OS LAB

ASSIGNMENT – 3

Rohin Srivastava

19BIT0177

L29 + L30

1. A pair of processes involved in exchanging a sequence of integers. The number of integers that can be produced and consumed at a time is limited to 100. Write a Program to implement the producer and consumer problem using POSIX semaphore for the above scenario.

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#define SIZE 100
#define NUMB_THREADS 10
#define PRODUCER_LOOPS 2
#define CONSUMER_LOOPS 2
#define TRUE 1
#define FALSE 0
typedef int buffer_t;
buffer_t buffer[SIZE];
int buffer_index;
pthread_mutex_t buffer_mutex;
sem_t full_sem;
sem_t empty_sem;
void insertbuffer(buffer_t value)
{
if (buffer_index < SIZE)
{
buffer[buffer_index++] = value;
}
else
{
printf("Buffer overflow\n");
}
}
buffer_t dequeuebuffer()
{
if (buffer_index > 0)
{
return buffer[--buffer_index];
}
else
{
printf("Buffer underflow\n"); }
```

```
return 0;
}
int isempty() {
if (buffer_index == 0)
return TRUE;return FALSE;
}
int isfull() {
if (buffer_index == SIZE)
return TRUE;
return FALSE;
}
void *producer2(void *thread_n)
{
int thread_numb = *(int *)thread_n;
buffer_t value;
int i=0;
while (i++ < PRODUCER_LOOPS)
{
sleep(rand() % 10);
value = rand() % 100;
pthread_mutex_lock(&buffer_mutex);
do
{
pthread_mutex_unlock(&buffer_mutex);
sem_wait(&full_sem);
pthread_mutex_lock(&buffer_mutex);
} while (isfull());
insertbuffer(value);
pthread_mutex_unlock(&buffer_mutex);
sem_post(&empty_sem);
printf("Producer %d added %d to buffer\n", thread_numb, value);
}
pthread_exit(0);
}
void *consumer2(void *thread_n)
{
int thread_numb = *(int *)thread_n;
buffer_t value;
int i=0;
while (i++ < PRODUCER_LOOPS)
{
pthread_mutex_lock(&buffer_mutex);
do {
pthread_mutex_unlock(&buffer_mutex);
sem_wait(&empty_sem);
pthread_mutex_lock(&buffer_mutex);
} while (isempty());
value = dequeuebuffer(value);
```

