Echtzeitsysteme
Wintersemester 2008/2009

Prof. Dr. Alois Knoll, Christian Buckl

TU München

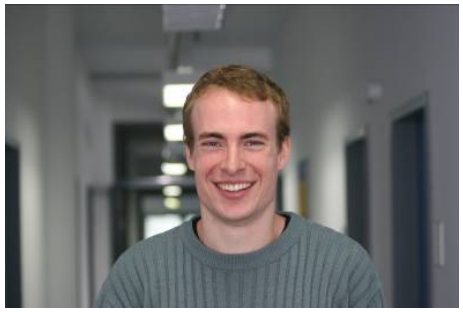
Lehrstuhl VI Robotics and Embedded Systems

Echtzeitsysteme: Organisation

Team

Prof. Dr. Alois Knoll

Christian Buckl



Übungen: Simon Barner, Michael Geisinger, Stephan Sommer

Homepage der Vorlesung mit Folien, Übungsaufgaben und weiterem Material:
http://www6.informatik.tu-muenchen.de/Main/TeachingWs2008Echtzeitsysteme

Bestandteile der Vorlesung

- Vorlesung:
  - Dienstag 10:15-11:45 Uhr MI HS 2
  - Mittwoch 12:15-13:00 Uhr MI HS 2
  - 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 29.10.2008
- Prüfung:
  - Schriftliche Klausur am Ende des Wintersemesters, falls ein Schein benötigt wird.

Informationen zur Übung

- Ziel: Praktisches Einüben von Vorlesungsinhalten
- Übungsaufgaben werden in 2er Gruppen am Computer gelöst
- Platz ist begrenzt (7 Computer ⇔ 14 Studenten) ⇒ **Anmeldung erforderlich**
- Übungsaufgaben sind auch auf der Vorlesungsseite verfügbar
- 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
- Die Anmeldung erfolgt über Email an buckl@in.tum.de
- Die Übungsinhalte sind nicht direkt prüfungsrelevant, tragen aber stark zum Verständnis bei.

Mögliche Übungsinhalte

- Modellierungssprachen/-werkzeuge (Esterel Studio, SCADE, Ptolemy, EasyLab, FTOS)
- Programmierung von Echtzeitsystemen (Programmiersprache C, Betriebssysteme VxWorks, PikeOS)
  - Nebenläufigkeit (Threads, Prozesse)
  - Interprozesskommunikation / Synchronisation: Semaphore, Signale, Nachrichtenwarteschlangen
  - Unterbrechungsbehandlung
- Kommunikation (Ethernet, CAN)
- Programmierung von Adhoc-Sensornetzwerken (TinyOS, ZigBee)
- Umsetzung von diversen Demonstratoren (Aufzug, Kugelfall, Murmelbahn)

- **Ihre Rückmeldung ist wichtig!!!**

Klausur

- Für Studenten, die einen Schein benötigen, wird am Ende der Vorlesung eine schriftliche Klausur angeboten.
- Stoff der Klausur sind die Inhalte der Vorlesung.
- Die Inhalte der Übung sind nicht direkt prüfungsrelevant, tragen allerdings zum Verständnis des Prüfungsstoffes bei.
- Voraussichtlicher Termin: letzte Vorlesungswoche (in Absprache mit den Studenten), vermutlich 03.02.2008 10:00-12:00 Uhr
- Voraussichtlich erlaubte Hilfsmittel: keine

Grundsätzliches Konzept

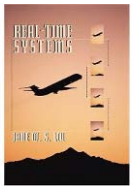
- Themen werden aus verschiedenen Blickrichtungen beleuchtet:
  - Stand der Technik in der Industrie
  - Stand der Technik in der Wissenschaft
  - Existierende Werkzeuge
  - Wichtig: nicht die detaillierte Umsetzung, sondern die Konzepte sollen erlernt werden
- Praktische Aufgaben in der Vorlesung und der Übung und Analogien zum Alltag werden zur Verdeutlichung theoretischer Inhalte verwendet
- In jedem Kapitel werden die relevanten Literaturhinweise referenziert
- Zur Erfolgskontrolle werden Klausuraufgaben der letzten Jahre am Ende eines Kapitels diskutiert
- Wir freuen uns über Fragen, Verbesserungsvorschläge und Feedback

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, Modellierungswerkzeuge, Busprotokolle, Objekterkennung und Lernen, Neurocomputing,
- Diplomarbeiten / Masterarbeiten
- Systementwicklungsprojekte / Bachelorarbeiten
- Guided Research, Stud. Hilfskräfte
- Unser gesamtes Angebot finden sie unter http://wwwknoll.in.tum.de

Literatur

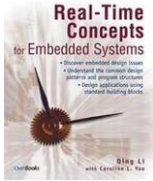
Hermann Kopetz: Real-Time Systems (Überblick)



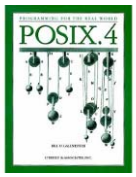
Jane W. S. Liu: Real-Time Systems
(Überblick, Schwerpunkt Scheduling)

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. Fehlertolerante Systeme
10. Spezielle Hardware
11. Regelungstechnik

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)
  - Allgemeine Einführung
  - Grundsätzlicher Aufbau, Models of Computation, Ptolemy
  - Synchrone Sprachen (Esterel, Lustre), SCADE, EasyLab
  - Zeitgesteuerte Systeme: Giotto, FTOS, TTA
  - Exkurs: Formale Methoden

Inhalt II

- Kapitel Nebenläufigkeit (2 Vorlesungswochen)
  - Prozesse, Threads
  - Interprozesskommunikation
- Kapitel Scheduling (2 Vorlesungswochen)
  - Kriterien
  - Planung Einrechner-System, Mehrrechnersysteme
  - EDF, Least Slack Time
  - Scheduling mit Prioritäten (FIFO, Round Robin)
  - Scheduling periodischer Prozesse
  - Scheduling Probleme

Inhalt III

- Kapitel Echtzeitbetriebssysteme (1 Vorlesungswoche)
  - 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 Fehlertoleranz (ca. 2 Vorlesungswochen)
  - Bekannte Softwarefehler
  - Definitionen
  - Fehlerarten
  - Fehlerhypothesen
  - Fehlervermeidung
  - Fehlertoleranzmechanismen

Potentielle zusätzliche Inhalte

- Kapitel: Spezielle Hardware (1 Vorlesungswoche)
  - Digital-Analog-Converter (DAC)
  - Analog-Digital-Converter (ADC)
  - Speicherprogrammierbare Steuerung (SPS)
- Kapitel: Regelungstechnik (ca. 2 Vorlesungswochen)
  - Definitionen
  - P-Regler
  - PI-Regler
  - PID-Regler
  - Fuzzy-Logic

Guided Research / Werkstudentenstellen

- Themen:
  - Modellbasierte Entwicklungswerkzeuge
    - Bedeutung von Model-zu-Modelltransformation für die Modellierung und die Codegenerierung
    - Entwicklung einer generischen Modellierungssprache zur Beschreibung von Mechanismen in verteilten Echtzeitsystemen
    - Integration von zeitgesteuerten Kommunikationsprotokollen
  - Kombination von formalen Methoden
    - Verwendung von formalen Methoden zur Berechnung von Ausführungsplänen
    - Nachweis der Korrektheit von (fehlertoleranten) Systemen
  - Entwicklung von Demonstratoren
    - Hardware / Software Co-Design
- Ziel:
  - Enge Anbindung von interessierten Studenten an den Lehrstuhl (direkte Mitarbeit in Forschungsprojekten)
  - Jeder Student bekommt einen „Mentor“ inkl. Arbeitsplatz zur Bearbeitung der Themen
- 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

Definition Eingebettetes System

Technisches System, das durch ein integriertes, von Software gesteuertes Rechensystem gesteuert wird. Das Rechensystem selbst ist meist nicht sichtbar und kann in der Regel nicht frei programmiert werden. Um die Steuerung zu ermöglichen ist zumeist eine Vielzahl von sehr speziellen Schnittstellen notwendig.

In der Regel werden leistungsärmere Mikroprozessoren mit starken Einschränkung in Bezug auf die Rechenleistung und Speicherfähigkeit eingesetzt.

Resultierende Eigenschaften

⇒ zeitliche Anforderungen

- Zeitliche Genauigkeit (nicht zu früh, nicht zu spät)
- Garantierte Antwortzeiten
- Synchronisation von Ereignissen / Daten
- **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, werden in der Regel nicht verwendet:
  - 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

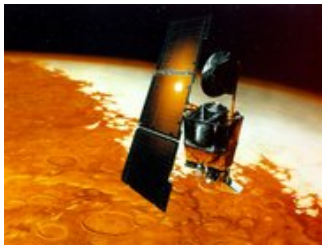
Echtzeitsysteme sind allgegenwärtig!









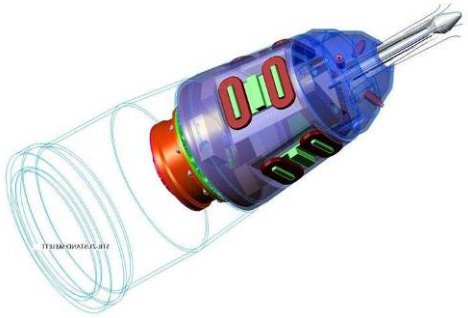


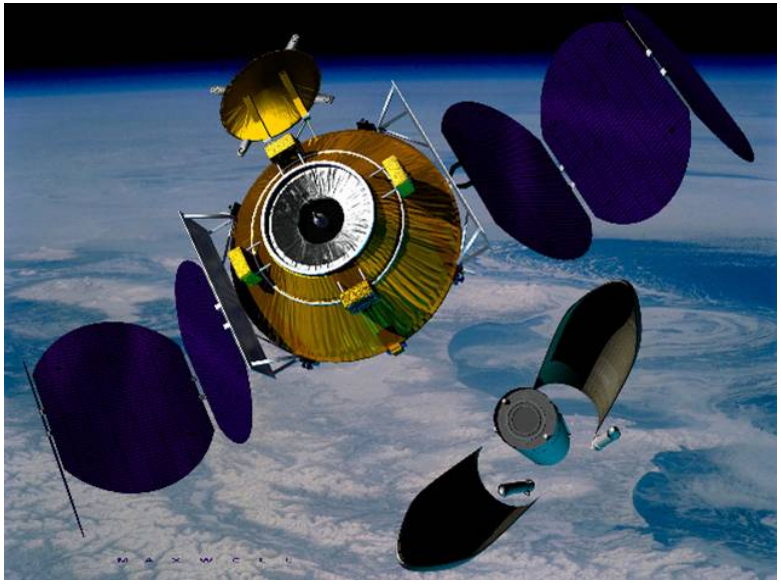
Beispiel: Kuka Robocoaster

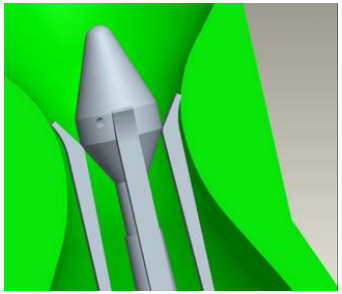


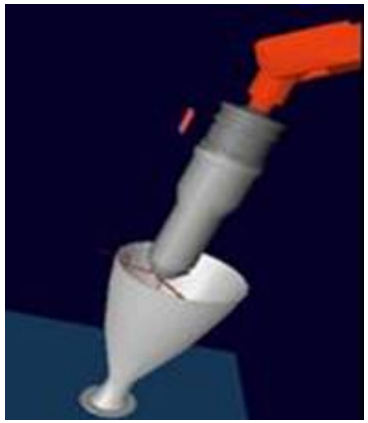

http://www.robocoaster.com

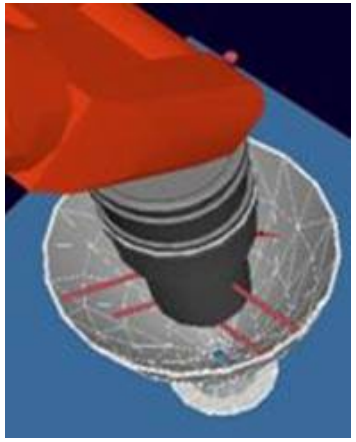
CxOlev - Lebenszeitverlängerung von Satelliten







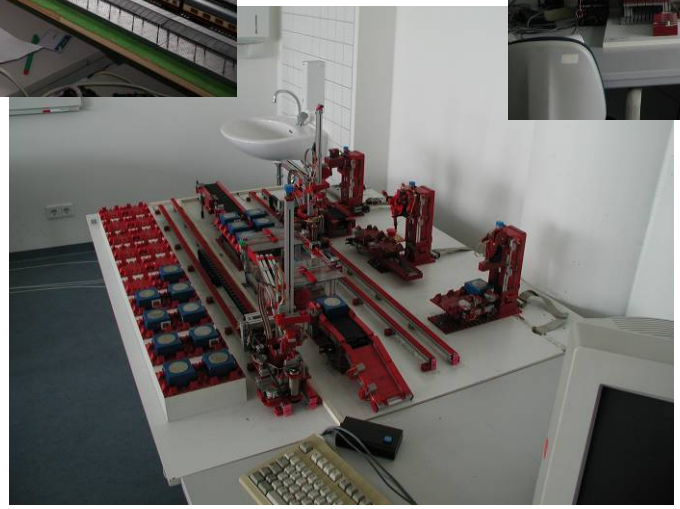




Einleitung Echtzeitsysteme

Anwendungen am Lehrstuhl

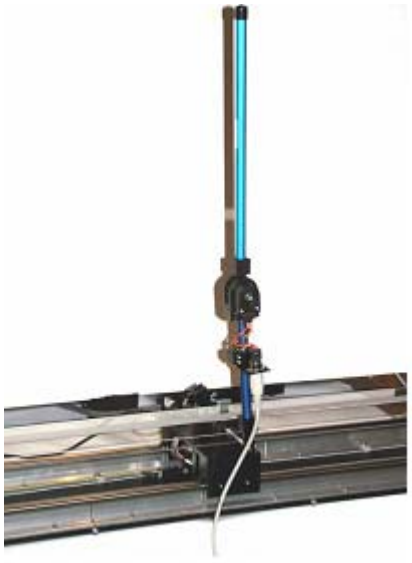
Steuerungsaufgaben (Praktika+Studienarbeiten)



Regelungsaufgaben (Praktika+Studienarbeiten)

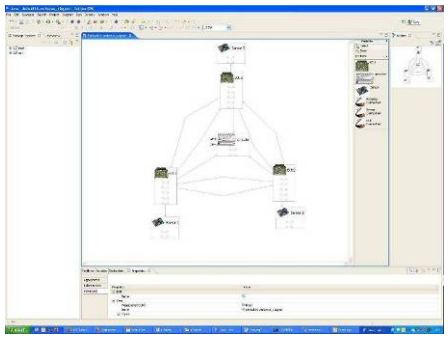
Schwebender Stab

Carrera Rennbahn



Invertiertes Pendel

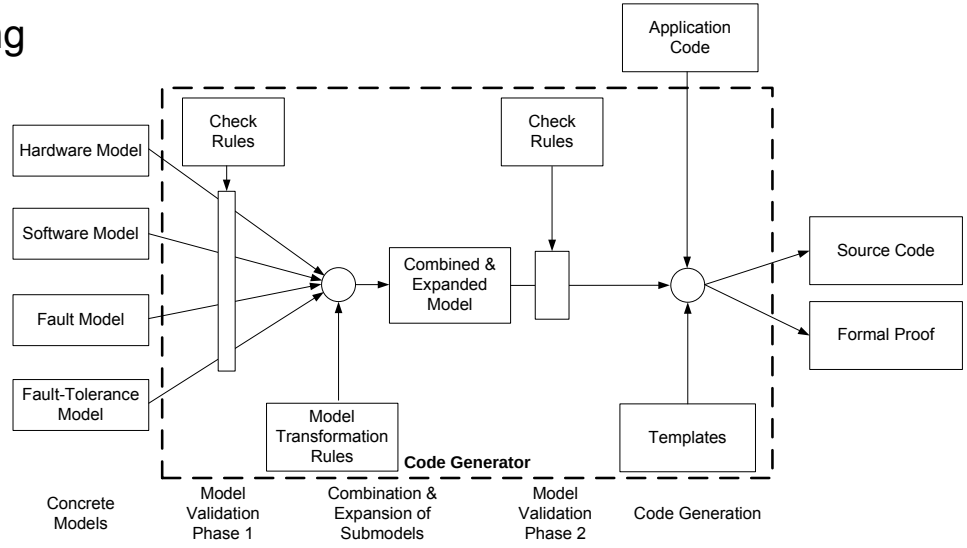
FTOS: Modellbasierte Entwicklung fehlertoleranter Echtzeitsysteme



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

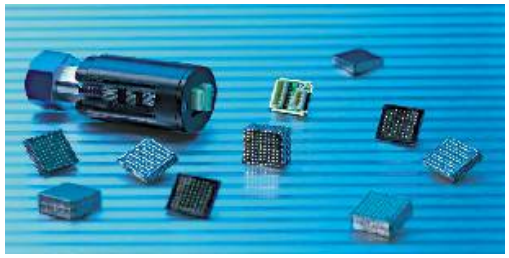
GEFÖRDERT VOM

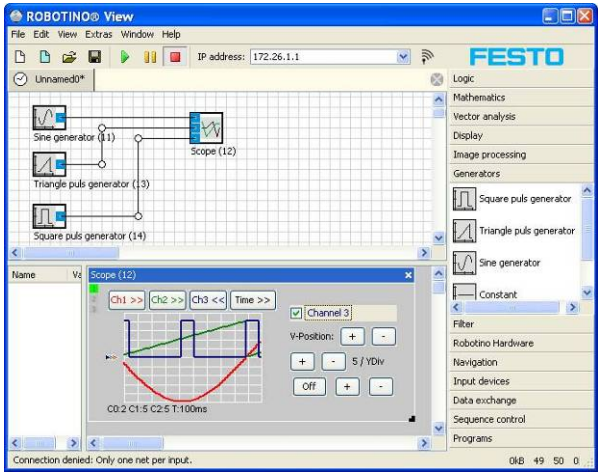
Bundesministerium
für Bildung
und Forschung



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

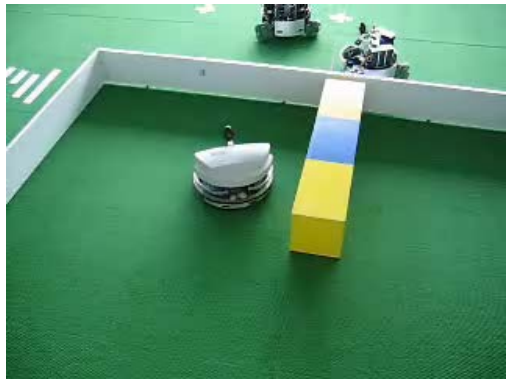


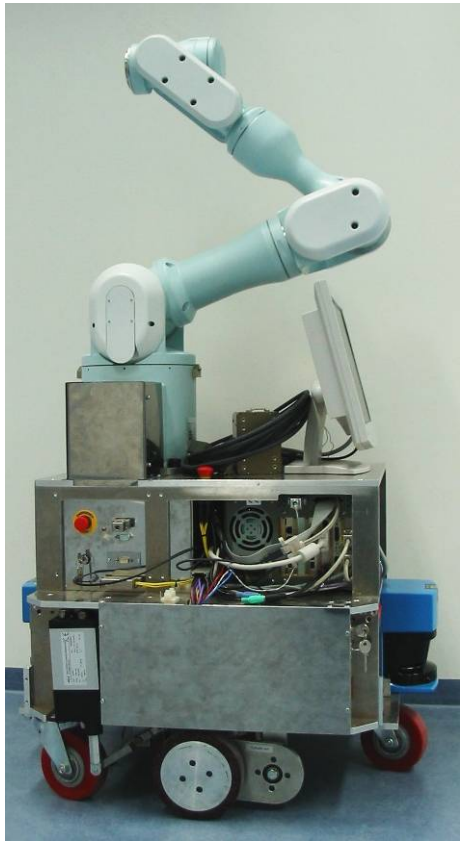


GEFÖRDERT VOM
Bundesministerium für Bildung und Forschung

Robotersteuerung

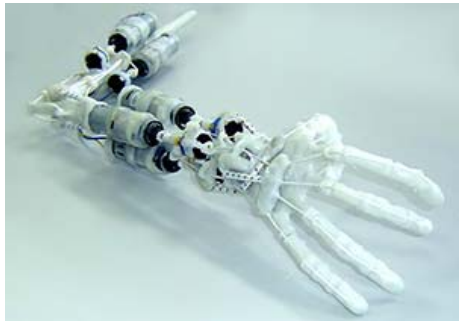
Robotino





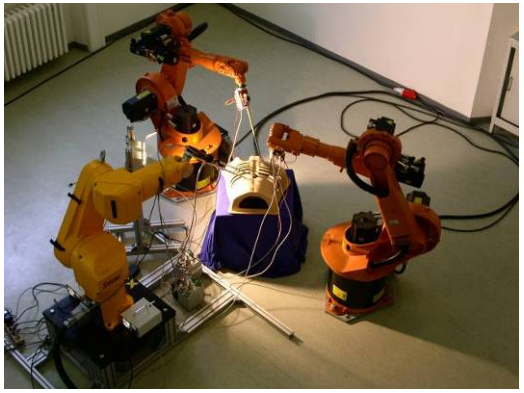
Leonardo

Stäubli



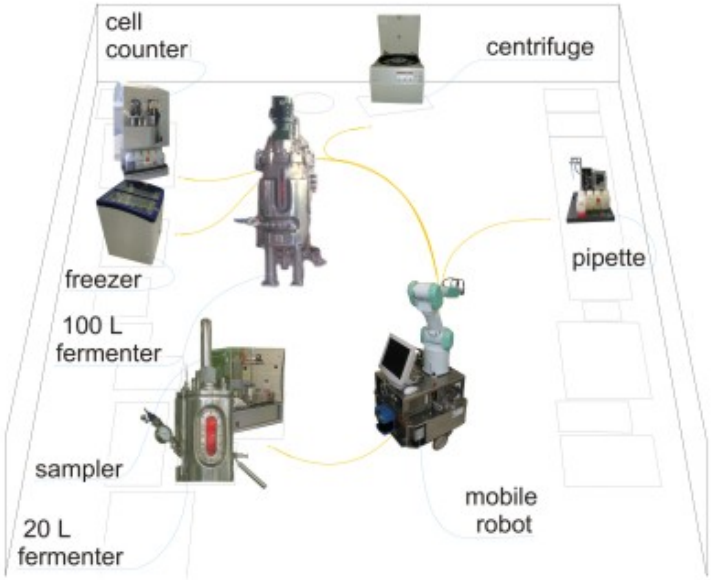
Tumanoid

Anwendungen der Robotik



Telemedizin

Jast



Automatisiertes biotechnisches Labor

Klausurfragen

- Klausur WS 06/07
  - Was ist der Unterschied zwischen harten und weichen Echtzeitsystemen?
  - Wieso sollte Virtual Memory nicht in Echtzeitsystemen verwendet werden?
- Wiederholungsklausur WS 06/07
  - Ordnen Sie folgende Anwendungen in die Kategorien harte bzw. weiche Echtzeitsysteme ein und begründen Sie Ihre Antwort:
    - Ampelsteuerung
    - Flugzeugregelung
    - Internettelefonie

Kapitel 2

Modellierung von Echtzeitsystemen und Werkzeuge

Inhalt

- Motivation
- Grundsätzlicher Aufbau, Models of Computation
  - Werkzeug Ptolemy
- Synchrone Sprachen (Esterel, Lustre)
  - Reaktive Systeme: Werkzeuge Esterel Studio, SCADE
  - Synchroner Datenfluss: EasyLab
- Zeitgesteuerte Systeme
  - Werkzeug Giotto
- Domänenspezifische Codegeneratoren
  - Werkzeug FTOS
- Verifikation durch den Einsatz formaler Methoden

Fokus dieses Kapitels

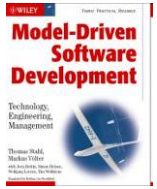
- Konzepte und Werkzeuge zur Modellierung **und** Generierung von Code für Echtzeitsysteme / eingebettete Systeme
- Voraussetzungen an Werkzeuge für ganzheitlichen Ansatz:
  - Explizite Modellierung des zeitlichen Verhaltens (z.B. Fristen)
  - Modellierung von Hardware und Software
  - Eindeutige Semantik der Modelle
  - Berücksichtigung von nicht-funktionalen Aspekten (z.B. Zeit*, Zuverlässigkeit, Sicherheit)
- Ansatz zur Realisierung:
  - Schaffung von domänenspezifischen Werkzeugen (Matlab/Simulink, Labview, SCADE werden überwiegend von spezifischen Entwicklergruppen benutzt)
  - Einfache Erweiterbarkeit der Codegeneratoren oder Verwendung von virtuellen Maschinen / Middleware-Ansätzen

---

* Zeit wird zumeist als nicht-funktionale Eigenschaft betrachet, in Echtzeitsystemen ist Zeit jedoch als funktionale Eigenschaft anzusehen (siehe z.B. Edward Lee: Time is a Resource, and Other Stories, May 2008)

Literatur

- Sastry et al:Scanning the issue - special issue on modeling and design of embedded software, Proceedings of the IEEE, vol.91, no.1, pp. 3-10, Jan 2003
- Thomas Stahl, Markus Völter: Model-Driven Software Development, Wiley, 2006



- Ptolemy: Software und Dokumentation http://ptolemy.eecs.berkeley.edu/ptolemyII/index.htm
- Benveniste et al.: The Synchronous Languages 12 Years Later, Proceedings of the IEEE, vol.91, no.1, pp. 64-83, Jan 2003
- 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
- Henzinger et al.: Giotto: A time-triggered language für embedded programming, Proceedings of the IEEE, vol.91, no.1, pp. 84-99, Jan 2003

Hinweis: Veröffentlichungen von IEEE, Springer, ACM können Sie kostenfrei herunterladen, wenn Sie den Proxy der TUM Informatik benutzen (proxy.in.tum.de)

Begriff: Modell

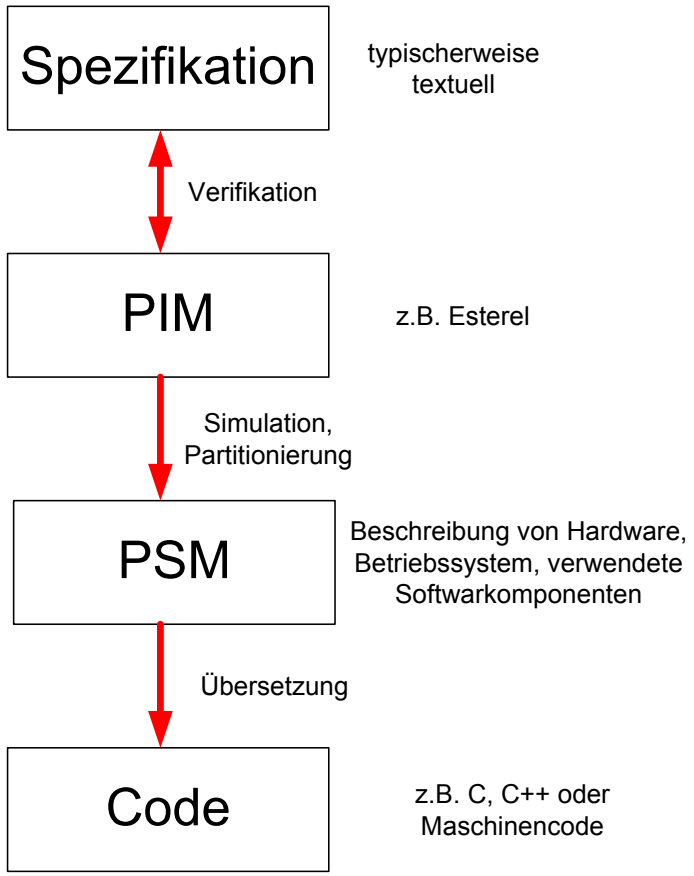
- Brockhaus:
  Ein Abbild der Natur unter der Hervorhebung für wesentlich erachteter Eigenschaften und Außerachtlassen als nebensächlich angesehener Aspekte. Ein M. in diesem Sinne ist ein Mittel zur Beschreibung der erfahrenen Realität, zur Bildung von Begriffen der Wirklichkeit und Grundlage von Voraussagen 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
  - Systeme können vorab simuliert werden. Hierdurch können Designentscheidungen vorab evaluiert werden und späte Systemänderungen minimiert werden.
  - 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

OMG: Model-Driven Architecture (MDA)

- 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)



MDA im Kontext von Echtzeitsystemen

- In Echtzeitsystemen / eingebetteten Systemen ist bei einem umfassenden Ansatz ein Hardwaremodell (z.B. Rechner im verteilten Systen, Topologie) schon in frühen Phasen (PIM) notwendig
- Das plattformspezifische Modell (PSM) erweitert das Hardware- & Softwaremodell um Implementierungskonzepte, z.B.
  - Implementierung als Funktion/Thread/Prozess
  - Prozesssynchronisation

Anwendungsbeispiele

- 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
- Beispiel 2: Ampel

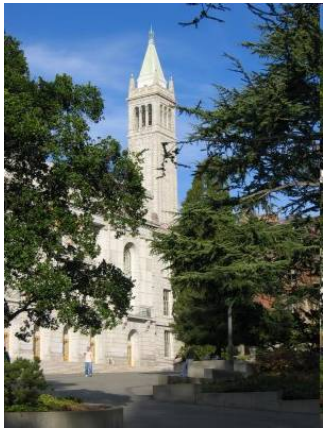
Modellierung von Echtzeitsystemen

Aktoren, Ausführungsmodelle

Werkzeuge: Ptolemy

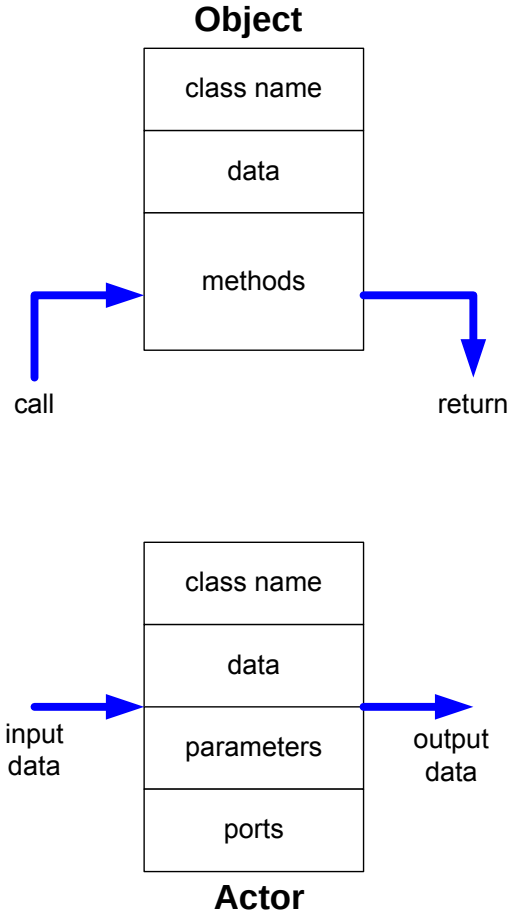
Ptolemy

- Das Ptolemy-Projekt an der UC Berkeley untersucht verschiedene Modellierungs-methodiken für eingebettete Systeme mit einem Fokus auf verschiedene Ausführungs-modelle (Models of Computation)
- Ptolemy unterstützt
  - Modellierung
  - Simulation
  - Codegenerierung
  - Formale Verifikation (teilweise)
- Weitere Informationen unter: http://ptolemy.eecs.berkeley.edu/



Aktororientiertes Design

- Ptolemy-Modelle basieren auf Aktoren anstelle von Objekten
- Objekte:
  - Fokus liegt auf Kontrollfluss
  - Objekte werden manipuliert
- Aktoren
  - Fokus liegt auf Datenfluss
  - Aktoren manipulieren das System
- Vorteil beider Ansätze: erhöhte Wiederverwendbarkeit
- Vorteil von Aktoren: leichtere Darstellung von Parallelität



Models of Computation

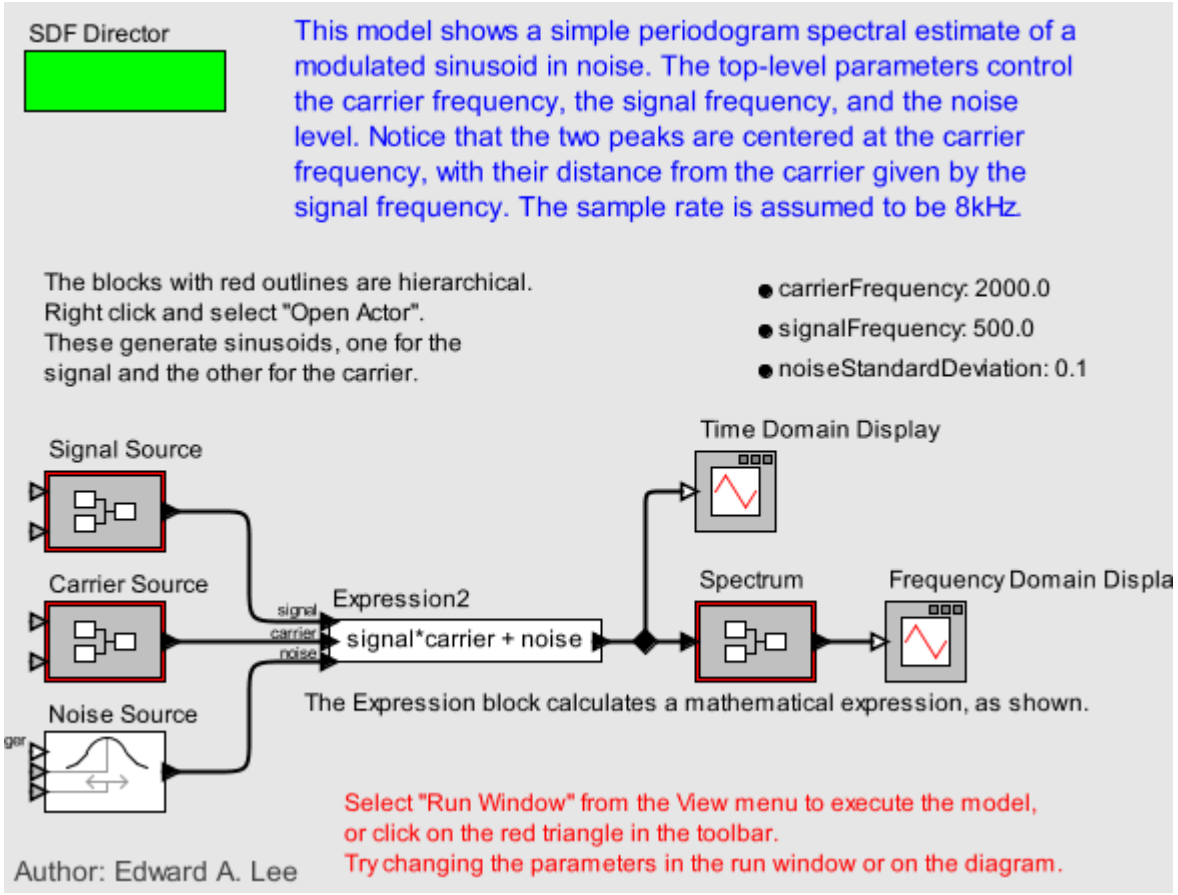
- Ausführungsmodelle (models of computation) bestimmen die Interaktion von Komponenten/Aktoren.
- Die Eignung eines Ausführungsmodells hängt von der Anwendungsdomäne, aber auch der verwendeten Hardware, ab.
- In Ptolemy wird durch die Einführung von Direktoren (director) die funktionale Ausführung (Verschaltung der Aktoren) von der zeitlichen Ausführung (Abbildung im Direktor) getrennt.
- Durch Domänenpolymorphismus (domain polymorphism) können Aktoren unter verschiedenen Ausführungsmodellen verwendet werden.
- Verschiedene Ausführungsmodelle können hierarchisch geschachtelt werden (modal models).
  - Typisches Beispiel: Synchroner Datenfluss und Zustandsautomaten

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

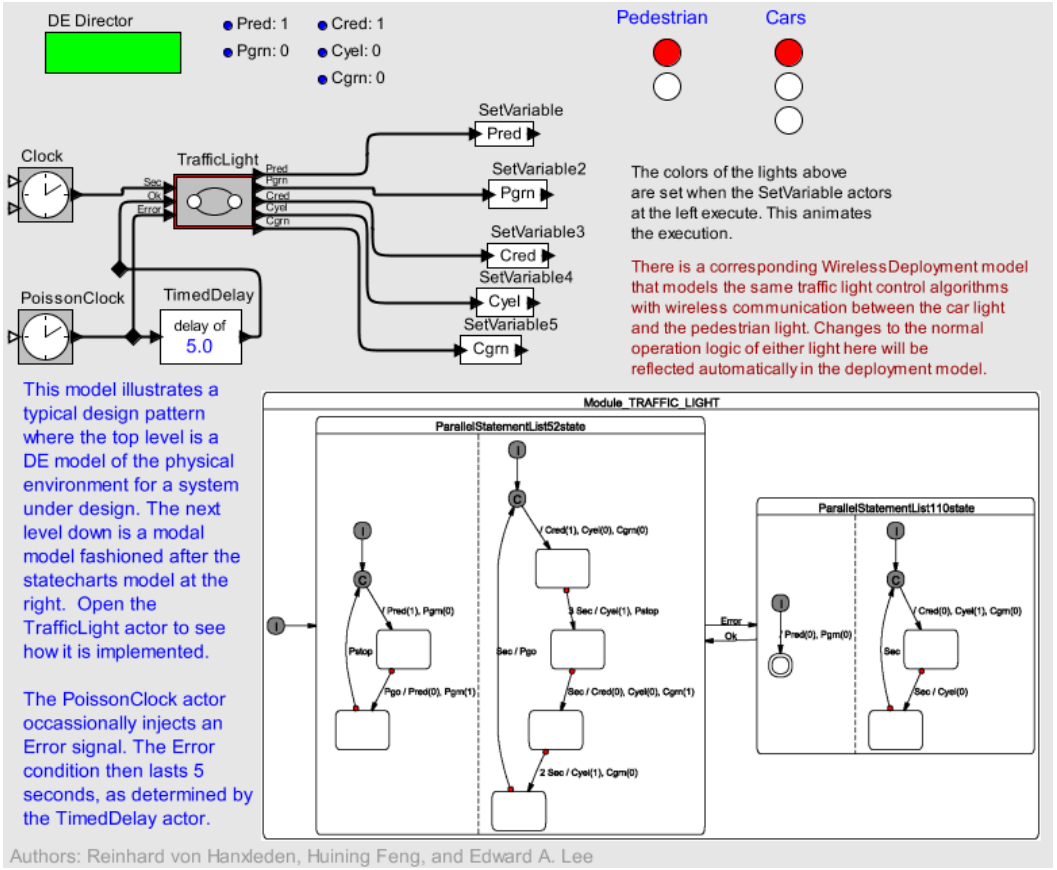
MoC: Synchronuous Dataflow

- Prinzip:
  - Annahme: unendlich schnelle Maschine
  - Daten werden zyklisch verarbeitet
  - Pro Runde wird genau einmal der Datenfluss ausgeführt
- Vorteile:
  - Statische Speicherallokation
  - Statischer Schedule berechenbar
  - Verklemmungen detektierbar
  - Laufzeit kann bestimmt werden
- Werkzeuge:
  - Matlab
  - Labview



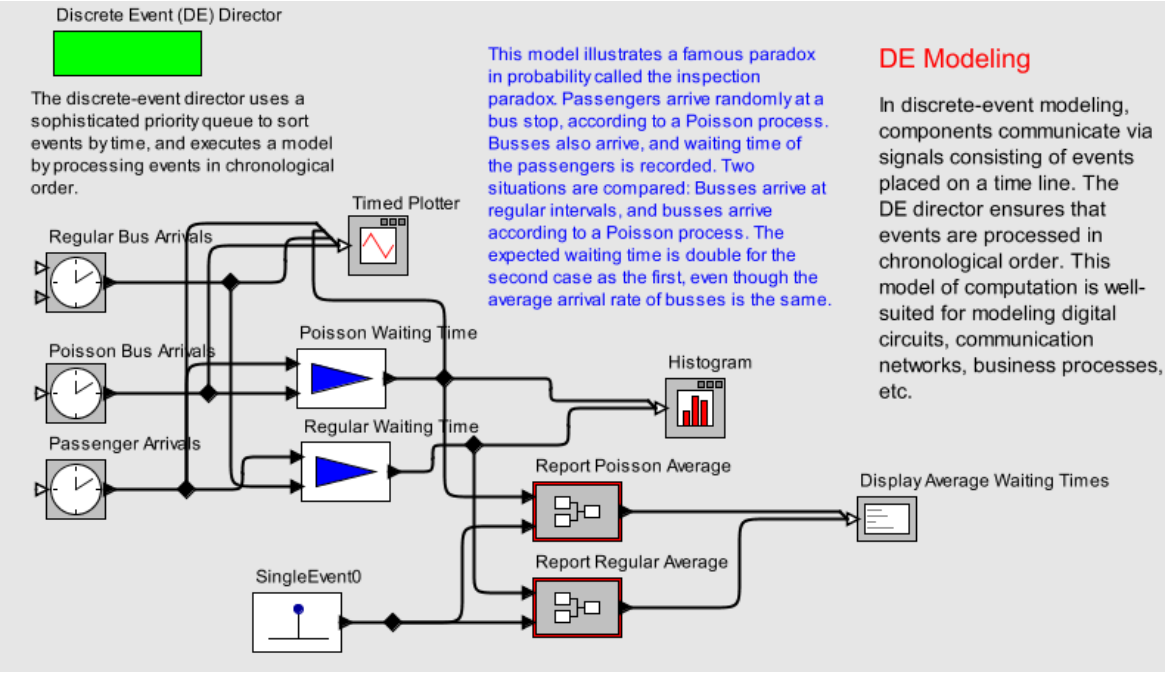
MoC: Synchronous Reactive

- Prinzip:
  - Annahme: unendlich schnelle Maschine
  - Ereignisse werden zyklisch verarbeitet (Ereignisse müssen nicht jede Runde eintreffen)
  - Pro Runde wird genau eine Reaktion berechnet
  - Häufig verwendet in Zusammenhang mit Finite State Machines
- Vorteile:
  - einfache formale Verifikation
- Werkzeuge:
  - Esterel Studio
  - Scade



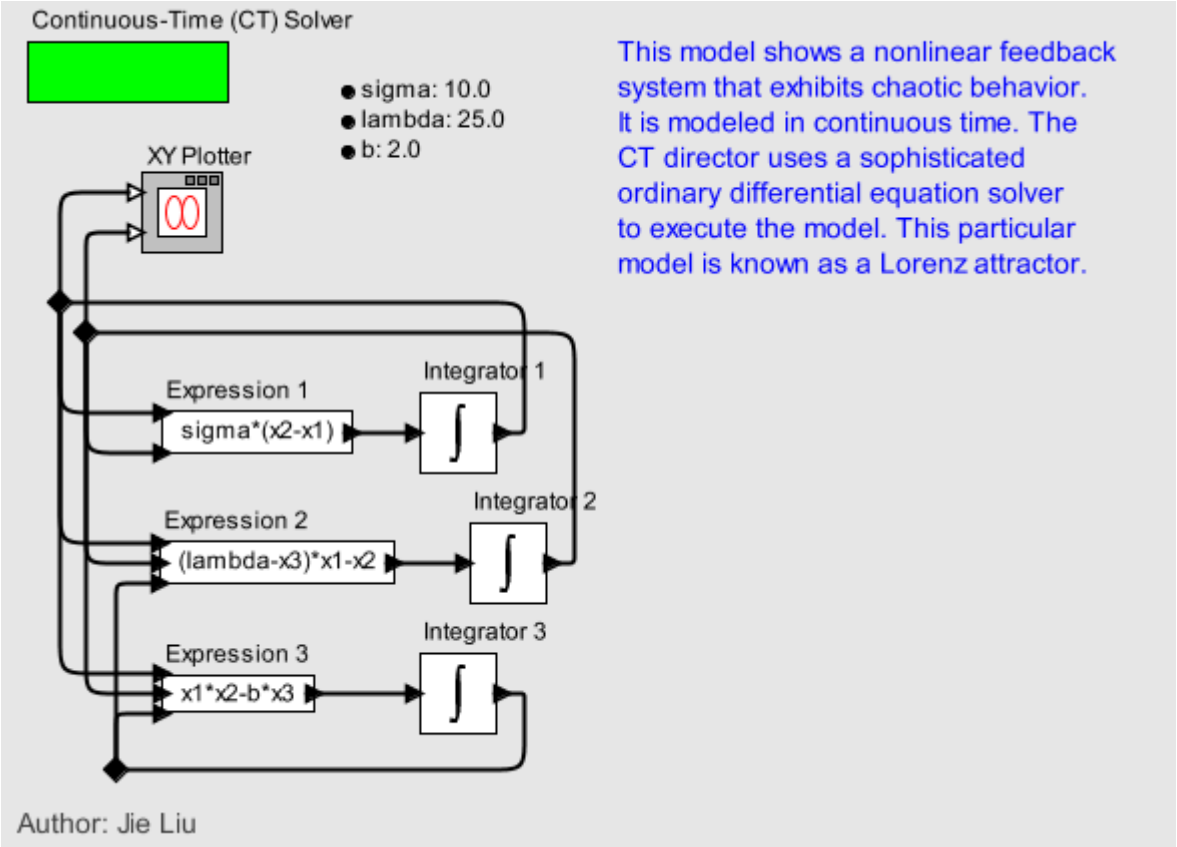
MoC: Discrete Event

- Prinzip:
  - Kommunikation über Ereignisse
  - Jedes Ereignis trägt einen Wert und einen Zeitstempel
- Anwendungsgebiet:
  - Digitale Hardware
  - Telekommunikation
- Werkzeuge:
  - VHDL
  - Verilog
- Varianten:
  - Distributed Discrete Events



MoC: Continuous Time

- Prinzip:
  - Verwendung kontinuierlicher Signale (zumeist Differential-gleichungen)
- Anwendungsgebiet:
  - Simulation
- Werkzeuge:
  - Simulink
  - Labview



Weitere MoCs

- Component Interaction:
  - Mischung von daten- und anfragegetriebener Ausführung
  - Beispiel: Web Server
- Discrete Time:
  - Erweiterung des synchronen Datenflussmodels um Zeit zwischen Ausführungen zur Unterstützung von Multiratensystemen
- Time-Triggered Execution
  - Die Ausführung wird zeitlich geplant
  - Anwendungsgebiet: kritische Regelungssysteme
  - Werkzeug: Giotto, FTOS
- Process Networks
  - Prozess senden zur Kommunikation Nachrichten über Kanäle
  - Kanäle können Nachrichten speichern

  $\Rightarrow$ asynchrone Nachrichten
  - Anwendungsgebiet: verteilte Systeme
- Rendezvous
  - synchrone Kommunikation verteilter Prozesse (Prozesse warten am Kommunikationspunkt bis Sender und Empfänger bereit sind)
  - Beispiele: CSP, CCS, Ada

Modellierung von Echtzeitsystemen

Reaktive Systeme

Werkzeuge: SCADE, Esterel Studio

Fragen zur letzten Vorlesung

- Was sind Aktoren?
  - Unterscheidung zum Begriff Aktoren aus der Mechatronik?
- Wie zuverlässig ist der generierte Code bei modellbasierten Entwicklungswerkzeugen?
- Wie gut ist der generierte Code zu lesen?

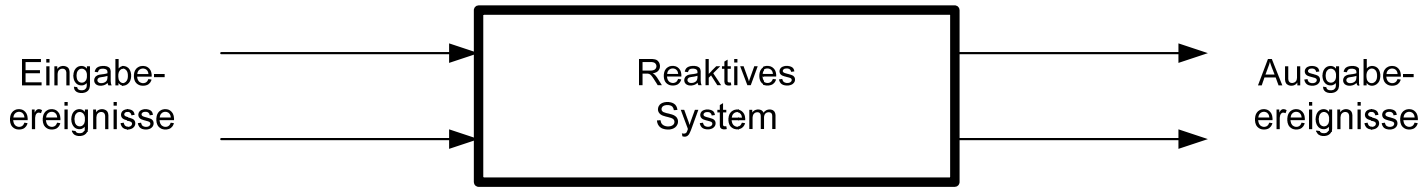
- Frage zu dieser Vorlesung:
  - Wann verwendet man synchrone Sprachen, wann synchronen Datenfluss?
    - Unterscheidung zwischen kontrollorientierten (Fragestellung: wie reagiert das System auf Ereignisse) und datenorientierten (Fragestellung: wie werden eingehende Daten verarbeitet) Anwendungen.

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.

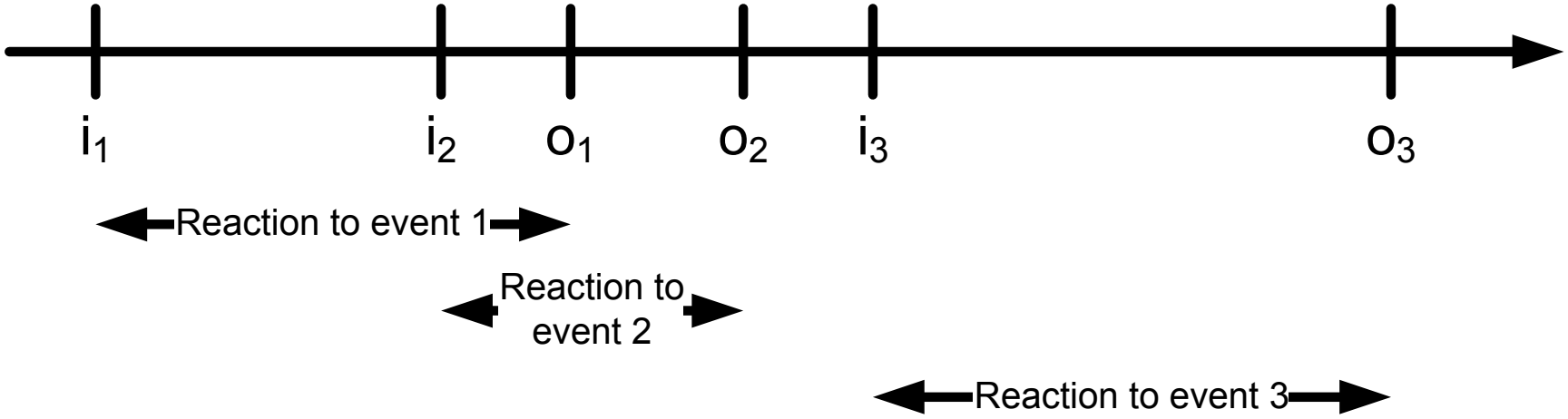
Einführung in Esterel

- 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).



Allgemein: Reaktive Systeme

- Bearbeitung der Ereignisse kann sich überlappen



Synchrony hypothesis

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

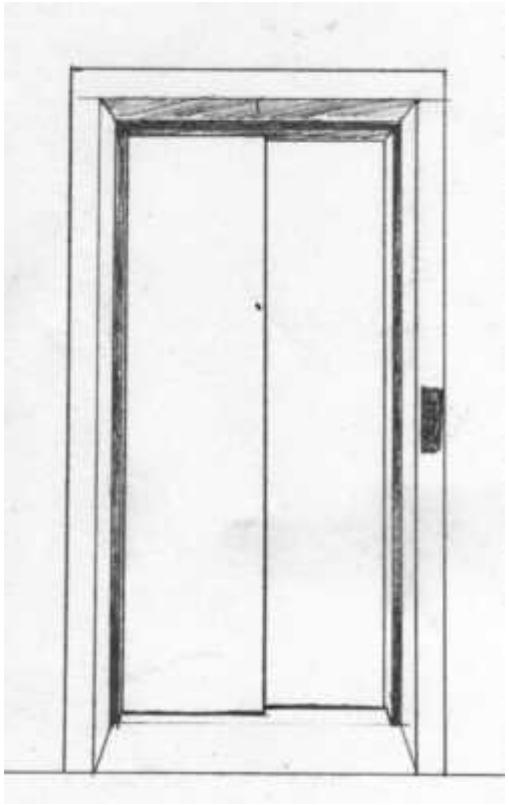
  → Die Reaktion des Systems auf ein Eingabeereignis erfolgt augenblicklich. Reaktionsintervale reduzieren sich zu Reaktionsmomenten (**reaction instants**).
- **Rechtfertigung**: Diese Annahme ist korrekt, wenn die Wahrscheinlichkeit des Eintreffens eines zweiten Ereignisses, während der initialen Reaktion 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.
- Durch den Determinismus wird die Verifikation von Anwendungen wesentlich vereinfacht, allerdings birgt es auch die Gefahr, dass Ereignisse „vergessen“ werden, falls sie zeitgleich mit wichtigeren Ereignissen eintreffen.

