VO 182.022 29. April 2010

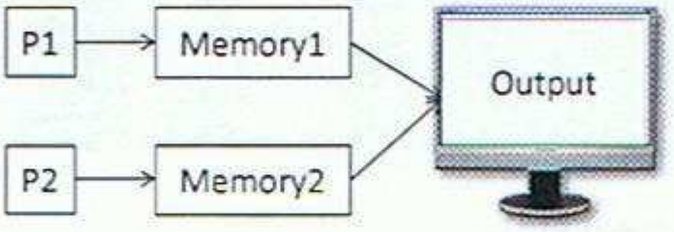
Prüfung Betriebssysteme

KNr. MNr. Zuname, Vorname

Ges.)(100)	1.)(22)	2.)(25)	3.)(30)	4.)(23)	Zusatzblätter:

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Synchronisation (25)



In einem Chatprogramm gibt es 2 Prozesse (P1 bzw. P2), die die eingehenden Nachrichten in jeweils einen Speicher (Memory1 und Memory2) schreiben. Sobald Daten in einem der beiden Speicher oder in beiden Speichern zum Lesen bereitgestellt sind, sollen diese Daten gelesen und ausgegeben werden (Output). Es darf erst wieder in einen Speicher geschrieben werden, nachdem die Daten gelesen und ausgegeben wurden.

Synchronisieren Sie den Arbeitsablauf der beiden Schreibprozesse und des Ausgabeprozesses mittels **Semaphoren**. Achten Sie auf maximale Parallelität und vermeiden Sie Starvation der Schreibprozesse. Verwenden Sie möglichst wenige Synchronisationskonstrukte. Integrieren Sie in Ihre Lösung eine Variable, die angibt in welchem der beiden Speicher Daten zum Lesen bereitgestellt sind.

Allgemeine zu verwendende Funktionen:

`initS(`***Semaphor, init***`)` Legt einen Semaphor mit dem angegebenen Namen *Semaphor* an und initialisiert ihn mit der Zahl *init*. Danach können die Funktionen **P**(***Semaphor***) und **V**(***Semaphor***) auf den Semaphor angewendet werden.

Zu verwendende Funktionen für die Schreibprozesse:

`write_mem1()` Mit dieser Funktion wird in den Speicher 1 geschrieben.

`write_mem2()` Mit dieser Funktion wird in den Speicher 2 geschrieben.

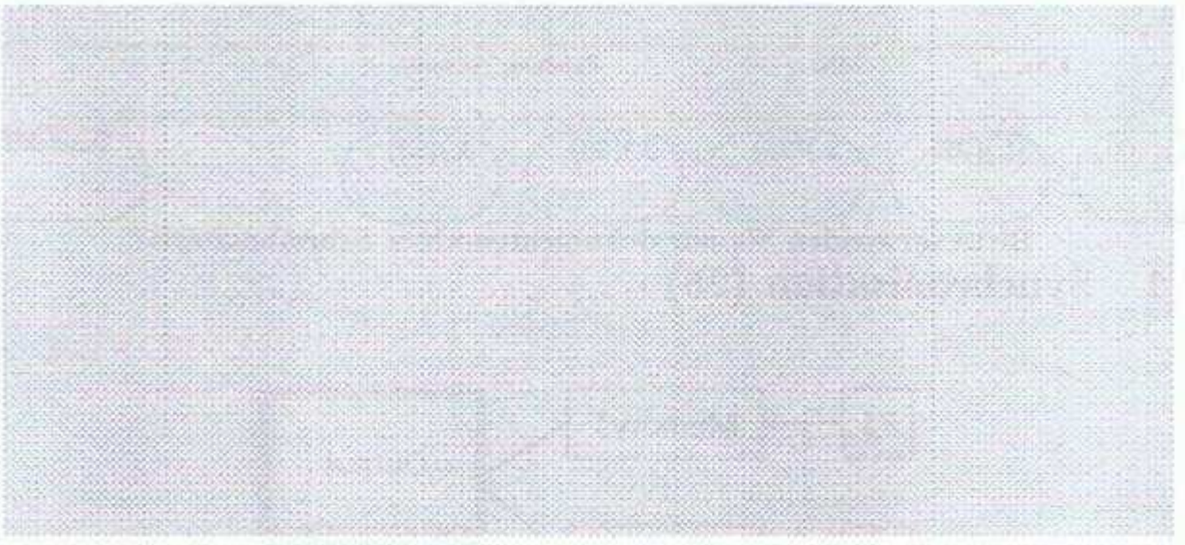
Zu verwendende Funktionen für den Ausgabeprozess:

`read_mem1()` Liest die Daten vom Speicher 1 und gibt sie aus.

