CSC 360/660, Operating Systems

Project #4: Process Synchronization Using Pthreads: The Producer / Consumer Problem With Prime Number Detector

---

OBJECTIVE

The purpose of this programming project is to explore process synchronization. This will be accomplished by writing a simulation to the Producer / Consumer problem described below. Your simulation will be implemented using Pthreads. Tutorials on the Pthread functions and their usage can be found in our text, in our notes, or online. This simulation is a modification to the programming project found at the end of Chapter 6 and most of the text of this project specification is directly from that project description. Name the program osproj4.cpp or osproj4.c and the executable as osproj4.

THE PRODUCER / CONSUMER PROBLEM

In Chapter 3, we developed a model of a system consisting of cooperating sequential processes or threads, all running asynchronously and possibly sharing data. We illustrated this model with the producer - consumer problem, which is representative of operating systems. Specifically, we discussed how a "bounded buffer" could be used to enable processes to share memory.

In Chapter 3, we described a technique using a circular buffer that can hold BUFFER_SIZE-1 items. By using a shared memory location count, the buffer can hold all BUFFER_SIZE items. This count is initialized to 0 and is incremented every time an item is placed into the buffer and decremented every time an item is removed from the buffer. The count data item can also be implemented as a counting semaphore.

The producer can place items into the buffer only if the buffer has a free memory location to store the item. The producer cannot add items to a full buffer. The consumer can remove items from the buffer if the buffer is not empty. The consumer must wait to consume items if the buffer is empty.

The "items" stored in this buffer will be integers. Your producer process will have to insert random numbers into the buffer. The consumer process will consume a number and detect if the number is prime.

PROJECT SPECIFICATIONS

The buffer used between producer and consumer processes will consist of a fixed-size array of type buffer_item. The queue of buffer_item objects will be manipulated using a circular array. The buffer will be manipulated with two functions, buffer_insert_item() and buffer_remove_item(), which are called by the producer and consumer threads, respectively. A skeleton outlining these function can be found in buffer.h.

The buffer_insert_item() and buffer_remove_item() functions will synchronize the producer and consumer using the algorithms similar to those in Slide 34 and 35 of ch5_v01 Process Sync.pptx. The buffer will also require an initialization function (not supplied in buffer.h) that initializes the mutual exclusion object "mutex" along with the "empty" and "full" semaphores.

The producer thread will alternate between sleeping for a random period of time and generating and inserting (trying to) an integer into the buffer. Random numbers will be generated using the rand_r() function. The sleep function used must be a "thread safe" sleep function. See the code listed below for an overview of the producer algorithm.

The consumer thread will alternate between sleeping for a random period of time (thread safe of course) and (trying to) removing a number out of the buffer. The number removed will then be verified if it is prime. See the code listed below for an overview of the consumer algorithm.

```
#include <stdlib.h>  /* required for rand() */
#include "buffer.h"
void *producer(void *param)
{
  buffer_item rand;
  while(1)
  {
    /* sleep for a random period of time */
    sleep(...);
    /* generate a random number */
    rand = rand();
    printf("producer produced %f\n", rand);
    if(insert_item(rand))
      /* report error condition */
  }
}
void *consumer(void *param)
{
  buffer_item rand;
  while(1)
  {
    /* sleep for a random period of time */
    sleep(...);
    printf("producer produced %f\n", rand);
    if(remove_item(&rand))
      /* report error condition */
    else
      printf("consumer consumed %d \n", rand);
  }
}
```

The main function will initialize the buffer and create the separate producer and consumer threads. Once it has created the producer and consumer threads, the main() function will sleep (thread safe) for duration of the simulation. Upon awakening, the main thread will signal other threads to quit by setting a simulation flag which is a global variable. The main thread will join with the other threads and then display the simulation statistics. The main() function will be passed five parameters on the command line:

- The length of time the main thread is to sleep before terminating (simulation length in seconds)
- The maximum length of time the producer and consumer threads will sleep prior to producing or consuming a buffer_item
- The number of producer threads
- The number of consumer threads
- A "yes" or "no" to output the individual buffer snapshots for each item produced and consumed

A skeleton for the main function appears as:

```
#include "buffer.h"
int main( int argc, char *argv[] )
{
  Get command line arguments
  Initialize buffer
  Create producer thread(s)
  Create consumer thread(s)
  Sleep
  Join Threads
  Display Statistics
```

```
  Exit
}
```

Creating Pthreads using the Pthreads API is discussed in Chapter 4 and in supplemental notes provided online. Please refer to those references for specific instructions regarding creation of the producer and consumer Pthreads.

