GRUPPE 15 | Tobias Schoch, Luis Nothvogel

Simulation wurde auf dem HTWG Container ausgeführt

27.1

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

```
---Thread-Announcement------------------------------------------

Thread #1 is the program's root thread

---Thread-Announcement------------------------------------------

Thread #2 was created
   at 0x49734BE: clone (clone.S:71)
   by 0x485FDDE: create_thread (createthread.c:101)
   by 0x486180D: pthread_create@@GLIBC_2.2.5 (pthread_create.c:826)
   by 0x483C6B7: pthread_create_WRK (hg_intercepts.c:427)
   by 0x10940C: Pthread_create (mythreads.h:51)
   by 0x1094CF: main (main-race.c:14)

----------------------------------------------------------------
```

```
Possible data race during read of size 4 at 0x10C07C by thread #1
Locks held: none
   at 0x1094D0: main (main-race.c:15)

This conflicts with a previous write of size 4 by thread #2
Locks held: none
   at 0x109497: worker (main-race.c:8)
   by 0x483C8B6: mythread_wrapper (hg_intercepts.c:389)
   by 0x4860FA2: start_thread (pthread_create.c:486)
   by 0x49734CE: clone (clone.S:95)
 Address 0x10c07c is 0 bytes inside data symbol "balance"
```

```
7   void* worker(void* arg) {
8       balance++; // unprotected access
9       return NULL;
10  }
11
12  int main(int argc, char *argv[]) {
13      pthread_t p;
14      Pthread_create(&p, NULL, worker, NULL);
15      balance++; // unprotected access
16      Pthread_join(p, NULL);
17      return 0;
18  }
```

Wie man anhand der Screenshots sieht, zeigt einem helgrind nicht nur mögliche Race conditions und Fehler an, sondern auch wo und wie viele Threads erstellt werden. Zudem kann es auch die richtige und genaue Codestelle anzeigen wo eventuelle Fehler auftreten.

27.2

```
7   void* worker(void* arg) {
8       //balance++; // unprotected access
9       return NULL;
10  }
11
12  int main(int argc, char *argv[]) {
13      pthread_t p;
14      Pthread_create(&p, NULL, worker, NULL);
15      balance++; // unprotected access
16      Pthread_join(p, NULL);
17      return 0;
18  }
```

```
==16818== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

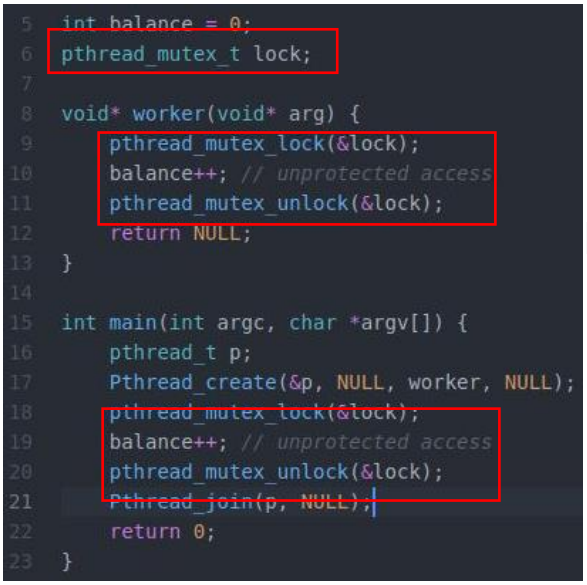
Hier haben wir zuerst im Thread die „Race Condition“ auskommentiert (siehe Screenshot). Wie zu erwarten gibt helgrind keine Fehlermeldung aus, da die Race Condition des Threads eliminiert wird. Das gleiche passiert auch, wenn man nur die „Race Condition“ in Main auskommentiert.

