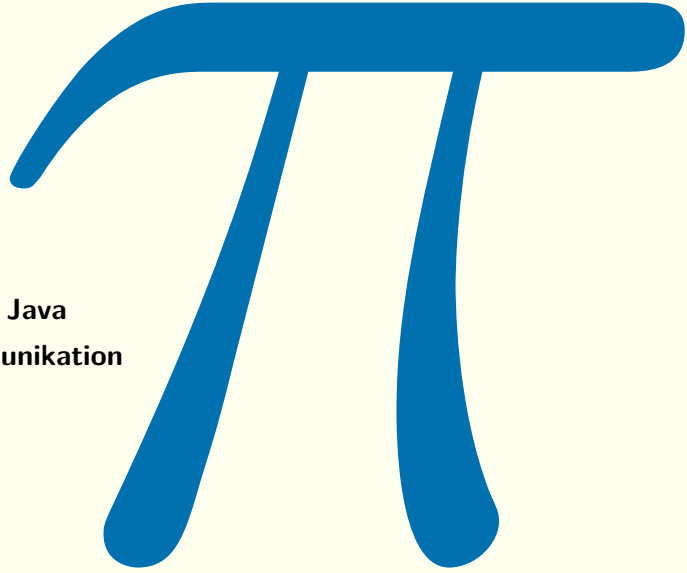
Threads und Synchronisation in Java
Prozesse und Interprozesskommunikation



Atomare Ausführung in Java

```
public class Counter {
    private int i = 0, j = 0;

    public synchronized void flip() { i++; j++; }

    public void flip1() { synchronized(this) { i++; j++; } }

    public void flip2() {
        synchronized(this) { i++; }
        synchronized(this) { j++; }
    }
}
```

Verwendung von `synchronized`

Faustregeln zu nebenläufigen Programmen:

Zugriff auf Objekt- und Klassenvariablen nur innerhalb von `synchronized`

kein `synchronized` nötig für einzelnen Variablenzugriff wenn Variable `volatile`

→ aber nicht atomar (und daher falsch) bei mehreren Zugriffen wie in `i++`

`java.util.concurrent.atomic.AtomicInteger` und `...AtomicLong` bieten ganze Zahlen mit atomaren Zugriffsmethoden ohne `synchronized`

`synchronized`-Methoden und -Blöcke sollen nur kurz laufen

Funktionsweise von `synchronized`

jedes Java-Objekt enthält einen **Lock** als unsichtbare Variable, die einen Thread als Wert enthalten kann (= auf den Thread gesetzt sein kann)

`synchronized` setzt Lock eines Objekts auf aktuellen (gerade ausgeführten) Thread

→ bei `synchronized`-Methoden ist Objekt `this`, bei `synchronized`-Blöcken wählbar
→ wenn Lock auf anderen Thread gesetzt ist: warten bis Lock frei (kein Thread)
→ wenn Lock auf aktuellen Thread gesetzt ist: weitermachen (Rekursion unterstützt)
→ am Ende des `synchronized`-Blocks oder der `synchronized`-Methode: Lock freigeben

→ `wait`, `notify` und `notifyAll` nur verwendbar, wenn in `synchronized`
→ `wait` gibt Lock frei und hängt aktuellen Thread in Warteliste des Objekts
→ `notify` weckt irgendeinen Thread in der Warteliste des Objekts auf
→ `notifyAll` weckt alle Threads in der Warteliste des Objekts auf
→ Threads in der Warteliste können auch grundlos aufgeweckt werden
→ aufgeweckter Thread bekommt Lock und darf weitermachen, sobald Lock frei

Komplexere Synchronisationsbedingungen

```
public class PrinterDriver {
    private boolean online = false;
    public synchronized void print(String s) {
        while (!online) {
            try { wait(); }
            catch(InterruptedException ex) { return; }
        }
        ... // send s to printer
    }
    public synchronized void onOff() {
        online = !online;
        if (online) notifyAll();
    }
    ...
}
```

Der von Threads ausgeführte Code

```
public class Producer implements Runnable {
    private PrinterDriver t;
    public Producer(PrinterDriver t) { this.t = t; }
    public void run() {
        String s = ....
        for (;;) {
            ...          // produce new value in s
            t.print(s); // send s to the printer server
        }
    }
}
```

