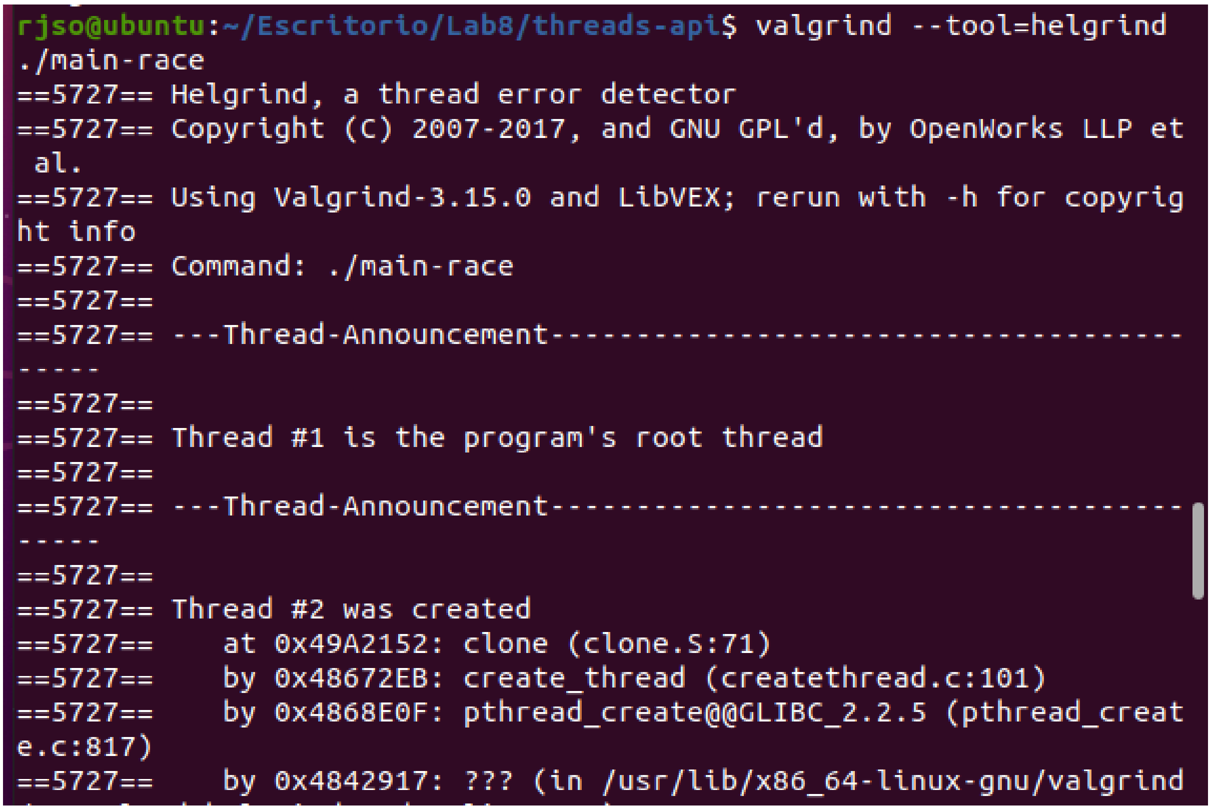
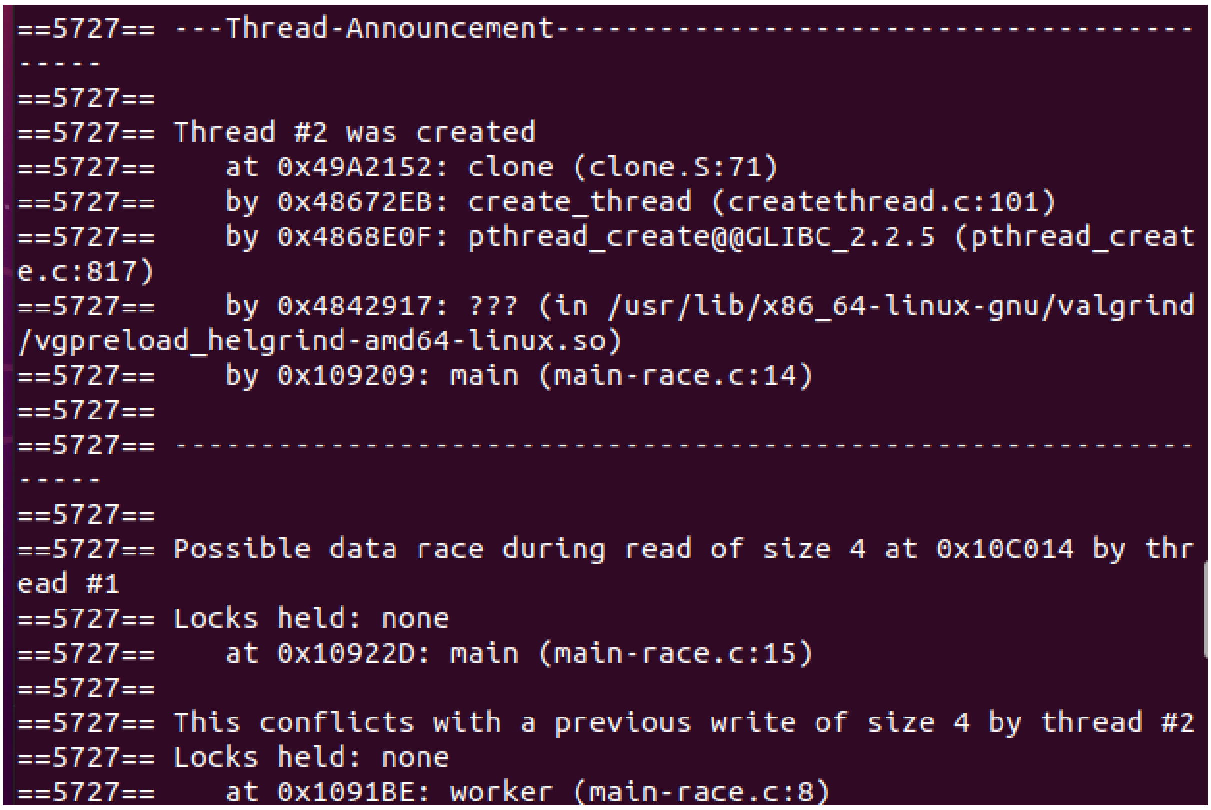
Rebeca Servellón Orellana
Nestor Pasos Rocha
José Arturo García Rodríguez

