High-Level Konstrukte

Primitivoperationen sind gefährlich, kompliziert anzuwenden und oft ineffizient

Es gibt wiederkehrende Grundmuster:

- Fertige Frameworks
- Threadsichere Variable
- Threadsichere Container
- Flexiblere Locking-Mechanismen
- Botschaftenaustausch (insbesondere Aktoren und aktive Objekte)
- andere Mechanismen

Fertige Frameworks

Viele Threadanwendungen beziehen sich auf standardisierbare Anwendungen (Datenbanken, Serverapplikationen usw.).

Hier wird in der Regel durch Standardmechanismen (Transaktionen) für Threadsicherheit gesorgt. Probleme entstehen nur da, wo man eigene Mechanismen hinzufügt (Vorsicht: immer die Threadsicherheit prüfen!)

Ähnliches gilt auch für viele Bibliotheksklassen, die von Natur aus threadsicher sind.

Wichtige Sonderfälle: konstante Objekte (Wertobjekte).

Datenparallelität und ForkJoin

Konstrukt zur Programmierung von Datenparallelität in Java (Idee: rekursive Zerlegung)

```
public class Worker extends RecursiveTask<Double> {
    // Konstruktur und Variablen

   protected Double compute() {
       if (Problem ist zu groß) {
           Worker w1 = new Worker(ein Teil der Arbeit);
           w1.fork();
           Worker w2 = new Worker(der andere Teil der Arbeit);
           double r2 = w2.compute();
           double r1 = w1.join();
           return verknuepfe(r1, r2);
      }
      else
          return computeDirectly();
   }
}
```

Threadsichere Container

Die Containerklassen in java.util sind in der Regel nicht threadsicher (z.B. `HashMap`)!

Man kann aber für viele Fälle ganz einfach ein sicheres Objekt erzeugen:

`Collections.synchronizedMap`
`Collections.synchronizedList`
usw.

z.B.
```
Map<String, Person> m =
    Collections.synchronizedMap(new HashMap<String, Person>())
```

Geschützt sind allerdings nur die elementaren Operationen (put, get). Wenn mehrere Operationen zwingend zusammengehören, ist eine entsprechende Synchronisation nötig! Insbesondere muss die Anwendung eines Iterator per **`synchronized`** geschützt werden:

```
Set<String> s = m.keySet();
synchronized (m) {
    for (String x : m) ....
}
```

Passives Warten (Semaphore)

Semaphore sind eine atomare Operation, die sowohl für das Sperren kritischer Abschnitte, als auch für das Warten auf Bedingungen verwendet werden kann (wird daher auch in BS behandelt).

Beispiel: Semaphore (= Ampel). Edgar Dijkstra hat dieses Konzept entwickelt. Es basiert auf 2 Operationen P (= *proberen*, testen) und V (= *verhogen*, erhöhen).

P: wenn die Semaphore >0 wird sie erniedrigt, stellt aber keine Barriere dar. Ist sie <= 0, so muss der Thread warten. (häufiger Name: `acquire()`)

V: die Semaphore wird um 1 erhöht. Wenn dabei der Zähler auf 1 springt, wird ein wartender Thread freigegeben. (häufiger Name: `release()`)

Dieser Mechanismus heißt **zählende Semaphore** (Alternative: binäre Semaphore).

Eine Semaphore kann als einzige Synchronisationsprimitive verwendet werden. Das sollte man aber nicht tun – Sie ist da sinnvoll, wo der Zugang zu einenm Codeabschnitt von einer bestimmten Anzahl abhängt (s. Beispiel)

