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 2017/2018
Übungsblatt 3

Abgabetermin: 13.11.2017, 18:00 Uhr

Besprechung: Besprechung der T-Aufgaben in den Tutorien vom 06. – 10. November 2017
Besprechung der H-Aufgaben in den Tutorien vom 13. – 17. November 2017

Aufgabe 11: (T) Einfache Boolesche Terme

(– Pkt.)

Gegeben sei die folgende Funktionstabelle von acht zweistelligen Booleschen Funktion $f_1, \ldots, f_8$.

A	B	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8
0	0	0	1	0	1	0	1	0	0
0	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	0	1
1	1	0	0	0	0	0	0	1	1

Schreiben Sie diese Funktionen als Boolesche Terme unter der Verwendung der Variable A bzw. falls erforderlich B. Verwenden Sie dazu ausschließlich UND ($\cdot$), ODER ($+$) und NICHT ($-$).

Aufgabe 12: (T) Unterprogramme und Stack

(– Pkt.)

In dieser Aufgabe soll das Zusammenspiel von Unterprogrammen und dem Stack untersucht werden. Ein Unterprogramm ist ein Programmstück, welches nur durch den Sprung an seine Anfangsadresse betreten und durch einen Rücksprung an die aufrufende Stelle beendet wird. Diese Art der Programmtechnik wird unter anderem zur Umsetzung rekursiver Aufrufe verwendet. Gegeben das folgende linear rekursive Unterprogramm `mult`:

```
1  function int mult(int a, int b){
2     if(a == 0) {
3        return 0;
4     }
5     else {
6        a = a - 1;
7        int c = mult(a, b);
8        int d = b + c;
9        return d;
10    }
11 }
```

Damit man Unterprogramme korrekt ausführen kann muss das Unterprogramm sowie das aufrufende Hauptprogramm bestimmte organisatorische Bedingungen erfüllen. Dazu kann unter anderem ein Stack verwendet werden.
Bearbeiten Sie in diesem Zusammenhang die folgenden Aufgaben:

a. Geben Sie eine Abfolge aller Unterprogrammaufrufe mit den entsprechenden Parametern an, die sich für den Aufruf von `mult(2,3)` ergeben.

b. Welche Zustandsinformationen müssen auf dem Stack gespeichert werden, damit das Hauptprogramm nach dem Unterprogrammaufruf korrekt fortgesetzt werden kann.

c. Geben Sie die aus der Vorlesung *Rechnerarchitekturen* bekannte vierstufige Aufrufkonvention an, die ein korrektes Zusammenarbeiten von Hauptprogramm und Unterprogramm mit Hilfe des Stacks gewährleistet. Geben Sie zu jedem der vier Aufrufe an, welche der in Aufgabe b) genannten Zustandsinformationen jeweils verarbeitet werden.

d. Gehen Sie im Folgenden davon aus, dass der CALL-Befehl für den Aufruf der Funktion `mult` an der Speicheradresse 1000 erfolgt und die Prozedur `mult` selbst an der Adresse 4000 beginnt. Skizzieren Sie nun den Stack mit den dazugehörigen Zustandsinformationen unmittelbar vor einem CALL, unmittelbar nach einem CALL und unmittelbar nach einem RET für den Aufruf von `mult(2,3)`. Geben Sie zudem an, welchen Teil der in Aufgabe c) erläuterten Aufrufkonventionen der gegenwärtige Stackzustand repräsentiert. Sie können davon ausgehen, dass die kleinste adressierbare Einheit auf dem Stack einem Wort (4 Byte) entspricht. Geben Sie in jedem Schritt stets den gesamten Inhalt des Stacks an.

Aufgabe 13: (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,

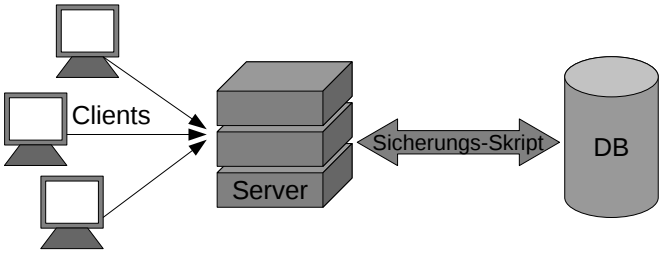
c. die Methode `goEast()`, welche die Züge, die nach Osten fahren, koordiniert, und

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

Aufgabe 14: (H) Threads in Java

(11 Pkt.)

In dieser Aufgabe soll eine einfache Client/Server Kommunikation in Java simuliert werden, bei welcher mehrere Clients Anfragen an eine Server-Schnittstelle stellen können, um dort Daten abzulegen. Die Daten werden in regelmäßigen Abständen von einem Sicherungs-Skript auf eine Datenbank geschrieben. Das beschriebene Szenario ist in folgender Abbildung schematisch dargestellt:



Aus Performanzgründen erlaubt der Server nur, dass eine maximale Anzahl von `maxClients` gleichzeitig Daten auf dem Server ablegen darf. Eine Client-Anfrage muss also ggf. mit dem Beginn der Datenablegung warten, damit diese Bedingung nicht verletzt wird.
Das Sicherungs-Skript wird regelmäßig aktiviert und speichert die abgelegten Daten zu einem dedizierten Zeitpunkt auf die Datenbank. Dazu meldet das Skript einen Sicherungswunsch am Server an, so dass kein weiterer Client mehr Daten ablegen darf. Anfragen können jedoch weiterhin gestellt werden und Clients, die zum Zeitpunkt des Sicherungswunsches bereits Daten ablegen, können natürlich ihre Aufgabe noch erledigen.
Das aktive Sicherungs-Skript muss solange warten, bis kein Client mehr Daten ablegt. Nach Beenden der Sicherung wird das Skript wieder deaktiviert und damit der Sicherwunsch aufgehoben, so dass die anfragenden Clients wieder Daten ablegen können.
Im Folgenden soll eine Klasse `Server` implementiert werden. Laden Sie sich bitte dazu zunächst von der Betriebssysteme-Homepage die Dateien `Simulation.java, Client.java, Sicherung.java` sowie den Code-Rahmen der Server-Klasse `ServerAbstract.java` herunter. Die Beispielimplementierungen der Klassen `Client`, `Sicherung` und `Simulation` sollen Ihnen verdeutlichen, wie die Klasse `Server` verwendet werden kann.
Bearbeiten Sie nun die folgenden Aufgaben:

a. Implementieren Sie den Konstruktor der Klasse `Server`. Verwenden Sie dabei den gegebenen Code-Rahmen! Die Klassenattribute sind dort bereits deklariert und müssen durch den Konstruktur initialisiert werden.

b. Implementieren Sie die Server-Methode `daten_ablegen(Client c)`, welche die Client Anfrage zum Daten ablegen modelliert, sowie die die Methode `daten_ablegen_beenden()`, welche vom Client aufgerufen wird, nachdem die Daten abgelegt wurden. Beachten Sie dazu die folgenden Randbedingungen:

   – Alle oben genannten Anforderungen müssen beachtet werden.
   – Maximal `maxClients` dürfen gleichzeitig Daten ablegen. Die Variable wird in der Klasse `Simulation` festgelegt.
   – Es darf kein neuer Client mehr Daten ablegen, sobald das Sicherungs-Skript den Sicherungswunsch geäußert hat. Neue Anfragen können aber weiterhin gestellt werden und noch laufende Daten-Ablegungen werden noch vollständig beendet.

   Ergänzen Sie dazu den Code-Rahmen am Ende der Aufgabe.
   Hinweis: Sie können davon ausgehen, dass die Methoden `daten_ablegen(Client c)` bzw. `daten_ablegen_beenden()` immer in einer sinnvollen Reihenfolge aufgerufen werden (siehe Beispielimplementierung der Klasse `Client`).

c. Implementieren Sie nun die Methoden für das Sicherungs-Skript. Vervollständigen Sie dazu den Code-Rahmen für die Methoden `sicherungAktivieren()` und `sicherungDeaktivieren()` in dem Code-Rahmen am Ende der Aufgabe.
Hinweis: Sie können davon ausgehen, dass die Methoden `sicherungAktivieren()` und `sicherungDeaktivieren()` immer in einer sinnvollen Reihenfolge aufgerufen werden (siehe Beispielimplementierung der Klasse `Sicherung`).

d. Zeigen Sie zwei kritische Bereiche in ihrem Programm auf. Wie wird hier sichergestellt, dass die Bedingung der wechselseitige Ausschluss erfüllt ist?

Aufgabe 15: (H) Einfachauswahlaufgabe: Programme und Unterprogramme

(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) Was stellt allgemein eine Schnittstelle zwischen Anwendungsprogrammen und Betriebssystem dar, durch die ein Anwendungsprogramm auf eine Ressource zugreifen kann, auf die es keinen direkten Zugriff hat?

(i) Systemaufrufe	(ii) Shared Memory	(iii) Sockets	(iv) Pipes

b) Welche Aussage zu offenen Unterprogrammen ist falsch?

(i) Der entsprech- ende Programmtext wird an den erforderlichen Stellen ins Hauptprogramm hineinkopiert.
(ii) Sie sind vor allem bei großen/langen Unterprogrammen effizient.
(iii) Nachträgliche Modifikationen am Unterprogramm müssen an jedem Vorkommen des Unterprogramms vorgenommen werden.
(iv) Die Speicheradressen z.B. für Sprungbefehle im Unterprogramm können bei jedem Vorkommen des Unterprogramms verschieden sein.

c) Welche Information wird zur Realisierung eines geschlossen Unterprogramms nicht explizit benötigt?

(i) Anfangsadresse des Unterprogramms	(ii) Aufrufparameter	(iii) Rücksprung-adresse zum Haupt-programm	(iv) Endadresse des Unterprogramms

d) Welche der folgenden Antworten zeigt eine korrekte Implementierung des `CALL`-Befehls?

(i)
```
COMMAND CALL addr
BEGIN
  PUSH (PC);
  PC := addr;
END
```

(ii)
```
COMMAND CALL addr
BEGIN
  PUSH (RA);
  PC := POP;
END
```

(iii)
```
COMMAND CALL addr
BEGIN
  RA := PC + 1;
  PC := addr;
END
```

(iv)
```
COMMAND CALL addr
BEGIN
  RA := PC + 1;
  PC := addr + 1;
END
```

e) Welche Aussage zu Threads ist falsch?

(i) Ein Prozess kann mehrere Threads enthalten.
(ii) Das reine Lesen gemeinsamer Daten durch mehrere Threads führt zu Inkonsistenzen.
(iii) Bei mehreren Threads, die um kritische Ressourcen konkurrieren, kann es zu Verklemmungen kommen.
(iv) Threads können echt parallel ausgeführt werden, sofern die Architektur (Betriebssystem, Hardware) es ermöglicht.