Estudiante:Nicole Araya Ballestero

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?

Código:

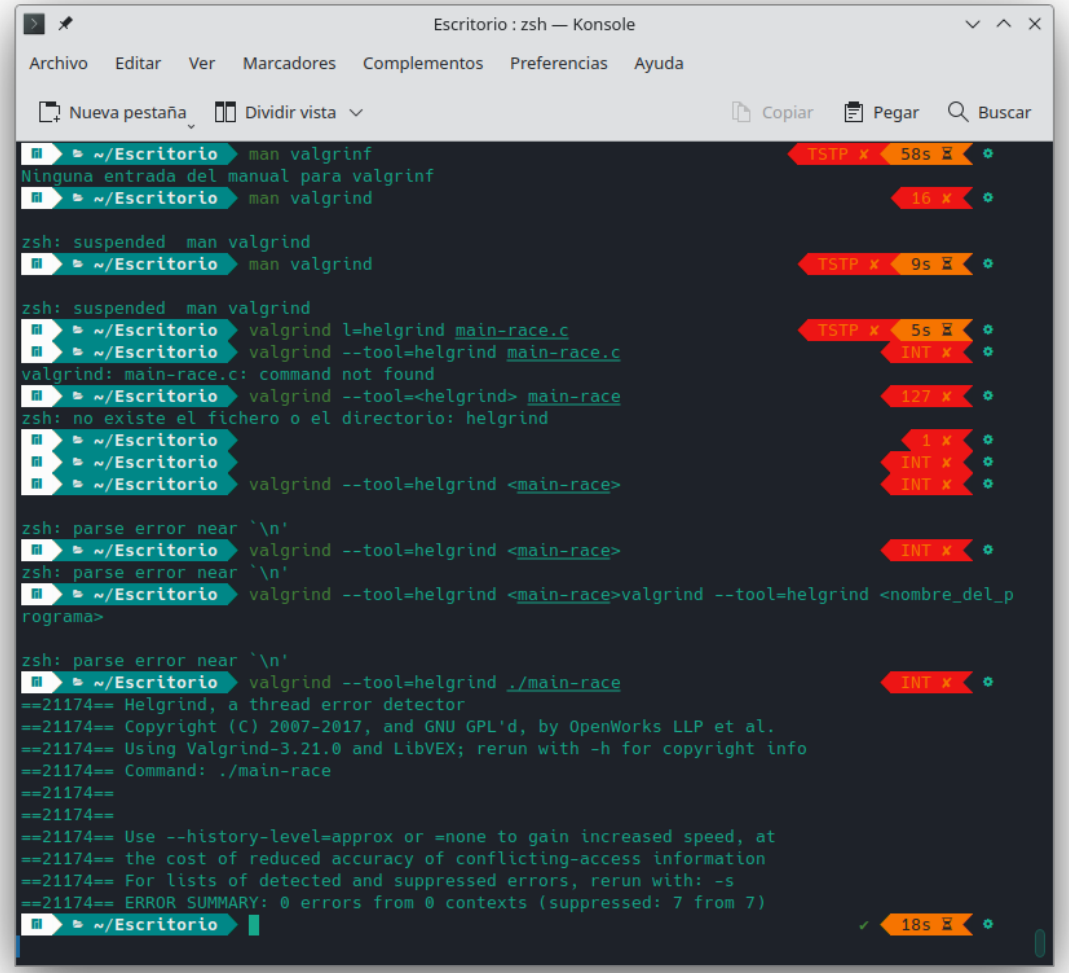
```
#ifndef __MYTHREADS_h__
#define __MYTHREADS_h__

#include <pthread.h>
#include <assert.h>
#include <stdlib.h>
#include <sys/time.h>

double Time_GetSeconds() {
        struct timeval t;
        int rc = gettimeofday(&t, NULL);
        assert(rc == 0);
        return (double) ((double)t.tv_sec + (double)t.tv_usec / 1e6);
}

void Pthread_mutex_init(pthread_mutex_t *mutex,
                const pthread_mutexattr_t *attr) {
        int rc = pthread_mutex_init(mutex, attr);
        assert(rc == 0);
}

void Pthread_mutex_lock(pthread_mutex_t *m) {
        int rc = pthread_mutex_lock(m);
        assert(rc == 0);
}

void Pthread_mutex_unlock(pthread_mutex_t *m) {
        int rc = pthread_mutex_unlock(m);
        assert(rc == 0);
}

void Pthread_cond_init(pthread_cond_t *cond,
```

```
                const pthread_condattr_t *attr) {
        int rc = pthread_cond_init(cond, attr);
        assert(rc == 0);
}

void Pthread_cond_wait(pthread_cond_t *cond,
                pthread_mutex_t *mutex) {
        int rc = pthread_cond_wait(cond, mutex);
        assert(rc == 0);
}

void Pthread_cond_signal(pthread_cond_t *cond) {
        int rc = pthread_cond_signal(cond);
        assert(rc == 0);
}

void Pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                void *(*start_routine)(void*), void *arg) {
        int rc = pthread_create(thread, attr, start_routine, arg);
        assert(rc == 0);
}

void Pthread_join(pthread_t thread, void **value_ptr) {
        int rc = pthread_join(thread, value_ptr);
        assert(rc == 0);
}



#endif // __MYTHREADS_h__
```