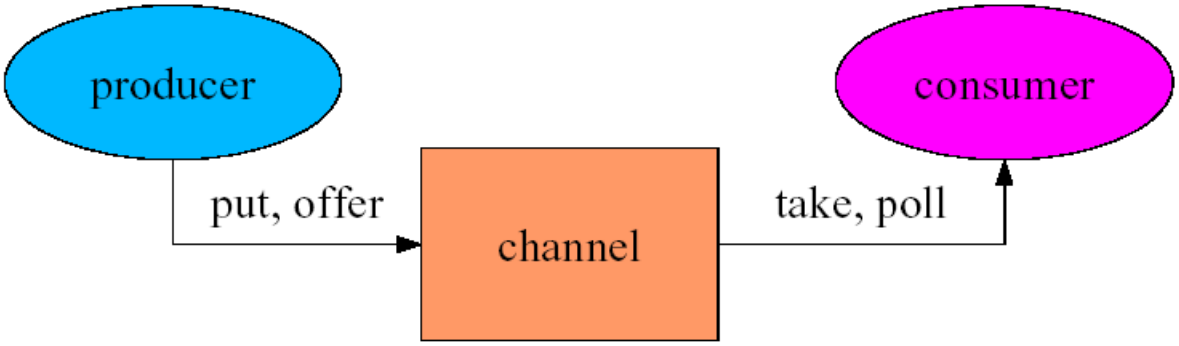
Anwendung: Bounded Buffer mit Semaphore

```
@ThreadSafe
public class BoundedBuffer<T> {
    private static final int SIZE = 10;
    private final Semaphore putSlots = new Semaphore(SIZE);
    private final Semaphore getSlots = new Semaphore(0);
    private final List<T> queue =
        Collections.synchronizedList(new ArrayList<T>(SIZE));

    public T get() throws InterruptedException {
        getSlots.acquire();
        T r = queue.remove();
        putSlots.release();
        return r;
    }

    public void put(T x) throws InterruptedException {
        putSlots.acquire();
        queue.add(x);
        getSlots.release();
    }
}
```

Wichtig: Beim Aufruf von `acquire()` darf der Thread keine Sperre besitzen!!!

Botschaftenkanäle entkoppeln Objekte

1. Kein gemeinsamer Speicher, also keine Race Conditions
2. Bewirkt gleichzeitig passives Warten auf Daten
3. Asynchrones Senden vermeidet Deadlocks
4. Future-Objekte ermöglichen auch asynchrones Empfangen
5. Das Konzept lässt sich auf verteilte Systeme übertragen



Botschaftenaustausch

Wir haben gesehen, dass das Problem der kritischen Abschnitte durch den Zugriff auf gemeinsame Variable / Objekte entsteht.

Funktionale Programmierung kennt keine gemeinsamen Variablen => Botschaftenmechanismus

Eine Möglichkeit der Organisation von Nebenläufigkeit besteht darin, diesen gemeinsamen Zugriff dadurch zu beseitigen, dass zwischen Threads auschließlich Botschaften verschickt werden.

Reiner Nachrichtenaustausch vermeidet Wettlaufbedingungen.
Nachrichtenaustausch hat weniger Deadlock-Probleme!

Historie des Botschaftenaustauschs

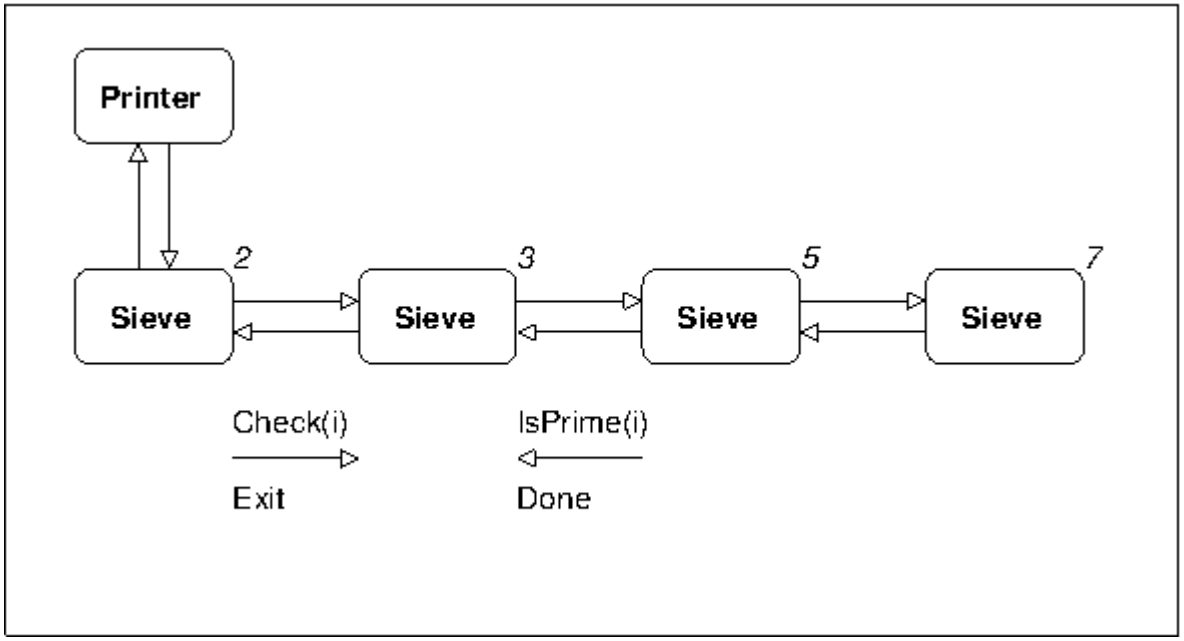
- Immense Komplexität von Nebenläufigkeit mit gemeinsamen Speicher
- Actor Modell (Hewitt 1973)
- CSP = communicating sequential processes (Hoare 1978)
- Botschaftenaustausch bei Parallelrechnern (z.B. SUPRENUM, MPI)
- Renaissance von gemeinsamen Speicher mit Java
- Actor-Modell wird mit Erlang und Scala populär (ab 2000)

