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 Personen quasi-gleichzeitig, im Timesharing-Betrieb, zu bedienen. Dies war ökonomisch sinnvoll, da die Rechner sehr teuer waren und die diversen Anwendungen viele Rechenpausen enthielten (E/A-Zugriff, Warten auf Benutzereingabe etc.), die produktiv für die Anwendungen Anderer verwendet werden konnten. Einige frühe Betriebssysteme, insbesondere Unix, waren auch schlank genug, um eine grössere Zahl pseudoparalleler Prozesse zu unterstützen, sodass Systeme aus kooperierenden Prozessen realisiert werden konnten und man 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, sodass 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

When executed on real hardware, programs are phenomena of the real world. At the same time, their written forms are formal descriptions of these phenomena. – Reino Kurki-Suonio

- Prozess = Programm in Ausführung
  - „Vorgang einer algorithmisch ablaufenden Informationsverarbeitung"
  - „Instanz" eines Programms
  - Programm selbst ist eine leblose Folge von Anweisungen in einer Datei

Vergleiche mit: → „Objekte sind Instanzen einer Klasse"

- Es können gleichzeitig mehrere Prozesse als verschiedene Instanzen des selben Programms existieren
  - Z.B. mehrere aktive Web-Browser in einem Display-Fenster
- Der Kontext (vereinfacht: „Zustand") eines Prozesses umfasst u.a.:
  - Aktuelle Stelle der Programmausführung („Befehlszähler")
  - Inhalt der CPU-Register
  - Werte aller Variablen (abgelegt in Speicherzellen)
  - Inhalt des Laufzeitstacks (dynamische Aufrufsequenz)
  - Zustand zugehöriger Betriebsmittel (z.B. geöffnete Dateien)

= individuelle dynamische Daten pro Instanz zu einem Zeitpunkt

Idee: Prozess einfrieren, Kontext wegspeichern, später reanimieren

In der Zwischenzeit die CPU etwas anderes animieren lassen

Prozessverwaltung und Betriebsmittel

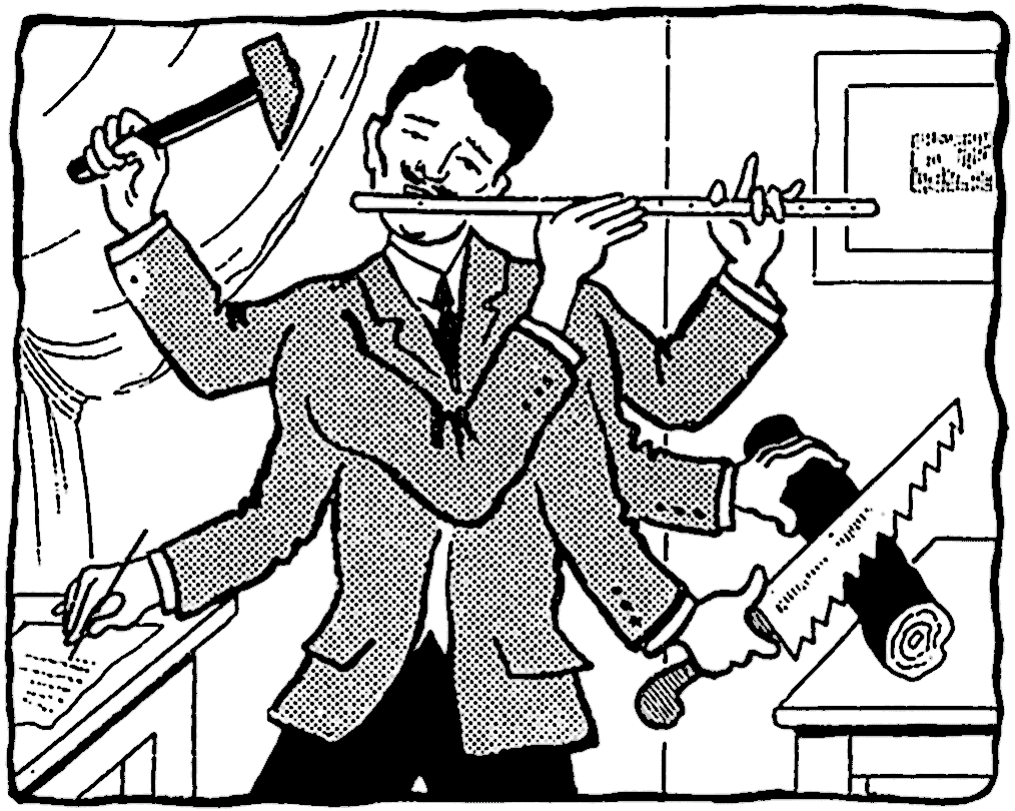
- Ein Prozess benötigt Betriebsmittel („Ressourcen“)
  - CPU-Zeit, Speicher (RAM), Dateien, Energie,...
  - Manche Ressourcen sind unteilbar / exklusiv und begehrt / kostbar
  - Prozesse konkurrieren um diese Betriebsmittel
- Nicht nur die Ressourcen, sondern auch die Prozesse selbst werden durch das 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 Komponente des Betriebssystems)
  - Vermittlung von Kommunikation zwischen den Prozessen (z.B. Signale, Ereignisse oder Nachrichten)
  - Mechanismen zur Synchronisation
  - Managen der Prozess-Konkurrenz

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

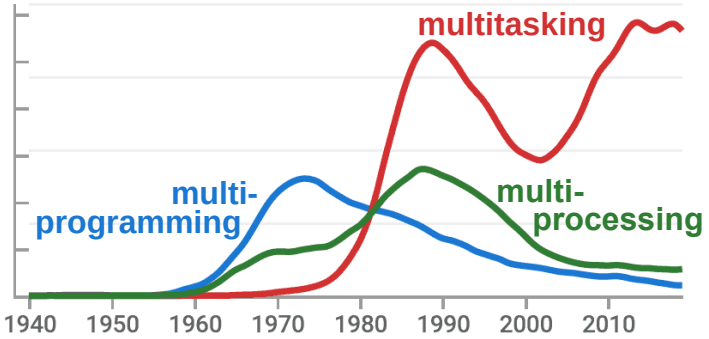
Multitasking

"I multitask every single second I am online. At this very moment, I am watching TV, checking my email every two minutes, reading a newsgroup about who shot JFK, burning some music to a CD and writing this message." – 17-year-old boy, 2001

- Mehrere Aufgaben (tasks) quasi-gleichzeitig ausführen



Das Phänomen scheint schon etwas älter zu sein, wie man am Stil des Bildes erkennt…

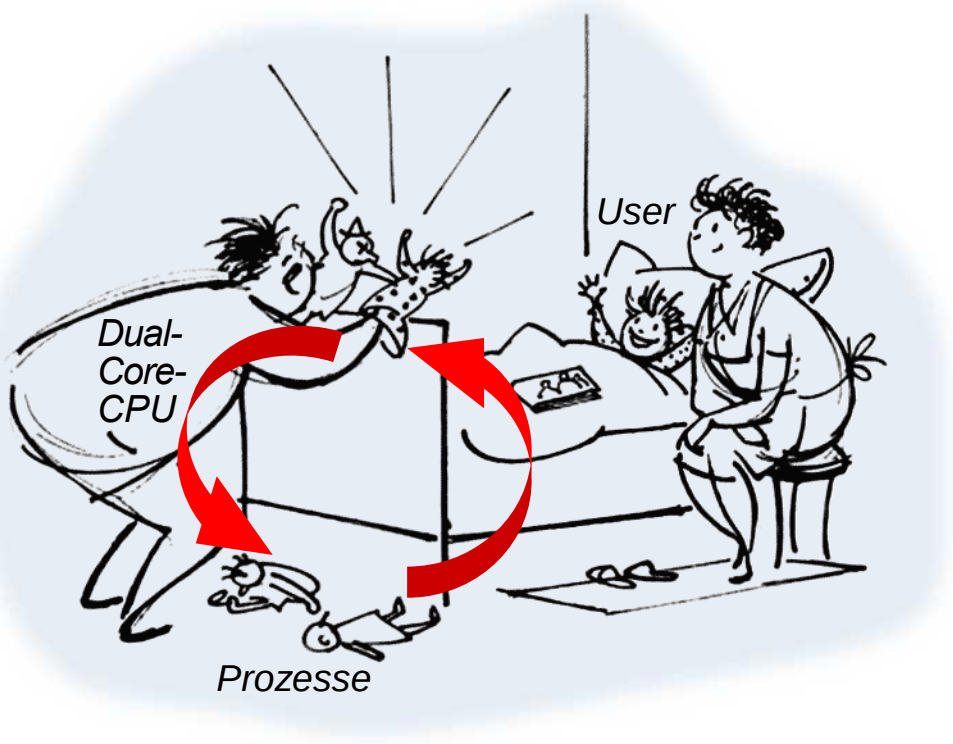


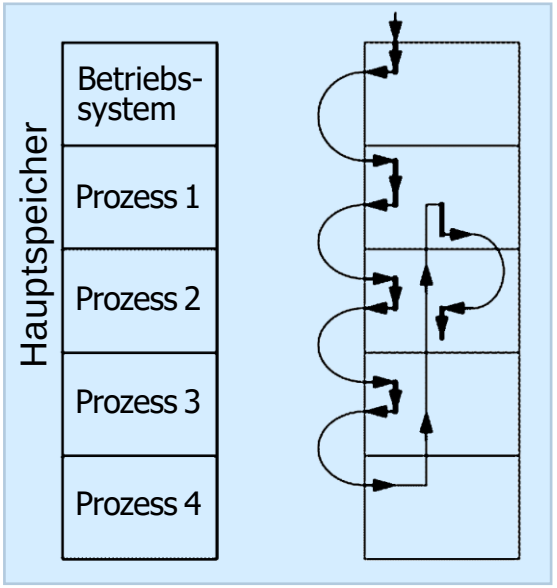
Das Wort „multitasking" wurde erstmalig 1965 verwendet, und zwar in einer Dokumentation zu einem Betriebssystem der Firma IBM: „IBM Operating System/360 – Concepts and Facilities". Es wird dort zunächst „multiprogramming" beschrieben als „general term that expresses use of the computing system to fulfill two or more different requirements concurrently" und dann „multitasking" so eingeführt: „…to express parallel processing not only of many programs, but also of a single reenterable program used by many tasks ". Ab der Jahrtausendwende wurde das Wort auch in der Psychologie für Phänomene verwendet, die bis dahin unter Begriffen wie „divided attention" oder „dual-task performance" (bzw. sperrig-deutsch „Mehrfachaufgabenperformanz") diskutiert wurden und fand sehr schnell Einzug in den populären Wortschatz.

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)





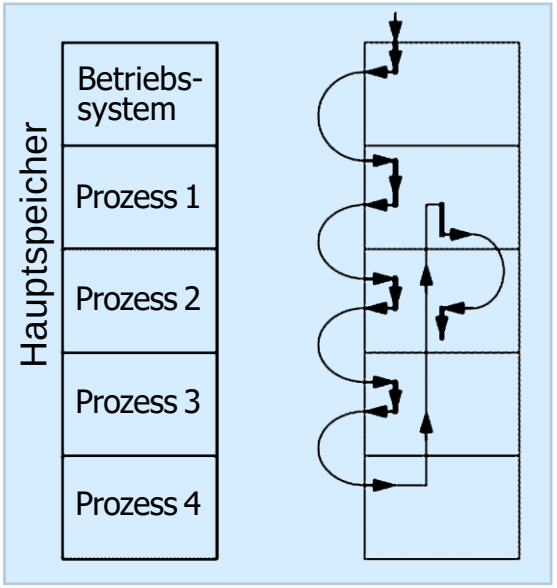
Prozesse in so kurzen Abständen immer abwechselnd aktivieren, dass der Eindruck der Gleichzeitigkeit entsteht

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)



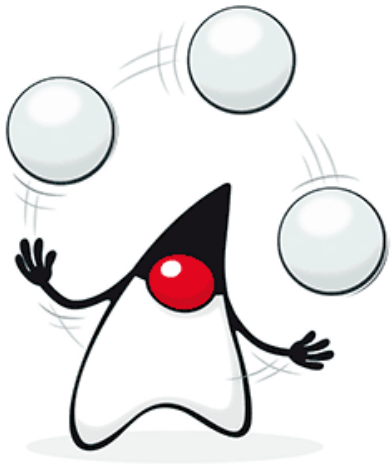
Schnelles Hin- und Herschalten geht dann gut, wenn sich mehrere Prozesse bereits im Hauptspeicher befinden (und nicht erst jedes Mal geladen werden müssen)

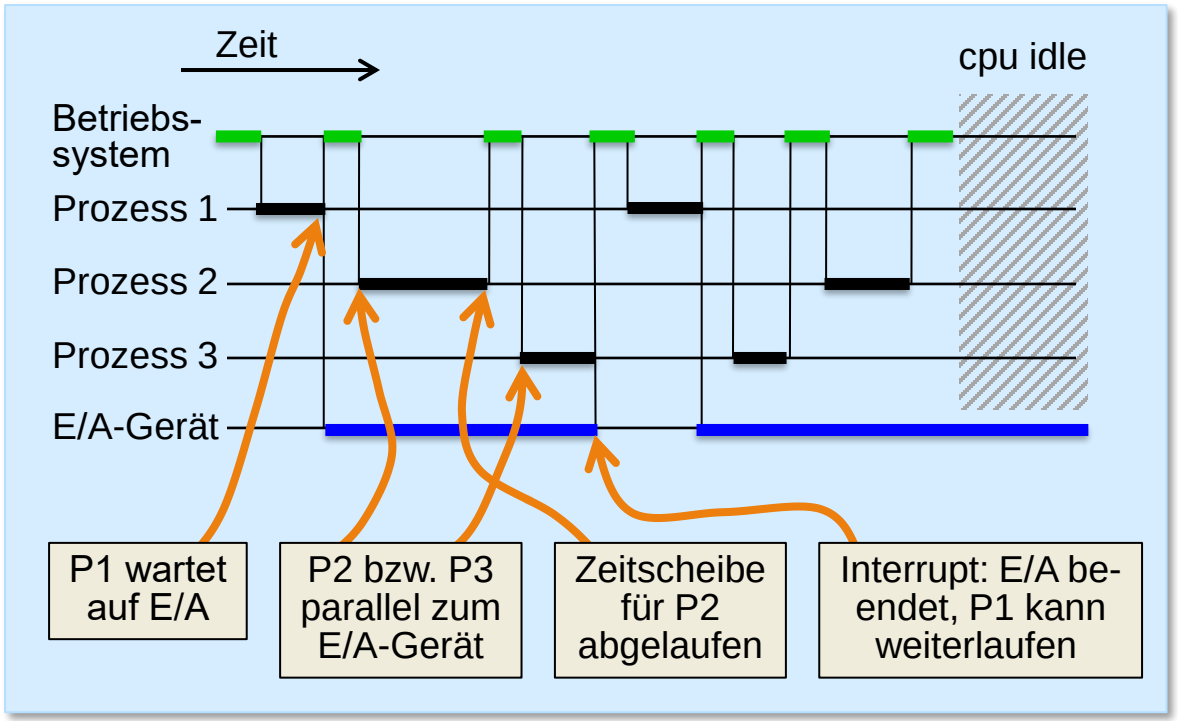
- Zweck:
  1) Gleichzeitiges Steuern / Überwachen mehrerer Abläufe der realen Welt
  2) Bessere CPU-Nutzung
     - Während ein Prozess auf externe Ereignisse wartet (z.B. Daten vom Netz oder einem angeschlossenen Gerät), kann ein anderer ausgeführt werden
     - Beispiel: 10 ms Wartezeit → ca. 10^{10}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

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

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

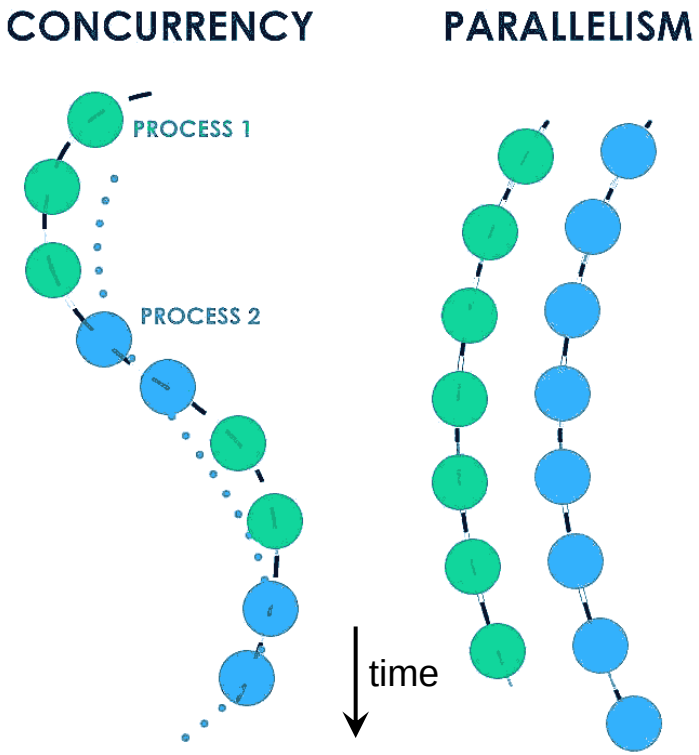
→ Optimierungsziel: möglichst alle Geräte (und die CPU) ständig beschäftigen und einen möglichst hohen Überlappungsgrad erzielen





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

- Sprechweise und Modellierung sind leider nicht ganz einheitlich: Oft (jedoch nicht immer!) wird im Englischen die Pseudoparallelität als „concurrency“ bezeichnet, während „parallelism“ als Begriff für eine “echt gleichzeitige” Ausführung reserviert ist, 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 along-side each other at the same time.*

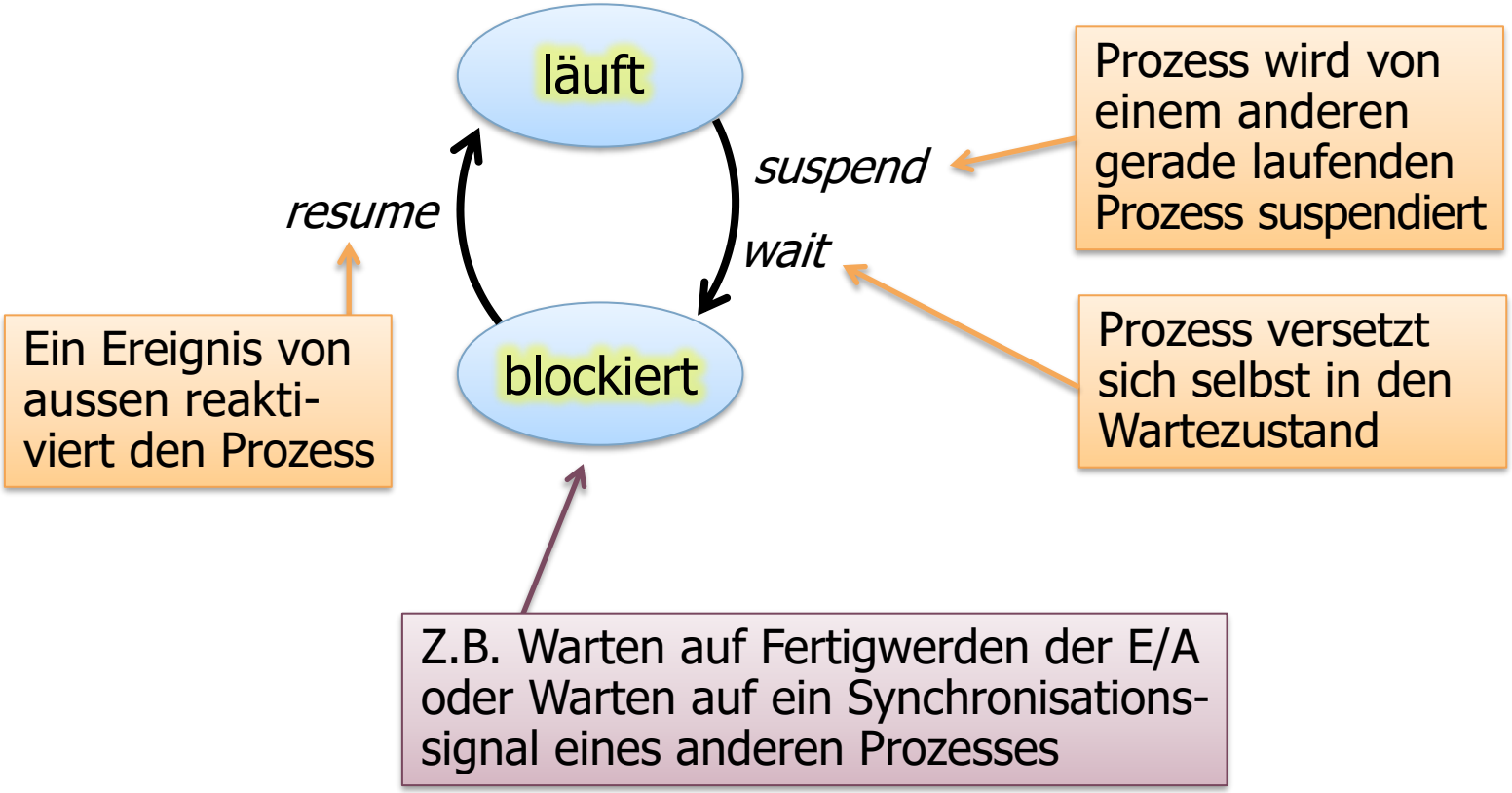


Zitat und Bildquelle: https://medium.com/platformer-blog/node-js-concurrency-with-async-await-and-promises-b4c4ae8f4510

Man beachte auch folgendes häufig anzutreffende Begriffsverständnis [Wikipedia]: „Der Begriff Multitasking bzw. Mehrprozessbetrieb bezeichnet die Fähigkeit eines Betriebssystems, mehrere Aufgaben (quasi-)nebenläufig auszuführen. Besitzt ein Computer mehrere CPU-Kerne, sodass er mehrere Aufgaben ‚echt‘ gleichzeitig ausführen kann, so spricht man von Multiprocessing.“

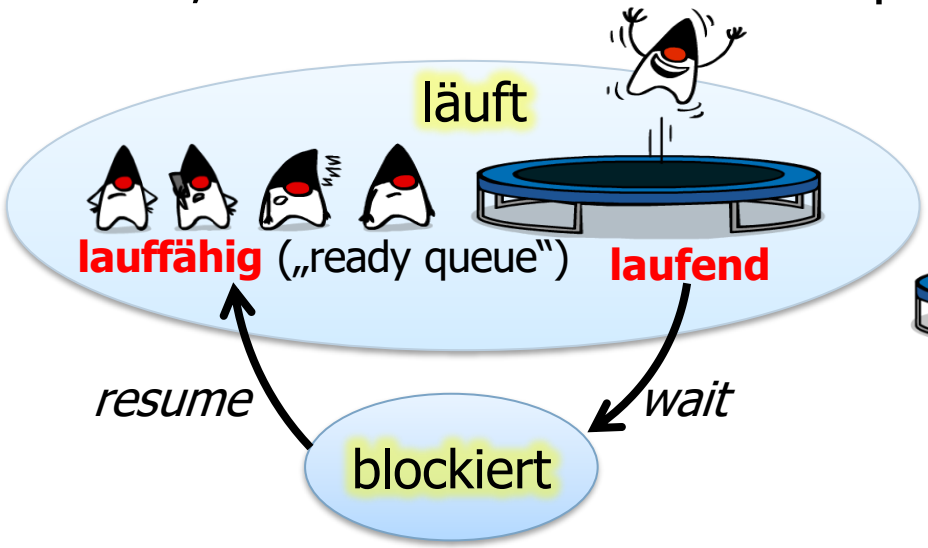
Prozesszustände

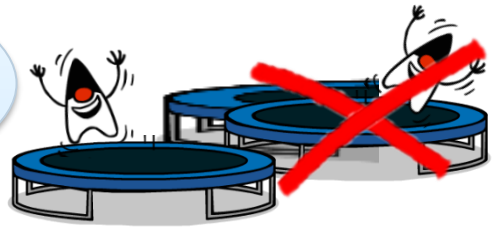
- Von aussen gesehen, kann ein Prozess in zwei Zuständen sein:



Prozesszustände: Verfeinerung

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



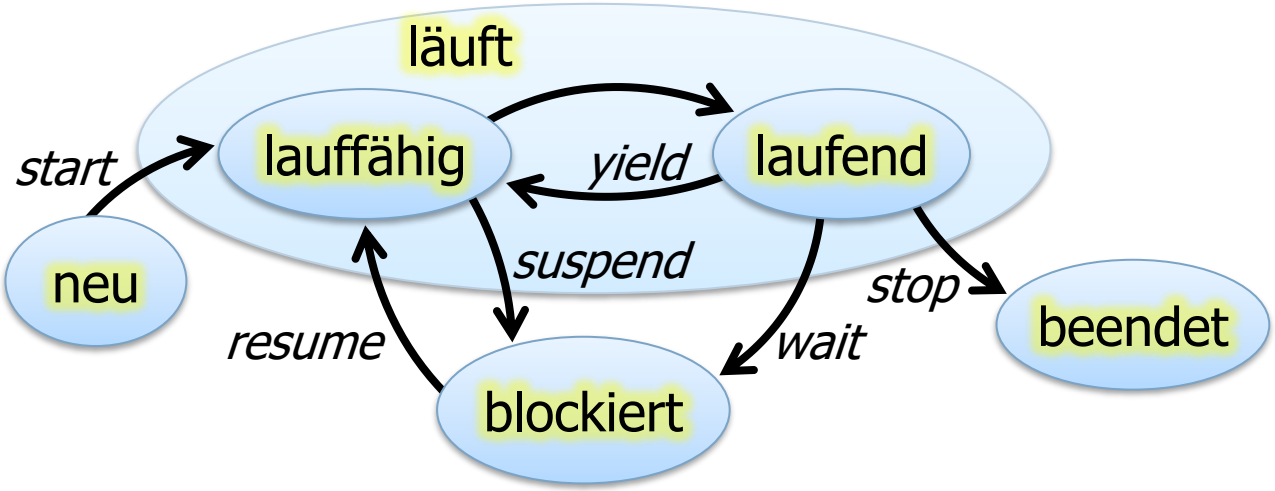


Multicore-Architekturen mit mehreren CPUs betrachten wir hier nicht

- 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

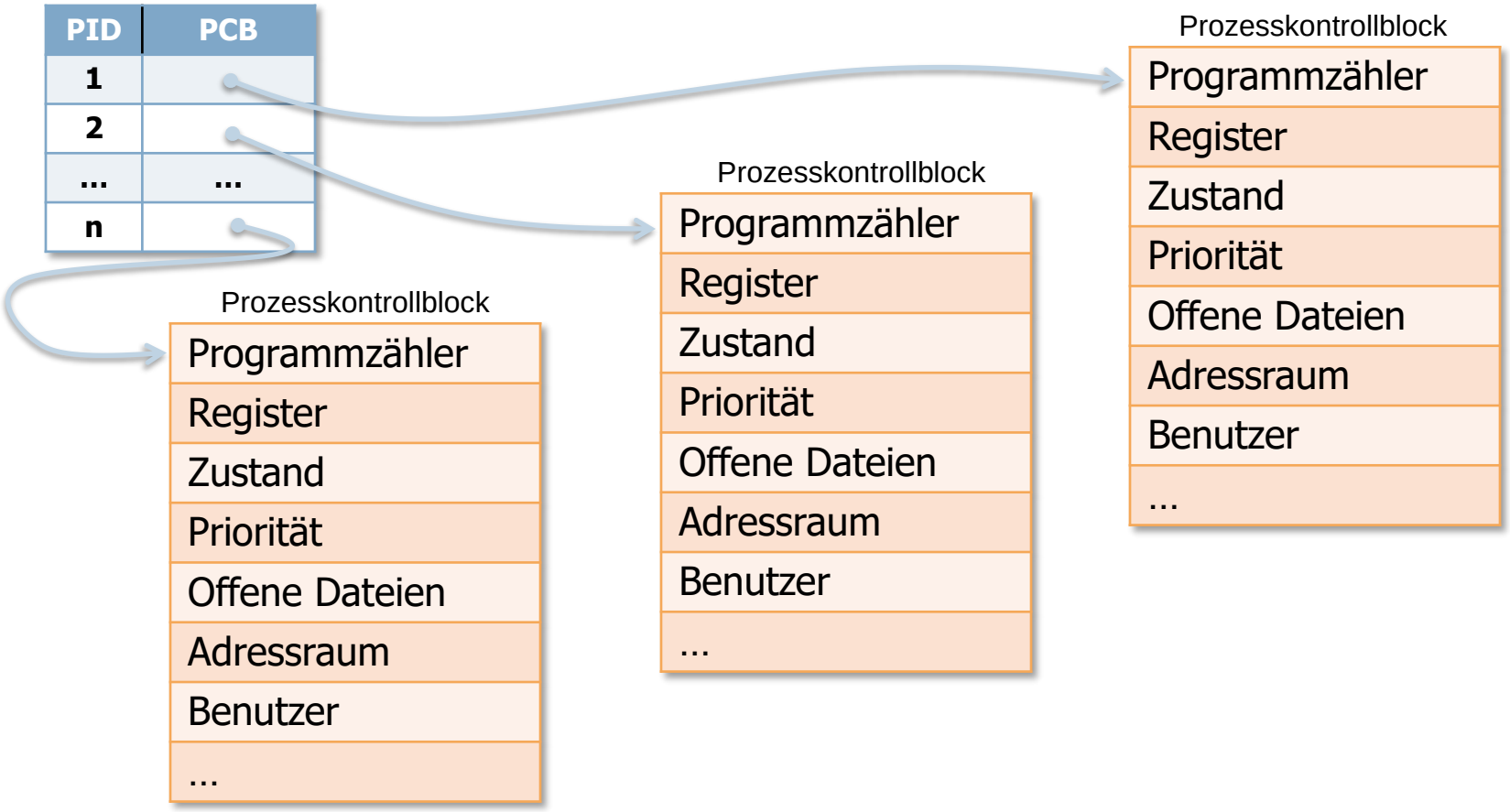
- Prozesse werden durch das Betriebssystem verwaltet
  - Multiplexen der CPU („Scheduling“)
  - Überwachung des Ressourcenverbrauchs

Vom Lateinischen ***schedula***, Diminutiv von *scheda* (bzw. *scida,* altgriech. *σχέδη*) für einen abgespalteten Streifen der Papyrusstaude (vgl. auch *Schisma*, Kirchenspaltung), woraus auch über das mittellateinische *cedula* und das mittelhochdeutsche *zedele* schliesslich das deutsche Wort Zettel entstand. Vom 14. bis zum 17. Jh. war in England das Substantiv schedule für ein „slip of paper“ bzw. eine kurze Notiz gebräuchlich. Der franz. Begriff *cédule hypothécaire* bedeutet Schuld- bzw. Pfandbrief.

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

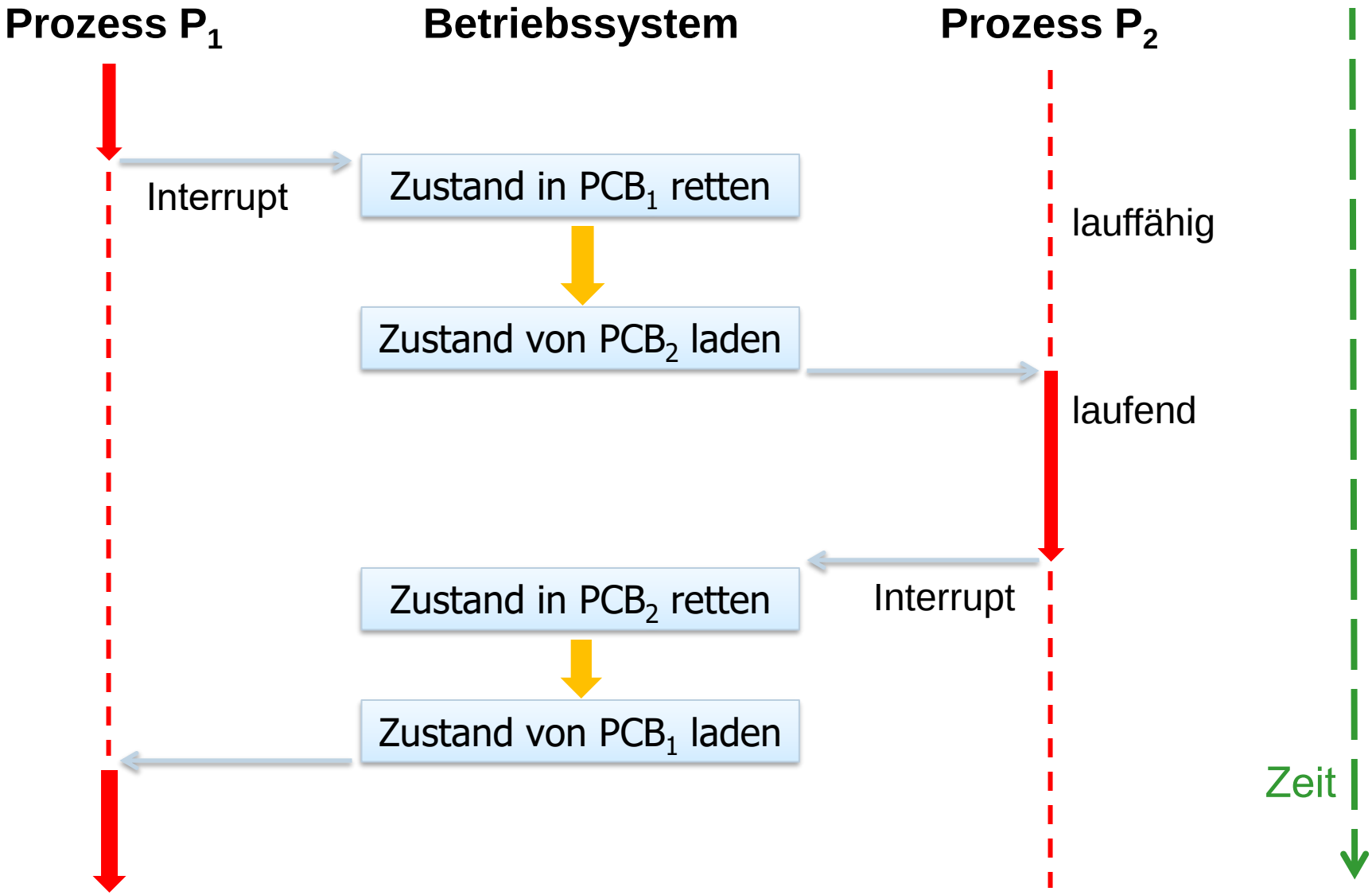
- Wird der laufende Prozess unterbrochen, muss der aktuelle Kontext des Prozesses gesichert werden; hierzu dienen diverse Felder im Prozesskontrollblock

- Der Prozesskontrollblock enthält u.a.:
  - …
  - Programmzähler
  - Inhalt der Register
  - …

} (wenn nicht laufend)

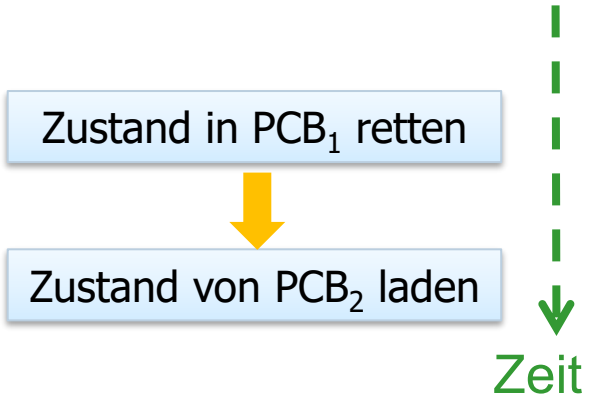
Wenn der Prozess wieder laufend wird, lädt das Betriebssystem dies in die CPU zurück

Kontextwechsel



Kosten eines Kontextwechsels

- Der zu sichernde Prozesszustand ist recht umfangreich

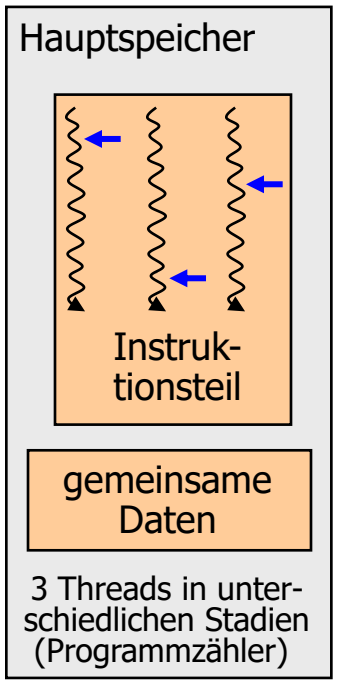


- Hinzu kommen diverse Verwaltungsaktionen, die das Betriebssystem bei einem Kontextwechsel durchführt
  - Speicherbereich des neuen Prozesses vor Zugriffen anderer abschotten
  - Zugriffsberechtigung auf Ressourcen prüfen
  - …
- Kontextwechsel ist daher relativ teuer
  - Kostet typischerweise einige zehntausend Instruktionen →
  - Man kann sich nicht allzu viele Prozesswechsel pro Sekunde erlauben

Leichtgewichtsprozesse (Threads)

Dadurch anfällig gegenüber Programmierfehlern!

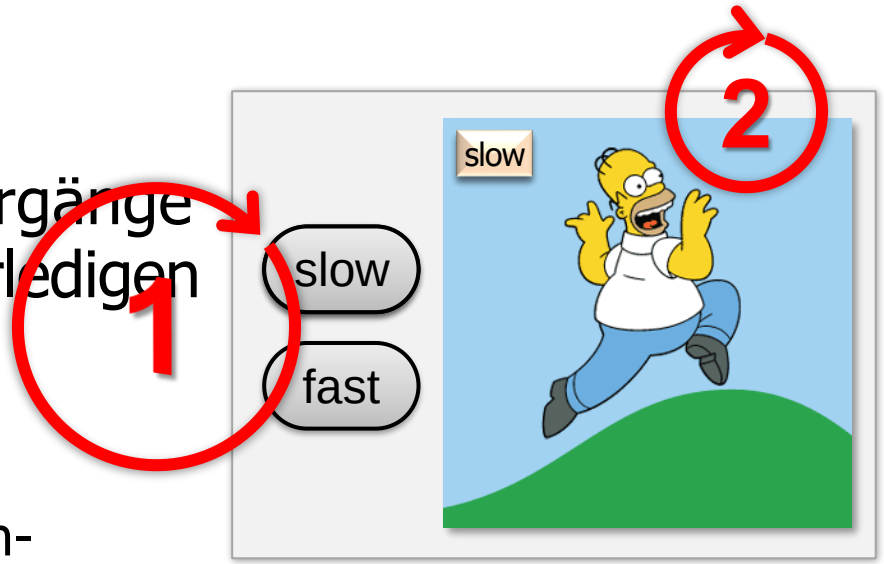
- Threads („of control“) sind (pseudo)parallele Aktivitätsträger, die nicht gegeneinander abgeschottet sind
  - Laufen innerhalb eines gemeinsamen Adressraumes
  - Teilen sich gemeinsame Ressourcen
- Kontextwechsel zwischen den Threads ist viel effizienter als Kontextwechsel zwischen Prozessen
  - Kein Adressraumwechsel
  - Oft kein aufwändiges / automatisches Scheduling
  - Kein Retten / Restaurieren des Kontextes (Ausnahme: Register, Programmzähler etc. analog zu Methodenaufruf)



- 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 beiden input buttons
    2. Animation des Cartoons



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 oft ineffizient

Klasse java.lang.Thread

Hier nur ein Auszug; weiter Aspekte → Java-Dokumentation

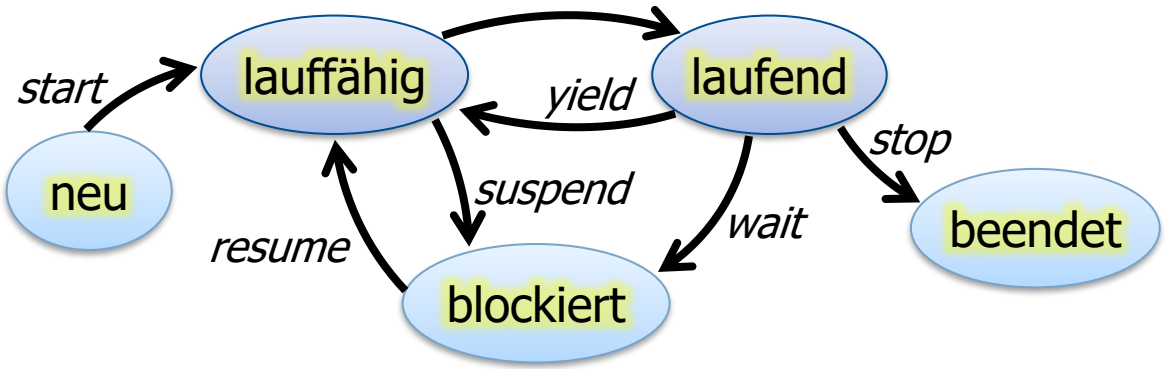
*Was in Java „Thread" genannt wird, ist **nicht leichtgewichtig**; Java-Threads entsprechen eher Prozessen. Mit den „**virtual threads**" besitzen neuere Java-Versionen aber auch leichtgewichtige Threads.*

- Methoden:

```
- void start()
- void wait()
- void yield()
```

```
- void stop()
- void suspend()
- void resume()
```

Diese Methoden nicht verwenden!




```
- 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

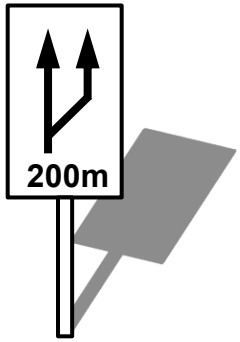
„Hintergrundprozess": terminiert *nicht* mit dem Erzeuger

Programmstruktur eines Threads

- Jeder Thread (genauer: jede von „Thread“ abgeleitete Klasse) muss eine void-Methode run() enthalten
  - Diese macht die eigentlichen Anweisungen des Threads 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
}
```

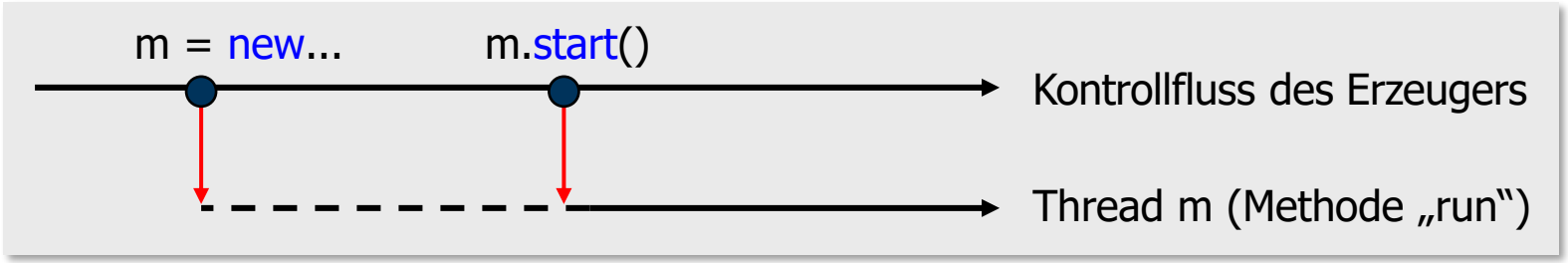
Erzeugen eines Thread
(aus einem anderen Thread heraus)




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

Damit kann man auf den Thread zugreifen und ihn „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();
```

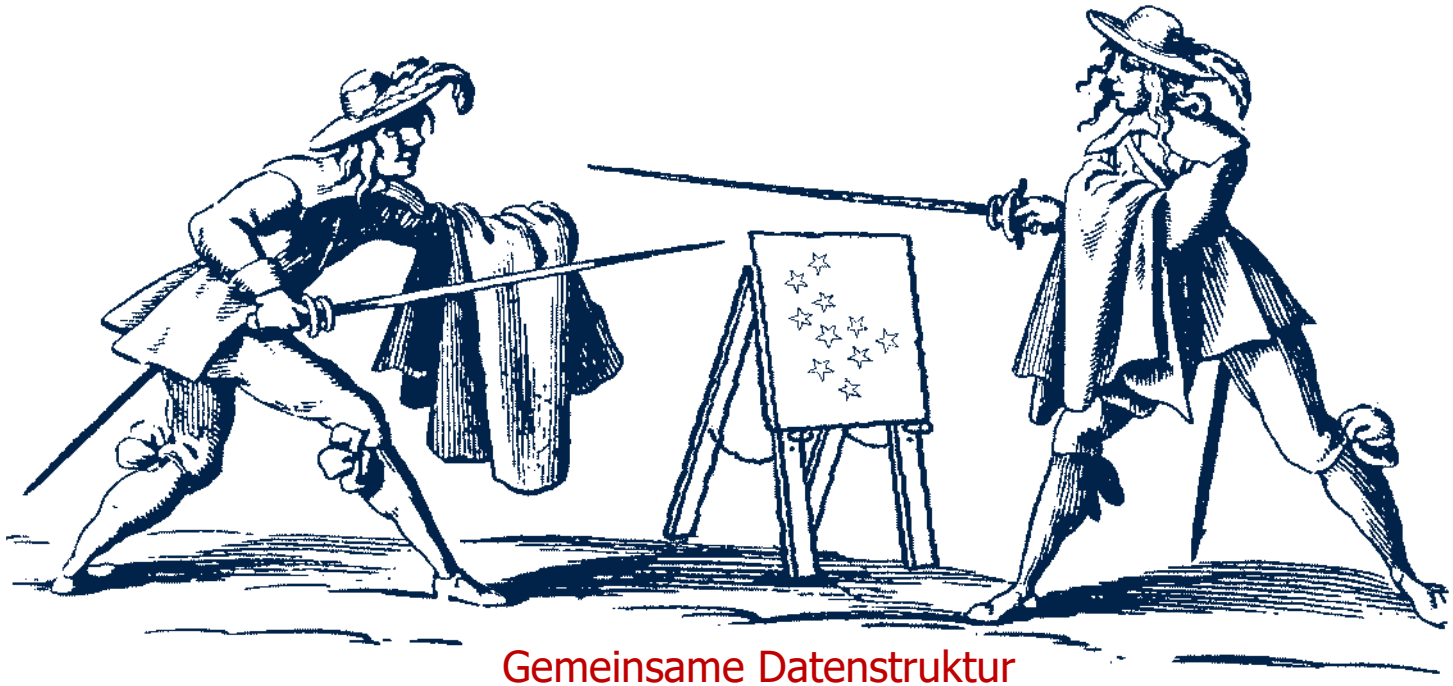
Wie wäre es, wenn „start“ direkt im Konstruktor von Beispiel_Thread stattfände?

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

Ein Thread-Beispiel („hin und her“)

Zwei Seelen wohnen,
ach! in meiner Brust.
– Goethe, Faust I, 1808

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



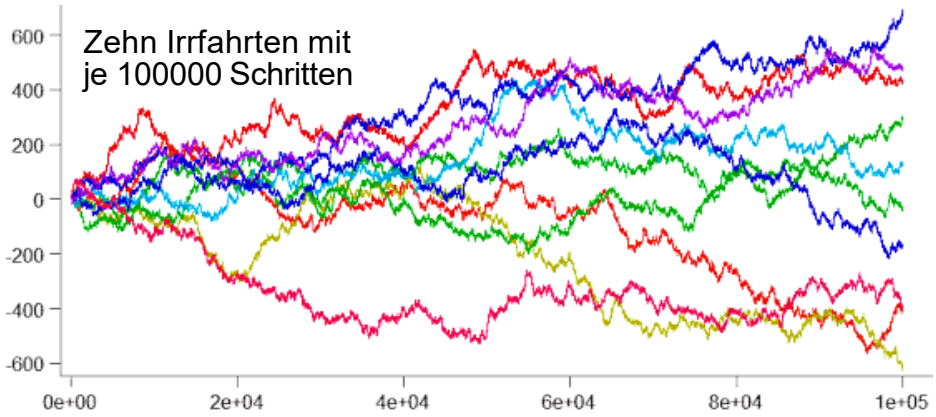
Gemeinsame Datenstruktur

Ein Thread-Beispiel („hin und her“)

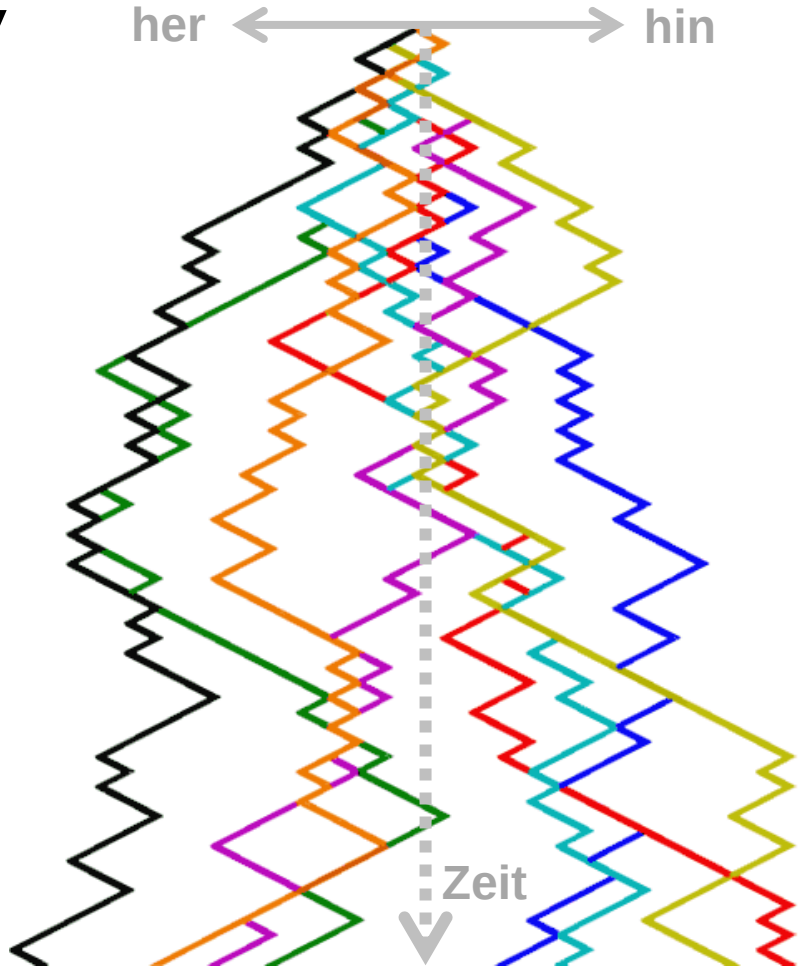
- **********

- Ein Thread „Hin“ schreibt Sterne; ein 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.



Acht verschiedene „Random Walks“ bei der symmetrischen einfachen Irrfahrt



Ein Thread-Beispiel („hin und her“)

- **********

- Ein Thread „Hin“ schreibt Sterne;
  ein Thread „Her“ löscht Sterne;
  beide arbeiten (quasi)parallel

```
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“

Kommt es dazu überhaupt noch? Oder behält „Hin“ die ganze Zeit über die Kontrolle?

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 passiert, wenn ein Thread aufwacht, während der andere gerade mitten in „paint" ist?

Zufallszahl zwischen 0.0 und 1.0

Gemacht für die Ewigkeit

Aber liefert das so eine mathem. saubere Irrfahrt?

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")

InterruptedException

In primitiver Weise können Threads über Interrupts „kommunizieren“: Man kann in einem anderen Thread ein Interrupt-Flag setzen, und jeder Thread sollte regelmässig sein Flag prüfen und dann ggf. seine Ausführung abbrechen. Hängt aber ein Thread in einem sleep (oder einem join oder wait), kann er das Flag nicht zügig prüfen. Zu diesem Zweck gibt es die InterruptedException, welche den Thread (im Interrupt-Fall) sofort daraus befreit.

InterruptedException is a confusing beast – it shows up in seemingly innocuous methods like Thread.sleep(), but handling it incorrectly leads to hard-to-manage code that behaves poorly in concurrent environments.

At its most basic, if an InterruptedException is caught it means someone, somewhere, called Thread.interrupt() on the thread your code is currently running in. You might be inclined to say “It's my code! I’ll never interrupt it!” and therefore do something like this:

```
try {
  Thread.sleep(1000);
} catch (InterruptedException e) {
  // disregard
}
```

But this is exactly the wrong way to handle an “impossible” event occurring. If you know your application will never encounter an InterruptedException you should treat such an event as a serious violation of your program’s assumptions and exit as quickly as possible. More often, however, you cannot guarantee that your thread will never be interrupted. […] It’s critical that your code responds promptly to interrupts, otherwise your application will stall or even deadlock.

In such cases the best thing to do is generally to allow the InterruptedException to propagate up the call stack, adding a throws InterruptedException to each method in turn. This may seem kludgy but it’s actually a desirable property – your method’s signatures now indicates to callers that it will respond promptly to interrupts.

