Echtzeitsysteme
Lehrstuhl Informatik VI - Robotics and Embedded Systems

Echtzeitsysteme
Wintersemester 2013/2014

Dr. Christian Buckl

fortiss GmbH

Lehrstuhl VI Robotics and Embedded Systems

fortiss

Überblick

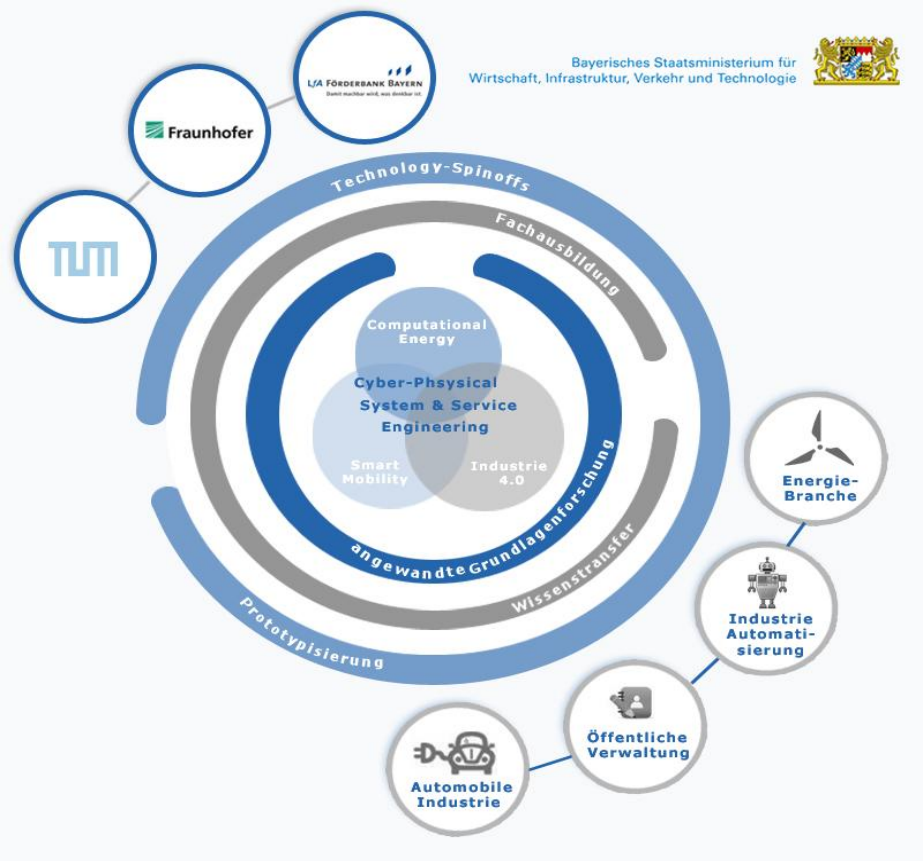
Forschungs- und Transferinstitut assoziiert mit der **Technischen Universität München**
In der Rechtsform einer **gemeinnützigen GmbH**.

Gesellschafter

TUM K.d.ö.R.
Fraunhofer Gesellschaft
LfA Förderbank Bayern

Wissenschaftliche Direktoren

Prof. Dr. Dr. h.c. Manfred Broy
Prof. Dr. Alois Knoll
Prof. Dr. Helmut Krcmar



fortiss

Organisation

DIREKTORIUM

Prof. Dr. Dr. h.c. Manfred Broy (Sprecher)
Prof. Dr. Helmut Krcmar
Prof. Dr. Alois Knoll

GESCHÄFTSLEITUNG

Dr. Harald Rueß
Dr. rer. nat. habil. Bernhard Schätz (Stellv.)

FINANZEN UND PERSONAL

Ines Lenz

FB I

SOFTWARE & SYSTEMS ENGINEERING

Prof. Dr. Dr. h.c. Manfred Broy
Dr. rer. nat. habil Bernhard Schätz

FB II

CYBER-PHYSICAL SYSTEMS

Prof. Dr. Alois Knoll
Dr. rer. nat. Christian Buckl

FB III

INFORMATION SYSTEMS &SERVICES

Prof. Dr. Helmut Krcmar
Dr. Petra Wolf

FB IV

INDUSTRIAL AUTOMATION

N.N.
Dr. Alois Zoitl

FB V

EMBEDDED SYSTEMS SOFTWARE ENGINEERING INSTITUTE

Prof. Manfred Hajek
Werner Burger (Industrie)

EIT ICT LABS

Satellite Co-Location Center Munich

Echtzeitsysteme: Organisation

Team

Dr. Christian Buckl

Philipp Heise

Übungen: Rafael Hostettler, Martin Eder, Manuel Schiller, Hardik Shah, Mohammadali Nasseri

Homepage der Vorlesung mit Folien, Übungsaufgaben und weiterem Material:
http://www6.in.tum.de/Main/TeachingWs20123chtzeitsysteme

Bestandteile der Vorlesung

- Vorlesung:
  - Dienstag 10:15-11:45 Uhr MI HS 2
  - Mittwoch 10:05-10:50 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
  - Termine werden Anfang nächster Woche angekündigt
  - Beginn: voraussichtlich ab 28.10.2013, Anmeldung ab Anfang über TUMonline (http://www.tumonline.de, siehe Tutorübung zu Echtzeitsysteme)
- Prüfung:
  - Schriftliche Klausur am Ende des Wintersemesters, falls ein Schein benötigt wird.

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 verstanden werden
- Ihre Beteiligung ist uns wichtig: bitte fragen Sie, wenn etwas unklar ist, beteiligen Sie sich an Übungen und wenn für Sie interessant: halten Sie auch selbst einmal die Vorlesung
- Zur Verdeutlichung theoretischer Inhalte wird versucht, Analogien zum Alltag herzustellen. Wichtig: Praktische Aufgaben in der Vorlesung und der Übung
- In jedem Kapitel werden die relevanten Literaturhinweise referenziert
- Zur Erfolgskontrolle werden Klausuraufgaben der letzten Jahre am Ende eines Kapitels diskutiert
- Folien werden in der Regel kurz vor Beginn der Vorlesung auf die Webseite gestellt
- **Wir freuen uns jederzeit über Fragen, Verbesserungsvorschläge und konstruktive Kommentare!**

Informationen zur Übung

- Ziel: Praktisches Einüben von Vorlesungsinhalten
- Übungsaufgaben werden in Gruppen zu zweit am Computer gelöst
- Platz ist begrenzt (8 Computer, 16 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 **TUMonline** (Tutorübungen zu Echtzeitsysteme (IN2060))
- Falls Bedarf an weiteren Terminen besteht, senden Sie bitte eine Mail an Philipp Heise (heise@in.tum.de).
- Die Übungsinhalte sind nicht direkt prüfungsrelevant, **tragen aber stark zum Verständnis bei.**
- **Ihre Rückmeldung ist wichtig, denn sie bestimmt über die Inhalte!**

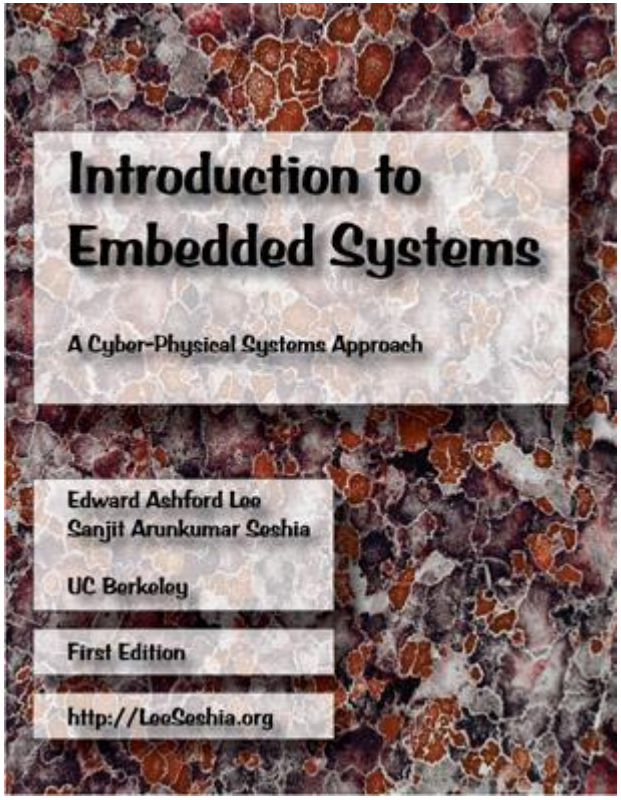
Klausur

- 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 (Rückmeldung mit Prüfungsamt steht noch aus)
- Voraussichtlich erlaubte Hilfsmittel: keine

Weitere Angebote des Lehrstuhls

- Weitere Vorlesungen: Robotik, Digitale Signalverarbeitung, Maschinelles Lernen und bioinspirierte Optimierung I&II, Sensor- und kamerageführte Roboter
- Praktika: Microcontrollerprogrammierung, 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



- Lee, Seshia: Introduction to Embedded Systems
  - Buch deckt die meisten Kapitel der Vorlesung (bis auf Kommunikation) ab
  - Kostenlos online verfügbar unter http://leeseshia.org/

Weitere Literatur

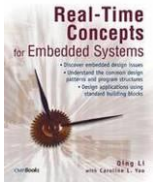
Weitere Literaturangaben befinden sich in den jeweiligen Abschnitten.

Hermann Kopetz: Real-Time Systems (Überblick)

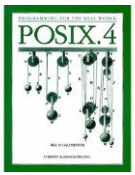
Jane W. S. Liu: Real-Time Systems
(Überblick, Schwerpunkt Scheduling)

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)

Vorlesungsinhalte

1. Einführung Echtzeitsysteme
2. Uhren
3. Modellierung und Werkzeuge
4. Nebenläufigkeit
5. Scheduling
6. Kommunikation
7. Echtzeitbetriebssysteme
8. Programmiersprachen
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 Uhren (1 Vorlesungswoche)
  - Uhren
  - Synchronisation von verteilten Uhren

Inhalt II

- Kapitel Modellierung/Werkzeuge (ca. 1-2 Vorlesungswochen)
  - Allgemeine Einführung
  - Grundsätzlicher Aufbau, Models of Computation, Ptolemy
  - Synchrone Sprachen (Esterel, Lustre), SCADE, EasyLab
  - Zeitgesteuerte Systeme: Giotto, FTOS, TTA
- Kapitel Nebenläufigkeit (ca. 3 Vorlesungswochen)
  - Prozesse, Threads
  - Interprozesskommunikation
  - Scheduling Probleme

Inhalt III

- Kapitel Scheduling (ca. 2 Vorlesungswochen)
  - Kriterien
  - Planung Einrechner-System, Mehrrechnersysteme
  - EDF, Least Slack Time, Scheduling mit Prioritäten (FIFO, Round Robin)
  - Scheduling periodischer Prozesse
- Kapitel Echtzeitfähige Kommunikation (ca. 1 Vorlesungswoche)
  - Token-Ring
  - CAN-Bus
  - TTP, FlexRay
  - Real-Time Ethernet

Inhalt IV

- Kapitel Echtzeitbetriebssysteme (ca. 1 Vorlesungswoche)
  - QNX, VxWorks, PikeOS
  - RTLinux, RTAI, Linux Kernel 2.6
  - TinyOS, eCos
  - OSEK
- Kapitel Fehlertoleranz (ca. 2 Vorlesungswochen)
  - Bekannte Softwarefehler
  - Definitionen
  - Fehlerarten
  - Fehlerhypothesen
  - 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

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) Uhrensynchronisation notwendig
  - Für die einzelnen Berechnungen ist jeweils ein Zeitslot reserviert) 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

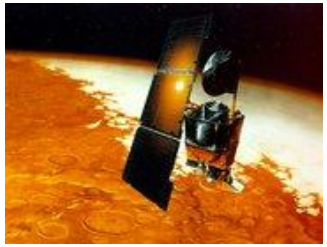
Echtzeitsysteme sind allgegenwärtig!







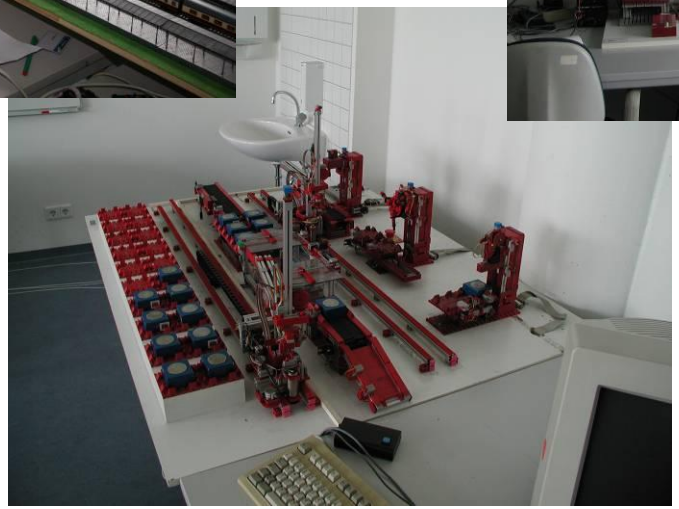




Einleitung Echtzeitsysteme

Anwendungen am Lehrstuhl / fortiss

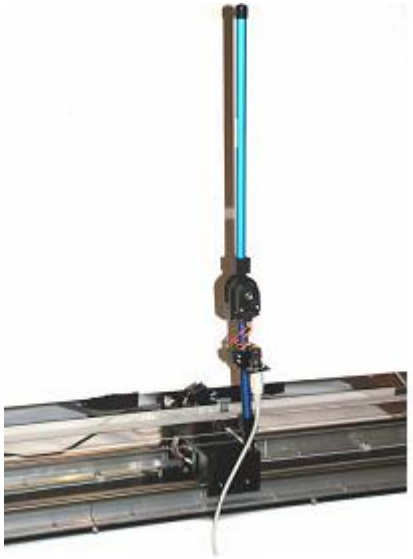
Steuerungsaufgaben (Praktika+Studienarbeiten)



Regelungsaufgaben (Praktika+Studienarbeiten)

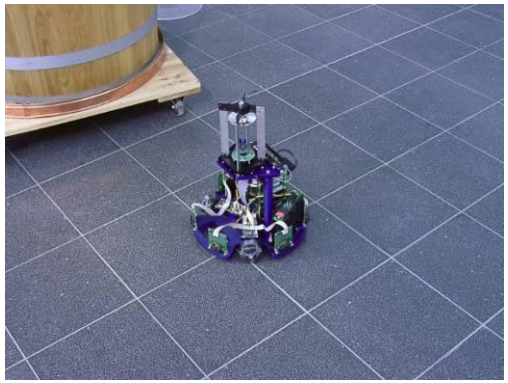
Schwebender Stab

Produktionstechnik

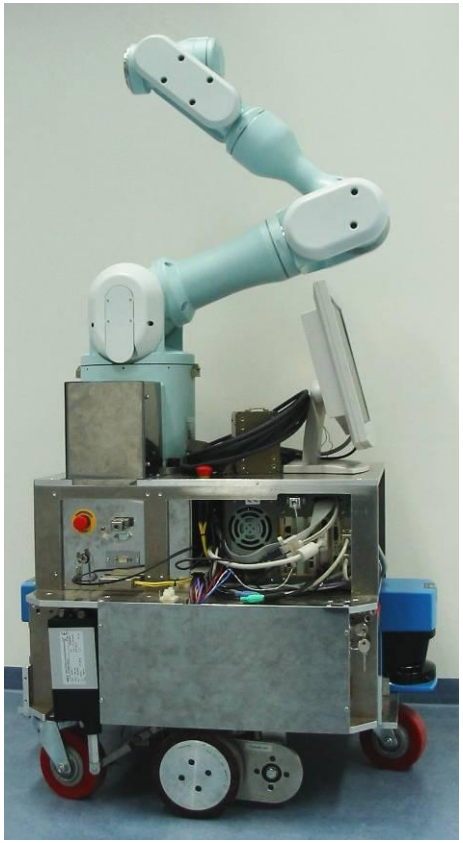


Invertiertes Pendel

Robotersteuerung

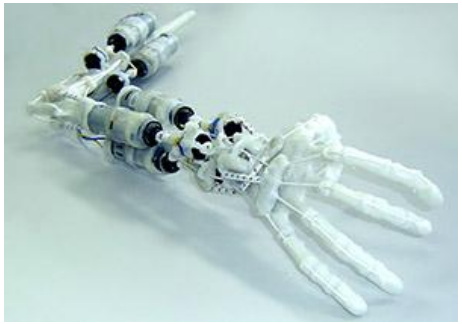


Robotino



Leonardo

Stäubli



Tumanoid

Aktuelle Entwicklung: Cyber-Physical Systems

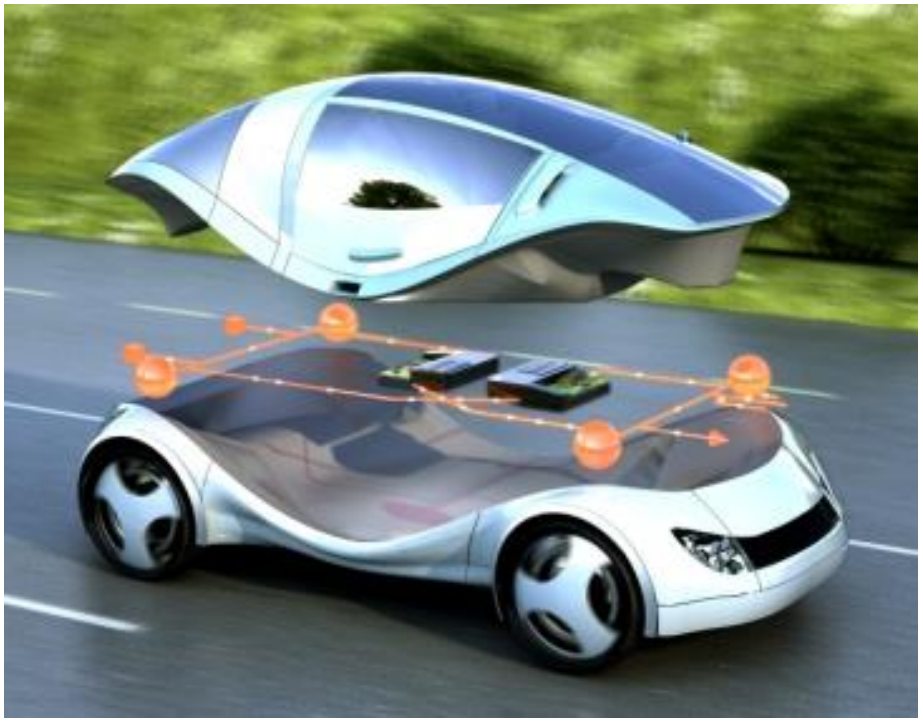
- Cyber-Physical System (CPS) bestehen aus mehreren Subsystemen von denen mindestens eines dieser Subsysteme mit der Umgebung interagiert (→ eingebettetes System mit Echtzeitanforderungen). Die Subsysteme sind typischerweise unabhängig voneinander entwickelt worden, sollen aber doch die Integration miteinander eine bessere Funktionalität liefern. Wandel ist ein ständiger Begleiter.
- Abgrenzung zu anderen Forschungsthemen:
  - Aktive Interaktion mit der Umgebung: Sensornetzwerke sind zunächst keine CPS
  - Unabhängigkeit der Subsysteme: Unterscheidung zu "klassischen" eingebetteten, vernetzten Systemen
  - Generell: die Integration muss sehr häufig autonom erfolgen, es gibt keine Firma, die die Integratorenrolle übernimmt → Integration über Standards anstelle durch Personen
- Mehr zu dem Thema unter: http://www.acatech.de/?id=1405 (Agenda CPS)

Leitprojekte von fortiss im Bereich Echtzeitsysteme / Cyber-Physical Systems

- Am fortiss wird eine große Zahl von Projekten im Bereich Echtzeitsysteme bzw. Cyber-Physical Systems für die Anwendungsgebiete Automotive, Industrieautomatisierung und Robotik durchgeführt
- Nahezu alle Projekte werden zusammen mit attraktiven Industriepartnern durchgeführt (u.a. Audi, BMW, EADS, Festo,Siemens)
- Für die Hörer dieser Vorlesung bieten sich zum Einstieg drei Projekte besonders an: die Leitprojekte "**RACE**", "**AutoPnP**" und "**SME Robotics**"
- Wir freuen uns über interessierte Studenten und bieten:
  - Spannende Studienarbeiten (Bachelor-, Master-, Guided Research)
  - Interessante Werkstudententätigkeiten
  - Möglichkeiten zur Promotion als wissenschaftlicher Mitarbeiter
- Mehr unter: http://www.fortiss.org/karriere/ueberblick/

Projektidee RACE

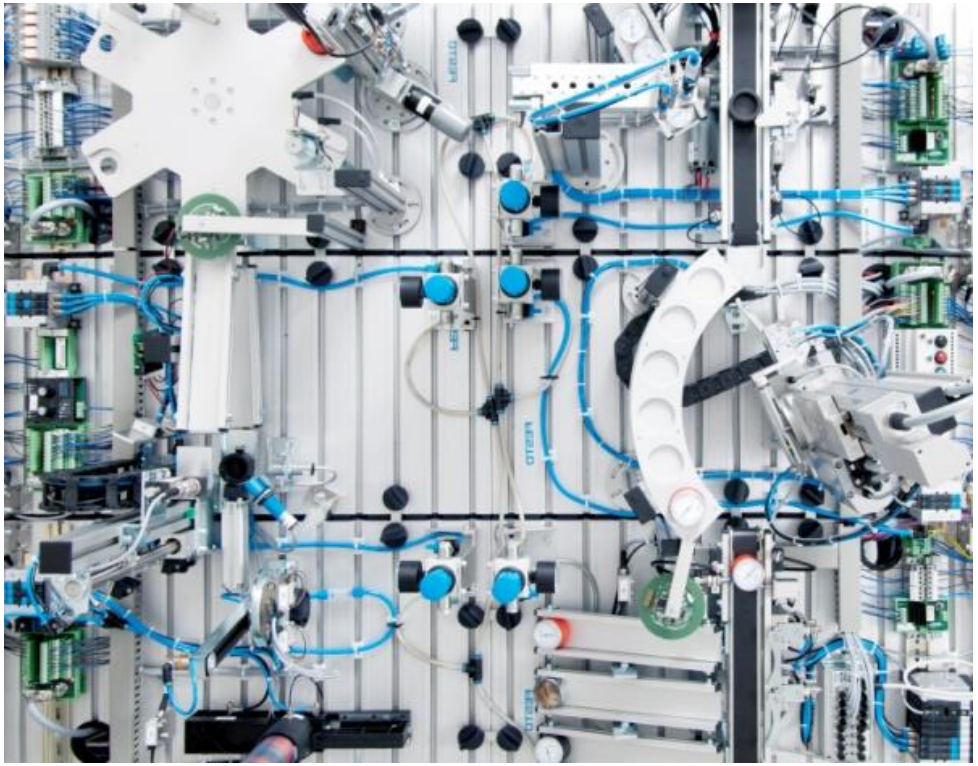
- Entwicklung eines Versuchsträgers für Plug&Play-fähige Fahrzeuge auf Basis eines Zentralrechnerkonzeptes
- Deutliche Vereinfachung der Entwicklung auf von komplexen Funktionen, wie dem autonomen Fahren
- Im Rahmen eines Studentenwettbewerbes können Studenten eigene Funktionen für das neue Elektrofahrzeug entwickeln
- Mehr Infos unter
  www.fortiss.org/ikt2030
  www.race-projekt.de



RACE Demonstrator – Autonomes Fahren

Projekt AutoPnP – Industrie 4.0

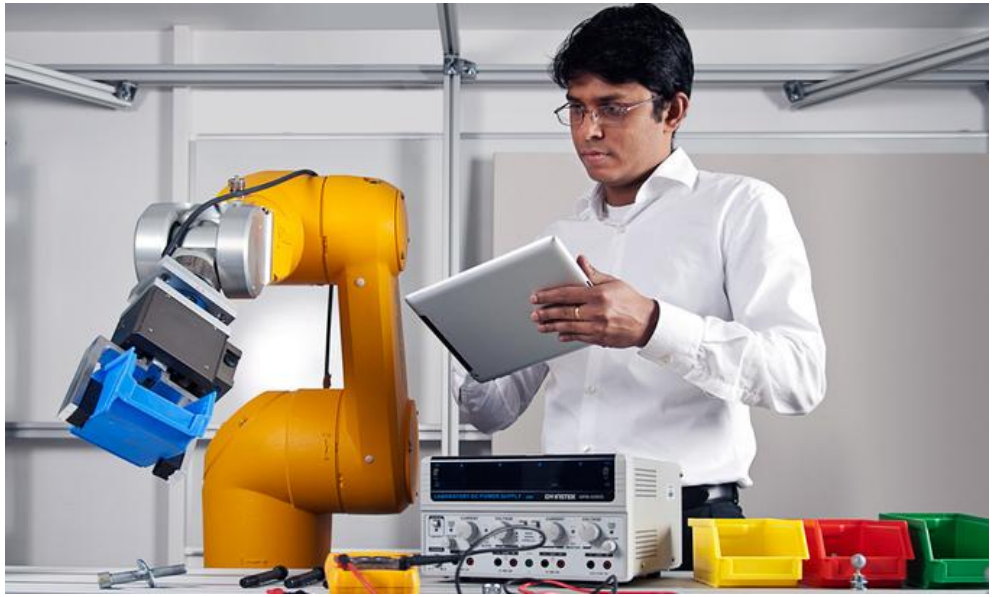
- Entwicklung einer wandelbaren Fabrik, die vollautomatisch ihre Topologie erkennt und das Produktionsprogramm synthetisiert und so sehr flexibel an neue Produkte angepasst werden kann.



Mehr Infos unter
http://www.fortiss.org/forschung/projekte/autopnp/

Projekt SME Robotics

- Entwicklung von intuitiv programmierbaren Robotern für den produzierenden Mittelstand



- Mehr Infos unter http://www.fortiss.org/forschung/projekte/smerobotics-1/

Erfolgskontrolle: Was sollten Sie aus dem Kapitel mitgenommen haben?

- Definition und Unterscheidung der Begriffe:
  - Echtzeitsystem und eingebettete Systeme
  - Harte / weiche Echtzeitsysteme
- Kenntnis von Beispielen für Echtzeitsysteme
- Wichtigste Aussage: Echtzeitsysteme heißt nicht schnell, sondern die Garantierung von maximalen Antwort- / Berechnungszeiten

  → Grundlage zur Beurteilung einer Technologie, ob diese echtzeitfähig ist

Klausurfragen

- Klausur WS 06/07
  - Was ist der Unterschied zwischen harten und weichen Echtzeitsystemen? (3 Punkte = 3 min)
    - **Antwort**: siehe Folien
  - Wieso sollte Virtual Memory nicht in Echtzeitsystemen verwendet werden? (3 Punkte = 3 min)
    - **Antwort**: Bei Verwendung von Virtual Memory hängt die Ausführungszeit davon ab, ob bei einem Speicherzugriff der Speicherplatz schon im Virtual Memory ist oder erst aus dem Hintergrundspeicher nachgeladen werden muss → die Zugriffszeiten unterscheiden sich für beide Situationen deutlich und erschweren die Zeitanalyse.
- Wiederholungsklausur WS 06/07 (5 Punkte = 5 min)
  - Ordnen Sie folgende Anwendungen in die Kategorien harte bzw. weiche Echtzeitsysteme ein und begründen Sie Ihre Antwort:
    - Ampelsteuerung
    - Flugzeugregelung
    - Internettelefonie
  - **Antwort**: Harte Echtzeitsysteme: Flugzeugregelung (Verzögerung kann zu Absturz führen), Ampelsteuerung (durch verzögerte Nachrichtenübermittlung können zwei Ampeln gleichzeitig auf grün schalten → Unfallgefahr), weiche Echtzeitsysteme: Internettelefonie (Verzögerungen führen zu Aussetzern im Gespräch → verminderte Qualität, aber keine großen Schäden); Zeitablauf einer Ampel: wenn die Ampel länger rot bleibt (aber das Gesamtsystem konsistent ist) führt die zu einer verminderten Qualität

Klausurfragen

- Klausur WS 10/11
- Gegeben sind folgende Aussagen über Echtzeitsysteme:
  - (i) Harte Echtzeitsysteme sind Systeme, die besonders schnell sind, also besonders kurze Berechnungszeiten haben.
  - (ii) Harte Echtzeitsysteme sind Systeme, bei denen ein bestimmtes Zeitverhalten garantiert werden kann, wobei das Zeitverhalten aber langsam sein kann.

- Welche dieser Aussagen ist richtig? Geben Sie für die richtige Aussage ein Beispiel an, das ein Echtzeitsystem ist. Geben Sie für die falsche Aussage ein Beispiel an, das zwar die Aussage erfüllt, aber kein Echtzeitsystem ist.
- **Antwort**:
  - Aussage (i): Falsch. Beispiel: eine Anfrage in einer Suchmaschine wird zwar sehr schnell beantwortet, allerdings ist es kein hartes Echtzeitsystem.
  - Aussage (ii): Richtig. Beispiel: Ampelsteuerung. Das Zeitverhalten ist im Sekunden- bzw. Minutenbereich, allerdings muss sichergestellt sein, dass alle Ampeln gleichzeitig umschalten.

Kapitel 2

Uhren & Synchronisation

Inhalt

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

Literatur

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

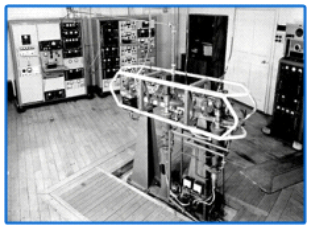
Relevanz für Echtzeitsysteme: Zeit und Ordnung

- Da viele Vorgänge in der Umwelt und Physik auf Zeit basieren, müssen Echtzeitsysteme häufig die Zeit wahrnehmen und darauf basierend interagieren können.
- Weitere Anforderung: Einordnen von Ereignissen im verteilten System (bevorzugt im Sinne einer zeitlichen Abfolge) ➔ globale Zeitbasis hilft bei der Ableitung einer zeitlichen Ordnung ➔ Zeitsynchronisation wird benötigt

Definition Zeit

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

Sonnenuhr
Deutsches Museum



Erste Cäsiumatomuhr

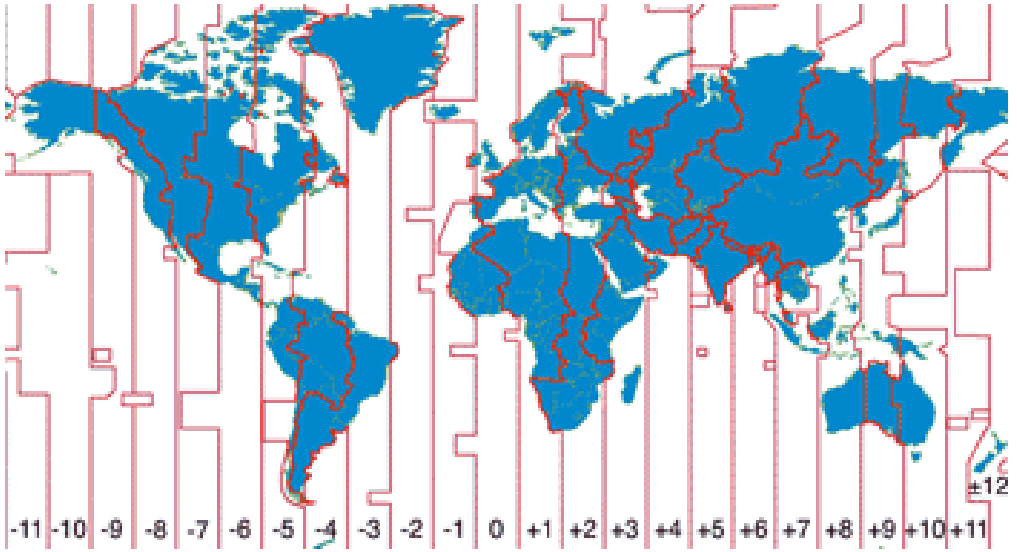
TAI (Temps Atomique International)

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

Atomuhr der Physikalisch-Technischen Bundesanstalt in Braunschweig

UTC (Coordinated Universal Time)

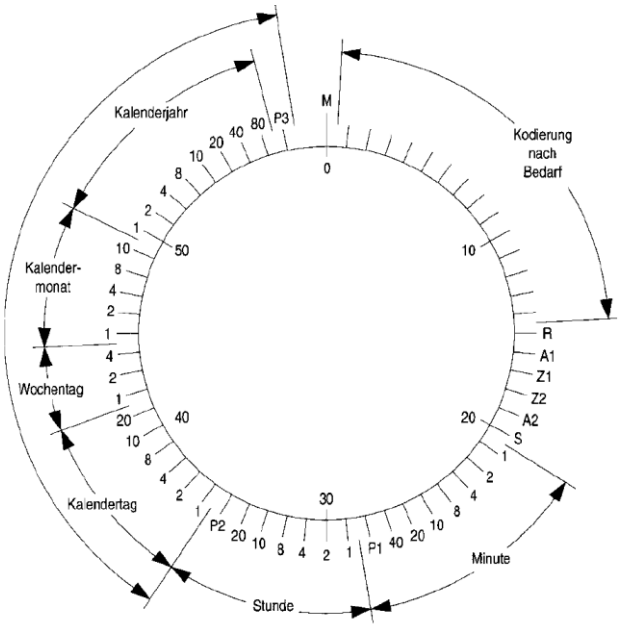
- eigentlicher Nachfolger der Greenwichzeit
- realisiert durch Atomuhren, die Zeiteinheit ist die SI-Sekunde
  → hochkonstante Zeiteinheit
- zusätzlich Übereinstimmung mit dem Sonnenlauf
  → einheitliche Grundlage zur Zeitbestimmung im täglichen Leben
- Durch Einfügen von Schaltsekunden wird die UTC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



Zeitzonen

DCF77

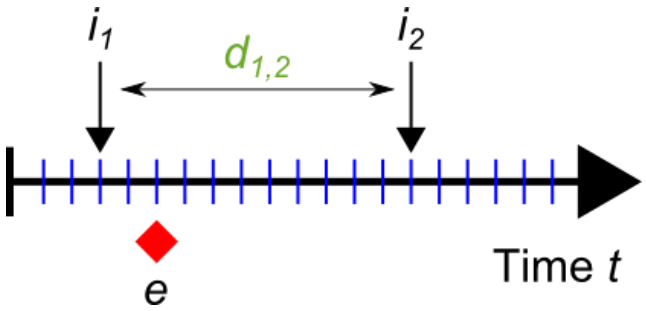
- Das PTB überträgt die aktuelle Uhrzeit über den Langwellen-sender DCF77
- Die Zeitinformationen werden als digitales Signal (negative Modulation → Absenkung der Trägeramplitude) im Sekunden-takt übertragen.



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

Zeit als Modell

- Zeit kann durch unendliche Menge T von Zeitpunkten („Instants“) modelliert werden
  - T ist dann eine geordnete Menge, d.h. falls p und q zwei Zeitpunkte sind so treten sie entweder:
    - Gleichzeitig oder
    - Eines nach dem anderen ein → p vor q oder q vor p (mutually exclusive)
  - Ordnung zwischen den Zeitpunkten heißt temporale Ordnung
  - T ist dicht
    - zwischen zwei Zeitpunkten p und r liegt zumindest ein weiterer Zeitpunkt q, dann und nur dann wenn p und r nicht gleichzeitig sind.
  - Intervall aus T heißt Zeitraum oder Dauer



Uhren und Synchronisation

Uhren

Aufgaben

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





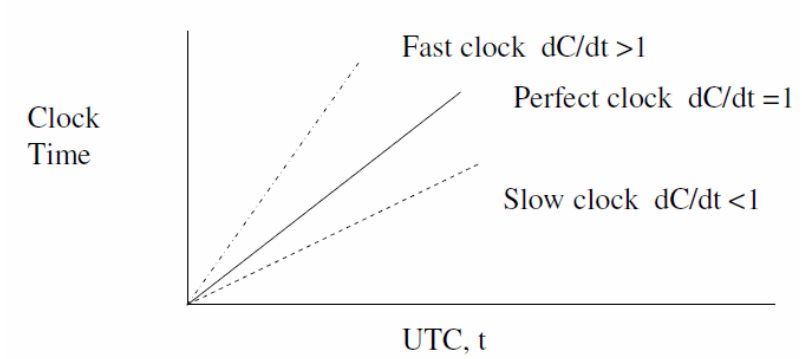
Genauigkeit von Uhren

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

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

Uhrenverhalten

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



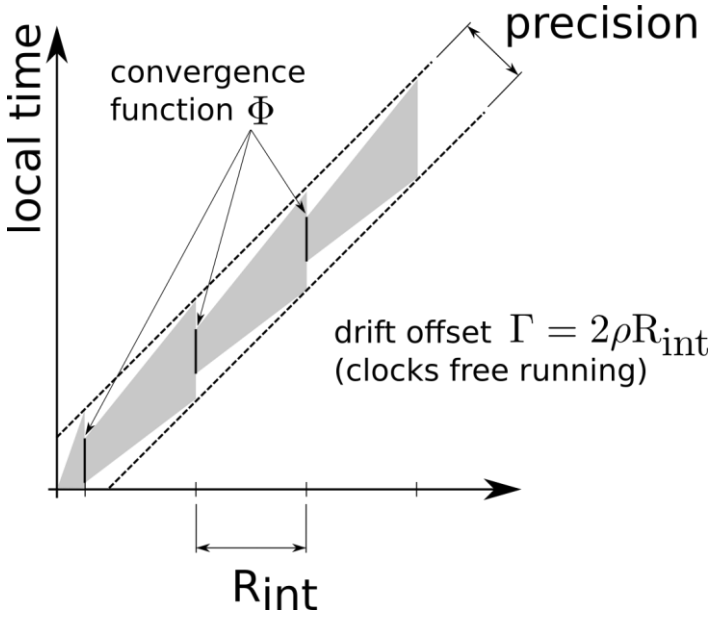
➔ Die Gangabweichung zweier korrekter Uhren kann beliebig groß werden, wenn die Uhren nicht synchronisiert sind.

Uhren und Synchronisation

Synchronisation

Ziele der Uhrensynchronisation

- Ziel ist es verschiedenen, unabhängigen Prozessen, die auf unterschiedlichen Rechnern laufen, zu ermöglichen konsistente Entscheidungen über die Ordnung verschiedener Ereignisse zu ermöglichen
- Unterscheidung zwischen:
  - Logischer Uhr: Zuordnung von Sequenznummern zu Ereignissen
  - Physikalische Uhren: Zuordnung eines Zeitstempels zu Ereignissen
- Im Rahmen dieses Abschnittes wird nur die Synchronisation von physikalischen Uhren betrachtet. Mehr zur Synchronisation von logischen Uhren findet man unter: http://www.cs.rutgers.edu/~pxk/417/notes/content/08-clocks.pdf



Grundlagen

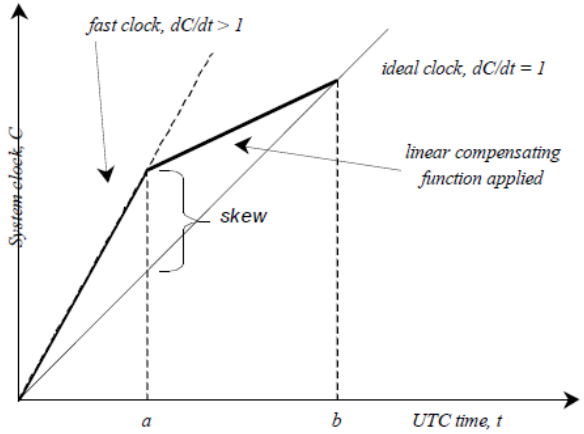
- Folgende Annahmen werden im Zusammenhang mit der Synchronisation gemacht:
  1. Alle Uhren besitzen zu Beginn in etwa die gleiche Zeit (in diesem Abschnitt wird vor allem die laufenden Synchronisation betrachtet).
  2. Die Uhren fehlerfreier Prozesse gehen annähernd richtig, d.h. sie besitzen eine ähnliche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

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

Arten der Synchronisation

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

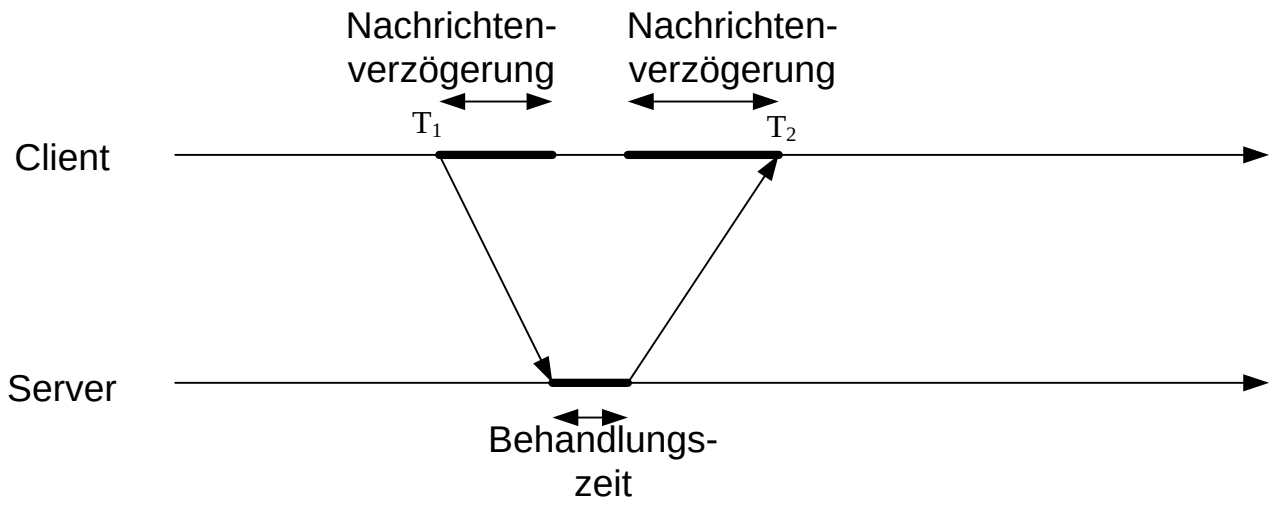
  Vorteil der externen Synchronisation: der maximal tolerierte Fehler kann halb so groß wie bei der internen Synchronisation gewählt werden.
- Ort der Synchronisation:
  - zentrale Synchronisation: Synchronisation wird von einer Einheit koordiniert → fehleranfällig
  - verteilte Synchronisation: alle Einheiten berechnen die globale Zeitbasis → hohes Datenaufkommen
- Umgang mit Synchronisationsunterschieden:
  - Anpassung der Uhrenfrequenz
  - Kleine Zeitsprünge: wichtig ist, dass hier Probleme mit der Programmlogik ausgeschlossen werden



© Krzyzanoski, Rutgers University, CS 417: Distributed Systems

Algorithmus von Cristian (1989)

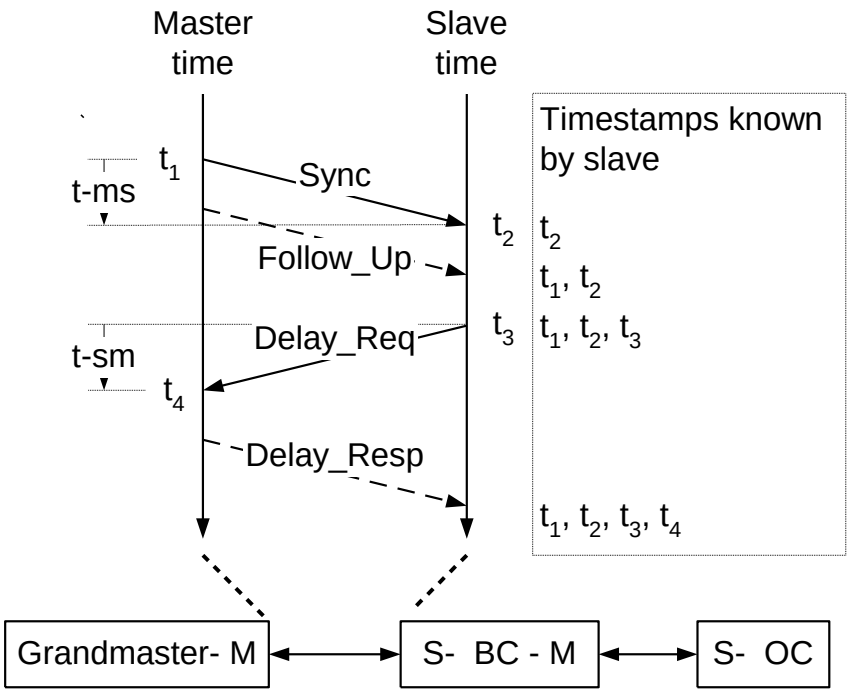
- Das Verfahren basiert auf verteilter, externer Synchronisation.
- Innerhalb des Systems existiert ein Time-Server, zumeist ein UTC-Empfänger.
- In regelmäßigen Abständen senden die anderen Einheiten einen Time-Request, der so schnell wie möglich vom Server beantwortet wird.



Algorithmus von Cristian

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

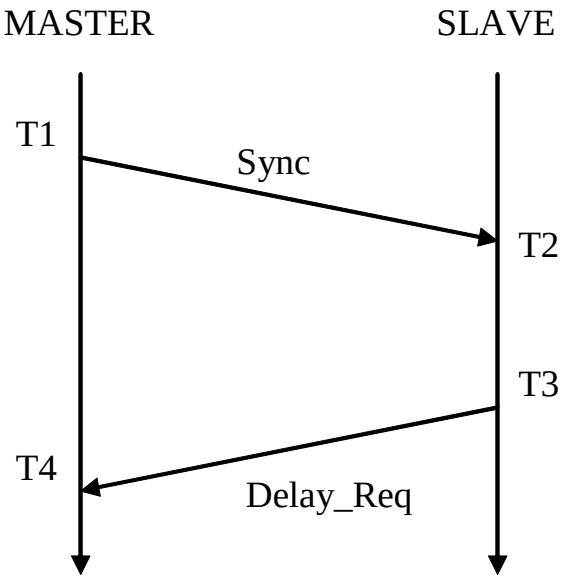
Erhöhung der Genauigkeit: Standard IEEE 1588

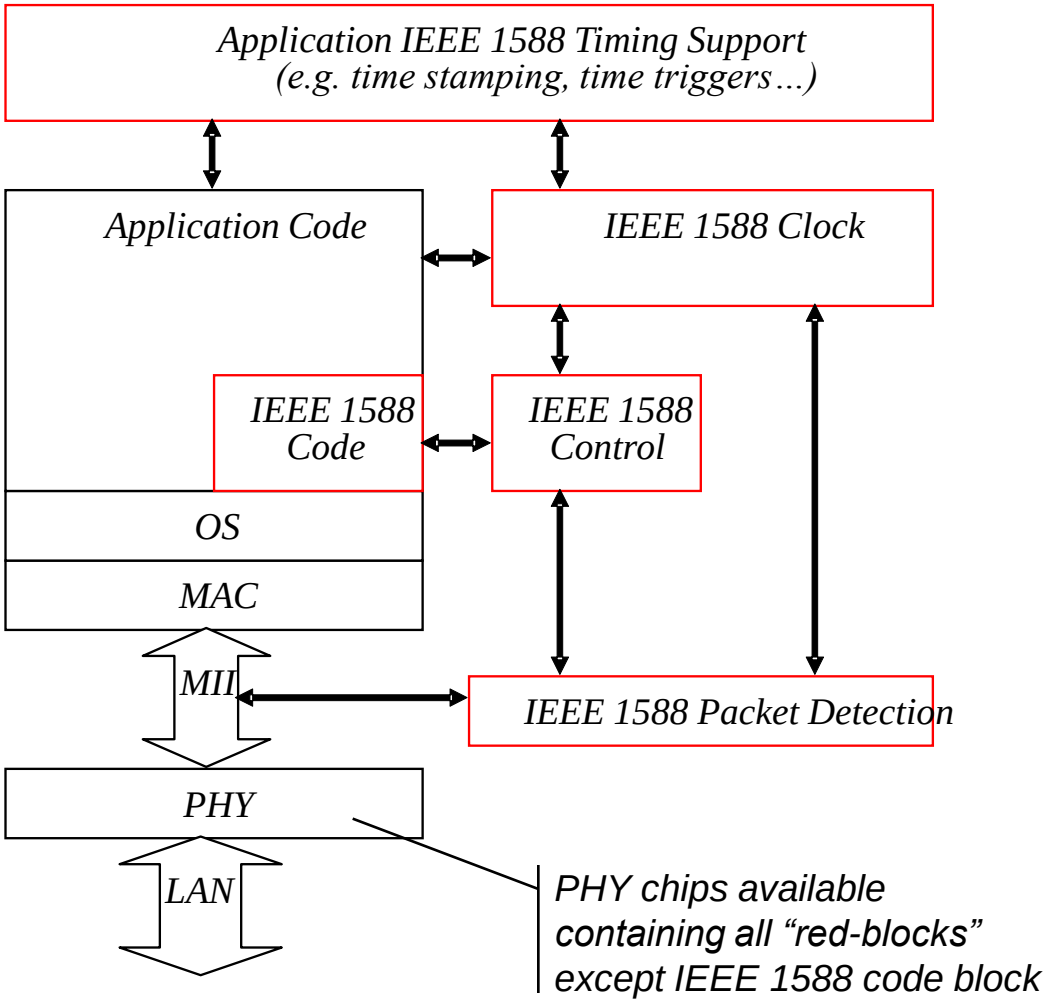


© John Eidson, Vortrag 09.10.2009, München

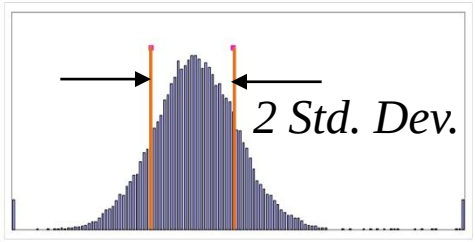
- Under the assumption that the link is symmetric
- Offset = (Slave time) – (Master time) = $[(t_2 – t_1) – (t_4 – t_3)]/2$ = [(t-ms) – (t-sm)]/2
- (propagation time) = $[(t_2 – t_1) + (t_4 – t_3)]/2$ = [(t-ms) + (t-sm)]/2
- Can rewrite the offset as
- Offset = $t_2 – t_1$ – (propagation time) = (t-ms) – (propagation time)
- If the link is not symmetric
  - The propagation time computed as above is the mean of the master-to-slave and slave-to- master propagation times
  - The offset is in error by the difference between the actual master-to-slave and mean propagation times
  - IF you know the asymmetry, the standard specifies how to correct for it.

Ordinary clock:

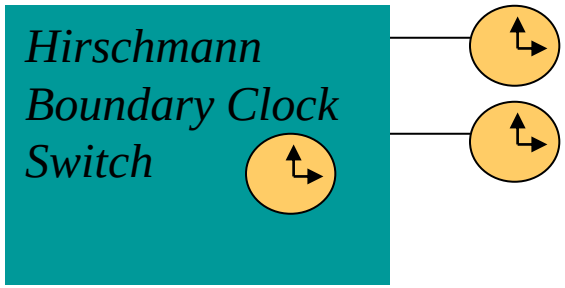


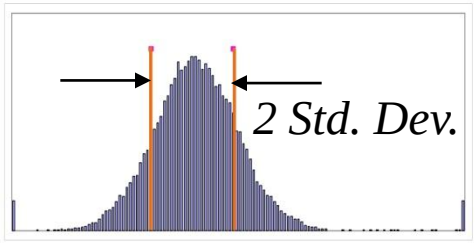


Synchronization accuracy results

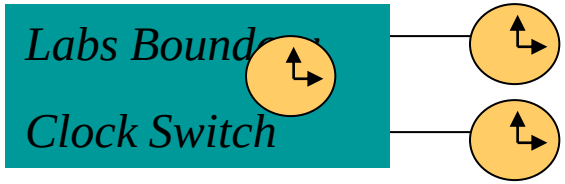


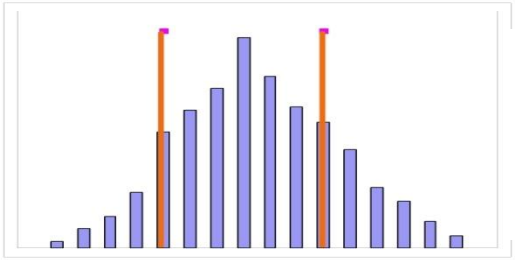
Offset: nanoseconds
Std. Dev: 121 ns
Max/min: -430,+720 ns
RCS switch, 8 hours





Offset: nanoseconds
Std. Dev: 24 ns
Max/min: -100,+100 ns
84 hours



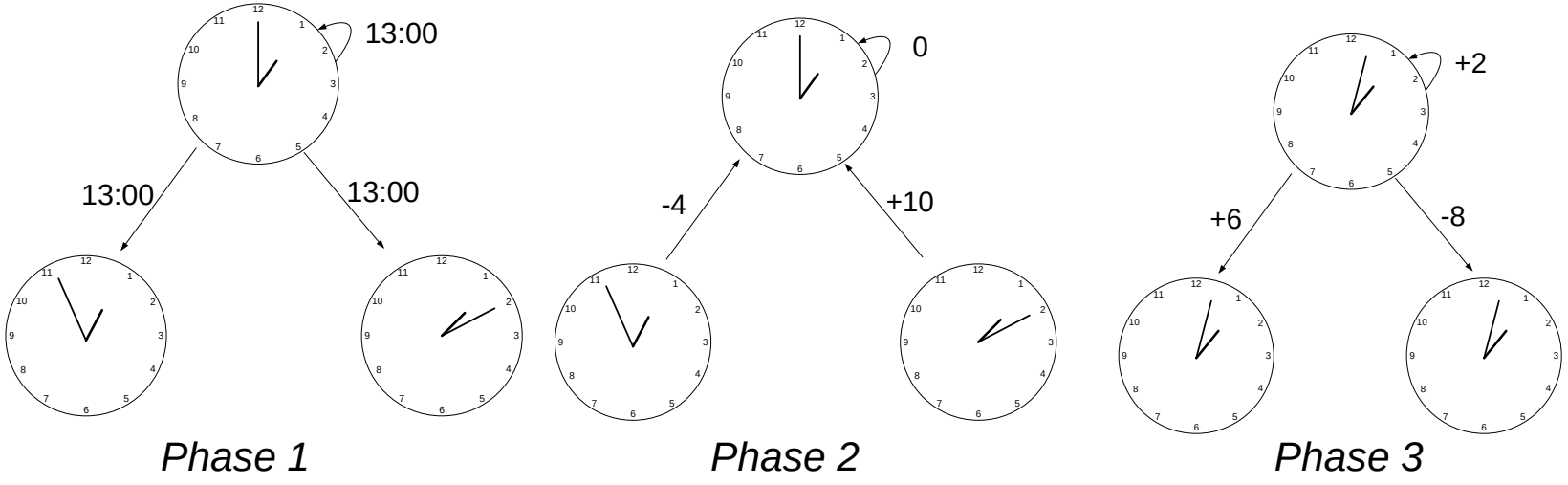


Offset: nanoseconds
Std. Dev: 15 ns
Max/min: -35,+45 ns
1 hour

For all of the above the clock resolution was 20-25 ns

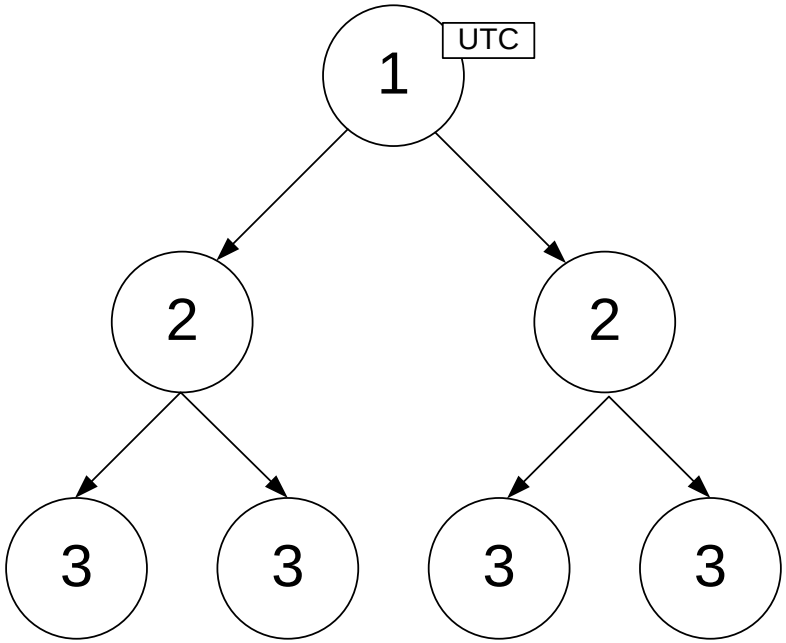
Algorithmus von Berkeley (1989)

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



NTP: Network Time Protocol (1982)

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



Uhren und Synchronisation

Synchronisation bei fehlerbehafteten Uhren

Problemstellung

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

Konvergenzalgorithmus (Leslie Lamport, 1985) [1]

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

[1] Synchronizing Clocks in the Presence of Faults, Leslie Lamport and P.M. Melliar-Smith, SRI International, Menlo Park, California

Konvergenzalgorithmus (Leslie Lamport, 1985)

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

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

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

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

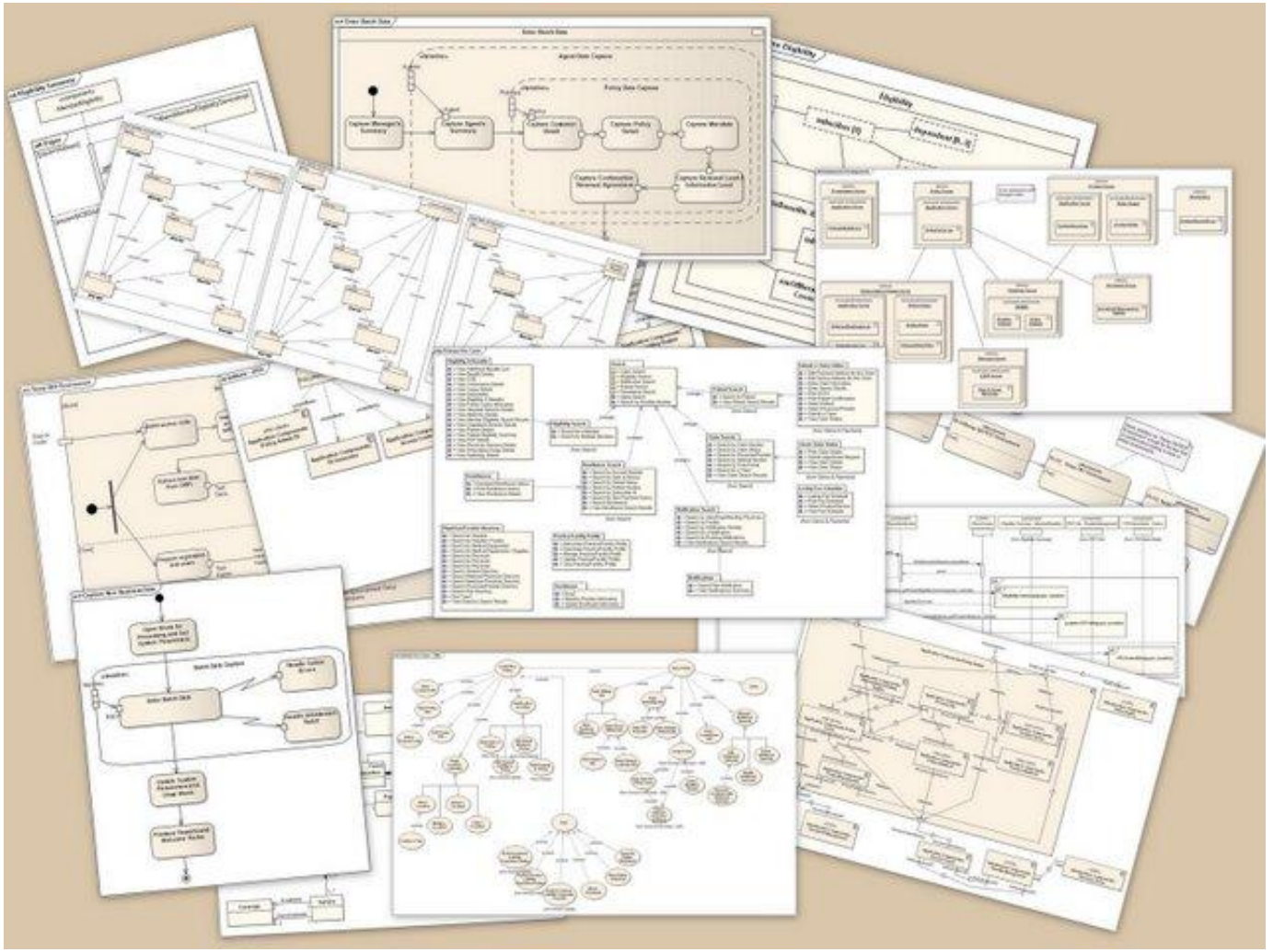
Erfolgskontrolle: Was sollten Sie aus dem Kapitel mitgenommen haben?

