Gruppe 15 | Tobias Schoch, Luis Nothvogel

Simulation wurde auf dem HTWG Container ausgeführt

30.2

Buffergröße:

Das Verhalten mit einem größeren Buffer verändert sich in dem Sinne, dass der Producer mehr Werte in seinen Buffer schreiben kann bevor er schlafen geht.

```
0 [*---  ---  --- ] p1
1 [u  3 f---  --- ] p4
1 [u  3 f---  --- ] p5
1 [u  3 f---  --- ] p6
1 [u  3 f---  --- ] p0
1 [u  3 f---  --- ] p1
2 [u  3    4 f--- ] p4
2 [u  3    4 f--- ] p5
2 [u  3    4 f--- ] p6
2 [u  3    4 f--- ] p0
2 [u  3    4 f--- ] p1
3 [*  3    4    5 ] p4
3 [*  3    4    5 ] p5
3 [*  3    4    5 ] p6
3 [*  3    4    5 ] p0
3 [*  3    4    5 ] p1
3 [*  3    4    5 ] p2
```

```
to852sch@ct-bsys-ss20-15
 NF        P0 C0 C1
  0 [*--- ] p0
  0 [*--- ]       c0
  0 [*--- ]    c0
  0 [*--- ] p1
  1 [*  0 ] p4
  1 [*  0 ] p5
  1 [*  0 ] p6
  1 [*  0 ]    c1
  0 [*--- ]    c4
```

Dadurch können die Consumer ebenfalls erst mehrere Werte verarbeiten, bevor diese wiederrum schlafen gehen. Das erhöht die Effizienz, da es so weniger Context Switches gibt.

Hier sieht man den Unterschied zwischen 2 verschiedenen Ausführungen des Simulators. Bei der linken Abbildung sieht man, wie der Buffer (mit der Größe 5) so lange von dem Producer gefüllt wird, bis der Buffer ganz voll ist. Im Anschluss gibt es einen Context Switch zu einem Consumer. Bei der rechten Ausführung ist der Buffer kleiner und damit auch die Anzahl der Context Switches größer.

Buffergröße & Anzahl der Produkte:

Erster Versuch: `./main-two-cvs-while -p 1 -c 1 -l 1 -m 1 -C 0,0,0,0,0,0,1 -v`
Buffergröße ist 1, die Anzahl der Produkte die der Producer herstellt ist ebenfalls 1.

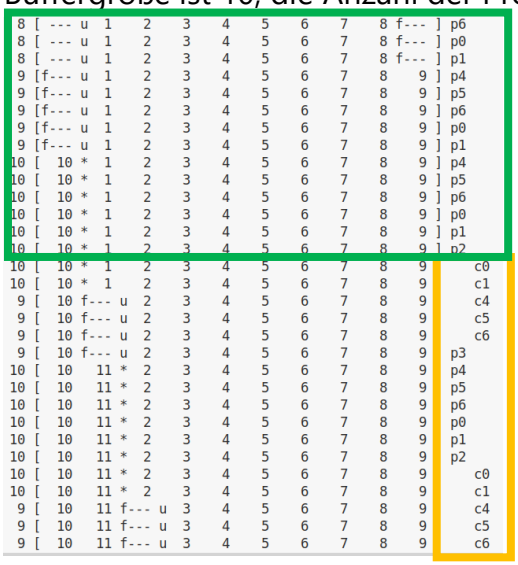
```
NF        P0 C0
 0 [*--- ] p0
 0 [*--- ]    c0
 0 [*--- ] p1
 1 [*  0 ] p4
 1 [*  0 ] p5
 1 [*  0 ] p6
 1 [*  0 ]    c1
 0 [*--- ]    c4
 0 [*--- ]    c5
 0 [*--- ]    c6
 1 [*EOS ] [main: added end-of-stream marker]
 1 [*EOS ]    c0
 1 [*EOS ]    c1
 0 [*--- ]    c4
 0 [*--- ]    c5
 0 [*--- ]    c6
```

Da nur ein Produkt hergestellt wird, brauchen wir auch nur eine maximale Buffergröße von 1.

Daher ist num_full auch 1.

