Betrachten Sie folgenden x86 Code:

```
.var flag   2
.var turn
.var count

.main

lea flag, %fx

mov %bx, %cx
neg %cx
add $1, %cx

.acquire
mov $1, 0(%fx,%bx,4)
mov %cx, turn

.spin1
mov 0(%fx,%cx,4), %ax
test $1, %ax
jne .fini

.spin2
mov turn, %ax
test %cx, %ax
je .spin1

.fini

mov count, %ax
add $1, %ax
mov %ax, count

.release
mov $0, 0(%fx,%bx,4)


mov %cx, turn
halt
```

Fragen:

a.) Um welche Art der Synchronisierung handelt es sich hier?

→ Peterson's Algorithmus

b.) Was ist die ‚besondere' Aufgabe der turn Variablen?

→ sie gibt an, welcher Prozess dran ist, um das lock zu bekommen und verhindert so Deadlocks

c.) Wie müssen **ax** und **bx** Register gesetzt werden, so dass der Code funktioniert (**Erklärung**) ?

ax = ist ein temporäres Register (nichts festes gesetzt)

bx = ID des aktuellen Thread (0, 1)

Betrachten Sie folgende ‚Bounded Buffer' Lösung:

```
int buffer[max];

void *producer(void *arg) {
  for (i=0; i<loops; i++) {
    Pthread_mutex_lock(&mutex) ;              // p1
    while (count == max)                      // p2
       Pthread_cond_wait(&empty, &mutex);     // p3
    put(i);                                   // p4
    Pthread_cond_signal(&fill);               // p5
    Pthread_mutex_unlock(&mutex) ;            // p6
  }
}

void *consumer(void *arg) {
  while (1) {
      Pthread_mutex_lock(&mutex) ;            // c1
      while (count == 0)                      // c2
        Pthread_cond_wait(&fill, &mutex);     // c3
      int tmp = get();                        // c4
      Pthread_cond_signal(&empty) ;           // c5
      Pthread_mutex_unlock(&mutex);           // c6
      printf("%d\n", tmp);
      }
  }
```

Gehen Sie im Weiteren von folgenden Annahmen aus:

- **ein Thread blockiert nur dann, wenn er entweder durch eine Condition Variable oder einen Lock (Mutex) blockiert wird.**
- **Mögliche Threadwechsel durch Interrupts, wie z.B. Ablauf der Zeitschreibe (Scheduling) tritt nicht auf!**

Ein möglicher Trace (Zeit verläuft von links nach rechts) von den beiden Threads Producer ‚a' und Consumer ‚a' kann folgendermassen angegeben werden:

```
Thread Pa: 1,2,4,5,6,1,2,3
Thread Ca:                  1,2,4,5,6,1,2
```

Fragen [6P gesamt]: Geben Sie im weiteren den Trace für folgende Szenarien an:

Trace 1 (0,5P):

Ein Producer (*Pa*) und ein Consumer (*Ca*), `max=1`

Pa startet zuerst. Ablauf stoppt, wenn Consumer *Ca* ein Element ‚konsumiert' hat.

Thread Pa: 12456123

Thread Ca: 124

Trace 2 (0,5P):

Ein Producer (*Pa*) und ein Consumer (*Ca*), `max=3`

Pa startet zuerst. Ablauf stoppt, wenn Consumer *Ca* ein Element ‚konsumiert' hat.

Thread Pa: 12456 12456 12456 123

Thread Ca: 124

Trace 3 (1P):

Ein Producer (*Pa*) und ein Consumer (*Ca*), `max=1`

Ca startet zuerst. Ablauf stoppt, wenn Consumer *Ca* ein Element ‚konsumiert' hat.

Thread Pa: 12456123

Thread Ca: 123 24

Trace 4 (1P):

Ein Producer (*Pa*) und zwei Consumer (*Ca*,*Cb*), `max=1`

Ca startet zuerst, dann Cb und dann Pa. Ablauf stoppt, wenn Producer P*a* ein Element ‚produziert' hat.

Thread Pa: 1 2 4

Thread Ca: 1 2 3

Thread Cb: 1 2 3

Trace 5 (1,5P):

Die while Loops im Code werden für diese Trace Aufgabe mit if Statements ersetzt.

Zeigen Sie mit einem Producer (*Pa*) und zwei Consumer (*Ca*,*Cb*) und einem `max=` Wert Ihrer Wahl, dass dies zu einem Problem führt.

setzt Ca in READY, aber Cb läuft zuerst und wenn Ca läuft ist Buffer leer

max = 1

Thread Pa: 1 2 4 5 6 1 2 3

Thread Ca: 1 2 3

4 → Deadlock

Thread Cb: 1 2 4 5 6 1 2 3

Trace 6 (1,5P):

Nun werden wieder while Loops im Code verwendet, aber nur eine(!) Condition Variable statt der bisherigen 2 CVs.

Zeigen Sie mit einem Producer (*Pa*) und zwei Consumer (*Ca*,*Cb*) und einem `max=` Wert Ihrer Wahl, dass dies zu einem Problem führen kann (Erklärung!)

max = 1

Thread Pa: 1 2 4 5 6 1 2 3

Thread Ca: 1 2 3

1 2 4 5 6 1 2 3 → 50%

1 2 3 → Deadlock

Thread Cb:

Erklärung:

Producer und Consumer verwenden selbe CV und dann gibt es eine 50/50 ob der Producer (alles OK) oder Consumer (Deadlock) geweckt wird