`read_mem2()` Liest die Daten vom Speicher 2 und gibt sie aus.

a) Initialisierungen (5)

Initialisieren Sie die notwendigen Semaphoren und die geforderte Variable.



d) Schreibprozesse (10)

Entwerfen Sie die beiden Schreibprozesse *P1* und *P2*:

Prozess *P1*:

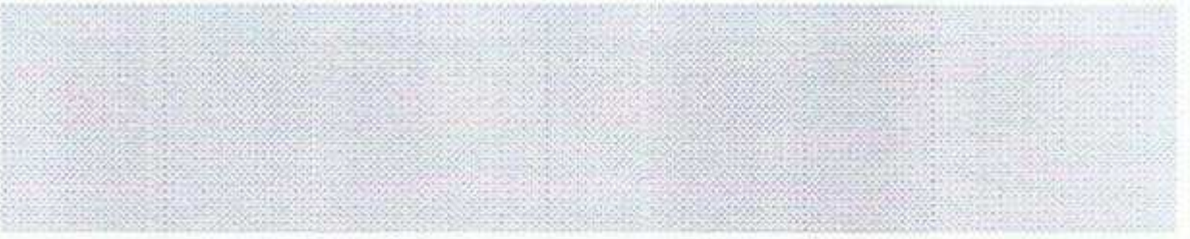
```
do forever() {
                                                    }
```

Prozess *P2*:

```
do forever() {
                                                    }
```

b) Ausgabeprozess (10)

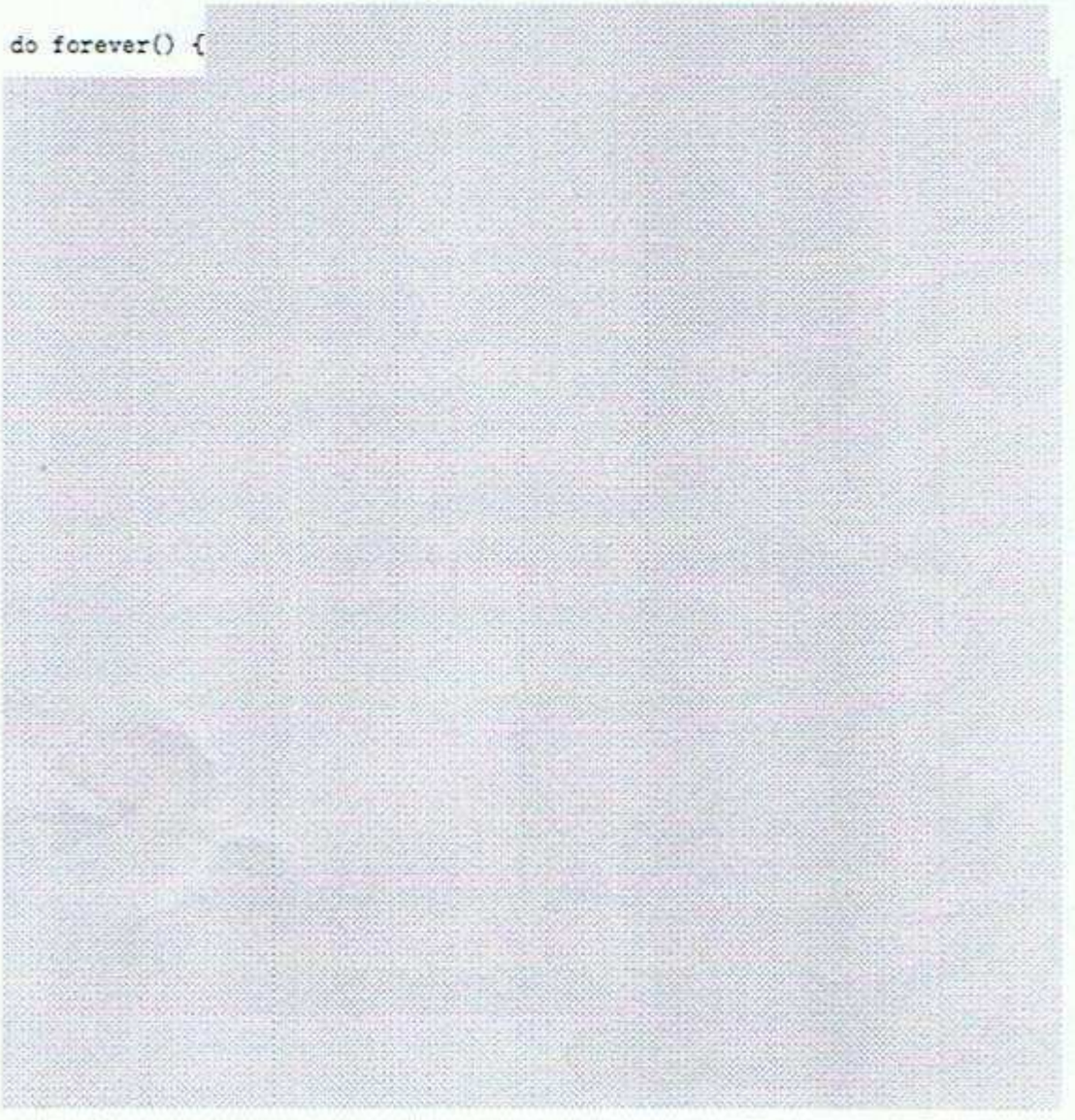
Definition von Hilfsvariablen:

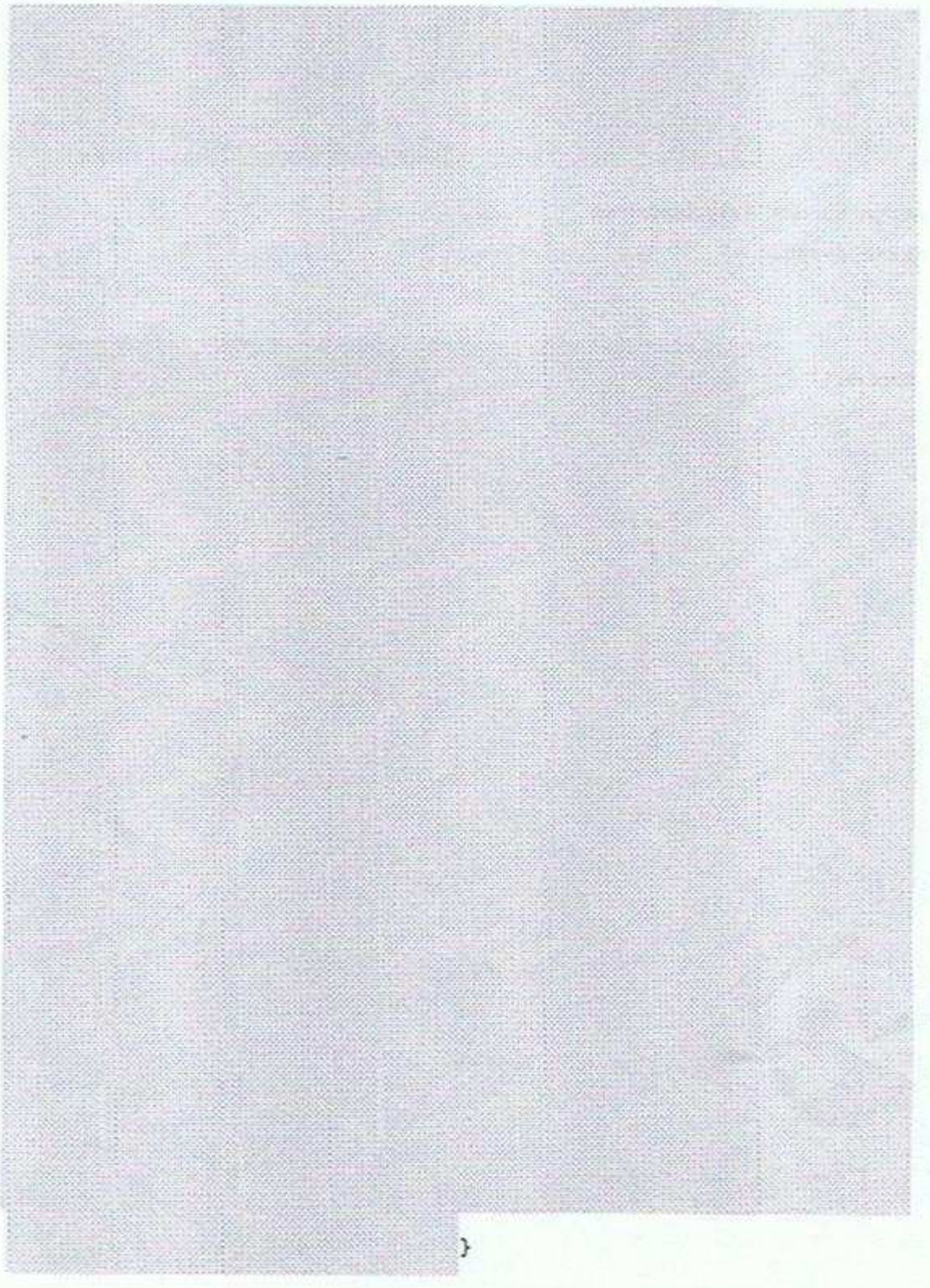


Entwerfen Sie den *Ausgabeprozess*:

```
Prozess Output:

do forever() {
```





2 Scheduling (25)

RR und SRT Scheduling (10)

Schedulen Sie das nebenstehende Taskset mit den Verfahren *Round Robin* und *Shortest Remaining Time* (SRT). Der Overhead für den Taskwechsel ist vernachlässigbar.

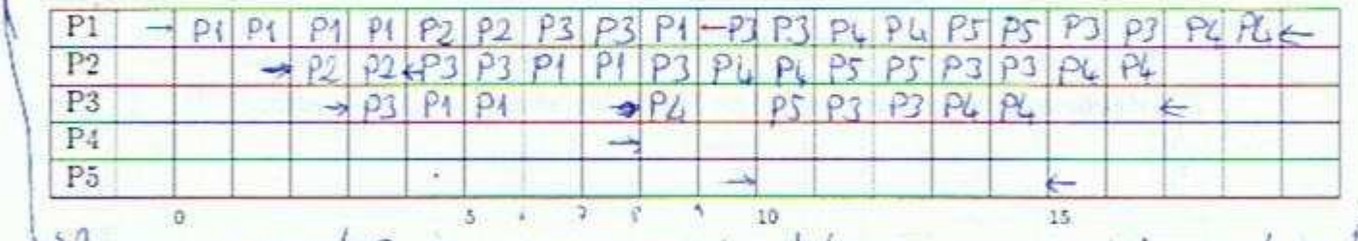
Task	Arrival Time	Service Time
P1	0	5
P2	2	2
P3	3	6
P4	8	4
P5	10	2

Beim Round Robin Verfahren soll die *Zeitscheibenlänge* 2 verwendet werden. Fügen Sie zuerst Tasks mit abgelaufener Zeitscheibe vor neu ankommenden Tasks in die Ready Queue ein. Wird ein Task beendet, bevor seine Zeitscheibe abgelaufen ist, wird sofort einem neuen Task die Ressource für 2 Zeiteinheiten zugewiesen.

Beim *SRT* Verfahren ist zu beachten, dass Tasks zum jeweiligen Ankunftszeitpunkt sofort gescheduled werden können. Stehen mehrere Tasks mit der selben verbliebenen Service Time zur Auswahl, soll der Task mit der niedrigsten Nummer gewählt werden (also beispielsweise *P4* vor *P5*).

