Vorlesungsprüfung aus Systemprogrammierung 2001-03-21

KNr. MNr. Zuname, Vorname

Ges.)(100)	1.)(25)	2.)(20)	3.)(30)	4.)(25)	Zusatzblätter:

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Deadlock (25)

Gegeben sind zwei Prozesse, P_1 und P_2, die jeweils die Ressourcen A, B und C benötigen. Jede der drei Ressourcen ist nur einmal vorhanden und kann immer nur von einem Prozess belegt werden. Benötigt ein Prozess eine vom anderen Prozess belegte Ressource, so wird er auf jeden Fall bis zum Freiwerden der Ressource verzögert. Die Abbildung unten zeigt für jeden der beiden Prozesse, zu welchem Zeitpunkt ihrer Abarbeitung sie jeweils die einzelnen Ressourcen benötigen. Die Anforderungen von Prozess P_1 sind entlang der x-Achse, die Anforderungen von Prozess P_2 entlang der y-Achse aufgetragen.

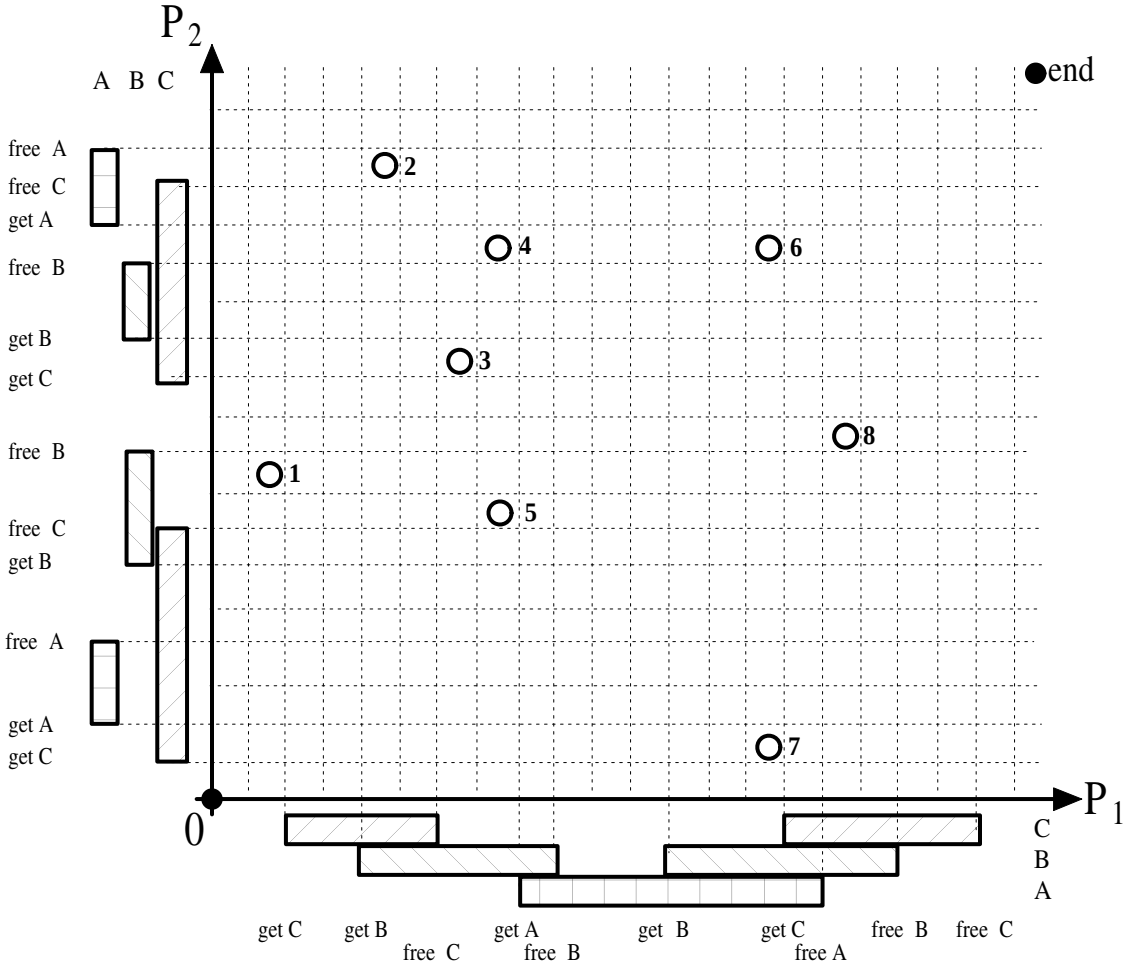
Der Fortschritt der Prozesse P_1 und P_2 bei der (quasi)parallelen Abarbeitung kann als Kantenzug zwischen dem Punkt *0* und dem Punkt *end* in der Grafik eingetragen werden (siehe Buch zur Vorlesung: W. Stallings, Operating Systems).

