Operating system & Computer Networks Laboratory

Program	Master of Computer Applications
Semester	1
Course Title	Operating system & Computer Networks Laboratory
Course Code	23OMC109
Course Credits	2
Course Type	Core Course

1. Course Summary

The Operating System and Computer Network Lab course offers practical experience in computer networks and operating systems. Students learn network commands, configure network devices, implement point-to-point networks, and analyze network performance. They also simulate CPU scheduling algorithms, memory management techniques, disk scheduling algorithms. This hands-on approach enhances their understanding of network protocols, resource allocation, and system performance. By completing the term works, students gain valuable skills in network configuration, command-line tools, and practical implementation of operating system algorithms..

2. Course Outcomes (COs)

After the successful completion of this course, the student will be able to:

CO-1. Explain the use of network commands in command prompt to diagnose and troubleshoot network connectivity issues. L2)

CO-2. Configure and demonstrate the use of hub, switch, and router in a simple network topology.(L3)

CO-3. Analyze the impact of queue size and bandwidth on network performance by implementing a point-to-point network with varying parameters and observing packet drops.(L4)

CO-4. Simulate CPU scheduling algorithms such as FCFS and SJF and analyze their performance using different process arrival times and burst times .(L4)

CO-5. Use the semaphore mechanism to control access to the shared buffer, ensuring proper synchronization between producer and consumer threads. (L4)

Operating systems

1.	Simulate the following CPU scheduling algorithms-FCFS, SJF
2.	Simulate the following Memory management algorithm-First fit, Best fit
3.	Implement the optimal page replacement algorithm
4.	Implement the FCFS Disk scheduling and SSTF Disk scheduling
5.	Implement the producer consumer problem with solution using semaphore
6.	Implement a program in C to extract process ID (PID) and parent process ID (PPID).

Experiment 1: CPU Scheduling Algorithms - FCFS and SJF

Objective: To simulate the First-Come-First-Serve (FCFS) and Shortest Job First (SJF) CPU scheduling algorithms.

Procedure:

FCFS Scheduling:

Write a C program that simulates the FCFS scheduling algorithm.

Input a list of processes with their burst times.

Calculate the waiting time and turnaround time for each process.

Display the scheduling order, waiting time, and turnaround time for each process.

SJF Scheduling:

Write a C program that simulates the SJF scheduling algorithm.

Input a list of processes with their burst times.

Sort the processes based on burst time (shortest job first).

Calculate the waiting time and turnaround time for each process.

Display the scheduling order, waiting time, and turnaround time for each process.

```
/* FCFS CPU Scheduling */
#include <stdio.h>
#include <stdlib.h>
struct Process {
   int pid;    // process ID
   int bt;     // burst time
   int at;     // arrival time
};
void fcfs_scheduling(struct Process *proc, int n) {
   int i, wt[n], tat[n], total_wt = 0, total_tat = 0;
   // calculate waiting time for each process
   wt[0] = 0;
   for (i = 1; i < n; i++) {
        wt[i] = wt[i-1] + proc[i-1].bt;
    }
    // calculate turnaround time for each process
    for (i = 0; i < n; i++) {
        tat[i] = wt[i] + proc[i].bt;
    }
    // calculate total waiting and turnaround time
    for (i = 0; i < n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
    }
    // print result
    printf("\nPID\tBT\tAT\tWT\tTAT\n");
    for (i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t%d\n", proc[i].pid, proc[i].bt, proc[i].at, wt[i], tat[i]);
    }
    printf("\nAverage waiting time: %.2f\n", (float)total_wt / n);
    printf("Average turnaround time: %.2f\n", (float)total_tat / n);
}
int main() {
    int n, i;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    struct Process proc[n];
    for (i = 0; i < n; i++) {
        printf("Enter the burst time and arrival time for process %d: ", i+1);
        scanf("%d%d", &proc[i].bt, &proc[i].at);
        proc[i].pid = i+1;
    }
```

```
   fcfs_scheduling(proc, n);
   return 0;
}
```

SAMPLE OUTPUT

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

PID   BT    AT    WT    TAT
1     3     0     0     3
2     3     0     3     6
3     2     0     6     8
4     5     0     8     13
Average waiting time: 4.25
Average turnaround time: 7.50
```

