CS 111: Operating System Principles
Lecture 15

Locking

1.0.0

Jon Eyolfson
May 11, 2021

Locks Ensure Mutual Exclusion

Only one thread at a time can be between the `lock` and `unlock` calls

It does not help you ensure ordering between threads

Assume you had a circular buffer you want to use in a producer/consumer scenario

e.g. `ls | wc`

Semaphores are Used for Signaling

Semaphores have a `value` that's shared between threads/processes

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value)
```

There may up to `value` number of things with the semaphore simultaneously

It has two fundamental operations `wait` and `post`

- `wait` decrements the value atomically
- `post` increments the value atomically

If `wait` will not return until the value is greater than 0

Semaphore API is Similar to `pthread` Locks

```
#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);
```

All functions return 0 on success

The `pshared` argument is a boolean, you can set it to 1 for IPC

For IPC the semaphore needs to be in shared memory

How Could We Make This Print “Thread 1” then “Thread 2”?

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

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

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

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;
}
```

This Code Prints “Thread 1” then “Thread 2”

```
static 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[])
{
  sem_init(&sem, 0, 0);
  /* rest as before */
}
```

No Matter Which Thread Executes First, We Get the Same Order

The `value` is initially 0

Assume “Thread 2” executes first

It executes `sem_wait`, which is 0, and doesn’t continue

“Thread 1” doesn’t have to wait, it prints first before it increments the `value`

“Thread 2” can then execute its print statement

What happens if we initialized the `value` to 1?

We Can Use a Semaphore as a Mutex

How?

Using a Semaphore as a Mutex, Note the `value`

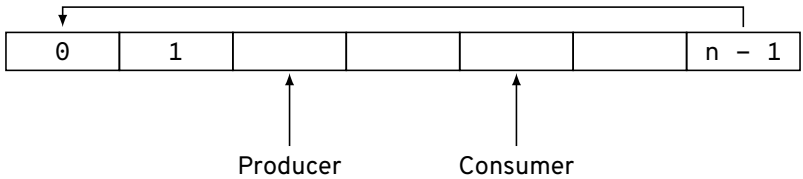
```
...
static sem_t sem;
static int counter = 0;

void* run(void* arg) {
    for (int i = 0; i < 100; ++i) {
        sem_wait(&sem);
        ++counter;
        sem_post(&sem);
    }
}

int main(int argc, char *argv[])
{
  // Create 8 threads
  // Join 8 threads
  sem_init(&sem, 0, 1);
  printf("counter = %i\n", counter);
}
```

Can We Come Up with a Solution for a Producer/Consumer Problem?

Assume you have a circular buffer:



The producer should write to the buffer

As long as the buffer is not full

The consumer should read to the buffer

As long as the buffer is not empty

We Would Create Two Semaphores, What `values` Should We Use?

```
sem_t full;
sem_t empty;

sem_init(&full, 0, /* ??? */);
sem_init(&empty, 0, /* ??? */);

void producer() {
  // produce data
  sem_wait(empty);
  // fill a slot
  sem_post(full);
}

void consumer() {
  sem_wait(full);
  // empty a slot
  sem_post(empty);
  // consume data
}
```

The Previous `values` Depend on the Buffer Size

`full` should always be initialized to 0

`empty` should be initialized to the size of the buffer — `N`

Do we need any extra locking?

The Previous `values` Depend on the Buffer Size

`full` should always be initialized to 0

`empty` should be initialized to the size of the buffer — `N`

Do we need any extra locking?

- No, if there's a single producer and consumer
- Yes, otherwise

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`, `signal`, and `broadcast` must already hold the mutex

Why? `wait` needs to add itself to the queue safely (without data races)
It needs the mutex as an argument to unlock it before going to sleep

One mutex can also protect multiple condition variables

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

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);
  // fill a slot
  if (nfilled == N) {
    pthread_cond_wait(&has_empty,
                      &mutex);
  }
  ++nfilled;
  pthread_cond_signal(&has_filled);
  pthread_mutex_unlock(&mutex);
}
```

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

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?

Lab 3

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 coruse 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

```
Get Lock 1
Get Lock 2
Release Lock 2
Release Lock 1
```

Thread 2

```
Get Lock 2
Get 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(&l1);
}
```

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(&l1);
}
```

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

We Explored More Advanced Locking

Before we did mutual exclusion, now we can ensure order

- Semaphores are an atomic value that can be used for signaling
- Condition variables are clearer for complex condition signaling
- Locking granularity matters, you'll find out in Lab 3
- You must prevent deadlocks