- Grundwissen über Zeit, Größenordnung der Uhrenabweichungen, typische Uhrenfehler und Notwendigkeit der Uhrensynchronisation
- Uhrensynchronisation:
  - Kenntnisse der verschiedenen Synchronisationstypen / -protokolle
  - Annahmen dieser Protokolle
  - Mechanismen zur Korrektur der eigenen Uhr
- Beispiel für eine Klausurfrage (Wiederholungsklausur 2006/2007):
  - Wieso kann bei der externen Synchronisation der maximal tolerierte Fehler als halb so groß, wie bei der internen Synchronisation gewählt werden?
  - **Antwort**: Bei der internen Synchronisation müssen gegensätzliche Fehler der involvierten Uhren angenommen werden (mindestens eine Uhr läuft zu schnell, mindestens eine Uhr zu langsam) – siehe auch Abbildung auf Folie 55.

Kapitel 3

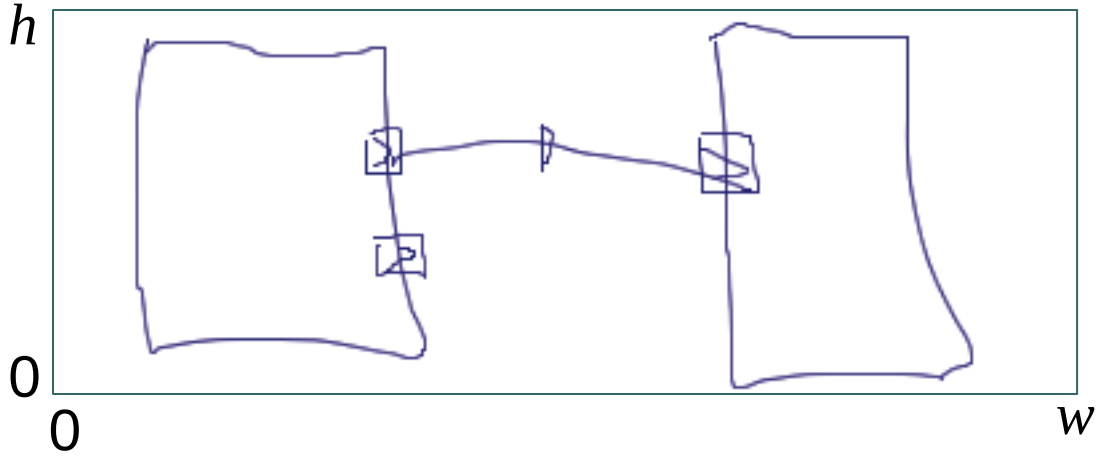
Modellgetriebene Entwicklung von Echtzeitsystemen (inkl. Werkzeuge)

One way to build heterogeneous models: UML : Unified?



[Image from Wikipedia Commons. Author: Kishorekumar 62]

The Truly Unified Modeling Language *TUML*



Achtung Ironie!!!

A *model* in TUML is a function of the form

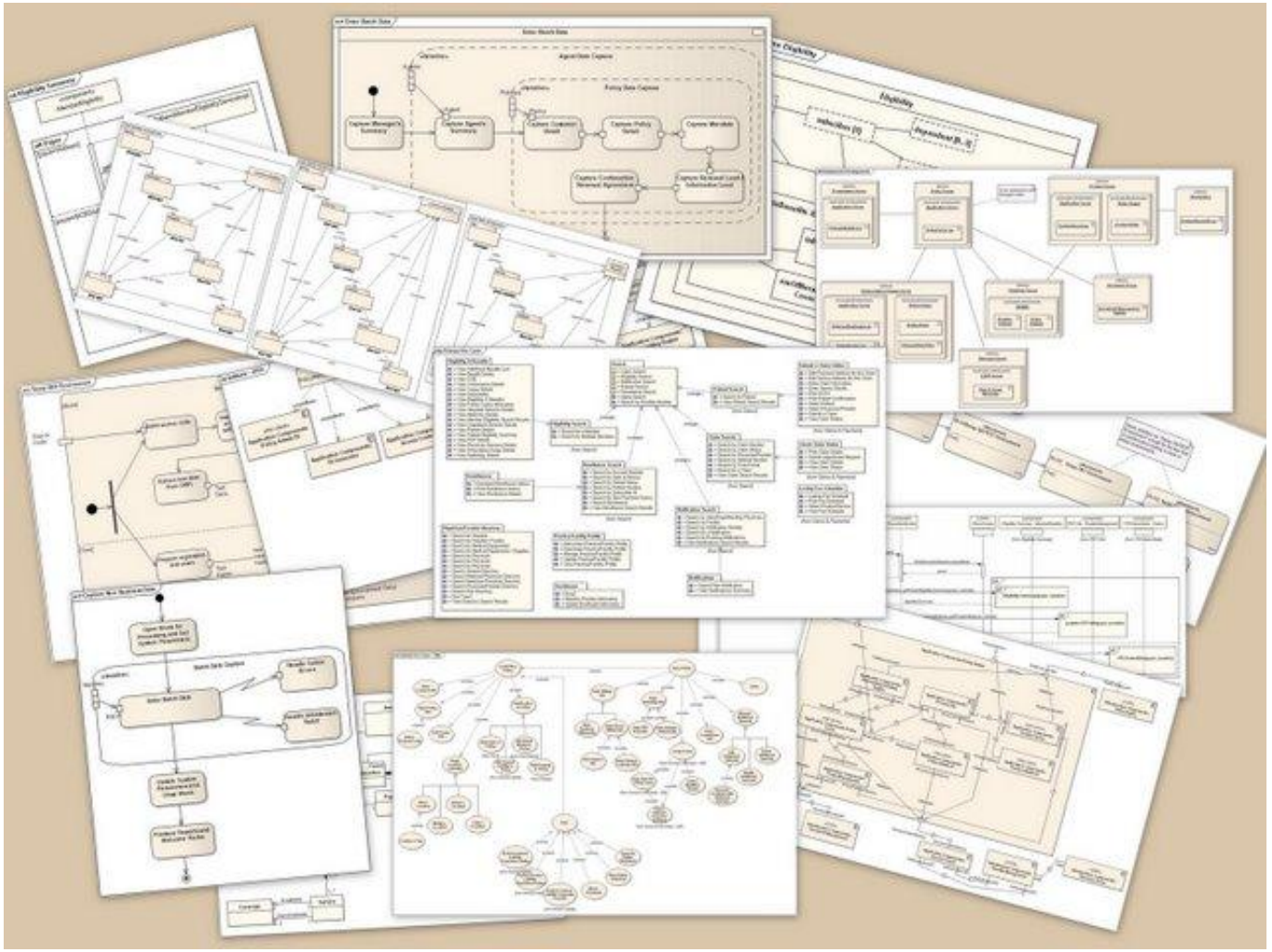
$$f : [0, w] \times [0, h] \rightarrow \{0, 1\}, \quad w, h \in \mathbb{N}$$

(notice how nicely formal the language is!)

Tools already exist.

With the mere addition of a *TUML profile*, every existing UML notation is a special case!

Examples of TUML Models



[Image from Wikipedia Commons. Author: Kishorekumar 62]

My Claim

Modeling languages that are not executable, or where the execution semantics is vague or undefined are not much better than TUML.

We can do better.

Useful Modeling Languages with Strong Semantics

Useful executable modeling languages impose *constraints* on the designer.

The constraints may come with benefits.

We have to stop thinking of constraints as a universal negative!!!

Freedom from choice!!!

Inhalt

- Fokus: Konzepte und Werkzeuge zur Modellierung **und** Generierung von Code für Echtzeit- und eingebettete Systeme

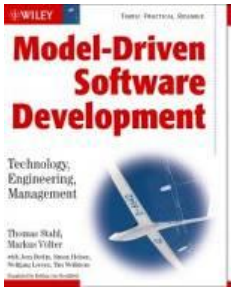
- Motivation
- Grundsätzlicher Aufbau, „Modelle“ und „Models of Computation“
  - Werkzeug Ptolemy
- Zeitgesteuerte Systeme
  - Werkzeug Giotto
- Synchrone Sprachen
  - Synchroner Datenfluss: EasyLab
  - Reaktive Systeme: Werkzeuge Esterel Studio

Fokus dieses Kapitels

- Voraussetzungen an Werkzeuge für ganzheitlichen Ansatz:
  - Explizite Modellierung des zeitlichen Verhaltens (z.B. Fristen)
  - Modellierung von parallelen Abläufen
  - 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 betrachtet, in Echtzeitsystemen ist Zeit jedoch als funktionale Eigenschaft anzusehen (siehe z.B. Edward Lee: Time is a Resource, and Other Stories, May 2008) http://chess.eecs.berkeley.edu/pubs/426/Lee_TimeIsNotAResource.pdf

Literatur



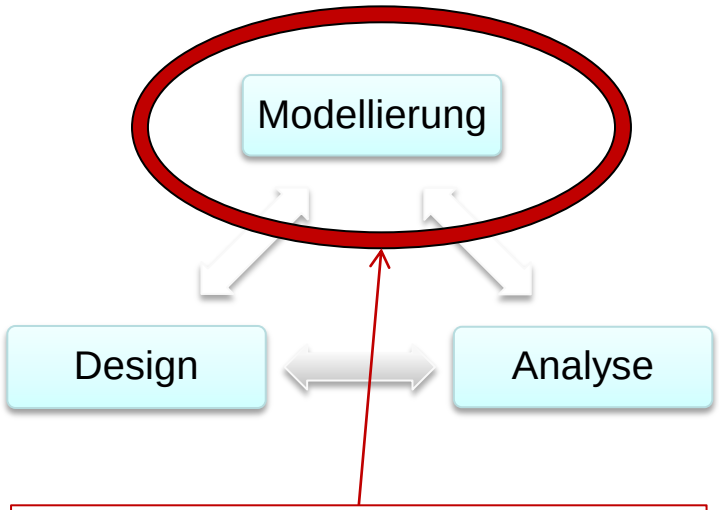
- Sastry et al: Scanning the issue – special issue on modeling and design of embedded software, Proceedings of the IEEE, vol.91, no.1, pp. 3-10, Jan 2003
- Thomas Stahl, Markus Völter: Model-Driven Software Development, Wiley, 2006
- Ptolemy: Software und Dokumentation http://ptolemy.eecs.berkeley.edu/ptolemyII/index.htm
- 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.
- Wikipedia:
  Von einem **Modell** spricht man oftmals als Gegenstand wissenschaftlicher Methodik und meint damit, dass eine zu untersuchende Realität durch bestimmte Erklärungsgrößen im Rahmen einer wissenschaftlich handhabbaren Theorie abgebildet wird.

Modellierung, Design und Analyse



Schwerpunkt dieses Kapitels:
-Wie sehen geeignete Modelle aus?
- Wie kann insbesondere die Implementierung abstrahiert werden?

- Modellierung
  - Modelle sind die Repräsentation des zu realisierenden Systems (evtl. Inklusive Umgebung) beschränkt auf die essentiellen Bestandteile
  - **Was** soll das System machen?
- Design:
  - Implementierung
  - Berücksichtigung der Hardware/Software-Aspekte
  - ***Wie*** *soll das System arbeiten?*
- Analyse:
  - Verstehen des Systems
  - ***Wieso*** *arbeitet das System so wie es arbeitet?*

Modellbasierte Entwicklung

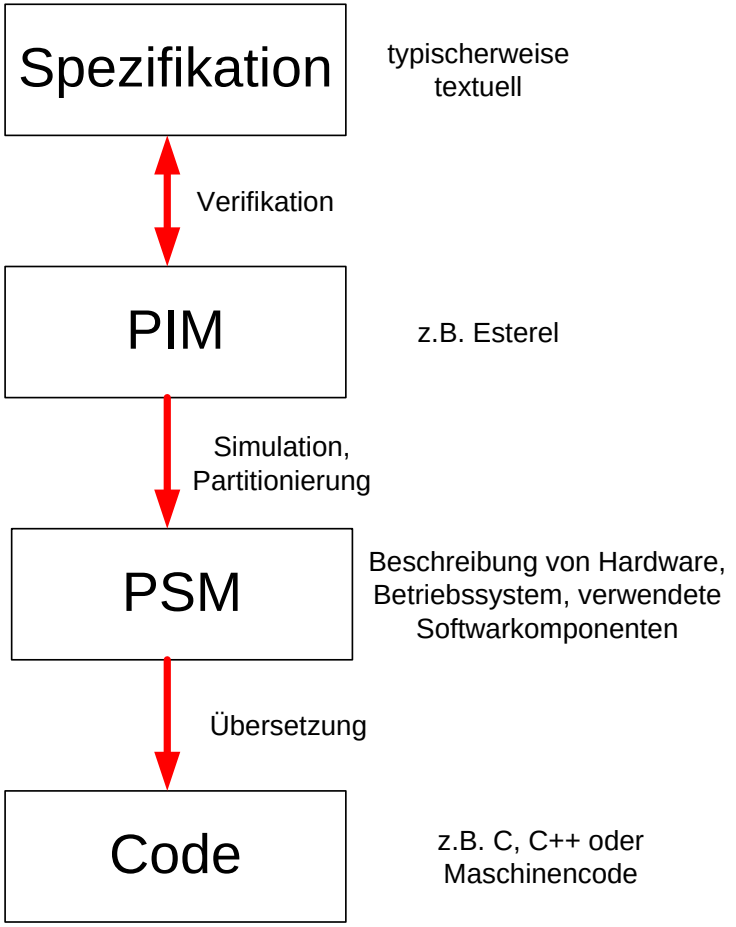
- Wir beobachten im Bereich eingebettete Systeme seit längerem einen Übergang von der klassischen Programmentwicklung zur einer Vorgehensweise, bei der Modelle (für physikalische Prozesse, für die Hardware eines Systems, für Verhalten eines Kommunikationsnetzes, usw.) eine zentrale Rolle spielen
- Modelle werden typischerweise durch verknüpfte grafische Notationselemente dargestellt (bzw. definiert)
- **Funktions-Modelle** sind dabei (meist Rechnerausführbare) Beschreibungen der zu realisierenden Algorithmen
- In unterschiedlichen Phasen der Systementwicklung kann ein Modell unterschiedliche Rollen annehmen – und etwa zum Funktionsdesign, zur Simulation, zur Codegenerierung verwendet werden
- Im günstigsten Fall kann ein System grafisch notiert („zusammengeklickt“) werden und unmittelbar Code erzeugt werden, der dann auf einem Zielsystem zur Ausführung gebracht wird
- Diese Entwicklung steht im Einklang mit der generellen Geschichte der Informatik, die immer mächtigere (Programmier-) Werkzeuge auf immer abstrakterem Niveau geschaffen hat

Vorteile Modellbasierter 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 Modelle in Code getestet werden
    - das Modell auf gewisse Eigenschaften hin überprüft werden

Beispiel 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)
- http://www.omg.org/mda



MDA im Kontext von Echtzeitsystemen

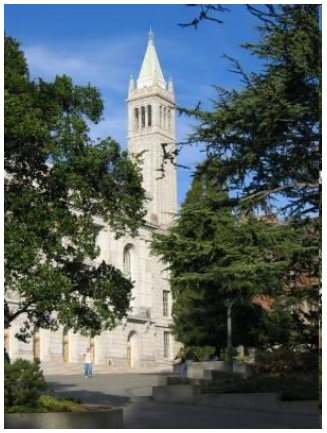
- In Echtzeitsystemen / eingebetteten Systemen ist bei einem umfassenden Ansatz ein Hardwaremodell (z.B. Rechner im verteilten System, Topologie) schon in frühen Phasen (PIM) notwendig
- Das plattformspezifische Modell (PSM) erweitert das Hardware- & Softwaremodell um Implementierungskonzepte, z.B.
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

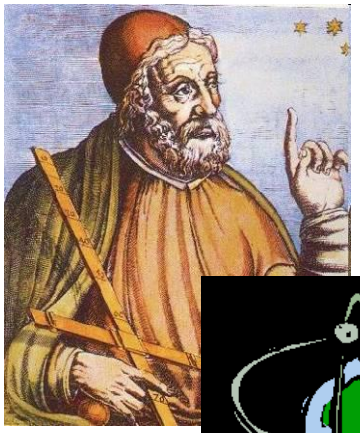
Modellierung von Echtzeitsystemen

Aktoren, Ausführungsmodelle

Werkzeuge: Ptolemy

Ptolemy

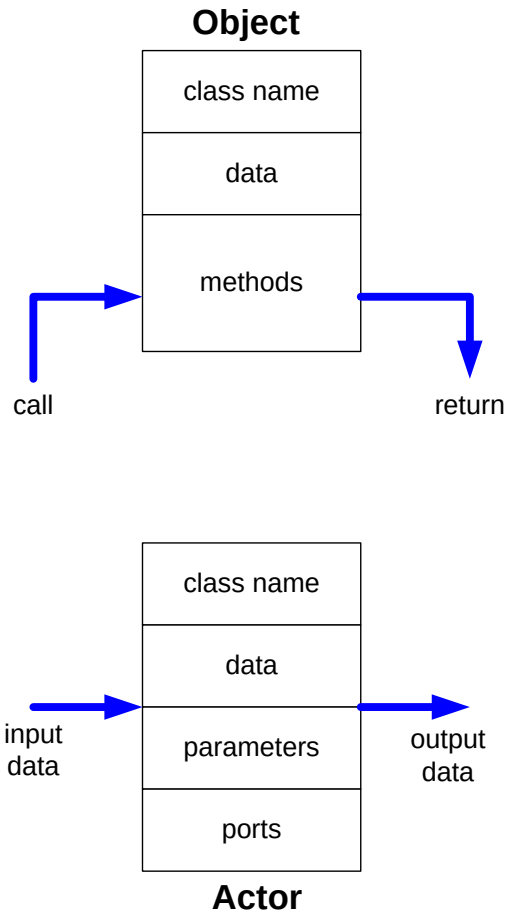




- Das Ptolemy*-Projekt an der UC Berkeley untersucht verschiedene Modellierungs-methodiken für eingebettete Systeme mit einem Fokus auf verschiedene Ausführungs-modelle (Models of Computation)
- Ptolemy unterstützt
  - Modellierung
  - Simulation
  - Codegenerierung
  - Formale Verifikation (teilweise)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

***Claudius Ptolemaeus**, (* um 100, vermutlich in Ptolemais Hermii, Ägypten; † um 175, vermutlich in Alexandria), war ein griechischer Mathematiker, Geograph, Astronom, Astrologe, Musiktheoretiker und Philosoph. Ptolemäus schrieb die *Mathematike Syntaxis* („mathematische Zusammen-stellung“), später *Megiste Syntaxis* („größte Zusammenstellung“), heute *Almagest* (abgeleitet vom Arabischen *al-Majisṭī*) genannte Abhandlung zur Mathematik und Astronomie in 13 Büchern. Sie war bis zum Ende des Mittelalters ein Standardwerk der Astronomie und enthielt neben einem ausführlichen Sternenkatalog eine Verfeinerung des von Hipparchos von Nicäa vorgeschlagenen geozentrischen Weltbildes, das später nach ihm *Ptolemäisches Weltbild* genannt wurde. (Wikipedia)

Ptolemy: Aktororientiertes Design



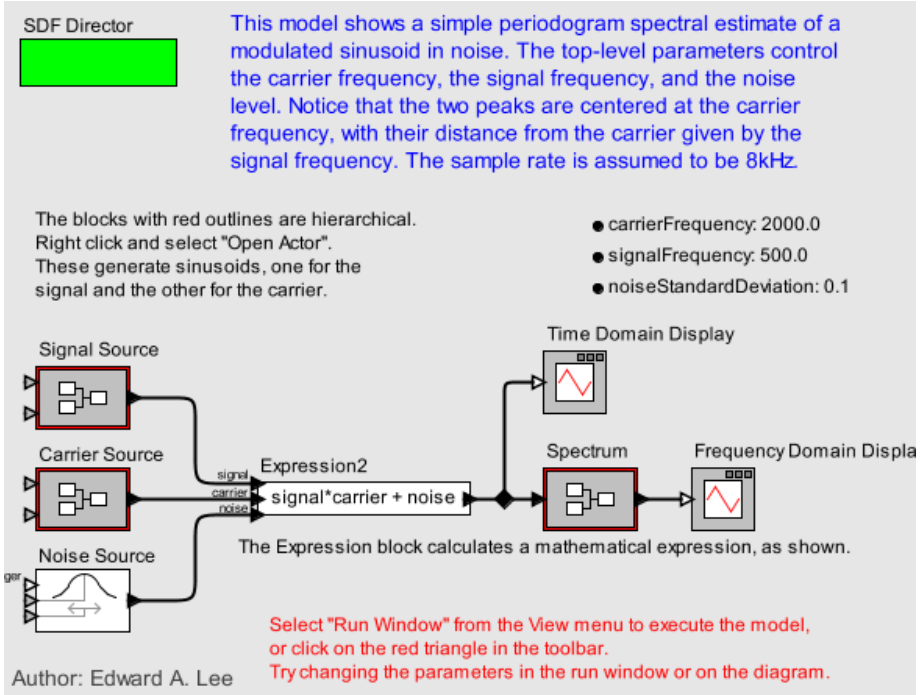
- Ptolemy-Modelle basieren auf Aktoren anstelle von Objekten
- Objekte:
  - Fokus liegt auf Kontrollfluss
  - Objekte werden manipuliert
- Aktoren
  - Fokus liegt auf Datenfluss
  - Aktoren manipulieren das System
- Vorteil beider Ansätze: erhöhte Wiederverwendbarkeit
- Vorteil von Aktoren: leichtere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

Ptolemy: Ausführungsmodelle

- Ausführungsmodelle (models of computation) bestimmen die Interaktion von Komponenten/Aktoren
- Die Eignung eines Ausführungsmodells hängt von der Anwendungs-domäne, aber auch der verwendeten Hardware, ab
- In Ptolemy wird durch die Einführung von „Dirigenten“ (director) die funktionale Ausführung (Verschaltung der Aktoren) von der zeitlichen Ausführung (Abbildung im Direktor) getrennt.
- Aktoren können unter verschiedenen Ausführungsmodellen verwendet werden (z.B. synchron, asynchron)
- Verschiedene Ausführungsmodelle können hierarchisch geschachtelt werden (modal models).
  - Typisches Beispiel: Synchroner Datenfluss und Zustandsautomaten

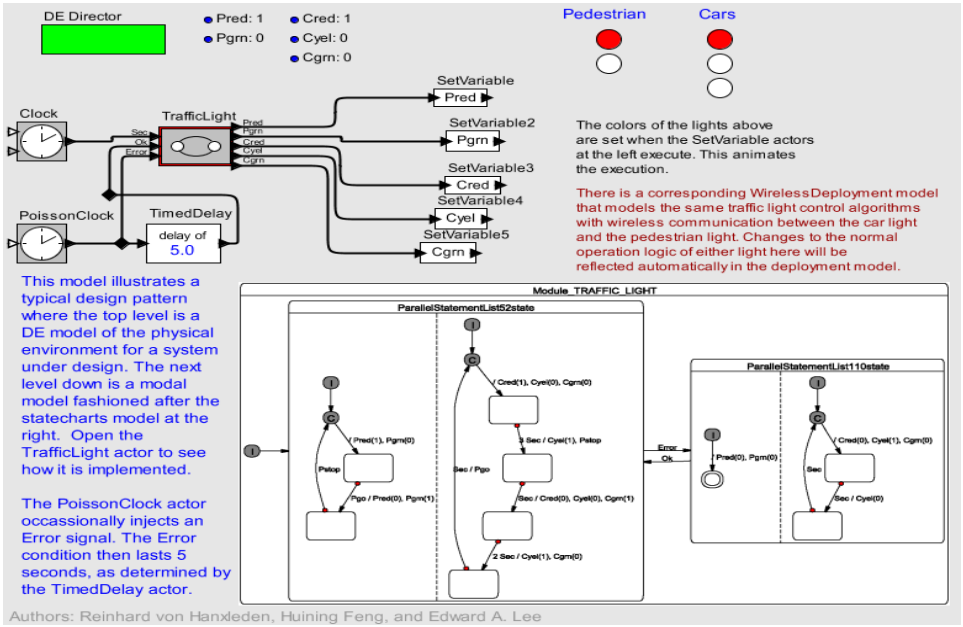
Example Ptolemy Model of Computation: Synchronuous Dataflow

- Prinzip:
  - Annahme: unendlich schnelle Maschine
  - Daten werden zyklisch verarbeitet (zeitgesteuert oder best effort)
  - Pro Runde wird genau einmal der Datenfluss ausgeführt
- Vorteile:
  - Statische Speicherallokation
  - Statischer Schedule berechenbar
  - Verklemmungen detektierbar
  - Laufzeit kann einfach bestimmt werden
- Werkzeuge:
  - Matlab/Simulink
  - Labview
  - EasyLab
- Anwendungsdomänen, u.a.:
  - Industrieautomatisierung (IEC 61131)
  - Regelungssysteme



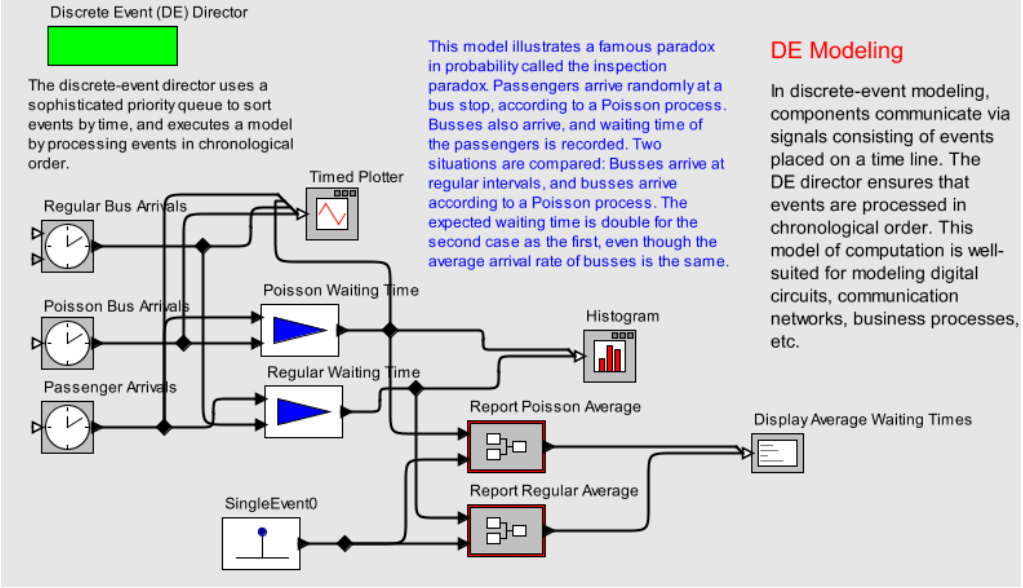
Example Ptolemy Model of Computation: Synchronous Reactive

- Prinzip:
  - Annahme: unendlich schnelle Maschine
  - Diskrete Ereignisse (DE) 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
- Anwendungsgebiete:
  - Ereignisbasierte Systeme, z.B. in der Avionik



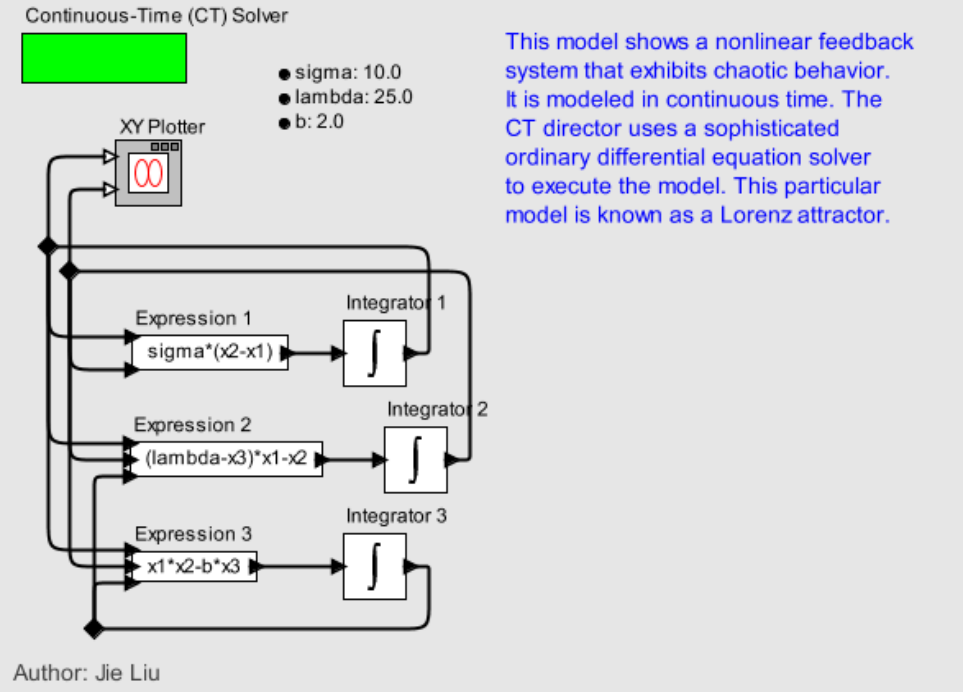
Example Ptolemy Model of Computation: Discrete Event

- Prinzip:
  - Kommunikation über Ereignisse
  - Jedes Ereignis trägt einen Wert und einen Zeitstempel
  - An jedem Aktor werden die Ereignisse in der Reihenfolge ihres Zeitstempels verarbeitet
- Variante:
  - Distributed Discrete Events
- Anwendungsgebiet:
  - Digitale Hardware
  - Telekommunikation
  - Verteilte, synchronisierte Systeme
- Werkzeuge:
  - VHDL
  - Verilog



Example Ptolemy Model of Computation: Continuous Time

- Prinzip:
  - Verwendung kontinuierlicher Signale (bestimmt gemäß Differential-gleichungen)
  - Ableitung des Codes durch Diskretisierung
- Anwendungsgebiet:
  - Simulation und Reglerauslegung
- Werkzeuge:
  - Simulink
  - Labview



Weitere Models of Computation

- 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
- Process Networks
  - Prozess senden zur Kommunikation Nachrichten über Kanäle
  - Kanäle können Nachrichten speichern: 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

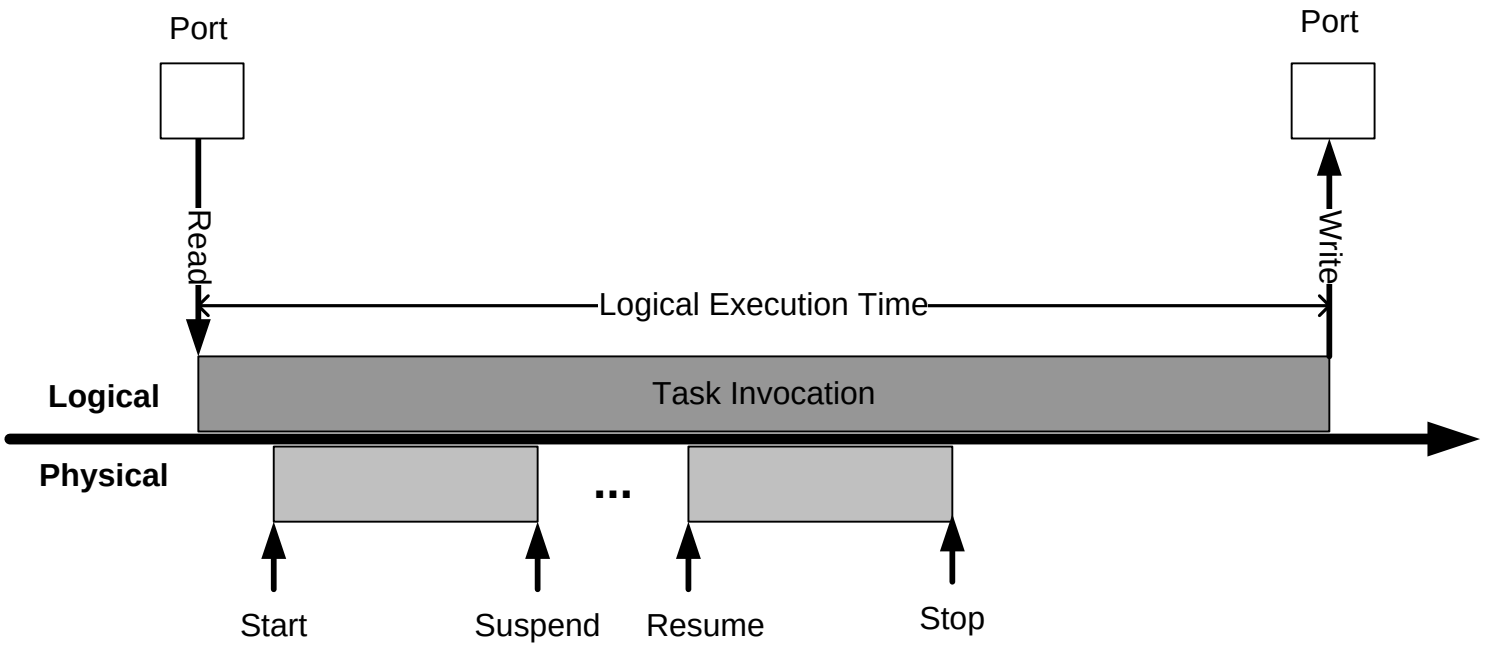
Logische Ausführungszeiten

Werkzeug: Giotto

Giotto: Hintergrund

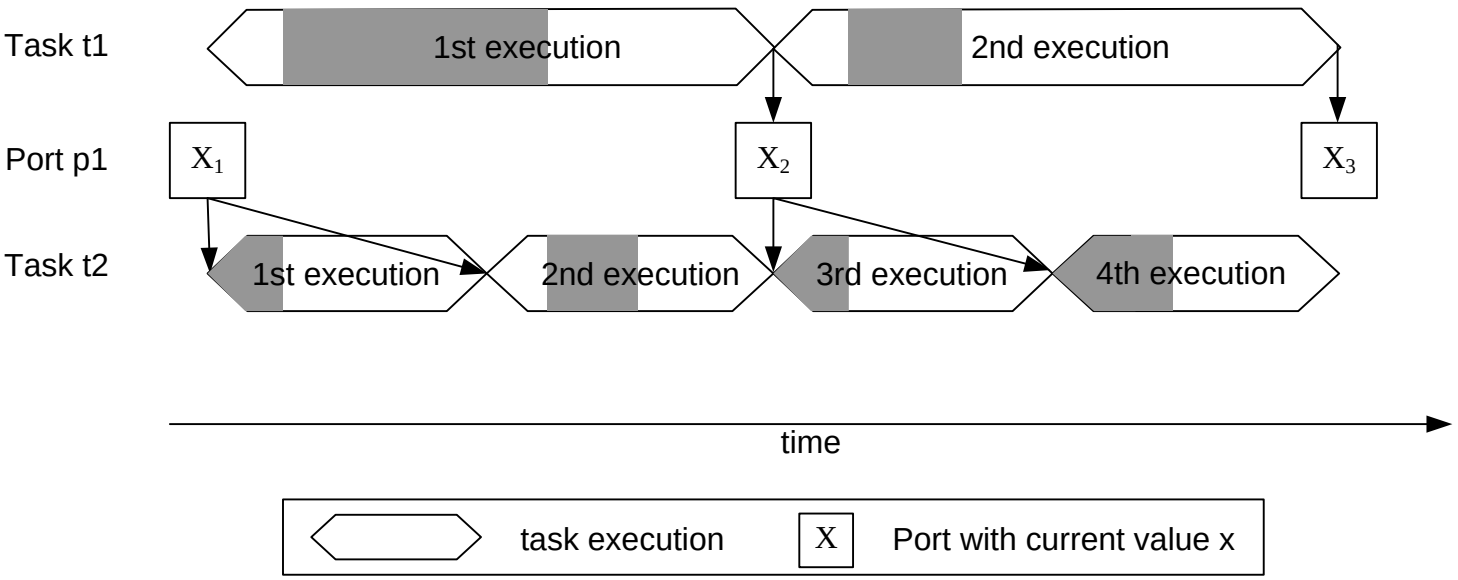
- Programmierumgebung für eingebettete Systeme (evtl. ausgeführt im verteilten System)
- Ziel:
  - strikte Trennung von plattformunabhängiger Funktionalität und plattformabhängigen Scheduling und Kommunikation
  - temporaler Determinismus ohne den Entwickler zu zwingen Implementierungsdetails zu definieren
- Hauptkonzept: Logische Ausführungszeiten
- Aktoren:
  - Tasks
    - Programmblock aus sequentiellen Code
    - keine Synchronisationspunkte, blockende Operationen erlaubt
    - Schnittstellen: Ports
- http://embedded.eecs.berkeley.edu/giotto/

Logische Ausführungszeit



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

Kommunikation zwischen Tasks



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)
- 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

Reaktive Systeme

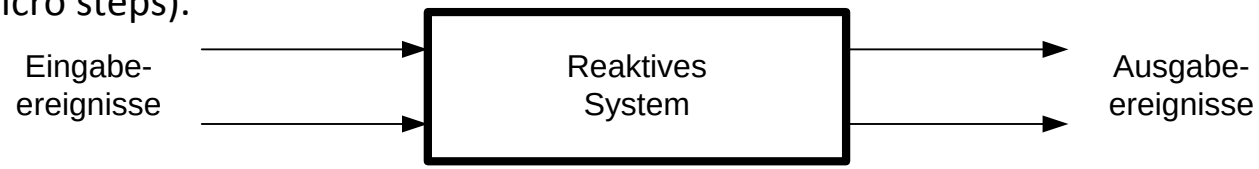
Werkzeuge: SCADE, Esterel Studio

Esterel

- Esterel ist im klassischen Sinne eher eine Programmiersprache, als eine Modellierungs-sprache
- 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:
  - In Esterel werden (parallele) Programme in **einen** endlichen Automaten umgewandelt
  - Aus dem endlichen Automaten wird ein Programm mit **einem** Berechnungsstrang erzeugt → deterministische Ausführung trotz paralleler Modellierung.
- SCADE (ein kommerzielles Tool, dass u.a. die Esterel-Sprache verwendet) wurde bei der Entwicklung von Komponenten für den Airbus A380 eingesetzt.
- Ein frei verfügbarer Esterel-Compiler kann unter http://www-sop.inria.fr/esterel.org/files/ bezogen werden (siehe Links auf der Vorlesungs-Homepage).

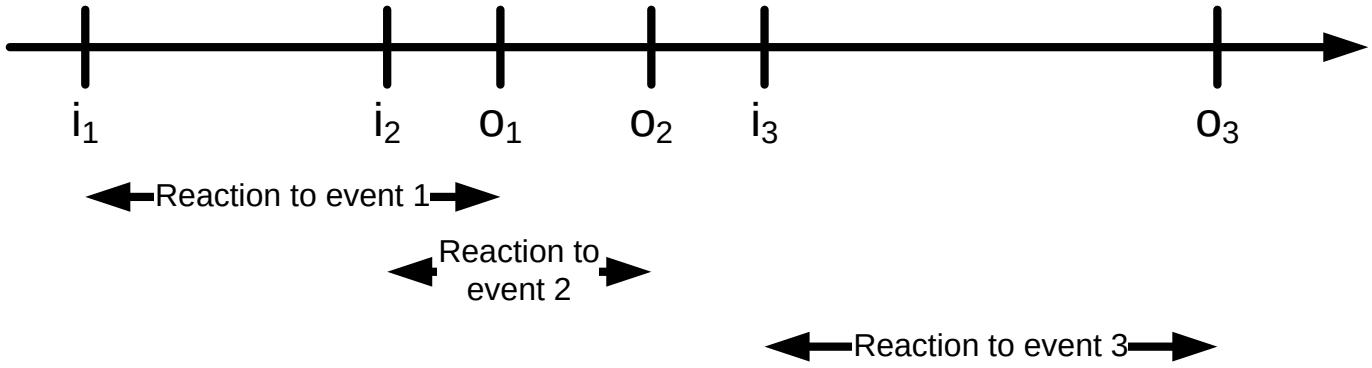
Einführung in Esterel

- Esterel gehört zu der Familie der **synchronen** Sprachen. Dies sind Programmiersprachen, die optimiert sind, um **reaktive Systeme** zu programmieren. Weitere Vertreter: Lustre, Signal, Statecharts
- Bei **reaktiven Systemen** erfolgen Reaktionen direkt auf **Eingabeereignisse**
- 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 die **logische Zeit** (die Anzahl der Interaktionen) verwendet wird.
  - Interaktionen, oft auch **macro steps** genannt, bestehen aus einzelnen Teilschritten (micro steps).



Reaktive Systeme - Allgemein

- In reaktiven Systemen (reactive / reflex systems) werden für Eingabeereignisse Ausgaben unter Einhaltung zeitlicher Rahmenbedingungen erzeugt.
- Reaktive Systeme finden u.a. Anwendung in der Industrie zur Prozesssteuerung und zur Steuerung / Regelung in Automobilen und Flugzeugen.
- Schwerpunkte bei der Umsetzung von reaktiven Systemen sind Sicherheit und Determinismus.
- Bearbeitung der Ereignisse kann sich überlappen (i input, o output)



Einschränkung in Esterel / synchronen Sprachen: Synchronitätshypothese

- 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 (ohne erkennbare Zeitverzögerung). Reaktionsintervalle reduzieren sich zu Reaktionsmomenten (reaction instants).

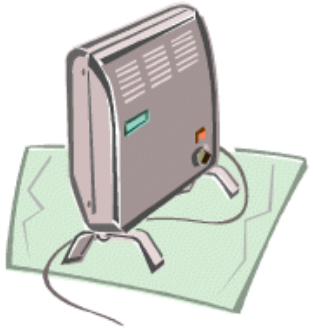
- **Rechtfertigung**: Diese Annahme ist korrekt, wenn die Wahrscheinlichkeit des Eintreffens eines zweiten Ereignisses, während der initialen Reaktion auf das vorangegangene Ereignis, sehr klein ist.

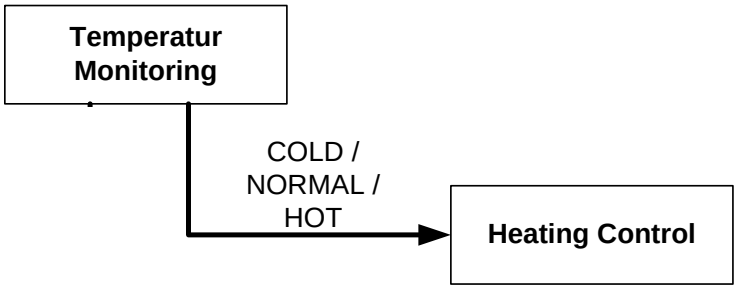
- Esterel erlaubt das gleichzeitige Auftreten von mehreren Eingabeereignissen. Die Reaktion ist in Esterel dann vollständig, wenn das System auf alle Ereignisse reagiert hat.

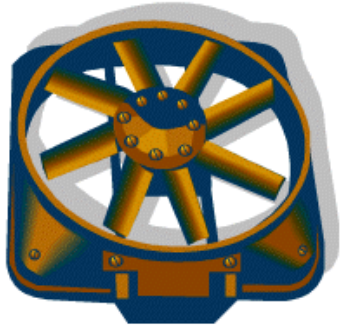
Determinismus

- Esterel ist deterministisch: auf eine Sequenz von Ereignissen (auch gleichzeitigen) muss immer dieselbe Sequenz von Ausgabe Ereignissen 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 er auch die Gefahr, dass Ereignisse „vergessen“ werden, falls sie exakt zeitgleich mit höher priorisierten Ereignissen eintreffen.

Beispiel: Einfache Temperaturregelung







Beschreibung Beispiel

- Ziel: Regelung der Temperatur (Betriebstemperatur 5-40 Grad Celsius) mittels eines sehr einfachen Reglers.
- 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 geschickt.

Esterel Code für Temperatur-Regelung (Auszug)

```
module TemperatureControler:
input TEMP: integer, SAMPLE_TIME, DELTA_T;
output HEATER_ON, HEATER_ON_STRONG,
       HEATER_OFF,VENTILATOR_ON, VENTILATOR_OFF,
       VENTILATOR_ON_STRONG, SIG_ABORT;

relation SAMPLE_TIME => TEMP;

  signal COLD, NORMAL, HOT in
    every SAMPLE_TIME do
      await immediate TEMP;
      if  ?TEMP<5 or ?TEMP>40 then emit SIG_ABORT
      elseif ?TEMP>=35 then emit HOT
      elseif ?TEMP<=10 then emit COLD
      else emit NORMAL
      end if
    end every
    ||
    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
```

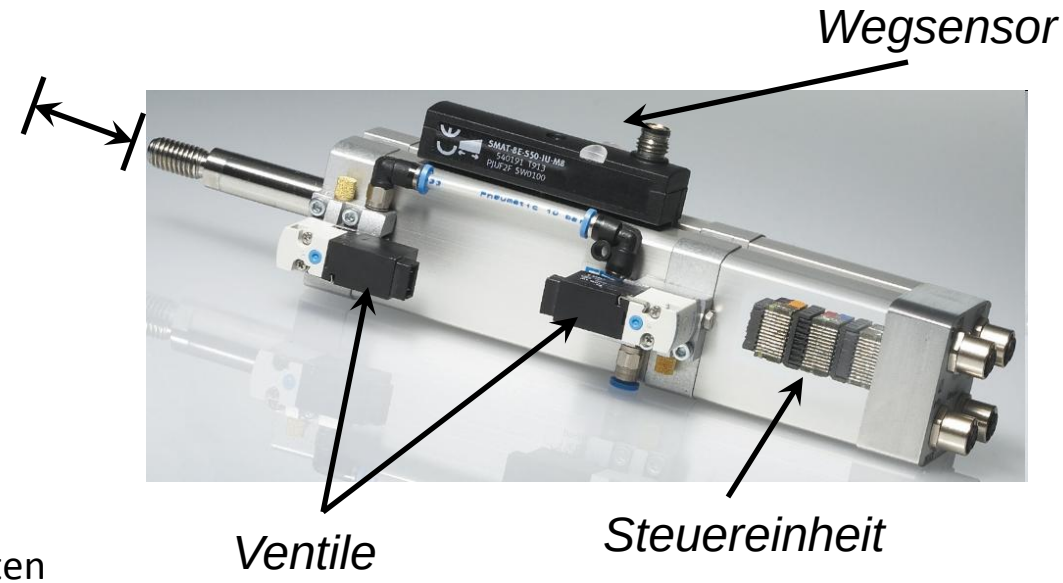
Modellierung von Echtzeitsystemen

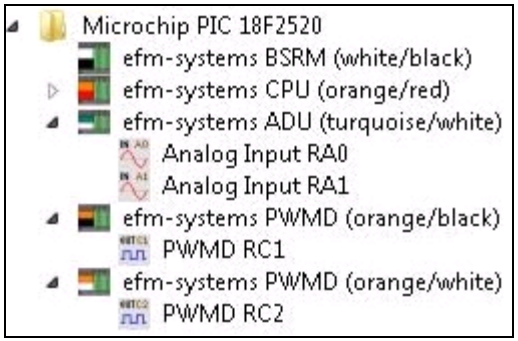
Synchroner Datenfluss

Werkzeug: EasyLab

Beispielanwendung – Pneumatischer Zylinder

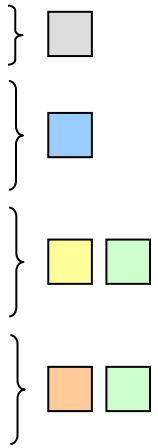
- Hardware
  - Zylinder
    - Positionssensor (Kolben)
    - Endlagenschalter
    - Zwei Magnetventile
  - Steuerungseinheit
    - Mikrocontroller
    - Analog-Digital-Wandler
    - Treiber für induktive Lasten
- Ziel: Positionssteuerung des Kolbens
- Umsetzung
  - Hardware-Modell aus Bibliothek für Match-X
  - Anwendungsmodell
    - Kleines Datenflussdiagramm
    - Integration der Hardwarefunktionalität

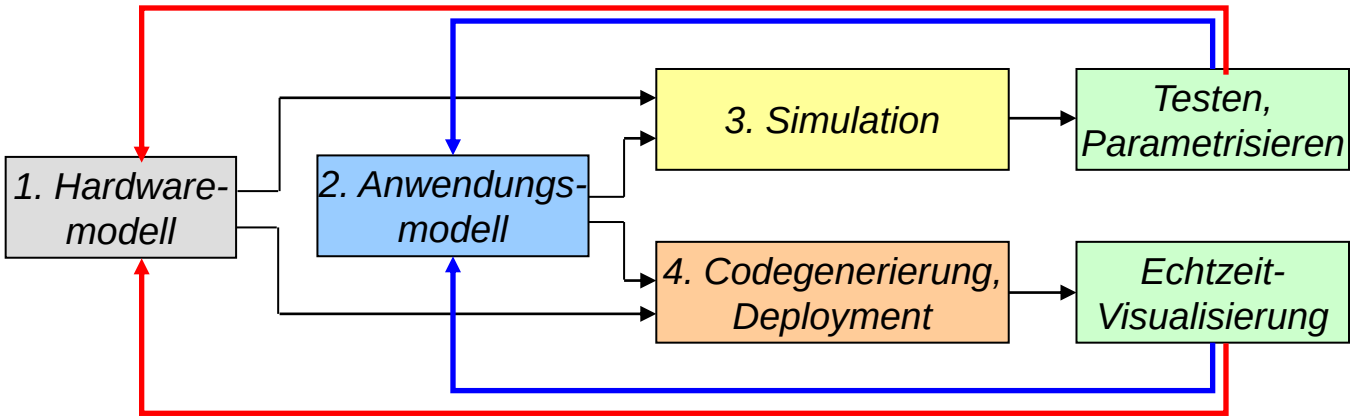




Entwicklungsprozess in EasyLab

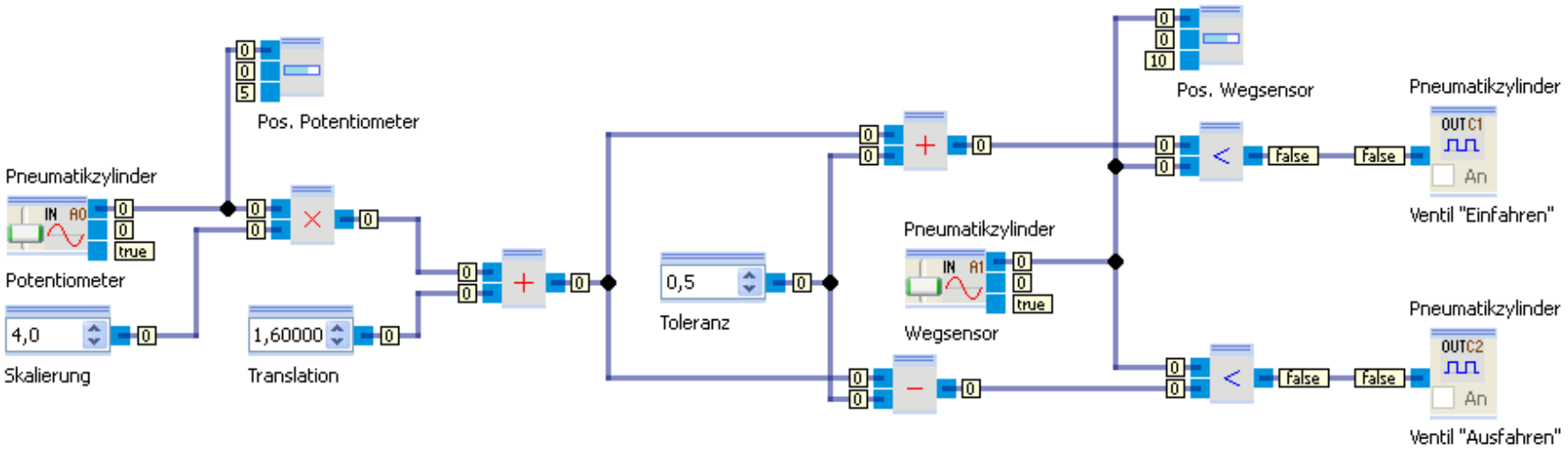
1. Spezifikation der Zielhardware
2. Modellierung der Zustandslogik sowie der abzuarbeitenden Aufgabe je Zustand
3. Simulation des Programms zum Testen und zur Erkennung von Fehlern
4. Codegenerierung und Echtzeit-Visualisierung des Zustands der Zielhardware





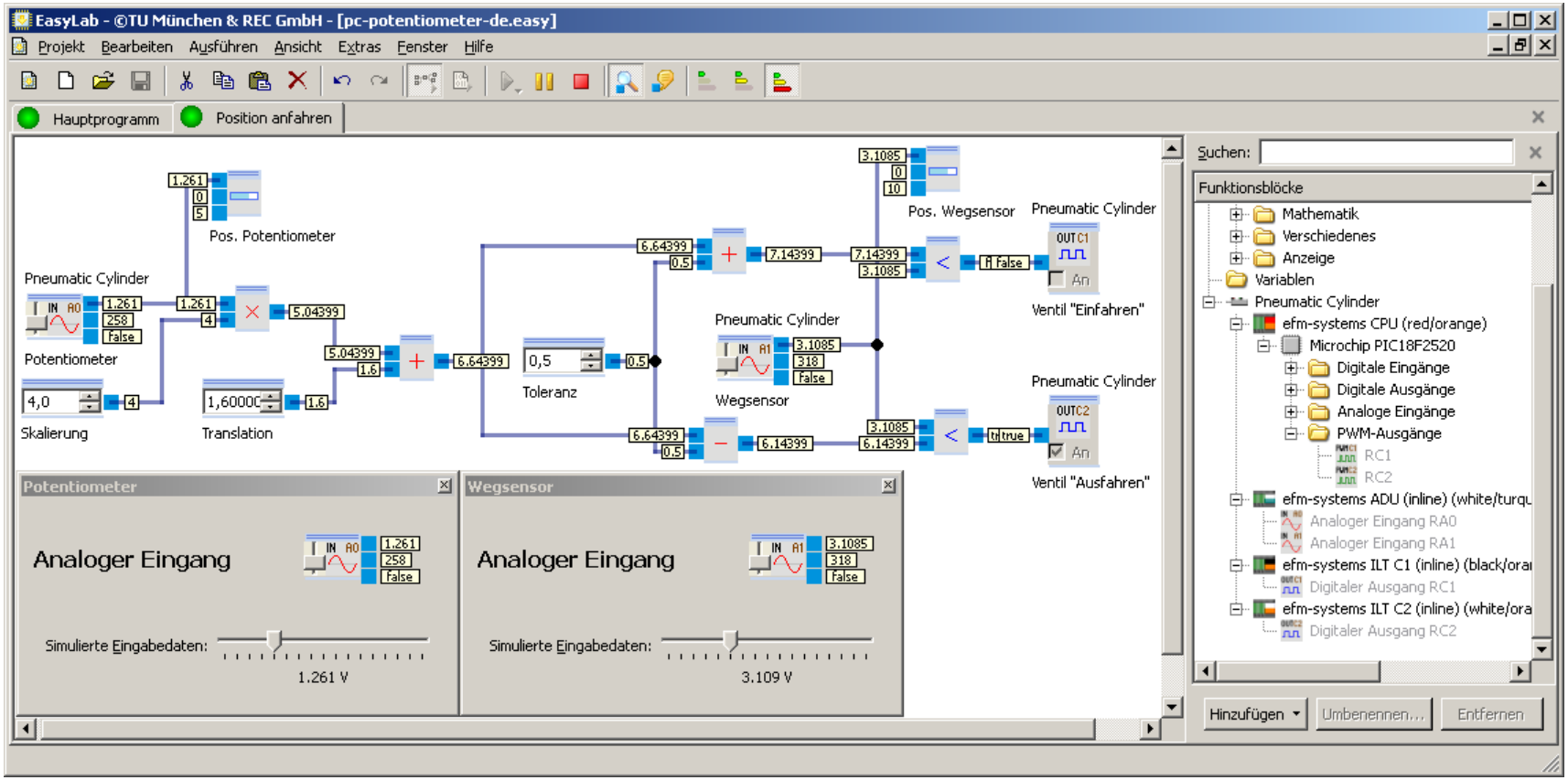
Synchroner Datenfluss

- Grundlagen
  - Synchronitätshypothese
  - „Black boxes“-Sicht
- Effizienz und Zuverlässigkeit
  - Berechnung statischer Schedules
  - Deterministisches Laufzeitverhalten
  - Statische Arbeitsspeicherallokation



Edward A. Lee and David G. Messerschmitt, “Static scheduling of synchronous data flow programs for digital signal processing,” IEEE Trans. Comput., vol. 36, no. 1, pp. 24–35, 1987

EasyLab - Demo



Beispielanwendung zur Regelung der Position eines Pneumatikzylinders

Erfolgskontrolle: Was sollten Sie aus dem Kapitel mitgenommen haben?

