Week 11 – Monday

COMP 3400

Last time

- What did we talk about last time?
  - Synchronization
  - Locks
  - POSIX mutexes

Questions?

Project 3

Back to Locks

POSIX mutex functions

```
int pthread_mutex_init (pthread_mutex_t *mutex,
    const pthread_mutexattr_t *attr);
```

- Create a mutex with the specified attributes

```
int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

- Destroy an existing mutex

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
```

- Acquire a mutex, blocking until you succeed

```
int pthread_mutex_trylock (pthread_mutex_t *mutex);
```

- Try to acquire a mutex, returning non-zero if another thread has the mutex

```
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

- Release the mutex

Mutex example

- Here's a thread that uses a mutex when incrementing a global variable

```
int global = 5;

// Each increment thread gets a pointer to the mutex
void *
increment (void *args)
{
  pthread_mutex_t *mutex = (pthread_mutex_t *) args;

  // Lock for the critical section, then release
  pthread_mutex_lock (mutex);
  global++;
  pthread_mutex_unlock (mutex);

  pthread_exit (NULL);
}
```

Main program

- The following program creates the mutex and passes it to two threads
- Note that the mutex lives on the stack, but that's okay since this function won't return until after the other threads are done

```
pthread_t threads[2];
pthread_mutex_t mutex;

// Initialize the mutex
pthread_mutex_init (&mutex, NULL);

// Create the child threads, passing pointers to the mutex
assert (pthread_create (&threads[0], NULL, increment, &mutex) == 0);
assert (pthread_create (&threads[1], NULL, increment, &mutex) == 0);

// Join the threads
pthread_join (threads[0], NULL);
pthread_join (threads[1], NULL);

// Confirm the result
assert (global == 7);
```

POSIX gotchas

- POSIX mutexes have a few weird things that you **should not do**, because there's no telling what will happen:
  - Trying to lock a mutex that the thread has already locked
    - This isn't a problem in Java, which allows a thread to lock a lock repeatedly without issue
  - Trying to unlock a mutex that a different thread has acquired
  - Trying to lock or unlock a mutex that hasn't been initialized
    - Like all variables in C, it's full of garbage before it's initialized

Spinlocks

- In the old days, we had multiple threads but not multiple cores
- Thus, unlocking a lock would mean that the other thread couldn't acquire the lock until it was scheduled (requiring a context switch)
- Now, we have multicore systems, so threads can run at exactly the same moment in time
- In these situations, it can *sometimes* be faster for a thread to constantly try to acquire a lock (called busy waiting)
  - Then, it can continue onward without a context switch
- Usually, regular mutexes are better because we won't have threads constantly taking up CPU cycles doing nothing
- Even so, POSIX defines a set of spinlock functions with the same functionality as the mutex functions, if you want them

How long should critical sections be?

- Now that you have locks that you can use to protect a critical section, how should you use them?
- In general, you want critical sections to be short so that one thread won't block another unnecessarily
- Nevertheless, breaking up one section of code into several critical sections will introduce penalties because acquiring and releasing locks isn't free
- Consider the examples on the next slide

Two different critical sections

```
// Acquire and release the lock 1,000,000 times
for (i = 0; i < 1000000; ++i)
  {
    pthread_mutex_lock (&mutex);
    global++;
    pthread_mutex_unlock (&mutex);
  }

// Acquire and release the lock only once
pthread_mutex_lock (&mutex);
for (i = 0; i < 1000000; ++i)
  global++;
pthread_mutex_unlock (&mutex);
```

Length of critical sections

- The first example on the previous slide will take much longer, since it has to lock and unlock 1,000,000 times
- On the other hand, the second example will block all other threads from running code that depends on the lock until it's finished
- Neither is very realistic, since incrementing a variable 1,000,000 times in a loop is ridiculous
- There's no simple solution: depends on the problem
- Always getting the right answer is the first goal and then tuning for better performance comes second

Semaphores

Semaphores

- We mentioned semaphores in the context of synchronizing processes that shared memory
- We can use semaphores to synchronize threads as well
- Recall that we think of a semaphore as a non-negative integer that can be incremented and decrementing atomically
  - Calling **`sem_wait()`** (decrement) on a semaphore at **`0`** will block until another thread calls **`sem_post()`** (increment)

Semaphore functions

