Echtzeitsysteme
Wintersemester 2007/2008

Prof. Dr. Alois Knoll

TU München

Lehrstuhl VI Robotics and Embedded Systems

Echtzeitsysteme: Organisation

Bestandteile der Vorlesung

- Vorlesung:
  - Donnerstag 10:15-11:00 Uhr MI HS 2
  - **Freitag 10:15-11:45 Uhr MI HS 1**
  - 6 ECTS Punkte
  - Wahlpflichtvorlesung im Gebiet Echtzeitsysteme (Technische Informatik)
  - Wahlpflichtvorlesung für Studenten der Elektro- und Informationstechnik
  - Pflichtvorlesung für Studenten des Maschinenbau Richtung Mechatronik
- Übung:
  - zweistündige Tutorübung, im Raum 03.05.012
  - Mittwochs 14:00-15:30 Uhr
  - Mittwochs 15:30-17:00 Uhr
  - Weitere Termine bei Bedarf nach Vereinbarung
  - Beginn: voraussichtlich ab 31.10.07
- Prüfung:
  - Schriftliche Klausur am Ende des Wintersemesters, falls ein Schein benötigt wird.

Team

Prof. Dr. Alois Knoll

Dipl.-Inf. Christian Buckl

Zentralübung

Dipl.-Inf. Matthias Regensburger

Übungen

Weitere Informationen

- Homepage der Vorlesung mit Folien, Übungsaufgaben und weiterem Material:

  http://www6.in.tum.de/tum6/lectures/courses/ws0708/esys

- Übungsleitung
  - Email: buckl@in.tum.de, regensbu@in.tum.de

Informationen zur Übung

- Die Übung dient der praktischen Umsetzung des Vorlesungsinhaltes durch Programmierung.
- Es werden diverse Aufgaben aus dem Bereich der Echtzeitprogrammierung angeboten, wie z.B. Aufgaben zu Threads, Semaphore, Kommunikation
- Programmiersprache ist überwiegend C, zu Beginn der Übung wird eine kurze Einführung in C angeboten
- Informationen zur Anmeldung:
  - Teilnehmerzahlen an der Übung sind begrenzt
  - Die Anmeldung erfolgt über die Grundstudiumsseite: https://grundstudium.in.tum.de
  - Zur Anmeldung ist ein Zertifikat der Rechnerbetriebsgruppe nötig. Nähere Informationen finden Sie unter http://ca.in.tum.de/userca/
  - Studenten, ohne Login für die Informatikhalle, erhalten ihr Zertifikat unter http://grundstudium.in.tum.de/zertifikat

Weitere Angebote des Lehrstuhls

- Weitere Vorlesungen: Robotik, Digitale Signalverarbeitung, Maschinelles Lernen und bioinspirierte Optimierung I&II, Sensor- und kamerageführte Roboter
- Praktika: Echtzeitsysteme, Roboterfußball, Industrieroboter, Neuronale Netze und Maschinelles Lernen, Bildverarbeitung, Signalverarbeitung
- Seminare: Sensornetzwerke, Objekterkennung und Lernen, Neurocomputing
- Diplomarbeiten / Masterarbeiten
- Systementwicklungsprojekte / Bachelorarbeiten
- Join-In
- Unser gesamtes Angebot finden sie unter http://wwwknoll.in.tum.de

Literatur

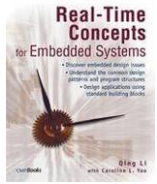
Hermann Kopetz: Real-Time Systems (Überblick)

Jane W. S. Liu: Real-Time Systems
(Überblick, Schwerpunkt Scheduling)

Stuart Bennet: Real-Time Computer Control:
An Introduction (Überblick, Hardware)

Alan Burns, Andy Wellings: Real-Time Systems and Programming
Languages (Schwerpunkt: Programmiersprachen)

Qing Li, Caroline Yao: Real-Time Concepts for
Embedded Systems (Schwerpunkt: Programmierung)



Bill O. Gallmeister: Programming for the Real-World:
POSIX.4 (Schwerpunkt: Posix)

Weitere Literaturangaben befinden sich in den jeweiligen Abschnitten.

Vorlesungsinhalte

1. Einführung Echtzeitsysteme
2. Modellierung und Werkzeuge
3. Nebenläufigkeit
4. Scheduling
5. Echtzeitbetriebssysteme
6. Programmiersprachen
7. Uhren
8. Kommunikation
9. Spezielle Hardware
10. Regelungstechnik
11. Fehlertolerante Systeme

Weitere Themen können bei Interesse aufgenommen werden. Melden Sie sich einfach nach der Vorlesung oder per Email.

Inhalt I

- Kapitel Einführung (ca. 1 Vorlesungswoche)
  - Definition Echtzeitsysteme
  - Klassifikation
  - Echtzeitsysteme im täglichen Einsatz
  - Beispielanwendungen am Lehrstuhl
- Kapitel Modellierung/Werkzeuge (ca. 3 Vorlesungswochen)
  - Allgemein
  - Synchrone Sprachen (Esterel, Lustre), SCADE
  - Zerberus
  - UML
  - Formale Methoden

Inhalt II

- Kapitel Nebenläufigkeit (1 Vorlesungswoche)
  - Prozesse, Threads
  - Interprozesskommunikation
- Kapitel Scheduling (1 Vorlesungswoche)
  - Kriterien
  - Planung Einrechner-System, Mehrrechnersysteme
  - EDF, Least Slack Time
  - Scheduling mit Prioritäten (FIFO, Round Robin)
  - Scheduling periodischer Prozesse
  - Scheduling Probleme

Inhalt III

- Kapitel Echtzeitbetriebssysteme (ca. 2 Vorlesungswochen)
  - QNX, VxWorks, PikeOS
  - RTLinux, RTAI, Linux Kernel 2.6
  - TinyOS, eCos
  - OSEK
- Kapitel Programmiersprachen (1 Vorlesungswoche)
  - Ada
  - Erlang
  - C mit POSIX.4
  - Real-time Java
- Kapitel Uhren (1 Vorlesungswoche)
  - Uhren
  - Synchronisation von verteilten Uhren

Inhalt IV

- Kapitel Echtzeitfähige Kommunikation (1 Vorlesungswoche)
  - Token-Ring
  - CAN-Bus
  - TTP, FlexRay
  - Real-Time Ethernet
- Kapitel: Spezielle Hardware (1 Vorlesungswoche)
  - Digital-Analog-Converter (DAC)
  - Analog-Digital-Converter (ADC)
  - Speicherprogrammierbare Steuerung (SPS)

Inhalt V

- Kapitel: Regelungstechnik (ca. 2 Vorlesungswochen)
  - Definitionen
  - P-Regler
  - PI-Regler
  - PID-Regler
  - Fuzzy-Logic
- Kapitel Fehlertoleranz (ca. 2 Vorlesungswochen)
  - Bekannte Softwarefehler
  - Definitionen
  - Fehlerarten
  - Fehlerhypothesen
  - Fehlervermeidung
  - Fehlertoleranzmechanismen

Offene Studienarbeiten / Werkstudentenstellen

- Themen:
  - Implementierung neuartiger Schedulingverfahren:
    für alle, die schon immer mal (teilweise) ein eigenes Betriebssystem implementieren wollten
  - Zertifizierung von modellbasierten Codegeneratoren (in Zusammenarbeit mit dem TÜV):
    für alle, die später in der Entwicklung von sicherheitskritischen Systemen arbeiten wollen (inkl. kostenlose Schulung vom TÜV)
  - Modell-basierte Entwicklung von Sensornetzwerken (in Zusammenarbeit mit Siemens):
    für alle, die an neuesten Technologien interessiert sind
  - Entwicklung sicherheitskritischer Software im Bahnbereich (in Zusammenarbeit mit Kayser-Threde):
    für alle, die ein konkretes, reales Projekt realisieren wollen
  - Umsetzung von Demonstratoren (z.B. invertiertes Pendel, TTP-Demonstrator, echtzeitfähige Robotiksteuerung, Carrera-Rennbahn):
    für alle, die schon immer die Informatik sehen wollten
  - Einbindung von Doors (Werkzeug für Anforderungsmanagment) in modell-basierten Codegenerator (in Zusammenarbeit mit Lehrstuhl Prof. Broy):
    für alle, die praktische Erfahrung mit in der Industrie eingesetzten Werkzeugen bekommen wollen
  - Modell-basierte Entwicklung von mechatronischen Systemen:
    für alle, die nach dem Motto klein, kleiner, am kleinsten leben
  - Einsatz von formalen Methoden in eingebetteten Systemen:
    für alle, die schon immer die Korrektheit eines Programms beweisen wollten
  - Und vieles mehr…
- Bei Interesse melden Sie sich bei Christian Buckl (buckl@in.tum,de)

Kapitel 1

Einführung Echtzeitsysteme

Inhalt

- Definition Echtzeitsysteme
- Klassifikation von Echtzeitsystemen
- Echtzeitsysteme im täglichen Leben
- Beispielanwendungen am Lehrstuhl

Definition Echtzeitsystem

Ein Echtzeit-Computersystem ist ein Computersystem, in dem die Korrektheit des Systems nicht nur vom logischen Ergebnis der Berechnung abhängt, sondern auch vom physikalischen Moment, in dem das Ergebnis produziert wird.

Ein Echtzeit-Computer-System ist immer nur ein Teil eines größeren Systems, dieses größere System wird Echtzeit-System genannt.

Hermann Kopetz

TU Wien

Resultierende Eigenschaften

⇒ zeitliche Anforderungen

- Zeitliche Genauigkeit
- Garantierte Antwortzeiten
- **Aber nicht:** Allgemeine Geschwindigkeit

⇒ Eigenschaften aufgrund der Einbettung

- Echtzeitsysteme sind typischerweise sehr Eingabe/Ausgabe (E/A)-lastig
- Echtzeitsysteme müssen fehlertolerant sein, da sie die Umgebung beeinflussen
- Echtzeitsysteme sind häufig verteilt

Zeitlicher Determinismus vs. Leistung

- Konsequenz der Forderung nach deterministischer Ausführungszeit: Mechanismen, die die allgemeine Performance steigern, aber einen negativen, nicht exakt vorhersehbaren Effekt auf einzelne Prozesse haben können, dürfen nicht verwendet werden:
  - Virtual Memory
  - Garbage Collection
  - Asynchrone IO-Zugriffe
  - rekursive Funktionsaufrufe

Klassifikation von Echtzeitsystemen

- Echtzeitsysteme können in verschiedene Klassen unterteilt werden:
  - Nach den Konsequenzen bei der Überschreitung von Fristen: harte vs. weiche Echtzeitsysteme
  - Nach dem Ausführungsmodell: zeitgesteuert (zyklisch, periodisch) vs. ereignisbasiert (aperiodisch)

Harte bzw. weiche Echtzeitsysteme

- **Weiche Echtzeitsysteme:**
  Die Berechnungen haben eine zeitliche Ausführungsfrist, eine Überschreitung dieser Fristen hat jedoch keine katastrophale Folgen. Eventuell können die Ergebnisse noch verwendet werden, insgesamt kommt es durch die Fristverletzung evtl. zu einer Dienstverschlechterung.
  Beispiel für ein weiches Echtzeitsystem: Video
  Konsequenz von Fristverletzungen: einzelne Videoframes gehen verloren, das Video hängt

- **Harte Echtzeitsysteme:**
  Eine Verletzung der Berechnungsfristen kann sofort zu fatalen Folgen (hohe Sachschäden oder sogar Gefährdung von Menschenleben) führen. Die Einhaltung der Fristen ist absolut notwendig.
  Beispiel für ein hartes Echtzeitsystem: Raketensteuerung
  Konsequenz von Fristverletzung: Absturz bzw. Selbstzerstörung der Rakete

Unterteilung nach Ausführungsmodell

- Zeitgesteuerte Applikationen:
  - Der gesamte zeitliche Systemablauf wird zur Übersetzungszeit festgelegt
  - Notwendigkeit einer präzisen, globalen Uhr $\Rightarrow$ Uhrensynchronisation notwendig
  - Für die einzelnen Berechnungen ist jeweils ein Zeitslot reserviert $\Rightarrow$ Abschätzung der maximalen Laufzeiten (**worst case execution times - WCET**) notwendig
  - **Vorteil**: Statisches Scheduling möglich und damit ein vorhersagbares (**deterministisches**) Verhalten
- Ereignisgesteuerte Applikationen:
  - Alle Ausführungen werden durch das Eintreten von Ereignissen angestoßen
  - Wichtig sind bei ereignisgesteuerten Anwendungen garantierte Antwortzeiten
  - Das Scheduling erfolgt dynamisch, da zur Übersetzungszeit keine Aussage über den zeitlichen Ablauf getroffen werden kann.

Einführung Echtzeitsysteme

Echtzeitsysteme im Alltag

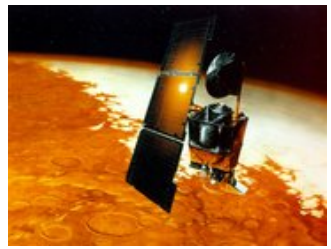
Echtzeitsysteme sind allgegenwärtig!









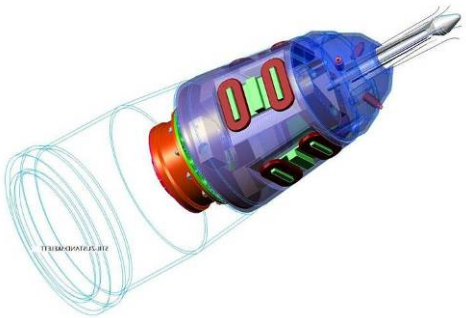


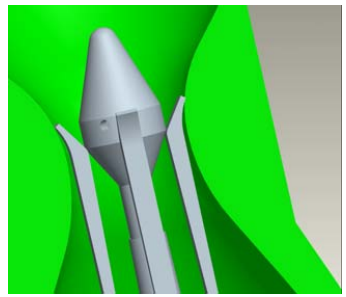
Beispiel: Kuka Robocoaster

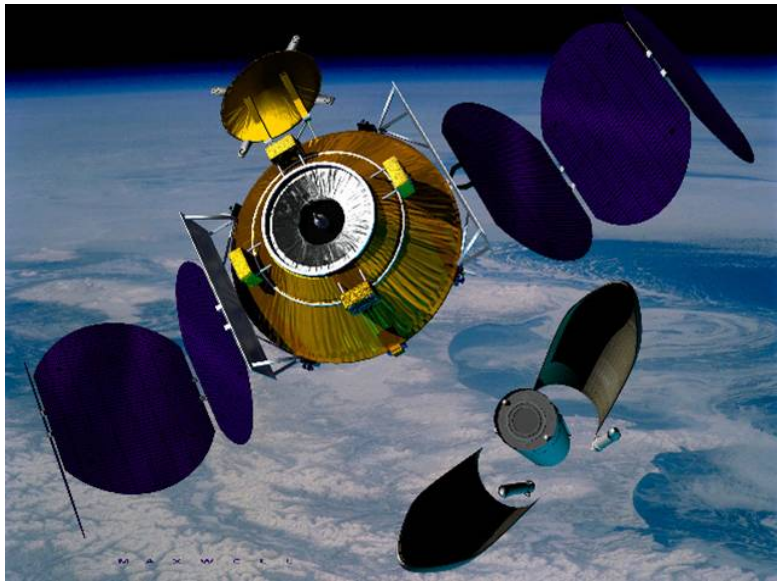


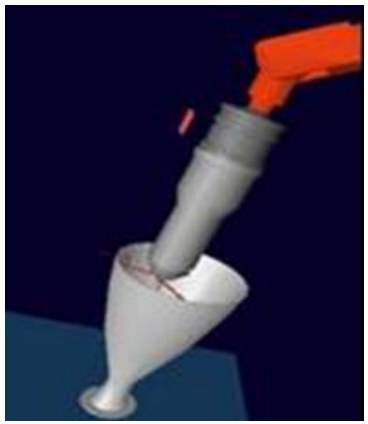

http://www.robocoaster.com

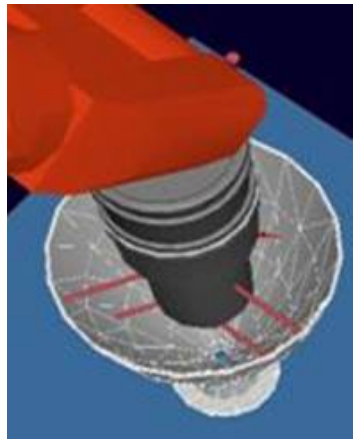
CxOlev - Lebenszeitverlängerung von Satelliten







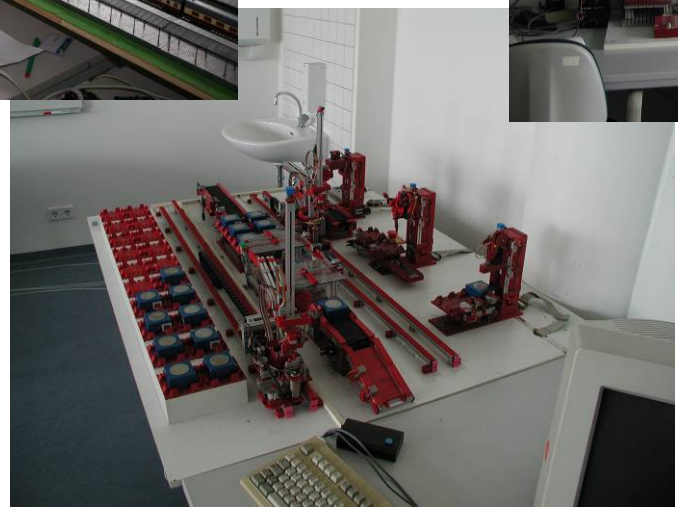




Einleitung Echtzeitsysteme

Anwendungen am Lehrstuhl

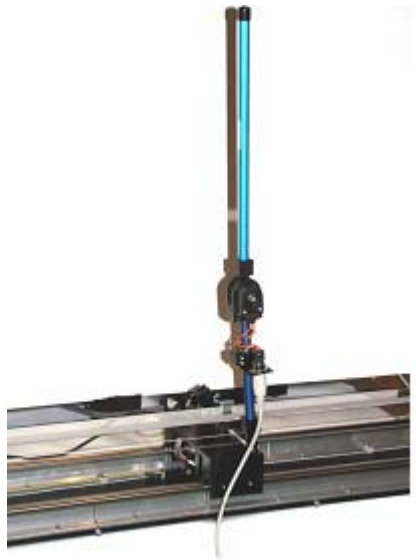
Steuerungsaufgaben (Praktika+Studienarbeiten)



Regelungsaufgaben (Praktika+Studienarbeiten)

Schwebender Stab

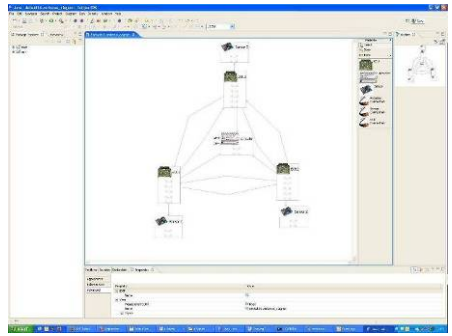
Carrera Rennbahn

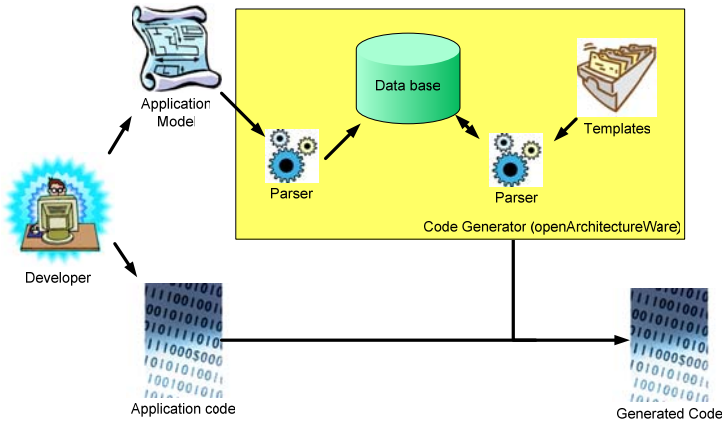


Invertiertes Pendel

Modellbasierte Entwicklung fehlertoleranter Echtzeitsysteme

- Ziele:
  - Umfangreiche Generierung von Code auf Systemebene:
    - Fehlertoleranzmechanismen
    - Prozessmanagement, Scheduling
    - Kommunikation
  - Erweiterbarkeit der Codegenerierung durch Verwendung eines vorlagen-basierten Codegenerators
  - Zertifizierung des Codegenerators

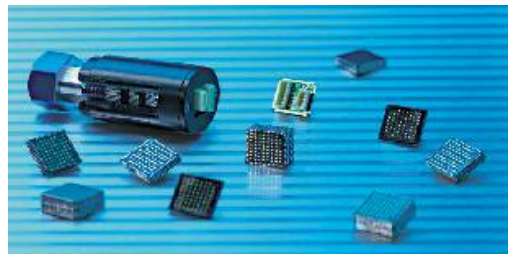


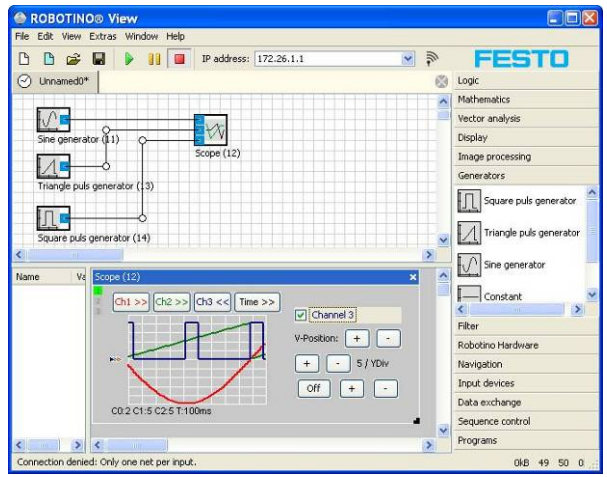


GEFÖRDERT VOM
Bundesministerium für Bildung und Forschung

Modell-basierte Software-Entwicklung für mechatronische Systeme

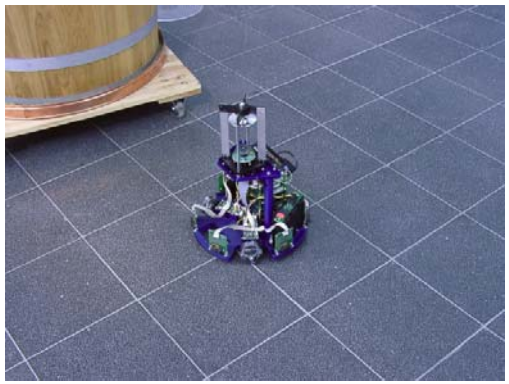
- Entwicklung von komponentenbasierten Architekturen für mechatronische Systeme (Mechanik, Elektronik, Software)
- Ziele:
  - Reduzierung der Entwicklungszeiten
  - Vereinfachung des Entwicklungsprozesses
- Komponenten:
  - Hardwaremodule
  - Softwaremodule
  - Werkzeugkette:
    - Codegenerierung
    - Graphische Benutzerschnittstelle
    - Debugging-Werkzeug



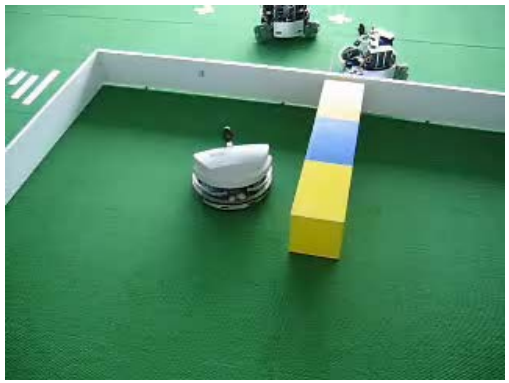


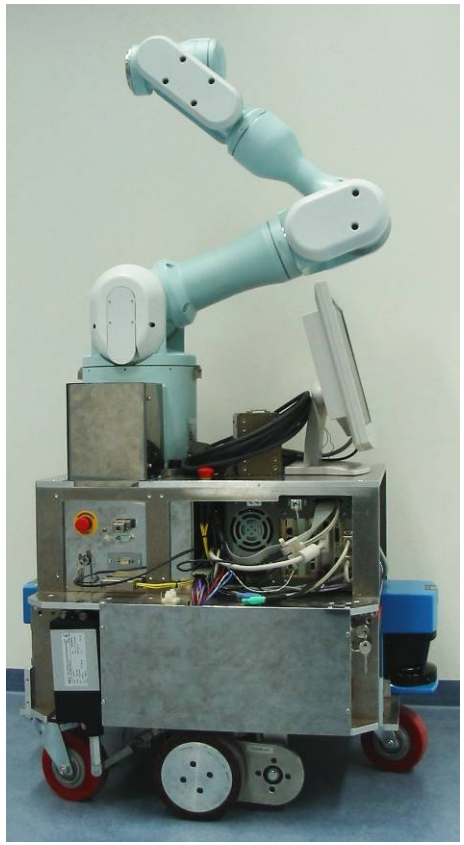
GEFÖRDERT VOM
Bundesministerium für Bildung und Forschung

Robotersteuerung



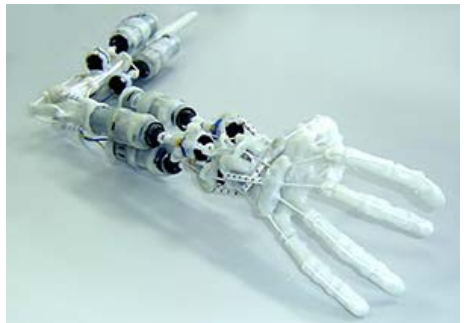
Robotino





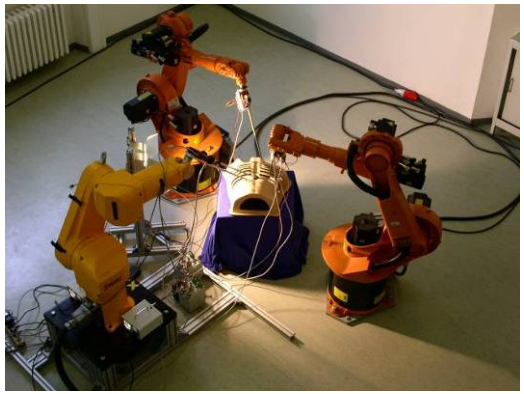
Leonardo

Stäubli



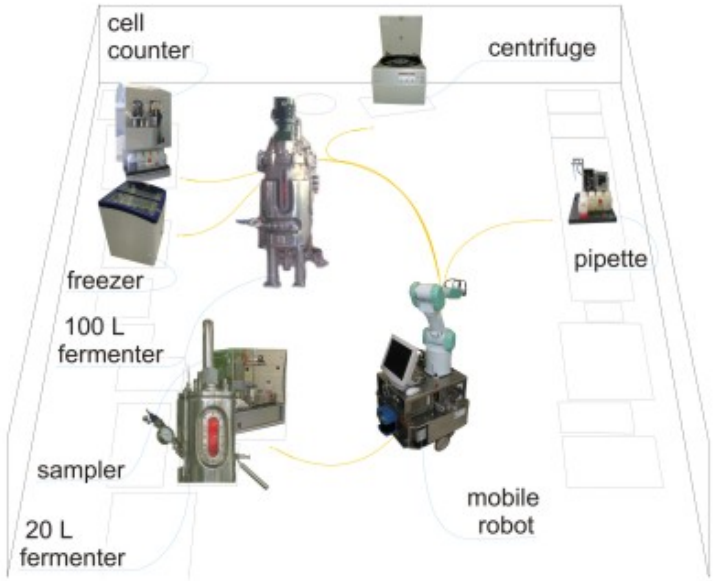
Tumanoid

Anwendungen der Robotik



Telemedizin

Jast



Automatisiertes biotechnisches Labor

Kapitel 2

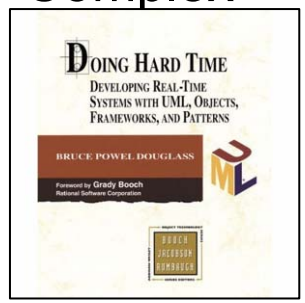
Modellierung von Echtzeitsystemen und Werkzeuge

Inhalt

- Motivation
- Synchrone Sprachen (Esterel, Lustre)
- Werkzeuge Esterel Studio, SCADE
- Real-Time UML
- Zerberus
- Verifikation durch den Einsatz formaler Methoden

Literatur

- Gérard Berry, Georges Gonthier, The Esterel Synchronous Programming Language: Design, Semantics, Implementation, 1992
- http://www-sop.inria.fr/meije/esterel/esterel-eng.html
- N. Halbwachs: A tutorial of Lustre, 1993, http://www-verimag.imag.fr/~halbwach/lustre-tutorial.html
- Diverse Texte zu Esterel, Lustre, Safe State Machines: http://www.esterel-technologies.com/technology/scientific-papers/
- David Harrel, Statecharts: A Visual Formalism For Complex Systems, 1987
- http://www.uml.org
- Bruce Powell Douglass, Doing Hard Time, 2001



Begriff: Modell

- Brockhaus:
  Ein Abbild der Natur unter der Hervorhebung für wesentlich erachteter Eigenschaften und Außerachtlassen als nebensächlich angesehener Aspekte. Ein M. in diesem Sinne ist ein Mittel zur Beschreibung der erfahrenen Realität, zur Bildung von Begriffen der Wirklichkeit und Grundlage von Voraussagen über künftiges Verhalten des erfassten Erfahrungsbereichs. Es ist um so realistischer oder wirklichkeitsnäher, je konsistenter es den von ihm umfassten Erfahrungsbereich zu deuten gestattet und je genauer seine Vorhersagen zutreffen; es ist um so mächtiger, je größer der von ihm beschriebene Erfahrungsbereich ist.

Modellbasierte Entwicklung

- Für die modellbasierte Entwicklung sprechen diverse Gründe:
  - Modelle sind häufig einfacher zu verstehen als der Programmcode (graphische Darstellung, Erhöhung des Abstraktionslevels)
  - Vorwissen ist zum Verständnis der Modelle häufig nicht notwendig:
    - Experten unterschiedlicher Disziplinen können sich verständigen
  - Es existieren Werkzeuge um Code automatisch aus Modellen zu generieren:
    - Programmierung wird stark erleichtert
    - Ziel: umfassende Codegenerierung (Entwicklung konzentriert sich ausschließlich auf Modelle)
  - Mittels formaler Methoden kann
    - die Umsetzung der Modell in Code getestet werden
    - das Modell auf gewisse Eigenschaften hin überprüft werden

Modellbasierte Entwicklung von Echtzeitsystemen

- Modellbasierte Ansätze in Echtzeitsysteme unterliegen anderen Anforderungen als bei „normalen“ Systemen.
- Die Modelle müssen:
  - die zeitliche Ausführung,
  - die Hardwarearchitektur
  - und die Interaktion zwischen Software und Hardware

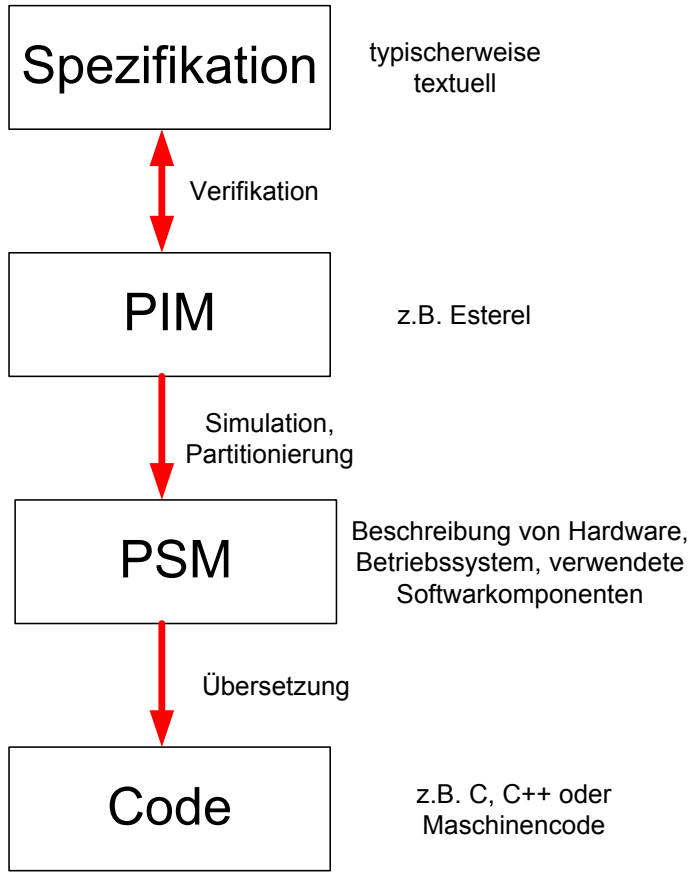
  spezifizieren.
- Andere Ansätze gehen von einer Abstraktion der Hardware aus: die logische Ausführung der Anwendung wird spezifiziert, die Kompatibilität mit der Plattform wird bei der Übersetzungszeit sichergestellt. Die Umsetzung erfolgt zumeist durch ein Laufzeitsystem oder eine virtuelle Maschine.

Probleme mit Zeit und Daten

Software Zeit: Programmiersprachen unterstützen häufig keine Zeit Daten: sind digital
Digitale synchrone Hardware Zeit: typischerweise zyklische Abarbeitung Daten: sind digital
Digitale asynchrone Hardware Zeit: physikalische Zeit Daten: sind digital
Umgebung Zeit: physikalische Zeit Daten: analog/kontinuierlich

Modellbasierte Entwicklung

- Die Entwicklung des Systems erfolgt in diversen Schritten:
  - textuelle Spezifikation
  - PIM: platform independent model
  - PSM: platform specific model
  - Code: Maschinencode bzw. Quellcode
- Aus der Spezifikation erstellt der Entwickler das plattformunabhängige Modell
- Hoffung: weitgehende Automatisierung der Transformationen PIM → PSM → Code (Entwickler muss nur noch notwendige Informationen in Bezug auf die Plattform geben)



Vorteile des modellbasierten Ansatzes

- Beschleunigung des Entwicklungsprozesses
- Umfassende Modellierung, u.a. auch der Umgebung und der analogen und mechanischen Komponenten des Systems
- Frühe Kostenabschätzung
- Möglichkeit zur Simulation verschiedenster Designvarianten
- Möglichkeit zur formale und umfassenden Verifikation in der frühen Projektphase ⇒ frühes Entdecken von Fehlern
- Erleichterung der Umsetzung durch Codegenerierung in Bezug auf die Herausforderungen (verteilte Systeme,…) und typischerweise erhöhte Qualität des Codes
- Garantie zur Erfüllung der Echtzeitbedingungen
- **Problem**: derzeit existieren noch keine umfassenden Werkzeuge, häufig werden nur Teilaspekte der Systeme (z.B. Anwendungslogik, siehe SCADE) unterstützt, die Codegenerierungsmöglichkeiten sind eingeschränkt.

Laufendes Beispiel: Aufzugssteuerung

- Zur Illustration diverser Konzepte werden wir in der Vorlesung / Übung mehrere Beispiele verwenden.
- Beispiel 1: Aufzugssteuerung
  - Verteiltes System:
    - Steuerungsrechner (einfach oder redundant ausgelegt)
    - Microcontroller zur Steuerung der Sensorik / Aktorik
    - CAN-Bus zur Kommunikation
  - Inwiefern ist der Aufzug ein Echtzeitsystem?

Modellierung von Echtzeitsystemen

Synchrone Sprachen: Esterel, Lustre

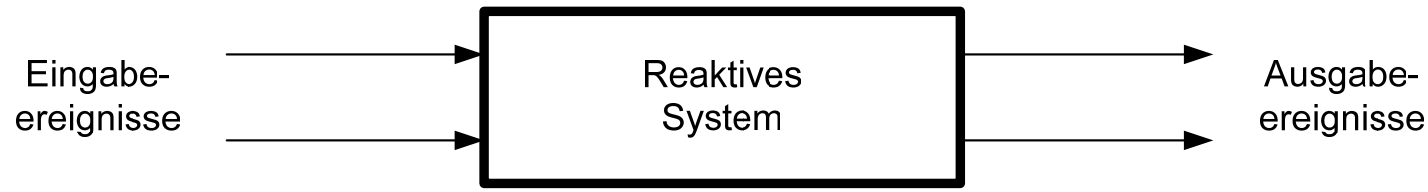
Werkzeuge: Esterel Studio, SCADE

Esterel

- Esterel ist im klassischen Sinne eher eine Programmiersprache, als eine Modellierungssprache
- Esterel wurde von Jean-Paul Marmorat und Jean-Paul Rigault entwickelt um die Anforderungen von Echtzeitsystemen gezielt zu unterstützen:
  - direkte Möglichkeit zum Umgang mit Zeit
  - Parallelismus direkt in der Programmiersprache
- G. Berry entwickelt die formale Semantik für Esterel
- Es existieren Codegeneratoren zur Generierung von u.a. **sequentiellen** C, C++ Code:
  - Aus Esterel-Programmen mit parallelen Berechnungen wird ein Programm mit einem Berechnungsstrang erzeugt ⇒ deterministische Ausführung
  - Technik basiert aus der Erstellung eines endlichen Automaten.
- In der Übung setzen wir die kommerziellen Werkzeuge Esterel Studio / SCADE von Esterel Technology (www.esterel-technologies.com) zum Erlernen von Esterel / Lustre ein.
- SCADE wurde unter anderem zur Entwicklung des Airbus A380 eingesetzt.
- Ein Esterel-Compiler kann unter http://www-sop.inria.fr/meije/esterel/esterel-eng.html umsonst heruntergeladen werden.

Einführung in Esterel

- Esterel beschreibt reaktive Systemen, das System reagiert auf Eingabeereignisse
- Esterel gehört zu der Familie der synchronen Sprachen, weitere Vertreter: Lustre, Signal, Statecharts
- Synchrone Sprachen zeichnen sich vor allem dadurch aus, dass
  - Interaktionen (Reaktionen) des Systems mit der Umgebung die Basisschritte des Systems darstellen (**reaktives System**).
  - Anstelle von physikalischer Zeit logische Zeit (die Anzahl der Interaktionen) verwendet wird.
  - Interaktionen, oft auch **macro steps** genannt, bestehen aus Einzelschritten (micro steps).



Synchrony hypothesis

- Die Synchronitätshypothese (**synchrony hypothesis**) nimmt an, dass die zugrunde liegende physikalische Maschine des Systems unendlich schnell ist.

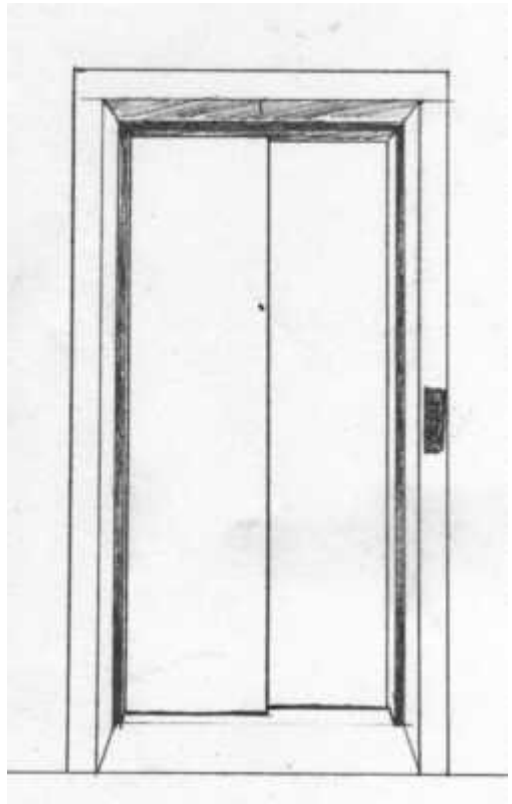
  → Die Reaktion des Systems auf ein Eingabeereignis erfolgt augenblicklich. Reaktionsintervale reduzieren sich zu Reaktionsmomenten (**reaction instants**).
- **Rechtfertigung**: Diese Annahme ist korrekt, wenn die Wahrscheinlichkeit des Eintreffens eines zweiten Ereignisses, während der initialen Reaktion auf das vorangegangene Ereignis, sehr klein ist.
- Esterel erlaubt das gleichzeitige Auftreten von mehreren Eingabeereignissen. Der Reaktionsmoment ist in Esterel dann komplettiert, wenn das System auf alle Ereignisse reagiert hat.

Determinismus

- Esterel setzt den Determinismus der Anwendung voraus: auf eine Sequenz von Ereignissen (auch gleichzeitigen) muss immer dieselbe Sequenz von Ausgabeereignissen folgen.
- Alle Esterel Anweisungen und Konstrukte sind garantiert deterministisch. Die Forderung nach Determinismus wird durch den Esterel Compiler überprüft.

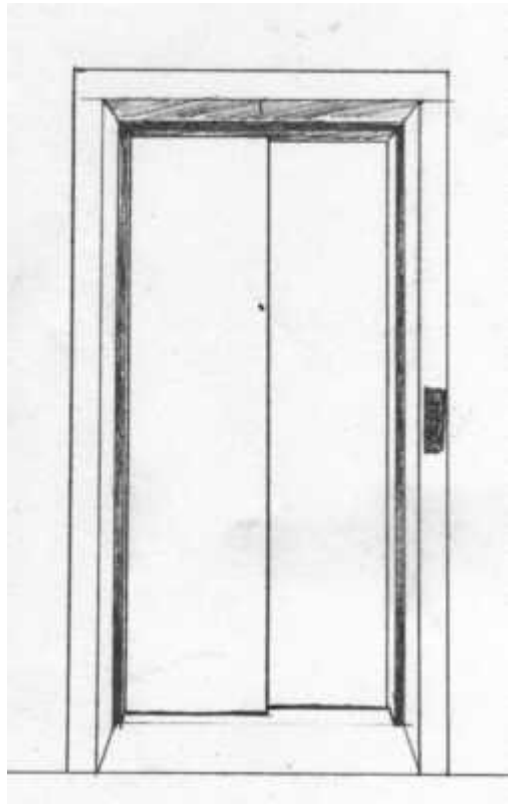
Beispiel: Aufzugstür (vereinfacht)

- Aufgabe: Öffnen und Schließen der Aufzugstür
- Sicherheitsfunktion:
  - Tür darf während der Fahrt nicht geöffnet werden
- 1. Schritt: Definition der Eingangssignale:
  - OPEN (Tür soll geöffnet werden)
  - CLOSE (Tür soll geschlossen werden)
  - GO (Kabine fährt los)
  - REACHED (Kabine erreicht neues Stockwerk)
- 2. Schritt: Definition der Ausgangssignale:
  - DOOR_CLOSE
  - DOOR_OPEN
- 3. Schritt: Definition des Anfangszustandes:
  - Tür offen, Aufzug hält
- 4. Schritt: Definition der Zustandsübergänge



Beispiel: Aufzugstür (vereinfacht)

```
module Door:
input OPEN, CLOSE, GO, REACHED;
output DOOR_OPEN, DOOR_CLOSE;
    loop
        await
          case OPEN do emit DOOR_OPEN;
          case CLOSE do emit DOOR_CLOSE;
          case GO do await REACHED;
        end await;
    end loop;
end module;
```



Module

- **Module** definieren in Esterel (wieder verwendbaren) Code. Module haben ähnlich wie Subroutinen ihre eigenen Daten und ihr eigenes Verhalten.
- Allerdings werden Module nicht aufgerufen, vielmehr findet eine Ersetzung des Aufrufs durch den Modulcode zur Übersetzungszeit statt.
- Globale Daten werden nicht unterstützt. Ebenso sind rekursive Moduldefinitionen nicht erlaubt.
- Syntax:

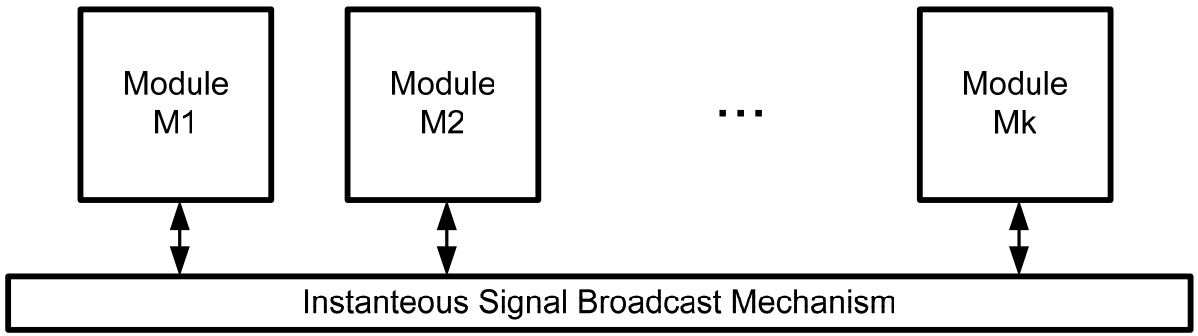
```
%this is a line comment
module module-name:
declarations and compiler directives
%signals, local variables etc.
body
end module % end of module body
```

Signale

- Zur Modellierung der Kommunikation zwischen Komponenten (Modulen) werden Signale eingeführt. Signale sind eine logische Einheit zum Informationsaustausch und zur Interaktion.
- **Deklaration**: Die Deklaration eines Signals erfolgt am Beginn des Moduls. Der Signalname wird dabei typischerweise in Großbuchstaben geschrieben. Zudem muss der Signaltyp festgelegt werden.
- Esterel stellt verschiedene Signale zur Verfügung. Die Klassifikation erfolgt nach:
  - Sichtbarkeit: Schnittstellen (interface) Signale vs. lokale Signale
  - enthaltener Information: pure Signale vs. wertbehaftete Signale (typisiert)
  - Zugreifbarkeit der Schnittstellensignale: Eingabe (input), Ausgabe(output), Ein- und Ausgabe (inputoutput), Sensor (Signal, dass immer verfügbar ist und das nur über den Wert zugreifbar ist)

Signal Broadcast Mechanismus

- **Versand**: Der Versand von Signalen durch die `emit` Anweisung (terminiert sofort) erfolgt über einen Broadcast Mechanismus. Signale sind immer sofort für alle anderen Module verfügbar. Die `sustain` Anweisung erzeugt in jeder Runde das entsprechende Signal und terminiert nicht. Es muss mit Hilfe von `abort` abgebrochen werden.
- **Verfügbarkeit**: Signale sind nur für den bestimmten Moment verfügbar.
- Nach Ende des aktuellen Moments wird der Bus zurückgesetzt.
- **Zugriff**: Prozesse können per `await` auf Signale warten oder prüfen, ob ein Signal momentan vorhanden ist (`if`). Auf den Wert eines wertbehaftete Signale kann mittels des Zugriffsoperator ? zugegriffen werden.



await Anweisung

```
await
    case Occurence-1 do Body-1
    case Occurence-2 do Body-2
    ...
    case Occurence-n do Body-n
end await;
```

- Mit Hilfe dieser Anweisung wird auf das Eintreten einer Bedingung gewartet. Im Falle eines Auftretens wird der assoziierte Code gestartet. Werden in einem Moment mehrere Bedingungen wahr, entscheidet die textuelle Reihenfolge. So kann eine deterministische Ausführung garantiert werden.

Ereignisse (Events)

- **Ereignisse** setzen sich zu einem bestimmten Zeitpunkt (**instant**) aus den Eingabesignalen aus der Umwelt und den Signalen, die durch das System als Reaktion ausgesandt werden, zusammen.
- Esterel Programme können nicht direkt auf das ehemalige oder zukünftige Auftreten von Signalen zurückgreifen. Auch kann nicht auf einen ehemaligen oder zukünftigen Moment zugegriffen werden.
- Einzige Ausnahme ist der Zugriff auf den letzten Moment. Durch den Operator `pre` kann das Auftreten in der vorherigen Runde überprüft werden.

Zeitdauer

- Die Zeitachse wird in Esterel in diskrete Momente (instants) aufgeteilt. Über die Granularität wird dabei in Esterel keine Aussage getroffen.
- Zur deterministischen Vorhersage des zeitlichen Ablaufes von Programmen wird jede Anweisung in Esterel mit einer genauen Definition der Ausführungszeitdauer verknüpft.
- So terminiert beispielsweise `emit` sofort, während `await` so viel Zeit benötigt, bis das assoziierte Signal verfügbar ist.

Unendliche Schleife (infinite loop)

```
loop Body end loop;
```

- Mit Hilfe dieser Anweisung wird ein Stück Code Body endlos ausgeführt. Sobald eine Ausführung des Codes beendet wird, wird der Code wieder neu gestartet.
- **Bedingung**: die Ausführung des Codes darf nicht im gleichen Moment, indem sie gestartet wurde, terminieren.

Beziehungen (relations)

- Der Esterel-Compiler erzeugt aus der Esterel-Datei einen endlichen Automaten. Hierzu müssen für jeden Zustand (Block) sämtliche Signalkombinationen getestet werden.
- Um bei der automatischen Generierung des endlichen Automaten des Systems die Größe zu reduzieren, können über die `relation` Anweisung Einschränkungen in Bezug auf die Signale spezifiziert werden:
- `relation Master-signal-name => Slave-signal-name;`

  Bei jedem Auftreten des Mastersignals muss auch das Slave-Signal verfügbar sein.
- `relation Signal-name-1 # Signal-name-2 # ... # Signal-name-n;`

  In jedem Moment darf maximal eines der spezifizierten Signale Signal-name-1, Signal-name-2 ,..., Signal-name-n präsent sein.
- Viel einfacher bzw. anschaulicher und fehlersicherer ist aber die direkte Modellierung des Programms als endlicher Automat.

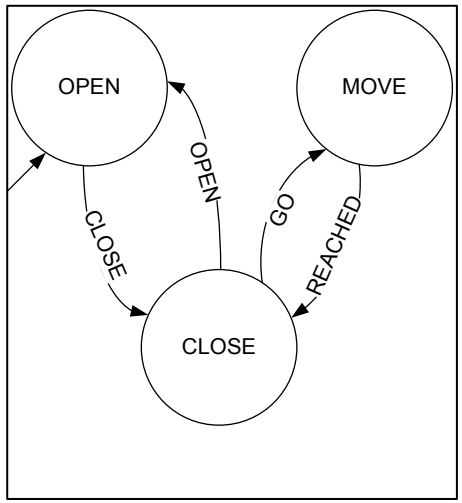
Exkurs: Automaten zur Modellierung von reaktiven Systemen

Automaten im Kontext von reaktiven Systemen

- Durch graphische Darstellung (z.B. Automaten) kann die Verständlichkeit des Codes stark verbessert werden.
- Ein reaktives System kann durch die zyklische Ausführung folgender Schritte beschrieben werden:
  1. Lesen der Eingangssignale
  2. Berechnung der Reaktionen
  3. Auslösen der Ausgangssignale
- Die zyklische Ausführung kann im Automatenmodell wie folgt interpretiert werden:
  1. Lesen der Eingabe im aktuellen Zustand
  2. Berechnen der Zustandsübergangsfunktion und ggfs. Zustandswechsel
  3. Erzeugung von Ausgangssignalen (abhängig von alten Zustand und gelesenem Eingangssignal)
- Die kennen gelernte Synchronitätshypothese bedeutet im Bezug auf den Automaten der im Vergleich zur Zeit für Änderungen der Umgebung eine vernachlässigbare Zeit für die Berechnung der Zustandsübergange und Ausgabefunktion.
- Pro Runde wird im Allgemeinen (1 Ausnahme) genau ein Zustandsübergang berechnet.

Endliche Automaten

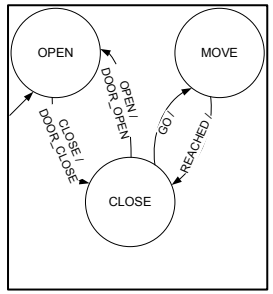
- Ein klassischer endlicher Automat (Q, $\sum$, δ, s_0, F) ist eine endliche Menge von Zuständen und Zustandsübergängen mit:
  - endlicher Menge von Zuständen Q
  - endliches Eingabealphabet $\sum$
  - Übergangsfunktion δ : Q x $\sum \rightarrow$ Q
  - Menge von Endzuständen F $\subseteq$ Q



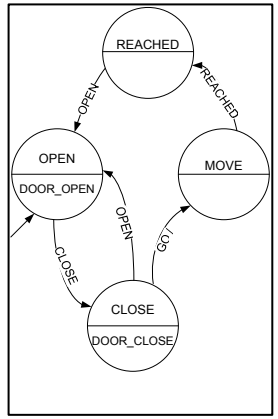
- Um die Verständlichkeit zu verbessern, werden nur deterministische Automaten modelliert, also |S|=1 und $\delta(q,\alpha)=q' \wedge \delta(q,\alpha)=q'' \Rightarrow p'=q''$
- Problem bei der klassischen Definition von Automaten: Ausgaben können nicht modelliert werden.

Automaten mit Ausgaben

- Mealy- und Moore-Automaten unterstützen die Ausgabe von Signalen
- Die Ausgaben von Mealy-Automaten $(Q,\Sigma,\Omega, \delta, \lambda, q_0,F)$ sind dabei an die Übergänge gebunden mit
  - $Q, \Sigma, \delta, q_0, F$ wie bei klassischem Automat
  - Ausgabealphabet Ω
  - Ausgabefunktion λ: $Q \times \Sigma \rightarrow \Omega$



- Bei Moore-Automaten $(Q,\Sigma,\Omega, \delta, \lambda, q_0,F)$ ist die Ausgabe dagegen von den Zuständen abhängig:
  - $Q, \Sigma, \Omega, \delta, q_0, F$ wie bei Mealy-Automat
  - Ausgabefunktion λ: $Q \rightarrow \Omega$
- Moore- und Mealy-Automat sind gleich mächtig: sie können ineinander konvertiert werden.



Harel-Automaten / Statecharts

- Heutiger Standard zur Beschreibung von reaktiven Systemen sind die von David Harel 1987 vorgeschlagenen Statecharts.
- Statecharts zeichnen sich durch folgende Eigenschaften aus:
  - Vereinigung der Eigenschaften von Mealy- und Moore-Automaten
    - Ausgaben in Abhängigkeit von Zustandsübergängen möglich
    - Durchführung von Ausgaben beim Erreichen eines Zustands (**`onExit`**) oder Verlassen eines Zustandes (**`onEntry`**)
  - Zur Erhöhung der Lesbarkeit: Hierarchische Strukturierung von Teilautomaten möglich inkl. Gedächtnisfunktionalität
  - Darstellung paralleler Abläufe durch parallele Teilautomaten.
  - Verknüpfung von Zuständen mit Aktionen: Befehle **`do`** (zeitlich begrenzte Aktivität), **`throughout`** (zeitlich unbegrenzte Aktivität)
  - Einführung spontaner bzw. überwachter (guarded) Übergänge

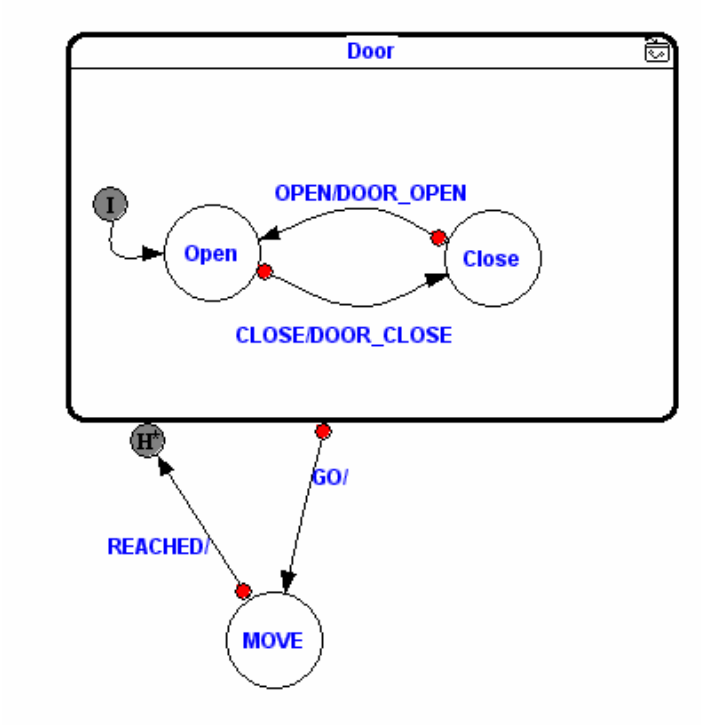
Safe State Machine

- Esterel benutzt eine eigene Klasse von Automaten, die den Statecharts sehr ähnlich sind:
  - Vereinigung der Eigenschaften von Mealy- und Moore-Automaten
    - Ausgaben in Abhängigkeit von Zustandsübergängen möglich
    - Durchführung von Ausgaben beim Erreichen eines Zustands (**`onExit`**) oder Verlassen eines Zustandes (**`onEntry`**)
  - Zur Erhöhung der Lesbarkeit: Hierarchische Strukturierung von Teilautomaten möglich inkl. Gedächtnisfunktionalität
  - Darstellung paralleler Abläufe durch parallele Teilautomaten.
  - Verknüpfung von Zuständen mit Aktionen: Befehle **`do`** (zeitlich begrenzte Aktivität), **`throughout`** (zeitlich unbegrenzte Aktivität)
  - Einführung spontaner bzw. überwachter (guarded) Übergänge
  - Zusätzliche Esterel abhängige Konstrukte (z.B. `pre` Operator)

Modellierung von Echtzeitsystemen

Fortsetzung synchrone Sprachen

Beispiel: Automat in Esterel Studio (leicht verändert)

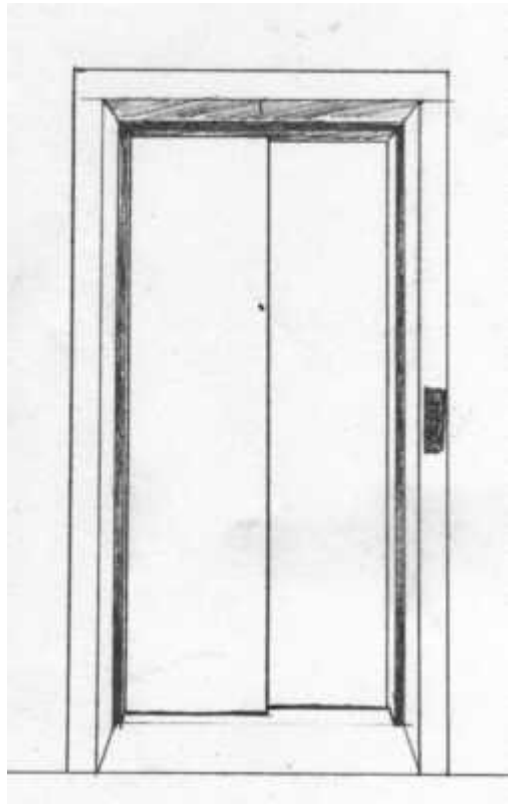


Abort Konstrukt

- Zur einfacheren Modellierung können Abbruchbedingungen nicht nur durch Zustandsübergänge, sondern auch direkt mit Makrozuständen verbunden werden.
- Dabei wird zwischen zwei Arten des Abbruches unterschieden:
  - weak abort: die in der Runde vorhandenen Signale werden noch verarbeitet, danach jedoch der Abbruch vollzogen
  - strong abort: der Abbruch wird sofort vollzogen, eventuell vorhandene Signale ignoriert.
- In der Sprache Esterel wird eine Abbruchbedingung durch das Konstrukt
  `abort Body when Exit_Condition`
  bzw.
  `abort Body when immediate Exit_Condition`
  ausgedrückt.

Beispiel: Aufzugstür mit abort (Beispiel nicht 100% identisch mit Automat)

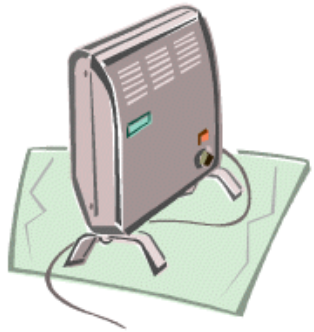
```
module Door_Abort:
input OPEN, CLOSE, GO, REACHED;
output DOOR_OPEN, DOOR_CLOSE;
loop
   abort
        loop
            await
              case OPEN do emit DOOR_OPEN;
              case CLOSE do emit DOOR_CLOSE;
            end await;
        end loop;
    when immediate GO;
    await REACHED;
end loop;
end module;
```

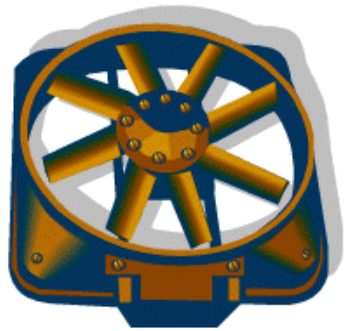


Parallelismus

- Zur parallelen Komposition stellt Esterel den Operator || zur Verfügung. Sind P1 und P2 zwei Esterel Programme, so ist auch P1||P2 ein Esterel Programm mit folgenden Eigenschaften:
  - Alle Eingabeereignisse stehen sowohl P1 als auch P2 zur Verfügung.
  - Jede Ausgabe von P1 (oder P2) ist im gleichen Moment für P2 (oder P1) sichtbar.
  - Sowohl P1 als auch P2 werden parallel ausgeführt und die Anweisung P1||P2 endet erst, wenn beide Programme beendet sind.
  - Es können keine Daten oder Variablen von P1 und P2 gemeinsam genutzt werden.
- Zur graphischen Modellierung stehen parallele Teilautomaten zur Verfügung.

Beispiel: Temperaturregelung im Aufzug





Beschreibung Beispiel

- Ziel: Regelung der Temperatur (Betriebstemperatur 5-40 Grad Celsius)
- Ansatz: Nähert sich die Temperatur einem der Grenzwerte, so wird der Lüfter bzw. die Heizung (Normalstufe) eingeschaltet. Verbleibt der Wert dennoch im Grenzbereich, so wird auf die höchste Stufe geschaltet.
  Ist der Wert wieder im Normalbereich, so wird (zur Vereinfachung) der Lüfter bzw. die Heizung wieder ausgeschaltet.
  Wird die Betriebstemperatur über- bzw. unterschritten, so wird ein Abbruchssignal an den Aufzug geschickt.

Esterel Code 1. Teil

```
module Temperature:
input TEMP: integer, SAMPLE_TIME, DELTA_T;
output HEATER_ON, HEATER_ON_STRONG,
       HEATER_OFF,VENTILATOR_ON, VENTILATOR_OFF,
       VENTILATOR_ON_STRONG, ABORT;
input relation SAMPLE_TIME => TEMP;
    signal COLD, NORMAL, HOT in
        every SAMPLE_TIME do
            await immediate TEMP;
            if
                case ?TEMP<5 or ?TEMP>40 do emit ABORT
                case ?TEMP>=35 do emit HOT
                case ?TEMP<=10 do emit COLD
                default do emit NORMAL
            end if
        end every
        ||
```

Esterel Code 2. Teil

```
        loop
             await
                 case COLD do
                     emit HEATER_ON;
                     abort
                         await NORMAL;
                         emit HEATER_OFF;
                     when DELTA_T do
                         emit HEATER_ON_STRONG;
                         await NORMAL;
                         emit HEATER_OFF;
                     end abort
                 case HOT do
               ...
             end await
         end loop
     end signal
end module
```

Lokale Signale und wertbehaftete Signale

```
signal Signal-decl-1,Signal-decl-2,...,Signal-
decl-n in
    Body
end;
```

- Durch diese Anweisung werden lokale Signale erzeugt, die nur innerhalb des mit Body bezeichneten Code verfügbar sind.

