A Lab File

on

Operating System

Submitted

for

Bachelor of Technology

in

Computer Science and Engineering

at

Indian Institute of Information Technology Una

2021-2022

Submitted To:

Dr. Sahil

Faculty in SoC

IIIT Una

Submitted By:

Name: Saurabh Singh

Class: CSC403

Roll no. 20154

Lab Experiment No. 1: File-related System Calls Implementation

Aim: To get acquainted with how the system calls are invoked for file-related functionalities.Lab Experiment No. 2: Implementation of FCFS & SJF CPU Scheduling Algorithms

Aim: To get the idea of how logics for FCFS and SJF CPU Scheduling Algorithms are developed.

FCFS Background:

Simplest of all CPU Scheduling Algorithms

Criteria: Arrival Time

Mode: Non-Preemptive

FCFS pick the job from the RQ, which has the lowest AT.

Background:

File-related system calls let you create, open, and close files, read and write files, randomly access files, alias and remove files, get information about files, check the accessibility of files, change protections, owner, and group of files, and control devices. A channel is a connection between a process and a file that appears to the process as an unformatted stream of bytes. The kernel presents and accepts data from the channel as a process reads and writes that channel.

1. using O_RDONLY, O_CREAT, O_TRUNC

Code:

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
int main(){
int fd;
fd = creat("File1.txt", S_IREAD | S_IWRITE);
if (fd != -1){
printf("Created file opened for read/write access\n");
printf("File1 is currently empty\n");
}
else
printf("Error in opening file\n");

close(fd);
exit (0);
```

Output:

```
Created file opened for read/write access
File1 is currently empty
```

2. USING O_RDWR AND S_IREAD, S_IREAD, O_CREAT simultaneously

Code:

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
static char message[] = "HELLO WORLD";
int main()
{
int fd;
char buffer[80];
fd = open("file2.txt",O_RDWR | O_CREAT | O_EXCL, S_IREAD | S_IWRITE);
if (fd != -1)
{
printf("File has been opened for READ/WRITE access\n");
write(fd, message, sizeof(message));
lseek(fd, 0, 0); /* go back to the beginning of the file */
if (read(fd, buffer, sizeof(message)) == sizeof(message))
printf("\"%s\" Message has been written to file \n", buffer);
else
printf("error in reading file \n");
close (fd);
}
else
printf("File already exists\n");

exit(0);
}
```

Output:

```
File has been opened for READ/WRITE access
"HELLO WORLD" Message has been written to file
```

3.READING FROM FILE

Code:

```c
// I/O system Calls
#include<stdio.h>
#include<string.h>
#include<unistd.h>
#include<fcntl.h>

int main (void)
{
   int fd[2];
   char buf1[12] = "hello world";
   char buf2[12];

   // assume foobar.txt is already created
   fd[0] = open("foobar.txt", O_RDWR);
   fd[1] = open("foobar.txt", O_RDWR);

   write(fd[0], buf1, strlen(buf1));
   write(1, buf2, read(fd[1], buf2, 12));

   close(fd[0]);
   close(fd[1]);

   return 0;
}
```

Output:

```
hello world
```

4.Seeking a File

Code:

```c
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
int fd;
long position;
fd = open("file2.txt", O_RDONLY);
if ( fd != -1)
{
position = lseek(fd, 0, 2); /* seek 0 bytes from end-of-file */
if (position != -1)
printf("The length of file2.txt is %ld bytes.\n", position);
else
perror("lseek error");
}
else
printf("can't open file2.txt\n");
close(fd);
}
```

Output:

```
The length of file2.txt is 12 bytes.
```

5. read character from particular position in the file

Code:

```c
// read system Call
#include<stdio.h>
#include<unistd.h>
#include<fcntl.h>
#include<stdlib.h>

int main()
{
   char c;
   int fd1 = open("File1.txt", O_RDONLY, 0);
   int fd2 = open("File1.txt", O_RDONLY, 0);
   read(fd1, &c, 1);
   read(fd2, &c, 1);
   printf("c = %c\n", c);
   exit(0);
}
```

Output:

```
c = H
```

Lab Experiment No. 2: Implementation of FCFS & SJF CPU Scheduling Algorithms

Aim: To get the idea of how logics for FCFS and SJF CPU Scheduling Algorithms are developed.

FCFS Background:

- Simplest of all CPU Scheduling Algorithms
- Criteria: Arrival Time
- Mode: Non-Preemptive
- FCFS pick the job from the RQ, which has the lowest AT.

1. FCFS : same arrival time

Code:

```
#include<stdio.h>
int main()
{
int bt[20], wt[20], tat[20], i, n;
float cwt, ctat;
printf("\nEnter the number of processes -- ");
scanf("%d",&n);
for (i = 0; i< n; i++)
{
printf("\nEnter Burst Time for Process %d -- ", i);
scanf("%d",&bt[i]);
}
wt[0] = cwt = 0;
tat[0] = ctat = bt[0];
for(i=1;i<n;i++)
{
wt[i] = wt[i-1] +bt[i-1];
tat[i] = tat[i-1] +bt[i];
cwt = cwt + wt[i];
ctat = ctat + tat[i];
}
```

```
printf("\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", i, bt[i], wt[i], tat[i]);
printf("\nAverage Waiting Time -- %f", cwt/n);
printf("\nAverage Turnaround Time -- %f", ctat/n);

return 0;
}
```

Output:

```
Enter the number of processes -- 4

Enter Burst Time for Process 0 -- 2

Enter Burst Time for Process 1 -- 3

Enter Burst Time for Process 2 -- 1

Enter Burst Time for Process 3 -- 4
         PROCESS        BURST TIME       WAITING TIME    TURNAROUND TIME

         P0              2               0               2
         P1              3               2               5
         P2              1               5               6
         P3              4               6               10
Average Waiting Time -- 3.250000
Average Turnaround Time -- 5.750000
```

2. SJF : same arrival time

Code:

```
#include <stdio.h>
int main()
{
   int p[20], bt[20], wt[20], tat[20], i, k, n, temp;
   float cwt, ctat;
   printf("\nEnter the number of processes -- ");
   scanf("%d", &n);
   for (i = 0; i < n; i++)
   {
       p[i] = i;
       printf("Enter Burst Time for Process %d -- ", i);
       scanf("%d", &bt[i]);
   }
   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;
           }
   wt[0] = cwt = 0;
   tat[0] = ctat = bt[0];
   for (i = 1; i < n; i++)
   {
       wt[i] = wt[i - 1] + bt[i - 1];
       tat[i] = tat[i - 1] + bt[i];
       cwt = cwt + wt[i];
       ctat = ctat + tat[i];
   }
   printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND
TIME\n");
   for (i = 0; i < n; i++)
```

```
      printf("\n\t P%d \t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i],
tat[i]);
   printf("\nAverage Waiting Time -- %f", cwt / n);
   printf("\nAverage Turnaround Time -- %f", ctat / n);
   return 0;
}
```

Output:

```
Enter the number of processes -- 4
Enter Burst Time for Process 0 -- 3
Enter Burst Time for Process 1 -- 2
Enter Burst Time for Process 2 -- 4
Enter Burst Time for Process 3 -- 1

         PROCESS        BURST TIME       WAITING TIME     TURNAROUND TIME

         P3              1               0                1
         P1              2               1                3
         P0              3               3                6
         P2              4               6                10
Average Waiting Time -- 2.500000
Average Turnaround Time -- 5.000000
```

3. FCFS : different arrival time

Code:

```
#include <iostream>
using namespace std;

// Function to find the waiting time for all processes
void findWaitingTime(int processes[], int n, int bt[], int wt[], int at[])
{
   int service_time[n];
   service_time[0] = at[0];
   wt[0] = 0;
   // calculating waiting time
   for (int i = 1; i < n; i++) {
       // Add burst time of previous processes
       service_time[i] = service_time[i - 1] + bt[i - 1];
       // Find waiting time for current process =
       // sum - at[i]
       wt[i] = service_time[i] - at[i];
       // If waiting time for a process is in negative
       // that means it is already in the ready queue
       // before CPU becomes idle so its waiting time is 0
       if (wt[i] < 0)
           wt[i] = 0;
   }
}
// Function to calculate turn around time
void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int
tat[]){
   // Calculating turnaround time by adding bt[i] + wt[i]
   for (int i = 0; i < n; i++)
       tat[i] = bt[i] + wt[i];
}
// Function to calculate average waiting and turn-around times.
void findavgTime(int processes[], int n, int bt[], int at[]){
   int wt[n], tat[n];
   // Function to find waiting time of all processes
   findWaitingTime(processes, n, bt, wt, at);
   // Function to find turn around time for all processes
```

```
    findTurnAroundTime(processes, n, bt, wt, tat);
    // Display processes along with all details
    cout << "Processes "<< " Burst Time " << " Arrival Time " << " Waiting
Time " << " Turn-Around Time " << " Completion Time \n";
    int total_wt = 0, total_tat = 0;
    for (int i = 0; i < n; i++)  {
        total_wt = total_wt + wt[i];
        total_tat = total_tat + tat[i];
        int compl_time = tat[i] + at[i];
        cout << " " << i + 1 << "\t\t" << bt[i] << "\t\t"
             << at[i] << "\t\t" << wt[i] << "\t\t "
             << tat[i] << "\t\t " << compl_time << endl;
    }
    cout << "Average waiting time = "  << (float)total_wt / (float)n;
    cout << "\nAverage turn around time = "  << (float)total_tat /
(float)n;
}

// Driver code
int main(){
    int n;
    cout << "Enter no of processes: ";
    cin >> n;
    cout << endl;
    int processes[n];
    for (int i = 0; i < n; i++)
        processes[i] = i;
    // Burst Time for processes
    int burst_time[n];
    for (int i = 0; i < n; i++) {
        cout << "Enter burst time for process " << i << ": ";
        cin >> burst_time[i];
        cout << endl;
    } cout << endl;
    // Arrival Time for processes
    int arrival_time[n];
    for (int i = 0; i < n; i++) {
        cout << "Enter arrival time for process " << i << ": ";
        cin >> arrival_time[i];
        cout << endl;
```

```
    }
    findavgTime(processes, n, burst_time, arrival_time);
    return 0;
}
```