Homework (Code)
In this section, we'll write some simple multi-threaded programs and use a specific tool, called helgrind, to find problems in these programs. Read the README in the homework download for details on how to build the programs and run helgrind.
Questions

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?



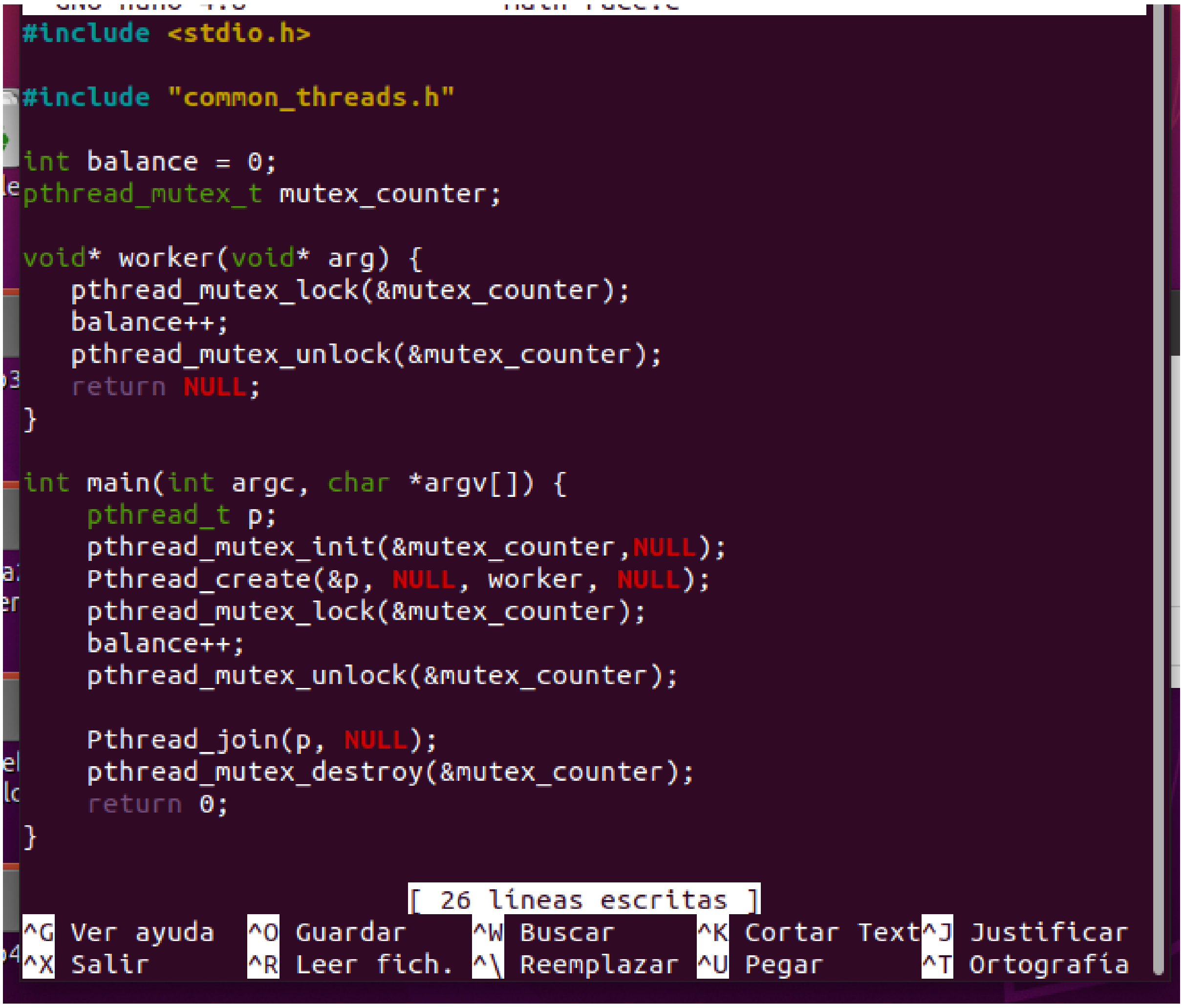
```
rjso@ubuntu:~/Escritorio/Lab8/threads-api$ valgrind --tool=helgrind
./main-race
==5727== Helgrind, a thread error detector
==5727== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et
 al.
==5727== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyrig
ht info
==5727== Command: ./main-race
==5727==
==5727== ---Thread-Announcement-------------------------------------
-----
==5727==
==5727== Thread #1 is the program's root thread
==5727==
==5727== ---Thread-Announcement-------------------------------------
-----
==5727==
==5727== Thread #2 was created
==5727==    at 0x49A2152: clone (clone.S:71)
==5727==    by 0x48672EB: create_thread (createthread.c:101)
==5727==    by 0x4868E0F: pthread_create@@GLIBC_2.2.5 (pthread_creat
e.c:817)
==5727==    by 0x4842917: ??? (in /usr/lib/x86_64-linux-gnu/valgrind
```



```
==5727== ---Thread-Announcement------------------------------------------
-----
==5727==
==5727== Thread #2 was created
==5727==    at 0x49A2152: clone (clone.S:71)
==5727==    by 0x48672EB: create_thread (createthread.c:101)
==5727==    by 0x4868E0F: pthread_create@@GLIBC_2.2.5 (pthread_creat
e.c:817)
==5727==    by 0x4842917: ??? (in /usr/lib/x86_64-linux-gnu/valgrind
/vgpreload_helgrind-amd64-linux.so)
==5727==    by 0x109209: main (main-race.c:14)
==5727==
==5727== ----------------------------------------------------------------