```
sem_t *sem_open (const char *name, int oflag,
/* mode_t mode, unsigned int value */ );
```

- Return (and possibly create) a named semaphore, using the usual **`oflag`** and **`mode`** flags
- **`value`** determines the initial value of the semaphore (often 0)

```
int sem_wait (sem_t *sem);
```

- Block if the semaphore's value is 0, decrement after continuing

```
int sem_post (sem_t *sem);
```

- Increment the semaphore's value, unblocking a process if the value is 0

```
int sem_close (sem_t *sem);
```

- Close a semaphore

```
int sem_unlink (const char *name);
```

- Delete a semaphore

Semaphores for signaling

- We can use semaphores to signal some event to another thread
- As in our earlier examples with semaphores, we initialize the semaphore to **`0`**
  - The thread waiting for the event will call **`sem_wait()`** on the semaphore
  - The thread signaling that the event has happened will call **`sem_post()`**
  - The waiting thread will be awoken when the signaling thread posts
  - If the signaling thread posts before the waiting starts waiting, it won't have to wait

Semaphore signaling example

- The following code waits for keyboard input and posts on the semaphore when it's done reading it

```
#define MAX_LENGTH 40

struct args {
  sem_t *semaphore;
  char buffer[MAX_LENGTH];
};

// Reads input
void *keyboard_listener (void *args) {
  struct args *data = (struct args *) args;
  printf ("Enter your name here: ");
  assert (fgets (data->buffer, MAX_LENGTH, stdin) != NULL);

  // After reading input, up the semaphore
  sem_post (data->semaphore);
  pthread_exit (NULL);
}
```

Semaphore signaling example continued

- The following code waits on the semaphore and then prints a message based on the string that was entered

```
void *keyboard_echo (void *args)
{
  struct args *data = (struct args *) args;

  // Wait on the signal from the semaphore
  sem_wait (data->semaphore);

  // Trim off at the newline
  char *newline = strchr (data->buffer, '\n');
  if (newline != NULL)
    *newline = '\0';

  // Echo back the name
  printf ("Hello, %s\n", data->buffer);
  pthread_exit (NULL);
}
```

Semaphore signaling example continued

- The following code creates the semaphore and runs the two threads

```
pthread_t threads[2];
sem_t *sem = sem_open ("/COMP3400_Sema", O_CREAT | O_EXCL, S_IRUSR | S_IWUSR, 0);
assert (sem != SEM_FAILED);

// Set up struct instance and pass it to threads
struct args args;
args.semaphore = sem;

assert (pthread_create (&threads[0], NULL, keyboard_listener, &args) == 0);
assert (pthread_create (&threads[1], NULL, keyboard_echo, &args) == 0);

// Wait for both threads to finish, then unlink the semaphore
pthread_join (threads[0], NULL);
pthread_join (threads[1], NULL);
sem_unlink ("/COMP3400_Sema");
```

Mutual exclusion with semaphores

- It should be unsurprising that we can use semaphores instead of locks (POSIX mutexes)
- To do so, we initialize the semaphore to a value of **1**
  - When entering a critical section, a thread waits on (downs) the semaphore
  - When leaving a critical section, the thread posts on (ups) the semaphore
- The first thread reaching the critical section is allowed in because the value is **1**
- If we had initialized to **0**, no threads could enter the critical section

Semaphore as lock example

- The following code adds 10 to a shared value 100,000 times, using a semaphore for mutual exclusion

```
struct args {
  sem_t *semaphore;
  int value;
};

// Adder thread that repeatedly adds 10
void *add (void *args)
{
  struct args *data = (struct args *)args;

  // Atomically add 10 to value 100000 times
  for (int i = 0; i < 100000; ++i)
    {
      sem_wait (data->semaphore);
      data->value += 10;
      sem_post (data->semaphore);
    }
  pthread_exit (NULL);
}
```

Semaphore as lock example continued

- The following code subtracts 10 from a shared value 100,000 times, using the same semaphore for mutual exclusion