a) (8)

Umranden Sie in der Grafik jene Bereiche deutlich, durch die ein solcher Kantenzug aufgrund von Ressourcenkonflikten nicht gehen kann.

b) (9)

Umranden und schraffieren Sie in der Grafik die Bereiche, die ein Kantenzug nicht passieren darf, wenn eine Abarbeitung von P_1 und P_2 deadlockfrei erfolgen soll. Beschriften Sie diese Bereiche zusätzlich deutlich mit einem "D".



c) (8)

In der Grafik sind acht Punkte durch einen Kreis gekennzeichnet und nummeriert. Tragen Sie in der folgenden Tabelle für jeden Punkt ein, ob eine durch diesen Punkt gehende Abarbeitung der beiden Prozesse ab dem Punkt (a) immer deadlockfrei erfolgt, (b) zu einem Deadlock führen kann aber nicht muss, (c) immer zu einem Deadlock führt, oder (d) gar nicht möglich ist.

	1	2	3	4	5	6	7	8
(a) immer deadlockfrei	○	○	○	○	○	○	○	○
(b) Deadlock möglich	○	○	○	○	○	○	○	○
(c) sicherer Deadlock	○	○	○	○	○	○	○	○
(d) Abarbeitung unmöglich	○	○	○	○	○	○	○	○

2 Scheduling (20)

a) Earliest Deadline First Scheduling (EDF) (10)

Schedulen Sie das gegebene Task Set nach dem Earliest Deadline First Verfahren *mit Preemption* bis zum Zeitpunkt $t = 26$ ms unter Beachtung der angegebenen Deadlines.

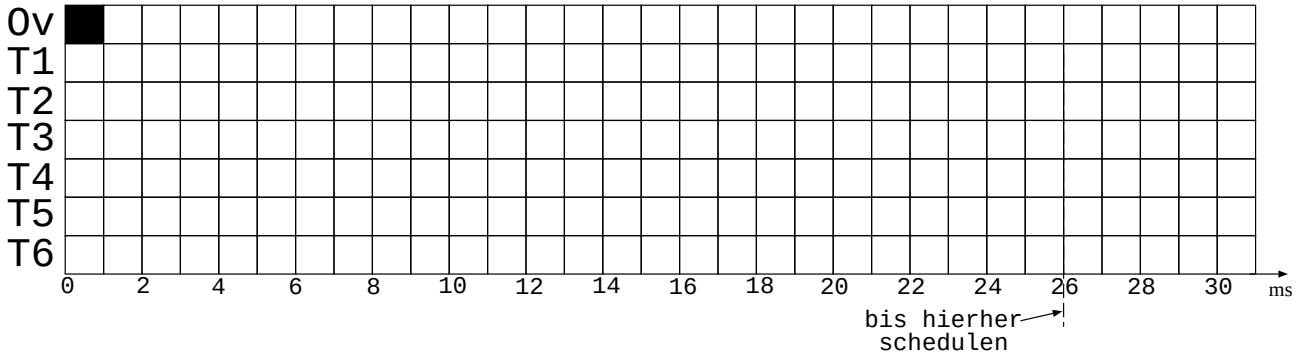
Sollte das Taskset nicht gescheduled werden können, so zeichnen Sie das gültige Schedule bitte bis zum Zeitpunkt der Verletzung der Deadline und kennzeichnen Sie diesen Zeitpunkt.

