Parallelprogrammierung – Multithreading (in Java)

Programmierparadigmen

Johannes Brauer

4. April 2020

Ziele

- Verständnis für mögliche Probleme nebenläufiger Programme erlangen
- Grundkenntnisse von Multithreading-Techniken in Java erwerben

Multithreading – Grundlagen

Die Ausführungen in diesem Kapitel orientieren sich an [Kal15].

Überblick

- Vorteile von Multithreading gegenüber Multiprocessing:
  - Threads verursachen geringere „Kosten“ als Threads.
  - Ein Prozess besteht aus einem oder mehreren Threads.
  - Der Umschaltprozess zwischen Prozessen (context switch) kostet viel Rechenzeit (im Millisekundenbereich).
  - Das Umschalten zwischen zwei Threads desselben Prozesses ist nahezu kostenlos.
- Nachteile von Multithreading gegenüber Multiprocessing:
  - Threads eines Prozesses teilen sich den Adressraum.
  - Dafür, dass es nicht zu unkoordiniertem Zugriff auf dieselbe Speicherzelle kommt ist jetzt nicht das Betriebssystem sondern der Programmierer selbst zuständig.
  - So muss der Programmierer Wettläufe (*race conditions*) verhindern.
  - Koordinierungsmaßnahmen (Sperren) können zu Verklemmungen (*deadlocks*) führen.

Handhabung von Threads durch Betriebssysteme

- Manche Systeme (z. B. Linux) implementieren Threads als Prozesse mit gemeinsamen Adressraum.
- Andere (z. B. Windows) besitzen besondere Unterstützung von Threads im Kernel.
- Allen Systemen gemeinsam ist der geteilte Adressraum für Threads desselben Prozesses.

Beispiel für einen Wettlauf in Java

Benutzung von `RaceC` von der Kommandozeile:

```
bash-3.2$ javac RaceC.java
bash-3.2$ java Main
n's value is: -332920
n's value is: 237836
n's value is: 73816
n's value is: 61941
n's value is: 59569
```

```
n's value is: -2
n's value is: -2
n's value is: 80561
```

Quellcode von RaceC.java

```
/** To compile: javac RaceC.java
    To run:     java Main

    Flow of control for Threads referenced by t1 and t2 (for convenience,
    call the Threads referenced by t1 and t2 simply 't1' and 't2'):

    The 'main thread' is the thread that executes the method 'main', in
    this case in class Main (see below).

                creates
    main thread------------>t1 and t2

                 starts
    main thread------------>t1 and t2

                calls run()
    Java runtime-------------->on started Threads t1 and t2

    At this point, three threads in the app are executing: main, t1, and t2.

    An instance of a Java Thread represents the sequence of instructions in the
    body of the encapsulated 'run' method (and any other instructions in methods
    that 'run' invokes).

    A thread that exits run() terminates, and cannot be restarted.
*/
public class RaceC {
    static int n;                               // a single instance on n
    static void race() {
        n = 0;                                  // initialize to zero before threads
                                                // alter its value
        long limit = Integer.MAX_VALUE * 2L; // four billion and change
        Thread t1 = new Thread() {              // incrementing thread
                public void run() {
                    for (long i = 0; i < limit; i++) n = n + 1; // increment limit times
                }
            };
        Thread t2 = new Thread() {              // decrementing thread
                public void run() {
                    for (long i = 0; i < limit; i++) n = n - 1; // decrement limit times
                }
            };
        t1.start();  // start t1's execution
        t2.start();  // start t2's execution
        try {
            t1.join();  // wait here until t1 terminates
            t2.join();  // wait here until t2 terminates
        } catch(Exception e) { }
        System.out.println("n's value is: " + n);
    }
}

class Main {
```

```
    public static void main(String[ ] args) {      //*** main thread executes method main
        for (int i = 0; i < 8; i++) RaceC.race();
    }
}
/** Output from a sample run:

    n's value is: -265936
    n's value is: -7317
    n's value is: 47128
    n's value is: -219153
    n's value is: 82805
    n's value is: -7944
    n's value is: 87322
    n's value is: -2
*/
```