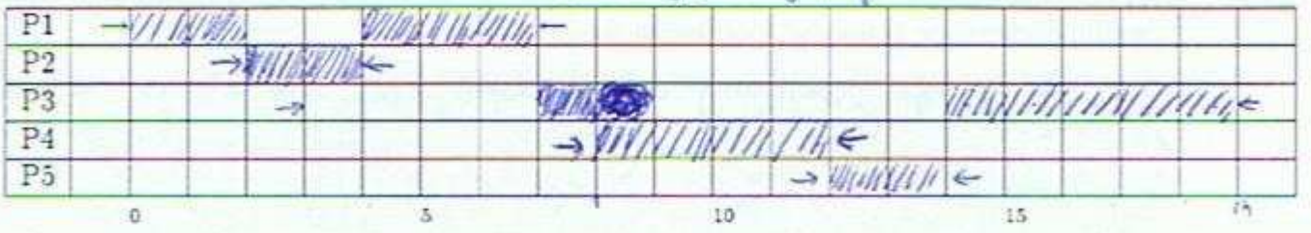
Verwenden Sie die nachstehenden Vorlagen. Markieren Sie in die Vorlage zu jedem Zeitpunkt den jeweils aktiven Task. Bezeichnen Sie weiters deutlich die *arrival time* jedes Tasks (mit einem Pfeil →) und den Zeitpunkt, wenn ein Task die gesamte benötigte *service time* konsumiert hat (←). Eine der Vorlagen dient als Ersatz, streichen Sie gegebenenfalls eine falsch ausgefüllte Vorlage deutlich durch.

Scheduling nach dem **RR**-Verfahren:

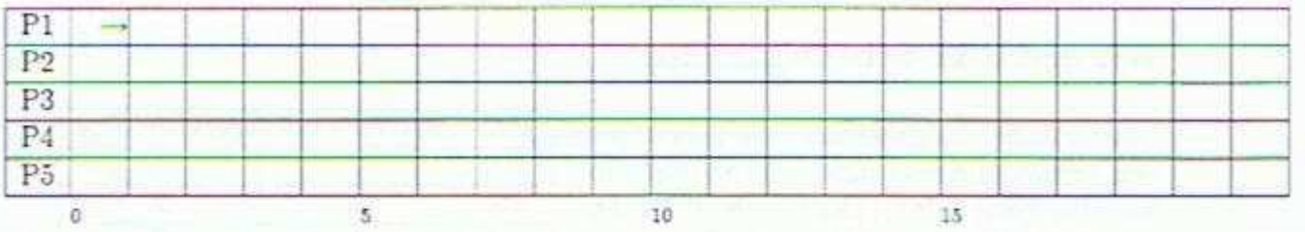


Prozesse werden in einer Warteschlang eingereiht und in FIFO Ordnung ausgewählt. Ein rechnender Prozess wird von BS nach dem Ablauf einer Zeitscheibe unterbrochen und wieder hinten in die Warteschlange eingefügt

Scheduling nach dem **SRT**-Verfahren:



Ersatztabelle: Scheduling nach dem _____ -Verfahren:



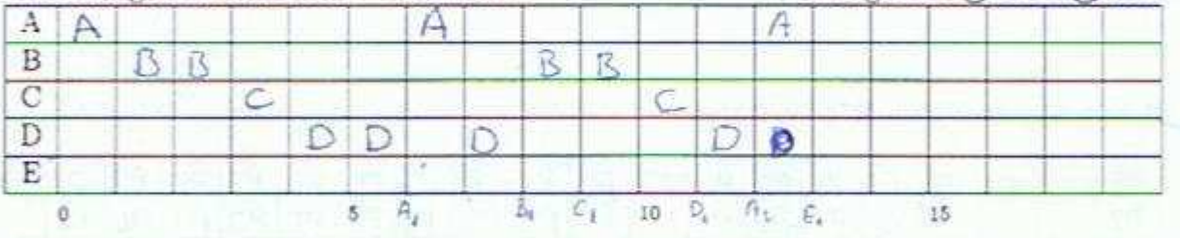
2.1 Real-Time Scheduling (9)

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.