```
/* SJF Final */
#include <stdio.h>
#include <stdlib.h>
struct Process {
   int pid;
   int burst_time;
   int waiting_time;
   int turnaround_time;
};
void main()
```

```
{
    int n, i, j;
    struct Process temp;
    float avg_waiting_time = 0, avg_turnaround_time = 0;
    struct Process *processes;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    // Allocate memory for the array of processes
    processes = (struct Process* ) malloc (n *  sizeof (struct Process) );

    // Reading burst time of each process
    for(i=0; i<n; i++) {
        printf("Enter the burst time for process %d: ", i+1);
        scanf("%d", &processes[i].burst_time);
        processes[i].pid = i+1;
    }
    // Sort the processes based on their burst time in ascending order
    for(i=0; i<n-1; i++) {
        for(j=i+1; j<n; j++) {
            if(processes[i].burst_time > processes[j].burst_time) {
                // Swap the processes
                temp = processes[i];
                processes[i] = processes[j];
                processes[j] = temp;
            }
        }
    }
    // Calculate waiting time and turnaround time for each process
    processes[0].waiting_time = 0;
    processes[0].turnaround_time = processes[0].burst_time;
    for(i=1; i<n; i++) {
```

```
    processes[i].waiting_time = processes[i-1].waiting_time + processes[i-1].burst_time;
    processes[i].turnaround_time = processes[i].waiting_time + processes[i].burst_time;
  }
  // Calculate average waiting time and turnaround time
  for(i=0; i<n; i++) {
    avg_waiting_time += processes[i].waiting_time;
    avg_turnaround_time += processes[i].turnaround_time;
  }
  avg_waiting_time /= n;
  avg_turnaround_time /= n;
  // Print the results
  printf("Process ID\tBurst Time\tWaiting Time\tTurnaround Time\n");
    // Sort the processes based on their ID in ascending order to print in proper order
  for(i=0; i<n-1; i++) {
    for(j=i+1; j<n; j++) {
      if(processes[i].pid > processes[j].pid) {
        // Swap the processes
        temp = processes[i];
        processes[i] = processes[j];
        processes[j] = temp;


      }
    }
  }


  for(i=0; i<n; i++) {
    printf("%d\t\t%d\t\t%d\t\t%d\n", processes[i].pid, processes[i].burst_time, processes[i].waiting_time,
processes[i].turnaround_time);
  }
  printf("\nAverage Waiting Time = %0.2f ms", avg_waiting_time);
  printf("\nAverage Turnaround Time = %0.2f ms\n", avg_turnaround_time);
  // Free the allocated memory
```

```
    free(processes);
}
```

In this program, we define a structure Process to store information about each process, including its process ID, burst time, waiting time, and turnaround time. The program first reads the burst time of each process and then sorts them in ascending order based on their burst time. It then calculates the waiting time and turnaround time for each process and finally prints the results, including the average waiting time and turnaround time.

SAMPLE OUTPUT 1:

```
Enter the number of processes: 3
Enter the burst time for process 1: 5
Enter the burst time for process 2: 2
Enter the burst time for process 3: 8
Process ID    Burst Time    Waiting Time    Turnaround Time
1             5             2               7
2             2             0               2
3             8             7               15

Average Waiting Time = 3.000000
Average Turnaround Time = 8.000000
```

SAMPLE OUTPUT 2:

```
Enter the number of processes: 4
Enter the burst time for process 1: 6
Enter the burst time for process 2: 8
Enter the burst time for process 3: 5
Enter the burst time for process 4: 2
Process ID    Burst Time    Waiting Time    Turnaround Time
1             6             7               13
2             8             13              21
3             5             2               7
4             2             0               2
```