- Sensibilisierung für Semantik der verschiedenen modellbasierten Entwicklungswerkzeuge
- Kenntnisse der verschiedenen Models of Computation und ihrer Anwendungsfelder
- Beispiel für Klausurfragen (Klausur 2010/2011):

1. Reaktive, synchrone Sprachen wie Esterel basieren auf der Synchronitätshypothese. Erläutern Sie diese Hypothese, erklären Sie den wesentlichen Vorteil dieser Hypothese und beschreiben Sie, wie eine Implementierung aussieht. (6 Punkte = 6 Minuten)

   Antwort: siehe Folien, Implementierung durch Blockierung (z.B. Abschalten von Interrupts oder Zwischenspeichern)

2. Gegeben seien folgende drei Ausführungsmodelle
   - (i) Synchronous Dataflow (Synchroner Datenfluss)
   - (ii) Synchronous Reactive
   - (iii) Time-Triggered Execution (zeitgesteuerte Ausführung)

   und folgende drei Anwendungen:
   - 1. Fahrzeugsteuerung: Mehrere periodische Prozesse werden im verteilten System ausgeführt.
   - 2. Transaktionssystem: Eine Ampelsteuerung mit einem zentralen Steuerrechner soll entwickelt werden. Die Steuerung soll auf verschiedenste Ereignisse reagieren können und abhängig von diesen Ereignissen verschiedene Aktionen ausführen.
   - 3. Regelungsanwendung: Eine Spannungsquelle soll periodisch geregelt werden.

   Finden Sie eine passende Zuordnung von den Ausführungsmodellen (jeweils nur einmal verwenden) zu den Anwendungen und begründen Sie kurz Ihre Antwort. (6 Punkte = 6 Minuten)

   Antwort:
   - (i) Synchronous Data Flow <-> Regelungsanwendung: Synchronous Data Flow eignet sich vor allem für periodischen Berechnungen, die auf einem Rechner ausgeführt werden (da die Synchronitätshypothese so leichter umgesetzt werden kann).
   - (ii) Synchronous Reactive <-> Transaktionssystem: Synchronous Reactive eignet sich vor allem für die Modellierung und Entwicklung von ereignisbasierten Systemen, wie der Ampelsteuerung.
   - (iii) Time-Triggered Execution <-> Fahrzeugsteuerung: Zeitgesteuerte Ausführung ist ideal um periodische Prozesse im verteilten System auszuführen.

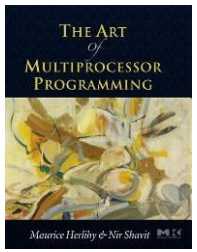
Kapitel 4

Nebenläufigkeit

Inhalt

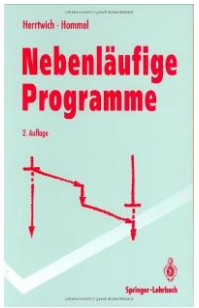
- Motivation
- Unterbrechungen (Interrupts)
- (Software-) Prozesse
- Threads
- Interprozesskommunikation (IPC)

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

- Edward Lee: The Problem with Threads: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf
- http://www.beyondlogic.org/interrupts/interupt.htm
- http://www.llnl.gov/computing/tutorials/pthreads/

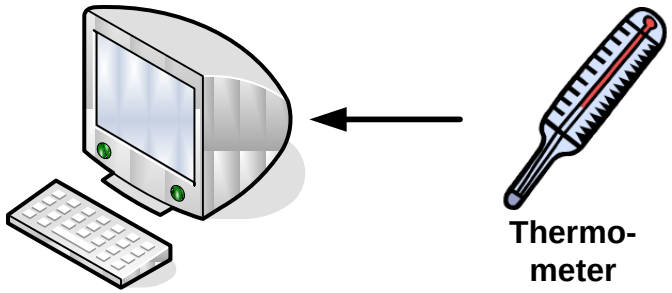
Definition von Nebenläufigkeit

- **Allgemeine Bedeutung**: Nebenläufige Ereignisse sind nicht kausal abhängig. Ereignisse (bzw. Ereignisfolgen) sind dann nebenläufig, wenn keines eine Ursache im anderen hat.
- **Bedeutung in der Informatik**: Nebenläufig bezeichnet hier die Eigenschaft von Programmcodes, nicht linear hintereinander ausgeführt werden zu müssen, sondern zeitlich parallel zueinander ausführbar zu sein.
- Aktionen (Programmschritte) können parallel (gleichzeitig oder quasi gleichzeitig) ausgeführt werden, wenn keine das Resultat der anderen benötigt. Die parallele Ausführung von mehreren unabhängigen *Prozessen* (siehe später) auf einem oder mehreren Prozessoren bezeichnet man als *Multitasking*. Die parallele Ausführung von Teilsequenzen innerhalb eines Prozesses heißt *Multithreading*.

Motivation

- Gründe für nebenläufige Ausführung von Programmen 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
- Aber: kleinere (Monoprozessor-)Echtzeit-Systeme verzichten häufig auf die parallele Ausführung von Code, weil der Aufwand für die Prozessverwaltung zu hoch ist.
  Dennoch auch hier: typischerweise Parallelverarbeitung in „Hauptprogramm“ und „Unterbrechungsbehandler“ (interrupt service routine, interrupt handler)

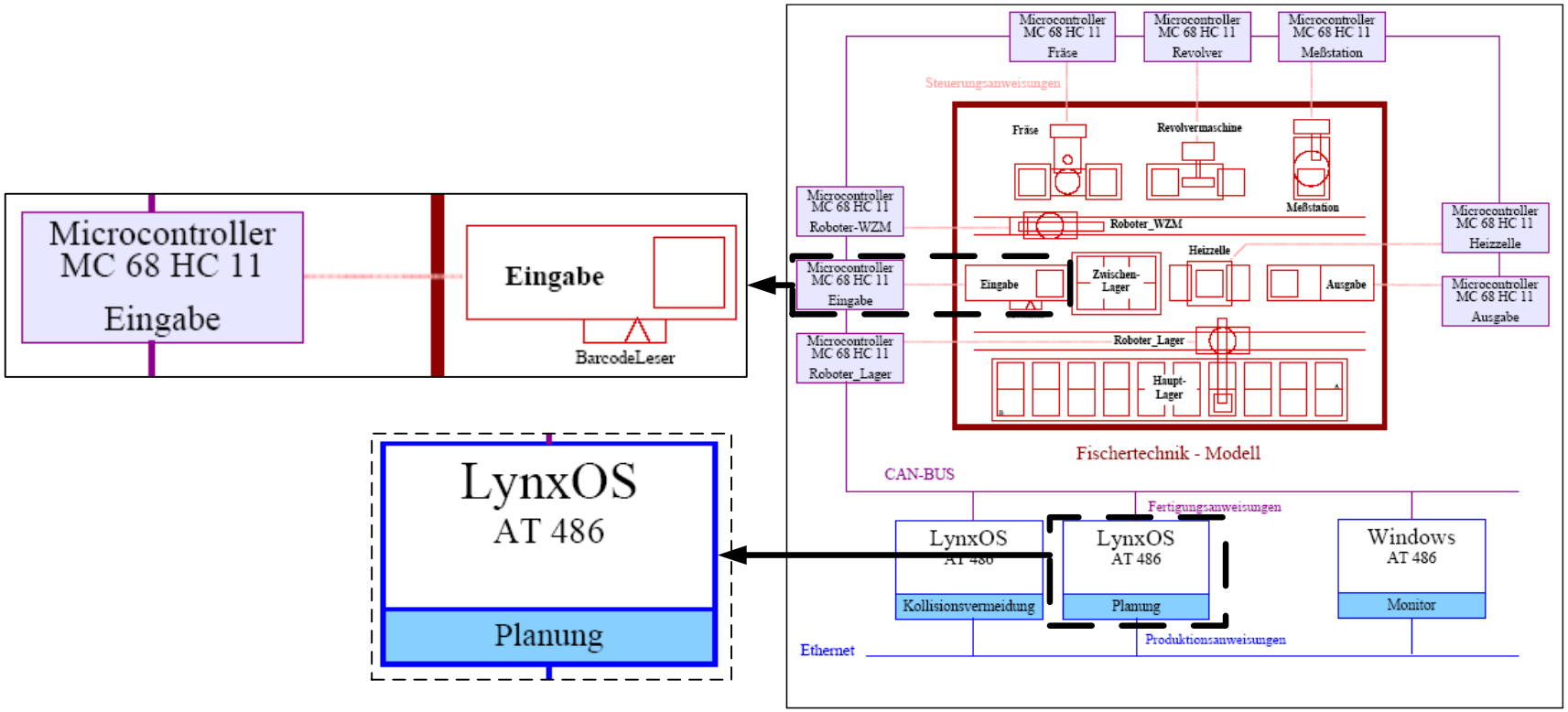
Anwendungsfälle für Nebenläufigkeit (Unterbrechungen)



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

Allgemeines Anwendungsgebiet: hauptsächlich zur Anbindung von externer Hardware

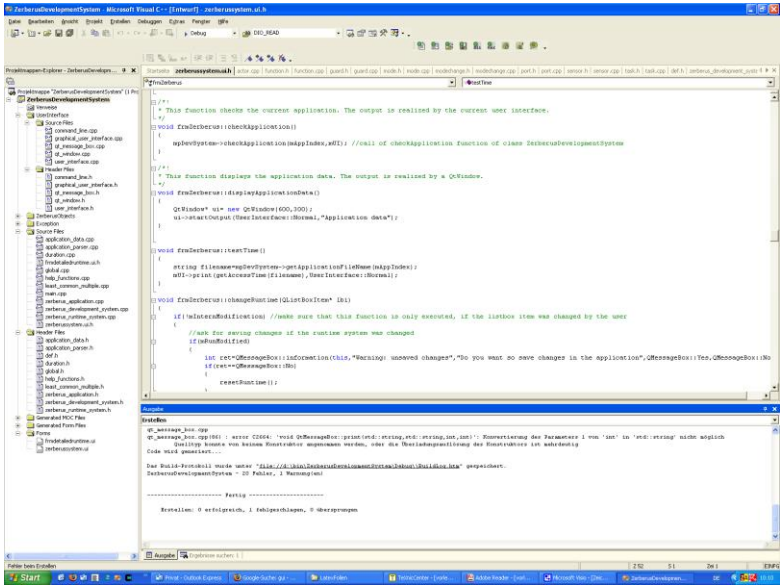
Anwendungsfälle für Nebenläufigkeit (Prozesse)



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)**

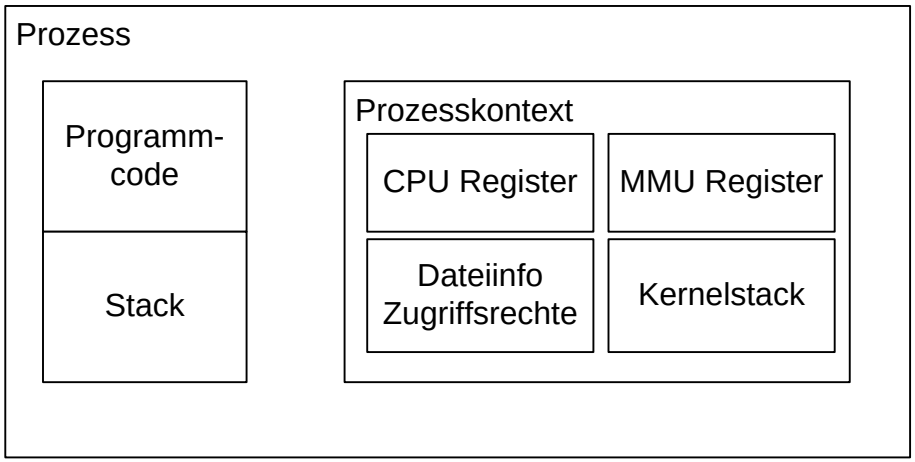
Allgemeines Anwendungsgebiet: unterschiedliche Berechnungen im gleichen Anwendungskontext

Nebenläufigkeit

Prozesse

Definition

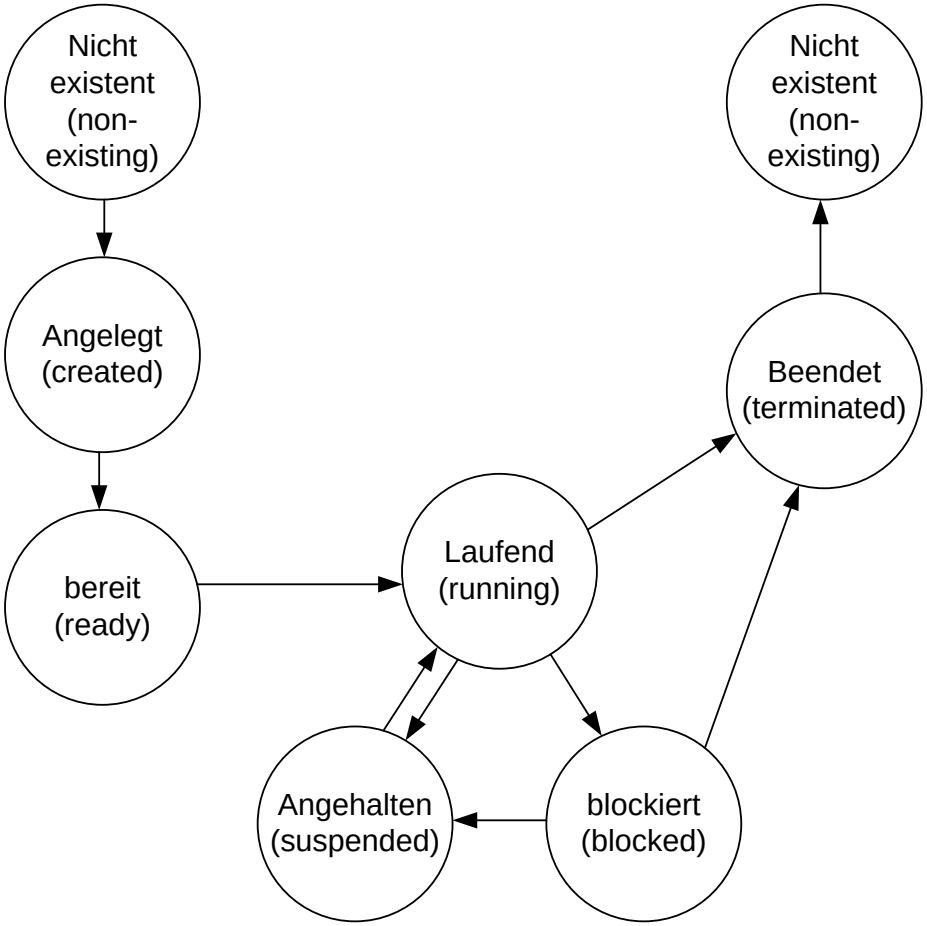
- **Prozess**: Abstraktion eines sich in Ausführung befindlichen Programms
- Die gesamte Zustandsinformation der Betriebsmittel für ein Programm wird als eine Einheit angesehen und als Prozess bezeichnet.
- Prozesse können weitere Prozesse erzeugen ⇨ Vater-,Kinderprozesse.



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

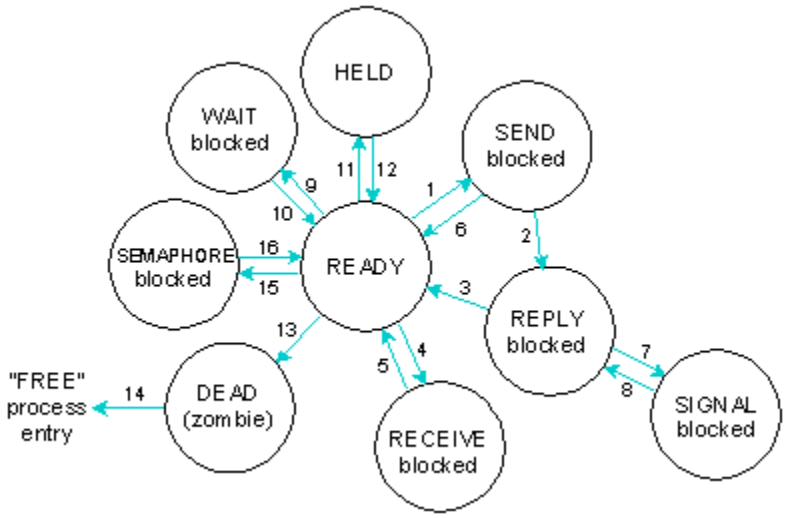
Prozesszustände (allgemein)



Prozesse in QNX[1]

The transactions depicted in the previous diagram are as follows:

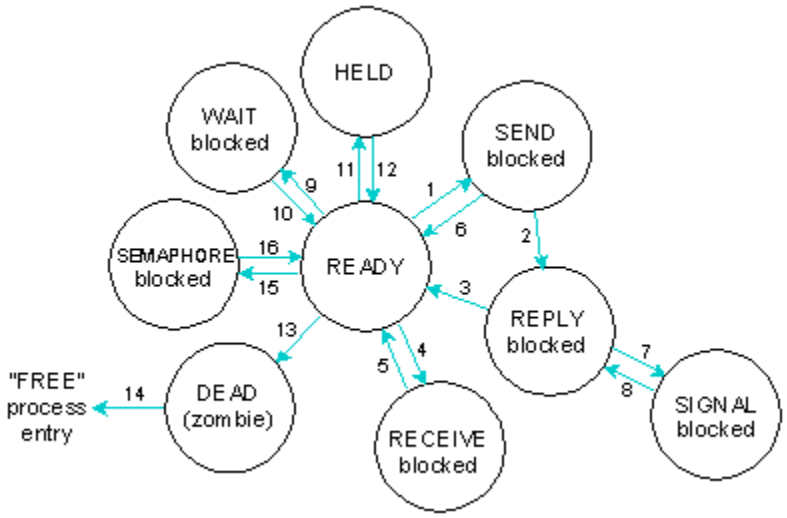
1. Process sends message.
2. Target process receives message.
3. Target process replies.
4. Process waits for message.
5. Process receives message.
6. Signal unblocks process.
7. Signal attempts to unblock process; target has requested message signal catching.
8. Target process receives signal message.



[1] http://www.qnx.com/developers/docs/qnx_4.25_docs/qnx4/sysarch/proc.html#LIFECYCLE

Prozesse in QNX

9. Process waits on death of child.
10. Child dies or signal unblocks process.
11. SIGSTOP set on process.
12. SIGCONT set on process.
13. Process dies.
14. Parent waits on death, terminates itself or has already terminated.
15. Process calls *semwait()* on a non-positive semaphore.
16. Another process calls *sempost()* or an unmasked signal is delivered.



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

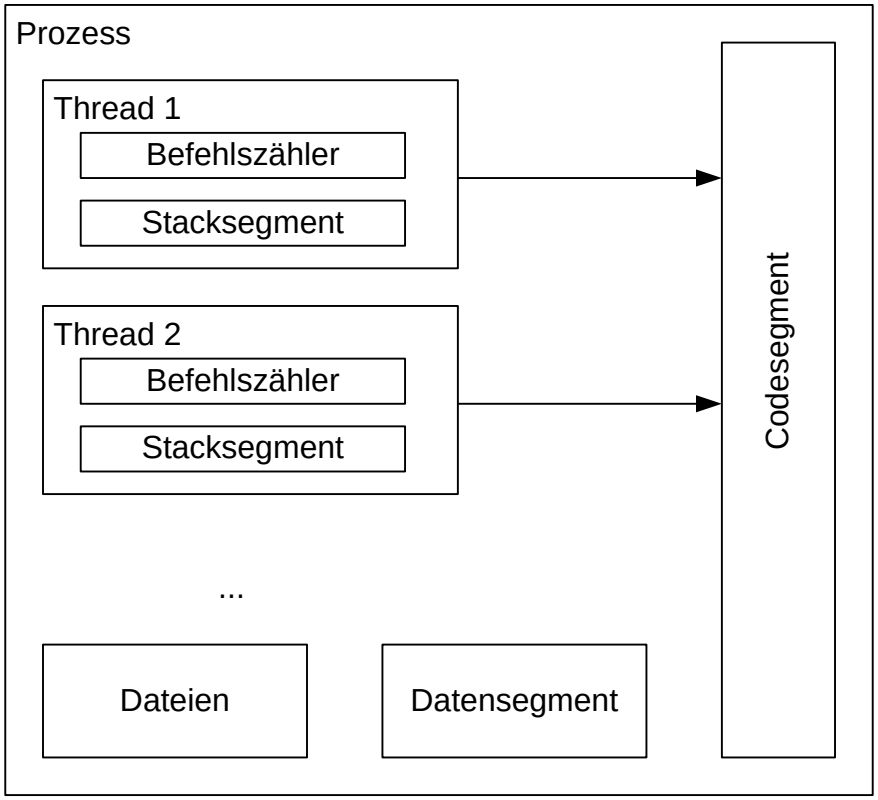
Threads

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

Unterbrechungen

Binding Rechnersystem-Umwelt

- Es muss ein Mechanismus gefunden werden, der es erlaubt, Änderungen der Umgebung (z.B. Druck einer Taste) zu registrieren.
- **1. Ansatz**: Abfrage (Polling)
  Es werden die E/A-Register reihum nach Änderungen abgefragt und bei Änderungen spezielle Antwortprogramme ausgeführt.
  - Vorteile:
    - bei wenigen EA-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: Einführung des Begriffs der Unterbrechung

- **2. Ansatz**: Unterbrechung (Interrupt)
- Eine Unterbrechung stoppt die Verarbeitung des laufenden Programms. Die Wichtigkeit des Ereignisses, welches die Unterbrechung ausgelöst hat, wird überprüft. Darauf basierend erfolgt die Entscheidung, welche Reaktion erfolgt.
- Vorteile:
  - Prozessorressourcen werden nur dann beansprucht, wenn es nötig ist
- Nachteile:
  - Nicht-Determinismus: Unterbrechungen asynchron zum Programmlauf (und zum Prozessorzustand) eintreffen.

Unterbrechungen

- **Unterbrechungen**: Stopp des Hauptprogrammablaufs, Aufnahme der Programmausführung eines „Unterbrechungsbehandlers (UBB)“ an einer anderen Stelle; nach Beendigung des UBB (zumeist) Rückkehr an die Stelle des Auftritts der Unterbrechung im Hauptprogramm.
- **Synchrone** Unterbrechungen: treten, falls sie auftreten, immer an *derselben Stelle* im Programmcode auf. Man bezeichnet sie auch als *Traps* oder *Exceptions* bzw. „Software-Interrupts“
- **Asynchrone** Unterbrechungen: Auftrittszeitpunkt ist unbestimmt; es kann nicht gesagt werden, an welcher Stelle der Hauptprogrammausführung der Prozessor zum Zeitpunkt der Unterbrechung ist. Asynchrone Unterbrechungen werden auch als Interrupts bezeichnet; weil sie von der Hardware-Peripherie erzeugt werden, auch als *Hardware-Interrupts*. Sie üben „Brückenfunktion“ zwischen Hardware und Software aus.

Synchrone Unterbrechungen (Traps/Exceptions)

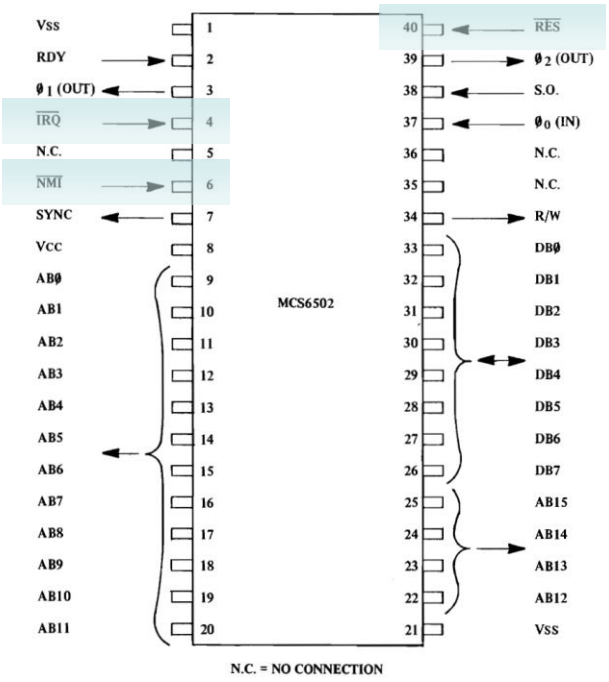
- Werden durch das Programm selbst ausgelöst, d.h. dasselbe Programm, ausgeführt mit denselben Parametern wird in der Regel an derselben Stelle dieselbe Unterbrechung auslösen (vorhersagbar in dieselbe „Falle“ laufen)
- **Auslösung bei Fehler** – Ausnahme/Exception, Beispiele:
  - Arithmetikfehler (Division by zero, overflow, not-a-number NaN, …)
  - Speicherfehler (Page Fault, segment Fault, memory full, …)
  - Befehlsfehler (Illegal instruction, privileged instruction, bus error, …)
  - Peripheriefehler (End-of-file EOF, channel blocked, unknown device, …)
- Bei Exceptions **nur dann** Rückkehr an den Auftrittspunkt, wenn die Fehlerbedingung im Ausnahmebehandler beseitigt werden kann, andernfalls Abbruch (resumption vs. termination)
- **Auslösung durch spezifische Instruktion**: Breakpoint, SWI, TRAP, INT, … entweder zum Zwecke des „Debuggings“ oder zum Aufruf von Betriebssystem-Diensten (z.B. MS-DOS „INT 21h“, siehe z.B. http://en.wikipedia.org/wiki/MS-DOS_API)
- Traps können auch benutzt werden, um einen Hardware-Interrupthandler zu testen.

Asynchrone Unterbrechungen (Interrupts)

- Werden durch externe Prozesse ausgelöst, d.h. sind bezüglich des genauen Auftrittszeitpunkts unvorhersagbar und zumeist nicht reproduzierbar
- Beispiele:
  - Signalisierung „normaler“ externer Ereignisse durch periphere Einheiten (Timer, Schalter, Grenzwertüber-/unterschreitung, …)
  - Warnsignale der Hardware (Energiemangel, „Watchdog-timer“ abgelaufen, …)
  - Beendigung einer Ein-/Ausgabeoperation (Wort von serieller Schnittstelle komplett empfangen oder komplett gesendet, Operation von Coprozessor (DMA, FPU) komplett, …)
- Die Unterbrechungsbehandlung muss *nebeneffektfrei* verlaufen, d.h. das Hauptprogramm darf nach Abschluss der Behandlung keinen (ungewünschten) veränderten Ausführungs-Kontext vorfinden.
- Aber: typischerweise wird der Behandler zum Zwecke der Kommunikation über globale Variable mit dem Hauptprogramm kommunizieren.

Technische Realisierung von Interrupts

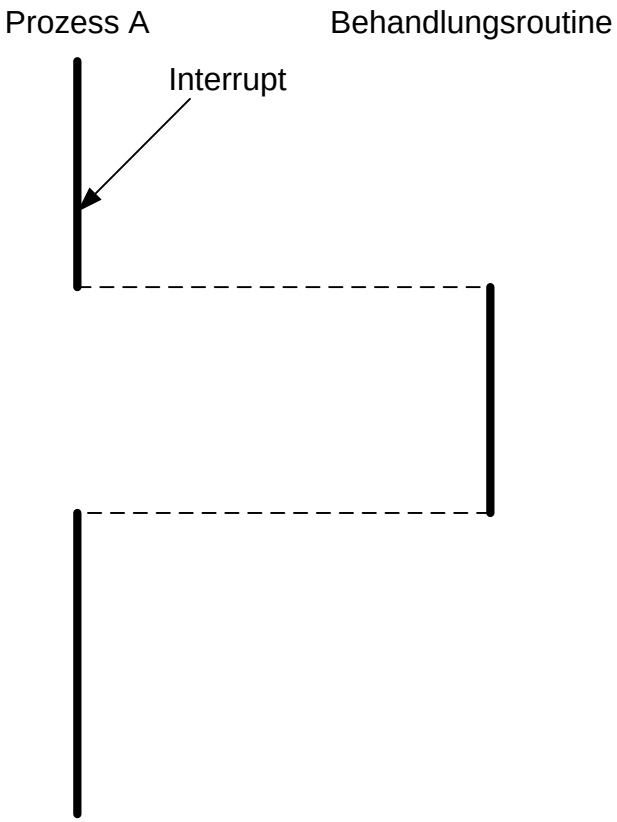
- Zur Realisierung besitzen Prozessoren einen oder mehrere spezielle Interrupt-Eingänge (typ. IRQ oder INT-Anschluß). Wird ein Interrupt aktiviert, so führt dies zur Ausführung einer Unterbrechungsbehandlungsroutine.
- Das Auslösen der Unterbrechungsroutine entspricht einem Unterprogrammaufruf. Der Programmablauf wird an einer anderen Stelle fortgesetzt und nach Beendigung der Routine (normalerweise) an der unterbrochenen Stelle fortgefahren.



Pin-Belegung des MOS 6502-Prozessors (z.B. Commodore 64).
Quelle: MCS 6500 HW-Manual, MOS Technology, Jan. 1976

Durchführung einer einfachen INT-Behandlung

1. IRQ-Anschluß wird durch peripheres Gerät aktiviert
2. Wenn Interrupts momentan zugelassen sind: Beendigung der Abarbeitung der gerade noch laufenden Instruktion
3. Sperren von Unterbrechungen mit gleicher oder niedrigerer Priorität
4. Sicherung der Register des Prozessors (Prozessorkontext) auf dem Stapelspeicher (Stack) – insbesondere Sicherung des Programmzählers; dafür spezielle Instruktionen verfügbar
5. Sprung an den Behandler (entspricht Laden des Programmzählers mit der Programmstartadresse des Behandlers) – dies kann auf verschiedene Arten erfolgen, siehe unten
6. Ausführung des Codes des Behandlers, an dessen Ende steht ein „Return from Interrupt“-Befehl; dabei Signalisierung an Peripherie, dass Behandlung abgeschlossen
7. Rückkehr zur Programmausführung (nicht unbedingt der unterbrochene Prozess) durch Zurückladen des gesicherten Prozessorregistersatzes.



Beispiel INT beim 6502

- Bei Vorliegen eines IRQ wird der 16-Bit Programmzähler mit dem Inhalt der 8-Bit Adressen FFFE und FFFF geladen.
- FFFE/F enthält Adresse des (= *Vektor* auf) IRQ-Behandler
- FFFC/D enthält Adresse des Reset-Behandlers
- FFFA/B enthält Adresse des NMI-Behandlers. NMI: Non-Maskable-Interrupt, kann nicht abgeschaltet werden (also auch nicht durch fehlerhaftes Programm)

Speicher-Aufteilung für 6502 (in Hex-Adressen)

0000-00FF - RAM for Zero-Page & Indirect-Memory Addressing

0100-01FF - RAM for Stack Space & Absolute Addressing

0200-3FFF - RAM for programmer use

4000-7FFF - Memory mapped I/O

8000-FFF9 - ROM for programmer usage

FFFA - Vector address for NMI (low byte)

FFFB - Vector address for NMI (high byte)

FFFC - Vector address for RESET (low byte)

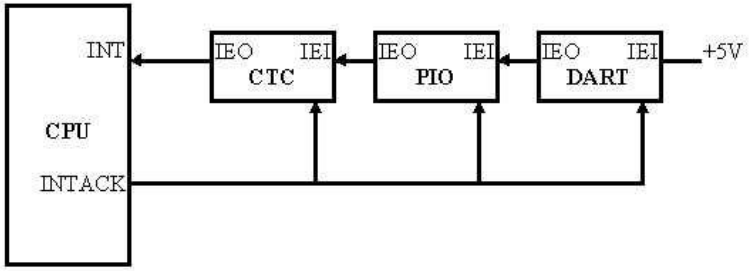
FFFD - Vector address for RESET (high byte)

FFFE - Vector address for IRQ & BRK (low byte)

FFFF - Vector address for IRQ & BRK (high byte)

Behandlung mehrerer Quellen von Interrupts

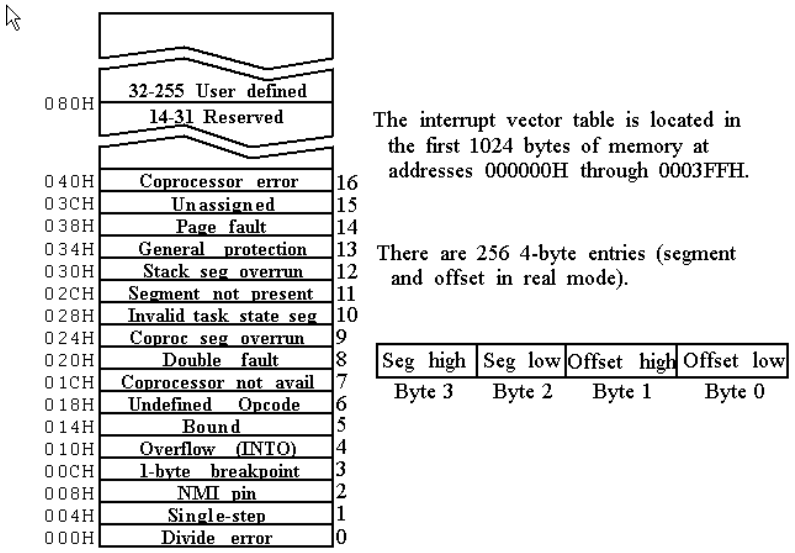
- Beim 6502 nur *ein* IRQ-Eingang und nur *ein* Vektor auf *einen* Behandler. Bei mehreren Quellen für Interrupts (z.B. serielle Schnittstelle, Parallele Schnittstelle, Timer) werden daher die Interrupt-Ausgänge dieser Einheiten „verodert“ und an den IRQ-Eingang gelegt. Alle Quellen sind damit gleichberechtigt.
- Damit ist keine automatische HW-Priorisierung (nach Wichtigkeit der eintreffenden Interrupts) möglich.
- Der Behandler muß nach dem Auftritt des Interrupts Quelle für Quelle abfragen, welche den Interrupt verursacht hat (implizite Priorisierung je nach Abfragereihenfolge)
- Einfache Möglichkeit der HW-Priorisierung: Daisy-Chain (Prioritätskette). Die Einheit, die am nächsten an der CPU liegt, hat die höchste Priorität (siehe Bild rechts) und sperrt Interrupts anderer Quellen solange aus, bis der eigene Interrupt abgearbeitet ist (INTACK durch Prozessor signalisiert).



Daisy-Chaining beim Z80-Prozessor
CTC: Counter-Timer-Circuit
PIO: Parallel In-/Out
DART: Dual Asynchronous Receiver Transmitter

Vektorisierte Interrupt-Behandlung

- Verfügt ein Rechnersystem über viele Interrupt-Quellen (wie typischerweise im Bereich eingebetteter Systeme), ist es zweckmäßig, für jedes Gerät (mindestens) einen Behandler vorzusehen und diesen nach dem Ereignisauftritt auch direkt ausführen zu können.
- Dazu Einführung von vektorbasierten Interrupt-Systemen.
- Prinzip: Gerät erzeugt nicht nur einen Interrupt, sondern liefert dem Prozessor parallel auch eine eigene Kennung (z.B. 8-Bit Wert A). Dieser Wert A verweist auf einen Tabelleneintrag, an dem sich die Einsprung-Adresse des Behandlers befindet.
- Exceptions und Interrupts folgen dem gleichen Schema, d.h. Tabelleneintrag = Einsprungadresse für Exception/Interrupt-Handler



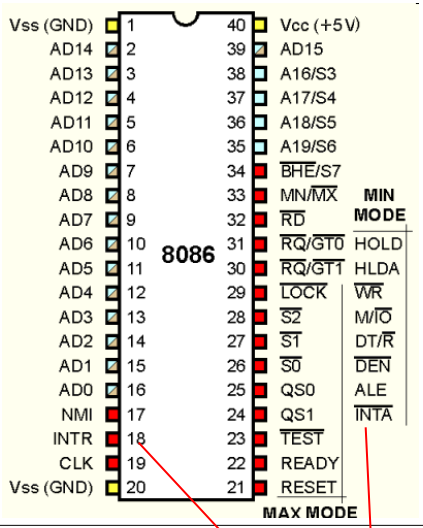
- Typische Interrupt-Vektor-Tabelle eines 8086-Systems im „Real Mode“
- 32-Bit-Adresse (4 Byte) pro Eintrag
- 0 bis 31 sind Prozessor-interne Ausnahmen
- Hardware-Interrupts können über Interrupt-Controller auf beliebige Tabellenplätze gelegt werden

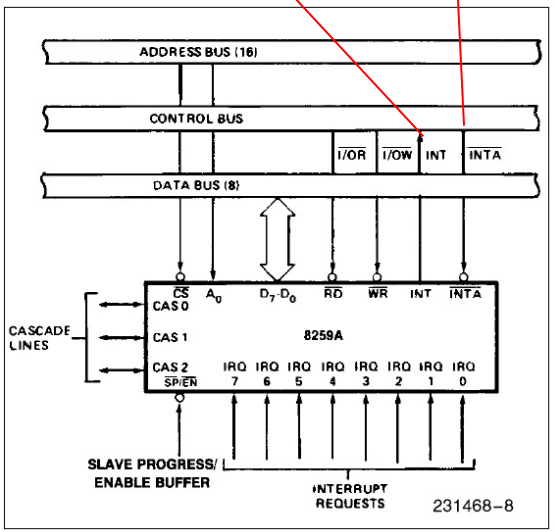
Interrupt-(Priority)-Controller

- Typischerweise wird mit der vektorbasierten Verwaltung auch eine Prioritätsverwaltung eingeführt. Klassisches Beispiel: der Interrupt-Controller 8259A des IBM-PC.
- Verwaltet 8 Hardware-Interrupts und kann diese mit Prioritäten belegen. Ist kaskadierbar, der PC/AT hatte zwei 8259A.
- Typen der Interrupts beim PC/AT:

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 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 IDE-Kanal

- Heute Standard (seit ca. 2002): Nachfolgekonzept, Intel APCI 82093AA, sehr viel höhere Anzahl von Interrupts realisiert





Nebenläufigkeit

Probleme

Probleme

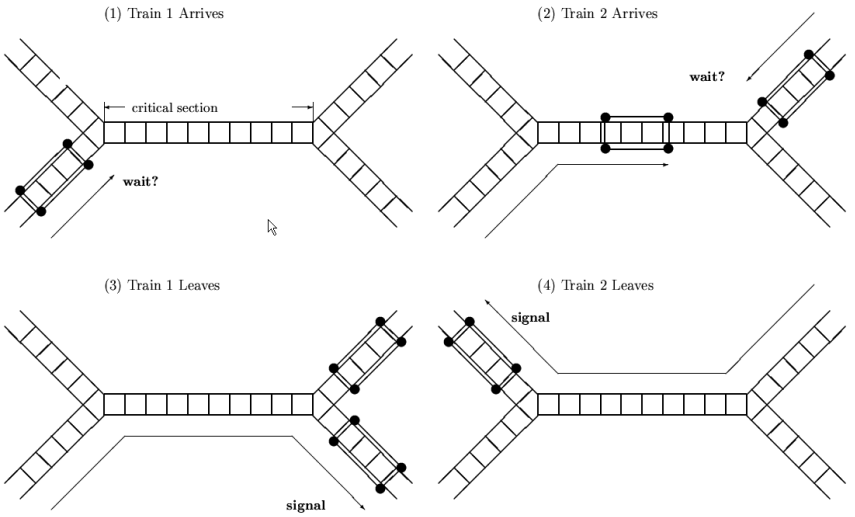
- **Race Conditions**:
  - Situationen, in denen zwei oder mehrere Threads/Prozesse, die gleichen geteilten Daten lesen oder schreiben und das Resultat davon abhängt, wann genau welcher Prozess ausgeführt wurde, werden Race Conditions genannt.
  - Lösung: Einführung von **kritischen Bereichen** und **wechselseitiger 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.

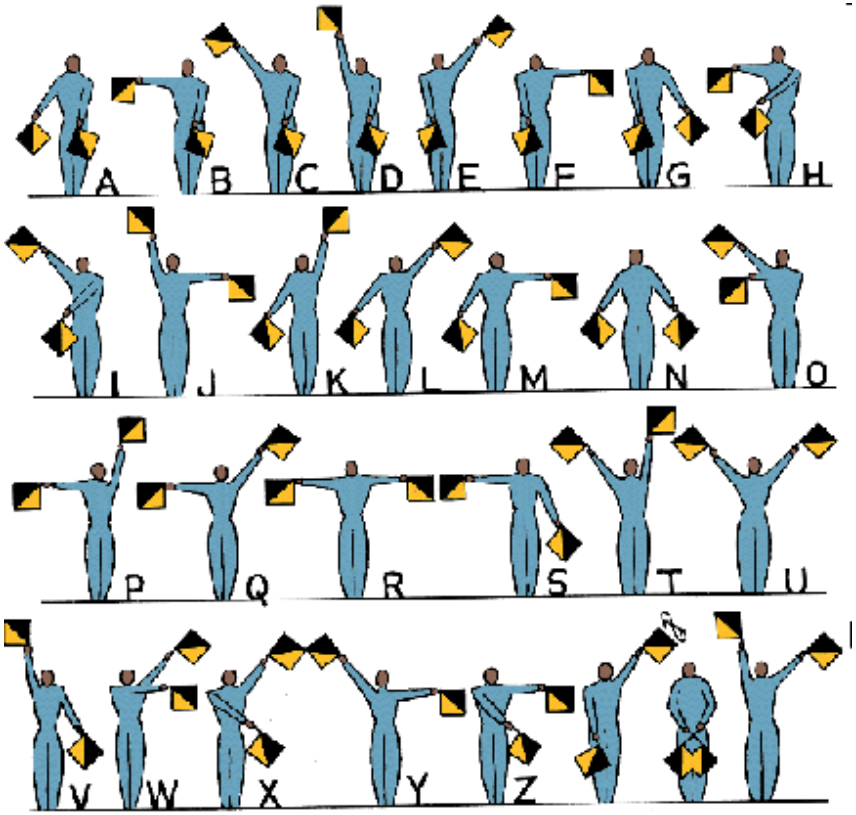
Kritische Bereiche

- Um einen kritischen 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).



Programmtechnik zum Schutz kritischer Bereiche

- Aus dem Alltag sind diverse Mechanismen zum Schutz kritischer Bereiche bekannt:
  - *Signale* im Bahnverkehr
  - *Ampeln* zum Schutz der Kreuzung
  - *Schlösser* für einzelne Räume
  - *Vergabe* von Tickets
- Erstes Mittel zum Schutz kritischer Bereiche in der Softwaretechnik: der/das Semaphor, griech. für „Zeichenträger“ (siehe später und rechts)



Falsche Lösung: Verwendung einer globalen Variable

Prozeß A

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

Prozeß B

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

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

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

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

Deklaration globale Variablen

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

Prozess 0

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

Prozess 1

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

- Ausschluss ist garantiert, aber „busy waiting“ verschwendet immer noch Rechenzeit
- Die Realisierung für N Prozesse ist als „Lamport‘s Bakery Algorithmus“ bekannt: http://en.wikipedia.org/wiki/Lamport‘s_bakery_algorithm

2. Möglichkeit: Ausschalten von Unterbrechungen zum WA

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

5.4.5 Interrupt Enable Clear register (VICIntEnClear - 0xFFFF F014)

This is a write only register. This register allows software to clear one or more bits in the Interrupt Enable register (see Section 5.4.4 "Interrupt Enable register (VICIntEnable - 0xFFFF F010)" on page 52), without having to first read it.

Table 42: Software Interrupt Clear register (VICIntEnClear - address 0xFFFF F014) bit allocation
Reset value: 0x0000 0000

Bit	31	30	29	28	27	26	25	24
Symbol	-	-	-	-	-	-	-	-
Access	WO	WO	WO	WO	WO	WO	WO	WO
Bit	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**
Symbol	-	-	AD1	BOD	I2C1	AD0	EINT3	EINT2
Access	WO	WO	WO	WO	WO	WO	WO	WO
Bit	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**
Symbol	EINT1	EINT0	RTC	PLL	SPI1/SSP	SPI0	I2C0	PWM0
Access	WO	WO	WO	WO	WO	WO	WO	WO
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Symbol	UART1	UART0	TIMER1	TIMER0	ARMCore1	ARMCore0	-	WDT
Access	WO	WO	WO	WO	WO	WO	WO	WO

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 Zählvariable `s`, 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.

Realisierungen von Semaphoren

- Die Implementierung eines Semaphors erfordert spezielle Mechanismen auf Maschinenebene; der Semaphor ist für sich ein kritischer Bereich.
  → 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. **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 anderen Lösungen mit Hilfe von Warteschlangen sehr effizient realisiert werden.
  3. **Test&Set**-Maschinenbefehl: Die meisten Prozessoren verfügen heute über einen Befehl „**Test&Set**“ (oder auch Test&SetLock). Dieser lädt atomar den Inhalt (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.

Beispiel: Bankkonto

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

Prozess A

```
down(semAccount);
x=readAccount(account);
x=x+500;
writeAccount(x,account);
up(semAccount);
```

Prozess B

```
down(semAccount);
y=readAccount(account);
y=y-200;
writeAccount(y,account);
up(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);
```

- Leser-Schreiber-Probleme sind vielfältig modifizierbar, je nach Priorität der prozesse. LS-Problem: Keine Prioritäten. Erstes LS-Problem: Leserpriorität. Zweites LS-Problem: Schreiber-Priorität.

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?

Leser-Schreiber-Realisierung mit Semaphoren

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 –
siehe spätere Lösung

Verbessertes Konzept: Monitore

- Ein Nachteil von Semaphoren ist die Notwendigkeit zur expliziten Anforderung P/DOWN und Freigabe V/UP des kritischen Bereiches durch den Programmierer
- Vergisst der Entwickler z.B. die Freigabe V/UP 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.

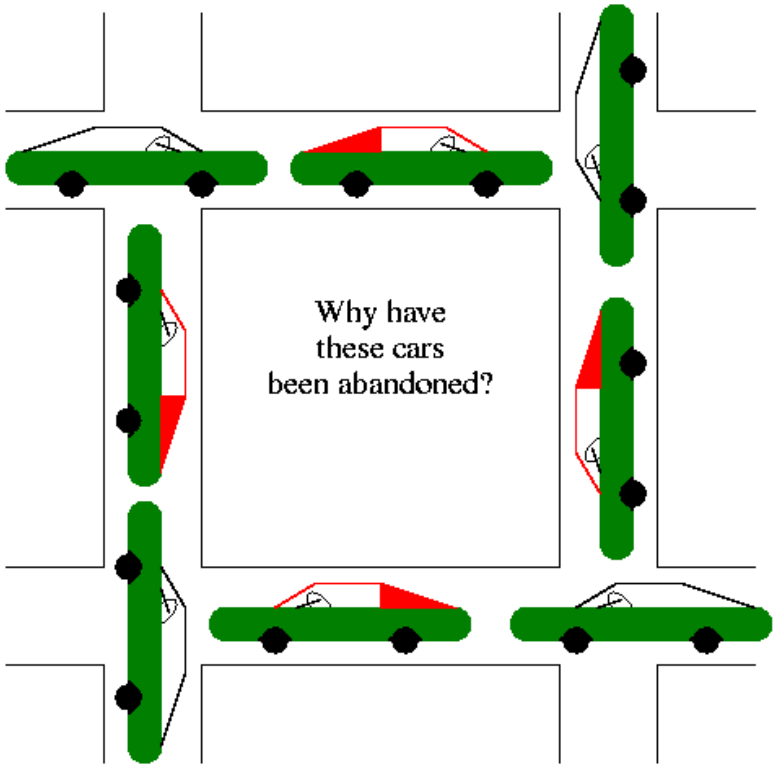
Beispiel: Monitore in Java

- 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- -;
        }
}
```

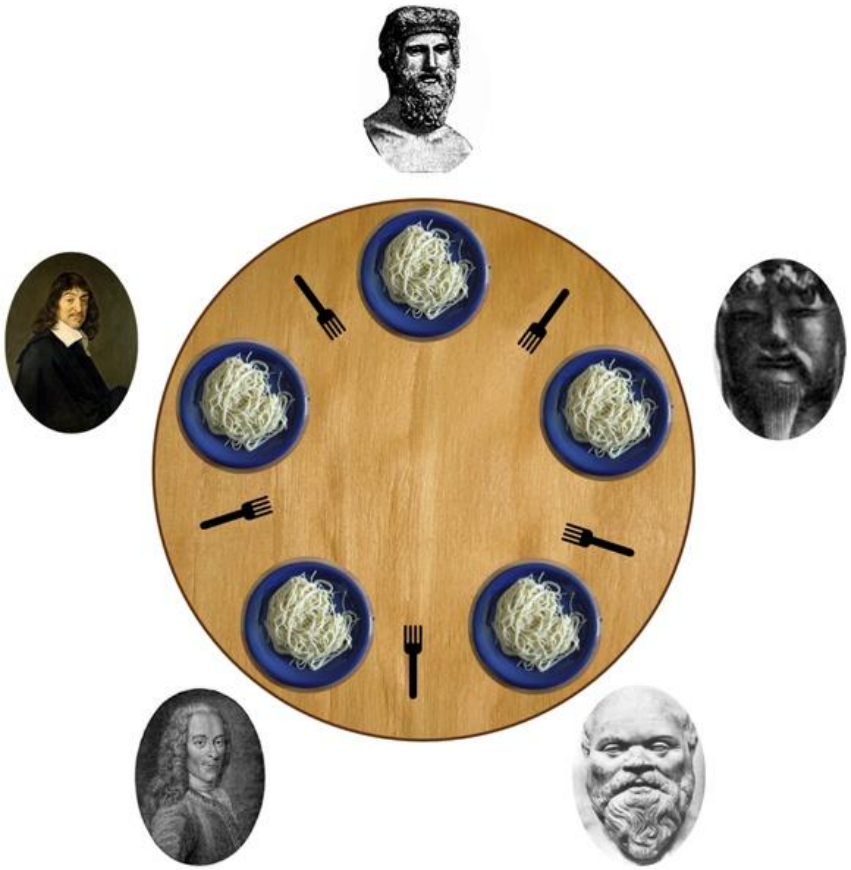
Bemerkung zu Verklemmungen (Deadlocks)

- 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

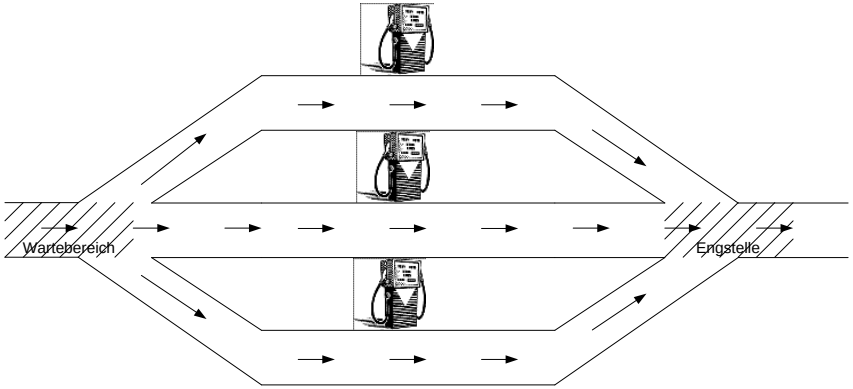
- Klassisches Beispiel aus der Informatik für Verklemmungen: "Dining Philosophers" (speisende Philosophen, Dijkstra 1971, Hoare 1971)
- 5 Philosophen (Prozesse) sitzen an einem Tisch. Vor ihnen steht jeweils ein Teller mit Essen. Zum Essen benötigen sie zwei Gabeln (Betriebsmittel), insgesamt sind aber 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)**.



Klausur WS06/07 – Nebenläufigkeit (15 Punkte = 15min)

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.

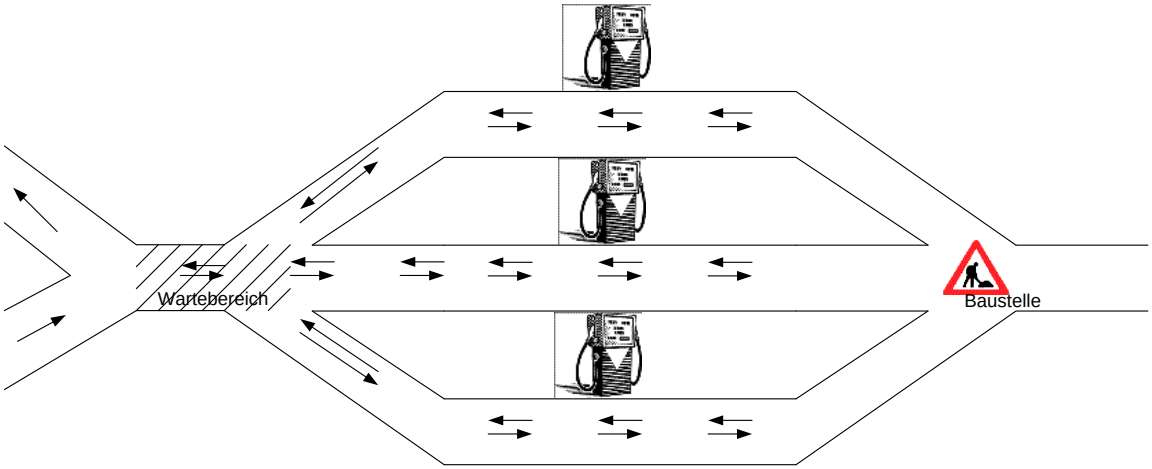
b) Ergänzen Sie den folgenden Autoprozess mit passenden up() und down()-Methoden, um Kollisionen zu vermeiden. Achten Sie darauf, dass es zu keiner Verklemmung kommt. **Anmerkung**: Es muss nicht an jeder freien Stelle Code eingefügt werden.
Beispiel: 1: down(semaAuto); up(semaAuto); bedeutet das Einfügen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

Klausur WS06/07 - Nebenläufigkeit

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?

Prozess: tankendes Auto

```
fahreInWartebereich();

fahreAnZapfsaeule();

tanke();

bezahle();

fahreInEngstelle2();

verlasseEngstelle2();
```



Klausur WS06/07 – Nebenläufigkeit Lösung

Aufgabe 3 Nebenläufigkeit **(15 Punkte)**

a) semWartebereich(1);semZapfsaeule(3), semEngstelle(1);

b) siehe Algorithmus

Algorithm 1 Prozess: tankendes Auto

