Enter the priority: 1
Enter the details of the process C
Enter the burst time: 7
Enter the priority: 3

Process_name        Burst Time          Waiting Time        Turnaround Time
-----------------------------------------------------------
   C                7               0                       7
-----------------------------------------------------------
   A                5               7                       12
-----------------------------------------------------------
   B                6               12                      18
-----------------------------------------------------------


Average Waiting Time: 6.333333
Average Turnaround Time: 12.333333
```

The output depicts the same order of execution as we have seen in the example section,C --> A --> B. Since the average waiting time of the above example is very less, we can assume that the algorithm is a good fit for the input processes.

The time complexity for the priority scheduling program in c for best, worst and average time is,

- **Θ(N)** for **best** case time complexity
- **Θ(N)** for **Average** case time complexity.
- **Θ(N^2)** for **Worst** case time complexity as here we have sorted according to the priority using the selection sort algorithm. The space complexity for the priority scheduling program in C is **Θ(1)**.

The N in the time complexity represents the number of processes.

Advantages and Disadvantages

Some of the advantages of priority scheduling are,

- The priority scheduling algorithm provides a way to assign priorities to a process based on factors such as time of execution, memory requirements, etc...
- Important processes can be assigned higher priorities to be assured of faster execution.
- The algorithm is well defined for cases where there is a limited number of processes.

Some of the disadvantages of priority scheduling are,

- There is a possibility for a process with lower priority to not get a chance of execution for a long time. This property is called **starvation**.
- It is harder to choose the factor based on which priorities will be assigned.
- In case of crashes or damage to the computer, the low-priority process will be lost completely because of the not scheduling of these processes.
- The resources can't be utilized in parallel using this algorithm.

Conclusion

- The priority scheduling algorithm determines the order of execution of the process based on the priority assigned to the process.

- The priority is assigned to a process based on factors such as time required for execution, memory required by the process, etc...
- There are preemptive and non-preemptive types of priority scheduling algorithms.
- The priority scheduling program in C is implemented by swapping process having lower priorities with the process having higher priority
- The efficiency of the algorithm is calculated based on the average waiting and turnaround time.
- There are possibilities for a process having low priority to fall into a state of starvation for resources.
- The algorithm provides a method to assign priority to important processes for faster execution.

4. Implement Round Robin Scheduling algorithm.

What is Round Robin Scheduling?

Round Robin Scheduling is a scheduling algorithm used by the system to schedule CPU utilization. This is a preemptive algorithm. There exist a fixed time slice associated with each request called the quantum. The job scheduler saves the progress of the job that is being executed currently and moves to the next job present in the queue when a particular process is executed for a given time quantum.

No process will hold the CPU for a long time. The switching is called a context switch. It is probably one of the best scheduling algorithms. The efficiency of this algorithm depends on the quantum value.

ROUND ROBIN SCHEDULING ALGORITHM

- We first have a queue where the processes are arranged in first come first serve order.
- A quantum value is allocated to execute each process.
- The first process is executed until the end of the quantum value. After this, an interrupt is generated and the state is saved.
- The CPU then moves to the next process and the same method is followed.
- Same steps are repeated till all the processes are over.

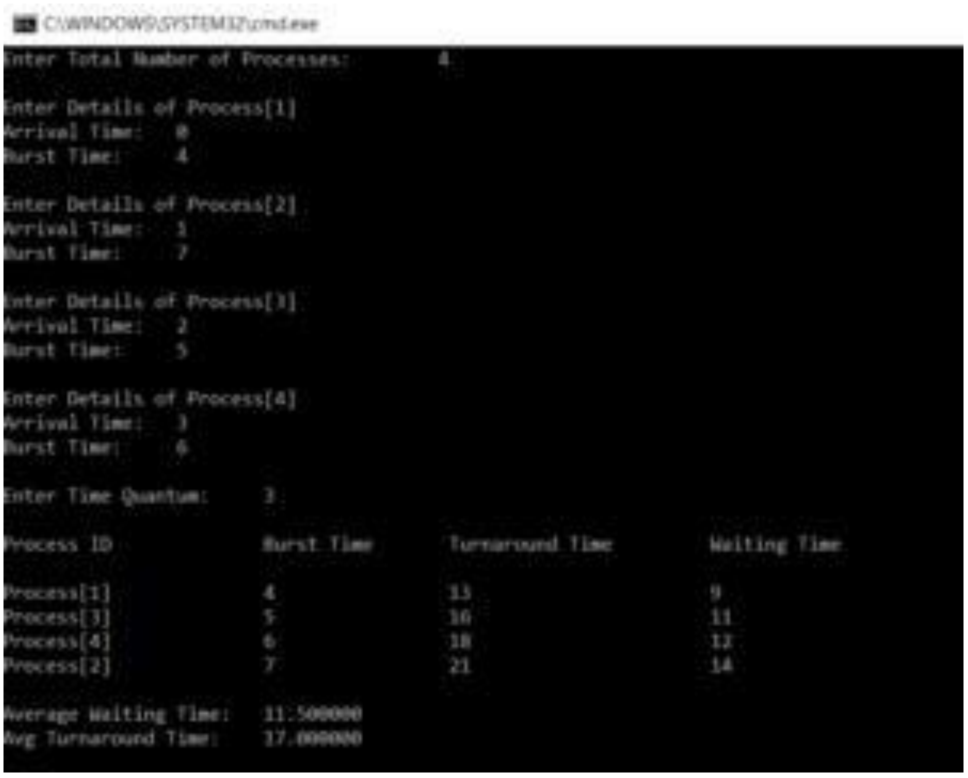
Consider the Example Code

