Übung 5

Abgabe: 14.06.2018 8:30

Aufgabe 5.1: Game of (critical) Zones (1 + 1 + 1 + 2 = 5 Punkte)

In dieser Aufgabe wollen wir Drachen bekämpfen und gleichzeitig etwas zum Thema Synchronisation lernen. Dazu spielen wir das Spiel *The Deadlock Empire* auf `https://deadlockempire.github.io/`.

Konkret für diese Aufgabe: Geben Sie Ihre Lösung in Form eines Schedules für die folgenden Unterlevel des Spiels an:

a) Boolean Flags Are Enough For Everyone

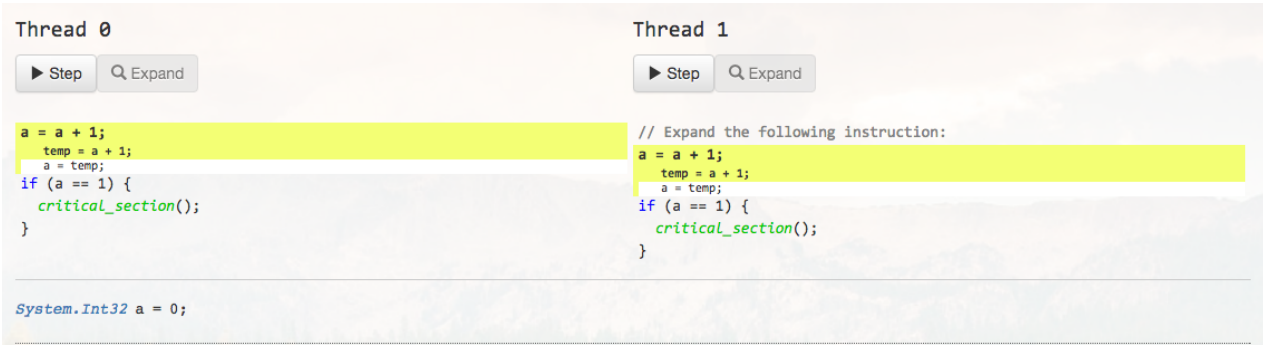
b) Simple Counter

c) Deadlock

d) Semaphores

Geben Sie Ihre Schedules in der Form an, wie es das folgende Beispiel zum Level **Tutorial 2: Non-Atomic Instructions** zeigt. Ein Latex-Template für die Tabellen finden Sie im L2P unter Lernmaterialien/Vorlagen.

Level:



Lösung:

Thread 0	Thread 1	Variable (a = 0)
temp = a + 1		
	temp = a + 1	
	a = temp	a = 1
a = temp		a = 1
if (a == 1)		
critical_section()		
	if (a == 1)	
	critical_section()	

Hinweis: Weitere Tipps sowie Unterschiede zu den in der Vorlesung vorgestellten Mechanismen.

Das Spiel benutzt Mechanismen der Programmiersprache C#. Diese lassen sich aber sehr einfach in die aus der Vorlesung bekannten Mechanismen übersetzen.

Das Konzept des `Monitors` (siehe Lock Level) existiert auf diese Art und Weise nicht in C. Der Monitor erlaubt es, ein beliebiges Objekt zu sperren oder zu entsperren; dann darf nur ein Thread gleichzeitig seine Methoden nutzen. Dies kann man in C nachbilden, indem man einen Mutex pro Objekt verwendet und in jeder Methode mit diesem Mutex den Zugriff prüft.

Aufgabe 5.2: Wechselseitiger Ausschluss (1 + 1 + 1 = 3 Punkte)

Der wechselseitige Ausschluss ist ein Grundproblem bei der Synchronisation nebenläufiger Prozesse. Es muss sichergestellt werden, dass ein Prozess in seinem kritischen Bereich nicht von anderen Prozessen unterbrochen werden kann, weil sonst das Ergebnis der Operation nicht mehr deterministisch ist.

