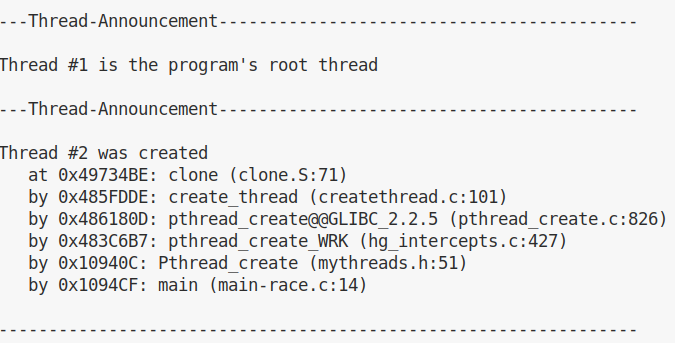
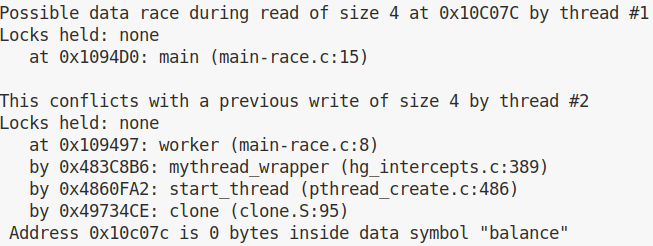
**GRUPPE 15 | Tobias Schoch, Luis Nothvogel**

**Simulation wurde auf dem HTWG Container ausgeführt**

**27.1**









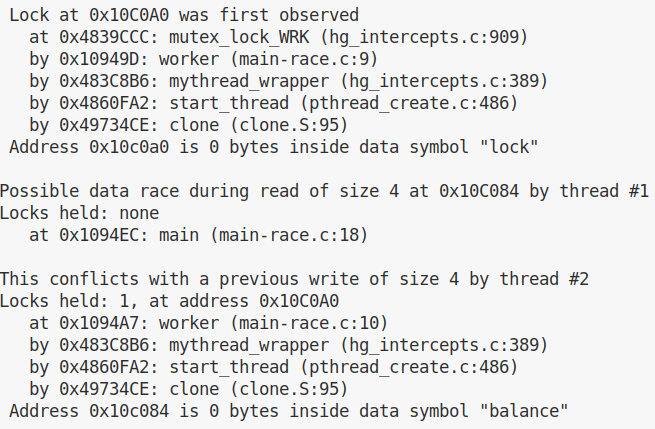
Wie man anhand der Screenshots sieht, zeigt einem helgrind nicht nur mögliche Race conditions und Fehler an, sondern auch wo und wie viele Threads erstellt werden. Zudem kann es auch die richtige und genaue Codestelle anzeigen wo eventuelle Fehler auftreten.

**27.2**

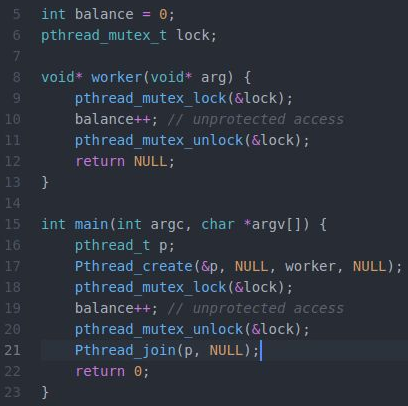




Hier haben wir zuerst im Thread die „Race Condition“ auskommentiert (siehe Screenshot). Wie zu erwarten gibt helgrind keine Fehlermeldung aus, da die Race Condition des Threads eliminiert wird. Das gleiche passiert auch, wenn man nur die „Race Condition“ in Main auskommentiert.



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.

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

Somit haben wir keine Fehler oder Race Conditions mehr.