Zweiter Versuch: `./main-two-cvs-while -p 1 -c 1 -l 100 -m 10 -C 0,0,0,0,0,0,1 -v`

Buffergröße ist 10, die Anzahl der Produkte die der Producer herstellt ist 100.



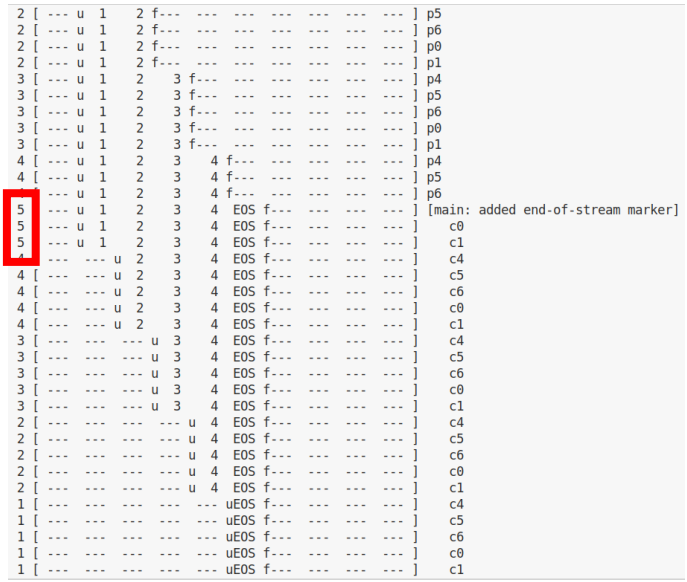
Der Producer versucht so schnell wie möglich den Buffer komplett aufzufüllen (grün markiert), nachdem der Consumer einmal einen Wert aus dem Buffer nimmt und für 1s verarbeitet. Nachdem der Buffer ganz voll ist, gibt es nun einen stetigen Schlagabtausch (orange markiert) zwischen Consumer, der nach jeder Ausführung eine Sekunde warten muss und Producer der im Anschluss den Buffer wieder direkt auffüllt.

Dies geht solange, bis alle Produkte hergestellt wurden vom Producer und der Consumer die letzten Produkte aus dem Buffer verarbeitet.

Daher schwankt num_full logischerweise meistens stets zwischen 9 und 10 Elementen im Buffer.

Dritter Versuch: `./main-two-cvs-while -p 1 -c 1 -l 5 -m 10 -C 0,0,0,0,0,0,1 -v`

Buffergröße ist 10, die Anzahl der Produkte die der Producer herstellt ist 5.



Hier sehen wir ein sehr ähnliches Muster wie zuvor beim zweiten Versuch. Nach der ersten Ausführung des Consumer schläft dieser für 1s. Der Producer nutzt die Situation, um den Buffer so voll zu machen wie möglich. Der Buffer wird logischerweise nicht ganz gefüllt, da wir nur 5 Werte haben + den EOS Wert.

Daher steigt num_full auf einen Bestand von 5 (rot markiert) und sinkt anschließend wieder langsam ab, da der Producer keine Arbeit mehr hat und der Consumer nun in Ruhe alle Produkte im Buffer abarbeiten kann.

30.4

```
to852sch@ct-bsys-ss20-15:~/Desktop$ ./main-two-cvs-while -p 1 -c 3 -m 1 -C 0,0,0,1,0,0,0:0,0,0,1,0,0,
0,:0,0,0,1,0,0,0 -l 10 -v
```

Vermutung: Da der Producer insgesamt mit „-l 10“ 10 Produkte herstellt, ist die Laufzeit des Programmes mindestens 10 Sekunden lang, da die Produkte eine Sekunde lang in der Instruktion C3 vom Consumer „verarbeitet“ werden.

Dies wird definiert mit -C 0,0,0,1,0,0: … Dabei ist die erste Stelle C0, die zweite C1 usw. Wenn man also nun bei cond_wait() angekommen ist, dann schläft der Consumer für 1 Sekunde.

```
  while (tmp != END_OF_STREAM) { c0;
Mutex_lock(&m);                  c1;
while (num_full == 0) {          c2;
    Cond_wait(&fill, &m);        c3;
      }
```

Test: Wenn wir das Programm ausführen mit dem Tag -t erhalten wir die Laufzeit von 13.01 Sekunden: `Total time: 13.01 seconds`

Warum also erhalten wir bei 10 Ausführungen eine Zeit von mehr als 13 Sekunden?

