Ludwig-Maximilians-Universität München
Institut für Informatik
Lehrstuhl für Mobile und Verteilte Systeme
Prof. Dr. Claudia Linnhoff-Popien

mobile and distributed systems group

Übungsblatt 10
Betriebssysteme im WiSe 2020/2021

Zu den Modulen L, M

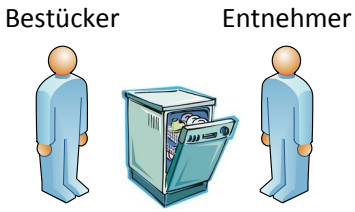
Abgabetermin: 24.01.2020, 18:59 Uhr

Besprechung: Besprechung der Übungsaufgaben in den Übungsgruppen vom 25. – 29. Januar 2021

Aufgabe Ü25: Semaphore

(13 Pkt.)

In dieser Aufgabe sollen Sie das Konzept der Semaphore am Beispiel einer Spülmaschine umsetzen. Zu besseren Arbeitsteilung gibt es eine Person, welche die Spülmaschine mit Geschirr bestückt und eine, die es wieder entnimmt, nachdem die Maschine gelaufen ist. Der genaue Ablauf ist wie folgt: Der „Bestücker“ befüllt die Spülmaschine mit Geschirr. Danach muss die Spülmaschine einen Spülgang durchführen. Sie wird erst aktiv, nachdem sie vollständig beladen wurde, was durch den „Bestücker“ auslöst werden muss. Ist der Spülgang abgeschlossen, muss dies dem „Entnehmer“ signalisiert werden, damit dieser wiederum das Geschirr entnimmt. Erst nachdem das komplette Geschirr entnommen wurde, kann der „Bestücker“ wieder tätig werden. Es sollen keine einzelnen Geschirrstücke modelliert werden, sondern der „Bestücker“ und der „Entnehmer“ be- bzw. entladen die Maschine immer komplett, was sie entsprechend signalisieren sollen. Zu Beginn ist die Spülmaschine leer. Die nachfolgende Abbildung zeigt den schematischen Aufbau.



Bearbeiten Sie nun auf Grundlage dieses Szenarios folgende Aufgaben:

a. Tragen Sie in folgende Tabelle die aus Ihrer Sicht benötigten Semaphoren ein, um den beschrieben Sachverhalt zu synchronisieren. Geben Sie zu jedem von Ihnen angedachten Semaphor einen Bezeichner und eine kurze Beschreibung, wofür er verwendet werden soll, an.

Bezeichner	Zweck	

b. Geben Sie in Pseudocode an, wie die benötigten Semaphore initialisiert werden müssen. Verwenden Sie dabei die Notation, die bei den gegebenen Semaphoren verwendet wurde:

Pseudocode	Bedeutung

c. Vervollständigen Sie nun den folgenden Pseudocode für den „Bestücker“, den Geschirrspüler und den „Entnehmer“, so dass der Ablauf wie beschrieben synchronisiert wird. Der „Bestücker“ soll nur tätig werden, wenn die Maschine leer ist (was zu Beginn der Fall ist). Dem Geschirrspüler soll der Start des Spülens signalisiert werden. Nachdem diese Fertig ist, soll der „Entnehmer“ tätig werden, der die Maschine komplett entlädt, wonach der „Bestücker“ wieder tätig werden kann.

```
Bestücker() {
      while(true) {
            //auf leeren Geschirrspüler warten



            <Geschirrspüler mit Geschirr bestücken>


            //Geschirrspüler den Start signalisieren



      }
}
```

```
Geschirrspüler() {
      while(true) {


            //auf Signal zum Start warten


            <Geschirr spülen>




           //Fertigstellung signalisieren


      }
}
```

```
Entnehmer() {
      while(true) {


            //auf Geschirrspüler warten


             <den Geschirrspüler entladen>;


            //leeren Geschirrspüler signalisieren


       }
}
```

Verwenden Sie für den Zugriff auf Ihre Semaphore folgende Notation:

Pseudocode	Beispiel	Bedeutung
`wait(<semaphor>);`	`wait(mutex);`	Verringert den Wert des Semaphor `mutex` um eins
`signal(<semaphor>);`	`signal(mutex);`	Erhöht den Wert des Semaphor `mutex` um eins
`<Aktion>`	`<den Geschirrspüler bestücken>`	Führe die gelistete Aktion aus
`//Kommentar`	`//Start signalisieren`	Kommentarzeile

Aufgabe Ü26: Threads/Monitore in Java

(15 Pkt.)