—----------

```
#include <stdio.h>
#include <stdlib.h>

#include "mythreads.h"

int balance = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void* worker(void* arg) {
        Pthread_mutex_lock(&lock);
        balance++; // unprotected access
        Pthread_mutex_unlock(&lock);
        return NULL;
}
```

```
int main(int argc, char *argv[]) {
        pthread_t p;
        Pthread_create(&p, NULL, worker, NULL);

        Pthread_mutex_lock(&lock);
        balance++; // unprotected access
        Pthread_mutex_unlock(&lock);

        Pthread_join(p, NULL);
        return 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?

1.Código: Eliminando la línea ofensiva sin protección

```
#include <stdio.h>
#include <stdlib.h>
#include "mythreads.h"

int balance = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void* worker(void* arg) {
        Pthread_mutex_lock(&lock);
        balance++; // unprotected access
        Pthread_mutex_unlock(&lock);
        return NULL;
}

int main(int argc, char *argv[]) {
        pthread_t p;
        Pthread_create(&p, NULL, worker, NULL);

        Pthread_mutex_lock(&lock);
        // balance++; // unprotected access
        Pthread_mutex_unlock(&lock);

        Pthread_join(p, NULL);
        return 0;
}
```

2.Código: Agregando un bloqueo alrededor de una actualización de la variable compartida

```
#include <stdio.h>
#include <stdlib.h>

#include "mythreads.h"

int balance = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void* worker(void* arg) {
        Pthread_mutex_lock(&lock);
        balance++; // unprotected access
        Pthread_mutex_unlock(&lock);
        return NULL;
}

int main(int argc, char *argv[]) {
        pthread_t p;
```

```
	Pthread_create(&p, NULL, worker, NULL);

	Pthread_mutex_lock(&lock);
	balance++; // protected access
	Pthread_mutex_unlock(&lock);

