Version vom
2. März 2020

16.

Parallele Prozesse und Threads

The Java Tutorials (https://docs.oracle.com/javase/tutorial/),
Lesson “Concurrency” (“Doing Two or More Tasks At Once”):
https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

Lernziele Kapitel 16 Parallele Prozesse und Threads

- Prinzip von Multitasking und das Prozess-Zustandsmodell verstehen
- Kontextwechsel und Prozesskontrollblock (PCB) verstehen
- Class „Thread" mit zugehörigen Steuermethoden anwenden können
- Problematik von race conditions, Atomarität, lost update sowie Deadlocks verstehen und damit umgehen können
- Anforderungen an den wechselseitigen Ausschluss kennen

Thema / Inhalt

Multitasking und generell „**Parallelität**" sind nichts neues in der Informatik. Schon Mitte der 1960er-Jahre hatten Grossrechner genügend Ressourcen, um mehrere Anwender quasi-gleichzeitig, im Timesharing-Betrieb, zu bedienen. Dies war ökonomisch sinnvoll, da die Rechner sehr teuer waren und die Anwendungen der Nutzer viele Rechenpausen enthielten (E/A-Zugriff, Warten auf Benutzereingabe etc.), die produktiv für die Anwendungen anderer Nutzer verwendet werden konnten. Einige frühe Betriebssysteme, insbesondere Unix, waren auch schlank genug, um eine grössere Zahl pseudoparalleler Prozesse zu unterstützen, so dass Anwendungen aus kooperierenden Prozessen realisiert werden konnten und ein Nutzer mehrere Softwareanwendungen „virtuell gleichzeitig" ausführen konnte.

Thema / Inhalt (2)

Von Anfang an, als Multitasking / Multiprogramming / Multiprocessing aufkam, musste dafür gesorgt werden, dass die miteinander um Rechenzeit der CPU konkurrierenden Prozesse (bzw. „Tasks“ oder „Threads [of Control]“) durch das Betriebssystem kontrolliert und koordiniert werden können und dass diese teilweise sich selbst und die ihnen zugeordneten Prozesse verwalten können. Kommandos wie „start“, „stop“, „yield“ etc. waren daher zusammen mit Zeitscheiben und einer „ready queue“ rechenwilliger und auf CPU-Zuteilung wartender Prozesse schon früh die kanonischen Ausprägungen eines einfachen Multitasking-Modells.

Wenn mehrere nebenläufige Prozesse um gemeinsame Ressourcen wetteifern und auf gemeinsame Speicherbereiche bzw. Variablen zugreifen können, dann kann es leicht zu einem **nichtdeterministischen Verhalten** und ungewollten Effekten kommen. Eine Lösung für einige dieser Probleme (wie das „**Lost-Update-Problem**“) besteht darin, dass eine Folge von Anweisungen „**atomar**“ gemacht wird, so dass diese Folge während ihrer Ausführung nicht unterbrochen wird. Auch die Realisierung des **wechselseitigen Ausschlusses** als Dienstleistung hilft, manche unerwünschten Phänomene zu vermeiden, ähnlich wie im Strassenverkehr die Regel „rechts vor links“ oder Verkehrsampeln helfen, Zusammenstösse zu vermeiden. Damit steht man aber vor neuen Problemen: Wie realisiert man Atomarität oder den wechselseitigen Ausschluss und was tut man, wenn sich zwei oder mehr Prozesse auf unglückliche Weise in einem Deadlock verkeilt haben?

Das Beherrschen der Phänomene und überraschenden Nebenwirkungen der Parallelität ist schwierig. Bis vor einigen Jahren wurden fast nur Spezialisten für Systemsoftware damit konfrontiert. Nun hat aber die Hardwareentwicklung dafür gesorgt, dass es immer mehr Rechenkerne („cores“) in einem Prozessor gibt und viele Anwendungen diese Hardwareparallelität aus Effizienzgründen ausnutzen wollen, da die Rechengeschwindigkeit einzelner Prozessoren von

Thema / Inhalt (3)

einer Generation zur nächsten kaum mehr gesteigert werden kann. In Konsequenz werden auch Entwickler von Anwendungssoftware damit konfrontiert, Parallelität (z.B. in Form von Multithreading) zu verwenden – was wiederum bedingt, dass man die Phänomene und Probleme der Parallelität verstehen und beherrschen sollte.

In faktischer Hinsicht beginnt das Kapitel mit der Klärung grundlegender Begriffe und Prinzipien wie Multitasking, **Quasi-Parallelität** versus **„echter“ Parallelität** und **Threads** versus **Prozesse**. Anschliessend wird das **Prozesszustandsmodell** eingeführt und die darauf aufbauende **Java-Klasse „Thread“** mit ihren wichtigsten Methoden vorgestellt. Ein kleines Beispiel, bei dem zwei Java-Threads im Wettbewerb um sichtbare Effekte auf der Konsole miteinander stehen, illustriert dies sowie das Gründen und Kontrollieren von Threads.

Danach folgt die Vorstellung und Diskussion der wichtigsten Phänomene und Probleme des parallelen Programmierens: **Race conditions**, **lost updates**, **Atomarität**, **kritische Abschnitte** (und deren Java-Realisierung mittels „synchronized“) sowie **Deadlocks**.

Beim thematischen Kontext und der Historie gehen wir kurz auf den **Fairness**-Begriff ein, ferner schildern wir, wie fast die **Mars-Mission** von 1997 („Pathfinder“) an einem Thread-Synchronisationsproblem (priority inversion) gescheitert wäre, erwähnen die Software-Probleme, die andere Mars-Missionen plagten („Spirit“ 2004; „Curiosity“ 2011; „Schiaparelli“ 2016) und schildern dann, wie 2003 eine race condition für einen grossen **Blackout** im Nordosten der USA und Kanada führte. Die historisch erste Lösung für das Problem kritischer Abschnitte gibt Gelegenheit, über einige biographische Begebenheiten aus dem Leben des Informatik-Pioniers **E.W. Dijkstra** zu berichten – schliesslich, so Sebastian Stiller, findet sich heute „Dijkstra in jedem Navi“.

Prozesse

Abstrakter: „Vorgang einer algorithmisch ablaufenden Informationsverarbeitung"

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

Kontext wird oft einfach als Zustand bezeichnet

Idee: Kontext einfrieren, wegspeichern, später wiedererwecken

Prozessverwaltung und Betriebsmittel

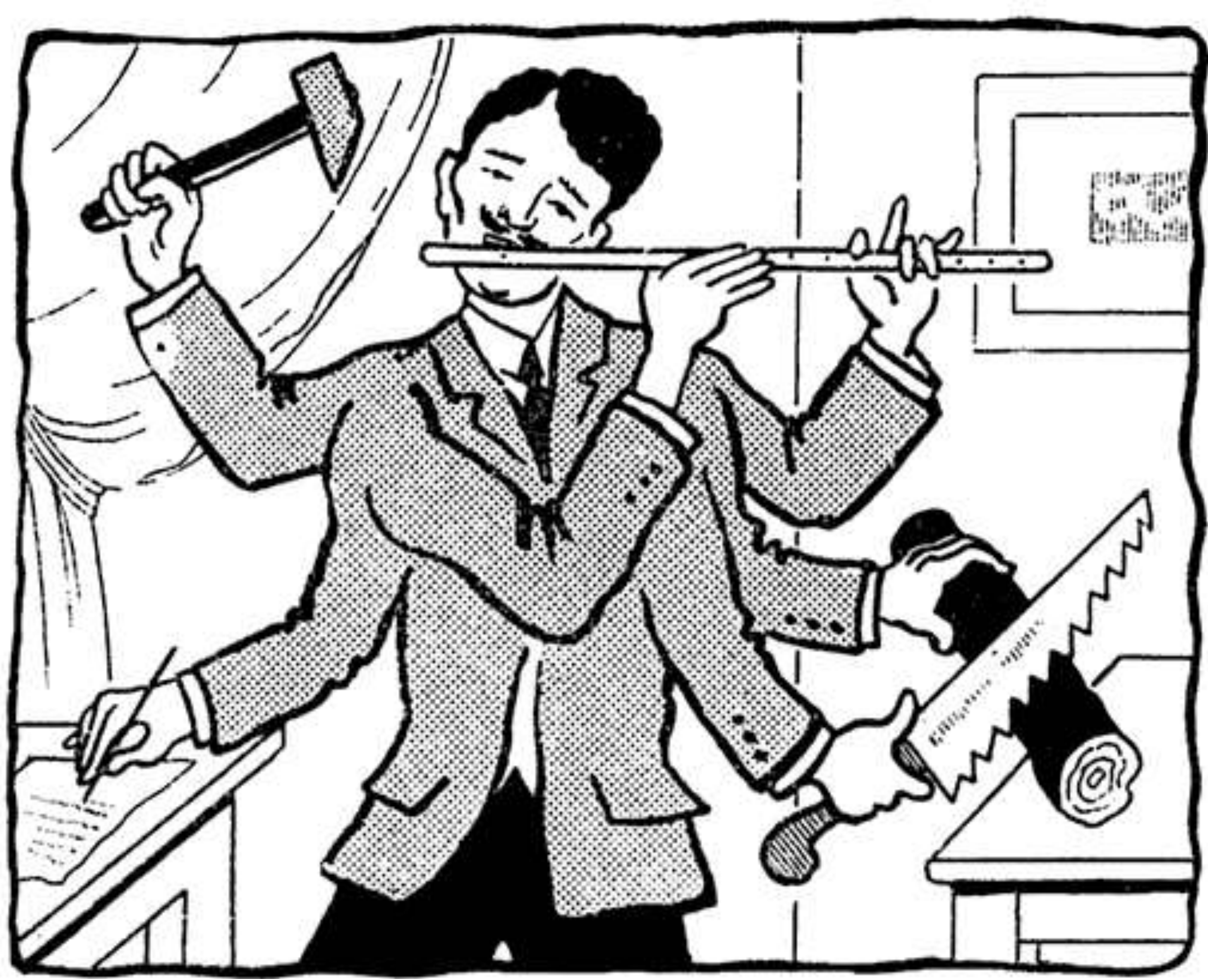
- Ein Prozess benötigt Betriebsmittel („Ressourcen“)
  - CPU-Zeit, Speicher (RAM), Dateien...
  - Mehrere Prozesse konkurrieren um diese Betriebsmittel

- Nicht nur die Ressourcen, sondern auch die Prozesse selbst 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; der Scheduler ist eine eigenständige Komponenten des Betriebssystems)
  - Mechanismen zur Synchronisation
  - Vermittlung von Kommunikation zwischen den Prozessen (z.B. Signale, Ereignisse oder Nachrichten)
  - Managen der Prozess-Konkurrenz

> Betriebssysteme sind wie Behörden: Eigentlich machen sie selbst nichts wirklich Produktives, aber ohne sie läuft nichts. -- Till Tantau

Multitasking

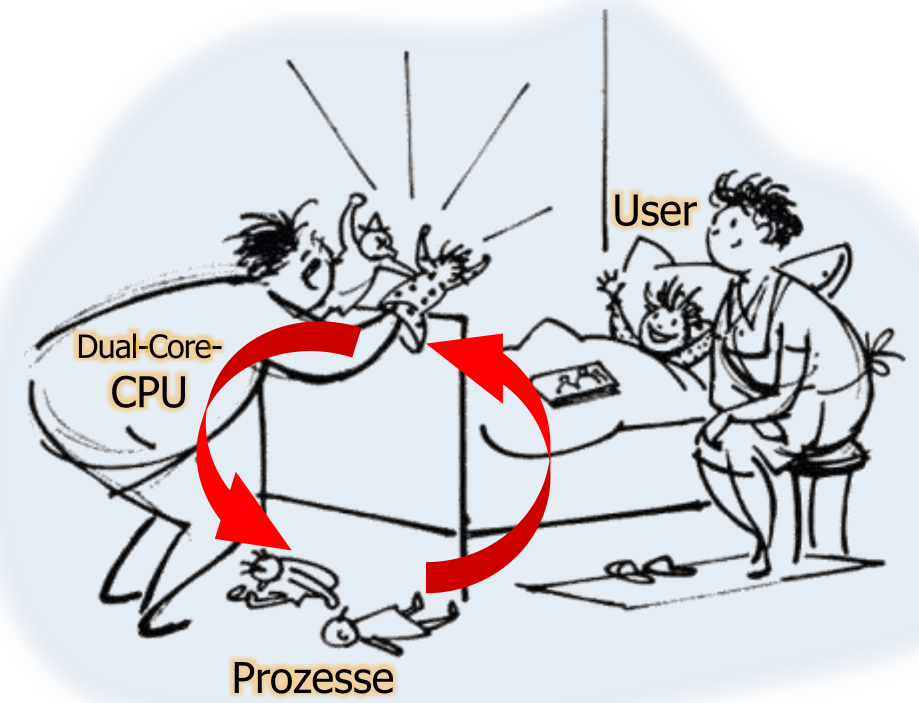
- Mehrere Aufgaben (tasks) quasi-gleichzeitig ausführen

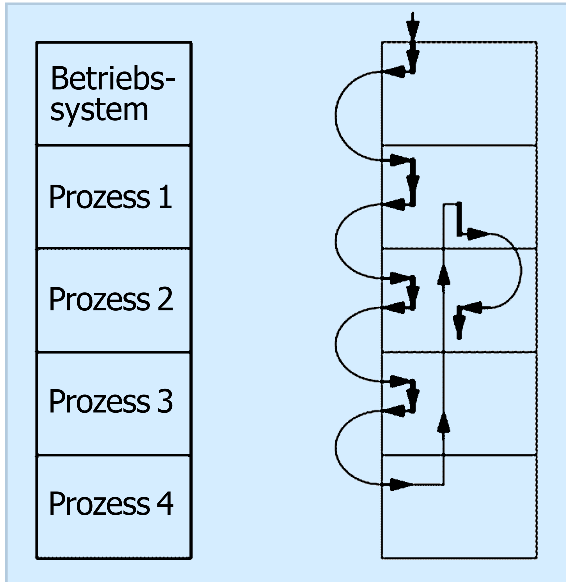


Multitasking

Hier: Prozesse, daher auch „Multiprocessing"

- Mehrere Aufgaben (tasks) quasi-gleichzeitig ausführen
  - Prozess unterbrechen; später fortsetzen
  - Multiplexen der CPU: time-sharing durch Zeitscheiben (Interrupt durch „timer" erzwingt Freigabe der CPU)





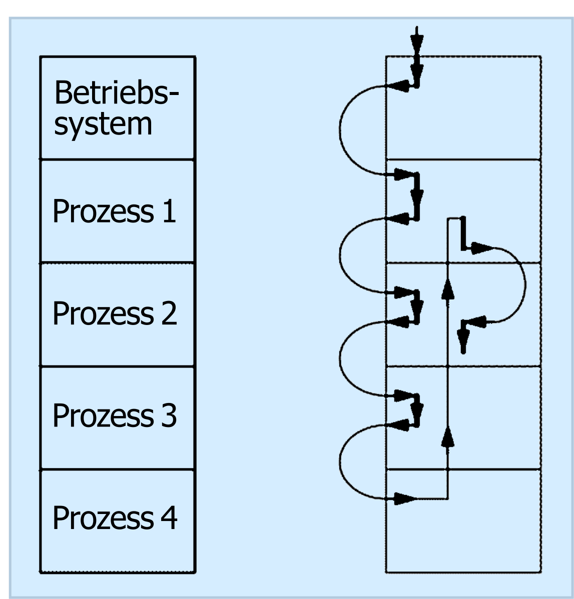
Prozesse in so kurzen Abständen immer abwechselnd aktivieren, dass der Eindruck der Gleichzeitigkeit entsteht

Vergleiche mit dem Modell der ereignisorientierten Simulation

Multitasking

Hier: Prozesse, daher auch „Multiprocessing"

- Mehrere Aufgaben (tasks) quasi-gleichzeitig ausführen
  - Prozess unterbrechen; später fortsetzen
  - Multiplexen der CPU: time-sharing durch Zeitscheiben (Interrupt durch „timer" erzwingt Freigabe der CPU)



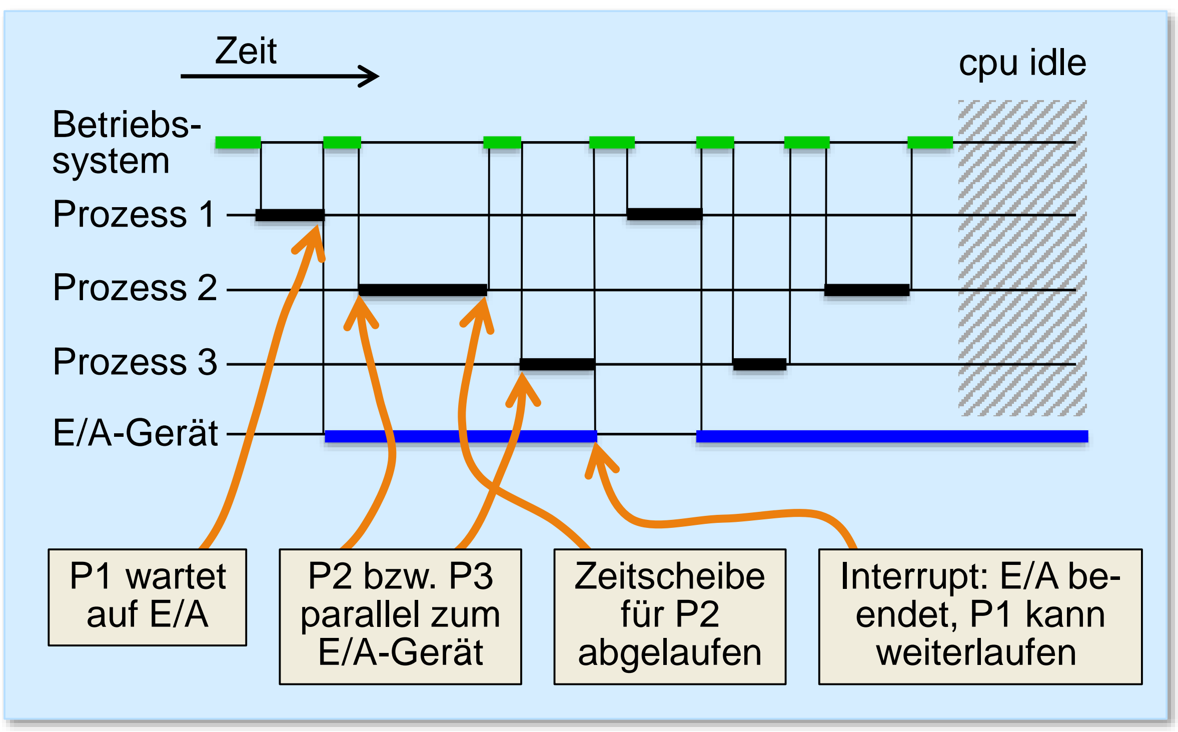
Schnelles Hin- und Herschalten geht dann gut, wenn sich mehrere Prozesse bereits im Hauptspeicher befinden (und nicht erst jedes Mal geladen werden müssen)

- Zweck:
  1) Gleichzeitiges Steuern / Überwachen mehrerer Abläufe der realen Welt
  2) Bessere CPU-Nutzung
     - Während ein Prozess auf externe Ereignisse wartet (z.B. Daten vom Netz oder einem angeschlossenen Gerät), kann ein anderer ausgeführt werden
     - Beispiel: 10 ms Wartezeit → ca. 10^{10} Instruktionen bei anderen Prozessen

Echt und quasi-gleichzeitige Abläufe
(Nebenläufigkeit, „concurrency“)

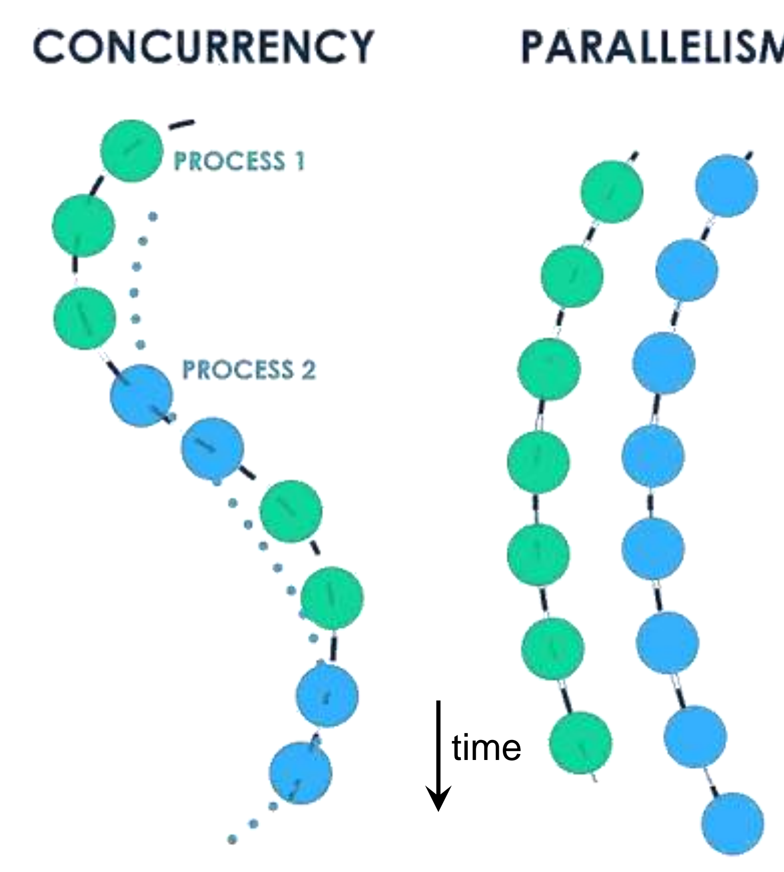
E/A-Geräte (z.B. WLAN-Controller oder externe Devices wie Speichermedien oder Drucker) können i.a. „echt“ parallel zur CPU arbeiten

→ Optimierungsziel: möglichst alle Geräte (und die CPU) ständig beschäftigen und einen möglichst hohen Über-lappungsgrad erzielen



Echt und quasi-gleichzeitige Abläufe (Nebenläufigkeit, „concurrency“)

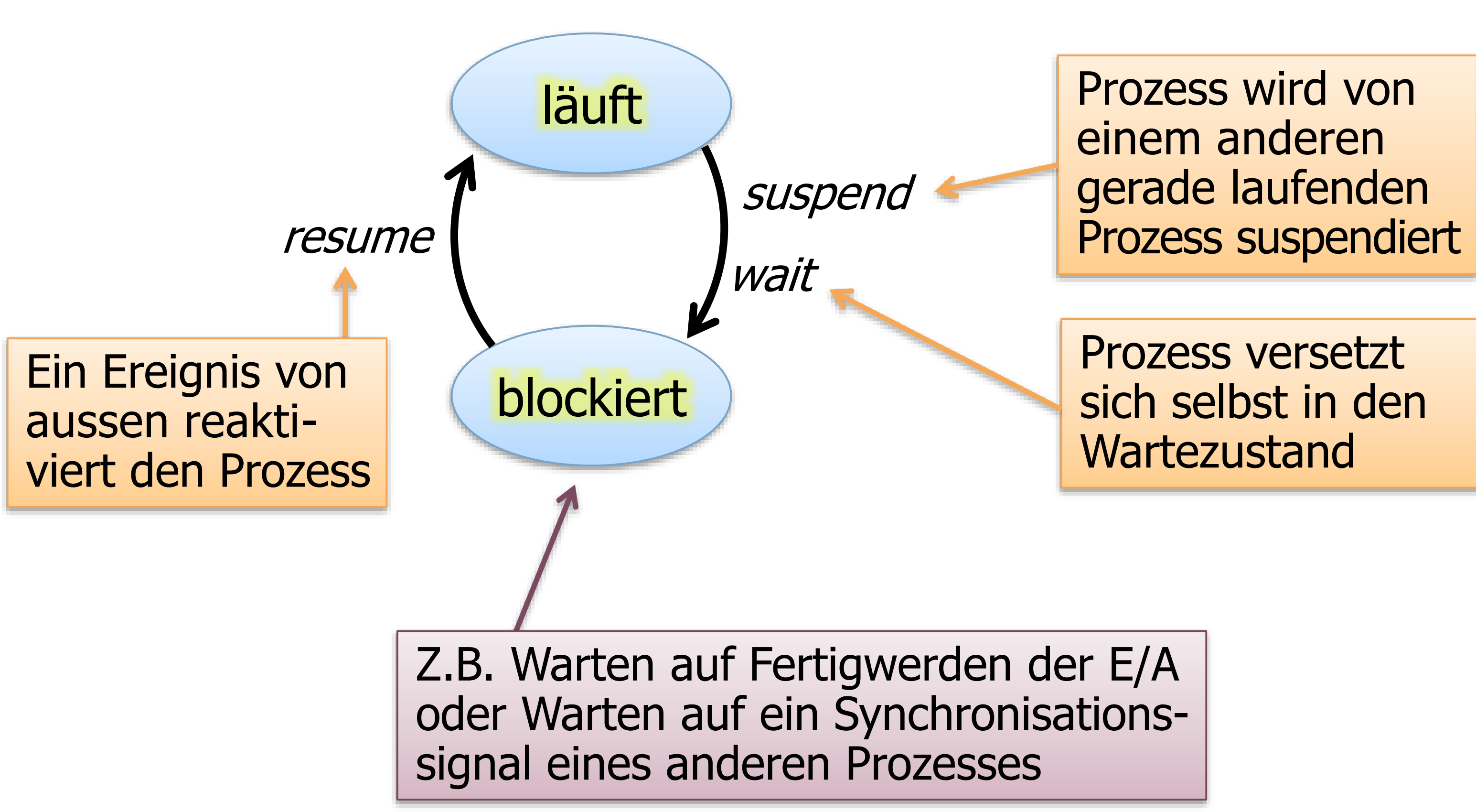
- Die Sprechweise und Modellierung ist leider nicht ganz einheitlich: Oft (aber nicht immer!) wird im Englischen die Pseudoparallelität als „concurrency“ bezeichnet, während „parallelism“ als Begriff dann für eine “echt gleichzeitige” Ausführung reserviert bleibt, deren primärer Zweck die Beschleunigung ist:
- *Concurrency: two or more processes (or threads) run together, but not at the same time; only one process executes at once.*
- *Parallelism: processes (or threads) run in parallel; meaning they execute alongside each other at the same time.*



Zitat und Bildquelle: https://medium.com/platformer-blog/node-js-concurrency-with-async-await-and-promises-b4c4ae8f4510

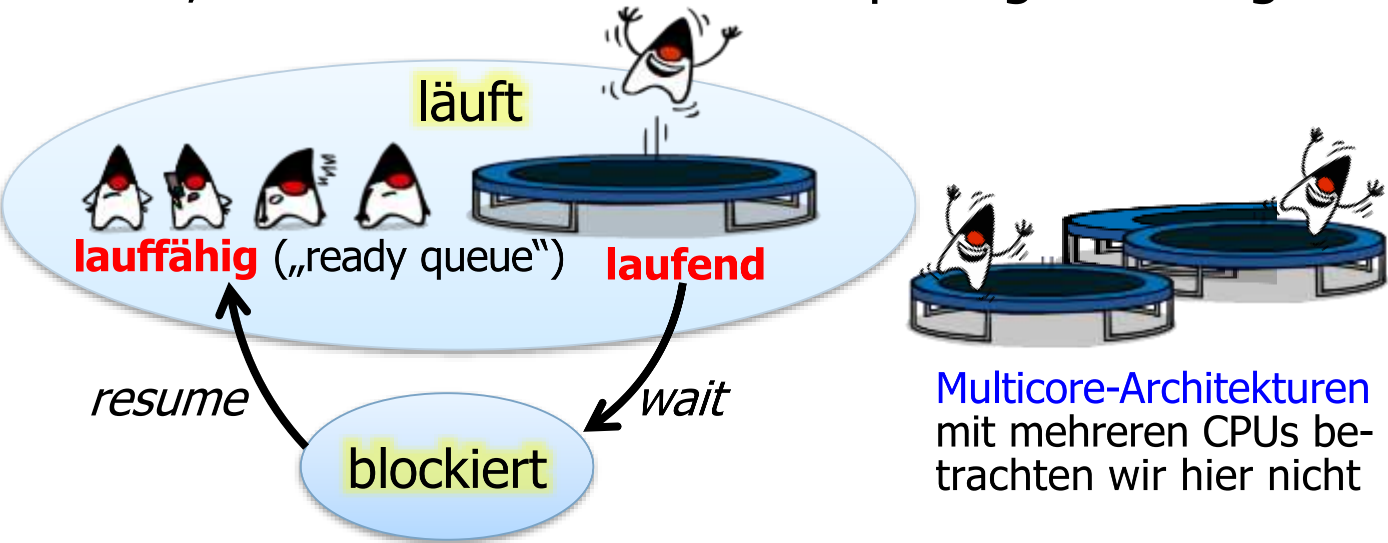
Prozesszustände

- Von aussen gesehen, kann ein Prozess in zwei Zuständen sein:



Prozesszustände: Verfeinerung

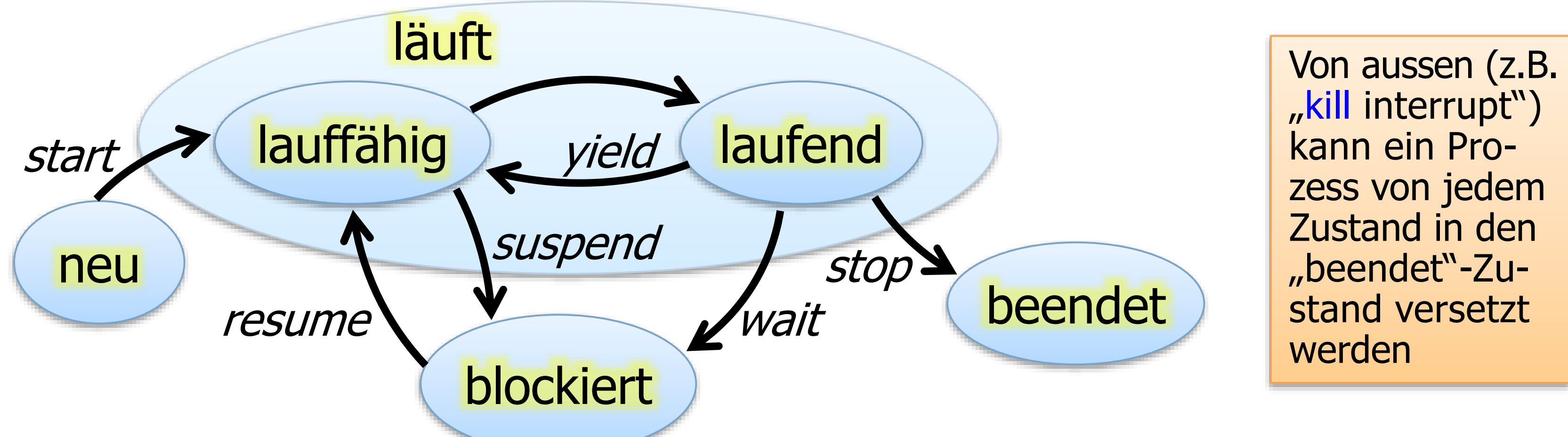
- Bild verfeinern, wenn mehrere Prozesse quasi-gleichzeitig laufen:



- Zu einem Zeitpunkt ist stets nur ein einziger Prozess tatsächlich laufend; die lauffähigen warten darauf, ein bisschen CPU-Zeit zu bekommen
  - Zustandswechsel lauffähig („ready“) / laufend („running“) wird vom Betriebssystem vorgenommen (ohne dass der Prozess selbst dies merkt: aus seiner Sicht geht es nur mal wieder sehr langsam voran)

Prozesszustände: Verfeinerung

- Bild verfeinern, wenn mehrere Prozesse quasi-gleichzeitig laufen:



Von aussen (z.B. „kill interrupt") kann ein Prozess von jedem Zustand in den „beendet"-Zustand versetzt werden

- Zu einem Zeitpunkt ist stets nur ein einziger Prozess tatsächlich laufend; die lauffähigen warten darauf, ein bisschen CPU-Zeit zu bekommen
  - Zustandswechsel lauffähig („ready") / laufend („running") wird vom Betriebssystem vorgenommen (ohne dass der Prozess selbst dies merkt: aus seiner Sicht geht es nur mal wieder sehr langsam voran)
- Unterscheide „blockiert" (fehlende Ressourcen zum Weiterlaufen) und „lauffähig" (könnte, aber darf nicht weitermachen – wartet nur auf CPU)

Prozesskontrollblock

- Prozesse werden durch das Betriebssystem verwaltet
  - Multiplexen der CPU („Scheduling“)
  - Überwachung des Ressourcenverbrauchs

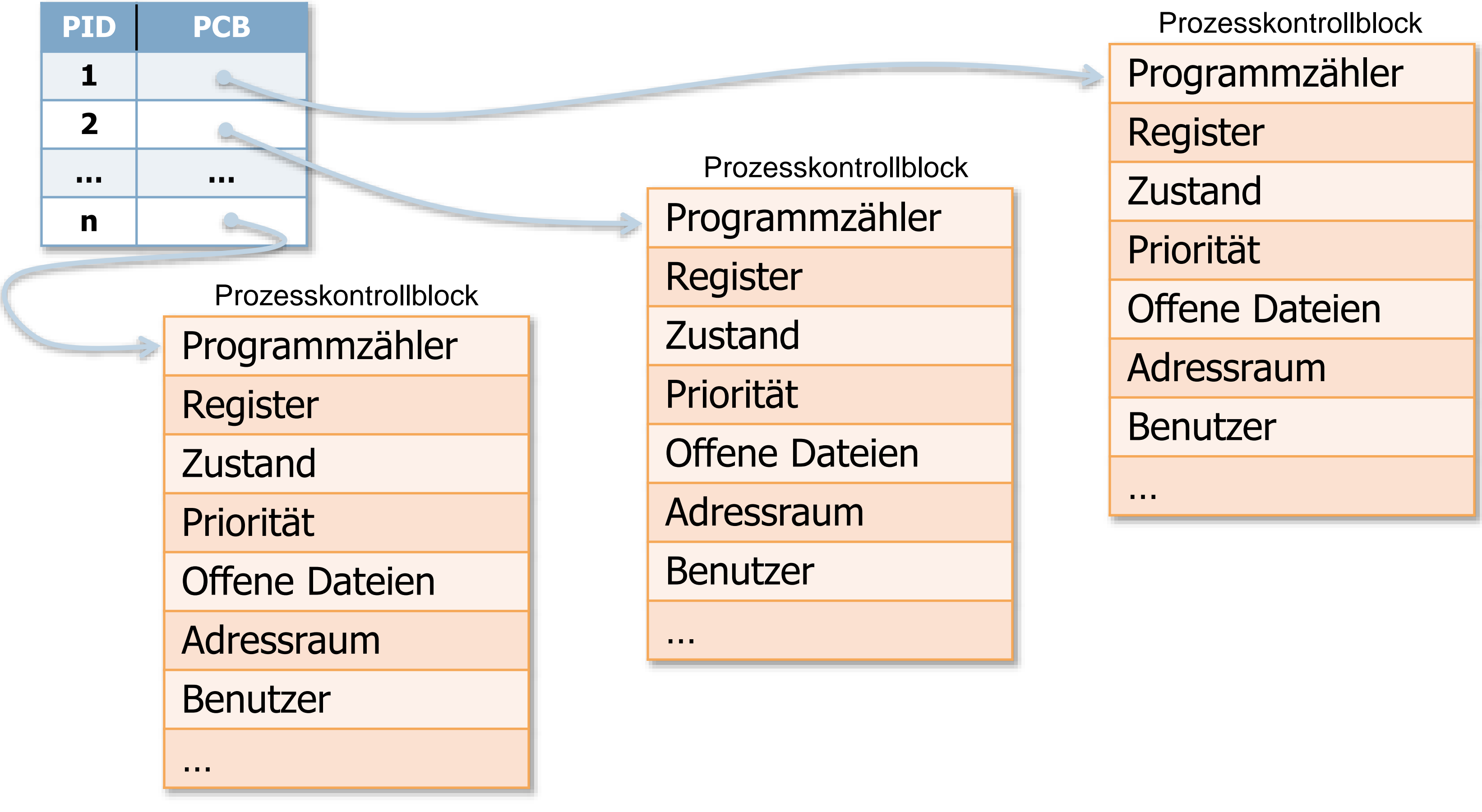
Vom Lateinischen schedula, Diminutiv von *scheda* (bzw. *scida*) für einen abgespalteten Streifen der Papyrusstaude, woraus auch über das mittellateinische *cedula* und das mittelhochdeutsche *zedele* dann das deutsche Wort *Zettel* entstand. Vom 14. bis zum 17. Jahrhundert war im Englischen das Substantiv *schedule* für ein „slip of paper“ bzw. eine kurze Notiz gebräuchlich.

- Dazu wird für jeden Prozess ein Prozesskontrollblock (PCB, „Process Control Block“) angelegt, der alle notwendige Verwaltungsinformationen zum Kontext enthält
  - Eindeutige Prozessnummer (PID)
  - Scheduling-Zustand (blockiert, lauffähig...)
  - Programmzähler
  - Inhalt der Register
  - Priorität

  (Programmzähler, Inhalt der Register, Priorität: wenn „eingefroren“, d.h. nicht laufend)
  - Maximal erlaubte Hauptspeichernutzung
  - Zeiger auf verwendete Speicherbereiche und eigene Kind-Prozesse
  - Zeiger auf Betriebsmittel-Listen (z.B. geöffnete Dateien)
  - Rechte und Schutz-Information (z.B. zugehöriger „User“)
  - Abrechnungsinformation (Zeitlimit, verbrauchte Zeit, Startzeit,...)
  - ...