In dieser Aufgabe soll der Arbeitsablauf auf einer stilisierten Apfelplantage in Java mit dessen Implementierung des Monitor-Konzepts simuliert werden. Auf der Apfelplantage arbeiten 2 Feldarbeiter und ein Koch. Feldarbeiter müssen essen, um Äpfel pflücken zu können. Eine Portion Apfelmus reicht aus, damit sich ein Feldarbeiter genug gestärkt fühlt, um 2 Äpfel pflücken zu können (außer Apfelmus gibt es nichts zu essen). Vor dem Pflücken muss ein Arbeiter satt sein. Nach dem Pflücken ist der Feldarbeiter wieder hungrig. Der Koch kann aus 12 Äpfeln 8 Portionen Apfelmus kochen. Er beginnt immer erst dann mit dem Kochen, wenn er mindestens 12 Äpfel hat. Um die schwere Kocharbeit verrichten zu können, muss sich der Koch selbst zuvor auch mit einer Portion Apfelmus stärken. Danach ist er wieder hungrig. Zu Beginn sind sowohl der Koch als auch die Feldarbeiter hungrig. Die Anzahl der Äpfel bzw. die Anzahl der Portionen Apfelmus im Lager darf in Ihrer Implementierung nicht unter 0 fallen.
Im Folgenden soll die Klasse `Lager` implementiert werden. Die Beispielimplementierungen der Klassen `Apfelplantage`, `Koch` und `Feldarbeiter` sollen Ihnen verdeutlichen, wie die Klasse `Lager` verwendet werden kann:

Die Klasse `Apfelplantage`:

```
public class Apfelplantage {
    public final static int Aepfel = 6;
    public final static int Apfelmus = 3;

    public static void main(String[] args) {
```

```
6           Lager lager = new Lager(Aepfel, Apfelmus);
7
8           Koch koch = new Koch(-1, lager);
9           koch.start();
10
11          for (int i = 0; i < 2; i++) {
12              Feldarbeiter arbeiter = new Feldarbeiter(i, lager);
13              arbeiter.start();
14          }
15      }
16  }
```

Die Klasse `Koch`:

```
1  public class Koch extends Thread {
2      private int id;
3      private Lager lager;
4
5      public Koch(int id, Lager lager) {
6          this.id = id;
7          this.lager = lager;
8      }
9
10     public void run() {
11         while (true) {
12             try {
13                 lager.apfelmusEntnehmen(id, 1);
14                 lager.aepfelEntnehmen(12);
15                 System.out.println("Koch kocht");
16                 Thread.sleep(1000);
17                 lager.apfelmusEinlagern(8);
18             } catch (InterruptedException e) {
19                 e.printStackTrace();
20             }
21         }
22     }
23 }
```

Die Klasse `Feldarbeiter`:

```
1  public class Feldarbeiter extends Thread {
2      private int id;
3      private Lager lager;
4
5      public Feldarbeiter(int id, Lager lager) {
6          this.id = id;
7          this.lager = lager;
8      }
9
10     public void run() {
11         while (true) {
12             try {
13                 lager.apfelmusEntnehmen(id, 1);
14
15                 // Arbeiter pflückt Äpfel
16                 Thread.sleep(1000);
17
18                 lager.aepfelEinlagern(2);
19
20             } catch (InterruptedException e) {
```

```
                    e.printStackTrace();
                }
            }
        }
    }
```

Bearbeiten Sie die folgenden Aufgaben.

a. Implementieren Sie den Konstruktor der Klasse `Lager`. Die Klassenattribute sind dort bereits deklariert und sollen durch den Konstruktor explizit initialisiert werden.

b. Implementieren Sie die Methode `apfelmusEntnehmen(int id, int anzahl)`, welche das Entnehmen von Apfelmus aus dem Lager zum Verzehr modelliert.

c. Implementieren Sie die Methode `aepfelEntnehmen(int anzahl)`, welche das Entnehmen von Äpfeln aus dem Lager durch den Koch modelliert.

d. Vervollständigen Sie nun die Methode `apfelmusEinlagern(int anzahl)`. Auch hier sollten die oben genannten Anforderungen Beachtung finden.

e. In der Methode `aepfelEinlagern(int anzahl)`, die vom Feldarbeiter aufgerufen wird, ist bereits der Aufruf des Befehls *notifyAll()* vorgegeben. Welche andere Semantik hat der *notify()* Befehl im Vergleich dazu und welche Änderungen im Ablauf würden sich potenziell ergeben, wenn *notify()* anstelle von *notifyAll()* aufgerufen werden würde.