  `Signal-name: Signal-type`

- Der Typ eines wertbehaften Signals kann durch diese Konstruktion spezifiziert werden.

every Anweisung

- Mit Hilfe der `every` Anweisung kann ein periodisches Wiederstarten implementiert werden.
- Syntax:

```
every Occurence do
    Body
end every
```

- Semantik: Jedes Mal falls die Bedingung `Occurence` erfüllt ist, wird der Code Body gestartet. Falls die nächste Bedingung `Occurence` vor der Beendigung der Ausführung von Body auftritt, wird die aktuelle Ausführung sofort beendet und eine neue Ausführung gestartet.
- Es ist auch möglich eine Aktion in jedem Moment zu starten:

```
every Tick do
    Body
end every;
```

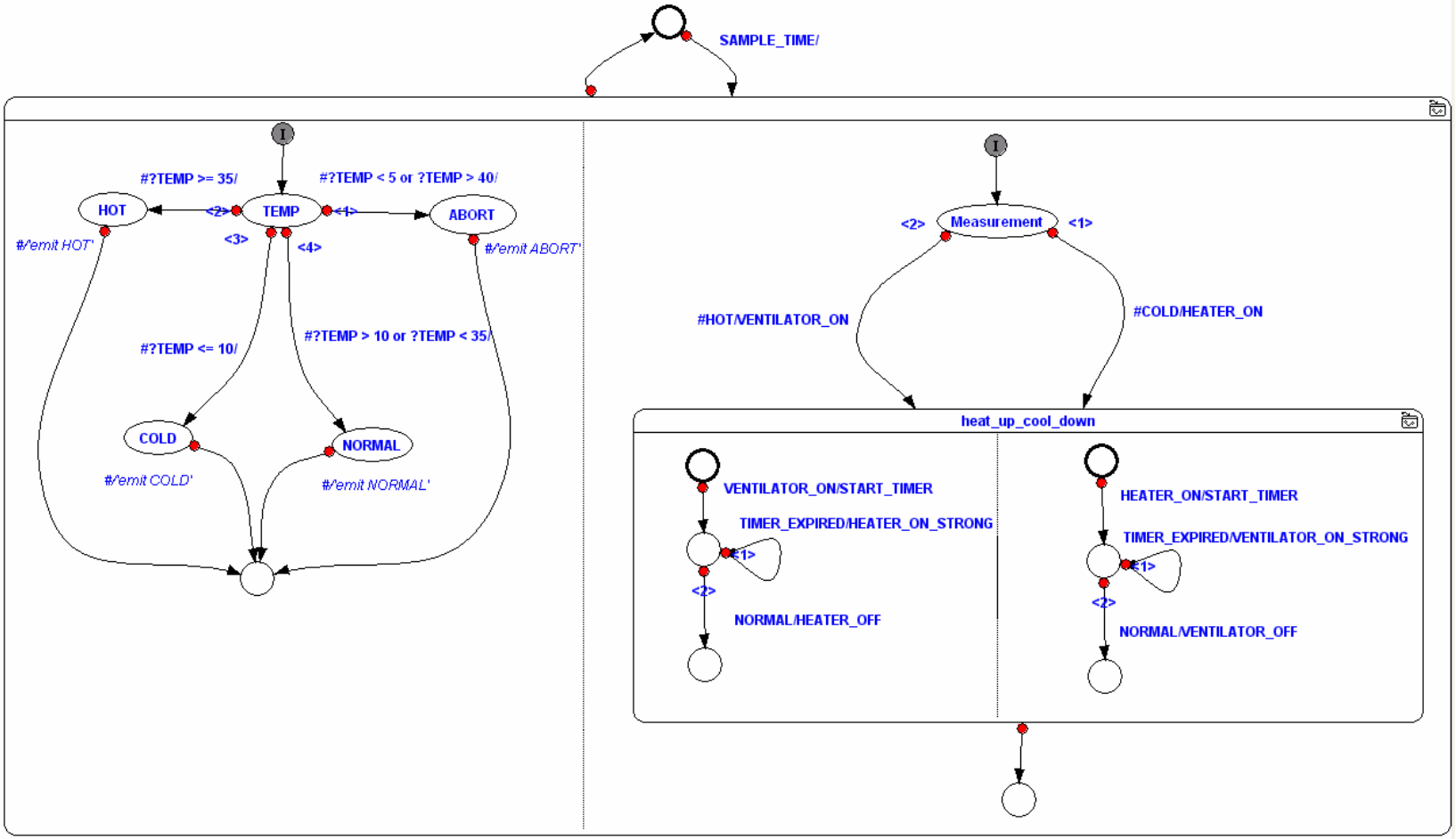
if Anweisung in Bezug auf Signale

- Durch Verwendung der if- Anweisung kann auch die Existenz eines Signals geprüft werden.
- **Syntax**:

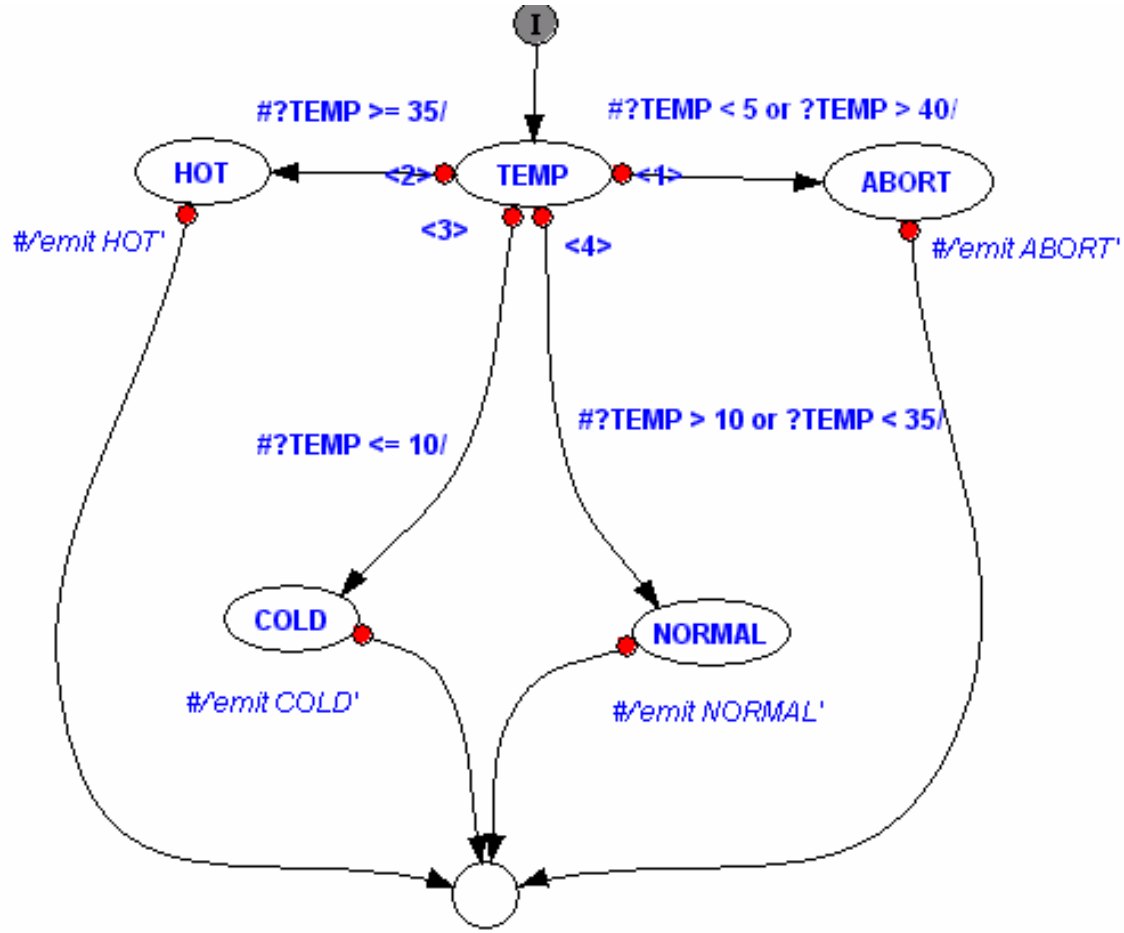
```
if Signal-Name then
    Body-1
else
    Body-2
```

- **Semantik**: Bei Start dieser Anweisung wird geprüft, ob das Signal `Signal-Name` verfügbar ist. Ist es verfügbar, so wird der Code von `Body-1` ausgeführt, anderenfalls von Body-2. Innerhalb der Anweisung `if` kann auch entweder der `then Body-1` oder der `else Body-2` -Teil weggelassen werden.

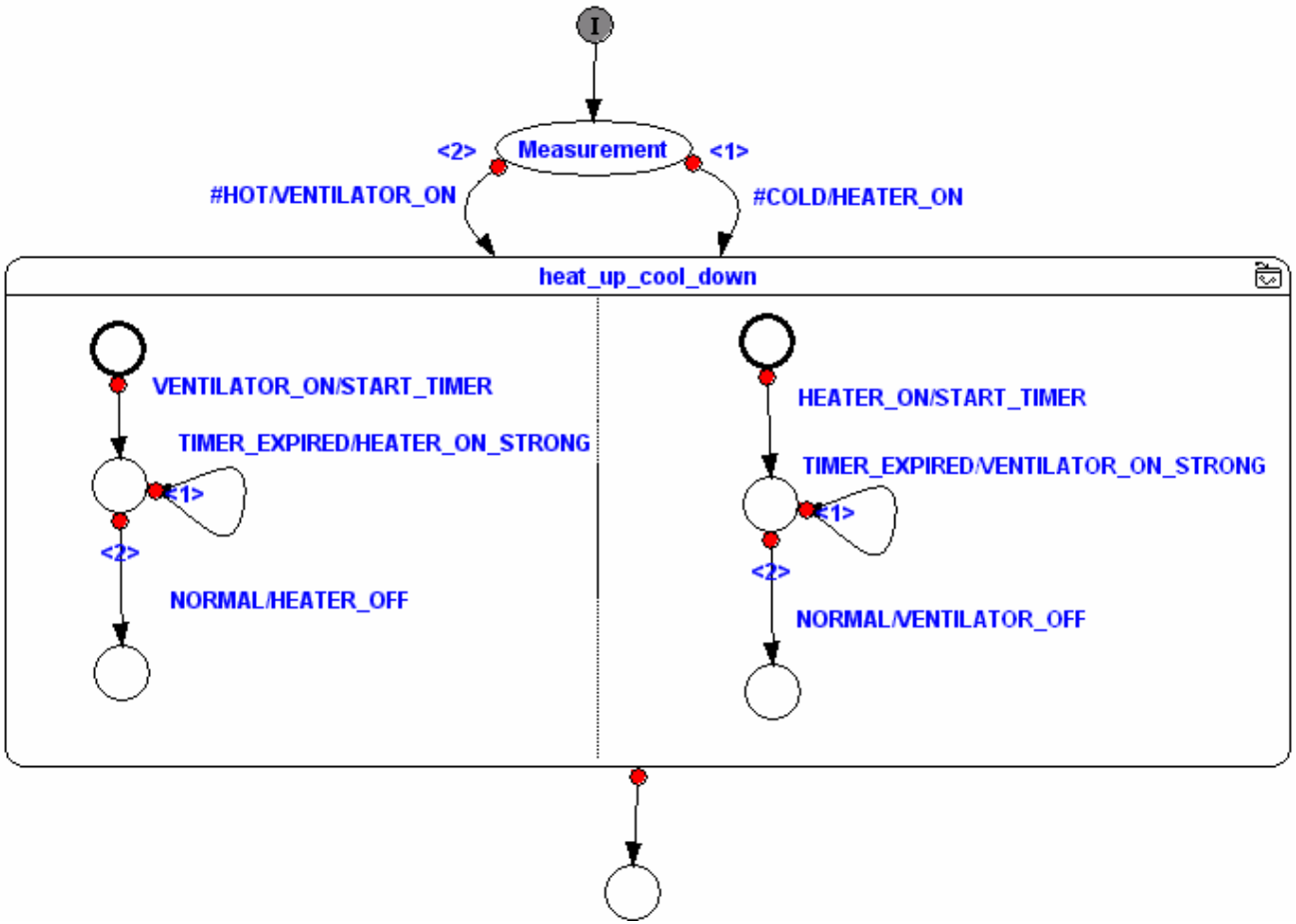
Beispiel als Automat



Beispiel als Automat – Teil 1



Beispiel als Automat – Teil 2



Modellierung von Echtzeitsystemen

Real-Time UML

Wieso UML?

- UML- *Unified Modeling Language*

  Standard in Entwicklungsprozessen von Standardsoftware
- Breite Unterstützung durch Tools

  Hersteller: Artisan (RealTime Studio), Rationale, I-Logix, objectiF
- Intuitiver Ansatz

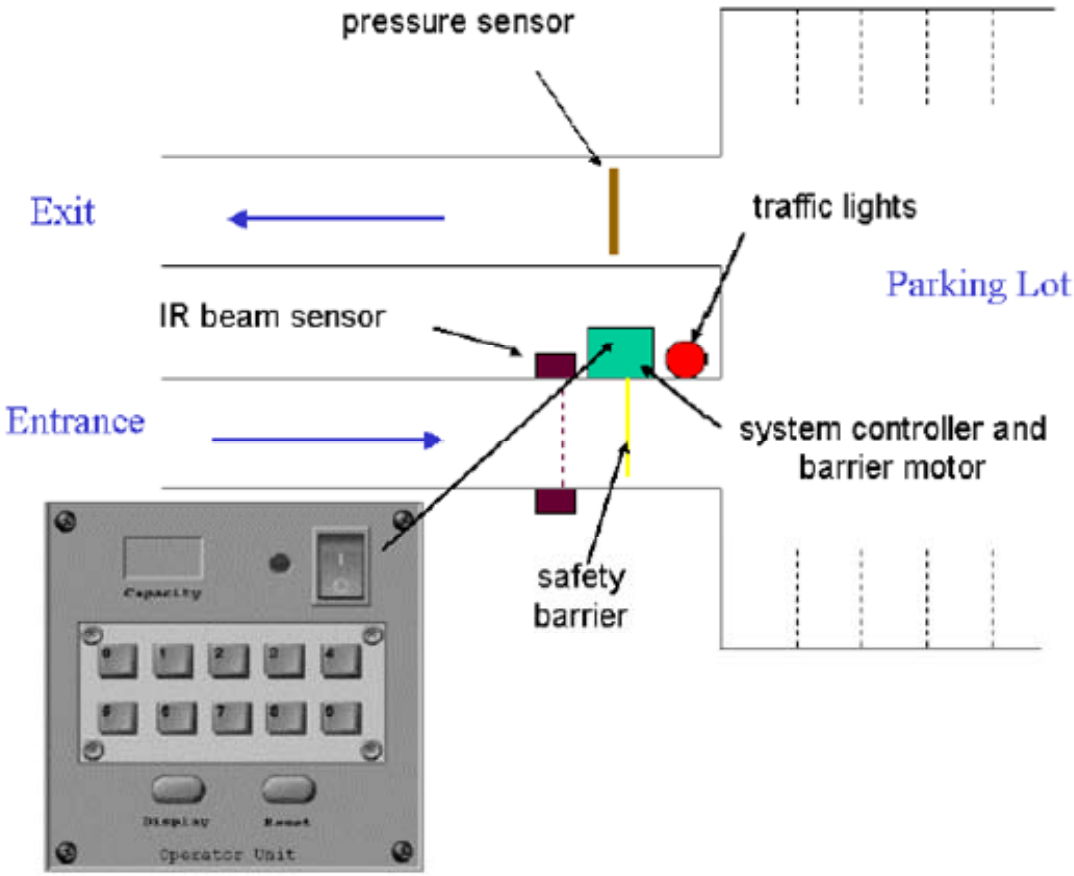
Erweiterungen der Standardmodelle

- UML ist in seiner Grundform nicht für Echtzeitsysteme geeignet.
- Die OMG (object management group) führte als Möglichkeit zur Erweiterung sogenannte Stereotypen (stereotypes) ein.
- Stereotypen erlauben die Erzeugung von neuen Meta-Modellen der UML-Modelle.
- Beispiel: Mit dem Stereotyp <<active>> kann eine Klasse markiert werden, die die Basis für einen eigenständigen Prozess ist

Beispiele: UML-Stereotypen für Echtzeitsysteme

UML Basistyp	Stereotyp	Beschreibung
Class	<<active>>	UML Klasse ist Basis für einen Thread
Message	<<synchronous>>	Assozation wird als einfache Funktion oder Methodenaufruf realisiert
	<<blocking-local>>	Assoziation überschreitet Thread-Grenze, der aufrufende Thread wird bis zur Aufrufrückkehr blockiert
	<<asynchronous-local>>	Assoziation überschreitet Thread-Grenze durch Senden der Nachricht an die Eingangswarteschlange des Zielthreads
	<<waiting-local>>	Sender wartet auf Empfänger mit maximaler Wartezeit
	<<synchronous-remote>>	Assoziation überschreitet Prozessorgrenze und Sender und Empfänger warten jeweils aufeinander
	…	…
…	…	…

Anwendungsbeispiel: Parkanlage



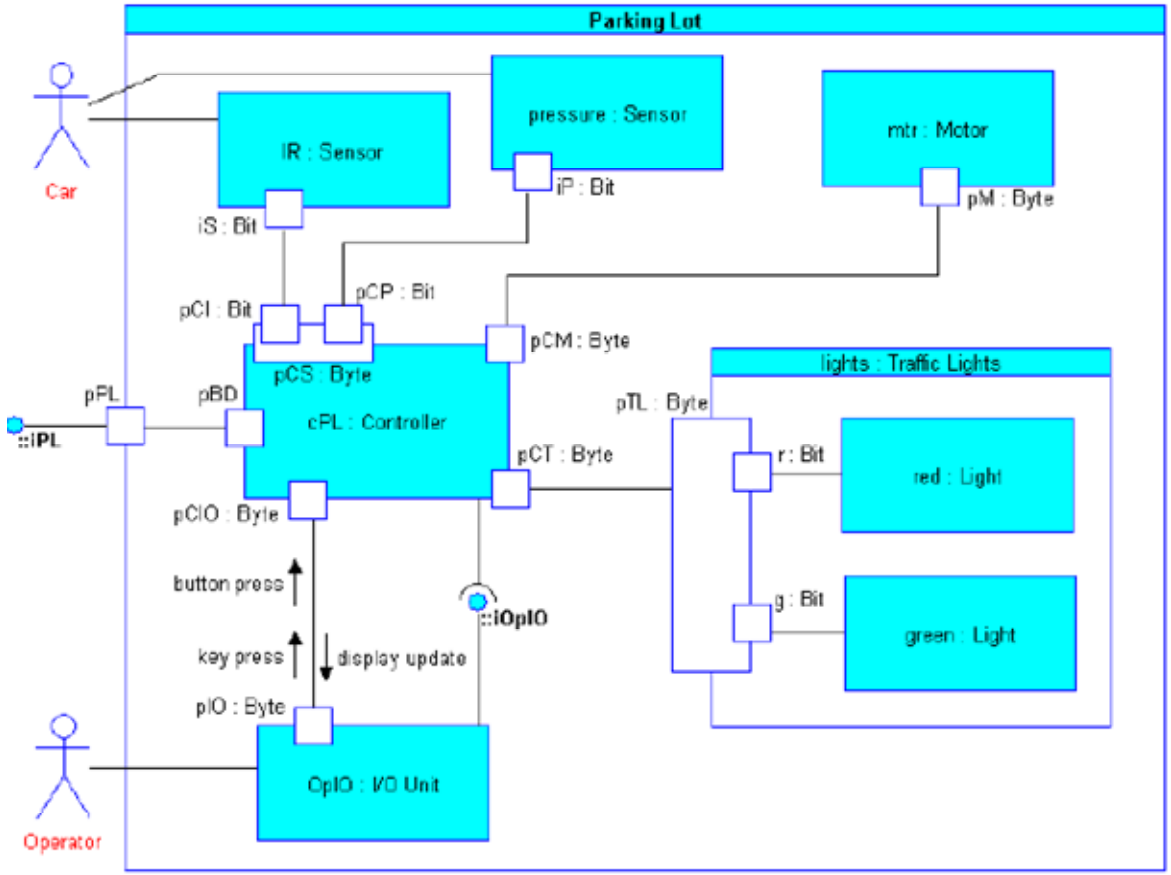
Beschreibung des Beispiels

- Parkanlage regelt die Einfahrt in die Parkgarage
- Freie Parkplätze werden kontrolliert und Autos werden nur in die Parkgarage gelassen, falls noch ein Parkplatz frei ist.
- Anlage verwendet zwei Sensoren: Infrarotsensor am Eingang, Drucksensor am Ausgang.
- Durch eine Schranke werden Autos gehindert in die Parkgarage einzufahren.
- Ein rotes und ein grünes Licht signalisieren dem Fahrer, ob Plätze frei sind.
- Über eine Bedieneinheit kann die Kapazität angepasst werden.

Structure Diagrams (1)

- Ziele:
  - Aufteilung des Systems in seine Subsysteme und Komponenten
  - Identifikation der Kommunikation
  - Identifikation der Speichertypen
  - Identifikation der Komponententypen
  - Spezifikation der Schnittstellen
- Zeitpunkt: Analysephase

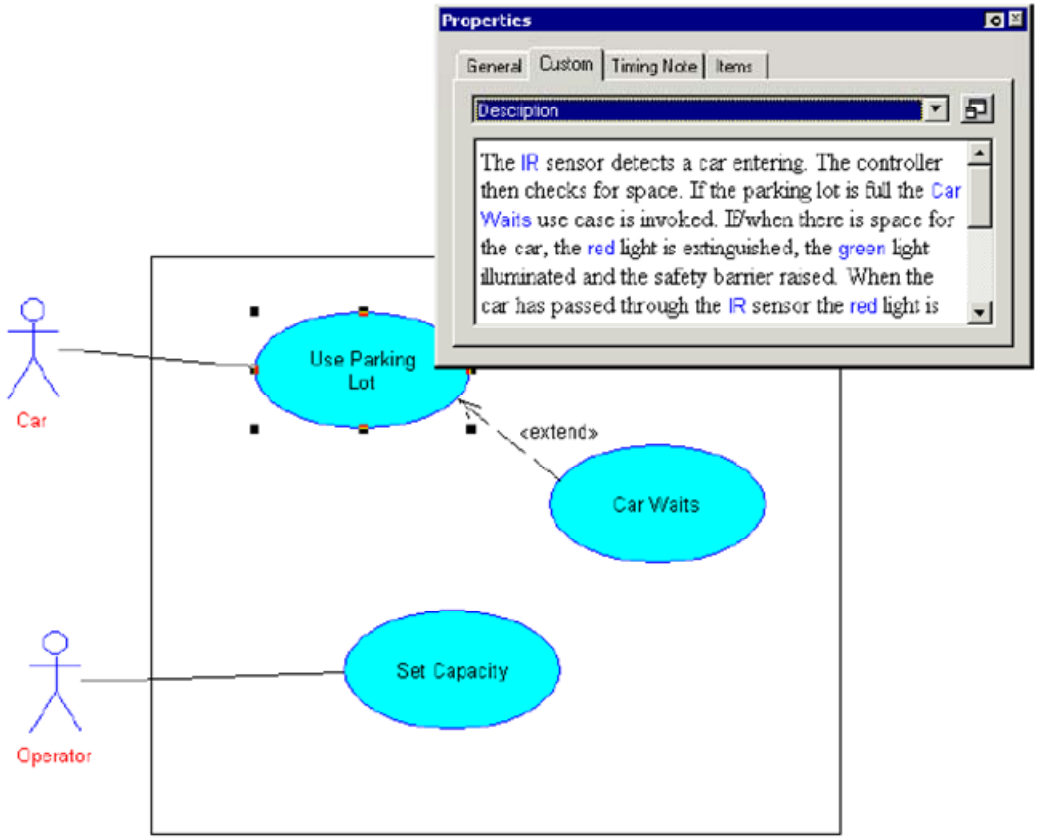
Structure Diagrams (2)



Use Cases (1)

- Ziele:
  - Identifikation der Anwendungsfunktionalität durch textuelle und graphische Beschreibung
- Methoden:
  - Testfälle beschreiben immer nur ein Anwendungsszenario
  - Testfälle beginnen immer mit einer Aktion einer externen Einheit
  - Testfälle beschreiben Funktionalität in textueller Form aus Sicht des externen Benutzers
- Zeitpunkt: Analysephase

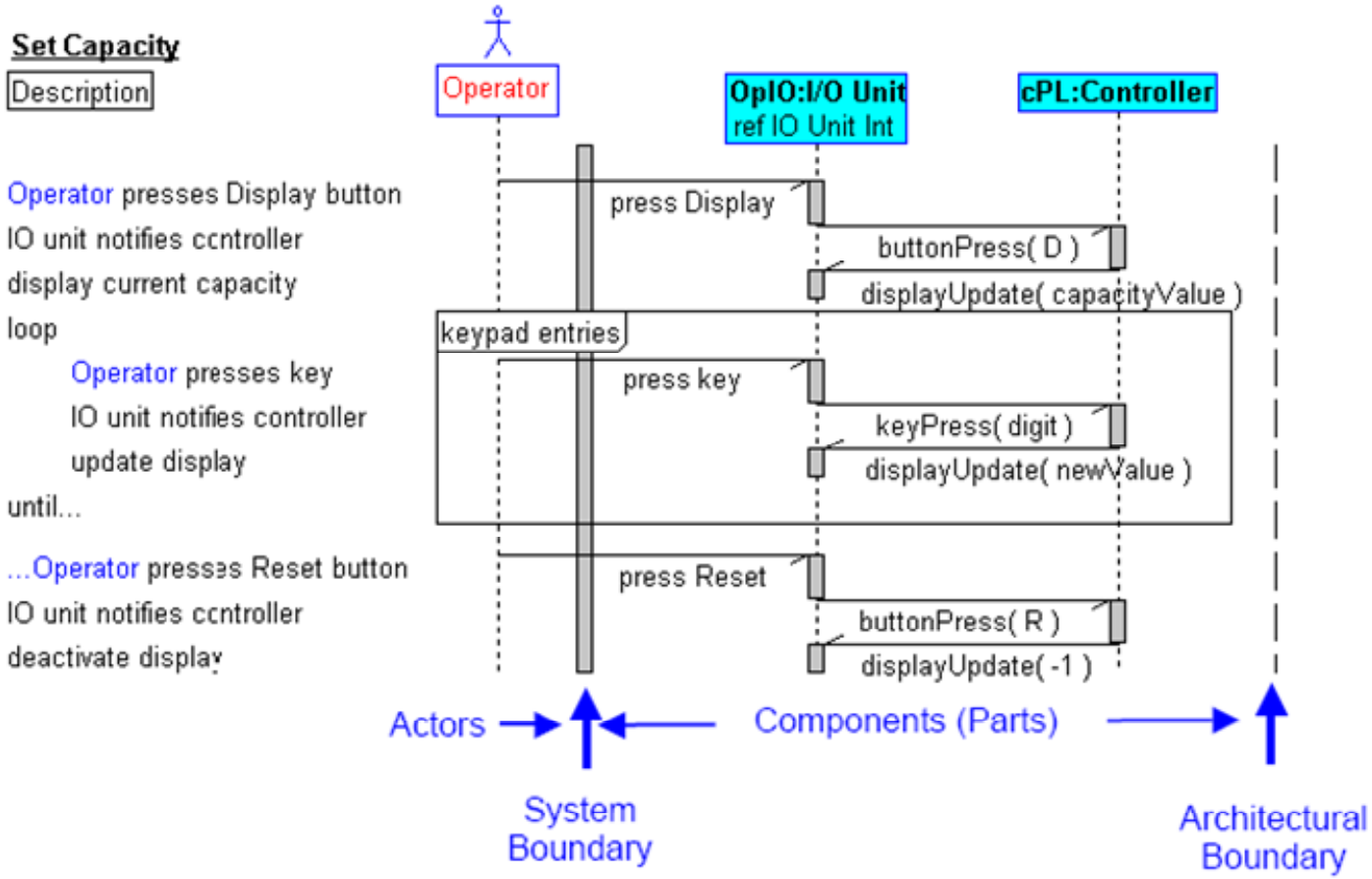
Use Cases (2)


Properties
General
Custom
Timing
Note
Items
Description
The IR sensor detects a car entering. The controller then checks for space. If the parking lot is full the Car Waits use case is invoked. If/when there is space for the car, the red light is extinguished, the green light illuminated and the safety barrier raised. When the car has passed through the IR sensor the red light is
Use Parking Lot
Car
«extend»
Car Waits
Set Capacity
Operator

Sequence Diagram (1)

- Ziele:
  - Analyse der dynamischen Interaktion der einzelnen Komponenten
- Methoden:
  - Definition von Nachrichtensequenzen zwischen einzelnen Komponenten
  - Sequenzdiagramme basieren zumeist auf Use Cases
  - Einzelnen Komponenten werden als Black Boxes betrachtet
  - Klassifikation der Kommunikation (synchron, asynchron)
- Zeitpunkt: Designphase

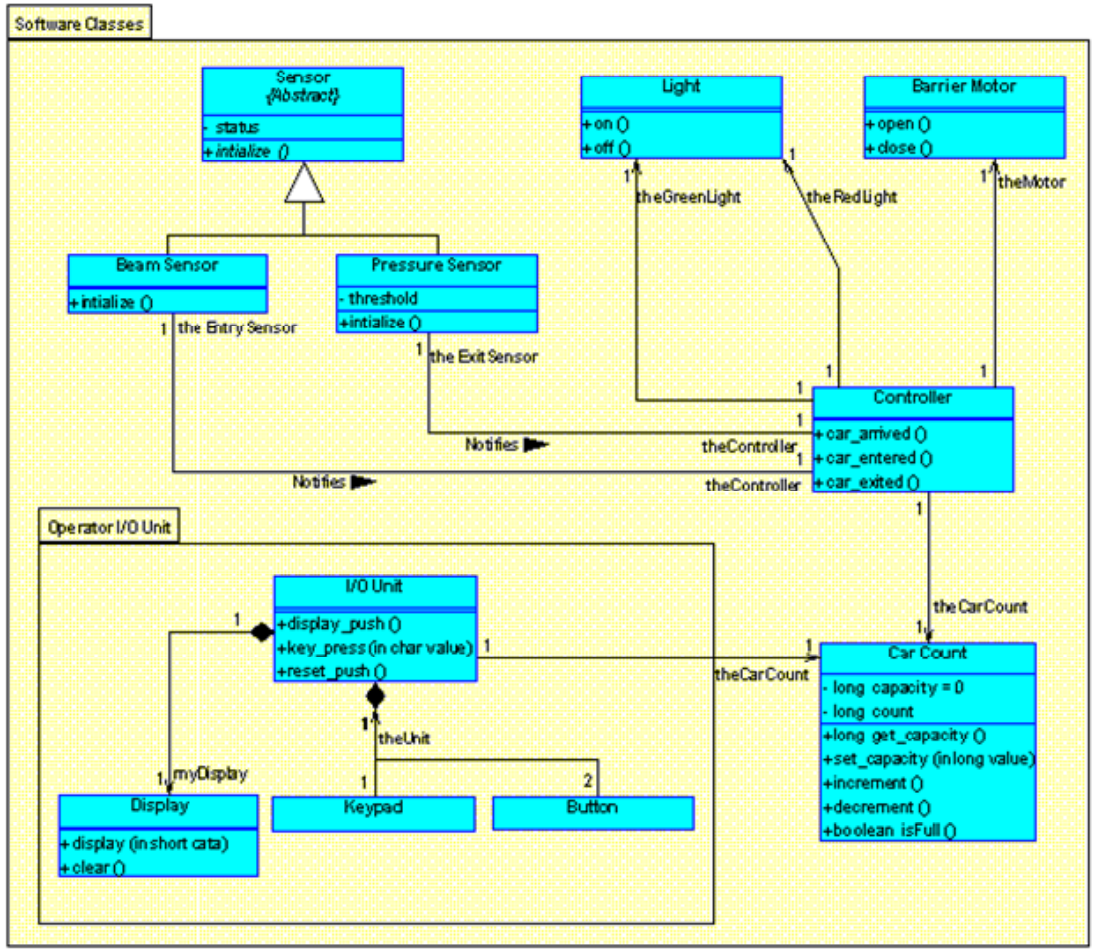
Sequence Diagram (2)



Class Diagram (1)

- Ziele:
  - Identifikation der statischen Struktur der Softwarekomponenten
- Methode:
  - Identifikation der Objekte
  - Festlegung der Attribute der Objekte
  - Festlegung der Methoden
  - Spezifikation der Interaktion von verschiedenen Objekten
  - Spezifikation von Aggregationen von verschiedenen Objekten
- Zeitpunkt: Designphase

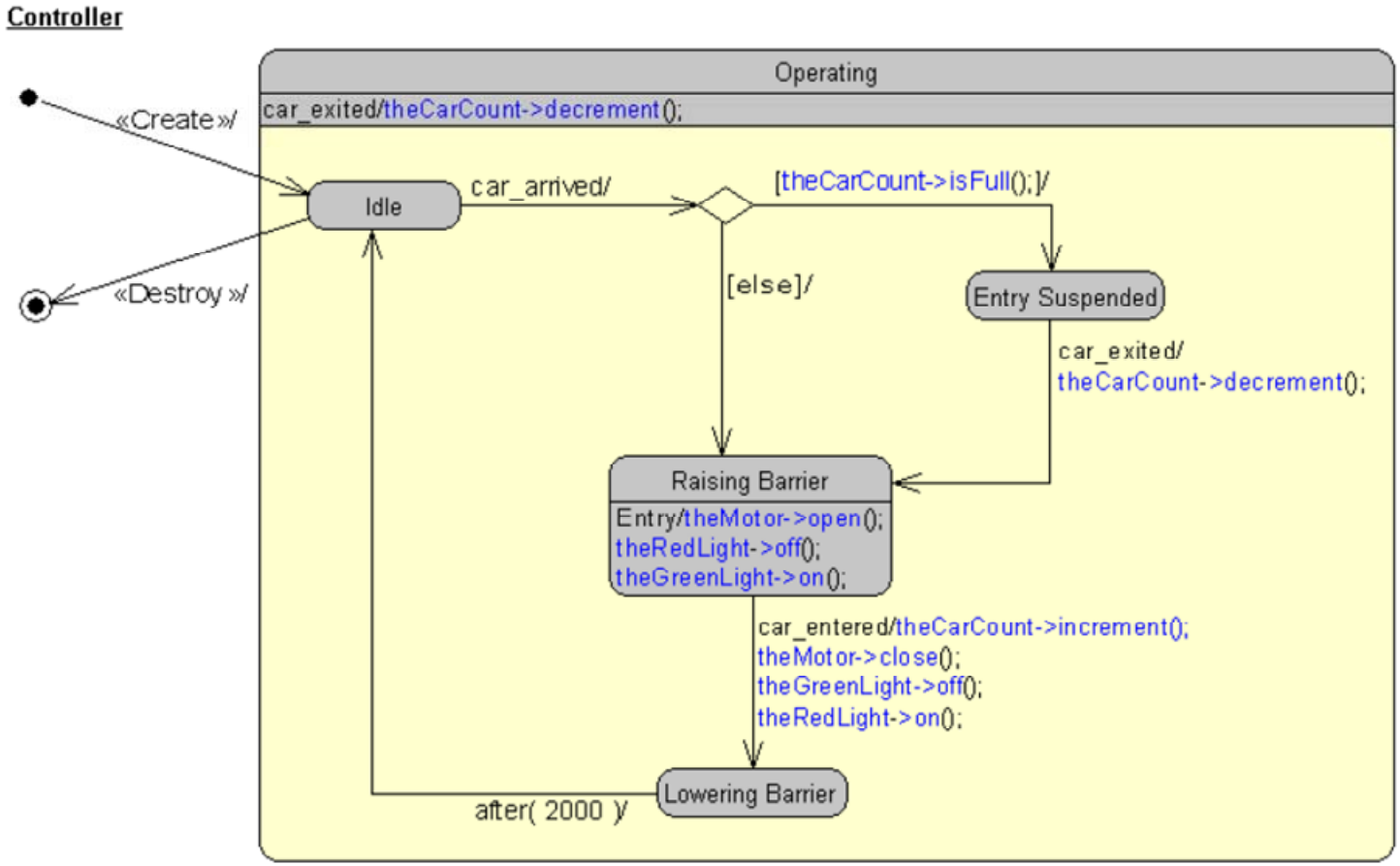
Class Diagram (2)



State Diagram (1)

- Ziele:
  - Identifikation des dynamischen Verhaltens von Objekten
- Methoden:
  - Identifikation der Zustände einer Komponente
  - Identifikation der möglichen Zustandsübergänge
  - Identifikation der Ereignisse die zu Übergängen führen
- Zeitpunkt: Designphase

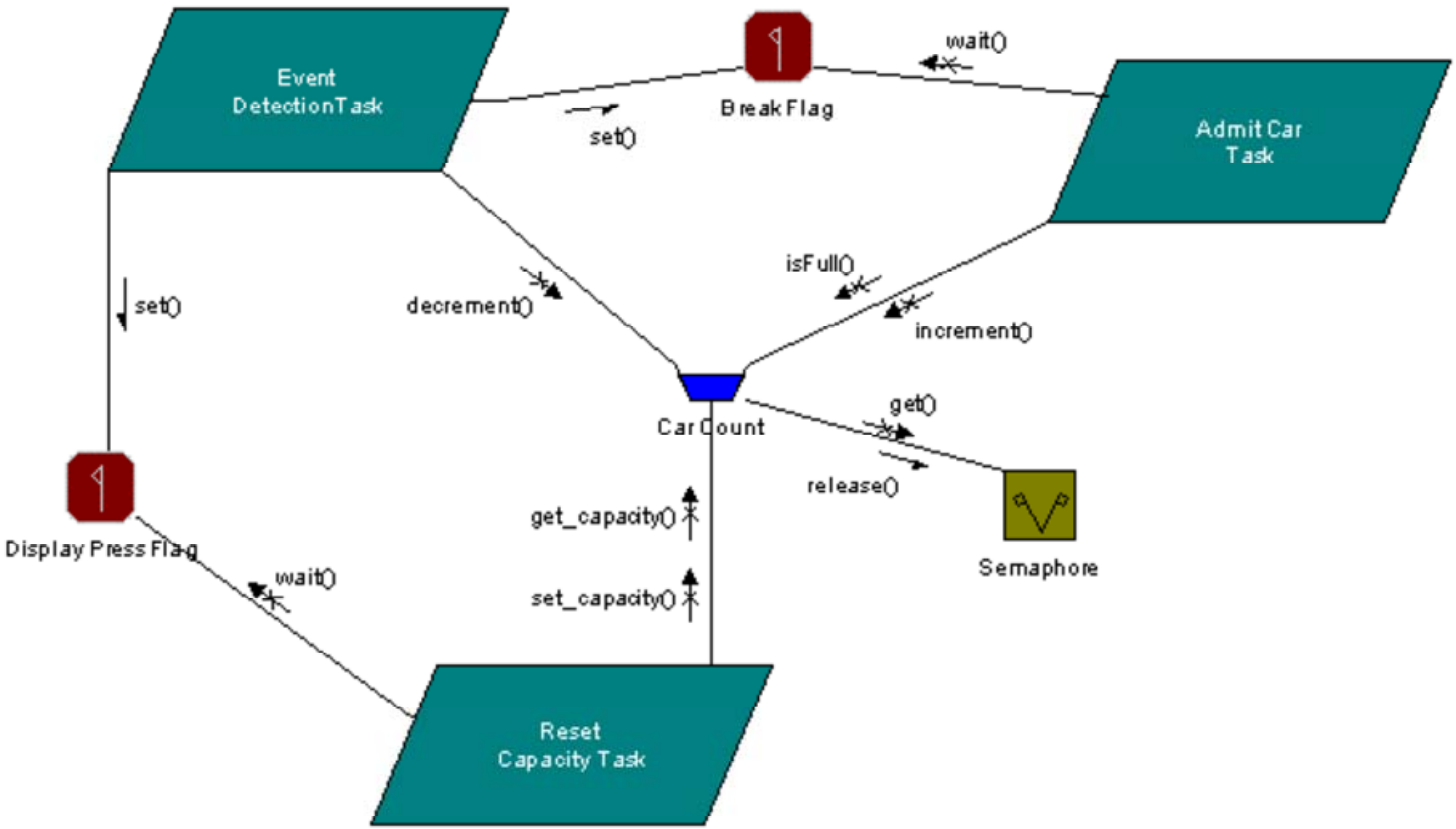
State Diagram (2)



Concurrency Diagram (1)

- Ziele:
  - Spezifikation von Prozessen und deren Interaktion
- Methoden:
  - Identifikation der einzelnen Prozesse
  - Spezifikation der Prozessinterkommunikation
  - Spezifikation des wechselseitigen Ausschlusses
  - Synchronisation von Prozessen
- Zeitpunkt: Designphase

Concurrency Diagram (2)



Diagrammtypen (Beispiel Artisan)

Diagrammtyp	Zweck
Aktivitätsdiagramme Activity D. (UML)	Zeigt Aktionssequenzen in verschiedenen Kontexten
Klassendiagramme Class D. (UML)	Zeigen Pakete und Paketabhängigkeiten, Klassen und deren Eigenschaften und Beziehungen
Nebenläufigkeitsdiagramme Concurrency D. (Artisan)	Modelliert Prozesse und Prozessinteraktionsmechanismen
Bedingungsdiagramme Constrain D. (Artisan)	Spezifiziert nicht-funktionale Systemanforderungen
Objektinteraktionsdiagramme Object Collaboration D. (UML)	Zeigt Objektinteraktionen für spezielle Anwendungsfälle oder für den allgemeinen Fall
Sequenzdiagramme Object Sequence D. (UML)	Zeigt Subsystem/Gerät/Objekt/Prozess-Interaktionen für einen kompletten Testfall als Sequenz von Systemnachrichten
Zustandsdiagramme State D. (UML)	Beschreibt zustandsbasiert das dynamische Verhalten von Klassen oder Subsystemen

Diagrammtypen Fortsetzung

Strukturdiagramm Structure D. (UML)	Modelliert die Zusammensetzung von strukturierten Klassen durch die Benutzung von Teilen, Ports, Schnittstellen und IO-Flüssen
Systemarchitekturdiagramm System Architecture D. (Artisan)	Zeigt Systemhardwarekomponenten, sowie deren Zusammenhänge und Eigenschaften
Beziehungstabellendiagramme Table Relationships D. (Artisan)	Spezifiziert die Beziehungen zwischen persistenten Dateneinheiten
Anwendungsfalldiagramme Use Case D. (UML)	Identifiziert Systemdienste in Form einer funktionellen Beschreibung und verbindet diese mit externen Aktoren
Generelles Grafikdiagramm General Graphics D. (Artisan)	Unterstützt nicht-spezifische Diagrammmöglichkeiten
Text-Diagramm Text D. (Artisan)	Unterstützt die Möglichkeit zur Speicherung von Textdokumenten in einem Modell

Modellierung von Echtzeitsystemen

Zerberus

Gefördert durch:

Bundesministerium
für Bildung
und Forschung

KAYSER-THREDE

Lehrstuhl Prof. Knoll
Lehrstuhl Prof. Broy

BECK
THE POINT OF SYNERGY

Motivation – Mögliche Zielanwendungen

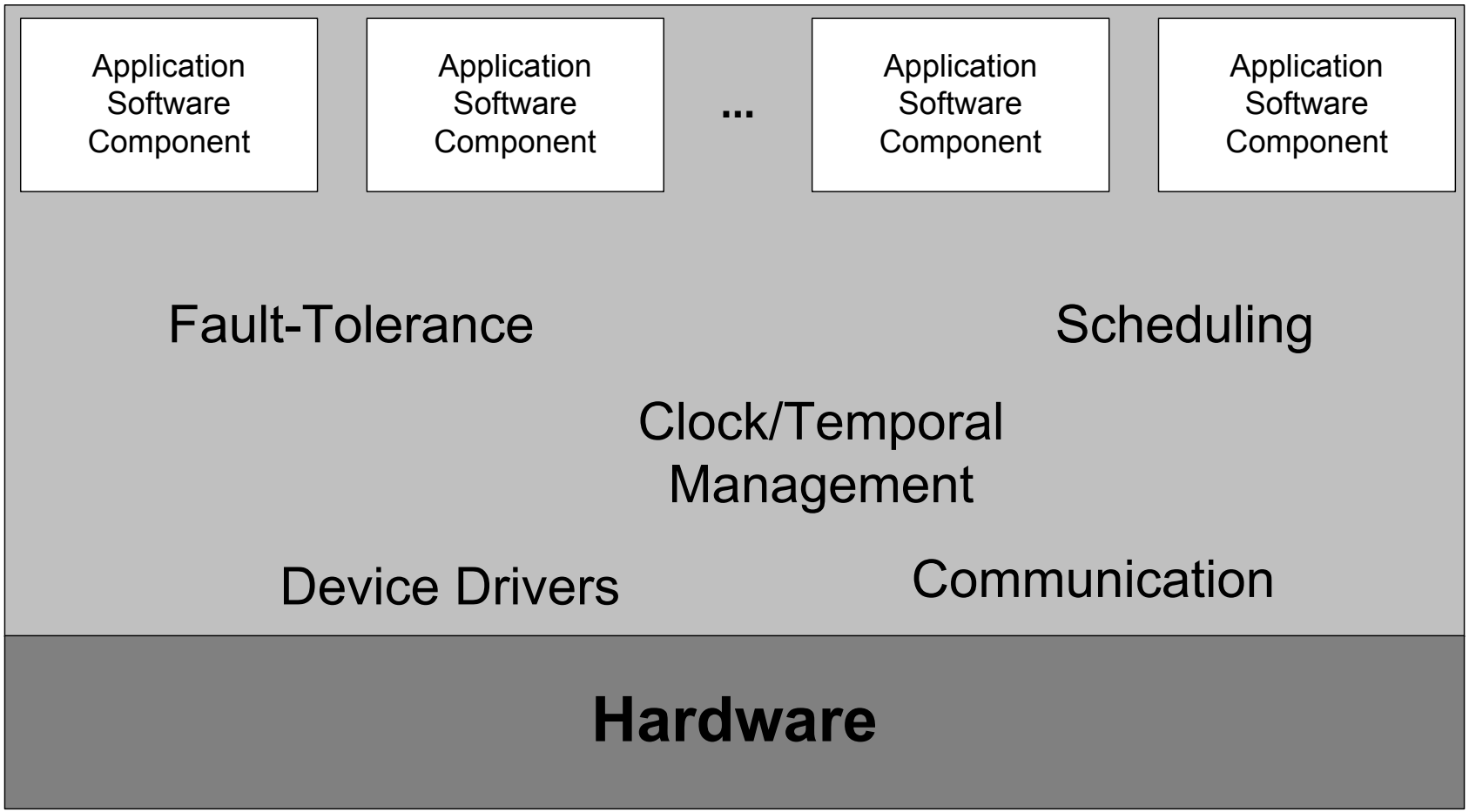
Robotik

Steuerung von Windkrafträdern

Medizinanwendungen

Bei all diesen Systemen ist neben der korrekten Umsetzung der Anwendungsfunktionalität auch noch die Umsetzung von Fehlertoleranzmechanismen, Kommunikation im verteilten System, sowie das Scheduling zu beachten.

Anwendungsfunktionalität ist nur ein kleiner Teil des Systems

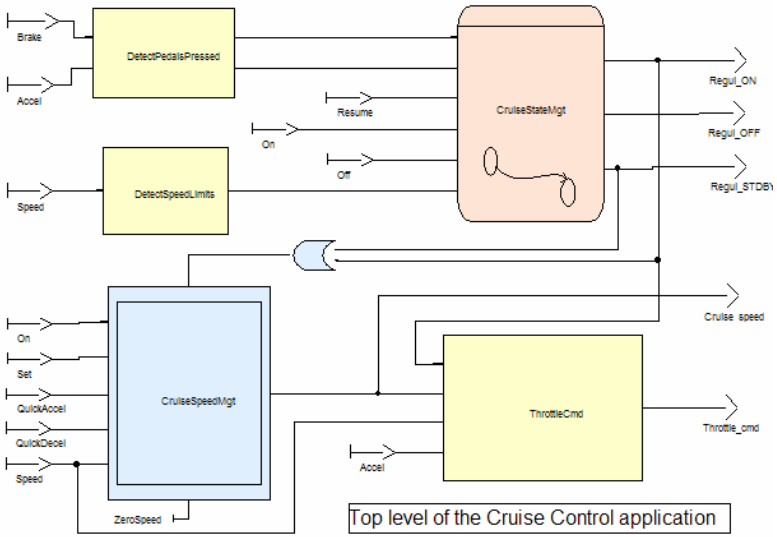


Anforderungen von Echtzeitsystemen an Entwicklungswerkzeuge

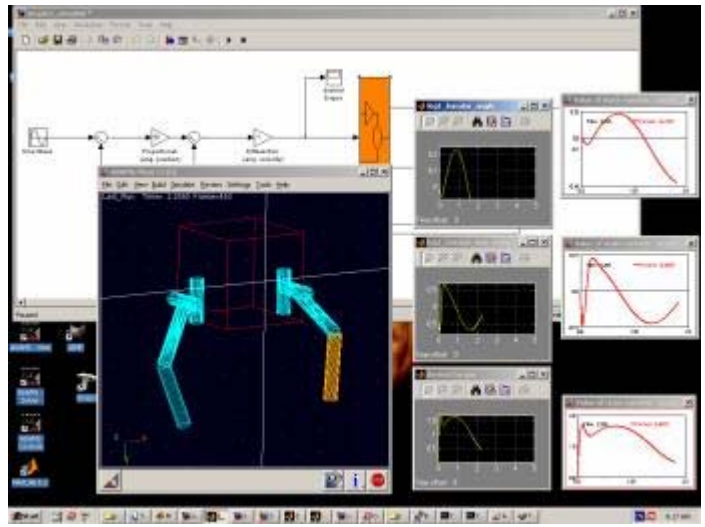
- Echtzeitsysteme sind verteilte Systeme
  - Prozess-/Threadmanagement
  - Kommunikation (zwischen Prozessen und zwischen Rechnern)
  - Determinismus in der Ausführung (Vermeidung von Race Conditions)
  - Schutz der kritischen Bereiche
- Unterstützung bei hardwarenaher Programmierung
- Notwendigkeit eines vorhersagbaren Ausführungsverhaltens (v.a. bezogen auf die Zeit)
- Bedienbarkeit durch Experten der Anwendungsdomäne

Modellbasierte Entwicklung: Anwendungslogik

– Für Anwendungslogik existieren bereits einige Werkzeuge mit automatischer Codegenerierung:



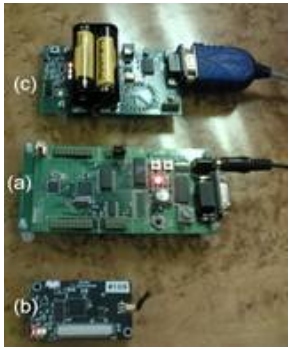
SCADE

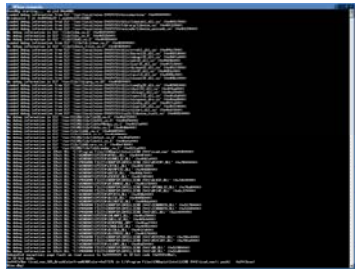


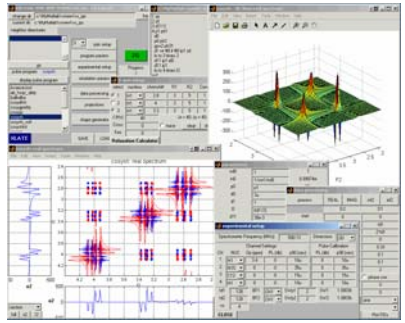
Matlab / Simulink

– Wieso gibt es keine solchen Werkzeuge für Code auf Systemebene?

Echtzeitsysteme sind typischerweise heterogen







Spezialbetriebssysteme	Echtzeitbetriebssysteme	Standardbetriebssysteme
C, Assembler	C, C++	Java, C++

⇒ **Ein allumfassender Codegenerator kann a priori gar nicht entwickelt werden**

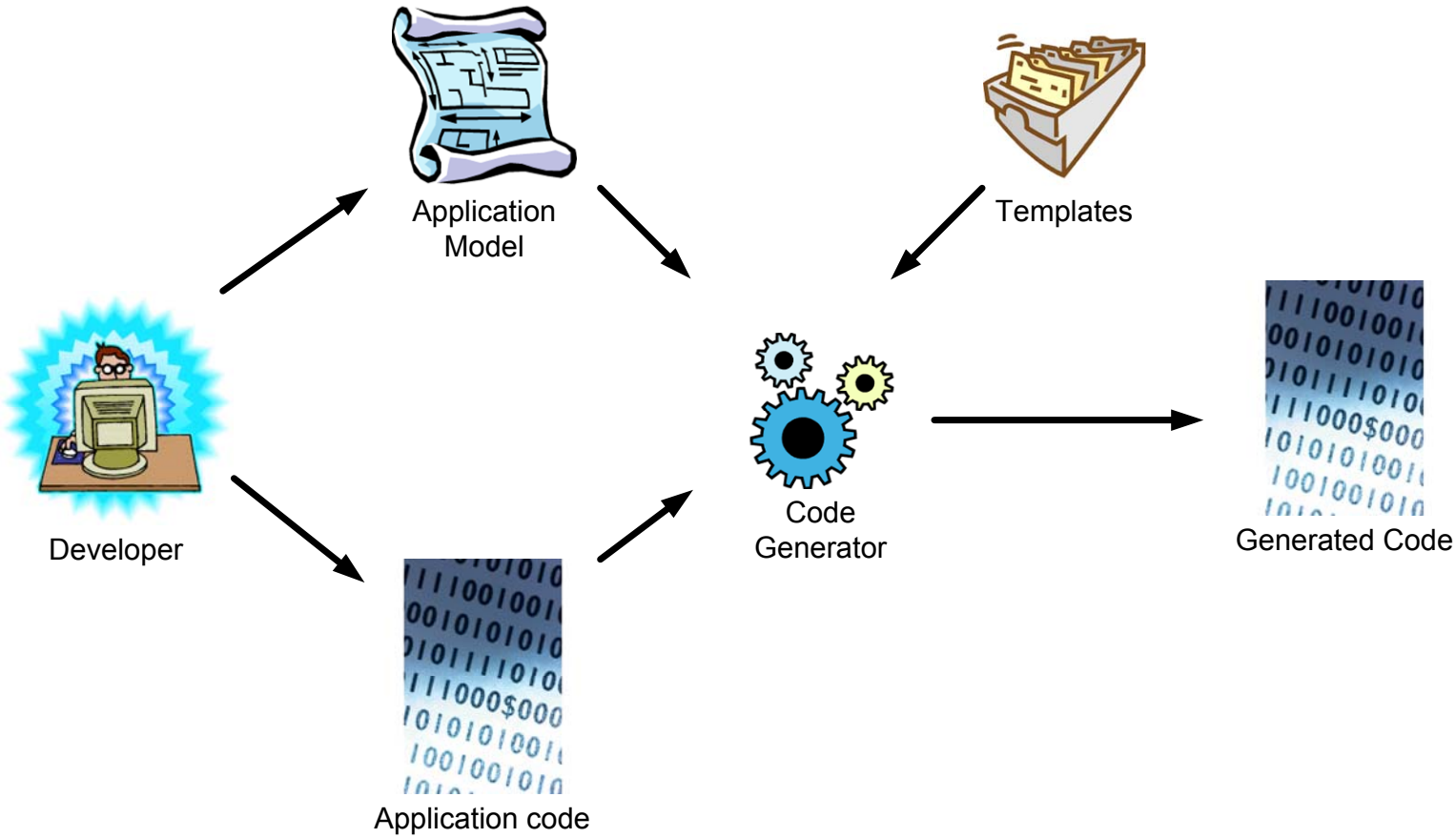
Ziele

- Maximaler Grad an Codegenerierung erforderlich
  - Neben der **Anwendungsfunktionalität** soll auch Code auf **Systemebene** (Prozessmanagement, Kommunikation, Fehlertoleranzmechanismen) generiert werden.

    ⇒ Notwendigkeit geeigneter Modelle
    ⇒ Problem: Heterogenität eingebetteter Systeme, ein allumfassender Codegenerator ist a priori nicht möglich.

- Forderungen an zukünftige Codegeneratoren: Notwendigkeit der einfachen Erweiterbarkeit in Bezug auf
  - das zugrunde liegende Metamodell
  - die Codegenerierungsfähigkeiten

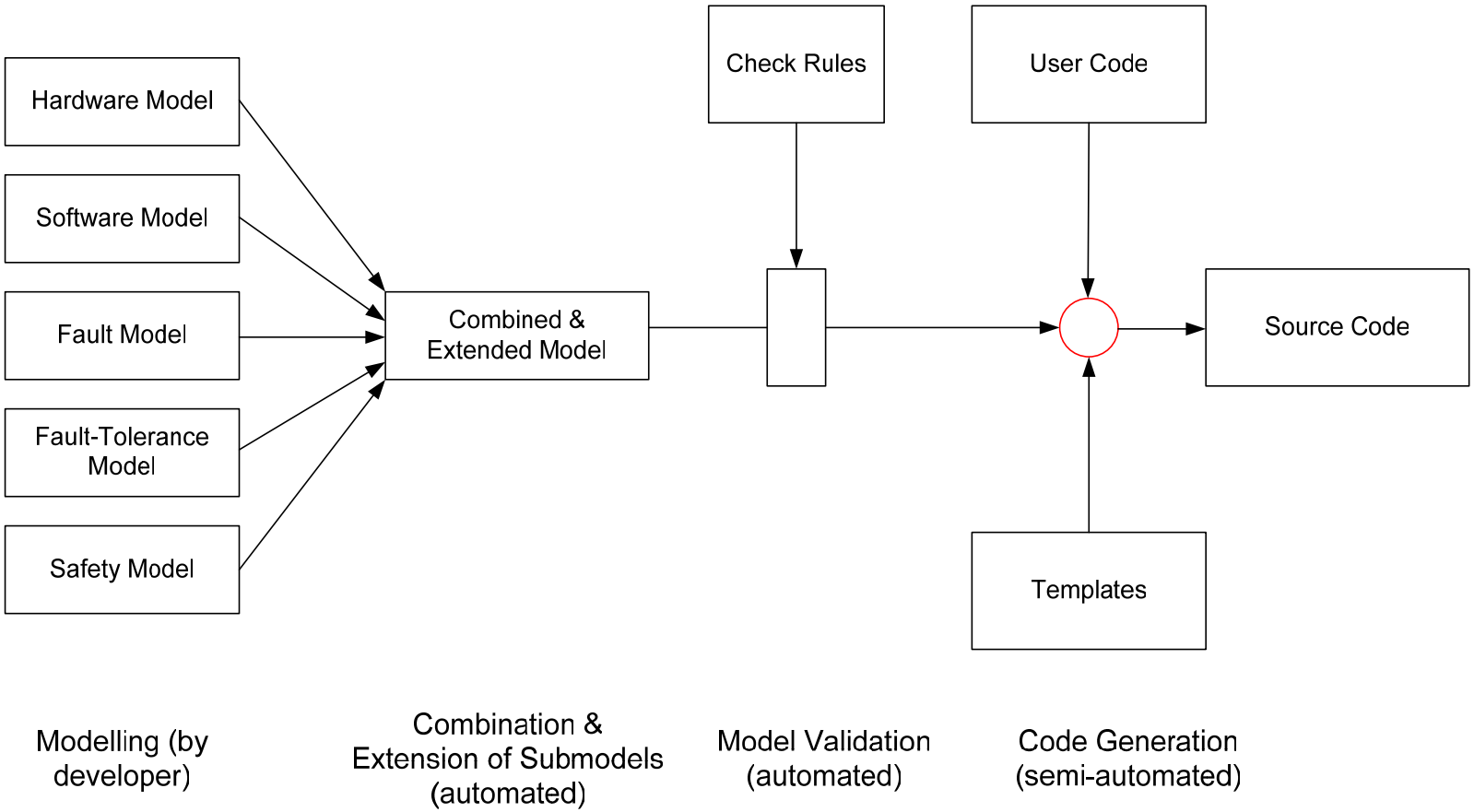
Ansatz: Vorlagenbasierte Codegenerierung



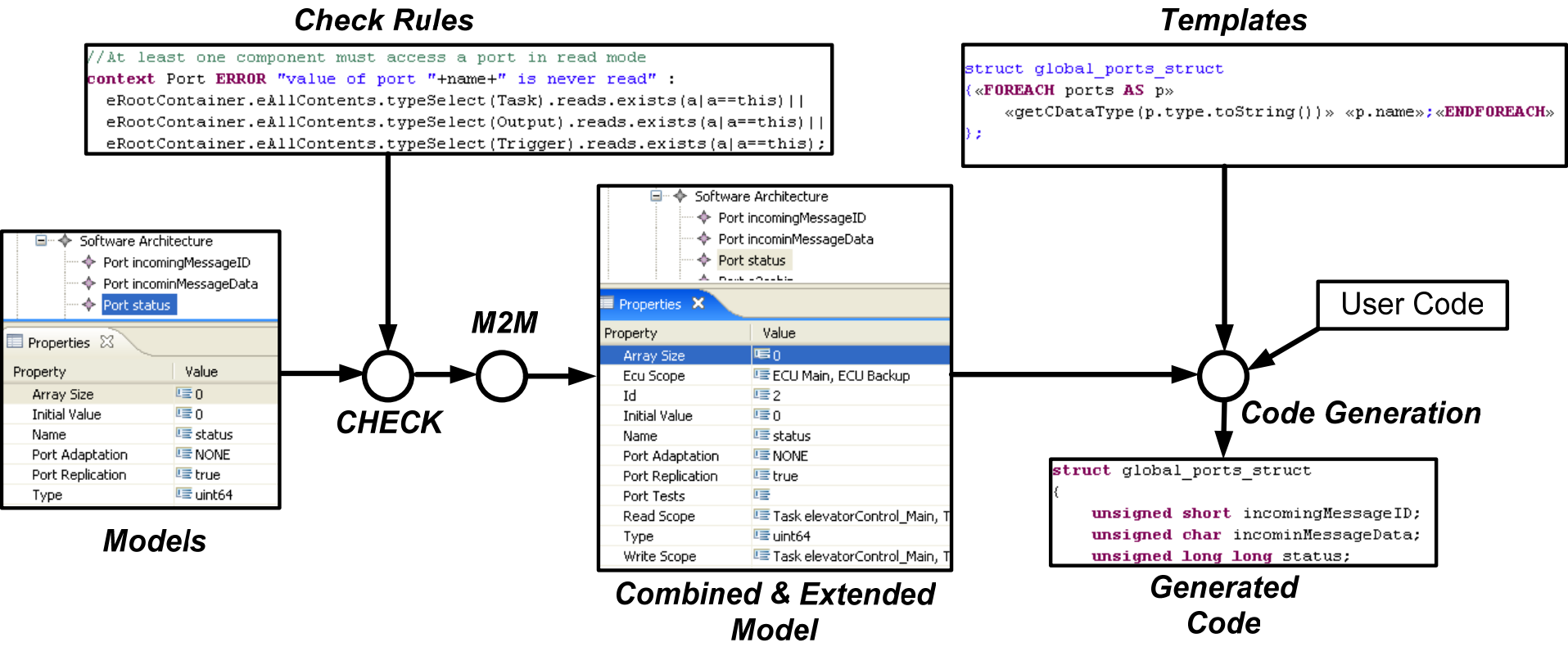
Vorteile des Ansatzes

- Vorlagen können wiederbenutzt werden.
- Vorlagen decken Einzelaspekte des Systems ab und können von Spezialisten implementiert werden.
- Erleichterte Erstellung der Vorlagen durch Verwendung einer Technik ähnlich von Präprozessoranweisungen.
- Codegeneratoren sind einfach erweiterbar:
  - In Bezug auf die Generierungsfähigkeit durch Hinzufügen von Vorlagen
  - In Bezug auf das Metamodell durch Verwendung von Metacodegeneratoren
- Erfahrungen zeigen deutliche Effektivitätssteigerungen (Faktor 3-10)

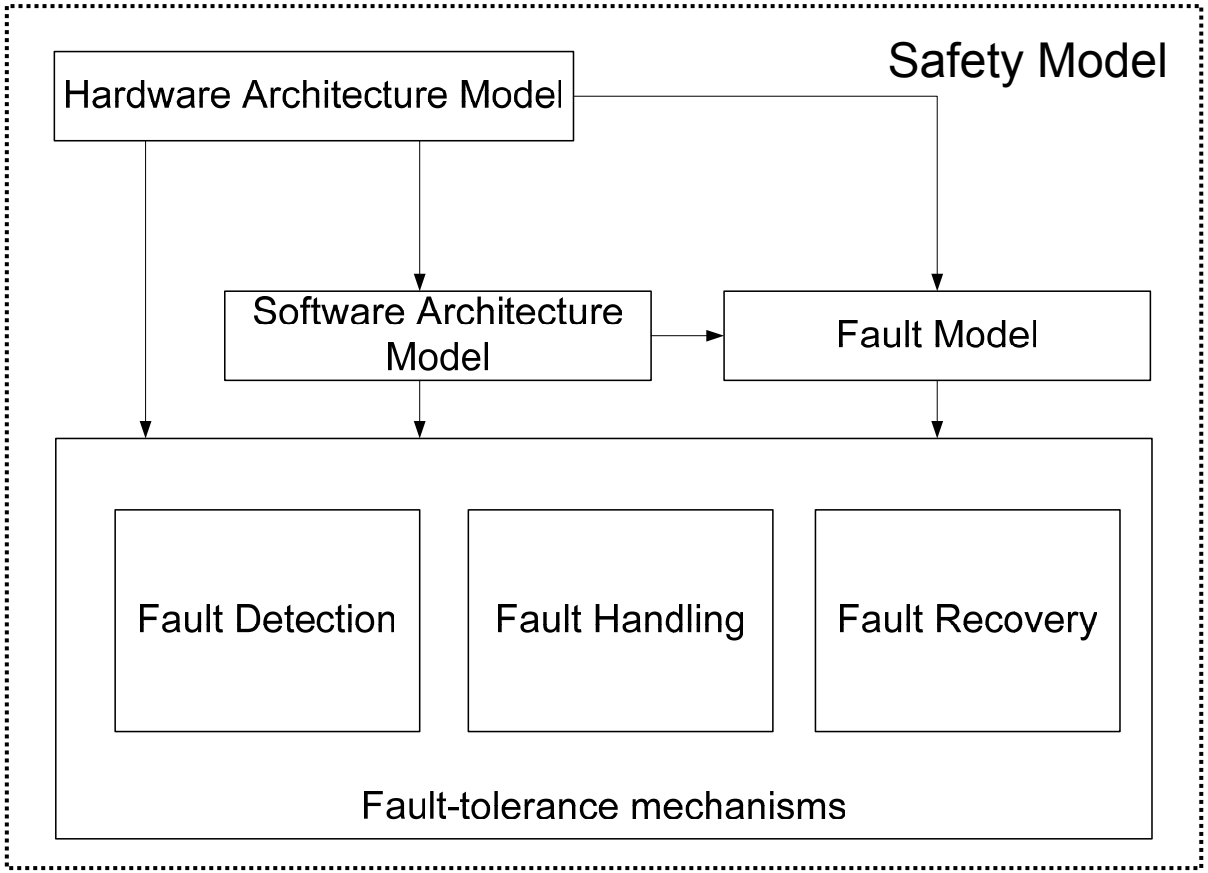
Zerberus: Entwicklungsprozess



Ein konkretes Beispiel



Abhängigkeiten zwischen Modellen



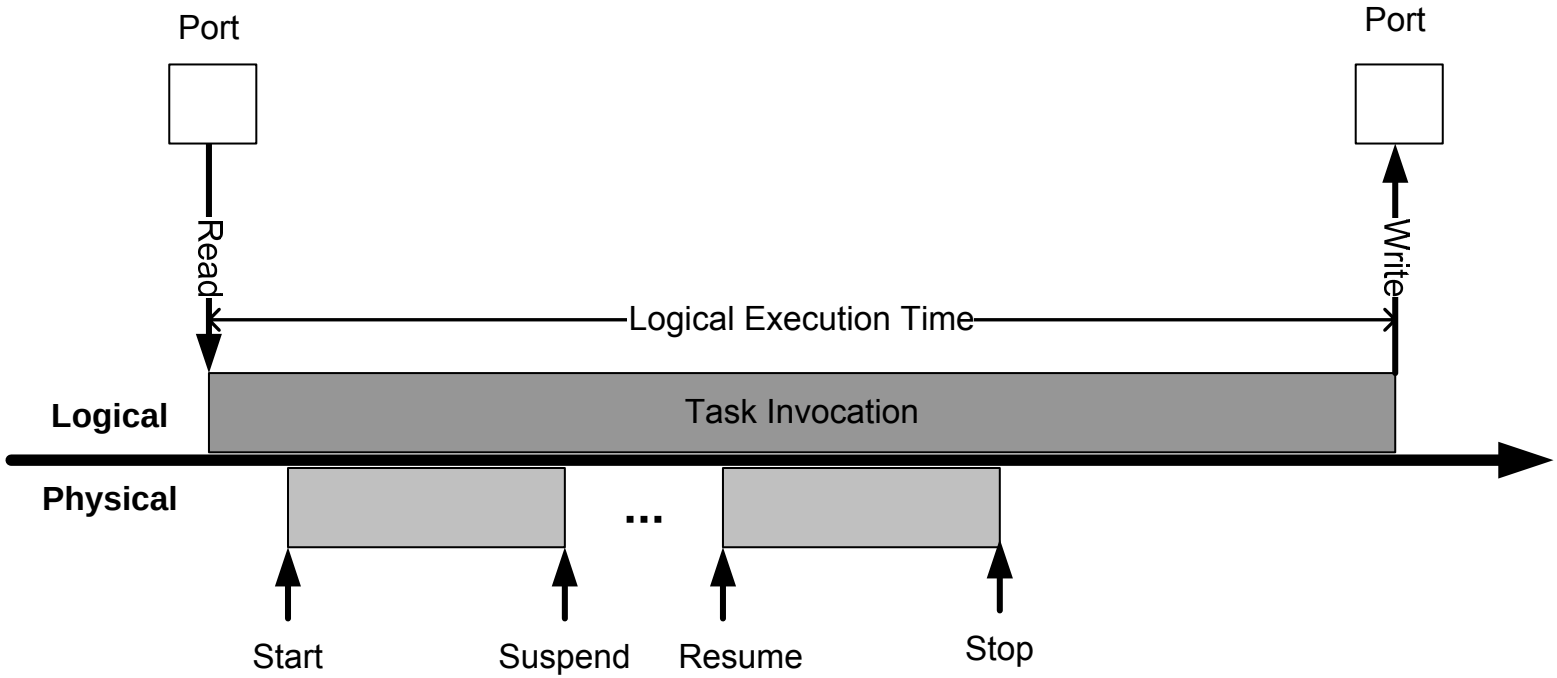
Hardwaremodell

- Das Hardwaremodell dient zur Beschreibung der verwendeten Recheneinheiten (ECU: electronic control units) und der Netzwerke
- Attribute der ECUs:
  - Prozessortyp
  - Sensoren / Aktuatoren
  - Betriebssystem / Programmiersprache
  - Speichergröße
  - Uhreneigenschaften
  - …
- Unterstützte Netzwerktypen:
  - Ethernet
  - CAN
  - TTP

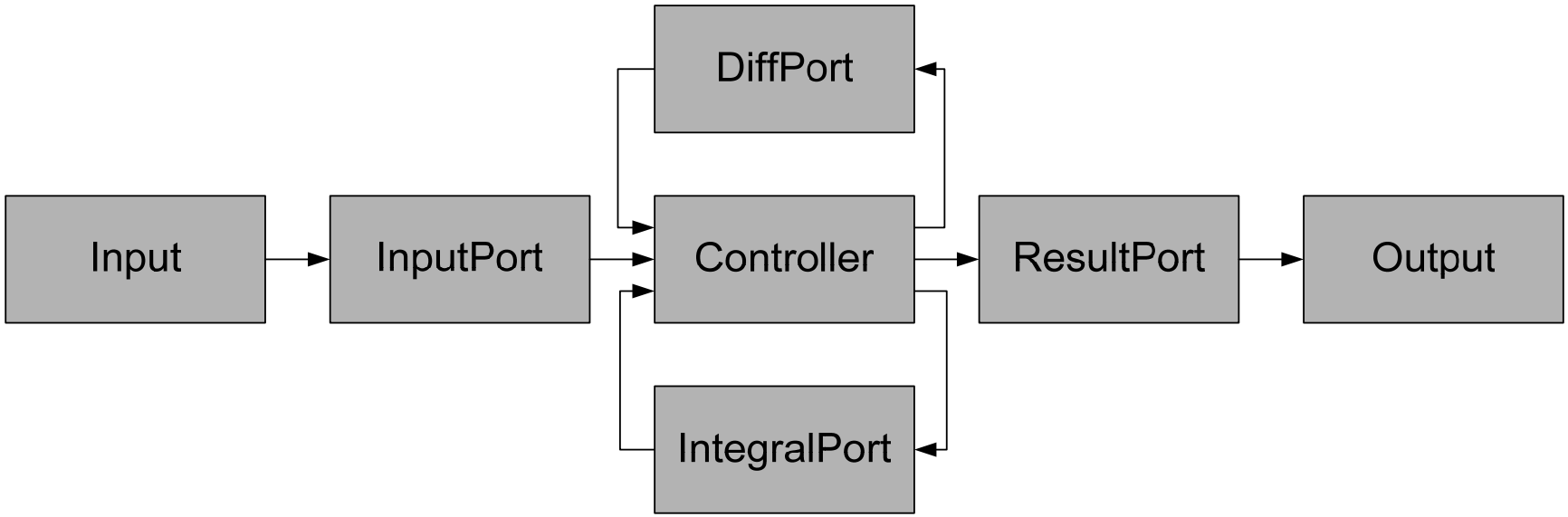
Softwaremodell

- Das Softwaremodell enhält eine Beschreibung von Komponenten:
  - Input: Einlesen von Daten über einen Sensor
  - Output: Ausgeben von Ergebnissen über einen Aktor
  - Task: Berechnen von Regelungsfunktionen
  - Port: globale Variable zur Kommunikation
- Das Softwaremodell enthält einen eindeutigen Ausführungsplan für diese Komponenten:
  - Zeitgesteuerte Ausführung mit logischen Ausführungszeiten
  - Zustände zum Abbilden einzelner Systemzustände (z.B. Initialisierung, Normalbetrieb, Fehlerbetrieb)
- Das Modell ist deterministisch und trennt Funktionalität (Input/Output/Task) vom Zustand (Port)
- Das Modell basiert dabei auf Vorarbeiten von TTA (Kopetz) und Giotto (Henzinger).

