CS61 Section Notes (SOLUTIONS)

Section 10 (Fall 2011)

Note: Solutions to the coding exercises are now available in the vote-sol.c file in the same shared folder as the distribution code (see instructions for accessing this on the next page.)

Synchronization Recap

Remember from last week that when multiple threads are attempting to access a shared variable at the same time, bad things can happen. To keep our concurrent programs running correctly, we must synchronize the execution of the threads. So far, we've seen one method of synchronization: mutual exclusion. Sometimes we have a block of code (called a **critical section**) that must be run atomically in order for the behavior of the program to be correct. We must ensure that while one thread is running the code within the critical section, no other thread may enter the critical section. We can do this using the mutex primitives provided by the `pthreads` library. A mutex of type `pthread_mutex_t` is defined as a global variable. When a thread wishes to enter a critical section, it calls `pthread_mutex_lock(&mutex)` and passes a pointer to the mutex to acquire the lock. When the thread is done, it releases the lock using `pthread_mutex_unlock(&mutex).`

If a thread is currently holding a lock, all other threads must wait to acquire it until the thread holding the lock has released it. Thus, mutual exclusion can dramatically slow down a program. This is generally unavoidable, but we can get the best performance by keeping critical sections as small as possible, and by having multiple mutexes if there are multiple shared variables.

Enough talking. While synchronization is fairly easy to understand as an abstract concept, reasoning through real examples can be very, very hard. The purpose of today's section is to work through a (somewhat) realistic example of a concurrent program to give you practice reasoning about concurrent execution and synchronization, which you will have to do on a larger scale in Lab 5.

Synchronization Exercise Part 1

After the contested 2009 election, the Harvard Undergraduate Council has decided to switch over to a more secure, terminal-based voting system for this week's presidential election. A number of threads, defined with the macro `THREADS`, accept votes from the terminal (we will simulate this by reading them in from a file). Each vote is one character, a single digit from 0 to `CANDIDATES-1` (we must limit ourselves to 10 or fewer candidates). The threads increment the totals for the respective candidates, and the results are tallied after all votes are entered.

There's just one problem: the code isn't synchronized. Your job is to add mutexes to the code so that the vote totals are correct at the end of execution.

Start by getting the code. Login to your CS61 VM as usual. Then, run the following command to copy the directory containing the code to your home directory:

```
cp /home/shared/cs61/section10 -r ~
```

The code for the voting program is in cs61-section/vote.c. You can compile the code by running `make`, and then run it using `make run`. Note that the totals will come out different each time you run it, because the code isn't properly synchronized.

At first, you should ignore the code in `vcthread`, and any code within an `if(secondchoice)` block. These will come in later. You should add synchronization code (mostly calls to `pthread_mutex_lock` and `pthread_mutex_unlock)` in `tallyvote`, where marked in the code. You **should not** modify our code, you should only add your own.

Your first step should be to think about what locks you want to use. Define them as global variables and then initialize them in `main()` using `pthread_mutex_init(&mutex, NULL).`

Notes

- The vote is passed into `tallyvote` in the integer variable `vote1`. Don't worry about `vote2` yet.
- We've split up the vote-incrementing code into two lines: one that loads the total into a temporary local variable, and one that stores the new value, separated by a call to `desynch()`, a function that causes the thread to sleep for a random length of time or not at all. This is to increase the amount of indeterminacy in the execution of your program so it will be easier for you to test your synchronization. Do not remove these calls.
- You may want to start with simple synchronization code, but once you have this working, you should consider whether you can make your code faster by using a more complex locking mechanism. You can time the execution of your program by compiling it with `make timing` and then using `make timerun`. Note that this will disable the behavior of `desynch()` and therefore make it harder to test correctness, so only do this once you're convinced your solution is correct. This will also use a much larger file containing approximately 10,000,000 votes.
- We've provided a script to automatically generate vote trace files in addition to the one that was packaged with the code, to use this script, run
  ```
  py gentrace.py <candidates> <number of votes> <output file>
  ```
  You can input this to `vote` by running `./vote [-sc] < <output file>`

Deadlock

Deadlock is a common problem when dealing with multiple mutexes. Suppose thread A runs the following code:

```
pthread_mutex_lock(&mutex1);
pthread_mutex_lock(&mutex2);
```

At the same time, thread B runs the following code:

```
pthread_mutex_lock(&mutex2);
pthread_mutex_lock(&mutex1);
```

Suppose thread A runs its first line and acquires mutex 1, then the system context switches to thread B, which acquires mutex 2. Since thread A already holds mutex 1, the second line in thread B will cause it to block and return to thread A, which will attempt to acquire mutex 2. Since this mutex is already held by thread B, thread A will also block. Here, we can see that neither thread will ever make any progress, and the program will grind to a halt.

