Index

Experiment No: 01

Experiment Name: First-Come, First-Served (FCFS) Scheduling.

Objective: Study to First-Come, First-Served (FCFS) Scheduling in Linux Operating System.

Apparatus Required: Linux Operating System.

Algorithm:

Step 1: Start and Initialize Array for brush time and waiting time.
Step 2: Input the Process no and Brush Time.
Step 3: Set waiting time for first process is 0
Step 4: Calculate waiting for all process using for loop
Step 5: Print the Process no, Brush time, waiting time and turnaround time
Step 6: Stop the program

C Program:

```
#include <stdio.h>
int main(){
    int n, bt[20], wt[20],i,total=0;
    printf("Enter Process Number : ");
    scanf("%d",&n);
    printf("Enter Brust Time of Each Process: \n");
    for(i=0;i<n;i++){
        printf("Process %d : ", i+1);
        scanf("%d",&bt[i]);
    }
    wt[0]=0;
    for(i=0;i<n;i++){
        wt[i+1]= wt[i]+bt[i];
        total += wt[i];
    }
    printf("First Come, First Serve Schedule is .......\n");
    for(i=0;i<n;i++){
        printf("Process No = %d, Brust Time = %d, Waiting Time = %d, Turnaround: %d\n", i+1, bt[i], wt[i],
bt[i]+wt[i]);
    }
    printf("Avarage Waiting Time = %d\n", total / n);
    return 0;
}
```

Execution in Terminal:

```
mrhacker@kali:~/Desktop/Operating-System/vm$ gcc first-come-first-out.c -o first-come-first-out

mrhacker@kali:~/Desktop/Operating-System/vm$ ./first-come-first-out
Enter Process Number : 3
Enter Brust Time of Each Process:
Process 1 : 9
Process 2 : 4
Process 3 : 6
First Come, First Serve Schedule is .......
Process No = 1, Brust Time = 9, Waiting Time = 0, Turnaround: 9
Process No = 2, Brust Time = 4, Waiting Time = 9, Turnaround: 13
Process No = 3, Brust Time = 6, Waiting Time = 13, Turnaround: 19
Avarage Waiting Time = 7
```

Sample Input:

No of Process 3
Brush Time p1 = 9, p2 = 4, p3 = 6

Sample Output:

Process No: 1, Brust Time: 9, Waiting Time: 0, Turnaround: 9
Process No: 2, Brust Time: 4, Waiting Time: 9, Turnaround: 13
Process No: 3, Brust Time: 6, Waiting Time: 13, Turnaround: 19
Avarage Waiting Time: 7

Experiment No: 02

Experiment Name: Shortest-Job-First (SJF) Scheduling.

Objective: Study to Shortest-Job-First (SJF) Scheduling in Linux Operating System.

Apparatus Required: Linux Operating System.

Algorithm:

Step 1: Start and Initialize Array for brush time and waiting time.
Step 2: Input the Process no and Brush Time.
Step 3: Sort those process in ascending order by Brush Time using selection sort
Step 4: Calculate waiting for all process using for loop
Step 5: Print the Process no, Brush time, waiting time and turnaround time
Step 6: Stop the program

C Program:

```
#include <stdio.h>
int main(){
    int n, bt[20], wt[20],i,j,p[20],pos,temp;
    printf("Enter Process Number : ");
    scanf("%d",&n);
    printf("Enter Brust Time of Each Process: \n");
    for(i=0;i<n;i++){
        printf("Process %d : ", i+1);
        scanf("%d",&bt[i]);
        p[i] = i + 1;
    } //sorting algo selection
    for (i = 0; i < n;i++){
        pos = i;
        for (j = i + 1; j < n;j++){
            if(bt[j]<bt[pos])
                pos = j;
        }
        temp = bt[i];
        bt[i] = bt[pos];
        bt[pos] = temp;
        temp = p[i];
        p[i] = p[pos];
        p[pos] = temp;
    }
    wt[0] = 0;
    for(i=0;i<n;i++){
        wt[i+1]= wt[i]+bt[i];
    }
    printf("Shortest Job First ........\n");
    for(i=0;i<n;i++){
```

```
        printf("Process No: %d, Brust Time: %d, Waiting Time: %d, Turnaround: %d\n", p[i], bt[i], wt[i],
bt[i]+wt[i]);
    }
    return 0;
}
```

Execution in Terminal:

```
mrhacker@kali:~/Desktop/Operating-System/vm$ gcc shortest-job.c -o shortest-job

mrhacker@kali:~/Desktop/Operating-System/vm$ ./shortest-job
Enter Process Number : 3
Enter Brust Time of Each Process:
Process 1 : 3
Process 2 : 8
Process 3 : 7
Shortest Job First  .......
Process No: 1, Brust Time: 3, Waiting Time: 0, Turnaround: 3
Process No: 3, Brust Time: 7, Waiting Time: 3, Turnaround: 10
Process No: 2, Brust Time: 8, Waiting Time: 10, Turnaround: 18
```