Logische Ausführungszeit



Beispiel: PID-Regler



Fehlermodell

- Das Fehlermodell beschreibt Fehlerbereiche und die Effekte der entsprechenden Fehler
- Fehlerbereich: Menge von Komponenten, die von einem Fehler betroffen sind. Dabei wird nicht unterschieden, welche Teilkomponente fehlerhaft ist, es wird immer der ganze Bereich als fehlerhaft angenommen.
- Fehlereffekte: Fehler können sich unterschiedlich auswirken, je nach Art des Fehlers und nach der Art der betroffenen Komponente
- Beispiele für Fehlereffekte:
  - Auf Taskebene: Nicht-Einhalten der Ausführungszeit, fehlerhaftes Ergebnis
  - Auf Netzwerkebene: Verfälschung einer Nachricht, Verlust einer Nachricht

Fehlertoleranzmechanismen

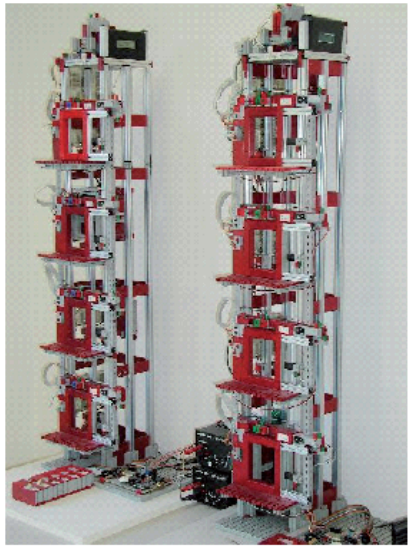
- Das Modell unterscheidet Tests zur Fehlererkennung, Mechanismen zur Reaktion auf Fehler der korrekten Teilmenge zur Fehlerbehandlung und der fehlerhaften Menge zur Fehlerbehebung.
- Beispiele:
  - Tests: Voter (Vergleich von redundanten Ergebnissen), Zeitüberschreitungstests, Absoluttests (falls Werte z.B. in einem Intervall liegen müssen)
  - Fehlerbehandlung: Umschalten auf redundante Einheit (z.B. 2-aus-3-Rechnersysteme)
  - Fehlerbehebung: Neustarten
- Weitere Mechanismen können sehr einfach hinzugefügt werden

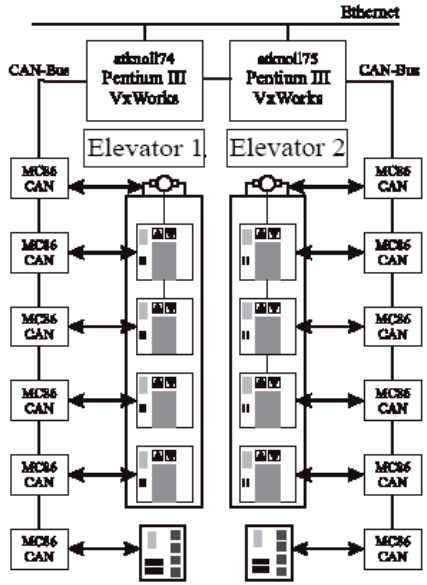
Demonstratoren

Schwebendes Pendel (2-aus-3)

⇒ Regelungszeiten von 2,5 ms

⇒ Nur 70 Zeilen Code zur Entwicklung notwendig





Aufzugsteuerung (Hot-Standby)

⇒ 500 Zeilen Code notwendig

⇒ SCADE Modell wird gerade integriert

Verifikation von Echtzeitsystemen

Einsatz von Formalen Methoden

Problemstellung

„As soon as we started programming, we found to our surprise that it wasn‘t as easy to get programs right as we had thought.
Debugging had to be discovered.
I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.“

Maurice Wilkes.
(Turing Award 1967)

Verifikation & Validierung

- Verifikation: Um die Korrektheit von Programmen in Bezug auf die Spezifikation zu garantieren, wird eine formale Verifikation benutzt. Dazu werden mathematische Korrektheitsbeweise durchgeführt.
- Validierung: Durch eine Validierung kann überprüft werden, dass das System als Modell hinreichend genau nachgebildet wird. Techniken:
  - Inspektion
  - Plausibilitätsprüfung
  - Vergleich unabhängig entwickelter Modelle
  - Vergleichmessung an einem Referenzobjekt

Übersicht über formale Methoden

- Deduktive (SW-)Verifikation
  - Beweissysteme, Theorem Proving
- Model Checking
  - für Systeme mit endlichem Zustandsraum
  - Anforderungsspezifikation mit temporaler Logik
- Testen
  - spielt in der Praxis eine große Rolle
  - sollte systematisch erfolgen $\rightarrow$ ausgereifte Methodik
  - ... stets unvollständig

Verifikation in der Realität

- In der Industrie wird der Begriff Verifikation häufig im Zusammenhang mit nicht funktionalen Methoden verwendet:
  - Testen, Strategien:
    - 100% Befehlsabdeckung (Statement Coverage)
    - 100% Zweigüberdeckung (Branch Coverage)
    - 100% Pfadüberdeckung (Path Coverage)
    - Siehe auch http://www.software-kompetenz.de/?10764
  - Code reviews
  - Verfolgbarkeitsanalysen

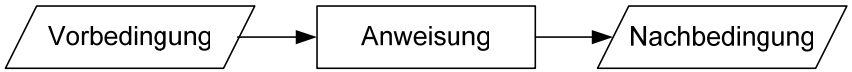
Testen

Mit Testen ist es möglich die Existenz von Fehlern nachzuweisen, nicht jedoch deren Abwesenheit.

- Testen ist von Natur aus unvollständig (non-exhaustive)
- Es werden nur ausgewählte Testfälle / Szenarien getestet, aber niemals alle möglichen.

Deduktive Methoden

- Nachweis der Korrektheit eines Programms durch math.-logisches Schließen
- Anfangsbelegung des Datenraums $\Rightarrow$ Endbelegung
- Induktionsbeweise, Invarianten
  - klass. Bsp: Prädikatenkalkül von Floyd und Hoare, Betrachten von Einzelanweisungen eines Programms:



- Programmbeweise sind aufwändig, erfordern Experten
- i.A. nur kleine Programme verifizierbar
- Noch nicht vollautomatisch, aber es gibt schon leistungsfähige Werkzeuge

Temporale Logik

- Mittels Verifikation soll überprüft werden, dass:
  - Fehlerzustände nie erreicht werden
    - Der Aufzug soll nie mit offener Tür fahren.
  - ein System irgendwann einen bestimmten Zustand erreicht (und evtl. dort verbleibt)
    - Nach einer endlichen Initialisierungsphase, geht der Aufzug in den Betriebsmodus über.
  - Zustand x immer nach Eintreten des Zustandes y auftritt.
    - Nach Drücken des Tasters im Stockwerk wird der Aufzug in einem späteren Zustand auch dieses Stockwerk erreichen.
- Um solche Aussagen auch für Rechner lesbar auszudrücken, kann temporale Logik, z.B. in Form von LTL (linear time temporal logic), verwendet werden.
- In LTL wird Zustandsübergänge und damit auch die Zeit als diskrete Folge von Zuständen interpretiert.

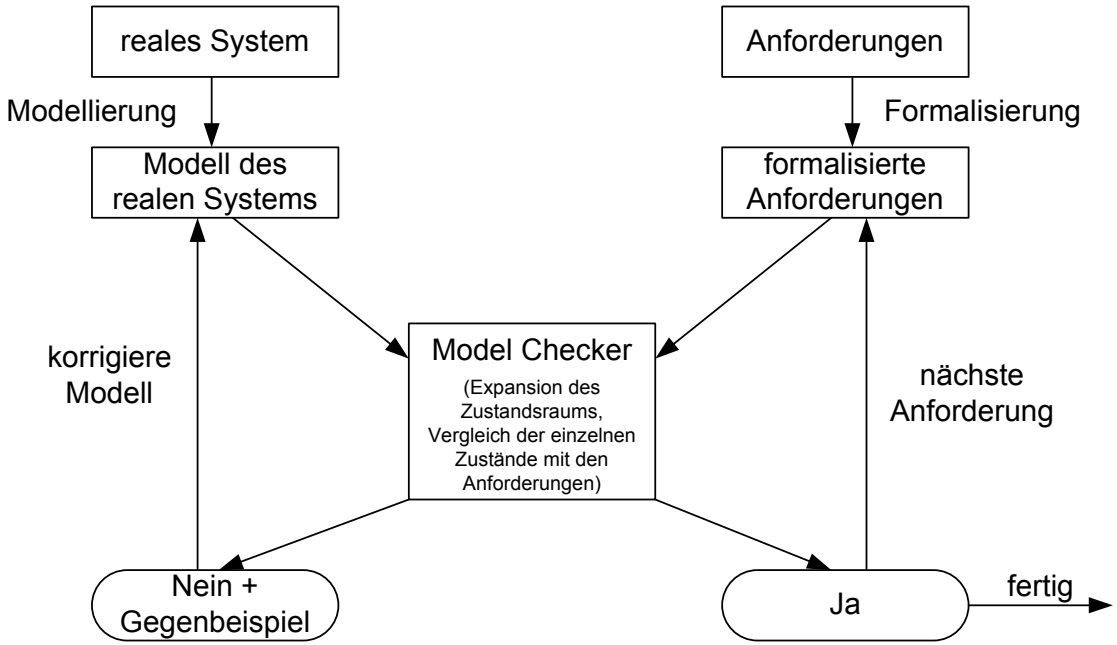
Kripke-Struktur

- Zur Darstellung eines Systems werden Kripke-Strukturen K = (V,I,R,B) und eine endliche Menge P von atomaren logischen Aussagen verwendet.
  - V: Menge binärer Variablen (z.B. Tür offen, Aufzug fährt)
  - Die Zustandsmenge S ergibt sich aus allen möglichen Kombinationen über V, somit gilt $S = B^V$
  - Menge der möglichen Anfangszustände $I \subseteq S$
  - R: Transitionsstruktur $R \subseteq S \times S$
  - B: Bewertungsfunktion $S \times P \rightarrow \{true,false\}$ zur Feststellung, ob ein Zustand eine Eigenschaft auf P erfüllt
- Mittels Model-Checking muss nun nachgewiesen werden, dass eine gewisse Eigenschaft P ausgehend von den Anfangszuständen
  - immer gilt
  - schließlich erfüllt wird
  - …

Explizites Model Checking: Verfahren

- Ausgehend von den Startzuständen exploriert der Model Checker mögliche Nachbarzustände:
  - Auswahl eines noch nicht evaluierten Zustandes
  - Prüfung aller möglichen Zustandsübergange:
    - bereits bekannter Zustand: verwerfen
    - unbekannter Zustand, Eigenschaft prüfen
      - falls Eigenschaft nicht erfüllt, Abbruch und Präsentation eines Gegenbeispiels
      - falls erfüllt, zur Menge der nicht evaluierten Zustände hinzufügen
  - Abbruchbedingung: alle erreichbaren Zustände wurden überprüft
- Problem: Zustandsexplosion

Umgang mit Model Checking



Weitere Strategien

- Symbolische Model Checker:
  - Grundidee: durch eine einfache Formel können viele Zustände zu einem Zustand gekapselt werden
  - Verwendung von binären Entscheidungsdiagrammen (binary decision diagrams – BDD)
- Bounded Model Checker:
  - Grundidee: durch Abstraktion können viele Zustände zusammengefasst werden (z.B. Aufteilung der ganzen Zahlen in positive, negative Zahlen und 0)
  - Häufig sind diese Model Checker pessimistisch (Präsentation von Gegenbeispielen, die keine sind)

Probleme mit formalen Methoden

- Entwickler empfinden formale Methoden häufig als zu kryptisch
- Beispiel TLA:

$$
\begin{aligned}
HCini \triangleq &\ \wedge\ hr \in \{0, \ldots, 23\} \\
HCnxt \triangleq &\ \wedge\ hr' = IF\ hr \neq 23\, THEN\ hr + 1\ ELSE\ 0 \\
HC \triangleq &\ \wedge\ HCini \\
&\ \wedge\ \Box HCnxt
\end{aligned}
$$

- Neue Ansätze: Erweiterung der Programmier / Modellierungssprachen, automatische Übersetzung

1. Beispiel: Verifikation in Esterel Studio

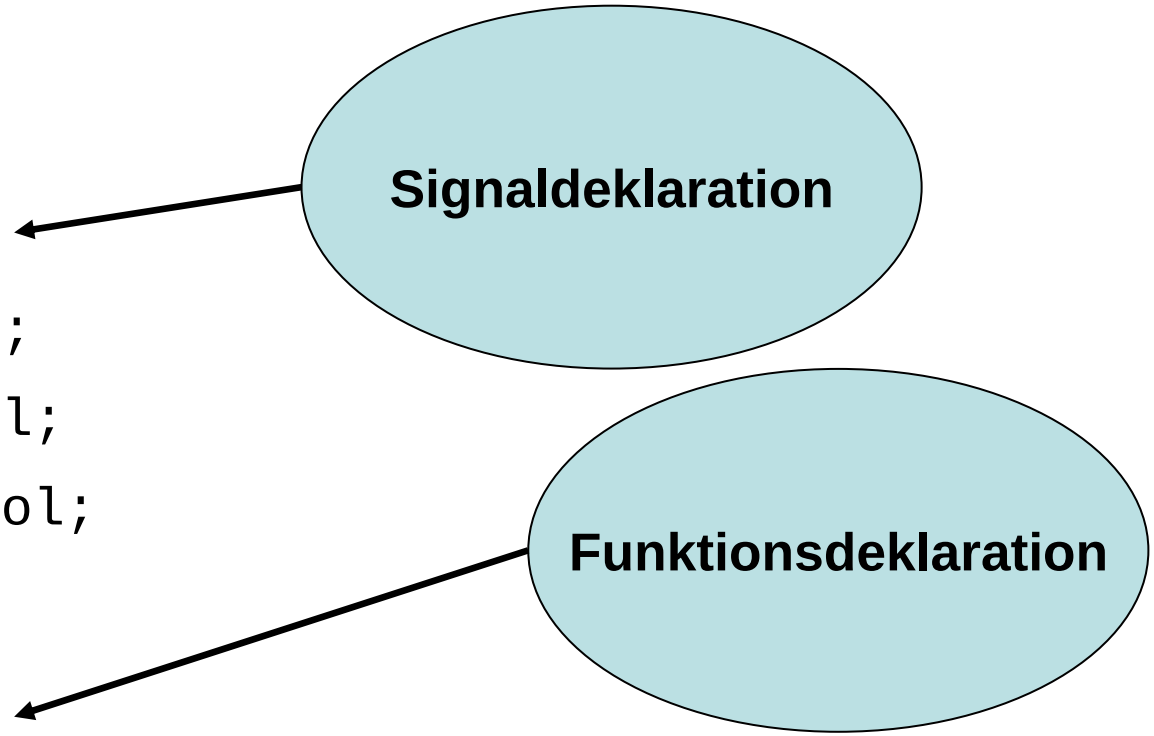
- Esterel Studio bietet eine eingebaute Verifikationsfunktionalität zur einfachen Verifikation von Programmen
- Zur Modellierung der verschiedenen Eigenschaften kann das Schlüsselwort `assert` verwendet werden.
- Im Verifikationsmodus können die Eigenschaften dann getestet werden, dabei stehen Methoden zum unbegrenzten / in der Testtiefe begrenzten Modell Checking, sowie zum symbolischen Model Checking zur Verfügung.
- Grundsätzliche Vorgehensweise:
  - Finden von Fehlern in den Annahmen / Modellen mit begrenztem Model Checker
  - Nachweis der Korrektheit des verbesserten Modells in Bezug auf die korrigierten Eigenschaften mit unbegrenztem Model Checking / symbolischen Model Checking
- Details siehe Demonstration

2. Beispiel: BoogiePL in Kombination mit Z3

- Grundidee: Verifikation von C# Programmen durch Erweiterung Spec# von Microsoft
- Das Spec#-Programm wird in Zwischensprache BoogiePL übersetzt. Die geforderten Eigenschaften werden dann mit Hilfe des SMT-Solvers Z3 nachgewiesen.
- Grundkonstrukte (Ausschnitt):
  - assert: Annahmen die durch den Beweiser verifiziert werden müssen
  - assume: Annahme durch den Benutzer, Zuständsübergänge, die der Annahme widersprechen werden vom Beweiser ignoriert
  - havoc: Zuweisung eines beliebigen Wertes an eine Variable (z.B. zur Simulation der Umgebung)

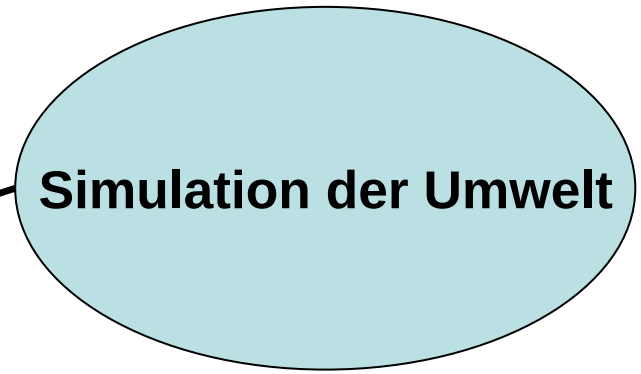
Boogie-Programm: Türbeispiel vereinfacht

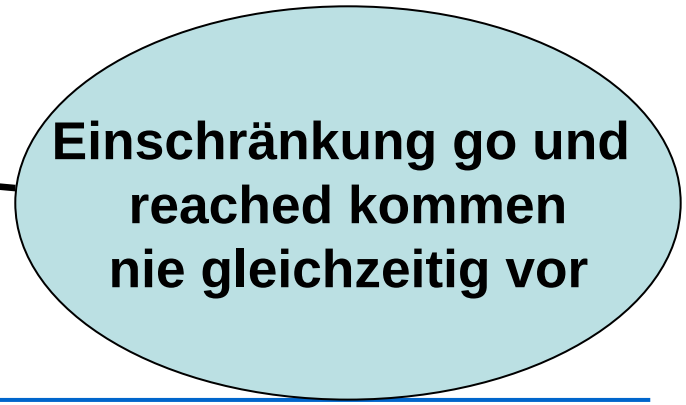
```
var open: bool;
var close: bool;
var go: bool;
var reached: bool;
var sig_open: bool;
var sig_close: bool;

procedure Door();
  modifies open,close,go,reached,sig_open,sig_close;
```



Boogie-Programm: Türbeispiel vereinfacht

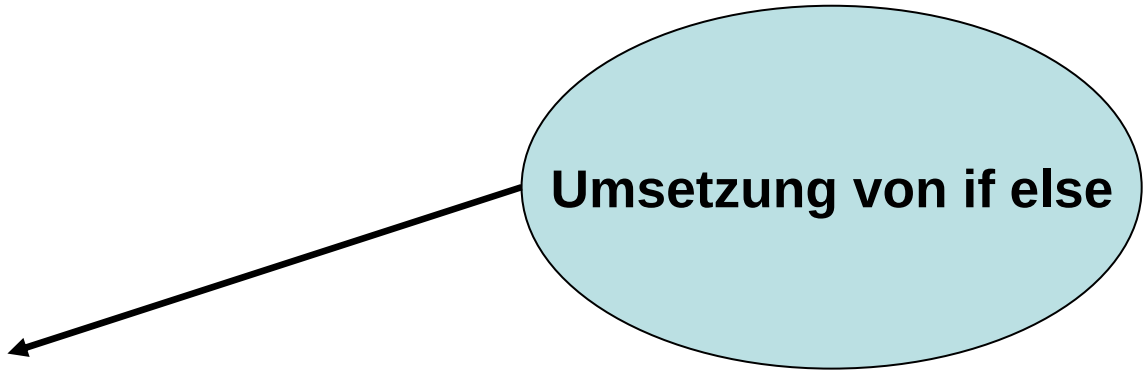
```
implementation Door()
{
Begin:
   havoc open;
   havoc close;
   havoc go;
   havoc reached;
   assume !go && !reached;
```





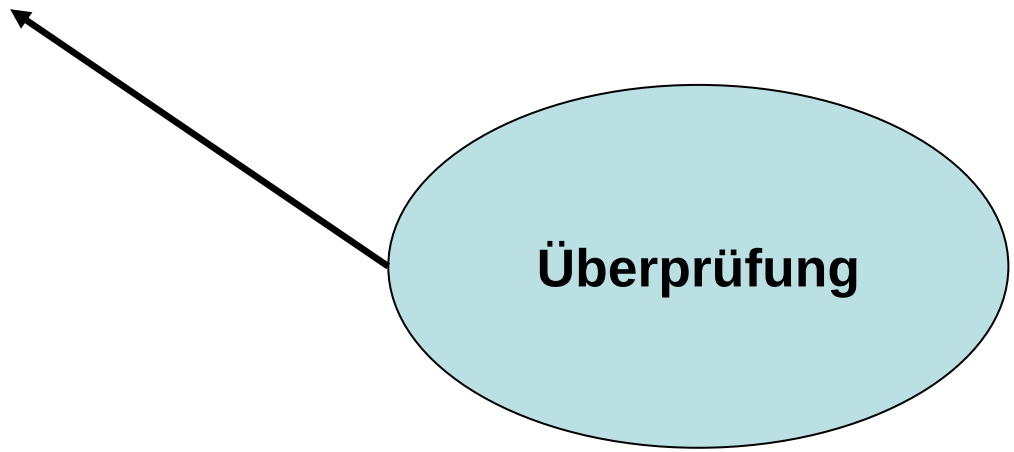
Boogie-Programm: Türbeispiel vereinfacht

```
  goto Open,Close;
Open:
  assume open;
  sig_open:=true;
  goto End;
Close:
  assume !open && close;
  sig_close:=true;
  goto End;
```



Boogie-Programm: Türbeispiel vereinfacht

```
End:
   assert open ==> sig_open;
   assert close ==> sig_close;
   goto Begin;
}
```



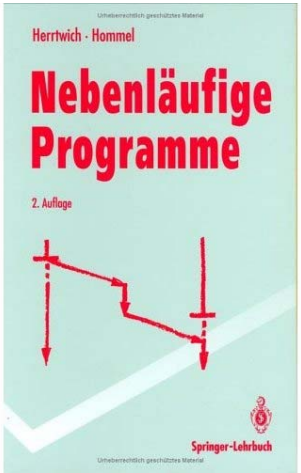
Kapitel 3

Nebenläufigkeit

Inhalt

- Motivation
- Unterbrechungen (Interrupts)
- (Software-) Prozesse
- Threads
- Interprozesskommunikation (IPC)

Literatur



R.G.Herrtwich, G.Hommel, Nebenläufige Programme 1998

A.S.Tanenbaum, Moderne Betriebssysteme, 2002

- Links:
  - http://www.beyondlogic.org/interrupts/interupt.htm
  - http://www.llnl.gov/computing/tutorials/pthreads/

Definition von Nebenläufigkeit

- **Allgemeine Bedeutung**: Nebenläufigkeit bezeichnet das Verhältnis von Ereignissen, die nicht kausal abhängig sind, die sich also nicht beeinflussen. Ereignisse sind nebenläufig, wenn keines eine Ursache des anderen ist. Oder anders ausgedrückt: Aktionen können nebenläufig ausgeführt werden (sie sind parallelisierbar), wenn keine das Resultat der anderen benötigt.
- **Bedeutung in der Programmierung**: Nebenläufigkeit bezeichnet hier die Eigenschaft von Programmcode nicht linear hintereinander ausgeführt zu werden, sondern parallel ausführbar zu sein.

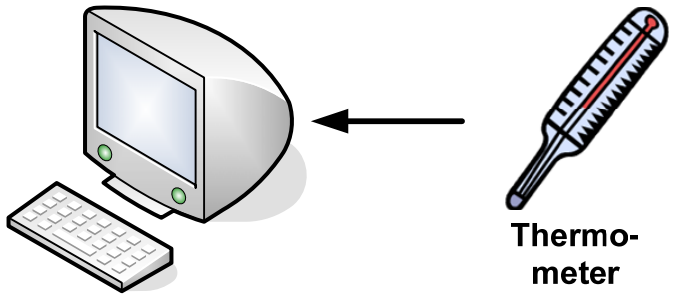
  Die Nebenläufigkeit von mehreren unabhängigen Prozessen bezeichnet man als *Multitasking*;

  Nebenläufigkeit innerhalb eines Prozesses als *Multithreading*.

Motivation

- Gründe für Nebenläufigkeit in Echtzeitsystemen:
  - Echtzeitsysteme sind häufig verteilte Systeme (Systeme mit mehrere Prozessoren).
  - Zumeist werden zeitkritische und zeitunkritische Aufgaben parallel berechnet.
  - Bei reaktiven Systemen ist die maximale Antwortzeit häufig limitiert.
  - Abbildung der parallelen Abläufe im technischen Prozeß

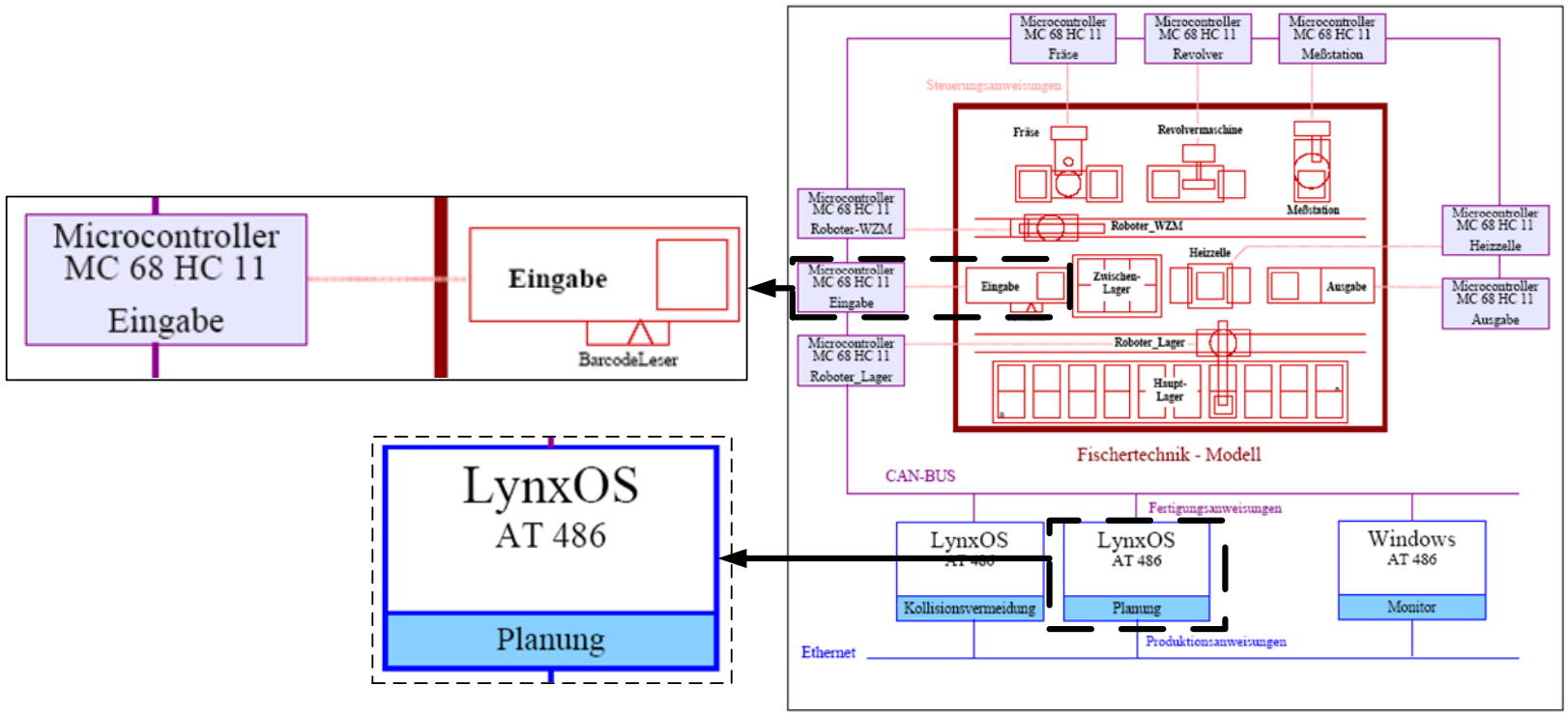
Anwendungsfälle für Nebenläufigkeit (Unterbrechungen)



Signal falls Temperaturwert überschritten wird
⇨ **Unterbrechungen (interrupts)**

Allgemeines Anwendungsgebiet: hauptsächlich zur Anbindung von externer Hardware

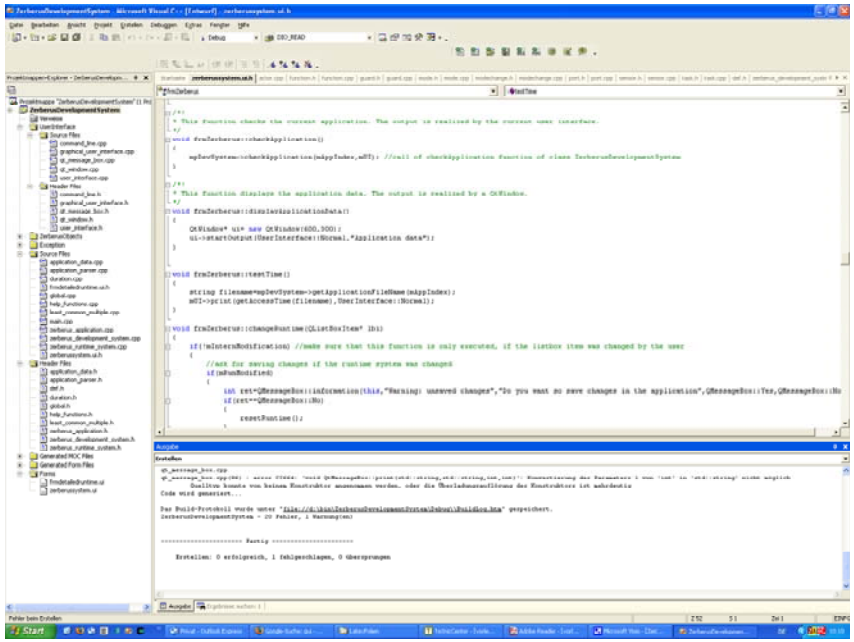
Anwendungsfälle für Nebenläufigkeit (Prozesse)



Verteiltes System zur Steuerung der Industrieanlage ⇨ **Prozesse (tasks)**

Allgemeine Anwendungsgebiete: verteilte Systeme, unterschiedlichen Anwendungen auf einem Prozessor

Anwendungsfälle für Nebenläufigkeit (Threads)



Reaktion auf Nutzereingaben trotz Berechnungen (z.B. Übersetzen eines Programms)

⇨ **leichtgewichtige Prozesse (Threads)**

Allgemeines Anwendungsgebiet: unterschiedliche Berechnungen im gleichen Anwendungskontext

Nebenläufigkeit

Unterbrechungen

Anbindung an die Umwelt

- Es muss ein Mechanismus gefunden werden, der es erlaubt Änderungen der Umgebung (z.B. einen Mausklick) zu registrieren.
- 1. Ansatz: **Polling**
  Es werden die IO-Register reihum nach Änderungen abgefragt und bei Änderungen spezielle Antwortprogramme ausgeführt.
  - Vorteile:
    - bei wenigen IO-Registern sehr kurze Latenzzeiten
    - bei einer unerwarteten Ereignisflut wird das Zeitverhalten des Programms nicht übermäßig beeinflusst
    - Kommunikation erfolgt synchron mit der Programmausführung
  - Nachteile:
    - die meisten Anfragen sind unnötig
    - hohe Prozessorbelastung
    - Reaktionszeit steigt mit der Anzahl an Ereignisquellen

Lösung: Interruptkonzept

- **Interrupt**: Ein Interrupt ist ein durch ein Ereignis ausgelöster, automatisch ablaufender Mechanismus, der die Verarbeitung des laufenden Programms unterbricht und die Wichtigkeit des Ereignisses überprüft. Darauf basierend erfolgt die Entscheidung, ob das bisherige Programm weiter bearbeitet wird oder eine andere Aktivität gestartet wird.
- Vorteile:
  - sehr geringe Extrabelastung der CPU
  - Prozessor wird nur dann beansprucht, wenn es nötig ist
- Nachteile:
  - Nicht-Determinismus: Unterbrechungen können zu einem beliebigen Zeitpunkt eintreffen.

Technische Realisierung

- Zur Realisierung besitzen Rechner einen oder mehrere spezielle Interrupt-Eingänge. Wird ein Interrupt aktiviert, so führt dies zur Ausführung der entsprechenden Unterbrechungsbehandlungs-routine (**interrupt handler, interrupt service routine (ISR)**).
- Das Auslösen der Unterbrechungsroutine ähnelt einem Unterprogrammaufruf. Der Programmablauf wird an einer anderen Stelle fortgesetzt und nach Beendigung der Routine an der unterbrochenen Stelle fortgefahren. Allerdings tritt die Unterbrechungsroutine im Gegensatz zum Unterprogrammsaufruf asynchron, also an beliebigen Zeitpunkten, auf.

Sperren von Interrupts

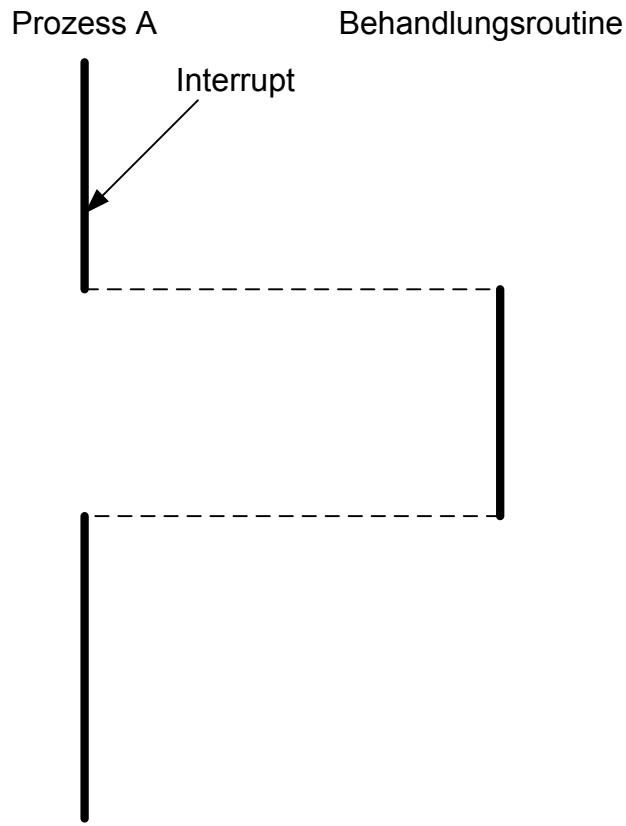
- Durch die Eigenschaft der Asynchronität kann eine deterministische Ausführung nicht gewährleistet werden. Aus diesem Grund kann eine kurzfristige Sperrung von Interrupts nötig sein, um eine konsistente Ausführung der Programme zu erlauben.
- Durch das Sperren werden Interrupts in der Regel nur verzögert, nicht jedoch gelöscht.

Interrupt Prioritäten

- Unterbrechungen besitzen unterschiedliche Prioritäten. Beim Auftreten einer Unterbrechung werden Unterbrechungen gleicher oder niedrigerer Priorität gesperrt.
- Tritt dagegen während der Ausführung der Behandlungsroutine eine erneute Unterbrechung mit höherer Priorität auf, so wird die Unterbrechungsbehandlung gestoppt und die Behandlungsroutine für die Unterbrechung mit höherer Priorität durchgeführt.

Ablauf einer Unterbrechung

1. Sperren von Unterbrechungen mit gleicher oder niedrigerer Priorität
2. Retten des Prozessorstatus
3. Bestimmen der Interruptquelle
4. Laden des Interruptvektors (Herstellung des Anfangszustandes für Behandlungsroutine)
5. Abarbeiten der Routine
6. Rückkehr zur Programmausführung (nicht unbedingt der unterbrochene Prozess)



Hardware Interrupts

- Nachfolgend wird eine typische Belegung (Quelle von 2002) der Interrupts angegeben:

00	Systemtaktgeber	08	Echtzeitsystemuhr
01	Tastatur	09	Frei
02	Programmierbarer Interrupt-Controller	10	Frei
03	Serielle Schnittstelle COM2 (E/A-Bereich 02F8)	11	Frei
04	Serielle Schnittstelle COM1 (E/A-Bereich 03F8)	12	PS/2-Mausanschluss
05	Frei, oft Soundkarte (Soundblaster-Emulation) oder LPT2	13	Koprozessor (ob separat oder in CPU integriert)
06	Diskettenlaufwerk	14	Primärer IDE-Kanal
07	Parallel (Drucker-) Schnittstelle LPT1 (E/A-Bereich 0378)	15	Sekundärer DIE-Kanal

Programmieren von Interrupts

- Implementierung der Unterbrechungsbehandlungsroutine

```
void interrupt yourisr() /* Interrupt Service
   Routine (ISR) */
{
   disable();
   /* Body of ISR goes here */
   oldhandler();
   outportb(0x20,0x20); /* Send EOI to PIC1 */
   enable();
}
```

Erläuterung

- `void interrupt yourisr`: Deklaration einer Interrupt Service Routine
- `disable()`: Ist eine weitere Unterbrechung von höher priorisierten Interrupts nicht gewünscht, so können auch diese gesperrt werden (Vorsicht bei der Verwendung).
- `oldhandler()`: Oftmals benutzen mehrere Programme einen Interrupt (z.B. die Uhr), in diesem Fall sollte man die bisherige ISR sichern (siehe nächste Folie) und an den neuen ISR anhängen
- `outportb()`: Dem PIC (Programmable Interrupt Controler) muss signalisiert werden, dass die Behandlung des Interrupts beendet ist.
- `enable`: Die Interrupt-Sperre muss aufgehoben werden.

Einfügen der Routine in Interrupt Vector Table

```
#include <dos.h>
#define INTNO 0x0B /* Interupt Number 3*/
void main(void)
{
   oldhandler = getvect(INTNO); /* Save Old Interrupt Vector */
   setvect(INTNO, yourisr); /* Set New Interrupt Vector Entry */
   outportb(0x21,(inportb(0x21) & 0xF7)); /*Un-Mask(Enable)IRQ3 */
   /* Set Card - Port to Generate Interrupts */
   /* Body of Program Goes Here */
   /* Reset Card - Port as to Stop Generating Interrupts */
   outportb(0x21,(inportb(0x21) | 0x08)); /*Mask (Disable) IRQ3 */
   setvect(INTNO, oldhandler); /*Restore old Vector Before Exit*/
}
```

Erläuterung

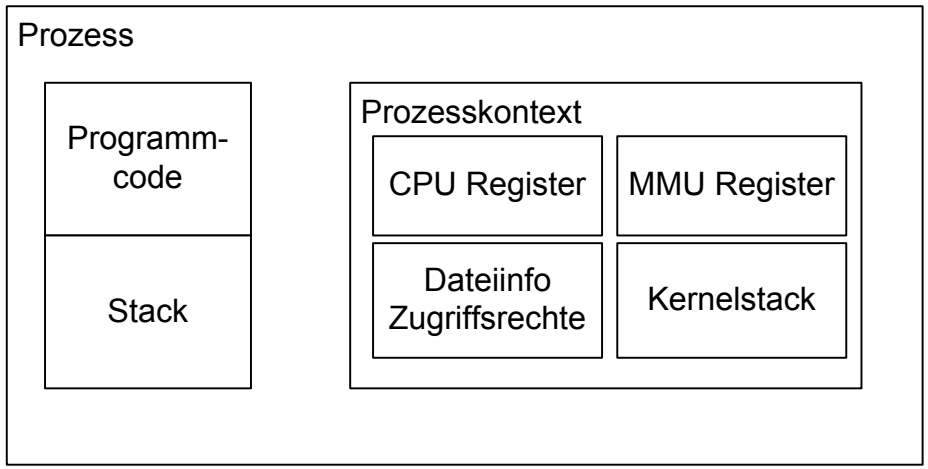
- Die Unterbrechungsvektortabelle enthält einen Verweis auf die entsprechende Unterbrechungsbehandlung für die einzelnen Unterbrechungen
- `INTNO`: Es soll der Hardwareinterrupt IRQ 3 (serielle Schnittstelle) verwendet werden, dieser Interrupt entspricht der Nummer 11 (insgesamt 255 Interrupts (vor allem Softwareinterrupts) vorhanden).
- `oldhandler=getvect(INTNO)`: Durch die Funktion `getvect()` kann die Adresse der Behandlungsfunktion zurückgelesen werden. Diese wird in der vorher angelegte
- `setvect`: setzen der neuen Routine
- `outportb`: setzen einer neuen Maskierung

Nebenläufigkeit

Prozesse

Definition

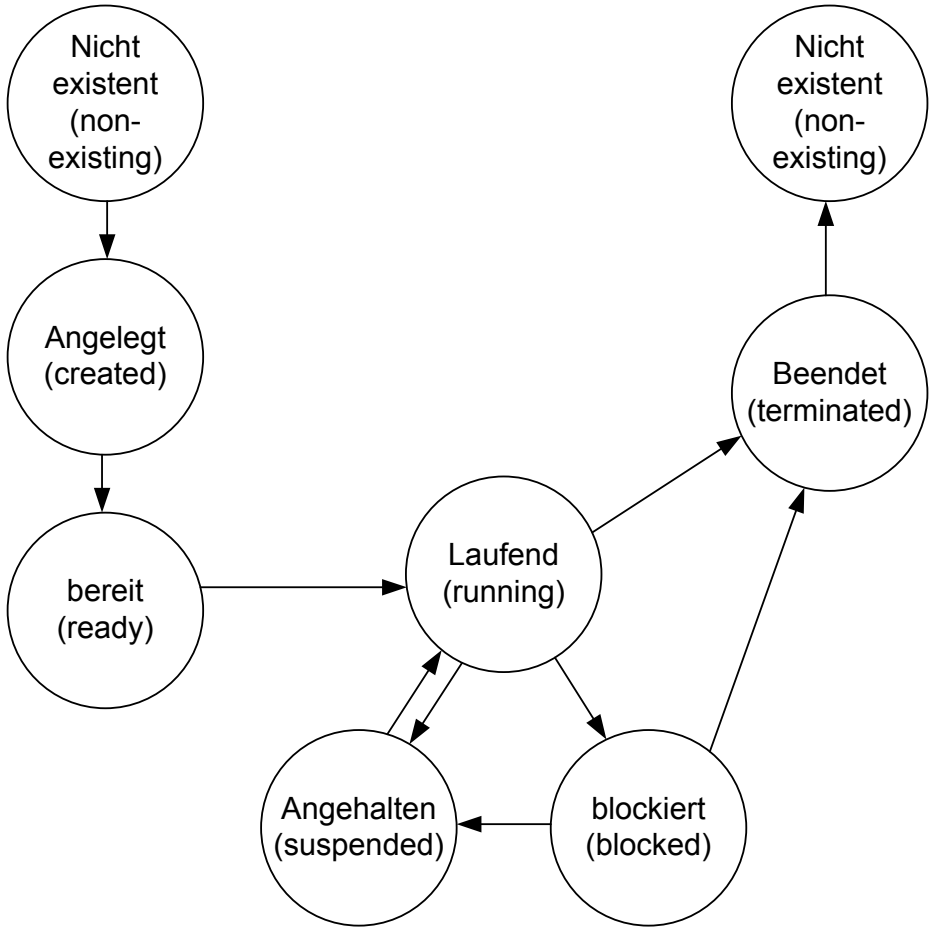
- **Prozess**: Abstraktion eines sich in Ausführung befindlichen Programms
- Die gesamte Zustandsinformation der Betriebsmittel für ein Programm wird als eine Einheit angesehen und als Prozess bezeichnet.
- Prozesse können weitere Prozesse erzeugen ⇨ Vater-,Kinderprozesse.



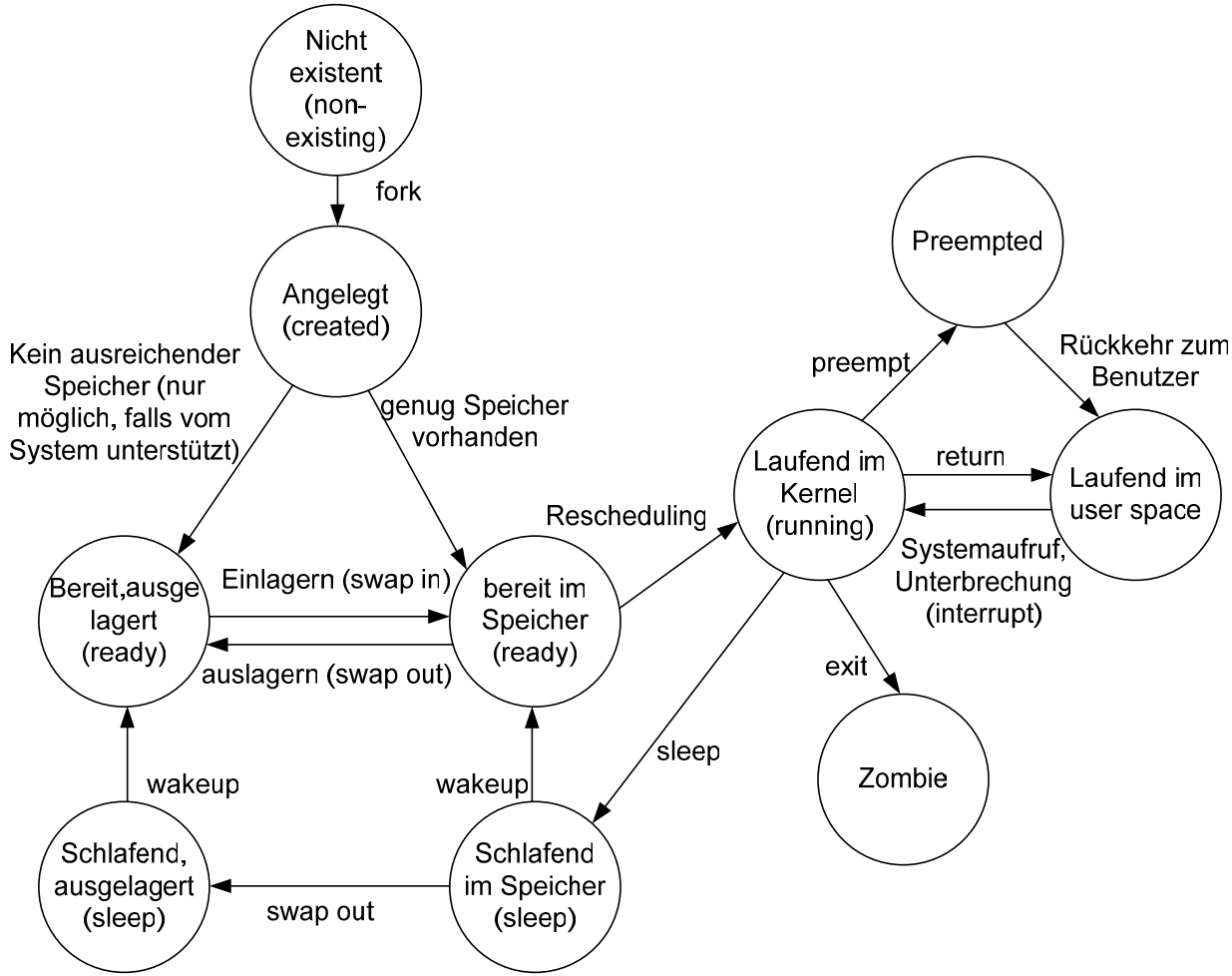
Prozessausführung

- Zur Prozessausführung werden diverse Resourcen benötigt, u.a.:
  - Prozessorzeit
  - Speicher
  - sonstige Betriebsmittel (z.B. spezielle Hardware)
- Die Ausführungszeit ist neben dem Programm abhängig von:
  - Leistungsfähigkeit des Prozessors
  - Verfügbarkeit der Betriebsmittel
  - Eingabeparametern
  - Verzögerungen durch andere (wichtigere) Aufgaben

Prozesszustände (allgemein)



Prozeßzustände in Unix



Fragen bei der Implementierung

- Welche Betriebsmittel sind notwendig?
- Welche Ausführungszeiten besitzen einzelne Prozesse?
- Wie können Prozesse kommunizieren?
- Wann soll welcher Prozess ausgeführt werden?
- Wie können Prozesse synchronisiert werden?

Klassifikation von Prozessen

- periodisch vs. aperiodisch
- statisch vs. dynamisch
- Wichtigkeit der Prozesse (kritisch, notwendig, nicht notwendig)
- speicherresident vs. verdrängbar
- Prozesse können auf
  - einem Rechner (Pseudoparallelismus)
  - einem Multiprozessorsystem mit Zugriff auf gemeinsamen Speicher
  - oder auf einem Multiprozessorsystem ohne gemeinsamen Speicher

  ausgeführt werden.

Nebenläufigkeit

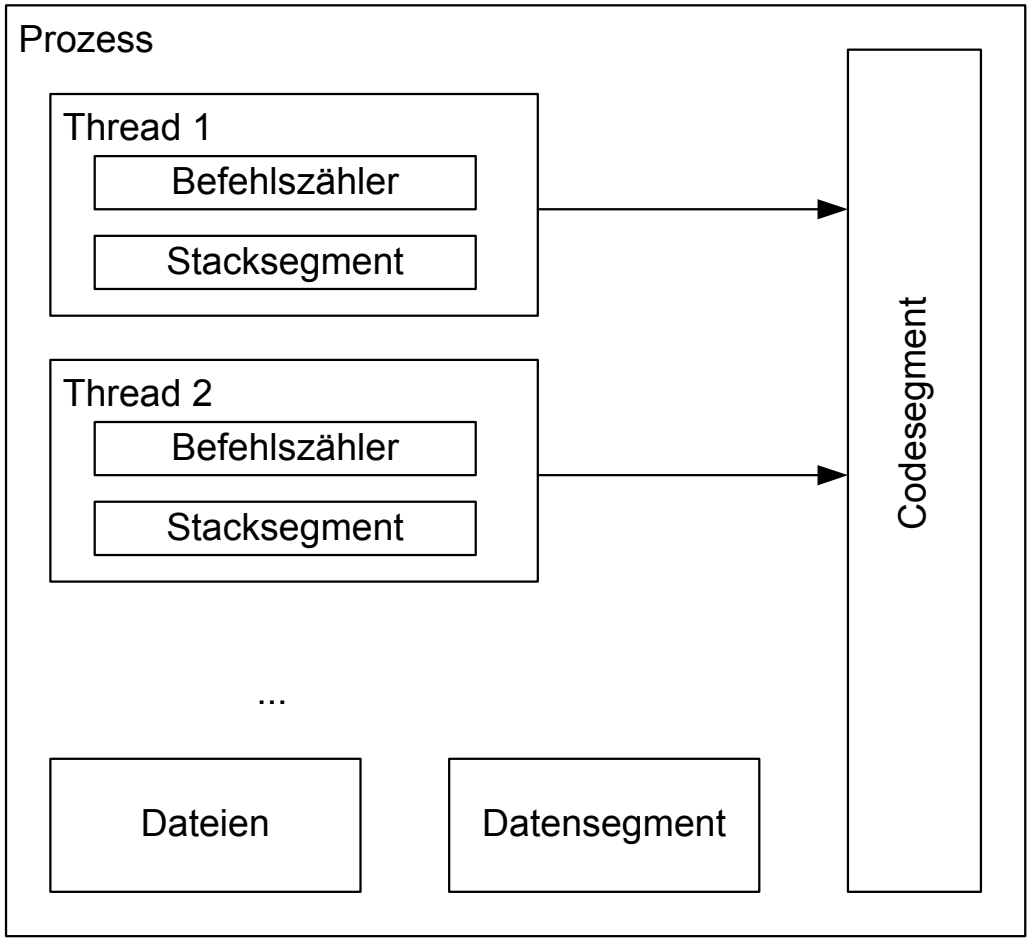
Threads

Leichtgewichtige Prozesse (Threads)

- Der Speicherbedarf von Prozessen ist in der Regel groß (CPU-Daten, Statusinformationen, Angaben zu Dateien und EA-Geräten...).
- Bei Prozesswechsel müssen die Prozessdaten ausgetauscht werden ⇒ hohe Systemlast, zeitaufwendig.
- Viele Systeme erfordern keine komplett neuen Prozesse.
- Vielmehr sind Programmabläufe nötig, die auf den gleichen Prozessdaten arbeiten.

⇨ Einführung von Threads

Threads


Prozess
Thread 1
Befehlszähler
Stacksegment
Thread 2
Befehlszähler
Stacksegment
...
Dateien
Datensegment
Codesegment

Prozesse vs. Threads

- Verwaltungsaufwand von Threads ist deutlich geringer
- Effizienzvorteil: bei einem Wechsel von Threads im gleichen Prozessraum ist kein vollständiger Austausch des Prozesskontextes notwendig.
- Kommunikation zwischen Threads des gleichen Prozesses kann über gemeinsamen Speicher erfolgen.
- Zugriffe auf den Speicherbereich anderer Prozesse führen zu Fehlern.
- Probleme bei Threads: durch die gemeinsame Nutzung von Daten kann es zu Konflikten kommen.

Nebenläufigkeit

Probleme

Probleme

- **Race Conditions**: Situationen, in denen zwei oder mehrere Threads/Prozesse, die gleichen geteilten Daten lesen oder schreiben und das Resultat davon abhängt, wann genau welcher Prozess ausgeführt wurde, werden Race Conditions genannt.
  - Lösung: Einführung von **kritischen Bereichen** und **wechselseitiger Ausschluss**.
- **Starvation** (**Aussperrung**): Situation, in der ein Prozess unendlich lange auf ein Betriebsmittel wartet. Wichtig: sinnvolle Realisierung von Warteschlangen bei der Betriebsmittelvergabe, z.B. Prioritätenbasierte Warteschlangen
- **Priority Inversion** (**Prioritätsinversion**): wichtige Prozesse können durch unwichtigere Prozesse, die Betriebsmittel belegt haben verzögert werden, genaue Problemstellung siehe Kapitel Scheduling

Bedingungen an Lösung für wechselseitigen Ausschluss

- An eine gute Lösung für den wechselseitigen Ausschluss (WA) können insgesamt vier Bedingungen gestellt werden:
  1. Es dürfen niemals zwei Prozesse gleichzeitig im kritischen Bereich sein.
  2. Es dürfen keine Annahmen über die Geschwindigkeit oder Anzahl der Prozessoren gemacht werden.
  3. Kein Prozess darf außerhalb von kritischen Regionen andere Prozesse blockieren.
  4. Kein Prozess soll unendlich auf das Eintreten in den kritischen Bereich warten müssen.

Kritische Bereiche

- Um einen solchen Bereich zu schützen, sind Mechanismen erforderlich, die ein gleichzeitiges Betreten verschiedener Prozesse bzw. Prozeßklassen dieser Bereiche verhindern.
  - Darf maximal nur ein Prozess gleichzeitig auf den kritischen Bereich zugreifen, so spricht man vom **wechselseitigen Ausschluss**.
  - Wird verhindert, daß mehrere (unterschiedlich viele) Instanzen unterschiedlicher Prozeßklassen auf den Bereich zugreifen, so entspricht dies dem Leser-Schreiber-Problem (so dürfen beispielsweise mehrere Instanzen der Klasse `Leser` auf den Bereich gleichzeitig zugreifen, Instanzen der Klasse `Schreiber` benötigen den exklusiven Zugriff).
- Aus dem Alltag sind diverse Mechanismen zum Schutz solcher Bereiche bekannt:
  - Signale im Bahnverkehr
  - Ampeln zum Schutz der Kreuzung
  - Schlösser für einzelne Räume
  - Vergabe von Tickets

Falsche Lösung: Verwendung einer globalen Variable

```
bool block = false; //global variable
```

```
...
while(block){}; //busy wait
block=true;
... critical section ...
block=false;
...
```

- Die obige Implementierung ist nicht korrekt,
  - da der Prozess direkt nach dem while-Abschnitt unterbrochen werden könnte und evtl. dann fortgesetzt wird, wenn block bereits durch einen anderen Prozess belegt ist.
  - Zudem ist die Lösung ineffizient (busy wait)

1.Möglichkeit: Peterson 1981 (Lösung für zwei Prozesse)

Deklaration globale Variablen

```
int turn=0;
boolean ready[2];
ready[0]=false;
ready[1]=false;
```

Prozess 0

```
...
ready[0]=true;
turn = 1;
while(ready[1]
    && turn==1);//busy waiting
... critical section ...
ready[0]=false;
...
```

Prozess 1

```
...
ready[1]=true;
turn = 0;
while(ready[0]
    && turn==0); //busy waiting
... critical section ...
ready[1]=false;
...
```

- Das Problem der Realisierung für n Prozesse ist unter dem Bakery Algorithmus bekannt.

2. Möglichkeit: Ausschalten von Unterbrechungen zum WA

- Prozesswechsel beruhen immer auf dem Eintreffen einer Unterbrechung (Interrupt) (z.B. neues Ereignis, Ablauf einer Zeitdauer)
- Die einfachste Möglichkeit einen Kontextwechsel zu verhindern ist das Ausschalten von Unterbrechungen während sich ein Prozess im kritischen Bereich befindet.
- Vorteile:
  - einfach zu implementieren, keine weiteren Konzepte sind nötig
  - schnelle Ausführung
- Nachteile:
  - Für Multiprozessorsysteme ungeeignet
  - Keine Gerätebehandlung während der Sperre
  - Lange Sperren kritisch bei Echtzeitanwendungen

3. Möglichkeit: Semaphor

- Semaphor (griechisch von Zeichenträger, Signalmast) wurden von Edsger W. Dijkstra im Jahr 1965 eingeführt.
- Ein Semaphor ist eine Datenstruktur, bestehend aus einer Zählvariablen, sowie den Funktionen `down()`oder `wait()` (bzw. `P()`, von probeer te verlagen) und `up()` oder `signal()` (bzw. `V()`, von verhogen).

```
Init(Semaphor s, Int v)
{
  s = v;
}

V(Semaphor s)
{
  s = s+1;
}

P(Semaphor s)
{
  while (s <= 0) {} ;  // Blockade, unterschiedliche Implementierungen
  s = s-1 ;            // sobald s>0 belege eine Ressource
}
```

- Bevor ein Prozess in den kritischen Bereich eintritt, muss er den Semaphor mit der Funktion `down()` anfordern. Nach Verlassen wird der Bereich durch die Funktion `up()` wieder freigegeben.
- **Wichtige Annahme**: die Ausführung der Funktionen von `up` und `down` darf nicht unterbrochen werden (atomare Ausführung), siehe Realisierung
- Solange der Bereich belegt ist (Wert des Semaphors <=0), wird der aufrufende Prozess blockiert.

Beispiel: Bankkonto

- Durch Verwendung eines gemeinsamen Semaphors `semAccount` kann das Bankkonto auch beim Zugriff von zwei Prozessen konsistent gehalten werden:

Prozess A

```
P(semAccount);
x=readAccount(account);
x=x+500;
writeAccount(x,account);
V(semAccount);
```

Prozess B

```
P(semAccount);
y=readAccount(account);
y=y-200;
writeAccount(y,account);
V(semAccount);
```

- Zur Realisierung des wechselseitigen Ausschlusses wird ein binärer Semaphor mit zwei Zuständen: 0 (belegt), 1 (frei) benötigt. Binäre Semaphore werden auch *Mutex* (von *mutal exclusion*) genannt.

Erweiterung: zählender Semaphore

- Nimmt ein Wert auch einen Wert größer eins an, so wird ein solch ein Semaphor auch als **zählender Semaphor** (counting semaphore) bezeichnet.
- Beispiel für den Einsatz von zählenden Semaphoren: In einem Leser-Schreiber-Problem kann die Anzahl der Leser aus Leistungsgründen z.B. auf 100 gleichzeitige Lesezugriffe beschränkt werden:

```
semaphore sem_reader_count;
init(sem_reader_count,100);
```

- Jeder Leseprozess führt dann folgenden Code aus:

```
P(sem_reader_count);
read();
V(sem_reader_count);
```

Realisierungen von Semaphoren

- Die Implementierung eines Semaphors erfordert spezielle Mechanismen auf Maschinenebene; der Semaphor ist für sich ein kritischer Bereich.
  $\Rightarrow$ Die Funktionen `up()` und `down()` dürfen nicht unterbrochen werden, da sonst der Semaphor selbst inkonsistent werden kann.
- Funktionen die nicht unterbrechbar sind, werden **atomar** genannt.
- Realisierungsmöglichkeiten:
  1. Kurzfristige Blockade der Prozeßwechsel während der Bearbeitung der Funktionen `up()` und `down()`. Implementierung durch Verwendung einer Interrupt-Sperre, denn sämtliche Prozesswechsel werden durch **Unterbrechungen** (**Interrupts**) ausgelöst.
  2. **Test&Set**-Maschinenbefehl: Die meisten Prozessoren verfügen heute über einen Befehl „**Test&Set**“ (oder auch Test&SetLock). Dieser lädt atomar den In-halt (typ. 0 für frei, 1 für belegt) eines Speicherwortes in ein Register und schreibt ununterbrechbar einen Wert (typ. $\neq$ 0, z.B. 1 für belegt) in das Speicherwort.
  3. **Spinlock**: Programmiertechnik auf der Basis von Busy Waiting. Vorteil: Unabhängig vom Betriebssystem und auch in Mehrprozessorsystemen zu implementieren, jedoch massive Verschwendung von Rechenzeit. Im Gegensatz dazu können die Lösungen von 1 und 2 mit Hilfe von Warteschlangen sehr effizient realisiert werden.

