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

Using Semaphore:

producer():
1. sem_wait(&empty)
2. sem_wait(&mutex)
3. item = produce_item()
4. data.add(item)
5. sem_post(&mutex)
6. sem_post(&full)

consumer():
1. sem_wait(&full)
2. sem_wait(&mutex)
3. item = data.back()
4. consume_item(item)
5. data.pop()
6. sem_post(&mutex)
7. sem_post(&empty)

Using Mutex:

producer():
1. mutex_lock(&mutex)
2. if (buffer.is_full)
3. cond_wait(&p_cond, &mutex)
4. item = produce_item()
5. data.add(item)
6. mutex_unlock(&mutex)
6. cond_signal(&c_cond) /* signal the consumer */

consumer():

1. mutex_lock(&mutex)
2. if (buffer.is_empty)
3. cond_wait(&c_cond, &mutex)
3. item = data.back()
4. consume_item(item)
5. data.pop()
6. mutex_unlock(&mutex)
7. cond_signal(&p_cond) /* signal the producer */

6. Presentation of Results

Using Semaphore:

```
#include <iostream>
#include <limits>
#include <memory>
#include <random>
#include <vector>

#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define MAX_SIZE 5

struct buffer_t {
   std::vector<int> data;
   sem_t empty;
   sem_t full;
   sem_t mutex;
   int N;

   buffer_t(const int N);
} buffer(10);

buffer_t::buffer_t(const int N = 10) : N(N) {
   sem_init(&(this->empty), 0, N);
   sem_init(&(this->full), 0, 0);
   sem_init(&(this->mutex), 0, 1);
}

std::random_device rd;
std::mt19937 mt(rd());
std::uniform_int_distribution<int> dist(0, std::numeric_limits<int>::max());

int produce_item() {
   const int item = dist(mt);
   printf("Produced : %d\n", item);
```

```
    return item;
}

void consume_item(const int& item) {
    printf("Consumed : %d\n", item);
}

void* producer(void* args) {
    for (int i = 0; i < buffer.N; i++) {
        sem_wait(&buffer.empty);
        sem_wait(&buffer.mutex);
        int item = produce_item();
        buffer.data.push_back(item);
        sem_post(&buffer.mutex);
        sem_post(&buffer.full);
    }

    pthread_exit(NULL);
}

void* consumer(void* args) {
    for (int i = 0; i < buffer.N; i++) {
        sem_wait(&buffer.full);
        sem_wait(&buffer.mutex);
        int item = buffer.data.back();
        consume_item(item);
        buffer.data.pop_back();
        sem_post(&buffer.mutex);
        sem_post(&buffer.empty);
    }

    pthread_exit(NULL);
}

int main(int argc, char* argv[]) {
    pthread_t t_producer, t_consumer;

    pthread_create(&t_producer, NULL, producer, NULL);
    pthread_create(&t_consumer, NULL, consumer, NULL);

    pthread_join(t_producer, NULL);
    pthread_join(t_consumer, NULL);
}
```

Using Mutex:

```
#include <iostream>
#include <limits>
#include <memory>
#include <random>
#include <vector>

#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define MAX_SIZE 5

struct buffer_t {
   std::vector<int> data;
   int buf_idx;
   int N;

   pthread_mutex_t mutex;
   pthread_cond_t c_cond, p_cond;

   bool is_empty = false;
   bool is_full = false;

   buffer_t(const int N);
} buffer(10);

buffer_t::buffer_t(const int N = 10) : N(N) {
   c_cond = PTHREAD_COND_INITIALIZER;
   p_cond = PTHREAD_COND_INITIALIZER;
   mutex = PTHREAD_MUTEX_INITIALIZER;
}

std::random_device rd;
std::mt19937 mt(rd());
std::uniform_int_distribution<int> dist(0, std::numeric_limits<int>::max());

int produce_item() {
   const int item = dist(mt);
   printf("Produced : %d\n", item);
   return item;
}

void consume_item(const int& item) {
   printf("Consumed : %d\n", item);
}

void* producer(void* args) {
   for (int i = 0; i < buffer.N; i++) {
```

```
        pthread_mutex_lock(&buffer.mutex);
        buffer.is_full = buffer.data.size() == buffer.N;
        if (buffer.is_full) {
            pthread_cond_wait(&buffer.p_cond, &buffer.mutex); /* wait for the consumer */
        }
        int item = produce_item();
        buffer.data.push_back(item);
        pthread_mutex_unlock(&buffer.mutex);
        pthread_cond_signal(&buffer.c_cond); /* signal the consumer */
    }

    pthread_exit(NULL);
}

void* consumer(void* args) {
    for (int i = 0; i < buffer.N; i++) {
        pthread_mutex_lock(&buffer.mutex);
        buffer.is_empty = buffer.data.size() == 0;
        if (buffer.is_empty) {
            pthread_cond_wait(&buffer.c_cond, &buffer.mutex); /* wait for the producer */
        }
        int item = buffer.data.back();
        consume_item(item);
        buffer.data.pop_back();
        pthread_mutex_unlock(&buffer.mutex);
        pthread_cond_signal(&buffer.p_cond); /* singal the producer */
    }

    pthread_exit(NULL);
}

int main(int argc, char* argv[]) {
    pthread_t t_producer, t_consumer;

    pthread_create(&t_producer, NULL, producer, NULL);
    pthread_create(&t_consumer, NULL, consumer, NULL);

    pthread_join(t_producer, NULL);
    pthread_join(t_consumer, NULL);
}
```

