Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

Lab: Pthreads Synchronization

In this lab, you will learn the basics of multi-threaded programming, and synchronizing multiple threads using locks and condition variables. You will use the `pthreads` (POSIX threads) API in this assignment.

Before you begin

Please familiarize yourself with the `pthreads` API thoroughly. Many helpful tutorials and sample programs are available online. The chapter on pthreads in OSTEP is also an excellent guide. Practice writing simple programs with multiple threads, using locks and condition variables for synchronization across threads. Remember that you must include the header file `<pthread.h>` and compile code using the `-lpthread` flag when using this API. For example, below is how you will compile a multi-threaded program.

```
$gcc threads.c -lpthread
$./a.out
```

Warm-up exercises

You may write the following simple programs before you get started with this lab.

1. Write a program that has a counter as a global variable. Spawn 10 threads in the program, and let each thread increment the counter 1000 times in a loop. Print the final value of the counter after all the threads finish—the expected value of the counter is 10000. Run this program first without using locking across threads, and observe the incorrect updation of the counter due to race conditions (the final value will be slightly less than 10000). Next, use locks when accessing the shared counter and verify that the counter is now updated correctly.

2. Write a program where the main default thread spawns N threads. Thread i should print the message "I am thread i" to screen and exit. The main thread should wait for all N threads to finish, then print the message "I am the main thread", and exit.

   Note that you must learn how to pass arguments (say, thread number) to the thread start function in order to solve this lab. When you pass a variable as an argument, care should be taken to ensure that the value of the variable is retained until the thread actually runs. So, you must use different variables for different threads, and not reuse the same variable to store different values for different threads.

3. Write a program where the main default thread spawns N threads. When started, thread i should sleep for a random interval between 1 and 10 seconds, print the message "I am thread i" to screen, and exit. Without any synchronization between the threads, the threads will print their messages in any order. Add suitable synchronization using condition variables such that the threads print their messages in the order 1, 2, ..., N. You may want to start with $N = 2$ and then move on to larger values of N. Note that you must solve all these exercises using condition variables, and not just inefficient busy waiting, for synchronization.

4. Write a program with N threads. Thread i must print number i in a continuous loop. Without any synchronization between the threads, the threads will print their numbers in any order. Now, add synchronization to your code such that the numbers are printed in the order 1, 2, ..., N, 1, 2, ..., N, and so on. You may want to start with $N = 2$ and then move on to larger values of N.

Part A: Master-Worker Thread Pool

In this part of the lab, you will implement a simple master and worker thread pool, a pattern that occurs in many real life applications. The master threads produce numbers continuously and place them in a buffer, and worker threads will consume them, i.e., print these numbers to the screen. This simple program is an example of a master-worker thread pool pattern that is commonly found in several real-life application servers. For example, in the commonly used multi-threaded architecture for web servers, the server has one or more master threads and a pool of worker threads. When a new connection arrives from a web client, the master accepts the request and hands it over to one of the workers. The worker then reads the web request from the network socket and writes a response back to the client. Your simple master-worker program is similar in structure to such applications, albeit with much simpler request processing logic at the application layer.

You are given a skeleton program `master-worker-skeleton.c`. This program takes 4 command line arguments: how many numbers to "produce" (M), the maximum size of the buffer in which the produced numbers should be stored (N), the number of worker threads to consume these numbers (C), and the number of master threads to produce numbers (P). The skeleton code spawns P master threads, that produce the specified number of integers from 0 to $M - 1$ into a shared buffer array. The main program waits for these threads to join, and then terminates. The skeleton code given to you does not have any logic for synchronization.

You must write your solution in the file `master-worker.c`. You must modify this skeleton code in the following ways. You must add code to spawn the required number of worker threads, and write the function to be run by these threads. This function will remove/consume items from the shared buffer and print them to screen. Further, you must add logic to correctly synchronize the producer and consumer threads in such a way that every number is produced and consumed exactly once. Further, producers must not try to produce when the buffer is full, and consumers should not consume from an empty buffer. While you need to ensure that all C workers are involved in consuming the integers, it is not necessary to ensure perfect load balancing between the workers. Once all M integers (from 0 to $M - 1$) have been produced and consumed, all threads must exit. The main thread must call `pthread_join` on the master and worker threads, and terminate itself once all threads have joined. Your solution must only use `pthreads` condition variables for waiting and signaling: busy waiting is not allowed.