Verbessertes Konzept: Monitore

- Ein Nachteil von Semaphoren ist die Notwendigkeit zur expliziten Anforderung und Freigabe des kritischen Bereiches durch den Programmierer
- Vergißt der Entwickler z.B. die Freigabe des Semaphors nach dem Durchlaufen des kritischen Abschnitts, dann kann es schnell zu einer Verklemmung kommen; solche Fehler sind sehr schwer zu finden!
- Zum einfacheren und damit weniger fehlerträchtigen Umgang mit kritischen Bereichen wurde deshalb das Konzept der *Monitore* (Hoare 1974, Brinch Hansen 1975) entwickelt:
  - Ein **Monitor** ist eine Einheit von Daten und Prozeduren auf diesen Daten, auf die zu jeden Zeitpunkt nur maximal ein Prozeß zugreifen kann. Oder: Gemeinsames Objekt, in dem jede Methode einen kritischen Abschnitt darstellt.
  - Wollen mehrere Prozesse gleichzeitig auf einen Monitor zugreifen, so werden alle Prozesse bis auf einen Prozess in eine Warteschlange eingereiht und blockiert.
  - Verlässt ein Prozess den Monitor, so wird ein Prozess aus der Warteschlange entnommen und dieser kann auf die Funktionen und Daten des Monitors zugreifen.
  - Signalisierung innerhalb des Monitors festgelegt, dies braucht dem Programmierer nicht zu kümmern

Beispiel: Monitore in Java

- In Java werden Monitore durch `synchronized`-Methoden implementiert. Zu jedem Zeitpunkt darf nur ein Prozess sich **aktiv** in einer dieser Methoden befinden.
- **Anmerkung**: normalerweise werden höhere Konstrukte wie Monitore durch einfachere Konstrukte wie den Semaphor implementiert.
  Siehe auch die Realisierung von Semaphoren durch das einfachere Konzept TSL-Befehl.
- In Java kann man das Monitorkonzept allerdings auch nutzen um selber Semaphore zu implementieren (siehe nebenstehenden Code).
- `wait()` und `notify()` sind zu jedem Objekt in Java definierte Methoden.

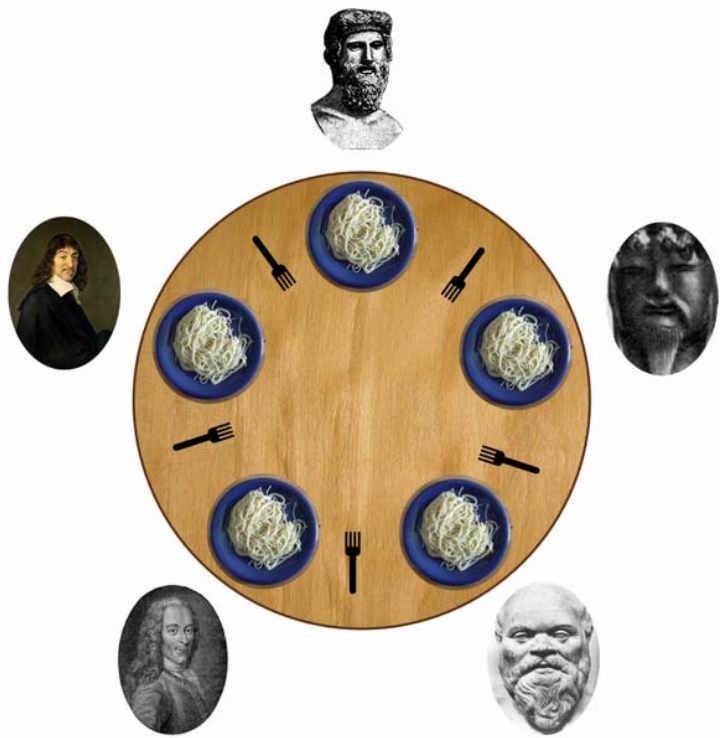
```
public class Semaphore {
    private int value;

    public Semaphore (int initial) {
        value = initial;
    }

    synchronized public void up() {
        value++;
        if(value==1) notify();
    }

    synchronized public void down() {
        while(value==0) wait();
        value- -;
    }
}
```

Bemerkung zu Verklemmungen / Deadlocks

- Auch bei der korrekten Verwendung von Semaphoren und Monitoren kann es zu Deadlocks kommen, siehe Beispiel auf der folgenden Seite.
- Coffman, Elphick und Shoshani haben 1971 die vier konjunktiv notwendigen Voraussetzungen für einen Deadlock formuliert:
  1. Wechselseitiger Ausschluss: Es gibt eine Menge von exklusiven Ressourcen R_{exkl}, die entweder frei sind oder genau einem Prozess zugeordnet sind.
  2. Hold-and-wait-Bedingung: Prozesse, die bereits im Besitz von Ressourcen aus R_{exkl} sind, fordern weitere Ressourcen aus R_{exkl} an.
  3. Ununterbrechbarkeit: Die Ressourcen R_{exkl} können einem Prozess nicht entzogen werden, sobald er sie belegt. Sie müssen durch den Prozess explizit freigegeben werden.
  4. Zyklische Wartebedingung: Es muss eine zyklische Kette von Prozessen geben, die jeweils auf Ressourcen warten, die dem nächsten Prozess in der Kette gehören.
- Umgekehrt (und positiv) formuliert: ist eine der Bedingungen nicht erfüllt, so sind Verklemmungen ausgeschlossen.

Klassisches Beispiel: Speisende Philosophen

- Klassisches Beispiel aus der Informatik für Verklemmungen: "Dining Philosophers" (speisende Philosophen, Dijkstra 1971, Hoare 1971)
- 5 Philosophen (Prozesse) sitzen an einem Tisch. Vor ihnen steht jeweils ein Teller mit Essen. Zum Essen benötigen sie zwei Gabeln (Betriebsmittel), insgesamt sind aber nur 5 Gabeln verfügbar.
- Die Philosophen denken und diskutieren. Ist einer hungrig, so greift er zunächst zur linken und dann zur rechten Gabel. Ist eine Gabel nicht an ihrem Platz, so wartet er bis die Gabel wieder verfügbar ist (ohne eine evtl. in der Hand befindliche Gabel zurückzulegen). Nach dem Essen legt er die Gabeln zurück.
- Problem: sind alle Philosophen gleichzeitig hungrig, so nehmen sie alle ihre linke Gabel und gleichzeitig ihrem Nachbarn die rechte Gabel weg. Alle Philosophen warten auf die rechte Gabel und es entsteht eine Verklemmung (deadlock).
- Gibt ein Philosoph seine Gabel nicht mehr zurück, so stirbt der entsprechende Nachbar den **Hungertod (starvation)**.



Nebenläufigkeit

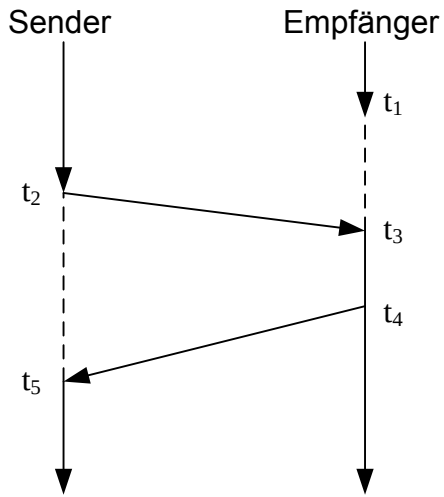
Interprozesskommunikation (IPC)

Interprozesskommunikation

- Notwendigkeit der Interprozesskommunikation
  - Prozesse arbeiten in unterschiedlichen Prozessräumen oder sogar auf unterschiedlichen Prozessoren.
  - Prozesse benötigen evtl. Ergebnisse von anderen Prozessen.
  - Zur Realisierung von wechselseitigen Ausschlüssen werden Mechanismen zur Signalisierung benötigt.
- Klassifikation der Kommunikation
  - synchrone vs. asynchrone Kommunikation
  - pure Ereignisse vs. wertbehaftete Nachrichten

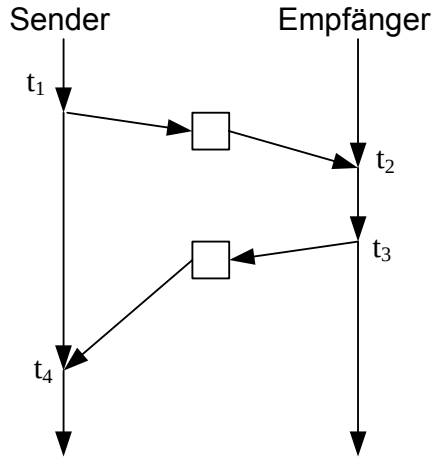
Synchron vs. Asynchron

Synchrone Kommunikation



t_1 : Empfänger wartet auf Nachricht
t_2 : Sender schickt Nachricht und blockiert
t_3 : Empfänger bekommt Nachricht, die Verarbeitung startet
t_4 : Verarbeitung beendet, Antwort wird gesendet
t_5 : Sender empfängt Nachricht und arbeitet weiter

Asynchrone Kommunikation



t_1 : Sender schickt Nachricht an Zwischenspeicher und arbeitet weiter
t_2 : Empfänger liest Nachricht
t_3 : Empfänger schreibt Ergebnis in Zwischenspeicher
t_4 : Sender liest Ergebnis aus Zwischenspeicher

(Nicht eingezeichnet: zusätzliche Abfragen des Zwischenspeichers und evtl. Warten)

IPC-Mechanismen

- Übermittlung von Datenströmen:
  - direkter Datenaustausch
  - Pipes
  - Nachrichtenwarteschlangen (Message Queues)
- Signalisierung von Ereignissen:
  - Signale
  - Semaphore
- Synchrone Kommunikation
  - Barrieren/Rendezvous
  - Kanäle wie z.B. Occam
- Funktionsaufrufe:
  - RPC
  - Corba

Nebenläufigkeit

IPC: Kommunikation durch Datenströme

Direkter Datenaustausch

- Mit Semaphoren und Monitoren geschützte Datenstrukturen eignen sich sehr gut für den Austausch von Daten:
  - schnelle Kommunikation, da auf den Speicher direkt zugegriffen werden kann.
- Allerdings kann die Kommunikation nur lokal erfolgen und zudem müssen die Prozesse eng miteinander verknüpft sein.
- Programmiersprachen, Betriebssysteme, sowie Middlewareansätze bieten komfortablere Methoden zum Datenaustausch.
- Grundsätzlich erfolgt der Austausch über das Ausführen von Funktionen `send(receiver address, &message)` und `receive(sender address, &message)`.

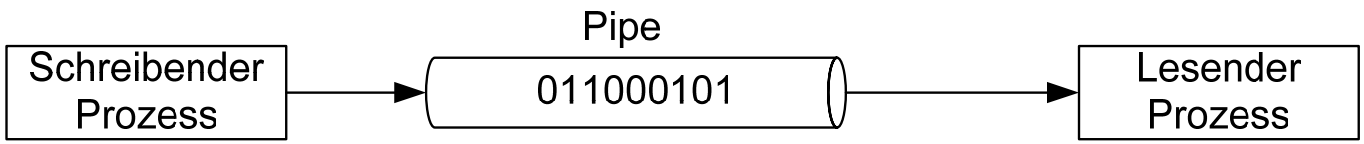
Fragestellungen beim Datenaustausch

- Nachrichtenbasiert oder Datenstrom?
- Lokale oder verteilte Kommunikation?
- Kommunikationsparameter:
  - mit/ohne Bestätigung
  - Nachrichtenverluste
  - Zeitintervalle
  - Reihenfolge der Nachrichten
- Adressierung
- Authentifizierung
- Performance
- Sicherheit (Verschlüsselung)

Heute: vor allem lokale Kommunikation, echtzeitfähige Protokolle zur Kommunikation in eigenem Kapitel

Pipes

- Die Pipe bezeichnet eine gepufferte, unidirektionale Datenverbindung zwischen zwei Prozessen nach dem **First-In-First-Out-** (**FIFO**-)Prinzip.
- Über den Namen der Pipe (ähnlich einem Dateinamen) können Prozesse unterschiedlichen Ursprungs auf eine Pipe lesend oder schreibend zugreifen. Zur Kommunikation zwischen Prozessen gleichen Ursprungs (z.B. Vater-, Kindprozess) können auch anonyme Pipes verwendet werden. Die Kommunikation erfolgt immer asynchron.



Pipes in Posix

- POSIX (Portable Operating System Interface) versucht durch Standardisierung der Systemaufrufe die Portierung von Programmen zwischen verschiedenen Betriebssystemen zu erleichtern.
- POSIX.1 definiert folgende Funktionen für Pipes:

```
int mkfifo(char* name, int mode);       /*Erzeugen einer benannten Pipe*/
int unlink ( char *name );              /*Loeschen einer benannten Pipe*/
int open ( char *name, int flags);      /*Oeffnen einer benannten Pipe*/
int close ( int fd );                   /*Schliessen des Lese- oder Schreibendes einer
                                          Pipe*/
int read ( int fd, char *outbuf, unsigned bytes ); /*Lesen von einer Pipe*/
int write ( int fd, char *outbuf,unsigned bytes ); /*Schreiben an eine Pipe*/
int pipe ( int fd[2] );                 /*Erzeugen eine unbenannte Pipe*/
```

Nachteile von Pipes

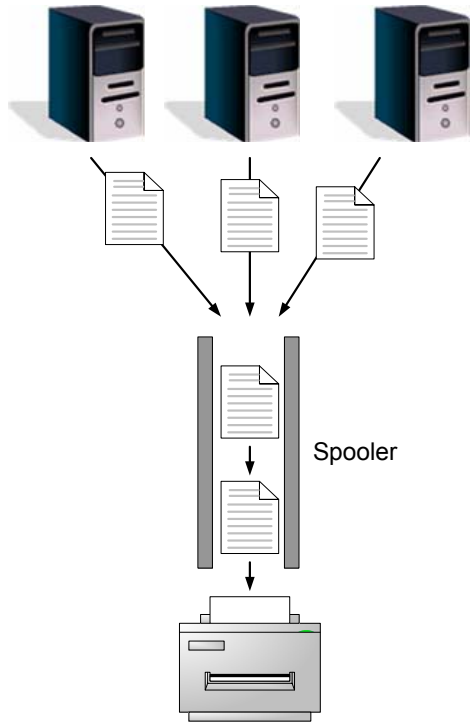
- Pipes bringen einige Nachteile mit sich:
  - Pipes sind nicht nachrichtenorientiert (keine Bündelung der Daten in einzelne Pakete (Nachrichten) möglich).
  - Daten sind nicht priorisierbar.
  - Der für die Pipe notwendige Speicherplatz wird erst während der Benutzung angelegt.
- Wichtig für die Implementierung:
  - Es können keine Daten aufgehoben werden.
  - Beim Öffnen blockiert der Funktionsaufruf, bis auch das zweite Zugriffsende geöffnet wird (Verhinderung durch O_NDELAY Flag).
- Lösung: Nachrichtenwartschlangen

Nachrichtenwarteschlangen (message queues)

- Nachrichtenschlangen (Message Queues) sind eine Erweiterung von Pipes. Im Folgenden werden Nachrichtenwarteschlangen, wie in POSIX 1003.1b (Echtzeiterweiterung von POSIX) definiert, betrachtet.
- Eigenschaften der POSIX MessageQueues:
  - Beim Anlegen einer MessageQueue wird der benötigte Speicher reserviert. ⇒ Speicher muss nicht erst beim Schreibzugriff angelegt werden.
  - Die Kommunikation erfolgt nachrichtenorientiert. Die Anzahl der vorhandenen Nachrichten kann dadurch abgefragt werden.
  - Nachrichten sind priorisierbar → Es können leichter Zeitgarantien gegeben werden.

Nachrichtenwarteschlangen

- Schreibzugriff in Standardsystemen: Der schreibende/sendende Prozess wird nur dann blockiert werden, falls der Speicher der Datenstruktur bereits voll ist. **Alternative in Echtzeitsystemen: Fehlermeldung ohne Blockade**.
- Lesezugriff in Standardsystemen: Beim lesenden/empfangenden Zugriff auf einen leeren Nachrichtenspeicher wird der aufrufende Prozess blockiert bis eine neue Nachricht eintrifft. **Alternative: Fehlermeldung ohne Blockade**.
- Ein anschauliches Beispiel für den Einsatzbereich ist der Spooler eines Druckers: dieser nimmt die Druckaufträge der verschiedenen Prozesse an und leitet diese der Reihe nach an den Drucker weiter.



Message Queues in POSIX

- POSIX definiert folgende Funktionen für Nachrichtenwartschlangen:

```
mqd_t mq_open(const char *name, int oflag, ...);  /*Oeffnen einer Message Queue*/
int mq_close(mqd_t mqdes);                        /*Schliessen einer Message Queue*/
int mq_unlink(const char *name);                  /*Loeschen einer
                                                    Nachrichtenwarteschlange*/
int mq_send(mqd_t mqdes, const char *msg_ptr,
    size_t msg_len, unsigned int msg_prio);       /*Senden einer Nachricht*/
size_t mq_receive(mqd_t mqdes, char *msg_ptr,
    size_t msg_len, unsigned int *msg_prio);      /*Empfangen einer Nachricht*/
int mq_setattr(mqd_t mqdes, const struct
    mq_attr *mqstat, struct mq_attr *mqstat);     /*Aendern der Attribute*/
int mq_getattr(mqd_t mqdes,
    struct mq_attr *mqstat);                      /*Abrufen der aktuellen
                                                    Eigenschaften*/
int mq_notify(mqd_t mqdes,
    const struct sigevent *notification);         /*Anforderung eines Signals bei
                                                    Nachrichtenankunft*/
```

Nebenläufigkeit

IPC: Kommunikation durch Ereignisse

Signale

- **Signale** werden in Betriebssystemen typischerweise zur Signalisierung von Ereignissen an Prozessen verwendet.
- Signale können verschiedene Ursachen haben:
  - Ausnahmen, z.B. Division durch Null (SIGFPE) oder ein Speicherzugriffsfehler (SIGSEGV)
  - Reaktion auf Benutzereingaben (z.B. Ctrl / C)
  - Signal von anderem Prozess zur Kommunikation
  - Signalisierung von Ereignissen durch das Betriebssystem, z.B. Ablauf einer Uhr, Beendigung einer asynchronen I/O-Funktion, Nachrichtankunft an leerer Nachrichtenwarteschlange (siehe mq_notify())

Prozessreaktionen auf Signale

- Der Prozess hat drei Möglichkeiten auf Signale zu reagieren:
  1. Ignorierung der Signale
  2. Ausführen einer Signalbehandlungsfunktion
  3. Verzögerung des Signals, bis Prozess bereit für Reaktion ist
- Zudem besteht die Möglichkeit mit der Standardreaktion auf das bestimmte Signal zu reagieren. Da aber typischerweise die Reaktion auf Signale die Beendigung des Empfängerprozesses ist, sollte ein Programm über eine vernünftige Signalbehandlung verfügen, sobald ein Auftreten von Signalen wahrscheinlich wird.

POSIX Funktionen für Signale

- POSIX 1003.1 definiert folgende Funktionen:

Funktion	Bedeutung
kill	Senden eines Signals an einen Prozess oder eine Prozessgruppe
sigaction	Spezifikation der Funktion zur Behandlung eines Signals
sigaddset	Hinzufügen eines Signals zu einer Signalmenge
sigdelset	Entfernen eines Signals von einer Signalmenge
sigemptyset	Initialisierung einer leeren Signalmenge
sigfillset	Initialisierung einer kompletten Signalmenge
sigismember	Test, ob ein Signal in einer Menge enthalten ist
sigpending	Rückgabe der aktuell angekommenen, aber verzögerten Signale
sigprocmask	Setzen der Menge der vom Prozess blockierten Signale
sigsuspend	Änderung der Liste der blockierten Signale und Warten auf Ankunft und Behandlung eines Signals

Einschränkungen der Standardsignale

- POSIX 1003.1 Signale haben folgende Einschränkungen:
  - Es existieren zu wenige Benutzersignale (`SIGUSR1` und `SIGUSR2`)
  - Signale besitzen keine Prioritäten
  - Blockierte Signale können verloren gehen (beim Auftreten mehrerer gleicher Signale)
  - Das Signal enthält keinerlei Informationen zur Unterscheidung von anderen Signalen gleichen Typs (z.B. Absender)

Erweiterungen in POSIX 1003.1b

- Zur Benutzung von Echtzeitsystemen sind in POSIX 1003.1b
- folgende Erweiterungen vorgenommen worden:
  - Einen Menge von nach Priorität geordneten Signalen, die Benutzern zur Verfügung stehen (Bereich von `SIGRTMIN` bis `SIGRTMAX`)
  - Einen Warteschlangenmechanismus zum Schutz vor Signalverlust
  - Mechanismen zur Übertragung von weiteren Informationen
  - schnellere Signallieferung beim Ablauf eines Timers, bei Ankunft einer Nachricht an einer leeren Nachrichtenwarteschlange, bei Beendigung einer I/O-Operation
  - Funktionen, die eine schnellere Reaktion auf Signale erlauben

POSIX Funktionen für Signale

- POSIX 1003.1b definiert folgende zusätzliche Funktionen:

Funktion	Bedeutung
sigqueue	Sendet ein Signal inklusive identifizierende Botschaften an Prozess
sigtimedwait	Wartet auf ein Signal für eine bestimmte Zeitdauer, wird ein Signal empfangen, so wird es mitsamt der Signalinformation zurückgeliefert
sigwaitinfo	Wartet auf ein Signal und liefert das Signal mitsamt Information zurück

Beispiel: Programmierung von Signalen

- Im Folgenden wird der Code für ein einfaches Beispiel dargestellt: die periodische Ausführung einer Funktion.
- Der Code besteht aus folgenden Codeabschnitten:
  - Initialisierung eines Timers und der Signale
  - Setzen eines periodischen Timers
  - Wiederholtes Warten auf den Ablauf des Timers
  - Löschen des Timers
  - Hauptfunktion

Beispiel: Programmierung von Signalen I

```
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <signal.h>

int main ()
{
    int i=0;
    int test;
    struct timespec current_time;
    struct sigevent se;
    sigset_t set;   /* our signal set */
    timer_t timerid; /* timerid of our timer */
    struct itimerspec timer_sett; /* timer settings */

    timer_sett.it_interval.tv_sec = 0;          /* periodic interval length s */
    timer_sett.it_interval.tv_nsec = 500000000; /* periodic interval length ns */
    timer_sett.it_value.tv_sec = 0;             /* timer start time s */
    timer_sett.it_value.tv_nsec = 500000000; /* timer start time ns */
```

Beispiel: Programmierung von Signalen II

```
    se.sigev_notify = SIGEV_SIGNAL;  /* timer should send signals */
    se.sigev_signo = SIGUSR1;     /* timer sends signal SIGUSR1 */

    sigemptyset(&set); /* initialize signal set */
    sigaddset(&set, SIGUSR1); /* add signal which will be catched */
    timer_create(CLOCK_REALTIME, &se, &timerid); /* create timer */
    timer_settime(timerid, 0, &timer_sett, NULL);  /* set time settings for timer */

    for (i = 0; i < 5; i++)
    {
        sigwait(&set,&test); /* wait for signal defined in signal set */
        clock_gettime(CLOCK_REALTIME, &current_time); /* retrieve startup time */
        printf("Hello\n");
    }

    timer_delete(timerid); /* delete timer */
    return 0;
}
```

Semaphore zur Vermittlung von Ereignissen

- Semaphore können neben der Anwendung des wechselseitigen Ausschlusses auch zur Signalisierung von Ereignissen verwendet werden.
- Es ist zulässig, dass Prozesse (Erzeuger) Semaphore andauernd freigeben und andere Prozesse (Verbraucher) Semaphore dauern konsumieren.
- Es können auch benannte Semaphoren erzeugt werden, die dann über Prozessgrenzen hinweg verwendet werden können.
- Notwendige Funktionen sind dann:
  - sem_open(): zum Erzeugen und / oder Öffnen eines benannten Semaphors
  - sem_unlink(): zum Löschen eines benannten Semaphors

Signalisierung durch Semaphore: Beispiel

- Beispiel: ein Prozeß **Worker** wartet auf einen Auftrag (abgespeichert z.B. in einem char-Array job) durch einen Prozess **Contractor**, bearbeitet diesen und wartet im Anschluß auf den nächsten Auftrag:

Worker:

```
while(true)
{
        down(sem); /*wait for
                   next job*/
        execute(job);
}
```

Contractor:

```
...
job=...  /*create new job and save
         address in global variable*/
up(sem); /*signal new job*/
...
```

Probleme

- Problematisch an der Implementierung des Beispiels auf der letzten Folie ist, dass der Zeiger auf den Auftrag `job` nicht geschützt ist und es so zu fehlerhaften Ausführungen kommen kann.
- Durch Verwendung eines zusätzlichen Semaphors kann dieses Problem behoben werden.
- Ist die Zeit zwischen zwei Aufträgen zu kurz um die rechtzeitige Bearbeitung sicherzustellen, so kann es zu weiteren Problemen kommen:
  - Problem 1: Der Prozess **Contractor** muss warten, weil der Prozeß **Worker** den letzten Auftrag noch bearbeitet.
  - Problem 2: Der letzte Auftrag wird überschrieben, falls dieser noch gar nicht bearbeitet wurde. Abhängig von der Implementierung des Semaphors könnte dann der neue Auftrag zudem zweifach ausgeführt werden.

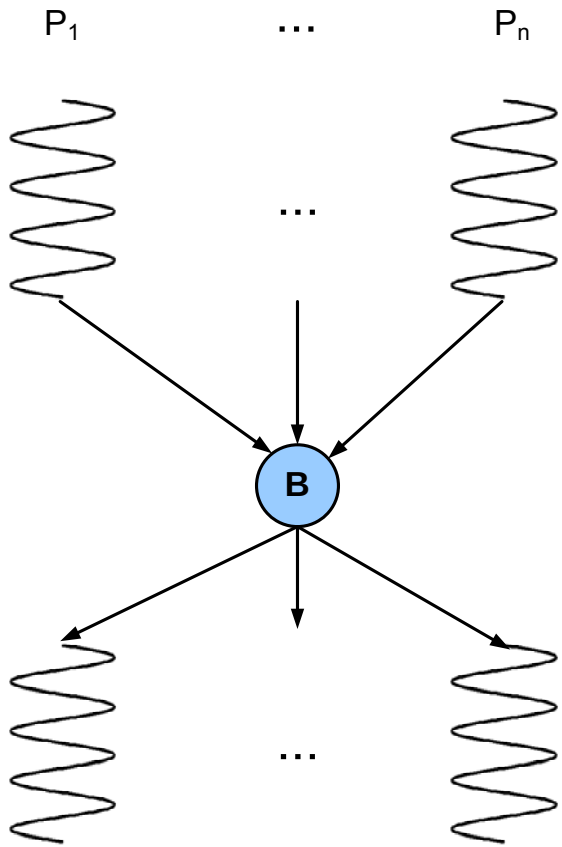
→ mit Semaphoren sind nur einfache Signalisierungsprobleme (ohne Datentransfer) zu lösen, ansonsten sollten Warteschlangen verwendet werden

Nebenläufigkeit

Synchrone Kommunikation: Barrieren, Occam

Synchrone Kommunikation: Barrieren

- **Definition**: Eine Barriere für eine Menge *M* von Prozessen ist ein Punkt, den alle Prozesse $P_i \in M$ erreichen müssen, bevor irgendein Prozess aus *M* die Berechnung über diesen Punkt hinaus fortfahren kann.
- Der Spezialfall für $|M|=2$ wird als Rendezvous, siehe auch Ada, bezeichnet.
- Barrieren können mit Hilfe von Semaphoren implementiert werden.



Occam

- Als Programmiersprache wurde Occam verwendet, mit der man parallel Abläufe festlegen konnte.
- Als Namenspate fungierte der Philosoph William of Ockham. Sein Postulat „*Dinge sollten nicht komplizierter als unbedingt notwendig gemacht werden*“ war Motto der Entwicklung.
- Occam basiert auf dem Modell CSP (communicating sequential processes) von C.A.R. Hoare; siehe auch CCS (Calculus of Communicating Systems) von R. Milner
- Occam ist eine Sprache, die die parallele Ausführung von Aktionen direkt mit einbezieht
- Die Kommunikation zwischen den einzelnen Prozessen erfolgt synchron über unidirektionale Kanäle.
- Die Realisierung auf dem Transputer ist 1:1. Als Kanal zwischen zwei Prozessen auf unterschiedlichen Transputern kann ein (halber) Link benutzt werden. Befinden sich die beiden Prozesse auf einem Transputer, so kann der Kanal über Speicherplätze simuliert werden.
- Siehe http://vl.fmnet.info/occam/

William of Ockham

C.A.R. Hoare

Occam

- Code wird in Occam zu Blöcken zusammengefasst, indem die einzelnen Zeilen alle gleichweit eingerückt werden
- Eine Anweisung wird durch das Ende der Zeile beendet
- Sprachelemente:
  - Eingabe ? :
  - Ausgabe !:
  - Sequentielle Ausführung SEQ:
  - Parallele Ausführung PAR:
  - Alternative Ausführung ALT*:

```
keyboard ? c
```

```
screen ! c
```

```
SEQ
    x:=1
    y:=2
```

```
PAR
    keyboard ? x
    screen ! y
```

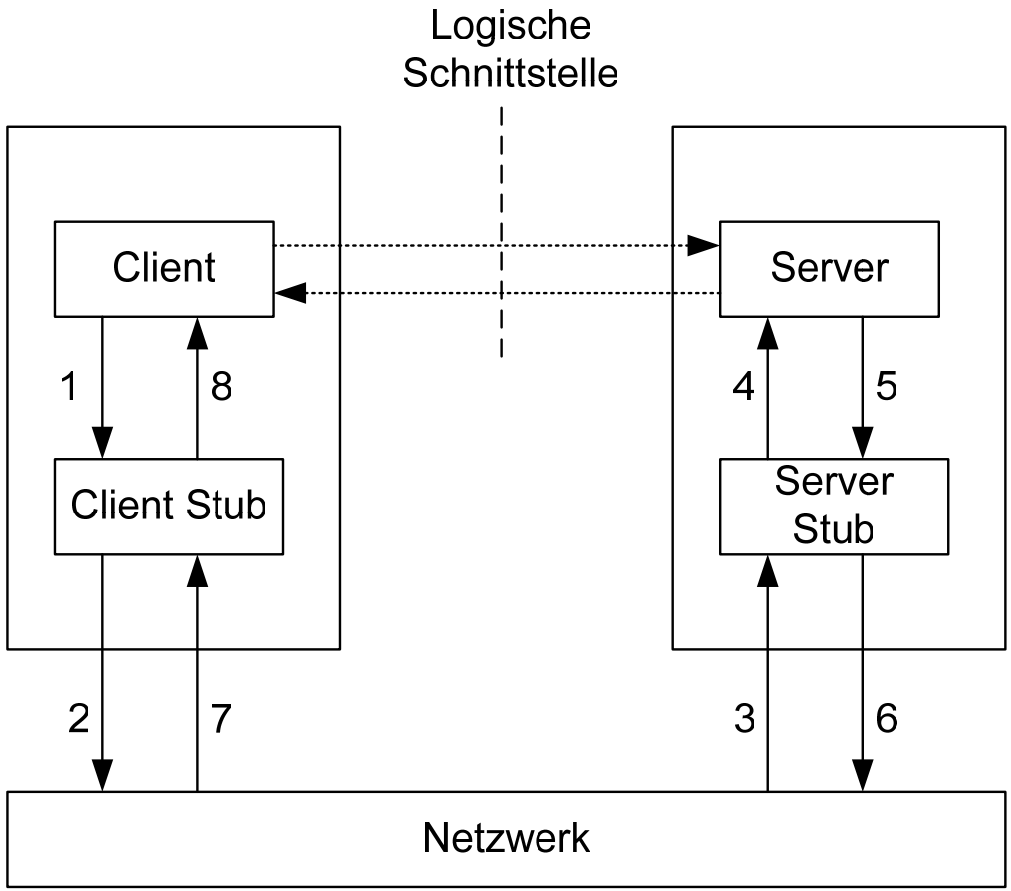
```
ALT
    x<10 & chan1 ? y
        screen ! y
    x<20 & chan2 ? y
        screen ! y
```

*Bei der ALT kann für jeden Block eine Bedingung, sowie eine Eingabe (beide optional) angegeben werden. Es wird derjenige Block ausgeführt, dessen Bedingung wahr ist und auf dem Daten eingehen. Trifft dies für mehrere Blöcke zu, so wird ein Block gewählt und ausgeführt.

Nebenläufigkeit

Funktionsaufrufe als Kommunikation

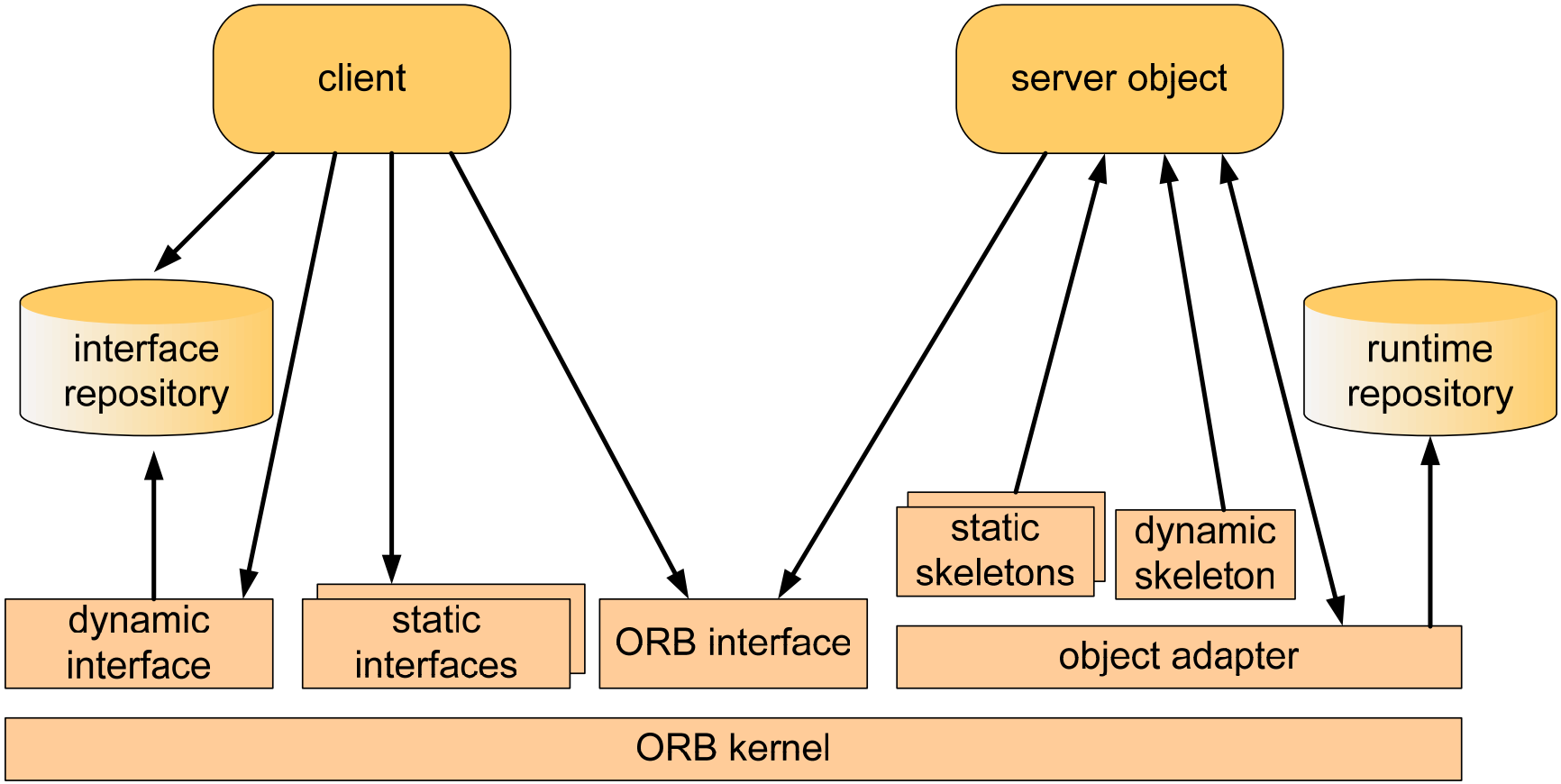
Remote Procedure Call (RPC)



Ablauf RPC

- Bei einem Funktionsaufruf über RPC werden folgende Schritte ausgeführt:
  1. Lokaler Funktionsaufruf vom Client an Client Stub
  2. Konvertierung des Funktionsaufrufs in Übertragungsformat und Senden der Nachricht
  3. Empfang der Nachricht von Kommunikationschicht
  4. Entpacken der Nachricht und lokaler Funktionsaufruf
  5. Übermittlung des Ergebnisses von Server an Server Stub
  6. Konvertierung des Funktionsergebnisses in Übertragungsformat und Senden der Nachricht
  7. Empfang der Nachricht von Kommunikationschicht
  8. Entpacken der Nachricht und Übermittlung des Ergebnisses an Client
- Voraussetzung für Echtzeitfähigkeit: Echtzeitfähiges Kommunikationsprotokoll und Mechanismus zum Umgang mit Nachrichtenverlust

Corba (Common Object Request Broker Architecture)



Komponenten in Corba

- **ORB (Object Request Broker)**: vermittelt Anfragen zwischen Server und Client, managt die Übertragung, mittlerweile sind auch echtzeitfähige ORBs verfügbar
- **ORB Interface**: Schnittstelle für Systemdienstaufrufe
- **Interface repository**: speichert die Signaturen der zur Verfügung stehenden Schnittstellen, die Schnittstellen werden dabei in der IDL-Notation (Interface Definition Language) gespeichert.
- **Object Adapter**: Überbrückt die Lücke zwischen Corba-Objekten mit IDL-Schnittstelle und Serverobjekten in der jeweiligen Programmiersprache
- **Runtime repository**: enthält die verfügbaren Dienste und die bereit instantiierten Objekte mitsamt den entsprechenden IDs
- **Skeletons**: enthalten die Stubs für die Serverobjektaufrufe

Nebenläufigkeit

Zusammenfassung

Zusammenfassung

- Folgende Fragen wurden in dieser Vorlesung erklärt und sollten nun verstanden sein:
  - Was ist Nebenläufigkeit / Parallelität?
  - Mit welchen Techniken kann man Nebenläufigkeit erreichen und wann wird welche Technik angewendet?
  - Wie können **race conditions** vermieden werden?
  - Welche Arten der Interprozesskommunikation gibt es (+allgemeine Erklärung)?

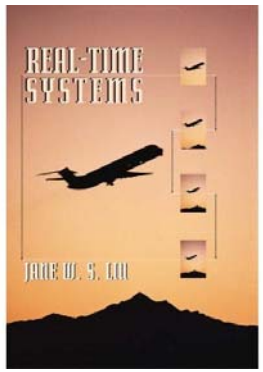
Kapitel 4

Scheduling

Inhalt

- Definitionen
- Kriterien zur Auswahl des Scheduling-Verfahrens
- Scheduling-Verfahren
- Prioritätsinversion
- Exkurs: Worst Case Execution Times

Literatur



Jane W. S. Liu, Real-Time Systems, 2000

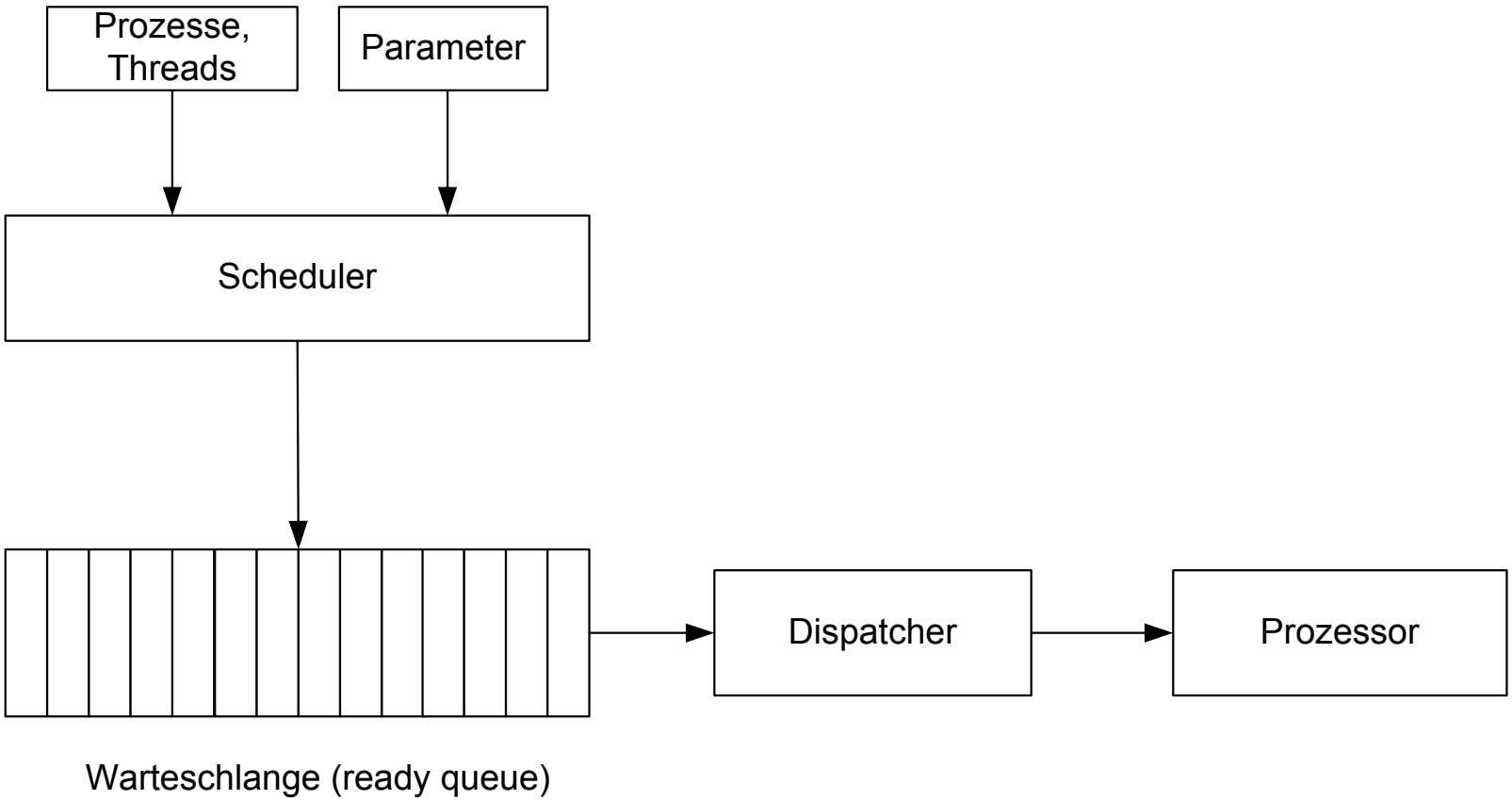
Fridolin Hofmann: Betriebssysteme - Grundkonzepte und Modellvorstellungen, 1991

- Journals:
  - John A. Stankovic, Marco Spuri, Marco Di Natale, and Giorgio C. Buttazzo: Implications of classical scheduling results for real-time systems. IEEE Computer, Special Issue on Scheduling and Real-Time Systems, 28(6):16–25, June 2005.
  - Giorgio C. Buttazzo: Rate Monotonic vs. EDF: Judgement Day (http://www.cas.mcmaster.ca/~downd/rtsj05-rmedf.pdf)
  - Puschner, Peter; Burns, Alan: A review of Worst-Case Execution-Time Analysis, Journal of Real-Time Systems 18 (2000), S.115-128

Scheduling

Definitionen

Scheduler und Dispatcher

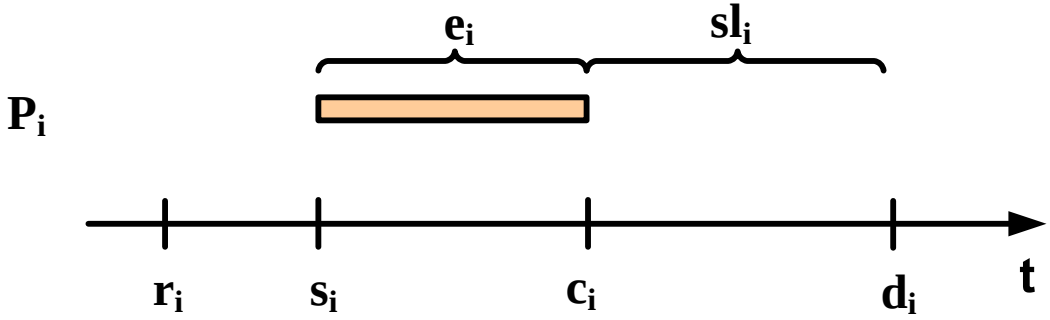


Scheduler und Dispatcher

- **Scheduler**: Modul eines Betriebssystems, das die Rechenzeit an die unterschiedlichen Prozesse verteilt. Der ausgeführte Algorithmus wird als Scheduling-Algorithmus bezeichnet. Aufgabe des Schedulers ist also die langfristige Planung (Vergleich: Erstellung eines Zugfahrplans).
- **Dispatcher**: Übersetzung: Einsatzleiter, Koordinator, Zuteiler (v.a. im Bereich der Bahn gebräuchlich). Im Rahmen der Prozessverwaltung eines Betriebssystems dient der Dispatcher dazu, bei einem Prozesswechsel dem derzeit aktiven Prozess die CPU zu entziehen und anschließend dem nächsten Prozess die CPU zuzuteilen. Die Entscheidung, welcher Prozess der nächste ist, wird vom Scheduler im Rahmen der Warteschlangenorganisation getroffen.

Zeitliche Bedingungen

- Folgende Größen sind charakteristisch für die Ausführung von Prozessen:
  1. P_i bezeichnet den i. **Prozess** (bzw. Thread)
  2. r_i: **Bereitzeit** (**ready time**) des Prozesses P_i und damit der früheste Zeitpunkt an dem der Prozess dem Prozessor zugeteilt werden kann.
  3. s_i: **Startzeit**: der Prozessor beginnt P_i auszuführen.
  4. e_i: **Ausführungszeit** (**execution time**): Zeit die der Prozess P_i zur reinen Ausführung auf dem Prozessor benötigt.
  5. c_i: **Abschlußzeit** (**completion time**): Zeitpunkt zu dem die Ausführung des Prozesses P_i beendet wird.
  6. d_i: **Frist** (**deadline**): Zeitpunkt zu dem die Ausführung des Prozesses P_i in jeden Fall beendet sein muss.



Spielraum (slack time)

- Mit dem Spielraum (slack time) sl_i eines Prozesses P_i wird Zeitraum bezeichnet, um den ein Prozess noch maximal verzögert werden darf:
  - Die Differenz zwischen der verbleibenden Zeit bis zum Ablauf der Frist und der noch benötigten Ausführungszeit zur Beendigung des Prozesses Pi .
- Der Spielraum eines Prozesses, der aktuell durch den Prozessor ausgeführt wird, bleibt konstant, während sich die Spielräume aller nicht ausgeführten Prozesse verringern.

Faktoren bei der Planung

- Für die Planung des Schedulings müssen folgende Faktoren berücksichtigt werden:
  - Art der Prozesse (periodisch, nicht periodisch, sporadisch)
  - Gemeinsame Nutzung von Ressourcen (**shared ressources**)
  - Fristen
  - Vorrangrelationen (**precedence constraints**: Prozess P_i muss vor P_j ausgeführt werden)

Arten der Planung

- Es kann zwischen unterschiedlichen Arten zum Planen unterschieden werden:
  - offline vs. online Planung
  - statische vs. dynamische Planung
  - präemptives vs. nicht-präemptives Scheduling

Offline Planung

- Mit der offline Planung wird die Erstellung eines Ausführungsplanes zur Übersetzungszeit bezeichnet. Zur Ausführungszeit arbeitet der Dispatcher den Ausführungsplan dann ab.
- **Vorteile**:
  - deterministisches Verhalten des Systems
  - wechselseitiger Ausschluss in kritischen Bereichen wird direkt im Scheduling realisiert
- **Nachteile**:
  - Bereitzeiten, Ausführungszeiten und Abhängigkeit der einzelnen Prozesse müssen schon im Voraus bekannt sein.
  - Die Suche nach einem Ausführungsplan ist im Allgemeinen ein NP-hartes Problem. Es werden jedoch keine optimalen Pläne gesucht, vielmehr ist ein gute Lösung (Einhaltung aller Fristen) ausreichend.

Online Scheduling

- Alle Schedulingentscheidungen werden online, d.h. auf der Basis der Menge der aktuell lauffähigen Prozesse und ihrer Parameter getroffen.
- Im Gegensatz zur offline Planung muss wechselseitiger Ausschluss nun über den expliziten Ausschluss (z.B. Semaphoren) erfolgen.
- Vorteile:
  - Flexibilität
  - Bessere Auslastung der Ressourcen
- Nachteile:
  - Es müssen zur Laufzeit Berechungen zum Scheduling durchgeführt werden $\Rightarrow$ Rechenzeit geht verloren.
  - Garantien zur Einhaltung von Fristen sind schwieriger zu geben.
  - Problematik von Race Conditions

Statische vs. dynamische Planung

- Bei der statischen Planung basieren alle Entscheidungen auf Parametern, die vor der Laufzeit festgelegt werden.
- Zur statischen Planung wird Wissen über:
  - die Prozessmenge
  - ihre Prioritäten
  - das Ausführungsverhalten

  benötigt.
- Bei der dynamischen Planung können sich die Scheduling-Parameter (z.B. die Prioritäten) zur Laufzeit ändern.
- **Wichtig**: Statische Planung und Online-Planung schließen sich nicht aus: z.B. Scheduling mit festen Prioritäten.

Präemption

- Präemptives (bevorrechtigt, entziehend) Scheduling: Bei jedem Auftreten eines relevanten Ereignisses wird die aktuelle Ausführung eines Prozesses unterbrochen und eine neue Schedulingentscheidung getroffen.
- Präemptives (unterbrechbares) Abarbeiten:
  - Aktionen (Prozesse) werden nach bestimmten Kriterien geordnet (z.B. Prioritäten, Frist,…).
  - Diese Kriterien sind statisch festgelegt oder werden dynamisch berechnet.
  - Ausführung einer Aktion wird sofort unterbrochen, sobald Aktion mit höherer Priorität eintrifft.
  - Die unterbrochene Aktion wird an der Unterbrechungsstelle fortgesetzt, sobald keine Aktion höherer Priorität ansteht.
  - Typisch für Echtzeitaufgaben (mit Ausnahme von Programmteilen, die zur Sicherung der Datenkonsistenz nicht unterbrochen werden dürfen).
  - Nachteil: häufiges Umschalten reduziert Leistung.

Ununterbrechbares Scheduling

- Ein Prozess, der den Prozessor zugewiesen bekommt, wird solange ausgeführt, bis der Prozess beendet wird oder er aber den Prozess freigibt.
- Scheduling-Entscheidungen werden nur nach der Prozessbeendigung oder dem Übergang des ausgeführten Prozesses in den blockierten Zustand vorgenommen.
- Eine begonnene Aktion wird beendet, selbst wenn während der Ausführung Aktionen höherer Dringlichkeit eintreffen
  ⇒ Nachteil: evtl. Versagen (zu lange Reaktionszeit) des Systems beim Eintreffen unvorhergesehener Anforderungen
- Anmerkung: Betriebssysteme unterstützen allgemein präemptives Scheduling solange ein Prozess im Userspace ausgeführt, Kernelprozesse werden häufig nicht oder selten unterbrochen.

  ⇒ Echtzeitbetriebssysteme zeichnen sich in Bezug auf das Scheduling dadurch aus, dass nur wenige Prozesse nicht unterbrechbar sind und diese wiederum sehr kurze Berechnungszeiten haben.

Schedulingkriterien

- Kriterien in Standardsystemen sind:
  - Fairness: gerechte Verteilung der Prozessorzeit
  - Effizienz: vollständige Auslastung der CPU
  - Antwortzeit: interaktive Prozesse sollen schnell reagieren
  - Verweilzeit: Aufgaben im Batchbetrieb (sequentielle Abarbeitung von Aufträgen) sollen möglichst schnell ein Ergebnis liefern
  - Durchsatz: Maximierung der Anzahl der Aufträge, die innerhalb einer bestimmten Zeitspanne ausgeführt werden
- In Echtzeitsystemen:
  - Einhaltung der Fristen: d.h. $\forall i\ c_i<d_i$ unter Berücksichtigung von Kausalzusammenhängen (Synchronisation, Vorranggraphen, Präzedenzsystemen)
  - Zusätzliche Kriterien können anwendungsabhängig hinzugenommen werden, solange sie der Einhaltung der Fristen untergeordnet sind.

Scheduling

Verfahren

Allgemeines Verfahren

- Gesucht: Plan mit aktueller Start und Endzeit für jeden Prozess P_i.
- Darstellung zum Beispiel als nach der Zeit geordnete Liste von Tupeln (P_i, s_i, c_i)
- Falls Prozesse unterbrochen werden können, so kann jedem Prozess P_i auch eine Menge von Tupeln zugeordnet werden.
- Phasen der Planung:
  - Test auf Einplanbarkeit (feasibility check)
  - Planberechnung (schedule construction)
  - Umsetzung auf Zuteilung im Betriebssystem (dispatching)
- Bei Online-Verfahren können die einzelnen Phasen überlappend zur Laufzeit ausgeführt werden.
- Zum Vergleich von Scheduling-Verfahren können einzelne Szenarien durchgespielt werden.

Definitionen

- **Zulässiger Plan**: Ein Plan ist zulässig, falls alle Prozesse einer Prozessmenge eingeplant sind und dabei keine Präzedenzrestriktionen und keine Zeitanforderungen verletzt werden.
- **Optimales Planungsverfahren**: Ein Verfahren ist optimal, falls es für jede Prozessmenge unter gegebenen Randbedingung einen zulässigen Plan findet, falls ein solcher existiert.

Test auf Einplanbarkeit

- Zum Test auf Einplanbarkeit können zwei Bedingungen angegeben werden, die für die Existenz eines zulässigen Plans notwendig sind (Achtung: häufig nicht ausreichend):
  1. $r_i + e_i \leq d_i$, d.h. jeder Prozess muss in dem Intervall zwischen Bereitzeit und Frist ausgeführt werden können.
  2. Für jeden Zeitraum $[t_i, t_j]$ muss die Summe der Ausführungszeiten e_x der Prozesse P_x mit $r_x \geq t_i \wedge d_x \leq$ tj kleiner als der Zeitraum sein.
- Durch weitere Rahmenbedingungen (z.B. Abhängigkeiten der einzelnen Prozesse) können weitere Bedingungen hinzukommen.

Schedulingverfahren

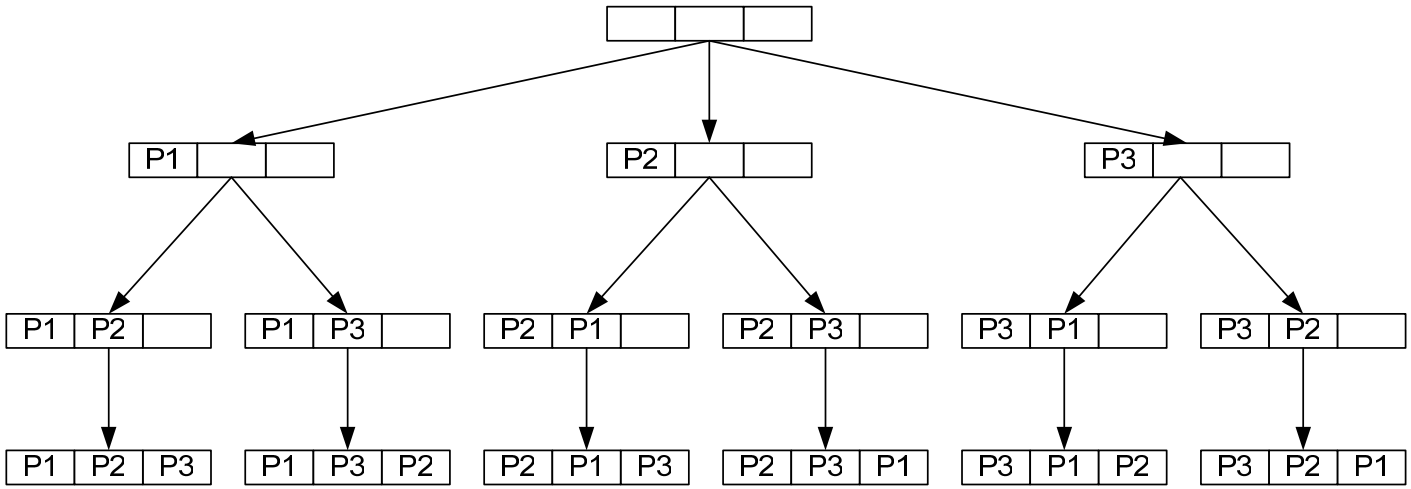
- Planen aperiodischer Prozesse
  - Planen durch Suchen
  - Planen nach Fristen
  - Planen nach Spielräumen
- Planen periodischer Prozesse
  - Planen nach Fristen
  - Planen nach Raten
- Planen abhängiger Prozesse

Scheduling

Scheduling-Verfahren für 1-Prozessor-Systeme

Planen durch Suchen

- Betrachtung: ununterbrechbare Aktionen/Prozesse vorausgesetzt
- Lösungsansatz: exakte Planung durch Durchsuchen des Lösungsraums
- Beispiel:
  - n=3 Prozesse P_1,P_2,P_3 und 1 Prozessor
  - Suchbaum:



Problem: Komplexität

- n! Permutationen müssen bewertet werden, bei Mehrprozessorsystemen ist das Problem der Planung NP-vollständig
- Durch präemptives Scheduling bzw. durch unterschiedliche Bereitzeiten kann das Problem weiter verkompliziert werden.
- Die Komplexität kann durch verschiedene Maßnahmen leicht verbessert werden:
  - Abbrechen von Pfaden bei Verletzung von Fristen
  - Verwendung von Heuristiken: z.B. Sortierung nach Bereitstellzeiten r_i
- Prinzipiell gilt jedoch: **Bei komplexen Systemen ist Planen durch Suchen nicht möglich**.

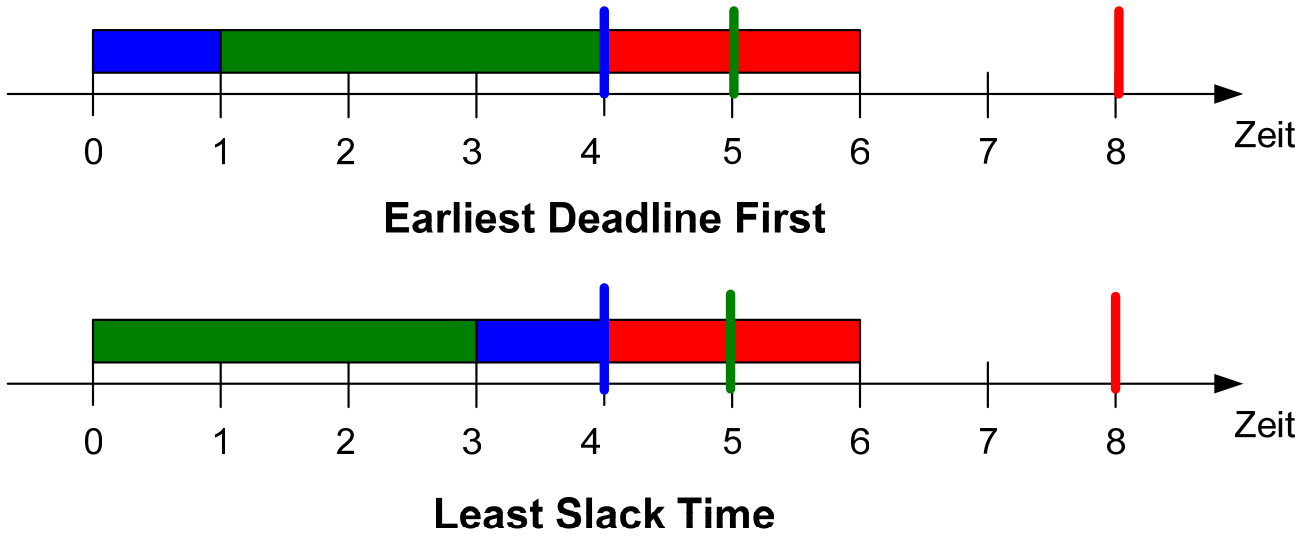
Scheduling-Strategien (online, nicht-präemptiv) für Einprozessorsysteme

1. EDF: Einplanen nach Fristen (Earliest Deadline First): Der Prozess, dessen Frist als nächstes endet, erhält den Prozessor.
2. LST: Planen nach Spielraum (Least Slack Time): Der Prozess mit dem kleinsten Spielraum erhält den Prozessor.
   - Der Spielraum berechnet sich wie folgt:
     Deadline-(aktuelle Zeit + verbleibende Berechnungszeit)
   - Der Spielraum für den aktuell ausgeführten Prozess ist konstant.
   - Die Spielräume aller anderen Prozesse nehmen ab.

- Vorteil von LST: LST erkennt Fristverletzungen früher als EDF.
  Nachteil: Ausführungszeiten der Prozesse müssen bekannt sein

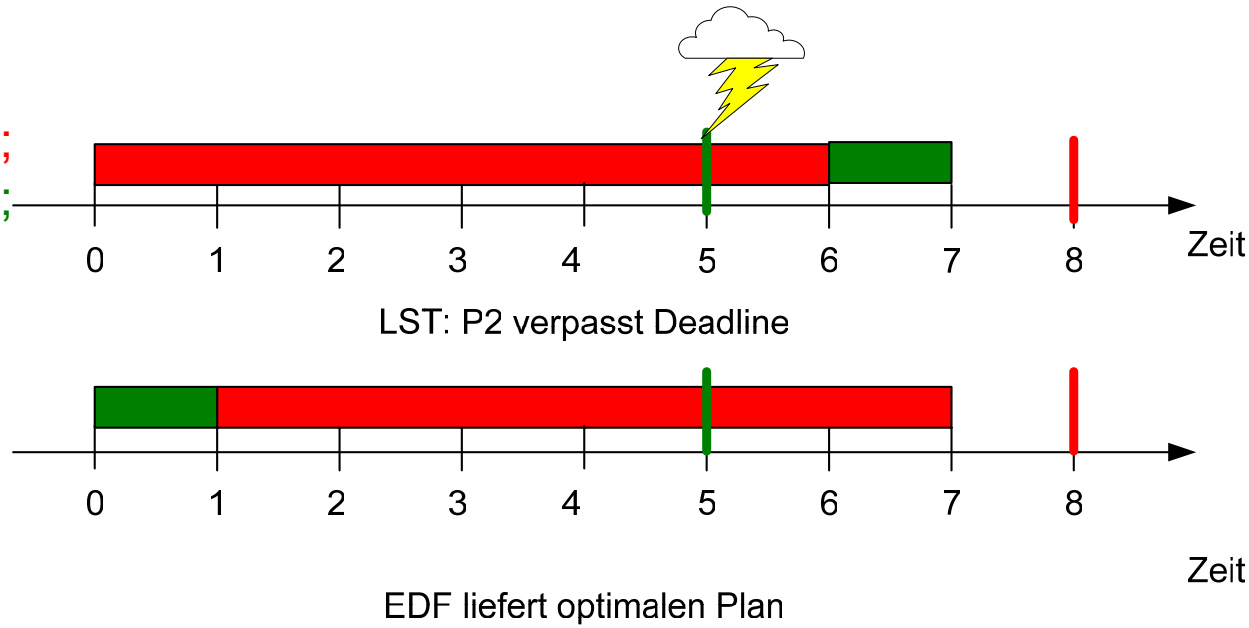
Beispiel

- 3 Prozesse:
  P_1: r_1=0; e_1=2; d_1=8;
  P_2: r_2=0; e_2=3; d_2=5;
  P_3: r_3=0; e_3=1; d_3=4;



Versagen von LST

- LST kann selbst bei gleichen Bereitzeiten im nicht-präemptiven Fall versagen.
- 2 Prozesse:
  P_1: r_1=0; e_1=6; d_1=8;
  P_2: r_2=0; e_2=1; d_2=4;



- Anmerkung: Aus diesem Grund wird LST nur in präemptiven Systemen eingesetzt. Bei Prozessen mit gleichen Spielräumen wird einem Prozess Δ eine Mindestausführungszeit garantiert.

Optimalität von EDF

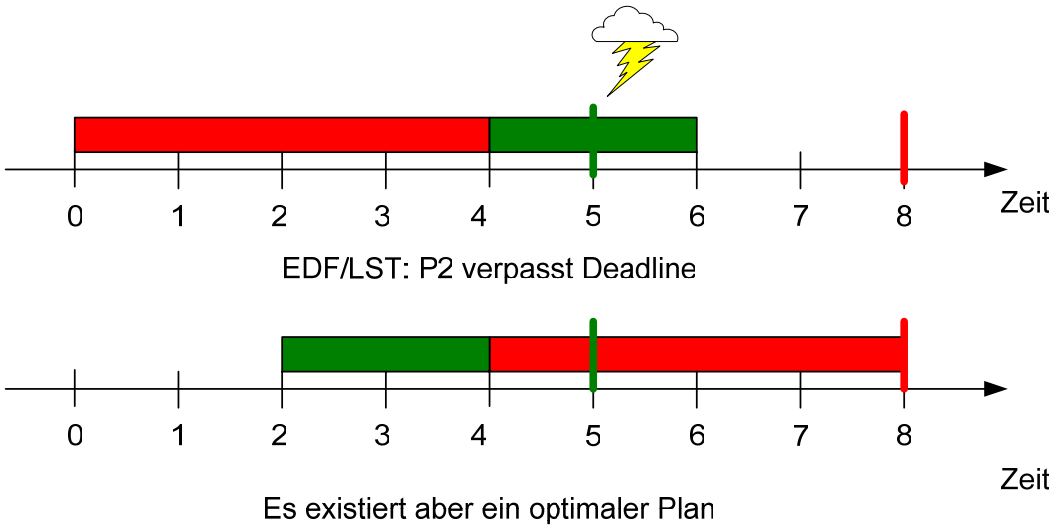
- Unter der Voraussetzung, dass alle Prozesse P_i eine Bereitzeit $r_i=0$ besitzen und das ausführende System ein Einprozessorsystem ist, ist EDF optimal, d.h. ein zulässiger Plan wird gefunden, falls ein solcher existiert.
- Beweisidee für EDF: Tausch in existierendem Plan
  - Sie $Plan_x$ ein zulässiger Plan.
  - Sei $Plan_{EDF}$ der Plan, der durch die EDF-Strategie erstellt wurde.
  - Ohne Einschränkung der Allgemeinheit: die Prozessmenge sei nach Fristen sortiert, d.h. $d_i \leq d_j$ für i<j.
  - Idee: Schrittweise Überführung des Planes $Plan_x$ in $Plan_{EDF}$
    - $P(Plan_x,t)$ sei der Prozess, der von $Plan_x$ zum Zeitpunkt t ausgeführt wird.
    - $Plan_x(t)$ ist der bis zum Zeitpunkt t in $Plan_{EDF}$ überführte Plan ($\Rightarrow$ $Plan_x(0)=Plan_x$).

