1. First build main-race.c. Examine the code so you can see the (hopefully obvious) data race in the code. Now run helgrind(by typing valgrind--tool=helgrind main-race) to see how it reports the race. Does it point to the right lines of code? What other information does it give to you?
   - Yes, it points to the right lines.
   - It gives us the name, address and size of the variable that is being affected (balance), and the function that is acting upon the variable (worker).

2. What happens when you remove one of the offending lines of code? Now add a lock around one of the updates to the shared variable, and then around both. What does helgrind report in each of these cases?
   - If we remove line 14, valgrind does not show any reports of data races.
   - Adding a lock in the function “worker” removed the issues with the data race.
   - Adding an extra lock around the variable “balance” on the main also makes valgrind to report an empty summary of problems.

3. Now let’s look at main-deadlock.c. Examine the code. This code has a problem known as deadlock(which we discuss in much more depth in a forthcoming chapter). Can you see what problem it might have?
   - When we are trying to wait for the threads to finish, both are waiting on each other to release the lock.

4. Now run helgrind on this code. What does helgrind report?
   - It reports when each thread got a hold of which lock (m1 or m2), as well as when the order to acquire the lock was sent, the order in which they were violated.

5. Now run helgrind on main-deadlock-global.c. Examine the code; does it have the same problem that main-deadlock.c has? Should helgrind be reporting the same error? What does this tell you about tools like helgrind?
   - Yes, it has the same issues as main-deadlock.c the only difference is that there is a lock at the very beginning of the function, which should prevent any data race issues, but the remaining 2 locks still cause problems.
   - No, it shouldn’t
   - It says that it’s not perfect at detecting tricky cases.

6. Let’s next look at main-signal.c. This code uses a variable (done) to signal that the child is done and that the parent can now continue. Why is this code inefficient? (what does the parent end up spending its time doing, particularly if the child thread takes a long time to complete?)
   - The parent falls into the while loop and spends its time just spinning, waiting for the child to complete.

7. Now run helgrind on this program. What does it report? Is the code correct?
   - It reports that there is a possible data race when writing to the variable “done”, but the code is correct, the first line to be printed is printed, and the last line is printed at the end.