Brandon J Wong

[bjwong@sfu.ca](mailto:bjwong@sfu.ca)

301173884

**Producer-Consumer Problem and POSIX**

The producer-consumer problem is a problem that highlights issues that can occur in a system consisting of more than one process, which share a common set of resources. In the producer-consumer problem, the two types of processes that are present are the consumer and the producer, all of which share a common fixed sized buffer. The producer(s) produce data and puts it into the buffer, while the consumer(s) consume data.

Let’s consider the solution to this problem in terms of the usage of POSIX threads.

When implementing this scenario via multiprogramming, a producer thread must not attempt to add data when the buffer is full, and a consumer thread must not try to consume data when the buffer is empty. Because the producer(s) and the consumer(s) are acting on what can effectively be viewed as their own timelines, the solution must employ inter-process communication in order to ensure that the producer(s) and the consumer(s) do not enter their critical sections at the same time, (that is, that no two processes attempt to alter a shared set of resources at the same time (which would cause the data to be corrupted)), by way of mutual exclusion.

There are two main sub-problems that need to be addressed. The first of which is the most obvious: ensuring that producers/consumers cannot add/consume data when the buffer is full/empty. This can be solved by way of two semaphores, *pro* and *con*. *Pro* can be used to track how many more times a producer can add to the buffer. When *pro* is set to 0, this means that there is no more space for an additional unit of data, and so the producer(s) are put to sleep. When *pro* is not 0, a producer adds some data, decrements *pro*, increments *con*, and wakes up any sleeping consumers. *Con* can be used to track how many times a consumer can consume a unit of data. When *con* is set to 0, this means that the buffer is empty, and so the consumer sleeps. When *con* is not 0, the consumer consumes some data, decrements *con*, increments *pro*, and wakes any sleeping producers. For the first problem, the blocking done here would only be unbounded if at some time, both *pro* and *con* can be set to 0. This would occur at a time in which the buffer is both full and empty, which, if we assume the implementation has been done correctly, is impossible. Thus, because both *pro* and *con* will never both be set to 0, unbounded blocking is avoided.

The second sub-problem is that of preventing producers and consumers from reading or writing shared data simultaneously. This can be done by enlisting the help of a mutex, which encapsulates the operations corresponding to the critical section. This results in the blocking of other threads during the time in which one thread is in its critical section, and the waking of those threads once that thread leaves its critical section. If this type of blocking was unbounded, then it must also be true that there is a possibility for the mutex to be denying *all* threads access to their critical sections, while no thread is in its critical section, or that there is a possibility for a thread to stay within its critical section forever. We can eliminate the first possibility, as a mutex only blocks other threads when there exists a thread is in its critical section. The second possibility can be dismissed as well as the executions of the critical sections of these threads have no reliance on the states of anything else (other than the emptiness/fullness of the buffer, which the semaphores have dealt with) and because there are no infinite loops within our critical sections.

The usage of circular buffers in this scenario would be ideal, as a FIFO data structure would probably be preferable to that of a LIFO data structure. As producers add data, it probably makes more sense for the oldest data to be consumed first, opposed to the newest, so that our buffer is always characterized by relatively newer data. If we use a LIFO data structure, then the data at the front of the buffer is likely to be considerably more aged than that of the data at the end. Also, if for some reason we want to incorporate overwriting to be a part of our buffer’s design, then overwriting the oldest data first probably makes the most sense, which a circular buffer allows us to achieve.

Below is a specific coding example of all we have discussed using POSIX threads and semaphores in C.

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#define MAX 10000000000

#define EMPTY 0

#define FULL 1

pthread\_mutex\_t the\_mutex;

sem\_t pro, con;

int inputBuffer[MAX\_BUFFER\_SIZE];

int bFront;

int bCount;

void produce();

void consume();

void\* producer(void \*ptr) {

int i;

for (i = 1; i < MAX; i++) {

sem\_wait(&pro);

pthread\_mutex\_lock(&the\_mutex);

produce();

pthread\_mutex\_unlock(&the\_mutex);

sem\_post(&con);

}

pthread\_exit(0);

}

void\* consumer(void \*ptr) {

int i;

for (i = 1; i < MAX; i++) {

sem\_wait(&con);

pthread\_mutex\_lock(&the\_mutex);

consume();

pthread\_mutex\_unlock(&the\_mutex);

sem\_post(&pro);

}

pthread\_exit(0);

}

void produce() {

int bBack = (bFront + bCount) % MAX\_BUFFER\_SIZE;

inputBuffer[bBack] = FULL;

bCount++;

}

void consume() {

inputBuffer[bFront] = EMPTY;

bFront = (bFront + 1) % MAX\_BUFFER\_SIZE;

bCount--;

}

int main (int argc, char \*\*argc) {

pthread\_t pro[NUM\_PRODUCERS], con[NUM\_CONSUMERS];

// Initialize the mutex and semaphore variables

pthread\_mutex\_init(&the\_mutex, NULL);

sem\_init(&pro, 0, MAX\_BUFFER\_SIZE);

sem\_init(&con, 0, 0);

bFront = 0;

bCount = 0;

// Create the threads

for (i=0; i<NUM\_PRODUCERS; i++) {

pthread\_create(&pro[i], NULL, producer, NULL);

}

for (i=0; i<NUM\_CONSUMERS; i++) {

pthread\_create(&con[i], NULL, consumer, NULL);

}

// Wait for the threads to finish

for (i=0; i<NUM\_GENERATORS; i++) {

pthread\_join(gen[i], NULL);

}

for (i=0; i<NUM\_OPERATORS; i++) {

pthread\_join(oper[i], NULL);

}

// Cleanup

pthread\_mutex\_destroy(&the\_mutex);

sem\_destroy(&pro);

sem\_destroy(&con);

}