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.

6.2 Synchronisationsalgorithmen

Gegeben sei ein einfacher Algorithmus für den gegenseitigen Ausschluss zweier Prozesse $(P_i \text{ und } P_j)$ 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.

Vor Start des Algorithmus werden folgende Datenstrukturen initialisiert:

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

action zeigt hierbei an, welcher der beiden Prozesse P_i bzw. P_j als nächstes an der Reihe ist, den kritischen Abschnitt zu betreten. Das Array marker indiziert mit einer 1 (= true), ob ein Prozess bereit ist, in den kritischen Abschnitt einzutreten.

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

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

a) Handelt es sich bei dem Algorithmus um eine valide Lösung zur Koordination eines kritischen Abschnitts zwischen zwei Prozessen?



b) Welche Schwachstellen könnte dieser Algorithmus in der Praxis haben?



globale Variable: int mutex = 1;

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.

```
thread_1
save_world{
  while( mutex == 0 )
  { /*wait*/ }
  mutex = 0;
  //critical section
  mutex = 1;
}
```

```
destroy_world{
  while( mutex == 0 )
  { /*wait*/ }
  mutex = 0;
  //critical section
  mutex = 1;
}
```

thread_2



6.3 Synchronisation mit Mutex und Semaphoren

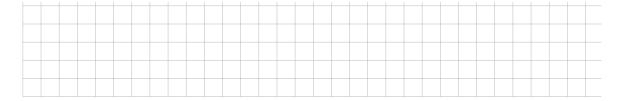
a) Die folgende Tabelle beschreibt eine nebenläufige Ausführung von zwei Prozessen. Als Ergebnis erhält man eine Folge von [tick] und [tock]. Welche Ausführungsreihenfolgen sind möglich?

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

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
<pre>print(kein)</pre>	<pre>print(Erste Regel,)</pre>	<pre>print(ihr)</pre>
<pre>print(den)</pre>	<pre>print(verliert)</pre>	print(über)
<pre>print(club.)</pre>	<pre>print(Wort)</pre>	<pre>print(Fight)</pre>

6.4 Anwendungsfall: Erzeuger-Verbraucher-Problem

Eine Implementierung von Semaphoren für C findet sich in der POSIX-Standard Bibliothek. Semaphoren werden hier als Variablen vom Typ sem_t deklariert und müssen mit Hilfe der Funktion sem_init(3) initialisiert werden:

int sem_init(sem_t *sem, int pshared, unsigned int value)

- sem ist hierbei ein Zeiger auf den zu initialisierenden Semaphor.
- pshared ist ein Flag, das angibt, ob der Semaphor zwischen Prozessen geteilt wird.
- value ist der Startwert für den Semaphor.
- die Rückgabe ist 0 im Erfolgsfall und -1 im Fehlerfall.

Zum Erhöhen und Heruntersetzen des Semaphors werden die Funktionen int sem_post(sem_t *sem) und int sem_wait(sem_t *sem) verwendet. Die Rückgabe ist bei beiden Funktionen identisch zu sem_init.

Der folgende Programmcode soll den Zugriff von mehreren Threads auf einen gemeinsam genutzten Speicherbereich thread-sicher realisieren. Hierfür muss gewährleistet sein, dass jeweils nur ein Thread exklusiv in den geteilten Speicherbereich schreiben darf. Um eine möglichst hohe Leistung zu erzielen, soll es möglich sein, gleichzeitig von mehreren Threads lesende Zugriffe auf den geteilten Speicher durchzuführen, solange parallel kein schreibender Zugriff erfolgt.

a) Vervollständigen Sie den nachfolgenden Programmcode mit POSIX-Semaphoren, um einen thread-sicheren Zugriff auf den geteilten Speicherbereich unter den oben beschriebenen Bedingungen zu realisieren.

```
int reader_counter = 0;
   sem_t sem_read, sem_write;
2
3
   void initialize()
4
   {
5
      sem_init(&sem_read, 0, ____);
6
      sem_init(&sem_write, 0, ____);
   }
10
   void write_data (blob_t *data)
11
12
13
14
      write_to_shared_memory(data);
15
16
17
   }
18
19
   void read_data(blob_t *buffer)
20
   {
21
22
       .____;
      reader_counter++;
23
      if(reader_counter == 1) _____;
24
25
26
      read_from_shared_memory(buffer);
27
28
29
30
      reader_counter--;
      if(reader_counter == 0) _____;
31
33
      ____;
   }
34
```

b) Können Leser oder Schreiber in diesem Algorithmus verhungern? Begründen Sie kurz Ihre Antwort.



Besprechung der Lösung am 04.12.2018 in der großen Übung