Esterel an einem Beispiel: Aufzugstür

- Aufgabe: Öffnen und Schließen der Aufzugstür
- Sicherheitsfunktion:
  - Tür darf während der Fahrt nicht geöffnet werden
- 1. Schritt: Definition der Module (parallelen Abläufe)
- 2. Schritt: Definition der Eingangssignale
- 3. Schritt: Definition der Ausgangssignale
- 4. Schritt: Definition der Zustände (inkl. Anfangszustand):
- 5. Schritt: Definition der Zustandsübergänge



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
```

Parallelismus

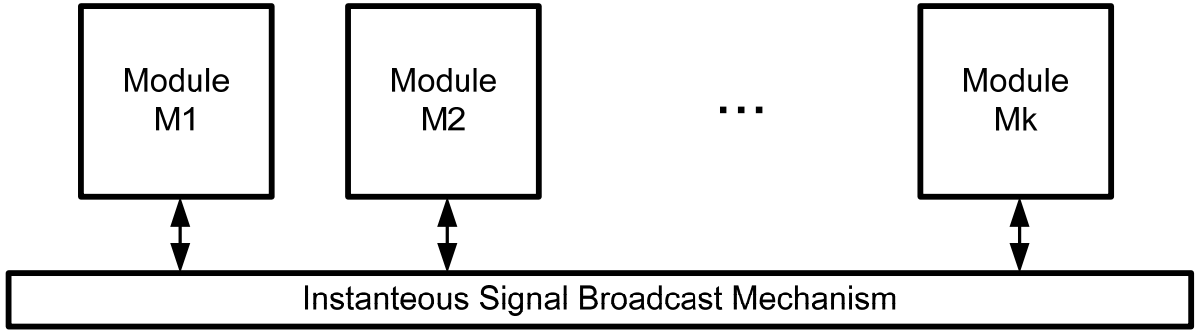
- Zur parallelen Komposition stellt Esterel den Operator || zur Verfügung. Sind P1 und P2 zwei Esterel Programme, so ist auch P1||P2 ein Esterel Programm mit folgenden Eigenschaften:
  - Alle Eingabeereignisse stehen sowohl P1 als auch P2 zur Verfügung.
  - Jede Ausgabe von P1 (oder P2) ist im gleichen 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.

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.



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.

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.

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.
- Auf den folgenden Folien werden die wichtigsten Konstrukte erläutert.

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.

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.

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.

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;
```

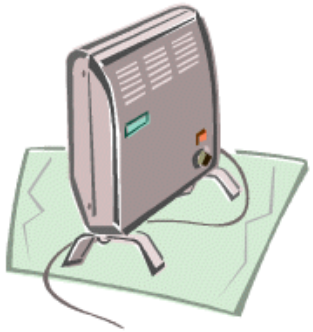
if Anweisung in Bezug auf Signale

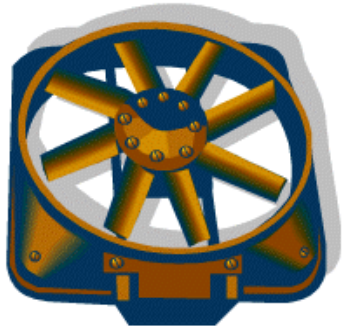
- Durch Verwendung der if- Anweisung kann auch die Existenz eines Signals geprüft werden.
- **Syntax**:

```
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: 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
```

Modellierung von Echtzeitsystemen

Reaktive Systeme

Werkzeuge: SCADE, Esterel Studio

Fragen zur letzten Vorlesung

- Wieso wird Rekursion in Esterel nicht unterstützt?
- Wie funktioniert die Umsetzung des Await-Statements?

```
module Temperature:
input IN1,IN2;
output OUT1,OUT2;
    loop
      await IN1; emit OUT1;
    end loop;
    ||
    loop
      await IN1; emit OUT1;
    end loop;
```

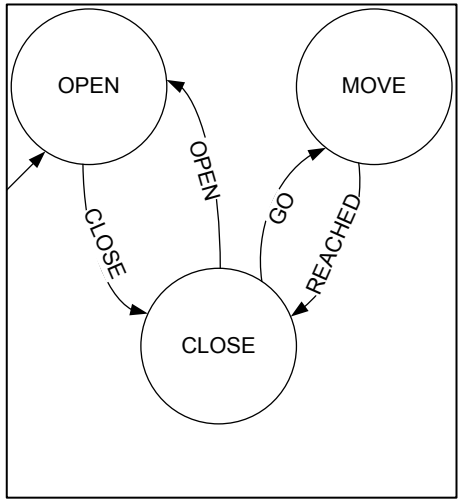
Exkurs: Automaten zur Modellierung von reaktiven Systemen

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

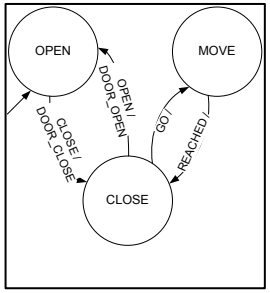
- Ein klassischer endlicher Automat $(Q, \Sigma, \delta, s_0, F)$ ist eine endliche Menge von Zuständen und Zustandsübergängen mit:
  - endlicher Menge von Zuständen Q
  - endliches Eingabealphabet Σ
  - Übergangsfunktion $\delta : Q \times \Sigma \rightarrow Q$
  - Menge von Endzuständen $F \subseteq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



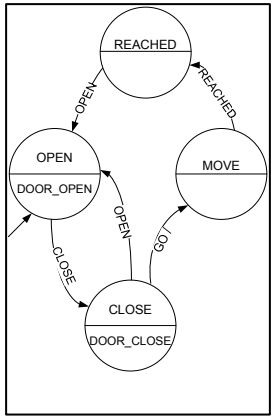
- Um die Verständlichkeit zu verbessern, werden nur deterministische Automaten modelliert, also $|S|=1$ und $\delta(q,\alpha)=q' \wedge \delta(q,\alpha)=q'' \Rightarrow p'=q''$
- Problem bei der klassischen Definition von Automaten: Ausgaben können nicht modelliert werden.

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.



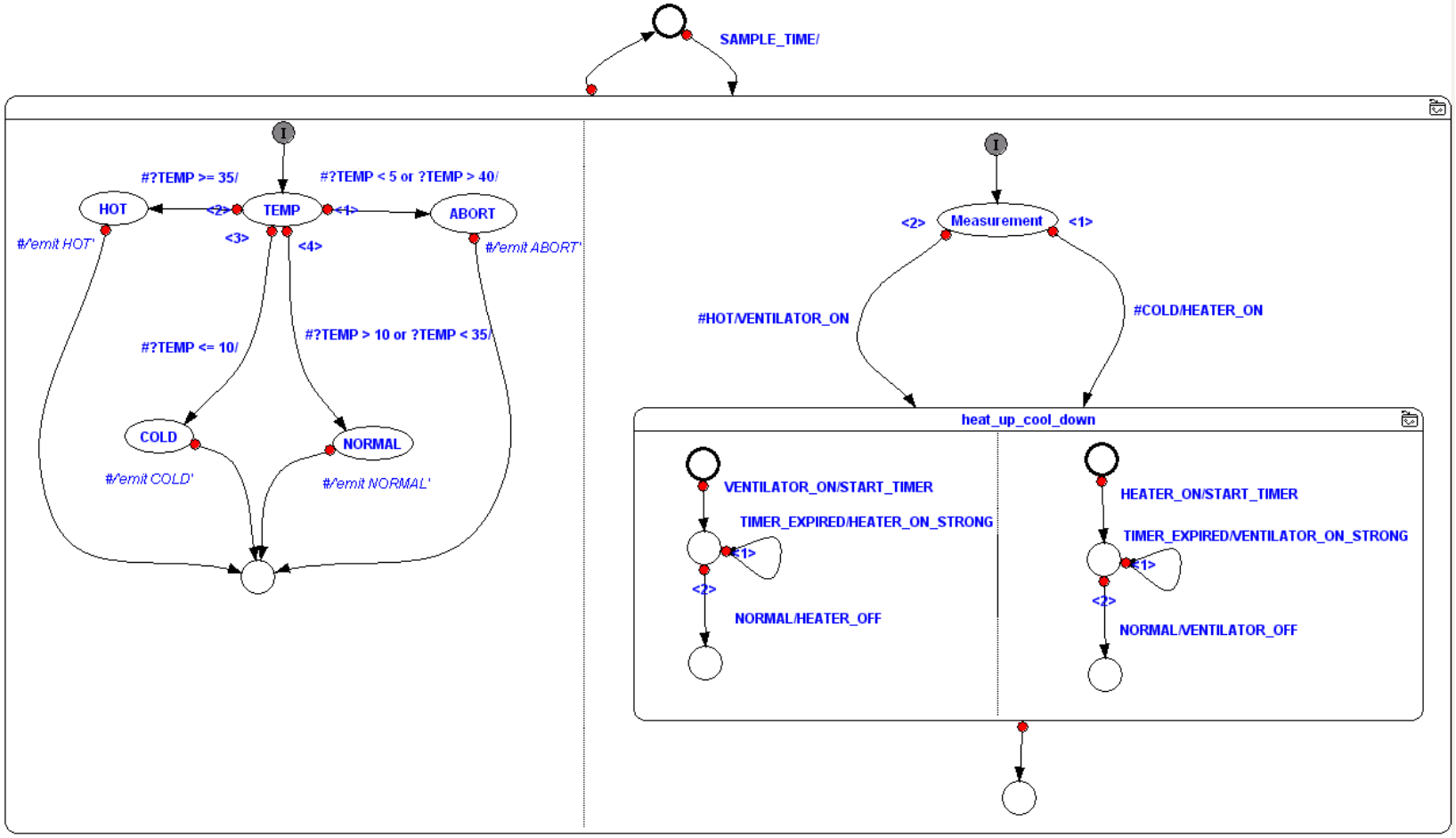
Harel-Automaten / Statecharts

- 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

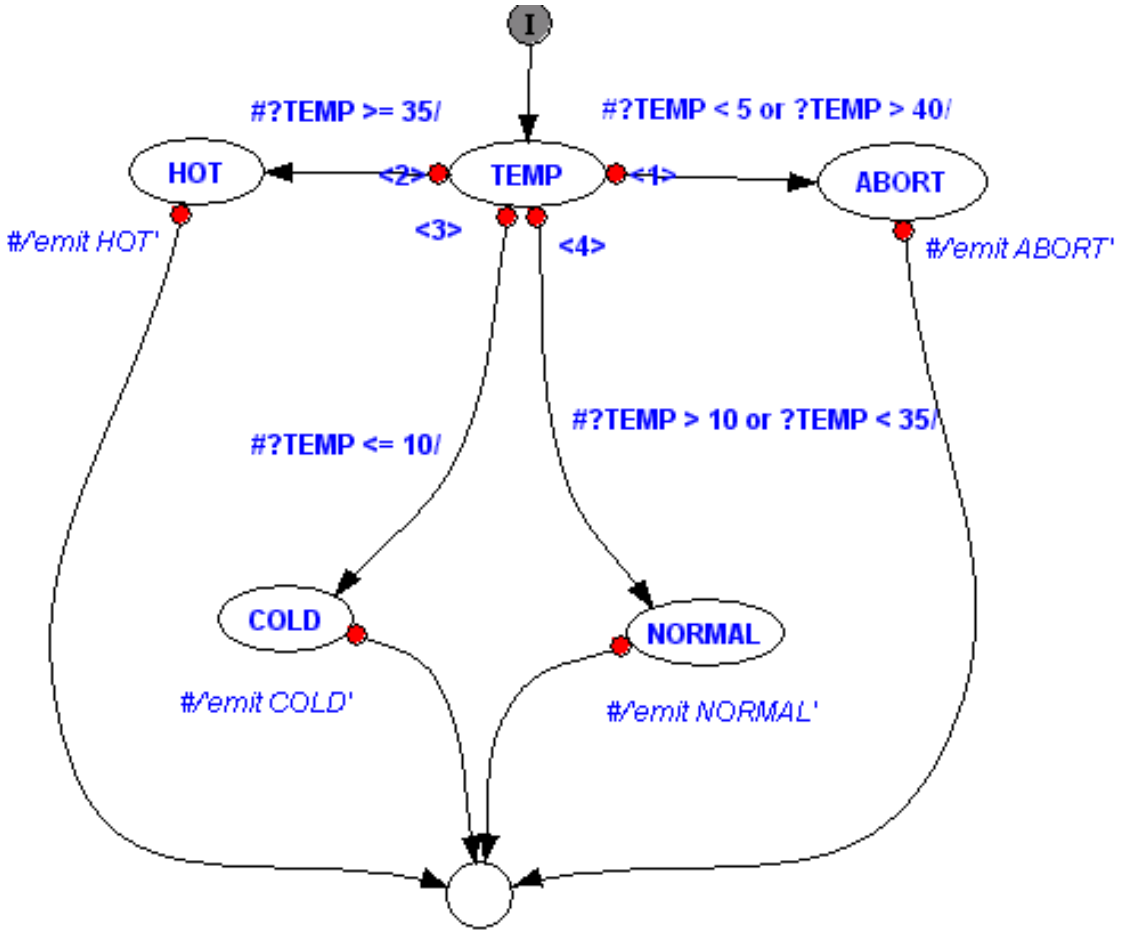
Safe State Machine

- 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)

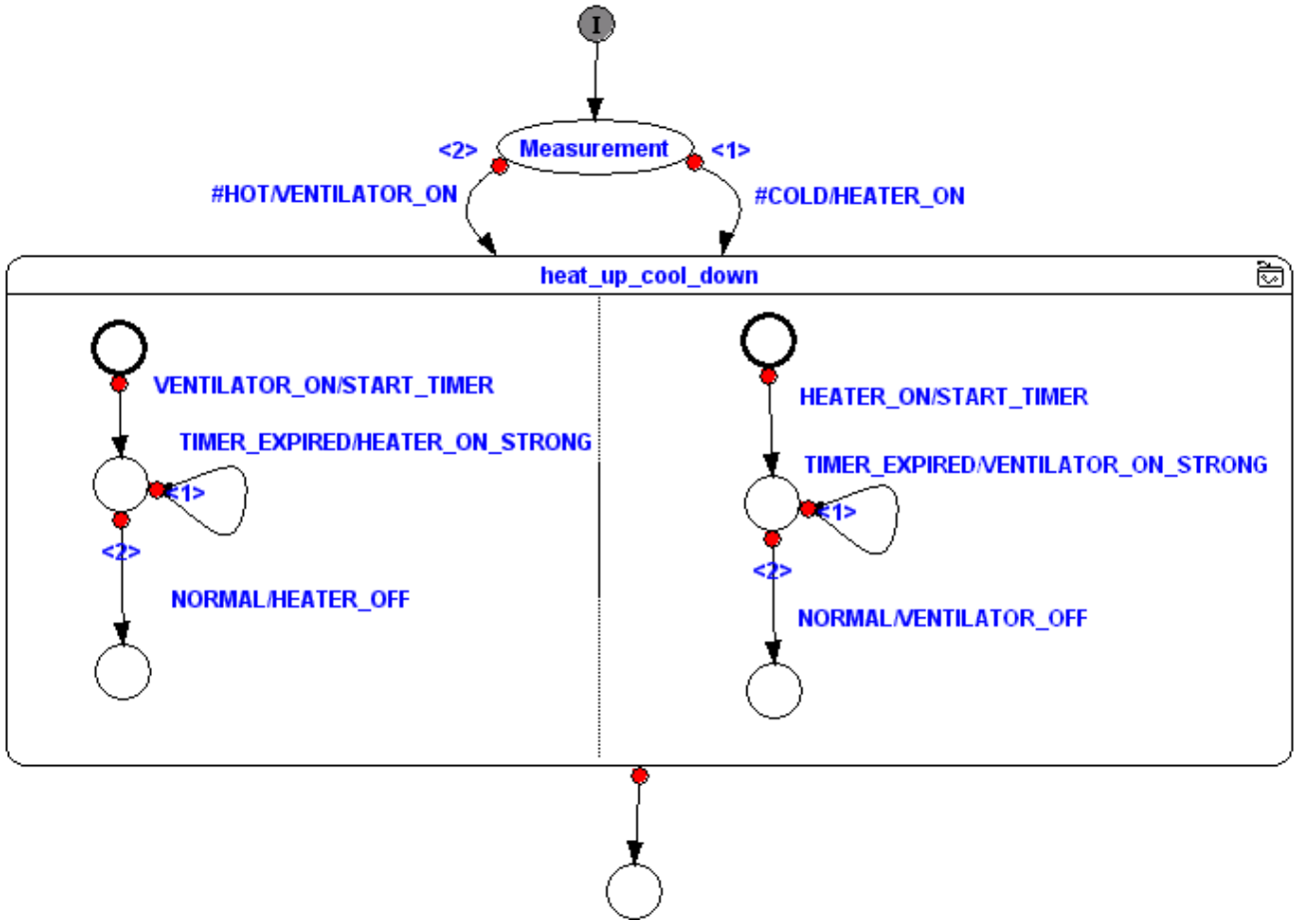
Beispiel als Automat



Beispiel als Automat – Teil 1



Beispiel als Automat – Teil 2



Modellierung von Echtzeitsystemen

Reaktive Systeme - Klausurfragen

Werkzeuge: SCADE, Esterel Studio

Auszug aus Klausur WS 06/07

a) Vervollständigen Sie folgende Testfälle, so dass das Modul xxx diese Testfälle erfüllt:

1. T1=({D},___),(___,{F})
2. T2=({D},___),({D},___})
3. T3=(___,{E}),(___,{F}),(___,{})
4. T4=(_____),(___,{N})
5. T5=(_____),(___,{E}),(___,{E})

Zur Erinnerung: ({A},{B}),({C},{D}) bedeutet: im ersten Moment erfolgt das Ereignis A als Eingabe, die Reaktion des Moduls ist B, im zweiten Moment erfolgt das Ereignis C als Eingabe mit der Reaktion D.

```
module xxx:
    input U, D;
    output E,F,N;
    var V=1;
    loop
        await
          case U do
                    if(?V>0)
                              V:=V+1;
                    else
                              V:=V+1;
                              emit F;
          case D do
                    if(?V>0) then
                              V:=V-1;
                              emit E;
                    else
                              emit N;
        end await;
    end loop;
```

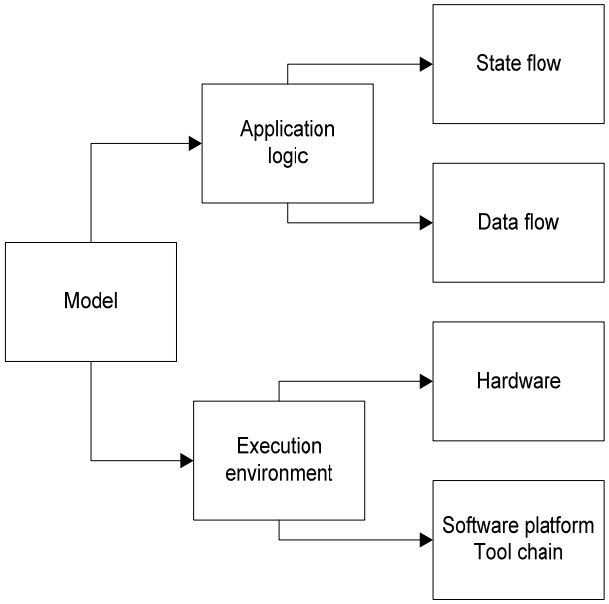
Modellierung von Echtzeitsystemen

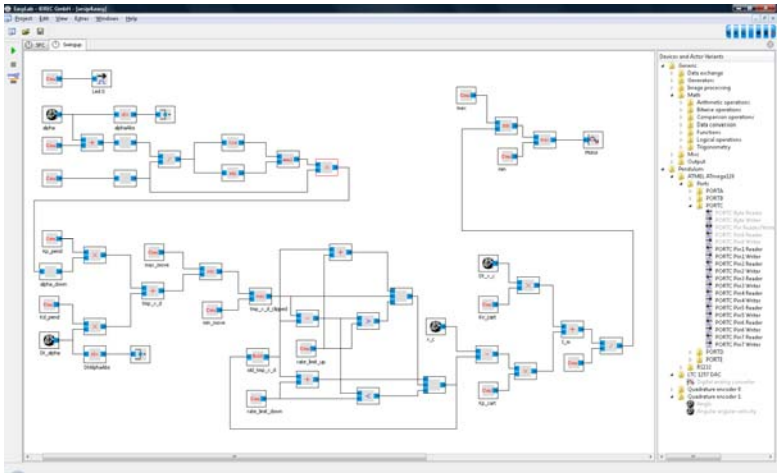
Synchroner Datenfluss

Werkzeug: EasyLab

EasyLab

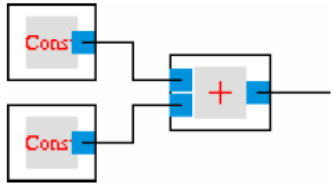
- Unterstützung verschiedener Sichten auf das System:
  - Anwendungslogik
    - Kontrollfluss und Datenfluss
  - Ausführungsumgebung
    - Hardware: Geräte kapseln den Zugriff auf Hardware-komponenten und bieten eine abstrakte Programmierschnittstelle
    - Die passende Werkzeugkette kann gewählt und angestoßen werden (Compilierung, Download)
- Inhalt der Vorlesung: Datenfluss





Synchroner Datenfluss

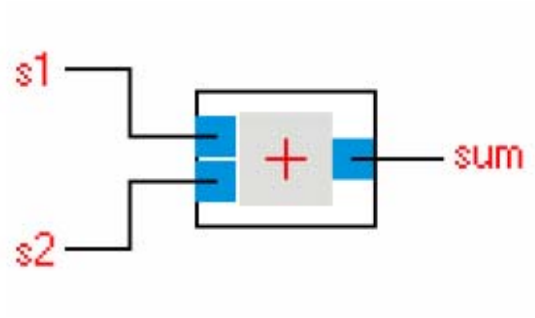
- Geeignet für datenzentrische Anwendungen
- Aktoren sind (z.T. typisierte) Funktionsblöcke,
  - die die einfließenden Daten verarbeiten/konvertieren und
  - das Ergebnis an Ausgängen zur Verfügung stellen

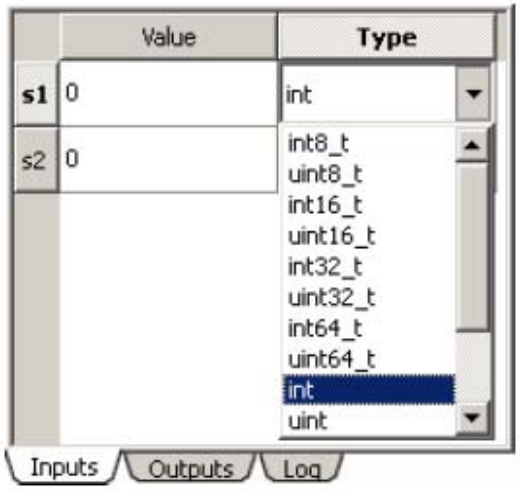


- Ausführungssemantik:
  - ein synchroner Datenfluss wird zyklisch gestartet, die Daten durchlaufen den kompletten Graphen
  - Parallelität ist direkt im Ausführungsmodell sichtbar
  - Zyklen sind erlaubt, allerdings müssen die Daten gepuffert werden

Überladen von Aktoren

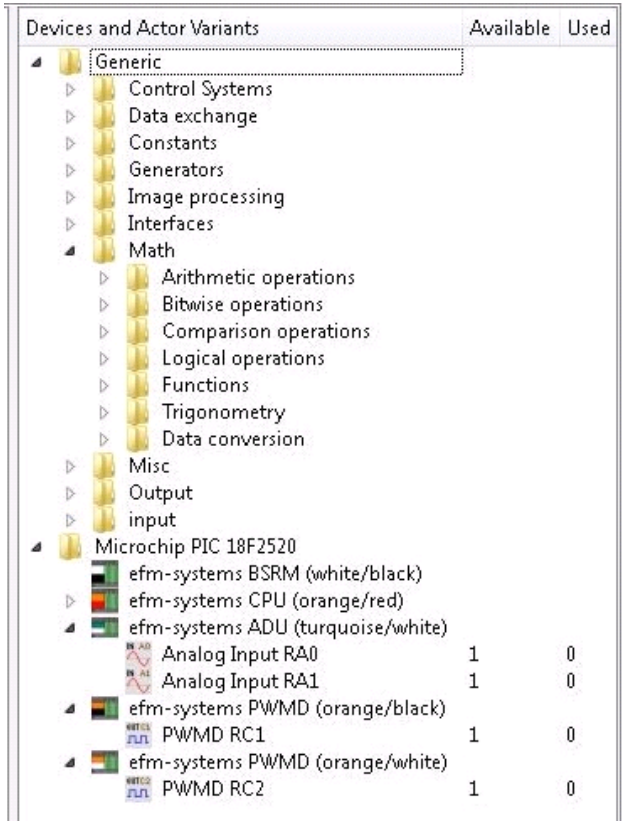
- Aktoren können typunabhängig angeboten werden
- Der Typ eines Aktors ist solange frei wählbar, bis sich durch Anschlussbelegung der Typ automatisch ergibt





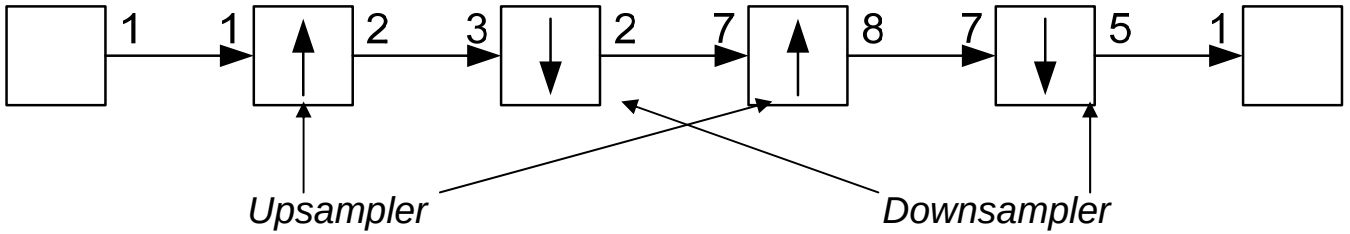
Geräte

- Neben Softwareaktoren sind in Echtzeitsystemen auch Aktoren zur Ansteuerung der Hardware interessant
- Lösung in EasyLab:
  - Geräte / Devices bieten Hardwareaktoren an
  - Hardwareaktoren bieten entweder Eingänge oder Ausgänge
  - Hardwareaktoren benötigen Ressourcen ⇒ die Verwendung der Aktoren ist in Ihrer Anzahl beschränkt



Multiratensysteme

- Häufig erfolgen Empfang und Verarbeitung von Daten in unterschiedlichen Raten:
  - Beispiel: Empfang über serielle Schnittstelle (bitweiser Empfang) und byteweise Verarbeitung
- Im synchronen Datenfluss können über Puffer Daten gesammelt und schließlich verarbeitet werden. Die Frequenzen der Aktoren werden bei den entsprechenden Eingängen angegeben:
  Beispiel: Konvertierung von 44,1 Hz (DAT) nach 48 Hz (CD)



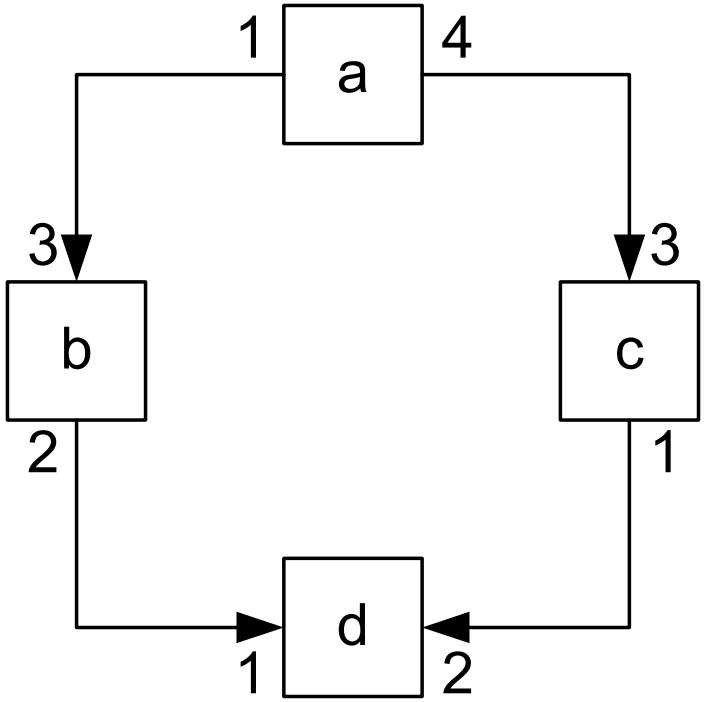
- Es kann notwendig sein, dass einige Felder bereits vorbelegt werden. Dies geschieht durch eine initiale Belegung der Kanten.

Ausführungssemantik von Multiratensystemen

- Ein Knoten kann erst schalten, wenn für alle eingehenden Kanten genügend Marken (Einzeldaten) vorliegen
- Vorteil von synchronen Datenfluss im Allgemeinen: der Ausführungsplan kann vorab berechnet werden
  - Ziel:
    - Sequenz von Aktorausführungen, die folgende Bedingungen erfüllt:
      - Jeder Aktor wird entsprechend seiner Rate ausgeführt
      - Die Aktoren sind an der Position im Ausführungsplan bereit für die Ausführung (Daten liegen an)
      - Kein Prozess darf zu oft laufen (Datenüberlauf)
  - Schritte:
    - Berechnung der relativen Ausführungsraten (Lösung von linearen Gleichungen)
    - Bestimmung eines periodischen Ausführungsplans durch Simulierung einer Runde (so dass die Belegung am Ende der Runde der Initialbelegung entspricht)
  - Typischerweise gibt es mehrere Ausführungspläne, durch entsprechende Wahl kann eine Reduktion der Codegröße, der Puffergröße erreicht werden

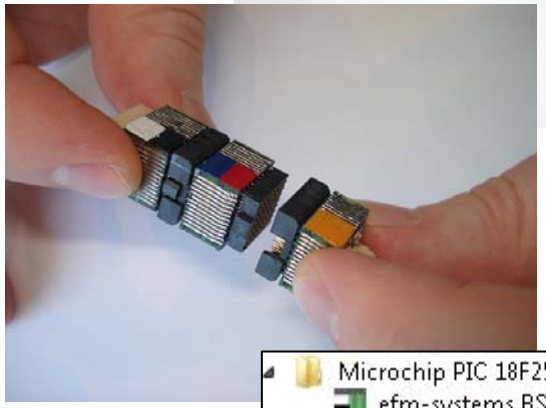
Berechnung eines Ausführungsplans

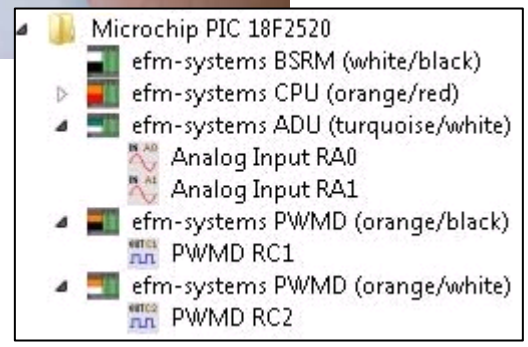
- Beispiel:
  - Raten der Aktoren:
    - a=3
    - b=1
    - c=4
    - d=2
  - Mögliche Ausführungspläne:
    - AAABCCCCDD
    - AAACCCCBDD
    - AAABCCDCCD
    - …

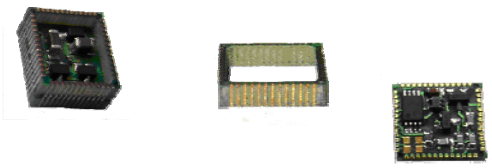


Anwendungsbeispiel

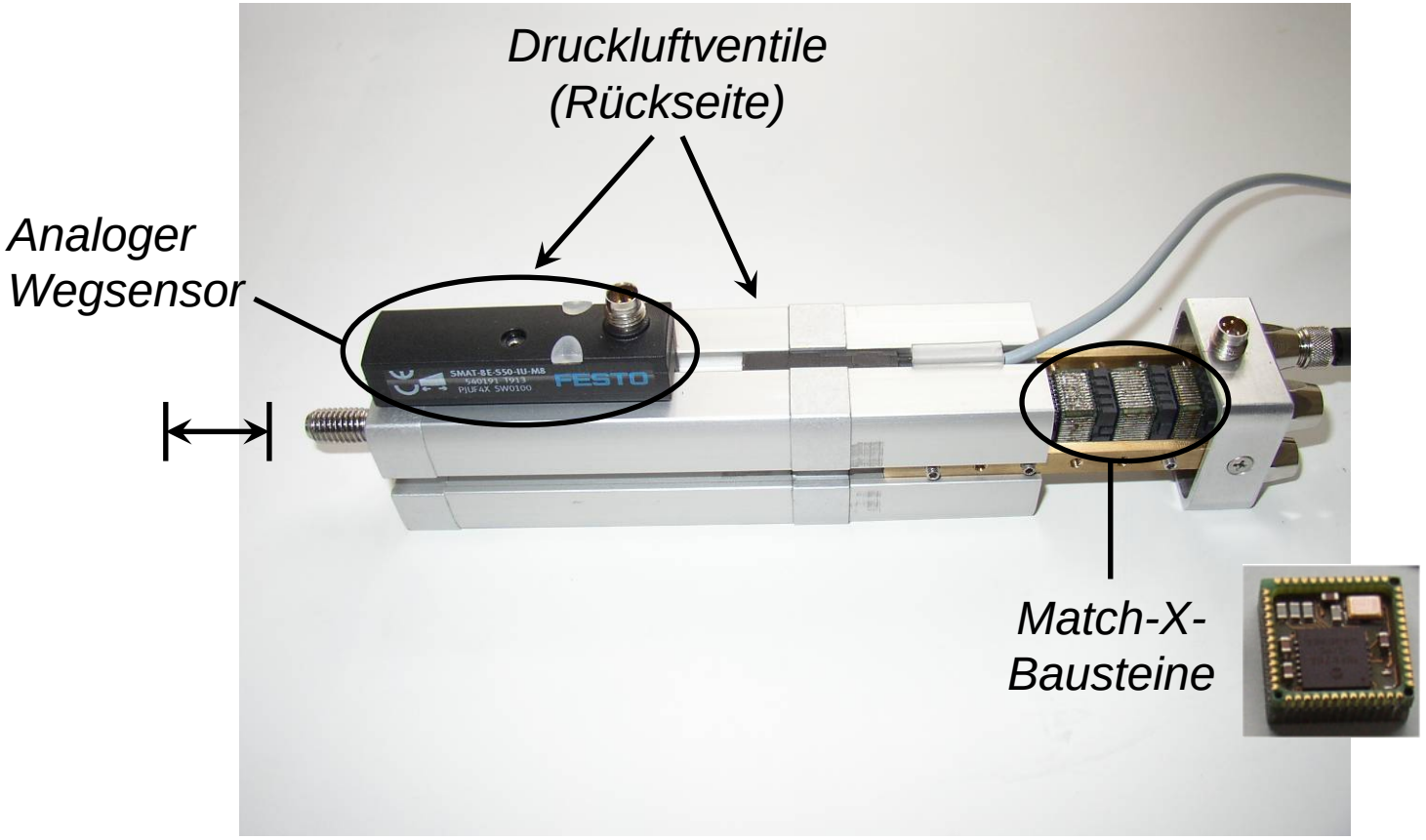
- Hardware
  - Zylinder: Kolbenposition, Einstellregler, 2 magnetische Ventile
  - Regelungsmodule: Match-X Bausteine:
    MCU Modul, ADC Modul,
    2 Treiberbausteine für induktive Lasten
- Ziel: Positionsregelung des Kolbens
- Implementierung:
  - Hardwaremodell basiert auf Bibliothek der Match-X Bausteine (Microchip PIC18F MCU)
  - Anwendungslogik
    - Einfaches Datenflussdiagramm
    - Integration der Hardwarelogik







Pneumatikzylinder



Modellierung von Echtzeitsystemen

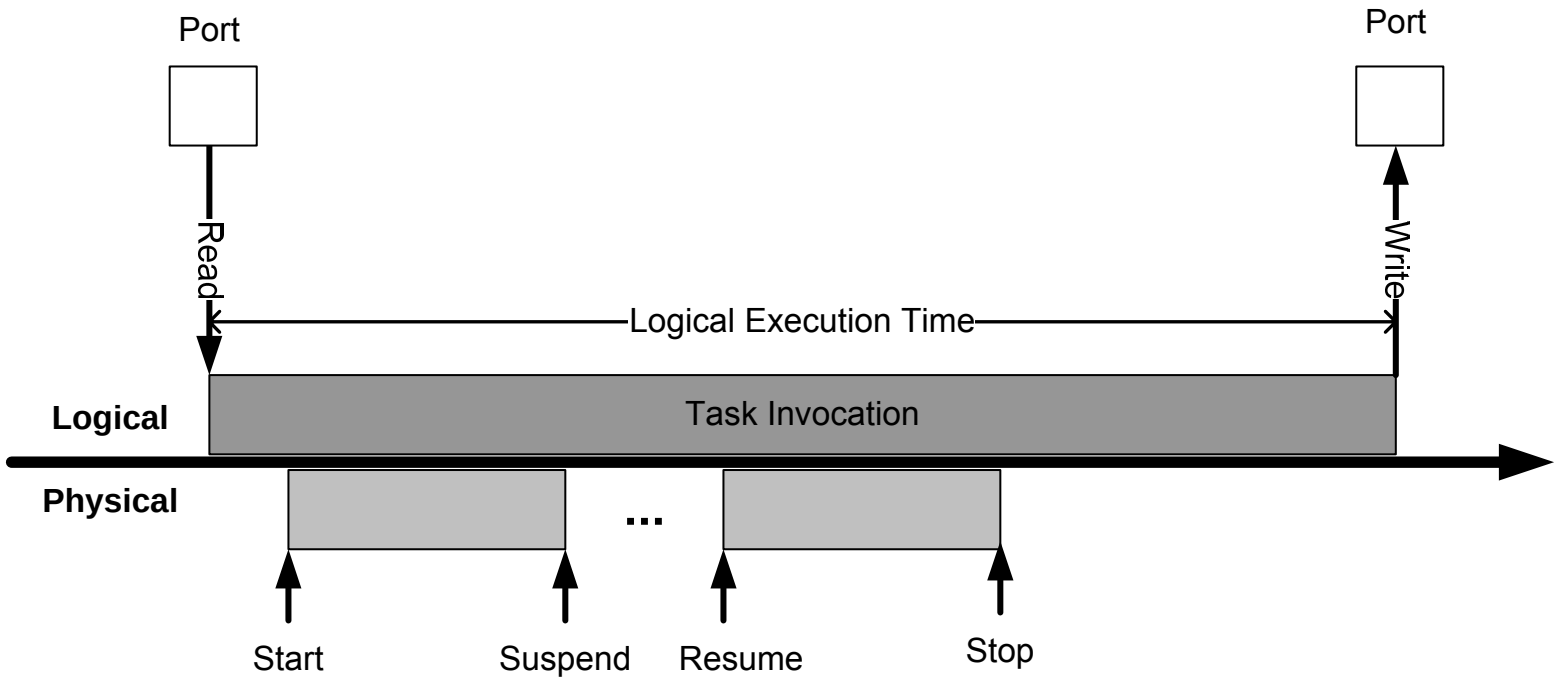
Zeitgesteuerte Systeme

Werkzeug: Giotto

Giotto: Hintergrund

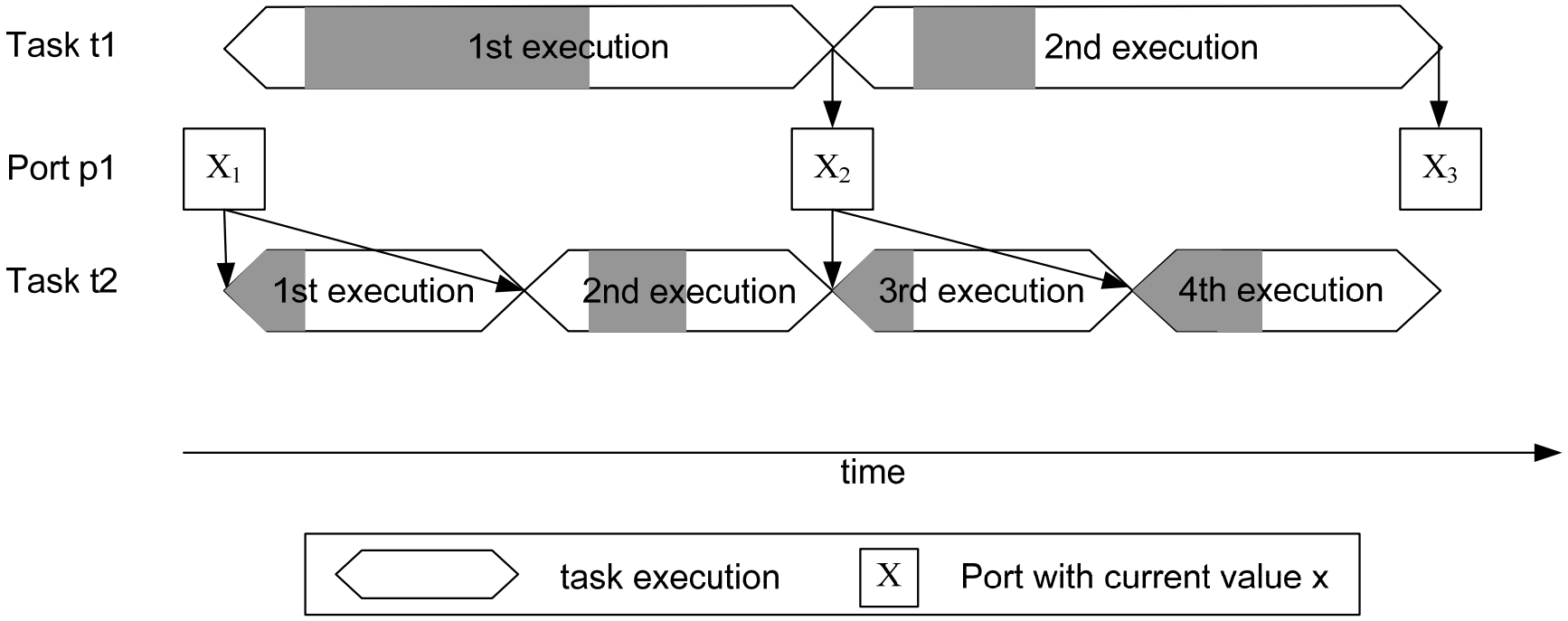
- Programmierumgebung für eingebettete Systeme (evtl. ausgeführt im verteilten System)
- Ziel:
  - strikte Trennung von plattformunabhängiger Funktionalität und plattformabhängigen Scheduling und Kommunikation
  - temporaler Determinismus
- Hauptkonzept: Logische Ausführungszeiten
- Aktoren:
  - Tasks
    - Programmblock aus sequentiellen Code
    - keine Synchronisationspunkte, blockende Operationen erlaubt
    - Schnittstellen: Ports
  - Drivers: realisieren die Kommunikation zwischen Ports
  - Flexibilität durch Modes/Guards
- Ausführung durch virtuelle Maschinen:
  - Embedded Machine: Reaktion der Tasks auf physikalische Ereignisse
  - Scheduling Machine: physikalisches Scheduling
- http://embedded.eecs.berkeley.edu/giotto/

Logische Ausführungszeit



Motivation siehe http://www.cs.uic.edu/~shatz/SEES/henzinger.slides.ppt

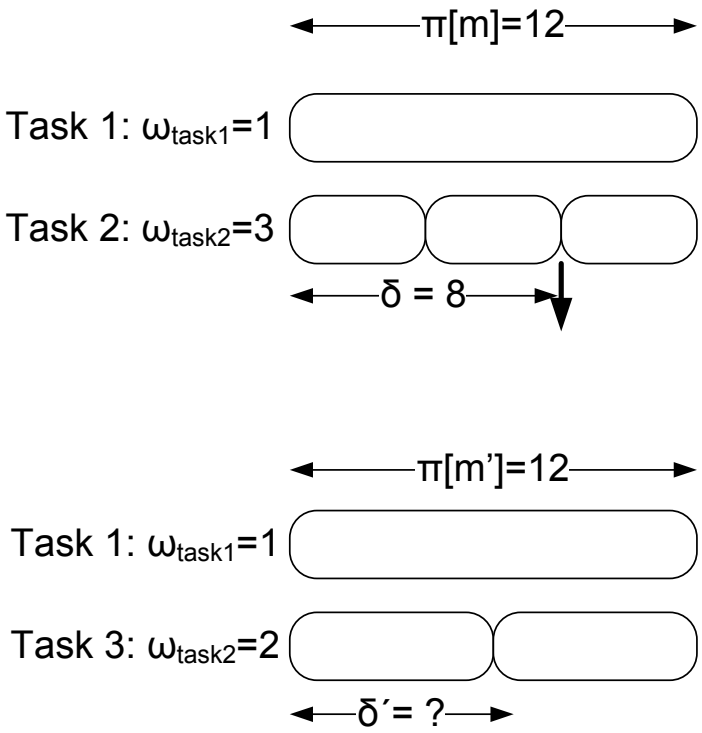
Kommunikation zwischen Tasks



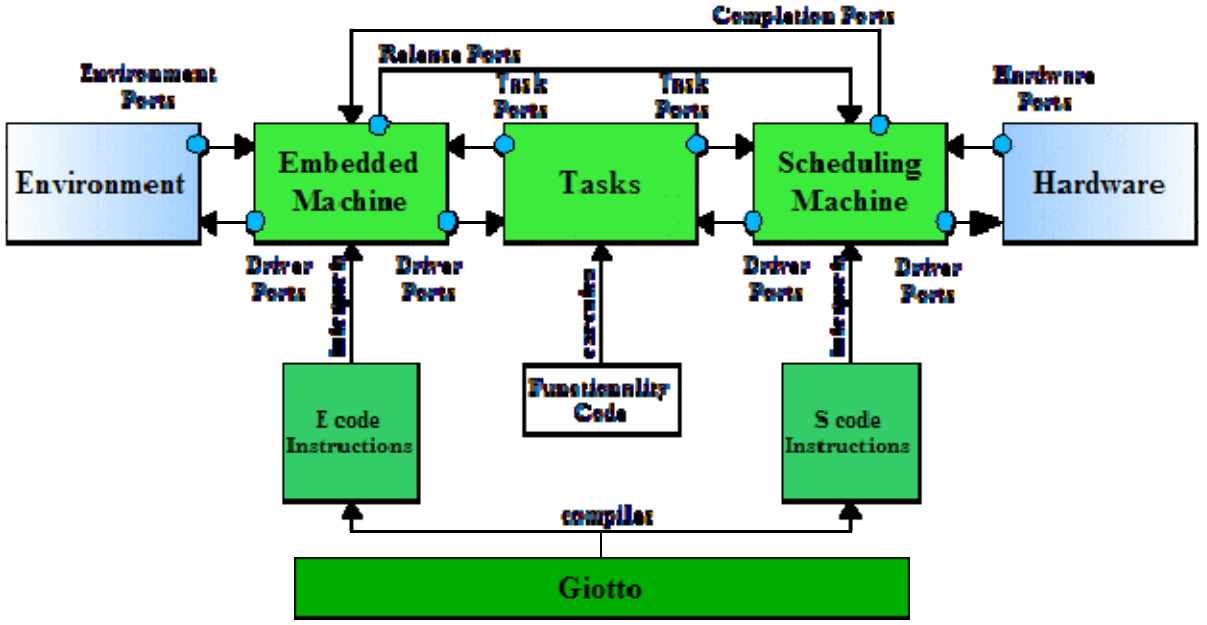
Modes / Guards

- Um die Ausführung flexibel zu gestalten, bietet Giotto Modes und Guards an
  - Guards: Boolsche Funktion, die über die Ausführung eines Tasks entscheidet (wird vor Start des Tasks aufgerufen)
  - Mode: Menge von Tasks und Drivers die zeitgleich ausgeführ werden, es kann immer nur ein Mode aktiv sein.
- Nicht-harmonischer Moduswechsel (Unterbrechung eines laufenden Modes):
  - Voraussetzung: $\pi[m]/\omega_{task} = \pi[m']/\omega'_{task}$
    m: Quellmodus, m': Zielmodus, $\pi[m]$: Modusdauer m, ω_{task}: Taskfrequenz $\Rightarrow$ Logische Ausführungszeit muss gleich sein
  - Wechselmechanismus:

    γ = LCM $\{\pi[m]/\omega_{task} | \{\omega_{task}, t,) \in$ Invokes[m],
    $\delta' = \pi[m']-(\varepsilon-\delta)$ mit $\varepsilon = n^* \gamma \geq \delta$
    LCM: least common multiple, δ: aktuelle Rundenzeit, δ': neue Rundenzeit in m', $\varepsilon - \delta$: Zeit bis zum nächsten gleichzeitigen Beendigungspunkt



Ausführungsumgebung



Zusammenfassung

- Das Konzept der logischen Ausführungszeiten erlaubt eine Abstrahierung von der physikalischen Ausführungszeit und somit die Trennung von plattformunabhängigem Verhalten (Funktionalität und zeitl. Verhalten) und plattformabhängiger Realisierung (Scheduling, Kommunikation)
- Die Ausführung erfolgt über zwei virtuelle Maschinen:
  - E-Machine: Interaktion mit der Umgebung (reaktiv)
  - S-Machine: Interaktion mit der ausführenden Plattform (proaktiv), Vorteil: Schedule kann vorab berechnet werden
- Weitere Literaturhinweise:
  - Henzinger et al.: Giotto: A time-triggered language für embedded programming, Proceedings of the IEEE, vol.91, no.1, pp. 84-99, Jan 2003
  - Henzinger et al.: Schedule-Carrying Code, Proceedings of the Third International Conference on Embedded Software (EMSOFT), 2003

Modellierung von Echtzeitsystemen

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

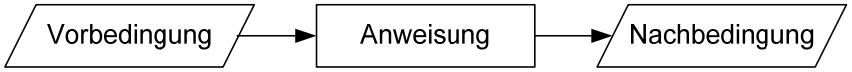
Testen

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 ⇒ 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

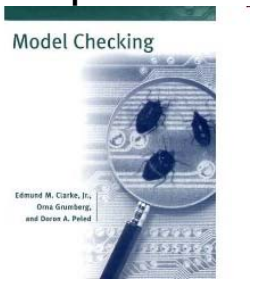
- Mittels Verifikation soll überprüft werden, dass:
  - Fehlerzustände nie erreicht werden
    - Der Aufzug soll nie mit offener Tür fahren.
  - ein System irgendwann einen bestimmten Zustand erreicht (und evtl. dort verbleibt)
    - Nach einer endlichen Initialisierungsphase, geht der Aufzug in den 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

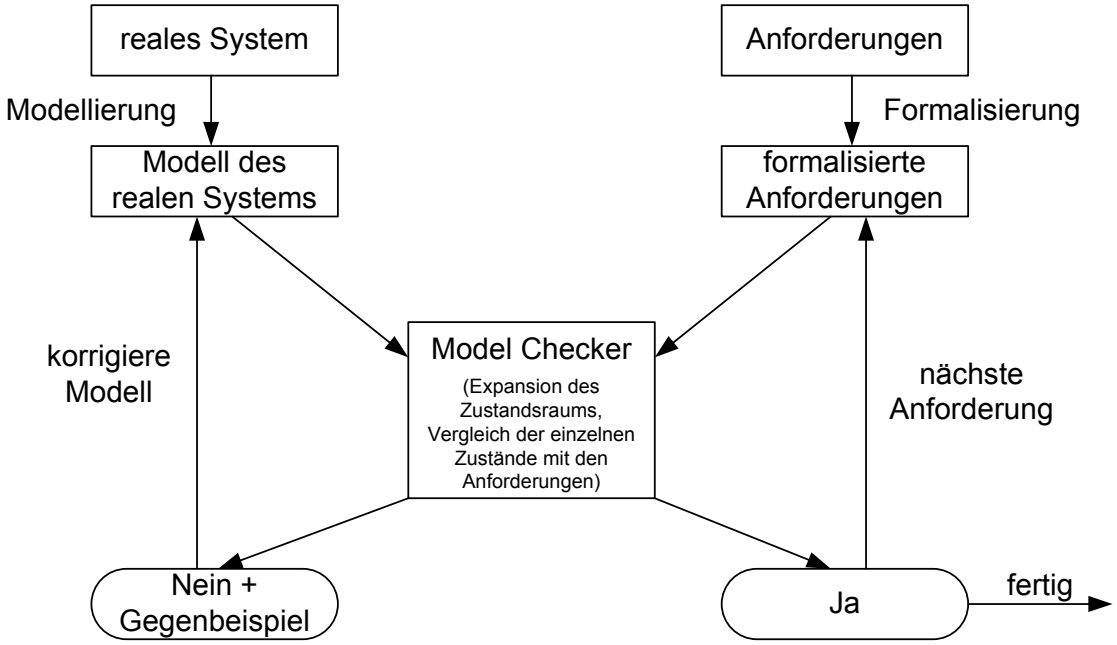
- Zur Darstellung eines Systems werden Kripke-Strukturen K = (V,I,R,B) und eine endliche Menge P von atomaren logischen Aussagen verwendet.
  - V: Menge binärer Variablen (z.B. Tür offen, Aufzug fährt)
  - Die Zustandsmenge S ergibt sich aus allen möglichen Kombinationen über V, somit gilt $S = B^V$
  - 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
- Literaturhinweis: Edmund M. Clarke, Orna Grumberg, Doron A. Peled, *Model Checking*, 1999, MIT Press



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:

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

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

1. Beispiel: Verifikation in Esterel Studio

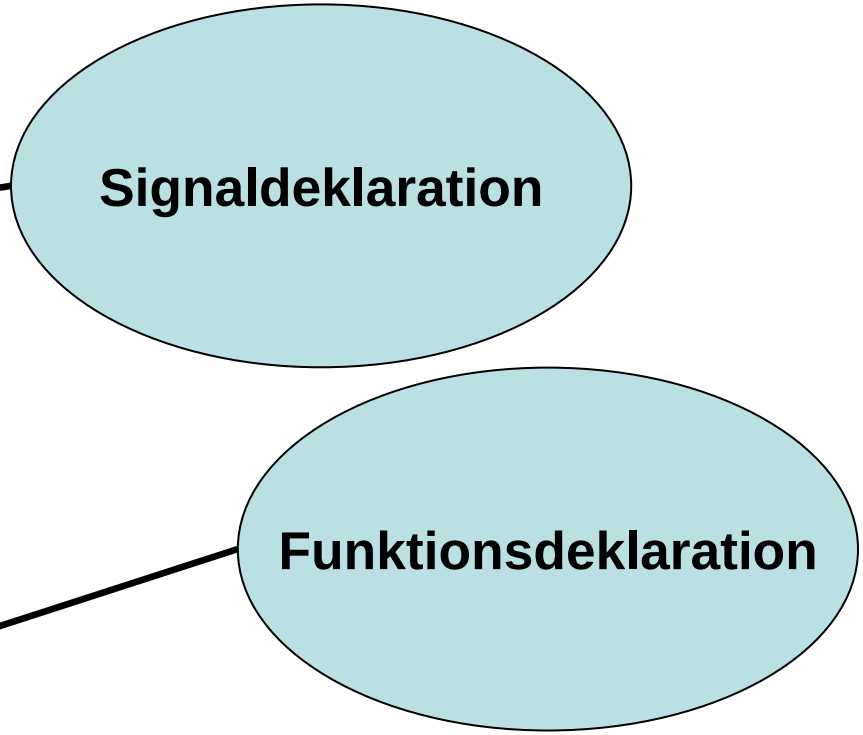
- Esterel Studio bietet eine eingebaute Verifikationsfunktionalität zur einfachen Verifikation von Programmen
- Zur Modellierung der verschiedenen Eigenschaften kann das Schlüsselwort `assert` verwendet werden.
- Im Verifikationsmodus können die Eigenschaften dann getestet werden, dabei stehen Methoden 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)