Fortsetzung des Beweises

- Wir betrachten ein Zeitinterval Δ_t.
- Zum Zeitpunkt t gilt:
  $i=P(Plan_{EDF},t)$
  $j=P(Plan_x,t)$
- Nur der Fall j>i ist interessant. Es gilt:
  - $d_i \leq d_j$
  - $t+\Delta_t \leq d_i$ (ansonsten wäre der $Plan_x$ nicht zulässig)
  - Da die Pläne bis zum Zeitpunkt t identisch sind und P_i im $Plan_{EDF}$ zum Zeitpunkt t ausgeführt sind, kann der Prozess P_i im $Plan_x$ noch nicht beendet sein.
    $\Rightarrow \exists\ t'>t+\Delta_t: (i=P(Plan_x,t')=P(Plan_x,t'+\Delta_t) \wedge t'+\Delta_t \leq d_i \leq d_j$
    $\Rightarrow$ Die Aktivitätsphase von P_i im Zeitintervall $t'+\Delta_t$ und P_j im Zeitintervall $t+\Delta_t$ können ohne Verletzung der Zeitbedingungen getauscht werden $\Rightarrow$ Übergang von $Plan_x(t)$ zu $Plan_x(t+\Delta_t)$

Versagen von EDF bei unterschiedlichen Bereitzeiten

- Haben die Prozesse unterschiedliche Bereitzeiten, so kann EDF versagen.
- Beispiel: P_1: r_1=0; e_1=4; d_1=8 P2:r_2=2;e_2=2;d_2=5



- **Anmerkung**: Jedes prioritätsgesteuerte, nicht präemptive Verfahren versagt bei diesem Beispiel, da ein solches Verfahren nie eine Zuweisung des Prozessors an einen laufbereiten Prozess , falls ein solcher vorhanden ist, unterlässt.

Modifikationen

- Die Optimalität der Verfahren kann durch folgende Änderungen sichergestellt werden:
  - Präemptive Strategie
  - Neuplanung beim Erreichen einer neuen Bereitzeit
  - Einplanung nur derjenigen Prozesse, deren Bereitzeit erreicht ist

  $\Rightarrow$ Entspricht einer Neuplanung, falls ein Prozess aktiv wird.
- Bei Least Slack Time müssen zusätzlich Zeitscheiben für Prozesse mit gleichem Spielraum eingeführt werden, um ein ständiges Hin- und Her Schalten zwischen Prozessen zu verhindern.
- Generell kann gezeigt werden, dass die Verwendung von EDF die Anzahl der Kontextwechsel in Bezug auf Online-Scheduling-Verfahren minimiert (siehe Paper von Buttazzo)

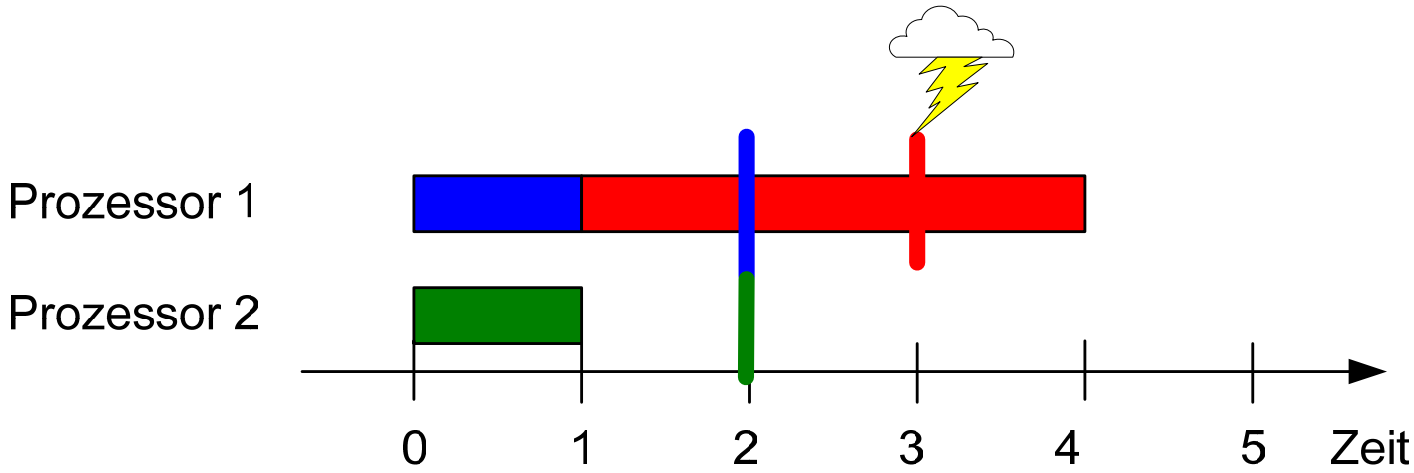
Zeitplanung auf Mehrprozessorsystemen

Zeitplanung auf Mehrprozessorsystemen

- Fakten zum Scheduling auf Mehrprozessorsystemen (Beispiele folgen):
  - EDF nicht optimal, egal ob präemptiv oder nicht präemptive Strategie
  - LST ist nur dann optimal, falls alle Bereitzeitpunkte r_i gleich
  - korrekte Zuteilungsalgorithmen erfordern das Abarbeiten von Suchbäumen mit NP-Aufwand oder geeignete Heuristiken
  - Beweisidee zur Optimalität von LST bei gleichen Bereitzeitpunkten: Der Prozessor wird immer dem Prozess mit geringstem Spielraum zugewiesen , d.h. wenn bei LST eine Zeitüberschreitung auftritt, dann auch, falls die CPU einem Prozess mit größerem Spielraum zugewiesen worden wäre.

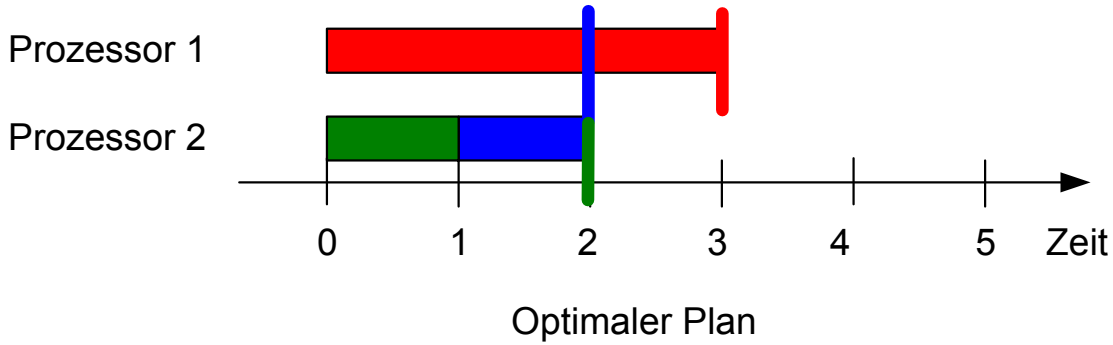
Beispiel: Versagen von EDF

- 2 Prozessoren, 3 Prozesse:
  P_1: r_1=0; e_1=3; d_1=3;
  P_2: r_2=0; e_2=1; d_2=2;
  P_3: r_3=0; e_3=1; d_3=2;

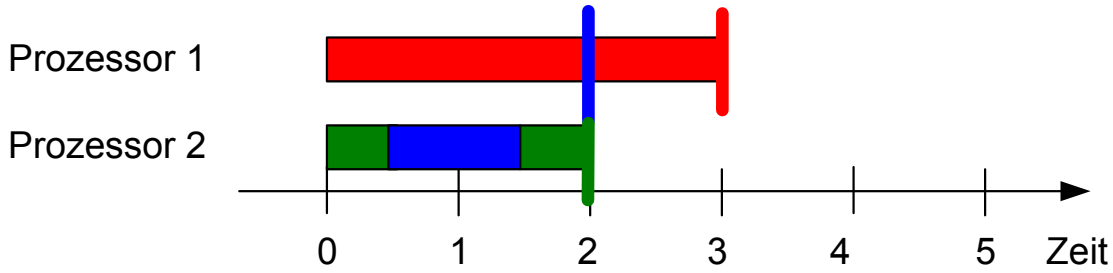


EDF-Verfahren: Deadline d1 wird verpasst

Beispiel: Optimaler Plan und LST-Verfahren



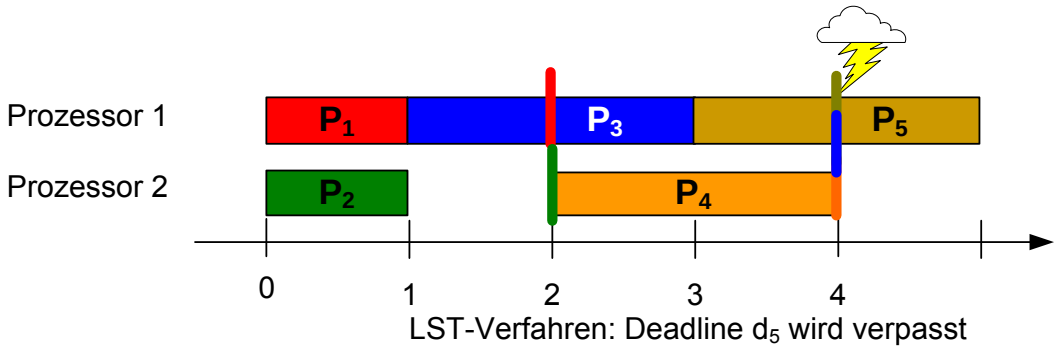
Optimaler Plan



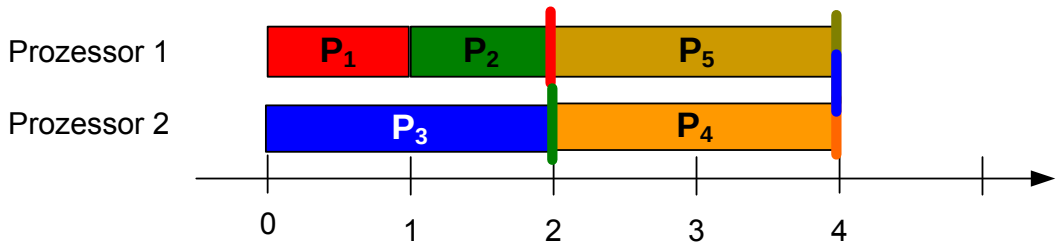
LST-Verfahren mit $\Delta_t = 0.5$

Beispiel: Versagen von LST

- 2 Prozessoren, 5 Prozesse, Δ_t=0,5:
  P_1: r_1=0; e_1=1; d_1=2;
  P_2: r_2=0; e_2=1; d_2=2;
  P_3: r_3=0; e_3=2; d_3=4;
  P_4: r_4=2; e_4=2;d_4=4;
  P_5: r_5=2; e_5=2;d_5=4;



LST-Verfahren: Deadline d_5 wird verpasst



Es existiert ein gültiger Plan

Versagen von präemptiven Schedulingverfahren

- Jeder präemptiver Algorithmus versagt, wenn die Bereitstellzeiten unterschiedlich sind und nicht im Voraus bekannt sind.
  Beweis:
  - n CPUs und n-2 Prozesse ohne Spielraum (n-2 Prozesse müssen sofort auf n-2 Prozessoren ausgeführt werden) $\Rightarrow$ Reduzierung des Problems auf 2-Prozessor-Problem
  - Drei weitere Prozesse sind vorhanden und müssen eingeplant werden.
  - Die Reihenfolge der Abarbeitung ist von der Strategie abhängig, in jedem Fall kann aber folgender Fall konstruiert werden, so dass:
    - es zu einer Fristverletzung kommt,
    - aber ein gültiger Plan existiert.

Fortsetzung Beweis

- Szenario:
  P_1: r_1=0; e_1=1; d_1=1;
  P_2: r_2=0; e_2=2; d_2=4;
  P_3: r_3=0; e_3=1; d_3=2;

  ⇒ Prozess P_1 (kein Spielraum) muss sofort auf CPU1 ausgeführt werden.

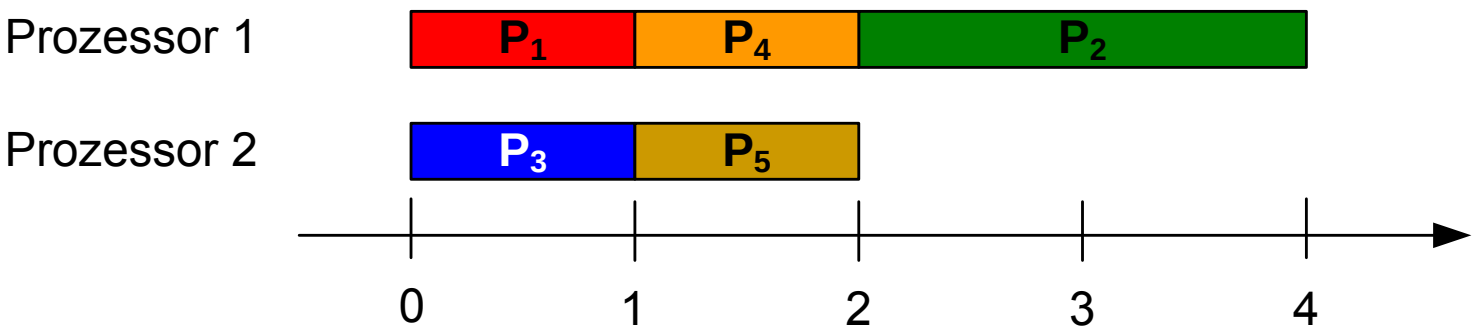
  ⇒ Es gibt je nach Strategie zwei Fälle zu betrachten: P2 oder P3 wird zunächst auf CPU2 ausgeführt.

1. Fall

- P_2 wird zum Zeitpunkt 0 auf CPU2 ausgeführt.
  - Zum Zeitpunkt 1 muss dann P_3 (ohne Spielraum) ausgeführt werden.
  - Zum Zeitpunkt 1 treffen aber zwei weitere Prozesse P_4 und P_5 mit Frist 2 und Ausführungsdauer 1 ein.

  $\Rightarrow$ Es gibt drei Prozesse ohne Spielraum, aber nur zwei Prozessoren.

- Aber es gibt einen gültigen Ausführungsplan:

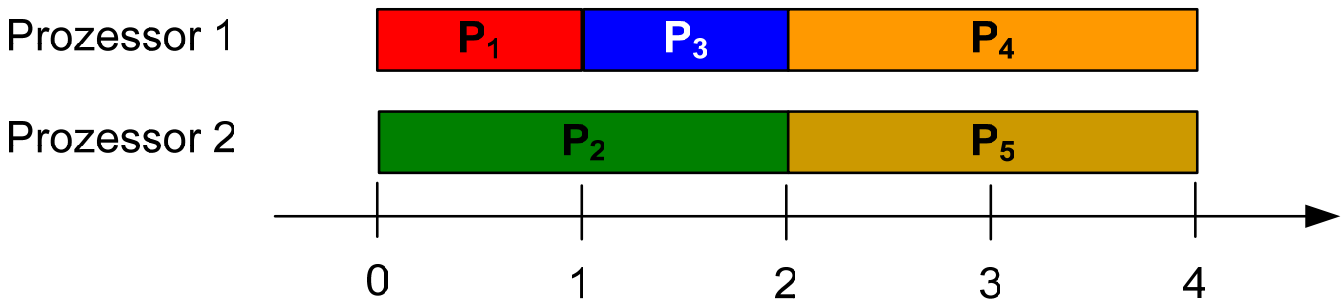


2. Fall

- P_3 wird zum Zeitpunkt 0 auf CPU2 ausgeführt.
  - Zum Zeitpunkt 1 sind P_1 und P_3 beendet.
  - Zum Zeitpunkt 1 beginnt P_2 seine Ausführung.
  - Zum Zeitpunkt 2 treffen aber zwei weitere Prozesse P_4 und P_5 mit Deadline 4 und Ausführungsdauer 2 ein.

  $\Rightarrow$ Anstelle der zum Zeitpunkt 2 noch notwendigen 5 Ausführungseinheiten sind nur 4 vorhanden.

- Aber es gibt einen gültigen Ausführungsplan:



Strategien in der Praxis

- Die Strategien EDF und LST werden in der Praxis selten angewandt. Gründe:
  - In der Realität sind keine abgeschlossenen Systeme vorhanden (Alarme, Unterbrechungen erfordern eine dynamische Planung)
  - Bereitzeiten sind nur bei zyklischen Prozessen oder Terminprozessen bekannt.
  - Die Abschätzung der Laufzeit sehr schwierig ist (siehe Exkurs).
  - Synchronisation, Kommunikation und gemeinsame Betriebsmittel verletzen die Forderung nach Unabhängigkeit der Prozesse.

Ansatz in der Praxis

- Zumeist basiert das Scheduling auf der Zuweisung von statischen Prioritäten.
- Prioritäten werden zumeist durch natürliche Zahlen zwischen 0 und 255 ausgedrückt. Die höchste Priorität kann dabei sowohl 0 (z.B. in VxWorks) als auch 255 (z.B. in POSIX) sein.
- Die Priorität ergibt sich aus der Wichtigkeit des technischen Prozesses und der Abschätzung der Laufzeiten und Spielräume. Die Festlegung erfolgt dabei durch den Entwickler.
- Bei gleicher Priorität wird zumeist eine FIFO-Strategie (d.h. ein Prozess läuft solange, bis er entweder beendet ist oder aber ein Prozess höherer Priorität eintrifft) angewandt.
  Alternative Round Robin: Alle laufbereiten Prozesse mit der höchsten Priorität erhalten jeweils für eine im Voraus festgelegte Zeitdauer die CPU.

Scheduling

Zeitplanen periodischer Prozesse

Zeitplanung periodischer Prozesse

- Annahmen für präemptives Scheduling
  - Alle Prozesse treten periodisch mit einer Frequenz f_i auf.
  - Die Frist eines Prozesses entspricht dem nächsten Startpunkt.
  - Sind die maximalen Ausführungszeiten e_i bekannt, so kann leicht errechnet werden, ob ein ausführbarer Plan existiert.
  - Die für einen Prozesswechsel benötigten Zeiten sind vernachlässigbar.
  - Alle Prozesse sind unabhängig.
- Eine sehr gute Zusammenfassung zu dem Thema Zeitplanung periodischer Prozesse liefert Giorgio C. Buttazzo in seinem Paper „Rate Monotonic vs. EDF: Judgement Day“ (http://www.cas.mcmaster.ca/~downd/rtsj05-rmedf.pdf).

Einplanbarkeit

- Eine notwendige Bedingung zur Einplanbarkeit ist die Last:
  - Last eines einzelnen Prozesses: $\rho_i = e_i * f_i$
  - Gesamte Auslastung bei n Prozessen:

$$\rho = \sum_{i=0}^{n} \rho_i$$

  - Bei m Prozessoren ist $\rho < m$ eine notwendige aber nicht ausreichende Bedingung.

Zeitplanen nach Fristen

- **Ausgangspunkt**: Wir betrachten Systeme mit einem Prozessor und Fristen der Prozesse, die relativ zum Bereitzeitpunkt deren Perioden entsprechen, also $d_i=1/f_i$.
- **Aussage**: Die Einplanung nach Fristen ist optimal.
- **Beweisidee**: Vor dem Verletzen einer Frist ist die CPU nie unbeschäftigt $\Rightarrow$ die maximale Auslastung liegt bei 100%.
- Leider wird aufgrund von diversen Vorurteilen EDF selten benutzt.
- Betriebssysteme unterstützen selten ein EDF-Scheduling
  $\Rightarrow$ Die Implementierung eines EDF-Scheduler auf der Basis von einem prioritätsbasierten Scheduler ist nicht effizient zu implementieren (Ausnahme: zeitgesteuerte Systeme)

Zeitplanung nach Raten

- Rate Monotonic bezeichnet ein Scheduling-Verfahren mit festen Prioritäten Prio(i), die sich proportional zu den Frequenzen verhalten.

  $\Rightarrow$ Prozesse mit hohen Raten werden bevorzugt. Das Verfahren ist optimal, falls eine Lösung mit statischen Prioritäten existiert. Verfahren mit dynamischen Prioritäten können allerdings eventuell bessere Ergebnisse liefern.

- Liu und Layland haben 1973 in einer Worst-Case-Analyse gezeigt, dass Ratenplanung sicher erfolgreich ist, falls bei n Prozessen auf einem Prozessor gilt:

$$\rho \leq \rho_{max} = n*(2^{1/n} - 1)$$

$$\lim_{n \to \infty} \rho_{max} = \ln 2 \approx 0{,}69$$

- Derzeit zumeist verwendetes Scheduling-Verfahren im Bereich von periodischen Prozessen.

Scheduling

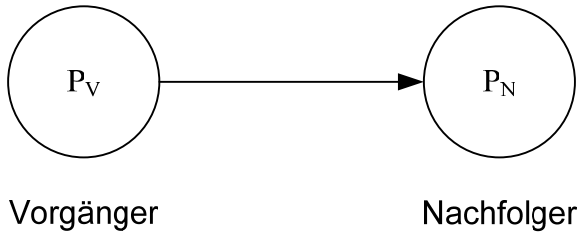
Planen abhängiger Prozesse

Allgemeines zum Scheduling in Echtzeitsystemen

- Grundsätzlich kann der Prozessor neu vergeben werden, falls:
  - ein Prozess endet,
  - ein Prozess in den blockierten Zustand (z.B. wegen Anforderung eines blockierten Betriebsmittels) wechselt,
  - eine neuer Prozess gestartet wird,
  - ein Prozess vom blockierten Zustand in den Wartezustand wechselt (z.B. durch die Freigabe eines angeforderten Betriebmittels durch einen anderen Prozess)
  - oder nach dem Ablauf eines Zeitintervals, siehe z.B. Round Robin.
- Hochpriorisierte Prozesse dürfen in Echtzeitsystemen nicht durch unwichtigere Prozesse behindert werden $\Rightarrow$ Die Prioritätsreihenfolge muss bei allen Betriebsmitteln (CPU, Semaphore, Netzkommunikation, Puffer, Peripherie) eingehalten werden, d.h. Vordrängen in allen Warteschlangen.

Präzedenzsysteme

- Zur Vereinfachung werden zunächst Systeme betrachtet, bei denen die Bereitzeiten der Prozesse auch abhängig von der Beendigung anderer Prozesse sein können.
- Mit Hilfe von Präzedenzsystemen können solche Folgen von voneinander abhängigen Prozessen beschrieben werden.
- Zur Beschreibung werden typischerweise Graphen verwendet:



- Der Nachfolgerprozess kann also frühestens beim Erreichen der eigenen Bereitzeit **und** der Beendigung der Ausführung des Vorgängerprozesses ausgeführt werden.

Probleme bei Präzedenzsystemen

- Bei der Planung mit Präzedenzsystemen muss auch berücksichtigt werden, dass die Folgeprozesse noch rechtzeitig beendet werden können.
- Beispiel:
  P_V: $r_V=0$; $e_V=1$; $d_V=3$;
  P_N: $r_N=0$; $e_N=3$; $d_N=5$;
- Falls die Frist von P_V voll ausgenutzt wird, kann der Prozess P_N nicht mehr rechtzeitig beendet werden.

  $\Rightarrow$ Die Fristen müssen entsprechend den Prozessabhängigkeiten neu berechnet werden (Normalisierung von Präzedenzsystemen).

Normalisierung von Präzedenzsystemen

- Anstelle des ursprünglichen Präzedenzsystems PS wird ein normalisiertes Präzedenzsystem PS' mit folgenden Eigenschaften:
  - $\forall i: e'_i = e_i$
  - $$\forall i : d'_i = \begin{cases} d_i, falls N_i = \emptyset \\ min(d_i, min(d'_q - e'_q | q \in N_i) \end{cases}$$

    wobei N_i die Menge der Nachfolger im Präzedenzgraph bezeichnet und d'_i rekursiv beginnend bei Prozessen ohne Nachfolger berechnet wird.
  - Falls die Bereitzeiten von externen Ereignissen abhängig sind, gilt $r'_i = r_i$. Sind die Bereitzeiten dagegen abhängig von der Beendigung der Prozesse, so ergeben sie sich aus dem konkreten Scheduling.

  eingeführt.

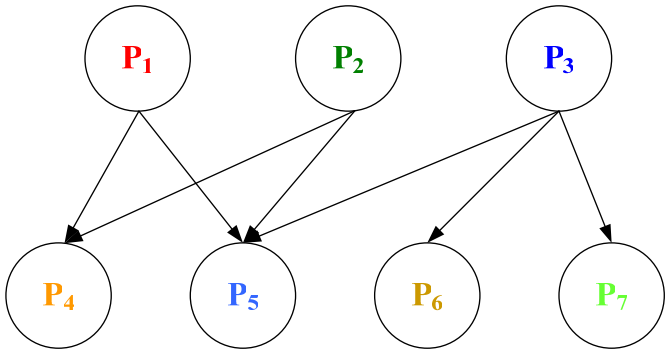
  $\Rightarrow$ Ein Präzedenzsystem ist nur dann planbar, falls das zugehörige normalisierte Präzedenzsystem planbar ist.

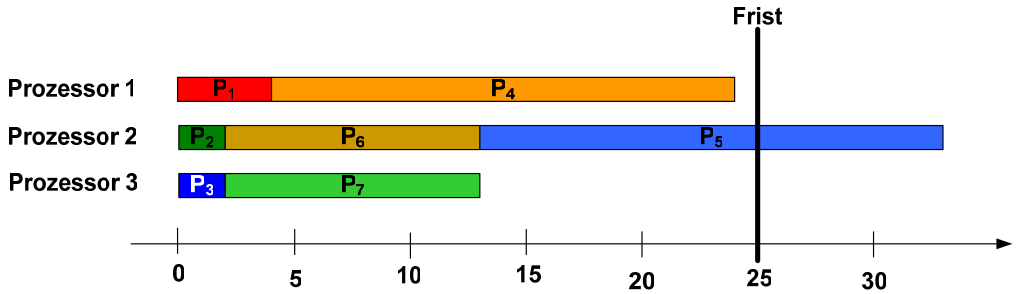
Anomalien bei nicht präemptiven Scheduling

- Wird zum Scheduling von Präzedenzsystemen ein nicht präemptives prioritätenbasiertes Verfahren (z.B. EDF, LST) verwendet, so können Anomalien auftreten:
  - Durch Hinzufügen eines Prozessors kann sich die gesamte Ausführungszeit verlängern.
  - Durch freiwilliges Warten kann die gesamte Ausführungszeit verkürzt werden.

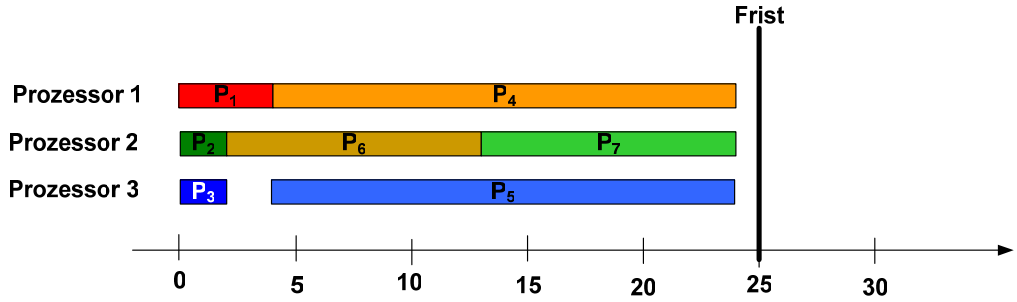
Beispiel: Verkürzung durch freiwilliges Warten

- Beispiel: 3 Prozessoren, 7 Prozesse (r_i=0, e_1=4; e_2=2; e_3=2; e_4=20; e_5=20; e_6=11; e_7=11, d_i=25), Präzedenzgraph:





Prioritätenbasiertes Scheduling

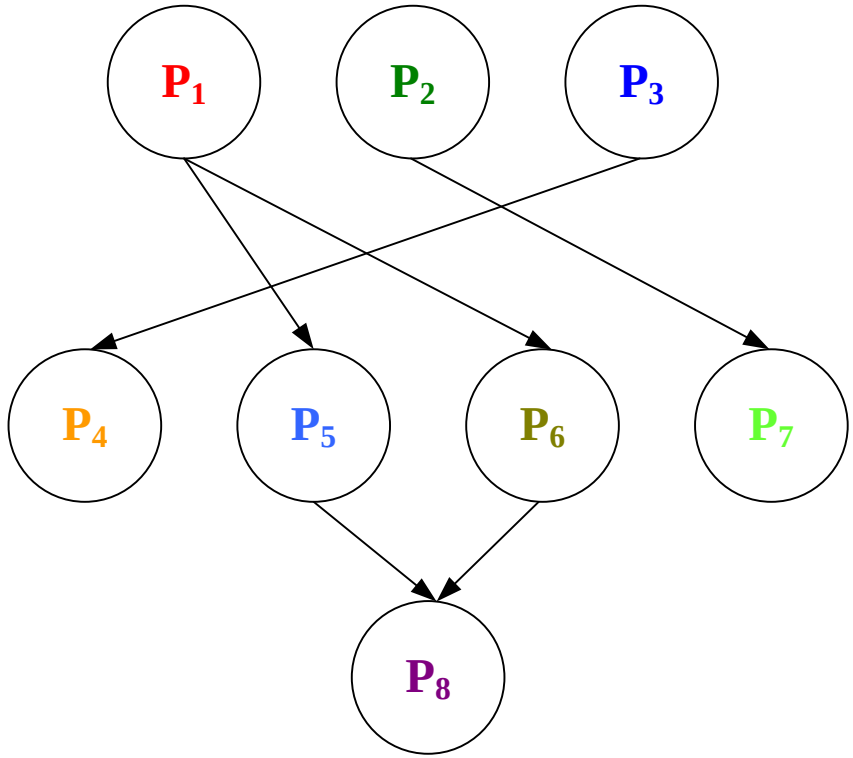


Optimaler Plan

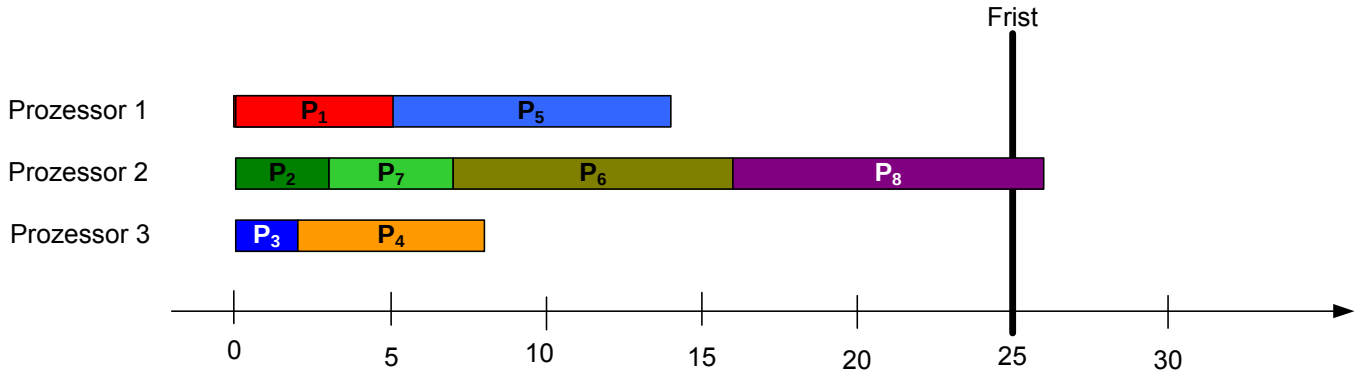
Beispiel: Laufzeitverlängerung durch zusätzlichen Prozessor II

- Beispiel:
  - 2 bzw. 3 Prozessoren
  - 8 Prozesse:
    - Startzeiten $r_i=0$
    - Ausführungszeiten
      $e_1=5$;
      $e_2=3$;
      $e_3=2$;
      $e_4=6$;
      $e_5=9$;
      $e_6=9$;
      $e_7=4$,
      $e_8=10$
    - Frist: $d_i=25$

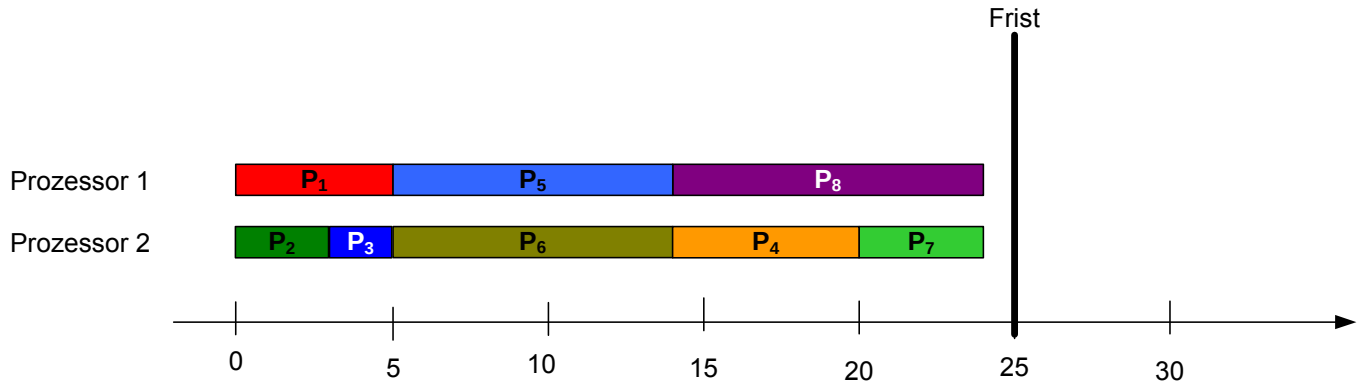
- Präzedenzgraph:



Beispiel: Laufzeitverlängerung durch zusätzlichen Prozessor II



Prioritätenbasiertes Scheduling (LST) auf 3 Prozessoren



Prioritätenbasiertes Scheduling (LST) auf 2 Prozessoren

Scheduling

Problem: Prioritätsinversion

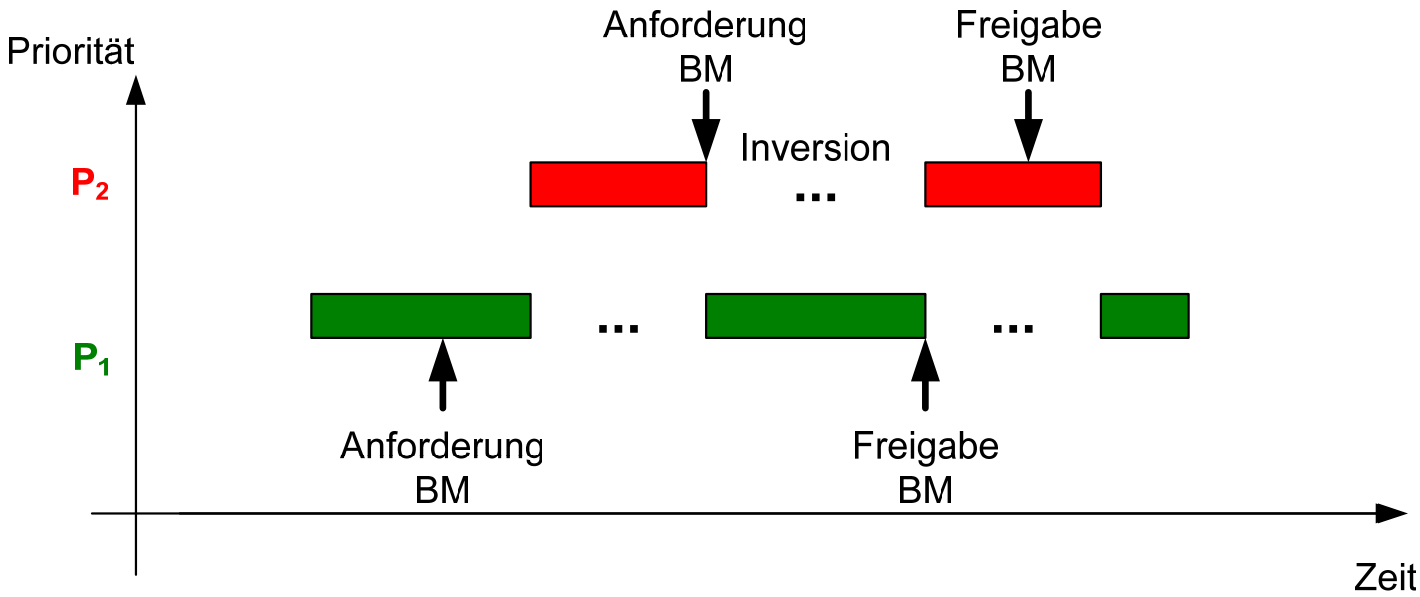
Motivation des Problems

- Selbst auf einem Einprozessoren-System mit präemptiven Scheduling gibt es Probleme bei voneinander abhängigen Prozessen.
- Abhängigkeiten können diverse Gründe haben:
  - Prozesse benötigen Ergebnisse eines anderen Prozesses
  - Betriebsmittel werden geteilt
  - Es existieren kritische Bereiche, die durch Semaphoren oder Monitoren geschützt sind.
- Gerade aus den letzten zwei Punkten entstehen einige Probleme:
  - Die Prozesse werden unter Umständen unabhängig voneinander implementiert $\Rightarrow$ das Verhalten des anderen Prozesses ist nicht bekannt.
  - Bisher haben wir noch keinen Mechanismus zum Umgang mit blockierten Betriebsmitteln kennengelernt, falls hochpriore Prozesse diese Betriebsmittel anfordern.

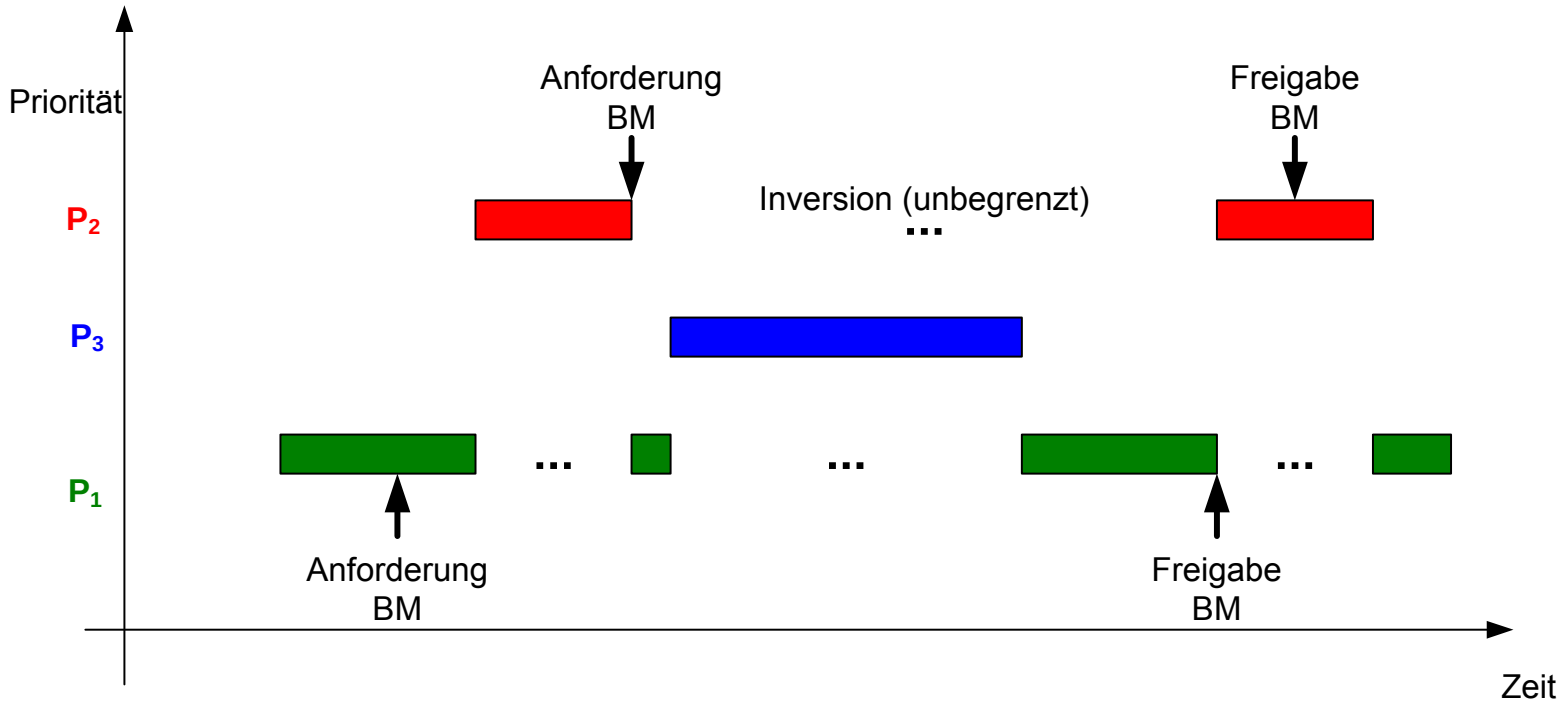
Prioritätsinversion

- **Definition**: Das Problem der **Prioritätsinversion** bezeichnet Situationen, in denen ein Prozess mit niedriger Priorität einen höherpriorisierten Prozess blockiert.
- Dabei unterscheidet man zwei Arten der Prioritätsinversion:
  - **begrenzte** (bounded) Prioritätsinversion: die Inversion ist durch die Dauer des kritischen Bereichs beschränkt.
  - **unbegrenzte** (unbounded) Prioritätsinversion: durch weitere Prozesse kann der hochpriorisierte Prozess auf unbestimmte Dauer blockiert werden.
- Während das Problem der begrenzten Prioritätsinversion aufgrund der begrenzten Zeitdauer akzeptiert werden kann (muss), ist die unbegrenzte Prioritätsinversion in Echtzeitsystemen unbedingt zu vermeiden.

Begrenzte Inversion



Unbegrenzte Inversion



Reales Beispiel: Mars Pathfinder

- **System**: Der Mars Pathfinder hatte zur Speicherung der Daten einen Informationsbus (vergleichbar mit Shared Memory). Der Informationsbus war durch einen binären Semaphore geschützt. Ein Bus Management Prozess verwaltete den Bus mit hoher Priorität.
  Ein weiterer Prozess war für die Sammlung von geologischen Daten eingeplant. Dieser Task lief mit einer niedrigen Priorität. Zusätzlich gab es noch einen Kommunikationsprozess mittlerer Priorität.
- **Symptome**: Das System führte in unregelmäßigen Abständen einen Neustart durch. Daten gingen dadurch verloren.
- **Ursache**: Der binäre Semaphore war nicht mit dem Merkmal zur Unterstützung von Prioritätsvererbung (siehe später) erzeugt worden. Dadurch kam es zur Prioritätsinversion. Ein Watchdog (Timer) erkannte eine unzulässige Verzögerung des Bus Management Prozesses und führte aufgrund eines gravierenden Fehlers einen Neustart durch.

Ansätze zur Lösung der Prioritätsinversion

- Es existieren verschiedene Ansätze um das Problem der unbegrenzten Prioritätsinversion zu begrenzen:
  - Prioritätsvererbung (priority inheritance)
  - Prioritätsobergrenzen (priority ceiling)
  - Unmittelbare Prioritätsobergrenzen (immediate priority ceiling)
- Anforderungen an Lösungen:
  - leicht zu implementieren
  - Anwendungsunabhängige Implementierung
  - Eventuell Ausschluss von Verklemmungen

Prioritätsvererbung (priority inheritance)

- Sobald ein Prozess höherer Priorität ein Betriebsmittel anfordert, das ein Prozess mit niedrigerer Priorität besitzt, erbt der Prozess mit niedrigerer Priorität die höhere Priorität. Nachdem das Betriebsmittel freigegeben wurde, fällt die Priorität wieder auf die ursprüngliche Priorität zurück.

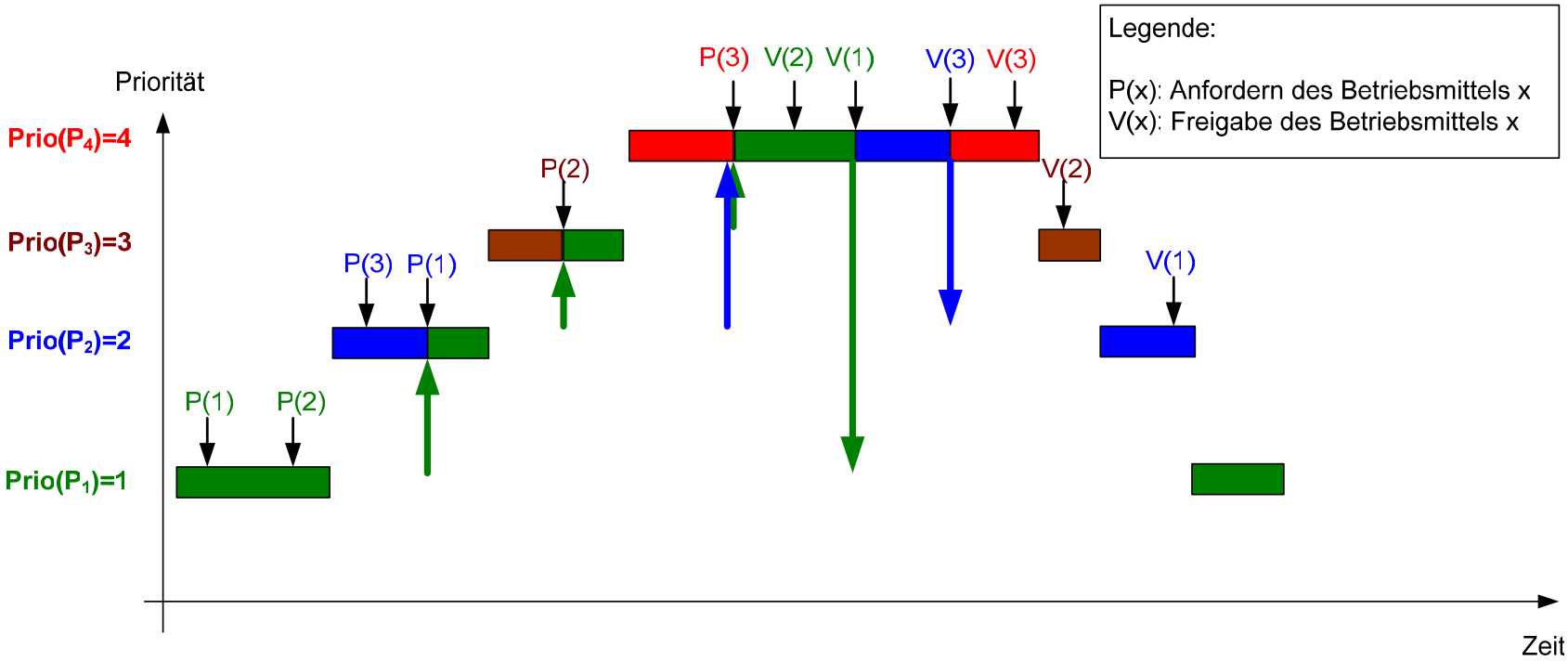
  ⇒ Unbegrenzte Prioritätsinversion wird verhindert.

  ⇒ Die Dauer der Blockade wird durch die Dauer des kritischen Abschnittes beschränkt.

  ⇒ Blockierungen werden hintereinander gereiht (Blockierungsketten).

  ⇒ Verklemmungen durch Programmierfehler werden nicht verhindert.

Beispiel: Prioritätsvererbung



Prioritätsobergrenzen (priority ceiling)

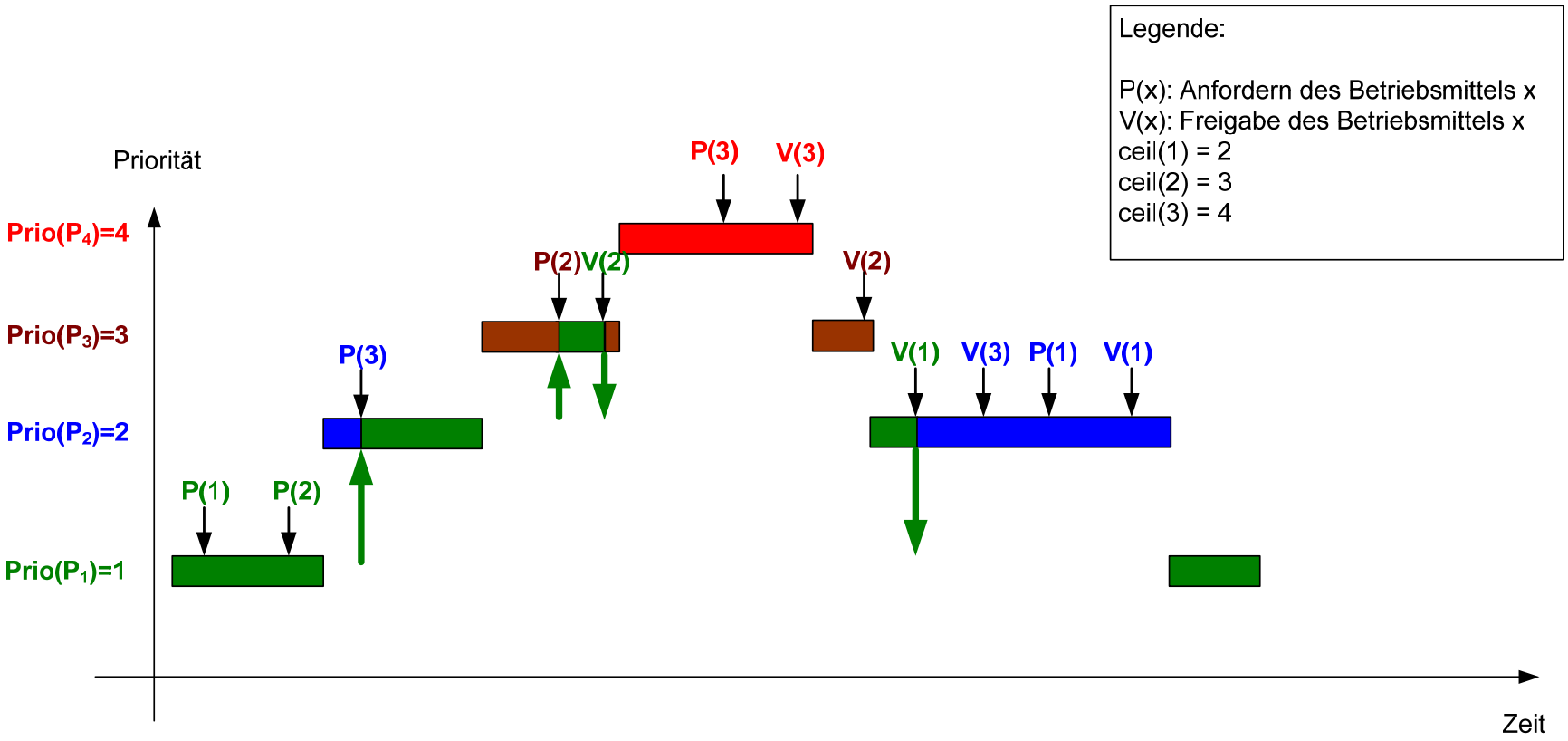
- Jedem Betriebsmittel (z.B. Semaphor) s wird eine Prioritätsgrenze ceil(s) zugewiesen, diese entspricht der maximalen Priorität der Prozesse, die auf s zugreifen.
  - Ein Prozess p darf ein BM nur blockieren, wenn er von keinem anderen Prozess, der andere BM besitzt, verzögert werden kann.
  - Die aktuelle Prioritätsgrenze für Prozess p ist
    aktceil(p) = max{ ceil(s) | s $\in$ locked} mit locked = Menge aller von anderen Prozessen blockierten BM
  - Prozess p darf Betriebsmittel s benutzen, wenn für seine aktuelle Priorität aktprio gilt:
    aktprio(p) > aktceil(p)
  - Andernfalls gibt es genau einen Prozess, der s besitzt. Die Priorität dieses Prozesses wird auf aktprio(p) gesetzt.

  $\Rightarrow$ Blockierung nur für die Dauer eines kritischen Abschnitts

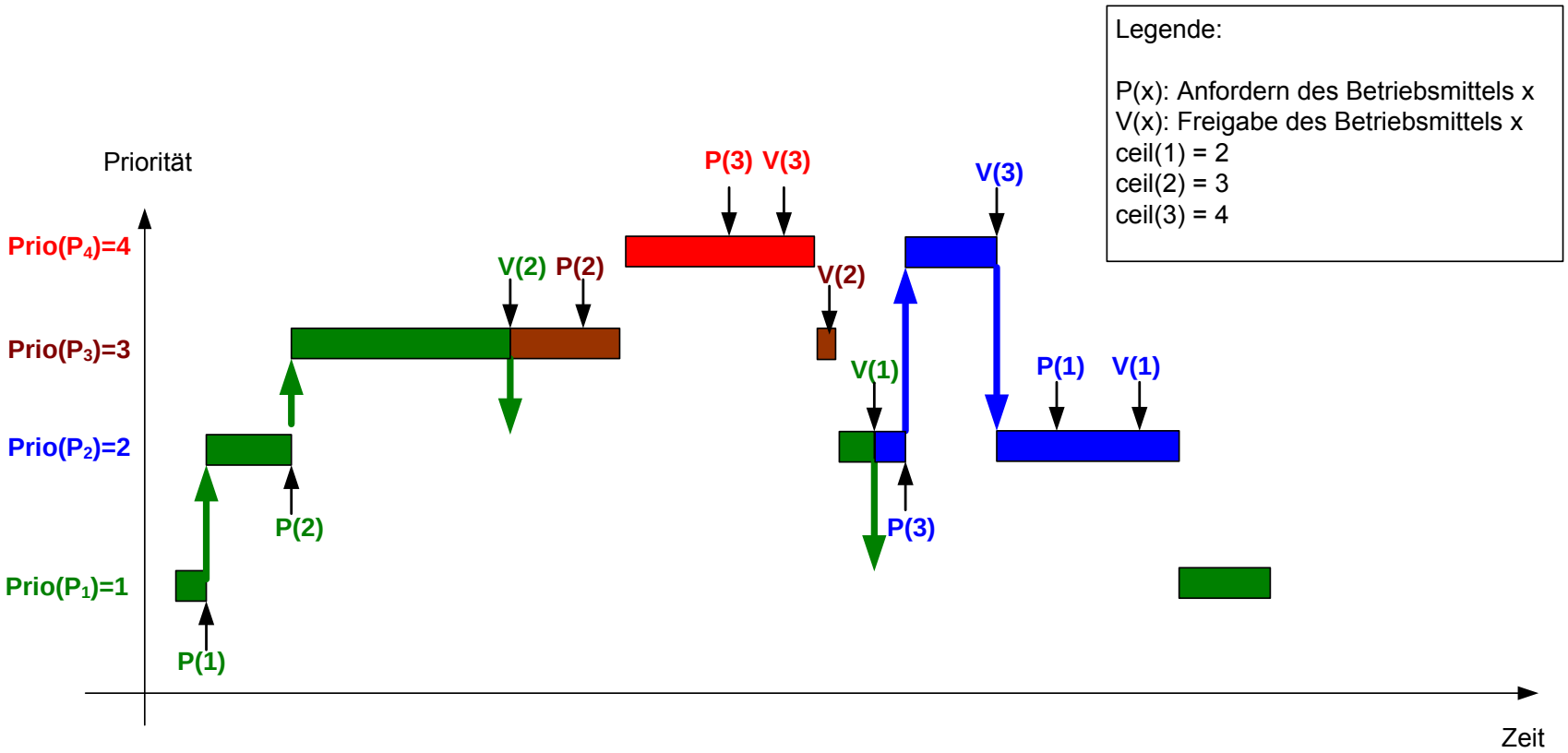
  $\Rightarrow$ Verhindert Verklemmungen

  $\Rightarrow$ schwieriger zu realisieren, zusätzlicher Prozesszustand
- Vereinfachtes Protokoll: **Immediate priority ceiling**: Prozesse, die ein Betriebsmittel s belegen, bekommen sofort die Priorität ceil(s) zugewiesen.

Beispiel: Prioritätsobergrenzen



Beispiel: Immediate Priority Ceiling

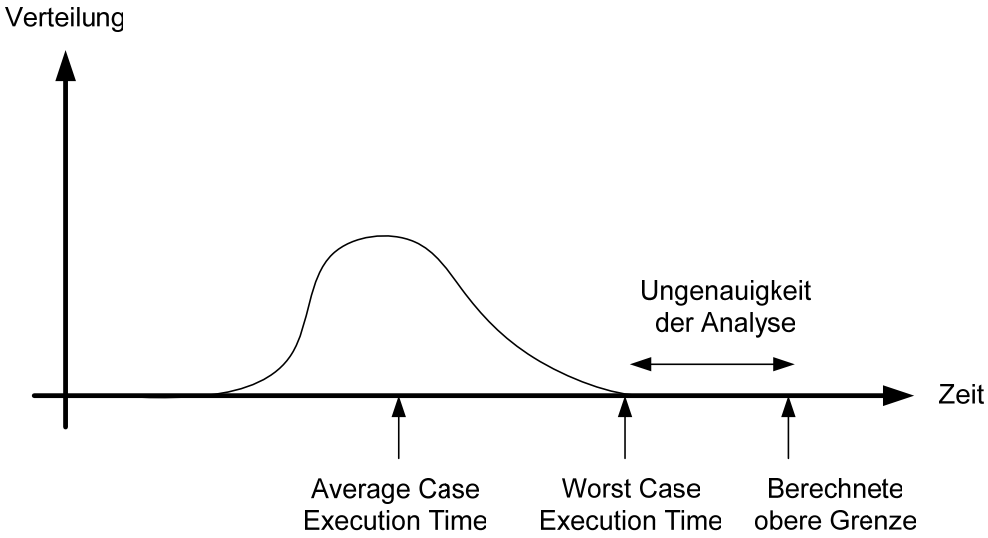


Scheduling

Exkurs: WCET (Worst Case Execution Time) - Analyse

WCET Analyse

- Ziel der Worst Case Execution Time Analyse ist die Abschätzung der maximalen Ausführungszeit einer Funktion



- Die Laufzeit ist abhängig von den Eingabedaten, dem Prozessorzustand, der Hardware,…

Probleme bei der WCET Analyse

- Bei der Abschätzung der maximalen Ausführungszeiten stößt man auf einige Probleme:
  - Es müssen unter anderem die Auswirkungen der Hardwarearchitektur, des Compilers und des Betriebssystems untersucht werden. Dadurch erschwert sich eine Vorhersage.
  - Zudem dienen viele Eigenschaften der Beschleunigung des allgemeinen Verhaltens, jedoch nicht des Verhaltens im schlechtesten Fall, z.B.:
    - Caches, Pipelines, Virtual Memory
    - Interruptbehandlung, Präemption
    - Compileroptimierungen
    - Rekursion

	Zugriffszeit	Größe
Register	0.25 ns	500 bytes
Cache	1 ns	64 KB
Hauptspeicher	100 ns	512 MB
Festplatte	5 ms	100 GB

Zugriffszeiten für verschiedene Speicherarten

  - Noch schwieriger wird die Abschätzung falls der betrachtete Prozess von der Umgebung abhängig ist.

Unterscheidungen bei der WCET-Analyse

- Die Analyse muss auf unterschiedlichen Ebenen erfolgen:
  - Was macht das Programm?
  - Was passiert im Prozessor?

- Bei der Analyse werden zwei Methoden unterschieden:
  - **statische** WCET Analyse: Untersuchung des Programmcodes
  - **dynamische** Analyse: Bestimmung der Ausführungszeit anhand von verschiedenen repräsentativen Durchläufen

Statische Analyse

- Aufgaben:
  - Bestimmung von Ausführungspfaden in der Hochsprache
  - Transformation der Pfade in Maschinencode
  - Bestimmung der Laufzeit einzelner Maschinencodesequenzen
- Probleme:
  - Ausführungspfade lassen sich oft schlecht vollautomatisch ableiten (zu pessimistisch, zu komplex)
  - Ausführungspfade häufig abhängig von Eingabedaten
- Lösungsansatz: Annotierung der Pfade mit Beschränkungen (wie z.B. maximale Schleifendurchläufe)

Dynamische Analyse

- Statische Analysen können zumeist die folgenden Wechselwirkungen nicht berücksichtigen:
  - Wechselwirkungen mit anderen Programmen (siehe z.B. wechselseitiger Ausschluss)
  - Wechselwirkungen mit dem Betriebssystem (siehe z.B. Caches)
  - Wechselwirkungen mit der Umgebung (siehe z.B. Interrupts)
  - Wechselwirkungen mit anderen Rechnern (siehe z.B. Synchronisation)
- Durch dynamische Analysen können diese Wechselwirkungen abgeschätzt werden.
- Problem: Wie können die Testläufe sinnvoll ausgewählt werden.

Dimensionierung der Rechenleistungen

- Aufstellen der Worst-Case Analyse:
  - Rechenaufwand für bekannte periodische Anforderungen
  - Rechenaufwand für erwartete sporadische Anforderungen
  - Zuschlag von 100% oder mehr zum Abfangen von Lastspitzen
- Unterschied zu konventionellen Systemen:
  - keine maximale Auslastung des Prozessors
  - keine Durchsatzoptimierung
  - Abläufe sollen determiniert abschätzbar sein

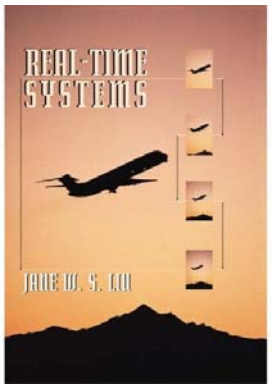
Kapitel 5

Echtzeitbetriebssysteme

Inhalt

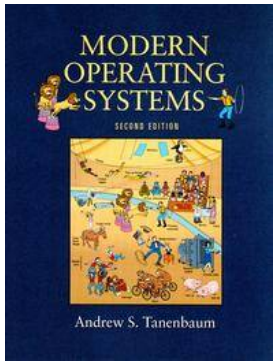
- Grundlagen
- Betrachtung diverser Betriebssysteme:
  - Domänenspezifische Betriebssysteme:
    - OSEK
    - TinyOS
  - Klassische Echtzeitbetriebssysteme
    - QNX
    - VxWorks
    - PikeOS
  - Linux- / Windows-Echtzeitvarianten
    - RTLinux/RTAI
    - Linux Kernel 2.6
    - Windows CE

Literatur



Jane W. S. Liu, Real-Time Systems, 2000

Dieter Zöbel, Wolfgang Albrecht: Echtzeitsysteme: Grundlagen und Techniken, 1995



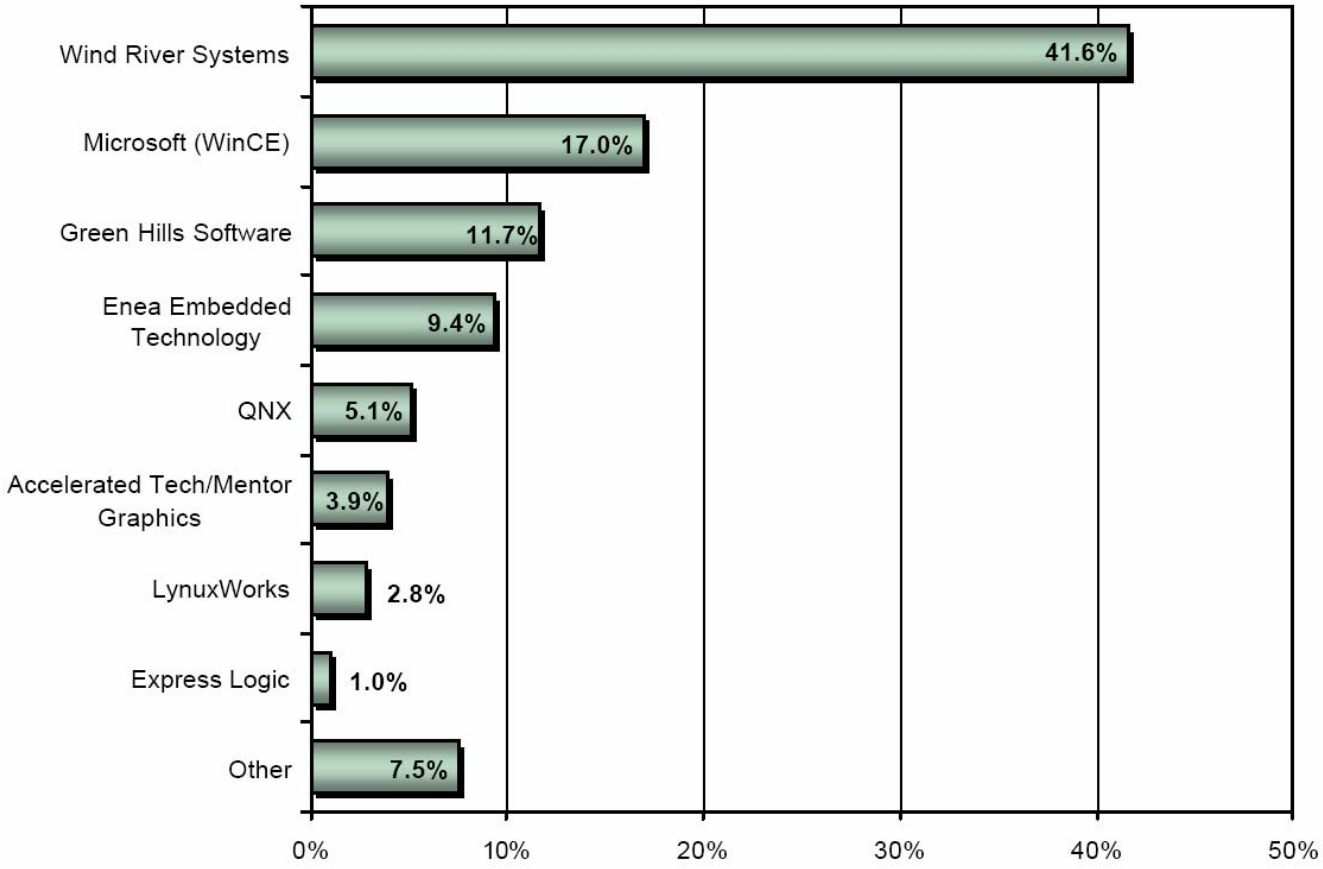
Andrew S. Tanenbaum: Modern Operating Systems, 2001

Arnd Heursch et al.: Time-critical tasks in Linux 2.6, 2004
http://inf3-www.informatik.unibw-muenchen.de/research/linux/hannover/automation_conf04.pdf

Interessante Links

- http://www.mnis.fr/en/support/doc/rtos/
- http://aeolean.com/html/RealTimeLinux/RealTimeLinuxReport-2.0.0.pdf
- http://www.osek-vdx.org/
- http://www.qnx.com/
- http://www.windriver.de
- http://www.fsmlabs.com
- http://www.rtai.org
- http://www.tinyos.net/

Marktaufteilung (Stand 2004)



Marktanteil am Umsatz, Gesamtvolumen 493 Mio. Dollar, Quelle: The Embedded Software Strategic Market Intelligence Program 2005

Anforderungen an Echtzeitbetriebssysteme

- Echtzeitbetriebssysteme unterliegen anderen Anforderungen als Standardbetriebssysteme:
  - stabiler Betrieb rund um die Uhr
  - definierte Reaktionszeiten
  - parallele Prozesse
  - Unterstützung von Mehrprozessorsystemen
  - schneller Prozesswechsel (geringer Prozesskontext)
  - echtzeitfähige Unterbrechensbehandlung
  - echtzeitfähiges Scheduling
  - echtzeitfähige Prozesskommunikation
  - umfangreiche Zeitdienste (absolute, relative Uhren, Weckdienste)
  - einfaches Speichermanagement

Fortsetzung

- Unterstützung bei der Ein- und Ausgabe
  - vielfältigste Peripherie
  - direkter Zugriff auf Hardware-Adressen und -Register durch den Benutzer
  - Treiber in Benutzerprozessen möglichst schnell und einfach zu implementieren
  - dynamisches Binden an den Systemkern
  - direkte Nutzung DMA
  - keine mehrfachen Puffer: direkt vom Benutzerpuffer auf das Gerät
- Einfachste Dateistrukturen
- Protokoll für Feldbus oder LAN-Bus, möglichst hardwareunterstützt
- Aufteilung der Betriebssystemfunktionalität in optionale Komponenten (Skalierbarkeit)

Echtzeitbetriebssysteme

Kriterien zur Beurteilung

Beurteilung von Echtzeitbetriebssystemen

- Folgende Aspekte werden wir genauer betrachten:
  - Schedulingverfahren
  - Prozessmanagement
  - Speicherbedarf (Footprint)
  - Garantierte Reaktionszeiten

Schedulingverfahren

- Fragestellung:
  - Welche Konzepte sind für das Scheduling von Prozessen verfügbar?
  - Gibt es Verfahren für periodische Prozesse?
  - Wie wird dem Problem der Prioritätsinversion begegnet?
  - Wann kann eine Ausführung unterbrochen werden?