Das Problem des wechselseitigen Auschlusses ist genau dann korrekt gelöst, wenn die drei Bedingungen

- Mutual Exclusion
- Progress
- Bounded Waiting

erfüllt sind. Gegeben seien die folgenden drei Ansätze zur Synchronisation von zwei Prozessen.

a)
```
Initialisierung:
turn = 0;
```

```
P0:
while TRUE {
    while (turn != 0){
        noop;
    }
    criticalSection(P0);
    turn = 1;
    remainderSection(P0);
}
```

```
P1:
while TRUE {
    while (turn != 1){
        noop;
    }
    criticalSection(P1);
    turn = 0;
    remainderSection(P1);
}
```

b)
```
Initialisierung:
flag[0] = FALSE;
flag[1] = FALSE;
```

```
P0:
while TRUE {
    while (flag[1]) {
        noop;
    }
    flag[0] = TRUE;
    criticalSection(P0);
    flag[0] = FALSE;
    remainderSection(P0);
}
```

```
P1:
while TRUE {
    while (flag[0]) {
        noop;
    }
    flag[1] = TRUE;
    criticalSection(P1);
    flag[1] = FALSE;
    remainderSection(P1);
}
```

c)
```
Initialisierung:
flag[0] = FALSE;
flag[1] = FALSE;
```

```
P0:
while TRUE {
    flag[0] = TRUE;
    while (flag[1]) {
        noop;
    }
    criticalSection(P0);
    flag[0] = FALSE;
    remainderSection(P0);
}
```

```
P1:
while TRUE {
    flag[1] = TRUE;
    while (flag[0]) {
        noop;
    }
    criticalSection(P1);
    flag[1] = FALSE;
    remainderSection(P1);
}
```

Zeigen Sie für jedes der 3 Verfahren an einem konkreten Beispiel, dass eine der Bedingungen für den wechselseitigen Ausschluss verletzt wird.

Aufgabe 5.3: Atomare Operationen (2 + 2 = 4 Punkte)

Gegeben seien zwei nicht teilbare (d.h. atomare) Operationen 'Decrement-and-Test': `dectest(x)` und 'Test-and-Increment': `testinc(x)`. Die beiden Operationen arbeiten wie folgt:

```
int dectest(x) {
    x--;
    if (x > 0)
      return 1;
    else if (x < 0)
      return -1;
    else
      return 0;
}
```

```
int testinc(x) {
    int result;
    if(x > 0)
      result = 1;
    else if (x < 0)
      result = -1;
    else
      result = 0;
    x++;
    return result;
}
```

Dabei sei `x` eine gemeinsame Variable aller Prozesse, d.h. eine Veränderung ihres Werts ist für alle Prozesse sichtbar.

a) Unternehmen Sie einen Lösungsversuch für den wechselseitigen Ausschluss mit Hilfe von `dectest(x)` und `testinc(x)`. Mutual Exclusion muss in Ihrer Lösung auf jeden Fall erfüllt sein, aber es macht nichts, wenn nicht alle drei Bedingungen erfüllt sind.

b) Geben Sie Pseudocode für die Realisierung der Funktionen `signal()` und `wait()` von Semaphoren (mit Warteschlangen) mit Hilfe dieser atomaren Operationen an.

Aufgabe 5.4: Reordering & Barriers (3 Punkte)

Instruktionen werden nicht immer in der Reihenfolge ausgeführt wie sie im Quelltext stehen.

- Der Compiler kann zu Optimierungszwecken Instruktionen vertauschen.
- Die CPU kann zur besseren Auslastung Instruktionen in einer anderen Reihenfolge ausführen.
- Die CPU kann trotz korrekter Reihenfolge in der Abarbeitung der Instruktionen die Ergebnisse in einer anderen Reihenfolge an den Hauptspeicher übermitteln.

Ein solches Vertauschen ist nur möglich für unabhängige Speicherzugriffe. **In dieser Aufgabe sind Speicherzugriffe unabhängig genau dann wenn nicht die gleichen Variablen verwendet werden**.

Im folgenden Beispiel werden zwei Prozesse Parallel auf zwei Prozessoren ausgeführt. Innerhalb jedes Prozesses können die Speicherzugriffe vertauscht werden, da die Instruktionen auf verschiedenen Variablen arbeiten, die Speicherzugriffe also unabhängig sind.

```
{ A == 1; B == 2 }

CPU 1                                CPU 2
==================================   ==================================
01 A = 3;                            01 x = B;
02 B = 4;                            02 y = A;
```

Für `x` und `y` gibt nach Ausführung beider Prozesse vier mögliche Ergebnisse.

```
x == 2, y == 1
x == 2, y == 3
x == 4, y == 1
x == 4, y == 3
```

Insbesondere `x == 4, y == 1` ist nur dann möglich, falls die Reihenfolge von Speicherzugriffen vertauscht wurde.

```
CPU 1                                CPU 2
==================================   ==================================
02 B = 4
                                     01 x = B;
                                     02 y = A;
01 A = 3;
```

Um ein Vertauschen zu unterbinden, kann eine Memory Barrier in den Quelltext eingefügt werden. Eine Memory Barrier ist eine Instruktion die das vertauschen von Speicherzugriffen über die Barrier hinaus verbietet.