Prozesstabelle

- Alle Prozesskontrollblöcke zusammen werden in einer Prozesstabelle gehalten



Kontextsicherung

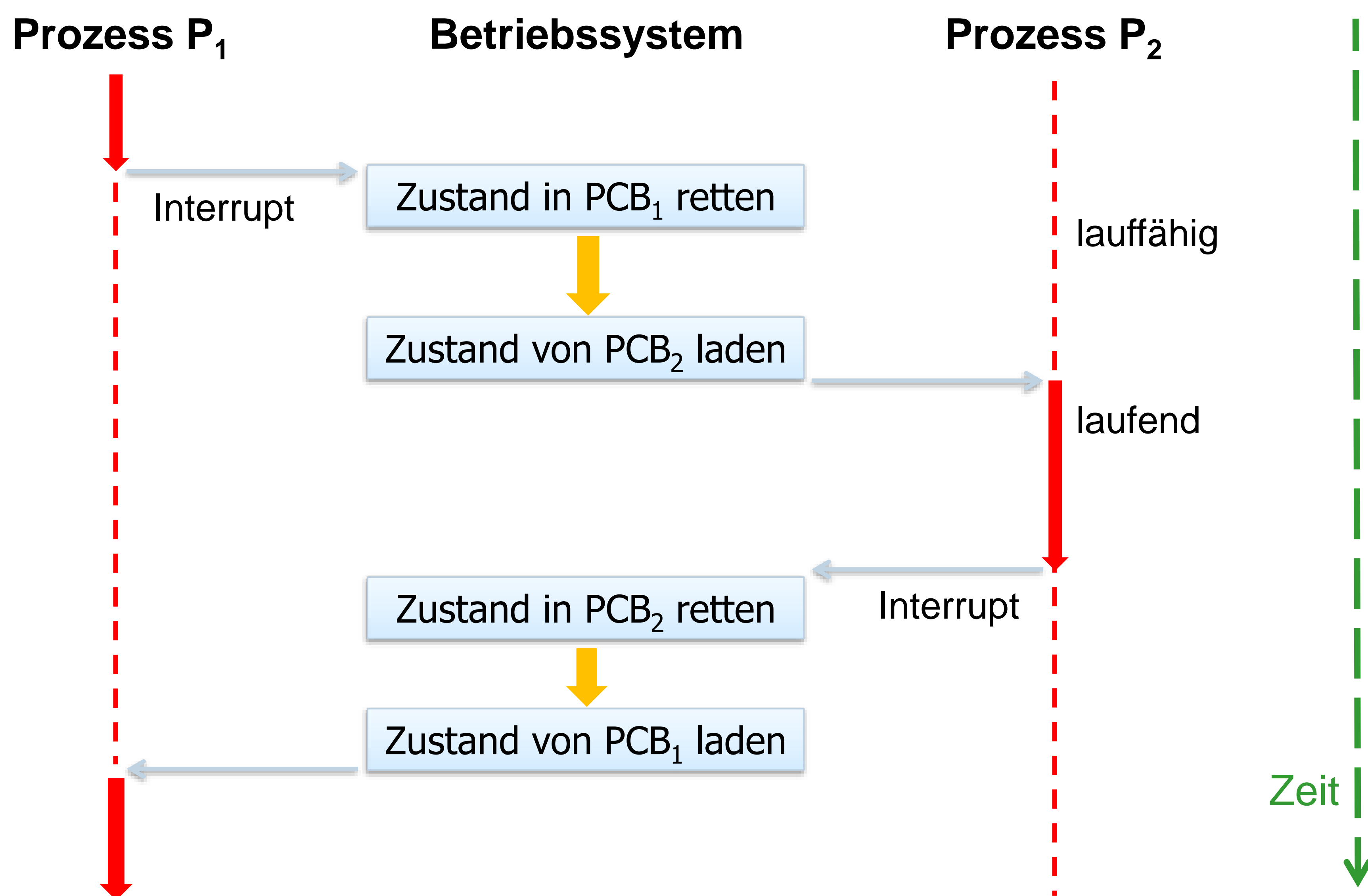
„laufend" → „lauffähig" / „blockiert"

- Wird der laufende Prozess unterbrochen, muss der aktuelle Kontext des Prozesses gesichert werden; hierzu dienen diverse Felder im Prozesskontrollblock
- Der Prozesskontrollblock enthält u.a.:
  - ...
  - Programmzähler
  - Inhalt der Register
  - ...

} (wenn nicht laufend)

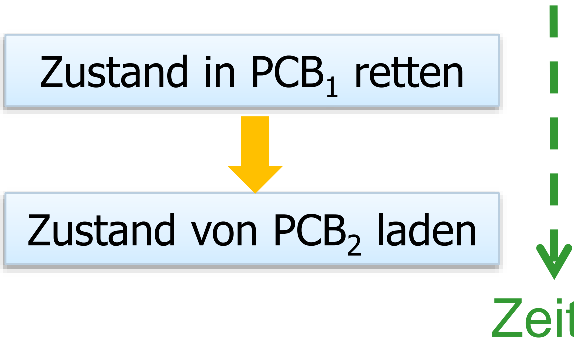
Wenn der Prozess wieder laufend wird, lädt das Betriebssystem dies in die CPU zurück

Kontextwechsel



Kosten eines Kontextwechsels

- Der zu sichernde Prozesszustand ist recht umfangreich



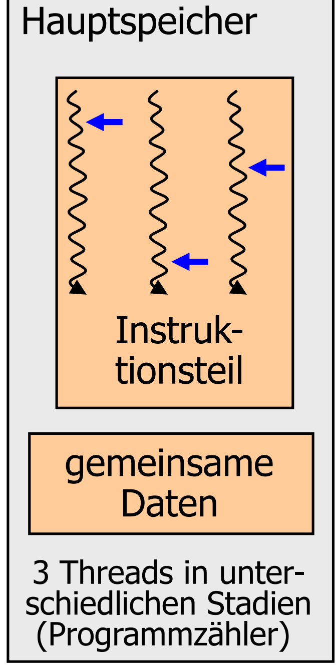
- Hinzu kommen diverse Verwaltungsaktionen, die das Betriebssystem bei einem Kontextwechsel durchführt
  - Speicherbereich des neuen Prozesses vor Zugriffen anderer abschotten
  - Zugriffsberechtigung auf Ressourcen prüfen
  - …

- Kontextwechsel ist daher relativ teuer
  - Kostet typischerweise einige zehntausend Instruktionen →
  - Man kann sich nicht allzu viele Prozesswechsel pro Sekunde erlauben

Leichtgewichtsprozesse (Threads)

Dadurch anfällig gegenüber Programmierfehlern!

- Threads („of control") sind (pseudo)parallele Aktivitätsträger, 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
  - Oft kein aufwändiges / 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 in einer einzigen Anwendung erledigen
- Oft angewendet bei interaktiven Programmen
  - Z.B. „endlose" Animation, wobei Interaktion jederzeit möglich sein soll
  - Lösung: Zwei „gleichzeitige" Threads:
    1. Verwalter der input buttons
    2. Animation

slow
fast
slow

Andere typische Anwendung: Ein 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 komplex, fehleranfällig und ineffizient

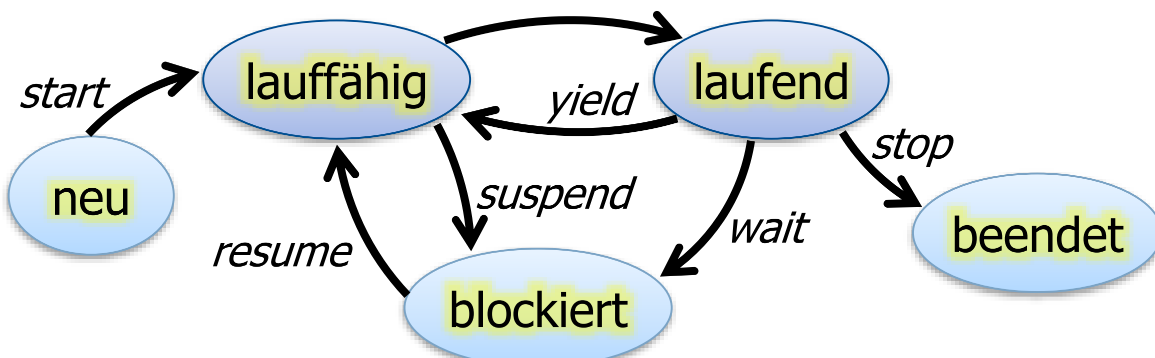
Klasse java.lang.Thread

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

- Methoden:

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

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




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

Besprechen wir gleich

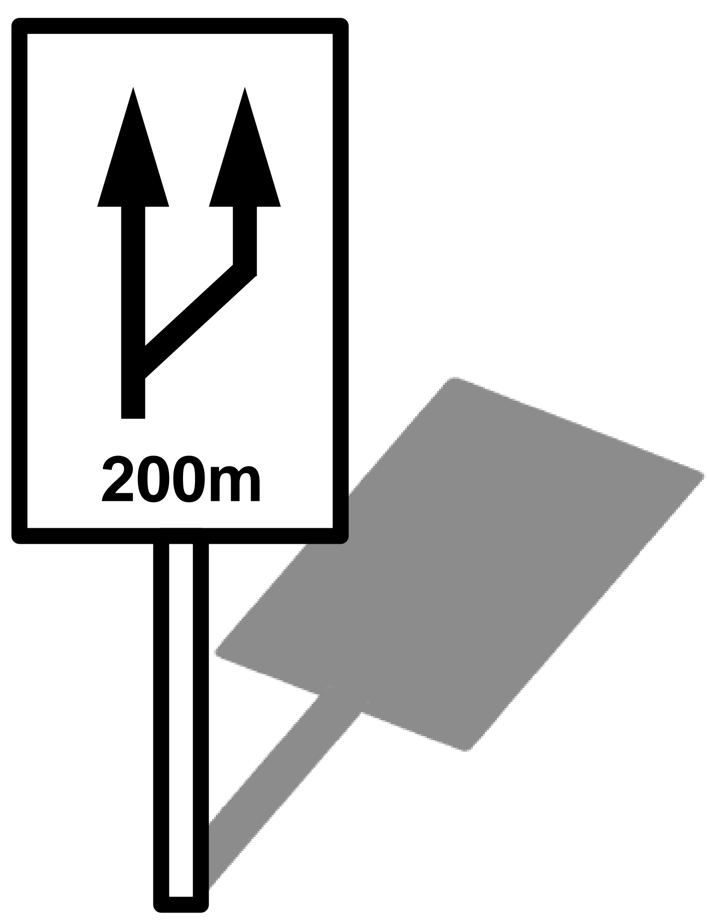
„Hintergrundprozess": terminiert *nicht* mit dem Erzeuger

Programmstruktur eines Thread

- Jeder Thread (genauer: jede von „Thread“ abgeleitete Klasse) muss eine void-Methode run() enthalten
  - Diese macht die eigentlichen Anweisungen des Thread aus!
  - „run“ ist in der Oberklasse „Thread“ nur abstrakt definiert
- Ein typisches Gerüst für einen Thread:

```
class Beispiel_Thread extends Thread {
  int myNumber;
  public Beispiel_Thread(int i) { // Konstruktor
    myNumber = i;
  }
  public void run() {  // läuft erst los bei "start"
    // hier die Anweisungen des Thread, z.B.:
    System.out.println("Gruss von Thread " + myNumber);
  }
  // hier weitere Methoden
}
```

Erzeugen eines Thread
(aus einem anderen Thread heraus)



Erzeugen eines Thread

```
Beispiel_Thread m = new Beispiel_Thread(5);
m.start();
```

Damit kann man auf den Thread zugreifen und diesen „kontrollieren" (z.B. m.suspend();)

Mit dieser Nummer identifizieren wir einen Thread individuell; jeder Thread merkt sich „seine" Nummer

- Alternativ: „anonyme" Erzeugung & Start

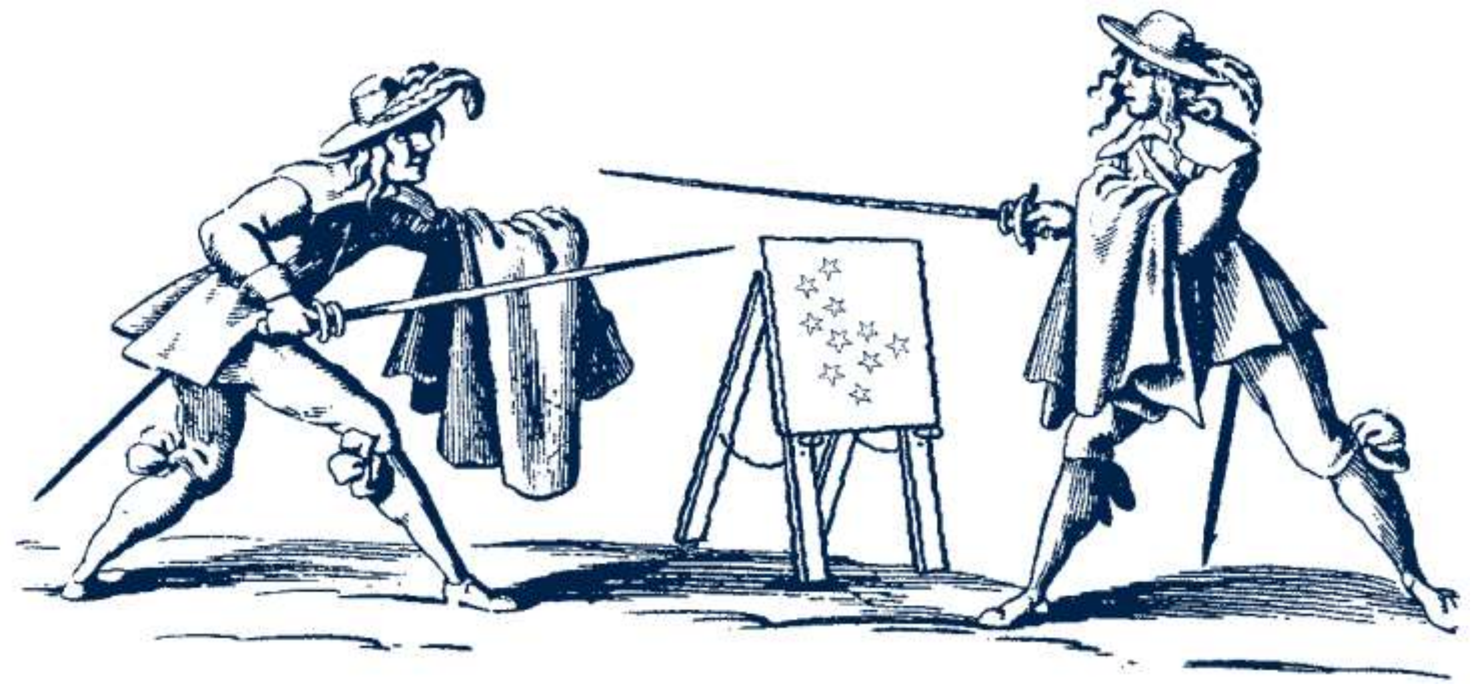
```
new Beispiel_Thread(5).start();
```

Wie wäre es, wenn „start" direkt im Konstruktor von Beispiel_Thread stattfände?

  - Zusammenfassen der beiden Anweisungen zum Gründen und Starten
  - Dann aber keine Kontrolle möglich, da kein Zugriff auf den Thread!

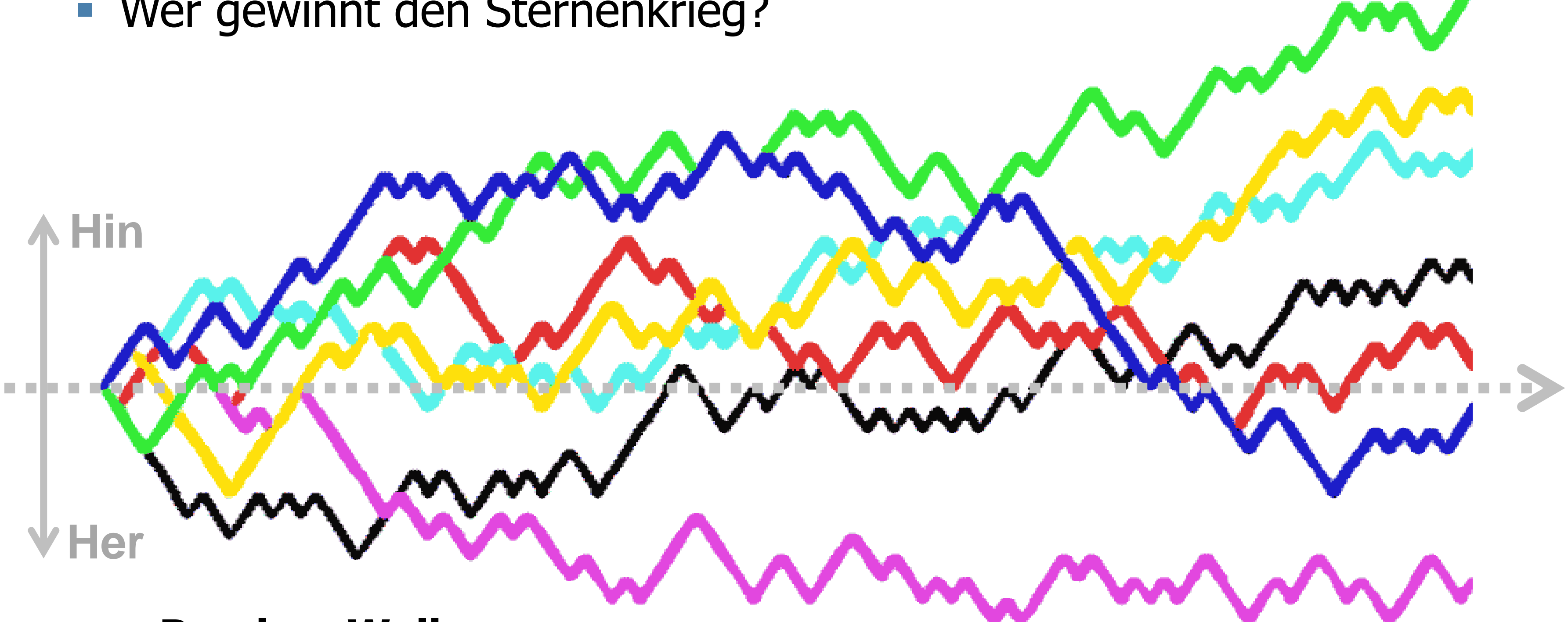
Ein Thread-Beispiel („Hin-Her")

- Ein Thread „Hin" schreibt Sterne; ein anderer Thread „Her" löscht Sterne; beide arbeiten (quasi)parallel
  - Wer gewinnt den Sternenkrieg?



Ein Thread-Beispiel („Hin-Her“)

- ********************************
- Ein Thread „Hin“ schreibt Sterne; ein anderer Thread „Her“ löscht Sterne; beide arbeiten (quasi)parallel
  - Wer gewinnt den Sternenkrieg?



Random Walk

Zwei Seelen wohnen, ach! in meiner Brust. – Goethe, Faust I.

Ein Thread-Beispiel („Hin-Her")

- ******************************
- Ein Thread „Hin" schreibt Sterne; ein anderer Thread „Her" löscht Sterne; beide arbeiten (quasi)parallel
  - Wer gewinnt den Sternenkrieg?

Gambler's ruin:
A gambler with a finite amount of money will eventually lose when playing a fair game against a bank with an infinite amount of money. The gambler's money will perform a random walk, and it will reach zero at some point, and the game will be over.

```
public class HinHer {
  public static void main (String args[]) {
    // Sternchenvorrat fuer den Anfang:
    System.out.print("********************");
    System.out.flush();
    new Hin().start();
    new Her().start();
  }
} // Zwei Seelen wohnen, ach! in meiner Brust.
```

Ausgabe des durch „print" gefüllten Puffers ohne „newline" (→ `System.out.flush();`)

Kommt es dazu überhaupt noch? Oder behält „Hin" die ganze Zeit über die Kontrolle? (→ `new Her().start();`)

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

Hin und Her

```
class Hin extends Thread {

  public void run() {
    try {
      while(true) {
        sleep((int)(Math.random() * 100));
        paint();
        System.out.flush();
      }
    }
    catch (InterruptedException e) {return;}
  }

  public void paint() { // Sternchen hinzufügen:
    System.out.print("*");
  }
}

class Her extends Hin {
  public void paint() { // Sternchen löschen:
    System.out.print("\b \b");
  }
}
```

Denkübung: Was geschieht, wenn ein Thread aufwacht, während der andere gerade mitten in „paint" ist?

Gemacht für die Ewigkeit

Exception, falls während des sleeps ein Interrupt ausgelöst wird

Methode „run" (und damit den Thread) beenden

Wer früh stirbt, lebt länger ewig

„Her" erbt die Methode „run" von „Hin"

Redefinition von „paint": Sternchen löschen mit \b (backspace) sowie Überschreiben mit Leerzeichen

Ein Leerzeichen („space")

Das Backspace-Steuerzeichen \b

Als „Steuerzeichen" bezeichnet man diejenigen Zeichen eines Zeichensatzes, die keine darstellbaren Zeichen repräsentieren. Ursprünglich wurden sie zur Ansteuerung von Fernschreibern oder Textdruckern (analog zu elektrischen Schreibmaschinen) verwendet. Durch Steuerzeichen ist es möglich, Steuerungsbefehle für die Ausgabegeräte – z.B. Zeilenvorschub („line feed"), Wagenrücklauf („carriage return"), Klingel – innerhalb des Zeichenstroms selbst zu übertragen.

Steuerzeichen werden in Zeichenketten durch ein vorangestelltes „\" codiert, z.B. \n („line feed" bzw. „new line") oder \b („backspace"). Letzteres bewegte bei klassischen Textdruckern den Druckkopf eine Position zurück; bei Displays soll der Cursor um ein Zeichen nach links rücken (allerdings ohne das zuvor Geschriebene zu löschen).

Ob die Steuerzeichen allerdings durch das jeweilige Ausgabegerät (bzw. die Systemroutinen des Betriebssystems) wirklich „korrekt" interpretiert werden und damit der gewünschte Effekt auftritt, kann Java nicht garantieren. Insbesondere dann, wenn nicht die Systemkonsole als Ausgabe verwendet wird, kann es daher geschehen, dass ein \b als „nicht druckbares Zeichen" interpretiert wird und auf dem Display z.B. als ☐ dargestellt wird.

Paralleles Inkrementieren – ein Rätsel

```
class ParIncr extends Thread {
  int i;
  static int j = 0;

  public void run() {
    for (i = 0; i < 400000000; i++) j++;
    System.out.println("i: " + i + "   j: " + j);
  }

  public static void main(String [] args) {
    for (int k = 0; k < 5; k++) new ParIncr().start();
  } // 5 parallele Threads erzeugen
}
```

i: 400 000 000	j: 259 355 470
i: 400 000 000	j: 312 541 405
i: 400 000 000	j: 352 604 350
i: 400 000 000	j: 298 886 974
i: 400 000 000	j: 400 046 632

?

- Was ist hier los? Wackelkontakt? Kann Java (bei static?) nicht mehr zählen?
- → Parallelität kann zu unerwartet merkwürdigen Phänomenen führen
  → Exaktes Verständnis, genaue Analyse und extreme Vorsicht nötig!
- Hand aufs Herz: Ist es (gesellschaftlich) zu verantworten, dass ein menschengemachtes System, ein Konzept,... ein solch unerwartetes Verhalten zeigt?

Paralleles Inkrementieren – ein Rätsel

- Ändert man „int i" in „static int i", dann wird das Ergebnis noch wilder:

```
i: 200088575   j: 200200195
i: 399627389   j: 212180131
i: 212677952   j: 212723778
i: 331412928   j: 331466194
i: 400000000   j: 400002083
```

- Ändert man dagegen „static int j" in „int j", dann verhält es sich normal und langweilig:

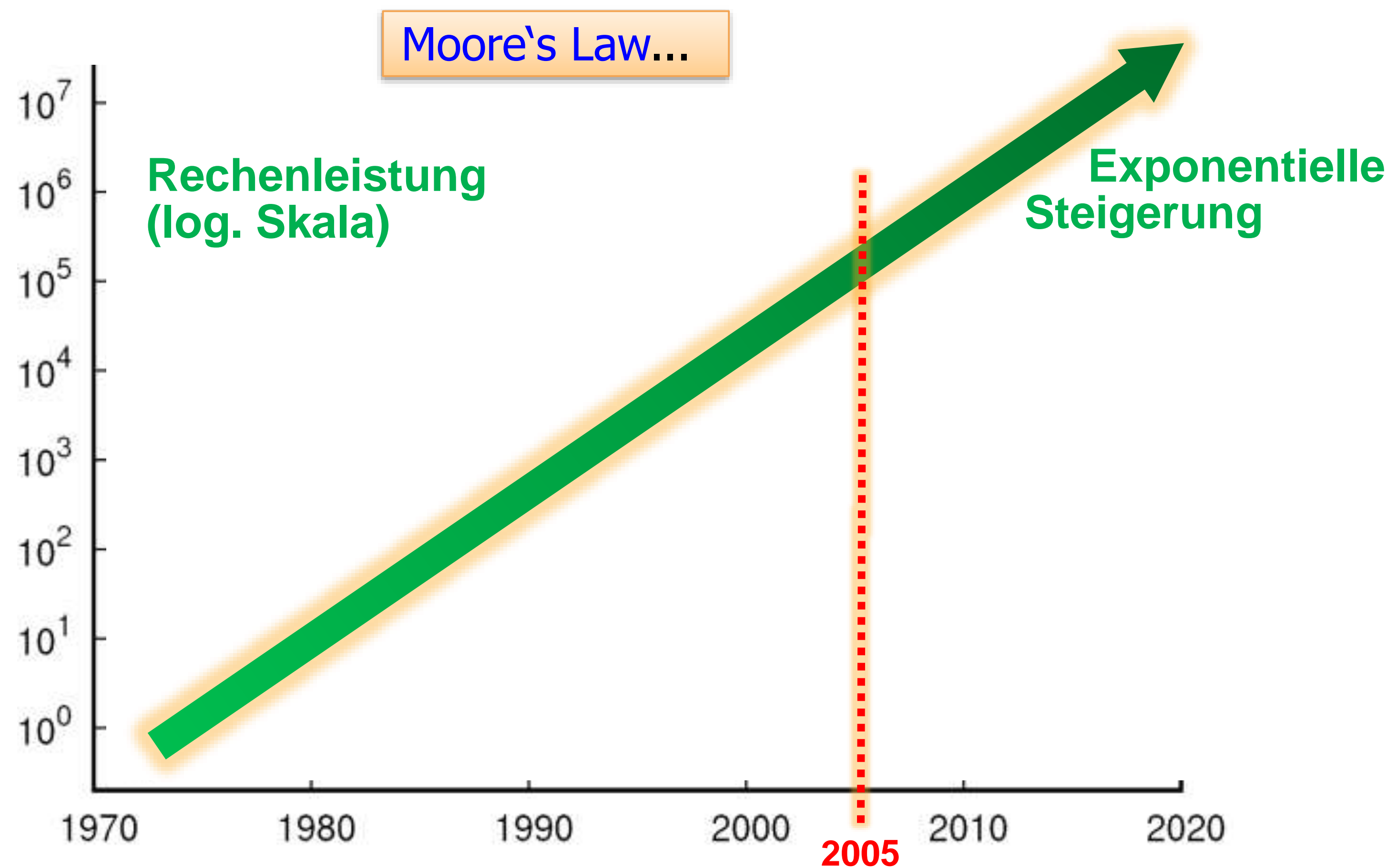
```
i: 400000000   j: 400000000
i: 400000000   j: 400000000
i: 400000000   j: 400000000
i: 400000000   j: 400000000
i: 400000000   j: 400000000
```

- Wenn man das wiederholt ausprobiert, dann können die „wilden" Zahlen jeweils ein bisschen anders aussehen; auf die kann man sich also auch nicht verlassen – haben wir hier einen Zufallszahlengenerator?
  - Was ist die Quelle diese Zufalls?
- Und vor allem: Was ist die Wurzel allen Übels?

Parallelität – schwierig, aber...

- Beherrschung der Phänomene der Parallelität ist
- Effizienzsteigerung mittels Parallelität ist ebenfalls

mühsam

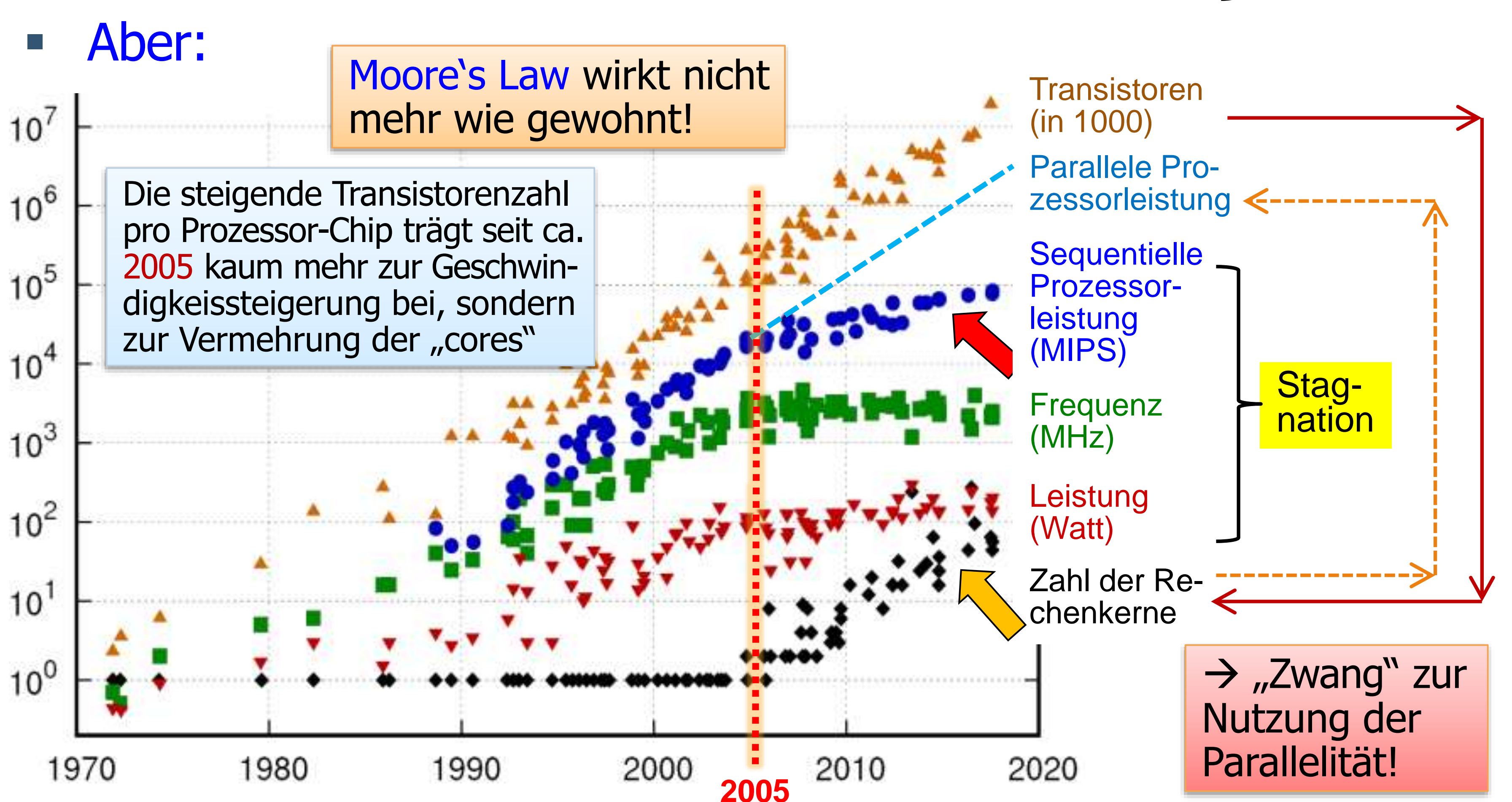


Parallelität – schwierig, aber immer wichtiger

- Beherrschung der Phänomene der Parallelität ist
- Effizienzsteigerung mittels Parallelität ist ebenfalls

} mühsam

- Aber:



Stichwort "Moore's Law"

Electronics, April 1965

Cramming more components onto integrated circuits

With unit cost falling as the number of components per circuit rises, by 1975 economics may dictate squeezing as many as 65,000 components on a single silicon chip

By Gordon E. Moore

Director, Research and Development Laboratories, Fairchild Semiconductor division of Fairchild Camera and Instrument Corp.

Gordon Moore, ca. 1965

Es begann mit einem Artikel im Jahr 1965, den Gordon Moore (Jahrgang 1929), späterer Mitgründer und CEO von Intel, im Fachjournal „Electronics" veröffentlichte.

Will lead to such wonders as home computers

Integrated circuits will lead to such wonders as home computers —or at least terminals connected to a central computer—automatic controls for automobiles, and personal portable communications equipment.

Darin prophezeite er aufgrund seiner Beobachtungen als Halbleiteringenieur der letzten Jahre, dass, falls die Technikentwicklung im Halbleiterbereich mit der „kostenneutralen“ jährlichen Verdoppelung der Zahl der Komponenten (also i.W. Transistoren) auf einem Chip länger anhalten sollte, wundersame Anwendungen möglich würden, u.a. Heimcomputer, Mobiltelefone sowie automatisch gesteuerte Autos. Kaum ein anderer vernünftiger Mensch hielt derartige Science-Fiction-Dinge seinerzeit für möglich – die Herausgeber der Zeitschrift fügten zur Besänftigung der Leser eine Karikatur in den Artikel ein, wo in einem Kaufhaus die handlichen Jedermann-Computer zwischen Kurzwaren und Kosmetikartikel angeboten wurden. Ein echter Witz!

A factor of two per year

„The complexity for minimum component costs has increased at a rate of roughly a factor of two per year (see graph on next page). Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000.“

Die empirische Grundlage für das, was ab ca. 1970 dann das mooresche Gesetz genannt wurde, war allerdings etwas dünn: Moore verband in einem halblogarithmischen Diagramm fünf Punkte der Jahre 1959 bis 1965 zu einer geraden Linie und extrapolierte diese 10 Jahre in die Zukunft, bis 1975. Er kam so auf ca. 65000 integrierte Transistoren pro Chip für das Jahr 1975. Tatsächlich war diese Vorhersage eine Punktlandung – 1975 präsentierte z.B. Hewlett-Packard mit dem HP-65 den ersten programmierbaren Taschenrechner der Welt, der etwa aus dieser Anzahl von Transistoren bestand. Das mooresche Gesetz behielt, gelegentlich etwas vage formuliert, über Jahrzehnte (und liberal interpretiert bis heute) Gültigkeit; es wurde für die Industrie zu einer „self fulfilling prophecy“ und stellt den basistechnologischen Treiber dessen dar, was heute gerne als „Digitalisierung“ bezeichnet wird.

Leseempfehlung: Peter Denning, Ted Lewis: Exponential Laws of Computing Growth. Commun. of the ACM, 60(1), pp. 54-65, Jan. 2017

1975: 65 000 Transistoren auf einem Chip!

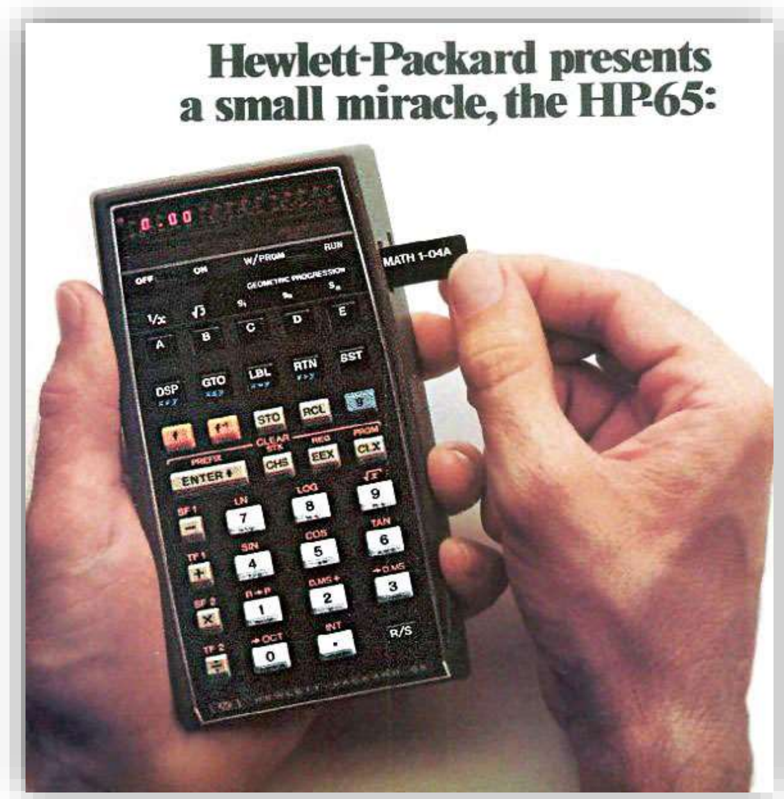
LOG$_2$ OF THE NUMBER OF COMPONENTS PER INTEGRATED FUNCTION

16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1959 1960 1961 1962 1963 1964 1965 1966 1967 1968 1969 1970 1971 1972 1973 1974 1975

YEAR

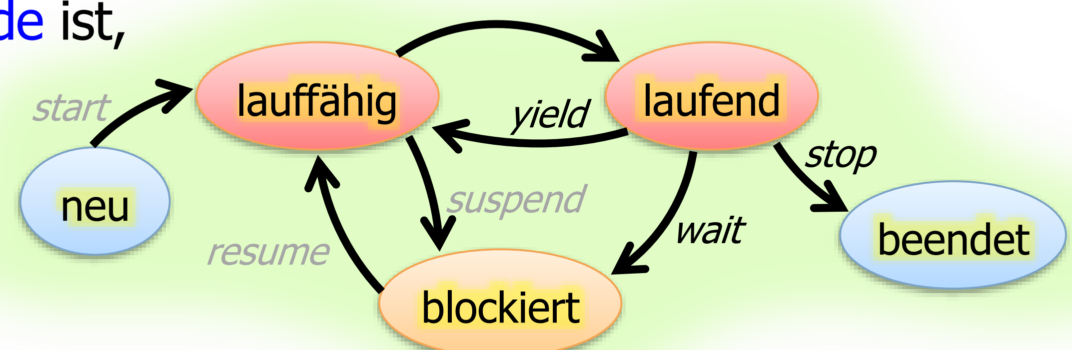
Die Extrapolation in Moores Artikel von 1965.



1975: Ein erstes Wunder wird wahr! Damalige Studenten träumten davon und sparten, um sich einen HP-65 leisten zu können; mancher Schüler liess ihn sich auch zum Abschluss der Schule schenken. (Steve Wozniak allerdings verkaufte ihn wieder, um seinen Anteil am Startkapital für die Firma Apple aufzubringen.)