Analyse des Wettlaufs

- `T1` and `T2` sind Threads im Prozess `P`: `T1` und `T2` haben Zugriff auf dieselben Speicherzellen.
- `N` ist eine Speicherzelle im gemeinsamen Adressraum von `T1` und `T2`.
- `T1` inkrementiert, `T2` dekrementiert `N`.

```
   N = N + 1  +---+  N = N - 1        ## Sind diese Zuweisungen ,,atomar‘‘?
T1----------->| 5 |<-----------T2
      +---+
        N
```

- Jede Instruktion beinhaltet eine Addition/Subtraktion und eine Zuweisung.
  - Angenommen das Ergebnis der arithmetischen Operation wird in einem CPU-Register oder auf dem Stack (jedenfalls an einem flüchtigen Platz) zwischengespeichert:

```
Temp1 = N + 1  ### incrementiere N und speichere das Ergenbis zwischen
                   (N ist noch unverändert)
N = Temp1      ### Weise den zwischengespeicherten Wert N zu
```

Beispielszenario

- Die Maschine besitzt mehrere Rechenkerne: `T1` und `T2` laufen auf unterschiedlichen Kernen.
- Im Folgenden wird ein möglicher Ausgang eines Wettlaufs dargestellt (`Temp1` und `Temp2` sind die flüchtigen Zwischnspeicher für `T1` resp. `T2`.):

```
Clock-Takte:

C1: Temp1 = 5 + 1 = 6   ## T1's Addition
C2: Temp2 = 5 - 1 = 4   ## T2's Subtraction (T1's Zuweisung hat noch nicht
                        ##                   stattgefunden.)
C3: N = Temp2           ## T2's Zuweisung (N erhält den Wert 4)
C4: N = Temp1           ## T1's Zuweisung (N erhält den Wert 6)
```

- Nach einmaliger Inkrementierung und Dekrementierung um 1 ist der Endwert von `N` 6 anstelle von 5.
- Worin besteht die Ursache des Fehlers und wie könnte er behoben werden?
- Problem des überlappenden Ablaufs von Operationen:
  - Der Thread, der seine arithmetische Operation beginnt, muss die Möglichkeit haben, auch die Zuweisung ohne Unterbrechung auszuführen.
  - Die Threads müssten in der Weise *synchronisiert* werden, dass durch Sperren (*locking*) von `N` die beiden Teiloperationen nur von einem Thread ausgeführt werden können.

Explizite Sperrverfahren

Code-Beispiele

```
/* Thread synchronization mechanisms: code examples */

/** C example **/
static pthread_mutex_t lock;  /* declare a lock */              /** line 1 **/
void increment_n() {                                           /** line 2 **/
    pthread_mutex_lock(&lock);                                 /** line 3 **/
    n = n + 1;  /* critical section code */                    /** line 4 **/
    pthread_mutex_unlock(&lock);                               /** line 5 **/
}

/** Java example **/
class Test {
    static int n; // a single, shared storage location         /** line 6 **/
    static Object lock = new Object(); // can't be null       /** line 7 **/
    void incrementN() {
        synchronized(lock) {                                   /** line 8 **/
            n = n + 1; /* critical section code */             /** line 9 **/
        }                                                      /** line 10 **/
    }
}
```

Verfahren für die Koordination von Threads

- Die im Folgenden genannten Begriffe können sowohl auf Prozesse als auch auf Threads angewendet werden. Hier stehen die Threads im Vordergrund.
- Ein *kritischer Abschnitt* im Programmcode ist eine Anweisungsfolge, die nur von einem Thread zur Zeit ausgeführt werden darf.
  - Beispiel: Wenn die Threads `T1` und `T2` beide den Code

    ```
    N = N + 1  ## dieselbe Speicherzelle N ist für T1 und T2 zugreifbar
    ```

    ausführen können, handelt es sich um einen kritischen Abschnitt. `T1` und `T2` dürfen den Code nicht gleichzeitig ausführen.
- Ein Sperrmechanismus muss folgendes leisten:
  - Wechselseitiger Ausschluss (*mutual exclusion*) muss gewährleistet werden: Wenn `T1` die Sperre erlangt, wird `T2` solange vom Betreten des kritischen Abschnitts ausgeschlossen, wie `T1` die Sperre hält.
  - Wenn kein Thread eine Sperre hält, darf jeder Thread die Sperre erwerben und den kritischen Abschnitt betreten.