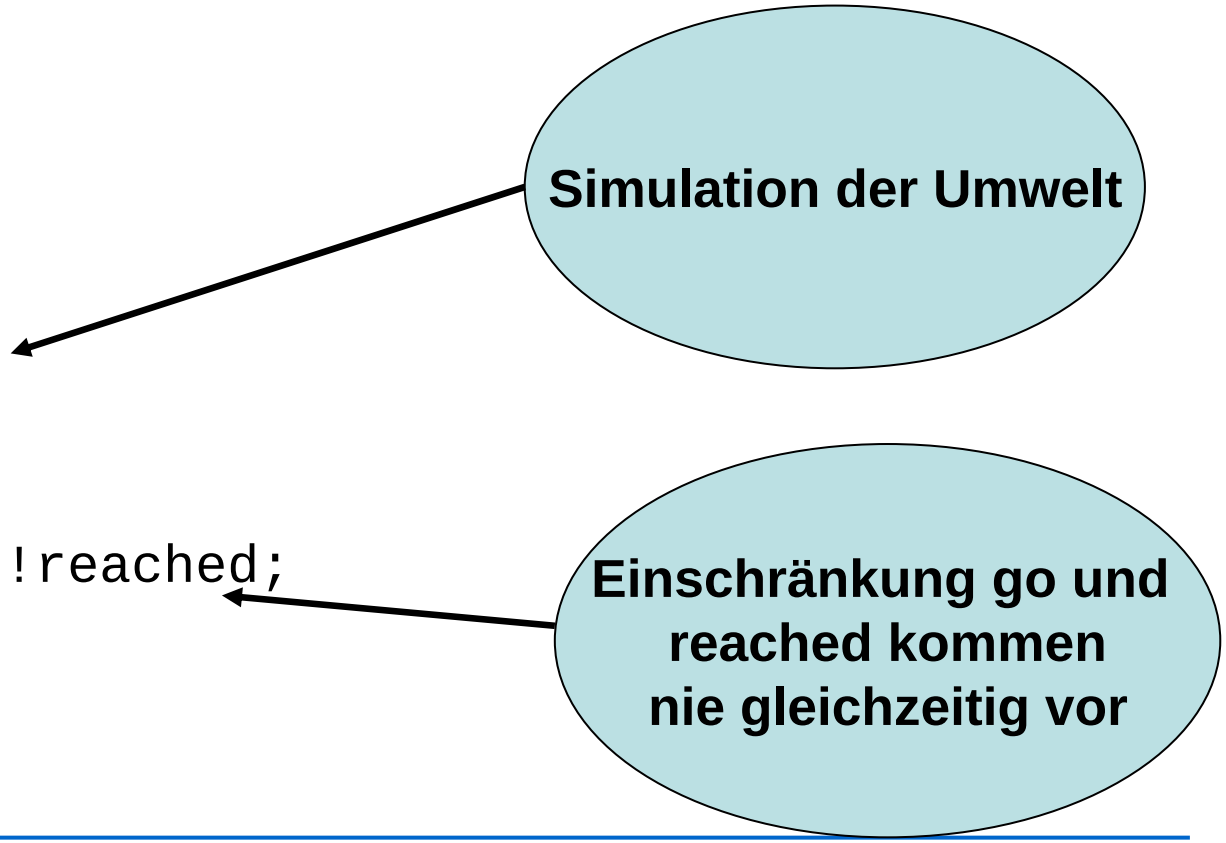
Boogie-Programm: Türbeispiel vereinfacht

```
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;
```



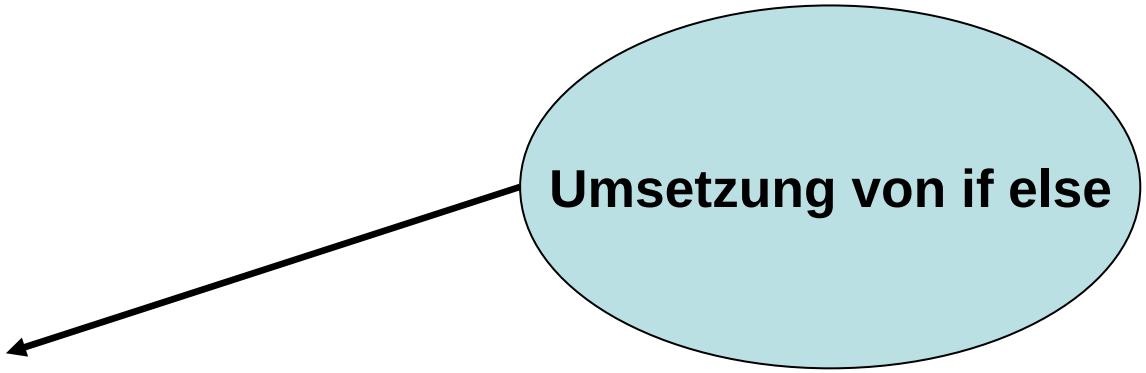
Boogie-Programm: Türbeispiel vereinfacht

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



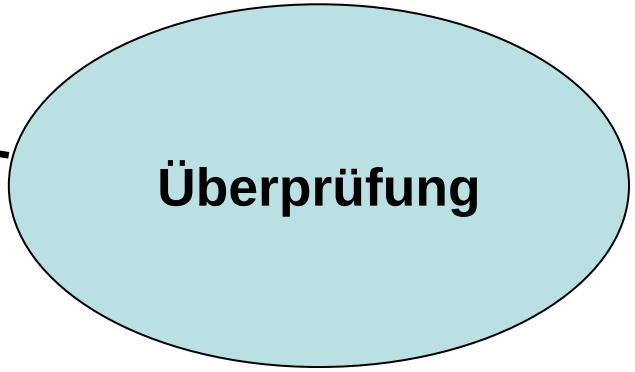
Boogie-Programm: Türbeispiel vereinfacht

```
   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;
}
```



Boogie-Ergebnis:

```
D:\boogie>boogie door.bpl -enhancedErrorMessages:1
Spec# Program Verifier Version 0.87, Copyright (c) 2003-2007, Microsoft.
Information extracted from prover model:
close == True
sig_close == False
Failing assertion: close ==> sig_close
door.bpl(35,2): Error BP5001: This assertion might not hold.
Execution trace:
    door.bpl(13,1): Begin
    door.bpl(23,1): Open
    door.bpl(33,1): End

Spec# Program Verifier finished with 0 verified, 1 error
```

Modellierung von Echtzeitsystemen

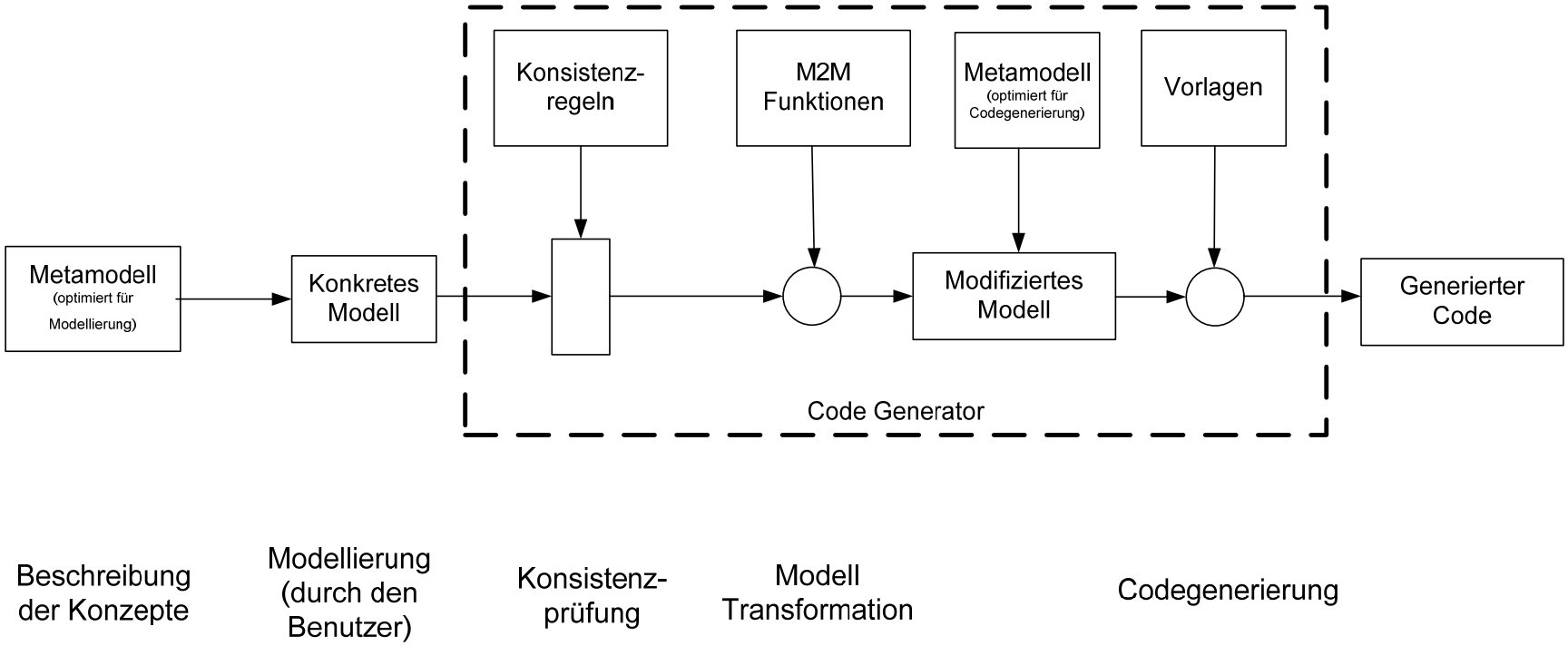
Entwicklung von Domänenspezifischen Codegeneratoren

Beispiel: FTOS

Domänenspezifische Codegeneratoren

- Beobachtung: Viele existierende Werkzeuge beschränken sich auf die Generierung der Anwendungsfunktionalität
- Grund: es kann plattformunabhängiger Code (z.B. ANSI-C) generiert werden
- Die Interaktion mit der zugrunde liegenden Hardware wird von wenigen Codegeneratoren unterstützt
  ⇒ Trend zu domänenspezifischen Codegeneratoren
- Domänenspezifische Codegeneratoren
  - Konzentrieren sich auf Konzepte/Mechanismen der Anwendungsdomäne
  - Zeichnen sich häufig durch gute Erweiterbarkeit aus
  - Können als eine Weiterentwicklung von komponentenorientierten Entwicklungsansätzen interpretiert werden
- Es existieren gute Infrastrukturen zum schnellen Erstellen von solchen Entwicklungswerkzeugen, z.B. openArchitectureWare, metaEdit, GME

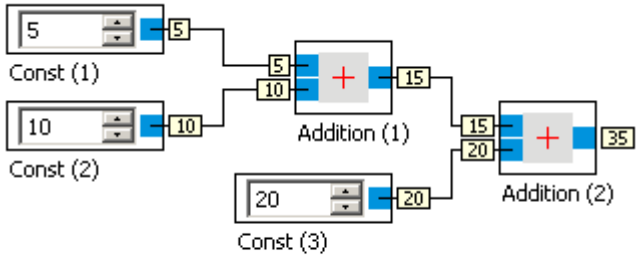
Bestandteile eines Codegenerators



Bestandteile eines Codegenerators

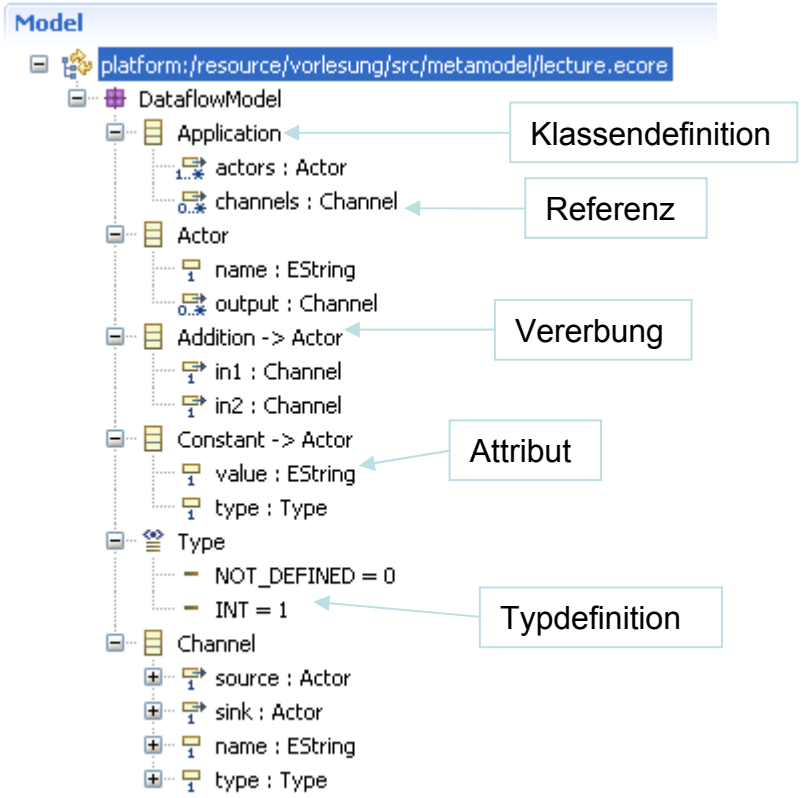
- Metamodelle:
  - Das Metamodell definiert die Modellierungskonzepte und die Zusammenhänge
  - Auf Basis des Metamodells werden Konsistenzregeln, Modell-zu-Modelltransformationsregeln, sowie die Codegenerierungsregeln definiert
- Konkretes Modell
  - Das konkrete Modell durch den Entwickler basiert auf dem Metamodell und beschreibt die konkrete Anwendung
- Konsistenzregeln:
  - Die fehlerfreie Modellierung wird durch entsprechende Konsistenzregeln geprüft
- Modell-zu-Modell-Transformation:
  - Durch eine Modell-zu-Modell-Transformation können mehrere Modelle zusammengefasst und die leichte Modellierung, sowie Codegenerierung gewährleistet werden
  - Die Modell-zu-Modell-Transformation überführt ein Modell basierend auf einer für eine einfache Modellierung optimiertes Metamodell in ein Modell basierend auf ein für die Codegenerierung optimiertes Modell
- Vorlagen:
  - Vorlagen beschreiben die Regeln zur Generierung von Code auf der Basis eines konkreten Modells

Beispiel: Codegenerator für Synchronen Datenfluss



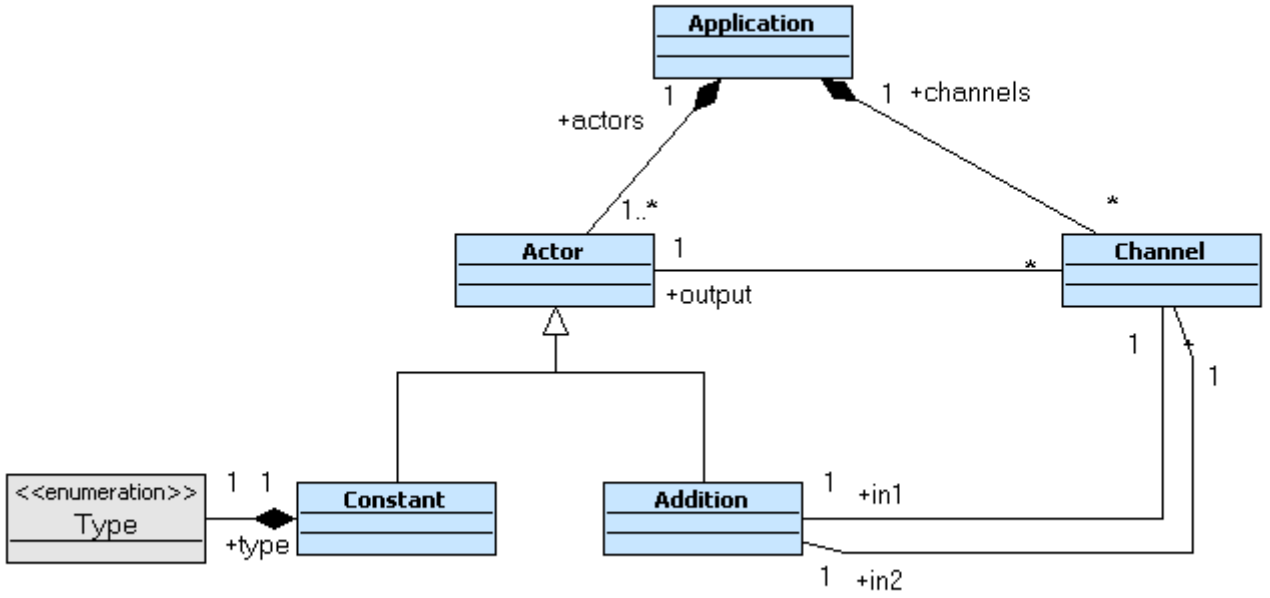
- Einschränkungen zur Vereinfachung:
  - Jeder Aktor hat nur einen Ausgang namens output
  - Maximal ein Aktor darf einen nicht verbundenen Ausgang besitzen ⇒ Ergebnis
- Verwendung von openArchitectureWare als Codegenerierungsinfrastruktur http://www.openarchitectureware.org/
- Download Komplettinstallation: http://www.gentleware.com/oaw.html
- Den Codegenerator können Sie auf der Vorlesungsseite herunterladen

Metamodelle



- Das Metamodell definiert, wie eine Anwendung beschrieben werden kann
- Zur Definition der Metamodelle werden in der Regel Klassendiagramme bestehend aus Klassen, sowie deren Attribute und Referenzen, verwendet.
- oAW verwendet dazu EMF (eclipse modeling framework)
- Eigene Datentypen können in EMF durch Aufzählungen definiert werden
- Bei Referenzen wird zwischen „normalen Referenzen“ und Komposition unterschieden

Definition of Underlying Meta-Model: UML-Model



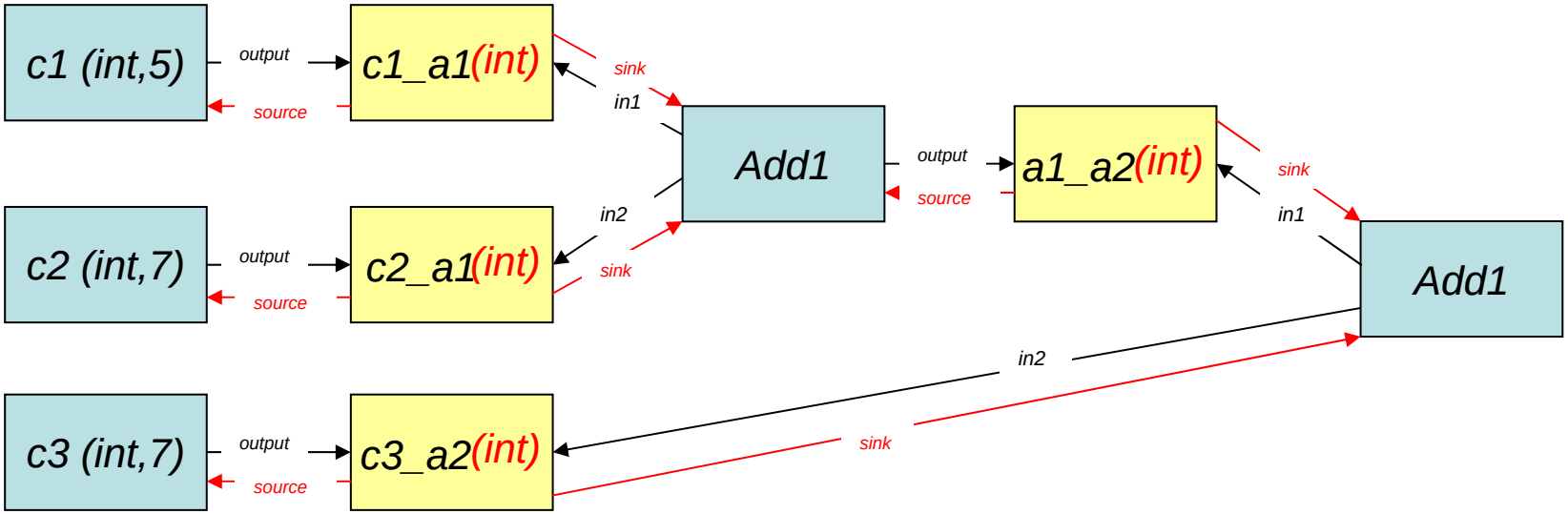
Konsistenzprüfung

- Vor der Codegenerierung ist eine Prüfung der Korrektheit der Modelle möglich
- oAW unterstützt die Definition von Testbedingungen in Prädikatenlogik 1. Stufe
- Beispiel:

```
context Actor ERROR "Actor "+ this+": Name must be set and be unique":
    (this.name!=null) &&
    (eRootContainer.eAllContents.typeSelect(Actor).select(a | a.name== this.name).size==1);
```

- Erläuterung:
  - this: referenziert das aktuelle Objekt
  - eRootContainer: referenziert das Wurzelelement
  - eAllContents: liefert eine Liste aller vom aktuellen Objekt enthaltenen Objekte
  - typeSelect: beschränkt eine Menge auf alle Objekte des angegebenen Typs
  - select: wählt alle Element aus einer Menge, die die angegebene Bedingung erfüllen

Aktueller Stand - Codegenerator



Durch Modell-zu-Modelltransformation hinzugefügte Information

M2M-Transformation

- Mittels Modell-zu-Modell-Transformationen können Modelle ineinander überführt werden.
- Ziel:
  - Explizites Aufführen von Informationen, die bereits implizit enhalten sind
  - Zusammenführen von verschiedenen Modellen
- M2M im Beispiel:
  - Berechnung der Datentypen für jeden Kanal
  - Zuweisung der Endpunkte jedes Kanals
- openArchitectureWare bietet die Sprache EXTEND zur Definition der M2M-Transformation
  - EXTEND unterstützt Polymorphismus und Aufrufe von Java-Funktionen
  - Auf Attribute und Referenzen mit Kardinalität 1 kann über entsprechende set- und get-Operationen zugegriffen werden

```
Void addActors2Channel(Addition a):
    a.output.setSource(a)->
    a.in1.setSink(a)->
    a.in2.setSink(a);
```

  - Attribute und Referenzen mit Kardinalität >1 werden als Listen behandelt, oAW sucht automatisch die entsprechende Funktion bei Listen als Parameter (entweder Aufruf der Funktion mit Liste als Parameter oder Aufruf für jedes einzelne Element der Liste)
  - Für eine einfache Behandlung der Liste stehen diverse Operationen zur Verfügung (select,typeSelect,remove…)

```
Void transformModel(Application app):
    app.actors.addActors2Channel()->
    app.actors.typeSelect(Constant).propagateType();
```

Codegenerierung

- Die Codegenerierung erfolgt als Textersetzung ähnlich Makroprozessoranweisungen
- oAW bietet mit EXPAND eine entspreche Makrosprache
  - EXTEND ist eine iterative Sprache mit wenigen und einfachen Konzepten (siehe nächste Folie)
  - Polymorphismus wird zur einfachen Codegenerierung unterstützt
  - Genereller Ablauf:
    - **Prinzip**: Alles zwischen den Escapezeichen wird interpretiert, der Rest kopiert
    - Tags werden durch die Escapezeichen « und » (Tastatur **Ctrl+<** und **Ctrl+>**) gekennzeichnet

EXPAND Sprachkonstrukte I:

- DEFINE: Definition einer Generierungsfunktion

```
«DEFINE generateGetFunction FOR Const-»
 ...
«ENDDEFINE»
```

- EXPAND: Aufruf einer Generierungsfunktion

```
«EXPAND generateActorCode FOR a
```

- FILE: Erzeugung einer Datei

```
«FILE "actor_code.c"-»

 ...

«ENDFILE»
```

- FOREACH: Generierung von Code für jedes Objekt einer Liste

```
«FOREACH actors AS a»«EXPAND generateActorVariables FOR a-»«ENDFOREACH-»
«FOREACH actors AS a»«EXPAND generateActorCode FOR a»
```

```
«FOREACH max.channelList AS ch ITERATOR c-»
```

EXPAND Sprachkonstrukte II

- IF / ELSE zur Verzweigung innerhalb der Codegenerierung:

```
«IF type==Type::INT»
        ...
«ELSE»
        ...
«ENDIF»
```

- Auf Attribute kann direkt über den Namen zugegriffen werden:

```
int channel_«source.name»_«sink.name»;
```

- REM: Kommentare

```
«REM»
Abstract class MinMaxActor
«ENDREM»
```

- ERROR: Abbruch der Codegenerierung mit Fehlernachricht:

```
«ERROR "Type of constant "+name+" currently not supported in code generation"»
```

Beschreibung des Arbeitsbereiches

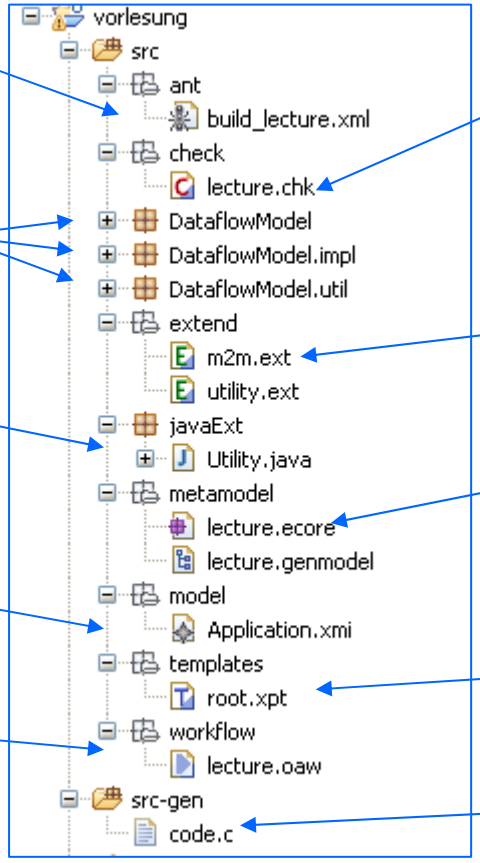
ANT Funktionen zur Automatisierung

Automatisch generierte Java-Klassen

Java-Funktionen für Modell-zu-Modell-Transformation und Codegenerierung

Konkretes Modell

Prozessablaufsbeschreibung



Konsistenzregeln

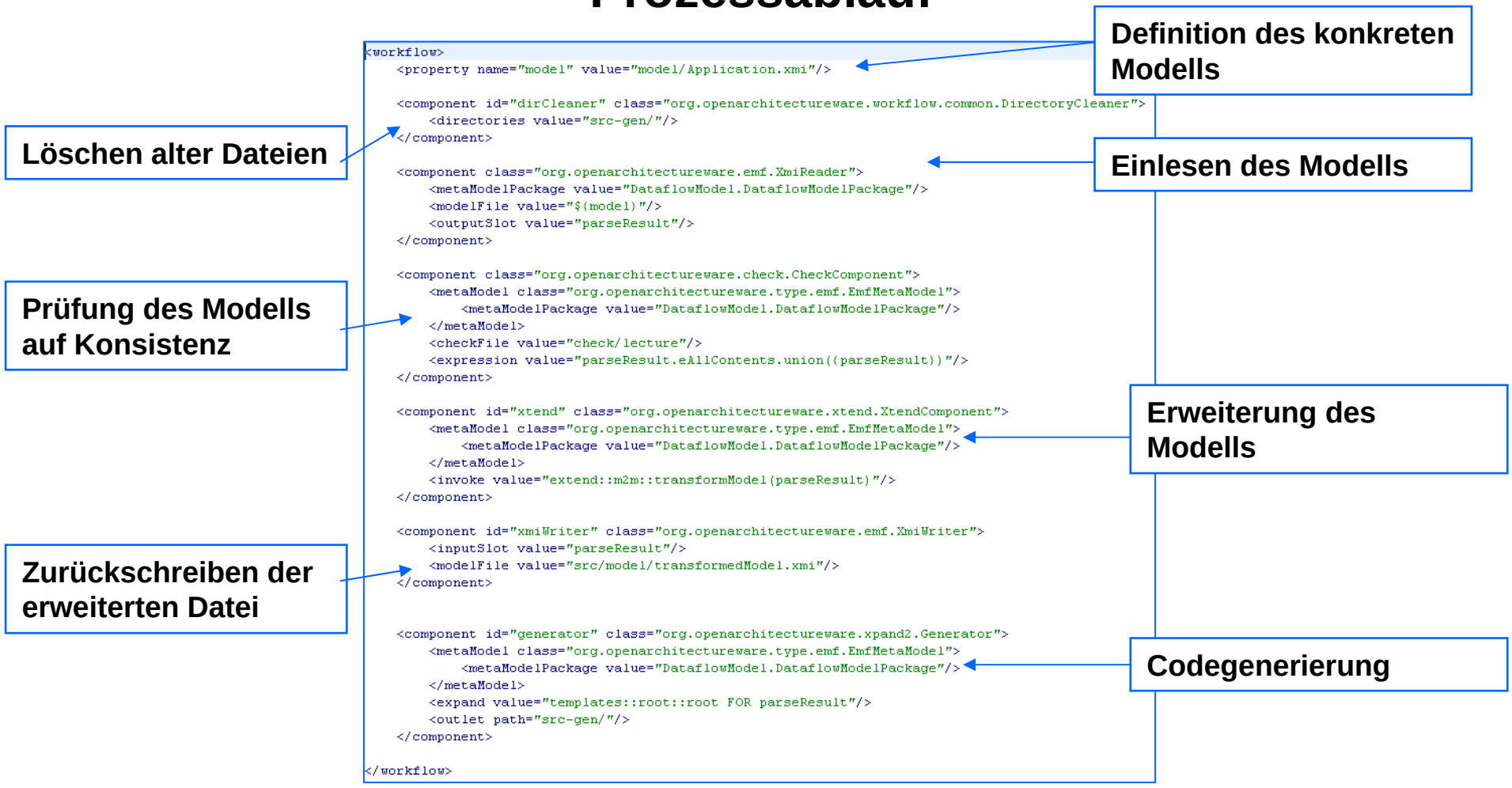
Funktionen zur Modell-zu-Modell-Transformation

Definition des Meta-Modells

Vorlagen zur Codegenerierung

Generierte Dateien

Prozessablauf




```
<workflow>
    <property name="model" value="model/Application.xmi"/>

    <component id="dirCleaner" class="org.openarchitectureware.workflow.common.DirectoryCleaner">
        <directories value="src-gen/"/>
    </component>

    <component class="org.openarchitectureware.emf.XmiReader">
        <metaModelPackage value="DataflowModel.DataflowModelPackage"/>
        <modelFile value="${model}"/>
        <outputSlot value="parseResult"/>
    </component>

    <component class="org.openarchitectureware.check.CheckComponent">
        <metaModel class="org.openarchitectureware.type.emf.EmfMetaModel">
            <metaModelPackage value="DataflowModel.DataflowModelPackage"/>
        </metaModel>
        <checkFile value="check/lecture"/>
        <expression value="parseResult.eAllContents.union({parseResult})"/>
    </component>

    <component id="xtend" class="org.openarchitectureware.xtend.XtendComponent">
        <metaModel class="org.openarchitectureware.type.emf.EmfMetaModel">
            <metaModelPackage value="DataflowModel.DataflowModelPackage"/>
        </metaModel>
        <invoke value="extend::m2m::transformModel(parseResult)"/>
    </component>

    <component id="xmiWriter" class="org.openarchitectureware.emf.XmiWriter">
        <inputSlot value="parseResult"/>
        <modelFile value="src/model/transformedModel.xmi"/>
    </component>


    <component id="generator" class="org.openarchitectureware.xpand2.Generator">
        <metaModel class="org.openarchitectureware.type.emf.EmfMetaModel">
            <metaModelPackage value="DataflowModel.DataflowModelPackage"/>
        </metaModel>
        <expand value="templates::root::root FOR parseResult"/>
        <outlet path="src-gen/"/>
    </component>

</workflow>
```

Beispiel FTOS: Hintergrund

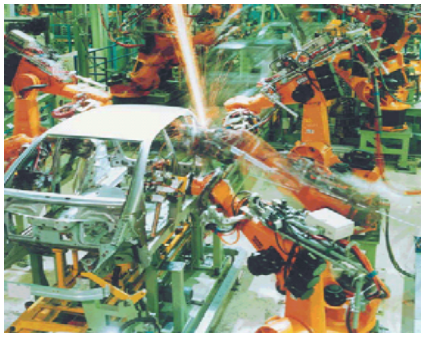
- Programmierumgebung für fehlertolerante, verteilte Echtzeitsysteme
- Komplette Werkzeugkette von der Modellierung bis zur Codegenerierung für verschiedenste Plattformen
- Fokus liegt auf der Generierung von nicht-funktionalen Aspekten
- Ausgelegt auf Anwendungen, die traditionell nicht oder nur minimal fehlertolerant entwickelt wurden.
- Es wird kein eingeschränktes Fehlermodell zugrunde gelegt.

The operating system must provide basic support for guaranteeing real-time constraints, supporting fault tolerance and distribution, and integrating time-constrained resource allocations and scheduling across a spectrum of resource types, including sensor processing, communications, CPU, memory, and other forms of I/O.

John A. Stankovic, Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems, 1988

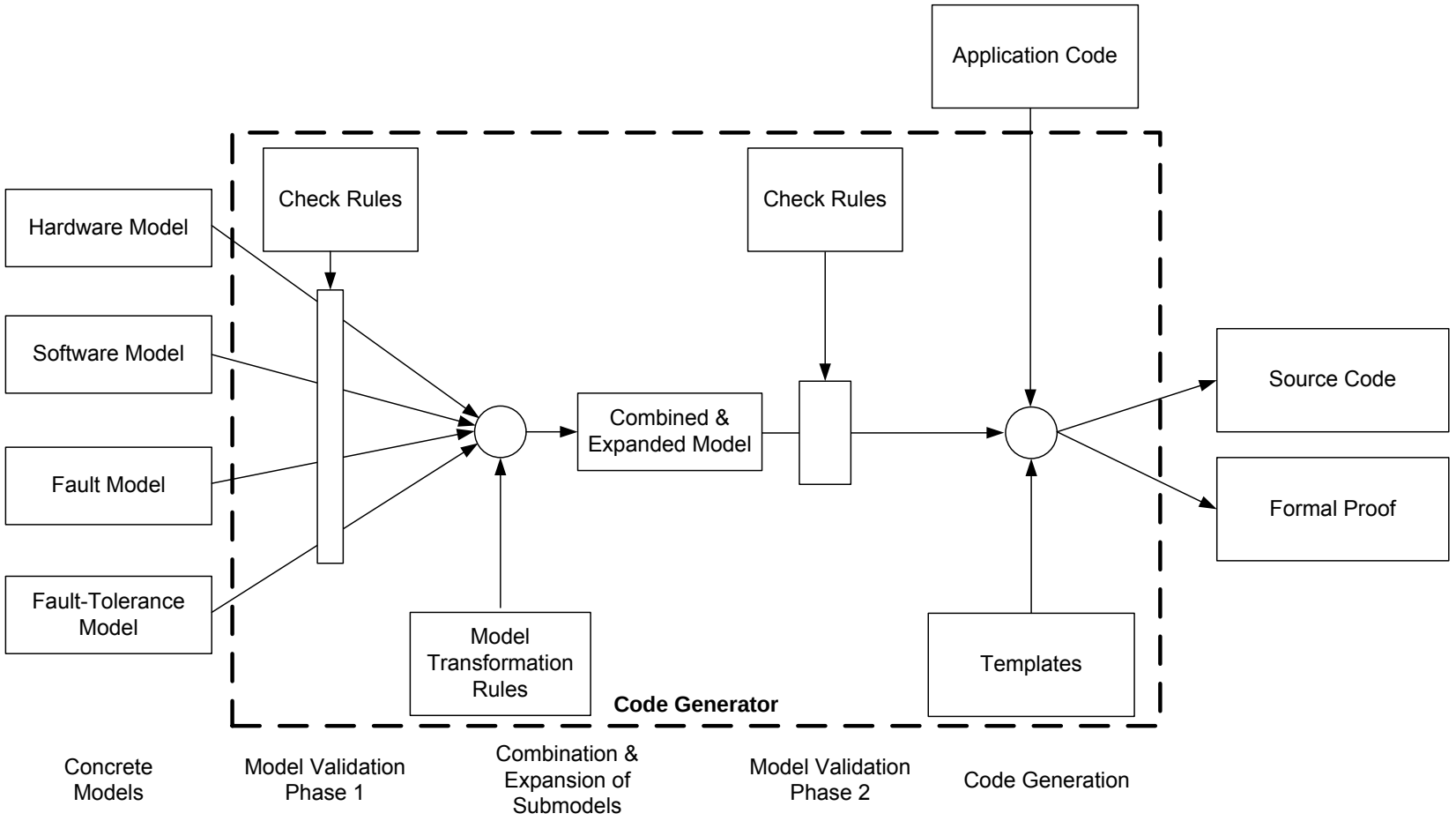
Stromerzeugung

Medical

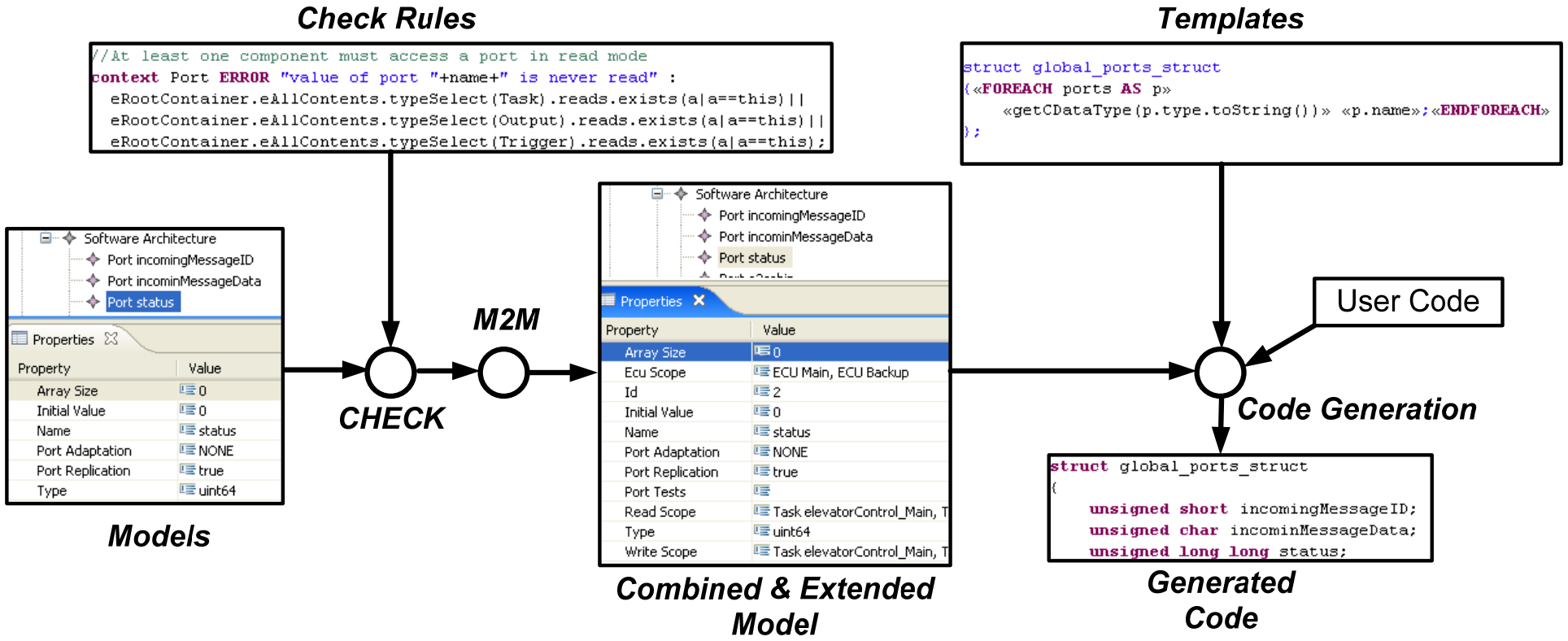


Automatisierung

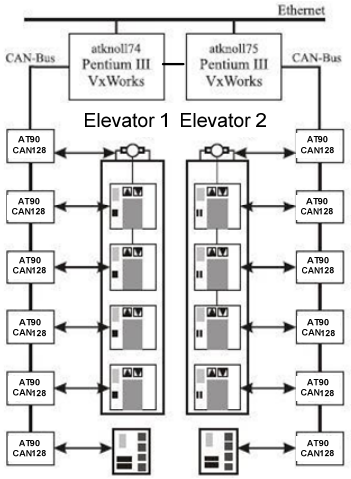
FTOS: Codegenerator-Architektur



Ein konkretes Beispiel



Demonstratoren



Schwebender Stab (auf 2-aus-3-Rechnersystem)

→ Regelungsrate 2,5 ms

→ nur 24 Zeilen zur Programmierung des PID-Reglers notwendig

Aufzugssteuerung

→ Codegenerierung für 8-Bit Atmel-Controller bzw. Standarddesktop-rechner

→ In Kombination mit EasyLab 100% modellbasierte Entwicklung möglich

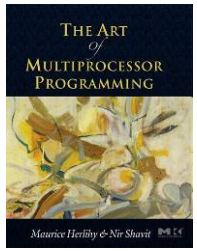
Kapitel 3

Nebenläufigkeit

Inhalt

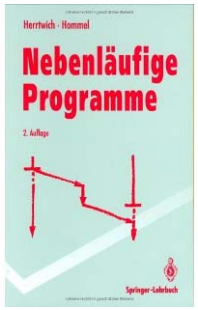
- Motivation
- Unterbrechungen (Interrupts)
- (Software-) Prozesse
- Threads
- Interprozesskommunikation (IPC)

Literatur



Maurice Herlihy, Nir Shavit,
The Art of Multiprocessor
Programming, 2008

A.S.Tanenbaum, Moderne
Betriebssysteme, 2002



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

- 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.

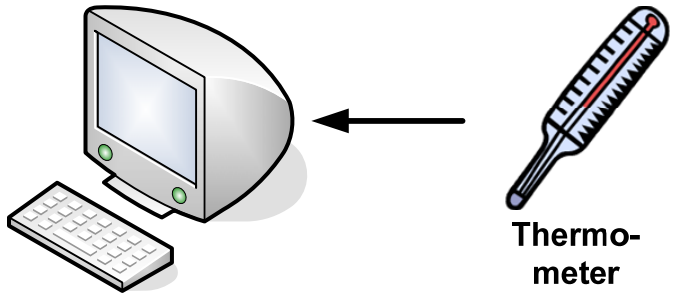
  Die Nebenläufigkeit von mehreren unabhängigen Prozessen bezeichnet man als *Multitasking*;

  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

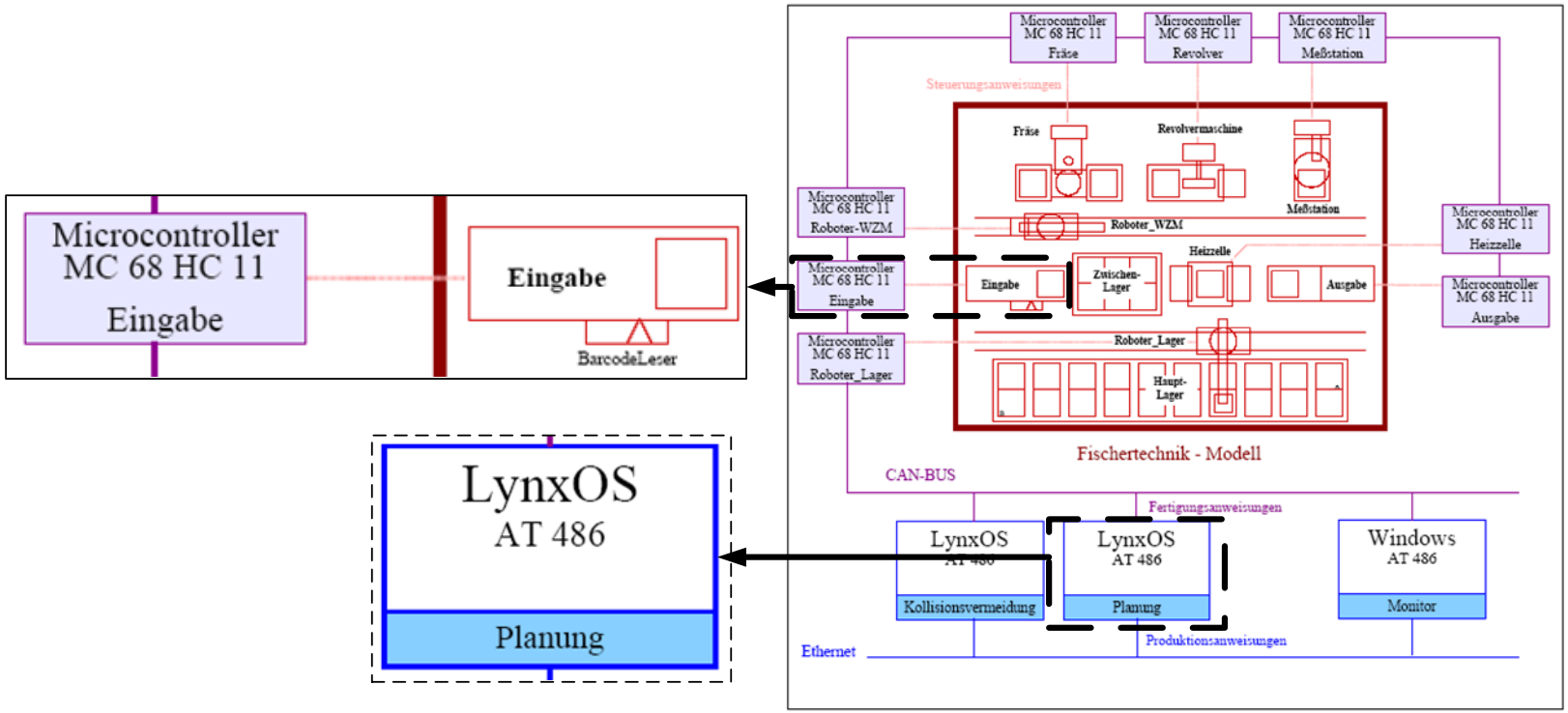
Anwendungsfälle für Nebenläufigkeit (Unterbrechungen)



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

Allgemeines Anwendungsgebiet: hauptsächlich zur Anbindung von externer Hardware

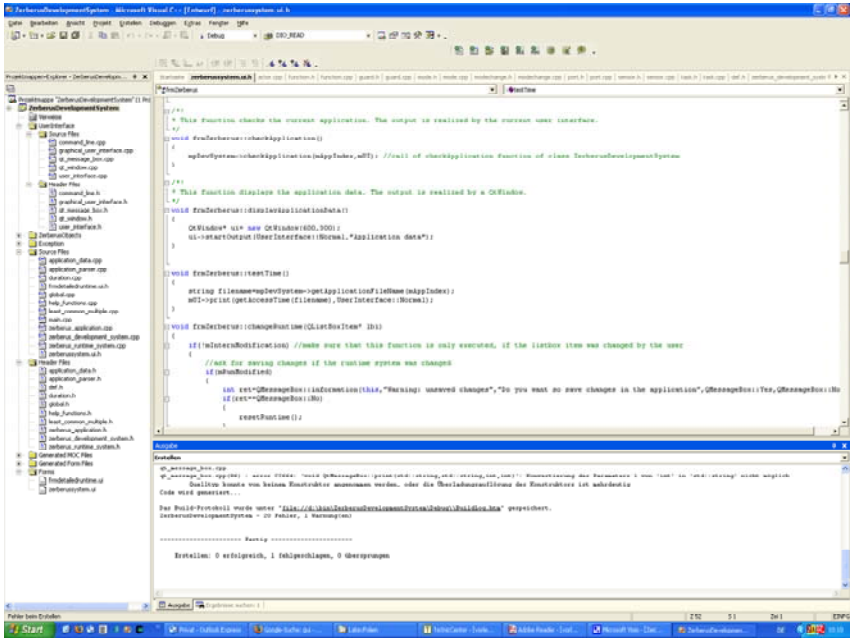
Anwendungsfälle für Nebenläufigkeit (Prozesse)



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. Tastatur) 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

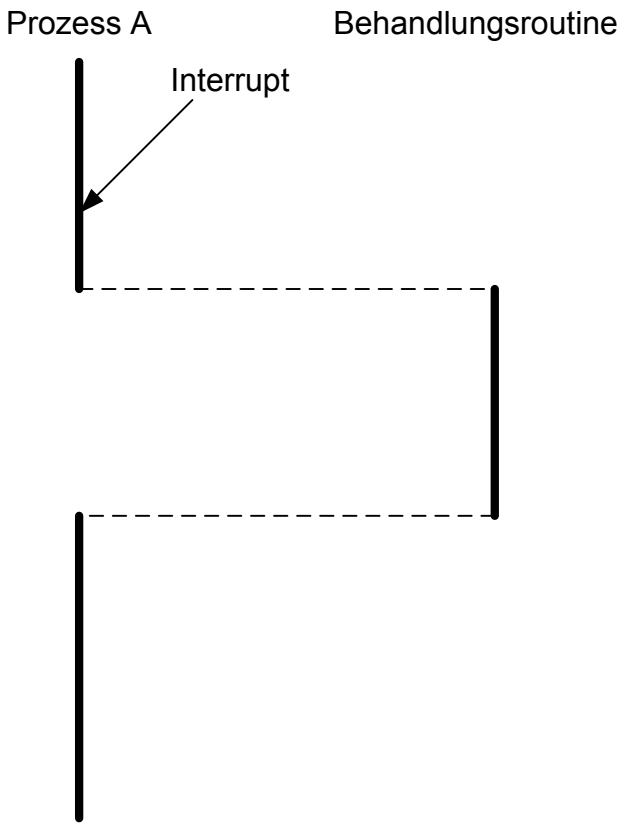
- Durch die Eigenschaft der Asynchronität kann eine deterministische Ausführung nicht gewährleistet werden. Aus diesem Grund kann eine kurzfristige Sperrung von Interrupts nötig sein, um eine konsistente Ausführung der Programme zu erlauben.
- Durch das Sperren werden Interrupts in der Regel nur verzögert,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

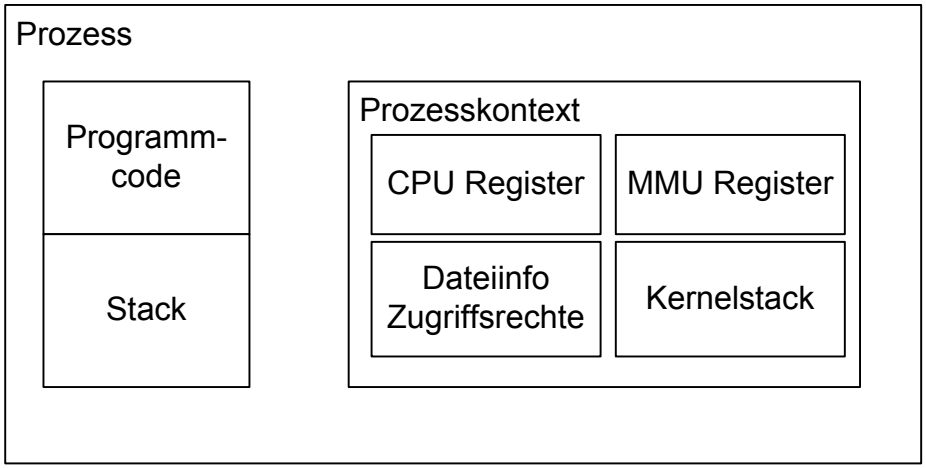
- Die Unterbrechungsvektortabelle enthält einen Verweis auf die entsprechende Unterbrechungsbehandlung für die einzelnen Unterbrechungen
- `INTNO`: Es soll der Hardwareinterrupt IRQ 3 (serielle Schnittstelle) verwendet werden, dieser Interrupt entspricht der Nummer 11 (insgesamt 255 Interrupts (vor allem 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

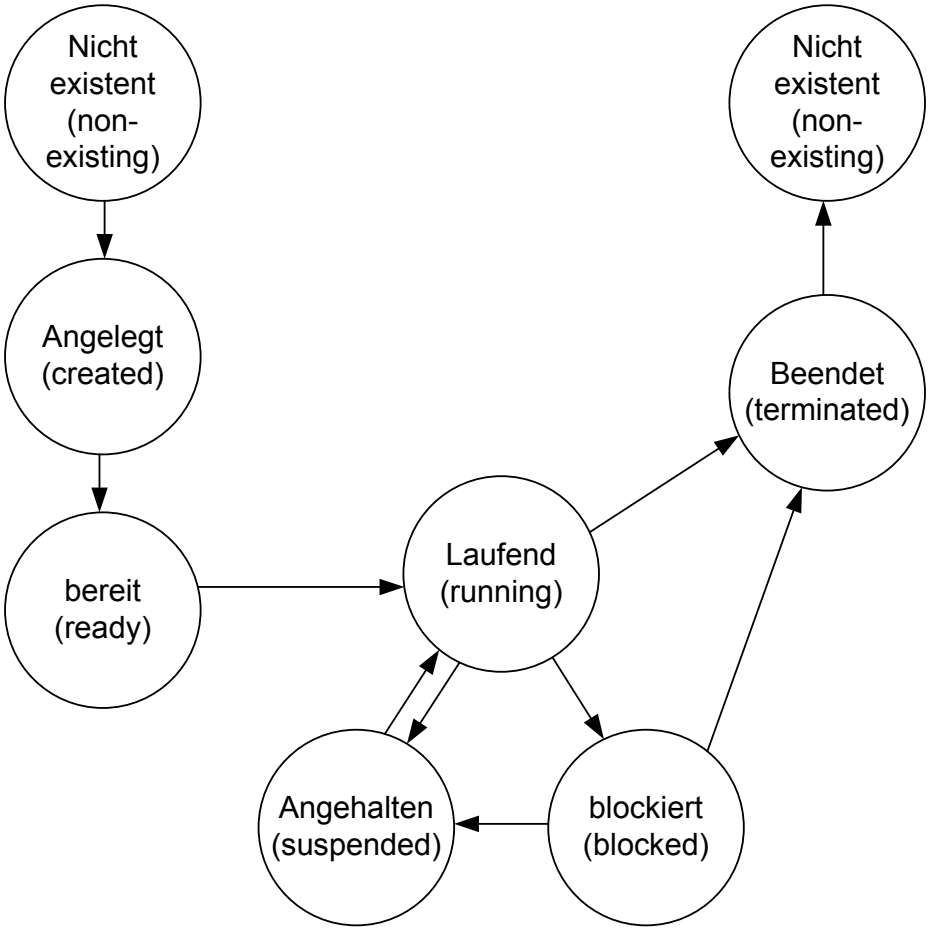
- **Prozess**: Abstraktion eines sich in Ausführung befindlichen Programms
- Die gesamte Zustandsinformation der Betriebsmittel für ein Programm wird als eine Einheit angesehen und als Prozess bezeichnet.
- Prozesse können weitere Prozesse erzeugen ⇨ Vater-,Kinderprozesse.



