OPERATING SYSTEMS

IILM UNIVERSITY

LABORATORY MANUAL

B.TECH

(III YEAR – 5 SEM) (2023-2027)

SCHOOL OF COMPUTER SCIENCE & ENGINEERING

NAME- Pranav Chauhan

Roll No.-CS-23411214

Class-3CSE6

TABLE OF CONTENTS

EXP.NO	NAMEOF THE EXPERIMENT	PAGE.NO
1	CPU SCHEDULING ALGORITHMS	
	A) FIRST COME FIRST SERVE(FCFS)	
	B) SHORTEST JOB FIRST(SJF)	
	ROUND ROBIN	
	PRIORITY	
2	PRODUCER-CONSUMER PROBLEM USING SEMAPHORES	
3	DINING-PHILOSOPHERS PROBLEM	
4	MEMORYMANAGEMENT TECHNIQUES	
	A) MULTI PROGRAMMING WITH FIXED NUMBER OF TASKS(MFT)	
	B) MULTI PROGRAMMING WITH VARIABLE NUMBER OF TASKS(MVT)	
5	CONTIGUOUS MEMORY ALLOCATION	
	BEST FIT	
6	PAGE REPLACEMENT ALGORITHMS	
	A) FIRST IN FIRST OUT(FIFO)	
	B) LEAST RECENTLY USED(LRU)	
	OPTIMAL	
7	FILE ORGANIZATION TECHNIQUES	
	A) SINGLE LEVEL DIRECTORY	
	B) TWO LEVEL DIRECTORY	
8	FILE ALLOCATION STRATEGIES	
	LINKED	
9	DEAD LOCK AVOIDANCE	
10	DEAD LOCK PREVENTION	
11	DISK SCHEDULING ALGORITHMS	
	FCFS	

EXPERIMENT NO.1

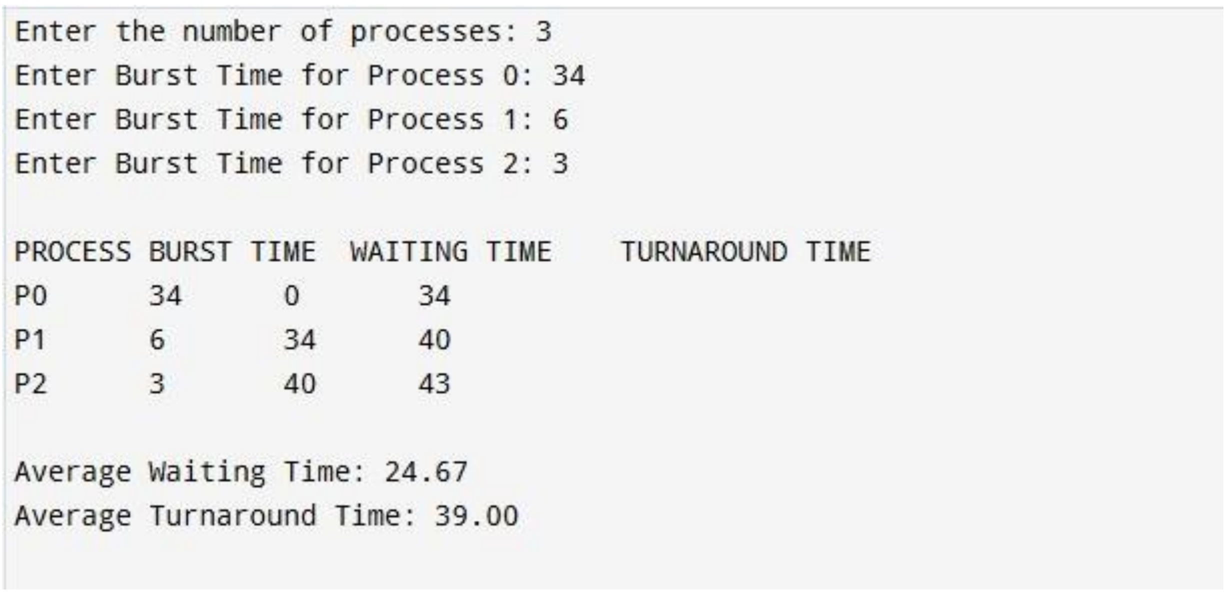
CPU SCHEDULING ALGORITHMS

A). FIRST COME FIRST SERVE:

AIM: To write a c program to simulate the CPU scheduling algorithm First Come First Serve (FCFS)

SOURCE CODE:

```
#include <stdio.h>

int main() {     int bt[20], wt[20],
tat[20], i, n;
   float wtavg = 0, tatavg = 0;

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

   for (i = 0; i < n; i++) {
      printf("Enter Burst Time for Process %d: ", i);
scanf("%d", &bt[i]);
   }

   wt[0] = 0;
tat[0] = bt[0];
   tatavg = tat[0];

   for (i = 1; i < n; i++) {
wt[i] = wt[i - 1] + bt[i - 1];
tat[i] = tat[i - 1] + bt[i];
wtavg += wt[i];         tatavg +=
tat[i];
   }

   printf("\nPROCESS\tBURST TIME\tWAITING TIME\tTURNAROUND TIME\n");
for (i = 0; i < n; i++) {
      printf("P%d\t\t%d\t\t%d\t\t%d\n", i, bt[i], wt[i], tat[i]);
}

   printf("\nAverage Waiting Time: %.2f", wtavg / n);
printf("\nAverage Turnaround Time: %.2f\n", tatavg / n);

   return 0;
}
```


```
Enter the number of processes: 3
Enter Burst Time for Process 0: 34
Enter Burst Time for Process 1: 6
Enter Burst Time for Process 2: 3

PROCESS BURST TIME  WAITING TIME    TURNAROUND TIME
P0      34      0       34
P1      6       34      40
P2      3       40      43

Average Waiting Time: 24.67
Average Turnaround Time: 39.00
```

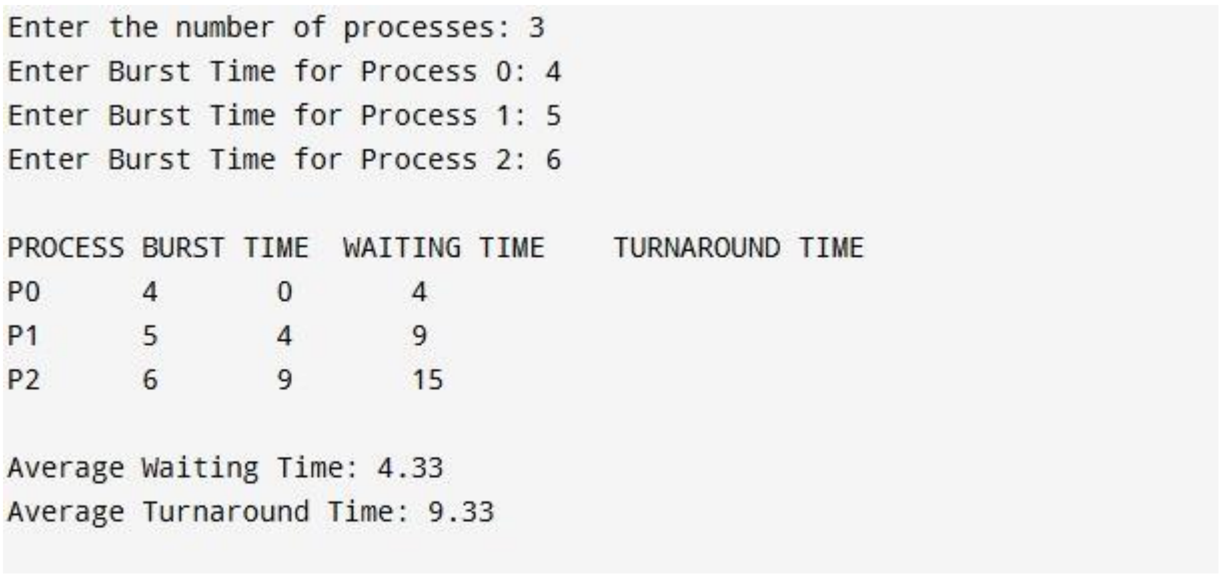
B). SHORTEST JOB FIRST:

AIM: To write a program to stimulate the CPU scheduling algorithm Shortest job first (Non- Preemption)

<u>SOURCE CODE :</u>

```
#include <stdio.h>
int main() {
   int p[20], bt[20], wt[20], tat[20];
   int i, k, n, temp;
   float wtavg = 0, tatavg = 0;

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

   for (i = 0; i < n; i++) {
p[i] = i;  // process IDs
       printf("Enter Burst Time for Process %d: ", i);
scanf("%d", &bt[i]);
   }

   // Sort processes by burst time (SJF)
   for (i = 0; i < n; i++) {
for (k = i + 1; k < n; k++) {
          if (bt[i] > bt[k]) {
temp = bt[i];
bt[i] = bt[k];
              bt[k] = temp;

              temp = p[i];
p[i] = p[k];
p[k] = temp;
          }
```

```
        }
    }

    // Calculate waiting time and turnaround
time     wt[0] = 0;     tat[0] = bt[0];
    tatavg = tat[0];

    for (i = 1; i < n; i++) {
wt[i] = wt[i - 1] + bt[i - 1];
tat[i] = wt[i] + bt[i];
wtavg += wt[i];        tatavg +=
tat[i];
    }

    // Output results
    printf("\nPROCESS\tBURST TIME\tWAITING TIME\tTURNAROUND TIME\n");
for (i = 0; i < n; i++) {
        printf("P%d\t\t%d\t\t%d\t\t%d\n", p[i], bt[i], wt[i], tat[i]);
}

    printf("\nAverage Waiting Time: %.2f", wtavg / n);
printf("\nAverage Turnaround Time: %.2f\n", tatavg / n);

    return 0;
}
```