```
5   int balance = 0;
6   pthread_mutex_t lock;
7
8   void* worker(void* arg) {
9       pthread_mutex_lock(&lock);
10      balance++; // unprotected access
11      pthread_mutex_unlock(&lock);
12      return NULL;
13  }
14
15  int main(int argc, char *argv[]) {
16      pthread_t p;
17      Pthread_create(&p, NULL, worker, NULL);
18      balance++; // unprotected access
19      Pthread_join(p, NULL);
20      return 0;
21  }
```

```
Lock at 0x10C0A0 was first observed
   at 0x4839CCC: mutex_lock_WRK (hg_intercepts.c:909)
   by 0x10949D: worker (main-race.c:9)
   by 0x483C8B6: mythread_wrapper (hg_intercepts.c:389)
   by 0x4860FA2: start_thread (pthread_create.c:486)
   by 0x49734CE: clone (clone.S:95)
 Address 0x10c0a0 is 0 bytes inside data symbol "lock"

Possible data race during read of size 4 at 0x10C084 by thread #1
Locks held: none
   at 0x1094EC: main (main-race.c:18)

This conflicts with a previous write of size 4 by thread #2
Locks held: 1, at address 0x10C0A0
   at 0x1094A7: worker (main-race.c:10)
   by 0x483C8B6: mythread_wrapper (hg_intercepts.c:389)
   by 0x4860FA2: start_thread (pthread_create.c:486)
   by 0x49734CE: clone (clone.S:95)
 Address 0x10c084 is 0 bytes inside data symbol "balance"

ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Im Anschluss haben wir einen Lock zum Programm hinzugefügt. Der Lock wurde ausserhalb der Klassen erstellt, dass keine Fehler hervorgerufen werden. Danach haben wir unseren Lock für den zweiten Thread verwendet. Helgrind gibt uns allerdings eine Fehlermeldung

zurück, allerdings mit der Notation, dass wir einen Lock verwenden und wo dieser eingesetzt wird. Die Fehler kommen von dem Thread in der Main Klasse, da es hier immer noch zu „Race Conditions“ kommen kann, da diese noch keinen Lock hat.



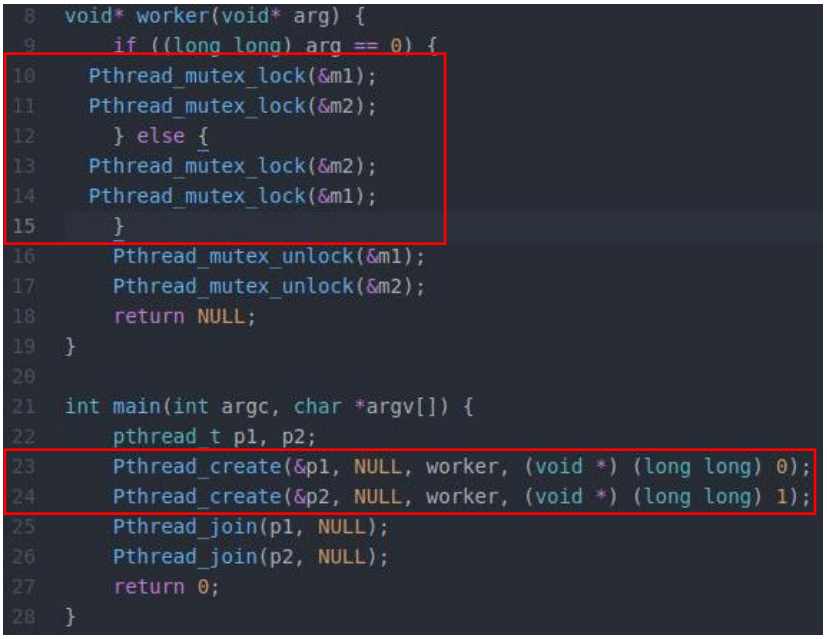
```
int balance = 0;
pthread_mutex_t lock;

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;
}
```

Unser Fehler wird nun behoben, indem wir unser Lock nun auf beide Threads anwenden.