```
pthread_mutex_unlock(&buffer_mutex);
sem_post(&full_sem); // post (increment) fullbuffer semaphore
printf("Consumer %d dequeue %d from buffer\n", thread_numb, value);
}
pthread_exit(0);
}
int main(int argc, char **argv){
buffer_index = 0;
pthread_mutex_init(&buffer_mutex, NULL);
sem_init(&full_sem,0,SIZE); // unsigned int value. Initial value
sem_init(&empty_sem,0,0);
pthread_t thread[NUMB_THREADS];
int thread_numb[NUMB_THREADS];
int i;
for (i = 0; i < NUMB_THREADS; )
{
thread_numb[i] = i;
pthread_create(thread + i,NULL,producer2,thread_numb + i); // void *arg
i++;
thread_numb[i] = i;
pthread_create(&thread[i],NULL,consumer2,&thread_numb[i]); // void *arg
i++;
}
for (i = 0; i < NUMB_THREADS; i++)
pthread_join(thread[i], NULL);
pthread_mutex_destroy(&buffer_mutex);
sem_destroy(&full_sem);
sem_destroy(&empty_sem);
return 0;
}
```

OUTPUT:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#define SIZE 100
#define NUMB_THREADS 10
#define PRODUCER_LOOPS 2
#define CONSUMER_LOOPS 2
#define TRUE 1
#define FALSE 0
typedef int buffer_t;
buffer_t buffer[SIZE];
int buffer_index;
pthread_mutex_t buffer_mutex;
sem_t full_sem;
sem_t empty_sem;
void insertbuffer(buffer_t value)
{
if (buffer_index < SIZE)
{
buffer[buffer_index++] = value;
}
else
{
printf("Buffer overflow\n");
}
}
buffer_t dequeuebuffer()
{
if (buffer_index > 0)
{
return buffer[--buffer_index];
}
else
{
printf("Buffer underflow\n"); }
return 0;
}
int isempty() {
if (buffer_index == 0)
return TRUE;return FALSE;
}
int isfull() {
if (buffer_index == SIZE)
return TRUE;
return FALSE;
}
void *producer2(void *thread_n)
{
int thread_numb = *(int *)thread_n;
buffer_t value;
int i=0;
while (i++ < PRODUCER_LOOPS)
{
sleep(rand() % 10);
value = rand() % 100;
pthread_mutex_lock(&buffer_mutex);
do
{
pthread_mutex_unlock(&buffer_mutex);
sem_wait(&full_sem);
pthread_mutex_lock(&buffer_mutex);
} while (isfull());
insertbuffer(value);
pthread_mutex_unlock(&buffer_mutex);
sem_post(&empty_sem);
printf("Producer %d added %d to buffer\n", thread_numb, value);
}
pthread_exit(0);
}
```

```c
void *consumer2(void *thread_n)
{
int thread_numb = *(int *)thread_n;
buffer_t value;
int i=0;
while (i++ < PRODUCER_LOOPS)
{
pthread_mutex_lock(&buffer_mutex);
do {
pthread_mutex_unlock(&buffer_mutex);
sem_wait(&empty_sem);
pthread_mutex_lock(&buffer_mutex);
} while (isempty());
value = dequeuebuffer(value);
pthread_mutex_unlock(&buffer_mutex);
sem_post(&full_sem); // post (increment) fullbuffer semaphore
printf("Consumer %d dequeue %d from buffer\n", thread_numb, value);
}
pthread_exit(0);
}
int main(int argc, char **argv){
buffer_index = 0;
pthread_mutex_init(&buffer_mutex, NULL);
sem_init(&full_sem,0,SIZE); // unsigned int value. Initial value
sem_init(&empty_sem,0,0);
pthread_t thread[NUMB_THREADS];
int thread_numb[NUMB_THREADS];
int i;
for (i = 0; i < NUMB_THREADS; )
{
thread_numb[i] = i;
pthread_create(thread + i,NULL,producer2,thread_numb + i); // void *arg
i++;
thread_numb[i] = i;
pthread_create(&thread[i],NULL,consumer2,&thread_numb[i]); // void *arg
i++;
}
for (i = 0; i < NUMB_THREADS; i++)
pthread_join(thread[i], NULL);
pthread_mutex_destroy(&buffer_mutex);
sem_destroy(&full_sem);
sem_destroy(&empty_sem);
return 0;
}
```

```
main.c:55:1: warning: implicit declaration of function ‘sleep’ [-Wimplicit-function-declaration]
Producer 0 added 35 to buffer
Consumer 1 dequeue 35 from buffer
Producer 8 added 92 to buffer
Consumer 5 dequeue 92 from buffer
Producer 2 added 21 to buffer
Consumer 3 dequeue 21 from buffer
Producer 4 added 27 to buffer
Producer 4 added 59 to buffer
Consumer 7 dequeue 59 from buffer
Consumer 7 dequeue 27 from buffer
Producer 6 added 63 to buffer
Consumer 1 dequeue 63 from buffer
Producer 2 added 40 to buffer
Consumer 5 dequeue 40 from buffer
Producer 0 added 26 to buffer
Consumer 3 dequeue 26 from buffer
Producer 8 added 72 to buffer
Consumer 9 dequeue 72 from buffer
Producer 6 added 36 to buffer
Consumer 9 dequeue 36 from buffer
```

