CS492 Programming Assignment 1

The due date for this assignment is *11:59pm Monday, March 5, 2018.* This assignment is worth 10% of your final grade.

- Late penalty: Late assignment (even by 2 seconds) will be given a -25% decrease penalty per day, for the first 2 days after the deadline. So, if you send an assignment 1 second late, you will receive 75% of your grade for the assignment. If you send it, 25 hours late, you will receive 50% of your grade for the assignment etc. After 48hrs from the deadline there will be a -90% decrease penalty, so you will receive 10% of your grade.
- The program must be written in C or C++, and run on a linux machine. (coordinate with your CA)
- ALL source code you submit must be well documented (documentation is an indicator of understanding!)
- Programs that cannot be compiled by CA will receive an automatic grade of zero (0)
- Any sign of collaboration between teams will result in a 0 and will be reported to the Honor Board. The submissions might be tested for similarity using the MOSS, or similar software.

The goal of this assignment is to simulate a producer-consumer system using threads, mutexes, condition variables and scheduling. You will experiment with varying the numbers of available producer and consumer threads, the number of products, and the scheduling algorithm used to schedule the consumption of products. Your program must accept the following parameters at the command prompt in the order specified:

- P1: Number of producer threads
- P2: Number of consumer threads
- P3: Total number of products to be generated by all producer threads
- P4: Size of the queue to store products for both producer and consumer threads (0 for unlimited queue size)
- P5: 0 or 1 for type of scheduling algorithm: 0 for First-Come-First-Serve, and 1 for Round-Robin
- P6: Value of quantum used for round-robin scheduling
- P7: Seed for random number generator

The scheduling algorithms your program must simulate are:

- 0: First Come, First Serve
- 1: Round-Robin

Man pages that you may need to consult are:

- *man pthread*

- *man 3 random*
- *man 3 srandom*
- *man 3 drand48*
- *man 3 srand48*
- *man 2 gettimeofday*
- *man 3 clock*

Products

Each product consists of

- A unique product id, which can be either a string or an integer.
- A time-stamp of when the product was generated, and
- The "life" of the product, which is a positive integer number that is randomly generated. The value should be capped at 1024 numbers by calling *random* in the following manner: *random()%1024*.

To seed the random generator uniquely, use parameter *P7* given at the command prompt. Doing so enables you to test out the same dataset (since the same set of random numbers will be generated given the same seed) on the different scheduling algorithms. To generate different sets of random numbers, use a different seed.

Use the *gettimeofday* or *clock* functions to obtain a timestamp when products are generated.

Producer and Consumer Threads

There are multiple producer threads running in parallel. The number of these threads is given by the input parameter *P1.* Each producer thread must have a unique producer id. Producer threads generate products to be placed in a fixed-size queue. After a product is inserted into the queue, it calls *usleep(x)* to sleep 100 milliseconds. The *usleep(x)*function suspends the caller thread from execution for *x* microseconds, so that the caller thread does not hold onto the CPU forever and allow other threads to run. Type man *usleep* for detailed information on the usleep library. **Note:** the input parameter for usleep() function is **microsecond**, while you need to let the thread sleep for 100 **milliseconds**.

You should keep track of the total number of products that have been produced by all producer threads. When it reaches the given input parameter *P3*, all producer threads terminate producing and return.

On the other side of the queue, multiple consumer threads take products out of the queue for consumption. The number of these threads is given by the input parameter *P2.*In which orders the products are taken out of the queue is decided by a scheduling algorithm.
The **consumption** of a product is simulated by calling the *fn(10)* function *N*times, where *N* is the product's life (see Products for definition of "life"), and *fn(10)* is to generate the 10th Fibonacci number. After a product is consumed, the consumer thread prints out the product ID and calls usleep() to sleep 100 milliseconds.

You should also keep track of the number of products that have been consumed. When the number reaches the value of input parameter P3, which is the total number of products, all consumer threads terminate consuming and return.

In both producer and consumer threads, products are maintained in a fixed-size queue whose size is given by the input parameter *P4*. For producer threads, when there is no space in the queue to store a product, the producer thread must wait till a space is free. When there are no products in a queue to take, the consumer threads have to wait. You need to protect insertion and removal of the products from the queue via critical sections.

