VO 182.022 24. Oktober 2007

Prüfung Betriebssysteme

KNr. MNr. Zuname, Vorname

Ges.)(100)	1.)(30)	2.)(20)	3.)(18)	4.)(32)	Zusatzblätter:

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Synchronisation (30)

a) Definition von Semaphoroperationen

Geben Sie eine Implementierung der Semaphoroperationen *init()*, *wait()* und *signal()* für *Counting Semaphores* an, indem Sie die folgenden Codeskelette mit Pseudocode ergänzen.

```
/* *** Semaphore init operation *** */

init(                                    )

{

}
```

```
/* *** Semaphore wait operation *** */

wait(                                    )

{

}
```

```
/* *** Semaphore signal operation *** */

signal(                                    )

{

}
```

b) Verwendung von Semaphoren

In einem Multiprozessor-System gibt es vier Prozessoren *P1*,..., *P4* mit eigenem, privatem Speicher, sowie weiters zwei getrennte Shared Memory Blöcke *ShM1* bzw. *ShM2*, über die die Prozesse, die auf den vier Prozessoren laufen, kommunizieren.

Datenobjekte (*data*) werden mit den Operationen *read(ShMx, data)* bzw. *write(ShMx, data)*, wobei *ShMx* für *ShM1* oder *ShM2* steht, von einem der Shared Memory Blöcke gelesen bzw. auf einen Shared Memory Block geschrieben.

Ergänzen Sie den Code von vier Tasks *T1*,...,*T4*, die jeweils in einer Endlosschleife auf einem der Prozessoren laufen, mit geeigneten Operationen zum Lesen bzw. Schreiben der Shared Memory Blöcke sowie Semaphoroperationen. Erfüllen Sie dabei die geforderten Kommunikations- und Synchronisationsanforderungen.

- *T1* generiert mit der Funktion *generate(data)* Daten und schreibt diese auf *ShM1*, von wo sie von *T2* und *T3* gelesen werden. Jeder von *T1* generierte Datensatz soll von *T2* und *T3* genau einmal gelesen werden bevor *T1* den nächsten Datensatz generiert.
- *T2* und *T3* haben identisches Verhalten. Nach jedem Auslesen von *ShM1* (siehe voriger Punkt) lesen sie den gerade aktuellen Inhalt von *ShM2* und verarbeiten die gelesenen Inhalte mit der Funktion *process(data1, data2)* weiter.
- *T4* kopiert den aktuellen Inhalt von *ShM1* auf *ShM2*.

Initialisierungen:

Code für Task *T1*:

```
loop forever
```

```
end loop
```

Code für Task *T2*:

```
loop forever
```

```
end loop
```

Code für Task *T3*:

```
loop forever
```



```
end loop
```

Code für Task *T4*:

```
loop forever
```



```
end loop
```

2 Uniprocessor Scheduling (20)

Gegeben ist nebenstehendes Taskset. Alle Tasks sind periodisch, wobei die Deadlines mit dem Ende der jeweiligen Periode gleichzusetzen sind. Der Overhead für den Taskwechsel ist vernachlässigbar.

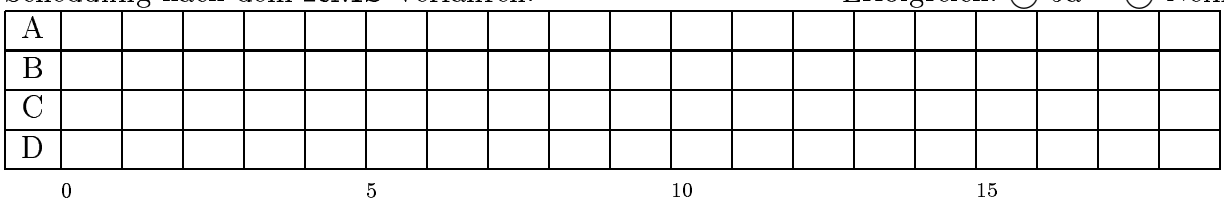
Task	Ausführungszeit	Periodendauer
A	1	7
B	1	6
C	2	4
D	1	9

Ermitteln Sie für dieses Taskset die *notwendige* und die *hinreichende* Bedingung für das *Rate Monotonic Scheduling* (RMS) Verfahren. Berechnen Sie die **Zahlenwerte** überschlagsmäßig ($\sqrt[2]{2} \approx 1,41$ $\sqrt[3]{2} \approx 1,26$ $\sqrt[4]{2} \approx 1,19$ $\sqrt[5]{2} \approx 1,15$ $\sqrt[6]{2} \approx 1,12$).