Nebenläufigkeit in der Praxis

wegen Komplexität Basiskonzepte eher meiden, aber Vermeidung nicht immer möglich

Alternativen für Standardanwendungen:

in `java.util.concurrent` und `java.util.concurrent.atomic`

`Future` und `FutureTask` für Berechnungen im Hintergrund

→ Ergebnis über Getter lesbar, Getter wartet bis Berechnung fertig

→ Hintergrundberechnung darf nicht von anderen Berechnungen beeinflusst werden

`Executor` für mehrere kleine, voneinander unabhängige Aufgaben

→ je nach verfügbaren Ressourcen hintereinander oder parallel ausgeführt

→ z. B. für die Darstellung einzelner Elemente im Webbrowser

→ unterschiebliche Implementierungen, z. B. `ThreadPoolExecutor`

→ über viele Parameter steuerbar, aber meist nur auf einfache Weise verwendet

Java-8-Streams für Map-Reduce-Aufgaben

Threadsicherheit von Datenstrukturen

threadsicher: Datenstruktur kümmert sich selbst um Synchronisation
→ wegen der Gefahr von Deadlocks nicht innerhalb von `synchronized` aufrufen

nicht threadsicher: Datenstruktur kümmert sich nicht selbst um Synchronisation
→ in nebenläufigen Anwendungen nur innerhalb von `synchronized` aufrufen

historische bedingt werden vorgefertigte Datenstrukturen unterschiedlich gehandhabt
→ Dokumentation lesen

Beispiele bezüglich `Map` (bei anderen Strukturen ist es ähnlich):

- `ConcurrentHashMap`: threadsicher, effizient bei vielen gleichzeitigen Zugriffen
- `HashMap`: nicht threadsicher, umfangreich, effizient bei wenig gleichzeitigem Zugriff
- `Collections.synchronizedMap(new HashMap(...))`: threadsichere Variante
- `java.util.concurrent`: viele Klassen für Datensammlungen und robuste Iteratoren

Praktische Vorgehensweise für nebenläufige Aufgaben

hauptsächlich davon leiten lassen, ob und wie Teilaufgaben voneinander abhängen:

- unabhängig: parallele Ströme, `Executor`, eventuell `Future`
- gemeinsame Daten, unabhängige Ausführungsreihenfolge: `ConcurrentHashMap`
- abhängig: viel höherer Aufwand, Aufgabe so aufzuteilen versuchen, dass unabhängig
- Abarbeitung in Phasen aufteilbar: `Phaser` (statt vordefiniertem `Executor`)
- Ist Nebenläufigkeit wirklich nötig? Sonst Sequenzialisierung der Aufgabe
- wenn es nicht anders geht, Monitor als grundlegendes Sprachkonzept verwenden

Faustregel: Synchronisation muss einfach gehalten werden

Synchronisation und objektorientierte Sicht

Unabhängigkeit und objektorientierte Strukturen passen nicht zusammen → Konflikte

Synchronisation verhindert Parallelausführung → vermindert Effizienz (bis Stillstand)

Vermeidung von Deadlocks häufig durch lineare Anordnung aller Objekte versucht
→ wenn `x` vor `y`, darf `synchronized(x)` nicht innerhalb von `synchronized(y)` vorkommen
→ damit zyklische Strukturen verhindert – in der Praxis sehr restriktiv, aber schwer zu vermeiden
→ manchmal empfohlen, Liveness-Probleme zu ignorieren, aber seltene Probleme sind gefährlich
→ vorzugsweise gut bekannte Lösungsansätze verwenden

Synchronisation und Subtyping schwierig
→ Client-kontrollierte History-Constraints sind zusätzlich nötig
→ Untertypen dürfen über `wait`, ... nicht stärker einschränken als Obertypen
→ Vererbungsanomalie nicht lösbar ohne Widerspruch zu Untertypbeziehungen

Erzeugen von Prozessen in einer Shell

Prozess = Ausführungseinheit des Betriebssystems (nicht innerhalb von Java)

Beispiele für Shell-Komandos:

```
java Test arg1 arg2
echo $PATH
cat file
cat <file
cat file >file2
cat file >>file2
cat file &>file2
cat file &>>file2
cat <file >file2
cat file | wc
cat <file | wc >file2
cat <file |& wc >file2
```

```
cat file &
cat file ; wc <file
cat file ; wc <file &
cat file >file2 && wc <file2
cat file || cat file2
cat *
cat */?/A*[a-zI].java
cat \*
cat 'a * 2'
cat "a * 2"
cat `cat file`
/usr/bin/cat ~username/file
```

