

**Aufgabe 1: Multi-Core Theorie**

- a) Gegeben sei folgender Algorithmus:

**Require:**  $c$  Cores, CoreID  $P \in \{0, \dots, c-1\}$ , Array  $A$  und Array  $B$  mit gleichem Datentyp, Anzahl  $n$  der Einträge in  $A$  bzw.  $B$ , Int  $mystery$

**Ensure:** Länge von  $A$  = Länge von  $B$ ,  $mystery = 0$

```

1: for ( $i = n/c * P; i < n/c * (P + 1); i++$ ) do
2:   if ( $A[i] == B[n - i - 1]$ ) then
3:     atomicIncrement(mystery);
4:   end if
5: end for
6: synch();
7:
8: if ( $P == 0$ ) then
9:   if ( $mystery == n$ ) then
10:    print("1");
11:   else if ( $mystery > (3/4) * n$ ) then
12:    print("2");
13:   else
14:    print("3");
15:   end if
16: end if

```

- (i) Beschreiben Sie in eigenen Worten was dieser Algorithmus macht.

Man vergleicht  
zwei Arrays,  
spiegelverkehrt

- (ii) Funktioniert der Algorithmus für beliebiges  $c \in \mathbb{N}$ ? Begründen Sie Ihre Antwort.

Funktioniert nur wenn  $C \leq N$  und  $m \bmod c = 0$

- b) (i) Beschreiben Sie in wenigen Sätzen eine Situation aus dem Alltag, die als Beispiel für ein *Data Race* dient. Wie könnte das *Data Race* in der beschriebenen Situation verhindert werden?  
*Hinweis:* Achten Sie darauf, dass alle in der Vorlesung gelernten Aspekte eines Data Races in Ihrem Beispiel vorkommen.

- (ii) Wie werden *Data Races* allgemein in der Softwareentwicklung verhindert, und was wird dazu benötigt?

SC bedeutet: Programm wird in der Reihenfolge ausgeführt, wie es geschrieben wurde.

**Aufgabe 2: Multicores – SC-Verletzung**

Finden Sie für das Beispiel aus der VO eine Exekutionsreihenfolge, die zu einer SC-Verletzung führt. Dabei sollen die beiden Threads auf je einem Core laufen, und jeder Core soll einen eigenen Write-Through-Cache besitzen.

sum = 0; a[0] = 3; a[1] = 7

Thread 0	Thread 1
sum := sum + a[0];	sum := sum + a[1];
...	
/* after Thread 1 has finished */	
... := sum;	

```
0: read() | mem: 0, cache: 0
1: read() | mem: 0, cache: 0
0: add(3) | mem: 3, cache: 3
1: add(7) | mem: 7, cache: 7
0: read() | mem: 7, cache: 3
```

### Aufgabe 3: Synchronisation – Blocking und Nonblocking

Gegeben sei folgendes Programmstück um zwei Variablen A und B mit neuen Werten zu versorgen. Der Zugriff ist mittels Lock/Unlock vor gleichzeitigem Zugriff geschützt.

```
1: Lock(S);  
2: A := A+1;  
4: B := B-1;  
5: Unlock(S);
```

Schreiben Sie alternativen Code mit derselben Funktionalität, der aber nonblocking sein soll, unter Verwendung von (1) RMW Operationen und (2) LL/SC. Neben den schon oben gegebenen Statements können auch Ifs, Schleifen, Labels und Gotos verwendet werden. Die Funktion **Address(V)** soll die Adresse liefern, unter der Variable V gespeichert ist.

Welche Probleme können bei der RMW- und bei der LL/SC-Implmentierung auftreten?

1)  
A\_local := A  
If not RMW(A, A\_local, A\_local + 1) then goto 1  
B\_local := B  
If not RMW(B, B\_local, B\_local - 1) then goto 3

ABA Problem  
Writes auf Variable werden nicht erkannt

Nicht richtig

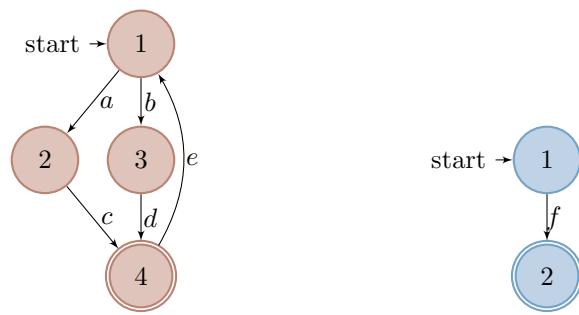
2)  
A\_local := LL(Address(A))  
if not SC(Address(A), A\_local + 1) then goto 1  
B\_local := LL(Address(B))  
if not SC(Address(B), B\_local - 1) then goto 3

Start: if not RMW(S,1,0) then goto start  
A := A+1  
B := B-1  
S := 1

Start: S\_local := LL(S\_Address)  
If S == 0 then goto start  
If not SC(S\_Address, 0) then goto start  
A := A+1  
B := B-1  
S := 1

#### **Aufgabe 4: Kronecker Algebra**

Gegeben seien die beiden Graphen (Automaten, Kontrollflussgraphen) A und B:



Berechnen Sie den Interleavingsgraphen von A und B mittels Kronecker Summe.

