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.

```
./x86.py -p flag.s -M flag,count -R ax,bx -c -a bx=100 -i 11 -c
```

```
0   198     197     1   ------ Interrupt ------   ------ Interrupt ------
0   198       0     1   1000 mov  flag, %ax
0   198       0     1   1001 test $0, %ax
0   198       0     1   1002 jne  .acquire
1   198       0     1   1003 mov  $1, flag
1   198     198     1   1004 mov  count, %ax
1   198     199     1   1005 add  $1, %ax
1   199     199     1   1006 mov  %ax, count
0   199     199     1   1007 mov  $0, flag
0   199     199     0   1008 sub  $1, %bx
0   199     199     0   1009 test $0, %bx
0   199     199     0   1010 jgt .top
0   199     198     1   ------ Interrupt ------   ------ Interrupt ------
0   199       0     1                            1000 mov  flag, %ax
0   199       0     1                            1001 test $0, %ax
0   199       0     1                            1002 jne  .acquire
1   199       0     1                            1003 mov  $1, flag
1   199     199     1                            1004 mov  count, %ax
1   199     200     1                            1005 add  $1, %ax
1   200     200     1                            1006 mov  %ax, count
0   200     200     1                            1007 mov  $0, flag
0   200     200     0                            1008 sub  $1, %bx
0   200     200     0                            1009 test $0, %bx
0   200     200     0                            1010 jgt .top
0   200     199     0   ------ Interrupt ------   ------ Interrupt ------
0   200     199     0   1011 halt
0   200     200     0   ----- Halt;Switch -----   ----- Halt;Switch -----
0   200     200     0                            1011 halt
```

28.5

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

```
.acquire
mov  $1, %ax
xchg %ax, mutex     # atomic swap of 1 and mutex
test $0, %ax        # if we get 0 back: lock is free!
jne  .acquire       # if not, try again
```

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.

```
# release lock
mov  $0, mutex
```

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.

```
./x86.py -p test-and-set.s -M mutex,count -R ax,bx -a bx=100 -i 1 -c
```

```
1   199     0     1   ------ Interrupt ------  ------ Interrupt ------
1   199     199   1                            1004 mov  count, %ax
1   199     199   1   ------ Interrupt ------  ------ Interrupt ------
1   199     200   1                            1005 add  $1, %ax
1   199     200   1   ------ Interrupt ------  ------ Interrupt ------
1   200     200   1                            1006 mov  %ax, count
1   200     200   1   ------ Interrupt ------  ------ Interrupt ------
0   200     200   1                            1007 mov  $0, mutex
0   200     200   1   ------ Interrupt ------  ------ Interrupt ------
0   200     200   0                            1008 sub  $1, %bx
0   200     200   0   ------ Interrupt ------  ------ Interrupt ------
0   200     200   0                            1009 test $0, %bx
0   200     200   0   ------ Interrupt ------  ------ Interrupt ------
0   200     200   0                            1010 jgt .top
0   200     200   0   ------ Interrupt ------  ------ Interrupt ------
0   200     200   0                            1011 halt
```

```
1   198     198     2                          1004 mov  count, %ax
1   198     198     2   ------ Interrupt ------  ------ Interrupt ------
1   198     199     2                          1005 add  $1, %ax
1   199     199     2                          1006 mov  %ax, count
0   199     199     2                          1007 mov  $0, mutex
0   199     199     1                          1008 sub  $1, %bx
0   199     199     1                          1009 test $0, %bx
0   199     199     1                          1010 jgt .top
0   199       1     1                          1000 mov  $1, %ax
1   199       0     1                          1001 xchg %ax, mutex
1   199       0     1                          1002 test $0, %ax
1   199       0     1                          1003 jne  .acquire
1   199     199     1                          1004 mov  count, %ax
1   199     200     1                          1005 add  $1, %ax
1   199     200     1   ------ Interrupt ------  ------ Interrupt ------
1   200     200     1                          1006 mov  %ax, count
0   200     200     1                          1007 mov  $0, mutex
0   200     200     0                          1008 sub  $1, %bx
0   200     200     0                          1009 test $0, %bx
0   200     200     0                          1010 jgt .top
0   200     200     0                          1011 halt
```