In der folgenden Variante ist das Ergebniss `x == 4, y == 1` nicht möglich, da in beiden Prozessen die Instruktionen nicht über die Barrier hinweg vertauscht werden dürfen.

```
{ A == 1; B == 2 }

CPU 1                                CPU 2
==================================   ==================================
01 A = 3;                            01 x = B;
02 barrier;                          02 barrier;
03 B = 4;                            03 y = A;
```

Der Peterson Algorithmus wie er in der Vorlesung vorgestellt wurde, ist nicht gegen vertauschen von Speicherzugriffen abgesichert.

```
flag = {false, false};

CPU 1                                  CPU 2
=================================      ==================================
01 repeat                              01 repeat
02   flag[1] = true;                   02   flag[0] = true;
03   turn = 0;                         03   turn = 1;
04   while (flag[0] and turn == 0)     04   while (flag[1] and turn == 1)
05     do no-op;                       05     do no-op;
06   critical section 1;               06   critical section 0;
07   flag[1] = false;                  07   flag[0] = false;
08   remainder section 1;              08   remainder section 0;
09 until false;                        09 until false;
```

Nennen sie zwei benachbarte Zeilen deren Speicherzugriffe unabhängig sind und die bei Ausführen in falscher Reihenfolge einer der drei Eigenschaften (Mutual Exclusion, Progress Requirement, Bounded Waiting) verletzt. Nennen sie die Eigenschaft die dadurch verletzt ist. Geben sie einen Schedule an, der zeigt dass die Eigenschaft verletzt ist.

Aufgabe 5.5: Reader-/Writer-Problem (3 + 2 = 5 Punkte)

Eine Variable, die von verschiedenen Prozessen modifiziert werden kann (z.B. ein Datenbankeintrag), darf nicht von zwei Prozessen gleichzeitig verändert werden. Dies wird erreicht, indem der Zugriff auf die Variable gekapselt und zu einem kritischen Bereich gemacht wird. Allerdings ist es unproblematisch, wenn sich mehrere Prozesse in diesem Bereich aufhalten, welche die Variable nur lesen, also ihren Wert nicht verändern. In der Vorlesung wurden drei Varianten zur Koordination von lesenden und schreibenden Prozessen angesprochen.

Entwerfen Sie in C-ähnlichem Pseudocode ein Programm, das mehrere Lese- und Schreibprozesse koordiniert. Folgende Regeln sollen eingehalten werden:

- In dem kritischen Bereich dürfen sich entweder nur ein Schreibprozess oder beliebig viele Leseprozesse aufhalten.
- Solange sich mindestens ein Leseprozess in dem kritischen Bereich befindet, dürfen beliebig viele weitere Leseprozesse den Bereich betreten.
- Ein Schreibprozess darf erst dann in den kritischen Bereich, wenn sich kein Leseprozess darin befindet.
- Ein auf Einlass wartender Schreibprozess darf die ihm folgenden Leseprozesse nicht am Eintreten hindern, solange sich noch mindestens ein lesender Prozess im kritischen Bereich befindet.

a) Konzipieren Sie zwei Funktionen `reader()` und `writer()`, die in beliebig vielen Ausführungen nebenläufig auftreten können und sich nach den genannten Regeln gegenseitig koordinieren. Schreiben Sie die Funktionen in einem C-ähnlichen Pseudocode. Benutzen sie binäre Semaphoren, die Sie mit den Funktionen `init()`, `wait()` und `signal()` bearbeiten.

b) Nun sollen die Regeln leicht geändert werden. Modifizieren Sie die Funktionen aus Teil a) derart, dass ein wartender Schreibprozess alle ihm folgenden Leseprozesse am Betreten des kritischen Bereichs hindert.