Java: Thread-Steuerung

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



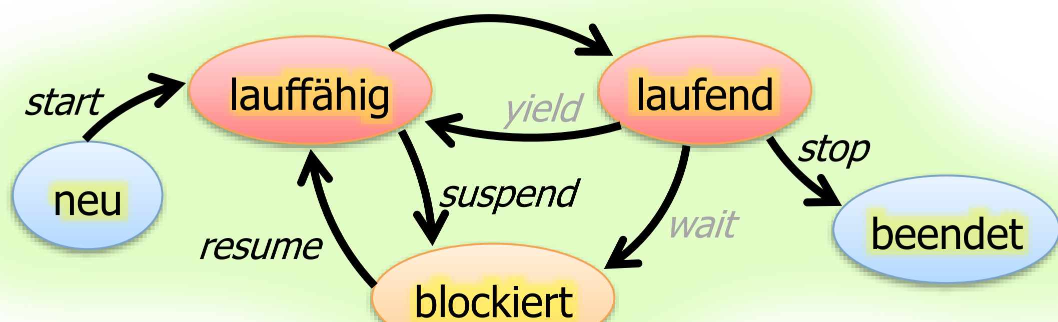
- Ein laufender Thread kann sich selbst
  - die CPU entziehen: yield()
    (Übergang in den Zustand „lauffähig"; wird automatisch wieder „laufend", wenn keine wichtigeren Threads mehr laufen möchten)
  - anhalten: wait()
  - schlafen legen: sleep(…) (wie wait, aber automatisches resume nach gegebener Zeit)
  - beenden: stop()
  - in der Priorität verändern: setPriority(…)

Priorität: min 1, max 10; anfangs: Priorität des erzeugenden Threads

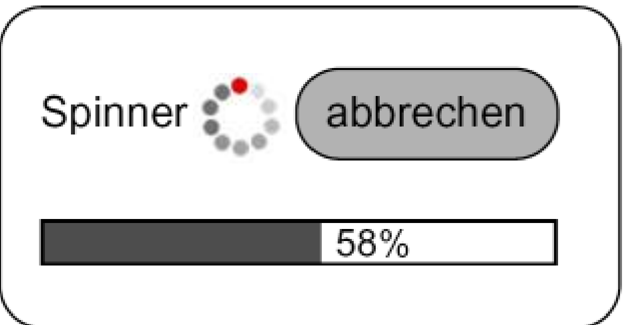
Java: Thread-Steuerung (2)

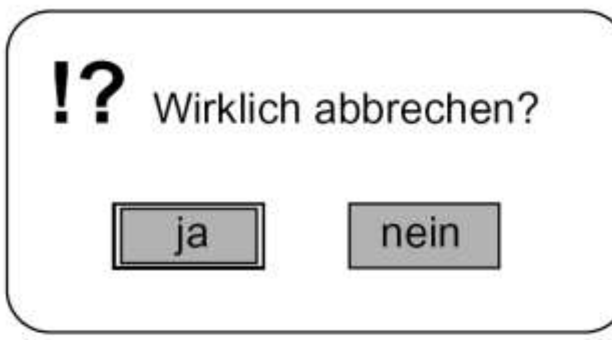
Hierzu ist eine Referenz auf den Thread notwendig

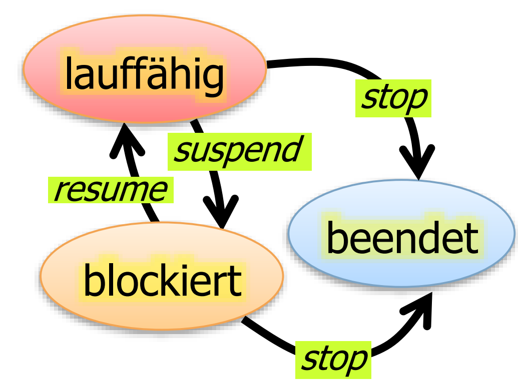
- Ein Thread kann <u>einen anderen</u> Thread t
  - starten: t.start()
  - anhalten: t.suspend()
  - fortsetzbar machen: t.resume();
  - beenden: t.stop()
  - in der Priorität verändern: t.setPriority(…)



Beispiel: Thread-Steuerung mit suspend / resume








```
class Spinner extends Thread // "endloser" Thread
    ...
void HitCancel() {   // In einem anderen Thread:
  Spinn.suspend(); // anhalten
  if (askYesNo("Wirklich abbrechen?","ja","nein"))
    Spinn.stop(); // abbrechen
  else
    Spinn.resume(); // weiter
} ...
```

- Für so etwas scheinen suspend, resume und stop ganz bequem
  - Aber: Diese drei Steuerungskommandos führen, unbedacht angewendet, zu unsicheren Programmen und sollten daher generell vermieden werden!

Stop is Being Deprecated

Auszug aus: Guido Krüger: Handbuch der Java-Programmierung.
Addison-Wesley, 2004, ISBN 3-8273-2201-4

Mit dem JDK 1.2 wurde die Methode stop als deprecated markiert, d.h., sie sollte nicht mehr verwendet werden. Der Grund dafür liegt in der potentiellen Unsicherheit des Aufrufs, denn es ist nicht voraussagbar und auch nicht definiert, an welcher Stelle ein Thread unterbrochen wird, wenn ein Aufruf von stop erfolgt. Es kann nämlich insbesondere vorkommen, dass der Abbruch innerhalb eines kritischen Abschnitts erfolgt (der mit dem synchronized-Schlüsselwort geschützt wurde) oder in einer anwendungsspezifischen Transaktion auftritt, die aus Konsistenzgründen nicht unterbrochen werden darf.

Die alternative Methode, einen Thread abzubrechen, besteht darin, im Thread selbst auf Unterbrechungsanforderungen zu reagieren. So könnte beispielsweise eine Membervariable cancelled eingeführt und beim Initialisieren des Thread auf false gesetzt werden. Mit Hilfe einer Methode cancel kann der Wert der Variable zu einem beliebigen Zeitpunkt auf true gesetzt werden. Aufgabe der Bearbeitungsroutine in run ist es nun, an geeigneten Stellen diese Variable abzufragen und für den Fall, dass sie true ist, die Methode run konsistent zu beenden.

Dabei darf cancelled natürlich nicht zu oft abgefragt werden, um das Programm nicht unnötig aufzublähen und das Laufzeitverhalten des Thread nicht zu sehr zu verschlechtern. Andererseits darf die Abfrage nicht zu selten erfolgen, damit es nicht zu lange dauert, bis auf eine Abbruchanforderung reagiert wird. Insbesondere darf es keine potentiellen Endlosschleifen geben, in den cancelled überhaupt nicht abgefragt wird. Die Kunst besteht darin, diese gegensätzlichen Anforderungen sinnvoll zu vereinen.

Stop, suspend, resume…

Thread.stop is being deprecated because it is inherently unsafe. Stopping a thread causes it to unlock all the monitors that it has locked. (The monitors are unlocked as the ThreadDeath exception propagates up the stack.) If any of the objects previously protected by these monitors were in an inconsistent state, other threads may now view these objects in an inconsistent state. Such objects are said to be damaged. When threads operate on damaged objects, arbitrary behavior can result. This behavior may be subtle and difficult to detect, or it may be pronounced. Unlike other unchecked exceptions, ThreadDeath kills threads silently; thus, the user has no warning that the program may be corrupted. The corruption can manifest itself at any time after the actual damage occurs, even hours or days in the future.

If you have been using Thread.stop in your programs, you should substitute that use with code that provides for a gentler termination. Most uses of stop can and should be replaced by code that simply modifies some variable to indicate that the target thread should stop running. The target thread should check this variable regularly, and return from its run method in an orderly fashion if the variable indicates that it is to stop running.

Thread.suspend is inherently deadlock-prone so it is also being deprecated, thereby necessitating the deprecation of **Thread.resume**. If the target thread holds a lock on the monitor protecting a critical system resource when it is suspended, no thread can access this resource until the target thread is resumed. If the thread that would resume the target thread attempts to lock this monitor prior to calling resume, deadlock results. Such deadlocks typically manifest themselves as “frozen” processes.

As with Thread.stop, the prudent approach is to have the “target thread” poll a variable indicating the desired state of the thread (active or suspended). When the desired state is suspended, the thread waits using Object.wait. When the thread is resumed, the target thread is notified using Object.notify.

http://docs.oracle.com/javase/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html

Thread-Ende

- Ein Thread ist beendet, wenn seine run-Methode beendet wird (evtl. auch „vorzeitig“ aufgrund einer nicht abgefangenen Exception) oder er durch „stop“ abgebrochen wird

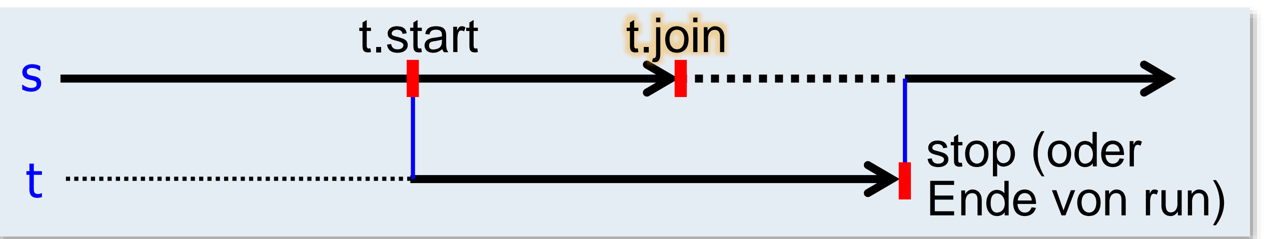
- Das Objekt eines beendeten Thread aber existiert weiter
  - Auf dessen Zustand kann also noch zugegriffen werden

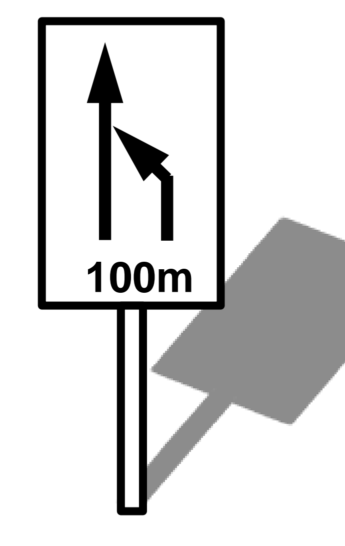
- Ein beendeter Thread kann mit start wieder neu loslaufen
  - Die run-Methode wird dann erneut ausgeführt

- join verwenden, wenn auf die Beendigung eines anderen Threads gewartet werden soll („Synchronisation“)
  - Z.B. weil man auf die von ihm berechneten Daten zugreifen will

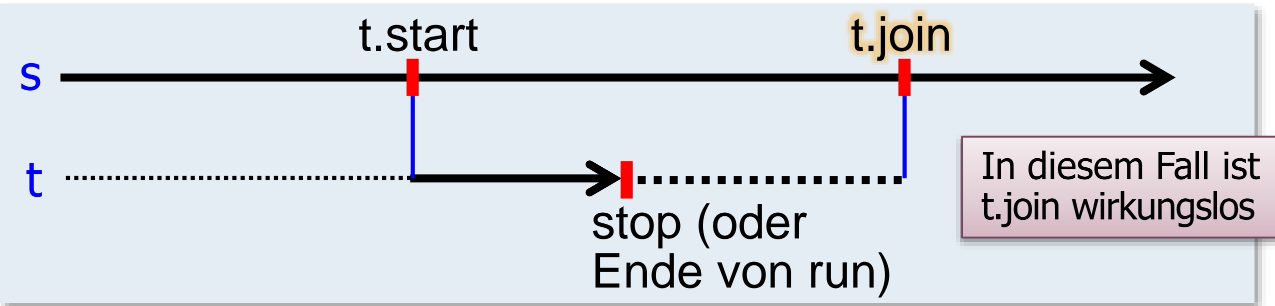
Join: Warten auf Ende eines anderen Threads

- Bsp: Thread s wartet so lang, bis t beendet ist:





- Alternativer Fall: Thread t ist schon früher fertig:



Nach t.join hat s in jedem Fall die Garantie, dass t beendet ist

- Variante mit Timeout-Parameter: t.join(m)
  - s wartet m Millisekunden auf das Ende von t; nach m Millisekunden (oder bereits früher: nach Beendigung von t) wird s wieder lauffähig

Rendezvous-Synchronisation

Join realisiert ein sogen. Rendezvous: Der erste wartet auf den anderen („Synchronisationspunkt")

Syn (griech.) = zusammen, gleich

Chronos (griech.) = Zeit

→ synchron = gleichzeitig

Beide treffen sich quasi gleichzeitig!
(bei „bewusstlosem" Warten)

Verallgemeinerung auf mehr als zwei Prozesse: Es geht erst weiter, wenn auch der letzte soweit ist („Barrierensynchronisation")

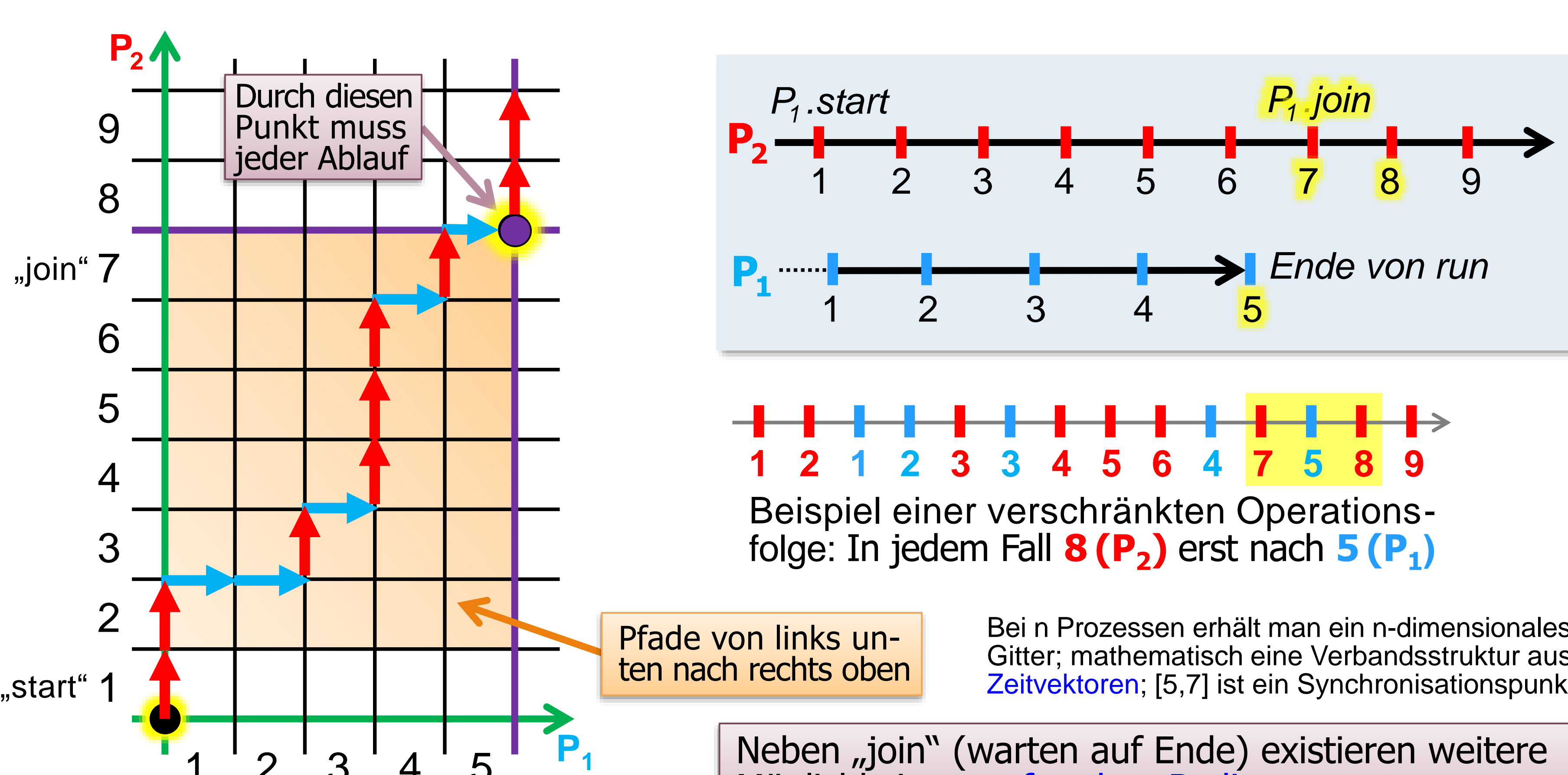
Gleiches Ergebnis, egal wer im Einzelfall zuerst da ist

Wir idealisieren die Realität durch Abstraktion!

Synchronisationspunkte im Interleaving-Modell

- Operationen seien instantan (brauchen keine Zeit)
- Zwei Operationen geschehen nie gleichzeitig

} Zeitlich verschränkte Operationsausführung (vgl. ereignisgesteuerte Simulation!)



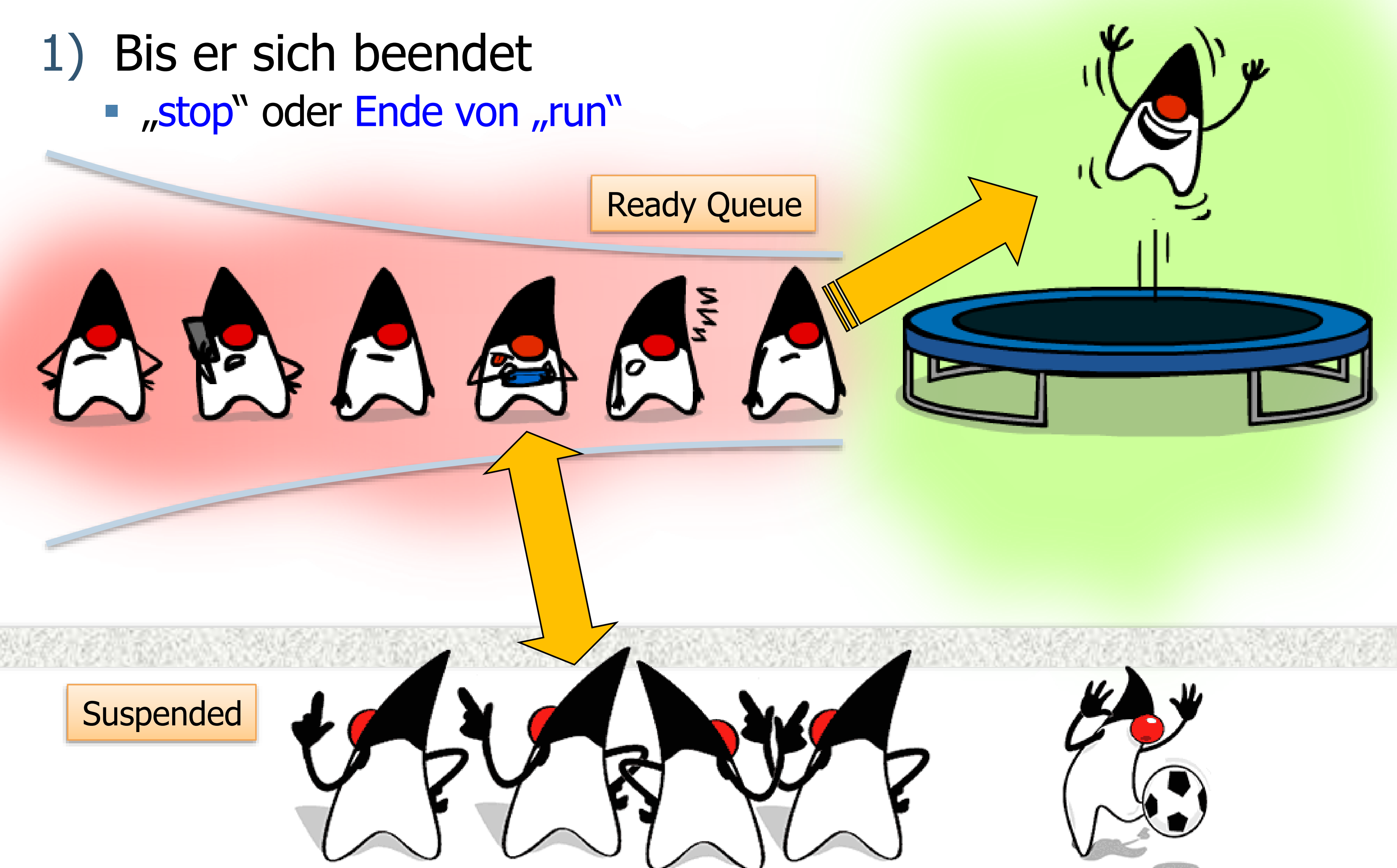
Beispiel einer verschränkten Operationsfolge: In jedem Fall 8 (P_2) erst nach 5 (P_1)

Bei n Prozessen erhält man ein n-dimensionales Gitter; mathematisch eine Verbandsstruktur aus Zeitvektoren; [5,7] ist ein Synchronisationspunkt

Neben „join" (warten auf Ende) existieren weitere Möglichkeiten, auf andere Bedingungen zu warten

Wie lange ist ein Thread laufend?

1) Bis er sich beendet
   - „stop“ oder Ende von „run“



Wie lange ist ein Thread laufend?

1) Bis er sich beendet
   - „stop“ oder Ende von „run“
2) Bis ein Thread höherer Priorität lauffähig wird
   - → Rückstufung nach „lauffähig“
   - Sofortiger Threadwechsel ist aber nicht garantiert!
3) Bis er mit „yield“ die Kontrolle dem Scheduler übergibt
   - Bzw. vom Scheduler zwangsweise die CPU entzogen bekommt
4) Bis er in den „blockiert“-Zustand übergeht
   - Explizit mit „suspend“, „wait“, „sleep“ etc.
   - Evtl. implizit bei warten auf E/A
     (es ist aber nicht garantiert, dass ein auf E/A wartender Thread die CPU tatsächlich für andere freigibt!)

Bei einer Multicore-CPU können mehrere Threads „echt“ gleichzeitig laufen!

Thread-Scheduling

- Scheduling: Planvolle Zuordnung der CPUs an die einzelnen Threads (jeweils für eine gewisse Zeitspanne)

- Genaue Scheduling-Strategie ist in Java nicht standardisiert
  - Kann jede VM-Implementierung für sich festlegen (und damit Eigenheiten des zugrundeliegenden Betriebssystems effizient nutzen)
  - Man darf sich daher nicht auf „Erfahrungen" verlassen (konkret: nicht auf die Wirkung von Zeitscheiben / Prioritäten)

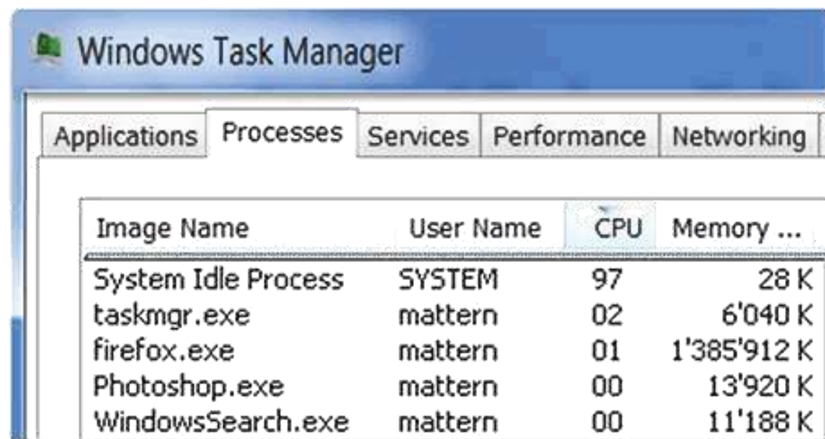
Schränkt Determiniertheit und Portabilität ein!

- Präemptives Scheduling mit Zeitscheiben kann von der VM realisiert sein (muss aber nicht)
  - Thread ist dann längstens bis zum Ablauf des aktuellen time slice „laufend"; danach entscheidet der Scheduler, wer den nächsten Zeitschlitz bekommt
  - Typischerweise dabei zyklisches Scheduling unter Threads gleicher Priorität

Thread 1
Thread 2
Thread 3
Thread 4
Thread 5

Prinzip eines zyklischen Zeitscheiben-Schedulers

```
// Ich bin der Anfang und das
// Ende, der Erste und der Letzte
while(true){
  if (current != last)
    next = current + 1;
  else
    next = 1;
  threadlist[current].suspend();
  threadlist[next].resume();
  current = next;
  sleep(TIMESLICE);
}
```



Der „System Idle"-Prozess nutzt die CPU am meisten

Hallo Leute,

Frage 1: Ist der System Idle Process wichtig?

Frage 2: Kann/darf man Ihn deaktivieren bzw. ihm weniger % in der Auslastung geben? Wenn ja, wie? Er nervt ja ganz schön, nimmt viel CPU-Leistung weg […]

Liebe Grüße, pc-freak

http://forum.chip.de/windows-vista/system-idle-process-1073965.html#post6475328

- Der Scheduler selbst sollte mit höchster Priorität laufen
  - Der „System Idle"-Prozess dagegen mit niedrigster Priorität
  - Bei Endlosschleifen in anderen Threads – wie könnte man garantieren, dass der Scheduler schliesslich wieder die Kontrolle erhält?

Prioritäten

- Implementierungsvorgabe: Ein Thread-Scheduler *soll* Threads mit höherer Priorität bevorzugen
  - Priorität entspricht initial der des Erzeuger-Thread
  - Priorität kann verändert werden (setPriority)
  - Wenn ein Thread mit höherer Priorität als der gegenwärtig ausgeführte lauffähig wird, wird der gegenwärtige i.Allg. unterbrochen

- Verwendung von Prioritäten
  - Niedrige Priorität für dauernd laufende „Hintergrundaktivitäten"
  - Höhere Priorität für seltene aber wichtige und eher kurze Aktionen (Benutzereingaben, Unterbrechungen...)
  - Prioritäten sollten nicht als Synchronisationsmittel (Erzwingen einer bestimmten Reihenfolge etc.) eingesetzt werden

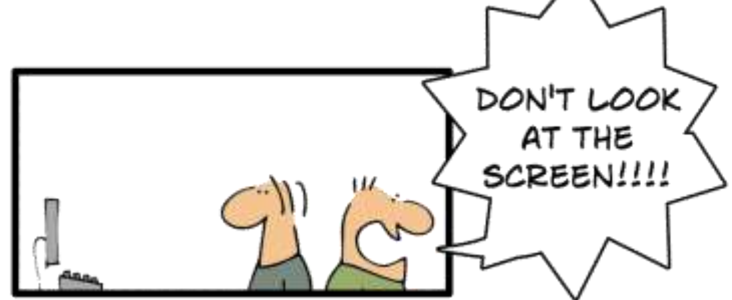
Threads: Schwierigkeiten

> The only thing worse than a problem that happens all the time is a problem that does not happen all the time – *J. Ousterhout*

- Ein Thread mit Endlosschleife kann u.U. das ganze System blockieren (so dass andere Threads „verhungern“)
  - Daher mit „yield“ dem Scheduler rücksichtsvoll und kooperativ helfen
  - Insbes. bei nicht-präemptivem Scheduling (keine Zeitscheiben) wichtig!

- Programmieren und „Debugging“ von Threads ist schwierig
  - Alle denkbaren verzahnten Abläufe („interleavings“) berücksichtigen
    - Menge der verzahnten Abläufe durch geeignete Synchro-nisation einschränken (nur „korrekte“ Abläufe zulassen)
    - Synchronisationsfehler finden ist besonders mühsam, da schlecht reproduzierbar (manchmal „Heisenberg-Effekt“: Testen ändert das Verhalten bzgl. des Fehlers)

HOW TO DEBUG “HEISENBUGS”



http://emeryblogger.com/category/concurrency/

Threads: Schwierigkeiten (2)

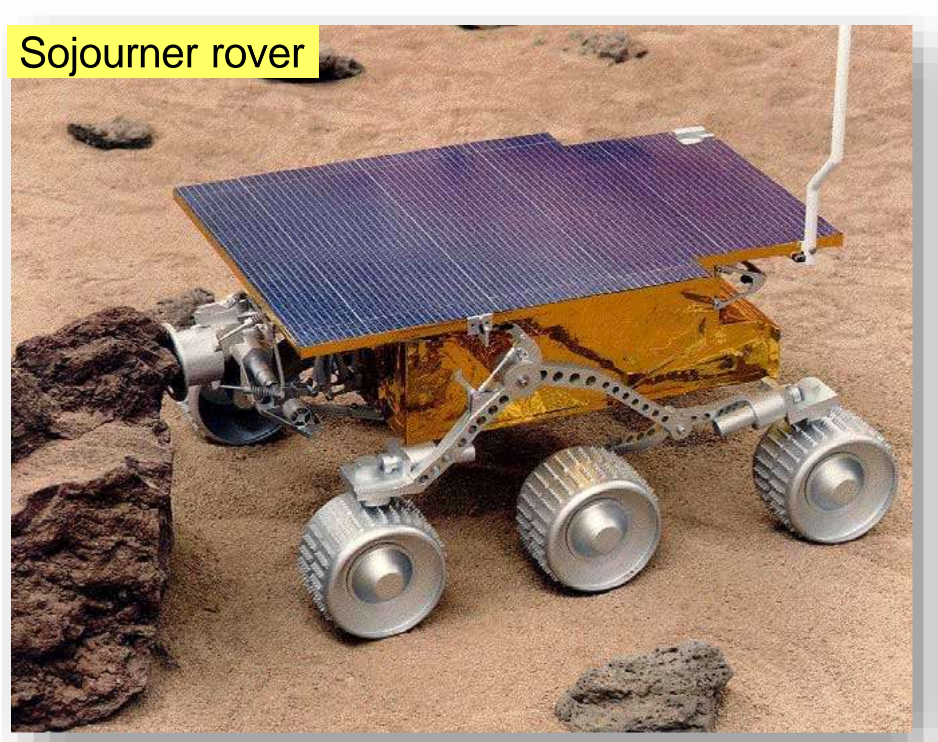
- Bei Prozessoren mit mehreren CPUs bzw. Rechenkernen („multicore") könnten entsprechend viele Threads „echt gleichzeitig" ausgeführt werden („multiprocessing")
  - Schon deswegen kein Verlass, dass Synchronisation bzw. wechselseitiger Ausschluss, realisiert mittels Prioritäten, funktioniert!
  - Böses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

- Portabilität ist bei dilettantischer Thread-Steuerung gefährdet
  - *"The setPriority and yield methods are advisory. They constitute hints from the application to the JVM. Properly written, robust, platform-independent code can use setPriority() and yield() to optimize the performance of the application, but should not depend on these attributes for correctness."*

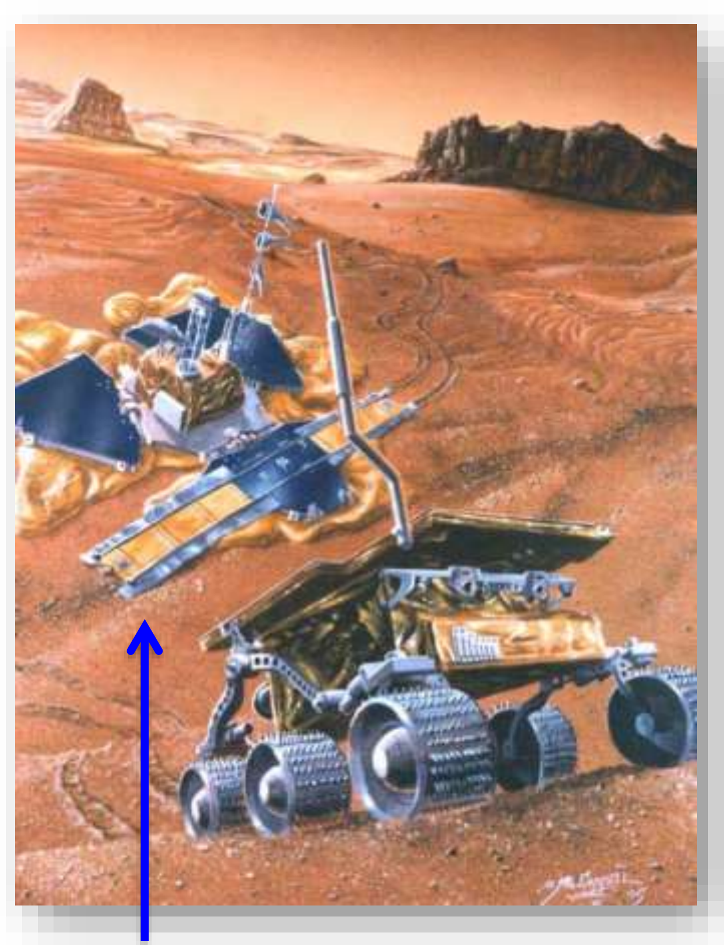
Historisch Notiz

Parallele Threads auf dem Mars: 1997

Mars Pathfinder Mission

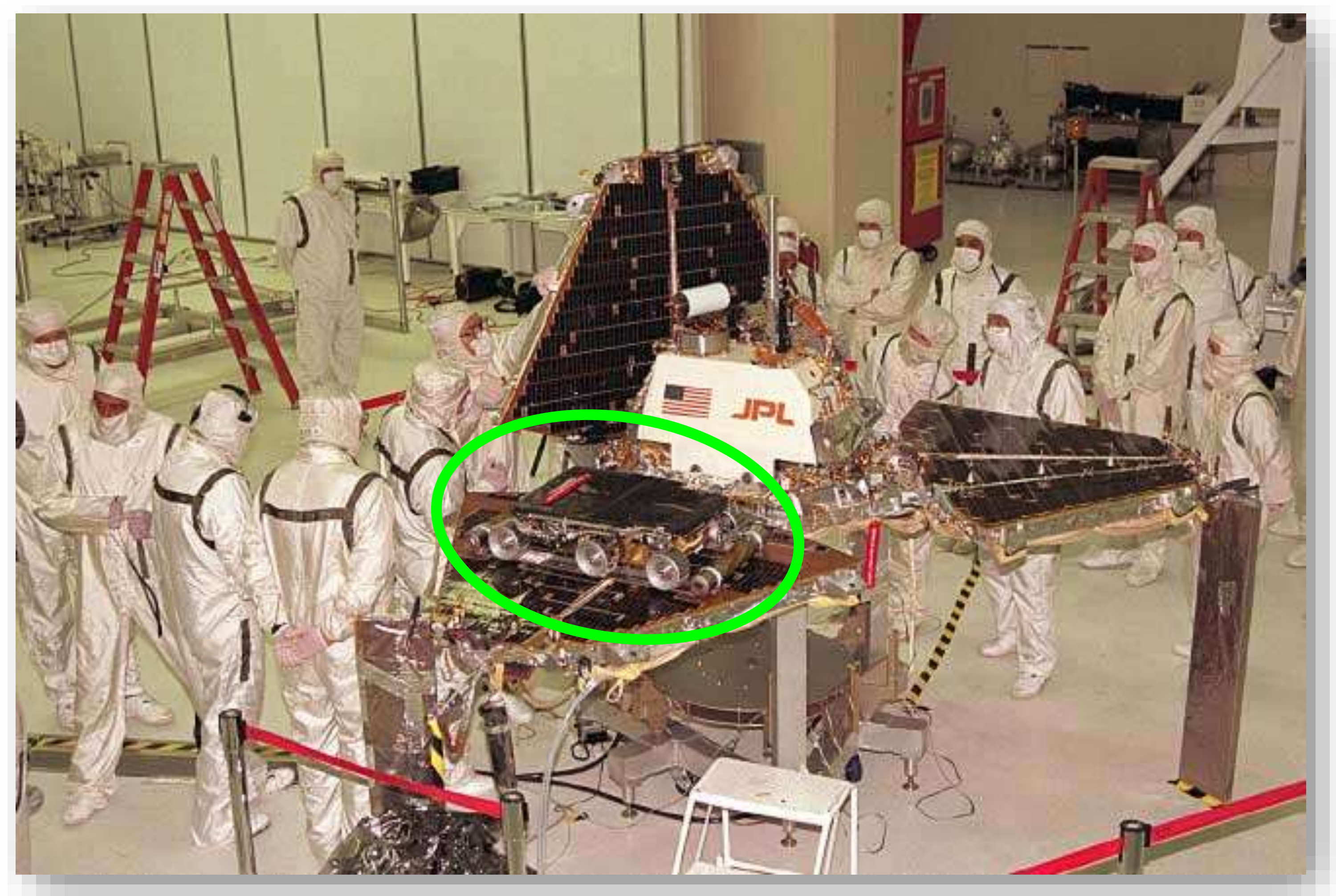


Computer im „Sojourner rover": 2 MHz Intel-80C85 8-Bit CPU, 512 kB RAM, 176 kB Flash



Im „Pathfinder lander": Strahlungsgehärteter 20 MHz PowerPC mit VxWorks Realzeit-Betriebssystem, 128 MB RAM, 6 MB EEPROM

Zusammenbau der Marssonde



Start 4. Dezember 1996

Landung 4. Juli 1997

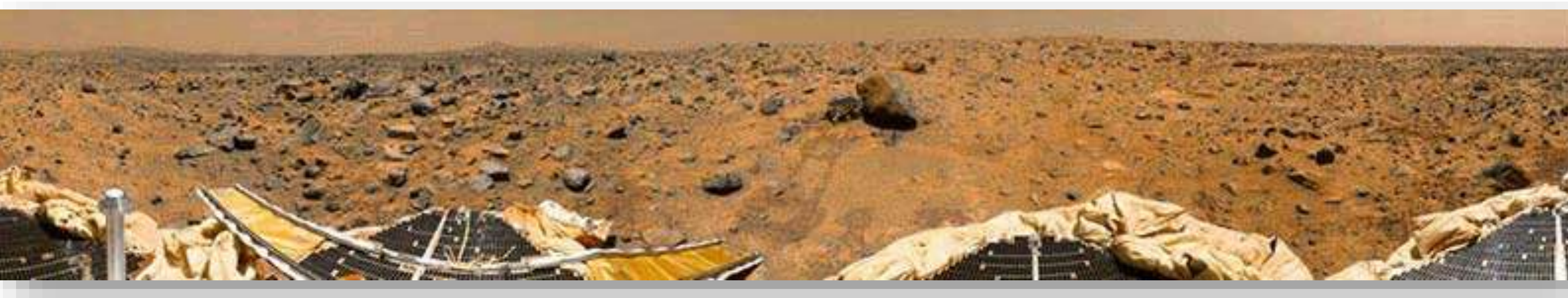
The lander first transmitted the engineering and atmospheric science data collected during landing.

The Lander's Computer Appeared to Reset Itself

Mission Status Report

JET PROPULSION LABORATORY
CALIFORNIA INSTITUTE OF TECHNOLOGY
NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
PASADENA, CALIF.
MISSION STATUS – 14 July 1997, 10:00 am PDT

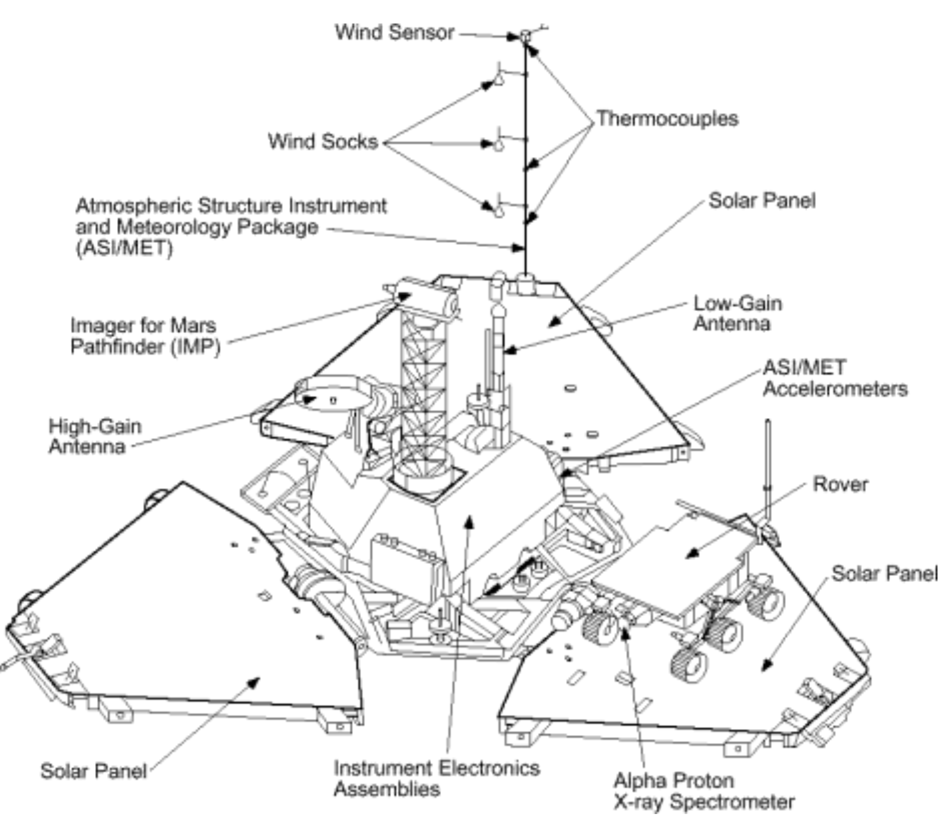
Mars Pathfinder's lander sent about an hour's worth of data to Earth last night – including portions of a 360-degree color panorama image – before the lander's computer appeared to reset itself, terminating the downlink session.



