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

**Simulation wurde auf dem HTWG Container ausgeführt**

**28.4**

Zuerst sollten wir definieren was ein gutes Ergebnis und was ein schlechtes Ergebnis ist.

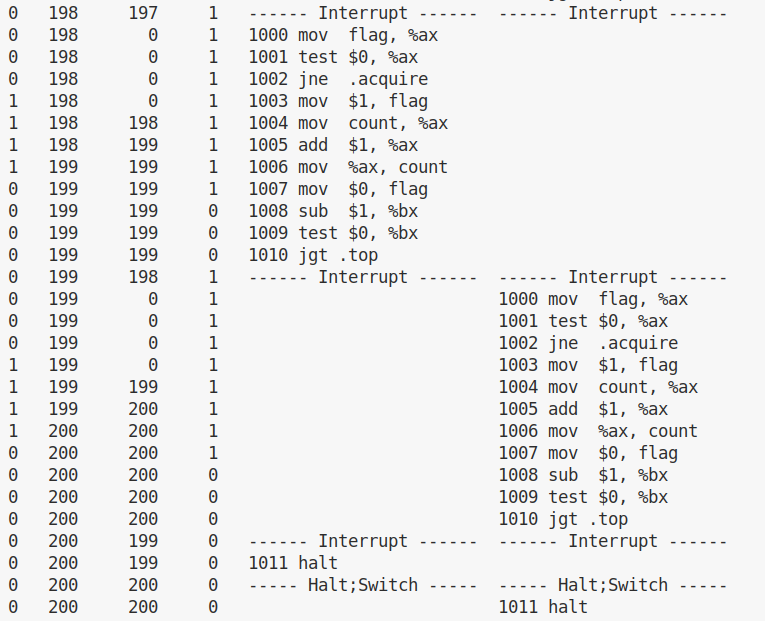
Ein gutes Ergebnis ist, wenn alle Additionen beider Threads nicht überschrieben worden sind und wir bei 10 Loops mit 2 Threads als Ergebnis 20 erhalten.

Alles andere wäre für uns ein schlechtes Ergebnis.

Wenn wir also nun schauen, wie viele Befehle unser Programm hat, kommen wir auf 11.

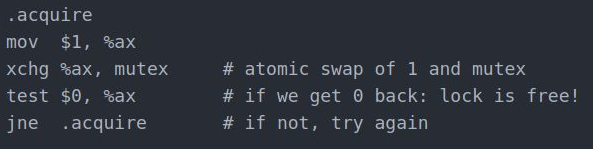
Wenn ein Thread also einen Interrupt Intervall von 11 oder einem Vielfachen davon bekommt, kann das Programm im Thread zu Ende laufen ohne ein Interrupt und somit wird es auch auf viele Loops kein Problem geben. Dies kann man schön an dem Beispiel sehen, aus dem Screenshot unten.

****



**28.5**

Der Code ist fast genau derselbe, wie noch zuvor in der Datei flag.s, allerdings mit einem Unterschied.



Zuerst wird %ax 1 gesetzt. Dieser Wert wird im Anschluss über xchg miteinander geswappt.

Im Anschluss wird überprüft, welchen Wert „mutex“ hatte und an ax weitergegeben hat.

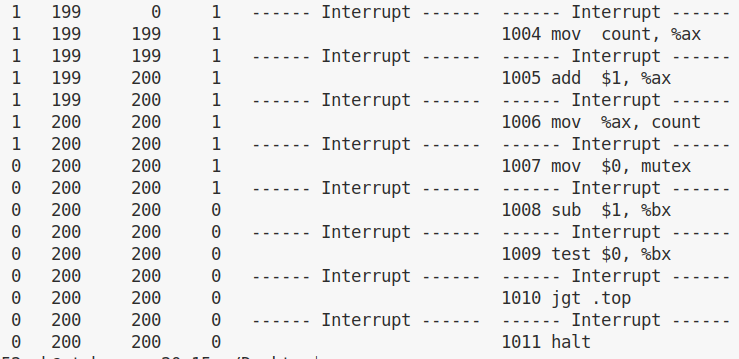
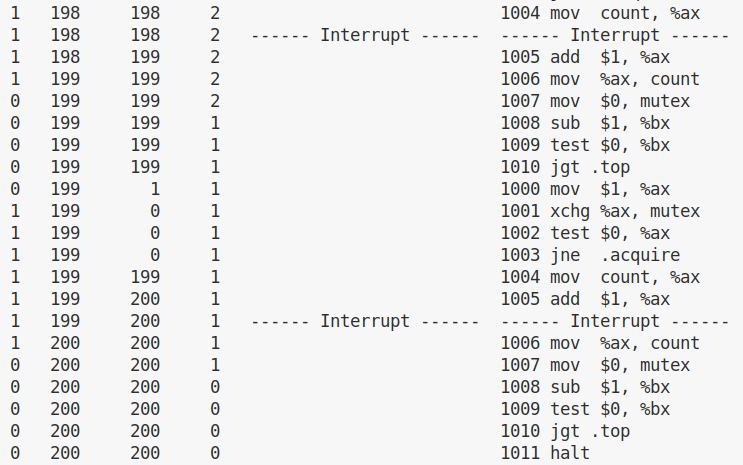
Wenn der Wert 0 war wird weiter mit dem Programm fortgefahren, da der Lock frei ist.