Output:

```
Enter no of processes: 4

Enter burst time for process 0: 2

Enter burst time for process 1: 1

Enter burst time for process 2: 3

Enter burst time for process 3: 2


Enter arrival time for process 0: 3

Enter arrival time for process 1: 2

Enter arrival time for process 2: 1

Enter arrival time for process 3: 3

Processes  Burst Time  Arrival Time  Waiting Time  Turn-Around Time  Completion Time
 1              2              3              0                2              5
 2              1              2              3                4              6
 3              3              1              5                8              9
 4              2              3              6                8              11
Average waiting time = 3.5
Average turn around time = 5.5
```

4. SJF : different arrival time

Code:

```
#include <iostream>
using namespace std;
int mat[10][6];
void swap(int *a, int *b){
   int temp = *a;
   *a = *b;
   *b = temp;
}

void arrangeArrival(int num, int mat[][6]){
   for (int i = 0; i < num; i++) {
       for (int j = 0; j < num - i - 1; j++)    {
           if (mat[j][1] > mat[j + 1][1])   {
               for (int k = 0; k < 5; k++)
                swap(mat[j][k], mat[j + 1][k]);
           }
       }
   }
}
void completionTime(int num, int mat[][6]){
   int temp, val;
   mat[0][3] = mat[0][1] + mat[0][2];
   mat[0][5] = mat[0][3] - mat[0][1];
   mat[0][4] = mat[0][5] - mat[0][2];
   for (int i = 1; i < num; i++){
       temp = mat[i - 1][3];
       int low = mat[i][2];
       for (int j = i; j < num; j++)  {
           if (temp >= mat[j][1] && low >= mat[j][2])   {
               low = mat[j][2];
               val = j;
           }
       }
       mat[val][3] = temp + mat[val][2];
       mat[val][5] = mat[val][3] - mat[val][1];
       mat[val][4] = mat[val][5] - mat[val][2];
```

```
        for (int k = 0; k < 6; k++) swap(mat[val][k], mat[i][k]);
    }
}
int main(){
    int num, temp;
    cout << "Enter number of Process: ";
    cin >> num;
    cout << "...Enter the process ID...\n";
    for (int i = 0; i < num; i++)
    {
        cout << "...Process " << i + 1 << "...\n";
        cout << "Enter Process Id: ";
        cin >> mat[i][0];
        cout << "Enter Arrival Time: ";
        cin >> mat[i][1];
        cout << "Enter Burst Time: ";
        cin >> mat[i][2];
    }

    cout << "Before Arrange...\n";
    cout << "Process ID\tArrival Time\tBurst Time\n";
    for (int i = 0; i < num; i++)
    {
        cout << mat[i][0] << "\t\t" << mat[i][1] << "\t\t"
             << mat[i][2] << "\n";
    }

    arrangeArrival(num, mat);
    completionTime(num, mat);
    cout << "Final Result...\n";
    cout << "Process ID\tArrival Time\tBurst Time\tWaiting "
            "Time\tTurnaround Time\n";
    for (int i = 0; i < num; i++)
    {
        cout << mat[i][0] << "\t\t" << mat[i][1] << "\t\t"
             << mat[i][2] << "\t\t" << mat[i][4] << "\t\t"
             << mat[i][5] << "\n";
    }
}
```

Output:

```
Enter number of Process: 4
...Enter the process ID...
...Process 1...
Enter Process Id: 2
Enter Arrival Time: 1
Enter Burst Time: 2
...Process 2...
Enter Process Id: 3
Enter Arrival Time: 1
Enter Burst Time: 3
...Process 3...
Enter Process Id: 4
Enter Arrival Time: 2
Enter Burst Time: 1
...Process 4...
Enter Process Id: 3
Enter Arrival Time: 5
Enter Burst Time: 2
Before Arrange...
Process ID      Arrival Time    Burst Time
2               1               2
3               1               3
4               2               1
3               5               2
Final Result...
Process ID      Arrival Time    Burst Time      Waiting Time    Turnaround Time
2               1               2               0               2
4               2               1               1               2
3               1               3               3               6
3               5               2               2               4
```

5. SJF: using heap

Code:

```
#include <iostream>
#include <algorithm>
using namespace std;
void heapify(int arr[], int n, int i, int t[]){
   int largest = i;
   int l = 2 * i + 1;
   int r = 2 * i + 2;
   if (l == n && arr[l] > arr[largest]) largest = l;
   if (r < n && arr[r] > arr[largest])  largest = r;
   if (largest != i) {
       swap(arr[i], arr[largest]);
       swap(t[i], t[largest]);
       heapify(arr, n, largest, t);
   }
}
void heapSort(int arr[], int n, int t[]){
   for (int i = n / 2; i >= 0; i++){
       heapify(arr, n, i, t);
   }
   for (int i = n - 1; i > 0; i--){
       swap(arr[0], arr[i]);
       swap(t[i], t[0]);
       heapify(arr, i, 0, t);
   }
}
void printArray(int arr[], int n){
  for (int i = 0; i < n; ++i)
       cout << arr[i] << " ";
      cout << "/n";
}
int main(){
  int p[20], bt[20], wt[20], tat[20], i, k, n, temp;
   float cwt, ctat;
   printf("\n Enter the number of Process : ");
   scanf("%d", &n);
   for (i = 0; i < n; i++){
```

```
        p[i] = i;
        printf("Enter Burst time for processe %d : ", i);
        scanf("%d", &bt[i]);
    }
    heapSort(bt, n, p);
    wt[0] = cwt = 0;
    tat[0] = ctat = bt[0];
   for (int i = 1; i < n; i++){
        wt[i] = wt[i - 1] + bt[i];
        tat[i] = tat[i - 1] + bt[i];
        cwt = cwt + wt[i];
        ctat = ctat + tat[i];
    }
    printf("\n\tProcess\tBurst Time \t Waiting Time \t Turnaround Time\n");
    for (int i = 0; i < n; i++){
        printf("\n\tP%d\t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i],
tat[i]);
        printf("\nAverage waiting time : %f", cwt /n);
        printf("\nAverage Turaround time : %f", ctat /n);
    }
}
```

Output:

```
Enter the number of processes -- 4
Enter Burst Time for Process 0 -- 2
Enter Burst Time for Process 1 -- 3
Enter Burst Time for Process 2 -- 1
Enter Burst Time for Process 3 -- 4

         PROCESS        BURST TIME       WAITING TIME    TURNAROUND TIME

         P2              1               0               1
         P0              2               1               3
         P1              3               3               6
         P3              4               6               10
Average Waiting Time -- 2.500000
```

Lab Experiment No. 3: Implementation of Round Robin & Priority CPU Scheduling Algorithms

Aim: To get the idea of how logics for Round Robin & Priority CPU Scheduling Algorithms are developed.

Round Robin Background:

- Criteria: Arrival Time
- Mode: Preemptive
- Process runs for a given Time Quantum (TQ) and preempts and goes to the end of RQ
- Practically Implementable, as AT in RQ.
- Using Queues, RR can be implemented, unlike of complex data structure like Heaps.
- No starvation, as no process is going to wait for CPU forever.
- Every process gets a chance after some amount of time and keep on getting chance after some time.
-

1. RR with same arrival time:

Round Robin is a CPU scheduling algorithm where each process is assigned a fixed time slot in a cyclic way. It is simple, easy to implement, and starvation-free as all processes get fair share of CPU. One of the most commonly used technique in CPU scheduling as a core. It is preemptive as processes are assigned CPU only for a fixed slice of time at most.