Software that Manages a Number of Different Activities Simultaneously

MISSION STATUS – 14 July 1997, 10:00 am PDT

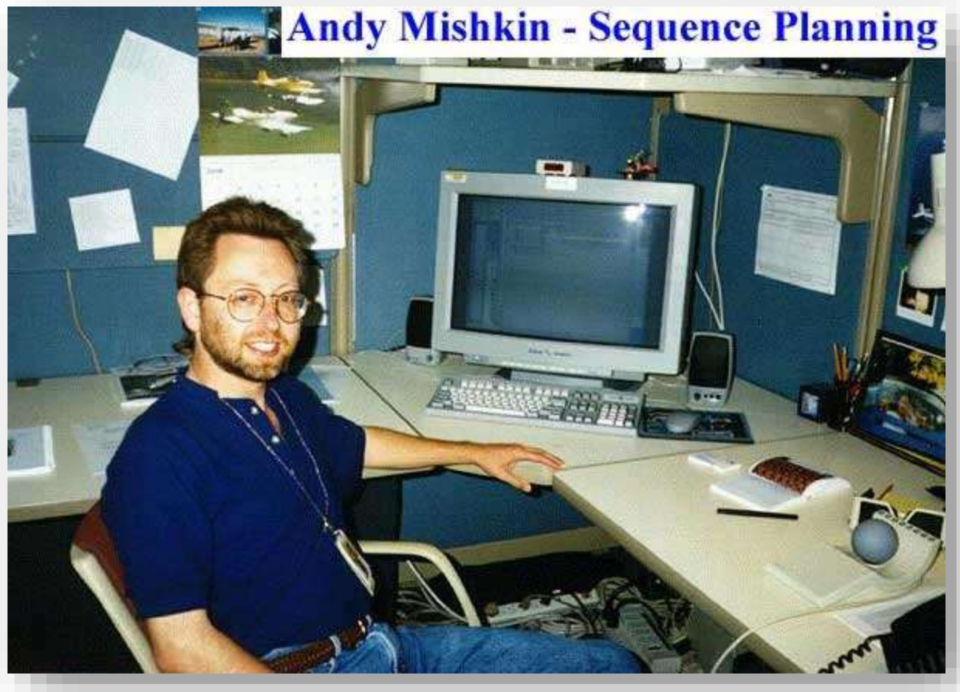
Engineers are continuing to debug the reset problem, which appears to be related to software that manages how the lander's computer handles a number of different activities simultaneously. "*Saturday night, we 'serialized' activities by having the lander do one thing at a time, whereas last night the lander was handling a number of activities when the reset occurred,*" said Brian Muirhead, Mars Pathfinder flight system manager. "*Tonight we will return to a 'serialized' approach to try to avoid the possibility of a reset.*" The reset occurred at 1:06 a.m. PDT, about halfway through a two-hour downlink session.



Handle One Activity at a Time!

MISSION STATUS – 15 July 1997, noon PDT

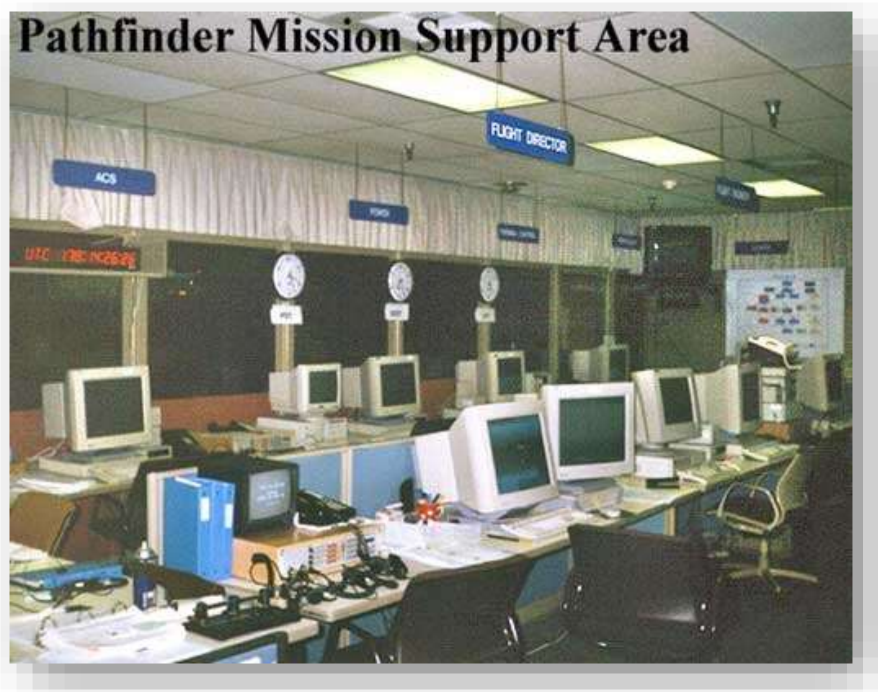
Recent incidents in which the Pathfinder lander's computer reset itself were discussed by Glenn Reeves, flight software team leader. According to him, computer resets have occurred a total of four times during the mission – on July 5, 10, 11 and 14. The flight team has attempted to avoid future resets by instructing the computer to handle one activity at a time – 'serializing' activities – rather than juggling a number of activities at once.



Considering Changes in the Flight Software

MISSION STATUS – 15 July 1997, noon PDT

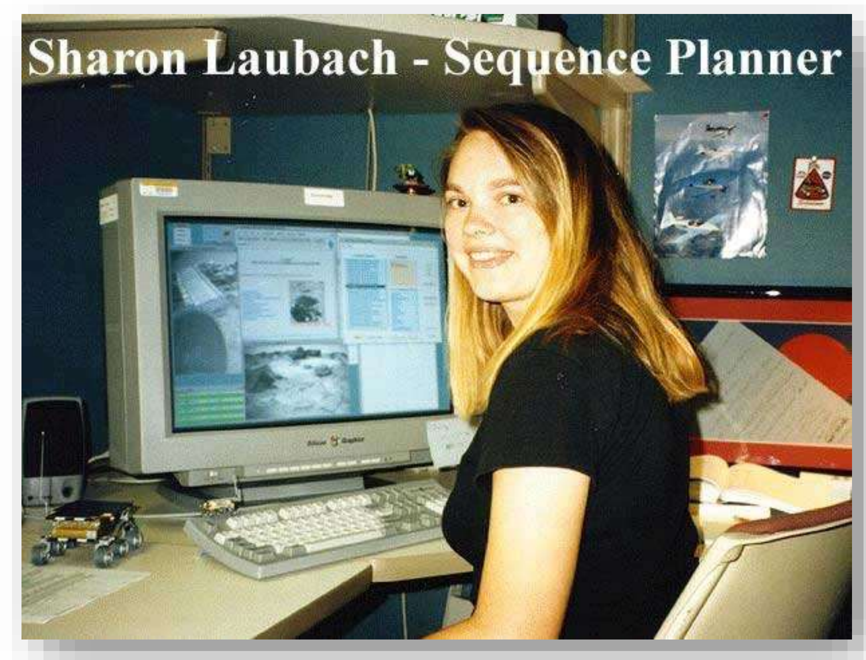
The team continues to trouble-shoot the problem by testing all of the sequences leading up to reset in JPL's Mars Pathfinder testbed; considering changes in the flight software that would allow for immediate recovery if the flight computer were to reset itself; and modifying operational activities to minimize data loss if a reset should occur again. "*In a sense, the reset itself is not harmful because it brings us back into a safe state*," said Reeves. "*But it does cause a disruption of the operational activities*."



The Task Had Not Been Given High Enough Priority

MISSION STATUS – July 17, 1997, 11 am PDT

Mars Pathfinder engineers... also noted that they have found and are in the process of fixing a software bug that had caused the lander's computer to reset itself four times in recent days. "*The resets on the lander computer were caused by a software task that was unable to complete the task in the allotted time*," said Flight Director Brian Muirhead. "*We found that the task was being cut short because it had not been given high enough priority to run through to completion. Basically, we just need to add one instruction to the computer software to raise the priority of that task.*"



The Problem was Reproduced and Isolated

MISSION STATUS – July 17, 1997, 11 am PDT

The problem was reproduced and isolated in testing at JPL. Further tests and verification will be completed today and tomorrow, with radio transmission of a software patch to change the lander's software scheduled for Saturday, Muirhead said.

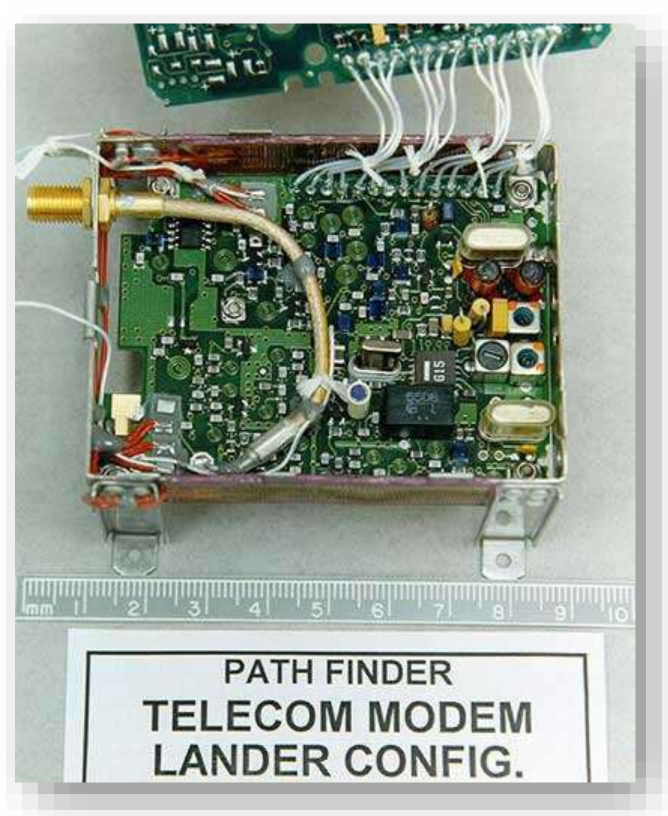
Sent a Software Update to Mars

MISSION STATUS – July 21, 1997, 10 am PDT

The team... also sent a software up-date to correct sequences on-board the flight computer which have caused it to automatically reset itself.

MISSION STATUS – July 24, 1997, 2:30 pm PDT

Flight Director Dave Gruel reported that no further flight software resets have occurred since the team sent modified flight software...



Robot Science & Technology - Premier Issue, 1998
www.doc4net.com/doc/1860848865700

What Really Happened on Mars?

From: Mike Jones
<mbj@microsoft.com>

I heard a fascinating keynote address by David Wilner, Chief Technical Officer of Wind River Systems. Wind River makes VxWorks, the real-time embedded systems kernel that was used in the Mars Pathfinder mission. In his talk, he explained in detail the actual software problems …

VxWorks provides preemptive priority scheduling of threads. Tasks on the Pathfinder spacecraft were executed as threads with priorities that were assigned in the usual manner reflecting the relative urgency of these tasks.

Pathfinder contained an "information bus", which you can think of as a shared memory area used for passing information between different components of the spacecraft. A bus management task ran frequently with high priority to move certain kinds of data in and out of the information bus. Access to the bus was synchronized with mutual exclusion locks (mutexes).

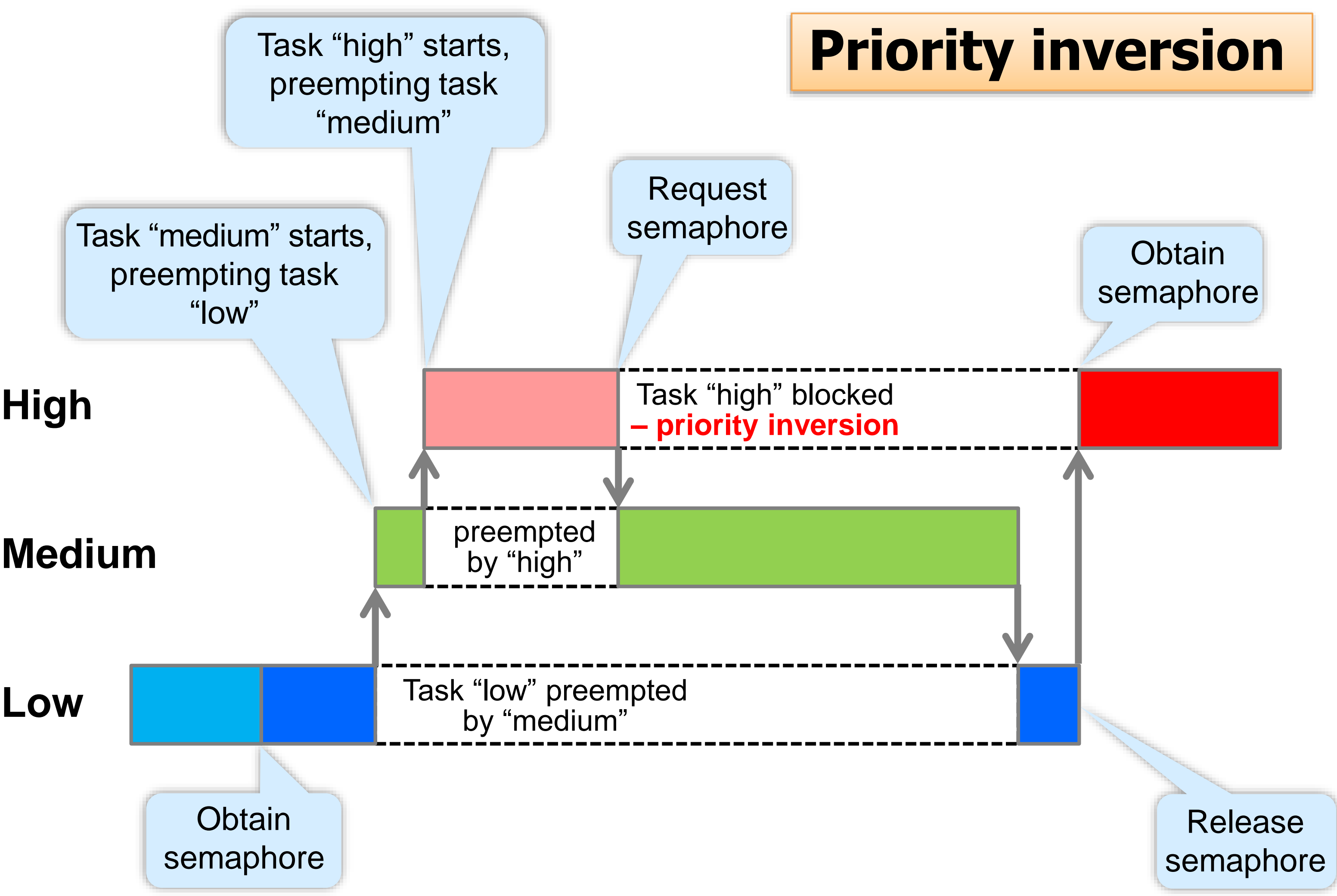
The meteorological data gathering task ran as an infrequent, low priority thread, and used the information bus to publish its data. When publishing its data, it would acquire a mutex, do writes to the bus, and release the mutex. If an interrupt caused the information bus thread to be scheduled while this mutex was held, and if the information bus thread then attempted to acquire this same mutex in order to retrieve published data, this would cause it to block on the mutex, waiting until the meteorological thread released the mutex before it could continue. The spacecraft also contained a communications task that ran with medium priority.

What Really Happened on Mars? (2)

Most of the time this combination worked fine. However, very infrequently it was possible for an interrupt to occur that caused the (medium priority) communications task to be scheduled during the short interval while the (high priority) information bus thread was blocked waiting for the (low priority) meteorological data thread. In this case, the long-running communications task, having higher priority than the meteorological task, would prevent it from running, consequently preventing the blocked information bus task from running. After some time had passed, a watchdog timer would go off, notice that the data bus task had not been executed for some time, conclude that something had gone drastically wrong, and initiate a total system reset. This scenario is a classic case of priority inversion.

How was this debugged? VxWorks can be run in a mode where it records a total trace of all interesting system events, including context switches, uses of synchronization objects, and interrupts. After the failure, JPL engineers spent hours and hours running the system on the exact spacecraft replica in their lab with tracing turned on, attempting to replicate the precise conditions under which they believed that the reset occurred. Early in the morning, after all but one engineer had gone home, the engineer finally reproduced a system reset on the replica. Analysis of the trace revealed the priority inversion. …

What Really Happened on Mars? (3)



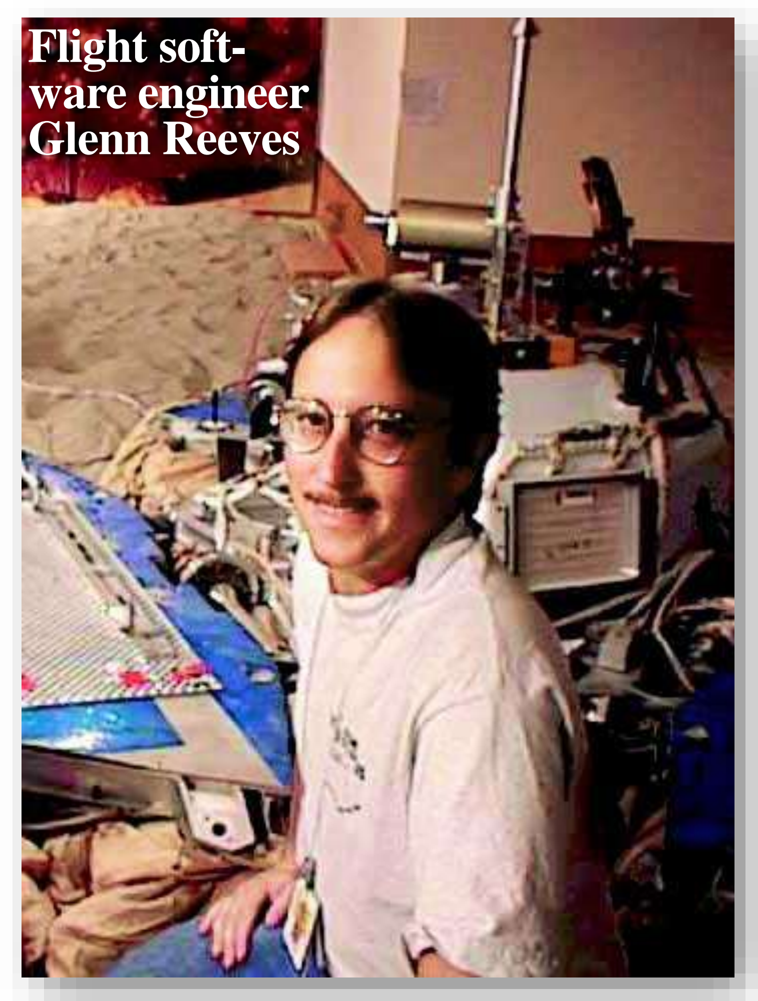
www.quora.com/What-really-happened-to-the-software-on-the-Mars-Pathfinder-spacecraft

What Really Happened on Mars? (4)

What caused the priority inversion was that Pathfinder's antenna performed better than expected. "It turned out that we got a much higher meteorological data rate, because we could point the antenna at Earth much better than we ever imagined," Reeves said. "We didn't expect it. We had never actually tested the thing with that high a set of data rates." …

When asked for any final comments on the priority inversion problem, Reeves said, "Even when you think you've tested everything that you can possibly imagine, you're *wrong*."

Robot Science & Technology - Premier Issue, 1998
www.doc4net.com/doc/1860848865700



What Really Happened on Mars? (5)

David told us that the JPL engineers later confessed that one or two system resets had occurred in their months of pre-flight testing. They had never been reproducible or explainable, and so the engineers, in a very human-nature response of denial, decided that they probably weren't important, using the rationale "it was probably caused by a hardware glitch". …

David also said that some of the real heroes of the situation were some people from CMU who had published a paper he'd heard presented many years ago who first identified the priority inversion problem and proposed the solution. [L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. IEEE Transactions on Computers, vol. 39, pp. 1175-1185, Sep. 1990]

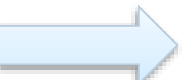
From: Mike Jones <mbj@microsoft.com>
Sent: Sunday, December 07, 1997 6:47 PM
Subject: What really happened on Mars?

http://research.microsoft.com/enus/um/people/mbj/mars_pathfinder/mars_pathfinder.html

Andere Softwareprobleme im Weltraum: 1962

Der meistzitierte Softwarefehler der IT-Geschichte

- 22. Juli 1962, Cape Canaveral / Florida: Start der ersten amerikanischen Venus-sonde „Mariner 1"
- Ausschnitt aus dem FORTRAN-Programm zur Steuerung der Flugbahn der Trägerrakete



- Der Fehler beruhte darauf, dass damals bei FORTRAN Leerzeichen nicht signifikant waren; DO 5 K = 1.3 (anstatt 1, 3) wurde (syntaktisch korrekt!) vom Compiler als Zuweisung von 1.3 an eine (implizit definierte) Variable DO5K verstanden!
- Der Start scheiterte; die Trägerrakete Atlas Agena B kam vom Kurs ab und wurde 290 Sekunden nach dem Start durch Funkbefehl gesprengt

```
IF (TVAL .LT. 0.2E-2) GOTO 40
DO 40 M = 1, 3
W0 = (M-1)*0.5
X = H*1.74533E-2*W0
DO 20 N0 = 1, 8
EPS = 5.0*10.0**(N0-7)
CALL BESJ(X, 0, B0, EPS, IER)
IF (IER .EQ. 0) GOTO 10
20 CONTINUE
DO 5 K = 1. 3
T(K) = W0
Z = 1.0/(X**2)*B1**2+3.0977E-
    4*B0**2
D(K) = 3.076E-
    2*2.0*(1.0/X*B0*B1+3.0977E-
    4**(B0**2-X*B0*B1))/Z
E(K) = H**2*93.2943*W0/SIN(W0)*Z
H = D(K)-E(K)
5 CONTINUE
10 CONTINUE
Y = H/W0-1
40 CONTINUE
```

Aber war dieser Programmierfehler wirklich die Ursache für den Fehlschlag? Dazu mehr auf der nächsten Slide.

Einzelne falsche Zeichen (z.B. „i" statt „a") können verwirren: „The Manchester Guardian had once famously misprinted that Queen Victoria had 'pissed over Westminster Bridge'. That story developed legs of its own and was probably applied by some jokers to most bridges she crossed for the rest of her life." www.irishexaminer.com/ireland/

Dirk Hoffmann schreibt dazu in seinem Buch „Software-Qualität“ (Springer-Verlag 2013):

Ein genauer Blick auf das Schleifenkonstrukt in Zeile 20 zeigt, dass die Intervallgrenzen nicht mit einem Komma, sondern mit einem Punkt separiert wurden. Dieser mit bloßem Auge kaum zu entdeckende Fehler hat dramatische Auswirkungen. Durch das Fehlen des Kommas erkennt FORTRAN den Ausdruck nicht mehr länger als Zählschleife, sondern interpretiert den Ausdruck schlicht als einfache Variablenzuweisung: DO5K = 1.3

Dass der Fehler durch den FORTRAN-Compiler nicht als solcher erkannt wird, liegt an zwei Besonderheiten der Sprache, die sie von den meisten anderen Programmiersprachen unterscheidet. Zum einen erlaubten frühe FORTRAN-Versionen, Leerzeichen an beliebiger Stelle einzufügen – insbesondere auch innerhalb von Bezeichnern und Variablennamen. Diese Eigenschaft erscheint aus heutiger Sicht mehr als fahrlässig. Zur damaligen Zeit bot sie jedoch durchaus Vorteile, da die ersten FORTRAN-Programme noch auf Lochkarten gespeichert wurden. Werden alle Leerzeichen ignoriert, kann ein Programm selbst dann noch erfolgreich eingelesen werden, wenn zwischen zwei gestanzten Zeilen versehentlich eine ungestanzte übrig bleibt.

Zum anderen ist es in FORTRAN gar nicht nötig, Variablen vor ihrer ersten Nutzung zu deklarieren. Hier wird besonders deutlich, wie wertvoll die Bekanntmachung von Funktionen und Variablen in der Praxis wirklich ist. Dass eine Variable DO5K bereits an anderer Stelle deklariert wurde, ist so unwahrscheinlich, dass jeder Compiler die Übersetzung der mutmaßlichen Zuweisung verweigert hätte. Kurzum: Die Verwendung einer deklarationsbasierten Programmiersprache, wie z. B. C, C++ oder Java, hätte den Software-Fehler des Mercury-Projekts vermieden – der Fehler wäre bereits zur Übersetzungszeit durch den Compiler entdeckt worden.

Es bleibt die Frage zu klären, wie der hier vorgestellte FORTRAN-Bug eine derart große Berühmtheit erlangen konnte, um heute zu den meistzitierten Softwarefehlern der IT-Geschichte zu zählen? Die Antwort darauf ist simpel. Der vorgestellte Fehler demonstriert nicht nur wie kaum ein anderer die Limitierungen der Sprache FORTRAN, sondern zeichnet zugleich für eine der größten Legenden der Computergeschichte verantwortlich. Auf zahllosen Internet-Seiten, wie auch in der gedruckten Literatur, wird der FORTRAN-Bug beharrlich als Ursache für den Absturz der Raumsonde Mariner I gehandelt, die am 22. Juli 1962 von der NASA an Bord einer Atlas-Trägerrakete auf den Weg zur Venus gebracht werden sollte. Kurz nach dem Start führte die Trägerrakete unerwartet abrupte Kursmanöver durch und wich deutlich von der vorbestimmten Flugbahn ab. Alle Versuche, korrigierend einzugreifen, schlugen fehl. Nach 290 Sekunden fällte die Flugkontrolle schließlich die Entscheidung, die Trägerrakete aus Sicherheitsgründen zu sprengen.

Die Ursache für den Absturz der Mariner-Trägerrakete ist weit unspektakulärer als gemeinhin angenommen und geht schlicht auf die falsche Umsetzung der Spezifikation zurück. Obwohl die Anforderungsbeschreibung der Flugsteuerung korrekt vorgab, die Verlaufskurve eines Messwerts geglättet zu verwenden, wurde diese in der Implementierung ungeglättet weiterverarbeitet. Trotzdem: Der FORTRAN-Bug der Mercury-Mission wird heute immer noch so beständig mit dem Mariner-Absturz in Verbindung gebracht, dass er wahrscheinlich auch in Zukunft als spektakuläre Erklärung für das Scheitern dieser Mission herhalten muss.

Andere Softwareprobleme im Weltraum: 1999

The Mars Climate Orbiter that never made it into orbit

Zum Mars-Surveyor-'98-Programm der NASA zitieren wir nochmals Dirk Hoffmann:

Im Rahmen der Mission wurden zwei Raumsonden zum Mars geschickt, um dessen atmosphärische Bedingungen detailliert zu erkunden. Während eine der Sonden, der Polar Lander, auf der Marsoberfläche aufsetzen sollte, war es die Aufgabe des Climate Orbiters, den roten Planeten auf einer kreisförmigen Umlaufbahn zu umrunden und aus sicherer Entfernung zu analysieren.

[…] Pünktlich am 23. September 1999 wurde die Anflugphase [des Climate Orbiters] eingeleitet […]. Die Konzeption sah vor, die letzte Phase der Annäherung mit Hilfe eines als Aerobraking bezeichneten Prinzips durchzuführen. […] Hierzu tritt die Sonde zunächst in einen elliptischen Orbit ein, dessen nächster Punkt nur ca. 150 km von der Marsoberfläche entfernt ist und damit bereits die obersten Schichten der Atmosphäre streift. Durch die atmosphärische Reibung wird die Raumsonde leicht abgebremst, so dass sich die elliptische Umlaufbahn mit jedem Umlauf langsam an den später einzunehmenden kreisförmigen Orbit annähert. Nach 57 Tagen ist die Aerobraking-Phase beendet und die finale Kreisbahn erreicht.

Um 9:06 Uhr trat der Mars Climate Orbiter in den Funkschatten des roten Planeten ein, den er um 9:27 verlassen und den Kontakt wiederherstellen sollte. Die Kontrollstation wartete jedoch vergeblich und nach kurzer Zeit war klar, dass die Sonde unwiderruflich verloren war. Später stellte sich heraus, dass sich die Sonde bis auf 57 km der Marsoberfläche genähert hatte – rund 100 km weniger als vorberechnet. Die hohe atmosphärische Reibung in dieser niedrigen Höhe ließ den Orbiter in kürzester Zeit in einem hellen Feuerball verglühen.

Die Analyse der Telemetriedaten brachte den Fehler noch am selben Tag zum Vorschein. Zur Kurskorrektur beim Landeanflug griff der Orbiter auf eine von Lockheed Martin bereitgestellte Lookup-Tabelle zurück, das sogenannte small forces file. Lockheed Martin legte die Daten in imperialen Einheiten lbs x s *(pound force seconds)* ab, von der NASA wurden die Werte aber, wie international üblich, nach dem metrischen System als N x s *(Newton-Sekunden)* interpretiert. Die verwendeten Werte waren dadurch um den Faktor 4.45 zu groß, und die Überkompensation der zu korrigierenden Bahnabweichung drängte den Mars-Orbiter schließlich auf die fatale Umlaufbahn in nur noch 57 km Höhe.

Andere Softwareprobleme im Weltraum: 1999

Mars Polar Lander killed by premature touchdown celebration

Zur Schwestersonde Mars Polar Lander wieder Dirk Hoffmann:

Der Verlust des Mars Climate Orbiters war ein herber Rückschlag für die NASA. Zum vollständigen Desaster wurde das Projekt schließlich am 3. Dezember 1999, als auch noch der Mars Polar Lander verloren ging. Gegen 21:10 mitteleuropäischer Zeit, just im Moment des Eintritts in die Marsatmosphäre, brach der Funkkontakt für immer ab. […] Die mutmaßliche Ursache für das Versagen wird, wie im Falle des Climate Orbiters, ebenfalls der Software zugeschrieben. Vermutlich wurden durch das Ausfahren der Landebeine Vibrationen erzeugt, die von der Software als das Aufsetzen der Sonde auf den Mars interpretiert wurden. Die fatale Fehleinschätzung führte zum sofortigen Abschalten der Bremsdüse, so dass die Sonde [aus einer Höhe von ca. 40m] mit unverringerter Geschwindigkeit auf der Marsoberfläche aufschlug und zerschellte.

Der Misserfolg hindert nicht daran, das 20-jährige Jubiläum zu feiern: 2019 erinnern Briefmarken der Malediven und der Zentralafrikanischen Republik daran.

Andere Softwareprobleme im Weltraum: 2004

Mars Rover Spirit caught in a rebooting cycle

Im Januar 2004 landeten die beiden Weltraum-sonden „Spirit" und „Opportunity" auf dem Mars. Dabei gab es ernste Probleme mit „Spirit":

January 22, 2004 – The team is still trying to diagnose the cause of earlier communications difficulties that have prevented any data being returned from Spirit since early Wednesday.

January 23, 2004 – Spirit's flight software is not functioning normally. It appears to have rebooted the rover's computer more than 60 times in the past three days.

January 24, 2004 – Hours before NASA's Opportunity rover will reach Mars, engineers have found a way to communicate reliably with its twin, Spirit, and to get Spirit's computer out of a cycle of rebooting many times a day.

January 26, 2004 – Engineers found a way to stop Spirit's computer from resetting itself about once an hour by putting the spacecraft into a mode that avoids use of flash memory.

February 1, 2004 – NASA's Mars Exploration Rover Spirit is healthy again. Part of the cure has been deleting thousands of files from the rover's flash memory. Many of the deleted files were left over from the seven-month flight from Florida to Mars. Onboard software was having difficulty managing the flash memory, triggering Spirit's computer to reset itself about once an hour.

www.jpl.nasa.gov/releases/2004/

Andere Softwareprobleme im Weltraum: 2011

Mars Rover Curiosity: reboots triggered by star tracker

Der Mars-Rover „Curiosity“ wurde am 26. 11. 2011 an Bord einer Atlas V von Cape Canaveral gestartet. Bereits drei Tage nach dem Start traten Problemen beim Navigations-system auf: Zur Ermittlung der aktuellen Position und der Ausrichtung im Weltraum verfügt Curiosity über Sternsen-soren und Sonnensensoren. Erstere beobachten mehrere speziell ausgewählte Fixsterne, welche als Leitsterne die-nen. Sonnensensoren benutzten dagegen ausschliesslich die Sonne als Referenzpunkt. Als das Navigationssystem der Raumsonde von den während der Startphase aktiven Sonnen- auf die Sternsensoren umgeschaltet wurde, kam es zu einer Exception mit Computer-Reset.

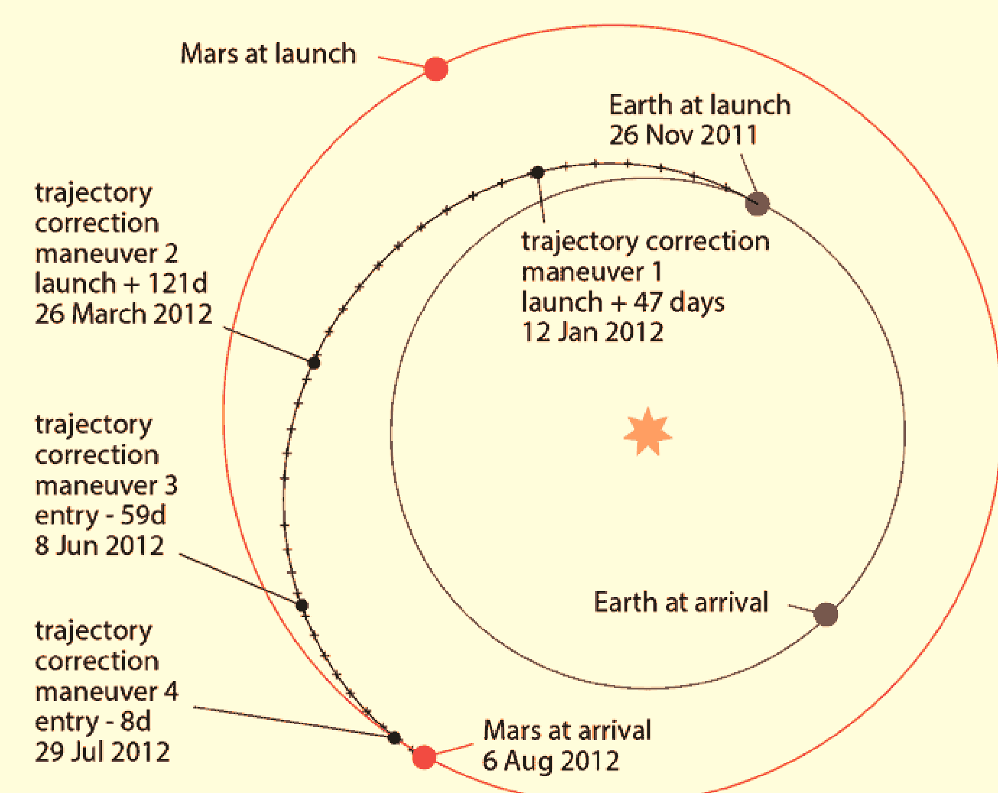
Mission status report, December 1, 2011: *The space-craft experienced a computer reset on Tuesday apparently related to star-identifying software in the attitude control system. The reset put the spacecraft briefly into a precau-tionary safe mode. Engineers restored it to normal oper-ational status for functions other than attitude control while planning resumption of star-guided attitude control.*

www.collectspace.com/review/msl_launch01-lg.jpg

Später stellte sich heraus, dass ein Fehler in der Software des Speichermanagements un-ter bestimmten Umständen zu falschen Zugriffen auf den Befehlscache des Prozessors führte. Dadurch gingen einige Kommandos verloren, was eine fehlerhafte Programmaus-führung bewirkte und zum Reset und Übergang in den „sicheren Modus“ führte.

Engineers rushed to develop a solution (2011)

Einige Passagen aus dem Buch "The Design and Engineering of Curiosity" von Emily Lakdawalla:



It took weeks to track down the root cause of the problem. […] Without use of the star scanner, the spacecraft could not turn to keep its solar panels and radio antenna precisely pointed at Earth. A planned December 11 trajectory correction maneuver couldn't be performed without turning the spacecraft. Without the star scanner they couldn't determine the spacecraft's ori-fentation and spin rate precisely, as required to time the position and duration of the multiple jet firings for the maneuver. With every passing day, the spacecraft's orientation drifted farther away from the optimal direction, so engineers rushed to develop a solution to the problem.

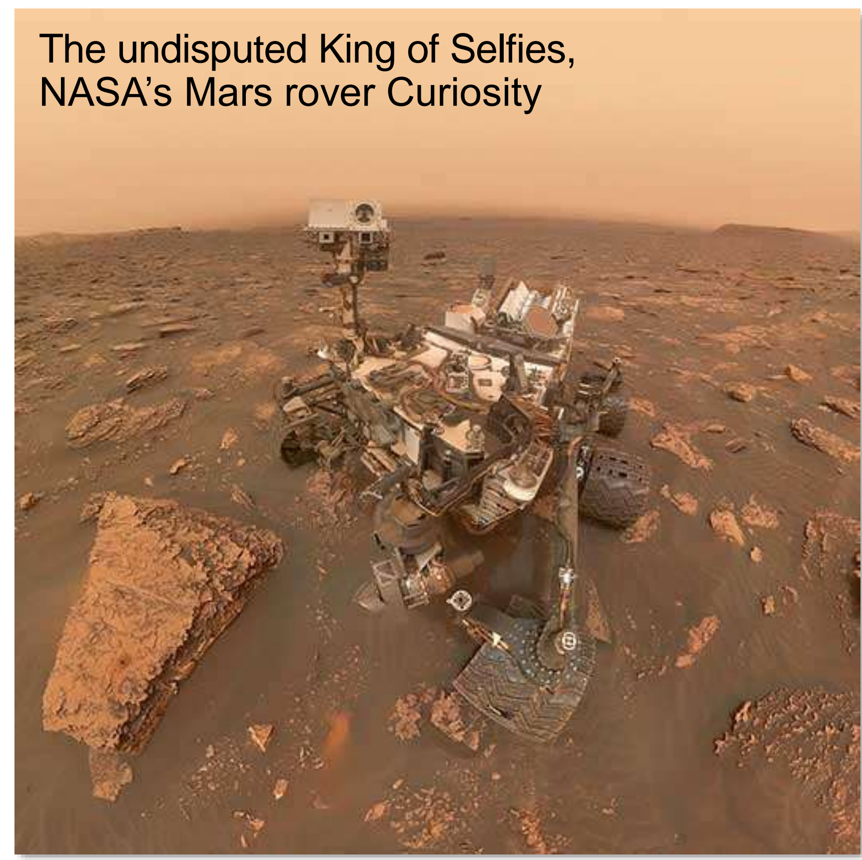
A Tiger Team tried to understand (2011)

The mission formed a Tiger Team to try to understand the reboots triggered by the use of the star tracker. Fortunately, the initial trajectory toward Mars was so close to predictions that the mission was able to delay the necessary maneuver by a month. Still, they were unable to solve the problem before the need for the maneuver pressed. To get the orientation and spin rate information that they needed, navigators employed a trick that had been developed during a similar circumstance on Pathfinder. They measured the minute Doppler shift of the spacecraft's radio signal, caused by the spacecraft's spin; the Doppler showed up as a sine wave in the radio frequency. From the magnitude of the Doppler signal, they determined the orientation of the spacecraft. With that knowledge, they were able to command the maneuver with sufficient precision. … It wasn't perfectly aligned, but it was close enough for later maneuvers to clean up any errors.

