

Übung zur Vorlesung ”Grundlagen Betriebssysteme und Systemsoftware”

(Prof. Dr. J. Schlichter, WS 2008 / 2009)

Übungsleitung: Dr. Georg Groh (grohg@in.tum.de)

Tutoren: Dipl. Inform. Vivian Prinz (prinzv@in.tum.de), Dr. Nils Kammenhuber (kammenhuber@net.in.tum.de), Dipl. Inform. Robert Schmohl (schmohl@in.tum.de), Dipl. Inform. Dipl. Geogr. Jan Herrmann (hermanj@in.tum.de), Dipl. Inform. Robert Eigner, David Brodski (brodski@in.tum.de), Yang Guo (yang.guo@gmx.de), Jan Finis (finis@in.tum.de), Martin Levihn (levihn@in.tum.de)

<http://www11.in.tum.de/Veranstaltungen/GrundlagenBetriebssystemeundSystemsoftware0809>

<http://www11.in.tum.de/Veranstaltungen/GrundlagenBetriebssystemeundSystemsoftware0809/uebung>

Blatt 6

- Abgabe: bis 01.12.2008 12:00 Uhr per E-Mail an den Tutor der eigenen Gruppe. Die Mail soll einen Zip-Ordner als attachment haben, der für jede Hausaufgabe einen Unterordner enthält, in dem die Lösung als .txt-File(s), als .c-File(s) o.ä. enthalten ist.
- Musterlösungen Hausaufgaben: ab 01.12.2008 12:00 Uhr auf der Übungswebseite zum Download.
- Musterlösungen Tutoraufgaben: ab 08.12.2008 12:00 Uhr auf der Übungswebseite zum Download.

Stoff

Es wird empfohlen, die empfohlene Literatur von Blatt 5 durchzuarbeiten, sofern noch nicht geschehen. Insbesondere wichtig ist Abschnitt 2.3.7 ”Monitors”.

Weiterhin sei empfohlen folgende Literatur durchzuarbeiten:

- <http://java.sun.com/docs/books/tutorial/essential/concurrency/>

Da es in diesem Blatt um Monitore geht, und Java eine weitverbreitete Implementierung von Monitoren zur Thread-Synchronisation bereit stellt, ist dieser Teil des Java-Tutorials hilfreich. Diejenigen von Ihnen, die grundlegenden Aspekte von Java-Monitoren bereits in anderen LV kennengelernt haben, können sich mit den zahlreichen ab Java 1.5 hinzugekommenen High-Level Concurrency Klassen und Konzepten vertraut machen.

1 Hausaufgabe (Java-Monitore)

Lernziele

Kennenlernen bzw Wiederholen bzw Vertiefen der "Java-Variante" des Monitor-Konzepts.

Aufgabe

Machen Sie sich (bspw. anhand der empfohlenen Literatur) die folgenden Sachverhalte (in der Sprechweise von Java bzw des Java Tutorials) klar:

- den Unterschied zwischen `notify()` und `notifyAll()`
- den Unterschied zwischen Thread Interference und Memory Consistency Errors
- das Konzept der happenedBefore-Relation
- das Konzept des intrinsic Lock
- das Konzept des Guarded Block mit `wait()` und `notify()`
- Unterschiede und Gemeinsamkeiten zwischen **Guarded Block** mit `wait()` und `notify()` einerseits und **condition variables** mit `wait` und `signal` im Tanenbaum S 116 andererseits.

Entwerfen Sie eine Java-Klasse `Semaphor`, die Semaphore zur Synchronisation von Java-Threads bereitstellt. Realisieren Sie die P- und V-Operationen. Verwenden Sie die Methoden `wait()` und `notifyAll()`, um in einer P-Operation wartende Threads zu blockieren und wieder freizugeben. Ignorieren Sie dabei, dass eine Klasse `Semaphor` seit Java 1.5 bereits im SDK enthalten ist.

Abgabe

Programmtext als .java-Datei,

2 Hausaufgabe (Synchronisation mit Java-Monitoren)

Lernziele

Kennenlernen bzw Wiederholen bzw Vertiefen der "Java-Variante" des Monitor-Konzepts

Aufgabe

In der folgenden Aufgabe soll die Benutzung eines Druckers mit Hilfe von Java-Monitoren synchronisiert werden.

2.1 Teilaufgabe

Implementieren Sie eine Klasse Drucker in Java. Die Klasse biete unter anderem folgende Methoden:

```
public void setzeZeilenBreite( int Breite );
public void setzeAnzahlZeilen( int Laenge );
public void druckeZeichen( String EinZeichen );
public void neueZeile();
public void neueSeite();
```

Die Klasse besitze ausserdem einen Seitenzähler, der das aktuell bedruckte Blatt zählt. Der Zähler kann nur erhöht werden. Von aussen kann er weder gelesen noch geschrieben werden. Alle Ausgaben von Drucker erscheinen auf dem Bildschirm. Dabei wird an jedem Zeilenbeginn der Seitenzähler angezeigt. Eine neue Seite wird mit der Angabe des aktuellen Threads begonnen. Benutzen Sie dazu

```
System.out.println( Thread.currentThread() );
```

2.2 Teilaufgabe

Schreiben Sie eine Hauptklasse, die den Drucker testet, indem sie auf drei Seiten jeweils die Zahlen 0,1,2,...,9 schreiben, bis je Seite 100 Zeichen gedruckt sind.

2.3 Teilaufgabe

Schreiben Sie die Hauptklasse so um, dass drei Threads den angesprochenen Test parallel durchführen. Achten Sie darauf, dass auch die Benutzung durch mehrere parallele Threads sinnvolle Ausdrücke liefert.