Arten von Betriebssystemen

- Betriebsysteme werden in drei Klassen unterteilt:
  - Betriebssysteme mit **kooperativen Scheduling**: es können verschiedene Prozesse parallel ausgeführt werden. Der Dispatcher kann aber einem Prozess den Prozessor nicht entziehen, vielmehr ist das Betriebssystem auf die Kooperation der Prozesse angewiesen (z.B. Windows 95/98/ME)
  - Betriebssysteme mit **präemptiven Scheduling**: einem laufenden Prozess kann der Prozessor entzogen werden, falls sich der Prozess im Userspace befindet. (z.B. Linux, Windows 2000/XP)
  - **Präemptible Betriebssysteme**: der Prozessor kann dem laufenden Prozess jederzeit entzogen werden, auch wenn sich dieser im Kernelkontext ausgeführt wird.

$\Rightarrow$ Echtzeitsysteme müssen präemptibel sein.

Prozessmanagement

- Bewertung eines Betriebssystems nach:
  - Beschränkung der Anzahl von Prozessen
  - Möglichkeiten zur Interprozesskommunikation
  - Kompatibilität der API mit Standards (z.B. POSIX) zur Erhöhung der Portabilität

Speicherbedarf

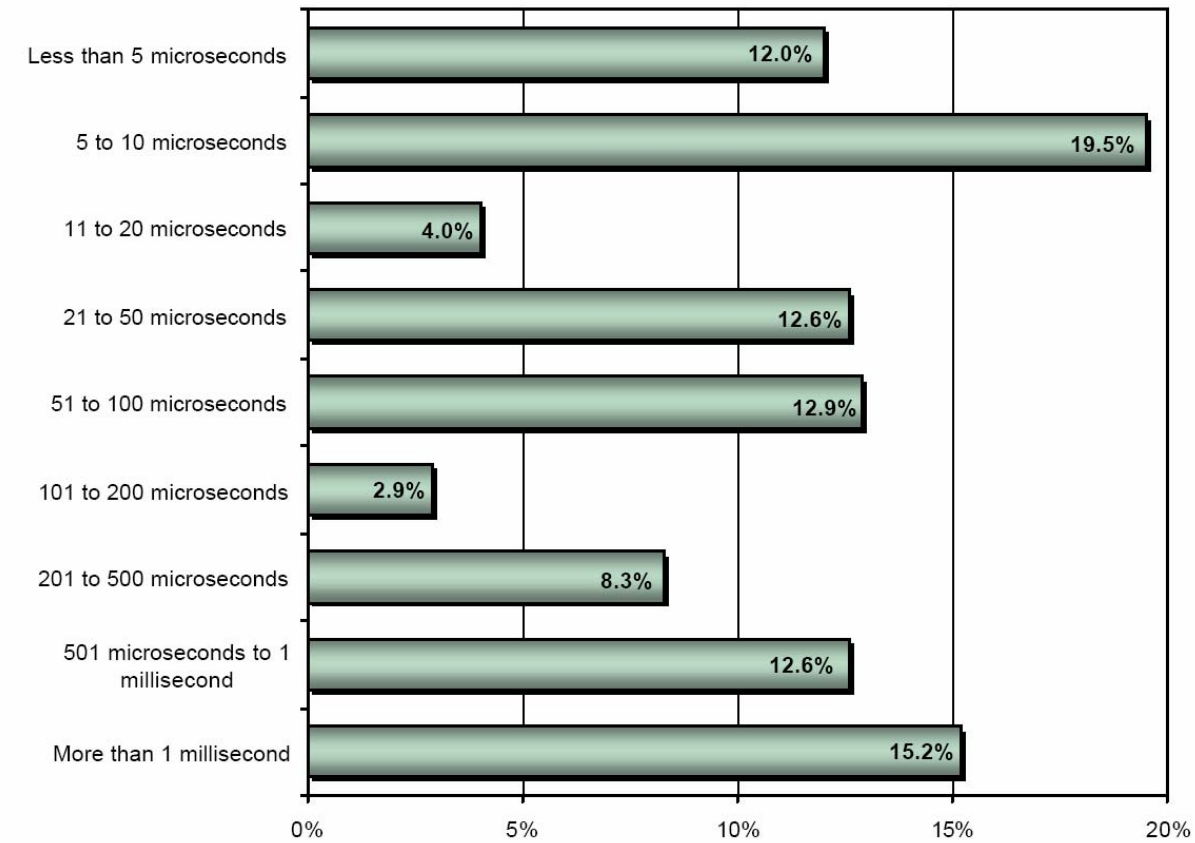
- Echtzeitbetriebssysteme werden auf sehr unterschiedlicher Hardware ausgeführt
  - Der verfügbare Speicher variiert sehr stark.
  - Typische Betriebssystemfunktionalitäten (z.B. Dateisysteme, graphische Oberfläche) werden oft gar nicht benötigt.

  ⇒ Echtzeitsysteme müssen aus diesen Gründen skalierbar sein:
  - Möglichkeit zur Auswahl einzelner Module entsprechend den Anforderungen an die Funktionalität der Anwendung.
  - Entscheidend ist der **minimale Speicherbedarf** (**Footprint**).

Reaktionszeiten

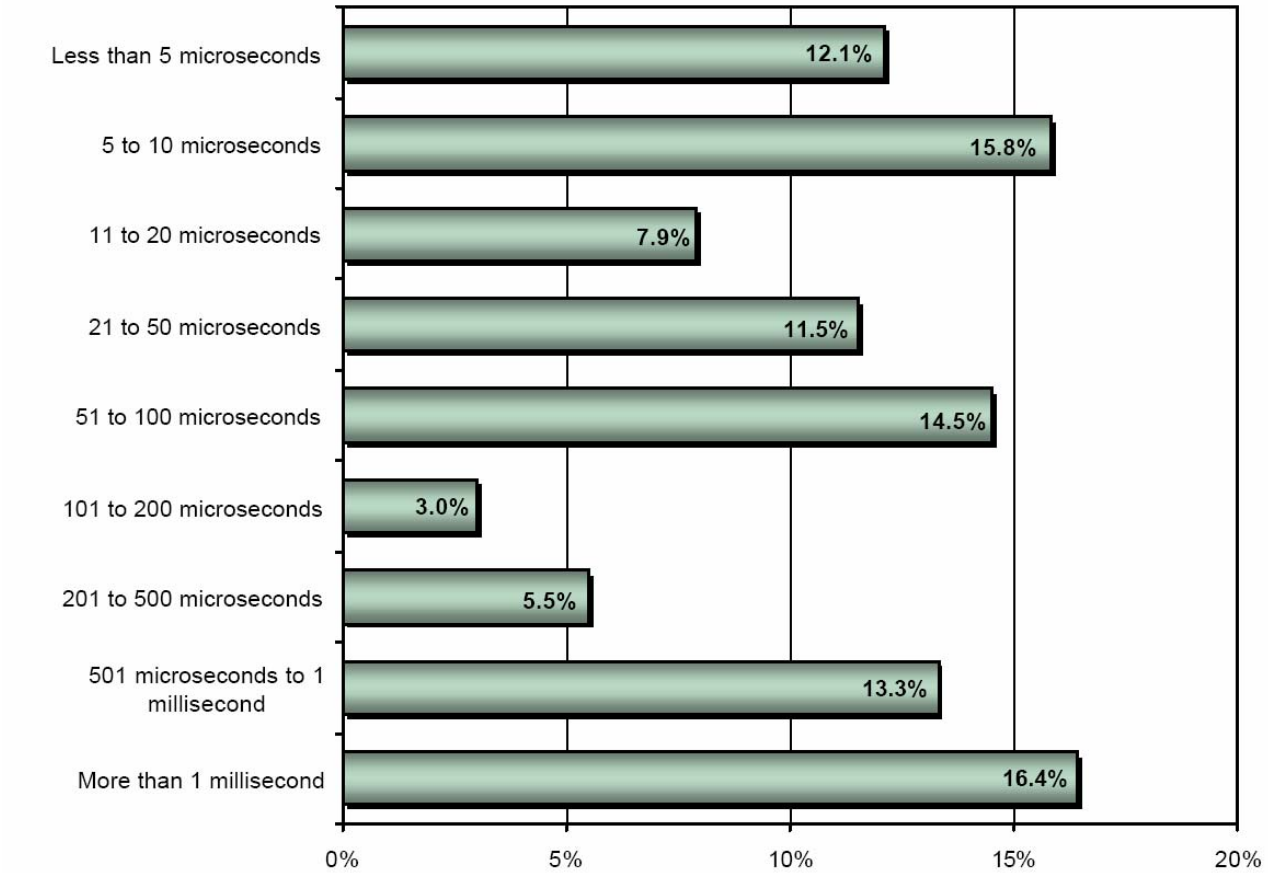
- Die Echtzeitfähigkeit wird durch die Messung folgender Zeiten bestimmt:
  - **Unterbrechungsantwortzeiten** (**interrupt latency**): der Zeitraum zwischen dem Auftreten einer Unterbrechung und der Ausführung des ersten Befehls der dazugehörigen Unterbrechungsbehandlungsroutine
  - **Schedulinglatenz** (**scheduling latency**): Zeit von der Ausführung des letzten Befehls des Unterbrechungsbehandlers bis zur Ausführung der ersten Instruktion des Prozesses, der durch das Auftreten des Unterbrechung in den bereiten Zustand wechselt.
  - Zeiten für einen **Kontextwechsel** (**context switch latency**): Zeit von der Ausführung des letzten Befehls eines Prozesses im Userspace bis zur Ausführung der ersten Instruktion des nächsten Prozesses im Userspace.

Anforderungen an Unterbrechungsantwortzeiten



Typische Anforderungen an Antwortzeiten, Quelle: The Embedded Software Strategic Market Intelligence Program 2005

Anforderungen an Kontextwechselzeiten



Typische Anforderungen an den Kontextwechsel, Quelle: The Embedded Software Strategic Market Intelligence Program 2005

Echtzeitbetriebssysteme

OSEK

Hintergrund

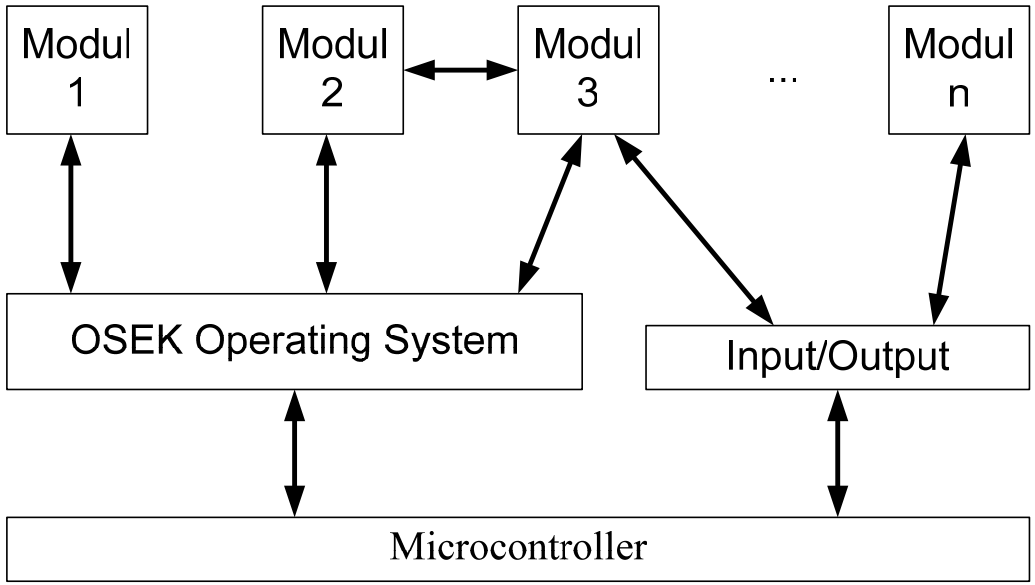
- Gemeinschaftsprojekt der deutschen Automobilindustrie (u.a. BMW, DaimlerChrysler, VW, Opel, Bosch, Siemens)
- OSEK: **O**ffene **S**ysteme und deren Schnittstellen für die **E**lektronik im **K**raftfahrzeug
- Ziel: Definition einer Standard-API für Echtzeitbetriebssysteme
- Standard ist frei verfügbar (http://www.osek-vdx.org), aber keine freien Implementierungen.
- Es existieren ebenso Ansätze für ein zeitgesteuertes Betriebssystem (OSEKTime), sowie eine fehlertolerante Kommunikationsschicht.

Anforderungen

- Designrichtlinien bei der Entwicklung von OSEK:
  - harte Echtzeitanforderungen
  - hohe Sicherheitsanforderungen an Anwendungen
  - hohe Anforderungen an die Leistungsfähigkeit
  - typische: verteilte Systeme mit unterschiedlicher Hardware (v.a. Prozessoren)

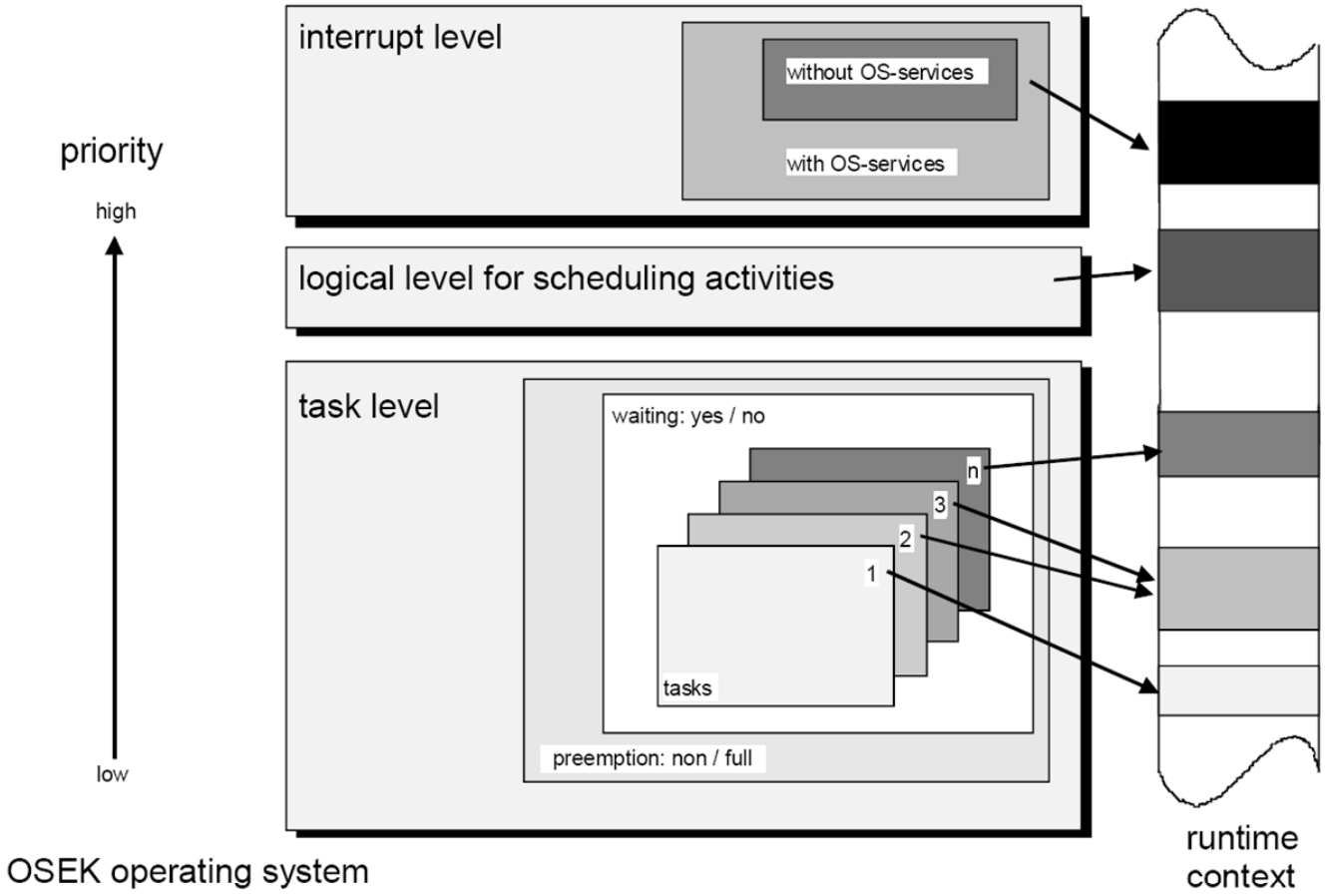
  $\Rightarrow$ typische Anforderungen von Echtzeitsystemen
- Weitere Ziele:
  - Skalierbarkeit
  - einfache Konfigurierbarkeit des Betriebssystems
  - Portabilität der Software
  - Statisch allokiertes Betriebssystem

OSEK Architektur



- Die Schnittstelle zwischen den einzelnen Anwendungsmodulen ist zur Erhöhung der Portierbarkeit standardisiert. Die Ein- und Ausgabe ist ausgelagert und wird nicht näher spezifiziert.

Ausführungsebenen in OSEK



Scheduling und Prozesse in OSEK

- Scheduling:
  - ausschließlich Scheduling mit statischen Prioritäten.
- Prozesse:
  - OSEK unterscheidet zwei verschiedene Arten von Prozessen:
    1. Basisprozesse
    2. Erweiterte Prozesse: haben die Möglichkeit über einen Aufruf der Betriebssystemfunktion waitEvent() auf externe asynchrone Ereignisse zu warten und reagieren.
  - Der Entwickler kann festlegen, ob ein Prozess unterbrechbar oder nicht unterbrechbar ist.
  - Es existieren somit vier Prozesszustände in OSEK: running, ready, waiting, suspended.

Betriebssystemklassen

- Der OSEK-Standard unterscheidet vier unterschiedliche Klassen von Betriebsystemen. Die Klassifizierung erfolgt dabei nach der Unterstützung:
  1. von mehrmaligen Prozessaktivierungen (einmalig oder mehrfach erlaubt)
  2. von Prozesstypen (nur Basisprozesse oder auch erweiterte Prozesse)
  3. mehreren Prozessen der selben Priorität
- Klassen:
  - BCC1: nur einmalig aktivierte Basisprozesse unterschiedlicher Priorität werden unterstützt.
  - BCC2: wie BCC1, allerdings Unterstützung von mehrmalig aufgerufenen Basisprozessen, sowie mehreren Basisprozessen gleicher Priorität.
  - ECC1: wie BCC1, allerdings auch Unterstützung von erweiterten Prozessen
  - ECC2: wie ECC1, allerdings Unterstützung von mehrmalig aufgerufenen Prozessen, sowie mehreren Prozessen gleicher Priorität.
- Die Implementierung unterscheidet sich vor allem in Bezug auf den Scheduler.

Unterbrechungsbehandlung

- In OSEK wird zwischen zwei Arten von Unterbrechungsbehandlern unterschieden:
  - ISR Kategorie 1: Der Behandler benutzt keine Betriebssystemfunktionen.
    - typischerweise die schnellsten und höchstpriorisierten Unterbrechungen.
    - Im Anschluss der Behandlung wird der unterbrochene Prozess fortgesetzt.
  - ISR Kategorie 2: Die Behandlungsroutine wird durch das Betriebssystem unterstützt, dadurch sind Aufrufe von Betriebssystemfunktionen erlaubt.
    - Falls ein Prozess unterbrochen wurde, wählt der Scheduler nach Beendigung der ISR den nächsten auszuführenden Prozess.

Prioritätsinversion

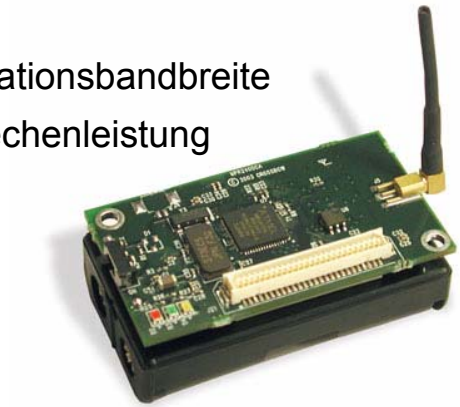
- Zur Vermeidung von Prioritätsinversion und Verklemmungen schreibt OSEK ein Immediate Priority Ceiling Protokoll vor:
  - Jeder Ressource wird eine Grenze (Maximum der Priorität der Prozesse, die die Ressource verwenden) zugewiesen.
  - Falls ein Prozess eine Ressource anfordert, wird die aktuelle Priorität des Prozesses auf die entsprechende Grenze angehoben.
  - Bei Freigabe fällt der Prozess auf die ursprüngliche Priorität zurück.

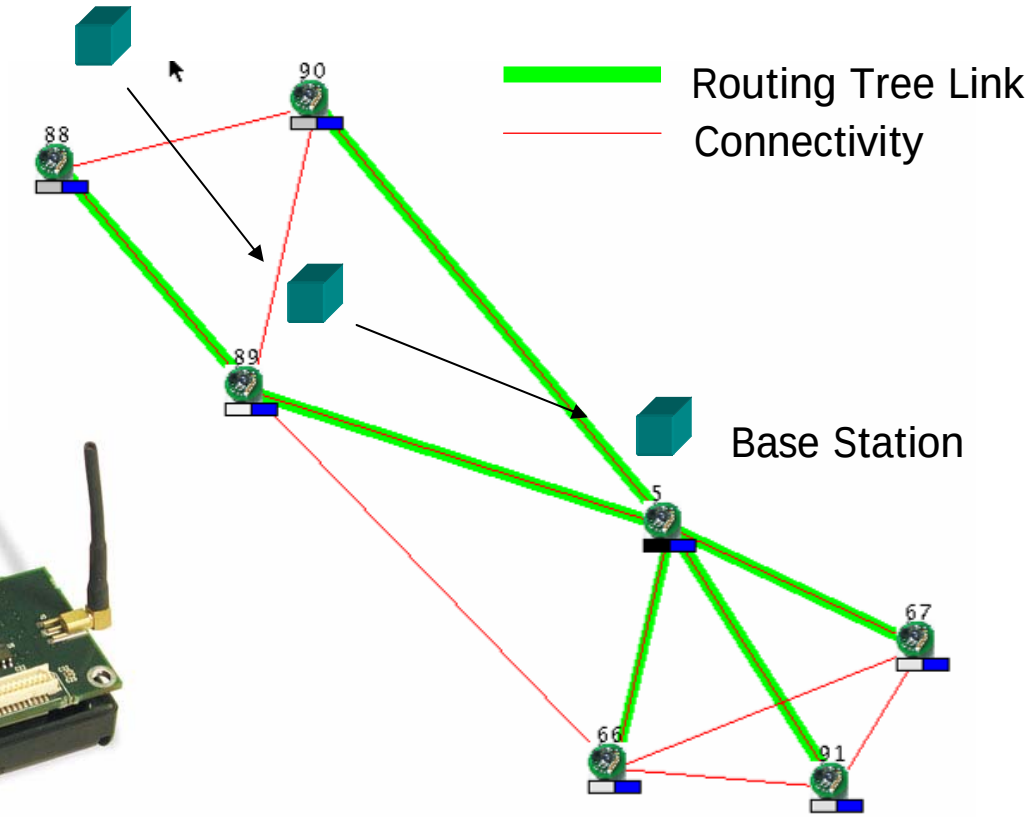
Echtzeitbetriebssysteme

TinyOS

Einsatzgebiet: AdHoc-Sensornetzwerke

- Begriff Smart-Dust: Viele kleine Sensoren überwachen die Umgebung
- Ziele: robuste und flächendeckende Überwachung
- Probleme:
  - eingeschränkte Lebensdauer (Batterie)
  - eingeschränkter Speicherplatz
  - geringe Kommunikationsbandbreite
  - geringe Rechenleistung

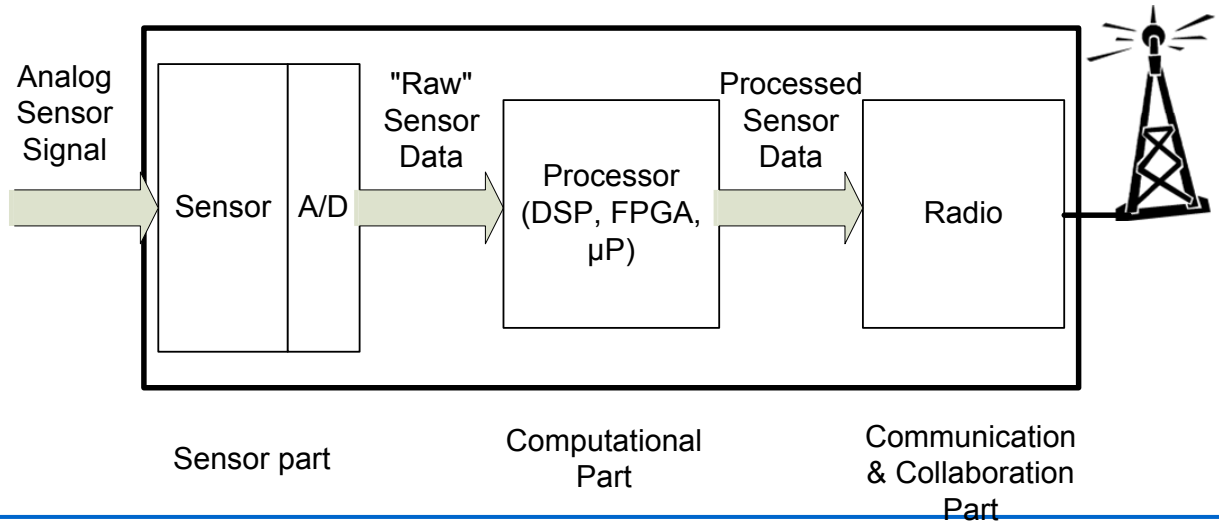




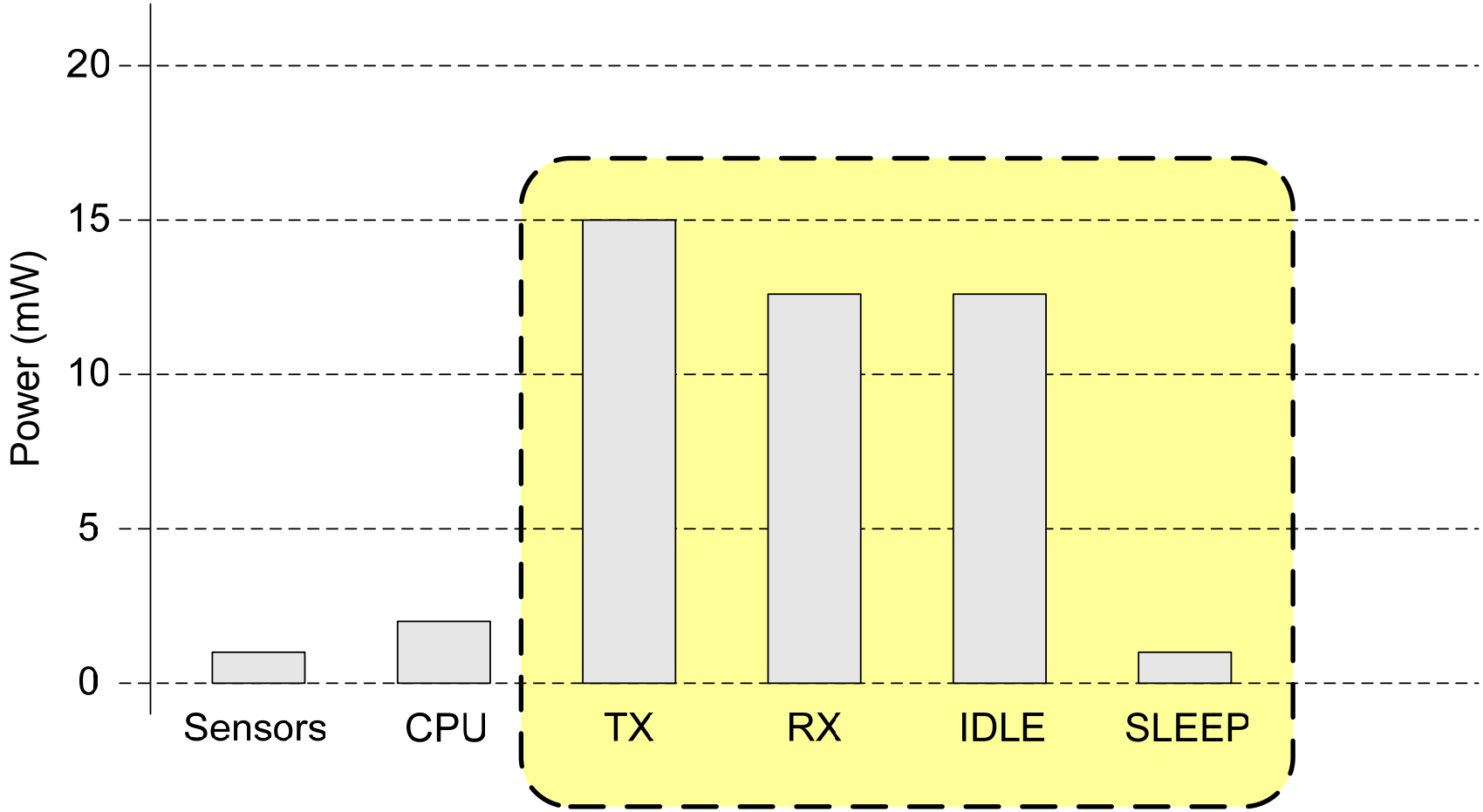
Quelle: http://tinyos.millennium.berkeley.edu

Hardware

- CPU: 4MHz, 8Bit, 512 Byte Ram
- Flash-Speicher: 128 kByte
- Funkmodul: 2,4 MHz, 250 kbps
- Diverse Sensormodule: z.B. Digital/Analog, Licht, Feuchtigkeit, Druck

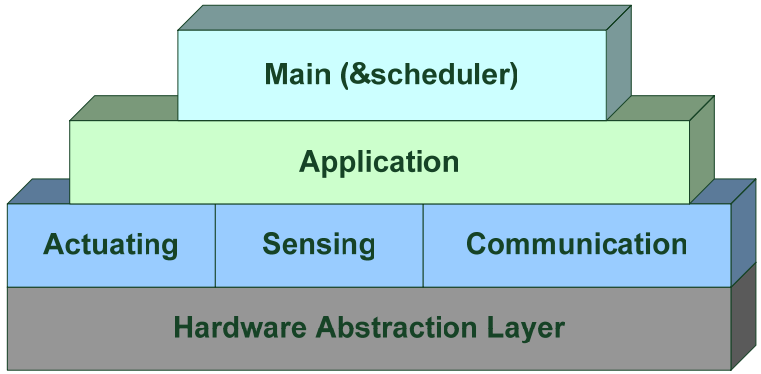


Stromverbrauch

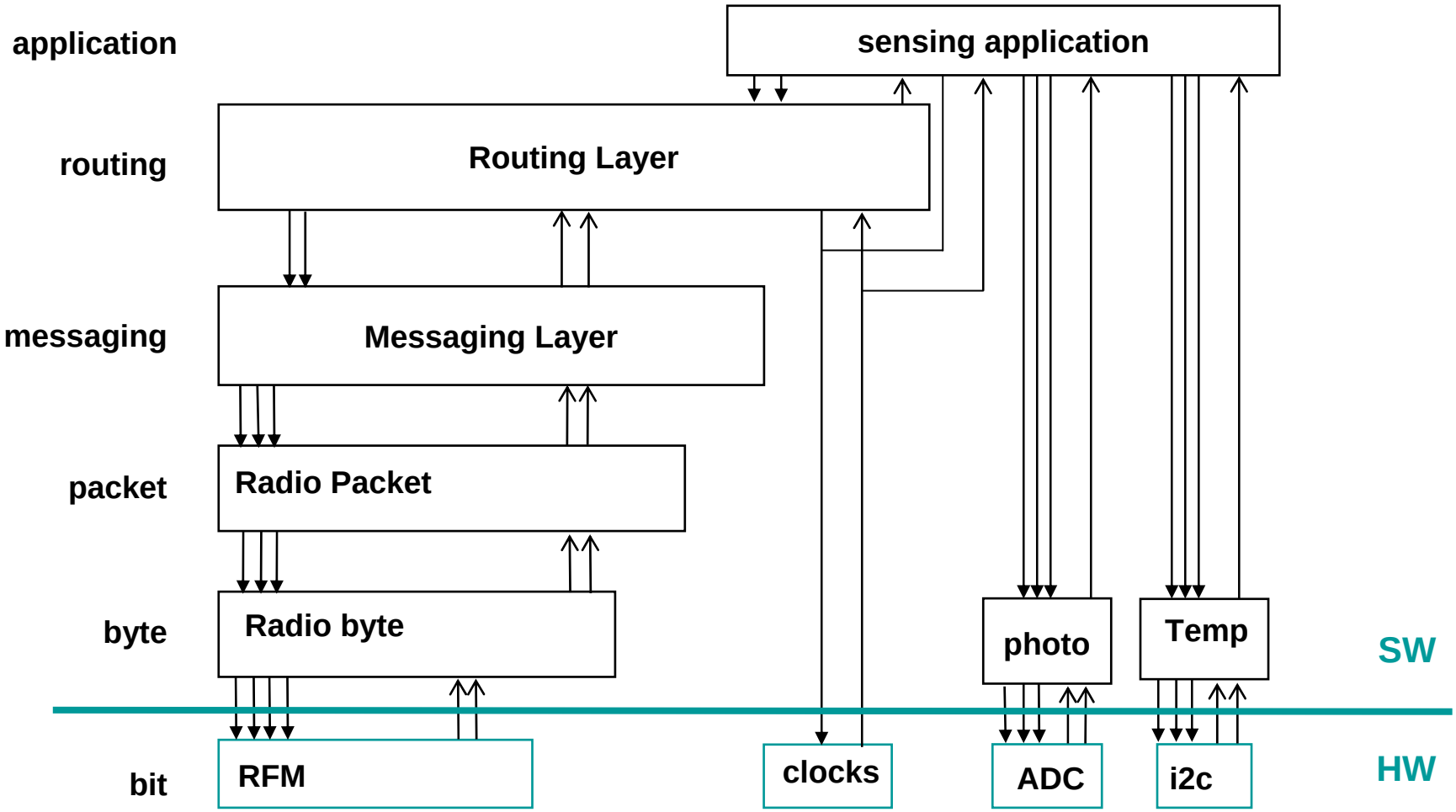


TinyOS

- TinyOS ist kein wirkliches Betriebssystem im traditionellen Sinn, eher ein anwendungs-spezifisches Betriebssystem
  - keine Trennung der Anwendung vom OS ⇒ Bei Änderung der Anwendung muss komplettes Betriebssystem neu geladen werden.
  - kein Kernel, keine Prozesse, keine Speicherverwaltung
  - Es existiert nur ein Stack (single shared stack)
- Ereignisbasiertes Ausführungsmodell
- Nebenläufigkeitskonzept:
  - Aufgaben können in unterschiedlichen Kontext ausgeführt werden:
  - Vordergrund: Unterbrechungsereignisse
  - Hintergrund: Tasks
  - Prozesse können durch Ereignisse, nicht jedoch durch andere Prozesse unterbrochen werden.
  - Scheduling für Tasks: Fifo
- Implementierung erfolgt in NesC (Erweiterung von C)
- Statische Speicherallokation



TinyOS - Architektur



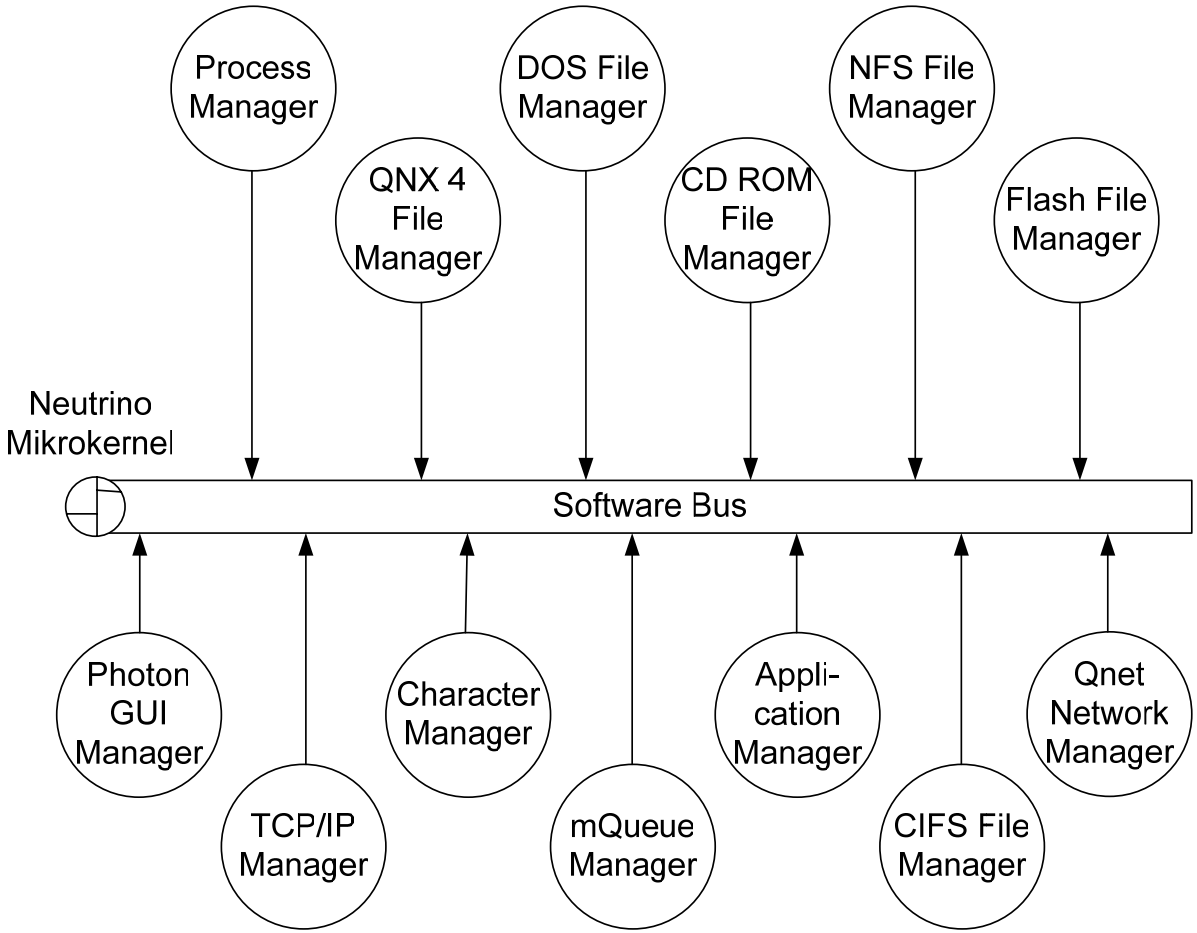
Echtzeitbetriebssysteme

QNX

Einführung

- Geschichte:
  - 1980 entwickeln Gordon Bell und Dan Dodge ein eigenes Echtzeitbetriebssystem mit Mikrokernel.
  - QNX orientiert sich nicht an Desktopsystemen und breitet sich sehr schnell auf dem Markt der eingebetteten Systeme aus.
  - Ende der 90er wird der Kernel noch einmal komplett umgeschrieben, um den POSIX-Standard zu erfüllen. $\Rightarrow$ Ergebnis: QNX Neutrino.
- Besonderheiten von QNX
  - stark skalierbar, extrem kleiner Kernel (bei Version 4.24 ca.11kB)
  - Grundlegendes Konzept: Kommunikation erfolgt durch Nachrichten

QNX Architektur



Neutrino Microkernel

- Der Mikrokernel in QNX enthält nur die notwendigsten Elemente eines Betriebssystems:
  - Umsetzung der wichtigsten POSIX Elemente
    - POSIX Threads
    - POSIX Signale
    - POSIX Thread Synchronisation
    - POSIX Scheduling
    - POSIX Timer
  - Funktionalität für Nachrichten
- Eine ausführliche Beschreibung findet sich unter
  http://www.qnx.com/developers/docs/momentics621_docs/neutrino/sys_arch/kernel.html

Prozessmanager

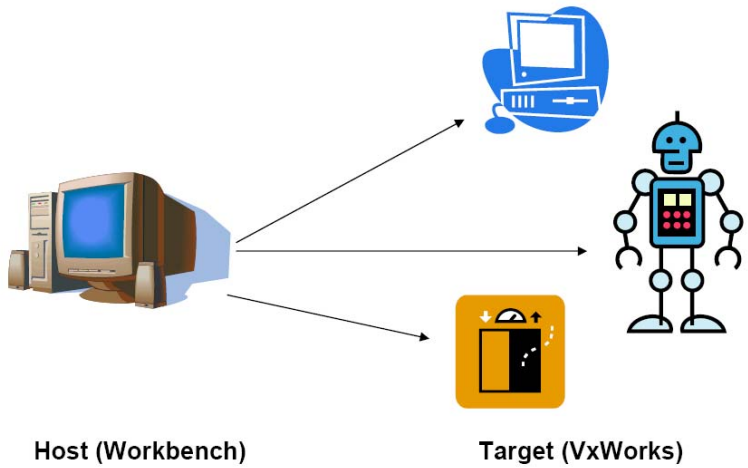
- Als wichtigster Prozess läuft in QNX der Prozessmanager.
- Die Aufgaben sind:
  - Prozessmanagement:
    - Erzeugen und Löschen von Prozessen
    - Verwaltung von Prozesseigenschaften
  - Speichermanagement:
    - Bereitstellung von Speicherschutzmechanismen,
    - von gemeinsamen Bibliotheken
    - und POSIX Primitiven zu Shared Memory
  - Pfadnamenmanagement
- Zur Kommunikation zwischen und zur Synchronisation von Prozessen bietet QNX Funktionalitäten zum Nachrichtenaustausch an.

Echtzeitbetriebssysteme

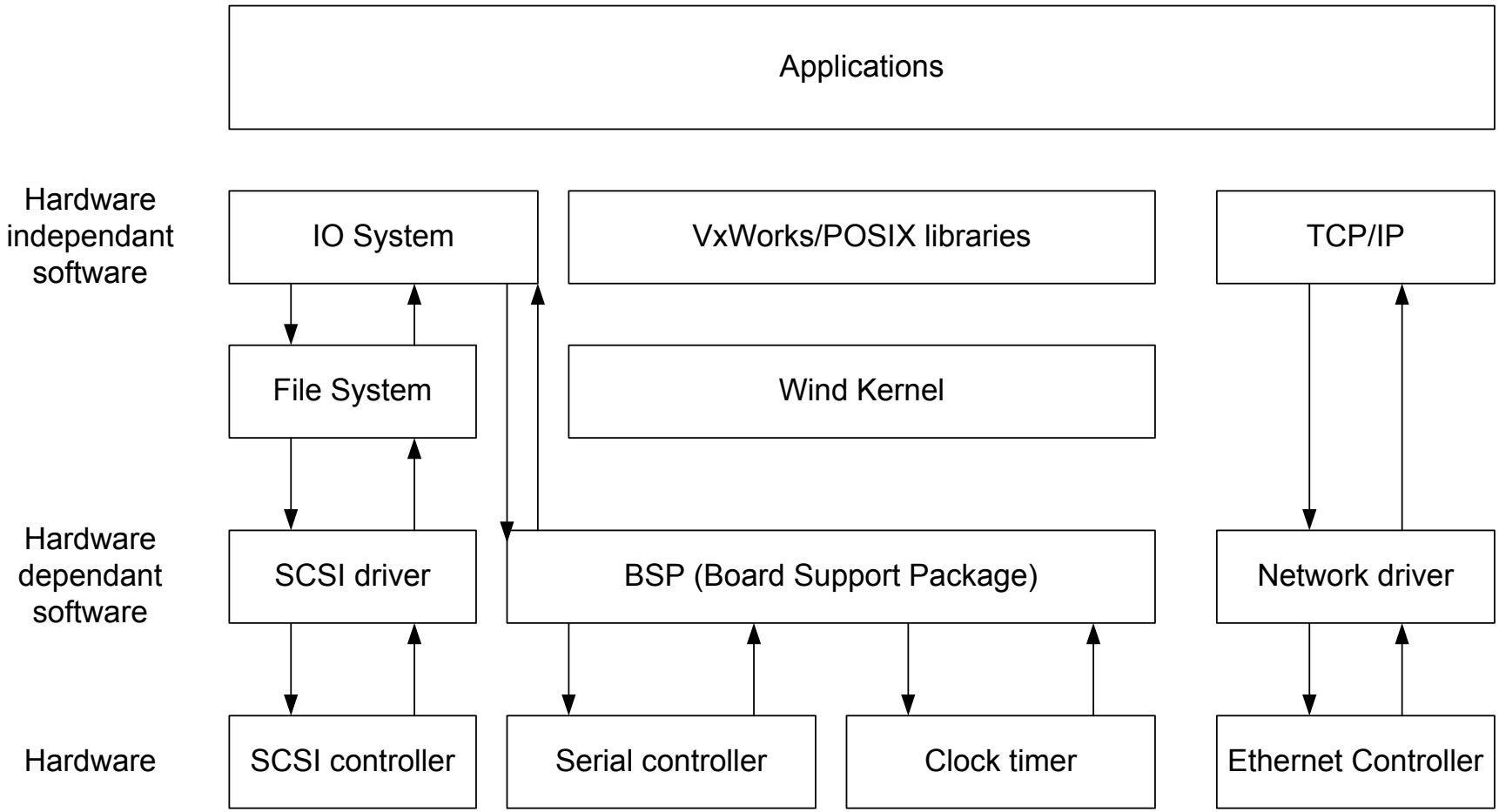
VxWorks

Eigenschaften

- Host-Target-Entwicklungssystem
- Eigene Entwicklungsumgebung Workbench mit Simulationsumgebung und integriertem Debugger basierend auf Eclipse
- Zielplattformen der Workbench 2.0: VxWorks, Linux Kernel 2.4/2.6
- Auf der Targetshell wird auch ein Interpreter ausgeführt ⇒ C-Code kann direkt in die Shell eingegeben werden
- Kernel kann angepasst werden, allerdings muss der Kernel dazu neu kompiliert werden
- Marktführer im Bereich der Echtzeitbetriebssysteme



Architektur



Prozessmanagement

- **Schedulingverfahren**: Es werden nur die beiden Verfahren FIFO und RoundRobin angeboten. Ein Verfahren für periodische Prozesse ist nicht verfügbar.
- **Prioritäten**: Die Prioritäten reichen von 0 (höchste Priorität) bis 255.
- **Uhrenauflösung**: Die Uhrenauflösung kann auf eine maximale Rate von ca. 30 KHz (abhängig von Hardware) gesetzt werden.
- **Prozessanzahl**: Die Anzahl der Prozesse ist nicht beschränkt (aber natürlich abhängig vom Speicherplatz)
- **API**: VxWorks bietet zum Management von Prozessen eigene Funktionen, sowie POSIX-Funktionen an.

Interprozesskommunikation und Speichermanagement

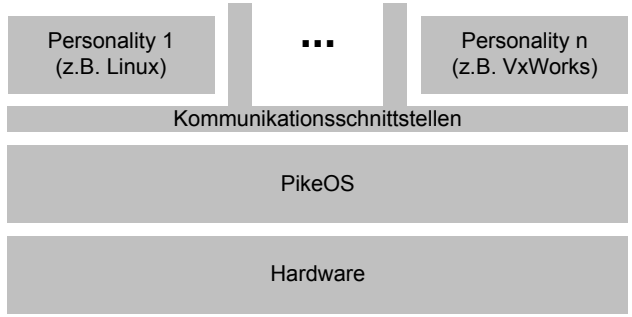
- Zur Interprozesskommunikation werden folgende Konzepte unterstützt:
  - Semaphor
  - Mutex (mit Prioritätsvererbung)
  - Nachrichtenwartschlangen
  - Signale
- Seit Version 6.0 wird zudem Speichermanagement angeboten:
  - Der Entwickler kann Benutzerprozesse mit eigenem Speicherraum entwickeln.
  - Bisher: nur Threads im Kernel möglich.

Echtzeitbetriebssysteme

PikeOS

PikeOS: Betriebssystem mit Paravirtualisierung

- Idee: Virtualisierung der Hardware – jede Partition (Personality) verhält sich als hätte sie eine eigene CPU zur Verfügung
- Mehrere Betriebssysteme können auf der gleichen CPU nebenläufig ausgeführt werden.



Architektur

- Die Speicherbereiche, sowie CPU-Zeiten der einzelnen Partitionen werden statisch während der Implementierung festgelegt.
- Durch die Partitionierung ergeben sich diverse Vorteile:
  - Bei einer Zertifizierung muss nur der sicherheitskritische Teil des Gesamtsystems zertifiziert werden.
  - Reduzierung der Steuergeräte durch Zusammenführung der Funktionalitäten mehrerer Steuergeräte
  - Echtzeitkomponenten können einfacher von nicht-kritischen Komponenten getrennt werden – Nachweis der Fristeneinhaltung wird einfacher

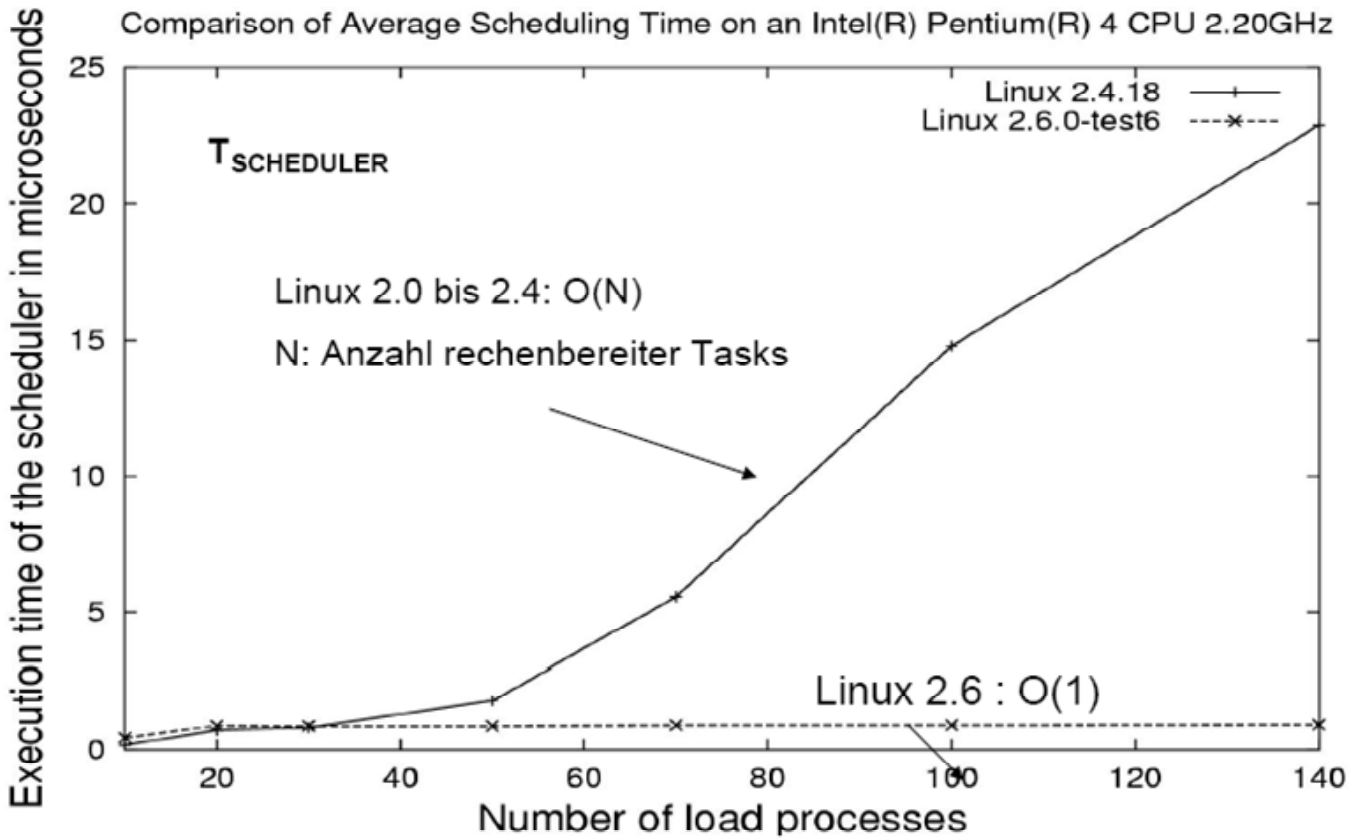
Echtzeitbetriebssysteme

Linux Kernel 2.6

Bestandsaufnahme

- Für die Verwendung von Linux Kernel 2.6 in Echtzeitsystemen spricht:
  - die Existenz eines echtzeitfähigen Schedulingverfahrens (prioritätenbasiertes Scheduling mit FIFO oder RoundRobin bei Prozessen gleicher Priorität)
  - die auf 1 ms herabgesetzte Zeitauflösung der Uhr (von 10ms in Kernel 2.4)
- Gegen die Verwendung spricht:
  - die Ununterbrechbarkeit des Kernels.

Vergleich Schedulerlaufzeiten Kernel 2.4/2.6



Quelle: A. Heursch

Unterbrechbarkeit des Kernels

- Im Kernel ist der **Preemptible Kernel Patch** als Konfigurationsoption enthalten ⇒ Erlaubt die Unterbrechung des Kernels.
- **Problem**: Existenz einer Reihe von kritischen Bereichen, die zu langen Verzögerungszeiten führen.
- **Low Latency Patches** helfen bei der Optimierung, aber harte Echtzeitanforderungen können nicht erfüllt werden.
- Weitere Ansätze: z.B. Verwendung von binären Semaphoren (Mutex) anstelle von generellen Unterbrechungssperren, Verhinderung von Prioritätsinversion durch geeignete Patches, siehe Paper von A. Heursch

Speichermanagement

- Linux unterstützt Virtual Memory
- Die Verwendung von Virtual Memory führt zu zufälligen und nicht vorhersagbaren Verzögerung, falls sich eine benötigte Seite nicht im Hauptspeicher befindet.
  $\Rightarrow$ Die Verwendung von Virtual Memory in Echtzeitanwendungen ist nicht sinnvoll.
- Vorgehen: Zur Vermeidung bietet Linux die Funktionen `mlock()` und `mlockall()` zum **Pinning** an.
- Pinning bezeichnet die Verhinderung des Auslagerns eines bestimmten Speicherbereichs oder des kompletten Speichers eines Prozesses.

Uhrenauflösung

- Die in Linux Kernel 2.6 vorgesehene Uhrenauflösung von 1ms ist häufig nicht ausreichend.
- Problemlösung: Verwendung des **High Resolution Timer Patch (hrtimers)**
  - Durch Verwendung des Patches kann die Auflösung verbessert werden.
  - Der Patch erlaubt z.B. die Erzeugung einer Unterbrechung in 3,5 Mikrosekunden von jetzt an.
  - Einschränkung: Zeitliche Angabe muss schon vorab bekannt sein $\Rightarrow$ keine Zeitmessung möglich
  - Gründe für die hrtimers-Lösung findet man unter: http://www.kernel.org/git/?p=linux/kernel/git/torvalds/linux-2.6.git;a=blob;f=Documentation/hrtimers.txt

RTLinux/RTAI

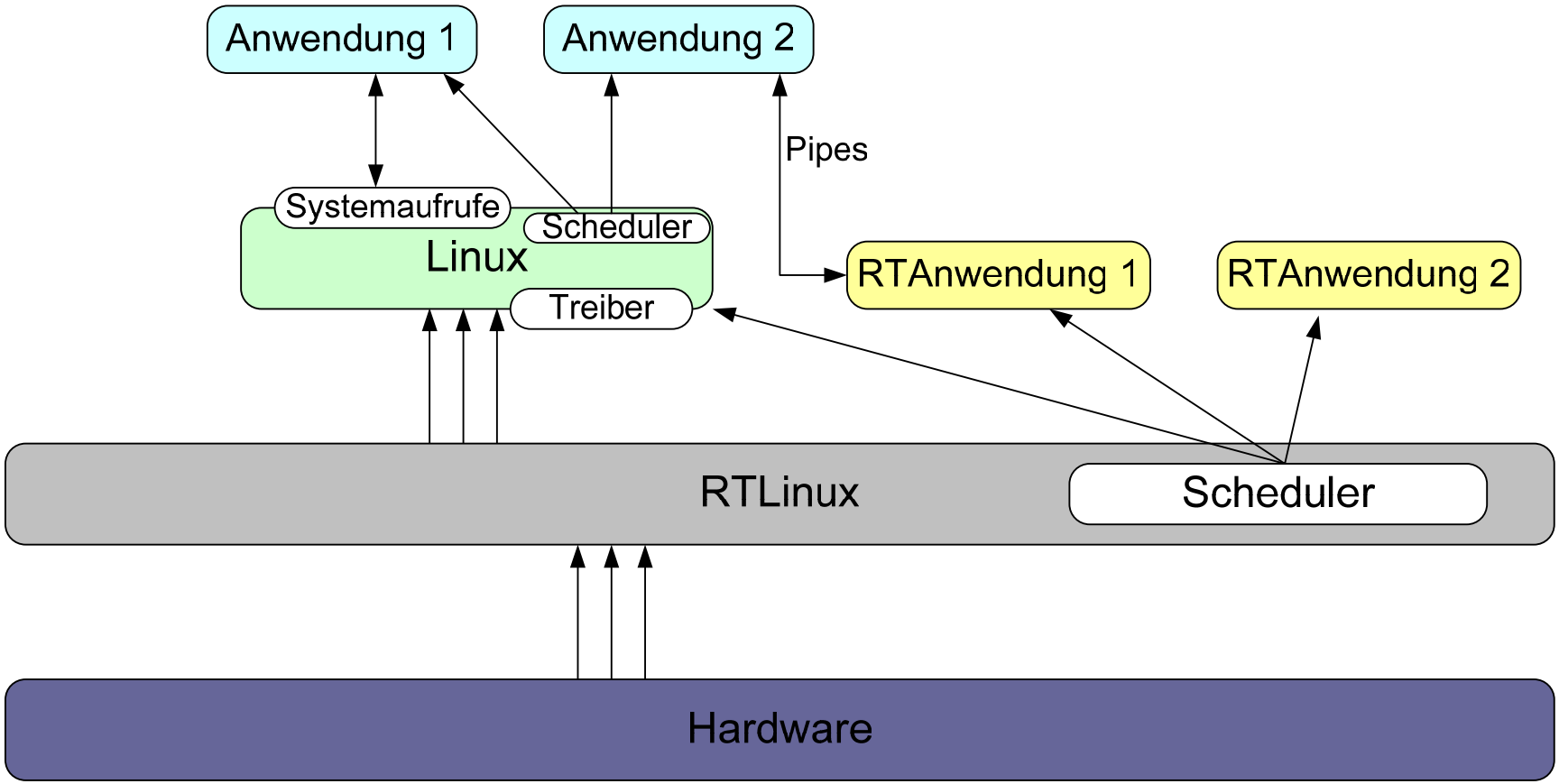
Motivation

- Aus diversen Gründen ist die Verwendung von Linux in Echtzeitsystemen erstrebenswert:
  - Linux ist weitverbreitet
  - Treiber sind sehr schnell verfügbar
  - Es existieren viele Entwicklungswerkzeuge $\Rightarrow$ die Entwickler müssen nicht für ein neues System geschult werden.
  - Häufig müssen nur geringe Teile des Codes echtzeitfähig ausgeführt werden.
- **Probleme**:
  - grobgranulare Synchronisation
  - trotz Patches oft zu lange Latenzzeiten
  - Hochpriorisierte Prozesse können durch andere Prozesse mit niedrigerer Priorität blockiert werden, Grund: Hardwareoptimierungsstrategien (z.B. Speichermanagement)
- **Ansatz**: Modifikation von Linux, so dass auch harte Echtzeitanforderungen erfüllt werden.

Ansatz

- Anstelle von Patches wird eine neue Schicht zwischen Hardware und Linux-Kernel eingefügt:
  - Volle Kontrolle der Schicht über Unterbrechungen
  - Virtualisierung von Unterbrechungen (Barabanov, Yodaiken, 1996): Unterbrechungen werden in Nachrichten umgewandelt, die zielgerichtet zugestellt werden.
  - Virtualisierung der Uhr
  - Anbieten von Funktionen zum virtuellen Einschalten und Ausschalten von Unterbrechungen
  - Das Linux-System wird als Prozess mit niedrigster Priorität ausgeführt.