Dies kommt dadurch, dass zum Schluss noch die 3 Consumer als beendet markiert werden mit einem end-of-stream marker und zu Ende laufen.

Wenn wir von der Grundzeit der Wiederholungen ausgehen ohne die abschließenden c3’s (blau umrandet), haben wir eine Zeit von 10.01 Sekunden.

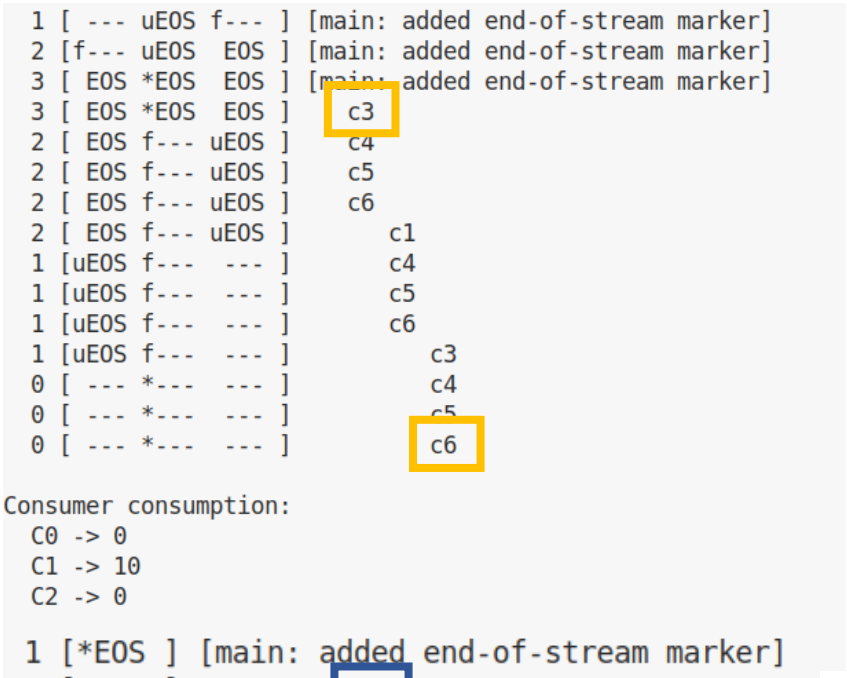
```
1 [*EOS ] [main: added end-of-stream marker]
1 [*EOS ]          c3
0 [*--- ]          c4
0 [*--- ]          c5
0 [*--- ]          c6
0 [*--- ]       c1
0 [*--- ]       c2
1 [*EOS ] [main: added end-of-stream marker]
1 [*EOS ]    c3
0 [*--- ]    c4
0 [*--- ]    c5
0 [*--- ]    c6
1 [*EOS ] [main: added end-of-stream marker]
1 [*EOS ]       c3
0 [*--- ]       c4
0 [*--- ]       c5
0 [*--- ]       c6
```

30.5

Vermutung: Das Programm sollte mit dem größeren Buffer keinen spürbaren zeitlichen Unterschied machen, da die Consumer immer noch jeweils 1 Sekunde warten müssen und lediglich der Producer durch den größeren Buffer mehr Produkte generieren kann bevor er sich schlafen legt.
Die Zeit sollte zudem über 10 Sekunden bleiben, da wir immer noch 10 Ausführungen haben.