Quelle: https://riptutorial.com/java/example/2116/handling-interruptedexception

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:

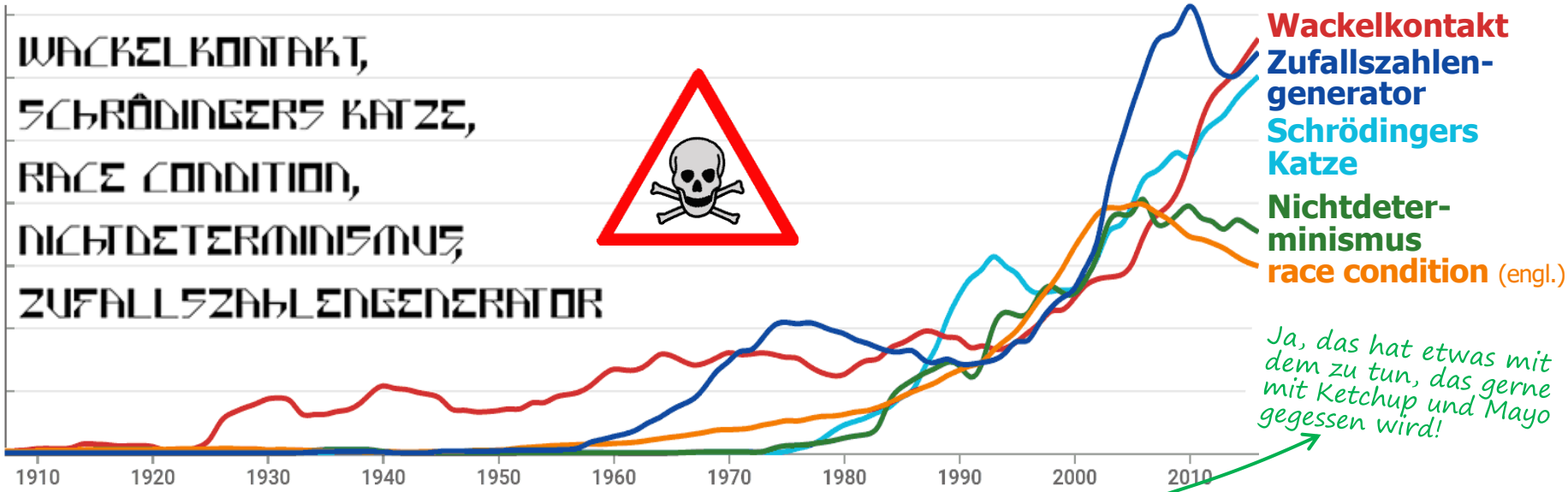
```
i: 400000000   j: 400000000
i: 400000000   j: 400000000
i: 400000000   j: 400000000
i: 400000000   j: 400000000
i: 400000000   j: 400000000
```

- 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 aber ist die Quelle dieses „Zufalls“?
- Und vor allem: Was ist die Wurzel allen Übels?
  (Wir glauben doch nicht ernsthaft an einen „Software-Wackelkontakt“, oder?)

Das ist der Font „Wackelkontakt“!

Hat im Übrigen nichts mit einer On-off-Beziehung zu tun

Der WACKELKONTAKT und seine Kumpanen



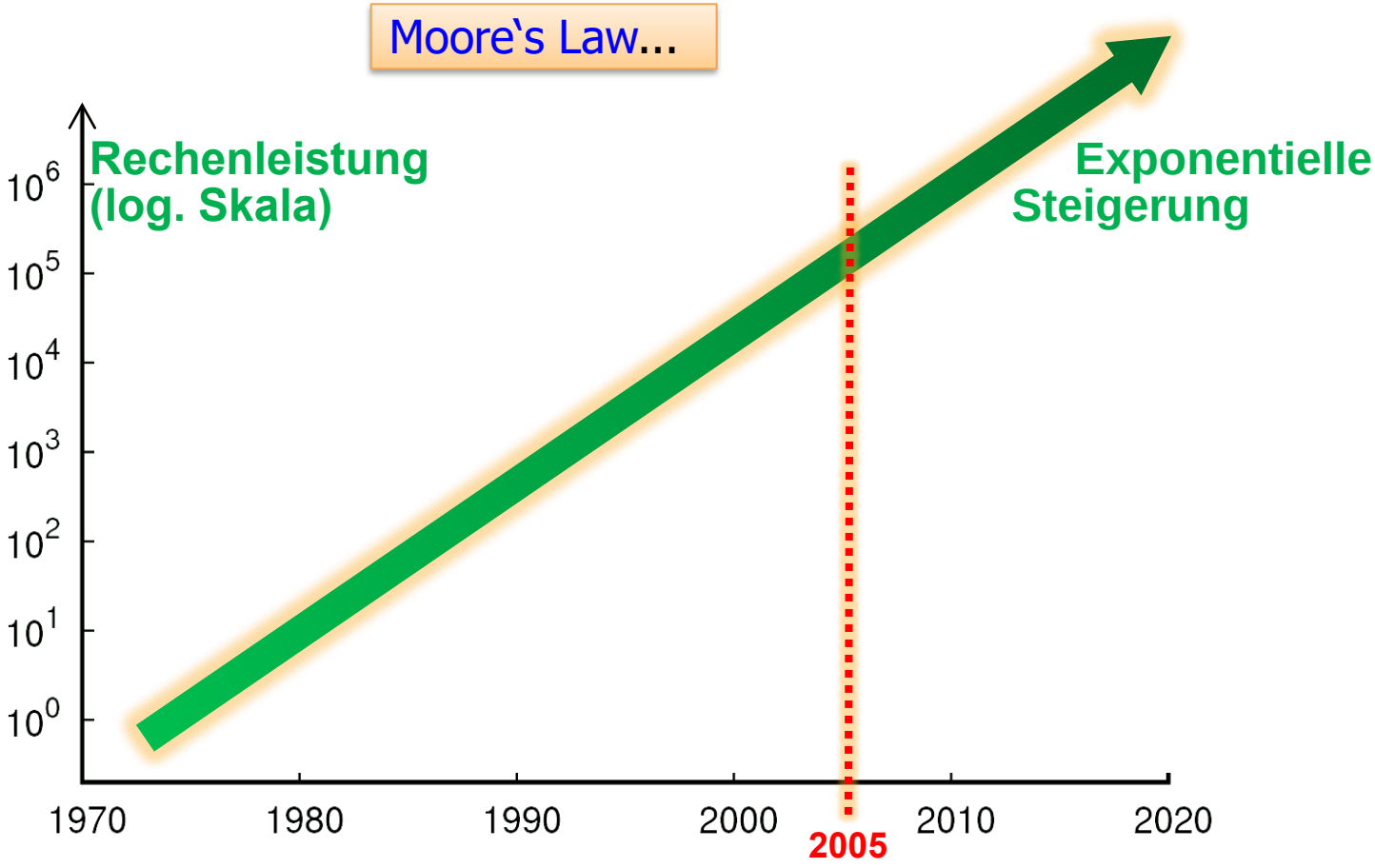
Ja, das hat etwas mit dem zu tun, das gerne mit Ketchup und Mayo gegessen wird!

„So konstruirte Branly einen Empfänger, der in nichts weiterem bestand, als einem Wackelkontakt, einer Röhre, welche mit losem Metallpulver oder Schrauben gefüllt war, den sogenannten Kohärer oder die Frittröhre. Verbindet man diesen Kohärer mit einer Klingel und einem Element in geeigneter Weise und erzeugt mittels einer kleinen Influenzmaschine elektrische Wellen, so ertönt die Klingel, sobald das im Kohärer enthaltene, im gewöhnlichen Zustande schlecht leitende Metallpulver von den Hertz’schen Wellen getroffen wird, durch welche es eine hohe Leitungsfähigkeit annimmt.“ [Zeitschrift der Deutschen Gesellschaft für Mechanik und Optik, 1. August 1901, S. 143]. Gemeint ist hier der französische Physiker und Pionier der Funktechnik Édouard Branly (1844 – 1940), der Wesentliches zur Funktelegraphie und drahtlosen Kommunikation beigetragen hat. 1941 wurde in Paris der quai Branly nach ihm benannt. *Il n'existe pas de nº 69 du quai Branly, la numérotation passant directement du nº 67 bis au nº 71. La municipalité aurait estimé que cela aurait été « un lieu de rendez-vous mal famé »* [Wikipedia]. (Zwar bedeutet „branler“, hier auf nette Art passend, „wackeln“, hat daneben aber noch eine vulgäre Bedeutung.) 2021 wurde der größte Teil des Quai Branly in Quai Jacques Chirac umbenannt.

Parallelität – schwierig, aber...

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

} mühsam

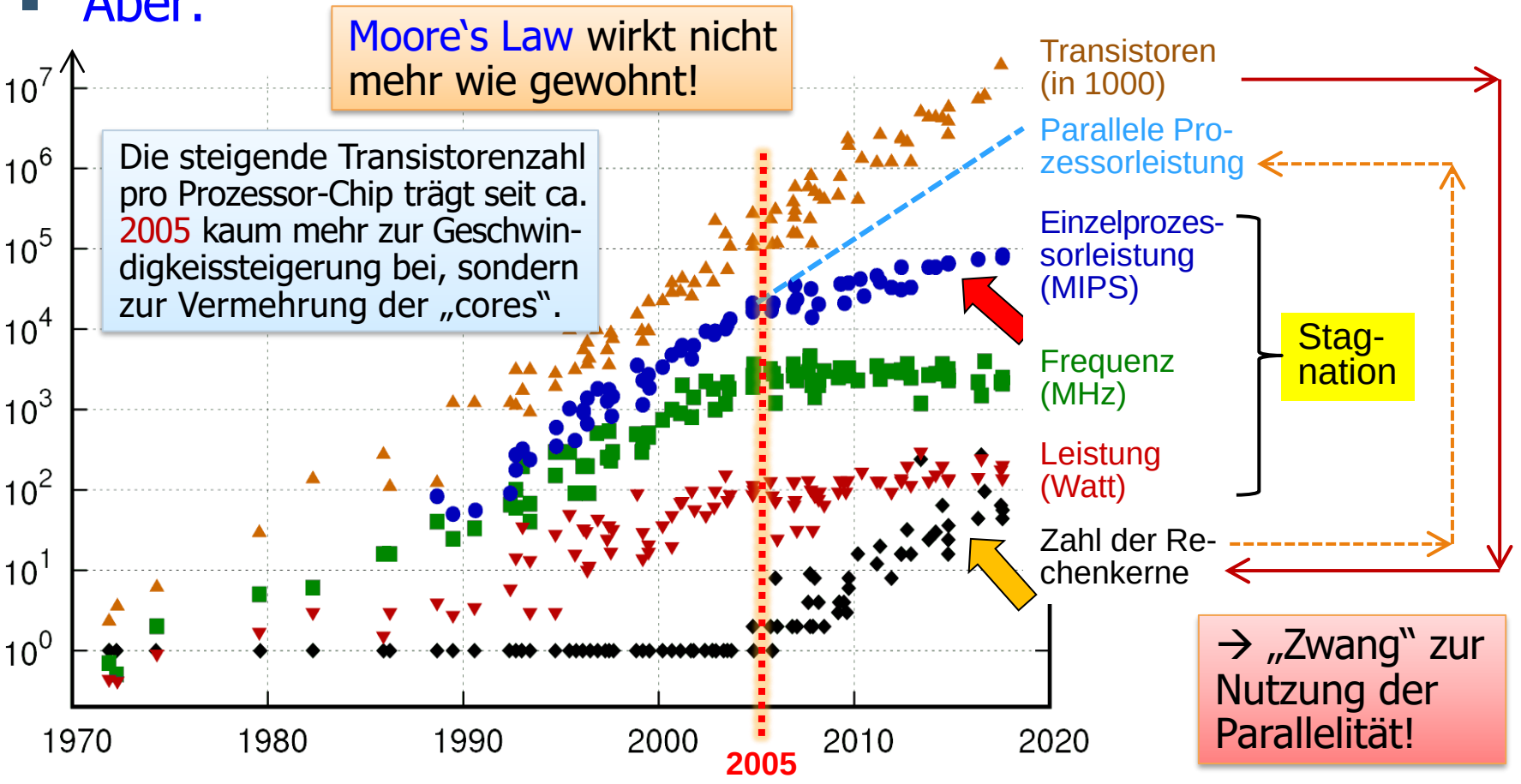


Parallelität – schwierig, aber immer wichtiger

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

} mühsam

- Aber:



www.karlrupp.net/wp-content/uploads/2018/02/42-years-processor-trend.png

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.

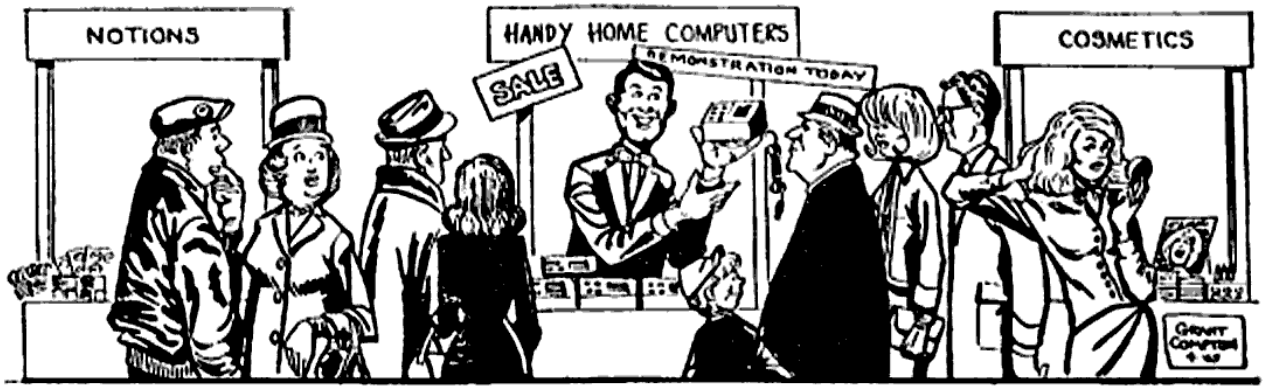
Gordon Moore, ca. 1965

Es begann mit einem Artikel am 19. April 1965, den Gordon Moore (1929 – 2023), späterer Mitgründer und CEO von Intel, im Fachjournal „Electronics“ veröffentlichte.

Will lead to such wonders as home computers

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!

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.



A factor of two per year

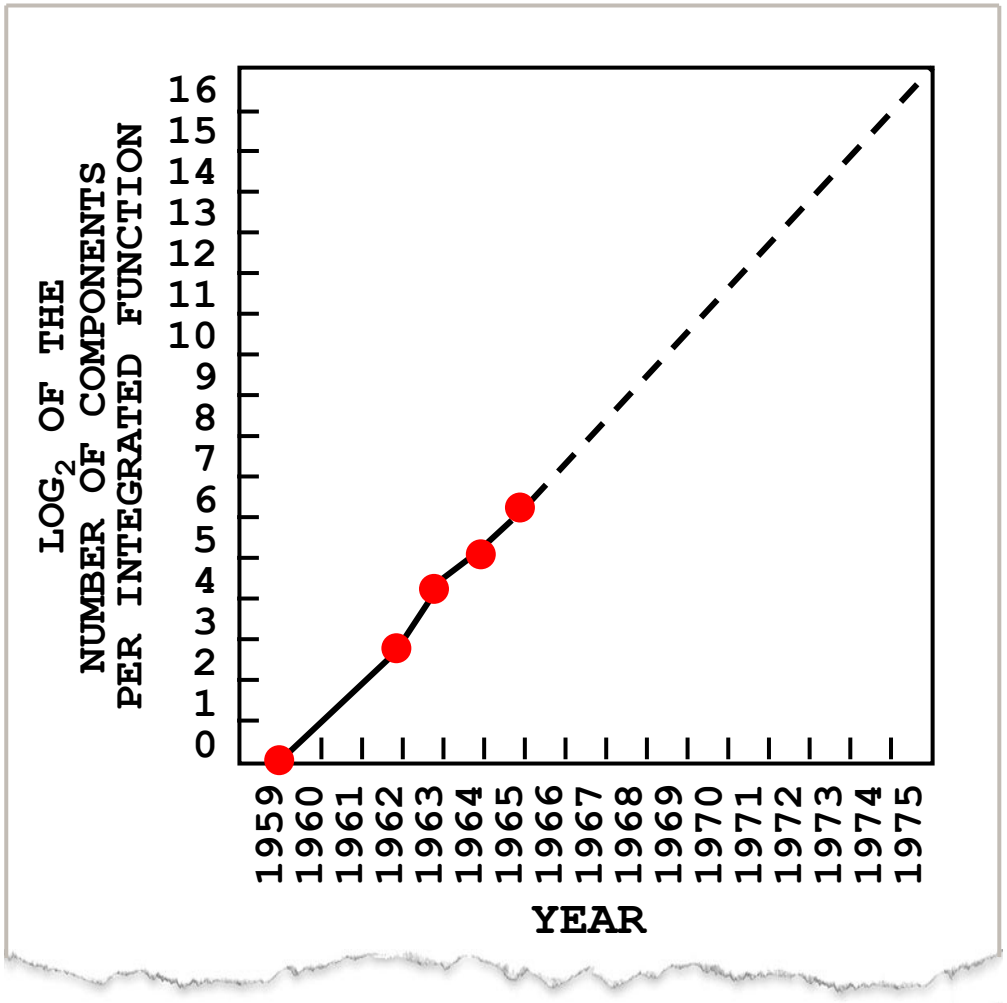
"Computers today are smaller, faster, cheaper, and more reliable, but they are essentially the same machines we had 30 years ago. The important question has been – and continues to be – how we use them." -- Jay Forrester

„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."

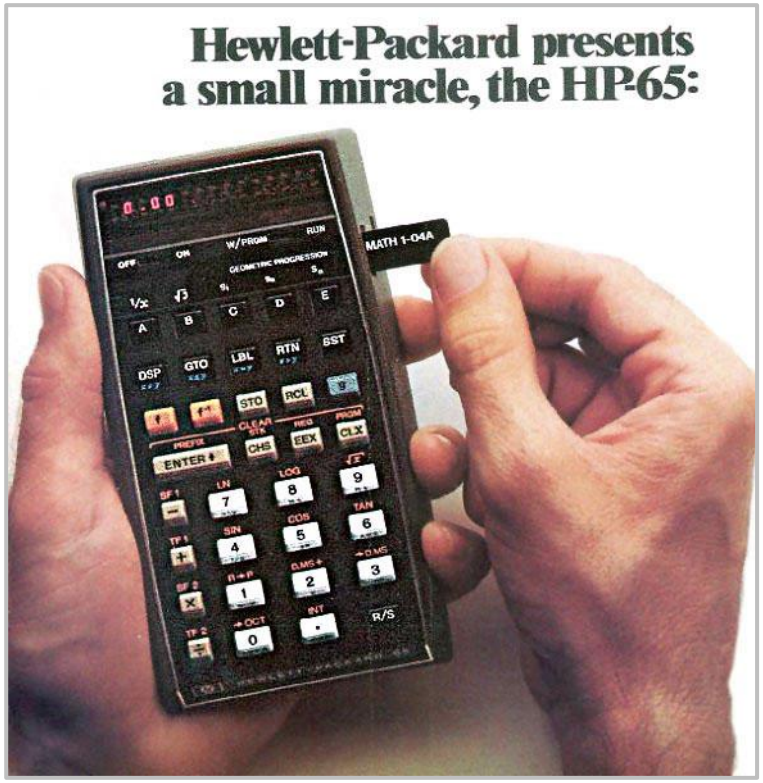
Die empirische Grundlage für das, was ab ca. 1970 dann das mooresche Gesetz genannt wurde, war allerdings etwas dünn: Moore verband in einem halblogarithmischen Diagramm fünf Punkte der Jahre 1959 bis 1965 zu einer geraden Linie und extrapolierte diese 10 Jahre in die Zukunft, bis 1975. Er kam so auf 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: 65000 Transistoren auf einem Chip!

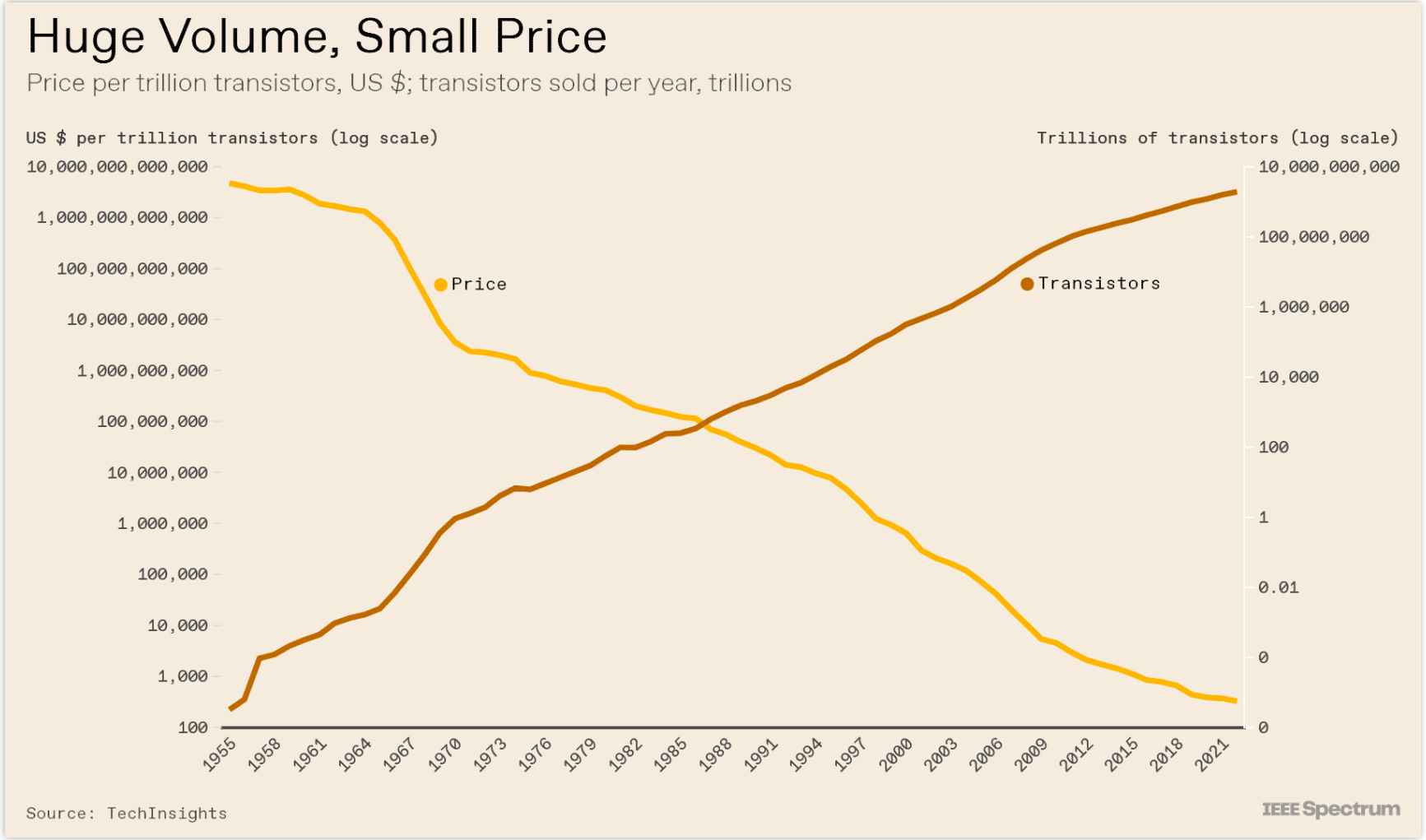


Die Extrapolation in Moores Artikel von 1965.

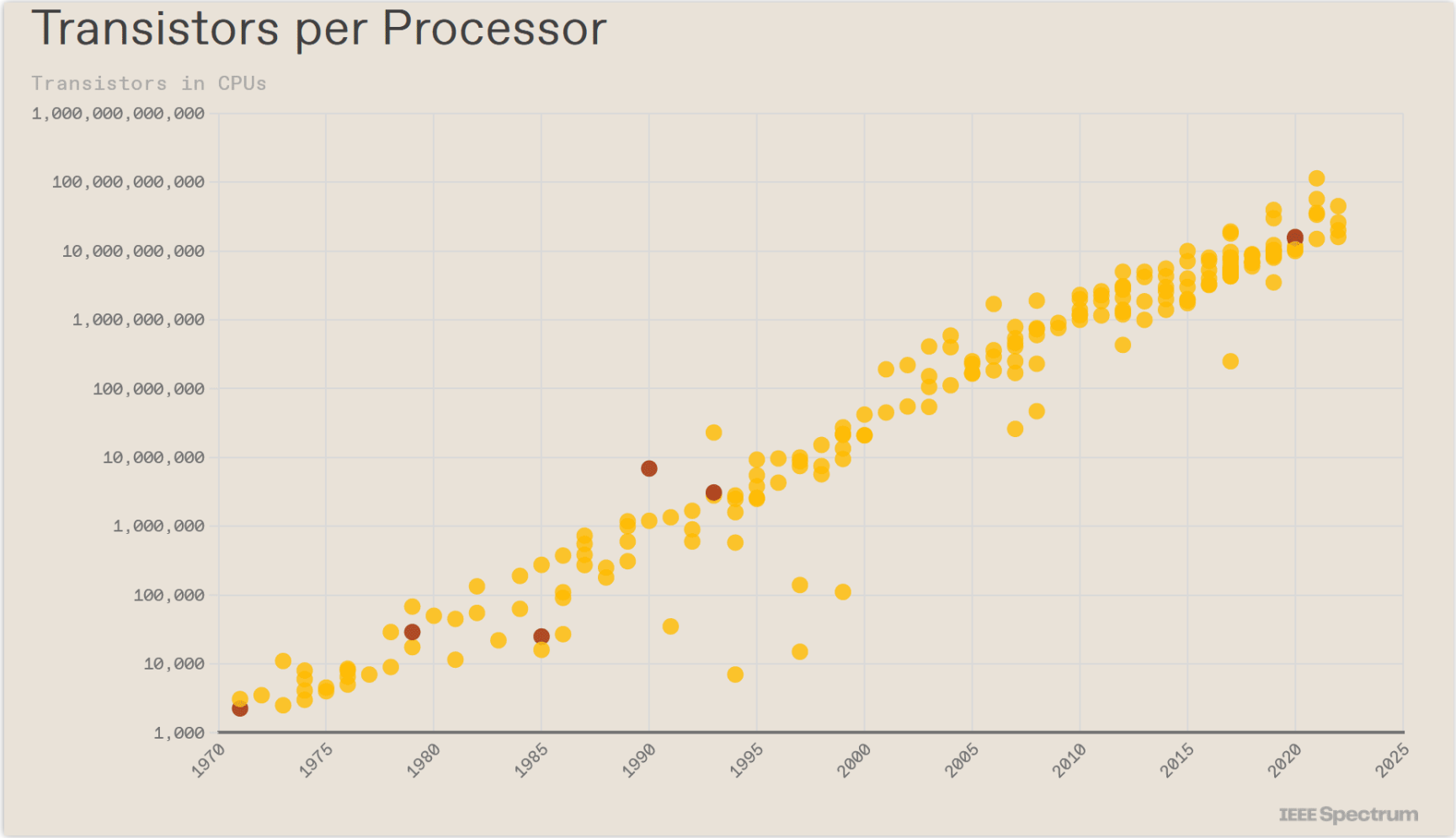


1975: Ein erstes Wunder wird wahr! Damalige Studenten träumten davon und sparten, um sich einen HP-65 leisten zu können; mancher Schüler liess ihn sich auch zum Abschluss der Schule schenken. (Steve Wozniak allerdings verkaufte ihn wieder, um seinen Anteil am Startkapital für die Firma Apple aufzubringen.)

Moore's Law


Huge Volume, Small Price
Price per trillion transistors, US $; transistors sold per year, trillions
US $ per trillion transistors (log scale)
Trillions of transistors (log scale)
10,000,000,000,000
1,000,000,000,000
100,000,000,000
10,000,000,000
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
1,000
100
10,000,000,000
100,000,000
1,000,000
10,000
100
1
0.01
0
0
Price
Transistors
1955
1958
1961
1964
1967
1970
1973
1976
1979
1982
1985
1988
1991
1994
1997
2000
2003
2006
2009
2012
2015
2018
2021
Source: TechInsights
IEEE Spectrum

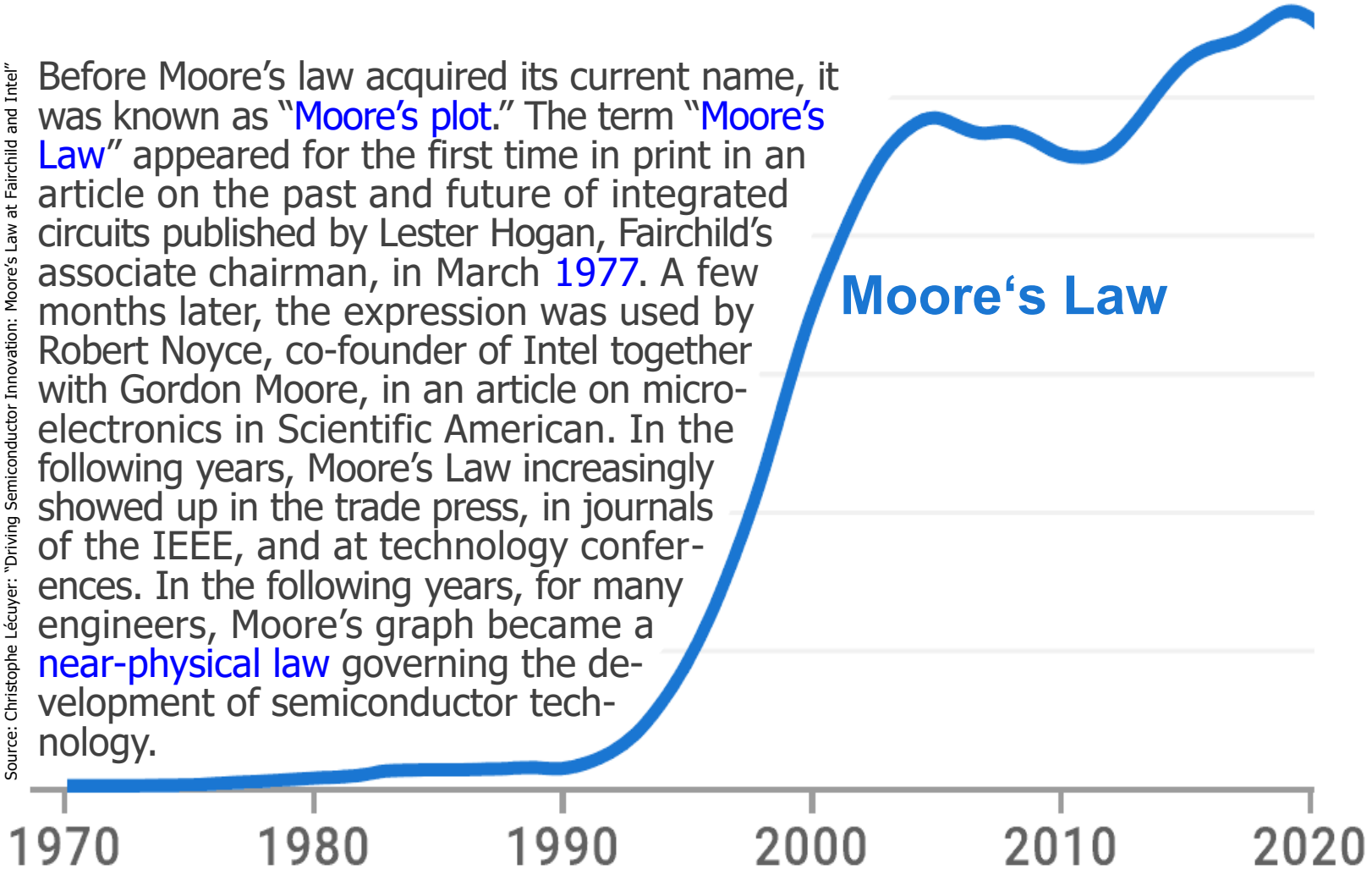
Moore's Law (2)


Transistors per Processor
Transistors in CPUs
1,000,000,000,000
100,000,000,000
10,000,000,000
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
1,000
1970
1975
1980
1985
1990
1995
2000
2005
2010
2015
2020
2025
IEEE Spectrum

Moore's Law (3)

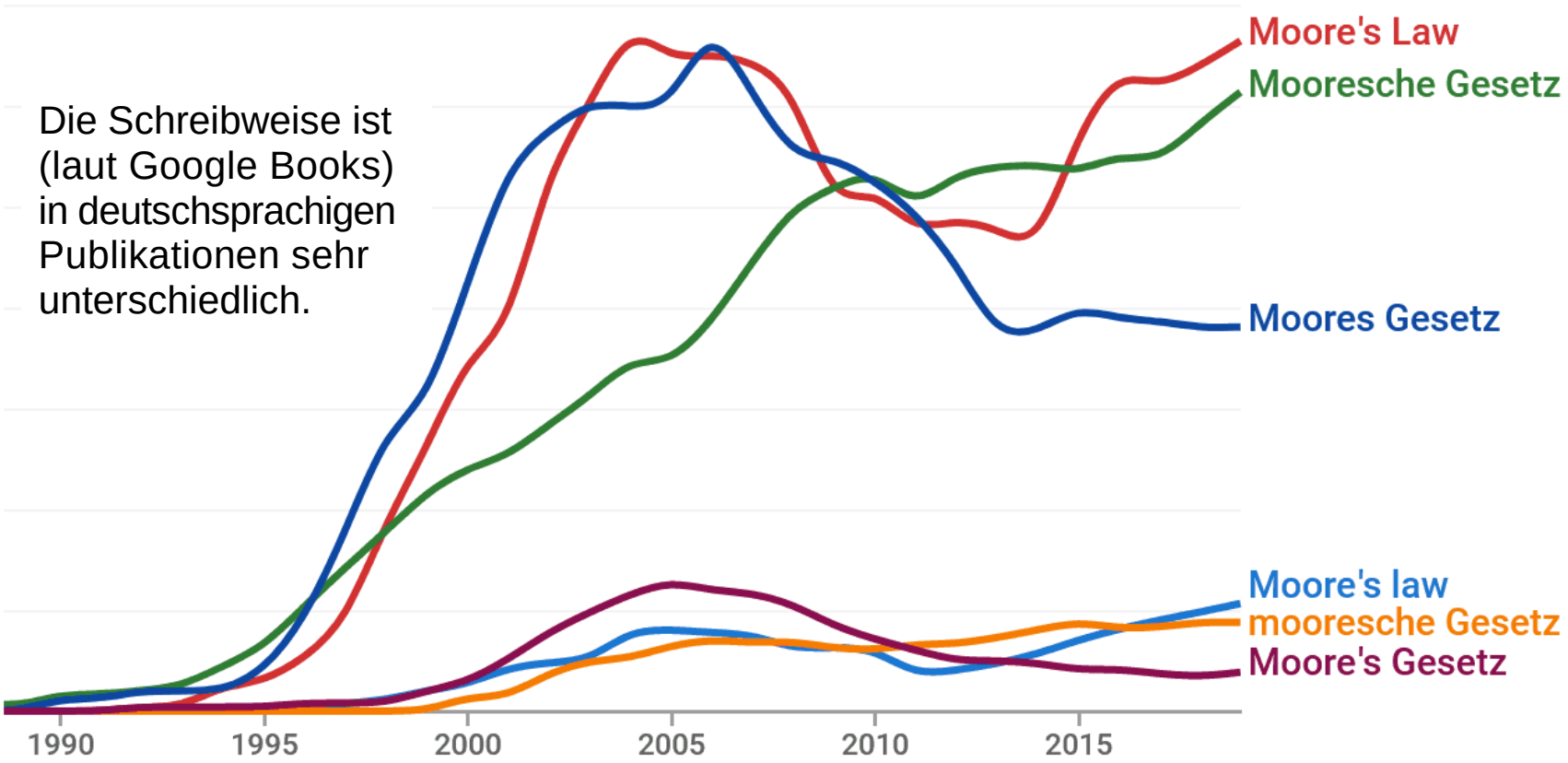
Before Moore's law acquired its current name, it was known as "Moore's plot." The term "Moore's Law" appeared for the first time in print in an article on the past and future of integrated circuits published by Lester Hogan, Fairchild's associate chairman, in March 1977. A few months later, the expression was used by Robert Noyce, co-founder of Intel together with Gordon Moore, in an article on micro-electronics in Scientific American. In the following years, Moore's Law increasingly showed up in the trade press, in journals of the IEEE, and at technology conferences. In the following years, for many engineers, Moore's graph became a near-physical law governing the development of semiconductor technology.

Source: Christophe Lécuyer: "Driving Semiconductor Innovation: Moore's Law at Fairchild and Intel"



Potemkinsche / potemkinsche / Potemkin’sche Dörfer

Die Schreibweise ist (laut Google Books) in deutschsprachigen Publikationen sehr unterschiedlich.



Der hilfreiche und glasklare Duden: Von Personennamen abgeleitete Adjektive werden wie andere Adjektive kleingeschrieben: die einsteinsche Relativitätstheorie, das ohmsche Gesetz. Großgeschrieben werden diese Formen aber dann, wenn die Grundform des Personennamens durch einen Apostroph verdeutlicht wird: die Einstein’sche Relativitätstheorie, die Goethe’schen Dramen. Groß schreibt man aus Personennamen abgeleitete Adjektive als Bestandteile einiger weniger Eigennamen und namenähnlicher Fügungen: der Halleysche Komet, die Magellanschen Wolken. In Zweifelsfällen sind mehrere Möglichkeiten richtig: potemkinsche (oder Potemkinsche oder Potemkin’sche) Dörfer.

Gordon Moore im Kontext

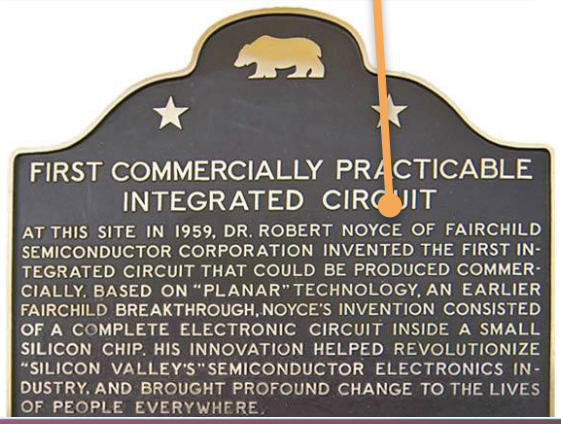
Eugene Kleiner aus Wien (1923 – 2003): 1938 Flucht der Familie vor den Nazis über Belgien und Portugal in die USA; Master in industrial engineering an der New York University; Mitgründer (1972) der venture capital firm Kleiner Perkins (Google, Netscape, Sun, Compaq,…).

Jean Hoerni aus Genf (1924 – 1997): Physik-Doktorat Genf sowie Cambridge (UK), Post-doc Caltech (USA).

https://spectrum.ieee.org/robert-noyce-and-the-tunnel-diode

Gordon Moore (1929 – 2023), Ph.D. in Chemie und Physik am Caltech. Moore war zunächst Mitarbeiter am Shockley Semiconductor Laboratory, der ersten High-Tech-Firma des Silicon Valleys. Mit 7 anderen leitenden Mitarbeitern verliess er 1957 die Firma und gründete Fairchild Semiconductors. Fairchild selbst galt als führend auf dem Gebiet der Halbleiter, und aus der Firma entstand viele Spin-offs, die (neben der Stanford University und mehreren Unternehmen für feinmechanische und elektronische Instrumente sowie Rüstungsunternehmen wie Lockheed Missiles) massgeblich den Ruf des Silicon Valleys begründeten. Auch Gordon Moore und Robert Noyce gründeten 1968 von Fairchild aus eine neue Firma: „Integrated Electronics“, bekannter unter der Kurzform „Intel“.

[Jay Last] “We were all focused on the single goal of producing our first product... We were all very young, 27 to 32... We were a very compatible group, and spent a lot of time outside our working hours. Most of the founders were married, busy starting their families and raising small children in addition to all the time and effort they were spending building Fairchild.”



The engineer as a rich guy

Robert Noyce coinvented the integrated circuit. The other coinventor was Jack Kilby, working far away at Texas Instruments. Noyce used a silicon substrate and Kilby used germanium. Silicon proved easier to work with. Otherwise the world would be talking about "Germanium Valley". – Richard Tedlow

The "traitorous eight", clockwise from Robert Noyce (front and center): Jean Hoerni, Julius Blank, Victor Grinich, Eugene Kleiner, Gordon Moore, Sheldon Roberts and Jay Last (Oct. 1957).

"The traitorous eight was a group of eight employees who left Shockley Semiconductor Laboratory in 1957 to found Fairchild Semiconductor. Before the mid-20th century, it was unheard of to leave a company and start out on your own – college grads were expected to stay at their first real-world job until they retired. When the traitorous eight left Shockley they blew up that notion, and ultimately proved that starting your own company could prove to be highly rewarding both professionally and for the bank account. The engineer as rich guy archetype was created."

"Without Robert Noyce and Gordon Moore today's Silicon Valley startups wouldn't exist. Moore and Noyce were among the first to walk away from a stable corporate job and pursue their own startup dreams. Perhaps Noyce's biggest break from then accepted corporate rules, and a perk that Silicon Valley entrepreneurs and investors now say is crucial to any chance of success is the practice of handing out stock options to lure and reward employees."

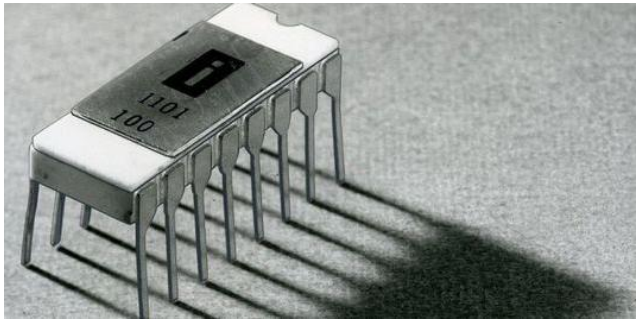
Textzitate (Auszüge) und Bild: www.wired.com/2013/02/silicon-valley-pbs/

INTEL

Zum Aufstieg von Intel zitieren wir aus dem HNF-Blog:

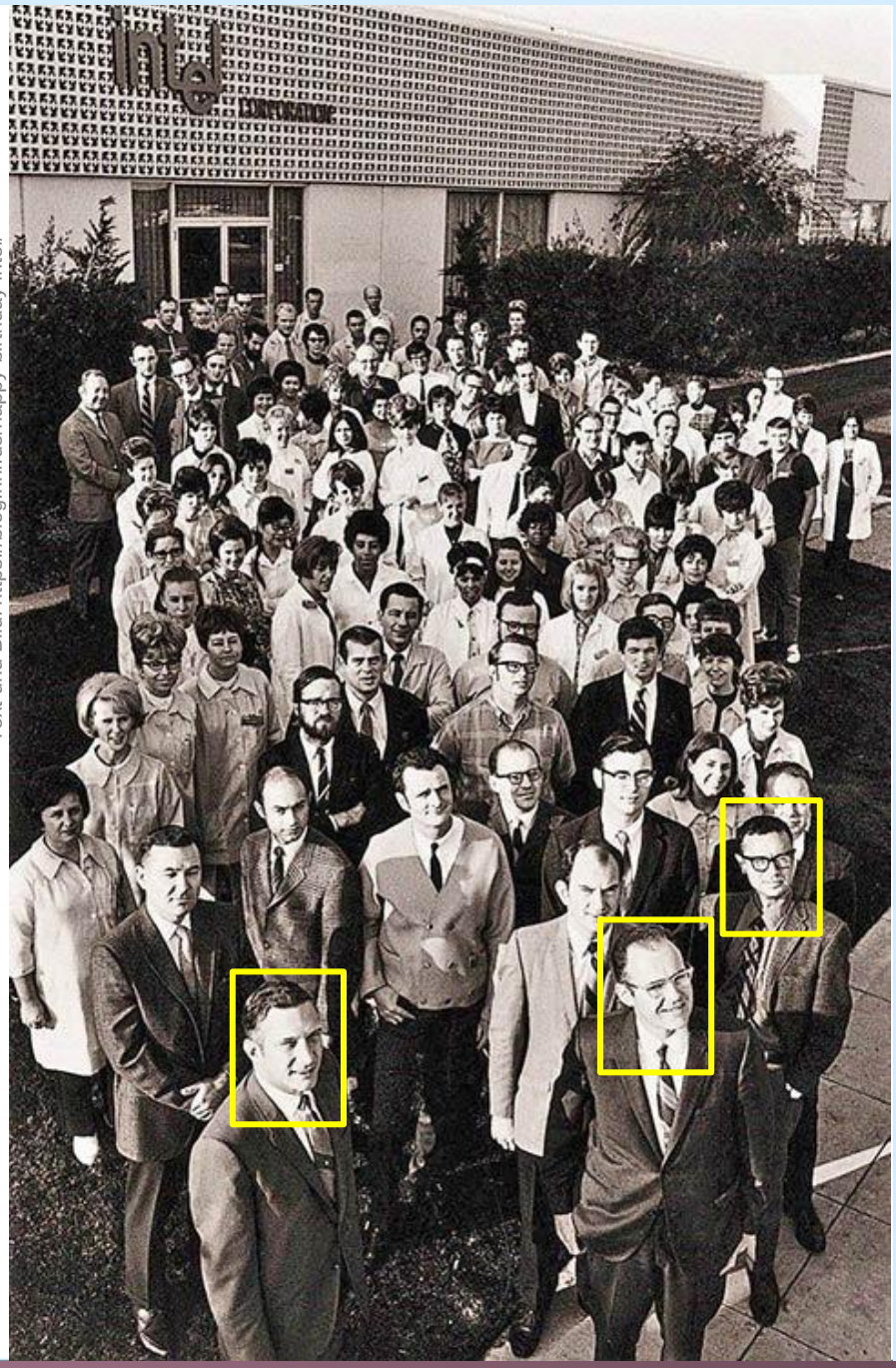
„Robert Noyce wurde Firmenchef und Gordon Moore sein Vertreter. Den wichtigen Posten des technischen Direktors erhielt der Chemieingenieur Andrew Grove. Er war 31, gebürtiger Ungar und hatte ebenfalls bei Fairchild gearbeitet.

Im August 1968 begannen die Arbeiten an drei konkurrierenden Projekten. Im April 1969 kam das erste Resultat auf den Markt, der Speicherchip Intel 3101. Er fasste 64 Bit – also acht Byte – bei einer Zugriffzeit von sechzig Nanosekunden und kostete hundert Dollar. Im Juli erschien der parallel zum 3101 entwickelte Intel 1101. Der Chip war langsamer, speicherte aber 265 Bit.



Der SRAM-Chip 1101. Mit seiner MOS-Technik war er zukunftsweisend.

Die dritte Entwicklungslinie führte zur Erfindung des EPROM, des löschbaren und neu zu beschreibenden Festwertspeichers. Im Oktober 1970 brachte die Firma

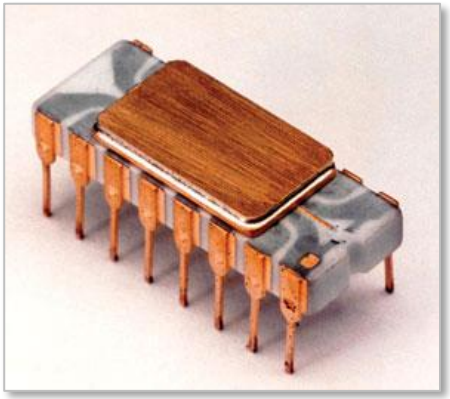


Die 106 Köpfe starke Intel-Belegschaft 1969 vor dem ersten Firmensitz in Mountain View. Vorne links Robert Noyce, rechts daneben Gordon Moore, rechts hinter diesem Andrew Grove.

Text und Bild: https://blog.hnf.de/happy-birthday-intel/

INTEL (2)

den 1024-Bit-Chip Intel 1103 heraus. Sein Preis von 10.24 Dollar läutete das Ende des Kernspeichers aus magnetisierbaren Ringen ein. Der erste Mikroprozessor aus Mountain View, der Intel 4004, wurde ab November 1971 verkauft. Er verarbeitete vier Bit lange Datenreihen. Im April 1972 folgte der Intel 8008 für acht Bit.“



Der 4004-Mikroprozessor und eine Anzeige dazu von 1971.

Die bekannteste Baureihe von Intel sind die x86-Mikroprozessoren, deren erstes Modell im Jahre 1978 mit dem 8086/8088 erschien. Einen Durchbruch bedeutete 1981 die Nutzung des Chips im Personal Computer von IBM.

Noch bis Mitte der 1980er-Jahre waren allerdings DRAM-Speicherchips das Hauptgeschäft von Intel. Die wachsende Konkurrenz durch japanische Halbleiterhersteller und der damit verbundene Umsatzrückgang führten aber 1983 zu einer Umorientierung des Geschäftsmodells. Der Erfolg des IBM-PCs veranlasste Intel, sich auf Mikroprozessoren zu konzentrieren, und mit der x86er-Reihe erlangte Intel schliesslich eine marktbeherrschende Stellung in der PC-Industrie. Konkurrentin in diesem Bereich wurde AMD; im wachsenden Markt von Prozessoren für mobile Geräte wie Smartphones und Tablets konnten andere Firmen wie ARM schneller punkten als Intel; bei KI-Prozessoren war Nvidia schneller als Intel.

In den 1980er- und 1990er-Jahren leitete Andy Grove (1936 – 2016) das Unternehmen. Grove wurde in Ungarn als Kind einer jüdischen Kaufmannsfamilie geboren. Während der nazideut-

A micro-programmable computer on a chip!

Intel introduces an integrated CPU complete with a 4-bit parallel adder, sixteen 4-bit registers, an accumulator and a push-down stack on one chip. It's one of a family of four new ICs which comprise the MCS-4 micro computer system—the first system to bring you the power and flexibility of a dedicated general-purpose computer at low cost in as few as two dual in-line packages.

MCS-4 systems provide complete computing and control functions for test systems, data terminals, billing machines, measuring systems, numeric control systems and process control systems.

The heart of any MCS-4 system is a Type 4004 CPU, which includes a powerful set of 45 instructions. Adding one or more Type 4001 ROMs for program storage and data tables gives you a fully functioning micro-programmed computer. To this you may add Type 4002 RAMs for read-write memory and Type 4003 registers to expand the output ports.

Using no circuitry other than ICs from this family of four, you can create a system with 4096 8-bit bytes of ROM storage and 5120 bits of RAM storage. When you require rapid turn-around or need only a few systems, Intel's erasable and re-programmable ROM, Type 1701, may be substituted for the Type 4001 mask-programmed ROM.

MCS-4 systems interface easily with switches, keyboards, displays, teletypewriters, printers, readers, A-D converters and other popular peripherals.

The MCS-4 family is now in stock at Intel's Santa Clara headquarters and at our marketing headquarters in Europe and Japan. In the U.S., contact your local Intel representative for technical information and literature.

INTEL (3)

schen Besatzung im Zweiten Weltkrieg wurde er mit seiner Mutter bei einer christlichen Familie versteckt; sein Vater überlebte die Deportation in ein Arbeitslager. Nach dem missglückten Volksaufstand Ende 1956 flüchtete er mittellos von Ungarn nach Österreich und weiter in die USA, obwohl er anfangs so gut wie kein Englisch sprach. Er hielt sich zunächst mit Hilfsarbeiten über Wasser, studierte Chemie am City College of New York, machte 1963 seinen PhD in Berkeley und fing anschliessend bei Fairchild an; am Tag der Gründung von Intel wechselte er zu diesem Start-up. ("When I came to Intel, I was scared to death. I left a very secure job where I knew what I was doing and started running R&D for a brand new venture in untried territory. It was terrifying.") Ende 1997 kürte ihn das Magazin TIME zum Mann des Jahres.

Im Jahr 2023 verzeichnete Intel mit mehr als 124000 Mitarbeitern und Mitarbeiterinnen einen Umsatz von 54 Milliarden Dollar und einen Gewinn von 1.7 Milliarden Dollar.

Probably no one person has had a greater influence in shaping Intel, Silicon Valley, and all we think about today in the technology world than Andy Grove.
-- Pat Gelsinger, Wall Street Journal, 2016

Von li. nach re.: Andy Grove, Robert Noyce und Gordon Moore (1978).

INTEL (4)

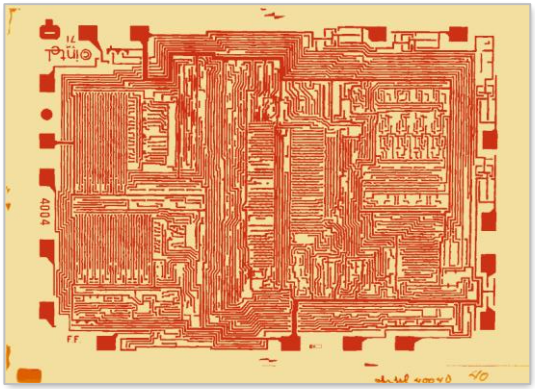
Der kommerzielle Erfolg von Mikroprozessoren wurde von Intels Marketingabteilung zunächst kritisch gesehen, man dachte nur an Prozessoren für „Minicomputer“. Erst später erkannte man das grosse unabhängige Potential für eingebettete Systeme. Die Herausforderungen der Marktentwicklung waren jedoch gross:

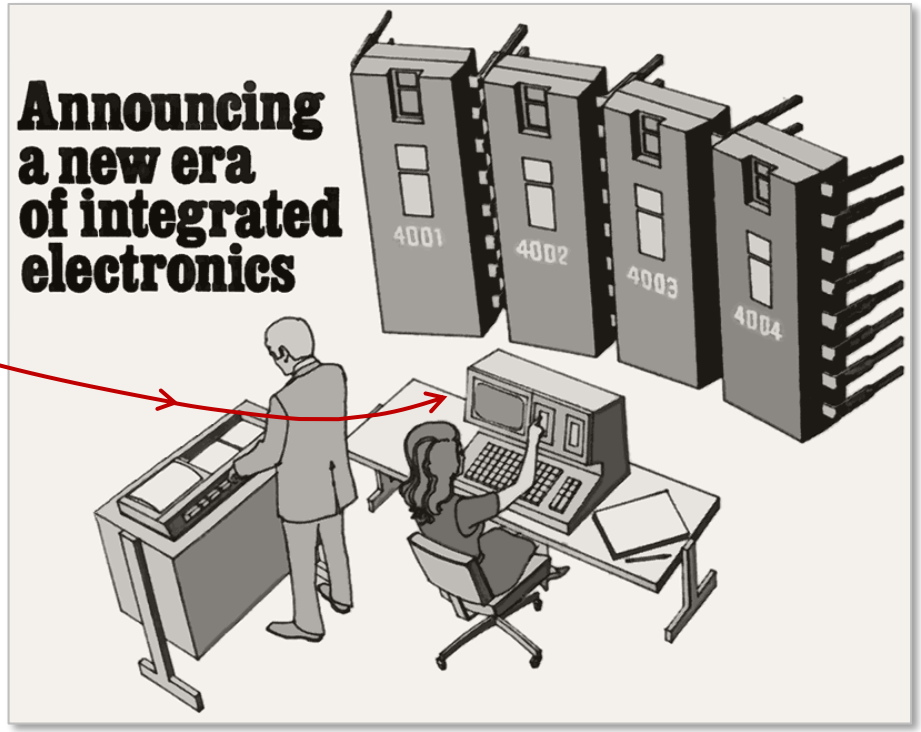
„Intel could offer little in software and development tools to support this entirely new class of product. It was nearly impossible to get computer programmers to consider joining a semiconductor company, especially to work on such a contemptible little toy as the microprocessor. That was the era of large computers and many programmers felt they would lose prestige if they worked on anything less than a mainframe.