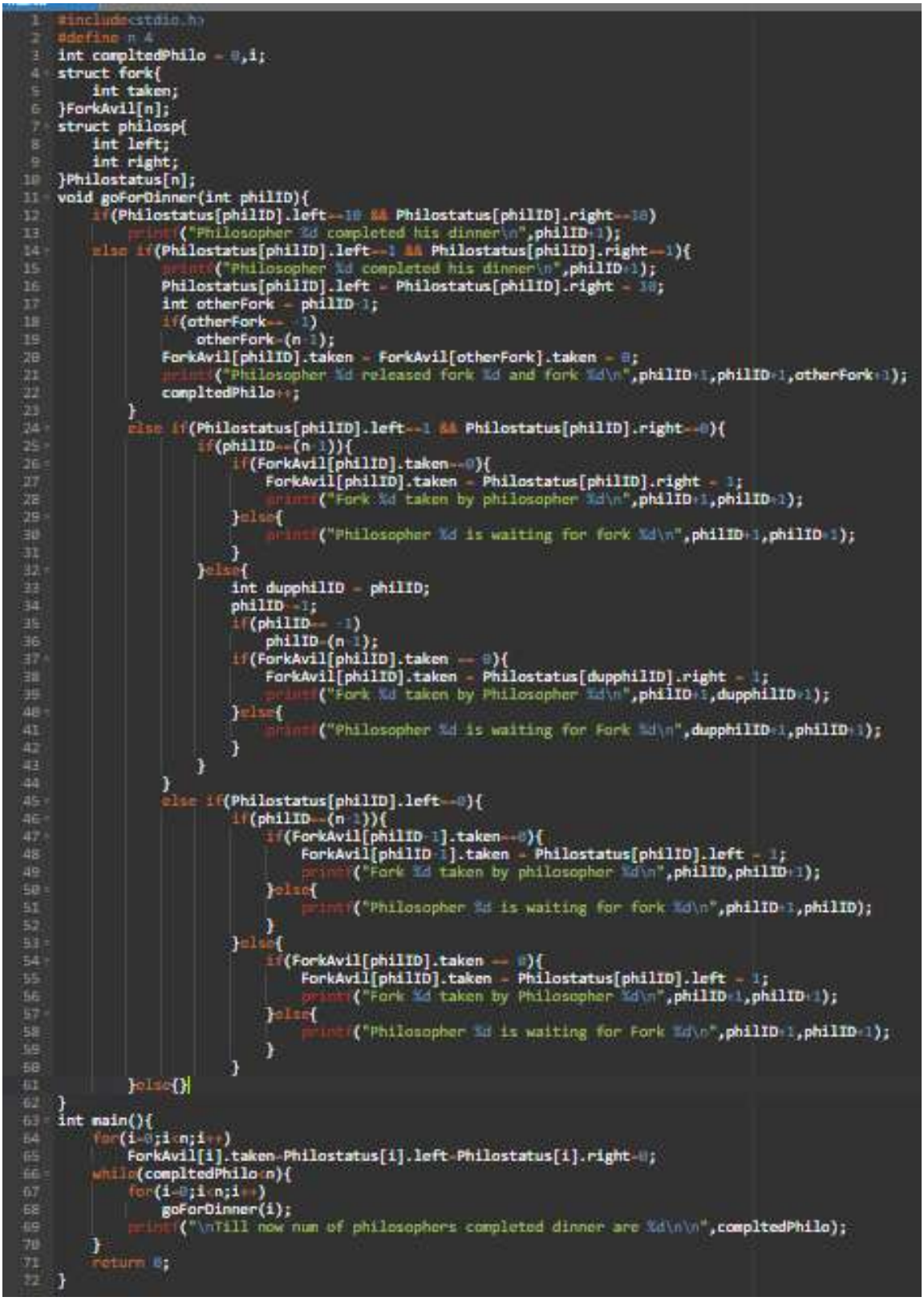
2. Write a Program to implement the solution for dining philosopher’s problem.

CODE:

```
#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;
}
```

OUTPUT:



```c
#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;
}
```

```
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



...Program finished with exit code 0
Press ENTER to exit console.
```

3. Servers can be designed to limit the number of open connections. For example, a server may wish to have only N socket connections at any point in time. As soon as N connections are made, the server will not accept another incoming connection until an existing connection is released. Write a program to illustrate how semaphores can be used by a server to limit the number of concurrent connections.

CODE:

```
#include<semaphore.h>
#include<pthread.h>
#include<stdio.h>
#include<stdlib.h>
#define size 100
pthread_mutex_t mutex;
pthread_t acquire[100],release[100];
sem_t full,empty;
int count;
int buffer[size];
void initial()
{
   pthread_mutex_init(&mutex,NULL);
   sem_init(&full,1,0);
   sem_init(&empty,1,size);
   count=0;
}
void add_connection(int integer)
{
```

