Operating Systems
Homework 3
Chad Weigle

1.

Main Thread	Thread 1	Thread 2
Create Thread 1	Created	-
Create Thread 2	-	Created
-	C1(1) = 1 – C2(1), C1 = 0	-
-	-	C2(1) = 1 – C1(0), C2 = 1
-	Critical Section	-
-	-	Iteration, no change
-	C1 = 1	-
-	-	C2(1) = 1 – C1 (1), C2 = 0
-	Iteration, no change	-
-	-	Critical Section
-	Iteration, no change	-
-	-	C2 = 1
-	C1(1) = 1 – C2(1), C1 = 0	-
-	-	Iteration, no change
-	Critical Section	-
-	-	Iteration, no change
-	C1 = 1	-

The above tables shows the chain of events of each thread according to the code provided. If we assume the time management is even between thread 1 and thread 2 and neither thread fails outside of its critical section, then there should be no reason for any kind of mutual exclusion problem to exist. However, it is possible that one thread is editing one of the shared variables while the other is checking the same shared variable, thus causing a mutual exclusion problem. AND on top of the previous error, if both threads were to execute the first loop at the same time, they would both get through to the critical section also causing a mutual exclusion problem.

2.

```
quicksort_threaded(A,p,r)
1. if(p < r)
2. then lockMutex(mutex)
3.       q ←partition(A,p,r)
4.       unlockMutex(mutex)
5.       createThread( quicksort(A, p, q-1) )
6.       createThread( quicksort(A, q+1, r) )
7.       waitForChildThreads(pids)
```

3.
This solution is mutually exclusive because no philosophers will grab for the same fork. This is assured by the waiting on the semaphores i and i+1. Also, no deadlocks will occur because a post on the semaphores occurs right after the eating as finished. If any philosopher were to fail before posting on a semaphore, a deadlock would occur because both semaphores (forks) would remain locked. This means the program is not loosely connected.