```
#include<stdio.h>

int main()
{
      int i, limit, total = 0, x, counter = 0, time_quantum;
      int wait_time = 0, turnaround_time = 0, arrival_time[10], burst_time[10], temp[10];
      float average_wait_time, average_turnaround_time;
      printf("nEnter Total Number of Processes:t");
      scanf("%d", &limit);
      x = limit;
      for(i = 0; i < limit; i++)
      {
            printf("nEnter Details of Process[%d]n", i + 1);
```

```
    printf("Arrival Time:t");

    scanf("%d", &arrival_time[i]);

    printf("Burst Time:t");

    scanf("%d", &burst_time[i]);

    temp[i] = burst_time[i];
}

printf("nEnter Time Quantum:t");
scanf("%d", &time_quantum);
printf("nProcess IDttBurst Timet Turnaround Timet Waiting Timen");
for(total = 0, i = 0; x != 0;)
{
    if(temp[i] <= time_quantum && temp[i] > 0)
    {
        total = total + temp[i];
        temp[i] = 0;
        counter = 1;
    }
    else if(temp[i] > 0)
    {
        temp[i] = temp[i] - time_quantum;
        total = total + time_quantum;
    }
    if(temp[i] == 0 && counter == 1)
    {
        x--;
        printf("nProcess[%d]tt%dtt %dttt %d", i + 1, burst_time[i], total - arrival_time[i], total - arrival
        wait_time = wait_time + total - arrival_time[i] - burst_time[i];
        turnaround_time = turnaround_time + total - arrival_time[i];
        counter = 0;
    }
    if(i == limit - 1)
    {
        i = 0;
    }
    else if(arrival_time[i + 1] <= total)
    {
        i++;
    }
    else
    {
        i = 0;
    }
}
```

```
    average_wait_time = wait_time * 1.0 / limit;
    average_turnaround_time = turnaround_time * 1.0 / limit;
    printf("nnAverage Waiting Time:t%f", average_wait_time);
    printf("nAvg Turnaround Time:t%fn", average_turnaround_time);
    return 0;
}
```

OUTPUT:



```
Enter Total Number of Processes:        4

Enter Details of Process[1]
Arrival Time:   0
Burst Time:     4

Enter Details of Process[2]
Arrival Time:   1
Burst Time:     7

Enter Details of Process[3]
Arrival Time:   2
Burst Time:     5

Enter Details of Process[4]
Arrival Time:   3
Burst Time:     6

Enter Time Quantum:     3

Process ID              Burst Time      Turnaround Time         Waiting Time

Process[1]              4               13                      9
Process[3]              5               16                      11
Process[4]              6               18                      12
Process[2]              7               21                      14

Average Waiting Time:   11.500000
Avg Turnaround Time:    17.000000
```

EXPLANATION:

In the above code, we ask the user to enter the number of processes and arrival time and burst time for each process. We then calculate the waiting time and the turn around time using the round-robin algorithm.

The main part here is calculating the turn around time and the waiting time. Turn around time is calculated by adding the total time taken and subtracting the arrival time.

The waiting time is calculated by subtracting the arrival time and burst time from the total and adding it t0 the waiting time. This is how the round-robin scheduling takes place.

ADVANTAGES:

- Low overhead for decision making.
- Unlike other algorithms, it gives equal priority to all processes.
- Starvation rarely occurs in this process.

DISADVANTAGES:

- The efficiency of the system is decreased if the quantum value is low as frequent switching takes place.
- The system may become unresponsive if the quantum value is high.

5. Write a program to implement Producer-Consumer Problem using semaphores.

Producer Consumer Problem in C

The producer-consumer problem is an example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer that shares a common fixed-size buffer use it as a queue.

- The producer's job is to generate data, put it into the buffer, and start again.
- At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time.

Problem: Given the common fixed-size buffer, the task is to make sure that the producer can't add data into the buffer when it is full and the consumer can't remove data from an empty buffer.

Solution: The producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same manner, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

Note: An inadequate solution could result in a deadlock where both processes are waiting to be awakened.

Approach: The idea is to use the concept of parallel programming and Critical Section to implement the Producer-Consumer problem in C language using **OpenMP**.

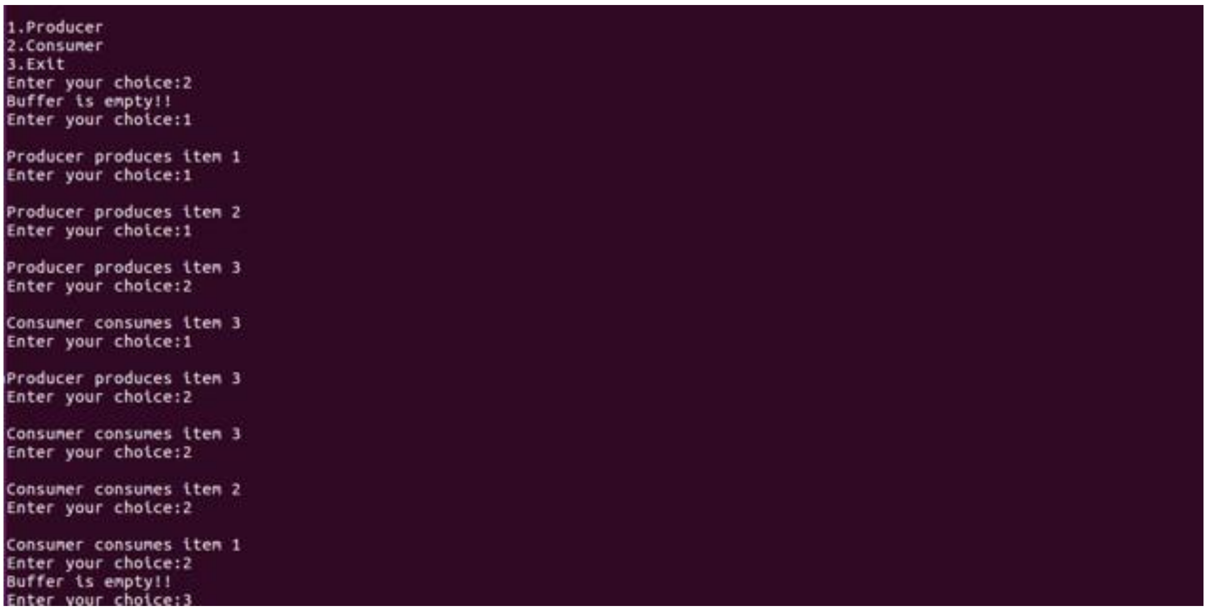
Below is the implementation of the above approach:

```
// C program for the above approach

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

// Initialize a mutex to 1
int mutex = 1;

// Number of full slots as 0
int full = 0;

// Number of empty slots as size
// of buffer
```

```
int empty = 10, x = 0;

// Function to produce an item and
// add it to the buffer
void producer()
{
    // Decrease mutex value by 1
    --mutex;

    // Increase the number of full
    // slots by 1
    ++full;

    // Decrease the number of empty
    // slots by 1
    --empty;

    // Item produced
    x++;
    printf("\nProducer produces"
           "item %d",
           x);

    // Increase mutex value by 1
    ++mutex;
}

// Function to consume an item and
// remove it from buffer
void consumer()
{
    // Decrease mutex value by 1
    --mutex;

    // Decrease the number of full
    // slots by 1
    --full;

    // Increase the number of empty
    // slots by 1
    ++empty;
    printf("\nConsumer consumes "
           "item %d",
           x);
    x--;

    // Increase mutex value by 1
    ++mutex;
}
```

```
// Driver Code
int main()
{
    int n, i;
    printf("\n1. Press 1 for Producer"
           "\n2. Press 2 for Consumer"
           "\n3. Press 3 for Exit");

// Using '#pragma omp parallel for'
// can  give wrong value due to
// synchronization issues.

// 'critical' specifies that code is
// executed by only one thread at a
// time i.e., only one thread enters
// the critical section at a given time
#pragma omp critical

    for (i = 1; i > 0; i++) {

        printf("\nEnter your choice:");
        scanf("%d", &n);

        // Switch Cases
        switch (n) {
        case 1:

            // If mutex is 1 and empty
            // is non-zero, then it is
            // possible to produce
            if ((mutex == 1)
                && (empty != 0)) {
                producer();
            }

            // Otherwise, print buffer
            // is full
            else {
                printf("Buffer is full!");
            }
            break;

        case 2:

            // If mutex is 1 and full
            // is non-zero, then it is
            // possible to consume
            if ((mutex == 1)
            && (full != 0)) {
            consumer();
        }

        // Otherwise, print Buffer
        // is empty
        else {
            printf("Buffer is empty!");
        }
        break;

    // Exit Condition
    case 3:
        exit(0);
        break;
    }
  }
}
```

Output:



```
1.Producer
2.Consumer
3.Exit
Enter your choice:2
Buffer is empty!!
Enter your choice:1

Producer produces item 1
Enter your choice:1

Producer produces item 2
Enter your choice:1

Producer produces item 3
Enter your choice:2

Consumer consumes item 3
Enter your choice:1

Producer produces item 3
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
```

Aim:

To write a C Program to Implement the Memory Management Scheme Using First Fit.

Algorithm:

1. **First Fit Algorithm**
2. Get no. of Processes and no. of blocks.
3. After that get the size of each block and process requests.
4. Now allocate processes
5. if(block size >= process size)
6. else
7. Display the processes with the blocks that are allocated to a respective process.
8. Stop.

- Given memory partition of 100 KB, 500 KB, 200 KB, 300 KB and 600 KB (in order), how would each of the first-fit, best fit and worst fit algorithms place processes of 212 KB, 417 KB, 112 KB and 426 KB (in order)? Which algorithm makes the most efficient use of memory? Either it is first fit.

Program:

```
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
int main()
{
int n,m,i,count=0,j,pn[100];
int p[100],size[100];
bool flag[100];
printf("ENTER THE NO PROCESS AND MEMOR :\n ");
scanf("%d%d",&n,&m);
printf("ENTER THE SIZE OF PROCESS \n");
for(i=0;i<n;i++)
scanf("%d",&p[i]);
printf("ENTER THE SIZE OF MEMOR PARTION \n\n");
for(i=0;i<m;i++)
{
scanf("%d",&size[i]);
flag[i]=0;
}
for(i=0;i<n;i++)
{
```