Das Taskset:

Task	Laufzeit (ms)	Periode (ms)	Deadline (ms)
T1	1	11	9
T2	2	13	10
T3	1	16	16
T4	1	12	7
T5	2	25	14
T6	3	13	12

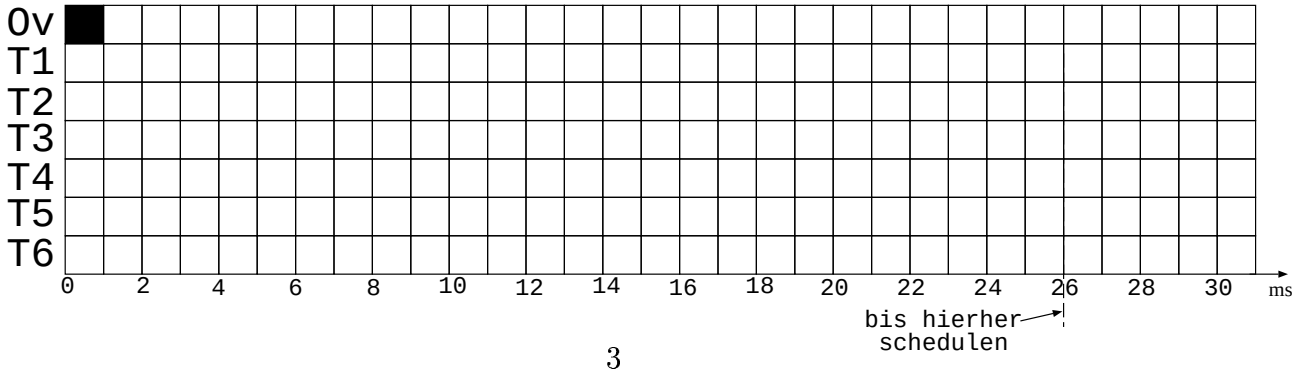
- Alle Tasks sind periodisch.
- Der Scheduling Overhead beträgt 1 ms.
- Alle Tasks sind unabhängig.
- Die erste Ankunftszeit (arrival time) ist für alle Tasks der Zeitpunkt 0.

Bitte schedulen Sie das obige Taskset.



Ersatzvorlage

Bitte streichen Sie die andere Vorlage durch, wenn Sie diese Vorlage verwenden.



b) Fragen zu Scheduling (10)

Werden bei Rate-Monotic Scheduling die Prioritäten den Tasks statisch oder dynamisch zugeordnet?

◯ dynamisch ◯ statisch

Begründung:

Was sind Vorteile von Scheduling-Mechanismen, die die Priorität statisch zuweisen?

Ist Scheduling nach dem First-In-First-Out Prinzip preemptive?

◯ ja ◯ nein

Begründung:

Fragen zu Scheduling nach dem Round-Roubin-Verfahren:

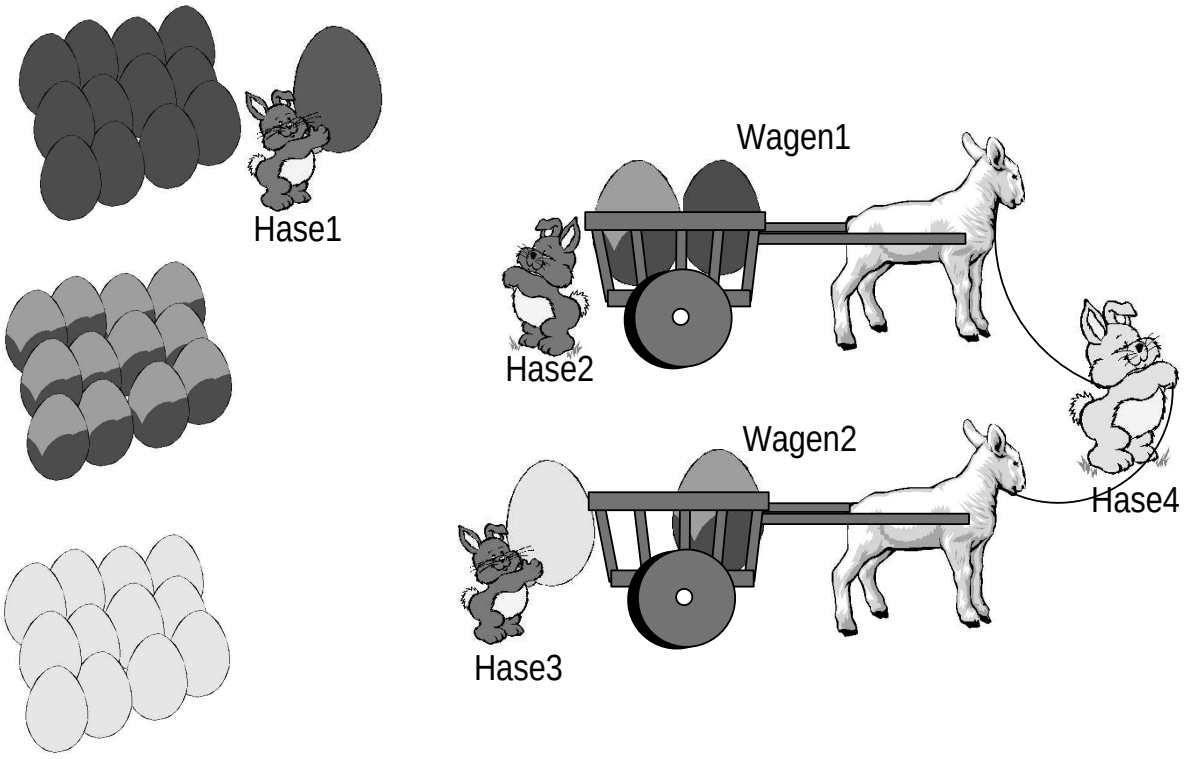
Was sind die Vorteile von großen Zeitquanten?

Was sind die Nachteile von großen Zeitquanten?

Was sind die Vorteile von kleinen Zeitquanten?

Was sind die Nachteile von kleinen Zeitquanten?

3 Synchronisation (30)



Immer wenn Ostern vor der Tür steht, herrscht bei den Osterhasen Hochkonjunktur. Eine Abteilung von vier Osterhasen ist mit dem Beladen und Versenden von bemalten Ostereiern beauftragt. Die Osterhasen sollen dabei nach folgendem Schema arbeiten:

- Jeder der drei Osterhasen *Hase1*, *Hase2* und *Hase3* hat einen eigenen unendlichen Eiervorrat, von dem er immer genau ein Ei entnehmen kann.
- Die Eier werden auf zwei Wägen geladen, von denen jeder genau zwei Eier fassen kann.
- *Hase1* belädt immer Wagen 1, *Hase3* belädt immer Wagen 2, *Hase2* belädt **abwechselnd** Wagen 1 und Wagen 2. *Hase4* beteiligt sich nicht am Beladen.
- Die beiden Wägen können unabhängig voneinander beladen werden, aber es kann immer nur **ein Ei nach dem anderen** auf einen Wagen geladen werden.
- Sind beide Wägen voll, so werden die Zuglämmer von *Hase4* zum Kunden geführt, die anderen Hasen müssen dann warten, bis *Hase4* mit den leeren Wägen wieder zurückkehrt.