Bekannte Sperrmechanismen

- Eine *Semaphore* begrenzt die Anzahl der Threads, die gleichzeitig auf ein geteiltes Betriebsmittel Zugriff haben.
  - Eine Semaphore mit dem Wert 3 erlaubt maximal 3 Threads den Zugriff auf die durch sie geschützte Ressource.
- Eine *Mutex* ist eine Semaphore mit dem Wert 1: Der Thread, der die Mutex besitzt (hält), hat als einziger Zugriff auf die geschützte Ressource.
  - Eine Mutex stellt wechselseitigen Ausschluss sicher.
- Ein *Monitor* stellt auch wechselseitigen Ausschluss sicher. Darüber hinaus stellt er weitere Mechanismen für die Thread-Kooperation bereit.

- Der *synchronized*-Block in Java stellt
  - den *wait*-Mechanismus bereit, der nicht-aktives Warten auf das Freigeben einer Sperre unterstützt und
  - den *notify*-Mechanismus, der wartende Threads über eine freigegebene Sperre unterrichtet, bereit.

Beispiel für eine Verklemmung (in Java)

Benutzung von `Deadlock` von der Kommandozeile:

```
bash-3.2$ javac Deadlock.java
bash-3.2$ java Deadlock
thread2 holds lock2
thread1 holds lock1
thread2 waiting for lock1
        (thread2 needs lock1 to release lock2...)
thread1 waiting for lock2
        (thread1 needs lock2 to release lock1...)
  ^C ^Cbash-3.2$
bash-3.2$ java Deadlock
thread1 holds lock1
thread2 holds lock2
thread2 waiting for lock1
thread1 waiting for lock2
        (thread2 needs lock1 to release lock2...)
        (thread1 needs lock2 to release lock1...)
  ^C ^Cbash-3.2$
```

Quellcode von Deadlock.java

```
/** To compile and run from the command-line:
      javac Deadlock.java
      java Deadlock
*/
public class Deadlock {
    static Object lock1 = new Object(); // a single lock1                    /** line 1 **/
    static Object lock2 = new Object(); // a single lock2

    /** Each thread executes its run() method. **/
    public static void main(String args[]) {
        Thread thread1 = new Thread() {
                public void run() {
                    synchronized(lock1) {                                    /** line 2 **/
                        print("thread1 holds lock1");
                        try { Thread.sleep(2); } catch(Exception e) { }      /** line 3 **/
                        print("thread1 waiting for lock2");
                        print("\t(thread1 needs lock2 to release lock1...)");
                        synchronized(lock2) {                                /** line 4 **/
                            print("thread1 holds lock1 and lock2");
                        } // lock2 released here                             /** line 5 **/
                    } // lock1 released here                                 /** line 6 **/
                }
            };
        Thread thread2 = new Thread() {
                public void run() {
                    synchronized(lock2) {                                    /** line 7 **/
                        print("thread2 holds lock2");
                        try { Thread.sleep(2); } catch(Exception e) { }
                        print("thread2 waiting for lock1");
```

```
                    print("\t(thread2 needs lock1 to release lock2...)");
                    synchronized(lock1) {                                   /** line 8 **/
                        print("thread2 holds lock2 and lock1");
                    } // lock1 released here                                /** line 9 **/
                } // lock2 released here                                    /** line 10 **/
            }
        };
        thread1.start();                                                    /** line 11 **/
        thread2.start();                                                    /** line 12 **/
    }
    private static void print(Object obj) { System.out.println(obj); }
}
```

Behandlung der Nebenläufigkeit von Threads

- Durch Multithreading werden dem Programmierer Aufgaben aufgebürdet, die beim Multiprocessing durch das Betriebssystem erledigt werden.
  - Es hindert Prozesse auf die Speicherbereiche von anderen Prozessen zuzugreifen.
  - Für Threads muss dies der Programmierer leisten.
- Wie kann Thread-Sicherheit, d. h. Sicherheit von Wettläufen, gewährleistet werden?
  - Beispiel: Eine änderbare Liste und eine `ADD`-Operation

```
        ADD 3  +---+---+
thread------->| 1 | 2 |        ## Liste L vor Ausführung von ADD
               +---+---+
               +---+---+---+
               | 1 | 2 | 3 |    ## Ausführung von ADD verändert die Original-Liste
               +---+---+---+
```

Basis-Ansatz in Java

- Sicherstellen des wechselseitigen Ausschlusses durch Sperren.
  - Sperre erlaubt nur einem einzigen Thread Zugriff auf die Liste während einer `ADD`-Operation.