RTLinux Architektur



Unterschiede RTAI/RTLinux

- RTLinux verändert Linux-Kernel-Methoden für den Echtzeiteingriff ⇒ Kernel-Versions-Änderungen haben große Auswirkungen.
- RTAI fügt Hardware Abstraction Layer (HAL) zwischen Hardware und Kernel ein. Hierzu sind nur ca. 20 Zeilen Code am Originalkern zu ändern. HAL selbst umfasst kaum mehr als 50 Zeilen ⇒ Transparenz.
- RTAI ist frei, RTLinux in freier (Privat, Ausbildung) und kommerzieller Version.
- Beide Ansätze verwenden ladbare Kernel Module für Echtzeitprozesse.
- RTAI (mit Variante LXRT) erlaubt auch die Ausführung von echtzeitkritischen Prozessen im User-Space, Vorteil ist beispielsweise der Speicherschutz

Echtzeitbetriebssysteme

Windows CE & Windows Embedded

Eigenschaften

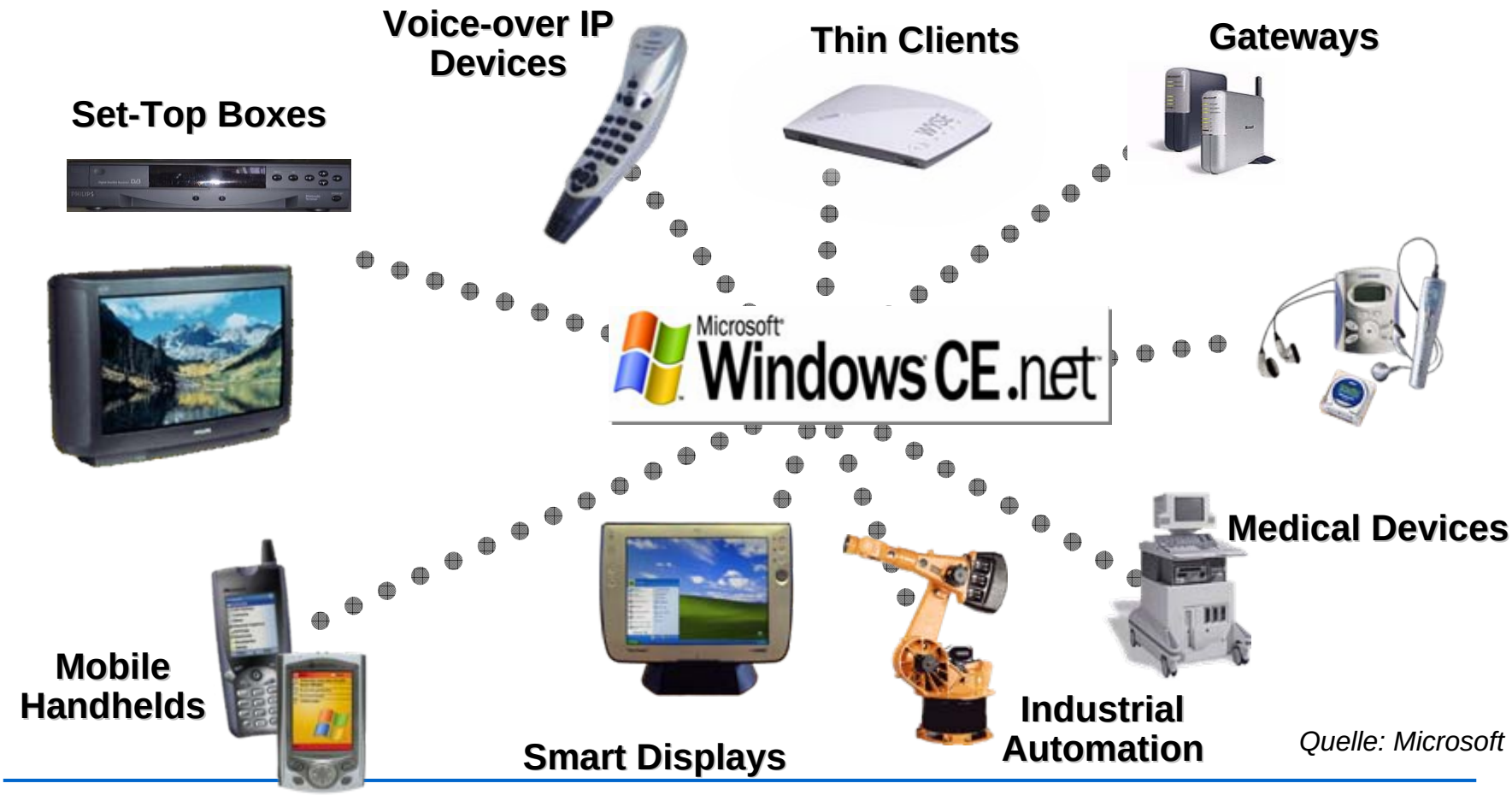
- Windows CE
  - 32-bit, Echtzeitbetriebssystem
  - Unterstützung von Multitasking
  - Stark modularer Aufbau
  - Skalierbar entsprechend der gewünschten Funktionalität
- Windows Embedded
  - „Skalierbares Windows XP“
  - Komponenten von XP können entfernt werden um den benötigten Speicherplatz zu minimieren

Windows CE und Embedded im Vergleich

Windows XP Embedded		Windows CE.net
x86 processors	Processor Support	Multiple processors / power management
Full Win32 API compatibility	Win32 API Compatibility	Requires additional effort
Basic images from 8MB (“Hello World”)	Footprint	Basic images from 350 KB
With 3rd party extensions	Real-time	Native

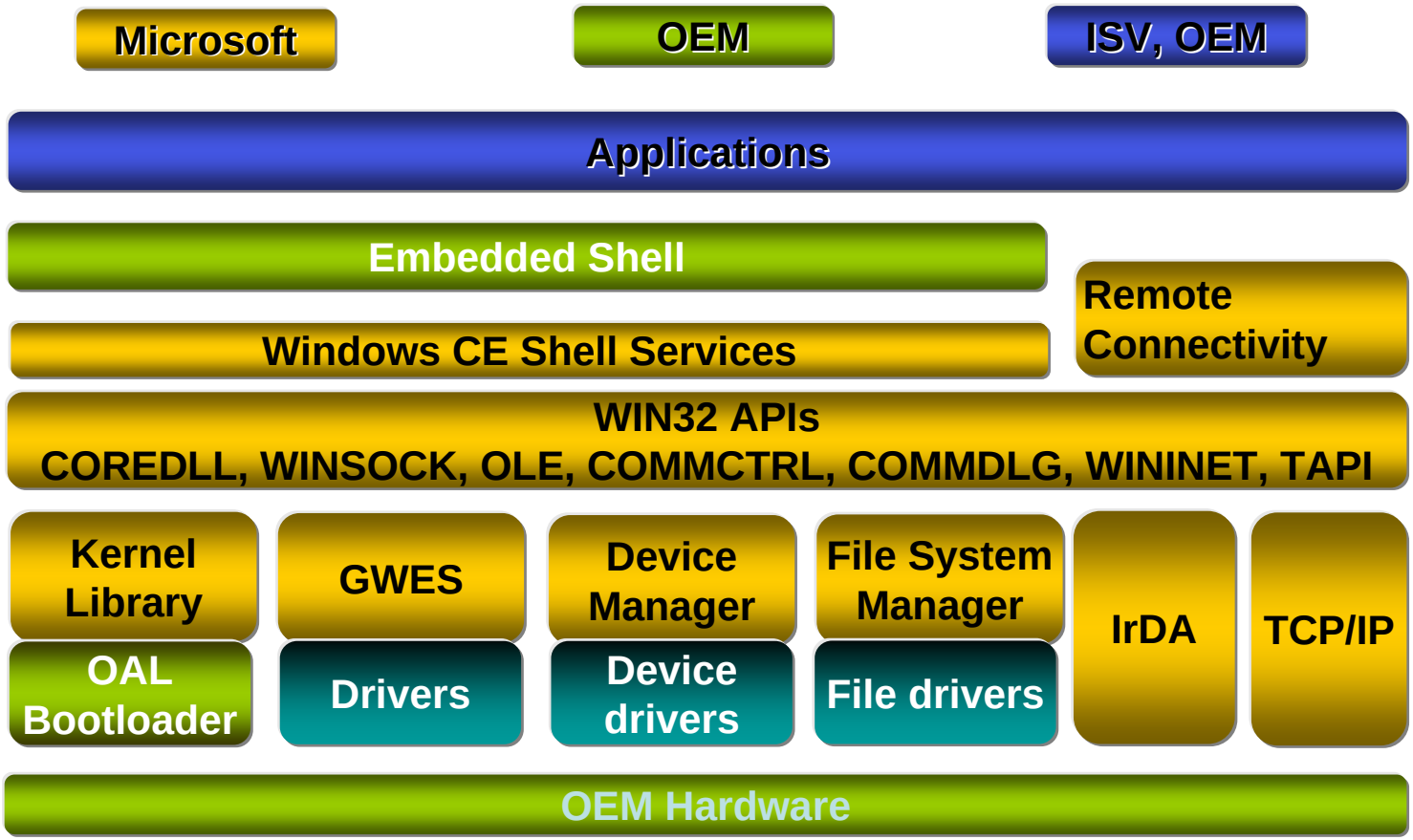
Quelle: Microsoft

Einsatzbereiche



Quelle: Microsoft

Windows CE Architektur



Quelle: Microsoft

Funktionen des Betriebssystemkerns

- Kernel, Speicherverwaltung
  - Shared heap
  - Unterstützung von Watchdogs
  - 64 Systeminterrupts
- Geräteunterstützung
  - Unterstützung diverser Massenspeicher, z.B. USB, Flash,..
- Browser
- Multimedia
  - Diverse Graphiktreiber
  - umfassende Codecunterstützung
- Kryptographie-Funktionen

Echtzeitunterstützung

- Unterstützung verschachtelter Interrupts
- 256 Prioritätslevel
- Thread quantum level control
- Speicherschutz (Pinning) zur Umgehung von Virtual Memory
- Eingebaute Leistungsüberwachungswerkzeuge
- Niedrige ISR/IST Latenz
  - ISR/IST Latenz von 2.8/26.4 Mikrosekunden auf Intel 100MHz Board

Echtzeitbetriebssysteme

Zusammenfassung

Zusammenfassung

- Es gibt kein typisches Echtzeitbetriebssystem da je nach Einsatzbereich die Anforderungen sehr unterschiedlich sind.
- Der minimale Speicherbedarf reicht von wenigen Kilobyte (TinyOS, QNX) bis hin zu mehreren Megabyte (Windows CE / XP Embedded).
- Die Betriebssysteme sind typischerweise skalierbar. Zur Änderung des Leistungsumfangs von Betriebssystemen muss das System entweder neu kompiliert werden (VxWorks) oder neue Prozesse müssen nachgeladen werden (QNX).
- Die Echtzeitfähigkeit von Standardbetriebssysteme kann durch Erweiterungen erreicht werden (RTLinux/RTAI).
- Die Schedulingverfahren und die IPC-Mechanismen orientieren sich stark an den in POSIX vorgeschlagenen Standards.
- Das Problem der Prioritätsinversion wird zumeist durch Prioritätsvererbung gelöst.

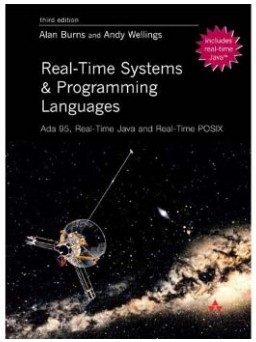
Kapitel 6

Programmiersprachen für Echtzeitsysteme

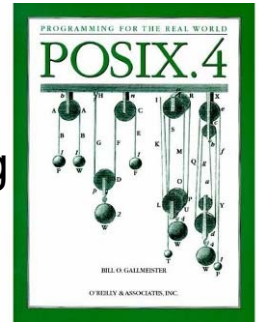
Inhalt

- Motivation
  - Anforderungen von Echtzeitsystemen
  - Geschichte
- PEARL
- Ada
- Real-Time Java
- Zusammenfassung

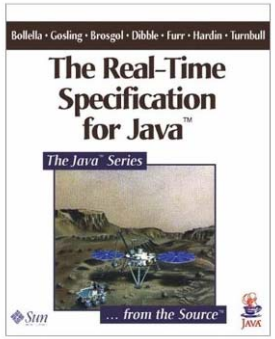
Literatur



A. Burns, A. Wellings: Real-Time Systems & Programming Languages, 2001



B. Gallmeister: POSIX.4 Programming for the Real World, 1995



G. Bollella: The Real-Time Specification for Java, 2000

Paper:

- N. Wirth: Embedded Systems and Real-time Programming, EMSOFT 2001
- Ascher Opler: Requirements for Real-Time Languages, Communications of ACM 1966

Programmiersprachen für Echtzeitsysteme

Anforderungen

Anforderungen

- Die Anforderungen an Programmiersprachen für Echtzeitsysteme fallen in verschiedene Bereiche:
  - Unterstützung bei der Beherrschung komplexer und nebenläufiger Systeme
  - Möglichkeit zur Spezifikation zeitlicher Anforderungen
  - Unterstützung der hardwarenahen Programmierung
  - Erfüllung hoher Sicherheitsanforderungen (Fehlersicherheit)
  - Möglichkeiten zum Umgang mit verteilten Systemen

Beherrschung komplexer nebenläufiger Systeme

- Anforderungen an Programmiersprachen
  - Konstrukte zur Aufteilung der Anwendung in kleinere, weniger komplexe Subsysteme
  - Unterstützung von Nebenläufigkeit (Prozesse, Threads)
  - Daten- und Methodenkapselung in Modulen zur Erleichterung der Wiederverwendbarkeit
  - Eignung für unabhängiges Implementieren, Übersetzen und Testen von Modulen durch verschiedene Personen

Einhalten zeitlicher Anforderungen

- Projektierbares Zeitverhalten
  - Möglichkeit zur Definition von Prioritäten
  - wenig (kein) Overhead durch Laufzeitsystem (z.B. Virtual Machine)
- Bereitstellung umfangreicher Zeitdienste
- Zeitüberwachung aller Wartezustände
- Möglichkeit zur Aktivierung von Prozessen
  - sofort
  - zu bestimmten Zeitpunkten
  - in bestimmten Zeitabständen
  - bei bestimmten Ereignissen

Unterstützung hardwarenaher Programmierung

- Ansprechen von Speicheradressen, z.B. „memory mapped I/O“
- Unterbrechungs- und Ausnahmebehandlung
- Unterstützung vielseitiger Peripherie
- Definition virtueller Geräteklassen mit einheitlichen Schnittstellen
- einheitliches Konzept für Standard- und Prozesse- Ein-/Ausgabe

Erfüllung hoher Sicherheitsanforderungen

- Lesbarkeit, Übersichtlichkeit, Einfachheit durch wenige Konzepte
- Modularisierung und strenge Typüberprüfung als Voraussetzung zur frühen Fehlererkennung durch Übersetzer, Binder und Laufzeitsystem
- Überprüfbare Schnittstellen (-beschreibungen) der Module
- Verifizierbarkeit von Systemen

Sicherheit fängt schon im Kleinen an

- Lexikalische Konventionen können Fehler verhindern.
- Negatives Beispiel: FORTRAN
  - In FORTRAN werden Leerzeichen bei Namen ignoriert.
  - Variablen müssen in FORTRAN nicht explizit definiert werden
- Problem in Mariner 1:
  Aus einer Schleife

```
DO 5 K = 1,3
```

  wird durch versehentliche Verwendung eines Punktes

```
DO5K=1.3
```

  eine Zuweisung an eine nicht deklarierte Variable.
  $\Rightarrow$ Zerstörung der Rakete, Schaden 18,5 Millionen $

Anforderungen durch verteilte Systeme:

- Notwendigkeit vielseitiger Protokolle zur Kommunikation (Feldbus, LAN)
- Unterstützung von Synchronisation auch in verteilten Systemen
- Möglichkeit zur Ausführung von Operationen auf Daten anderer Rechner
- Konfigurationsmöglichkeit zur Zuordnung von Programmen/Modulen zu Rechnern
- Möglichkeit zur automatischen Neukonfigurierung in Fehlersituationen

Programmiersprachen für Echtzeitsysteme

Geschichte

Geschichte: 1960-1970

- 1960-1970
  - Verwendung von Assemblerprogrammen, da der Speicher sehr teuer ist
  - Programme sind optimiert ⇒ jedes Bit wird genutzt
- ab ca. 1966
  - erster Einsatz von höheren Sprachen, z.B.
    - CORAL und RTL/2
    - ALGOL 60
    - FORTRAN IV
  - Prozeduraufrufe für Echtzeitdienste des Betriebssystems
  - Probleme:
    - viel Wissen über Betriebssystem notwendig
    - wenig portabel
    - keine semantische Prüfung der Parameter durch den Übersetzer, da keine speziellen Datentypen für Prozesse, Uhren oder Semaphoren existierten ⇒ schwierige Fehlersuche

Geschichte: 1970-1980

- Existenz erster Echtzeitsprachen (nationale bzw. internationale Normen):
  - PEARL (Deutschland): Process and Experiment Automation Realtime Language
  - HAL/S (USA)
  - PROCOL (Japan)
  - RT-FORTRAN
  - RT-BASIC
- Neue Datentypen (z.B. task, duration, sema, interrupt) mit zugehörigen Operationen sind in die Sprache integriert
- Einführung einheitlicher Anweisungen vor Ein-/Ausgabe und die Beschreibung von Datenwegen

Geschichte: 1970-1980

- Vorteil:
  - Benutzerfreundliche Sprachelemente
  - Prüfung der Semantik der Parameter bei Betriebssystemaufrufen durch Übersetzer möglich
  - Weitgehende Portabilität
- Nachteil: geeignete Betriebssysteme sind nicht vorhanden Möglichkeiten
  1. Entwicklung eines eigenen Betriebssystems $\Rightarrow$ hohe Entwicklungskosten
  2. Anpassung eines vorhandenen Standardbetriebssystems $\Rightarrow$ Gefahr der Existenz überflüssiger Teile im Betriebssystem, eingeschränkte Portabilität

Geschichte ab 1978

- universelle sichere hohe Sprachkonzepte für alle Anwendungsbereiche
  - Standardisierung, inbesondere durch Department of Defense (DOD): Ada
  - Datentypen (z.B. task, duration, interrupt) oder systemabhängige Parameter werden in sprachlich sauberer Weise mittels Module /Packages eingebunden
- Beispiele:
  - Ada83,Ada95
  - CHILL
  - PEARL, PEARL 90, Mehrrechner.PEARL

Geschichte heute:

- Trend hin zu universellen Sprachen (z.B. C,C++ oder Java) mit Bibliotheksprozeduren für Echtzeitdienste angereichert (z.B. POSIX)
- herstellerspezifische Speziallösungen für eingeschränkte Anwendungsbereiche, z.B.
  - Prüfsysteme
  - Standardregelungsaufgaben
  - Förderungstechnik
  - Visualisierung (Leitstand)
  - Telefonanlagen
- Beispiele:
  - SPS-Programmierung (Speicherprogrammierbare Steuerung)
  - ATLAS (Abbreviated Test Language for All Systems) für Prüfsysteme (v.a. Flugzeugelektronik)
  - ESTEREL

Programmiersprachen für Echtzeitsysteme

PEARL

Daten

- Process and Experiment Automation Real-Time Language
- DIN 66253
- Ziele:
  - Portabilität
  - Sicherheit
  - sichere und weitgehend rechnerunabhängige Programmierung
- lauffähig z.B. unter UNIX, OS/2, Linux
- Versionen: BASIC PEARL (1981), Full PEARL (1982), Mehrrechner PEARL (1988), PEARL 90 (1998)
- http://www.irt.uni-hannover.de/pearl/

Eigenschaften

- strenge Typisierung
- modulbasiert
- unterstützt (prioritätenbasiertes) Multitasking
- E/A-Operationen werden von eigentlicher Programmausführung separiert
- Synchronisationsdienste: Semaphore, Bolt-Variablen
- Zugriff auf Unterbrechungen
- erleichterte Zeitverwaltung

Grundstruktur

```
/*Hello World*/
MODULE Hello;
    SYSTEM;
          termout: STDOUT;

    PROBLEM;
          DECLARE x FLOAT;
          T: TASK MAIN;
          x := 3.14;                !PI
          PUT 'Hello' TO termout;
    END;
MODEND;
```

Erläuterung

- **Modularität**: Anwendungen können in einzelne Module aufgeteilt werden (`MODULE`, `MODEND`).
- Aufspaltung in System- und Problemteil:
  - **Systemteil** (`System;`): Definition von virtuellen Geräten für alle physischen Geräte, die das Modul benutzt. Der Systemteil muss auf den entsprechenden Computer angepasst sein
    $\Rightarrow$ Hardwareabhängigkeit
  - **Problemteil** (`PROBLEM;`): eigentlicher, portabler Programmcode
- Sonstige Notationen typisch für prozedurale Sprachen:
  - Kommentare `!`,`/*…*/`
  - Semikolon zur Terminierung von Anweisungen

Datentypen

Schlüsselwort	Bedeutung	Beispiel
FIXED	Ganzzahlige Variable	-2
FLOAT	Gleitkommazahl	0.23E-3
CLOCK	Zeitpunkt	10:44:23.142
DURATION	Zeitdauer	2 HRS 31 MIN 2.346 SEC
CHAR	Folge von Bytes	'Hallo'
BIT	Folge von Bits	'1101'B1

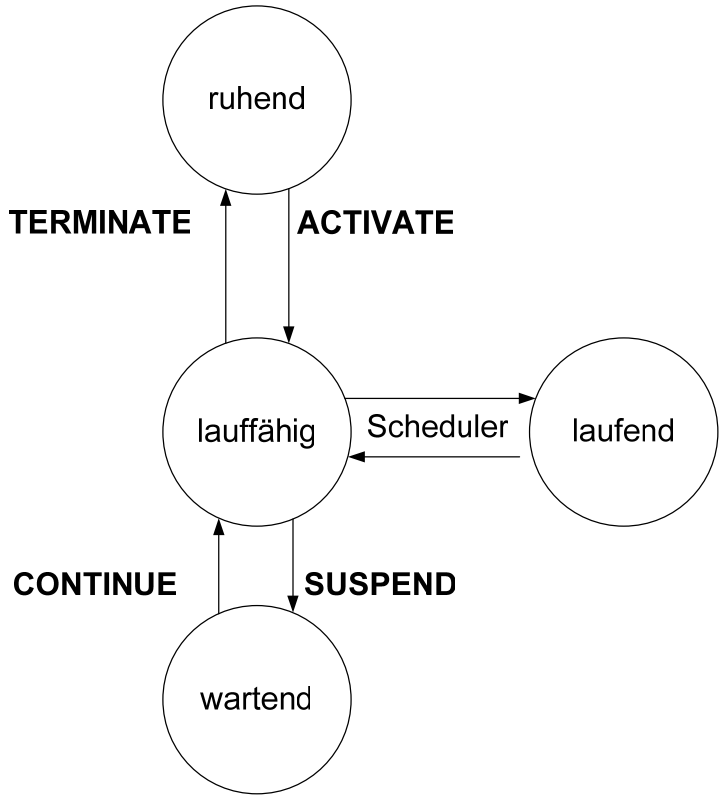
- Variablen werden durch `DECLARE` deklariert und mittels `INIT` initialisiert.
- Durch das Schlüsselwort `INV` werden Konstanten gekennzeichnet.
- Die temporalen Variablen bieten eine Genauigkeit von einer Millisekunde
- Die Genauigkeit der Datentypen kann angegeben werden
- Zeiger auf Datentypen werden unterstützt

Prozessmodell

- Initial sind alle Prozesse bis auf MAIN ruhend
- Zustandswechsel sind unter Angabe einer exakten Zeit möglich:

```
AFTER 5 SEC ACTIVATE Task1;

AT 10:15:0 ALL 2 MIN
     UNTIL 11:45:0
     ACTIVATE Student;
```

- Scheduling präemptives, prioritätenbasiertes Schedulingverfahren mit Zeitscheiben (Round-Robin)
- Zuweisung der Prioritäten durch den Benutzer
- Zeitscheibenlänge abhängig vom Betriebssystem



Prozess-Synchronisation

- Zur Synchronisation bietet PEARL Semaphore und Bolt-Variablen:
  - Semaphore (Datentyp: SEMA):
    - Deklaration wie bei einer Variablen
    - Operationen REQUEST und RELEASE zum Anfordern und Freigeben des Semaphores
    - Mittels der Operation TRY kann versucht werden den Semaphore nicht blockierend zu lesen
    - Es werden keine Möglichkeiten zur Vermeidung von Prioritätsinversion geboten
  - Bolt-Variablen (Datentyp: BOLT):
    - Bolt-Variablen besitzen wie Semaphoren die Zustände belegt und frei und zusätzlich einen 3. Zustand: Belegung nicht möglich
    - RESERVE und FREE funktionieren analog zu Semaphore-Operationen REQUEST bzw. RELEASE
    - exklusive Zugriffe mit RESERVE haben Vorrang von (nicht exklusiven) Zugriffen mit ENTER (Freigabe mit LEAVE)
    - Eine elegante Formulierung des Leser-Schreiber-Problems ist damit möglich

Lösung: Leser-Schreiber-Problem

```
PROBLEM;
     DECLARE content CHAR(255); ! Speicher: 255 Bytes
           DCL key BOLT;      ! Default: frei

    LESER1: TASK;
          DECLARE local1 CHAR(255);
          ENTER key; local1=content; LEAVE key;
    END;

    LESER2: TASK;
          DECLARE local2 CHAR(255);
          ENTER key; local2=content; LEAVE key;
    END;

    SCHREIBER1: TASK;
          DECLARE newcontent1 CHAR(255);
          RESERVE key; content=newcontent1; FREE  key;
    END;

    SCHREIBER2: TASK;
          DECLARE newcontent2 CHAR(255);
          RESERVE key; content=newcontent2; FREE  key;
    END;
```

Unterbrechungen

- Es können nur Prozesse, die durch WHEN eingeplant sind, aktiviert oder fortgesetzt werden.
- Es ist möglich Unterbrechungen durch DISABLE/ENABLE zu sperren bzw. freizugeben.
- Beispiel: Student 2 weckt Student 1 beim Eintreffen der Unterbrechung auf.

```
MODULE Vorlesung:
   System;
        alarm: IR*2;

   PROBLEM;
        SPECIFY alarm INTERRUPT;

   student2: TASK PRIORITY 20
        WHEN alarm ACTIVATE student1;
        DISABLE alarm;
        ...
        ENABLE alarm;
   END;

MODEND;
```

Programmiersprachen für Echtzeitsysteme

Ada

Einleitung

- 1970 von Jean Ichbiah (Firma Honeywell Bull) entworfen
- Durch das Department of Defense gefördert
- Mitglied der Pascal Familie
- Häufige Verwendungen für Systeme mit hohen Anforderungen an die Sicherheit.
- Bis 1997 mussten alle Systeme im Rahmen von DOD-Projekten mit einem Anteil von mehr als 30% neuen Code in ADA implementiert werden.
- Versionen: Ada 83, Ada 95
- Freie Compiler sind verfügbar: z.B. http://www.adahome.com
- http://www.ada-deutschland.de/

Eigenschaften

- Sicherheit durch
  - sehr **strenges Typsystem**
  - zahlreiche Prüfungen zur Laufzeit: z.B. zur Erkennung von Speicherüberläufen, Zugriff auf fremden Speicher, Off-by-One-Fehlern
  - Verhinderung von Fehlern bei nebenläufiger Programmierung (Rendezvous-Konzept, geschützte Typen)
- Unterstützung der modularen Programmierung (insbesondere auch **information hiding**, also Aufteilung separate Schnittstellen und Implementierung)
- Unterstützung der Ausnahmebehandlung
- Eignung zur Implementierung generischer Systeme
- Ab Ada 95:
  - objektorientierte Programmierung
  - dynamische Polymorphie
- Offener Standard: http://www.adaic.org/standards/95lrm/html/RM-TOC.html

Strukturierung

- Die Programme können beliebig in Blöcke/ Unterprogramme/ Pakete/ Tasks aufgeteilt werden.
- Pakete und Tasks müssen, Unterprogramme können in eine Spezifikation (head) und einen Rumpf (body) aufgeteilt werden
  - Kopf: Definition der Schnittstellen und Variablen auf die andere Pakete/Tasks/Unterprogramme zugreifen können
  - Rumpf: private spezifiziert lokale Objekte, deren Realisierung verborgen bleiben
  - Der Anweisungsteil des Pakets wird einmalig beim Abarbeiten der Paketdeklaration ausgeführt
- Benutzung von Paketen:
  - Durch den Befehl `WITH` kann ein Paket benutzt werden.

```
PACKAGE <name> IS
   <sichtbare Vereinbarungen>;
   [PRIVATE <Vereinbarungen>;]
END <name>

PACKAGE BODY <name> IS
   <lokale Vereinbarungen>;
BEGIN
   <Anweisungen>
   [EXCEPTION
   <Ausnahmebehandler>]
END <name>;
```

Generische Einheiten

- Durch das Schlüsselwort `GENERIC` können Unterprogramme/Pakete als Programmschablonen implementiert werden.
- Parameter sind Objekte und Objekttypen
- Freie Parameter werden bei der Übersetzung durch aktuelle Parameter ersetzt (entspricht templates in C++)

Spezifikation:

```
GENERIC
   TYPE sometype IS PRIVATE;
PACKAGE queue_handling IS
   TYPE queue (maxlength: NATURAL)
   IS PRIVATE;
PROCEDURE enqueue (q: IN OUT queue;
   elem: IN sometype);
   ...
PROCEDURE dequeue ...
PRIVATE
   SUBTYPE index IS CARDINAL RANGE
        0..1000;
   …
```

Benutzung:

```
DECLARE
   PACKAGE int_queue
   IS NEW queue_handling (INTEGER);
   …
```

Prozesse

- Prozesse (Datentyp TASK) werden wie Variablen behandelt:
  - Verwendung als Komponenten von Feldern oder Records möglich.
  - Verwendung als Parameter erlaubt.
- Der Spezifikationsteil darf ausschließlich die Deklaration von Eingängen (Schlüsselwort ENTRY) enthalten.
  - Ein Eingang ist ein Bestandteil eines Tasks, der von außen aufgerufen werden kann.
  - Es ist zu jedem Zeitpunkt immer nur höchstens ein Eingang aktiviert. In der Zwischenzeit eintreffende Aufrufe werden in einer Warteschlange eingereiht.

```
TASK [TYPE] name IS
   ENTRY ename (<Parameter>);
   ENTRY ...
END name;


TASK BODY name IS
   <deklarationen>
BEGIN
   …
   ACCEPT ename(<Parameter>) DO
        …
   END ename;
   …
   EXCEPTION [<exception handler>]]
END name;
```

Lebenszyklus eines Prozesses

- Start:
  - Prozesse werden automatisch beim Abarbeiten der Deklaration aktiv, aber erst am Ende des Deklarationsteils gestartet.
  - Durch die Blockstruktur können Prozessaufrufe geschachtelt auftreten.
- Beendigung:
  - Es gibt nur die Operation `ABORT` zum Datentyp TASK (gewaltsames Beenden)
  - Prozesse terminieren automatisch beim Erreichen des Blockendes, falls sie nicht auf das Ende von untergeordneten Prozesse warten müssen.
  - Der umfassende Prozess wird durch implizite Synchronisation des Betriebssystems erst beendet, wenn alle in ihm deklarierten und damit alle gestarteten Prozesse beendet sind.
  - Ein Block wird erst verlassen, wenn alle in ihm vereinbarten Prozesse beendet sind.

Partitionen

- Seit Ada 95 werden auch Partitionen unterstützt.
- Eigenschaft einer Partition:
  - Partitionen haben einen eigenen Adressraum
  - Partitionen können Prozesse enthalten
  - Die Programme können durch Partitionen auf verschiedenen Rechnern ausgeführt werden
  - Aktive Partitionen enthalten Prozesse und main()
  - Passive Partitionen enthalten nur Daten und/oder Unterprogramme
  - Eine Partition wird erst beendet, wenn all ihre Prozesse beendet sind
  - Partitionen werden von außen oder durch einen sogenannten Environment-Task angestoßen, bei deren Abarbeitung, die in ihr enthaltene Main-Prozedur aufgerufen wird
  - Zur Kommunikation zwischen Partitionen können RPC oder gemeinsame Daten einer dritten Partition benutzt werden.

Prozess-Synchronisation: Rendezvous-Konzept

- Ada bietet mit Rendezvous ein Konzept zur synchronen Kommunikation:
  - Definition eines Eingangs (`ENTRY`) in einem Prozess
  - `ACCEPT`-Anweisung zu den Eingängen in den Prozessen
  - Der Aufruf des Eingangs eines anderen Prozesses erfolgt wie ein Prozeduraufruf mit Parametern.
  - Die Ausführung erfolgt erst, wenn beide Prozesse bereit sind: der externe Prozess den Aufruf durchführt und der eigentliche Prozess die `ACCEPT`-Anweisung erreicht.
  - Sowohl der aufrufende als auch der aufgerufene Prozess warten, bis die Anweisungen im `ACCEPT`–Block durchgeführt sind.
  - Alternatives Warten durch `SELECT` mit Guards (`WHEN`).
  - Eine zeitliche Begrenzung der Wartezeit (watchdog) ist möglich.
- Eine ausführliche Beschreibung ist unter http://www.ada-deutschland.de/AdaTourCD2004/ada_dokumentation/parallelprozesse/10_6_rendezvous.html zu finden.

Beispiel: Realisierung eines gemeinsamen Speichers (Leser-Schreiber-Problem mit Schreiberpriorität)

- Grundgerüst des Codes:
  - Deklaration eines generischen Datentyps `item`
  - Das Paket `sharedmemory` biete nach außen die beiden Funktionen `read` und `write` an.
  - Intern wird der Speicher in der Variablen `item` gesichert.
  - Zusätzlich besitzt das Paket einen Prozess `control`, der den Zugriff auf die Variable `value` überwacht.

```
GENERIC
        TYPE item IS PRIVATE

PACKAGE sharedmemory IS
        PROCEDURE read(x: OUT item)
        PROCEDURE write(x: IN item)
END;

PACKAGE BODY sharedmemory IS
        value: item;

  TASK control IS
        ... (siehe folgende Folien)
  END control;
  PROCEDURE read(x:OUT item) IS
  BEGIN
        ... (siehe folgende Folien)
  PROCEDURE write(x:IN item) IS
  BEGIN
        ... (siehe folgende Folien)
END shared memory;
```

Beispiel: Fortsetzung

- Schnittstelle des Prozesses `control`: der Prozess bietet insgesamt drei Funktionen als Rendezvous an: `start`, `write`, `stop`
- Die Prozedur `read` benutzt die Schnittstelle `start` zum Signalisierung des Lesebeginns und `stop` zur Signalisierung der Beendigung.
- Die Prozedur `write` benutzt die Schnittstellenfunktion `write`.
- Unterschied zwischen `read` und `write`: mehrere Leser dürfen gleichzeitig auf die Daten zugreifen, aber nur ein Schreiber.

```
TASK control IS
        ENTRY start;
        ENTRY stop;
        ENTRY write(x:in item);
END control;

PROCEDURE read(x:OUT item) IS
BEGIN
        control.start;
        x:=value;
        control.stop;
END read;

PROCEDURE write(x:IN item) IS
BEGIN
        control.write(x);
END write;
```

Beispiel (Fortsetzung): Code des Prozesses `control`

- Die Anzahl der aktuellen Leser wird in der Variable `readers` gespeichert
- Bevor ein Prozess lesend auf den Speicher zugreifen darf, muß er erstmalig beschrieben werden
- Im Anschluß führt der Prozess eine Endlosschleife mit folgenden Möglichkeiten aus:
  1. Falls kein Schreiber auf den Schreibzugriff wartet (`WHEN write'count=0`), so wird ein Schreibwunsch akzeptiert und die Anzahl der Leser erhöht, sonst wird der Wunsch bis zur Ausführung des Schreibwunsches verzögert (**Schreiberpriorität**).
  2. Beendet ein Leser den Zugriff, so wird die Anzahl erniedrigt.
  3. Falls kein Leser mehr aktiv ist (`WHEN readers=0`), werden Schreibwünsche akzeptiert, ansonsten wird dieser verzögert.
- Entscheidend: Die Auswahl zwischen den Rendezvous-Alternativen erfolgt nicht deterministisch (durch Würfeln)

```
TASK BODY control IS
    readers: integer :=0;
BEGIN
    ACCEPT write(x:IN item) DO
        value:=x;
    END;
    LOOP
        SELECT
            WHEN write'count=0 =>
            ACCEPT start;
            readers:=readers+1;
        OR
            ACCEPT stop;
            reader:=readers-1;
        OR
            WHEN readers=0 =>
            ACCEPT write(x:IN item) DO
            value:=x;
        OR
            DELAY 3600.0;
            exit;
        END SELECT;
    END LOOP;
END control;
```

Wechselseitiger Ausschluss

- In Ada95 bietet zum wechselseitigen Ausschluss geschützte Typen (`PROTECTED TYPE`):
  - Die Objekte können Typen und Daten sowie die benötigten Operationen (Funktionen, Prozeduren, Eingänge) enthalten.
  - Das Laufzeitsystem sichert, dass Prozeduren in einem `PROTECTED TYPE` exklusiv ausgeführt werden.
  - Auf lesende Funktionen (`FUNCTION`) in einem `PROTECTED TYPE` können mehrere Prozesse gleichzeitig zugreifen.
  - Prioritätsvererbung wird bei geschützten Typen unterstützt.
  - Beim Auftreten von Ausnahmen wird der Block verlassen und die Belegung automatisch aufgehoben

$\Rightarrow$ Vorgehen ähnelt Monitoren

Beispiel: Realisierung eines Semaphors

```
PROTECTED TYPE sema (init:
        INTEGER := 1) IS
   ENTRY P;
   PROCEDURE V;
   PRIVATE
     count: INTEGER := init;
END sema;


PROTECTED BODY sema IS
   ENTRY P WHEN count > 0 IS
   BEGIN
       count := count - 1;
   END P;

PROCEDURE V IS
   BEGIN
       count := count + 1;
   END V;
END sema;
```

Benutzung:

```
s : sema;
...
s.P;
... -- Exklusive Anweisungen
s.V;
```

Ausnahmen

- Ausnahmen können in Anweisungen, bei Deklarationen und im Rendezvous auftreten
- Der Benutzer kann Ausnahmen selbst definieren: `exc1: EXCEPTION`
- Ausnahmen können durch `RAISE` ausgelöst werden, die Behandlung erfolgt typischerweise am Ende des Rahmens.
- Beim Auftreten einer Ausnahme wird der Rahmen verlassen und die entsprechende Behandlung gestartet.
- Ist keine Behandlung angegeben, so wird die Ausnahme an den umgebenden Rahmen weitergeleitet (exception propagation), bis eine Behandlung oder ein Programmabbruch erfolgt

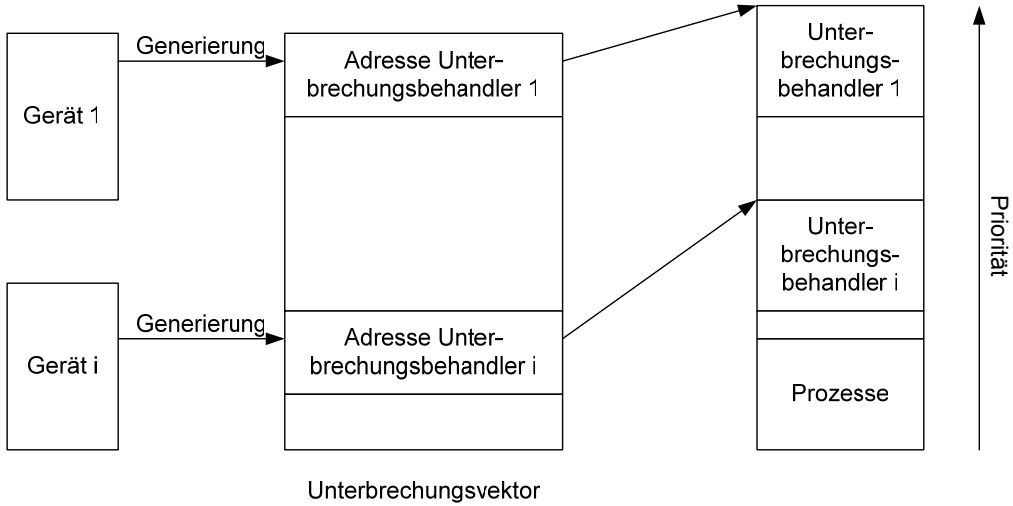
- Syntax der Behandlung

```
EXCEPTION
     WHEN exceptionname =>
          <Anweisungsfolge>;
     ...
     WHEN OTHERS =>
          <Anweisungsfolge>;
```

- Mit `OTHERS` können beliebige Ausnahmen behandelt werden
- Es gibt viele vordefinierte Ausnahmen:
  - `CONSTRAINT_ERROR`
  - `NUMERIC_ERROR`
  - `PROGRAMM_ERROR`
  - `STORAGE_ERROR`
  - `TASKING_ERROR`

Unterbrechungen

- Zur Behandlung von Unterbrechungen können PROTECTED PROCEDURES verwendet werden.
- Diese Prozeduren werden mit hoher Priorität (abhängig vom Betriebssystem, höher als Prozesspriorität) exklusiv ausgeführt.
- Die Zuordnung der Prozeduren zu den Unterbrechungen erfolgt statisch oder dynamisch
- Die möglichen Unterbrechungen sind im implementierungsabhängigen Paket `Ada.Interrupt.Names` beschrieben.



Unterbrechungen: statisch vs. dynamisch

```
PROTECTED PROCEDURE alarm IS
    PROCEDURE response;
        -- parameterlos
    PRAGMA ATTACH_HANDLER
        (response, Alarm_ID);
END alarm;
PROTECTED PROCEDURE BODY alarm
    IS
    PROCEDURE response IS
    ...
    END response;
END alarm;
```

```
PROTECTED PROCEDURE alarm IS
    PROCEDURE response;
        --parameterlos
    PRAGMA INTERRUPT_HANDLER
        (response);
END alarm;
PROTECTED PROCEDURE BODY alarm
    IS
    -- wie oben
    -- spaeterer Prozeduraufruf:
    ATTACH_HANDLER(alarm.response,
    Alarm_ID);
    …
END alarm;
```

Programmiersprachen für Echtzeitsysteme

Real-Time Java

Motivation

- Java ist eine sehr weit verbreitete Programmiersprache
- Vorteile:
  - Portabler Code durch virtuelle Maschine
  - Objektorientierte Paradigma
  - Strengere Typisierung
  - Einfacher Umgang mit Speicher (keine Zeiger, Garbage Collection)
- Nachteil: nicht echtzeitfähig (siehe nächste Folien)

$\Rightarrow$ RTSJ (Real-Time Specification for Java) **erweitert** Java:

  - Erweiterung der Spezifikation der Sprache Java
  - Erweiterung der Java Virtual Machine Spezifikation
  - Entwicklung einer Echtzeit-API

Design-Prinzipien RTSJ

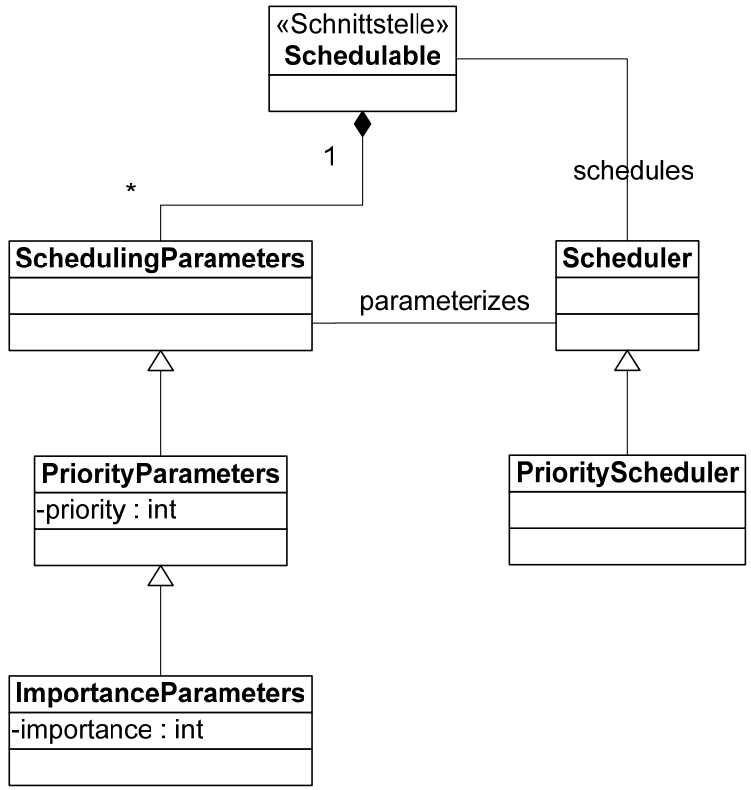
1. Keine Einschränkungen auf bestimmte Java-Umgebung (z.B. eine bestimmte Version von JDK)
2. Rückwärtskompatibilität
3. „Write once carefully, run anywhere conditionally“
4. Unterstützung aktueller Entwicklungsprozesse für Echtzeitsysteme
5. (Zeitlich) vorhersagbare Ausführung
6. Keine syntaktischen Erweiterungen
7. Kein Verbot von Implementierungsabweichungen (allerdings sollen diese sorgfältig dokumentiert werden).

Scheduling in Java

- Java Spec: „…threads with higher priority are generally executed in preference to threads with lower priority…"
- Scheduler:
  - Algorithmus nicht festgelegt
  - Keine vorgeschriebene Anzahl von Prioritäten
  - Verwendung von Round-Robin oder FIFO bei Prozessen gleicher Priorität nicht spezifiziert

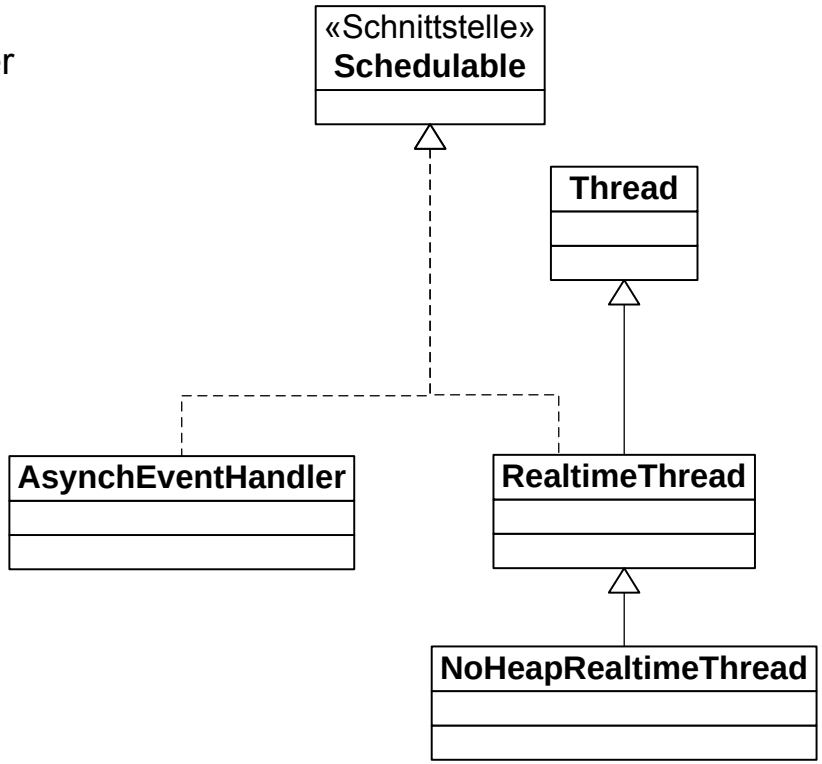
Scheduling in RTJS

- Einführung des PrioritySchedulers:
  - feste Prioritäten
  - präemptives Scheduling
  - mindestens 28 Prioritätsebenen für Echtzeitprozesse
- Unterschiedliche Parameter:
  - Schedulingparameter (traditionelle Priorität, importance-Feld für Überlastsituationen)
  - Freigabeparameter (Parameter für periodische, aperiodische, sporadische Prozesse)
  - Speicherparameter: definiert notwendigen Speicherplatz
  - Prozessgruppenparameter: zur Verwaltung einer Menge von aperiodischen oder sporadischen Prozesse als Meta-periodischen Prozess



Threads in RTJS

- `RealtimeThread`:
  - Kontrolliert durch den Scheduler.
  - Kann neben dem Heap auch eigenen Speicher benutzen.
  - Zugriff auf physikalischen Speicher möglich.
- `NoHeapRealtimeThread`:
  - Zugriff auf Objekte im Heap verboten.
  - Manipulation von Referenzen zu Objekten im Heap verboten.
  - Muss mit einem geschützten Speicherbereich erzeugt werden.
  - Kann den GarbageCollector unverzüglich unterbrechen.
- `AsynchEventHandler`:
  - realisiert Unterbrechungsbehandlungen



Speichermanagement in Java

- Der Garbage Collector ist einer der Hauptgründe, die gegen die Verwendung von Java in Echtzeitsystemen sprechen:
  - In regelmäßigen Abständen wird der Garbage Collector als Prozess im Hintergrund ausgeführt.
  - Der GC ermittelt diejenigen Objekte, auf die nicht mehr verwiesen wird. Diese Objekte werden markiert und in einem zweiten Durchgang entfernt.
  - Problem: Garbage Collector benötigt langwierige Ausführungszeiten und kann nicht unterbrochen werden.
- Ansatz in RTSJ: Veränderung des Begriffs der Lebenszeit
  - manuelle Steuerung: Kontrolle der Lebenszeit via Programmlogik.
  - automatische Steuerung: wie bisher über Sichtbarkeit der Objekte.

Speichermanagement in RTSJ (1)

- RTSJ unterscheidet zwischen vier Speicherarten:
  - Heap memory (Standardspeicher von Java):
    - Verwaltung erfolgt durch den Garbage Collector
  - Immortal memory
    - Wird durch alle RealtimeThreads gemeinsam benutzt.
    - Pro Instanz der Virtual Machine existiert genau ein solcher Bereich.
    - Der GarbageCollector hat auf den Bereich keinen Zugriff $\Rightarrow$ alloziierte Objekte bleiben bis zum Ende der Ausführung der Virtual Machine im Speicher
    - Es existiert kein Mechanismus zur Freigabe von Objekten.

Speichermanagement in RTSJ (2)

- Fortsetzung:
  - Scoped memory (Speicher mit eingeschränktem Lebensraum)
    - Der Benutzer kann scoped memory Speicher manuell anlegen.
    - Dabei wird zwischen zwei Arten von Speicher unterschieden:
      - LTMemory: Objektallokationen sind in linearer Zeit durchführbar
      - VTMemory: es werden keine Zeitgarantien gegeben
    - Mit dem Schlüsselwort `enter` kann die Lebensdauer (Klammern begrenzen Lebensraum) definiert werden:
      ```
      myScopedMemArea.enter(){ … }
      ```
    - Alle mit `new` in dem Bereich erzeugten Objekte werden im `ScopedMemory` Bereich alloziiert.
  - RawMemoryAccess: Zusätzlich erlaubt RTSJ im Gegensatz zu Java auch Zugriff auf physikalischen Speicher durch Einführung der zusätzlichen Speicherart `RawMemoryAccess`

Weitere Ergänzungen

- Synchronisation:
  - Bei Monitoren muss jede RTSJ-Implementierung den Priority Inheritance Algorithmus implementieren.
  - Die Implementierung von Priority-Ceiling ist optional.
  - Zwischen Real-Time Threads und Standard Threads können Wait Free Queues (nicht blockierende Nachrichtenwarteschlangen) verwendet werden.
- Zeit:
  - Einführung der Klasse `Time` (mit den Unterklassen `AbsoluteTime`, `RelativeTime`)
  - Einführung der abstrakten Klasse `Clock`
  - Jede RTSJ-Implementierung muss die Klasse `RealtimeClock` enthalten.
  - Einführung der Klasse `Timer` (Unterklassen `OneShotTimer`, `PeriodicTimer`)

Programmiersprachen für Echtzeitsysteme

Zusammenfassung

Zusammenfassung

- Kriterien bei der Auswahl der Sprache sind:
  - Sicherheit
  - Komfort bei der Entwicklung (v.a. Existenz geeigneter Entwicklungswerkzeuge)
  - projektierbares Zeitverhalten
  - Möglichkeit zur hardwarenahen, nebenläufigen Programmierung
  - Portabilität
- Zur Sicherheit tragen eine strenge Typisierung und Prüfungen zur Laufzeit bei.
- Zur Erhöhung der Portabilität werden hardwareabhängige und –unabhängige Codeteile häufig getrennt.
- Virtuelle Laufzeitumgebungen (wie z.B. Virtual Machine in RTSJ) eignen sich nur bedingt zur Verwendung in Echtzeitsystemen

Kapitel 8 (vorgezogen)

Echtzeitfähige Kommunikation

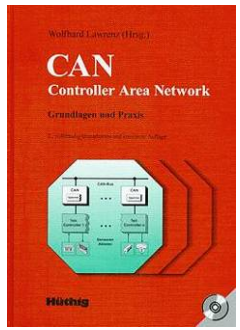
Inhalt

- Grundlagen
- Medienzugriffsverfahren und Vertreter
  - CSMA-CD: Ethernet
  - CSMA-CA: CAN-Bus
  - Tokenbasierte Protokolle: Token Ring, FDDI
  - Zeitgesteuerte Protokolle: TTP
  - Real-Time Ethernet

Literatur

Andrew S. Tanenbaum, Computernetzwerke, 2005

Wolfhard Lawrenz: CAN Controller Area Network. Grundlagen und Praxis, 2000



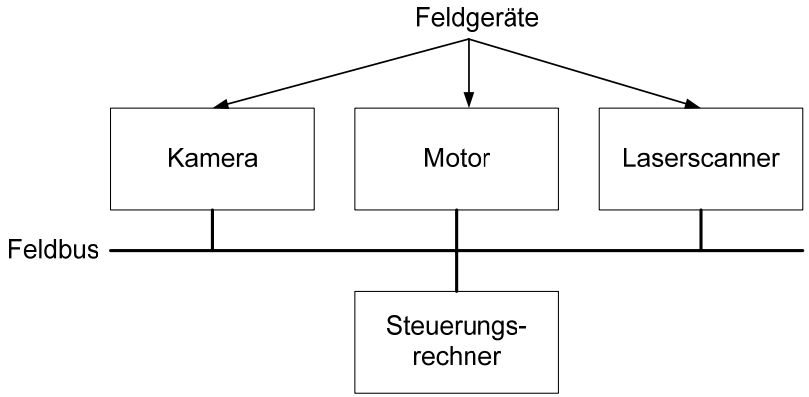
- Spezifikationen:
  - TTTech Computertechnik AG, Time Triggered Protocol TTP/C High-Level Specification Document, 2003 (http://www.vmars.tuwien.ac.at/projects/ttp/)
  - http://www.can-cia.org/
  - http://standards.ieee.org/getieee802/portfolio.html

Anforderungen

- Echtzeitsysteme unterscheiden sich in ihren Anforderungen an die Kommunikation von Standardsystemen.
- Anforderungen speziell von Echtzeitsystemen:
  - vorhersagbare maximale Übertragungszeiten
  - kleiner Nachrichtenjitter
  - garantierte Bandbreiten
  - effiziente Protokolle: kurze Latenzzeiten
  - teilweise Fehlertoleranz
- Kriterien bei der Auswahl:
  - maximale Übertragungsrate
  - maximale Netzwerkgröße (Knotenanzahl, Länge)
  - Materialeigenschaften (z.B. für Installation)
  - Störungsempfindlichkeit (auch unter extremen Bedingungen)
  - Kosten, Marktproduktpalette

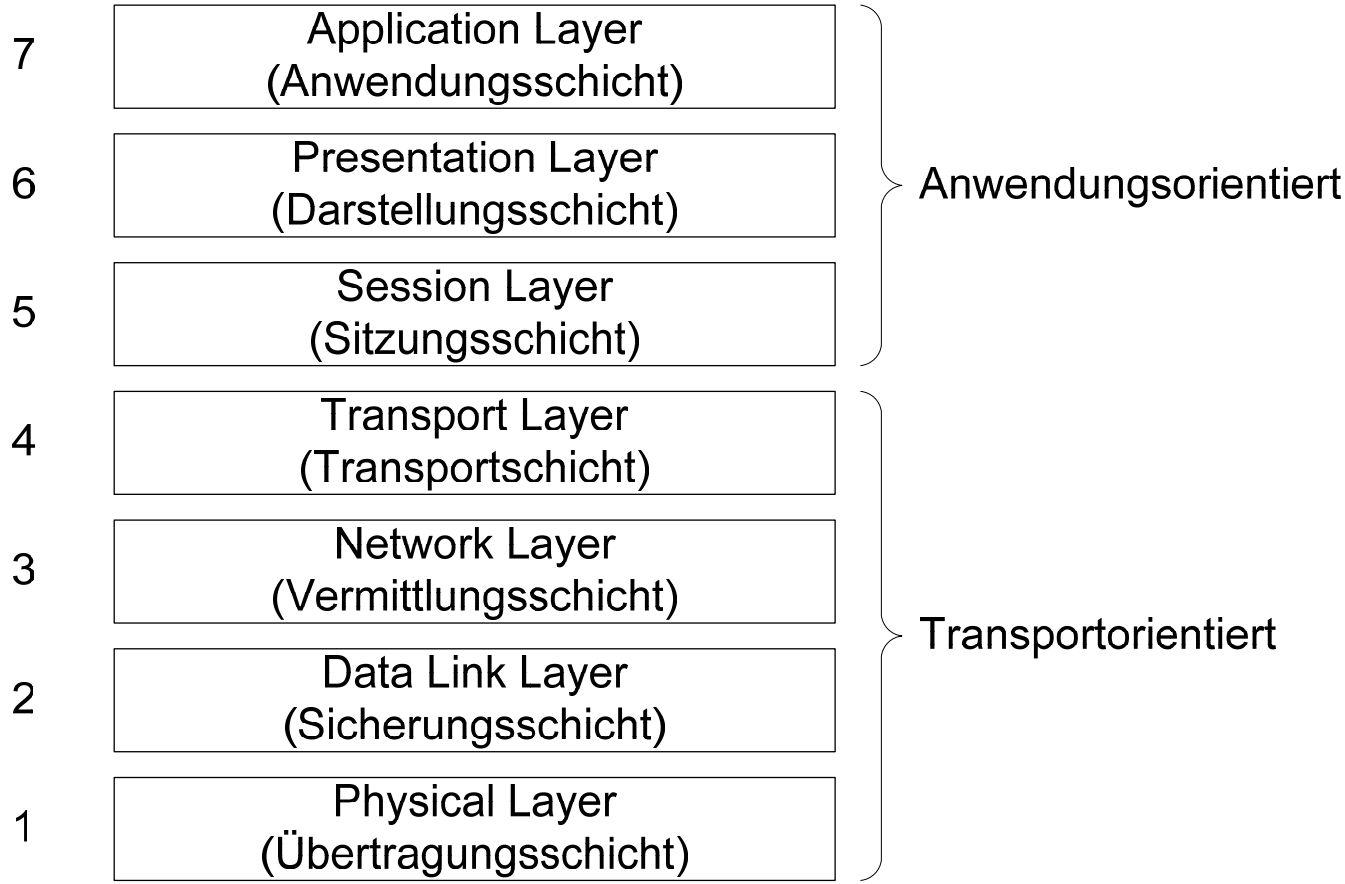
Definition Feldbus

- Die Kommunikation in Echtzeitsystemen erfolgt häufig über **Feldbusse**:



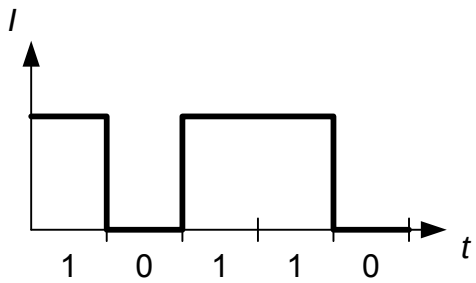
- Feldgeräte sind dabei Sensoren/Aktoren, sowie Geräte zur Vorverarbeitung der Daten.
- Der Feldbus verbindet die Feldgeräte mit dem Steuerungsgerät.
- Beobachtung: echtzeitkritische Nachrichten sind in der Regel kürzer als unkritische Nachrichten.
- Es existiert eine Vielzahl von Feldbus-Entwicklungen: MAP (USA - General Motors), FIP (Frankreich), PROFIBUS (Deutschland), CAN (Deutschland – Bosch), …

Schichtenmodell: ISO/OSI-Modell

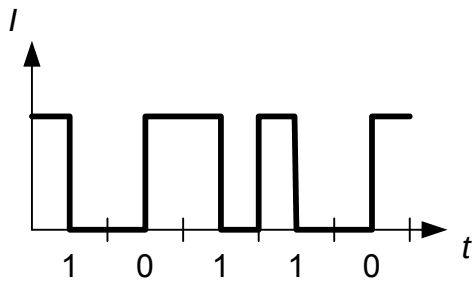


Beschreibung der einzelnen Schichten: Übertragungsschicht

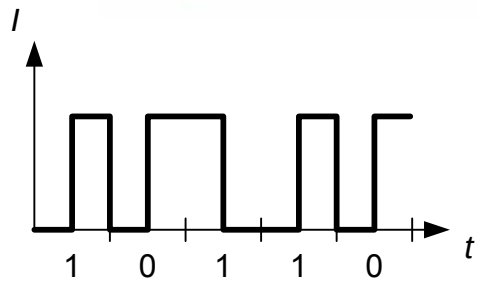
- Aufgaben:
  - Bitübertragung auf physikalischen Medium
  - Festlegung der Medien
    - elektrische, optische Signale, Funk
    - Normung von Steckern
  - Festlegung der Übertragungsverfahren/Codierung
    - Interpretation der Pegel
    - Festlegung der Datenrate



Non-return-to-zero Code



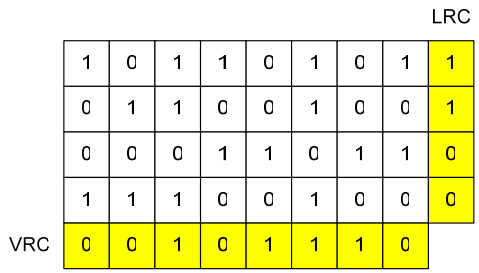
Manchester-Code



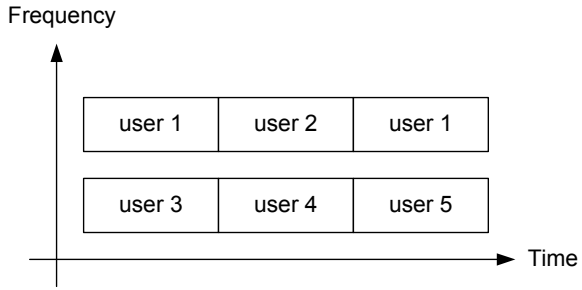
Differentieller Manchester-Code

Beschreibung der einzelnen Schichten: Sicherungsschicht

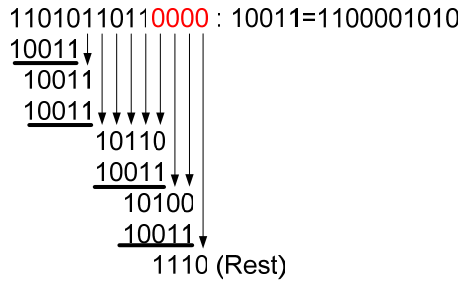
- Aufgaben:
  - Fehlererkennung
    - Prüfsummen
    - Paritätsbits
  - Aufteilung der Nachricht in Datenpakete
  - Regelung des Medienzugriffs
  - Flusskontrolle



Paritätsbits



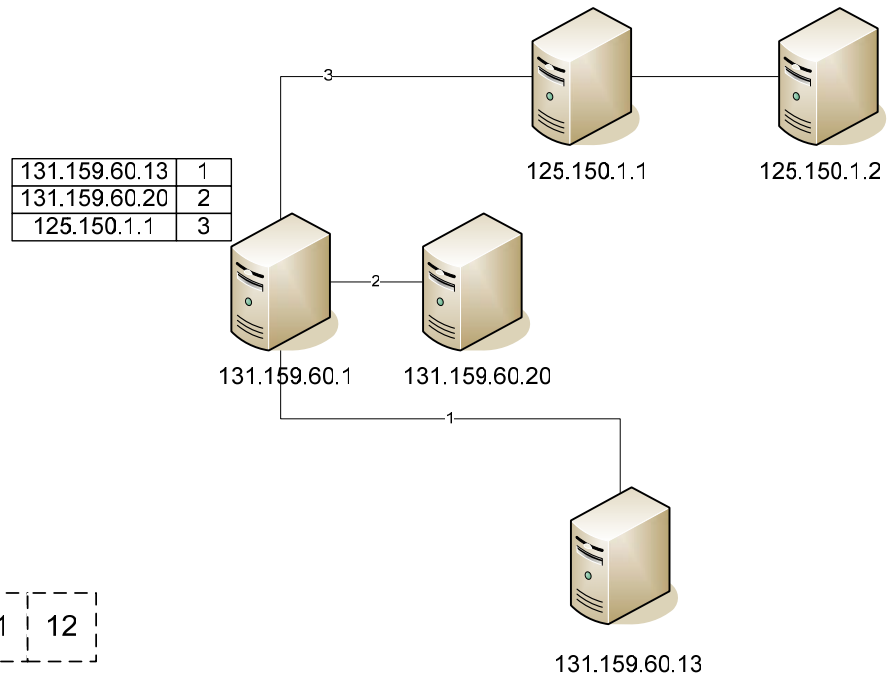
TDMA+FDMA

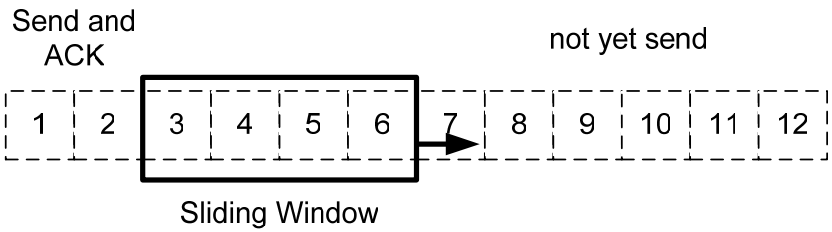


CRC-Verfahren

Beschreibung der einzelnen Schichten: Vermittlungsschicht

- Aufgaben:
  - Aufbau von Verbindungen
  - Weiterleitung von Datenpaketen
    - Routingtabellen
    - Flusskontrolle
    - Netzwerkadressen





Sliding Window Protokoll

Weitere Schichten

- Transportschicht:
  - Transport zwischen Sender und Empfänger (End-zu-End-Kontrolle)
  - Segmentierung von Datenpaketen
  - Staukontrolle (congestion control)
- Sitzungsschicht:
  - Auf- und Abbau von Verbindungen auf Anwendungsebene
  - Einrichten von Check points zum Schutz gegen Verbindungsverlust
  - Dienste zur Organisation und Synchronisation des Datenaustauschs
  - Spezifikation von Mechanismen zum Erreichen von Sicherheit (z.B. Passwörter)
- Darstellungsschicht:
  - Konvertierung der systemabhängigen Daten in unabhängige Form
  - Datenkompression
  - Verschlüsselung
- Anwendungsschicht:
  - Bereitstellung anwendungsspezifischer Übertragungs- und Kommunikationsdienste
  - Beispiele:
    - Datenübertragung
    - E-Mail
    - Virtual Terminal
    - Remote Login
    - Video-On-Demand
    - Voice-over-IP

Schichten in Echtzeitsystemen

- Die Nachrichtenübertragungszeit setzt sich aus folgenden Komponenten zusammen:
  - Umsetzung der Protokolle der einzelnen Schichten durch den Sender
  - Wartezeit auf Medienzugang
  - Übertragungszeit auf Medium
  - Entpacken der Nachricht in den einzelnen Schichten durch den Empfänger

⇒ Jede zu durchlaufende Schicht verlängert die Übertragungszeit und vergrößert die zu sendenden Daten.

⇒ in Echtzeitsystemen wir die Anzahl der Schichten zumeist reduziert auf:

  - Anwendungsschicht
  - Sicherungsschicht
  - Physikalische Schicht

Echtzeitfähige Kommunikation

Medienzugriffsverfahren

Problemstellung

- Zugriffsverfahren regeln die Vergabe des Kommunikationsmediums an die einzelnen Einheiten.
- Das Kommunikationsmedium kann in den meisten Fällen nur exklusiv genutzt werden, Kollisionen müssen zumindest erkannt werden um Verfälschungen zu verhindern.
- Zugriffsverfahren können dabei in unterschiedliche Klassen aufgeteilt werden:
  - Erkennen von Kollisionen, Beispiel: CSMA/CD
  - Vermeiden von Kollisionen, Beispiel: CSMA/CA
  - Ausschluss von Kollisionen, Beispiel: token-basiert, TDMA

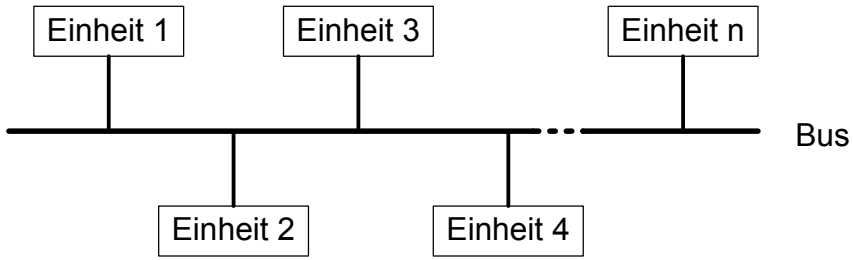
Echtzeitfähige Kommunikation

Carrier Sense Multiple Access/Collision Detection (CSMA/CD)

Vertreter: Ethernet (nicht echtzeitfähig!)

CSMA/CD

- CSMA/CD: Carrier Sense Multiple Access - Collision Detection
  - alle am Bus angeschlossenen Einheiten können die aktuell versendeten Daten lesen (**Carrier Sense**).
  - mehrere Einheiten dürfen Daten auf den Bus schreiben (**Multiple Access**).