	Pthread_join(p, NULL);
	return 0;
}
```

3.Código: Agregando un bloqueo alrededor de ambas actualizaciones de la variable compartida

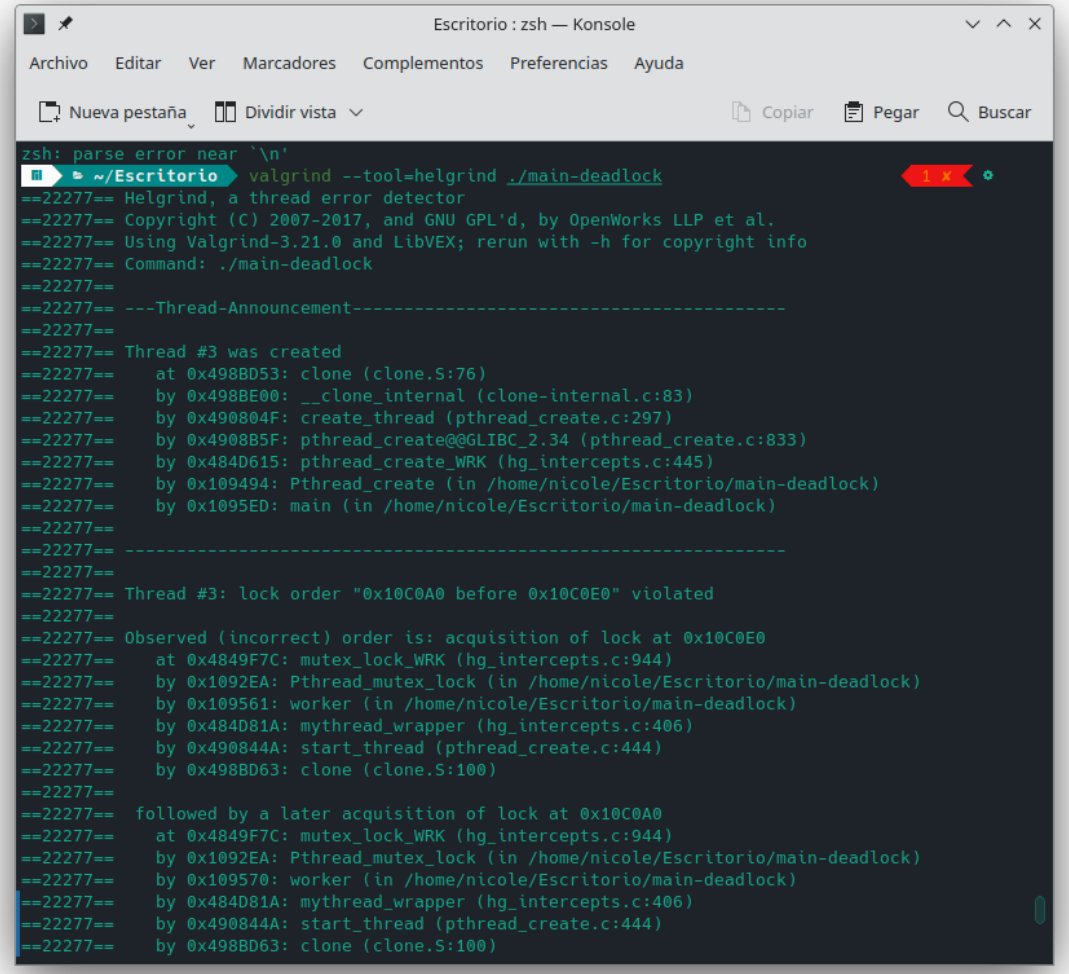
```
#include <stdio.h>
#include <stdlib.h>

#include "mythreads.h"

int balance = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void* worker(void* arg) {
	Pthread_mutex_lock(&lock);
	balance++; // unprotected access
	Pthread_mutex_unlock(&lock);
	return NULL;
}

int main(int argc, char *argv[]) {
	pthread_t p;
	Pthread_create(&p, NULL, worker, NULL);

	Pthread_mutex_lock(&lock);
	balance++; // protected access
	Pthread_mutex_unlock(&lock);

	Pthread_join(p, NULL);
	return 0;
}
```

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?

Codigo:

```
#include <stdio.h>

#include "mythreads.h"

pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m2 = PTHREAD_MUTEX_INITIALIZER;

void* worker(void* arg) {
        if ((long long) arg == 0) {
   Pthread_mutex_lock(&m1);
   Pthread_mutex_lock(&m2);
        } else {
   Pthread_mutex_lock(&m2);
   Pthread_mutex_lock(&m1);
        }
        Pthread_mutex_unlock(&m1);
        Pthread_mutex_unlock(&m2);
        return NULL;
}