Test:
```
Total time: 12.01 seconds
```

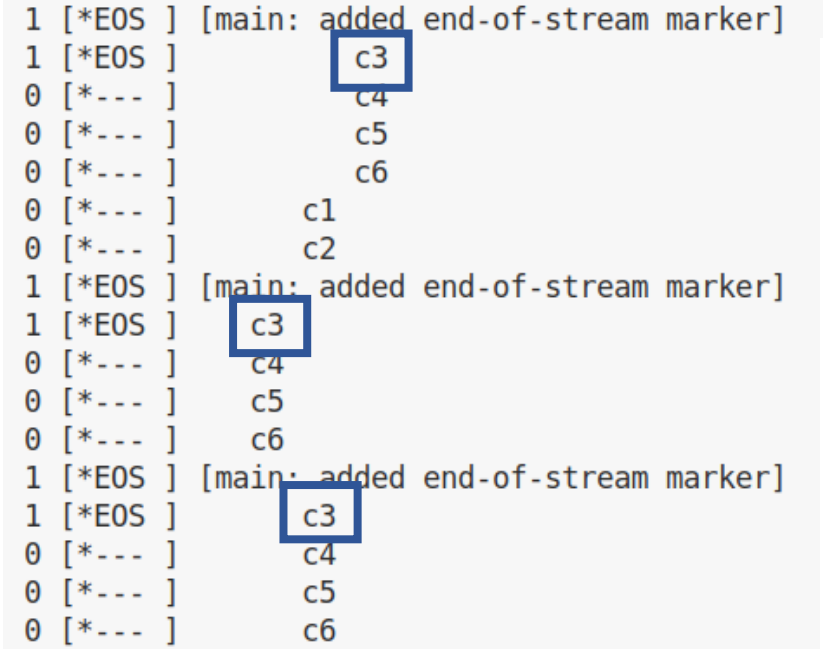
Allerdings haben wir nun eine geringere Laufzeit von 12.01 statt 13.01 Sekunden



Wenn wir hier ebenfalls die abschließenden 2 c3's ausblenden, haben wir ebenfalls eine Zeit von 10.01 Sekunden logischerweise.

Abbildung zeigt die Ausführung mit einer Buffer Größe von 3.

```
Consumer consumption:
  C0 -> 0
  C1 -> 10
  C2 -> 0
```



Diese kommt dadurch, dass wir nur zwei c3 Instruktionen (orange umrandet) haben beim EOS statt 3 wie zuvor (blau umrandet)

Abbildung zeigt Programmausführung mit einer Buffer Größe von 1.

30.6

Vermutung: Bisher hatten wir die Reihenfolge, dass ein Lock gesetzt wurde und im Lock hat der Consumer daraufhin die Produkte des Producers bearbeitet 1 Sekunde lang (sleep), woraufhin erst im Anschluss der Lock wieder befreit wurde.

Nun ist es so, dass der Consumer erst den Lock setzt, ihn im Anschluss freigibt und danach die Produkte bearbeitet 1 Sekunde lang bei c6 (rot umrandet).

Dadurch ist der Lock offen und ein anderer Consumer kann nun den Lock benutzen, während der andere schläft.

Da wir 3 Consumer haben, vermute ich, dass die 10 Wiederholungen auch 3x (~3,34 Sekunden | 3s-4s) so schnell sind, da die Consumer nicht mehr 1s auf die Freigabe des Locks warten müssen, sondern direkt nacheinander ohne große Wartezeiten den Lock reservieren können.

Nachdem die Consumer aufgeweckt werden, geht das Spiel von vorne los.

```
  while (tmp != END_OF_STREAM) { c0;
Mutex_lock(&m);              c1;
while (num_full == 0) {      c2;
    Cond_wait(&fill, &m);    c3;
      }
tmp = do_get();              c4;
Cond_signal(&empty);         c5;
Mutex_unlock(&m);            c6;
```

Test: `Total time: 5.00 seconds`

Wir erhalten eine Laufzeit von 5 Sekunden. In der 5-sekündigen Laufzeit ist auch die verwendete Zeit mit 1s für EOS mitinbegriffen. Daraus ergibt sich eine Laufzeit ohne EOS von 4 Sekunden.

30.7

Vermutung: Da weiterhin lediglich die Consumer die Produkte abarbeiten mit einer Geschwindigkeit von 1s und die Producer keine „Produktionsverzögerung“ haben, sollte sich durch den größere Buffer keine Zeitänderung ergeben.

Es könnte eine kleine Zeitersparnis geben, dadurch dass weniger Context Switches stattfinden, denn durch den größeren Buffer kann der Producer nun mehr Produkte generieren bevor er sich schlafen legt.