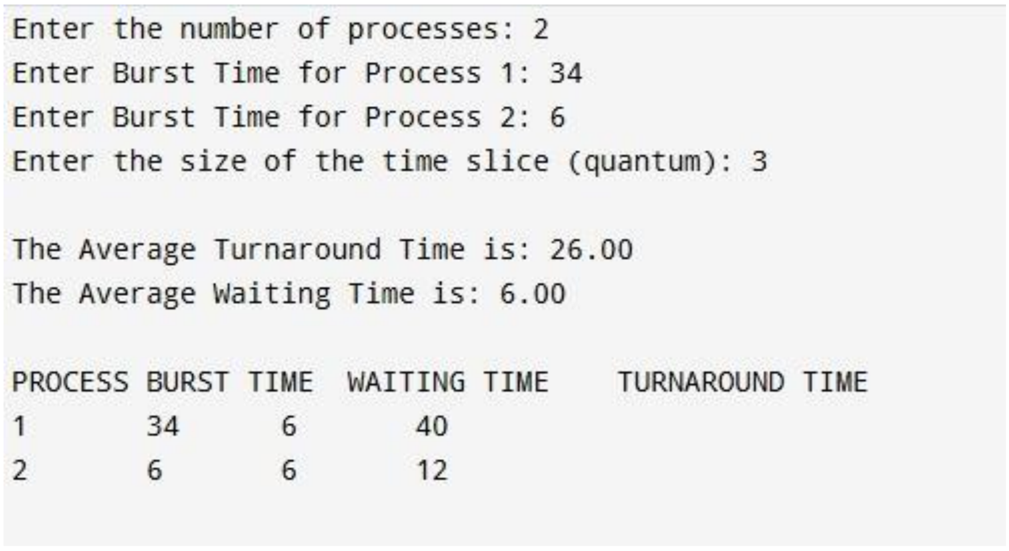


```
Enter the number of processes: 3
Enter Burst Time for Process 0: 4
Enter Burst Time for Process 1: 5
Enter Burst Time for Process 2: 6

PROCESS BURST TIME  WAITING TIME    TURNAROUND TIME
P0      4       0       4
P1      5       4       9
P2      6       9       15

Average Waiting Time: 4.33
Average Turnaround Time: 9.33
```

C). ROUND ROBIN:
AIM: To simulate the CPU scheduling algorithm round-robin.

SOURCE CODE

```
#include <stdio.h> int
main() {
    int i, j, n, bu[10], wa[10], tat[10], t, ct[10], max;
float awt = 0, att = 0, temp = 0;

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

    for (i = 0; i < n; i++) {
        printf("Enter Burst Time for Process %d: ", i + 1);
scanf("%d", &bu[i]);
        ct[i] = bu[i]; // store original burst time
    }

    printf("Enter the size of the time slice (quantum): ");
scanf("%d", &t);

    // Find maximum burst time
max = bu[0];     for (i = 1; i <
n; i++)          if (max < bu[i])
            max = bu[i];

    // Round Robin scheduling simulation
for (j = 0; j < (max / t) + 1; j++) {
for (i = 0; i < n; i++) {             if (bu[i]
!= 0) {                   if (bu[i] <= t) {
tat[i] = temp + bu[i];                     temp
+= bu[i];                    bu[i] = 0;
} else {                    bu[i] -= t;
temp += t;
                }
            }
        }
    }

    // Calculate waiting time and averages
    for (i = 0; i < n; i++) {
wa[i] = tat[i] - ct[i];
att += tat[i];         awt +=
wa[i];
    }

    printf("\nThe Average Turnaround Time is: %.2f", att / n);
printf("\nThe Average Waiting Time is: %.2f\n", awt / n);

    printf("\nPROCESS\tBURST TIME\tWAITING TIME\tTURNAROUND TIME\n");
for (i = 0; i < n; i++)
        printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, ct[i], wa[i], tat[i]);

    return 0;
}
```



```
Enter the number of processes: 2
Enter Burst Time for Process 1: 34
Enter Burst Time for Process 2: 6
Enter the size of the time slice (quantum): 3

The Average Turnaround Time is: 26.00
The Average Waiting Time is: 6.00

PROCESS BURST TIME  WAITING TIME    TURNAROUND TIME
1       34      6       40
2       6       6       12
```

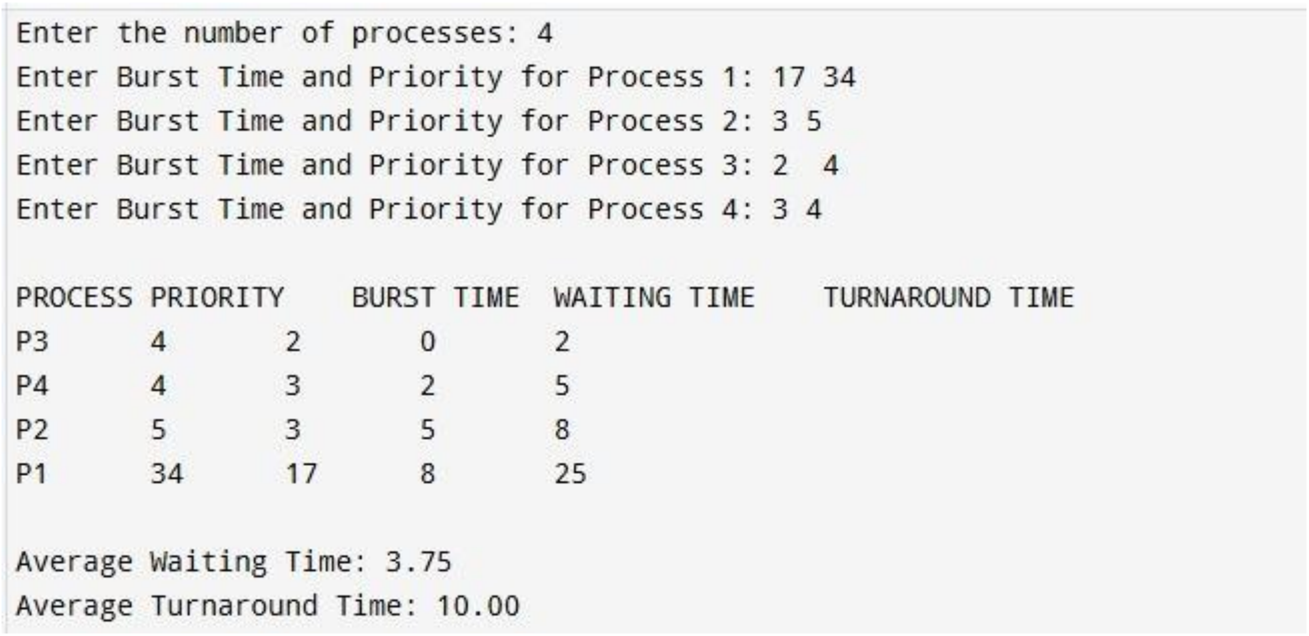
D). PRIORITY:

AIM: To write a c program to simulate the CPU scheduling priority algorithm.

SOURCE CODE:

```
#include <stdio.h>

int main() {
   int p[20], bt[20], pri[20], wt[20], tat[20];
int i, k, n, temp;
   float wtavg = 0, tatavg = 0;

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

   for (i = 0; i < n; i++) {
p[i] = i + 1;
       printf("Enter Burst Time and Priority for Process %d: ", i + 1);
scanf("%d %d", &bt[i], &pri[i]);
   }

   // Sort by priority (lower number = higher priority)
   for (i = 0; i < n; i++) {
for (k = i + 1; k < n; k++) {
if (pri[i] > pri[k]) {
temp = pri[i];              pri[i] =
pri[k];
            pri[k] = temp;

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

```
                temp = p[i];
        p[i] = p[k];
        p[k] = temp;
            }
        }
    }

    // Calculate Waiting and Turnaround Times
    wt[0] = 0;     tat[0] = bt[0];
    tatavg = tat[0];

    for (i = 1; i < n; i++) {
    wt[i] = wt[i - 1] + bt[i - 1];
    tat[i] = wt[i] + bt[i];
    wtavg += wt[i];        tatavg +=
    tat[i];
    }

    // Output
    printf("\nPROCESS\tPRIORITY\tBURST TIME\tWAITING TIME\tTURNAROUND TIME\n");
    for (i = 0; i < n; i++) {
        printf("P%d\t\t%d\t\t%d\t\t%d\t\t%d\n", p[i], pri[i], bt[i], wt[i], tat[i]);
    }

    printf("\nAverage Waiting Time: %.2f", wtavg / n);
    printf("\nAverage Turnaround Time: %.2f\n", tatavg / n);

    return 0;
}
```



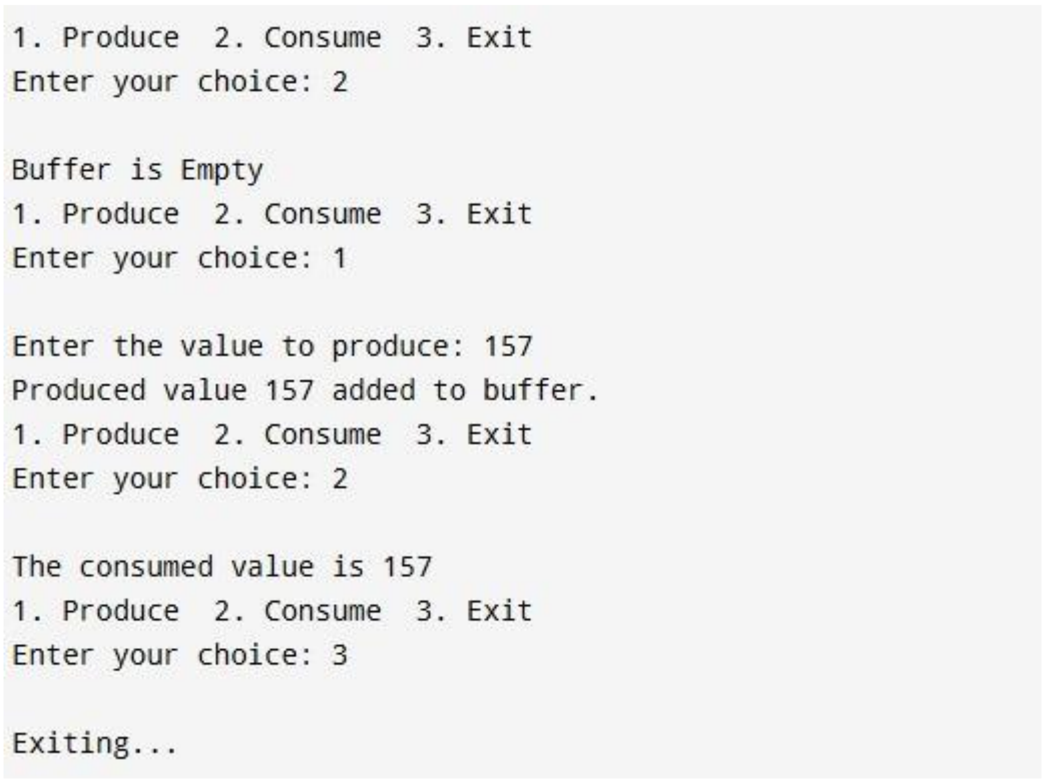
```
Enter the number of processes: 4
Enter Burst Time and Priority for Process 1: 17 34
Enter Burst Time and Priority for Process 2: 3 5
Enter Burst Time and Priority for Process 3: 2  4
Enter Burst Time and Priority for Process 4: 3 4

