Informatik II

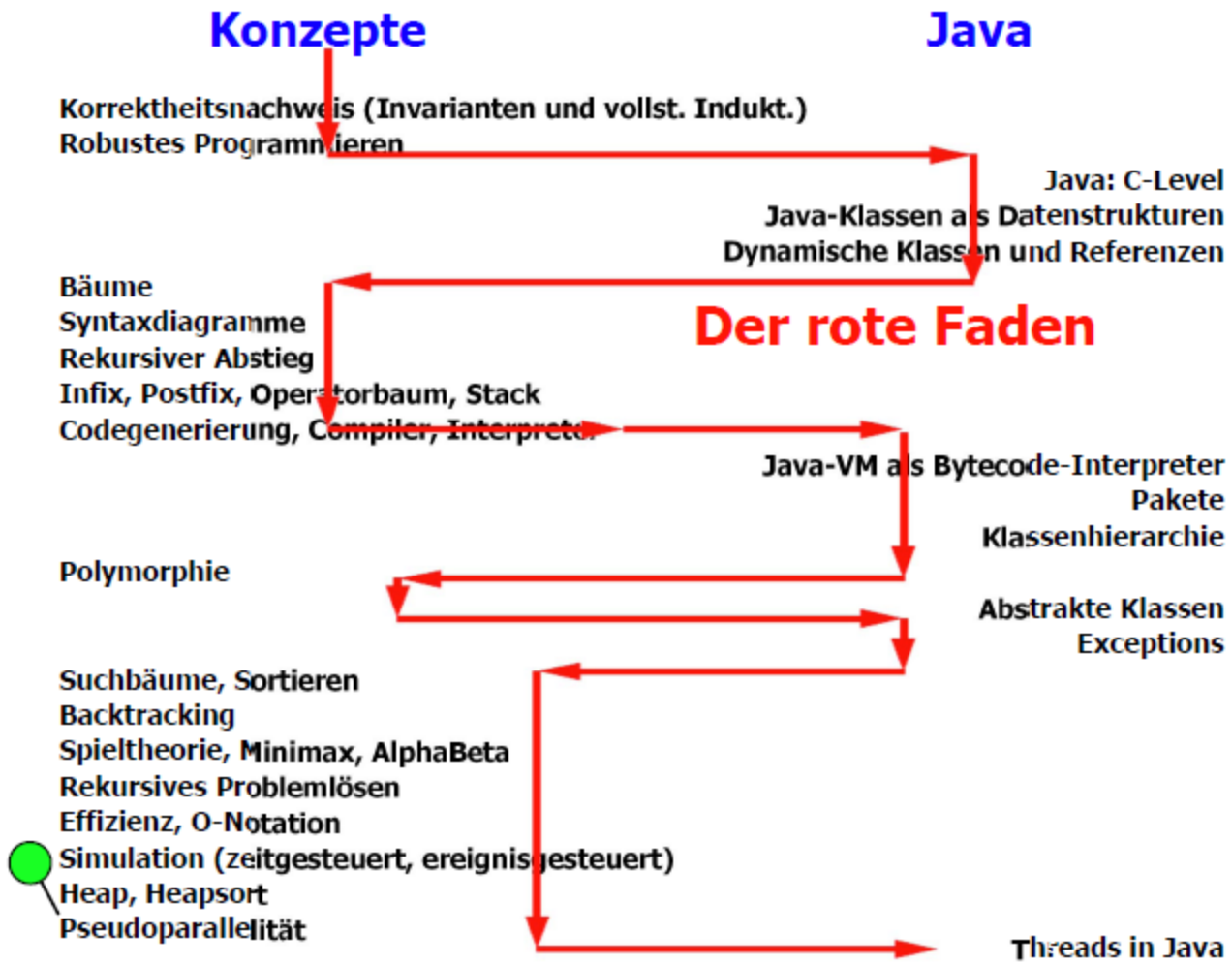
Allerletzte Übungsstunde

simon.mayer@inf.ethz.ch
Distributed Systems Group, ETH Zürich

Serie 11

Ablauf

- Besprechung der Vorlesung
- Uebungsbezogene Themen:
  Threads und Prozesse
- Zeit zum Programmieren...
  und fuer noch mehr Fragen


Konzepte
Java
Korrektheitsnachweis (Invarianten und vollst. Indukt.)
Robustes Programmieren
Java: C-Level
Java-Klassen als Datenstrukturen
Dynamische Klassen und Referenzen
Bäume
Syntaxdiagramme
Rekursiver Abstieg
Infix, Postfix, Operatorbaum, Stack
Codegenerierung, Compiler, Interpreter
Der rote Faden
Java-VM als Bytecode-Interpreter
Pakete
Klassenhierarchie
Polymorphie
Abstrakte Klassen
Exceptions
Suchbäume, Sortieren
Backtracking
Spieltheorie, Minimax, AlphaBeta
Rekursives Problemlösen
Effizienz, O-Notation
Simulation (zeitgesteuert, ereignisgesteuert)
Heap, Heapsort
Pseudoparallelität
Threads in Java

