15213 Lecture 24: Synchronization Basic

Learning Objectives

- Recognize the necessity of mutual exclusion and the situations in which mutual exclusion is required to ensure coherence and consistency of shared state.
- Apply semaphores to protect shared state in simple situations and indicate the penalties or shortcomings of guaranteeing mutual exclusion.

1 Getting Started

The directions for today's activity are on this sheet, but refer to accompanying programs that you'll need to download. To get set up, run these commands on a shark machine:

1. `$ wget http://www.cs.cmu.edu/~213/activities/lec24.tar`
2. `$ tar xf lec24.tar`
3. `$ cd lec24`

2 Sharing and Mutual Exclusion

The file `badcount.c` contains the code from the `badcnt.c` example shown in lecture which spawns two threads, each of which increments the shared count variable a specified number of times. After joining the two threads, the main thread inspects the value of count. Open and inspect the file in your favorite text editor.

1. Build the program with `$ make badcount`. Run `badcount` with a relatively small value for `niters`, e.g. `$ ./badcount 100`. Then, run `badcount` with a larger value for `niters`, e.g. `$ ./badcount 10000`. What did you notice? Were the counts for both values of `niters` correct?

2. If you observed an incorrect `count`, was it lower or higher than the expected output of $2 \times$ `niters`? Exactly why was this the case? If you didn't observe an incorrect output, will the output always be correct?

3. If you observed incorrect output, could the incorrect output be higher than $2 \times$ `niters`?

4. Write down the values of `niters` you tried as well as the times elapsed for each run of `badcount`. You may wish to try a few other values for `niters` as well, such as 5000, 50000, 100000, 500000, or even 1000000.

`niters`	microseconds (μs)

3 Using Mutexes to Ensure Mutual Exclusion

Shared state and the consistency issues that arise are unavoidable in multithreaded programming. To ensure correctness when multiple threads may be accessing the same data or memory at the same time, programmers use synchronization constructs to enforce *mutual exclusion*, where only one thread can be executing a particular *critical section of code* (usually accessing and/or mutating shared state) at a time.

5. Where is the critical section in the `badcount` program?

6. Implement your proposed changes in the file `goodcount.c` (as provided, a copy of `badcount.c`) using the `pthread_mutex_t` type from the included `pthread.h` header file.

   You can initialize a declared Pthreads mutex with the `pthread_mutex_init` (`$ man pthread_mutex_init`) function, where the `attr` argument should be `0` or `NULL` to indicate default mutex initialization settings. The functions `pthread_mutex_lock` and `pthread_mutex_unlock` will also be useful.

   Did this fix the problem? Briefly explain why.

7. Run your fixed program with the values of `niters` you tried in question 4. Is there a performance penalty? If so, how severe is it? If not, why not? You can copy over your times from question 4 into the table below to compare `badcount` and your revised `goodcount`.

`niters`	`badcount` microseconds (μs)	`pthread_mutex_t` microseconds (μs)

4 *(Advanced)* Using Semaphores to Ensure Mutual Exclusion

The synchronization construct we will now learn about is called the `semaphore`, invented by Dutch computer scientist Edsger Dijkstra in the 1960s. A semaphore, in the most basic terms, is a counter, initialized to some starting (almost always positive) value. There are two operations for interacting with a semaphore: P and V[1]. The P operation waits until a semaphore's value is positive, then decrements the semaphore, while the V operation increments a semaphore's value.

A binary semaphore (that is, a semaphore initialized to 1 and which will always either take the value 0 or 1) is equivalent to a mutex, as we will see in this section. The next lectures will cover more sophisticated situations and applications of semaphores, such as the producer-consumer and the readers-writers problem.

8. Describe how you would use semaphores to protect the shared `count` variable in the `badcount` program. With what value would you initialize the semaphore? Where would you put P and V operations?

9. Implement your proposed changes in the file `goodcount.c` (as provided, a copy of `badcount.c`) using the `sem_t` type from the included `semaphore.h` header file.

   You can initialize a declared semaphore with the `void sem_init(sem_t *sem, int pshared, unsigned int value)` function, where the `pshared` argument should be `0` to indicate that the semaphore is to be shared between threads and not processes. The functions `void P(sem_t *sem)` and `void V(sem_t *sem)` will also be useful.

   Did this fix the problem? Briefly explain why.

[1] P is often said to stand for *proberen* (Dutch for "to test" or "to try"), while V is often said to stand for *verhogen* (Dutch for "to increase"). See `https://en.wikipedia.org/wiki/Semaphore_%28programming%29#Operation_names` for more information.

10. Run your revised program with the values of `niters` you tried in questions 4 and 7. Do semaphores carry a performance penalty? Is the penalty more, less, or just as severe as when using mutexes? Take a guess as to why that might be the case. You can copy over your times from question 4 and 7 into the table below to compare `badcount` and your two versions of `goodcount`.

`niters`	`badcount` microseconds (μs)	`pthread_mutex_t` microseconds (μs)	`sem_t` microseconds (μs)