Name: JUPALLY SAI NISHITH
Date: 15-01-2020
Signature: JSN

Producer-Consumer Problem

(With and Without Semaphores)

Producer-Consumer is classic synchronization problem due to race conditions in the critical section of Producer(who makes an attempt to write something to the buffer) and Consumer(who tries to retrieve the data from buffer). Due to synchronization issues, there is a scope for lot of errors. This report explains how usage of semaphores will solve this issue of synchronization and race conditions between producer and consumer.

Without Semaphores:

When we don’t use semaphores or any other locks to solve the producer-consume, things can go wrong in a lot of ways.
I have used some of the code snippets in this report as well for easier interpretation of the report.

PRODUCER:

```
while(produced < 10*N)
   {
         while(count == N)
                  ;
         buffer[in] = syscall(SYS_gettid);
         printf("Produced %d \n", buffer[in]);
         in = (in + 1)%(N);
         count++;
         produced++;

   }
```

As per above snippet of code, producer(each of p threads is a separate producer) does busy waiting when the count i.e the total number of elements in buffer currently is equal to the buffer size or else it writes its thread id into the buffer and increments the current index and count by one. Also it increments the total number of items produced by the thread by 1 and the thread stops producing if it produces 10*N (as per question and it is the exit condition for the while loop).

CONSUMER:

```
void *consumer(int buffer[])
{
   while(tconsumer < ((10*N)*p))
   {
         while(count == 0)
                  ;
         int consumed = buffer[out];
         tconsumer++;
         printf("Consumed  %d \n", consumed);
         out = (out + 1)%N;
         count--;
  }
}
```

As per above snippet of code,consumer(each of n thread is a seperate consumer) does busy waiting when count i.e the number of elements in the buffer currently is equal to zero(as there is nothing to consume) or else it consumes the data from buffer and increments the tconsumer(total number of items consumed by all the consumers) and out index by 1 and decreases by total number of elements by 1. Also the exit condition for the consumer is tconsumer < ((10*N)*p) because each producer can produce maximum 10*N things and there are p of them and consumer should leave after consuming all of these.

ERROR:

The solution might seem fine but isn't because the operation count++ isn't atomic. Suppose count has a value of 3(say). Now if producer is executed then value of count should become 4 and when consumer is executed then count should become 3 again.

But consider a case where context switch happens when value of 3 is stored in register for performing count++ and control is taken by consumer who then stores the value 3 itself in the register and n0w if there is a switch to producer, the value of count will 4 and then in the consumer count-- would be 2(not 3 as value 3 was stored in register). So we get incorrect value for count as both processes are manipulating this shared variable concurrently.

For the code i've written, there is a possibility that multiple threads enter the produce function at instant of time and also for the consumers.We can avoid this by any ticket based solution for consumers and producer but we still face the issue of atomic operation for increment, so we'll have to use some lock like the semaphores for perfect solution.

With Semaphores:

TO RESOLVE THIS ISSUE (non atomic operation of increment),WE USE SEMAPHORES WHICH ARE NON-NEGATIVE INTEGER WITH ATOMIC INCREMENT AND DECREMENT.

Semaphore S can be modified only by :
S.wait()- decrement or block if already 0
S.signal()-increment and wake up process if any

```
sem_t mutex,empty,full;
sem_init(&mutex,0,1);
sem_init(&full,0,0);
 sem_init(&empty,0,n);
```

```
void producerf(int array[])
{
   int totalProduced = 0;
   while(totalProduced < (10*n))
   {
         sem_wait(&empty);
         sem_wait(&mutex);
         array[head%n] = syscall(SYS_gettid);
         printf("Produced by %d\n",array[head%n]);
         head++;
         tp++;
         totalProduced++;
         sem_post(&mutex);
         sem_post(&full);
   }
}
```

Three semaphores mutex(for mutual exclusion),full(to check if the buffer is full),empty(to check is empty) are initialised

As per above snippet of code, producer(each of p threads is a separate producer) does busy waiting when the when there no more space in the buffer(because of semaphore empty) ,else it will proceed to next line code and check if there is any other producer in the critical section with the help of mutex semaphore. If there is no other thread inside critical section then the further statements execute.Then after executing the further statement, signal semaphore is released 1st one for mutex and then for full.

Similarly for consumer, we first have full semaphore and it blocks the thread if the buffer is empty and then we check for mutex lock. If all the locks are open then consumer code executes.

The semaphore full and empty help to check the number of elements in the buffer currently and they ensure that producer doesn't produce when the buffer is full and consumer doesn't consume when there are no elements in the buffer. With the help of mutex semaphore we are making sure that no two threads enter critical section once and ensuring no synchronisation issues. I have also included the print statements in the critical section and at any instant during output the number of print statements from producer is greater than or equal to number of print statements from consumer. Hence we can assure that semaphores are able to overcome the synchronisation issues.

Thus we clearly we that any solution like Peterson’s algorithm, Dekker’s solution,etc don’t work without locks. Semaphores are very easy to use locks with wait and signal function which cater our requirement of atomic increment/decrement algorithm.