3A Synchronisation: Lösungen

1.a.) wird aktiv gewartet

1.b.) zur Durchsetzung des wechselseitigen Ausschlusses

1.c.) Zählvariable, Warteschlange

1.d.) alle drei richtig

1.e.) um andere Werte als 1 erhöht und gesenkt werden, außer mit 0 oder 1 auch mit anderen Werten initialisiert werden

1.f.) zur Erzeugung einer neuen Semaphorgruppe, zum Zugriff auf eine existierende Semaphorgruppe

1.g.) zur Zuweisung von Anfangswerten, zur Löschung von Semaphoren

1.h.) join(), wait()/notify()

2.a.) Synchronisationsmechanismen

2.b.) sigaction()

2.c.) Monitor

2.d.) Event

2.e.) synchronized

2.f.) Deadlock

3.a.) Falsch. Es sind Synchronisationsbedingungen, die dann mit Synchronisationsmechanismen (Signale, Lock-Dateien, Semaphore, ...) durchgesetzt werden.

3.b.) Wahr. Ein Deadlock ergibt sich, wenn in einer Gruppe von Prozessen jeder Prozess auf eine Aktion eines anderen Prozesses wartet.

3.c.) Falsch. Die Synchronisation mit Lock-Dateien stützt sich auf die Information, ob eine Lock-Datei existiert; die Datei kann dabei leer sein.

3.d.) Wahr. Lock-Dateien setzen wechselseitige Ausschlüsse durch; dies kann man auch mit Semaphoren erreichen.

3.e.) Falsch. Beispielsweise kann man Java-Threads durch Semaphore synchronisieren.

3.f.) Wahr. Mit ihr wird eine Synchronisationsbedingung bezüglich dieses Monitors durchgesetzt.

3.g.) Falsch. Deadlocks können nur bei P-Operationen auftreten; V-Operationen blockieren Prozesse nicht.

3.h.) Falsch. wait() in Java wartet auf einen notify()-Aufruf im selben Objekt; wait() in UNIX/Linux wartet auf das Ende eines Prozesses.

4.a.) Thread, denn er ist kein Synchronisationsmechanismus.

4.b.) wechselseitiger Ausschluss, denn es ist eine Synchronisationsbedingung, also kein Synchronisationsmechanismus wie die anderen.

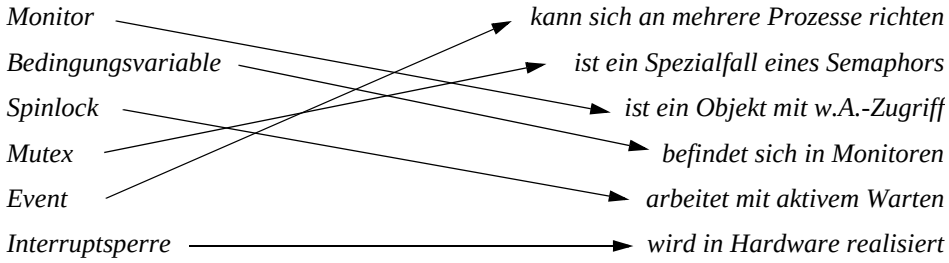
4.c.) Signal, denn es ist ein Synchronisationsmechanismus für Reihenfolgebedingungen; die anderen sind Mechanismen für den wechselseitigen Ausschluss.

4.d.) fork(), denn diese Funktion bezieht sich nicht direkt auf Signale.

4.e.) IPC_PRIVATE, denn es ist kein Kommando-Code, der an semctl übergeben werden kann.

4.f.) notify(), denn diese Methode wird nicht in der Java-Klasse Semaphore definiert.

5.)Begriffe und Eigenschaften:



6.a.) Durchsetzen zeitlicher Bedingungen bei der nebenläufigen Ausführung von Prozessen.

6.b.) 1.) wechselseitiger Ausschluss: Höchstens ein Prozess darf in einem „kritischen Abschnitt“ aktiv sein (wobei aber die Reihenfolge der Prozesse gleichgültig ist).
2.) Reihenfolge: Die Reihenfolge, in der die Prozesse ablaufen, ist festgelegt.

6.c.) Werkzeuge zum Durchsetzen von Synchronisationsbedingungen.

6.d.) Eine Situation, in die mehrere Prozesse verwickelt sind, indem jeweils ein Prozess auf eine Aktion eines anderen Prozesses wartet.

6.e.) Es wird aktiv gewartet, was die CPU belastet.

6.f.) P-Operation: Senkt den Zähler des Semaphors, sofern dieser dadurch nicht unter 0 sinkt. Läßt ansonsten den Zählerwert zunächst unverändert und blockiert den ausführenden Prozess.
V-Operation: Erhöht den Zähler des Semaphors. Entblockiert einen Prozess, der auf diesem Semaphor wartet.

6.g.) Ein Objekt, dessen Zugriffsoperationen wechselseitig ausgeschlossen ausgeführt werden.

6.h.) Ja: Da die Zugriffsoperationen wechselseitig ausgeschlossen ausgeführt werden, können sich Threads, die den Monitor benutzen, dabei nicht gegenseitig stören.

6.i.) Eine Bedingungsvariable ist stets einem Monitor zugeordnet, kann also nicht außerhalb dieses Monitors benutzt werden. Sie wird verwendet, um eine Synchronisationsbedingung bezüglich der Zugriffsfunktionen ihres Monitors durchzusetzen.

6.j.) Mit kill() schickt ein Prozess ein Signal an einen anderen Prozess, mit pause() wartet ein Prozess auf ein Signal.

6.k.) Mit sigaction() kann ein Signal-Handler zur Reaktion auf ein bestimmtes Signal definiert werden. Er wird ausgeführt, wenn das entsprechende Signal per kill() von einem anderen Prozess geschickt wurde.

6.l.) P-Operationen auf mehreren Semaphoren einer Gruppe „gleichzeitig“ möglich (d.h. Dekrementierungen werden atomar erst dann ausgeführt, wenn keiner der beteiligten Semaphore blockiert).

6.m.)ipcrm, semctl(..., IPC_RMID).

6.n.) Condition.

7.)Synchronisationsmechanismen mit den Synch.bedingungen, die sie durchsetzen:

	w.A.	*Reihenfolge*
Interruptsperrung	*x*	
Spinlock	*x*	
Signale		*x*
Semaphore	*x*	*x*
Monitore	*x*	*(x)*

8.)links V-Operation, rechts P-Operation

In der P-Operation müssen der Test und die Dekrementierung von COUNT als atomares Programmstück ausgeführt werden.

Hat die Zählvariable den Wert 1, so könnten zwei Prozesse unmittelbar hintereinander den Test ausführen, beide im if-Teil weiterlaufen und somit den Variablenwert negativ werden lassen.
Alternativ/ergänzend: Gesamte V-Operation muss atomar sein, da sonst ein gerade ankommender Prozess durch seine P-Operation durchlaufen könnte und zusätzlich ein wartender Prozess entblockiert werden könnte.

9.)wechselseitiger Ausschluss:
Initialisierung: S_WA.INIT(1)
Prozess A: S_WA.P(); kritischer Abschnitt; S_WA.V();
Prozess B: S_WA.P(); kritischer Abschnitt; S_WA.V();
Reihenfolge:
Initialisierung: S_REIHE.INIT(0)
Prozess 1: Vorgängeraktion; S_REIHE.V();
Prozess 2: S_REIHE.P(); Nachfolgeraktion;

3A.1 Sprachunabhängige Anwendungsaufgaben

1.)Die Feststellung, ob es noch Karten gibt, und der Versuch der Kartenreservierung / des Kartenkaufs waren zwei zeitlich getrennte Vorgänge. Sie hätten gleich im Telefonanruf (der eine atomare Operation ist) eine Karte reservieren müssen.