Code:

```
#include<stdio.h>

int main()
{ int i,j,n,bt[10],ibt[10],wt[10],tat[10],t,max,p[50], g=0;
float cwt=0,ctat=0,temp=0;
printf("Enter the no of processes -- ");
 scanf("%d",&n);
  for(i=0;i<n;i++) {
  printf("\nEnter Burst Time for process %d -- ", i);
  scanf("%d",&bt[i]);
  ibt[i]=bt[i];
  }

  printf("\nEnter the size of time slice -- ");
  scanf("%d",&t);
  max=bt[0];

  for(i=1;i<n;i++) {
 if(max<bt[i])
   max=bt[i];
  }

  for(j=0;j<(max/t)+1;j++)
  for(i=0;i<n;i++)
  if(bt[i]!=0){
  if(bt[i]<=t){
    tat[i]=temp+bt[i];
    temp=temp+bt[i];
    bt[i]=0;
    p[g++]=i;
   }
  else{
    bt[i]=bt[i]-t;
    temp=temp+t;
    p[g++]=i;
   }
}
```

```
  for(i=0;i<n;i++)
   {
     wt[i]=tat[i]-ibt[i];
     ctat+=tat[i];
     cwt+=wt[i];
   }
printf("\nThe Average Turnaround time is -- %f",ctat/n);
printf("\nThe Average Waiting time is -- %f ",cwt/n);

printf("\n");
for(i=0;i<g;i++){
printf("pid - >%d --",p[i]);
}

printf("\n\tPROCESS\t BURST TIME \t WAITING TIME\tTURNAROUND TIME\n");
for(i=0;i<n;i++)
 printf("\t%d \t %d \t\t %d \t\t %d \n",i,ibt[i],wt[i],tat[i]);
return 0;
}
```

Output:

```
Enter the no of processes -- 4

Enter Burst Time for process 0 -- 2

Enter Burst Time for process 1 -- 1

Enter Burst Time for process 2 -- 4

Enter Burst Time for process 3 -- 3

Enter the size of time slice -- 2

The Average Turnaround time is -- 6.000000
The Average Waiting time is -- 3.500000
pid - >0 --pid - >1 --pid - >2 --pid - >3 --pid - >2 --pid - >3 --
        PROCESS  BURST TIME      WAITING TIME   TURNAROUND TIME
        0        2              0               2
        1        1              2               3
        2        4              5               9
        3        3              7               10
```

2. Priority scheduling: same arrival time

Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems. Each process is assigned a priority. Process with highest priority is to be executed first and so on. Processes with same priority are executed on first come first served basis.

Implementation –

- First input the processes with their arrival time, burst time and priority.
- First process will schedule, which have the lowest arrival time, if two or more processes will have lowest arrival time, then whoever has higher priority will schedule first.
- Now further processes will be schedule according to the arrival time and priority of the process. (Here we are assuming that lower the priority number having higher priority).
- If two process priority are same then sort according to process number.
- They will clearly mention, which number will have higher priority and which number will have lower priority. Once all the processes have been arrived.

Code:

```
#include<stdio.h>
int main()
{
int p[20],bt[20],pri[20], wt[20],tat[20],i, k, n, temp;
float cwt, ctat;
printf("Enter the number of processes --- ");
scanf("%d",&n);
for(i=0;i<n;i++)
{
p[i] = i;
printf("Enter the Burst Time & Priority of Process %d --- ",i);
scanf("%d %d",&bt[i], &pri[i]);
}
for(i=0;i<n;i++)
 for(k=i+1;k<n;k++)
  if(pri[i] > pri[k])
   { temp=p[i];
    p[i]=p[k];
    p[k]=temp;
     temp=bt[i];
```

```
    bt[i]=bt[k];
    bt[k]=temp;
     temp=pri[i];
    pri[i]=pri[k];
    pri[k]=temp;
    }
   cwt = wt[0] = 0;
   ctat = tat[0] = bt[0];
for(i=1;i<n;i++)
{
wt[i] = wt[i-1] + bt[i-1];
tat[i] = tat[i-1] + bt[i];
cwt = cwt + wt[i];
ctat = ctat + tat[i];
 }
 printf("\nPROCESS\t\tPRIORITY\tBURST TIME\tWAITING TIME\tTURNAROUND
TIME");
for(i=0;i<n;i++)
printf("\n%d \t\t %d \t\t %d \t\t %d \t\t %d
",p[i],pri[i],bt[i],wt[i],tat[i]);
printf("\nAverage Waiting Time is --- %f",cwt/n);
printf("\nAverage Turnaround Time is --- %f",ctat/n);
return 0;
}
```

Output:

```
Enter the number of processes --- 4
Enter the Burst Time & Priority of Process 0 --- 4 3
Enter the Burst Time & Priority of Process 1 --- 1 4
Enter the Burst Time & Priority of Process 2 --- 3 1
Enter the Burst Time & Priority of Process 3 --- 5 2

PROCESS         PRIORITY        BURST TIME      WAITING TIME    TURNAROUND TIME
2                1               3               0               3
3                2               5               3               8
0                3               4               8               12
1                4               1               12              13
Average Waiting Time is --- 5.750000
Average Turnaround Time is --- 9.000000
```

3. RR: different arrival time

Code:

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

```
    wait_time+=time-at[count]-bt[count];
    turnaround_time+=time-at[count];
    flag=0;
  }
  if(count==n-1)
    count=0;
  else if(at[count+1]<=time)
    count++;
  else
    count=0;
 }
 printf("\nAverage Waiting Time= %f\n",wait_time*1.0/n);
 printf("Avg Turnaround Time = %f",turnaround_time*1.0/n);
  return 0;
}
```

Output:

```
Enter Total Process:     4
Enter Arrival Time and Burst Time for Process Process Number 1 :1 3
Enter Arrival Time and Burst Time for Process Process Number 2 :2 4
Enter Arrival Time and Burst Time for Process Process Number 3 :4 3
Enter Arrival Time and Burst Time for Process Process Number 4 :3 3
Enter Time Quantum:     2


Process |Turnaround Time|Waiting Time

P[1]    |       8       |       5
P[2]    |       9       |       5
P[3]    |       8       |       5
P[4]    |       10      |       7

Average Waiting Time= 5.500000
Avg Turnaround Time = 8.750000
```

4. Priority scheduling: different arrival time

Code:

```
#include<stdio.h>
int main(){
   Int bt[20], p[20], wt[20], tat[20], pr[20], i,j,n, total=0,
pos ,temp,avg_wt,avg_tat;
   printf("Enter Total Number of Process:");
   scanf("%d",&n);
   printf("\nEnter Burst Time and Priority\n");
   for(i=0;i<n;i++){
       printf("\nP[%d]\n",i+1);
       printf("Burst Time:");
       scanf("%d",&bt[i]);
       printf("Priority:");
       scanf("%d",&pr[i]);
       p[i]=i+1;           //contains process number
   }
   //sorting burst time, priority and process number in ascending order
using selection sort
   for(i=0;i<n;i++){
       pos=i;
       for(j=i+1;j<n;j++){
           if(pr[j]<pr[pos])
               pos=j;
       }
       temp=pr[i];
       pr[i]=pr[pos];
       pr[pos]=temp;
        temp=bt[i];
       bt[i]=bt[pos];
       bt[pos]=temp;
       temp=p[i];
       p[i]=p[pos];
       p[pos]=temp;
   } wt[0]=0;    //waiting time for first process is zero
   //calculate waiting time
   for(i=1;i<n;i++){
       wt[i]=0;
```

```
        for(j=0;j<i;j++)
            wt[i]+=bt[j];
        total+=wt[i];
    }
    avg_wt=total/n;      //average waiting time
    total=0;
    printf("\nProcess\t    Burst Time    \tWaiting Time\tTurnaround Time");
    for(i=0;i<n;i++){
        tat[i]=bt[i]+wt[i];     //calculate turnaround time
        total+=tat[i];
        printf("\nP[%d]\t\t  %d\t\t
%d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);
    }
    avg_tat=total/n;     //average turnaround time
    printf("\n\nAverage Waiting Time=%d",avg_wt);
    printf("\nAverage Turnaround Time=%d\n",avg_tat);
  return 0;
}
```

Output:

```
Enter Total Number of Process:4

Enter Burst Time and Priority

P[1]
Burst Time:2 1
Priority:
P[2]
Burst Time:4 3
Priority:
P[3]
Burst Time:2 2
Priority:
P[4]
Burst Time:5 4
Priority:
Process     Burst Time          Waiting Time    Turnaround Time
P[1]              2                   0                   2
P[3]              2                   2                   4
P[2]              4                   4                   8
P[4]              5                   8                   13

Average Waiting Time=3
Average Turnaround Time=6
```

