

## 4. Übung zur Vorlesung „Betriebssysteme und Netzwerke“ (IBN)

Abgabedatum: 21.05.2019, 11:00 Uhr

---

### Aufgabe 1

(6 Punkte)

Implementieren Sie ein multi-threaded Programm, dessen Threads in einer Endlosschleife immer wieder um einen Mutex wettstreiten. In der kritischen Region jedes Threads passiert jeweils nichts, außer zu registrieren, dass der Mutex erfolgreich zugeteilt wurde (per Zähler). Einziger Parameter des Programms ist die Anzahl  $n$  der Threads, die das Programm starten soll.

Finden Sie heraus wie „fair“ es dabei zugeht: Wie oft wird jedem Thread der Mutex zugeteilt? Der Hauptthread sollte zuerst alle Threads starten, und daraufhin eine Sekunde warten, um sicherzugehen, dass alle Threads laufen. Dann setzt er alle Zähler zurück und legt sich 3 Sekunden schlafen. Anschließend gibt er die Zählerstände aus und beendet das Programm. **Bitte reichen Sie Ihr Programm in Moodle ein.**

- a) Führen Sie das Programm für 1, 2, 5, 10 und 100 Threads aus. Wie oft konnte der Mutex jeweils insgesamt erlangt werden? Ist diese Anzahl ungefähr gleich auf alle Threads verteilt?
- b) Die Fairness hängt auch davon ab, ob die wettstrebenden Threads auf verschiedenen CPUs oder Cores derselben CPU ausgeführt werden. Stellen Sie sicher, dass ihr Programm jetzt nur auf einem Core läuft (etwa durch den Befehl `pthread_setaffinity_np()`). Unter Linux können Sie auch einfach ihr Programm via `taskset -c 3 ./myProgram 5` ausführen. Das beschränkt die Ausführung auf den Core mit dem Index 3. Wiederholen Sie die Tests aus a). Wie wirkt sich diese Änderung auf die Fairness aus?

### Aufgabe 2

(3 Punkte)

Beim web-basierten Spiel *The Deadlock Empire*<sup>1</sup> nehmen Sie die Rolle des Schedulers ein, der die Ausführung von Threads steuert. Lösen Sie die 6 Level aus den Kapiteln *Unsynchronized Code* und *Locks*. Geben Sie an, wie Sie beide Threads in die kritische Region manövrieren bzw. wie Sie die Assertion auslösen. Erklären Sie, wann die Operation **Expand** zur Verfügung steht. Ist es realistisch, dass Sie als Scheduler in beliebiger Abfolge **Step** auslösen können?

---

<sup>1</sup><https://deadlockempire.github.io/>

### Aufgabe 3

(2 Punkte)

Vergegenwärtigen Sie sich die Hardwarelösung „Test and Set Lock“ zum wechselseitigen Ausschluss aus der Vorlesung 7. Beschreiben Sie folgende Situationen Zeile für Zeile anhand des Assemblercodes und beschreiben Sie dabei den Inhalt des Registers `RX` und der Variable `LOCK`. Beachten Sie dabei, dass der betrachtete Thread auch mit anderen Threads interagiert und gehen Sie darauf ein.

Zur Erinnerung: Der Befehl `CMP RX, #0` vergleicht den Inhalt von Register `RX` mit der Zahl 0. Der Befehl `JNE` führt einen bedingten Sprung (hier: zum Label `enter_region`) aus bei Nicht-Gleichheit (`NE`, non-equality) bei der letzten `CMP`-Operation. Der Befehl `RET` verursacht, dass die aktuelle Routine (hier: `enter_region`) verlassen wird und man zur Stelle ihres Aufrufs zurückkehrt.

1. Die gesamte Routine `enter_region` wird genau einmal durchlaufen.
2. Die Schleife in `enter_region` wird beim zweiten Durchlauf verlassen.

### Aufgabe 4

(2 Punkte)

Schauen Sie sich das Video "Lecture 3, Unit 2: using condition variables" aus Vorlesung 8 an und beantworten Sie folgende Fragen:

- a) Wie unterscheiden sich Semaphore im Bezug auf „Gedächtnis“ von Mutexen und Bedingungsvariablen? Was passiert, wenn ein Thread ein *signal* broadcastet, aber zu diesem Zeitpunkt kein Thread darauf wartet?
- b) Warum muss der Mutex zu *done* aufrechterhalten werden, wenn man das signal broadcastet?

### Aufgabe 5

(3 Punkte)

In der Vorlesung wurde gezeigt, wie Semaphore benutzt werden können, um den wechselseitigen Ausschluss zu gewährleisten. Hier ist der zugehörige Pseudocode:

```
wait(S);  
    // critical section  
signal(S);
```

Beschreiben Sie, was passieren würde, wenn der Programmierer einen der folgenden Fehler begeht:

- a) die Reihenfolge von `signal()` und `wait()` wird vertauscht;
- b) der Aufruf `signal()` wird durch den (erneuten) Aufruf `wait()` ersetzt;
- c) der Aufruf von `wait()` und/oder von `signal()` wird ausgelassen.