2.)Die Vorgehensweise ist nicht sicher: Hat die S-Bahn Verspätung, so stoßen die Züge zusammen. Eine „Synchronisation“ durch sleep()-Aufrufe, durch die Nachfolgerprozesse für eine festgelegte Zeit blokkiert werden, ist daher nicht zulässig, denn es ist nicht vorherbestimmt, wie lange die Vorgängerprozesse für ihre Operationen brauchen.

3.)Der Vorschlag ist nicht fair (verglichen mit der Behandlung der „normalen“ Kunden), denn es ist recht unwahrscheinlich, dass jemals vier zusammenhängende Plätze frei werden. Man sollte also besser so vorgehen, dass man einen freien Platz (oder, falls vorhanden, zwei oder drei zusammenhängende Plätze) reserviert und dann wartet, bis auch ein bzw. mehrere dazu benachbarte Plätze frei werden.

Bezogen auf ein Prozess-System bedeutet das, dass man bei einem Prozess mit viel Speicherbedarf nicht warten sollte, bis genügend Speicher frei wird, sondern Speicher schrittweise reservieren sollte.

4.)Die Synchronisation erfolgt durch eine zentrale Stelle (also durch einen zentralen Steuerungsprozess), während sich bei Semaphoren etc. die beteiligten Prozesse untereinander selbst synchronisieren.

5.)P1 P2 P2 P3 P3 und P1 P3 P3 P2 P2.

P1 P2 P3 P2 P3 ist nicht möglich, da die Prozesse 2 und 3 ihre Ausgaben wechselseitig ausgeschlossen machen; P2 P2 P1 P3 P3 ist nicht möglich, da P2 stets nach P1 ausgeführt wird.

6.)S1 Reihenfolgebedingung, S2 wechselseitiger Ausschluss.

Prozess A wird als erster ausgeführt, dann Prozess B und Prozess C. B und C sind wechselseitig ausgeschlossen, ihre Reihenfolge untereinander ist nicht festgelegt.

7.)Die Lösung kann zu einem Deadlock führen, da P2 den Drucker belegt und dann auf das Ende von P1 wartet. P1 kann aber nicht terminieren, da er den Drucker nicht belegen kann. Man könnte in P2 die Reihenfolge der beiden P-Operationen vertauschen – oder ganz einfach S_WA weglassen.

8.)Solange Prozess 1 im kritischen Abschnitt arbeitet, darf keiner der übrigen Prozesse dort arbeiten. Von den Prozessen 2-10 dürfen höchstens drei gleichzeitig im kritischen Abschnitt arbeiten.

9.)Man erreicht das Ziel nicht, denn die if-Abfrage und die nachfolgende P-Operation laufen nicht atomar ab. Damit ist folgender Ablauf möglich: Prozess 1 führt die Abfrage aus und erhält das Resultat true. Bevor er die P-Operation ausführen kann, wird der Prozessor zu Prozess 2 umgeschaltet, der seinerseits die Abfrage mit positivem Ergebnis durchführt und in seinen kritischen Abschnitt weiterläuft. Nach einer erneuten Umschaltung blockiert Prozess 1 in seiner P-Operation.

Die Lösung ist aber nicht unsicher, denn der Semaphor setzt den wechselseitigen Ausschluss wirksam durch.

10.)Prozess-System:

<table>
<tr><td colspan="4">Semaphore: S1.INIT(0); S2.INIT(0)</td></tr>
<tr><td>Prozess 1:
arbeite
S1.V()</td><td>Prozess 2:
arbeite
S1.V()</td><td>Prozess 3:
S1.P()
S1.P()
arbeite
S2.V()</td><td>Prozess 4:
S2.P()
arbeite</td></tr>
</table>

11.)Prozess-System:

Semaphore: S_AB.INIT(0); S_BA.INIT(0); S_AC.INIT(0)		
Prozess A: *Ausgabe* S_AB.V() S_BA.P() *Ausgabe* S_AC.V()	Prozess B: S_AB.P() *Ausgabe* S_BA.V()	Prozess C: S_AC.P() *Ausgabe*

12.)Prozess-System:

Semaphore: S_REIHE.INIT(0); S_WA.INIT(2)	
Professor P: *Aufgabenblatt schreiben* S_REIHE.V() S_REIHE.V() S_REIHE.V()	Studenten S1-S3: S_REIHE.P() S_WA.P() *arbeiten* S_WA.V()

13.)Prozesse: Ritter, Diener, Hund

Aktionen der Prozesse:

Ritter:	Diener:	Hund:
auf Wein warten *fünfmal:* *Kotelett nehmen, wenn da* *Kotelett essen* *Knochen wegwerfen* *zum Sessel gehen* *evtl. warten, bis Sessel frei* *ein Stunden im Sessel schlafen* *Sessel freigeben*	*Wein bringen* *Kotelett braten* *Kotelett bringen* *Kotelett braten* *Kotelett bringen*	*auf Knochen warten* *Knochen schnappen* *zum Sessel laufen* *evtl. warten, bis Sessel frei* *eine Stunde im Sessel schlafen* *Sessel freigeben*

4 Synchronisationsbedingungen:
1.) Ritter isst, nachdem Diener Wein gebracht hat (Reihenfolge)
2.) Ritter kann ein Kotelett erst essen, wenn es da ist (Reihenfolge)
3.) Hund schnappt Knochen, nachdem Ritter sie zur Seite geworfen hat (Reihenfolge)
4.) Sessel kann nur von einem gleichzeitig benutzt werden (w. A.)

Semaphore:
Synch.bedingung 1: S_WEIN.INIT(0);
Synch.bedingung 2: S_KOTELETT.INIT(3)
Synch.bedingung 3: S_KNOCHEN.INIT(0)
Synch.bedingung 4: S_SESSEL.INIT(1)

vollständiges Prozess-System:

Ritter:	Diener:	Hund:

S_WEIN.P()
// auf Wein warten
fünfmal:
S_KOTELETT.P()
// Kotelett nehmen
Kotelett essen
S_KNOCHEN.V()
// Knochen wegwerfen
zum Sessel gehen
S_SESSEL.P()
// warten, bis Sessel frei
eine Stunde im Sessel
schlafen
S_SESSEL.V()
// Sessel freigeben
	S_WEIN.V()
	// Wein bringen
	Kotelett braten
	S_KOTELETT.V()
	// Kotelett bringen
	Kotelett braten
	S_KOTELETT.V()
	// Kotelett bringen
		S_KNOCHEN.P()
		// auf Knochen warten
		Knochen schnappen
		zum Sessel laufen
		S_SESSEL.P()
		// warten, bis Sessel frei
		eine Stunde im Sessel
		schlafen
		S_SESSEL.V()
		// Sessel freigeben

14.)Prozesse: Friseur, Kunden

Aktionen der Prozesse:

Friseur:	Kunde:

while (true) {
den Kunden die Bereitschaft
zur Bedienung signalisieren
warten, wenn kein Kunde da ist
den Kunden bedienen
„Fertig" rufen
}
	dem Friseur die Anwesenheit signalisieren,
	ihn ggf. wecken
	warten, bis der Friseur bedienen kann
	im Stuhl Platz nehmen
	warten, bis der Friseur fertig bedient hat

Synchronisationsbedingungen:
1.) Der Friseur muss schlafen, solange kein Kunde da ist.
2.) Ein Kunde muss warten, bis der Friseur ihn an die Reihe nimmt.
3.) Ein Kunde muss im Stuhl sitzen bleiben, bis der Friseur „Fertig!“ ruft.

