

# Programmieren II

## Threads



Heusch 8 (2.Bd)  
Ratz 18

Institut für Automation und angewandte Informatik

```
final List<String> allResults = new ArrayList<String>();  
final Map<String, Integer> typeWordResultCount = new HashMap<String, Integer>();  
final Map<String, Integer> typePoints = new HashMap<String, Integer>();  
evaluation.put(type, typePoints);  
  
for (final Sheet sheet : this.sheets) {  
    final String sheetResult = sheet.getPlayerInput(type);  
    if (sheetResult.startsWith(start) && this.isValidWord(sheetResult, type)) {  
        validWordCountForType++;  
        allResults.add(sheetResult);  
    }  
}
```

# Parallelität

- Im Wortsinne: „Gleichartige Beschaffenheit“.
- Rechner ermittelt automatisch, ob einzelne Abarbeitungselemente (etwa: Prozeduren, Befehle, Maschinen-Instruktionen) „gleichzeitig“ ausgeführt werden können und führt dies, falls möglich, für den Anwender und Programmierer transparent durch.
- Voraussetzung: Existenz logisch trennbarer Aktivitäten.
- Tatsächliche Abarbeitungsreihenfolge kann von der „codierten“ abweichen!
- Auf Maschinen mit nur einer einzelnen CPU lassen sich im allgemeinen nur quasi-parallele Abläufe realisieren. Echter Parallelismus erfordert mehr als eine CPU bzw. entsprechende Recheneinheiten („Kerne“).
- Parallelverarbeitung kann auch die Verarbeitung auf mehreren physisch getrennten vollständigen Maschinen (d.h. mit eigenen CPU, Speicher, E/A-Einheiten) umfassen.

# Nebenläufigkeit (Concurrency)

- Parallele Ausführung von Anweisungen auf einem oder mehreren Prozessoren bzw. Prozessorkernen.
- Organisation erfolgt durch Programmierer in geeigneter Hochsprache.
- Nebenläufige Ausführung ist auf eine einzige physische Maschine beschränkt, die jedoch mehrere vollständige CPUs enthalten kann.
- Bei Java-Threads handelt es sich daher um eine Möglichkeit der nebenläufigen Ablaufsteuerung, da diese explizit (konkret: in Form von API-Aufrufen) durch den Applikationsprogrammierer festgelegt wird:
  - Codierter Kontrollfluss  $\leftrightarrow$  vom Programmierer nicht beeinflussbarer Nichtdeterminismus.
  - Reihenfolge der Ausführung von Anweisungen obliegt dem Prozessorzeit zuteilenden Betriebssystem.
- Unterscheidung zwischen Parallelität und Nebenläufigkeit nicht eindeutig.

# Prozess

- Im Speicherzugriff befindliches ablauffähiges Programm mit seinen dafür notwendigen betriebssystemseitigen Datenstrukturen wie zugeordneten Eigenschaften (z.B. Stack- und Programmzähler, Prozesszustand, sowie Eigenschaften der Speicher- und Dateiverwaltung).
- Ein Prozess wird durch das Betriebssystem als eigenständige Instanz – in der Regel unabhängig und geschützt von anderen – ausgeführt.
- Multitaskingsysteme (praktisch alle heutigen Betriebssysteme) gestatten die parallele Prozessausführung.

# Thread (1)

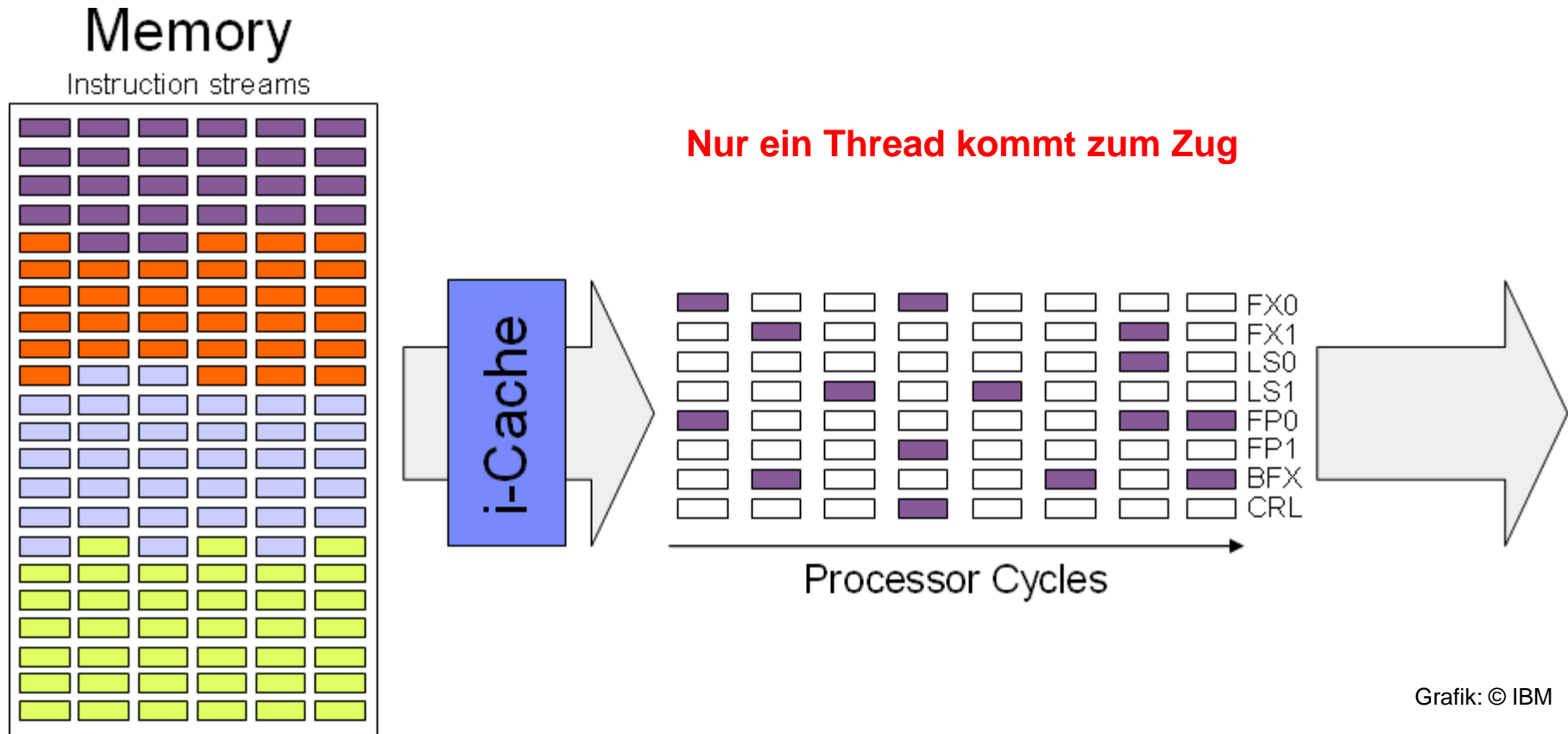
- Ein Thread (engl. für „Faden“) stellt eine nebenläufige Ausführungseinheit innerhalb genau eines Prozesses dar.
- Aufgrund dieser Definition erben Threads viele der prozesstypischen Eigenschaften, wie Zustand, Programmzähler etc.
- Den Hauptunterschied zu voll entwickelten Prozessen bildet der zwischen allen Threads eines Prozesses **geteilte Speicher**.
- Gleichzeitig besitzt jeder Thread seinen **eigenen lokalen Speicherbereich** in dem u.a. die lokalen Variablen verwaltet werden.
- Aus diesen Gründen werden Threads oft auch als **leichtgewichtige Prozesse** („Lightweight Processes“) bezeichnet.
- Alle Threads bewegen sich im Ausführungskontext des erzeugenden Prozesses. „Externe Operationen“ wie Plattenzugriffe etc. wirken sich über Threadgrenzen hinaus (auf den gesamten Prozess) aus.

