**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?**

It points to the address that race happen. It also telling us which threads are trying to access the critical section. It also tells us is there exist a lock on the critical section, and size of the critical variable.

```
==46591== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==46591== Command: ./main-race
==46591==
==46591== ---Thread-Announcement------------------------------------------
==46591==
==46591== Thread #1 is the program's root thread
==46591==
==46591== ---Thread-Announcement------------------------------------------
==46591==
==46591== Thread #2 was created
==46591==    at 0x4975CF7: clone (clone.S:62)
==46591==    by 0x490D1EB: create_thread (pthread_create.c:295)
==46591==    by 0x490DC0F: pthread_create@@GLIBC_2.34 (pthread_create.c:828)
==46591==    by 0x486F24B: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-arm64-linux.so)
==46591==    by 0x108943: main (main-race.c:17)
==46591==
==46591== ----------------------------------------------------------------
==46591==
==46591== Possible data race during read of size 4 at 0x119014 by thread #1
==46591== Locks held: none
==46591==    at 0x108974: main (main-race.c:19)
==46591==
==46591== This conflicts with a previous write of size 4 by thread #2
==46591== Locks held: none
==46591==    at 0x1088F4: worker (main-race.c:10)
==46591==    by 0x486F41B: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-arm64-linux.so)
==46591==    by 0x490D5C7: start_thread (pthread_create.c:442)
==46591==    by 0x4975D1B: thread_start (clone.S:79)
==46591==  Address 0x119014 is 0 bytes inside data symbol "balance"
==46591==
==46591== ----------------------------------------------------------------
==46591==
==46591== Possible data race during write of size 4 at 0x119014 by thread #1
==46591== Locks held: none
==46591==    at 0x108984: main (main-race.c:19)
==46591==
==46591== This conflicts with a previous write of size 4 by thread #2
==46591== Locks held: none
==46591==    at 0x1088F4: worker (main-race.c:10)
==46591==    by 0x486F41B: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-arm64-linux.so)
==46591==    by 0x490D5C7: start_thread (pthread_create.c:442)
==46591==    by 0x4975D1B: thread_start (clone.S:79)
==46591==  Address 0x119014 is 0 bytes inside data symbol "balance"
==46591==
==46591==
==46591== Use --history-level=approx or =none to gain increased speed, at
==46591== the cost of reduced accuracy of conflicting-access information
==46591== For lists of detected and suppressed errors, rerun with: -s
==46591== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

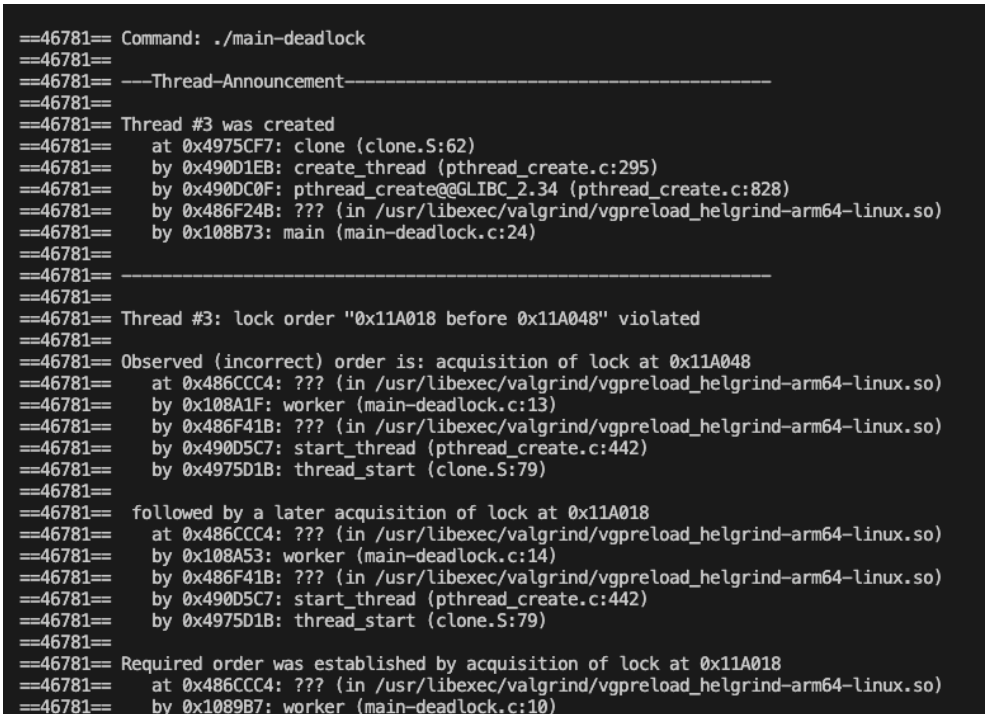
**2.What happens when you remove one of the offending lines of code? Now add a lock around one of the updates to the shared variable, and then around both. What does helgrind report in each of these cases?**

If we remove the one of the offending line, the errors also gone.