Der folgende Rahmen steht Ihnen zur Bearbeitung der Aufgaben a), b), c und d) zur Verfügung:

```
public class Lager {
    private int aepfel;
    private int apfelmus;

    public Lager(int aepfel, int apfelmus) {




    }

    public synchronized void aepfelEinlagern(int anzahl) {
        aepfel = aepfel + anzahl;
        System.out.println(anzahl + " Aepfel eingelagert, Anz. Aepfel:" + aepfel);
        notifyAll();
    }

    public synchronized void apfelmusEntnehmen(int id, int anzahl) throws InterruptedException
        while (                                                          ) {
            System.out.println("Identitaet " + id +
                                    " muss warten. Anz. apfelmus:" + apfelmus);




        }
        System.out.println(anzahl + " Apfelmus entnommen von " + id +
                                    " , Anz. Apfelmus:" + apfelmus);



    }

```

```
34      public synchronized void aepfelEntnehmen(int anzahl) throws InterruptedException {
35          while (                                                    ) {
36              System.out.println("Koch muss warten. Anz. Aepfel:" + aepfel);
37
38
39
40
41          }
42          System.out.println(anzahl + " Aepfel entnommen, Anz. Aepfel:" + aepfel);
43
44
45
46
47
48
49
50      }
51
52      public synchronized void apfelmusEinlagern(int anzahl) {
53
54
55
56
57
58      }
59  }
```

Hinweis: Vergessen Sie nicht die Bearbeitung von Aufgabenteil e).

Aufgabe Ü27: Einfachauswahlaufgabe: Prozesskoordination

(5 Pkt.)

Für jede der folgenden Fragen ist eine korrekte Antwort auszuwählen („1 aus n“). Nennen Sie dazu in Ihrer Abgabe explizit die jeweils ausgewählte Antwortnummer ((i), (ii), (iii) oder (iv)). Eine korrekte Antwort ergibt jeweils einen Punkt. Mehrfache Antworten oder eine falsche Antwort werden mit 0 Punkten bewertet.

a) Wie betritt ein Prozess einen Monitor?			
(i) Durch den Zugriff auf **public** Variablen.	(ii) Durch den Zugriff auf bestimmte Monitorprozeduren.	(iii) Durch den Aufruf von *notify()*.	(iv) Durch den Aufruf von *notifyAll()*.
b) Welches Kommunikationsschema liegt vor, wenn der Sender nach dem Absenden mit seiner Ausführung fortfahren kann aber der Empfänger bis zum Erhalt einer Nachricht wartet.			
(i) Blocking Send, Blocking Receive.	(ii) Blocking Send, Nonblocking Receive.	(iii) Nonblocking Send, Blocking Receive.	(iv) Nonblocking Send, Nonblocking Receive.
c) Welcher der folgenden Betriebssystemmechanismen dient vorrangig der Prozesssynchronisation und weniger der Prozesskommunikation?			
(i) Message Queues	(ii) Shared Memory	(iii) Semaphore	(iv) Pipes
d) Wie bezeichnet man die Thread-Implementierung, bei der das Threadmanagement Aufgabe der jeweiligen Anwendung ist und der Betriebssystemkern keine Informationen über die Existenz solcher Threads hat bzw. haben muss?			
(i) User-Level-Threads	(ii) Kernel-Level-Threads	(iii) Hardware-Level-Threads	(iv) Low-Level-Threads

e) Wie bezeichnet man die Phase, in der sich zu einem Zeitpunkt nur ein Prozess befinden darf, da sich sonst z.B. inkonsistente Zustände bei gemeinsam genutzten Datenstrukturen ergeben?			
(i) einfacher Bereich	(ii) schwieriger Bereich	(iii) unkritischer Bereich	(iv) kritischer Bereich