## Thread (2)

- In Multithreading-Systemen besitzt jeder ablaufende Prozess mindestens einen Thread, der den Kontrollfluss realisiert.
- Multitasking und Multithreading bedingen sich daher gegenseitig. Ohne die systemseitige Unterstützung der parallelen Taskausführung kann keine prozessinterne Threadabarbeitung erfolgen.

# Unterstützung durch Prozessoren (1)

## ■ Beispiel POWER 5, ohne Multithreading

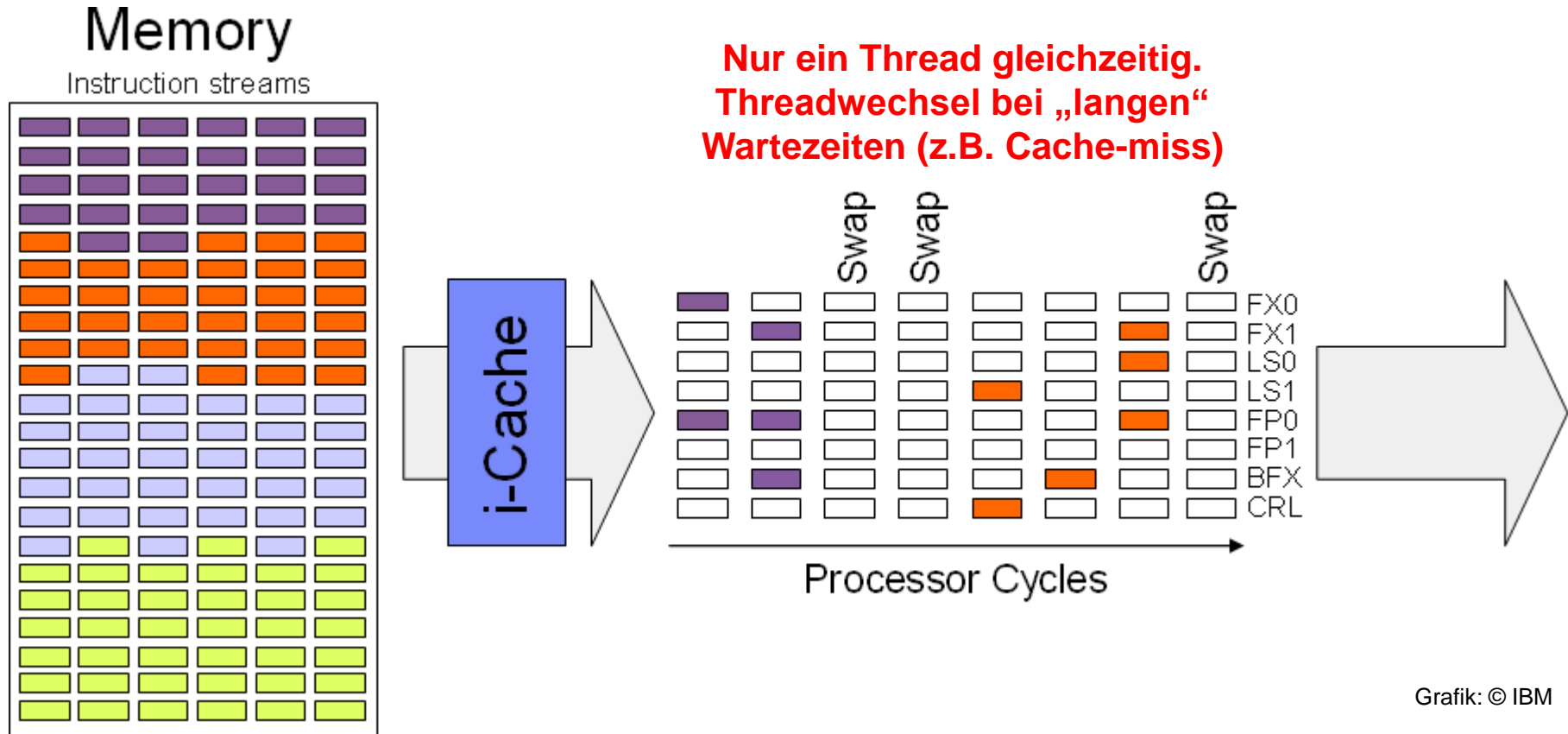


Grafik: © IBM

Ausführungs-Einheiten: FX=Festkomma (2x); LS=Load/Store (2x); FP=Gleitkomma (2x); BFX=Verzweigung (Branch Execution); CRL=Logik

