Vorlesungsprüfung aus Systemprogrammierung 2001-01-24

KNr. MNr. Zuname, Vorname

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

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Synchronisation (30)

Die Werkstätte der Firma 'Ho Ruck' hat zusätzlich zu ihren Arbeitsgeräten eine neue Drehbank bekommen. Jedes Gerät kann zu jeder beliebigen Zeit gebraucht werden. Unglücklicherweise ist der Stromverbrauch, wenn alle elektrischen Geräte eingeschaltet sind, jetzt grösser als die Zuleitung zur Werkstätte verkraftet. Als Provisorium installiert der Elektriker ein System, in dem Geräte nur eingeschaltet werden können, solange der Strom in der Leitung nicht zu gross wird.

a) (4)

Welche Art von Synchronisation wird dafür benötigt?

◯ keinerlei Synchronisation
◯ wechselseitiger Ausschluss
◯ Bedingungssynchronization
◯ Mutual & Condition Synchronisation

Die Nennleistung der Zubringerleitung ist **16 kW**.

Stk	Maschinenart	Leistungsbedarf / Stk
3	Hebebühnen	3.5kW
1	Bestoßmaschine	4.5kW
1	neue Drehbank	6.0kW

Tabelle 1: Auflistung der Verbraucher

b) (4)

Der Elektriker installiert folgende Routinen für ein Gerät mit einer Leistung von `r` kW. **Achtung:** Eine Leistungseinheit stellt **0.5 kW** dar.

```
vor_dem_Einschalten(Leistungsbedarf r){
  for(int i=0;i<(2*r);i++){
    P(A);
  }
}


nach_dem_Ausschalten(Leistungsbedarf r){
  for(int i=0;i<(2*r);i++){
    V(A);
  }
}
```

Das Semaphor `A` wurde mit der maximal verfügbaren Anzahl an Leistungseinheiten initialisiert.

```
initS(A,32);
```

Das schlaue Lehrmädchen meint dazu: „Mir scheint, das ist nicht der Weisheit letzter Schluss!“ Welche(s) Problem(e) kann/können bei dieser Lösung auftreten?

◯ Deadlock
◯ Starvation
◯ Busy Waiting
◯ Überlastung der Zuleitung

c) (22)

Glücklicherweise stellt das verwendete System neben Semaphoren auch Routinen für Eventcounter zur Verfügung:

- `initEC(E,Zahl)` legt einen Eventcounter an und initialisiert ihn mit `Zahl`.
- `await(E,Zahl)` wartet solange bis der Eventcounter `E` grösser gleich dem Wert von `Zahl` ist.
- `read(E)` liest den Wert des Eventcounters aus.
- `advance(E,Zahl)` erhöht den Eventcounter um den Wert `Zahl`.

Gemeinsam erstellen die beiden nun ein Programm, welches einen Semaphor und zwei Eventcounter verwendet.

Der erste Eventcounter (`ECused`) entspricht der Anzahl der verbrauchten Ressourcen im System. Der zweite Eventcounter (`ECavail`) entspricht der Anzahl der zurückgegebenen Ressourcen plus der noch verfügbaren Ressourcen im System.

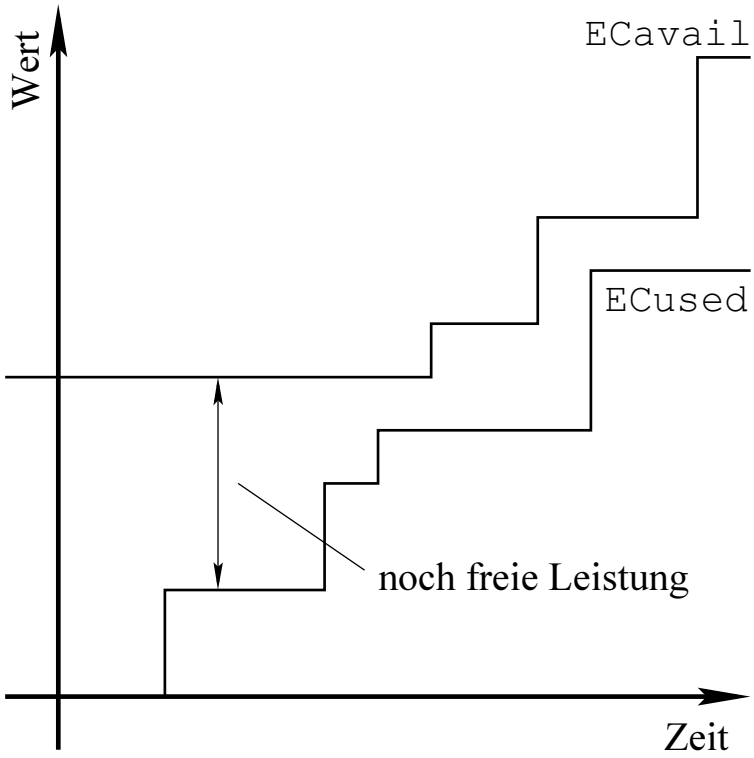


Abbildung 1: Bild eines Leistungsbedarfes

1. Ein Gerät wartet zunächst bis `ECavail` mindestens den Wert Leistungsbedarf des Gerätes plus dem Wert von `ECused` erreicht hat.

2. Danach testet das Gerät, ob inzwischen ein anderes Gerät möglicherweise weitere Ressourcen verbraucht hat. Dazu greift es auf beide EC zu und prüft, ob die Differenz mindestens dem Leistungsbedarf des Gerätes entspricht. Ist die Differenz kleiner als der Leistungsbedarf des Gerätes, so muss es den Vorgang bei Schritt 1 wiederholen. Ansonsten erhöht das Gerät `ECused` um den Wert seines Leistungsbedarfes und schaltet sich ein.

3. Wenn das Gerät ausgeschaltet wird, erhöht es `ECavail` um den Wert seines Leistungsbedarfes.

Beachten Sie, dass manche Programmteile möglicherweise exclusiv ausgeführt werden müssen. Weiters darf kein `goto` verwendet werden (umwandeln in eine Schleife). Wie kann eine solche Lösung aussehen?

Initialisierung:

```

```

```
vor_dem_Einschalten(Leistungsbedarf r){
```

```

```

```
}
```

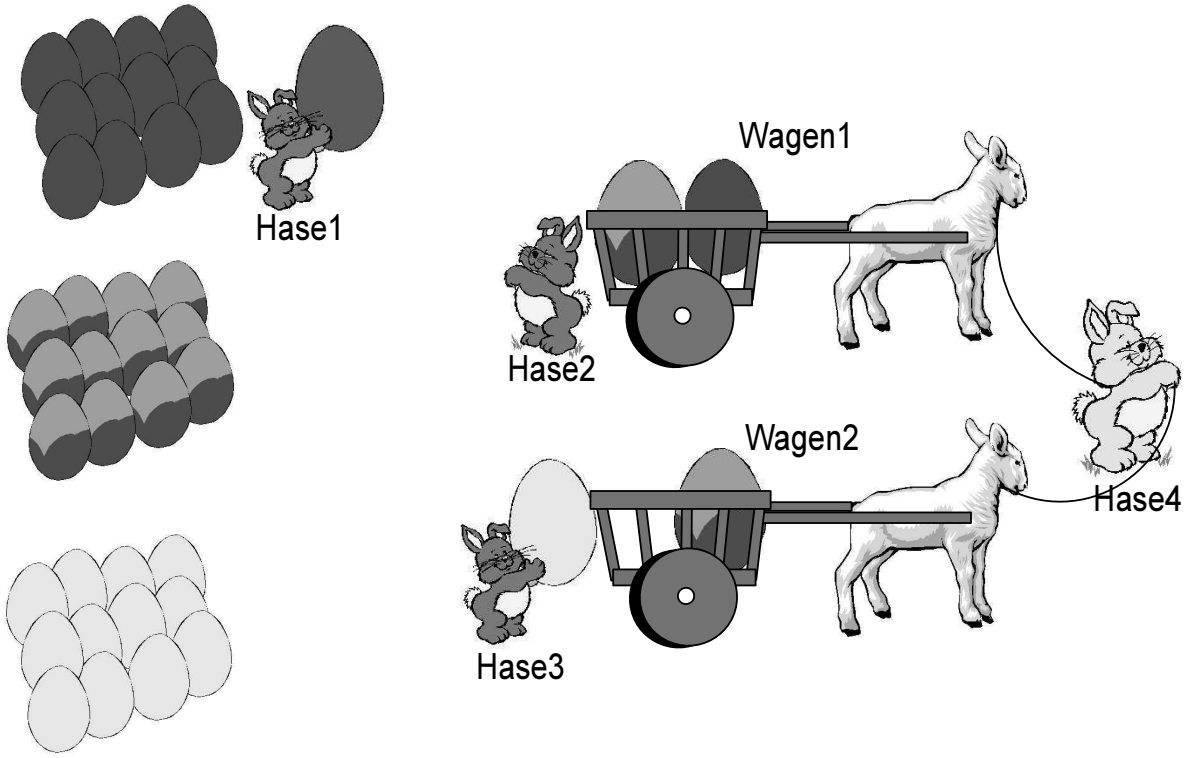
```
nach_dem_Ausschalten(Leistungsbedarf r){
```

```

```

```
}
```

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() {

}
```

Vorlesungsprüfung aus Systemprogrammierung 2001-05-16

KNr. MNr. Zuname, Vorname

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

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Synchronisation (30)

Bei der Studienbeihilfenbehörde: Es gibt **ein Büro mit zwei Sachbearbeitern**. Deshalb können sich im Büro maximal zwei Studenten aufhalten. Der Student kann durch die Glastüre sehen, ob es einen Sachbearbeiter gibt, der keinen Studenten betreut. Ein Sachbearbeiter wartet so lange, bis ein Student eintritt und auf sich aufmerksam macht. Ein freier Sachbearbeiter wird nun aufsehen und den Studenten bitten seine Matrikelnummer zu nennen, um den entsprechenden Akt zu holen.

Der Sachbearbeiter geht daraufhin zum Aktenschrank: Ist der Akt nicht im Schrank bedeutet dies, dass er gerade vom Kollegen bearbeitet wird. In diesem Fall wird dies dem Studenten mitgeteilt und er wird gebeten sich draußen am Ende der Reihe anzustellen (Das Programm Student terminiert und wird neu gestartet). Ist der Akt verfügbar, holt ihn der Sachbearbeiter vom Aktenschrank. **Der Student wartet bis der** seinen Fall bearbeitende **Sachbearbeiter beim Schreibtisch ankommt**. Dann bringt der Student sein Anliegen vor und nach der Klärung des Problems mit dem Sachbearbeiter verabschiedet er sich und verlässt das Büro.

a) Erstellen und initialisieren Sie alle benötigten Semaphore. (4)

Zum Anlegen eines Semaphores steht Ihnen das Statement `SemInit` zur verfügung. Mit diesem kann man sowohl einfache Semaphore (`Seminit(SemaphorName,InitWert)`) als auch Semaphorarrays (`Seminit(SemaphorArrayName[Anzahl],InitWert1,InitWert2,...)`) erzeugen und initialisieren. Der Aufruf `P(SemaphorName)` sperrt eine Semaphor und mit `V(SemaphorName)` gibt man es frei! (**Personalnummern der Sachbearbeiter:** $\in \{0, 1\}$.)

```
#define ANZ_SB 2
```

b) Vervollständigen Sie das Programm für einen Studenten. (14)

Es stehen Ihnen die Routinen `BueroBetreten` und `BueroVerlassen` zur Verfügung, welche keinerlei Rückgabewerte liefern, sowie die Funktion `SBAuswaehlen` welche die Personalnummer (0 oder 1) des aufsehenden Sachbearbeiters zurückgibt. Um dem Sachbearbeiter seine Matrikelnummer zu nennen und den Verbleib des Aktes zu erfahren verwende man `MatNrNennen(MatNr)`. Diese Funktion gibt einen negativen Wert zurück wenn der Akt gerade bearbeitet wird. **(C Syntax ist nicht erforderlich; es reicht Pseudocode**

Personalnummern der Sachbearbeiter: $\in \{0, 1\}$**.)**

```
/* Deklaration von Variablen. */
```

```
/* Warten, bis ein Platz im Buero frei ist. */
```

```
/* Buero betreten */
```

```
/* Begruessen und einen freien Sachbearbeiter *
 * in seiner Beschaeftigung unterbrechen.     */
Begruessen
```

```
/* Den SB auswaehlen und die Matrikelnummer    *
 * nennen. Wenn der entsprechende Ordner       *
 * nicht frei ist, das Buero wieder verlassen. */
```

```
if(                                  ){
  /* Der Akt ist nicht verfuegbar; Buero verlassen */
```

```
} else {
  /* Der Akt ist verfuegbar; warten bis ihn der *
   * Bearbeiter bei sich am Tisch hat.          */
```

```
  AnliegenVortragen
  AufSpaeterVertroestenLassen
  MitDankVerabschieden
  /* Das Buero Verlassen */
```

```
}
```

c) Vervollständigen Sie das Programm eines Sachbearbeiters (12)

Verwenden Sie die Routinen: `ZumAktenschrankGehen` und `ZumSchreibtischGehen`, sowie `AktHerausnehmen(Matrikelnummer)` und `AktZurueckstellen(Matrikelnummer)`. Weiters existiert eine Funktion mit Namen `WoIstDerAkt(Matrikelnummer)`, welche den Verbleib des Aktes signalisiert (`AKT_IST_BEIM_KOLLEGEN` bzw. `AKT_IST_VERFUEGBAR`). Um die Matrikelnummer des Studenten zu erhalten, verwende man `MatNrFragen`. Die Konstante `PERSONAL_NR` ist ebenfalls vordefiniert.

(**Personalnummern der Sachbearbeiter:** $\in \{0, 1\}$.)

```
/* Deklaration der Variablen. */
```

```
while((Zeit>=0900)&&(Zeit<=1200)){
  /* Warten bis ein Student kommt */
```

```
  Begruessen
  /* Anhand der Matrikelnummer entscheiden, wo der Akt ist. */
```

```
  if(                              != AKT_BEIM_KOLLEGEN){
```

```
    /* Der Akt ist verfuegbar! Na dann holen wir ihn. */
```

```

```

```
    /* Dem Studenten signalisieren, dass man bereit zur Arbeit ist. */
```

```

```

```
    DemAnliegenLauschen
    AufSpaeterVertroesten
    Verabschieden
    /* Den Akt zurueckstellen */
```