Basis-Ansatz in Clojure

- Objekte im Speicher sind grundsätzlich (bis auf wenige wohl-definierte Ausnahmen) unveränderbar (immutable).
  - Die `ADD`-Operation fertigt zuerst eine Kopie der Original-Liste an und ändert dann die Kopie.

```
        ADD 3  +---+---+
thread------->| 1 | 2 |        ## Liste L
               +---+---+
                   L

               +---+---+
               | 1 | 2 |        ## Kopie von Liste L
               +---+---+
                   Lc

               +---+---+---+
               | 1 | 2 | 3 |    ## Kopie nach Ausführung von ADD
               +---+---+---+    ## (Original ist unverändert)
                   Lc
```

Basis-Ansatz in Go

- Es gibt nur einen Thread, der den Zugriff auf die Liste steuert.
- Andere Threads senden Nachrichten an diesen, z. B. die Nachricht `ADD`.
  - Diese Nachrichten werden an einen thread-sicheren Kanal geschickt.

```
                +---+---+
                | 1 | 2 |      ## ein einzelner Thread T verwaltet den Zugriff auf L
                +---+---+

          ADD 3                 ADD 33
  thread1------->channel to T<--------thread2  ## der Kanal verfügt über
                                               ## eingebaute Synchronisation
```

Zusammenfassung

- Multithreading kann eine effiziente Methode für nebenläufige Programme sein, da der Kontextwechsel zwischen Threads nur geringe Kosten verursacht.
  - Bevorzugte Methode für Nebenläufigkeit z. B. in Java und C#.
- Multithreading ruft drei Herausforderungen für Programmierer hervor:
  1. Vermeidung von Wettläufen bei Zugriff auf gemeinsam genutzte Speicherbereiche
  2. Vermeidung von „Übersychronisation" – dadurch könnte die Parallelarbeit behindert werden (Thema Sperrgranularität)
  3. Vermeidung von Verklemmungen: Planung von sukzessiven Sperren.
- Programmiersprachen bieten heutzutage Konstrukte und Typen auf höherer Ebene an, um die nebenläufige Programmierung zu erleichtern.
  - Manchmal ist eine einfache Mutex aber das Mittel der Wahl.

Multithreading – Vertiefung

Die Ausführungen in diesem Kapitel orientieren sich an [Kal15].

Das Geizhals-Verschwender-Problem in Java

- Der Geizhals (miser) inkrementiert ein Konto um 1.
- Der Verschwender (spendthrift) dekrementiert dasselbe Konto um 1.
- Die Anzahl der Inkrementierungen/Dekrementierungen wird als Kommandozeilenargument übergeben.

Übersicht - drei Java-Implementierungen

```
                          code
                           |
       +-------------------+-------------------+
       |                   |                   |
     acct                acct2               acct3        ## Java Package-Namen
Account.java        Account.java        Account.java
RaceCondition.java  RaceCondition.java  RaceCondition.java ## Applikationsklasse (main)
   acct Implementierung: keine Thread-Koordination, Wettläufe möglich
  acct2 Implementierung: explizite Thread-Koordination (low-level) mit
        synchronized blocks
  acct3 Implementierung: Verwendung von AtomicInteger (eingebaute Synchronisation)
```

- Der Quellcode ist in miserSpendthriftJava.zip zu finden.