Test: `Total time: 5.00 seconds`

Wir erhalten dieselbe Laufzeit von 5 Sekunden, da wie bereits erwähnt der Producer zwar den Buffer vollmacht, aber dennoch stets 1s gewartet werden muss bis die Produkte abgearbeitet wurden durch den Consumer.

```
0 [*---  ---  --- ] p1
1 [u  3 f---  --- ] p4
1 [u  3 f---  --- ] p5
1 [u  3 f---  --- ] p6
1 [u  3 f---  --- ] p0
1 [u  3 f---  --- ] p1
2 [u  3    4 f--- ] p4
2 [u  3    4 f--- ] p5
2 [u  3    4 f--- ] p6
2 [u  3    4 f--- ] p0
2 [u  3    4 f--- ] p1
3 [*  3    4    5 ] p4
3 [*  3    4    5 ] p5
3 [*  3    4    5 ] p6
3 [*  3    4    5 ] p0
3 [*  3    4    5 ] p1
3 [*  3    4    5 ] p2
```

Hier sehen wir, wie der größere Buffer gefüllt wird durch den Producer, aber dennoch dieselbe Laufzeit erzielt wird durch die Wartezeiten.

30.8

Nein, das Problem mit sleep strings zu verursachen wäre mit diesem Beispiel nicht möglich, denn dafür bräuchte man entweder mehr als 1 Producer bzw. mehr als 1 Consumer und wir haben lediglich ein Consumer und ein Producer.

Das Problem wird verursacht, da wir nur eine Condition Variable haben. Nehmen wir an, wir haben 2 Consumer und einen Producer. Wenn nun die beiden Consumer sich schlafen legen, weil der Buffer leer ist, legt der Producer einen Wert in den Buffer und aktiviert einen anderen Thread (Consumer 1). Consumer 1 wiederrum verarbeitet den Wert, leert den Buffer und aktiviert den Consumer 2. Im Anschluss legt Consumer 1 sich schlafen. Danach prüft Consumer 2 den Buffer, welcher leer ist und legt sich auch schlafen.

Dies ist jedoch in der Aufgabenstellung mit einem Producer und einem Consumer nicht möglich.

30.9

Ja, dies ist mit der Aufgabenstellung möglich da wir nun mind. 2 Threads derselben Klasse haben.

```
./main-one-cv-while -p 1 -c 2 -m 1 -v
```

Zuerst haben wir versucht das Programm mit einem Producer, zwei Consumer und einem Buffer der Größe eins laufen zu lassen. Das hat nicht funktioniert.

Als nächstes haben wir versucht die Anzahl der „Produkte" des Producers zu erhöhen mit -l auf 1000.

```
to852sch@ct-bsys-ss20-15:~/Desktop$ ./main-one-cv-while -p 1 -c 2 -m 1 -l 1000 -v
```

```
  1 [*999 ] p4
  1 [*999 ] p5
  1 [*999 ] p6
  1 [*999 ]    c1
  0 [*--- ]    c4
  0 [*--- ]    c5
  0 [*--- ]       c3
  0 [*--- ]    c6
  0 [*--- ]       c2
  0 [*--- ]    c0
  0 [*--- ]    c1
  0 [*--- ]    c2
  1 [*EOS ] [main: added end-of-stream marker]
  1 [*EOS ]       c3
  0 [*--- ]       c4
  0 [*--- ]       c5
  0 [*--- ]       c6
  0 [*--- ]    c3
  0 [*--- ]    c2
```

Dies hat bei der zweiten Ausführung funktioniert. Dadurch erhalten wir allerdings keinen garantierten Fehler, denn der Fehler passiert nur durch Zufall und passiert nur alle paar Ausführungen (nicht deterministisch).

Consumer 1 weckt Consumer 2 auf (orange markiert).

```
./main-one-cv-while -p 1 -c 2 -m 1 -P 0,0,0,0,0,0,1 -v
```

```
to852sch@ct-bsys-ss20-15:~/Desktop$ ./main-one-cv-while -p 1 -c 2 -m 1  -P 0,0,0,0,0,0,1 -v
 NF        P0 C0 C1
  0 [*--- ] p0
  0 [*--- ]       c0
  0 [*--- ]    c0
  0 [*--- ] p1
  1 [*  0 ] p4
  1 [*  0 ] p5
  1 [*  0 ] p6
  1 [*  0 ]    c1
  0 [*--- ]    c4
  0 [*--- ]    c5
  0 [*--- ]    c6
  0 [*--- ]       c1
  0 [*--- ]       c2
  0 [*--- ]    c0
  0 [*--- ]    c1
  0 [*--- ]    c2
  1 [*EOS ] [main: added end-of-stream marker]
  1 [*EOS ]       c3
  0 [*--- ]       c4
  0 [*--- ]       c5
  0 [*--- ]       c6
  0 [*--- ]    c3
  0 [*--- ]    c2
```

Im Anschluss haben wir versucht, den Producer nach der Lock Befreiung eine Sekunde zu beschäftigen (p6), so dass die Consumer sich nur gegenseitig aufwecken können(c5), was auch passiert (rot markiert).