```

```

```
  }
}
```

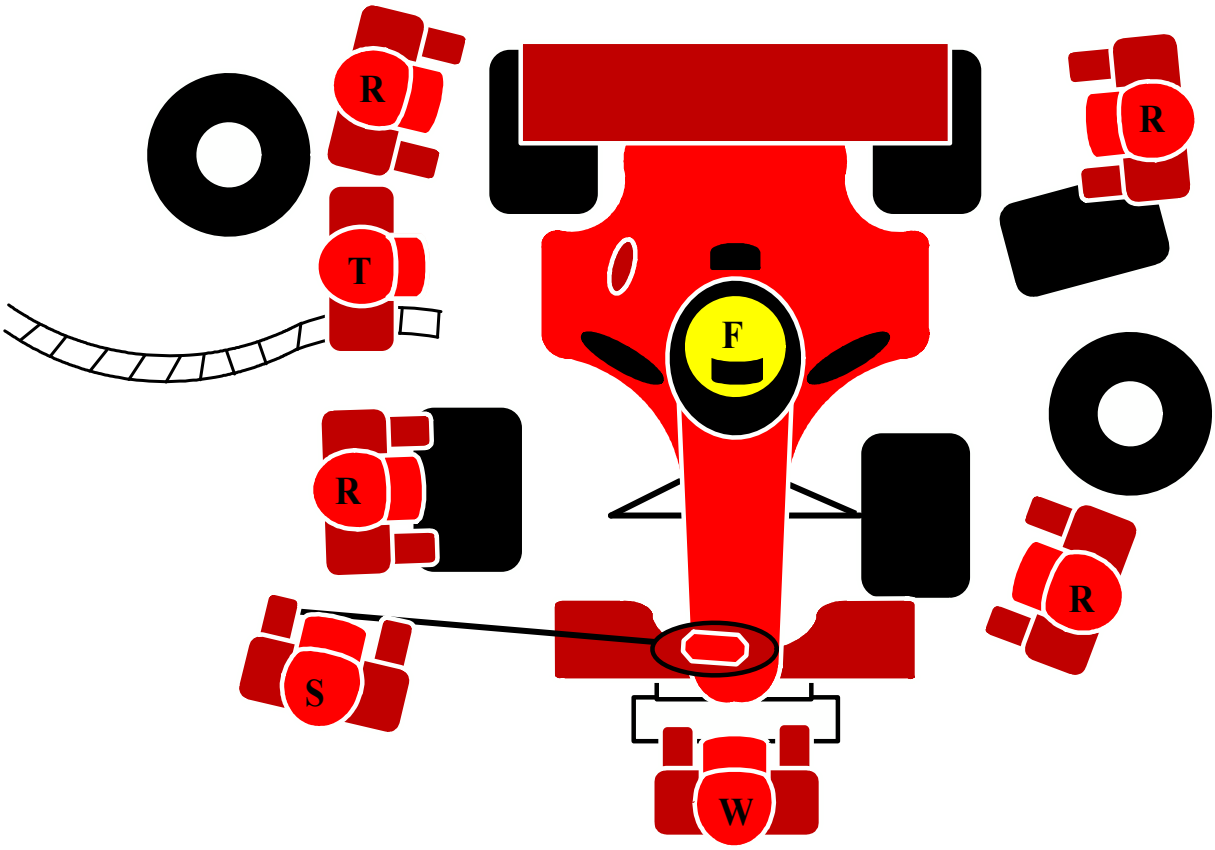
Vorlesungsprüfung aus Systemprogrammierung 2001-10-24

KNr. MNr. Zuname, Vorname

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

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Synchronisation (30)



Ein **Boxenstopp** in der Formel 1 läuft folgendermaßen ab: Sobald der Wagen eingefahren ist, beginnt der Tankwart (T) mit dem auffüllen des Tanks, während der Mann mit dem Wagenheber (W) den Wagen anhebt. Ist der Wagen in der Luft, beginnen die vier Reifenwechsler (R) mit dem Austausch der Reifen. Sind alle vier Reifen gewechselt, lässt der Mann mit dem Wagenheber das Fahrzeug wieder zu Boden und geht mit dem Wagenheber

zur Seite. Wenn der Wagenheber beiseite geräumt *und* der Tankvorgang beendet ist, gibt der Signalmann dem Fahrer das Signal zum Wegstarten.

Bedenken Sie bitte, dass es beim **Boxenstopp** auf optimale Synchronisation zwischen den Boxenmitarbeitern untereinander und zwischen Box und Fahrer ankommt, um keine wertvolle Zeit zu verlieren. Verwenden Sie außerdem nur unbedingt notwendige Synchronisationskonstrukte um dieses Ziel zu erreichen.

Zu verwendende Funktionen:

`initE(` ***Name*** `)` Legt einen Eventcounter mit dem angegebenen Namen *Name* an und initialisiert ihn mit der Zahl 0. Danach können die Funktionen **Advance(*Name*)** und **Wait(*Name*, *Wert*)** auf den Eventcounter angewendet werden.

`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.

`reFuel()` Initiert einen Tankvorgang. Diese Funktion wird vom Tankwart ausgeführt und blockiert, bis der Tankvorgang beendet ist und der Tankwart aus dem Gefahrenbereich verschwunden ist.

`changeTire()` Wechselt den Reifen. Diese Funktion wird von den Reifenwechslern ausgeführt und blockiert, bis der Vorgang beendet ist und der Reifenwechsler aus dem Gefahrenbereich verschwunden ist.

`liftCar()` Hebt den Wagen an. Diese Funktion wird vom Mann mit dem Wagenheber ausgeführt und blockiert, bis der Vorgang beendet ist.

`lowerCar()` Lässt den Wagen wieder sinken. Diese Funktion wird vom Mann mit dem Wagenheber ausgeführt und blockiert, bis der Vorgang beendet ist.

`removeJack()` Entfernt den Wagenheber aus dem Gefahrenbereich. Diese Funktion wird vom Mann mit dem Wagenheber ausgeführt und blockiert, bis der Vorgang beendet ist und der bedienende Mann aus dem Gefahrenbereich verschwunden ist.

Synchronisieren Sie den Arbeitsablauf der sieben Mitarbeiter des Boxenteams für *einen einzigen* Boxenstopp mittels **Eventcounter**. Sie können davon ausgehen, dass am Anfang der Wagen zum Stillstand gekommen ist, der Wagenheber bereits unter dem Wagen positioniert ist, der Wagen aber noch nicht angehoben ist und weder Reifenwechsler noch Tankwart mit ihrer Arbeit begonnen haben.

Synchronisieren Sie weiters die Kommunikation zwischen Fahrer und Signalgeber mittels **Semphor(e)**.

a) Initialisierungen (5)

Nehmen Sie hier die notwendigen Initialisierungen, welche beim Einfahren in die Box ausgeführt werden, vor:

b) (5)

Entwerfen Sie hier das Programm für einen beliebigen Reifenwechsler (R):

c) (5)

Entwerfen Sie hier das Programm für den Mann der den Wagenheber bedient (W):

d) (5)

Entwerfen Sie hier das Programm für den Signalgeber (S):

e) (5)

Entwerfen Sie hier das Programm für den Tankwart (T):

f) (5)

Entwerfen Sie hier das Programm für den Fahrer (F):

```
gib_gas();
```

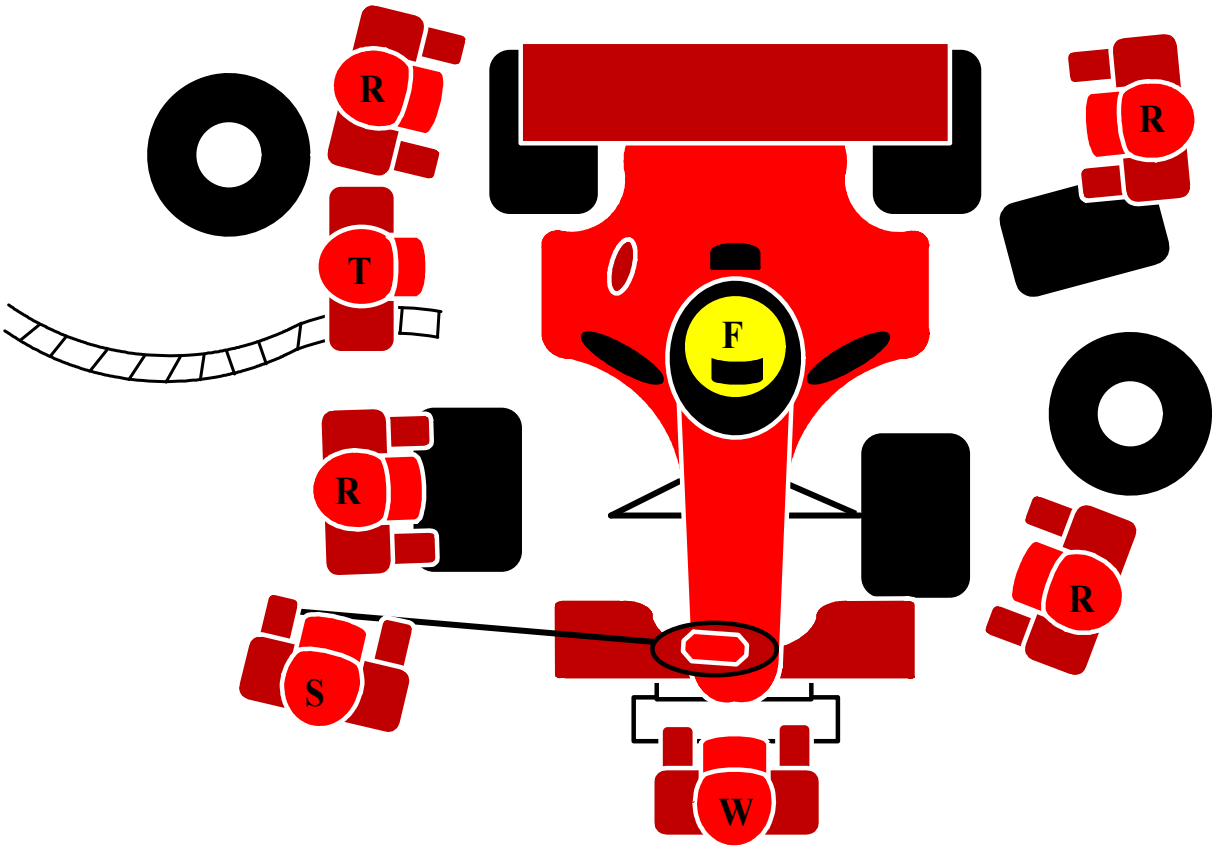
Vorlesungsprüfung aus Systemprogrammierung 2001-10-24

KNr. MNr. Zuname, Vorname

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

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Synchronisation (30)



Ein **Boxenstopp** in der Formel 1 läuft folgendermaßen ab: Sobald der Wagen eingefahren ist, beginnt der Tankwart (T) mit dem auffüllen des Tanks, während der Mann mit dem Wagenheber (W) den Wagen anhebt. Ist der Wagen in der Luft, beginnen die vier Reifenwechsler (R) mit dem Austausch der Reifen. Sind alle vier Reifen gewechselt, lässt der Mann mit dem Wagenheber das Fahrzeug wieder zu Boden und geht mit dem Wagenheber

zur Seite. Wenn der Wagenheber beiseite geräumt *und* der Tankvorgang beendet ist, gibt der Signalmann dem Fahrer das Signal zum Wegstarten.

Bedenken Sie bitte, dass es beim **Boxenstopp** auf optimale Synchronisation zwischen den Boxenmitarbeitern untereinander und zwischen Box und Fahrer ankommt, um keine wertvolle Zeit zu verlieren. Verwenden Sie außerdem nur unbedingt notwendige Synchronisationskonstrukte um dieses Ziel zu erreichen.

Zu verwendende Funktionen:

`initE(` ***Name*** `)` Legt einen Eventcounter mit dem angegebenen Namen *Name* an und initialisiert ihn mit der Zahl 0. Danach können die Funktionen **Advance(*Name*)** und **Wait(*Name*, *Wert*)** auf den Eventcounter angewendet werden.

`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.

`reFuel()` Initiert einen Tankvorgang. Diese Funktion wird vom Tankwart ausgeführt und blockiert, bis der Tankvorgang beendet ist und der Tankwart aus dem Gefahrenbereich verschwunden ist.

`changeTire()` Wechselt den Reifen. Diese Funktion wird von den Reifenwechslern ausgeführt und blockiert, bis der Vorgang beendet ist und der Reifenwechsler aus dem Gefahrenbereich verschwunden ist.

`liftCar()` Hebt den Wagen an. Diese Funktion wird vom Mann mit dem Wagenheber ausgeführt und blockiert, bis der Vorgang beendet ist.

`lowerCar()` Lässt den Wagen wieder sinken. Diese Funktion wird vom Mann mit dem Wagenheber ausgeführt und blockiert, bis der Vorgang beendet ist.

`removeJack()` Entfernt den Wagenheber aus dem Gefahrenbereich. Diese Funktion wird vom Mann mit dem Wagenheber ausgeführt und blockiert, bis der Vorgang beendet ist und der bedienende Mann aus dem Gefahrenbereich verschwunden ist.

Synchronisieren Sie den Arbeitsablauf der sieben Mitarbeiter des Boxenteams für *einen einzigen* Boxenstopp mittels **Eventcounter**. Sie können davon ausgehen, dass am Anfang der Wagen zum Stillstand gekommen ist, der Wagenheber bereits unter dem Wagen positioniert ist, der Wagen aber noch nicht angehoben ist und weder Reifenwechsler noch Tankwart mit ihrer Arbeit begonnen haben.

Synchronisieren Sie weiters die Kommunikation zwischen Fahrer und Signalgeber mittels **Semphor(e)**.

a) Initialisierungen (5)

Nehmen Sie hier die notwendigen Initialisierungen, welche beim Einfahren in die Box ausgeführt werden, vor:

b) (5)

Entwerfen Sie hier das Programm für einen beliebigen Reifenwechsler (R):

c) (5)

Entwerfen Sie hier das Programm für den Mann der den Wagenheber bedient (W):

d) (5)

Entwerfen Sie hier das Programm für den Signalgeber (S):

e) (5)

Entwerfen Sie hier das Programm für den Tankwart (T):

f) (5)

Entwerfen Sie hier das Programm für den Fahrer (F):

```
gib_gas();
```

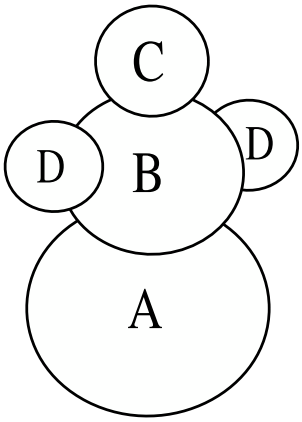
Vorlesungsprüfung aus Systemprogrammierung 2002-01-23

KNr. MNr. Zuname, Vorname

Ges.)(100)	1.)(35)	2.)(20)	3.)(20)	4.)(25)	Zusatzblätter:

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Synchronisation (35)



Die Informatik-Studenten feiern das kommende Semesterende, indem sie gemeinsam Schneemänner bauen. Die einzelnen Teile eines Schneemannes werden gemäß obigen Bauplanes mit *A* bis *D* benannt.

Sie wollen Algorithmen zum Bauen des Schneemannes entwickeln:

- Es sind die Funktionen **setze_A()**, **setze_B()**, **setze_C()** und **setze_D()** zu entwickeln, die jeweils das bezeichnete Teil am nächst möglichen Schneemann anbringen. Jeder Student kann *mehrmals* beliebig eine dieser Funktionen verwenden, auch *gleichzeitig* mit anderen Studenten. Die jeweilige Funktion soll solange blockieren, bis ein Schneemann frei ist, an dem das Teil montiert werden kann.

Die erforderten vorhandenen Teile für das Anbringen eines neuen Teiles sind aus folgender Tabelle auszulesen:

Anzubringendes Teil	Benötigte(s) Teil(e)
A	-
B	A
C	A B
D	A B

- Es steht ihnen dazu Funktion **setze(x)** zum Plazieren eines Teiles zur Verfügung, wobei `x` ein Teil aus `{A,B,C,D}` bezeichnet. Die Funktion fügt den Teil automatisch am ersten Schneemann an, der diesen noch nicht vollständig enthält.

a) Die "parallele Klasse" (18)

Die Studenten der "parallelen Klasse" haben folgende Regel festgelegt:

- Es darf an maximal **MAX** Schneemännern gleichzeitig gebaut werden. Ein weiterer Schneemann darf erst angefangen werden, wenn ein anderer fertig geworden ist.

Die Synchronisation ist mit (*möglichst wenig!*) Semaphoren durchzuführen. Verwenden Sie zur Initialisierung der Semaphoren die Funktion **init(s,v)**, welche als ersten Parameter den Semaphor und als zweiten Parameter den entsprechenden Initialisierungswert erhält. Danach können die Funktionen **P(s)** und **V(s)** auf den Semaphor angewendet werden.

Die Verwendung von globalen Variablen bzw. Busy-Waiting zur Synchronisation ist verboten!

a1) Initialisierungen (3)

Geben Sie hier die notwendigen Initialisierungen von Semaphoren an.

a2) (5)

Programm zum Platzieren einer Kugel A:
`setze_A()`

```
BEGIN
```



```
END
```

a3) (4)

Program zum Platzieren der Kugel B:
`setze_B()`

```
BEGIN
```



```
END
```

a4) (3)

Program zum Platzieren der Kugel C:
`setze_C()`

```
BEGIN
```



```
END
```

a5) (3)

Programm zum Platzieren eines Armes D:
`setze_D()`

```
BEGIN
```



```
END
```

b) Die "sequentielle Klasse" (17)

Die Studenten der "sequentiellen Klasse" haben folgende Regel festgelegt:

- Es darf an maximal **einem** Schneemann gleichzeitig gebaut werden. Ein weiterer Schneeman darf erst angefangen werden, wenn der vorherige fertig geworden ist.

Die Synchronisation ist mit (*möglichst wenig!*) Sequencern und Eventcountern durchzuführen. Auf Sequencer kann die Funktion **sticket(s)** angewendet werden. Auf Eventcounter können die Funktionen **eawait(e,v)** und **eadvance(e)** angewendet werden.

Die Verwendung von globalen Variablen bzw. Busy-Waiting zur Synchronisation ist verboten!

b1) Deklarationen (3)

Geben Sie hier die Deklarationen von Sequenzern/Eventcountern an. Verwenden Sie hierzu die Notation `Sequencer X;` bzw. `Eventcounter Y;`.

b2) (4)

Programm zum Platzieren einer Kugel A:
`setze_A()`