Semaphore:
Synch.bedingung 1: FRISEUR_BEREIT.INIT(0);
Synch.bedingung 2: KUNDEN_DA.INIT(0)
Synch.bedingung 3: FERTIG.INIT(0)

vollständiges Prozess-System:

Friseur:	Kunde:
while (true) { FRISEUR_BEREIT.V() *// den Kunden die Bereitschaft* *// zur Bedienung signalisieren* KUNDEN_DA.P() *// warten, wenn kein Kunde da ist* *den Kunden bedienen* FERTIG.V() *// „Fertig" rufen* *}*	KUNDEN_DA.V() *// dem Friseur die Anwesenheit signali-* *// sieren, ihn ggf. wecken* FRISEUR_BEREIT.P() *// warten, bis der Friseur bedienen kann* *im Stuhl Platz nehmen* FERTIG.P() *// warten, bis Friseur mit Arbeit fertig*

15.)wechselseitige Ausschlüsse (ein Stift kann nur von einem Kind gleichzeitig benutzt werden)

Prozess-System mit Semaphoren:

Semaphore: ROT.INIT(1); GRÜN.INIT(1); BLAU.INIT(1)		
Anna:	Bolle:	Christina:
ROT.P() GRÜN.P() *Baum malen* ROT.V() GRÜN.V() BLAU.P() *Himmel malen* BLAU.V()	ROT.P() GRÜN.P() BLAU.P() *Haus malen* GRÜN.V() BLAU.V() *Feuerwehr malen* ROT.V()	GRÜN.P() *grüne Teile malen* GRÜN.V() BLAU.P() *blaue Teile malen* BLAU.V() ROT.P() *rote Teile malen* ROT.V()

Ein Deadlock kann wie folgt auftreten: Anna greift den roten, Bolle den blauen und Christina den grünen Stift. Alle warten dann, dass "ihr" zweiter Stift frei wird.

Der Deadlock könnte verhindert werden durch das Alles-oder-Nichts-Prinzip (jedes Kind muss beide Stifte gleichzeitig nehmen) oder eine festgelegte Reihenfolge (erst rot nehmen, dann grün und dann blau).

16.)Es kommt zu einem Deadlock, wenn z.B. der Verbraucher bei leerem Puffer blockiert und dabei den wechselseitigen Ausschluss nicht aufhebt. Die Reihenfolge der P-Operationen im Erzeuger und im Verbraucher muss daher jeweils vertauscht werden.

17.)Alle Bücher A sind an eine Gruppe von Studenten ausgeliehen, alle Bücher B an eine andere Gruppe, und niemand gibt seine Bücher frei.

Geordnete Ressourcen-Anforderung: Man muss immer zuerst Buch A ausleihen und dann Buch B.
Alles-oder-nichts-Prinzip: Man muss beide Bücher in einem Vorgang ausleihen.
Gemeinsame Ressourcen-Nutzung: Mehrere Studenten bilden eine Arbeitsgruppe, die sich ihre Bücher teilt.
Temporäre Ressourcen-Freigabe: Ein Student gibt sein Buch kurzfristig an einen anderen weiter.

Bei jedem Ausleihvorgang wird geprüft, ob das System durch ihn auf einen Deadlock hinläuft; er wird dann verweigert (aufwendig!).

Aufstockung der Ressourcen: Die Bibliothek beschafft mehr Bücher.
Begrenzung der Ressourcen-Nutzung: Bücher müssen nach einer Leihfrist zurückgegeben werden.
Freimachen einzelner Ressorcen: Anderweitig benötigte Bücher werden zurückgefordert
Freimachen mehrerer oder aller Ressourcen durch Beendigung einzelner Prozesse oder Neustart des Systems: Einer oder alle Studenten werden erschossen ;-)

18.)begrenzter Stack als Monitor:

Monitor Stack
- voll : cond - leer : cond - kapazität : int - inhalt : int[kapazität] - spitze : int = -1
+ schreiben(neu : int) + lesen() : int

```
schreiben(neu : int):
  while (spitze==kapazität-1)
    wait(voll)
  spitze++
  inhalt[spitze] = neu
  signal(leer)
```

```
lesen():
  while (spitze == -1)
    wait(leer)
  hilfsvariable = inhalt[spitze]
  spitze--
  signal(voll)
  return hilfsvariable
```

(cond = Bedingungsvariable)

19.)Gesicherter Speicher als Monitor (zwei Entsperroperationen nötig):

Monitor GesicherterSpeicher
- sperreCond : cond - gesperrt1, gesperrt2 : boolean = true - inhalt : int
+ schreiben(neu : int) + lesen : int + entsperren1() + entsperren2()

```
schreiben(neu : int):
  inhalt = neu
```

```
lesen():
  while (gesperrt1 || gesperrt2)
    wait(sperreCond)
  hilfsvariable = inhalt
  gesperrt1 = true
  gesperrt2 = true
  return hilfsvariable
```

```
entsperren1():
  gesperrt1 = false
  signal(sperreCond)
```

```
entsperren2():
  gesperrt2 = false
  signal(sperreCond)
```

(cond = Bedingungsvariable)

Gesicherter Speicher als Monitor (nur eine Entsperroperation nötig):

Monitor GesicherterSpeicher
- sperreCond : cond - gesperrt : boolean = true - inhalt : int
+ schreiben(neu : int) + lesen : int + entsperren1() + entsperren2()

```
schreiben(neu : int):
  inhalt = neu
```

```
lesen():
  while (gesperrt)
    wait(sperreCond)
  hilfsvariable = inhalt
  gesperrt = true
  return hilfsvariable
```

```
entsperren1():
  gesperrt = false
  signal(sperreCond)
```

```
entsperren2():
  gesperrt = false
  signal(sperreCond)
```

(cond = Bedingungsvariable)

3A.2 Programmierung unter UNIX/Linux

1.)Durch das sleep(1) ist es zwar sehr wahrscheinlich, dass Sohn 1 erst aktiv wird, wenn Sohn 2 bereits fertig ist. Es ist aber nicht sicher, denn die Scheduling-Entscheidungen des Betriebssystems können unter ungünstigen Umstönden dazu führen, dass Sohn 2 länger als eine Sekunde verzögert wird. Die Vorgehensweise ist also nicht korrekt.

2.)Das Programm ist nicht korrekt, da das Lesen und Schreiben der lock-Variablen (while (lock==1) bzw. lock=1) nicht atomar ist. Bei ungünstigem Scheduling können beide Prozesse gleichzeitig in den kritischen Bereich gelangen. Korrektur: Semaphore benutzen.

3.)CCC BBB AAA DDD (Söhne warten in den pause()-Aufrufen auf Signale des Vaters; Vater gibt nach 10 Sekunden CCC aus und schickt dann ein Signal an den zweiten Sohn; zweiter Sohn gibt BBB aus und terminiert dann; Vater wartet auf Terminierung des zweiten Sohns und schickt dann Signal an den ersten Sohn; erster Sohn gibt AAA aus und terminiert dann; Vater wartet auf Terminierung des ersten Sohns und gibt dann DDD aus).

Nein – die Laufzeit hängt davon ab, wann der Scheduler den Prozessen die CPU zuteilt.

4.)
```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
void sighand() {}
main() {
 int pa, pb, pc, i;
 struct sigaction sigact;
 sigact.sa_handler = sighand;
 sigemptyset(&sigact.sa_mask);
 sigact.sa_flags = 0;
 sigaction(SIGUSR1,&sigact,NULL);
 if ((pc=fork())==0 ) {
  pause();
  for (;;) {
   sleep(1);
   printf("CCC\n");
  }
  exit(0);
 }
 if ((pb=fork())==0 ) {
  pause();
  for (;;) {
   sleep(1);
   printf("BBB\n");
  }
  exit(0);
 }
 if ((pa=fork())==0 ) {
```

```
   for (i=0;i<3;i++) {
    sleep(1);
    printf("AAA\n");
   }
   kill(pb,SIGUSR1);
   kill(pc,SIGUSR1);
   exit(0);
  }
 sleep(10);
 kill(pb,SIGKILL);
 kill(pc,SIGKILL);
}
```

5.)SEM.P() durch pause() ersetzen, SEM.V() durch kill() ersetzen.

