Homework for Thread API

1. **Instruction Run**: valgrind –tool=helgrind ./main-race

   (By the way the instruction is run, it is clear that *valrind* is the name of a suite of test tools, and *helgrind* is a *sub-tool* within this suite).

2. The race condition is obvious: balance++ can be incremented by either the worker or the main thread.

   Now as for helgrind: it seems to work by first running the program, with a series of stdout declarations to the terminal marked as "Thread Announcement." It does this for each thread created, in the order they were created, and seems to list some memory address next to each executed instruction (perhaps the location of the instruction in memory...need to look into this).

   It subsequently reports two data races (which makes sense...as there are two threads that make up the race) one by thread #1 and one by thread #2. For each of these it gives a stack trace, followed by the conflicting data access variable (balance). Note it even gives the size of the data subjected to the race condition: in this case, 4 bytes. So yes, it does give both offending lines (8 and 15) in the code.

3. **Instruction Run**: valgrind –tool=helgrind ./main-race

   Now after the above is run with the "balance++" commented out in the worker function, it doesn't even report the creation of the threads, which is interesting. So not only are the error messages removed, basically all of the output is removed as well.

   Next, we uncomment the offending line in the worker function, and instead add a lock around both instances of balance. The result is the same as if we removed one of the offending lines: no error reported.

4. No instruction was run here, just examined the code. The deadlock occurs because of the ordering of the locking/unlocking of the mutex: the first created thread locks on m1, the second created thread on m2, and the main thread waits for the first thread it created to join. Every thread gets stuck and the program dead-locks.

5.

6.

7.

8.

9.