```
BEGIN
```



```
END
```

b3) (3)

Program zum Platzieren der Kugel B:
`setze_B()`

```
BEGIN
```



```
END
```

b4) (4)

Program zum Platzieren der Kugel C:
`setze_C()`

```
BEGIN
```



```
END
```

b5) (3)

Programm zum Platzieren eines Armes D:
`setze_D()`

```
BEGIN
```



```
END
```

Vorlesungsprüfung aus Systemprogrammierung 2002-03-20

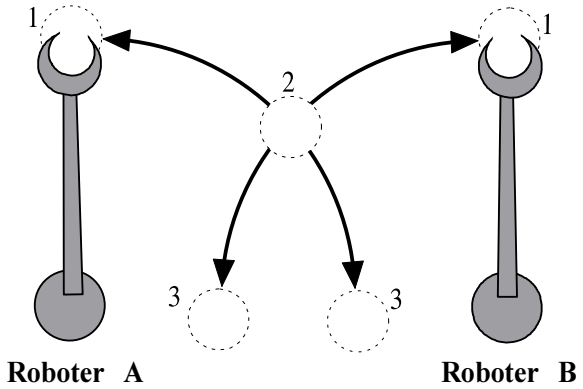
KNr. MNr. Zuname, Vorname

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

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Synchronisation (30)

In einer Fabrik sind zwei Roboter so angeordnet, dass sich ihre Arbeitsbereiche teilweise überlappen. Jeder Roboter kann eine von drei Arbeitspositionen anfahren, um Objekte aufzunehmen oder abzulegen.



Implementieren Sie die Teile eines Programms `Transport`(*von*, *nach*), welches einen Roboter ein Objekt von Position *von* nach Position *nach* legen lässt.

Ihnen stehen folgende Funktionen zur Verfügung:

`Grab()` lässt den Roboter ein Objekt von jener Position aufnehmen, auf welcher der Roboterarm gerade steht.

`Release()` lässt den Roboter ein Objekt an jener Position ablegen, auf welcher der Roboterarm gerade steht.

`GoTo(`***Position***`)` Fährt den Roboterarm an die angegebene Position.

Beachten Sie folgende Randbedingungen:

- Die Arme der Roboter A und B dürfen sich niemals überkreuzen. Weiters dürfen sich beide Arme nie gleichzeitig auf Position 2 oder 3 befinden. Position 2 kann aber z. B. angefahren werden, wenn der andere Greifer auf Position 3 steht.
- Gehen Sie davon aus, dass bei Aufruf der Funktion `Transport`() ein Objekt auf *von* vorhanden ist, die Robotergreifer auf Position 1 stehen und kein Objekt gegriffen wurde.
- Es können beliebig viele Objekte an einem Ort abgelegt werden, d.h es muss *nicht* überprüft werden, ob eine Position frei ist.
- Stellen Sie den Greifer nach erfolgter Aktion in die Position 1 zurück.
- Gehen Sie davon aus, dass je eine Instanz von Transport() auf jedem der Roboter gleichzeitig laufen kann.
- Stellen Sie sicher, dass während des Be- oder Entladens der andere Roboter die nicht blockierten Positionen anfahren kann, sofern dadurch kein Deadlock entstehen kann.
- Verwenden Sie ausschließlich Semaphore zur Synchronisation. Es sind möglichst wenig Synchronisationskonstrukte zu verwenden, die Verwendung von globalen Variablen ist verboten.

Verständnisfrage

Zu welchen Problemen kann es kommen, wenn Roboter A nicht jedesmal auf Position 1 zurückgestellt wird, sondern auf der letzten Entladeposition stehenbleiben würde? Kreuzen Sie die richtige(n) Antwort(en) an und begründen Sie Ihre Antwort!

◯ Deadlock ◯ Starvation ◯ Lifelock

Implementierung

Legen Sie die notwendigen Synchronisationskonstrukte mit der Funktion InitSem(*Semaphorname*,*Initialwert*) an und initialisieren Sie sie entsprechend der Situation auf dem Bild.

Transport()

Das Programm Transport(*von*, *nach*) verwendet die Unterprogramme Transport1To(*nach*), Transport2To(*nach*) und Transport3To(*nach*). Implementieren Sie diese, so dass Transport() wie angegeben funktioniert:

```
Transport(von,nach)
{
  switch(von) {
    case 1:
      Transport1To(nach);
      break;
    case 2:
      Transport2To(nach);
      break;
    case 3:
      Transport3To(nach);
      break;
  }
}
```

```
Transport1To(nach)
{
}
```

Vorlesungsprüfung aus Systemprogrammierung 2002-05-29

KNr. MNr. Zuname, Vorname

Ges.)(100) 1.)(35) 2.)(20) 3.)(25) 4.)(20) Zusatzblätter:

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Synchronisation (35)

In der Produktionsanlage eines Zulieferbetriebs werden Komponenten aus Aluminiumzylindern und Kunststoffringen von Industrierobotern hergestellt. Die Aluminiumzylinder und Kunststoffringe stehen in einem Materiallager bereit. Fertige Teile werden in einem Endlager verstaut. Die Abfolge der Arbeitsschritte bei der Produktion ist in Abbildung 1 veranschaulicht.

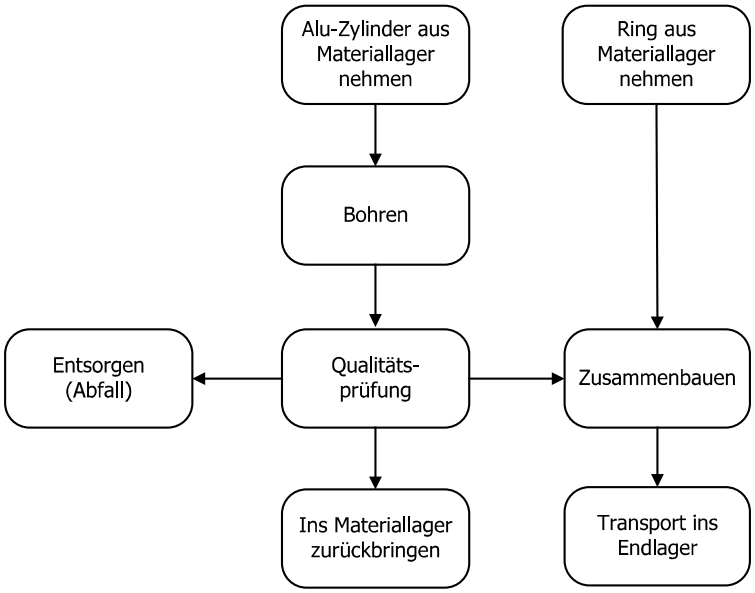


Abbildung 1: Abfolge der Arbeitsschritte

Für die Herstellung wird ein Kunststoffring an einem Aluminiumzylinder angebracht. Hierzu muss ein Aluminiumzylinder aus dem Materiallager geholt und gebohrt werden. Danach wird eine Materialprüfung durchgeführt, die den tatsächlich erzielten Bohrdurchmesser bestimmt. Ist der erzielte Bohrdurchmesser kleiner als 30 mm, so wird der Aluminiumzylinder

ins Materiallager retourniert, um später erneut gebohrt zu werden. Bei einem ermittelten Bohrdurchmesser über 31 mm muss der Aluminiumzylinder entsorgt werden. Nach erfolgreicher Materialprüfung, dh. der Bohrdurchmesser beträgt 30 mm oder 31 mm, kann ein Kunststoffring am Aluminiumzylinder angebracht und das fertige Teil ins Endlager abtransportiert werden.

Zur Durchführung dieser Verarbeitungsschritte stehen Industrieroboter bereit, die jeweils eine bestimmte Teilaufgabe ausführen können. Diese Roboter müssen außerdem Teile an andere Roboter weitergeben bzw. von diesen Teile übernehmen.

- Roboter 1 und Roboter 2 können Teile (Zylinder oder Ring) aus dem Materiallager holen.
- Roboter 3 führt Bohrungen am Zylinder durch und kann Teile ins Materiallager, sowie in die Entsorgungsstätte bringen.
- Roboter 4 ist mit der Qualitätsprüfung betraut und hat fehlerhafte Teile zu entsorgen.
- Roboter 5 kann zwei Teile zusammenbauen und fertige Teile ins Endlager transportieren.

Erstellen Sie die Programme der Industrieroboter und sorgen Sie für eine korrekte Abfolge der Arbeitsschritte durch Synchronisation mit Semaphoren!

Beachten Sie dabei folgende Hinweise:

- Die Verwendung von globalen Variablen bzw. Busy-Waiting zur Synchronisation ist verboten!
- Verhindern Sie das Auftreten von Deadlocks!
- Die Roboter 1, 2, 3 und 4 können zu einem bestimmten Zeitpunkt jeweils nur ein einziges Teil bearbeiten.
- Roboter 5 kann maximal zwei zum Zusammenbau bestimmte Teile aufnehmen.
- Achten Sie auf ein hohes Maß an Parallelismus bei der Verarbeitung der Teile durch die Roboter!
- Verwenden Sie möglichst wenige Semaphore!

a) Erstellen und initialisieren Sie alle benötigten Semaphore. (5)

Zum Anlegen eines Semaphores steht Ihnen das Statement `SemInit` zur Verfügung. Mit diesem kann man sowohl einfache Semaphore (`SemInit(SemaphorName,InitWert)`) als auch Semaphorarrays (`SemInit(SemaphorArrayName[Anzahl],InitWert1,InitWert2,...)`) erzeugen und initialisieren. Das Sperren eines Semaphors erfolgt durch den Aufruf von `P(SemaphorName)` bzw. `P(SemaphorArrayName[Index])`, mit `V(SemaphorName)` bzw. mit `V(SemaphorArrayName[Index])` gibt man es frei!

b) Vervollständigen Sie die Programme der Industrieroboter. (30)

Verwenden Sie hierzu die folgenden Routinen:

- *Nimm(roboter)* nimmt das Teil vom Industrieroboter *roboter* $\in$ *{roboter1,...,roboter5}* entgegen.
- *HoleAusLager(obj)* besorgt das Teil *obj* aus dem Materiallager. Diese Aktivität ist Roboter 1 und Roboter 2 vorbehalten.
  (obj $\in$ *{Aluzylinder,Kunststoffring}*).
- *Gib(ort)* reicht das Teil zum Standort *ort* weiter, wobei *ort* ein Industrieroboter *(roboter1,...,roboter5)*, das Endlager *(endlager)*, das Materiallager *(materiallager)*, oder die Entsorgungsstätte *(abfall)* ist. Wird das Teil an einen anderen Roboter übergeben, dann darf der Roboter nach dem Aufruf von *Gib(ort)* solange kein weiteres Teil aufnehmen, bis der andere Roboter das weitergereichte Teil mittels eines Aufrufs von *Nimm(roboter)* übernommen hat.
- *Bohren()* veranlasst einen Roboter zur Durchführung einer Bohrung am Teil. Diese Routine kann nur von Roboter 3 ausgeführt werden.
- Die Routine *int BestimmeDurchmesser()* ermittelt den erzielten Bohrdurchmesser in mm. Diese Routine kann nur von Roboter 4 ausgeführt werden.
- Zwei Teile werden mit der Routine *Zusammenbauen()* zu einem Teil verarbeitet. Diese Routine kann nur von Roboter 5 ausgeführt werden.

Roboter 1

Roboter 2

Roboter 3

Roboter 4

Roboter 5

2 Synchronisation (30)

Zum 5. Dezember wollen Sysprog Studenten ein "Krampuskränzchen" feiern. Dazu beabsichtigen sie, Maroni zu braten sowie Glühwein zu kochen. Folgende Randbedigngungen müssen berücksichtigt werden.

- Es gibt 5 Kochtöpfe (à 70 Portionen) ...
- sowie 3 Backbleche (à 120 Portionen).
- Der Ofen hat 3 Herdplatten und 1 Backrohr.
- Im Backrohr hat genau ein Backblech Platz.
- Wenn das Backrohr in Betrieb ist, darf nur 1 Herdplatte verwendet werden. Wird das Backrohr nicht verwendet, können alle 3 Herdplatten verwendet werden.
- Zu Beginn des "Krampuskränzchens" sind alle Backbleche und Töpfe leer.
- Erst wenn ein Topf bzw. Blech vollständig geleert ist, wird vom Nächsten entnommen.
- Die Gäste geben ein Backblech bzw. einen Topf nur vollständig entleert wieder heraus.
- Die Lösung soll ein hohes Maß an Parallelität besitzen und dabei keine globalen Variablen verwenden.

Es stehen Ihnen die folgenden Routinen für die Implementierung der drei Programme `hungriger_Student`, `Maroni_Brater` und `Gluehwein_Koch` zur Verfügung:

- `SInit(`*SQ*`,`*val*`)` Initialisiert den Sequencer *SQ* mit dem Wert *val*.
- `STicket(`*SQ*`,`*val*`)` Erhöht den Sequencer *SQ* um den Wert *val* und retourniert den **neuen** Wert.
- `EInit(`*EC*`,`*val*`)` Initialisiert den Eventcounter *EC* mit dem Wert *val*.
- `EWait(`*EC*`,`*val*`)` Ein Aufruf der Routine `EWait` kehrt zurück, sobald der Wert des Eventcounters $EC \geq val$ ist.
- `EAdvance(`*EC*`,`*val*`)` Erhöht den Wert des Eventcounters *EC* um *val*.
- `HoleGluehwein()` Ein durstiger Student kann durch Aufruf dieser Prozedur an Glühwein herankommen. Die Prozedur blockiert solange bis Glühwein vorhanden ist.
- `HoleMaroni()` Mit dieser Prozedur versorgt sich ein hungriger Student mit Maroni. Auch diese Prozedur blockiert bis Maroni verfügbar sind.
- `BlechHolen()` Ein Maronibrater nimmt durch diese Prozedur ein leeres Blech und stellt es bereit für den nächsten Arbeitsschritt.
- `MaroniAufsBlech()` Richtet **120 Portionen** Maroni auf einem bereitgestellten Backblech an.

- `BlechInDenOfen()` Stellt das Backblech in den Ofen, nimmt es in Betrieb und wartet bis die Maroni fertig sind. Beim Terminieren ist das Backrohr bereits abgedreht und frei.
- `TopfHolen()` Nimmt einen Topf und stellt diesen zum Glühweinmixen bereit.
- `GluehweinMixen()` Gibt Gewürze sowie Wein & Wasser für **70 Portionen** Glühwein in einen bereitgestellen Topf.
- `TopfAufDenOfen()` Setzt einen gefüllten Topf auf eine freie Herdplatte, nimmt diese in Betrieb und terminiert wenn der Glühwein fertig ist. Beim Terminieren ist die Herdplatte bereits abgedreht und frei.

2.1 Initialisierung (5)

2.2 Hungriger Student (3)

Implementieren Sie das Programm `hungriger_Student`, das auf allen Gästen parallel abläuft.

```
while(hungrig || durstig){
  if(durstig){
```



```
    durstig = Trinken() /* trinkt den Gluehwein & aktualisiert durstig */
  }
  if(hungrig){
```



```
    hungrig = Essen() /* Iszt die Maroni & aktualisiert hungrig */
  }
}
Heimtorkeln()
```

2.3 Maronibrater (11)

Implementieren Sie das Programm `Maroni_Brater` das auf allen Maronibratern parallel exekutiert wird.