6.)einer (siehe zweiter Parameter von semget()).

Reihenfolge.

0 (Semaphor wird mit 0 initialisiert, p1 zählt zunächst den Semaphorwert um 1 hinauf, p2 zählt ihn dann um 1 herunter).

semctl(s,0,IPC_RMID,0)

7.)Der semget-Aufruf erzeugt eine neue Gruppe von 5 Semaphoren. Der semctl-Aufruf initialisiert den zweiten Semaphor in dieser Gruppe mit dem Wert 0 und die übrigen Semaphore mit dem Wert 1.

Der Prozess wird blockiert, da die Zählvariable von Semaphor Nr. 1 (also dem zweiten in dieser Gruppe) den Wert 0 hat und daher nicht weiter gesenkt werden kann. Die Werte sämtlicher Semaphore bleiben vorerst unverändert („Alles-oder-nichts-Prinzip“).

Die Werte des vorletzten und letzten Semaphors würden auf 0 gesenkt; der Prozess würde nicht blockiert.

8.)int semid;
struct sembuf sem_p[2];
semid = semget(IPC_PRIVATE,4,IPC_CREAT|0777);
...
sem_p[0].sem_num = 1; sem_p[0].sem_op = -1; sem_p[0].sem_flg = 0;
sem_p[1].sem_num = 3; sem_p[1].sem_op = -1; sem_p[1].sem_flg = 0;
semop(semid,sem_p,2);

9.) ausgewählte Programme:

zu Aufgabe 3A.2.10. (Umsetzung eines Ablaufplans):

```
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>
main() {
 int status;
 int sem1, sem2;
```

```
unsigned short init_array[1];
struct sembuf sem_p, sem_v;
/* zwei Semaphore erzeugen */
sem1 = semget(IPC_PRIVATE,1,IPC_CREAT|0777);
sem2 = semget(IPC_PRIVATE,1,IPC_CREAT|0777);
/* Semaphore mit 0 vorbesetzen, da Reihenfolge */
init_array[0] = 0;
semctl(sem1,0,SETALL,init_array);
semctl(sem2,0,SETALL,init_array);
/* P- und V-Operationen vorbereiten */
sem_p.sem_num = 0; sem_v.sem_num = 0;
sem_p.sem_op = -1; sem_v.sem_op = 1;
sem_p.sem_flg = 0; sem_v.sem_flg = 0;
/* vier Sohnprozesse erzeugen */
if (fork()==0) {
 sleep(2);
 printf("Prozess 1\n");
 semop(sem1,&sem_v,1);
 exit(0);
}
if (fork()==0) {
 sleep(1);
 printf("Prozess 2\n");
 semop(sem1,&sem_v,1);
 exit(0);
}
if (fork()==0) {
 semop(sem1,&sem_p,1);
 semop(sem1,&sem_p,1);
 printf("Prozess 3\n");
 semop(sem2,&sem_v,1);
 exit(0);
}
if (fork()==0) {
 semop(sem2,&sem_p,1);
 printf("Prozess 4\n");
 exit(0);
}
/* Ende der Söhne abwarten */
wait(&status);
wait(&status);
wait(&status);
wait(&status);
/* Semaphore löschen */
semctl(sem1,0,IPC_RMID,0);
```

```
 semctl(sem2,0,IPC_RMID,0);
}
```

zu Aufgabe 3A.2.13. (Ritter / Diener / Hund):

```
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>
main() {
 int i, status;
 int s_wein, s_kotelett, s_knochen, s_sessel;
 unsigned short init_array[1];
 struct sembuf sem_p, sem_v;
 /* Semaphore erzeugen */
 s_wein = semget(IPC_PRIVATE,1,IPC_CREAT|0777);
 s_kotelett = semget(IPC_PRIVATE,1,IPC_CREAT|0777);
 s_knochen = semget(IPC_PRIVATE,1,IPC_CREAT|0777);
 s_sessel = semget(IPC_PRIVATE,1,IPC_CREAT|0777);
 /* Semaphore vorbesetzen, da Reihenfolge */
 init_array[0] = 0;
 semctl(s_wein,0,SETALL,init_array);
 init_array[0] = 3;
 semctl(s_kotelett,0,SETALL,init_array);
 init_array[0] = 0;
 semctl(s_knochen,0,SETALL,init_array);
 init_array[0] = 1;
 semctl(s_sessel,0,SETALL,init_array);
 /* P- und V-Operationen vorbereiten */
 sem_p.sem_num = 0; sem_v.sem_num = 0;
 sem_p.sem_op = -1; sem_v.sem_op = 1;
 sem_p.sem_flg = 0; sem_v.sem_flg = 0;
 /* Ritter */
 if (fork()==0) {
  printf("Ritter: Ich warte auf Wein\n");
  semop(s_wein,&sem_p,1);
  printf("Ritter: Ich habe den Wein\n");
  for (i=0;i<5;i++) {
    printf("Ritter: Ich will ein Kotelett\n");
    semop(s_kotelett,&sem_p,1);
    printf("Ritter: Ich esse ein Kotelett\n");
    sleep(2);
  }
  printf("Ritter: Ich werfe die Knochen\n");
  semop(s_knochen,&sem_v,1);
  printf("Ritter: Ich gehe zum Sessel\n");
  sleep(2);
```

```
 semop(s_sessel,&sem_p,1);
 printf("Ritter: Ich sitze im Sessel\n");
 sleep(4);
 printf("Ritter: Ich stehe vom Sessel auf\n");
 semop(s_sessel,&sem_v,1);
 exit(0);
}
/* Diener */
if (fork()==0) {
 sleep(2);
 printf("  Diener: Ich bringe den Wein\n");
 semop(s_wein,&sem_v,1);
 sleep(2);
 printf("  Diener: Ich brate ein Kotelett\n");
 sleep(4);
 printf("  Diener: Ich bringe das Kotelett\n");
 semop(s_kotelett,&sem_v,1);
 printf("  Diener: Ich brate ein Kotelett\n");
 sleep(4);
 printf("  Diener: Ich bringe das Kotelett\n");
 semop(s_kotelett,&sem_v,1);
 exit(0);
}
/* Hund */
if (fork()==0) {
 printf("    Hund: Ich will Knochen\n");
 semop(s_knochen,&sem_p,1);
 printf("    Hund: Ich habe die Knochen\n");
 sleep(2);
 printf("    Hund: Ich laufe zum Sessel\n");
 semop(s_sessel,&sem_p,1);
 printf("    Hund: Ich liege im Sessel\n");
 sleep(4);
 printf("    Hund: Ich gebe den Sessel frei\n");
 semop(s_sessel,&sem_v,1);
 exit(0);
}
/* Ende der Prozesse abwarten */
wait(&status);
wait(&status);
wait(&status);
/* Semaphore löschen */
semctl(s_wein,0,IPC_RMID,0);
semctl(s_kotelett,0,IPC_RMID,0);
semctl(s_knochen,0,IPC_RMID,0);
```

```
 semctl(s_sessel,0,IPC_RMID,0);
}
```