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.

```
0   198     197     1   ------ Interrupt ------  ------ Interrupt ------
0   198       1     1   1000 mov  $1, %ax
1   198       0     1   1001 xchg %ax, mutex
1   198       0     1   1002 test $0, %ax
1   198       0     1   1003 jne  .acquire
1   198     198     1   1004 mov  count, %ax
1   198     199     1   1005 add  $1, %ax
1   199     199     1   1006 mov  %ax, count
0   199     199     1   1007 mov  $0, mutex
0   199     199     0   1008 sub  $1, %bx
0   199     199     0   1009 test $0, %bx
0   199     199     0   1010 jgt .top
0   199     198     1   ------ Interrupt ------  ------ Interrupt ------
0   199       1     1                           1000 mov  $1, %ax
1   199       0     1                           1001 xchg %ax, mutex
1   199       0     1                           1002 test $0, %ax
1   199       0     1                           1003 jne  .acquire
1   199     199     1                           1004 mov  count, %ax
1   199     200     1                           1005 add  $1, %ax
1   200     200     1                           1006 mov  %ax, count
0   200     200     1                           1007 mov  $0, mutex
0   200     200     0                           1008 sub  $1, %bx
0   200     200     0                           1009 test $0, %bx
0   200     200     0                           1010 jgt .top
0   200     199     0   ------ Interrupt ------  ------ Interrupt ------
0   200     199     0   1011 halt
0   200     200     0   ----- Halt;Switch -----  ----- Halt;Switch -----
0   200     200     0                           1011 halt
```

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).

```
mutex count      ax    bx          Thread 0                Thread 1

    0     0       0     3
    0     0       1     3   1000 mov  $1, %ax
    1     0       0     3   1001 xchg %ax, mutex
    1     0       0     3   ------ Interrupt ------  ------ Interrupt ------
    1     0       1     3                           1000 mov  $1, %ax
    1     0       1     3                           1001 xchg %ax, mutex
    1     0       0     3  ------ Interrupt ------   ------ Interrupt ------
    1     0       0     3   1002 test $0, %ax
    1     0       0     3   1003 jne  .acquire
    1     0       1     3   ------ Interrupt ------  ------ Interrupt ------
    1     0       1     3                           1002 test $0, %ax
    1     0       1     3                           1003 jne  .acquire
    1     0       0     3  ------ Interrupt ------   ------ Interrupt ------
    1     0       0     3   1004 mov  count, %ax
    1     0       1     3   1005 add  $1, %ax
    1     0       1     3   ------ Interrupt ------  ------ Interrupt ------
    1     0       1     3                           1000 mov  $1, %ax
    1     0       1     3                           1001 xchg %ax, mutex
    1     0       1     3    ------ Interrupt ------ ------ Interrupt ------
    1     1       1     3   1006 mov  %ax, count
    0     1       1     3   1007 mov  $0, mutex
```

Um dieses Szenario zu erreichen, darf jeweils ein Thread zwei Befehle ausführen.
Der Code agiert genauso, wie er es sollte.

```
./x86.py -p test-and-set.s -M mutex,count -R ax,bx -a bx=3 -P 0011 -c
```

Ein weiteres interessantes Szenario für mich war folgendes:

```
./x86.py -p test-and-set.s -M count,mutex -R ax -a bx=3 -P 001111111 -c
```

```
0     0       1   1000 mov  $1, %ax
0     1       0   1001 xchg %ax, mutex
0     1       0   ------ Interrupt ------  ------ Interrupt ------
0     1       1                           1000 mov  $1, %ax
0     1       1                           1001 xchg %ax, mutex
0     1       1                           1002 test $0, %ax
0     1       1                           1003 jne  .acquire
0     1       1                           1000 mov  $1, %ax
0     1       1                           1001 xchg %ax, mutex
0     1       1                           1002 test $0, %ax
0     1       0   ------ Interrupt ------  ------ Interrupt ------
0     1       0   1002 test $0, %ax
0     1       0   1003 jne  .acquire
0     1       1   ------ Interrupt ------  ------ Interrupt ------
0     1       1                           1003 jne  .acquire
0     1       1                           1000 mov  $1, %ax
0     1       1                           1001 xchg %ax, mutex
0     1       1                           1002 test $0, %ax
0     1       1                           1003 jne  .acquire
0     1       1                           1000 mov  $1, %ax
0     1       1                           1001 xchg %ax, mutex
0     1       0   ------ Interrupt ------  ------ Interrupt ------
0     1       0   1004 mov  count, %ax
0     1       1   1005 add  $1, %ax
0     1       1   ------ Interrupt ------  ------ Interrupt ------
0     1       1                           1002 test $0, %ax
0     1       1                           1003 jne  .acquire
0     1       1                           1000 mov  $1, %ax
0     1       1                           1001 xchg %ax, mutex
0     1       1                           1002 test $0, %ax
0     1       1                           1003 jne  .acquire
0     1       1                           1000 mov  $1, %ax
0     1       1   ------ Interrupt ------  ------ Interrupt ------
1     1       1   1006 mov  %ax, count
1     0       1   1007 mov  $0, mutex
```

