Programmieren II

Dr. Klaus Höppner

Hochschule Darmstadt – Sommersemester 2010

Nebenläufigkeit

Moderne Betriebssysteme unterstützen das Prinzip der *Nebenläufigkeit*. Dies bedeutet, dass Prozesse parallel ausgeführt werden können.

Nur auf Multiprozessor-Systemen können Prozesse echt parallel ablaufen. Ansonsten laufen diese nur quasi-parallel ab, indem das Betriebssystem den einzelnen Prozessen regelmäßig kleine Portionen CPU-Zeit zuordnet, so dass diese als gleichzeitig ablaufend erscheinen.

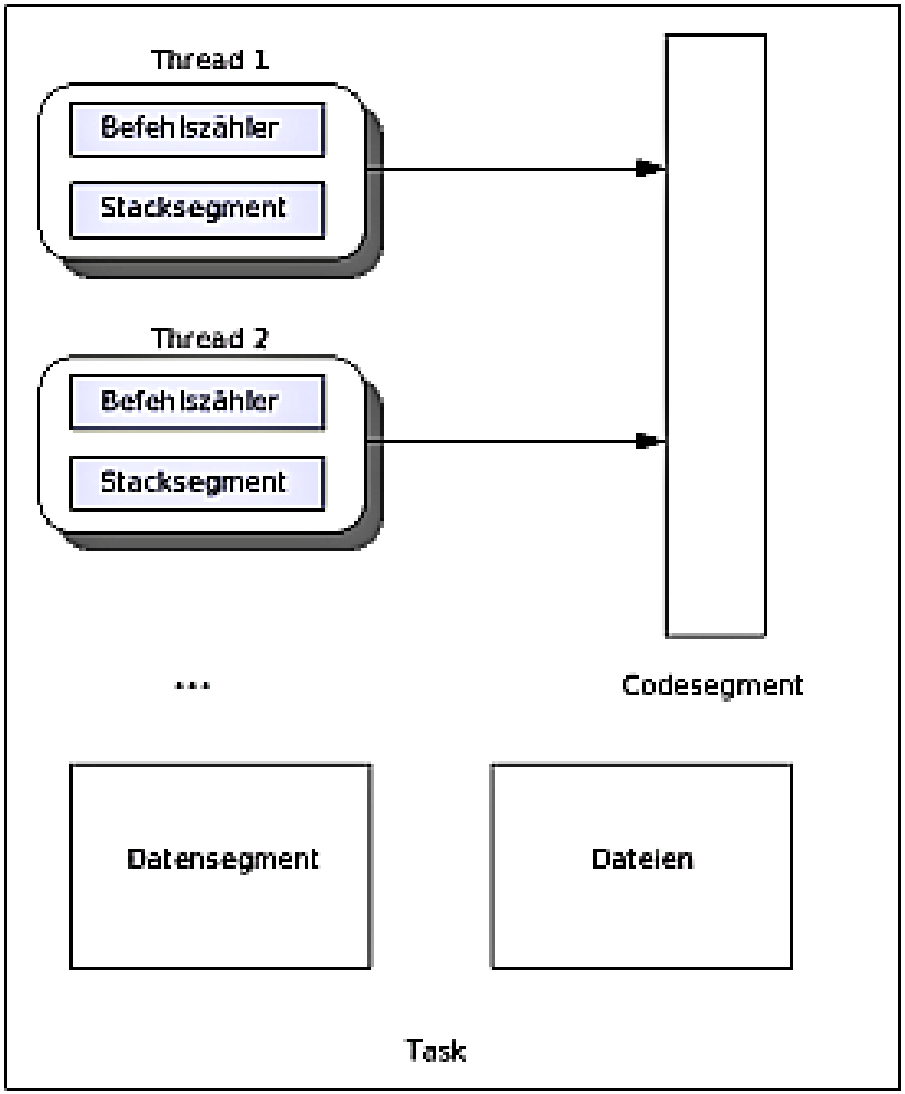
Da ein einzelner Prozess selten ständig 100 % der CPU-Zeit benötigt, ist hiermit oft kein wesentlicher Geschwindigkeitsverlust feststellbar.