Somit haben wir keine Fehler oder Race Conditions mehr.

```
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 7 from 7)
```

27.3



```
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;
}
```

```
Thread #3: lock order "0x10C0A0 before 0x10C0E0" violated
```

```
ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 7 from 7)
```

Das Deadlock-Problem in der Datei wird durch die markierten Zeilen hervorgerufen. Hier wird zuerst der Lock für Thread 1 aktiviert und direkt danach der Lock für Thread 2. Daher sind beide Locks aktiv und können nicht weitermachen, da beide darauf warten, dass der andere Thread den Lock deaktiviert. Das Programm kann daher nicht enden.

27.4

```
Thread #3: lock order "0x10C0A0 before 0x10C0E0" violated
```

```
ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 7 from 7)
```

Helgrind zeigt hier einen „Lock Order Violated“ Error an. Wie bereits in 27.3 erwähnt, hängt dieser Fehler zusammen mit unserem Deadlock Fehler. Beide Locks für die jeweiligen Threads werden aktiviert und können sich dadurch auch nicht mehr deaktivieren.

27.5

```
Thread #3: lock order "0x10C0E0 before 0x10C120" violated

Observed (incorrect) order is: acquisition of lock at 0x10C120
   at 0x4839CCC: mutex_lock_WRK (hg_intercepts.c:909)
   by 0x10928F: Pthread_mutex_lock (mythreads.h:23)
   by 0x1094CA: worker (main-deadlock-global.c:15)
   by 0x483C8B6: mythread_wrapper (hg_intercepts.c:389)
   by 0x4860FA2: start_thread (pthread_create.c:486)
   by 0x49734CE: clone (clone.S:95)

 followed by a later acquisition of lock at 0x10C0E0
   at 0x4839CCC: mutex_lock_WRK (hg_intercepts.c:909)
   by 0x10928F: Pthread_mutex_lock (mythreads.h:23)
   by 0x1094D6: worker (main-deadlock-global.c:16)
   by 0x483C8B6: mythread_wrapper (hg_intercepts.c:389)
   by 0x4860FA2: start_thread (pthread_create.c:486)
   by 0x49734CE: clone (clone.S:95)

Required order was established by acquisition of lock at 0x10C0E0
   at 0x4839CCC: mutex_lock_WRK (hg_intercepts.c:909)
   by 0x10928F: Pthread_mutex_lock (mythreads.h:23)
   by 0x1094B0: worker (main-deadlock-global.c:12)
   by 0x483C8B6: mythread_wrapper (hg_intercepts.c:389)
   by 0x4860FA2: start_thread (pthread_create.c:486)
   by 0x49734CE: clone (clone.S:95)

 followed by a later acquisition of lock at 0x10C120
   at 0x4839CCC: mutex_lock_WRK (hg_intercepts.c:909)
   by 0x10928F: Pthread_mutex_lock (mythreads.h:23)
   by 0x1094BC: worker (main-deadlock-global.c:13)
   by 0x483C8B6: mythread_wrapper (hg_intercepts.c:389)
   by 0x4860FA2: start_thread (pthread_create.c:486)
   by 0x49734CE: clone (clone.S:95)

 Lock at 0x10C0E0 was first observed
   at 0x4839CCC: mutex_lock_WRK (hg_intercepts.c:909)
   by 0x10928F: Pthread_mutex_lock (mythreads.h:23)
   by 0x1094B0: worker (main-deadlock-global.c:12)
   by 0x483C8B6: mythread_wrapper (hg_intercepts.c:389)
   by 0x4860FA2: start_thread (pthread_create.c:486)
   by 0x49734CE: clone (clone.S:95)
 Address 0x10c0e0 is 0 bytes inside data symbol "m1"

 Lock at 0x10C120 was first observed
   at 0x4839CCC: mutex_lock_WRK (hg_intercepts.c:909)
   by 0x10928F: Pthread_mutex_lock (mythreads.h:23)
   by 0x1094BC: worker (main-deadlock-global.c:13)
   by 0x483C8B6: mythread_wrapper (hg_intercepts.c:389)
   by 0x4860FA2: start_thread (pthread_create.c:486)
   by 0x49734CE: clone (clone.S:95)
 Address 0x10c120 is 0 bytes inside data symbol "m2"
```

```c
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;
}
```

```
ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 7 from 7)
```