- Verzeichnisstruktur

```
code/
code/acct/
code/acct/Account.java
code/acct/RaceCondition.java
code/acct2/
code/acct2/Account.java
code/acct2/RaceCondition.java
code/acct3/
code/acct3/Account.java
code/acct3/RaceCondition.java
```

- Übersetzen und Starten aus dem `code`-Verzeichnis von der Kommandozeile:

```
javac acct/*.java
java acct.RaceCondition 50000000

javac acct2/*.java
java acct2.RaceCondition 600000

javac acct3/*.java
java acct3.RaceCondition 7700000
```

Demo

```
bash-3.2$ cd code
bash-3.2$ javac acct/*.java
bash-3.2$  java acct.RaceCondition 500000
Final balance: -48910
bash-3.2$  java acct.RaceCondition 500000
Final balance: 237043
bash-3.2$  java acct.RaceCondition 500000
Final balance: -30470
bash-3.2$  java acct.RaceCondition 500000
Final balance: -26171
bash-3.2$ javac acct2/*.java
bash-3.2$ java acct2.RaceCondition 500000
Final balance: 0
bash-3.2$ java acct2.RaceCondition 500000
Final balance: 0
bash-3.2$ java acct2.RaceCondition 500000
Final balance: 0
bash-3.2$ javac acct3/*.java
bash-3.2$ java acct3.RaceCondition 500000
Final balance: 0
bash-3.2$ java acct3.RaceCondition 500000
Final balance: 0
bash-3.2$ java acct3.RaceCondition 500000
Final balance: 0
```

Quellcode (Version ohne Synchronisation – acct)

```
package acct;

public class Account {
    public static int balance = 0;                                    /** line 1 **/
}

class Miser extends Thread {       // deposit                         /** line 2 **/
```

```
    Miser(int howMany) { this.howMany = howMany; }

    @Override
    public void run() {
        for (int i = 0; i < howMany; i++)
            Account.balance++;
    }
    private int howMany; // how many times to increment
}

class Spendthrift extends Thread { // withdraw                          /** line 3 **/
    Spendthrift(int howMany) { this.howMany = howMany; }

    @Override
    public void run() {
        for (int i = 0; i < howMany; i++)
            Account.balance--;
    }
    private int howMany; // how many times to decrement
}

/*********************************************************************/

package acct;
public class RaceCondition {                                           /** line 4 **/
    public static void main(String[ ] args) {
        if (args.length < 1) {
            System.err.println("RunCondition <times to iterate>");
            return;
        }
        int n = Integer.parseInt(args[0]);
        Miser miser = new Miser(n);                                    /** line 5 **/
        Spendthrift spendthrift = new Spendthrift(n);                  /** line 6 **/
        miser.start();       // start Miser                            /** line 7 **/
        spendthrift.start(); // start Spendthrift */                   /** line 8 **/
        try {
            miser.join();       // wait for Miser to terminate         /** line 9 **/
            spendthrift.join(); // wait for Spendthrift to terminate   /** line 10 **/
        } catch(Exception e) { System.err.println(e); }
        System.out.println("Final balance: " + Account.balance);
    }
}
```

Quellcode (Version mit `synchronized blocks` – acct2)

```
/*** Fixing the miser/spendthrift problem with explicit thread synchronization. ***/

package acct2;

public class Account {
    public static int balance = 0;                              /** line 1 **/
    public static final Object lock = new Object();             /** line 2 **/
}

class Miser extends Thread {       // deposit                   /** line 3 **/
    Miser(int howMany) { this.howMany = howMany; }

    @Override
    public void run() {
```

```
        for (int i = 0; i < howMany; i++)
            synchronized(Account.lock) {                    /** line 4 **/
                Account.balance++;                          /** line 5 **/
            }
    }
    private int howMany;
}

class Spendthrift extends Thread { // withdraw              /** line 6 **/
    Spendthrift(int howMany) { this.howMany = howMany; }

    @Override
    public void run() {
        for (int i = 0; i < howMany; i++)
            synchronized(Account.lock) {                    /** line 7 **/
                Account.balance--;                          /** line 8 **/
            }
    }
    private int howMany;
}
```