7. Analysis and Discussions

Using Semaphore:

```
└─ /mnt/d/University-Work/University-Work-SEM-05/OS-Lab/Lab05/using_semaphore/build/Lab05
Produced : 937307065
Produced : 1780076150
Produced : 2094359772
Consumed : 2094359772
Consumed : 1780076150
Consumed : 937307065
Produced : 1421145800
Produced : 8554448
Produced : 1978421493
Produced : 1160795803
Consumed : 1160795803
Consumed : 1978421493
Consumed : 8554448
Produced : 421413859
Produced : 1348636457
Produced : 692632324
Consumed : 692632324
Consumed : 1348636457
Consumed : 421413859
Consumed : 1421145800
```

Figure 0-1 Producer-Consumer using Semaphore

Using Mutex:

```
└─ /mnt/d/University-Work/University-Work-SEM-05/OS-Lab/Lab05/using_mutex/build/Lab05
Produced : 1187389707
Produced : 1267207385
Produced : 584435806
Produced : 1181495914
Produced : 1651519673
Produced : 1369313700
Produced : 1769653850
Produced : 253456831
Produced : 421923963
Produced : 452391991
Consumed : 452391991
Consumed : 421923963
Consumed : 253456831
Consumed : 1769653850
Consumed : 1369313700
Consumed : 1651519673
Consumed : 1181495914
Consumed : 584435806
Consumed : 1267207385
Consumed : 1187389707
```

Figure 0-2 Producer-Consumer using Mutex

8. Conclusions

The Producer-Consumer problem can be solved using Semaphores and Mutexes which are present in the pthread library in C.

Mutex

Mutex is a mutual exclusion object that synchronizes access to a resource. It is created with a unique name at the start of a program. The Mutex is a locking mechanism that makes sure only one thread can acquire the Mutex at a time and enter the critical section. This thread only releases the Mutex when it exits the critical section.

This is shown with the help of the following example:

```
wait (mutex);
    ...
Critical Section
    ...
signal (mutex);
```

A Mutex is different than a semaphore as it is a locking mechanism while a semaphore is a signalling mechanism. A binary semaphore can be used as a Mutex but a Mutex can never be used as a semaphore.

Semaphore

A semaphore is a signalling mechanism and a thread that is waiting on a semaphore can be signalled by another thread. This is different than a mutex as the mutex can be signalled only by the thread that called the wait function.

A semaphore uses two atomic operations, wait and signal for process synchronization.

The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.

```
wait(S) {
   while (S<=0);
    S--;
}
```

The signal operation increments the value of its argument S.

```
signal(S) {
   S++;
}
```

There are mainly two types of semaphores i.e. counting semaphores and binary semaphores.

Counting Semaphores are integer value semaphores and have an unrestricted value domain. These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources.

The binary semaphores are like counting semaphores but their value is restricted to 0 and 1. The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0.

9. Comments

1. Limitations of Experiments

Semaphores involve a queue in its implementation. For a FIFO queue, there is a high probability for a priority inversion to take place wherein a high priority process which came a bit later might just have to wait when a low priority one is in the critical section.

2. Limitations of Results

To test the semaphores and locks, the number of items is very small, due to this the testing is not accurate, there could be a wrong implementation and the program might still work if the thread did not get de-scheduled, hence the simulation should be done for a relatively larger number of items.

3. Learning happened

We learnt the use of semaphores and mutex to solve the producer-consumer problem in context of multi-threading.

4. Recommendations

Use an arbitrary length buffer and run the program for a relatively larger number of items and compare the performance trade-offs between the two methods.