DUE DATE: **Before the end of your lab session**

CSE-381: Systems 2

Exercise #9

Max Points: 20

Objective: The objective of this exercise is to:

- Review producer-consumer model with shared queues
- Understand the explicit use of monitors
- Explore the effectiveness of using monitors to coordinate threads

Submission: Save this MS-Word document using the naming convention *MUid*`_Exercise9.docx` prior to proceeding with this exercise. Upload the following at the end of the lab exercise:

1. This MS-Word document saved with the naming convention *MUid*`_Exercise9.docx`.
2. Program developed in second part of this exercise.

You may discuss the questions with your instructor or TA

For this exercise, you may work as a team of 2. If you are working as a team:

- Each team member's name must be in the copyright message
- **Each member must submit the source codes & document independently**

Name (if working as team include both team members names):	Ce Zhang

Part #1: Review concept of busy waiting vs. Monitors

Estimated time to complete: 15 minutes

Background: The producer-consumer model that uses a shared, finite-size queue for sharing data between multiple threads is widely used in many applications, such as: database servers, shared-printer spoolers, disk I/O, etc.

Exercise: Provide brief answers to the following questions to improve your understanding of pertinent concepts:

1. What is a critical section? Why is it used? How is it accomplished in a program?

   A critical section is a region of code in which one thread can work on shared resource. There will be no race conflict. A binary is used to create a critical section in the following manner: threads try to lock the mutex as system call to let other threads know they must block. If it is unsuccessful the OS waits until the mutex is zero what means it has been locked.

2. Using online C++ API documentation (http://en.cppreference.com/w/cpp/thread/mutex), briefly describe the 3 key methods associated with a `std::mutex`.

   The 3 operations:
   1. lock() that locks a mutex and blocks until it can lock the mutex
   2.try_lock() that lock but returns very quickly if that cannot lock
   3.unlock(): Unlocks the mutex

3. If a `std::mutex` has methods to lock and unlock it, then what is the purpose of a `std::lock_guard`?

   We want that all method be deadlock. And the lock_guard will help us to ensure the mutex will not be unlock, it will ensure all is unclock in the end.

4. What is a `std::unique_lock`? What is the key difference between a `std::unique_lock` and a `std::lock_guard`?

   Unique ensure that it is always be a mutex in the one unique thread.
   lock_guard is a 1-time locking/unlocking operation!

5. What is a Monitor or `std::condition_variable`? How is it different from a `std::unique_lock`?

   A condition_variable has two ways.
   1. block and wait until a specific condition is good
   2. a lock on a given mutex will be required.

   A std::unique_lock does not operate on a condition but is just used for locking/unlocking a mutex.

Part #2: Using Monitors instead of Mutex [15 points]

Estimated time to complete: 60 minutes

Background: The producer-consumer model for sharing data between multiple threads (performing operations that can take different amounts of time) is widely used. There are two different approaches to implementing the producer-consumer model depending on multi-threaded or multi-process scenarios:

1. **Busy-wait approach**: A mutex (*i.e.*, a binary semaphore) based approach that involves busy waiting (this implementation is given to you) which is useful in certain cases where the application can monopolize the resources and real-time interactions are highly desired.

This approach of busy waiting is also referred to as a "spin lock" (the program spins a loop waiting for the mutex to be unlocked).

2. **Monitor-based approach**: A monitor-based approach that avoids busy waiting (you need to implement this version of the program) and is the most commonly used approach as provides an efficient use of CPU/energy.

In this part of the exercise you are expected to:

1. Convert a given spin-lock based implementation of a producer-consumer type application into a version that uses monitors (implemented by `std::condition_variable` in C++).
2. Compare the CPU-utilization of the two versions of the same program. **Note that the 2 version of the program should generate exactly the same output**.

Exercise: Complete this part of the exercise using the following procedure:

1. Create a `NetBeans` project. Download the supplied starter code to your project. Review the operation of the program to study the `producer()` and `consumer()` methods. Ensure you can explain the 2 scenarios when the program "busy waits" or "spins" – i.e., keep trying to do some operation until the thread can proceed with operation.
2. Compile the program in `NetBeans`.
3. Linux provides a `time` utility to measuring the time taken to run a program that can be used as shown in the sample output below (the actual timings you observe will be different and that is to be expected):

```
Exercise9$ /usr/bin/time ./Exercise9_Part2 > ex9_orig.txt
13.14user 0.03system 0:06.62elapsed 199%CPU (0avgtext+0avgdata 5520maxresident)k
0inputs+264outputs (0major+394minor)pagefaults 0swaps
```

Understanding the statistics reported by `/usr/bin/time` above:

User time	Sum of time for all threads for which program was running on the CPU.
Elapsed time	The actual time taken for the program to run
%CPU	In Linux each core is counted as 100%. So, if a program runs 2 threads that use 2 cores, the %CPU will be reported as 200%

Each time a program is run, the actual time taken to run the program will vary depending on the load and other activities occurring on the system. Consequently, on multi-user, multi-tasking systems timing measurements have to be repeated in order to ensure that consistent timings are obtained and the consistent timings are averaged to obtain a suitable runtime value.

Using the time command shown above, run the given program without any modifications three times (such that the timings are consistent) and note the timings in the table below. Note that your %CPU should be about 199% because 2 threads are running, each using 100% CPU and Linux will report this as 100% + 100% == 200%:

Timings from given semaphore-based busy waiting application		
	Elapsed time (sec)	**%CPU**

Observation #1	5.99	199%
Observation #2	5.99	199%
Observation #3	5.99	199%
Average	5.99	199%

4. Now modify the `producer` and `consumer` methods to use monitors instead of semaphores to avoid busy waiting. You will need to use a `std::condition_variable` and `std::unique_lock` to implement a monitor (using `wait()` and `notify_one()` methods). **Refer to the lecture slides for examples.**

   NOTE: The output from the revised version of the program should be exactly the same as that of the original version. You may verify that the outputs are identical using `diff` as shown below:

```
$ exercise9$ /usr/bin/time ./Exercise9_Part2 > ex9_monitor.txt
$ diff ex9_orig.txt ex9_monitor.txt
$
```

5. Now that you have successfully re-implemented the application using monitors and you have verified it is generating the same output it is important to appreciate the efficiency implications associated with it. Using the time command shown above, run the given program without any modifications three times (such that the timings are consistent) and note the timings in the table below. Note that unlike in the previous case, we expected the elapsed time to be about the same (maybe very slightly higher) but the %CPU to be close to 100%.

Timings from given semaphore-based busy waiting application		
	Elapsed time (sec)	**%CPU**
Observation #1	7.35	103%
Observation #2	7.35	103%
Observation #3	7.35	103%
Average	7.35	103%

6. Based on the timings from the two version of the same program answer the following questions:
   a. Which version of the program has a lower average elapsed time? Show average elapsed times for the two versions and the difference in elapsed times as well.

      Both version have about the almost same time with the busy-waiting strategy is faster than the sleep-notify solution.

   b. Which version of the program has a lower average %CPU? Show average %CPU for the two versions and the difference in %CPU as well.

The sleep-notify solution use lower CPU utilization. The revised version utilizes 50% the %CPU as the busy-wait solution.

c. Using the difference in average elapsed time and average %CPU which version of the program seems to be performing better? Record your inferences in the space below and provide suitable explanation in support of your interference.

From the performance the busy-wait solution performs a t bit faster than the sleep-notify solution. but, the sleep-notify solution has lower cpu usage than another.

Part 3: Submit files to Canvas

Upload just the following files to Canvas:

1. This MS-Word document saved as a PDF file.
2. Program developed in Part #2 of this exercise.