-----
==5727==
==5727== Possible data race during read of size 4 at 0x10C014 by thr
ead #1
==5727== Locks held: none
==5727==    at 0x10922D: main (main-race.c:15)
==5727==
==5727== This conflicts with a previous write of size 4 by thread #2
==5727== Locks held: none
==5727==    at 0x1091BE: worker (main-race.c:8)
```

En el final muestra las dos partes en las que hay una zona crítica, y además que no tiene un método para tratarlas.

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?



```
#include <stdio.h>

#include "common_threads.h"

int balance = 0;
pthread_mutex_t mutex_counter;

void* worker(void* arg) {
   pthread_mutex_lock(&mutex_counter);
   balance++;
   pthread_mutex_unlock(&mutex_counter);
   return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    pthread_mutex_init(&mutex_counter,NULL);
    Pthread_create(&p, NULL, worker, NULL);
    pthread_mutex_lock(&mutex_counter);
    balance++;
    pthread_mutex_unlock(&mutex_counter);

    Pthread_join(p, NULL);
    pthread_mutex_destroy(&mutex_counter);
    return 0;
}

                        [ 26 líneas escritas ]
```

Al modificar el código de la forma que aparece arriba, se puede apreciar que se agregan dos “locks” para los “balance++;”, esto remueve la zona de riesgo. Se puede evidenciar con el siguiente reporte de valgrind:

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?

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

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?

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?)

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

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?

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