```
1:  down(semWartebereich);
2:  fahreInWartebreiche();
3:  down(semZapfsaeule);
4:  fahreAnZapfsaeule();
5:  up(semWartebereich);
6:  tanke();
7:
8:  bezahle();
9:  down(semEngstelle);
10: fahreInEngstelle();
11: up(semZapfsaeule);
12: verlasseEngstelle();
13: up(semEngstelle);
```

c) Ja. Bevor der Semaphor semWartebereich angefordert wird, muss noch der Semaphor für die Zapfsäule angefordert werden. Ansonsten kann es zur Verklemmung kommen.

Nebenläufigkeit

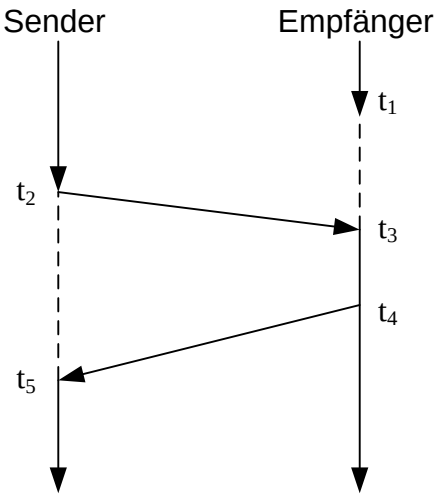
Interprozesskommunikation (IPC)

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

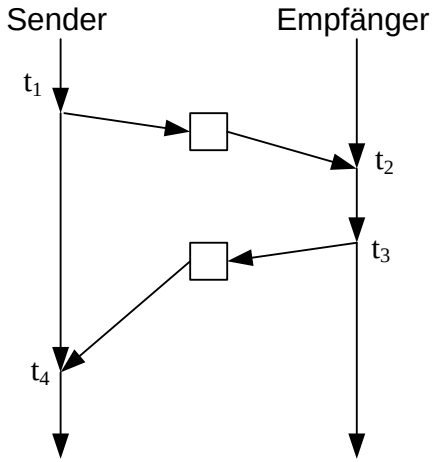
Synchron vs. Asynchron

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)`.

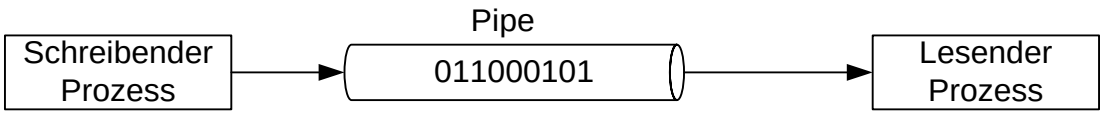
Fragestellungen beim Datenaustausch

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

In diesem Kapitel 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

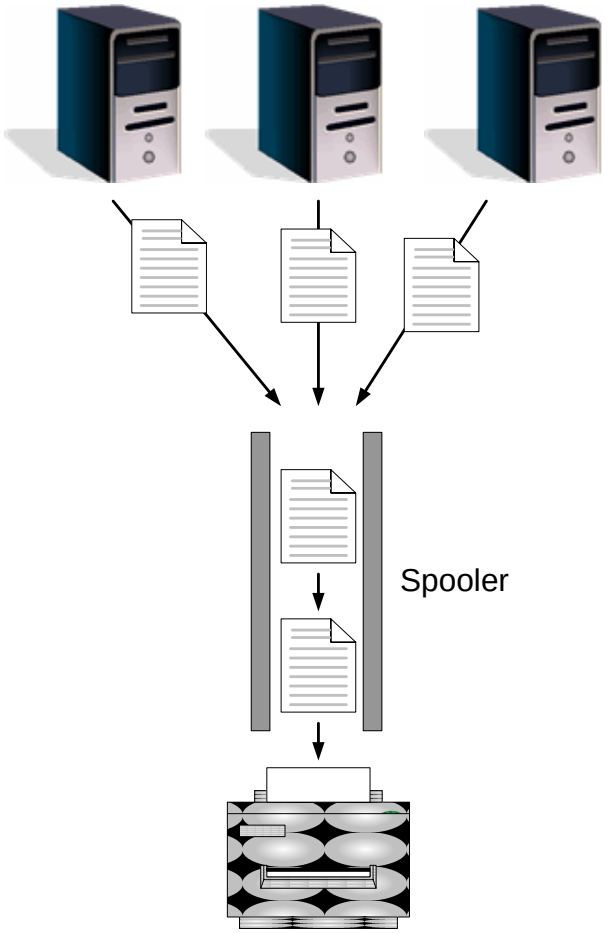
- Pipes bringen einige Nachteile mit sich:
  - Pipes sind nicht nachrichtenorientiert (keine Bündelung der Daten in einzelne Pakete (Nachrichten) möglich).
  - Daten sind nicht priorisierbar.
  - Der für die Pipe notwendige Speicherplatz wird erst während der Benutzung angelegt.
- Wichtig für die Implementierung: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

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:

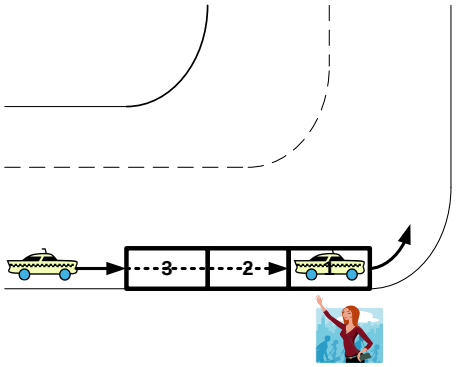
```
...
            down(semWriter);
            down(semCounter);
            rcounter++;
            if(rcounter==1)
                        down(semReader);
            up(semCounter);
            up(semWriter);

            read();

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

Writer:

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

            write();


            up(semWriter);
...
```

Klausur WS07/08 – Nebenläufigkeit (20 Punkte = 20min)

- 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.

Klausur WS07/08 – Nebenläufigkeit - Lösung

Algorithm 1 Lösung Nebenläufigkeit: Benötigte Semaphoren

```
1: semBucht3(1);
2: semBucht2(1);
3: semBucht1(1);
4: semTaxi(0);
5: semPassagier(0);
6: semTest(1);
7: int frei=1;
```

Algorithm 2 Lösung Nebenläufigkeit: Taxiprozess

```
1: down(semBucht3);
2: fahreInBucht3();
3: down(semBucht2);
4: fahreInBucht2();
5: up(semBucht3);
6: down(semBucht1);
7: fahreInBucht1();
8: up(semBucht2);
9: up(semTaxi);
10: down(semPassagier);
11: fahreLos();
12: up(semBucht1);
```

Algorithm 3 Lösung Nebenläufigkeit: Passagierprozess

