COMP421
Unix Environment for Programmers
Lecture 16: Thread Synchronization

Jeff Wiegley, Ph.D.
Computer Science

jeffw@csun.edu

10/17/2005

> ‘‘Alright, alright. Synchronize your watches. You, you, you and you, paint him red, then kill him!..’’
>
> *–Bhuta (“Help!” The Beatles, 1964)*

Synchronization:

With all parallel tasks there is some requirement for synchronization between the tasks.

In the `fork()` model this is accomplished through the use of Interprocess Communication facilties (IPC):

- Shared memory.
- Semaphores.
- Pipes.
- Signals.
- Message Queues.

None of these are very well designed for the purpose or are easy to use.

Thread problems.

Some aspects of schronization are made easier by threads because they share the same memory space. So you don't have to pass around data on a plate, so to speak.

But, with shared memory space comes a concurrency problem known as "shared data".[a]

```
finetime = 1000000*timestruct.secs + timestruct.millisecs;
```

The above code has a problem if a different thread is responsible for updating the time because the computation actually consists of several atomic computations.

An update could occur between the addition operation of the two parts and you could obtain in incorrect result in this thread.

Two threads cannot be manipulating a complicated data structure, such as a Red-Black tree, simultaneously.

[a] This problem is a large concern for embedded software applications as well.

Thread Synchronization:

Thread synchronization is based on the concepts of:

- *Mutex* (mutually exclusive areas of execution),
- *Reader-Writer* locks, and
- *Condition Variables*.

All of these have been designed for simplicity and ease of use compared to their IPC counterparts.

Mutexes:

The clasic definition of a mutex is an area of code in which only one thread can be executing. Like Java's synchronize keyword.

But in POSIX threads mutex are extended to mean a lock that is held by a single thread. Any thread wanting to obtain the lock must wait until the current owner releases the lock.

A mutex must be created before it can be locked;

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

the mutex *mutex* is a structure and can be allocated either statically or dynamically (by `malloc()`). If allocated dynamically, then the mutex must be destroyed before the dynamic memory is deallocated.

Using Mutexes:

Once you have an initialized mutex you can use it:

Before entering a restricted area of code that you want the thread to execute without fear of another interfering you need to lock the mutex:

```
int pthread_mutex_lock(pthread_mutex t *mutex);
int pthread_mutex_trylock(pthread_mutex t *mutex);
```

`lock()` blocks until the lock is granted. If blocking is unacceptable then `trylock()` can be used and behavior can be switched based on whether or not the lock was granted.

When the thread is done executing the protected code it needs to release the lock so that other threads get an opportunity to execute.

```
int pthread_mutex_unlock(pthread_mutex t *mutex);
```

Some rules:

- Mutex protected areas should be as small and efficient as possible.
  - This maximizes the amount of parallelization of the problem that the machine can take advantage of.
  - Excessively small areas may excessive locking and unlocking which robs computation cycles.
- There is no guarantee as to which blocked thread will obtain the lock when it is released. So effort must be made to make sure that starvation of a thread does not occur.
- Attempting to lock a mutex that the thread already holds will result in deadlock.
- If an area needs two mutexes then all threads should lock the two mutexes in the same order otherwise deadlock can occur.

Reader-Writer locks:

Mutex only allow a single thread to be executing in a mutually exclusive area.

This reduces the amount of parallelism that can be taken advantage of in ceratin circumstances.

- Database activity for example. Since the data is static many `SELECT` queries can be run simultaneously.

To account for such circumstances POSIX threads provide
Reader-Writer locks.

They are similar to mutex except:

1. Any number of threads may hold a reader lock at the same time.
2. Only one thread may hold a writer lock.
3. When a thread holds a writer lock no reader locks are held by any other thread.

Creating/Destroying Reader-Writer locks:

Creating a destroying Reader-Writer locks is similar to mutexes:

```
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
                        const pthread_rwlockattr_t *restrict attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

Using Reader-Writer locks:

Using the Reader-Writer lock is also similiar to using mutexes except there is a second type of non-exclusive read lock possible:

```
int pthread_rwlock_rdlock(pthread_rwlock t *mutex);
int pthread_rwlock_wrlock(pthread_rwlock t *mutex);


int pthread_rwlock_unlock(pthread_rwlock t *mutex);
```

There also exists non-blocking versions of the lock calls:

```
int pthread_rwlock_tryrdlock(pthread_rwlock t *mutex);
int pthread_rwlock_trywrlock(pthread_rwlock t *mutex);
```

Condition Variables:

Mutexes and Reader-Writer locks prevent threads from executing in the same areas[a]

It is some times necessary for threads to coordinate their efforts in the same time or place.

Conditional variables allows threads to rendezvous at a particular point in time.

[a] same "areas" means conceptual areas, not necessarily code areas.

Creating/Destroying Conditional Variables:

Creating a destroying Conditional Variables is similar to the other concurrency facilities for threads:[a]

```
int pthread_cond_init(pthread_cond_t *restrict cond,
                      const pthread_condattr_t *restrict attr);
int pthread_rwlock_destroy(pthread_cond_t *cond);
```

[a]This makes for a really boring lecture but makes pthreads concepts easy to understand, remember and use.

Using Conditional Variables:

There are times when you want to synchronize threads. You want one, or more, threads to wait until another thread catches up or some event occurs.

Conditional variables handle this.

First, you need to create a conditional variable:

```
int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *cond);
int pthread_cond_destroy(pthread_cond t *cond);
```

Conditional Variables need a mutex:

To wait on a condition variable you also need to have a mutex:

```
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);
```

A call to wait does two things:

1. The mutex is unlocked. notice: **un**locked!.
2. The thread is suspended until the condition variable is signaled and the thread is selected to be run again.

You **must** lock the mutex before calling `pthread_cond_wait()` otherwise `pthread_cond_wait()` will not have anything to unlock.

Waking threads up:

Waking from a conditional variable happens in one of two ways:

1. pthread_cond_signal(), or
2. pthread_cond_broadcast()

pthread_cond_signal:

```
int pthread_cond_signal(pthread_cond_t *cond);
```

When a thread makes this call, Exactly one of the waiting threads will be chosen at random and woken up.

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

When a thread makes this call, All waiting threads will be woken up.

When a thread is woken it will atomically lock the mutex again.

Since the mutex is returned to its locked state that was established prior to the wait call, the thread sees life as though the call to wait never occurred.

The predicate loop.

You are going to decide to wait or not based on some predicate (a *predicate* is any test).

There exist reasons why the thread might be woken up even though the predicate is still true. Think about false alarms. When you wake up you should test the predicate again and be prepared to go back to sleep.

This should be done in a loop.

```
status = pthread_mutex_lock(&themutex);

while (test == true) // we want to wait
   status = pthread_cond_wait(&thecond, &themutex);

// do immediate stuff,
pthread_mutex_unlock(&themutex);
```