The following code sample illustrates how mutex locks available in the Pthread API can be used to protect a critical section:

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create the mutex lock */
pthread_mutex_init( &mutex, NULL );

/* aquire the mutex lock */
pthread_mutex_lock( &mutex );

/*** CRITICAL SECTION ***/

/* release the mutex lock */
pthread_mutex_unlock( &mutex );
```

Pthreads uses the pthread_mutex_t data type for mutex locks. A mutex is created with the pthread_mutex_init() function, with the first parameter being a pointer to the mutex. By passing NULL as a second parameter, we initialize the mutex to its default attributes. The mutex is acquired and released with the pthread_mutex_lock() and pthread_mutex_unlock() functions. If the mutex lock is unavailable when pthread_mutex_lock() is invoked, the calling thread is blocked until the owner invokes pthread_mutex_unlock(). All mutex functions return a value of 0 with correct operaton; if an error occurs, these functions return a nonzero error code.

Pthreads provides two types of semaphores: named and unnamed. For this project, we will use unnamed semaphores. The code below illustrates how a semaphore is created:

```
#include <semaphore.h>
sem_t sem;

/* create the semaphore and initialize it to 5 */
sem_init( &sem, 0, 5 );
```

The sem_init() function creates and initializes a semaphore. This function is passed three parameters: A pointer to the semaphore, a flag indicating the level of sharing, and the semaphore's initial value. In this example, by passing the flag 0, we are indicating that this semaphore can only be shared by threads belonging to the same process that created the semaphore. A nonzero value would allow other processes to access the semaphore as well. In this example, we initialize the semaphore to the value 5.

In attahced slides for Chapter 5 Process Synchronization, we described the classical wait() and signal() semaphore operations. Pthread names the wait() and signal() operations sem_wait() and sem_post(), respectively. The code example below creates a binary semaphore mutex with an initial value 1 and illustrates it use in protecting a critical section:

```
#include <semaphore.h>

sem_t mutex;

/* create the semaphore */
sem_init( &mutex, 0, 1 );

/* acquire the semaphore */
```

```
sem_wait( &mutex );

/*** CRITICAL SECTION ***/

/* release the semaphore */
sem_post( &mutex );
```

PROGRAM OUTPUT

Output for this simulation is critical to verify that your simulation program is working correctly. Use this sample as to determine what your simulation should output when various conditions occur (buffer empty/full, location of next producer/consumer, etc.) Your program output format should be identical to the following:

```
% osproj4 30 3 2 2 yes
Starting Threads...
(buffers occupied: 0)
buffers:  -1   -1   -1   -1   -1
         ---- ---- ---- ---- ----
          WR

Producer 12348 writes 31
(buffers occupied: 1)
buffers:  31   -1   -1   -1   -1
         ---- ---- ---- ---- ----
           R   W

Producer 12349 writes 4
(buffers occupied: 2)
buffers:  31    4   -1   -1   -1
         ---- ---- ---- ---- ----
           R        W

Consumer 12350 reads 31   * * * PRIME * * *
(buffers occupied: 1)
buffers:  31    4   -1   -1   -1
         ---- ---- ---- ---- ----
                R   W

...SOME TIME GOES BY...

Consumer 12350 reads 4
(buffers occupied: 0)
buffers:   3   4   19   31   97
         ---- ---- ---- ---- ----
                    WR

All buffers empty.  Consumer 12351 waits.

All buffers empty.  Consumer 12350 waits.

...SOME TIME GOES BY...

Producer 12348 writes 41
(buffers occupied: 5)
buffers:   28   41   23   45   6
         ---- ---- ---- ---- ----
                    RW

All buffers full.  Producer 12349 waits.

...SOME TIME GOES BY...
```

```
Producer 12349 writes 10
(buffers occupied: 1)
buffers:  11   10   18   68   94
         ---- ---- ---- ---- ----
                R    W

Consumer 12350 reads 10
(buffers occupied: 0)
buffers:  11   10   18   68   94
         ---- ---- ---- ---- ----
                   WR

...SOME TIME GOES BY...

PRODUCER / CONSUMER SIMULATION COMPLETE
=======================================
Simulation Time:                       30
Maximum Thread Sleep Time:             3
Number of Producer Threads:            2
Number of Consumer Threads:            2
Size of Buffer:                        5

Total Number of Items Produced:        50
  Thread 1:                            30
  Thread 2:                            20

Total Number of Items Consumed:        48
  Thread 3:                            22
  Thread 4:                            26

Number Of Items Remaining in Buffer:   2
Number Of Times Buffer Was Full:       3
Number Of Times Buffer Was Empty:      4
```

ASSESSMENT AND GRADING

This is an individual assignment. Your program must be written using C or C++ and you are required to use the Pthread with mutex and semaphore libraries. Comment and document all code submitted! Your project must follow the documentation standards defined here. Use good programming practices by implementing procedures and functions where necessary. You may use the STL in your solution. This project is worth 100 points.

PROJECT SUBMISSION

Please submit your source code on Blackboard.