PROCESS PRIORITY    BURST TIME  WAITING TIME    TURNAROUND TIME
P3      4       2       0       2
P4      4       3       2       5
P2      5       3       5       8
P1      34      17      8       25

Average Waiting Time: 3.75
Average Turnaround Time: 10.00
```

EXPERIMENT.NO 2

AIM: To Write a C program to simulate producer-consumer problem using semaphores.

PROGRAM

```
#include <stdio.h>

int main() {     int
buffer[10];     int bufsize
= 10;     int in = 0, out =
0;     int produce,
consume;     int choice =
0;

    while (choice != 3) {
        printf("\n1. Produce\t2. Consume\t3.
Exit");         printf("\nEnter your choice: ");
scanf("%d", &choice);

        switch (choice) {
case 1:
                if ((in + 1) % bufsize == out) {
printf("\nBuffer is Full");
                } else {
                    printf("\nEnter the value to produce:
");                     scanf("%d", &produce);
buffer[in] = produce;                     in = (in + 1) %
bufsize;
                    printf("Produced value %d added to buffer.", produce);
                }
break;
            case 2:
if (in == out) {
                    printf("\nBuffer is Empty");
                } else {
                    consume = buffer[out];
                    printf("\nThe consumed value is %d", consume);
out = (out + 1) % bufsize;
                }
break;
            case 3:
printf("\nExiting...");
                break;
            default:
printf("\nInvalid choice!");
                break;
        }
    }

    return 0;
}
```



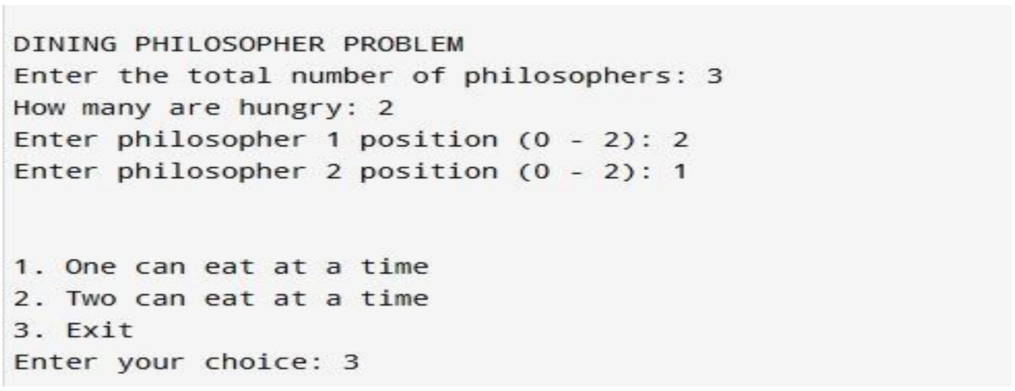
```
1. Produce  2. Consume  3. Exit
Enter your choice: 2

Buffer is Empty
1. Produce  2. Consume  3. Exit
Enter your choice: 1

Enter the value to produce: 157
Produced value 157 added to buffer.
1. Produce  2. Consume  3. Exit
Enter your choice: 2

The consumed value is 157
1. Produce  2. Consume  3. Exit
Enter your choice: 3

Exiting...
```

EXPERIMENT.NO 3

AIM: To Write a C program to simulate the concept of Dining-Philosophers problem.

PROGRAM

```
#include <stdio.h>
#include <stdlib.h> int tph, philname[20], status[20],
  howhung, hu[20], cho;

  void one();
  void two();

  int main() {
     int i;

     printf("\nDINING PHILOSOPHER PROBLEM\n");
     printf("Enter the total number of philosophers: ");
  scanf("%d", &tph);

     for (i = 0; i < tph; i++) {
  philname[i] = i + 1;
  status[i] = 1; // thinking
     }

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

     if (howhung == tph) {
        printf("\nAll are hungry... Deadlock stage will occur.");
  printf("\nExiting.\n");
        return 0;
  } else {
        for (i = 0; i < howhung; i++) {
           printf("Enter philosopher %d position (0 - %d): ", (i + 1), tph - 1);
  scanf("%d", &hu[i]);
           status[hu[i]] = 2; // hungry
        }
         do
  {
           printf("\n\n1. One can eat at a time");
  printf("\n2. Two can eat at a time");
           printf("\n3. Exit");
  printf("\nEnter your choice: ");
  scanf("%d", &cho);

           switch (cho) {

   case 1:
              one();           break;
  case 2:            two();
break;                case 3:
exit(0);               default:
printf("\nInvalid option.");
                 break;
}
       } while (1);
    }

    return 0; } void
one() {     int pos
= 0, x, i;
    printf("\nAllow one philosopher to eat at any time\n");

    for (i = 0; i < howhung; i++, pos++) {
        printf("\nP%d is granted to eat",
philname[hu[pos]]);         for (x = pos + 1; x < howhung;
x++)             printf("\nP%d is waiting", philname[hu[x]]);
    }
}

void two() {
    int i, j, s = 0, t, r, x;
    printf("\nAllow two philosophers to eat at the same time\n");

    for (i = 0; i < howhung; i++) {
for (j = i + 1; j < howhung; j++) {
            if (abs(hu[i] - hu[j]) != 1 && abs(hu[i] - hu[j]) != (tph - 1)) {
printf("\n\nCombination %d:\n", (s + 1));                t = hu[i];
r = hu[j];                s++;
                printf("P%d and P%d are granted to eat", philname[t], philname[r]);

                for (x = 0; x < howhung; x++) {
if (hu[x] != t && hu[x] != r)
                        printf("\nP%d is waiting", philname[hu[x]]);
                }
            }
        }
    }
}
```



```
DINING PHILOSOPHER PROBLEM
Enter the total number of philosophers: 3
How many are hungry: 2
Enter philosopher 1 position (0 - 2): 2
Enter philosopher 2 position (0 - 2): 1


1. One can eat at a time
2. Two can eat at a time
3. Exit
Enter your choice: 3
```

EXPERIMENT.NO 4

MEMORY MANAGEMENT

A). MEMORY MANAGEMENT WITH FIXED PARTITIONING TECHNIQUE (MFT)

AIM: To implement and simulate the MFT algorithm.

DESCRIPTION:

In this the memory is divided in two parts and process is fit into it. The process which is best suited will be placed in the particular memory where it suits. In MFT, the memory is partitioned into fixed size partitions and each job is assigned to a partition. The memory assigned to a partition does not change. In MVT, each job gets just the amount of memory it needs. That is, the partitioning of memory is dynamic and changes as jobs enter and leave the system. MVT is a more ``efficient'' user of resources. MFT suffers with the problem of internal fragmentation and MVT suffers with external fragmentation.

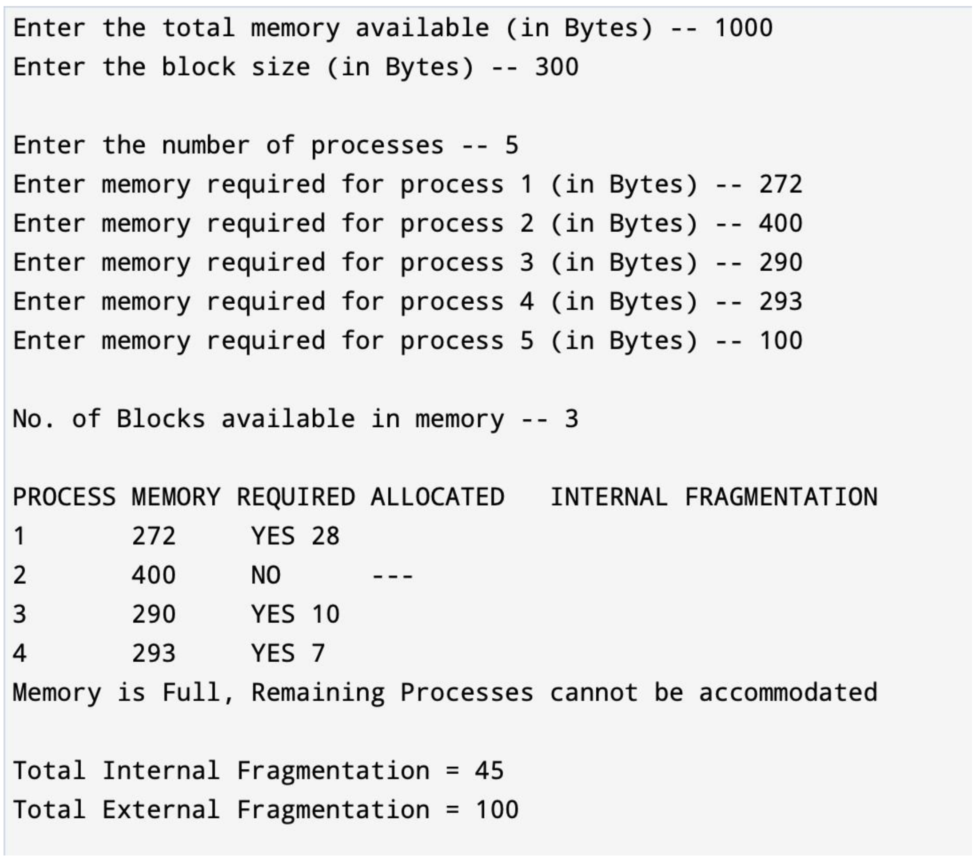
ALGORITHM:

```
Step1:  Start  the  process.
    Step2: Declarevariables.
    Step3: Enter total memory size ms.
    Step4: Allocate memory for os.
    Ms=ms-os
    Step5: Read the no partition to be divided n Partition size=ms/n.
    Step6: Read the process no and process size.
    Step 7: If process size is less than partition size allot else blocke the process. While
allocating update memory wastage-external fragmentation.
    if(pn[i]==pn[j])f=
1; if(f==0){ if(ps[i]<=siz)
    {
    extft=extft+size-
    ps[i];avail[i]=1; count++;
    }
    }
    Step 8: Print the results
```

SOURCE CODE :

```
#include <stdio.h>

int main()
{
   int ms, bs, nob, ef, n, mp[10], tif = 0;
int i, p = 0;

   printf("Enter the total memory available (in Bytes) -- ");
scanf("%d", &ms);

   printf("Enter the block size (in Bytes) -- ");
   scanf("%d", &bs);

   nob = ms / bs;
ef = ms - nob * bs;

   printf("\nEnter the number of processes -- ");
scanf("%d", &n);

   for (i = 0; i < n; i++)
   {
      printf("Enter memory required for process %d (in Bytes) -- ", i + 1);
scanf("%d", &mp[i]);
   }

   printf("\nNo. of Blocks available in memory -- %d", nob);
   printf("\n\nPROCESS\tMEMORY REQUIRED\tALLOCATED\tINTERNAL
FRAGMENTATION");

   for (i = 0; i < n && p < nob; i++)
   {
      printf("\n%d\t\t%d", i + 1, mp[i]);

      if (mp[i] > bs)
printf("\t\tNO\t\t---");
      else
      {
         printf("\t\tYES\t%d", bs - mp[i]);
         tif += (bs - mp[i]);
         p++;
      }
   }
    if (i <
n)
```

```
printf("\n
Memory
is Full,
Remainin
g
Processes
cannot
beaccom
modated")
;

    printf("\n\nTotal Internal Fragmentation = %d", tif);
printf("\nTotal External Fragmentation = %d\n", ef);     return
0;
```



```
Enter the total memory available (in Bytes) -- 1000
Enter the block size (in Bytes) -- 300

Enter the number of processes -- 5
Enter memory required for process 1 (in Bytes) -- 272
Enter memory required for process 2 (in Bytes) -- 400
Enter memory required for process 3 (in Bytes) -- 290
Enter memory required for process 4 (in Bytes) -- 293
Enter memory required for process 5 (in Bytes) -- 100

No. of Blocks available in memory -- 3

PROCESS MEMORY REQUIRED ALLOCATED   INTERNAL FRAGMENTATION
1       272     YES 28
2       400     NO      ---
3       290     YES 10
4       293     YES 7
Memory is Full, Remaining Processes cannot be accommodated

Total Internal Fragmentation = 45
Total External Fragmentation = 100
```

B) MEMORY VARIABLE PARTIONING TYPE (MVT)

AIM: To write a program to simulate the MVT algorithm

ALGORITHM:

```
Step1:  start  the  process.
        Step2: Declare variables.
        Step3: Enter total memory size ms.
        Step4: Allocate memory for os.
        Ms=ms-os
       Step5: Read the no partition to be divided n Partition size=ms/n.
        Step6: Read the process no and process size.
```

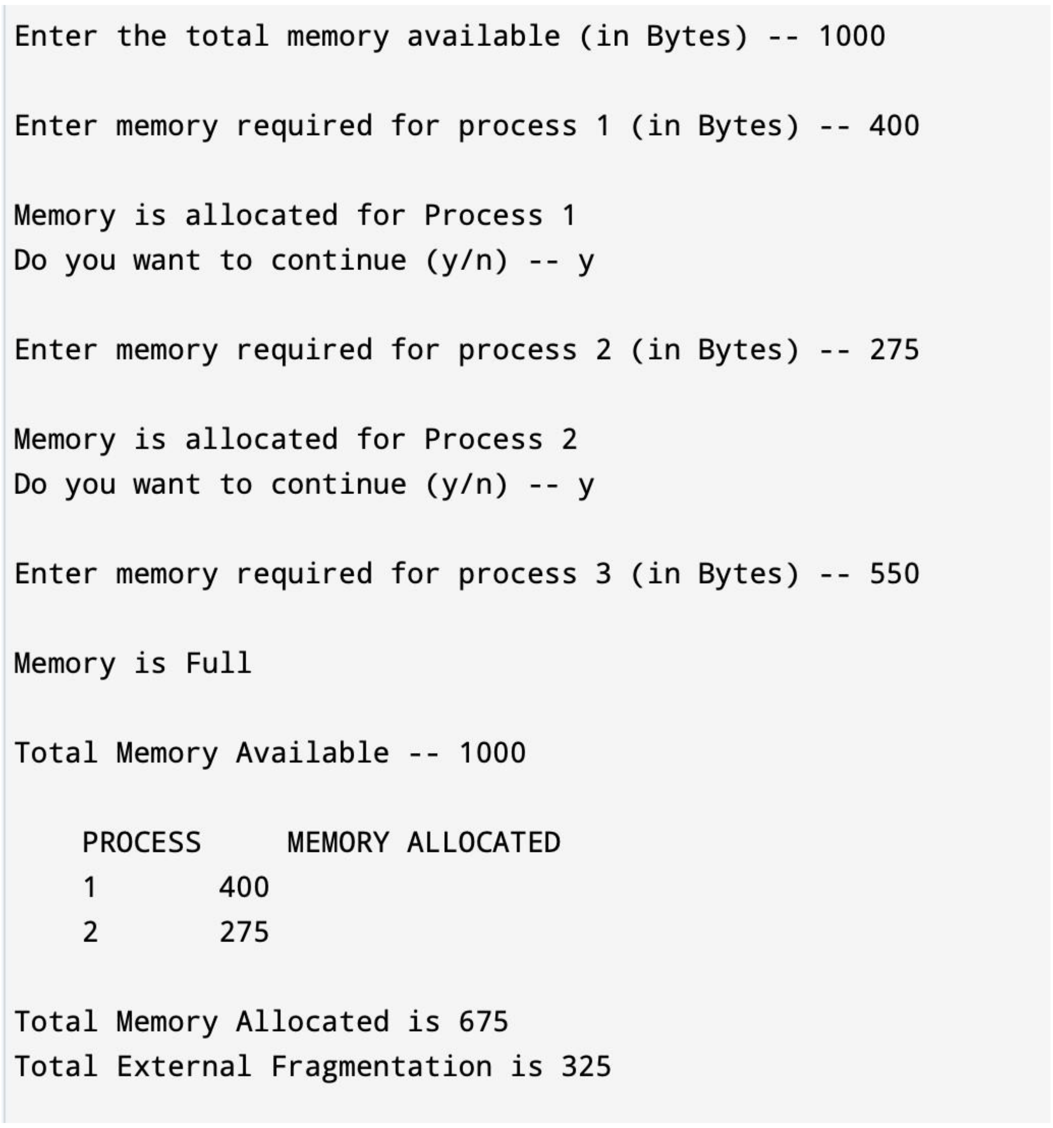
Step 7: If process size is less than partition size allot alse blocke the process. While allocating update memory wastage-external fragmentation. if(pn[i]==pn[j])
f=1; if(f==0){ if(ps[i]<=size)
{
extft=extft+size-
ps[i];avail[i]=1; count++;
} }
Ste
p
8:
Pri
nt
the
results Step 9: Stop the
process.

SOURCE CODE:

```
#include <stdio.h>

int main()
{
   int ms, mp[10], i, temp, n = 0;
   char ch = 'y';

   printf("\nEnter the total memory available (in Bytes) -- ");
scanf("%d", &ms);

   temp = ms;

   for (i = 0; ch == 'y'; i++, n++)
   {
      printf("\nEnter memory required for process %d (in Bytes) -- ", i + 1);
scanf("%d", &mp[i]);

      if (mp[i] <= temp)
      {
         printf("\nMemory is allocated for Process %d", i + 1);
temp -= mp[i];
      }
else
      {
         printf("\nMemory is Full");
         break;
      }

      printf("\nDo you want to continue (y/n) -- ");
scanf(" %c", &ch);
   }
```

```
    printf("\n\nTotal Memory Available -- %d", ms);
    printf("\n\n\tPROCESS\t\tMEMORY ALLOCATED");

    for (i = 0; i < n; i++)
        printf("\n\t%d\t\t%d", i + 1, mp[i]);

    printf("\n\nTotal Memory Allocated is %d", ms - temp);
printf("\nTotal External Fragmentation is %d\n", temp);

    return 0;
}
```

OUTPUT:



```
Enter the total memory available (in Bytes) -- 1000

Enter memory required for process 1 (in Bytes) -- 400

Memory is allocated for Process 1
Do you want to continue (y/n) -- y

Enter memory required for process 2 (in Bytes) -- 275

Memory is allocated for Process 2
Do you want to continue (y/n) -- y

Enter memory required for process 3 (in Bytes) -- 550

Memory is Full

Total Memory Available -- 1000

    PROCESS     MEMORY ALLOCATED
    1       400
    2       275

Total Memory Allocated is 675
Total External Fragmentation is 325
```

EXPERIMENT.NO 5

MEMORY ALLOCATION TECHNIQUES

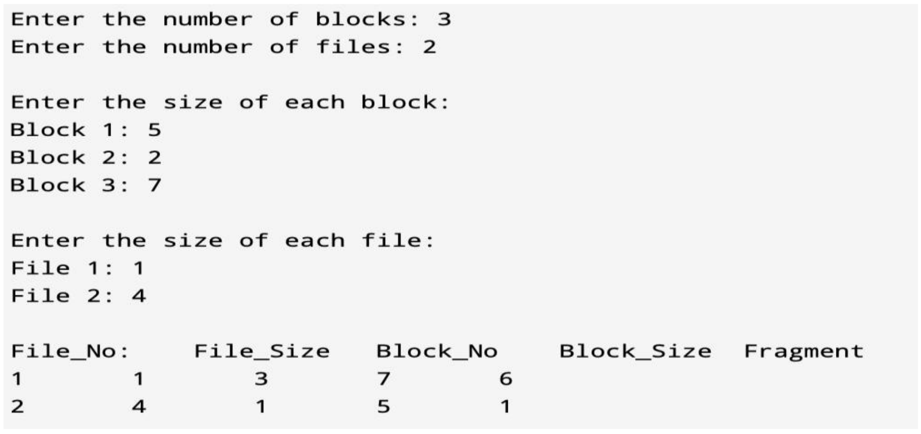
AIM: To Write a C program to simulate the following contiguous memory allocation techniques
a) Worst-fit b) Best-fit c) First-fit

DESCRIPTION

One of the simplest methods for memory allocation is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. In this multiple-partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. The operating system keeps a table indicating which parts of memory are available and which are occupied. Finally, when a process arrives and needs memory, a memory section large enough for this process is provided. When it is time to load or swap a process into main memory, and if there is more than one free block of memory of sufficient size, then the operating system must decide which free block to allocate. Best-fit strategy chooses the block that is closest in size to the request. First-fit chooses the first available block that is large enough. Worst-fit chooses the largest available block.

PROGRAM

```
#include <stdio.h>
#define MAX 25
int main()
{
   int frag[MAX], b[MAX], f[MAX];
   int bf[MAX] = {0}, ff[MAX] = {0};
int nb, nf;
   int i, j, temp, highest;
printf("\nMemory Management Scheme - Worst Fit\n");
   printf("\nEnter the number of blocks: ");
scanf("%d", &nb);
   printf("Enter the number of files: ");
scanf("%d", &nf);
   printf("\nEnter the size of each block:\n");
for (i = 1; i <= nb; i++)
   {
        printf("Block %d: ", i);
        scanf("%d", &b[i]);
    }
    printf("\nEnter the size of each file:\n");
for (i = 1; i <= nf; i++)
    {
        printf("File %d: ", i);
scanf("%d", &f[i]);
    }
// BEST FIT LOGIC
for (i = 1; i <= nf; i++)
{
highest = -1; for (j = 1;
j <= nb; j++) {if (bf[j]
== 0) // block not
allocated
      {
      temp = b[j] - f[i];

      if (temp >= 0)
      {
      if (highest == -1 || temp > (b[highest] - f[i]))
      {
      highest = j;
      }
      }
      }
      }

      ff[i] = highest; if
      (highest != -1)
      {
      frag[i] = b[highest] - f[i]; bf[highest]
      = 1;
      }
      else
      {
      frag[i] = -1; // Not allocated
      }
      }

      printf("\nFile_No:\tFile_Size\tBlock_No\tBlock_Size\tFragment\n");

      for (i = 1; i <= nf; i++)
      {
      if (ff[i] != -1)
      printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n",
      i, f[i], ff[i], b[ff[i]], frag[i]);
      else
      printf("%d\t\t%d\t\tNot Allocated\n", i, f[i]);
      }

      return 0;
              }
```



```
Enter the number of blocks: 3
Enter the number of files: 2

Enter the size of each block:
Block 1: 5
Block 2: 2
Block 3: 7

Enter the size of each file:
File 1: 1
File 2: 4

File_No:    File_Size   Block_No    Block_Size  Fragment
1       1       3       7       6
2       4       1       5       1
```

EXPERIMENT NO.6

PAGE REPLACEMENT ALGORITHMS

AIM: To implement FIFO page replacement technique.

a) FIFO   b) LRU

DESCRIPTION:

Page replacement algorithms are an important part of virtual memory management and it helps the OS to decide which memory page can be moved out making space for the currently needed page. However, the ultimate objective of all page replacement algorithms is to reduce the number of page faults.
FIFO-This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

LRU-In this algorithm page will be replaced which is least recently used

OPTIMAL- In this algorithm, pages are replaced which would not be used for the longest duration of time in the future. This algorithm will give us less page fault when compared to other page replacement algorithms.

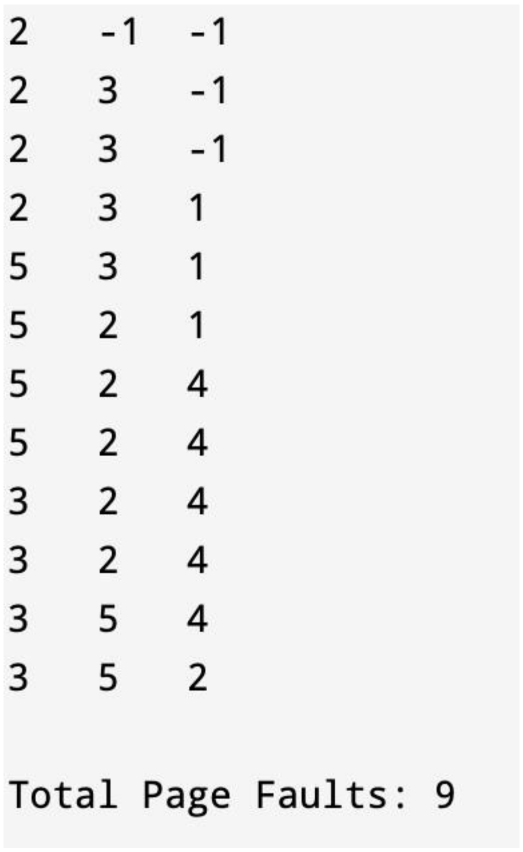
ALGORITHM:

1. Start the process
2. Read number of pages n
3. Read number of pages no
4. Read page numbers into an array a[i]
5. Initialize avail[i]=0 .to check page hit
6. Replace the page with circular queue, while re-placing check page availability in the frame Place avail[i]=1 if page is placed in the frame Count page faults
7. Print the results.
8. Stop the process.

A) <u>FIRST IN FIRST OUT</u> <u>SOURCE</u> <u>CODE</u>
:

```
#include <stdio.h>

int fr[3];   // frame array

void display()
{     for (int i = 0; i < 3;
i++)         printf("%d\t",
fr[i]);     printf("\n");
}

int main()
{
   int page[12] = {2,3,2,1,5,2,4,5,3,2,5,2};
   int i, j;
   int flag1 = 0, flag2 = 0;     int
pf = 0, frsize = 3, top = 0;

   // Initialize frames
for (i = 0; i < 3; i++)
      fr[i] = -1;

   // FIFO Logic     for
(j = 0; j < 12; j++)
   {
      flag1 = flag2 = 0;

      // Check if page is already in frame
      for (i = 0; i < frsize; i++)
      {
         if (fr[i] == page[j])
         {
            flag1 = flag2 = 1;  // hit
            break;
         }
      }

      // If empty frame is available
if (flag1 == 0)
      {
         for (i = 0; i < frsize; i++)
         {
            if (fr[i] == -1)
            {                 fr[i] =
page[j];                  flag2 = 1;
pf++;  // page fault
break;
                }
            }
        }

        // If no empty frame → replace using FIFO
        if (flag2 == 0)
        {            fr[top] =
page[j];
            top = (top + 1) % frsize;  // circular increment
            pf++;
        }

        display();
    }

    printf("\nTotal Page Faults: %d\n", pf);

    return 0;
}
```

OUTPUT:



```
2   -1  -1
2   3   -1
2   3   -1
2   3   1
5   3   1
5   2   1
5   2   4
5   2   4
3   2   4
3   2   4
3   5   4
3   5   2


Total Page Faults: 9
```

B) LEAST RECENTLY USED

AIM: To implement LRU page replacement technique.

ALGORITHM:

1. Start the process
2. Declare the size
3. Get the number of pages to be inserted
4. Get the value
5. Declare counter and stack
6. Select the least recently used page by counter value
7. Stack them according the selection.
8. Display the values
9. Stop the process

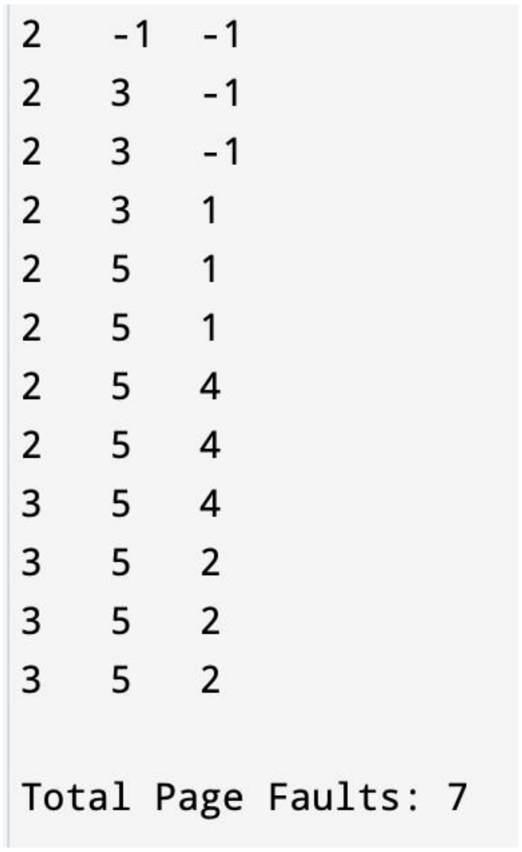
SOURCE CODE :

```
#include <stdio.h>

int fr[3]; // frames

void display() {     for (int
i = 0; i < 3; i++)
printf("%d\t", fr[i]);
printf("\n");
}

int main() {     int p[12] =
{2,3,2,1,5,2,4,5,3,2,5,2};     int i, j,
fs[3], frsize = 3;     int index = -1;
int flag1, flag2;
    int pf = 0;

    // Initialize frames     for
(i = 0; i < frsize; i++)
fr[i] = -1;

    // LRU Logic     for (j
= 0; j < 12; j++)
    {
        flag1 = flag2 = 0;

        // Check if page is already present (HIT)
for (i = 0; i < frsize; i++)
```

```
        {
           if (fr[i] == p[j])
           {                flag1 =
flag2 = 1;                break;
           }
        }