5. Premptive scheduling using min heap

Code:

```
#include<stdio.h>
struct process
{
     char process_name;
     int arrival_time, burst_time, ct, waiting_time, turnaround_time,
priority;
     int status;
}process_queue[10];
int limit;
void Arrival_Time_Sorting()
{
     struct process temp;
     int i, j;
     for(i = 0; i < limit - 1; i++)
     {
           for(j = i + 1; j < limit; j++)
           {
                 if(process_queue[i].arrival_time >
process_queue[j].arrival_time)
                 {
                       temp = process_queue[i];
                       process_queue[i] = process_queue[j];
                       process_queue[j] = temp;
                 }
           }
     }
}
int main()
{
     int i, time = 0, burst_time = 0, largest;
     char c;
     float wait_time = 0, turnaround_time = 0, average_waiting_time,
average_turnaround_time;
     printf("\nEnter Total Number of Processes:\t");
     scanf("%d", &limit);
     for(i = 0, c = 'A'; i < limit; i++, c++)
```

```
    {
          process_queue[i].process_name = c;
          printf("\nEnter Details For Process[%C]:\n",
process_queue[i].process_name);
          printf("Enter Arrival Time:\t");
          scanf("%d", &process_queue[i].arrival_time );
          printf("Enter Burst Time:\t");
          scanf("%d", &process_queue[i].burst_time);
          printf("Enter Priority:\t");
          scanf("%d", &process_queue[i].priority);
           process_queue[i].status = 0;
          burst_time = burst_time + process_queue[i].burst_time;
    }
    Arrival_Time_Sorting();
    process_queue[9].priority = -9999;
    printf("\nProcess Name\tArrival Time\tBurst Time\tPriority\tWaiting
Time");
    for(time = process_queue[0].arrival_time; time < burst_time;)
    {
          largest = 9;
          for(i = 0; i < limit; i++)
          {
                if(process_queue[i].arrival_time <= time &&
process_queue[i].status != 1 &&
                           process_queue[i].priority >
process_queue[largest].priority)
                {
                      largest = i;
                }
          }
          time = time + process_queue[largest].burst_time;
          process_queue[largest].ct = time;
          process_queue[largest].waiting_time = process_queue[largest].ct
-
                       process_queue[largest].arrival_time -
process_queue[largest].burst_time;
          process_queue[largest].turnaround_time =
process_queue[largest].ct - process_queue[largest].arrival_time;
          process_queue[largest].status = 1;
          wait_time = wait_time + process_queue[largest].waiting_time;
```

```
            turnaround_time = turnaround_time +
process_queue[largest].turnaround_time;
            printf("\n%c\t\t%d\t\t%d\t\t%d\t\t%d",
process_queue[largest].process_name,
                    process_queue[largest].arrival_time,
process_queue[largest].burst_time,
                        process_queue[largest].priority,
process_queue[largest].waiting_time);
      }
      average_waiting_time = wait_time / limit;
      average_turnaround_time = turnaround_time / limit;
      printf("\n\nAverage waiting time:\t%f\n", average_waiting_time);
      printf("Average Turnaround Time:\t%f\n", average_turnaround_time);
return 0;
}
```

Output:

```
Enter Total Number of Processes:        4

Enter Details For Process[A]:
Enter Arrival Time:     3
Enter Burst Time:       2
Enter Priority: 4

Enter Details For Process[B]:
Enter Arrival Time:     1
Enter Burst Time:       2
Enter Priority: 3

Enter Details For Process[C]:
Enter Arrival Time:     2
Enter Burst Time:       2
Enter Priority: 3

Enter Details For Process[D]:
Enter Arrival Time:     1
Enter Burst Time:       1
Enter Priority: 1

Process Name    Arrival Time    Burst Time      Priority        Waiting Time
B               1               2               3               0
A               3               2               4               0
C               2               2               3               3

Average waiting time:   0.750000
Average Turnaround Time:        2.250000
```

Lab Experiment No. 4: Implementation of Multilevel Queue Scheduling Algorithm

Aim: To get the idea of how logic for Multilevel Queue Scheduling is developed.

MLQ Background:

- Ready queue is partitioned into separate queues, e.g.:
    - **System Processes RQ**
    - **User Processes RQ**
- Processes doesn't change queues and remain in the given queues.
- Each queue has its own scheduling algorithm:
    - System Process Ready Queue – SJF
    - User Process Ready Queue – FCFS
- NOTE: Implementation of Scheduling Algorithms for different Queue depends upon the application requirements. Ex: foreground processes can have RR and background processes can have FCFS.
- Scheduling must be done between the queues:
    - Fixed priority scheduling; (i.e., serve all from higher priority queue first than from lower priority queue). Possibility of starvation.
    - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes.

MLQ Code (for same arrival time):

- Lower Queue Priority (QP) number means higher queue priority.
- System processes are assigned to a queue having QP=0 and called as System Process Ready Queue.
- User processes are assigned to a queue having QP=1 and called as User Process Ready Queue.
- System Process Ready Queue implements SJF
- User Process Ready Queue implements FCFS

TASK 1 : i:- Multilevel queue where every process have same arrival time and user process have less priority then system process. Implement FCFS for user process and SJF for system process.

Code:

```
#include<stdio.h>
void main()
{
// qp is Queue Prioriy, cwt is Cummulative Waiting Time, and ctat is
Cummulative Turn Around Time
int pid[20],bt[20], qp[20], wt[20],tat[20],i, k, n, temp;
float cwt, ctat;
printf("Enter the number of processes --- ");
scanf("%d",&n);
for(i=0;i<n;i++)
{
   pid[i] = i;
   printf("Enter the Burst Time of Process %d --- ", i);
scanf("%d",&bt[i]);
   printf("System/User Process (0/1) ? --- ");
   scanf("%d", &qp[i]);
}


for(i=0;i<n;i++)
for(k=i+1;k<n;k++)
   if(qp[i] > qp[k])
   {
       temp=pid[i];
       pid[i]=pid[k];
       pid[k]=temp;

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

       temp=qp[i];
       qp[i]=qp[k];
       qp[k]=temp;
   }


for(i=0;i<n;i++)
```

```
for(k=i+1;k<n;k++)
   {
       if(qp[i]==0 && qp[k]==0 && bt[i]>bt[k])
       {
           temp=pid[i];
           pid[i]=pid[k];
           pid[k]=temp;

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

           temp=qp[i];
           qp[i]=qp[k];
           qp[k]=temp;
       }

else if (qp[i]==1 && qp[k]==1 && pid[i]>pid[k])
       {
           temp=pid[i];
           pid[i]=pid[k];
           pid[k]=temp;

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

           temp=qp[i];
           qp[i]=qp[k];
           qp[k]=temp;
       }

   }


cwt = wt[0] = 0;
ctat = tat[0] = bt[0];
for(i=1;i<n;i++)
{
   wt[i] = wt[i-1] + bt[i-1];
```

```
   tat[i] = tat[i-1] + bt[i];

   cwt = cwt + wt[i];
   ctat = ctat + tat[i];
}
printf("\nPROCESS\t\t SYSTEM/USER PROCESS \tBURST TIME\tWAITING
TIME\tTURNAROUND TIME");
for(i=0;i<n;i++)
   printf("\n%d \t\t\t\t %d \t\t %d \t\t %d \t\t %d
",pid[i],qp[i],bt[i],wt[i],tat[i]);

printf("\nAverage Waiting Time is --- %f",cwt/n);
printf("\nAverage Turnaround Time is --- %f",ctat/n);
}
```

Output:

```
Enter the number of processes --- 5
Enter the Burst Time of Process 0 --- 2
System/User Process (0/1) ? --- 0
Enter the Burst Time of Process 1 --- 3
System/User Process (0/1) ? --- 1
Enter the Burst Time of Process 2 --- 4
System/User Process (0/1) ? --- 0
Enter the Burst Time of Process 3 --- 3
System/User Process (0/1) ? --- 1
Enter the Burst Time of Process 4 --- 9
System/User Process (0/1) ? --- 0

PROCESS          SYSTEM/USER PROCESS    BURST TIME      WAITING TIME    TURNAROUND TIME
0                                0              2               0               2
2                                0              4               2               6
4                                0              9               6               15
1                                1              3               15              18
3                                1              3               18              21
Average Waiting Time is --- 8.200000
Average Turnaround Time is --- 12.400000
```

ii. Multilevel queue where every process have different arrival time and user process have less priority then system process. Implement FCFS for user process and SJF for system process.

