Question 1

First build main-race.c. Examine the code so you can see the (hopefully obvious) data race in the code.
Now run helgrind
`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?

It does point to the right lines of code.
Additionally it tells us when a new thread is created and which thread might cause which error

```
==1649602== Possible data race during read of size 4 at 0x10C014 by thread #1
==1649602== Locks held: none
==1649602==    at 0x109236: main (main-race.c:15)
==1649602==
==1649602== This conflicts with a previous write of size 4 by thread #2
```

Question 2

`valgrind --tool=helgrind ./main-race`
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?

With the offending lines removed, helgrind doesn't complain anymore.
When adding both locks helgrind doesn't complain either.

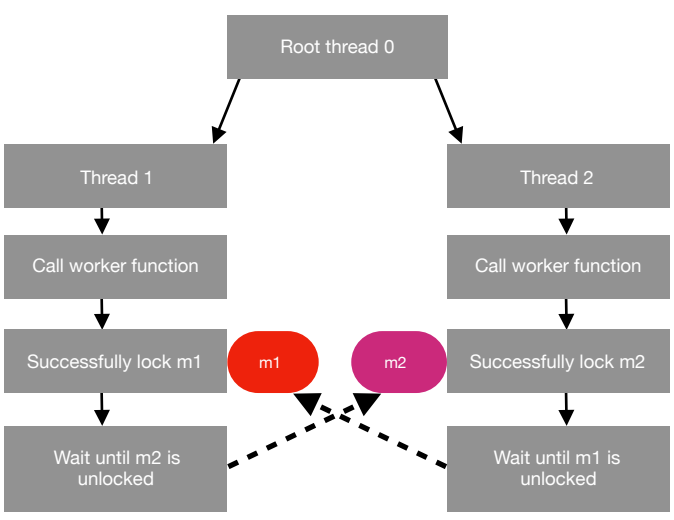
```c
#include <stdlib.h>
//...
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
//...
Pthread_mutex_lock(&lock);
balance++; // unprotected access
Pthread_mutex_unlock(&lock);
```

Question 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?

Thread T1 executing worker function
locks m1 successfully.
Thread T2 executing worker function
locks m2 successfully
Thread T1 tries to lock m2, but it is already locked by T2.
wait for T2
Thread T2 tries to lock m1, but it is already locked by T1.
wait for T1

Like this, each thread is waiting for the other to finish and unlock the needed lock.



Question 4

Now run helgrind on this code. What does helgrind report?
`valgrind --tool=helgrind ./main-deadlock`

```
==1654875== Thread #3: lock order "0x10C040 before 0x10C080" violated
==1654875== Observed (incorrect) order is: acquisition of lock at 0x10C080
==1654875==  followed by a later acquisition of lock at 0x10C040
==1654875== Required order was established by acquisition of lock at 0x10C040
==1654875==  followed by a later acquisition of lock at 0x10C080
```

Question 5

`valgrind --tool=helgrind ./main-deadlock-global`
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?

Since we now have a lock for the entire worker function, only one thread at a time is able to use the function, thus a deadlock is not possible.
However helgrind still thinks we have a deadlock thus making the tool basically useless since you can never actually trust it.

Question 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 main thread constantly checks if the 'done' variable is set to 0 and then exits. This is really inefficient and can slow down the CPU

Question 7

`valgrind --tool=helgrind ./main-signal`
Now run helgrind on this program.
What does it report?
Is the code correct?

Helgrind still reports a possible data race. This is because both threads access the same global variable at the same time. This is not an issue since only one thread is able to modify the variable.

Question 8

Now look at a slightly modified version of the code, which is found in main-signal-cv.c. This version uses a condition variable to do the sig- naling (and associated lock).
Why is this code preferred to the previous version?
Is it correctness, or performance, or both?

The only difference seems to be that the main thread sleeps until the other thread is done. This seems to only be a performance increase

Question 9

`valgrind --tool=helgrind ./main-signal-cv`
Once again run helgrind on main-signal-cv. Does it report any errors?

Since we don't actually check the 'done' variable in the main thread we don't get any errors