```
1: down(semTaxi);
2: steigeEin();
3: up(semPassagier);
```

Algorithm 4 Lösung Nebenläufigkeit: Überprüfung

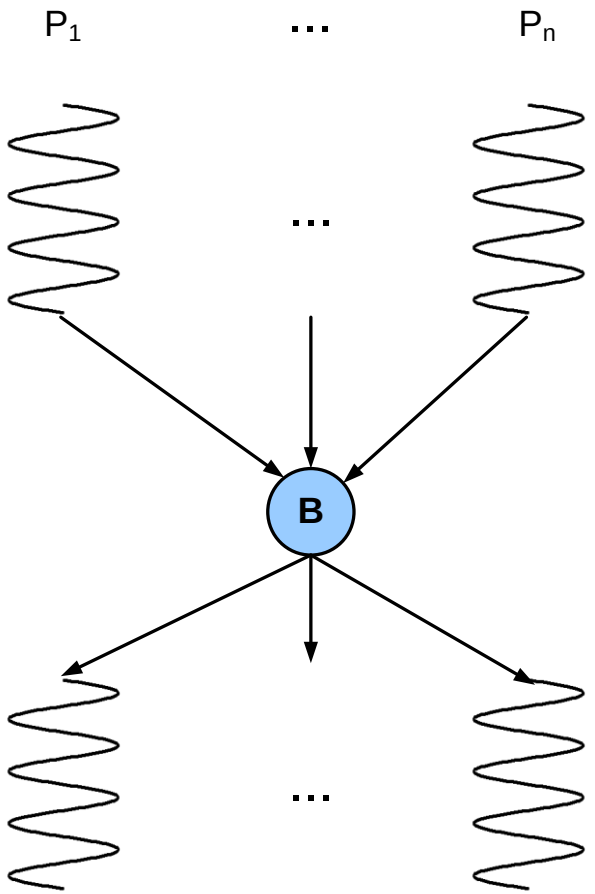
```
1: down(semTest);
2: if frei==1 then
3:     frei=0;
4:     up(semTest);
5:     fahreInBucht3();
6:     down(semBucht2);
7:     fahreInBucht2();
8:     down(semTest);
9:     frei=1;
10:    up(semTest);
11:    ...
12: else
13:    up(semTest);
14:    fahreWeiter();
15: end if
```

Nebenläufigkeit

Synchrone Kommunikation: Barrieren

Synchrone Kommunikation: Barrieren

- **Definition**: Eine Barriere für eine Menge *M* von Prozessen ist ein Punkt, den alle Prozesse $P_i 2 M$ 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 (→ Hausaufgabe).



Erfolgskontrolle: Was Sie aus dem Kapitel mitgenommen haben?

- Definition und Gründe für Nebenläufigkeit
- Arten der Umsetzung von Nebenläufigkeit (Prozesse, Threads, Interrupts), Technische Umsetzung, Unterschiede und Anwendungsgebiete
- Probleme (Race Conditions, Verklemmungen, Starvation) , die durch Nebenläufigkeit entstehen und passende Lösungen
  - Insbesondere Anwendung von Semaphoren
- Mechanismen zur Interprozesskommunikation

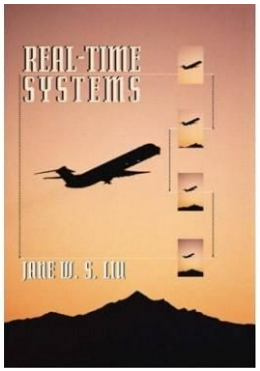
Kapitel 5

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

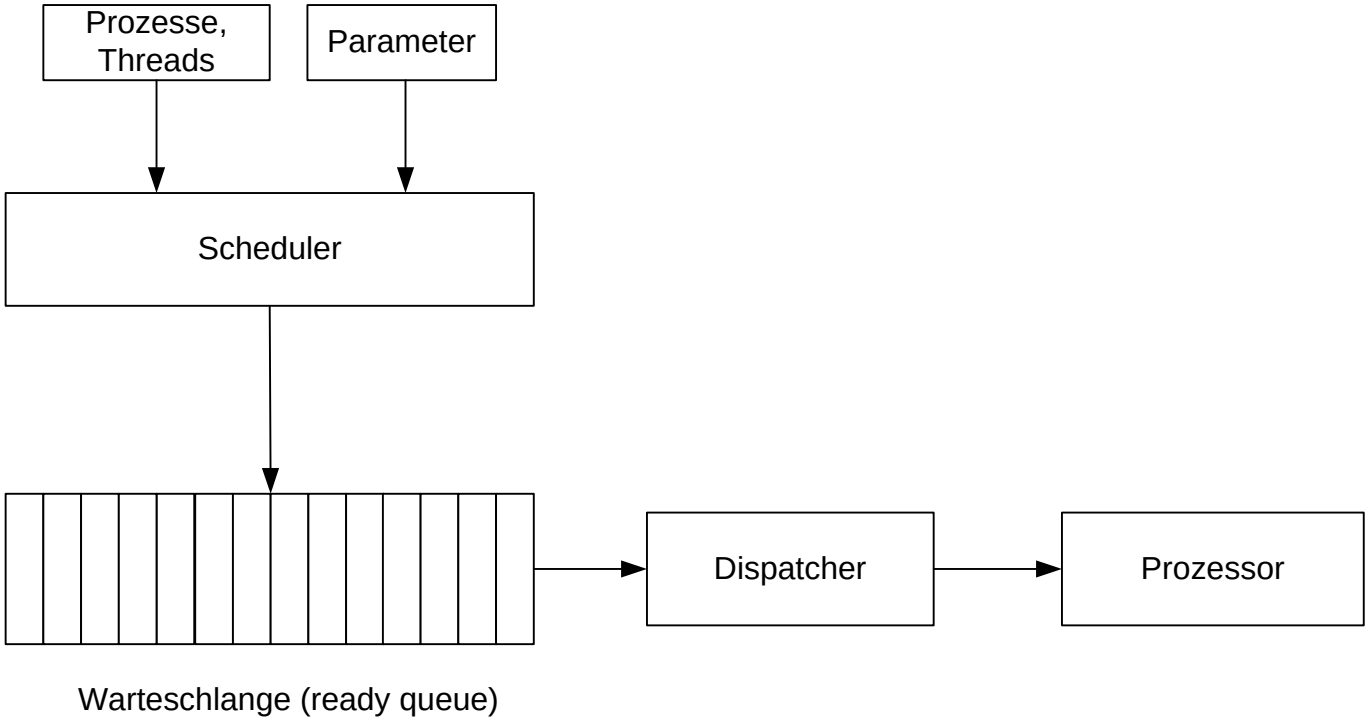
Fridolin Hofmann: Betriebssysteme - Grundkonzepte und Modellvorstellungen, 1991

- 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

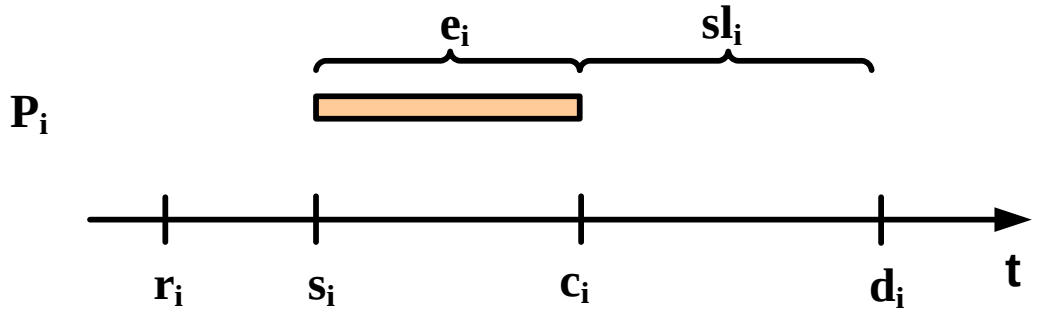


Scheduler und Dispatcher

- **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.
  7. sl_i: **Slack**: Deadline-(aktuelle Zeit + verbleibende Berechnungszeit)



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)
    - Periodisch: Prozesse sind regelmäßig mit fixer Frequenz startbereit
    - Nicht-Periodisch: Prozessbereitzeiten können mit gewissen Schranken vorhergesagt werden
    - Sporadisch: Es können keine/kaum Aussagen darüber getroffen werden, wann und wie oft Prozesse startbereit sind
  - Gemeinsame Nutzung von Ressourcen (**shared resources**)
  - 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 → 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. für alle Prozesse P_i gilt $c_i<d_i$ unter Berücksichtigung von Kausalzusammenhängen (Synchronisation, Vorranggraphen, Präzedenzsystemen)
  - Determinismus des Verfahrens
  - 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. $r_i + e_i < d_i$, 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 $r_x > t_i \wedge d_x < t_j$ kleiner als der Zeitraum sein (bei Multicoreprozessoren mit n gleichen Prozessoren kleiner dem n-fachen des Zeitraums).
- Durch weitere Rahmenbedingungen (z.B. Abhängigkeiten der einzelnen Prozesse) können weitere Bedingungen hinzukommen.

Schedulingverfahren

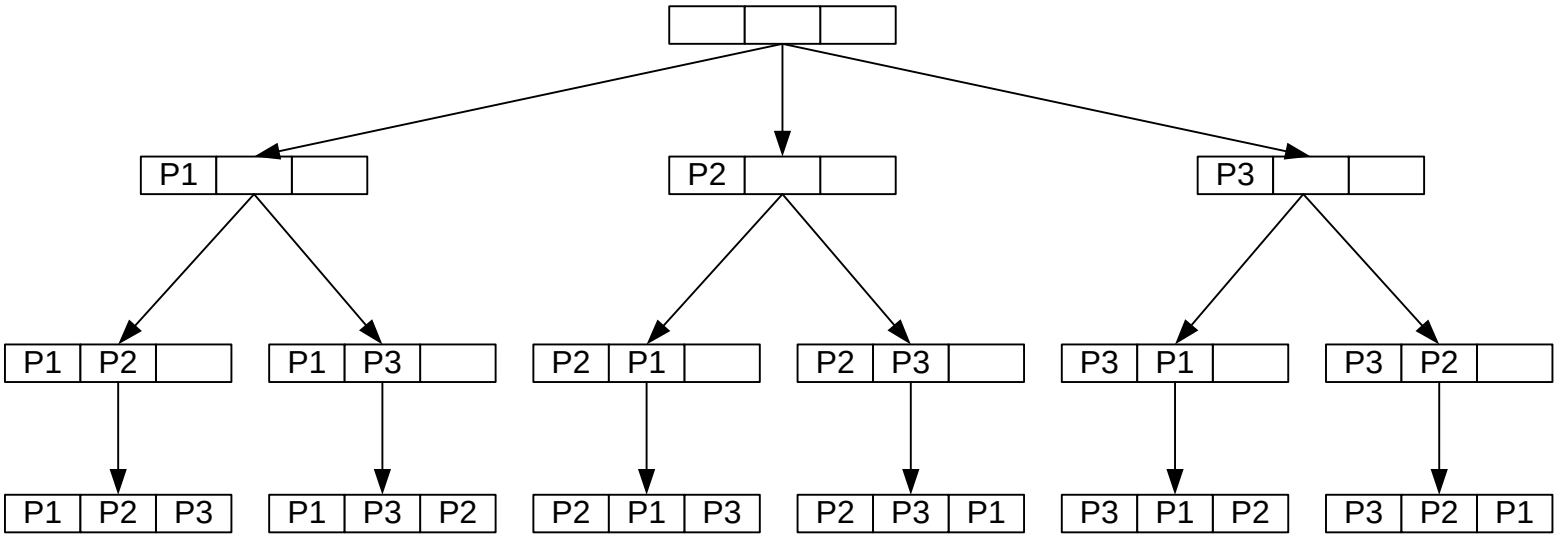
- Planen aperiodischer Prozesse
  - Planen durch Suchen
  - Planen nach Fristen
  - Planen nach Spielräumen
  - Prioritätenbasiertes Scheduling
- Planen periodischer Prozesse
  - Planen nach Fristen
  - Planen nach Raten
- Planen abhängiger Prozesse

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 reduziert 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**.

Bewertung

- Vorteile:
  - Sehr geringer Overhead zur Laufzeit
  - Sehr gut zu analysieren wegen Determinismus
  - Einfache Lösung um Race Conditions zu vermeiden
- Nachteil:
  - Alle Schedulingparameter (Startzeiten, Ausführungszeiten, Fristen) müssen bekannt sein oder abgeschätzt werden
  - Falls Schätzung erfolgt: häufig ineffizient wegen schlechter Ressourcenausnutzung
  - Komplexität der Berechnung
- Anwendung in der Praxis:
  - Einsatz eher selten und dann bei hochkritischen Systemen als zeitgesteuertes Scheduling
  - Zeitvergabe durch den Entwickler
  - Häufig in Kombination mit anderen Verfahren (z.B. prioritätenbasiertes Scheduling): Eine Teilmenge der Prozesse bekommt einen gemeinsamen Zeitabschnitt zugewiesen, innerhalb des Abschnittes erfolgt das Scheduling nach anderen Kriterien

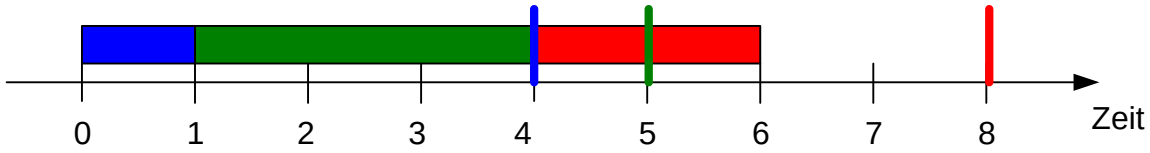
Scheduling-Strategien (online, nicht-präemptiv) für Einprozessorsysteme

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.

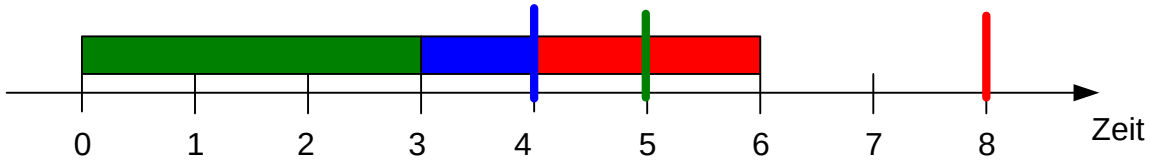
- Vorteil und Nachteile:
  - LST erkennt Fristverletzungen früher als EDF.
  - Für LST müssen die Ausführungszeiten der Prozesse bekannt sein.

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;



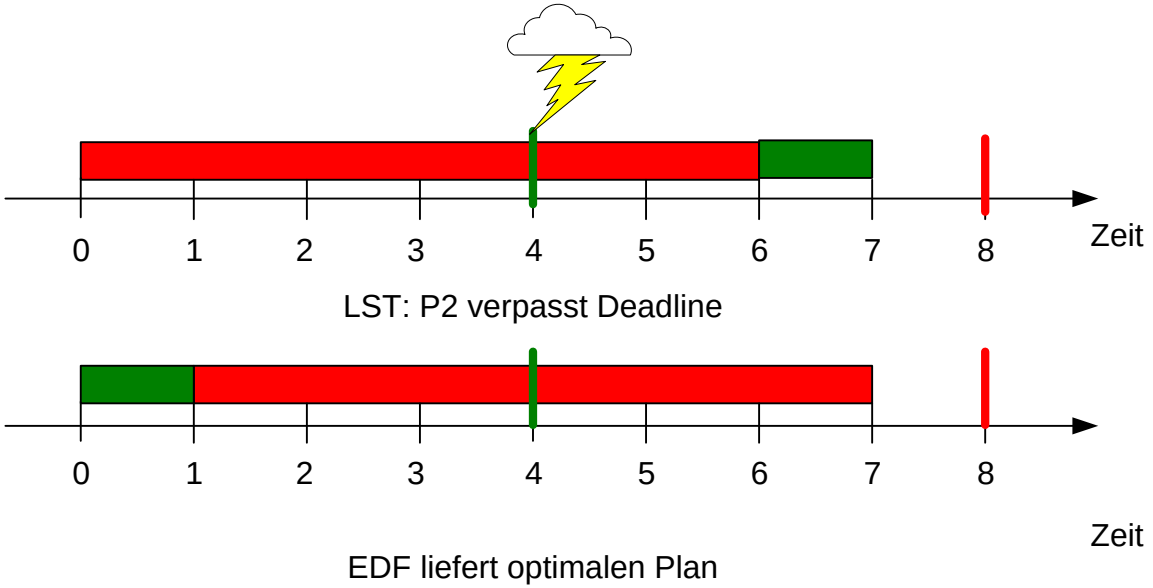
Earliest Deadline First



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. $d_i <= d_j$ 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 mit $Plan_x(0)=Plan_x$.

Fortsetzung des Beweises

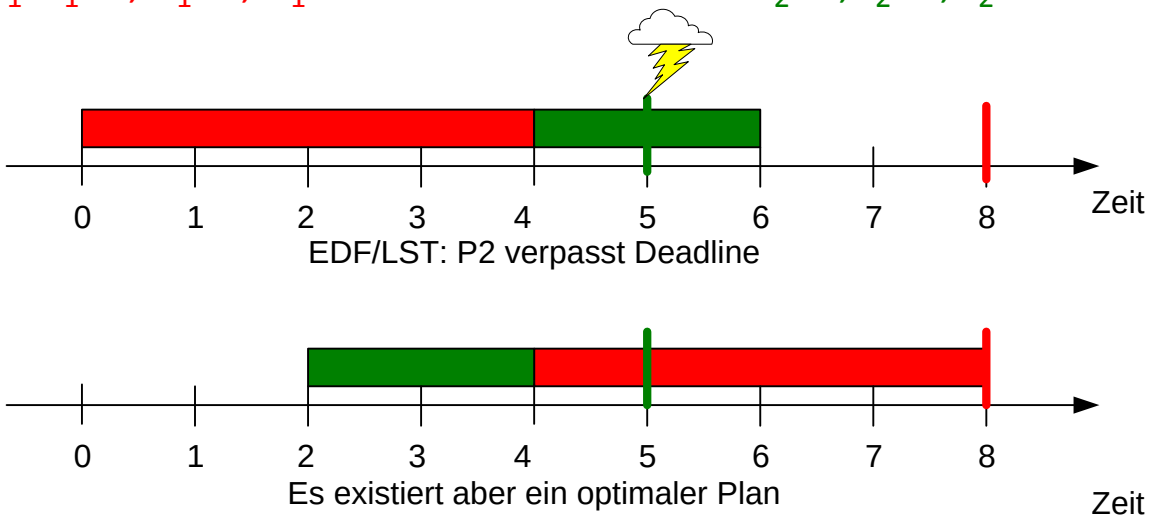
- Wir betrachten ein beliebig kleines Zeitinterval Δ_t.
- Zum Zeitpunkt t gilt:
  $i=P(Plan_{EDF},t)$
  $j=P(Plan_x,t)$
- Nur der Fall j>i ist interessant. Es gilt:
  - $d_i \leq d_j$
  - $t+\Delta_t \leq d_i$ (ansonsten wäre der $Plan_x$ nicht zulässig)
  - Da die Pläne bis zum Zeitpunkt t identisch sind und P_i im $Plan_{EDF}$ zum Zeitpunkt t ausgeführt wird, kann der Prozess P_i im $Plan_x$ noch nicht beendet sein.

    →EXISTS $t'>t+\Delta_t$: $(i=P(Plan_x,t')=P(Plan_x,t'+\Delta_t) \wedge t'+\Delta_t \leq d_i \leq d_j$

    → 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 → 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

  → 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)

Scheduling

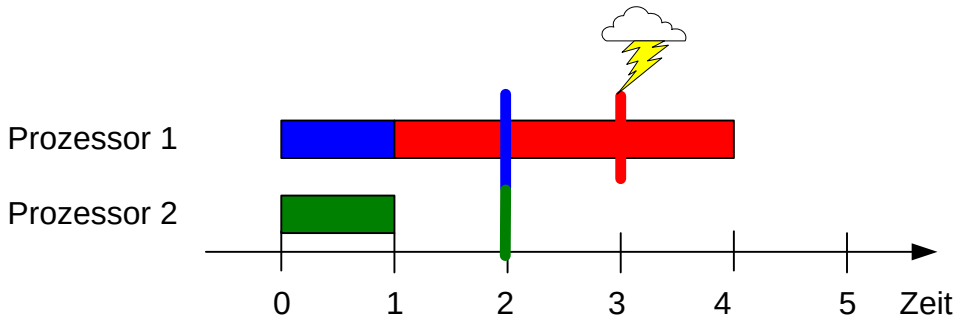
Scheduling-Verfahren für Mehrprozessor-Systeme

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 und präemptive Strategie gewählt wird
  - 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.

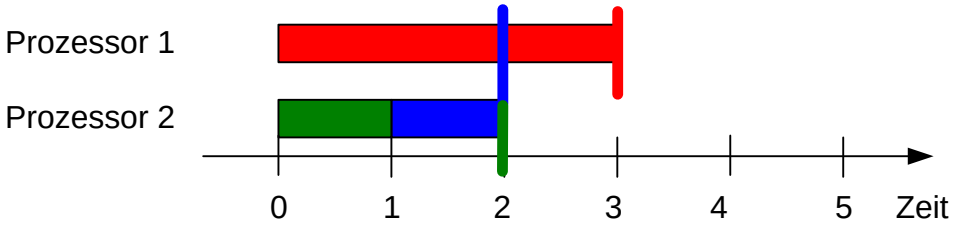
Beispiel: Versagen von EDF

- 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;

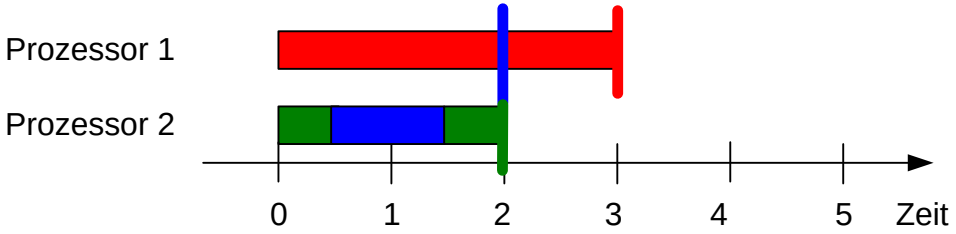


EDF-Verfahren: Deadline d1 wird verpasst

Beispiel: Optimaler Plan und LST-Verfahren



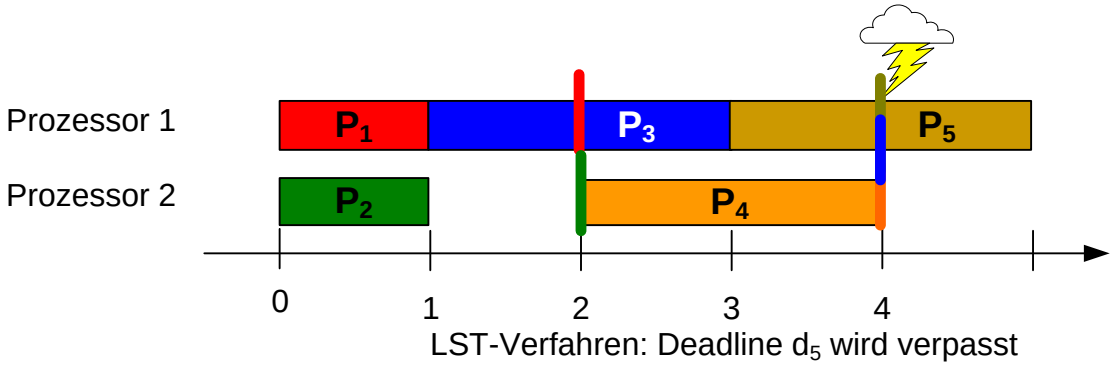
Optimaler Plan



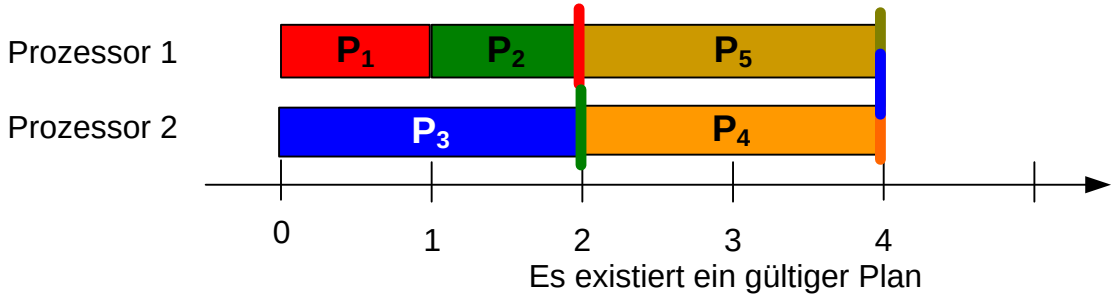
LST-Verfahren mit $\Delta_t = 0.5$

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;

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

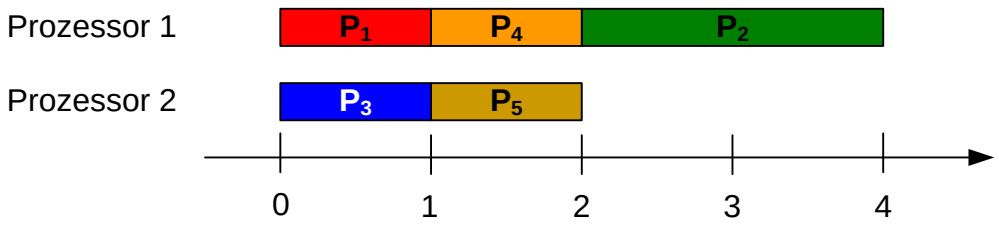
  ⟶ Es gibt je nach Strategie zwei Fälle zu betrachten: P2 oder P3 wird zunächst auf CPU2 ausgeführt.

1. Fall

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

  ⟶ Es gibt drei Prozesse ohne Spielraum, aber nur zwei Prozessoren.

- Aber es gibt einen gültigen Ausführungsplan:

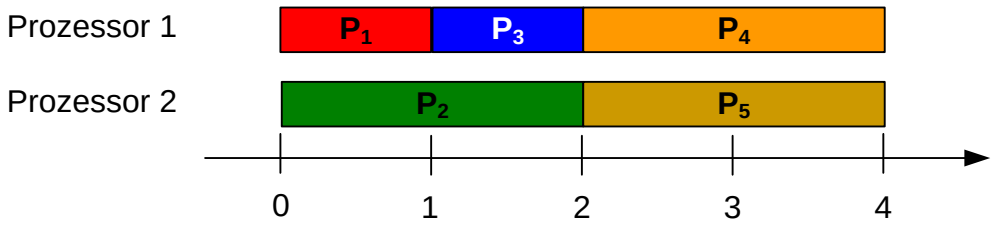


2. Fall

- 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.

  → Anstelle der zum Zeitpunkt 2 noch notwendigen 5 Ausführungseinheiten sind nur 4 vorhanden.

- Aber es gibt einen gültigen Ausführungsplan:



Administratives / Fragen zur letzten Vorlesung

- Administratives:
  - Prüfungsanmeldung ist jetzt freigeschaltet. Die Anmeldung ist bis zum 15.1.2014 möglich.
  - Bitte rechtzeitig anmelden, damit wir entsprechend der Teilnehmerzahlen planen können.
- Fragen zur Vorlesung:
  - Diskussion zum Thema Optimalität:
    - Optimalität im Allgemeinen: wichtig ist vor allem, dass der Entwickler eine Abschätzung machen kann, ob es zu Deadlineverletzungen kommen kann (100% Last ist aus Gründen der Robustheit sowieso zu vermeiden)
    - Optimalität in Multicoresystemen: in Echtzeitsystemen wird im überwiegenden Fall ohnehin eine Zuweisung der Prozesse auch Cores vorgenommen. Insofern reduziert sich das Scheduling-Problem auf die Single-Core Variante (→ ausgenommen der Problematik der Abhängigkeit von Prozessen, sowie der Worst Case Execution Time Analyse, siehe später)

Bewertung

- Die Strategien EDF und LST werden in der Praxis selten angewandt.
- Gründe:
  - Bereitzeiten sind nur bei zyklischen Prozessen oder Terminprozessen bekannt.
  - Häufig liegen nicht einmal konkrete Fristen vor → diese werden erst bei Tests ermittelt
  - Die Abschätzung der für LST benötigten Laufzeit ist sehr schwierig (siehe Exkurs).
  - Die Implementierung erfordert ein Sortieren nach Fristen bzw. Slack Time → zeitaufwändig und generell werden dynamische Datenstrukturen gerne vermieden.
  - 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 (→ Fehlerquelle).**
- 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. (Vorteil: robuster gegenüber Laufzeitüberschreitungen, Nachteil: mehr Kontextwechsel)

Klausur SS 07 – Szenario (20 Punkte = 20 min) – leicht modifiziert

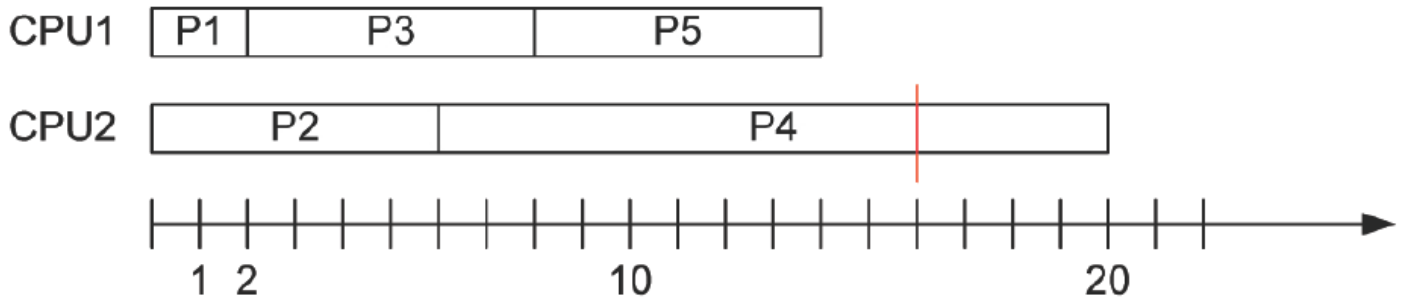
a) Welches Schedulingverfahren wurde verwendet? Welche Änderungen würden sich ergeben, wenn das Verfahren präemptiv wäre?

b) Welche optimalen Schedulingverfahren unter welchen Voraussetzungen existieren für Mehrprozessorsysteme?

c) Zeichnen Sie unter Zuhilfenahme eines optimalen Schedulingverfahrens einen korrekten Ausführungsplan.

d) In der Praxis werden diese Schedulingverfahren nicht angewandt. Was spricht dagegen und welcher Ansatz wird stattdessen gewählt?

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;



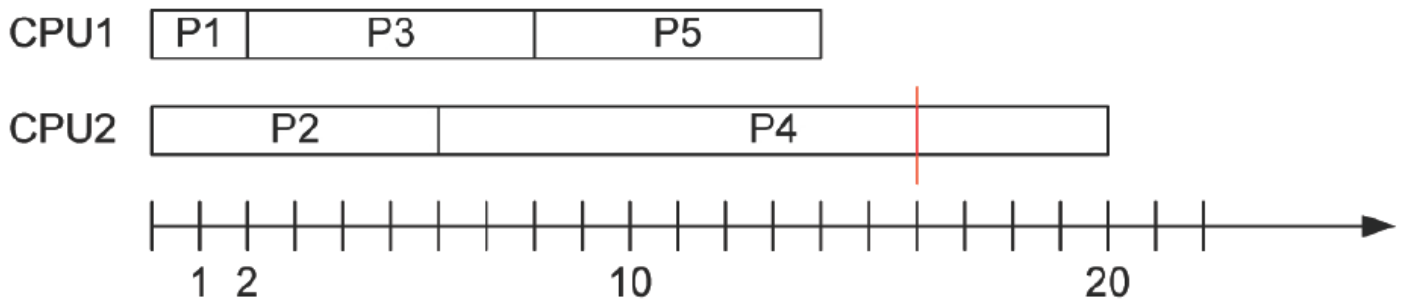
Klausur SS 07 – Antworten

a) EDF, es ergeben sich keine Änderungen, wenn das Verfahren präemptiv ausgeführt wird

b) Präemptives Planen (wenn alle Prozesse inkl. Startzeit, Laufzeit und Deadline vorab bekannt sind), präemptives Least Slack Time (gleiche Voraussetzungen wie planen, aber zusätzlich müssen alle Prozesse die gleiche Startzeit haben)

c) CPU 1: P1-> P4; CPU 2: P2->P3->P5

d) Es werden prioritätenbasierte Verfahren angewandt, da diese sehr leicht umzusetzen sind und zur Laufzeit neben der Priorität keine weiteren Informationen verfügbar sein müssen



Scheduling

Zeitplanen periodischer Prozesse

Zeitplanung periodischer Prozesse

- Annahmen für (präemptives) Scheduling periodischer Prozesse
  - 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**: siehe Folien bei Einführung von EDF
- Leider wird aufgrund von diversen Vorurteilen EDF selten benutzt.
- Betriebssysteme unterstützen selten ein EDF-Scheduling
  → 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.

  ⟶ 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:

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

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

- Bei harmonischen Frequenzen liefert RM bis zu einer Auslastung von 100% gültige Schedules.
- Derzeit zumeist verwendetes Scheduling-Verfahren im Bereich von periodischen Prozessen.

Scheduling

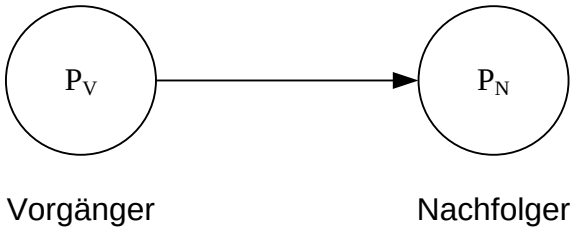
Planen abhängiger Prozesse

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 → 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.

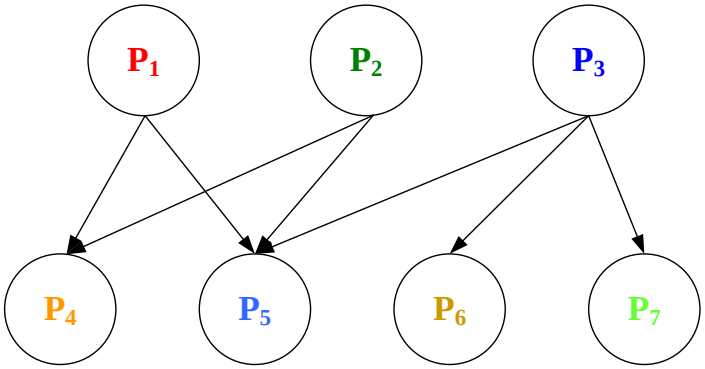
  $\rightarrow$ Ein Präzedenzsystem ist nur dann planbar, falls das zugehörige normalisierte Präzedenzsystem planbar ist.

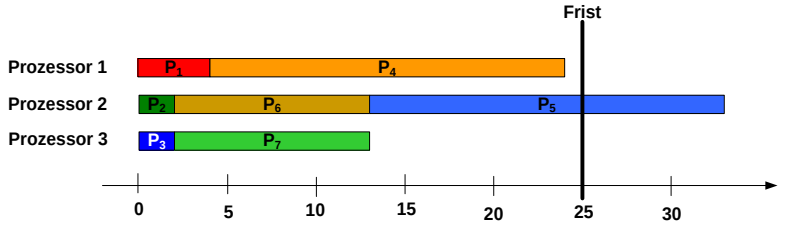
Anomalien bei nicht präemptiven Scheduling

- 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.

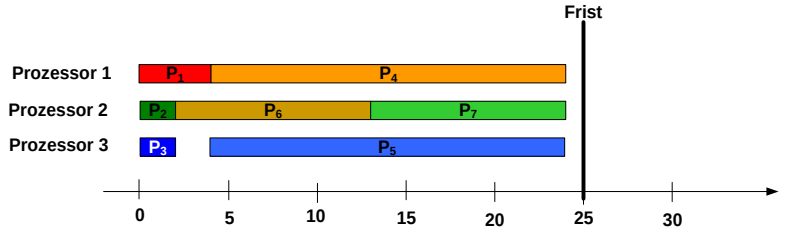
Beispiel: Verkürzung durch freiwilliges Warten

- 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 nicht-präemptives Scheduling (z.B. LST)

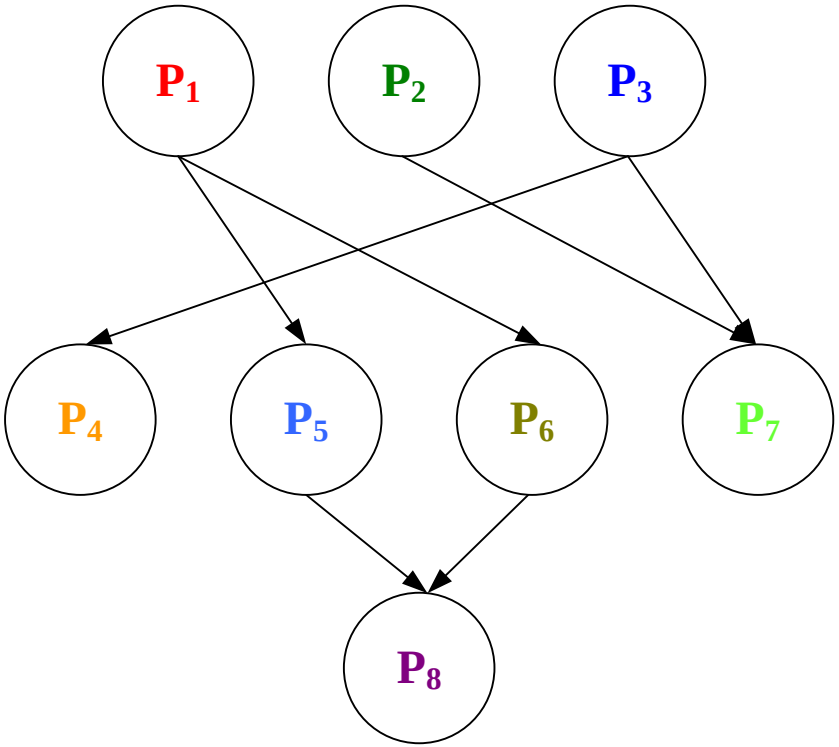


Optimaler nicht-präemptiver Plan

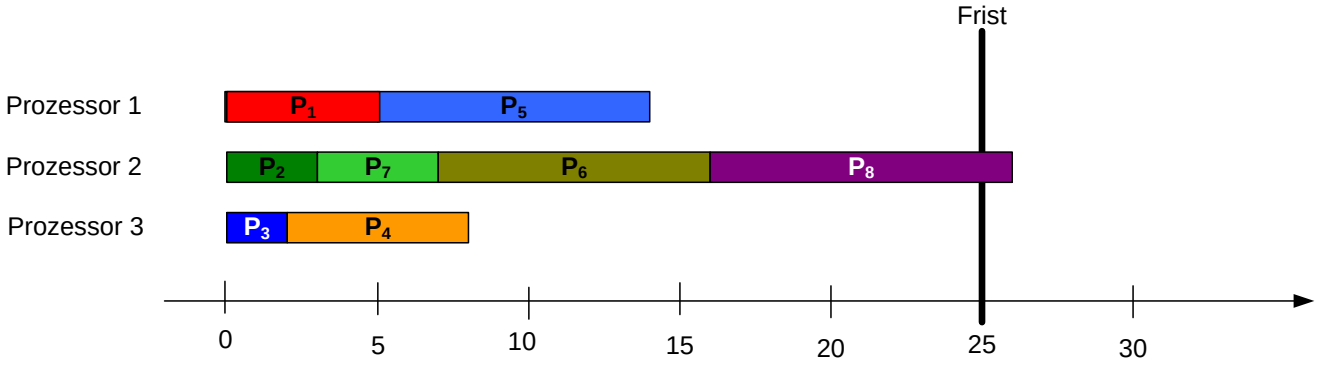
Beispiel: Laufzeitverlängerung durch zusätzlichen Prozessor II

- 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: $d_i=25$

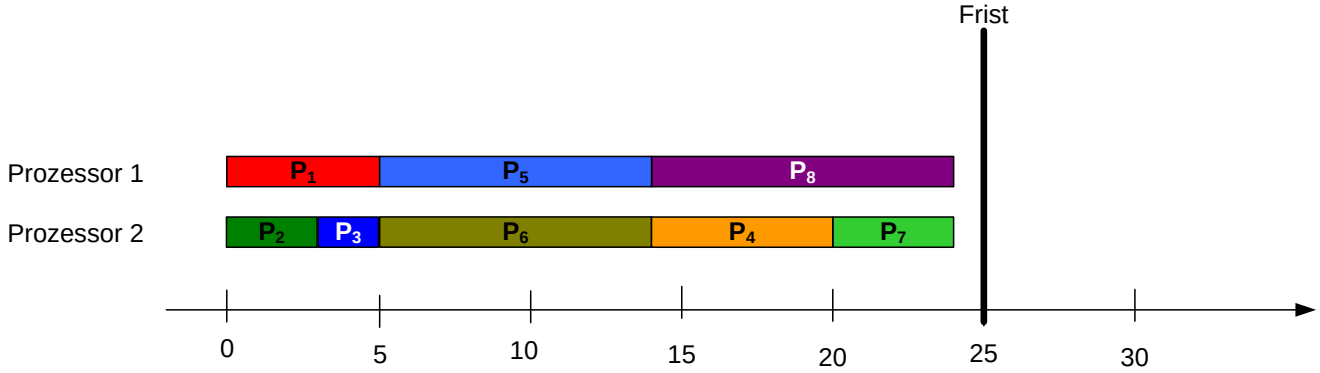
- Präzedenzgraph:



Beispiel: Laufzeitverlängerung durch zusätzlichen Prozessor II



Prioritätenbasiertes nicht-präemptives Scheduling (LST) auf 3 Prozessoren



Prioritätenbasiertes nicht-präemptives Scheduling (LST) auf 2 Prozessoren

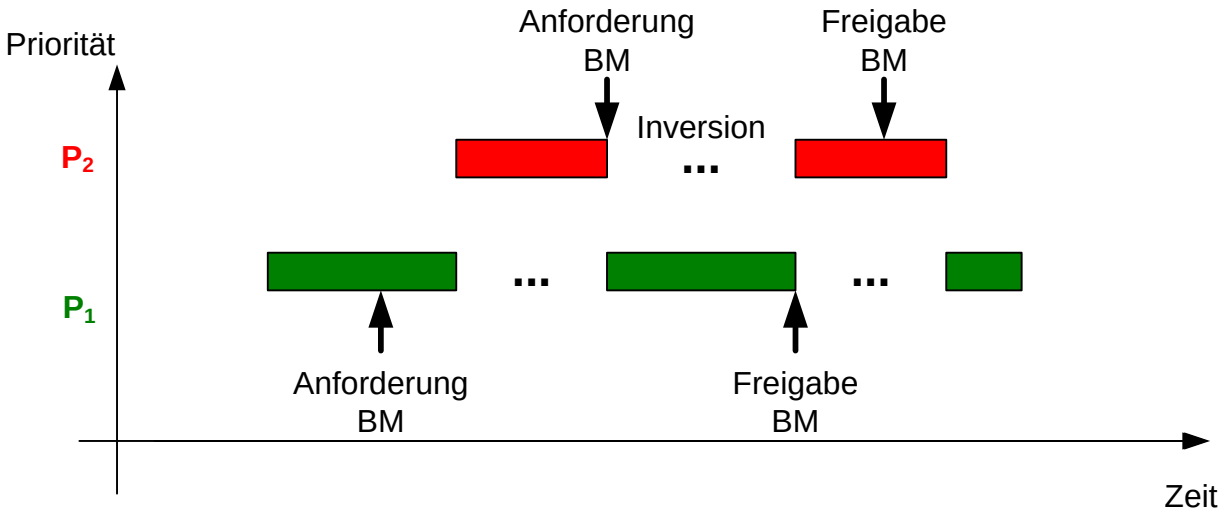
Scheduling

Problem: Prioritätsinversion

Motivation des Problems

- Beim Zugriff auf Betriebsmittel kann es dazu kommen, dass ein niederpriorer Prozess durch den Besitz eines Betriebsmittel einen hochprioren Prozess blockiert (**Begriff: Prioritätsinversion**)
- Beispiel aus dem Alltag:
  - Computer wird vom Bruder zum Spielen blockiert, obwohl man den Computer dringend für Hausaufgaben braucht

Begrenzte Inversion

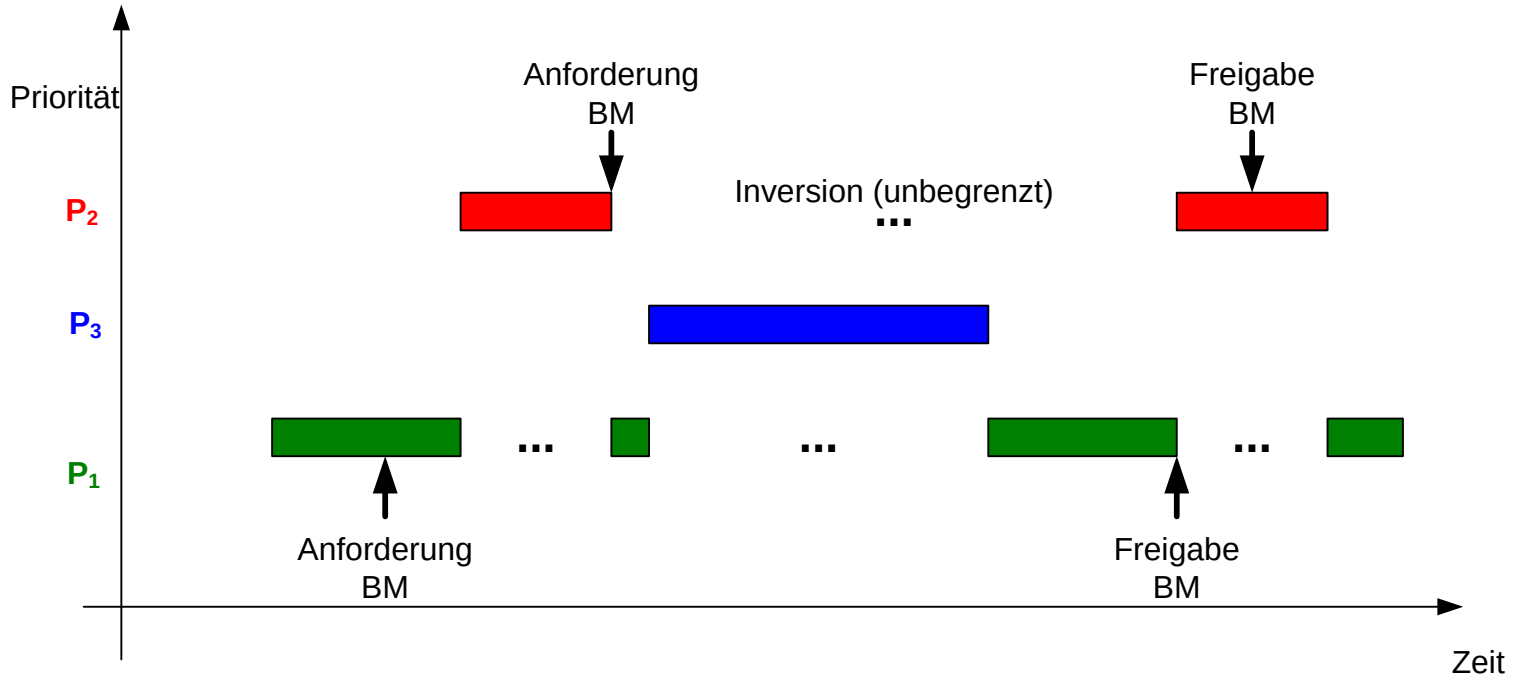


- Dieses Problem ist unvermeidbar, wenn beide Prozesse auf die Ressourcen zugreifen sollen, allerdings können die Auswirkungen durch folgende **Designregel** minimiert werden:
  - Niederpriore Prozesse müssen Ressourcenbelegung so kurz wie möglich halten
  - Die Blockierung durch niederpriore Prozesse muss bei der Zeitabschätzung berücksichtigt werden

Motivation des Problems - Fortsetzung

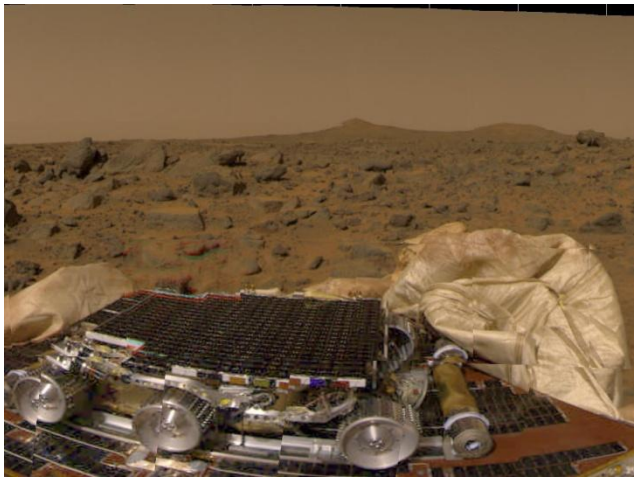
- Trotz Beachtung der vorgenannten Designregel kann es zu einer Verschärfung der Problematik kommen
- Fortsetzung des Beispiels: die Ressourcen können nicht einfach entzogen werden und der niederpriore Prozess wird von einem weiteren, mittelprioren Prozess für eine längere Zeit unterbrochen
  - Beispiel: Computer ist gesperrt, Bruder telefoniert mit seiner Freundin

Unbegrenzte Inversion



Reales Beispiel: Mars Pathfinder (Projektkosten: 175 Mio USD)

- **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 und griff ebenfalls regelmäßig auf den Informationsbus zu. 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 Semaphor 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.



Quelle: Nasa

Ansätze zur Lösung der Prioritätsinversion

- Es existieren verschiedene Ansätze, um das Problem der unbegrenzten Prioritätsinversion zu begrenzen:
  - Nicht-unterbrechbare kritische Bereiche
  - Prioritätsvererbung (priority inheritance)
  - Prioritätsobergrenzen (priority ceiling)
  - Unmittelbare Prioritätsobergrenzen (immediate priority ceiling)
- Anforderungen an Lösungen:
  - Leichte und anwendungsunabhängige Implementierung
  - Minimierung und gute Abschätzung der maximalen Verzögerung (insbesondere für hochpriore Prozesse)
  - Optional: Ausschluss von Verklemmungen

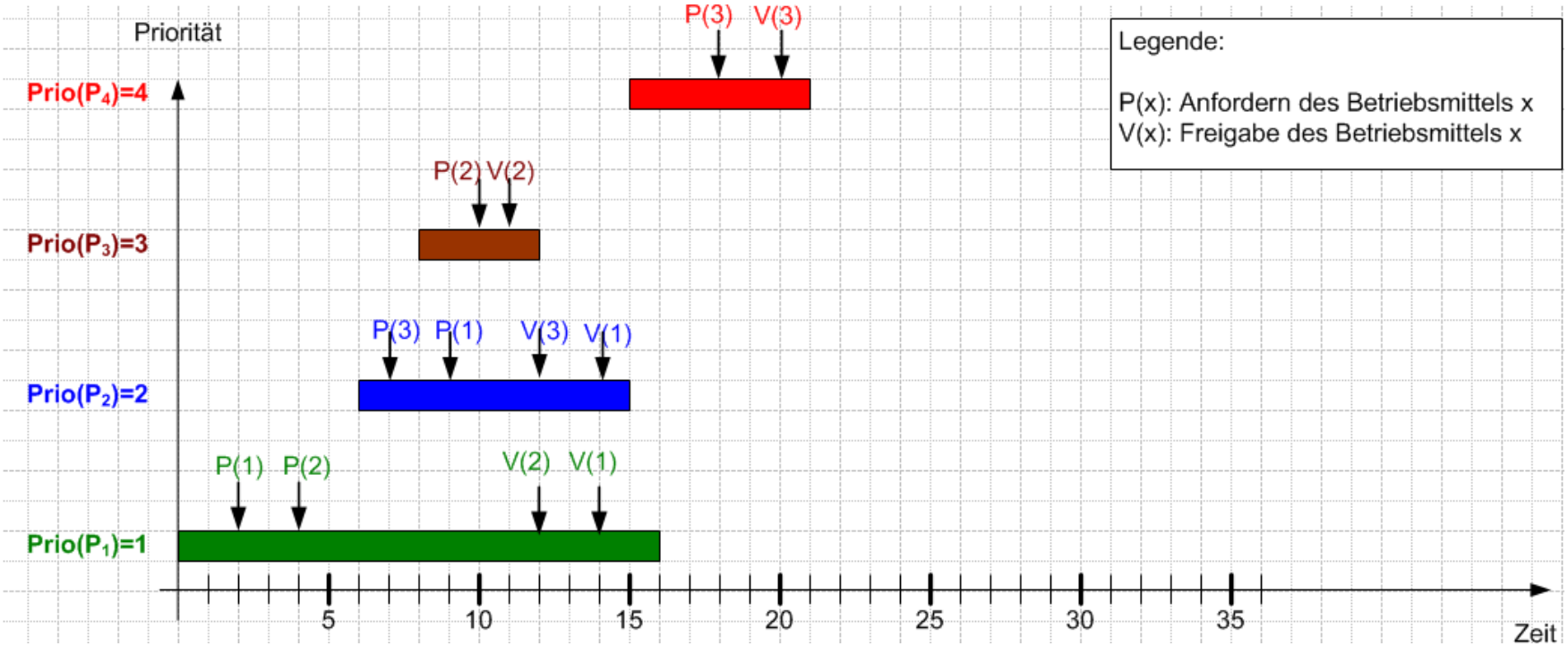
Nicht-unterbrechbare kritische Bereiche (Nonpreemptive critical section)

- Lösung: der Prozess wird für den kritischen Bereich nicht unterbrochen. Die unbegrenzte Prioritätsinversion wird somit verhindert.
- Bewertung der Lösung:
  - \+ einfach und anwendungsunabhängig zu implementieren
  - \+ leichte Abschätzung der maximalen Blockierung
  - \- Verzögert höherpriorisierte Prozesse auch dann, wenn diese das Betriebsmittel nicht benötigen

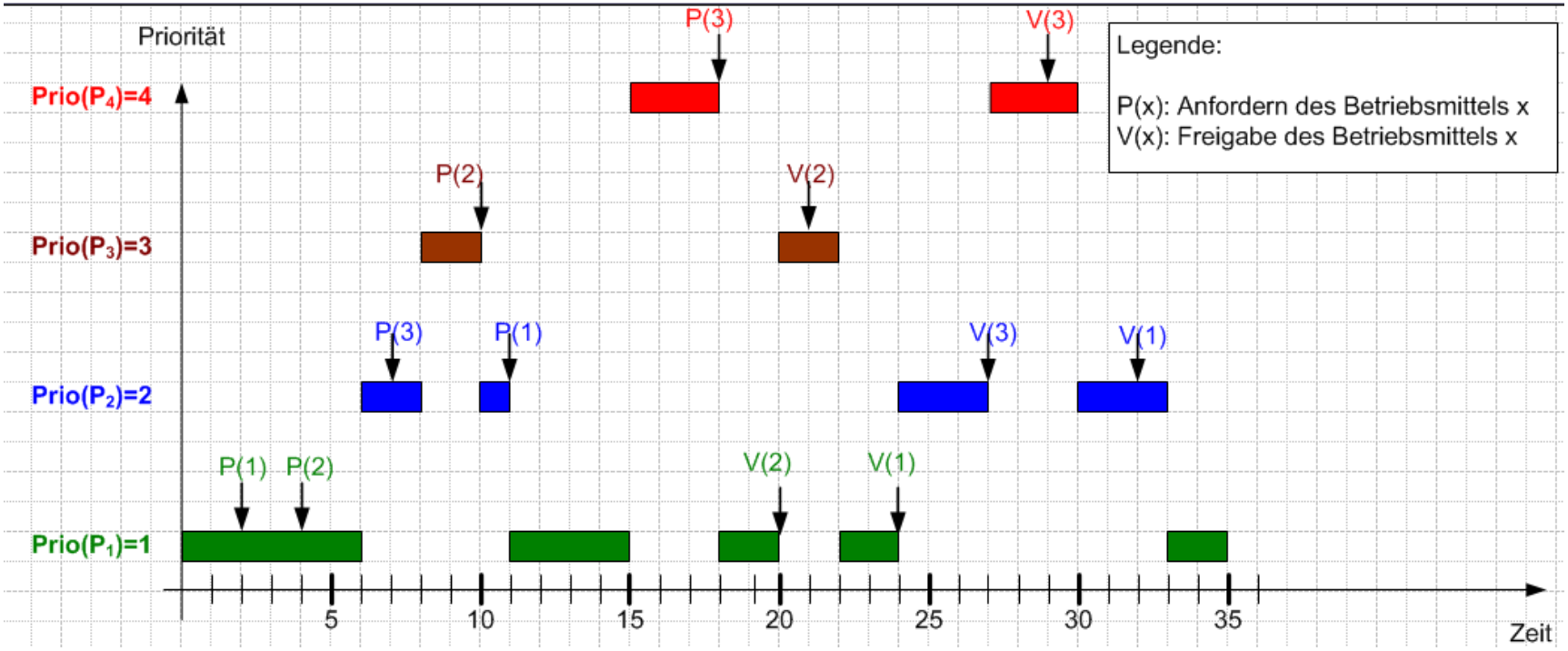
Prioritätsvererbung (priority inheritance)

- 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
  - \+ Einfache, anwendungsunabhängige Implementierung
  - \- Blockierungen werden hintereinander gereiht (Blockierungsketten)
  - \- Verklemmungen durch Programmierfehler werden nicht verhindert

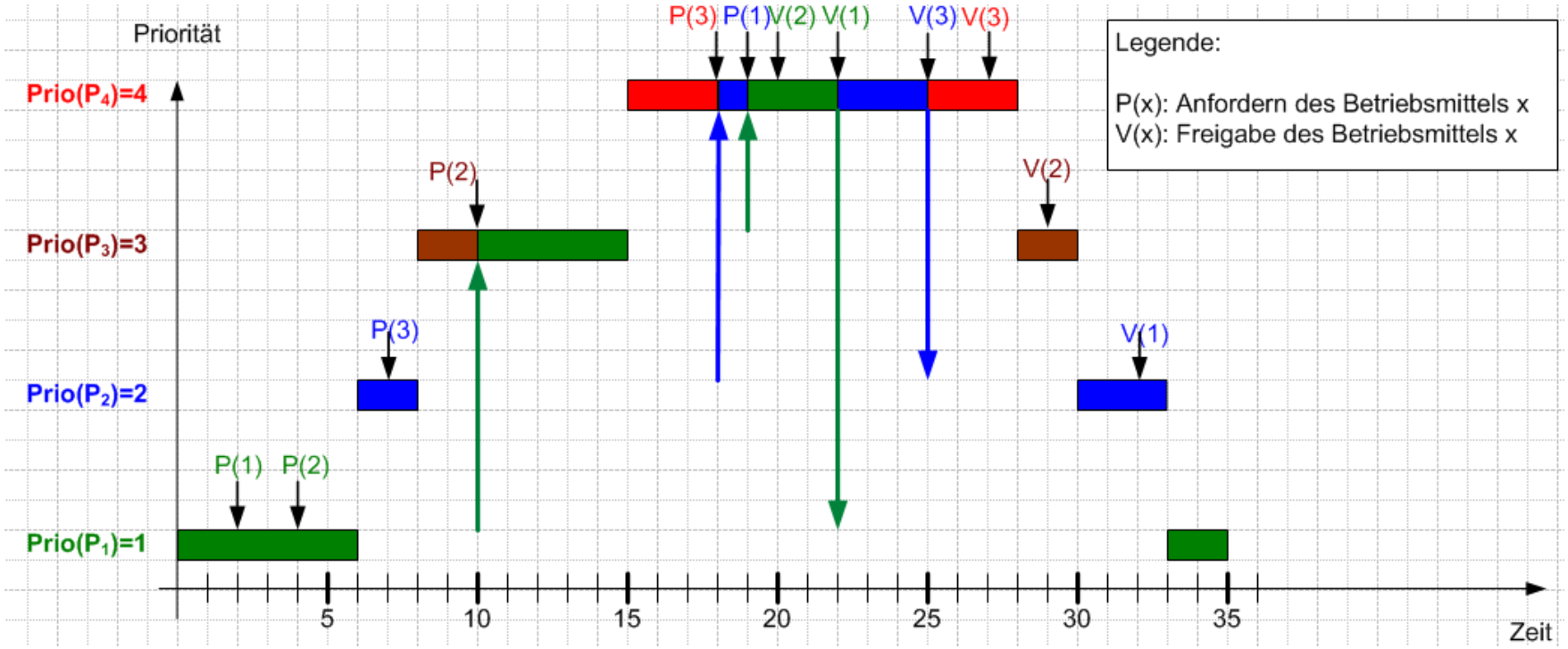
Laufendes Beispiel: Ausgangszustand



Laufendes Beispiel: Ohne Lösung gegen Prioritätsinversion



Beispiel: Prioritätsvererbung

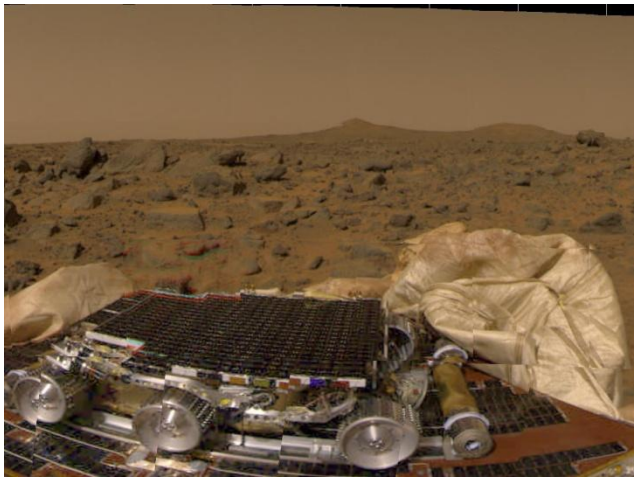


Dieses Beispiel ist natürlich sehr konstruiert, zeigt aber sehr schön die Problematik von Blockierungsketten auf.

➔ **Designregel**: die Vermischung von Betriebsmittelanforderungen sollte möglichst vermieden werden.

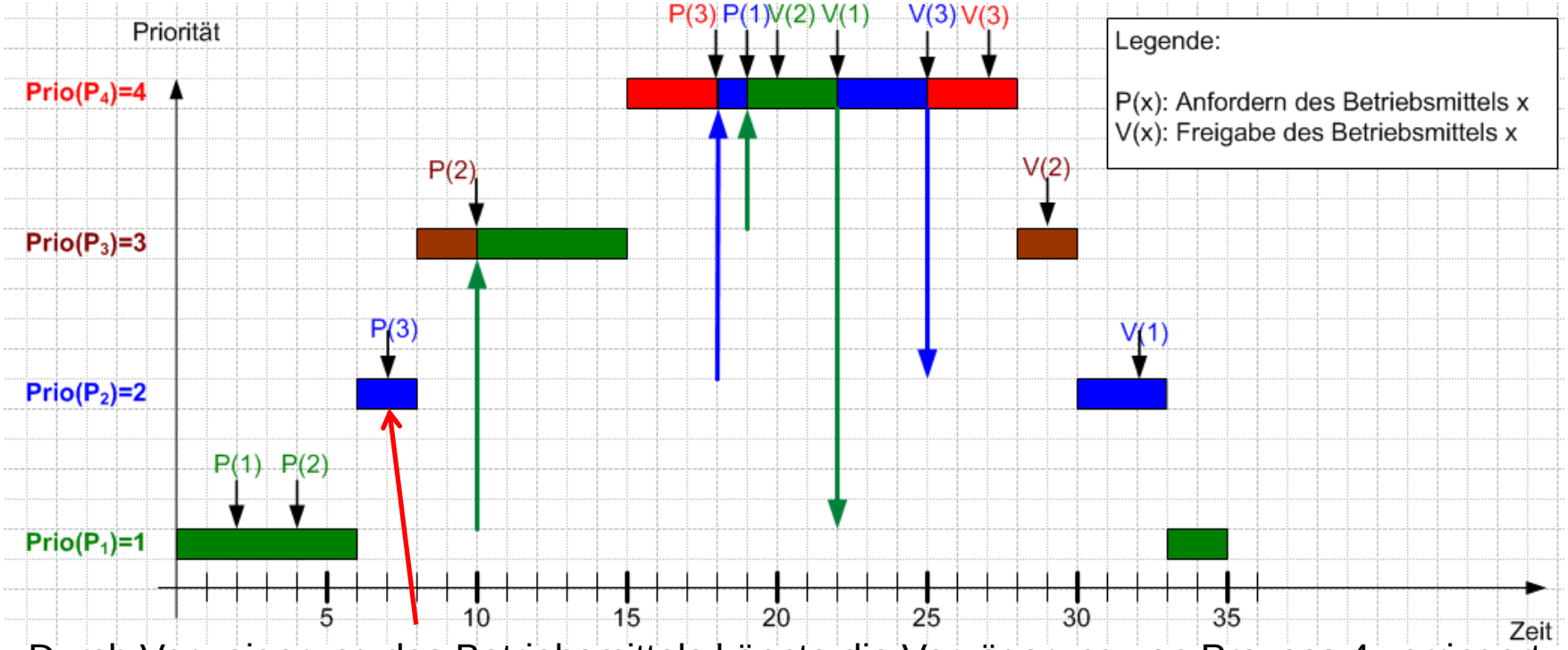
Lösung des Mars Pathfinder Problems

- Über eine Remoteverbindung konnte das Prioritätsvererbungsprotokoll nachträglich eingeschaltet werden



Quelle: Nasa

Analyse des vorherigen Beispiels



Durch Verweigerung des Betriebsmittels könnte die Verzögerung von Prozess 4 verringert werden.

Lösungsidee: Es wird die höchste Priorität der Prozesse berechnet (Prioritätsobergrenze), die von den momentan blockierten Betriebsmitteln betroffen ist. Neue Betriebsmittel werden nur dann Prozessen zugewiesen, wenn deren Priorität höher ist.

Prioritätsobergrenzen (priority ceiling)

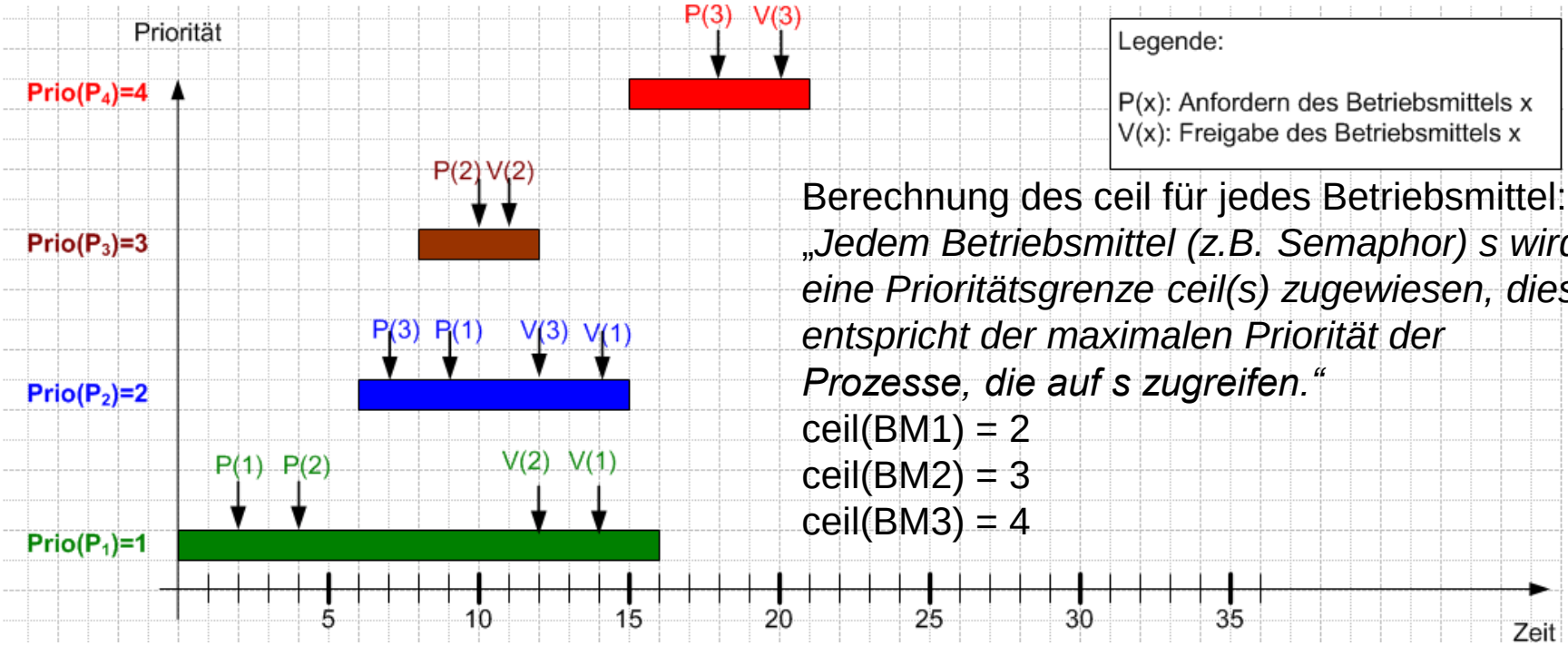
- Jedem Betriebsmittel (z.B. Semaphor) s wird eine Prioritätsgrenze ceil(s) zugewiesen, diese entspricht der maximalen Priorität der Prozesse, die auf s zugreifen.
  - Ein Prozess p darf ein BM nur blockieren, wenn er von keinem anderen Prozess, der andere BM besitzt, verzögert werden kann.
  - Die aktuelle Prioritätsgrenze für Prozess p ist
    aktceil (p)= max{ ceil(s) | s ϵ 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 blockierte Betriebsmittel mit der höchsten Prioritätsgrenze besitzt. Die Priorität dieses Prozesses wird auf aktprio gesetzt.
    - Beweisidee, dass es nur einen Prozess gibt: klar, da Prozesse nur dann auf Betriebsmittel zugreifen dürfen, wenn aktprio(p) > aktceil(p)
    - Beweisidee, dass die Priorität dieses Prozesses vorher kleiner ist als aktprio: die Priorität von Prozess p muss größer sein als vom Prozess, der das entsprechende Betriebsmittel hat, da sonst p gar nicht laufen würde
  - Gibt ein Prozess ein Betriebsmittel wieder frei, muss aktprio wieder neu berechnet werden

+ Blockierung nur für die Dauer eines kritischen Abschnitts

+ Verhindert Verklemmungen

- schwieriger zu realisieren, zusätzlicher Prozesszustand

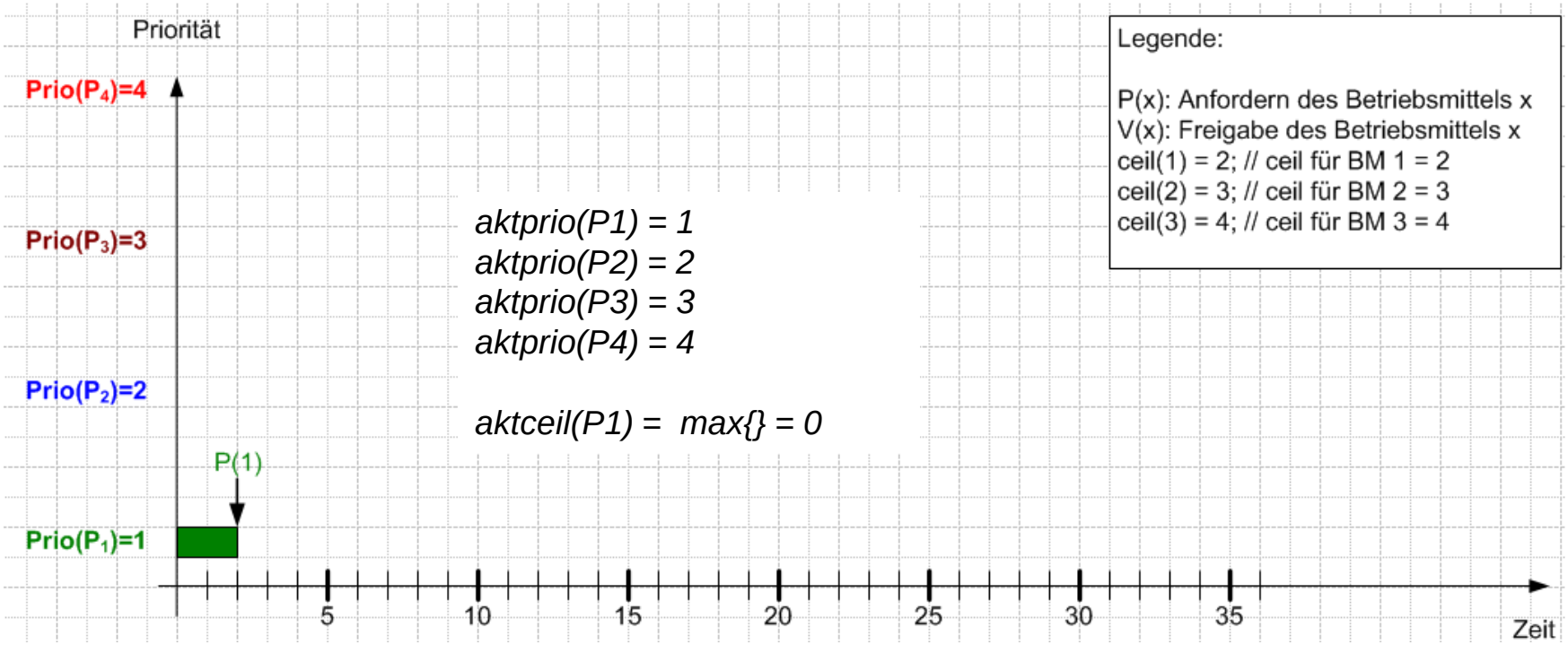
Beispiel: Prioritätsobergrenzen: Ausgangszustand



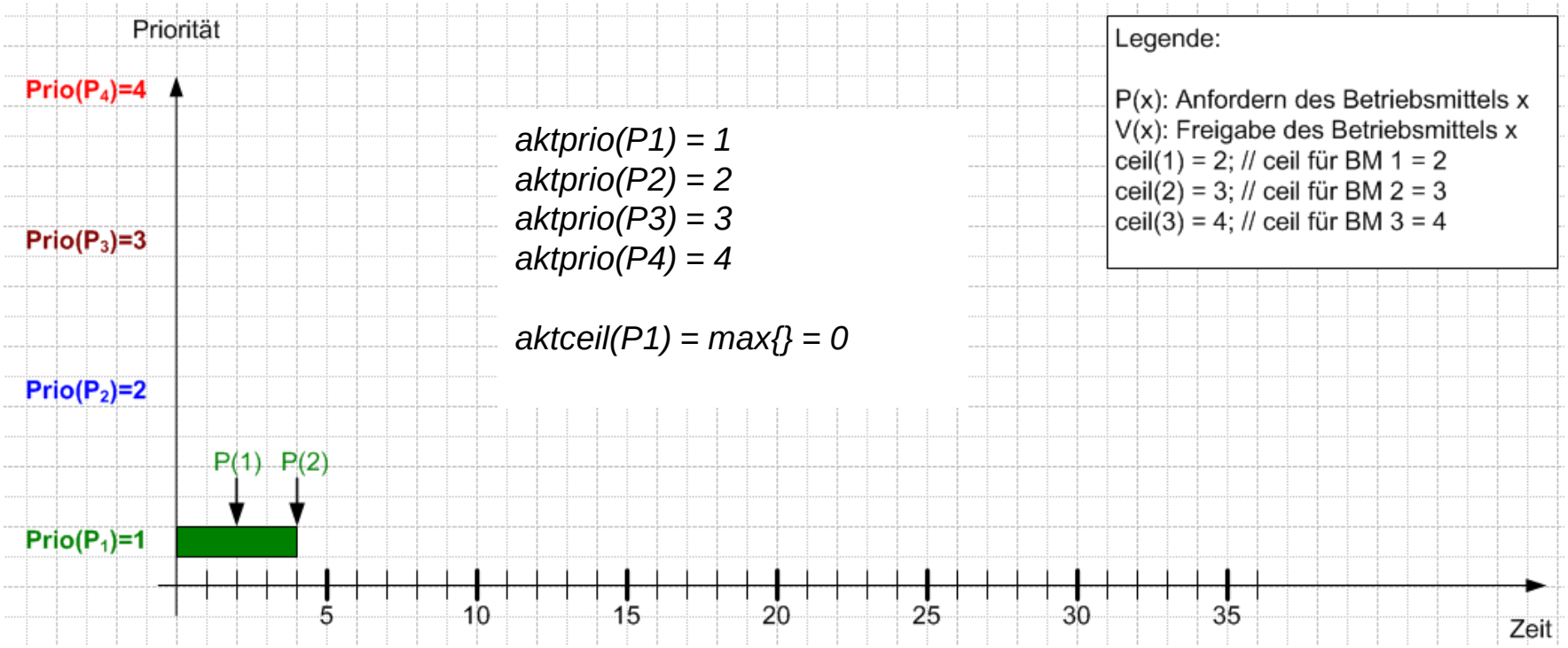
Berechnung des ceil für jedes Betriebsmittel:
„Jedem Betriebsmittel (z.B. Semaphor) s wird eine Prioritätsgrenze ceil(s) zugewiesen, diese entspricht der maximalen Priorität der Prozesse, die auf s zugreifen.“

ceil(BM1) = 2
ceil(BM2) = 3
ceil(BM3) = 4

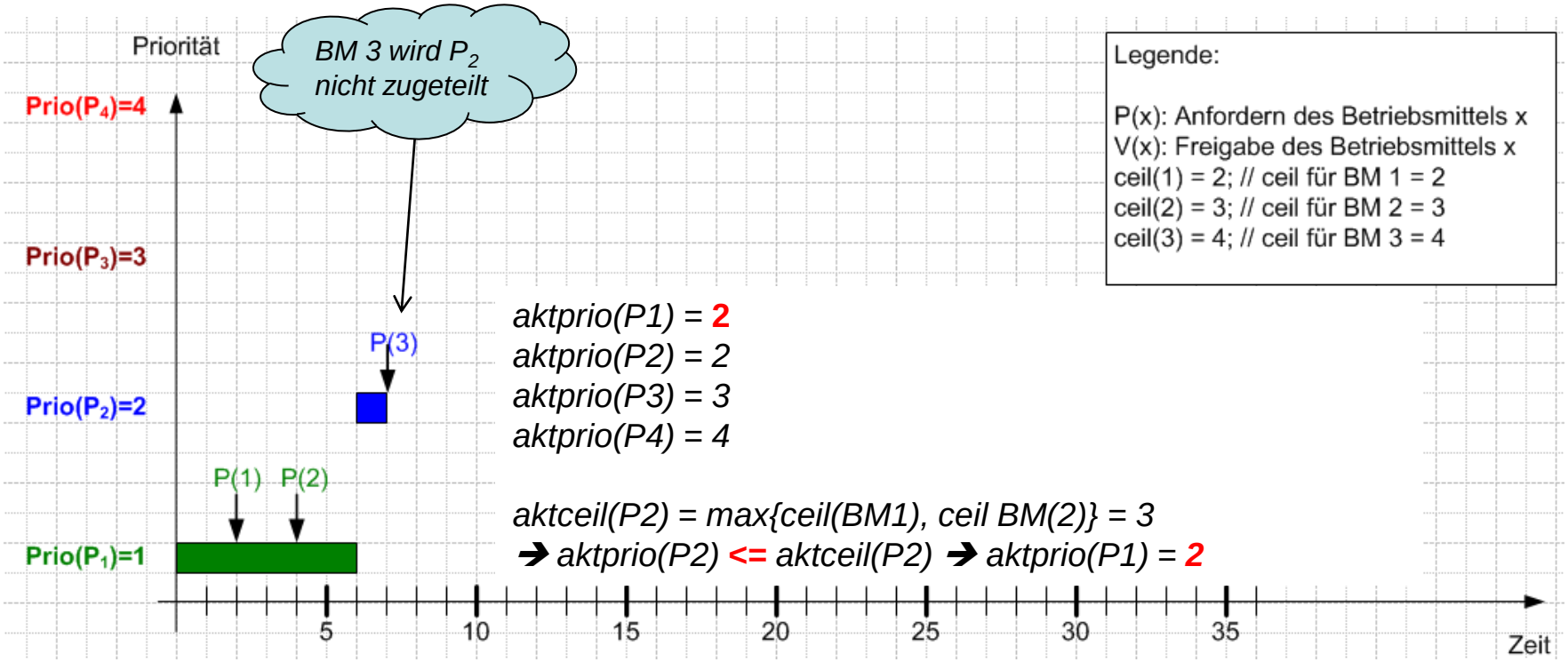
Beispiel: Prioritätsobergrenzen



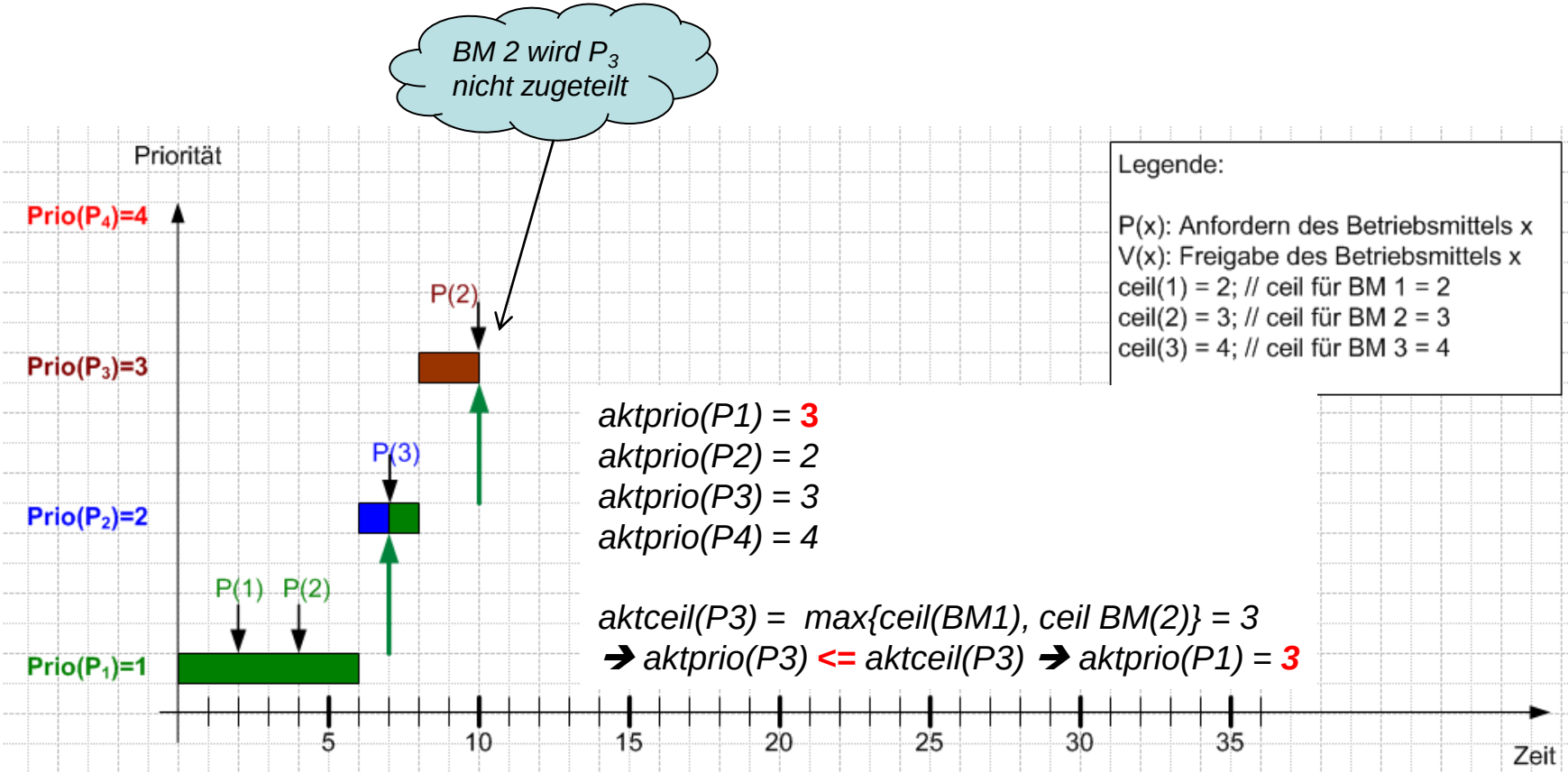
Beispiel: Prioritätsobergrenzen



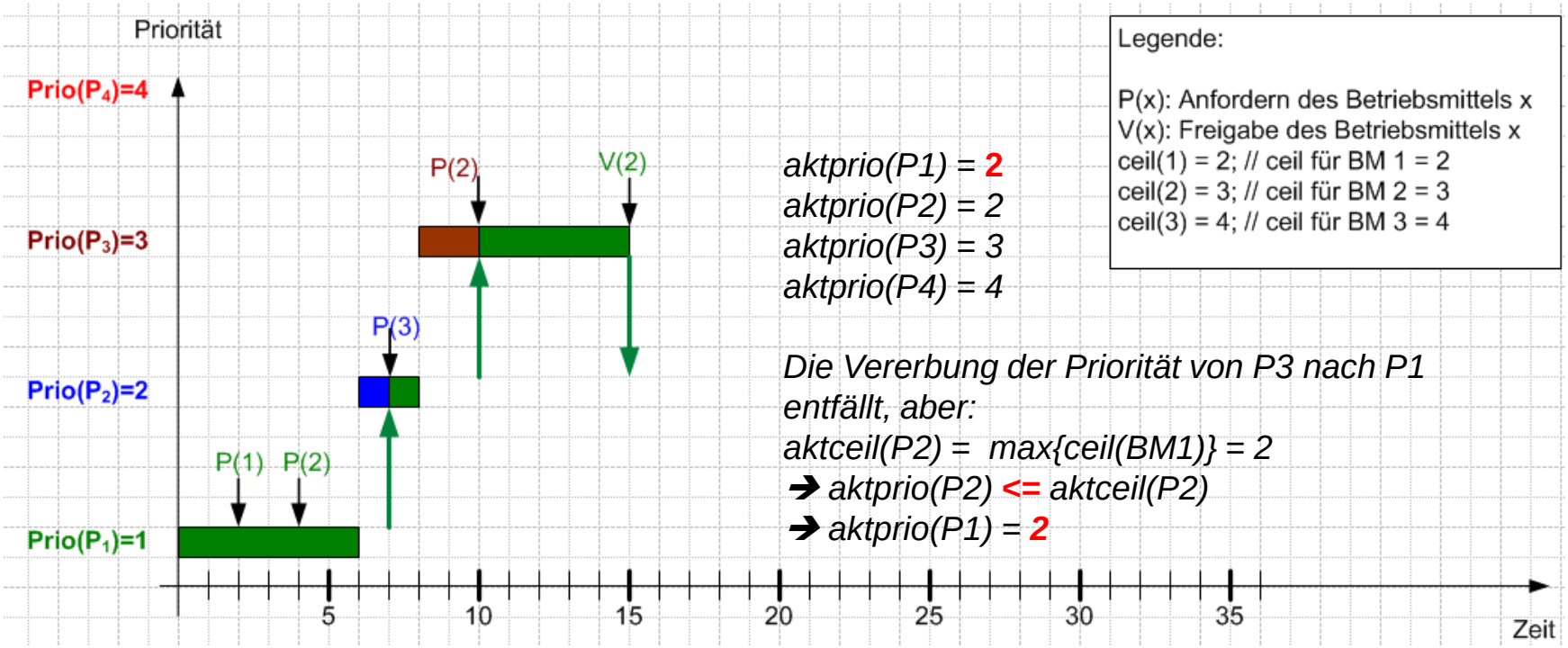
Beispiel: Prioritätsobergrenzen



Beispiel: Prioritätsobergrenzen



Beispiel: Prioritätsobergrenzen



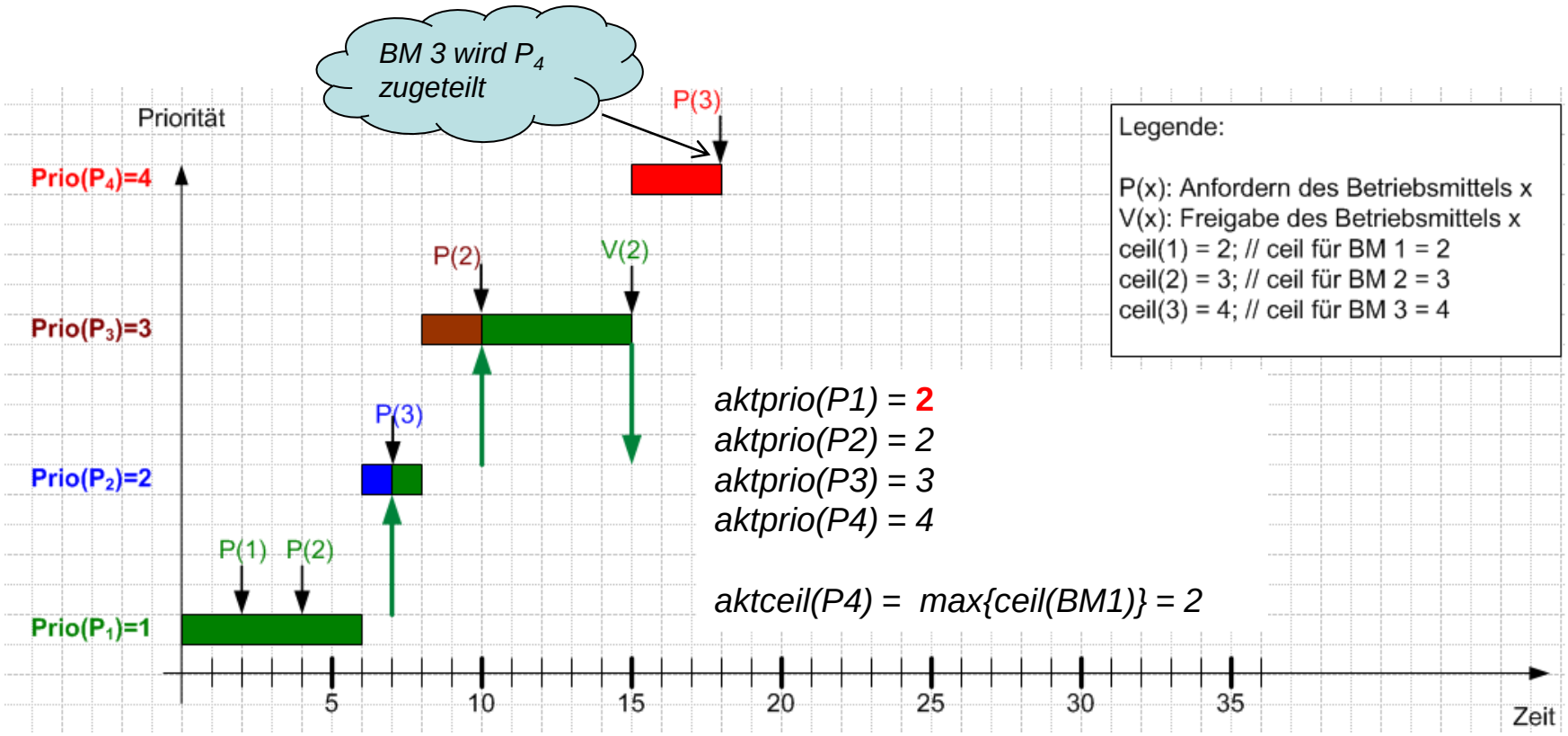
*aktprio(P1) = **2***
aktprio(P2) = 2
aktprio(P3) = 3
aktprio(P4) = 4

Legende:

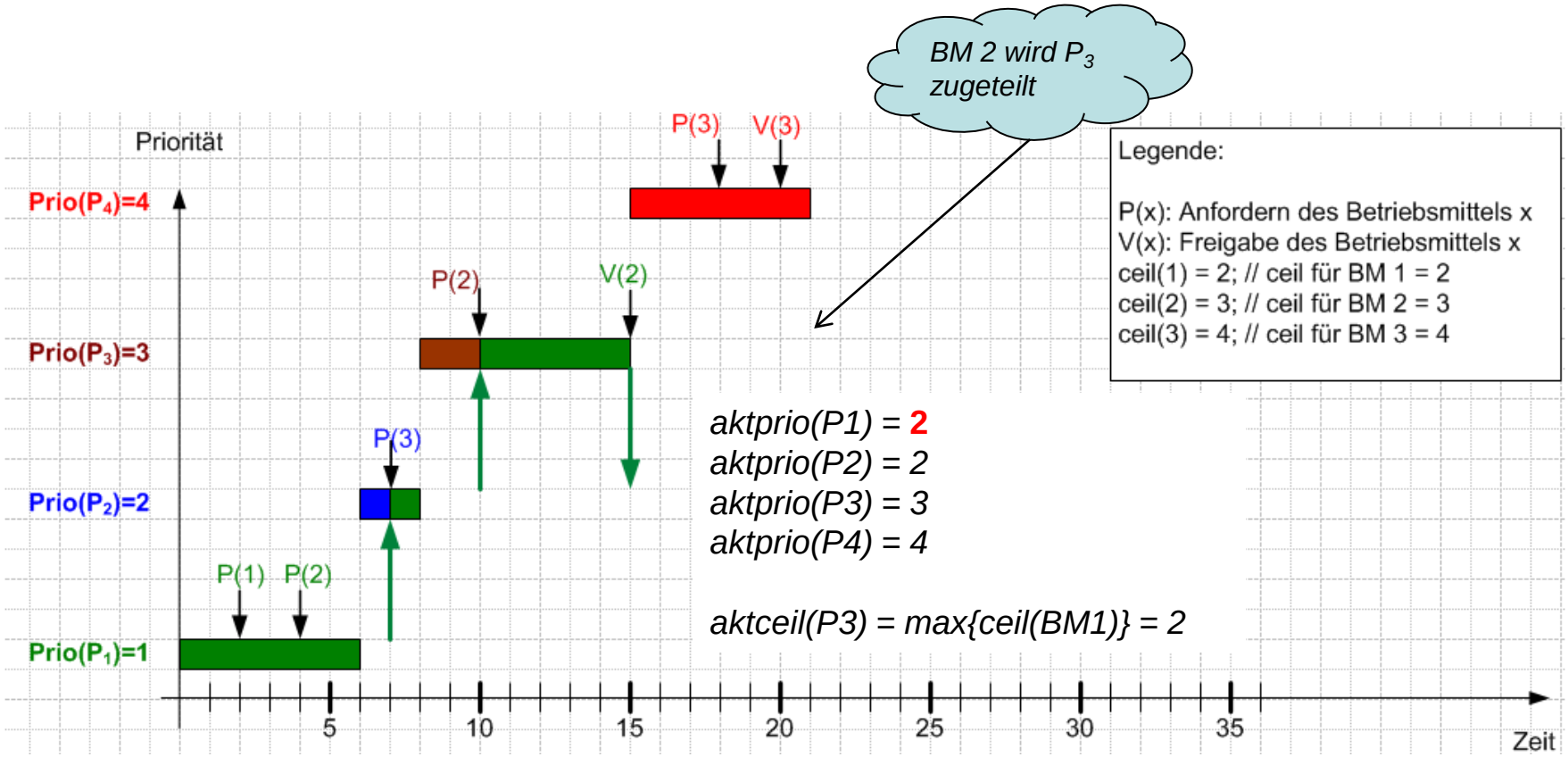
P(x): Anfordern des Betriebsmittels x
V(x): Freigabe des Betriebsmittels x
ceil(1) = 2; // ceil für BM 1 = 2
ceil(2) = 3; // ceil für BM 2 = 3
ceil(3) = 4; // ceil für BM 3 = 4

Die Vererbung der Priorität von P3 nach P1 entfällt, aber:
aktceil(P2) = max{ceil(BM1)} = 2
➔ *aktprio(P2) <= aktceil(P2)*
➔ *aktprio(P1) = **2***

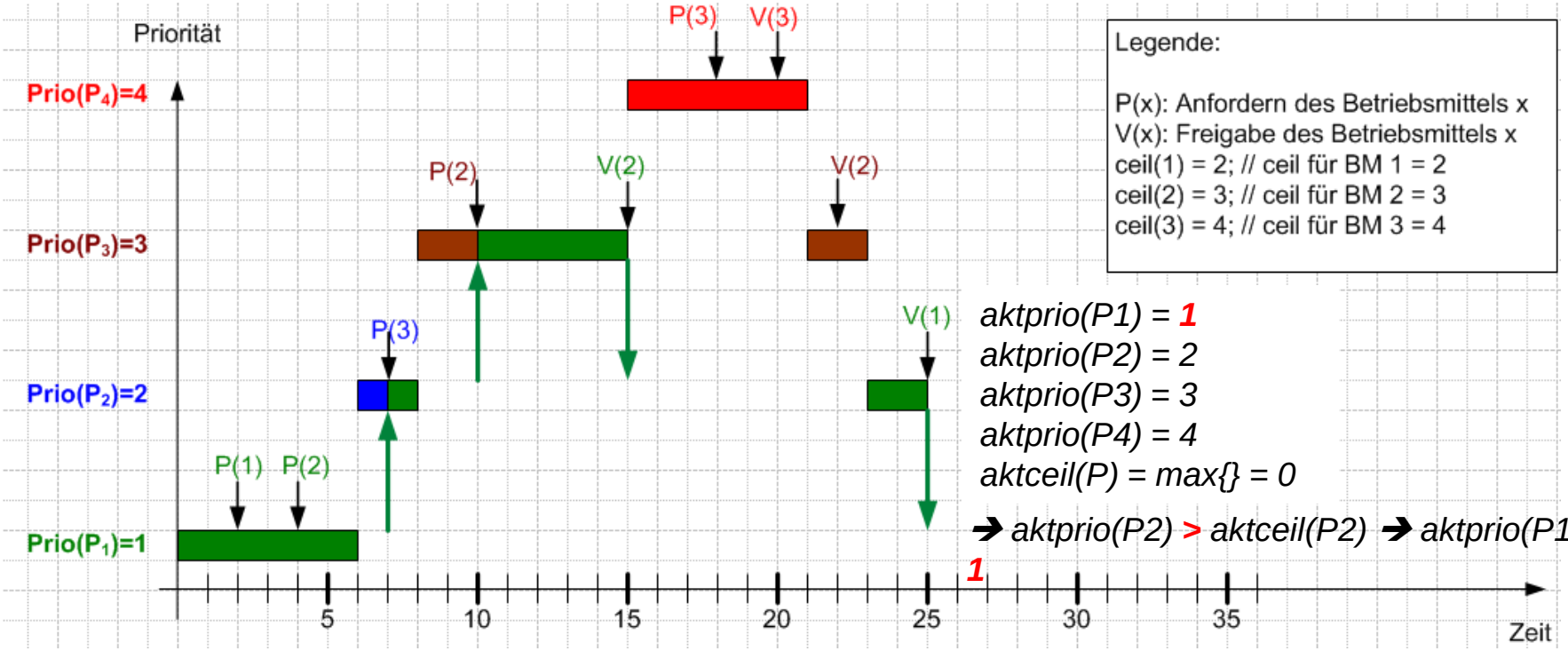
Beispiel: Prioritätsobergrenzen



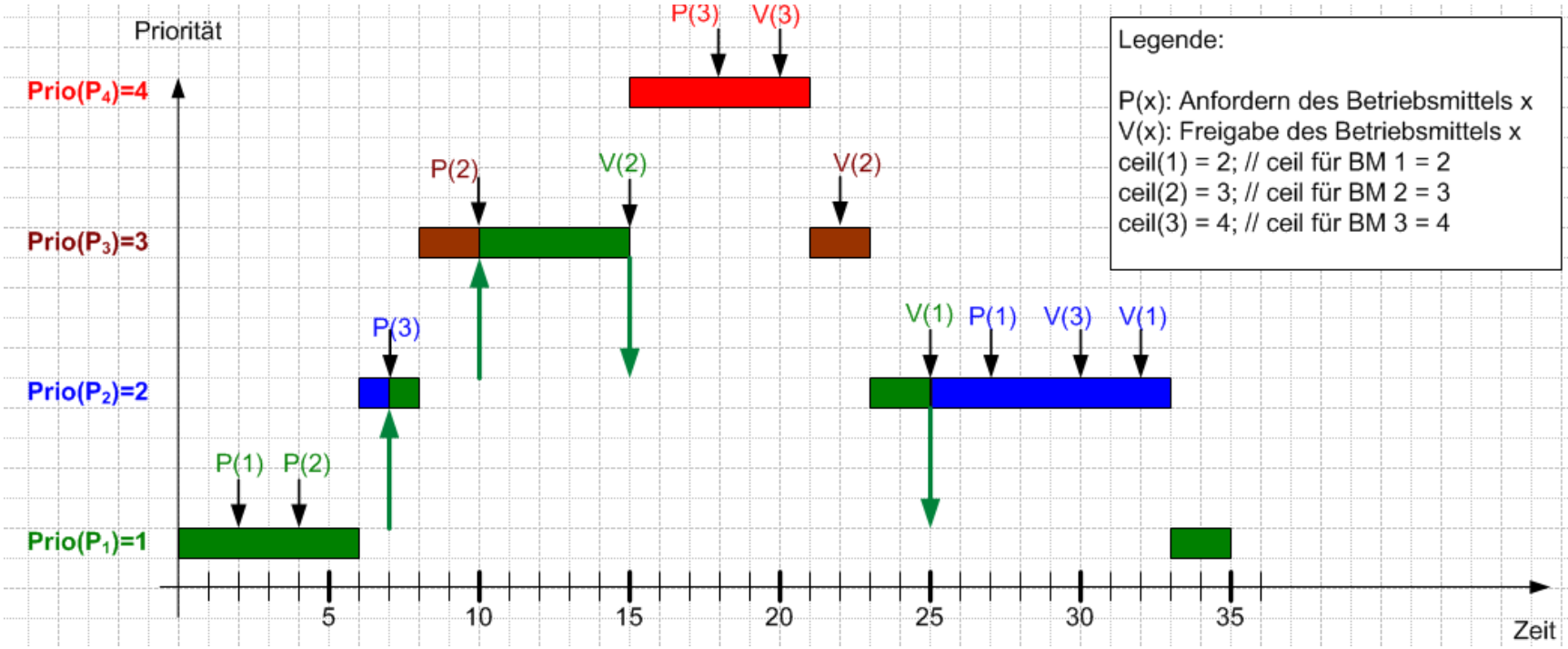
Beispiel: Prioritätsobergrenzen



Beispiel: Prioritätsobergrenzen



Beispiel: Prioritätsobergrenzen

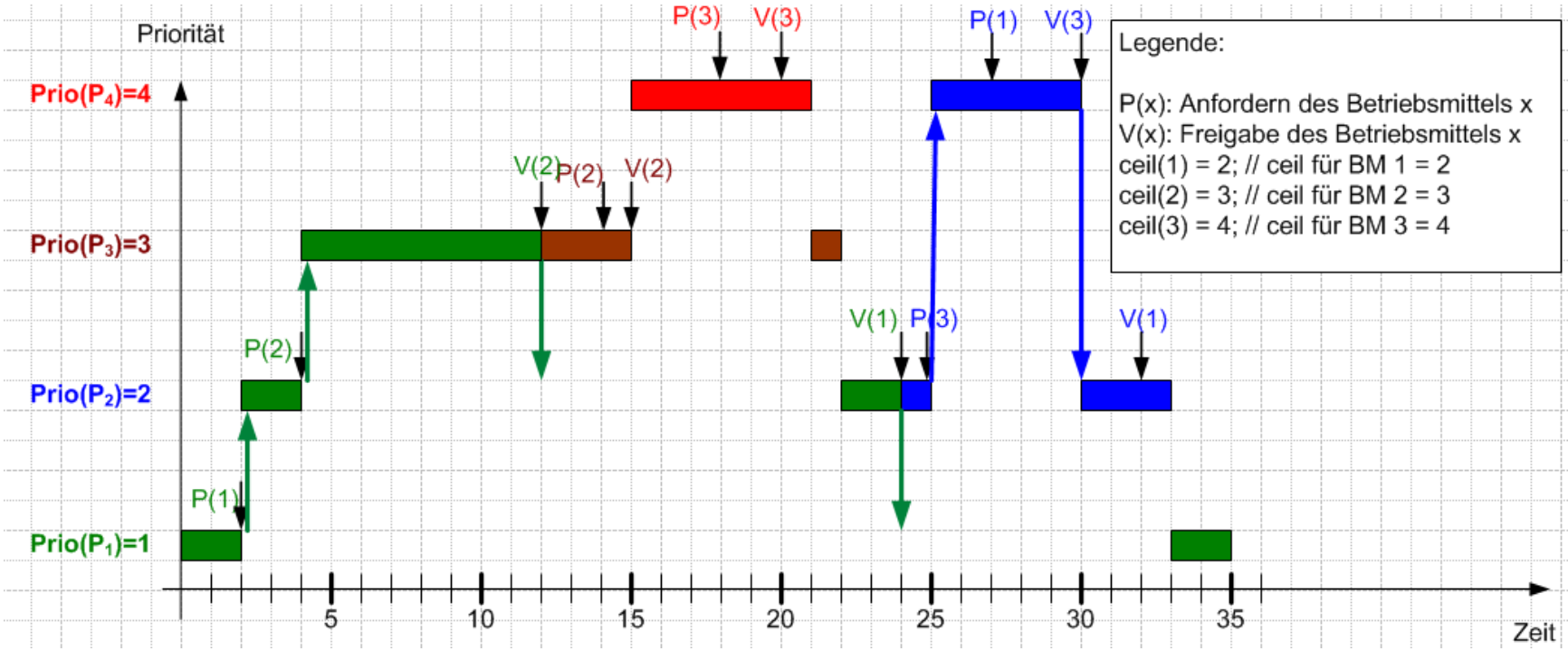


Immediate Priority Ceiling

- Vereinfachtes Protokoll: **Immediate priority ceiling**: Prozesse, die ein Betriebsmittel s belegen, bekommen sofort die Priorität ceil(s) zugewiesen.
- Anwendungsgebiet:
  - Verwendung vor allem bei Systemen, in denen sich mehrere Jobs einen gemeinsamen Stack teilen (➔ Protokoll wird häufig auch Stack Based Priority Ceiling genannt)
  - Stack wird nach LIFO-Prinzip verwaltet, d.h. ein Job muss sich erst beenden und den Speicherplatz auf den Stack freigeben, bevor ein vorherig unterbrochener Job seine Ausführung fortsetzen kann

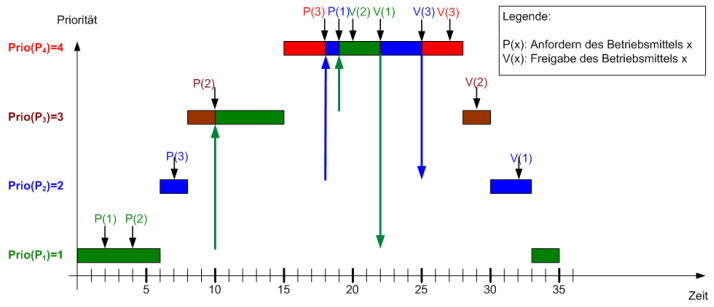
  ➔ Notwendigkeit, dass alle benötigten Betriebsmittel frei sind, da es sonst zu Verklemmungen kommen kann

Beispiel: Immediate Priority Ceiling

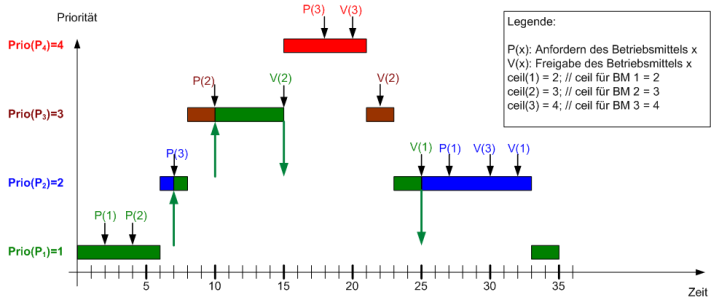


Vergleich der verschiedenen Lösungen

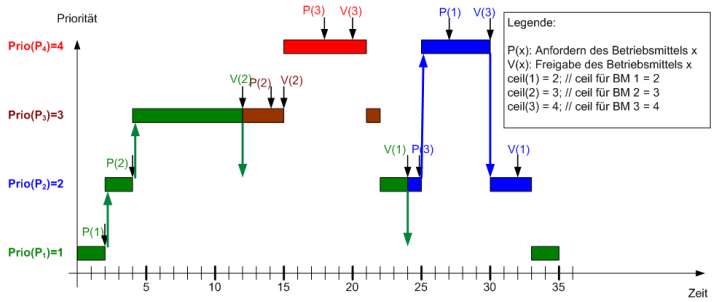
Prioritätsvererbung



Prioritätsobergrenzen

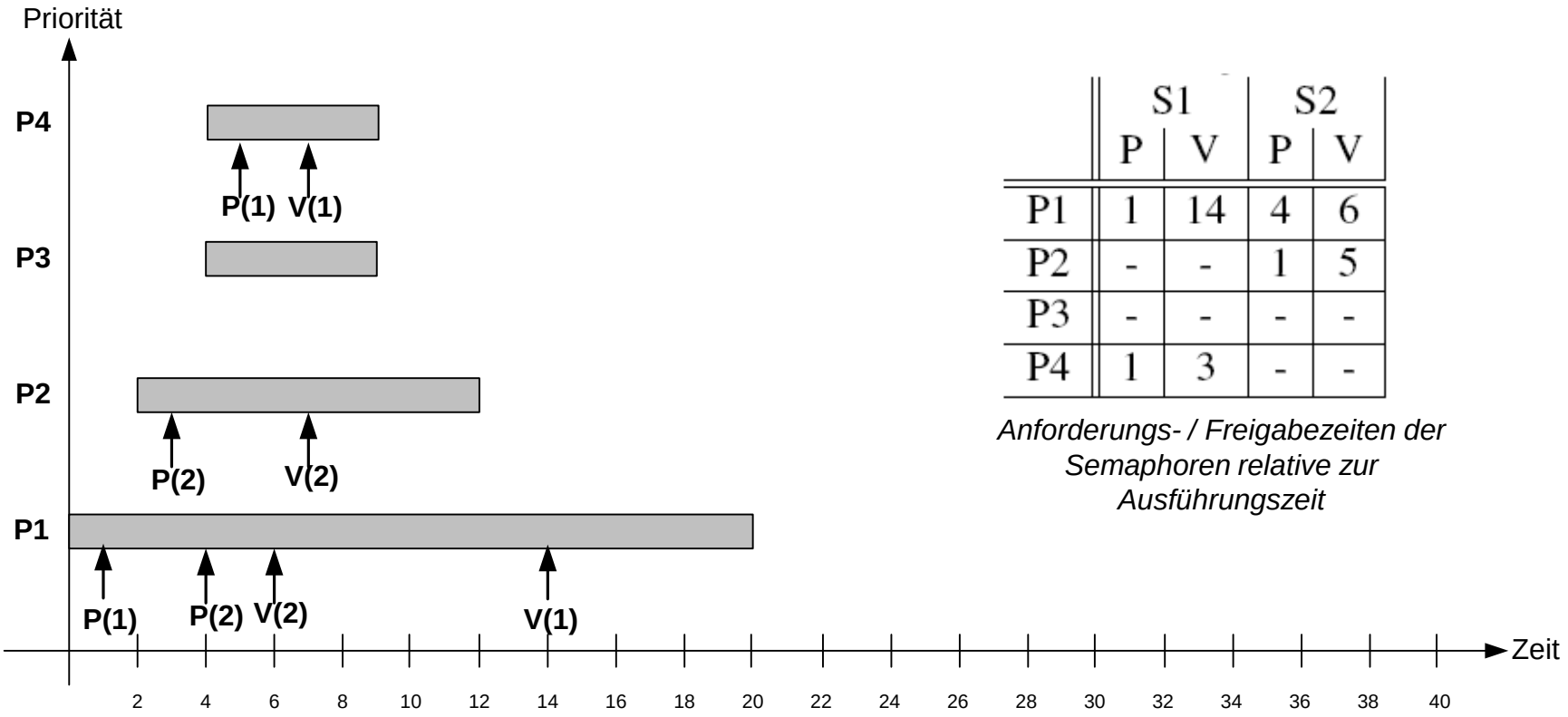


Immediate Priority Ceiling



- Prioritätsvererbung ist sehr einfach und insbesondere Anwendungsunabhängig zu implementieren
- Prioritätsobergrenzen können die Verzögerungen der höchstprioren Prozesse häufig reduzieren, sind jedoch wesentlich aufwändiger zu implementieren und benötigen insbesondere eine Analyse alle Prozesse
- Wesentlicher Vorteil von Prioritätsobergrenzen: Blockierungsketten und damit auch Verklemmungen werden ausgeschlossen

Klausur WS 06/07 – Szenario (15 Punkte = 15min)



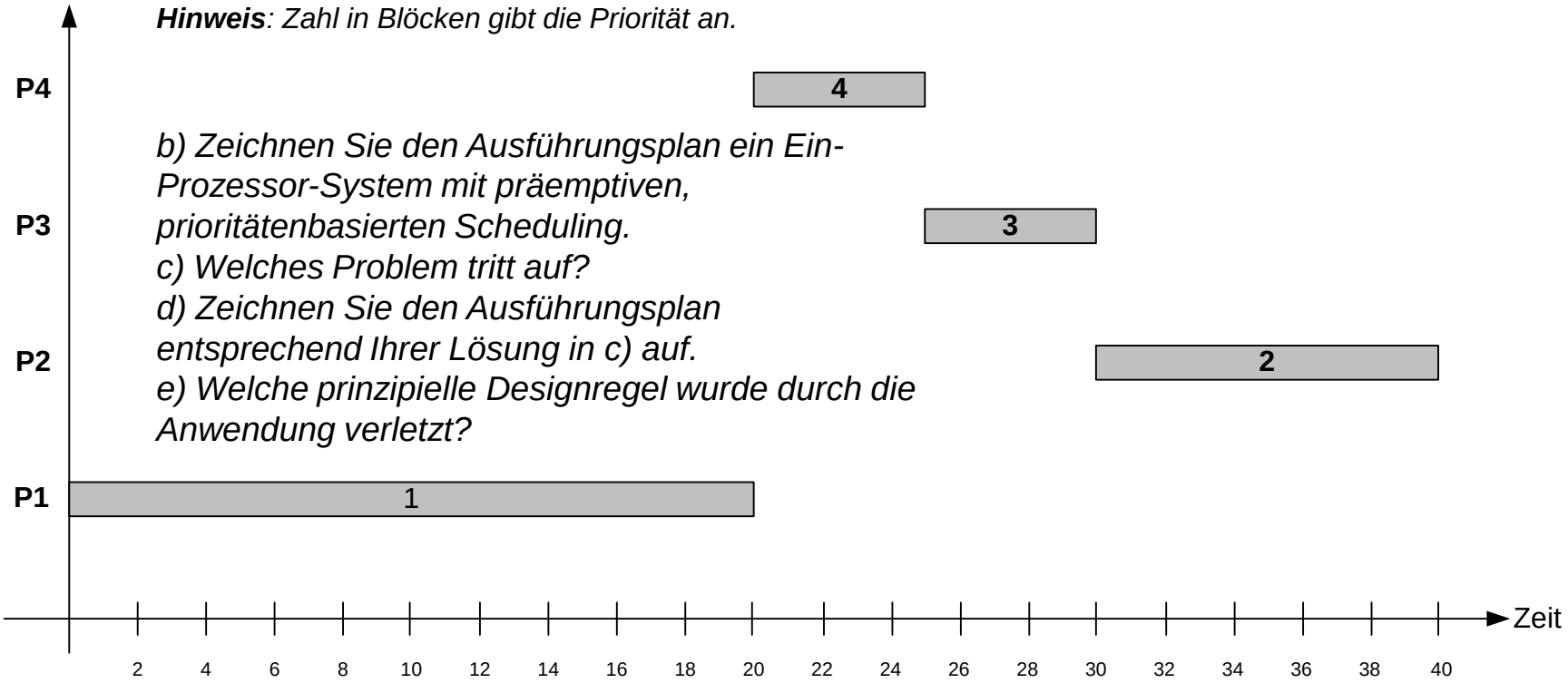
	S1		S2	
	P	V	P	V
P1	1	14	4	6
P2	-	-	1	5
P3	-	-	-	-
P4	1	3	-	-

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.*

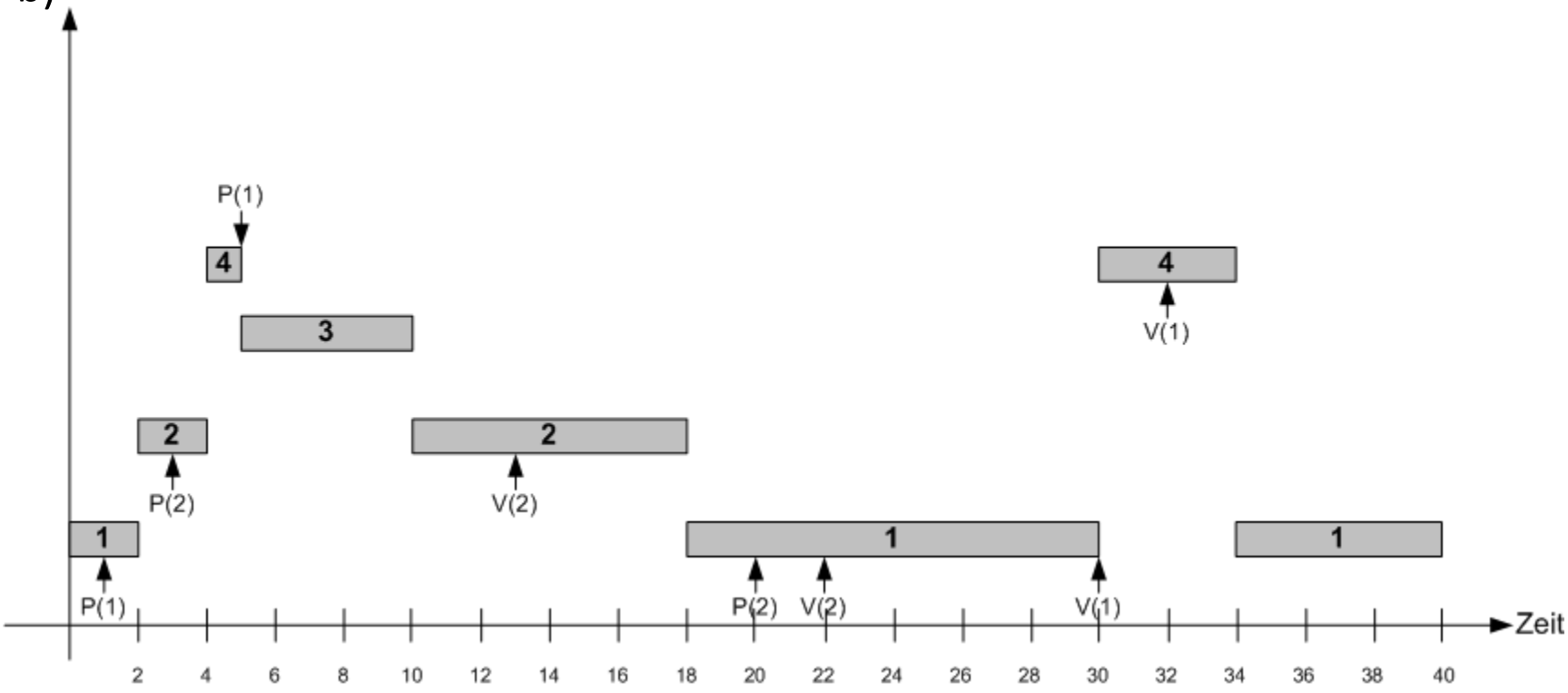
b) Zeichnen Sie den Ausführungsplan ein Ein-Prozessor-System mit präemptiven, prioritätenbasierten Scheduling.
c) Welches Problem tritt auf?
d) Zeichnen Sie den Ausführungsplan entsprechend Ihrer Lösung in c) auf.
e) Welche prinzipielle Designregel wurde durch die Anwendung verletzt?



Lösung Teilaufgabe a) + b)

a) Nicht-präemptives, prioritätenbasiertes Scheduling

b)



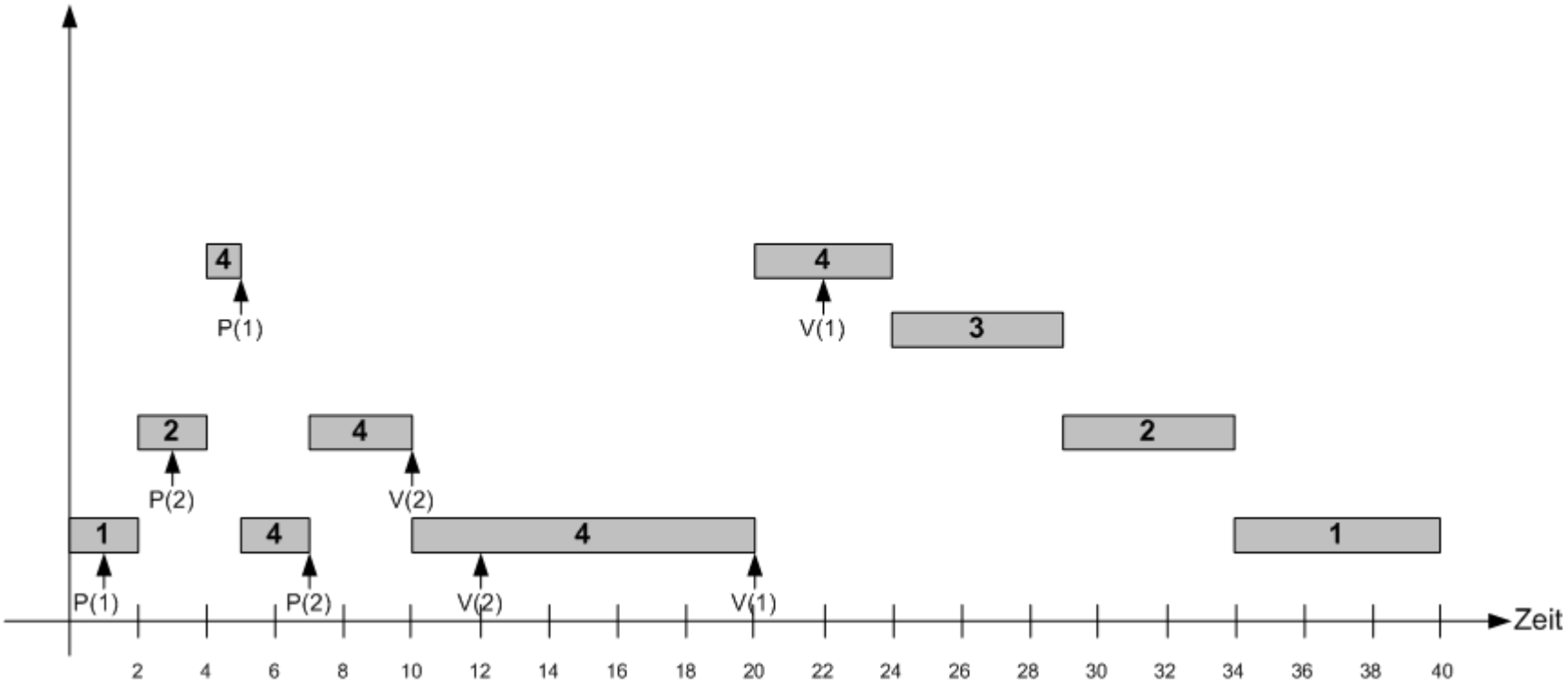
Lösung

c) Prioritätsinversion, Möglichkeiten zur Behebung Prioritätsvererbung, (sofortige) Prioritätsgrenzen

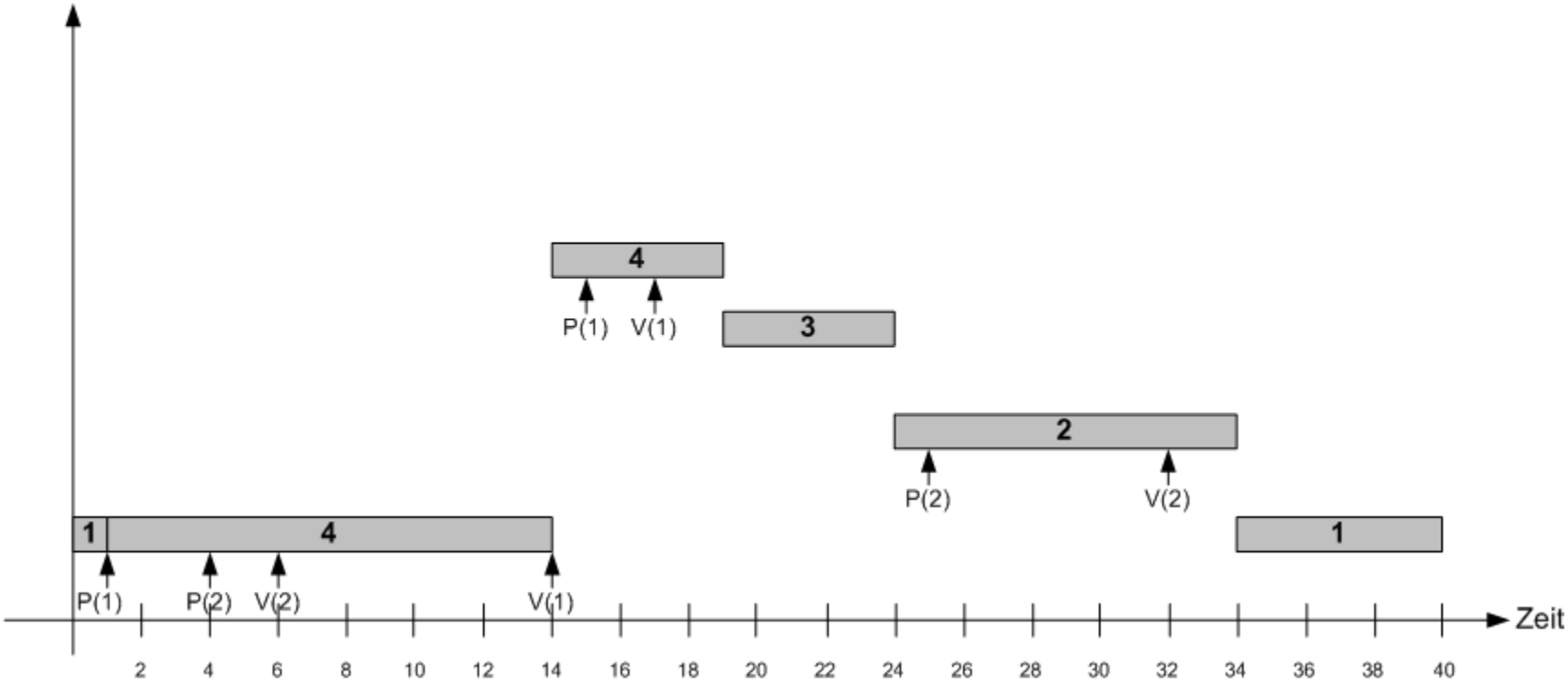
d) Siehe folgende Folien

e) Prozesse mit geringerer Priorität sollten keine Betriebsmittel so lange blockieren, wenn diese auch von Prozessen mit höherer Priorität benötigt werden.

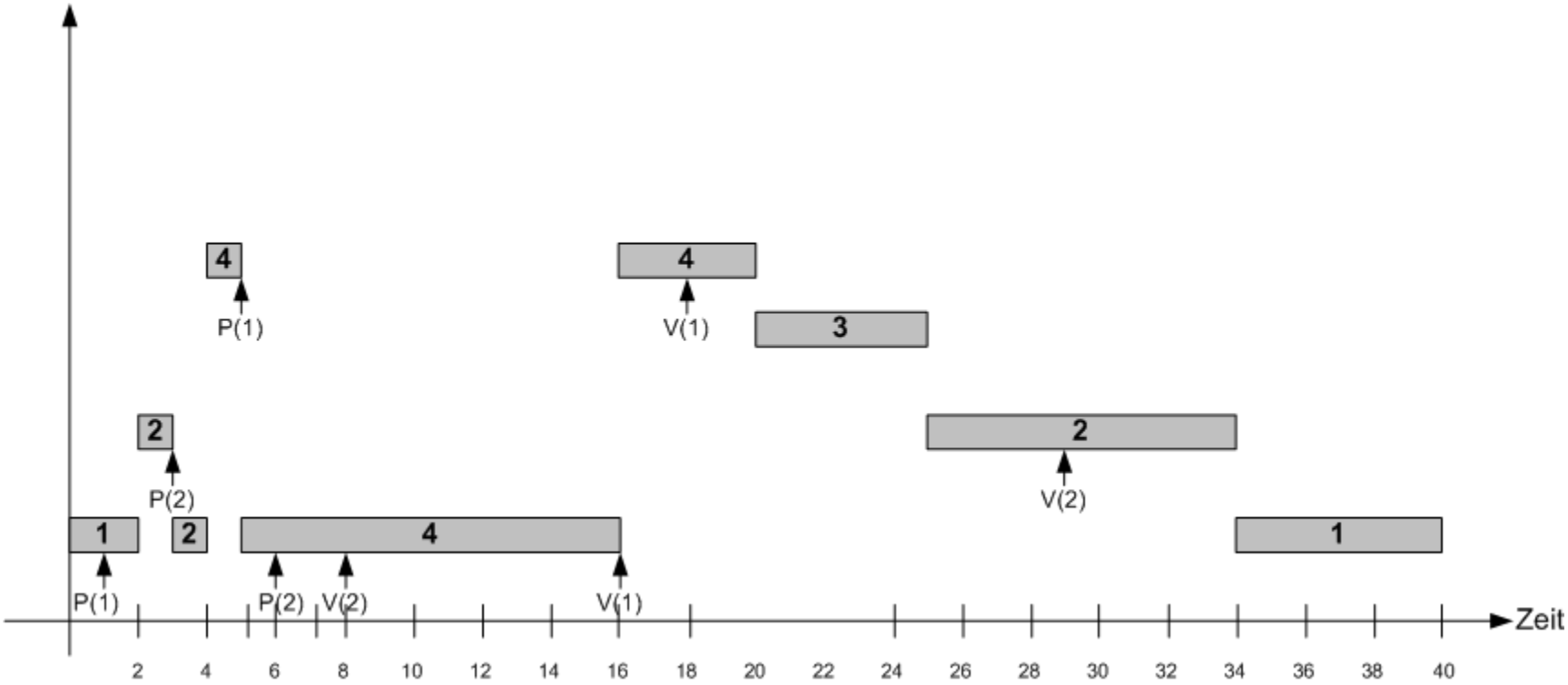
Lösung Teilaufgabe d) mit Prioritätsvererbung



Lösung Teilaufgabe d) mit sofortigen Prioritätsgrenzen



Lösung Teilaufgabe d) mit Prioritätsgrenzen

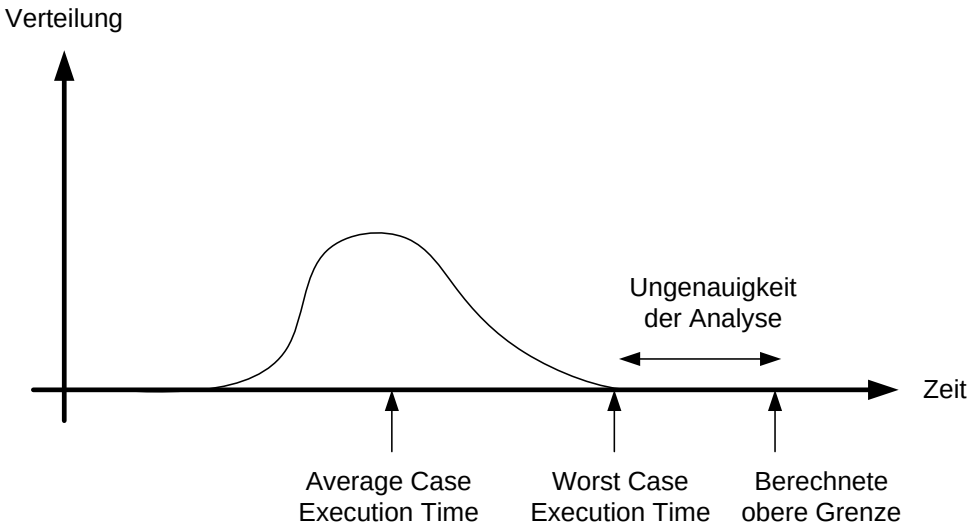


Scheduling

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)
  - Abhängigkeiten zwischen verschiedenen Prozessen werden häufig nicht berücksichtigt oder es werden hierzu strikte Annahmen getroffen (Beispiel hochkritische Software in der Avionik: zeitgesteuertes Scheduling und Löschen des Caches zu Beginn jedes Zeitslots)

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

Erfolgskontrolle: Was Sie aus dem Kapitel mitgenommen haben?

- 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
- Zeitgesteuertes Scheduling / Planen hat den Vorteil, dass es robuster gegenüber Einzelfehlern ist
- Bei prioritätsbasierten Verfahren (sowohl EDF, LST, Prioritäten) wird es umso schwieriger Aussagen zu machen, desto niedriger die Priorität ist
- Problematik der Abhängigkeiten zwischen Prozessen
- Problem der Prioritätsinversion, sowie Lösungsverfahren
- Problematik der WCET-Analyse

Kapitel 6

Echtzeitfähige Kommunikation

Zusammenhang mit Scheduling-Kapitel

- Grundsätzlich gleiches Problem:
  - Zugriff auf eine exklusive Ressource (Scheduling → Prozessor, Kommunikation → Kommunikationsmedium)
  - Protokoll muss es ermöglichen zumindest für eine Teilmenge der Nachrichten (hochpriore Nachrichten) die maximale Übertragungslatenz zu begrenzen / abzuschätzen
- Wesentlicher Unterschied:
  - Während beim Scheduling eine zentrale Entscheidung getroffen werden kann, muss bei der Kommunikation eine dezentrale Entscheidung (in jedem Rechner) getroffen werden
- Lösungsansätze (analog zum Scheduling):
  - Priorisierung → CAN, TokenRing, Flexray
  - Zeitsteuerung → TTP, Flexray

Inhalt

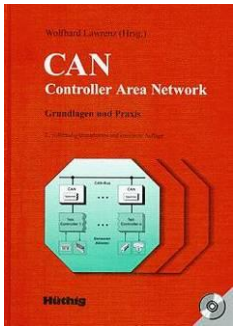
- Grundlagen
- Medienzugriffsverfahren und Vertreter
  - CSMA-CD: Ethernet
  - CSMA-CA: CAN-Bus
  - Tokenbasierte Protokolle: Token Ring, FDDI
  - Zeitgesteuerte Protokolle: TTP
  - Gemischte Verfahren: Flexray
  - Varianten Echtzeit-Ethernet

Literatur

- 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

Andrew S. Tanenbaum, Computernetzwerke, 2005

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

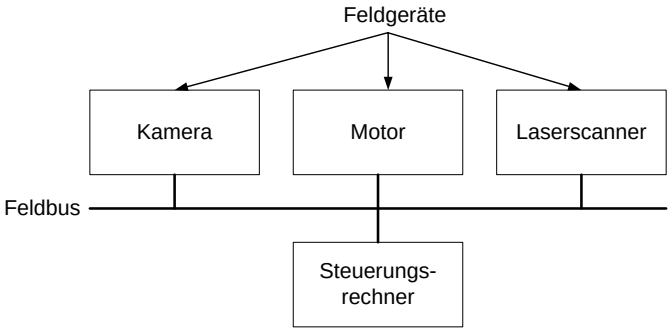


Anforderungen

- 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

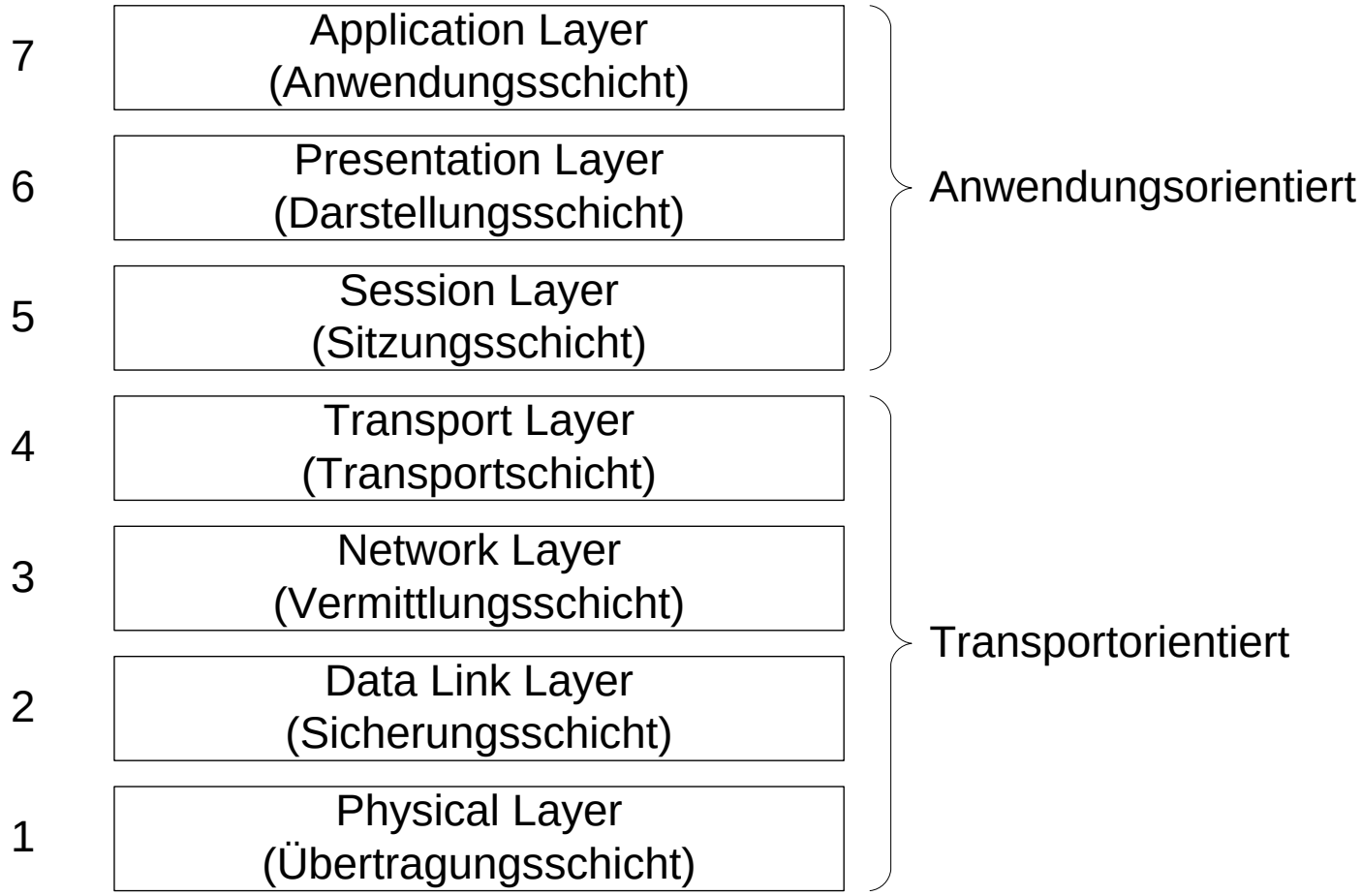
Definition Feldbus

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



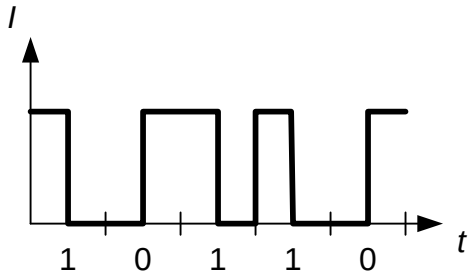
- Feldgeräte sind dabei Sensoren/Aktoren, sowie Geräte zur Vorverarbeitung der Daten.