Mission status report
January 6, 2012*: The inertial measurement unit is used as an alternative to the spacecraft's onboard celestial navigation system due to an earlier computer reset. Diagnostic work continues in response to the reset triggered by use of star-identifying software on the spacecraft on Nov. 29. In tests at JPL, that behavior has been reproduced a few times out of thousands of test runs on a duplicate of the spacecraft's computer, but no resets were triggered during similar testing on another duplicate. The spacecraft itself has redundant main computers. While the spacecraft is operating on the "A side" computer, engineers are beginning test runs of the star-identifying software on the redundant "B side" computer to check whether it is susceptible to the same reset behavior.*

A previously unknown design idiosyncrasy (2011)

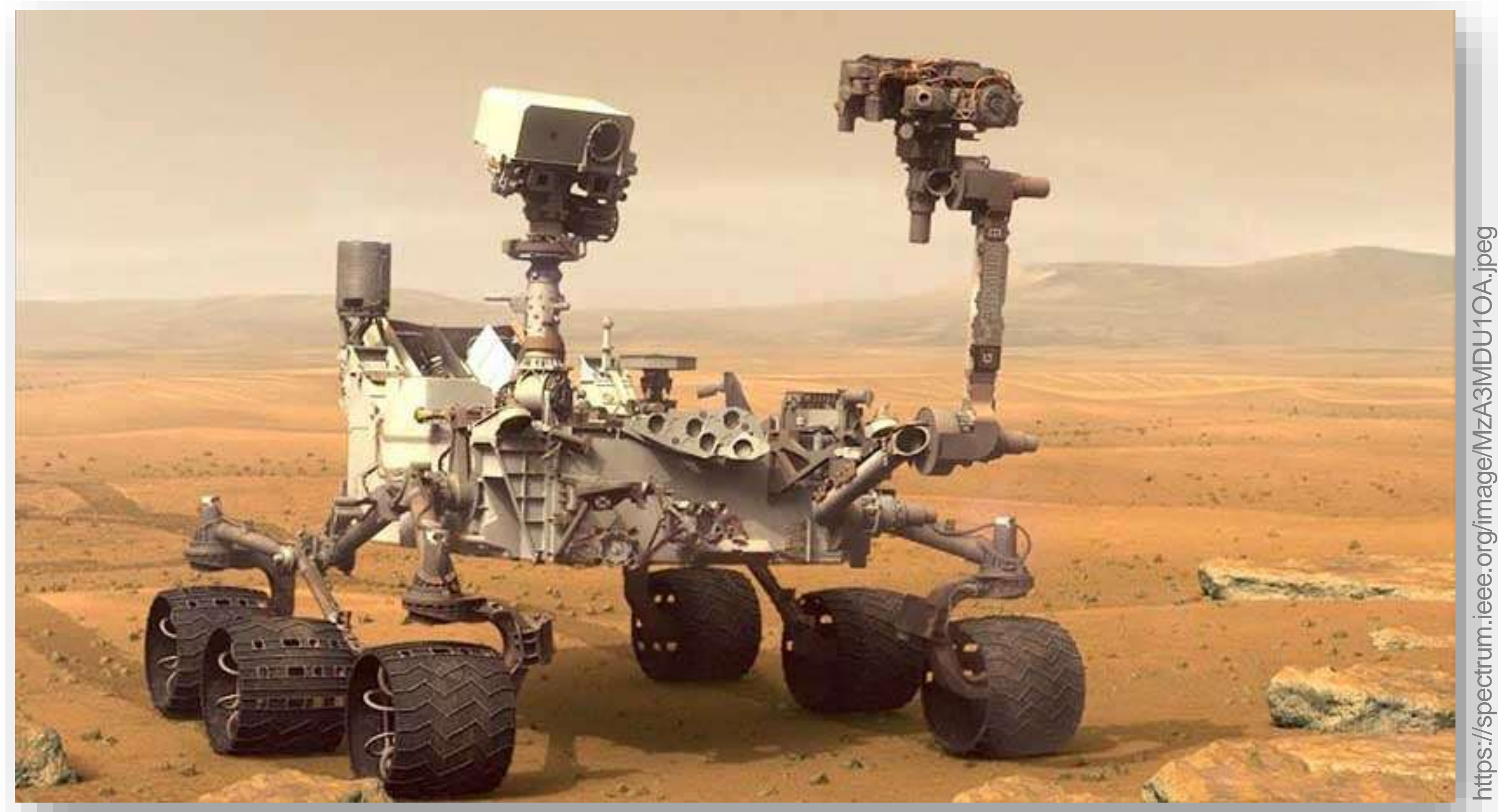
February 9, 2012: *Engineers have found the root cause of a computer reset that occurred two months ago on NASA's Mars Science Laboratory and have determined how to correct it. The fix involves changing how certain unused data-holding locations, called registers, are configured in the memory management of the type of computer chip used on the spacecraft. [...] The cause has been identified as a previously unknown design idiosyncrasy in the memory management unit of the Mars Science Laboratory computer processor. In rare sets of circumstances unique to how this mission uses the processor, cache access errors could occur, resulting in instructions not being executed properly. This is what happened on the spacecraft on Nov. 29. […] The spacecraft began normal use of its star tracker and true celestial navigation this week after its software update to avoid use of the memory functions that triggered the safe mode.*


The undisputed King of Selfies, NASA's Mars rover Curiosity

"The Curiosity Rover in the middle of a Martian dust storm. Much like most of the photography shared by the rover, it looks simultaneously real and otherworldly.[...] And for the nit-pickers among you, looking for the selfie stick and wondering, *how the hell is Curiosity even taking this photo?*"
www.cnet.com/news/the-curiosity-rover-took-this-selfie-in-the-middle-of-a-dust-storm-on-mars/

Curiosity lands safely after epic descent (2011)

***6 Aug 2012, 2:42 GMT:** SUCCESS: YES, according the data we have so far, the Mars Science Laboratory Curiosity has successfully landed at Gale Crater on Mars! The mood here at JPL is absolutely insane! People are going nuts. Oh, and they're also handing out Mars bars. As soon as we're done being nuts and scarfing candy, we'll get settled in for the press conference, which should be full of good news!*



https://spectrum.ieee.org/image/MzA3MDU1OA.jpeg

Andere Softwareprobleme im Weltraum: 2016

Schiaparelli – hey, wir sind ja schon unter der Marsoberfläche!

www.ard-text.de/index.php?page=552 (24.11.2016)

```
"Schiaparelli" hatte Softwarefehler

Der Absturz der Mars-Sonde "Schiaparel-
li" auf dem Roten Planeten vor gut ei-
nem Monat wurde laut Europäischer Welt-
raumagentur (ESA) durch einen Software-
Fehler beim Navigationsprogramm verur-
sacht. Wie der ESA-Verantwortliche
Thierry Blancquaert der Nachrichten-
agentur AFP mitteilte, registrierte die
Sonde aufgrund einer Reihe falscher
Messungen eine negative Höhe, als sie
noch mehrere Kilometer über der Ober-
fläche war.

Wegen dieser falschen Berechnungen war
die Sonde schließlich mit 540 Stunden-
kilometer auf dem Mars aufgeschlagen.
```

Leserkommentare bei www.heise.de:

> In der Behandlung der Ausnahmesituation [...] daß *sinnvolle* Fallbacks definiert werden. Auf keinen Fall Unsinn à la 'Hey, wir sind ja schon unter der Marsoberfläche, höchste Zeit für die Bremsfallschirme'.

> Das erinnert mich schon arg an eine Kurzgeschichte aus dem Buch *Pilot Pirx* von Stanislaw Lem. Die Raumschiffsteuerung ist während der eingeleiteten Landung auf dem Planeten Mars (sic) durch die Annäherung an diesen so hoffnungslos überfordert (übrigens auch durch die Sensorik und deren Informationslast), dass das basale System das Problem als Kometenannäherung interpretiert und, in dem Fall viel zu spät, ein Ausweichmanöver initiiert.

...bewegte sich „Schiaparelli" offenbar beim Weg durch die immer dichter werdende Marsatmosphäre stärker als vorhergesehen hin und her. Das wiederum sorgte dafür, dass ein Messgerät, das die Eigendrehung der Sonde überwachte, die sogenannte IMU, überlastet wurde. Statt wie vom Hersteller versprochen schon 15 Millisekunden nach einer Überlastung wieder betriebsbereit zu sein, blockierte das Problem den Bordcomputer für rund eine Sekunde. Dadurch wiederum wurde aus den Bewegungsdaten eine falsche Höhenangabe berechnet. [Der Spiegel 24.05.2017]

Andere Softwareprobleme im Weltraum: 2019

Beresheet moon lander: unexpected computer resets

The Beresheet moon lander built by the Israeli startup SpaceIL missed a planned maneuver to steer the spacecraft along its eight-week journey to the lunar surface. The maneuver was originally scheduled for 12 a.m. local time in Israel on Tuesday (Feb. 26, 2019) as Beresheet orbited the Earth out of communications range with its mission control center.

During the pre-maneuver phase the spacecraft computer reset unexpectedly, causing the maneuver to be automatically cancelled. According to Doron, the official from IAI, the glitch was software-related. “The problems were small glitches that were solved by the commands in the software,” he said. “Nothing that’s very serious; it just takes time to iron them out.”

SpaceIL’s Beresheet lander launched toward the moon Thursday (Feb. 21) aboard a SpaceX Falcon 9 rocket. The spacecraft is the first privately developed lander launched to the moon. The $100 million lander Beresheet (its name means “in the beginning” in Hebrew) is a joint effort between the nonprofit organization SpaceIL and Israel Aerospace Industries (IAI).

April 11, 2019: “We had a failure of the spacecraft,” said Opher Doron, “We unfortunately have not managed to land successfully.” A chain of events caused by a command sent from the SpaceIL control room turned off the spacecraft’s main engine and prevented proper engine activation, making a crash landing on the moon inevitable.

Ende der historischen Notiz

www.space.com/israel-moon-lander-suffers-glitch.html

Viel zu viel Milch in der WG

Zeit	Person B	Person C	Person A
7:00	Schläft	Schläft	**Keine Milch im Kühlschrank!**
7:15	Wacht auf	Schläft	Zur Vorlesung aufbrechen
7:30	**Keine Milch!**	Schläft	Unterwegs
7:45	Einkaufen gehen	Wacht auf	Ankunft ETH
8:00	Ankunft Laden	**Keine Milch!**	Zum Hörsaal gehen
8:15	**Milch kaufen**	Einkaufen gehen	Vorlesung
8:30	Heimgehen	Ankunft Laden	Vorlesung (gähn)
8:45		**Milch kaufen**	Einkaufen gehen
9:00		Heimgehen	Ankunft Laden
9:15			**Milch kaufen**
9:45			Heimgehen

Zwischenzeit

Viel zu viel Milch in der WG

Würde nicht passieren, wenn A ganz schnell wieder zurück wäre → Ergebnis daher abhängig von der relativen Geschwindigkeit → „race condition"

Zeit	Person B	Person C	Person A
7:00	Schläft	Schläft	Keine Milch im Kühlschrank! Zur Vorlesung aufbrechen Unterwegs In Hörsaal gehen Vorlesung Einkaufen gehen Milch kaufen Heimgehen
7:15	Wacht auf	Schläft	
7:30	Keine Milch!	Schläft	
7:45	Einkaufen gehen	Wacht auf	
8:00	Ankunft Laden	Keine Milch!	
8:15	Milch kaufen	Einkaufen gehen	
8:30	Heimgehen	Ankunft Laden	
8:45		Milch kaufen	
9:00		Heimgehen	
9:15			
9:45			

Don't panic!
Ich bringe auf dem Rückweg von der Vorlesung Milch mit!

B asks C, "Could you please go shopping and buy one bottle of milk, and if they have eggs, get 3!" A short time later C comes back with 3 bottles of milk. B asks him, "Why the hell did you buy three bottles of milk?" "They had eggs", C replies.

Viel zu viel Milch in der WG

Und wenn alle gleich-zeitig aufwachen?

Zeit	Person B	Person C	Person A
7:00	Keine Milch!	Keine Milch!	Keine Milch im Kühlschrank!
7:15	...	...	Zur Vorlesung aufbrechen
7:30	...	...	Unterwegs
7:45	Einkaufen gehen	...	Ankunft ETH
8:00	Ankunft Laden	...	Zum Hörsaal gehen
8:15	Milch kaufen	Einkaufen gehen	Vorlesung
8:30	Heimgehen	Ankunft Laden	Vorlesung (gähn)
8:45		Milch kaufen	Einkaufen gehen
9:00		Heimgehen	Ankunft Laden
9:15			Milch kaufen
9:45			Heimgehen

Don't panic! Ich bringe auf dem Rückweg von der Vorlesung Milch mit!

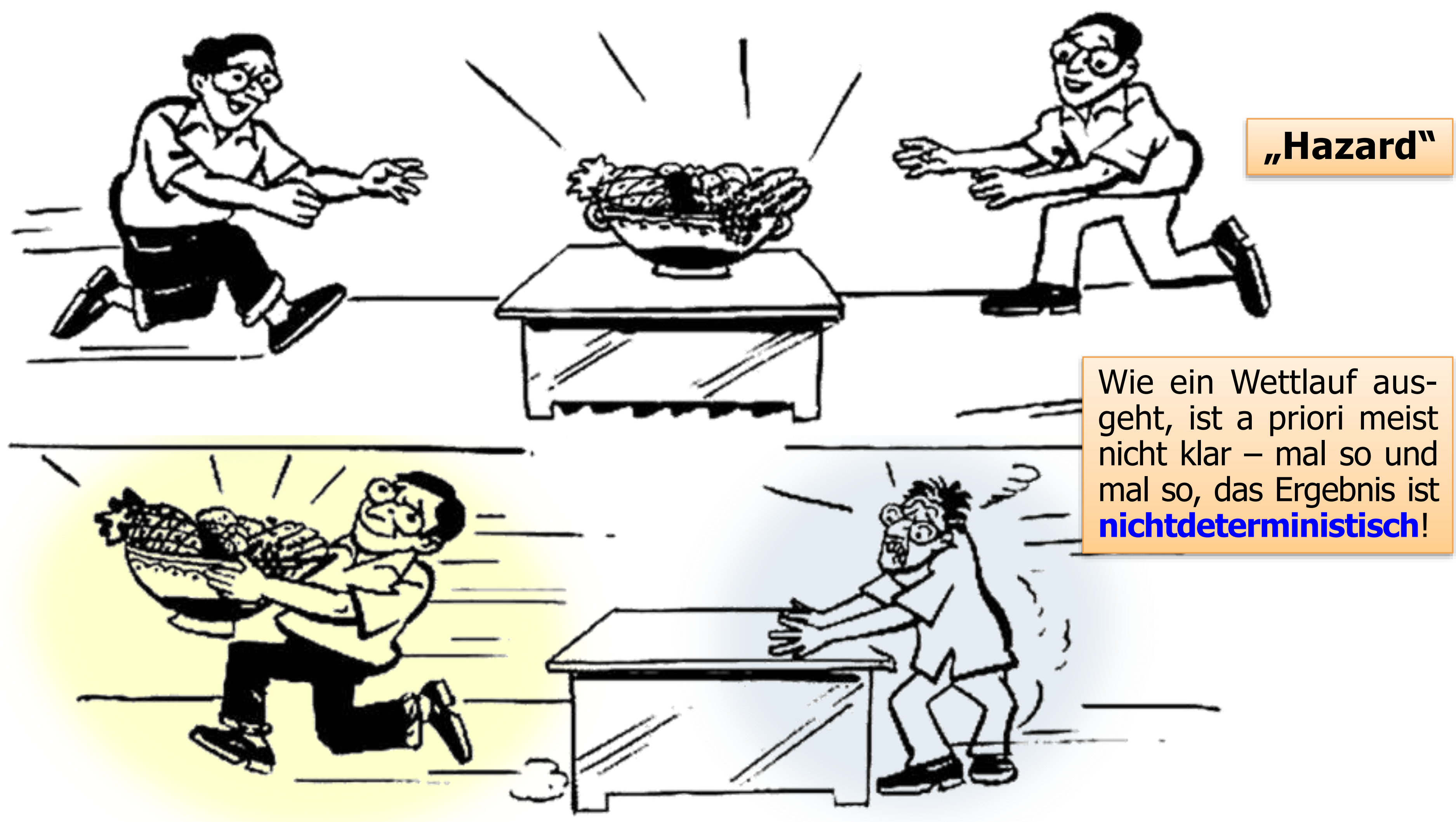
Evtl. bringen dann alle einen Zettel exakt gleichzeitig an...

...und gehen Milch kaufen!

Muss man also auch den Zugriff auf einen „Koordinationsmechanismus" koordinieren?

Race condition [Wettlaufsituation]

Konstellation, bei der das Ergebnis abhängt vom zeitlichen Ablauf nicht kontrollierbarer Ereignisse (z.B. in anderem Thread)



„Hazard“

Wie ein Wettlauf ausgeht, ist a priori meist nicht klar – mal so und mal so, das Ergebnis ist **nichtdeterministisch**!

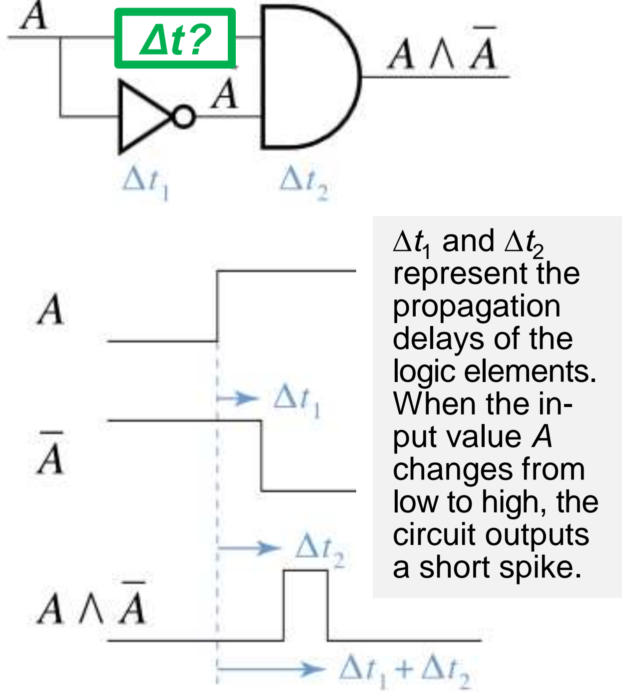
The winner takes it all – the loser’s standing small … beside the victory

Hazard / race condition bei Logikschaltungen

https://en.wikipedia.org/wiki/Race_condition

Eine Wettlaufsituationen analog zur Situation bei Threads (Nichtdeterminismus aufgrund zeitlicher Unbestimmtheit paralleler Aktivitäten) kann auch im Hardwarebereich auftreten:

- A race hazard is a circuit transient, which under certain changes of an input signal in a circuit with certain delays, can occur at the output of a gate.
- A typical example of a race condition may occur when a logic gate combines signals that have traveled along different paths from the same source. The inputs to the gate can change at slightly different times in response to a change in the source signal. The output may, for a brief period, change to an unwanted state before settling back to the designed state. Certain systems can tolerate such glitches but…

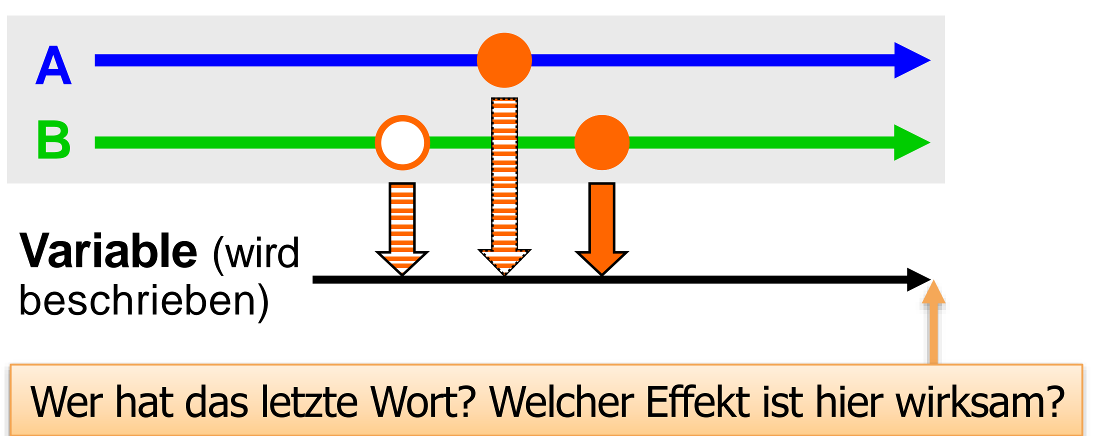


Δt_1 and Δt_2 represent the propagation delays of the logic elements. When the input value A changes from low to high, the circuit outputs a short spike.

Consider, for example, a two-input AND gate fed with a logic signal A on one input and its negation, NOT A, on another input. In theory the output (A AND NOT A) should never be true. If, however, changes in the value of A take longer to propagate to the second input than the first when A changes from false to true then a brief period will ensue during which both inputs are true, and so the gate's output will also be true.

Race conditions bei parallelen Threads

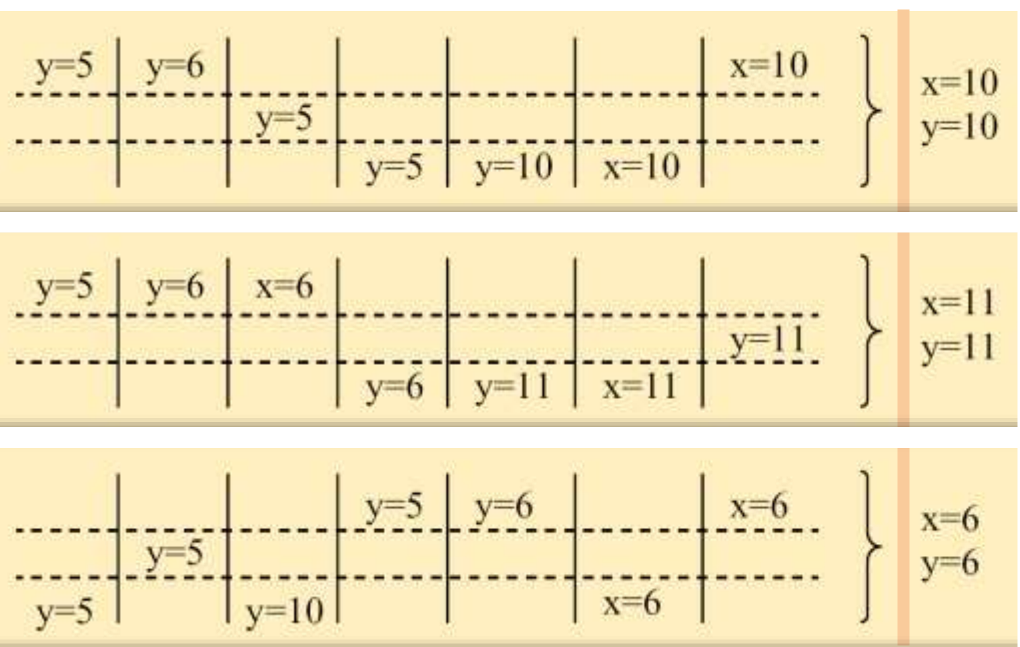
- Threads nutzen oft gemeinsame Variablen („shared variables")
  - Zum Beispiel, um Daten untereinander auszutauschen
  - Bei Java: klassenbezogene Variablen (d.h. mit „static" deklariert)
- Unterschiedliche Ergebnisse, je nachdem, in welcher Reihenfolge die Threads auf eine Variable zugreifen → race condition



Unbeabsichtigte Wettlaufsituationen sind ein häufiger Grund für schwer auffindbare Programmfehler; bezeichnend für solche Situationen ist nämlich, dass bereits die veränderten Bedingungen zum Programmtest zu einem völligen Verschwinden der Symptome führen können.

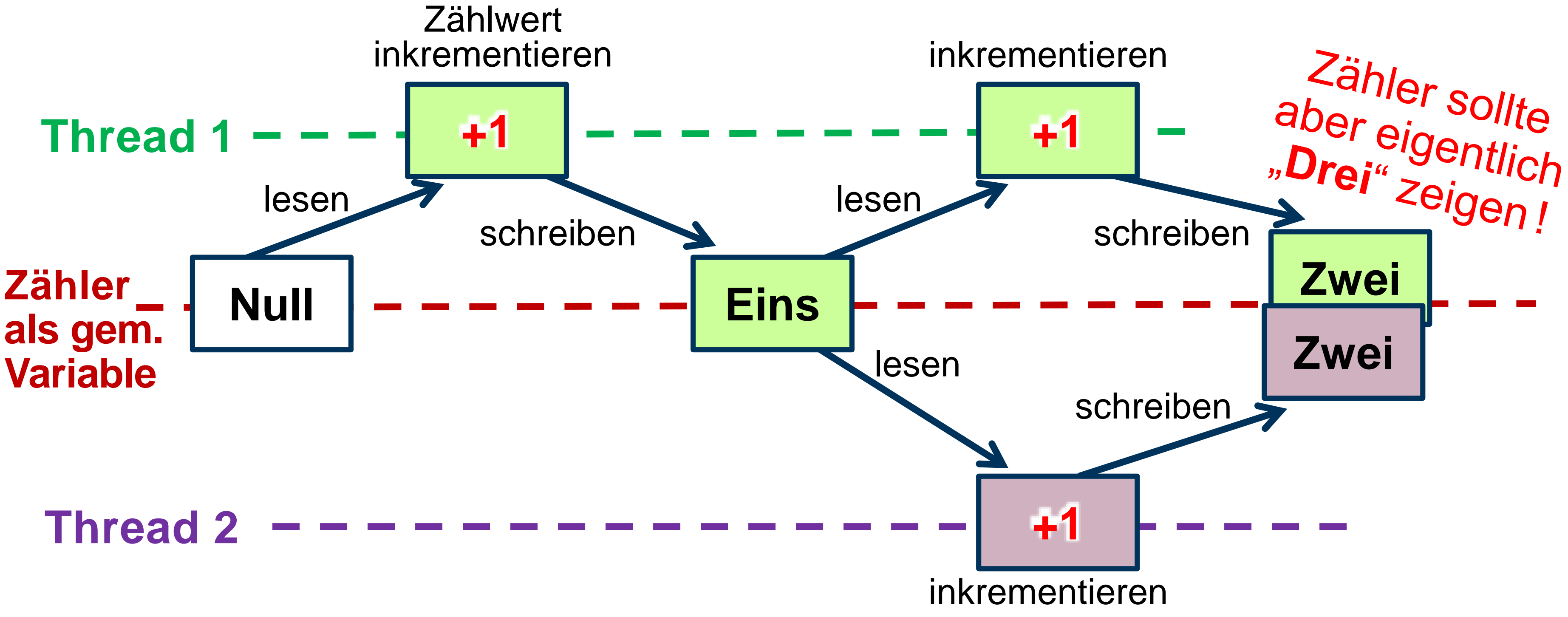
Ein Beispiel für race conditions

```
int x = 5;  // initial

Thread 1:
{ y = x; y = y+1; x = y; }

Thread 2:
{ y = x; }

Thread 3:
{ y = x; y = y+5; x = y; }
```



- Welches Ergebnis ist das richtige?
  - Nichtdeterministischer Ablauf → es gibt mehrere „gleich richtige" (wie viele?)
  - Oder soll nur das richtig sein, was der Programmierer „eigentlich" gemeint hat??
- Wovon hängt es in der Praxis ab, welches Ergebnis (von mehreren möglichen Ergebnissen) bei einem nichtdeterministischen Ablauf „erscheint"?

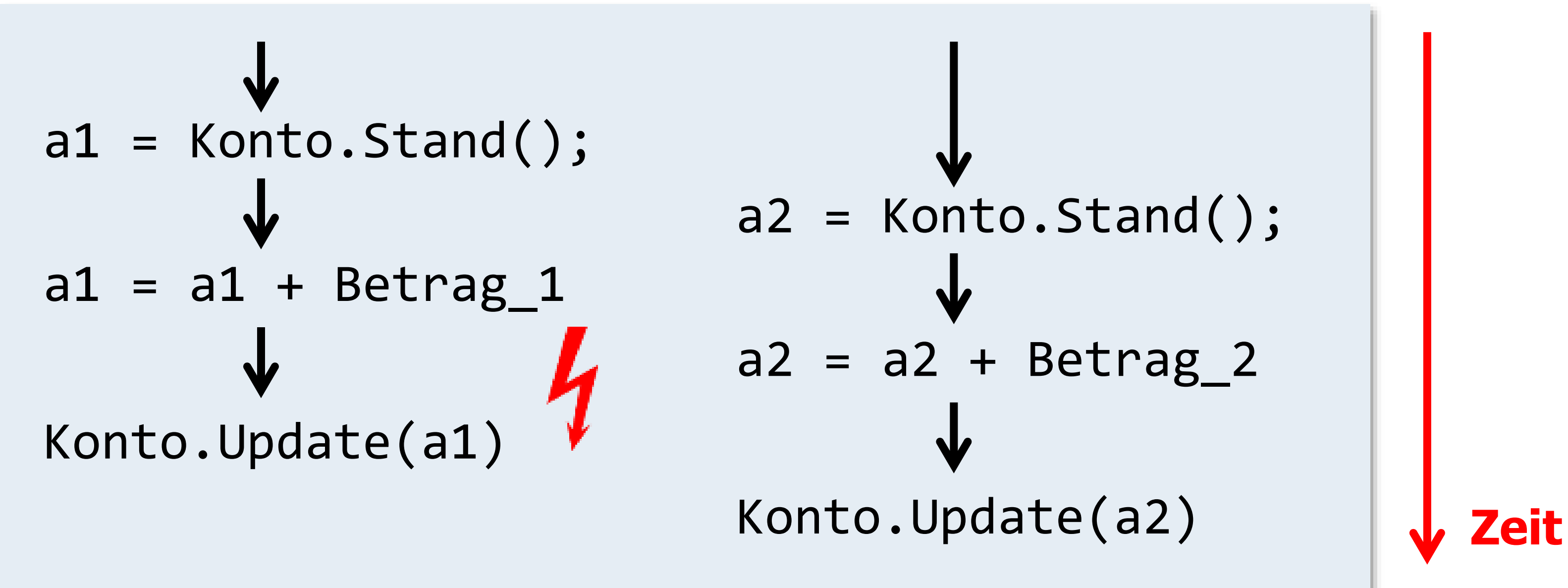
Race condition beim parallelen Zählen



- Hier gibt es nicht mehrere „gleich richtige Ergebnisse“, sondern dieses Resultat ist schlichtweg falsch!
  - Jedenfalls hinsichtlich der Erwartung, dass jedes Zählereignis auch „zählt“
- „Lost update problem“

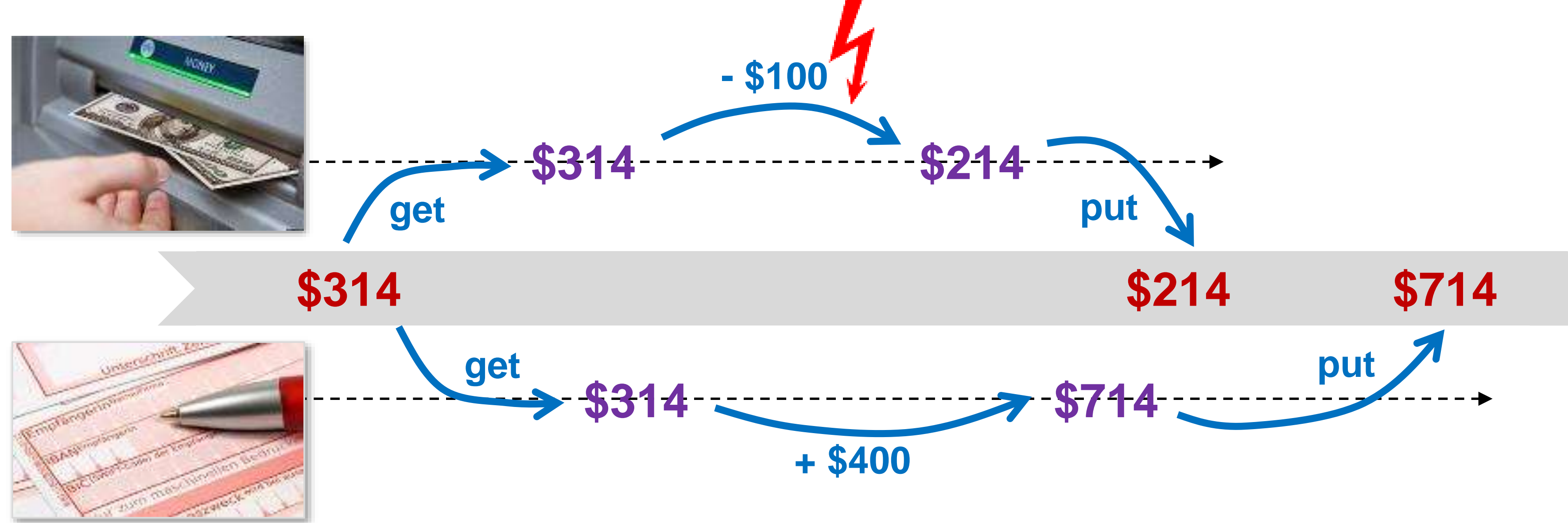
Race condition: „lost update problem“

- Bsp.: Zwei parallele Threads nehmen Kontobuchung vor:



Race condition: „lost update problem“ (2)

- Bsp.: Zwei parallele Threads nehmen Kontobuchung vor:



- Kontobuchung des oberen Threads geht verloren! Lösung?
  - Unterer Thread gewinnt („wer zuletzt lacht, lacht am besten“)
  - Es ist aber a priori unbestimmt bzw. Zufall, wer der lachende Letzte ist

Race condition: „lost update problem“ (3)

- Wie wäre es, alles in ein einzige Statement zu packen?

```
...
Konto.Update(Betrag1+
        Konto.Stand());
...
```

```
...
Konto.Update(Betrag2+
        Konto.Stand());
...
```

- Oder vielleicht jeweils Aufruf einer Methode „Buchung“?

```
...
Buchung(k, 300.00);
...
```

```
...
Buchung(k, -154.23);
...
```

```
void Buchung (... Konto, float Betrag) {
  float a = Konto.Stand();
  Konto.Update(a + Betrag);
}
```

Ist das eigentlich kommutativ?

Race condition verursacht Blackout

Race condition verursacht Blackout

- August 2003: Grossflächiger Stromausfall im Nordosten der USA und Teilen Kanadas
  - Betraf ca. 55 Millionen Leute für viele Stunden
  - New York, Pittsburgh, Detroit, Toronto, Ottawa,...
- Eine Hochspannungsleitung berührte einen Baum und fiel wegen des resultierenden Kurzschlusses aus
- Dies wäre durch Rekonfiguration beherrschbar gewesen; der Stromversorger „FirstEnergy" bemerkte dies aber nicht, genauso wenig wie die resultierende Netzüberlastung, die den Ausfall drei weiterer Stromleitungen innerhalb von 40 Minuten nach sich zog
- Grund dafür war, dass das Überwachungssystem auf heimtückische Weise versagte: Es zeigte im computerisierten Kontrollraum einen vergangen, „guten" Systemzustand an, weil die Alarmereignisse nicht durchkamen – wegen Blockade aufgrund einer Endlosschleife, die durch eine race condition verursacht wurde

A Race Condition…
…in an Energy Management System

A race condition in GE Energy's Unix-based XA/21 energy management system stalled FirstEnergy's control room alarm system for over an hour. System operators were unaware of the malfunction; the failure deprived them of both audio and visual alerts for important changes in system state.

[http://en.wikipedia.org/wiki/Northeast_blackout_of_2003]

The glitch kept FirstEnergy's control room operators "in the dark" while three of the company's high voltage lines sagged into unkempt trees and "tripped" off. Because the computerized alarm failed silently, control room operators didn't know they were relying on outdated information; trusting their systems, they even discounted phone calls warning them about worsening conditions on their grid.

[Tracking the blackout bug, Kevin Poulsen, SecurityFocus, 2004-04-07, www.securityfocus.com/news/8412]

A Race Condition… (2)

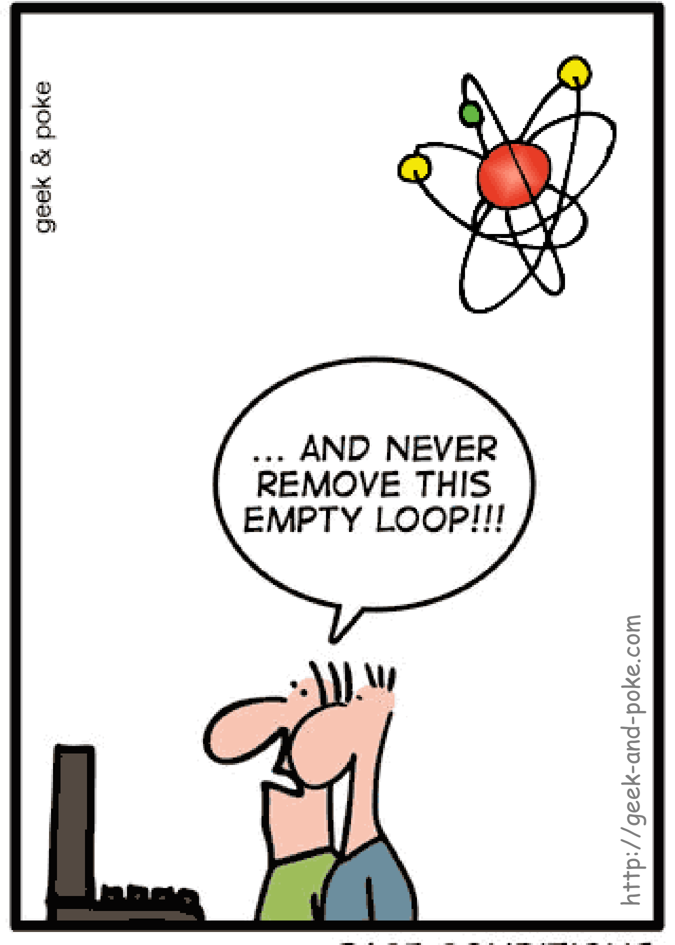
Calls were pouring in from industrial customers, neighboring utilities, and FirstEnergy’s own power plant operators, who were all trying to interpret signs of trouble on the grid. “I’m still getting a lot of voltage spikes and swings on the generator,” said an operator at the Perry nuclear power plant in Ohio, who was worried about his unit shutting down automatically. “I don’t know how much longer we’re going to survive.” Only after the lights went out in the FirstEnergy control room did operators know for sure that it was their system and not somebody else’s that was about to collapse. By then it was too late.