```
Average Waiting Time = 5.500000
Average Turnaround Time = 10.750000
```

Experiment 2: Memory Management Algorithms - First Fit and Best Fit

Objective: To simulate the First Fit and Best Fit memory allocation algorithms.

Procedure:

First Fit Allocation:

Write a C program that simulates the First Fit memory allocation algorithm.

Maintain a list of available memory blocks and their sizes.

Input a list of processes with their memory requirements.

Allocate memory to processes based on the First Fit algorithm.

Display the memory allocation details.

Best Fit Allocation:

Write a C program that simulates the Best Fit memory allocation algorithm.

Maintain a list of available memory blocks and their sizes.

Input a list of processes with their memory requirements.

Allocate memory to processes based on the Best Fit algorithm.

Display the memory allocation details.

// C program to implement First Fit memory allocation algorithm

```
#include<stdio.h>

// Function to allocate memory

void firstFit(int m, int n, int Blocks[], int Process[])

{

        int i, j;

        // Stores block id of the

        int allocation[n]; // To srote the allocated block number

        // Initialize to -1 as no block is assigned to processes

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

        {

                allocation[i] = -1;
```

```
}
// Look for availble block to the process and allocate if fits

for (i = 0; i < n; i++)     //n number of processes
{
        for (j = 0; j < m; j++)     //m number of blocks
        {
                if (Blocks[j] >= Process[i])
                {
                        // allocating block j to the ith process
                        allocation[i] = j;

                        // Reduce available memory in this block.
                        Blocks[j] = Blocks[j] - Process[i];

                        break; //go to the next process in the queue
                }
        }
}

printf("\nProcess No.\tProcess Size\t\tBlock no.\n");
for (int i = 0; i < n; i++)
{
        printf(" %d\t\t\t", i+1);
        printf("%d\t\t\t", Process[i]);
        if (allocation[i] != -1)
                printf("%i", allocation[i] + 1);
        else
                printf("Not Allocated");
```

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

int main()
{
          int m; //number of blocks in the memory
          int n; //number of processes in the input queue
          int i;
          int Blocks[10];
          int Process[10];
          printf("Enter the number of blocks\n");
          scanf("%d", &m);
          printf("Enter the number of processes\n");
          scanf("%d", &n);
          printf("Enter the block sizes\n");
          for(i=0; i<m; i++)
             scanf("%d", &Blocks[i]);
          printf("Enter the process sizes\n");
          for(i=0; i<n; i++)
             scanf("%d", &Process[i]);

          firstFit(m, n, Blocks, Process);
          return 0 ;
}
```

SAMPLE OUTPUT 1

```
Enter the number of blocks

4

Enter the number of processes

3

Enter the block sizes

220

300

200

300

Enter the process sizes

250

100

600


Process No.   Process Size      Block no.

 1             250            2

 2             100            1

 3             600            Not Allocated
```

SAMPLE OUTPUT 2

```
Enter the number of blocks

5

Enter the number of processes

4

Enter the block sizes

100

500

200

300
```

```
600

Enter the process sizes

225

450

120

350


Process No.   Process Size      Block no.

 1             225             2

 2             450             5

 3             120             2

 4             350             Not Allocated
```

SAMPLE OUTPUT 3

```
Enter the number of blocks

5

Enter the number of processes

4

Enter the block sizes

100

500

200

300

600

Enter the process sizes

220

430

110

425
```

Process No.	Process Size	Block no.
1	220	2
2	430	5
3	110	2
4	425	Not Allocated

```
// Implementation of Best Fit memory allocation algorithm in C
#include<stdio.h>
// Function to allocate memory
void BestFit(int m, int n, int partitions[], int process[])
{
        int i, j;
        int allocation[n];  // To store the partition allocated to process
        // Initialize to -1 as no block is assigned to processes
        for(i = 0; i < n; i++)
        {
                allocation[i] = -1;
        }
        // For every process select the best fit partition
        for (int i = 0; i < n; i++)     // Select process
        {
                int bestPart  = -1;
                for (int j = 0; j < m; j++)     //Search the partition
                {
                        if (partitions[j] >= process[i])
                        {
                                if (bestPart == -1)
                                        bestPart = j;
                                else if (partitions[bestPart] > partitions[j])
                                        bestPart = j;
                        }
                }
        // If bestPart is found
                if (bestPart != -1)
                {
                        // allocate Partition j to p[i] process
                        allocation[i] = bestPart;
                        // Reduce the partition size of this block
                        partitions[bestPart] = partitions[bestPart]  - process[i];
                }
        }
        printf("\nProcess No.\tProcess Size\t\tBlock no.\n");
        for (int i = 0; i < n; i++)
        {
                printf(" %d\t\t\t", i+1);
                printf("%d\t\t\t", process[i]);
                if (allocation[i] != -1)
                        printf("%d", allocation[i] + 1);
                else
                        printf("Not Allocated");
                printf("\n");
        }
}


int main()
{
        int m; //number of partitions in the memory
        int n; //number of processes in the input queue
        int i;
        int partitions[10];
        int process[10];
        printf("Enter the number of partitions\n");
        scanf("%d", &m);
        printf("Enter the number of processes\n");
        scanf("%d", &n);
        printf("Enter the partition sizes\n");
        for(i=0; i<m; i++)
           scanf("%d", &partitions[i]);
        printf("Enter the process sizes\n");
        for(i=0; i<n; i++)
           scanf("%d", &process[i]);
        BestFit(m, n, partitions, process);
        return 0 ;
}
```