Once we add lock to both critical section, all the errors are gone.

**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?**

In a situation p1 just have lock the m1, then immediately switch to p2, then p2 lock the m2. Then the next line for p2 is to lock the m2, but m2 in p1's hand. For p1 the next line is to lock lock the m2, but the m2 in p2's hand. In this case the both thread will into infinity loop.

**4.Now run helgrind on this code. What does helgrind report?**



Helgrind detected the incorrect order.

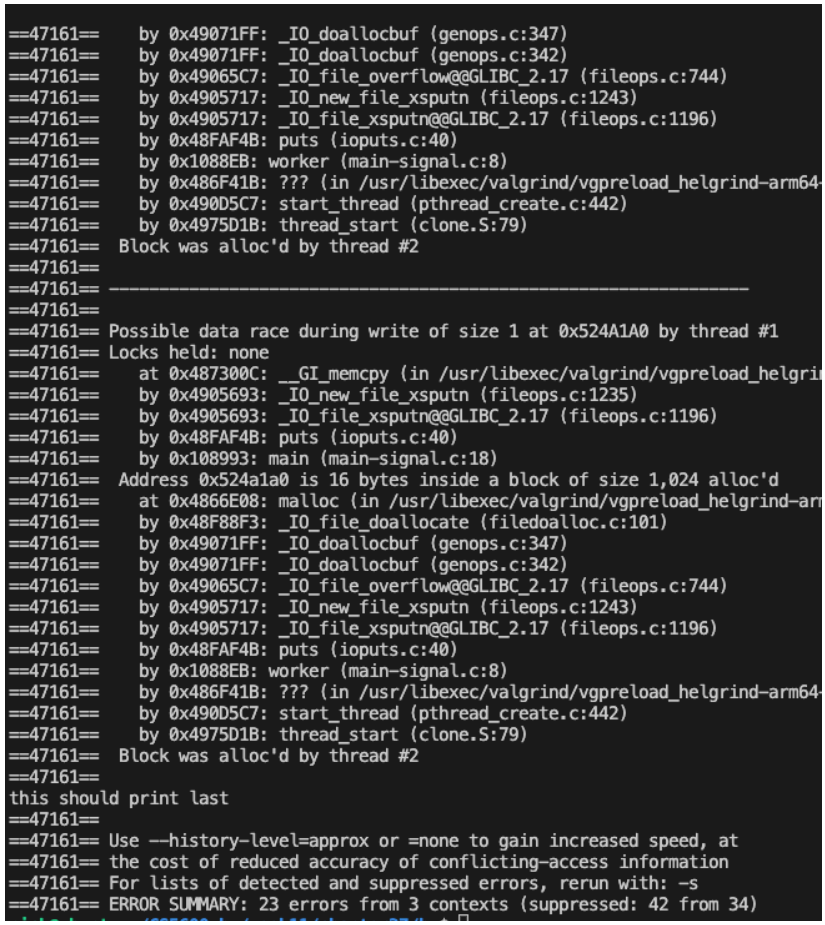
**5. Now run helgrind on main-deadlock-global.c. Examine the code; does it have the same problem that main-deadlock.c has? Should helgrind be reporting the same error? What does this tell you about tools like helgrind?**

It shouldn't have same problem that main-deadlock.c has, but the helgrind still report the same error. The reason for that is helgrind only check the sequence when the lock happen.

**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?)**

This code is inefficient because instead let thread go to sleep to wait for other thread complete, it is actual spinning in the while loop waste CPU time.

**7.Now run helgrind on this program. What does it report? Is the code correct?**



It report 23 errors, and the race condition become variable done.

**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 signaling (and associated lock). Why is this code preferred to the previous version? Is it correctness, or performance, or both?**

Both, the performance is much better, the thread won't spinning on a while loop instead into the sleep. It is also a correct way to do the multi threading.

**9. Once again run helgrind on main-signal-cv. Does it report any errors?**

There is no error at all.