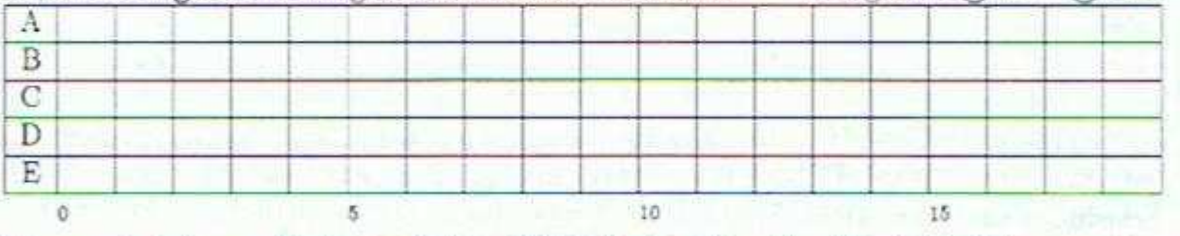
Task	Ausführungszeit	Periodendauer
A	1	6
B	2	8
C	1	9
D	3	11
E	1	13

Versuchen Sie das Taskset mit dem RMS 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 weiters 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: → Deadline von E Erfolgreich: ○ Ja ⊗ Nein



Ersatzvorlage: Scheduling nach dem **RMS**-Verfahren: Erfolgreich: ○ Ja ○ Nein



Kann es bei diesem Taskset mit dem *EDF* (Earliest Deadline First) Verfahren zu einer Deadlineverletzung kommen? Begründen Sie ihre Antwort!

Da Auslastige Bedingung erfüllt, kann keine Deadline verletzung kommen.

2.2 Verständnisfragen (6)

Werden beim Round-Robin IO-intensive oder CPU-intensive Prozesse benachteiligt? Beschreiben Sie eine Variante des Round-Robin Verfahrens, die dieses Problem zu umgehen zu versucht.

Benachteiligt I/O-intensive Prozesse.
Virtual Round Robin versucht dieses Problem zu lösen, da Auxiliary Queues mit höherer Priority

Was versteht man unter dem Begriff Starvation? Nennen Sie ein Scheduling Verfahren, bei dem es zu Starvation kommen kann.

Ein Prozess wartet auf eine Ressource, diese bekommt er aber nicht, weil sie von anderen Prozessen gebraucht wird. Der Prozess kann nicht terminieren und der Benutzer des Prozesses muss ewig auf das Resultat warten.
Shortest Remaining Time Sch.

Bei welchen der folgenden Scheduling Verfahren muss die Service Time der Tasks bekannt sein (beziehungsweise geschätzt werden)? Fehlende Antworten werden negativ, falsche Antworten werden doppelt negativ gewertet!

- ○ Ja ⊗ Nein Virtual Round Robin
- ○ Ja ⊗ Nein Earliest Deadline First
- ⊗ Ja ○ Nein Highest Response Ratio Next
- ○ Ja ⊗ Nein Rate Monotonic Scheduling
- ⊗ Ja ○ Nein Shortest Process Next
- ○ Ja ⊗ Nein Feedback Scheduling

3 Deadlock (23)

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* und 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 ein solcher Punkt vorhanden ist.