int main(int argc, char *argv[]) {
        pthread_t p1, p2;
        Pthread_create(&p1, NULL, worker, (void *) (long long) 0);
        Pthread_create(&p2, NULL, worker, (void *) (long long) 1);
        Pthread_join(p1, NULL);
        Pthread_join(p2, NULL);
        return 0;
}
```

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

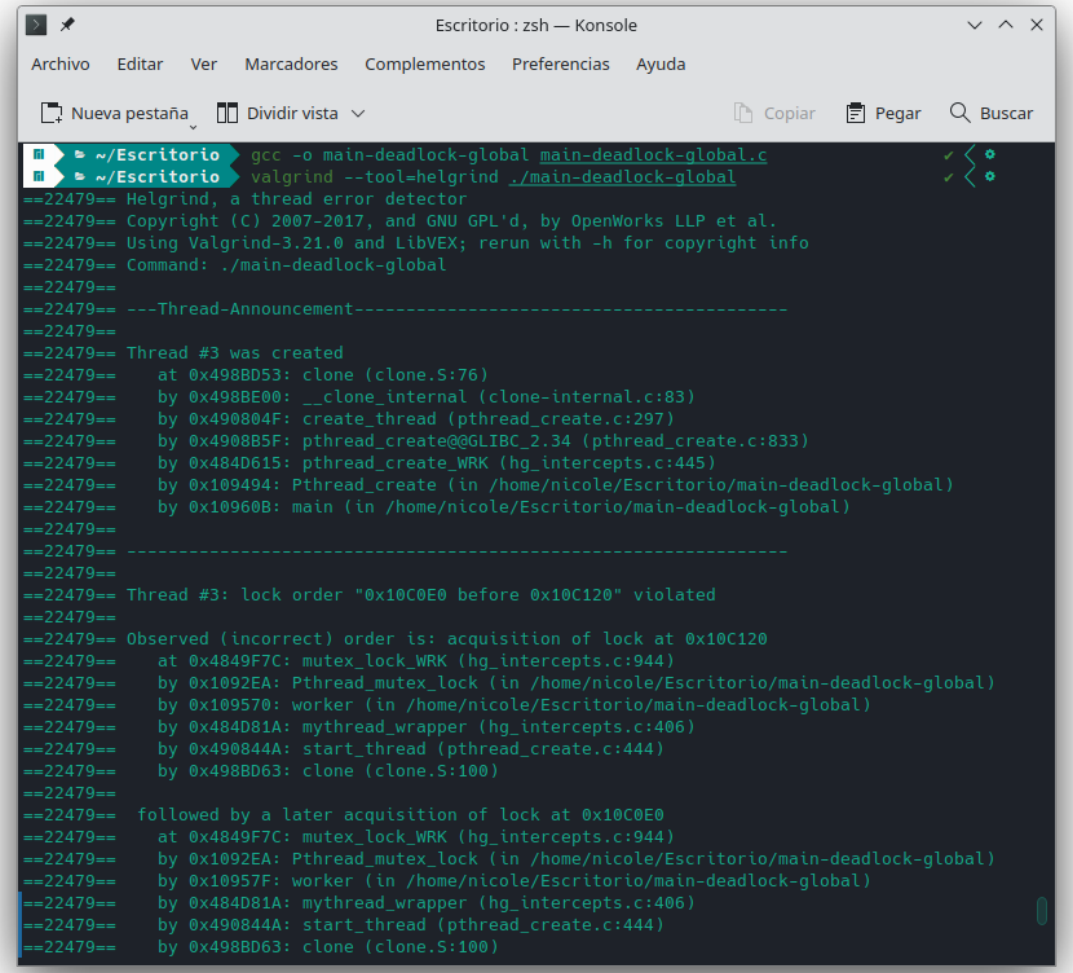


5. Now run helgrind on main-deadlock-global.c. Examine the code;
does it have the same problem that main-deadlock.c has? Should helgrind
be reporting the same error? What does this tell you about tools like helgrind?

código:

```
#include <stdio.h>

#include "mythreads.h"

pthread_mutex_t g = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m2 = PTHREAD_MUTEX_INITIALIZER;

void* worker(void* arg) {
        Pthread_mutex_lock(&g);
        if ((long long) arg == 0) {
   Pthread_mutex_lock(&m1);
```