```
while(fest == IM_GANGE){
```



```
  GebtDenHungerden()
}
Aufraeumen()
```

2.4 Glühweinkoch (11)

Implementieren Sie das Programm `Gluehwein_Koch` das auf allen Glühweinköchen parallel abgearbeitet wird.

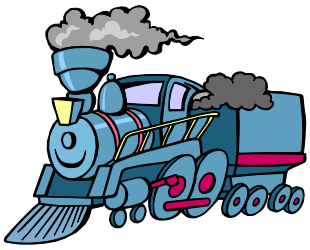
```
while(fest == IM_GANGE){
```

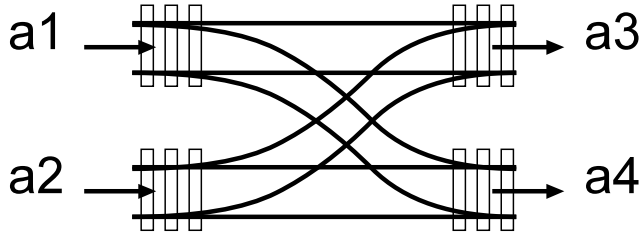


```
  GebtDenDurstigen()
}
Aufraeumen()
```

2 Synchronisation (30)

Der Verschubbahnhof *Innovativa* hat eine relativ einfache Gleisstruktur, wie in folgender Abbildung dargestellt ist:





Der Verschubbahnhof hat nur ein einziges Weichensystem, mit dem es möglich ist, von dem einen Gleis auf das andere zu wechseln. Die Anschlusspunkte des Weichensystems sind mit **a1, a2, a3** und **a4** benannt. Wichtig ist, dass entsprechend den eingezeichneten Pfeilen ein Zug nur von **a1** oder **a2** kommend nach **a3** oder **a4** fahren darf.

Für den Verschubbahnhof ist eine Synchronisations-Software zu entwickeln, um Zugkollisionen während des Passierens des Weichensystems zu vermeiden. Das Weichensystem stellt **drei Betriebsarten** zur Verfügung:

KEINE: Keine Züge dürfen passieren.

GERADE: Züge dürfen nur geradeaus passieren.

ALLE: Züge dürfen sowohl geradeaus als auch diagonal passieren.

Es sind folgende Funktionen zu implementieren:

zug_kontrolle(von, nach) zur Kontrolle von Zügen. Diese Funktion steuert einen Zug, fahrend von **von** nach **nach**. Der Zug soll entsprechend der aktuellen Weichenbetriebsart und anderen Zügen synchronisiert werden. Sollte **von**$\in\{$**a1,a2**$\}$ oder **nach**$\in\{$**a3,a4**$\}$ gelten (d.h. eine nicht im System vorgesehende Fahrtrichtung), so ist die unten beschriebene Funktion **setze_stop(von)** aufzurufen, wobei **von** die Richtung ist, aus der dieser Zug kommt (der Zug selbst bleibt in diesem Fall einfach stehen).
Wenn das Weichensystem passierbar ist (d.h., kein Konflikt mit der aktuellen Weichenbetriebsart und anderen Zügen), ist die Funktion **stelle_weichen(von, nach)** aufzurufen, um die Weiche korrekt zu stellen. Anschließend ist zum eigentlichen Durchfahren der Weichenanlage die Funktion **passiere()** aufzurufen.

setze_stop(von) zum Sichern der Weichenanlage. Der Parameter **von** bezeichnet die aktuelle Position eines Zuges (hier **von**$\in\{$**a1, a2, a3, a4**$\}$). Die Funktion soll für die Position **von** die Strecken zu den beiden gegenüberliegenden Weichenanschlusspunkten sperren. Ein Zug in gleicher Fahrtrichtung am Parallelgleis geradeaus fahrend, soll jedoch weiterhin passieren dürfen.

Beispiel: Angenommen, ein Zug kommt fälschlicherweise von (**von=a4**). Somit sind alle weiteren Züge kommend aus **a1** oder **a2** und nach **a4** fahrend, zu blockieren. Ein Zug, kommend von **a1** oder **a2** und nach **a3** fahrend, darf jedoch weiterhin passieren.

weichen_betriebsart_uebernahme() Diese Prozedur ermittelt in einer Endlosschleife durch **p = hole_perm()** die aktuelle Weichenbetriebsart **p ∈ {KEINE, GERADE, ALLE}** und setzt entsprechend die im Programm benötigten Synchronisationskonstrukte.

Implementieren Sie alle Funktionen sowie Initialisierungen derart, sodass **defaultmäßig Weichenbetriebsart GERADE** aktiv ist (z.B.: wegen mechanischem Fehler in der Weichenumschaltung). Das heißt, Züge dürfen defaultmäßig nur geradeaus passieren.

Hinweis: Der Fall, dass hinter einem wartenden Zug ein neuer Zug nachkommt, braucht nicht berücksichtigt zu werden.

Die Verwendung von globalen Variablen bzw. Busy-Waiting zur Synchronisation ist verboten!

a) Initialisierungen (4)

Die Synchronisation ist mit (*möglichst wenig!*) Semaphoren durchzuführen wobei unnötige Einschränkungen der Parallelität zu vermeiden sind. Verwenden Sie zur Initialisierung der Semaphoren die Funktion **init(s,v)**, welche als ersten Parameter den Semaphor und als zweiten Parameter den entsprechenden Initialisierungswert erhält. Danach können die Funktionen **P(s)** und **V(s)** auf den Semaphor angewendet werden.

Geben Sie hier die notwendigen Initialisierungen von Semaphoren an.

b) Setzen einer neuen Weichenbetriebsart (8)

Programm zum Setzen der Weichenbetriebsart für Wartungsarbeiten.
`weichen_betriebsart_uebernahme()`

```
BEGIN

  current = GERADE;
```



```
END
```

c) Zugkontrolle (14)

Programm zur Kontrolle eines Zuges:
`zug_kontrolle(von, nach)`

BEGIN

END

d) Stopsignal setzen (4)

Programm zum Stoppen der für einen Zug stehend auf `von` kritischen entgegenkommenden Züge:
`setze_stop(von)`

```
BEGIN
```

```

```

```
END
```

VO 182.022 29. Oktober 2003

Prüfung Betriebssysteme

KNr. MNr. Zuname, Vorname

Ges.)(100)	1.)(35)	2.)(25)	3.)(20)	4.)(20)	Zusatzblätter:

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Synchronisation (35)

Die Fähre *M/s Mariella* transportiert Fahrzeuge und Personen zwischen Helsinki und Stockholm. Um eine maximale Auslastung der Fähre zu erreichen, beschließt der Kapitän, die Fahrt erst dann zu beginnen, wenn die Fähre voll beladen ist.

Die Fähre besitzt Platz für 2500 Passagiere, 200 PKW und 30 LKW.

Implementieren Sie Synchronisationsroutinen für den Betrieb der Fähre. Berücksichtigen Sie dabei folgende Randbedingungen:

- Für die Verwendung von *Semaphoren* stehen Ihnen die Funktionen `initsem(`*`name`*`,` *`initialwert`*`)` und `P(`*`name`*`)` sowie `V(`*`name`*`)` zur Verfügung.
- Für die Verwendung von *Eventcountern* stehen Ihnen die Funktionen `initevc(`*`name`*`,` *`initialwert`*`)`, `read(`*`name`*`)`, `eawait(`*`name`*`)` und `advance(`*`name`*`,`*`value`*`)` zur Verfügung.
- Für die Verwendung von *Sequencern* stehen Ihnen die Funktionen `initseq(`*`name`*`,` *`initialwert`*`)`, `read(`*`name`*`)` und `sticket(`*`name`*`)`. Die Synchronisation mit globalen Variablen ist verboten.
- Verwenden Sie so wenig Synchronisationskonstrukte wie möglich.

1.1 Ressourcen anlegen und initialisieren

Die Initialisierung wird nur ein einziges Mal am Anfang aufgerufen. Bitte definieren Sie hier alle benötigten Ressourcen.

1.2 Kapitän

Der Kapitän legt mit dem Schiff an, koordiniert die Entladung der Fahrzeuge und schickt die Passagiere vom Schiff. Danach wartet der Kapitän bis das Schiff wieder voll beladen ist und die Fahrzeugrampe und der Passagierzugang geschlossen sind und legt dann wieder in Richtung Zielhafen ab. Das Beladen der Fähre mit PKW, LKW und das Aufnehmen von Personen soll gleichzeitig erfolgen.

```
do forever{
 Fahre_in_Hafen()
 Anlegen()
 Ankern()
 Installiere_Passagierrampe()
 Schicke_Passagiere_vom_Schiff()
```

```
 Oeffne_Fahrzeugrampe()
 Entlade_Fahrzeuge()
```

```
 Schliesse_Fahrzeugrampe()

 Entferne_Passagierrampe()

 Anker_lichten()

 Fahre_zum_Zielhafen()

}
```

1.3 PKW beladen

Dieses Programm dient zum Beladen der Fähre mit einem PKW. Es kann parallel mit anderen Programmen und mehrfach gestartet werden. Achten Sie darauf, dass die Fähre am Anlegeplatz steht, die Laderampe heruntergefahren ist und noch genug Platz auf der Fähre ist, andernfalls soll die Funktion blockieren, bis die nächste Fähre zum Beladen bereit ist. Diese Funktion kann gleichzeitig in mehreren Instanzen aufgerufen werden, wobei immer nur **ein** Parkvorgang (sowohl PKW als auch LKW) gleichzeitig durchgeführt werden können. Pro Fahrzeug muss jeweils ein Passagierplatz für den Fahrer reserviert werden.

```

Parke_PKW_auf_Fähre()

```

```
Fahrer_bezieht_Kabine()
```

1.4 LKW beladen

Dieses Programm dient zum Beladen der Fähre mit einem LKW. Es kann parallel mit anderen Programmen und mehrfach gestartet werden. Achten Sie darauf, dass die Fähre am Anlegeplatz steht, die Laderampe heruntergefahren ist und noch genug Platz auf der Fähre ist, andernfalls soll die Funktion blockieren, bis die nächste Fähre zum Beladen bereit ist. Diese Funktion kann gleichzeitig in mehreren Instanzen aufgerufen werden, wobei immer nur **ein** Parkvorgang (sowohl PKW als auch LKW) gleichzeitig durchgeführt werden darf. Pro Fahrzeug muss jeweils ein Passagierplatz für den Fahrer reserviert werden.

```

```

`Parke_LKW_auf_Fähre()`

```

```

`Fahrer_bezieht_Kabine()`

```

```

1.5 Neuer Passagier

Dieses Programm dient zum Aufnehmen eines neuen Passagiers. Ein neuer Passagier soll sich folgendermaßen verhalten: Ist er unter den glücklichen 2500 Passagieren, so versucht er an Bord zu gehen, ansonsten wartet er auf die nächste Fähre.

```

```

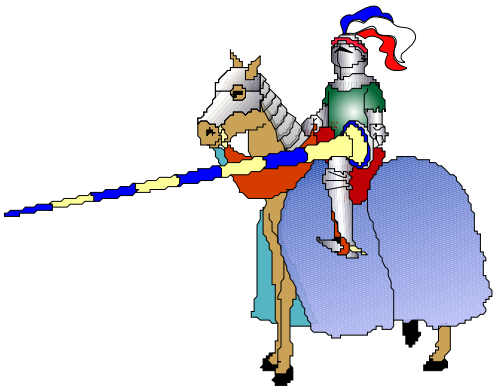
`Passagier_bezieht_Kabine()`

```

```

2 Synchronisation (25)

2.1 Semaphore (20)



Wir schreiben das Jahr 867. Ritter Kunibert ist ein angesehener und gefürchteter Turnierritter. Zum Leidwesen seiner zwei Knechte, ist Ritter Kunibert dem Vergnügen vor dem Turnier nicht abgeneigt. Und so kommt es wie es kommen muss, Ritter Kunibert ist am Morgen des Turniers noch völlig erschöpft von der Turnierparty und benötigt die Hilfe seiner Diener um sich passend einzukleiden. Da nicht mehr viel Zeit bis zu Kuniberts Auftritt bleibt ist es wichtig, dass

- Ritter Kunibert in der richtigen Reihenfolge eingekleidet und für das Turnier vorbereitet wird
- und dies unter Ausnutzung maximaler Parallelität geschieht.

Die folgende Tabelle listet die dafür notwendigen Schritte auf:

ID	Aktion	Funktion	benötigt	Akteur
A	Aufstehen	`stand_up()`	-	Kunibert
B	Stillhalten	`hold_still()`	A	Kunibert
C	Hose anziehen	`pants()`	B	Ruprecht
D	Kettenhemd anziehen	`shirt()`	B	Vladimir
E	Schuh links anziehen und zubinden	`shoe(left)`	C	Ruprecht
F	Schuh rechts anziehen und zubinden	`shoe(right)`	C	Vladimir
G	Beinschoner links befestigen	`guard(shin, left)`	E	Ruprecht
H	Beinschoner rechts befestigen	`guard(shin, right)`	F	Vladimir
I	Armprotektor links befestigen	`guard(arm, left)`	D	Ruprecht
J	Armprotektor rechts befestigen	`guard(arm, right)`	D	Vladimir
K	Brustpanzer fixieren	`protector(front)`	G,H,I,J	Ruprecht
L	Rückenpanzer fixieren	`protector(back)`	K	Ruprecht
M	Helm aufsetzen	`helmet()`	G,H,I,J	Vladimir
N	aufs Pferd setzen	`get_on_horse()`	L, M	Kunibert
O	Schild in die Hand geben	`shield()`	N	Ruprecht
P	Lanze ausrichten	`lance()`	N	Vladimir
Q	Kämpfen	`fight_for_honor()`	O,P	Kunibert

Das Feld `ID` enthält eine Kurzbezeichnung für eine `Funktion`, welche im Feld `Aktion` beschrieben ist. Um eine solche `Funktion` ausführen zu können, müssen davor eine oder mehrere andere Funktionen fertiggestellt worden sein: die IDs dieser Funktionen sind im Feld `benötigt` angegeben. Das Feld `Akteur` gibt die ausführende Person an.

Die Synchronisation ist mittels einer minimalen Anzahl von Semaphoren zu realisieren. Die Verwendung von globalen Variablen und busy waiting ist verboten.

Initialisierung

Benutzen Sie zur Initialisierung der Semaphore die Funktion `init_sem(sem,value)`, wobei `sem` der Name des Semaphores ist und `value` der entsprechende Initialisierungswert.

```
init() {
```

```
}
```

Ritter Kunibert

```
Kunibert() {
```

```
}
```

Knecht Ruprecht

```
Ruprecht() {
```

```
}
```

Knecht Vladimir

```
Vladimir() {
```



```
}
```

2.2 The Producer/Consumer Problem (5)

Welche Grundthematik behandelt das *Producer/Consumer* Problem:

VO 182.022 04. November 2004

Prüfung Betriebssysteme

KNr. | MNr. | Zuname, Vorname

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

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Synchronization (25)

In der Stadt *Peja* gibt es eine Bier-Fabrik. In der Fabrik will man den Bierabfüllungsprozess automatisieren, in dem man 4 Maschinen gekauft hat, die folgende Aufgaben (Funktionen) erfüllen:

Maschine 1: Bierflasche in die Produktionslinie platzieren - `flasche_platzieren()`
Bierflasche von der Produktionslinie entfernen - `flasche_entfernen()`

Maschine 2: Bierflasche mit Bier abfüllen - `flasche_abfüllen()`

Maschine 3: Bierflasche zumachen (mit Kronenkorken) - `flasche_zumachen()`

Maschine 4: Klebt Etikette an die Bierflasche - `etikette_kleben()`

Aufgrund der Ressourcenbeschränkungen kann jeweils nur 1 Flasche in die Produktionslinie platziert werden. Synchronisieren Sie die Arbeit der Maschinen effizient mittels Semaphore! Zusätzlich zu den Operatoren `P()` und `V()` stehen folgende Funktionen zur Verfügung:

- `INIT(sem, value)`: erzeugt und initialisiert den Semaphor `sem` mit dem Wert `value`
- `finshed()`: evaluiert zu `TRUE` wenn die Tagesschicht beendet wird, oder aus anderen Gründen der Abfüllungsprozess gestoppt werden muss. Wenn das der Fall ist `(finshed()=TRUE)`, dann soll die platzierte Flasche fertig abgefüllt, etikettiert, zugemacht und entfernt werden. Erst dann beenden die Maschinen ihre Arbeit.

Passen Sie auf, dass folgende Reihenfolge eingehalten wird:

`Flasche platzieren` -> `Flasche abfüllen` -> `Flasche zumachen` -> `Flasche entfernen`

Nachdem Flaschen platziert und bevor die Flaschen von der Produktionslinie entfernt werden, können die Etiketten geklebt werden. Achten Sie darauf, dass `flasche_abfüllen()` und `flasche_zumachen()` parallel zu der Funktion `etikette_kleben()` ausgeführt werden können. Um den Abfüllungsprozess zu optimieren, soll keine fixe Reihenfolge der Funktionen angegeben werden.

1.1 Initialisierung

1.2 Maschine1_Flaschen

1.3 Maschine2_Bier

1.4 Maschine3_Kronenkorken

1.5 Maschine4_Etiketten

2 Synchronisation (25)

Die folgenden Codestücke skizzieren eine Lösung des *Reader-Writer Problems*.

Initialisierungen

```
init(x,1); init(y,1); init(z,1);
init(wsem,1); init(rsem,1);
rc := 0; wc := 0;
```

Writer

```
 1   loop
 2       P(y);
 3       wc := wc + 1;
 4       if (wc == 1) then
 5           P(rsem);
 6       V(y);
 7
 8       write(...);
 9
10       P(y);
11       wc := wc - 1;
12       if (wc == 0) then
13           V(rsem);
14       V(y);
15   end loop;
```

Reader