```
for(j=0;j<m;j++)
{
if(p[i]<=size[j]&&flag[j]==0)
{
flag[j]=true;
pn[j]=i;
count++;
j=j+m;
}
}
}
printf("NO OF PROCESS CAN ACOMADATE :%d\n\n",count);
printf("MEMOR\tPROCESS\n");
for(i=0;i<m;i++ )
{
if(flag[i]==1)
{
printf("%d <-->%d\n",size[i],p[pn[i]]);
}
else
printf("%d\tMEMOR NOT ALLOCATED\n",size[i]);
}
return 0;
}
```

Execution:

Input:

```
4 5 212 417 112 426 100 500 200 300 600
```

Output:

```
ENTER THE NO PROCESS AND MEMOR :
 ENTER THE SIZE OF PROCESS
ENTER THE SIZE OF MEMOR PARTION

NO OF PROCESS CAN ACOMADATE :3

MEMOR   PROCESS
100         MEMOR NOT ALLOCATED


500 <-->212


200 <-->112


300         MEMOR NOT ALLOCATED


600 <-->417
```

Result:

Thus Memory Management Scheme using the First Fit program was executed Successfully.

.

6.b To write a C Program to Implement the Memory Management Scheme Using Best Fit.

Algorithm:

1. **Best Fit Algorithm**
2. Get no. of Processes and no. of blocks.
3. After that get the size of each block and process requests.
4. Then select the best memory block that can be allocated using the above definition.
5. Display the processes with the blocks that are allocated to a respective process.
6. Value of Fragmentation is optional to display to keep track of wasted memory.
7. Stop.

Exercise:

- Given memory partition of 100 KB, 500 KB, 200 KB, 300 KB and 600 KB (in order), how would each of the first-fit, best fit and worst fit algorithms place processes of 212 KB, 417 KB, 112 KB and 426 KB (in order)? Which algorithm makes the most efficient use of memory? Either it is best fit.

Program:

```
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
struct p{
int acum[100];
int jp[100];
}st;
int main()
{
int n,m,i,count=0,j,pn[100];
int p[100],size[100];
bool flag[100];
printf("ENTER THE NO PROCESS AND MEMORY :\n ");
scanf("%d%d",&n,&m);
printf("ENTER THE SIZE OF PROCESS \n");
for(i=0;i<n;i++)
scanf("%d",&p[i]);
printf("ENTER THE SIZE OF MEMORY PARTION \n");
for(i=0;i<m;i++)
{
scanf("%d",&size[i]);
flag[i]=0;
}
for(i=0;i<n;i++)
{
int ic=0,in=0;
for(j=0;j<m;j++)
{

if(p[i]<=size[j]&&flag[j]==0)
{

int k;
st.acum[in]=size[j];
st.jp[in]=j;
in++;
ic++;
for(k=ic-1;k>0;k--)
{
if(st.acum[k]<=st.acum[k-1])
{
int temp=st.acum[k];
st.acum[k]=st.acum[k-1];
st.acum[k-1]=temp;
temp=st.jp[k];
st.jp[k]=st.jp[k-1];
st.jp[k-1]=temp;
}
}
}
}
if(ic>0)
{
j=st.jp[0];
flag[j]=true;
pn[j]=i;
count++;
}
}
printf("NO OF PROCESS CAN ACOMADATE :%d\n\n",count);
printf("MEMORY\tPROCESS\n");
for(i=0;i<m;i++ )
{
if(flag[i]==1)
{
printf("%d <-->%d\n",size[i],p[pn[i]]);
}
else
printf("%d\tMEMORY NOT ALLOCATED\n",size[i]);
}
return 0;
}
```

Pause

Input:

```
4 5 212 417 112 426 100 500 200 300 600
```

Output:

```
ENTER THE NO PROCESS AND MEMORY :
ENTER THE SIZE OF PROCESS
ENTER THE SIZE OF MEMORY PARTION
NO OF PROCESS CAN ACOMADATE :4

MEMORY          PROCESS
100        MEMORY NOT ALLOCATED
500 <-->417
200 <-->112
300 <-->212
600 <-->426
```

Result:

Thus Memory Management Scheme using the Best Fit program was executed Successfully.

6.C

To write a C Program to Implement the Memory Management Scheme Using Worst Fit.

Algorithm:

- **Worst fit Algorithm**
- Get no. of Processes and no. of blocks.
- After that get the size of each block and process requests.
- Now allocate processes
- if(block size >= process size)
- else
- Display the processes with the blocks that are allocated to a respective process.
- Stop.

Exercise:

- Given memory partition of 100 KB, 500 KB, 200 KB, 300 KB and 600 KB (in order), how would each of the first-fit, best fit and worst fit algorithms place processes of 212 KB, 417 KB, 112 KB and 426 KB (in order)?
- Which algorithm makes the most efficient use of memory?
- Either it is worst fit.

Program:

<ins>Puzzle Game In Python With Source Code 2020 Free Download</ins>

```
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
struct p{
int acum[100];
int jp[100];
}st;
int main()
{
int n,m,i,count=0,j,pn[100];
int p[100],size[100];
bool flag[100];
printf("ENTER THE NO PROCESS AND MEMORY :\n ");
scanf("%d%d",&n,&m);
printf("ENTER THE SIZE OF PROCESS \n");
for(i=0;i<n;i++)
scanf("%d",&p[i]);
printf("ENTER THE SIZE OF MEMORY PARTION \n");
for(i=0;i<m;i++)
{
scanf("%d",&size[i]);
flag[i]=0;
int k;
st.acum[i]=size[i];
st.jp[i]=i;
for(k=i;k>0;k--)
{
```

