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

mobile and distributed systems group

Betriebssysteme im Wintersemester 2015/2016
Übungsblatt 12

Abgabetermin: 25.01.2016, 16:00 Uhr

Besprechung: Besprechung der T-Aufgaben in den Tutorien vom 18. – 22. Januar
Besprechung der H-Aufgaben in den Tutorien vom 25. – 29. Januar

Ankündigungen: Die **Klausur** findet am **5. Februar 2016 von 18.30 - 20.30 Uhr** statt. Bitte melden Sie sich **bis spätestens 2. Februar 2015** zur Klausur über Uniworx an.

Des Weiteren können Sie bei Interesse unsere 2. Veranstaltung „Nebenläufigkeit mit Java“ zur Vertiefung der Java-Programmierung mit Threads, immer Montags von 18.00 - 20.00 Uhr im Hauptgebäude, Raum M010, besuchen. Weitere Informationen auf: `http://www.mobile.ifi.lmu.de/lehrveranstaltungen/nebenlaeufigkeit-mit-java`

Aufgabe 51: (T) Nachbildung eines Zählsemaphor in Java

(– Pkt.)

Schreiben Sie eine Java Klasse `SimpleCountingSemaphore`, die es ermöglicht, Instanzen zu erzeugen, welche in Analogie zu der von Dijkstra vorgeschlagenen Datenstruktur eines Zählsemaphor verwendet werden können. Verwenden Sie dazu den Java `synchronized`-Mechanismus. Verwenden Sie außerdem eine *minimale* Anzahl an `wait()`- und `notify()`-Aufrufen!

Hinweis: Für die Lösung dieser Aufgabe reicht es aus, wenn Sie das auftreten einer etwaigen `InterruptedException` ohne weitere Fehlerbehandlung abfangen.

Des Weiteren sollen Sie sich in dieser Aufgabe die Unterschiede und Gemeinsamkeiten zwischen Java-Synchronisation und Monitoren klar machen.

a. Beschreiben Sie das grundlegende Konzept der Synchronisation in Java. Gehen Sie dabei auf die Verwendung von Objekt-Locks, Threads und Warteschlangen ein.

b. Wie werden `wait()` und `signal()` in Java umgesetzt?

c. Erläutern Sie den grundlegenden Unterschied des Java Synchronisationsmechnismus im Gegensatz zu „echten“ Monitoren (ohne Beachtung der Signalisierungsmechanismen).

d. Beschreiben Sie die Einschränkungen bei der Verwendung von `wait()` und `notify()` gegenüber „echten“ Monitoren.

   Hinweis: Überlegen Sie sich dazu, wie im Falle echter Monitore bzw. im Falle des Java Synchronisierungsmechanismus der nächste aktive Thread selektiert wird.

e. Überlegen Sie sich, wie diese Einschränkungen umgangen werden können.

f. Es gibt zwei Modelle, wie die Ausführung in einem Monitor nach dem Aufruf von `signal()` fortfährt (A: signalisierender Prozess, B: aufgeweckter Prozeß):

- Signal-and-wait: A muß nach seiner Signalisierung den Monitor sofort freigeben und warten, bis B den Monitor verlassen hat oder auf eine andere Bedingung wartet.
- Signal-and-continue: B muß warten, bis A den Monitor verlassen hat oder auf eine andere Bedingung wartet.

Welchem Modell folgt Java?

Aufgabe 52: (T) Java: Koordination von Threads

(– Pkt.)

In dieser Aufgabe sollen Sie eine Lösung implementieren, die es ermöglicht, Züge koordiniert über einen **eingleisigen** Streckenabschnitt (AB) fahren zu lassen. Der Streckenabschnitt AB unterliegt folgenden Einschränkungen:

- Der Streckenabschnitt AB verfügt über genau ein Gleis, d.h. es kann gleichzeitig nur in genau eine Richtung gefahren werden (entweder West oder Ost).
- Es können sich maximal drei Züge gleichzeitig auf dem Streckenabschnitt befinden.
- Jeder Zug verlässt den Streckenabschnitt nach endlicher Zeit.

Die Klassen `TrainNet` und `Train` sind bereits gegeben. Sie können sich den Quelltext von der Website zur Vorlesung herunterladen.

Die Klasse `TrainNet` erzeugt einen Streckenabschnitt AB (Instanz der Klasse `RailAB` und startet die Züge (Instanzen der Klasse `Train`).

Die Klasse `Train` repräsentiert Züge und ist als Thread implementiert. Innerhalb der `run()`-Methode werden auf die Instanz der Klasse `RailAB` die Methoden `goEast()` und `goWest()` aufgerufen. Diese dienen dazu, einen Zug auf den Streckenabschnitt von West nach Ost bzw. von Ost nach West zu schicken. Zudem wird die Methode `leaveAB()` aufgerufen, durch deren Aufruf ein Zug den Streckenabschnitt AB wieder verlässt.