```
21   loop
22       P(z);
23       P(rsem);
24       P(x);
25       rc := rc + 1;
26       if (rc == 1) then
27           P(wsem);
28       V(x);
29       V(rsem);
30       V(z);
31
32       read(...);
33
34       P(x);
35       rc := rc - 1;
36       if (rc == 0) then
37           V(wsem);
38       V(x);
39   end loop;
```

a) (8)

Beschreiben Sie kurz und prägnant für jeden Semaphor, welche Rolle er in der gegebenen Lösung spielt. Verwenden Sie, falls notwendig, die angegebenen Zeilennummern um Sich auf Stellen in den Codestücken zu beziehen.

x:

y:

z:

`rsem:`

`wsem:`

b) (17)

Wenn die maximale Anzahl von gleichzeitig aktiven *Reader*-Prozessen nach oben hin abgeschrankt wird, kann man eine Lösung für das *Reader-Writer Problem* angeben, die nur Semaphore (und keine globalen Variablen) zur Synchronisation verwendet. Schreiben Sie eine solche Lösung unter der Annahme, dass maximal K Reader gleichzeitig laufen. Sie brauchen in Ihrer Lösung nicht auf die Priorität von Lesern oder Schreibern zu achten.

Initialisierungen

Reader

Writer

VO 182.022 28. April 2005

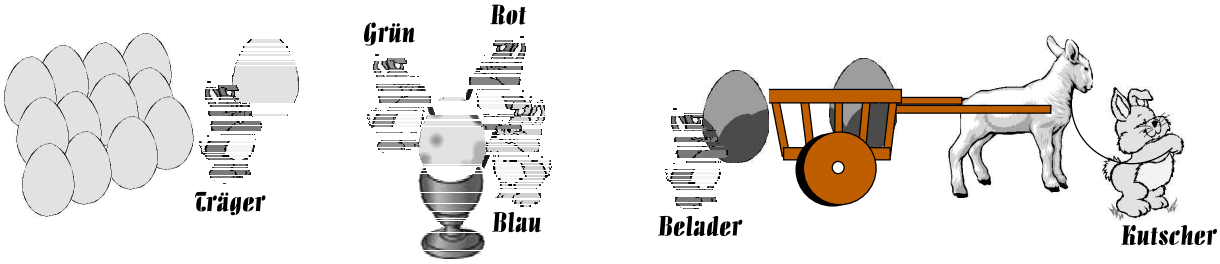
Prüfung Betriebssysteme

KNr. MNr. Zuname, Vorname

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

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Synchronisation (30)



Immer wenn Ostern vor der Tür steht, herrscht bei den Osterhasen Hochkonjunktur. Eine Abteilung von sechs Osterhasen mit Namen *Träger*, *Rot*, *Grün*, *Blau*, *Belader* und *Kutscher* ist mit der Herstellung und dem Versenden von bemalten Ostereiern beauftragt.

Die Hasen sollen dabei nach folgendem Schema arbeiten:

- *Träger* holt jeweils ein frisches Ei aus einem unbegrenzten Vorrat und platziert dieses im Eierbecher, wenn dieser leer ist.
- *Rot*, *Grün* und *Blau* bemalen die Eier im Eierbecher in den entsprechenden Farben. Das Bemalen eines Eis kann prinzipiell gleichzeitig durchgeführt werden, aus ästhetischen Gründen darf jedes Ei aber nur jeweils **zweifarbig** bemalt werden. Es sollen dabei aber alle Farbkombinationen möglich sein. Welche beiden Farben dabei verwendet werden, darf zufällig sein.
- Wenn das Ei fertig bemalt ist, signalisieren *Rot*, *Grün* und *Blau* ihrem Kollegen *Belader*, dass das Ei abzuholen ist.
- *Belader* transportiert das bemalte Ei zum Wagen und belädt diesen.
- Ist der Wagen mit zwei Eiern voll, so signalisiert *Belader* dem *Kutscher* eine Lieferung zu machen.

- *Kutscher* liefert die Eier dann beim Kunden ab. Solange er unterwegs ist, dürfen keine neuen Eier auf den Wagen gelegt werden.

Synchronisieren Sie den Arbeitsablauf der sechs Hasen mittels **Semaphoren**. Achten Sie auf maximale Parallelität. 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 `Vorrat`, `Eierbecher` oder `Wagen` angegeben werden.

`nimmEi()` Lässt den Hasen ein Ei aufnehmen. Funktioniert nur, wenn der Hase neben dem Vorrat bzw. neben dem Eierbecher steht.

`bemaleEi()` Diese Funktion lässt einen Hasen das im Eierbecher befindliche Ei verzieren. Diese Funktion blockiert solange bis der Hase mit dem Bemalen in seiner Farbe fertig ist.

`legeEi()` Der Hase legt das Ei dort ab, wo er gerade steht. Befindet er sich vor dem Wagen, so belädt er diesen. Befindet sich der Hase vor dem Eierbecher so legt er das Ei in diesem ab. Diese Funktion beachtet nicht, für das Ei ausreichend Platz zur Verfügung steht!

`lieferung()` Mit dieser Funktion liefert *Kutscher* die Eier beim Kunden ab. Diese Funktion blockiert solange, bis der Wagen wieder leer an seinem Platz steht.

a) Initialisierungen (8)

Initialisieren Sie die notwenigen Semaphore. Der **Anfangszustand** des Systems entspricht obigem Bild, d.h. *Träger* steht mit einem Ei beim Vorrat, *Rot, Grün* und *Blau* schicken sich gerade an, ein Ei zu bemalen und *Belader* steht mit einem Ei vor dem Wagen, auf dem sich bereits ein weiteres Ei befindet.

b) (10)

Entwerfen Sie die Prozesse, die in den Hasen *Träger* sowie *Rot, Grün* und *Blau* ablaufen:

Prozess *Träger*:

```
do forever() {

}
```

Prozess *Rot*|*Grün*|*Blau*:

```
do forever() {

}
```

c) (12)

Entwerfen Sie die Prozesse, die in den Hasen *Belader* und *Kutscher* ablaufen:

Prozess *Belader*:

```
do forever() {

}
```

Prozess *Kutscher*:

```
do forever() {

}
```

VO 182.022 20. Oktober 2005

Prüfung Betriebssysteme

KNr. MNr. Zuname, Vorname

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

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Synchronisation (25)

In einem Computersystem, das zur Vereinfachung nur ein relevantes Register besitzt, soll eine Folge von Instruktionen mit möglichst hoher Parallelität ausgeführt werden. Es gibt zwei Arten von Operationen auf dem Register: Operationen, die das Register lesen und dann eine Funktion auf dem Registerwert ausführen (kurz Leseoperationen genannt), und Operationen, die einen neuen Wert in das Register schreiben (Schreiboperationen).

Ein Programm für das Computersystem besteht aus einer (unendlichen) Folge von Lese- bzw. Schreiboperationen, die vom Computersystem unter folgenden Einschränkungen abgearbeitet werden.

- Verschiedene Leseoperationen des Programms (vom Typ `READ_INSTRUCTION`) können in beliebiger Reihenfolge und insbesondere auch gleichzeitig abgearbeitet werden.
- Eine Schreiboperation (Typ `WRITE_INSTRUCTION`) und eine beliebige andere Instruktion (Lese- oder Schreiboperation) müssen immer in der Reihenfolge abgearbeitet werden, in der sie im Programmcode stehen (d.h., steht eine Schreiboperation S1 im Code an irgendeiner Stelle vor einer anderen Operation O2, so muss S1 auch vor O2 abgearbeitet werden; genauso muss jede Operation O1, die im Code irgendwo vor einer Schreiboperation S2 steht, vor S2 abgearbeitet werden).

a) Ergänzung von Synchronisationskonstrukten

Das Computersystem arbeitet das Programm unter höchst möglicher Parallelität ab, indem es für jede Instruktion einen eigenen Prozess erzeugt. Dieser Prozess liest die Instruktion ein, bestimmt, ob es sich um eine Lese- oder Schreiboperation handelt, und führt die Operation schliesslich unter Beachtung der oben genannten Einschränkungen mit Hilfe der Funktionen `execute_read_instruction()` bzw. `execute_write_instruction()` aus.

Ergänzen Sie das folgende Stück Pseudocode für die Prozesse zur Instruktionsausführung so mit geeigneten Synchronisationskonstrukten, dass die korrekte Instruktionsabarbeitung laut der oben angegebenen Regeln gesichert ist. Geben Sie geeignete Initialisierungen der Synchronisationskonstrukte an.

Code fuer Prozess zur Instruktionsausfuehrung:

```
    instruktion = get_next_instruction_from_file()

    if (instruction_type(instruktion) == READ_INSTRUCTION)
    {   Behandlung fuer Leseinstruktion

        execute_read_instruction(instruktion)

    } else {    Behandlung fuer Schreibinstruktion (WRITE_INSTRUCTION)

        execute_write_instruction(instruktion)
```

```
}
```

Initialisierungen:

```
```

2 Synchronization (25)

Synchronisieren Sie die Tätigkeiten der Mechaniker während eines Boxenstopps in der Formel 1. Ein Boxenstopp läuft folgendermaßen ab:

- Der Pilot fährt in die Boxengasse (`pit_stop()`). Ein Teammitglied (Signalgeber) gibt via der Signaltafel welche auf 'Brake on' gedreht ist, die Anweisung das Fahrzeug in der Box des Teams zum Stillstand zu bringen (`brake()`). Der Fahrer darf erst wieder Gas geben (`accelerate()`), wenn die Mechaniker fertig sind und die Signaltafel auf 'GO' gestellt ist. Damit ist der Boxenstopp beendet.

- Ein Teammitglied (Signalgeber) signalisiert dem Fahrer mit der Tafel 'Brake on/GO' (`signal_brake()` und `signal_go()`), ab wann das Fahrzeug wieder bereit ist, die Box zu verlassen. Der Signalgeber dreht das Signal auf 'GO', sobald das alle vier Reifen wieder Bodenkontakt haben.

- Um einen Reifenwechsel zu ermöglichen, muss das Auto angehoben werden. Diese Aufgabe übernehmen zwei Teammitglieder, sobald das Auto still steht und der Signalgeber dies mittels 'Break on' anzeigt. Einer bockt das Auto vorne (`lift_front()`) und einer hinten (`lift_back()`) auf. Erst dann können die Reifenmechaniker, die Tankwarte und der Visierreiniger mit ihrer Arbeit beginnen. Sobald diese fertig sind und die Hand gehoben haben, kann das Fahrzeug wieder abgesenkt werden (`down_front()` und `down_back()`) damit alle vier Reifen wieder Bodenkontakt haben.

- Insgesamt 12 Teammitglieder stehen bereit zum Reifenwechsel, je drei davon für ein Rad. Ein Mechaniker löst (`unscrew_wheel_nut()`) und befestigt die Radmutter (`fasten_wheel_nut()`) und hebt den Arm sobald er das Rad fertig gewechselt ist (`ready()`). Ein andere Mechaniker zieht das abgefahrene Rad ab (`remove_old_wheel()`) und der dritte steckt das neue Rad auf (`put_new_wheel()`).

- Zwei Teammitglieder sind für das Betanken des Wagens zuständig (`fill_up()`). Sobald das Fahrzeug fertig betankt ist, wird dies durch das Heben einer Hand signalisiert (`ready()`). Damit kann das Fahrzeug aus Sicht der Tankwarte wieder abgesenkt werden.

- Ein Teammitglied (Visierreiniger) säubert das Helmvisier des Piloten (`clean_visor()`). Sobald das Helmvisier gereinigt ist, wird dies durch das Heben einer Hand signalisiert (`ready()`). Damit kann das Fahrzeug aus Sicht des Visierreinigers wieder abgesenkt werden.

Ergänzen Sie den folgenden Code entsprechend. Beachten Sie dabei folgende Punkte

- Die Synchronisation hat durch Semaphore zu erfolgen, welche in der Funktion `Init()` zu initialisieren sind. Dafür steht die Funktion `initsem(semaphor,value)` zur Verfügung. Zur Synchronisation sind `P(semaphor)` und `V(semaphor)` zu verwenden. Verwenden Sie eine minimale Anzahl von Ressourcen!
- Achten Sie auf maximale Parallelität, um den Boxenstopp so schnell wie möglich zu beenden!

Initialisierung:

```
Init() {

    initsem(signal, 0);
```

```
}
```

Der Fahrer:

```
Driver() {

    pit_stop();

    break();

    V(signal);
```

```
}
```

Mechaniker für den Reifenwechsel:

Obwohl diese Funktion viermal aufgerufen wird (für jedes Rad), braucht diese Funktion nur einmal programmiert werden. Diese Funktion implementiert die Tätigkeit von je drei Mechanikern pro Rad.

```
Mechanic_Wheel() {
```



```
}
```

Mechaniker zum Aufbocken des Fahrzeugs:

Diese Funktion implementiert die Tätigkeit der zwei Mechaniker, welche das Auto aufbocken (vorne und hinten).

```
Mechanic_Lifter() {
```



```
}
```

Der Visierreiniger:

```
Visor() {
```



```
}
```

Betanken des Wagens:

```
Fuel() {
```



```
}
```

Der Signalgeber:

```
Signal() {
    signal_break(); //wait for standing car
    P(signal);
```



```
}
```

VO 182.022
Prüfung Betriebssysteme

27. April 2006

KNr. | MNr. | Zuname, Vorname

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

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Synchronisation (30)

Beim Kartenspiel "Jag the Gitter" erhält jeder *Mitspieler* vom Kartengeber *5 Karten* verdeckt vor sich auf den Tisch gelegt (Funktion `karte_geben(`*Spieler*`)`). Nachdem der Kartengeber (welcher selbst **nicht** am Spiel teilnimmt) die Karten **reihum einzeln** ausgeteilt hat, nehmen alle Spieler **gleichzeitig** ihre Karten auf (Funktion `karten_nehmen()`).

Nachdem alle Mitspieler ihre Karten aufgenommen haben, wird rundenweise jeweils eine Karte pro Mitspieler (Funktion `karte_spielen()`) gespielt. Dabei beginnt jeweils Spieler 1 mit dem Ausspielen, dann kommt Spieler 2 dran usw. Nachdem alle Karten ausgespielt wurden ist das Spiel beendet und der Kartengeber teilt erneut Karten für die nächste Runde aus.

Zu verwendende Funktionen (alle Funktionen blockieren so lange, bis der angegebene Task erledigt ist):

`karte_geben(`*Spieler*`)` teilt eine Karte an den angegebenen *Spieler* ($\in \{1, 2, 3, ...\}$) aus. Diese Funktion wird nur vom Kartengeber verwendet.

`karten_nehmen()` lässt den Spieler alle seine Karten aufnehmen. Diese Funktion darf erst aufgerufen werden, wenn alle Karten dieses Spielers ausgeteilt sind.

`karte_spielen()` lässt den Spieler eine seiner Karten ausspielen.

`initS(`*semname*`,`*n*`)`, `initE(`*evcname*`,`*n*`)` zum Anlegen und Initialisieren von Semaphoren und Eventcountern

`P(),V(),advance(),wait()` mit der üblichen Semantik für Semaphore und Eventcounter.

Weiters zu beachten:

- Die Verwendung von globalen Variablen zur Synchronisation ist verboten!
- Achten Sie auf maximale Parallelität!
- Verwenden Sie (unter Berücksichtigung der oberen beiden Punkte) nur so viele Ressourcen wie notwendig!

a) (14)

Zuerst soll das Kartenspiel für *2 Spieler* implementiert werden. Synchronisieren Sie die folgenden Prozesse (Kartengeber, Spieler 1 und Spieler 2) mit **Semaphoren**, sodass Sie das Spiel "Jag the Gitter" wie oben beschrieben spielen.

Initialisierungen		
Kartengeber	**Spieler 1**	**Spieler 2**
FOREVER()	FOREVER()	FOREVER()
{	{	{
}	}	}

c) (16)

Nun soll das Kartenspiel auf eine beliebige Anzahl N von Spielern erweitert werden (N ist konstant). Ergänzen Sie im folgenden Gerüst die Aufrufe von `await()`, um die Prozesse (Kartengeber, Spieler i) mit **einem Eventcounter** derartig zu synchronisieren, sodaß Sie das Spiel "Jag the Gitter" wie oben beschrieben spielen. Im Spielerprozess steht ihnen die entsprechende Nummer des Spielers (zwischen 1 und N) in der Variablen i zur Verfügung.

Tipp: Sie brauchen zur Lösung dieses Beispiel *keine* Sequencer!

Initialisierungen

Kartengeber	**Spieler** i
`FOREVER()`	`FOREVER()`
`{`	`{`
`}`	`}`

VO 182.022 17. November 2006

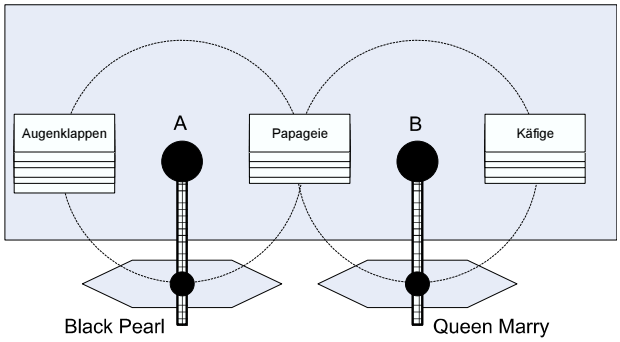
Prüfung Betriebssysteme

KNr. MNr. Zuname, Vorname

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

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Synchronisation (30)



Die Reederei *Titanic Transports* besitzt zwei Frachtschiffe namens *Queen Marry* und *Black Pearl*. Beide Frachtschiffe haben einen eigenen Liegeplatz mit dazugehörigem Verladekran. Die Queen Marry liefert regelmäßig *Papageie* und dazugehörige *Käfige* an Tierliebhaber in ein weit entferntes Land. Die Black Pearl liefert an eine von Piraten bewohnte Insel *Papageie* und *Augenklappen* (Piraten tragen ihren Papagei auf der Schulter und brauchen daher keine Käfige).

Die Papageie, Käfige und Augenklappen sind in Containern verpackt, welche auf dem Hafen gestapelt stehen.

Das Beladen der Schiffe läuft nach folgenden Regeln ab:

- Die Queen Marry soll mit einem Container voller Papageie und einem Container voller Käfige beladen werden. Die Black Perl soll mit einem Container voller Augenklappen und mit einem Container voller Papageie beladen werden.
- Zum Beladen werden die Kräne verwendet.
- Der Kran *A*, welcher der Black Pearl zugeordnet ist kann über die Container mit den Papageien, über die Container mit den Augenklappen und über die Black Pearl bewegt werden.
- Der Kran *B*, welcher der Queen Marry zugeordnet ist, kann über die Container mit den Papageien, über die Container mit den Käfigen und über die Queen Marry bewegt werden.

- Um ein Zusammenstoßen der Kräne zu verhindern dürfen niemals beide Kräne gleichzeitig über die Container mit den Papageien bewegt werden.
- Wenn ein Schiff fertig beladen ist, soll es sofort mit seiner Auslieferung beginnen.
- Mit dem Beladen eines Schiffes kann natürlich erst dann wieder begonnen werden wenn das Schiff zurückgekehrt ist. Ein Kran kann allerdings einen Container schon vom Stapel nehmen bevor das Schiff wieder zurückgekehrt ist.

Synchronisieren Sie den Arbeitsablauf der beiden Kräne und der beiden Schiffe mittels **Semaphoren**. Achten Sie auf maximale Parallelität. 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.

`bewege(` ***Richtung*** `)` Bewegt den Kran um 90 Grad in die gewünschte *Richtung*. Als Richtung kann `Uhrzeigersinn (UZS)` oder `Gegen_Uhrzeigersinn (GUZS)` angegeben werden.

`nimm()` Lässt den Kran einen Container von dem Stapel über dem er sich gerade befindet aufnehmen. Funktioniert nur, wenn sich der Kran über einem Containerstapel befindet.

`lege_ab()` Mittels dieser Funktion legt ein Kran den Container, den er gerade hält, an der aktuellen Position ab

`lieferung()` Mit dieser Funktion liefert ein Schiff seine Ware aus, und kehrt anschliessend wieder zurück.

a) Initialisierungen (8)

Initialisieren Sie die notwendigen Semaphore. Der **Anfangszustand** des Systems entspricht dem obigem Bild. Beide Schiffe sind unbeladen und beide Kräne stehen über den jeweiligen Schiffen.

b) (14)

Entwerfen Sie je einen Prozess für *Kran A* und *Kran B*.

Prozess *Kran A*:

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

Prozess *Kran B*:

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

c) (8)

Entwerfen Sie die Prozesse für die *Black Pearl* und die *Queen Marry*:

Prozess ***Black Pearl***:

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

Prozess ***Queen Marry***:

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

VO 182.022 17. November 2006

Prüfung Betriebssysteme

KNr. MNr. Zuname, Vorname

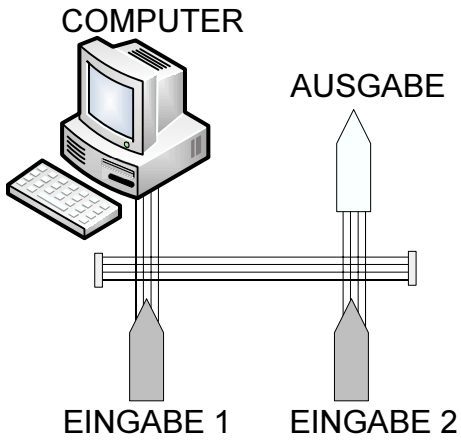
Ges.)(100)	1.)(30)	2.)(25)	3.)(23)	4.)(22)	Zusatzblätter:

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Synchronisation (25)

Ein Computerprogramm simuliert das Verhalten eines Systems, das aus den folgenden Einheiten besteht:

- ein Computer
- zwei Eingabegeräte (Input),
- ein Ausgabegerät (Output), und
- ein Bus-Netzwerk, das den Computer mit den Eingabe- und Ausgabegeräten verbindet.



Das Programm simuliert die Funktionalität des Computersystems mittels verschiedener Prozesse, wobei ein Prozess das Verhalten des Computers, ein Prozess das Verhalten eines Eingabegerätes und ein Prozess das Verhalten des Ausgabegerätes simuliert. Diese Prozesse sollen folgende Funktionalität implementieren:

- Der Eingabeprozess liest die Daten mittels Funktion *read_input_data()* aus dem Eingabegerät und schickt die Daten mittels Funktion *send_input_data()* zum Computer sobald das Netwerk frei von Nachrichten ist.
- Der Computerprozess liest die emfangenen Daten mittels Funktion *PC_read_input_data()*.

- Der Computer bearbeitet die gelesenen Daten mittels Funktion *process_data()* und sendet die abgearbeiteten Daten durch das Netzwerk mittels Funktion *PC_send_output_data()* zum Ausgabeprozess.
- Der Ausgabeprozess liest die empfangenen Daten mittels Funktion *rcv_output_data()* und verwendet sie mittels Funktion *apply_output_data()* am Ausgabegerät.
- Der Computer und der Ausgabeprozess können die Daten der Eingabeprozesse hintereinander bearbeiten.
- Der Computerprozess und die Eingabeprozesse dürfen das Netzwerk zum Senden der Daten nicht gleichzeitig benutzen.
- Die zwei Eingabegeprozesse sind identisch und arbeiten unabhängig voneinander.

Passen Sie auf, dass folgende Reihenfolge eingehalten wird:

read_input_data -> send_input_data ->

PC_read_input_data -> process_data -> PC_send_output_data ->

rcv_output_data -> apply_output_data

Synchronisieren Sie den Arbeitsablauf der Prozesse mit Semaphoren. Achten Sie auf maximale Parallelität. Verwenden Sie möglichst wenige Synchronisationskonstrukte. Die Verwendung von globalen Variablen ist verboten.

Verwenden Sie folgende Funktionen für Operationen auf Semaphoren:

`initS(`***Sem,init***`)` legt einen Semaphor mit dem angegebenen Namen *Sem* an und initialisiert ihn mit der Zahl *init*.

P(*Sem*) implementiert ein *wait* auf dem Samaphore, und

V(*Sem*) implementiert ein *signal* auf dem Samaphore.

a) Initialisierungen

Initialisieren Sie die notwendigen Semaphore. Der **Anfangszustand** ist wie folgt:

- keine Daten im Eigabengerät,
- keine Daten im Ausgabegerät,
- das Netzwerk und ist frei von Nachrichten, und
- der Computer hat keine Daten zu bearbeiten.

Entwerfen Sie je einen Prozess für *Eingabegerät 1* und *2*

Prozess *Eingabegerät 1*:

```
do forever() {

}
```

Prozess *Eingabegerät 2*:

```
do forever() {

}
```

Entwerfen Sie die Prozesse für den *Computer* und das *Ausgabegerät*

Prozess *Computer*:

```
do forever() {

}
```

Prozess *Ausgabegerät*:

```
do forever() {

}
```

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
```

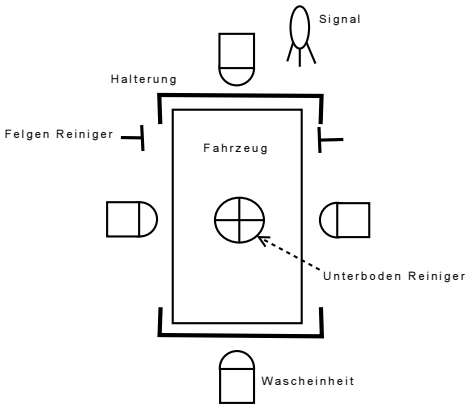
Code für Task *T4*:

```
loop forever
```



```
end loop
```

2 Synchronization (31)



Synchronisieren Sie die Tätigkeiten der Komponenten einer Waschanlage während einer Autowäsche in der Waschbox. Ein Waschvorgang läuft folgendermaßen ab:

- Der Fahrer fährt in die Waschbox (`enter_box()`). Ein Signalgeber gibt mittels eines Lichtsignals, welches auf 'Rot/Red' steht, die Anweisung das Fahrzeug in der Box zum Stillstand zu bringen (`brake()`). Der Fahrer darf erst wieder Gas geben (`accelerate()`), wenn die Waschеinheiten fertig sind und das Lichtsignal auf 'Grün/Green' gestellt ist. Damit ist der Waschvorgang beendet.

- Das Lichtsignal signalisiert dem Fahrer mit der Anzeige 'Rot/Red' bzw. 'Grün/Green' (`signal_red()` und `signal_green()`), ab wann das Fahrzeug wieder bereit ist, die Waschbox zu verlassen. Das Lichtsignal stellt sich auf 'Grün/Green' um, sobald der Waschvorgang beendet ist und das Fahrzeug nicht mehr fixiert ist.

- Um den Waschvorgang zu ermöglichen, muss das Auto in der Waschbox fixiert werden. Diese Aufgabe übernehmen zwei Halterungen, sobald das Auto still steht und das Lichtsignal dies mittels 'Rot/Red' anzeigt. Eine Halterung fixiert das Auto vorne (`lock_front()`) und eine weitere hinten (`lock_back()`). Erst dann können die Waschеinheiten, die Felgenreiniger und die Unterbodenwascheinheit mit dem Waschvorgang beginnen. Sobald diese fertig sind und dies melden, kann das Fahrzeug wieder von der Haltevorrichtung befreit werden (`unlock_front()` und `unlock_back()`) damit es die Waschbox verlassen kann.

- Insgesamt stehen 4 Waschеinheiten bereit für die Fahrzeugreinigung, je eine für jede Seite (vorne, hinten, rechts, links). Jede Waschеinheit besteht aus 4 Funktionen: Eine Düse sprüht Wasser (`sprinkle_water()`) bzw. verteilt Luft nach dem Waschvorgang (`sprinkle_air()`) und gibt ein Signal sobald der Waschvorgang beendet ist (`ready()`). Eine andere Düse verteilt Waschmittel (`sprinkle_detergent()`). Nach dem Versprühen von Wasser und dem Einbringen von Waschmittel werden die Waschbürsten bewegt (`activate_brushes()`).

- Zwei Waschеinheiten sind für die Reinigung der Felgen (engl.: Rim) zuständig (`clean_rim()`). Sobald die Felgen gereinigt sind, wird das mittels eines Signals gemel-

det (`ready()`). Damit kann das Fahrzeug seitens der Felgenreinigungseinheit wieder aus der Halterung gelöst werden.

- Eine Unterbodenreinigungseinheit säubert den Unterboden des Fahrzeugs (`clean_underbody()`). Sobald der Unterboden gereinigt ist, wird ein Signal gegeben (`ready()`). Damit kann das Fahrzeug von Seiten der Unterbodenreinigungseinheit freigegeben werden.

Ergänzen Sie den folgenden Code entsprechend. Beachten Sie dabei folgende Punkte

- Die Synchronisation hat durch Semaphore zu erfolgen, welche in der Funktion `Init()` zu initialisieren sind. Dafür steht die Funktion `initsem(semaphor,value)` zur Verfügung. Zur Synchronisation sind `P(semaphor)` und `V(semaphor)` zu verwenden. Verwenden Sie eine minimale Anzahl von Ressourcen!
- Achten Sie auf maximale Parallelität, um den Boxenstopp so schnell wie möglich zu beenden!

Initialisierung:

```
Init() {
```



```
}
```

Der Fahrer:

```
Driver() {
```

```
}
```

Wascheinheiten:

Obwohl diese Funktion viermal aufgerufen wird (für jede Seite), braucht diese Funktion nur einmal programmiert werden. Diese Funktion implementiert die Tätigkeit von je drei Wascheinheiten pro Fahrzeugseite.

```
Cleaning_Units() {
```

```
}
```

Halterungen zur Fixierung des Fahrzeugs:

Diese Funktion implementiert die Tätigkeit der zwei Halterungen, welche das Auto fixieren (vorne und hinten).

```
Lock_Car() {
```

```
}
```

Die Unterbodenreinigungseinheit:

```
Underbody_Unit() {
```



```
}
```

Reinigung der Felgen:

```
Rim_Unit() {
```



```
}
```

Das Signallicht:

```
Signal() {
```



```
}
```

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
```

VO 182.022 22. Oktober 2008

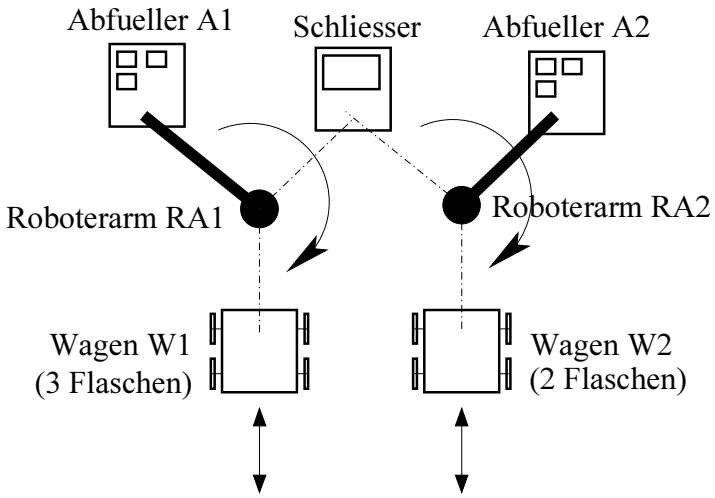
Prüfung Betriebssysteme

KNr. MNr. Zuname, Vorname

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

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Synchronisation (30)



Die Molkerei *Lolatte* besitzt zwei Abfüllanlagen sowie eine gemeinsame Schließanlage für die Flaschen nach dem Befüllen. Je ein Roboterarm nimmt fertig gefüllte Flaschen von der jeweiligen Abfüllanlage und bringt sie zum Schließer, wo die Flasche luftdicht abgeschlossen wird. Anschließend wird die Flasche in einen Wagen gestellt, welcher regelmäßig ausgelehrt wird. Der Wagen "W1" kann drei Flaschen aufnehmen, bevor er geleert werden muss. Der Wagen "W2" kann nur zwei Flaschen aufnehmen, bevor er wieder geleert werden muss. Die Roboterarme haben die Eigenheit, dass sie sich beide **nur im Uhrzeigersinn drehen** lassen, wie auch in obiger Abbildung dargestellt.

Das Beladen der beiden Wagen läuft nach folgenden Regeln ab:

- Jeder Roboterarm kann direkt ohne Zeitverzögerung von seiner Abfüllanlage eine Flasche holen.
- Wenn ein Roboterarm sich gerade zum Schließer hinbewegt oder bereits wieder wegbewegt, wäre die Gefahr gegeben, dass er mit dem zweiten Roboterarm kollidiert. Daher muss dieser Abschnitt entsprechend gesichert werden.
- Der *Schließer* wartet bis ein Roboterarm direkt vor ihm eine Flasche platziert hat und verschließt diese dann.

- Der Drehwinkel für den Roboterarm ist zwischen dem Schließer und der Abfüllanlage bzw. dem Wagen genau 120 Grad.
- Wenn ein Wagen fertig beladen ist, soll er sofort mit seiner Auslieferung beginnen.
- Mit dem Anfüllen eines Wagens kann natürlich erst dann wieder begonnen werden wenn der Wagen zurückgekehrt ist. Der Roboterarm kann allerdings schon die nächste Flache verschließen bevor der Wagen zurueck ist.

Synchronisieren Sie den Arbeitsablauf der beiden Roboterarme, des Schließers und der beiden Wagen mittels **Semaphoren**. Achten Sie auf maximale Parallelität. Verwenden Sie möglichst wenige Synchronisationskonstrukte. Die Verwendung von globalen Variablen ist verboten.

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 den Schließer:

`close()` Mit dieser Funktion wird eine Flasche verschlossen. Funktioniert nur, wenn einer der Roboterarme zum Abfüller ausgerichtet ist und eine Flasche häelt. Die Flasche braucht zum Schließen vom Roboterarm nicht losgelassen werden.

Zu verwendende Funktionen für die Roboterarme:

`turn(` ***Winkel*** `)` Bewegt den Roboterarm im Uhrzeigersinn um *Winkel* Grad weiter (Winkel ist ein Wert im Bereich von 0-360 Grad). Beachten Sie, dass Roboterarm RA2 komplett baugleich wie RA1 ist und sich daher ebenfalls ausschließlich im Uhrzeigersinn dreht!

`get()` Nimmt von der Abfüllanlage eine Flasche. Funktioniert nur, wenn der Roboterarm zur Abfüllanlage ausgerichtet ist.

`put()` Stellt die Flasche in den Wagen. Funktioniert nur, wenn der Roboterarm zum jeweiligen Wagen ausgerichtet ist.

Zu verwendende Funktionen für die Wagen:

`deliver()` Mit dieser Funktion fährt der Wagen in die Vertriebshalle und kommt anschließend wieder leer zurück.

a) Initialisierungen (4)

Initialisieren Sie die notwendigen Semaphore. Der **Anfangszustand** des Systems entspricht dem obigem Bild. Beide Wagen sind unbeladen und beide Roboterarme sind zur jeweiligen Abfüllanlage platziert.

b) (4)

Entwerfen Sie einen Prozess für *Schliesser*.

`Prozess` ***`Schliesser`***`:`