Code :

```
#include<bits/stdc++.h>
using namespace std;

int main()
{
   int n;
   printf("Enter the number of process u want:\t");
   cin>>n;

   int AT[n],type[n],BT[n];
   for(int i=0;i<n;i++){
       printf("Enter the Arrival time of the process:\t");
       cin>>AT[i];
       printf("Enter the Burst time of the process: \t");
       cin>>BT[i];
       printf("Enter the type of the process: 0/1 system/user\t");
       cin>>type[i];
   }

   int o_tym=-1,tym=0,RQ[4][n],count=0,process_done=0;
   float cwt=0,ctat=0;
       printf("Process id\t Arrival Time\t Burst Time\t Waiting Time\t
Turn around time\n");
   while(1){
       for(int i=0;i<n;i++){
           if(AT[i]>o_tym && AT[i]<=tym){
               RQ[0][count]=AT[i];
               RQ[1][count]=type[i];
               RQ[2][count]=i+1;
               RQ[3][count]=BT[i];
               count++;
               // cout<<"count="<<i+1<<" ";
           }
       }
       o_tym=tym;
```

```
        int min=INT_MAX,index=-1;
        for(int i=0;i<count;i++){
            if(RQ[3][i]<min && RQ[1][i]==0 && RQ[3][i]>0){
                min=RQ[3][i];
                // RQ[3][i]=-1;
                index=i;
            }
        }


        if(index!=-1){
            process_done++;
            tym+=RQ[3][index];
            ctat=tym -RQ[0][index];
            cwt=ctat - RQ[3][index];
            cout<<RQ[2][index]<<"\t\t  "<<RQ[0][index]<<"\t\t "<<min<<"\t\t
"<<cwt<<"\t\t  "<<ctat<<endl;
            // o_tym=tym;
            RQ[3][index]=-1;
        }else if(process_done<count){
            int min=INT_MAX,index=-1;
            for(int i=0;i<count;i++){
                // cout<<i<<endl;
                // cout<<"The vlaue of RQ[0][3] is: "<<RQ[0][3]<<endl;
                if(RQ[3][i]>0 && RQ[1][i]==1 && RQ[0][i]<min){
                    min=RQ[0][i];
                    index=i;
                    // cout<<index<<endl;
                }
            }
            process_done++;
            tym+=RQ[3][index];
            ctat=tym - RQ[0][index];
            cwt=ctat-RQ[3][index];
            cout<<RQ[2][index]<<"\t\t "<<RQ[0][index]<<"\t\t "<<RQ[3]
[index]<<"  \t\t"<<cwt<<"  \t\t"<<ctat<<endl;
            // cwt+=RQ[3][index];
            // o_tym=tym;
            RQ[3][index]=-1;
        }else{
```

```
            tym++;
            // cwt++;
        }
        if(process_done==n){
            break;
        }

            // cwt+=min;
    }

    return 0;
}
```

Output:

```
Enter the number of process u want:     2
Enter the Arrival time of the process:  0
Enter the Burst time of the process:    2
Enter the type of the process: 0/1 system/user  1
Enter the Arrival time of the process:  2
Enter the Burst time of the process:    3
Enter the type of the process: 0/1 system/user  0
Process id       Arrival Time    Burst Time      Waiting Time    Turn around tim
e
1                0               2              0               2
2                 2              3                0              3
```

Task 2. Multilevel Feedback queue with time counter for each queue.

Code:

```
#include<bits/stdc++.h>
using namespace std;

int main()
{
   int n;
   printf("Enter the number of process u want:\t");
   cin>>n;

   int type[n],BT[n];
   for(int i=0;i<n;i++){
       printf("Enter the Burst time of the process: \t");
       cin>>BT[i];
       printf("Enter the type of the process: 0/1 system/user\t");
       cin>>type[i];
   }
    int RQ[3][n],count=0,process_done=0,s_count=0,u_count=0;
   float cwt=0,ctat=0;
   printf("Process id\t Burst Time\t Waiting Time\t Turn around time\n");
       for(int i=0;i<n;i++){
           if(BT[i]<=6 && type[i]==0){
               RQ[0][count]=BT[i];  //Burst time
               // RQ[0][count]=AT[i];
               RQ[1][count]=type[i]; //Process type
               RQ[2][count]=i+1;    //Pid
               count++;
               s_count++;
               // cout<<"count="<<i+1<<" ";
           }else if(BT[i]<=10 && type[i]==1){
               RQ[0][count]=BT[i];
               // RQ[0][count]=AT[i];
               RQ[1][count]=type[i];
               RQ[2][count]=i+1;
               count++;
               u_count++;
           }
```

```
        }

            int p=0;
   while(1){
            // o_tym=tym
        int min=INT_MAX,index=-1;
        if(p==0){
            if(p==0 && s_count>0){
            for(int i=0;i<count;i++){
            if(RQ[0][i]<min && RQ[1][i]==0 && RQ[0][i]>0){
                min=RQ[0][i];
                // RQ[3][i]=-1;
                index=i;
        }
            }
           if(index!=-1){
            ctat+=min;
            process_done++;
            cout<<RQ[2][index]<<"\t\t  "<<min<<"\t\t  "<<cwt<<"\t\t
"<<ctat<<endl;
            cwt+=RQ[0][index];
            // o_tym=tym;
            // tym+=RQ[0][index];
            RQ[0][index]=-1;
            p=1;
        }
        s_count--;
            }else{
                p=1;
            }
        }else if(p==1 && u_count>0){
            {
                u_count--;
            p=0;
            for(int i=0;i<count;i++){
                // cout<<i<<endl;
                // cout<<"The vlaue of RQ[0][3] is: "<<RQ[0][3]<<endl;
                if(RQ[0][i]>0 && RQ[1][i]==1){
                    min=RQ[0][i];
                    index=i;
```

```
                    break;
                    // cout<<index<<endl;
                }
            }
            ctat+=RQ[0][index];
            process_done++;
            cout<<RQ[2][index]<<"\t\t "<<RQ[0][index]<<"   \t\t"<<cwt<<"
\t\t"<<ctat<<endl;
            cwt+=RQ[0][index];
            // o_tym=tym;
            // tym+=RQ[0][index];
            RQ[0][index]=-1;
            }
            }else{
            if(p==0)
            p=1;
            else
            p=0;
        }       if(process_done==count)    break;// cwt+=min;
    }
 return 0;
}
```

Output:

```
Enter the number of process u want:     4
Enter the Burst time of the process:    2
Enter the type of the process: 0/1 system/user  1
Enter the Burst time of the process:    1
Enter the type of the process: 0/1 system/user  0
Enter the Burst time of the process:    3
Enter the type of the process: 0/1 system/user  1
Enter the Burst time of the process:    3
Enter the type of the process: 0/1 system/user  1
Process id       Burst Time      Waiting Time    Turn around time
2                 1               0               1
1                2              1               3
3                3              3               6
4                3              6               9
```

Lab Experiment No. 5: Implementation of Synchronized Producer Consumer using Semaphores and Mutex through Threads

Aim: To get the idea of how Semaphores and Mutex are used for synchronizing a critical section.

Producer-Consumer Background:

- Sleep and Wake are the system calls.
- When a process calls sleep( ), it get blocked.
- When a process calls wakeup( ), one of the blocked processes get awake up.
- *Producer* process produces or adding/writing something in buffer.
- *Consumer* process consumes or deleting/reading something from buffer.
- Buffer is a memory
- In Sleep and Wake solution, we came across that if a process is not sleeping and if another process is trying to wake him up, wakeup signal goes lost.
- Instead of using that, Dijkstra has proposed that no need to send the wakeup signal to sleeping process, just write the signal in variable, so that it will be stored and be there, so anyone can use it, whenever wanted (go to sleep).
- If Semaphore is implemented at user-mode, then two processes can access it simultaneously to add a wakeup call and Semaphore value become inconsistent and inefficient.
- Therefore, most of OS provides some primitives (function calls), use to access shared variables (Semaphores) atomically.
    - Means, if a process is reading Semaphore, all other processes stop using it.
    - Means, Read, Update and Write is used atomically.
- Hence, to need atomicity, we need OS support in Kernel-mode.
- Variables on which Read, Modify and Update happens atomically in Kernel-mode (means no preemption).
- Types of Semaphores
    1. Counting Semaphores
    2. Binary Semaphores (Mutex)
- Counting Semaphore variable shows how many processes can enter into CS simultaneously.
- Suppose a resource i.e. printer, can be shared between 4 processes but not more than 4 processes. Then, Counting Semaphore=4.
- If, Semaphore = + N (means N processes can enter simultaneously in CS at present).
- If, Semaphore = - Z (means Z processes tried to enter in fully occupied CS and get blocked. Now, they are in WAITING Queue).
- The difference between counting and binary semaphore is,

- In counting semaphore, we counts the value of semaphore as the number of processes allowed to be in CS at any time (+ve value) and processes in BLOCKED Queue at any time (-ve value).
- But, binary semaphores are not used for counting, they are just used for entering or exiting a process in and from CS in ME manner.