Abgabe

Programmtexte als .java-Datei,

3 Hausaufgabe (Synchronisation mit Java-Monitoren 2)

Lernziele

Kennenlernen bzw Wiederholen bzw Vertiefen der "Java-Variante" des Monitor-Konzepts

Aufgabe

Betrachten wir im folgenden ein Reservierungssystem. Angenommen in diesem System werden Reservierungen für Flüge und Übernachtungen verwaltet. Folgendes Angebot wird über das Reservierungssystem abgewickelt:

	Kapazität
Flug A->B	10
Flug C->B	8
Hotel B1	12
Hotel B2	5
Hotel B3	9

Das Reservierungssystem kennt die Operationen **anfragen**, **buchen**, **stornieren**. Dabei betrifft eine Operation entweder einen Flug oder ein Hotel. Beides kombiniert wird nicht angeboten. Für einen kompletten Urlaub braucht ein Kunde einen Flug zum Ort B und dort eines der Hotels B1, B2, B3.

3.1 Teilaufgabe

Welche Konfliktsituationen bzw. Inkonsistenzen können bei parallelem Betrieb entstehen? Nennen Sie zwei verschieden geartete Situationen. Entstehen die Konflikte auch, wenn man die Flüge bzw. die Übernachtungen isoliert betrachtet?

3.2 Teilaufgabe

Welche Attribute und Methoden dieses Reservierungssystems gehören in einen Monitor, um die oben erkannten Konflikte bzw. Inkonsistenzen zu vermeiden? Reicht das? Wieviel Parallelität bleibt in diesem Beispiel?

3.3 Teilaufgabe

Mit der Angabe von **synchronized** im Methodenkopf aller Zugriffsmethoden kann man in Java einen Monitor innerhalb eines Objekts verwirklichen. Implementieren Sie Methoden **buchenHotelB1()** und **stornierenHotelB1()** in Java so, dass auch bei paralleler Ausführung keine Inkonsistenzen in der Anzahl der Reservierungen für Hotel B1 auftreten können.

3.4 Teilaufgabe

In Java kann man auch das Konstrukt der Synchronized Statements (**synchronized(obj) { ... }**) benutzen. Lösen Sie mit diesem Konstrukt den Konflikt, der durch unterschiedliche Buchungsreihenfolgen auftreten kann.

Abgabe

Programmtexte als .java-Datei, Antworten als .doc, .pdf, oder .txt-Datei.

4 Tutoraufgabe (Das Producer-Consumer-Problem mit Monitoren)

Lernziele

Zum letzten Mal (-:-) betrachten wir das Producer Consumer Problem, diesmal die Lösung mit Monitoren. Ziel ist es das inzwischen mutmasslich gut verstandene Monitorkonzept nochmal in anderer Notation kennenzulernen um das Verständnis zu prüfen und zu vertiefen.

4.1 Teilaufgabe

Auf dem letzten Blatt hatten wir den Versuch, das Producer Consumer Problem mit sleep und wakeup zu lösen als C-Programm-Ausschnitt gegeben. Eine Aufgabe war es, dabei auftretende Probleme zu charakterisieren. Als Wiederholung und Transfer in die "Java-Sprechweise": Wo können Thread Interference und wo Memory Consistency Errors auftreten?

```
final int N=100 /* Anzahl Plätze im Puffer */
int count = 0; /* Anzahl der Elemente im Puffer */
void producer(void) {
    int item;
    while (TRUE) { /* Endlosschleife */
        item = produce_item(); /* erzeuge nächstes Element */
        if (count == N)
            sleep(); /* wenn Puffer voll, schlafe */
        insert_item(item); /* schreibe Element in Puffer */
        count = count + 1; /* erhöhe Anzahl der Elemente */
        if (count == 1)
            wakeup(consumer); /* war der Puffer leer? */
    }
}
void consumer(void) {
    int item;
    while (TRUE) { /* Endlosschleife */
        if (count == 0)
            sleep; /* wenn Puffer leer, schlafe */
        item = remove_item(); /* hole Element aus Puffer */
        count = count - 1; /* erniedrige Anzahl der Elemente */
        if (count == N-1)
            wakeup(producer); /* war der Puffer voll? */
        consume_item(item);
    }
}
```

Die Lösung mit den im Tanenbaum beschriebenen Monitoren (S 115f) sieht in "Pidgin Pascal" so aus wie in Abb. 1 (Quelle: Tanenbaum (deutsche Ausgabe)): Warum kann es nicht zu Memory Consistency Errors kommen? Warum kann es nicht zu Tread Interference Errors kommen, wenn bspw an der Stelle * die betreffende Anweisung "mittendrin" unterbrochen wird?

```

monitor ErzeugerVerbraucher
  condition full, empty;
  integer count;
  procedure insert(item: integer);
  begin
    if count = N then wait(full);
    insert_item(item);
    count := count + 1;
    if count = 1 then signal(empty)
  end;
  function remove: integer;
  begin
    if count = 0 then wait(empty);
    remove = remove_item;
    count := count - 1;
    if count = N - 1 then signal(full)
  end;
  count := 0;
end monitor;

procedure Erzeuger;
begin
  while true do
    begin
      item = produce_item;
      ErzeugerVerbraucher.insert(item)
    end
  end;
procedure Verbraucher;
begin
  while true do
    begin
      item = ErzeugerVerbraucher.remove;
      consume_item(item)
    end
  end;
end;

```

Figure 1: Erzeuger-Verbraucher Problem mit Monitoren gelöst. (Quelle Tanenbaum (deutsche Ausgabe))

Abgabe

Die Aufgabe wird gemeinsam in den Tutorübungen erarbeitet und soll nicht abgegeben werden.