ECE 40800 Operating Systems
Project 2: Synchronization

Due Date: Wednesday, April 8, 2015, 11:55pm

Overview

This assignment will familiarize you with the process synchronization primitives **Semaphore**, **Lock**, and **Condition**. You will be using these primitives to solve two classic synchronization problems: **Bounded Buffer**, and **Readers/Writers**. Upon completion you should be more comfortable creating robust multithreaded applications that use shared variables.

Description:

Read two popular synchronization problems: Bounder buffer and reader-writer problems.

Bounded Buffer

The bounded-buffer problem (or, produce – consumer problem) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer) one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

The solution for the producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer. The solution can be reached by means of inter-process communication, typically using semaphores. An inadequate solution could result in a deadlock where both processes are waiting to be awakened. The problem can also be generalized to have multiple producers and consumers.

Readers/Writers

The readers-writers problem is an example of a common computing problem in concurrency. The problem deals with situations in which many threads must access the same shared memory at one time, some reading and some writing, with the natural constraint that no process may access the share for reading or writing while another process is in the act of writing to it. (In particular, it is allowed for two or more readers to access the share at the same time.) A readers-writer lock is a data structure that solves one or more of the readers-writers problems. The solution of the text book and slides gives waiting readers priority over waiting writers. For example, if a writer is waiting for access to the database and a reader is currently reading, another reader can come and start reading without waiting at all. The writer must wait for all readers that arrive and start executing to finish reading before it can start writing. For this MP, we will be changing the priority. We want updates to the database to have priority. So, if there is a writer waiting to write, any readers that arrive must wait for all writers that are waiting to write as well as any writers that arrive in the meantime. As long as there is a reader reading *and no writer is waiting*, or new

reader arrives while another reader is reading, readers get to read the data. But when a reader arrives, and if there is at least one writer waiting to write, all the writers that follow (while the reader is waiting) should go through before the reader.
A good idea would be to implement the algorithm in the book, get it working, and then to update it accordingly to implement the writer priority. Not only will this be easier, but it will help you to understand what the assignment is asking you to do.

Requirements:

- You are required to implement a C/C++ multi-threaded program that uses **pthread** library and avoids deadlock
- You are required to use **pthread synchronization tools** such as semaphores and mutex locks to do synchronization. But, only using mutex locks for synchronization is never a right solution to this problem. Remember: **do right synchronization while maximizing concurrency!**
- You are required to output all of the necessary events into two files named "processevent.log" and "flag.log". For "processevent.log", you need sequentially number (processID) each process and indicates the process's arrival time, the time to start processing (start-time), and the time to exit (end-time). The "processevent.log" looks like:

processID	Resource	arrival-time	start-time	end-time
1	Rq	12:25:11	12:25:12	12:25:13
2	Rl	12:25:13	12:25:13	12:25:14

The "flag.log" tracks when the flag person goes to sleep and when he/she is woken up to work.

Time	State
12:10:11	sleep
12:25:11	woken-up

Hints:

- This problem is similar to the producer/consumer problem which we have discussed in class.
- Treat the readers/writers two queues (one for each direction), and we need two producers each putting processes into the queues at the appropriate times.
- To get an 80% chance of something, you can generate a random number modulus 10, and see if its value is less than 8. It's like flipping an unfair coin.
- Using STL queue definitely eases your work on queue structure.
- When debugging your program, be reminded of using "kill" to kill your process/threads. Otherwise, if you bugs create too many threads and takes too much resource, the administrator will block your account activities for certain time period for security concerns.

Resources:

- In this project, you will use pthread. The example code below makes your threads sleep for a specific time by using pthread synchronization mechanism.

```
#include    <stdio.h>
#include    <stdlib.h>
#include    <pthread.h>
#include    <time.h>

/*****************************************************************************
pthread_sleep takes an integer number of seconds to pause the current thread We provide this function because one does not exist in the standard pthreads library. We simply use a function that has a timeout.
 *****************************************************************************/
int pthread_sleep (int seconds)
{
    pthread_mutex_t mutex;
    pthread_cond_t conditionvar;
    struct timespec timetoexpire;

    if(pthread_mutex_init(&mutex,NULL))
    {
        return -1;
    }
    if(pthread_cond_init(&conditionvar,NULL))
    {
        return -1;
    }

    //When to expire is an absolute time, so get the current time and add
    //it to our delay time
    timetoexpire.tv_sec = (unsigned int)time(NULL) + seconds;
    timetoexpire.tv_nsec = 0;

    return pthread_cond_timedwait(&conditionvar, &mutex, &timetoexpire);
}

/*****************************************************************************
 * This is an example function that becomes a thread. It takes a pointer
 * parameter so we could pass in an array or structure.
 *****************************************************************************/
int *worker(void *arg)
{
    while(1)
    {
        printf("Thread Running\n");
        fflush(stdout);
        pthread_sleep(1);
    }
}

/*****************************************************************************
 * The main function is just an infinite loop that spawns off a second thread
 * that also is an infinite loop. We should see two messages from the worker
 * for every one from main.
 *****************************************************************************/
int main()
{
    pthread_t t_id;

    if ( -1 == pthread_create(&t_id, NULL, worker, NULL) )
```

```
    {
        perror("pthread_create");
        return -1;
    }

    while(1)
    {
        printf("Main Running\n");
        fflush(stdout);
        pthread_sleep(2);
    }

    return 0;
}
```

- For more on Bounded Buffer see : https://www.cs.mtu.edu/~shene/NSF-3/e-Book/SEMA/TM-example-buffer.html

- For Pthread semaphores and mutexes, also refer to **alp-ch04-threads.pdf**, section 4.4.2 – 4.45

- A full pthread reference can be found online at http://www.llnl.gov/computing/tutorials/pthreads/

- The following functions may be needed in your program; use "man" for more details:
  - <pthread.h > *thread control*: pthread_attr_init(), pthread_create(), pthread_join()
  - <semaphore.h> sem_init(), sem_wait(), sem_trywait(), sem_post()
  - <time.h> time(), localtime(), strftime()

Grading and Submission:

- Your grade will be based on the correctness of your code and largely based on your code's ability to produce correct output in the test cases. Style and legibility is important, especially if the code is hard to find correct: if we can't see that your code is correct, we can't give you points. **Crashing or failing to compile is of course not correct behavior**.

The final project submission	
Bounded Buffer Producer/Consumer	50%
Readers/Writers	50%
Total	100%

- What to submit: A readme file that briefly describes each file and how to run the program, a Makefile file, and all source file. Use **tar/winzip** to pack all these files into a package named **project2.tar**
- **Submission :** Create a folder project2 in your OnCourse dropbox and upload the zip/tar file of your project.

Grading Policy:

- If you do not have the files specified above, it will result in a score of zero.
- If the program cannot be compiled and run, it will result in a score of zero.
- If you and your team miss the checkpoint 1, it will result in a score of zero.