Prozesse

- Prozess = Programm in Ausführung
  - „Instanz" eines Programms
  - Programm selbst ist passives Ding auf Papier oder in einer Datei
- Es kann gleichzeitig mehrere Prozesse als verschiedene Instanzen des gleichen Programms geben
  - Z.B. mehrere aktive Web-Browser
- Kontext eines Prozesses (zu einem Zeitpunkt) umfasst u.a.:
  - Aktuelle Stelle („Befehlszähler")
  - Inhalt der CPU-Register
  - Werte aller Variablen
  - Inhalt des Laufzeitstacks (dynamische Aufrufsequenz)
  - Zustand zugeordneter Betriebsmittel (z.B. geöffnete Dateien)

Kontext wird oft einfach als Zustand bezeichnet

Prozesse und Betriebsmittel

- Ein Prozess benötigt Betriebsmittel („Ressourcen")
  - CPU-Zeit, Hauptspeicher (RAM), Dateien...
- Prozesse werden durch das Betriebssystem verwaltet
  - Gründen (z.B. im Auftrag anderer Prozesse)
  - Terminieren (dann Freigabe aller belegten Betriebsmittel)
  - Kontrolle des Ressourcenverbrauchs (Schranken, Monopolisierung)
  - Scheduling („suspend", „resume" etc. zum Multiplexen der CPU)
  - Mechanismen zur Synchronisation
  - Vermittlung von Kommunikation zwischen den Prozessen (z.B. Signale, Ereignisse oder Nachrichten)

Prozesskontrollblock (2)

„laufend" → „lauffähig" / „blockiert"

- Wird der laufende Prozess unterbrochen, muss der aktuelle Kontext des Prozesses gesichert werden; hierzu dienen diverse Felder in einem Prozesskontrollblock
- Der Prozesskontrollblock enthält u.a.:
  - Eindeutige Prozessnummer
  - Scheduling-Zustand (blockiert, lauffähig...)
  - Programmzähler
  - Inhalt der Register

  } (wenn nicht laufend)

  - Priorität
  - Zeiger auf verwendete Speicherbereiche
  - Zeiger auf eigene Kind-Prozesse
  - Zeiger auf Betriebsmittel-Listen (z.B. geöffnete Dateien)
  - Rechte und Schutz-Information (z.B. zugehöriger „User")
  - Abrechnungsinformation (Zeitlimit, verbrauchte Zeit, Startzeit,...)

Wenn der Prozess wieder laufend wird, lädt das Betriebssystem dies in die CPU zurück

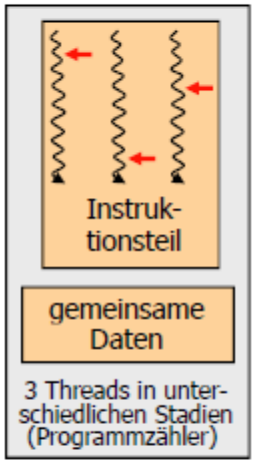
Kontextwechsel

- Der zu sichernde Prozesszustand ist recht umfangreich
- Hinzu kommen diverse Verwaltungsaktionen, die das Betriebssystem bei einem Kontextwechsel durchführt
  - Z.B. den Speicherbereich des neuen Prozesses vor Zugriffen anderer abschotten
  - Zugriffsberechtigung auf Ressourcen prüfen
  - ...
- Kontextwechsel ist daher relativ teuer
  - Kostet typischerweise einige zehntausend Instruktionen →
  - Man kann sich nicht viele „Prozesswechsel" pro Sekunde erlauben

Leichtgewichtsprozesse (Threads)

Dadurch anfällig gegenüber Programmierfehlern!

- Threads („of control") sind parallele Kontrollflüsse, die nicht gegeneinander abgeschottet sind
  - Laufen innerhalb eines gemeinsamen Adressraumes
  - Teilen sich gemeinsame Ressourcen
- Kontextwechsel zwischen den Threads ist viel effizienter als Kontextwechsel zwischen Prozessen
  - Kein Adressraumwechsel
  - Kein automatisches Scheduling
  - Kein Retten / Restaurieren des Kontextes (Ausnahme: Register, Programmzähler etc. analog zu Unterprogrammaufruf)
- Pro Zeiteinheit viel mehr Threadwechsel als Prozesswechsel möglich
  - Wichtig für Server (z.B. Datenbanken oder Suchmaschinen), die pro Sekunde tausende von Anfragen quasi-gleichzeitig bearbeiten müssen



Multithreading

- Quasi-gleichzeitig mehrere Vorgänge innerhalb einer einzigen Anwendung erledigen

- Oft angewendet bei interaktiven Programmen
  - Z.B. „endlose" Animation, wobei Interaktion jederzeit möglich sein soll
  - Lösung: Zwei Threads (Animation und Verwalter der input buttons)

Andere typische Anwendung: Window-Manager, der mehrere Fenster auf dem Display verwaltet

