Quiz 3, CSCI U511, Fall 2019 **Name: _____Key_____**

Instructions: You must show your work and put your final answers in the blanks. If you round a numerical answer, you must give at least 3 significant digits.

1)

```
#define NTHREADS 10
thread_t threads[NTHREADS];
main() {
    for (i = 0; i < NTHREADS; i++)
thread_create(&threads[i], &go, i);
    for (i = 0; i < NTHREADS; i++) {
        exitValue = thread_join(threads[i]);
        printf("Thread %d returned with %ld\n", i,
exitValue);
    }
    printf("Main thread done.\n");
    return 0;
}
void go (int n) {
    printf("Hello from thread %d\n", n);
    thread_exit(100 + n);
    // REACHED?
  }
```

For the above code fragments, answer the followings:

(a) The procedure go() has the parameter np and the local variable n. Are these variables *per-thread* or *shared state*? Where does the compiler store these variables' states?

ANS: Note that the threadHello program has a parameter but no local variables. However, if it did: Parameters and local variables in a forked thread are per-thread state—different threads can have different values for these variables. The compiler would typically store these variables on the thread's stack or in the processor registers.

(b) Suppose that we delete the second for loop so that the main routine simply creates NTHREADS threads and then prints "Main thread done." What are the possible outputs of the program now?

ANS: "Main thread done." Could display at any point when the other threads are printing. Since, we are no longer waiting for the thread to exit.

2) Why condition variable 'wait' must always be called from within a loop, like the following code fragment? Explain.

```
    ...
while (predicateOnStateVariables(...)) {
     wait(&lock);
    }
    ...
    and not:
          ...
          if(predicateOnStateVariables(...)) {
               wait(&lock);
          }
    …
```

ANS: wait release the lock, and there is no guarantee of atomicity between signal or broadcast and the return of a call to wait, there is no guarantee that the checked-for state still holds. Therefore a wait thread must always wait in a loop.

3) Explain, for the following code fragments, why is it essential to use the same lock in both procedures?

```
char *malloc (int n) {
    char *p;

    heaplock.acquire();
 // Code for single-threaded malloc()
 // p = allocate block of memory
 //     of size n.
    heaplock.release();
    return p;
}
```

```
void free (char *p) {

    heaplock.acquire();
 // Code for single-threaded free()
 // Put p back on free list.

    heaplock.release();

}
```

ANS: Since, malloc and free read and modify the same data structures, it is essential to use the same lock in both procedures, heaplock.