Deadlock can only occur when all of the following conditions hold:
- Mutual Exclusion: A resource (e.g. lock) can only be used by one thread (or more generally, a bounded number of threads) at a time.
- Hold-and-Wait: A thread that already holds some resources can request more.
- No Preemption: A resource can't forcibly be removed from the thread that holds it.
- Circular Wait: Two or more threads form a circular chain in which each thread is waiting for a resource that another thread holds.

To prevent deadlock, we must prevent at least one of these conditions from arising.
1. Propose some mechanisms to prevent deadlock.

Ordering locks is not the only solution! There are many real systems that don't or can't do this.
- Breaking mutual exclusion is not usually possible.
- At the start of a critical section, threads may request all the resources they need at once.
- Threads may be required to abandon resources before waiting (e.g. dropping all your locks before acquiring a new one). Depending on the implementation this could be viewed as breaking hold-and-wait, or as breaking cycles.
- Some systems allow preemption! Many databases use "optimistic" synchronization schemes in which a transaction may fail because some other simultaneous transaction made use of the resources the first transaction ultimately depended on. Other systems use cycle detection schemes and preempt resources if a cycle occurs.
- And of course, ordering locks.

Synchronization Exercise Part 2

Suppose we extend the functionality of the voting program so that voters may enter a first choice vote, followed by a second choice vote. The first choice then receives 2 votes and the second choice (passed to `tallyvote` as `vote2`) receives 1 vote.

Add code to `tallyvote` to synchronize the second-choice voting code. As before, try to make your code run as quickly as possible while still being correct. This time, you must be on the lookout for a potential deadlock. You can test your code by compiling using `make` and then running using `make second`.

You may also notice that, once we start reading a second vote from the terminal, the vote-reading code in `votethread` may have a concurrency bug (Why?) Add some synchronization code to fix this as well.

Semaphores

A semaphore is a synchronization primitive with a built-in integer counter. The function `sem_wait` decrements the counter; the function `sem_post` increments the counter (these functions are sometimes called, respectively, `wait` and `signal` or `P` and `V`). If a thread calls `sem_wait` on a semaphore whose counter is zero, then that thread goes to sleep "on the semaphore" and does not decrement the counter until it is woken up. If a thread calls `sem_post` on a semaphore whose counter is zero, then that thread also wakes up exactly one other sleeping thread.

A semaphore generally used when the counter is associated with some countable resources, and threads want to wait until those resources becomes available.

2. To implement a semaphore, what parts do you need?

There's 3 fundamental components: a counter, a way to make `sem_post` and `sem_wait` atomic, and a way to know which threads are sleeping on the semaphore. There are various ways to do each of these; a simple semaphore might look like:

```
struct semaphore{
      int counter;
      spinlock mutex;
      queue wait_queue;
};
```

Note that the internals of a lock are very similar; a lock has a boolean instead of a counter. In contrast, a condition variable has no internal variable; it's just a spinlock and wait queue!

Condition Variables

A condition variable is a synchronization primitive that has no built-in counter whatsoever; in contrast, a lock has a built-in binary counter (locked/unlocked), and a semaphore has a built-in integer counter. For this reason, condition variables are sometimes thought of as the "most flexible" synchronization primitive.

The function `pthread_cond_wait` takes both a CV and a **locked** lock, and first releases the lock, and puts the calling thread to sleep “on the CV”. When the thread wakes, it will immediately try to re-acquire the lock. The function `pthread_cond_signal` wakes one thread sleeping on the CV, and the function `pthread_cond_broadcast` wakes all the threads.

A CV is generally used when there is some associated condition or event, and threads want to wait until that condition is true, or the event occurs. Note that the CV does not embody its own condition; a CV itself is never “true” or “false”! A lock is kind of like a CV that waits for the condition “unlocked”, and a semaphore is kind of like a CV that waits for the condition “count > 0”.