zu Aufgabe 3A.2.14. (Sleeping Barber):

```
#define ANZKUNDEN 3
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
main() {
 int i, status;
 int pid_friseur;
 int friseur_bereit, kunden_da, fertig;
 unsigned short init_array[1];
 struct sembuf sem_p, sem_v;
 /* Semaphore erzeugen */
 friseur_bereit = semget(IPC_PRIVATE,1,IPC_CREAT|0777);
 kunden_da = semget(IPC_PRIVATE,1,IPC_CREAT|0777);
 fertig = semget(IPC_PRIVATE,1,IPC_CREAT|0777);
 /* Semaphore vorbesetzen */
 init_array[0] = 0;
 semctl(friseur_bereit,0,SETALL,init_array);
 semctl(kunden_da,0,SETALL,init_array);
 semctl(fertig,0,SETALL,init_array);
 /* P- und V-Operationen vorbereiten */
 sem_p.sem_num = 0; sem_v.sem_num = 0;
 sem_p.sem_op = -1; sem_v.sem_op = 1;
 sem_p.sem_flg = 0; sem_v.sem_flg = 0;
 /* Friseur */
 if ((pid_friseur=fork())==0)
  while (1) {
   printf("Friseur: Ich bin bereit\n");
   semop(friseur_bereit,&sem_v,1);
   semop(kunden_da,&sem_p,1);
   printf("Friseur: Kunde kommt dran\n");
   sleep(2);
   printf("Friseur: Kunde ist fertig\n");
   semop(fertig,&sem_v,1);
  }
 sleep(2);
 /* Kunden */
 for (i=0;i<ANZKUNDEN;i++)
  if (fork()==0) {
   printf("    Kunde %d: Ich moechte bedient werden\n",i);
   semop(kunden_da,&sem_v,1);
```

```
  semop(friseur_bereit,&sem_p,1);
  printf("    Kunde %d: Ich komme dran\n",i);
  semop(fertig,&sem_p,1);
  printf("    Kunde %d: Ich bin fertig\n",i);
  exit(0);
 }
 /* Ende der Kunden abwarten */
 for (i=0;i<ANZKUNDEN;i++)
  wait(&status);
 /* Friseur terminieren */
 kill(pid_friseur, SIGKILL);
 /* Semaphore löschen */
 semctl(friseur_bereit,0,IPC_RMID,0);
 semctl(kunden_da,0,IPC_RMID,0);
 semctl(fertig,0,IPC_RMID,0);
}
```

10.)

```
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>
main() {
 int sema, status;
 unsigned short init_array[3]; // 0 = rot, 1 = gruen, 2 = blau
 struct sembuf sem_p[3], sem_v[3];
 /* Semaphore erzeugen */
 sema = semget(IPC_PRIVATE,3,IPC_CREAT|0777); // 0 = rot, 1 = gruen, 2 = blau
 /* Semaphore vorbesetzen */
 init_array[0] = init_array[1] = init_array[2] = 1;
 semctl(sema,0,SETALL,init_array);
 /* P- und V-Operationen vorbereiten (bis auf die Semaphornummern) */
 sem_p[0].sem_op = sem_p[1].sem_op = sem_p[2].sem_op = -1;
 sem_v[0].sem_op = sem_v[1].sem_op = sem_v[2].sem_op = 1;
 sem_p[0].sem_flg = sem_p[1].sem_flg = sem_p[2].sem_flg = 0;
 sem_v[0].sem_flg = sem_v[1].sem_flg = sem_v[2].sem_flg = 0;
 /* Anna */
 if (fork()==0) {
  sem_p[0].sem_num = sem_v[0].sem_num = 0;
  sem_p[1].sem_num = sem_v[1].sem_num = 1;
  semop(sema,sem_p,2);
  printf("Anna: Ich male den Apfelbaum (rot, gruen)\n");
  sleep(1);
  printf("Anna: Ich bin mit dem Apfelbaum fertig\n");
  semop(sema,sem_v,2);
  sem_p[0].sem_num = sem_v[0].sem_num = 2;
  semop(sema,sem_p,1);
```

```
 printf("Anna: Ich male den Himmel (blau)\n");
 sleep(1);
 printf("Anna: Ich bin mit dem Himmel fertig\n");
 semop(sema,sem_v,1);
 exit(0);
 }
/* Bolle */
if (fork()==0) {
 sem_p[0].sem_num = sem_v[0].sem_num = 0;
 sem_p[1].sem_num = sem_v[1].sem_num = 1;
 sem_p[2].sem_num = sem_v[2].sem_num = 2;
 semop(sema,sem_p,3);
 printf("Bolle: Ich male das Haus (rot, gruen, blau)\n");
 sleep(1);
 printf("Bolle: Ich bin mit dem Haus fertig\n");
 semop(sema,sem_v,3);
 semop(sema,sem_p,1);
 printf("Bolle: Ich male das Feuerwehrauto (rot)\n");
 sleep(1);
 printf("Bolle: Ich bin mit dem Feuerwehrauto fertig\n");
 semop(sema,sem_v,1);
 exit(0);
 }
 /* Christina */
if (fork()==0) {
 sem_p[0].sem_num = sem_v[0].sem_num = 0;
 sem_p[1].sem_num = sem_v[1].sem_num = 1;
 sem_p[2].sem_num = sem_v[2].sem_num = 2;
 semop(sema,&sem_p[1],1);
 printf("Christina: Ich male gruene Farbe\n");
 sleep(1);
 printf("Christina: Ich bin mit gruener Farbe fertig\n");
 semop(sema,&sem_v[1],1);
 semop(sema,&sem_p[0],1);
 printf("Christina: Ich male rote Farbe\n");
 sleep(1);
 printf("Christina: Ich bin mit roter Farbe fertig\n");
 semop(sema,&sem_v[0],1);
 semop(sema,&sem_p[2],1);
 printf("Christina: Ich male blaue Farbe\n");
 sleep(1);
 printf("Christina: Ich bin mit blauer Farbe fertig\n");
 semop(sema,&sem_v[2],1);
 exit(0);
 }
```

```
 /* auf das Ende der Prozesse warten */
 wait(&status);
 wait(&status);
 wait(&status);
 /* Semaphore löschen */
 semctl(sema,0,IPC_RMID,0);
}
```