- Ohne Multithreading müsste der einzige Kontrollfluss selbst schnell genug zwischen den Aufgaben hin- und herschalten
  - Dies dann entweder aktiv durch regelmässiges Nachfragen („liegt jetzt eine Anforderung vor?")
  - Oder durch Interrupts („bei Mausklick kurz mal etwas anderes machen")
  - Dies ist i.Allg. ineffizient, komplex und fehleranfällig

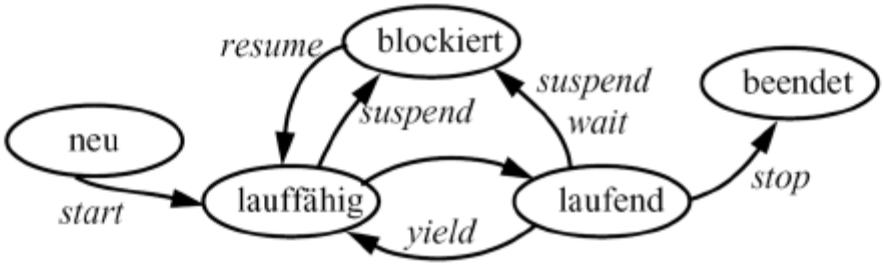
Klasse java.lang.Thread

Hier nur ein Auszug; zu weiteren Aspekten vgl. die Dokumentation (online bzw. in Büchern)

- Konstruktor: `public Thread()`
- Methoden:

```
- void start()
- void suspend()
- void stop()
```

```
- void resume()
- void wait()
- void yield()
```



Wegen Fehleranfälligkeit wird empfohlen, suspend, stop und resume nicht zu verwenden – daran halten wir uns aber nicht!

Vgl. auch: http://tinyurl.com/3dhrofr

```
- void sleep(long millis) // blockiert einige ms
- void join() // Synchronisation zweier Threads
- int getPriority()
- void setPriority(int prio)
- void setDaemon(boolean on)
```

„Hintergrundprozess": terminiert *nicht* mit dem Erzeuger

Ein Thread-Beispiel („Hin-Her")

Nach einer Idee von Ralf Kühnel („Die Java-Fibel", Addison-Wesley)

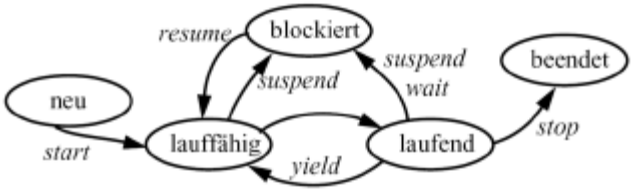
- ********** **********
- Ein Thread „Hin" schreibt Sterne; ein anderer Thread „Her" löscht Sterne; beide arbeiten (quasi)parallel
  - Übung: Ausprobieren, wer gewinnt!

```
public class HinHer {
  public static void main (String args[]) {
  System.out.print("**********");
    System.out.flush();
     new Hin().start();
     new Her().start();
   }
 }
```

Kommt es dazu überhaupt noch? Oder behält jetzt „Hin" die ganze Zeit die Kontrolle?

Thread-Steuerung

- Ein Thread läuft (d.h. ist laufend oder lauffähig) so lange, bis
  - seine run-Methode zu Ende ist
  - er mit stop() abgebrochen wird (von aussen oder durch sich selbst)



- Ein Thread kann *sich selbst*
  - Die CPU entziehen: yield()
    (Übergang in den Zustand „lauffähig"; wird automatisch wieder „laufend", wenn keine wichtigeren Threads mehr laufen möchten)
  - Schlafen legen: sleep(...) (wie suspend, aber automatisches resume nach gegebener Zeit)
  - Anhalten: suspend() bzw. wait()
  - Beenden: stop()
  - In der Priorität verändern: setPriority(...)

Prioritäten: normal 5, minimal 1, maximal 10 (anfangs: Priorität des erzeugenden Threads)

Threads: Schwierigkeiten

- Ein Thread mit Endlosschleife kann u.U. das ganze System blockieren (so dass andere Threads „verhungern“)
  - Eventuell rücksichtsvoll mit „yield“ dem Scheduler helfen
  - Ist insbesondere bei Systemen ohne Zeitscheiben wichtig

- Bei Prozessoren mit mehreren CPUs („multicore“) könnten entsprechend viele Threads „echt gleichzeitig“ ausgeführt werden
  - Schon deswegen kein Verlass, dass Synchronisation bzw. wechselseitiger Ausschluss, realisiert mittels Prioritäten, funktioniert!
  - Böses Erwachen, wenn ein solches Programm dann irgendwann einmal auf einem Multicore-Prozessor ausgeführt wird...

Ablauf

- Besprechung der Vorlesung
- Uebungsbezogene Themen:
  Threads und Prozesse
- Zeit zum Programmieren...
  und fuer noch mehr Fragen

Übung 12

1. Threaded Mergesort
   Macht das! Ist super!

Ablauf

- Besprechung der Vorlesung
- Uebungsbezogene Themen:
  Threads und Prozesse
- Zeit zum Programmieren...
  und fuer noch mehr Fragen

Informatik II

Übungsstunde 12

simon.mayer@inf.ethz.ch
Distributed Systems Group, ETH Zürich