Ex. 7A Producer Consumer Problem

Dr S. RAJARAJAN

SASTRA

In the producer-consumer problem, one process (the producer) generates data items and another process (the consumer) receives and uses them. They communicate using a queue of maximum size N and are subject to the following conditions:

- The consumer must wait for the producer to produce something if the queue is empty.
- The producer must wait for the consumer to consume something if the queue is full.
- Both must mutually exclude each other

The semaphore solution to the producer-consumer problem tracks the state of the queue with two semaphores: **`emptyCount`**, the number of empty places in the queue, and **`fullCount`**, the number of elements in the queue. To maintain integrity, **`emptyCount`** may be lower (but never higher) than the actual number of empty places in the queue, and **`fullCount`** may be higher (but never lower) than the actual number of items in the queue. Empty places and items represent two kinds of resources, empty boxes and full boxes, and the semaphores **`emptyCount`** and **`fullCount`** maintain control over these resources.

The producer does the following repeatedly:

```
produce:
     P(emptyCount)
     P(mutex)
     putItemIntoQueue(item)
     V(mutex)
     V(fullCount)
```

The consumer does the following repeatedly:

```
consume:
     P(fullCount)
     P(mutex)
     item ← getItemFromQueue()
     V(mutex)
     V(emptyCount)
```

Example

1. A single consumer enters its critical section. Since ***fullCount*** is 0, the consumer blocks.
2. Several producers enter the producer critical section. No more than N producers may enter their critical section due to ***emptyCount*** constraining their entry.
3. The producers, one at a time, gain access to the queue through ***mutex*** and deposit items in the queue.
4. Once the first producer exits its critical section, ***fullCount*** is incremented, allowing one consumer to enter its critical section.

Note that **emptyCount** may be much lower than the actual number of empty places in the queue, for example in the case where many producers have decremented it but are waiting their turn on `mutex` before filling empty places. Note that **emptyCount + fullCount ≤ N** always holds, with equality if and only if no producers or consumers are executing their critical sections.

Program
Note: To compile the program, type cc –pthread <programname>.c

```
#include <stdio.h>
#include <semaphore.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>

int front = -1;
int rear = -1;
int array[5];
```

```
sem_t mutex;
sem_t emptyCount;
sem_t fullCount;

int p_tid;
int c_tid;

int produce_item();
void insert_item(int);
int remove();

void * produce()
{
 int item;
 while(1)
{
        item=produce_item();
        sem_wait(&emptyCount);
        sem_wait(&mutex);
        printf("\nproducer entering the critical section");
        insert_item(item);
        printf("\n producer %d inserting an item %d at %d",item, rear);
        sem_post(&mutex);
        sem_post(&fullCount);
}
}
void *consumer()
{
         int item;
        while(1)
        {
                 sleep(20);
                sem_wait(&fullCount);
                sem_wait(&mutex);
                printf("\n sonsumer %d entering the critical regiion");
                item=remove_item();
                printf("\n comsumer %d leaving the critical region");
                sem_post(&mutex);
                sem_post(&emptyCount);
                printf("\n consumer %d consumed item %d",item);
        }
}
```

```
int produce_item()
{
        static int a=100;
        return a++;
}
void insert_item(int item)
{
        rear = rear+1;
        rear = rear % 5;
        array[rear] = item;
}
int remove_item()
{
        front = front+1;
        front = front%5;
        int item= array[front];
        return item;
}
int main()
{
        int a[8] = {0,1,2,3,4};
        int i;

        pthread_attr_t *attr = NULL;
        pthread_t p_tid;
        pthread_t c_tid;
        sem_init(&mutex,0,1);
        sem_init(&emptyCount,0,5);
        sem_init(&fullCount,0,0);
        pthread_create(&p_tid,attr,produce,0);
        pthread_create(&c_tid,attr,consumer,0);
        pthread_join(p_tid,NULL);
        pthread_join(c_tid,NULL);
        return 0;
}
```