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

```
./x86.py -p peterson.s -M flag,turn -a bx=0,bx=1 -i 5 -c
```

```
flag  turn          Thread 0                Thread 1

   0     0
   0     0   1000 lea flag, %fx
   0     0   1001 mov %bx, %cx
   0     0   1002 neg %cx
   0     0   1003 add $1, %cx
   1     0   1004 mov $1, 0(%fx,%bx,4)
   1     1   1005 mov %cx, turn
   1     1   1006 mov 0(%fx,%cx,4), %ax
   1     1   1007 test $1, %ax
   1     1   ------ Interrupt ------  ------ Interrupt ------
   1     1                           1000 lea flag, %fx
   1     1                           1001 mov %bx, %cx
   1     1                           1002 neg %cx
   1     1                           1003 add $1, %cx
   1     1                           1004 mov $1, 0(%fx,%bx,4)
   1     0                           1005 mov %cx, turn
   1     0                           1006 mov 0(%fx,%cx,4), %ax
   1     0                           1007 test $1, %ax
   1     0   ------ Interrupt ------  ------ Interrupt ------
   1     0   1008 jne .fini
   1     0   1012 mov count, %ax
   1     0   1013 add $1, %ax
   1     0   1014 mov %ax, count
   0     0   1015 mov $0, 0(%fx,%bx,4)
   0     1   1016 mov %cx, turn
   0     1   1017 halt
   0     1   ----- Halt;Switch -----  ----- Halt;Switch -----
   0     1                           1008 jne .fini
   0     1   ------ Interrupt ------  ------ Interrupt ------
   0     1                           1009 mov turn, %ax
   0     1                           1010 test %cx, %ax
   0     1                           1011 je .spin1
   0     1                           1012 mov count, %ax
   0     1                           1013 add $1, %ax
   0     1                           1014 mov %ax, count
   0     1                           1015 mov $0, 0(%fx,%bx,4)
   0     0                           1016 mov %cx, turn
   0     0   ------ Interrupt ------  ------ Interrupt ------
   0     0                           1017 halt
```