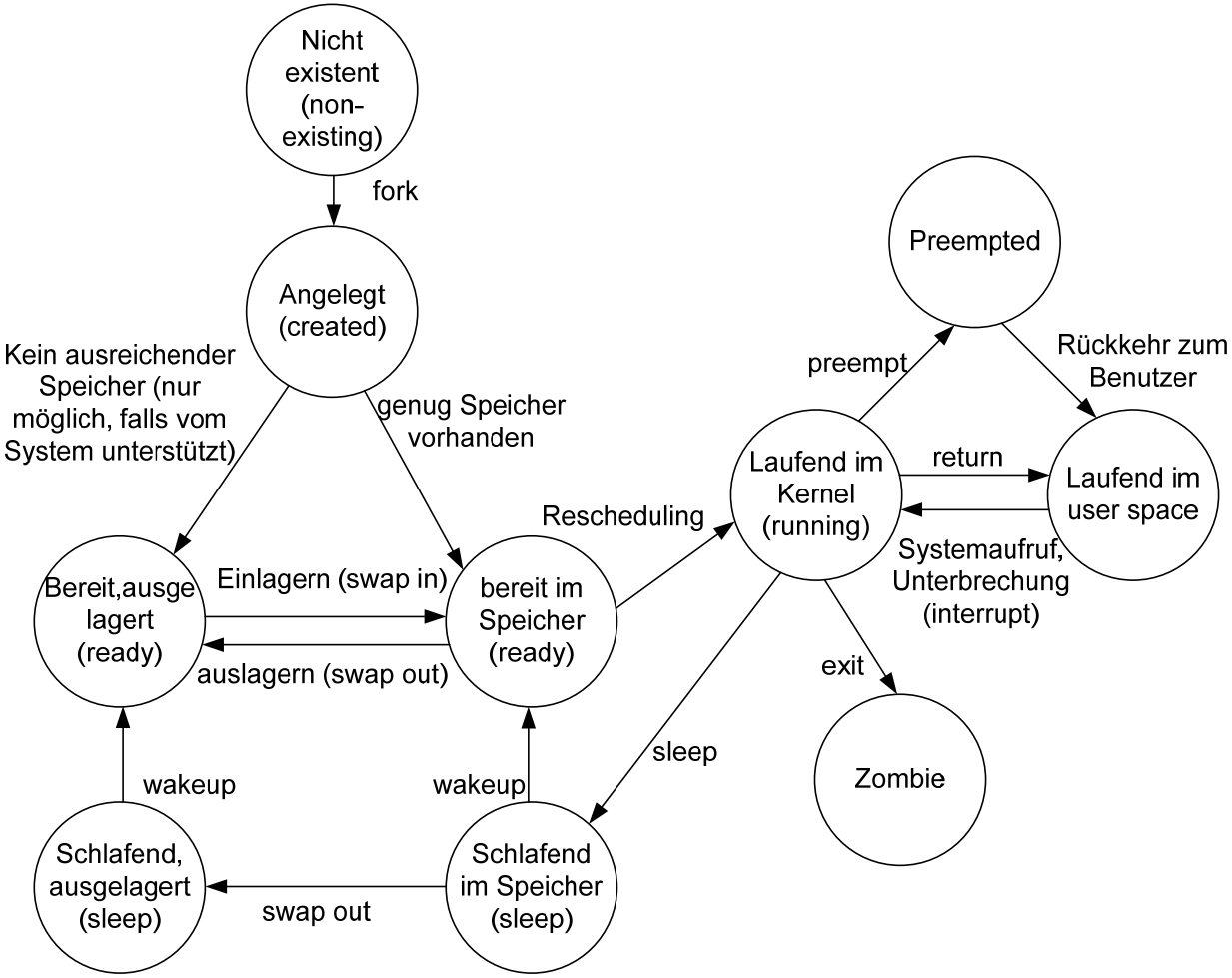
Prozessausführung

- 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

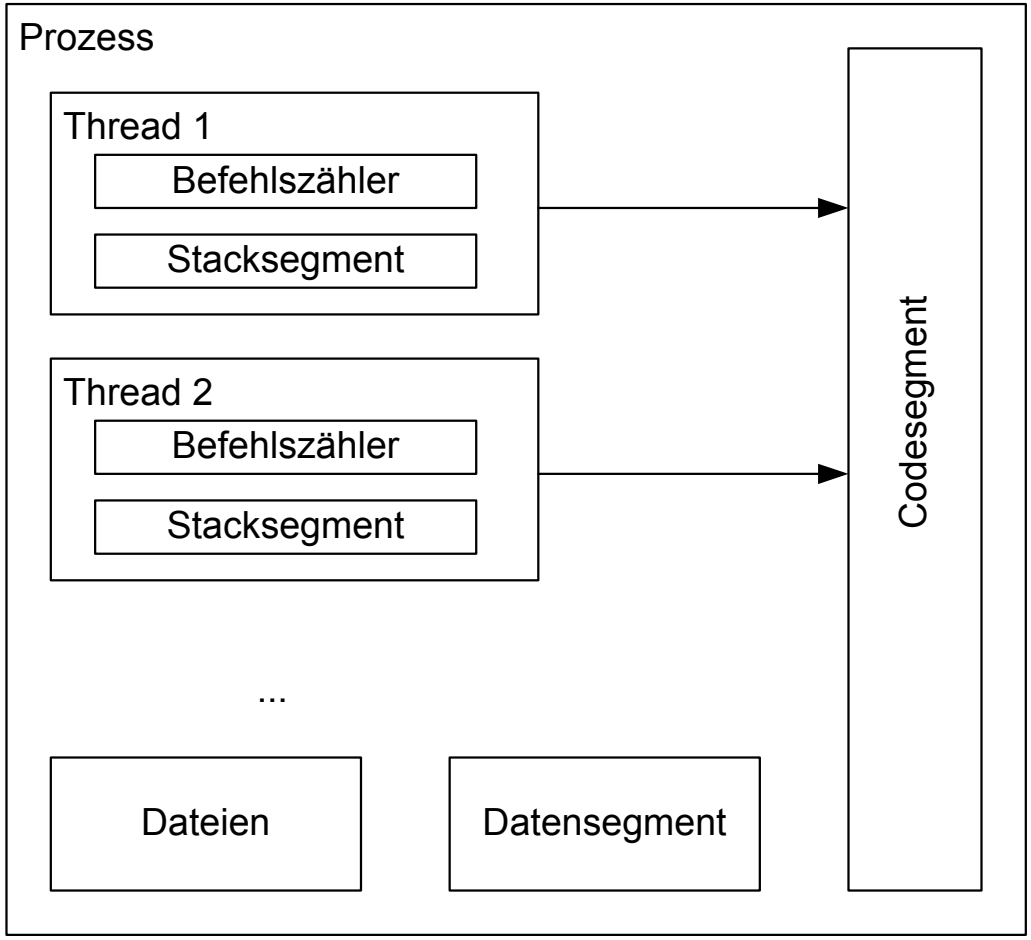
Threads

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==0);//busy waiting
... critical section ...
ready[0]=false;
...
```

Prozess 1

```
...
ready[1]=true;
turn = 0;
while(ready[0]
    && turn==1); //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);
```

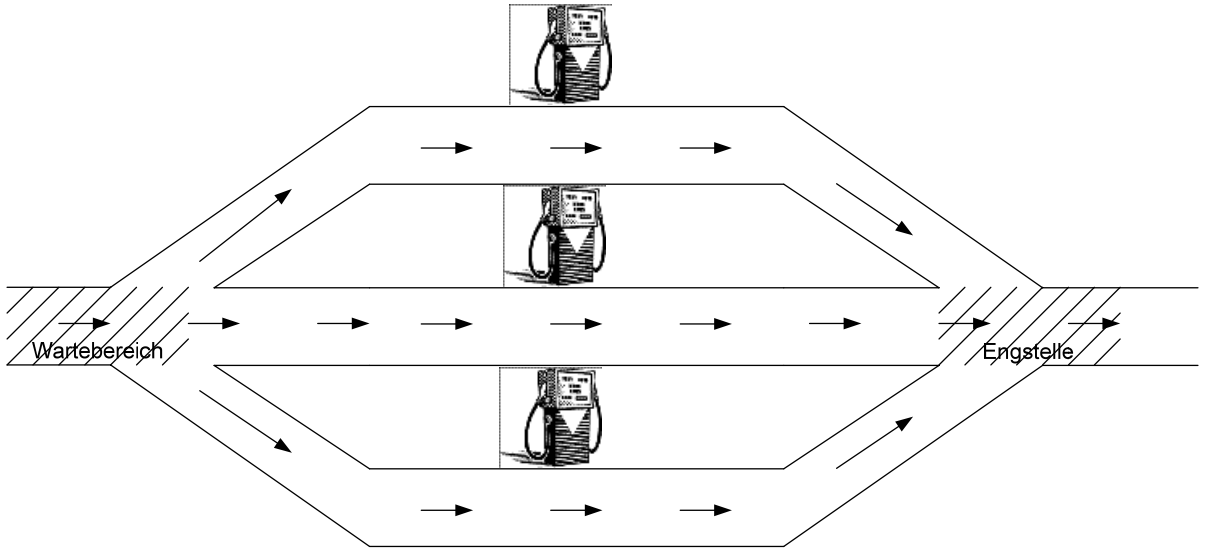
- Jeder Leseprozess führt dann folgenden Code aus:

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

Klausur WS06/07 - Nebenläufigkeit

Prozess: tankendes Auto

```
fahreInWartebereich();

fahreAnZapfsaeule();

tanke();

bezahle();

fahreInEngstelle2();

verlasseEngstelle2();
```



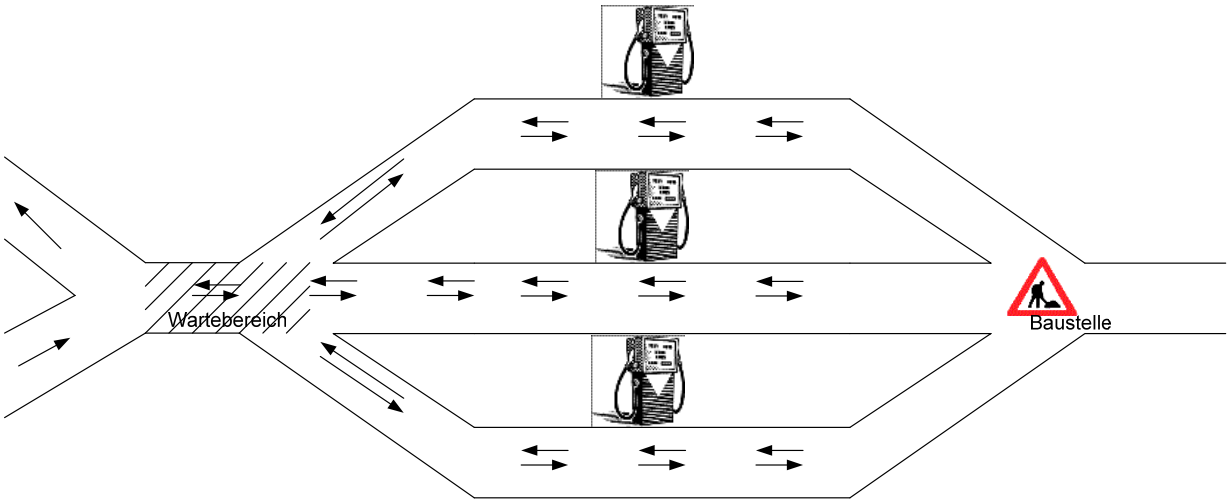
a) Geben Sie die notwendigen Semaphore (mitsamt Initialisierung) an, um das gegebene Problem zu lösen. Beispiel: semAuto(1) würde bedeuten, Sie verwenden einen Semaphor semAuto, der mit 1 initialisiert ist.

b) Ergänzen Sie den folgenden Autoprozess mit passenden up() und down()-Methoden, um Kollisionen zu vermeiden. Achten Sie darauf, dass es zu keiner Verklemmung kommt. **Anmerkung**: Es muss nicht an jeder freien Stelle Code eingefügt werden.
Beispiel: 1: down(semaAuto); up(semaAuto); bedeutet das Einfügen der beiden Operationen in Zeile 1.

Klausur WS06/07 - Nebenläufigkeit

Prozess: tankendes Auto

```
fahreInWartebereich();

fahreAnZapfsaeule();

tanke();

bezahle();

fahreInEngstelle2();

verlasseEngstelle2();
```



c) Aufgrund einer Baustelle ist die Ausfahrt blockiert (siehe Abbildung), so dass die Wartebereich sowohl zur Einfahrt, als auch zur Ausfahrt genutzt werden muss. Ergeben sich notwendige Änderungen im Vergleich zur Lösung der Aufgabe b) und wenn ja welche?

Fragestellung: Invers zählender Semaphor

- Aufgabenstellung: Implementierung des Leser-Schreiber-Problems mit Schreiber-Priorität
- Erläuterung:
  - Auf einen Datensatz können mehrere Leser gleichzeitig oder aber ein Schreiber zugreifen.
  - Sobald ein Schreiber den Schreibwunsch äußert, soll kein weiterer Leser (oder Schreiber) mehr auf den Datensatz zugreifen können. Zum Zeitpunkt der Signalisierung bestehende Lesevorgänge können regulär beendet werden, erst danach darf der Schreiber auf die Daten zugreifen.
- Problem: Häufig wird versucht das Problem mit einem „*invers zählenden Semaphor*“ zu lösen, also einem Semaphor, der bei 0 freigibt und sonst blockiert.
- Wie geht es richtig?

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.
  - 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.
  - Die Signalisierung ist innerhalb des Monitors festgelegt, der Programmierer muss sie nicht selbstständig implementieren.

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.

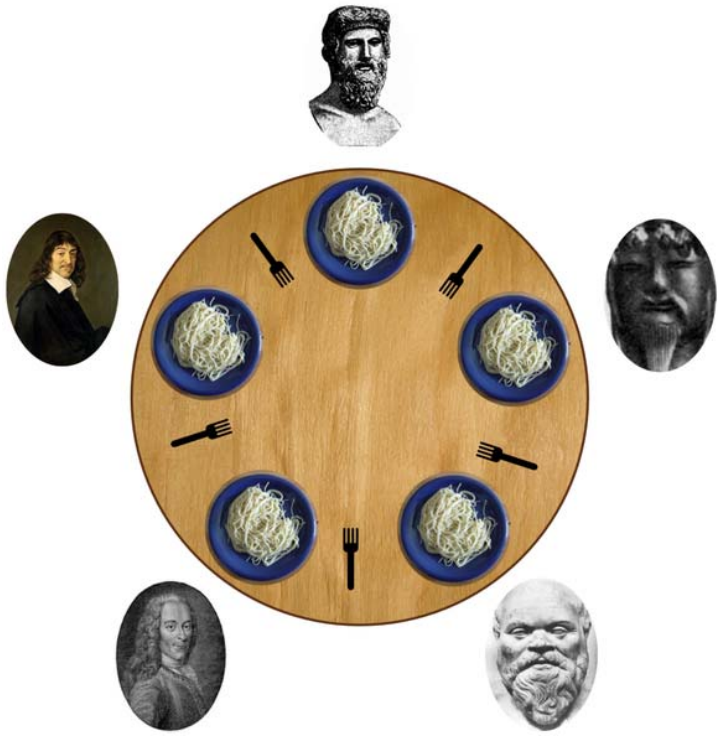
```
public class Semaphore {
    private int value;

    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

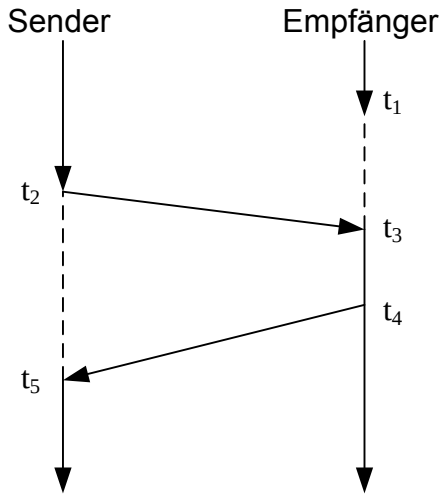
Interprozesskommunikation (IPC)

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

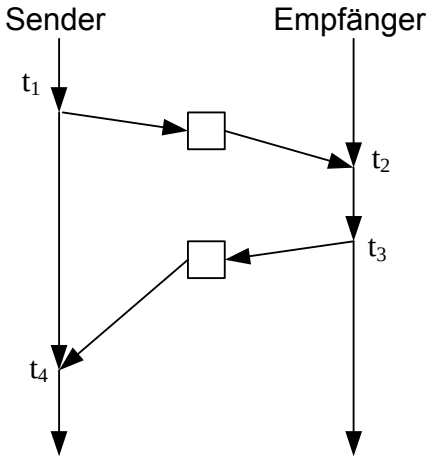
Synchron vs. Asynchron

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)`.

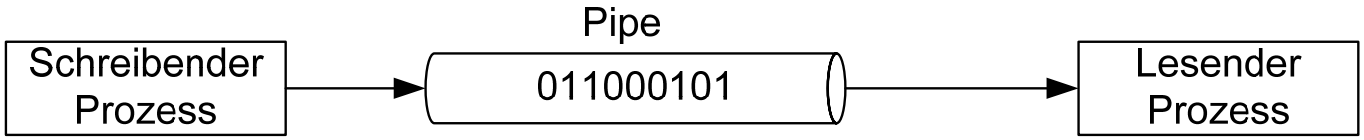
Fragestellungen beim Datenaustausch

- 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

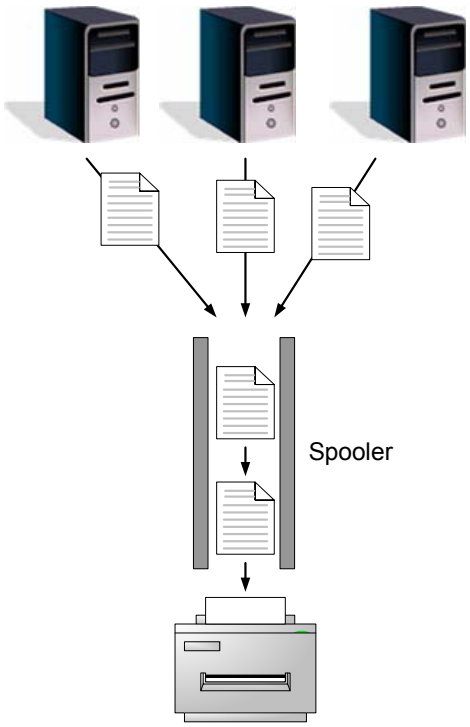
- Pipes bringen einige Nachteile mit sich:
  - Pipes sind nicht nachrichtenorientiert (keine Bündelung der Daten in einzelne Pakete (Nachrichten) möglich).
  - Daten sind nicht priorisierbar.
  - Der für die Pipe notwendige Speicherplatz wird erst während der Benutzung angelegt.
- Wichtig für die Implementierung:
  - 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*/
...
```

** sehr stark vereinfachte Lösung, da zu einem Zeitpunkt nur ein Job verfügbar sein darf*

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

Signalisierung durch Semaphore: Leser-Schreiber-Beispiel

- Vorherige Lösung:

Reader:

```
...
        down(semWriter);
        down(semCounter);
        rcounter++;
        up(semCounter);
        up(semWriter);

        read();

        down(semCounter);
        rcounter--;
        up(semCounter);
...
```

Writer:

```
...
        down(semWriter);

        while(true)
        {
                down(semCounter);
                if(rcounter==0)
                        break;
                up(semCounter);
        }
        up(semCounter);

        write();

        up(semWriter);
...
```

Problem: Busy Waiting

Signalisierung durch Semaphore: Leser-Schreiber-Beispiel

- Lösung mit Signalisierung:

Reader:

```
...
            down(semWriter);
            down(semCounter);
            rcounter++;
            if(rcounter==1)
                        down(semReader);
            up(semCounter);
            up(semWriter);

            read();

            down(semCounter);
            rcounter--;
            if(rcounter==0)
                        up(semReader);
            up(semCounter);
...
```

Writer:

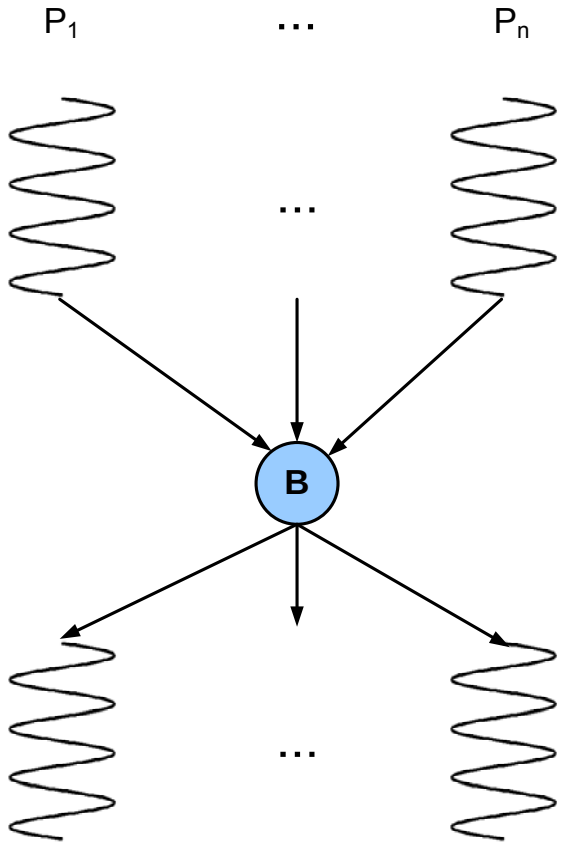
```
...
            down(semWriter);
            down(semReader);
            up(semReader);

            write();


            up(semWriter);
...
```

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
```

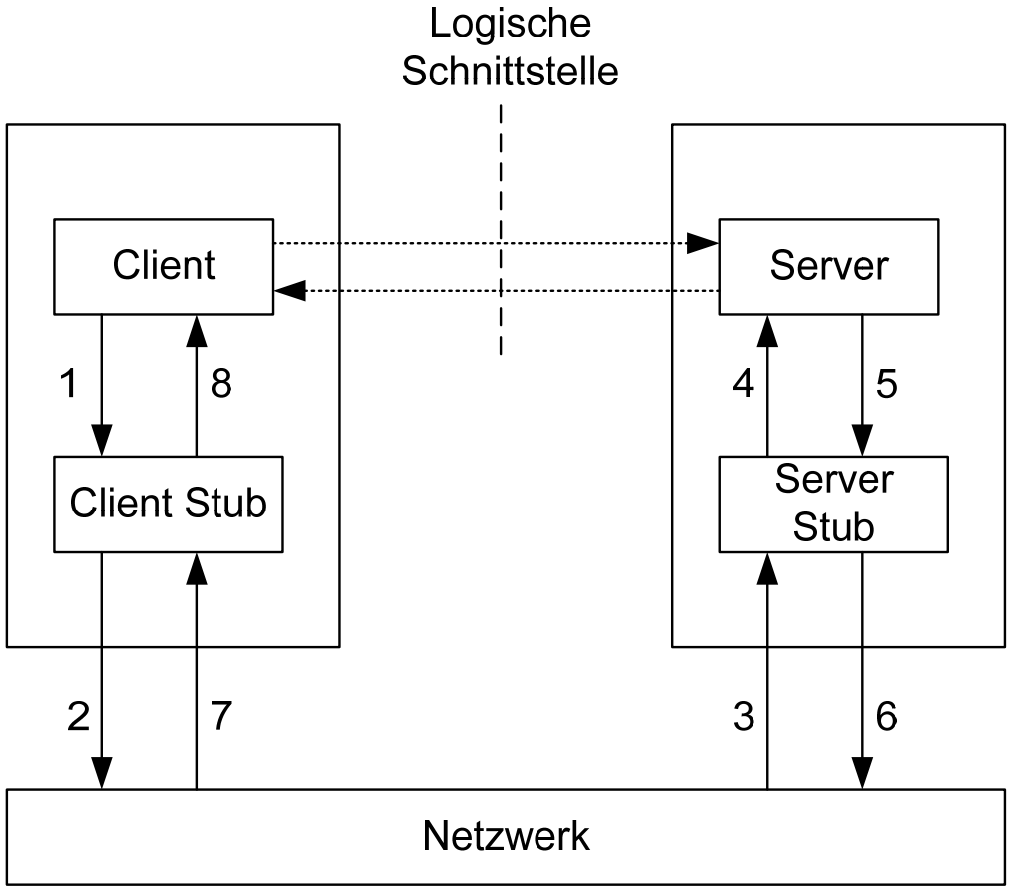
```
ALT
    x<10 & chan1 ? y
        screen ! y
    x<20 & chan2 ? y
        screen ! y
```

*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

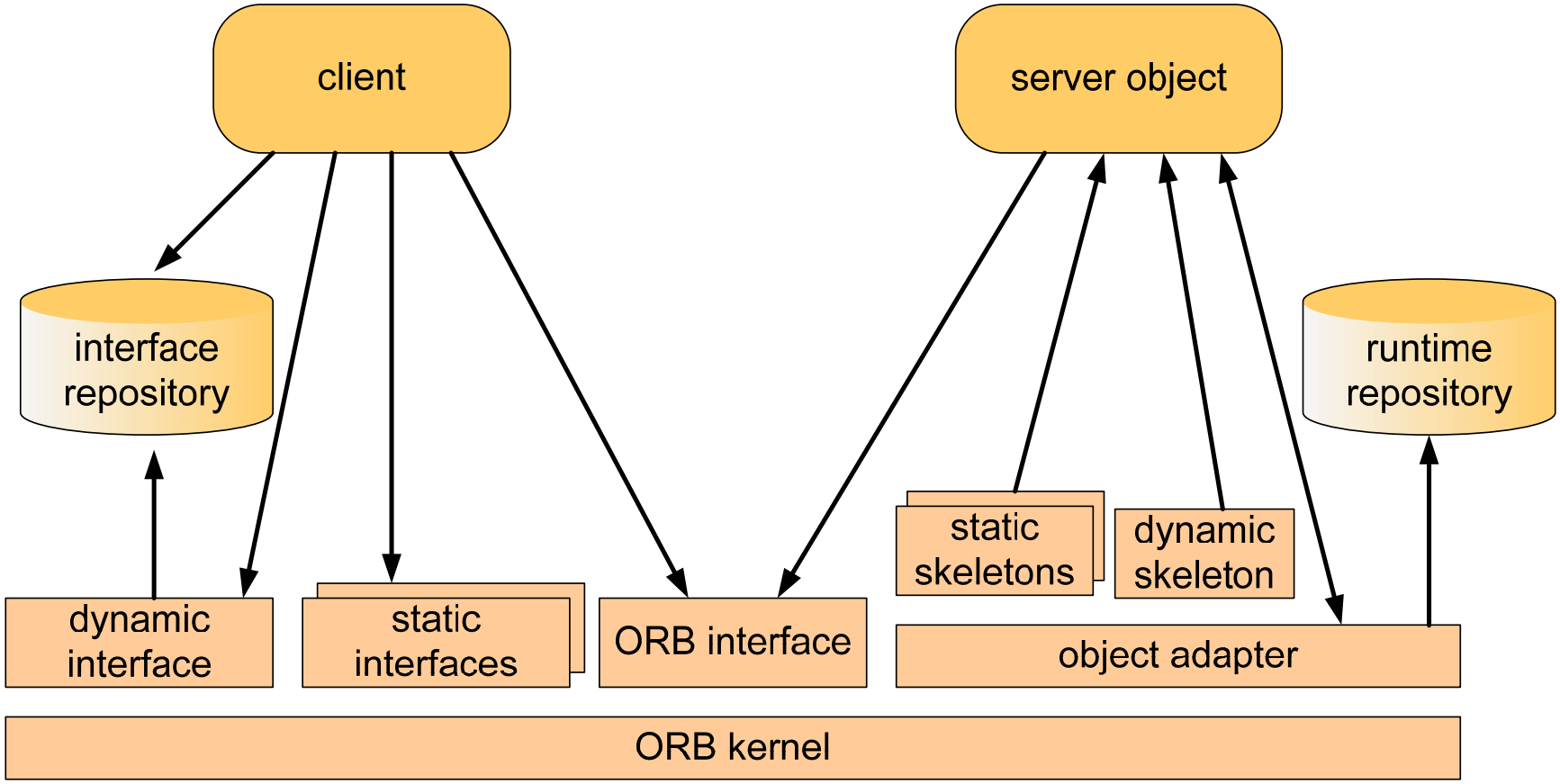
Remote Procedure Call (RPC)



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 & Wiederholung

Zusammenfassung

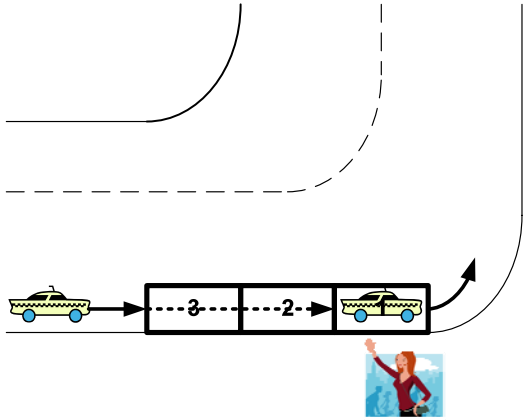
- Folgende Fragen wurden in dieser Vorlesung erklärt und sollten nun verstanden sein:
  - Was ist Nebenläufigkeit / Parallelität?
  - Mit welchen Techniken kann man Nebenläufigkeit erreichen und wann wird welche Technik angewendet?
  - Wie können **race conditions** vermieden werden?
  - Welche Arten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

Klausurfragen - Nebenläufigkeit

- Klausur WS 06/07
  - Nennen Sie zwei Gründe, wieso nebenläufige Programmierung häufig in Echtzeitsystemen angewandt wird.
  - Was sind Race Conditions?
- Wiederholungsklausur WS 06/07:
  - In der Vorlesung haben Sie die Konzepte Nachrichtenwarteschlangen, Semaphor, sowie Barrieren kennengelernt.
    a) Beschreiben Sie, wie man mit Hilfe von Semaphoren Nachrichtenwarteschlangen implementieren kann.
    b) Beschreiben Sie, wie man mit Hilfe von Nachrichtenwarteschlangen Semaphoren implementieren kann.
    c) Beschreiben Sie, wie man mit Hilfe von Semaphoren Barrieren implementieren kann.
- Klausur WS 07/08
  - **Annahme**: Für folgende Aufgaben können Sie davon ausgehen, dass maximal ein Signal pro Runde an den Automaten geschickt wird. Die Prozesse, die die modellierten Komponenten benutzen, müssen Sie nicht modellieren. Vermeiden Sie Busy Waiting.
  - Modellieren Sie einen Automaten, der das Rendezvouskonzept umsetzt.
  - Modellieren Sie eine Komponente, die das Leser-Schreiber-Problem für 1 Schreiber und beliebig viele Leser mit Schreiberpriorität umsetzt.

Klausur WS07/08 - Nebenläufigkeit

- Gegeben Sie folgendes Szenario: am Münchner Odeonsplatz gibt es eine Wartebucht für Taxis. Zur Vereinfachung gehen wir davon aus, dass die Wartebucht aus drei Plätzen besteht und immer nur ein Passagier gleichzeitig auf ein Taxi wartet. Passagiere steigen an der ersten Wartebucht ein, die Taxis rücken nach, sobald das Taxi vor ihnen losgefahren ist. Implementieren Sie nun schrittweise eine Prozesssynchronisation, so dass es zu keinen Auffahrunfällen kommt, die Taxis in der Ankunftsreihenfolge auch wieder losfahren, Taxis nur mit Passagier losfahren, Passagiere nicht aus Versehen ein nicht-existentes Taxi betreten und es zu keinen Verklemmungen kommt.



a) Notieren Sie die wichtigen Programmabschnitte des Taxiprozesses und des Passagierprozesses. Lassen Sie genügend Platz für spätere Synchronisationsoperationen.
Beispiel: `fahreInErsteWartebucht();`

b) Geben Sie die zur Synchronisation der Taxis und Passagiere benötigten Semaphore, sowie der Initialwerte an. Gehen Sie dabei davon aus, dass zu Beginn kein Taxi in der Wartebucht und keine wartenden Passagiere vorhanden sind.
Beispiel: `semTaxi(1)` würde bedeuten, Sie verwenden einen Semaphor semTaxi, der mit 1 initialisiert ist.
`int i=0;` wenn sie eine ganzzahlige Variable mit Initialisierungswert 1 benutzen wollen.

c) Ergänzen Sie den Taxiprozess und Passagierprozess mit passenden up() und down()-Methoden, um die Aufgabenstellung zu erfüllen.
Beispiel: `down(semTaxi);` bedeutet das Anfordern des Semaphors semTaxi
Beispiel: `up(semTaxi);` bedeutet das Freigeben des Semaphors semTaxi

d) Der Wartebereich am Odeonsplatz ist begrenzt. Stellen Sie sicher, dass maximal 3 Taxis auf Fahrgäste warten und kein Rückstau entsteht. Die Überprüfung ob der Wartebereich belegt ist, soll dabei so schnell wie möglich erfolgen um den Straßenverkehr nicht zu behindern. Andererseits, sollen die Taxifahrer auf jeden Fall in den letzten Warteplatz fahren, falls dieser frei ist.

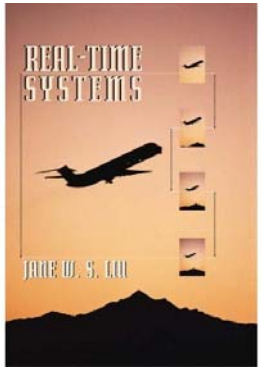
Kapitel 4

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

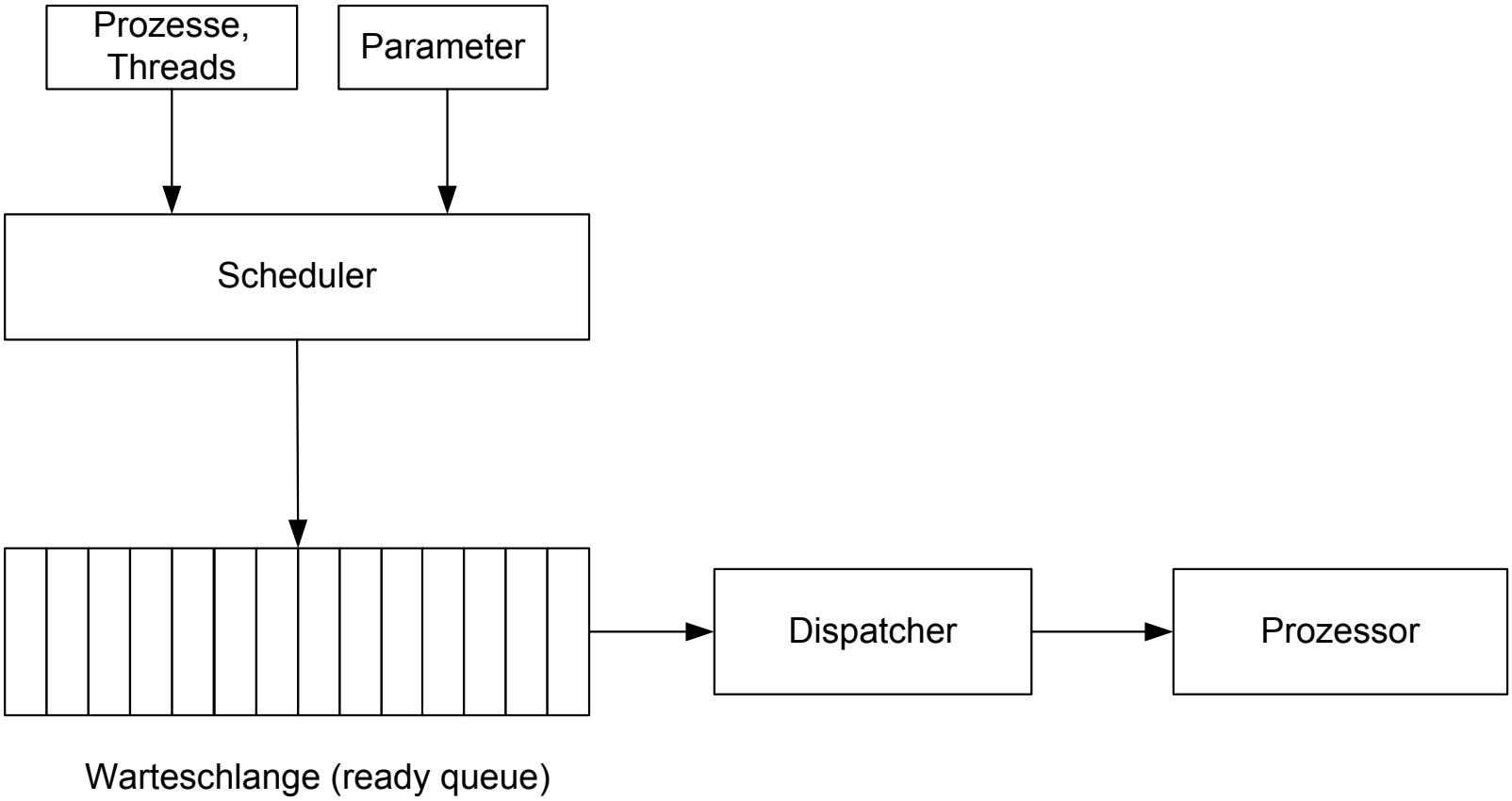
Fridolin Hofmann: Betriebssysteme - Grundkonzepte und Modellvorstellungen, 1991

- 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

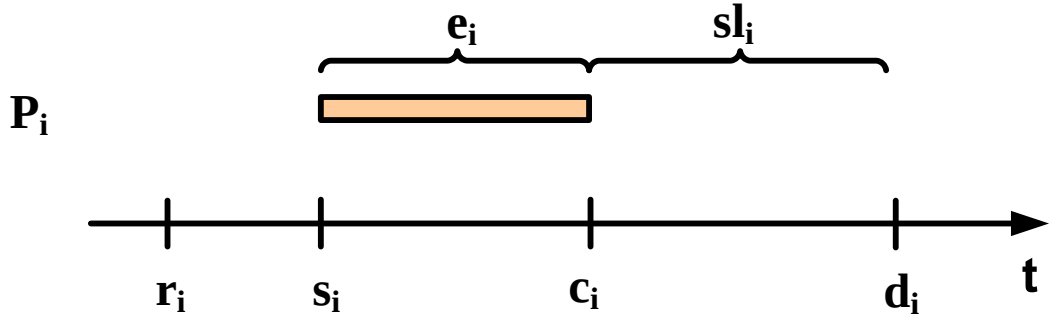


Scheduler und Dispatcher

- **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

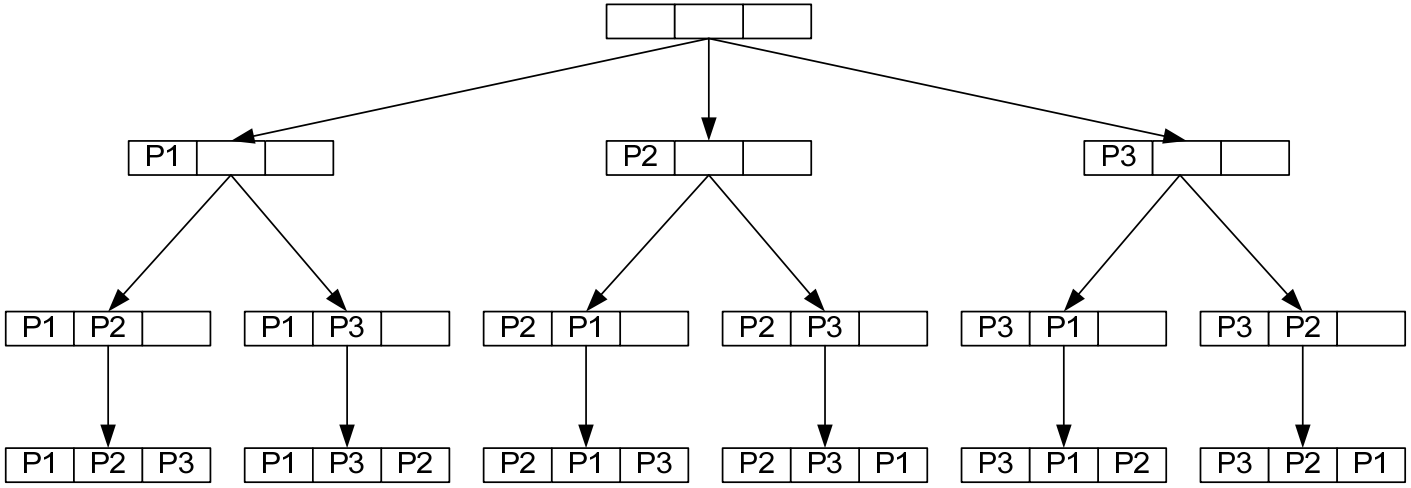
- Planen aperiodischer Prozesse
  - Planen durch Suchen
  - Planen nach Fristen
  - Planen nach Spielräumen
- Planen periodischer Prozesse
  - Planen nach Fristen
  - Planen nach Raten
- Planen abhängiger Prozesse

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**.

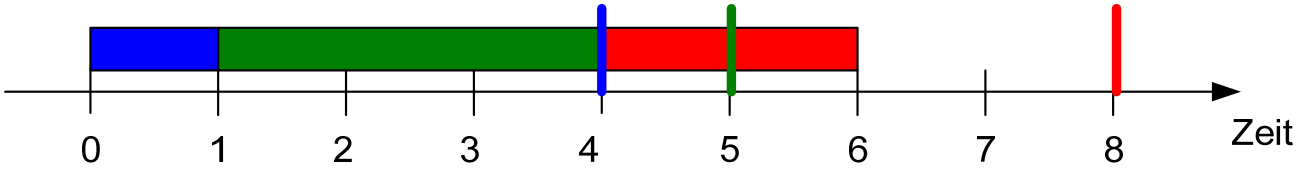
Scheduling-Strategien (online, nicht-präemptiv) für Einprozessorsysteme

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.

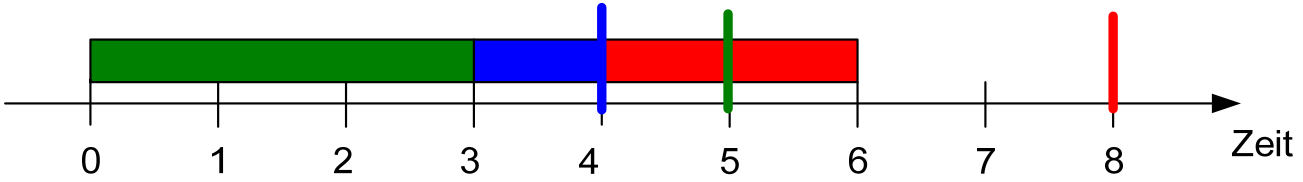
- Vorteil und Nachteile:
  - LST erkennt Fristverletzungen früher als EDF.
  - Für LST müssen die Ausführungszeiten der Prozesse bekannt sein.

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;



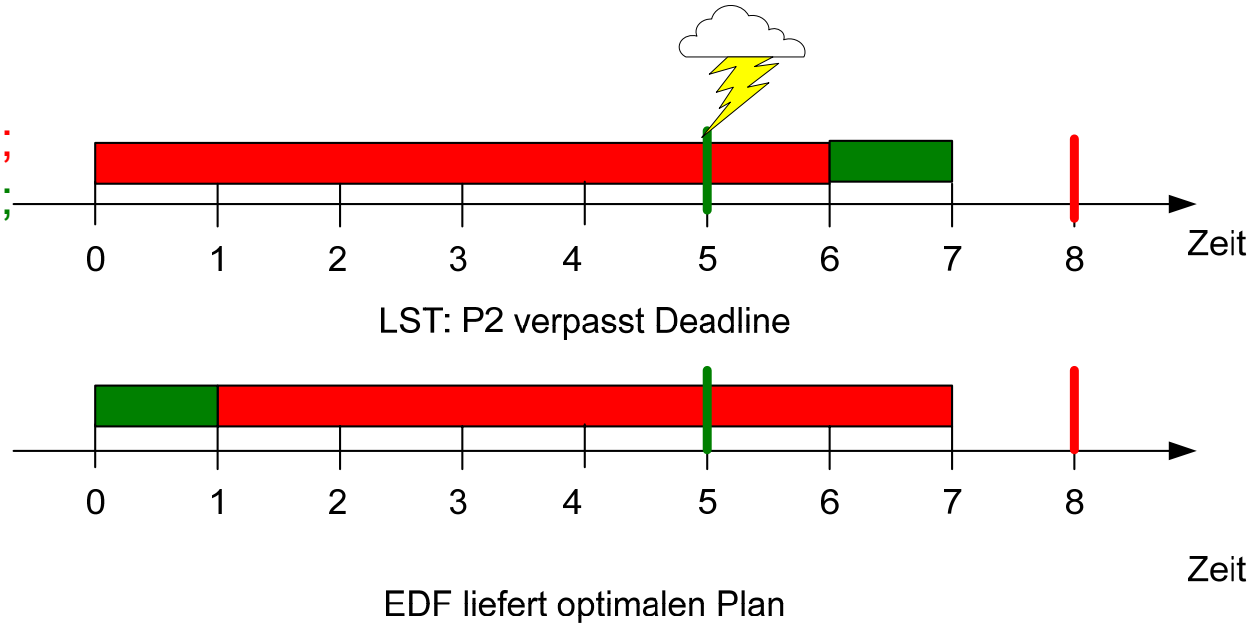
Earliest Deadline First



Least Slack Time

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

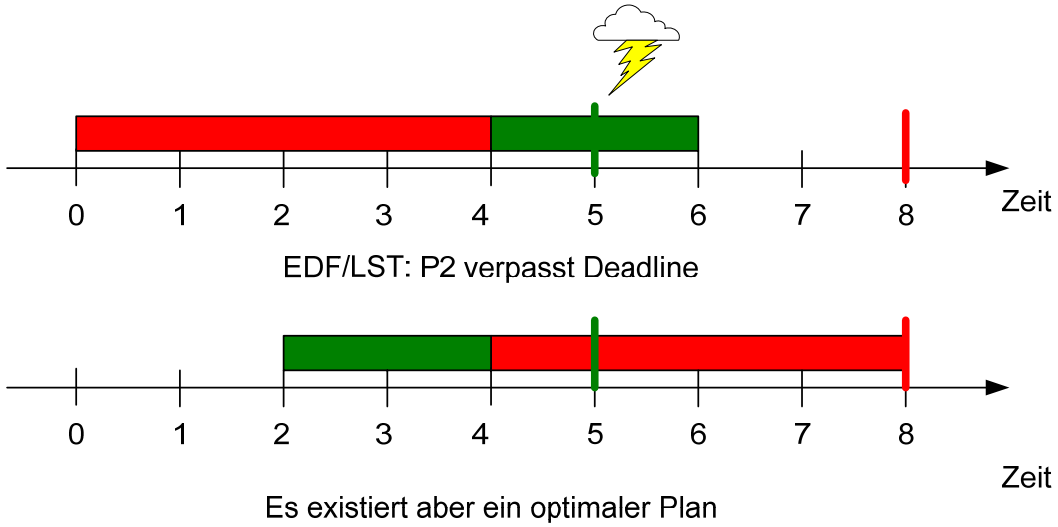
- Unter der Voraussetzung, dass alle Prozesse P_i eine Bereitzeit $r_i=0$ besitzen und das ausführende System ein Einprozessorsystem ist, ist EDF optimal, d.h. ein zulässiger Plan wird gefunden, falls ein solcher existiert.
- Beweisidee für EDF: Tausch in existierendem Plan
  - Sie $Plan_x$ ein zulässiger 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)

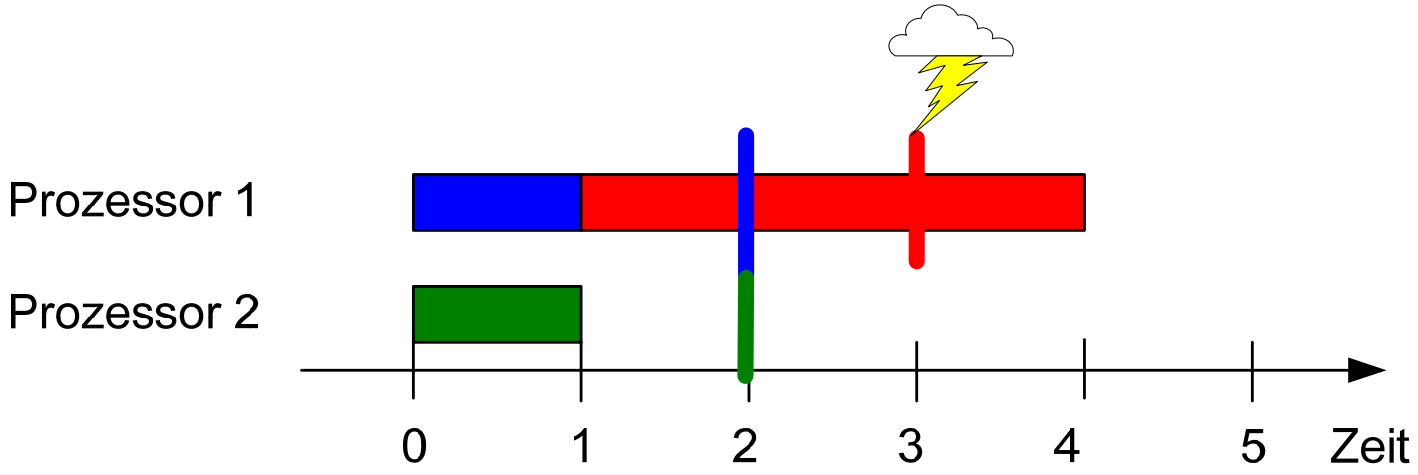
Zeitplanung auf Mehrprozessorsystemen

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.

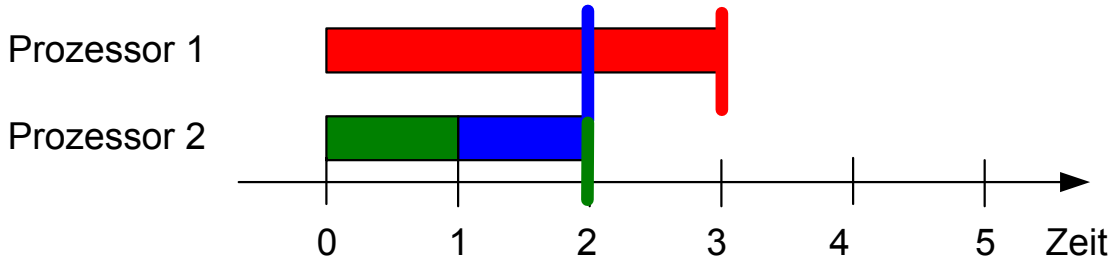
Beispiel: Versagen von EDF

- 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;

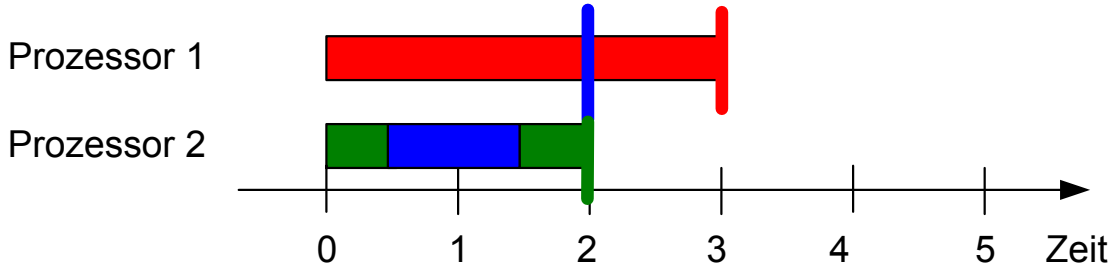


EDF-Verfahren: Deadline d1 wird verpasst

Beispiel: Optimaler Plan und LST-Verfahren



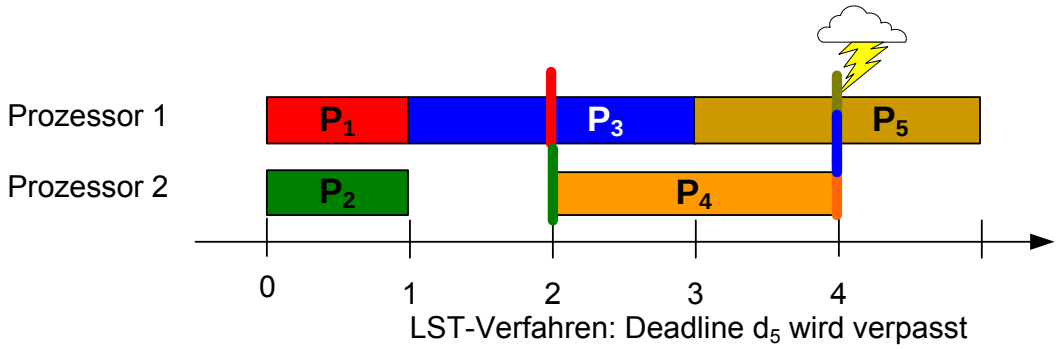
Optimaler Plan



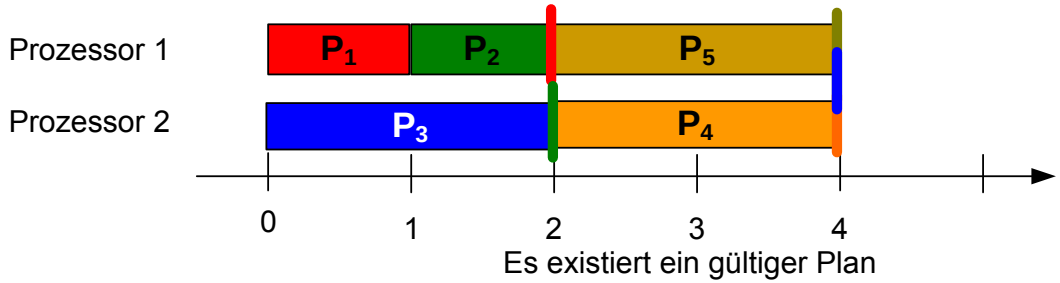
LST-Verfahren mit $\Delta_t = 0.5$

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) ⇒ 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;

  $\Rightarrow$ Prozess P_1 (kein Spielraum) muss sofort auf CPU1 ausgeführt werden.