Synchronisieren Sie den Arbeitsablauf der vier Hasen mittels **Semaphore**. Sie können davon ausgehen, dass am Anfang beide Wägen leer bereitstehen und die Beladehasen unbeladen bei ihren Stapeln stehen. *Hase2* belädt zuerst Wagen 1. Verwenden Sie möglichst wenige Synchronisationskonstrukte. Die Verwendung von globalen Variablen ist verboten.

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.

`gehezu( `***Ziel*** `)` Bewegt den Hasen zum *Ziel*. Als Ziel kann `Stapel1`, `Stapel2`, `Stapel3`, `Wagen1`, `Wagen2` und für *Hase4* `Beladestation` und `Kunde` angegeben werden.

`nimmEi()` Lässt den Hasen ein Ei aufnehmen. Funktioniert nur, wenn der Hase neben einem Stapel steht.

`beladeWagen( `***Wagen*** `)` Der Hase belädt den Wagen mit dieser Funktion. Als Argument kann für *Wagen* `Wagen1` und `Wagen2` angegeben werden. Diese Funktionen beinhaltet keine Synchronisationskonstrukte!

`entladeWagen( `***Wagen*** `)` Entlädt einen Wagen. Diese Arbeit muss leider von *Hase4* allein erledigt werden. Als Argument kann für *Wagen* `Wagen1` und `Wagen2` angegeben werden.

a) Initialisierungen (6)

Initialisieren Sie die notwenigen Semaphore:

b) (12)

Entwerfen Sie die Prozesse, die in den Hasen *Hase1* und *Hase2* ablaufen:

Prozess *Hase1*:

```
do forever() {

}
```

Prozess *Hase2*:

```
do forever() {

}
```

c) (12)

Entwerfen Sie die Prozesse, die in den Hasen *Hase3* und *Hase4* ablaufen:

Prozess *Hase3*:

```
do forever() {

}
```

Prozess *Hase4*:

```
do forever() {

}
```

4 Speicherverwaltung (25)

a) Segmentierung (6)

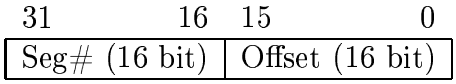
Das im folgenden beschriebene Speicherverwaltungssystem verwendet zur Adressierung 32-bit Adressen (virtuell & physikalisch). Für die angegebenen virtuellen Speicheradressen sind die entsprechenden physikalischen Adressen zu ermitteln. Von den angebenen Adressen sind die niederwertigen 16 Bit der Offset der Adresse. Die Verwendung der höherwertigen 16 Bit ist aus dem angegebenen Adressformat ersichtlich.

Alle Werte sind als Hexadezimalzahlen angegeben. Ergibt sich bei der Umwandlung aufgrund einer ungültigen Segmentnummer eine ungültige Adresse, so schreiben Sie bitte **"ungültig/Nummer"** in das entsprechende Feld. Sollte sich aufgrund eines unzulässigen Offsets eine ungültige Adresse ergeben, so schreiben Sie bitte **"ungültig/Offset"** in das entsprechende Feld.

Der Zugriff auf die Segmenttabellle erfolgt assoziativ (associative mapping). Es werden folgende Begriffe (englische Notation) aus dem Buch zur Vorlesung verwendet:

Base	Basisadresse des Segmentes	Virt.Addr.	Virtuelle Adresse
Length	Länge des Segmentes	Seg#	Segmentnummer

Das verwendete Adressformat ist folgendes:



Verwenden Sie für die Adressumsetzung folgende Segmenttabelle:

Segmenttabelle		
Seg#	Base	Length
0xFFFF	0x15FF B000	0xCCDD
0x1122	0x0808 B000	0x1001
0x1234	0x1266 0000	0x8000

Ermitteln Sie unter Benützung obiger Segmenttabelle die physikalischen Adressen zu folgenden virtuellen Adressen:

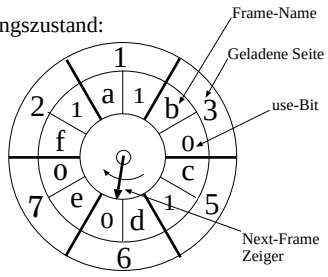
Virtuelle Adresse	Physikalische Adresse (zu ermitteln)
0xFFF1 3201	
0x1234 8000	
0x1234 8001	
0x1234 2001	
0x1121 1003	
0xFFFF 1503	

b) Replacement Policy (14)

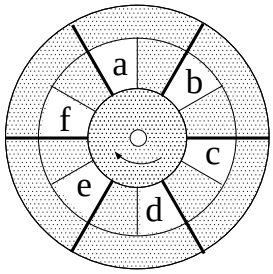
Simulieren Sie das Verhalten einer *clock policy* zum Auslagern von Speicher-Seiten, wenn neue Seiten geladen werden müssen.

Für jede Seite im Speicher existiert ein *use*-Bit. Die einzelnen Plätze im Hauptspeicher sind in diesem Beispiel mit Buchstaben von `a` ... `f` benannt. Der *next frame*-Zeiger bewegt sich in den verwendeten Grafiken im Uhrzeigersinn weiter.

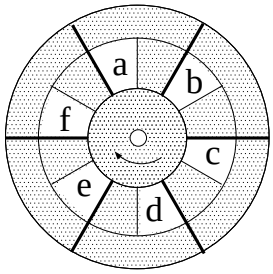
Anfangszustand:



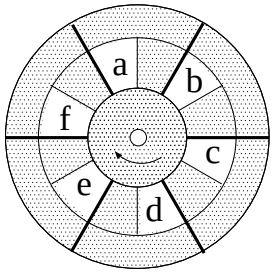
1.Zugriff: `access[1]`



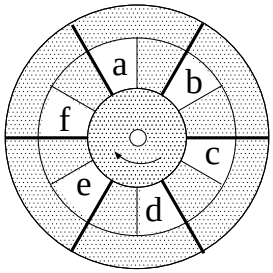
2.Zugriff: `access[8]`



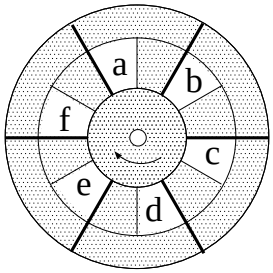
3.Zugriff: `access[3]`



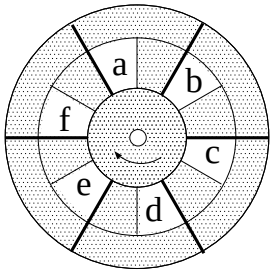
4.Zugriff: `access[6]`



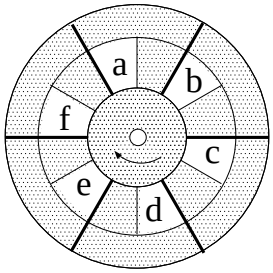
5.Zugriff: `access[7]`



6.Zugriff: `access[3]`



7.Zugriff: `access[9]`



Zeichnen Sie nun in den obigen Grafiken

- die Position des *next frame*-Zeigers
- die Inhalte aller Frames (geladene Seiten)

für die Ausführung von Befehlen mit Speicherzugriffen ein. Ein Befehl `access[x]` steht für einen Speicherzugriff auf die Seite `x`. Geben Sie den gefragten Speicherzustand **nach** der Ausführung des jeweiligen Befehls an, wobei Sie für die Befehle von einer sequentiellen Reihenfolge ausgehen können.

c) (3)

Nennen Sie drei Szenarien für den fehlerfreien Fall, unter denen Programme von nicht privilegierten Benutzern die gleiche physikalische Speicheradresse benutzen:

1.

2.

3.

d) (2)

Um wieviel Prozent **vergrößert** (nicht vervielfacht!) sich der virtuelle Adressraum, wenn Sie statt 32 bit langen 34 bit lange virtuelle Adressen verwenden?

Der virtuelle Adressraum vergrößert sich um % .