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

```
./x86.py -p peterson.s -M flag,turn -a bx=0,bx=1 -P 000000111111
```

```
flag  turn count          Thread 0                Thread 1

   0     0     0
   0     0     0   1000 lea flag, %fx
   0     0     0   1001 mov %bx, %cx
   0     0     0   1002 neg %cx
   0     0     0   1003 add $1, %cx
   1     0     0   1004 mov $1, 0(%fx,%bx,4)
   1     1     0   1005 mov %cx, turn
   1     1     0   ------ Interrupt ------  ------ Interrupt ------
   1     1     0                           1000 lea flag, %fx
   1     1     0                           1001 mov %bx, %cx
   1     1     0                           1002 neg %cx
   1     1     0                           1003 add $1, %cx
   1     1     0                           1004 mov $1, 0(%fx,%bx,4)
   1     0     0                           1005 mov %cx, turn
   1     0     0   ------ Interrupt ------  ------ Interrupt ------
   1     0     0   1006 mov 0(%fx,%cx,4), %ax
   1     0     0   1007 test $1, %ax
   1     0     0   1008 jne .fini
   1     0     0   1009 mov turn, %ax
   1     0     0   1010 test %cx, %ax
   1     0     0   1011 je .spin1
   1     0     0   ------ Interrupt ------  ------ Interrupt ------
   1     0     0                           1006 mov 0(%fx,%cx,4), %ax
   1     0     0                           1007 test $1, %ax
   1     0     0                           1008 jne .fini
   1     0     0                           1009 mov turn, %ax
   1     0     0                           1010 test %cx, %ax
   1     0     0                           1011 je .spin1
   1     0     0   ------ Interrupt ------  ------ Interrupt ------
   1     0     0   1012 mov count, %ax
   1     0     0   1013 add $1, %ax
   1     0     1   1014 mov %ax, count
   0     0     1   1015 mov $0, 0(%fx,%bx,4)
   0     1     1   1016 mov %cx, turn
   0     1     1   1017 halt
   0     1     1   ----- Halt;Switch -----  ----- Halt;Switch -----
   0     1     1                           1006 mov 0(%fx,%cx,4), %ax
   0     1     1                           1007 test $1, %ax
   0     1     1                           1008 jne .fini
   0     1     1                           1012 mov count, %ax
   0     1     1                           1013 add $1, %ax
   0     1     2                           1014 mov %ax, count
   0     1     2                           1015 mov $0, 0(%fx,%bx,4)
   0     0     2                           1016 mov %cx, turn
   0     0     2                           1017 halt
```

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

```
1992  1993  1992    1991    2  1991   1011 test $0, %bx
1992  1993  1992    1991    2  1991   1012 jgt .top
1992  1993  1992       1    2  1991   1000 mov $1, %ax
1992  1994  1992    1993    2  1991   1001 fetchadd %ax, ticket
1992  1994  1992    1993    2  1992   1002 mov turn, %cx
1992  1994  1992    1993    2  1992   1003 test %cx, %ax
1992  1994  1992    1993    2  1992   1004 jne .tryagain
1992  1994  1992    1993    2  1992   1002 mov turn, %cx
1992  1994  1992    1993    2  1992   1003 test %cx, %ax
1992  1994  1992    1993    2  1992   1004 jne .tryagain
1992  1994  1992    1993    2  1992   1002 mov turn, %cx
1992  1994  1992    1993    2  1992   1003 test %cx, %ax
1992  1994  1992    1993    2  1992   1004 jne .tryagain
1992  1994  1992    1993    2  1992   1002 mov turn, %cx
1992  1994  1992    1993    2  1992   1003 test %cx, %ax
1992  1994  1992    1993    2  1992   1004 jne .tryagain
1992  1994  1992    1993    2  1992   1002 mov turn, %cx
1992  1994  1992    1993    2  1992   1003 test %cx, %ax
1992  1994  1992    1993    2  1992   1004 jne .tryagain
1992  1994  1992    1993    2  1992   1002 mov turn, %cx
1992  1994  1992    1993    2  1992   1003 test %cx, %ax
1992  1994  1992    1993    2  1992   1004 jne .tryagain
1992  1994  1992    1993    2  1992   1002 mov turn, %cx
1992  1994  1992    1993    2  1992   1003 test %cx, %ax
1992  1994  1992    1993    2  1992   1004 jne .tryagain
1992  1994  1992    1993    2  1992   1002 mov turn, %cx
1992  1994  1992    1993    2  1992   1003 test %cx, %ax
1992  1994  1992    1993    2  1992   1004 jne .tryagain
1992  1994  1992    1993    2  1992   1002 mov turn, %cx
1992  1994  1992    1993    2  1992   1003 test %cx, %ax
1992  1994  1992    1993    2  1992   1004 jne .tryagain
1992  1994  1992    1993    2  1992   1002 mov turn, %cx
1992  1994  1992    1993    2  1992   1003 test %cx, %ax
1992  1994  1992    1993    2  1992   1004 jne .tryagain
1992  1994  1992    1993    2  1992   1002 mov turn, %cx
1992  1994  1992    1993    2  1992   1003 test %cx, %ax
1992  1994  1992    1993    2  1992   1004 jne .tryagain
1992  1994  1992    1993    2  1992   1002 mov turn, %cx
1992  1994  1992    1993    2  1992   1003 test %cx, %ax
1992  1994  1992    1992    6  1991   ------ Interrupt ------  ------ Interrupt ------
1992  1994  1992    1992    6  1991                            1003 test %cx, %ax
```

```
./x86.py -p ticket.s -M count,ticket,turn -R ax,bx,cx -a bx=1000 -c
```