To spread the word about the microcomputer, Intel began an ambitious program of seminars, customer training, and promotion. The company was seeking to replace hard-wired logic rather than computer systems. Hardware circuit designers thus had to become convinced it was worthwhile to switch to software-based solutions.“

[Robert Noyce, Marcian Hoff: A history of microprocessor development at Intel. IEEE Micro, Feb. 1981]

Der promovierte Physiker Federico Faggin entwarf 1970 den 4004-Mikroprozessor; seine Initialen „FF“ finden sich links unten auf der Chip-Maske. 1974 verliess er Intel und gründete die Firma Zilog, wo er u.a. den Z80-Mikroprozessor schuf.





Die erste Annonce für einen „micro-programmable computer on a chip“ (der Begriff „microprocessor“ wurde erst 1972 von Intel verwendet) vom November 1971 spricht dreist von einem neuen Zeitalter – was sich aber als sehr zutreffend herausstellen sollte!

INTegrated ELectronic Circuit

Als zum Ende des 20. Jahrhunderts in den Vereinigten Staaten an die wichtigsten Ereignisse, Aspekte oder Personen aus den letzten 100 Jahren erinnert werden sollte, durfte der integrierte Schaltkreis in der dafür aufgelegten Briefmarkenserie „Celebrate the Century“ nicht fehlen.

Die Serie besteht aus insgesamt 10 Bögen, einem für jedes Jahrzehnt, wobei jeder Bogen 15 Briefmarken mit Motiven des entsprechenden Jahrzehnts (hier: 1960er-Jahre) umfasst. Auf der gummierten Rückseite findet sich jeweils eine kurze Beschreibung des Motivs.

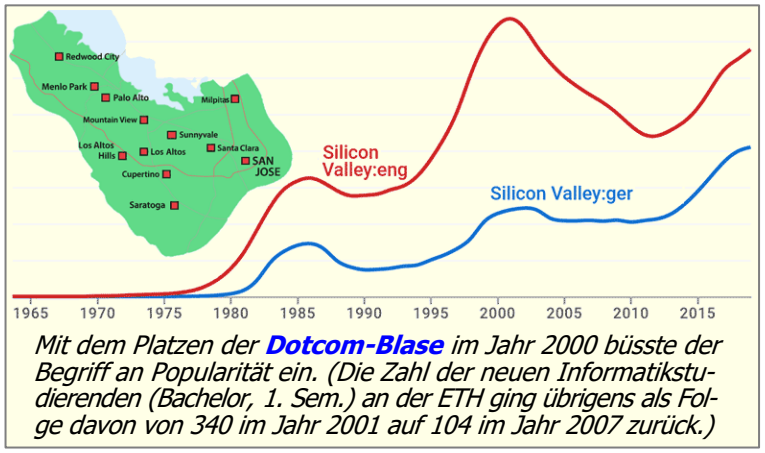
Andere Marken zu den 1960er-Jahren waren z.B. der Mondlandung, Martin Luther King, dem Woodstock-Musikfestival und dem Vietnamkrieg gewidmet. Weitere Motive des 20. Jahrhunderts waren u.a. Charlie Chaplin, Präsident Franklin D. Roosevelt, das Empire State Building, Jazz, Superman, das Monopoly-Spiel, der 2. Weltkrieg, Fernsehen, Antibiotika, Rock and Roll, der PC, WWW sowie das Mobiltelefon. (Die britischen Beatles wurden offenbar auch als amerikanisches Phänomen verstanden – zumindest deren Auftritt im US-Fernsehen im Februar 1964.)

Silicon Valley

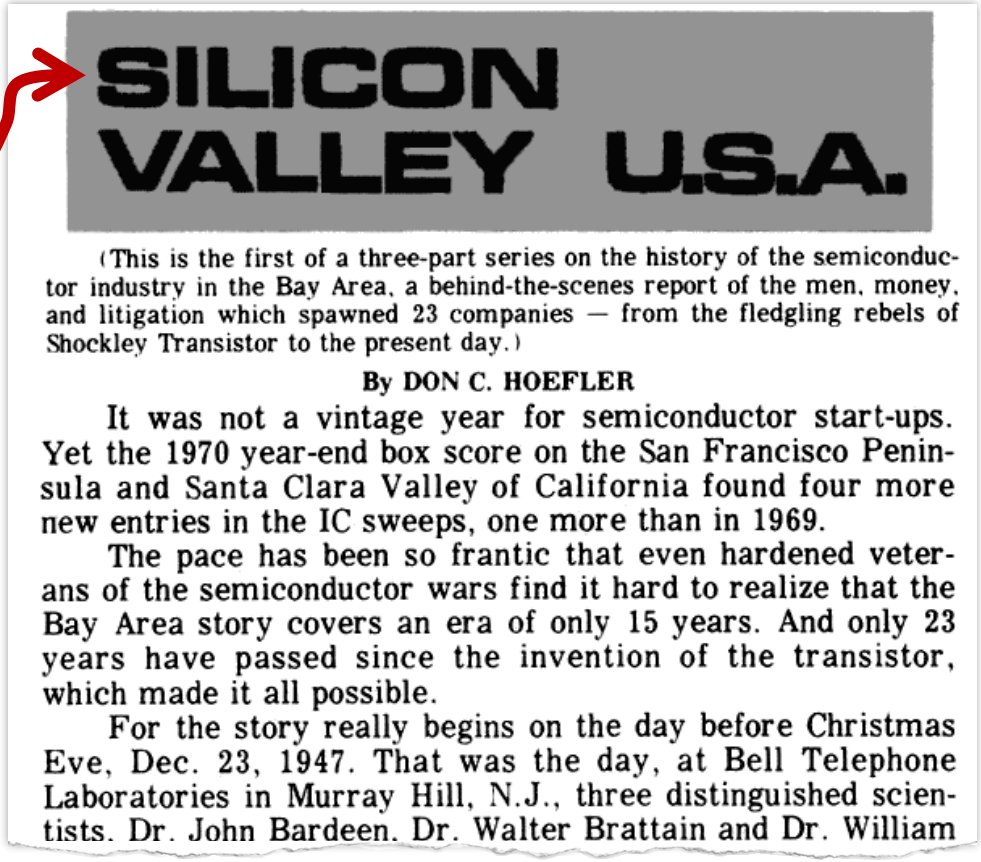
At that time, the tabloid press assumed silicon was a misspelling of silicone, a chemical compound better known to their readers for amplifying the female anatomy. – David Laws

Geprägt wurde die Bezeichnung „Silicon Valley" 1971 durch den Kolumnisten Don Hoefler (1922 – 1986). Er hatte den Begriff aufgeschnappt und platzierte ihn in die Überschrift eines Artikels der Wochenzeitschrift „Electronic News" zur Geschichte der Halbleiterindustrie.

Geographisch umfasst das Silicon Valley das Santa Clara Valley und die südliche Hälfte der Halbinsel von San Francisco („San Francisco Bay Area") in Kalifornien; das Gebiet ist ca. 100 Kilometer lang, 30 Kilometer breit und enthält u.a. die Städte San Jose (als grösste Stadt mit über einer Million Einwohnern), Palo Alto (HP, Tesla), Sunnyvale, Mountain View (Google / Alphabet), Menlo Park (Meta / Facebook), Santa Clara (Intel, AMD) und Cupertino (Apple). Silicon (deutsch: Silizium) bezieht sich auf das chemische Element, das den Hauptbestandteil integrierter Schaltkreise bildet.



*Mit dem Platzen der **Dotcom-Blase** im Jahr 2000 büsste der Begriff an Popularität ein. (Die Zahl der neuen Informatikstudierenden (Bachelor, 1. Sem.) an der ETH ging übrigens als Folge davon von 340 im Jahr 2001 auf 104 im Jahr 2007 zurück.)*



SILICON VALLEY U.S.A.

(This is the first of a three-part series on the history of the semiconductor industry in the Bay Area, a behind-the-scenes report of the men, money, and litigation which spawned 23 companies — from the fledgling rebels of Shockley Transistor to the present day.)

By DON C. HOEFLER

It was not a vintage year for semiconductor start-ups. Yet the 1970 year-end box score on the San Francisco Peninsula and Santa Clara Valley of California found four more new entries in the IC sweeps, one more than in 1969.

The pace has been so frantic that even hardened veterans of the semiconductor wars find it hard to realize that the Bay Area story covers an era of only 15 years. And only 23 years have passed since the invention of the transistor, which made it all possible.

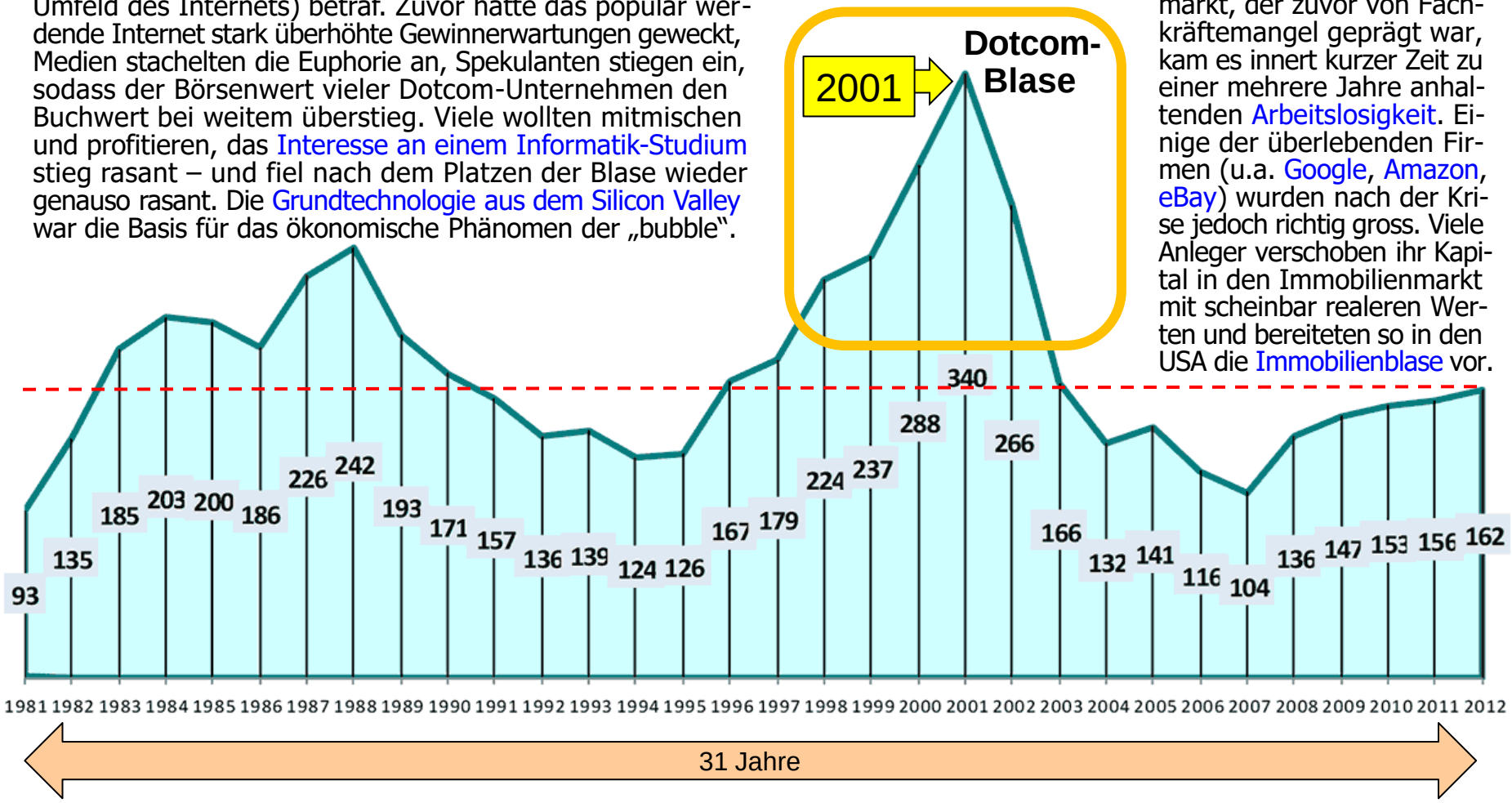
For the story really begins on the day before Christmas Eve, Dec. 23, 1947. That was the day, at Bell Telephone Laboratories in Murray Hill, N.J., three distinguished scientists, Dr. John Bardeen, Dr. Walter Brattain and Dr. William

Ab den 1950er-Jahren siedelten sich Unternehmen aus dem Elektronik-, Halbleiter- und Computerbereich an, viele initiiert durch Absolventen der dortigen Stanford University. Die direkt und indirekt aus der Firma von William B. Shockley hervorgegangenen Ausgründungen spielten, wie oben skizziert, eine wesentliche Rolle bei der Entwicklung des Silicon Valley, ebenso wie potente Risikokapitalgeber sowie der schnell wachsende Hochtechnologiebedarf des US-Militärs im kalten Krieg.

Stichwort „Dotcom-Blase“: Informatik-Studierende der ETH (Bachelor, 1. Semester)

„Dotcom-Blase“ bezeichnet eine im März 2000 geplatzte Spekulationsblase, die insbesondere die sogenannten Dotcoms (Unternehmen mit seinerzeit neuen Geschäftsmodellen im Umfeld des Internets) betraf. Zuvor hatte das populär werdende Internet stark überhöhte Gewinnerwartungen geweckt, Medien stachelten die Euphorie an, Spekulanten stiegen ein, sodass der Börsenwert vieler Dotcom-Unternehmen den Buchwert bei weitem überstieg. Viele wollten mitmischen und profitieren, das Interesse an einem Informatik-Studium stieg rasant – und fiel nach dem Platzen der Blase wieder genauso rasant. Die Grundtechnologie aus dem Silicon Valley war die Basis für das ökonomische Phänomen der „bubble“.

Das Platzen der Blase führte zu Insolvenzen und Stellenabbau. Auf dem IT-Arbeitsmarkt, der zuvor von Fachkräftemangel geprägt war, kam es innert kurzer Zeit zu einer mehrere Jahre anhaltenden Arbeitslosigkeit. Einige der überlebenden Firmen (u.a. Google, Amazon, eBay) wurden nach der Krise jedoch richtig gross. Viele Anleger verschoben ihr Kapital in den Immobilienmarkt mit scheinbar realeren Werten und bereiteten so in den USA die Immobilienblase vor.



Students flee field as computer 'fad' fades

BOSTON (AP) — College students have stopped flocking to computer science programs after learning job prospects are not as bright and that the field is more difficult than they had thought, university officials say.

"They found that they had to take calculus, they had to take physics. It's not a video games major," John Rice, chairman of Purdue University's Department of Computer Science, said in an interview Monday.

"Five years ago, computers looked like they were the land of good money and easy opportunity," said Paul Kalaghan, dean of the College of Computer Science at Northeastern University.

"I think today people understand it's a scientific discipline. Students found it was more difficult, that the mathematical rigor was large. It's not an easy business, really, when you couple that to the negative press the computer industry is getting."

The industry is concerned about the drop in enrollment, said Dev Glaser, college recruitment manager for Digital Equipment Corp.

"I see it reducing the applicant pool," she said. "There certainly is going to be a need for more and better technical talent in the high-tech industry."

A survey of 552 colleges by the University of California at Los Angeles found that about 1.6 percent of the freshmen who entered college last fall wanted to major in computer science. That compared with 2.1 percent in 1985 and 4 percent in 1982.

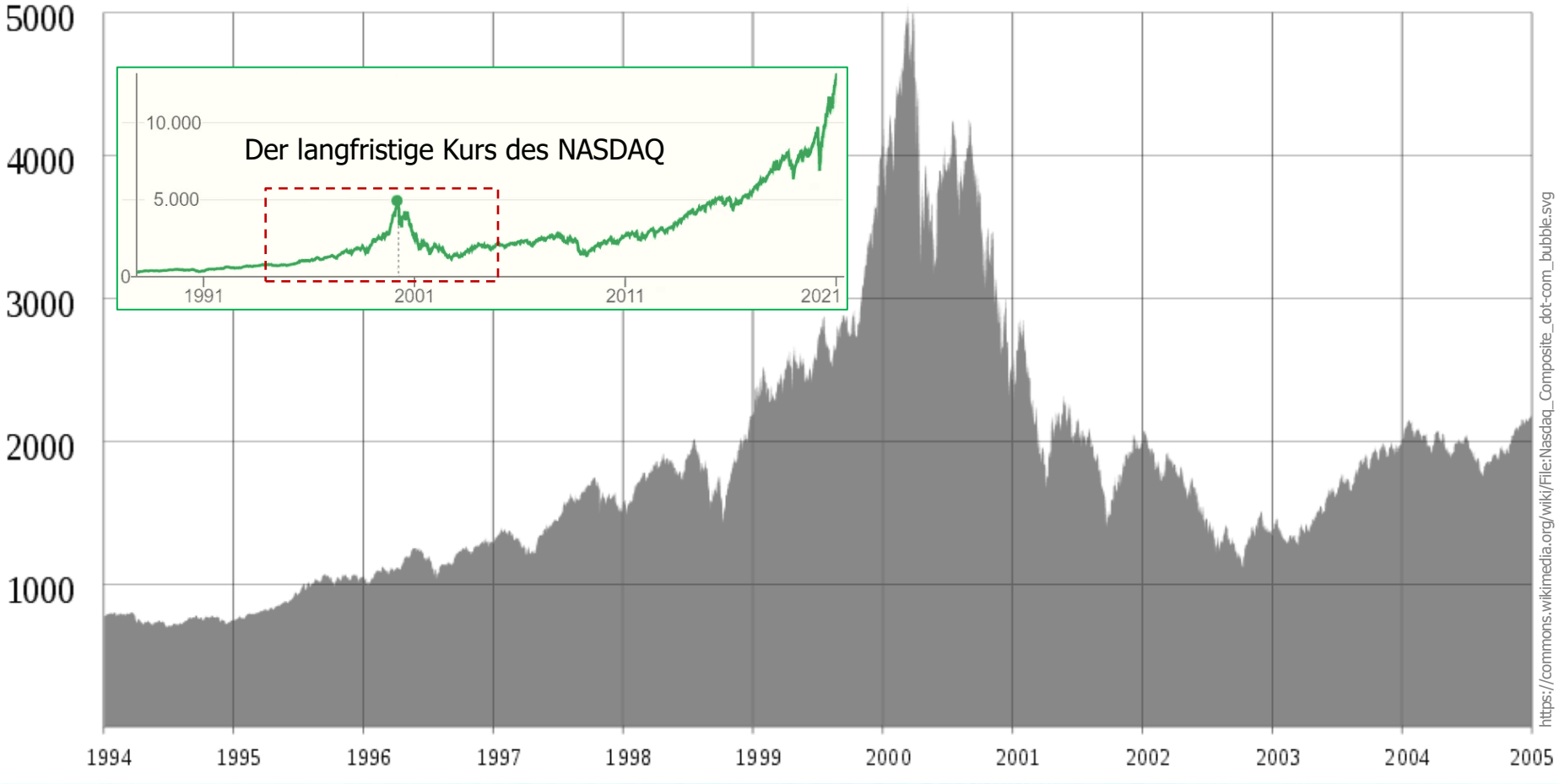
No enrollment figures were available, according to the Chronicle of Higher Education, which supplied the UCLA figures.

"For a long time it was a fairly specialized, technical field," said Jay Nievergelt, chairman of the Department of Computer Science at the University of North Carolina at Chapel Hill. "Then five years ago, personal computers hit the home and everybody thought you had to be a student in computer science. It was a fad."

Das Interesse am Informatik-Studium schwankt über die Jahre stark. Hier eine Meldung der „Bismarck Tribune" aus den USA vom 20. Jan. 1987, dass die „Freshmen"-Zahlen seit einigen Jahren zurückgehen. Man erkennt auch an den Statistiken der ETH Zürich (vorherige slide), dass der starke Anstieg Anfang der 1980er-Jahre um 1985 zu einem Ende kommt – im Wesentlichen ging es ab dann, von einem kleinen Zwischenhoch abgesehen, 10 Jahre lang bergab. Der chairman des computer science departments der University of North Carolina erläutert die wahrscheinlichen Gründe. Er wird im Artikel „Jay Nievergelt" genannt. Tatsächlich handelt es sich um ETH-Professor Jürg Nievergelt (1938 – 2019); er war von 1985 bis 1989 „on leave from ETH".

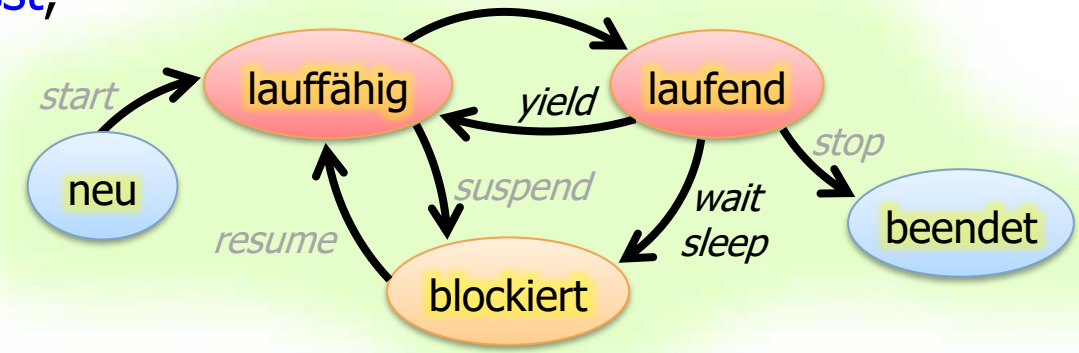
Stichwort „Dotcom-Blase“: NASDAQ Composite

Der Nasdaq Composite ist ein Aktienindex der USA; er umfasst über 3000 Unternehmen, überwiegend aus dem Bereich der Informationstechnologie. Der Indexstand von 5123 Punkten am 10. März 2000 markiert den Höhepunkt der Dotcom-Spekulationsblase. Bis zum 9. Oktober 2002 fiel der Index auf einen Stand von 1114 Punkten, ein Rückgang gegenüber dem vorherigen Höchststand um 77.9 Prozent. Dieser Tag markierte dann allerdings den Wendepunkt der gut zweieinhalbjährigen Talfahrt.



Java: Thread-Steuerung

- Ein Thread lebt (ist laufend / lauffähig / blockiert) so lange, bis er
  - seine run-Methode verlässt,
  - abgebrochen wird (z.B. durch eine exception)



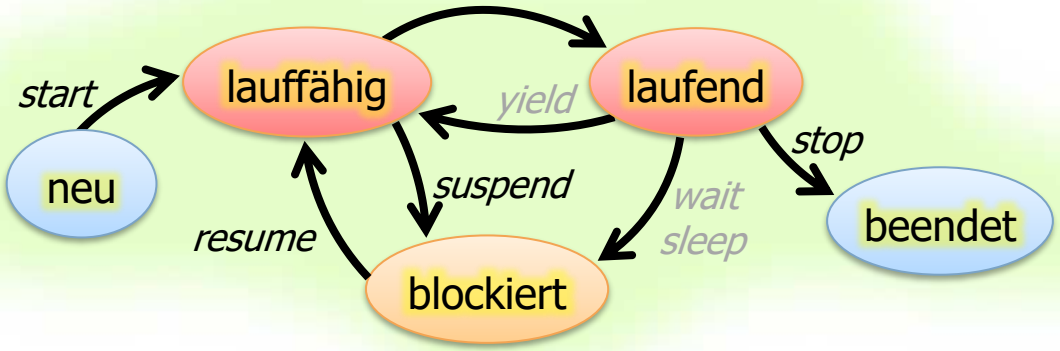
- Ein laufender Thread kann sich selbst
  - die CPU entziehen: yield()
    (Übergang in den Zustand „lauffähig"; wird automatisch wieder „laufend", wenn keine wichtigeren Threads mehr laufen möchten)
  - wartend anhalten: wait()
  - schlafend anhalten: sleep(…)
    (automatisches resume nach gegebener Zeit)
  - 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

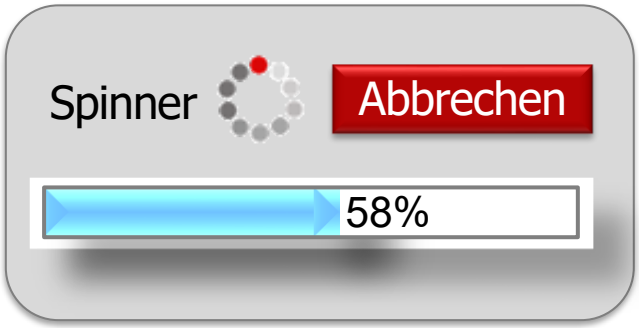
- Ein laufender Thread kann einen anderen Thread t
  - starten: t.start()
  - in der Priorität verändern: t.setPriority(…)

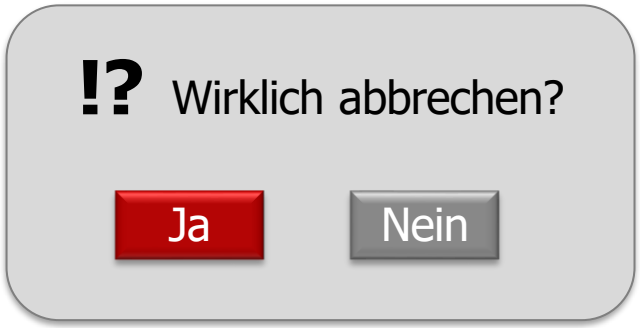


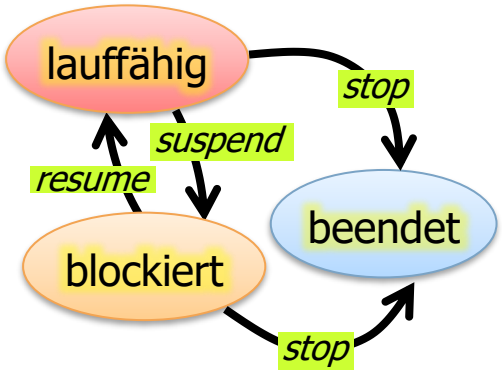
- und in früheren Java-Versionen auch
  - beenden: t.stop()
  - anhalten: t.suspend()
  - fortsetzbar machen: t.resume();

Kurze Begründung, wieso dies aus Java wieder entfernt wurde (Genaueres auf nachfolgenden slides): Ein Thread kann sich möglicherweise in einer Phase eines kritischen Abschnittes befinden und Daten teilweise geändert haben. Wird er angehalten, dann ist der kritische Abschnitt blockiert, und deadlocks sind die Folge. Wird er beendet, d.h. abgebrochen (und die Blockierung sodann vom System aufgehoben), dann sind Daten evtl. inkonsistent.

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 in vielen Situationen, unbedacht angewendet, zu unsicheren Programmzuständen oder Deadlocks
  - Sie sind daher aus Java schliesslich wieder entfernt worden!

Stop, suspend, resume…

Seit Java-Version 20 (März 2023) führt der Aufruf von stop, suspend oder resume zu einer UnsupportedOperationException – die Methoden können nun nicht mehr verwendet werden!

Dazu ein Auszug aus dem „Handbuch der Java-Programmierung" von Guido Krüger:

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… (2)

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

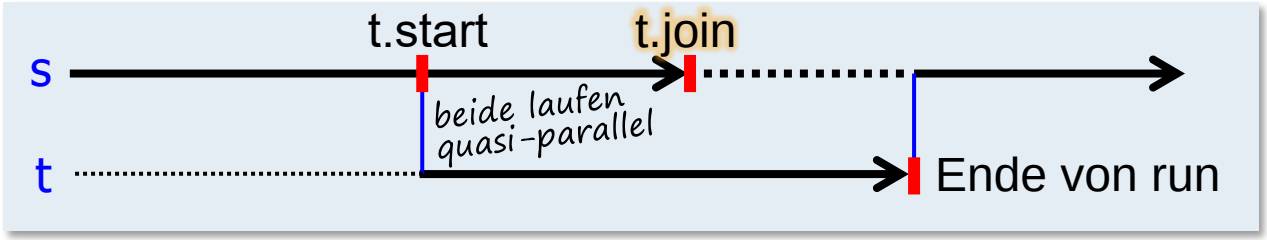
Thread-Ende

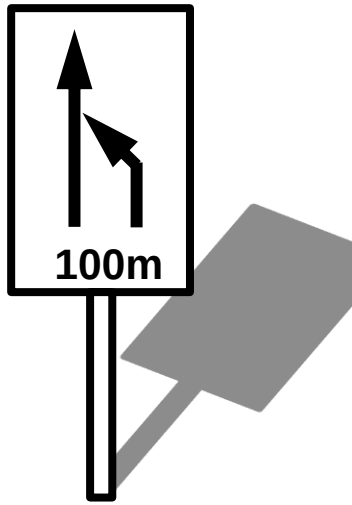
- Ein Thread ist beendet, wenn seine run-Methode beendet wird
  - evtl. auch vorzeitig aufgrund einer nicht abgefangenen Exception
- Das Objekt eines beendeten Threads aber existiert weiter
  - Auf dessen Zustand kann also noch zugegriffen werden
- Ein beendeter Thread kann mit start wieder neu loslaufen
  - 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

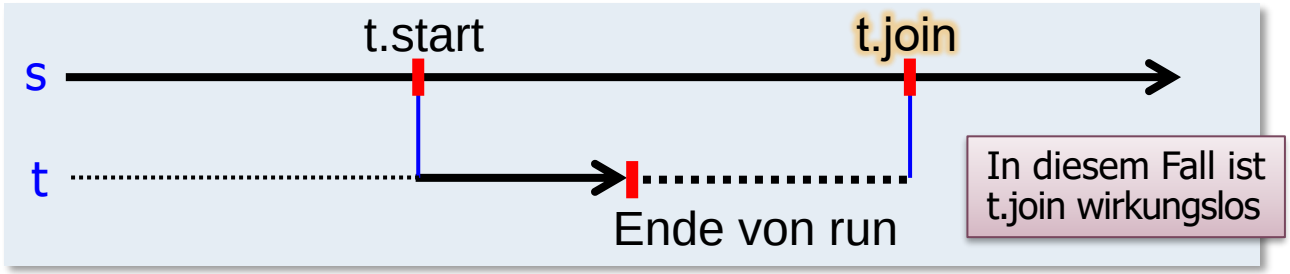
- Bsp: Thread s wartet so lang, bis t beendet ist:

Synchronisation





- 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 so weit ist („Barrierensynchronisation“)

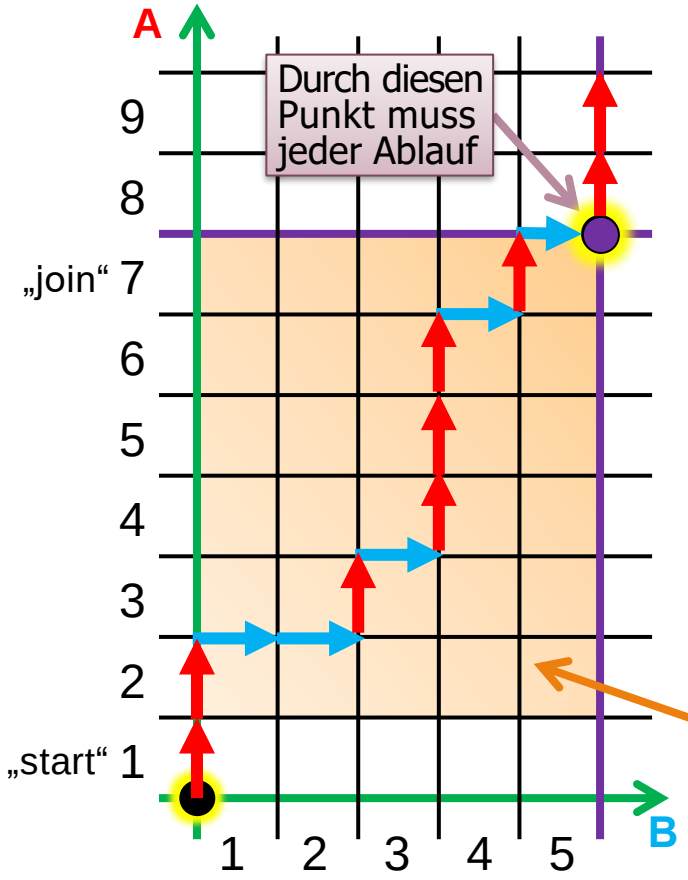
Gleiches Ergebnis, egal wer im Einzelfall zuerst da ist

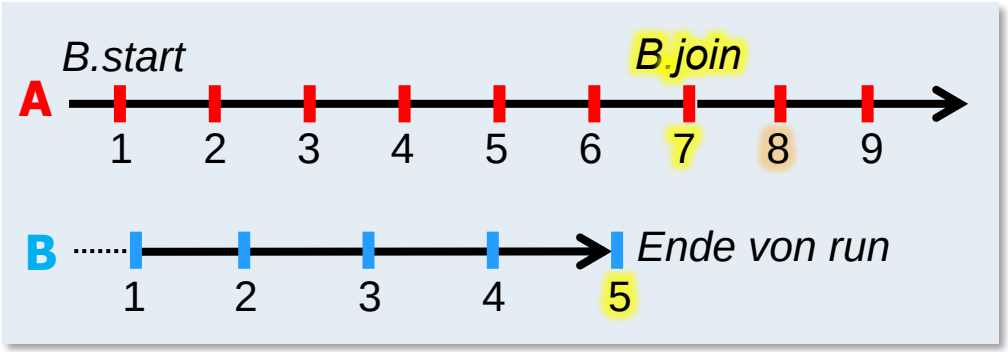
Wir idealisieren die Realität durch Abstraktion!

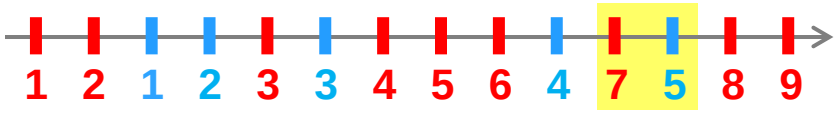
Synchronisationspunkte im Interleaving-Modell

- 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 (A)** erst nach **5 (B)**

Pfade von links unten nach rechts oben

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

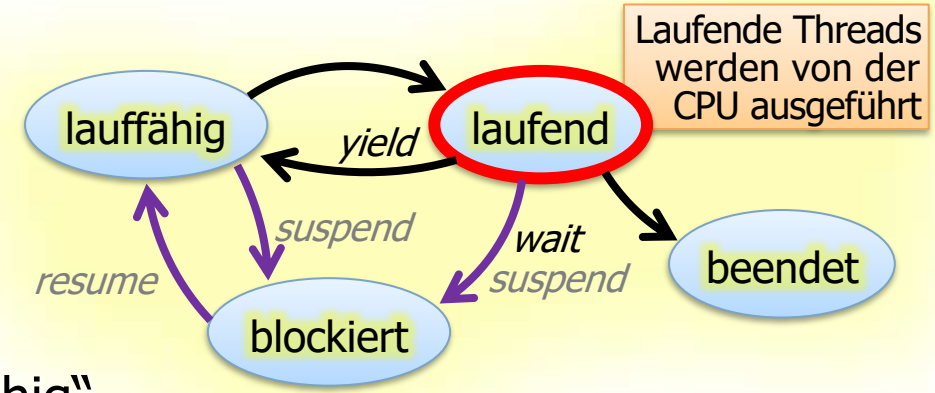
Neben „join“ (warten auf Ende) existieren weitere Möglichkeiten, auf andere Bedingungen zu warten

Threads:
Laufend, wartend, abseits

Wie lange ist ein Thread laufend?

1) Bis er sich beendet
   - 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 „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!)

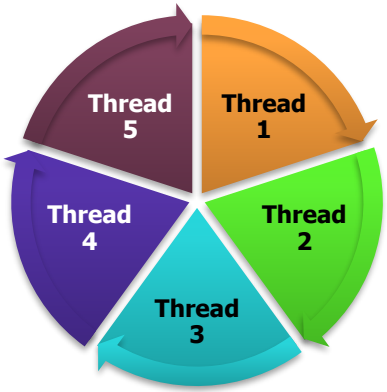
Bei einer Multicore-CPU können mehrere Threads „echt“ gleichzeitig laufen!

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)

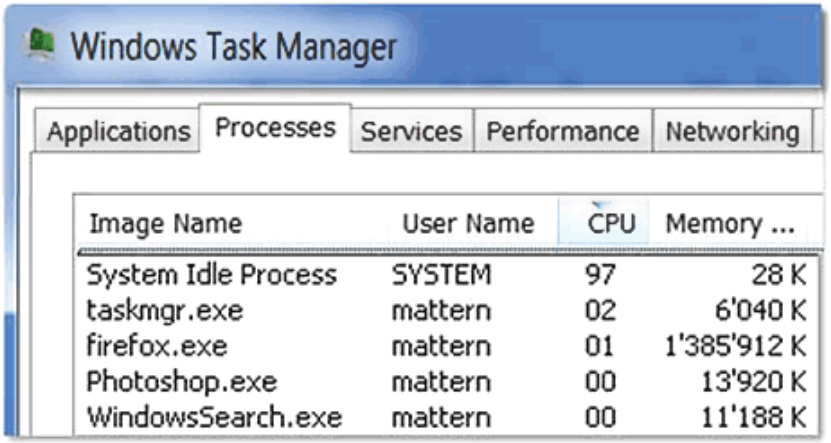
Schränkt Determiniertheit und Portabilität ein!

- 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



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);
}
```



In einer früheren Windows-Version konnte man den „System Idle"-Prozess sehen: Er dominiert hier die CPU

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 derjenigen 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 (sodass 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 Synchronisation einschränken (nur „korrekte“ Abläufe zulassen)
  - Synchronisationsfehler aufzuspüren ist besonders mühsam, da schlecht reproduzierbar (manchmal „Heisenberg-Effekt“: Testen ändert das Verhalten bzgl. des Fehlers)

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

Threads: Schwierigkeiten (2)

> Imagine sitting at a table with a fork, about to spear the last piece of food on a platter, and as your fork reaches for it, the food suddenly vanishes — because your thread was suspended and another diner came in and ate the food. That's the problem you're dealing with when writing concurrent programs. -- Bruce Eckel

- Bei Prozessoren mit mehreren CPUs bzw. Rechenkernen(„multicore") könnten entsprechend viele Threads „echt gleichzeitig" ausgeführt werden („multiprocessing")
  - Auch deswegen kein Verlass, dass mittels Prioritäten realisierte Synchronisation bzw. wechselseitiger Ausschluss 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."*

Virtual Threads

Ab JDK 21 (Sommer 2023) gibt es in Java „Virtual Threads" als neues feature. Hier einige Ausschnitte aus der entsprechenden Sprachbeschreibung:

Virtual threads are lightweight threads that dramatically reduce the effort of writing, maintaining, and observing high-throughput concurrent applications.

Goals: Enable server applications written in the simple thread-per-request style to scale with near-optimal hardware utilization; enable easy troubleshooting, debugging, and profiling of virtual threads. Non-Goals: It is not a goal to remove the traditional implementation of threads, or to silently migrate existing applications to use virtual threads; it is not a goal to change the basic concurrency model of Java.

Java developers rely on threads as the building block of concurrent server applications. Every statement in every method is executed inside a thread and, since Java is multithreaded, multiple threads of execution happen at once. The thread is Java's unit of concurrency: a piece of sequential code that runs concurrently with — and largely independently of — other such units. Each thread provides a stack to store local variables and coordinate method calls.

Server applications generally handle concurrent user requests that are independent of each other, so it makes sense for an application to handle a request by dedicating a thread to that request for its entire duration. This thread-per-request style is easy to understand, easy to program, and easy to debug.

Unfortunately, the number of available threads is limited because the JDK implements threads as wrappers around operating system (OS) threads. OS threads are costly, so we cannot have too many of them, which makes the implementation ill-suited to the thread-per-request style. If each request consumes an OS thread for its duration, then the number of threads often becomes the limiting factor long before other resources, such as CPU or network connections, are exhausted.

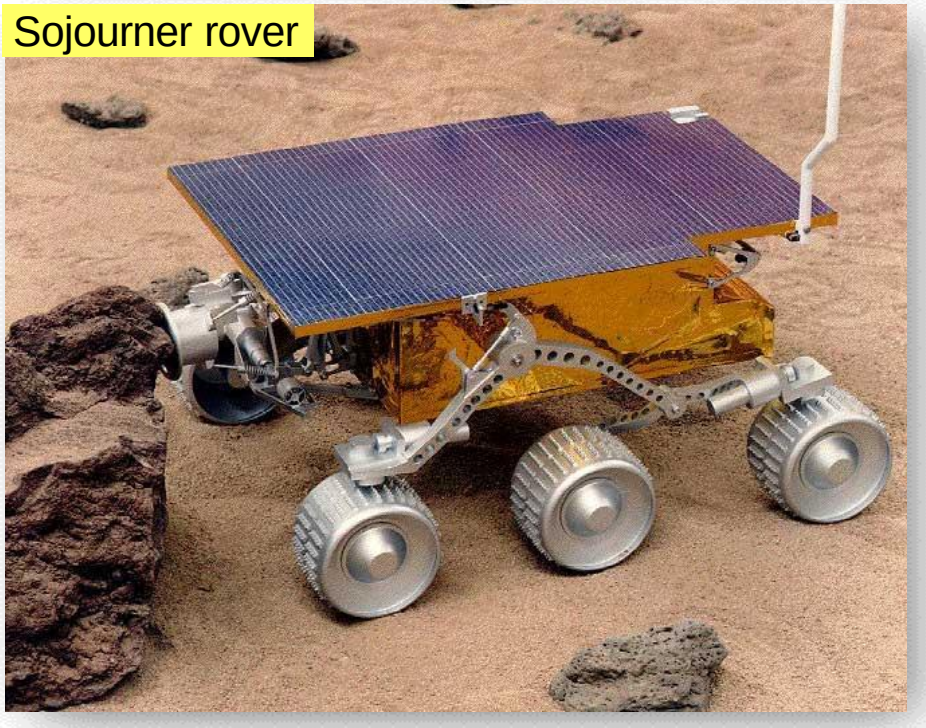
To enable applications to scale while remaining harmonious with the platform, we should strive to preserve the thread-per-request style. Just as operating systems give the illusion of plentiful memory by mapping a large virtual address space to a limited amount of physical RAM, a Java runtime can give the illusion of plentiful threads by mapping a large number of virtual threads to a small number of OS threads. When code running in a virtual thread calls a blocking I/O operation, the runtime performs a non-blocking OS call and automatically suspends the virtual thread until it can be resumed later. The number of virtual threads can be much larger than the number of OS threads. To Java developers, virtual threads are simply threads that are cheap to create and almost infinitely plentiful. Hardware utilization is close to optimal, allowing a high level of concurrency.

Virtual threads preserve the reliable thread-per-request style that is harmonious with the design of the Java Platform while utilizing the available hardware optimally.

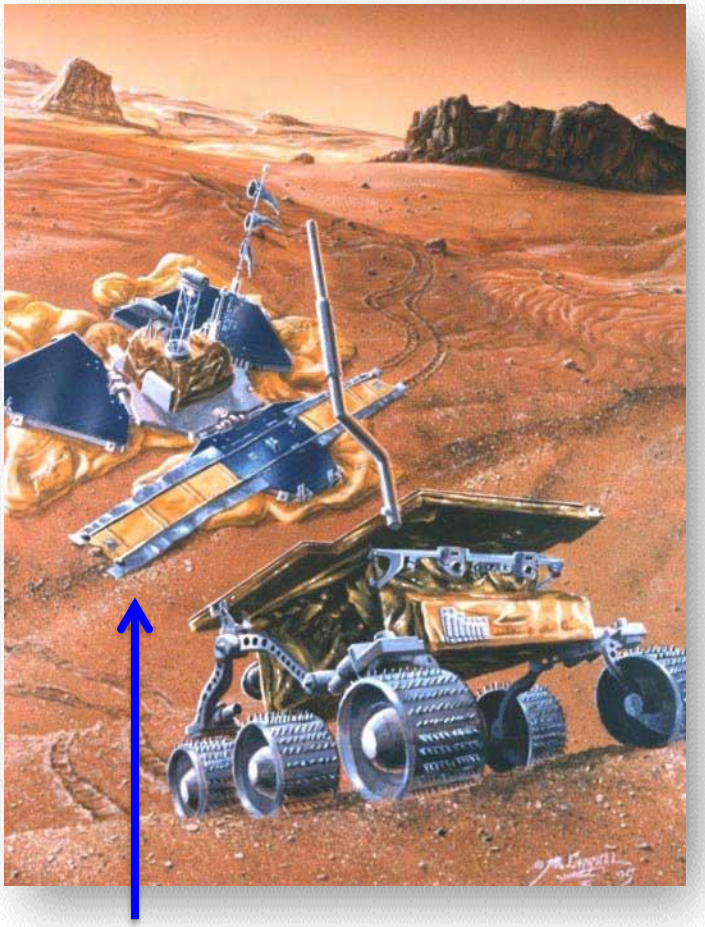
Historische Notiz

Parallele Threads auf dem Mars: 1997

Mars Pathfinder Mission

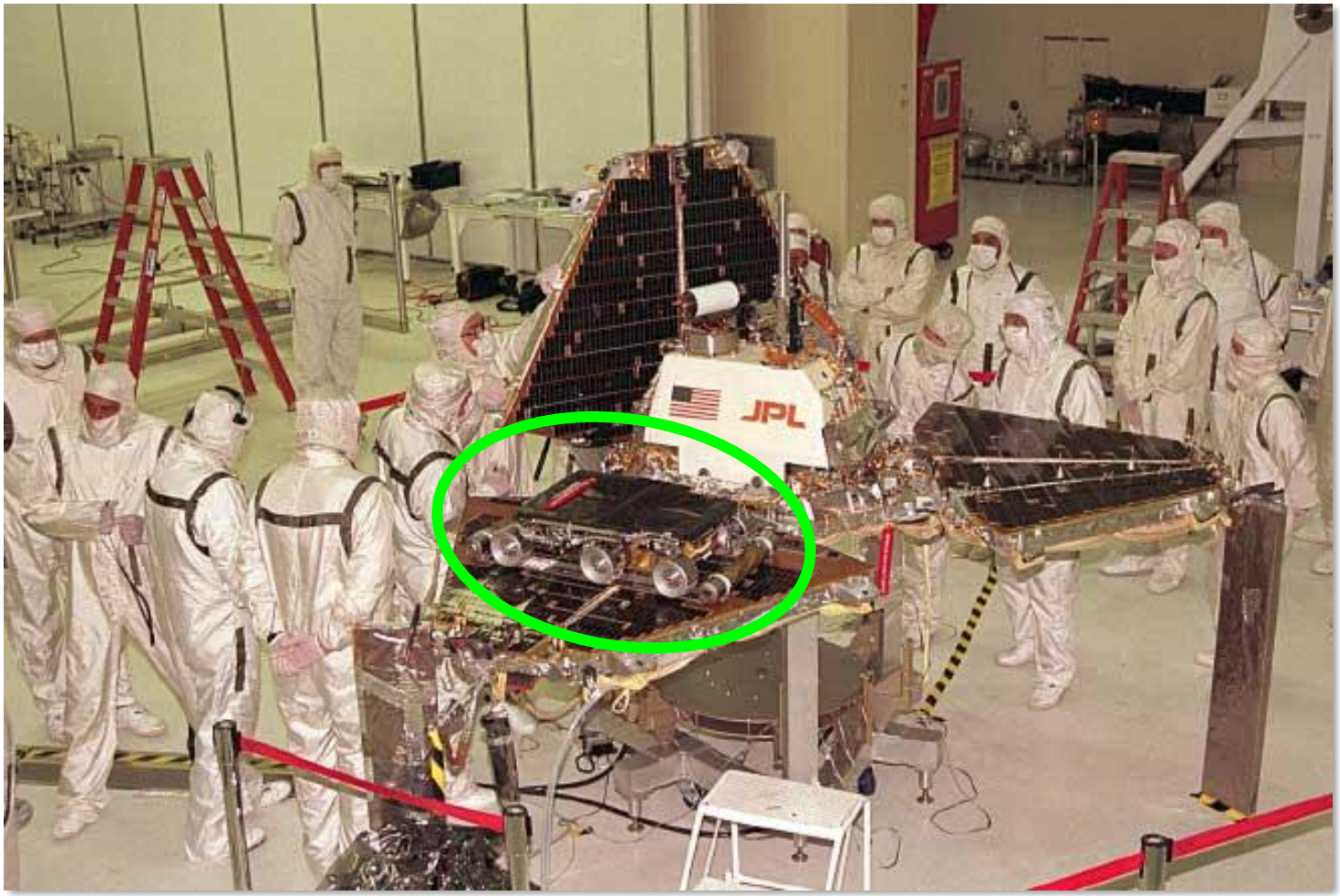


Computer im „Sojourner rover“: 2 MHz Intel-80C85 8-Bit CPU, 512 kB RAM, 176 kB Flash



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

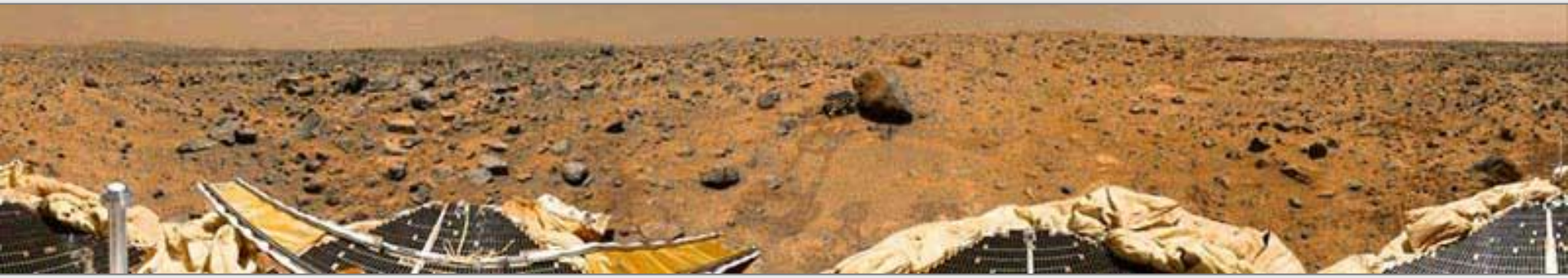
Dass die Landung am amerikanischen Nationalfeiertag erfolgte, war wohl kein reiner Zufall.

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

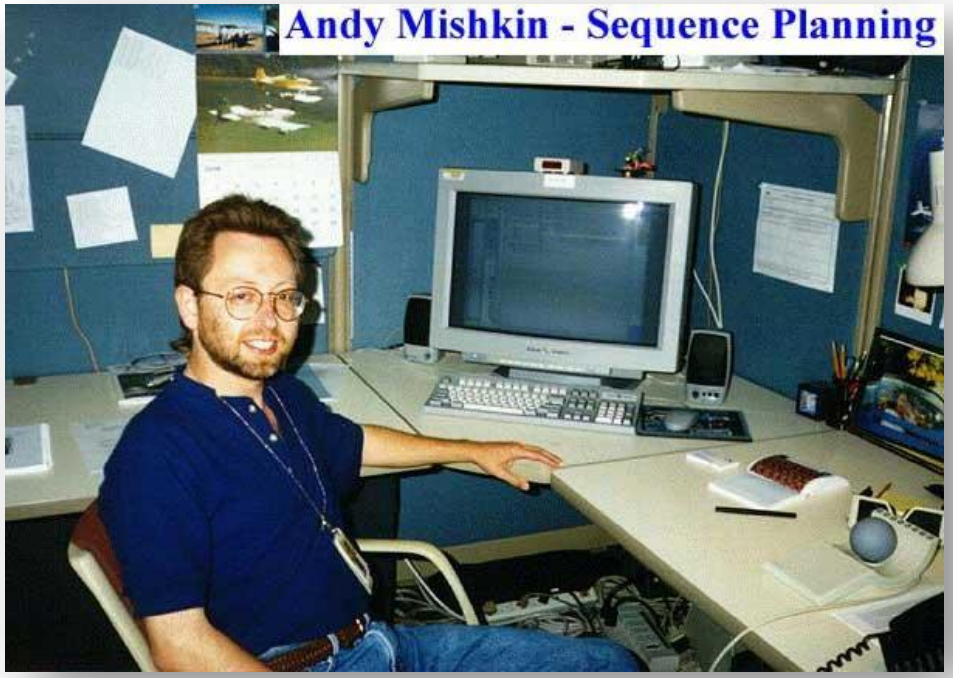
MISSION STATUS – 14 July 1997, 10:00 am PDT

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.

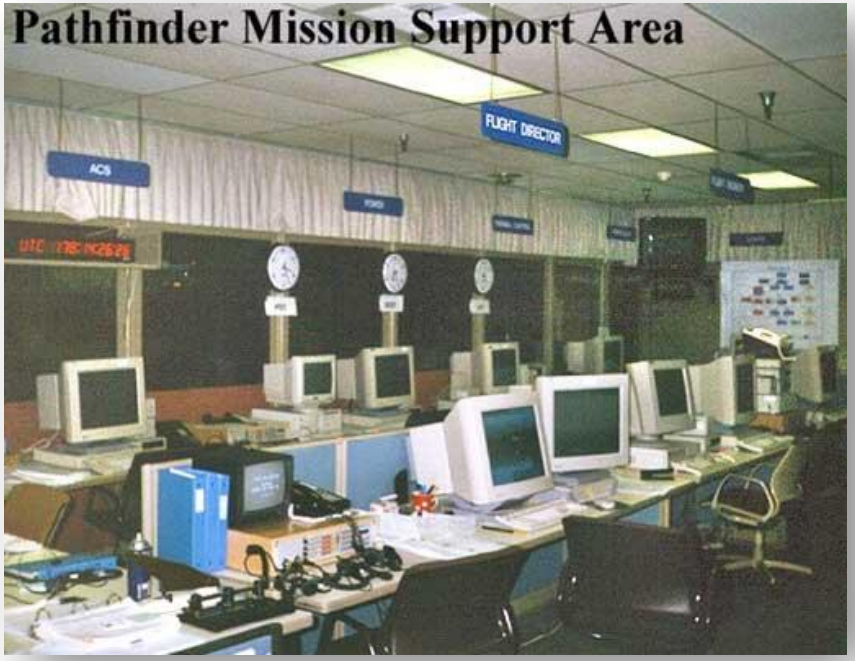


"There is time enough for everything in the course of the day, if you do but one thing at once, but there is not time enough in the year, if you will do two things at a time." -- Lord Chesterfield, ca. 1740

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

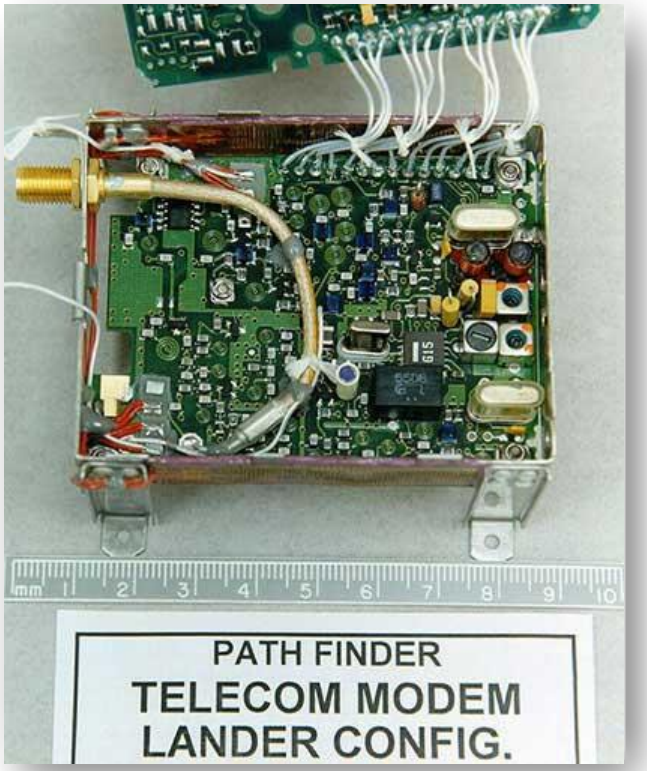
The problem was reproduced and isolated in testing at JPL. Further tests and verification will be completed today and tomorrow, with radio transmission of a software patch to change the lander's software scheduled for Saturday, Muirhead said.

How the patch was uploaded? VxWorks contained a C language interpreter to execute statements on the fly during debugging. The JPL engineers decided to launch the spacecraft with this feature still enabled ("test what you fly and fly what you test"). A short C program was uploaded to the spacecraft, which when interpreted, changed the values of the mutex flag for priority inheritance from false to true. [https://hownot2code.com/2016/12/19/the-first-bug-on-mars/]

Sent a Software Update to Mars

MISSION STATUS – July 21, 1997, 10 am PDT

The team... also sent a software update 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 Path-finder 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).

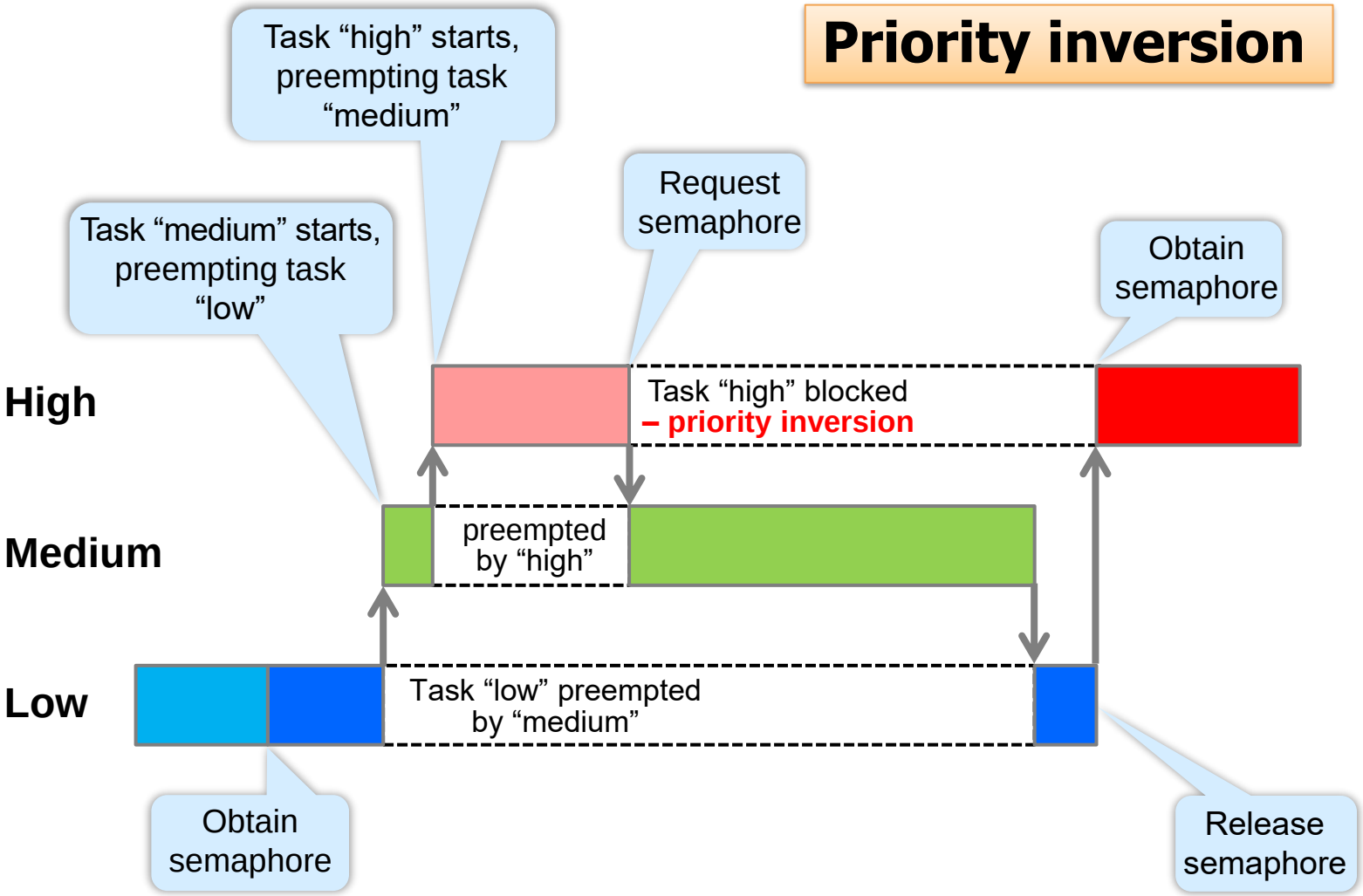
The meteorological data gathering task ran as an infrequent, low priority thread, and used the information bus to publish its data. When publishing its data, it would acquire a mutex, do writes to the bus, and release the mutex. If an interrupt caused the information bus thread to be scheduled while 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 space-craft 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)

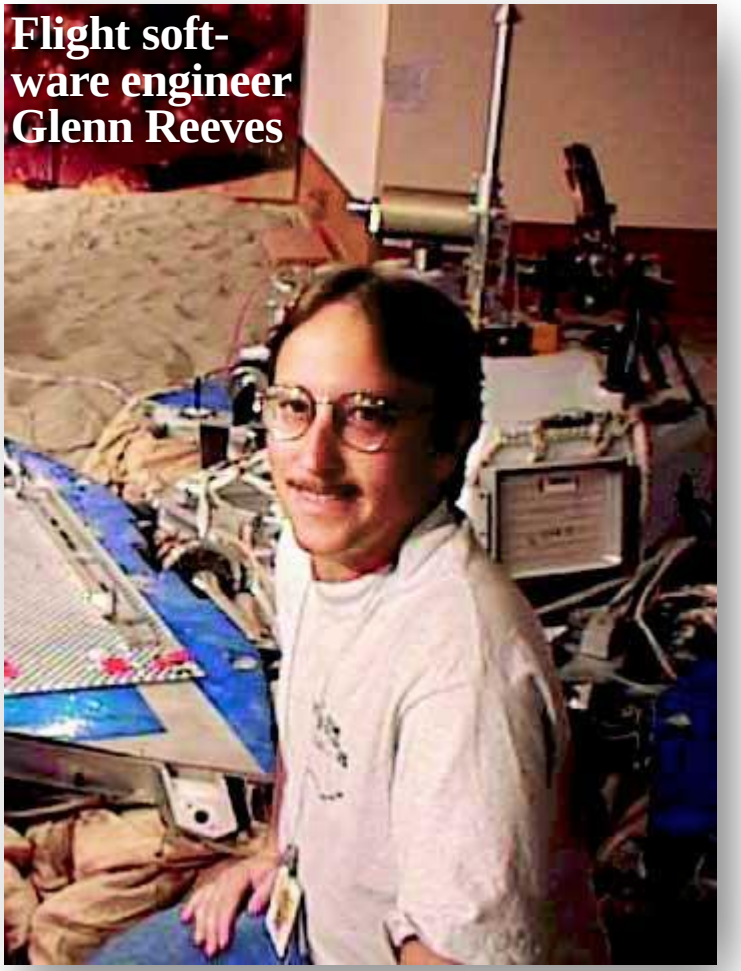


www.quora.com/What-really-happened-to-the-software-on-the-Mars-Pathfinder-spacecraft

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

What Really Happened on Mars? (6)

"We did see the problem before landing, but could not get it to repeat when we tried to track it down. It was not forgotten nor was it deemed unimportant. Yes, we were concentrating heavily on the entry and landing software. Yes, we considered this problem lower priority. Yes, we would have liked to have everything perfect before landing. However, I didn't see any problem, other than that we ran out of time to get the lower priority issues resolved.

We did have one other thing on our side – we knew how robust our system was because that is the way we designed it. We knew that if this problem occurred, we would reset. We built in mechanisms to recover the current activity so that there would be no interruptions in the science data (although this wasn't used until later in the landed mission). We built in the ability (and tested it) to go through multiple resets while we were going through the Martian atmosphere. We designed the software to recover from radiation induced errors in the memory or the processor. The spacecraft would have even done a 60-day mission on its own, including deploying the rover, if the radio receiver had broken when we landed. There were a large number of safeguards in the system to ensure robust, continued operation in the event of a failure of this type. These safeguards allowed us to designate problems of this nature as lower priority. We had our priorities right."

Glenn E. Reeves: Priority Inversion: How We Found It, How We Fixed It.
www.drdobbs.com/architecture-and-design/really-remote-debugging-a-conversation-w/228700403

Andere Softwareprobleme im Weltraum: 1962

Der meistzitierte Softwarefehler der IT-Geschichte

- 22. Juli 1962, Cape Canaveral / Florida: Start der ersten amerikanischen Venussonde „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 schnell vom Kurs ab und wurde zur Sicherheit 293 Sekunden nach dem Start per 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
```