        // If not present → check for empty frame
        if (flag1 == 0)
        {
           for (i = 0; i < frsize; i++)
           {
              if (fr[i] == -1)
{                  fr[i] =
p[j];                   pf++;
flag2 = 1;
break;
              }
           }
        }

        // If no empty frame → LRU replacement
if (flag2 == 0)
        {
           for (i = 0; i < frsize; i++)
              fs[i] = 0; // reset usage tracker

           // Mark recently used frames
int k = j - 1;
           for (int count = 1; count <= frsize - 1; count++, k--)
           {
              if (k < 0) break;  // SAFE CHECK

              for (i = 0; i < frsize; i++)
              {
                 if (fr[i] == p[k])
fs[i] = 1; // recently used
              }
           }

           // Find LRU frame (fs[i] == 0)
           for (i = 0; i < frsize; i++)
           {
              if (fs[i] == 0)
              {
index = i;
break;
```

```
                }           }
fr[index] = p[j];
pf++;
        }

        display();
    }

    printf("\nTotal Page Faults: %d\n", pf);

    return 0;
}
```

OUTPUT:



```
2   -1  -1
2   3   -1
2   3   -1
2   3   1
2   5   1
2   5   1
2   5   4
2   5   4
3   5   4
3   5   2
3   5   2
3   5   2


Total Page Faults: 7
```

EXPERIMENT NO. 7

FILE ORGANIZATION TECHNIQUE

SINGLE LEVEL DIRECTORY:

AIM: Program to simulate Single level directory file organization technique.

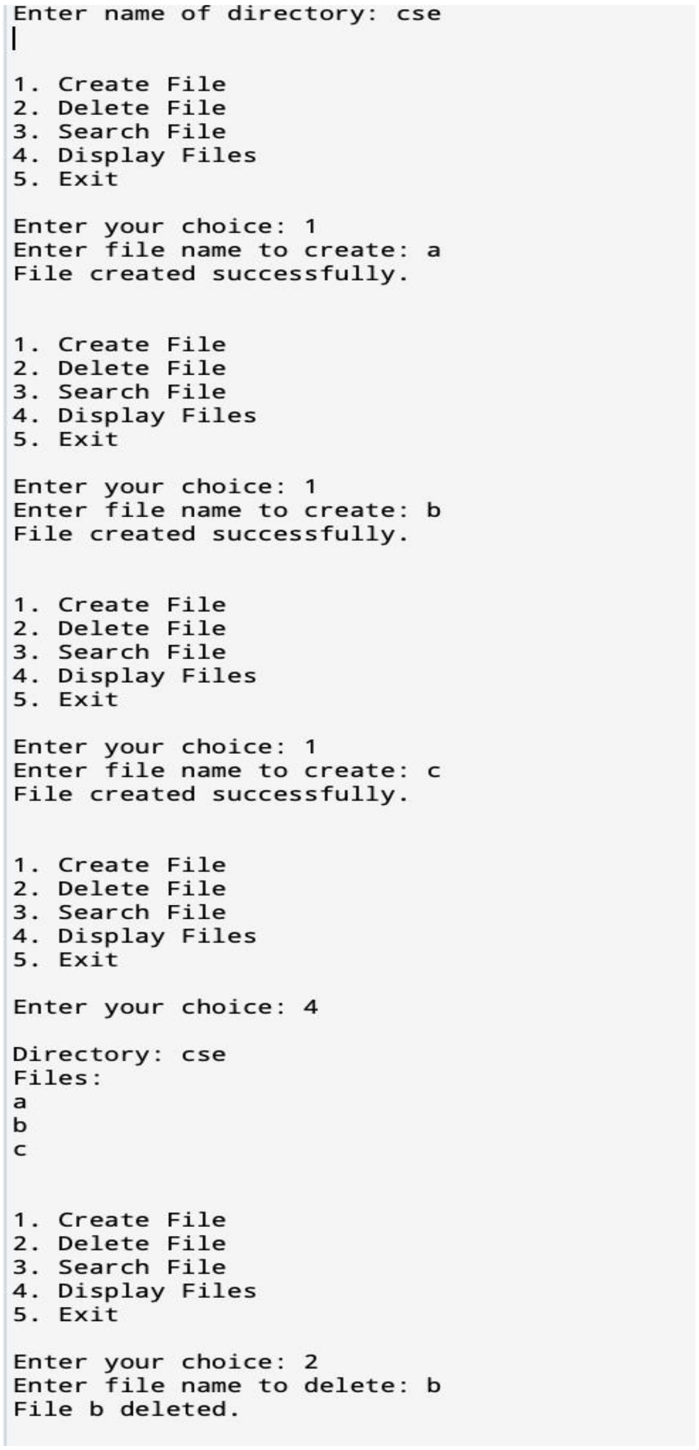
DESCRIPTION:

The directory structure is the organization of files into a hierarchy of folders. In a single-level directory system, all the files are placed in one directory. There is a root directory which has all files. It has a simple architecture and there are no sub directories. Advantage of single level directory system is that it is easy to find a file in the directory.

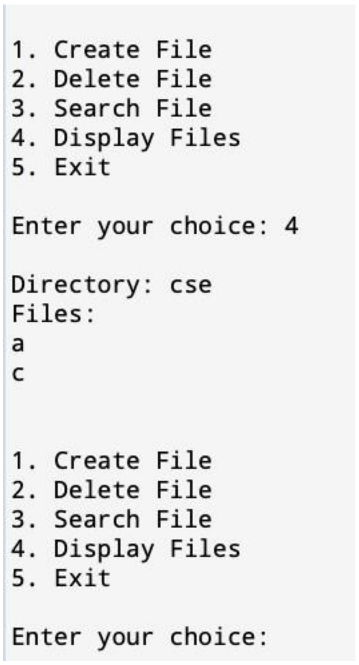
SOURCE CODE :

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

struct {
   char dname[10];
   char fname[10][10];
int fcnt;
} dir;

int main()
{
   int i, ch;
   char f[30];

   dir.fcnt = 0;

   printf("Enter name of directory: ");
scanf("%s", dir.dname);

   while (1)
   {
        printf("\n\n1. Create File");
printf("\n2. Delete File");         printf("\n3.
Search File");         printf("\n4. Display
Files");         printf("\n5. Exit");

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

        switch (ch)
        {
case 1:
                printf("Enter file name to create: ");
scanf("%s", dir.fname[dir.fcnt]);
dir.fcnt++;
                printf("File created successfully.\n");
                break;
             case
2:
                printf("Enter file name to delete: ");
                scanf("%s", f);

                for (i = 0; i < dir.fcnt; i++)
                {
                    if (strcmp(f, dir.fname[i]) == 0)
                    {
                        printf("File %s deleted.\n", f);

                        // Replace deleted file with last file
strcpy(dir.fname[i], dir.fname[dir.fcnt - 1]);
                        dir.fcnt--;
break;
                    }
                }

                if (i == dir.fcnt)
                    printf("File %s not found.\n", f);

                break;
             case
3:
                printf("Enter file name to search: ");
                scanf("%s", f);

                for (i = 0; i < dir.fcnt; i++)
                {
                    if (strcmp(f, dir.fname[i]) == 0)
                    {
                        printf("File %s found.\n", f);
                        break;
                    }
                }

                if (i == dir.fcnt)
                    printf("File %s not found.\n", f);

                break;
             case
4:
                printf("\nDirectory: %s\n", dir.dname);

                if (dir.fcnt == 0)
printf("No files.\n");
                else
{
                    printf("Files:\n");
for (i = 0; i < dir.fcnt; i++)
                        printf("%s\n", dir.fname[i]);
                }
break;
case 5:
                printf("Exiting...\n");
                return 0;

default:
                printf("Invalid choice.\n");
        }
    }

    return 0;
}
```

OUTPUT:



```
Enter name of directory: cse
|

1. Create File
2. Delete File
3. Search File
4. Display Files
5. Exit

Enter your choice: 1
Enter file name to create: a
File created successfully.


1. Create File
2. Delete File
3. Search File
4. Display Files
5. Exit

Enter your choice: 1
Enter file name to create: b
File created successfully.


1. Create File
2. Delete File
3. Search File
4. Display Files
5. Exit

Enter your choice: 1
Enter file name to create: c
File created successfully.


1. Create File
2. Delete File
3. Search File
4. Display Files
5. Exit

Enter your choice: 4

Directory: cse
Files:
a
b
c


1. Create File
2. Delete File
3. Search File
4. Display Files
5. Exit

Enter your choice: 2
Enter file name to delete: b
File b deleted.
```



```

1. Create File
2. Delete File
3. Search File
4. Display Files
5. Exit

Enter your choice: 4

Directory: cse
Files:
a
c


1. Create File
2. Delete File
3. Search File
4. Display Files
5. Exit

Enter your choice:
```

EXPERIMENT.NO.8

FILE ALLOCATION STRATEGIES

LINKED:

AIM: To implement linked file allocation technique.