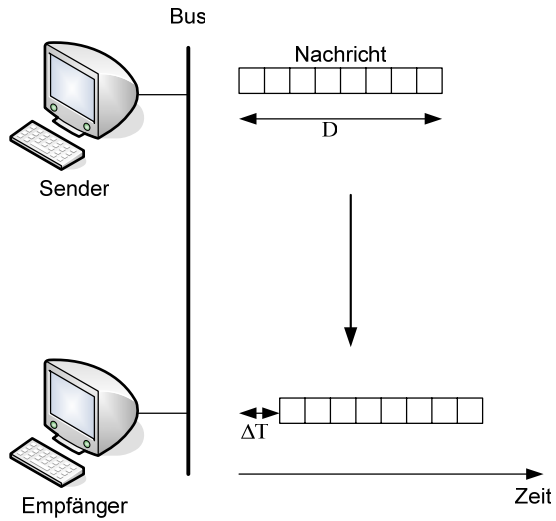
  - Während der Übertragung überprüft der sendende Knoten gleichzeitig das Resultat auf dem Bus, ergibt sich eine Abweichung, so wird eine Kollision angenommen (**Collision Detection**)

CSMA/CD: Ablauf

- Beschrieben wird im Folgenden das 1-persistente CSMA/CD- Verfahren (Spezifikation in der Norm IEEE 802.3)
- Ablauf zum Senden eines Paketes:
  1. Test, ob Leitung frei ist (**carrier sense**)
  2. Falls Leitung für die Zeitdauer eines IFS (**inter frame spacing**) frei ist, wird die Übertragung gestartet, ansonsten Fortfahren mit Schritt 5.
  3. Übertragung der Daten inklusive Überwachung der Leitung. Im Fall einer Kollision: senden eines **JAM**-Signals, fortfahren mit Schritt 5.
  4. Übertragung erfolgreich beendet: Benachrichtige höhere Schicht, Beendigung
  5. Warten bis Leitung frei ist
  6. Sobald Leitung frei: weitere zufälliges Warten (z.B. **Backoff-Verfahren**) und Neustarten mit Schritt 1, falls maximale Sendeversuchsanzahl noch nicht erreicht.
  7. Maximale Anzahl an Sendeversuchen erreicht: Fehlermeldung an höhere Schicht.

Kollisionen

- Um Kollisionen rechtzeitig zu erkennen muss die Signallaufzeit ΔT deutlich kleiner als die Nachrichtenübertragungsdauer D sein.
- Das Störsignal (JAM) wird geschickt um alle anderen Nachrichten auf die Kollision aufmerksam zu machen $\Rightarrow$ Verkürzung der Zeit zur Kollisionserkennung
- Würden die Rechner nach einer Kollision nicht eine zufällige Zeit warten, käme es sofort zu einer erneuten Kollision.
- Lösung im Ethernet: Die Sender wählen eine zufällige Zahl d aus dem Interval $[0...2^i]$, mit i = Anzahl der bisherigen Kollisionen (Backoff-Verfahren).
  $\Rightarrow$ Mit ansteigendem i wird eine Kollision immer unwahrscheinlicher.
  $\Rightarrow$ Bei i = 16 wird die Übertragung abgebrochen und ein Systemfehler vermutet.



TCP vs. UDP

- TCP (Transmission Control Protocol) ist ein zuverlässiges, verbindungsorientiertes Transportprotokoll:
  - Vor der Übertragung der Daten wird zunächst eine Verbindung zwischen Sender und Empfänger aufgebaut (Handshake).
  - Datenverluste werden erkannt und automatisch behoben durch Neuversenden des entsprechenden Datenpakets.

  ⇒ Aufgrund von unvorhersehbaren Verzögerungen (Backoff-Verfahren) und hohem Overhead ist TCP nicht für den Einsatz in Echtzeitsystemen geeignet.
  - Weiteres Problem: Slow Start der Congestion Control Strategie von TCP/IP ⇒ zu Beginn der Übertragung wird nicht die volle Bandbreite ausgenutzt
- UDP (User Datagram Protocol) ist ein minimales, verbindungsloses Netzprotokoll:
  - Verwendung vor allem bei Anwendungen mit kleinen Datenpaketen (Overhead zum Verbindungsaufbau entfällt)
  - UDP ist nicht-zuverlässig: Pakete können verloren gehen und in unterschiedlicher Reihenfolge beim Empfänger ankommen.

  ⇒ Einsatz in weichen Echtzeitsystemen, in denen der Verlust einzelner Nachrichten toleriert werden kann (z.B. Multimedia-Protokollen wie z.B. VoIP, VoD) möglich.

RTP, RTSP: Motivation

- Problem von UDP/IP in Multimediasystemen:
  - keine Möglichkeit zur Synchronisation
  - verschiedene Multimediaströme können kollidieren (z.B. in VoD)
  - Qualitätskontrolle ist wünschenswert

  $\Rightarrow$ in Multimediasystemen werden zusätzliche Protokolle (RTP, RTCP) verwendet.
- Multimediaverbindung mit RTP/RTCP
  - Zur Übertragung der **Steuerungsnachrichten** (in der Regel nicht zeitkritisch) werden zuverlässige Protokolle eingesetzt (z.B. TCP/IP)
  - Zur **Datenübertragung** wird ein **RTP** (**Real-Time Transport Protocol**)-Kanal eingesetzt.
  - Jeder RTP-Kanal wird mit einem **RTCP** (**Real-Time Control Protocol**)-Kanal zur Überwachung der Qualität verknüpft.
  - RTP/RTCP setzen in der Regel auf UDP/IP auf und sind End-zu-End-Protokolle

RTP, RTCP

- RTP:
  - Multicasting
  - Bestimmung des Datenformats (PT)
  - Zeitgebend durch Zeitstempel, die Berechnung des Jitters wird dadurch möglich
  - Möglichkeit zur Ordnung der Pakete und zum Erkennen von verlorenen Paketen durch Sequenznummer

Byte 0								Byte 1								Byte 2								Byte 3							
Bit 0	1	2	3	4	5	6	7	Bit 0	1	2	3	4	5	6	7	Bit 0	1	2	3	4	5	6	7	Bit 0	1	2	3	4	5	6	7
V=2		P	X	CC				M	PT							sequence number															
timestamp (in sample rate units)																															
sychronization source (SSRC) identifier																															
contributing source (CSRC) identifiers (optional)																															
Header Extension (optional)																															

RTP Header

- RTCP:
  - Überwachung der Qualität der Datenkanäl: versandte Daten/Pakete, verlorene Pakete, Jitter, Round trip delay
  - Unterschiedliche Pakete stehen zur Verfügung: Sender report, receiver report, source description und anwendungsspezifische Pakete

Zusammenfassung Ethernet

- Ethernet ist aufgrund des CSMA/CD Zugriffsverfahrens für harte Echtzeitsysteme nicht geeignet:
  - unbestimmte Verzögerungen durch Backoff-Verfahren
  - keine Priorisierung von Nachrichten möglich
- Aufgrund der starken Verbreitung ($\Rightarrow$ niedrige Kosten, gute Unterstützung) wird Ethernet dennoch häufig in Echtzeitsystemen eingesetzt:
  - Durch Verwendung von echtzeitfähigen Protokollen in weichen Echtzeitsystemen (z.B. Multimediakontrolle).
  - Durch Verringerung der Kollisionswahrscheinlichkeit durch Aufteilung des Netzes in verschiedene Kollisionsdomänen (z.B. switched ethernet).
- Mittlerweile werden auch diverse Implementierungen von Real-Time Ethernet eingesetzt, allerdings gibt es noch keinen allgemein anerkannten Standard (siehe Zusammenfassung/Trends).

Echtzeitfähige Kommunikation

Carrier Sense Multiple Access/Collision Avoidance (CSMA/CA)

Vertreter: CAN

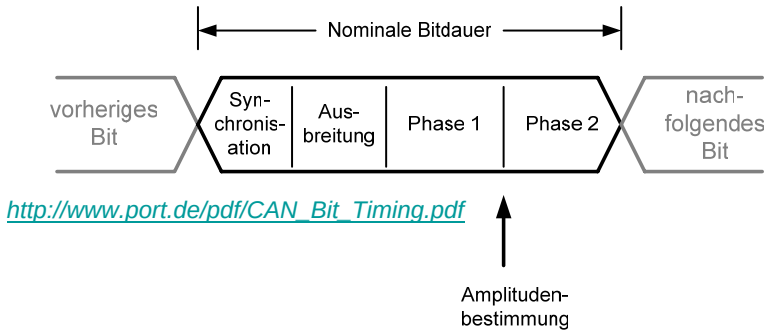
CAN-Protokoll

- Grundidee von Collision Avoidance:
  - Kollisionen werden rechtzeitig erkannt, bevor Nachrichten unbrauchbar werden
  - Wichtigere Nachrichten werden bevorzugt $\Rightarrow$ Priorisierung der Nachrichten
- Daten:
  - CAN (Controller Area Network) wurde 1981 von Intel und Bosch entwickelt.
  - Einsatzbereich: vor allem Automobilbereich, Automatisierungstechnik
  - Datenübertragungsraten von bis zu 1Mbit/s, Reichweite 1km
  - Implementierung der Schichten 1,2 und 7 des ISO/OSI-Modells

CAN: Schicht 1

- Busmedium:
  - Kupfer oder Glasfaser
  - Empfehlung Twisted Pair: Möglichkeit zur differentiellen Übertragung (robuster gegenüber Störungen)
- Codierung: NRZ-L (Non-Return-to-Zero-Level)
  - Problem mit NRZ-L: lange Sequenzen monotone Sequenzen von 0 oder 1 können zu Problemen bei der Synchronisation führen, in CAN wird deshalb nach fünf gleichen Bits ein inverses Bit eingefügt (**Bitstuffing**)
- Daten werden **bitsynchron** übertragen:

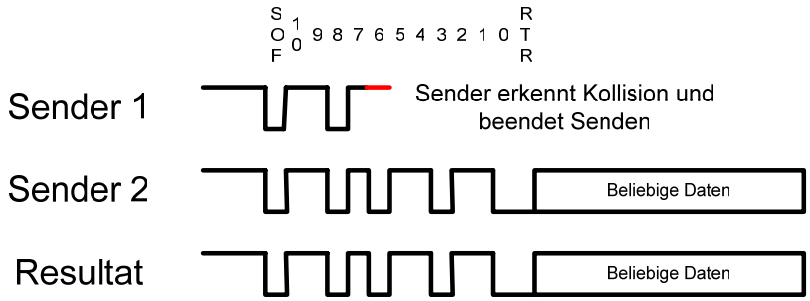
  ⇒ Datenübertragungsrate und maximale Kabellänge sind miteinander verknüpft.
  - Konfigurationsmöglichkeiten:
    - 1 MBit/s, maximale Länge: 40m
    - 500 kBit/s, maximale Länge: 100m
    - 125 kBit/s, maximale Länge: 500m
  - Maximale Teilnehmerzahl: 32-128



http://www.port.de/pdf/CAN_Bit_Timing.pdf

CAN: Schicht 2

- Realisierung eines CSMA/CA-Verfahrens:
  - Bei der Übertragung wirken Bits je nach Wert entweder **dominant** (typischerweise 0) oder **rezessiv** (1).
  - Dominante Bits überschreiben rezessive Bits, falls sie gleichzeitig gesendet werden.
  - Jedem Nachrichtentyp (z.B. Sensorwert, Kontrollnachricht) wird ein Identifikator zugewiesen, der die Wichtigkeit des Typs festlegt.
  - Jeder Identifikator sollte nur einem Sender zugewiesen werden.
  - Wie bei Ethernet wartet der Sender bis der Kanal frei ist und startet dann die Versendung der Nachricht.



  - Beim gleichzeitigen Senden zweier Nachrichten, dominiert der Identifikator des wichtigeren Nachrichtentyps, den Sender der unwichtigeren Nachricht beendet das Senden.

⇒ Verzögerung von hochprioren Nachrichten auf die maximale Nachrichtenlänge begrenzt (in Übertragung befindliche Nachrichten werden nicht unterbrochen)

CAN: Framearten

- Datenframe:
  - Versand von maximal 64bit Daten
- Remoteframe:
  - Verwendung zur Anforderung von Daten
  - Wie Datenframe, nur RTR-Feld auf 1 gesetzt
- Fehlerframe:
  - Signalisierung von erkannten Fehlerbedingungen
- Überlastframe:
  - Zwangspause zwischen Remoteframe und Datenframe

Länge in Bit	1	11	1	1	1	4	0...64	15	1	1	1	7	3
Zweck	Start of frame	Identifier (Extended CAN 27bit)	Remote Transmission Bit	Identifier Extension Bit	reserviert	Datenlängenfeld	Datenfeld	CRC-Prüfsumme	CRC Delimeter	Bestätigungsslot	Bestätigungsdelimeter	End of Frame	Intermission

CAN: Schicht 7

- Im Gegensatz zu Schicht 1 und 2 ist die Schicht 7 nicht in einer internationalen Norm spezifiziert.
- Es existieren jedoch diverse Implementierungen (z.B. CANOpen) für Dienste der Schichten 3-7 zur Realisierung von:
  - Flusskontrolle
  - Geräteadressierung
  - Übertragung größerer Datenmengen
  - Grunddienste für Anwendungen (Request, Indication, Response, Confirmation)
- Zudem gibt es Versuche eine Norm CAL (CAN Application Layer) einzuführen.
- Ziele:
  - Einheitliche Sprache zur Entwicklung von verteilten Anwendungen
  - Ermöglichung der Interaktion von CAN-Modulen unterschiedlicher Hersteller

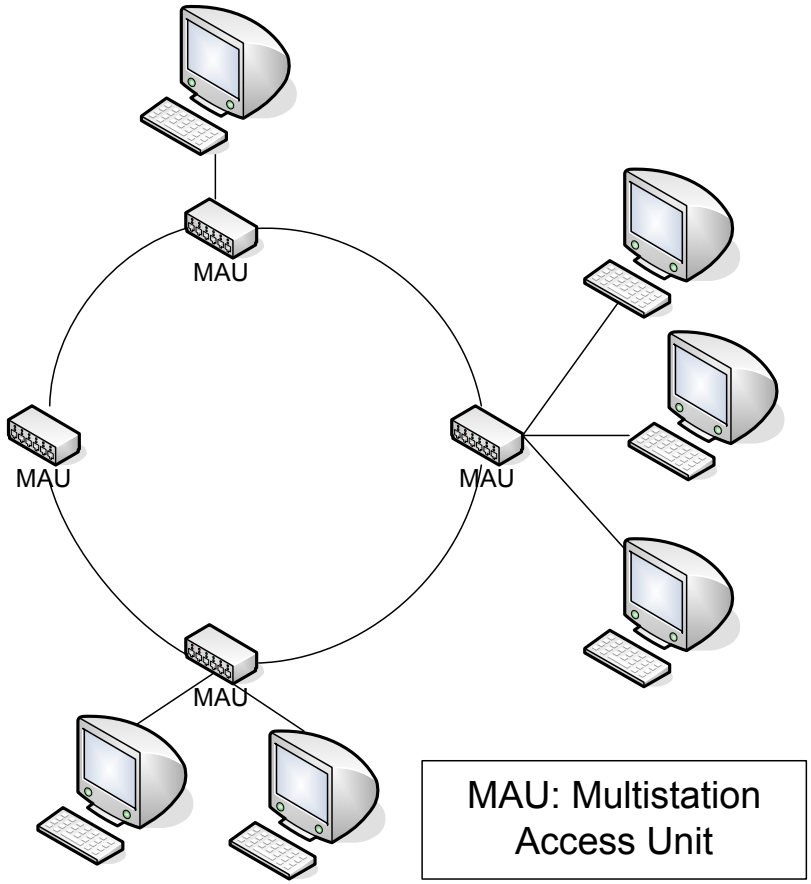
Echtzeitfähige Kommunikation

Tokenbasierte Verfahren

Vertreter: Token Ring

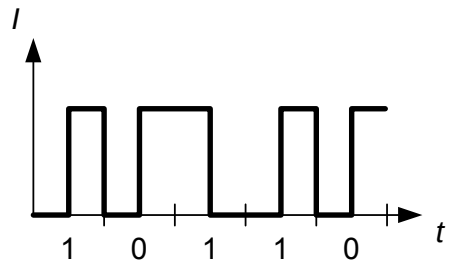
Tokenbasierte Verfahren

- Nachteil von CSMA/CA: Begrenzung der Datenrate und der Netzlänge durch Bitsynchronität
- Tokenbasierter Ansatz: Eine Einheit darf nur dann senden, wenn sie eine Berechtigung (Token) besitzt.
- Die Berechtigung wird zumeist zyklisch weitergegeben ⇒ Token Ring.
- Die Berechtigung / das Token ist dabei eine spezielle Bitsequenz.



Token Ring: Schicht 1

- Token Ring wird im Standard IEEE 802.5 spezifiziert.
- Erreichbare Geschwindigkeiten: 4 bzw. 16 MBit/s
  $\Rightarrow$ aufgrund der Kollisionsfreiheit mit den effektiven Datenübertragungsraten von 10 bzw. 100 MBit/s Ethernet vergleichbar
- Codierung:
  - differentieller Manchester-Code
  - somit selbstsynchronisierend
- Topologie:
  - Ring
  - aufgrund der möglichen Verwendung von MAUs auch sternförmige Verkabelung möglich



Differentieller Manchester-Code

Token Ring: Zugriffsverfahren

1. Die Station, die das Token besitzt, darf Daten versenden.
2. Das Datenpaket wird von Station zu Station übertragen.
3. Die einzelnen Stationen empfangen die Daten und regenerieren sie zur Weitersendung an den nächsten Nachbarn.
4. Der Empfänger einer Nachricht kopiert die Nachricht und leitet die Nachricht mit dem gesetzten C-Bit (siehe Nachrichtenaufbau) zur Empfangsbestätigung weiter.
5. Empfängt der Sender seine eigene Nachricht, so entfernt er diese aus dem Netz.
6. Nach Ende der Übertragung wird auch das Token weitergesendet (maximale Token-Wartezeit wird vorher definiert, Standardwert: 10ms)
7. Im 16 MBit/s Modus wird das Token direkt im Anschluß an das Nachrichtenpaket versendet (**early release**) $\Rightarrow$ es können sich gleichzeitig mehrere Token im Netz befinden

Token Ring: Prioritäten

- Token Ring unterstützt Prioritäten:
  - Insgesamt gibt es 8 Prioritätsstufen (3 Bit)
  - Jeder Station wird eine Priorität zugewiesen.
  - Der Datenrahmen besitzt ebenfalls einen Speicherplatz für die Priorität.
  - Eine Station kann in die Priorität in dem Prioritätsfeld von Nachrichten vormerken, allerdings darf die Priorität nur erhöht werden.
  - Stationen dürfen Tokens nur dann annehmen, wenn ihre Priorität mindestens so hoch ist, wie die Priorität des Tokens.
  - Applet zum Ablauf: http://www.nt.fh-koeln.de/vogt/mm/tokenring/tokenring.html

Token Ring: Token Paket

- Das Token besteht aus:
  - Startsequenz (1 Byte, JK0JK000)
    - J, K: Codeverletzungen entsprechend Manchester-Code (kein Übergang in Taktmitte)
  - Zugriffskontrolle (1 Byte, PPPTMRRR)
    - P: Zugriffspriorität
    - T: Tokenbit (0: freies Token, 1:Daten)
    - M: Monitorbit
    - R: Reservierungspriorität
  - Endsequenz (1 Byte, JK1JK1IE)
    - I: Zwischenrahmenbit (0: letztes Paket, 1: weitere Pakete folgen)
    - E: Fehlerbit (0: fehlerfrei, 1: Fehler entdeckt)

Token Ring: Tokenrahmen

- Der Datenrahmen besteht aus:
  - Startsequenz wie Token
  - Zugriffskontrolle wie Token
  - Rahmenkontrolle (1 Byte, FFrrZZZZ)
    - FF: Paketart (00: Protokollsteuerpaket, 01: Paket mit Anwenderdaten)
    - rr: reserviert für zukünftige Anwendungen
    - ZZZZ: Informationen zur Paketpufferung
  - Zieladresse (6 Byte): Adresse eines spezifischen Geräts oder Multicast-Adresse
  - Quelladresse (6 Byte)
  - Routing Informationen (0-30 Bytes): optional
  - Daten
  - Prüfsumme FCS (4 Byte): Berechung auf Basis der Daten zwischen Start- und Endsequenz
  - Endsequenz wie Token
  - Paketstatus (1 Byte ACrrACrr)
    - A: Paket wurde vom Empfänger als an in adressiert erkannt
    - C: Paket wurde vom Empfänger erfolgreich empfangen

Token Ring: Monitor

- Für den fehlerfreien Ablauf des Protokolls existiert im Token Ring ein Monitor.
- Aufgaben:
  - Entfernung von fehlerhaften Rahmen
  - Neugenerierung eines Tokens bei Verlust des Tokens (nach Ablauf einer Kontrollzeit)
  - Entfernung endlos kreisender Nachrichten bei Ausfall der Senderstation (Markierung der Nachricht beim Passieren des Monitors, Löschen der Nachricht beim 2. Passieren)
  - Signalisierung der Existenz des Monitors (durch Active Monitor Present Nachricht)

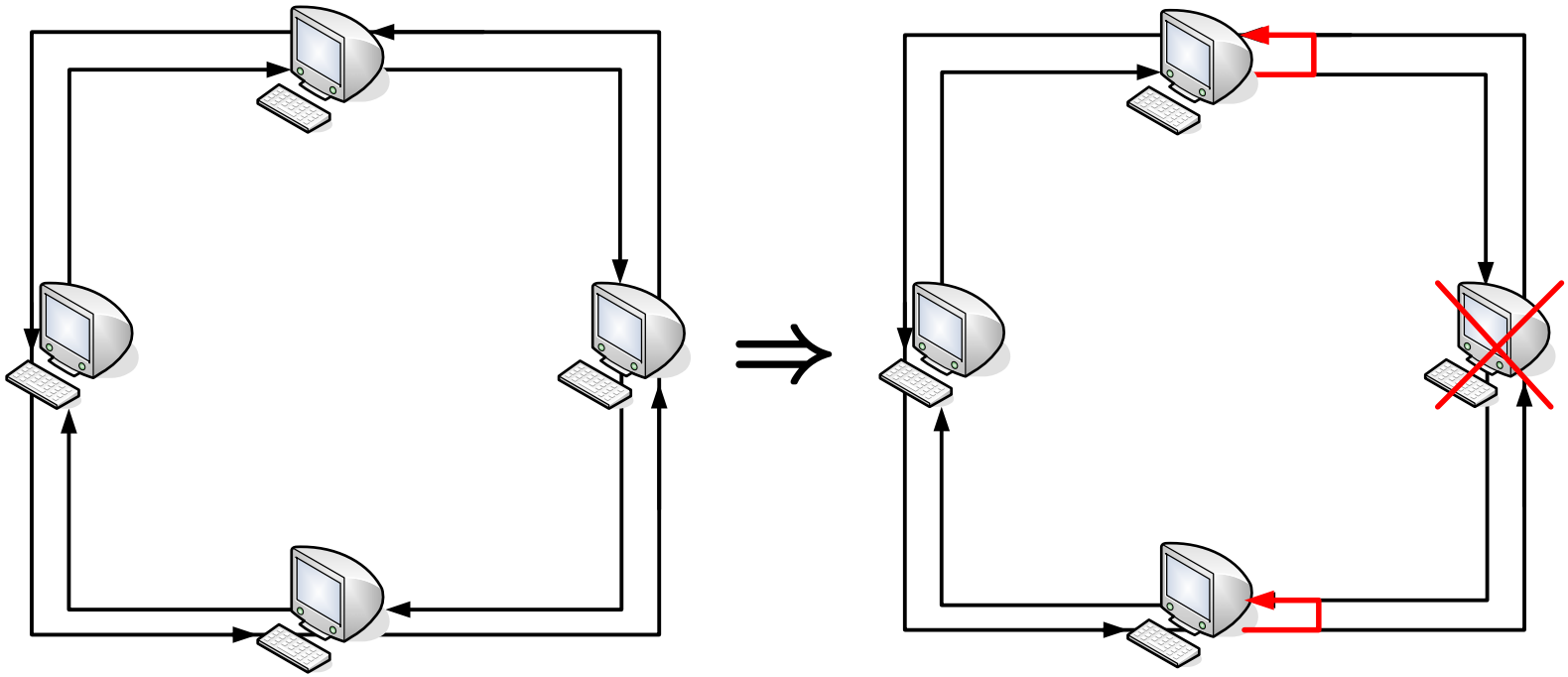
Token Ring: Initialisierung / Rekonfigurierung

- Bei der Initialisierung bzw. dem Ablauf des Standby Monitor Timer (Mechanismus zur Tolerierung des Ausfalls des Monitors)
  1. Senden eines Claim Token Paketes
  2. Überprüfung, ob weitere Pakete die Station passieren
  3. Falls nein $\Rightarrow$ Station wird zum Monitor
  4. Generierung eines Tokens
  5. Jede Station überprüft mittels des Duplicate Adress Test Paketes, ob die eigene Adresse bereits im Netzwerk vorhanden ist.
- Der Ausfall einer Station kann durch das Netzwerk erkannt werden und evtl. durch Überbrückung kompensiert werden.

FDDI

- Fiber Distributed Data Interface (FDDI) ist eine Weiterentwicklung von Token Ring
- Medium: Glasfaserkabel
- doppelter gegenläufiger Ring (aktiver Ring, Reservering) mit Token-Mechanismus
- Datenrate: 100 MBit/s, 1000 MBit/s
- Codierung: 4B5B (wie in FastEthernet)
- maximal 1000 Einheiten
- Ringlänge: max. 200 km
- Maximaler Abstand zwischen zwei Einheiten: 2 km
- Fehlertoleranz (maximal eine Station)
- Nachrichten können hintereinander gelegt werden (early release)
- Weitere Entwicklungen FDDI-2

Fehlerkonfiguration in FDDI



MAP / Token Bus

- **MAP**: **Manufacturing Automation Protocol** (Entwicklung ab 1982 von General Motors)
- Einsatz hauptsächlich im Produktionsbereich
- Schicht 1: anstelle von Ring-Topologie nun beliebige Topologie durch den Einsatz von Bridges, Gateways und Routern
- Medienzugriffsverfahren:
  - Token Bus, spezifiziert in IEEE 802.4
  - ähnlich Token-Ring, die benachbarte Station zur Weiterleitung des Tokens wird anhand einer Adresse bestimmt.
- In MAP werden zudem alle sieben Schichten des ISO/OSI-Modells spezifiziert.
- Aufgrund des Umfangs und der Komplexität konnte sich MAP nicht durchsetzen.
- Maximale Übertragungsrate: 10 MBit/s

Echtzeitfähige Kommunikation

Zeitgesteuerte Verfahren

Vertreter: TTP

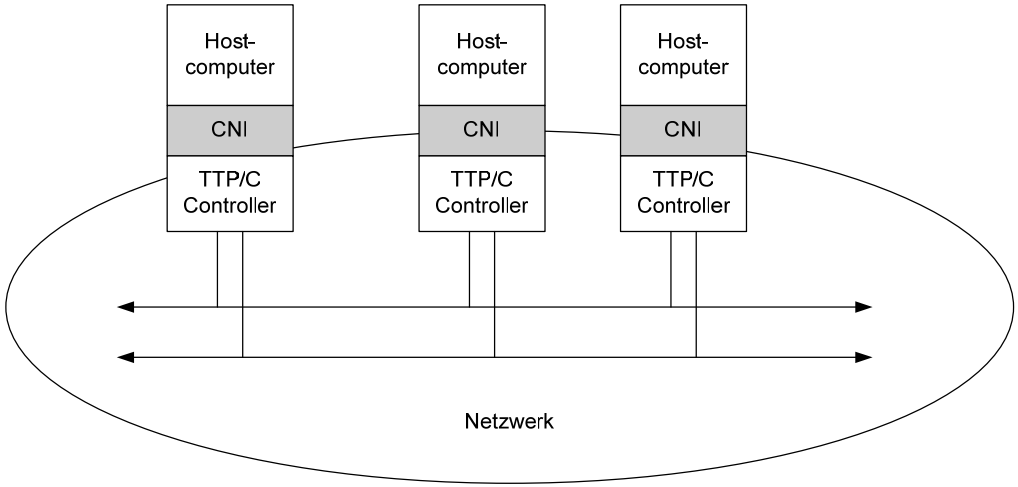
Zugriffsverfahren: TDMA

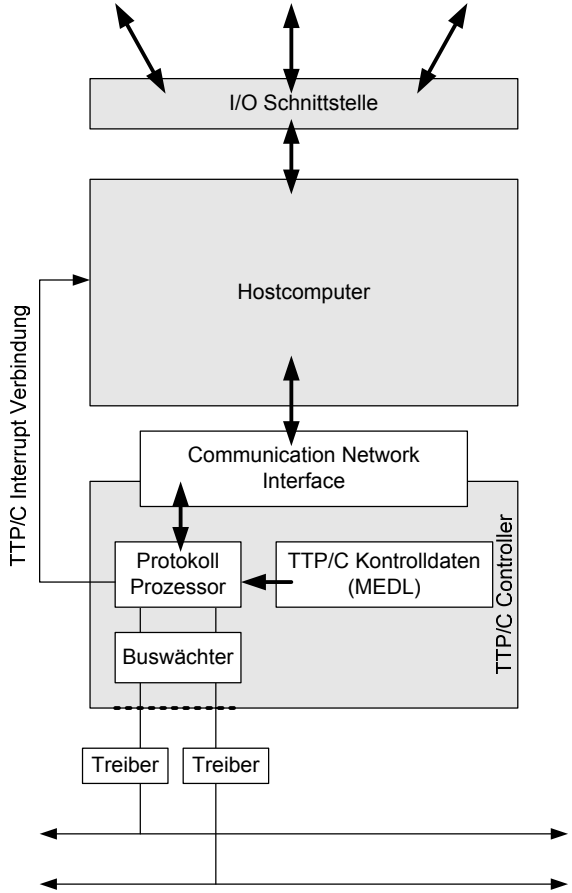
- **TDMA** (**Time Division Multiple Access**) bezeichnet ein Verfahren, bei dem der Zugriff auf das Medium in Zeitscheiben (slots) eingeteilt wird.
- Die Zeitscheiben werden für jeweils einen Sender zur Verfügung gestellt.
- Vorteile:
  - Kollisionen sind per Design ausgeschlossen
  - Einzelnen Sendern kann eine Bandbreite garantiert werden.
  - Das zeitliche Verhalten ist vollkommen deterministisch.
  - Synchronisationsalgorithmen können direkt im Protokoll spezifiziert und durch Hardware implementiert werden.
- Nachteil:
  - keine dynamische Zuteilung bei reinem TDMA-Verfahren möglich

Einführung TTP

- Entstanden an der TU Wien (SpinOff TTTech)
- TTP steht für Time Triggered Protocol
- TTP ist geeignet für harte Echtzeitsysteme:
  - verteilter, fehlertoleranter Uhrensynchronisationsalgorithmus (Einheit: 1 µs), toleriert beliebige Einzelfehler.
  - Zwei redundante Kommunikationskanäle ⇒ Fehlersicherheit
  - Einheiten werden durch Guards geschützt (Vermeidung eines babbling idiots).
  - Kommunikationsschema wird in Form einer **MEDL** (**Message Descriptor List**) a priori festgelegt und auf die Einheiten heruntergeladen.
- Einsatz unter anderem im Airbus A380

TTP-Architektur

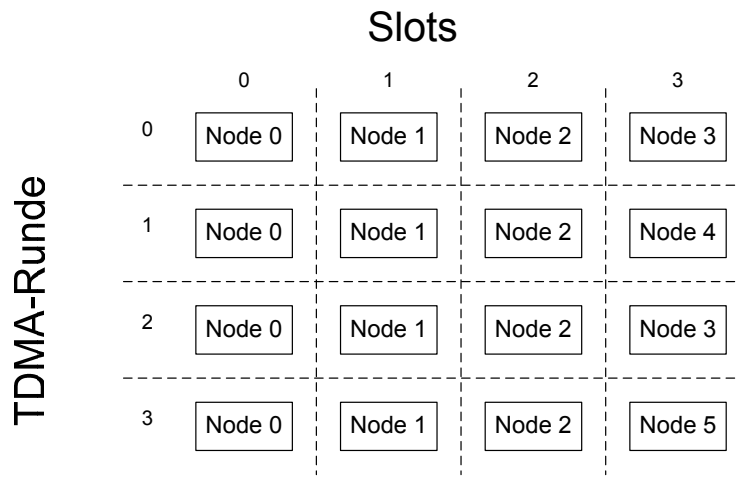




- Erläuterung:
  - Hostcomputer: Ausführung der eigentlichen Anwendung
  - CNI: Gemeinsamer Speicherbereich von Hostcomputer und TTP/C-Kontroller
  - Unterbrechungsverbindung: zur Übermittlung von Ticks der globalen Uhr und außergewöhnlicher Ereignisse an den Hostcomputer
  - MEDL: Speicherplatz für Kontrolldaten

TTP: Arbeitsprinzip

- Die Controller arbeiten autonom vom Hostcomputer (notwendige Daten sind in MEDL enthalten)
  - für jede zu empfangende und sendende Nachricht: Zeitpunkt und Speicherort in der CNI
  - zusätzliche Informationen zur Ausführung des Protokolls
- In jeder TDMA-Runde sendet ein Knoten genau einmal
  - Unterscheidung zwischen
    - reellen Knoten: Knoten mit eigenem Sendeschlitz
    - virtuelle Knoten: mehrere Knoten teilen sich einen Sendeschlitz
    - Die Länge der Sendeschlitze kann sich dabei unterscheiden, für einen Knoten ist die Länge immer gleich
      ⇒ TDMA-Runde dauert immer gleich lang



Protokolldienste

- Das Protokoll bietet:
  - Vorhersagbare und kleine, nach oben begrenzte Verzögerungen aller Nachrichten
  - Zeitliche Kapselung der Subsysteme
  - Schnelle Fehlerentdeckung beim Senden und Empfangen
  - Implizite Nachrichtenbestätigung durch Gruppenkommunikation
  - Unterstützung von Redundanz (Knoten, Kanäle) für fehlertolerante Systeme
  - Unterstützung von Clustermoduswechseln
  - Fehlertoleranter, verteilter Uhrensynchronisationsalgorithmus ohne zusätzliche Kosten
  - Hohe Effizienz wegen kleinem Protokollaufwand
  - Passive Knoten können mithören, aber keine Daten versenden.
  - Schattenknoten sind passive redundante Knoten, die im Fehlerfall eine fehlerhafte Komponente ersetzen können.

Fehlerhypothese

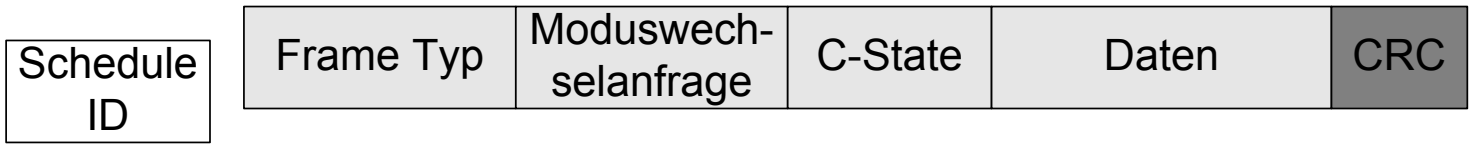
- Interne physikalische Fehler:
  - Erkennung einerseits durch das Protokoll, sowie Verhinderung eine babbling idiots durch Guards.
- Externe physikalische Fehler:
  - Durch redundante Kanäle können diese Fehler toleriert werden.
- Designfehler des TTP/C Kontrollers:
  - Es wird von einem fehlerfreien Design ausgegangen.
- Designfehler Hostcomputer:
  - Protokollablauf kann nicht beeinflusst werden, allerdings können inkorrekte Daten erzeugt werden.
- Permanente Slightly-Off-Specification-Fehler:
  - können durch erweiterte Guards toleriert werden.
- Regionale Fehler (Zerstören der Netzwerkverbindungen eines Knotens):
  - Folgen können durch Ring- und Sternarchitektur minimiert werden.

Zustandsüberwachung

- Das Protokoll bietet Möglichkeiten, dass Netzwerk zu analysieren und fehlerbehaftete Knoten zu erkennen.
- Der Zustand des Netzwerkes wird dabei im Kontrollerzustand (C-State) gespeichert.
- Der C-State enhält:
  - die globale Zeit der nächsten Übertragung
  - das aktuelle Fenster im Clusterzyklus
  - den aktuellen, aktiven Clustermodus
  - einen eventuell ausstehenden Moduswechsel
  - den Status aller Knoten im Cluster
- Das Protokoll bietet einen Votierungsalgorithmus zur Überprüfung des eigenen Zustands an.
- Ein Knoten ist korrekt, wenn er in seinem Fenster eine korrekte Nachricht versendet hat.
- Knoten können sich durch die Übernahme der Zeit und der Schedulingposition integrieren, sobald ein integrierender Rechner eine korrekte Nachricht sendet, erkennen in die anderen Knoten an.

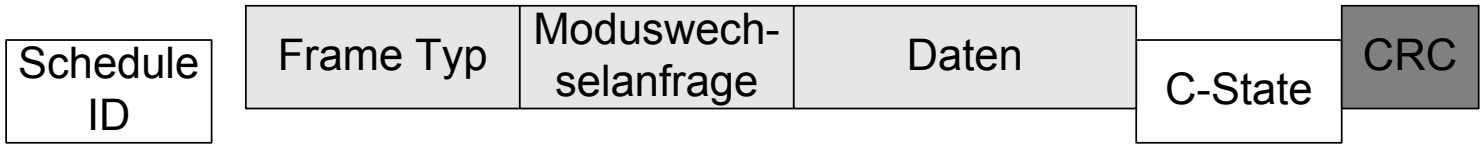
Datenpakete in TTP

- Paket mit explizitem C-State



- Kaltstartpaket

- Paket mit implizitem C-State



In Frame enthalten, in CRC eingerechnet	Nicht in Frame enthalten, in CRC eingerechnet	Berechneter CRC

TTP: Clusterstart

- Der Start erfolgt in drei Schritten:
  1. Initialisierung des Hostcomputers und des Controllers
  2. Suche nach Frame mit expliziten C-State und Integration
  3. a) Falls kein Frame empfangen wird, werden die Bedingungen für einen Kaltstart geprüft:
     - Host hat sein Lebenszeichen aktualisiert
     - Das Kaltstart Flag in der MEDL ist gesetzt
     - die maximale Anzahl der erlaubten Kaltstarts wurde noch nicht erreicht

     Sind die Bedingungen erfüllt, sendet der Knoten ein Kaltstartframe.
  3. b) Falls Frame empfangen wird: Versuch zur Integration

TTP: Sicherheitsdienste / Synchronisation

- Sicherheitdienste:
  - Korrektheit: Alle Knoten werden über die Korrektheit der anderen Knoten mit einer Verzögerung von etwa einer Runde informiert.
  - Cliquenentdeckung: Es werden die Anzahl der übereinstimmenden und entgegen gesetzten Knoten gezählt. Falls mehr entgegen gesetzte Knoten gezählt werden, so wird ein Cliquenfehler angenommen.
  - Host/Kontroller Lebenszeichen: der Hostcomputer muss seine Lebendigkeit dem Kontroller regelmäßig zeigen. Sonst wechselt der Kontroller in den passiven Zustand.
- Synchronisation:
  - In regelmäßigen Abständen wird die Uhrensynchronisation durchgeführt.
  - Es werden die Unterschiede der lokalen Uhr zu ausgewählten (stabilen) Uhren (mind.4) anderer Rechner anhand den Sendezeiten gemessen.
  - Die beiden extremen Werte werden gestrichen und vom Rest der Mittelwert gebildet.
  - Die Rechner einigen sich auf einen Zeitpunkt für die Uhrenkorrektur.

Echtzeitfähige Kommunikation

Zusammenfassung

Zusammenfassung

- Die Eignung eines Kommunikationsmediums für die Anwendung in Echtzeitsystemen ist vor allem durch das Medienzugriffsverfahren bestimmt.
- Die maximale Wartezeit ist bei
  - CSMA/CD: unbegrenzt und nicht deterministisch ($\Rightarrow$ keine Eignung für Echtzeitsysteme)
  - CSMA/CA,tokenbasierten Verfahren: begrenzt, aber nicht deterministisch (abhängig von anderen Nachrichten)
  - zeitgesteuerten Verfahren: begrenzt und deterministisch.
- Die Priorisierung der Nachrichten wird von CSMA/CA und tokenbasierten Verfahren unterstützt.
- Nachteil der zeitgesteuerten Verfahren ist die mangelnde Flexibilität (keine dynamischen Nachrichten möglich).
- Trotz diverser Nachteile geht der Trend hin zum Ethernet.

Trends: Real-Time Ethernet

- Es existieren verschiedene Ansätze
  - Beispiel: Ethercat von Beckhoff
    - Die Nachrichten entsprechen dem Standardnachrichtenformat von Ethernet
    - Pakete werden von einem Master initiiert und werden von den Teilnehmern jeweils weitergeleitet.
    - Jeder Knoten entnimmt die für ihn bestimmten Daten und kann eigene Daten anfügen.
    - Die Bearbeitung erfolgt on-the-fly, dadurch kann die Verzögerung minimiert werden.
  - Beispiel: Profinet von Siemens
    - Drei verschiedene Protokollstufen (TCP/IP – Reaktionszeit im Bereich von 100ms, Real-time Protocol - bis 10ms, Isochronous Real-Time - unter 1ms)
    - Profinet IRT benutzt vorher bekannte, reservierte Zeitschlitze zur Übertragung von echtzeitkritischen Daten, in der übrigen Zeit wird das Standard-Ethernet Protokoll ausgeführt

Kapitel 7

Uhren & Synchronisation

Inhalt

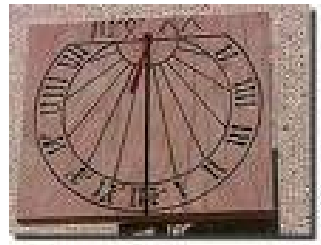
- Motivation
  - Definition Zeit
- Uhren
- Synchronisation
  - Algorithmus von Cristian
  - Algorithmus aus Berkeley
  - NTP-Protokoll
  - Synchronisation bei fehlerbehafteten Uhren

Literatur

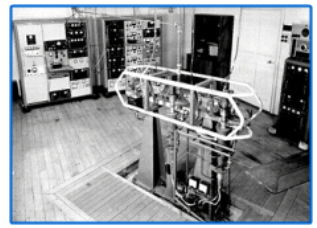
- Links zum Thema Zeit:
  - http://www.ptb.de/de/zeit/uhrzeit.html
  - http://www.maa.mhn.de/Scholar/dt_times.html
- Uhrensynchronisation:
  - Leslie Lamport: Synchronizing clocks in the presence of faults, 1985
  - http://www.ntp.org/

Definition Zeit

- Historisch:
  - Jeden Tag gegen Mittag erreicht die Sonne ihren höchsten Punkt am Himmel.
  - Die Zeitspanne zwischen zwei aufeinander folgenden Ereignissen dieses Typs heißt Tag (genauer gesagt: ein Sonnentag).
  - Eine Sonnensekunde ist 1/86400 dieser Spanne.
- Zeitmessung heute:
  - Verwendung von Atomuhren: eine Sekunde ist die 9.192.631.770-fache Periodendauer, der dem Übergang zwischen den beiden Hyperfeinstrukturniveaus des Grundzustands von ^{133}Cäsium-Atomen entsprechenden Strahlung.
  - Am 01.01.1958 entsprach die Atomsekunde genau einer Sonnensekunde.
  - Aufgrund von unregelmäßigen Schwankungen, sowie einer langfristigen Verlangsamung der Erdrotation unterscheiden sich die Atomsekunde und die Sonnensekunde.



Sonnenuhr
Deutsches Museum



Erste Cäsiumatomuhr

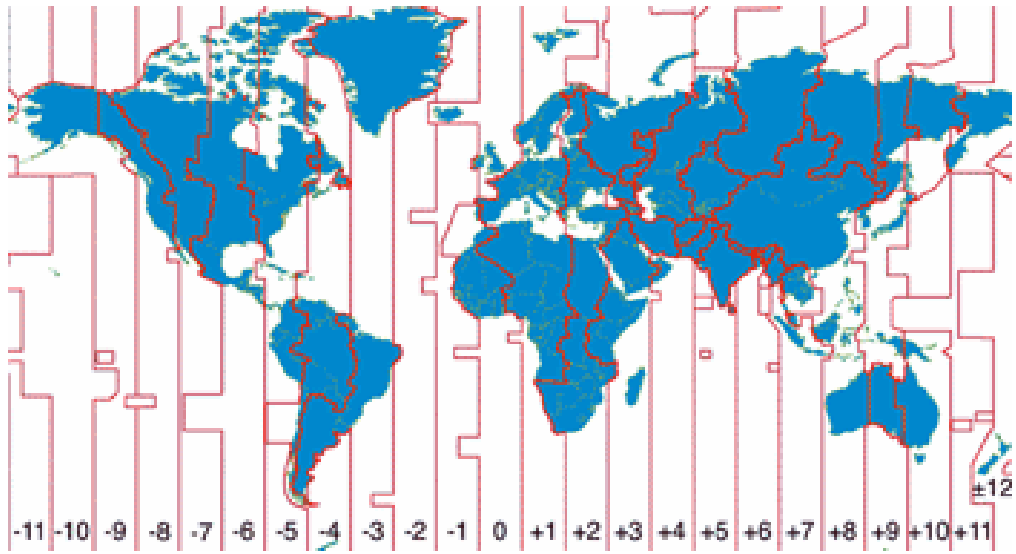
TAI (Temps Atomique International)

- TAI: Atomzeitskala, die zur Koordination nationaler Atomzeiten ermittelt wird:
  - Beteiligung von 50 verschiedene Zeitinstitute mit ca. 250 Atomuhren
  - Zeit basiert auf der Atomsekunde
  - Referenzzeitpunkt ist der 1.Januar 1970
  - relative Genauigkeit von +/ - 10^{-15}, aber keine exakte Übereinstimmung mit der Sonnenzeit

Atomuhr der Physikalisch-Technischen Bundesanstalt in Braunschweig

UTC (Coordinated Universal Time)

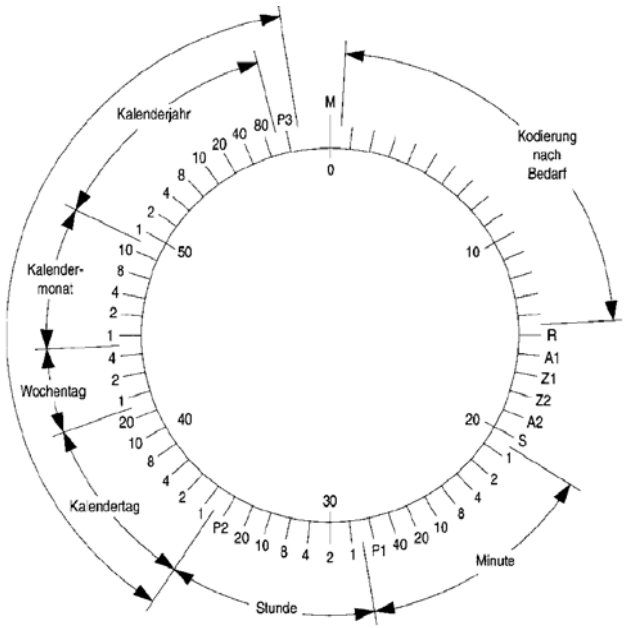
- eigentlicher Nachfolger der Greenwichzeit
- realisiert durch Atomuhren, die Zeiteinheit ist die SI-Sekunde
  ⇒ hochkonstante Zeiteinheit
- zusätzlich Übereinstimmung mit dem Sonnenlauf
  ⇒ einheitliche Grundlage zur Zeitbestimmung im täglichen Leben
- Durch Einfügen von Schaltsekunden wird die UTC mit der universellen Sonnenzeit (UT1) synchronisiert
- Anpassung erfolgt zumeist zu Ende oder Mitte des Jahres (typischer Abstand: alle 18 Monate)



Zeitzonen

DCF77

- Das PTB überträgt die aktuelle Uhrzeit über den Langwellensender DCF77
- Die Zeitinformationen werden als digitales Signal (negative Modulation ⇒ Absenkung der Trägeramplitude) im Sekundentakt übertragen.



- '0' und '1' werden durch eine Absenkung um 100ms bzw. 200
ms codiert. In der Sekunde 59 erfolgt keine Absenkung
⇒
Markierung der Beginn einer neuen Minute bei nächster Amplitudenabsenkung.
- Pro Minute stehen somit 59 Bit zur Verfügung (wobei Bit 0-14 für Betriebsinformationen verwendet werden)

Uhren und Synchronisation

Uhren

Aufgaben

- Absolutzeitgeber
  - Datum, Uhrzeit
  - zeitabhängige Aufträge
  - Zeitstempel, Logbuch
  - Ursache-Wirkung-Feststellung
- Relativzeitgeber
  - Verzögerungen
  - Messen von Zeitabständen
  - Zyklische Ausführung, Messungen
  - Zeitüberwachung von Wartezuständen




Genauigkeit von Uhren

- Eine Uhr arbeitet korrekt, wenn sie die vom Hersteller angegebene maximale Driftrate τ einhält, auch wenn sie dann etwas zu schnell oder zu langsam ist.
- Typische Driftraten:

Uhrentyp	**Driftrate τ**	**Abweichung pro Jahr**
Quarzuhr	10^{-5}	$\pm$ 300 sec
Pendeluhr	10^{-6}	$\pm$ 30 sec
Atomuhr	$1{,}5*10^{-14}$	$\pm$ 0,5 Mikrosekunden
Atomuhr (lasergekühlte Atome)	10^{-15}	$\pm$ 0.03 Mikrosekunden

Uhrenverhalten

- Korrekt:
  1. Absolutwert der Abweichung kleiner der zugesicherten Gangabweichung
- Fehlerbehaftet:
  2. Überschreiten der zugesicherten Gangabweichung
  3. Zustandsfehler (z.B. Sprung im Zählerwert)
  4. Stehenbleiben der Uhr
- Unmöglich:
  5. Rückwärtslaufende Uhr
  6. Unendlich schnell laufende Uhr

$\Rightarrow$ Die Gangabweichung zweier korrekter Uhren kann beliebig groß werden, wenn die Uhren nicht synchronisiert sind.

Uhren und Synchronisation

Synchronisation

Grundlagen

- Folgende Annahmen werden im Zusammenhang mit der Synchronisation gemacht:
  1. Alle Uhren besitzen zu Beginn in etwa die gleiche Zeit.
  2. Die Uhren fehlerfreier Prozesse gehen annähernd richtig, d.h. sie besitzen eine ähnliche Ganggenauigkeit.
  3. Ein fehlerfreier Prozess *p* kann die Differenz seiner Uhr von der Uhr von Prozess *q* mit einer Genauigkeit ε bestimmen.
- Anforderungen an die Synchronisation:
  1. Zu jedem Zeitpunkt zeigen die Uhren zweier fehlerfreier Prozesse ungefähr den gleichen Wert.
  2. Durch die Synchronisation entstehen keine bzw. nur sehr kleine Zeitsprünge
  3. Insbesondere darf die Kausalität nicht verletzt werden (z.B. Zurückstellen der Zeit)

  $\Rightarrow$ Notwendig, da sonst keine konsistente Ausführung (z.B. wegen Anweisungen mit absoluten Zeitangaben) garantiert werden kann.

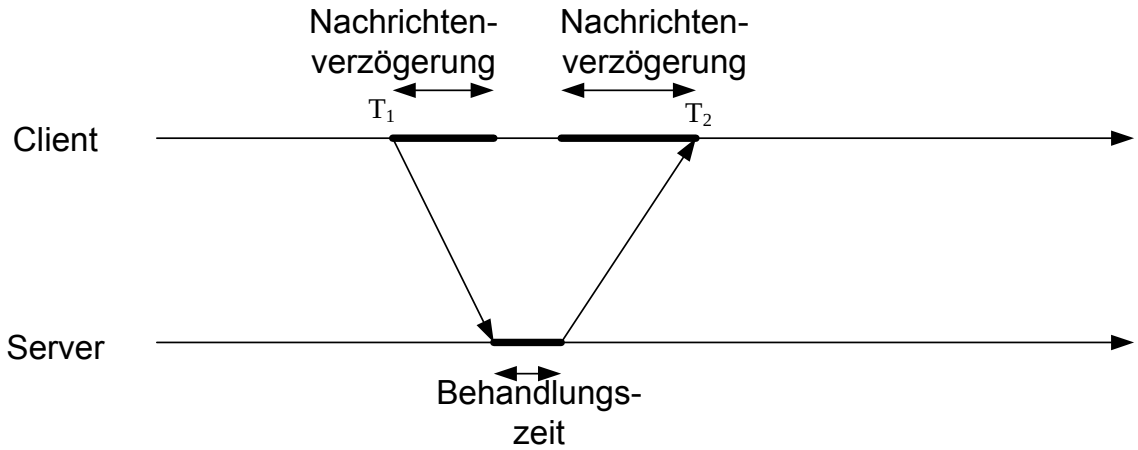
Arten der Synchronisation

- Zeitpunkt: typischerweise erfolgt die Synchronisation periodisch
- Rollen der Knoten:
  - externe Synchronisation: die Synchronisation erfolgt anhand einer externen, als perfekt angenommenen Uhr
  - interne Synchronisation: die Uhren ermitteln basierend auf den einzelnen Zeitwerten eine korrekte, globale Zeitbasis

  Vorteil der externen Synchronisation: der maximal tolerierte Fehler kann halb so groß wie bei der internen Synchronisation gewählt werden.
- Ort der Synchronisation:
  - zentrale Synchronisation: Synchronisation wird von einer Einheit koordiniert ⇒ fehleranfällig
  - verteilte Synchronisation: alle Einheiten berechnen die globale Zeitbasis ⇒ hohes Datenaufkommen

Algorithmus von Cristian (1989)

- Das Verfahren basiert auf verteilter, externer Synchronisation.
- Innerhalb des Systems existiert ein Time-Server, zumeist ein UTC-Empfänger.
- In regelmäßigen Abständen senden die anderen Einheiten einen Time-Request, der so schnell wie möglich vom Server beantwortet wird.

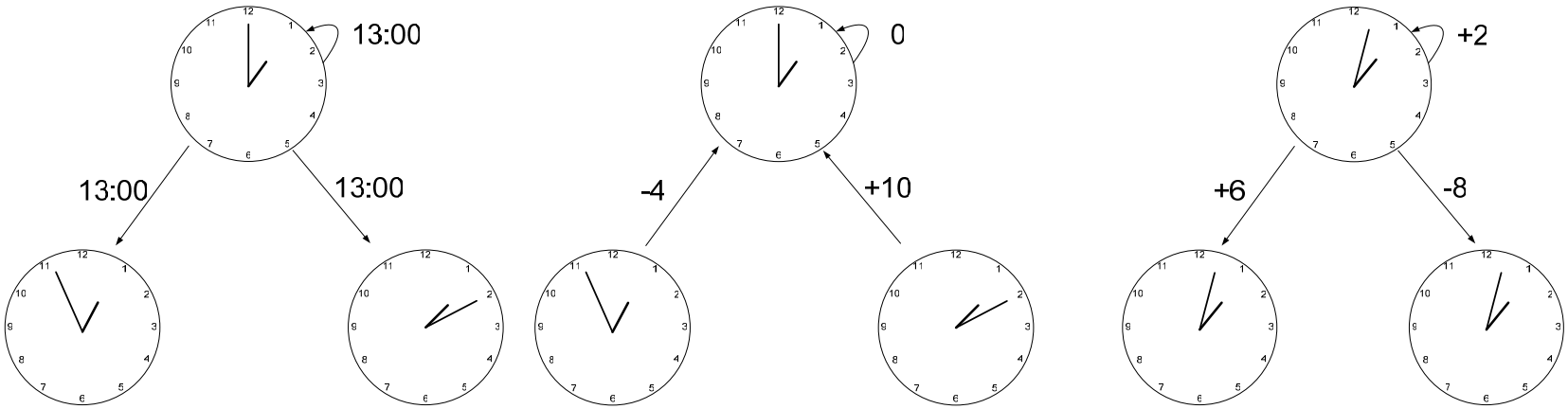


Algorithmus von Cristian

- Nach Empfang kann die Uhr auf die empfangene Uhrzeit gesetzt werden.
- **1. Problem**: Zeitsprünge würden entstehen.
- **Lösung**: Die Uhr wird graduell angepasst (Beispiel: Herabsetzung des Intervalls zwischen zwei Uhrenticks von 1ms auf 0.9ms, falls lokale Uhr zu langsam war).
- **2. Problem**: Nachricht ist veraltet, wenn die Nachrichtenverzögerung nicht vernachlässigbar ist.
- **Lösung**: Messung der Nachrichtenverzögerung
  - Abschätzung, falls Informationen fehlen: $(T_1-T_2)/2$
  - Falls die Behandlungszeit bekannt ist, kann das Ergebnis weiter verbessert werden.
  - Zusätzliche Verbesserung: Ermittelung eines Durchschnittswertes, Ausreißer müssen dabei außer acht gelassen werden.

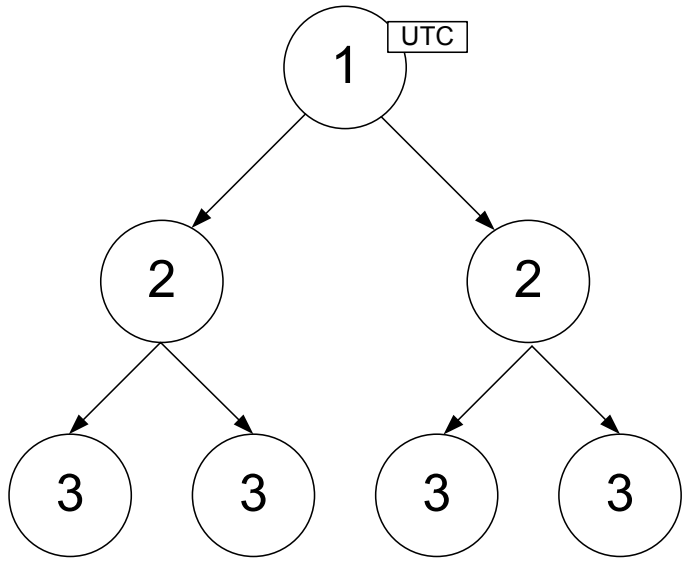
Algorithmus von Berkeley (1989)

- Annahme: kein UTC-Empfänger verfügbar
- Algorithmus (zentral, intern):
  - ein Rechner agiert als aktiver Time-Server.
  - Der Server fragt periodisch die Zeiten/Unterschiede aller anderen Rechner ab (Phase 1) und ermittelt den Durchschnittswert (Phase2).
  - In Phase 3 wird der errechnete Wert an alle anderen Uhren ausgegeben.



NTP: Network Time Protocol (1982)

- Problem: Die angegebenen Algorithmen funktionieren nur in kleinen statischen Netzen.
- Das NTP Protokoll bietet eine Möglichkeit in großen Netzen eine Synchronisation zu gewährleisten.
- Die Netze können dabei dynamisch konfiguriert werden, um eine zuverlässige Synchronisation zu gewährleisten.
- Die Grundstruktur von NTP ist ein hierarchisches Modell (mit verschiedenen Strata/Schichten).
  - Der Dienst wird durch ein verteiltes Serversystem geleistet.
  - Primäre Server sind direkt mit einer UTC-Quelle verbunden.
  - Sekundäre Server synchronisieren sich mit primären Servern usw.
  - Jede zusätzliche Schicht verursacht einen zusätzlichen Zeitversatz von 10-100ms.



Uhren und Synchronisation

Synchronisation bei fehlerbehafteten Uhren

Problemstellung

- Die bisherigen Algorithmen basierten alle auf der Annahme von fehlerfreien Uhren.
- Im Folgenden werden Algorithmen betrachtet, die mit einer maximalen Anzahl von *m* fehlerbehafteten Uhren umgehen können.
- Insgesamt soll das System aus n Uhren bestehen. Betrachtet werden im Besonderen auch byzantinische Fehler (die fehlerhafte Einheit kann beliebige Ausgaben produzieren).
- Die maximal zulässige Abweichung zweier Uhren bezeichnen wir mit ε.
- In Frage kommen dabei nur verteilte Algorithmen, um einen Single-Point-of-Failure auszuschließen.

Konvergenzalgorithmus (Leslie Lamport, 1985)

- Algorithmus:
  - Jede Einheit liest die Uhr der anderen Rechner und berechnet den Mittelwert.
  - Ist die Abweichung einer Uhr größer als ε, so verwendet der Algorithmus stattdessen den Wert der eigenen Uhr.
- Aussage:
  - Der Algorithmus arbeitet erfolgreich, falls gilt: $n \geq 3m$.
- Annahmen:
  - vernachlässigbare Ausführungszeit
  - Einheiten lesen zeitgleich die Uhren ab bzw. Unterschiede sind vernachlässigbar

Konvergenzalgorithmus (Leslie Lamport, 1985)

- Beweis:
  - Seien *p*,*q* zwei fehlerfreie Einheiten, *r* eine beliebige Einheit.
  - Sei *t(p,r)* die Uhrzeit, die die Einheit *p* für die Mittelwertsberechnung verwendet.

    $\Rightarrow$ r fehlerfrei: $t(p,r) \approx t(q,r)$

    $\Rightarrow$ r fehlerbehaftet $|t(p,r)-t(q,r)|<3\varepsilon$
  - Einheit p stellt seine Uhr auf: $1/n * \sum_r t(p,r)$
  - Einheit q stellt seine Uhr auf: $1/n * \sum_r t(q,r)$
  - Schlechtester Fall:
    - (n-m) Uhren fehlerfrei: $t(p,r) \approx t(q,r)$
    - m Uhren fehlerbehaftet $|t(p,r)-t(q,r)|<3\varepsilon$

  $\Rightarrow$ Differenz beider Uhren: $\Delta(p,q)=1/n\ |\sum_r t(p,r)-\sum_r t(q,r)| \leq m/n*3\varepsilon < \varepsilon$

Kapitel 9

Fehlertoleranz

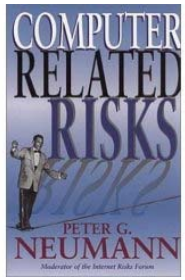
Inhalt

- Einleitung
- Grundlagen
- Fehlertoleranzmechanismen
- Quantitative Bewertung fehlertoleranter Systeme

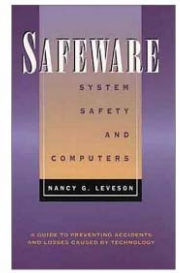
Literatur

Dhiraj K. Pradhan: Fault-Tolerant Computer System Design, Prentice Hall 1996

W.A.Halang, R.Konakovsky: Sicherheitsgerichtete Echtzeit-systeme, Oldenburg 1999



Peter G.Neumann: Computer Related Risks, ACM Press 1995



Nancy G.Leveson: Safeware, Addison-Wesley 1995

Klaus Echtle: Fehlertoleranzverfahren, Springer-Verlag 1990 (elektronisch unter http://dc.informatik.uni-essen.de/Echtle/all/buch ftv/)

http://www.system-safety.org/

Fehlertoleranz

Negativbeispiele (Motivation)

Ariane 5 (1996)

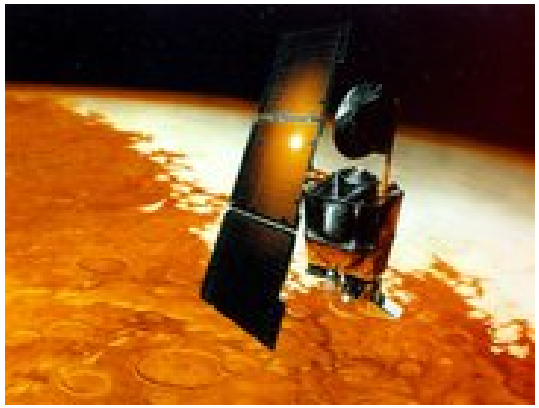
- Selbstzerstörung bei Jungfernflug:
- Design:
  - 2 redundante Meßsysteme (identische Hardware und Software) bestimmen die Lage der Rakete (hot-standby)
  - 3-fach redundante On-Board Computer (OBC) überwachen Meßsysteme
- Ablauf:
  - Beide Meßsysteme schalten aufgrund eines identischen Fehlers ab
  - OBC leitet Selbstzerstörung ein
- Ursache:
  - Wiederverwendung von nicht-kompatiblen Komponenten der Ariane 4 (Speicherüberlauf, weil Ariane 5 stärker beschleunigt)

Weitere Informationen unter
http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html

Therac-25 (1985-1987)

- Computergesteuerter Elektronenbeschleuniger zur Strahlentherapie
- Das System beinhaltete 3 schwere Mängel:
  - Sicherheitsprüfungen im Programm wurden durch einen Softwarefehler bei jeder 64. Benutzung ausgelassen (wenn ein 6-bit Zähler Null wurde).
  - Behandlungsanweisungen konnten mittels Editieren am Bildschirm so abgeändert werden, dass die Maschine für die nächste Behandlung nicht den gewünschten Zustand einnahm (nämlich Niederintensität).
  - Mehrere Sicherheitsverriegelungen, die beim Vorgängermodell Therac-20 in Hardware realisiert waren, wurden nicht übernommen, sondern durch Software ersetzt.
- Folgen:
  - Mehrere Patienten erhielten anstatt der vorgesehenen Dosis von 80-200 rad Strahlungsdosen von bis zu 25000 rad (mehrere Tote und Schwerverletzte).
- Weitere Informationen unter http://sunnyday.mit.edu/papers/therac.pdf

Mars Climate Orbiter (1998)



- Verglühen beim Eintritt in die Atmosphäre
- Ursache:
  - Verwendung von unterschiedlichen Maßeinheiten (Zoll, cm) bei der Implementierung der einzelnen Komponenten.
  - Mangelnde Erfahrung, Überlastung und schlechte Zusammenarbeit der Bodenmannschaften

Weitere Informationen unter http://mars.jpl.nasa.gov/msp98/orbiter/

Explosion einer Chemiefabrik (1992)

- Explosion einer holländischen Chemiefabrik aufgrund eines Bedienfehlers
- Ablauf:
  - Computergesteuertes Mischen von Chemikalien.
  - Operateur (in Ausbildung) verwechselt beim Eintippen eines Rezeptes 632 (Harz) mit 634 (Dicyclopentadien).
- Folgen:
  - Explosion fordert 3 Menschenleben, Explosionsteile finden sich noch im Umkreis von 1 km.