Sample Input:

No. of process 3
Brush time of process p1 = 3, p2 = 8, p3 = 7

Sample Output:

Shortest Job First .....
Process No: 1, Brust Time: 3, Waiting Time: 0, Turnaround: 3
Process No: 3, Brust Time: 7, Waiting Time: 3, Turnaround: 10
Process No: 2, Brust Time: 8, Waiting Time: 10, Turnaround: 18

Experiment No: 03

Experiment Name: Priority Scheduling

Objective: Study to Priority Scheduling in Linux Operating System.

Apparatus Required: Linux Operating System.

Algorithm:

Step 1: Start and Initialize Array for brush time, Priority and waiting time.
Step 2: Input the Brush Time and Priority of all process .
Step 3: Sort those process according to priority using selection sort.
Step 4: Calculate waiting for all process using for loop
Step 5: Print the Process no, Brush time, waiting time and turnaround time
Step 6: Stop the program

C Program:

```
#include <stdio.h>
int main()
{
    int n, bt[20], wt[20], i, j, p[20],pr[20], pos, temp;
    printf("Enter Process Number : ");
    scanf("%d", &n);
    printf("Enter Brust Time and Priority of Each Process: \n");
    for (i = 0; i < n; i++){
        printf("Process %d : \nBrush Time : ", i + 1);
        scanf("%d", &bt[i]);
        printf("Priority : ");
        scanf("%d", &pr[i]);
        p[i] = i + 1;
    } // sorting algo selection
    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;
    for (i = 0; i < n; i++){
        wt[i + 1] = wt[i] + bt[i];
    }
    for (i = 0; i < n; i++)
    {
        printf("Process No: %d, Brust Time: %d, Waiting Time: %d, Turnaround: %d\n", p[i], bt[i], wt[i], bt[i] +
wt[i]);
    }
    return 0;
}
```

Execution in Terminal:

```
mrhacker@kali:~/Desktop/Operating-System$ gcc priority-scheduling.c -o priority

mrhacker@kali:~/Desktop/Operating-System$ ./priority
Enter Process Number : 4
Enter Brust Time and Priority of Each Process:
Process 1 :
Brush Time : 5
Priority : 2
Process 2 :
Brush Time : 7
Priority : 3
Process 3 :
Brush Time : 9
Priority : 1
Process 4 :
Brush Time : 5
Priority : 2
Process No: 3, Brust Time: 9, Waiting Time: 0, Turnaround: 9
Process No: 1, Brust Time: 5, Waiting Time: 9, Turnaround: 14
Process No: 4, Brust Time: 5, Waiting Time: 14, Turnaround: 19
Process No: 2, Brust Time: 7, Waiting Time: 19, Turnaround: 26
```

Sample Input:

No. of process 4

Brush time and priority

b1= 5, p1 = 2
b2= 7, p2 = 3
b3= 9, p3 = 1
b4= 5, p4 = 2

Sample Output:

Process No: 3, Brust Time: 9, Waiting Time: 0, Turnaround: 9
Process No: 1, Brust Time: 5, Waiting Time: 9, Turnaround: 14
Process No: 4, Brust Time: 5, Waiting Time: 14, Turnaround: 19
Process No: 2, Brust Time: 7, Waiting Time: 19, Turnaround: 26

Experiment No: 04

Experiment Name: Round Robin.

Objective: Study to Round Robin in Linux Operating System.

Apparatus Required: Linux Operating System.

Algorithm:

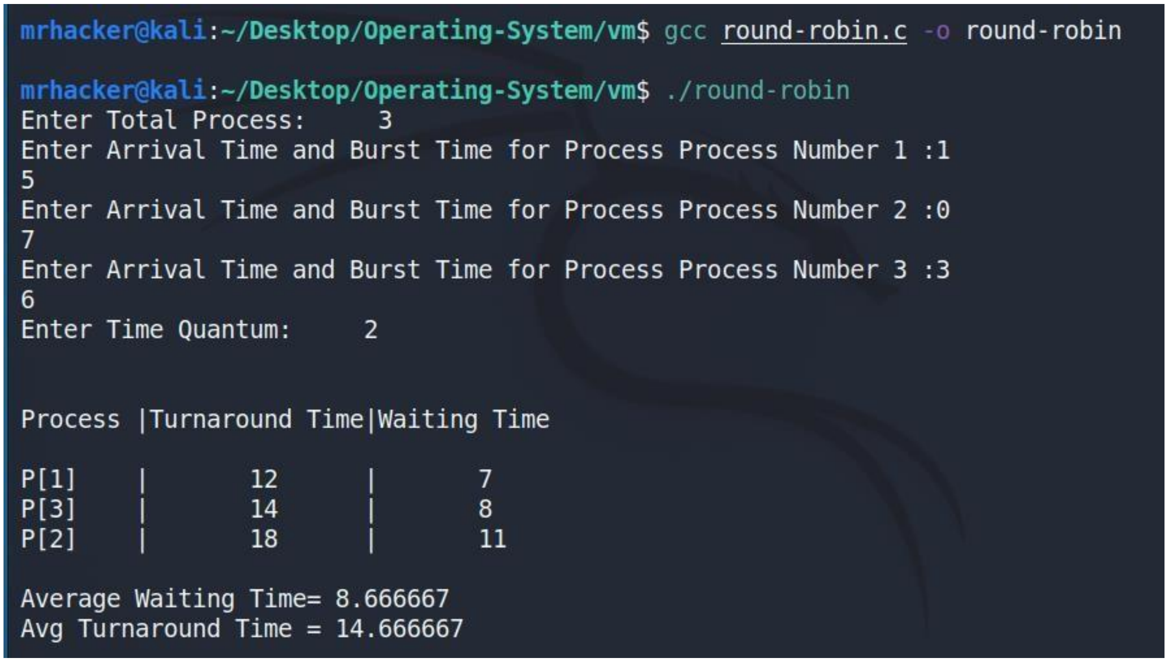
Step 1: Start and Initialize Array for brush time, Priority and waiting time.
Step 2: Input the Brush Time and Priority of all process .
Step 3: Sort those process according to priority using selection sort.
Step 4: Calculate waiting for all process using for loop
Step 5: Print the Process no, Brush time, waiting time and turnaround time
Step 6: Stop the program

C Program:

```
#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;
      }
```

Execution in Terminal:



```
mrhacker@kali:~/Desktop/Operating-System/vm$ gcc round-robin.c -o round-robin

mrhacker@kali:~/Desktop/Operating-System/vm$ ./round-robin
Enter Total Process:     3
Enter Arrival Time and Burst Time for Process Process Number 1 :1
5
Enter Arrival Time and Burst Time for Process Process Number 2 :0
7
Enter Arrival Time and Burst Time for Process Process Number 3 :3
6
Enter Time Quantum:     2


Process |Turnaround Time|Waiting Time

P[1]    |       12      |       7
P[3]    |       14      |       8
P[2]    |       18      |       11

Average Waiting Time= 8.666667
Avg Turnaround Time = 14.666667
```

Sample Input:

No. of process 3

Arrival time and brush time

P1	1	5
P2	0	7
P3	3	6

Sample Output:

Process	Turnaround Time	Waiting Time
P [1]	12	7
P [2]	14	8
P [3]	18	11

Average Waiting Time= 8.6666667

Avg Turnaround Time = 14.666667

Experiment No: 05

Experiment Name: Producer/Consumer Problem

Objective: Study to Producer/Consumer Problem in Linux Operating System.

Apparatus Required: Linux Operating System.

Algorithm:

Step 1: Start the program.
Step 2: Declare the required variables.
Step 3: Initialize the buffer size and get maximum item you want to produce.
Step 4: Get the option, which you want to do either producer, consumer or exit from the operation.
Step 5: If you select the producer, check the buffer size if it is full the producer should not produce the item or otherwise produce the item and increase the value buffer size.
Step 6: If you select the consumer, check the buffer size if it is empty the consumer should not consume the item or otherwise consume the item and decrease the value of buffer size.
Step 7: If you select exit come out of the program.
Step 8: Stop the program.

C Program:

```
#include<stdio.h>
#include<stdlib.h>
int mutex = 1, full = 0, empty = 3, x = 0;
int Wait(int s){
   return --s;
}
int Signal(int s){
   return ++s;
}
void producer(){
   mutex = Wait(mutex);
   full = Signal(mutex);
   empty = Wait(empty);
   x++;
   printf("\nProducer produces item %d\n", x);
   mutex = Signal(mutex);
}
void consumer(){
   mutex = Wait(mutex);
   full = Wait(full);
   empty = Signal(empty);
   printf("\nConsumer Consume item %d\n", x);
   x--;
   mutex = Signal(mutex);
}
```

```
int main(){
    int n;
    while(1){
        printf("Chose Option...\n1.Producer\t2.Consumer\t3.Exit\nEnter Choice : ");
        scanf("%d", &n);
        switch (n)
        {
        case 1:
            if(mutex==1 && empty!=0)
                producer();
            else
                printf("Buffer is Full\n");
            break;
        case 2:
            if (mutex == 1 && full != 0)
                consumer();
            else
                printf("Buffer is Empty\n");
            break;
        case 3:
            exit(0);
        default:
            break;
        }
    }
    return 0;
}
```