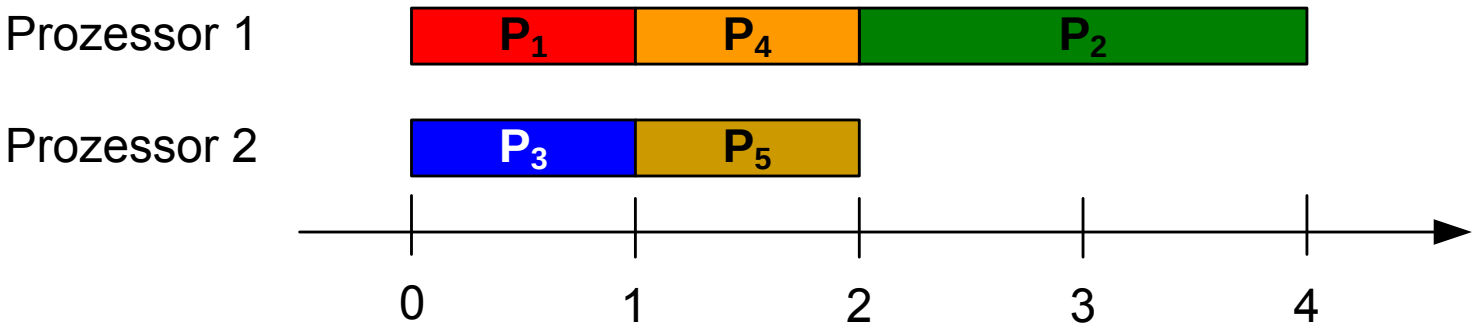
  $\Rightarrow$ Es gibt je nach Strategie zwei Fälle zu betrachten: P2 oder P3 wird zunächst auf CPU2 ausgeführt.

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:

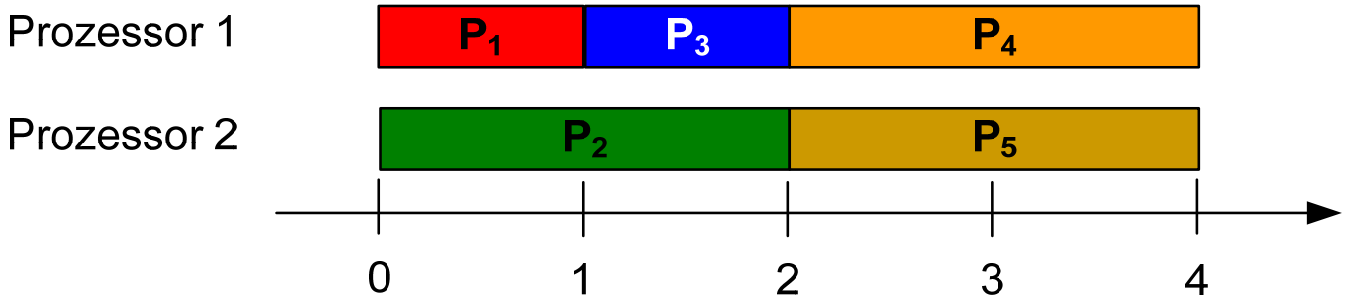


2. Fall

- 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:



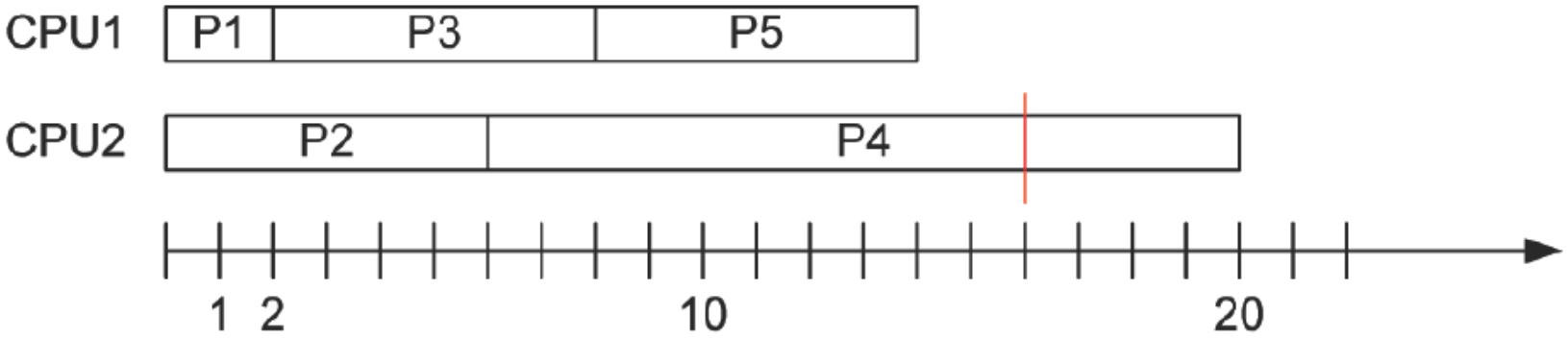
Strategien in der Praxis

- 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.

Klausur SS 07 - Szenario



Startzeiten s: s(P1)=0; s(P2)=0; s(P3)=0; s(P4)=0; s(P5)=0;
Ausführungszeiten e: e(P1)=2; e(P2)=6; e(P3)=6; e(P4)=14; e(P5)=6;
Deadlines e: e(P1)=4; e(P2)=8; e(P3)=12; e(P4)=16; e(P5)=18;

a) Welches Schedulingverfahren wurde verwendet? Welche Änderungen würden sich ergeben, wenn das Verfahren präemptiv wäre?
b) Welche optimalen Schedulingverfahren existieren für Mehrprozessorsysteme?
c) Welche Voraussetzungen müssen für ein optimales Schedulingverfahren in Mehrprozessorsystemen erfüllt sein?
d) Zeichnen Sie einen unter Zuhilfenahme eines optimalen Schedulingplanes einen korrekten Ausführungsplan.
e) In der Praxis werden diese Schedulingverfahren nicht angewandt. Was spricht dagegen und welcher Ansatz wird stattdessen gewählt?

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.

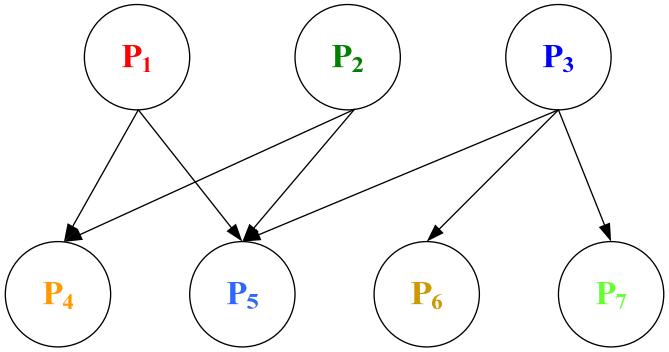
  $\Rightarrow$ Ein Präzedenzsystem ist nur dann planbar, falls das zugehörige normalisierte Präzedenzsystem planbar ist.

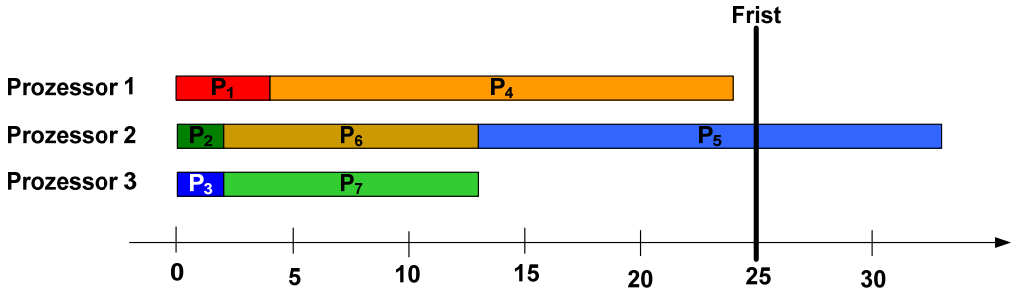
Anomalien bei nicht präemptiven Scheduling

- 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.

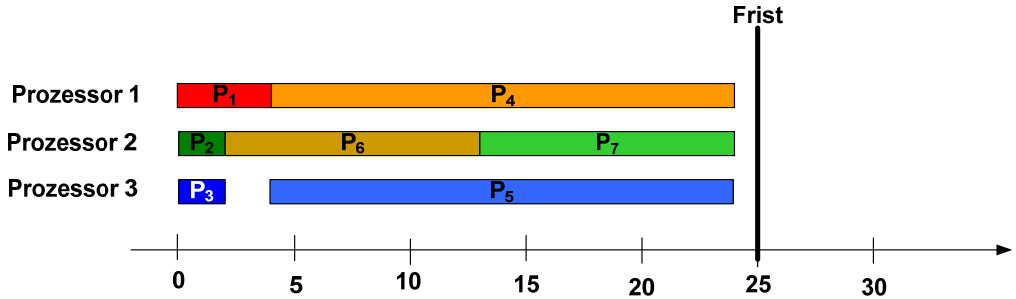
Beispiel: Verkürzung durch freiwilliges Warten

- 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

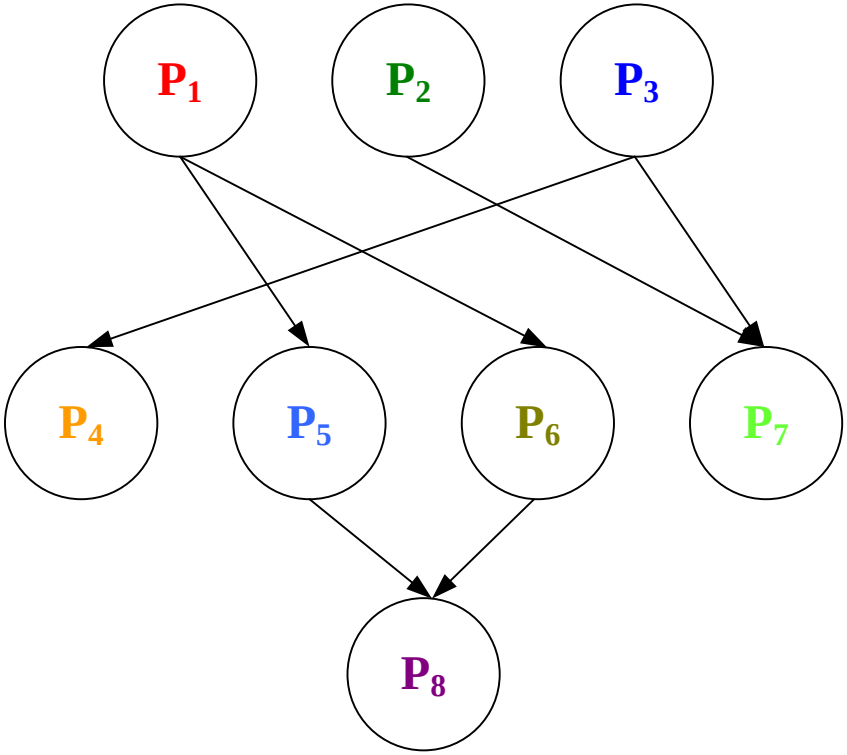


Optimaler Plan

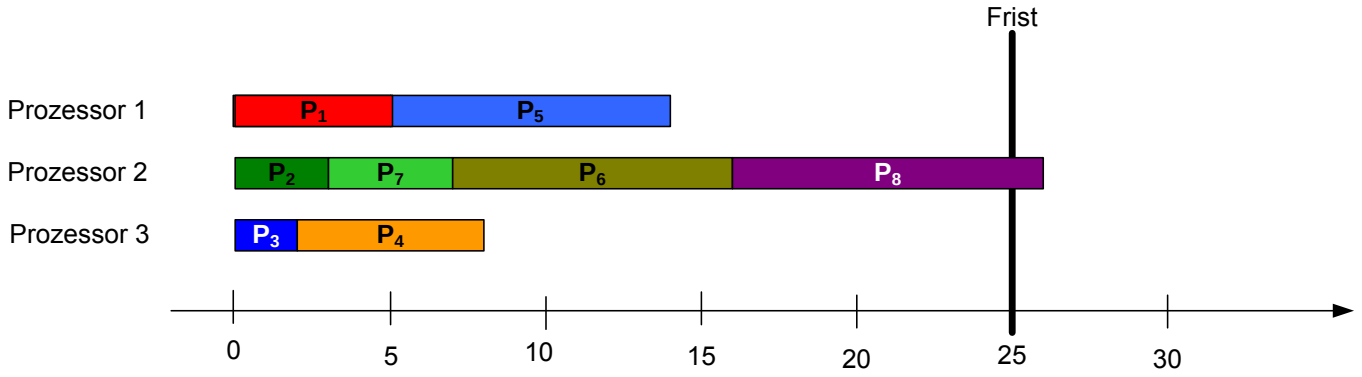
Beispiel: Laufzeitverlängerung durch zusätzlichen Prozessor II

- 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: $d_i=25$

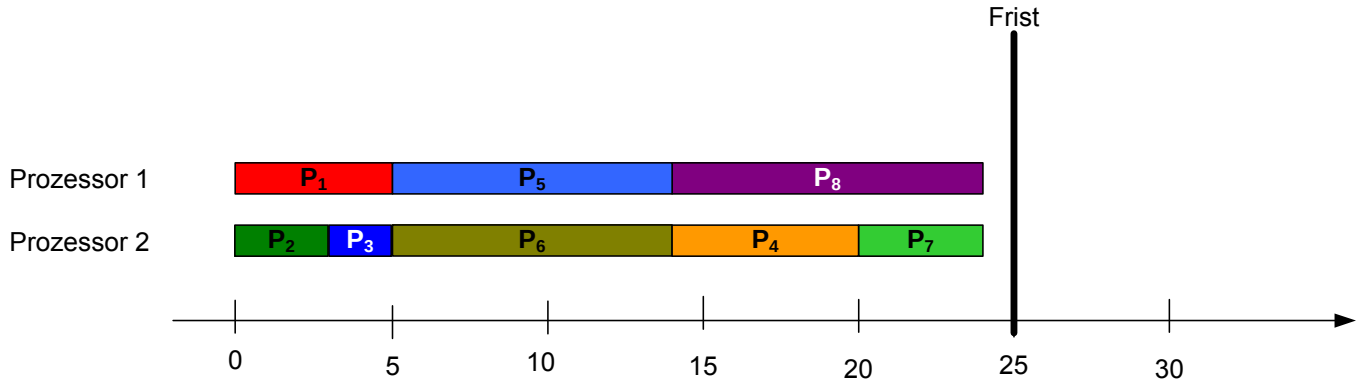
- Präzedenzgraph:



Beispiel: Laufzeitverlängerung durch zusätzlichen Prozessor II



Prioritätenbasiertes Scheduling (LST) auf 3 Prozessoren



Prioritätenbasiertes Scheduling (LST) auf 2 Prozessoren

Scheduling

Problem: Prioritätsinversion

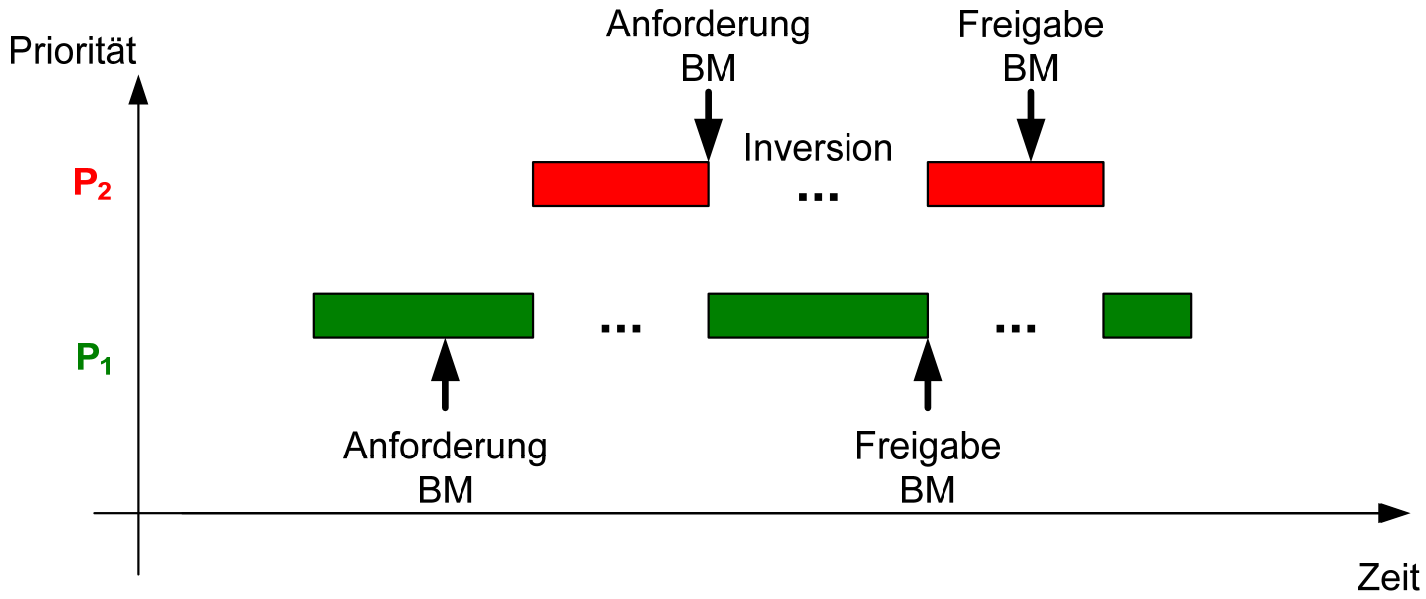
Motivation des Problems

- 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.

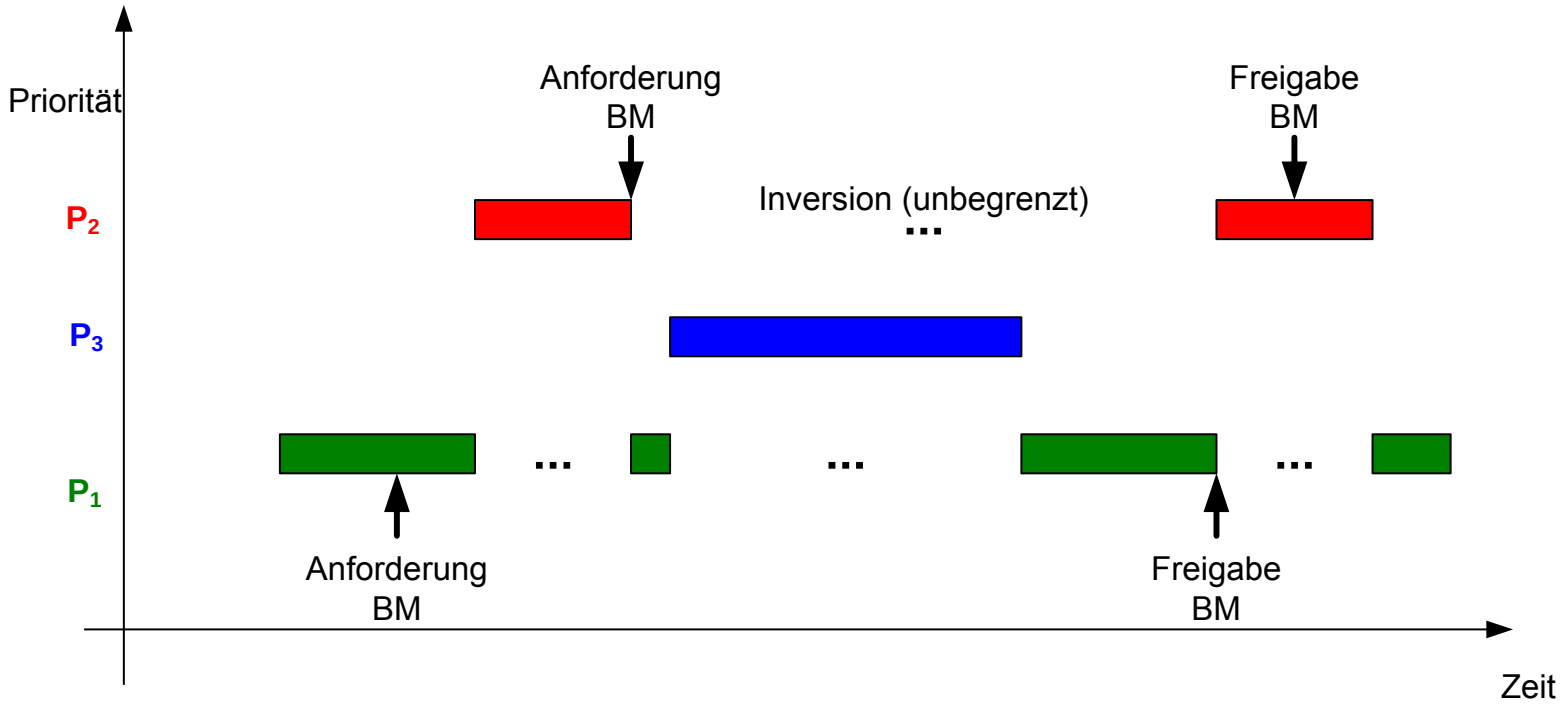
Prioritätsinversion

- **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.

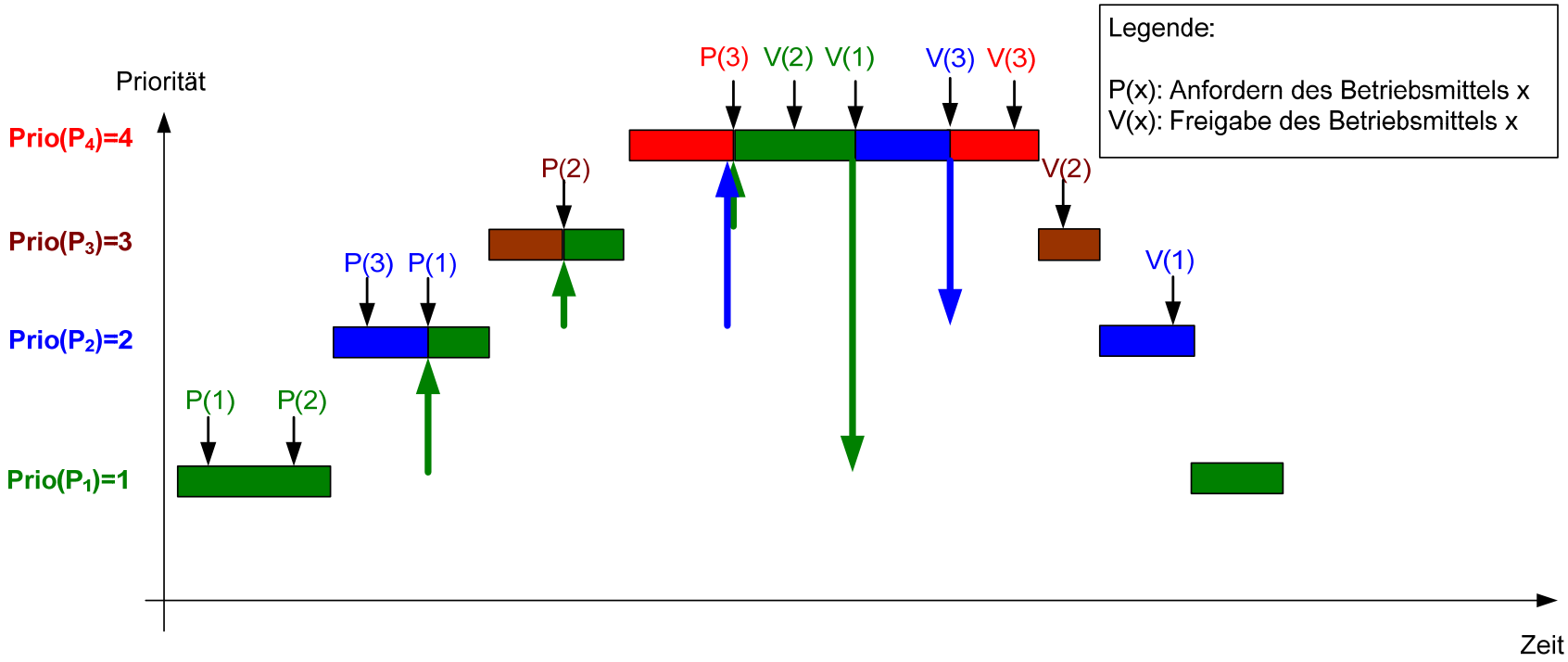
  ⇒ Unbegrenzte Prioritätsinversion wird verhindert.

  ⇒ 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)

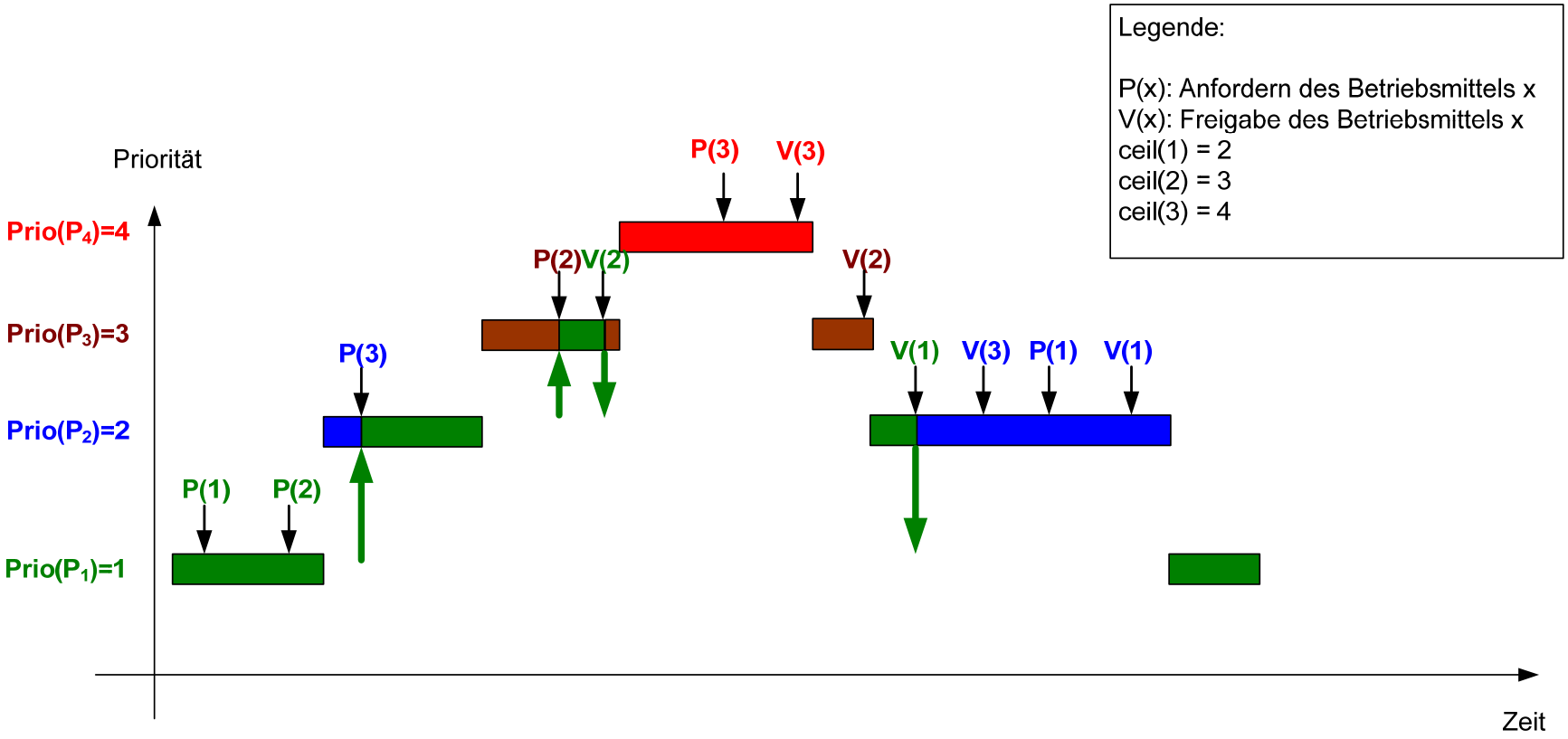
- Jedem Betriebsmittel (z.B. Semaphor) s wird eine Prioritätsgrenze ceil(s) zugewiesen, diese entspricht der maximalen Priorität der Prozesse, die auf s zugreifen.
  - Ein Prozess p darf ein BM nur blockieren, wenn er von keinem anderen Prozess, der andere BM besitzt, verzögert werden kann.
  - Die aktuelle 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

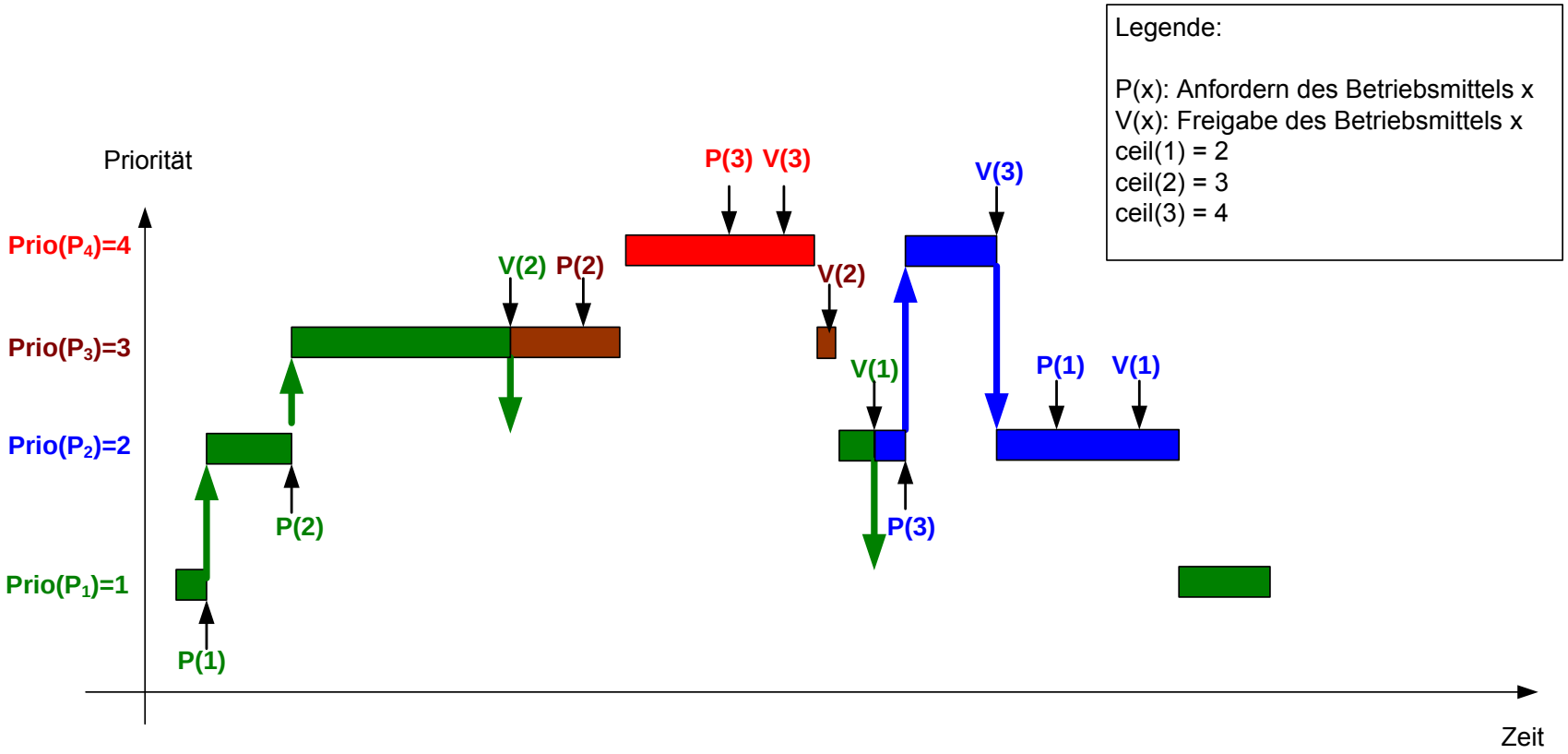
  $\Rightarrow$ Verhindert Verklemmungen

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

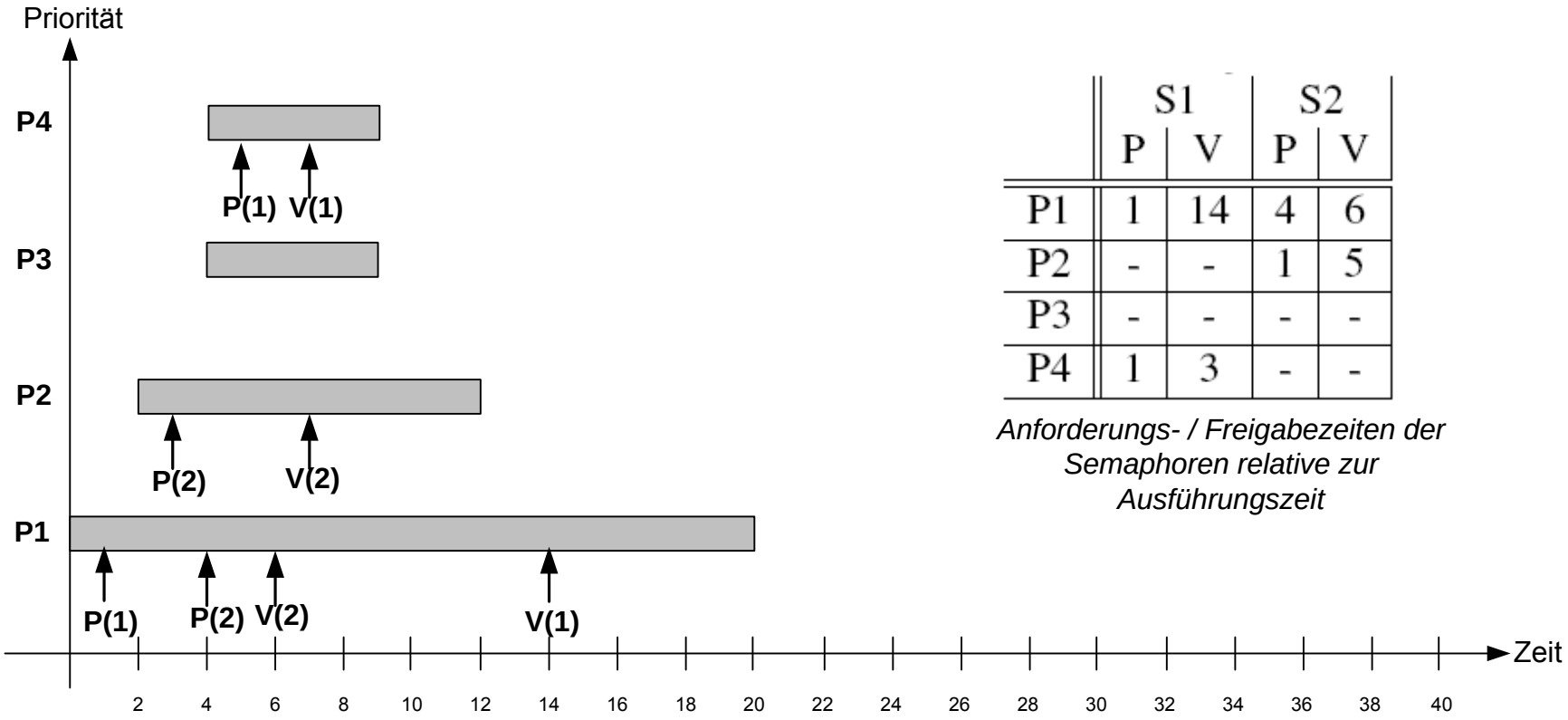
Beispiel: Prioritätsobergrenzen



Beispiel: Immediate Priority Ceiling



Klausur WS 06/07 - Szenario



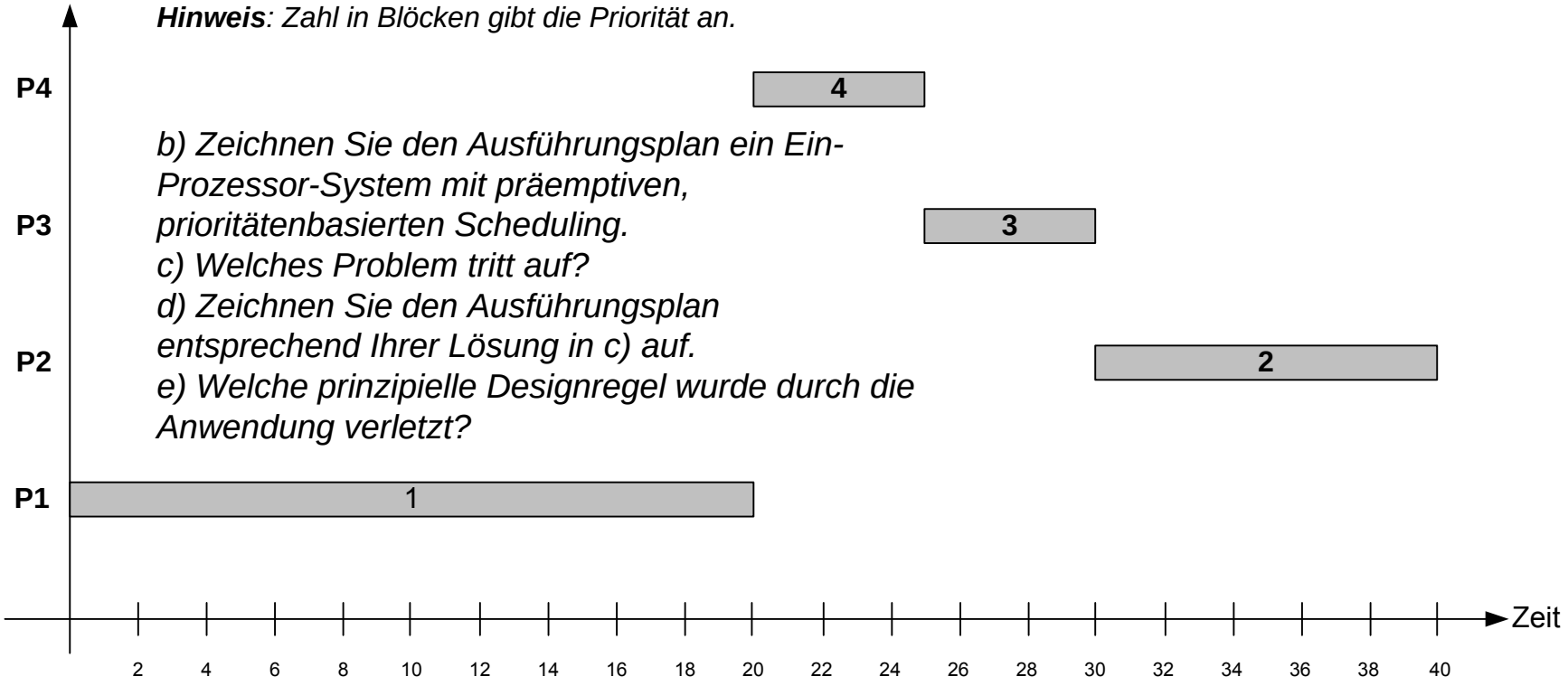
	S1		S2	
	P	V	P	V
P1	1	14	4	6
P2	-	-	1	5
P3	-	-	-	-
P4	1	3	-	-

Anforderungs- / Freigabezeiten der Semaphoren relative zur Ausführungszeit

Fortsetzung – Möglicher Ausführungsplan für ein 1-Prozessor-System

a) Welches Verfahren wird hier angewandt?
***Hinweis**: Zahl in Blöcken gibt die Priorität an.*

b) Zeichnen Sie den Ausführungsplan ein Ein-Prozessor-System mit präemptiven, prioritätenbasierten Scheduling.
c) Welches Problem tritt auf?
d) Zeichnen Sie den Ausführungsplan entsprechend Ihrer Lösung in c) auf.
e) Welche prinzipielle Designregel wurde durch die Anwendung verletzt?

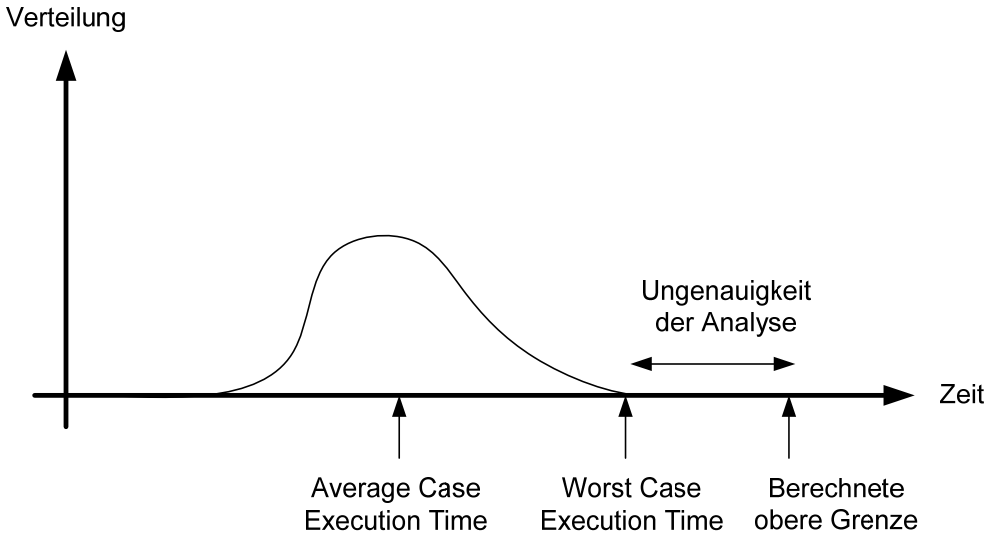


Scheduling

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

Scheduling

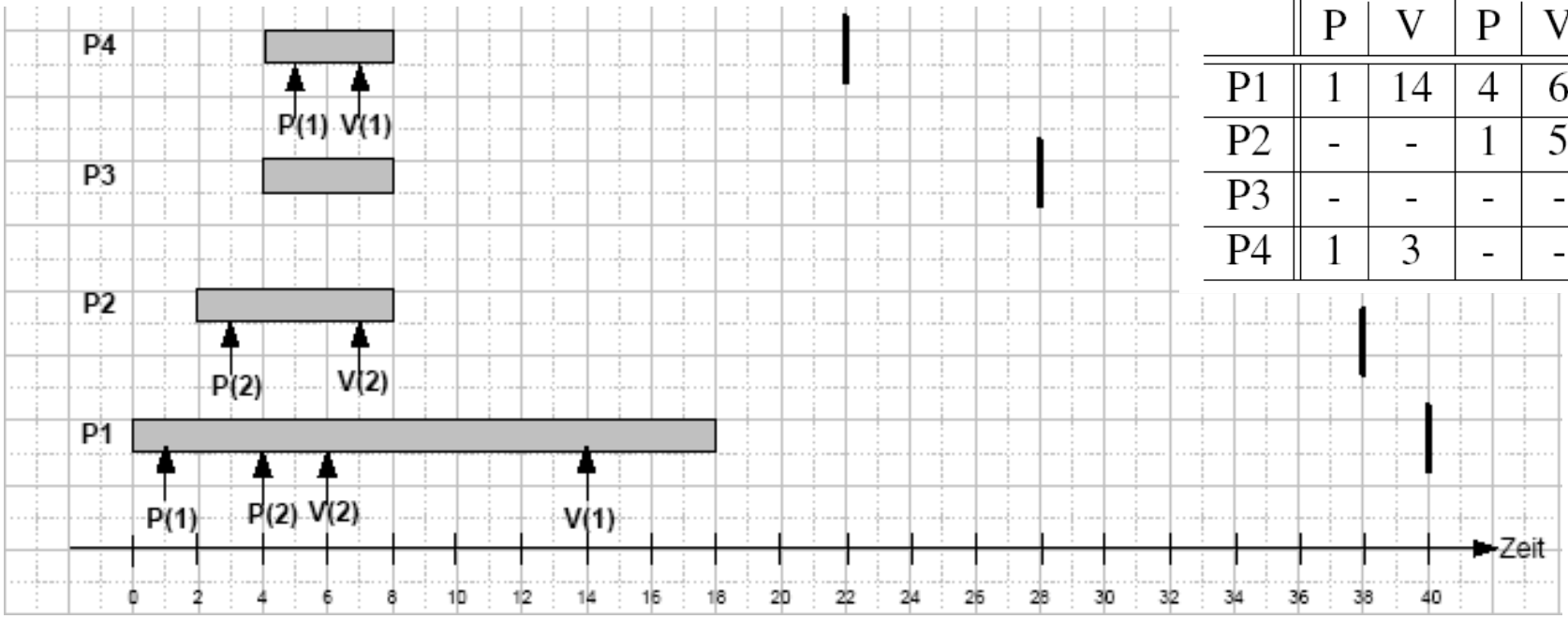
Zusammenfassung

Zusammenfassung

- Kenntniss der Schedulingkriterien (Einhalten von Fristen, Fairness,…) , sowie der verschiedenen Prozessparameter (Startzeit, Laufzeit, Deadline, Priorität).
- Klassische Verfahren (EDF, LST, RM) und Anforderungen an die Optimalität dieser Verfahren
- Problem der Prioritätsinversion, sowie Lösungsverfahren
- Problematik der WCET-Analyse

Klausur WS 07/08

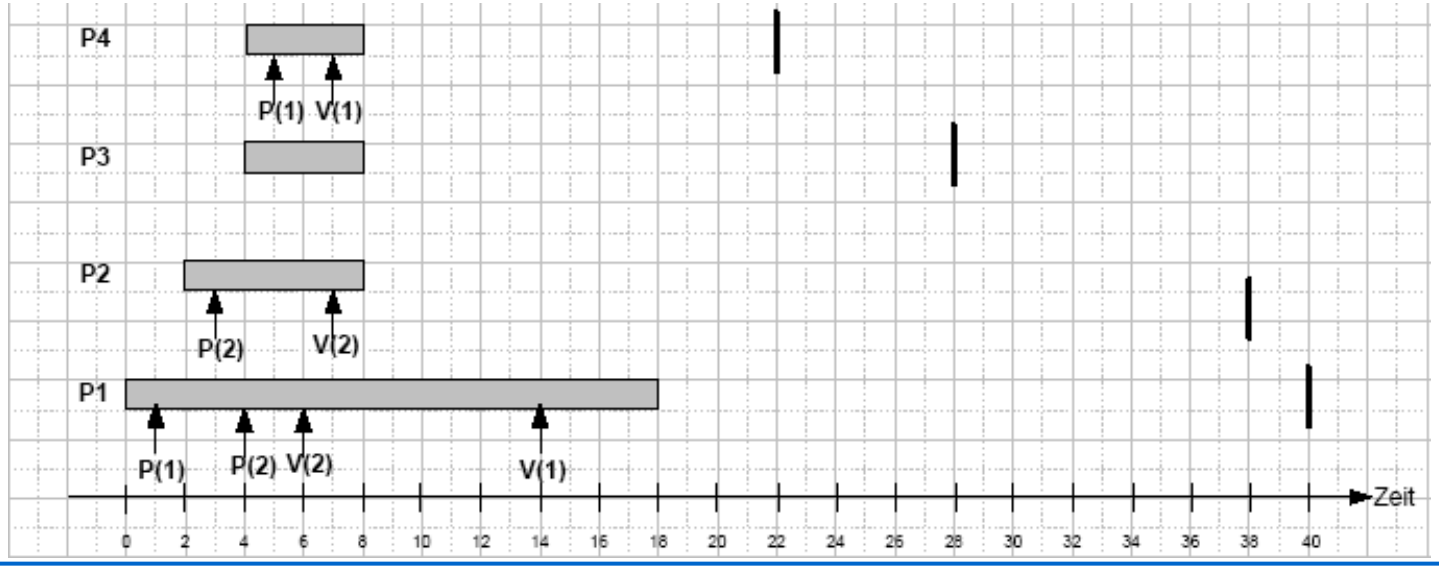
- Startzeiten s: s(P1)=0; s(P2)=2; s(P3)=4; s(P4)=4;
- Ausführungszeiten e: e(P1)=18; e(P2)=6; e(P3)=4; e(P4)=4;
- Fristen d: d(P1)=40; e(P2)=38; e(P3)=28; e(P4)=22;



	S1		S2	
	P	V	P	V
P1	1	14	4	6
P2	-	-	1	5
P3	-	-	-	-
P4	1	3	-	-

Klausur WS 07/08

- Ignorieren Sie zunächst die Semaphore. Zeichnen Sie einen Ausführungsplan für das Schedulingverfahren Earliest-Deadline-First.
- Ignorieren Sie zunächst die Semaphore. Zeichnen Sie einen Ausführungsplan für das Schedulingverfahren Least-Slack-Time (Zeitscheiben: 1).
- Zeichnen Sie nun den Ausführungsplan für das Schedulingverfahren Earliest-Deadline-First unter Berücksichtigung der Semaphoren.
- Wie könnte man das Schedulingverfahren modifizieren, um das in Teilaufgabe c) aufgetretene Problem zu beheben.