Wenn wir allerdings eine 1 zurückbekommen, dann ist der Lock reserviert und wir springen zurück zum Anfang der Klasse .acquire.



Zum Schluss wird mutex gleich 0 gesetzt und damit wieder befreit, so dass der andere Thread nun darauf zugreifen kann.

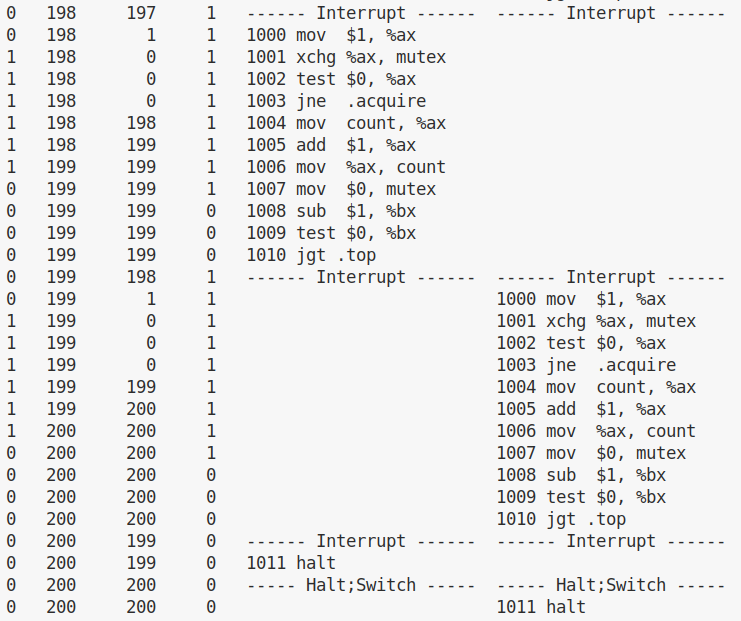
**28.6**

Der Code funktioniert wie er sollte, auch bei unterschiedlichen Intervall Interrupts.

Wenn jedoch der Timer interrupt zu kurz wird z.B. 1 oder 2, ist die CPU sehr ineffizient genutzt, da erst der Thread gewechselt werden muss nach jedem Befehl.

Wenn wir hingegen einen Timer Interrupt von 11 (Befehle im Programm) oder ein Vielfaches davon nehmen, dann wird das Programm einmal ausgeführt, bevor es zum nächsten Thread springt. Daher wäre 11 oder ein Vielfaches davon für die Effizienz der perfekte Timer Intervall, da ansonsten evtl. die Lock für ein Thread offen ist und damit CPU-Ressourcen verschwendet werden. Die orangenen Kästen zeigen die perfekte Ausführung.

Wir sehen unten im Bild einen Screenshot. Dabei ist jeweils ein Thread in Benutzung, während der Andere im Wartezustand verweilt. Das ist ebenfalls eine ineffiziente Nutzung der CPU-Ressourcen.