```
// Subtractor thread that repeatedly subtracts 10
void *subtract (void *args)
{
  struct args *data = (struct args *) args;

  // Atomically subtract 10 from value 100000 times
  for (int i = 0; i < 100000; ++i)
    {
      sem_wait (data->semaphore);
      data->value -= 10;
      sem_post (data->semaphore);
    }
  pthread_exit (NULL);
}
```

Semaphore as lock example continued

- The following code creates the semaphore and runs the two threads

```
pthread_t threads[2];
// Create semaphore with value 1
sem_t *sem = sem_open ("/COMP3400_Sema", O_CREAT | O_EXCL,
                       S_IRUSR | S_IWUSR, 1);
assert (sem != SEM_FAILED);

// Set up a struct instance with semaphore and initial value 0
struct args args = { sem, 0 };
assert (pthread_create (&threads[0], NULL, add, &args) == 0);
assert (pthread_create (&threads[1], NULL, subtract, &args) == 0);
pthread_join (threads[0], NULL);
pthread_join (threads[1], NULL);
sem_unlink ("/COMP3400_Sema");
printf ("Value: %d\n", args.value); // Should be 0
```

Semaphores as locks

- Semaphores can be used to build a lock library that functions the same as POSIX mutexes

```
typedef struct lock {
  sem_t *semaphore;
  pthread_t owner;
} lock_t;

int mutex_lock (lock_t *lock)
{
  int retvalue = sem_wait (lock->semaphore); // Wait on semaphore
  lock->owner = pthread_self ();             // Set self as owner
  return retvalue;
}

int mutex_unlock (lock_t *lock)
{
  if (lock->owner != pthread_self ())        // Only the owner can unlock
    return -1;
  lock->owner = 0;                           // Clear owner
  return sem_post (lock->semaphore);         // Post on semaphore
}
```

Semaphores as locks

- For mutual exclusion, POSIX mutexes are preferred over semaphores because they're already implemented to work correctly
- With semaphores, you have to initialize them to 1 or face the consequences
  - **0** means that no thread can acquire the lock
  - Greater than **1** means that more than one thread can be in the critical section
- But it's still good to stretch you brain thinking about these things because concurrent programming is hard

Semaphores as multiplexing

- Semaphores can also be used for multiplexing, in which a maximum number of threads are allowed to access a resource
- Consider a club where the bouncer only lets 100 people in
- This kind of synchronization is used less than signaling and mutexes, but it can be useful to prevent slowdown from too many threads using a resource
- Also, it can be used to prevent possible race conditions when there's a fixed number of items but the threads themselves have to select the one they want
    - No more than the maximum number of threads will be allowed to do selection

Multiplexing example with 10 possible ports

- In the following example, **`pool_semaphore`** is initialized to 10, preventing more than 10 threads from selecting ports at the same time

```
sem_wait (pool_semaphore); // Get access to resources

for (int i = 0; i < 10; ++i)    // Try to acquire a port, move to the next if not available
  if (pthread_mutex_trylock (incoming_mutex[i]))
    {
      in = i;
      break;
    }

// Work with incoming port, even if no outgoing port is yet needed

for (int i = 0; i < 10; ++i)    // When an outgoing port is needed, acquire it like incoming
  if (pthread_mutex_trylock (outgoing_mutex[i]))
    {
      out = i;
      break;
    }

pthread_mutex_unlock (incoming_mutex[in]);   // Release incoming port lock
pthread_mutex_unlock (outgoing_mutex[out]);  // Release outgoing port lock
sem_post (pool_semaphore);                   // Leave the port selection area
```

Semaphore summary

- Semaphores are a flexible tool that can be used for signaling, mutual exclusion, and multiplexing
- The key is the initial value of the semaphore
  - **0** for signaling
  - **1** for mutual exclusion
  - Greater than **1** for multiplexing
- Conceptually, the initial value of the semaphore is the maximum number of concurrent accesses

Upcoming

Next time…

- Barriers
- Condition variables

Reminders

- Work on Project 3
- Read sections 7.5 and 7.6