Threads

Neben Multitasking (quasi-parallele Ausführung unabhängiger Prozesse mit eigenem Adressraum) unterstützen moderne Betriebssysteme auch *Multithreading*.

Hier laufen innerhalb eines Programms mehrere Ausführungsstränge (*Threads*, dt. Fäden) nebenläufig ab. Diese Thread besitzen zwar einen eigenen Stack, teilen sich aber die sonstigen Ressourcen des Programms. Da die Threads den gleichen Adressraum verwenden, ist somit eine direkte Kommunikation zwischen Threads möglich. Threads können sich so aber auch negativ beeinflussen. Daher müssen manchmal kritische Bereiche geschützt werden, was wieder zu Verklemmungen (*deadlock*) führen kann.

Schaubild


Thread 1
Befehlszähler
Stacksegment
Thread 2
Befehlszähler
Stacksegment
...
Codesegment
Datensegment
Dateien
Task

Threads in Java

Java unterstützt seit jeher Threads, sogar unter Betriebssystemen, die nicht multithreaded sind (dann übernimmt die JVM die Threadverwaltung, d. h. Threadumschaltung einschließlich Stackverwaltung).

Für Java-Threads existiert die Klasse `Thread`. Diese erwartet ein Objekt, das das Interface `Runnable` implementiert.

`Runnable` definiert dabei die abstrakte Methode `run()`, die ein Objekt dann implementieren muss. Ein Thread wird dann mit `start()` gestartet, wobei dieser dann für das Objekt, das `Runnable` implementiert, die Methode `run()` nebenläufig ausführt.

Beispiel

```
Runnable r1 = new Runnable() {
    public void run() {
        for (int i=0; i<100000; i++) {
            double summe = 0.0;
            for (int j=0; j<100; j++) summe += Math.sin(Math.rando
            if (i%1000 == 0) System.out.println(new Date());
        }
    }
};
Runnable r2 = new Runnable() {
    public void run() {
        for (int i=0; i<100000; i++) {
            double summe = 0.0;
            for (int j=0; j<100; j++) summe += Math.sin(Math.rando
            if (i%1000 == 0) System.out.println(i);
        }
    }
};
```

Starten der Threads

Im vorigen Beispiel werden zwei `Runnable`-Objekte angelegt, die in einer Schleife Sinuswerte aufaddieren (dies dient hier ausschließlich zum „Verbraten“ von Rechenzeit) und regelmäßig Meldungen auf der Konsole aufgeben.

Damit die Methoden `run()` beider Objekte parallel ablaufen, werden diese nun per Thread gestartet:

```
new Thread(r1).start();
new Thread(r2).start();
```

Nun erscheinen die Ausgaben der beiden Treads miteinander vermischt auf der Konsole.

Unterbrechen von Threads

Die beiden Threads im vorigen Beispiel laufen solange, bis bei beiden `Runnable`-Objekten `run()` fertig ist (und solange läuft das Programm auch in der JVM, auch wenn `main` schon zu Ende ist).

Ein Thread kann mit `interrupt()` zum Beenden aufgefordert werden. In diesem Fall wird ein Interrupt-Flag gesetzt. Dass die Methode `run()` hierauf reagiert, muss aber selber implementiert werden.

Im folgenden Beispiel werden 2 Sekunden lang Zufallszahlen ausgegeben. Hierbei wird eingeführt, dass `Thread` auch `Runnable` implementiert, man also von `Thread` erben und dort `run()` überladen kann.

Beispiel für `interrupt()`

```
Thread t = new Thread() {
    @Override
    public void run() {
        while (!isInterrupted())
            System.out.println(Math.random());
    }
};

t.start();
try {
    Thread.sleep(2000);
} catch (InterruptedException e) {
} finally {
    t.interrupt();
}
```

Treads und Dämonen

Die JVM führt ein Programm solange aus, solange noch Threads des Programms aktiv sind. Dies führt dazu, dass ein Programm noch weiter laufen kann, auch wenn die Methode `main(...)` abgearbeitet ist (d. h. der Hauptthread des Programms zu Ende ist).