Implementieren Sie nun die Klasse `RailAB` unter Berücksichtigung der oben genannten Einschränkungen. Die Lösung muss frei von Deadlocks sein und darf Züge nicht unnötig blockieren. Implementieren Sie

a. einen passenden Konstruktor (siehe Klasse `TrainNet`),

b. die Methode `goWest()`, welche die Züge, die nach Westen fahren, koordiniert,

c. die Methode `goEast()`, welche die Züge, die nach Osten fahren, koordiniert, und

d. die Methode `leaveAB()`, welche von den Zügen aufgerufen wird, die den Streckenabschnitt wieder verlassen.

Aufgabe 53: (H) Synchronisation von Threads in Java

(11 Pkt.)

In dieser Aufgabe soll die schreiberfreundliche Variante des Leser-/Schreiberproblems in Java umgesetzt werden. Laden Sie sich bitte dazu zunächst von der Betriebssysteme-Homepage die Dateien `Vaterprozess.java`, `Prozess.java` und `Speicher.java` herunter.
Das Problem ist folgendermaßen definiert:

- Es gibt einen gemeinsamen Speicher *S*.
- Es gibt n Prozesse, die jeweils entweder *lesend* oder *schreibend* auf den gemeinsamen Speicher *S* zugreifen dürfen.

- Prozesse sind entweder Leserprozesse oder Schreiberprozesse.
- Schreiberprozesse müssen vor dem Schreiben vom Speicher *S* ein Schreibrecht erhalten. Leserprozesse benötigen ebenfalls ein entsprechendes Leserecht bevor sie aus dem Speicher *S* lesen dürfen.
- Sowohl Leser- als auch Schreiberprozesse geben das jeweilige Lese- bzw. Schreiberecht nach Abschluss des Lesens- bzw. Schreibens wieder frei.
- Besitzt ein Schreiberprozess das Schreiberecht, so darf währenddessen kein Leserprozess das Leserecht besitzen.
- Zu jedem Zeitpunkt darf nur maximal ein Schreiberprozess das Schreiberecht besitzen.
- Fragt ein Schreiberprozess das Schreiberecht an, so dürfen ab diesem Zeitpunkt keine Leserprozesse mehr mit dem Lesen beginnen.
- Die Prozesse werden von einem Vaterprozess erzeugt.

Im Folgenden soll die noch unvollständige Klasse `Speicher` aus der Datei `Speicher.java` fertig implementiert werden. Die Prozesse und der erzeugende Vaterprozess werden ebenfalls durch Java–Klassen und Threads simuliert. Die Beispielimplementierungen der Klassen `Vaterprozess` und `Prozess` soll verdeutlichen, wie die Klasse `Speicher` verwendet werden kann.
Bearbeiten Sie nun die folgenden Teilaufgaben unter der Berücksichtigung des oben definierten schreiberfreundlichen Leser-/Schreiberproblems:

a. Was versteht man allgemein unter einem kritischen Bereich?

b. Implementieren Sie den Konstruktor der Klasse `Speicher`. Ergänzen Sie dabei den Coderahmen aus der Datei `Speicher.java` und *kommentieren Sie Ihre Lösung ausführlich*! Achten Sie insbesondere auf die korrekte Intialisierung der bereits deklarierten Klassenattribute.

c. Implementieren Sie die Methode `leserechte_holen(int prozess_id)`, welche von Leserprozessen aufgerufen wird, um Leserechte zu erhalten. Implementieren Sie außerdem die Methode `leserechte_freigeben(int prozess_id)`, welche nach abgeschlossenem Lesevorgang aufgerufen wird, um die Leserechte wieder freizugeben. Ergänzen Sie dabei den Coderahmen aus der Datei `Speicher.java` und *kommentieren Sie Ihre Lösung ausführlich!*
Hinweis: Sie können davon ausgehen, dass die Methoden `leserechte_holen(int prozess_id)` bzw. `leserechte_freigeben(int prozess_id)` immer in einer sinnvollen Reihenfolge aufgerufen werden (siehe Beispielimplementierung der Klasse `Prozess`).

d. Implementieren Sie die Methode `schreibrecht_holen(int prozess_id)`, welche von Schreiberprozessen aufgerufen wird, um Schreibrechte zu erhalten. Implementieren Sie außerdem die Methode `schreibrecht_freigeben(int prozess_id)`, welche nach abgeschlossenem Schreibvorgang aufgerufen wird, um das Schreibrecht wieder freizugeben. Ergänzen Sie dabei den Coderahmen aus der Datei `Speicher.java` und *kommentieren Sie Ihre Lösung ausführlich!*
Hinweis: Sie können davon ausgehen, dass die Methoden `schreibrecht_holen(int prozess_id)` bzw. `schreibrecht_freigeben(int prozess_id)` immer in einer sinnvollen Reihenfolge aufgerufen werden (siehe Beispielimplementierung der Klasse `Prozess`).

e. Zeigen Sie zwei kritische Bereiche in ihrem Programm auf. Wie wird hier sichergestellt, dass die Bedingung der Mutual Exclusion erfüllt ist?