[Peter Miller: The Smart Swarm – How to Work Efficiently, Communicate Effectively, and Make Better Decisions Using the Secrets of Flocks, Schools, and Colonies, Penguin, 2010]

Manhattan ohne Subway

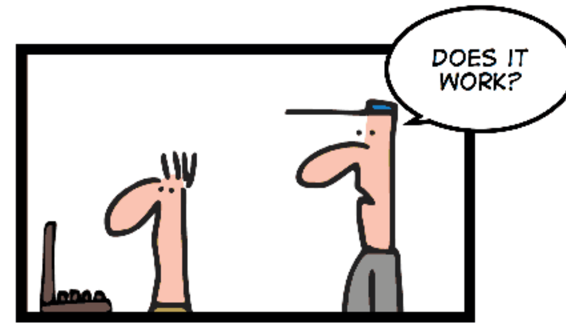
A Race Condition… (3)

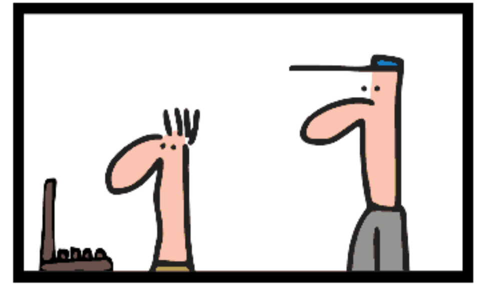
A half-a-dozen workers at GE Energy began working feverishly with the utility […] to figure out what went wrong. […] Sometimes working late into the night and the early hours of the morning, the team pored over the approximately one-million lines of code that comprise the XA/21's Alarm and Event Processing Routine, written in the C and C++ programming languages. Eventually they were able to reproduce the alarm crash in GE Energy's laboratory, says Unum. "It took us a considerable amount of time to go in and reconstruct the events." In the end, they had to slow down the system, injecting deliberate delays in the code while feeding alarm inputs to the program.

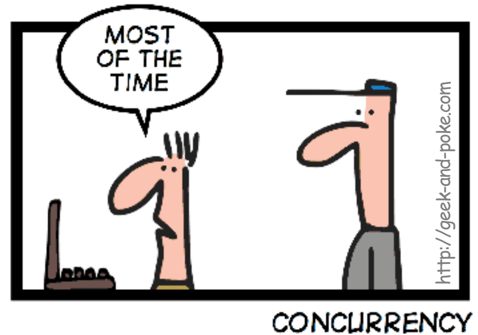


RACE CONDITIONS

A Race Condition... (4)







CONCURRENCY

About eight weeks after the blackout, the bug was unmasked as a particularly subtle incarnation of a common programming error called a "race condition," triggered by a perfect storm of events and alarm conditions on the equipment being monitored. The bug had a window of opportunity measured in milli-seconds. "There was a couple of processes that were in contention for a common data structure, and through a software coding error in one of the appli-cation processes, they were both able to get write access to a data structure at the same time," says Unum. "And that corruption led to the alarm event application getting into an infinite loop and spinning."

Last fall the company gave its customers a patch against the bug, along with installation instructions and a utility to repair any alarm log data corrupted by the glitch. According to Unum, the company sent the package to every customer – more than 100 utilities around the world.

A Race Condition… (5)

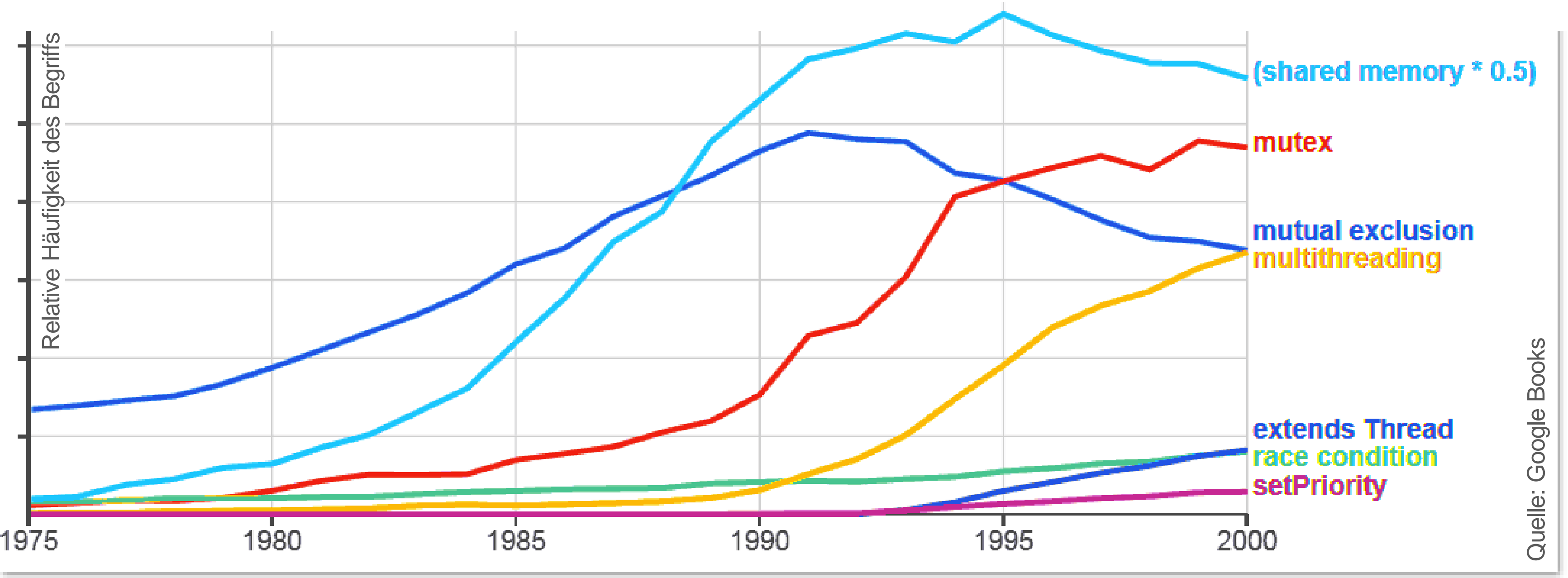
"I'm not sure that more testing would have revealed that," says Unum. Unfortunately, that's kind of the nature of software... you may never find the problem. I don't think that's unique to control systems or any particular vendor software." Tom Kropp, manager of the enterprise information security program at the Electric Power Research Institute, an industry think tank, agrees. He says faulty software may always be a part of the electric grid's DNA. "Code is so complex, that there are always going to be some things that, no matter how hard you test, you're not going to catch," he says.

But Peter Neumann, principal scientist at SRI International and moderator of the Risks Digest, says that the root problem is that makers of critical systems aren't availing themselves of a large body of academic research into how to make software bulletproof.

[Tracking the blackout bug, Kevin Poulsen, SecurityFocus, 2004-04-07, www.securityfocus.com/news/8412]

Program testing can at best show the presence of errors, but never their absence. -- E.W. Dijkstra

Evolution des parallelen Programmierens



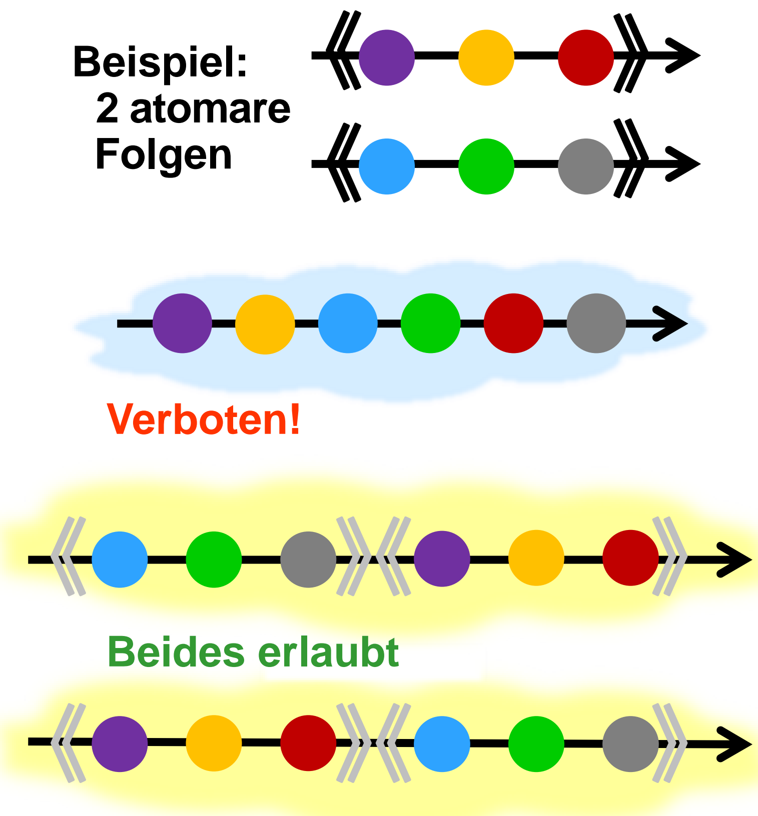
- Der wechselseitige Ausschluss war schon früh bei klassischen Betriebssystemen für singuläre Prozessoren im Rahmen der Prozessverwaltung ein Problem
- Mehrprozessorsysteme mit gemeinsamem Speicher („shared memory") akzentuierten das Problem, insbesondere für parallele Anwendungen
- Mit „multithreading" wurden Parallelisierungskonzepte auf die Anwendungs-ebene gehoben und Applikationsprogrammierern verfügbar gemacht
- Bei „extends Thread" und „set Priority" handelt es sich um Java-Konstrukte
- Probleme wie „race conditions" wurden im Laufe der Zeit zunehmen relevanter

Atomarität

- Ist eine Java-Anweisung „atomar“?
  - Z.B.: Konto.Update(Betrag + Konto.Stand());
  - Oder zumindest: Buchung(k, …);

Auch dann gäbe es aber zwei mögliche Abläufe: Konto würde bei „unglücklicher“ Reihenfolge evtl. kurzzeitig überzogen!

- Atomare Folge von Operationen – das wünschen wir uns:
  - Während die Folge ausgeführt wird, werden keine anderen Operationen (quasi) gleichzeitig ausgeführt
  - Wenn die (einzige?) CPU mit der ersten Operation der atomaren Folge beginnt, arbeitet sie diese bis zur letzten ab, ohne zwischendrin etwas anderes („störendes“) zu tun
  - Unterbrechungen sind höchstens zwischen atomaren Folgen erlaubt



Relative Atomarität

- Aber: „unkritische Dinge" könnten eigentlich doch parallel zu einer atomaren Folge ausgeführt werden
  - Quasi „heimlich", aus Optimierungsgründen?

- Und was genau ist unkritisch?
  - Teile eines ganz anderen Programms?
  - Auf den Kontostand nur lesend zugreifen?
  - Leere Schnittmenge bzgl. gemeinsamer Variablen?

Problematisch sind sogen. „Shared-Memory-Systeme": Mehrere Prozessoren oder „Kerne", aber ein gemeinsamer Hauptspeicher

➔ „Atomar" ist ein interpretationsbedürftiger relativer Begriff!
  - → Formalisierung notwendig, um sich darauf verlassen zu können
  - Informell: Für andere sieht es so aus, als ob die atomare Anweisungsfolge instantan ausgeführt würde; niemand kann „dazwischenfunken"

Inkonsistenzen

- Durch die Nicht-Atomarität von Anweisungsfolgen kann es bei paralleler Ausführung evtl. zu unerwünschten Effekten kommen
  - Z.B. inkonsistente Zustände, die widersprüchlich sind und nicht auftreten dürften

Inkonsistenz: Ein Zustand, in dem mehrere Dinge, die alle als gültig angesehen werden sollen, nicht miteinander vereinbar sind.

Inkonsistenzen

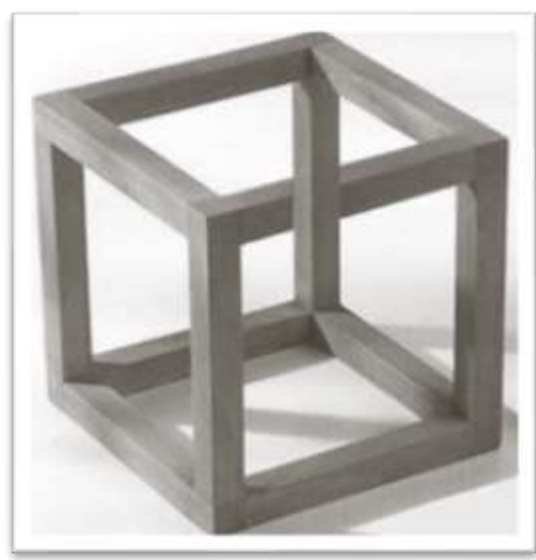
Ein „Tribar"
(Oscar Reutersvärd / Roger Penrose)

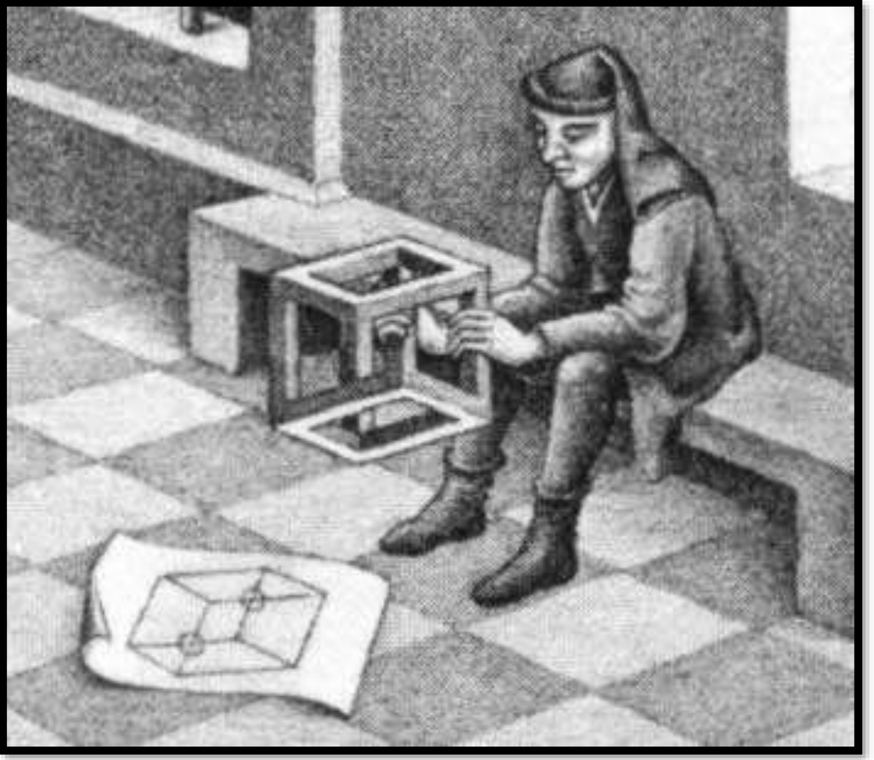
Inkonsistenz: Ein Zustand, in dem mehrere Dinge, die alle als gültig angesehen werden sollen, nicht miteinander vereinbar sind.

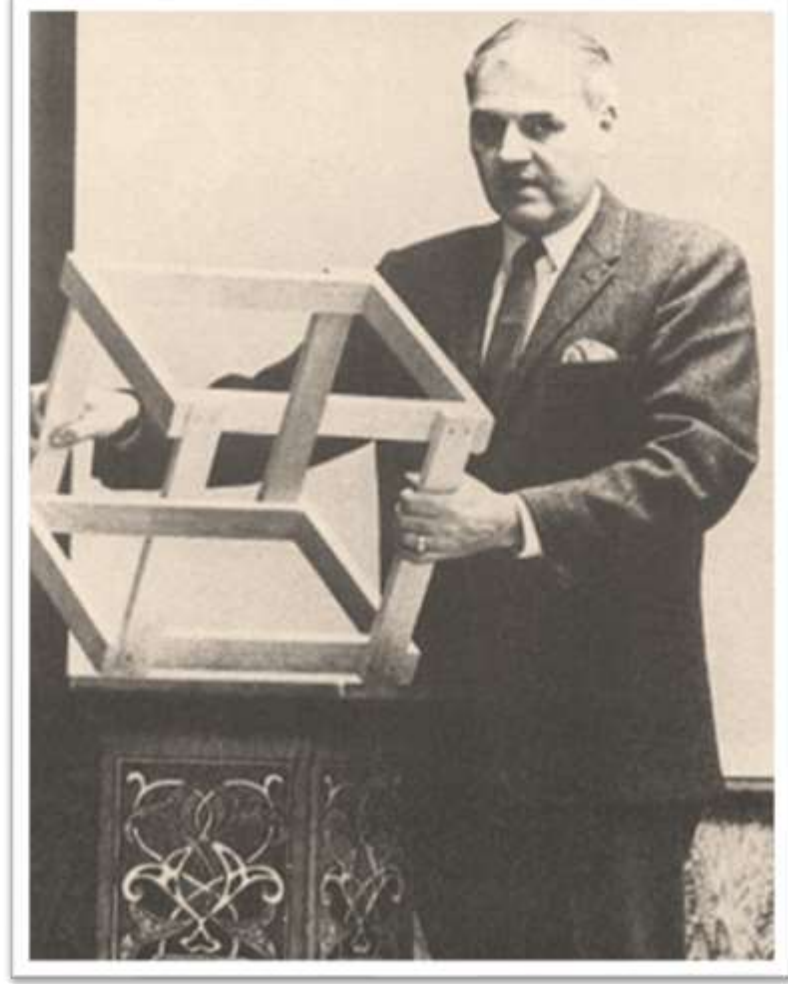
Inkonsistenzen

https://nosenosocurrio.files.wordpress.com/2008/09/dice1.jpg

www.roz.at/rozweb/images/Penrose_A_e_swp_q.JPG

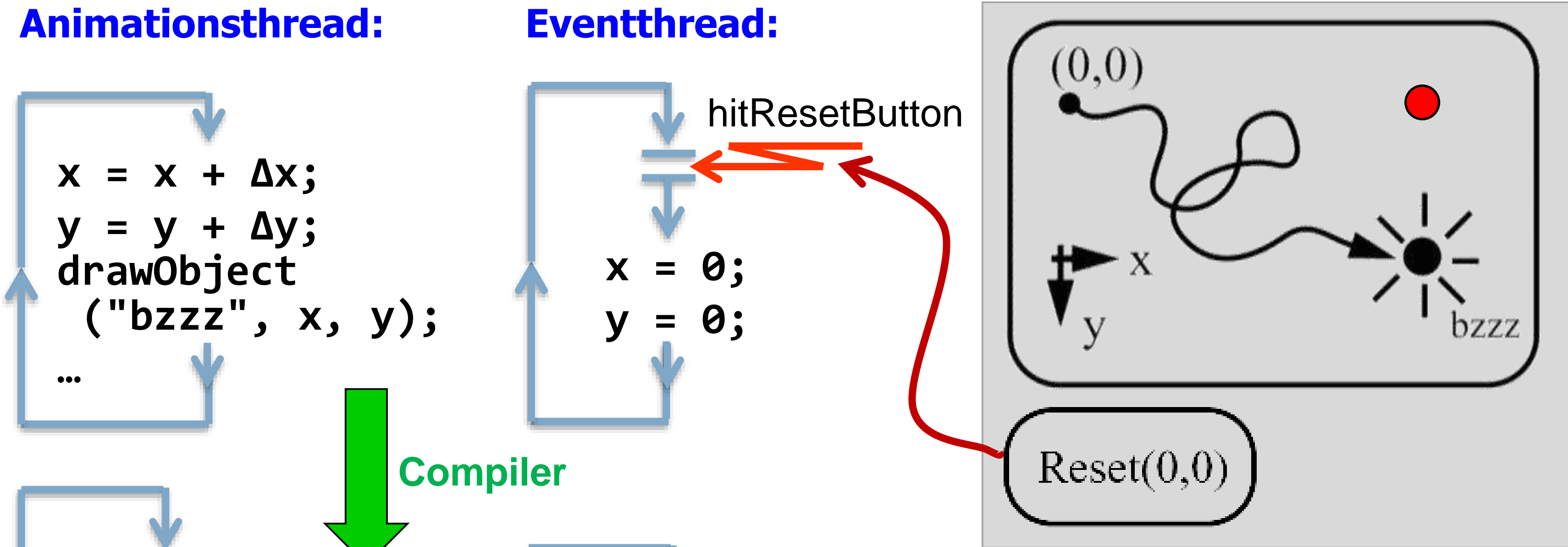






Inkonsistenzen: Beispiel
(Animation einer Teilchenbewegung)

Objekt springt bei „reset“ (gelegentlich!) an eine falsche Stelle (x-Koordinate ist dann nicht Null)



read(x)
…
write(x,…)
read(y)
…
write(y,…)
drawObject

write(x,0)
write(y,0)

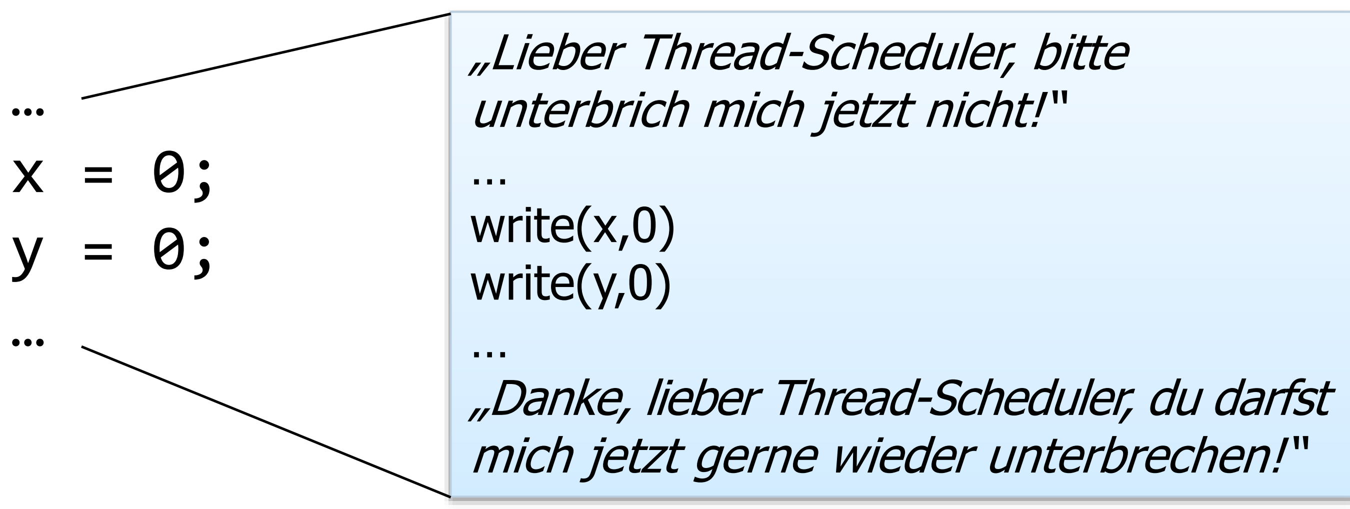
Das „window of opportunity“ für den lost update ist sehr klein (Eventthread zwischen den beiden „write“ unterbrechen und eine einzige Aktion aus dem Animationsthread ausführen), aber es existiert!

Hier stellen „read“ und „write“ Maschineninstruktionen dar (Datenaustausch zwischen CPU-Register und Hauptspeicher)

Denkübung: Kann auch der duale Fall auftreten mit x = 0, aber y-Koordinate nicht 0?

Lieber Thread-Scheduler!

Wie kann man Atomarität von Anweisungsfolgen erreichen?



Unterbrechungssperren auf Java-Ebene mittels Prioritäten?

```
…
int p = getPriority();
setPriority(Thread.MAX_PRIORITY);
x = 0; y = 0; // kritischer Abschnitt
setPriority(p);
```

Ist das eine funktionieren-de und gute Lösung für Unterbrechungssperren?

- Beachte: Auch bei einer „Unterbrechungssperre" können durchaus andere Dinge parallel ablaufen
  - Der Systemprozess, wo der Thread-Scheduler eingebettet ist (z.B. die Java-VM), kann vom Betriebssystem zeitweise suspendiert werden
  - Ein Mehrkernprozessor könnte andere Threads „echt" parallel ausführen
  - Die „Umwelt" ändert sich auch während einer Unterbrechungssperre
- Es kommt also darauf an, in welcher „Hinsicht" Atomarität gewährleistet werden soll / kann!

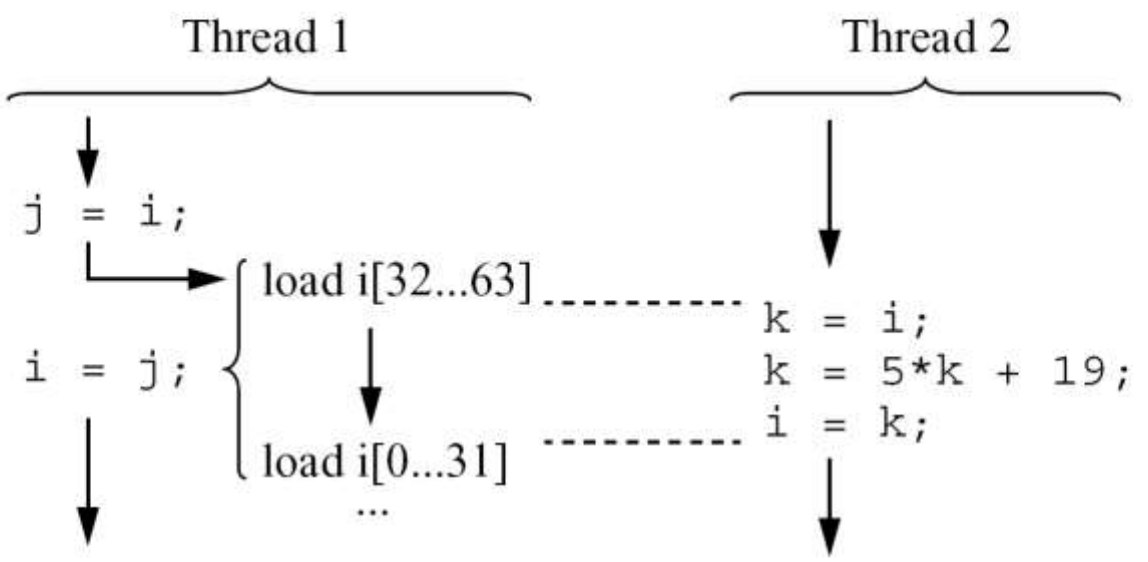
Java: Nicht-Atomarität von double und long

- Double- und long-Variablen sind 64 Bit lang
  - Float und int benötigen nur 32 Bit

- 32-Bit-Prozessoren benötigen zum Schreiben / Lesen dafür je zwei Speicherzugriffe; diese sind zwischendrin unterbrechbar!
  - Unglaublich, aber leider wahr!

- Bei Threads, die auf die gleiche 64-Bit-Variable zugreifen, kann es daher zu Inkonsistenzen kommen: Ergebnis entspricht dann meist nicht einem möglichen Interleaving auf Sprachebene
  - Ein in der Praxis jahrelang funktionierendes Programm könnte also nach „harmloser" Typänderung *int → long* einen latenten Fehler aktivieren bzw. eine race condition begründen!

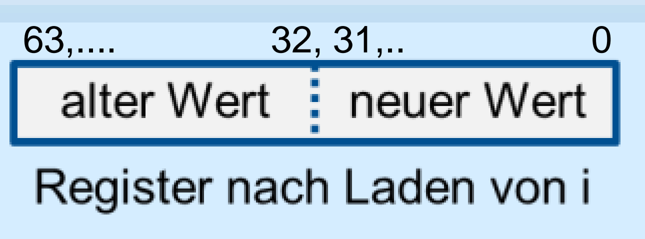
Java: Nicht-Atomarität von double und long (2)



"Programmers are cautioned always to explicitly synchronize access to shared double or long variables"

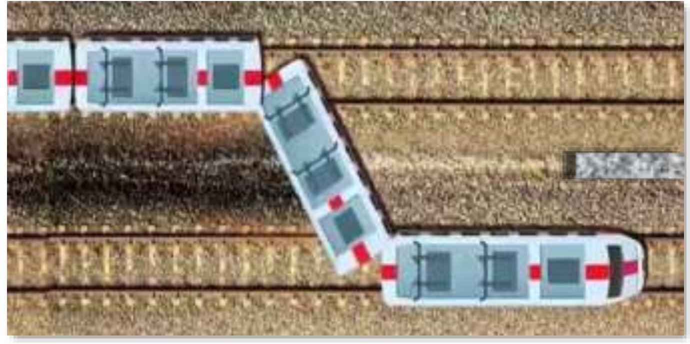
(The Java Virtual Machine Specification)

Danach steht in i ein „unmöglicher" Wert!



Wie erreicht man Konsistenz, wenn es hardwaremässig keine atomaren Befehle zum Zugriff auf 64 Bits gibt?

Nicht-Atomarität führt zu Inkonsistenzen...



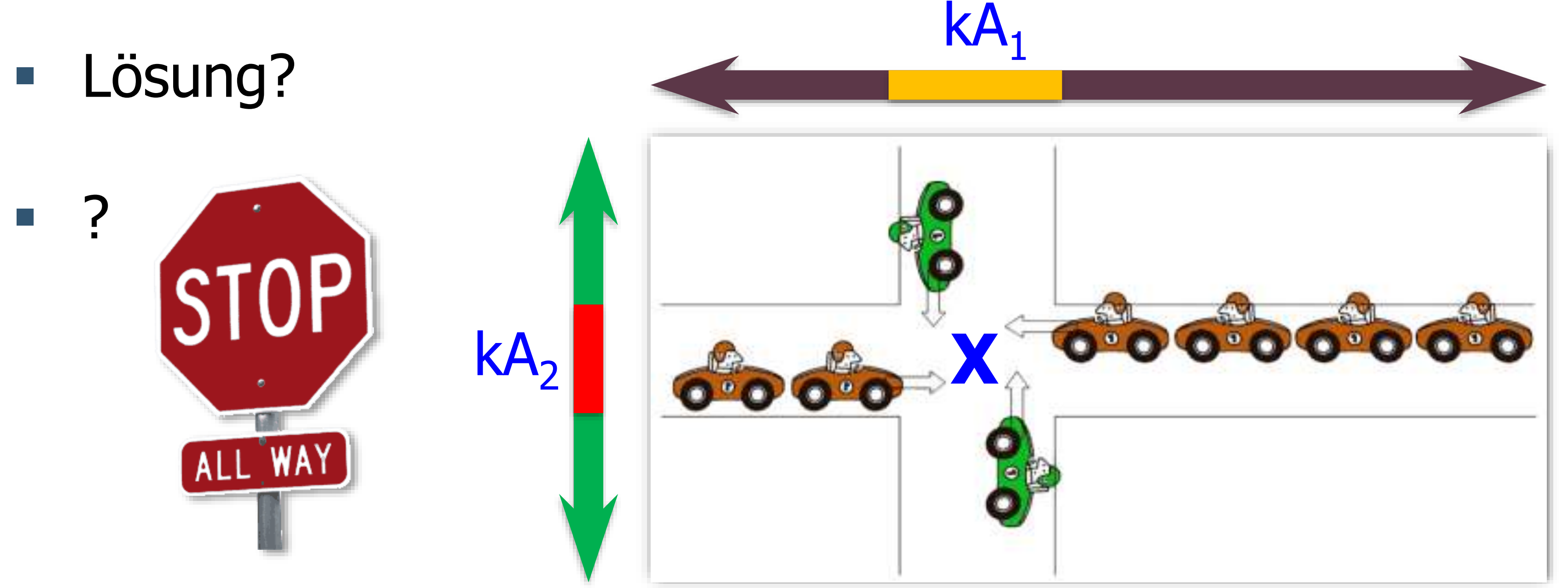
Am 17.2.2019, einem Sonntagabend, ist der Intercity ICE 373 bei Basel entgleist. Das Unglück passierte, als der Zug aus Berlin in Richtung Interlaken den Badischen Bahnhof verlassen hatte: Nach einer Weiche fuhr die Lok auf dem rechten von zwei Gleisen, der Rest des Zuges aber auf dem linken. Der erste Personenwagen hängt als einziges Verbindungsstück der beiden Zugteile schräg zwischen den Spuren und sammelt allmählich einen Schotterberg an. So fährt der ICE 800 Meter zweigleisig. Rund 20 Meter weiter werden die beiden Gleise durch eine Betonwand getrennt. SBB-Einsatzleiter Martin Spichale sprach am Montag vor Ort vor den Medien von Glück im Unglück, dass nicht mehr passiert ist. Ein Fahrleitungs- oder Signalmasten zwischen den beiden Gleisen oder gar ein Gegenzug hätte zu einem ungleich grösseren Schaden geführt.

Kritischer Abschnitt

- Kritischer Abschnitt = Folge von Anweisungen, die bezüglich anderen „entsprechenden" kritischen Abschnitten wechselseitig ausgeschlossen ist
  - D.h. während ein Thread im kritischen Abschnitt ist, darf kein anderer Thread einen entsprechenden kritischen Abschnitt betreten
  - Höchstens einer hat also die Erlaubnis
  - „Mutual exclusion" der Threads („mutex")
- In einen kritischen Abschnitt kommen solche Operationen, die ungestört als Ganzes ausgeführt werden müssen
  - Z.B. Einfügen eines Elementes in den nächsten freien Array-Platz zusammen mit Hochzählen der Variablen, die diesen Platz anzeigt
  - Oder: Zugriff auf ein exklusives Betriebsmittel in Konkurrenz zu anderen Prozessen (z.B. zu beschreibende Datei)

Kritischer Abschnitt: Beispiel

- Ein Auto auf der Ost-West-Strasse darf nicht im kritischen Abschnitt kA_1 dieser Strasse sein, wenn gleichzeitig ein Auto auf der Nord-Süd-Strasse in deren kritischem Abschnitt kA_2 ist
  - Die beiden „entsprechenden" kA sollen sich gegenseitig ausschliessen
- Lösung?
- ?



- Hauptstrasse / Nebenstrasse wäre unsymmetrisch („unfair") und könnte zum Verhungern der Autos der Nebenstrasse führen

Kritischer Abschnitt: Anforderungsspezifikation

- Das Problem besteht darin, ein Regelwerk („Protokoll") zu entwerfen, an das sich dann alle Prozesse bzw. Threads halten, und das die Semantik des kritischen Abschnitts (kA) realisiert
- Was gehört zur Semantik des kA?
- Drei Anforderungen:
  - Safety
  - Liveness
  - Fairness

Die Semantik muss exakt definiert werden (und das Protokoll relativ dazu verifiziert werden), damit man sich auf korrektes Verhalten parallel agierender Einheiten verlassen kann, die bezüglich eines kritischen Abschnitts wechselseitig ausgeschlossen werden sollen.

Man denke z.B. an selbstfahrende Autos, die gleichzeitig eine Kreuzung erreichen; gleiches gilt aber auch allgemein für parallele Prozesse bzw. Threads.

Safety, Liveness, Fairness – Aller guten Dinge sind drei

(1) **Safety** *(„nothing bad will ever happen")*

Wenn ein Prozess im kA ist, dann kein anderer

→ Kritische Abschnitte sollten kurz / schnell sein, um andere Threads nicht zu behindern!

Das alleine genügt aber nicht; sonst wäre ein Protokoll, das keinem Prozess je den Zutritt erlaubt, korrekt! (Alle Verkehrsampeln auf Dauerrot machen Zürich sicher!)

(2) **Liveness** *(bzw. „progess": „something good will eventually happen")*

Wenn kein Prozess im kA ist, aber einige sich „bewerben", dann kommt einer von diesen baldmöglichst in den kA

Liveness ohne Safety ist auch wieder trivial! Gesucht ist eine Lösung, die Safety *und zugleich* Liveness erfüllt

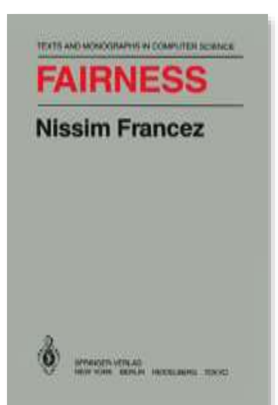
(3) **Fairness**

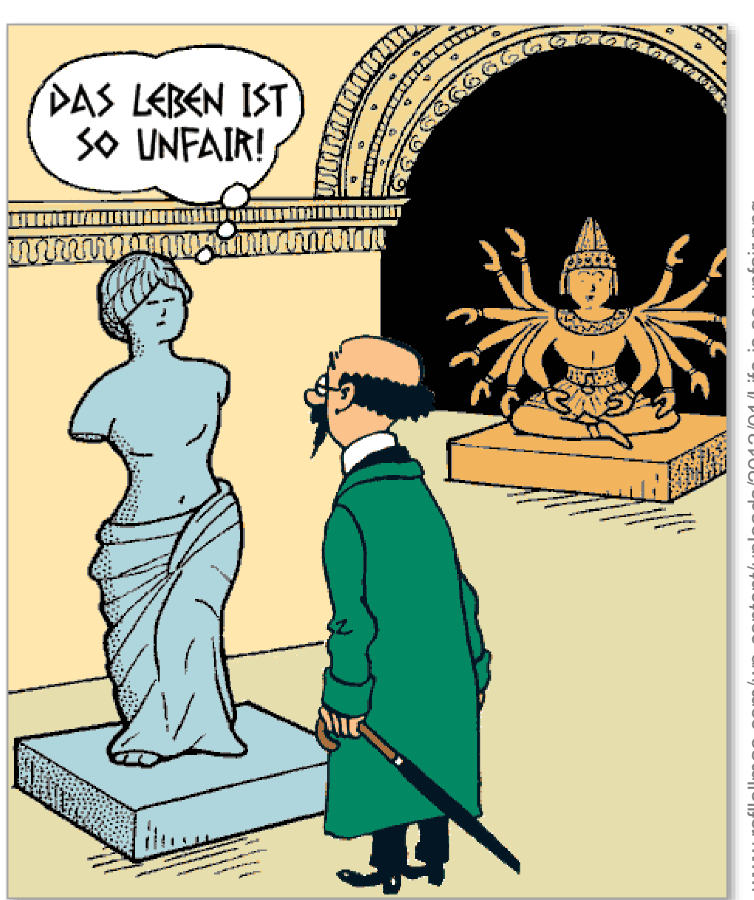
Ein sich bewerbender Prozess darf nicht dauernd (bzw. allzu oft) von anderen Prozessen übergangen werden

Sonst könnten sich etwa zwei Prozesse abwechselnd das Recht zuspielen und einen dritten Prozess verhungern lassen („starvation")

Fairness?