11.)

```
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
main() {
 int i, status, zufzahl;
 int raucher1, raucher2, raucher3, lieferant;
 int sema;
 unsigned short init_array[4];
                        // 0 = wechselseitiger Ausschluss, 1 = Tabak, 2 = Papier, 3 = Streichhoelzer
 struct sembuf sem_p[2], sem_v[2], sem_p_lieferant, sem_v_lieferant;
 /* Semaphore erzeugen */
 sema = semget(IPC_PRIVATE,4,IPC_CREAT|0777);
                        // 0 = Blockieren des Lieferanten, 1 = Tabak, 2 = Papier, 3 = Streichhoelzer
 /* Semaphore vorbesetzen */
 init_array[0] = 1;
 init_array[1] = init_array[2] = init_array[3] = 0;
 semctl(sema,0,SETALL,init_array);
 /* P- und V-Operationen fuer den w.A. vorbereiten */
 sem_p_lieferant.sem_num = sem_v_lieferant.sem_num = 0;
 sem_p_lieferant.sem_op  = -1; sem_v_lieferant.sem_op = 1;
 sem_p_lieferant.sem_flg = sem_v_lieferant.sem_flg = 0;
 /* P- und V-Operationen fuer Tabak / Papier / Streichhoelzer vorbereiten
(bis auf die Semaphornummern) */
 sem_p[0].sem_op = sem_p[1].sem_op = -1;
 sem_v[0].sem_op = sem_v[1].sem_op = 1;
 sem_p[0].sem_flg = sem_p[1].sem_flg = sem_v[0].sem_flg = sem_v[1].sem_flg = 0;
 /* Raucher, der Tabak besitzt */
 if ((raucher1=fork())==0) {
  sem_p[0].sem_num = 2;
  sem_p[1].sem_num = 3;
  while (1) {
   printf("Raucher 1: Ich warte auf Papier und Streichhoelzer\n");
   semop(sema,sem_p,2);
   printf("Raucher 1: Ich habe Papier und Streichhoelzer - ich rauche jetzt\n");
   sleep(1);
   printf("Raucher 1: Ich bin mit dem Rauchen fertig\n");
```

```
  semop(sema,&sem_v_lieferant,1);
 }
}
/* Raucher, der Papier besitzt */
if ((raucher2=fork())==0) {
 sem_p[0].sem_num = 1;
 sem_p[1].sem_num = 3;
 while (1) {
  printf("Raucher 2: Ich warte auf Tabak und Streichhoelzer\n");
  semop(sema,sem_p,2);
  printf("Raucher 2: Ich habe Tabak und Streichhoelzer - ich rauche jetzt\n");
  sleep(1);
  printf("Raucher 2: Ich bin mit dem Rauchen fertig\n");
  semop(sema,&sem_v_lieferant,1);
 }
}
/* Raucher, der Streichhoelzer besitzt */
if ((raucher3=fork())==0) {
 sem_p[0].sem_num = 1;
 sem_p[1].sem_num = 2;
 while (1) {
  printf("Raucher 3: Ich warte auf Tabak und Papier\n");
  semop(sema,sem_p,2);
  printf("Raucher 3: Ich habe Tabak und Papier - ich rauche jetzt\n");
  sleep(1);
  printf("Raucher 3: Ich bin mit dem Rauchen fertig\n");
  semop(sema,&sem_v_lieferant,1);
 }
}
/* Lieferant */
if ((lieferant=fork())==0)
 while (1) {
  semop(sema,&sem_p_lieferant,1);
  zufzahl = rand()/(RAND_MAX/3);
  switch (zufzahl) {
   case 0: printf("Lieferant: Ich liefere Tabak und Papier\n");
           sem_v[0].sem_num = 1;
           sem_v[1].sem_num = 2;
           break;
   case 1: printf("Lieferant: Ich liefere Tabak und Streichhoelzer\n");
           sem_v[0].sem_num = 1;
           sem_v[1].sem_num = 3;
           break;
   case 2: printf("Lieferant: Ich liefere Papier und Streichhoelzer\n");
           sem_v[0].sem_num = 2;
```

```
            sem_v[1].sem_num = 3;
            break;
     }
     semop(sema,sem_v,2);
    }
   sleep(20);
   /* Prozesse terminieren */
   kill(raucher1, SIGKILL);
   kill(raucher2, SIGKILL);
   kill(raucher3, SIGKILL);
   kill(lieferant, SIGKILL);
   /* Semaphore löschen */
   semctl(sema,0,IPC_RMID,0);
}
```

12.)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define STACKKAP 5
#define ERZZEIT 1
#define VBRZEIT 5
#define GESAMTLAUFZEIT 30
pthread_mutex_t mutex;
pthread_cond_t cond_voll, cond_leer;
int stack[STACKKAP], spitze;
void *erzeuger_loop(void *p) {
  int wert = 10;
  while (1) {
   sleep(ERZZEIT);
   pthread_mutex_lock(&mutex);
   while (spitze==STACKKAP-1) {
    printf("Stack voll => Erzeuger blockiert\n");
    pthread_cond_wait(&cond_voll,&mutex);
   }
   printf("Erzeuger: Schreiben des Werts %d beginnt\n",wert);
   sleep(1);
   spitze++;
   stack[spitze] = wert;
   wert += 10;
   printf("Erzeuger: Schreiben endet\n");
   pthread_cond_signal(&cond_leer);
   pthread_mutex_unlock(&mutex);
  }
}
void *verbraucher_loop(void *p) {
  while (1) {
```

```
    pthread_mutex_lock(&mutex);
    while (spitze==-1) {
     printf("Puffer leer => Verbraucher blockiert\n");
     pthread_cond_wait(&cond_leer,&mutex);
    }
    printf("Verbraucher: Lesen beginnt\n");
    sleep(1);
    printf(" gelesen: %d\n",stack[spitze]);
    spitze--;
    printf("Verbraucher: Lesen endet\n");
    pthread_cond_signal(&cond_voll);
    pthread_mutex_unlock(&mutex);
    sleep(VBRZEIT);
   }
}
int main (int argc, char *argv[]) {
   pthread_t erzeuger, verbraucher;
   pthread_mutex_t mutex;
   int rc;
   rc = pthread_mutex_init(&mutex, NULL);
   printf("mutex_init(): return code = %d\n",rc);
   rc = pthread_cond_init(&cond_voll, NULL);
   printf("cond_init(cond_voll): return code = %d\n",rc);
   rc = pthread_cond_init(&cond_leer, NULL);
   printf("cond_init(cond_leer): return code = %d\n",rc);
   spitze = -1;
   pthread_create(&erzeuger, NULL, erzeuger_loop, NULL);
   pthread_create(&verbraucher, NULL, verbraucher_loop, NULL);
   sleep(GESAMTLAUFZEIT);
   pthread_cancel(erzeuger);
   pthread_cancel(verbraucher);
   pthread_mutex_destroy(&mutex);
   pthread_cond_destroy(&cond_voll);
   pthread_cond_destroy(&cond_leer);
}
```

3A.3 Programmierung in Java

1.)zu Aufgabe 3A.2.10. (Umsetzung eines Ablaufplans):

```
import java.util.concurrent.*;
class Thread1 extends Thread {
 public void run() {
  System.out.println("Hier ist Thread 1");
  try { sleep(1000); } catch (InterruptedException e) {}
```

```
  Prog.sem1.release();
 }
}
class Thread2 extends Thread {
 public void run() {
  System.out.println("Hier ist Thread 2");
  try { sleep(1000); } catch (InterruptedException e) {}
  Prog.sem1.release();
 }
}
class Thread3 extends Thread {
 public void run() {
  try {
   Prog.sem1.acquire();
   Prog.sem1.acquire();
   System.out.println("Hier ist Thread 3");
   sleep(1000);
   Prog.sem2.release();
  } catch (InterruptedException e) {}
 }
}
class Thread4 extends Thread {
 public void run() {
  try {
   Prog.sem2.acquire();
   System.out.println("Hier ist Thread 4");
  } catch (InterruptedException e) {}
 }
}
public class Prog {
 static Semaphore sem1 = new Semaphore(0);
 static Semaphore sem2 = new Semaphore(0);
 public static void main(String[] args) {
  (new Thread1()).start();
  (new Thread2()).start();
  (new Thread3()).start();
  (new Thread4()).start();
 }
}
```

zu Aufgabe 3A.2.13. (Ritter / Diener / Hund):

```
import java.util.concurrent.*;
class Ritter extends Thread {
 public void run() {
  try {
   System.out.println("Ritter: Ich warte auf Wein");
```

```
   Prog.s_wein.acquire();
   System.out.println("Ritter: Ich habe den Wein");
   for (int i=0;i<5;i++) {
    System.out.println("Ritter: Ich will ein Kotelett");
    Prog.s_kotelett.acquire();
    System.out.println("Ritter: Ich esse ein Kotelett");
    sleep(2000);
   }
   System.out.println("Ritter: Ich werfe die Knochen");
   Prog.s_knochen.release();
   System.out.println("Ritter: Ich gehe zum Sessel");
   sleep(2000);
   Prog.s_sessel.acquire();
   System.out.println("Ritter: Ich sitze im Sessel");
   sleep(4000);
   System.out.println("Ritter: Ich stehe vom Sessel auf");
   Prog.s_sessel.release();
  } catch (InterruptedException e) {}
 }
}
class Diener extends Thread {
 public void run() {
  try {
   sleep(2000);
   System.out.println("  Diener: Ich bringe den Wein");
   Prog.s_wein.release();
   sleep(2000);
   System.out.println("  Diener: Ich brate ein Kotelett");
   sleep(4000);
   System.out.println("  Diener: Ich bringe das Kotelett");
   Prog.s_kotelett.release();
   System.out.println("  Diener: Ich brate ein Kotelett");
   sleep(4000);
   System.out.println("  Diener: Ich bringe das Kotelett");
   Prog.s_kotelett.release();
  } catch (InterruptedException e) {}
 }
}
class Hund extends Thread {
 public void run() {
  try {
   System.out.println("    Hund: Ich will Knochen");
   Prog.s_knochen.acquire();
   System.out.println("    Hund: Ich habe die Knochen");
   sleep(2000);
```