Execution in Terminal:

```
mrhacker@kali:~/Desktop/Operating-System/vm$ gcc producer-consumer.c -o producer-consumer

mrhacker@kali:~/Desktop/Operating-System/vm$ ./producer-consumer
Chose Option...
1.Producer      2.Consumer      3.Exit
Enter Choice : 1

Producer produces item 1
Chose Option...
1.Producer      2.Consumer      3.Exit
Enter Choice : 2

Consumer Consume item 1
Chose Option...
1.Producer      2.Consumer      3.Exit
Enter Choice : 1

Producer produces item 1
Chose Option...
1.Producer      2.Consumer      3.Exit
```

Experiment No: 06

Experiment Name: Bounded-Buffer Problem

Objective: Study to Bounded-Buffer Problem in Linux Operating System.

Apparatus Required: Linux Operating System.

Algorithm:

Step 1: Start the program.
Step 2: Declare the required variables.
Step 3: Initialize the buffer size and get maximum item you want to produce.
Step 4: Get the option, which you want to do either producer, consumer or exit from the operation.
Step 5: If you select the producer, check the buffer size if it is full the producer should not produce the item or otherwise produce the item and increase the value buffer size.
Step 6: If you select the consumer, check the buffer size if it is empty the consumer should not consume the item or otherwise consume the item and decrease the value of buffer size.
Step 7: If you select exit come out of the program.
Step 8: Stop the program.

C Program:

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

int mutex = 1, full = 0, empty = 3, x = 0;
int Wait(int s)
{
    return --s;
}
int Signal(int s)
{
    return ++s;
}
void producer()
{
    mutex = Wait(mutex);
    full = Signal(mutex);
    empty = Wait(empty);
    x++;
    printf("\nProducer produces item %d\n", x);
    mutex = Signal(mutex);
}
```

```
void consumer()
{
    mutex = Wait(mutex);
    full = Wait(full);
    empty = Signal(empty);
    printf("\nConsumer Consume item %d\n", x);
    x--;
    mutex = Signal(mutex);
}
int main()
{
    int n;
    while (1)
    {
        printf("Chose Option...\n1.Producer\t2.Consumer\t3.Exit\nEnter Choice : ");
        scanf("%d", &n);
        switch (n)
        {
        case 1:
            if (mutex == 1 && empty != 0)
                producer();
            else
                printf("Buffer is Full\n");
            break;
        case 2:
            if (mutex == 1 && full != 0)
                consumer();
            else
                printf("Buffer is Empty\n");
            break;
        case 3:
            exit(0);
        default:
            break;
        }
    }
    return 0;
}
```

Execution in Terminal:

```
mrhacker@kali:~/Desktop/Operating-System/vm$ gcc bounded-buffer.c -o bounded-buffer

mrhacker@kali:~/Desktop/Operating-System/vm$ ./bounded-buffer
Chose Option...
1.Producer      2.Consumer      3.Exit
Enter Choice : 2
Buffer is Empty
Chose Option...
1.Producer      2.Consumer      3.Exit
Enter Choice : 1

Producer produces item 1
Chose Option...
1.Producer      2.Consumer      3.Exit
Enter Choice : 2