Das Programm main-deadlock-glocal.c hat eigentlich nicht mehr den Fehler, den das Programm aus 27.3 und 27.4 hat. Diesen Fehler gibt es nicht mehr, da wir nun mit einem Coarse-Grained-Lock arbeiten, dass die Fine-Grained-Locks beinhaltet und so den Fehler behebt.

Helgrind hingegen will uns erzählen, dass es den Fehler immer noch gibt. Daraus können wir schließen, dass Helgrind nicht ganz so perfekt ist und die Optimierung nicht beachtet.

27.6

```
7   void* worker(void* arg) {
8       printf("this should print first\n");
9       done = 1;
10      return NULL;
11  }
12
13  int main(int argc, char *argv[]) {
14      pthread_t p;
15      Pthread_create(&p, NULL, worker, NULL);
16      while (done == 0)
17    ;
18      printf("this should print last\n");
19      return 0;
20  }
```

Dieser Code ist ineffizient, da während der Worker Thread ausgeführt wird, im Main Thread die ganze Zeit die While schleife durchlaufen wird und damit Ressourcen verbraucht werden, anstatt sich lediglich schlafen zu legen.

27.7

```
Possible data race during write of size 1 at 0x523F1A5 by thread #1
Locks held: none
   at 0x48425C6: mempcpy (vg_replace_strmem.c:1536)
   by 0x48F59E4: _IO_new_file_xsputn (fileops.c:1243)
   by 0x48F59E4: _IO_file_xsputn@@GLIBC_2.2.5 (fileops.c:1204)
   by 0x48EB9DD: puts (ioputs.c:40)
   by 0x109501: main (main-signal.c:18)
 Address 0x523f1a5 is 21 bytes inside a block of size 1,024 alloc'd
   at 0x48367CF: malloc (vg_replace_malloc.c:299)
   by 0x48E971B: _IO_file_doallocate (filedoalloc.c:101)
   by 0x48F76FF: _IO_doallocbuf (genops.c:347)
   by 0x48F6987: _IO_file_overflow@@GLIBC_2.2.5 (fileops.c:752)
   by 0x48F5A8E: _IO_new_file_xsputn (fileops.c:1251)
   by 0x48F5A8E: _IO_file_xsputn@@GLIBC_2.2.5 (fileops.c:1204)
   by 0x48EB9DD: puts (ioputs.c:40)
   by 0x1094AD: worker (main-signal.c:8)
   by 0x483C8B6: mythread_wrapper (hg_intercepts.c:389)
   by 0x4860FA2: start_thread (pthread_create.c:486)
   by 0x49734CE: clone (clone.S:95)
 Block was alloc'd by thread #2
```

```
ERROR SUMMARY: 23 errors from 2 contexts (suppressed: 40 from 40)
```

Wie man sieht gibt helgrind hier eine „Possible data race condition“ zurück. Dies ist ersichtlich, da die while schleife über den gesamten Zeitraum auf die globale Variable „done“ zugreift.

Die hohe Fehleranzahl kommt durch die jeweiligen Aufrufe in der While loop, bis Thread 2 auf Thread 1 gesetzt wird.

27.8

Da das Programm nur warten muss und keine While Schleife abarbeiten muss, ist der Code auch logischerweise performanter. Zudem ist der Code auch korrekter, da Helgrind uns 0 Errors zurück liefert, statt ursprünglich 23.

```
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 7 from 7)
```

27.9

Wie bereits erwähnt liefert Helgrind keine Errors zurück.

```
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 7 from 7)
```