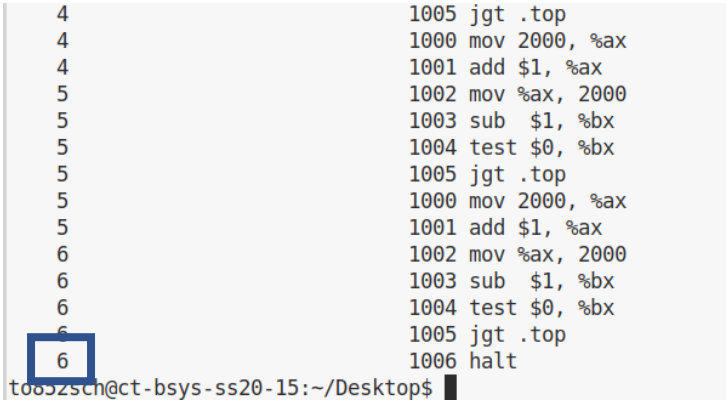
Gruppe 15 | Tobias Schoch, Luis Nothvogel
Simulation wurde auf dem HTWG Container ausgeführt

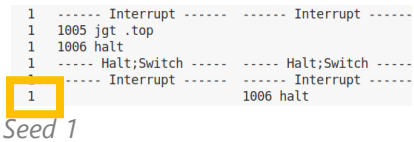
26.5

Die Loop wird für jeden Thread dreimal ausgeführt, da „bx" gleich 3 gesetzt wurde. %bx ist eine Variable in der .s Datei die testet ob noch eine weitere loop stattfindet. Somit erhalten wir als Ergebnis statt 2 mit einer Ausführung, sogar 6, da wir 3 Ausführungen haben.

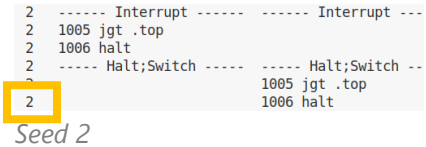


26.6

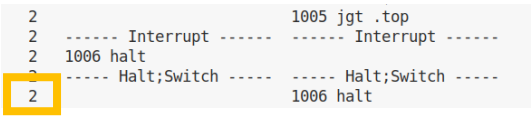
Der Timing Interrupt hat auf jeden Fall einen Einfluss, was ersichtlich wird, wenn man die Simulation mit verschiedenen Seeds laufen lässt und sich die Ergebnisse anschaut: Bei Seed 0 und 2 kommt als Ergebnis 2 raus, während bei Seed 1 das Ergebnis 1 rauskommt.



Seed 1

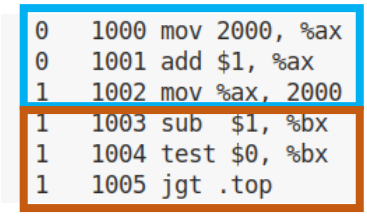


Seed 2



Seed 0

Folgende Befehle sind in der Datei *looping-race-nolock.s* vorhanden:



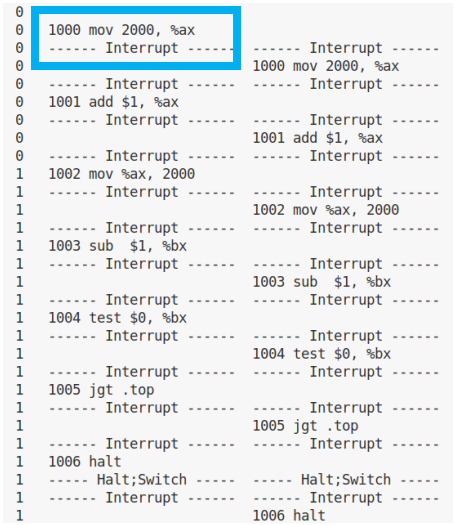
Die Befehle im blau umrandeten Kasten sind in der sogenannten **critical section**.
Die Befehle im braun umrandeten Kasten hingegen sind nicht critical.

Warum sind nur die ersten 3 Befehle critical? Da die ursprüngliche Variable noch geändert werden könnte, durch ein Timer Interrupt und das Ausführen der Befehle in einem anderen Thread.
Wenn nun wieder zum ursprünglichen Thread gesprungen wird, dann überschreibt dieser die Änderung des vorherigen Threads.
Daher beginnt die critical section beim Laden der Variable und endet beim Speichern.
Umgedreht heißt es, dass die braun umrandeten Befehle in welchen geschaut wird, ob noch eine Loop ausgeführt wird sicher für ein Timer Interrupt ist.