Im Anschluss haben sich dann die einzelnen Threads schlafen gelegt, da der Buffer nicht mehr aufgefüllt wurde durch den Producer.

30.10

Mit lediglich einem Consumer und einem Producer kann der Fehler nicht reproduziert werden. Der Fehler wird hervorgerufen, indem der Consumer 1 ein leeren Buffer sieht und danach sich schlafen legt und auf ein Signal wartet. Davor lässt er den Lock wieder frei. Wenn nun der Producer einen Wert in den Buffer legt, könnte ein zweiter Consumer kommen und diesen Wert auslesen und aus dem Buffer nehmen. Wenn nun Consumer 1 wiederkommt und auf den Buffer zugreifen will in der if Schleife, dann liest er aus einem leeren Buffer.

Da es mindestens 2 Consumer geben muss um das Problem zu reproduzieren, kann das Szenario nicht erschaffen werden mit einem Consumer und einem Producer.

```
to852sch@ct-bsys-ss20-15:~/Desktop$ ./main-two-cvs-if -p 1 -c 2 -m 1 -v
 NF        P0 C0 C1
  0 [*--- ] p0
  0 [*--- ]    c0
  0 [*--- ] p1
  0 [*--- ]       c0
  1 [*  0 ] p4
  1 [*  0 ] p5
  1 [*  0 ] p6
  1 [*  0 ]    c1
  0 [*--- ]    c4
  0 [*--- ]    c5
  0 [*--- ]    c6
  0 [*--- ]       c1
  0 [*--- ]    c0
  0 [*--- ]       c2
  1 [*EOS ] [main: added end-of-stream marker]
  1 [*EOS ]    c1
  0 [*--- ]    c4
  0 [*--- ]    c5
  0 [*--- ]    c6
  0 [*--- ]       c3
error: tried to get an empty buffer
```

```
void *consumer(void *arg) {
    int id = (int) arg;
    int tmp = 0;
    int consumed_count = 0;
    while (tmp != END_OF_STREAM) { c0;
  Mutex_lock(&m);              c1;
  if (num_full == 0) {         c2;
      Cond_wait(&fill, &m);    c3;
        }
  tmp = do_get();              c4;
  Cond_signal(&empty);         c5;
  Mutex_unlock(&m);            c6;
  consumed_count++;
    }
```

Zuerst haben wir versucht diesen Error zu reproduzieren mit simplen Flags.
Wenn man das Programm laufen lässt, dann erhält man nach ein paar Ausführungen unseren gesuchten Error. Diese Parameter liefern uns jedoch nicht einen deterministischen Fehler. c3 wird noch ausgeführt, aber bei c4 wird ein leerer Buffer gelesen, was zu einem Error führt (rot umrandet).

```
to852sch@ct-bsys-ss20-15:~/Desktop$ ./main-two-cvs-if -p 1 -c 2 -m 1 -l 10 -P 1,0,0,0,0,0,0 -C 1,0,0,0,0,0,0:0,0,0,1,0,0,0 -v
```

```
 NF        P0 C0 C1
  0 [*--- ]       c0
  0 [*--- ]    c0
  0 [*--- ] p0
  0 [*--- ]       c1
  0 [*--- ]       c2
  0 [*--- ]    c1
  0 [*--- ]    c2
  0 [*--- ] p1
  1 [*  0 ] p4
  1 [*  0 ] p5
  1 [*  0 ] p6
  1 [*  0 ]       c3
  1 [*  0 ] p0
  0 [*--- ]       c4
  0 [*--- ]       c5
  0 [*--- ]       c6
  0 [*--- ] p1
  0 [*--- ]       c0
  1 [*  1 ] p4
  1 [*  1 ] p5
  1 [*  1 ] p6
  1 [*  1 ]       c1
  1 [*  1 ] p0
  0 [*--- ]       c4
  0 [*--- ]       c5
  0 [*--- ]       c6
  0 [*--- ]    c3
error: tried to get an empty buffer
```