- Der Feldbus verbindet die Feldgeräte mit dem Steuerungsgerät.
- Beobachtung: echtzeitkritische Nachrichten sind in der Regel kürzer als unkritische Nachrichten.
- Es existiert eine Vielzahl von Feldbus-Entwicklungen: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

Schichtenmodell: ISO/OSI-Modell

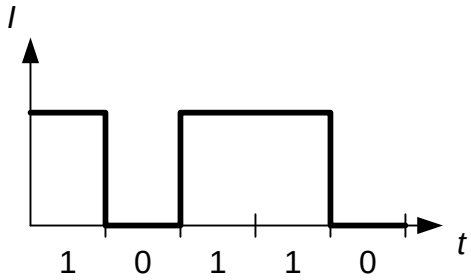


Beschreibung der einzelnen Schichten: Übertragungsschicht

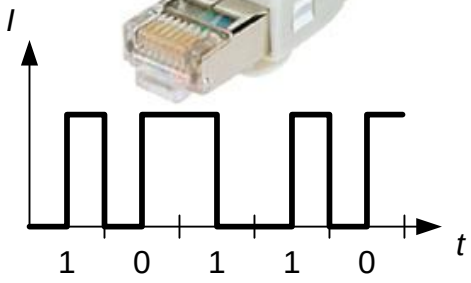
- Aufgaben:
  - Bitübertragung auf physikalischen Medium
  - Festlegung der Medien
    - elektrische, optische Signale, Funk
    - Normung von Steckern
  - Festlegung der Übertragungsverfahren/Codierung
    - Interpretation der Pegel
    - Festlegung der Datenrate



Manchester-Code



Non-return-to-zero Code

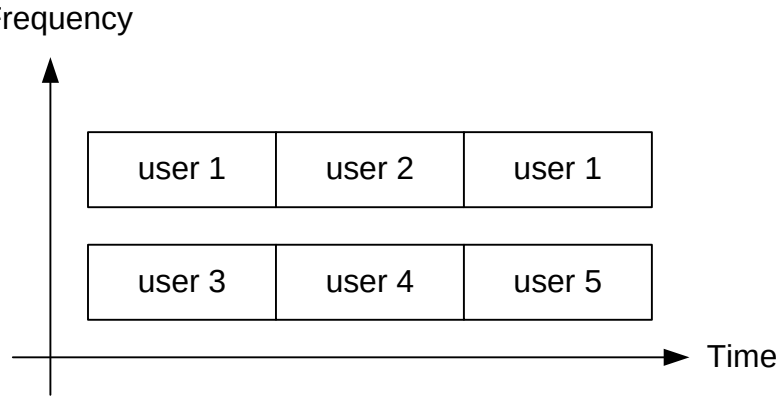


Differentieller Manchester-Code

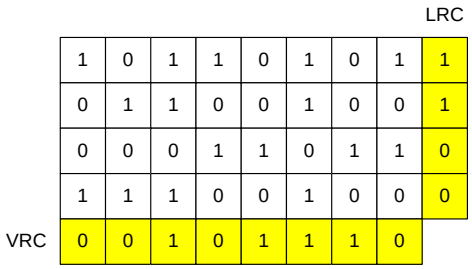
Vorteil vom Manchestercode: Taktsignal kann direkt rückgewonnen werden und Gleichanteilsfreiheit des resultierenden Signals.
Nachteil des Manchestercodes: notwendige Bandbreite doppelt so hoch wie bei Binärcodierung.

Beschreibung der einzelnen Schichten: Sicherungsschicht

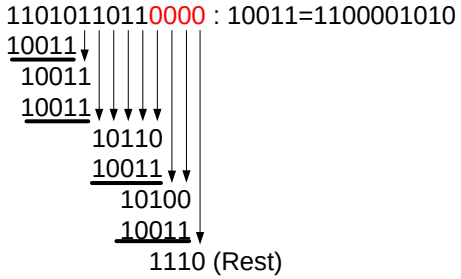
- Aufgaben:
  - Fehlererkennung
    - Prüfsummen
    - Paritätsbits
  - Aufteilung der Nachricht in Datenpakete
  - Regelung des Medienzugriffs
  - Flusskontrolle



TDMA+FDMA



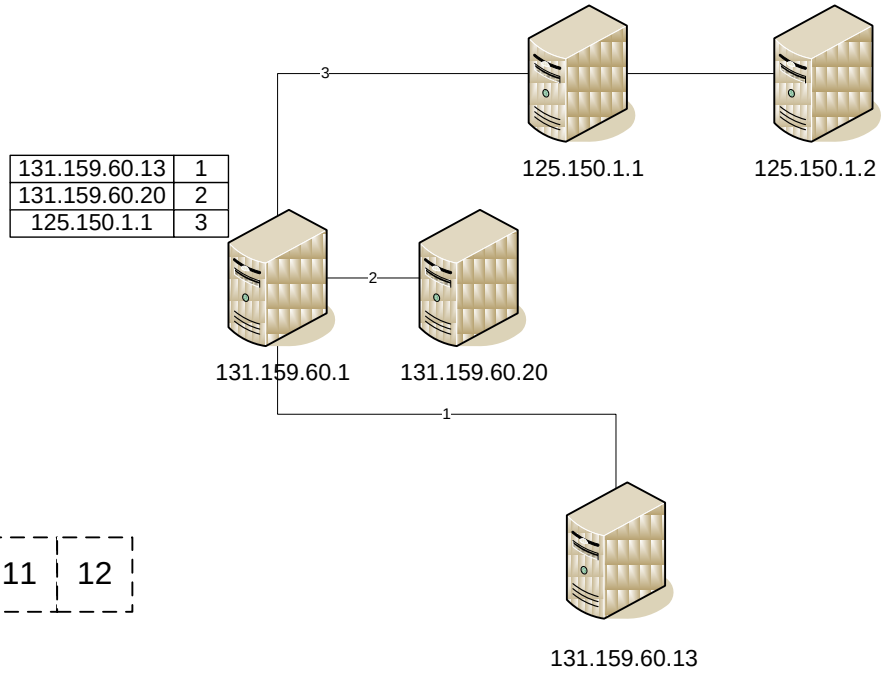
Paritätsbits

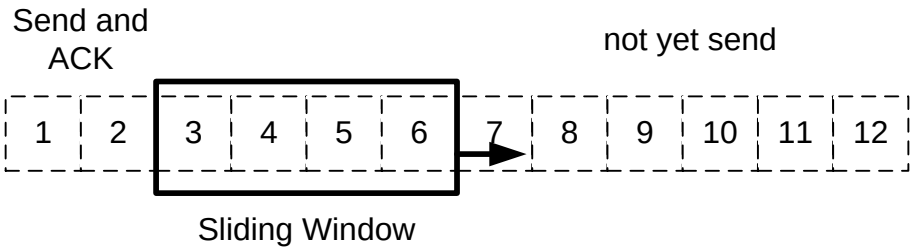


CRC-Verfahren

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
  - Übertragungsschicht

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

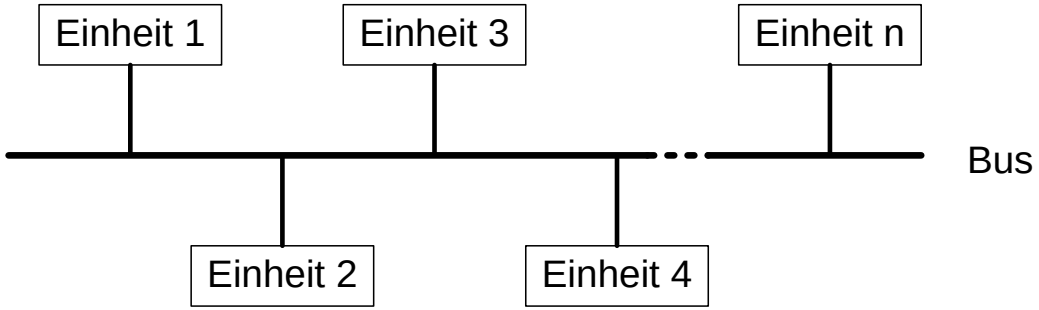
Echtzeitfähige Kommunikation

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**).



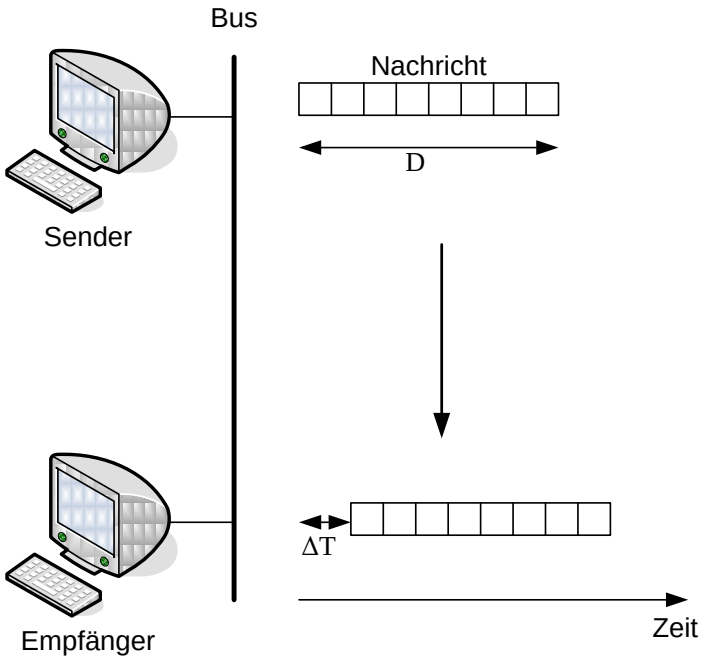
  - Während der Übertragung überprüft der sendende Knoten gleichzeitig das Resultat auf dem Bus, ergibt sich eine Abweichung, so wird eine Kollision angenommen (**Collision Detection**)

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 → 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).
  → Mit ansteigendem i wird eine Kollision immer unwahrscheinlicher.
  → 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

  → 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
- Switched Ethernet:
  - Durch den Einsatz von Switches ist das Problem von Kollisionen heute nicht mehr vorhanden: Switches puffern Nachrichten und leiten diese dann weiter
  - Bestehende Problematik: eine Priorisierung der Nachrichten beim Weiterleiten ist nicht möglich und die Zwischenspeicher sind begrenzt (→ Nachrichtenverlust bei schlechter Systemauslegung)
- Weitere Problematik: relativ großer Header → schlecht, falls nur wenig Daten übertragen werden sollen
- Aufgrund der starken Verbreitung (→niedrige Kosten, gute Unterstützung) wird Ethernet dennoch häufig in Echtzeitsystemen eingesetzt:
  - Durch Verwendung von echtzeitfähigen Protokollen in weichen Echtzeitsystemen (z.B. Multimediakontrolle).
- 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.

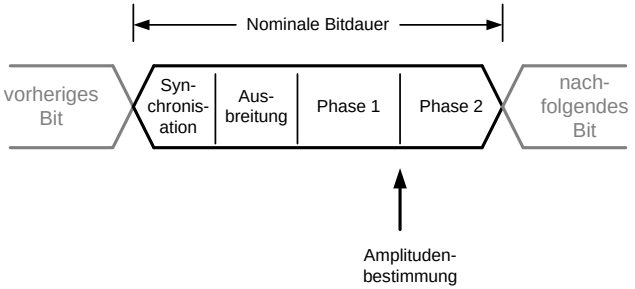
CAN-Protokoll

- 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 monotone Sequenzen von 0 oder 1 können zu Problemen bei der Synchronisation führen, in CAN wird deshalb nach fünf gleichen Bits ein inverses Bit eingefügt (**Bitstuffing**)
- Daten werden **bitsynchron** übertragen:

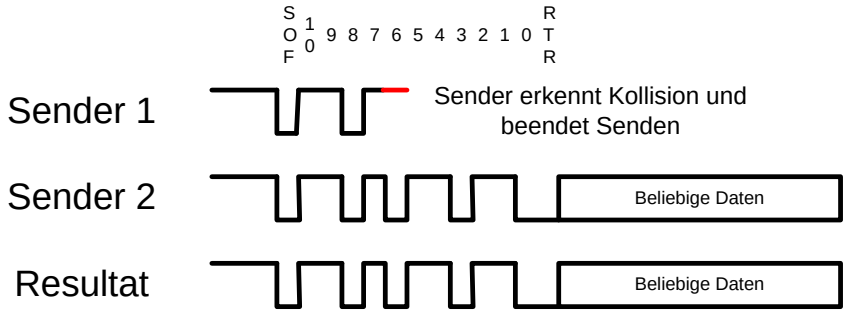
  → Datenübertragungsrate und maximale Kabellänge sind miteinander verknüpft.
  - Konfigurationsmöglichkeiten:
    - 1 MBit/s, maximale Länge: 40m
    - 500 kBit/s, maximale Länge: 100m
    - 125 kBit/s, maximale Länge: 500m
  - Maximale Teilnehmerzahl: 32-128



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

CAN: Schicht 2

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



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

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

CAN: Framearten

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

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

Datenframe

CAN: Schicht 7

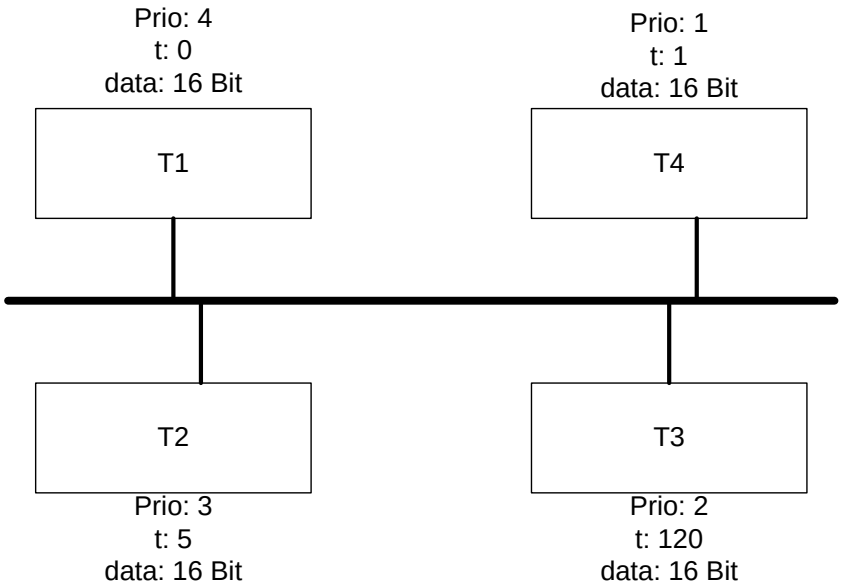
- Im Gegensatz zu Schicht 1 und 2 ist die Schicht 7 nicht in einer internationalen Norm spezifiziert.
- Es existieren jedoch diverse Implementierungen (z.B. CANOpen) für Dienste der Schichten 3-7 zur Realisierung von:
  - Flusskontrolle
  - Geräteadressierung
  - Übertragung größerer Datenmengen
  - 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

Zusammenfassung CAN

- CAN ist aufgrund des CSMA/CA Zugriffsverfahrens für harte Echtzeitsysteme geeignet:
  - Insbesondere für die hochprioren Nachrichten ist eine Abschätzung der maximalen Verzögerung leicht berechenbar (ähnliche zu nicht-präemptiven, prioritätenbasiertem Scheduling)
- Nachteile:
  - Bitsynchrones Versenden beschränkt die maximale Kabellänge → nur für lokal begrenzte Systeme sinnvoll einsetzbar
  - Die maximale Nutzdatenlänge von 8 Byte pro Frame ist sehr gering

Klausur 06/07 (modifiziert) – CAN (8 Punkte = 8 min)

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 44 Bit Steuerungsdaten pro Nachricht notwendig. Zwischen den einzelnen Nachrichten ist eine Lücke von mindestens 3 Bit.



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

Lösung: Nachricht von T1 (einziger Rechner der zunächst senden will),
Nachricht von T4 (Priorität),
Nachricht von T3 (Priorität),
Nachricht von T2

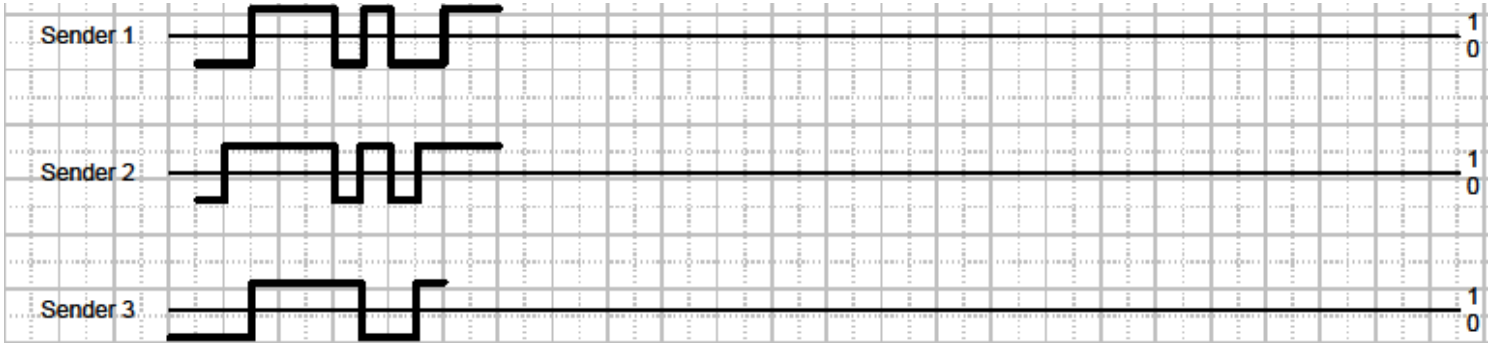
Klausur Wintersemester 07/08 (20 Punkte = 20min)

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.

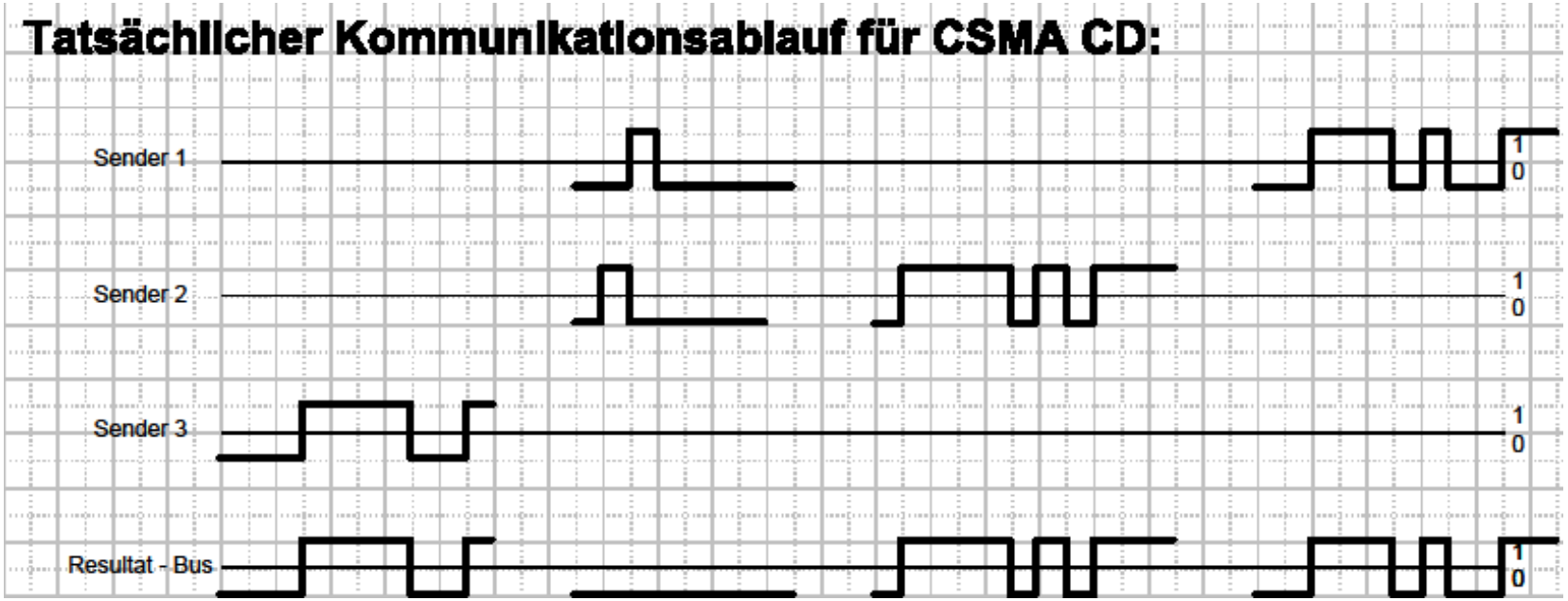
Gehen Sie davon aus, dass für die Lösung der Aufgabe alle Daten bitsynchron übertragen werden. Das JAM-Signal soll aus einer Folge von 5 0-Bits bestehen. Das 0-Bit ist dominant. Zwischen zwei Nachrichten gibt es eine Pause (interframe gap) von mindestens 3 Bits.

- 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

Lösung: 4 Zeiteinheiten

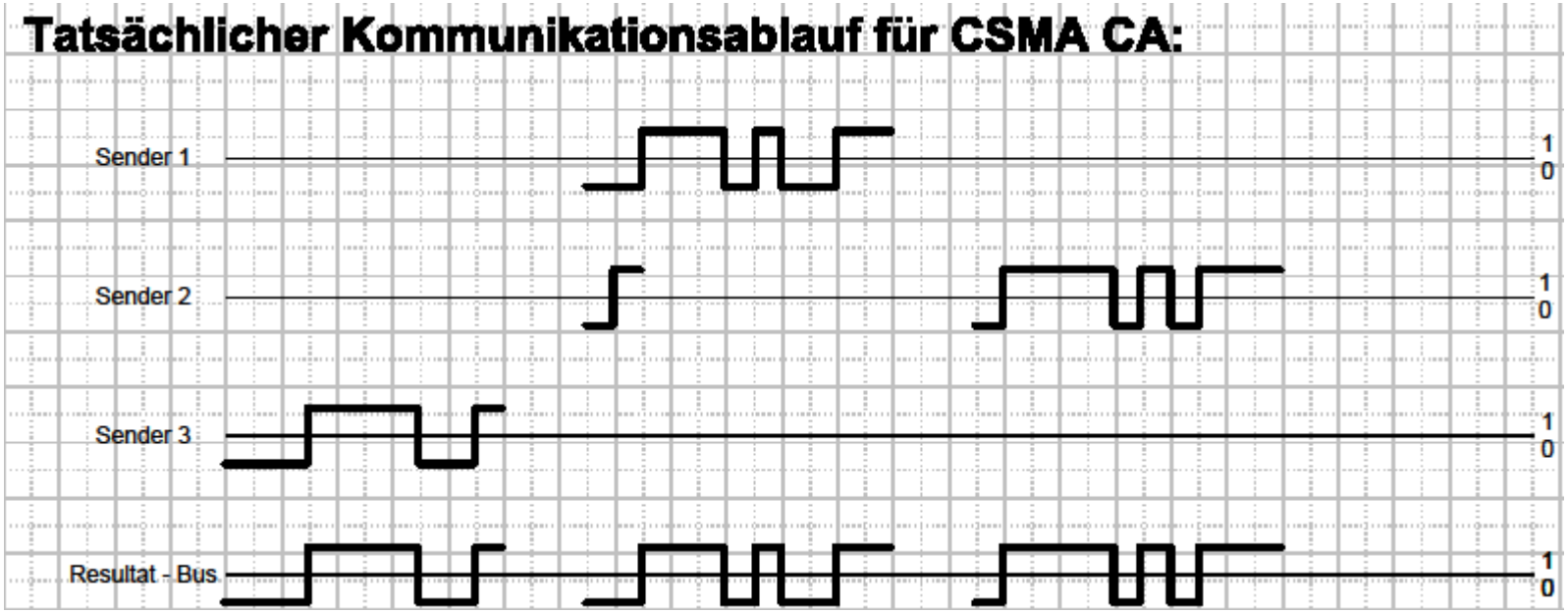


Klausur Wintersemester 07/08 – Lösung CSMA/CD



Grundsätzlich gibt es natürlich sehr viele Lösungen durch die verschiedenen möglichen Wartezeiten aufgrund des Backoff-Verfahrens.

Klausur Wintersemester 07/08 – Lösung CSMA/CA



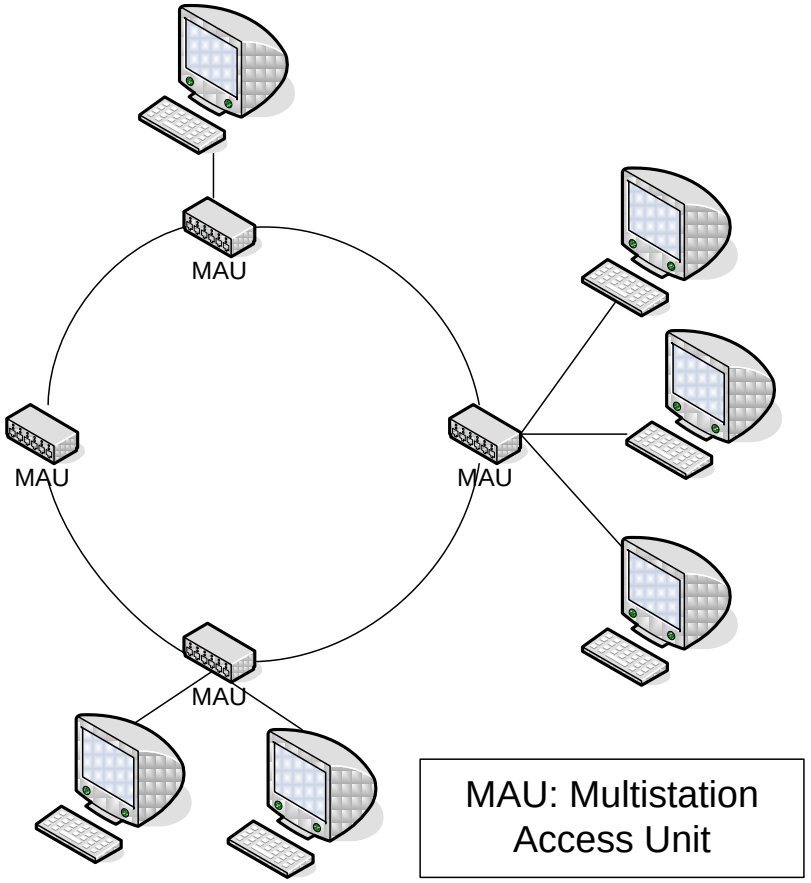
Echtzeitfähige Kommunikation

Tokenbasierte Verfahren

Vertreter: Token Ring

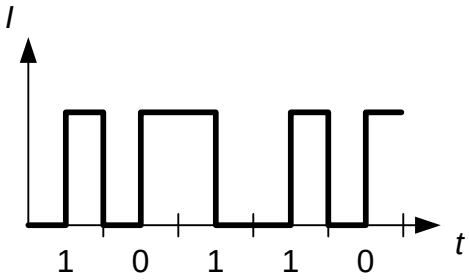
Tokenbasierte Verfahren

- 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
  → 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
- On-the-Fly Verarbeitung: Nachrichten werden im Durchlauf analysiert / modifiziert



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 (on-the-fly).
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**) es können sich gleichzeitig mehrere Token im Netz befinden

Token Ring: Prioritäten

- Token Ring unterstützt Prioritäten:
  - Insgesamt gibt es 8 Prioritätsstufen (3 Bit)
  - Jeder Nachricht wird eine Priorität zugewiesen.
  - Der Datenrahmen enthält zwei Prioritätsfelder: die Priorität der Nachricht bzw. des Tokens, sowie ein Reservierungsfeld.
  - Eine Station kann seine Priorität in dem Reservierungsfeld 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)

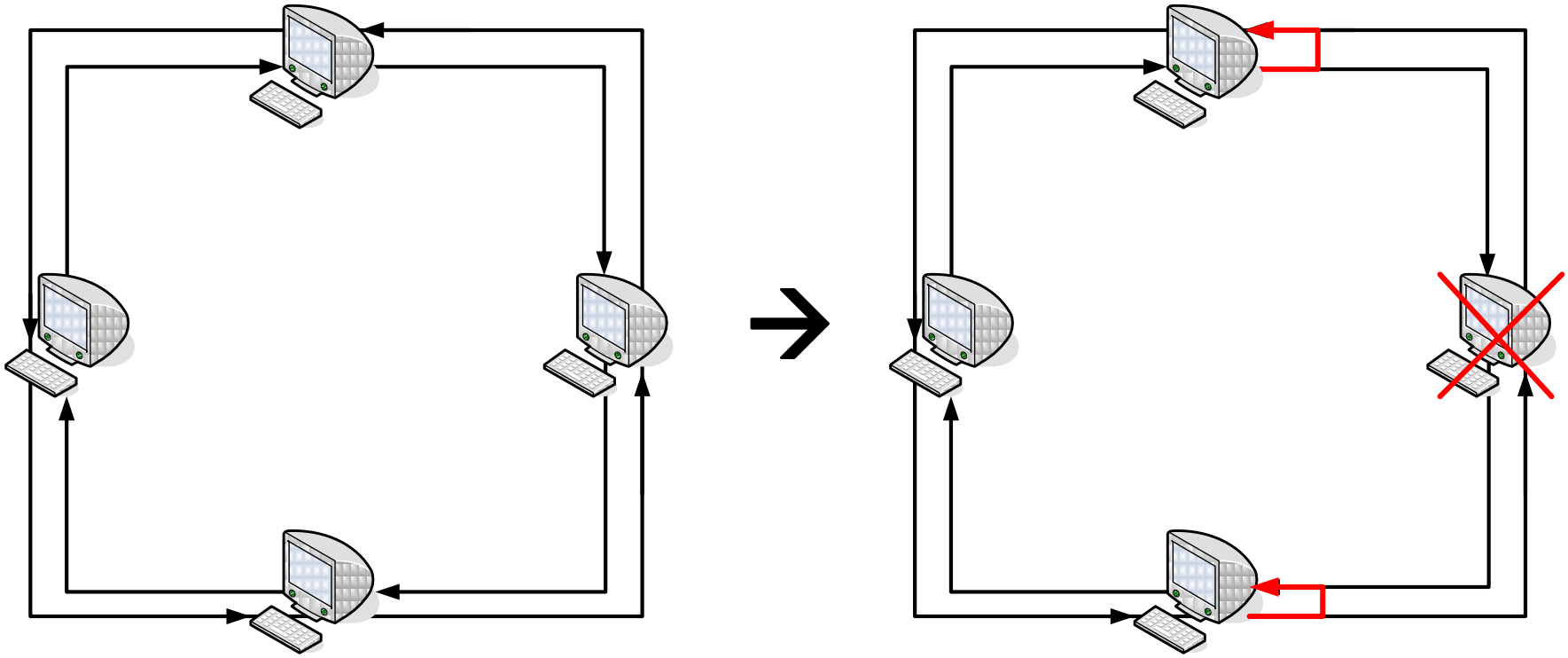
Token Ring: Initialisierung / Rekonfigurierung

- 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 → 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



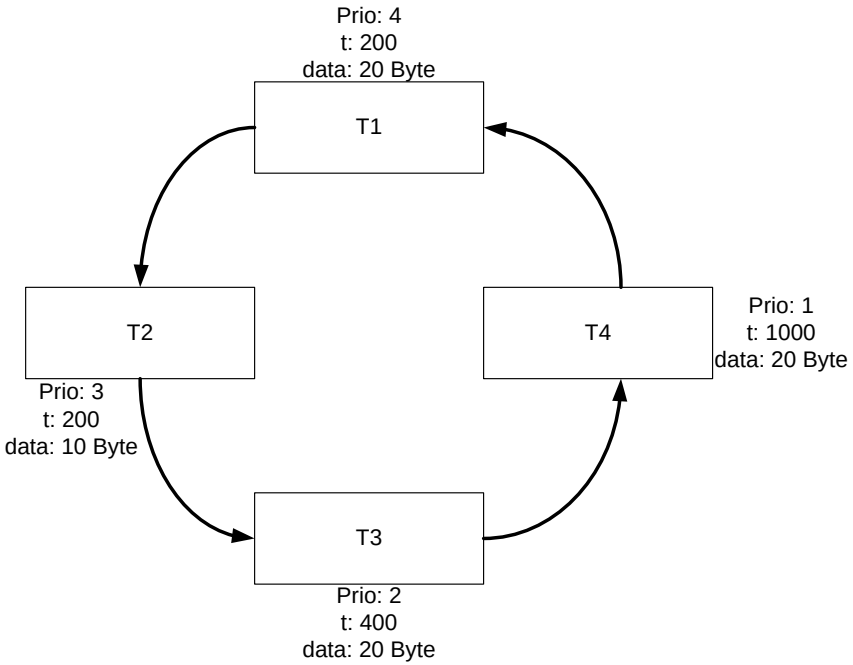
MAP / Token Bus

- **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 (8 Punkte = 8 min)

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 21 Byte.

Lösung: Token, Nachricht von T2, T3 reserviert, T1 kann wegen höherer Priorität von T3 nicht reservieren, Token, Nachricht von T3, T4 reserviert, T1 kann wegen höherer Priorität von T4 nicht reservieren, Token, Nachricht von T4, T1 reserviert, Token, Nachricht von T1



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

Echtzeitfähige Kommunikation

Zeitgesteuerte Verfahren

Vertreter: TTP, Flexray

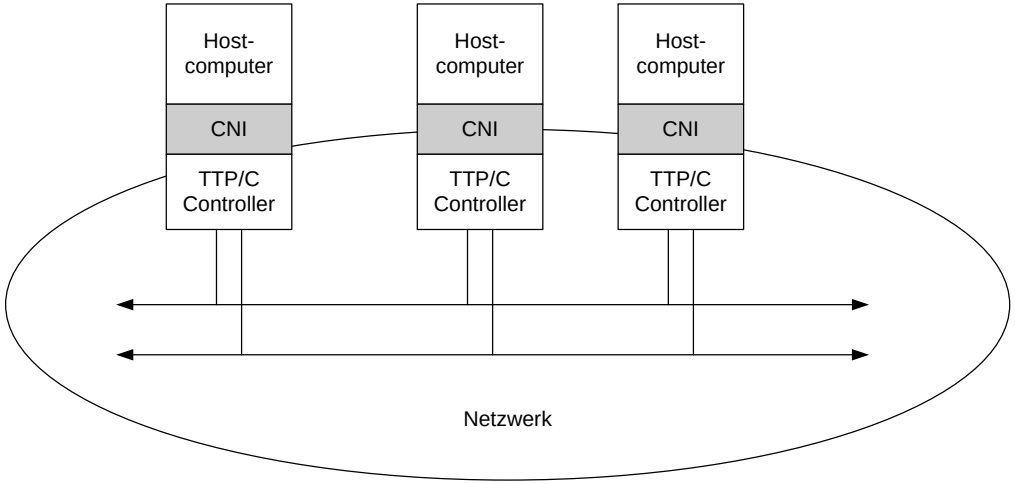
Zugriffsverfahren: TDMA

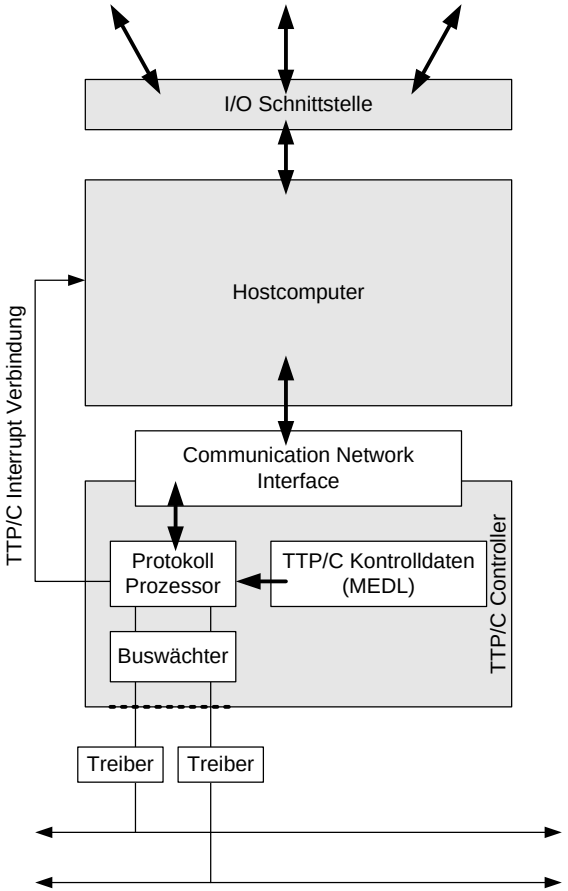
- **TDMA** (**Time Division Multiple Access**) bezeichnet ein Verfahren, bei dem der Zugriff auf das Medium in Zeitscheiben (slots) eingeteilt wird.
- Die Zeitscheiben werden für jeweils einen Sender zur Verfügung gestellt.
- Vorteile:
  - Kollisionen sind per Design ausgeschlossen
  - Einzelnen Sendern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

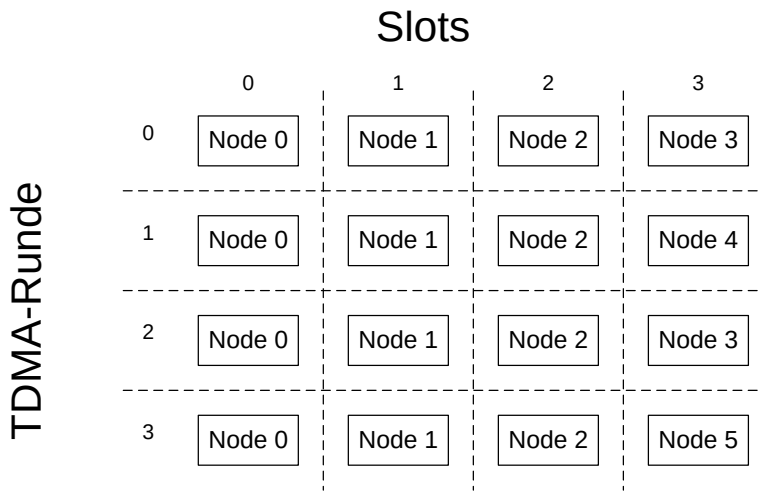




- Erläuterung:
  - Hostcomputer: Ausführung der eigentlichen Anwendung
  - CNI: Gemeinsamer Speicherbereich von Hostcomputer und TTP/C-Kontroller
  - Unterbrechungsverbindung: zur Übermittlung von Ticks der globalen Uhr und außergewöhnlicher Ereignisse an den Hostcomputer
  - MEDL: Speicherplatz für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

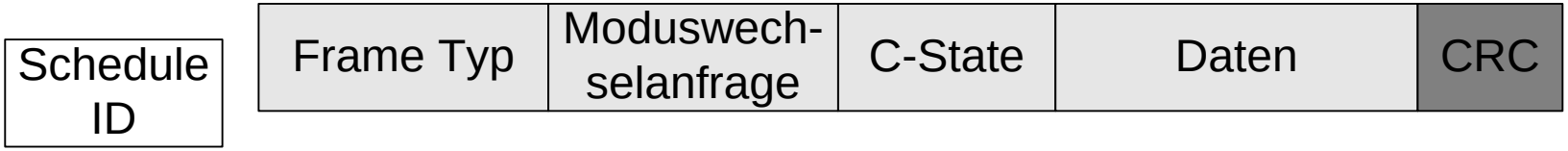
- Interne physikalische Fehler:
  - Erkennung einerseits durch das Protokoll, sowie Verhinderung eine babbling idiots durch Guards.
- Externe physikalische Fehler:
  - Durch redundante Kanäle können diese Fehler toleriert werden.
- Designfehler des TTP/C Kontrollers:
  - Es wird von einem fehlerfreien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

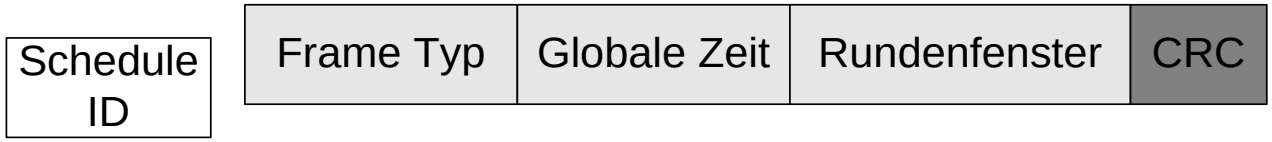
- Das Protokoll bietet Möglichkeiten, dass Netzwerk zu analysieren und fehlerbehaftete Knoten zu erkennen.
- Der Zustand des Netzwerkes wird dabei im Kontrollerzustand (C-State) gespeichert.
- Der C-State enhält:
  - die globale Zeit der nächsten Übertragung
  - das aktuelle Fenster im Clusterzyklus
  - 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

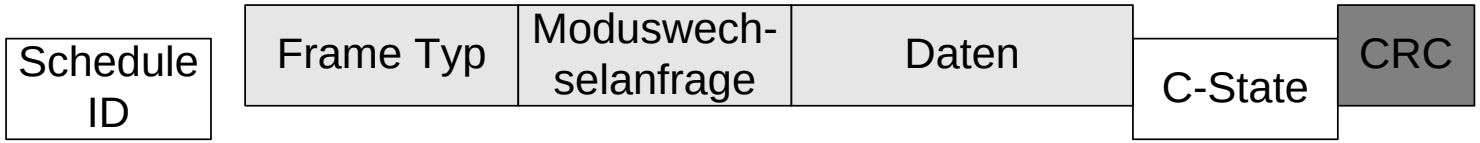
- Paket mit explizitem C-State



- 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

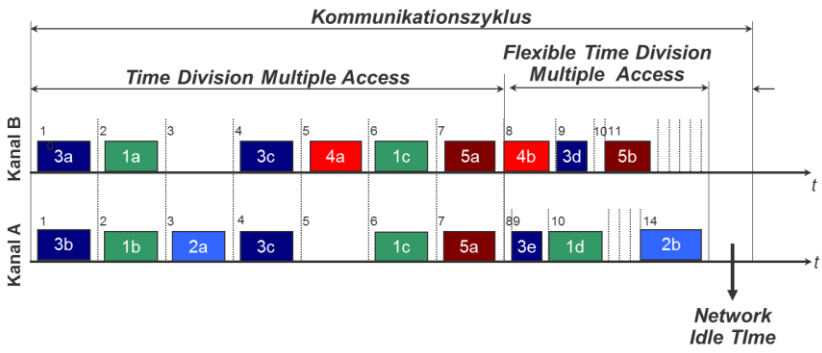
Flexray

Flexray

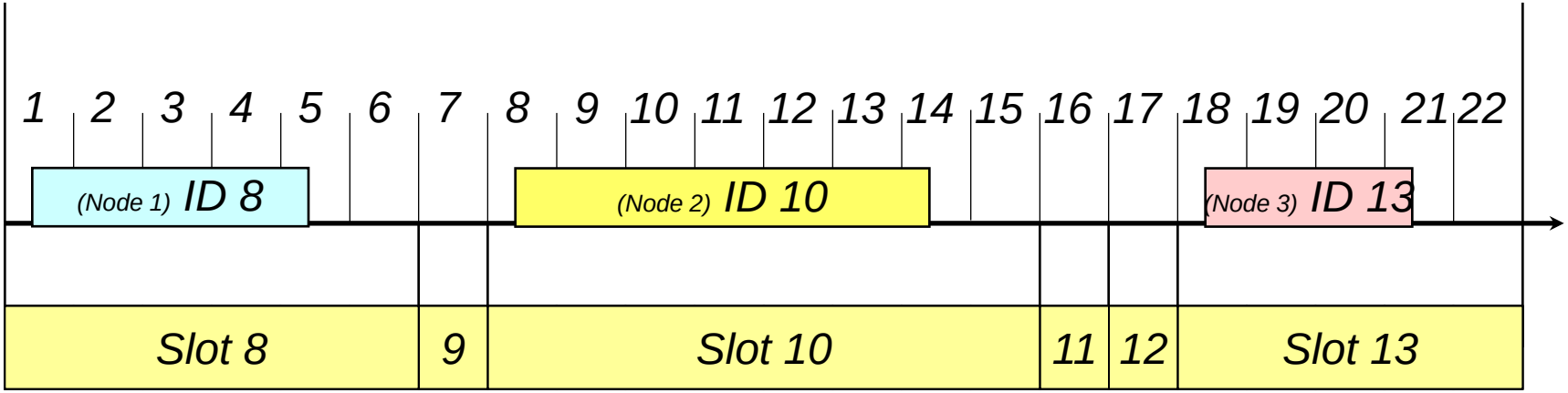
- Motivation:
  - Entwicklung eines kostengünstigen und damit automotivetauglichen Kommunikationsprotokolls, dass sowohl zeit- als auch ereignisgesteuerte Datenübertragung ermöglicht.
- Zwei Schemata integriert:
  1. TDMA – time divison multiple access (statisches Segment)
  2. FTDMA – flexible time divison multiple access (dynamisches Segment), zusätzlich implementiert FTDMA bei Flexray eine Priorisierung
- Grundlegende Kommunikationsstruktur:
  - Unterteilung in Zyklen (Zyklus besteht aus statischem u. dynamischen Segment, sowie Bereichen zur Synchronisierung)
  - Es gibt 64 dieser Zyklen, wobei diese sich unterscheiden können

	lin	CAN	FlexRay
Kanäle	1	1	1/ 2
Bus Datenrate	20 Kbit/sec	<= 1 Mbit/sec	10 Mbits/sec
Zeitgesteuert	nein	nein	ja
Deterministisch	nein	nein	ja
Redundanz	nein	nein	möglich
Arbitrierung	Master	CSMA	TDMA

FlexRay komplementiert CAN und LIN



FTDMA in Flexray



- Einführung von Minislots, die genau einem Knoten zur Verfügung stehen
- Will dieser Knoten in einem Minislot senden, so verlängert sich der Minislot entsprechend → es fallen am Ende des Zyklus Minislots weg, es muss aber sichergestellt werden, dass die Nachricht auf jeden Fall bis zum Ende des gesamten Zyklus beendet ist
- Frühere Minislots stehen somit für eine höhere Priorität

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 bzw. bei Switched Ethernet keine Priorisierung ↔ keine Eignung für Echtzeitsysteme
  - CSMA/CA,tokenbasierten Verfahren: begrenzt, aber nicht deterministisch (abhängig von anderen höherprioren 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
  - IEEE Time-Sensitive Networking Task Group (IEEE 802.1)
    - Entwicklung eines neuen Standards auf Ethernetbasis, u.a. mit Nachrichtenpräemption, Uhrensynchronisation (auf Basis von IEEE 1588), Reservierung
    - Nachfolgestandard von AVB (Audio-Video-Bridging) mit dem Ziel weitere Domänen zu unterstützen

Was Sie aus diesem Kapitel mitgenommen haben sollten

- Kenntnisse der für Echtzeitsysteme relevanten Schichten des ISO/OSI-Modells
- Verschiedenen Mechanismen der Medienzugriffsschicht und ihre Bewertung hinsichtlich der Echtzeitfähigkeit, insbesondere für welche Nachrichten einfache Aussagen zur maximalen Verzögerung gemacht werden können
- Verschiedene Kommunikationstechnologien und deren wesentliche Konzepte hinsichtlich der ISO/OSI-Schichten
- Ansätze für Echtzeit-Ethernet (wesentliche Konzepte)

Klausurfragen

- Klausur Wintersemester 07/08 (4 Punkte = 4min)
  - Erläutern Sie kurz die wesentlichen Unterschiede zwischen TokenRing, TokenBus und Ethercat in Bezug auf Topologie und Mediumszugriffverfahren.

    A: TokenRing benötigt einen physikalischen Ring, TokenBus und Ethercat beruhen auf einer beliebigen Topologie und realisieren den Ring auf logischer Ebene; bei TokenRing & TokenBus kann jeder Knoten eine Nachricht senden, wenn er im Besitz des Tokens ist, bei Ethercat werden Nachrichten immer vom Master initiiert und sonstige Knoten hängen ihre Daten an
- Wiederholungsfragen:
  1. Was ist der Unterschied zwischen dominanten und rezessiven Bits.

     A: Falls gleichzeitig dominante und rezessive Bits gesendet werden, ist am Bus das dominante Bit zu sehen.
  2. Nennen Sie zwei Mechanismen zur Bitsynchronisierung und erklären Sie diese.

     A: Bit Stuffing oder Manchestercodierung
  3. Was ist der Unterschied zwischen CSMA/CD und CSMA/CA?

     A: CD kann Kollisionen nur erkennen, CA löst diese, so dass keine Nachricht verloren geht.
  4. Erläutern Sie zwei verschiedene Ansätze um Ethernet echtzeitfähig zu machen.

     A: siehe Folie 361
  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.

     A: siehe Folie 360

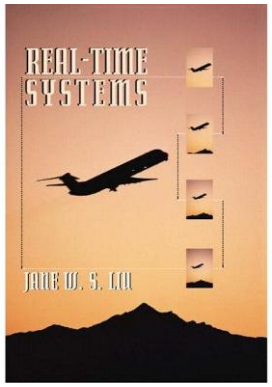
Kapitel 7

Echtzeitbetriebssysteme

Inhalt

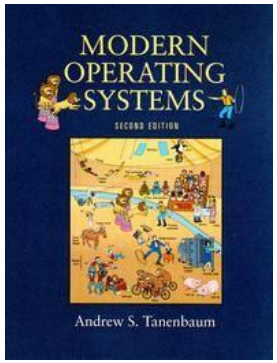
- Grundlagen
- Betrachtung diverser Betriebssysteme:
  - Domänenspezifische Betriebssysteme:
    - OSEK
    - TinyOS
  - Klassische Echtzeitbetriebssysteme
    - QNX
    - VxWorks
    - PikeOS
  - Linux- / Windows-Echtzeitvarianten
    - RTLinux/RTAI
    - Linux Kernel 2.6
    - Windows CE

Literatur



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

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



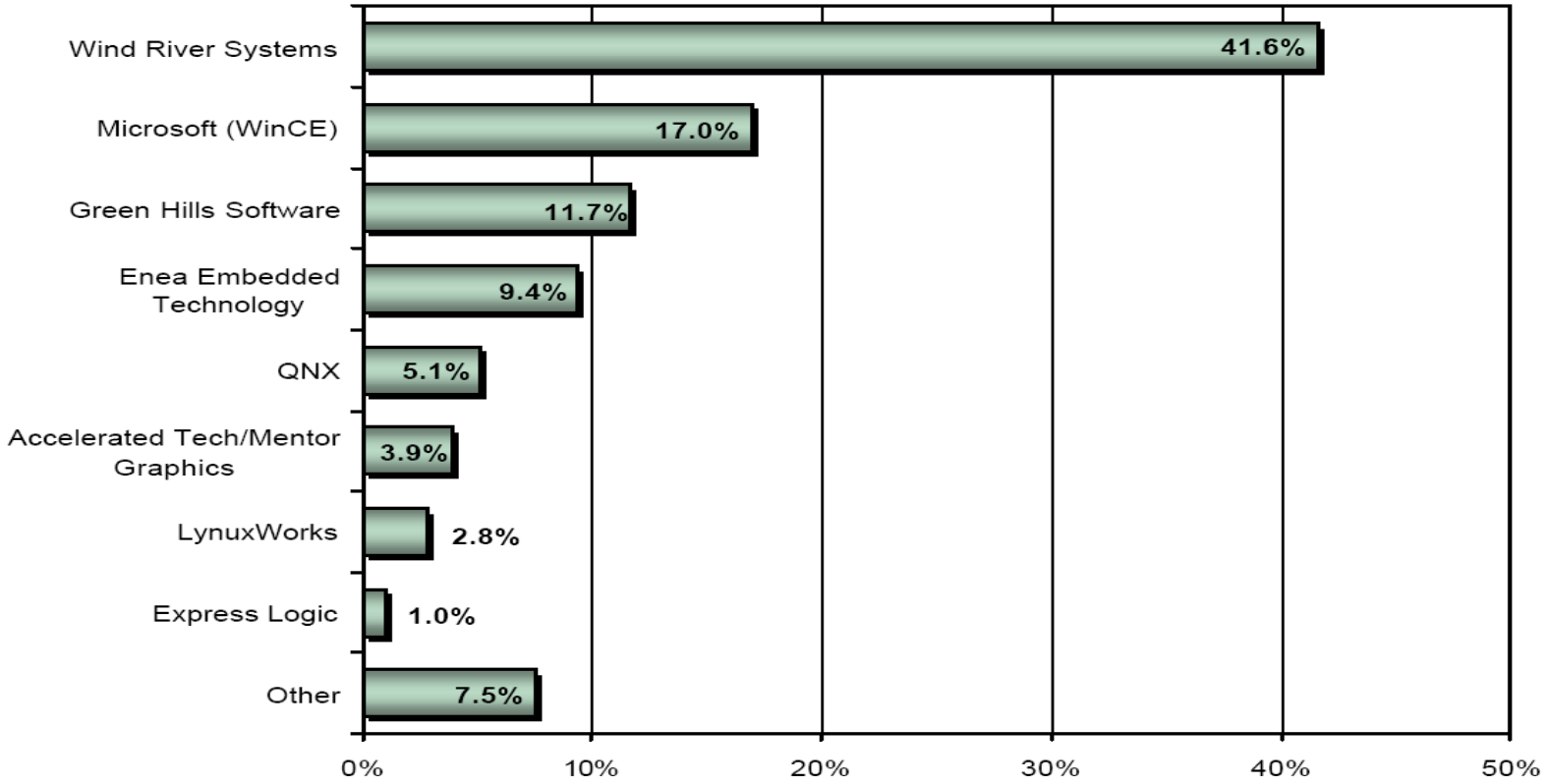
Andrew S. Tanenbaum: Modern Operating Systems, 2001

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/
- http://www.sysgo.com/

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
  - 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.

→ 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.

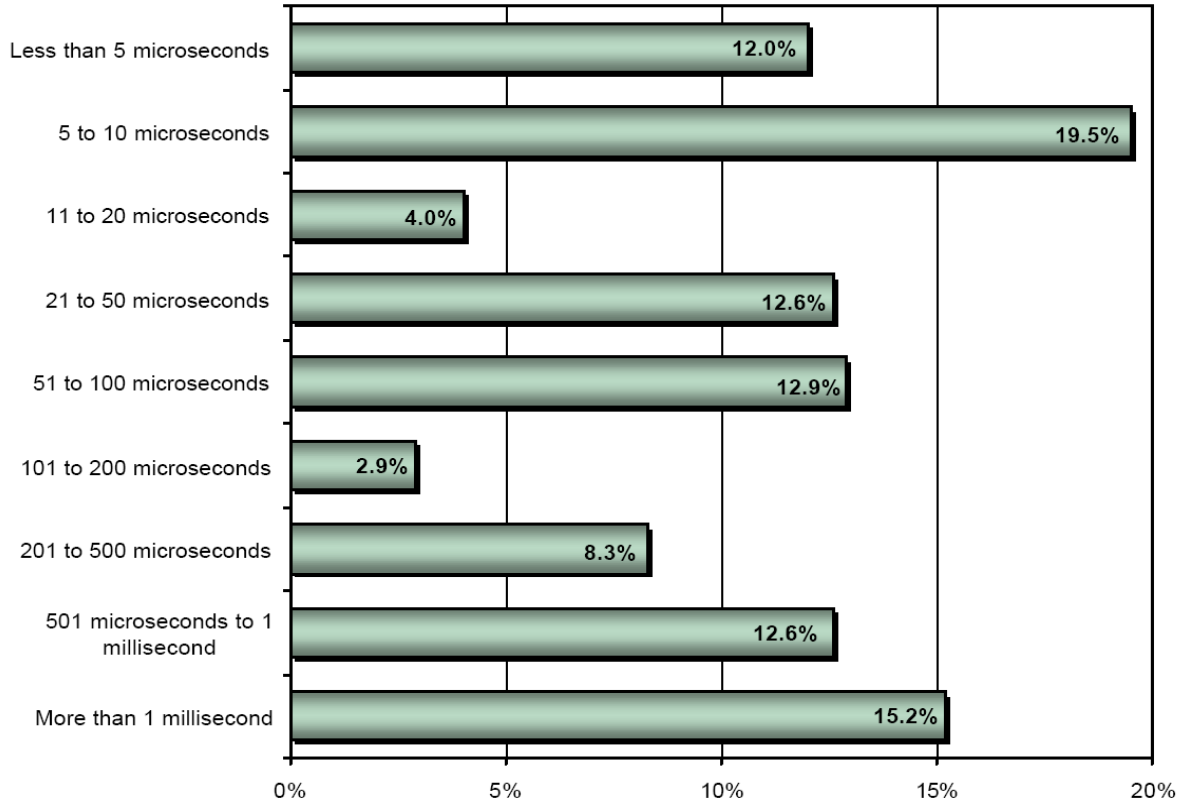
→ Echtzeitsysteme müssen aus diesen Gründen skalierbar sein:

  - Möglichkeit zur Auswahl einzelner Module entsprechend den Anforderungen an die Funktionalität der Anwendung.
  - Entscheidend ist der **minimale Speicherbedarf** (**Footprint**).

Reaktionszeiten

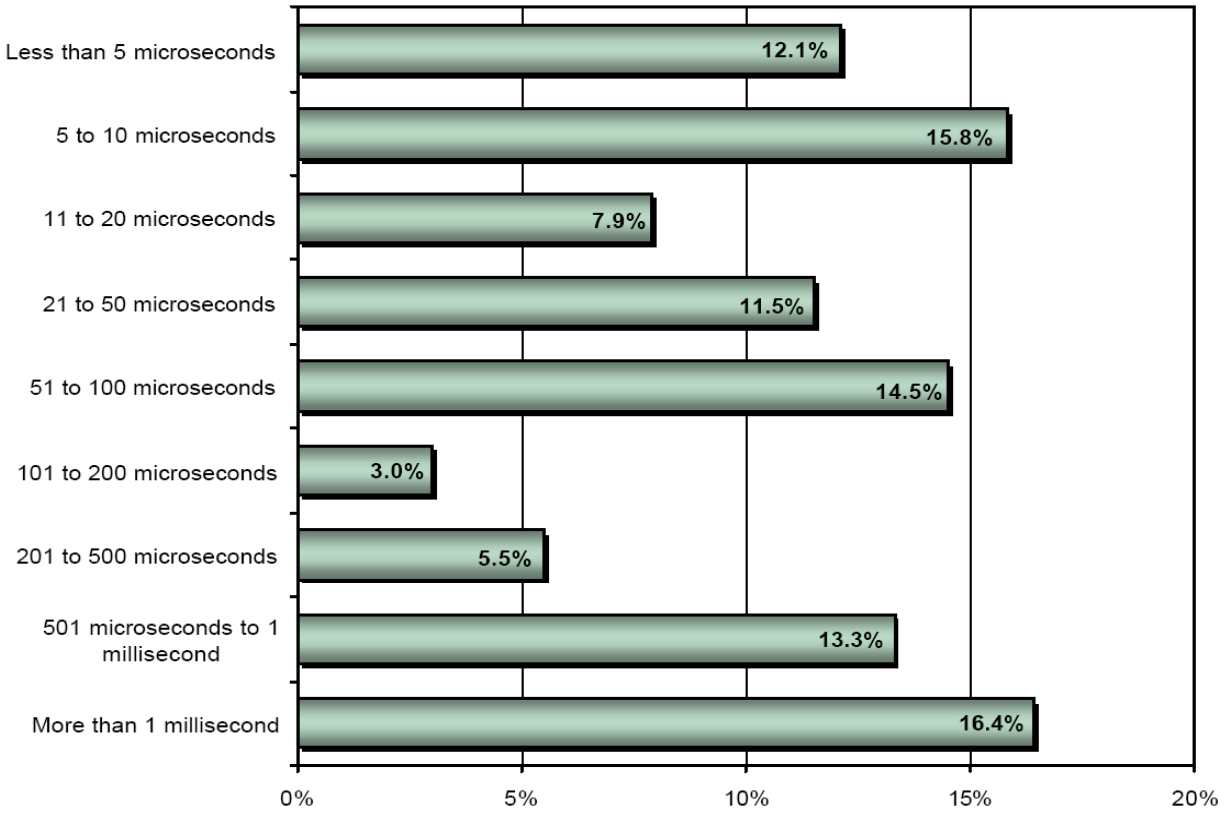
- Die Echtzeitfähigkeit wird durch die Messung folgender Zeiten bestimmt:
  - **Unterbrechungsantwortzeiten** (**interrupt latency**): der Zeitraum zwischen dem Auftreten einer Unterbrechung und der Ausführung des ersten Befehls der dazugehörigen Unterbrechungsbehandlungsroutine
  - **Schedulinglatenz**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

In Vorlesung betrachtete Echtzeitbetriebssysteme

- Der Markt der Echtzeitbetriebssysteme ist aufgrund der heterogenen Anforderungen sehr vielfältig
- Im Rahmen der Vorlesung betrachten wir verschiedene Vertreter, um typische Konzepte von Echtzeitbetriebssystemen darzustellen:
  - Anpassung an die Anforderungen der Anwendung: OSEK
  - Geringer Ressourcenverbrauch: TinyOS
  - Skalierbarkeit/Microkernelkonzept: QNX
  - Host-Target-Entwicklungsumgebung: VxWorks
  - Virtualisierung: PikeOS
- Zudem betrachten wir die Möglichkeiten Standardbetriebssysteme (Linux, Windows) für Echtzeitaufgaben einzusetzen

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.

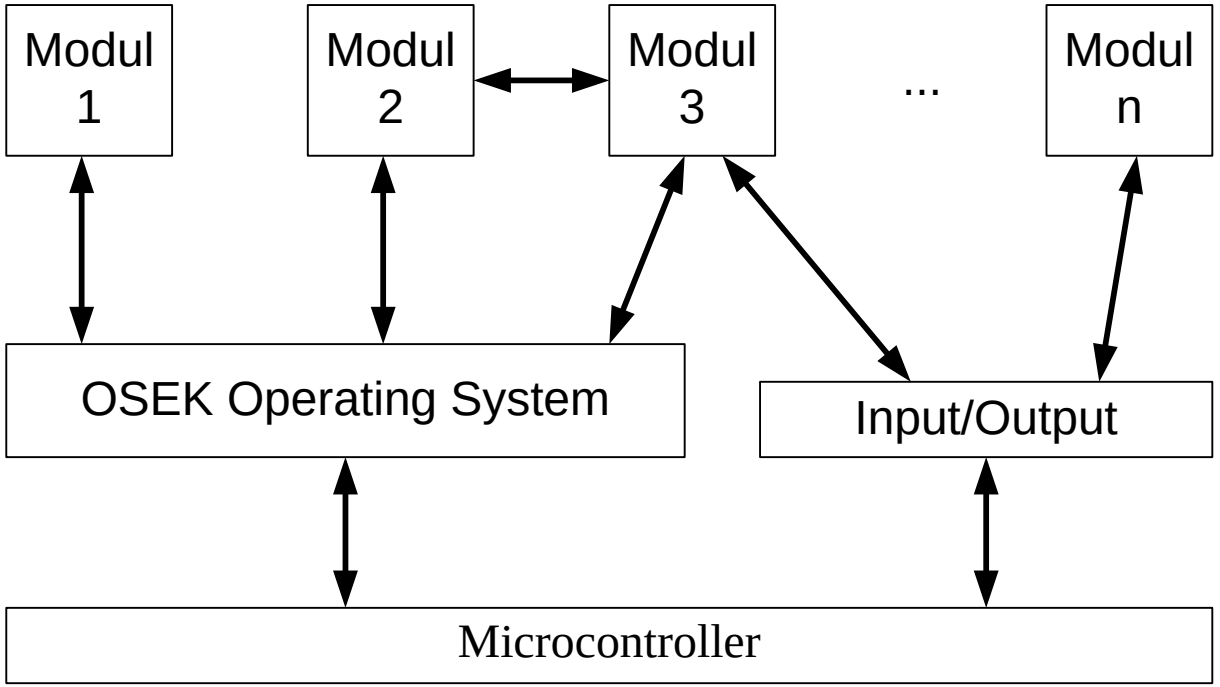
Anforderungen

- 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)

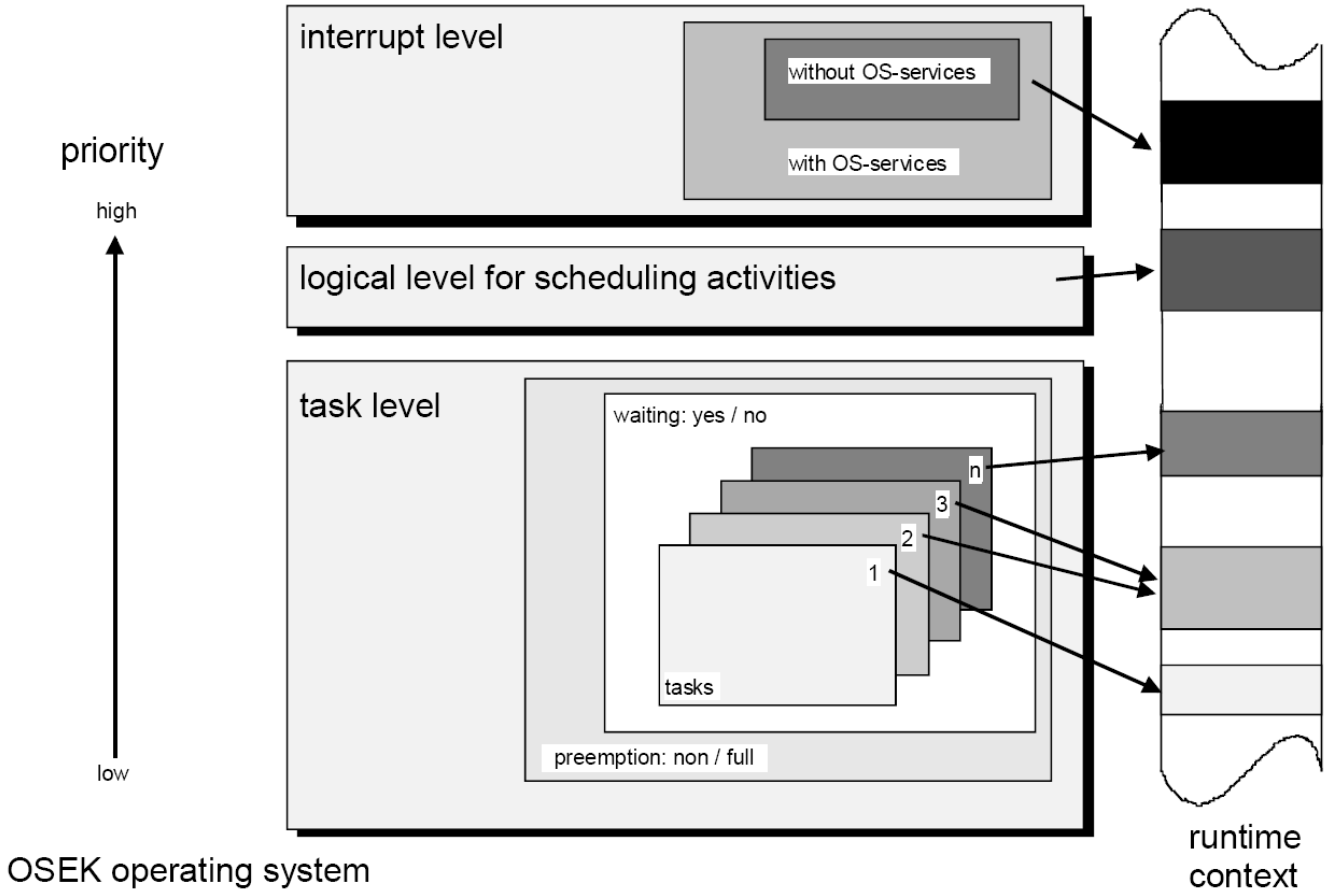
  → typische Anforderungen von Echtzeitsystemen
- Weitere Ziele:
  - Skalierbarkeit
  - einfache Konfigurierbarkeit des Betriebssystems
  - Portabilität der Software
  - Statisch allokiertes Betriebssystem

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

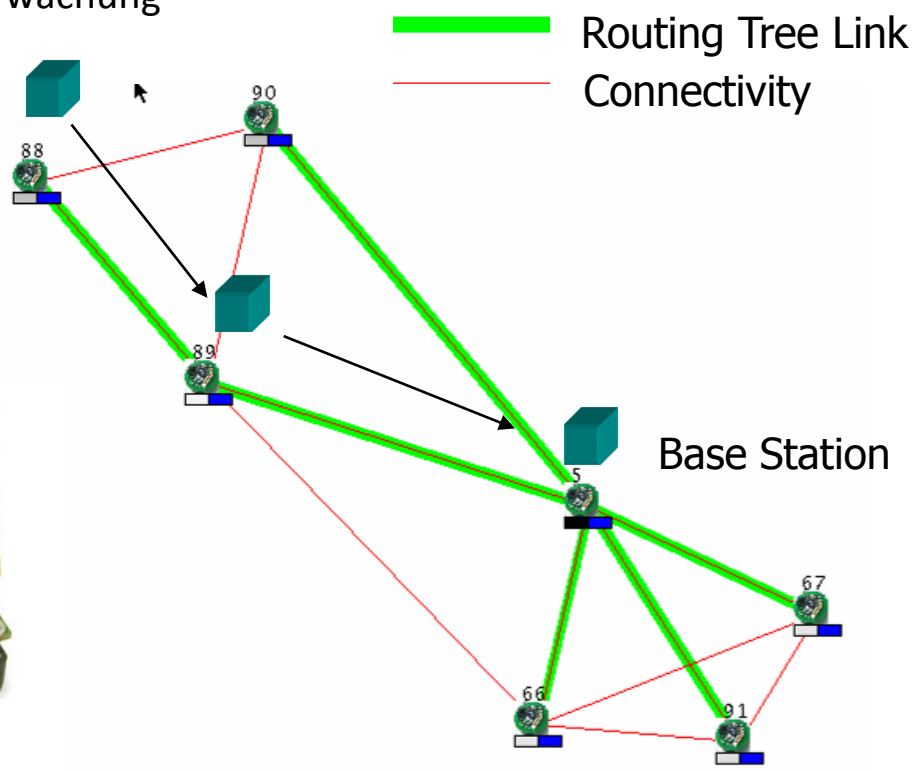
- Zur Vermeidung von Prioritätsinversion und Verklemmungen schreibt OSEK ein Immediate Priority Ceiling Protokoll vor:
  - Jeder Ressource wird eine Grenze (Maximum der Priorität der Prozesse, die die Ressource verwenden) zugewiesen.
  - Falls ein Prozess eine Ressource anfordert, wird die aktuelle Priorität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

Echtzeitbetriebssysteme

TinyOS

Einsatzgebiet: AdHoc-Sensornetzwerke

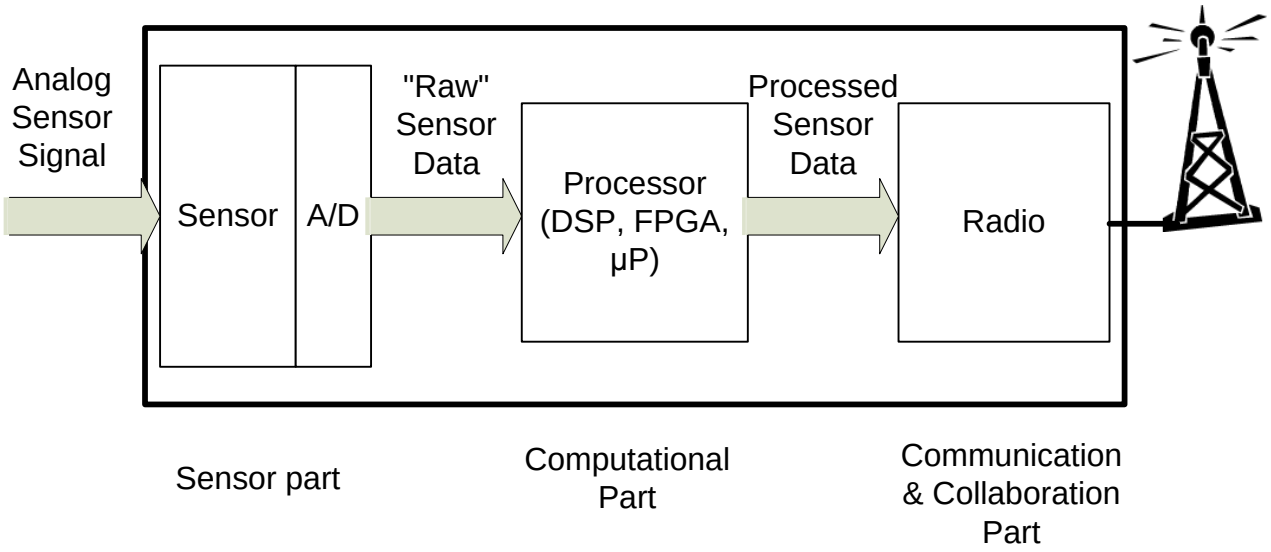
- Begriff Smart-Dust: Viele kleine Sensoren überwachen die Umgebung
- Ziele: robuste und flächendeckende Überwachung
- Probleme:
  - eingeschränkte Lebensdauer (Batterie)
  - eingeschränkter Speicherplatz
  - geringe Kommunikationsbandbreite
  - geringe Rechenleistung



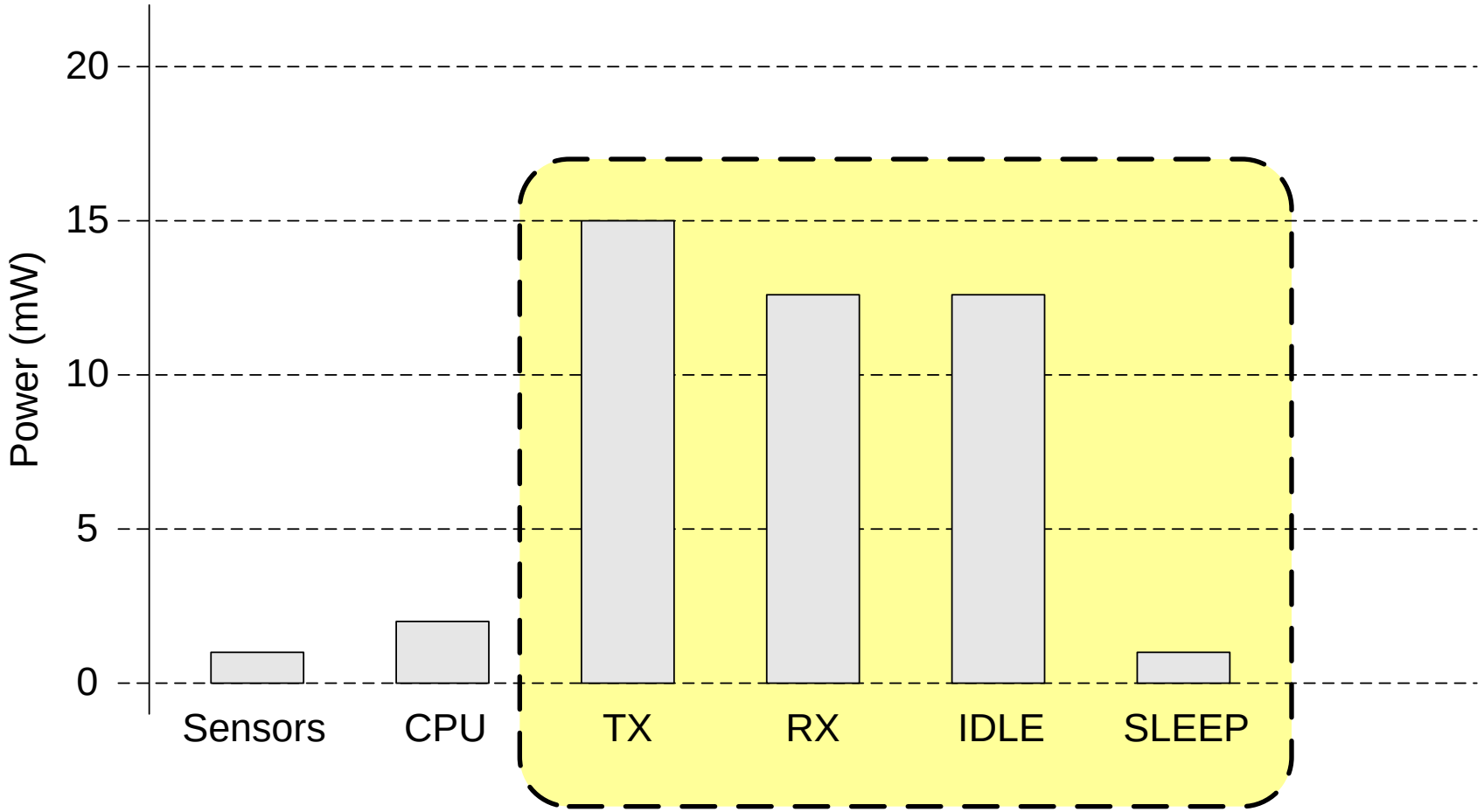
Quelle: http://tinyos.millennium.berkeley.edu

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

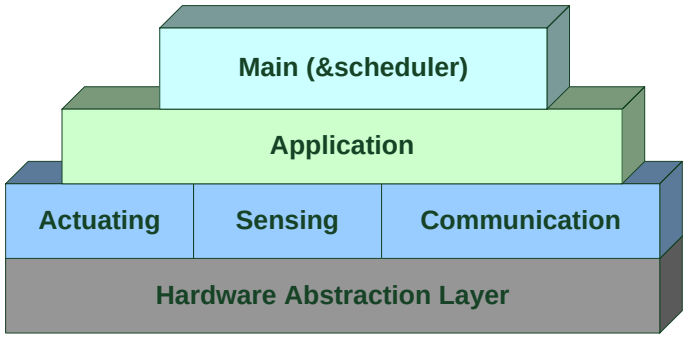


Stromverbrauch

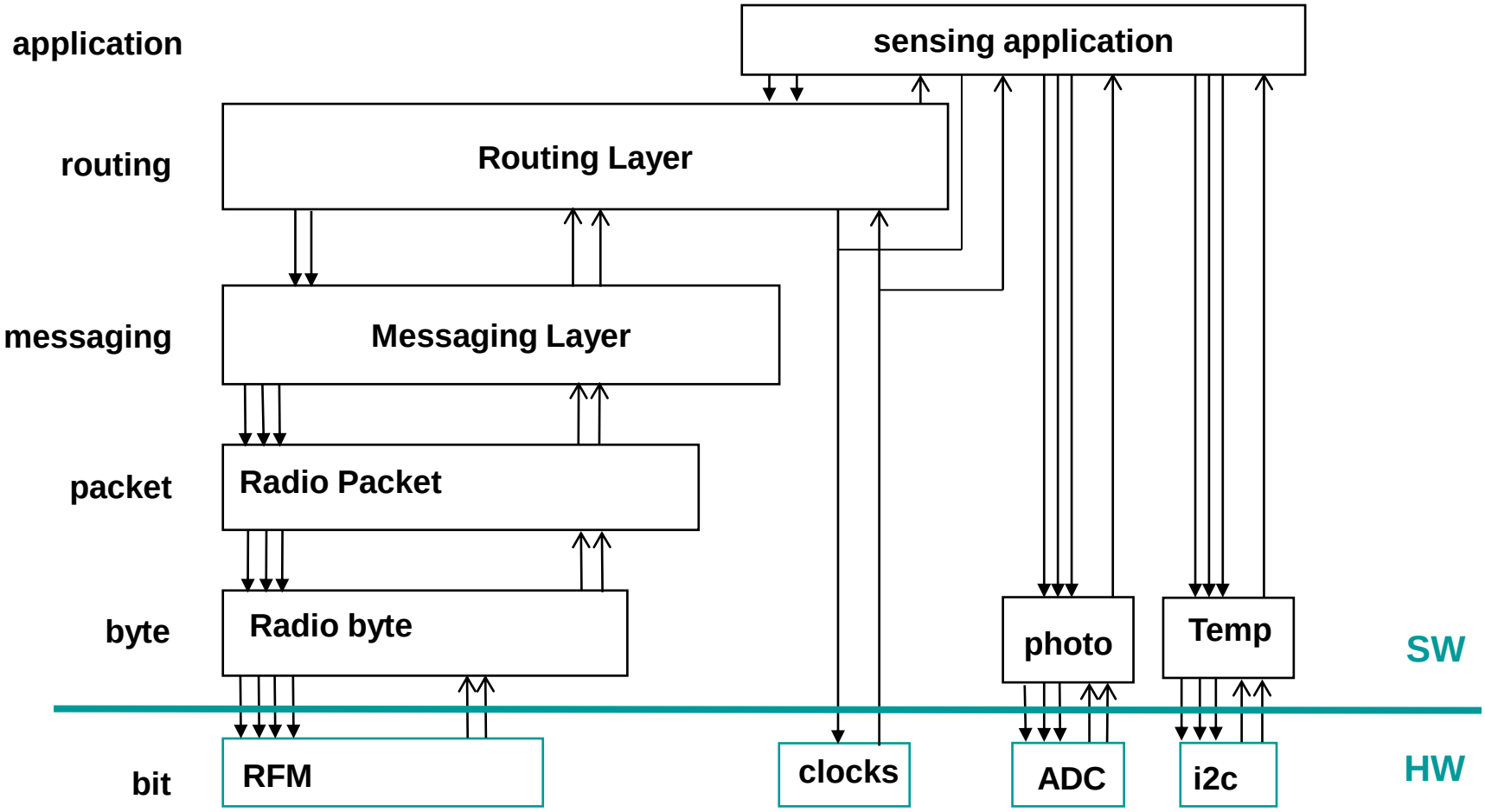


TinyOS

- 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



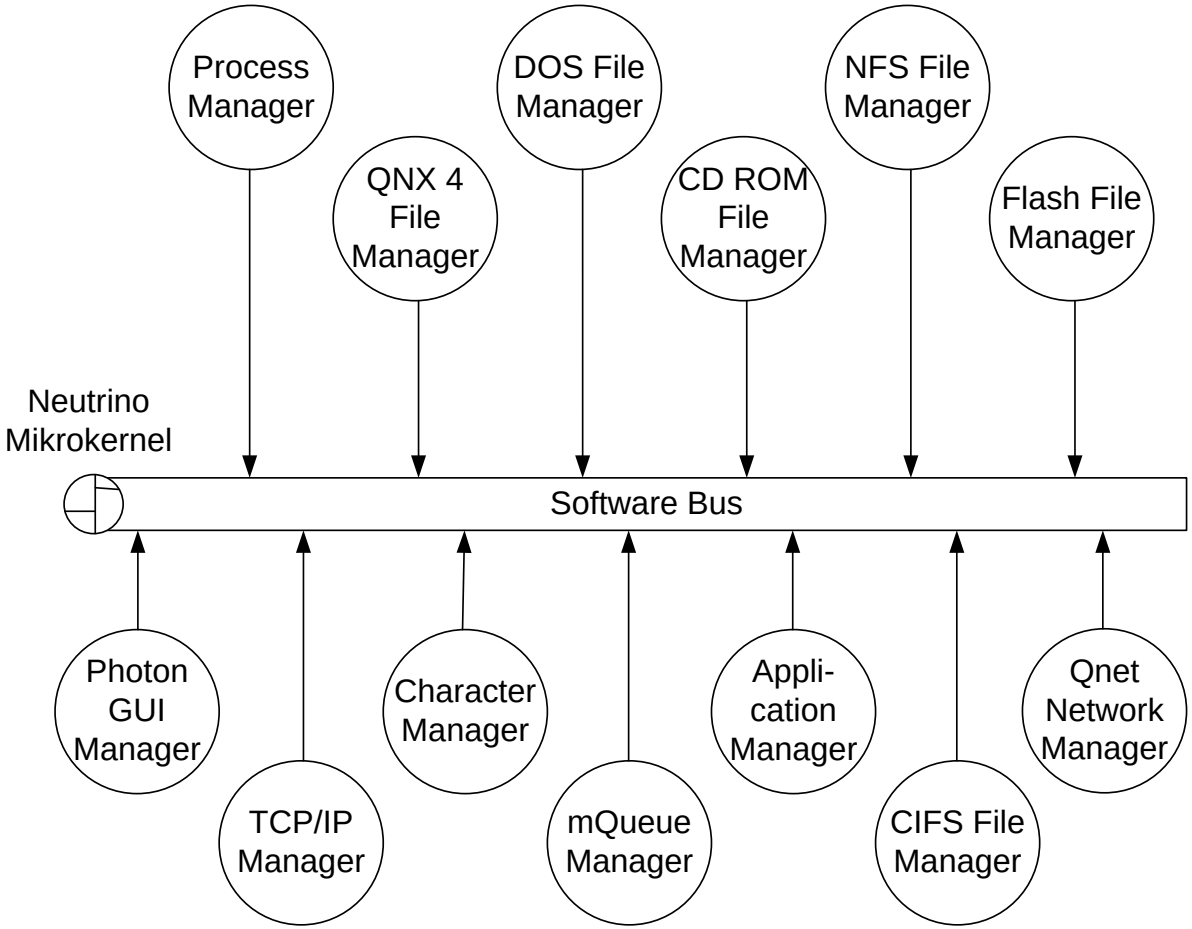
Echtzeitbetriebssysteme

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

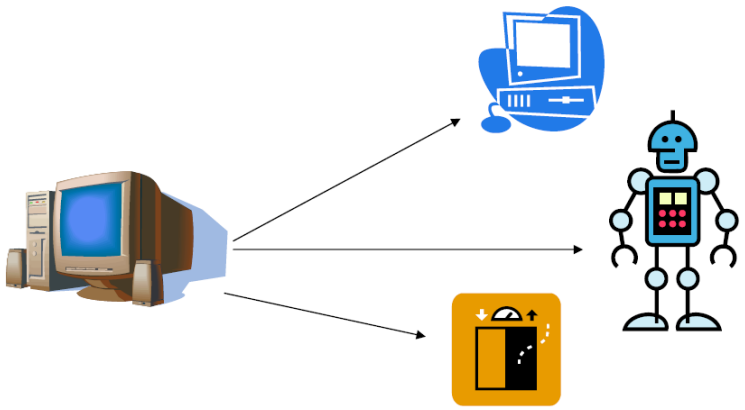
- Als wichtigster Prozess läuft in QNX der Prozessmanager.
- Die Aufgaben sind:
  - Prozessmanagement:
    - Erzeugen und Löschen von Prozessen
    - Verwaltung von Prozesseigenschaften
  - Speichermanagement:
    - Bereitstellung von Speicherschutzmechanismen,
    - von gemeinsamen Bibliotheken
   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

Echtzeitbetriebssysteme

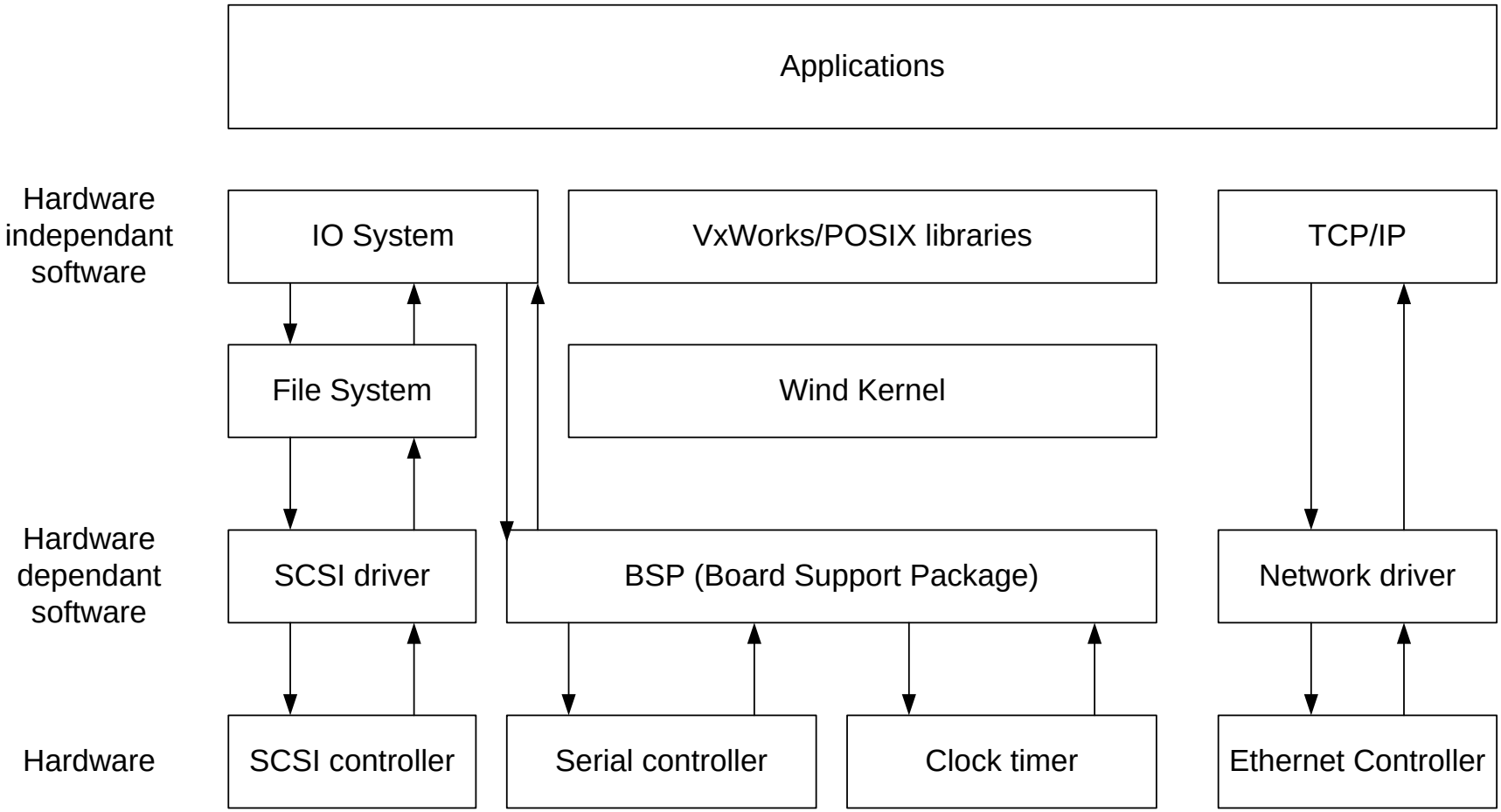
VxWorks

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, allerding 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.
- **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

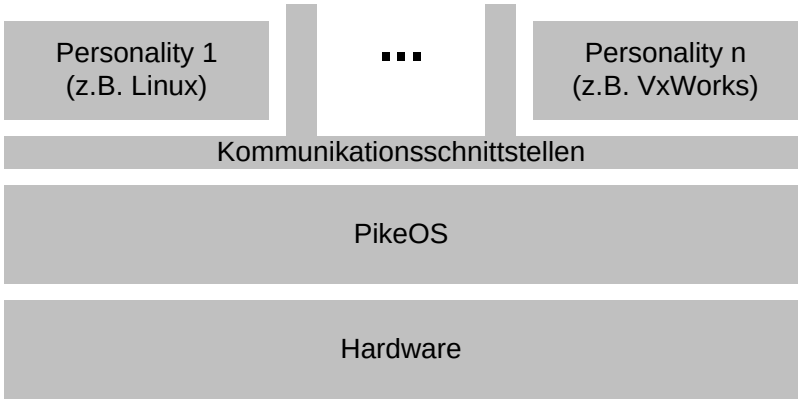
- Zur Interprozesskommunikation werden folgende Konzepte unterstützt:
  - Semaphor
  - Mutex (mit Prioritätsvererbung)
  - Nachrichtenwartschlangen
  - Signale
- Seit Version 6.0 wird zudem Speichermanagement angeboten:
  - Der Entwickler kann Benutzerprozesse mit eigenem Speicherraum entwickeln.
  -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



Architektur

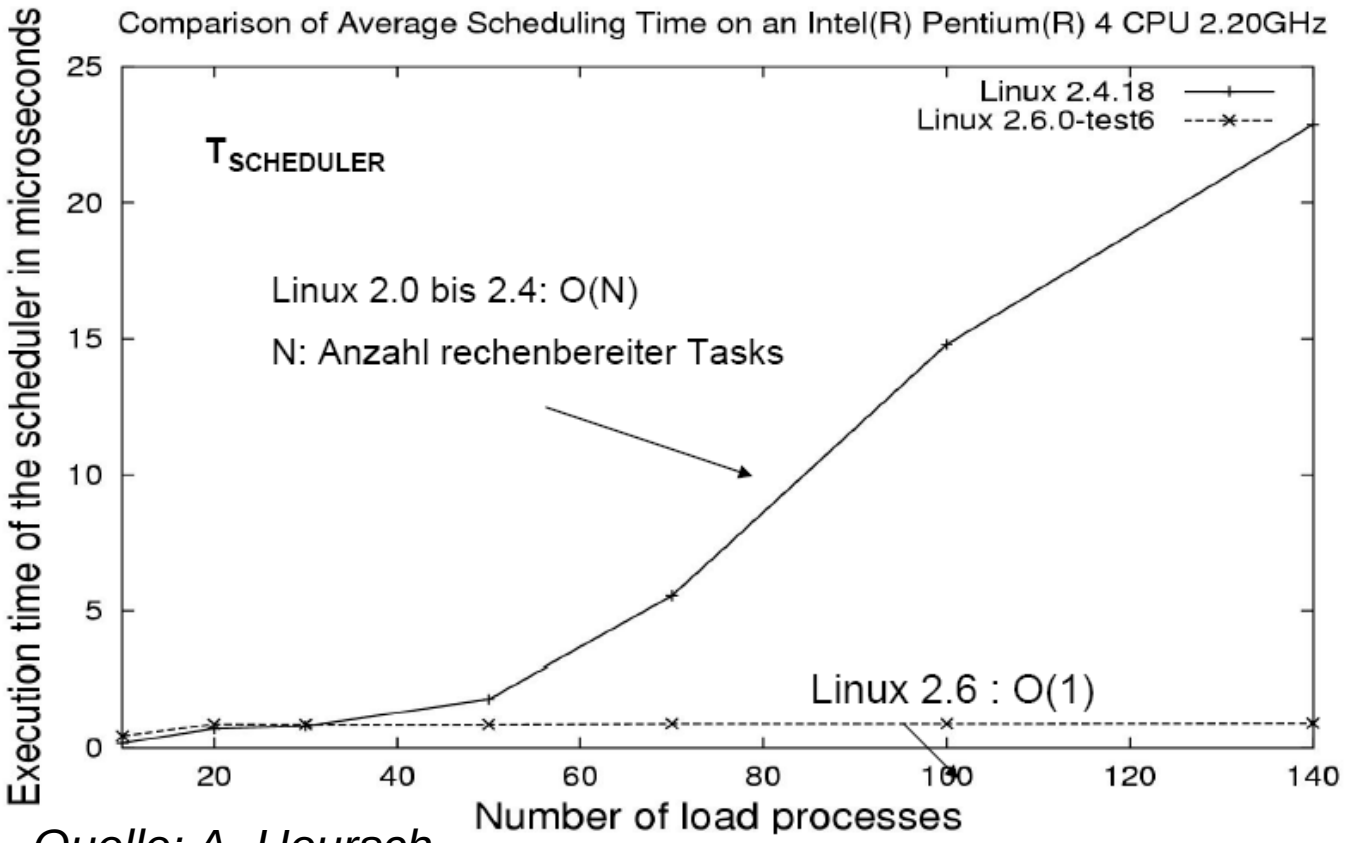
Echtzeitbetriebssysteme

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

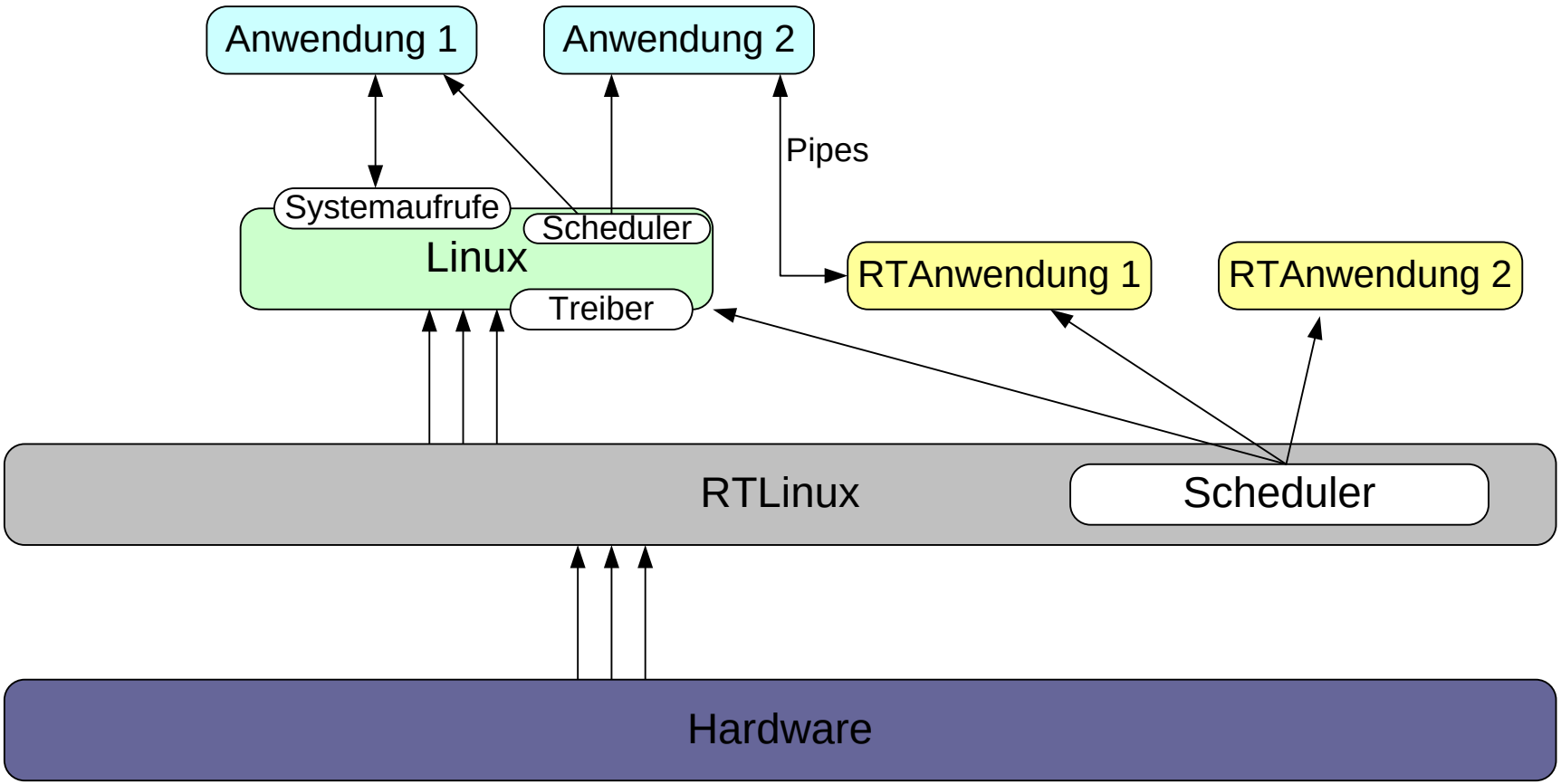
Motivation

- Aus diversen Gründen ist die Verwendung von Linux in Echtzeitsystemen erstrebenswert:
  - Linux ist weitverbreitet
  - Treiber sind sehr schnell verfügbar
  - Es existieren viele Entwicklungswerkzeuge → 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

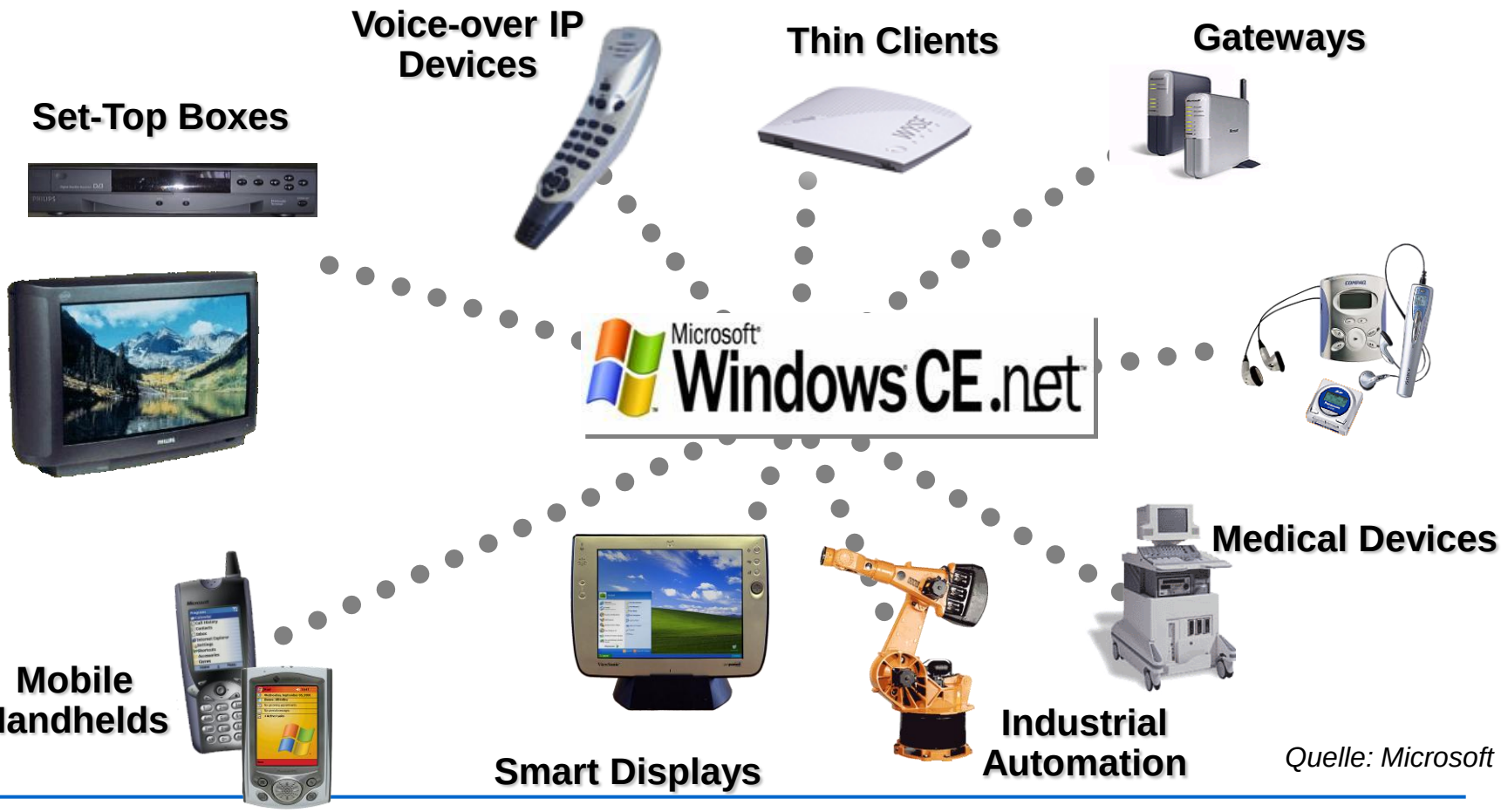
- Windows CE
  - 32-bit, Echtzeitbetriebssystem
  - Unterstützung von Multitasking
  - Stark modularer Aufbau
  - Skalierbar entsprechend der gewünschten Funktionalität
- Windows Embedded
  - „Skalierbares Windows XP“
  - Komponenten von XP können entfernt werden um den benötigten Speicherplatz zu minimieren

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

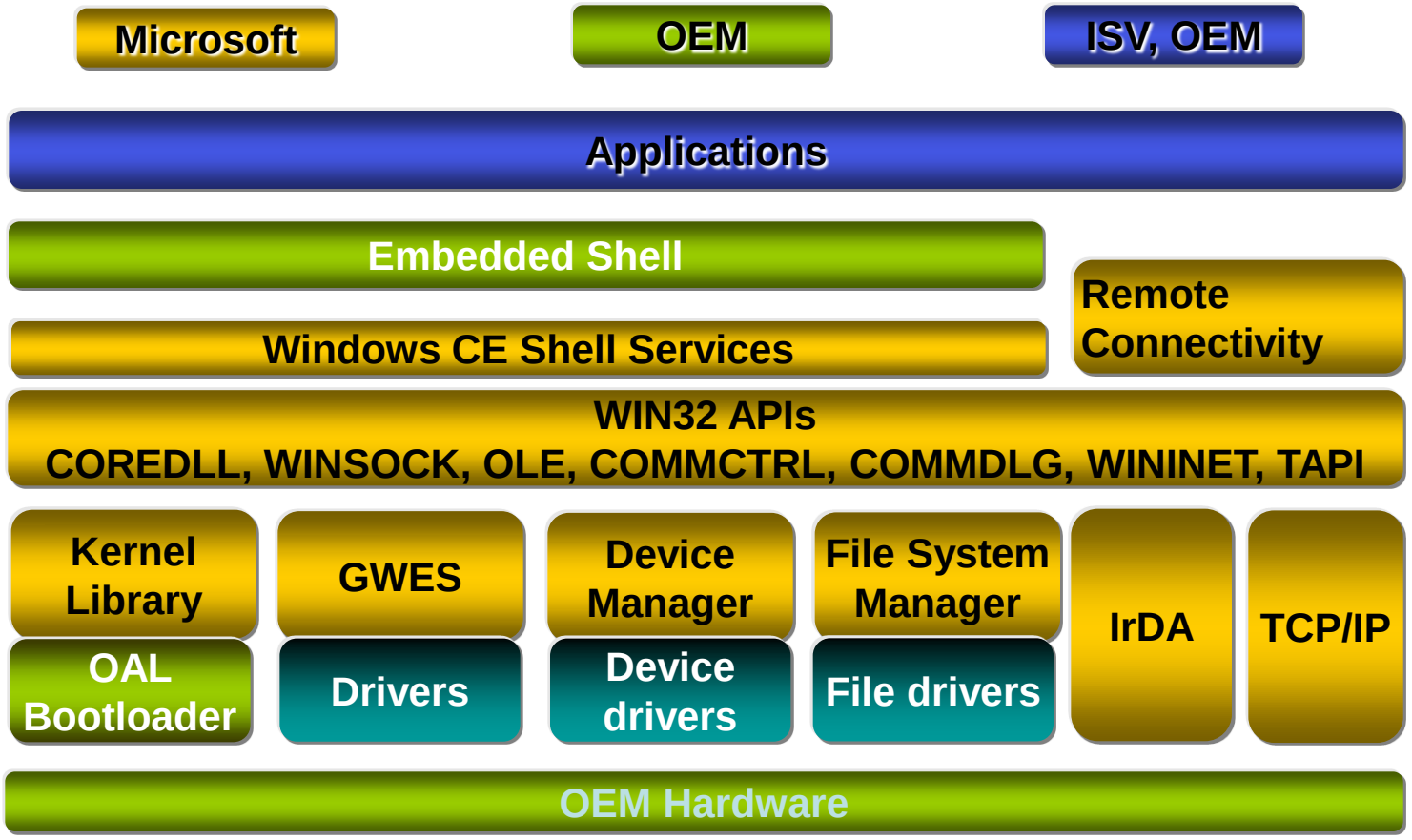
Quelle: Microsoft

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.

Erfolgskontrolle: Was Sie aus dem Kapitel mitgenommen haben sollten?

- Kenntniss der besonderen Anforderungen von Echtzeitsystemen an Betriebssysteme
- Wesentliche Konzepte der in der Vorlesung besprochenen Betriebssysteme
- Zuordnung von Anwendungsklassen zu den verschiedenen Betriebssystemen: „wann wurde man welches Betriebssystem einsetzen?“
- Möglichkeiten zum Einsatz von Standardbetriebssystemen

Klausurfragen

- Wiederholungsklausur WS 2006/2007 (5 Punkte = 5 min)
  - Erläutern Sie die Unterschiede zwischen Betriebssystemen mit kooperativem Scheduling, mit präemptiven Scheduling und präemptiblen Betriebssystemen.
  - Lösung:
    - Beim kooperativen Scheduling bekommt ein Prozess den Prozessor bis zu dem Zeitpunkt an dem dieser freiwillig freigegeben wird, der Prozessor kann nicht entzogen werden.
    - Beim präemptiven Scheduling kann der Prozessor dem Prozess entzogen werden, solange dieser nicht im Kernelbereich ausgeführt wird.
    - Beim präemptiblen Scheduling kann der Prozessor einem Prozess auch bei Ausführung im Kernelbereich entzogen werden.
- Klausur WS 2007/2008 (4 Punkte = 4 min)
  - Erläutern Sie kurz (jeweils 1-2 Sätze) die Hauptkonzepte von TinyOS, QNX und PikeOS.
  - Lösung: TinyOS ist für den Einsatz in ressourcenbeschränkten Systemen konzipiert und ist eher als Middleware anzusehen, die an die Anwendung angepasst werden kann. QNX hat einen sehr kleinen, modularen Kernel und benutzt zur Interaktion der Komponenten Nachrichten. PikeOS bietet eine Virtualisierung der Hardware an und erlaubt die Ausführung unterschiedlicherBetriebssysteme mit klaren Speicher- und Zeitzuteilungen auf einer CPU.

Klausurfragen - Klausur WS 2006/2007 (10 Punkte = 10 min)

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.

Klausurfragen - Klausur WS 2006/2007 (10 Punkte = 10 min), Lösung

Aufgabe 5 Echtzeitbetriebssysteme **(10 Punkte)**

- TinyOS - ZebraNet: Auslegung auf Sensornetzwerke mit limitierter Stromversorgung.
- OsekTime - Zugsteuerung: Zeitgesteuertes Betriebssystem und damit klare Trennung einzelner Softwarekomponenten.
- QNX - Sicherheitsüberwachung: Hartes, stark skalierbares (kleiner Kernel) Echtzeitbetriebssystem.
- VxWorks - Fabrikanlagensteuerung: Hartes Echtzeitbetriebssystem mit guter Unterstützung unterschiedlichster Sensorik.
- WindowsCE - PDA: weiches Echtzeitbetriebssystem mit Schwerpunkt auf Benutzerfreundlichkeit (z.B. GUI), Kompatibilität mit Anwendungen von Standardbetriebssystemen

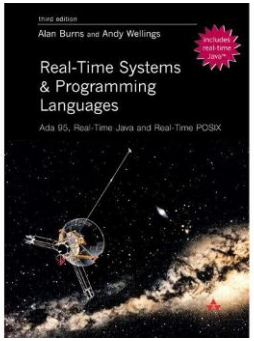
Kapitel 8

Programmiersprachen für Echtzeitsysteme

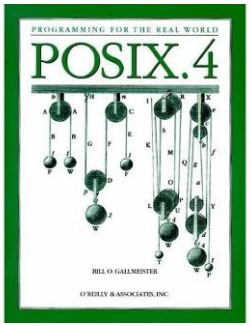
Inhalt

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

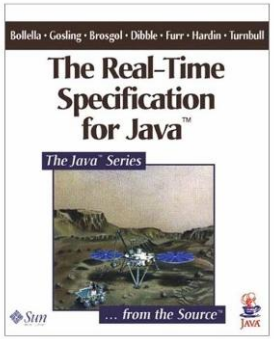
Literatur



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



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



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

Paper:

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

Programmiersprachen für Echtzeitsysteme

Anforderungen

Anforderungen

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

Beherrschung komplexer nebenläufiger Systeme

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

Einhalten zeitlicher Anforderungen

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

Unterstützung hardwarenaher Programmierung

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

Erfüllung hoher Sicherheitsanforderungen

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

Sicherheit fängt schon im Kleinen an

- Lexikalische Konventionen können Fehler verhindern.
- Negatives Beispiel: FORTRAN
  - In FORTRAN werden Leerzeichen bei Namen ignoriert.
  - Variablen müssen in FORTRAN nicht explizit definiert werden

- Problem in Mariner 1:
  Aus einer Schleife

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

  wird durch versehentliche Verwendung eines Punktes