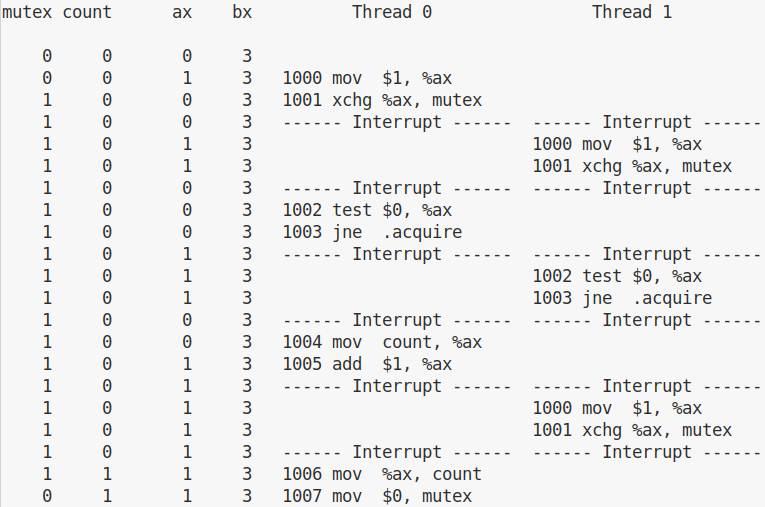


**28.7**

Der erste Thread reserviert den Lock (blau). Im nächsten Schritt kommt der zweite Thread und führt ebenfalls zwei Schritte aus (orange). Danach testet Thread 0 ob der Lock frei ist, was auch der Fall ist (grün). Als Thread 1 hingegen schaut ob der Lock frei ist, bekommt er ein Not Equal und wird wieder an den Anfang von Klasse acquire transportiert (rot).

Thread 0 kann nun weiter fortfahren, da der Lock frei ist (braun).

Thread 1 fängt wieder in der acquire Klasse von vorne an (lila).

****

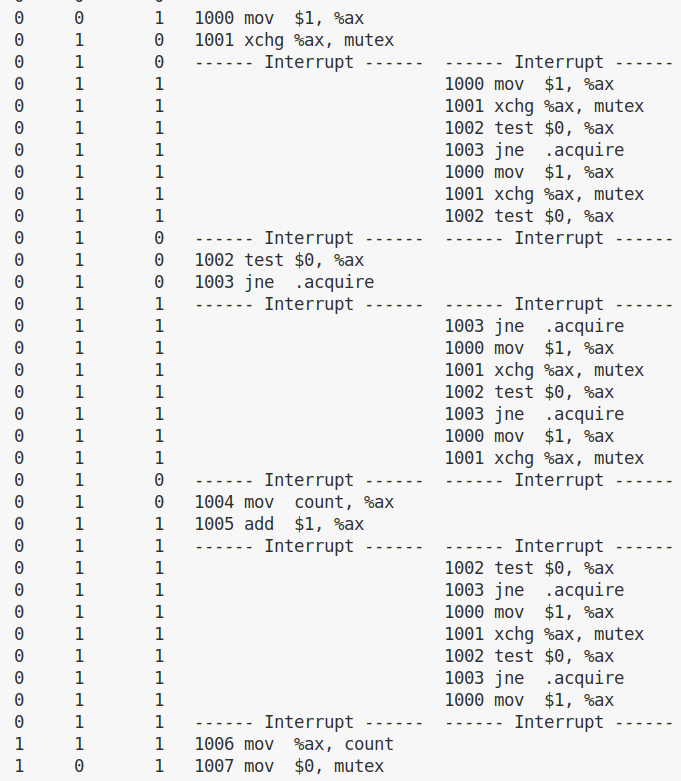
Um dieses Szenario zu erreichen, darf jeweils ein Thread zwei Befehle ausführen.

Der Code agiert genauso, wie er es sollte.



Ein weiteres interessantes Szenario für mich war folgendes:

