COMP 3413 Operating Systems Lab 3 Hand–in

Family Name	Given Name	Mark:
Pontes	Joao	20

Student Number

[2] 1a. Initializing a pthread mutex variable: You can initialize a pthread mutex variable by either using "int pthread_mutex_init()" or "pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER". The prior is used when the mutex is required to be recursive or used between processes. The latter is only able to be used the statically defined variables.

[2] 1b. `pthread_mutex_destroy()`: A destroyed mutex object can be reinitialized using "pthread_mutex_init()". Only unlocked mutex objects can be destroyed. Otherwise, if the mutex is locked, then destroying it results in undefined behavior.

[3] 1c. `pthread_mutex_trylock()`: "trylock()" behaves the same as "lock()", except if the mutex object which is referenced is already locked. If the object is already locked then "trylock()" will return an error number, as opposed to "lock()" which blocks until the mutex becomes available. "trylock()" is useful if you wish to do other operations while waiting for the mutex to be unlocked.

[3] 2a. `mutex_recursive.c`: I get the output –

```
Testing recursive mutex.
locking
 locking
```

What is happening – The program locks the mutex a in the first loop. When the program loops a second time, the mutex locks again. What happens in the second loop is that the thread is waiting for the first lock to unlock before proceeding. But since the first lock is never getting unlocked, because "unlock()" is in a separate loop, the program gets stuck. This behavior in the second loop is known as a deadlock.

[3] 2b. modified `mutex_recursive.c`: To fix this, I changed the mutex type to "PTHREAD_MUTEX_RECURSIVE". This now gives us the following output –

```
Testing recursive mutex.
locking
 locking
  locking
   locking
    locking
     locking
      locking
       locking
        locking
         locking
         unlocking
        unlocking
       unlocking
      unlocking
     unlocking
    unlocking
   unlocking
  unlocking
 unlocking
unlocking

End of processing
```

This behavior is useful when you wish to use locks to prevent simultaneous resource usage, but want to avoid deadlocking.

[1] 3a. expected value of shared variable: If all threads perform "thrfunc()" completely, then the shared value should be "`shared = MAX*NUM_THREADS`" which should be 1000 (e.g. 20*50).

[1] 3b. results of 10 runs before changes: After running the program 10 times I recorded the following results for the shared value – 106, 198, 148, 108, 125, 197, 175, 149, 184, 241. What I believe is happening is that the threads are overlapping when they are trying to use the shared resource. The result is that none of the threads are taking the full time to perform the operations, they use stale results from previous threads, and the program is ending prematurely.

[5] 3c. results of 10 runs after changes: After adding the mutex, the results for all 10 tests for the shared variable was 1000. Which corresponds to my predicted result from 3a.