Quellcode (Version mit `AtomicInteger` – acct3)

```
/*** Fixing the miser/spendthrift problem with a thread-safe data type **/

package acct3;
import java.util.concurrent.atomic.AtomicInteger;                   /** line 1 **/

public class Account {
    public static AtomicInteger balance = new AtomicInteger();      /** line 2 **/
}

class Miser extends Thread {        // deposit
    Miser(int howMany) { this.howMany = howMany; }
    @Override
    public void run() {
        for (int i = 0; i < howMany; i++)
            Account.balance.incrementAndGet();                      /** line 3 **/
    }
    private int howMany;
}

class Spendthrift extends Thread { // withdraw
    Spendthrift(int howMany) { this.howMany = howMany; }
    @Override
    public void run() {
        for (int i = 0; i < howMany; i++)
            Account.balance.decrementAndGet();                      /** line 4 **/
    }
    private int howMany;
}
```

Thread-Synchronisation für Kooperation

- Bisher hauptsächlich konkurrierende Threads betrachtet.
- Kooperierende Threads benötigen auch Synchronisationsmechanismen
- Javas `synchronized`-Block realisiert einen *Monitor*, der sowohl wechselseitigen Ausschluss für die Koordination konkurrierender aber auch kooperierender Threads unterstützt.

- Musterbeispiel kooperierender Prozesse: eine Warteschlange mit beliebig vielen Erzeugern und Verbrauchern (consumer-producer-problem):

```
                back                    front
                 \                      /
          insert +---+---+---+---+---+---+ remove
producer------->| 6 | 8 | 4 | 9 | 3 | 7 |------->consumer
   ...          +---+---+---+---+---+---+    ...  ## any number of producers/consumers
  A queue of six items
```

- Vereinfachte Sicht:
  - Zwei Threads bearbeiten eine geteilte Datenstruktur: der eine (*Erzeuger*) fügt Elemente in die Datenstruktur (z. B. eine Warteschlange) ein, während der andere (*Verbraucher*) Elemente entnimmt.
  - Ist die Warteschlange leer, muss der Verbraucher warten, bis der Erzeuger wieder Elemente hinzugefügt hat.
  - Ein Erzeuger weckt nach dem Einfügen eines Elements einen eventuell wartenden Verbraucher auf.
  - Ist die Warteschlange voll, wartet der Erzeuger bis der Verbraucher wieder ein Element entnommen hat.
  - Ein Verbraucher weckt nach der Entnahme eines Elements einen eventuell wartenden Erzeuger auf. .
- Der Einfachheit halber betrachten wir einen Stapel (*stack*) anstelle einer Warteschlange:

```
+---+
| 7 |<-------top
+---+
| 3 |
+---+
| 9 |
+---+
 ...
```

Thread-sichere Implementierung des Stapels

Java-Quellcode ist in StackTS.java zu finden.

```
/** javac StackTS.java
    java MainTS

    Kill at the command-line with Control-C.
*/

/** A thread-safe producer/consumer application with thread
    coordination and cooperation.
*/

import java.util.Random;

public class StackTS {
    private static final int capacity = 8;
    private int top = -1; // empty stack when top == -1
    private int[ ] stack = new int[capacity];

    // When an entire method is synchronized, the implicit lock is the current object
    // --'this' in Java.
```

```
    // Hence, methods push and pop are effectively locked together: only one may be
    // accessed at a time.
    public synchronized void push(Integer n) {              /** line 1 **/
        while ((top + 1) == capacity) { // full?            /** line 2 **/
            try {
                wait(); // if so, wait for a pop            /** line 3 **/
            } catch(InterruptedException e) { }
        }
        log(n + " pushed at " + (top + 1));                 /** line 4 **/
        stack[++top] = n;                                   /** line 5 **/
        notifyAll();                                        /** line 6 **/
    }

    public synchronized void pop() {                        /** line 7 **/
        while (top < 0) { // empty?                         /** line 8 **/
            try {
                wait();                                     /** line 9 **/
            } catch(InterruptedException e) { }
        }
        log(stack[top] + " popped at " + top);              /** line 10 **/
        top--;                                              /** line 11 **/
        notifyAll();                                        /** line 12 **/
    }

    private void log(String msg) {
        System.out.println(msg);
    }
}

class Pusher extends Thread {
    private Random rand = new Random();
    private StackTS stack;

    Pusher(StackTS stack) { this.stack = stack; }           /** line 12 **/

    @Override
    public void run() {
        while (true) {                                      /** line 13 **/
            stack.push(rand.nextInt(100)); // 0 through 99
            try {
                Thread.sleep(rand.nextInt(200)); // sleep 0 to 199 milliseconds
            }  catch(InterruptedException e) { }
        }
    }
}

class Popper extends Thread {
    private Random rand = new Random();
    private StackTS stack;

    Popper(StackTS stack) { this.stack = stack; }           /** line 14 **/

    @Override
    public void run() {
        while (true) {
            stack.pop();                                    /** line 15 **/
            try {
                Thread.sleep(rand.nextInt(100));
            } catch(InterruptedException e) { }
```

```
        }
    }
}

class MainTS {
    public static void main(String[ ] args) {
        StackTS stack = new StackTS();                    /** line 16 **/
        new Popper(stack).start();                        /** line 17 **/
        new Pusher(stack).start();                        /** line 18 **/
        new Pusher(stack).start();                        /** line 19 **/
    }
}
```