SAMPLE OUTPUT 1

```
Enter the number of partitions
4
Enter the number of processes
3
Enter the partition sizes
220
300
200
```

```
300

Enter the process sizes

250

100

600


Process No.    Process Size        Block no.

 1              250               2

 2              100               3

 3              600               Not Allocated


SAMPLE OUTPUT 2

Enter the number of partitions

5

Enter the number of processes

4

Enter the partition sizes

100

500

200

300

600

Enter the process sizes

225

650

120

350


Process No.    Process Size        Block no.
```

```
1           225         4
2           650         Not Allocated
3           120         3
4           350         2
```

SAMPLE OUTPUT 3

```
Enter the number of partitions
5
Enter the number of processes
4
Enter the partition sizes
100
500
200
300
600
Enter the process sizes
220
430
110
425

Process No.  Process Size    Block no.
1           220         4
2           430         2
3           110         3
4           425         5
```

SAMPLE OUTPUT 4

```
Enter the number of partitions

3

Enter the number of processes

2

Enter the partition sizes

100

600

300

Enter the process sizes

350

200


Process No.   Process Size        Block no.

 1             350          2

 2             200          2
```

Experiment 3: Optimal Page Replacement Algorithm

Objective: To implement and evaluate the Optimal Page Replacement algorithm for virtual memory management.

Procedure:

Write a C program to simulate the Optimal Page Replacement algorithm.

Input the reference string and the number of frames in memory.

Implement the Optimal algorithm to replace pages when a page fault occurs.

Calculate and display the page fault rate and the pages replaced.

```
/* Program to implement Optimal page replacement algorithm */
#include<stdio.h>
int main()
{
  int num_frames, num_pages, frames[10], pages[30], temp[10];
  int flag1, flag2, flag3, i, j, k, pos, max, faults = 0, hit;
  printf("Enter number of frames: ");
  scanf("%d", &num_frames);

  printf("Enter number of pages: ");
  scanf("%d", &num_pages);

  printf("Enter page reference string: ");
  for(i = 0; i < num_pages; i++){
    scanf("%d", &pages[i]);
  }

  /* assigning initial value */
  for(i = 0; i < num_frames; i++){
    frames[i] = -1;
  }
```

```
for(i = 0; i < num_pages; i++) {  // Trying to pagein pages[i]
  flag1 = flag2 = 0;
  hit = 0;
  for(j = 0; j < num_frames; j++) {
    if(frames[j] == pages[i]){   // Page hit
        flag1 = flag2 = 1;
        hit = 1;
        break;
      }
  }

  if(flag1 == 0){           // Page fault
    for(j = 0; j < num_frames; j++){  // Trying to find free frame
      if(frames[j] == -1){
        faults++;
        frames[j] = pages[i];   // pagein pages[i]
        flag2 = 1;
        break;
      }
    }
  }

  if(flag2 == 0){   // No free frame
   flag3 =0;
    for(j = 0; j < num_frames; j++){    // Trying to find victim page
     temp[j] = -1;
     for(k = i + 1; k < num_pages; k++){   // Consider future reference
     if(frames[j] == pages[k]){          // pages[k] will be referred
```

```
        temp[j] = k;
        break;
        }
        }
       }

       for(j = 0; j < num_frames; j++){
        if(temp[j] == -1){    // if temp[j] == -1 then that page will not be referred
        pos = j;             // use this pos to pagein
        flag3 = 1;
        break;
        }
       }
       if(flag3 ==0){      // All the pages in the frame will be referred
        max = temp[0];
        pos = 0;
        for(j = 1; j < num_frames; j++){  // Find the page that will not be referred for long time
        if(temp[j] > max){
        max = temp[j];
        pos = j;              // use this pos to pagein
        }
        }
       }
frames[pos] = pages[i];   // Page in
faults++;
   }

   if (hit == 0)
   {
        printf("\n");


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

SAMPLE OUTPUT