Das Actor-Modell von Scala

Das Actor-Modell ist eine Variante des Botschaftenaustauschs. Es basiert auf den folgenden Annahmen:

- Anstelle von Threads fungieren Aktor (actor) genannte Objekte. Aktoren werden von außen nur über Nachrichten angesprochen.
- **Aktoren verfügen nicht über gemeinsame veränderliche Daten (in der funktionalen Variante haben sie keine veränderlichen Daten).**
- Die Kommunikation zwischen Aktoren findet über (unveränderliche) **Nachrichtenobjekte** statt.
- Aktoren verfügen über eine **Message-Queue** aus der sie die erwarteten Nachrichten auswählen.
- Liegt keine passende Nachricht vor, dann blockiert der Aktor.
- Das Versenden einer Nachricht ist nicht blockierend (**asynchron**). Der Empfänger kann später eine Antwort an den Absender zurücksenden.
- Es gibt synchrone **Nachrichten**, bei denen der Sender unmittelbar auf die Antwort wartet.
- Als Variante kann die synchrone Antwort in der Referenz auf ein **Future-Objekt** bestehen.
- Es gilt als guter Stil, wenn das Versenden und Empfangen von Nachrichten in Methoden gekapselt ist (**Muster des aktiven Objekts**).

Beispiel für Funktionale Parallelität (Pipeline)



Das Beispiel ist sehr instruktiv (auch wenn es kein guter Algorithmus ist)

Paralleles Primzahlsieb

```
class Sieve(val pred: Actor, val prime: Int) extends Actor {
  def act {
    var next: Option[Actor] = None
    while(true) receive {
      case Check(n) => if (n % prime != 0) next match {
          case Some(nxt) => nxt ! Check(n)
          case None =>
            pred ! IsPrime(n)
            val nxt = new Sieve(self, n)
            nxt.start()
            next = Some(nxt)
          }
      case IsPrime(n) =>
          pred ! IsPrime(n)
      case Exit => next match {
          case Some(nxt) => nxt ! Exit
          case None =>
            pred ! Done
            exit
          }
      case Done =>
          pred ! Exit
          exit
    }
  }
}
```

Bewertung des Nachrichtenmodells

- keine gemeinsamen Variablen
- Nachrichten sind in Objekte gekapselt: es können beliebige unveränderliche Objekte übertragen werden.
- Empfangen sucht eine passende Nachricht aus (partiell definierte Funktion)
- Wenn kein case-Muster passt, wird gewartet.
- Es wird immer nur eine Nachricht bearbeitet.
- Mittels reply(Ausdruck) kann eine Antwort gesendet werden
- `!` bedeutet Senden
- `!?` bedeutet Senden und Warten auf reply-Antwort
- `!!` bedeutet Senden und Rückgabe eines Future-Objekts, mit dem man später die Antwort erhält
- Falsche Nachrichten werden nie bearbeitet – keine Prüfung durch Compiler !!
- daher Muster „Aktives Objekt“ (Nachrichtenaustausch gekapselt)

Aktive Objekte

- haben eine Methodenschnittstelle wie andere Objekte
- Methoden werden in einem eigenen Thread ausgeführt
- Methoden ohne Rückgabe kehren sofort zurück
- Methoden mit Rückgabe kehren in der Regel sofort zurück
- Evtl. wird ein Future-Objekt zurückgegeben