Programmierung in einer Shell

```
export PATH=/home/me/bin:$PATH
echo ${PATH}

if ... ; then ... ; else ... ; fi
if test $x = $y; then echo $x; fi
if [ $x = $y ]; then echo $x; fi

for i in *.java; do javac $i; done
for i in *.java; do (javac $i &); done
for i in *.java; do (javac $i &>‘basename $i .java‘.out &); done
for f in ‘cat file‘; do javac $f.java; done
```

Dateien und I/O-Ströme in Java

Kommandozeilenargumente über `String[] args` in `main` an Java-Programm übergeben

Standardeingabe über `System.in` vom Typ `InputStream` lesbar,
Standardausgabe über `System.out` vom Typ `PrintStream` schreibbar,
Fehlerausgabe über `System.err` vom Typ `PrintStream` schreibbar

alle I/O-Ströme müssen nach dem Öffnen und Verwenden geschlossen werden (`close`);
die meisten Operationen können `IOException` auslösen, muss abgefangen werden
→ `try (FileReader r = new FileReader(p)) {...} catch (IOException e) {...}`

ungepufferte I/O für rasche Weiterleitung an Betriebssystem, gepufferte I/O effizienter;
durch `flush()` sofortige Weiterleitung, ohne `"\n"` häufig auch ungepuffert nötig

extern nur rohe Daten, intern I/O-Ströme für Zeichen (UTF-16) oder rohe Daten

hilfreich: `java.io.File` und `java.nio.file`

Serialisierung

Umwandlung zwischen internem und externem Format (Kodierung vs. Serialisierung)

`toString()` für Serialisierung ungeeignet (anderer Zweck, ineffizient)

Interface `Serializable` markiert Objekte, die automatisch serialisierbar sind

→ `writeObject(...)` in `ObjectOutputStream` schreibt Objekt
→ `readObject(...)` in `ObjectInputStream` liest Objekt
→ z. B. `new ObjectInputStream(System.in)`, `new ObjectOutputStream(System.out)`
→ als `static` oder `transient` deklarierte Variablen nicht berücksichtigt
→ setzt voraus, dass auf beiden Seiten gleiche Klassen vorhanden
→ `DataInputStream` und `DataOutputStream` für Austausch primitiver Daten
→ zahlreiche Klassen zur Unterstützung semistrukturierter Daten (XML, JSON, ...)

Zugriff auf Betriebssystem-Ressourcen

über `System.getEnv()` Shell-Variablen zugreifbar (`Map<String,String>`)

`Runtime.getRuntime` ermöglicht über Ergebnis Zugriff auf Java-Laufzeitumgebung

→ `availableProcessors()` liefert Anzahl nutzbarer Prozessor-Kerne

→ `freeMemory()` liefert Größe des verbliebenen Speichers

→ `gc()` empfiehlt Garbage-Collector, mehr Speicher freizugeben

→ `exec(...)` führt über Betriebssystem neuen Prozess (nicht Thread) aus

→ z. B.: `Process p = Runtime.getRuntime().exec("java -cp /home/me/java Test")`

über `p` vom Typ `Process` sind Verbindungen zum Prozess herstellbar

→ `p.getOutputStream()` liefert Strom, der mit Standardeingabe von `p` verbunden (Pipeline)

→ `p.getInputStream()` liefert Strom, der mit Standardausgabe von `p` verbunden

→ `p.getErrorStream()` liefert Strom, der mit Fehlerausgabe von `p` verbunden

→ `p.waitFor()` wartet auf Beendigung von `p`, liefert Return-Status

→ `p.destroy()` kann `p` abbrechen

Worauf bei Interprozesskommunikation zu achten ist

unendliches Warten auf Daten vermeiden, Kommunikationsverhalten genau analysieren

Daten müssen im gleichen Format gelesen und geschrieben werden (auch Kodierung)

auf verzögerte Datenübertragung durch Puffer achten, nötigenfalls `flush` verwenden

ausprobieren, ob gepufferte oder ungepufferte I/O effizienter, Intuition oft irreführend

`exec(...)` hat nicht die Mächtigkeit der Shell; Aufruf von Shell-Skripts möglich

Fehlersuche sehr aufwändig

→ Fehlermeldungen am einfachsten durch Schreiben in Datei sichtbar werden lassen

→ Weiterleitung von Fehlermeldungen möglich, Fehlerbehandlung aber sehr aufwändig

→ Kommunikationsverhalten möglichst einfach gestalten, sonst Übersicht schnell verloren

→ viel Ausprobieren nötig, auch um effiziente Ausführung zu erhalten