Laboratory 5

Title of the Laboratory Exercise: Solution to Producer Consumer Problem using Semaphore and Mutex

1. Introduction and Purpose of Experiment

In multitasking systems, simultaneous use of critical section by multiple processes leads to data inconsistency and several other concurrency issues. By solving this problem students will be able to use Semaphore and Mutex for synchronisation purpose in concurrent programs. 2. Aim and Objectives

Aim

- To implement producer consumer problem using Semaphore and Mutex

Objectives

At the end of this lab, the student will be able to

- Use semaphore and Mutex
- Apply semaphore and Mutex in the required context
- Develop multithreaded programs with Semaphores and Mutex

3. Experimental Procedure

i. Analyse the problem statement ii. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code iii. Implement the algorithm in C language iv. Compile the C program

v. Test the implemented program vi. Document the Results vii. Analyse and discuss the outcomes of your experiment

4. Questions

Implement producer consumer problem by using the following

a) Semaphore

b) Mutex

5. Calculations/Computations/Algorithms

START:

Step 1: Initialize all the header files.

Step 2: Initialize all the variables and put front =0 and rear=0 (formation of queue)

Step 3: Create a function called produce_item() for getting the random numbers to feed inside the queue.

Step 4: Create a function called insert_item() which will see if rear=size then initialize rear with 0 (empty) and if not then insert the item inside the queue.

Step 5: Create a function called remove_item() which will see the front part of the queue if front = size then assign front = 0 (empty) and if not then insert the item inside the queue.

Step 6: Create a function called consume_item() and print out the value of items consumed till now.

Step 7: Create a pointer function called producer which will do the following tasks:

- Decreasing empty semaphore; represents no: of empty slots
  if the number ofempty slots is 0 (i.e., the semaphore becomeszero), the thread waits (forth consumer to consume an item).
- Locking bin_sem to gain exclusive access to critical region.
- Inserting item: critical region.
- Unlocking bin_sem to release access to critical region
- Increasing full semaphore; represents no: of full slots.

Step 8: Create a pointer function called consumer which will do the following tasks:

- Decreasing full semaphore; represents no: of full slots. If the number of full slots is 0 (i.e., the semaphore becomes

zero), the thread waits (for the producer to produce an item).

- Locking bin_sem to gain exclusive access to critical region
- Removing item: critical region
- Unlocking the bin_sem to release access to critical region.
- Increasing empty semaphore; represents no: of empty slots.

Step 9: The main() function

- Initialize semaphores
- Initialize threads
- Create threads using p_thread.
- Join threads

Step 10: End

6. Presentation of Results

```c
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <time.h>
#include <pthread.h>

#define N 10
sem_t bin_sem, full, empty;
int front = 0;
int rear = 0;
int CQueue[N];

int produce_item()
{
    int p = 1 + rand() % 300;
    return p;
}

void insert_item(int item)
{
    if (rear == N)
        rear = 0;
    CQueue[rear++] = item;
    printf("The Item Inserted is %d\n", item);
}

int remove_item()
{
    if (front == N)
        front = 0;
    int e = CQueue[front++];
    return e;
}

void consume_item(int item)
{
    printf("The Item Consumed is %d\n", item);
}

void *producer(void *a)
{
    int item, i;
```

```
    for (i = 0; i < N; i++)
    {
        item = produce_item();
        // decreasing empty semaphore; represents no: of empty slots
        // if the number of empty slots is 0 (i.e., the semaphore becomes
        // zero), the thread waits (for the consumer to consume an item).
        sem_wait(&empty);
        // locking bin_sem to gain exclusive access to critical region
        sem_wait(&bin_sem);
        // inserting item: critical region
        insert_item(item);
        // unlocking bin_sem to release access to critical region
        sem_post(&bin_sem);
        // increasing full semaphore; represents no: of full slots
        sem_post(&full);
    }
}

void *consumer(void *a)
{
    int item, i;
    for (i = 0; i < N; i++)
    {
        // decreasing full semaphore; represents no: of full slots
        // if the number of full slots is 0 (i.e., the semaphore becomes)
        // zero), the thread waits (for the producer to produce an item)
        sem_wait(&full);
        // locking bin_sem to gain exclusive access to critical region
        sem_wait(&bin_sem);
        // removing item: critical region
        item = remove_item();
        consume_item(item);
        // unlocking the bin_sem to release access to critical region
        sem_post(&bin_sem);
        // increasing empty semaphore; represents no: of empty slots
        sem_post(&empty);
    }
}

int main(int argc, char **argv)
{
    // initializing semaphores
    sem_init(&bin_sem, 0, 1);
    sem_init(&full, 0, 0);
    sem_init(&empty, 0, N);

    //initializing
    pthread_t th1, th2;
    int a;

    // creating threads
    pthread_create(&th1, NULL, producer, NULL);
    pthread_create(&th2, NULL, consumer, NULL);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    return (EXIT_SUCCESS);
}
```

Figure 1: Using semaphore

```
/mnt/d/RUAS-sem-05/OS/lab5 main*
❯ gcc sem.c -pthread

/mnt/d/RUAS-sem-05/OS/lab5 main*
❯ ./a.out
The Item Inserted is 284
The Item Inserted is 287
The Item Inserted is 178
The Item Inserted is 116
The Item Inserted is 294
The Item Inserted is 236
The Item Inserted is 287
The Item Inserted is 193
The Item Inserted is 250
The Item Inserted is 122
The Item Consumed is 284
The Item Consumed is 287
The Item Consumed is 178
The Item Consumed is 116
The Item Consumed is 294
The Item Consumed is 236
The Item Consumed is 287
The Item Consumed is 193
The Item Consumed is 250
The Item Consumed is 122

/mnt/d/RUAS-sem-05/OS/lab5 main*
❯
```