Concurrent Buffer in Scala (Aktives Objekt)

```
import scala.actors.Actor._
Import scala.collection.mutable

class Buffer[T](n: Int) {
    private case class Put(x: Any)          // Nachrichtenobjekte
    private case object Get
    private case object Size

    val b = actor {
        val q = new mutable.Queue[T]
        while (true) receive {              // partiell definierte Funktion
            case Put(x) if q.size < n       // Muster + Guard
                => q.enqueue(x)             // Aktion
            case Get if ! q.isEmpty
                => reply(q.dequeue())
            case Size   => reply(q.size)
        }
    }

    def put(x: T): Unit = b ! Put(x)                 // asynchrones Senden
    def get: T = (b !? Get).asInstanceOf[T]          // synchrones Senden
    def size: Int = (b !? Size).asInstanceOf[Int]    //    und Empfangen
}
```

Futures

- Unter Futures versteht man Objekte, die die Ergebnisse einer erst in der Zukunft abgeschlossenen Berechnung transportieren.
- Futures ermöglichen es, dass eine in einem anderen Thread ausgeführte Methode unmittelbar ein Ergebnis zurückgibt.
- Erst beim Zugriff auf den Ergebniswert muss (evtl.) gewartet werden.
- Mittels Futures lassen sich Blockierungen und Deadlocks vermeiden.
- Futures bilden eine Grundlage für das Muster „aktiver Objekte“
- Scala: !! (anstelle von !?) liefert ein Future-Objekt
- Java: submitt (für einen Executor-Service) liefert ein Future-Objekt

Zugriff auf zukünftig eintreffende Werte

```
class Buffer[T](n: Int) {
    case class Put(x: Any)
    case object Get

    val b = actor {
        val q = new collection.mutable.Queue[T]
        while (true) receive {
            case Put(x) => q.enqueue(x)
            case Get    => reply(q.dequeue())
        }
    }
    def put(x: T): Unit = b ! Put(x)
    def get() = (b !! Get).asInstanceOf[Future[T]]
}
```

Get liefert Referenzen auf Ergebnisse, die erst in der Zukunft eintreffen (Subskription)

```
val futureResult = q.get()       // antwortet sofort
...
val result = futureResult()      // wartet auf Antwort
```

Callables und Futures in Java

Erlauben Methodenaufrufe in separaten Threads auszuführen.

Oft im Zusammenhang mit weiteren Bibliotheksklassen (Executors, ExecutorService, FutureTask).

```
public static void main(String[] args) throws InterruptedException {
    ExecutorService exe = Executors.newCachedThreadPool();
    Future<Integer> result = exe.submit(
        new Callable<Integer>() {
            public Integer call() throws Exception {
                System.out.println("computing");
                return 4;
            }
        });
     System.out.println("main");
     try {
         System.out.println(result.get()); // wartet (InterruptedExc.)!!
     }
     catch (ExecutionException e) {
         e.getCause().printStackTrace();
     }
}
```

Hier landen alle Exceptions im aufrufenden Thread !!

Phasenbezogene Mechanismen

Vielen nebenläufigen Lösungen ist gemeinsam, dass es immer bestimmte Punkte gibt, an denen die Abläufe mehrerer Threads abgestimmt werden müssen:

Ein oder mehrere Threads warten darauf, dass eine Anzahl Threads einen bestimmten Zustand erreicht haben.