Consumer Consume item 1
Chose Option...
1.Producer      2.Consumer      3.Exit
Enter Choice : 3
```

Experiment No: 07

Experiment Name: Readers and Writers Problem

Objective: Study to Readers and Writers Problem in Linux Operating System.

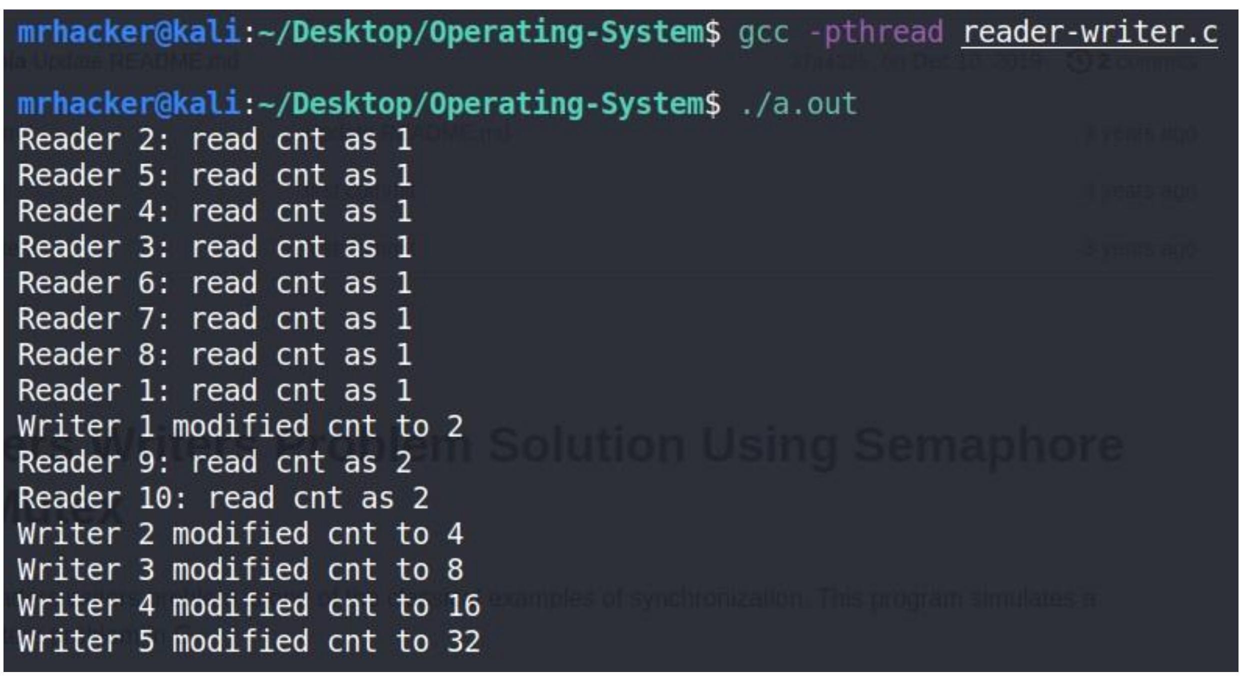
Apparatus Required: Linux Operating System.

C Program:

```
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
sem_t x, y;
pthread_t tid;
pthread_t writerthreads[100], readerthreads[100];
int readercount = 0;

void reader(void param)
{
    sem_wait(&x);
    readercount++;
    if (readercount == 1)
        sem_wait(&y);
    sem_post(&x);
    printf("%d reader is inside\n", readercount);
    usleep(3);
    sem_wait(&x);
    readercount--;
    if (readercount == 0)
    {
        sem_post(&y);
    }
    sem_post(&x);
    printf("%d Reader is leaving\n", readercount + 1);
    return NULL;
}

void writer(void param)
{
    printf("Writer is trying to enter\n");
    sem_wait(&y);
    printf("Writer has entered\n");
    sem_post(&y);
    printf("Writer is leaving\n");
    return NULL;
```

```
}

int main()
{
    int n2, i;
    printf("Enter the number of readers:");
    scanf("%d", &n2);
    printf("\n");
    int n1[n2];
    sem_init(&x, 0, 1);
    sem_init(&y, 0, 1);
    for (i = 0; i < n2; i++)
    {
        pthread_create(&writerthreads[i], NULL, reader, NULL);
        pthread_create(&readerthreads[i], NULL, writer, NULL);
    }
    for (i = 0; i < n2; i++)
    {
        pthread_join(writerthreads[i], NULL);
        pthread_join(readerthreads[i], NULL);
    }
}
```

Execution in Terminal:



```
mrhacker@kali:~/Desktop/Operating-System$ gcc -pthread reader-writer.c

mrhacker@kali:~/Desktop/Operating-System$ ./a.out
Reader 2: read cnt as 1
Reader 5: read cnt as 1
Reader 4: read cnt as 1
Reader 3: read cnt as 1
Reader 6: read cnt as 1
Reader 7: read cnt as 1
Reader 8: read cnt as 1
Reader 1: read cnt as 1
Writer 1 modified cnt to 2
Reader 9: read cnt as 2
Reader 10: read cnt as 2
Writer 2 modified cnt to 4
Writer 3 modified cnt to 8
Writer 4 modified cnt to 16
Writer 5 modified cnt to 32
```

Experiment No: 08

Experiment Name: Dining-Philosophers Problem

Objective: Study to Dining-Philosophers Problem in Linux Operating System.

Apparatus Required: Linux Operating System.

C Program:

```
#include <stdio.h>
#define n 4
int compltedPhilo = 0, i;
struct fork{
    int taken;
} ForkAvil[n];
struct philosp{
    int left;
    int right;
} Philostatus[n];
void goForDinner(int philID){
    if (Philostatus[philID].left == 10 && Philostatus[philID].right == 10)
        printf("Philosopher %d completed his dinner\n", philID + 1);
    else if (Philostatus[philID].left == 1 && Philostatus[philID].right == 1){
        printf("Philosopher %d completed his dinner\n", philID + 1);
        Philostatus[philID].left = Philostatus[philID].right = 10;
        int otherFork = philID - 1;
        if (otherFork == -1)
            otherFork = (n - 1);
        ForkAvil[philID].taken = ForkAvil[otherFork].taken = 0;
        printf("Philosopher %d released fork %d and fork %d\n", philID + 1, philID + 1, otherFork + 1);
        compltedPhilo++;
    }
    else if (Philostatus[philID].left == 1 && Philostatus[philID].right == 0){
        if (philID == (n - 1)){
            if (ForkAvil[philID].taken == 0){
                ForkAvil[philID].taken = Philostatus[philID].right = 1;
                printf("Fork %d taken by philosopher %d\n", philID + 1, philID + 1);
            }
            else{
                printf("Philosopher %d is waiting for fork %d\n", philID + 1, philID + 1);
            }
        }
        else{
            int dupphilID = philID;
            philID -= 1;

            if (philID == -1)
```

```
            philID = (n - 1);

        if (ForkAvil[philID].taken == 0)
        {
            ForkAvil[philID].taken = Philostatus[dupphilID].right = 1;
            printf("Fork %d taken by Philosopher %d\n", philID + 1, dupphilID + 1);
        }
        else
        {
            printf("Philosopher %d is waiting for Fork %d\n", dupphilID + 1, philID + 1);
        }
      }
    }
    else if (Philostatus[philID].left == 0){
      if (philID == (n - 1)){
        if (ForkAvil[philID - 1].taken == 0){
            ForkAvil[philID - 1].taken = Philostatus[philID].left = 1;
            printf("Fork %d taken by philosopher %d\n", philID, philID + 1);
        }
        else{
            printf("Philosopher %d is waiting for fork %d\n", philID + 1, philID);
        }
      }
      else{
        if (ForkAvil[philID].taken == 0){
            ForkAvil[philID].taken = Philostatus[philID].left = 1;
            printf("Fork %d taken by Philosopher %d\n", philID + 1, philID + 1);
        }
        else{
            printf("Philosopher %d is waiting for Fork %d\n", philID + 1, philID + 1);
        }
      }
    }
    else{
    }
}
int main(){
    for (i = 0; i < n; i++)
      ForkAvil[i].taken = Philostatus[i].left = Philostatus[i].right = 0;
    while (compltedPhilo < n){
      for (i = 0; i < n; i++)
        goForDinner(i);
      printf("\nTill now num of philosophers completed dinner are %d\n\n", compltedPhilo);
    }
    return 0;
}
```

Execution in Terminal:

```
mrhacker@kali:~/Desktop/Operating-System/vm$ gcc dining-philosopher.c -o dining

mrhacker@kali:~/Desktop/Operating-System/vm$ ./dining
Fork 1 taken by Philosopher 1
Fork 2 taken by Philosopher 2
Fork 3 taken by Philosopher 3
Philosopher 4 is waiting for fork 3

Till now num of philosophers completed dinner are 0

Fork 4 taken by Philosopher 1
Philosopher 2 is waiting for Fork 1
Philosopher 3 is waiting for Fork 2
Philosopher 4 is waiting for fork 3

Till now num of philosophers completed dinner are 0

Philosopher 1 completed his dinner
Philosopher 1 released fork 1 and fork 4
Fork 1 taken by Philosopher 2
Philosopher 3 is waiting for Fork 2
Philosopher 4 is waiting for fork 3

Till now num of philosophers completed dinner are 1

Philosopher 1 completed his dinner
Philosopher 2 completed his dinner
Philosopher 2 released fork 2 and fork 1
Fork 2 taken by Philosopher 3
Philosopher 4 is waiting for fork 3

Till now num of philosophers completed dinner are 2

Philosopher 1 completed his dinner
Philosopher 2 completed his dinner
Philosopher 3 completed his dinner
Philosopher 3 released fork 3 and fork 2
Fork 3 taken by philosopher 4

Till now num of philosophers completed dinner are 3

Philosopher 1 completed his dinner
Philosopher 2 completed his dinner
Philosopher 3 completed his dinner
Fork 4 taken by philosopher 4

Till now num of philosophers completed dinner are 3

Philosopher 1 completed his dinner
Philosopher 2 completed his dinner
Philosopher 3 completed his dinner
Philosopher 4 completed his dinner
Philosopher 4 released fork 4 and fork 3