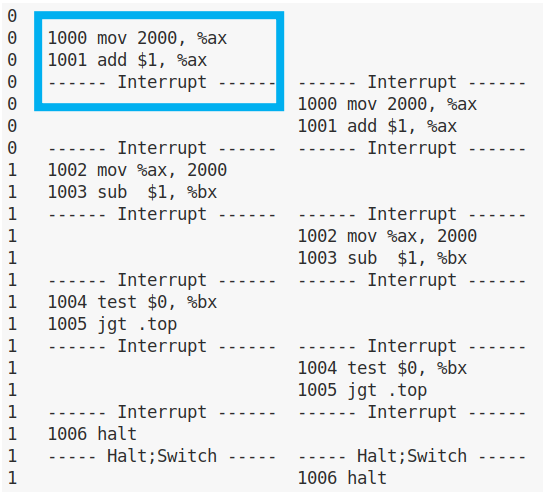
Der finale Wert sollte eigentlich 2 sein, da jeder Thread die Befehle ausführt und somit bei 2 Threads jeweils +1 auf die Variable gerechnet werden soll.

26.7

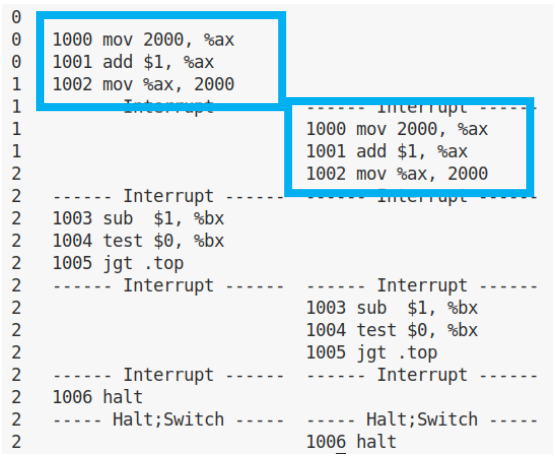
Der finale Wert für einen getakteten Timer Intervall von 1 ergibt die Zahl 1, da innerhalb der critical section ein Timer Interrupt geschieht, was dazu führt, dass unser Counter überschrieben wird.



Hier kann man gut sehen, dass auch hier bei einem Intervall von 2 die critical section durch ein Interrupt unterbrochen wird.



Die korrekte Antwort bekommt man ab einem Timer Intervall von 3 oder größer, da die critical section somit nicht unterbrochen wird von einem Timer Intervall, da aus 3 Befehlen besteht.
Bei einem Interrupt Intervall von kleiner als 3, wird die Critical Section unterbrochen, was dazu führt, dass unser Counter überschrieben wird.

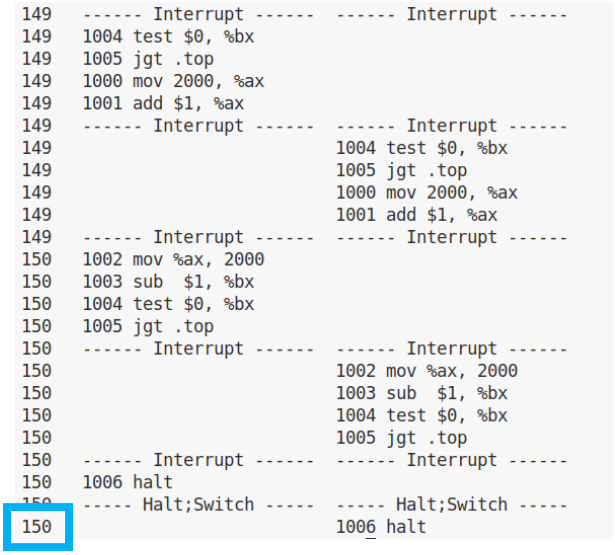


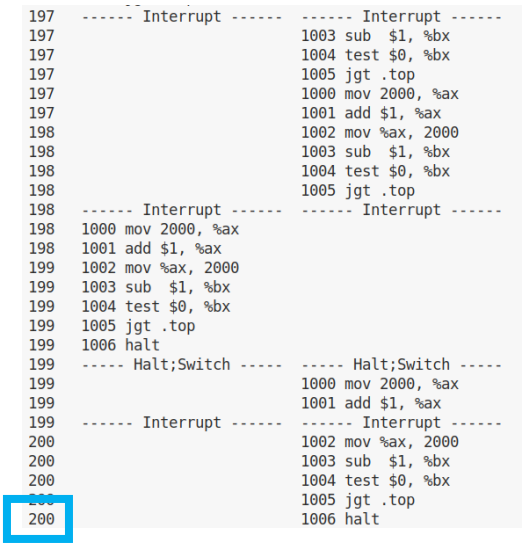
26.8

Hier kann man gut sehen, was passiert, wenn ein Timerinterrupt wie zum Beispiel 4 nimmt und eine Loopanzahl von 100.
Insgesamt gab es 50 Interrupts die nicht hätten passieren dürfen.