```
DO5K=1.3
```

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

Anforderungen durch verteilte Systeme:

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

Programmiersprachen für Echtzeitsysteme

Geschichte

Geschichte: 1960-1970

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

Geschichte: 1970-1980

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

Geschichte: 1970-1980

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

Geschichte ab 1978

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

Geschichte heute:

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

Programmiersprachen für Echtzeitsysteme

PEARL

Daten

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

Eigenschaften

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

Erläuterung für folgendes Beispiel

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

Grundstruktur

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

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

Datentypen

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

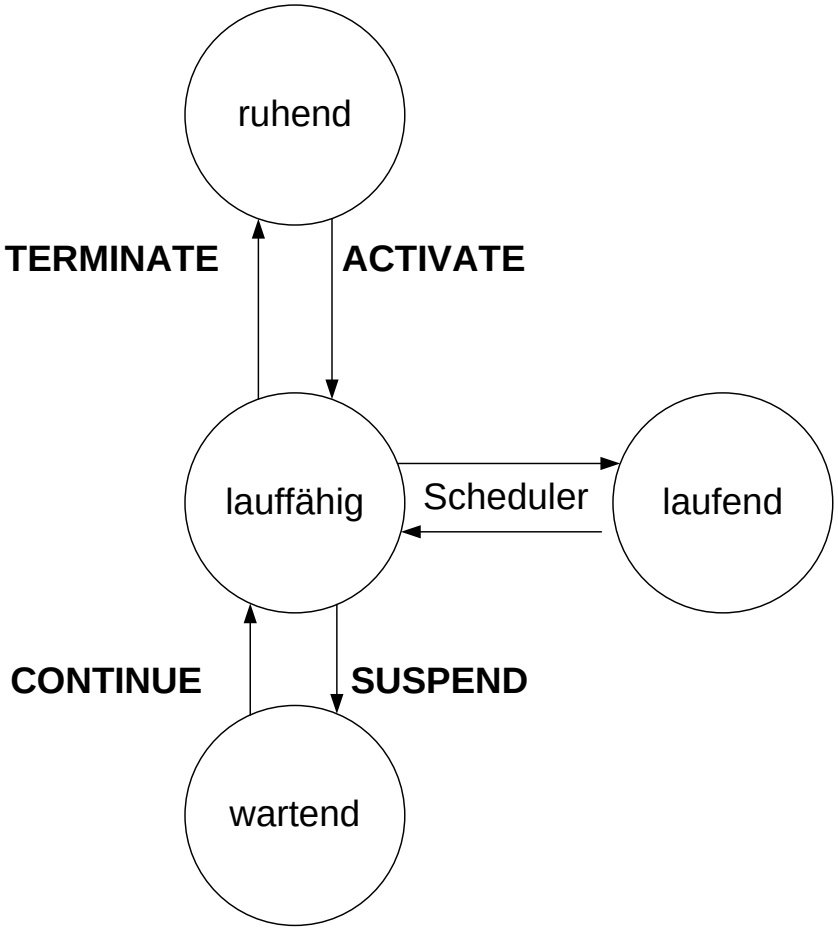
Schlüsselwort	Bedeutung	Beispiel
FIXED	Ganzzahlige Variable	-2
FLOAT	Gleitkommazahl	0.23E-3
CLOCK	Zeitpunkt	10:44:23.142
DURATION	Zeitdauer	2 HRS 31 MIN 2.346 SEC
CHAR	Folge von Bytes	'Hallo'
BIT	Folge von Bits	'1101'B1

Prozessmodell

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