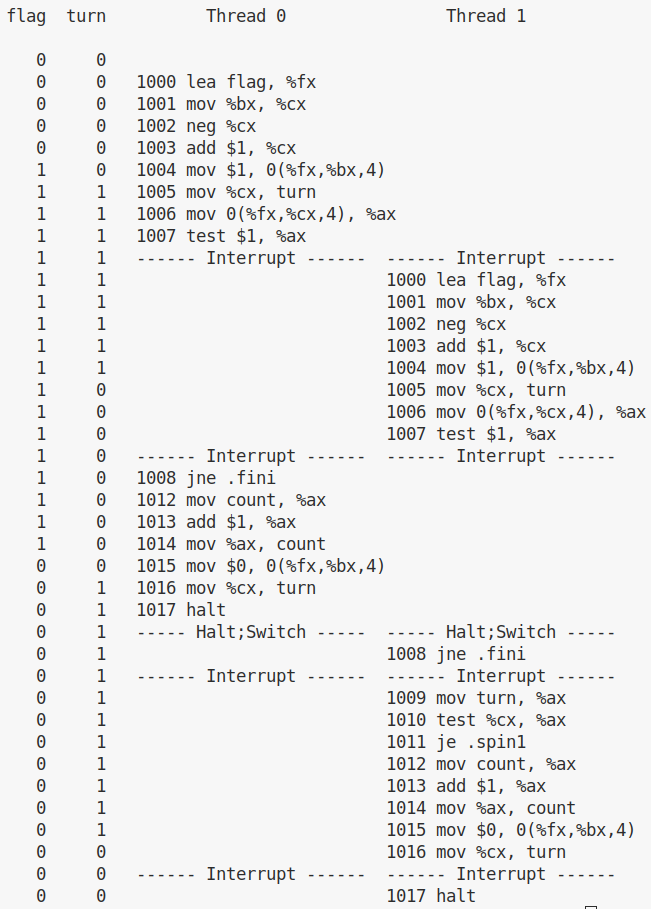


Dabei reserviert Thread 0 den Lock, während Thread 1 im Anschluss 7 time slices hat. Dabei ist zu sehen, dass Thread 1 die „übrige Zeit“ in der Schleife festsitzt.

Das führt natürlich zu einer sehr schlechten CPU-Ressourcen Nutzung.

**28.9**

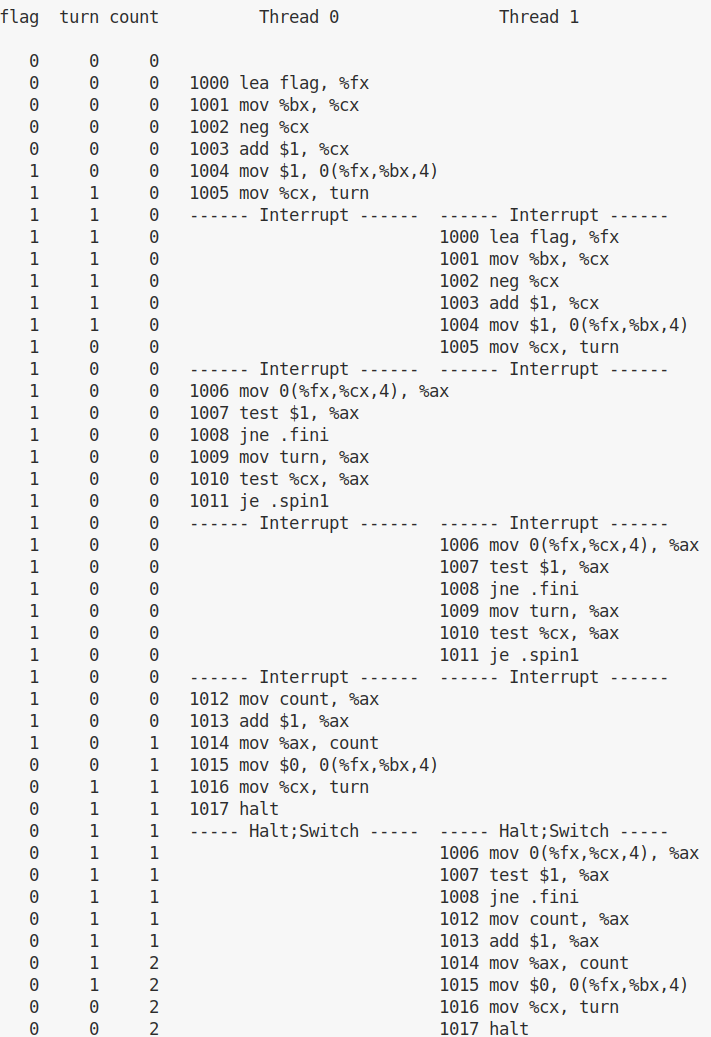


****

Uns ist aufgefallen, dass er nicht zwingend seinen Timeslice voll ausnutzen muss, da er eventuell vorher schon fertig ist. Ansonsten haben wir leider kein weiteres unterschiedliches Verhalten feststellen können. Ansonsten konnten wir noch ein Optimalintervall bei 2 Threads festellen, was bei 15 liegt.