100 (Loopanzahl) * 2 (Threads) = 200 Counteranzahl
200 (Counteranzahl Sollte) – 150 (Counteranzahl Ist) = 50 fehlgeschlagene Interrupts.

Zudem sehen wir, wie bei jedem Interrupt ein unterschiedlicher Befehl ausgeführt wird.





Logischerweise indem man das Vielfache der Länge der Befehle nimmt von der „critical section“. Bei unserem Beispiel mit einer „critical section“ von 3 Befehlen wäre das Vielfache: 3,6,9,12,15, … etc.

Wenn man es z.B. mit einem Timer Interrupt von 9 versucht, erhält man das richtige Ergebnis von 200.

Auch wenn es zum Schluss innerhalb des zweiten Threads zu einem Interrupt kommt, funktioniert es trotzdem, da dieser Interrupt innerhalb nur aufgrund einer verkürzten Befehlskette im ersten Thread zuvor kommt.

26.9

```
0       1
0       1   1000 test $1, %ax
0       1   1001 je .signaller
1       1   1006 mov  $1, 2000
1       1   1007 halt
1       0   ----- Halt;Switch -----  ----- Halt;Switch ----
1       0                          1000 test $1, %ax
1       0                          1001 je .signaller
1       0                          1002 mov  2000, %cx
1       0                          1003 test $1, %cx
1       0                          1004 jne .waiter
1       0                          1005 halt
```

Die Klasse main prüft, ob ax mit 1 den Code der Klasse signaler ausführt, oder mit 0 einfach den Code weiterlaufen lässt.

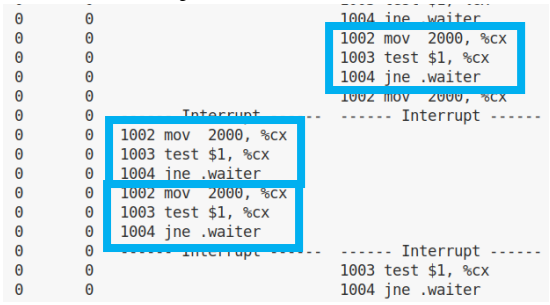
Da wir zuerst den Thread 0 ausführen, der über ax mit 1 initialisiert ist, wird zur Klasse signaler gesprungen. Dabei wird die Adresse 2000 mit 1 initialisiert.
Im Anschluss wird Thread 1 ausgeführt, der über ax mit 0 initialisiert ist, weshalb einfach die Befehle weiter ausgeführt werden, worüber man dann zur Klasse waiter kommt.

Dabei wird der Inhalt von der Adresse 2000 in %cx kopiert. Im Anschluss wird getestet ob die Variable %cx gleich 1 ist.
Wenn die Variable ungleich 1 ist, dann wird erneut zur Klasse waiter gesprungen.
Falls die Variable doch 0 ist, wird halt ausgeführt, was dazu führt, dass das Programm im Thread beendet wird.

Der finale Wert in der Adresse 2000 ist 1, da diese logischerweise mit 1 initialisiert wurde in der Klasse signaler.

26.10

Thread 0 und Thread 1 ist gefangen in einer Endlosschleife. Die blau umrandeten Befehle sind jeweils eine Schleife.



Wie man hier sieht, ruft sich die Klasse immer wieder selbst auf und kann sich nur lösen, wenn der Inhalt von Adresse 2000 ungleich 0 ist.
Da dies mit unserer neuen Konfiguration nicht der Fall ist und die Threads direkt in waiter springen, haben wir eine Endlosschleife als Resultat.

```
.waiter
mov  2000, %cx
test $1, %cx
jne .waiter
halt
```

Eine Änderung des Intervalls kann dieses Problem nicht lösen, da das Grundproblem mit der Endlosschleife in der Initialisierung von ax für die beiden Threads liegt.

Eine effiziente CPU-Nutzung ist Interpretationssache. Wir interpretieren eine effiziente CPU-Nutzung als 100% genutzte Zeit. Daher wird in dieser Sichtweise die CPU effizient genutzt. Allerdings ist eine Endlosschleife wohl auch nicht Sinn der Sache.