# Unterstützung durch Prozessoren (2)

## ■ Beispiel POWER 5, grobkörniges Multithreading



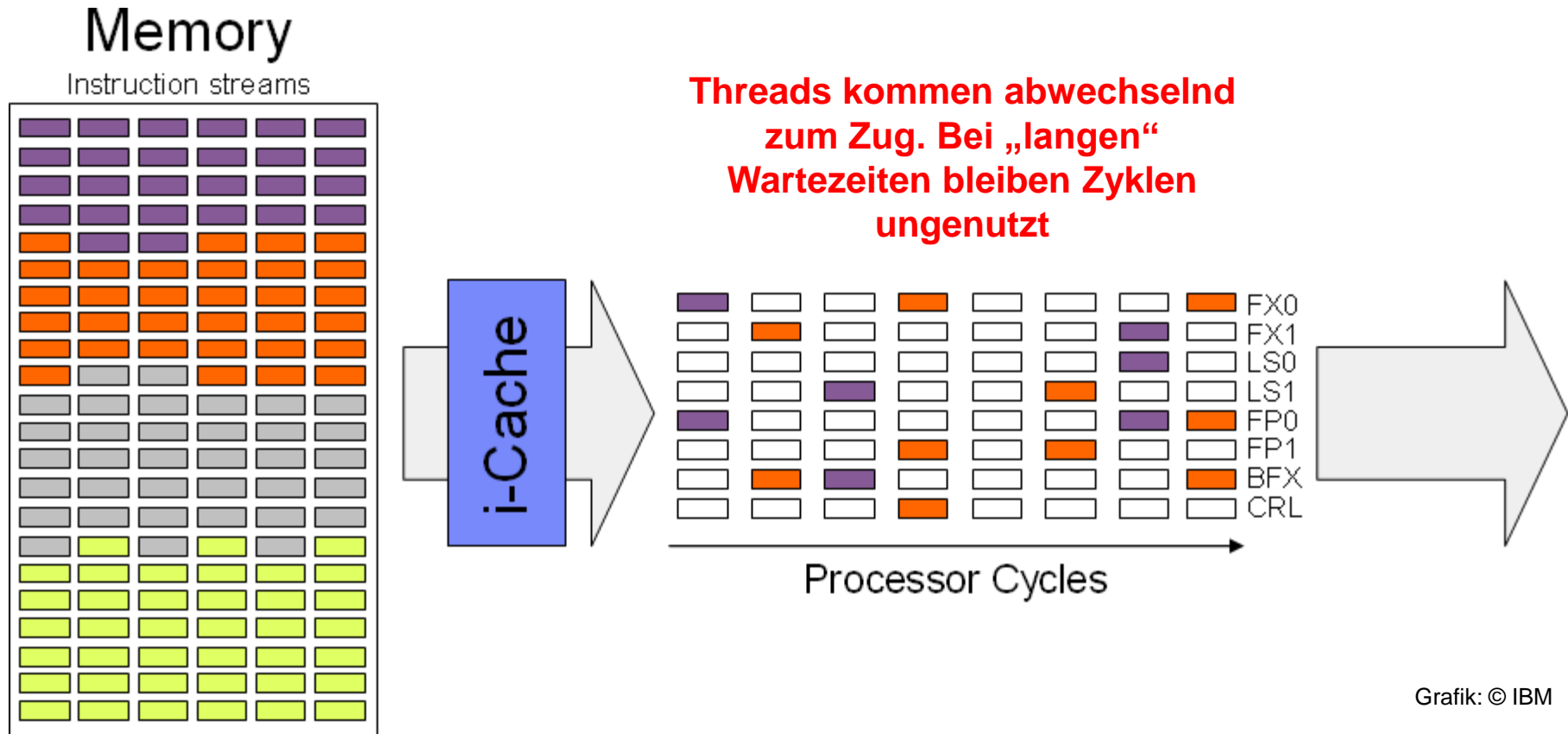
Grafik: © IBM

Ausführungs-Einheiten: FX=Festkomma (2x); LS=Load/Store (2x); FP=Gleitkomma (2x); BFX=Verzweigung (Branch Execution); CRL=Logik



# Unterstützung durch Prozessoren (3)

## ■ Beispiel POWER 5, feinkörniges Multithreading



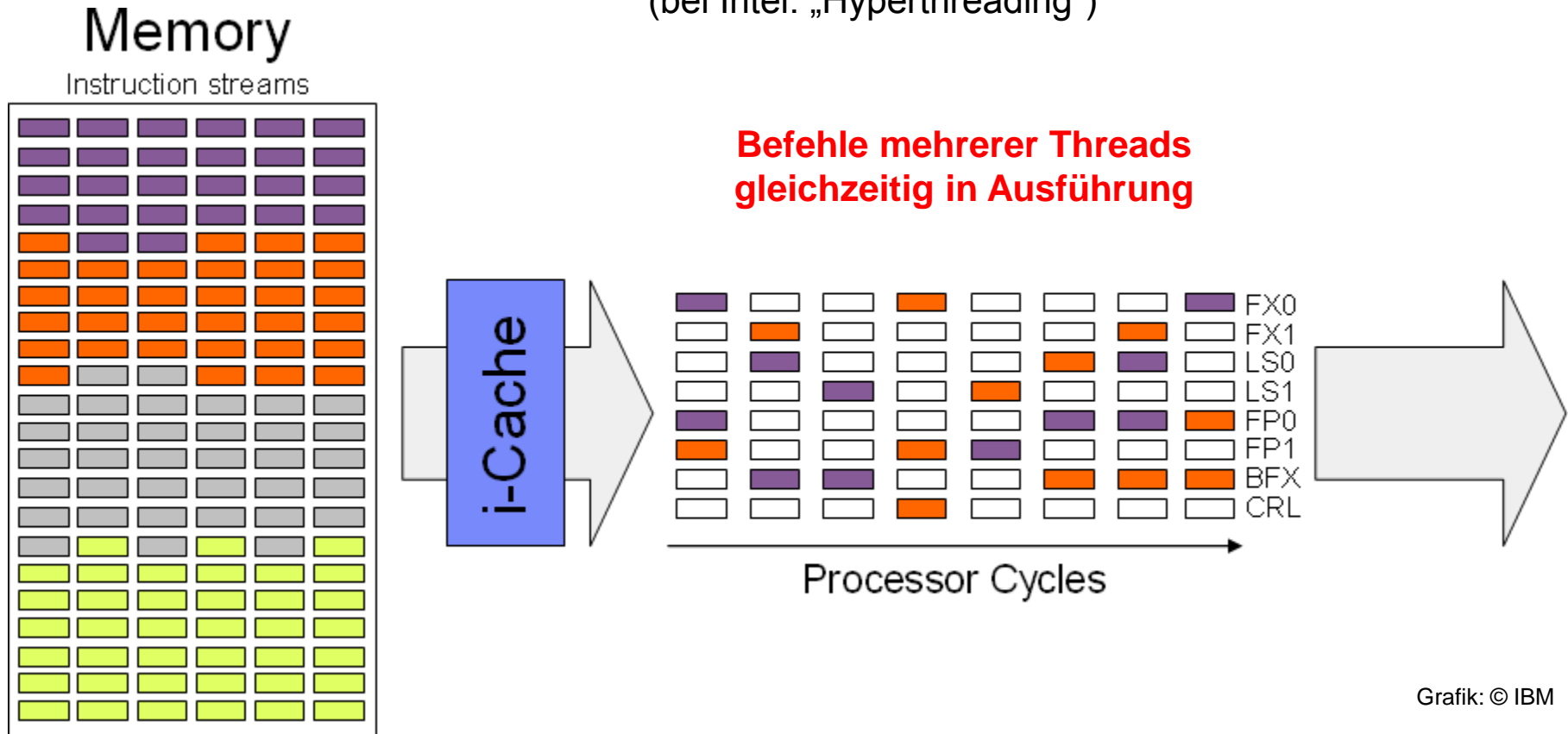
Grafik: © IBM

Ausführungs-Einheiten: FX=Festkomma (2x); LS=Load/Store (2x); FP=Gleitkomma (2x); BFX=Verzweigung (Branch Execution); CRL=Logik