```
if(st.acum[k]<=st.acum[k-1])
{
int temp=st.acum[k];
st.acum[k]=st.acum[k-1];
st.acum[k-1]=temp;
temp=st.jp[k];
st.jp[k]=st.jp[k-1];
st.jp[k-1]=temp;
}
}
}
int x=m-1;
for(i=0;i<n;i++)
{
if(p[i]<=st.acum[x]&&flag[st.jp[x]]==0)
{
flag[st.jp[x]]=true;
pn[st.jp[x]]=i;
x--;
count++;
}
}
printf("NO OF PROCESS CAN ACOMADATE :%d\n\n",count);
printf("MEMORY\tPROCESS\n");
for(i=0;i<m;i++ )
{
if(flag[i]==1)
{
printf("%d <-->%d\n",size[i],p[pn[i]]);
}
else
printf("%d\tMEMORY NOT ALLOCATED\n",size[i]);
}
return 0;
}
```

Execution:

Input:

```
4 5 212 417 112 426 100 500 200 300 600
```

Output:

```
ENTER THE NO PROCESS AND MEMORY :
 ENTER THE SIZE OF PROCESS
ENTER THE SIZE OF MEMORY PARTION
NO OF PROCESS CAN ACOMADATE :3

MEMORY          PROCESS
100       MEMORY NOT ALLOCATED
500 <-->417
```

```
200         MEMORY NOT ALLOCATED
300 <-->112
600 <-->212
```

Result:

Thus Memory Management Scheme using the Worst Fit program was executed Successfully.

7. Write a program to implement FIFO page replacement algorithm.

the operating system replaces an existing page from the main memory by bringing a new page from the secondary memory.

In such situations, the FIFO method is used, which is also refers to the First in First Out concept. This is the simplest page replacement method in which the operating system maintains all the pages in a queue. Oldest pages are kept in the front, while the newest is kept at the end. On a page fault, these pages from the front are removed first, and the pages in demand are added.

Algorithm for FIFO Page Replacement

- Step 1. Start to traverse the pages.
- Step 2. If the memory holds fewer pages, then the capacity else goes to step 5.
- Step 3. Push pages in the queue one at a time until the queue reaches its maximum capacity or all page requests are fulfilled.
- Step 4. If the current page is present in the memory, do nothing.
- Step 5. Else, pop the topmost page from the queue as it was inserted first.
- Step 6. Replace the topmost page with the current page from the string.
- Step 7. Increment the page faults.
- Step 8. Stop

FIFO Page Replacement Algorithm in C

```
// C program for FIFO page replacement algorithm
#include<stdio.h>
int main()
{
   int incomingStream[] = {4, 1, 2, 4, 5};
   int pageFaults = 0;
   int frames = 3;
   int m, n, s, pages;

   pages = sizeof(incomingStream)/sizeof(incomingStream[0]);

   printf("Incoming \t Frame 1 \t Frame 2 \t Frame 3");
   int temp[frames];
   for(m = 0; m < frames; m++)
   {
      temp[m] = -1;
   }
```

```
    for(m = 0; m < pages; m++)
    {
        s = 0;

        for(n = 0; n < frames; n++)
        {
            if(incomingStream[m] == temp[n])
            {
                s++;
                pageFaults--;
            }
        }
        pageFaults++;

        if((pageFaults <= frames) && (s == 0))
        {
            temp[m] = incomingStream[m];
        }
        else if(s == 0)
        {
            temp[(pageFaults - 1) % frames] = incomingStream[m];
        }

        printf("\n");
        printf("%d\t\t\t",incomingStream[m]);
        for(n = 0; n < frames; n++)
        {
            if(temp[n] != -1)
                printf(" %d\t\t\t", temp[n]);
            else
                printf(" - \t\t\t");
        }
    }

    printf("\nTotal Page Faults:\t%d\n", pageFaults);
    return 0;
}
```

Output –

```
Incoming Frame 1 Frame 2 Frame 3
4         4       -       -
1         4       1       -
2         4       1       2
4         4       1       2
5         5       1       2
Total Page Faults: 4
```

8. Write a program to implement LRU page replacement algorithm.

LRU Page Replacement Algorithm in C

Least Recently Used (LRU) page replacement algorithm works on the concept that the pages that are heavily used in previous instructions are likely to be used heavily in next instructions. And the page that are used very less are likely to be used less in future. Whenever a page fault occurs, the page that is least recently used is removed from the memory frames. Page fault occurs when a referenced page in not found in the memory frames.

Below program shows how to implement this algorithm in C.

Program for LRU Page Replacement Algorithm in C