Threads

Threads are implemented in the pthreads package (include pthread.h in your source code). Useful function calls include *pthread_create*, *pthread_exit*, *pthread_join*, and *pthread_cancel*. The calling process needs to wait for every thread to complete execution before exiting (because all running threads will terminate upon termination of the calling process). This is accomplished by having the calling process call *pthread_join* on all threads. An example is shown below for 2 threads. Type *man pthread* for detailed information on the pthreads library.

```
void main (int argc, char** argv)
{
pthread_t *thread1, *thread2;
int retval[1];

//Create first thread
thread1 = (pthread_t*) calloc (1, sizeof(pthread_t));
error = pthread_create (thread1, NULL, (void*(*)(void*))(&thread_func1), NULL);
if (error != 0) printf ("Error number: %i\n", error);

// Create second thread
thread2 = (pthread_t*) calloc (1, sizeof(pthread_t));
error = pthread_create (thread2, NULL, (void*(*)(void*))(&thread_func2), NULL);
if (error != 0)printf ("Error number: %i\n", error);
// Other statements ....
.....

// At end of calling function, wait for all threads to complete
pthread_join (*thread1, (void**)(&retval));
pthread_join (*thread2, (void**)(&retval));
}
```

In the above code snippet, *thread_func1* and *thread_func2* are the functions that the threads execute. Function arguments can also be specified for *thread_func1* and *thread_func2* in the last argument of the *pthread_create* call. Just cast the argument list to (void*). *man pthread_create* for details.

After the required number of products have been produced (respectively, consumed), the producer threads (respectively, consumer threads) have to terminate. You can call *pthread_exit* from the code of the threads to be terminated.

Critical Section, Mutex and Condition Variables

Critical Section

Whenever shared data is accessed (for example, inserting and removing from the queue), access to the data structure needs to be protected within a critical section. There are various approaches to implement the protection (see our lecture notes and the textbook). In this assignment, you will need to use mutexes and condition variables to implement protection within critical section.

Mutexes

Pthread library provides support for mutexes.

- The mutex object is defined as *pthread_mutex_t* type.
- To use mutex objects, you will need to first call *pthread_mutex_init()* function to initialize it.
- The mutex object referenced by mutex is locked by calling *pthread_mutex_lock()*. If the mutex is already locked, the calling thread blocks until the mutex becomes available.
- The *pthread_mutex_unlock()* function releases the mutex object referenced by mutex. If there are threads blocked on the mutex object referenced by mutex when pthread_mutex_unlock() is called, resulting in the mutex becoming available, the scheduling policy is used to determine which thread shall acquire the mutex.
- The *pthread_mutex_destroy()* function removes the mutex object from system.

Condition Variables

Pthread library also provides support for condition variables.

- The condition variable is defined as *pthread_cond_t* type.
- To use condition variables, you will need to first call *pthread_cond_init()* function to initialize it.
- The *pthread_cond_wait()* function is used to block on a condition variable. It is called with mutex locked by the calling thread. The function atomically release mutexand cause the calling thread to block on the condition variable *cond*;
- The *pthread_cond_signal()* function releases the lock on the condition variable.
- The *pthread_cond_destroy()* function removes the condition variable from system.

Here is a simple example of using condition variables.

```
void main (int argc, char** argv)
{
//initialize two condition variables
pthread_cond_t *notFull, *notEmpty;
(pthread_cond_t *) notFull = (pthread_cond_t *) malloc (sizeof (pthread_cond_t));
pthread_cond_init (q->notFull, NULL);
(pthread_cond_t *) notEmpty = (pthread_cond_t *) malloc (sizeof (pthread_cond_t));
pthread_cond_init (q->notEmpty, NULL);
}

//producer function
void *producer (void *q)
{
while (queue->full) pthread_cond_wait (notFull, ...);
//insert item
...
pthread_cond_signal (notEmpty);
...
}

//consumer function
void *consumer (void *q)
{
while (queue->empty)pthread_cond_wait (notEmpty, ...);
//Remove item
...
pthread_cond_signal (notFull);
...
}
```

For more details of mutexes and condition variables in pthread library, type "man *pthread*" or google "pthread".

Scheduling Algorithms

The consumer thread consumes the products in the queue based on a scheduling algorithm. The performance of your scheduling algorithms can be measured by the total time taken to complete all of the jobs and by the time taken to complete individual jobs (see Performance Analysis). Consult *Modern Operating Systems*, Ch.2.5, as well as our lecture notes (available on Canvas), for details on the scheduling algorithm. The data structure and method of insertion and removal from the data structure will depend on the scheduling algorithm.

With the *First Come, First Serve* scheduling algorithm, products are sorted in the order in which they were inserted into the queue. After a product is consumed (see Producer and Consumer Threads for definition of "consumption"), the consumer thread prints out the product ID and calls usleep() to sleep 100 milliseconds. Note that the producer thread does not implement a scheduler, and hence, essentially operates in a FCFS manner.