# Unterstützung durch Prozessoren (4)

## ■ Beispiel POWER 5, Simultaneous Multithreading SMT

(bei Intel: „Hyperthreading“)

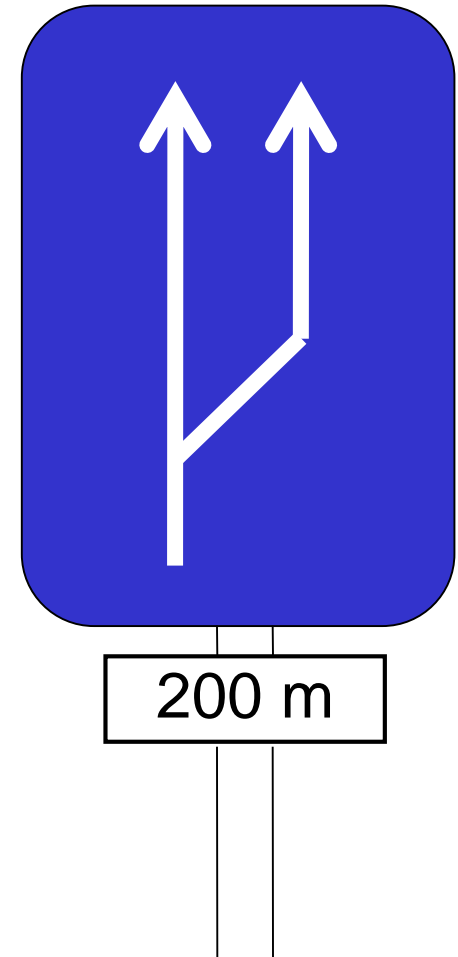


Grafik: © IBM

Ausführungs-Einheiten: FX=Festkomma (2x); LS=Load/Store (2x); FP=Gleitkomma (2x); BFX=Verzweigung (Branch Execution); CRL=Logik

# Warum mit Threads programmieren?

- Threads nutzen die vorhandenen Ressourcen moderner Betriebssysteme und Hardware effizienter aus, beispielsweise durch Weiterführung der Berechnungsoperationen, während andere Threads auf Ein-/Ausgaben warten.
- Threads werden durch ein geeignetes Betriebssystem für den Programmierer und Applikationsanwender transparent auf mehrere Prozessoren (falls vorhanden) verteilt.
- Threads erhöhen den Interaktionskomfort für den Applikationsanwender (Beispiel: GUI-intensive Anwendungen).
- Berechnungsaufwendige Arbeiten können „in den Hintergrund“ verlagert werden.



# Warum Threads über Java-API und JVM?

- Hochsprachenschnittstelle, da in Java kein direkter Hardware- und Betriebssystemzugriff möglich.
- Abstraktionsschicht von Thread-Implementierung in Java Virtual Machine.
- Erleichterung und dadurch Forcierung nebenläufiger Programmierung.
- Portabilität Thread-basierter Anwendungen auf andere Hardwareplattformen, die Java unterstützen.

## Und die Performanz?

- Die Verwendung von Threads führt (auf einem Prozessor) im technischen Sinne nicht zu einer beschleunigten Ausführung.
  - Im Gegenteil, durch Verwaltungsaufwände bei der Erzeugung und Koordination ergibt sich im allgemeinen sogar eine insgesamt vergrößerte Ausführungszeit.
  - Jedoch bedingt die effizientere Ressourcennutzung die bessere Auslastung der vorhanden Hardware und gleichzeitig entsteht für den Anwender der Eindruck einer flüssigeren Verarbeitung.
- Erst beim Einsatz mehrerer Prozessoren in einer Maschine ergibt sich ein echter positiver Laufzeiteffekt durch die Möglichkeit, Threads auf verschiedenen CPUs zur Ausführung zu bringen. Die Verteilung und Koordination obliegt hierbei dem Betriebssystem.

# API-Unterstützung zur Threadprogrammierung

- Zwei generelle Ansätze:
  - (Spezialisierende) Ableitung von der Klasse `Thread`
  - Implementierung der Schnittstelle `Runnable`
- Der erste Fall lässt sich auf den zweiten zurückführen, da die Klasse `Thread` selbst die Schnittstelle `Runnable` implementiert.

# Implementierung nebenläufiger Abläufe in Java

- Jeder nebenläufige Programmfaden wird generell durch eine eigenständige Klasse repräsentiert, welche die Schnittstelle `Runnable` implementiert.
- Die Schnittstelle `Runnable` definiert als einzige Methode `run()`, welche durch das Laufzeitsystem automatisch zu Beginn der Thread-gestützten Verarbeitung zur Ausführung gebracht wird.
- Daher sollte diese Methode niemals direkt auf einem `Thread`-Objekt aufgerufen werden, sondern jeder Thread ausschließlich mit der dafür vorgesehenen Methode `start()` dem Laufzeitsystem als rechenwillig gemeldet werden.

# run-Methode

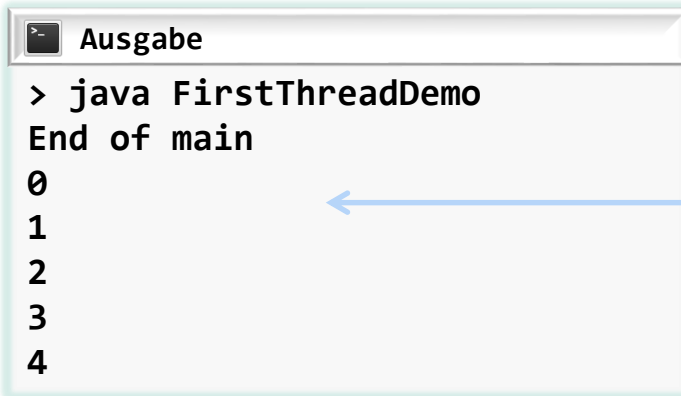
- Die `run()`-Methode wird nach der Threaderzeugung (Aufruf von `start()`) automatisch durch das Laufzeitsystem asynchron ausgeführt.
- Beispiel:

```
class FirstThread extends Thread {  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            System.out.println(i + " ");  
            try {  
                Thread.sleep(100); // 100 Millisekunden warten  
            } catch (InterruptedException e) {  
                System.out.println(e);  
            }  
        }  
        System.out.println("End of thread " + this.toString());  
    }  
}
```



# Beispiel (Fortsetzung)

```
public class FirstThreadDemo {  
  
    public static void main(String args[]) {  
        FirstThread thread = new FirstThread();  
        thread.start();  
        System.out.println("End of main");  
    }  
  
}
```



```
> java FirstThreadDemo  
End of main  
0  
1  
2  
3  
4
```

