**Operating Systems**

**Project #3:  Producer-Consumer Problem**

**DUE: March 28, 2011 by 09:59:59**

**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.

**THE PRODUCER / CONSUMER PROBLEM**

The producer consumer problem is an operating systems problem that demonstrates cooperating threads. Producer threads will place an item into a common shared buffer. Consumer threads will remove an item from the buffer. In this project the buffer is “bounded”; i.e. it has a finite length to it. This could be implemented as a circular array or you could use one of the STL linear data structures (list, vector, or queue).

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 prime numbers.  Your producer process will have to "randomly find" a prime number (described below).  The consumer process will have to "verify" that the number consumed from the buffer is a prime number.

**PROJECT SPECIFICATIONS**

Write a program that simulates the producer/consumer problem using C/C++ and pthreads. The producer thread will alternate between finding a prime number and inserting it into the buffer.  Prime numbers will be "randomly guessed".  That is the producer will use a random number generator to generate a prime number candidate.  Next the producer will test if the candidate is a prime number.  If the number is a prime, the producer will (try to) insert it into the buffer.

The consumer thread will alternate between (trying to) removing a number out of the buffer and verifying if it is prime.

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 wait for the producers to finish producing and the consumers to finish consuming. After joining all threads, the main thread will then display the simulation statistics. The main() function will be passed four parameters on the command line:

* The number of items to produce per producer thread (-n)
* The length of the buffer (-l)
* The number of consumer threads (-c)
* The number of producer threads (-p)

Perform any and all necessary validation on the command line arguments. A skeleton for the main function appears as:

int main( int argc, char \*argv[] )

{

Get command line arguments

Initialize buffer

Create producer thread(s)

Create consumer thread(s)

Join Threads

Display Statistics

Exit

}

**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, etc.)  To run your program with a 10 items per producer, buffer length of 5, 2 producer threads, and 2 consumer threads, also showing the debug statements, run your program as follows. Your program output format should be (nearly) identical to the following:

prompt$ osproj3 –n 10 –l 5 –p 2 -c 2

Starting Threads...

(PRODUCER 1 writes 1/10 37): (1): [ 37   ]

(PRODUCER 2 writes 1/10 17): (2): [ 37  17  ]

(CONSUMER 2 reads 1 37): (1): [ 17 ]

...SOME TIME GOES BY...

(CONSUMER 1 reads 12 97): (0): [ ] \*EMPTY\*

...SOME TIME GOES BY...

(PRODUCER 2 writes 9/10 17): (5): [ 37   17   61  67   5 ] \*FULL\*

...SOME TIME GOES BY...

PRODUCER / CONSUMER SIMULATION COMPLETE

=======================================

Number of Items Per Producer Thread: 10

Size of Buffer: 5

Number of Producer Threads: 2

Number of Consumer Threads: 2

Total Number of Items Consumed: 20

Thread 1: 12

Thread 2: 8

etc...

Total Number Of Times Buffer Was Full: 3

Total Number Of Times Buffer Was Empty: 4

prompt$

**ADDITIONAL NOTES**

Creating pthreads using the pthreads API is discussed in the notes and in supplemental notes provided online.  Please refer to those references for specific instructions regarding creation of the producer, consumer, and faulty producer 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 witht 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 operation; 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 Section 6.5, 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 );

**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 and semaphore libraries.  Comment and document all code submitted!  Use penguin.tamucc.edu for your program development and testing.  You may do development work on your personal machine but final submissions must compile without errors or warnings and execute without core dumping.  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 DELIVERABLES**

1.  Follow the project submission guidelines documented on our course web page.

2. Your project must have a Makefile and a README file.

2.  Turn in a printed listing of your source code on the due date.

4. Email your .tgz file (without object files or executables) to my email address using the email subject

SUBMIT OS PROJECT 3