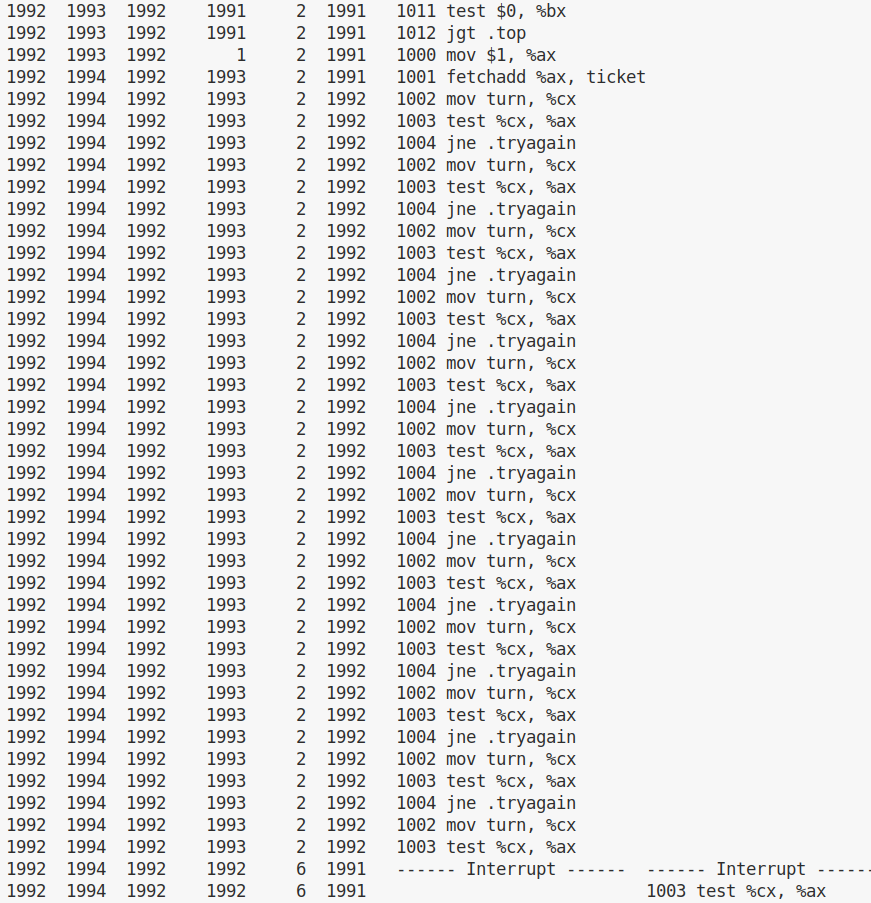
**28.10**





Mit diesem Beispiel beweisen wir das der Code funkioniert. Hier laufen beide Threads bis zu dem Punkt im Code wo beide ihr Flag & Turn setzen. Da der Code mit der -c variante zu ende läuft, gab es hier keine Deadlocks. Auch haben sich beide richtig ausgeschlossen, da am ende Count = 2 rauskommt, was die Richige Zahl ist.