Gemeint ist eine Schleife bis „5 CONTINUE“, welche mit K=1, K=2, K=3 iteriert.

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/

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

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.

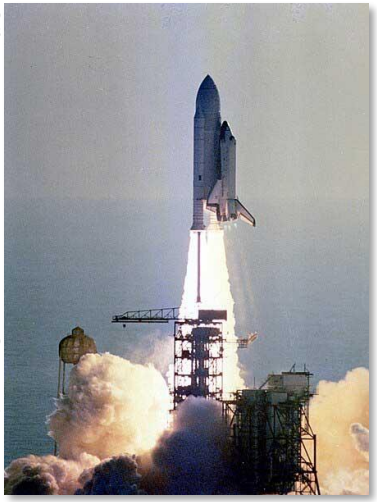
Es bleibt die Frage zu klären, wie der hier vorgestellte FORTRAN-Bug eine derart große Berühmtheit erlangen konnte, um heute zu den meistzitierten Softwarefehlern der IT-Geschichte zu zählen? Die Antwort darauf ist simpel. Der vorgestellte Fehler demonstriert nicht nur wie kaum ein anderer die Limitierungen 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.

*) „When the equations that would be used to process and translate tracking data into flight instructions were encoded onto punch cards, one critical symbol was left out: an overbar.” Der Überstrich steht für geglättete Werte. Durch die fälschliche Verwendung von Rohdaten stattdessen, registrierte der Computer sprunghafte Geschwindigkeitsänderungen, die er laufend mit drastischen Steuerbefehlen zu kompensieren versuchte.

Andere Softwareprobleme im Weltraum: 1981

Der Start des ersten Space-Shuttle-Flugs wurde abgebrochen


en.m.wikipedia.org/wiki/File:Sts1-liftoff-columbia.triddle.jpg

Das „Space-Shuttle", die bemannte Raumfähre der NASA, wurde von 1981 bis 2011 eingesetzt und konnte bis zu acht Personen und mehrere Tonnen Nutzlast in eine Erdumlaufbahn bringen. Das Space-Shuttle-Programm war sehr erfolgreich, auch wenn zwei der insgesamt fünf Orbiter, Challenger und Columbia, 1986 bzw. 2003 in tragischer Weise verunglückten, wobei jeweils alle sieben Besatzungsmitglieder ums Leben kamen.

Bzgl. Softwareprobleme ging allerdings der erste missglückte, aber weniger tragische, Startversuch in die Geschichte ein. Genau genommen manifestierte sich das Problem nicht im All, sondern „down earth" kurz vor dem geplanten Abheben am 10. April 1981. Bei Wikipedia lesen wir: „Die beiden Astronauten … lagen in der Kanzel des Orbiters in den Sitzen … Bei T-20 Minuten trat unerwartet ein Computerproblem auf, als in den fünf Hauptcomputern das Vorstartprogramm aktiviert wurde. … Bei T-9 Minuten, als der Countdown zum letzten Mal planmäßig angehalten wurde, zeigte sich, dass die fünf Computer ihre Daten nicht synchron austauschten: Zwischen Nr. 5 und den anderen bestand eine Differenz von 40 Millisekunden." Der Start wurde abgebrochen und um zwei Tage verschoben. Vier Monate später schrieb sich John R. Garman, Deputy Chief der NASA Spacecraft Software Division, seine teils frustrierenden Erkenntnisse in eindrücklicher Form vom Leibe. Wir zitieren daraus einige lesenswerte Passagen in gekürzter Form:

On April 10, 1981, about 20 minutes prior to the scheduled launching …, astronauts and technicians attempted to initialize the software system which "backs-up" the quad-redundant primary software system ..... and could not. In fact, there was no possible way, it turns out, that the BFS (Backup Flight Control System) in the fifth onboard computer could have been initialized properly with the PASS (Primary Avionics Software System) already executing in the other four computers. There was a "bug" – a very small, very improbable, very intricate, and very old mistake in the initialization logic of the PASS. It was the type of mistake that gives programmers and managers alike nightmares – and theoreticians and analysts endless challenge. It was the kind of mistake that "cannot happen" if

Der Start des ersten Space-Shuttle-Flugs wurde abgebrochen (2)

one "follows all the rules" of good software design and implementation. It was the kind of mistake that can never be ruled out in the world of real systems development: a world involving hundreds of programmers and analysts, thousands of hours of testing and simulation, and millions of pages of design specifications, implementation schedules, and test plans and reports. Because in that world, software is in fact "soft" – in a large complex real time control system like the Shuttle's avionics system, software is pervasive and, in virtually every case, the last subsystem to stabilize. Software by its nature is the easiest place to correct problems – but by that very nature, it becomes a tyrant to its users and a tenuous and murky unknown to the analysts. Software is the easiest to change .... but in change, it is the easiest to compromise.

The path to reliability in the Shuttle Orbiter spacecraft is through replication – replication of sensors, replication of effectors, replication of controls, computers, software, data buses, and power supplies. In fact, in order to satisfy a general Shuttle goal of "Fail Operational – Fail Safe" ("FO/FS") most components are replicated 4-deep … Four is the magic number for a very logical and intuitively obvious reason: FO/FS requires full operational capability after one failure, and a safe return capability after a second. It takes three to vote – so it initially takes four to still be able to vote after the first failure.

There are five onboard computers (called "GPC's") -- four operate with identical software loads during critical phases. That approach is excellent for computer or related hardware failures – but it doesn't fit the bill if one admits to the possibility of catastrophic software bugs … The thought of such a bug "bringing down" four otherwise perfect computer systems simultaneously and instantly converting the Orbiter to an inert mass of tiles, wires, and airframe in the middle of a highly dynamic flight phase was more than the project could bear. So, in 1976, the concept of placing an alternate software load in the fifth GPC, an otherwise identical component of the avionics system, was born.

That software system, plus an ingenious yet simple astronaut-managed control which permits only the fifth or the other four to have any control, is what makes up the BFS. … It was developed by an entirely different and remote organization and used an entirely different operating system. … The BFS, when not in control, would "listen" to all the input and some output data traffic to and from the PASS-controlled GPC's. In that way the "PFS" could independently "keep up" with the goings-on

Der Start des ersten Space-Shuttle-Flugs wurde abgebrochen (3)

of the Orbiter and mission and be ever-ready to take over control of the vehicle when switched in by the astronauts. This concept created several implicit but fundamental changes in the overall Orbiter design and operation.

First, the BFS had to "keep tabs" on the PASS and "stop listening" wherever it thought the PASS might be compromising data fetching. For in no way could any failure of the PASS be permitted to "pollute" the BFS. Second, the BFS could only possibly remain in a "ready" state for a short period after failure of the PASS. If there is no data to listen to, then only extrapolation and certain reinitialization techniques following switch-over would permit the BFS to take successful control of the vehicle. As such, astronauts were trained to make their choices quickly. …

The final fundamental effect of incorporating the BFS was that the system became even more complex. Another subsystem, especially one as intricately woven into the fabric of the avionics as is the BFS, carries with it solutions to some problems, but the creation of others. While it certainly increased the reliability of the system with respect to generic software failures, it is still argued academically within the project whether the net reliability is any higher today than it would have been had the PASS evolved to maturity without the presence of its cousin … On the other hand, almost everyone involved in the PASS-side "feels" a lot more comfortable!

Wir verzichten auf eine Schilderung der komplexen Problemursache, Interessierte können dies im paper von John Garman nachlesen: https://dl.acm.org/doi/pdf/10.1145/1005928.1005929. Nur so viel dazu: Es ist schon eine Herausforderung, 5 Computer synchron bzgl. der Zeit zu halten ("multiple computers with identical software and data can't look at "clocks" to see what time it is. If they did, they would each get slightly, ever so slightly, different values"). *Dann gab es eine winzige Programmänderung ein Jahr vor dem Start, die in einem von 67 Fällen das Problem auftauchen liess. Die eigentliche Ursache beschreibt Garman aber so:*

"It is complexity of design and process that got us (and Murphy's Law!). Complexity in the sense that we, the 'software industry' are still naive and forge into large systems such as this with too little computer, budget, schedule, and definition of the software role. We do it because these systems won't work, can't work, without computers and software."

Andere Softwareprobleme im Weltraum: 1999

1) 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, sodass 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

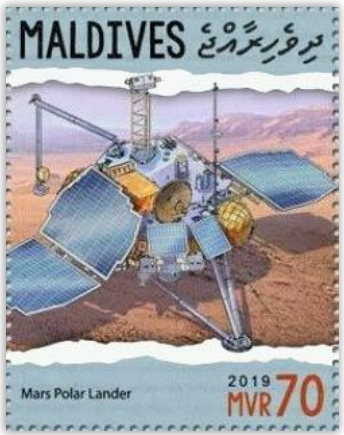
2) Mars Polar Lander killed by premature touchdown celebration

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, sodass die Sonde [aus einer Höhe von ca. 40m] mit unverringerter Geschwindigkeit auf der Marsoberfläche aufschlug und zerschellte.

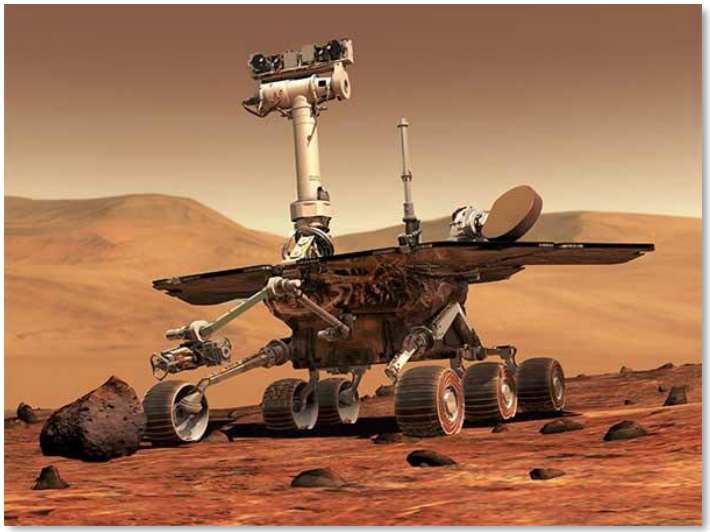
Der Misserfolg hindert nicht daran, das 20-jährige Jubiläum zu feiern: 2019 erinnern Briefmarken der Malediven und der Zentralafrikanischen Republik daran.

Kurz zuvor, 1996, geschah eine andere softwarebedingte Katastrophe: Die neue Ariane 5-Rakete explodierte schon kurz nach dem Start; wir beschrieben das an → früherer Stelle.



Andere Softwareprobleme im Weltraum: 2004

Mars Rover Spirit caught in a rebooting cycle



Im Januar 2004 landeten die beiden Weltraumsonden „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: 2007

ISS-Bordcomputer reagieren „allergisch“ auf Geräusche

Im Juni 2007 befand sich die Raumfähre „Atlantis“ für einige Tage zu einem Arbeitsbesuch bei der International Space Station (ISS); u.a. installierten die Atlantis-Astronauten neue Sonnensegel an der Raumstation. In dieser Zeit fielen plötzlich die Schubdüsen zur Lageregelung, die Sauerstoffversorgung und weitere Kontrollsysteme der ISS aus: Die zugehörigen Steuercomputer im russischen Teil der Raumstation waren abgestürzt und konnten nach Neustart nicht wieder in einen stabilen Zustand gebracht werden.

Zwar liess sich die Sauerstoffversorgung noch manuell regeln, eine die Bahnkorrektur der ISS musste man aber auf die Systeme der angekoppelten Raumfähre zurückgreifen. Da bekannt war, dass die Bordcomputer sensibel auf Probleme bei der Stromversorgung reagieren, vermuteten vor allem die russischen Beteiligten die Ursache bei den neu installierten amerikanischen Sonnensegeln.

RUSSLAND

ISS-Bordcomputer reagieren „allergisch“ auf Geräusche

Bei der Aufklärung des Computerausfalls auf der Raumstation ISS sind Russen und Amerikaner vorangekommen. Die russischen Computer reagierten sensibel auf Geräusche und schalteten sich automatisch ab, hieß es vom Nasa-Kontrollzentrum in Houston. Die Geräusche würden wahrscheinlich durch elektromagnetische Felder von Geräten ausgelöst. Russischen und amerikanischen Spezialisten ist es am Freitag den zweiten Tag in Folge nicht gelungen, die ausgefallenen Computer wieder in Betrieb zu nehmen. *dpa*

Wie kam es aber zu der Meldung über geräuschallergische russische Computer, welche die „Die Welt“ am 16. Juni veröffentlichte? Ihr liegt folgende NASA-Pressemitteilung zugrunde:

HOUSTON 14 June 9:42 p.m. ET -- Astronauts aboard the International Space Station (ISS) will physically unplug a pair of cables feeding power from new solar arrays to the outpost's Russian segment late Thursday in hopes that it might aid the recovery of critical computer systems, mission managers said. "The leading theory today is, we've introduced some noise at a level that, now, these computers are tripping themselves off," NASA ISS program manager Mike Suffredini said. The German-built computers used in the station's Russian segment are known to be susceptible to radiant or conductive "noise" in power feeds.

ISS-Bordcomputer reagieren „allergisch“ auf Geräusche (2)

Zwei Fehler in der Zeitungsmeldung fallen dabei auf: Es waren zum einen keine russischen Computer, sondern deutsche (hergestellt 10 Jahre zuvor von Daimler Benz), und zum anderen waren die „Geräusche“ im Original „noise“. Nun hat das englische „noise“ allerdings viele Bedeutungen neben „Geräusch“, dict.leo.org hat dazu 637 Einträge und nennt als Übersetzung u.a. *Furore*, *Krakeelerei*, *Störanfälligkeit*, *Störung* und *Rauschen*. Google übersetzt die kritische Passage übrigens recht elegant mit „anfällig für Strahlungs- oder Leitungsrauschen in der Stromversorgung“. Störungen in der Stromversorgung mögen die meisten Rechner nicht besonders, und das süffisante Lästern der „Welt“ über „geräuschallergische russische Computer“ war eine doppelte journalistische Fehlleistung.

Aber was war nun das eigentliche Problem? In drei Tagen „around-the-clock troubleshooting“ stellte die ISS-Besatzung fest, dass das Problem fortbestand, wenn man die neuen Solarpanels abkoppelte, es aber verschwand, wenn man ein den Bordcomputern vorgeschaltetes Gerät zur Stromüberwachung (und zum Schutz der Computer vor „unreinem“ Strom) überbrückte. Die Besatzung baute das Gerät auseinander und fand korrodierte und nasse Steckverbindungen vor und konnte auch einen Kurzschluss ausmachen. Wie Simulationen auf der Erde zeigten, war dieser dafür verantwortlich, dass allen Computern das Signal zum Herunterfahren gegeben wurde.

Diese Entdeckung war schockierend: Es handelt sich um einen „single point failure“, ein klarer Verstoss gegen die Entwurfsrichtlinien. Tatsächlich waren sechs kritische Bordcomputer gleichzeitig betroffen: Zwei, die für die wichtigen Steuerfunktionen verantwortlich waren, und – aufgrund der Dreifachredundanz – jeweils zwei weitere (in diesem Fall aber natürlich nutzlose) Backup-Systeme. Der gleichzeitige Ausfall aller Computer war beim Systems Engineering kein erwartbares Szenario gewesen. Dass es zu den Korrosionen kam, lag übrigens an einem schadhaften Luftaufbereitungsgerät in der Nähe, das unbemerkt Wasserdampf freisetzte.

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.

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

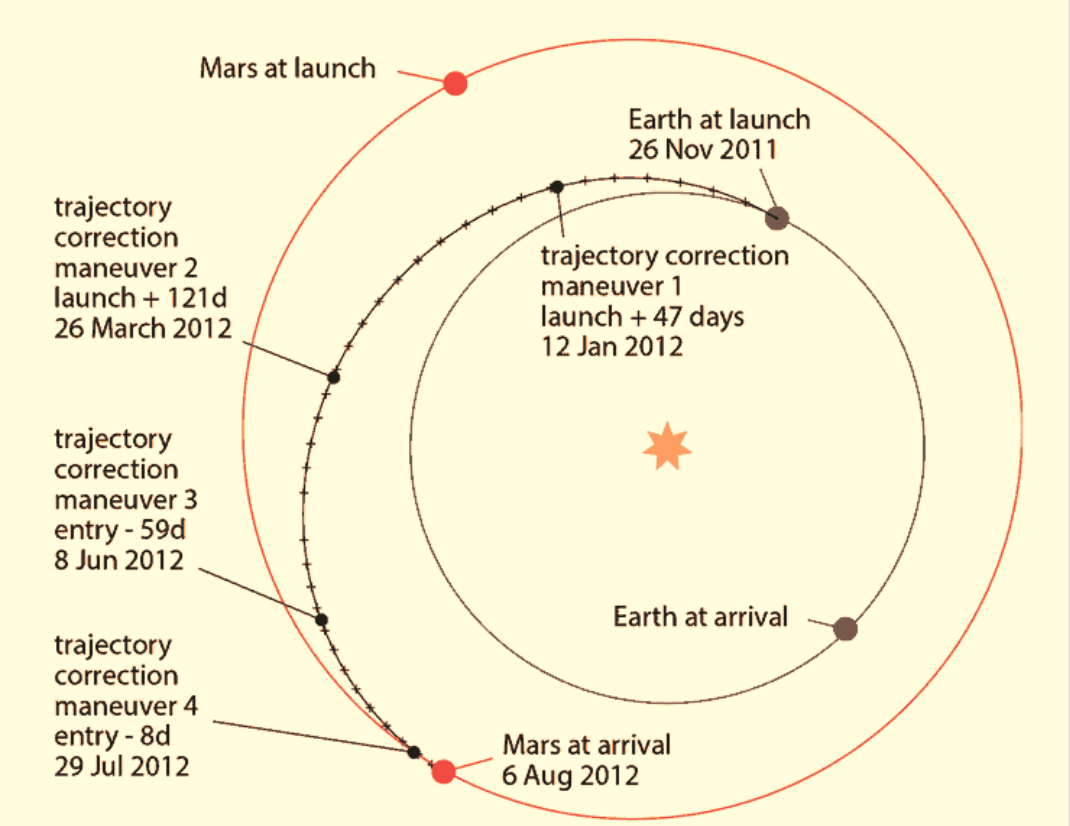
Mission status report, December 1, 2011: *The space-craft experienced a computer reset on Tuesday apparently related to star-identifying software in the attitude control system. The reset put the spacecraft briefly into a precau-tionary safe mode. Engineers restored it to normal oper-ational status for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

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 orientation 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.

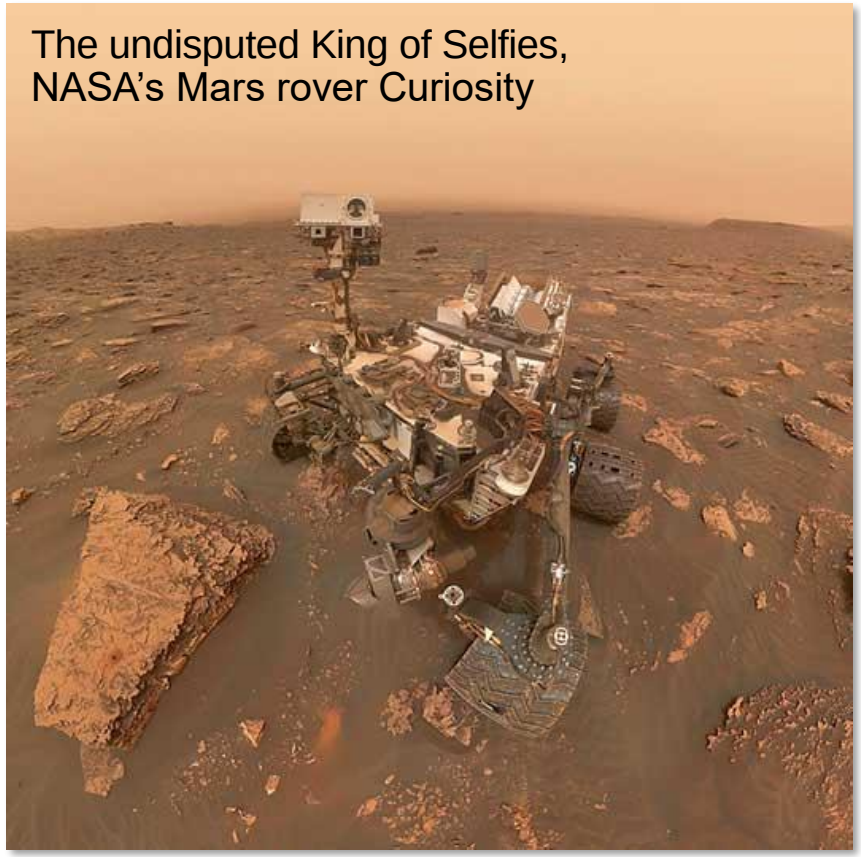
Mission status report

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

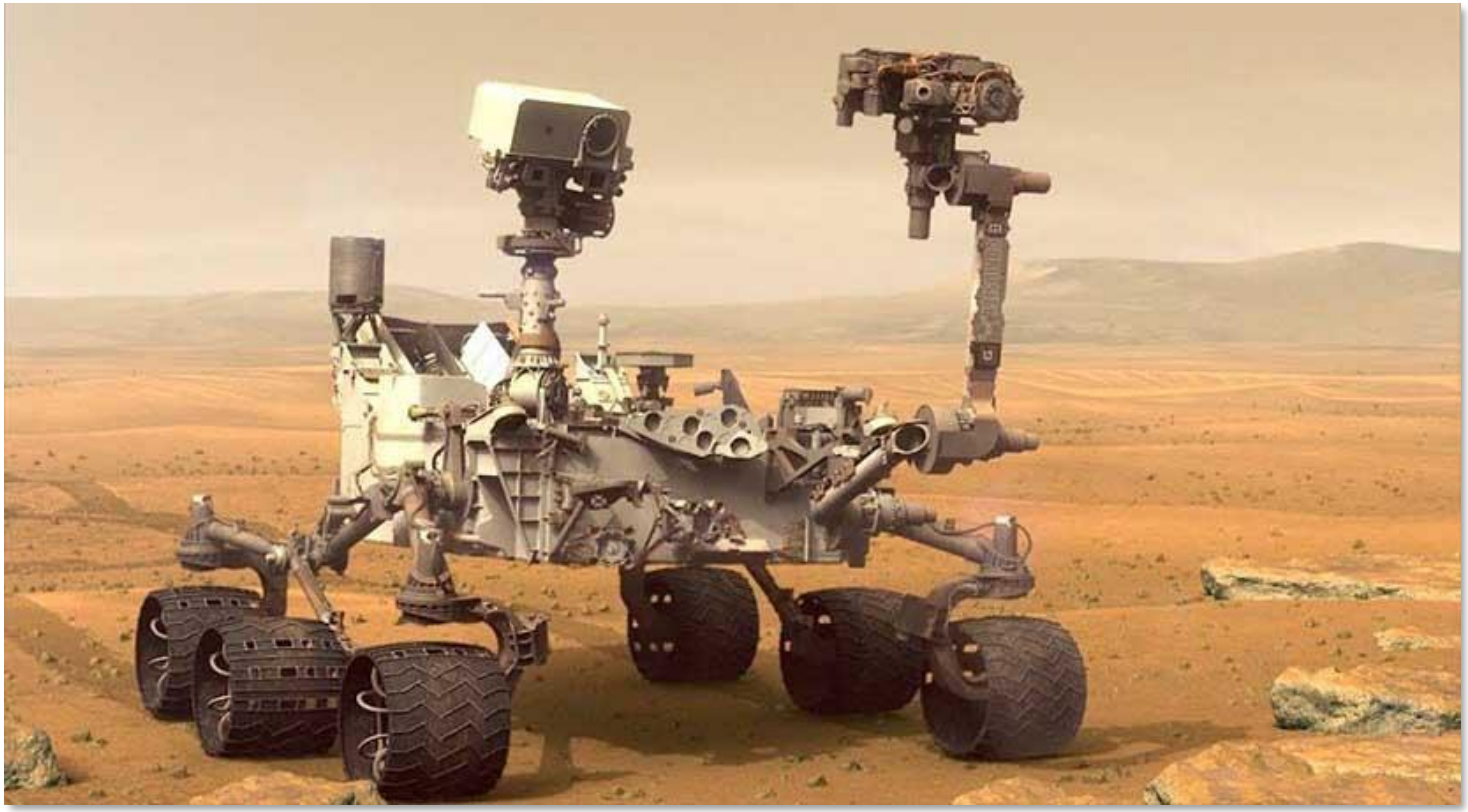


"The Curiosity Rover in the middle of a Martian dust storm. Much like most of the photography shared by the rover, it looks simultaneously real and otherworldly.[...] And for the nit-pickers among you, looking for the selfie stick and wondering, *how the hell is Curiosity even taking this photo?*"

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!

"Schiaparelli" hatte Softwarefehler

Der Absturz der Mars-Sonde "Schiaparelli" auf dem Roten Planeten vor gut einem Monat wurde laut Europäischer Weltraumagentur (ESA) durch einen Software-Fehler beim Navigationsprogramm verursacht. Wie der ESA-Verantwortliche Thierry Blancquaert der Nachrichtenagentur AFP mitteilte, registrierte die Sonde aufgrund einer Reihe falscher Messungen eine negative Höhe, als sie noch mehrere Kilometer über der Oberfläche war.

Wegen dieser falschen Berechnungen war die Sonde schließlich mit 540 Stundenkilometern auf dem Mars aufgeschlagen.

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

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 Stanisław 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: 2017

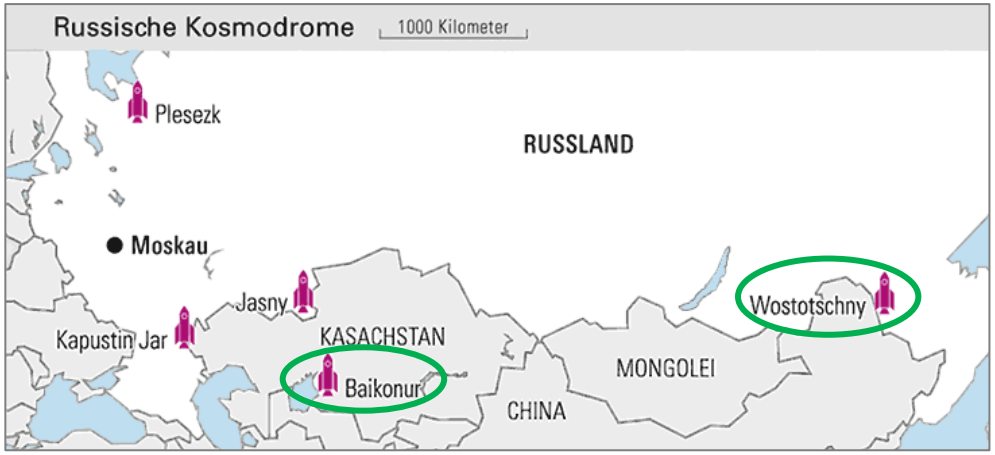
Sojus-2.1b: Wettersatellit Meteor-M – Falsche Startkoordinaten

NZZ, 28.12.2017

Peinlicher Programmierfehler führt zum Verlust einer russischen Weltraumrakete

Russland hat ein neues Kosmodrom eröffnet, aber bei einem Raketenstart im November irrtümlich die Koordinaten des alten Weltraumbahnhofs Baikonur verwendet.

Am 28. November hatte Russland in Wostotschny eine Trägerrakete des Typs Sojus-2.1b mit dem Wettersatelliten Meteor-M gestartet. Die Nutzlast umfasste auch 18 kleinere Satelliten, die zumeist von westlichen Kunden stammten. Doch kurz danach musste die Raumfahrtbehörde Roskosmos einen Misserfolg bekanntgeben. Es sei nicht gelungen, Kontakt mit den Satelliten aufzunehmen. Die Rakete habe diese nicht auf die angestrebte Umlaufbahn gebracht. Roskosmos geht inzwischen davon aus, dass die Rakete über dem Nordatlantik abgestürzt ist.



www.nzz.ch/international/peinlicher-programmierfehler-fuehrt-zu-verlust-einer-russischen-weltraumrakete-ld.1343124

Der russische Vizeregierungschef Dmitri Rogosin hat eine Erklärung für die Panne beim Satellitenstart von Ende November geliefert: Die Flugbahnberechnung sei nicht an den neuen Weltraumbahnhof Wostotschny angepasst gewesen. Die Einstellungen der Rakete seien jene für das Kosmodrom Baikonur in Kasachstan gewesen, sagte Rogosin am Mittwoch.

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).

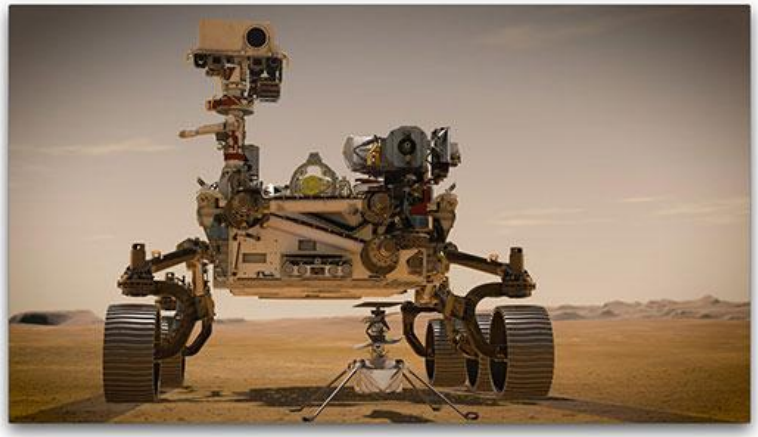
April 11, 2019: “We had a failure of the spacecraft,” said Opher Doron, “We unfortunately have not managed to land successfully.” A chain of events caused by a command sent from the SpaceIL control room turned off the spacecraft’s main engine and prevented proper engine activation, making a crash landing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

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

Andere Softwareprobleme im Weltraum: 2021

Mars Helicopter: Watchdog timer expiration

NASA's Ingenuity Mars Helicopter is the first aircraft humanity has sent to another planet to attempt powered, controlled flight. The helicopter is attached to the belly of NASA's Mars 2020 Perseverance rover. They landed together in Jezero Crater on Feb. 18, 2021.



https://d2pn8kiwq2w21t.cloudfront.net/original_images/jpegPIA23962.jpg

Perseverance and Ingenuity (Artist's Concept).

STATUS UPDATE | **April 10, 2021**

During a high-speed spin test of the rotors on Friday, the command sequence controlling the test ended early due to a "watchdog" timer expiration. This occurred as it was trying to transition the flight computer from 'Pre-Flight' to 'Flight' mode. The helicopter team is reviewing telemetry to diagnose and understand the issue. Following that, they will reschedule the full-speed test.

https://mars.nasa.gov/images/mepjpl/heliocopter-1200.jpg

STATUS UPDATE | **April 12, 2021**

The Ingenuity team has identified a software solution for the command sequence issue identified on Sol 49 (April 9). Over the weekend, the team considered and tested multiple potential solutions to this issue, concluding that minor modification and reinstallation of Ingenuity's flight control software is the most robust path forward. This software update will modify the process by which the two flight controllers boot up, allowing the hardware and software to safely transition to the flight state. Modifications to the flight software are being independently reviewed and validated today and tomorrow in testbeds at JPL.

Quelle aller Status Updates: https://mars.nasa.gov/technology/helicopter/status/

Mars Helicopter (2021)

“Have you tried turning it off and back on again?”

…just happens in space

https://twitter.com/NASA/status/1382003115554181125

April 14, 2021

MiMi Aung, Ingenuity project manager, called the shutdown of the high-speed rotor test “a surprise.” She explained the shutdown this way: “The issue is when the helicopter mode goes from pre-flight mode to flight mode it checks to make sure the helicopter flight computer is healthy. So there is the watchdog that looks at the computers…and saying they’re good. There was a subtle timing issue. The flight computers were working, watchdog was working properly, but it was a slight timing issue with the stroke from the computers arriving to the watchdog that led the watchdog to say the computers aren’t healthy, even though they were.”

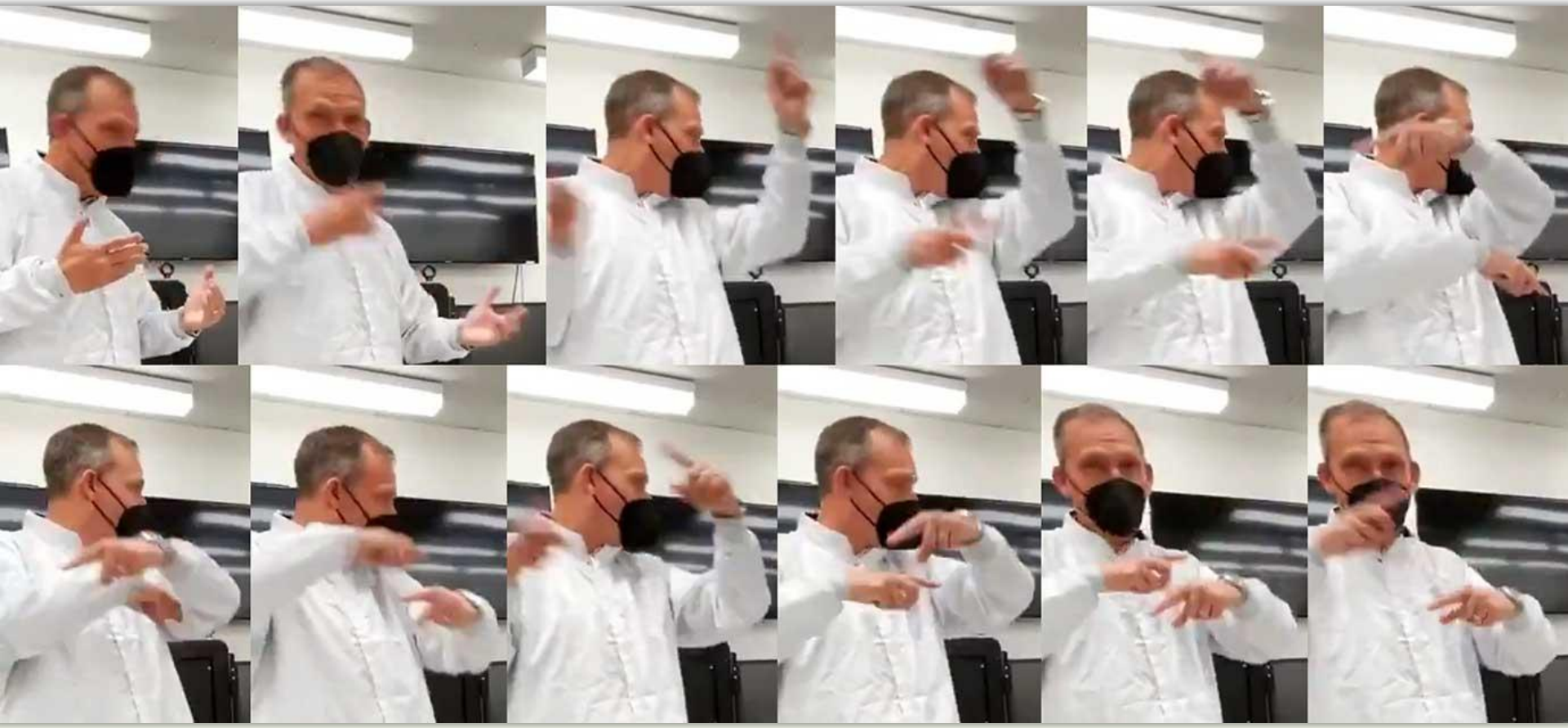
“It’s a very, very tiny mismatch that varies very, very slightly and so that’s why we have just encountered it,” she added. “We have tested this exact environment on Earth and in that time it did not occur. It is a very subtle issue.” Aung was joined by Thomas Zurbuchen, associate NASA administrator, both of them wearing lab coats and masks in a JPL lab setting. “These things really do happen,” he said, “it just happens in space.”

Quelle: www.fierceelectronics.com/electronics/ingenuity-gets-a-software-fix-first-flight-now-expected-next-week

Mars Helicopter (2021)

Direktor Zurbuchen stellt das verwickelte Problem anschaulich dar

“Have you tried slapping it? My TV remote starts working perfectly when I slap it.”

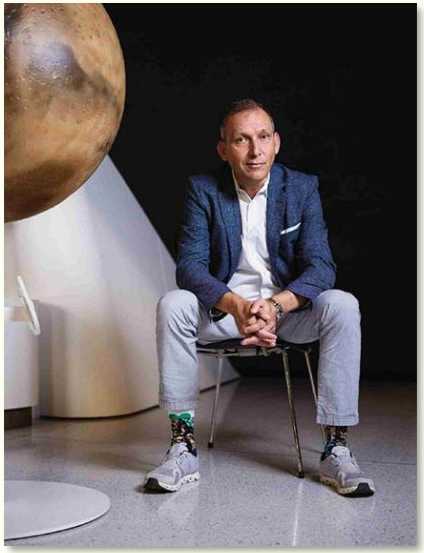


Thomas Zurbuchen, schweizerisch-amerikanischer Doppelbürger, ist der Sohn eines Freikirchen-Predigers aus dem Berner Oberland („der grösste Schritt in meinem Leben war von meinem Dorf Heiligenschwendi runter nach Thun“). Er studierte an der Universität Bern Physik und doktorierte dort 1996 in experimenteller Astrophysik. 2008 wurde er Professor an der University of Michigan und 2016 wissenschaftlicher Direktor der NASA. Im August 2023 schliesslich wurde er Professor für Weltraumwissenschaft und -technologie an der ETH Zürich.

Wenn ich Enten zähle, dann auf Berndeutsch, wenn es aber um eine quadratische Gleichung geht, ist es für mich auf Englisch einfacher. -- *Thomas Zurbuchen, 27.5.2023*

Mars Helicopter (2021)

Thomas Zurbuchen wird Ehrendoktor der ETH Zürich (2022)



Die Wochenzeitschrift „Schweizer Familie“ führte aus Anlass der Ehrendoktorwürde ein Interview mit Thomas Zurbuchen. Einige Auszüge daraus:

„Ich freue mich ausserordentlich, weil ich als junger Mann gern hier studiert hätte. Die ETH gehört zu den besten Universitäten der Welt. Aber der Weg vom Berner Bergdorf Heiligenschwendi, wo ich in bescheidenen Verhältnissen aufgewachsen war, nach Zürich an die ETH erschien mir damals zu weit und zu teuer. Zudem fürchtete ich, nicht gut genug zu sein für diese Universität.

Bei meinem ersten Launch im Jahr 2016 sassen neben mir auf dem Podest vor den TV-Kameras nur weisse Männer. Ich schämte mich in Grund und Boden für das Bild, das wir dort oben abgaben, und sorgte dafür, dass ab sofort immer mindestens eine Frau auf dem Podest sass.

Nach über fünfzig Jahren wollen wir wieder Menschen zum Mond bringen, und von dort soll es dereinst weitergehen zum Mars. Die Trägerrakete, die letzte Woche startete, ist die stärkste je gebaute in der Geschichte der Raumfahrt. Darum war der Launch, also der Raketenstart, ein wichtiger Meilenstein in meiner Karriere. Bei einem Launch bin ich sehr nervös. Ich weiss, wie viel schieflaufen könnte.

Ich begegne immer wieder erfolgreichen Menschen, die aus unterschiedlichen Gründen mit inneren Dämonen kämpfen und sich fragen: Bin ich gut genug? Die Angst, zu versagen, treibt sie zu Höchstleistungen an. […] Ich kenne die Zweifel gut. Aus meiner kleinen, christlichen Welt in den Bergen war ich als Teenager hinunter ins Gymnasium im Tal gekommen, später an die Uni in Bern und so weiter. Ich betrat neue Welten und war unsicher, ob ich dort hingehörte. Um es zu beweisen, arbeitete ich oft mehr als andere. Über solche Muster redet niemand gern, aber sie sind weitverbreitet.“

In den USA nannten sie ihn «Dr. Z» – weil niemand seinen Namen aussprechen konnte. -- *Coopzeitung, 12.9.2023*

Mars Helicopter (2021)

Thomas Zurbuchen wird Ehrendoktor der ETH Zürich (2022)

Bild: ETH Zürich

Ab August 2023 leitet Zurbuchen, der eine Hündin mit dem Namen „Luna“ hat, als ETH-Professor für Weltraumwissenschaft und -technologie die Initiative „ETH Zürich Space“. Diese Initiative wurde im Oktober 2022 gegründet, um Forschung und Lehre auf dem Gebiet der Weltraumwissenschaft an der ETH auszubauen.