```
   System.out.println("    Hund: Ich laufe zum Sessel");
   Prog.s_sessel.acquire();
   System.out.println("    Hund: Ich liege im Sessel");
   sleep(4000);
   System.out.println("    Hund: Ich gebe den Sessel frei");
   Prog.s_sessel.release();
  } catch (InterruptedException e) {}
 }
}
public class Prog {
 static Semaphore s_wein = new Semaphore(0);
 static Semaphore s_kotelett = new Semaphore(3);
 static Semaphore s_knochen = new Semaphore(0);
 static Semaphore s_sessel = new Semaphore(1);
 public static void main(String[] args) {
  (new Ritter()).start();
  (new Diener()).start();
  (new Hund()).start();
 }
}
```

zu Aufgabe 3A.2.14. (Sleeping Barber):

```
import java.util.concurrent.*;
class Friseur extends Thread {
 public void run() {
  try {
   while (!isInterrupted()) {
   System.out.println("Friseur: Ich bin bereit");
   Prog.friseur_bereit.release();
   Prog.kunden_da.acquire();
   System.out.println("Friseur: Kunde kommt dran");
   sleep(2000);
   System.out.println("Friseur: Kunde ist fertig");
   Prog.fertig.release();
   }
   System.out.println("Friseur: Ich bin beendet worden");
  } catch (InterruptedException e) {}
 }
}
class Kunde extends Thread {
 public void run() {
  try {
   System.out.println("    Kunde "+getId()+": Ich moechte bedient werden");
   Prog.kunden_da.release();
   Prog.friseur_bereit.acquire();
   System.out.println("    Kunde "+getId()+": Ich komme dran");
```

```
   Prog.fertig.acquire();
   System.out.println("    Kunde "+getId()+": Ich bin fertig");
  } catch (InterruptedException e) {}
 }
}
public class Prog {
 static final int anz_kunden = 3;
 static Semaphore friseur_bereit = new Semaphore(0);
 static Semaphore kunden_da = new Semaphore(0);
 static Semaphore fertig = new Semaphore(0);
 public static void main(String[] args) {
  Friseur friseur = new Friseur();
  Kunde kunden[] = new Kunde[anz_kunden];
  friseur.start();
  for (int i=0;i<anz_kunden;i++) {
   kunden[i] = new Kunde();
   kunden[i].start();
  }
 }
}
```

```
2.)class Ausgabeobjekt {
  synchronized public void ausgabe(String text) {
   for (int i=0; i<3; i++) {
    try {
     System.out.println(text);
     Thread.currentThread().sleep(200);
    } catch (InterruptedException e) {  }
   }
  }
}
class BeispielThread extends Thread {
 private String ausgabetext;
 BeispielThread(String ausgabetext) {
  this.ausgabetext = ausgabetext;
 }
 public void run() {
   Prog.aobj.ausgabe(ausgabetext);
 }
}
public class Prog {
 static Ausgabeobjekt aobj = new Ausgabeobjekt();
 public static void main(String[] args) {
  (new BeispielThread("AAA")).start();
  (new BeispielThread("   BBB")).start();
 }
```

```
}
3.)class Ausgabeobjekt {
  int zaehler = 0;
  public void ausgabe(String text) {
   synchronized(this) {
     zaehler++;
     System.out.println("Anfang. Threads im Ausgabe-Objekt: "+zaehler);
   }
   for (int i=0; i<3; i++) {
    try {
     System.out.println(text);
     Thread.currentThread().sleep(200);
    } catch (InterruptedException e) {  }
   }
   synchronized(this) {
     zaehler--;
     System.out.println("Ende. Threads im Ausgabe-Objekt: "+zaehler);
   }
  }
}
4.)import java.io.*;
class Buntstift {}
class Anna extends Thread {
  public void run() {
    synchronized(Kindergarten.rot) {
     synchronized(Kindergarten.gruen) {
      System.out.println("Anna: Ich male den Apfelbaum (rot und grün)");
      try { this.sleep(1000); } catch (InterruptedException e) {}
      System.out.println("Anna: Apfelbaum ist fertig");
     }
    }
    synchronized(Kindergarten.blau) {
      System.out.println("Anna: Ich male den Himmel (blau)");
      try { this.sleep(1000); } catch (InterruptedException e) {}
      System.out.println("Anna: Himmel ist fertig");
    }
  }
}
class Bolle extends Thread {
  public void run() {
    synchronized(Kindergarten.rot) {
     synchronized(Kindergarten.gruen) {
      synchronized(Kindergarten.blau) {
       System.out.println("Bolle: Ich male das Haus (rot, grün, blau)");
       try { this.sleep(1000); } catch (InterruptedException e) {}
```

```
      System.out.println("Bolle: Haus ist fertig");
     }
    }
   }
   synchronized(Kindergarten.rot) {
    System.out.println("Bolle: Ich male das Feuerwehrauto (rot)");
    try { this.sleep(1000); } catch (InterruptedException e) {}
    System.out.println("Bolle: Feuerwehrauto ist fertig");
   }
  }
}
class Christina extends Thread {
 public void run() {
  synchronized(Kindergarten.gruen) {
   System.out.println("Christina: Ich male abstrakt (grün)");
   try { this.sleep(1000); } catch (InterruptedException e) {}
   System.out.println("Christina: Grün ist fertig");
  }
  synchronized(Kindergarten.blau) {
   System.out.println("Christina: Ich male abstrakt (blau)");
   try { this.sleep(1000); } catch (InterruptedException e) {}
   System.out.println("Christina: Blau ist fertig");
  }
  synchronized(Kindergarten.rot) {
   System.out.println("Christina: Ich male abstrakt (rot)");
   try { this.sleep(1000); } catch (InterruptedException e) {}
   System.out.println("Christina: Rot ist fertig");
  }
 }
}
class Kindergarten {
 static Buntstift rot, gruen, blau;
 public static void main(String args[]) {
  rot = new Buntstift();
  gruen = new Buntstift();
  blau = new Buntstift();
  (new Anna()).start();
  (new Bolle()).start();
  (new Christina()).start();
 }
}
```

Deadlock:

```
class Anna extends Thread {
 public void run() {
  synchronized(Kindergarten.rot) {
```

```
      try { this.sleep(1000); } catch (InterruptedException e) {}
     synchronized(Kindergarten.gruen) {
     ...
class Bolle extends Thread {
  public void run() {
    synchronized(Kindergarten.gruen) {
      try { this.sleep(1000); } catch (InterruptedException e) {}
     synchronized(Kindergarten.rot) {
     ...
5.)/* Barriere, an der Threads ggf. blockiert werden. */
class Barriere {
 final int zuErreichendeAnzahl = 5;
  /* gibt die Zahl wartender Threads an, bei der entblockiert wird */
 int wartendeThreads = 0;
  /* gibt Anzahl der wartenden Threads an */
 /* Methode, mit der Threads der Wunsch anmelden, die Barriere zu passieren */
 synchronized void anfragePassieren() {
  wartendeThreads++;
  if (wartendeThreads<zuErreichendeAnzahl) {
   try {
    System.out.println("Thread blockiert sich. Anzahl wartender Threads: "+wartendeThreads);
    wait();
   } catch (InterruptedException e) {}
  }
  else {
    System.out.println("Threads werden entblockiert");
    notifyAll();
  }
  System.out.println("Thread läuft weiter");
 }
}
/* Threads zum Test der Barriere */
class BarriereTestThread extends Thread {
 public void run() {
  WaitNotifyAll.barriere.anfragePassieren();
 }
}
/* Klasse für das Hauptprogramm, in dem Threads zum Test der Barriere gestartet werden */
public class WaitNotifyAll {
 static Barriere barriere;
 public static void main(String[] args) {
  barriere = new Barriere();
  for (int i=0;i<barriere.zuErreichendeAnzahl;i++) {
   try {
    (new BarriereTestThread()).start();
```

```
   Thread.currentThread().sleep(1000);
  } catch (InterruptedException e) {}
 }
}
}
```

Lösung mit notify() statt notifyAll():

```
synchronized void anfragePassieren() {
 ...
 if (wartendeThreads<zuErreichendeAnzahl) {
  ... }
  else System.out.println("Threads werden entblockiert");
 System.out.println("Thread läuft weiter");
 notify();
}
```

6.)

