ECE 344: Operating Systems
Lecture 12

Locking

1.2.0

Jon Eyolfson
October 4/5, 2021

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License

Monitors Are Built Into Some Languages

With object oriented programming, developers wanted something easier to use

Could mark a method as monitored, and let the compiler handle locking

- An object can only have one thread active in its monitored methods

It's basically one mutex per object, created for you

- The compiler inserts calls to lock and unlock

Java's `synchronized` Keyword is an Example of a Monitor

```
public class Account {
  int balance;
  public synchronized void deposit(int amount)  { balance += amount; }
  public synchronized void withdraw(int amount) { balance -= amount; }
}
```

the compiler transforms to:

```
  public void deposit(int amount) {
    lock(this.monitor);
    balance += amount;
    unlock(this.monitor);
  }
  public void withdraw(int amount) {
    lock(this.monitor);
    balance -= amount;
    unlock(this.monitor);
  }
```

Condition Variables Behave Like Semaphores

You can create your own custom queue of threads

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr)
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
                           const struct timespec *abstime);
```

The `wait` functions add this thread to the queue

`signal` wakes up one thread, `broadcast` wakes up all threads

Condition Variables MUST Be Paired with a Mutex

Any calls to `wait` must already hold the mutex
(`signal` and `broadcast` may not)

Why? You may think `wait` needs to add itself to the queue safely (no data races)
It needs the mutex as an argument to unlock it before going to sleep
The thread will hold the locked `mutex` before and after the call

One mutex can also protect multiple condition variables

We'll only consider calls to `wait` and `signal`

The `wait` Call Does Not Contain Data Races

Understand what `wait` does (for condition variables!)

The thread calling `wait`:

1. Adds itself to the queue for the condition variable
2. Unlock the mutex
3. Gets blocked (it can no longer be scheduled to run)

The thread calling `wait` needs another thread to call `signal` or `broadcast`, then if it's selected:

1. Gets unblocked (it can be scheduled to run)
2. Tries to lock the mutex again, `wait` returns when it gets it

We Can Use Condition Variables for Our Producer/Consumer

```
pthread_mutex_t mutex;
int nfilled;
pthread_cond_t has_filled;
pthread_cond_t has_empty;

void producer() {
  // produce data
  pthread_mutex_lock(&mutex);
  while (nfilled == N) {
    pthread_cond_wait(&has_empty,
                      &mutex);
  }
  // fill a slot
  ++nfilled;
  pthread_cond_signal(&has_filled);
  pthread_mutex_unlock(&mutex);
}
```

```
void consumer() {
  pthread_mutex_lock(&mutex);
  while (nfilled == 0) {
    pthread_cond_wait(&has_filled,
                      &mutex);
  }
  // empty a slot
  --nfilled;
  pthread_cond_signal(&has_empty);
  pthread_mutex_unlock(&mutex);
  // consume data
}
```

What’s the Issue with the Following Code?

```
/* Thread 1 */
pthread_mutex_lock(&mutex);
while (!condition) {
  pthread_cond_wait(&cond, &mutex);
}
pthread_mutex_unlock(&mutex);

/* Thread 2 */
condition = true;
pthread_cond_signal(&cond);
```

Can We Change the `while` to an `if`?

```
/* Thread 1 */
pthread_mutex_lock(&mutex);
if (!condition) {
  pthread_cond_wait(&cond, &mutex);
}
// What could happen here?
pthread_mutex_unlock(&mutex);

/* Thread 2 */
pthread_mutex_lock(&mutex);
condition = true;
pthread_mutex_unlock(&mutex);
pthread_cond_signal(&cond);

/* Thread 3 */
pthread_mutex_lock(&mutex);
condition = false;
pthread_mutex_unlock(&mutex);
```

Condition Variables Serve a Similar Purpose as Semaphores

You can think of semaphores as a special case of condition variables

They'll go to sleep when the value is 0, when it's greater than 0 they wake up

You can implement one using the other, however it can get messy

For complex conditions condition variables offer much better clarity

Locking Granularity is the Extent of Your Locks

You need locks to prevent data races

Lock large sections of your program, or divide the locks and use smaller sections?

What if you want to parallelize your hash table?

Things to consider about locks:

- Overhead
- Contention
- Deadlocks

Locking Overheads

- Memory allocated
- Initialization and destruction time
- Time to acquire and release locks

The more locks you have, the greater each cost is going to be

You Do NOT Want Deadlocks

The more locks you have, the more you have to worry about deadlocks

Conditions for deadlocking:

1. Mutual Exclusion (of course for simple locks)
2. Hold and Wait (you have a lock and try to acquire another)
3. No Preemption (we can't take simple locks away)
4. Circular Wait (waiting for a lock held by another process)

A Simple Deadlock Example

Consider two processors trying to get two *locks*:

Thread 1	**Thread 2**
`Get Lock 1`	`Get Lock 2`
`Get Lock 2`	`Get Lock 1`
`Release Lock 2`	`Release Lock 1`
`Release Lock 1`	`Release Lock 2`

Thread 1 gets Lock 1, then Thread 2 gets Lock 2, now they both wait for each other

Deadlock

You Can Ensure Order to Prevent Deadlocks

```
void f1() {
    locktype_lock(&l1);
    locktype_lock(&l2);
    // protected code
    locktype_unlock(&l2);
    locktype_unlock(&ll);
}
```

This code will not deadlock, you can only get `l2` if you have `l1`

You Could Also Prevent A Deadlock by Using `trylock`

Remember, for `pthread` there's `trylock` that returns 0 if it gets the lock

```
void f2() {
    locktype_lock(&l1);
    while (locktype_trylock(&l2) != 0) {
        locktype_unlock(&l1);
        // wait
        locktype_lock(&l1);
    }
    // protected code
    locktype_unlock(&l2);
    locktype_unlock(&ll);
}
```

This code will not deadlock, it will give up `l1` if it can't get `l2`

We Explored More Advanced Locking

We have another tool to ensure order

- Condition variables are clearer for complex condition signaling
- Locking granularity matters
- You must prevent deadlocks