Anlässlich seiner Wahl sagte Zurbuchen: „Der Raumfahrtsektor erfährt weltweit eine rasante Dynamik. Wir wollen sicherstellen, dass die Schweiz und Europa die neuen Chancen nutzen, wettbewerbsfähig bleiben und ihre internationale Sichtbarkeit erhöhen. […] Ich möchte einen der weltweit besten interdisziplinären Masterstudiengänge in Weltraumwissenschaft und -technologie lancieren. […] Ich habe nach meinem Abschied von der Nasa nicht nur mit der ETH Zürich Gespräche geführt, sondern auch mit der ETH Lausanne und der Universität Bern. Ich habe denen gesagt, dass ich hier in Zürich das grösste Potenzial sehe. […] Vor allem aber ist an der ETH Zürich sehr viel technisches und elektronisches Know-how vorhanden. Wenn es uns gelingt, all diese Aktivitäten zu bündeln, können wir sehr viel bewegen. […] Die Tatsache, dass ich mit meiner Frau, meinen Kindern und meinem Hund in die Schweiz ziehe, zeigt, dass ich hier meinen Lebensmittelpunkt sehe.“

Mars Helicopter (2021)
Engineers believe in back-up plans

"It is fortunate that the helo doesn't run on Windows. Otherwise, those software upgrades would consist of stuffing advertisements onto the platform." – Pete, https://forums.theregister.com/forum/

STATUS UPDATE | **April 17, 2021**

Over the last week, we've been testing the two solutions to address the "watchdog" timer issue that prevented the helicopter from transitioning to "flight mode" […]. These solutions, which have each been verified for use in flight are: 1) adjusting the command sequence from Earth to slightly alter the timing of this transition, and 2) modifying and reinstalling the existing flight control software […]. The first solution requires adding a few commands to the flight operations sequence and has been tested on both Earth and Mars. From testing this technique on Ingenuity over the last few days, we know this approach is likely to allow us to transition to flight mode and prepare for lift-off about 85% of the time. […]

On Friday, we employed this solution to perform our first-ever high-speed spin test on Mars. […] We also know that if the first attempt does not work on Monday, we can try these commands again, with good probability that subsequent tries in the days following would work even if the first doesn't. For these reasons, we've chosen to pursue this path.

Because engineers believe in back-up plans, we have also been working this week on the second solution, which requires modification and reinstallation of Ingenuity's flight control software. The new software for this solution has been transmitted to NASA's Perseverance rover, which hosts the helicopter base station that ultimately communicates with Ingenuity. If our initial approach to flight does not work, the rover will send the new flight control software to the helicopter. We will then require several additional days of preparation to load and test the new software on Ingenuity.

https://mars.nasa.gov/resources/25790/perseverances-selfie-with-ingenuity/

The Mars rover took a selfie with the helicopter, seen here about 3.9 meters from the rover in this image taken by the camera located at the end of the rover's long robotic arm.

Mars Helicopter (2021)
Helicopter succeeds in historic first flight

RELEASE 21-039 | **April 19, 2021**

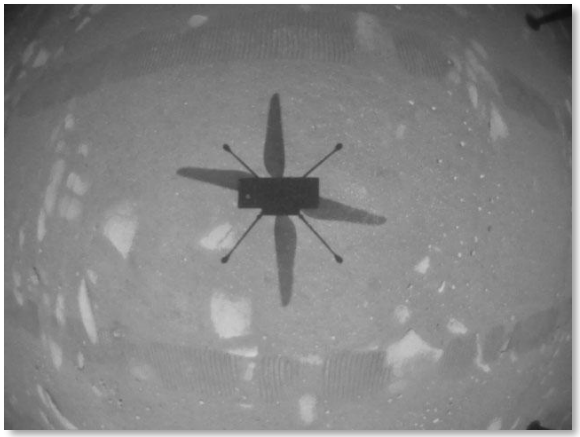
Monday, NASA's Ingenuity Mars Helicopter became the first aircraft in history to make a powered, controlled flight on another planet. The solar-powered helicopter first became airborne at 12:33 Local Mean Solar Time (Mars time), a time the Ingenuity team determined would have optimal energy and flight conditions. Altimeter data indicate Ingenuity climbed to its prescribed maximum altitude of 3 meters and maintained a stable hover for 30 seconds. It then descended, touching back down on the surface of Mars after logging a total of 39.1 seconds of flight.

The Red Planet has a significantly lower gravity – one-third that of Earth's – and an extremely thin atmosphere with only 1% the pressure at the surface compared to our planet. This means there are relatively few air molecules with which Ingenuity's two 1.2-meter-wide rotor blades can interact to achieve flight.

Parked about 64.3 meters away during Ingenuity's historic first flight, the Perseverance rover not only acted as a communications relay between the helicopter and Earth, but also chronicled the flight operations with its cameras.

https://mars.nasa.gov/multimedia/images/

Members of NASA's helicopter team in the Space Flight Operations Facility at NASA's Jet Propulsion Laboratory react to data showing that the helicopter completed its first flight.



https://mars.nasa.gov/multimedia/images/

The Mars Helicopter took this shot, capturing its own shadow, while hovering over the Martian surface.

Mars Helicopter (2021)
15% chance for watchdog timer expiration

STATUS UPDATE | **April 29, 2021**

Data received from the Mars Ingenuity helicopter on Thursday morning shows the helicopter did not execute its planned fourth flight as scheduled. […] Data indicates the helicopter did not transition to flight mode, which is required for the flight to take place. An issue identified earlier this month showed a 15% chance for each time the helicopter attempts to fly that it would encounter a watchdog timer expiration and not transition to flight mode. Today's delay is in line with that expectation and does not prevent future flights.

https://scitechdaily.com/images/NASA-Ingenuity-Mars-Helicopter-Fourth-Flight.jpg

NASA's Mars Perseverance rover acquired this image of the Ingenuity Mars Helicopter (upper right) using its left Mastcam-Z camera.

https://mars.nasa.gov/internal_resources/1044/

During its fourth flight, the Ingenuity Mars Helicopter caught a glimpse of Perseverance from the air.

STATUS UPDATE | **April 30, 2021**

Ingenuity successfully completed its fourth flight today, and we couldn't be happier. […] In total, we were in the air for 117 seconds. During this flight, we saved even more images than we did on our previous flights: about 60 total during the last 50 meters before the helicopter returned to its landing site.

Mars Helicopter (2021)
Fixing the Watchdog Issue / Updating the Navigation Software

STATUS UPDATE | **June 25, 2021**

[…] The issue stems from a computer protection and reliability feature between our Flight Controller microcontrollers and our Field Programmable Gate Array (FPGA). When attempting to transition to the "flight-state," the watchdog on the FPGA would detect violations to its strict timing requirements. […]

https://pbs.twimg.com/media/EyOC_7YU8AAevDd?format=jpg

The Ingenuity team is proud to say that last week we completed a flight software update of the Flight Controller microcontrollers on the helicopter, with the intent of permanently fixing the watchdog issue. This patch provides much needed reliability in the operations demonstration, ensuring that the heli and rover teams can plan for successful flights in the future. […] Flight 8 confirmed that the flight controller software update was a success.

Up next for the Ingenuity team is to tackle the only remaining flight software update, which will update a large portion of the Ingenuity's navigation-computer software. This update will address the Flight 6 anomaly, where image timing delays manifested into aircraft estimation and control challenges. […] The team's hypothesis is that the large CPU load involved in capturing the 13-megapixel color images, could result in rare instances of navigation camera images being dropped in the pipeline. Those nav. camera image drops are what caused the Flight 6 anomaly***)**. That is why Flights 7 and 8 did not have any color images captured. This update will provide a mechanism for the helicopter to detect and correct when image timestamps in the pipeline become out-of-synch/are dropped/skipped, while also re-enabling the capture of high-resolution 13-megapixel color images.

***)** About 54 seconds into the flight, a small glitch occurred in the transition of navigation images to the helicopter's computer. The chopper lost just one image, but that meant that each following photo was delivered with the wrong timestamp. That made the helicopter roll and pitch.

Mars Helicopter (2021)

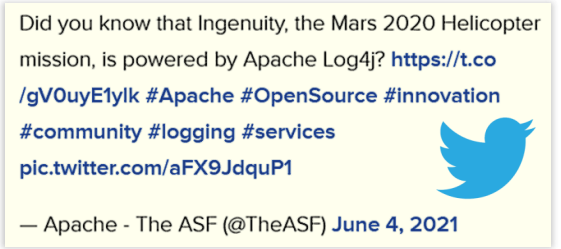
Mars helicopter could be vulnerable to log4j


... sich das alles überhaupt an? Er grinst und erzählt von *Ingenuity*, dem Nasa-Helikopter, der auch das Logging-Programm Log4j verwendet: „Meine Software läuft auf dem Mars. Ich persönlich werde es wahrscheinlich nie dorthin schaffen. Aber es ist einfach ein ziemlich gutes Gefühl, dass ein paar Zeilen Code, die ich geschrieben habe, auf dem Mars sind."

December 16, 2021 www.techradar.com, https://futurism.com/

The flawed logger powers the most remote helicopter ever built.

The panic over Log4Shell, the now infamous log4j vulnerability, has spread to Mars, where strange behavior by the NASA Ingenuity helicopter was a cause for concern. The Ingenuity helicopter took off for the 17th time on December 5th. During the descent, however, there was an "unexpected interruption of the in-flight data stream," which meant that the NASA team did not know exactly what the status of the spacecraft was, the organization said.



Since the incident coincided with the disclosure of the log4j vulnerability, some people have started connecting the dots. The theory arose from a Twitter post by the Apache Software Foundation in June that said the helicopter mission is "powered by Apache log4j". Log4j is a Java logger that was recently discovered to hold a critical flaw, which allows malicious actors (even those with very little skill) to run arbitrary code on millions of endpoints, and push out malware, ransomware and crypto-miners. Since this week's press release celebrating the copter's flight, NASA hasn't released any more statements, and the original didn't say anything about Log4j. We hope that even if they aren't saying anything publicly, they're upping security measures behind the scenes.

June 6, 2022 JPL reported Ingenuity's inclination sensor had stopped working. Its purpose was to determine the helicopter's orientation at the start of each flight. Mission controllers developed a workaround using the craft's inertial measurement unit (IMU) to provide equivalent data to the onboard navigation computer.

June 7, 2022 NASA is sending a software update to grounded Mars helicopter so that one sensor can impersonate a broken one.

Mars Helicopter (2021)
Helicopter Mission Ends

Beim Mars-Helikopter habe ich am Schluss zwei meiner grössten Kritiker in eine Kommission gesetzt, die herausfinden musste, ob die Maschine auf dem Mars tatsächlich fliegen kann. Als einer von ihnen zu mir kam und sagte, «du hast recht, das geht», wusste ich, dass wir es wagen können. -- *Thomas Zurbuchen, 10.02.2024*

November 23, 2022

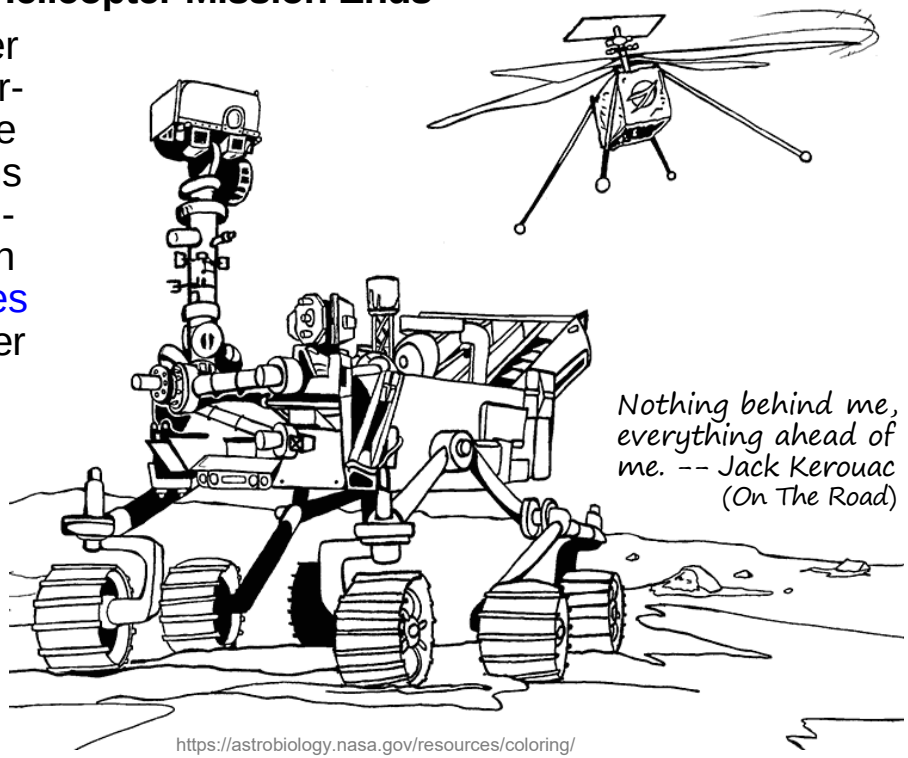
Over the past few weeks, the operations team has been at work installing a major software update aboard the helicopter. This update provides Ingenuity two major new capabilities: hazard avoidance when landing and the use of digital elevation maps to help navigate.

January 25, 2024

After Three Years on Mars, NASA's Ingenuity Helicopter Mission Ends

NASA's history-making Ingenuity Mars Helicopter has ended its mission at the Red Planet after surpassing expectations and making dozens more flights than planned. While the helicopter remains upright and in communication with ground controllers, imagery of its Jan. 18 flight sent to Earth this week indicates one or more of its rotor blades sustained damage during landing and it is no longer capable of flight.

Originally designed as a technology demonstration to perform up to five experimental test flights over 30 days, the first aircraft on another world operated from the Martian surface for almost three years, performed 72 flights, and flew more than 14 times farther than planned while logging more than two hours of total flight time.



https://astrobiology.nasa.gov/resources/coloring/

Andere Softwareprobleme im Weltraum: 2023

Hakuto-R lunar lander: The software did not perform as expected

Die Hakuto-R-Mondsonde des privaten japanischen Raumfahrtunternehmens ispace wurde am 11. Dezember 2022 gestartet. Der Lander schlug am 25. April 2023 nach 88 Sekunden freiem Fall mit etwa 556 km/h auf der Mondoberfläche auf. Einen Monat später gab ispace bekannt:

"[…] During the period of descent, an unexpected behavior occurred with the lander's altitude measurement. While the lander estimated its own altitude to be zero, or on the lunar surface, it was later determined to be at an altitude of approximately 5 kms above the lunar surface. After reaching the scheduled landing time, the lander continued to descend at a low speed until the propulsion system ran out of fuel. At that time, the controlled descent of the lander ceased, and it is believed to have free-fallen to the Moon's surface.

The most likely reason for the lander's incorrect altitude estimation was that the software did not perform as expected. Based on the review of the flight data, it was observed that, as the lander was navigating to the planned landing site, the altitude measured by the onboard sensors rose sharply when it passed over a large cliff approximately 3 kms in elevation on the lunar surface, which was determined to be the rim of a crater. According to the analysis of the flight data, a larger-than-expected discrepancy occurred between the measured altitude value and the estimated altitude value set in advance. The onboard software determined in error that the cause of this discrepancy was an abnormal value reported by the sensor, and thereafter the altitude data measured by the sensor was intercepted. This filter function, designed to reject an altitude measurement having a large gap from the lander's estimation, was included as a robust measure to maintain stable operation of the lander in the event of a hardware issue including an incorrect altitude measurement by the sensor. […]

The analysis reveals that the cause of the lander's failure to make a soft landing was due to the software, especially in the phase just prior to landing. This information will be incorporated into software design. […] 'Never Quit the Lunar Quest.' In this spirit, we will continue to move forward."

Keine Panik; es war ja nur die Software!

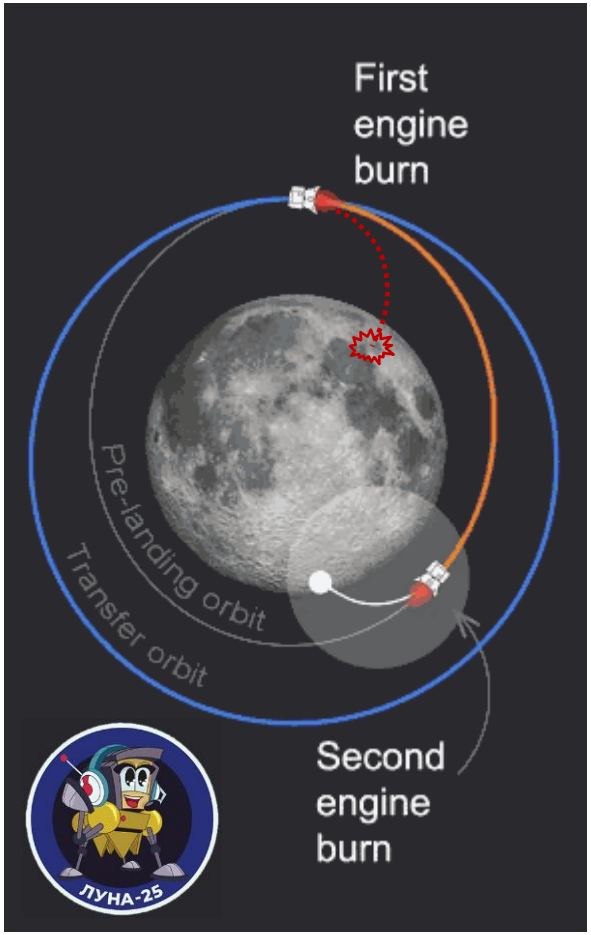
Andere Softwareprobleme im Weltraum: 2023

Luna-25: Erste russische Mondmission nach 50 Jahren gescheitert

Der österreichische „Standard“ meldete am 21. August 2023: „Was die erste russische Mondlandung seit fast 50 Jahren hätte werden sollen, endete am Wochenende mit einem Crash. Luna-25 sei in eine falsche Umlaufbahn geraten und auf dem Mond zerschellt, wie die russische Raumfahrtbehörde Roskosmos am Sonntag mitteilte.“

Die unbemannte Raumsonde wurde am 11. August gestartet. Nachdem sie eine vorläufige Mondumlaufbahn („transfer orbit“) erreichte, wurde am 19. August ein Manöver eingeleitet, das die Sonde in den für die Landung nötigen elliptischen Orbit bringen sollte. Dies schlug wegen einer Panne bei der Triebwerksteuerung fehl; anstatt der beabsichtigten ca. 84 Sekunden waren die Bremsdüsen 127 Sekunden in Betrieb und brachten durch die zu starke Verzögerung die Sonde zum Absturz.

Es handelte sich dabei anscheinend um ein Softwareproblem im „Bordsteuerungskomplex“: Die Sonde ist mit zwei redundanten Beschleunigungssensoren ausgestattet. Ein Sensor fiel (wie schon einmal früher während des Flugs) während des Manövers aus, das Flugsteuersystem hatte jedoch den zweiten Sensor schon im Vorfeld gar nicht erst aktiviert und erhielt daher keinerlei Daten zur Geschwindigkeit, Lage und Höhe. Dadurch konnte die Antriebseinheit nicht planmässig im richtigen Moment abgestellt werden, dies geschah erst deutlich zu spät durch einen „emergency timer“.



Ende der historischen Notiz

Andere Softwareprobleme…

„Das Forum sieht seine Aufgabe darin, Wissen und Erfahrung über die Wirkung der Informationstechnik auf Gesellschaft und Umwelt zusammenzutragen, zu erarbeiten und zur Diskussion zu stellen.“ -- Aus der FIFF-Gründungserklärung

Ein fehlerhaftes Zeichen in 7-Segment-Darstellung

Rundbrief 1, 1986, S. 16

Das „Forum Informatiker für Frieden und gesellschaftliche Verantwortung“ („FIFF“) wurde 1984 gegründet und ging aus der damaligen deutschen Friedensbewegung (Engagement für atomare Abrüstung anstelle des Wettrüstens NATO / Warschauer Pakt) hervor. In Zeiten, als E-Mail und Internet noch nicht so verbreitet waren, hielt ein Rundbrief die Mitglieder auf dem Laufenden.

Ein besorgniserregendes Thema war anfangs vor allem der „Atomkrieg aus Versehen“ aufgrund eines Softwarefehlers (oder auch Hardwarefehlers, etwa eines defekten Sensors). Laien, Politikern und Medien war vor allem die Softwareproblematik seinerzeit nicht bewusst.

Man sollte auch heute nicht davon ausgehen, dass Software in immer komplexeren Kommunikations- und Waffensystemen grundsätzlich zuverlässiger ist als Software im Weltraum.

K.H. Bläsius schrieb im Januar 2023: „Die nukleare Abschreckungsstrategie beinhaltet auch den Betrieb von Frühwarnsystemen zur Erkennung eines Angriffs mit Atomwaffen. Hierbei kann es aber zu Fehlalarmen kommen, bei denen nukleare Angriffe gemeldet werden, obwohl kein Angriff vorliegt. In der Vergangenheit gab es einige Situationen, in denen es nur durch großes Glück nicht zu einem Atomkrieg aus Versehen kam. Die Abschreckungsstrategie schützt also nicht vor einem ‚Atomkrieg aus Versehen‘. [...] In Zusammenhang mit Atomwaffen können in Frühwarn- und Entscheidungssystemen *plötzlich, völlig unerwartet und ohne Vorwarnung* innerhalb weniger Minuten Prozesse ablaufen, die zum Einsatz vieler Atomwaffen führen und das *Überleben der gesamten Menschheit* gefährden. Solche Prozesse sind *irreversibel*, die gefallenen Entscheidungen endgültig. Es gibt hinterher keine Möglichkeiten mehr zu einer Korrektur.“

Andere Softwareprobleme…

> The way we build and ship software these days is mostly ridiculous, leading to apps using millions of lines of code to open a garage door. -- Bert Hubert.

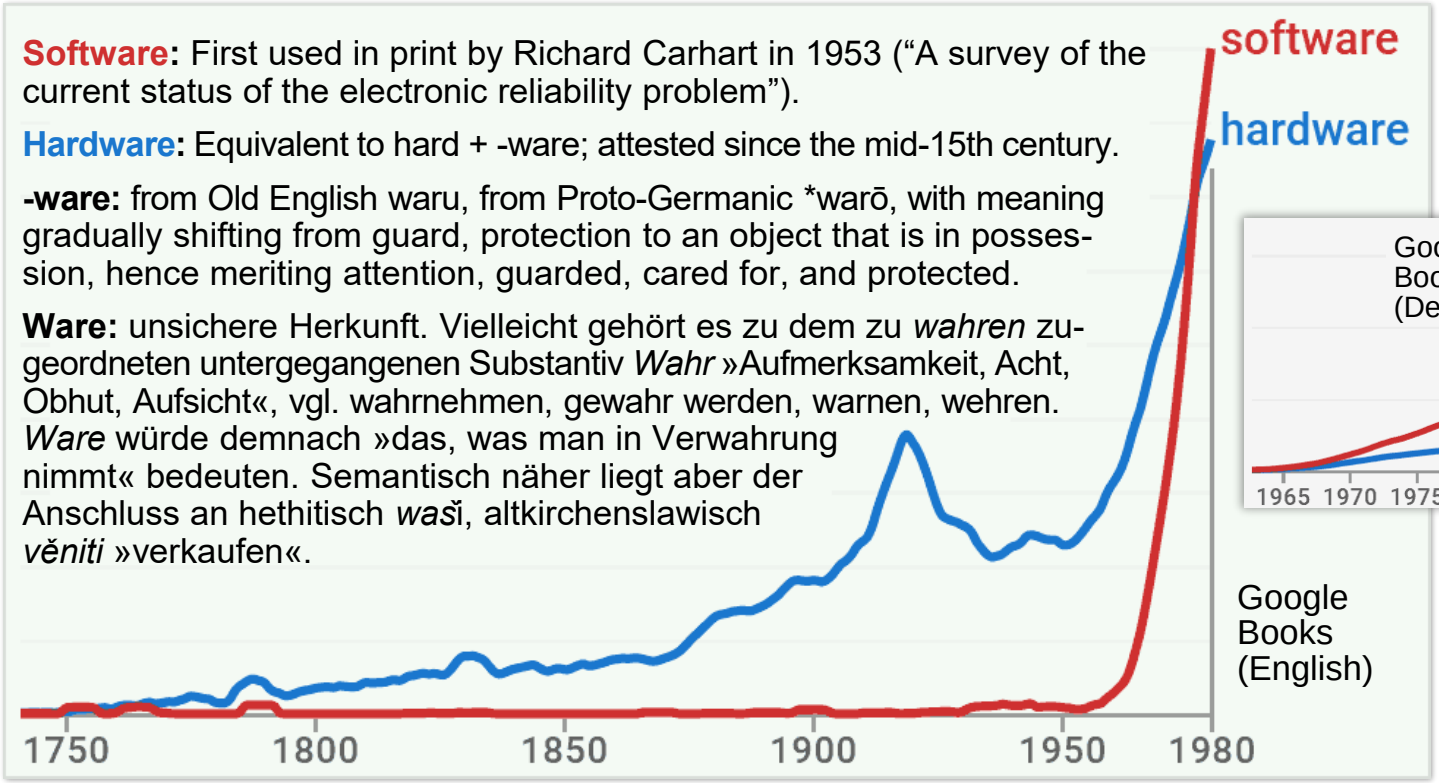
Auf den letzten slides war viel von Software und Softwareproblemen die Rede. “Software is eating the world” lautet der erste Satz des vielzitierten Essays “Why Software Is Eating The World” des Softwareentwicklers und Investors Marc Andreessen im Wall Street Journal vom 20. August 2011. Dass Software einmal so kritisch und ökonomisch bedeutsam werden würde, konnte sich vor wenigen Jahrzehnten allerdings noch kaum jemand vorstellen. Den Begriff selbst gibt es auch noch gar nicht so lange:

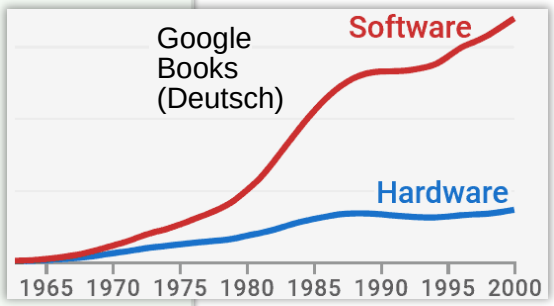
Software: First used in print by Richard Carhart in 1953 (“A survey of the current status of the electronic reliability problem”).

Hardware: Equivalent to hard + -ware; attested since the mid-15th century.

-ware: from Old English waru, from Proto-Germanic *warō, with meaning gradually shifting from guard, protection to an object that is in possession, hence meriting attention, guarded, cared for, and protected.

Ware: unsichere Herkunft. Vielleicht gehört es zu dem zu *wahren* zugeordneten untergegangenen Substantiv *Wahr* »Aufmerksamkeit, Acht, Obhut, Aufsicht«, vgl. wahrnehmen, gewahr werden, warnen, wehren. *Ware* würde demnach »das, was man in Verwahrung nimmt« bedeuten. Semantisch näher liegt aber der Anschluss an hethitisch *wasi*, altkirchenslawisch *věniti* »verkaufen«.





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

Zwi-schen-zeit

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!~~
7:15	Wacht auf	Schläft	Vorlesung aufbrechen
7:30	~~Keine Milch!~~	Schläft	Zum Vorlesungssaal gehen
7:45	Einkaufen gehen	Wacht auf	Vorlesung (gähn)
8:00	Ankunft Laden	~~Keine Milch!~~	Vorlesung (gähn)
8:15	**Milch kaufen**	Einkaufen gehen	**Milch kaufen**
8:30	Heimgehen	Ankunft Laden	Heimgehen
8:45		**Milch kaufen**	
9:00		Heimgehen	
9:15			
9:45			

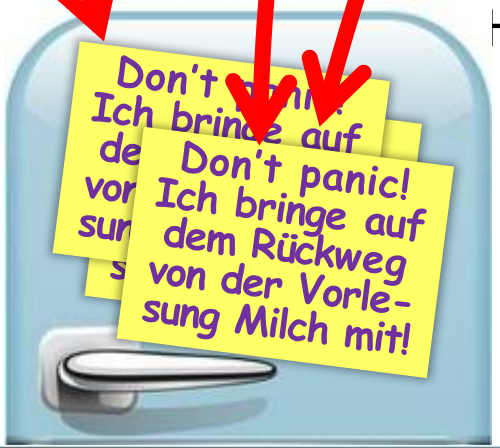
Andere Lösung: Vorsichtshalber Zettel am Kühlschrank?

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



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

...und gehen Milch kaufen!

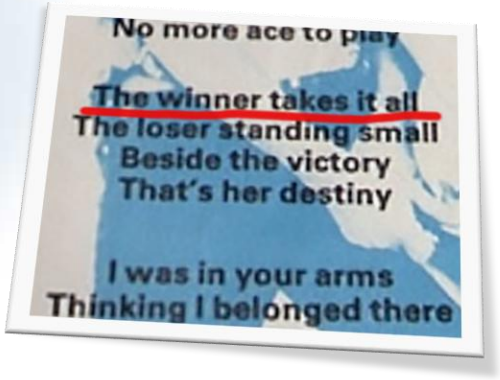
Muss man also auch den Zugriff auf einen „Koordinationsmechanismus" koordinieren?

Race condition
[Wettlaufsituation; kritischer Wettlauf]

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**!

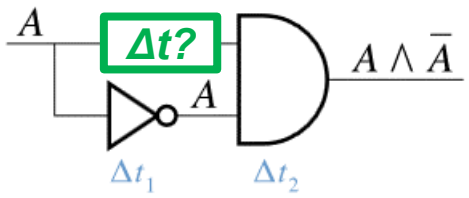


Hazard / race condition bei Logikschaltungen

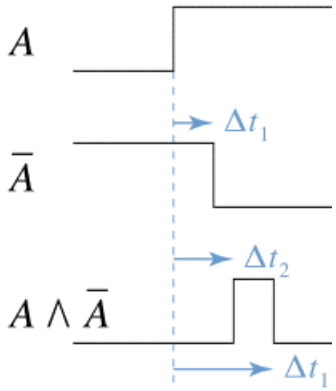
https://en.wikipedia.org/wiki/Race_condition

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…

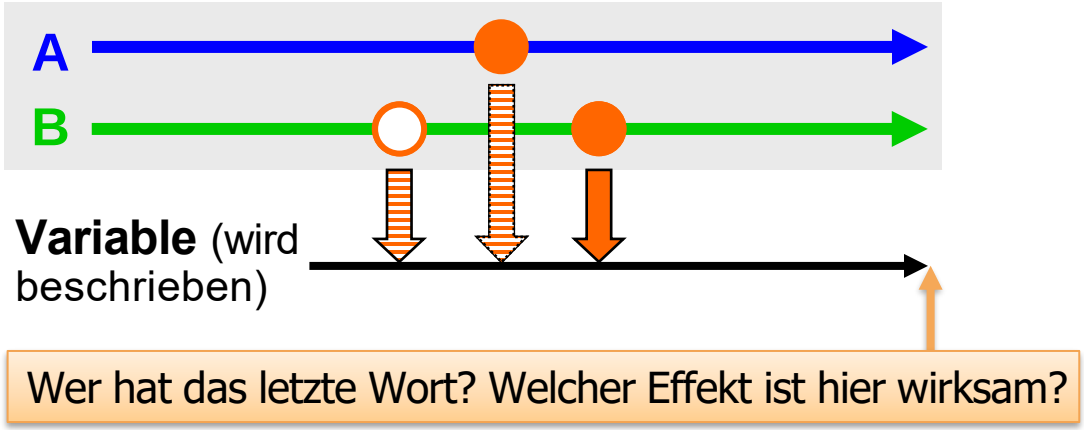


Δt_1 and Δt_2 represent the propagation delays of the logic elements. When the input value *A* changes from low to high, the circuit outputs a short spike.

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



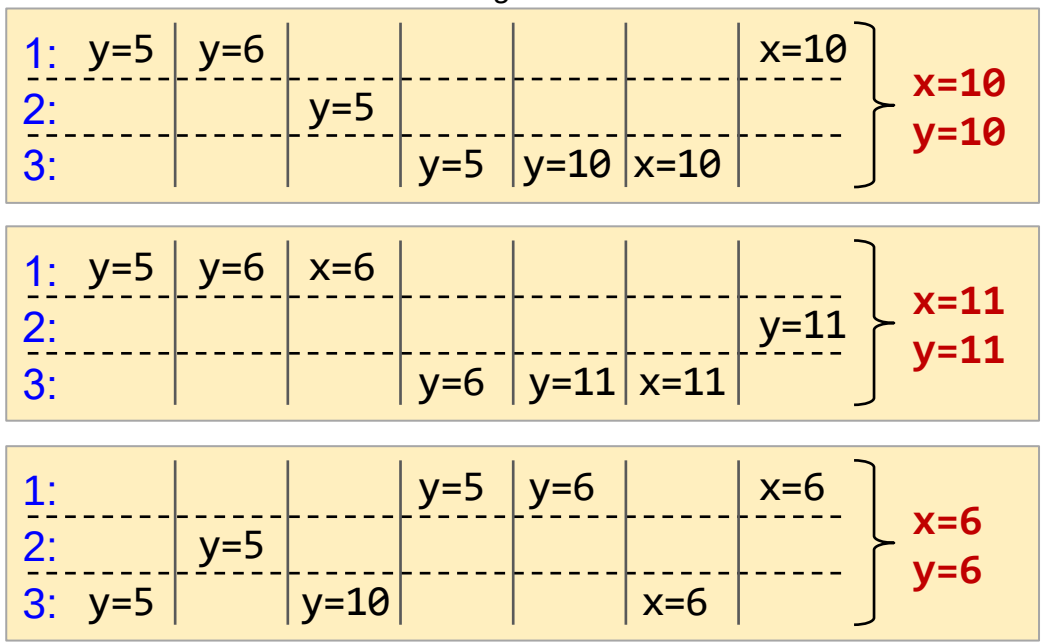
„Unbeabsichtigte Wettlaufsituationen sind ein häufiger Grund für schwer auffindbare Programmfehler; bezeichnend für solche Situationen ist nämlich, dass bereits die veränderten Bedingungen zum Programmtest zu einem völligen Verschwinden der Symptome führen können.“ *de.wikipedia*