**27.3**





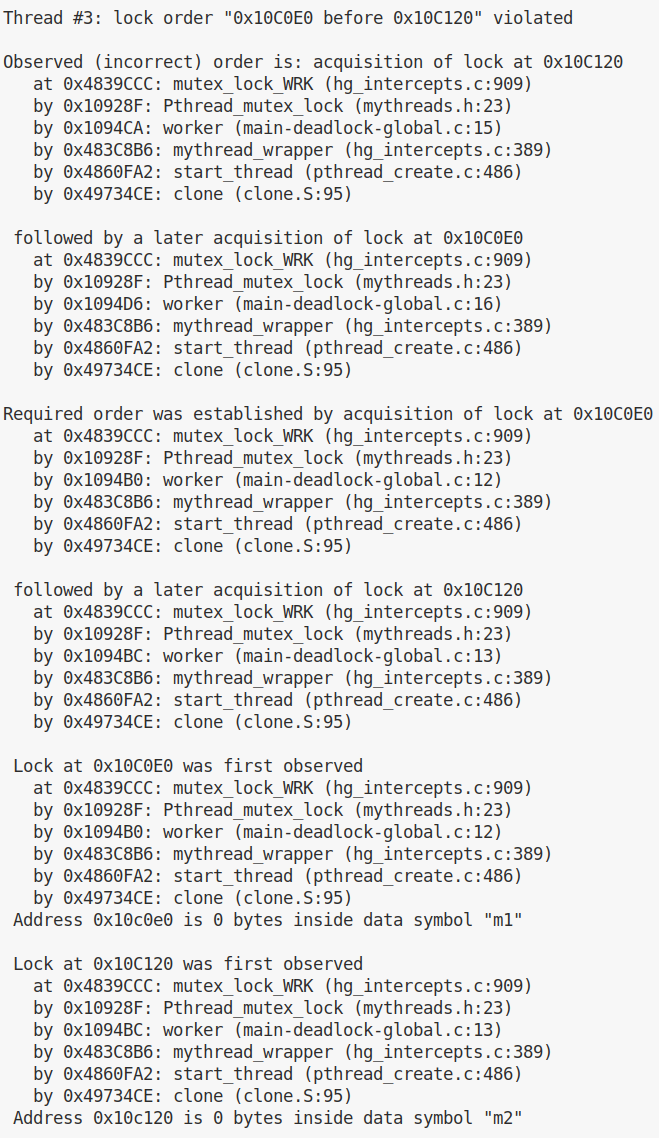
Das Deadlock-Problem in der Datei wird durch die markierten Zeilen hervorgerufen. Hier wird zuerst der Lock für Thread 1 aktiviert und direkt danach der Lock für Thread 2. Daher sind beide Locks aktiv und können nicht weitermachen, da beide darauf warten, dass der andere Thread den Lock deaktiviert.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

**27.4**





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.

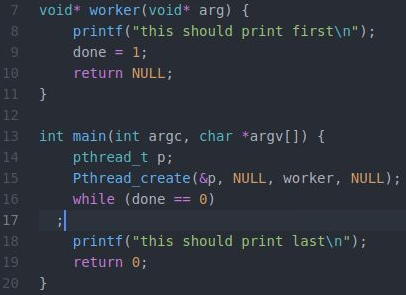
**27.5**



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.

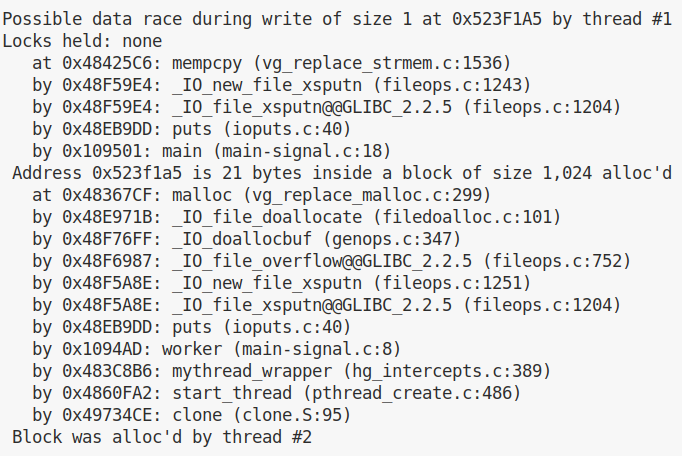
Helgrind hingegen will uns erzählen, dass es den Fehler immer noch gibt. Daraus können wir schließen, dass Helgrind nicht ganz so perfekt ist und die Optimierung nicht beachtet.

**27.6**

****

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**

****

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.

****

**27.9**

Wie bereits erwähnt liefert Helgrind keine Errors zurück.

****