**i Reihenfolge kann abweichen!**

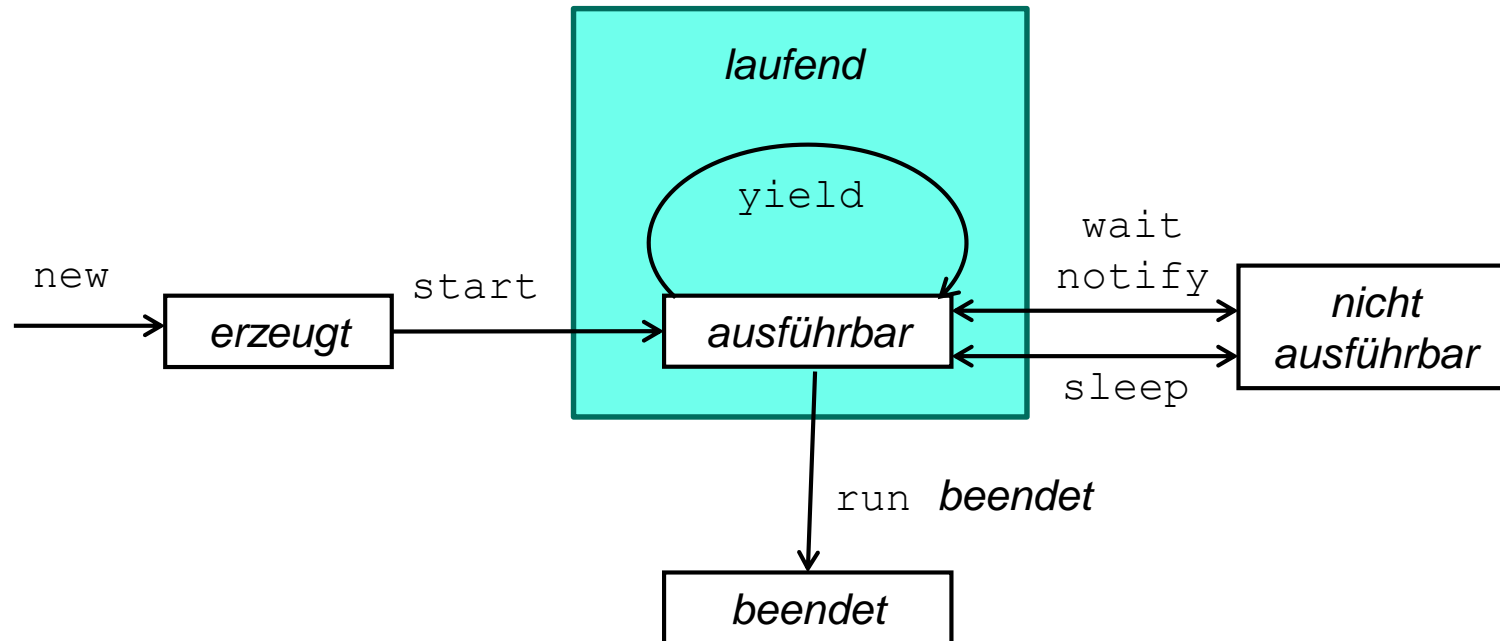
# Methoden von Thread (Auswahl) (1)

- `static Thread currentThread()`  
Referenz zum gerade ausgeführten Thread-Objekt.
- `ThreadGroup getThreadGroup()`  
Liefert die `ThreadGroup`, zu der dieser Thread gehört.
- `static boolean holdsLock(Object obj)`  
Hält dieser Thread einen Monitor (s. Synchronisation) auf `obj`?
- `void interrupt()`  
Unterbricht diesen Thread.
- `boolean isAlive()`
- `void join()`  
Wartet auf das Beenden dieses Threads.
- `void setDaemon(boolean on)`  
Markiert diesen Thread als Daemon oder User-Thread.
- `static void sleep(long millis)`  
Legt diesen Thread für `millis` Millisekunden „zum Schlafen“.

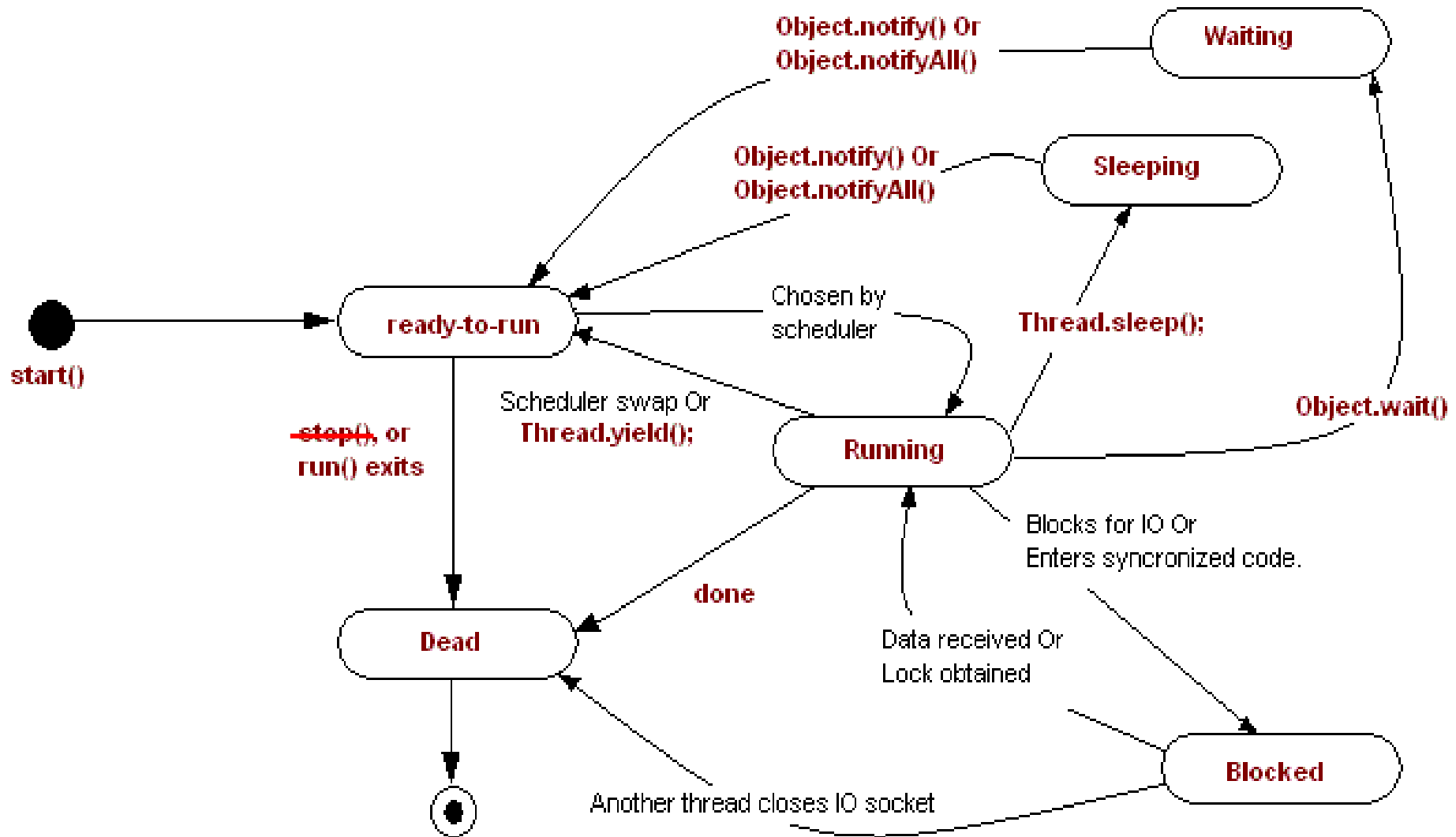
## Methoden von Thread (Auswahl) (2)