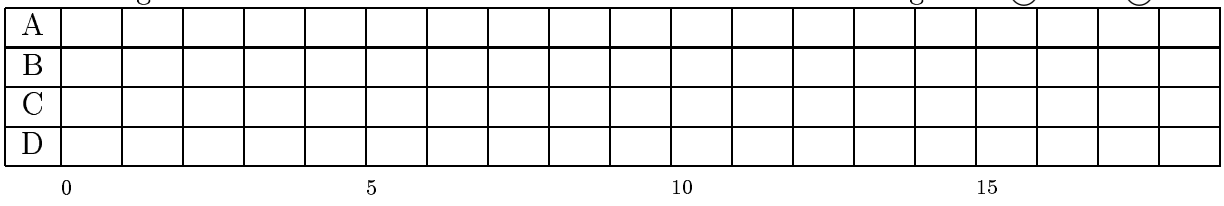
Ist die notwendige Bedingung erfüllt? ◯ Ja ◯ Nein
Ist die hinreichende Bedingung erfüllt? ◯ Ja ◯ Nein

Versuchen Sie das Taskset einmal mit dem RMS und einmal mit dem *Earliest-Deadline-First* (EDF) Verfahren zu schedulen. Verwenden Sie dazu die nachstehenden Vorlagen. Tragen Sie bei jeder Vorlage die aktiven Taskzeiten ein und bezeichnen Sie deutlich eventuelle Deadlineverletzungen. Kreuzen Sie jeweils an, ob das Scheduling erfolgreich war. Eine Vorlage dient als Ersatz, streichen Sie gegebenenfalls eine falsch ausgefüllte Vorlage deutlich durch.

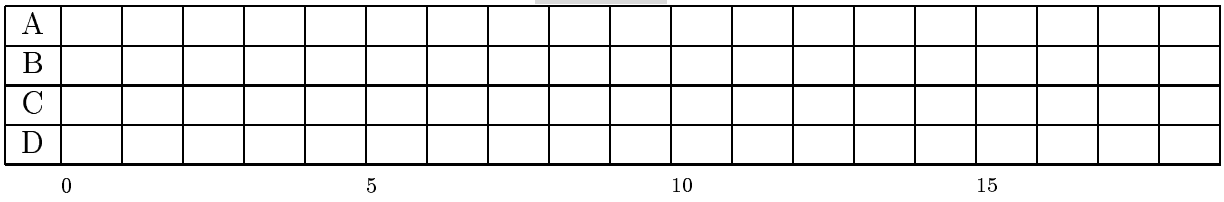
Scheduling nach dem **RMS**-Verfahren: Erfolgreich: ◯ Ja ◯ Nein



Scheduling nach dem **EDF**-Verfahren: Erfolgreich: ◯ Ja ◯ Nein



Ersatzvorlage: Scheduling nach dem -Verfahren: Erfolgreich: ◯ Ja ◯ Nein



3 Deadlock (18)

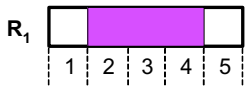
Gegeben sind zwei Prozesse, P_1 und P_2, die jeweils die Ressourcen R_1 und R_2 benötigen. Jede der zwei Ressourcen ist zweimal vorhanden. Benötigt ein Prozess eine vom anderen Prozess belegte Ressource, so wird er auf jeden Fall bis zum Freiwerden der Ressource verzögert.

Der Fortschritt von P_1 und P_2 bei der (quasi)parallelen Abarbeitung kann als Kantenzug zwischen den Punkten *begin* und *end* in der Grafik eingetragen werden. Die Achsenbeschriftung entspricht dabei der Zeilennummer des gerade auszuführenden Befehls.

Unterhalb bzw. links der Diagrammachsen sind Balken vorgesehen, in denen die Anforderungen von Ressourcen für P_1 bzw. P_2 eingetragen werden.

1. Tragen Sie die Anforderungen von Ressourcen für P_1 bzw. P_2 ein. Dabei ist anzunehmen, dass eine Ressource bereits ab Start der Anweisung `get()` als belegt gilt und erst nach Beendigung der Anweisung `free()` als wieder freigegeben gilt:

   ```
   2: get(R1)
   3: ...
   4: free(R1)
   ```

   Sind mehrere Instanzen einer Ressource belegt, so geben Sie die zuletzt belegte Instanz frei.

2. Umranden und schraffieren Sie in der Grafik jene Bereiche, durch die der Kantenzug einer (quasi)parallelen Abarbeitung aufgrund von Ressourcenkonflikten nicht möglich ist.

3. Kennzeichnen Sie auf unterschiedliche Weise die Bereiche, die von einem Kantenzug nicht passiert werden dürfen, wenn eine Abarbeitung von P_1 und P_2 deadlockfrei erfolgen soll.