```
Enter number of frames: 3
Enter number of pages: 10
Enter page reference string: 2
3
4
2
1
3
7
5
4
3


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

7	3	4
5	3	4

Total Page Faults = 6

Experiment 4: Disk Scheduling - FCFS and SSTF

Objective: To implement and compare the First-Come-First-Serve (FCFS) and Shortest Seek Time First (SSTF) disk scheduling algorithms.

Procedure:

FCFS Disk Scheduling:

Write a C program to simulate the FCFS disk scheduling algorithm.

Input the sequence of disk track requests.

Calculate and display the total head movement.

SSTF Disk Scheduling:

Write a C program to simulate the SSTF disk scheduling algorithm.

Input the sequence of disk track requests.

Implement the SSTF algorithm to find the next track to move to.

Calculate and display the total head movement.

```
// C program to implement FCFS Disk Scheduling algorithm
#include <stdio.h>
void FCFS(int cylinder[],int num_cyli, int head_pos)
{
        int movement = 0;
        int cur_cyli, distance;
        printf("\nHead Movements \n");
        printf("\nFrom\tTo\tMovement \n");

        for( int i = 0 ; i < num_cyli ; i++ )
        {
                cur_cyli = cylinder[i];

                // Find absolute distance
                distance = fabs(head_pos - cur_cyli);
                //Increment the total count
                movement = movement + distance;

                printf("%d\t%d\t%d \n" , head_pos, cur_cyli, distance);

                //Assign new head position
                 head_pos = cur_cyli;

        }
        printf(" Total number of head movemnts = %d ", movement);
}
int main()
{
        int cylinder[20];  // request array
        int num_cyli, i, head_pos ;

        printf("Enter the number of requests \n");
        scanf("%d", & num_cyli);

        printf("Enter the disk queue of cylinders \n");
        for(i = 0 ; i < num_cyli ; i++)
           scanf("%d", & cylinder[i]);

        printf("Enter the initial head position \n");
           scanf("%d", & head_pos);
        FCFS(cylinder, num_cyli, head_pos);

        return 0;
}
```

```
SAMPLE OUTPUT

Enter the number of requests

8

Enter the disk queue of cylinders

98

183

37

122

14

124

65

67

Enter the initial head position

53


Head Movements


From   To     Movement

53     98     45

98     183    85

183    37     146

37     122    85

122    14     108

14     124    110

124    65     59

65     67     2

 Total number of head movemnts = 640
```

```
//To implement the Shortest Seek Time First disk scheduling algorithm

#include<math.h>
```

```
#include<stdio.h>

void sjfs(int cylinder[], int num_cyli, int head_pos)
{
  int i,k,movement = 0,cur_pos,index[20],minimum,temp[20],j=0,min_index,distance;
    cur_pos = head_pos;
     for(k = 0 ; k < num_cyli ; k++)
  {
     for(i = 0 ; i < num_cyli ; i++)
     {
        // Calculate distance of each request from current position
        index[i]=fabs(cur_pos - cylinder[i]);
     }
     // Find the nearest request
     minimum = index[0];
     min_index = 0;
     for(i = 1 ; i < num_cyli ; i++)
     {
        if(minimum > index[i])
        {
           minimum = index[i];
           min_index = i;
        }
     }
     temp[j] = cylinder[min_index];
     j++;
     // change the current position value to next request
     cur_pos = cylinder[min_index];
     // Change the processed request so that it is not processed again
     cylinder[min_index] = 9999;
  }


  printf("\nHead Movements \n");
        printf("\nFrom\tTo\tMovement \n");


  movement = movement + fabs(head_pos - temp[0]);    // head movement
  printf("%d\t%d\t%d\n", head_pos, temp[0], movement);
  for(i = 1 ; i < num_cyli ; i++)
  {
     distance = fabs(temp[i]-temp[i-1]); ///head movement


     printf("%d\t%d\t%d\n", temp[i-1], temp[i], distance);
     movement = movement + distance;


  }
  printf("\n");
  printf("Total number of head movements = %d\n",movement);
}
int main()
{
        int cylinder[20];  // request array
        int num_cyli, i, head_pos ;
        printf("Enter the initial head position \n");
        scanf("%d", & head_pos);


        printf("Enter the number of requests \n");
        scanf("%d", & num_cyli);


        printf("Enter the disk queue of cylinders \n");
        for(i = 0 ; i < num_cyli ; i++)
        scanf("%d", &cylinder[i]);
        sjfs(cylinder, num_cyli, head_pos);
    return 0;
}
```

```
/* SAMPLE OUTPUT 1 */
Enter the initial head position
53
Enter the number of requests
8
Enter the disk queue of cylinders
98
183
37
122
14
124
65
67

Head Movements

From   To    Movement
53     65    12
65     67    2
67     37    30
37     14    23
14     98    84
98     122   24
122    124   2
```

```
124    183    59

Total number of head movements = 236
/* SAMPLE OUTPUT 2 */
Enter the initial head position
50
Enter the number of requests
2
Enter the disk queue of cylinders
60
10
Head Movements
From    To     Movement
50      60     10
60      10     50

Total number of head movements = 60
```