1. Problem of Synchronization using Threads

1.1. Code without any pre-emption

Code:

```
#include<pthread.h>
#include<stdio.h>
#include<unistd.h>
void *fun1();
void *fun2();

int shared=1; //shared variable

int main()
{
pthread_t thread1, thread2;
pthread_create(&thread1, NULL, fun1, NULL);
pthread_create(&thread2, NULL, fun2, NULL);
pthread_join(thread1, NULL);
pthread_join(thread2,NULL);
printf("Final value of shared is %d\n",shared); //prints the last
updated value of shared variable
}

void *fun1()
{
   int x;
   x=shared;//thread1 reads value of shared variable
   printf("Thread1 reads the value as %d\n",x);
   x++;  //thread1 increments its value
   printf("Local updation by Thread1: %d\n",x);
```

```
    shared=x; //thread one updates the value of shared variable
        printf("Value of shared variable updated by Thread1 is:
%d\n",shared);
}
void *fun2()
{
   int y;
   y=shared;//thread2 reads value of shared variable
   printf("Thread2 reads the value as %d\n",y);
   y--;  //thread2 increments its value
   printf("Local updation by Thread2: %d\n",y);
   shared=y; //thread2 updates the value of shared variable
        printf("Value of shared variable updated by Thread2 is:
%d\n",shared);
}
```

Output:

```
Thread1 reads the value as 1
Local updation by Thread1: 2
Value of shared variable updated by Thread1 is: 2
Thread2 reads the value as 1
Local updation by Thread2: 0
Value of shared variable updated by Thread2 is: 0
Final value of shared is 0
```

1.2. Code with pre-emption

Code:

```
#include<pthread.h>
#include<stdio.h>
#include<unistd.h>
void *fun1();
void *fun2();
int shared=1; //shared variable
int main()
{

pthread_t thread1, thread2;
pthread_create(&thread1, NULL, fun1, NULL);
pthread_create(&thread2, NULL, fun2, NULL);
pthread_join(thread1, NULL);
pthread_join(thread2,NULL);
printf("Final value of shared is %d\n",shared); //prints the last
updated value of shared variable
}
void *fun1()
{
   int x;
   x=shared;//thread1 reads value of shared variable
   printf("Thread1 reads the value as %d\n",x);
   x++;  //thread1 increments its value
   printf("Local updation by Thread1: %d\n",x);
   sleep(1); //thread1 is preempted by thread 2
   shared=x; //thread one updates the value of shared variable
       printf("Value of shared variable updated by Thread1 is:
%d\n",shared);

}
void *fun2()
{
   int y;
   y=shared;//thread2 reads value of shared variable
   printf("Thread2 reads the value as %d\n",y);
```

```
    y--;  //thread2 increments its value
    printf("Local updation by Thread2: %d\n",y);
    sleep(1); //thread2 is preempted by thread 1
    shared=y; //thread2 updates the value of shared variable
        printf("Value of shared variable updated by Thread2 is:
%d\n",shared);

}
```

Output:

```
Thread1 reads the value as 1
Local updation by Thread1: 2
Thread2 reads the value as 1
Local updation by Thread2: 0
Value of shared variable updated by Thread1 is: 2
Value of shared variable updated by Thread2 is: 0
Final value of shared is 0
```

2. **Synchronization in Threads using Semaphores**

Code:

```
#include<pthread.h>
#include<stdio.h>
#include<semaphore.h>
#include<unistd.h>
void *fun1();
void *fun2();
int shared=1; //shared variable
sem_t s; //semaphore variable
int main()
{
sem_init(&s,0,1);
pthread_t thread1, thread2;
pthread_create(&thread1, NULL, fun1, NULL);
pthread_create(&thread2, NULL, fun2, NULL);
pthread_join(thread1, NULL);
pthread_join(thread2,NULL);
printf("Final value of shared is %d\n",shared); //prints the last
updated value of shared variable
}
void *fun1()
{
   int x;
   sem_wait(&s); //executes wait operation on s
   x=shared;//thread1 reads value of shared variable
   printf("Thread1 reads the value as %d\n",x);
   x++;  //thread1 increments its value
   printf("Local updation by Thread1: %d\n",x);
   sleep(1); //thread1 is preempted by thread 2
   shared=x; //thread one updates the value of shared variable
       printf("Value of shared variable updated by Thread1 is:
%d\n",shared);
   sem_post(&s);
}
void *fun2()
{
   int y;
```

```
    sem_wait(&s);
    y=shared;//thread2 reads value of shared variable
    printf("Thread2 reads the value as %d\n",y);
    y--;  //thread2 increments its value
    printf("Local updation by Thread2: %d\n",y);
    sleep(1); //thread2 is preempted by thread 1
    shared=y; //thread2 updates the value of shared variable
        printf("Value of shared variable updated by Thread2 is:
%d\n",shared);
    sem_post(&s);
}
```

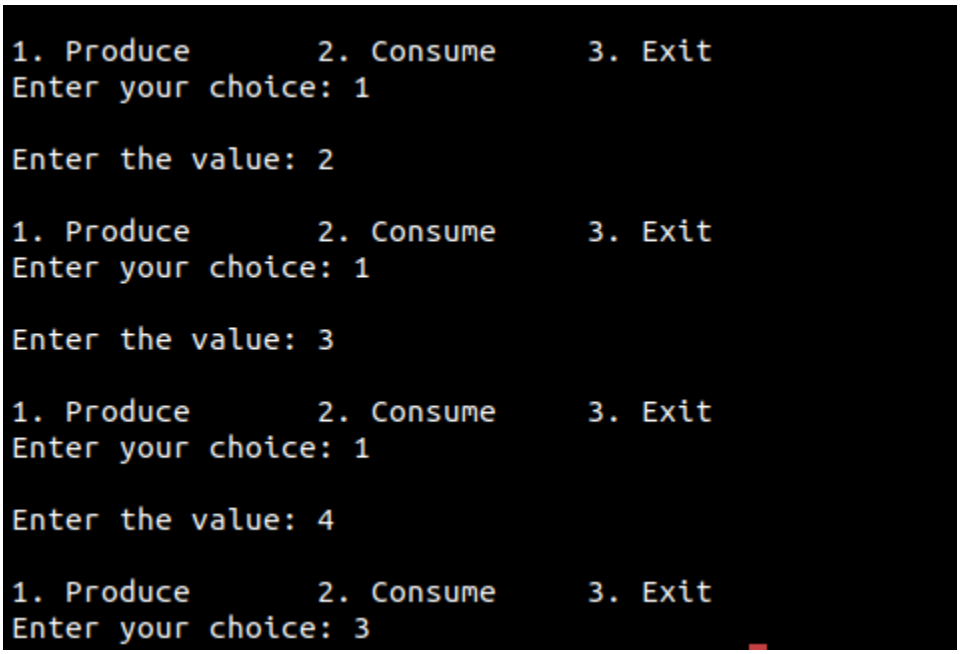
Output:

```
Thread1 reads the value as 1
Local updation by Thread1: 2
Value of shared variable updated by Thread1 is: 2
Thread2 reads the value as 2
Local updation by Thread2: 1
Value of shared variable updated by Thread2 is: 1
Final value of shared is 1
```

3. **Producer Consumer Code without Sleep() and wakeup().**

```
#include<stdio.h>
void main()
{
int buffer[10], bufsize, in, out, produce, consume, choice=0;
in = 0;
out = 0;
bufsize = 10;
while(choice !=3)
{
printf("\n1. Produce \t 2. 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: ");
scanf("%d", &produce);
buffer[in] = produce;
in = (in+1)%bufsize;
}
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;
}
} }
```

Output:


1. Produce 2. Consume 3. Exit
Enter your choice: 1
Enter the value: 2
1. Produce 2. Consume 3. Exit
Enter your choice: 1
Enter the value: 3
1. Produce 2. Consume 3. Exit
Enter your choice: 1
Enter the value: 4
1. Produce 2. Consume 3. Exit
Enter your choice: 3

4. **PC Code with Semaphores and Mutex:**

```
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <stdio.h>
#define MaxItems 5
#define BufferSize 5
sem_t empty;
sem_t full;
int in = 0;
int out = 0;
int buffer[BufferSize];
pthread_mutex_t mutex;


void *producer(void *pno)
{
int item;
for(int i = 0; i < MaxItems; i++)
{
item = rand();
sem_wait(&empty);
pthread_mutex_lock(&mutex);
buffer[in] = item;
printf("Producer %d: Insert Item %d at %d\n", *((int*)pno),buffer[in],in);
in = (in+1)%BufferSize;
pthread_mutex_unlock(&mutex);
sem_post(&full);
}
}

void *consumer(void *cno)
{
for(int i = 0; i < MaxItems; i++)
{
sem_wait(&full);
pthread_mutex_lock(&mutex);
int item = buffer[out];
```

```
printf("Consumer %d: Remove Item %d from %d\n",*((int *)cno),item, out);
out = (out+1)%BufferSize;
pthread_mutex_unlock(&mutex);
sem_post(&empty);
}
}


int main()
{
pthread_t pro[5],con[5];
pthread_mutex_init(&mutex, NULL);
sem_init(&empty,0,BufferSize);
sem_init(&full,0,0);
int a[5] = {1,2,3,4,5};
for(int i = 0; i < 5; i++)
{
pthread_create(&pro[i], NULL, (void *)producer, (void *)&a[i]);
}


for(int i = 0; i < 5; i++)
{
pthread_create(&con[i], NULL, (void *)consumer, (void *)&a[i]);
}
for(int i = 0; i < 5; i++)
{
pthread_join(pro[i], NULL);
}
for(int i = 0; i < 5; i++)
{
pthread_join(con[i], NULL);
}

pthread_mutex_destroy(&mutex);
sem_destroy(&empty);
sem_destroy(&full);
return 0;
```

Output:

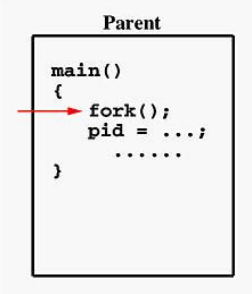
```
Producer 1: Insert Item 1804289383 at 0
Producer 1: Insert Item 1957747793 at 1
Producer 3: Insert Item 1681692777 at 2
Producer 4: Insert Item 1714636915 at 3
Consumer 2: Remove Item 1804289383 from 0
Consumer 2: Remove Item 1957747793 from 1
Consumer 2: Remove Item 1681692777 from 2
Producer 2: Insert Item 846930886 at 4
Producer 3: Insert Item 1649760492 at 0
Consumer 1: Remove Item 1714636915 from 3
Consumer 4: Remove Item 846930886 from 4
Producer 1: Insert Item 719885386 at 1
Producer 1: Insert Item 1350490027 at 2
Consumer 2: Remove Item 1649760492 from 0
Consumer 1: Remove Item 719885386 from 1
Producer 3: Insert Item 1025202362 at 3
Producer 5: Insert Item 424238335 at 4
Producer 4: Insert Item 596516649 at 0
Consumer 3: Remove Item 1350490027 from 2
Producer 2: Insert Item 1189641421 at 1
Consumer 5: Remove Item 1025202362 from 3
Consumer 2: Remove Item 424238335 from 4
Consumer 1: Remove Item 596516649 from 0
Producer 1: Insert Item 783368690 at 2
Consumer 4: Remove Item 1189641421 from 1
Producer 3: Insert Item 1102520059 at 3
Producer 5: Insert Item 2044897763 at 4
Producer 4: Insert Item 1967513926 at 0
Consumer 3: Remove Item 783368690 from 2
Producer 2: Insert Item 1365180540 at 1
Consumer 5: Remove Item 1102520059 from 3
Consumer 1: Remove Item 2044897763 from 4
Consumer 4: Remove Item 1967513926 from 0
Producer 3: Insert Item 1540383426 at 2
Consumer 3: Remove Item 1365180540 from 1
Producer 5: Insert Item 304089172 at 3
Producer 5: Insert Item 521595368 at 4
Producer 2: Insert Item 35005211 at 0
Consumer 5: Remove Item 1540383426 from 2
Producer 4: Insert Item 1303455736 at 1
Producer 5: Insert Item 294702567 at 2
Consumer 1: Remove Item 304089172 from 3
Consumer 4: Remove Item 521595368 from 4
Producer 2: Insert Item 1726956429 at 3
Consumer 3: Remove Item 35005211 from 0
Producer 4: Insert Item 336465782 at 4
Consumer 3: Remove Item 1303455736 from 1
Consumer 4: Remove Item 294702567 from 2
Consumer 5: Remove Item 1726956429 from 3
Consumer 5: Remove Item 336465782 from 4
```

Lab Experiment No. 6: Implementation of System Calls related to Process Creation

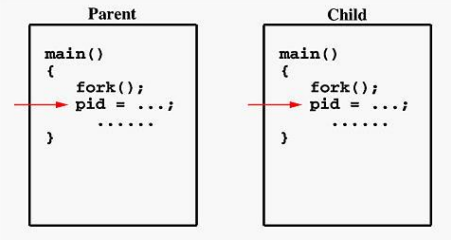
Aim: To get the idea of how processes are created using fork() system call.

fork() System Call:

- Fork system call is used for creating a new process, which is called child process, which runs concurrently with the process that makes the fork() call (parent process).
- After a new child process is 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 which use in the parent process.
- It takes no parameters and returns an integer value. Below are different values returned by fork().
    - Negative Value: 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 process ID of newly created child process
- Suppose the given program executes up to the point of the call to fork() (marked in red color):



- If the call to fork() is executed successfully, it will
    - make two identical copies of address spaces, one for the parent and the other for the child.
    - Both processes will start their execution at the next statement following the fork() call. In this case, both processes will start their execution at the assignment statement as shown below:



- Both processes start their execution right after the system call fork().
- Since both processes have identical but separate address spaces, those variables initialized before the fork() call have the same values in both address spaces.
- Since every process has its own address space, any modifications will be independent of the others.
- In other words, if the parent changes the value of its variable, the modification will only affect the variable in the parent process's address space.
- Other address spaces created by fork() calls will not be affected even though they have identical variable names.

Code-I: For tracing the execution of Parent and Child processes

Code:

```
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
int main(){
pid_t pid = fork();
if(pid == 0){
   printf("fork executed successfully: This is Child process with PID: %d
\n", getpid());
   printf("This is Child process. My Parent PID: %d \n",  getppid());

}
else if(pid>0){
wait(NULL);
   printf("I am Parent process with PID : %d \n",  getpid());
}
return 0;
}
```

Output:

```
fork executed successfully: This is Child process with PID: 39707
This is Child process. My Parent PID: 39706
I am Parent process with PID : 39706
```

Code-II: For tracing the execution of Parent and Child processes

Code:

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
int main(){
printf("Parent PPID %d\n",getppid());
printf("I am Parent with PID : %d \n", (int) getppid());
printf("Parent going to create a child process...\n");
pid_t pid = fork();
if(pid == 0){
   printf("fork executed successfully: Child PID: %d \n", (int) getpid());
   printf("My Parent with PID : %d \n", (int) getppid());
}
else{
wait(NULL);
printf("I am the parent, my PID %d \n",getpid());
}
return 0;
}
```

Output:

```
Parent PPID 39376
I am Parent with PID : 39376
Parent going to create a child process...
fork executed successfully: Child PID: 39935
My Parent with PID : 39934
I am the parent, my PID 39934
```

Code-III: For replacing the code image of Parent and Child processes

Code:

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[]){
pid_t pid = fork();
if(pid==0){
printf("Child PID: %d \n", (int) getpid());
execlp(argv[1],"arguments",NULL);
}
else if (pid>0){
wait(NULL);
//printf("Parent resumes from there");
printf("Parent PID: %d \n", (int) getpid());
execl(argv[2],"some arguments",NULL);
}
}
```

Output:

```
Child PID: 40100
Parent PID: 40099
```

Lab Experiment No. 7: Implementation of Deadlock Avoidance Algorithm

Aim: To get the idea of how processes' demand-based analysis can avoid deadlock.

Deadlock Avoidance:

- Deadlock may happen.
- So, better to check for deadlock existence in future by determining whether the system is in safe state or in unsafe state.
    - **Safe State**: If the available resources can be allocated to the requesting process
    - **Unsafe State**: If the available resources cannot be allocated to requesting process
- Deadlock avoidance take care of safe and unsafe state because unsafe state will result in deadlock.
- At an instance of time, processes from A to E has resources allocated and required as follows.

Process	R0	R1	R2	R3
P0	3	0	1	1
P1	0	1	0	0
P2	1	1	1	0
P3	1	1	0	1
P4	0	0	0	0

Resources Assigned

Process	R0	R1	R2	R3
P0	4	1	1	1
P1	0	2	1	2
P2	4	2	1	0
P3	1	1	1	1
P4	2	1	1	0

Maximum Requirement

- Total Available= (1,0,2,0)
- B' request = (0,0,1,0)
- Will the system be in safe state after catering B' request

Code:

```
#include<stdio.h>
struct file
{
int all[10];
int max[10];
int need[10];
int flag;
};
```

```
int main()
{
struct file f[10];
int fl;
int i, j, k, p, b, n, r, g, cnt=0, id, newr;
int avail[10],seq[10];
printf("Enter number of processes -- ");
scanf("%d",&n);
printf("Enter number of resources -- ");
scanf("%d",&r);

for(i=0;i<n;i++)
{
printf("Enter details for P%d",i);
printf("\nEnter allocation\t -- \t");

for(j=0;j<r;j++)
scanf("%d",&f[i].all[j]);

printf("Enter Max\t\t -- \t");

for(j=0;j<r;j++)
scanf("%d",&f[i].max[j]);

f[i].flag=0;
}

printf("\nEnter Available Resources\t -- \t");

for(i=0;i<r;i++)
scanf("%d",&avail[i]);

printf("\nEnter New Request Details -- ");
printf("\nEnter pid \t -- \t");
scanf("%d",&id);
printf("Enter Request for Resources \t -- \t");

for(i=0;i<r;i++)
```

```
{
scanf("%d",&newr);
f[id].all[i] += newr;
avail[i]=avail[i] - newr;
}

for(i=0;i<n;i++)
{
for(j=0;j<r;j++)
{
f[i].need[j]=f[i].max[j]-f[i].all[j];
if(f[i].need[j]<0)
f[i].need[j]=0;
}
}

cnt=0;
fl=0;

while(cnt!=n)
{
g=0;
for(j=0;j<n;j++)
{
if(f[j].flag==0)
{
b=0;
for(p=0;p<r;p++)
{
if(avail[p]>=f[j].need[p])
b=b+1;
else
b=b-1;
}

if(b==r)
{
printf("\nP%d is visited",j);
seq[fl++]=j;
f[j].flag=1;
```

```
for(k=0;k<r;k++)
avail[k]=avail[k]+f[j].all[k];

cnt=cnt+1;

printf("(");

for(k=0;k<r;k++)
printf("%3d",avail[k]);
printf(")"); g=1;
}
}
}
if(g==0)
{
printf("\n REQUEST NOT GRANTED -- DEADLOCK OCCURRED");
printf("\n SYSTEM IS IN UNSAFE STATE");
goto y;
}
}

printf("\nSYSTEM IS IN SAFE STATE");
printf("\nThe Safe Sequence is -- (");

for(i=0;i<fl;i++)
printf("P%d ",seq[i]);

printf(")");
y: printf("\nProcess\t\tAllocation\t\tMax\t\t\tNeed\n");

for(i=0;i<n;i++)
{
printf("P%d\t",i);

for(j=0;j<r;j++)
printf("%6d",f[i].all[j]);

for(j=0;j<r;j++)
```

```
printf("%6d",f[i].max[j]);

for(j=0;j<r;j++)
printf("%6d",f[i].need[j]);

printf("\n");
}
return 0;
}
```