`CountDownLatch`

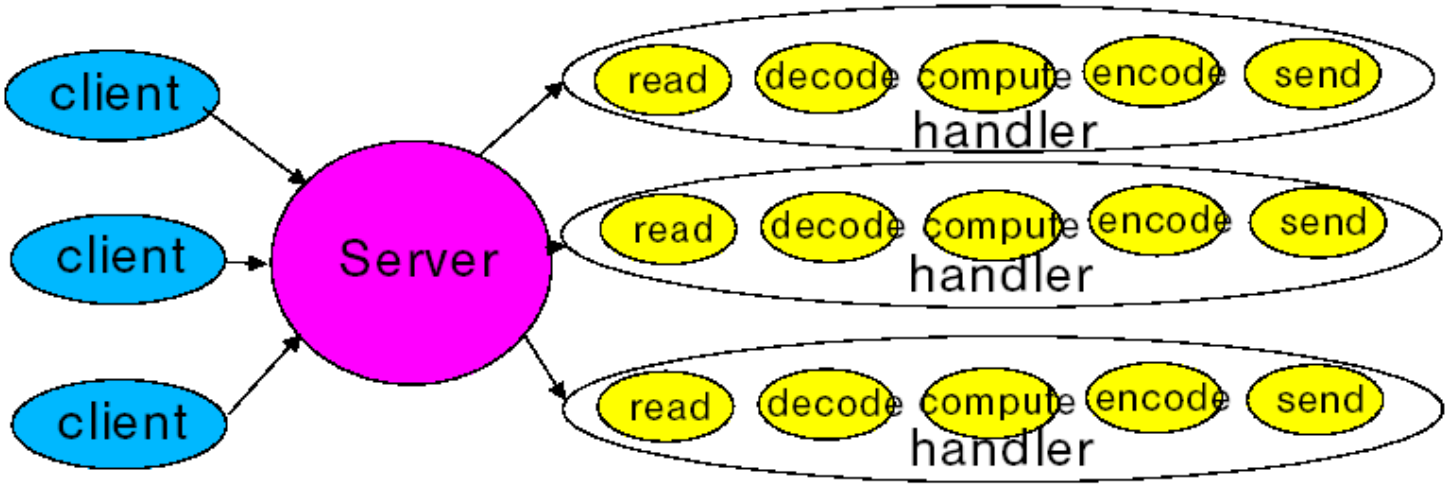
Zwei Threads synchronisisieren sich an bestimmten Stellen und tauschen dabei Daten aus.

`Exchanger`

Mehrere Threads synchronisieren sich immer wieder. Unmittelbar vor dem Fortführen der Berechnung kann eine Aktion ausgeführt werden.

`CyclicBarrier`

Paralleler Server



Handler sind Threads, die bei Bedarf (geht vom Client aus) vom Server zur Erledigung einer Aufgabe gestartet werden. Sie übernehmen nach dem Start die restliche Kommunikation mit dem Client. Die Handler haben untereinander keine direkte Kommunikation. Sie benutzen oft gemeinsame Objekte (z.B. Datenbank)

Verwaltung von Threadpools

Problem:
· neuer Thread = Objekterzeugung + Start des Threads
· der Start eines Threads ist langsam (Betriebssystem, Ressourcen)
· die Anzahl der Threads ist durch das Betriebssystem begrenzt

Effizente Programme vermeiden die Ausführung von Betriebssystemaktionen wo immer es geht!

=> es ist sinnvoll, dass Threads nicht für jede Aufgabe neu angelegt werden, sondern sich immer wieder neue Aufgaben holen (s. Praktikum)

die Java-Bibliothek unterstützt die Verwendung von Threadpools durch eigene Schnittstellen (`Excecutor`, `ExecutorService`) und durch eigene Klassen (s. Fabrikmethoden in `Executors`)

Server-Beispiel: eine (feste) Anzahl von Threads holt sich anstehende Handler-Objekte, bearbeitet den Auftrag und holt sich das nächste Objekt

Scala hat sogar als Default eine eigene Threadverwaltung für Actors.

In beiden Fällen kann man eigene Klassen hinzufügen.

Besonderheit: Auffangen von Exceptions

Exceptions werden „normal“ im Stack des Threads behandelt.

Nicht aufgefangene Exceptions führen zum Abbruch des Threads.

Es kann festgelegt werden, was mit nicht aufgefangenen Exceptions geschehen soll:

- mittels ThreadGroups (eher obsolet)
- mittels `Thread.setDefaultUncaughtExceptionHandler( UncaughtExceptionHandler e)`
- Für ein Thread-Objekt `t` mittels `t.setUncaughtExceptionHandler(UncaughtExceptionHandler e)`

```
public interface UncaughtExceptionHandler (
    public void uncaughtException(Thread t, Throwable e);
}
```

Aber: in beiden Fällen werden die Exceptions immer noch in dem entsprechenden Thread ausgeführt !

Terminierung von Threads

Wenn der Java-Prozess endet, werden automatisch alle Threads beendet.

Ansonsten dürfen Threads nicht von außen „abgewürgt“ werden.

Threads sollen sich selbst beenden, entweder aufgrund eigener Logik oder infolge von `interrupt()`.

`main()` terminiert erst, wenn alle wichtigen Threads beendet sind.

Threads, mit „dienenden“ Aufgaben für andere Threads können mit `setDaemon(true)` zu daemons (deutsch = Sklave) erklärt werden, auf die `main()` nicht zu warten braucht.

Wenn `notify()` verwendet wird und `interrupt()` vorkommt, muss darauf geachtet werden, dass kein Signal verloren geht!