### **Aufgabe 5: Programm (Um-)Ordnung**

Gegeben seien zwei Threads  $Th_1$  und  $Th_2$ , die jeweils eine Sequenz an Instruktionen ausführen. Die Variablen  $T, U, V, W$  sind dabei die gemeinsamen Variablen dieser Threads, wobei die initiale Belegung vor der Ausführung von  $Th_1$  und  $Th_2$   $(T, U, V, W) = (0, 0, 0, 0)$  ist. Die Reihenfolge der Instruktionen innerhalb der Threads ist jeweils durch die Zeilennummern gegeben.

$Th_1$ :

```
1 1   U := 1;
2 2   V := T + W;
3 3   U := V - T;
```

$Th_2$ :

```
1 4   T := 2;
2 5   W := U;
3 6   T := V;
```

Geben Sie alle Variablenbelegungen von  $T, U, V, W$  an, die nach der Ausführung beider Threads möglich sind, indem Sie den aus der Vorlesung bekannten Interleavingsgraphen erstellen. Erklären Sie dabei Ihre Vorgehensweise.

## Aufgabe 6: Locks

Gegeben seien die zwei Threads  $Th_1$  und  $Th_2$  mit ihren jeweilig auszuführenden Instruktionen. Die beiden Threads werden mittels einer Semaphore  $s$  synchronisiert. Die Initialisierung der globalen Variable  $x$  sei gegeben durch  $x := 14$ .

*Th*<sub>1</sub>:

```

1 1  Lock(s);
2 2  x_th1 := x;
3 3  x_th1 := x_th1 + 4;
4 4  x := x_th1;
5 5  Unlock(s);

```

*Th*<sub>2</sub>:

```

1 a  Lock(s);
2 b  x_th2 := x;
3 c  x_th2 := x_th2 - 4;
4 d  x := x_th2;
5 e  Unlock(s);

```

Achten Sie bei der Bearbeitung folgender Unteraufgaben jeweils darauf, eine Begründung Ihrer Antwort anzugeben.

- a) Geben Sie alle möglichen *Interleavings* von  $Th_1$  und  $Th_2$  an.
  - b) Hängt der Endwert von  $x$  von der Reihenfolge ab, in der  $Th_1$  und  $Th_2$  ausgeführt werden?
  - c) Welche Werte kann  $x$  in Folge der Interleavings von  $Th_1$  und  $Th_2$  annehmen?
  - d) Was passiert, wenn Zeile 5 in  $Th_1$  fehlt? Welche Werte kann  $x$  in Folge der *Interleavings* annehmen?
  - e) Was passiert, wenn Zeile a in  $Th_2$  fehlt? Welche Werte kann  $x$  in Folge der *Interleavings* annehmen?

### Aufgabe 7: Release-Acquire Model

Gegeben seien die zwei Threads  $Th_1$  und  $Th_2$  mit ihren jeweilig auszuführenden Instruktionen. Die initiale Variablenbelegung vor Ausführung der Threads sei  $(B, X, Y, Z, V1, V2, E) = (24, 4, 2, 3, 0, 0, 0)$ .

$Th_1$ :

```
1 1      V1 := atomic_load(B, Acquire);
2 2      X := X + 1;
3 3      V1 := V1 * X;
4 4      V1 := V1 - 1;
5 5      atomic_store(B, V1, Release);
6 6      atomic_store(E, 1, Release);
7 ift1 if atomic_load(E, Acquire) = 1
8 then goto ift1;
9 7      X := X * X;
10 8      Y := Z + 1;
11 9      V1 := V1 + 1;
12 10     Z := atomic_load(B, Acquire);
```

$Th_2$ :

```
1 ift2  if atomic_load(E, Acquire) != 1
2 then goto ift2
3 a      V2 := atomic_load(B, Acquire);
4 b      V2 := V2 / 7;
5 c      X := X * Z
6 d      Z := Z + 1;
7 e      Y := Y + V2;
8 f      atomic_store(B, V2, Release);
9 g      atomic_store(E, 0, Release);
```

- Berechnen Sie die endgültigen Werte der Variablen nach Ausführung der Threads. Begründen Sie Ihre Antwort.
- Geben Sie alle Release- und Acquire-Operationen im obigen Beispiel an, die ohne Auswirkung auf die Endwerte der Variablen *relaxed* werden können. Begründen Sie Ihre Antwort.
- Ist es nötig, dass die Variable  $B$  *atomic* ist? Begründen Sie Ihre Antwort.

Alles mit B  
relaxed also nein