Your code can be compiled as shown below.

```
gcc master-worker.c -lpthread
```

If your code is written correctly, every integer from 0 to $M-1$ will be produced exactly once by the master producer thread, and consumed exactly once by the worker consumer threads. We have provided you with a simple testing script (`test-master-worker.sh` which invokes the script `check.awk`) that checks this above invariant on the output produced by your program. The script relies on the two print functions that must be invoked by the producer and consumer threads: the master thread must call the function `print_produced` when it produces an integer into the buffer, and the worker threads must call the function `print_consumed` when it removes an integer from the buffer to consume. You must invoke these functions suitably in your solution. Please do not modify these print functions, as their output will be parsed by the testing script.

Please ensure that you test your case carefully, as tricky race conditions can pop up unexpectedly. You must test with up to a few million items produced, and with a few hundreds of master/worker threads. Test for various corner cases like a single master or a single worker thread or for a very small buffer size as well. Also test for cases when the number of items produced is not a multiple of the buffer size or the number of master/worker threads, as such cases can uncover some tricky bugs. Increasing the number of threads to large values beyond a few hundred will cause your system to slow down considerably, so exercise caution.

Part B: Reader-Writer Locks

Consider an application where multiple threads of a process wish to read and write data shared between them. Some threads only want to read (let's call them "readers"), while others want to update the shared data ("writers"). In this scenario, it is perfectly safe for multiple readers to concurrently access the shared data, as long as no other writer is updating it. However, a writer must still require mutual exclusion, and must not access the data concurrently with any other thread, whether a reader or a writer. A reader-writer lock is a special kind of a lock, where the acquiring thread can specify whether it wishes to acquire the lock for reading or writing. That is, this lock will expose two separate locking functions, say, *ReaderLock()* and *WriterLock()*, and analogous unlock functions. If a lock is acquired for reading, other threads that wish to read may also be permitted to acquire the lock at the same time, leading to improved concurrency over traditional locks.

There are two flavors of the reader-writer lock, which we will illustrate with an example. Suppose a reader thread R1 has acquired a reader-writer lock for reading. While R1 holds this lock, a writer thread W and another reader thread R2 have both requested the lock. Now, it is fine to allow R2 also to simultaneously acquire the lock with R1, because both are only reading shared data. However, allowing R2 to acquire the lock may prolong the waiting time of the writer thread W, because W has to now wait for both R1 and R2 to release the lock. So, whether we wish to permit more readers to acquire the lock when a writer is waiting is a design decision in the implementation of the lock. When a reader-writer lock is implemented with *reader preference*, additional readers are allowed to concurrently hold the lock with previous readers, even if a writer thread is waiting for the lock. In contrast, when a reader-writer lock is implemented with *writer preference*, additional readers are not granted the lock when a writer is already waiting. That is, we do not prolong the waiting time of a writer any more than required.

In this part of the lab, you must implement both flavors of the reader-writer lock. You must complete the definition of the structure that captures the reader-writer lock in `rwlock.h`. The following functions to be supported by this lock are also defined in the header file:

- The function `InitalizeReadWriteLock()` must initialize the lock suitably.

- The function `ReaderLock()` is invoked by a reader thread before entering a read-only critical section, and the function `ReaderUnlock()` is invoked by the reader when exiting the critical section.
- The function `WriterLock()` is invoked by a writer thread before entering a critical section with shared data updates, and the function `WriterUnlock()` is invoked by the writer when exiting the critical section.

You must write the code to implement these functions. You must write code that implements reader-writer locks with reader preference in `rwlock-reader-pref.cpp`. Similarly, you must implement reader-writer locks with writer preference in `rwlock-writer-pref.cpp`. Note that both implementations of the lock must share the same header file, including the definition of the reader-writer lock structure. Therefore, your lock structure may have some fields that are only used in one version of the code and not the other.

As part of the autograding scripts, you are given two tester programs that test each variant of your reader-writer lock, and an autograding script that runs both in one go. The tester program takes two command line arguments, say R and W. The program then spawns R reader threads, followed by W writer threads, followed by R *additional* reader threads. Each thread, upon creation, tries to acquire the same reader-writer lock as a reader or writer (as the case may be), holds the lock for a long period of time, and finally releases the lock. The program judges the correctness of your implementation by observing the relative ordering of the acquisitions and releases of these locks. By invoking this tester with different values of R and W, one can test the reader-writer lock code reasonably thoroughly. Of course, you are encouraged to write your own test programs that use the reader-writer lock as well.