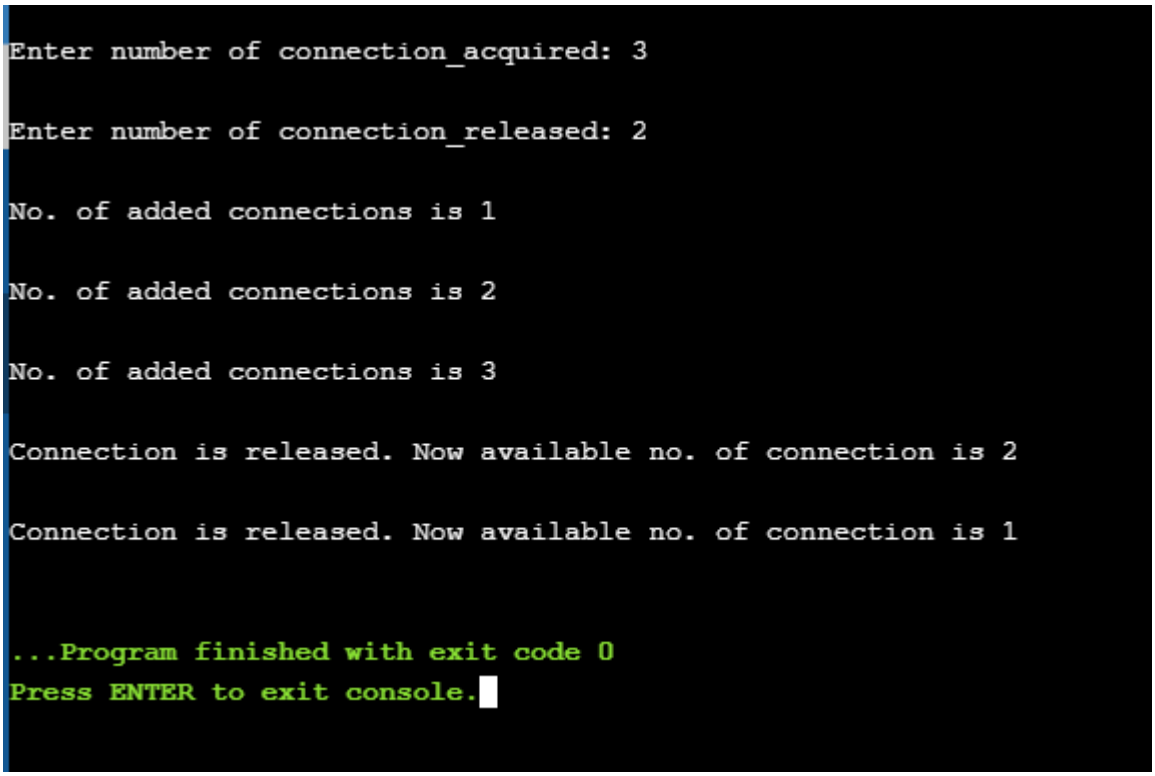
```
    buffer[count++]=integer;
}
int release_connection()
{
    buffer[count]=0;
    return(count--);
}
void *connection_acquired(void *p){
    int wait_time,integer,i;
    integer=1;
    wait_time=rand()%11;
    sem_wait(&empty);
    pthread_mutex_lock(&mutex);
    printf("\nNo. of added connections is %d\n",count+1);
    add_connection(integer);
    pthread_mutex_unlock(&mutex);
    sem_post(&full);
}
void *connection_released (void *p)
{
    int wait_time,integer;
    wait_time=rand()%11;
    sem_wait(&full);
    pthread_mutex_lock(&mutex);
    integer= release_connection();
    printf("\nConnection is released. Now available no. of connection is %d\n",count);
    pthread_mutex_unlock(&mutex);
    sem_post(&empty);
}
int main()
{
```

```
    int n,m,i;
    initial();
    printf("\nEnter number of connection_acquired: ");
    scanf("%d",&n);
    printf("\nEnter number of connection_released: ");
    scanf("%d",&m);
    for(i=0;i<n;i++)pthread_create(&acquire[i],NULL,connection_acquired,NULL);
    for(i=0;i<m;i++)
    pthread_create(&release[i],NULL,connection_released,NULL);
    for(i=0;i<n;i++)
    pthread_join(acquire[i],NULL);
    for(i=0;i<m;i++)
    pthread_join(release[i],NULL);
    exit(0);
}
```

OUTPUT:

main.c

```c
#include<semaphore.h>
#include<pthread.h>
#include<stdio.h>
#include<stdlib.h>
#define size 100
pthread_mutex_t mutex;
pthread_t acquire[100],release[100];
sem_t full,empty;
int count;
int buffer[size];
void initial()
{
    pthread_mutex_init(&mutex,NULL);
    sem_init(&full,1,0);
    sem_init(&empty,1,size);
    count=0;
}
void add_connection(int integer)
{
    buffer[count++]=integer;
}
int release_connection()
{
    buffer[count]=0;
    return(count--);
}
void *connection_acquired(void *p){
    int wait_time,integer,i;
    integer=1;
    wait_time=rand()%11;
    sem_wait(&empty);
    pthread_mutex_lock(&mutex);
    printf("\nNo. of added connections is %d\n",count+1);
    add_connection(integer);
    pthread_mutex_unlock(&mutex);
    sem_post(&full);
}
void *connection_released (void *p)
{
    int wait_time,integer;
    wait_time=rand()%11;
    sem_wait(&full);
    pthread_mutex_lock(&mutex);
    integer= release_connection();
    printf("\nConnection is released. Now available no. of connection is %d\n",count);
    pthread_mutex_unlock(&mutex);
    sem_post(&empty);
}
int main()
{
    int n,m,i;
    initial();
    printf("\nEnter number of connection_acquired: ");
    scanf("%d",&n);
    printf("\nEnter number of connection_released: ");
    scanf("%d",&m);
    for(i=0;i<n;i++)pthread_create(&acquire[i],NULL,connection_acquired,NULL);
    for(i=0;i<m;i++)
    pthread_create(&release[i],NULL,connection_released,NULL);
    for(i=0;i<n;i++)
    pthread_join(acquire[i],NULL);
    for(i=0;i<m;i++)
    pthread_join(release[i],NULL);
    exit(0);
}
```


Enter number of connection_acquired: 3
Enter number of connection_released: 2
No. of added connections is 1
No. of added connections is 2
No. of added connections is 3
Connection is released. Now available no. of connection is 2
Connection is released. Now available no. of connection is 1
...Program finished with exit code 0
Press ENTER to exit console.

4. Write a Program to implement banker's algorithm for Deadlock avoidance.

CODE:

```
#include <stdio.h>
int main()
{
   int n, m, i, j, k;
   n = 5;
   m = 3;
   int alloc[5][3] = { { 0, 1, 0 }, // P0    // Allocation Matrix
                  { 2, 0, 0 }, // P1
                  { 3, 0, 2 }, // P2
                  { 2, 1, 1 }, // P3
                  { 0, 0, 2 } }; // P4
   int max[5][3] = { { 7, 5, 3 }, // P0    // MAX Matrix
                { 3, 2, 2 }, // P1
                { 9, 0, 2 }, // P2
                { 2, 2, 2 }, // P3
                { 4, 3, 3 } }; // P4
   int avail[3] = { 3, 3, 2 };
   int f[n], ans[n], ind = 0;
   for (k = 0; k < n; k++) {
      f[k] = 0;
   }
   int need[n][m];
   for (i = 0; i < n; i++) {
      for (j = 0; j < m; 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 < m; j++) {
               if (need[i][j] > avail[j]){
                  flag = 1;
                   break;
```

```
                }
            }
            if (flag == 0) {
                ans[ind++] = i;
                for (y = 0; y < m; y++)
                    avail[y] += alloc[i][y];
                f[i] = 1;
            }
        }
    }
    printf("Following is the SAFE Sequence\n");
    for (i = 0; i < n - 1; i++)
        printf(" P%d ->", ans[i]);
    printf(" P%d", ans[n - 1]);
    return (0);
}
```

OUTPUT:

```c
#include <stdio.h>
int main()
{
    int n, m, i, j, k;
    n = 5;
    m = 3;
    int alloc[5][3] = { { 0, 1, 0 }, // P0    // Allocation Matrix
                        { 2, 0, 0 }, // P1
                        { 3, 0, 2 }, // P2
                        { 2, 1, 1 }, // P3
                        { 0, 0, 2 } }; // P4
    int max[5][3] = { { 7, 5, 3 }, // P0    // MAX Matrix
                      { 3, 2, 2 }, // P1
                      { 9, 0, 2 }, // P2
                      { 2, 2, 2 }, // P3
                      { 4, 3, 3 } }; // P4
    int avail[3] = { 3, 3, 2 };
    int f[n], ans[n], ind = 0;
    for (k = 0; k < n; k++) {
        f[k] = 0;
    }
    int need[n][m];
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; 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 < m; j++) {
                    if (need[i][j] > avail[j]){
                        flag = 1;
                        break;
                    }
                }
                if (flag == 0) {
                    ans[ind++] = i;
                    for (y = 0; y < m; y++)
                        avail[y] += alloc[i][y];
                    f[i] = 1;
                }
            }
        }
    }
    printf("Following is the SAFE Sequence\n");
    for (i = 0; i < n - 1; i++)
        printf(" P%d ->", ans[i]);
    printf(" P%d", ans[n - 1]);
    return (0);
}
```

```
Following is the SAFE Sequence
 P1 -> P3 -> P4 -> P0 -> P2

...Program finished with exit code 0
Press ENTER to exit console.
```