Solche Threads werden als *User Thread* bezeichnet. Wenn also der Hauptthread endet, wird die JVM nicht beendet, solange noch User Threads laufen.

Demgegenüber kann ein Thread als *Demon Thread* gekennzeichnet werden, durch Aufruf von `setDemon(true)` *vor* dem Starten des Threads. Diese werden von der JVM nicht beachtet, sobald also nur noch Dämonen laufen, wird die Anwendung endgültig beendet.

Konflikte bei konkurrierenden Threads

Während Threads zwar ihren eigenen Stack besitzen, greifen Sie ansonsten auf den selben Adressbereich zu. Sie teilen sich also Daten!

Dies muss bei der Programmierung berücksichtigt werden. Es kann passieren, dass in einem Thread Daten geändert werden, während ein anderer Daten verarbeitet (und dabei implizit davon ausgeht, dass die zu Daten sich während der Verarbeitung nicht ändern).

Probleme treten immer dann auf, wenn mitten in solch einem *kiritschen Bereich* eine Threadumschaltung passiert. Die Aufgabe des Programmierers besteht also darin, solche kritischen Bereiche zu identifizieren und zu schützen.

Beispiel

```
public class Konflikt1 extends Thread {
    private static double x;

    @Override
    public void run() {
        while (!isInterrupted()) {
            double x_lokal = Math.random();
            x = x_lokal;

            double summe = 0;
            for (int i=0; i<100; i++) summe+=Math.sin(Math.random(

            if (x != x_lokal)
                System.out.format("Ups: x=%10.8f x_lokal=%10.8f\n"
                                  x,x_lokal);
        }
    }
}
```

Analyse

Eigentlich sieht der Code der Methode `run()` sehr harmlos aus.
Die lokale Variable `x_lokal` wir auf einen zufälligen Wert gesetzt, dieser wird auch dem statischen Attribut `x` zugewiesen. Anschließend werden einige Sinus-Werte berechnet, um Rechenzeit zu verbrauchen.

Eigentlich sollte die Bedingung `x!=x_lokal` niemals zutreffen. Laufen jedoch mehrere Threads gleichzeitig

```
new Konflikt1().start();
new Konflikt1().start();
```

so kann es passieren, dass zwischen Zuweisung und Abfrage durch den ersten Thread der zweite das gemeinsam benutzte Attribut `x` ändert, während `x_lokal` des ersten sich nicht ändert, da diese im eigenen Stackbereich des Threads liegt.

Schützen kritischer Bereiche

Im vorigen Beispiel stellt der Code zwischen Zuweisen an das statische Attribut und Testen auf Gleichheit mit der lokalen Variable einen kritischen Bereich dar. Hier muss sicher gestellt werden, dass dieser von einem Thread immer komplett durchlaufen wird und ein anderer Thread hier nicht gleichzeitig eindringen kann.

Kritische Bereiche können in Java mit einem *Monitor* geschützt werden. Solch ein Monitor besitzt einen Lock-Mechanismus, der verriegelt wird, wenn ein Thread den kritischen Bereich betritt, und wieder entriegelt, wenn er den Bereich verlässt.

Java kennt dabei zwei wesentliche Methoden zum Schutz kritischer Bereiche:

- Direkt über ein `Lock`-Objekt,
- über ein **`synchronized`** für einen Bereich mit einem beliebigen Objekt als Monitor.

Beispiel mit Lock

```
public class Konflikt2 extends Thread {
    private static Lock lock = new ReentrantLock();
    private static double x;

    @Override
    public void run() {
        while (!isInterrupted()) {
            double x_lokal = Math.random();
            lock.lock();
            x = x_lokal;

            double summe = 0;
            for (int i=0; i<100; i++) summe+=Math.sin(Math.random(

            if (x != x_lokal) System.out.println("Ups");
            lock.unlock();
        }
    }
}
```

Das Interface Lock

Das Interface `Lock` (dessen wichtigste Implementierung die Klasse `ReentrantLock` ist) besitzt folgende Methoden:

`void lock()` wartet so lange, bis der kritische Bereich betreten werden kann und verriegelt ihn.

