

Musterlösungen zu den Hausaufgaben auf
Blatt 6
der Übungen zur Vorlesung
“Grundlagen Betriebssysteme und Systemsoftware

G.Groh, 20.11.2008

Aufgabe 1

Java-Klasse für Semaphore:

```
public class Semaphor {
    private int count;

    // Konstruktor
    public Semaphor (int init) {
        count = init;
    }

    public synchronized void P() {
        while (count == 0) {
            try {
                wait();
            }
            catch (InterruptedException E) {}
        }

        count --;
    }

    public synchronized void V() {
        count ++;
        notify();
    }
}
```

Aufgabe 1

Das Schlüsselwort `synchronized` stellt sicher, dass immer nur ein Thread eine P- bzw. V-Operation gleichzeitig aufrufen kann. Damit wird also der wechselseitige Ausschluss beim Zugriff auf die Variable `count` sichergestellt. Die Variable `count`, der eigentliche Semaphor, sollte als `private` deklariert werden, um eine Manipulation ohne Aufruf der P- oder V-Operation auszuschließen.

Die Methode `wait()` setzt den aufrufenden Thread auf wartend. Dies vermeidet ein "busy waiting". Ausserdem wird dabei der durch das `synchronized` definierte exklusive Zugriff auf das Objekt wieder aufgehoben, so dass ein anderer Thread `notify()` aufrufen kann. Mit `notifyAll()` werden alle bzgl. dieses Objekts wartender Threads "aufgeweckt". Mit `notify()` würde ein beliebiger (zufällig bestimmter) bzgl. dieses Objekts wartender Thread "aufgeweckt". Diese Auswahl ist nicht-deterministisch, so dass es zu einem Aushungern von Threads kommen kann. Die `while`-Schleife dient dazu, um einen Fehler zu vermeiden, wenn der wartende Thread durch eine Ausnahme aufgeweckt wird, aber der Semaphor noch belegt ist.

Aufgabe 2

Die Druckerklasse:

```
class Drucker {  
    private int seitenZahl;  
    private int breite, laenge;  
    private int aktZeile, aktSpalte;  
  
    public Drucker() {  
        seitenZahl = 1;  
        aktZeile   = 1;  
        aktSpalte  = 1;  
  
        breite = 30;  
        laenge = 10;  
    }  
  
    public synchronized void setzeZeilenBreite( int Breite ) {  
        breite = Breite;  
    }  
  
    public synchronized void setzeAnzahlZeilen( int Laenge ) {  
        laenge = Laenge;  
    }  
  
    public synchronized void druckeZeichen( String EinZeichen ) {  
        if (aktSpalte==1)  
            System.out.print( seitenZahl +": " );  
  
        System.out.print( EinZeichen );  
  
        aktSpalte++;  
    }  
}
```

Aufgabe 2

```
public synchronized void setzeAnzahlZeilen( int Laenge ) {
    laenge = Laenge;
}

public synchronized void druckeZeichen( String EinZeichen ) {
    if (aktSpalte==1)
        System.out.print( seitenZahl +": " );

    System.out.print( EinZeichen );

    aktSpalte++;
    if (aktSpalte > breite) {
        neueZeile();
    }
}

public synchronized void neueZeile() {
    System.out.print("\n");

    aktSpalte= 1;

    aktZeile++;
    if (aktZeile > laenge) {
        neueSeite();
    }
}

public synchronized void neueSeite() {
    System.out.print("\n");

    System.out.print( Thread.currentThread() + "\n");
    seitenZahl++;
    aktZeile=1;
    aktSpalte=1;
}
```

Aufgabe 2

Die Testklasse (ein Thread):

```
class TestDruck {
    Drucker drucker;

    public static void main( String[] args ) {
        TestDruck oDruck;

        oDruck = new TestDruck();
        oDruck.teste();
    }

    public TestDruck() {
        drucker = new Drucker();
    }

    public void teste() {
        for( int j=1; j<=3; j++ ) {
            drucker.neueSeite();

            for( int i=0; i<100; i++ )
                drucker.druckeZeichen( ""+i%10 ) ;
        }
    }
}
```