Übungsaufgabe Scheduling

Prozess	Startzeit	CPU-Zeit	Statische Priorität
P1	0	8	1 (niedrig)
P2	2	2	3
P3	2	5	4 (hoch)
P4	4	3	2

Gegeben seien die in der Tabelle angegebenen Prozesse:

1. *Zeichnen Sie einen Ausführungsplan für nicht präemptives, prioritätenbasiertes Scheduling (FIFO)?*
2. *Zeichnen Sie einen Ausführungsplan für präemptives, prioritätenbasiertes Scheduling (FIFO).*
3. *Zeichnen Sie einen Ausführungsplan für präemptives, prioritätenbasiertes Scheduling (Round Robin, Zeitscheiben 0.5).*
4. *Zeichnen Sie einen Ausführungsplan für präemptives, prioritätenbasiertes Scheduling (FIFO), wenn jeder Prozess zu Beginn den Semaphor S anfordert und bei Beendigung der Ausführungszeit freigibt und das Betriebssystem Prioritätsvererbung unterstützt.*
5. *Zeichnen Sie einen Ausführungsplan für präemptives, prioritätenbasiertes Scheduling (FIFO), wenn jeder Prozess zu Beginn den Semaphor S anfordert und bei Beendigung der Ausführungszeit freigibt und das Betriebssystem immediate priority ceiling unterstützt.*

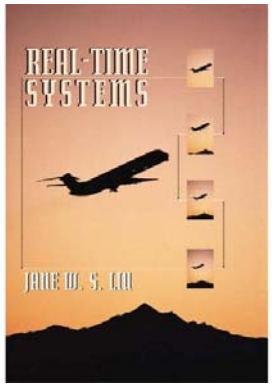
Kapitel 5

Echtzeitbetriebssysteme

Inhalt

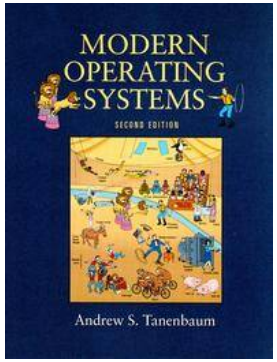
- Grundlagen
- Betrachtung diverser Betriebssysteme:
  - Domänenspezifische Betriebssysteme:
    - OSEK
    - TinyOS
  - Klassische Echtzeitbetriebssysteme
    - QNX
    - VxWorks
    - PikeOS
  - Linux- / Windows-Echtzeitvarianten
    - RTLinux/RTAI
    - Linux Kernel 2.6
    - Windows CE

Literatur



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

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



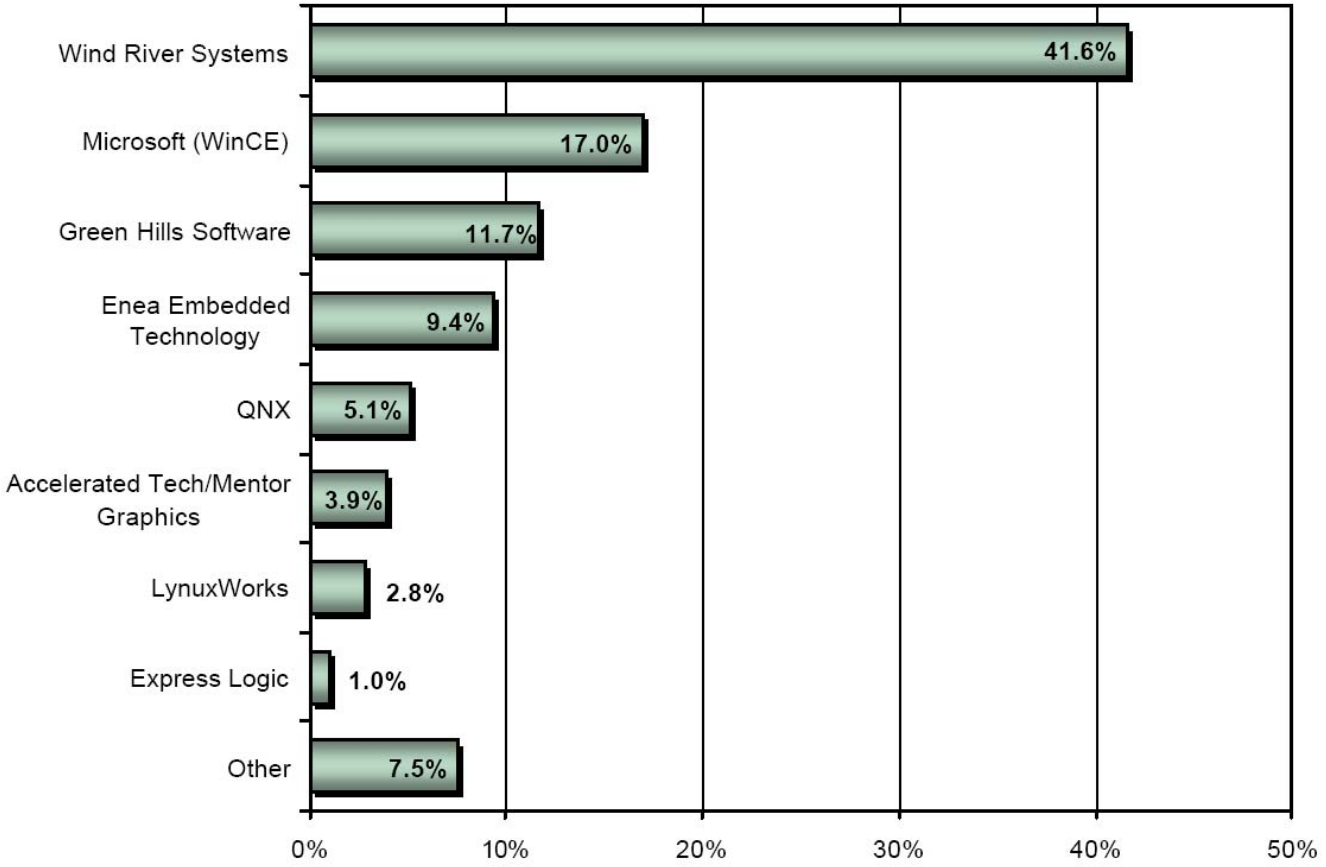
Andrew S. Tanenbaum: Modern Operating Systems, 2001

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

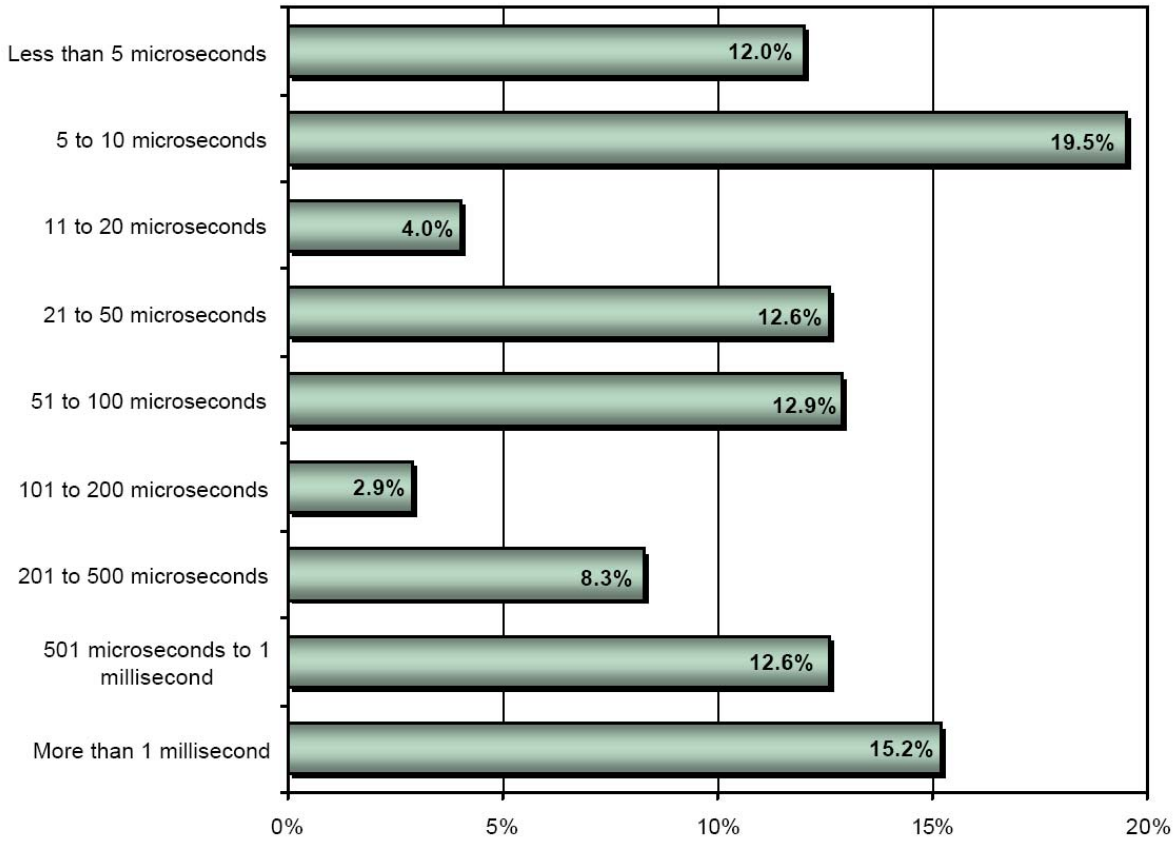
- Echtzeitbetriebssysteme werden auf sehr unterschiedlicher Hardware ausgeführt
  - Der verfügbare Speicher variiert sehr stark.
  - Typische Betriebssystemfunktionalitäten (z.B. Dateisysteme, graphische Oberfläche) werden oft gar nicht benötigt.

  ⇒ 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

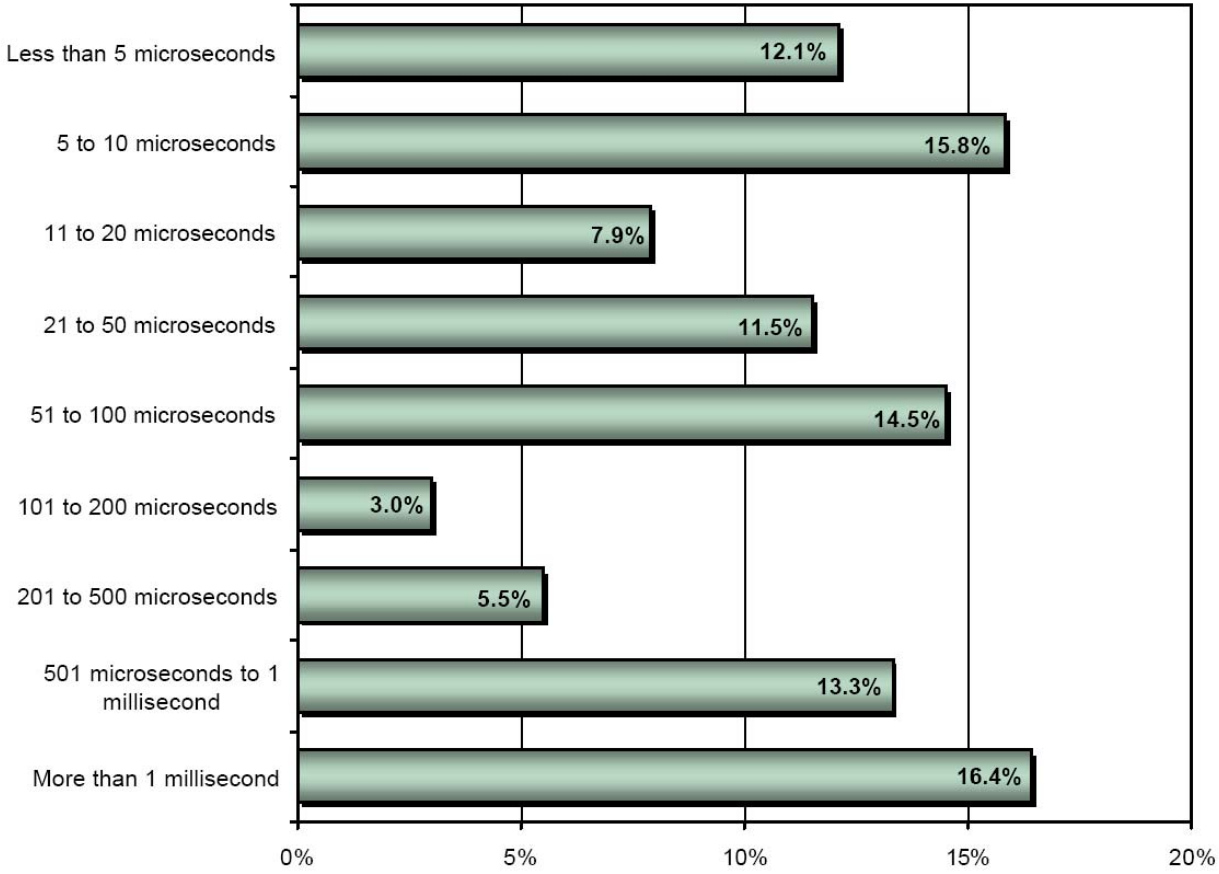
- Die Echtzeitfähigkeit wird durch die Messung folgender Zeiten bestimmt:
  - **Unterbrechungsantwortzeiten** (**interrupt latency**): der Zeitraum zwischen dem Auftreten einer Unterbrechung und der Ausführung des ersten Befehls der dazugehörigen Unterbrechungsbehandlungsroutine
  - **Schedulinglatenz** (**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

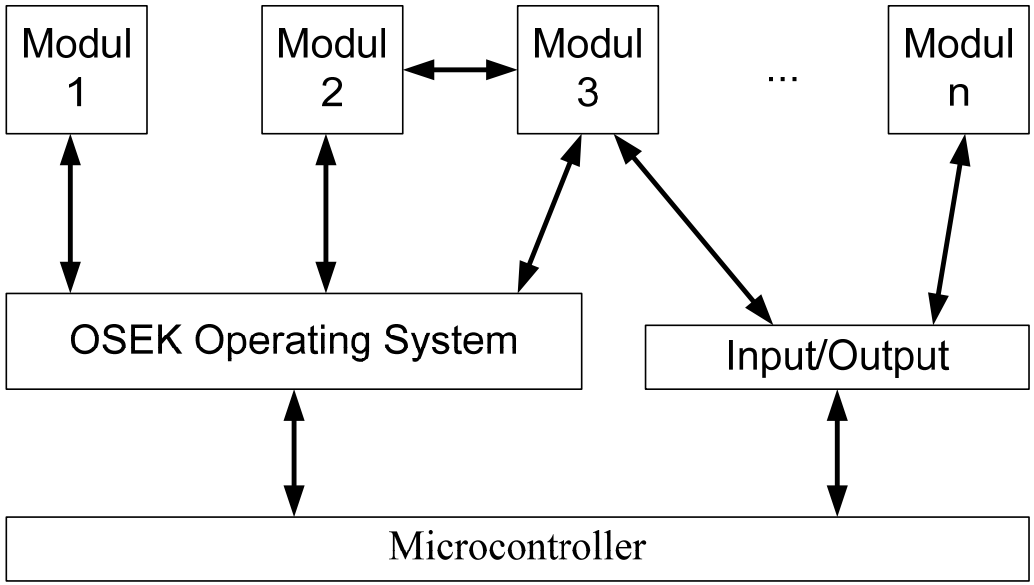
- Gemeinschaftsprojekt der deutschen Automobilindustrie (u.a. BMW, DaimlerChrysler, VW, Opel, Bosch, Siemens)
- OSEK: **O**ffene **S**ysteme und deren Schnittstellen für die **E**lektronik im **K**raftfahrzeug
- Ziel: Definition einer Standard-API für Echtzeitbetriebssysteme
- Standard ist frei verfügbar (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)

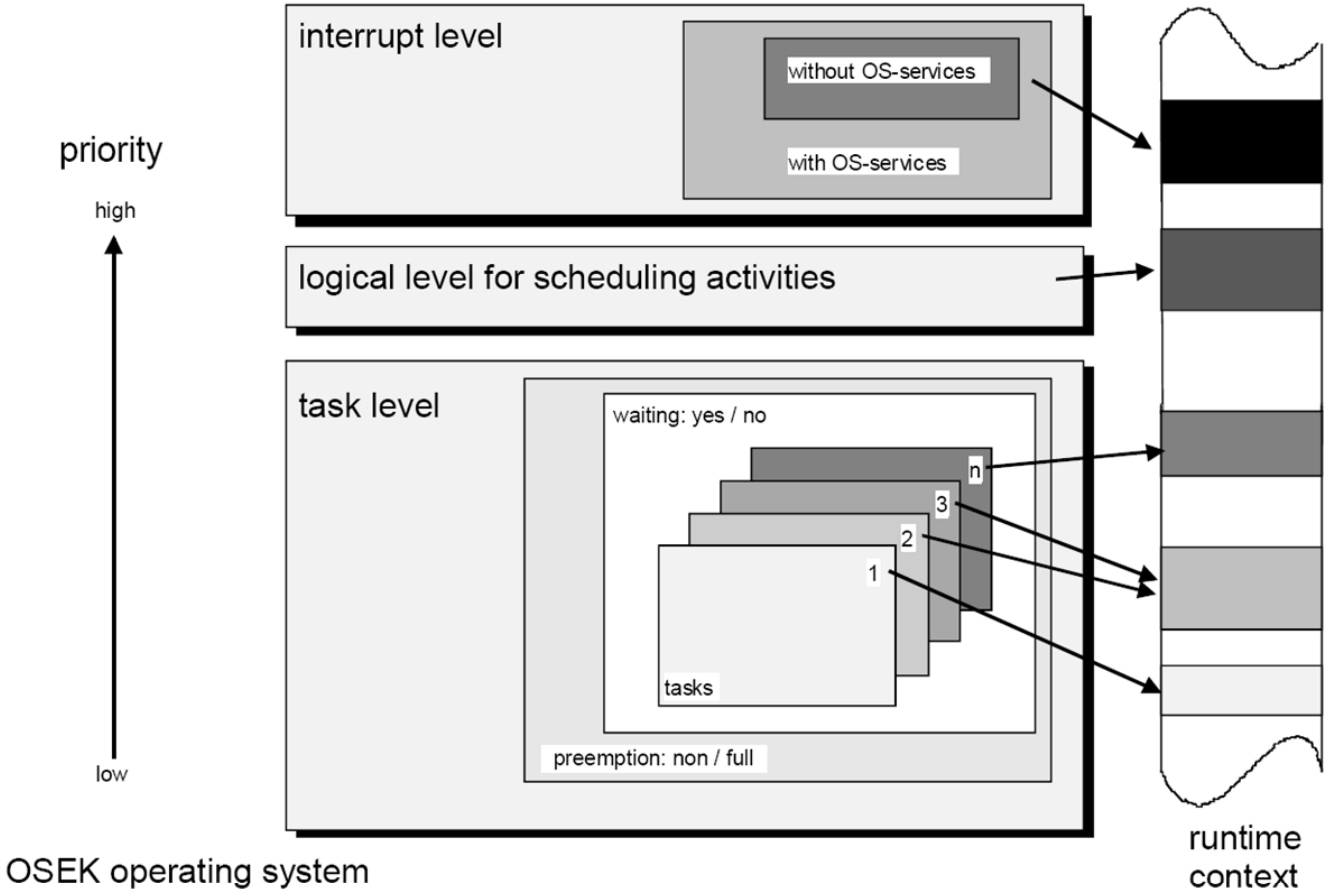
  $\Rightarrow$ typische Anforderungen von Echtzeitsystemen
- Weitere Ziele:
  - Skalierbarkeit
  - einfache Konfigurierbarkeit des Betriebssystems
  - Portabilität der Software
  - Statisch allokiertes Betriebssystem

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

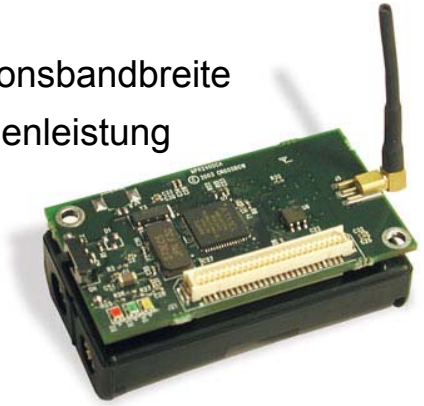
- Zur Vermeidung von Prioritätsinversion und Verklemmungen schreibt OSEK ein Immediate Priority Ceiling Protokoll vor:
  - Jeder Ressource wird eine Grenze (Maximum der Priorität der Prozesse, die die Ressource verwenden) zugewiesen.
  - Falls ein Prozess eine Ressource anfordert, wird die aktuelle Priorität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

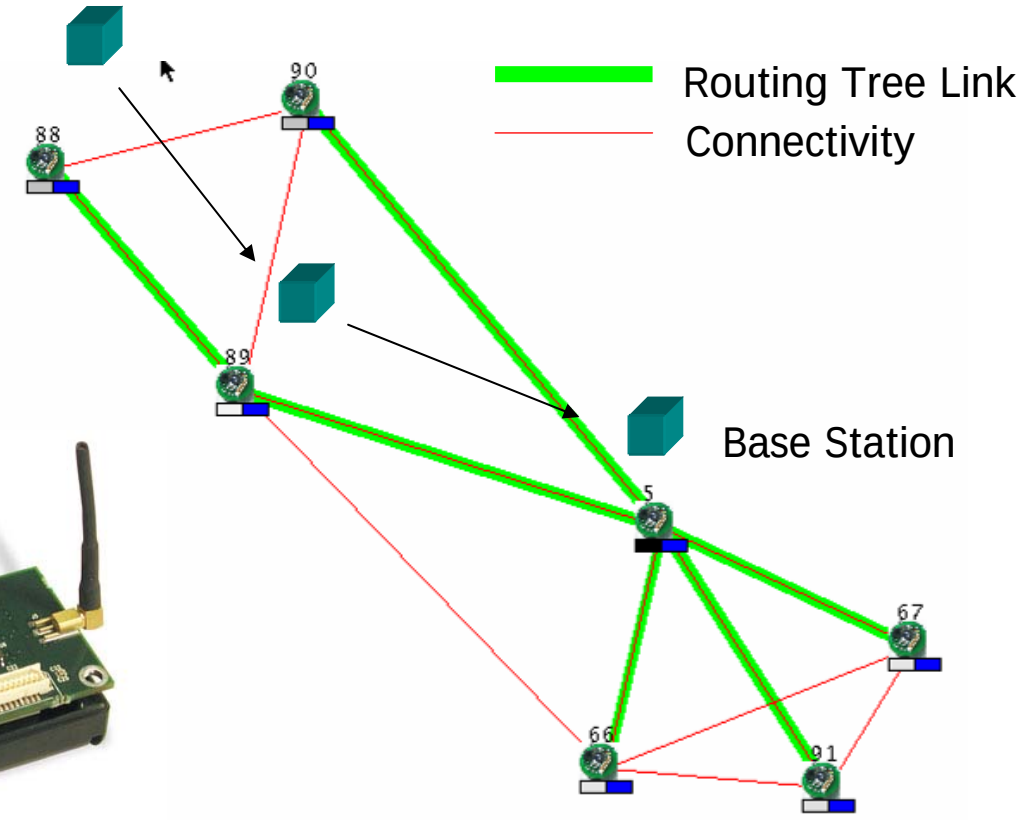
Echtzeitbetriebssysteme

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

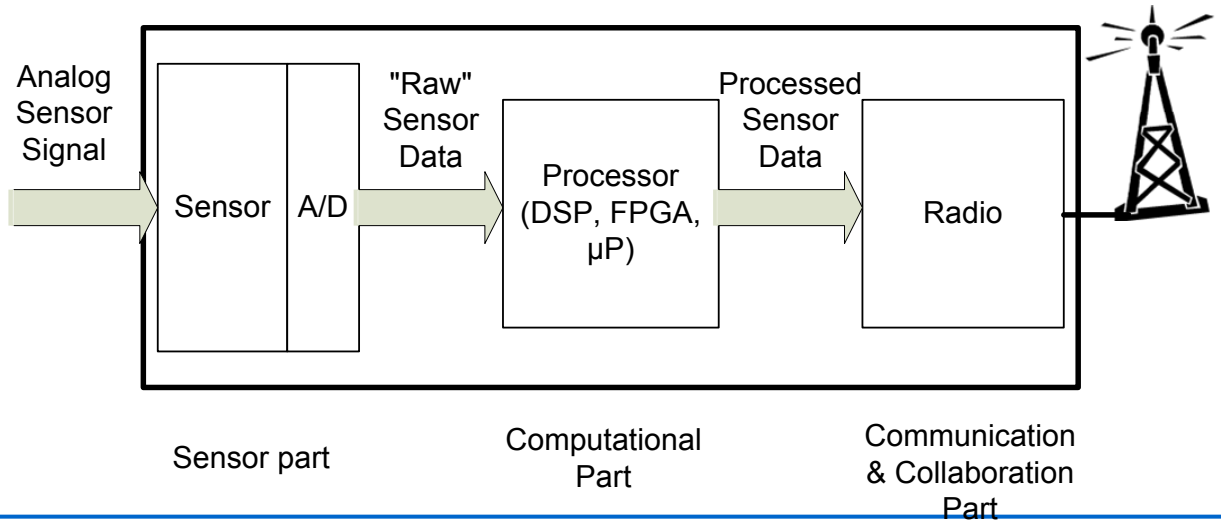




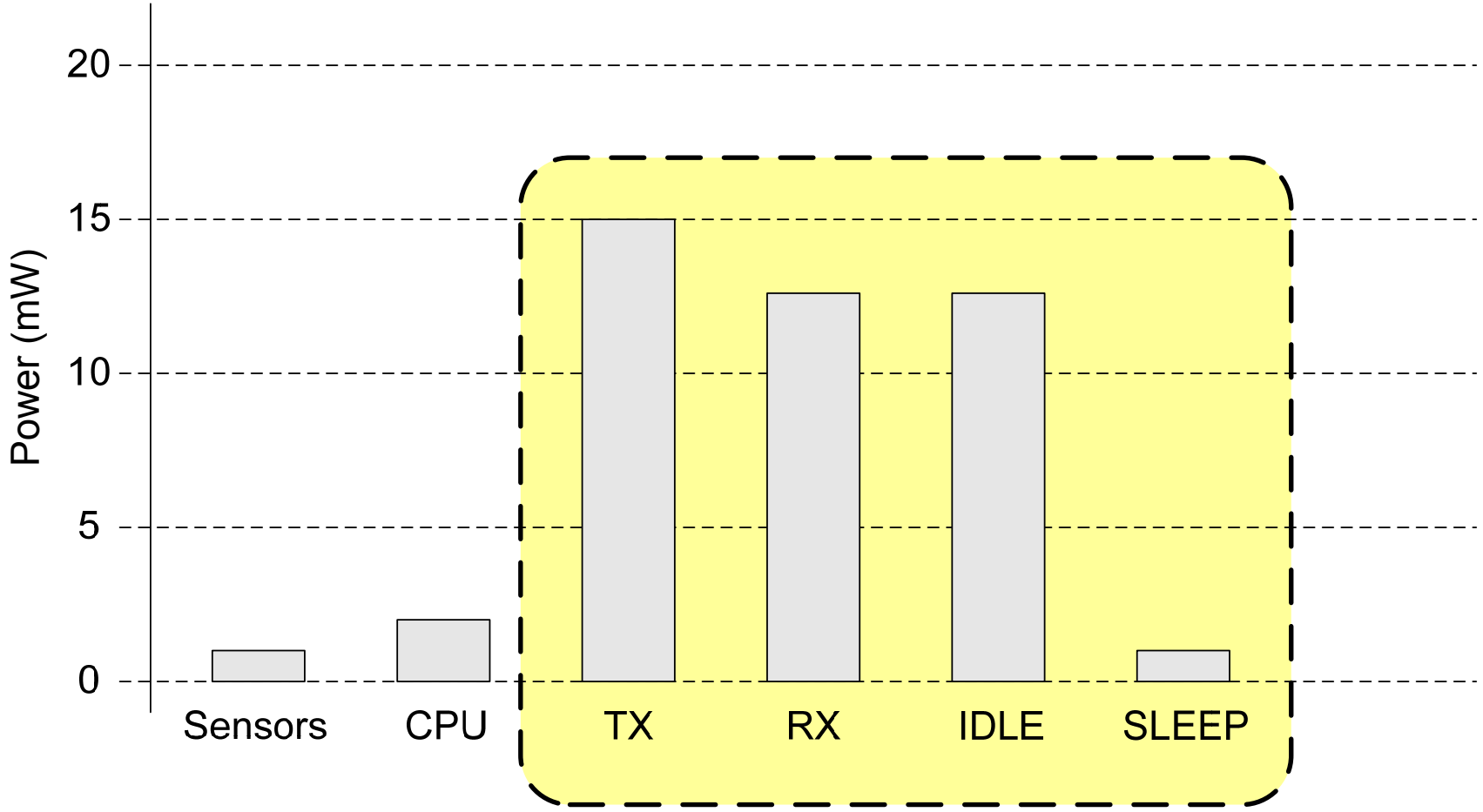
Quelle: http://tinyos.millennium.berkeley.edu

Hardware

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

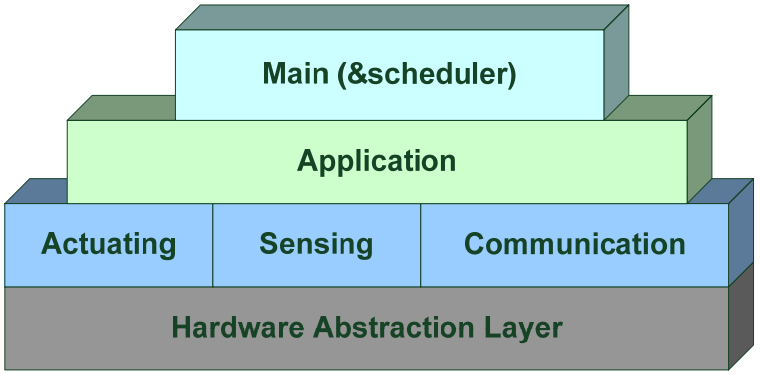


Stromverbrauch

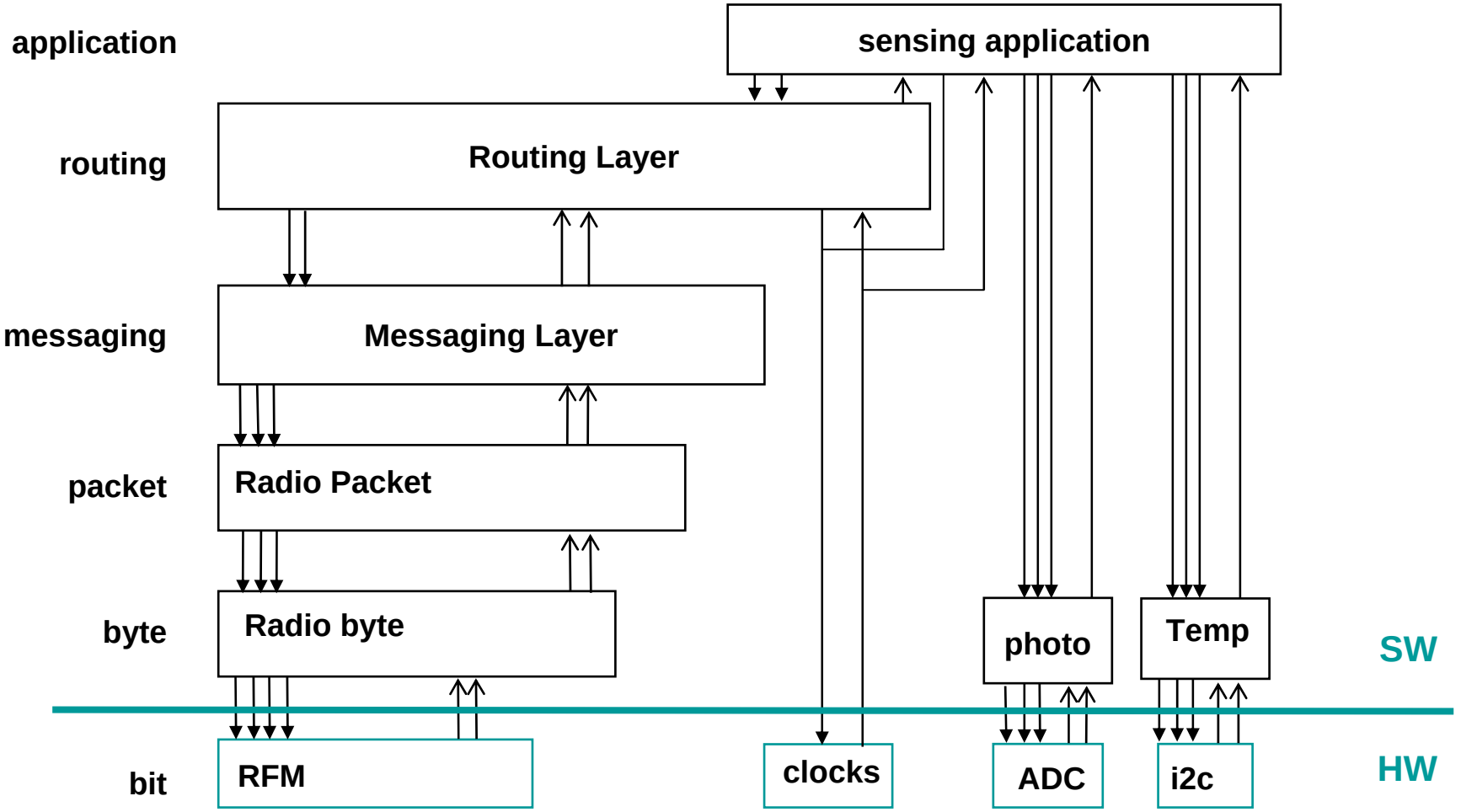


TinyOS

- 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



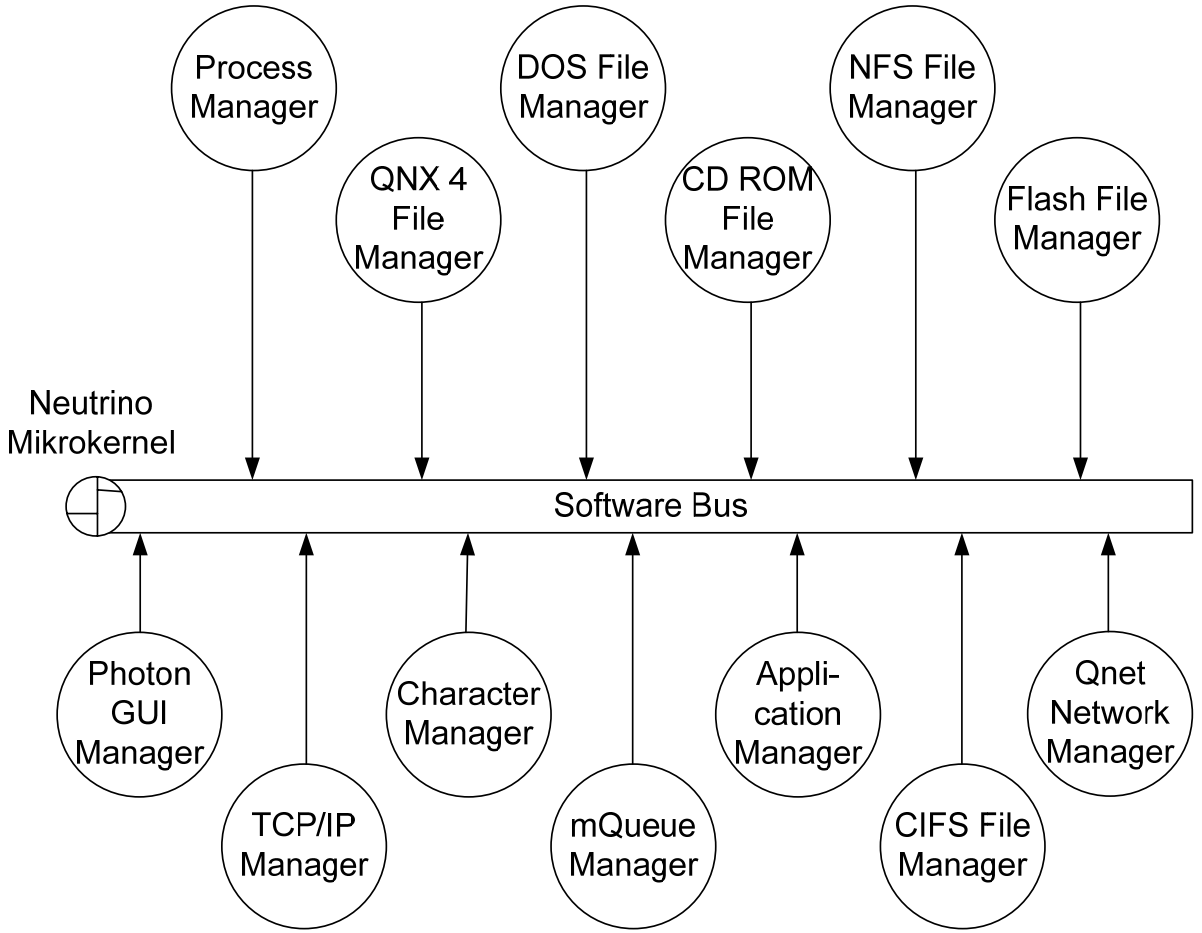
Echtzeitbetriebssysteme

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. ⇒ 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

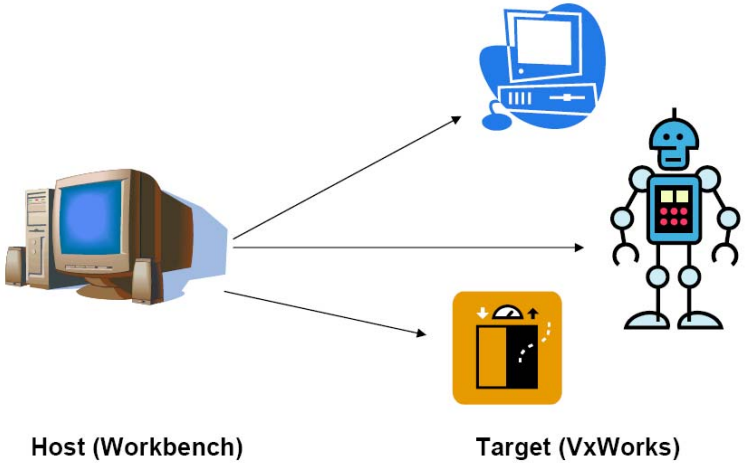
- Als wichtigster Prozess läuft in QNX der Prozessmanager.
- Die Aufgaben sind:
  - Prozessmanagement:
    - Erzeugen und Löschen von Prozessen
    - Verwaltung von Prozesseigenschaften
  - Speichermanagement:
    - Bereitstellung von Speicherschutzmechanismen,
    - von gemeinsamen Bibliotheken
   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

Echtzeitbetriebssysteme

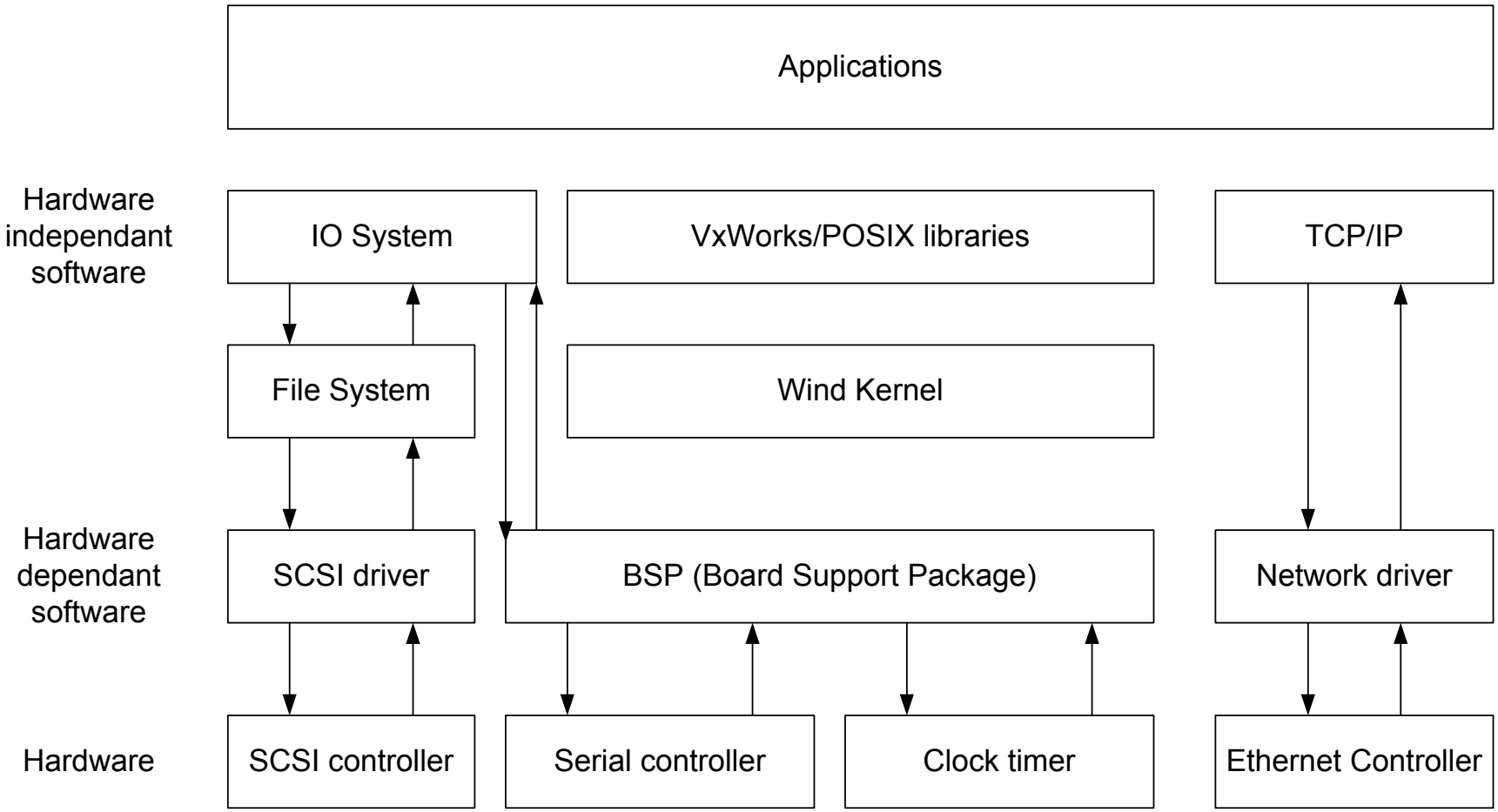
VxWorks

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

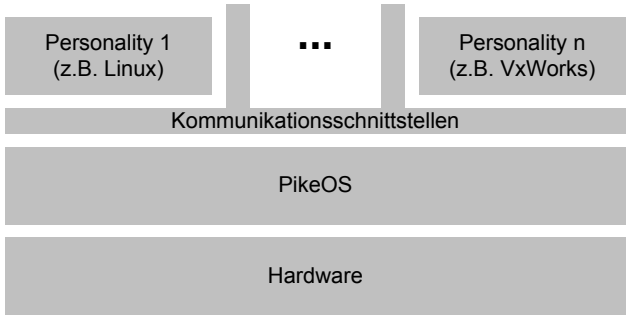
- Zur Interprozesskommunikation werden folgende Konzepte unterstützt:
  - Semaphor
  - Mutex (mit Prioritätsvererbung)
  - Nachrichtenwartschlangen
  - Signale
- Seit Version 6.0 wird zudem Speichermanagement angeboten:
  - Der Entwickler kann Benutzerprozesse mit eigenem Speicherraum entwickeln.
  -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

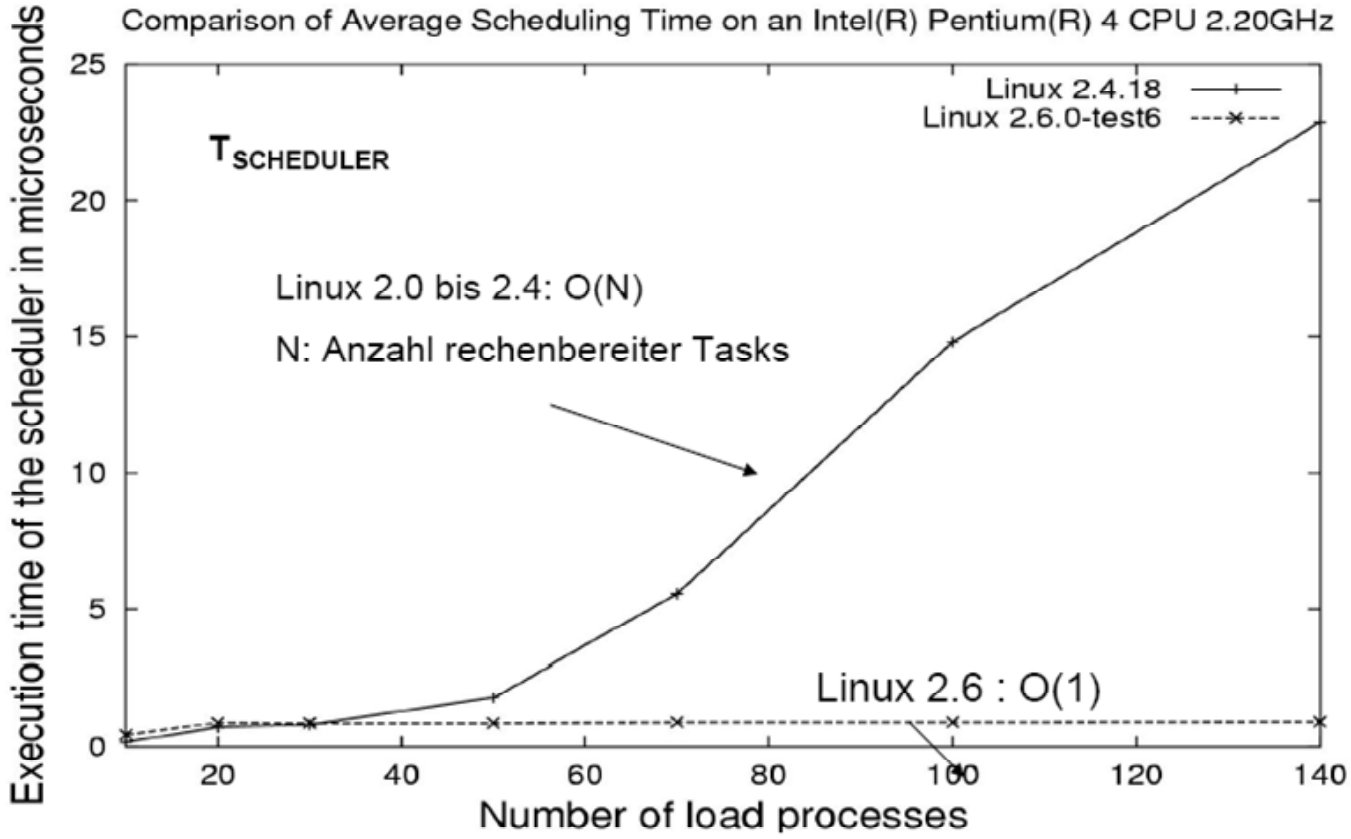
Echtzeitbetriebssysteme

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.
  ⇒ 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 ⇒ 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

Echtzeitbetriebssysteme

RTLinux/RTAI

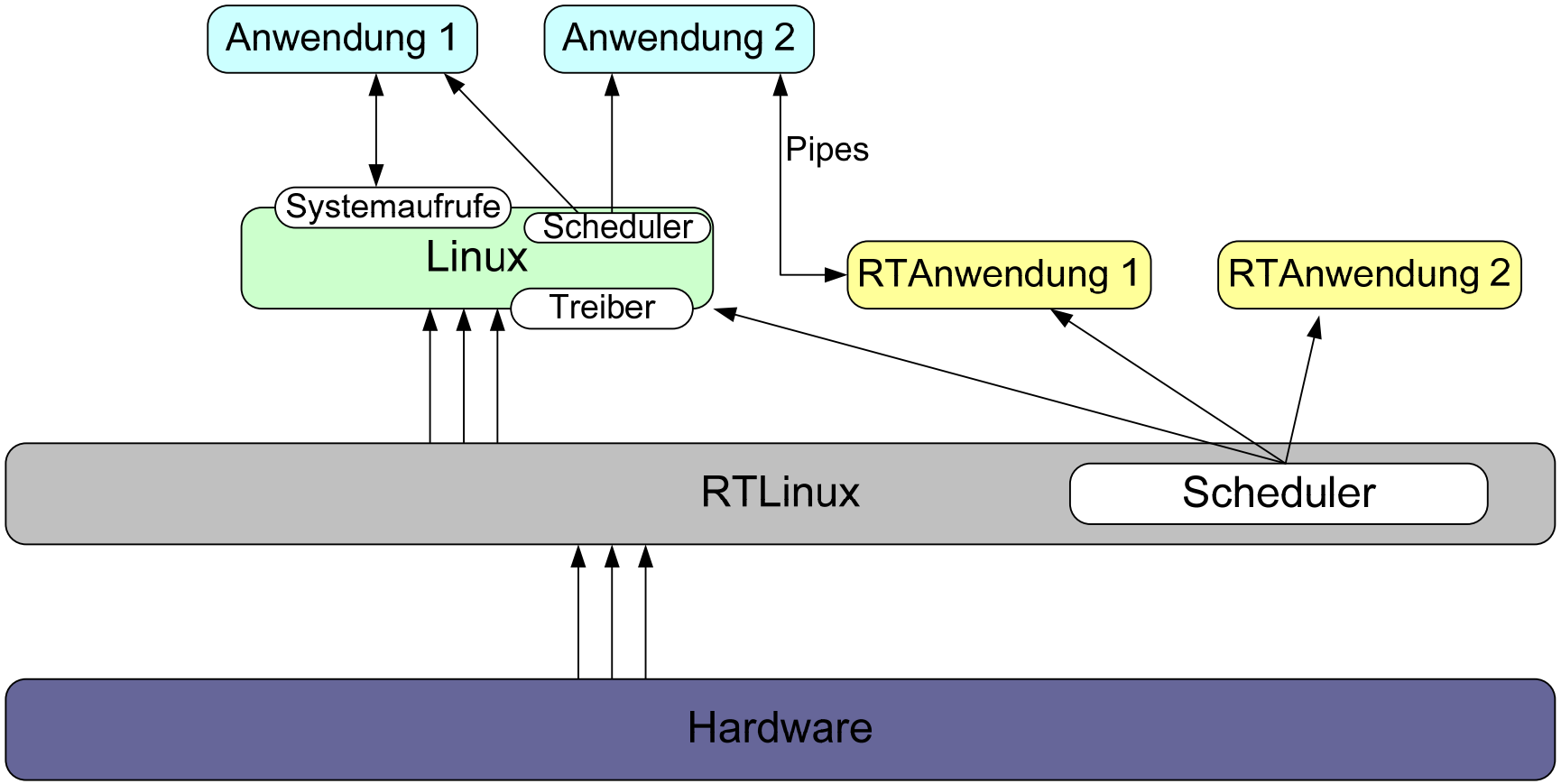
Motivation

- 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

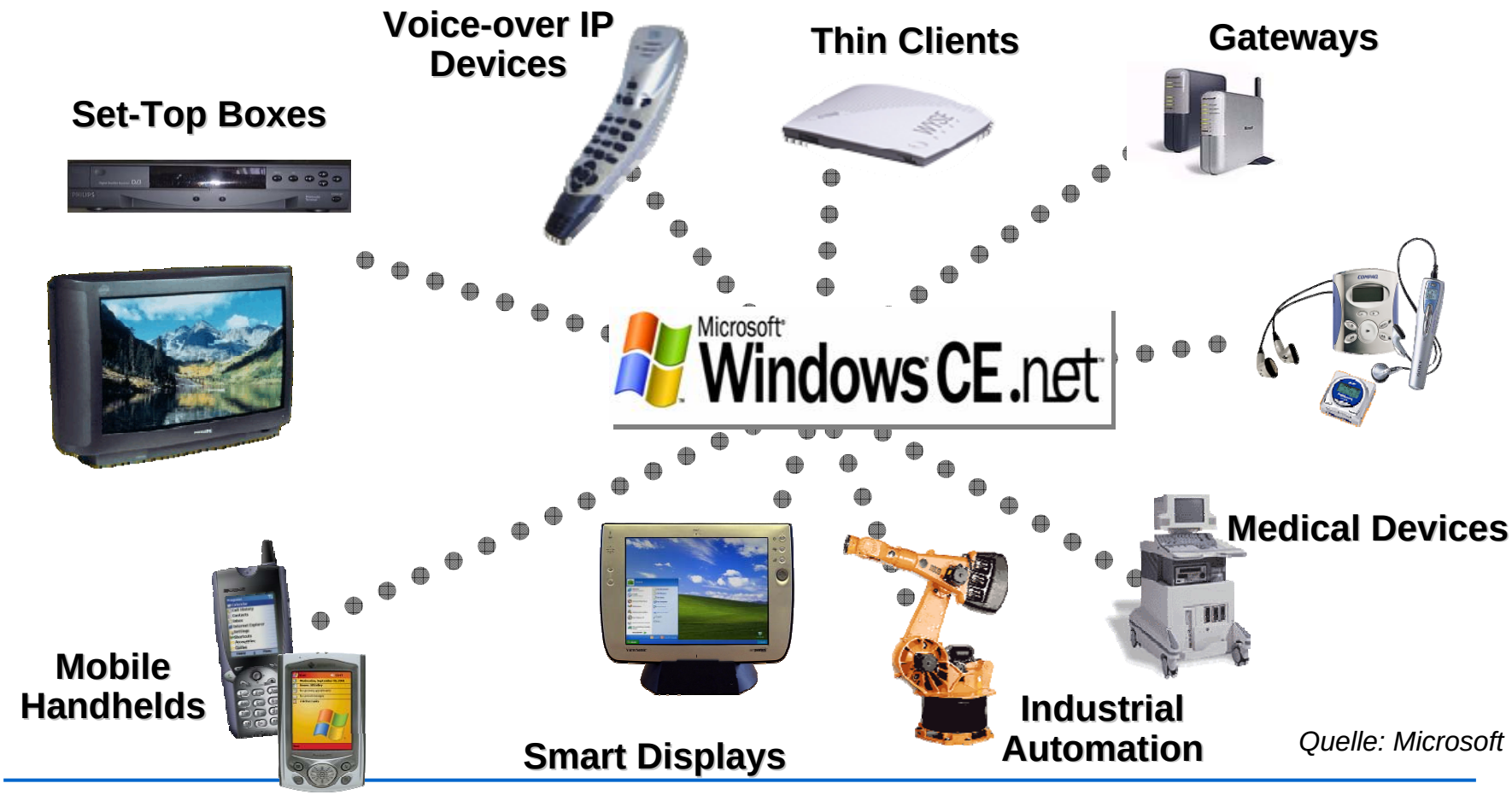
- Windows CE
  - 32-bit, Echtzeitbetriebssystem
  - Unterstützung von Multitasking
  - Stark modularer Aufbau
  - Skalierbar entsprechend der gewünschten Funktionalität
- Windows Embedded
  - „Skalierbares Windows XP“
  - Komponenten von XP können entfernt werden um den benötigten Speicherplatz zu minimieren

Windows CE und Embedded im Vergleich

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

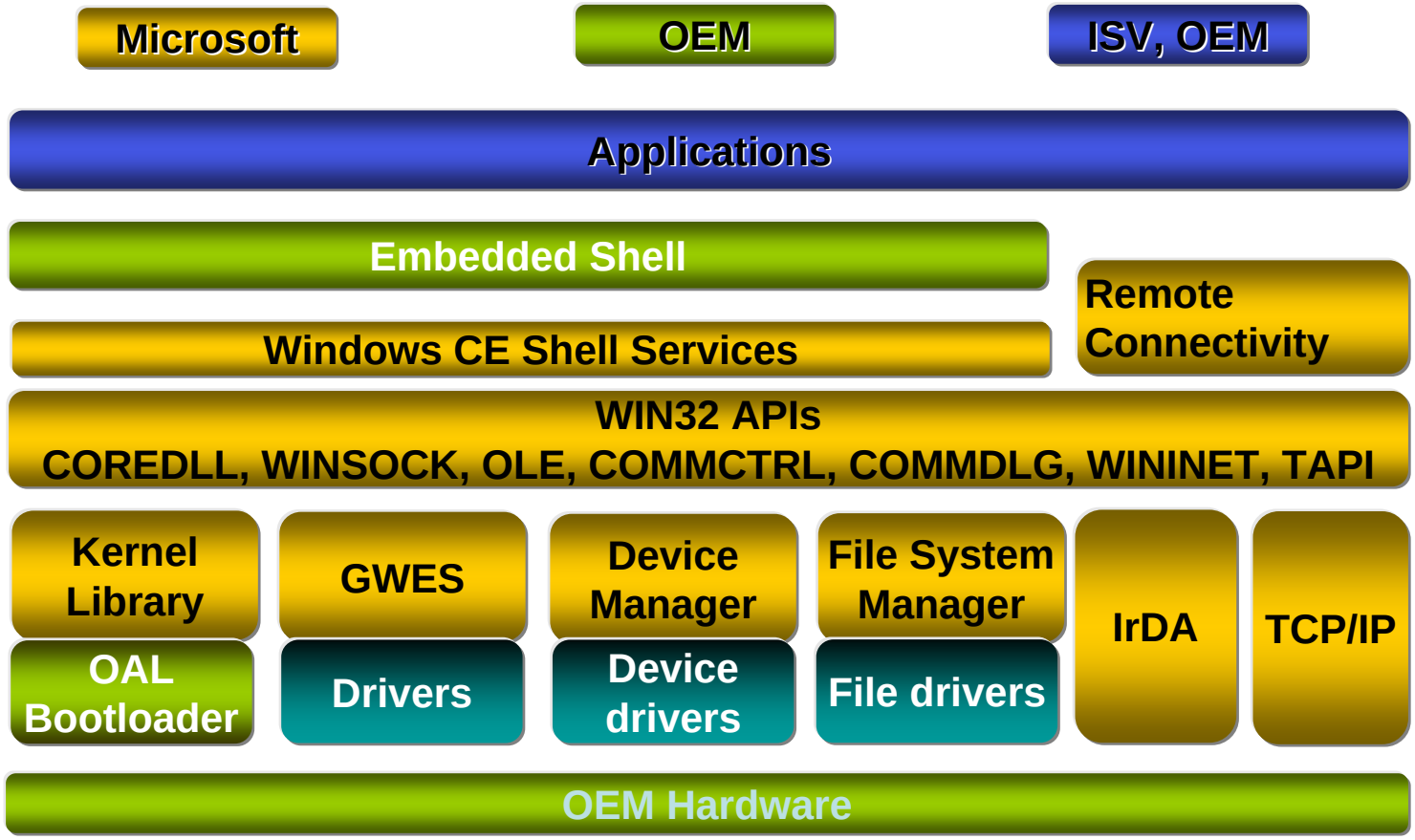
Quelle: Microsoft

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.