Aufgabe 54: (H) Einfachauswahlaufgabe: Prozesskoordination

(5 Pkt.)

Für jede der folgenden Fragen ist eine korrekte Antwort auszuwählen („1 aus n“). Eine korrekte Antwort ergibt jeweils einen Punkt. Mehrfache Antworten oder eine falsche Antwort werden mit 0 Punkten bewertet.

<table>
<tr><td colspan="4">a) Was ist keine der 3 atomaren Operationen, mit denen ein Semaphor <code>S</code> verändert werden kann?</td></tr>
<tr><td>(i) <code>init(S, Anfangswert)</code></td><td>(ii) <code>wait(S)</code> oder auch <code>P(S)</code></td><td>(iii) <code>block(S)</code> oder auch <code>B(S)</code></td><td>(iv) <code>signal(S)</code> oder auch <code>V(S)</code></td></tr>
<tr><td colspan="4">b) Angenommen ein Prozess möchte einen Semaphor, der einen kritischen Bereich schützt und dessen Wert bereits 0 beträgt weiter herabsetzen, um den kritischen Bereich zu betreten. Wodurch kann das busy waiting dieses Prozesses sinnvoll verhindert werden.</td></tr>
<tr><td colspan="4">(i) Durch das Beenden der Prozesse, die den Semaphor zuvor herabgesetzt haben.
(ii) Durch das Beenden des nun im busy waiting befindlichen Prozesses.
(iii) Indem jedem Prozess ohne Beachtung des Semaphors der Eintritt in den kritischen Bereich gewährt wird.
(iv) Durch das assoziieren des Semaphors mit einer Warteschlange und dem Suspendieren der wartenden Prozesse.</td></tr>
<tr><td colspan="4">c) Angenommen das Erzeuger/Verbraucher-Problem wurde mit Hilfe von Semaphoren gelöst. Dabei regelt ein binärer Semaphor <code>s</code> den wechselseitigen Zugriff auf den gemeinsamen Speicher, ein Zählsemaphor <code>b</code> repräsentiert die Anzahl der belegten Plätze und ein weiterer Zählsemaphor <code>p</code> gibt die Anzahl der freien Speicherplätze (wovon <code>MAX > 0</code> zur Verfügung stehen) an. Die Semaphoren werden wie folgt initialisiert: <code>init(s, 1); init(b, 0); init(p, MAX).</code>

Welche Aussage bezüglich des folgenden Quellcodes für den Erzeuger und den Verbraucher, welche parallel ausgeführt werden, ist korrekt?
<pre>(* Erzeuger: *)
REPEAT
  <erzeuge Element>;
  wait(s);
  wait(p);
  <Element in Speicher legen>;
  signal(s);
  signal(b);
UNTIL FALSE;</pre><pre>(* Verbraucher: *)
REPEAT
  wait(b);
  wait(s);
  <El. aus Speicher nehmen>;
  signal(s);
  signal(p);
  <verbrauche Element>;
UNTIL FALSE;</pre></td></tr>
<tr><td>(i) Die Lösung ist korrekt.</td><td>(ii) Es kann ein Deadlock entstehen.</td><td>(iii) Der Erzeuger kann niemals ein Element in den Speicher legen.</td><td>(iv) Der Verbraucher kann niemals ein Element aus dem Speicher nehmen.</td></tr>
<tr><td colspan="4">d) Was ist kein Bestandteil eines Monitors (Softwaremodul)?</td></tr>
<tr><td>(i) eine oder mehrere Prozeduren</td><td>(ii) lokale Daten</td><td>(iii) eine Warteschlange für ankommende Prozesse</td><td>(iv) eine Liste der Prozesse, die den Monitor verlassen haben</td></tr>
<tr><td colspan="4">e) Wie viele Prozesse dürfen sich zu jeder Zeit maximal in einem Monitor befinden?</td></tr>
<tr><td>(i) 0</td><td>(ii) 1</td><td>(iii) 2</td><td>(iv) 3</td></tr>
</table>