```
#include<stdio.h>

int findLRU(int time[], int n){
int i, minimum = time[0], pos = 0;

for(i = 1; i < n; ++i){
if(time[i] < minimum){
minimum = time[i];
pos = i;
}
}
return pos;
}

int main()
{
   int no_of_frames, no_of_pages, frames[10], pages[30], counter = 0, time[10], flag1,
flag2, i, j, pos, faults = 0;
printf("Enter number of frames: ");
scanf("%d", &no_of_frames);
printf("Enter number of pages: ");
scanf("%d", &no_of_pages);
printf("Enter reference string: ");
   for(i = 0; i < no_of_pages; ++i){
    scanf("%d", &pages[i]);
   }

for(i = 0; i < no_of_frames; ++i){
    frames[i] = -1;
   }

```

```
 for(i = 0; i < no_of_pages; ++i){
  flag1 = flag2 = 0;

  for(j = 0; j < no_of_frames; ++j){
  if(frames[j] == pages[i]){
  counter++;
  time[j] = counter;
flag1 = flag2 = 1;
break;
}
  }

  if(flag1 == 0){
for(j = 0; j < no_of_frames; ++j){
  if(frames[j] == -1){
  counter++;
  faults++;
  frames[j] = pages[i];
  time[j] = counter;
  flag2 = 1;
  break;
  }
  }
  }

  if(flag2 == 0){
  pos = findLRU(time, no_of_frames);
  counter++;
  faults++;
  frames[pos] = pages[i];
  time[pos] = counter;
  }

  printf("\n");

  for(j = 0; j < no_of_frames; ++j){
  printf("%d\t", frames[j]);
  }
}
printf("\n\nTotal Page Faults = %d", faults);

 return 0;
}
```

Output

```
Enter number of frames: 3
Enter number of pages: 6
Enter reference string: 5 7 5 6 7 3

5 -1 -1
5 7 -1
5 7 -1
5 7 6
5 7 6
3 7 6
```

9. Write a program to detect Deadlock.

1. Mark each process that has a row in the Allocation matrix of all zeros.
2. Initialize a temporary vectorW to equal the Available vector.
3. Find an index*i* such that process*i* is currently unmarked and the*i* th row ofQ is less than or equal to W . That is,*Q ik* … *Wk*, for 1 … *k* … *m* . If no such row is found, terminate the algorithm.
4. If such a row is found, mark process*i* and add the corresponding row of the allocation matrix to W . That is, set*Wk* = *Wk* + A*ik*, for 1 … *k* … *m* . Return to step 3.

Program:

```
#include<stdio.h>
static int mark[20];
int i,j,np,nr;

int main()
{
int alloc[10][10],request[10][10],avail[10],r[10],w[10];

printf("\nEnter the no of process: ");
scanf("%d",&np);
printf("\nEnter the no of resources: ");
scanf("%d",&nr);
for(i=0;i<nr;i++)
{
printf("\nTotal Amount of the Resource R%d: ",i+1);
scanf("%d",&r[i]);
}

printf("\nEnter the request matrix:");

for(i=0;i<np;i++)
for(j=0;j<nr;j++)
scanf("%d",&request[i][j]);

printf("\nEnter the allocation matrix:");
for(i=0;i<np;i++)
for(j=0;j<nr;j++)
scanf("%d",&alloc[i][j]);
/*Available Resource calculation*/
for(j=0;j<nr;j++)
{
avail[j]=r[j];
for(i=0;i<np;i++)
```

```
{
avail[j]-=alloc[i][j];

}
}

//marking processes with zero allocation

for(i=0;i<np;i++)
{
int count=0;
 for(j=0;j<nr;j++)
   {
      if(alloc[i][j]==0)
        count++;
      else
        break;
    }
 if(count==nr)
 mark[i]=1;
}
// initialize W with avail

for(j=0;j<nr;j++)
    w[j]=avail[j];

//mark processes with request less than or equal to W
for(i=0;i<np;i++)
{
int canbeprocessed=0;
 if(mark[i]!=1)
{
   for(j=0;j<nr;j++)
    {
      if(request[i][j]<=w[j])
      canbeprocessed=1;
      else
         {
         canbeprocessed=0;
         break;
          }
     }
if(canbeprocessed)
{
mark[i]=1;