The war, after all, is won by those who win the last battle. – Lord Nelson

Ein Beispiel für race conditions

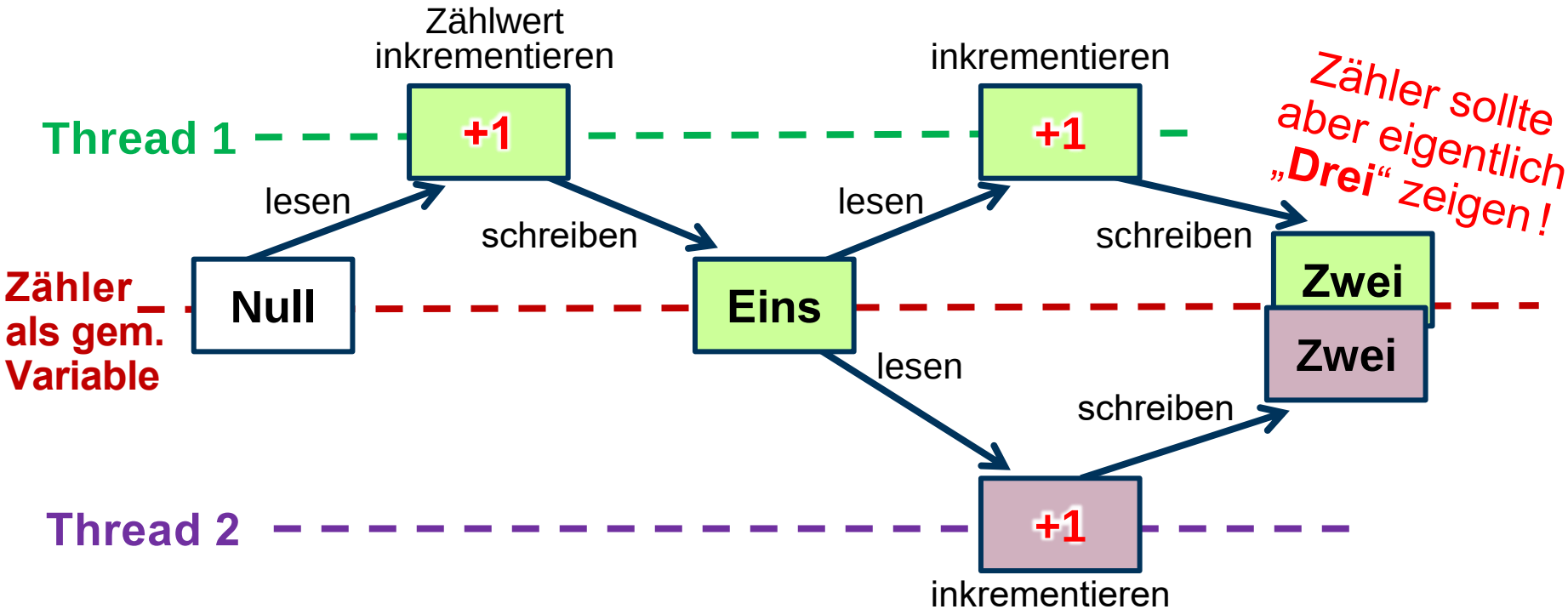
```
int x = 5;  // initial

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

Thread 2:
{ y = x; }

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

Drei verschiedene Schedulings:



- 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 resultiert?

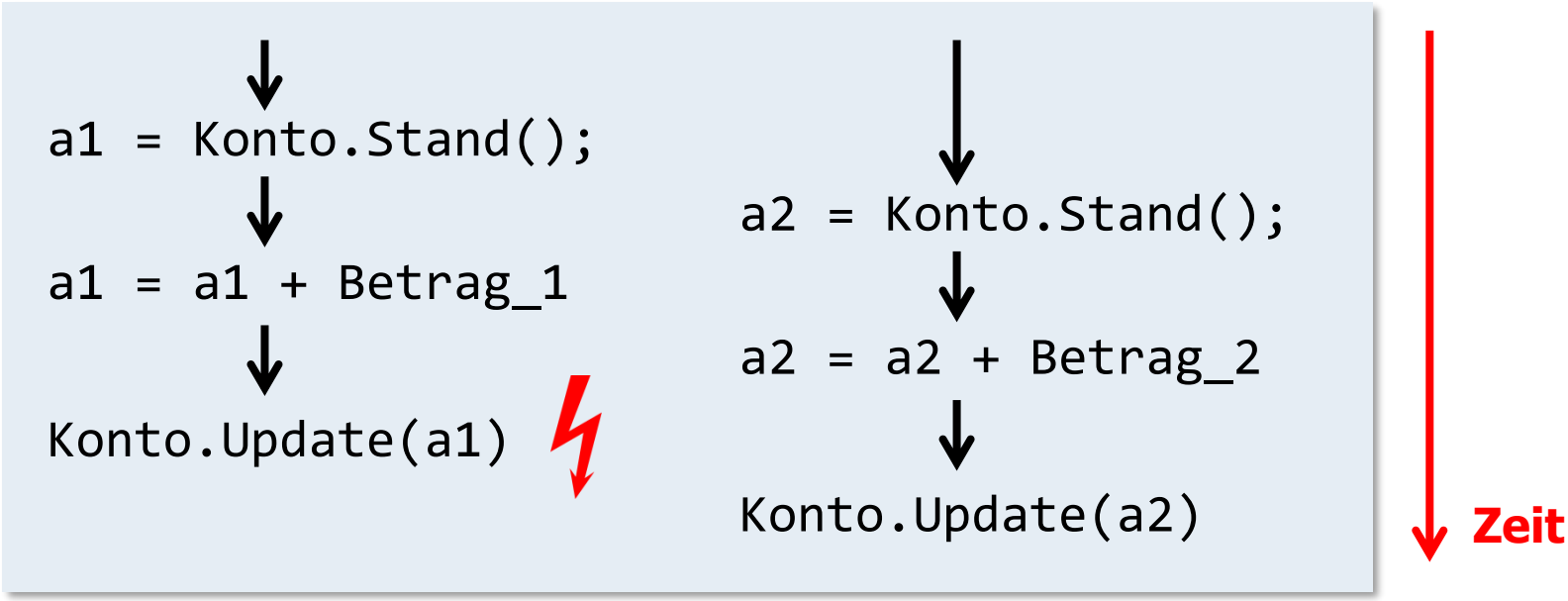
Race condition beim parallelen Zählen



- 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“

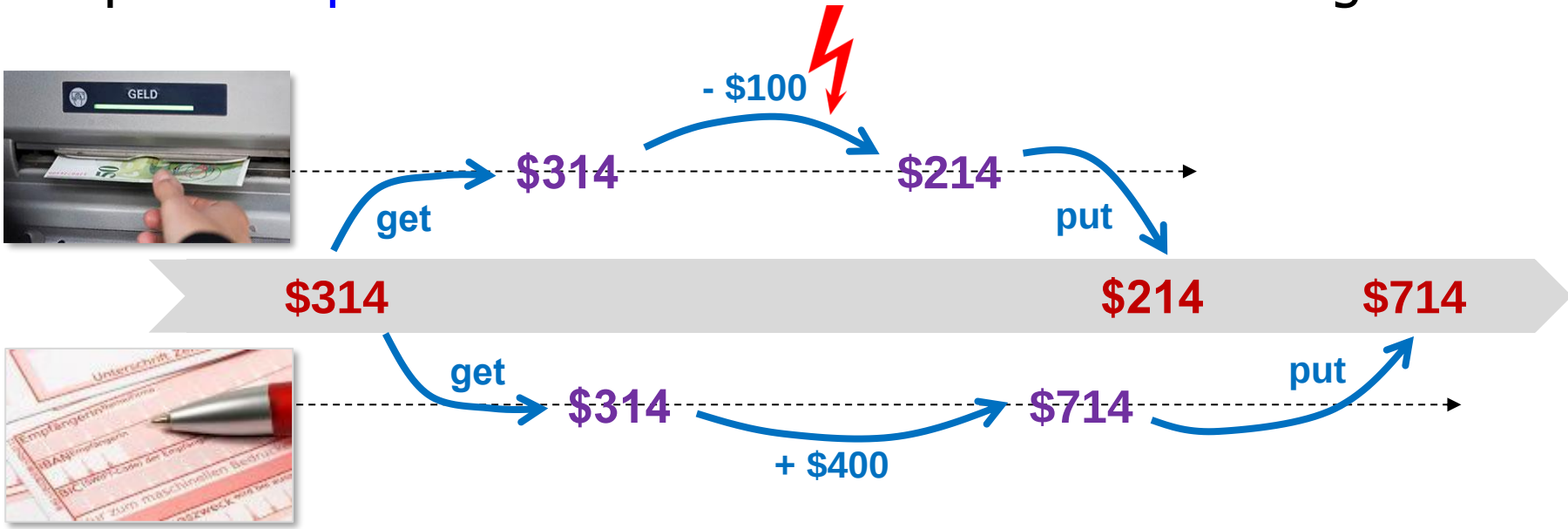
Race condition: „lost update problem“

- 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 einziges Statement zu packen?

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

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

- Oder vielleicht jeweils Aufruf einer Methode „Buchung“?

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

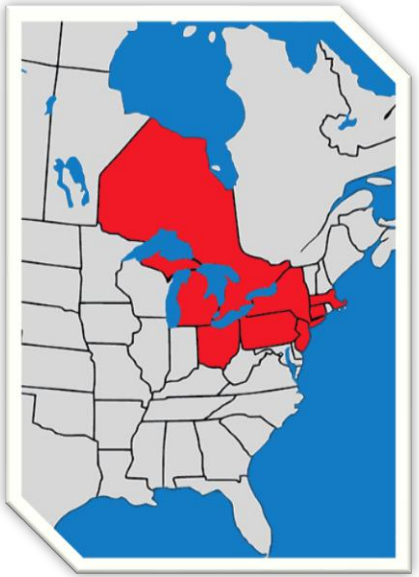
```
...
Buchung(k, -154.23);
...
```

Ist das eigentlich kommutativ?

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

Race condition verursacht Blackout

Race condition verursacht Blackout



- August 2003: Grossflächiger Stromausfall im Nordosten der USA sowie in 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 mittels 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

Eine Hochspannungsleitung berührte einen Baum…

Heated by the sunshine and the flowing current, the transmission lines sag dangerously close to the treetops. Suddenly the current jumps from line to tree branch, finding the path of least resistance and pouring through the tree into the ground. There's a bright flash as the current ionizes the air.

https://jobs.tva.com/power/rightofway/images/high_cost_tree.jpg

During this short circuit, the abruptly unleashed current reaches 10 to 20 times its normal level within a blink of an eye. Now the power grid's protection system must act fast. Within milliseconds, protection relays must recognize the fault and command the circuit breakers at both ends of the line to switch off the current, isolating the faulted line. The stakes are high: A sustained short-circuit current can trigger a chain reaction of failures throughout the grid and cause widespread blackouts, severely damaging expensive equipment in the process. The 2003 blackout in northeast North America was set off by a tree's contact with transmission lines in Ohio, which caused a cascade of failures that shut down more than 260 power plants, stopped the flow of 60,000 megawatts throughout the northeast grid, and darkened New York City.

[https://spectrum.ieee.org/inside-the-lab-that-pushes-supergrid-circuit-breakers-to-the-limit]

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)

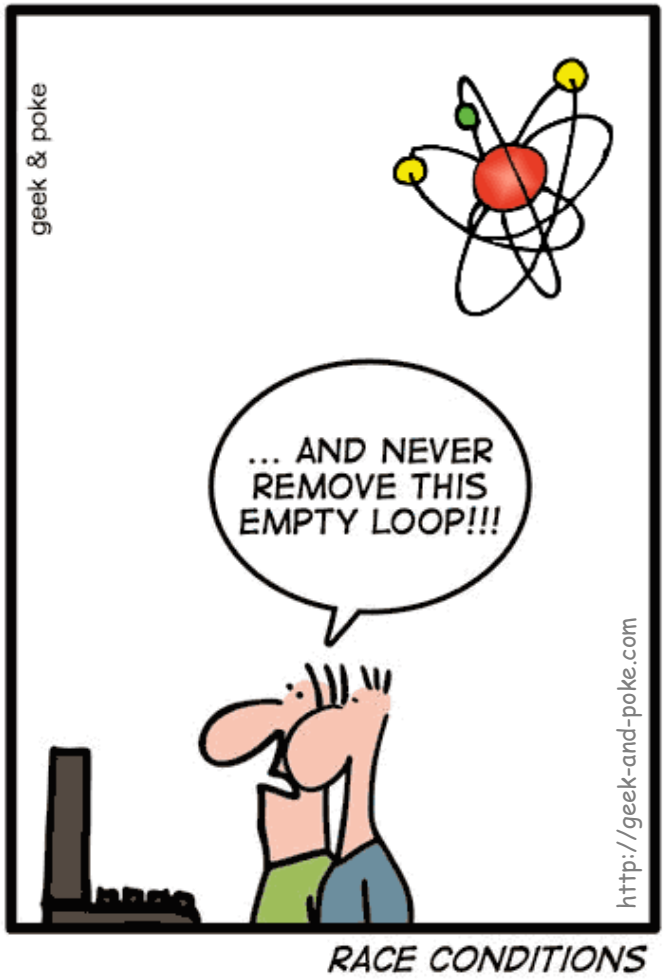
Calls were pouring in from industrial customers, neighboring utilities, and FirstEnergy's own power plant operators, who were all trying to interpret signs of trouble on the grid. "I'm still getting a lot of voltage spikes and swings on the generator," said an operator at the Perry nuclear power plant 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]

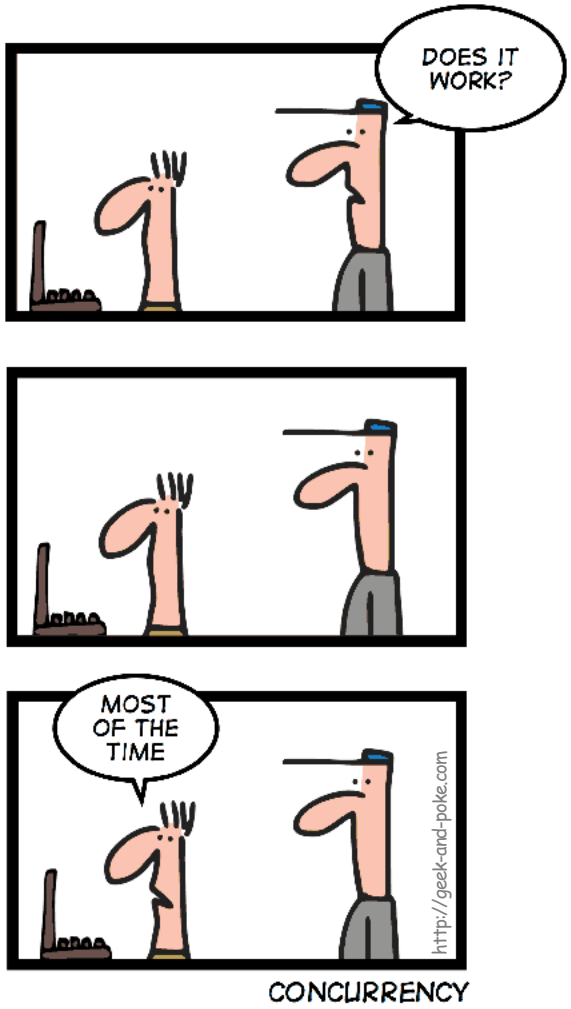
Manhattan ohne Subway

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.



A Race Condition… (4)



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.

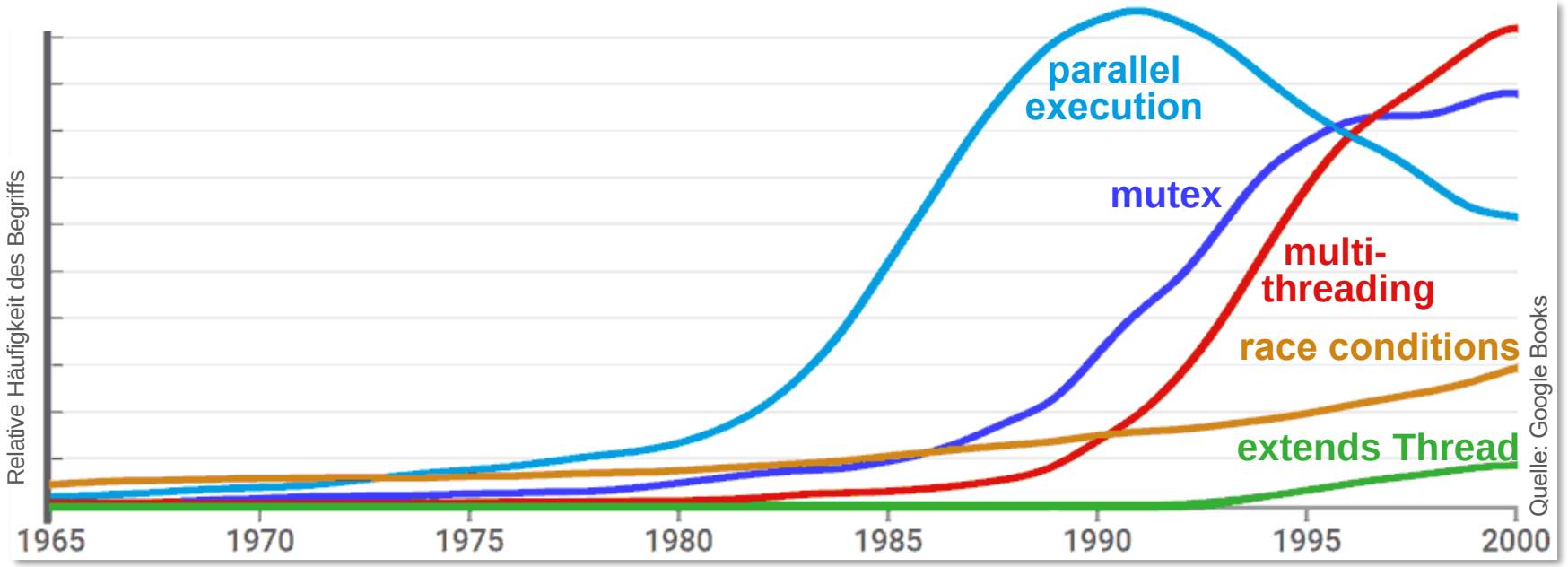
[Tracking the blackout bug, Kevin Poulsen, SecurityFocus, 2004-04-07, www.securityfocus.com/news/8412]

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

Absence of evidence is not evidence of absence.

Evolution des parallelen Programmierens

Illustriert anhand der Häufigkeit der Verwendung typischer Begriffe



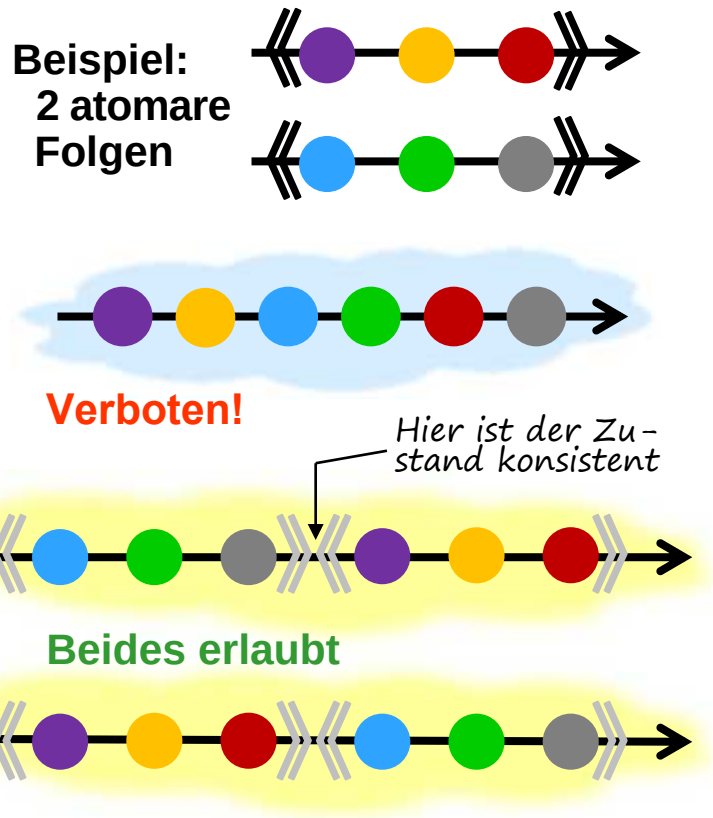
- Parallelausführung von Programmen und Prozessen gewann an Bedeutung, als ab den 1980er-Jahen zunehmend geeignete Hardware verfügbar wurde
- Mit „multithreading" wurden Parallelisierungskonzepte auf die Anwendungsebene gehoben und Applikationsprogrammierern verfügbar gemacht
- Ein entsprechendes Java-Konstrukt stellt „extends Thread" dar
- Zunehmende Nutzung von Parallelität und Multithreading akzentuierten das Problem des wechselseitigen Ausschlusses (mutex) und der race conditions

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?

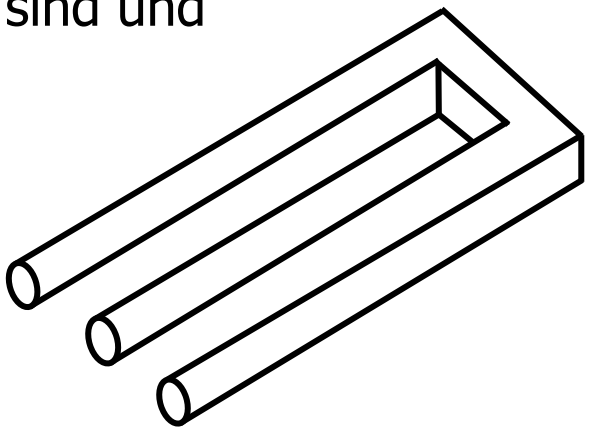
Problematisch sind sogen. „Shared-Memory-Systeme“ – mehrere Prozessoren oder „Kerne“, aber ein gemeinsamer Hauptspeicher

➔ „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 in sich widersprüchlich sind und nicht auftreten dürften



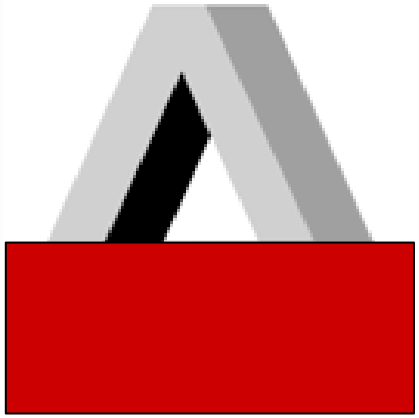
Inkonsistenz: Ein Zustand, in dem mehrere Aspekte, die alle als gültig angesehen werden sollen, nicht miteinander vereinbar sind.

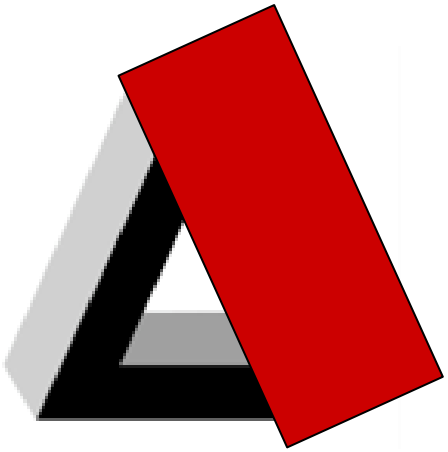
Inkonsistenzen (2)

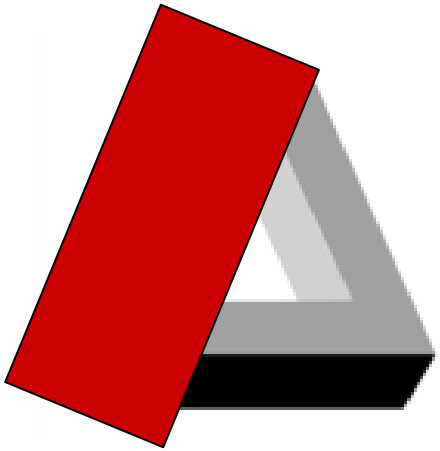
Ein „Tribar“

“Each individual part is acceptable as a representation of an object normally situated in three-dimensional space; and yet, owing to false connexions of the parts, acceptance of the whole figure on this basis leads to the illusory effect of an impossible structure.”

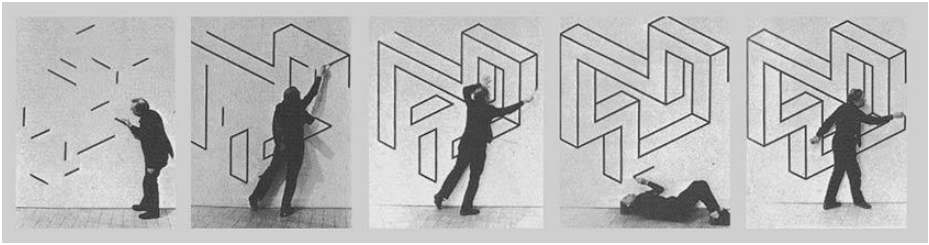
-- Lionel Penrose & Roger Penrose, 1958

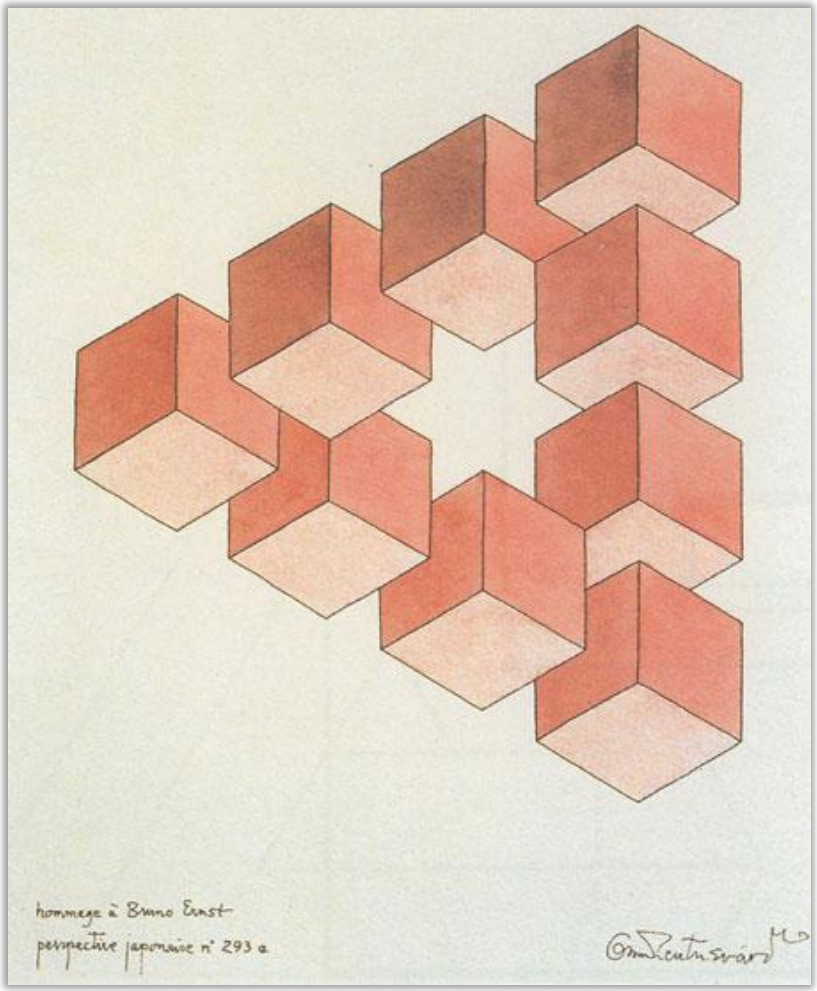






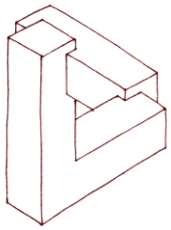
Inkonsistenzen (3)





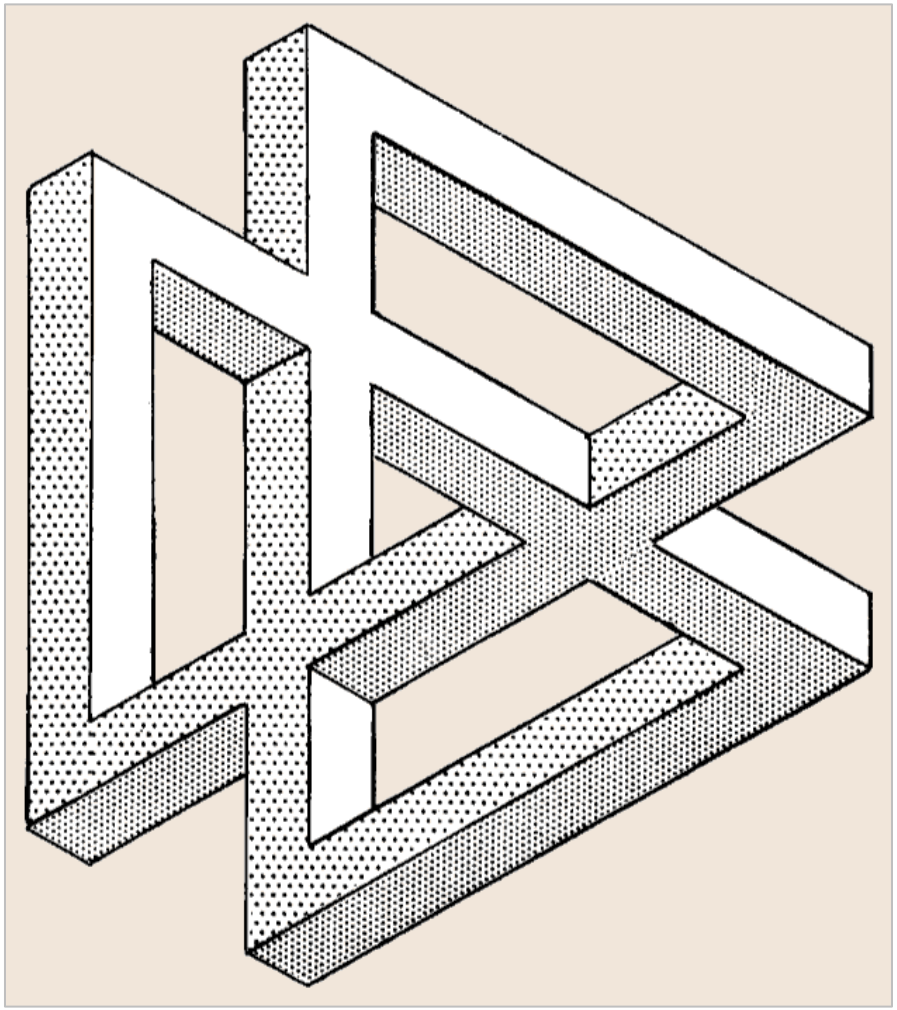
Hommage à Bruno Ernst, perspective japonaise nº 293 a

Als „Erfinder“ paradoxer 3D-Figuren, die als 2D-Skizze auf den ersten Blick realistisch erscheinen, aber unvereinbar mit der Raumgeometrie und somit unrealisierbar sind, gilt der schwedische Künstler Oscar Reutersvärd (1915 - 2002). Die Lateinstunde in der Schule langweilte den Achtzehnjährigen; er krizelte in seinem Grammatikbuch, so entstand 1934 auch eine dreiecksförmige Figur, deren balkenförmige Kanten sich nicht konsistent verbinden liessen – der Tribar war erfunden. Entsprechend dieser Skizze zeichnete er ohne Lineal sein erstes „unmögliches“ Bild, einen Tribar in Form von 9 Würfeln. Das Motiv wurde erst viel später, 1982, zusammen mit zwei weiteren seiner Kreationen von der schwedischen Post für eine Briefmarkenserie ausgewählt.



Reutersvärd wurde Professor für Kunstgeschichte und Kunsttheorie an der Universität Lund und schuf im Laufe seines Lebens mehr als 2500 unmögliche Figuren. Seine Arbeiten wurden jedoch erst spät allgemein bekannt, sodass 20 Jahre später Roger Penrose, angeregt durch einige Bilder von M. C. Escher, unabhängig von ihm den Tribar neu erfinden sollte.

Inkonsistenzen (4)

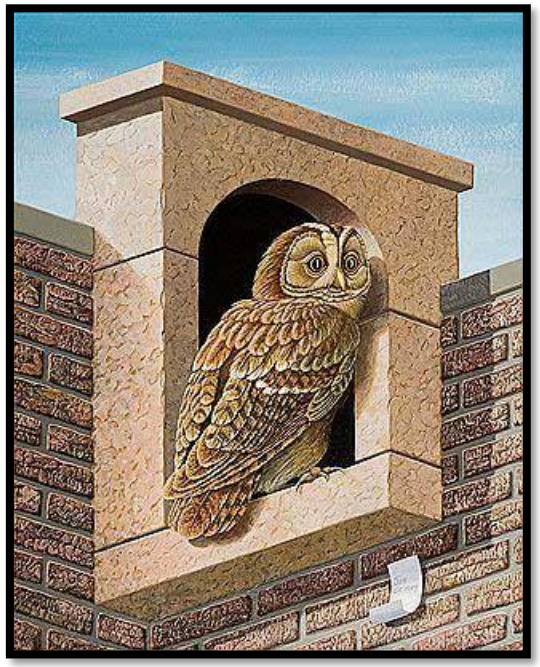


Lionel Penrose und Roger Penrose, 1958:
“Diagram of structure with multiple impossibilities”.

Der Internationale Mathematikerkongress (ICM) wird alle vier Jahre abgehalten; 1954 fand er während acht Tagen mit über 1500 Teilnehmern in Amsterdam statt, wo u.a. auch der Miracle-Computer von Ferranti bei Shell besichtigt werden konnte. Anlässlich des Kongresses widmete das Stedelijk-Museum dem damals noch unbekannten niederländischen Künstler und Grafiker M.C. Escher eine Ausstellung, dessen Werk in den Kongressunterlagen mit „shows many mathematical tendencies and is connected in a remarkable way with the mathematical way of thought“ charakterisiert wird.

Einer der Kongressteilnehmer war der seinerzeitige Doktorand Roger Penrose. Er war in der Ausstellung vor allem von gezeichneten Objekten fasziniert, die in der 3D-Realität eigentlich nicht existieren konnten. Angeregt dadurch, kam er selbst nach einigem Grübeln, ohne von Oscar Reutersvärd zu wissen, auf den Tribar. Später fand er zusammen mit seinem Vater Lionel Penrose (ein Psychiater, Genetiker und Mathematiker) weitere unmögliche Objekte – darunter eine endlos im Kreis an- bzw. absteigende Treppe; ein Motiv, das von Escher dankbar aufgegriffen und umgesetzt wurde.

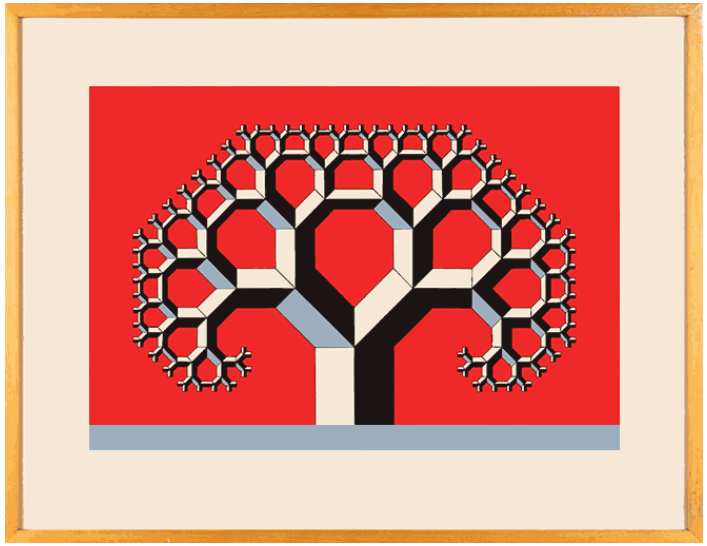
Inkonsistenzen (5)



Neben Oscar Reutersvärd und M. C. Escher bedienten sich auch einige andere Künstler des Stilmittels „unmöglicher“ Perspektiven – z.B. der flämische Künstler Jos de Mey (1928 - 2007), der bewusst surrealistische Elemente von René Magritte aufgriff und vielfach Motive von Escher zitierte (nebenstehend sein Bild „Toegangspoortje naar een Uilenspiegelnest“ aus dem Jahr 1999).

Jos de Mey hat neben den unmöglichen Figuren noch ein anderes mathematisch interessantes Motiv aufgegriffen und vielfach künstlerisch umgesetzt: Den Pythagoras-Baum. Er lernte diese Struktur 1963 auf einem Farbdruck kennen, beschäftigte sich aber erst 1975 ausführlicher damit – bis 1978 füllte er vier Skizzenbücher mit weit über 200 Zeichnungen verschiedener Variationen des Pythagoras-Baums und entwickelte später viele davon zu Aquarellen, Gemälden oder Siebdrucken weiter.

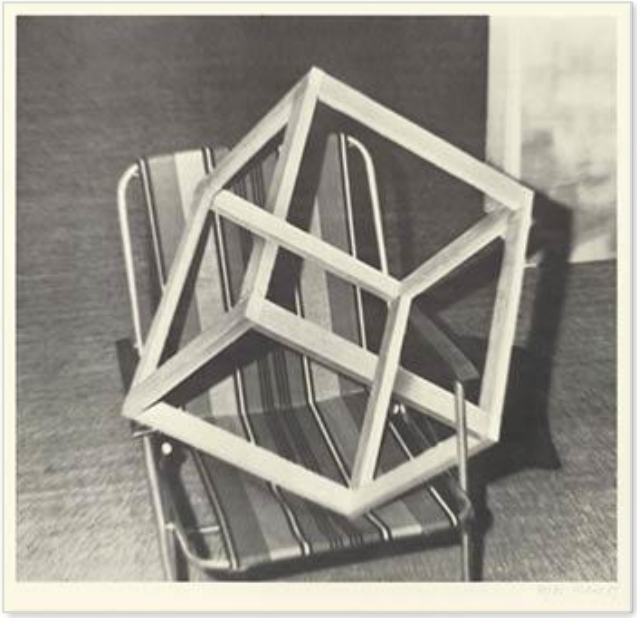
Pythagoras-Bäume hatten wir an → früherer Stelle erläutert.



Links: 'Vliegende bo(o)m geconstrueerd volgens de stelling van Pythagoras', 1975.

Rechts: 'Gefacetteerde versie van een boom volgens stelling van Pythagoras', 1976.

Inkonsistenzen (6)



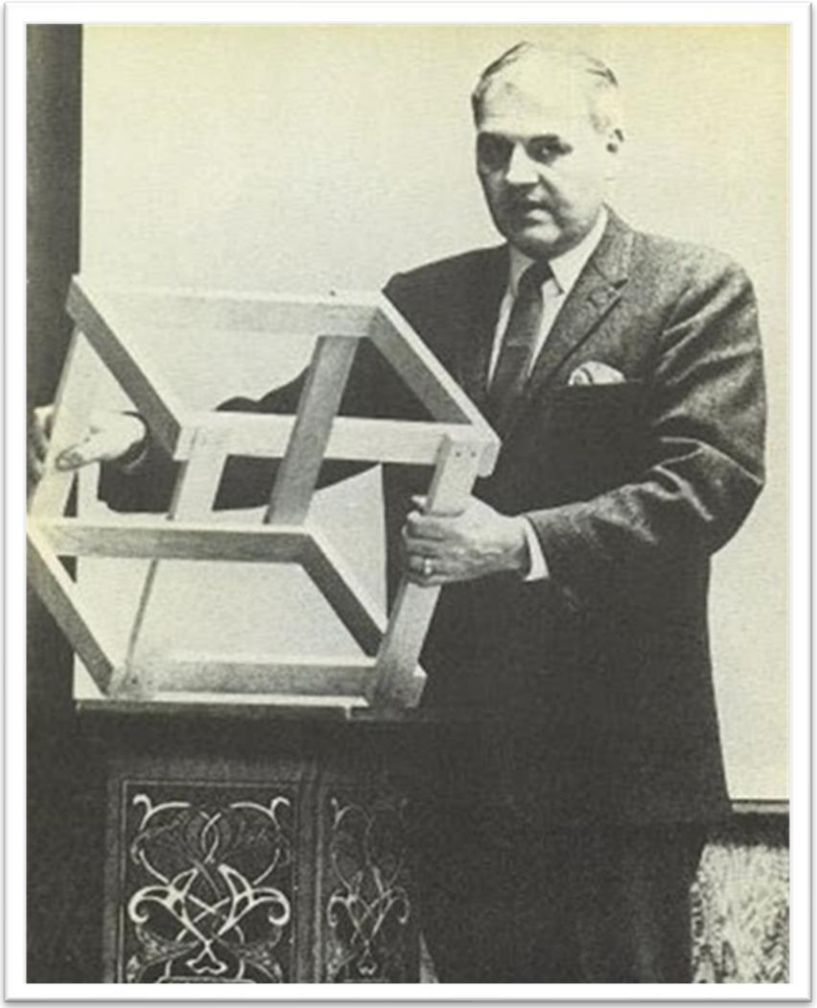
Gerhard Richter, Jahrgang 1934, ist einer der bekanntesten modernen deutschen Künstler. Aufgewachsen in der DDR, Studium ab 1951 an der Dresdner Kunstakademie, 1961 Flucht nach Westdeutschland; einige seiner Wandgemälde in der DDR wurden daraufhin leider übermalt. Ab 1961 Studium an der Kunstakademie Düsseldorf, 1971 bis 1993 dort Professor für Malerei.

Richters Werk ist stilistisch sehr vielseitig, er betätigte sich auch als Bildhauer und Fotograf. Die beiden Bilder sind Teil einer Serie von neun Blatt Offsetdrucken aus dem Jahr 1969, jeweils im Format 45 cm x 45 cm. Die Holzobjekte wurden von Gerhard Richter selbst gebaut und von ihm in einem vertraut-banal wirkenden Arrangement fotografiert, sodass sie wie zufällig an diesem Ort abgestellt erscheinen. Die Schwarz-Weiss-Aufnahmen liess er dann nach seinen Angaben professionell so retuschieren, dass danach die einzelnen Elemente räumlich unmögliche Verbindungen eingehen. Nach diesen Retuschen (natürlich in klassischer, nicht-digitaler Technik) fotografierte Richter die Fotos erneut und druckte das Resultat dann als Offset-Kleinauflage.

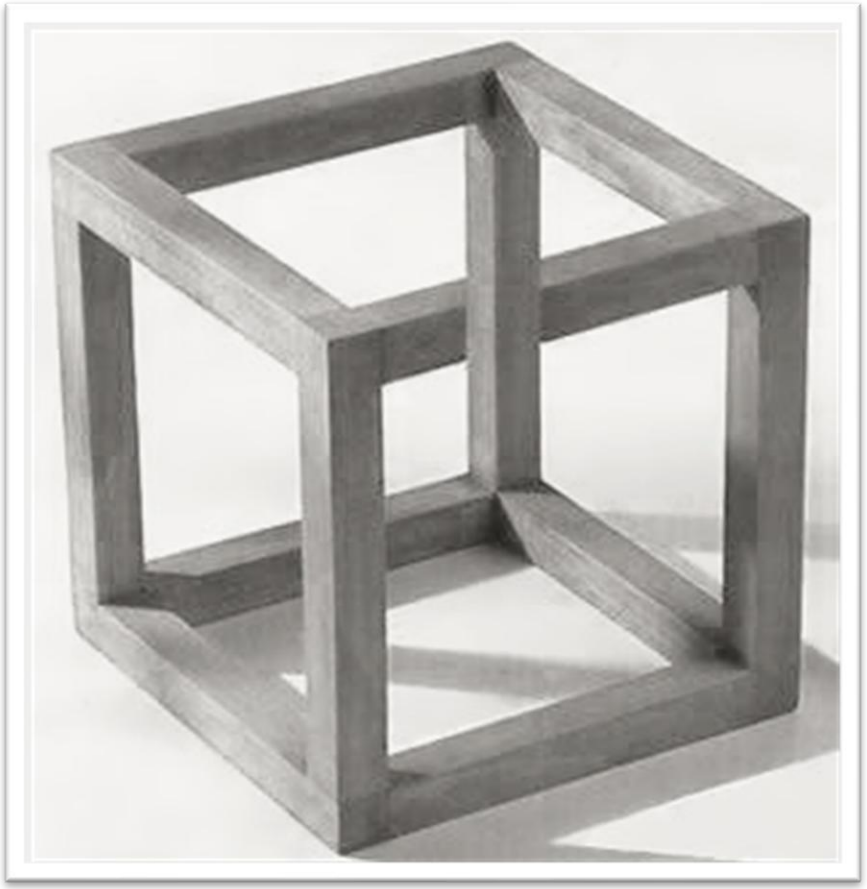
Die Bildserie ist Ausdruck von Richters Suche nach einem „dritten Weg“ zwischen dem Gegenständlichen und dem Abstrakten bzw. Unkörperlichen in der Kunst, er setzt sich dabei auch mit dem Phänomen der optischen Täuschung im Sinne einer Wahrnehmungskritik auseinander. „Dieser harmlose, geradezu kleinbürgerliche Dokumentarismus, gepaart mit der fast fotografischen Identität der Offsetdrucke, lässt uns Betrachter immer wieder erstaunt an den unbegreiflichen, weil scheinbar authentischen, aber eigentlich doch unmöglichen Gebilden zweifeln.“ [Ausstellungskatalog „Gerhard Richter. Editionen 1965 – 1993“, Kunsthalle Bremen, 1993]

Inkonsistenzen (7)

An impossible figure is a drawing making an *impression* of some three-dimensional object, although the object suggested by this three-dimensional *interpretation* of the drawing cannot exist. -- Zenon Kulpa



C. F. Cochran: Letter to Scientific American, Vol. 214, No. 6, p8, June 1966

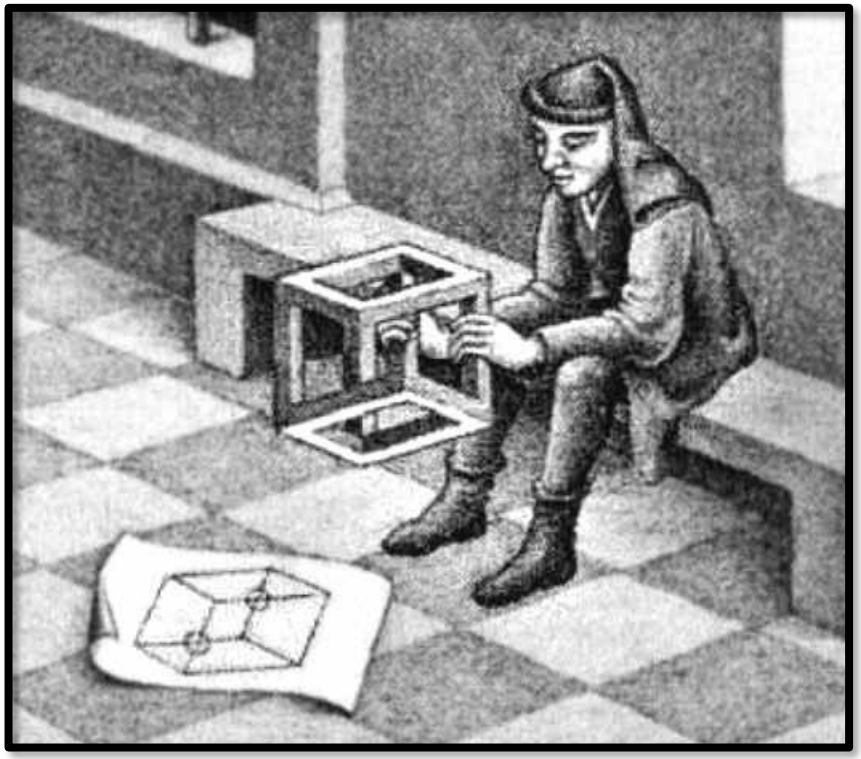


"Are impossible figures possible?" (Zenon Kulpa):

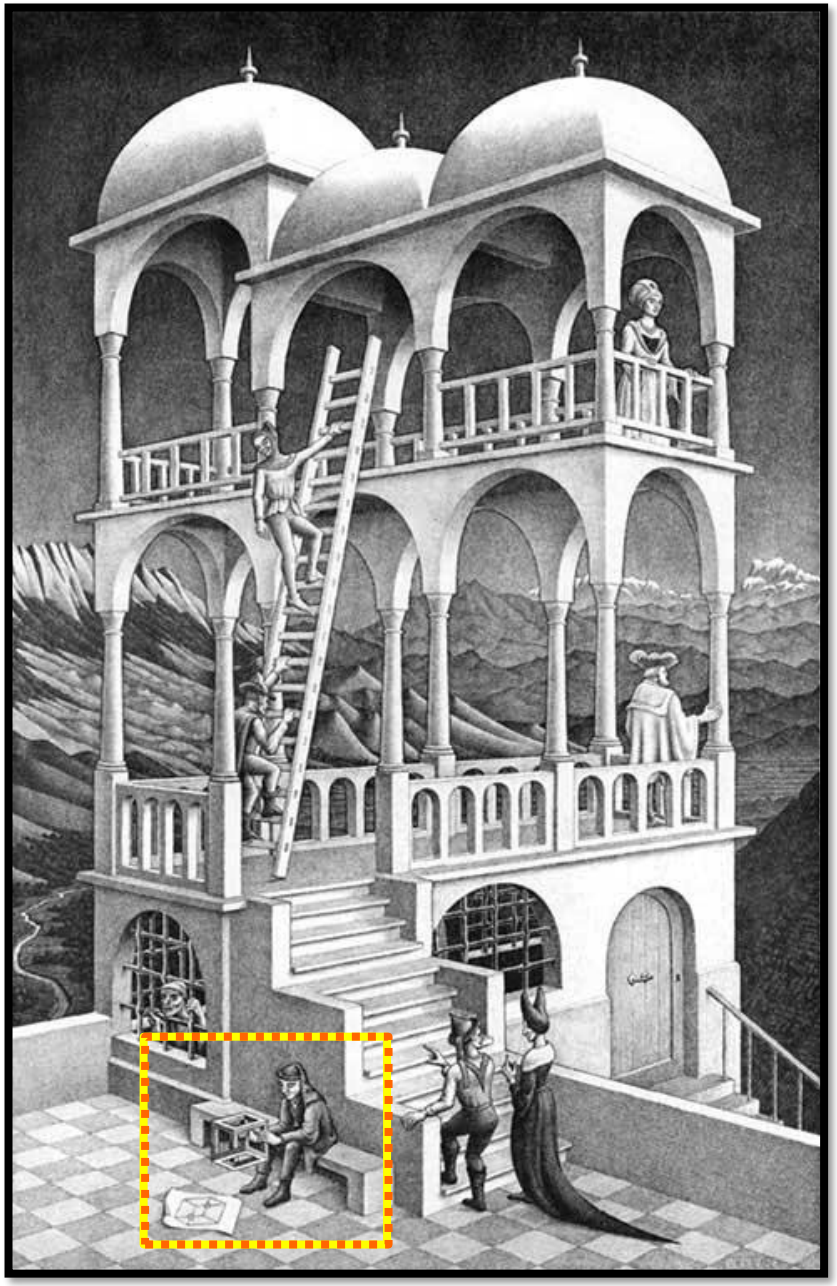
"The so-called 'impossible figures' or 'nonexisting objects' have drawn some attention of both artificial intelligence and psychology of vision researchers. For the psychology they are interesting as new types of illusion, being a source of additional information about our spatial interpretation mechanisms. For similar reasons they are of interest to artificial intelligence research, providing cues for organizing algorithms modelling human abilities to see the three-dimensional world in flat pictures."

Inkonsistenzen (8)

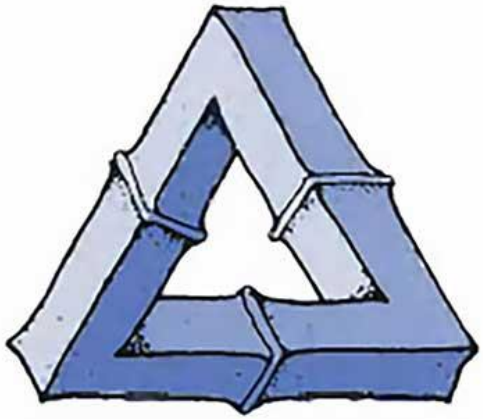
Der unmögliche Würfel ist eine Erfindung Eschers. Neben dem Würfel zeigt das Bild „Belvedere“ ein zweites unmögliches Motiv: Das Haus stellt einen „Quadbar“ dar, der auch auf der weiter vorne gezeigten schwedischen 50-Öre-Briefmarke nach Reutersvärd abgebildet ist. Escher schreibt: „Der auf der Bank sitzende Knabe hat eine würfelförmige Absurdität in den Händen. Er betrachtet sinnend den irrationalen Gegenstand und ist sich offenbar nicht bewusst, dass der Belvedere hinter seinem Rücken auf dieselbe unmögliche Weise gebaut worden ist.“



Maurits Escher (1898 – 1972): Belvedere, 1958



Inkonsistenzen (9)



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

Hier wurde ein reales Objekt fotografiert – dieses sieht allerdings unter anderen Kameraperspektiven ganz anders aus, als es mit diesem Bild suggeriert wird – dazu schaue man auf die nachfolgende slide und betrachte vor allem das Spiegelbild in der Abbildung rechts.

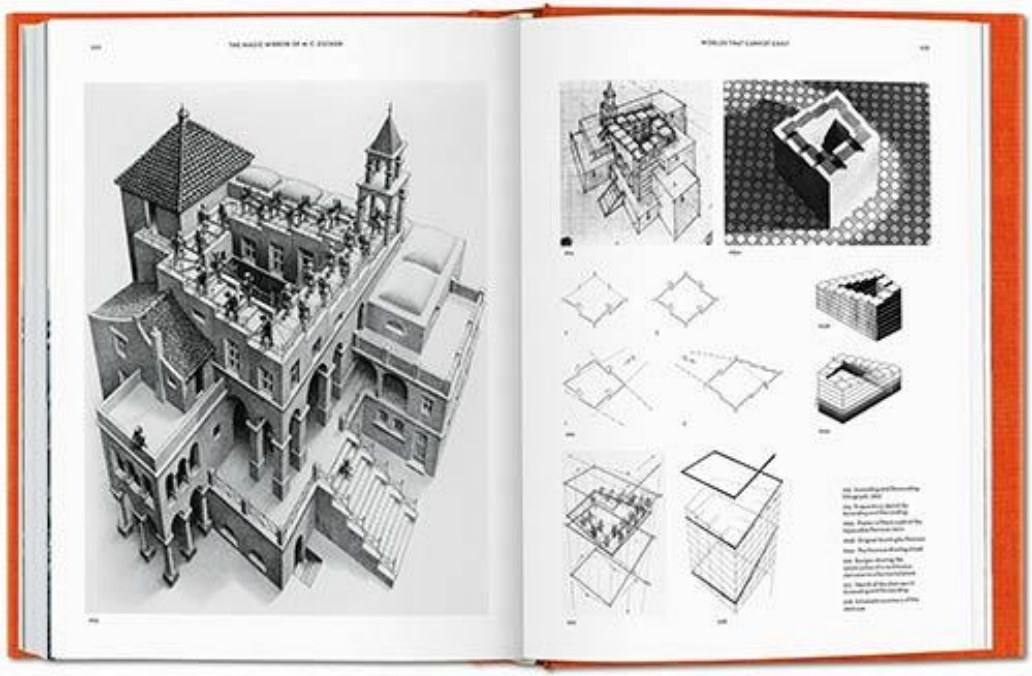
Ist der Schnappschuss nun also ein Abbild eines ***unmöglichen*** *Konstrukts oder nicht doch eher einer* ***möglichen*** *(da ja realen) Figur?*

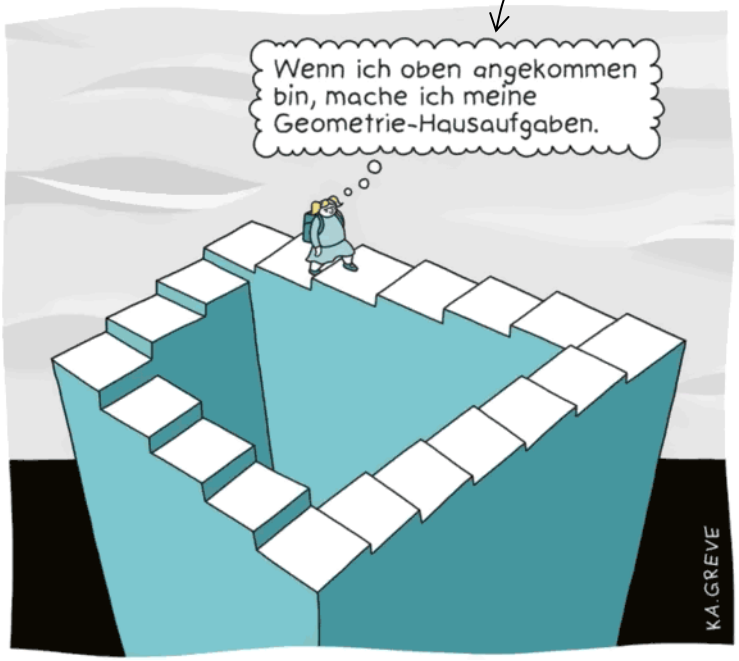
Links oben: Ein aus drei gleichen möglichen (!) Teilen zusammengeschweisster Tribar. Links unten: Dieser Tribar besteht aus 12 Einheitswürfeln mit je vier freien Flächen. Die Gesamtfläche beträgt daher 48 Einheiten, das Volumen 12 Einheiten – oder ist es gar nicht sinnvoll, solche Berechnungen für unmögliche Objekte anzustellen? Rechts: Tribar-Modell auf einem Spielplatz in Sankt Margareten im Rosental, Kärnten.

Inkonsistenzen (10)

Hier verrät uns der populäre niederländische Wissenschaftsautor Bruno Ernst (alias Hans de Rijk, 1926 – 2021, gestorben an Corona), Physiker, Mathematiklehrer und vor allem bekannter Herausgeber von kommentierten Bildbänden zu Eschers Werk, dass man Fotos nicht retuschieren oder anderweitig manipulieren muss, um Abbildungen „unmöglicher“ Figuren zu erhalten. Es genügt, geeignet konstruierte Objekte unter einem bestimmten Winkel zu fotografieren – der Spiegel im Bild rechts oben offenbart den „Betrug“ mit einem Tribar aus Einzelteilen eines Zigarrenkistchens. Da stets höchstens zwei der vier Seiten eines „Balkens“ sichtbar sind, genügen statt massiver Balken auch dünne Lamellen für das Täuschungsobjekt.

Inkonsistenzen (11)

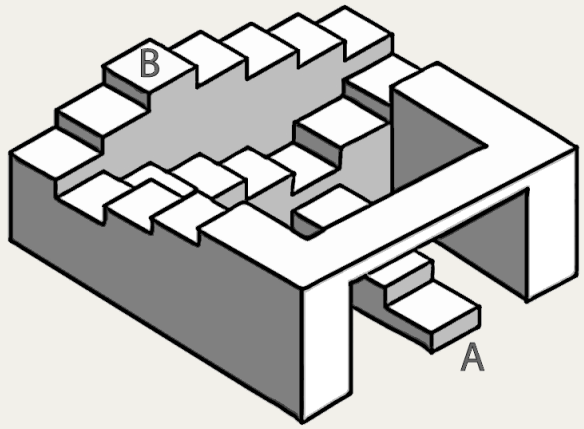




Eschers bekanntes Bild „Treppauf, Treppab" von 1960 geht auf eine Anregung von Roger Penrose und dessen Vater, den Psychiater, Genetiker und Mathematiker Lionel Penrose (1898 - 1972), zurück. Diese veröffentlichten 1958 einen kurzen illustrierten Aufsatz „Impossible objects: a special type of visual illusion" im British Journal of Psychology ("each part of the structure is acceptable as representing a flight of steps, but the connexions are such that the picture, as a whole, is inconsistent; the steps continually descend in a clockwise direction") und schickten eine Kopie an Escher. Auch der Tribar (vgl. die beiden vorherigen slides) findet sich im paper; die Autoren kannten offenbar die früheren Arbeiten von Oscar Reutersvärd nicht. Escher verwendete den Tribar 1961 in einem weiteren bekannten Bild mit einem Wasserfall, der sich selbst speist.

In ihrem Cartoon „Penrose-Treppe" (bzw. „Esornep-Treppe") greift die Berliner Künstlerin, Cartoonistin und Diplom-Architektin Katharina Greve (1991 - 1999 Studium der Architektur an der TU Berlin) das Motiv der unendlichen Treppe auf witzige Art auf.

Inkonsistenzen (12)

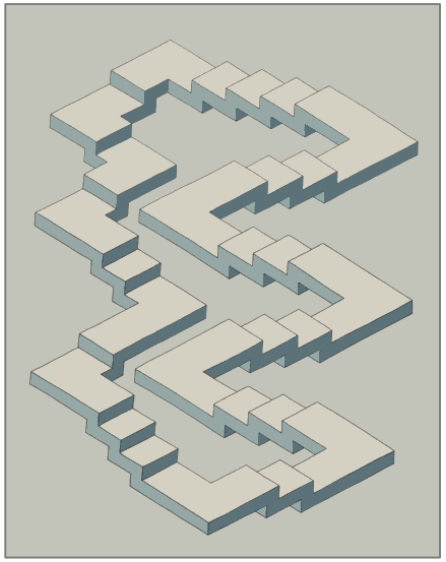


Staircase for lazy people

The problem is to go from A to B, mounting the fewest number of steps. If we turn to the right after four steps and then to the left, we climb eleven steps before reaching the top. Similarly, by turning to the left after four steps and to the right after four more, eleven steps are climbed. But B can be reached by mounting only ten steps and also descending three on the way. This is done by turning to the right after four steps and then to the right again and then crossing the bridge. The further problem is to find out why this is possible or, rather, why the staircase is impossible.

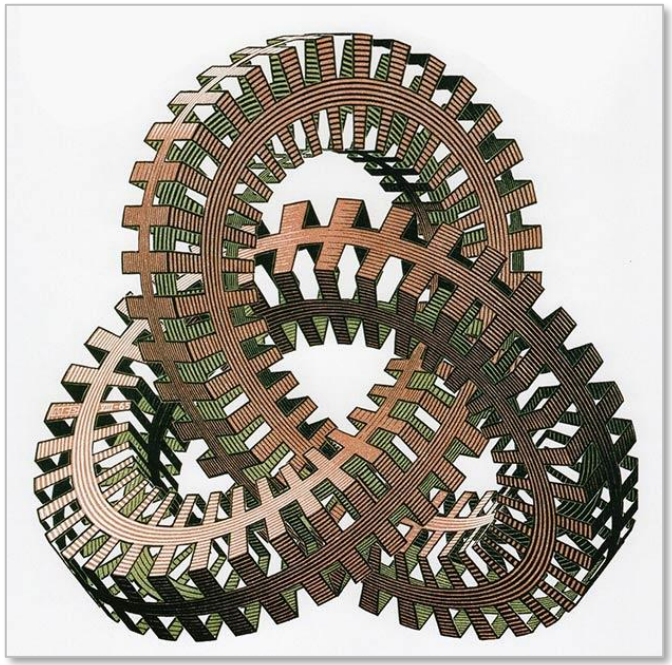
Roger und Lionel Penrose veröffentlichten im Dezember 1958 im "New Scientist" obiges Puzzle in der Rubrik "Puzzles for Christmas". Es stellt eine nette Variante des populären Bildes von Escher („klimmen en dalen", vgl. vorherige slide) mit der „Penrose staircase" dar.

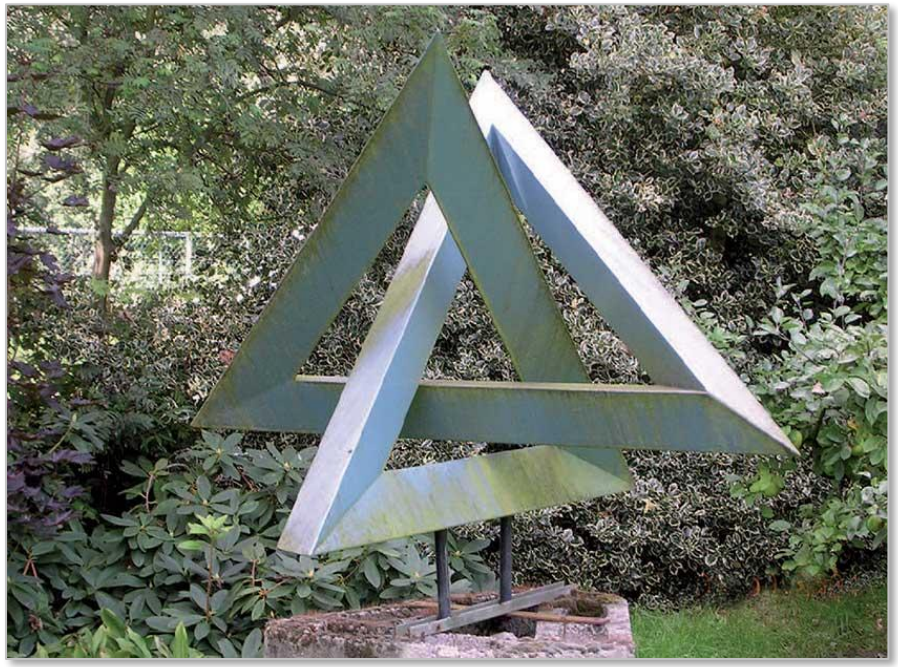
Auch Oscar Reutersvärd hat mehrfach „endlos" aufsteigende bzw. absteigende Treppen gemalt; ein Beispiel findet sich hier rechts. Das Motiv soll er bereits 1937 entdeckt haben.



Inkonsistenzen (13)

Natürlich ist nicht alles, was gezeichnet „topologisch merkwürdig“ aussieht, in der dreidimensionalen Realität tatsächlich unmöglich. Escher nutzte z.B. effektvoll Möbiusbänder (bekannt ist sein Holzschnitt „rote Waldameisen“ von 1963) oder topologische Knoten – links sein Kleeblattknoten, ein Dreifarbenholzschnitt von 1965.





https://archive.bridgesmathart.org/2018/bridges2018-3.pdf

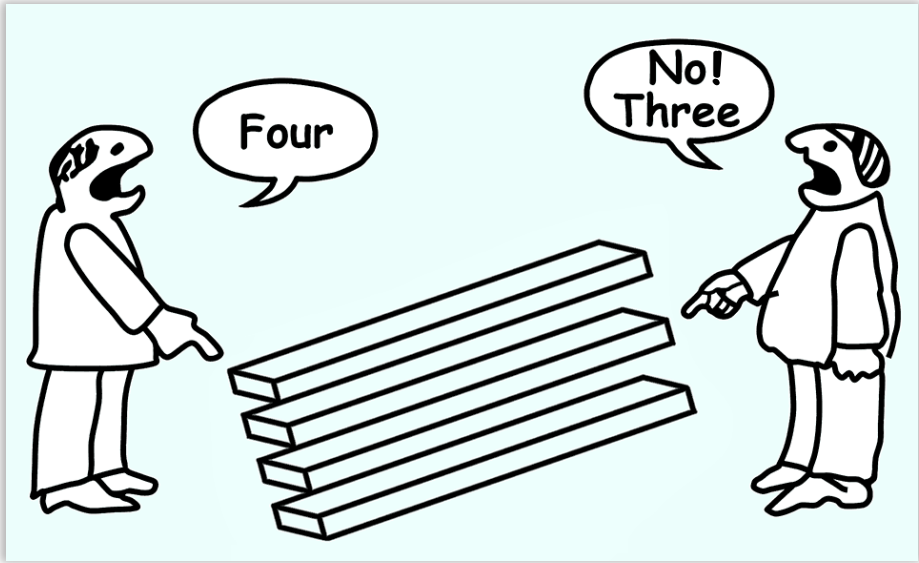
Rechts: Der niederländische Mathematikkünstler Koos Verhoeff (1927 – 2018) schuf einen „eckigen“ Kleeblattknoten mit einer minimalen Anzahl von Balken. Verhoeff studierte Mathematik, Physik, Astronomie und Philosophie in Leiden und Amsterdam. Er war zunächst am Mathematischen Zentrum in Amsterdam, dem heutigen CWI, beschäftigt. Dort lernte er auch seine spätere Frau Bertha Haanappel kennen, die als Rechnerin arbeitete. 1971 wurde er (als Nachfolger von Max Euwe) Professor für Informatik an der Universität Rotterdam. Er emeritierte 1988 und befasste sich dann intensiv mit Entwurf und Konstruktion mathematischer Kunst.

Weitere mathematische Kunst von Verhoeff und Anderen: www.welt-der-form.net/Mathematische_Skulptur/

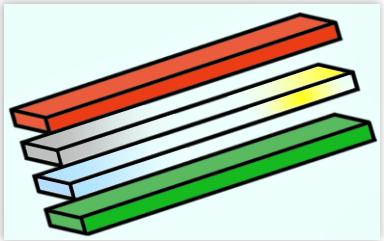
Inkonsistenzen (14)

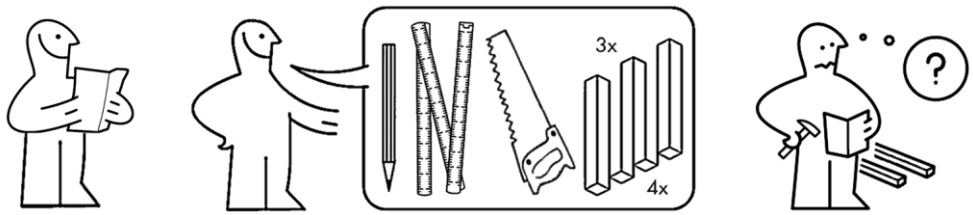
Kann es sein, dass man sich nicht einig wird, ob hier drei oder vier Bretter vorhanden sind? Stimmt eventuell doch die Aussage, die oft dem römischen Kaiser Mark Aurel zugeschrieben wird, dass alles, was wir sehen, nicht die Wahrheit ist, sondern nur eine Perspektive?

Jedenfalls führte in diversen Blogs die nebenstehende Skizze zusammen mit der Frage „drei oder vier?“ zu unterschiedlichsten Antworten:

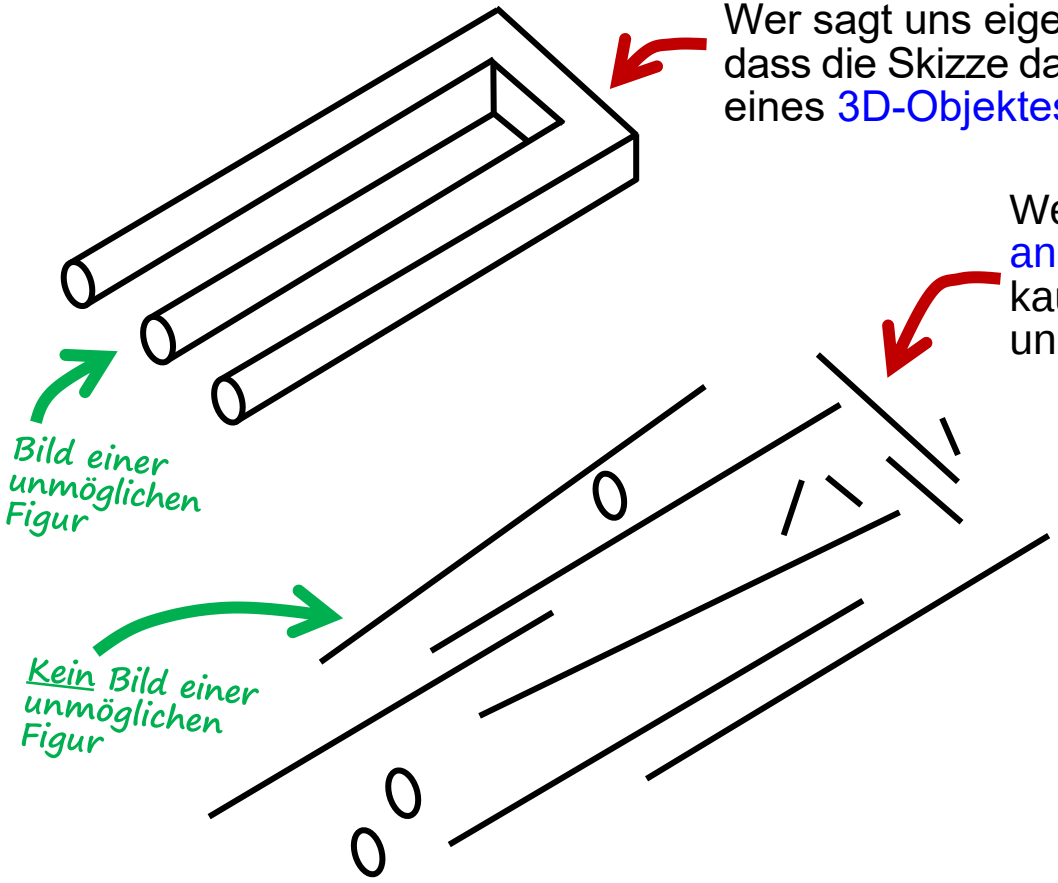


- There are four planks: you can see four rectangles turning into planks.
- Hurts my eyes. When I look at the bottom, I'd see four but looking at the drawing again I see three. My brain is malfunctioning.
- In optical reality, we are looking at 31 separate lines, some just touching.
- We should ask the person who has put them there.
- The artist needs to be brought to charge for inciting deliberate confusion and anxiety needlessly in the name of art.
- Both, its how you see it.
- Actually seven.





Inkonsistenzen (15)



Nochmal grundsätzlich: Wer sagt uns eigentlich, dass die Skizze das Bild eines 3D-Objektes sei?

Wenn die einzelnen Bestandteile etwas anders angeordnet sind, behauptet wohl kaum noch jemand, ein (mögliches oder unmögliches) 3D-Objekt zu erkennen…

Im ersten Bild („Teufelsgabel“) ist es unser Gehirn, das uns sagt, auf der Netzhaut sei der Eindruck eines 3D-Objekts – aufgrund dieser Fähigkeit erkennen wir ja schliesslich die reale Welt mit all ihren relevanten Objekten!

Aber beim Versuch, das Bild zu interpretieren, scheitert es dann.

Daher nochmal das Zitat von oben:

An impossible figure is a drawing making an *impression* of some three-dimensional object, although the object suggested by this three-dimensional *interpretation* of the drawing cannot exist. -- Zenon Kulpa

Inkonsistenzen (16)

Es lohnt sich daher zu erklären, welche Illusionen im Sinne selbst und welche im Urteil des Verstandes entstehen. -- Ptolemäus, Optik

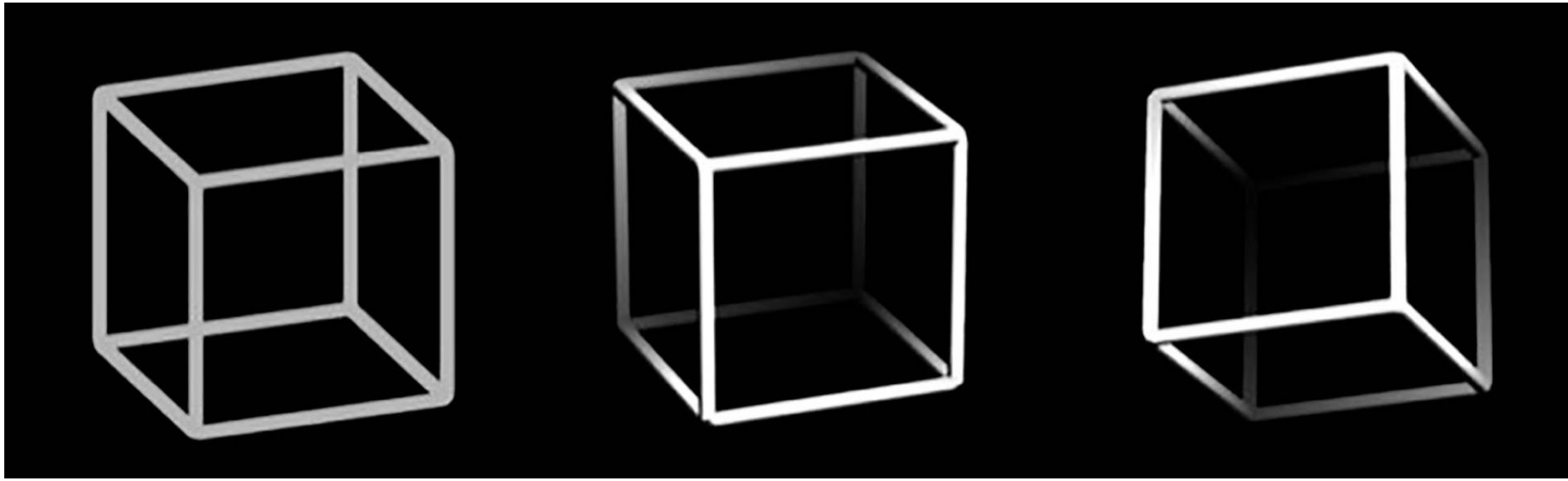


Bild: Universitätsklinikum Freiburg

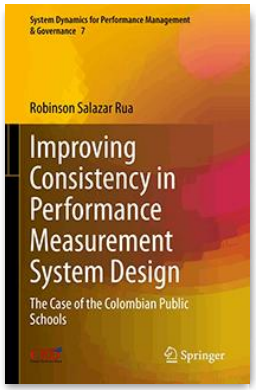
Das Drahtgebilde links lässt zwei alternative Interpretationen als Bild eines 3D-Objekts zu. Es handelt sich um den Necker-Würfel, benannt nach dem vom Schweizer Geologen Louis Albert Necker 1832 beschriebenen Effekt: „An observation … which has often occurred to me while examining figures and engraved plates of crystalline forms: I mean a sudden and involuntary change in apparent position of a crystal or solid represented in an engraved figure“.

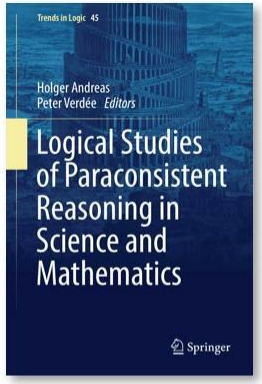
Sieht man den Würfel lange genug an, dann meint man zuweilen, die mittlere Figur zu erkennen (ein Würfel, auf den man schräg von oben draufsieht), andermals aber die rechte Figur (Würfel von schräg unten betrachtet). Beide Interpretationen sind für sich jeweils konsistent. Unser Gehirn sucht sich eine davon aus, wechselt aber nach einiger Zeit schlagartig zur anderen (für den Grund dazu gibt es bisher nur spekulative Erklärungen). Der Effekt tritt übrigens auch bei beim Betrachten eines 3D-Drahtmodells auf. Interessanterweise wird ein „unmöglicher“ Würfel (vgl. die Abbildungen von Holzgestellen mit Schatten auf früheren slides) von unserem Gehirn primär nicht als Interpretation vorgeschlagen.

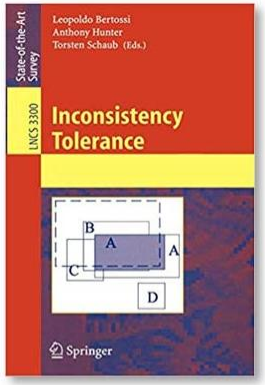
Bereits Ptolemäus beschrieb in seinem Werk „Optik“ den changierenden Eindruck konkaver und konvexer Oberflächen: Die konkave Wölbung des Segels eines entfernten Schiffes könnte fälschlicherweise als konvex wahrgenommen werden. Sinnestäuschung waren übrigens auch für Goethe ein Thema, insbesondere in seiner Farbenlehre. Er sprach auch von „Augentäuschung“, „Gesichtsbetrug“ und „Augengespenst“. Für die kognitive Psychologie sind solche Phänomene interessant, sie liefern Hinweise zur Funktion des Sehens und der Bilderkennung.

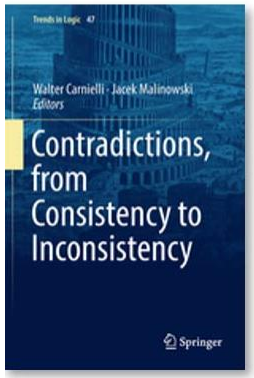
Inkonsistenzen (17)

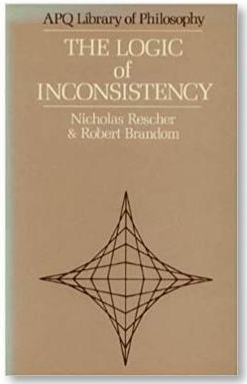
Die Aktualisierung der mehrteiligen mechanischen Anzeigetafel erfolgte erkennbar auf nicht-atomare Weise. Sah man im „falschen" Moment hin, gewahrte man einen unbeabsichtigten Zwischenzustand.











(In)konsistenz scheint wirklich ein relevanter Aspekt zu sein; jedenfalls kann man ganze Bücher darüber schreiben!

Ausgelacht – mittlerweile sind die ratternden mechanischen „Fallblätter" fast überall durch LED-Displays ersetzt (mit Inkonsistenzen höchstens noch im µs-Bereich!)

Lachen, Kanton Schwyz, liegt am oberen Teil des Zürichsees; die S25 verbindet Lachen mit dem HB Zürich. Der Ortsname geht auf das althochdeutsche lahha *(Lache, Sumpf) zurück.*

Inkonsistenzen: Beispiel
(Animation einer Teilchenbewegung)

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

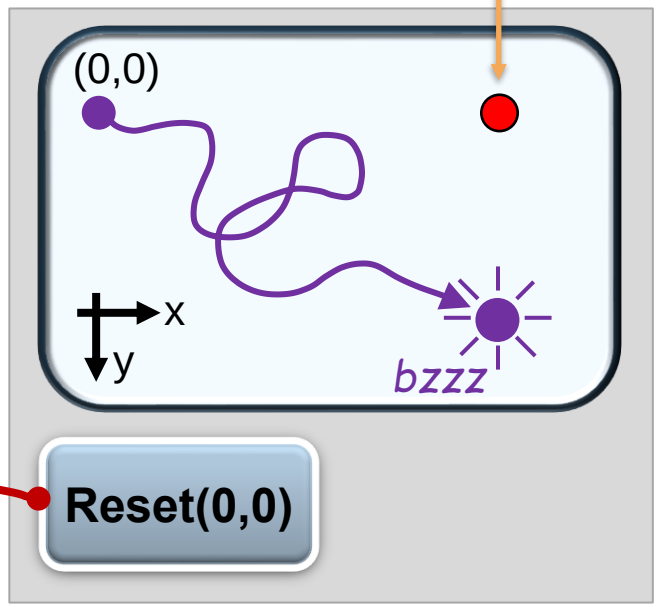
Animationsthread:

```
x = x + Δx;
y = y + Δy;
drawObject
 ("bzzz", x, y);
```
…Δx,Δy…

Eventthread:

hitResetButton

```
x = 0;
y = 0;
```



Compiler

```
read(x)
...
write(x,...)
read(y)
...
write(y,...)
drawObject
```

write(x,0)

write(y,0)

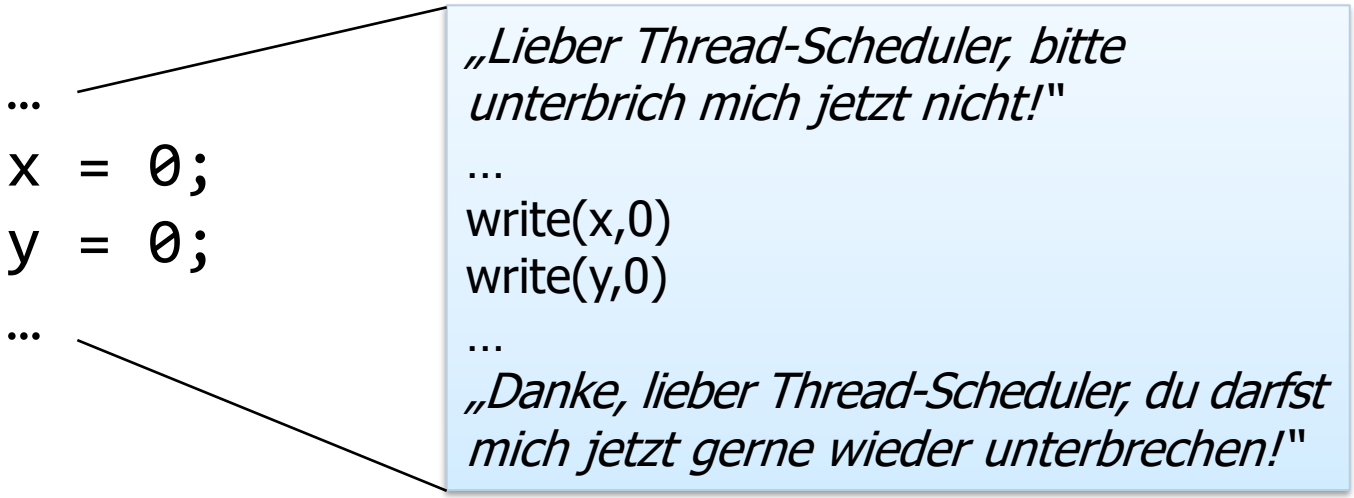
Das „window of opportunity" für den lost update ist sehr klein (Eventthread zwischen den beiden „write" unterbrechen und eine einzige Aktion aus dem Animationsthread ausführen), aber es existiert!

Hier stellen „read" und „write" Maschinenintruktionen 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 funktionierende 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!

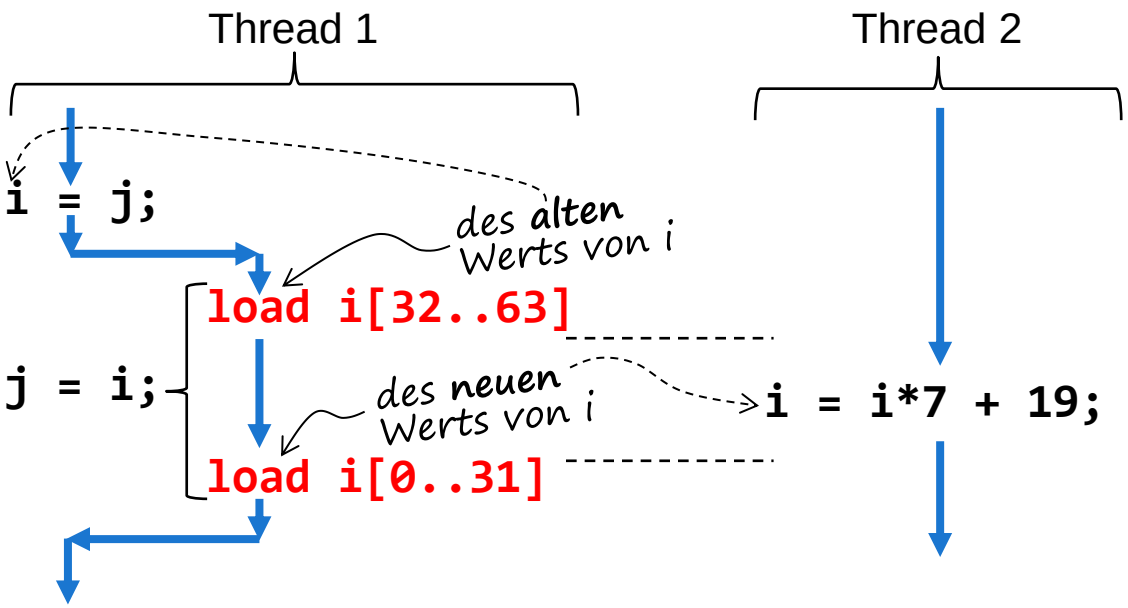
Java: Nicht-Atomarität von double und long

- 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? Ist aber leider wahr!

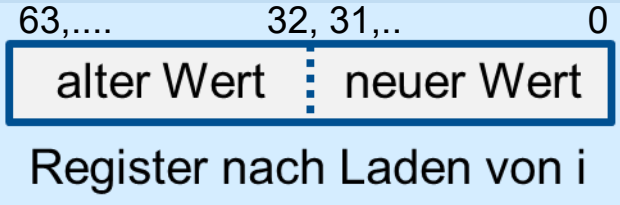
- Bei Threads, die auf die gleiche 64-Bit-Variable zugreifen, kann es daher zu Inkonsistenzen kommen: Ergebnis entspricht dann meist nicht einem möglichen Interleaving auf Sprachebene
  - Ein in der Praxis jahrelang funktionierendes Programm könnte also nach „harmloser" Typänderung *int* → *long* einen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

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



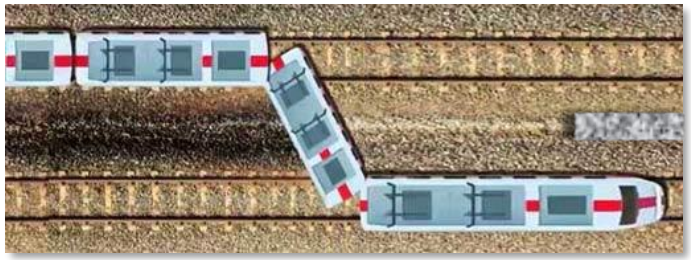
> *"Programmers are cautioned always to explicitly synchronize access to shared double or long variables"*
>
> -- The JVM Specification

Danach steht in i ein „unmöglicher" Wert!



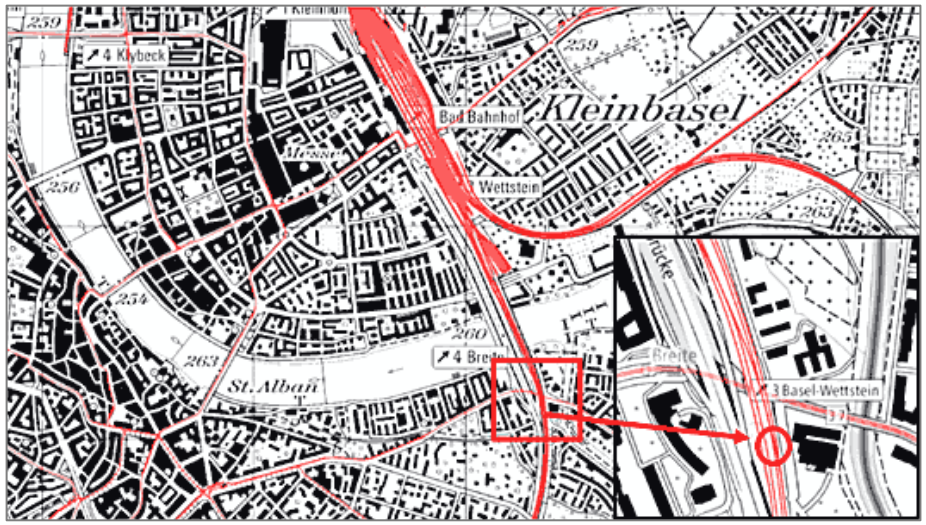
Wie erreicht man Konsistenz, wenn es hardwaremässig keine atomaren Befehle zum Zugriff auf 64 Bits gibt?

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.

Am 3. Sep. 2019 lag der **„Schlussbericht der Schweizerischen Sicherheitsuntersuchungsstelle SUST über Entgleisung eines ICE-Zuges vom 17. Februar 2019 in Basel Badischer Bahnhof“** vor. Als Ursache für die Entgleisung wird festgestellt: „Die Entgleisung […] ist auf das versehentliche, unzulässige Hilfsauflösen einer Zugfahrstrasse und dem folgenden Einstellen einer anderen Zugfahrstrasse zurückzuführen, wodurch unter dem fahrenden Zug eine Weiche umgesteuert wurde.“



Karte aus dem SUST-Bericht

Die Frankfurter Rundschau berichtete am 23.9.19; daraus einige kurze Passagen:

Es waren Sekunden, die über Leben und Tod entschieden – und das gleich doppelt. Hätte der Lokführer eineinhalb Sekunden später gebremst, wäre der Schnellzug mit 240 Passagieren möglicherweise heftig verunglückt. Wäre der ICE nur einige Sekunden schneller gewesen – es wäre gar nichts passiert.

Das hört sich wie eine race condition an!

An jenem Winterabend des 17. Februars nimmt es der ICE 373, aus Deutschland kommend, mit dem regulären Fahrplan nicht so genau. Am Badischen Bahnhof in Basel setzt er sich um 20.44 Uhr in Bewegung, gut eine Stunde hinter seiner Zeit. Das heißt: Der reguläre Folgezug ist direkt hintendran. Ein Sachverhalt, der entscheidend werden sollte. Denn während der ICE 373 losrollt, widmet sich im Stellwerk der Fahrdienstleiter der DB Netz dem zweiten Schnellzug. Der soll ein paralleles Gleis nehmen. Der erste, später verunglückte Zug würde an Weiche 194, etwa 900 Meter vor dem Singertunnel, auf das rechte Gleis abbiegen, der Folgezug auf das linke.

Beim Eintippen der Route des zweiten Zugs vertut sich der Fahrdienstleiter. Er hinterlegt statt dem linken den rechten Weg. Das Missgeschick dürfte schnell bemerkt, die falsche Route mittels einer sogenannten Hilfsauflösung gelöscht und mit der korrekten – Weiche 194 nach links – ersetzt sein.

Das Tragische: Die für den zweiten Zug falsche Route ist in weiten Teilen identisch mit dem richtigen Weg für den bereits fahrenden ICE 373. Und so ändert der Fahrdienstleiter – ungewollt – auch die Fahrstrecke für den ersten Zug. Die Software des Stellwerks entriegelt Weiche 194 und schaltet jenen Motor an, der die Weichenzunge nach links verschiebt. Es ist laut Bericht ein „stark erhöhter Stellstrom“ nötig; die Weiche will sich nicht gut bewegen. Heute ist klar, warum: Räder des ICE 373 sind im Weg. Er entgleist.

Aber selbst wenn die Routen der aufeinander folgenden ICE durcheinandergeraten – wieso stellt sich eine Weiche um, wenn ein Zug über sie rollt? Eigentlich gibt es dagegen technische Sicherungen. Diese „Gleisfreimeldung“ erkennt, ob ein Zug in einem Streckenabschnitt unterwegs ist. Für Weiche 194 beginnt die zugehörige Kontrollzone 80 Zentimeter vor der Weichenzungenspitze. ICE 373 hat zu jenem Zeitpunkt, als die Weiche das Umschaltsignal bekommt, „gerade noch nicht“ den Kontrollabschnitt erreicht. Deshalb bewegt sich die Weiche. Das dauert, wie das Diagnosesystem registriert, genau 4,8 Sekunden. Exakt in diesem Zeitraum rollt der verunglückte Zug auf die Weiche. Wäre er wenige Sekunden früher gewesen, hätte die Gleisfreimeldung angeschlagen, die Weiche hätte sich nicht bewegt, es wäre nichts passiert.

Die Bahnanlagen werden, obwohl auf Schweizer Boden, von der Deutschen Bahn nach deutschen Regeln betrieben. Das ist an einer Stelle ein Unterschied, der entscheidend wird. Schweizer Vorgaben sind strenger, wenn eingestellte Fahrstrecken geändert werden. „Notauflösezeitverschluss“ heißt die Technik, die in solchen Fällen verhindert, dass direkt danach Fahrstraßenelemente wie Weichen angesteuert werden. Mit dem Mechanismus wäre ICE 373 nicht entgleist. Die Schweizerische Sicherheitsuntersuchungsstelle spricht von einem „Sicherheitsdefizit“.

Als der Zug steht, öffnet der Lokführer ein Fenster und blickt zurück. Er setzt einen Notruf ab. Etwa zur gleichen Zeit erkennt der Fahrtdienstleister im Stellwerk, dass etwas nicht in Ordnung ist. Aber: „Es war dem Fahrdienstleiter Basel Badischer Bahnhof nicht mehr möglich, den Lokführer durch einen Aufruf zum Anhalten aufzufordern.“ Kurz vor der Weiche wechselt der Zugfunk des Lokführers vom deutschen in das schweizerische System. Das Stellwerk aber sendet deutsch. „Die Funkbereiche überlappen sich nicht und entsprechen nicht den zugeteilten Bereichen für die Betriebsführung“, stellt der Bericht fest.

Das letzte Kapitel des Schweizer Berichts zum Unfall vom 17. Februar trägt die Überschrift: „Seit dem Unfall getroffene Maßnahmen.“ Die Ausführung fällt kurz aus: „keine.“

Dass ein Zug auf zwei Gleisen gleichzeitig fährt, kam auch schon früher vor. Am 21. August 2012 kam es beispielsweise bei der Berliner S-Bahn zu einem entsprechenden Unfall; ein Zug der S 25 ist kurz nach dem Verlassen des Bahnhofs Tegel auf der Fahrt Richtung Hennigsdorf entgleist. Im offiziellen Un-

https://blog-imgs-53-origin.fc2.com/n/i/s/nishitetsu/5-format14.jpg

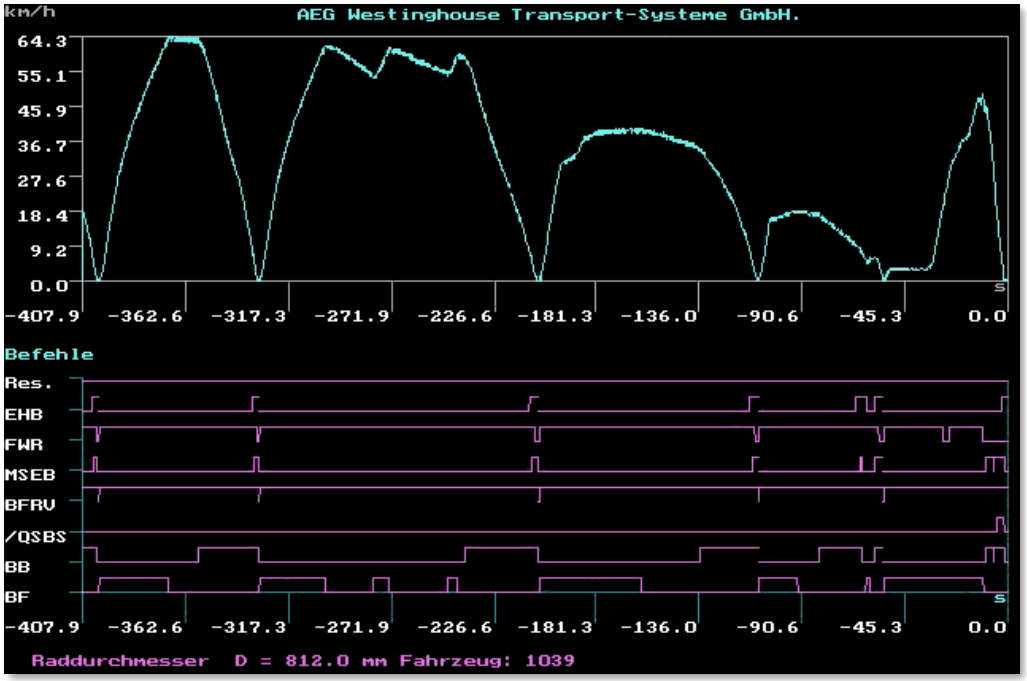
www.imago-images.de/bild/st/0058370772/s.jpg

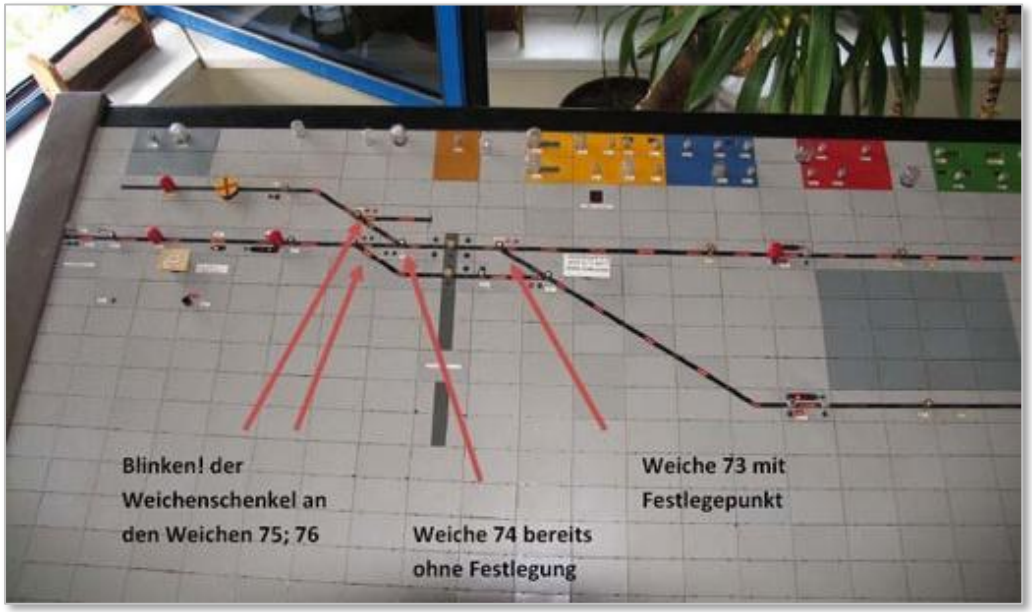
Untersuchungsbericht liest sich das später so:

„Nach dem Befahren der Weiche 74 mit den ersten beiden Wagen, lief die Weiche 74 unter dem fahrenden Zug vollständig in die Linkslage um, wodurch die nachfolgenden Fahrzeuge in Richtung Stumpfgleis der Niederbarnimer Eisenbahn abgeleitet wurden. Dabei entgleisten der 3. und 4. Wagen des Zuges mit allen Radsätzen, und der 3. Wagen kam in Schräglage. Der Triebfahrzeugführer setzte einen Notruf ab.

Der diensthabende Fahrdienstleiter und die in Einweisung tätige Fahrdienstleiterin machten unmittelbar nach dem Ereignis keine Aussagen. Auch nach Aufforderung durch die Bundespolizei erfolgte keine Aussage der beiden Betriebseisenbahner. In ihren späteren Stellungnahmen beim Ständigen Vertreter des Eisenbahnbetriebsleiter (EBL) machten beide zum Ereignisher-gang und ihren unmittelbaren Betriebshandlungen auch weiterhin keine Aussage.

Auf dem führenden Fahrzeug wurden Fahrverlaufsdaten elektronisch auf einem Gerät der AEG Westinghouse Transport-Systeme GmbH aufgezeichnet. […]





Bilder: Untersuchungsbericht 60 - 60uu2012-08/00129, Eisenbahn-Unfalluntersuchungsstelle des Bundes

Der S-Bahnzug wird demnach bis zur Entgleisung in der Weiche 74 auf eine Geschwindigkeit von ca. 38,5 km/h beschleunigt. Die Spitze der Weiche 74 befindet sich im km 11,214, ab hier ist eine weitere Geschwindigkeitserhöhung ausgewiesen auf ca. 48,2 km/h, die danach in ca. 79 m erreicht wird. Das Ende der Aufzeichnung mit ausgewiesenen 0 km/h wird dann nach weiteren ca. 69 m erreicht.

Beim Eintreffen auf dem Stellwerk stellte sich folgende Situation dar: Auf dem Stelltisch war das Gleis 1 rot ausgeleuchtet. […] Die Weiche 74 liegt abzweigend. Die Weichenschenkel der Weichen 75 und 76 blinken.“

Das Blinken zeigte eine Störung an. Tatsächlich hatte am Vortag ein Blitzeinschlag die Achsenzähler der Gleisfreimeldeanlage zerstört, sodass kein Normalbetrieb mit automatischer Signalisierung möglich war; die Fahrwege und Signale mussten per Hand am Stelltisch eingestellt werden. Weil das Stellwerkpersonal die Nordausfahrt nicht einsehen kann, ist vorgeschrieben, dass die Weiche erst dann gestellt werden darf, wenn die S-Bahn beim nächsten Bahnhof (Heiligensee) angekommen ist und der dortige Fahrdienstleiter dies dem Bahnhof Tegel zurückmeldet. Doch so lange warteten die Stellwerker nicht: Die Weiche 74 wurde gestellt, als sich der Zug dort noch befand.

Drei Jahre später, am 30. Juni 2015, meldet die Berliner Boulevardzeitung BZ unter der Schlagzeile „Entgleiste S-Bahn in Hoppegarten: Wieder die Weiche!“:

„Es ist gegen 21.45 Uhr, als der Zug der S 5 aus Strausberg Nord in Richtung Spandau unterwegs ist. In den acht Waggons sitzen 55 Fahrgäste. Der Zug rollt mit etwa 40 km/h in Richtung S-Bahnhof Hoppegarten. Kurz vor der Einfahrt kommt eine Weiche. Die ersten zwei Viertelzüge (je zwei Waggons) passieren ohne Probleme, die hinteren beiden aber werden aus den Schienen gehoben. […] Der Unfall in Hoppegarten war nicht der erste dieser Art. Bereits vor drei Jahren kam es in Tegel zu einem ähnlichen Unglück. Ein Zug der Linie S 25 entgleiste dort wegen eines Weichenfehlers.“

Die Eisenbahn-Unfalluntersuchungsstelle des Bundes schreibt in der Kurzfassung des Untersuchungsberichts: „Die Zugentgleisung war Folge einer unzeitigen Weichenbedienung durch den Fahrdienstleiter des Stellwerks ‚Hob‘. Dieser hatte die spitz befahrene Weiche 49 umgestellt, obwohl die Weiche durch S 5130 befahren wurde.“ Der Bericht enthält eine bemerkenswerte tabellarische Gegenüberstellung des „Sollverhaltens“ und des tatsächlichen Ablaufs:

„Soll / Ist-Vergleich […]

Sollzustand	Istzustand
Der Fahrdienstleiter überzeugt sich von der Vollständigkeit des eingefahrenen Zuges (Zugschlusserkennung durch Hinsehen) und legt den Fahrstraßensignalhebel und eventuell beanspruchte Fahrwegelemente in die Grundstellung zurück.	Der Fahrdienstleiter löst die Fahrstraße x/5 mittels Hilfsauflösung auf, indem er die Taste HT w/x betätigt. Er überzeugt sich nicht von der Vollständigkeit des Zuges. Der Fahrdienstleiter bringt den Fahrstraßensignalhebel in die Grundstellung zurück.
- Ende der Einfahrt -	**- Einfahrt noch nicht beendet -**
Der Fahrdienstleiter überzeugt sich vom Freisein des Abschnittes hinter Signal W und aller zur Fahrstraße w (-Ausfahrt aus Gleis 4-) gehörenden Elemente.	Der Fahrdienstleiter überzeugt sich nicht vom Freisein der Weiche 49.
Der Fahrdienstleiter stellt die zur Fahrstraße w gehörenden Elemente ein (bspw. Weiche 49 in Minuslage) bzw. kontrolliert deren, für die entsprechende Fahrstraße benötigte, Lage (bspw. Weiche 50 in Pluslage) und verschließt die Fahrstraße durch Bedienung des entsprechenden Fahrstraßen-Signalhebels (Drehung um 45º). Durch eine weitere Drehung auf 90º wird die Fahrstraße w festgelegt und das Signal W geht in die Fahrtstellung.	Der Fahrdienstleiter stellt die Weiche 49 in die Linkslage (Minusstellung des Weichenhebels), prüft jedoch zuvor nicht, ob diese frei von Fahrzeugen ist. Der Verschluss bzw. die Festlegung sind nicht mehr möglich, da das Fahrstraßenelement Weiche 49 keine Endlage erreicht.
	Die Weiche 49 wird unter dem fahrenden Zug S 5130 umgestellt, der Zug entgleist.

Zusammenfassend kann festgestellt werden, dass die Fahrstraße x/5 vorzeitig aufgelöst wurde. Die durchgeführten Hilfsbedienungen wurden nicht dokumentiert. Die Weiche 49 wurde ohne Prüfung auf Freisein unter der S-Bahn umgestellt. […] Die Vermeidung der Eingriffe des Bedieners in ein sicher arbeitendes Stellwerkssystem ohne betriebliche Notwendigkeit wäre ereignisverhindernd gewesen.“

Kritischer Abschnitt

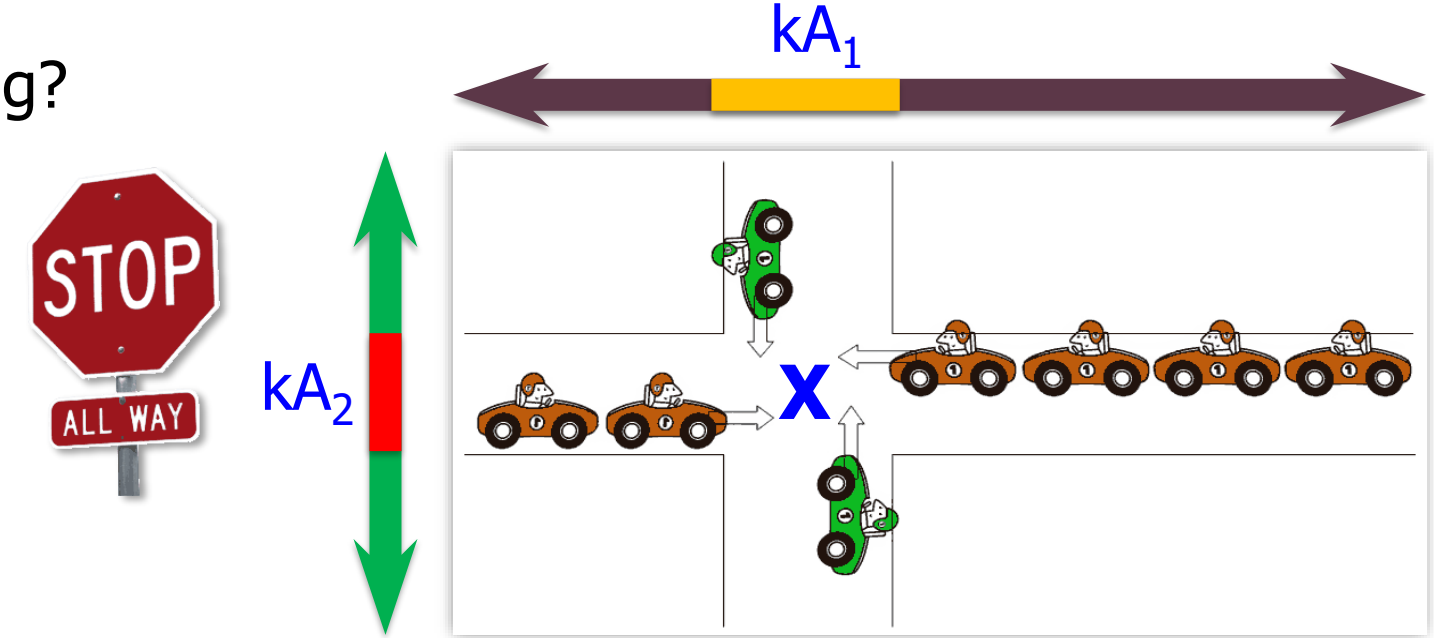
Vereinfacht mag man sich zunächst „ein und denselben“ kritischen Abschnitt vorstellen

- 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 Indexvariablen für diesen Platz
  - 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?
- Stop?
- Kreisverkehr?



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

Kritischer Abschnitt: Anforderungsspezifikation

- Das Aufgabe 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 gegenseitig ausgeschlossen werden sollen.

Man denke z.B. an selbstfahrende Autos, die gleichzeitig eine Kreuzung erreichen; gleiches gilt jedoch 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**

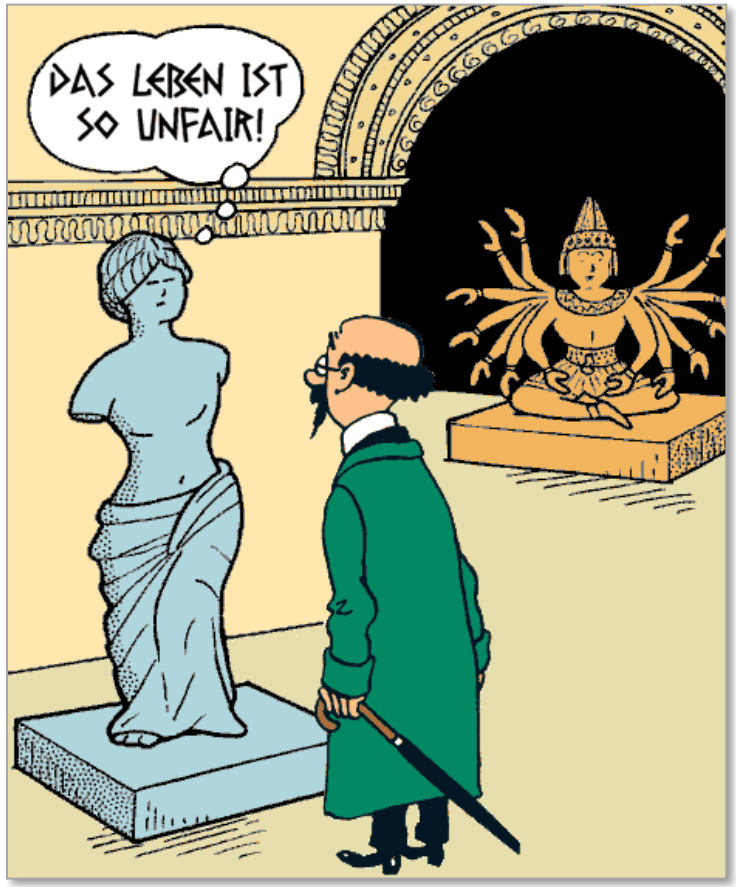
Ein sich bewerbender Prozess darf nicht dauernd (bzw. allzu oft) von anderen Prozessen übergangen werden

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

Fairness?

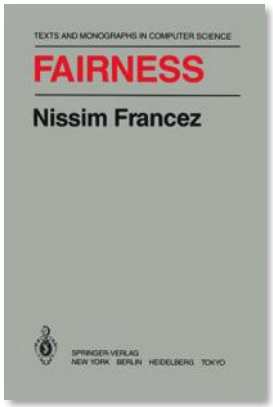
A fairness assumption says, in essence, that if one tries something often enough, one will eventually succeed. There is nothing in our understanding of the physical universe that supports such a belief.
-- Rob van Glabbeek & Peter Höfner

- 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?



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

- Zum formalen Fairness-Begriff wurden jedenfalls schon ganze Bücher geschrieben
  - Wir diskutieren das hier aber nicht weiter

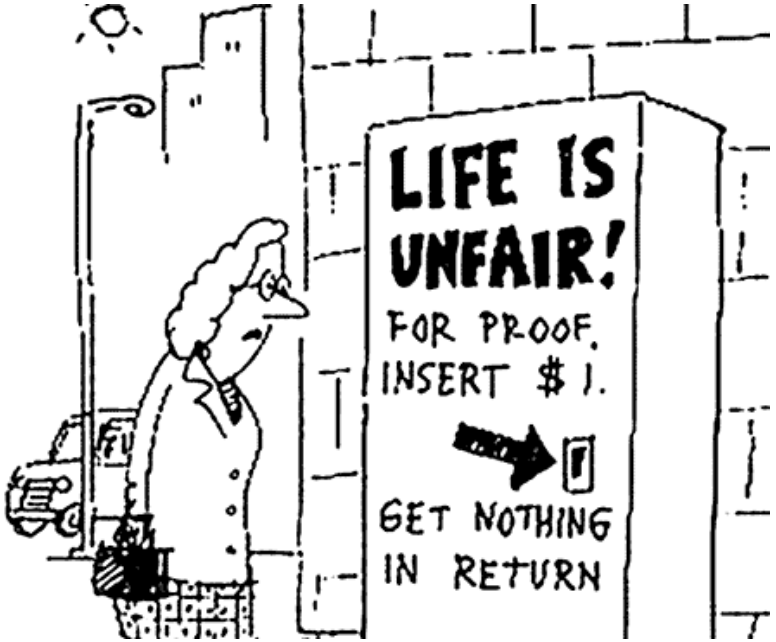


"My daughter and I taking a shower with equal frequency is a frightening thought for both of us." -- E.W. Dijkstra

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

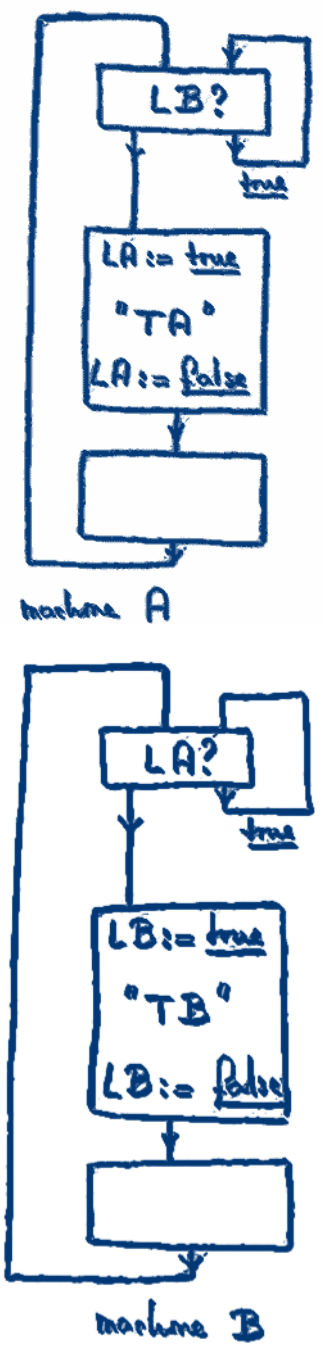
Kritischer Abschnitt: Historie

- 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



Kritischer Abschnitt: Historie (2)

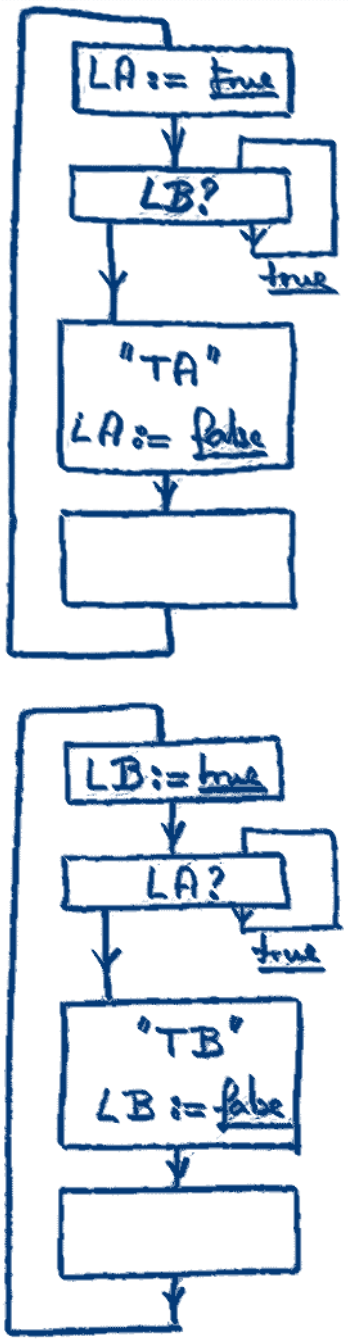
So we have been too optimistic. The error has been that the one machine did not know whether the other was already inquiring about his state of progress. The schemata of Fig. 2 make a much more reliable impression, and it is easy to verify that they are totally secure. E.g. machine A can only start to 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*?

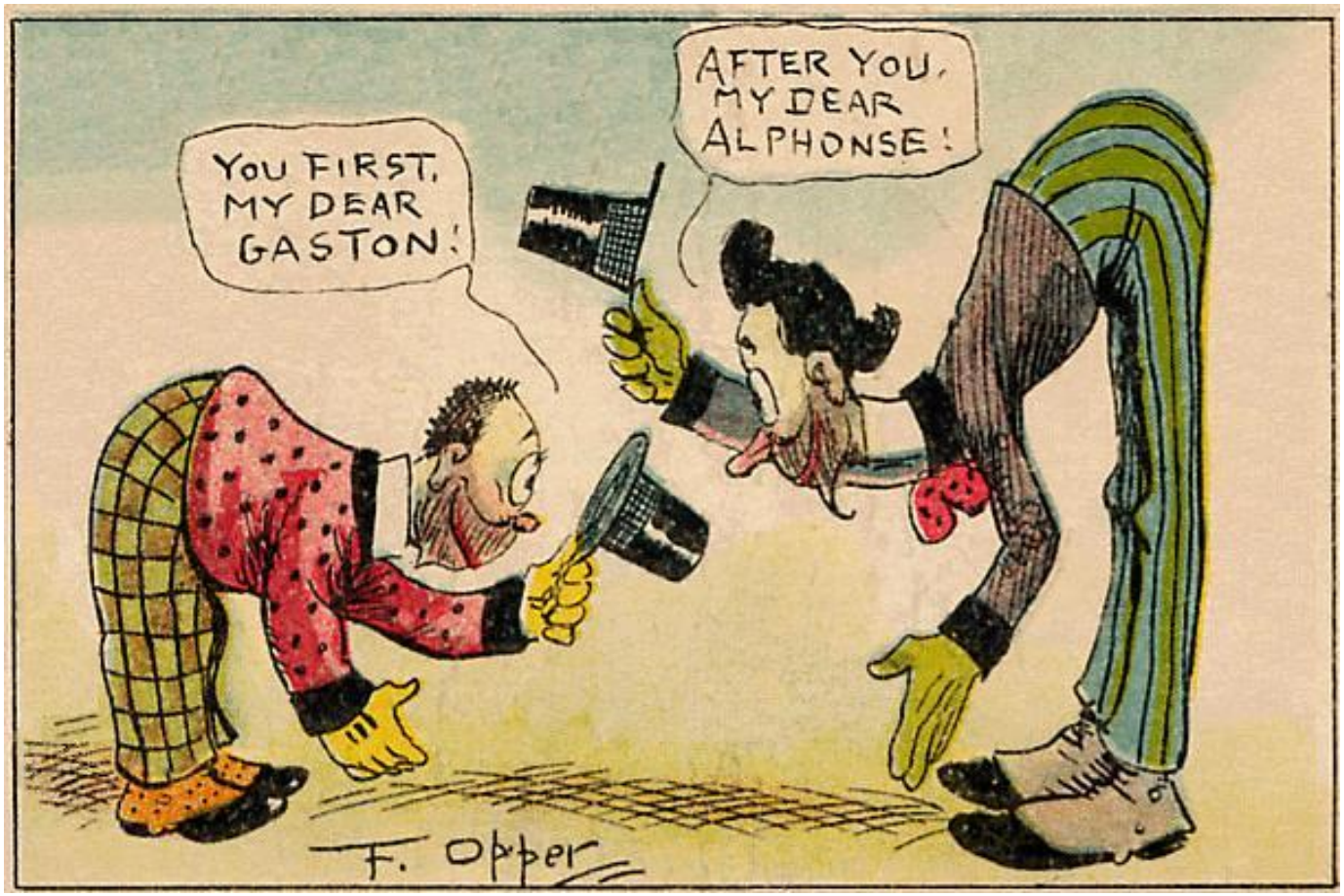
- Wir gehen an dieser Stelle nicht weiter auf die Lösung von Dekker und weitere im Aufsatz beschriebene Lösungen ein, in denen Dijkstra auch seine bekannten Semaphore (mit P- und V-Operationen) einführt



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]

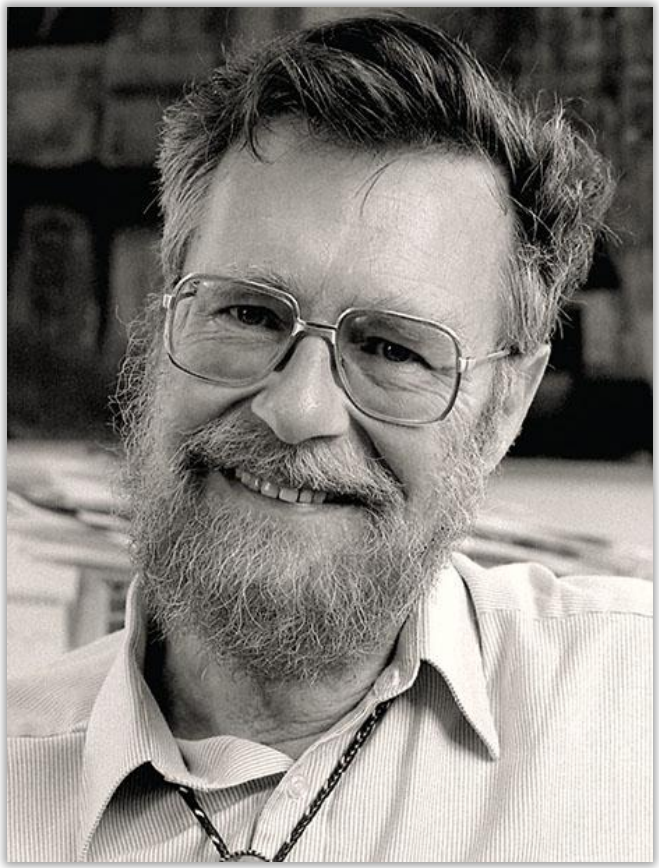
E.W. Dijkstra: Kritische Abschnitte und mehr

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 Multiprogramming-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. [...]

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 your-

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 (1931 – 2012) 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!"

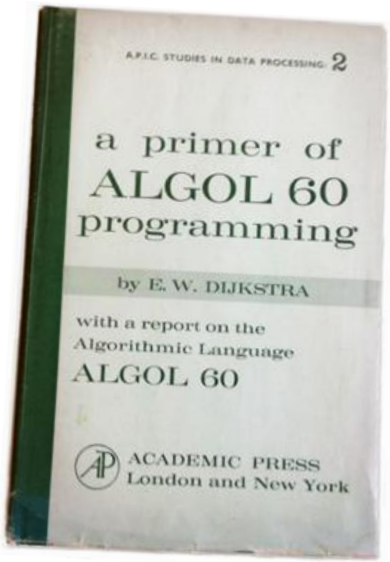
Ria Debets und Edsger Dijkstra, ca.1956

E.W. Dijkstra... (4)

After all, it is no longer the purpose of programs to instruct our machines; these days, it is the purpose of machines to execute our programs. – E.W. Dijkstra

self 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.



Das Buch „A Primer of Algol 60 Programming" war viele Jahre lang die Standard-Referenz zu Algol.

Im Jahr 2000 schrieb Dijkstra rückblickend [EWD1298:] "The ALGOL implementation was one of my proudest achievements. Its major significance was that it was done in 8 months at the investment of less than 3 man-years. The rumour was that IBM's FORTRAN implementation had taken 300 man-years and thus had been a project in size way beyond what a university could ever hope to undertake. By requiring only 1% of that labour, our compiler returned language implementation to the academic world, and that was its political significance."

E.W. Dijkstra... (5)

The guru effect can lead to funny consequences, as with the famous computer scientist whose disciples you could spot right away in conferences by their sandals and beards [...], carefully patterned after the master's. – Bertrand Meyer

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.

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.

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

ir.cwi.nl/pub/4155/04155D.pdf

Was mit ihren Bärten nach der Fertigstellung des Compilers zu geschehen habe, wurde nicht vereinbart. Während Dijkstra seinen Bart behielt, rasierte sich Jaap Zonneveld.

E.W. Dijkstra... (6)

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. □

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

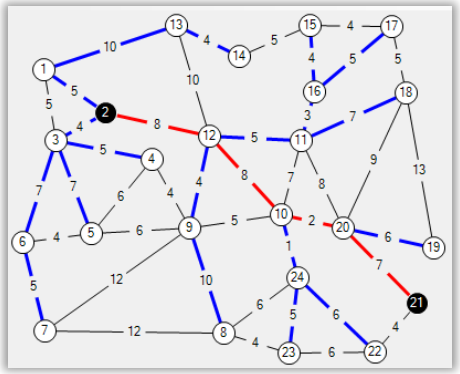
http://pulkomandy.tk/drop/GOTO500.png

1981: F.L. Bauer, Edsger Dijkstra, Tony Hoare während der Summer School "Theoretical Foundations of Programming Methodology" in Marktoberdorf.

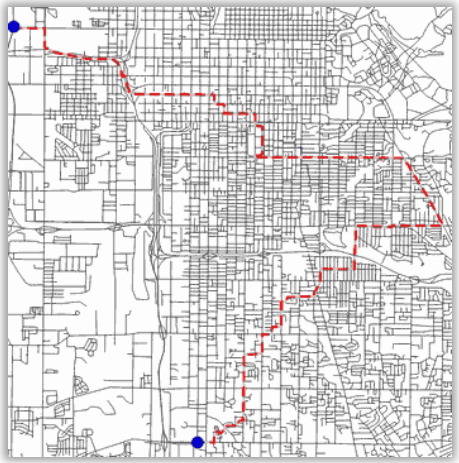
1990 berichtet ein Teilnehmer der jährlich stattfindenden Summer School: „Tony Hoare war ein liebenswürdiger britischer Professor. Am Wochenende zwischen den beiden Summer-School-Wochen machten wir eine Wanderung auf einen recht hohen Alpengipfel. Oben angekommen, sagten wir: *Tony, you should give a lecture now, to get in the books for the highest-level lecture ever given*. Seine etwas trockene Antwort oben auf dem Berg: *I guess the topic of the lecture will be:* 'On the avoidance of jumps."'

https://tweakers.net/nieuws/217146/pascal-bedenker-en-programmeertaalpionier-niklaus-wirth-is-overleden.html

Dijkstra in jedem Navi

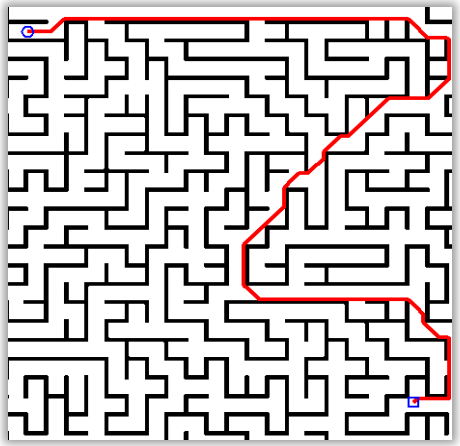


Sebastian Stiller erzählt in seinem lesenswerten Buch „Planet der Algorithmen“ (2015) die Geschichte des oben erwähnten „Dijkstra-Algorithmus“ zur Berechnung des kürzesten Weges zwischen einem Startknoten und den übrigen Knoten in einem (kantengewichteten) Graphen so:



„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 übrigens das gleiche Problem: Verbindungslänge in Metern oder Sekunden; schnell = kurz bzgl. Zeit.

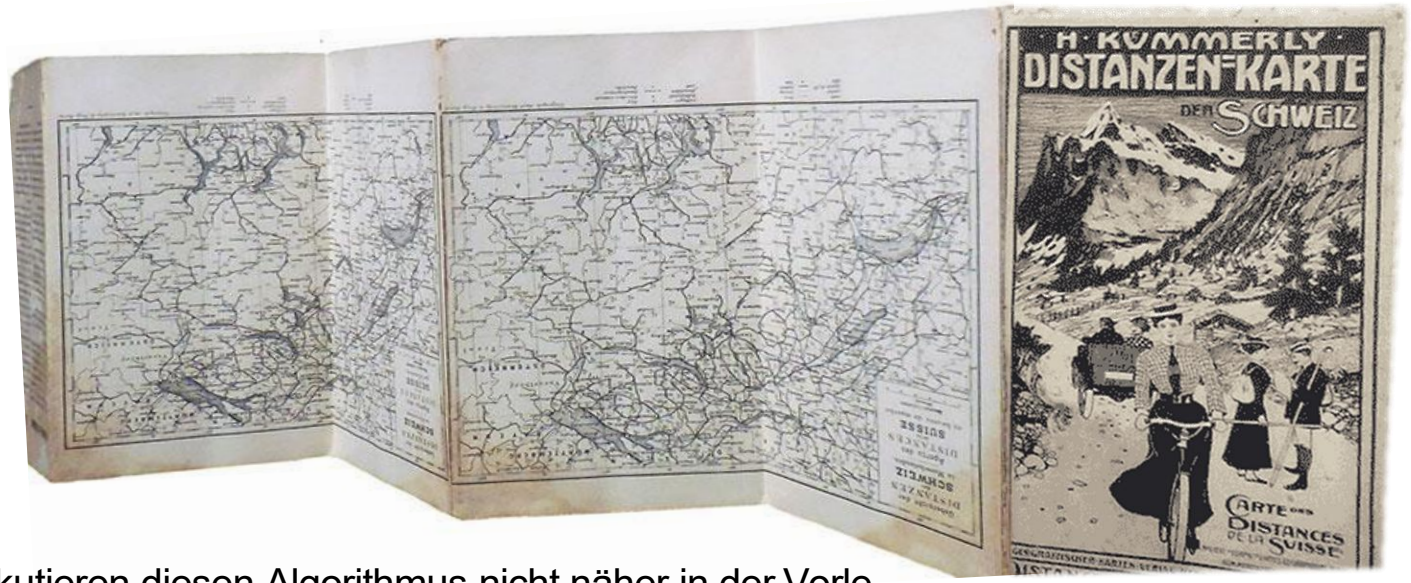
Die Grundidee des „Dijkstra-Algorithmus“

Tell: Nennt mir den nächsten Weg
nach Arth und Küssnacht.

Fischer: Die offne Strasse zieht sich über Steinen,
Doch einen kürzern Weg und heimlichern
Kann Euch mein Knabe über Lauerz führen.

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; in Informatik I tauchte er in einer Clicker-Übung auf. Mehr zum Algorithmus siehe z.B. de.wikipedia.org/wiki/Dijkstra-Algorithmus

Zur Laufzeit des „Dijkstra-Algorithmus“ [Nach Till Tantau]

Sei n die Zahl der Knoten; m die Zahl der Kanten:

- *Naive Implementation:*
  Man muss n mal den nächsten Knoten finden, was jeweils n^2 lang dauert, → insgesamt $O(n^3)$.
- *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 stets das Minimum der Liste nehmen und erreicht eine Laufzeit von $O(n^2)$.
- *Clevere Implementation:*
  Man nutzt eine Heap-Datenstruktur. Dies ermöglicht eine Laufzeit von $O((n+m) \log n)$.
- *Ganz clevere Implementation:*
  Man nutzt sog. Fibonacci-Heaps, was eine Laufzeit von $O(n \log n + m)$ erlaubt.

Ein typischer Graph des europäischen Strassennetzes besteht aus ca. 24 Millionen Knoten und 58 Millionen Kanten.

Optimierung des „Dijkstra-Algorithmus“

Wir zitieren aus www.armin-p-barth.ch/mathe/referate/pdf/20140108.pdf von Armin Barth:

„Die Hauptkritik an Dijkstras Algorithmus lautet so: Es macht oft wenig Sinn, den gesamten Graphen abzusuchen; sinnvoller wäre es, das Suchgebiet möglichst einzuschränken auf den im Hinblick auf die Zielsetzung relevanten Teil. Der klassische *A*-Search-Algorithmus* tut genau dies. Er benutzt geschickte Abschätzungen der Distanzen zum Zielknoten, um damit die Auswahl der Knoten zu steuern, die besucht werden sollen. Im Jahr 2005 hatten Andrew V. Goldberg und Chris Harrelson eine glänzende Idee, wie das Suchgebiet noch mehr eingeschränkt werden kann. Ihre sogenannten *ALT-Algorithmen* (A* plus Landmarks plus Triangle Inequality) wählen zuerst einige Landmarks $L_1, L_2, L_3, \ldots$ aus und berechnen und speichern dann die kürzesten Distanzen sämtlicher Knoten zu diesen Landmarks. Es bezeichne $d_i(v)$ eine solche im Voraus berechnete kürzeste Distanz des Knotens *v* zum Landmark L_i. Wenn nun jemand den kürzesten Weg von Knoten *v* zu Knoten *w* wissen will, so ist aufgrund der Dreiecksungleichung klar, dass

$$d(v,w) \geq d_i(v) - d_i(w)$$

sein muss. Indem man das Maximum all dieser unteren Schranken über alle Landmarks wählt, findet man eine gute untere Schranke für die gesuchte Distanz und kann diese benutzen bei der eigentlichen Berechnung des kürzesten Weges.“

Der kürzeste (oder schnellste) Weg ist nicht immer der beste. Aber in welcher Hinsicht ein Weg „besser“ als ein anderer ist, ist oft Ansichtssache. Wenn alle der Empfehlung eines Algorithmus folgen, hat das Konsequenzen für Anwohner, Ökologie und Ökonomie. Der Tagesspiegel schrieb: *„Stadtplaner sind manchmal entsetzt, wenn man ihnen zeigt, welche Wege die Algorithmen vorschlagen", sagt Johannes Schöning von der Universität Bremen. Er kritisiert, dass die digitalen Kartendienste von der realen Welt entkoppelte Entscheidungen treffen und fordert eine gesellschaftliche Debatte.*

Java: das Synchronized-Konstrukt

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

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

Hier steht Ω für irgendeine Objektreferenz („Sperrobjekt“; „lock“)

- Meist wird Ω 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!

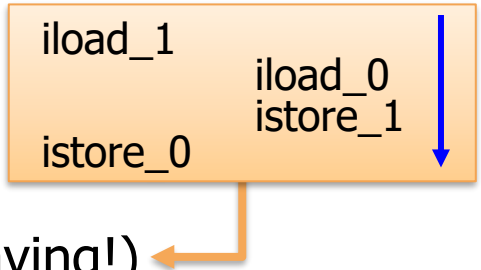
Verkehrsampeln bieten auch keine absolute Sicherheit (vor Rotlichtignoranten!)

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 Language 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!)



- 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();
  }
}
```