Part C: Semaphores using `pthreads`

In this part of the lab, you will implement the synchronization functionality of semaphores using `pthreads` mutexes and condition variables. Let's call these new userspace semaphores that you implement as `zemaphores`, to avoid confusing them with semaphores provided by the Linux kernel. You must define your zemaphore structure in the file `zemaphore.h`, and implement the functions `zem_init`, `zem_up` and `zem_down` that operate on this structure in the file `zemaphore.c`. The semantics of these zemaphore functions are similar to those of the semaphores you have studied in class.

- The function `zem_init` initializes the specified zemaphore to the specified value.
- The function `zem_up` increments the counter value of the zemaphore by one, and wakes up any one sleeping thread.
- The function `zem_down` decrements the counter value of the zemaphore by one. If the value is negative, the thread blocks and is context switched out, to be woken up by an up operation on the zemaphore at a later point.

Once you implement your zemaphores, you can use the program `test-zem.c` to test your implementation. This program spawns two threads, and all three threads (the main thread and the two new threads) share a zemaphore. Before you implement the zemaphore logic, the new threads will print to screen before the main thread. However, after you implement the zemaphore correctly, the main thread will print first, owing to the synchronization enabled by the zemaphore. You must not modify this test program in any way, but only use it to test your zemaphore implementation.

Next, you are given a simple program with three threads in `test-toggle.c`. In its current form, the three threads will all print to screen in any order. Modify this program in such a way that the print statements of the threads are interleaved in the order thread0, thread1, thread2, thread0, thread1, thread2, ... and so on. You must only use your zemaphores to achieve this synchronization between the threads. You must not directly use the mutexes and condition variables of the `pthreads` library in the file `test-toggle.c`.

The script `test-zem.sh` will compile and run these test programs for you and can be used for testing. Please see the compilation commands in the script to understand how to compile code that uses `pthreads` using the `-lpthread` flag.

Part D: Your own synchronization problem

In this part, you will implement your own synchronization pattern, much like the producer-consumer (master-worker) pattern in part A or the reader-writer locks in part B. You may look through the practice problems, or the "Little Book of Semaphores" for inspiration. You may pick any of the existing problems from these sources (except producer-consumer and reader-writer locks, which are covered in parts A and B already, or the simple patterns included as test files in part C), or you can come up with your own toy/real-life problem as well.

Your problem should have at least two different agents/threads, e.g., producers and consumers, each doing different things, and requiring some kind of synchronization between them. The synchronization requirement should be more complex than simple patterns like "thread 2 must run after thread 1" which are given as test cases in part C. Your program should spawn multiple threads to create these different agents, with some randomness in their start times, in order to achieve different interleavings of threads during testing. Each agent/thread should do some dummy work in its start function and print some message when it does the work, e.g., producer prints something when it produces and consumer prints something when it consumes. Without proper synchronization, the threads may function incorrectly, e.g., consumer may consume from an empty buffer. But with correct synchronization added, your print statements should indicate that the threads are synchronized correctly as expected. You must define the expected correct behavior of the various threads and demonstrate in your solution that the correct behavior is indeed achieved by looking at the output.

You will provide two different solutions to your synchronization problem, one using condition variables and the other using semaphores. For condition variables, you will use the CV and mutex abstractions from the `pthreads` API. For semaphores, you will use your own semaphore (zemaphore) abstraction implemented by you in part C above. You should develop, test, and demonstrate the CV and sempahore solution in separate C/C++ files. You will use condition variables/mutexes/semaphores as shared global variables in your program, that are available for use by all threads. During your testing, you should run your programs multiple times, with different interleavings of the threads, to demonstrate that your solution is correct.

There is no starter/template code provided for this part of the assignment. You may use example code from other parts of this assignment to help you get started.

Submission instructions

- For part A, you must submit `master-worker.c`.
  For part B, submit `rwlock.h`, `rwlock-reader-pref.cpp`, and `rwlock-writer-pref.cpp`.
  For part C, submit `zemaphore.h`, `zemaphore.c`, and your modified `test-toggle.c`.
  For part D, submit all code written by you.
- Place these files and any other files you wish to submit in your submission directory, with the directory name being your roll number (say, 12345678).
- Tar and gzip the directory using the command `tar -zcvf 12345678.tar.gz 12345678` to produce a single compressed file of your submission directory. Submit this tar gzipped file on Moodle.