

Aufgabe 6: Synchronisation

6.1 Kritischer Abschnitt

- a) Erläutern Sie den Begriff des kritischen Abschnitts.

- b) Welche Eigenschaften sollten sinnvolle Lösungen zur Behandlung eines kritischen Abschnitts haben?

- c) Erläutern Sie den Unterschied zwischen Semaphor und Mutex.

- d) Welche Möglichkeiten gibt es für die Synchronisation **ohne** aktives Warten? Erklären Sie diese und nennen Sie jeweils eine Realisierung.

- e) Können die Verfahren aus Aufgabe 6.1 d) in jedem Fall problemlos durch einen Benutzerprozess ausgeführt werden (User-Mode)? Bitte begründen Sie die Antwort.

Gegeben sei ein einfacher Algorithmus für den gegenseitigen Ausschluss zweier Prozesse (P_i und P_j) bezüglich des Zugriffs auf eine gemeinsame Ressource. Hierbei wird vorausgesetzt, dass primitive Datentypen atomar geschrieben werden. Wenn also zwei Prozesse auf die gleiche Variable schreibend zugreifen, so setzt sich der Schreibvorgang eines Prozesses durch.

```
1  int marker[2] = {0, 0};
2  int action = -1;
```

Hier nun der Algorithmus aus der Sicht von P_i , wobei $i = 0$ und $j = 1$:

```

1  while(1) {
2      marker[i] = 1;
3      action = j;
4      while ( marker[j] && (action==j) ) {
5          //do nothing
6      }
7      //critical section
8      marker[i] = 0;
9      //remainder section
10 }

```

c) Ist das nebenstehende Programm richtig synchronisiert oder kann es zur Verklemmung und/oder zum gleichzeitigen Betreten des kritischen Bereiches durch beide Threads kommen? Begründen Sie Ihre Antwort.

6.3 Synchronisation mit Mutex und Semaphoren

- a) Die folgende Tabelle beschreibt eine nebenläufige Ausführung von zwei Prozessen. Als Ergebnis erhält man eine Folge von [tick] und [tock]. Welche Ausführungsreihenfolgen sind möglich?

Semaphore baz = 1; Semaphore foo = 3; Semaphore bar = 5;	
Prozess A	Prozess B
repeat forever: down(baz) down(foo) print([tick]) up(bar) up(baz)	repeat 10x: down(baz) down(bar) down(bar) print([tock]) up(foo) up(baz)

- ☐ [tick] [tock] [tick] [tock] [tick] [tock] [tick] [tock] [tick] [tock]
☐ [tock] [tick] [tock] [tick] [tock] [tick] [tock] [tick] [tock] [tick]
☐ [tick] [tick] [tock] [tick] [tick] [tock] [tick] [tick] [tock] [tick]
☐ [tock] [tick] [tick] [tock] [tick] [tick] [tock] [tick] [tick] [tock]

- b) Ermitteln Sie die Ausführungsabfolgen der Prozesse (X, Y, Z) in Abhängigkeit der gegebenen Mutexbelegung.

1. semaphore `sx` = 1;; semaphore `sy` = 0;; semaphore `sz` = 0;
2. semaphore `sx` = 0;; semaphore `sy` = 1;; semaphore `sz` = 0;
3. semaphore `sx` = 0;; semaphore `sy` = 1;; semaphore `sz` = 1;

Prozess X	Prozess Y	Prozess Z
down(sx)	down(sy)	down(sz)
Print(Fiasco)	Print(eCos)	Print(RIoT)
up(sy)	up(sz)	up(sx)
down(sx)	down(sy)	down(Contiki)
Print(Barrelfish)	Print(RTEMS)	Print()
up(sz)	up(sx)	up(sy)
down(sx)	down(sy)	down(sz)
Print(SunOS)	Print(QNX)	Print(TinyOS)
up(sy)	up(sz)	up(sx)
down(sx)	down(sy)	down(sz)

- c) Ergänzen Sie das Programm durch Mutexe, sodass die Ausgabe *'Erste Regel, ihr verliert kein Wort über den Fight Club.'* erscheint und die Werte der Mutexe am Anfang und Ende der Ausführung identisch sind.

Prozess A	Prozess B	Prozess C
print(kein)	print(Erste Regel,)	print(ihr)
print(den)	print(verliert)	print(über)
print(club.)	print(Wort)	print(Fight)