- `void start()`  
Startet die Ausführung dieses Threads. Ruft asynchron die Methode `run()` auf.
- `void run()`  
Wird von `start()` aufgerufen. Wird die Methode nicht überschrieben, dann tut der Thread nichts.
- `static void yield()`  
Lässt den gerade ausgeführten Thread (kurz) pausieren und gibt anderen Threads eine Chance zur Ausführung.
- Nicht mehr verwenden (**deprecated!**):
  - `destroy()`
  - `resume()`
  - `stop()`
  - `suspend()`

# Lebenszyklus eines Java-Threads (Grundschemata)



# Lebenszyklus eines Java-Threads (im Detail)



# Die Klasse ThreadGroup

- Eine ThreadGroup repräsentiert eine Menge von Threads.
- Neben Threads, kann eine ThreadGroup auch weitere ThreadGroups beinhalten. Insofern formen ThreadGroups einen Baum, in dem bis auf die Wurzel jedes Element einen „Vater“ hat.
- Ein Thread kann auf Informationen innerhalb der eigenen ThreadGroup zugreifen, jedoch weder auf Informationen der „Vater“-ThreadGroup noch auf die fremder ThreadGroups.

# Methoden von ThreadGroup (Auswahl)

- `int activeCount()`  
„Geschätzte“ Zahl aktiver Threads in dieser ThreadGroup.
- `int activeGroupCount()`  
„Geschätzte“ Zahl aktiver Gruppen in dieser ThreadGroup.
- `void destroy()`  
Zerstört diese ThreadGroup inkl. aller ihrer Untergruppen.
- `int enumerate(Thread [] list, boolean recurse)`  
Kopiert alle enthaltenen Threads in das Array list.
- `int enumerate(ThreadGroup[] list, boolean recurse)`  
Kopiert alle enthaltenen Unter-ThreadGroups in das Array list.
- `ThreadGroup getParent()`  
Liefert die „Vater“-ThreadGroup (falls vorhanden)
- `boolean parentOf(ThreadGroup g)`  
Testet, ob diese ThreadGroup „Vater“ der Gruppe g ist.

# Beispiel: Digitale Uhr (1)

```
import javax.swing.*;
import java.util.Date;

public class ClockApp extends JFrame implements Runnable {

    Date theDate;
    Thread runner;
    JTextField dateField = new JTextField();

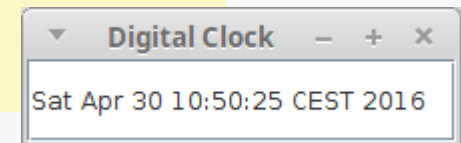
    public ClockApp() {
        super("Digital Clock");
        this.add(this.dateField);
        // Aus Runnable ein Thread-Objekt erzeugen
        this.runner = new Thread(this);
        // Keine Abfrage auf runner==null, da in Anweisung zuvor erzeugt
        this.runner.start();
    }

    // ... weiter auf der nächsten Folie
```

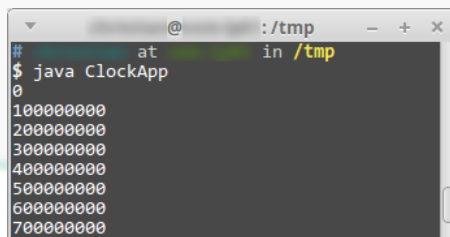


# Beispiel: Digitale Uhr (2)

```
public void run() {
    while (true) {
        try {
            this.theDate = new Date();
            this.dateField.setText(this.theDate.toString());
            Thread.sleep(1000);
        } catch (InterruptedException ex) { }
    }
}
```



```
public static void main(String args[]) {
    ClockApp cap = new ClockApp();
    cap.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    cap.setSize(220, 70);
    cap.setVisible(true);
    for (long i = 0;; i++) {
        if (i % 100000000 == 0) { System.out.println(i); }
    }
}
```



**Parallele Ausgabe auf Konsole**

# Synchronisation von Threads (1) - join

- Die Methode `join()` erlaubt es einem Thread, auf die Beendigung eines anderen Threads (unten: `t`) zu warten. Wenn also `t` ein (parallel laufendes) Thread-Objekt ist, sorgt

```
t.join();
```

dafür, dass der aktuelle Thread (der, in dessen Programmcode diese Zeile steht) so lange pausiert, bis `t` terminiert.

- Die Methode `join()` kann überladen werden, hängt aber (auch) vom Scheduling des Betriebssystems ab.
- `join()` **wirft eine** `InterruptedException`.

# Synchronisation von Threads (2) - Beispiel

```
class Name {  
  
    String firstname;  
    String surname;  
  
    public void setName(String fn,  
                        String sn) {  
        this.firstname = fn;  
        this.surname = sn;  
    }  
  
    public String getName() {  
        return this.firstname + " "  
               + this.surname;  
    }  
}
```

Problem:

```
Name n = new Name();  
// ---- thread 1 ----  
while(true){  
    n.setName("Hillary","Clinton");  
    n.setName("Donald","Trump");  
}  
  
// ---- thread 2 ----  
while(true) {  
    System.out.println(n.getName());  
}
```

Mögliche Resultate für `n.getName()` :

|                   |                 |
|-------------------|-----------------|
| "Hillary Clinton" | "Donald Trump"  |
| "Donald Clinton"  | "Hillary Trump" |

Ursache:

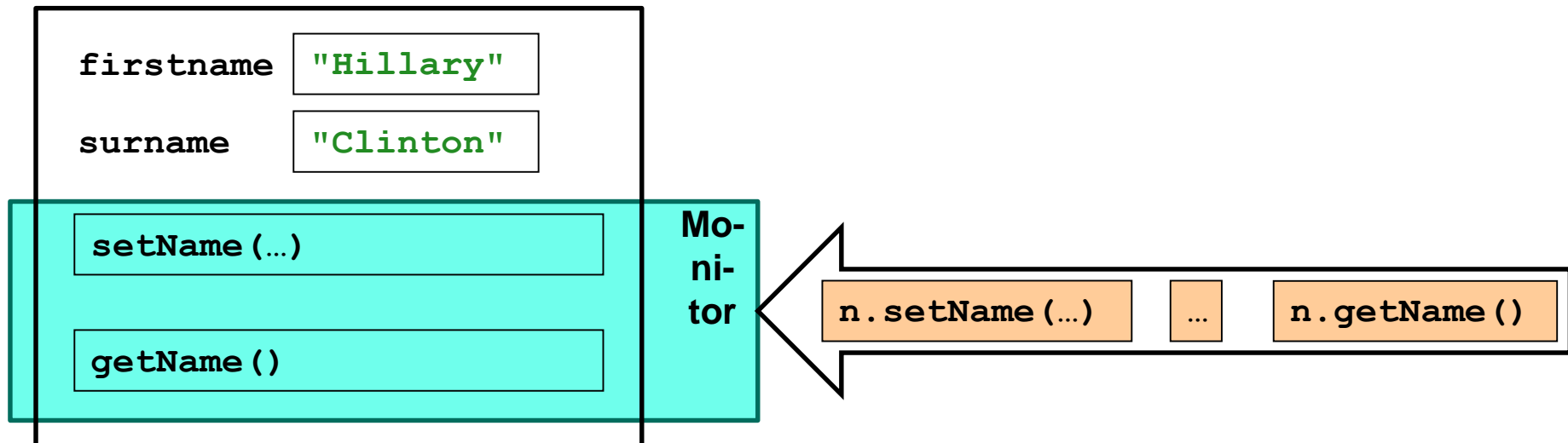
```
this.firstname = fn;  
this.surname = sn;
```

```
return this.firstname + " "  
       + this.surname;
```

sind unterbrechbare (teilbare) Operationen

# Synchronisation von Threads (3) – Monitore

- Lösung: Synchronisation mit Monitoren.
- Ein Monitor überwacht den Zugriff auf ein Objekt
  - Er sichert, dass bei Ausführung einer `synchronized`-Methode das zugehörige Objekt für andere `synchronized`-Methoden gesperrt ist.
  - Hierzu verwaltet er eine Sperre.
  - Ein Thread muss vor Ausführung einer `synchronized`-Methode die Sperre des Objekts erwerben.



# Synchronisation von Threads (4) – Beispiel

## ■ Synchronisation mit Monitoren

```
class Name {  
    String firstname;  
    String surname;  
  
    public synchronized void setName(String fn, String sn) {  
        this.firstname = fn;  
        this.surname = sn;  
    }  
  
    public synchronized String getName() {  
        return this.firstname + " " + this.surname;  
    }  
}
```

setName und getName sind unteilbare Operationen geworden, d.h. solange eine Routine ausgeführt wird, kann keine andere synchronized-Routine ausgeführt werden

# Synchronisation von Threads (5) – Beispiel

## ■ Syntax für Synchronisation mit Monitoren

```
synchronized Type method(/*...*/) {  
    // der gesamte Rumpf ist  
    // synchronisiert mit "this"  
    // ...  
}
```

**Komplette Methode**

*oder*

```
synchronized ( e ) {  
    // synchronized statement:  
    // Dieser Block wird  
    // synchronisiert  
    // mit dem Objekt, das durch  
    // "e" referenziert wird ...  
}
```

**Code-Block**

# Synchronisation mit `wait` und `notify` (1)

## ■ Beispiel (Erzeuger/Verbraucher-Problem):

Ein Erzeuger schreibt Daten in einen *Puffer*, ein Verbraucher leert ihn.

## ■ Mögl. *Problemsituationen*:

*Erzeuger* möchte Daten in den Puffer *schreiben*, dieser ist aber noch nicht vom Verbraucher geleert. *Verbraucher* möchte Daten aus dem Puffer *lesen*, der ist aber noch nicht gefüllt.

## ■ Lösungsansatz:

*Zugang* zum Puffer abhängig machen vom *Zustand* des Objekts.

Der Erzeuger soll mit seiner Aktion warten, wenn der Puffer noch nicht geleert ist, bis er leer ist. Verbraucher entsprechend umgekehrt.

## ■ Geeignete Methoden:

- `wait()`: Thread geht in Zustand "wartend" über. Er *gibt* die *Objekt-Sperre* (Monitor) für andere Threads *ab*. Kann `InterruptedException` werfen.
- `notify()`: stößt einen anderen Thread wieder an, der ein `wait()` ausführt.

Die Methoden `wait()` und `notify()` sind in der Klasse `Object` definiert, also für alle Objekte vorhanden.

# Synchronisation mit wait und notify (2)

```
class MyBuffer {  
    // buffer Variables  
    private int value;  
    private boolean empty = true;  
  
    // the producer routine  
    public synchronized void put(int v) {  
        if (!this.empty) {  
            try {  
                this.wait();  
            } catch (InterruptedException e) {  
            }  
        }  
        // buffer is empty, fill it!  
        this.value = v;  
        this.empty = false;  
        // notify waiting consumer  
        this.notify();  
        System.out.println("Put: " + v);  
    }  
}
```

// ... weiter auf der nächsten Folie

Producer



# Synchronisation mit `wait` und `notify` (3)

```
// the consumer routine
public synchronized int get() {
    int v;
    if (this.empty) {
        // consumer must wait
        // until buffer is full
        try {
            this.wait();
        } catch (InterruptedException e) {
        }
    }
    // buffer is full, empty it!
    v = this.value;
    this.empty = true;
    // notify waiting producer
    this.notify();
    System.out.println("Get: " + v);
    return v;
}
}
```

Consumer

# Synchronisation mit `wait` und `notify` (4)

## ■ Das Ganze mal ausprobieren:

```
public class BufferTest1 {  
  
    public static void main(String args[]) {  
        MyBuffer buf = new MyBuffer();  
        buf.put(5);  
        buf.put(6);  
        buf.get();  
    }  
}
```

? Ausgabe ?



? Warum ?

# Synchronisation mit `wait` und `notify` (5)

## ■ Lösung:

Zwei *separate Thread-Objekte* für das Befüllen und das Auslesen der Daten einrichten.