Aufgabe 2

Die Testklasse (drei Threads):

```
class TestDruck implements Runnable {
    Drucker drucker;

    public static void main( String[] args ) {
        TestDruck oDruck;

        oDruck = new TestDruck();
        oDruck.testeThreads();
    }

    public TestDruck() {
        drucker = new Drucker();
    }

    public void teste() {
        synchronized(drucker) {
            for( int j=1; j<=3; j++ ) {
                for( int i=1; i<=100; i++ )
                    drucker.druckeZeichen( ""+i%10 ) ;

                drucker.neueSeite();
            }
        }
    }
}
```

Aufgabe 2

```
public void run( ) {
    teste();
}

public void testeThreads() {
    Thread thread1, thread2, thread3;

    thread1 = new Thread( this );
    thread2 = new Thread( this );
    thread3 = new Thread( this );

    thread1.start();
    thread2.start();
    thread3.start();
}
}
```

Konfliktsituation Typ 1:

Ein Kunde storniert einen Flug, ein anderer bucht den selben Flug. Sind die Zugriffe auf die Variable für diesen Flug nicht synchronisiert, kann es passieren, dass sich eine der beiden Operationen nicht im Wert der Variable niederschlägt. Wenn beide Operationen zuerst gleichzeitig den Wert lesen, dann beide auf dem identischen gelesenen Zahlenwert operieren und dann beide das jeweilige Ergebnis zurückschreiben, wird die erste Änderung von der zweiten überschrieben. Dieses Problem kann auch auftreten, wenn zweimal eine Buchung erfolgt. Dies ist ein Beispiel für Inkonsistenz der Variablen.

Konfliktsituation Typ 2:

Es gelte folgende Belegung:

```
int FreiFlugCB = 1
int FreiHotelB2 = 1
```

Zwei Kunden K1 und K2 fragen nun parallel einen Flug von C nach B und das Hotel B2 an. Da beide die Aussage erhalten, der Flug und das Hotel sei frei, bucht K1 zuerst den Flug, K2 zuerst das Hotel. Nun kommt es zu einer Verklemmung.

Aufgabe 3

Konflikte vom Typ eins können auch durch Monitore gelöst werden, wenn man die Variablen isoliert betrachtet. Konflikte vom Typ zwei entstehen, obwohl man Konflikte vom Typ eins ausgeschlossen hat. Diese Konflikte entstehen durch die semantische Kopplung der Variablen. Ein Hotel nutzt nichts ohne Flug und umgekehrt. Das Schützen des Zugriff auf eine dieser Variablen bringt also nichts. Das Monitorkonzept alleine reicht hier so nicht.

Die Variablen `FreiFlugAB`, `FreiFlugCB`, `FreiHotelB1`, `FreiHotelB2`, `FreiHotelB3` gehören alle in den Monitor. Damit gehören auch alle Zugriffsmethoden auf diese Variablen in den Monitor. Damit kann ein Konflikt vom Typ zwei aber noch nicht verhindert werden. Damit bleibt aber für diese Implementierung des Reservierungssystems keinerlei Parallelität. Evtl. kann man die Anfragen aus dem Monitor nehmen. Man akzeptiert damit aber, dass die gelieferten Werte evtl. nicht korrekt sind. Somit darf man sich auf diese Methode auch im weiteren Programmcode nicht abstützen. Damit wären aber wenigstens parallele Anfragen möglich.

Aufgabe 3

```
class Reservierung {
    private int freiHotelB1 = 12;

    public synchronized boolean buchenHotelB1( ) {
        boolean erfolg = false;

        if ( freiHotelB1 > 0 ) {
            freiHotelB1--;
            erfolg = true;
        }

        return erfolg;
    }

    public synchronized boolean stornierenHotelB1( ) {
        boolean erfolg = false;

        if ( freiHotelB1 < 12 ) {
            freiHotelB1++;
            erfolg = true;
        }

        return erfolg;
    }
}
```

Aufgabe 3

```
class Kundenberatung {
    private Reservierung reservierung;
    ...

    public void abschlussCB2( ) {
        ...
        synchronized( reservierung ) {
            reservierung.buchenFlugCB( );
            reservierung.buchenHotelB2( );
        }
        ...
    }
}
```