DESCRIPTION:

In the chained method file allocation table contains a field which points to starting block of memory. From it for each bloc a pointer is kept to next successive block. Hence, there is no external fragmentation

ALGORTHIM:

Step 1: Start the program.
Step 2: Get the number of
files.
Step 3: Get the memory requirement of each file.
Step 4: Allocate the required locations by selecting a location randomly q= random(100);
a) Check whether the selected location is free .
b) If the location is free allocate and set flag=1 to the allocated locations. While allocating next location address to attach it to previous location

```
for(i=0;i<n;i++)
        {
        for(j=0;j<s[i];j++)
        {
        q=random(100);
    if(b[q].flag==0) b[q].flag=1; b[q].fno=j;
r[i][j]=q; if(j>0)
               {
        } } p=r[i][j-
1]; b[p].next=q;}
```

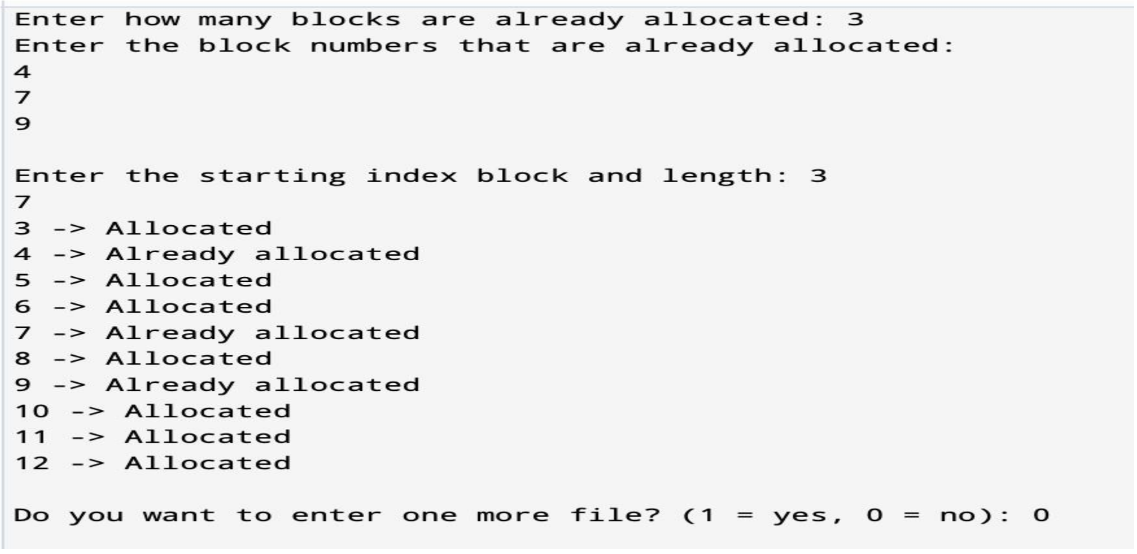
Step 5: Print the results file no, length ,Blocks allocated.
Step 6: Stop the program

SOURCE CODE :

```
#include <stdio.h>
#include <stdlib.h> int
main()
{
   int f[50], p, i, j, k, a, st, len, c;
// Initialize all blocks as free (0)
for (i = 0; i < 50; i++)
      f[i] = 0;
   printf("Enter how many blocks are already allocated: ");
scanf("%d", &p);

   printf("Enter the block numbers that are already allocated:\n");
   for (i = 0; i < p; i++)
   {
      scanf("%d", &a);
      f[a] = 1;
   }
while (1)
   {
      printf("\nEnter the starting index block and length: ");
scanf("%d %d", &st, &len);        k = len;
      for (j = st; j < (st + k); j++)
      {
         if (j >= 50)
```

```
            {
               printf("\nBlock number %d is out of range! Stopping.", j);
break;
            }
            if (f[j] == 0)
            {
f[j] = 1;
               printf("%d -> Allocated\n", j);
            }
else
            {
               printf("%d -> Already allocated\n", j);
k++;   // extend search region
            }
        }
        printf("\nDo you want to enter one more file? (1 = yes, 0 = no): ");
scanf("%d", &c);        if (c == 0)            break;
   }
return 0;
}
```

OUTPUT:



```
Enter how many blocks are already allocated: 3
Enter the block numbers that are already allocated:
4
7
9

Enter the starting index block and length: 3
7
3 -> Allocated
4 -> Already allocated
5 -> Allocated
6 -> Allocated
7 -> Already allocated
8 -> Allocated
9 -> Already allocated
10 -> Allocated
11 -> Allocated
12 -> Allocated

Do you want to enter one more file? (1 = yes, 0 = no): 0
```

EXPERIMENT.NO 9

DEAD LOCK AVOIDANCE

AIM: To Simulate bankers algorithm for Dead Lock Avoidance (Banker‘s Algorithm)

DESCRIPTION:

Deadlock is a situation where in two or more competing actions are waiting f or the other to finish, and thus neither ever does. When a new process enters a system, it must declare the maximum number of instances of each resource type it needed. This number may exceed the total number of resources in the system. When the user request a set of resources, the system must determine whether the allocation of each resources will leave the system in safe state. If it

-
-
-

will the resources are allocation; otherwise the process must wait until some other process release the resources. Data structures n-Number of process, m-number of resource types.

Available: Available[j]=k, k – instance of resource type Rj is available. Max: If max[i, j]=k, Pi may request at most k instances resource Rj.

Allocation: If Allocation [i, j]=k, Pi allocated to k instances of resource Rj Need: If

Need[I, j]=k, Pi may need k more instances of resource type Rj, Need[I, j]=Max[I, j]- Allocation[I, j];

Safety Algorithm

1. Work and Finish be the vector of length m and n respectively,
Work=Available and Finish[i] =False.

2. Find an i such that both Finish[i] =False
- Need<=Work If no such I exists go to step 4.

3. work= work + Allocation, Finish[i] =True;

4. if Finish[1]=True for all I, then the system is in safe state. Resource request algorithm

Let Request i be request vector for the process Pi, If request i=[j]=k, then process Pi wants k instances of resource type Rj.

1. if Request<=Need I go to step 2. Otherwise raise an error condition.

2. if Request<=Available go to step 3. Otherwise Pi must since the resources are available.

3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows; Available=Available-Request I; Allocation I=Allocation +Request I;

Need i=Need i- Request I;

If the resulting resource allocation state is safe, the transaction is completed and process Pi is allocated its resources. However if the state is unsafe, the Pi must wait for Request i and the old resource-allocation state is restored.

ALGORITHM:

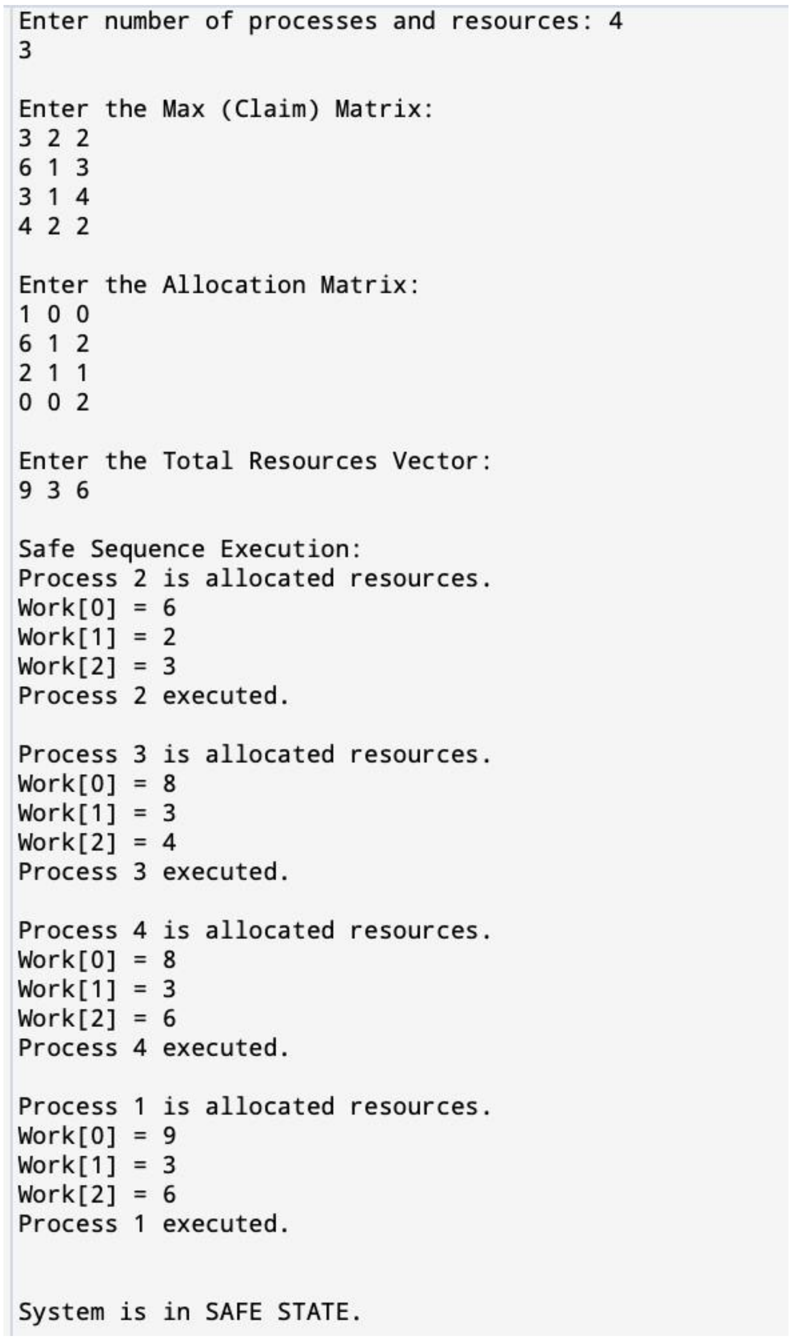
1. Start the program.
2. Get the values of resources and processes.
3. Get the avail value.
4. After allocation find the need value.
5. Check whether its possible to allocate.
6. If it is possible then the system is in safe state.
7. Else system is not in safety state.
8. If the new request comes then check that the system is in safety.
9. or not if we allow the request.
10. stop the program.
11. *end*