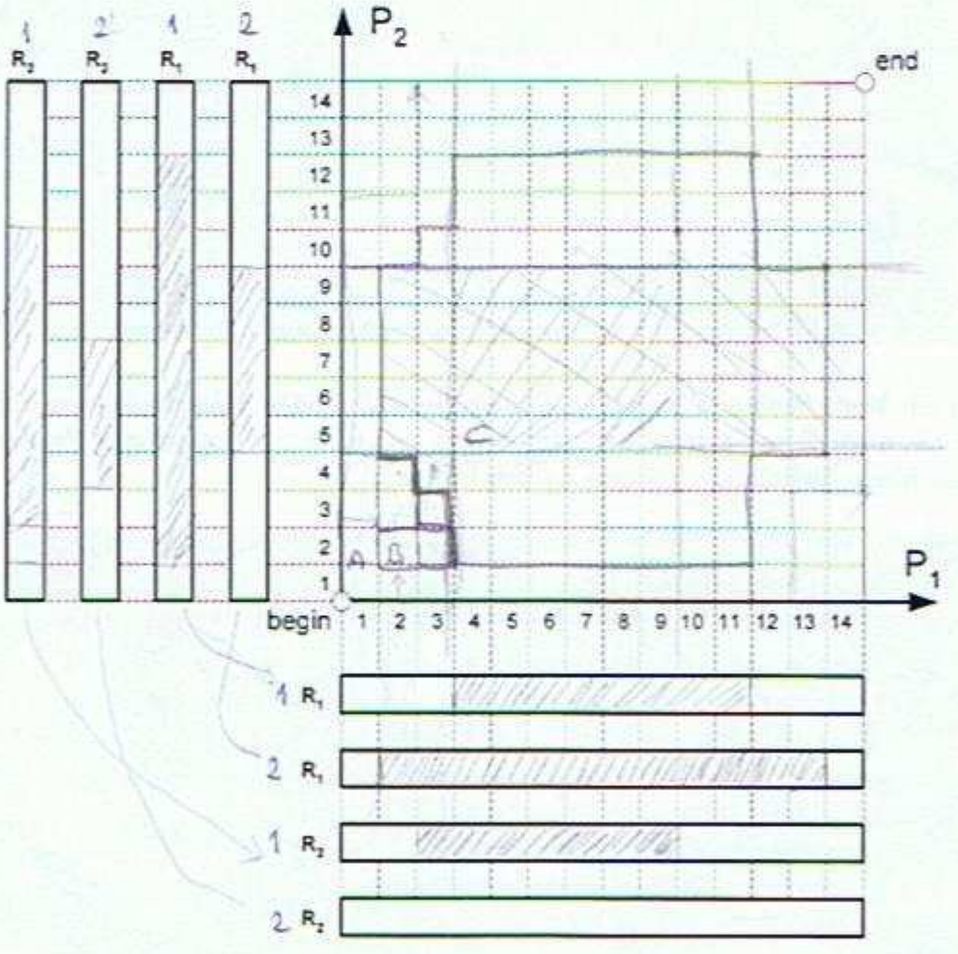
Anmerkung: Achten Sie bitte darauf, dass alle Lösungen gut erkennbar und die Lösungen zu den Teilaufgaben 2 und 3 *deutlich unterscheidbar* sind.

Program P_1:

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