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.

```
4995  4999  4995    4995    2  4992   ------ Interrupt ------  ------ Interrupt ------  ------ Interrupt ------  ------ Interrupt ------  ------ Interrupt ------
4995  4999  4995    4995    2  4992                                                                                                     1004 jne .tryagain
4995  4999  4995    4995    2  4995                                                                                                     1002 mov turn, %cx
4995  4999  4995    4995    2  4995                                                                                                     1003 test %cx, %ax
4995  4999  4995    4995    2  4995                                                                                                     1004 jne .tryagain
4995  4999  4995    4995    2  4995                                                                                                     1005 mov  count, %ax
4995  4999  4995    4996    2  4995                                                                                                     1006 add  $1, %ax
4996  4999  4995    4996    2  4995                                                                                                     1007 mov  %ax, count
4996  4999  4995       1    2  4995                                                                                                     1008 mov $1, %ax
4996  4999  4996    4995    2  4995                                                                                                     1009 fetchadd %ax, turn
4996  4999  4996    4995    1  4995                                                                                                     1010 sub  $1, %bx
4996  4999  4996    4995    1  4995                                                                                                     1011 test $0, %bx
4996  4999  4996    4995    1  4995                                                                                                     1012 jgt .top
4996  4999  4996       1    1  4995                                                                                                     1000 mov $1, %ax
4996  5000  4996    4999    1  4995                                                                                                     1001 fetchadd %ax, ticket
4996  5000  4996    4999    1  4996                                                                                                     1002 mov turn, %cx
4996  5000  4996    4999    1  4996                                                                                                     1003 test %cx, %ax
4996  5000  4996    4999    1  4996                                                                                                     1004 jne .tryagain
4996  5000  4996    4999    1  4996                                                                                                     1002 mov turn, %cx
4996  5000  4996    4999    1  4996                                                                                                     1003 test %cx, %ax
4996  5000  4996    4999    1  4996                                                                                                     1004 jne .tryagain
4996  5000  4996    4999    1  4996                                                                                                     1002 mov turn, %cx
4996  5000  4996    4999    1  4996                                                                                                     1003 test %cx, %ax
4996  5000  4996    4999    1  4996                                                                                                     1004 jne .tryagain
4996  5000  4996    4999    1  4996                                                                                                     1002 mov turn, %cx
4996  5000  4996    4999    1  4996                                                                                                     1003 test %cx, %ax
4996  5000  4996    4999    1  4996                                                                                                     1004 jne .tryagain
4996  5000  4996    4999    1  4996                                                                                                     1002 mov turn, %cx
4996  5000  4996    4999    1  4996                                                                                                     1003 test %cx, %ax
4996  5000  4996    4999    1  4996                                                                                                     1004 jne .tryagain
4996  5000  4996    4999    1  4996                                                                                                     1002 mov turn, %cx
4996  5000  4996    4999    1  4996                                                                                                     1003 test %cx, %ax
4996  5000  4996    4999    1  4996                                                                                                     1004 jne .tryagain
4996  5000  4996    4999    1  4996                                                                                                     1002 mov turn, %cx
4996  5000  4996    4999    1  4996                                                                                                     1003 test %cx, %ax
4996  5000  4996    4999    1  4996                                                                                                     1004 jne .tryagain
4996  5000  4996    4999    1  4996                                                                                                     1002 mov turn, %cx
4996  5000  4996    4999    1  4996                                                                                                     1003 test %cx, %ax
4996  5000  4996    4999    1  4996                                                                                                     1004 jne .tryagain
```

```
./x86.py -p ticket.s -M count,ticket,turn -R ax,bx,cx -a bx=1000 -t 5 -c
```

28.13

```
./x86.py -p test-and-set.s -M count -P 00011111111111 -c
```

```
./x86.py -p yield.s -M count -P 00011111111111 -c
```

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.

```
.acquire
mov  mutex, %ax
test $0, %ax
jne .acquire
mov  $1, %ax
xchg %ax, mutex
test $0, %ax
jne .acquire
```