4. Zeichnen Sie einen Kantenzug für eine gültige, deadlockfreie Abarbeitung von P_1 und P_2 in der Grafik ein.

5. Beschriften Sie einen Punkt im Koordinatensystem (d.h. schreiben Sie den jeweiligen Buchstaben im Kästchen links unterhalb) ...

   ... mit 'A', von welchem aus der Punkt *end* bzw. welcher vom Punkt *begin* erreichbar ist

   ... mit 'B', welcher unweigerlich zu einen Deadlock führt, sofern ein solcher Punkt vorhanden ist

   ... mit 'C', welcher einen nicht erlaubten Zustand darstellt, sofern eine solcher Punkt vorhanden ist

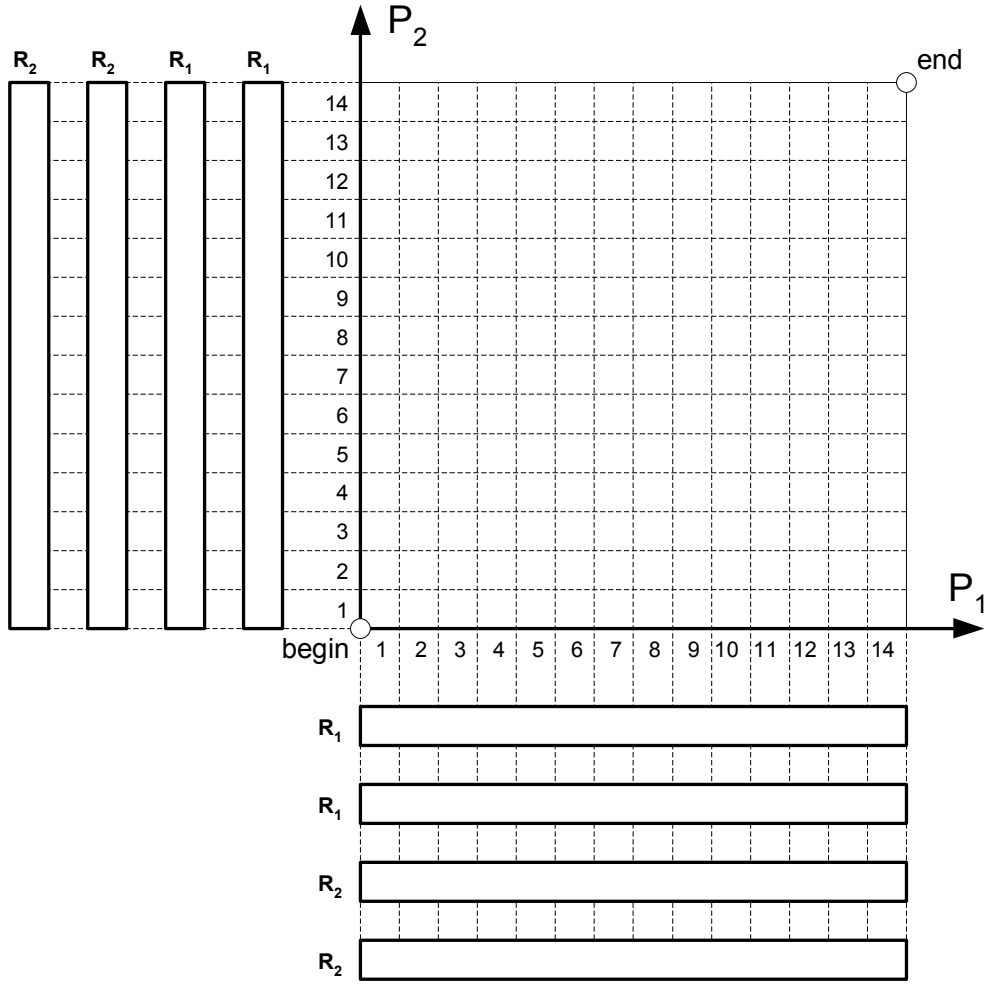
Anmerkung: Achten Sie bitte darauf, dass alle Lösungen gut erkennbar und die Lösungen zu den Teilaufgaben 2 und 3 *deutlich unterscheidbar* sind.

