Lecture 09—More Synchronization

January 23, 2015

Roadmap

Last Time: Race Conditions, Locking (mutexes)

Now: More Synchronization Mechanisms

Part I

More Synchronization

Mutexes Recap

Our focus is on how to use mutexes correctly:

- Call `lock` on mutex `m1`. Upon return from `lock`, you have exclusive access to `m1` until you `unlock` it.
- Other calls to `lock` `m1` will not return until `m1` is available.

For background on selection algorithms, look at Lamport's bakery algorithm.
(Not in scope for this course.)

More on Mutexes

Can also “try-lock”: grab lock if available, else return to caller (and do something else).

Excessive use of locks can serialize programs.

- Linux kernel used to rely on a Big Kernel Lock protecting lots of resources in the 2.0 era.
- Linux 2.2 improved performance on SMPs by cutting down on the use of the BKL.

Note: in Windows, “mutex” is an inter-process communication mechanism. Windows “critical sections” are our mutexes.

Spinlocks

Functionally equivalent to `mutex`.

- `pthread_spinlock_t`, `pthread_spin_lock`, `pthread_spin_trylock` and friends

Implementation difference: spinlocks will repeatedly try the lock and will not put the thread to sleep.

Good if your protected code is short.

Mutexes may be implemented as a combination between spinning and sleeping (spin for a short time, then sleep).

Read-Write Locks

Two observations:

- If there are only reads, there's no datarace.
- Often, writes are relatively rare.

With mutexes/spinlocks, you have to lock the data, even for a read, since a write could happen.

But, most of the time, reads can happen in parallel, as long as there's no write.

Solution: Multiple threads can hold a read lock
    (`pthread_rwlock_rdlock`)
but only one thread may hold the associated write lock
    (`pthread_rwlock_wrlock`);
grabbing the write waits until current readers are done.

Semaphores

Semaphores have a `value`. You specify initial `value`.

Semaphores allow sharing of a # of instances of a resource.

Two fundamental operations: `wait` and `post`.

- `wait` is like `lock`; reserves the resource and decrements the value.
  - If value is 0, sleep until value is greater than 0.
- `post` is like `unlock`; releases the resource and increments the value.

Barriers

Allows you to ensure that (some subset of) a collection of threads all reach the barrier before finishing.

Pthreads: A barrier is a `pthread_barrier_t`.

Functions: `_init()` (parameter: how many threads the barrier should wait for) and `_destroy()`.

Also `_wait()`: similar to `pthread_join()`, but waits for the specified number of threads to arrive at the barrier

Lock-Free Algorithms

We'll talk more about this in a few weeks.

Modern CPUs support atomic operations, such as compare-and-swap, which enable experts to write lock-free code.

Lock-free implementations are extremely complicated and must still contain certain synchronization constructs.

Semaphores Usage

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
```

- Also must link with `-pthread` (or `-lrt` on Solaris).
- All functions return `0` on success.
- Same usage as mutexes in terms of passing pointers.

How could you use as `semaphore` as a `mutex`?

Semaphores Usage

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
```

- Also must link with `-pthread` (or `-lrt` on Solaris).
- All functions return `0` on success.
- Same usage as mutexes in terms of passing pointers.

How could you use as `semaphore` as a `mutex`?

- If the initial `value` is 1 and you use `wait` to lock and `post` to unlock, it's equivalent to a `mutex`.

Semaphores for Signalling

Here's an example from the book. How would you make this always print "Thread 1" then "Thread 2" using semaphores?

```
#include <pthread.h>
#include <stdio.h>
#include <semaphore.h>
#include <stdlib.h>

void* p1 (void* arg) { printf("Thread 1\n"); }

void* p2 (void* arg) { printf("Thread 2\n"); }

int main(int argc, char *argv[])
{
    pthread_t thread[2];
    pthread_create(&thread[0], NULL, p1, NULL);
    pthread_create(&thread[1], NULL, p2, NULL);
    pthread_join(thread[0], NULL);
    pthread_join(thread[1], NULL);
    return EXIT_SUCCESS;
}
```

Semaphores for Signalling

Here's their solution. Is it actually correct?

```
sem_t sem;
void* p1 (void* arg) {
  printf("Thread 1\n");
  sem_post(&sem);
}
void* p2 (void* arg) {
  sem_wait(&sem);
  printf("Thread 2\n");
}

int main(int argc, char *argv[])
{
    pthread_t thread[2];
    sem_init(&sem, 0, /* value: */ 1);
    pthread_create(&thread[0], NULL, p1, NULL);
    pthread_create(&thread[1], NULL, p2, NULL);
    pthread_join(thread[0], NULL);
    pthread_join(thread[1], NULL);
    sem_destroy(&sem);
}
```

Semaphores for Signalling

1. `value` is initially 1.
2. Say `p2` hits its `sem_wait` first and succeeds.
3. `value` is now 0 and `p2` prints “Thread 2” first.

- If `p1` happens first, it would just increase `value` to 2.

Semaphores for Signalling

1. `value` is initially 1.
2. Say `p2` hits its `sem_wait` first and succeeds.
3. `value` is now 0 and `p2` prints "Thread 2" first.

- If `p1` happens first, it would just increase `value` to 2.

- Fix: set the initial `value` to 0.

  Then, if `p2` hits its `sem_wait` first, it will not print until `p1` posts (and prints "Thread 1") first.

volatile Keyword

- Used to notify the compiler that the variable may be changed by "external forces". For instance,

```
int i = 0;

while (i != 255) {
  ...
```

volatile prevents this from being optimized to:

```
int i = 0;

while (true) {
  ...
```

- Variable will not actually be volatile in the critical section and only prevents useful optimizations.
- Usually wrong unless there is a **very** good reason for it.

C++ atomics

Coming soon. Short version: wrap the type in `atomic`, e.g.
`atomic<int> x;`