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 man hier kein static hätte?

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

Des Rätsels Lösung –

Erläuterung zur Variante ohne „synchronized“

(angelehnt an Reinhard Schiedermeier: Programmieren mit Java II)

> With concurrency, you’re on your own, and only by being both suspicious and aggressive can you write multithreaded code in Java that will be reliable. – Bruce Eckel, Autor von „Thinking in Java“.

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

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

Der Anweisung j++ entspricht konkret folgender Bytecode:

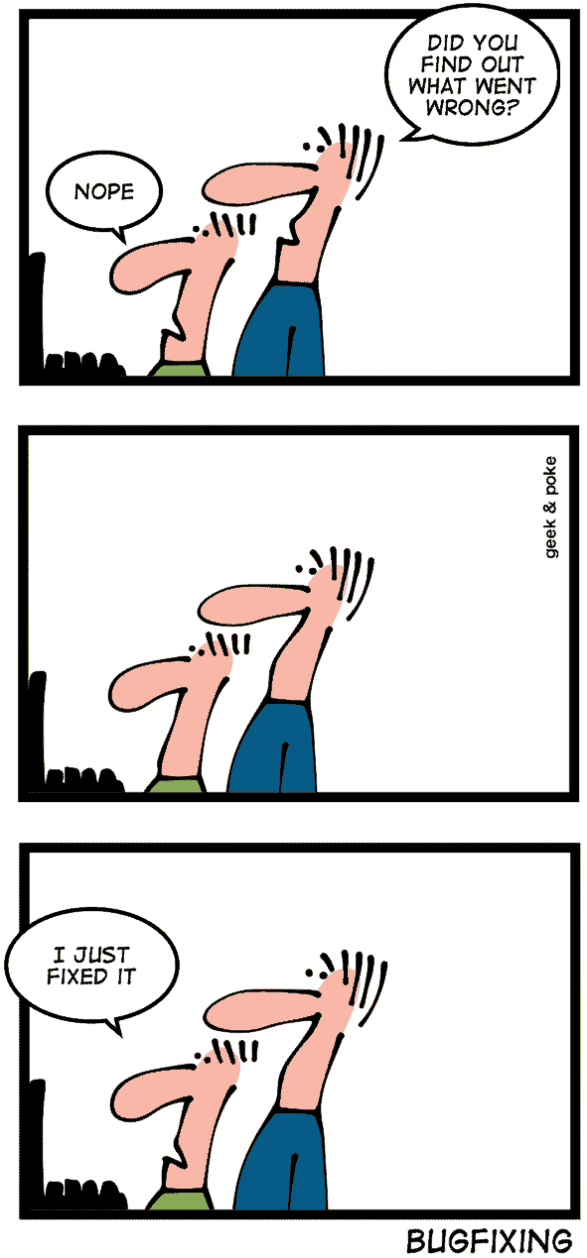
```
...
getstatic   -- Der Wert der Klassenvariablen j wird geholt
iconst_1    -- Die Konstante 1 wird bereitgelegt
iadd        -- Integer-Addition von Variablenwert j und Konstante 1
putstatic   -- Summe in die Klassenvariable j zurückschreiben
...
```

Das erweist sich manchmal als ein lost update!

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.

> Java is a multithreaded language, and concurrency issues are present whether you are aware of them or not. As a result, there are many Java programs in use that either just work by accident, or work most of the time and mysteriously break every now and again because of undiscovered concurrency flaws. – Bruce Eckel.



Verbotene Früchte? Die Kosten der Koordination

Die obige Bemerkung „Die Ausführung dauert ca. 200-mal länger als ohne ‚synchronized'" gibt zu denken. Tatsächlich ist generell die Synchronisation und Koordination von Prozessen zur Vermeidung möglicher Inkonsistenzen in parallelen Systemen, und insbesondere in verteilten Systemen, wo die einzelnen Prozesse nicht über einen gemeinsamen Speicher, sondern mittels relativ aufwändigen und vergleichsweise langsamen Nachrichten kommunizieren, zeitaufwändig, ressourcenfressend und daher „teuer" – Anwendungen können so spürbar verlangsamt werden (auch wenn der Faktor 200 des obigen Beispiels nicht verallgemeinerbar ist). Daher setzt man in der Praxis entsprechende Mechanismen nur zögerlich (oft vielleicht zu zögerlich!) ein, wenn es anders tatsächlich nicht geht und Inkonsistenzen nicht tolerierbar sind.

Joseph Hellerstein (University of California at Berkeley) und Peter Alvaro (University of California at Santa Cruz) drückten es im September 2020 so aus: „Unfortunately, the expense of coordination protocols can make them 'forbidden fruit' for programmers. James Hamilton from Amazon Web Services made this point forcefully […]: 'The first principle of successful scalability is to batter the consistency mechanisms down to a minimum, move them off the critical path, hide them in a rarely visited corner of the system, and then make it as hard as possible for application developers to get permission to use them.' The issue is not that coordination is tricky to implement, though that is true. The main problem is that coordination can dramatically slow down computation."

Lucas Cranach d. Ä., 1526

J. M. Hellerstein & P. Alvaro (2020). Keeping CALM: when distributed consistency is easy. Comm. ACM, 63(9), 72-81, 10.1145/3369736.

Sündenfall und Vertreibung aus dem Paradies

Wir notierten oben „wenn es anders tatsächlich nicht geht und Inkonsistenzen nicht tolerierbar sind“. In der Praxis wird dazu eine Kostenabwägung vorgenommen. Sind seltene Inkonsistenzen möglich, deren grundsätzliche Vermeidung aber sehr aufwändig, dann mutet man Anwendern eines Systems (z.B. Kunden eines Web-Shops) oft zu, gelegentlich mit einem inkonsistenten Zustand oder abgestürzten System zurechtzukommen und die Transaktion erneut durchzuführen oder den Reset-Knopf am Gerät zu betätigen. Warum es nicht funktioniert hat, erfährt der Kunde nicht – schon gar nicht, dass die Systemdesigner dies bewusst in Kauf genommen haben. Klar: Bei sicherheitskritischen Systemen ist so etwas natürlich nicht tolerierbar, es wäre eine Todsünde, um im Bild zu bleiben! Ansonsten aber darf man nicht der Illusion, dass die Praxis so sündenfrei wie das Paradies sei, unterliegen. Wir lassen nochmal Hellerstein und Alvaro zu Wort kommen, die ein von Dave Campbell und Pat Helland (seinerzeit beide bei Microsoft beschäftigt) eingeführtes „Entschuldigungsparadigma“ („You may have an ‘Oh, crap!’ moment. That may involve apologizing for your behavior“) aufgreifen und meinen, dass es manchmal adäquater sei, sich von der Sündentat einer Inkonsistenz durch eine „Entschuldigung“ reinzuwaschen, als diese durch ein Treuegelöbnis und eine aufwändige Koordination präventiv auszuschliessen:

“For example, when a retail site allows you to purchase an item, it should decrement the count of items in inventory. This […] suggests that coordination is required, for example, to ensure that the supply is not depleted before an item is allocated to you. In practice, this requires too much integration between systems for inventory, supply chain, and shopping. In the absence of such coordination, your purchase may fail non-deterministically after checkout. To account for this possibility, additional compensation code must be written to detect the out-of-stock exception and handle it by—for example—sending you an apologetic email with a loyalty coupon. Note that a coupon is not a clear mathematical inverse of any action in the original program; domain-aware compensation often goes beyond typical type system logic.”

Zeitmessungen zum parallelen Inkrementieren

- Wieviel kostet obige Synchronisation? Einige Zeitmessungen:

1. Ohne synchronized (die fehlerhafte Variante!): 0.82s real, 2.99s cpu
2. Mit synchronized (wie weiter oben dargestellt): 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 in 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

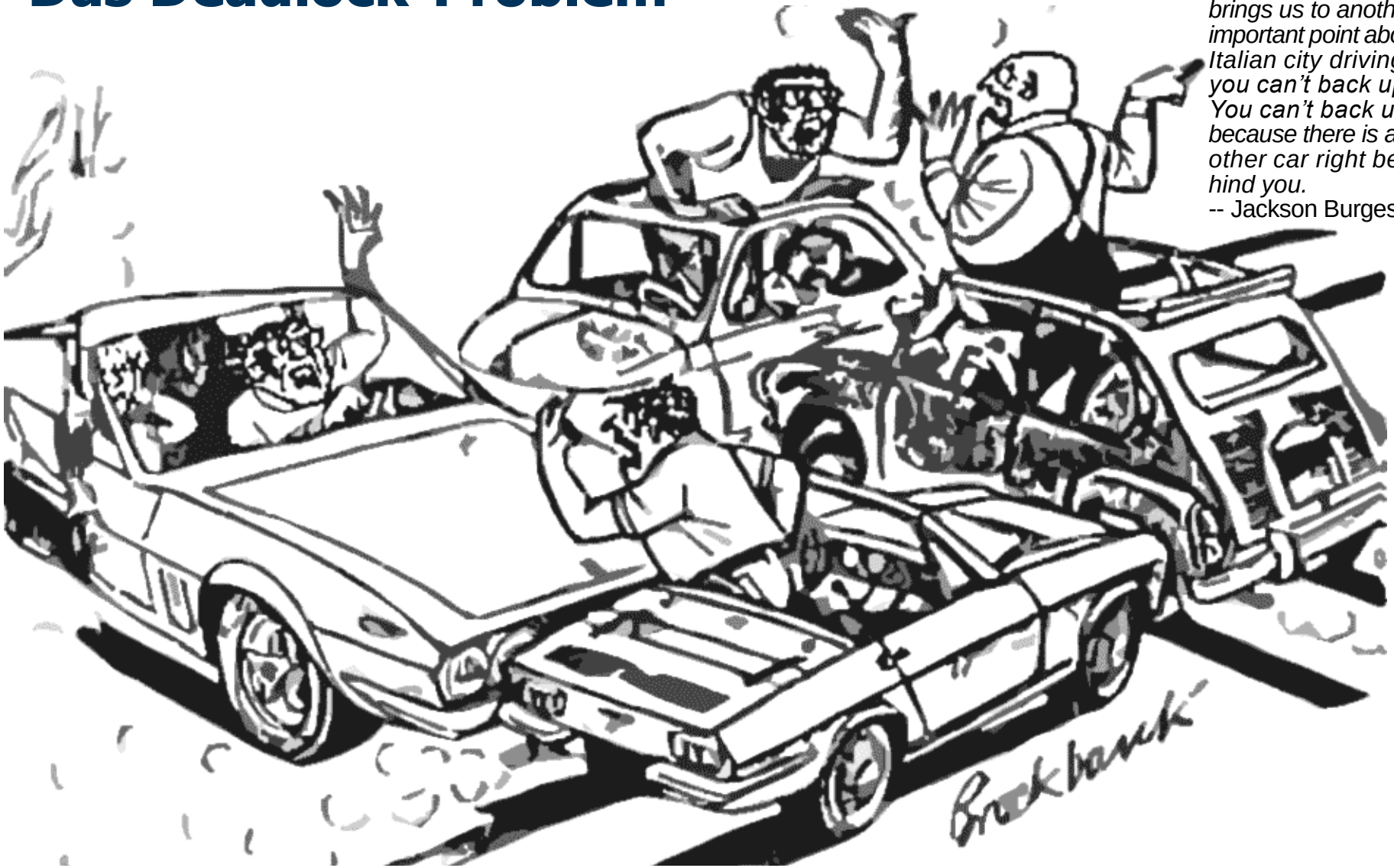
```
public class … {
  static int z = 0;
  public synchronized void Incr() {
    println(z++);
  }
  // Gründen von zwei parallelen Threads,
  // welche beide fortlaufend Incr aufrufen;
}
```

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

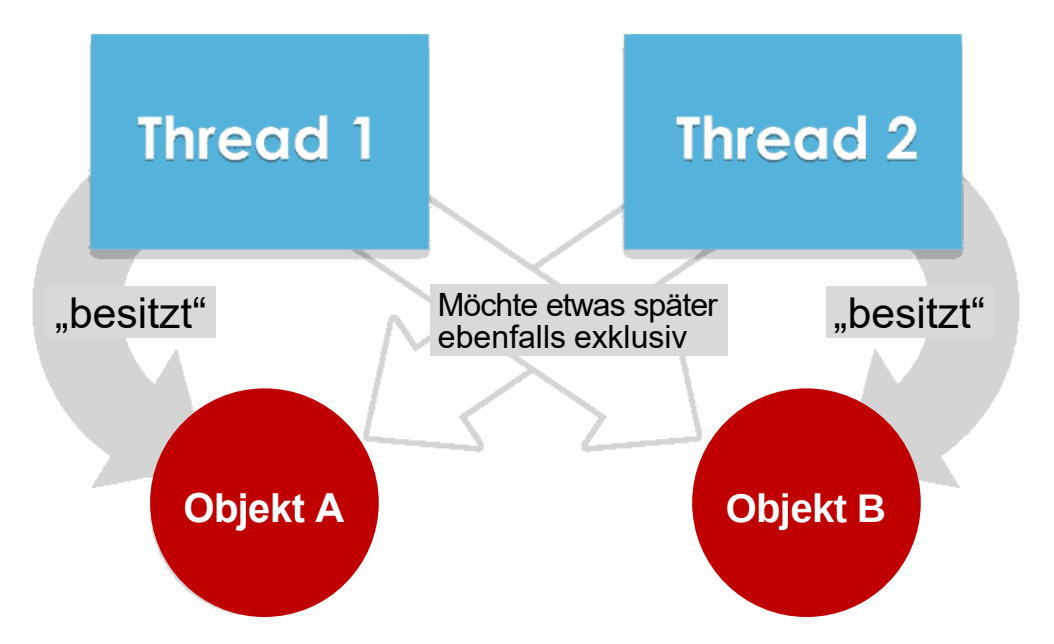
The basic rule of driving in Italian cities is: force your car as far as it will go into any opening in the traffic. It is this rule which produces the famous Sicilian Four-Way Deadlock. Sharp study suggests that the Deadlock can be broken if any one of the cars backs up. That brings us to another important point about Italian city driving: you can't back up. You can't back up because there is another car right behind you.
-- Jackson Burgess

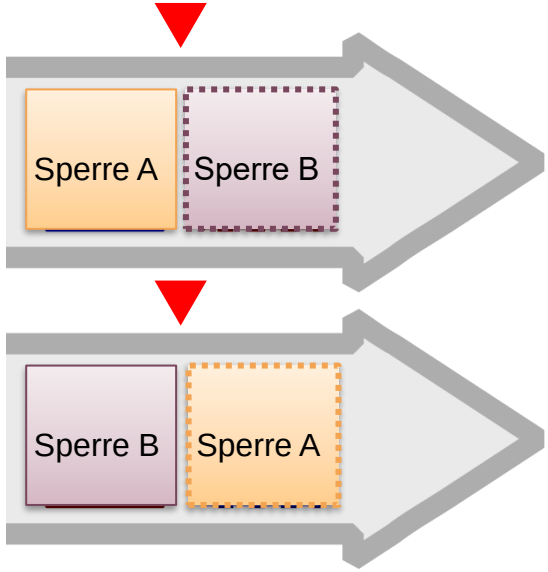


Neapolitanischer Hakenkreuzstau, nach Luciano De Crescenzos Beschreibung im Film „Also sprach Bellavista“: Ein unentwirrbar mit sich selbst verschränktes Problem – die Wohnung müsste neu gestrichen werden, was nur geht, wenn vorher das herumstehende Zeug aus dem Weg geschafft wird, wozu man immer schon 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

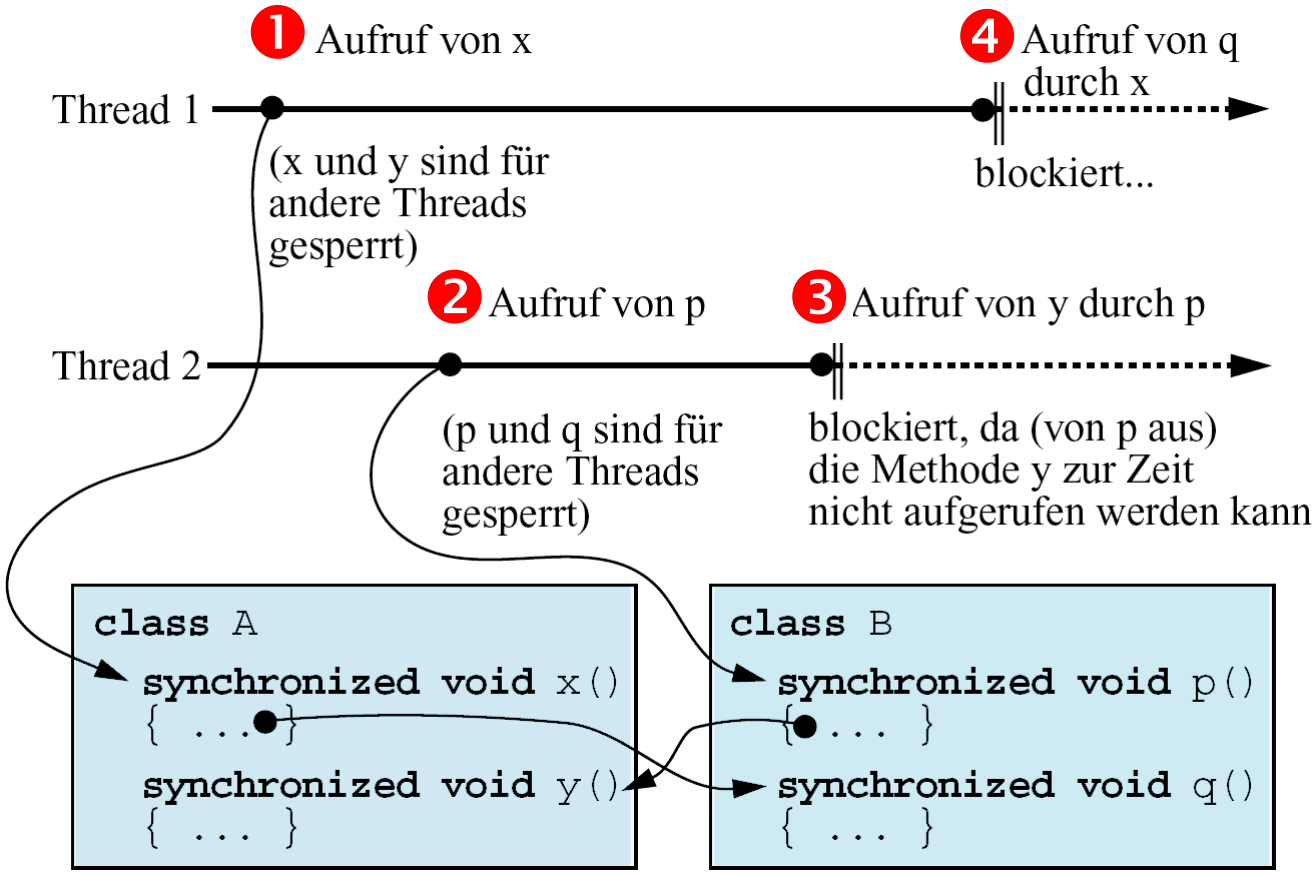
Verklemmung zweier Threads





- „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…

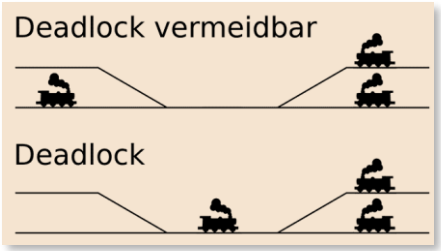
- Thread 1 wartet auf Freigabe von B durch Thread 2
- Thread 2 wartet auf Freigabe von A durch Thread 1

} **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.

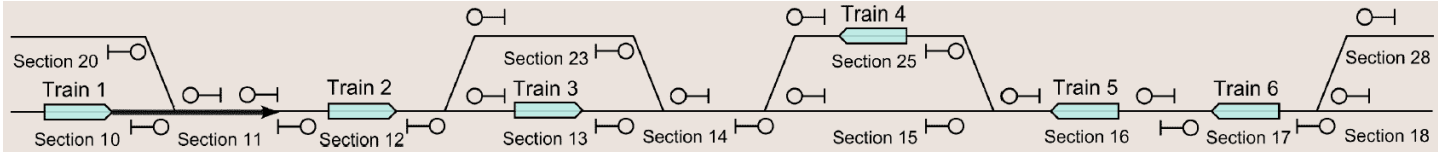
Auszüge aus de.wikipedia.org/wiki/Deadlock_(Eisenbahn), basierend auf *Jörn Pachl, Deadlock Avoidance in Railroad Operations Simulations, 2011:*



Ein Deadlock ist bei der Eisenbahn eine Situation, in der sich Züge gegenseitig blockieren, sodass 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".

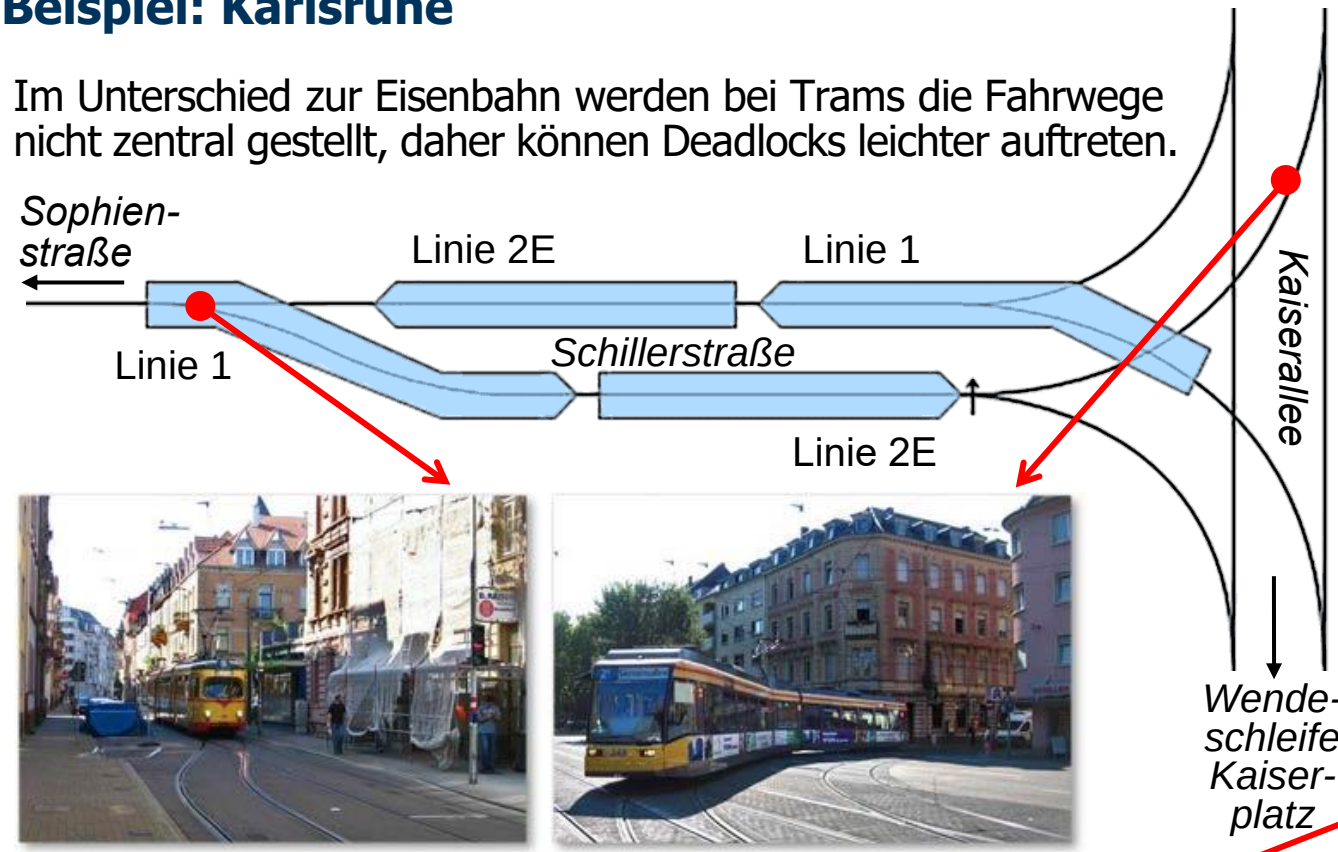


Fährt Zug 1 in Abschnitt 11, dann kommt es garantiert zu einem Deadlock, unabhängig von Geschwindigkeit und Scheduling der Züge.

Deadlocks bei der Strassenbahn

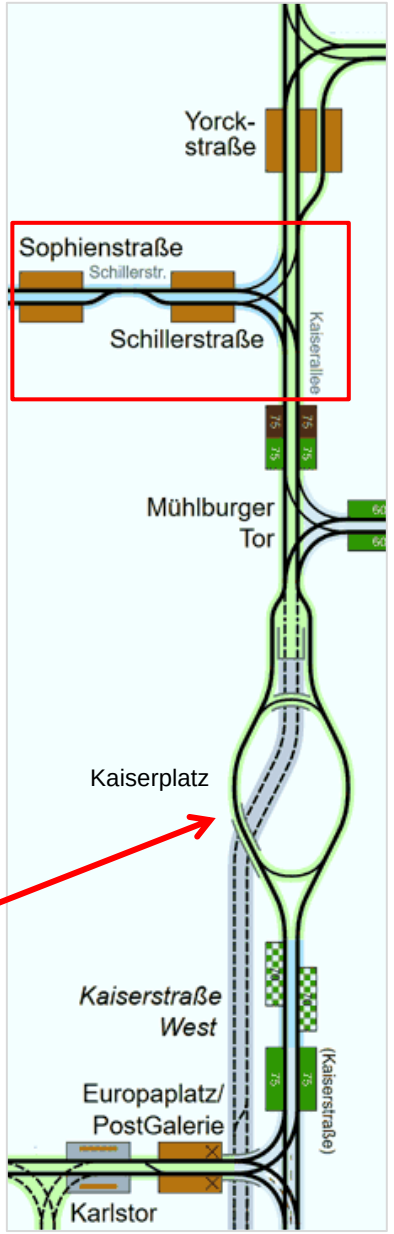
Beispiel: Karlsruhe

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



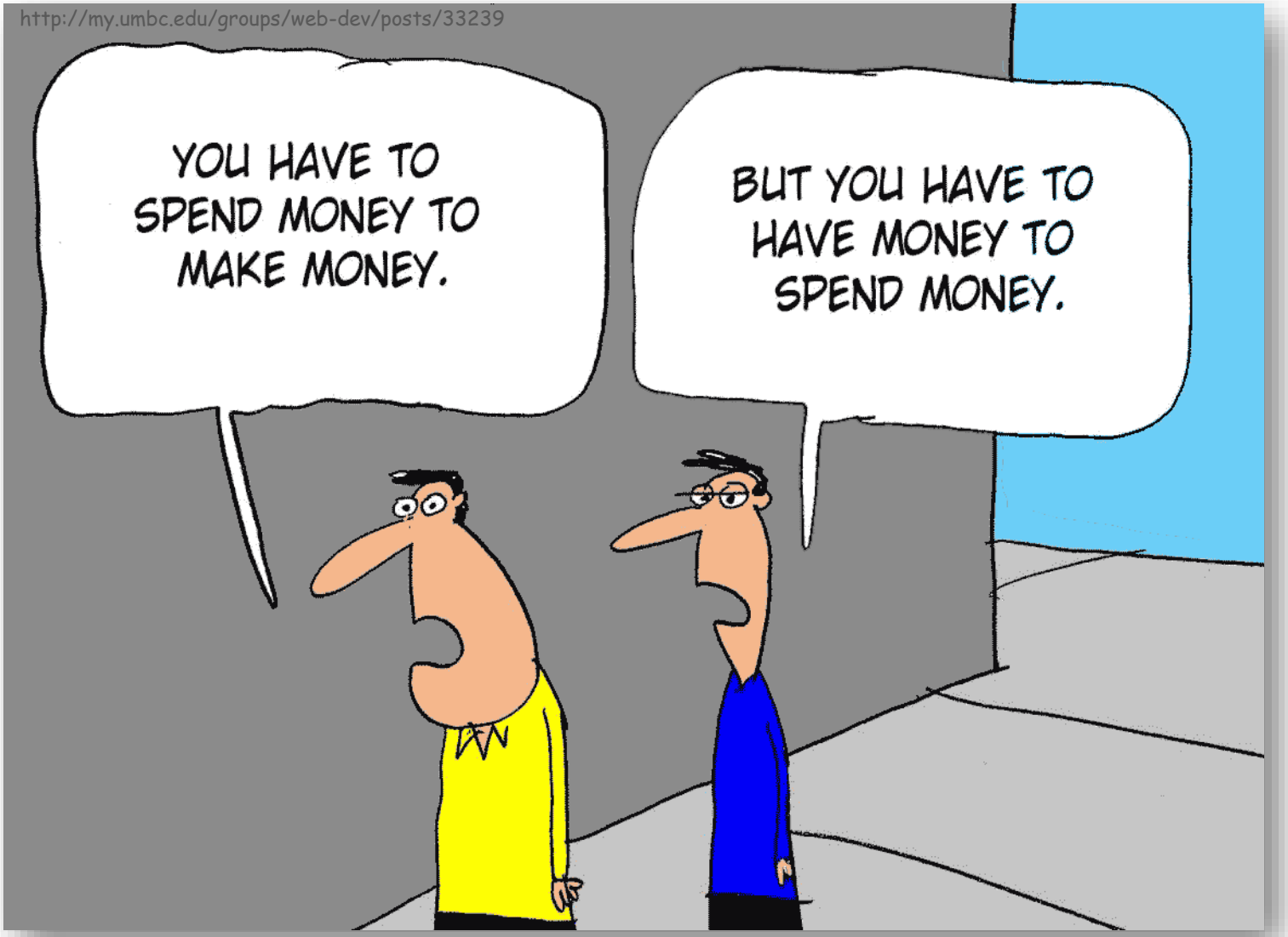
Eingleisige Schillerstraße

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


http://my.umbc.edu/groups/web-dev/posts/33239
YOU HAVE TO SPEND MONEY TO MAKE MONEY.
BUT YOU HAVE TO HAVE MONEY TO SPEND MONEY.

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

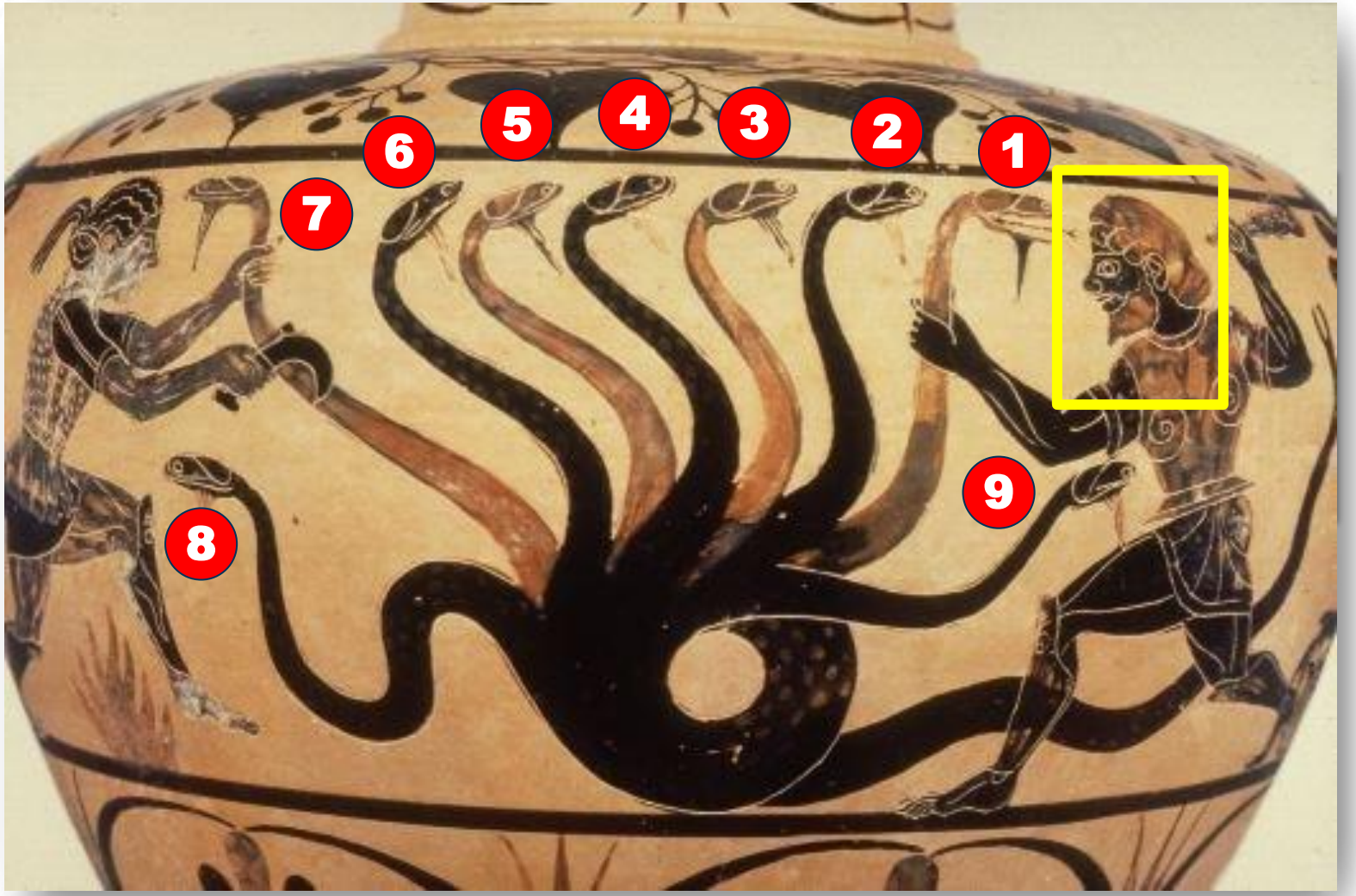
---

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

Das Ringen mit der Parallelität

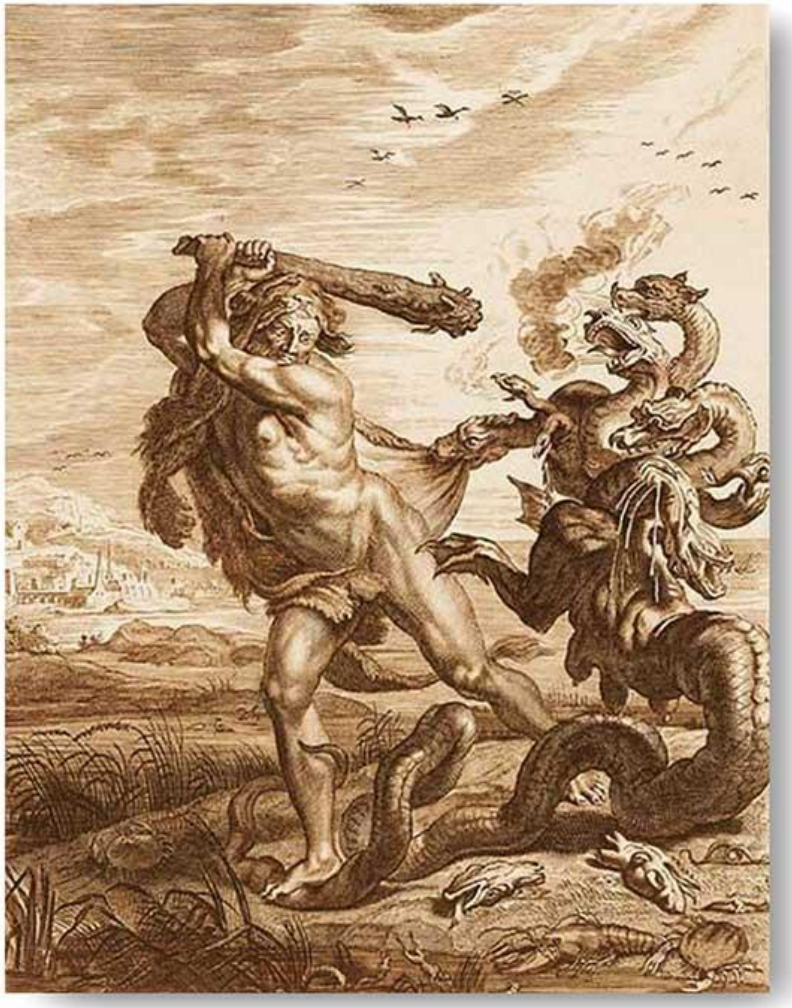
Der richtige Umgang mit Parallelität und die Beherrschung der damit verbundenen Phänomene ist eine Herausforderung!

Das Ringen mit der Parallelität - eine Herkulesaufgabe!



Das Ringen mit der Parallelität

Das Ringen mit der Parallelität

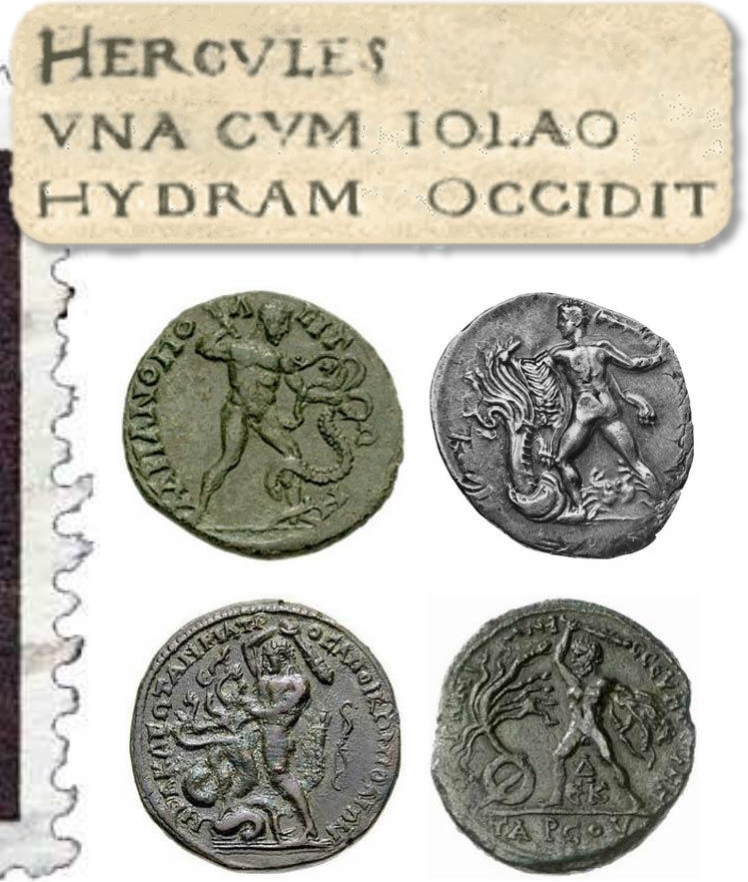


Das Ringen mit der Parallelität

Der Augsburger Herkulesbrunnen wurde ab 1597 vom niederländischen Bildhauer Adriaen de Vries modelliert und 1602 auf dem Weinmarkt aufgestellt. Auf einem mit Nymphen, Meeresgöttern und Putten kunstvoll verzierten Pfeilerblock steht eine drei Meter hohe Bronzegruppe mit dem muskulösen und bärtigen Herkules. In seiner Hand hält er eine Flammenkeule, um das siebenköpfige Ungeheuer, die Hydra, zu erschlagen. Nach der Sage benötigte Herkules die Flammenkeule, um die Wurzeln der abgeschlagenen Köpfe zu versengen und die Hydra so zu hindern, neue Köpfe hervorzutreiben.

Das Bezwingen der Parallelität

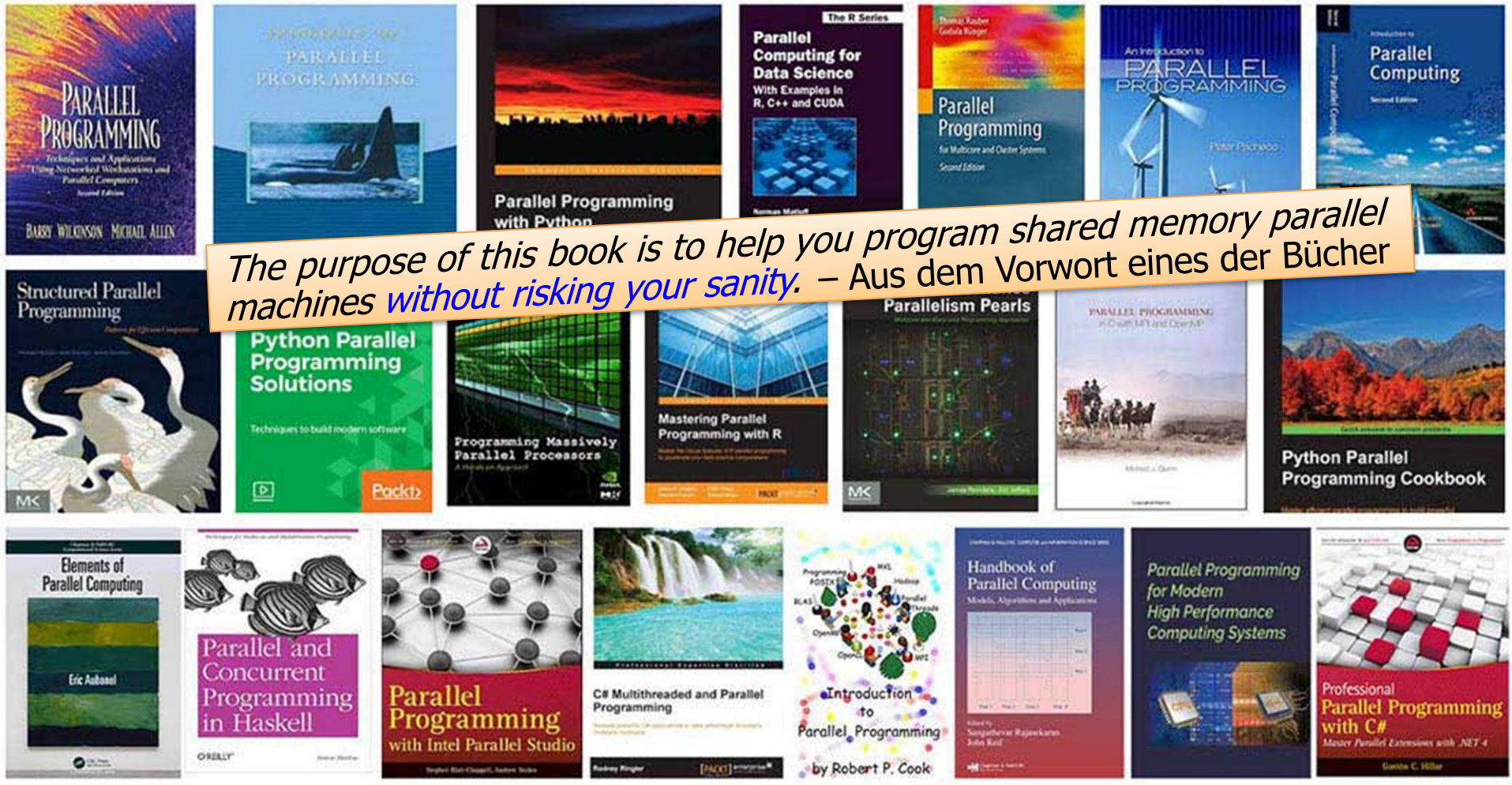
Das Bezwingen der Parallelität



Ein sagenhaftes und uraltes Problem!

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!

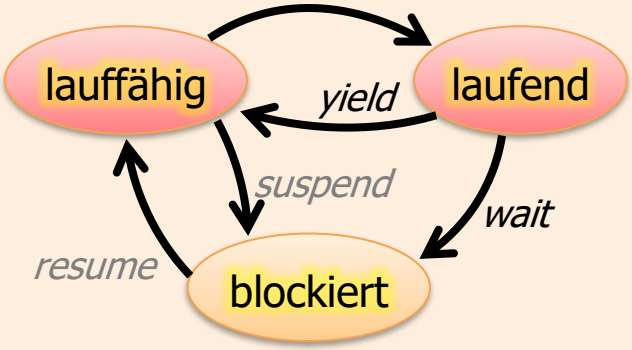
echte

- 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 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, join, (stop, suspend, resume)…
- 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