```
do forever() {
```

```
}
```

c) (16)

Entwerfen Sie je einen Prozess für *Roboterarm RA1* und *Roboterarm RA2*.

Prozess *Roboterarm RA1*:

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

Prozess *Roboterarm RA2*:

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

d) (6)

Entwerfen Sie die Prozesse für *Wagen W1* und *Wagen W2*:

Prozess *Wagen W1*:

```
do forever() {

}
```

Prozess *Wagen W2*:

```
do forever() {

}
```

VO 182.022 14. Jänner 2009

Prüfung Betriebssysteme

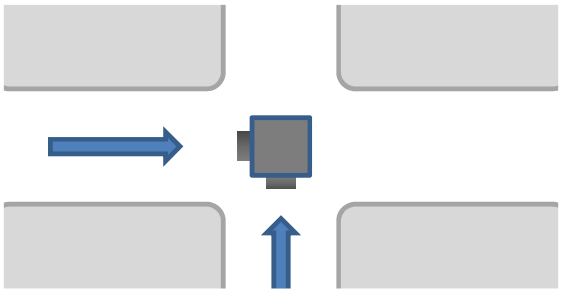
KNr. MNr. Zuname, Vorname

Ges.)(100)	1.)(36)	2.)(21)	3.)(18)	4.)(25)	Zusatzblätter:

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Synchronisation (36)

Der Autoverkehr einer Kreuzung von zwei Einbahnstraßen wird mit einer einfachen Ampel geregelt (siehe Skizze). Die Ampel hat auf jeder der beiden relevanten Seiten eine rote und eine grüne Lampe, um anzuzeigen, dass die Autos anhalten müssen (rot) bzw. passieren dürfen (grün). Die Autos der beiden Straßen erhalten jeweils abwechselnd eine Grünphase von 45 Sekunden. Am Anfang bzw. Ende jeder Grünphase wird ohne Verzögerung zwischen rotem und grünem Licht (bzw. umgekehrt) umgeschaltet.



Schreiben Sie drei Prozess-Templates zur Simulation des Verkehrs an der Ampel. Der Prozess *Ampel* soll die Ampel simulieren. Das Prozess-Template *Auto-W* simuliert ein von Westen und das Template *Auto-S* ein von Süden über die Kreuzung fahrendes Auto.

Die Mutual Exclusion und das korrekte Passieren der Ampel bei grün soll durch Semaphore geregelt werden. Der zyklische Prozess *Ampel* gibt die Kreuzung abwechselnd für die Autos der beiden Straßen frei und wartet zwischen den Umschaltvorgängen mit dem Funktionsaufruf *warte(45)* für 45 Sekunden.

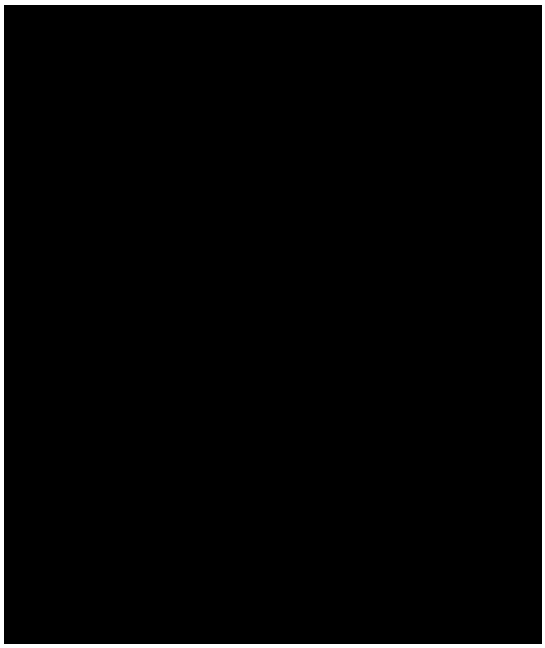
Das Template *Auto-W* simuliert die Fahrt eines von Westen kommenden Autos, das die Kreuzung passiert. Mehrere von Westen kommende Autos werden durch das Starten mehrerer Kopien von *Auto-W* simuliert. Entsprechendes gilt für das Template *Auto-S*, das ein von Süden kommendes Auto simuliert.

Zum Durchfahren der Kreuzung rufen Auto-Prozesse die Funktion *kreuzung_passieren()* auf. Dabei gilt: Ein Auto darf nur in die Kreuzung einfahren, wenn diese frei ist. Die Synchronisation muss sicherstellen, dass ein Auto nur bei grünem Ampelsignal in die Kreuzung einfährt und dass das Umschalten der Ampel nicht durch ankommende Autos verzögert wird. Schaltet die Ampel um, darf das gerade in der Kreuzung befindliche Auto die Kreu-

zung noch verlassen, dann müssen die Autos der anderen Fahrtrichtung in die Kreuzung einfahren können.

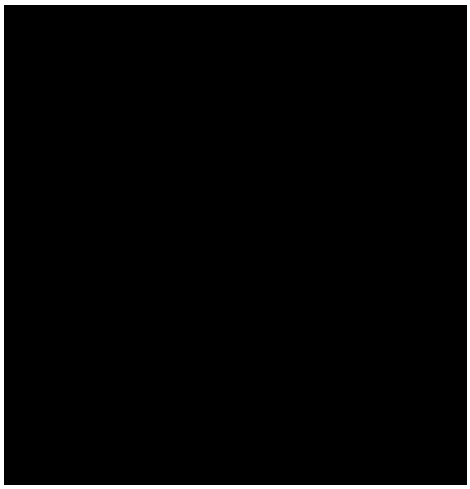
Schreiben Sie Codestücke für *Ampel*, *Auto-W* und *Auto-S* und geben Sie Initialisierungen für die benötigten Semaphore an.

Initialisierungen



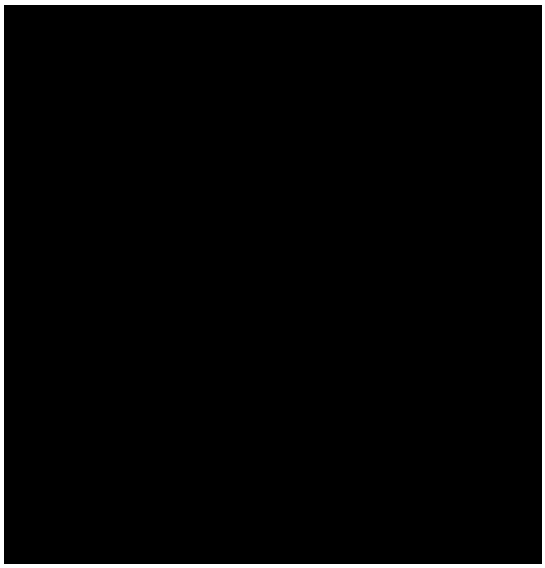
Ampel

```
forever{
```

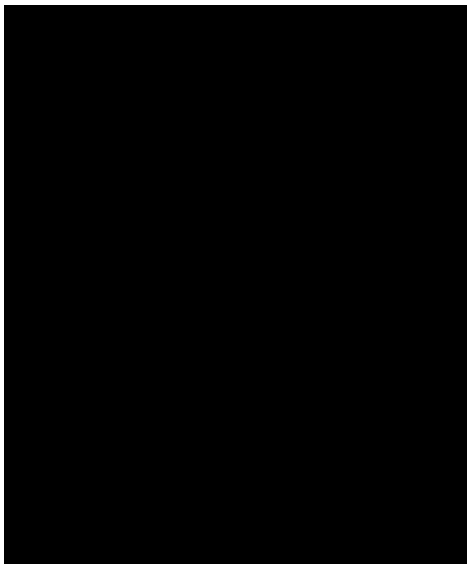


```
}
```

Auto-W



Auto-S



VO 182.022 27. Oktober 2009

Prüfung Betriebssysteme

KNr. MNr. Zuname, Vorname

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

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Synchronisation (25)

Gegeben sei ein System mit einer beliebigen Anzahl von parallel laufenden Prozessen vom Typ A und vom Typ B. Weiters ist in dem System ein Gerät vom Typ G 4-fach vorhanden. Die zu produzierende und konsumierende Ressource vom Typ R ist initial 0-mal vorhanden. Ein Prozess vom Typ A benötigt für die Abarbeitung einer Iteration exklusiven Zugriff auf 3 Geräte vom Typ G und produziert dabei eine Ressource vom Typ R. Ein Prozess vom Typ B benötigt für die Abarbeitung exklusiven Zugriff auf ein Gerät vom Typ G und konsumiert zusätzlich zwei Ressourcen vom Typ R.

Synchronisieren Sie den Arbeitsablauf der Prozesse mit Semaphoren. Achten Sie auf Vermeidung von Deadlocks und ermöglichen sie maximale Parallelität. Verwenden Sie möglichst wenige Synchronisationskonstrukte. Die Verwendung von globalen Variablen ist verboten.

Verwenden Sie folgende Funktionen für Operationen auf Semaphoren:

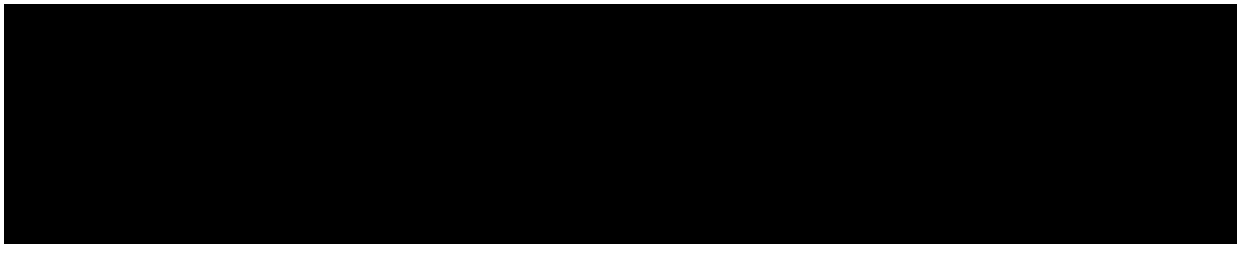
`initS(`***Sem,init***`)` legt einen Semaphor mit dem angegebenen symbolischen Namen *Sem* an und initialisiert ihn mit der Zahl *init*.

P(*Sem*) implementiert ein *wait* auf dem Semaphore, und

V(*Sem*) implementiert ein *signal* auf dem Semaphore.

a) Initialisierungen

Initialisieren Sie die notwendigen Semaphore.



b) Entwerfen Sie die Prozesse vom Typ A und B

Die Prozesse A und B benötigen in der Funktion *do_the_work_A()* bzw. *do_the_work_B()* jeweils exklusiven Zugriff auf die jeweilige Anzahl an Geräten G.

`Prozess Typ A:`

```
do forever() {
[redacted]
do_the_work_A();
[redacted]
}
```

`Prozess Typ B:`

```
do forever() {
[redacted]
do_the_work_B();
[redacted]
}
```

VO 182.022 25. Jänner 2010

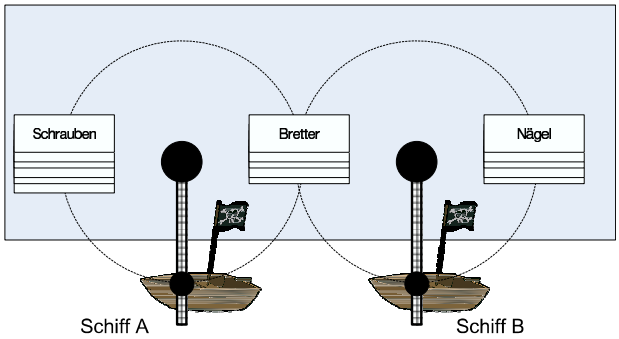
Prüfung Betriebssysteme

KNr. MNr. Zuname, Vorname

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

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Synchronisation (30)



Eine Reederei besitzt zwei Frachtschiffe, *Schiff A* und *Schiff B*. Beide Frachtschiffe haben einen eigenen Liegeplatz mit dazugehörigem Verladekran. Das *Schiff A* transportiert *Schrauben* und *Bretter*, das *Schiff B* transportiert *Bretter* und *Nägel*.

Die Schrauben, Bretter und Nägel sind in Containern verpackt, welche auf dem Hafen gestapelt stehen.

Das Beladen der Schiffe läuft nach folgenden Regeln ab:

- Schiff A soll mit einem Container voller Schrauben und einem Container voller Bretter beladen werden. Schiff B soll mit einem Container voller Bretter und mit einem Container voller Nägel beladen werden. Die Schrauben bzw. die Nägel müssen vor den Brettern geladen werden.
- Zum Beladen werden die Kräne verwendet.
- Der Kran *A*, welcher Schiff A zugeordnet ist kann über die Container mit den Schrauben, über die Container mit den Brettern und über das Schiff A bewegt werden.
- Der Kran *B*, welcher Schiff B zugeordnet ist, kann über die Container mit den Brettern, über die Container mit den Nägeln und über das Schiff B bewegt werden.
- Um ein Zusammenstoßen der Kräne zu verhindern dürfen niemals beide Kräne gleichzeitig über die Container mit den Brettern bewegt werden.
- Wenn ein Schiff fertig beladen ist, soll es sofort mit seiner Auslieferung beginnen.

- Mit dem Beladen eines Schiffes kann natürlich erst dann wieder begonnen werden wenn das Schiff zurückgekehrt ist. Ein Kran kann allerdings einen Container schon vom Stapel nehmen bevor das Schiff wieder zurückgekehrt ist.

Synchronisieren Sie den Arbeitsablauf der beiden Kräne und der beiden Schiffe mittels **Semaphoren**. Achten Sie auf maximale Parallelität. 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.

`bewege(` ***Richtung*** `)` Bewegt den Kran um 90 Grad in die gewünschte *Richtung*. Als Richtung kann `Uhrzeigersinn (UZS)` oder `Gegen_Uhrzeigersinn (GUZS)` angegeben werden.

`nimm()` Lässt den Kran einen Container von dem Stapel über dem er sich gerade befindet aufnehmen. Funktioniert nur, wenn sich der Kran über einem Containerstapel befindet.

`lege_ab()` Mittels dieser Funktion legt ein Kran den Container, den er gerade hält, an der aktuellen Position ab

`lieferung()` Mit dieser Funktion liefert ein Schiff seine Ware aus, und kehrt anschliessend wieder zurück.

a) Initialisierungen (8)

Initialisieren Sie die notwendigen Semaphore. Der **Anfangszustand** des Systems entspricht dem obigem Bild. Beide Schiffe sind unbeladen und beide Kräne stehen über den jeweiligen Schiffen.

b) (14)

Entwerfen Sie je einen Prozess für *Kran A* und *Kran B*.

Prozess *Kran A*:

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

Prozess *Kran B*:

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

c) (8)

Entwerfen Sie die Prozesse für *Schiff A* und *Schiff B*:

Prozess *Schiff_A*:

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

Prozess *Schiff B*:

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

VO 182.022
Prüfung Betriebssysteme

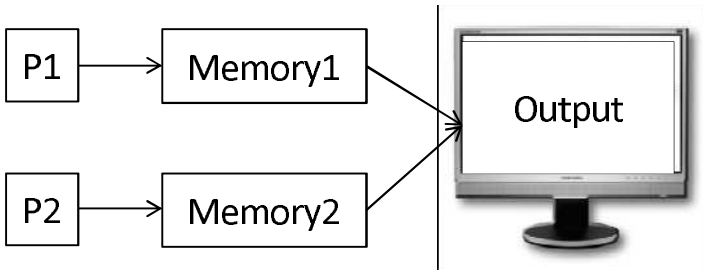
29. April 2009

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 Löseung 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() {
                                                    }
```