```
    Pthread_mutex_lock(&m2);
        } else {
    Pthread_mutex_lock(&m2);
    Pthread_mutex_lock(&m1);
        }
        Pthread_mutex_unlock(&m1);
        Pthread_mutex_unlock(&m2);
        Pthread_mutex_unlock(&g);
        return NULL;
}

int main(int argc, char *argv[]) {
        pthread_t p1, p2;
        Pthread_create(&p1, NULL, worker, (void *) (long long) 0);
        Pthread_create(&p2, NULL, worker, (void *) (long long) 1);
        Pthread_join(p1, NULL);
        Pthread_join(p2, NULL);
        return 0;
```

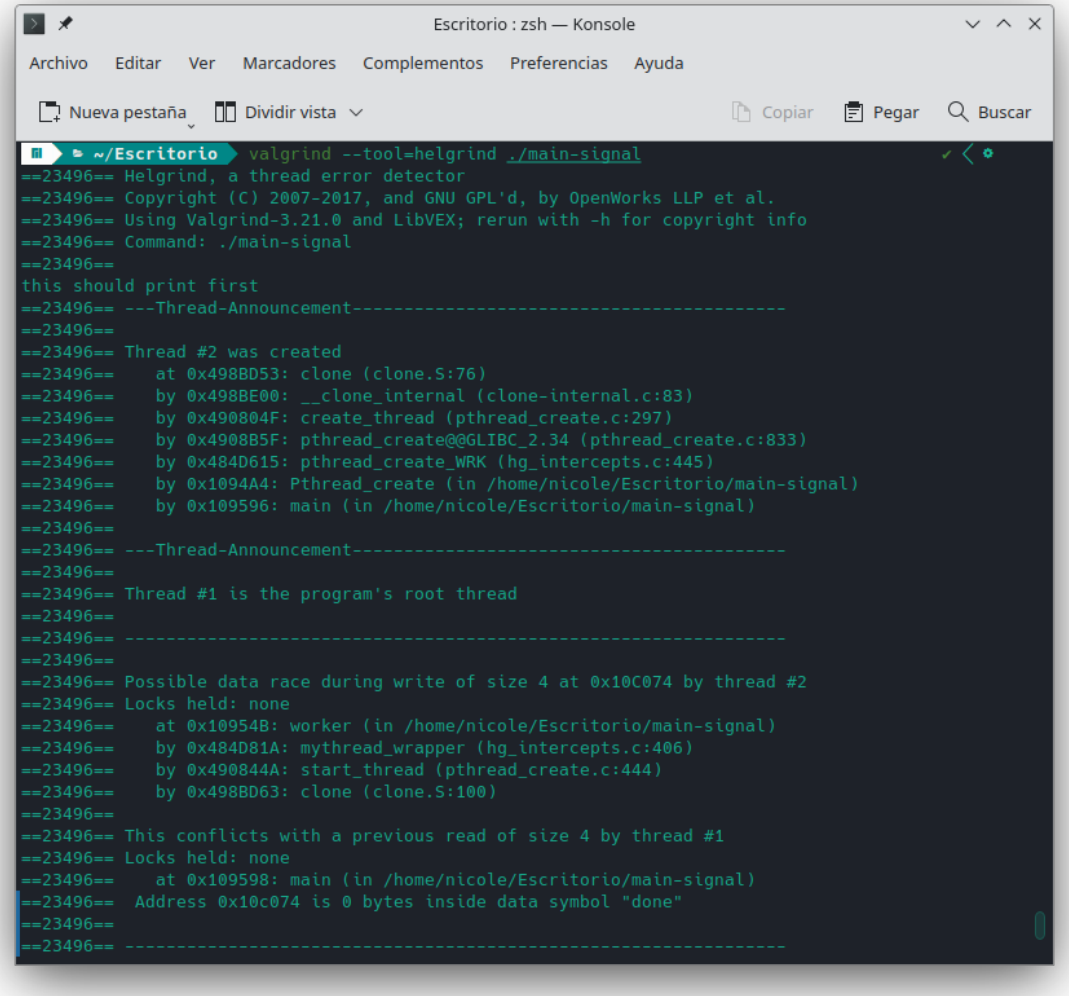


6. Let’s next look at main-signal.c. This code uses a variable (done) to signal that the child is done and that the parent can now continue. Why is this code inefficient? (what does the parent end up spending its time doing, particularly if the child thread takes a long time to complete?)

codigo:

```
#include <stdio.h>

#include "mythreads.h"

int done = 0;

void* worker(void* arg) {
        printf("this should print first\n");
        done = 1;
        return NULL;
}

int main(int argc, char *argv[]) {
        pthread_t p;
        Pthread_create(&p, NULL, worker, NULL);
        while (done == 0)
    ;
        printf("this should print last\n");
        return 0;
}
```

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



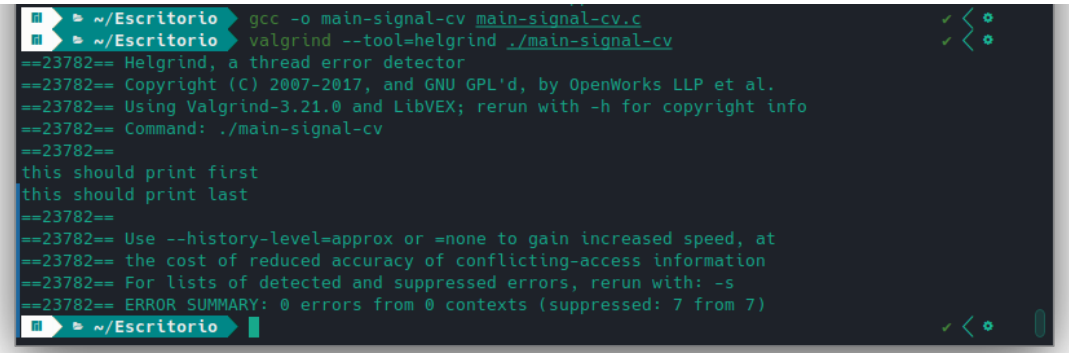
8. Now look at a slightly modified version of the code, which is found in main-signal-cv.c. This version uses a condition variable to do the signaling (and associated lock). Why is this code preferred to the previous version? Is it correctness, or performance, or both?

código:

```
#include <stdio.h>

#include "mythreads.h"

//
// simple synchronizer: allows one thread to wait for another
// structure "synchronizer_t" has all the needed data
// methods are:
//   init (called by one thread)
//   wait (to wait for a thread)
//   done (to indicate thread is done)
```

```
//
typedef struct __synchronizer_t {
        pthread_mutex_t lock;
        pthread_cond_t cond;
        int done;
} synchronizer_t;

synchronizer_t s;

void signal_init(synchronizer_t *s) {
        Pthread_mutex_init(&s->lock, NULL);
        Pthread_cond_init(&s->cond, NULL);
        s->done = 0;
}

void signal_done(synchronizer_t *s) {
        Pthread_mutex_lock(&s->lock);
        s->done = 1;
        Pthread_cond_signal(&s->cond);
        Pthread_mutex_unlock(&s->lock);
}

void signal_wait(synchronizer_t *s) {
        Pthread_mutex_lock(&s->lock);
        while (s->done == 0)
   Pthread_cond_wait(&s->cond, &s->lock);
        Pthread_mutex_unlock(&s->lock);
}

void* worker(void* arg) {
        printf("this should print first\n");
        signal_done(&s);
        return NULL;
}

int main(int argc, char *argv[]) {
        pthread_t p;
        signal_init(&s);
        Pthread_create(&p, NULL, worker, NULL);
        signal_wait(&s);
        printf("this should print last\n");

        return 0;
}
```

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


~/Escritorio gcc -o main-signal-cv main-signal-cv.c
~/Escritorio valgrind --tool=helgrind ./main-signal-cv
==23782== Helgrind, a thread error detector
==23782== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et al.
==23782== Using Valgrind-3.21.0 and LibVEX; rerun with -h for copyright info
==23782== Command: ./main-signal-cv
==23782==
this should print first
this should print last
==23782==
==23782== Use --history-level=approx or =none to gain increased speed, at
==23782== the cost of reduced accuracy of conflicting-access information
==23782== For lists of detected and suppressed errors, rerun with: -s
==23782== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 7 from 7)
~/Escritorio