**28.11**





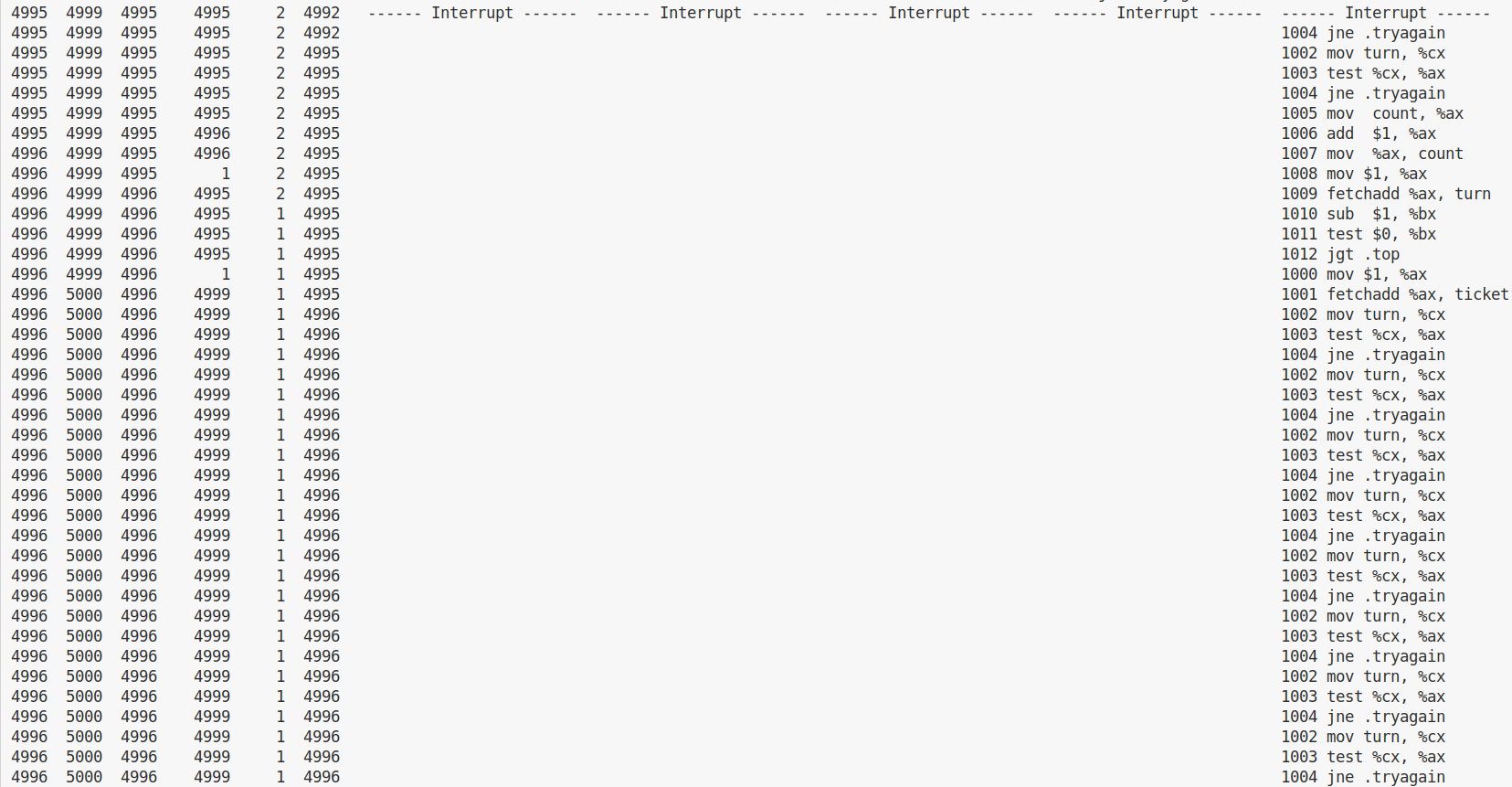
Ja, wie man hier im Bild sieht, ist das der größte Ausschnitt einer Loop. Dabei ist jeweils der Adressbereich 1002 bis 1004 der spin-waiting Bereich. Dieser ist sehr groß in den jeweiligen loops. In orange markiert ist der gesamte spin-waiting Bereich.

In einer loop hatten wir genau 50 Befehle. Davon waren lediglich 15 nicht im spin-waiting Zustand, also 30% insgesamt. Das bedeutet im Umkehrschluss, dass ganze 70% im spin-waiting Zustand sind. Daher würden wir sagen, dass sogar sehr viel Zeit in diesem Zustand verbracht wurde.

**28.12**

Der Ausführungszeit pro Thread ist immer noch gleichlang logischerweise, allerdings hat sich die Gesamtdauer des Prozesses verlängert: 1000(bx) \* Anzahl der Threads.

Auch hier kann man wieder schön sehen, wie es den langen spin-waiting Zustand gibt.

****



**28.13**





Dies sollte ganz gut aufzeigen das test-and-set am spinnen ist während bei yield.s einfach yield aufgerufen wird um den anderen thread wieder arbeiten zu lassen. Das wird im Simulator allerdings nicht so berücksichtigt denn dieser ruft yield auf aber der andere Thread kommt trotzdem noch nicht dran. Aber theoretisch sollte dies passieren.

Man hat immer dann einen Vorteil mit Yield sobald der Zeitinterrupt so gesetzt ist, dass er interrupt wenn die Flag von einem Thread noch nicht wider freigeben ist. Oder auch sobald sich ein Thread die Flag holt und dann aber bevorzugt der andere Thread ausgeführt wird.

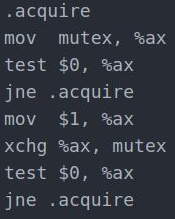
**28.14**

In der test-and-test-and-set.s hat eine Änderung im Vergleich zur test-and-set.s. Der orangen umrandete Kasten zeigt, was im Vergleich zur Vorgängerversion geändert wurde.

Hier wird getestet, ob wir überhaupt die Möglichkeit haben unseren Befehl auszuführen.

Denn wenn mutex (unser Lock) bereits auf 1 gesetzt ist durch einen anderen Thread, braucht es gar nicht erst xchg ausführen.

So kann das Programm bei jeder Ausführung, bei der der Lock im anderen Thread gesetzt ist, einen write sparen und ist damit im Gesamten schneller fertig.

****