Benutzung von der Kommandozeile

```
bash-3.2$ java MainTS
82 pushed at 0
82 popped at 0
68 pushed at 0
15 pushed at 1
15 popped at 1
31 pushed at 1
31 popped at 1
68 popped at 0
46 pushed at 0
31 pushed at 1
31 popped at 1
61 pushed at 1
75 pushed at 2
75 popped at 2
82 pushed at 2
82 popped at 2
80 pushed at 2
72 pushed at 3
72 popped at 3
24 pushed at 3
12 pushed at 4
12 popped at 4
24 popped at 3
94 pushed at 3
1 pushed at 4
1 popped at 4
...
  ^C ^Cbash-3.2$
```

Zusammenfassung: Thread-Koordination, -Kooperation

- Konstrukte für die Thread-Koordination reichen von expliziten Mutexs im Quellcode bis zu thread-sicheren Typen wie Java's `AtomicInteger`.
  - Darüber hinaus gibt es thread-sichere Datenstrukturen: Listen und Hashmaps
- Thread-Synchronisation in Form der Koordination dient der Vermeidung von Wettläufen und Verklemmungen.
- Thread-Synchronisation in Form der Kooperation dient darüber hinaus der Effizienzsteigerung.
  - Ein Thread, der in einem nicht-aktiven Zustand wartet, ist besser als ein Thread der Ressourcen benötigt, um eine Sperre zu erlangen.
  - Das `StackTF`-Beispiel in Java demonstriert dies.

Thread-Sicherheit – Höherwertige Concurrent Types in Java

Die Ausführungen in diesem Kapitel orientieren sich an [Kal15].

Überblick

`java.util.concurrent.atomic`

- Das Toolkit unterstützt thread-sichere Programmierung für Zähler (o. ä.) ohne explizite Sperren. Zu den Typen gehören:
  - `AtomicInteger`
  - `AtomicIntegerArray`
  - `AtomicLong`
  - `AtomicLongArray`
  - Beispiel für mögliche Implementierung eines *atomic counter*:

```
public final class Counter {
   private long value = 0;  // initialized for emphasis
   public synchronized long getValue() { return value; }
   public synchronized long increment() {
      if (value == Long.MAX_VALUE)
        throw new IllegalStateException("Counter overflow");
      return ++value;
   }
}
```

`java.util.concurrent.ConcurrentHashMap`

- Funktioniert wie eine gewöhnliche `Hashtable` nur sehr effizient.
  - Die Map wird in Teile aufgespalten, die separat gesperrt werden können.
  - Zwei Schreiboperationen können parallel ausgeführt werden, solange unterschiedliche Partitionen betroffen sind.
  - Leseoperationen werden ohne Blockierung ausgeführt.
  - Die Map kann nicht als ganze gesperrt werden.

`java.util.concurrent.CopyOnWriteArrayList`

- Funktioniert wie eine persistente Datenstruktur in Clojure, d. h. Änderungsoperationen erzeugen eine Kopie des Arrays.