- Fairness ist ein positiv konnotierter Begriff (auch wir Nicht-Bauern sind für einen „fairen Milchpreis"), der aber formal schwierig zu fassen ist
  - Ethisches Gerechtigkeitsgebot für programmierte Prozesse scheint unsinnig
  - Aber wie kann man bei begrenzter Zeit und Geduld feststellen, ob ein Prozess „nicht dauernd" übergangen wird?
  - Und was heisst „nicht allzu oft" genau?
- Zum formalen Fairness-Begriff wurden jedenfalls schon ganze Bücher geschrieben
  - Wir diskutieren das hier aber nicht weiter





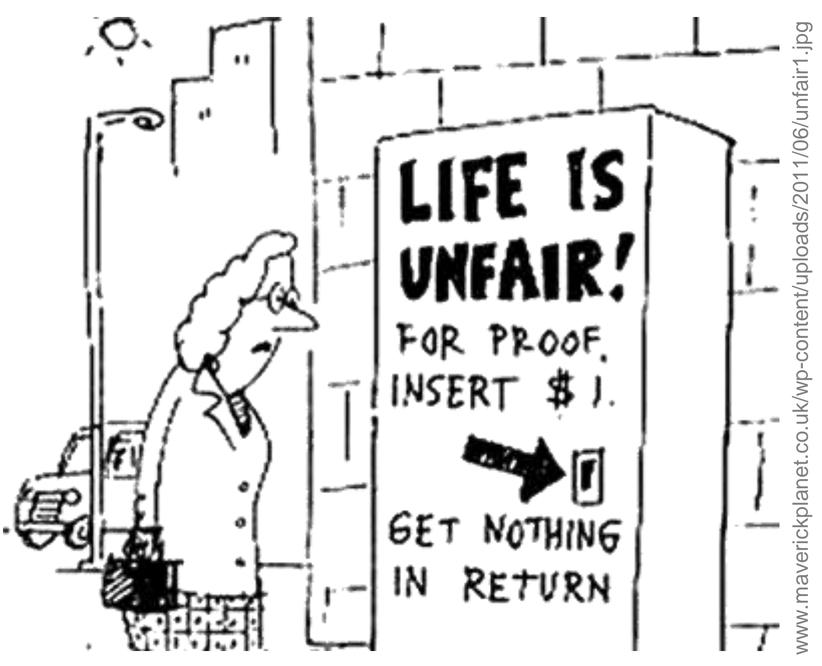
www.roflollmao.com/wp-content/uploads/2013/01/Life-is-so-unfair.png

"My daughter and I taking a shower with equal frequency is a frightening thought for both of us." -- E.W. Dijkstra

Fairness?

> "I can easily promise to think at least three times per week about you, but that is a very cheap promise because no one will ever be able to show that I failed to fulfil my commitment. … My conclusion from the above is that fairness, being an unworkable notion, can be ignored with impunity." -- EWD 1013

"The motivation behind any notion of fairness is to disallow infinite computations in which a system component is, for some reason, prevented from proceeding. All finite computations are fair; when infinite computations are considered, however, it may be necessary to distinguish between fair and unfair computations. Intuitively, fairness is a property of computations that can be expressed as follows: No component of the system that becomes possible sufficiently often should be delayed indefinitely. …To obtain a specific fairness property it is necessary to say explicitly what is meant by a 'system component', system component 'becoming possible', and 'sufficiently often'… Fairness and fairness-related notions stem from the observation that a certain undesirable phenomenon, often present in infinite computations admissible under a given semantics, and usually relating to the lack of progress of some component of a system, must be disallowed.…All fairness notions known to the author exclude some infinite behaviours, while all finite behaviours are considered fair… It has been maintained that fairness has no effect on partial correctness and, in general, safety properties. It does, however, affect liveness properties." [M.Z. Kwiatkowska, Survey of fairness notions. Inform. and Softw. Techn. 31 (7), 1989, pp. 371-386]



www.maverickplanet.co.uk/wp-content/uploads/2011/06/unfair1.jpg

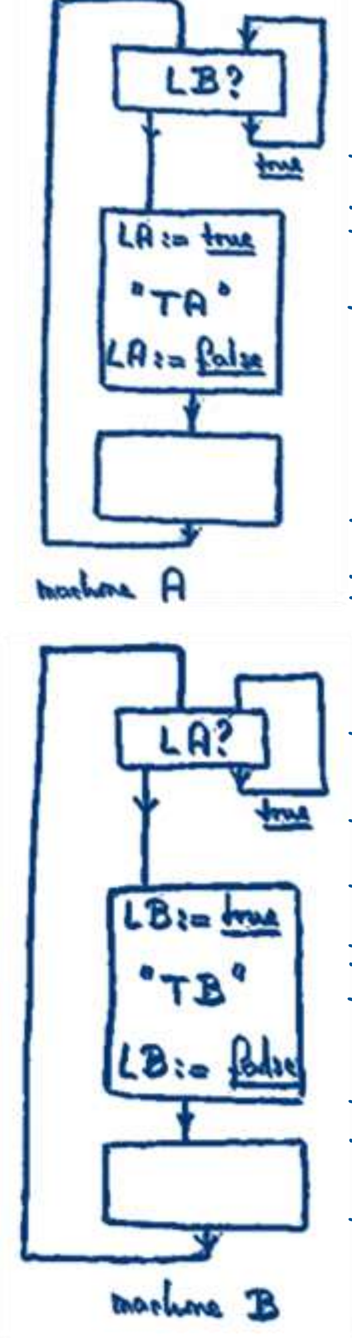
Kritischer Abschnitt: Historie

- E.W. Dijkstra beschrieb bereits 1962 in EWD 35 das Problem des wechselseitigen Ausschlusses kritischer Abschnitte
- Einige relevante Passagen in englischer Übersetzung des ursprünglich auf Holländisch („Over de sequentialiteit van procesbeschrijvingen“) verfassten Textes:

Given two machines A and B, both engaged in a cyclical process. In the cycle of machine A there is a certain critical section, called TA, and in that of machine B a critical section TB. The task is to make sure that never simultaneously both machines are each in their critical section. There should be no assumptions made on the relative speeds of the machines. […] It is clear that we can realize the mutual exclusion of the critical section only, if the two machines are in some way or another able to communicate with each other. For this communication, we establish some shared memory, i.e. a number of variables, which are accessible to both machines.

[…] There are two common logical variables in this diagram, LA and LB. LA means machine A is in its critical section, LB means machine B is in its critical section. […] In the block top machine A waits if it arrives at a moment when machine B is busy in section TB, until machine B has left its critical section, which is marked by the assignment “LB: = false”, which next lifts the wait of machine A. And vice versa. The schemata may be simple, they are unfortunately also wrong, because they are a bit too optimistic: they don’t exclude that both machines simultaneously enter their respective critical sections. If both machines are outside their critical section – say somewhere in the block left blank – then both LA and LB are false. If now simultaneously they enter in their upper block, they both find that the other machine does not impose any obstacle in their way, and they both go on and arrive simultaneously in their critical section.

Keine *Safety* nach heutiger Begriffsbildung



…dan vinden ze beide, dat de andere machine hen geen strobreed in de weg legt, en ze gaan beide door en komen tegelijkertijd in hun critische sectie.

Kritischer Abschnitt: Historie (2)

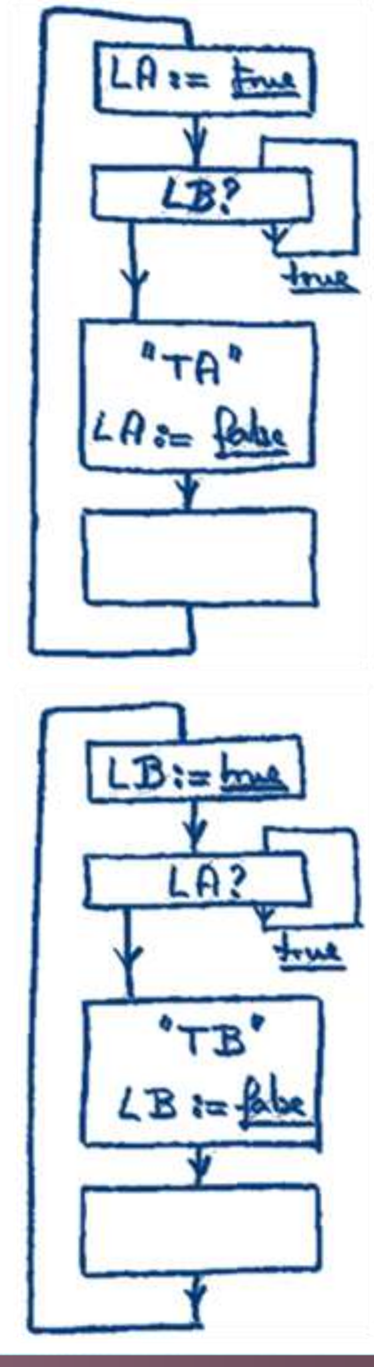
So we have been too optimistic. The error has been that the one machine did not know whether the other was already inquiring about his state of progress. The schemata of Fig. 2 make a much more reliable impression, and it is easy to verify that they are totally secure. E.g. machine A can only start to execute section TA, after, while LA is being true, it has been verified that LB is false. No matter how soon after the knowledge of this fact to machine A becomes obsolete, because machine B in its upper block performs "LB = true", machine A can safely enter section TA, because machine B will be neatly held up until after finishing section TA, LA is being changed to false. This solution is perfectly safe, but we should not think that with this we have solved the problem, because this solution is too safe, i.e. there is a chance that the whole teamwork of these machines halts. If they run the upper block in their schema simultaneously, then both machines run in the subsequent wait and continue into eternity of days politely facing the same door, saying, "After you", "After you".

Keine *Liveness* nach heutiger Begriffsbildung

We have been too pessimistic. You see, the problem is not trivial. It was to me at least, after these two try-outs, not at all obvious that there was a safe solution, that did not also contain the possibility of a dead end. I have passed the problem in this form to my former colleagues of the Computational Department of the Mathematical Centre, adding that I did not know if it has a solution. Dr. T. J. Dekker has the honour of being the first to have found a solution, which was also symmetrical in both machines. […] A third logical variable is being introduced, namely AP, which means that in case of doubt machine A has priority.

Ist das *fair*?

- Wir gehen an dieser Stelle nicht weiter auf die Lösung von Dekker und weitere im Aufsatz beschriebene Lösungen ein, in denen Dijkstra auch seine bekannten Semaphore (mit P- und V-Operationen) einführt



After-you-after-you blocking

Das „after-you-after-you blocking" (manchmal auch „politelock" genannt) wurde zu einem geflügelten Wort in der Informatik; es handelt sich um eine Instanz eines „livelock".

"I once almost got into a deadlock situation in an elevator with a distinguished computer scientist (Turing Award winner) when I was a graduate student. When the elevator door opened, I automatically paused to let my superior exit first, while he, an older gentleman, waited for me to exit, my being a woman. As soon as I realized what was going on, I exited, thinking that more respectful than insisting on my preferred protocol. (No, it wasn't Dijkstra)"

"If more than one computer is about to enter its critical section, it must be impossible to devise for them such finite speeds, that the decision to determine which one of them will enter its critical section first is postponed until eternity. In other words, constructions in which 'After you'-'After you'-blocking is still possible, although improbable, are not to be regarded as valid solutions." [E.W. Dijkstra, 1965]

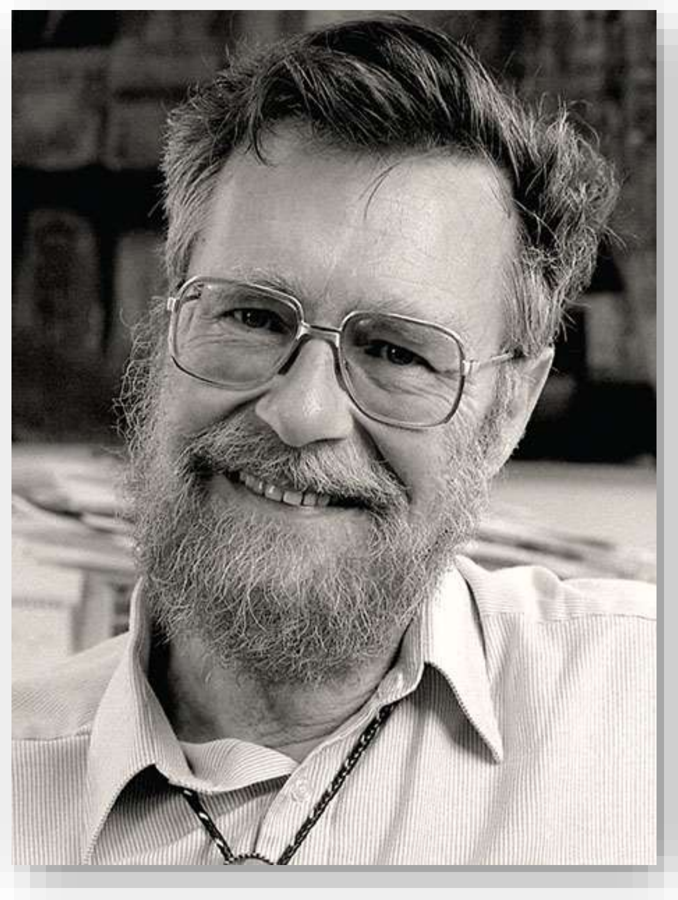
E.W. Dijkstra: Kritische Abschnitte und mehr

Edsger W. Dijkstra (1930 – 2002): Dijkstra studierte Mathematik und theoretische Physik an der Universität Leiden. *“On the whole I enjoyed the years at the university very much. We were very poor, worked very hard, never slept enough and often did not eat enough but life was incredibly exciting. My father, who was subscriber to ‘Nature’, had seen the announcement of a three-week course in Cambridge on programming for an electronic computer (for the EDSAC, to be precise), and when I had passed the midway exam before the end of the third year – which was relatively early – he offered me by way of reward the opportunity to attend that course. I thought it a good idea, because I intended to become an excellent theoretical physicist and I felt that in my effort to reach that goal, the ability to use an electronic computer might come in handy.”*

1952 arbeitete Dijkstra erst halbtags, nach seinem Diplom dann voll, quasi als erster Programmierer der Niederlande, am Mathematischen Zentrum in Amsterdam. 1959 schrieb er an der Universität von Amsterdam seine Doktorarbeit über den vom Mathematischen Zentrum entwickelten Computer „Electrologica X1“, dessen grundlegende Software er entwickelte. 1962 wurde er Professor für Mathematik in Eindhoven. Leistungen u.a.: „Dijkstra-Algorithmus“ (1959) zur Berechnung kürzester Wege in Graphen; Compiler für Algol (1960); Konzept der „Semaphore“ (1965) zur Synchronisation paralleler Prozesse; erstes Mutiprogramming-Betriebssystem („THE“) mit dynamischen Prozessen (um 1967); „Cooperating Sequential Processes“ (1968); Algorithmen zur verteilten Terminierung (1980 und 1986). 1972 erhielt er den Turing Award. Mehr zu Dijkstra findet man hier: *Krzysztof Apt: Edsger Wybe Dijkstra (1930–2002): A Portrait of a Genius. Formal Aspects of Computing 14.2 (2002): 92-98.*

Dijkstras Studentenausweis von 1951

E.W. Dijkstra... (2)



Ergänzend noch einige Auszüge aus einem Interview, das wenige Monate vor DijkstrasTod geführt wurde (Thomas Misa, Philip Frana: *An interview with Edsger W. Dijkstra*. Communications of the ACM 53(8), 2010, 41-47). Das Interview ist in seiner Gesamtheit sehr lesenswert!

It all started in 1951, when my father enabled me to go to a programming course in Cambridge, England. It was a frightening experience: the first time that I left the Netherlands, the first time I ever had to understand people speaking English. I was all by myself, trying to follow a course on a totally new topic. […] I concluded that the intellectual challenge of programming was greater than the intellectual challenge of theoretical physics. [...] It was in 1955 when I decided not to become a physicist, to become a programmer instead. [...] The physicists considered me as a deserter, and the mathematicians were dismissive and somewhat contemptuous about computing. In the mathematical culture of those days you had to deal with infinity to make your topic scientifically respectable. [...]

I designed a program that would find the shortest route between two cities in the Netherlands. […] One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path. As I said, it was a 20-minute invention. [...] One of the reasons that it is so nice was that I designed it without pencil and paper. Without pencil and paper you are almost forced to

E.W. Dijkstra... (3)

avoid all avoidable complexities. Eventually that algorithm became, to my great amazement, one of the cornerstones of my fame. [...] What's all the fuss about? It remained unpublished until Bauer asked whether we could contribute something. […] It was originally published in 1959 in *Numerische Mathematik* edited by F.L. Bauer. [...]

Dijkstra entwickelte seinen Kürzesten-Wege-Algorithmus 1956 im Alter von 26. David Gries (US-amerikanischer Informatiker Jg. 1939, Promotion 1966 an der TU München bei F.L. Bauer, später Professor in Stanford und an der Cornell University) schrieb dazu:

"Imagine that! He developed it in his head in 20 minutes. No paper and pencil. […] One could ask: Why wasn't Edsger talking to his fiancé instead of thinking about an algorithm? Ha! Well, his fiancé, Ria Debets, knew him well, and she was also a programmer. She was one of a dozen women who had completed high school with exceptionally high grades in mathematics and were hired to work in the new computing department at the Mathematical Center in Amsterdam. Edsger had taught Ria and the other women programming. He was working on his PhD, which he got in 1959. They were married about a year later, and they were close companions until he died in 2002."

Ria Debets erinnert sich an ihre erste Begegnung mit Dijkstra im Jahr 1951: "I still remember it well, the day my future husband entered my life. He was a good-looking man, 20 years of age. He entered our Computing Department with a cane!"

Ria Debets und Edsger Dijkstra, ca.1956

E.W. Dijkstra... (4)

For those first five years I had always been programming for non-existing machines. [...] All the programming I did was on paper. [...] There was not a way to test them, so you've got to convince yourself of their correctness by reasoning about them. [...] In 1959, I had challenged my colleagues at the Mathematical Centre with the following programming task. Consider two cyclic programs, and in each cycle a section occurs called the critical section. [...] My friends at the Mathematical Centre handed in their solutions, but they were all wrong. For each, I would sketch a scenario that would reveal the bug. People made their programs more sophisticated and more complicated. The construction and counterexamples became even more time-consuming, and I had to change the rules of the game. I said, "Sir, sorry, from now onward I only accept a solution with an argument why it is correct." Within three hours or so Th. J. Dekker came with a perfect solution and a proof of its correctness.

My international contacts had started in December 1958, with the meetings for the design of ALGOL 60. [...] The ALGOL 60 meetings were about the first time that I had to carry out discussions spontaneously in English. It was tough. Computing science started with ALGOL 60. Now the reason that ALGOL 60 was such a miracle was that it was not a university project but a project created by an international committee. It also introduced about a half-dozen profound novelties. [...] A second novelty was that at least for the context-free syntax, a formal definition was given. [...] It made compiler writing and language definition topics worthy of academic attention. [...] The fourth novelty was the introduction of recursion into imperative programming. [...] I phoned Peter Naur—that call to Copenhagen was my first international telephone call; I'll never forget the excitement!—and dictated to him one suggestion. It was something like "Any other occurrence of the procedure identifier denotes reactivation of the procedure." That sentence was inserted sneakily. And of all the people who had to agree with the report, none saw that sentence. That's how recursion was explicitly included. [...] F.L. Bauer would never have admitted it in the final version of the ALGOL 60 Report, had he known it.

E.W. Dijkstra… (5)

A fifth novelty that should be mentioned was the block structure. The concept of lexical scope was beautifully blended with nested lifetimes during execution, and I have never been able to figure out who was responsible for that synthesis, but I was deeply impressed when I saw it.

In 1967 was the ACM Conference on Operating Systems Principles in Gatlinburg. […] It was at that meeting where one afternoon I explained to Brian Randell and a few others why the GO TO statement introduced complexity. And they asked me to publish it. So I sent an article called "A Case Against the GO TO Statement" to *Commun. of the ACM*. The editor of the section wanted to publish it as quickly as possible, so he turned it from an article into a Letter to the Editor. And in doing so, he changed the title into, "GO TO Statement Considered Harmful. "That title became a template. Hundreds of writers have "X considered harmful," with X anything. The editor who made this change was Niklaus Wirth.

Numerische Mathematik 1, 269–271 (1959)

A Note on Two Problems in Connexion with Graphs

By

E. W. Dijkstra

We consider n points (nodes), some or all pairs of which are connected by a branch; the length of each branch is given. We restrict ourselves to the case where at least one path exists between any two nodes. We now consider two problems.

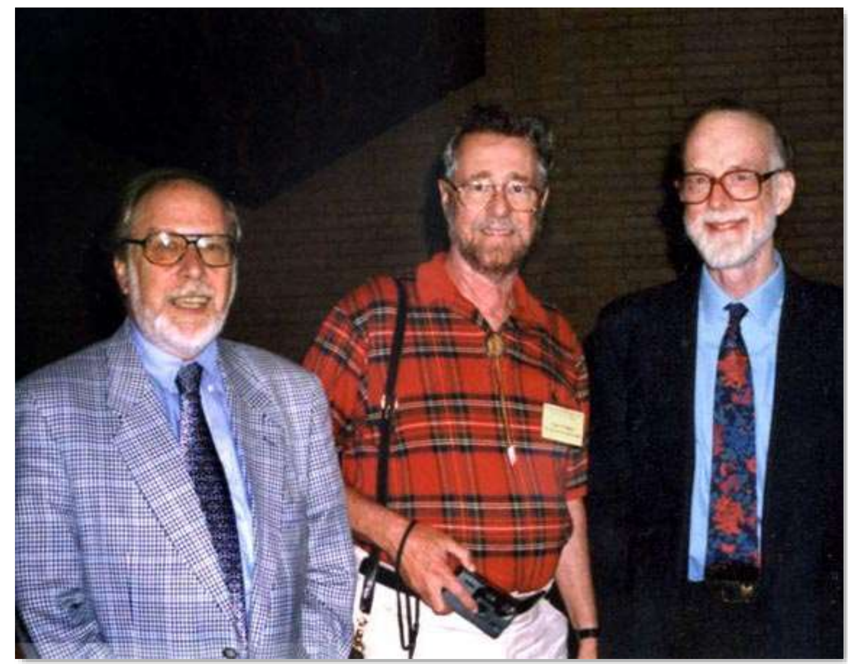
Problem 1. Construct the tree of minimum total length between the n nodes. (A tree is a graph with one and only one path between every two nodes.)

Problem 2. Find the path of minimum total length between two given nodes P and Q.

We use the fact that, if R is a node on the minimal path from P to Q, knowledge of the latter implies the knowledge of the minimal path from P to R. In the solution presented, the minimal paths from P to the other nodes are constructed in order of increasing length until Q is reached.

Wie Dijkstra zu seinem Bart kam (oder: Der unfertige Compiler): "Begin 1960 begon Jaap Zonneveld (1924 – 2016) samen met Edsger Dijkstra dag en nacht aan de ontwikkeling van een compiler voor de programmeertaal Algol-60. Zij spraken af zich niet te scheren voordat de ontwikkeling van de compiler klaar zou zijn. Op 24 augustus van dat jaar was deze gereed en was daarmee de eerste Algol-60 compiler ter wereld. Voor wat er na de voltooiing zou gebeuren werd niets afgesproken; Dijkstra hield de rest van zijn leven een baard." https://nl.wikipedia.org/wiki/Jaap_Zonneveld

E.W. Dijkstra... (6)



Niklaus Wirth, Edsger Dijkstra, Tony Hoare
(Anfang der 1990er-Jahre)

F.L. Bauer, Edsger Dijkstra, Tony Hoare während der Summer School “Theoretical Foundations of Programming Methodology“ in Marktoberdorf 1981. Dijkstra verfasste einen “Trip Report” [EWD799], daraus einige Passagen: The Summer School started in real earnest on Wednesday 29 July at 08:45 and lasted until Saturday 8 August at 17:00. […] The program was overloaded as usual. […] The British speakers were collectively by far the best, the French were by far the worst, condemned as they were -- and I am afraid, rightly so -- of the crime of “contempt of audience”. […] Tony Hoare gave an equally clear overview of his theory of Communicating Sequential Processes, his careful phrasing made him a pleasure to listen to. […] Maurice Nivat and Bruno Courcelle were very French. Of both I attended the first third of their lectures; the exposure confirmed my impression that for computing science, Bourbaki is not the ideal starting point. […] Our Manfreds -- Wirsing and Broy -- were very German: the first one encyclopedic, the second one belligerent. Being well-prepared and carefully presented, their lectures were not unpleasant to listen to. […] It was a good thing the Summer School did not last longer: at the end we were all exhausted. In Johanniskreuz (near Kaiserlautern) I started a series of 12-hour nights.

E.W. Dijkstra... (7)

Edsger Wybe Dijkstra
The University of Texas at Austin
Keine bestätigte E-Mail-Adresse

Einige der meistzitierten Papers von E.W. Dijkstra (Zitationswerte 1/2020) laut Google Scholar →

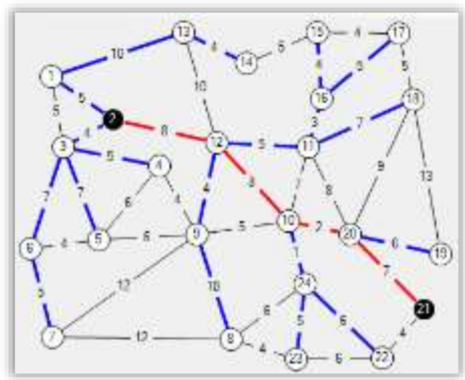
TITEL	ZITIERT VON	JAHR
A note on two problems in connexion with graphs EW Dijkstra Numerische mathematik 1 (1), 269-271	24880	1959
A discipline of programming EW Dijkstra, EW Dijkstra, EW Dijkstra, EU Informaticien, EW Dijkstra prentice-hall	7652	1976
Cooperating sequential processes EW Dijkstra The origin of concurrent programming, 65-138	2991	1968
Self-stabilizing systems in spite of distributed control EW Dijkstra Communications of the ACM 17 (11), 643-644	2607	1974
Guarded commands, nondeterminacy, and formal derivation of programs EW Dijkstra Programming Methodology, 166-175	2582	1978
Structured programming OJ Dahl, EW Dijkstra, CAR Hoare Academic Press Ltd.	2447	1972
Go to statement considered harmful EW Dijkstra Communications of the ACM 11 (3), 147-148	1843	1968
The structure of the "THE" multiprogramming system EW Dijkstra The origin of concurrent programming, 139-152	1795	1968

So then I typed GOTO 500 – and here I am!

http://pulkomandy.tk/drop/GOTO500.png

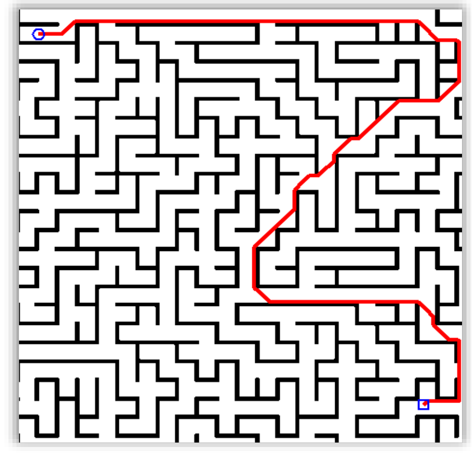
Dijkstra in jedem Navi

Sebastian Stiller erzählt in seinem lesenswerten Buch „Planet der Algorithmen“ (2015) die Geschichte des oben erwähnten „Dijkstra-Algorithmus“ zur Berechnung des kürzesten Weges zwischen einem Startknoten und den übrigen Knoten in einem (kantengewichteten) Graphen so:



„Er wurde 1956 von Edsger W. Dijkstra erdacht, allerdings nicht für Autofahrer, sondern für Bahnfahrer in den Niederlanden. Dijkstra arbeitete damals am niederländischen Zentrum für Informatik und Mathematik, das noch heute zu den weltweit führenden Forschungsinstituten für Algorithmen zählt. Der Staat hatte dem Zentrum einen unglaublich teuren Rechner gekauft.

Zur feierlichen und öffentlichen Einweihung des neuen Rechners sollte sich Dijkstra daher etwas offensichtlich Sinnvolles einfallen lassen, das der teure Rechner vormachen konnte. Er entschied sich, einen Algorithmus zu entwickeln, der im niederländischen Bahnnetz die kürzeste Verbindung zwischen zwei Städten bestimmen konnte: Die erste Anwendung des Dijkstra-Algorithmus hatte also ganze 64 Knoten. Eine Route zwischen zwei Bahnhöfen zu berechnen, dauerte etwa eine Minute. Die Einweihungsshow war ein voller Erfolg. Veröffentlicht hat Dijkstra seinen Algorithmus erst drei Jahre später, weil es damals noch keine Journale gab, die so etwas publizieren wollten. Heute zählt Dijkstras Artikel zu den meistzitierten des Fachs, und Dijkstra läuft in jedem Navigationssystem, ob im Telefon, im Auto oder auf Webseiten.“



Schnellste und kürzeste Wege sind das übrigens gleiche Problem: Verbindungslänge in Metern oder Sekunden; schnell = kurz bzgl. Zeit.

Die Grundidee des „Dijkstra-Algorithmus“

Der Dijkstra-Algorithmus berechnet einen kürzesten (gerichteten) Weg zwischen einem gegebenen Startknoten s und einem der (oder allen) übrigen Knoten in einem kantengewichteten Graphen (Kantengewichte seien dabei nicht negativ).

Prinzip: Der Algorithmus besucht alle vom Startknoten s erreichbare Knoten. Er speichert für jeden schon besuchten Knoten seine Entfernung vom Startknoten. In jedem Schritt besucht man als nächstes den Knoten, der durch eine Kante so mit einem schon besuchten Knoten verbunden werden kann, dass die Entfernung zu s minimal ist.

Wir diskutieren diesen Algorithmus nicht näher in der Vorlesung, behandeln ihn aber mit Beispielen in den Übungsaufgaben und Tutorien. Mehr zum Algorithmus siehe z.B. de.wikipedia.org/wiki/Dijkstra-Algorithmus

Zur Laufzeit des „Dijkstra-Algorithmus"

Ein typischer Graph für das europäische Strassennetz besteht aus 24 Millionen Knoten und 58 Millionen Kanten.

[Nach Till Tantau]

Sei n die Anzahl der Knoten und m die Anzahl der Kanten:

- *Naive Implementation:*
  Man muss n mal den nächsten Knoten finden, was jeweils n^2 lang dauert, insgesamt also $O(n^3)$.
- *Nicht ganz so naive Implementation:*
  Man merkt sich für jeden noch nicht besuchten Knoten den Abstand von s, wenn man ihn durch eine Kante zu den bereits besuchten verbinden würde. Dann kann man immer das Minimum dieser Liste nehmen. Man erreicht eine Laufzeit von $O(n^2)$.
- *Clevere Implementation:*
  Man benutzt eine Heap-Datenstruktur. Dies ermöglicht eine Laufzeit von $O((n+m) \log n)$.
- *Ganz clevere Implementation:*
  Man benutzt sogenannte Fibonacci-Heaps. Dies ermöglicht eine Laufzeit von $O(n \log n + m)$.

Java: das Synchronized-Konstrukt

- Bei Java wird ein kritischer Abschnitt durch eine in {...}-geklammerte Anweisungsfolge mit vorangestelltem Schlüsselwort „synchronized" spezifiziert:

```
synchronized (~~~) {
  // Anweisung 1
     ...
  // Anweisung n
}
```

Hier steht ~~~ für irgendeine Objektreferenz („Sperrobjekt"; „lock")

- Meist wird ~~~ innerhalb der Anweisungsfolge verwendet
  - D.h. es handelt sich um das „kritische Objekt", bezüglich dessen der wechselseitige Ausschluss realisiert sein soll
  - Es kann sich aber auch um ein eigenes stellvertretendes Sperrobjekt handeln, z.B.:

```
Object Zeus = new Object;
synchronized (Zeus) {...
```

Wirkung von „synchronized“

- Alle bezüglich des gleichen Sperrobjektes synchronisierten kritischen Abschnitte schliessen sich wechselseitig aus
  - Wird automatisch durch die Java-VM garantiert
  - Beispiel:

```
synchronized (Konto) {
  float a = Konto.Stand();
  a = a + Betrag;
  Konto.Update(a);
}
```

- Beachte: unkooperative Threads können aber weiterhin, z.B. über die Objektreferenz (hier: „Konto“), auf das „gesperrte“ Objekt zugreifen und es „parallel“ manipulieren
  - Die (richtige) Verwendung des Sperrmechanismus liegt in der Verantwortung des Programmierers!

Verkehrsampeln bieten auch keine absolute Sicherheit (vor Rotlichtignoranten!)

Beispiel für „synchronized“

```
Initial:   int a = 1, b = 2;
Thread 1:  a = b;
Thread 2:  b = a;
```

> *In the absence of explicit synchronization, a Java implementation is free to update the main memory in an order that may be surprising. Therefore the programmer who prefers to avoid surprises should use explicit synchronization.* (Java Lang. Spec.)

- Zuweisungen sind nicht atomar!
  1. Lesen aus dem Hauptspeicher in den Arbeitsbereich des Thread (u.a. Stack, CPU-Register...)
  2. Schreiben aus dem Arbeitsbereich in den Hauptspeicher
- Daher sind drei Ergebnisse möglich:
  - a = 2, b = 2 (Thread 1 ganz vor Thread 2)
  - a = 1, b = 1 (Thread 2 ganz vor Thread 1)
  - a = 2, b = 1 (vertauschte Werte durch Interleaving!)

```
iload_1
            iload_0
            istore_1
istore_0
```

- Durch „synchronized“ wird zumindest der 3. Fall vermieden:

```
Thread 1: synchronized(Sperre) {a = b;}
Thread 2: synchronized(Sperre) {b = a;}
```

jetzt neu

Paralleles Inkrementieren – mit „synchonized“

Vergleiche dies mit der früheren Variante des „Rätsels“ ohne „synchronized“!

```
class ParIncr extends Thread {
  int i; // individuelle Variable
  static int j = 0; // gemeinsame Var.
  static Object Sperre = new Object();

  public void run() {
    for (i = 0; i < 400000000; i++)
      synchronized(Sperre) { j++; }
    System.out.println("i: " + i + "   j: " + j);
  }

  public static void main(String [] args) {
    for (int k = 0; k < 5; k++) new ParIncr().start();
  }
}
```

Und wenn man hier kein static hätte?

i	j
i: 400 000 000	j: 1 963 358 528
i: 400 000 000	j: 1 985 600 635
i: 400 000 000	j: 1 986 266 174
i: 400 000 000	j: 1 997 524 428
i: 400 000 000	j: 2 000 000 000

Und wenn wir die ganze for-Schleife mit synchronized schützen würden?

Und wenn j vom Typ „long“ wäre?

- Die Ausführung dauert ca. 200 Mal länger als ohne „synchronized“
- Von Lauf zu Lauf zwar noch immer leicht unterschiedliche Werte bei j, aber meist aufsteigend und finale Ausgabe (2 000 000 000) ist garantiert korrekt

Paralleles Inkrementieren -

Vergleiche dies mit der früheren Va

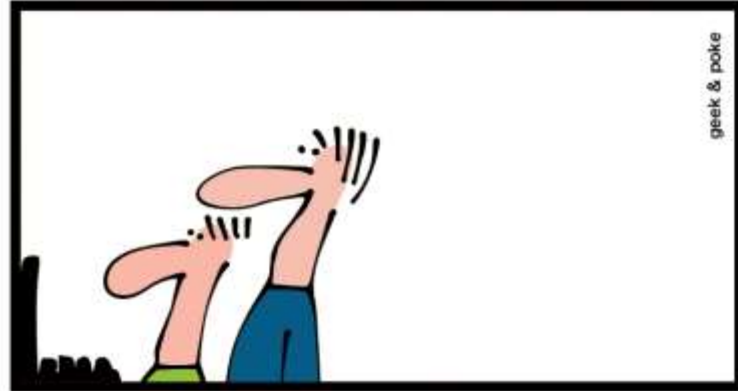
```
class ParIncr extends Thread {
   int i; /                 le
   static int j = 0; // gemeinsame
   static Object Sperre = new Objec

   public v
      for (i = 0; i < 400000000; i++
         synchronized(Sperre) { j++;
      System.out.println("i: " + i +
   }

   public static void main(String [
      for (int k = 0; k < 5; k++) ne
   }
}
```

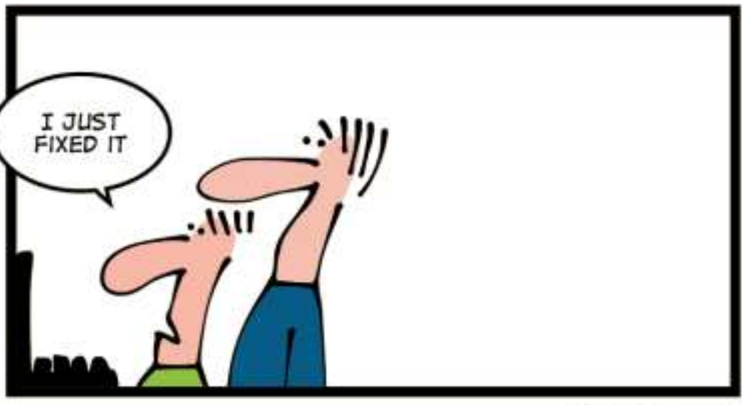
Und wenn j vom Typ „long" wäre?

Und wenn wir die ganze for-Schleife mit synchronized schützen würden?

Und wenn man hier kein static hätte?

- Die Ausführung dauert ca. 200 Mal länger a
- Von Lauf zu Lauf zwar noch immer leicht u meist aufsteigend und finale Ausgabe (2 00(





Des Rätsels Lösung – Erläuterung zur Variante ohne „synchronized“

(in Anlehnung an Reinhard Schiedermeier: Programmieren mit Java II)

Das Programm startet 5 Threads, die alle mittels j++ die gemeinsame („static“) Klassenvariable j „gleichzeitig“ hochzählen. Der Compiler übersetzt diese Anweisung analog zu j = j + 1.

Diese Anweisung läuft in einzelnen Schritten so ab:

1. Den bisherigen Wert von j lesen
2. Den Wert um 1 erhöhen
3. Den inkrementierten Wert wieder in j speichern

Ein Thread, der diese Anweisung alleine ausführt, wickelt die 3 Schritte genau in dieser Reihenfolge ab. Wenn aber zwei Threads A und B jeweils die Anweisung ausführen, können sich die sechs Schritte auf verschiedene Weise verzahnen. Mögliche Ausführungsfolgen sind z.B. (Xn bedeutet, dass Thread X Schritt n ausführt):

1. A1, A2, A3, B1, B2, B3
2. A1, B1, A2, B2, A3, B3

Die erste Folge der Schritte hinterlässt planmässig den Wert 2 in der mit 0 initialisierten Variablen. Die zweite Folge führt aber zum Ergebnis 1, weil zuerst A1 und B1 den gleichen alten Wert 0 lesen, dann A2 und B2 beide diesen auf 1 erhöhen und schliesslich A3 und B3 zweimal nacheinander den gleichen Wert in j speichern. Es gibt mehrere mögliche Ausführungsfolgen der sechs Schritte, von diesen führen einige zum intendierten, andere aufgrund der race condition zu einem unerwarteten („falschen“) Endergebnis.

Es erscheint zunächst nicht sehr wahrscheinlich, dass zwei Inkrement-Anweisungen zeitlich so nah zusammentreffen, dass sich die Schritte in der oben gezeigten Art verzahnen. Allerdings führt das

Des Rätsels Lösung – Erläuterung zur Variante ohne „synchronized" (2)

Programm sehr viele Inkrement-Anweisungen aus. Dabei kommen die problematischen Schrittfolge einige Male vor, welche dann zu den beobachteten, scheinbar falschen Ausgaben führen.