<u>SOURCE CODE :</u>

```
#include <stdio.h>
int main() {
   int alloc[10][10], max[10][10], need[10][10];
int total[10], avail[10], work[10];     char
finish[10];     int n, m, i, j, k;     int count = 0,
flag;
   printf("Enter number of processes and resources: ");
   scanf("%d %d", &n, &m)
   printf("\nEnter the Max (Claim) Matrix:\n");
   for (i = 0; i < n; i++)         for (j = 0; j <
m; j++)             scanf("%d", &max[i][j]);
printf("\nEnter the Allocation Matrix:\n");
   for (i = 0; i < n; i++)         for
(j = 0; j < m; j++)
scanf("%d", &alloc[i][j]);
   printf("\nEnter the Total Resources Vector:\n");
   for (i = 0; i < m; i++)
scanf("%d", &total[i]);     //
Initialize finish[]     for (i =
0; i < n; i++)         finish[i]
= 'n';
   // Calculate Available resources
   for (i = 0; i < m; i++)         avail[i] = 0;
for (i = 0; i < n; i++)         for (j = 0; j <
m; j++)             avail[j] += alloc[i][j];
for (i = 0; i < m; i++)         work[i] =
total[i] - avail[i];     // Calculate Need
matrix     for (i = 0; i < n; i++)         for (j
= 0; j < m; j++)             need[i][j] =
max[i][j] - alloc[i][j];     printf("\nSafe
Sequence Execution:\n");     while (count
< n)
   {
      int allocated = 0;
      for (i = 0; i < n; i++)
      {
         if (finish[i] == 'n')
         {
flag = 1;
            for (j = 0; j < m; j++)
            {
               if (need[i][j] > work[j])
               {
flag = 0;
break;
               }
            }
```

```
                // If all needed resources ≤ work → allocate
if (flag == 1)
                {
                   printf("Process %d is allocated resources.\n", i + 1);
for (k = 0; k < m; k++)
                    {
                       work[k] += alloc[i][k];
                       printf("Work[%d] = %d\n", k, work[k]);
                    }
finish[i] = 'y';
allocated = 1;
count++;
                   printf("Process %d executed.\n\n", i + 1);
                }
            }
        }
        // No allocation possible → deadlock
        if (allocated == 0)
        {
            printf("\nSystem is NOT in a safe state (Deadlock Possibility)\n");
return 0;
        }
    }
    printf("\nSystem is in SAFE STATE.\n");

    return 0;
                    }
```

OUTPUT



```
Enter number of processes and resources: 4
3

Enter the Max (Claim) Matrix:
3 2 2
6 1 3
3 1 4
4 2 2

Enter the Allocation Matrix:
1 0 0
6 1 2
2 1 1
0 0 2

Enter the Total Resources Vector:
9 3 6

Safe Sequence Execution:
Process 2 is allocated resources.
Work[0] = 6
Work[1] = 2
Work[2] = 3
Process 2 executed.

Process 3 is allocated resources.
Work[0] = 8
Work[1] = 3
Work[2] = 4
Process 3 executed.

Process 4 is allocated resources.
Work[0] = 8
Work[1] = 3
Work[2] = 6
Process 4 executed.

Process 1 is allocated resources.
Work[0] = 9
Work[1] = 3
Work[2] = 6
Process 1 executed.


System is in SAFE STATE.
```

EXPERIMENT.NO 10

DEAD LOCK PREVENTION

AIM: To implement deadlock prevention technique

Banker‘s Algorithm:

When a new process enters a system, it must declare the maximum number of instances of each resource type it needed. This number may exceed the total number of resources in the system. When the user request a set of resources, the system must determine whether the allocation of each resources will leave the system in safe state. If it will the resources are allocation; otherwise the process must wait until some other process release the resources.

DESCRIPTION:

Data structures

n- • Number of process, m-number of resource types.
-
- Available: Available[j]=k, k – instance of resource type Rj is available.
- Max: If max[i, j]=k, Pi may request at most k instances resource Rj.
- Allocation: If Allocation [i, j]=k, Pi allocated to k instances of resource Rj

Need:

If Need[I, j]=k, Pi may need k more instances of resource type Rj,
Need[I, j]=Max[I, j]-Allocation[I, j];

Safety Algorithm

Work and Finish be the vector of length m and n respectively, Work=Available and Finish[i] =False.

Find an i such that
both Finish[i] =False
•Need<=Work

If no such I exists go to step 4.

5. work=work+Allocation, Finish[i] =True;
if Finish[1]=True for all I, then the system is in safe state

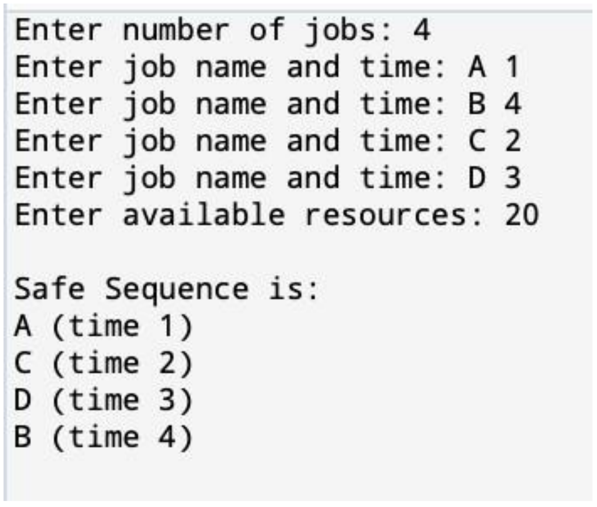
SOURCE CODE :

```
#include <stdio.h>
#include <string.h>
int main()
{
   char job[10][10];
   int time[10], temp[10], order[10];
```

```
    int safe[10];     int avail, n;     int
i, j, t, ind = 0;     printf("Enter
number of jobs: ");     scanf("%d",
&n);     for (i = 0; i < n; i++)
    {
        printf("Enter job name and time: ");
        scanf("%s %d", job[i], &time[i]);
    }
    printf("Enter available resources: ");
scanf("%d", &avail);     // Copy for
sorting     for (i = 0; i < n; i++)
    {         temp[i] =
time[i];
        order[i] = i;    // Track original index
    }
    // Sort by time (SJF sorting)
for (i = 0; i < n; i++)
    {
        for (j = i + 1; j < n; j++)
        {
            if (temp[i] > temp[j])
            {                 t =
temp[i];                 temp[i] =
temp[j];                 temp[j] =
t;                 t = order[i];
order[i] = order[j];
order[j] = t;
            }
        }
    }
    // Generate safe sequence
for (i = 0; i < n; i++)
    {         int idx =
order[i];
        if (time[idx] <= avail)
        {             safe[ind] =
idx;             avail -=
time[idx];             ind++;
}         else
        {
          printf("\nNo safe sequence possible.\n");
return 0;
        }
    }
    // Print safe sequence
printf("\nSafe Sequence is:\n");
for (i = 0; i < ind; i++)
```

```
        printf("%s (time %d)\n", job[safe[i]], time[safe[i]]);
return 0;
}
```

OUTPUT:



```
Enter number of jobs: 4
Enter job name and time: A 1
Enter job name and time: B 4
Enter job name and time: C 2
Enter job name and time: D 3
Enter available resources: 20

Safe Sequence is:
A (time 1)
C (time 2)
D (time 3)
B (time 4)
```

EXPERIMENT.NO 11

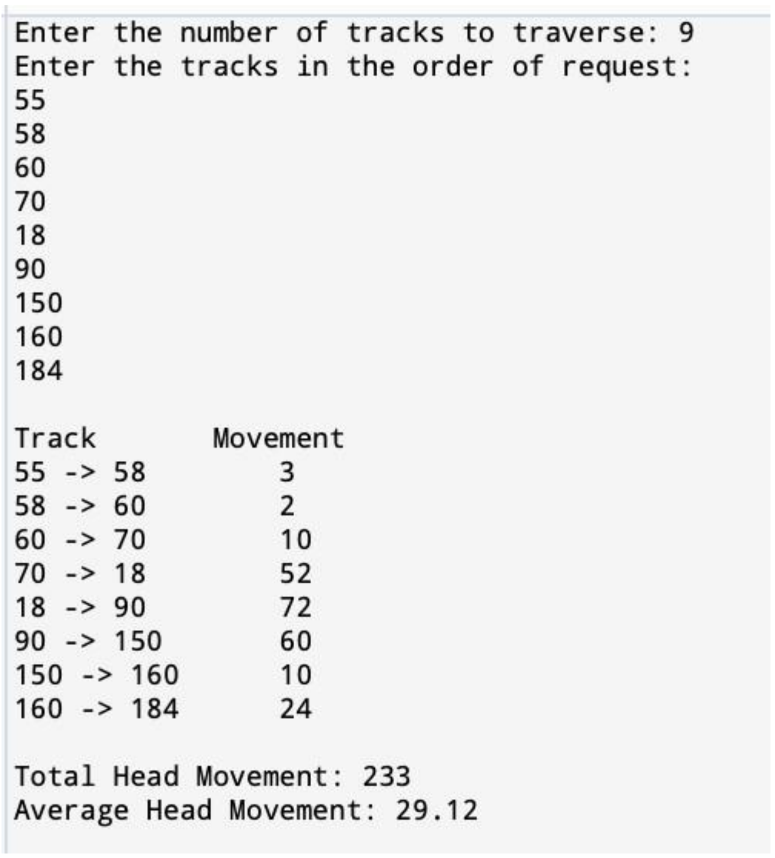
AIM: To Write a C program to simulate disk scheduling algorithms FCFS

DESCRIPTION
One of the responsibilities of the operating system is to use the hardware efficiently. For the disk drives, meeting this responsibility entails having fast access time and large disk bandwidth. Both the access time and the bandwidth can be improved by managing the order in which disk I/O requests are serviced which is called as disk scheduling. The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service. In the SCAN algorithm, the disk arm starts at one end, and moves towards the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk. C-SCAN is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip

PROGRAM

FCFS DISK SCHEDULING ALGORITHM
```
#include <stdio.h>
#include <stdlib.h>

int main()
{     int t[20],
diff[20];     int n, i,
total = 0;     float
avg;
   printf("Enter the number of tracks to traverse: ");
scanf("%d", &n);
   printf("Enter the tracks in the order of
request:\n");     for (i = 0; i < n; i++)
scanf("%d", &t[i]);     for (i = 0; i < n - 1; i++)
    {
       diff[i] = abs(t[i + 1] - t[i]);
total += diff[i];
    }
   avg = (float)total / (n - 1);
printf("\nTrack\t\tMovement\n");
   for (i = 0; i < n - 1; i++)
       printf("%d -> %d\t\t%d\n", t[i], t[i + 1], diff[i]);
printf("\nTotal Head Movement: %d", total);
   printf("\nAverage Head Movement: %.2f\n", avg);
return 0;
}
```

OUTPUT



```
Enter the number of tracks to traverse: 9
Enter the tracks in the order of request:
55
58
60
70
18
90
150
160
184

Track        Movement
55 -> 58         3
58 -> 60         2
60 -> 70         10
70 -> 18         52
18 -> 90         72
90 -> 150        60
150 -> 160       10
160 -> 184       24

Total Head Movement: 233
Average Head Movement: 29.12
```