Klausurfragen

- Wiederholungsklausur WS 2006/2007
  - Erläutern Sie die Unterschiede zwischen Betriebssystemen mit kooperativem Scheduling, mit präemptiven Scheduling und präemptiblen Betriebssystemen.
- Klausur WS 2007/2008
  - Erläutern Sie kurz (jeweils 1-2 Sätze) die Hauptkonzepte von TinyOS, QNX und PikeOS.

Klausurfragen - Klausur WS 2006/2007

- Gegeben seien folgende fünf Echtzeitbetriebssysteme:
  1. TinyOS
  2. OsekTime
  3. QNX
  4. VxWorks
  5. WindowsCE

  und folgende fünf Anwendungen:
  a. Sicherheitsüberwachung für Robotersteuerung: auf Basis eines Controllers mit geringen Rechen- und Speicherkapazitäten wird eine Anwendung zur Überwachung der Robotersteuerung implementiert. Dringt eine Person in den Arbeitsbereich des Roboters ein, so wird dieses Ereignis durch Lichtschranken detektiert und der Roboter unverzüglich gestoppt.
  b. ZebraNet: Zur Erforschung der Wanderwege von Zebras, werden kleine Funkmodule ausgestattet mit GPS-Sensoren zur Lokalisation und Solarzellen zur Stromversorgung an Zebras angebracht. Nähern sich zwei Zebras, so tauschen die Funkmodule die gesammelten Daten aus.
  c. Zugsteuerung: Zur Steuerung und Überwachung eines Zuges werden in einem verteilten System verschiedene periodische Regelungsfunktionen ausgeführt. Diese Funktionen werden dabei als Komponenten von unterschiedlichen Entwicklergruppen zur Verfügung gestellt. Insbesondere eine reibungslose Integration dieser Komponenten steht im Vordergrund.
  d. Fabrikanlagensteuerung: In einer Chemiefabrik wird die Produktion von leistungsfähigen Feldrechnern gesteuert. Diese Rechner sind mit einer Vielzahl von Sensorik verbunden. Auf kritische Ereignisse muss schnell reagiert werden.
  e. PDA: Auf einem Handheld werden unterschiedlichste Anwendungen ausgeführt: unter anderem Routenplaner, Browser, Musik-/Videospieler und Programme zur Terminverwaltung.

  Ordnen Sie jedem der fünf Anwendungsgebiete ein Echtzeitbetriebssystem zu und begründen Sie Ihre Zuordnung knapp mit einem Satz.

Kapitel 8 (vorgezogen)

Echtzeitfähige Kommunikation

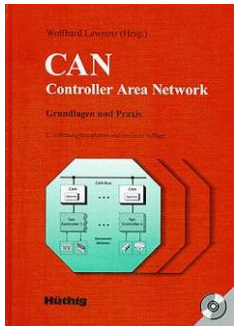
Inhalt

- 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



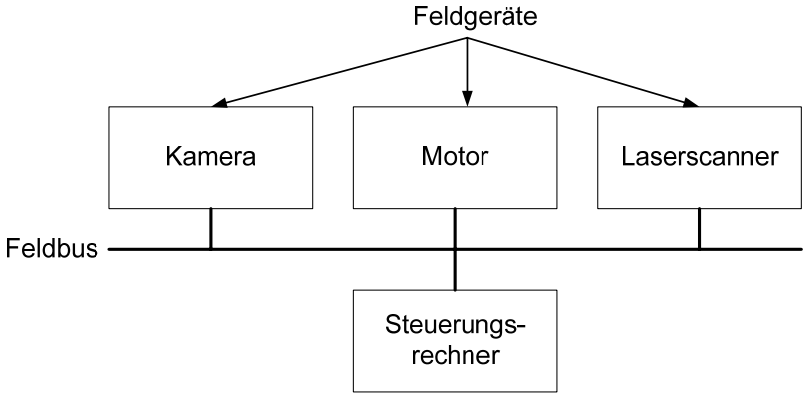
- Spezifikationen:
  - TTTech Computertechnik AG, Time Triggered Protocol TTP/C High-Level Specification Document, 2003 (http://www.vmars.tuwien.ac.at/projects/ttp/)
  - http://www.can-cia.org/
  - http://standards.ieee.org/getieee802/portfolio.html

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

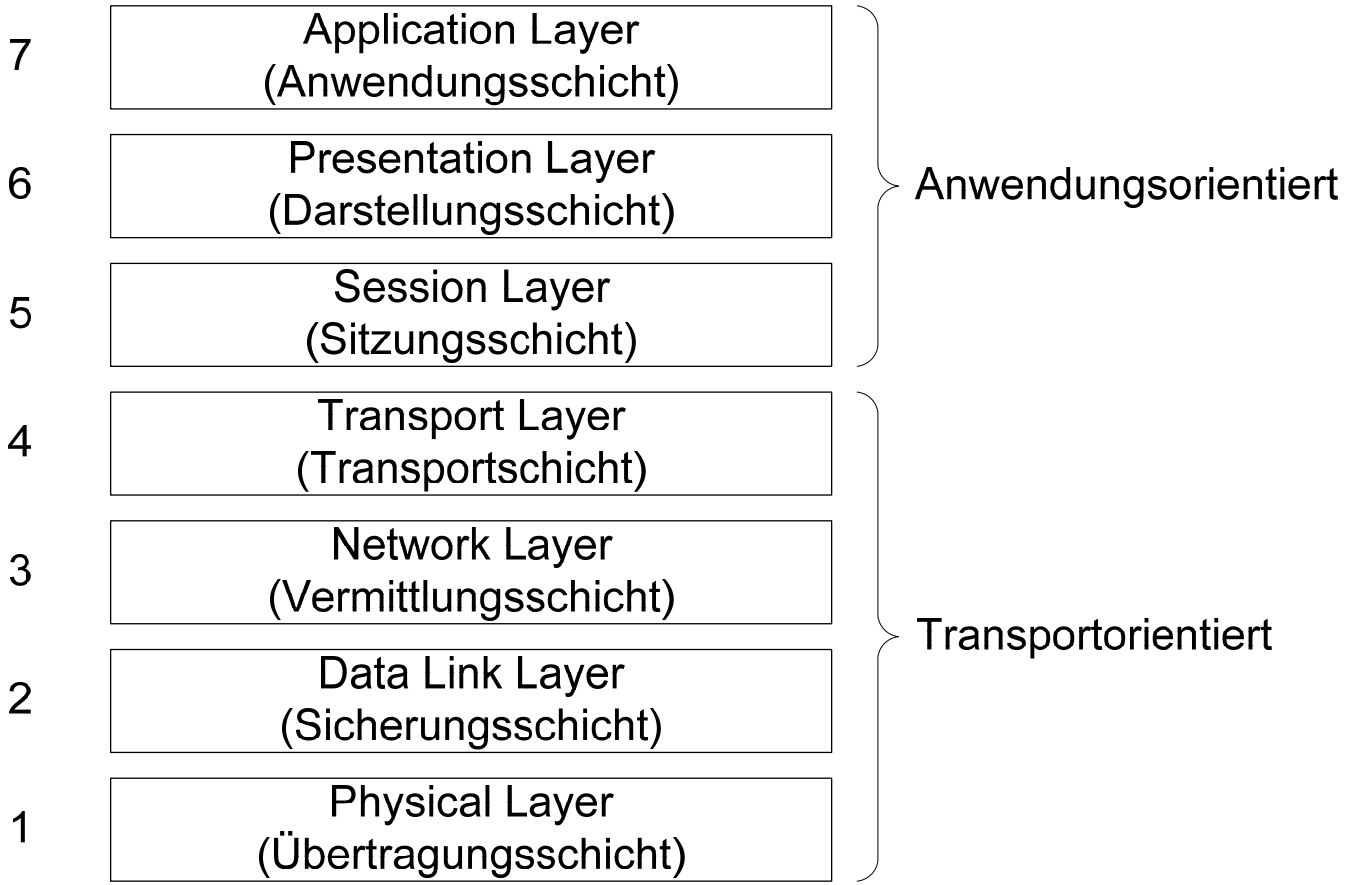
Definition Feldbus

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



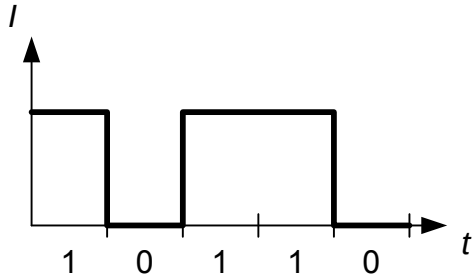
- Feldgeräte sind dabei Sensoren/Aktoren, sowie Geräte zur Vorverarbeitung der Daten.
- Der Feldbus verbindet die Feldgeräte mit dem Steuerungsgerät.
- Beobachtung: echtzeitkritische Nachrichten sind in der Regel kürzer als unkritische Nachrichten.
- Es existiert eine Vielzahl von Feldbus-Entwicklungen: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

Schichtenmodell: ISO/OSI-Modell

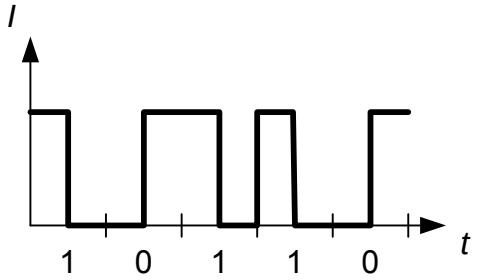


Beschreibung der einzelnen Schichten: Übertragungsschicht

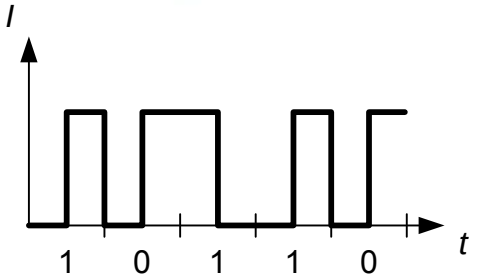
- Aufgaben:
  - Bitübertragung auf physikalischen Medium
  - Festlegung der Medien
    - elektrische, optische Signale, Funk
    - Normung von Steckern
  - Festlegung der Übertragungsverfahren/Codierung
    - Interpretation der Pegel
    - Festlegung der Datenrate



Non-return-to-zero Code



Manchester-Code



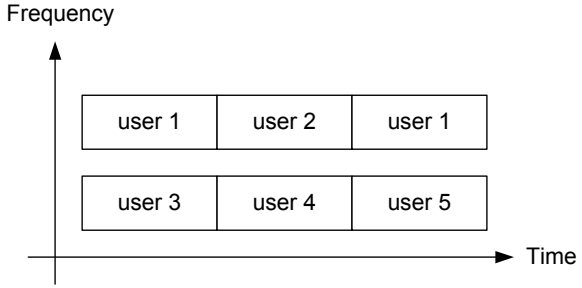
Differentieller Manchester-Code

Beschreibung der einzelnen Schichten: Sicherungsschicht

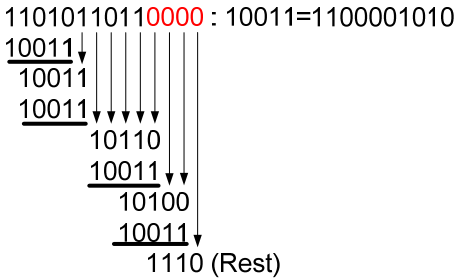
- Aufgaben:
  - Fehlererkennung
    - Prüfsummen
    - Paritätsbits
  - Aufteilung der Nachricht in Datenpakete
  - Regelung des Medienzugriffs
  - Flusskontrolle

								LRC
	1	0	1	1	0	1	0	1	1
	0	1	1	0	0	1	0	0	1
	0	0	0	1	1	0	1	1	0
	1	1	1	0	0	1	0	0	0
VRC	0	0	1	0	1	1	1	0	

Paritätsbits



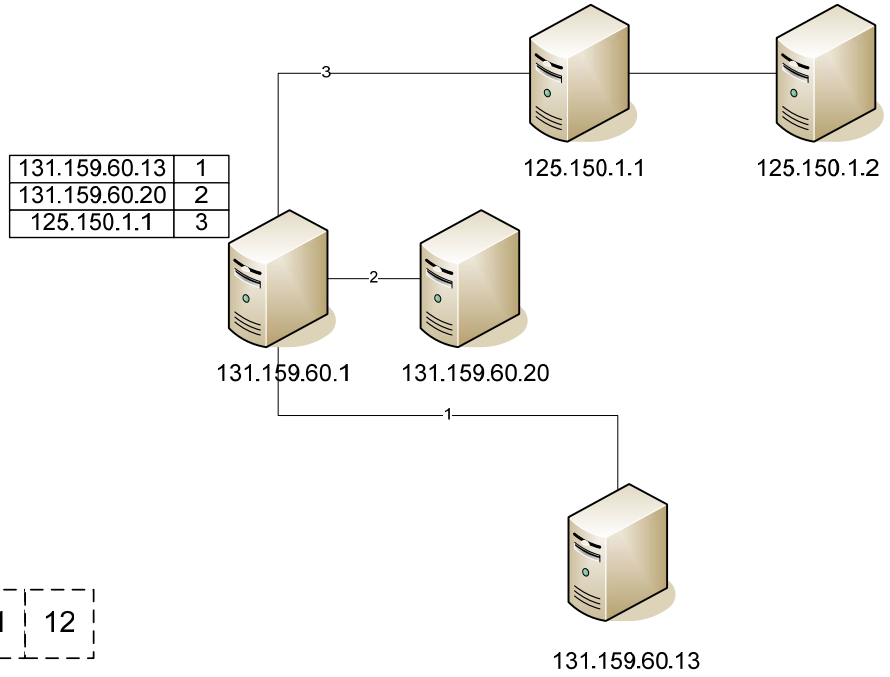
TDMA+FDMA

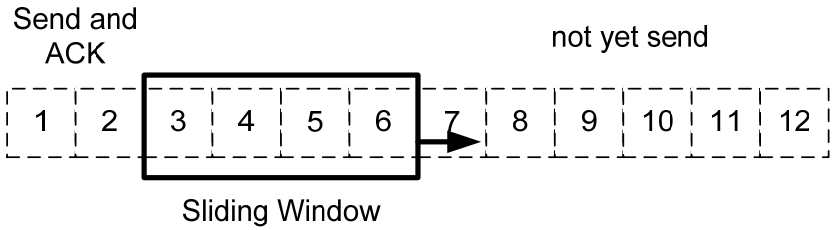


CRC-Verfahren

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

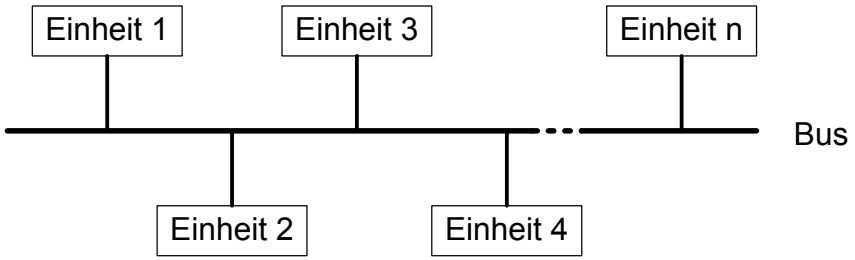
Echtzeitfähige Kommunikation

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**).



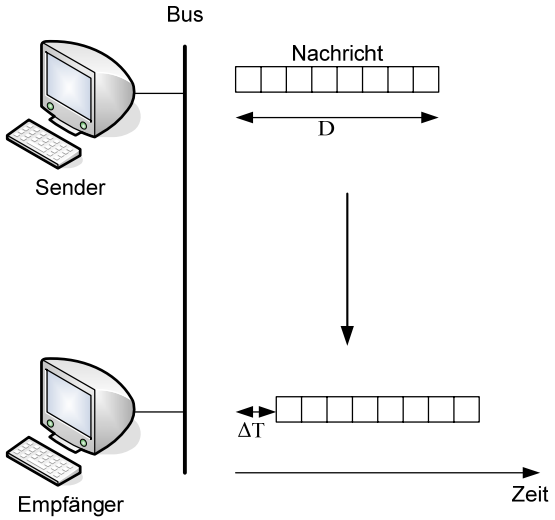
  - Während der Übertragung überprüft der sendende Knoten gleichzeitig das Resultat auf dem Bus, ergibt sich eine Abweichung, so wird eine Kollision angenommen (**Collision Detection**)

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

Teilweise wird die hier vorgestellte Methode auch CSMA/CR (Collision Resolution) genannt.

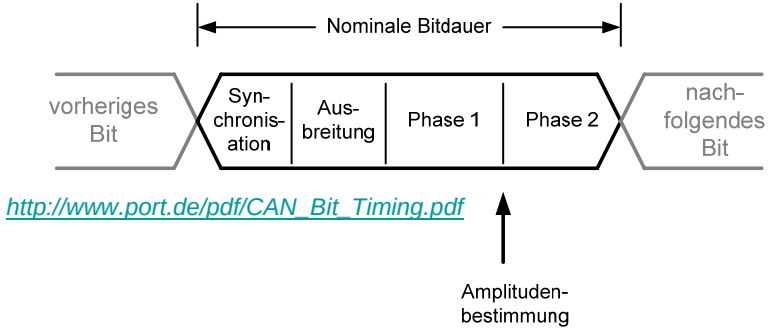
CAN-Protokoll

- Grundidee von Collision Avoidance:
  - Kollisionen werden rechtzeitig erkannt, bevor Nachrichten unbrauchbar werden
  - Wichtigere Nachrichten werden bevorzugt ⇒ 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:

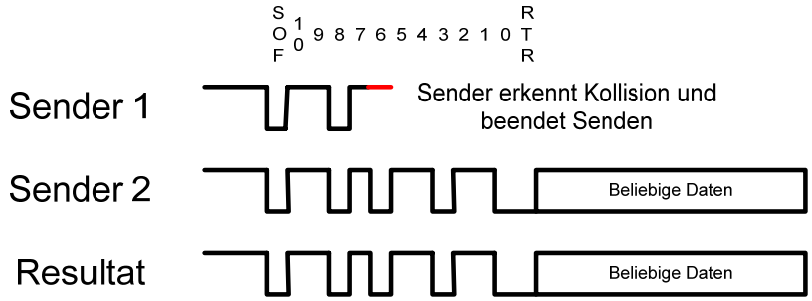
  ⇒ Datenübertragungsrate und maximale Kabellänge sind miteinander verknüpft.
  - Konfigurationsmöglichkeiten:
    - 1 MBit/s, maximale Länge: 40m
    - 500 kBit/s, maximale Länge: 100m
    - 125 kBit/s, maximale Länge: 500m
  - Maximale Teilnehmerzahl: 32-128



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

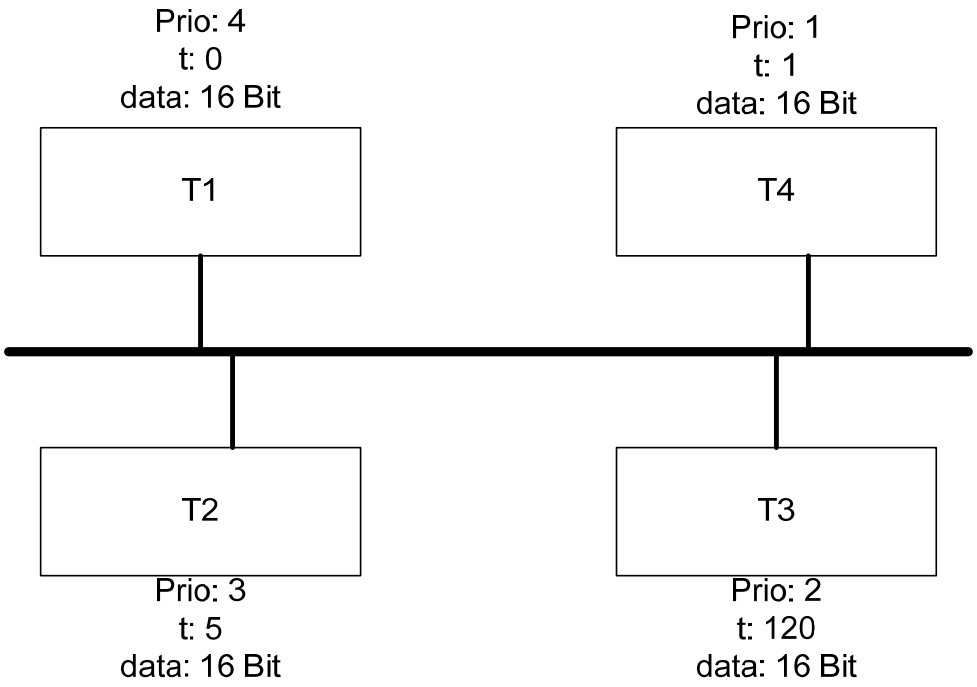
Länge in Bit	1	11	1	1	1	4	0...64	15	1	1	1	7	3
Zweck	Start of frame	Identifier (Extended CAN 27bit)	Remote Transmission Bit	Identifier Extension Bit	reserviert	Datenlängenfeld	Datenfeld	CRC-Prüfsumme	CRC Delimeter	Bestätigungsslot	Bestätigungsdelimeter	End of Frame	Intermission

- Fehlerframe:
  - Signalisierung von erkannten Fehlerbedingungen
- Überlastframe:
  - Zwangspause zwischen Remoteframe und Datenframe

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

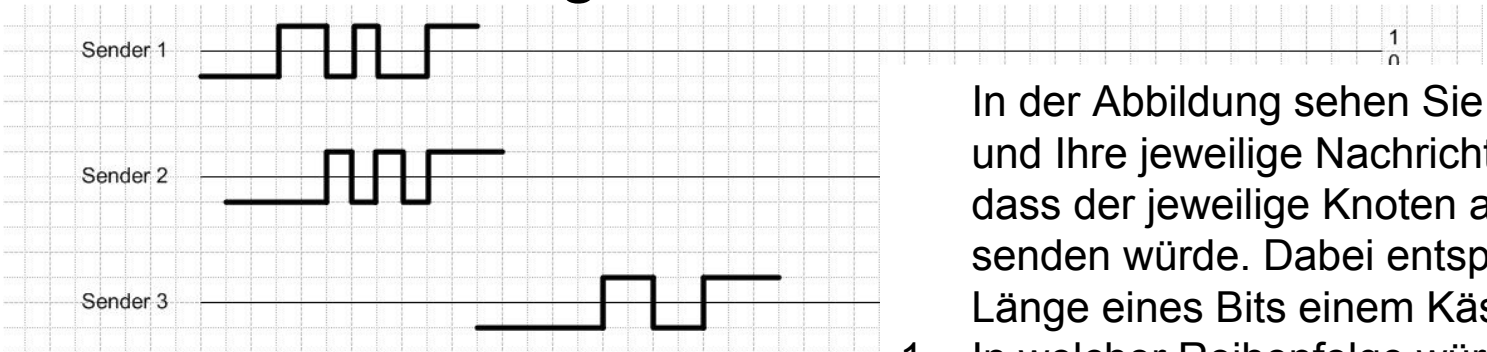
Klausur 06/07 (modifiziert) - CAN



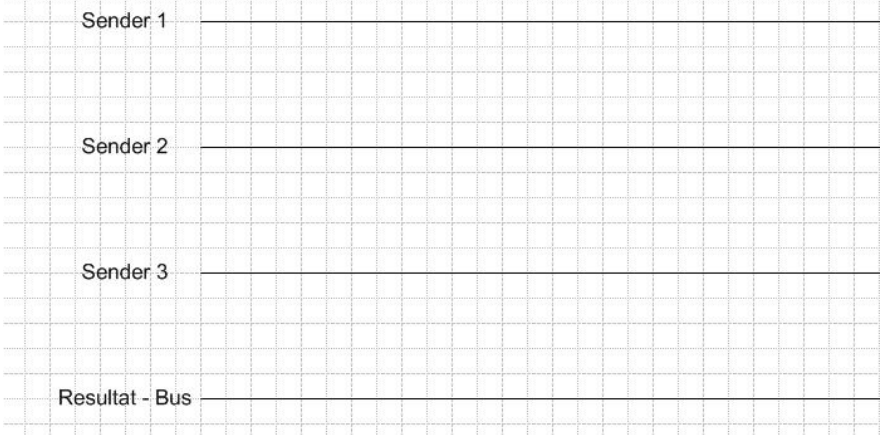
Annahmen: Bitsendedauer 1 Zeiteinheit
Priorität: 1 – hoch, 4 – niedrig

a) Geben Sie die Reihenfolge der Nachrichten an, die im Netzwerk bei Verwendung des CANProtokolls gesendet werden und begründen Sie ihre Antwort. **Zur Erinnerung**: Zusätzlich zu den Nutzdaten sind bei CAN 46 Bit Steuerungsdaten pro Nachricht notwendig. Zwischen den einzelnen Nachrichten ist eine Lücke von mindestens 3 Bit.

Wiederholungsklausur SS 07 – CAN-Protokoll



Tatsächlicher Kommunikationsablauf:



In der Abbildung sehen Sie drei Knoten und Ihre jeweilige Nachricht für den Fall, dass der jeweilige Knoten als einziger senden würde. Dabei entspricht die Länge eines Bits einem Kästchen.

1. In welcher Reihenfolge würden die Nachrichten gesendet werden, wenn alle Knoten gleichzeitig senden würden?
2. Geben Sie auf dem beigelegten Blatt den tatsächlichen Kommunikationsablauf an.
   Anmerkung: Das 0-Bit soll als dominant angenommen werden. Der Intermission Frame Space (also die Mindestanzahl der Bits zwischen zwei aufeinanderfolgenden Nachrichten soll 3 betragen).

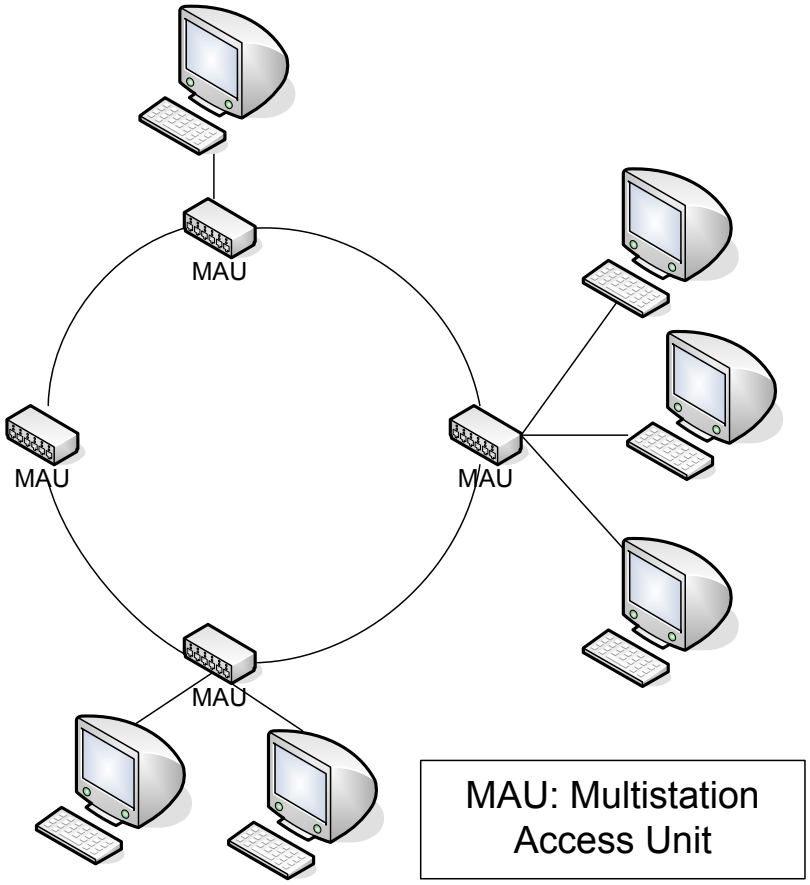
Echtzeitfähige Kommunikation

Tokenbasierte Verfahren

Vertreter: Token Ring

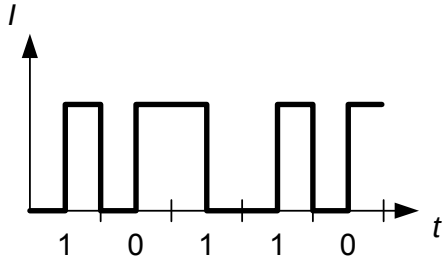
Tokenbasierte Verfahren

- 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)

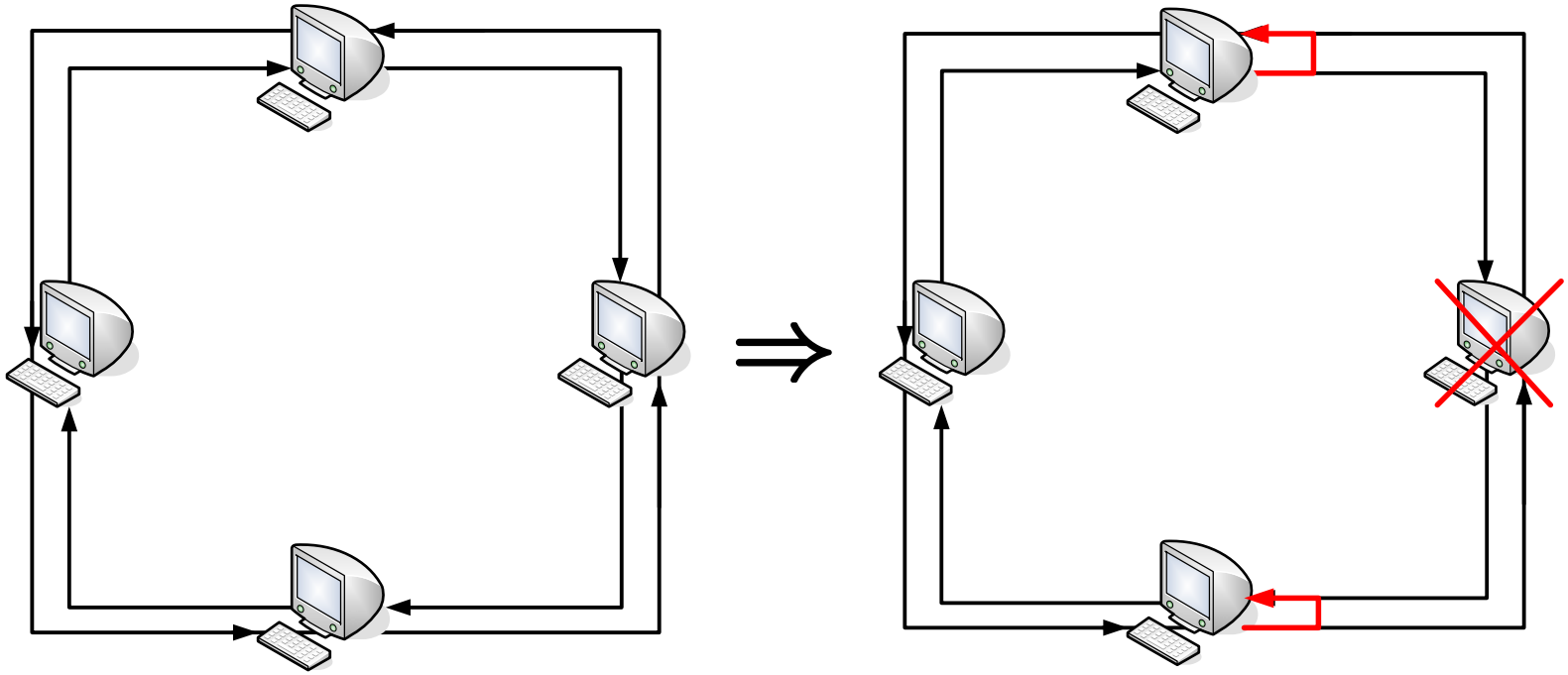
Token Ring: Initialisierung / Rekonfigurierung

- 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

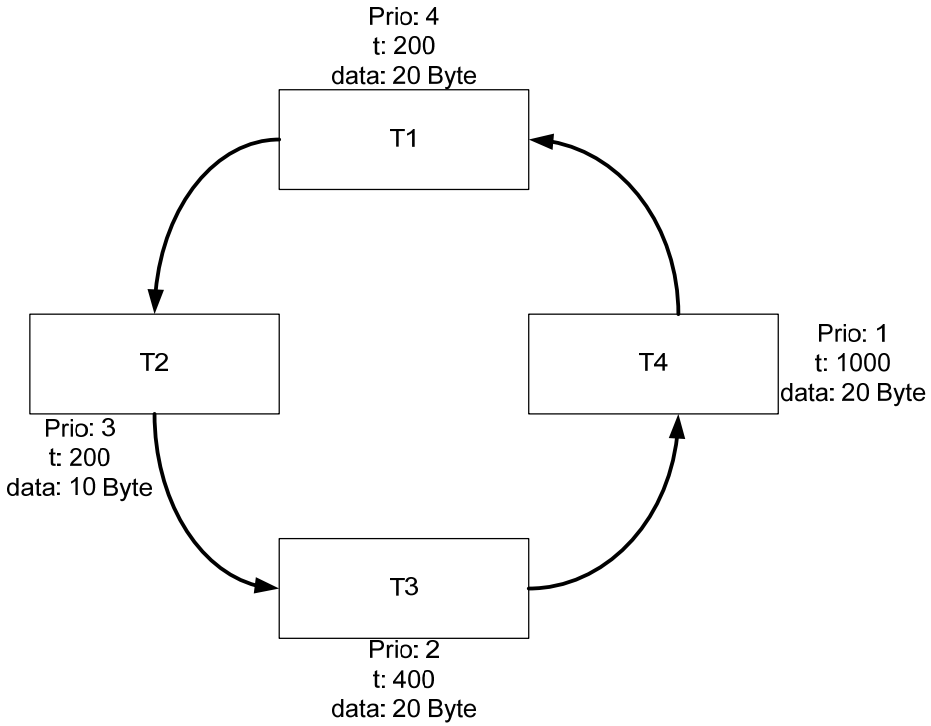
Fehlerkonfiguration in FDDI



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

Klausur 06/07 (modifiziert) - TokenRing



Annahmen: Bitsendedauer 1 Zeiteinheit
Laufzeit zwischen 2 Knoten 200 Zeiteinheiten
Priorität: 1 – hoch, 4 – niedrig

a) Geben Sie die Reihenfolge der Nachrichten an, die im Netzwerk bei Verwendung des TokenRing-Protokolls gesendet werden und begründen Sie ihre Antwort.
Zum Zeitpunkt 0 soll dabei der Teilnehmer T1 im Besitz des Tokens sein.
Zur Erinnerung: Ein Token besteht aus insgesamt 3 Byte (8 Bit Startbegrenzer, 8 Bit Zugriffskontrolle mit Zugriffspriorität und Reservierungspriorität, 8 Bit Endbegrenzer).
Der Header für ein Datenpaket besteht aus mindestens 20 Byte.

Echtzeitfähige Kommunikation

Zeitgesteuerte Verfahren

Vertreter: TTP

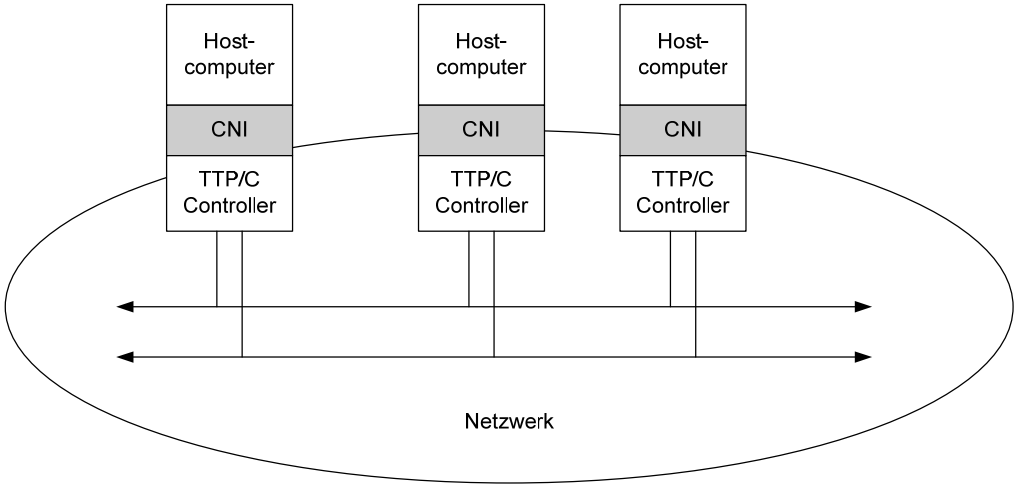
Zugriffsverfahren: TDMA

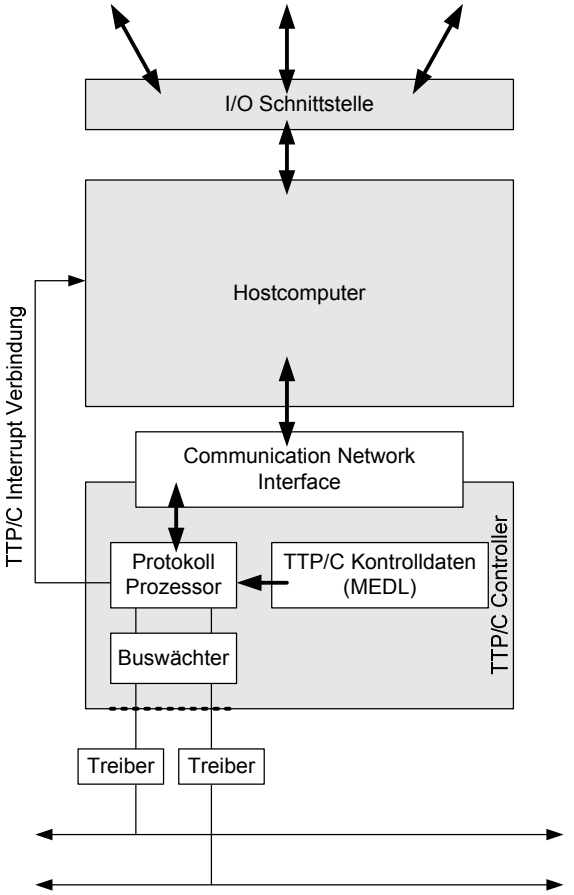
- **TDMA** (**Time Division Multiple Access**) bezeichnet ein Verfahren, bei dem der Zugriff auf das Medium in Zeitscheiben (slots) eingeteilt wird.
- Die Zeitscheiben werden für jeweils einen Sender zur Verfügung gestellt.
- Vorteile:
  - Kollisionen sind per Design ausgeschlossen
  - Einzelnen Sendern 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
- Bekannte Vertreter: TTP, Flexray (kombiniert zeitgesteuert und dynamische Kommunikation)

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

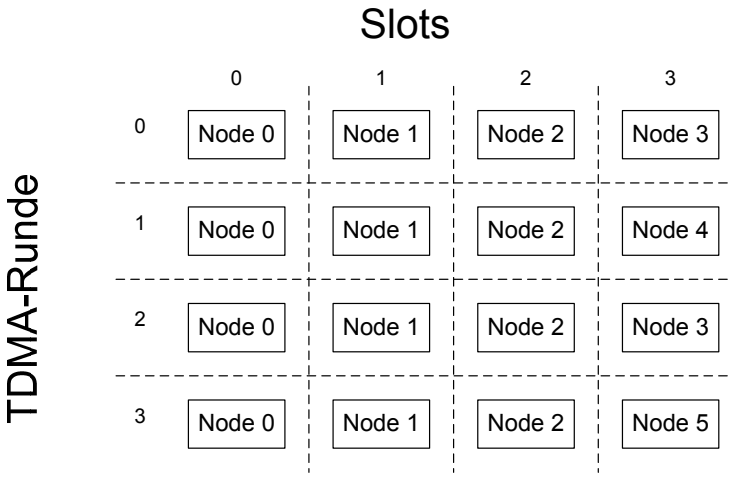




- Erläuterung:
  - Hostcomputer: Ausführung der eigentlichen Anwendung
  - CNI: Gemeinsamer Speicherbereich von Hostcomputer und TTP/C-Kontroller
  - Unterbrechungsverbindung: zur Übermittlung von Ticks der globalen Uhr und außergewöhnlicher Ereignisse an den Hostcomputer
  - MEDL: Speicherplatz für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

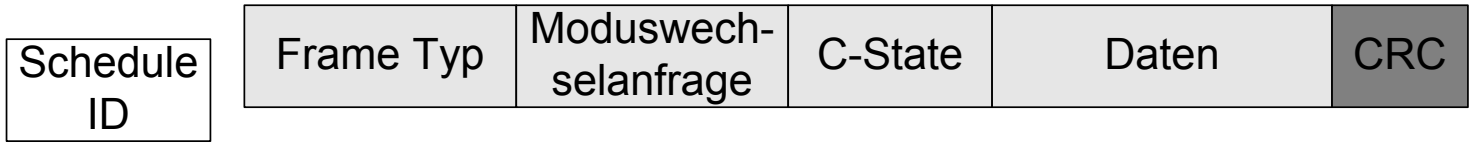
- Interne physikalische Fehler:
  - Erkennung einerseits durch das Protokoll, sowie Verhinderung eine babbling idiots durch Guards.
- Externe physikalische Fehler:
  - Durch redundante Kanäle können diese Fehler toleriert werden.
- Designfehler des TTP/C Kontrollers:
  - Es wird von einem fehlerfreien 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.

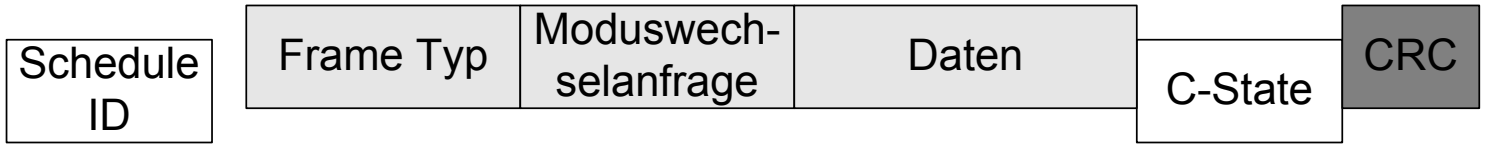
Datenpakete in TTP

- Paket mit explizitem C-State



- Kaltstartpaket

- Paket mit implizitem C-State



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.

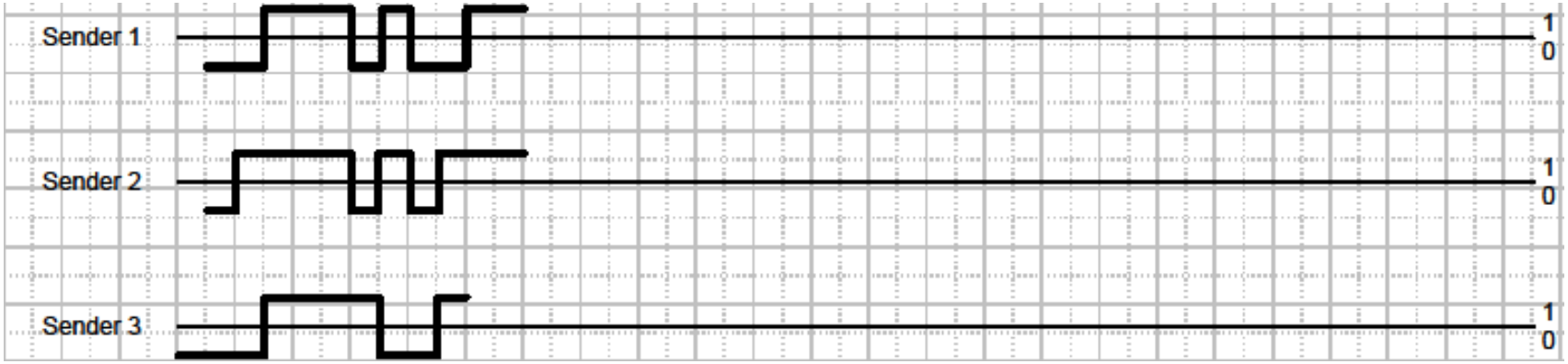
Trends: Real-Time Ethernet

- 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

Klausurfragen

- Klausur Wintersemester 07/08
  - Erläutern Sie kurz die wesentlichen Unterschiede zwischen TokenRing, TokenBus und Ethercat in Bezug auf Topologie und Mediumszugriffverfahren.
- Wiederholungsfragen:
  1. Was ist der Unterschied zwischen dominanten und rezessiven Bits.
  2. Nennen Sie zwei Mechanismen zur Bitsynchronisierung und erklären Sie diese.
  3. Was ist der Unterschied zwischen CSMA/CD und CSMA/CA?
  4. Erläutern Sie zwei verschiedene Ansätze um Ethernet echtzeitfähig zu machen.
  5. Beurteilen Sie die Kommunikationsprotokolle Ethernet, CAN, TTP nach Ihrer Echtzeitfähigkeit und gehen Sie vor allem auf die Möglichkeit zur Vorhersage der maximalen Nachrichtenlatenz ein.

Klausur Wintersemester 07/08



In der Abbildung sehen Sie drei Knoten und Ihre jeweilige Nachricht für den Fall, dass der jeweilige Knoten als einziger senden würde. Dabei entspricht die Länge eines Bits einem Kästchen.

Gehen Sie davon aus, dass für die Lösung der Aufgabe alle Daten bitsynchron übertragen werden. Das JAM-Signal soll aus einer Folge von 5 0-Bits bestehen. Das 0-Bit ist dominant. Zwischen zwei Nachrichten gibt es eine Pause (interframe gap) von mindestens 3 Bits.

- Zeigen Sie für die angegebenen Nachrichten einen möglichen Ablaufplan in CSMA-CD.
- Geben Sie den entsprechenden Plan in CSMA-CA an.
- Für ein konkretes Netzwerk ist die maximale Signallaufzeit mit einer Zeiteinheit angegeben. Welche der angegebenen Bitübertragungsdauern würden Sie für CSMA/CA auswählen. Geben Sie eine knappe Begründung für Ihre Antwort.
  - 0,5 Zeiteinheiten
  - 1 Zeiteinheit
  - 4 Zeiteinheiten
  - 10 Zeiteinheiten

Kapitel 9

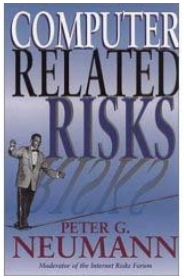
Fehlertoleranz

Inhalt

- Einleitung
- Grundlagen
- Fehlertoleranzmechanismen
- Quantitative Bewertung fehlertoleranter Systeme

Literatur

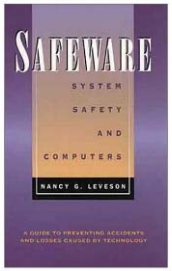
Dhiraj K. Pradhan: Fault-Tolerant Computer System Design, Prentice Hall 1996



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

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

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)

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.
  ⇒ Zerstörung der Rakete, Schaden 18,5 Millionen $

Ariane 5 (1996)

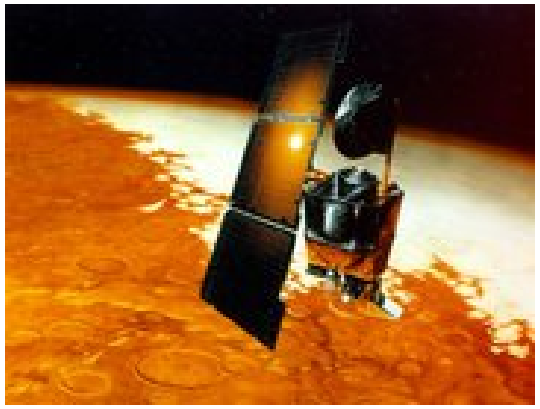
- Selbstzerstörung bei Jungfernflug:
- Design:
  - 2 redundante Meßsysteme (identische Hardware und Software) bestimmen die Lage der Rakete (hot-standby)
  - 3-fach redundante On-Board Computer (OBC) überwachen Meßsysteme
- Ablauf:
  - Beide Meßsysteme schalten aufgrund eines identischen Fehlers ab
  - 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)

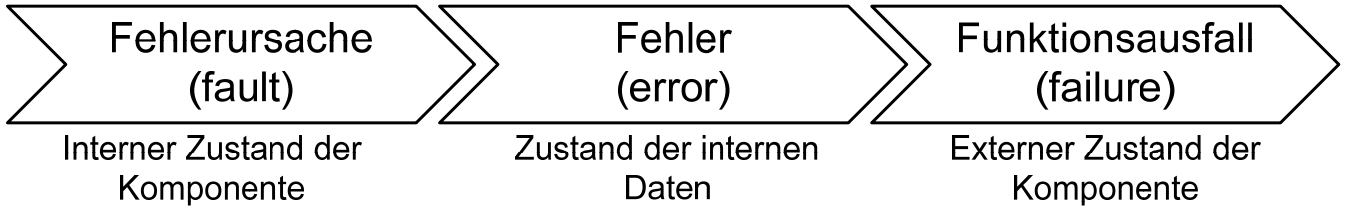
- Explosion einer holländischen Chemiefabrik aufgrund eines Bedienfehlers
- Ablauf:
  - Computergesteuertes Mischen von Chemikalien.
  - Operateur (in Ausbildung) verwechselt beim Eintippen eines Rezeptes 632 (Harz) mit 634 (Dicyclopentadien).
- Folgen:
  - Explosion fordert 3 Menschenleben, Explosionsteile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

Fehlertoleranz

Definitionen

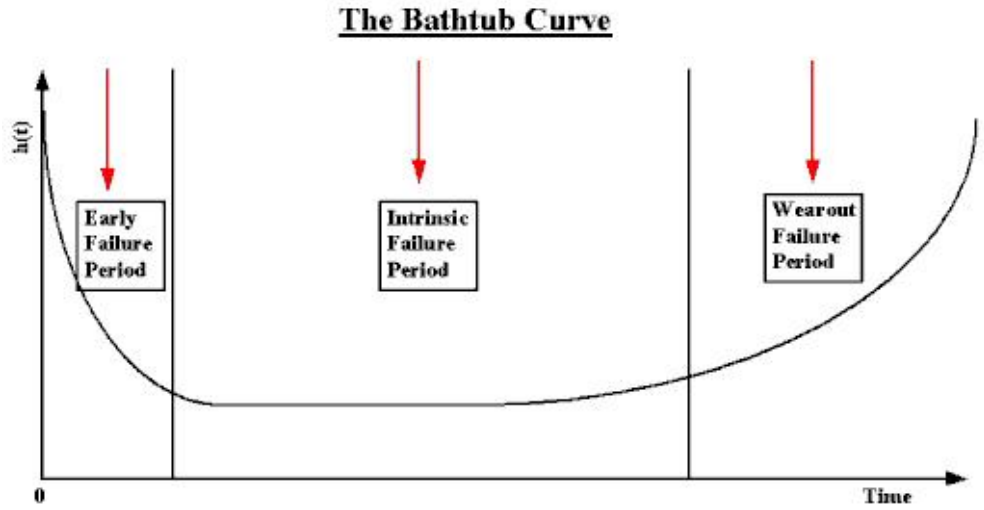
Begriffe:

- **Fehlerursache** (**fault**)**:** physikalischer Fehler oder Störstelle in einer Hardware- oder Softwarekomponente
- **Fehler** (**error**): Erscheinungsform eines Fehlzustands, z.B. durch das Abweichen eines Wertes vom erwarteten Wert in den internen Daten
- **Funktionsausfall** (**failure**): Ausfall oder fehlerhafte Durchführung von Funktionen eines Systems, Auftritt an der Benutzerschnittstelle



Fehlerrate

- Die Fehlerrate gibt die erwartete Anzahl an Fehler eines Gerätes oder eines Systems für eine gegebene Zeitperiode an.
- Typischerweise wird die Fehlerrate als konstant angenommen (siehe Badewannenkurve – gültig für Hardwarefehler) und mit λ bezeichnet. Typische Einheit der Fehlerrate ist Fehler pro Stunde.



Aspekte des Begriffs Fehlertoleranz

- Systeme zum Einsatz in sicherheitskritischen Anwendungen erfordern ein hohes Maß an Systemstabilität (**dependability**).
- Dieser Begriff umfasst:
  - Zuverlässigkeit
  - Sicherheit
  - Verfügbarkeit
  - Leistungsfähigkeit
  - Robustheit
  - Wartbarkeit
  - Testbarkeit

Zuverlässigkeit

- Definition: Die Zuverlässigkeit (**reliability**) eines Systems ist eine Funktion $0 \leq R(t) \leq 1$, definiert als die bedingte Wahrscheinlichkeit, dass das System korrekt während des Intervals $[t_0, t]$ funktioniert unter der Annahme, dass das System zum Zeitpunkt t_0 korrekt arbeitete.
- Wird eine konstante Fehlerrate angenommen, so kann die Zuverlässigkeit durch folgende Gleichung angegeben werden:

$$R(t) = e^{-(\lambda^*(t-t_0))}$$

Sicherheit

- Sicherheit (**safety**) ist die Wahrscheinlichkeit $0 \leq S(t) \leq 1$, dass ein System zum Zeitpunkt t entweder korrekt arbeitet oder seine Funktion auf eine Art und Weise beendet, so dass es nicht die Funktionsweise anderer Systeme gestört oder Menschen gefährdet werden.
- Sicherheit ist damit ein Maßstab für die Fähigkeit eines Systems auf eine sichere Art und Weise auszufallen.

Verfügbarkeit

- Verfügbarkeit (**availability**) wird als eine Funktion $0 \leq A(t) \leq 1$ über die Zeit ausgedrückt, die die Wahrscheinlichkeit angibt, dass ein System zum Zeitpunkt *t* korrekt arbeitet. Im Gegensatz zur Zuverlässigkeit wird bei der Verfügbarkeit neben der Häufigkeit der Dienstausfälle auch die Dauer der Reparaturen und Wartungsarbeiten berücksichtigt.
- Während bei der Zuverlässigkeit die Korrektheit des Systems zu allen Zeitpunkten eines gegebenen Intervalls gefordert wird, gibt die Verfügbarkeit die momentane Wahrscheinlichkeit der korrekten Ausführung des Systems an.
- Eine hohe Verfügbarkeit ist beispielsweise bei transaktionsbasierten Systemen, z.B. ein Fluglinienreservierungssystem, nötig. Wartungsarbeiten und Reparaturen sollten schnell durchgeführt werden, eine andauernde korrekte Funktion im Sinne der Zuverlässigkeit wird hingegen nicht gefordert.