Experiment 5: Producer-Consumer Problem with Semaphores

Objective: To implement a solution to the Producer-Consumer problem using semaphores in C.

Procedure:

Write a C program that demonstrates the Producer-Consumer problem.

Implement the producer and consumer functions using semaphores for synchronization.

Ensure that producers and consumers work concurrently without issues.

```
// C program for the Producer-Consumer

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

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

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

// Number of empty slots as size
// of buffer
int empty = 10, x = 0;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

case 2:

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

```
                    }

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

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

SAMPLE OUTPUT:

```
1. Press 1 for Producer
2. Press 2 for Consumer
3. Press 3 for Exit
Enter your choice:1

Producer producesitem 1
Enter your choice:1
```

```
Producer producesitem 2
Enter your choice:1

Producer producesitem 3
Enter your choice:2

Consumer consumes item 3
Enter your choice:2

Consumer consumes item 2
Enter your choice:2

Consumer consumes item 1
Enter your choice:2
Buffer is empty!
Enter your choice:1

Producer produces item 1
Enter your choice:2

Consumer consumes item 1
Enter your choice:2
Buffer is empty!
Enter your choice:3

...Program finished with exit code 0
```

Experiment 6: Process ID and Parent Process ID Extraction

Objective: To extract the Process ID (PID) and Parent Process ID (PPID) of a process in a C program.

Procedure:

Write a C program to obtain the Process ID (PID) and Parent Process ID (PPID) using system calls.

Display the PID and PPID of the current process.

Explain the significance of PID and PPID in process management.

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main( )
{
int pid;
pid=fork( );
if(pid== -1)
{
perror("fork failed");
exit(0);
}
if(pid==0)
{
printf("\n1-1 Child process is under execution");
printf("\n1-2 Process id of the child process is %d", getpid());
printf("\n1-3 Process id of the parent process is %d",getppid());
}
else
{
printf("\n2-1 Parent process is under execution");
printf("\n2-2 Process id of the parent process is %d", getpid());
printf("\n2-3 Process id of the parent of parent is %d", getppid());
}
return(0);
}
```

SAMPLE OUTPUT:

```
2-1 Parent process is under execution
2-2 Process id of the parent process is 1861
2-3 Process id of the parent of parent is 1860
1-1 Child process is under execution
1-2 Process id of the child process is 1865
1-3 Process id of the parent process is 1861
```