```
/* Barriere, an der Threads ggf. blockiert werden. */
import java.util.*;
/* Barriere, an der Threads ggf. blockiert werden. */
class Barriere {
 final int zuErreichendeAnzahl = 5;
  /* gibt die Zahl wartender Threads an, bei der entblockiert wird */
 LinkedList<Thread> wartendeThreads = new LinkedList<Thread>();
  /* Threads, die an dieser Barriere warten (sortiert nach FIFO) */
 Thread firstThread;
 /* Hilfsvariable: Der entblockierende Thread weist dieser Variablen eine Referenz auf den ersten Thread
    in 'wartendeThreads' zu und entfernt diese Referenz dabei aus 'wartendeThreads'.
    Die entblockierten Threads prüfen anhand dieser Variablen, ob sie weiterlaufen dürfen
    oder sich wieder blockieren müssen. */
 /* Methode, mit der Threads der Wunsch anmelden, die Barriere zu passieren */
 synchronized void anfragePassieren() {
  wartendeThreads.add(Thread.currentThread());
  if (wartendeThreads.size()<zuErreichendeAnzahl) {
   System.out.println("Thread "+Thread.currentThread().getId()+" blockiert sich.
                                                Anzahl wartender Threads: "+wartendeThreads.size());
   try { wait(); } catch (InterruptedException e) {}
  }
  else {
   System.out.println("Threads werden entblockiert");
   firstThread = wartendeThreads.remove();
   notifyAll();
   try { wait(); } catch (InterruptedException e) {}
  }
  while (firstThread!=Thread.currentThread())
    // wieder blockieren, wenn ein anderer Thread länger gewartet hat
   try { wait(); } catch (InterruptedException e) {}
  System.out.println("Thread "+Thread.currentThread().getId()+" läuft weiter");
```

```
 }
}
/* Threads zum Test der Barriere */
class BarriereTestThread extends Thread {
 public void run() {
  Prog.barriere.anfragePassieren();
 }
}
/* Klasse für das Hauptprogramm, in dem Threads zum Test der Barriere gestartet werden */
public class Prog {
 static Barriere barriere;
 public static void main(String[] args) {
  barriere = new Barriere();
  for (int i=0;i<barriere.zuErreichendeAnzahl+3;i++) {
   try {
    (new BarriereTestThread()).start();
    Thread.currentThread().sleep(1000);
   } catch (InterruptedException e) {}
  }
 }
}
```

```
7.)import java.util.concurrent.atomic.AtomicInteger;
// Klasse für Threads, die die Inkrementierungen durchführen
class InkrThread extends Thread {
 public void run() {
  while (!isInterrupted()) {
    AtomicTest.i++;
    AtomicTest.ati.getAndIncrement();
  }
  }
}
// Klasse für das Hauptprogramm
public class AtomicTest {
 // Variablen, die inkrementiert werden sollen
 static int i = 0;
 static AtomicInteger ati = new AtomicInteger(0);
 // Hauptprogramm
 public static void main(String[] args) {
  Thread t1 = new InkrThread();
  Thread t2 = new InkrThread();
  t1.start();
  t2.start();

  try {
   Thread.currentThread().sleep(4000);
```

```
 } catch (InterruptedException e) {}
 t1.interrupt();
 t2.interrupt();
 System.out.println("int-Wert:          "+i);
 System.out.println("AtomicInteger-Wert: "+ati.get());
 System.out.println("Differenz:         "+(ati.get()-i));
 }
}
```

Es zeigt sich, dass am Ende die int-Variable einen niedrigeren Wert als die AtomicInteger-Variable hat. Dies ist darauf zurückzuführen, dass der Zugriff auf die int-Variable nicht atomar ist. Offensichtlich haben die beiden Threads manchmal die Variable (fast) gleichzeitig gelesen, denselben Wert inkrementiert und wieder in die Variable zurückgeschrieben. Insgesamt hat dann nur eine Inkrementierung stattgefunden und nicht zwei.

8.)

```
import java.util.concurrent.locks.*;
/* Klasse für Stacks */
class Stack {
 private int kap;              // Kapazität des Stacks
 private int inhalt[];         // Wertspeicher des Stacks
 private int spitze;           // Index des Eintrags an der Kellerspitze
 private Lock lock;            // Lock für den wechselseitigen Ausschluss des Zugriffs
 private Condition voll, leer;  // Bedingungsvariablen
 Stack(int kap) {
  this.kap = kap;
  inhalt = new int[kap];
  spitze = -1;
  lock = new ReentrantLock();
  voll = lock.newCondition();
  leer = lock.newCondition();
 }
 public void schreiben(int neu) {
  lock.lock();
  try {
   while (spitze==kap-1)
     voll.await();
   inhalt[++spitze]=neu;
   leer.signal();
  } catch (InterruptedException e) {}
    finally { lock.unlock(); }
 }
 public int lesen() {
  int erg=-1;
  lock.lock();
  try {
   while (spitze==-1)
    leer.await();
```

```
    erg = inhalt[spitze--];
    voll.signal();
   } catch (InterruptedException e) {}
     finally { lock.unlock(); }
   return erg;
  }
 }
/* Erzeuger-Thread: schreibt Werte in den Stack */
class Erzeuger extends Thread {
 private Stack stack;  // Stack, in den der Erzeuger schreibt
 private int erzdauer; // Zeitdauer eines Erzeuge-Vorgangs
 private int wert;     // zu schreibender Wert
 Erzeuger(Stack stack, int erzdauer) {
   this.stack = stack;
   this.erzdauer = erzdauer;
   wert = 10;
 }
 public void run() {
   while (true)
    try {
      sleep(erzdauer);
      System.out.println("Erzeuger: Wert ist erzeugt");
      stack.schreiben(wert);
      System.out.println("Erzeuger: Wert "+wert+" ist geschrieben");
      wert += 10;
    } catch (InterruptedException e) {}
 }
}
/* Verbraucher-Thread: liest Werte aus dem Stack */
class Verbraucher extends Thread {
 private Stack stack;   // Stack, aus dem der Verbraucher liest
 private int verbdauer; // Zeitdauer eines Verbrauchs-Vorgangs
 Verbraucher(Stack stack, int verbdauer) {
   this.stack = stack;
   this.verbdauer = verbdauer;
 }
 public void run() {
   while (true)
    try {
      System.out.println("    Verbraucher: Ich will lesen");
      int gelesen = stack.lesen();
      System.out.println("    Verbraucher: Ich habe "+gelesen+" gelesen");
      sleep(verbdauer);
    } catch (InterruptedException e) {}
 }
```

```
}
/* Klasse für das Hauptprogramm, in dem Threads zum Test des Stacks gestartet werden */
public class LockCondition {
 public static void main(String[] args) {
  Stack stack = new Stack(5);
  (new Erzeuger(stack,1000)).start();
  (new Verbraucher(stack,5000)).start();
 }
}
```