Output:

```
Enter number of processes -- 5
Enter number of resources -- 3
Enter details for P0
Enter allocation        --      0 1 0
Enter Max               --      7 5 3
Enter details for P1
Enter allocation        --      2 0 0
Enter Max               --      3 2 2
Enter details for P2
Enter allocation        --      3 0 2
Enter Max               --      9 0 2
Enter details for P3
Enter allocation        --      2 1 1
Enter Max               --      2 2 2
Enter details for P4
Enter allocation        --      0 0 2
Enter Max               --      4 3 3

Enter Available Resources       --      3 3 2

Enter New Request Details --
Enter pid       --      0
Enter Request for Resources     --      1
3 2

 REQUEST NOT GRANTED -- DEADLOCK OCCURRED
 SYSTEM IS IN UNSAFE STATE
Process         Allocation              Max                     Need
P0          1     4     2     7     5     3     6     1     1
P1          2     0     0     3     2     2     1     2     2
P2          3     0     2     9     0     2     6     0     0
P3          2     1     1     2     2     2     0     1     1
P4          0     0     2     4     3     3     4     3     1
```

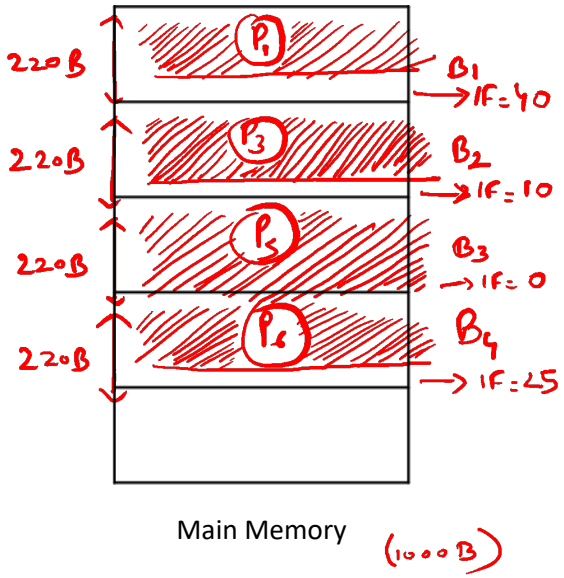
Lab Experiment No. 8: Simulation of MFT and MVT Memory Management Techniques

Aim: To get the idea of how contiguous allocation of processes get affected due to the fixed and dynamic memory partitioning schemes.

Background:

1. **MFT or fixed partitioning Scheme**

- Full form of MFT is Multiprogramming with Fixed number of Tasks
- In fixed scheme, the OS will be divided into fixed sized blocks. It takes place at the time of installation.
- Degree of multiprogramming is not flexible. This is because the number of blocks is fixed resulting in memory wastage due to fragmentation.
- Example:
    - Suppose, there are total of 1000 B of space in Main Memory.
    - OS has four fixed blocks (partitions) of size 220 B each.
    - There are total of six processes with following memory requirements:
        - P1=180 B
        - P2=250 B
        - P3=210 B
        - P4=310 B
        - P5=220 B
        - P6=195 B
    - The memory has the allocation of processes and internal fragmentations as shown in the figure, with external fragmentation (EF)=195 B.



Code:

```
#include<stdio.h>

int main()
{
int ms, bs, nob, ef, n, mp[10], tif=0, oop;
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;
oop=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\t ALLOCATED\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 = tif + bs-mp[i];
p++;
}
```

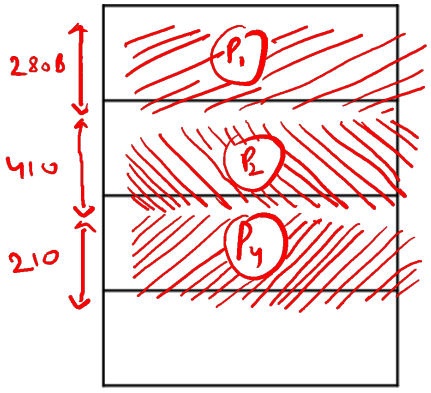
```
}

ef=oop+tif;

if(i<n)
printf("\nMemory is Full, Remaining Processes cannot be accomodated");

printf("\n\nTotal Internal Fragmentation is %d",tif);
printf("\nTotal External Fragmentation is %d",ef);

return 0;
}
```

Output:

```
Enter the total memory available (in Bytes) -- 256
Enter the block size (in Bytes) -- 8

Enter the number of processes -- 10
Enter memory required for process 1 (in Bytes)-- 7
Enter memory required for process 2 (in Bytes)-- 8
Enter memory required for process 3 (in Bytes)-- 4
Enter memory required for process 4 (in Bytes)-- 6
Enter memory required for process 5 (in Bytes)-- 8
Enter memory required for process 6 (in Bytes)-- 2
Enter memory required for process 7 (in Bytes)-- 1
Enter memory required for process 8 (in Bytes)-- 3
Enter memory required for process 9 (in Bytes)-- 3
Enter memory required for process 10 (in Bytes)-- 4

No. of Blocks available in memory -- 32

PROCESS MEMORY REQUIRED  ALLOCATED      INTERNAL FRAGMENTATION
 1              7               YES     1
 2              8               YES     0
 3              4               YES     4
 4              6               YES     2
 5              8               YES     0
 6              2               YES     6
 7              1               YES     7
 8              3               YES     5
 9              3               YES     5
 10             4               YES     4

Total Internal Fragmentation is 34
Total External Fragmentation is 34
```

2. MVT OR variable partitioning Scheme

- Full form of MVT is Multiprogramming with Variable number of Tasks
- In variable partitioning scheme there are no partitions at the beginning.
- There is only the OS area and the rest of the available RAM.
- The memory is allocated to the processes as they enter.
- This method is more flexible as there is no internal fragmentation and there is no size limitation.
- Example
    - Suppose, there are total of 1000 B of space in Main Memory.
    - There are total of four processes with following memory requirements:
        - P1=280 B
        - P2=410 B
        - P3=500 B
        - P4=210 B
    - The memory has the allocation of processes as shown in the figure, with external fragmentation (EF) =100 B.



(1000B)

```
#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 = temp - mp[i];
}
else printf("\nMemory is not available for the current request");
printf("\nDo you want to continue(y/n) -- ");
scanf(" %c", &ch);
if(ch=='n')
break;
}
printf("\n\nTotal Memory Available -- %d", ms);
printf("\n\n\tPROCESS\t\t MEMORY 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",temp);
return 0;
}
```

Output:

```
Enter the total memory available (in Bytes)-- 256

Enter memory required for process 1 (in Bytes) --
8

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

Enter memory required for process 2 (in Bytes) -- 7

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

Enter memory required for process 3 (in Bytes) -- 4

Memory is allocated for Process 3
Do you want to continue(y/n) -- y

Enter memory required for process 4 (in Bytes) -- 6

Memory is allocated for Process 4
Do you want to continue(y/n) -- y
```

```
Enter memory required for process 5 (in Bytes) -- 8

Memory is allocated for Process 5
Do you want to continue(y/n) -- y

Enter memory required for process 6 (in Bytes) -- 6

Memory is allocated for Process 6
Do you want to continue(y/n) -- y

Enter memory required for process 7 (in Bytes) -- 1

Memory is allocated for Process 7
Do you want to continue(y/n) -- y

Enter memory required for process 8 (in Bytes) -- 3

Memory is allocated for Process 8
Do you want to continue(y/n) -- y

Enter memory required for process 9 (in Bytes) -- 2

Memory is allocated for Process 9
Do you want to continue(y/n) -- n

Total Memory Available -- 256

        PROCESS          MEMORY ALLOCATED
        1               8
        2               7
        3               4
        4               6
        5               8
        6               6
        7               1
        8               3

Total Memory Allocated is 45
Total External Fragmentation is 211
```