`boolean tryLock()` betritt den kritischen Bereich, falls er frei ist, und verriegelt ihn. Wartet nicht! Gibt **`true`** zurück, falls der Bereich betreten wurde, sonst **`false`**.

`boolean tryLock(long time, TimeUnit unit)` wartet die angegebene Zeit auf das Betreten, unterbrechbar.

`void lockInterruptibly()` wirkt wie `lock`, aber unterbrechbar.

`void unlock()` entriegelt den kritischen Bereich.

Schützen mit synchronized

Der alternative Weg zum Schutz eines kritischen Bereiches besteht in dem Verwenden von **synchronized**.

Hierbei kann ein Bereich als synchronisiert gekennzeichnet werden, der von keinem anderen Thread betreten werden darf, während er von einem Thread durchlaufen wird. Für die Synchronisation wird hierbei ein beliebiges *Monitor-Objekt* verwendet.

Die grundlegende Syntax lautet dabei

```
synchronized (obj) {
    ....
}
```

(NB: Es ist natürlich sinnlos, ein Objekt als Monitor-Objekt zu verwenden, das je Thread *lokal* ist.)

Beispiel für synchronized

```
public class Konflikt3 extends Thread {
    private static Object o = new Object();
    private static double x;

    @Override
    public void run() {
        while (!isInterrupted()) {
            double x_lokal = Math.random();
            synchronized (o) {
                x = x_lokal;

                double summe = 0;
                for (int i=0; i<100; i++) summe+=Math.sin(Math.ran

                if (x != x_lokal) System.out.println("Ups");
            }
        }
    }
}
```

Beispiel

Im Folgenden soll für ein reales Beispiel geprüft werden, wo die kritischen Bereiche liegen.

Eine generische Klasse `BoundedArrayList` hat die Klasse `ArrayList` als Vorbild, soll aber die Zahl der Elemente in der Liste auf die Zahl `max` beschränken.

Code

```
public class BoundedArrayList<T> {
    public final static int max = 10;
    private Object[] data = new Object[max];
    int size = 0;

    public void add(T t) {
        if (size==max) throw new ArrayIndexOutOfBoundsException(ma
        data[size++] = t;
    }

    public void remove(int index) {
        if (index<0 || index>=size)
            throw new ArrayIndexOutOfBoundsException(index);
        for (int i=index; i<size-1; i++) data[i]=data[i+1];
        data[size-1]=null;
        size--;
    }
```

Code

```
    public T get(int index) {
        if (index<0 || index>=size)
            throw new ArrayIndexOutOfBoundsException(index);
        return (T) data[index];
    }

    public void set(int index, T t) {
        if (index<0 || index>=size)
            throw new ArrayIndexOutOfBoundsException(index);
        data[index] = t;
    }

    public int getSize() {
        return size;
    }
}
```

Analyse

Wo liegen hier nun die kritischen Bereiche?

- `add` prüft zunächst, ob die Liste voll ist, und hängt dann ein Element an. Was ist, wenn noch ein Platz frei ist, ein Thread erfolgreich prüft, dass noch Platz ist, aber vor dem Anhängen ein anderer Thread zuvor kommt und den letzten Platz schon belegt?
- `remove` kopiert diverse Elemente hin und her. Hier wäre es fatal, wenn währenddessen ein anderer Thread etwas an einem dieser Element ändert.
- `get` und `set` prüfen zuerst, ob der Index im gültigen Bereich liegt. Danach werden Operationen mit Datenelementen getätigt. Was ist, wenn zwischen Prüfen auf gültigen Index und Datenoperation in einem anderen Thread ein Element gelöscht wird?

Schützen kompletter Methoden

Die Methoden `add`, `remove`, `get` und `set` sind komplett kritisch. Diese müssen also über das Objekt **`this`** synchronisiert werden.

Dies kann z. B. wie folgt geschehen:

```
public void add(T t) {
    synchronized (this) {
        if (size==max)
            throw new ArrayIndexOutOfBoundsException(max);
        data[size++] = t;
    }
}
```

oder kürzer

```
public synchronized void add(T t) {
    if (size==max)
        throw new ArrayIndexOutOfBoundsException(max);
    data[size++] = t;
}
```