Leistungsfähigkeit

- In vielen Fällen ist es möglich und sinnvoll Systeme zu konstruieren, die nach Auftreten von Hardware oder Softwarefehler in einzelnen Komponenten (siehe spätere Einführung von Fehlerbereichen) in einem degradierten Modus weiterarbeiten.
- Unter **Leistungsfähigkeit** (**performability**) wird eine Funktion $0 \leq P(L,t) \leq 1$ über der Zeit verstanden, die eine Wahrscheinlichkeit angibt, dass die Funktionalität des Systems zum Zeitpunkt t mindestens das Niveau L erreicht. Im Gegensatz zur Zuverlässigkeit, bei der immer nur die Wahrscheinlichkeit angegeben wird, dass alle Funktionen korrekt funktionieren, können nun auch Teilmengen betrachtet werden.

Robustheit, Wartbarkeit, Testbarkeit

- Unter **Robustheit** (**robustness**) eines Systems wird die Fähigkeit verstanden auch unter erschwerten Betriebsbedingungen (z.B. Fehleingaben (siehe Chemiefabrik) oder widersprüchlichen Meßwerten) die korrekte Funktionalität zu wahren.
- **Wartbarkeit** (**maintainability**) ist ein Maßstab für die Reparaturfreundlichkeit eines Systems. Quantitativ kann die Wartbarkeit als die Wahrscheinlichkeit M(t) ausgedrückt werden, dass das fehlerhafte System innerhalb einer Zeitdauer t repariert werden kann.
- **Testbarkeit** (**testability**) ist ein Maßstab für die Möglichkeit bestimmte Eigenschaften eines Systems zu testen. So kann es möglich sein, bestimmte Tests zu automatisieren und als Mechanismen in das System zu integrieren.
- Die Testbarkeit eines Systems ist durch die hohe Bedeutung der schnellen Fehleranalyse direkt mit der Wartbarkeit eines Systems verbunden.

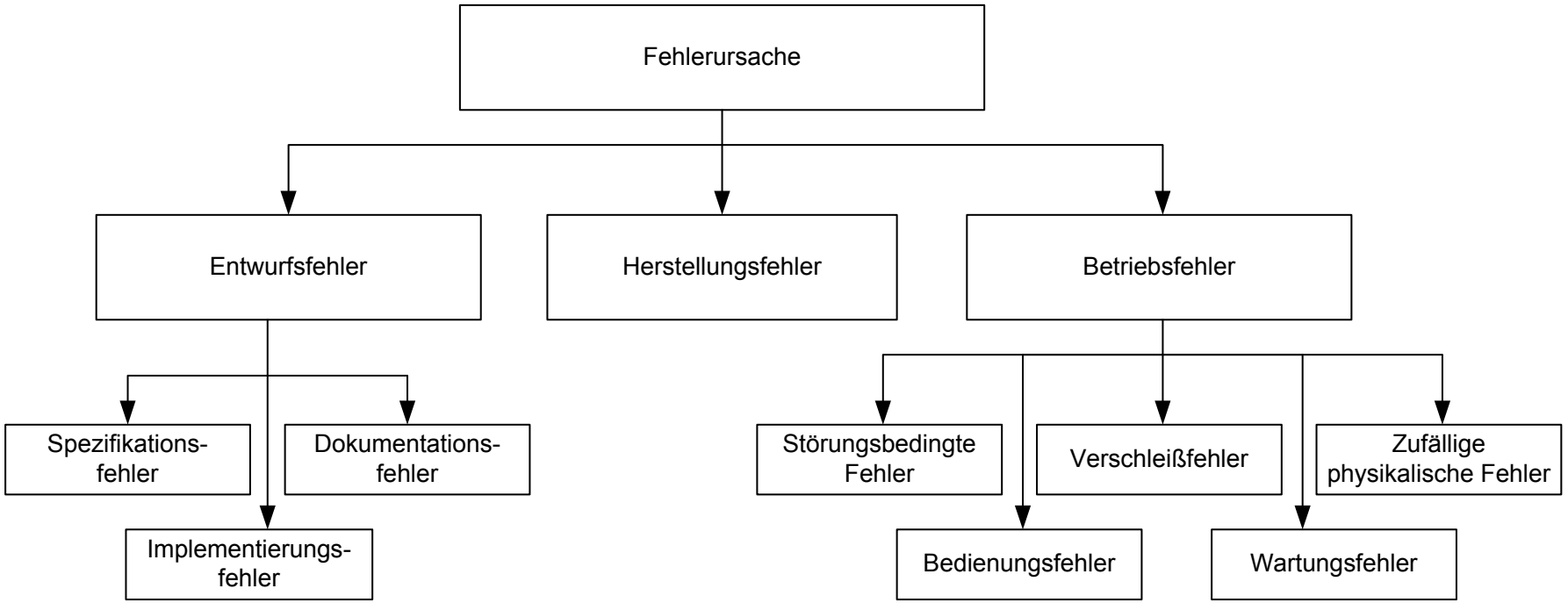
Konzepte zur Erhöhung der Systemstabilität

- Systemstabilität kann durch Anwendung der folgenden Konzepte erreicht werden:
  - Fehlervermeidung
    - Designmethoden + Werkzeugunterstützung
    - Modellierung mit Anwendung von Verifikations- und Validierungsmethoden
  - Fehlerentfernung
    - Einheitentests
    - Integrationstests
    - Back-To-Back Testing (Vergleich von Resultaten unterschiedlicher Versionen bei N-Versionsprogrammierung)
  - Fehlertoleranz

Fehlertoleranz

Fehlermodell

Fehlerursachen



Klassifizierung von Fehlern

- Unterscheidung nach Entstehungsort:
  - Hardware
  - Software

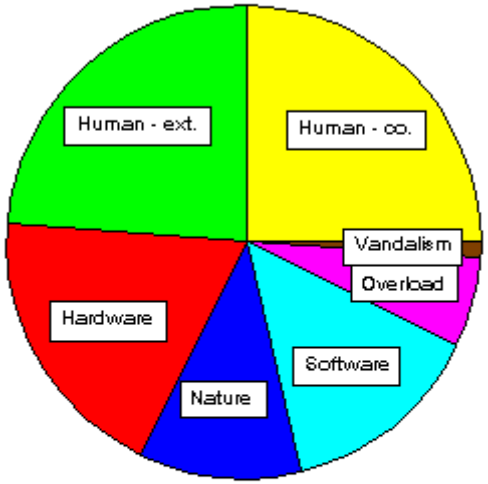
- Unterscheidung nach Fehlerdauer:
  - permanent
  - intermittierend (flüchtig)
    - periodisch
    - wiederkehrend
    - einmalig

Beispiel: Fehlerquellen im öffentlichen Telefonnetz

- Welche Ursachen können Fehler haben:
  - Fehler durch Menschen (intern/extern)
  - Hardwarefehler
  - Softwarefehler
  - Fehler verursacht durch die Natur
  - Überlast
  - Vandalismus
- Weitere Informationen unter http://hissa.ncsl.nist.gov/kuhn/pstn.html.

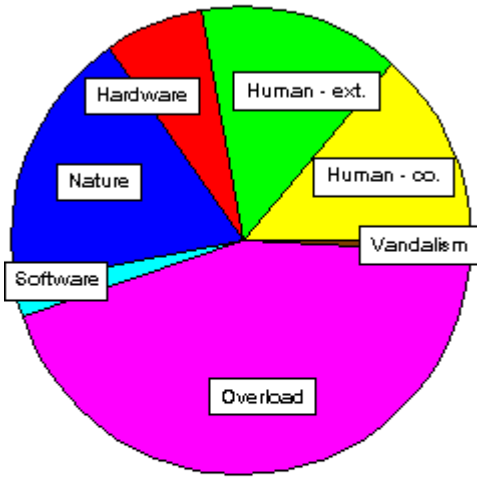
Ursachen und Wirkung

Figure 1: Number of Outages
(percent)



Human error - company: 25%
Human error - external: 24%
Hardware failure: 19%
Act of nature: 11%
Software failure: 14%
Overload - 6%
Vandalism - 1%

Figure 2: Magnitude of Failure
(customer minutes- percent of total)



Human eror - company: 14%
Human error - external: 14%
Hardware failure: 7%
Act of nature: 18%
Software failure: 2%
Overload: 44%
Vandalism - 1%

Fehlermodell

- Um die Fehlertoleranz-Fähigkeit eines Rechensystems spezifizieren zu können, ist eine Fehlervorgabe erforderlich, welche die Menge der zu tolerierenden Fehler auf ein formales Fehlermodell angibt.
- Ein Fehlermodell hat den Zweck zu jedem Zeitpunkt die Fehlermöglichkeiten eines Systems als eine Obermenge der Menge der zu tolerierenden Fehler anzugeben.
- Das Fehlermodell beinhaltet daher
  - die Komponenten, die von Fehlern betroffen sein können (strukturelle Fehlerbetrachtung) und
  - in welcher Art und Weise deren Funktion beeinträchtigt wird (funktionelle Fehlerbetrachtung)

Fehlerbereich

- Typischerweise wird angenommen, dass Fehler nur in bestimmten Teilmengen der Menge aller Komponenten *S* auftreten. Jede dieser Komponentenmengen wird als **Fehlerbereich** *Fb* bezeichnet.
- Die Annahmen
  - $Fb_1 \cup ... \cup Fb_n \neq S$ ($\Rightarrow$ es gibt einen Perfektionskern $S \setminus (Fb_1 \cup ... \cup Fb_n)$
  - $\exists i,j \in \{1...n\}: Fb_i \cap Fb_j \neq \emptyset$ ($\Rightarrow$ Überschneidungen sind erlaubt)

  sind zulässig.

k-Fehler-Annahme

- Da die Anzahl der Fehlerbereiche mitunter sehr groß werden kann, bietet sich als Spezialfall der Fehlerbereichsannahme die k-Fehler-Annahme an.
- Grundlage hierfür ist die disjunkte Zerlegung eines Systems S in Einzelfehlerbereiche $Eb_1,...,Eb_m$ mit $Eb_1 \cup ... \cup Eb_m = S$. Die k-Fehlerannahme fordert die Tolerierung von allen Fehlern, die sich auf bis zu k Einzelfehlerbereiche erstrecken.
- Die bei k-Fehler-Annahme mit $k \geq 2$ zu tolerierenden Fehlerfälle werden Mehrfachfehler genannt. Es wird jedoch nicht zwischen zufälligen und systematischen Mehrfachfehlern unterschieden. Dieser Unterschied muss jedoch bei der Anfälligkeitsanalyse genau betrachtet werden.
- Beispiel: 3-Rechner-System, als Einzelfehlerbereiche werden die einzelnen Rechner angesehen

Fehlfunktionsannahmen

- Detaillierung der Fehlervorgabe durch **Fehlfunktionsannahme**. Sinnvolle Annahmen sind:
  - Teil-Ausfall: nur manche Funktionen eines Systems fallen aus, die übrigen werden korrekt erbracht
  - Unterlassungs-Ausfall: es wird entweder ein richtiges oder gar kein Ergebnis ausgegeben (ommision fault)
  - Anhalte-Ausfall: sobald ein Fehler aufgetreten ist, gibt das System kein Ergebnis mehr aus (fail-stop) ! jedes ausgegebenen Ergebniss ist korrekt und es fehlt kein früheres Ergebnis
  - Haft-Ausfall: ab Auftreten eines Fehlers wird immer das gleiche Ergebnis ausgegeben
  - Inkonsistenz-Ausfall: ausgegebene fehlerhafte Ergebnisse sind in sich nicht konsistent (z.B. CRC)
  - Binärstellen-Ausfall (oder k-Binärstellenausfall): Fehler verfälschen maximal k Binärstellen eines Ergebnisses
  - Nicht-Angriffs-Ausfall: z.B. Schutz von fehlerfreien Komponenten vor falscher Authentifikation fehlerhafter Komponenten

Fehlerausbreitung und -eingrenzung

- Fehler breiten sich in der Regel ohne geeignete Maßnahmen innerhalb eines Systems aus. Fehlertoleranzverfahren basieren jedoch zumeist auf einer eingeschränkten Fehlervorgabe. So kann zumeist nur eine begrenzte Anzahl an fehlerhaften Komponenten toleriert werden.

⇒ Eingrenzungsmaßnahmen müssen getroffen werden.

- Typischerweise werden deshalb Maßnahmen zur Isolierung getroffen:
  - Hardwarekomponenten werden räumlich getrennt oder gekapselt.
  - Software wird so strukturiert, dass möglichst viele Berechnungen in einzelnen Modulen erfolgt.
  - An Schnittstellen werden Inkonsistenzprüfungen zwischen den einzelnen Komponenten durchgeführt.

Fehlertoleranz

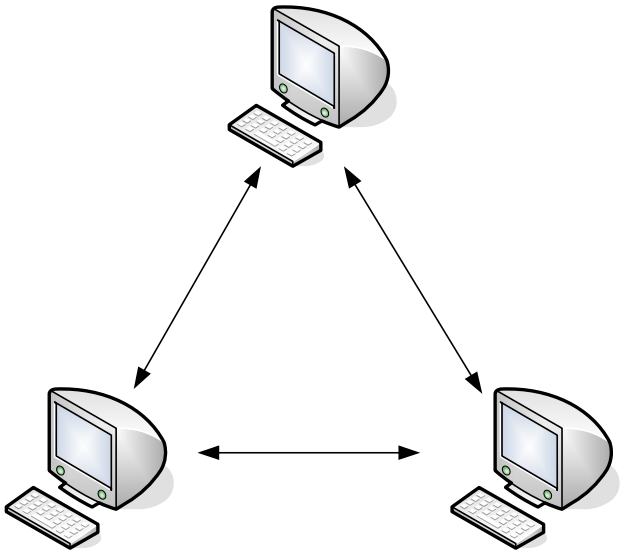
Redundanz

Grundlage der Fehlertoleranzmechanismen: Redundanz

- Die beiden grundsätzlichen Schritte eines Fehlertoleranzverfahrens, die Diagnose und Behandlung von Fehlern, benötigen zusätzliche Mittel, die über die Erfordernisse des Nutzbetriebs hinausreichen.
- All diese zusätzlichen Mittel sind unter dem Begriff **Redundanz** zusammengefasst.
- Redundanz bezeichnet also den Einsatz von mehr technischen Mitteln, als für die spezifizierte Nutzfunktion eines Systems benötigt werden.

Typische Ausprägung von Redundanz: 2-von-3-System

- Ein 2-von-3 System / TMR-System (triple modular redundacy) besteht aus 3 gleichwertigen Komponenten.
  - Ein Ausfall einer Komponente kann toleriert werden, ohne dass die Funktion beeinflusst wird.
  - Bei einem Ausfall einer zweiten Komponente muss in einen sicheren Modus geschaltet werden (nur eingeschränkt möglich).
- Betriebmodi:
  - sicherer und zuverlässiger Betrieb (2-von-3-Betrieb)
  - sicherer Betrieb (2-von-2-Betrieb)



Zuverlässigkeit redundanter Systeme

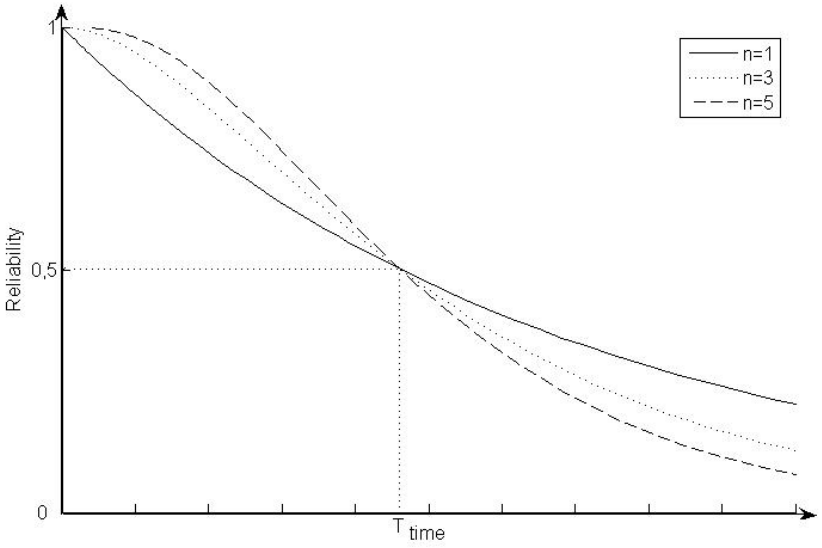
- Redundanz kann, muss aber nicht die Zuverlässigkeit verbessern:
- Beispiel: 2-von 3 System, stochastisch unabhängige Fehler, konstante Ausfallsrate λ, R_1: Zuverlässigkeit einer Komponte, R_3:Zuverlässigkeit des TMR-Systems

$$\rightarrow R_3(t) = R_1(t)^3 + 3 * R_1(t)^2 * (1 - R_1(t))$$

- Allgemeiner Fall m-von-n System:

$$\rightarrow R_n(t) = \sum_{k=m}^{n} \binom{n}{k} R_1^k * (1 - R_1)^{n-k}$$

$\Rightarrow$ ohne Möglichkeiten zur Reparatur sinkt die Zuverlässigkeit des Redundanten Systems nach einer Zeitdauer T unter die Zuverlässigkeit eines einfach ausgelegten Systems.



Redundanzarten

- Redundanz ist möglich in:
  - Hardware (strukturelle Redundanz)
  - Information
  - Zeit
  - Software (funktionelle Redundanz)
    - Zusatzfunktion
    - Diversität

⇒ Fehlertolerante Rechensysteme setzen zumeist Kombinationen verschiedener redundanter Mittel ein.

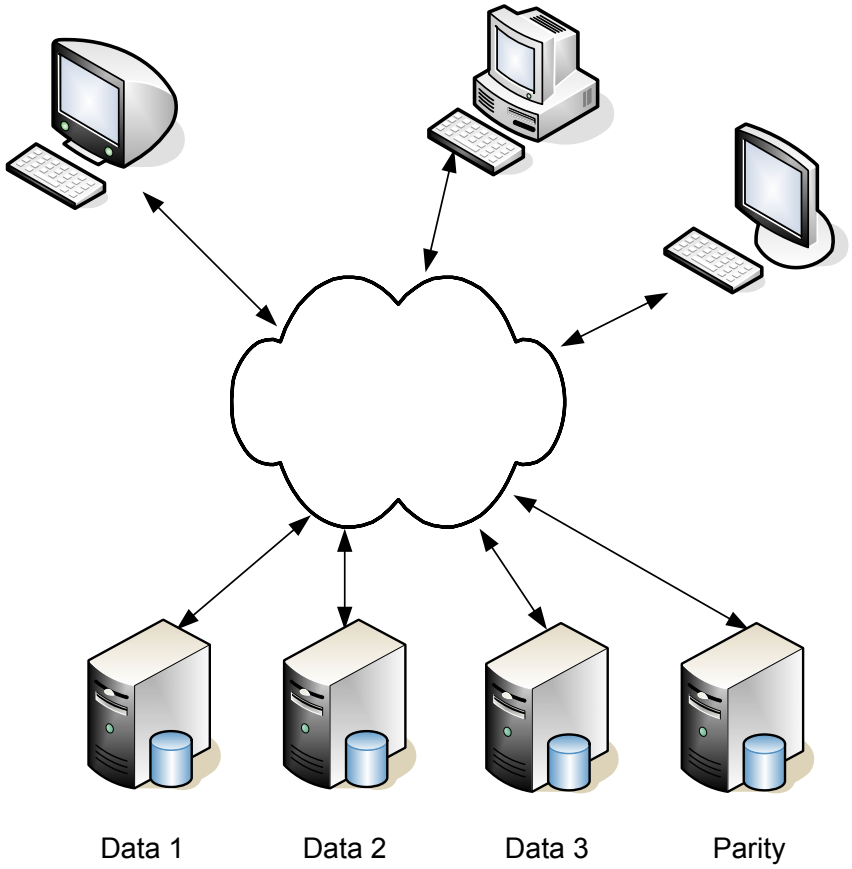
Strukturelle Redundanz

- Strukturelle Redundanz bezeichnet die Erweiterung eines Systems um zusätzliche (gleich- oder andersartige) für den Nutzbetrieb entbehrliche Komponenten.
- Beispiele:
  - 2-von-3-Rechnersysteme
  - redundante Kommunikationskanäle (siehe TTP, Flexray)
  - mehrfache Kopien einer Datei
  - unterschiedliche Sensoren

Anwendungsbeispiel:

- Verteilte Dateisysteme
  NetRAID (Lübeck),
  xFs (Berkeley)

- Ziel:
  - skalierbare Speichergröße
  - hohe Zugriffsraten
  - Ausfallstoleranz



Funktionelle Redundanz

- Funktionelle Redundanz bezeichnet die Erweiterung eines Systems um zusätzlich für den Nutzbetrieb entbehrliche Funktionen.
- Beispiele:
  - Testfunktionen
  - Funktionen zur Rekonfiguration im Mehrrechnerbetrieb
  - Erzeugung eines Paritätsbits
  - N-Versions-Programmierung

Diversität (N-Versions-Programmierung)

- Diversität bezeichnet die Erfüllung der Spezifikation einer Nutzbetriebs-Funktion durch mehrere verschiedenartig implementierte Funktionen.
- Um Entwurfsfehler in Hard- und / oder Softwaresystemen tolerieren zu können, ist der Einsatz von Diversität zwingend. Diversität verbessert die Zuverlässigkeit aber nicht unbegrenzt. Die Verbesserungsgrenze ist insbesondere von der Schwierigkeit des zu lösenden Systems vorgegeben.
- Um Diversität zu realisieren, muss der Entwurfsspielraum für die verschiedenen Varianten genutzt werden.
- Ansätze:
  - Unabhängiger Entwurf
  - Gegensätzlicher Entwurf

Beispiel aus dem Alltag: Arztbesuch

- Typisches Beispiel für das N-Versions-Konzept: Konsultation von verschiedenen Ärzten:
  - Unterschiedliche Spezialisierungen
  - Unterschiedliche Untersuchungen
  - Unterschiedliche Behandlungsmethoden

Informationsredundanz

- Informationsredundanz bezeichnet zusätzliche Informationen neben der Nutzinformation.
- Beispiele:
  - Fehlerkorrigierende Codes
  - Doppelt verkettete Listen
  - Paritätsbits
  - CRC: cyclic redundancy check
- Voraussetzung: Fehler dürfen sich nur auf einen beschränkten Teil der gesamten Information auswirken (z.B. Fehlfunktions-Annahme)

Cyclic Redundancy Check

- Nachrichten werden als Polynome interpretiert, korrekte Nachrichten müssen ein Vielfaches vom Generatorpolynom G(u) sein.
- Beispiel zur Berechnung der Kontrollstellen: $k = 3$, $G(u) = u^3 + u^1 + 1$, $m = 4$ Nachrichtenstellen, $n = k + m$ Gesamtstellen.
- Seien als Nachrichtenstellen gewählt: (**1001**). Also Codewort: (**1001???**).
- Polynomdivision:

$$\overbrace{(u^6\ u^5\ u^4\ u^3\ u^2\ u^1\ u^0)}^{X(u)} \ / \ \overbrace{(u^3\ u^2\ u^1\ u^0)}^{G(u)} = \overbrace{(u^3\ u^2\ u^1\ u^0)}^{Q(u)}$$

$$\begin{array}{rrrrrrrcrrrrcrrrr}
1 & 0 & 0 & 1 & 0 & 0 & 0 & / & 1 & 0 & 1 & 1 & = & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 1 & & & & & & & & & & & & \\
\cline{1-4}
 & 0 & 1 & 0 & 0 & & & & & & & & & & & \\
 & 0 & 0 & 0 & 0 & & & & & & & & & & & \\
\cline{2-5}
 & & 1 & 0 & 0 & 0 & & & & & & & & & & \\
 & & 1 & 0 & 1 & 1 & & & & & & & & & & \\
\cline{3-6}
 & & & 0 & 1 & 1 & 0 & & & & & & & & & \\
 & & & 0 & 0 & 0 & 0 & & & & & & & & & \\
\cline{4-7}
 & & & & 1 & 1 & 0 & \longleftarrow & \text{Rest der Division} & & & & & & &
\end{array}$$

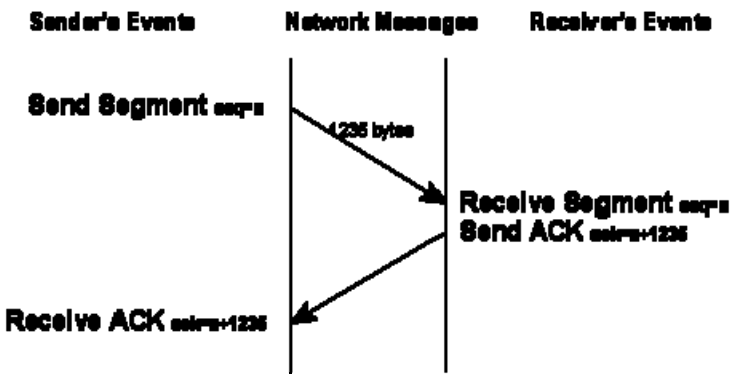
- Also Codewort: (**1001000**) – (**0000110**) = (**1001110**)

Zeitredundanz

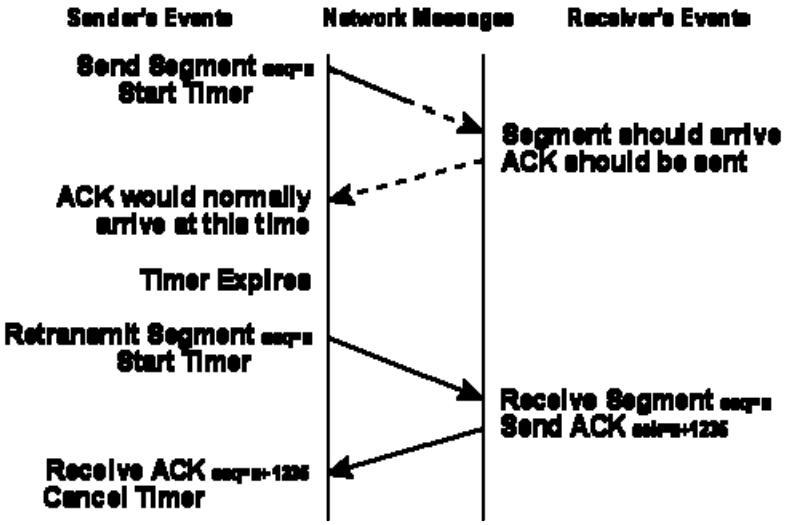
- Zeitredundanz bezeichnet über den Zeitbedarf des Normalbetriebs hinausgehende zusätzliche Zeit, die einem funktionell redundantem System zur Funktionsausführung zur Verfügung steht.
- Beispiele:
  - Wiederholungsbetrieb
  - Zeitbedarf für Konsistenzmechanismen in verteilten Dateisystemen

Beispiel: TCP

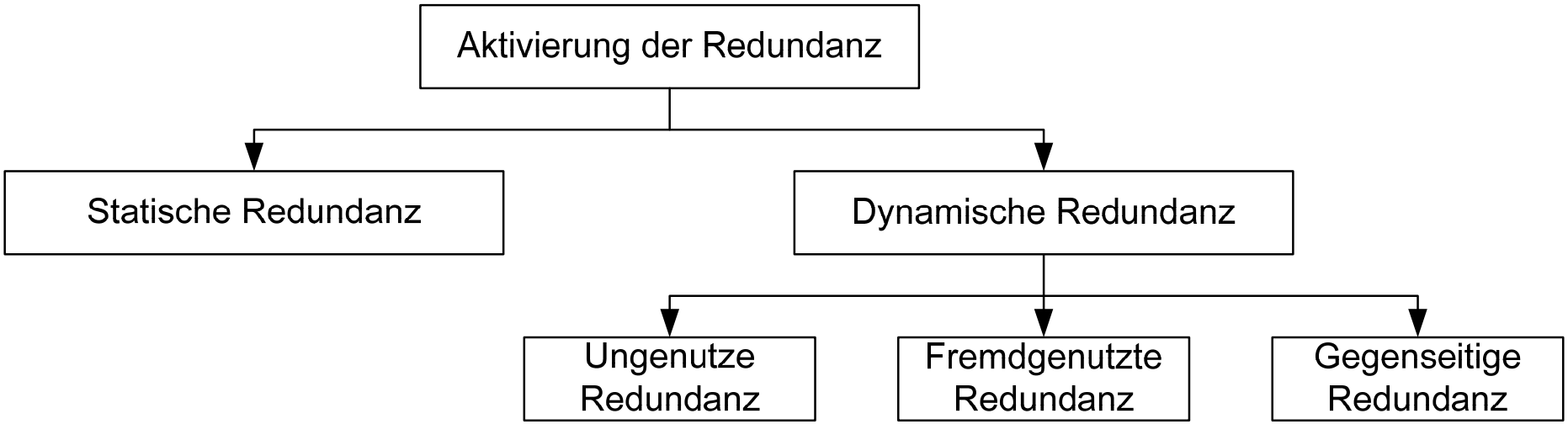
Fehlerfrei



Verlust einer Nachricht



Aktivierung der Redundanz



Statische Redundanz

- Statische Redundanz bezeichnet das Vorhandensein von redundanten Mitteln, die während des gesamten Einsatzzeitraums aktiv zu den zu unterstützenden Funktionen beitragen.
- Ausprägungen:
  - Statische strukturelle Redundanz: z.B. n-von-m System
  - Statische funktionelle Redundanz (Zusatzfunktionen): z.B. doppeltes Senden von Nachrichten auf unterschiedlichen Wegen
  - Statische funktionelle Redundanz (Diversität): N-Versions-Programmierung
  - Statische Informationsredundanz: fehlerkorrigierende Codes
  - Statische Zeitredundanz: statische Mehrfachausführung einer Funktion

Dynamische Redundanz

- Dynamische Redundanz bezeichnet das Vorhandensein von redundanten Mitteln, die erst im Ausnahmebetrieb (d.h. nach Auftreten eines Fehlers) aktiviert werden, um zu den zu unterstützenden Funktionen beizutragen.
- Typisch für dynamisch strukturelle Redundanz ist die Unterscheidung in **Primärkomponenten** und **Ersatzkomponenten**. Die Dauer der Umschaltung hängt im Wesentlichen von den ggf. erforderlichen Vorbereitungsmaßnahmen der Ersatzkomponenten ab. Hier wird zwischen heißer Reserve (**hot stand-by**) und kalter Reserve (**cold stand-by**) unterschieden.
- Die Definition verlangt kein vollkommen passives Verhalten. Folgende Szenarien sind möglich:
  - ungenutzte Redundanz: Ersatzkomponenten sind bis zur fehlerbedingten Aktivierung passiv
  - fremdgenutzte Redundanz: Ersatzkomponenten erbringen nur Funktionen, die von den zu unterstützenden Funktionen verschieden sind und im Fehlerfall storniert werden
  - gegenseitige Redundanz: Komponenten stehen sich gegenseitig als Reserve zur Verfügung. Im Fehlerfall übernimmt eine Komponente die Funktionen der anderen zusätzlich zu den eigenen.

Fehlertoleranz

Fehlertoleranzmechanismen

Grundlage: Fehlererkennung

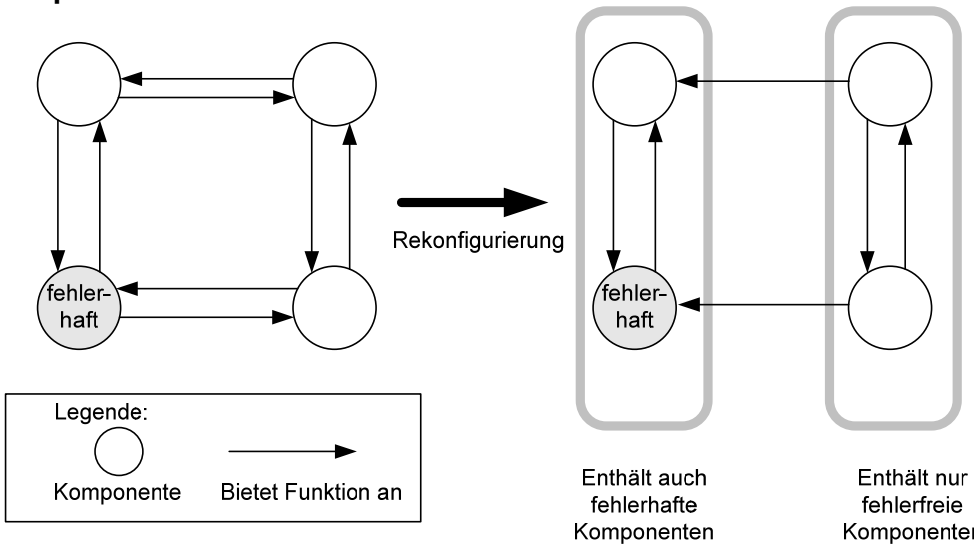
- Grundlage der Fehlertoleranzmechanismen ist die Fehlerdiagnose.
- Ziele der Fehlerdiagnose ist:
  - das Erkennen von Fehlern (im Nutzbetrieb)
  - die Lokalisierung von Fehlern (zumeist im Ausnahmebetrieb)
  - die Bestimmung des Behandlungsbereichs (zumeist im Ausnahmebetrieb)

Fehlererkennung

- Möglichkeiten zur Fehlererkennung:
  - Zeitschrankenüberwachung
  - Absoluttests: getestet wird direkt das Ergebnis (z.B. Anzahl der Elemente muss nach Sortieren gleich der eingegebenen Anzahl sein)
  - Relativtests: Vergleich von mehreren Ergebnissen redundanter Prozesse
    - bei deterministischen Prozessen
    - bei nicht deterministischen Prozessen
  - Nutzung von Informationsredundanz (z.B. CRC)

Rekonfigurierung

- Durch Rekonfigurierung werden fehlerhafte Komponenten ausgegrenzt und bestehende Funktionszuordnungen zwischen fehlerhaften und fehlerfreien Komponenten aufgelöst.
- Nach einer Rekonfigurierung ist das System in zwei Komponenten-Teilmengen partitioniert: eine enthält nur fehlerfreie, die andere auch fehlerhafte Komponenten.



Rekonfigurierung: Beitrag zur Fehlertoleranz

- Rekonfigurierung dient zur Behandlung von Funktionsausfällen, nicht aber der Behebung von Fehlzuständen.

  $\Rightarrow$ nicht ausreichend für erfolgreiche Fehlerbehandlung

  $\Rightarrow$ Verfahren zur Fehlerbehebung (Rückwärts-, Vorwärtsbehebung) oder Fehlerkompensierung (Fehlermaskierung, Fehlerkorrektur) müssen hinzukommen.

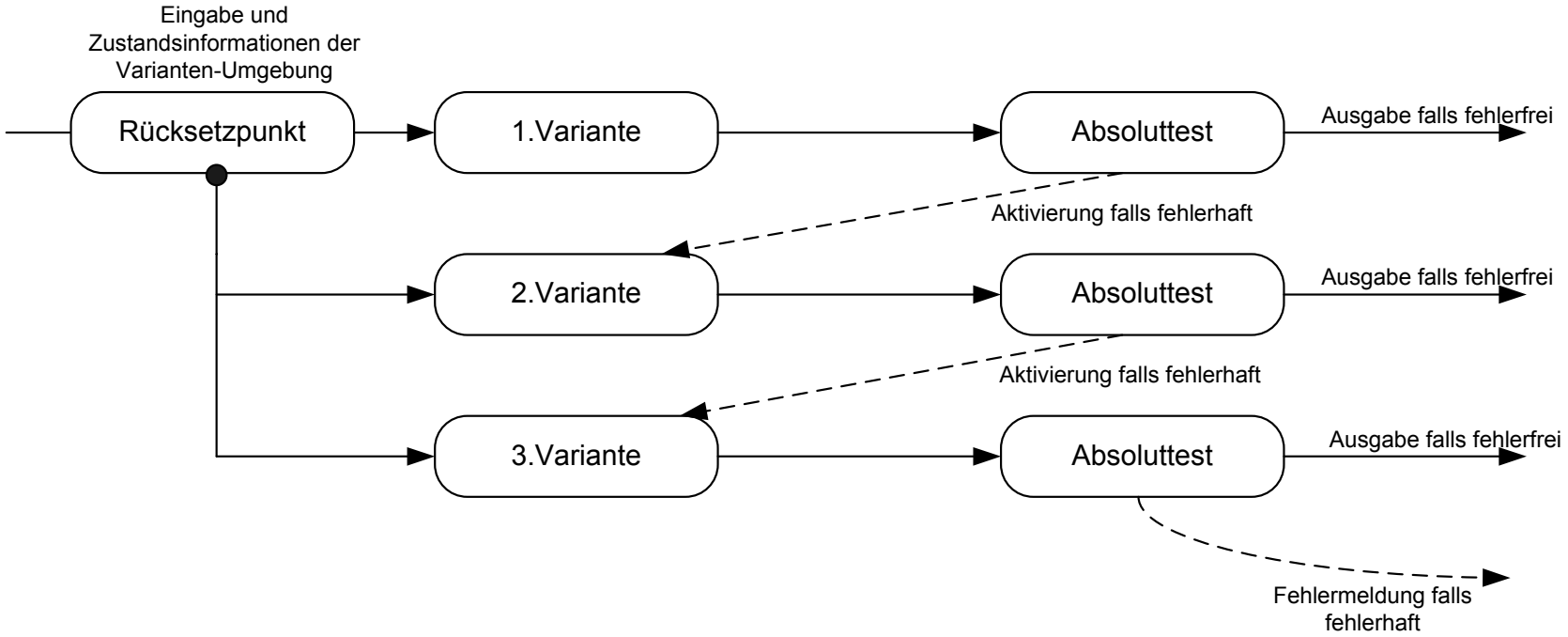
Rückwärtsbehebung (backward recovery)

- Rückwärtsbehebung versetzt Komponenten in einen Zustand, den sie bereits in der Vergangenheit angenommen hatten oder als konsistenten Zustand hätten annehmen können.
- „Konsistent“ bedeutet, dass die lokalen Komponentenzustände und die aktuellen Interaktionen mit anderen Komponenten die (Protokoll- bzw. Dienst-) Spezifikation nicht verletzen.
- Rückwärtsbehebung ist bei intermittierenden Fehlern ausreichend, bei permanenten Fehlern ist sie als Ergänzung zur Rekonfigurierung zu sehen.
- Bei reiner Rückwärtsbehebung kann die Fehlererkennung nur über Absoluttests (da keine redundanten Ergebnisse vorhanden) erfolgen. Diese Tests werden periodisch oder ereignisabhängig durchgeführt.

Rücksetzpunkte (recovery points)

- Nach Auftreten eines Fehlers lassen sich Zustandsinformationen aus der Zeit vor Auftreten eines Fehlers nur gewinnen, wenn die Informationen zuvor kopiert wurden und an einem getrennten Ort zwischengespeichert wurden. Die abgespeicherte Zustandsinformation wird als Rücksetzpunkt bezeichnet.
- Rücksetzpunkte werden periodisch oder ereignisbasiert erstellt und verursachen also schon im Normalbetrieb einen Extrazeitaufwand.
- Zumeist finden vor der Rücksetzpunkterstellung Absoluttests statt.
- Auch Rücksetzpunkte können fehlerhaft sein (Auftreten eines Fehlers direkt nach Absoluttest bzw. beim Speichern des Rücksetzpunktes, Fehlererkennung mit einer Wahrscheinlichkeit <1). Deshalb muss das System eventuell mehrfach zurückgesetzt werden.

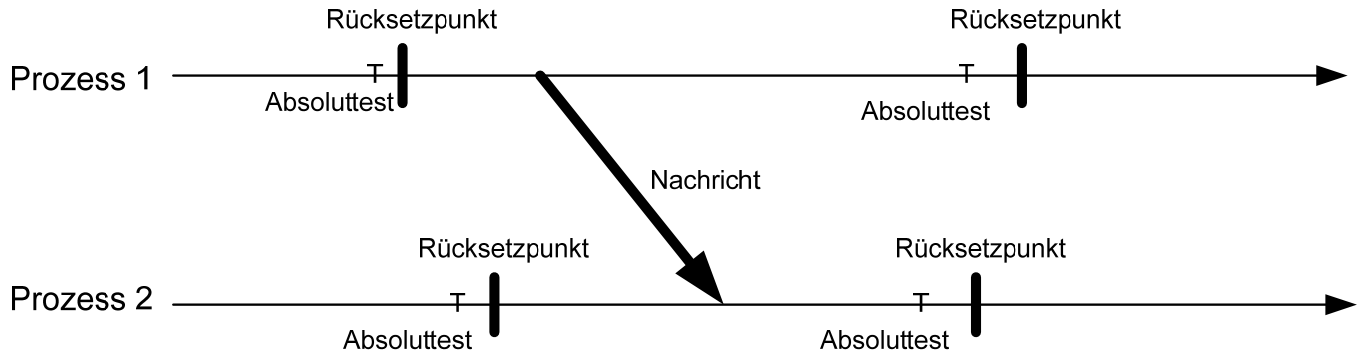
Rückwärtsbehebung diversitärer Systeme



Dieses Verfahren entspricht dem Ausprobieren mehrerer Funktionen. Z.B. das Testen von verschiedenen Browsern (bis Benutzer ein Programm gefällt, die Anzeige einer Internetseite korrekt ist).

Rücksetzlinien bei interagierenden Prozessen

- Betrachten wir folgendes Beispiel:



- Tritt bei einem der beiden Prozesse zwischen den jeweiligen Rücksetzpunkten ein Fehler auf, so müssen beide Prozesse zurückgesetzt werden.
  Warum?
- Definition: Die Menge der Rücksetzpunkte, auf die mehrere Prozesse zugleich zurückgesetzt werden können, heißt Rücksetzlinie (recovery line).
- Mögliche Probleme: Dominoeffekt

Vor- und Nachteile der Rückwärtsbehebung

- Rückwärtsbehebung verwendet die vorhandenen Betriebsmittel sparsam.
- Im Fehlerfall lässt sich ein Prozess wiederholt zurücksetzen (solange Rücksetzpunkte vorhanden), dadurch erhöht sich die Menge der tolerierenden Fehler.
- Wiederholungsbetrieb erfordert nicht zwangsläufig die gleichen Eingaben wie der zuvor erfolgte Nutzbetrieb (Nicht-Determinismus zulässig).
- Rückwärtsbehebung ist transparent (unabhängig von der Anwendung) implementierbar.
- Nur Absoluttests, keine Relativtests anwendbar.
- Menge der tatsächlichen tolerierten Fehler ist wegen der Abhängigkeit von verschiedenen Absoluttest-Algorithmen kaum formal spezifizierbar.
- Rücksetzpunkterstellung kostet schon im Normalbetrieb.
- Der im Fehlerfall erforderliche Wiederholungsbetrieb kann Zeitredundanz in beträchtlichem Umfang fordern.

Vorwärtsbehebung

- **Vorwärtsbehebung** bezeichnet Fehlerbehebungs-Verfahren, die keine Zustandsinformationen der Vergangenheit verwenden.
- Basis dieser Verfahren sind Fehlfunktions-Annahmen und anwendungsspezifisches Wissen.
- **Beispiel**: Geht aufgrund eines Fehlers in einem Rechner ein zuvor gelesener Temperaturwert verloren, so kann er durch zweimaliges Einlesen der aktuellen Temperatur und Extrapolation näherungsweise zurückgewonnen werden (Voraussetzung: zeitliche Ableitung der Temperatur ändert sich kaum).

Vor- und Nachteile der Vorwärtsbehebung

- Aufwand an struktureller Redundanz ist gering: nur Absoluttests und die erst im Ausnahmebetrieb zu aktivierenden Ausnahmebehandler sind hinzuzufügen
- Laufzeitaufwand im Normalbetrieb wird nur von Absoluttests verursacht und ist daher minimal
- Vorwärtsbehebung ist nicht transparent sondern anwendungsabhängig
- hoher Entwurfsaufwand
- Gelingen hängt stark vom Schwierigkeitsgrad der Anwendung ab
- Nur durch Absoluttests erkennbare Fehler sind überhaupt tolerierbar
- Oft nur degradierter Betrieb nach Vorwärtsbehebung möglich

Fehlermaskierung

- Das Verfahren der Fehlermaskierung berechnet aus redundant berechneten Ergebnissen ein korrektes Ergebnis zur Weitergabe.
- Typischerweise ist die „Maske“ durch einen Mehrheitsentscheider realisiert, der Ergebnisse durch einen Relativtest vergleicht. Dieses Verfahren toleriert fehlerhafte Ergebnisse solange diese in der Minderheit bleiben.
- Typische Ausprägungen sind 2-von-3-Systeme oder 3-von-5 Systeme.

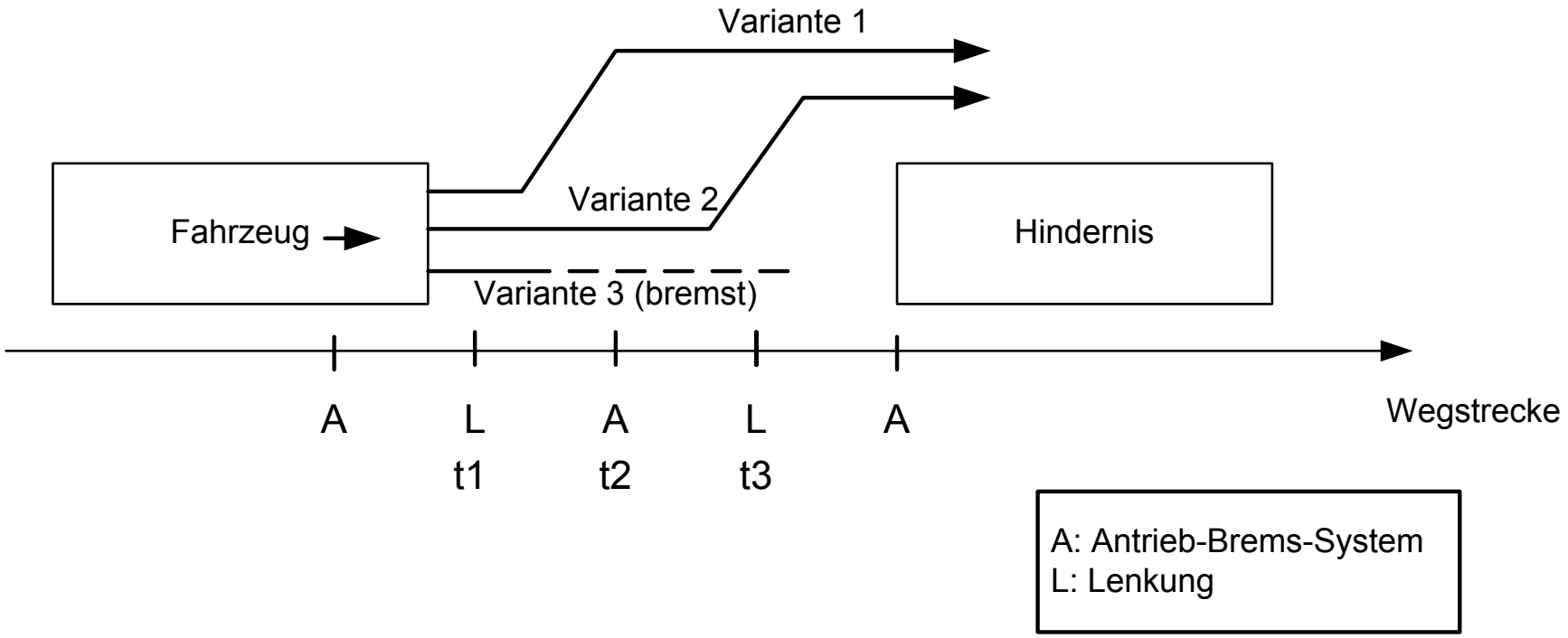
Maskierungsentscheidungen/Votierung

- für **deterministische** Prozesse:
  - Mehrheitsentscheidung: Mehrheit der Gesamtanzahl von Komponenten nötig
  - Paarentscheidung: Annahme, dass fehlerhafte Ergebnisse nie gleich sind zwei übereinstimmende Ergebnisse sind immer korrekt (Reduzierung der Anzahl nötiger Vergleiche)
  - Meiststimmenentscheidung
  - Einstimmigkeitsentscheidung: alle Komponenten müssen im Ergebnis übereinstimmen (Erhöhung der Sicherheit, gleichzeitig wird Zuverlässigkeit des Systems gesenkt)
- für **nicht deterministische** Prozesse:
  - Medianentscheidung: Mittleres Ergebnis wird für die Ausgabe übernommen
  - Intervallentscheidung: Annahme, dass korrekte Ergebnisse in einem Intervall liegen, ein Ergebniswert aus
  - Intervall wird gewählt
  - Kugelentscheidung: wie Intervallentscheidung nur mit mehrdimensionalen Ergebnissen, statt Intervall wird nach kürzesten Abständen gesucht

Vor- und Nachteile der Fehlermaskierung

- Fehlermaskierung reicht als **einziges** Fehlertoleranz-Verfahren aus.
- Maskierer lassen sich vergleichsweise einfach implementieren.
- Wiederholungsbetrieb entfällt, dadurch ist die Fehlerbehandlung schneller.
- Fehlerhafte Subsystemexemplare dürfen beliebiges fehlerhaftes Verhalten zeigen, da Relativtests angewandt werden.
- Fehlermaskierung ist transparent zu implementieren.
- Hoher Aufwand durch strukturelle Redundanz

Probleme bei Fehlermaskierung diversitärer Systeme



Erläuterung des Beispiels

- Das System berechnet zu unterschiedlichen Zeitpunkten neue Werte für Antrieb (Bremsen, Beschleunigen, konstant fahren) und Lenkung (gerade, links, rechts).
- Zum Zeitpunkt t1 wird ein neuer Wert für die Lenkung berechnet: alle Varianten entscheiden sich für geradeaus fahren.
- Zum Zeitpunkt t2 wird ein neuer Wert für den Antrieb berechnet: die Mehrheit (Variante 1 und 2) entscheiden sich für konstant fahren.
- Zum Zeitpunkt t3 wird ein neuer Wert für die Lenkung berechnet: die Mehrheit (Variante 1 und 3) entscheidet sich für geradeaus fahren.

  $\Rightarrow$ es kommt zur Kollision
- Forderung: Varianten, die überstimmt wurden, müssen für eine bestimmte Zeit ausgeschlossen werden.

Synchronisation von redundanten Einheiten

- Alternativen:
  - Gesteuerte Synchronisierung: Steuerung von zentraler Stelle
  - Geregelte Synchronisierung: Synchronisation durch Maskierer
  - Implizite Synchronisierung: anwendbar falls die Auftragsrate die Bearbeitungsrate stets unterschreitet oder Ergebnisse verschiedener Aufträge vergleichbar sind

Reparatur und Integration von redundanten Einheiten

- Wie bereits gesehen sinkt die Zuverlässigkeit eines redundanten Systems nach einer bestimmten Zeitdauer immer unter die Zuverlässigkeit eines einfach ausgelegten Systems falls keine Reparaturen möglich sind.
- Zuverlässige Systeme mit langen geplanten Betriebszeiten müssen deshalb Reparaturen unterstützen.
- Ablauf:
  - Erkennen eines Fehlers
  - Ausgliedern der fehlerhaften Einheit
  - Zeitunkritische Durchführung von Fehlerdetektions- und Fehlerbehebungsalgorithmen
  - Wiedereingliederung (Integration)
- Wie kann sich eine Einheit wieder integrieren (state synchronisation) ohne den normalen Betrieb zu stören?

Fehlerkorrektur

- **Fehlerkorrektur** bildet einzelne fehlerhafte Ergebnisse, die genügend Informationsredundanz enthalten auf fehlerfreie ab.
- Basis ist eine Einschränkung in der Fehlfunktionsannahme (zumeist k-Binärstellen-Ausfall)

  $\Rightarrow$ Anwendungsbereich vor allem, wo physikalische Gesetze diese Annahme rechtfertigen:

  – bei der Übertragung und

  – bei der Speicherung von Daten

Vor- und Nachteile der Fehlerkorrektur

- Strukturelle Redundanz und Zeitredundanz werden nur im geringen Umfang benötigt.
- Die erforderliche Informationsredundanz ist im Allgemeinen mit geringen Aufwand zu erzeugen und zu überprüfen, allerdings können die dadurch verbundenen Kosten (z.B. zur Speicherung bzw. Senden) dem Einsatz entgegenstehen:
  - können die Daten etwa durch erneutes Senden wieder hergestellt werden, so wird häufig auf die Fehlerkorrektur verzichtet und ausschließlich Fehlererkennungsalgorithmen eingesetzt (z.B. CRC)
  - ist die Datenwiederherstellung nicht möglich und handelt es sich um wichtige Daten werden Fehlerkorrekturmaßnahmen verwendet (z.B. RAID)
- Bezüglich der Fehlervorgabe weist der Absoluttest eine hohe Fehlererfassung auf.
- Fehlerkorrektur lässt bei der Fehlervorgabe keine beliebigen Ergebnisverfälschungen zu.
- Im Allgemeinen kein geeignetes Mittel um Entwurfsfehler zu korrigieren.

Fehlertoleranz

Zusammenfassung

Lerninhalt Fehlertoleranz

- Wesentliche Elemente / Phasen der Fehlertoleranz
  - Fehlererkennung
  - Fehlerdiagnose
  - Fehlerbehandlung / Fehlerbehebung (Reparatur)
- Die Auswahl und Realisierung der Fehlertoleranzmechanismen basiert immer auf der Fehlerhypothese (definiert Menge und Art der zu tolerierenden Fehler)
- Kenntnis der Mechanismen und Vergleich in Bezug auf zu tolerierende Fehler und Echtzeitfähigkeit

Ausblick

Ausblick

- Seminare:
  - Modellbasierte Entwicklung eingebetteter Systeme
  - Sensornetzwerke
  - Software Engineering für eingebettete Multicore-Systeme
- Praktikum:
  - Echtzeitsysteme
- Studienarbeiten (SEP, BA, MA, DA, Guided Research):
  - Sensornetzwerke
  - Mikroprozessorprogrammierung
  - Elektrofahrzeug
  - Fehlertolerante Systeme
- Studentische Hilfskräfte / Promotion (Lehrstuhl bzw. fortiss)