## ■ Klassen der Objekte:

- `class ProducerThread` für das Befüllen (Erzeuger-Thread)
- `class ConsumerThread` für das Auslesen (Verbraucher-Thread)

# Synchronisation mit `wait` und `notify` (6)

```
class ProducerThread extends Thread {  
    MyBuffer buf;  
  
    public ProducerThread(MyBuffer b) {  
        this.buf = b;  
    }  
    public void run() {  
        this.buf.put(5);  
        this.buf.put(6);  
    }  
}
```

```
class ConsumerThread extends Thread {  
    MyBuffer buffer;  
  
    public ConsumerThread(MyBuffer b) {  
        this.buffer = b;  
    }  
    public void run() {  
        this.buffer.get();  
    }  
}
```

# Synchronisation mit `wait` und `notify` (7)

- Das Ganze mal ausprobieren:

```
public class BufferTest2 {  
  
    public static void main(String args[]) {  
        MyBuffer buf = new MyBuffer();  
  
        ProducerThread thread1 = new ProducerThread(buf);  
        thread1.start();  
  
        ConsumerThread thread2 = new ConsumerThread(buf);  
        thread2.start();  
    }  
}
```

? Ausgabe ?

# Erweiterte Threadprogrammierung

- Das Package `java.util.concurrent` mit den Unterpackages `java.util.concurrent.atomic` und `java.util.concurrent.locks` liefert seit Java 1.5 komfortable Klassen zur Erzeugung und Verwaltung von Threads.
  - **Concurrent Collections:** Spezielle Implementierungen der Interfaces des Collection Frameworks `Map`, `Set`, `List`, `Queue`, `Deque` (Double Ended Queue).
  - **Executor Framework:** Ein Framework zur Ausführung asynchroner Tasks durch Threadpools.
  - **Synchronizers:** Diverse Hilfsklassen zur Koordination mehrerer Threads, zum Beispiel `Semaphore` oder `CyclicBarrier`.
  - **Locks und Conditions:** Flexiblere Objektrepräsentation des `synchronized` Schlüsselworts sowie der `java.lang.Object` Monitor Methoden `wait()`, `notify()` und `notifyAll()`.
  - **Atomic Variables:** Erlauben die atomare Manipulation einzelner Variablen (CAS - Compare And Set) zur Implementierung von Algorithmen ohne Sperren (lock-free algorithms).

# Beispiel: „Atomare Variablen“ (1)

- Beispiel-Klasse „Counter“ – noch nicht atomar (ununterbrechbar)

```
class Counter {  
    private int c = 0;  
  
    public void increment() {  
        this.c++;  
    }  
  
    public void decrement() {  
        this.c--;  
    }  
  
    public int value() {  
        return this.c;  
    }  
}
```

- Bei Zugriff auf den Zähler über parallele Threads kann es zu Problemen kommen.

Beispiel aus dem Java™ Tutorial „Concurrency“

# Beispiel: „Atomare Variablen“ (2)

## ■ Herkömmliche atomare Operationen/Synchronisierung

```
class SynchronizedCounter {  
    private int c = 0;  
  
    public synchronized void increment() {  
        this.c++;  
    }  
  
    public synchronized void decrement() {  
        this.c--;  
    }  
  
    public synchronized int value() {  
        return this.c;  
    }  
} // SynchronizedCounter-Instanz ("this")  
  // ist Monitor (für alle Beispiel-Methoden)
```

Beispiel aus dem Java™ Tutorial „Concurrency“



# Beispiel: „Atomare Variablen“ (3)

 Ab Java 1.5

- Das Ganze implementiert mit `AtomicInteger`

```
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);

    public void increment() {
        this.c.getAndIncrement(); // wie c++
    }

    public void decrement() {
        this.c.getAndDecrement(); // wie c--
    }

    public int value() {
        return this.c.get(); // liefert int-Wert
    }
} // AtomicCounter-Instanz dient nicht als Monitor, sondern c!
```

Beispiel aus dem Java™ Tutorial „Concurrency“

# Beispiel: Executor Framework (1)

- Das **Executor-Framework** unterstützt die asynchrone Ausführung von Tasks mit Hilfe eines **Thread-Pools**.
- Ein Thread-Pool ist eine Sammlung von (wiederverwendbaren) Threads, die mehrere Tasks (Runnable-Objekte) nacheinander ausführen können.
- Statische Methoden der Klasse

`java.util.concurrent`.**Executors**:

- `ExecutorService newFixedThreadPool(int nThreads)`  
Erzeugt und liefert einen Thread-Pool mit *maximal* `nThreads` *Threads*.
- `ExecutorService newCachedThreadPool()`  
Erzeugt und liefert einen Thread-Pool *unbegrenzter Größe*, der nach Bedarf wachsen und schrumpfen kann.
- `ExecutorService newSingleThreadExecutor()`  
Erzeugt einen einzelnen Arbeitsthread, der *übergebene Tasks sequentiell* abarbeitet.
- `ScheduledExecutorService newScheduledThreadPool(int corePoolSize)` Erzeugt einen Thread-Pool fester Größe, der **verzögerte** und **periodische Ausführungen** (Wiederholungen) unterstützt.

# Beispiel: Executor Framework (2)

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

// ...

final int NTHREADS = 10;

// ...

ExecutorService exec = Executors.newFixedThreadPool(NTHREADS);

// ...

for (int i = 0; i < 100; i++) { // 100 neue Threads
    Runnable worker = new MyRunnable(); // Thread erzeugen
    exec.execute(worker); // und zum Arbeiten anmelden
}

// ...

// Hindere den Executor am Annehmen neuer Threads
// und beende alle Threads in der Queue
exec.shutdown();

// ...

// Warte bis alle Threads fertig sind..
while (!exec.isTerminated()) {
}
```

# Literatur

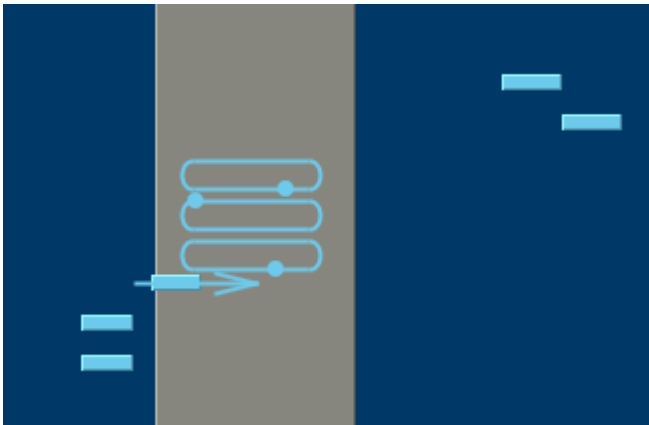
## ■ Literatur:

<http://download.oracle.com/javase/tutorial/essential/concurrency/>

<http://www.oio.de/public/java/concurrency/concurrency-utils.htm>

## ■ Animationen für erweiterte Threadprogrammierung:

<http://sourceforge.net/projects/javaconcurrenta/files/concurrentanimated.jar/download>



Screenshot aus „Java Concurrent Animated“