Outline
- Mutexes
- Condition Variables

Reference
- BLP: Chapter 12
- man pages (install glibc-doc package)
Lecture 22: Threads, Cont.

Linux pthreads Note

- **pthreads** have been implemented in Linux based on two different threading implementations: LinuxThreads and Native POSIX Threads Library (NPTL). NPTL is recommended. The documentation contains information on both so be careful!

- You can find out which version your Linux OS supports by entering (bash):

  ```bash
  $ getconf GNU_LIBPTHREAD_VERSION
  ```
We saw how *semaphores* can be used for thread synchronization in the last lecture. POSIX *unnamed semaphores* are ideally suited for thread synchronization.

The pthreads library provides three synchronization methods: (1) joins, (2) *mutexes*, and (3) condition variables. We looked at a thread join last time too.
A mutex (mutual exclusion object) or lock is used to avoid the simultaneous use of a common resource, such as a shared data structure (either shared explicitly via mmap or implicitly via a shared global variable).

A mutex can never be owned by two threads simultaneously. A thread trying to lock a mutex that is locked by another thread is suspended until the mutex is unlocked.
A **mutex** is similar to a binary semaphore, but typically has special attributes.

- Only the locking thread can unlock it.
- **Nested** calls by the thread that has the lock may be allowed (they won't block).
- Priority inversion avoidance may be implemented.
A mutex is created by declaring it:

```c
pthread_mutex_t mym mutex;
```

It must be initialized before it can be used. It can be initialized when it is declared:

```c
pthread_mutex_t mym mutex = PTHREAD_MUTEX_INITIALIZER;
```
• Or `pthread_mutex_init()` can be called to do the initialization:

```c
int pthread_mutex_init(
    pthread_mutex_t  *mutex,
    pthread_mutex_attr_t *mutex_attr);
```

• The `mutex_attr` argument can be `NULL` to create a mutex with default attributes.
Lecture 22: Threads, Cont.  
Creating a Mutex

- On Linux the attribute can be either fast, recursive, or error checking.
- The default mutex kind is fast. This can be changed by `pthread_mutex_init()` or by calling `pthread_mutexattr_settype()`.
- If the mutex is already locked by the calling thread, the behavior of the lock call depends on the mutex attribute.
Lecture 22: Threads, Cont.
Creating a Mutex

- If the mutex is of the *fast* kind a lock call will **block** (causing the thread to deadlock).

- If it is the *recursive* kind then a lock call will succeed. The number of times the thread has locked the mutex is recorded and an equal number of unlock calls must occur to unlock the mutex.

- If it is the *error checking* kind, the call returns immediately with the error code EDEADLK.
A thread can lock a mutex by calling `pthread_mutex_lock()`. If the mutex is unlocked, it becomes locked and owned by the calling thread, and the call returns immediately. If the mutex is already locked by another thread, the calling thread is suspended until the mutex is unlocked.

A call to `pthread_mutex_trylock()` locks the mutex if it is available or returns with the error code EBUSY if not.
A thread can unlock a mutex by calling `pthread_mutex_unlock()`. (See the man pages for differences in behavior depending on the mutex attribute.)

`pthread_mutex_destroy()` destroys a mutex freeing up any resources it might hold.
Condition variables provide another way for threads to synchronize. While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.
Without condition variables, the programmer would need to have threads continually polling, to check if the condition is met. This can be very resource consuming since the thread would be continuously busy in this activity. A condition variable is a way to achieve the same goal without polling.

A condition variable is always used in conjunction with a mutex lock.
Lecture 22: Threads, Cont.
Condition Variables

- Condition variables must be declared with type `pthread_cond_t`, and must be initialized before they can be used. There are two ways to initialize a condition variable. **Statically**, when it is declared:
  ```c
  pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;
  ```
Lecture 22: Threads, Cont.  
Condition Variables

- Dynamically, with the `pthread_cond_init()` routine. The ID of the created condition variable is returned to the calling thread through the condition parameter.

```c
int pthread_cond_init(
    pthread_cond_t *cond,
    pthread_condattr_t *attr);
```

- The `attr` argument should be NULL for the default attributes.
• The thread waiting on a condition should (1) lock the corresponding mutex, (2) check the value of the variable to make sure that the condition has not yet been met, (3) call `pthread_cond_wait()` to wait on the condition.

• `pthread_cond_wait()` releases the lock and waits (sleeps) for the condition variable to be signaled. `pthread_cond_wait()` reacquires the lock before returning.
Lecture 22: Threads, Cont.  
Condition Variables

- Other threads use `pthread_cond_signal()` or `pthread_cond_broadcast()` to signal waiting threads of a change in the condition.

- `pthread_cond_signal()` wakes up exactly one thread. `pthread_cond_broadcast()` wakes up all waiting threads.

- These calls should be made only after the mutex is locked and must unlock the mutex after (so that `pthread_cond_wait()` to return)
The `condvar.cpp` program demonstrates the use of a condition variable. One thread is waiting for the global counter to reach 12. Two other threads are incrementing the counter. One of the incrementing threads will signal the first thread when the counter reaches 12.
You can use `pthread_kill()` to send a signal to a specific thread. `pthread_sigmask()` can be used to specify a set of signals that should be blocked (ignored) by the calling thread.

The `sigwait()` routine can be used by a thread to wait on a specific signal.