Figure 2: Execution

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>
#include <sys/unistd.h>
#define N 10

// defining mutexes
pthread_mutex_t mutex;
pthread_cond_t condc, condp;

// defining queue vars
int front = 0;
int rear = 0;
int CQueue[N];

int current_buffer_size = 0;
int p, sum = 0;

int produce_item()
{
    // srand(time(NULL));
    p = 1 + rand() % 40;
    return p;
}

void insert_item(int item)
{
```

```
    if (rear == N)
        rear = 0;
    CQueue[rear++] = item;
    current_buffer_size++;
    printf("The Item Inserted is %d\n", item);
}
int remove_item(int item)
{
    if (front == N)
        front = 0;
    int e = CQueue[front++];
    current_buffer_size--;
    printf("%d was removed !\n\n", item);
    return e;
}
void consume_item(int item)
{
    sum += item;
    printf("The Item %d is Consumed\n", item);
}
void *producer(void *a)
{
    int i, item;
    for (i = 0; i < N; i++)
    {
        // locks the critical region
        pthread_mutex_lock(&mutex);

        item = produce_item();

        // if buffer is full, wait and unlock mutex
        // while loop used to avoid spurious wakeups
        while (current_buffer_size == N)
            pthread_cond_wait(&condp, &mutex);

        // insert item
        insert_item(item);
        // wake up consumer
        pthread_cond_signal(&condc);
        // unlock critical region
        pthread_mutex_unlock(&mutex);
    }
}
void *consumer(void *a)
{
    int i, item;
    for (i = 0; i < N; i++)
    {
        // locks the cirtical region
        pthread_mutex_lock(&mutex);
        // if buffer is empty
        // while loop used to avoid spurious wakeups
        while (current_buffer_size == 0)
            pthread_cond_wait(&condc, &mutex);
        // remove item
        item = remove_item(item);
        // consume item
        consume_item(item);
        // wake up producer if sleeping
        pthread_cond_signal(&condp);
        // unlock critical region
        pthread_mutex_unlock(&mutex);
    }
}
int main(int argc, char **argv)
{
    // creating muxtes
    pthread_mutex_init(&mutex, 0);
    pthread_cond_init(&condp, 0);
    pthread_cond_init(&condc, 0);
```

```c
    // creating threads
    pthread_t th1, th2;
    int a;
    // spawing threads
    pthread_create(&th1, NULL, producer, NULL);
    pthread_create(&th2, NULL, consumer, NULL);
    // joining threads
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    // destroying mutexes
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&mutex);
}
```

Figure 3: Using mutex

```
/mnt/d/RUAS-sem-05/OS/lab5 main*
❯ gcc mutex.c -pthread

/mnt/d/RUAS-sem-05/OS/lab5 main*
❯ ./a.out
The Item Inserted is 24
The Item Inserted is 7
The Item Inserted is 18
The Item Inserted is 36
The Item Inserted is 34
The Item Inserted is 16
The Item Inserted is 27
The Item Inserted is 13
The Item Inserted is 10
The Item Inserted is 22
0 was removed !

The Item 24 is Consumed
24 was removed !

The Item 7 is Consumed
7 was removed !

The Item 18 is Consumed
18 was removed !

The Item 36 is Consumed
36 was removed !

The Item 34 is Consumed
34 was removed !

The Item 16 is Consumed
16 was removed !

The Item 27 is Consumed
27 was removed !

The Item 13 is Consumed
13 was removed !

The Item 10 is Consumed
10 was removed !

The Item 22 is Consumed

/mnt/d/RUAS-sem-05/OS/lab5 main*
❯
```

Figure 4: Execution

7. Analysis and Discussions:

A mutex is a shared rejection system while semaphore is a flagging instrument. We use mutex where we need to permit a bit of code known as critical area which is accessed by different individuals to get to the memory, to be executed by each string in turn. We use semaphore to flag/tell about some occasion like for instance if the buffer is empty then semaphore can be utilized to flag the producer to start producing.

In this lab the current issue is that there are two entities the producer and the consumer as the name propose the job of producer is to create information in the buffer and the activity of the consumer is to consume the information delivered by the producer. The buffer utilized here is round structure (practically like roundabout line) there are sure things that we have to take care about is the point at which the information in the buffer gets empty, then the signal needs to be sent to the producer to begin delivering and the consumer to quit consuming till the information is created likewise if the block gets full the consumers begin consuming and the producers are signalled to end the production. So also, the block where the creating is produced the information presently the producer can't expend from a similar block at the end of the day producer and consumer can't be point to same block in the buffer at an occurrence.

8. Conclusions

The producer-consumer problem is a synchronization issue. There is a fixed size buffer and the producer produces things and enters them into the buffer. The customer eliminates the things from the buffer and consumes them.

A producer does not create things into the buffer when the consumer is consuming a thing from the buffer and the other way around. So, the buffer should just be gotten to by the producer or consumer at once. The producer and consumer issue can be settled utilizing semaphores and mutex.

9. Comments

1. Limitations of Experiments

The code gets unstable while impersonating the producer and consumer issue for enormous scale information either bigger number of producer or consumer for that the framework that we are executing the code on solidifies because of huge resource for running the code.

2. Limitations of Results

The code continues running until the user interfere with the execution yet the code can be adjusted such that it requests that user to enter the time/cycle it should run before it comes to end so user can pre decide the kind of output he/she needs as to interrupt on the program randomly.

3. Learning happened

We learned the working of producer consumer problem and its working also we learn how to code the following using c programming language and from doing that in-depth knowledge of the topic is gained

4. Recommendations

Utmost care has been taken while creating the document and the program in the given time constraint but there is always a chance of improvement and occurrence of error.