Assignment 9 (?):

Using POSIX Threads
Recitation for ECES 338
April 12th, 1999

Overview of Threads (vs. Processes)

```
+ lighter weight
+ same memory space (no goopy IPC)
+ robust scheduling (if system allows it)
- Confused implementation (user vs. kernel)
- Many variations, worsening implementation.
- Doesn’t support multiprocessor environments as well as processes.
- Cannot exec().
```

Welcome to POSIX

- The purpose behind POSIX is to define a set of specifications for compatibility between different UNIX-class operating systems. Thus, code that is written in order to adhere to the POSIX specifications should be useable on any operating system that is POSIX compliant. In terms of threads, each operating system has chosen to implement threads in their own way. However, they all support the POSIX standard, and have created an environment whereby the POSIX calls can actually manipulate the system-specific facilities. Therefore, if you learn the POSIX method, you can write programs for any compatible operating system.

- As a general note, be sure to `#define _REENTRANT`, and then `#include <pthread.h>`.

- Furthermore, although I haven’t seen the assignment yet, it’s likely that you shouldn't need to mess with attributes at all.

POSIX Thread Functions

- `int pthread_create (pthread_t *new_thread_ID, const pthread_attr_t *attr, void *(*start_func) (void *), void *arg)`
  - In general, this function will create a new thread of execution, and execute a given (presumably) local function in this thread. The first parameter represents a *“Thread Identifier”*. Basically, this is a process-wide unique integer, that references the newly-created thread. You should simply declare a `pthread_t` variable in your program, and pass a reference to this variable to the function call. The next parameter is a pointer to a pthread attribute structure, which can be used to modify the thread attributes at creation. Generally, you will want to make this `NULL`. The third parameter is a void pointer to a function, that you wish to execute in a new thread. This function should be coded so that it returns a void pointer, and accepts a void pointer as a parameter. The fourth (and final) parameter to `pthread_create()` is a pointer to an argument that is to be passed to the

aforementioned function. If you desire more than one argument, you should create a structure, and pass a pointer to that.

- `void pthread_exit (void *status)`
  - This function simply exits a thread. You can give it a pointer to a variable that contains the exit status that you wish to pass on to the parent.

- `int pthread_join (pthread_t target_thread, void **status)`
  - Essentially, this is the thread-equivalent of `wait()`. So, this system call will force the parent process to wait for the termination of a non-detached thread. The first parameter should be a *"Thread Identifier"*, as obtained in the `pthread_create()` system call. The second parameter should be a pointer to a region of memory suitable in order hold the return status (as specified in `pthread_exit()`) of the thread. It should be noted that only one thread of execution can successfully join with another thread – if multiple threads do so, then the extraneous `pthread_join()` attempts will fail. Also, when a non-detached thread exits, it's process-level resources will remain intact until it is joined, or until the parent-level process exits.

- `int pthread_detach (pthread_t threadID)`
  - This system call detaches a thread, such that when it finishes, its resources will be returned to the system *automagically*. The only parameter to this function is a *"Thread Identifier"*.

POSIX Mutex Functions

A mutex is conceptually similar to a binary semaphore – except that a value of zero represents an unlocked (un-owned) state, and a non-zero value represents a locked (owned) state. It is also strongly recommended that a mutex is only unlocked by its owner.

For the purposes of this lecture, I will only discuss mutexes that function in an intra-process threaded environment. Sharing a mutex between several threaded processes (inter-process model) is a more advanced topic.

Creating a mutex is as simple as declaring a variable in your program, like this:

```
pthread_mutex_t my_lock1;
```

Initializing this mutex can be handled by setting it equal to the constant `PTHREAD_MUTEX_INITIALIZER`, or via a system call:

- `int pthread_mutex_init (pthread_mutex_t *mp, const pthread_mutexattr_t *attr)`
  - This function initializes a given mutex with the attributes listed in the second parameter. If a NULL pointer is given as the second parameter, the mutex will receive the default attributes – which is probably good enough for the intra-process environment.

- `int pthread_mutex_lock (pthread_mutex_t *mp)`
  - This system call attempts to lock a mutex (the equivalent of `wait()`). If the mutex is currently unlocked, this call will lock it by taking ownership of the mutex. If the mutex is already locked, then this system call will cause the calling thread to block. As an aside, if a thread that owns a mutex issues a second `pthread_mutex_lock()` call on the same mutex, then deadlock will result.

- `int pthread_mutex_unlock (pthread_mutex_t *mp)`
  - This system call attempts to unlock a mutex (the equivalent of `signal()`). In order to avoid undefined behavior, only the thread that owns a locked mutex should unlock it.

- `int pthread_mutex_trylock (pthread_mutex_t *mp)`
  - This function is conceptually similar to `pthread_mutex_lock()`, except that it will not block the thread if the mutex is locked. Instead, it will return `EBUSY` (16), indicating that the mutex is locked. In this manner, a thread can determine the state of a mutex.

- `int pthread_mutex_destroy (pthread_mutex_t *mp)`
  - This system call removes the reference to a given mutex from the system. Note, however, that it doesn't de-allocate any memory occupied by the mutex variable, it only removes this mutex from the system tables.

POSIX Condition Variables

All implementations of POSIX threads include an implementation of Condition Variables. In essence, condition variables are useful in an environment where a thread has locked a critical section via a mutex, but during the course of the execution of this critical section, determines that it needs to wait for some other condition to be true. Therefore, pthread condition variables are the "other" synchronization element – they allow a synchronization method that interacts with a more global mutex.

In order to make use of a condition variable, it can be declared in a manner that is similar to a mutex:

```
pthread_cond_t my_condition = PTHREAD_COND_INITIALIZER;
```

- `int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex)`
  - This function will cause a calling thread to block if the given condition is not conducive to continuation. The first parameter is a pointer to a condition variable, the second parameter is a pointer to a mutex variable. If this function causes the calling thread to block, it will first release the mutex, and then go to sleep.

- `int pthread_cond_signal (ptrhead_cond_t *cond)`
  - This system call notifies a waiting thread that a given condition variable is ready for further processing. It will wake up a waiting thread, and atomically regain the mutex, so that the previously waiting thread can continue processing.