

# **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:

```
main.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <semaphore.h>
5 #define SIZE 100
6 #define NUMB_THREADS 10
7 #define PRODUCER_LOOPS 2
8 #define CONSUMER_LOOPS 2
9 #define TRUE 1
10 #define FALSE 0
11 typedef int buffer_t;
12 buffer_t buffer[SIZE];
13 int buffer_index;
14 pthread_mutex_t buffer_mutex;
15 sem_t full_sem;
16 sem_t empty_sem;
17 void insertbuffer(buffer_t value)
18 {
19     if (buffer_index < SIZE)
20     {
21         buffer[buffer_index++] = value;
22     }
23     else
24     {
25         printf("Buffer overflow\n");
26     }
27 }
28 buffer_t dequeuebuffer()
29 {
30     if (buffer_index > 0)
31     {
32         return buffer[--buffer_index];
33     }
34     else
35     {
36         printf("Buffer underflow\n"); }
37     return 0;
38 }
39 int isempty() {
40     if (buffer_index == 0)
41         return TRUE;return FALSE;
42 }
43 int isfull() {
44     if (buffer_index == SIZE)
45         return TRUE;
46     return FALSE;
47 }
48 void *producer2(void *thread_n)
49 {
50     int thread_numb = *(int *)thread_n;
51     buffer_t value;
52     int i=0;
53     while (i++ < PRODUCER_LOOPS)
54     {
55         sleep(rand() % 10);
56         value = rand() % 100;
57         pthread_mutex_lock(&buffer_mutex);
58         do
59         {
60             pthread_mutex_unlock(&buffer_mutex);
61             sem_wait(&full_sem);
62             pthread_mutex_lock(&buffer_mutex);
63         } while (isfull());
64         insertbuffer(value);
65         pthread_mutex_unlock(&buffer_mutex);
66         sem_post(&empty_sem);
67         printf("Producer %d added %d to buffer\n", thread_numb, value);
68     }
69     pthread_exit(0);
70 }
71 void *consumer2(void *thread_n)
```

```

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

```



OUTPUT:

```

1 #include <stdio.h>
2 #define n 4
3 int completedPhilo = 0;
4 struct fork{
5     int taken;
6 }ForkAvil[n];
7 struct philosp{
8     int left;
9     int right;
10 }Philostatus[n];
11 void goForDinner(int philID){
12     if(Philostatus[philID].left--10 && Philostatus[philID].right--10)
13         printf("Philosopher %d completed his dinner\n", philID+1);
14     else if(Philostatus[philID].left--1 && Philostatus[philID].right--1){
15         printf("Philosopher %d completed his dinner\n", philID+1);
16         Philostatus[philID].left = Philostatus[philID].right = 10;
17         int otherFork = philID+1;
18         if(otherFork--1)
19             otherFork--(n-1);
20         ForkAvil[philID].taken = ForkAvil[otherFork].taken = 0;
21         printf("Philosopher %d released fork %d and fork %d\n", philID+1, philID+1, otherFork+1);
22         completedPhilo++;
23     }
24     else if(Philostatus[philID].left--1 && Philostatus[philID].right--0){
25         if(philID--(n-1)){
26             if(ForkAvil[philID].taken--0){
27                 ForkAvil[philID].taken = Philostatus[philID].right = 1;
28                 printf("Fork %d taken by philosopher %d\n", philID+1, philID+1);
29             }else{
30                 printf("Philosopher %d is waiting for fork %d\n", philID+1, philID+1);
31             }
32         }else{
33             int dupphilID = philID;
34             philID--1;
35             if(philID--(n-1))
36                 philID--(n-1);
37             if(ForkAvil[philID].taken == 0){
38                 ForkAvil[philID].taken = Philostatus[dupphilID].right = 1;
39                 printf("Fork %d taken by Philosopher %d\n", philID+1, dupphilID+1);
40             }else{
41                 printf("Philosopher %d is waiting for Fork %d\n", dupphilID+1, philID+1);
42             }
43         }
44     }
45     else if(Philostatus[philID].left--0){
46         if(philID--(n-1)){
47             if(ForkAvil[philID+1].taken--0){
48                 ForkAvil[philID+1].taken = Philostatus[philID].left = 1;
49                 printf("Fork %d taken by philosopher %d\n", philID, philID+1);
50             }else{
51                 printf("Philosopher %d is waiting for fork %d\n", philID+1, philID);
52             }
53         }else{
54             if(ForkAvil[philID].taken == 0){
55                 ForkAvil[philID].taken = Philostatus[philID].left = 1;
56                 printf("Fork %d taken by Philosopher %d\n", philID+1, philID+1);
57             }else{
58                 printf("Philosopher %d is waiting for Fork %d\n", philID+1, philID+1);
59             }
60         }
61     }else{}
62 }
63 int main(){
64     for(i=0; i<n; i++)
65         ForkAvil[i].taken = Philostatus[i].left = Philostatus[i].right = 0;
66     while(completedPhilo < n){
67         for(i=0; i<n; i++)
68             goForDinner(i);
69         printf("\nTill now num of philosophers completed dinner are %d\n\n", completedPhilo);
70     }
71     return 0;
72 }