3. Why does `pthread_cond_wait` take a (locked) lock? Why re-acquire the lock on wakeup?

There is a race condition in which a first thread is about to sleep on a CV, and a second thread signals the CV; the first thread will sleep indefinitely. The lock is needed to protect the condition from changing before the first thread goes to sleep. The lock is reacquired by a waking thread because the thread has to check if the condition is still true, since it may have changed between wakeup and runtime. Locks and semaphores don’t have this problem because their conditions (unlocked and count > 0, respectively) are protected by their own internal atomicity (e.g. the spinlock in answer 2, above).

Synchronization Exercise Part 3

This part involves some fairly difficult reasoning, and is for those who would like an extra challenge. Suppose we want to be updated on the progress of the election every 1,000 votes. We do this by spawning an extra thread, which runs `vcthread` and monitors the total number of votes. When this total reaches `nextprint`, the thread calls `countvotes` and prints the current totals. It then updates `nextprint` to contain the total number of votes at which it will next do a count. You can run the code with this option using `make count`. Note that, even though `vcthread` checks that the total is equal to `nextprint`, the total number of votes printed is not always a multiple of 1,000. Why does this happen?

Add synchronization code to make sure the vote totals print every time the number of votes is a multiple of 1,000, and that the correct totals are printed each time. How fast can you make your solution while keeping it correct? To time your solution, run `make timing` and then `make timecount`.

Hints

- As before, you do not need to, and should not, modify any of the original code to build an efficient implementation.
- Our solution uses one CV and one semaphore, though you may find another solution.

Monitors

A monitor is, essentially, a formal style for using locks and CVs. A monitor is a set of functions, CVs, and shared data. The shared data can only be accessed by the monitor's functions. Only one thread is permitted to be running a function inside the monitor at any time; it's like there's one "big lock" around all the functions in the monitor. This big lock protects the CVs and their associated conditions (i.e., the monitor's shared data).

Most monitor implementations use **Mesa** semantics: this means that when a CV inside the monitor is signaled, the woken thread does not immediately run. Thus, the woken thread needs to check if the condition it was waiting for is still true when it actually gets to run! This is basically the same way that CVs work when used outside of monitors.

Some monitors use **Hoare** semantics: this means that a thread that signals a CV inside the monitor immediately relinquishes control of the monitor to the newly-woken thread. Thus, the woken thread knows the condition is still true when it runs.

A monitor is generally used whenever you would use one or more CVs with one lock and some self-contained variables that represent the conditions you want to wait on. This is a common scenario for CVs; the formalism of the monitor helps you get the implementation right.

Fairness, Starvation, and Speed

All synchronization primitives have to make scheduling decisions; in particular, who gets to run next if multiple threads are waiting? Scheduling should be **fair** for some reasonable definition of “fair”; usually this means that the thread chosen to run is random, or that the primitive follows an equal-opportunity policy such as first-in, first-out. A primitive should never be so unfair as to realistically permit **starvation**, the scenario where a waiting thread is never woken. However, sometimes a “fair” schedule is not optimal for speed!

4. A reader-writer lock is a lock that permits multiple threads to lock it in “read” mode **or** one thread to lock it in “write” mode. In Lab 5, we’ve asked you to implement “writer-priority” RW locks in which, if a writer is waiting for a reader to release the lock, it is not possible for other readers to acquire the lock. Why did we ask you to implement RW locks this way?

If readers are allowed to enter the lock while a writer is waiting, then a continuous stream of readers can starve writers indefinitely. However, the lock will be highly parallel! (But wrong).

5. There are a number of different RW lock implementations that might be called “writer-priority.” Suppose your implementation always prioritizes writers over readers; a reader cannot enter the lock if any writer is waiting. Is there a problem with this implementation? What are its advantages? (**Note that this implementation is acceptable for Lab 5, so don’t panic if you did it this way!**)

If you always prioritize writers, than a continuous stream of writers can starve readers indefinitely! But note that “many readers, few writers” is a common application scenario, so this kind of RW lock is actually useful. It is faster than a strict first-in, first-out implementation because the delayed readers then get to run in one massive batch job when the writers are finished.