Interface `java.util.concurrent.BlockingQueue`

- Leseoperation wartet, falls die Warteschlange leer ist, bevor sie das erste Element entnimmt.
- Beispielimplementierungen: `ArrayBlockingQueue` und `PriorityBlockingQueue`

Executors und Thread-pools

- Das `Executor`-Interface stellt eine Methode, `execute`, zur Verfügung, mit deren Hilfe die Thread-Erzeugung optimiert werden kann, indem ein Thread aus einem Pool existierender Threads gestartet wird.
- Thread-pools
  - Die meisten `Executor`-Implementierungen, benutzen Thread-pools, die aus so genannten *worker threads* bestehen.
    - Ein Worker kann für die Ausführung verschiedener Aufgaben wiederverwendet werden.
    - Das kontinuierliche Erzeugen und Zerstören von Threads wird eingespart.

Fork/Join Framework

- Geeignet zur Bearbeitung rekursiver Strukturen, z. B. Bäume.
- Aufgaben werden an Workers aus einem Thread-Pool verteilt.

Beispiel für die Nuzung des Fork-Join-Frameworks

Das Programm `FileSearcher`

- Beispiel für die Zerlegung eines Problems in Teilprobleme, die auf Threads verteilt werden, die auf verschiedenen Prozessoren laufen.
  - Das Thread-management ist hinter einer API verborgen.
  - Teilaufgaben werden durch `fork`-Aufrufe aktiviert.
  - Teilergebnisse werden durch `join`-Aufrufe zusammengeführt.
- Das Beispielprogramm `FileSearcher` durchsucht einen Verzeichnisbaum nach Dateien mit einer bestimmten Dateiendung.
  - Das Programm gibt eine Liste der gefundenen Dateinamen zurück.
- Basisfall und rekursiver Fall in `FileSearcher`:

  Basisfall Ein Thread findet eine Datei (kein Verzeichnis) mit der gesuchten Endung und fügt ihren Namen der Ergebnisliste hinzu..

  rekursiver Fall Ein Thread findet ein Unterverzeichnis, das parallel durchsucht werden könnte.

Quellcode von FileSearcher.java

```
/** javac FileSearcher.java
    java MainFS

    The program searches from the user's home directory for
    files extending with '.txt' as the extension. The list of
    such files is printed to the screen.
*/

import java.io.File;
import java.util.ArrayList;
import java.util.List;
import java.util.Collection;
import java.util.concurrent.RecursiveTask;                          /** line 0 **/

public class FileSearcher extends RecursiveTask<List<String>> {    /** line 1 **/
    private final String path;
    private final String extension;

    public FileSearcher(String path, String extension) {           /** line 2 **/
        this.path = path;
        this.extension = extension;
    }

    @Override
    public List<String> compute() {                                 /** line 3 **/
        List<String> listOfFileNames = new ArrayList<String>();
        File[ ] files = new File(path).listFiles();                 /** line 4 **/
        if (files == null) return listOfFileNames; // base case     /** line 5 **/

        List<FileSearcher> tasks = new ArrayList<FileSearcher>();   /** line 6 **/
        for (File file : files) {                                   /** line 7 **/
```

```
            String absolutePath = file.getAbsolutePath();                /** line 8 **/
            if (file.isDirectory()) {                                    /** line 9 **/
                FileSearcher task = new FileSearcher(absolutePath, extension); /** line 10 **/
                task.fork(); // recursive case                           /** line 11 **/
                tasks.add(task);                                         /** line 12 **/
            }
            else if (file.getName().endsWith(extension))                 /** line 13 **/
                listOfFileNames.add(absolutePath);
        }
        assembleResults(listOfFileNames, tasks);                         /** line 14 **/
        return listOfFileNames;                                          /** line 15 **/
    }
    private void assembleResults(List<String> list, List<FileSearcher> tasks) {
        for (FileSearcher task : tasks) {
            Collection<String> results = task.join();                   /** line 16 **/
            list.addAll(results);                                        /** line 17 **/
        }
    }
}
class MainFS {
    public static void main(String[ ] args) {
        List<String> list =
           new FileSearcher(System.getProperty("user.home"), ".txt").compute();  /** line 18 **/
        System.out.println("Files with extension .txt:");
        for (String string : list) System.out.println(string);           /** line 19 **/
    }
}
```

Demo

!!! live demo !!!

Literaturverzeichnis

Literatur

[Kal15] Martin Kalin. Concurrent and parallel programming concepts, 2015. zuletzt aufgerufen am 10.10.2017.