Program P_1:

```
 1: a=3;
 2: get(R1);
 3: b=4;
 4: get(R1);
 5: get(R2);
 6: get(R2);
 7: c=a+b;
 8: free(R2);
 9: a=b*4;
10: free(R2);
11: free(R1);
12: b=9;
13: free(R1);
14: return;
```

Program P_2:

```
 1: a=12;
 2: get(R2);
 3: get(R1);
 4: b=3*a;
 5: get(R2);
 6: c=a+b;
 7: a=b*c;
 8: free(R2);
 9: free(R1);
10: d=a+b;
11: a=b*5;
12: free(R2);
13: a=b+5;
14: return;
```



4 Memory Management (32)

a) Virtueller Speicher mit Kombination aus Segmentierung und Paging 18

In folgendem Beispiel sollen Sie ausgehend von virtuellen Adressen die physikalischen Adressen bestimmen. Es werden folgende Begriffe (englische Notation) aus dem Buch zur Vorlesung verwendet:

Base	Basisadresse Seitentabelle des Segmentes
Length	Länge des Segmentes (Anzahl der Seiten des Segmentes)
Virt.Addr.	Virtuelle Adresse
Frame#	Seitenrahmennummer (im physischen Speicher)
Page#	Seitennummer (im virtuellen Speicher)
Seg#	Segmentnummer

Es handelt sich bei dem System um ein System mit 32-Bit-Adressen, wobei die niederwertigen 16 Bit immer den Offset einer Adresse bilden und die höherwertigen 16 Bit Segment- und Seitennummer bilden:

Seg# (8 bit)	Page# (8 bit)	Offset (16 bit)

Es wird dabei direkter Zugriff (direct mapping) sowohl auf die Segmenttabelle als auch auf die Seitentabelle verwendet.

Bei Paging sind alle Seiten 65536 Bytes (hexadezimal `0x0001 0000`) lang. Alle Werte sind als Hexadezimalzahlen angegeben. Ergibt sich bei der Umwandlung eine ungültige Adresse, so schreiben Sie bitte "invalid segment nr." bzw. "invalid page nr." in das entsprechende Feld.

Verwenden Sie für die Adressumsetzung folgende Segmenttabelle:

Segmenttabelle		
Seg#	Base	Length
0x00	0x0102 7234	0x03
0x01	0x0300 6073	0x01
0x02	0xC9C0 8054	0x02
0x03	0x8A9F 23AA	0x10
0x04	0x2001 B578	0x0A

und folgende Seitentabelle:

Seitentabelle	
Address	Frame#
...	...
0x0102 7234	0x6752
0x0102 7235	0x7613
0x0102 7236	0x258A
0x0102 7237	0xB3CA
...	...
0x0300 6073	0x56EF
0x0300 6074	0x4567
...	...
0x2001 B57F	0x5014
0x2001 B580	0x5109
0x2001 B581	0xA145
0x2001 B582	0x2000
0x2001 B583	0x3234
...	...
0xC9C0 8054	0x7353
0xC9C0 8055	0xBABA
0xC9C0 8056	0x9789
0xC9C0 8057	0xAFFE
...	...
0x8A9F 23B7	0x5400
0x8A9F 23B8	0x5401
0x8A9F 23B9	0x0945
0x8A9F 23BA	0x1313
...	...

Ermitteln Sie unter Benützung obiger Tabellen die physikalischen Adressen zu den folgenden virtuellen Adressen. Markieren Sie die den Eintrag mit "invalid segment nr." bzw. "invalid page nr." im Fehlerfall.

Virtuelle Adresse	Physikalische Adresse (zu ermitteln)
0x030F 01CE	
0x0001 04B3	
0x8A9F 23B9	
0x0409 1025	
0x0409 0102	
0x04B0 1019	
0x0539 0715	
0x0407 294A	
0x0310 9728	

Bitte beachten Sie, dass bei diesem Beispiel falsche Antworten zu Punkteabzug führen.

b) Verständnisfragen (14)

Kreuzen Sie bitte die richtigen Antworten an! Achtung! Falsche Antworten werden negativ gewertet!

- Bei folgender Speicherverwaltungstechnik kann das Problem der internen Fragmentierung auftreten.
  - ◯ fixed partitioning
  - ◯ simple segmentation
  - ◯ virtual-memory paging
  - ◯ dynamic partitioning
  - ◯ simple paging
  - ◯ virtual memory with combination of paging and segmentation

- Beim *Fixed Partitioning* sind die Partionen *immer* von gleicher Größe.
  - ◯ richtig
  - ◯ falsch

- Bei folgender Speicherverwaltungstechnik tritt sowohl interne also auch externe Fragmentierung auf.
  - ◯ fixed partitioning
  - ◯ simple segmentation
  - ◯ virtual memory with combination of paging and segmentation
  - ◯ virtual-memory paging
  - ◯ dynamic partitioning
  - ◯ simple paging

- In einem fehlerfreien System können zwei oder mehr virtuelle Adressen auf eine physikalische Adresse abgebildet sein.
  - ◯ richtig
  - ◯ falsch