Für ein deterministisches Ergebnis, haben wir sowohl p0 des Producers, als auch c0 des ersten Consumers eine Sekunde schlafen gelegt, dass der erste Consumer an der if Schleife war um später dorthin zurückzukehren. Als der andere Consumer bereits den Wert aus dem Buffer rausgezogen hat und sich damit der State geändert hat, kommt der erste Consumer und versucht auch einen Wert auszulesen. Dies endet jedoch in einem Error, da der Buffer durch den anderen Consumer geleert wurde.

30.11

Wenn man den Lock freigibt und im Anschluss put() oder get() ausgeführt wird, könnte es einen Timer Interrupt während der Ausführung der Methoden geben. Die put() und get() Methoden gehören allerdings zur Critical Section, da über put() Werte in den Buffer geschrieben werden und get() die Daten ausliehst und im Anschluss aus dem Buffer entfernt. Somit wäre es eine schlechte Sache, wenn es zu einer Race Condition kommen würde und ein anderer Thread kommt um sich „vorzudrängeln".

```
./main-two-cvs-while-extra-unlock -p 1 -c 2 -C 0,0,0,0,3,0,0:0,0,0,0,0,0,0 -v
```

```
NF       P0 C0 C1
 0 [*--- ] p0
 0 [*--- ] p1
 0 [*--- ]    c0
 1 [*  0 ] p4
 1 [*  0 ]    c1
 1 [*  0 ] p5
 0 [*--- ]    c4
 0 [*--- ] p6
 0 [*--- ]    c5
 1 [*EOS ] [main: added end-of-stream marker]
 1 [*EOS ]    c6
 1 [*EOS ]       c0
 1 [*EOS ]    c0
 1 [*EOS ]       c1
 0 [*--- ]       c4
 0 [*--- ]    c1
 0 [*--- ]    c2
 0 [*--- ]       c5
 1 [*EOS ] [main: added end-of-stream marker]
 1 [*EOS ]    c3
 1 [*EOS ]       c6
 0 [*--- ]    c4
 0 [*--- ]    c5
 0 [*--- ]    c6

Consumer consumption:
  C0 -> 1
  C1 -> 0
```

Zuerst haben wir versucht c4 zum Schlafen zu bringen. Das Programm hat allerdings ohne Fehler zu Ende kompiliert. Wenn man dann genauer darüber nachdenkt und sich den Code anschaut, fällt einem folgendes auf:

```
  while (tmp != END_OF_STREAM) { c0;
Mutex_lock(&m);              c1;
while (num_full == 0) {      c2;
    Cond_wait(&fill, &m);    c3;
      }
Mutex_unlock(&m);
tmp = do_get();              c4;
Mutex_lock(&m);
Cond_signal(&empty);         c5;
Mutex_unlock(&m);            c6;
```

(grün umrandet) die do_get() Methode ist in der Critical Section und außerhalb des Locks.

Da der Sleep immer nach der Ausführung der jeweiligen Instruktion durchgeführt wird, würde es nur gehen, wenn wir den sleep string innerhalb der do_get() bzw. do_fill() Methode verwenden um dort eine Race Condition hervorzurufen.

Da in dem Code bei do_fill() und do_get() allerdings kein Punkt wie z.B. c0, p1, etc. gesetzt ist, können wir auch nicht mit sleep strings den Fehler hervorrufen.

Bei dem Thread des Producers sehen wir exakt dasselbe Muster wie zuvor oben. Do_fill() statt do_get() ist außerhalb des Locks.

```
  for (i = 0; i < loops; i++) {   p0;
Mutex_lock(&m);               p1;
while (num_full == max) {     p2;
    Cond_wait(&empty, &m);    p3;
}
Mutex_unlock(&m);
do_fill(base + i);            p4;
Mutex_lock(&m);
Cond_signal(&fill);           p5;
Mutex_unlock(&m);             p6;
```