In the *Round-Robin* algorithm, some finite time, i.e., a *quantum*, must be spent for each product in the queue. In this assignment, we simulate the quantum as the number of times a specific function is called. The value of the quantum is given by the input parameter P6. Let it be *q*. The round-robin algorithm can be simulated as following:

The products take turns to be ‘consumed’ by the consumer threads. Every time a consumer thread consumes a product,

- First, the thread checks whether the product's life *l>=q*.
    - If yes, the thread updates the product's "life" *l* to be *l-q*. As simulation of using quantum, the thread calls Fibonacci function *fn(10) q* times.
    - Otherwise, the thread removes the product from the queue, calls Fibonacci function *fn(10) l* times, and prints the product's ID.
- Next, the consumer thread sleeps 100 milliseconds, no matter whether it removes the product from the queue or not.

You should experiment with various settings for *quantum*. If it is too large, round-robin will essentially behave as *First Come, First Serve*. If it is too small, your consumers will spend too much time retrieving and restoring products and not enough time consuming.

Experimentation

Producers and Consumers

In order to analyze how the consumers and producers are behaving, you will need to print out the producer/consumer/product ids associated with when:

1. A producer has completed generating and inserting a product
2. A consumer has consumed a product

For example, your output may look like:
Producer 1 has produced product 1.
Producer 2 has produced product 3.
Consumer 2 has consumed product 3.
Producer 2 has produced product 7.

Initially, use 4 producers and 4 consumers, queue size of 10, and 100 products in total. Then analyze the behavior of the system as you change the program arguments. You can analyze the behavior by seeing how many products are consumed by each consumer. You can also use the *gettimeofday* function to look at the execution times. Some cases you may want to look at are below. Try to come up with other interesting cases and interesting ways to analyze the system.

- Lots more consumers than producers
- Lots more producers than consumers

Scheduling Algorithms

Experiment with different parameters for the scheduling algorithms. For example, with the *Round-Robin* algorithm, you can experiment with different quantum values. Try small ones first, then change to large ones. Again, try to come up with interesting cases and interesting ways for analysis.

- Small quantum, with products of large "life" value.
- Large quantum, with products of small "life" value.

Performance Analysis

The performance of your scheduling algorithms can be measured by the total time taken to complete all of the jobs and by the time taken to complete individual jobs. Use the *gettimeofday* or *clock* functions to obtain a timestamp when products are generated by producers and are printed out by consumers. Calculate the time taken to finish consumption by taking the difference between these two timestamps.

For each set of input parameters and variations on scheduling algorithms, calculate the following metrics:

- Total time for processing all products
- Min, max, and average *turn-around* times
- Min, max, and average *wait* times
- Producer throughput
- Consumer throughput

The *turn-around* time is the time between when a product is produced until when it has been completely consumed (printed out). The *wait* time is the time the product spends waiting in the consumer queue (time between when a product is inserted into the queue and when it is removed for consumption). The *producer throughput* is the number of products inserted into queue per **minute**. The *consumer throughput* is the number of products printed out per minute. Note that if the producer throughput is low, it will cause the consumer throughput to be low.

From the above metrics, what can you conclude about the different scheduling algorithms. Which algorithm is better, in terms of producer throughput, consumer throughput, etc.? What differences are there in average times when few products (for example, 100) are generated versus many products (for example, 5000)?

You MUST obtain these statistics for at least 3 runs of the program for EACH set of input parameters (number of producers, consumers, size of the queue, selected scheduling algorithm).

Implementation Requirements:

Grade Based On:

- Creation of products (5%)
- Creation of producer threads (10%)
- Creation of consumer threads (10%)
- Correct implementation of the queue with mutexes and condition variables (20%)
- Correct implementation of the global counter with mutual exclusion protection (5%)
- Correct implementation of First-come-first-serve scheduling (5%)
- Correct implementation of Round-robin scheduling (15%).
- Correct implementation of termination of product&consumer threads (5%)
- Experimentation and analysis of results (10%)
- Written analysis of experimentation (no more than 1 page long) (10%)
- Documentation (comments in code) (5%)

Submission:

- Create a tar/zip file of your assignment.

- Submit the tar/zip file using Canvas. Login using your Stevens Pipeline account which should have been created for you upon enrollment.
- Ensure your name and login name appear in each file you submit.

Hints and help:

- Start early. Tackle the problem in units. First, implement threads and make sure threads are created and exit properly. Second, implement insertion and removal from the fixed-size queue using mutex and condition variables. Finally, implement the scheduling algorithms.
- Sample Makefile is provided. (attached with the homework description)
- Use the Unix *man* command to find out about how to use the various function calls mentioned above.
- Remember to flush out standard out so that you can see your print statements (*fflush(stdout)*)