Till now num of philosophers completed dinner are 4
```

Experiment No: 09

Experiment Name: The Sleeping Barber Problem

Objective: Study to The Sleeping Barber Problem in Linux Operating System.

Apparatus Required: Linux Operating System.

C Program:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>
#include <time.h>
#include <sys/types.h>
#include <sys/time.h>

void *barber_function(void *idp);
void *customer_function(void *idp);
void serve_customer();
void *make_customer_function();

pthread_mutex_t srvCust;

sem_t barber_ready;
sem_t customer_ready;
sem_t modifySeats;

int chair_cnt;
int total_custs;

int available_seats;
int no_served_custs = 0;
time_t waiting_time_sum;

void *barber_function(void *idp){
    int counter = 0;
    while (1){
        sem_wait(&customer_ready);
        sem_wait(&modifySeats);
        available_seats++;
        sem_post(&modifySeats);
        sem_post(&barber_ready);
        pthread_mutex_lock(&srvCust);
        serve_customer();
        pthread_mutex_unlock(&srvCust);
```

```
        printf("Customer was served.\n");
        counter++;
        if (counter == (total_custs - no_served_custs))
            break;
    }
    pthread_exit(NULL);
}

void *customer_function(void *idp){
    struct timeval start, stop;
    sem_wait(&modifySeats);
    if (available_seats >= 1){
        available_seats--;

        printf("Customer[pid = %lu] is waiting.\n", pthread_self());
        printf("Available seats: %d\n", available_seats);
        gettimeofday(&start, NULL);
        sem_post(&customer_ready);
        sem_post(&modifySeats);
        sem_wait(&barber_ready);
        gettimeofday(&stop, NULL);

        double sec = (double)(stop.tv_usec - start.tv_usec) / 1000000 + (double)(stop.tv_sec - start.tv_sec);

        waiting_time_sum += 1000 * sec;
        printf("Customer[pid = %lu] is being served. \n", pthread_self());
    }
    else{
        sem_post(&modifySeats);
        no_served_custs++;
        printf("A Customer left.\n");
    }

    pthread_exit(NULL);
}
void serve_customer()
{
    int s = rand() % 401;
    s = s * 1000;
    usleep(s);
}

void *make_customer_function(){
    int tmp;
    int counter = 0;

    while (counter < total_custs){
        pthread_t customer_thread;
```

```
        tmp = pthread_create(&customer_thread, NULL, (void *)customer_function, NULL);
        if (tmp)
            printf("Failed to create thread.");
        counter++;
        usleep(100000);
    }
}

int main()
{
    srand(time(NULL));
    pthread_t barber_1;

    pthread_t customer_maker;

    int tmp;

    pthread_mutex_init(&srvCust, NULL);
    sem_init(&customer_ready, 0, 0);
    sem_init(&barber_ready, 0, 0);
    sem_init(&modifySeats, 0, 1);

    printf("Please enter the number of seats: \n");
    scanf("%d", &chair_cnt);

    printf("Please enter the total customers: \n");
    scanf("%d", &total_custs);

    available_seats = chair_cnt;
    tmp = pthread_create(&barber_1, NULL, (void *)barber_function, NULL);

    if (tmp)
        printf("Failed to create thread.");
    tmp = pthread_create(&customer_maker, NULL, (void *)make_customer_function, NULL);
    if (tmp)
        printf("Failed to create thread.");
    pthread_join(barber_1, NULL);
    pthread_join(customer_maker, NULL);

    printf("\n------------------------------------------------\n");
    printf("Average customers' waiting time: %f ms.\n", (waiting_time_sum / (double)(total_custs -
no_served_custs)));
    printf("Number of customers that were forced to leave: %d\n", no_served_custs);
}
```

Execution in Terminal:

```
mrhacker@kali:~/Desktop/Operating-System/vm$ gcc -pthread sleeping-barber.c

mrhacker@kali:~/Desktop/Operating-System/vm$ ./a.out
Please enter the number of seats:
3
Please enter the total customers:
4
Customer[pid = 140484444624448] is waiting.
Available seats: 2
Customer[pid = 140484444624448] is being served.
Customer was served.
Customer[pid = 140484436100672] is waiting.
Available seats: 2
Customer[pid = 140484436100672] is being served.
Customer[pid = 140484353717824] is waiting.
Available seats: 2
Customer[pid = 140484345325120] is waiting.
Available seats: 1
Customer was served.
Customer[pid = 140484353717824] is being served.
Customer was served.
Customer[pid = 140484345325120] is being served.
Customer was served.

------------------------------------------------
Average customers' waiting time: 133.500000 ms.
Number of customers that were forced to leave: 0
```

Sample Input:

No. of seat = 3
No. of Customer = 4

Experiment No: 10

Experiment Name: Resource-Allocation Graph Algorithm

Objective: Study to Resource-Allocation Graph Algorithm in Linux Operating System.

Apparatus Required: Linux Operating System.

Algorithm:

Step-01: First, find the currently available instances of each resource.

Step-02: Check for each process which can be executed using the allocated + available resource.

Step-03: Add the allocated resource of the executable process to the available resources and terminate it.

Step-04: Repeat the 2nd and 3rd steps until the execution of each process.

Step-05: If at any step, none of the processes can be executed then there is a deadlock in the system.