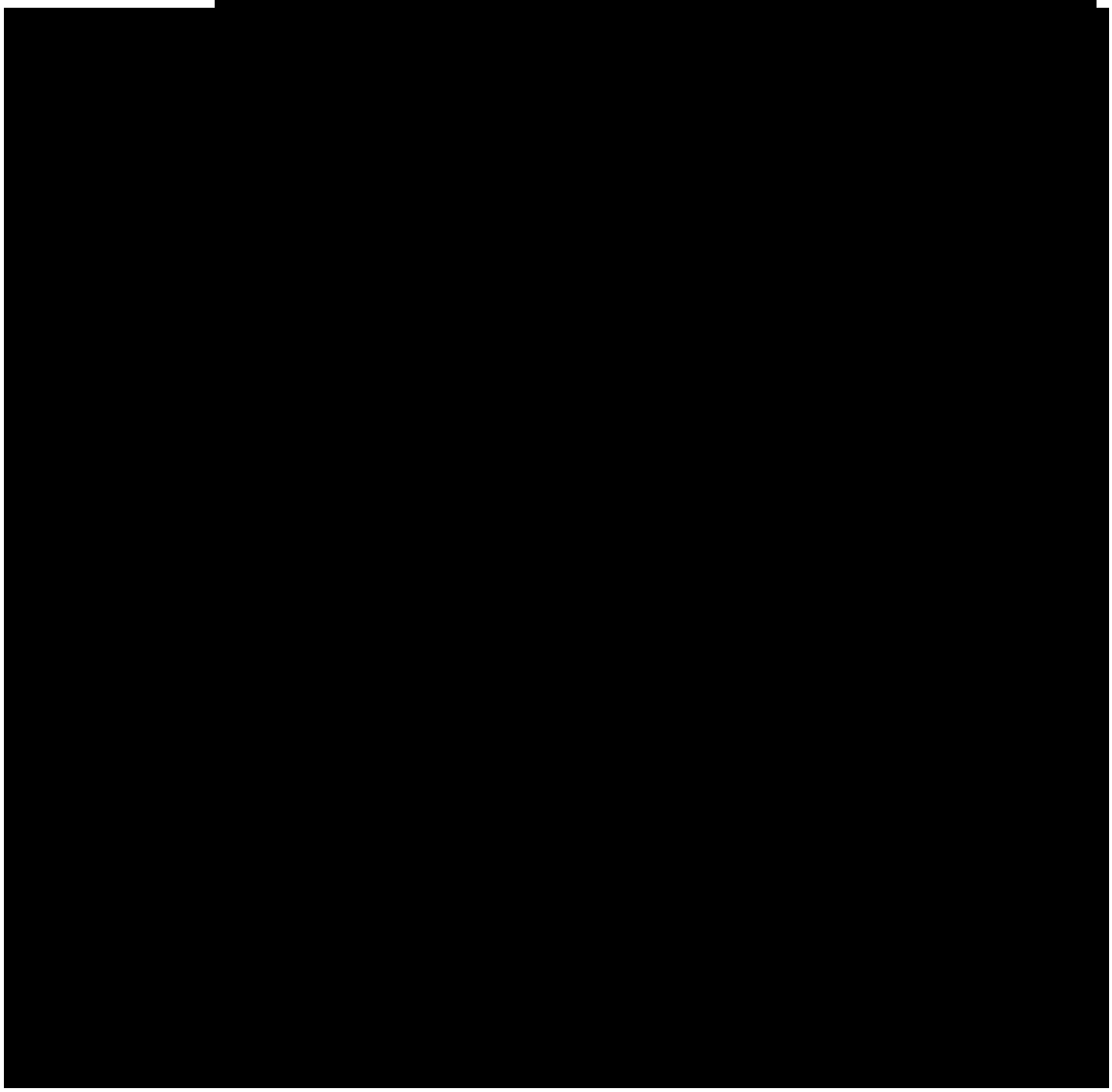
b) Ausgabeprozess (10)

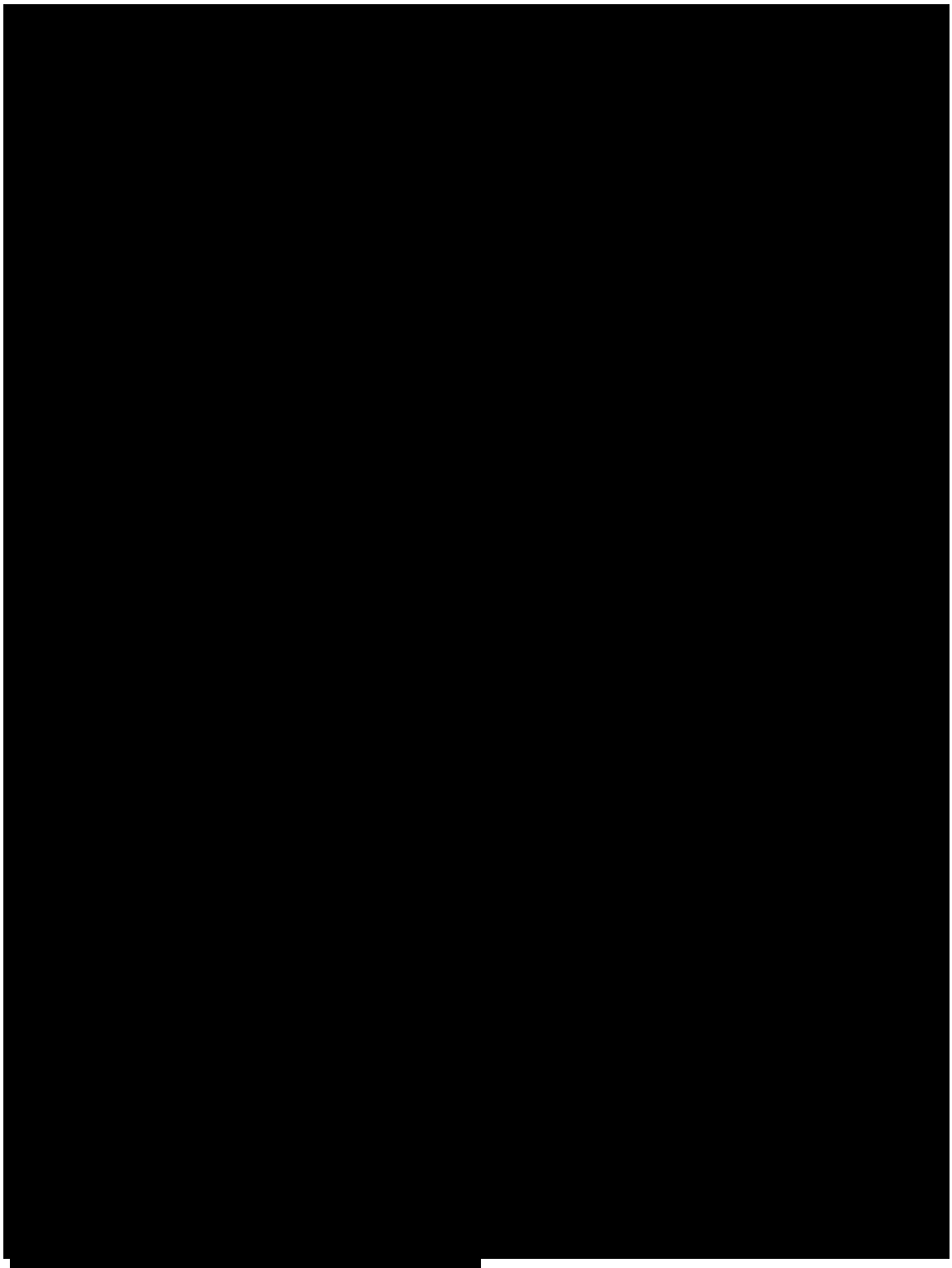
Definition von Hilfsvariablen:

Entwerfen Sie den *Ausgabeprozess*:

`Prozess` ***`Output`***`:`

```
do forever() {
```





VO 182.022 27. Oktober 2010

Prüfung Betriebssysteme

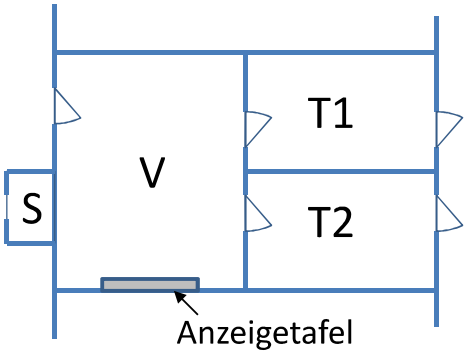
KNr. | MNr. | Zuname, Vorname

Ges.)(100) | 1.)(30) | 2.)(20) | 3.)(50) | Zusatzblätter:

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Synchronisation mit Semphoren (30)

In einem Fitnesscenter kann man seine Fitness testen lassen. Dazu gibt es ein Testzentrum mit folgenden Einrichtungen (siehe Abbildung):



- einen Schalter (S), bei dem man sich ein Anmeldeformular abholt bzw. das ausgefüllte Formulare abgibt und damit auch eine Identifikationsnummer (ID) für den Test bekommt. Beim Schalter darf sich immer nur maximal eine Person aufhalten.
- einen Vorbereitungsraum (V), in dem sich maximal 6 Personen aufhalten dürfen, um sich auf den Test vorzubereiten.
- zwei Testräume (T1 und T2), in denen sich jeweils maximal eine Person aufhalten darf, um den Fitnesstest zu absolvieren.
- Weiters gibt es eine Anzeigetafel, die die IDs der beiden Testteilnehmer anzeigt, die sich als nächstes für das Eintreten in einen der beiden Testräume bereithalten sollen. Diese Reihenfolge kann von der Reihenfolge der Anmeldung beim Schalter S abweichen. Die Aktualiserung der Anzeigetafel erfolgt mit folgendem Codestück:

```
P(Anzeige)
update_Anzeige()
V(Anzeige)
```

Schreiben Sie einen Prozess für einen Sportler *Sp*, der unter Einhaltung der obigen Einschränkungen einen Fitnesstest durchführt. Verwenden Sie *Semaphore* zur Synchronisation und vermeiden Sie unnötige Einschränkungen der Parallelität mehrerer Sportler. *Sp* führt folgende Schritte durch:

- *Sp* holt sich mit der Funktion `get_form()` ein Anmeldeformular vom Schalter, füllt dieses aus (`fill_form()`), gibt das Forumlar ab und bekommt eine ID (Der Aufruf der Funktion `submit_form()` dient zum Abgeben des Formulars und retourniert die ID).

- *Sp* tritt in den Vorbereitungsraum ein (`enter_V()`) und ruft abwechselnd `prepare()` und `check_board()` auf, die die Vorbereitung auf den Test bzw. das Ablesen der Anzeigetafel realisieren. An `check_board()` übergibt der Prozess die ID von *Sp*. Der Returnwert der Funktion gibt an, ob und für welchen Raum die Nummer von *Sp* auf der Anzeigetafel steht: (Wert 0: ID steht nicht auf der Anzeigetafel, Wert 1: *Sp* soll sich für das Eintreten in T1 bereithalten, Wert 2: *Sp* soll sich für das Eintreten in T2 bereithalten).
  Achten Sie darauf, dass Ihre Lösung das gleichzeitige Ablesen der Tafel durch mehrere Sportler erlaubt.

- *Sp* wartet bis der ihm zugewiesene Testraum frei wird, betritt dann den Testraum (Aufruf von `enter_T()` mit T1 oder T2 als Parameter), führt den Test durch (Aufruf von `fitness_test()`), und verlässt den Testraum durch die Ausgangstür, am Plan rechts (Aufruf von `exit_T()`).

Lösen Sie die Synchronisationsaufgabe mit *Semaphoren* und geben Sie Initialisierungen für die Semaphore an. Achten Sie bei der Implementierung von *Sp* darauf, dass die Lösung die Ausführung und Synchronisation einer "beliebigen" Anzahl von Sportlern erlaubt.

(a) Initialisierungen

(b) Code für *Sp*

VO 182.022 24. Januar 2011

Prüfung Betriebssysteme

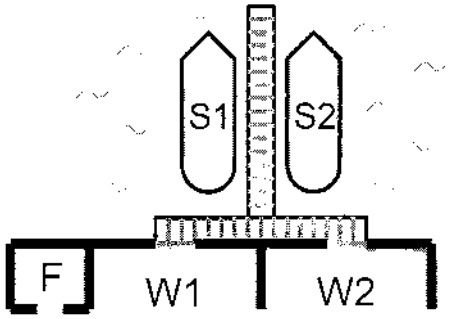
KNr. MNr. Zuname, Vorname

Ges.)(100) 1.)(30) 2.)(25) 3.)(45) Zusatzblätter:

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Synchronisation mit Semphoren (30)

Sie sollen einen Softwareentwickler beim Schreiben einer Simulationssoftware für eine Schiffsanlegestelle (siehe Abbildung) unterstützen. Diese Software simuliert die Aktionen von Schiffen und Passagieren durch Prozesse, die mit Semaphoren synchronisiert werden.



An der Schiffsanlegestelle gibt es Anlegeplätze für zwei Schiffe, S1 und S2, zwei Wartezonen, W1 und W2, die jeweils einem Schiff zugeordnet sind, und einen Fahrkartenschalter F.

Personen, die mit einem der Schiffe, S1 oder S2, verreisen, lösen am Schalter F eine Fahrkarte für ihr Schiff und gehen dann in die entsprechende Wartezone. Dort warten sie, bis sie an die Reihe kommen und gehen dann über den Steg auf ihr Schiff, um ihre Schiffsreise anzutreten.

Schiffe kommen jeweils leer am Steg an, warten dann, bis die erlaubte Anzahl von Passagieren an Bord ist, und verlassen dann den Steg wieder.

Ergänzen Sie in den Codestücken die fehlenden Semaphoroperationen, um die Prozesse unter folgenden Rahmenbedingungen zu synchronisieren.

- Ein Schiff kann N Passagiere aufnehmen. Schiffe fahren leer zur Anlegestelle, nehmen dort N Passagiere auf und fahren dann wieder ab.
- Für die Durchgänge von den Warteräumen zum Steg gilt, dass jeder Durchgang von maximal zwei Personen gleichzeitig passiert werden kann.
- Die Einstiegstelle der Schiffe ist sehr schmal. Passagiere können die Schiffe daher nur einzeln besteigen.

- Aus Sicherheitsgründen dürfen sich zu jedem Zeitpunkt maximal K Personen auf dem Steg befinden.
- Beim Anlegen bzw. Ablegen eines Schiffes dürfen sich keine Passagiere am Steg befinden.
- Schiffe und Passagiere sollen unter Einhaltung der angegebenen Regeln möglichst unabhängig und möglichst ohne Einschränkungen der Parallelität agieren können.

(a) Initialisierungen (vor Start der Prozesse)

(b) Prozesse für Schiffe

Bedeutung der verwendeten Funktionen:

`anlegen(S)`	Schiff S legt am Steg an
`abfahren(S)`	Schiff S legt vom Steg ab

```
/** Schiff S1 **/

anlegen(S1)

abfahren(S1)
```

```
/** Schiff S2 **/

anlegen(S2)

abfahren(S2)
```

(c) Prozesse für Passagiere

Bedeutung der verwendeten Funktionen:

`fahrkarte_kaufen()`	der Passagier kauft eine Fahrkarte
`warteraum_betreten(W)`	der Passagier betritt den Warteraum W
`steg_betreten()`	der Passagier betritt und geht auf dem Steg
`schiff_betreten(S)`	der Passagier betritt das Schiff S

```
/**  Passagier fuer S1  **/
fahrkarte_kaufen()
warteraum_betreten(W1)
steg_betreten()
schiff_betreten(S1)
```

```
/**  Passagier fuer S2  **/
fahrkarte_kaufen()
warteraum_betreten(W2)
steg_betreten()
schiff_betreten(S2)
```