for(j=0;j<nr;j++)
w[j]+=alloc[i][j];
}
}
```

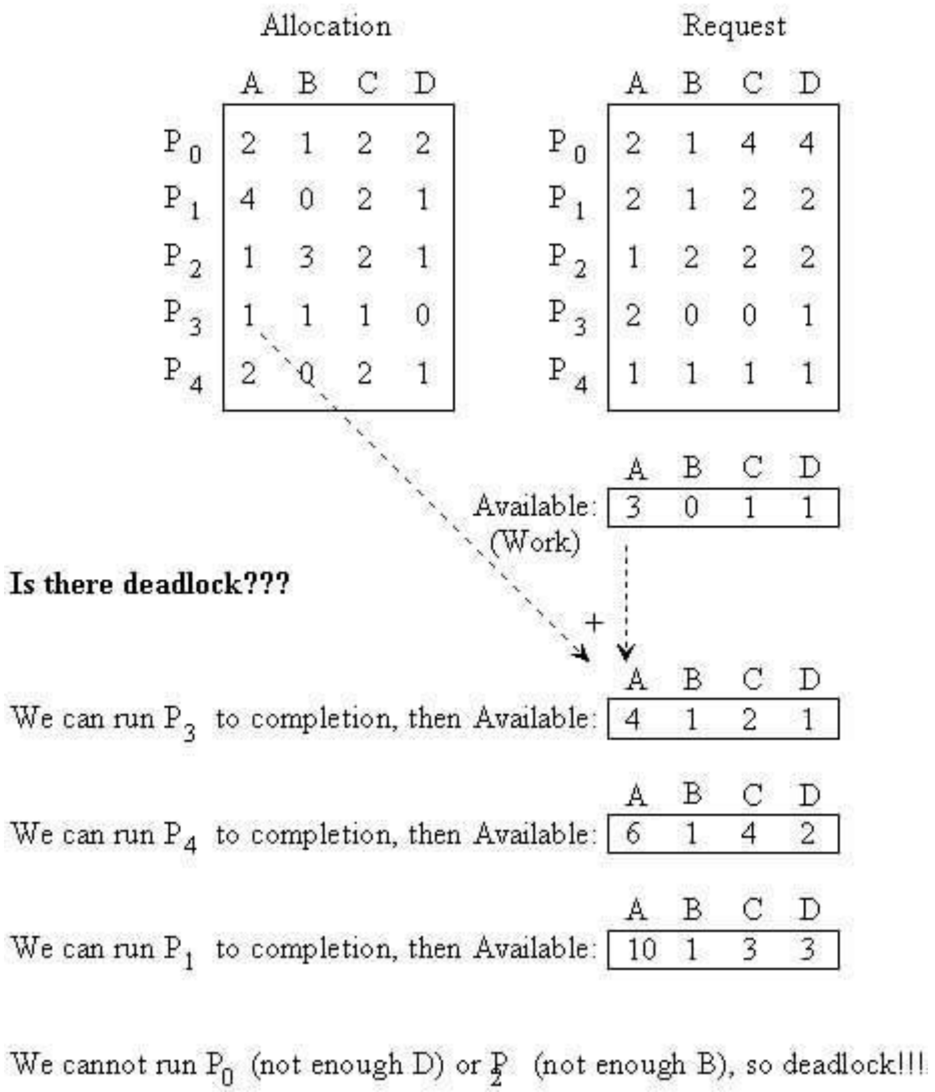
```
}

//checking for unmarked processes
int deadlock=0;
for(i=0;i<np;i++)
if(mark[i]!=1)
deadlock=1;


if(deadlock)
printf("\n Deadlock detected");
else
printf("\n No Deadlock possible");
}
```

OUTPUT:

```
Enter the no of process: 4
Enter the no of resources: 5

Total Amount of the Resource R1: 2
Total Amount of the Resource R2: 1
Total Amount of the Resource R3: 1
Total Amount of the Resource R4: 2
Total Amount of the Resource R5: 1

Enter the request matrix:0 1 0 0 1
0 0 1 0 1
0 0 0 0 1
1 0 1 0 1

Enter the allocation matrix:1 0 1 1 0
1 1 0 0 0
0 0 0 1 0
0 0 0 0 0

 Deadlock detected
-----------------
```

10. Write a program to implement Banker's algorithm for Deadlock avoidance.

This is the C Programming Implementation of bankers algorithm



Allocation

	A	B	C	D
P_0	2	1	2	2
P_1	4	0	2	1
P_2	1	3	2	1
P_3	1	1	1	0
P_4	2	0	2	1

Request

	A	B	C	D
P_0	2	1	4	4
P_1	2	1	2	2
P_2	1	2	2	2
P_3	2	0	0	1
P_4	1	1	1	1

	A	B	C	D
Available: (Work)	3	0	1	1

Is there deadlock???

+

	A	B	C	D
We can run P_3 to completion, then Available:	4	1	2	1
We can run P_4 to completion, then Available:	6	1	4	2
We can run P_1 to completion, then Available:	10	1	3	3

We cannot run P_0 (not enough D) or P_2 (not enough B), so deadlock!!!

The Banker's algorithm is a resource allocation and deadlock avoidance algorithm developed by Edsger Dijkstra that tests for safety by simulating the allocation of predetermined maximum possible amounts of all resources, and then makes an "s-state" check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue.

The algorithm was developed in the design process for the THE operating system and originally described (in Dutch) in EWD108. When a new process enters a system, it must declare the maximum number of instances of each resource type that it may ever claim; clearly, that number may not exceed the total number of resources in the system. Also, when a process gets all its requested resources it must return them in a finite amount of time.

Resources

For the Banker's algorithm to work, it needs to know three things:

How much of each resource each process could possibly request[MAX] How much of each resource each process is currently holding[ALLOCATED] How much of each resource the system currently has available[AVAILABLE]

Resources may be allocated to a process only if it satisfies the following conditions:

request ≤ max, else set error condition as process has crossed maximum claim made by it.
request ≤ available, else process waits until resources are available.

Some of the resources that are tracked in real systems are memory, semaphores and interface access.

The Banker's Algorithm derives its name from the fact that this algorithm could be used in a banking system to ensure that the bank does not run out of resources, because the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. By using the Banker's algorithm, the bank ensures that when customers request money the bank never leaves a safe state. If the customer's request does not cause the bank to leave a safe state, the cash will be allocated, otherwise the customer must wait until some other customer deposits enough.

Basic data structures to be maintained to implement the Banker's Algorithm:

Let n be the number of processes in the system and m be the number of resource types. Then we need the following data structures:

Available: A vector of length m indicates the number of available resources of each type. If Available[j] = k, there are k instances of resource type Rj available. Max: An n×m matrix defines the maximum demand of each process. If Max[i,j] = k, then Pi may request at most k instances of resource type Rj. Allocation: An n×m matrix defines the number of resources of each type currently

allocated to each process. If Allocation[i,j] = k, then process Pi is currently allocated k instances of resource type Rj.
Need: An n×m matrix indicates the remaining resource need of each process. If Need[i,j] = k, then Pi may need k more instances of resource type Rj to complete the task.

Note: Need[i,j] = Max[i,j] – Allocation[i,j].

Safe and Unsafe States

A state (as in the above example) is considered safe if it is possible for all processes to finish executing (terminate). Since the system cannot know when a process will terminate, or how many resources it will have requested by then, the system assumes that all processes will eventually attempt to acquire their stated maximum resources and terminate soon afterward. This is a reasonable assumption in most cases since the system is not particularly concerned with how long each process runs (at least not from a deadlock avoidance perspective). Also, if a process terminates without acquiring its maximum resources, it only makes it easier on the system. A safe state is considered to be the decision maker if it is going to process ready queue. Safe State ensures the Security.

Given that assumption, the algorithm determines if a state is safe by trying to find a hypothetical set of requests by the processes that would allow each to acquire its maximum resources and then terminate (returning its resources to the system). Any state where no such set exists is an unsafe state.