Der Anweisung j++ entspricht konkret folgender Bytecode:

```
…
16:  getstatic
17:  iconst_1
18:  iadd
19:  putstatic
…
```

Das erweist sich manchmal als ein lost update!

Dies bedeutet:

1. Der Wert der Klassenvariablen j wird geholt (16)
2. Die Konstante 1 wird bereitgelegt (17)
3. Integer-Addition von Variablenwert j und Konstante 1 (18)
4. Die Summe wird wieder in die Klassenvariable j geschrieben (19)

Potenziell kann ein Thread nach jeder dieser Bytecode-Instruktionen unterbrochen werden. Die Ursache für das Problem liegt im mehrteiligen Aufbau der täuschend kompakten, aber dennoch nicht atomar ablaufenden Inkrement-Anweisung j++. Die Anweisung wirkt zwar im Quelltext wie eine Einheit, aber der Anschein täuscht. Nur alle Einzelschritte zusammen bewirken die gewünschte Änderung.

Wenn erst nur ein Teil der Einzelschritte ausgeführt ist, befindet sich das Programm in einem Zwischenzustand mit teilweise „alten" und teilweise „neuen" Daten. Andere Threads, die diesen Zwischenzustand beobachten, sehen ein inkonsistentes Bild.

Zeitmessungen zum parallelen Inkrementieren

- Wieviel kostet die Synchronisation? Einige Zeitmessungen:

1. Ohne synchronized (die fehlerhafte Variante!): 0.82s real, 2.99s cpu
2. Mit synchronized (wie auf Seite weiter oben): 209.5s real, 377.8s cpu
3. Synchronized bezüglich der ganzen for-Schleife: 0.30s real, 0.31s cpu
4. Statt 5 ein einziger Thread, der bis 2 Mia. zählt: 0.26s real, 0.25s cpu

- Hierbei bedeuten „real" die tatsächliche Zeit („wall-clock") und „cpu" diejenige Zeit, die alle CPU-Kerne (eines Multicore-Prozessors) zusammen mit dem Programm beschäftigt waren

---

- Denkübungen:
  - Wie viele CPU-Kerne hat die Testmaschine vermutlich (mindestens)?
  - Was bedeutet es, wenn „real" und „cpu" fast gleich sind (vgl. 3.)?
  - Wieso lohnt sich bei diesem Beispiel die Parallelität nicht (vgl. 4.)?
  - Bei 3. waren die Werte für j: 400007077, 800000000, 1200027408, 1601124082, 2000000000 – wie ist das zu erklären?
  - Wieso ist 3. schneller als 1.?
  - Wieso ist **`synchronized public void`** `run()` keine Lösung?

Synchronized-Methoden

- Ganze Methoden können mit dem Attribut „synchronized“ gekennzeichnet werden:

```
synchronized void Update (int betrag)
{
  konto = konto + betrag;
  System.out.println(konto);
}
```

Best practice is that if a variable is ever to be assigned by one thread and used or assigned by another, then all accesses to that variable should be enclosed in synchronized methods or synchronized statements. (Java Language Specification)

- Synchronized-Methoden wirken wie Synchronized-Anweisungsfolgen, die bzgl. „this“ als Sperrobjekt wechselseitig ausgeschlossen sind

---

- Beim Beispiel „paralleles Inkrementieren“ führt „synchronized(this)“ übrigens nicht zum Ziel – der Zeiger „this“ wird für jeden der 5 Threads, die ja unterschiedliche Instanzen repräsentieren, neu vergeben; man hätte also 5 *verschiedene* locks! (Wie wäre es aber mit „synchronized(null)“?)

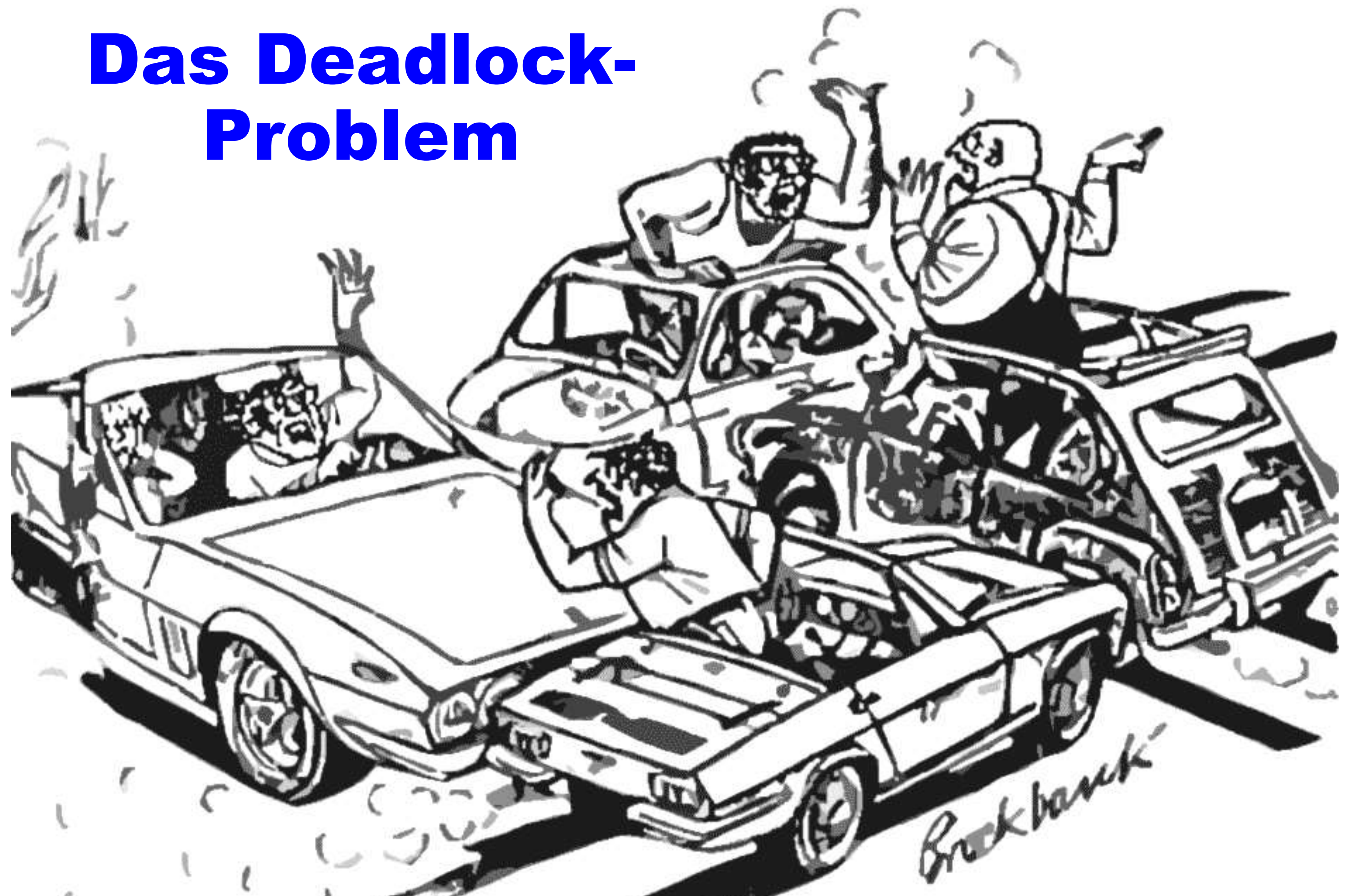
Eine Denkübung zu Synchronized-Methoden

```
public class … {
  static int z = 0;
  public synchronized void Incr() {
    println(z++);
  }
  // Gründen von zwei parallelen Threads,
  // welche beide fortlaufend Incr aufrufen;
}
```

1) Ist ausser der Ausgabefolge 0,1,2,3,4,**...** noch eine andere möglich?

2) Bei Weglassen von „synchronized“:
   a) Ist die Ausgabe dann immer aufsteigend sortiert?
   b) Können Werte doppelt ausgegeben werden?
   c) Können Werte dreifach ausgegeben werden?

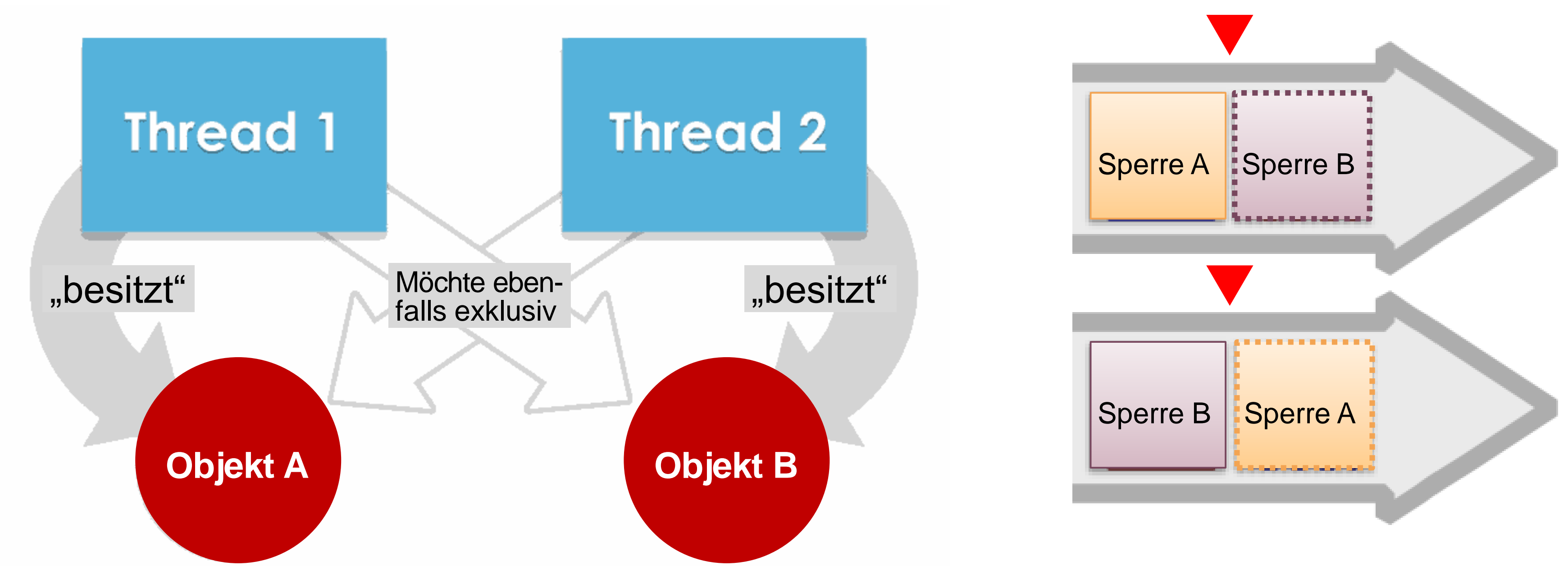
Das Deadlock-Problem



Neapolitanischer Hakenkreuzstau, nach Luciano DeCrescenzos Beschreibung im Film «Also sprach Bellavista»: Ein unentwirrbar mit sich selbst verschränktes Problem – die Wohnung müsste neu gestrichen werden, was nur geht, wenn vorher das herumstehende Zeug aus dem Weg geschafft wird, wozu man immer schon mal einige Regale anbringen wollte. An den vorher zu streichenden Wänden.

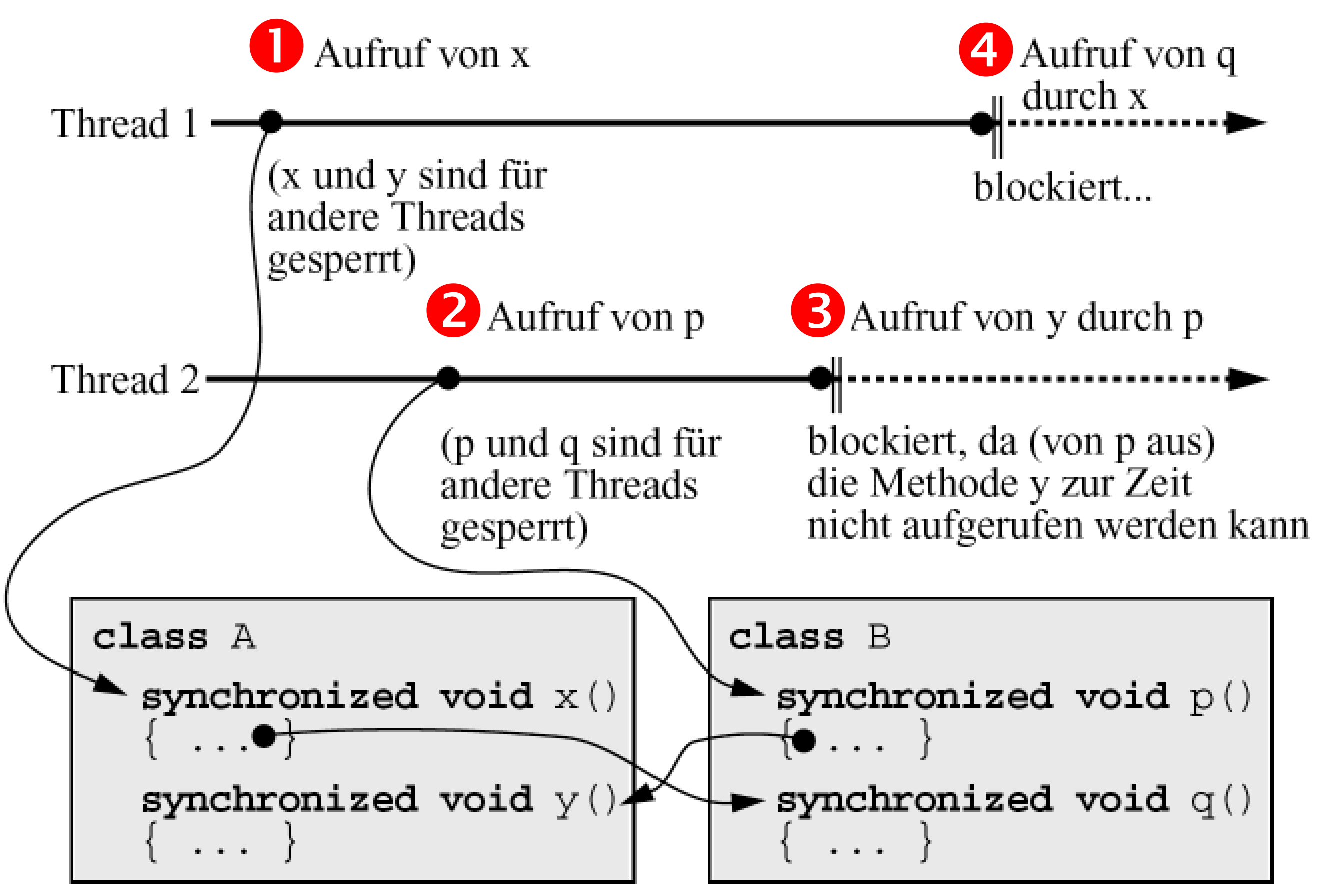
Eine solche Stauentwicklung als Video z.B. bei:
www.youtube.com/watch?v=DAXUzWnsiQk

Verklemmung zweier Threads



- „Besitzt“ bedeutet, dass der Thread das Objekt für andere Threads gesperrt hat, indem er einen Synchronized-Block dazu betreten hat
- Wenn man (zu) „streng“ synchronisiert, ist die Gefahr gross, dass es nicht nur zu einzelnen temporären Blockaden, sondern zu gegenseitigen und nicht mehr normal auflösbaren Blockaden (**Deadlocks**) kommt

Beispiel: Deadlock durch „synchronized“



Mit „Glück“ hätte das Scheduling der Threads so ausgesehen:

(1) ; (4) ; Freigabe von A ; (2) ; (3)

Auf Glück ist aber leider kein Verlass…

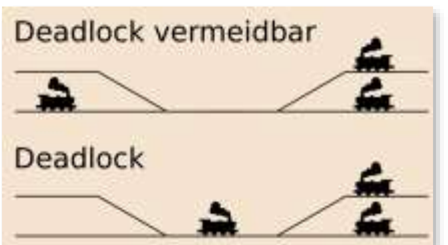
- Thread 1 wartet auf Freigabe von B durch Thread 2
- Thread 2 wartet auf Freigabe von A durch Thread 1

} **Deadlock**

Deadlocks bei der Eisenbahn

In railway operations, deadlocks are avoided by the timetable. The deadlock problem becomes evident when, in case of delay, the scheduled train sequence is changed by the dispatcher or when a railroad is operated on an unscheduled basis. – Jörn Pachl.

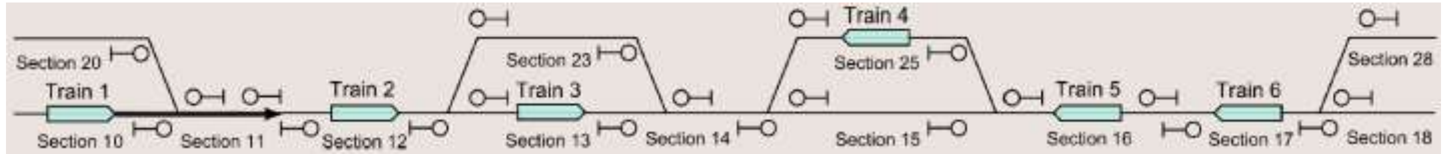
Auszüge aus de.wikipedia.org/wiki/Deadlock_(Eisenbahn), basierend auf *Jörn Pachl, Deadlock Avoidance in Railroad Operations Simulations, 2011:*



Ein Deadlock ist bei der Eisenbahn eine Situation, in der sich Züge gegenseitig blockieren, so dass keine Zugfahrt im Regelbetrieb mehr möglich ist. […] Ein möglicher Deadlock ist, wenn bei eingleisigem Zugbetrieb ein Bahnhof mit zwei Gleisen mit zwei in dieselbe Richtung fahrenden Zügen belegt ist (beispielsweise zum Überholen), aber zugleich ein Zug auf dem eingleisigen Abschnitt entgegenkommt. Ein Deadlock kann auch auf zweigleisigen Strecken auftreten. […] Der Aufwand, eine solche Situation durch Rangieren zu beseitigen, ist im Eisenbahnbetrieb sehr hoch. Es ist jedoch unmöglich, die Bahninfrastruktur eines komplexen Streckennetzes so zu bauen, dass Deadlocks grundsätzlich ausgeschlossen sind. […] Für die Entstehung eines Deadlocks in der Eisenbahn gelten dieselben vier Bedingungen wie beim Deadlock in der Informatik. Die ersten drei Kriterien sind dabei aufgrund der Struktur der Eisenbahn bzw. Zugsicherung immer erfüllt:

(1) Jeder Zugfolgeabschnitt (Blockabschnitt) kann nur von nur einem einzigen Zug belegt werden und ist dann für andere blockiert („Mutual Exclusion"). (2) Jeder Zug wartet, bis er in den nächsten Blockabschnitt einfahren kann und gibt erst danach das bisherige Gleis frei („Hold and Wait"). (3) Es können keine Züge aus dem System entfernt werden („No Preemption"). (4) Es besteht eine „Wartekette", dass ein Zug in einen Blockabschnitt einfahren muss, der wegen eines Zirkelbezugs aber erst dann frei werden kann, nachdem der Zug seinen eigenen Blockabschnitt verlassen hat.

In einem etwas komplexeren Bahnsystem mit eingleisigen Strecken oder Gleiswechselbetrieb ist es unmöglich, Deadlocks prinzipiell zu verhindern. Anders als bei Softwareprozessen, die grundsätzlich abgebrochen und neu gestartet werden können, ist es bei der Bahn im Normalfall nicht möglich, Züge kurzfristig vom Gleis zu nehmen und „neu zu starten".



Fährt Zug 1 in Abschnitt 11, dann kommt es garantiert zu einem Deadlock, unabhängig von Geschwindigkeit und Scheduling der Züge.

Deadlocks bei der Strassenbahn

Beispiel: Karlsruhe

Im Unterschied zur Eisenbahn werden bei Trams die Fahrwege nicht zentral gestellt, daher können Deadlocks leichter auftreten.

Sophien-straße

Linie 2E

Linie 1

Linie 1

Schillerstraße

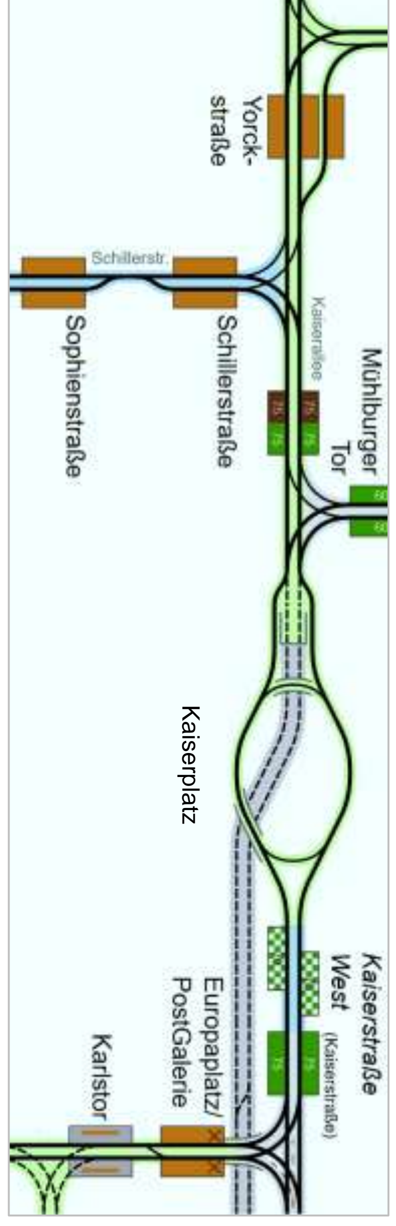
Linie 2E

Kaiserallee

Wende-schleife Kaiser-platz

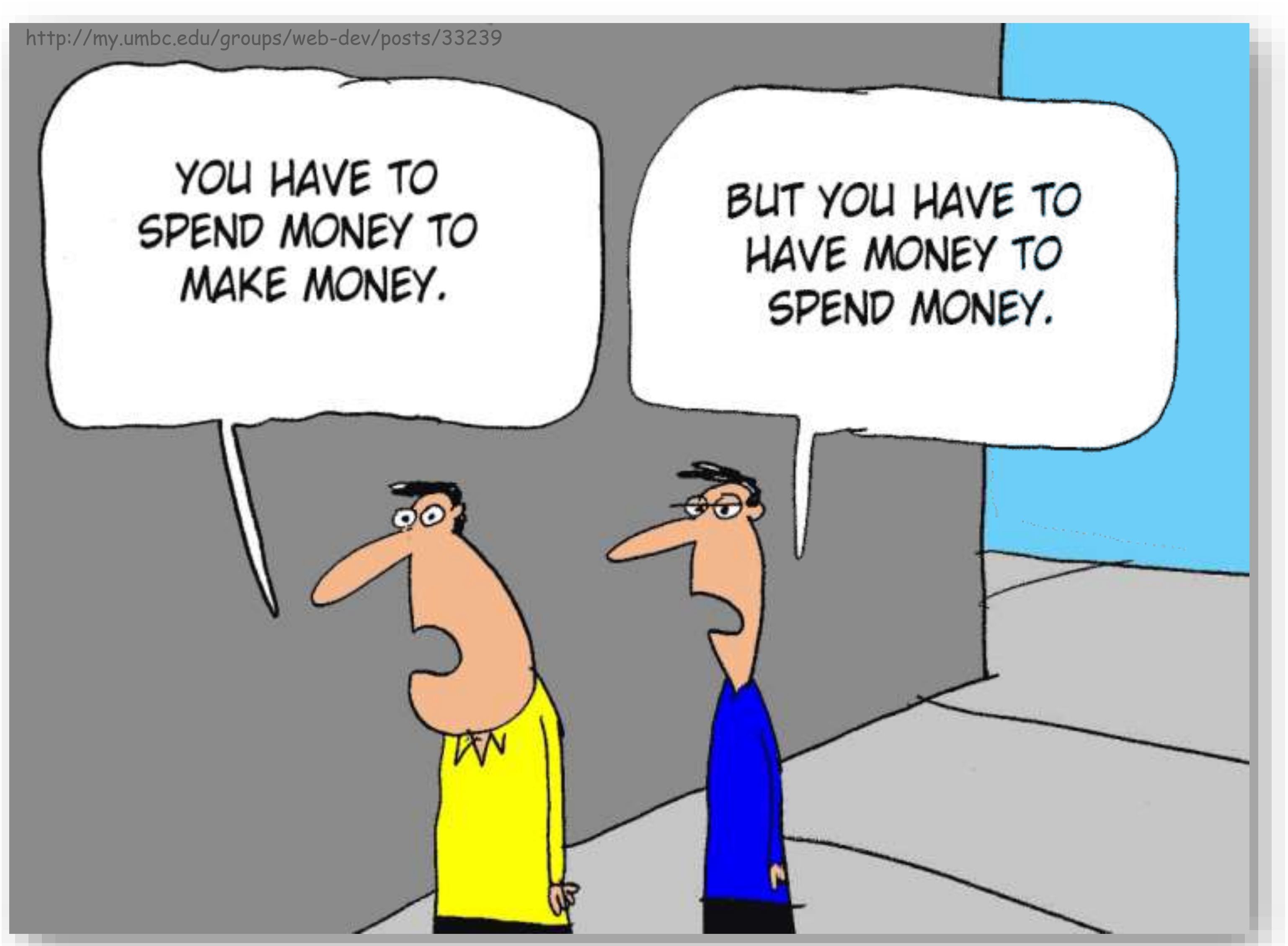
Eingleisige Schillerstraße

Schillerstraße → Kaiserallee



„Man hat dann die eigentlich nach links abbiegende Bahn nach rechts abbiegen und an einer nahe gelegenen Wendeschleife eine Ehrenrunde drehen lassen. Das war auch die einzig sinnvolle Möglichkeit, denn auf der Kaiserallee wartete inzwischen schon die nächste Bahn der Linie 1; man hätte die 1er-Doppeltraktion links im Bild über die gesamte eingleisige Strecke bis zur Sophienstraße zurücksetzen müssen.“ https://forum.zusi.de/viewtopic.php?p=223103#p223103

Deadlock



Synchronisieren?

- Ohne Synchronisation kann es zu unerwünschten Effekten kommen, z.B. → race conditions
- Mit (zu viel) Synchronisation ebenfalls, z.B.:

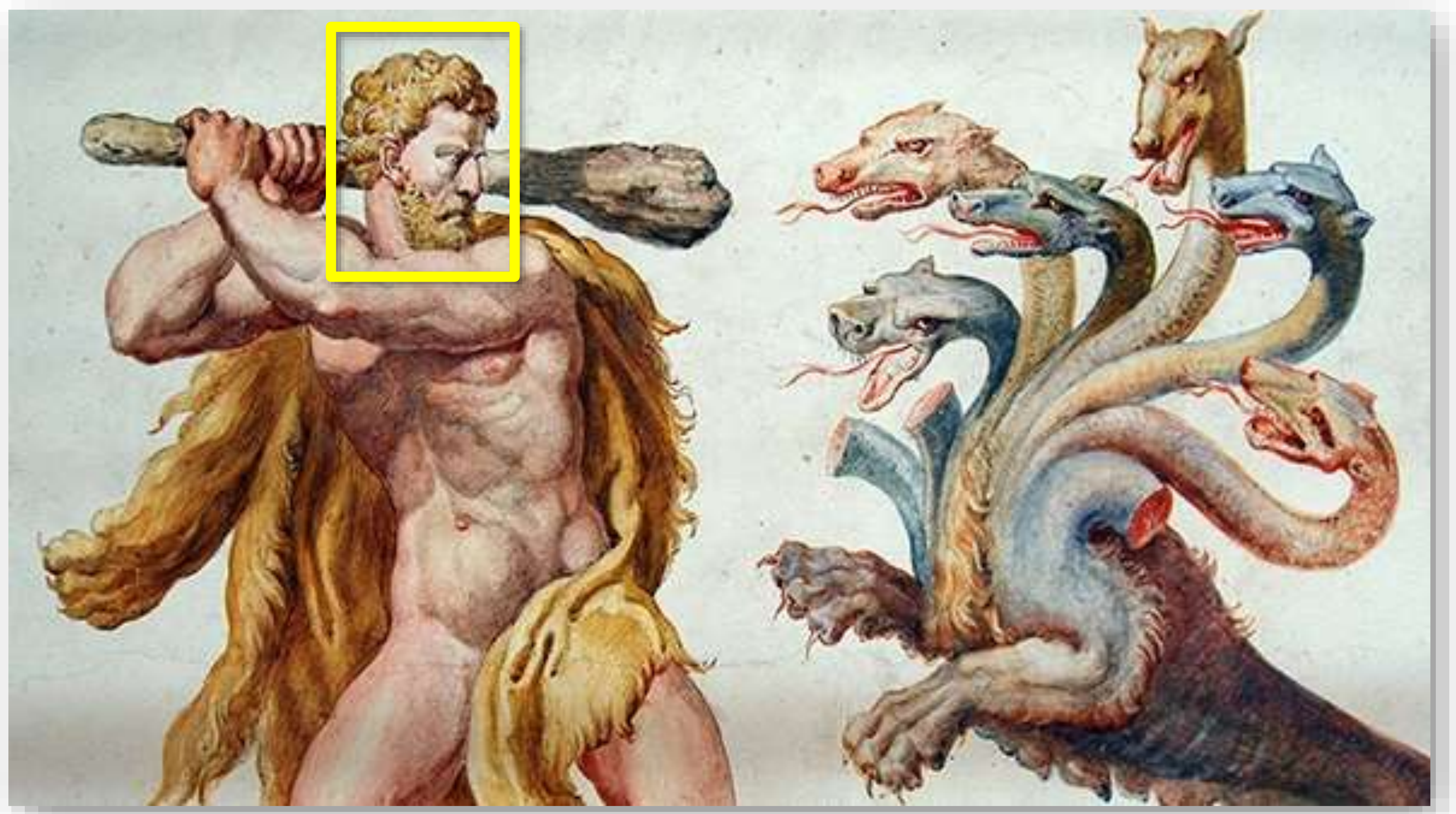
  → Effizienzverlust
  - Synchronisationsoverhead und evtl. starke Einschränkung der Parallelität

  → Deadlocks
  - Deadlocks stellen zyklische (→„ewige") Wartebedingungen dar
  - Die Auflösung eines Deadlocks geht nur gewaltsam
    → Abbrechen von Prozessen, Gefahr von Datenverlust etc.
  - Daher möglichst das Auftreten von Deadlocks präventiv ausschliessen

---

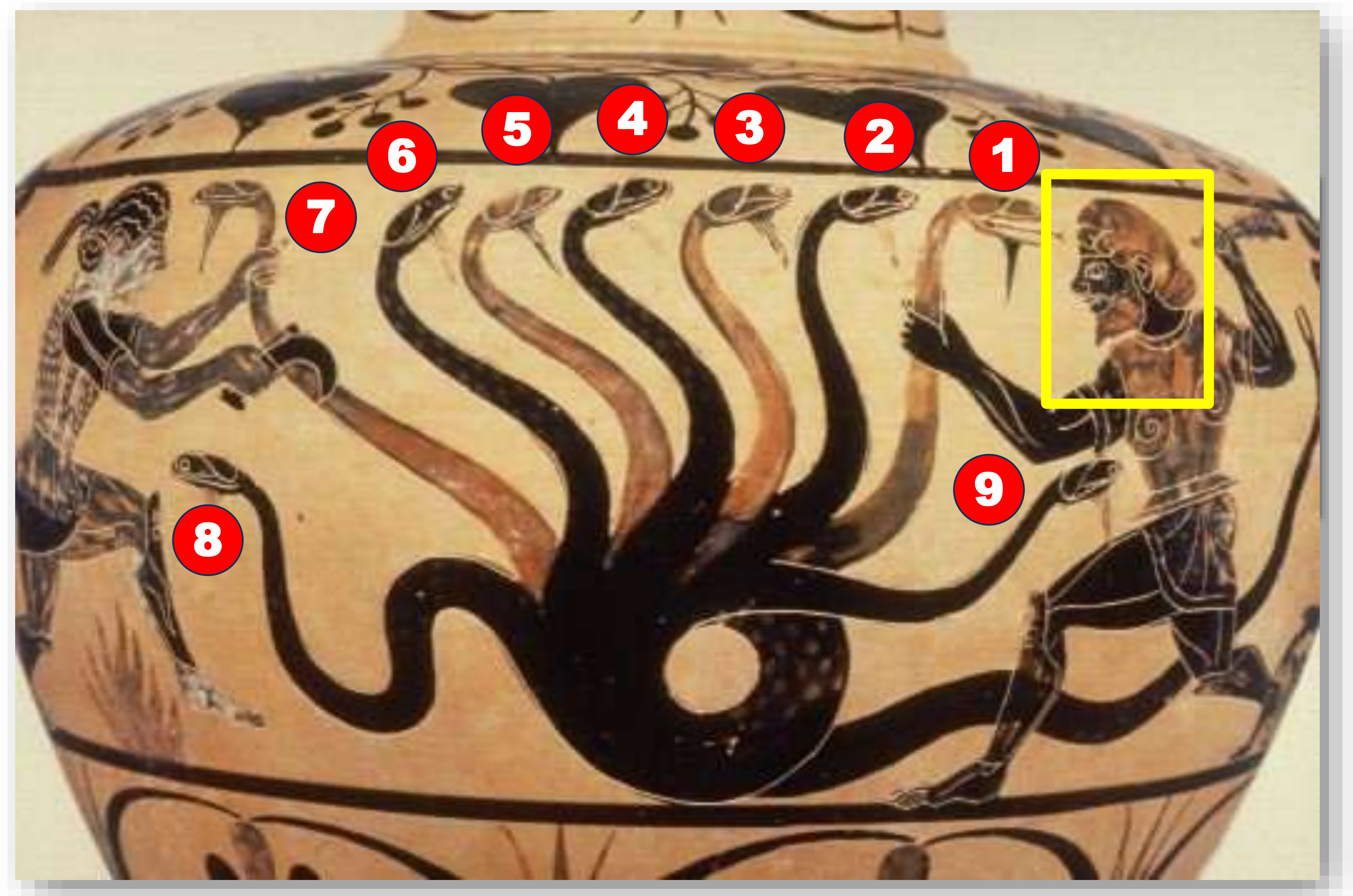
Der richtige Umgang mit Parallelität und die Beherrschung der damit verbundenen Phänomene ist eine Herausforderung!

Das Ringen mit der Parallelität



Der richtige Umgang mit Parallelität und die Beherrschung der damit verbundenen Phänomene ist eine Herausforderung!

Das Ringen mit der Parallelität - eine Herkulesaufgabe!



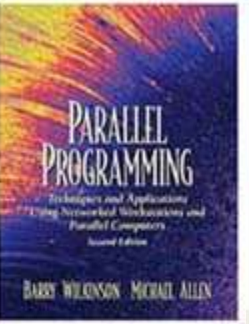
Das Ringen mit der Parallelität – vor 2000 Jahren

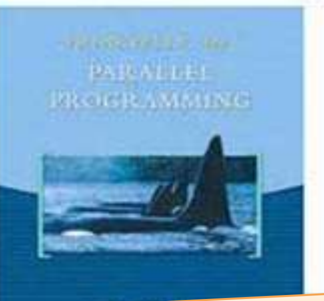
Das Ringen mit der Parallelität

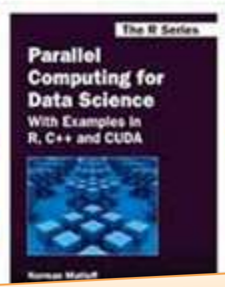
Das Bezwingen der Parallelität

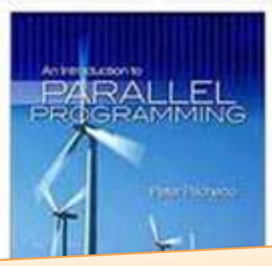
Der richtige Umgang mit Parallelität und die Beherrschung der damit verbundenen Phänomene ist eine Herausforderung!

Zahlreiche Lehrbücher → Das Thema ist relevant, aber keineswegs trivial!

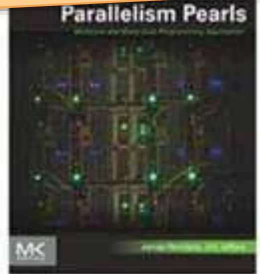


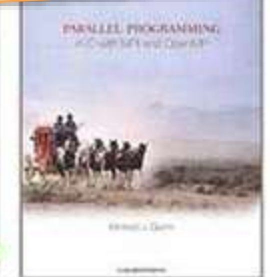
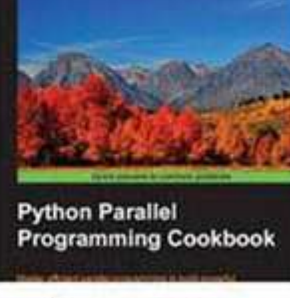


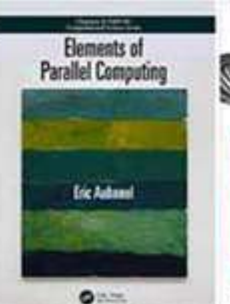


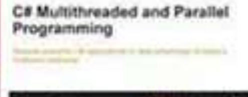


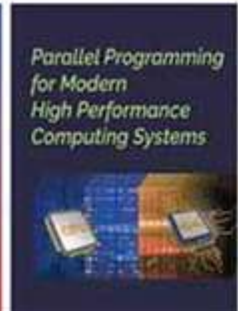
The purpose of this book is to help you program shared memory parallel machines without risking your sanity. – Aus dem Vorwort eines der Bücher

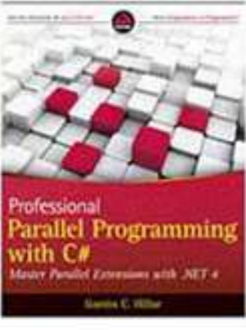










Der richtige Umgang mit Parallelität und die Beherrschung der damit verbundenen Phänomene ist eine Herausforderung!

echte

- Früher mussten nur Spezialisten für Systemsoftware damit umgehen können
- Jetzt erfordern immer mehr „normale" Anwendungen die Verwendung paralleler Prozesse und Threads!

→ Mehr zu diesem Thema in anderen Vorlesungen aus dem Lehrangebot von Informatik und ITET

Resümee des Kapitels

- Prozesse, Multitasking
  - Prozesszustände
  - Prozessverwaltung durch das Betriebssystem
  - Kontextwechsel (→ Pseudoparallelität)

- Java: Parallele Threads
  - Sprachelemente (class „Thread"), Programmierung
  - Beispiel für zwei parallele Threads („Hin-Her")

```
class Hin extends Thread {

  public void run() { ...
     while(true) {  ... paint();

  public void paint() { // Sternchen hinzu
    System.out.print("*"); ...

class Her extends Hin {
  public void paint() { // Sternchen weg
    System.out.print("\b \b"); ...
```

Resümee des Kapitels (2)

- Methoden zur Thread-Steuerung
  - start, stop, suspend, resume, join,…
- Scheduling von Threads
  - Prioritäten
  - Zeitscheiben
- Race conditions
- Atomarität
  - Inkonsistenzen bei Nicht-Atomarität
- Kritische Abschnitte
  - Safety, Liveness, Fairness
  - Realisierung mit dem Synchronized-Konstrukt von Java
- Deadlocks