```

```
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
1 #include<semaphore.h>
2 #include<pthread.h>
3 #include<stdio.h>
4 #include<stdlib.h>
5 #define size 100
6 pthread_mutex_t mutex;
7 pthread_t acquire[100],release[100];
8 sem_t full,empty;
9 int count;
10 int buffer[size];
11 void initial()
12 {
13     pthread_mutex_init(&mutex,NULL);
14     sem_init(&full,1,0);
15     sem_init(&empty,1,size);
16     count=0;
17 }
18 void add_connection(int integer)
19 {
20     buffer[count++]=integer;
21 }
22 int release_connection()
23 {
24     buffer[count]--;
25     return(count--);
26 }
27 void *connection_acquired(void *p){
28     int wait_time,integer,i;
29     integer=1;
30     wait_time=rand()%11;
31     sem_wait(&empty);
32     pthread_mutex_lock(&mutex);
33     printf("\nNo. of added connections is %d\n",count+1);
34     add_connection(integer);
35     pthread_mutex_unlock(&mutex);
36     sem_post(&full);
37 }
38 void *connection_released (void *p)
39 {
40     int wait_time,integer;
41     wait_time=rand()%11;
42     sem_wait(&full);
43     pthread_mutex_lock(&mutex);
44     integer=release_connection();
45     printf("\nConnection is released. Now available no. of connection is %d\n",count);
46     pthread_mutex_unlock(&mutex);
47     sem_post(&empty);
48 }
49 int main()
50 {
51     int n,m,i;
52     initial();
53     printf("\nEnter number of connection_acquired: ");
54     scanf("%d",&n);
55     printf("\nEnter number of connection_released: ");
56     scanf("%d",&m);
57     for(i=0;i<n;i++)pthread_create(&acquire[i],NULL,connection_acquired,NULL);
58     for(i=0;i<m;i++)
59         pthread_create(&release[i],NULL,connection_released,NULL);
60     for(i=0;i<n;i++)
61         pthread_join(acquire[i],NULL);
62     for(i=0;i<m;i++)
63         pthread_join(release[i],NULL);
64     exit(0);
65 }
```

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) {
                    f[i] = 1;
                    ind = i;
                }
            }
        }
    }
}
```

```

        }
    }
    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:

```

1  #include <stdio.h>
2  int main()
3  {
4      int n, m, i, j, k;
5      n = 5;
6      m = 3;
7      int alloc[5][3] = { { 0, 1, 0 }, // P0    // Allocation Matrix
8                          { 2, 0, 0 }, // P1
9                          { 3, 0, 2 }, // P2
10                         { 2, 1, 1 }, // P3
11                         { 0, 0, 2 } }; // P4
12      int max[5][3] = { { 7, 5, 3 }, // P0    // MAX Matrix
13                       { 3, 2, 2 }, // P1
14                       { 9, 0, 2 }, // P2
15                       { 2, 2, 2 }, // P3
16                       { 4, 3, 3 } }; // P4
17      int avail[3] = { 3, 3, 2 };
18      int f[n], ans[n], ind = 0;
19      for (k = 0; k < n; k++) {
20          f[k] = 0;
21      }
22      int need[n][m];
23      for (i = 0; i < n; i++) {
24          for (j = 0; j < m; j++)
25              need[i][j] = max[i][j] - alloc[i][j];
26      }
27      int y = 0;
28      for (k = 0; k < 5; k++) {
29          for (i = 0; i < n; i++) {
30              if (f[i] == 0) {
31
32                  int flag = 0;
33                  for (j = 0; j < m; j++) {
34                      if (need[i][j] > avail[j]){
35                          flag = 1;
36                          break;
37                      }
38                  }
39                  if (flag == 0) {
40                      ans[ind++] = i;
41                      for (y = 0; y < m; y++)
42                          avail[y] += alloc[i][y];
43                      f[i] = 1;
44                  }
45              }
46          }
47      }
48      printf("Following is the SAFE Sequence\n");
49      for (i = 0; i < n - 1; i++)
50          printf(" P%d ->", ans[i]);
51      printf(" P%d", ans[n - 1]);
52      return (0);
53 }

```

Following is the SAFE Sequence

P1 -> P3 -> P4 -> P0 -> P2

...Program finished with exit code 0

Press ENTER to exit console