Synchronization: Locks and Condition Variables

Harrison Vuong (hvuong@ucsc.edu)
David Zou (dzou@ucsc.edu)
Derek Frank (dmfrank@ucsc.edu)

1 Goal:

The goal of this assignment is to use the created Semaphores from the first part of the Synchronization project to implement locks and condition variables.

2 Available Resources:

Minix Documentations
Interprocess Communications and Synchronization Slides
Operating System Textbook

3 Design:

***file location instructions are located in the README*

void lock_init (struct lock *l)
Initializes a lock with default values for its semaphores and counter

```
    Initialize the semaphore for mutual exclusion (mutex)
    Initialize the semaphore for next high-priority process (next)
    Initialize the counter to keep track of the high priority processes (nextcount)
```

void lock_acquire (struct lock *l)
Acquires the lock in order to access the critical region

```
    Make sure the specified lock is already defined; if not, exit the program
    Down the 'mutex' semaphore to allow the process to enter the critical region
```

void lock_release (struct lock *l)
Releases the lock to let other processes get into the critical region

```
    Make sure the specific lock is already defined; if not, exit the program
    If the processes waiting on the current process
        Decrement the counter keeping track of the waiting processes by 1
    Else,
        Up the 'mutex' semaphore to allow the next process to enter the critical region
```

void cond_init (struct cond *cnd, struct lock *l)
Initializes a condition variable with default values for its lock, semaphore, and counter

```
    Make sure both the specified lock and condition variable are already defined; if not, exit the program
    Initialize the condition variable's lock to 'l'
    Intialize the condition variable's semaphore with a value of 0 and a random id (condSem)
    Initialize the counter that keeps track of how many processes are waiting to 0 (semCount)
```

void cond_wait (struct cond *cnd)
Tells a process/thread to wait on a certain condition (cnd)

```
    Make sure the specified condition variable is already defined; if not, exit the program.
    Increment the counter that keeps track of how many processes are waiting to by 1 (semCount)
    Release the lock for the process and call Down on the condition variable semaphore
    If there processes that are sharing the current lock
        Call down on the semaphore for the next process (next)
        Decrement the counter keeping track of the high priority processes by 1 (nextCount)
    Decrement the counter keeping track of the number of waiting processes by 1 (semCount)
```

void cond_signal (struct cond *cnd)
The process/thread that calls this method signals a certain condition, which wakes up processes waiting for specific condition.

```
    Make sure the specified condition variable is already defined; if not, exit the program.
    If there are processes waiting for the condition
        Increment the counter keeping track of high priority processes by 1 (nextCount)
        Call Up on the condition variable's semaphore to wake up the next waiting process
```

4 Testing:

Although we did not have the time to test out the locks and condition variables, we intended to create a program that uses threading (most likely using POSIX threads) and in each thread, we would call locks to manipulate a simple variable mutual exclusively (like decrementing/incrementing an integer within the critical region). As a whole, condition variables would be used for monitoring the locks for certain conditions to happen, if there are any. All this would be done inside the threads. Another way would be to run the same program as background processes and have one or more program trigger several conditions and checking if the data in the critical region changes after doing so.