C Program:

```
#include <stdio.h>
int proc, res, i, j, row = 0, flag = 0;
static int pro[3][3], req[3][3], st_req[3][3], st_pro[3][3];

int main()
{

    printf("\nEnter the number of Processes:");
    scanf("%d", &proc);

    printf("\nEnter the number of Resources:");
    scanf("%d", &res);

    printf("\nEnter the Process Matrix:");
    for (i = 0; i < proc; i++)
        for (j = 0; j < res; j++)
            scanf("%d", &pro[i][j]);

    printf("\nEnter the Request Matrix:");
    for (i = 0; i < res; i++)
        for (j = 0; j < proc; j++)
            scanf("%d", &req[i][j]);

    row = 0;
        for (i = 0; i < res; i++){
            if (pro[row][i] == 1){
                if (st_pro[row][i] > 1 && flag == 1){
```

```
                printf("\nDeadlock Occured");
                return 0;
            }
            st_pro[row][i]++;
            row = i;
            break;
        }
    }
    for (i = 0; i < proc; i++){
        if (req[row][i] == 1){
            if (st_req[row][i] > 1){
                printf("\nDeadlock Occured");
                return 0;
            }
            st_req[row][i]++;
            row = i;
            flag = 1;
            break;
        }
    }

printf("\nNo Deadlock Detected");

return 0;
}
```

Execution in Terminal:

```
mrhacker@kali:~/Desktop/Operating-System/vm$ gcc resource-allocation-graph.c -o resource

mrhacker@kali:~/Desktop/Operating-System/vm$ ./resource

Enter the number of Processes:3

Enter the number of Resources:3

Enter the Process Matrix:1 0 0
0 0 1
0 1 0

Enter the Request Matrix:0 1 0
1 0 0
0 0 1

No Deadlock Detected
```

Input:

Process	Allocation Resource			Request Resource		
	R1	R2	R3	R1	R2	R3
P1	1	0	0	0	1	0
P2	0	0	1	1	0	0
P3	0	1	0	0	0	1

Output:

No Deadlock

Experiment No: 11

Experiment Name: Banker’s Algorithm

Objective: Study to The Banker’s Algorithm in Linux Operating System.

Apparatus Required: Linux Operating System.

Algorithm:

Step-1: Start the program.
Step-2: Declare the memory for the process.
Step-3: Read the number of process, resources, allocation matrix and available matrix.
Step-4: Compare each and every process using the banker‟s algorithm.
Step-5: If the process is in safe state then it is a not a deadlock process otherwise it is a deadlock process
Step-6: produce the result of state of process
Step-7: Stop the program

C Program:

```
#include <stdio.h>
int main(){
   // P0, P1, P2, P3, P4 are the names of Process
   int n, r, i, j, k;
   n = 5;                         // Indicates the Number of processes
   r = 3;                         // Indicates the Number of resources
   int alloc[5][3] = {{0, 0, 1}, // P0 // This is Allocation Matrix
                      {3, 0, 0},  // P1
                      {1, 0, 1},  // P2
                      {2, 3, 2},  // P3
                      {0, 0, 3}}; // P4
   int max[5][3] = {{7, 6, 3}, // P0 // MAX Matrix
                    {3, 2, 2},  // P1
                    {8, 0, 2},  // P2
                    {2, 1, 2},  // P3
                    {5, 2, 3}}; // P4
   int avail[3] = {2, 3, 2}; // These are Available Resources
   int f[n], ans[n], ind = 0;
   for (k = 0; k < n; k++){
      f[k] = 0;
   }
   int need[n][r];
   for (i = 0; i < n; i++){
      for (j = 0; j < r; j++)
         need[i][j] = max[i][j] - alloc[i][j];
   }
   int y = 0;
```

```
    for (k = 0; k < 5; k++){
        for (i = 0; i < n; i++)
        {
            if (f[i] == 0)
            {

                int flag = 0;
                for (j = 0; j < r; j++){
                    if (need[i][j] > avail[j]){
                        flag = 1;
                        break;
                    }
                }
                if (flag == 0){
                    ans[ind++] = i;
                    for (y = 0; y < r; y++)
                        avail[y] += alloc[i][y];
                    f[i] = 1;
                }
            }
        }
    }
    printf("Th SAFE Sequence is as follows\n");
    for (i = 0; i < n - 1; i++)
        printf(" P%d ->", ans[i]);
    printf(" P%d\n", ans[n - 1]);
    return (0);
}
```

Execution in Terminal:

```
mrhacker@kali:~/Desktop/Operating-System/vm$ gcc banckers.c -o banckers

mrhacker@kali:~/Desktop/Operating-System/vm$ ./banckers
Th SAFE Sequence is as follows
 P1 -> P3 -> P4 -> P0 -> P2
```