```
#include <stdio.h>
int curr[5][5], maxclaim[5][5], avl[5];
int alloc[5] = {0, 0, 0, 0, 0};
int maxres[5], running[5], safe=0;
int count = 0, i, j, exec, r, p, k = 1;

int main()
{
    printf("nEnter the number of processes: ");
    scanf("%d", &p);

    for (i = 0; i < p; i++) {
        running[i] = 1;
        count++;
    }
    printf("nEnter the number of resources: ");
    scanf("%d", &r);

    for (i = 0; i < r; i++) {
        printf("nEnter the resource for instance %d: ", k++);
        scanf("%d", &maxres[i]);
    }

    printf("nEnter maximum resource table:n");
    for (i = 0; i < p; i++) {
        for(j = 0; j < r; j++) {
            scanf("%d", &maxclaim[i][j]);
        }
    }

    printf("nEnter allocated resource table:n");
    for (i = 0; i < p; i++) {
        for(j = 0; j < r; j++) {
            scanf("%d", &curr[i][j]);
        }
    }

    printf("nThe resource of instances: ");
    for (i = 0; i < r; i++) {
        printf("t%d", maxres[i]);
    }

    printf("nThe allocated resource table:n");
    for (i = 0; i < p; i++) {
        for (j = 0; j < r; j++) {
            printf("t%d", curr[i][j]);
        }

        printf("n");
    }

    printf("nThe maximum resource table:n");
    for (i = 0; i < p; i++) {
        for (j = 0; j < r; j++) {
            printf("t%d", maxclaim[i][j]);
        }

        printf("n");
    }

    for (i = 0; i < p; i++) {
        for (j = 0; j < r; j++) {
            alloc[j] += curr[i][j];
        }
    }

    printf("nAllocated resources:");
    for (i = 0; i < r; i++) {
        printf("t%d", alloc[i]);
    }

    for (i = 0; i < r; i++) {
        avl[i] = maxres[i] - alloc[i];
    }

    printf("nAvailable resources:");
    for (i = 0; i < r; i++) {
        printf("t%d", avl[i]);
    }
    printf("n");

    //Main procedure goes below to check for unsafe state.
    while (count != 0) {
        safe = 0;
        for (i = 0; i < p; i++) {
            if (running[i]) {
                exec = 1;
                for (j = 0; j < r; j++) {
                    if (maxclaim[i][j] - curr[i][j] > avl[j]) {
                        exec = 0;
                        break;
                    }
                }
                if (exec) {
                    printf("nProcess%d is executingn", i + 1);
                    running[i] = 0;
                    count--;
                    safe = 1;

                    for (j = 0; j < r; j++) {
                        avl[j] += curr[i][j];
                    }

                    break;
                }
            }
        }
        if (!safe) {
            printf("nThe processes are in unsafe state.n");
            break;
        } else {
            printf("nThe process is in safe state");
            printf("nSafe sequence is:");

            for (i = 0; i < r; i++) {
                printf("t%d", avl[i]);
            }

            printf("n");
        }
    }
}
```

OUTPUT:

```
Enter the number of resources:4

Enter the number of processes:5

Enter Claim Vector:8 5 9 7

Enter Allocated Resource Table:
2 0 1 1
0 1 2 1
4 0 0 3
0 2 1 0
1 0 3 0

Enter Maximum Claim table:
3 2 1 4
0 2 5 2
5 1 0 5
1 5 3 0
3 0 3 3

The Claim Vector is: 8 5 9 7
The Allocated Resource Table:
2 0 1 1
0 1 2 1
4 0 0 3
```

```
0 2 1 0
1 0 3 0

The Maximum Claim Table:
3 2 1 4
0 2 5 2
5 1 0 5
1 5 3 0
3 0 3 3

Allocated resources: 7 3 7 5
Available resources: 1 2 2 2

Process3 is executing

The process is in safe state
Available vector: 5 2 2 5
Process1 is executing

The process is in safe state
Available vector: 7 2 3 6
Process2 is executing

The process is in safe state
Available vector: 7 3 5 7
Process4 is executing

The process is in safe state
Available vector: 7 5 6 7
Process5 is executing

The process is in safe state
Available vector: 8 5 9 7
```