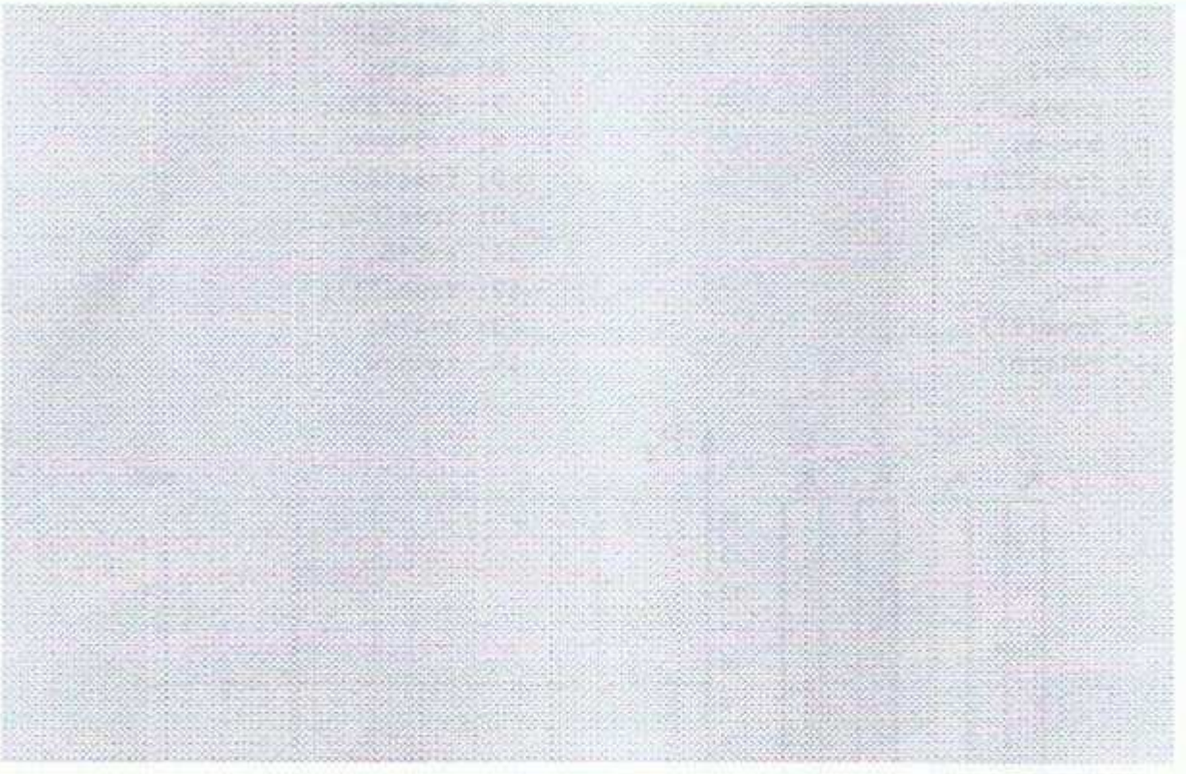
Program P_2:

```
 1: a=54;
 2: get(R1);
 3: get(R2);
 4: get(R2);
 5: get(R1);
 6: c=a+b;
 7: free(R2);
 8: a=2*b*c;
 9: free(R1);
10: free(R2);
11: a=b*5;
12: free(R1);
13: b=5+d;
14: return;
```

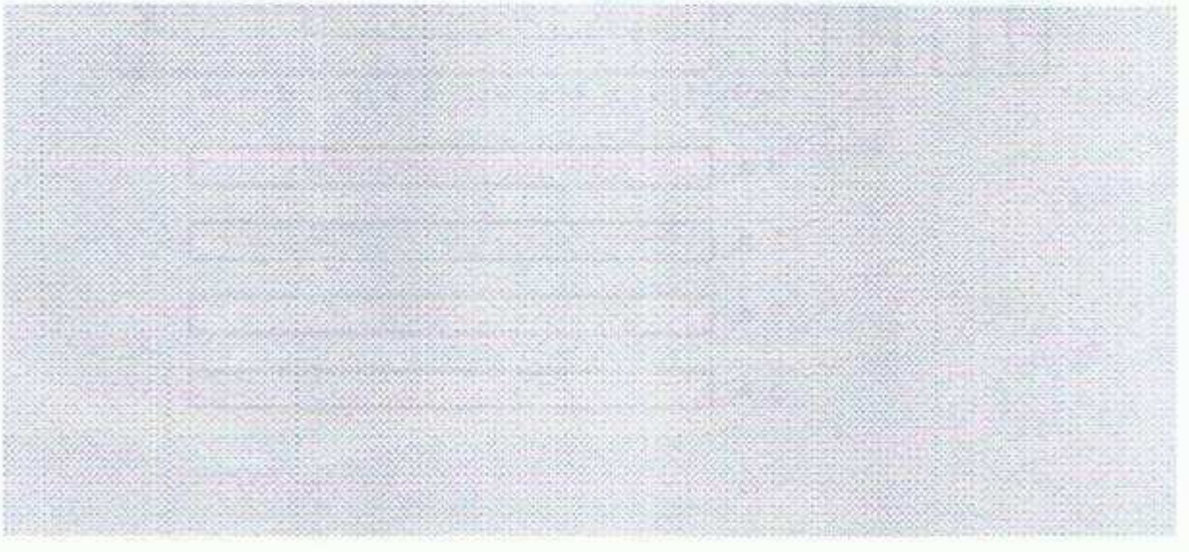


4 Betriebssyteme, Prozesse und Threads (23)

Was versteht man unter einem Microkernel-Betriebssystem? Welche Services stellt ein Microkernel zur Verfügung? Welche Eigenschaften zeichnen ein solches Betriebssystem aus?



Was ist ein *Mode Switch*? Was ist ein *Process Switch*? Erklären Sie die beiden Begriffe. Geben Sie an, wozu diese Switches benötigt werden und erläutern Sie, welcher Zusammenhang zwischen Mode Switch und Process Switch besteht.



Nennen Sie die drei Kategorien von Ereignissen, mit deren Hilfe das Betriebssystem die Kontrolle über das Computersystem übernimmt. Geben Sie für jede der Kategorien ein Beispiel an.

Beschreiben Sie *User-Level Threads* und *Kernel-Level Threads*. Wodurch unterscheiden sich diese beiden Arten der Thread-Implementierung?