```
AFTER 5 SEC ACTIVATE Task1;

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

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



Prozess-Synchronisation

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

Lösung: Leser-Schreiber-Problem mit Schreiberpriorisierung

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

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

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

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

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

Unterbrechungen

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

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

   PROBLEM;
        SPECIFY alarm INTERRUPT;

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

Programmiersprachen für Echtzeitsysteme

Ada

Einleitung

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

Eigenschaften

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

Strukturierung

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

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


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

Generische Einheiten

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

Spezifikation:

```
GENERIC
   TYPE sometype IS PRIVATE;

PACKAGE queue_handling IS
   TYPE queue (maxlength: NATURAL)
   IS PRIVATE;

PROCEDURE enqueue (q: IN OUT queue;
   elem: IN sometype);
   ...

PROCEDURE dequeue ...

PRIVATE
   SUBTYPE index IS CARDINAL RANGE
        0..1000;
   …
```

Benutzung:

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

Prozesse

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

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


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

Lebenszyklus eines Prozesses

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

Partitionen

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

Prozess-Synchronisation: Rendezvous-Konzept

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

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

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

```
GENERIC
        TYPE item IS PRIVATE

PACKAGE sharedmemory IS
        PROCEDURE readProc(x: OUT item)
        PROCEDURE writeProc(x: IN item)
END;

PACKAGE BODY sharedmemory IS
        value: item;

  TASK control IS
        ... (siehe folgende Folien)
  END control;
  PROCEDURE readProc(x:OUT item) IS
  BEGIN
        ... (siehe folgende Folien)
  PROCEDURE writeProc(x:IN item) IS
  BEGIN
        ... (siehe folgende Folien)
END sharedmemory;
```

Beispiel: Fortsetzung

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

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

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

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

Beispiel (Fortsetzung): Code des Prozesses `control`

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

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

Wechselseitiger Ausschluss

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

→ Vorgehen ähnelt Monitoren

Beispiel: Realisierung eines Semaphors

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


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

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

Benutzung:

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

Ausnahmen

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

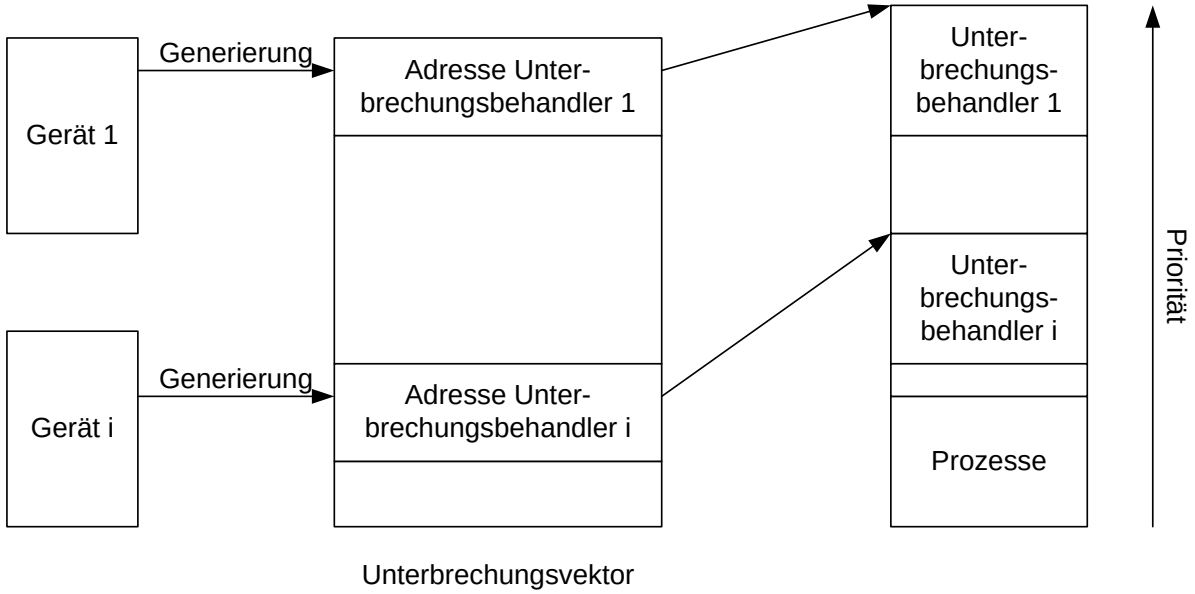
- Syntax der Behandlung

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

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

Unterbrechungen

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



Unterbrechungen: statisch vs. dynamisch

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

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

Programmiersprachen für Echtzeitsysteme

Real-Time Java

Motivation

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

→ RTSJ (Real-Time Specification for Java) **erweitert** Java:

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

Design-Prinzipien RTSJ

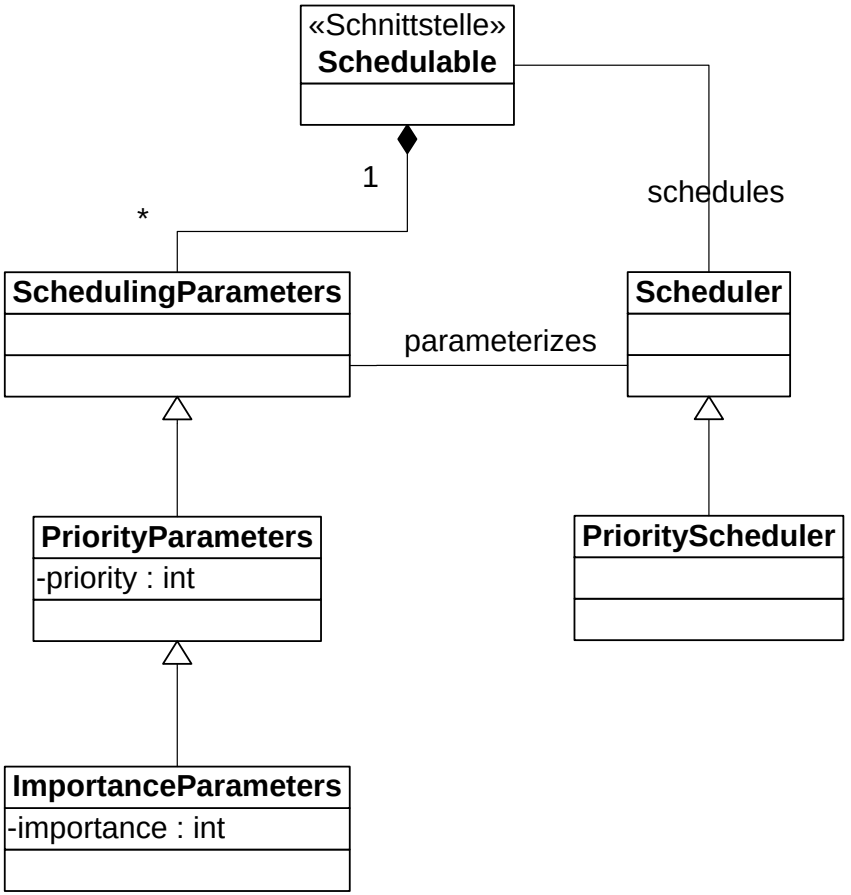
1. Keine Einschränkungen auf bestimmte Java-Umgebung (z.B. eine bestimmte Version von JDK)
2. Rückwärtskompatibilität
3. „Write once carefully, run anywhere conditionally“
4. Unterstützung aktueller Entwicklungsprozesse für Echtzeitsysteme
5. (Zeitlich) vorhersagbare Ausführung
6. Keine syntaktischen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

Scheduling in Java

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

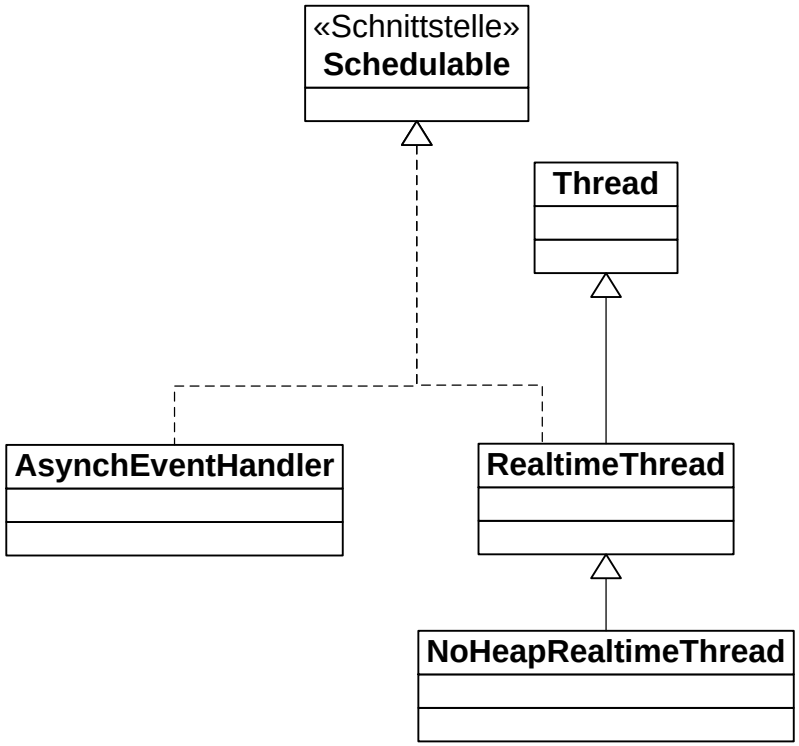
Scheduling in RTJS

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



Threads in RTJS

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



Speichermanagement in Java

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

Speichermanagement in RTSJ (1)

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

Speichermanagement in RTSJ (2)

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

Weitere Ergänzungen

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

Programmiersprachen für Echtzeitsysteme

Zusammenfassung

Was Sie aus diesem Kapitel mitgenommen haben sollten

- Kriterien bei der Auswahl der Sprache sind:
  - Sicherheit
  - Komfort bei der Entwicklung (v.a. Existenz geeigneter Entwicklungswerkzeuge)
  - projektierbares Zeitverhalten
  - Möglichkeit zur hardwarenahen, nebenläufigen Programmierung
  - Portabilität
- Zur Sicherheit tragen eine strenge Typisierung und Prüfungen zur Laufzeit bei.
- Zur Erhöhung der Portabilität werden hardwareabhängige und –unabhängige Codeteile häufig getrennt.
- Mechanismen zum Prozessmanagement / -synchronisation und dem Umgang mit Zeit erleichtern die Programmierung von Echtzeitsystemen
- Virtuelle Laufzeitumgebungen (wie z.B. Virtual Machine in RTSJ) eignen sich nur bedingt zur Verwendung in Echtzeitsystemen
- Kenntnis über dieKonzepte der besprochenen Programmiersprachen, die über Standardsprachen wie C bzw. JAVA hinausgehen
- Trotz der Existenz vieler komfortabler und sicherer Programmiersprachen für Echtzeitsysteme wird der überwiegende Teil der Echtzeitsysteme in C (plus POSIX) entwickelt.

Klausur WS 07/08 (8 Punkte)

- Erläutern Sie die Konzepte von Ada zur Unterstützung von Unterbrechungen (Interrupts) und Prozesssynchronisation.
- Erläutern Sie die Konzepte zur Speicherverwaltung in Real-Time Java.
- Antwort: siehe Folien

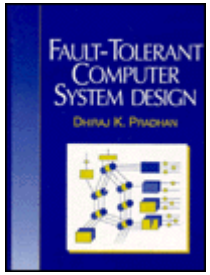
Kapitel 9

Entwicklung von sicherheitskritischen Systemen

Inhalt

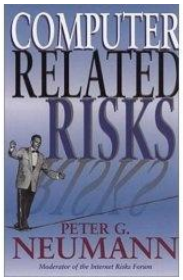
- Einleitung / Motivation / Definitionen
- Zertifizierungsstandards
- Sicherheitsanalyse & ASIL-Einstufung
- Analyse der möglichen Fehler
- Fehlererkennung
- Fehlertoleranzmechanismen
- Anforderungen an die Softwareentwicklung

Literatur

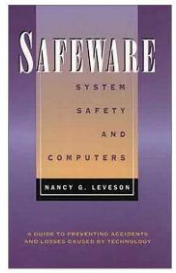


Dhiraj K. Pradhan: Fault-Tolerant Computer System Design, Prentice Hall 1996

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



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



Nancy G.Leveson: Safeware, Addison-Wesley 1995

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

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

Entwicklung sicherheitskritischer Systeme

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)

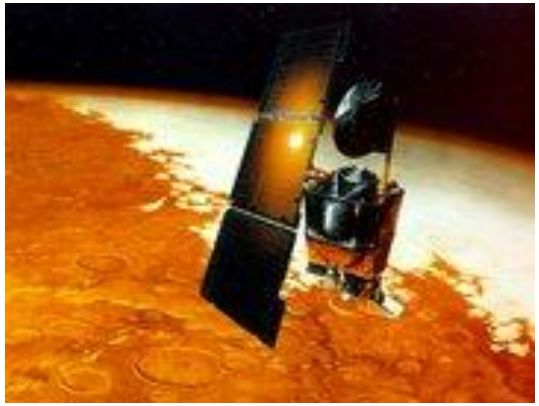
Weitere Informationen unter
http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html

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.

Entwicklung sicherheitskritischer Systeme

Definitionen

Definitionen

- Sicherheit = Freiheit von unvertretbaren Risiken
- Risiko = Kombination aus der Wahrscheinlichkeit, mit der ein Schaden auftritt, und dem Ausmaß dieses Schadens

vs.

- Mechanismen zur Risikominimierung:

Fehlervermeidung

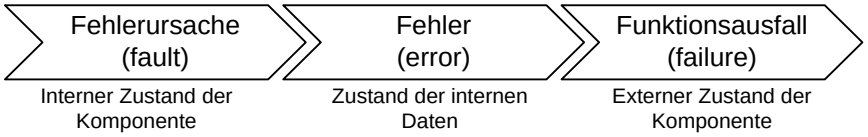
Fehlertoleranz durch Redundanz

Schadensminimierung

- Redundanz bezeichnet den Einsatz von mehr technischen Mitteln als für die spezifizierte Nutzfunktion eines Systems benötigt werden.
- Offene Frage: Wie definiert sich unvertretbar? → siehe Zertifizierungsstandards

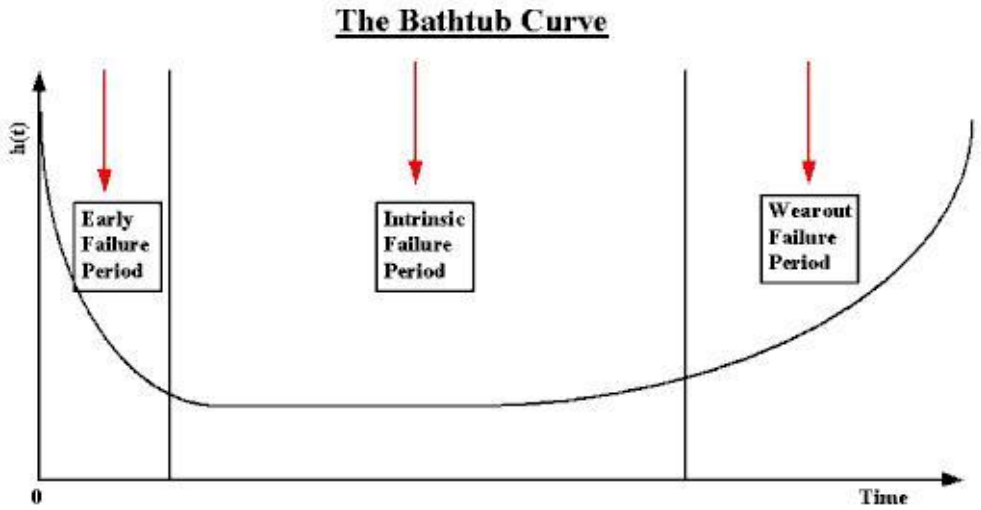
Begriff Fehler

- **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.
- Sicherheit und Zuverlässigkeit sind somit gegensätzliche Ziele:
  - Beispiel: Ampel, die keine Signale abgibt ist sicher, da die Autofahrer die Verkehrszeichen beachten, aber nicht zuverlässig

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 Verfügbarkeit, 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.

Entwicklung sicherheitskritischer Systeme

Zertifizierungsstandards

Zertifizierungsstandards

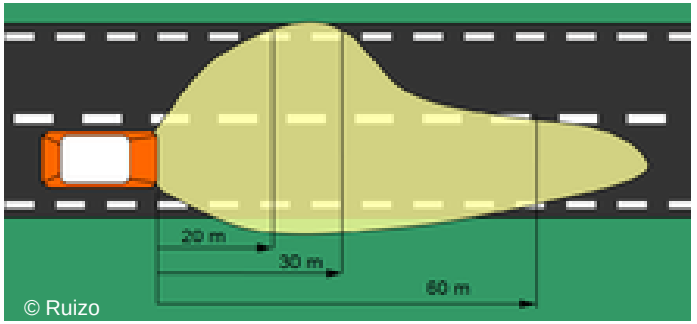
- Gründe für Entwicklung gemäß Zertifizierungsstandards:
  - Produkthaftung
  - Gesetzliche Anforderungen an die Zulassung
- Haftungsgesetz: ein Unternehmen haftet, wenn es zu Fehlern kommt, weil das Unternehmen nicht entsprechend dem Stand der Technik und der Wissenschaft entwickelt hat
- Zertifizierungsstandards sind Richtlinien für einen solchen Stand der Technik, sie definieren auch das akzeptable Risiko
- Anwendung von Zertifizierungsstandards ist notwendig, aber nicht hinreichend
- In manchen Industriebereichen wird die Standardeinhaltung zwingend gefordert (Nachweis durch Zertifizierungsbehörde z.B. TÜV)
- Abweichungen vom Standard sind immer möglich, sie müssen nur gut begründet sein

Zertifizierungsstandards im Überblick (Auswahl)

- IEC 61508 „Funktionale Sicherheit sicherheitsbezogener elektrischer/elektronischer/programmierbarer elektronischer Systeme“ ➔ Domänenübergreifend, aber vor allem Automatisierungsdomäne
- Anpassungen der IEC 61508 an bestimmte Domänen:
  - IEC 61513: Kernkraftwerke — Leittechnik für Systeme mit sicherheitstechnischer Bedeutung – Allgemeine Systemanforderungen
  - EN 50128: Bahnanwendungen — Telekommunikationstechnik, Signaltechnik und Datenverarbeitungssysteme - Sicherheitsrelevante elektronische Systeme für Signaltechnik
  - ISO 26262: Road vehicles – Functional safety
- Im Luftfahrbereich gilt für die Softwareentwicklung der Standard DO-178C

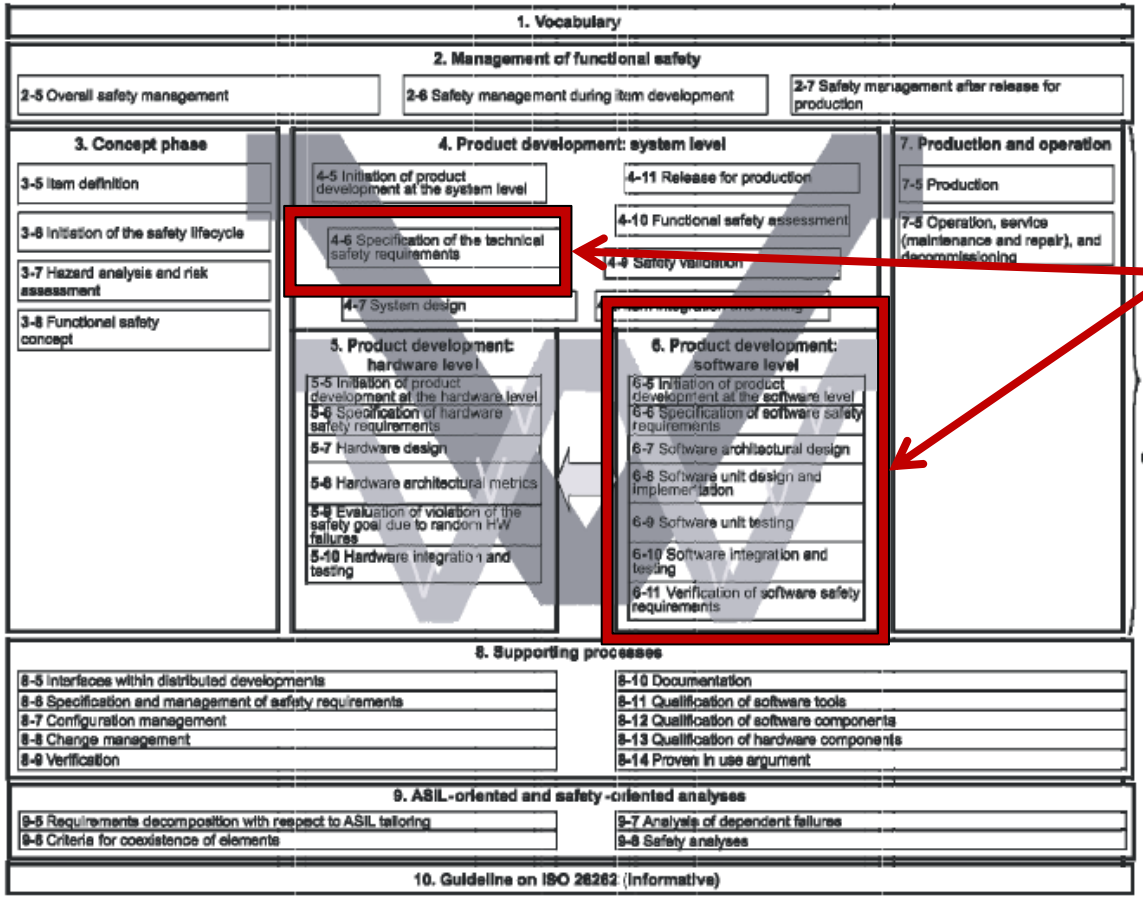
Laufendes Beispiel – Lichtmanagement im Fahrzeug

- Entwickelt werden soll die Ansteuerung des Abblendlichtes im Fahrzeug



- Mögliche Fehlfunktionen (vereinfacht):
  - Licht schaltet sich trotz Betätigung des Lichtschalters nicht ein
  - Licht schaltet sich plötzlich ohne Benutzerinteraktion aus

Für Beispiel relevanter Standard: ISO 26262



Schwerpunkte dieser Vorlesung (Softwareentwicklung)

Nicht behandelt werden:
-Management der funktionalen Sicherheit (Organisationsbezogene Massnahmen)
- Sicherheitslebenszyklus: Fokus Vorlesung auf Entwicklung
- Dokumentation

Entwicklung sicherheitskritischer Systeme

Sicherheitsanalyse & ASIL-Einstufung

ASIL-Einstufung

- Sicherheitskritikalitätslevel:
  - Der Automotive Safety Integrity Level gibt eine Einschätzung ab, wie kritisch eine Funktion einzustufen ist
  - Die niedrigste Stufe ist QM (Qualitätsmanagement), die höchste ASIL D
  - Es wird zwischen zwei Betriebsarten unterschieden: Betriebsart mit niedrigen Anforderungsraten (z.B. Auslösen des Airbags bei Unfall) bzw. Betriebsart mit hoher oder kontinuierlicher Anforderungsrate (z.B. Nichtauslösen des Airbags bei normaler Fahrt)
- Einstufung:
  - Im ISO 26262 Standard wird die Einstufung anhand von drei Parametern vorgenommen:
    - Faktor S (Severity): mögliches Schadensausmaß im Fall eines Systemversagens
    - Faktor E (Exposure): Aufenthaltsdauer / -wahrscheinlichkeit in der gefährlichen Fahrsituation
    - Faktor C (Controllability): Kontrollierbarkeit der gefährlichen Fahrsituation
  - Die Einstufung muss für jede Art der Fehlfunktion durchgeführt werden, dabei werden typischerweise verschiedenen Szenarien durchgespielt

ASIL-Einstufung - Konsequenzen

- Konsequenzen der Einstufung:
  - Auf Basis der Einstufung werden quantitative Anforderungen (Grenzwerte für verschiedene Parameter des Systems), sowie qualitative Anforderungen an den Entwicklungsprozess vorgegeben
- Ausfallsgrenzwert - Probability of Failure on Demand / per Hour (PFD/PFH)
  - Der ASIL gibt vor wie wahrscheinlich ein Ausfall pro Betriebsstunde höchstens sein darf (z.B. $<10^{-8}$ Ausfälle/Betriebsstunde bei ASIL D und kontinuierlichen Betrieb)
    Einstufung:
- Grenzwert für den minimalen Anteil der sicher erkannten Ausfälle (Safe Failure Fraction - SFF)
  - Die SFF ergibt sich aus der gesamten sicheren Fehlerrate (sichere Ausfälle und erkannte gefährliche Ausfälle) geteilt durch die gesamte Fehlerrate (gesamte sichere Fehlerrate plus unerkannte gefährliche Ausfälle)
  - Wichtig: Die Berechnung erfolgt auf Teilsystemebene und das Teilsystem mit dem niedrigsten erreichbaren SIL bestimmt den erreichbaren SIL der gesamten Sicherheitsfunktion
  - Die vorgegebenen Grenzwerte richten sich auch nach dem Redundanzgrad, sowie der Komplexität der Hardwarekomponenten (z.B. mind. 90% bei einfachen Teilsystemen mit Redundanzgrad 2 für SIL 4 in der IEC 61508)

Parameter S: Potentielles Schadensmaß

Table 1 — Classes of severity

Class	S0	S1	S2	S3
Description	No injuries	Light and moderate injuries	Severe and life-threatening injuries (survival probable)	Life-threatening injuries (survival uncertain), fatal injuries

- Beispiele für verschiedene Klassen:
  - S1: Oberflächliche Wunden bis hin zu Gehirnerschütterung mit Bewusstlosigkeit bis zu 15 Minuten
  - S2: Schädelbrüche ohne Gehirnverletzung
  - S3: Wirbelsäulenfrakturen
- Der Standard berücksichtigt auch die Verteilung der möglichen Verletzungen auf die einzelnen Klassen
- Einstufung im laufenden Beispiel (kein Gewähr):
  - Kein Licht trotz Einschalten:
    - S3
  - Lichtausfall:
    - S3

Parameter E: Aufenthaltsdauer

Table B.2 — Classes of probability of exposure regarding duration/probability of exposure in driving situations

Class	E1	E2	E3	E4
Description	Very low probability	Low probability	Medium probability	High probability
Definition of duration/ probability of exposure	Not specified	< 1% of average operating time	1% - 10% of average operating time	> 10% of average operating time

- Beispiele für verschiedene Klassen:
  - E1: Verlorenes Gepäck auf der Autobahn
  - E2: Anhängerbetrieb
  - E3: Tunnelfahrt
  - E4: Beschleunigen
- Einstufung im laufenden Beispiel (kein Gewähr):
  - Kein Licht trotz Einschalten:
    - E3
  - Lichtausfall:
    - E3

Parameter C: Beherrschbarkeit

Table B.4 — Examples of possibly controllable hazards by the driver or by the endangered persons

Class	C0	C1	C2	C3
Description	Controllable in general	Simply controllable	Normally controllable	Difficult to control or uncontrollable
Definition	Controllable in general	99% or more of all drivers or other traffic participants are usually able to avoid a specific harm.	90% or more of all drivers or other traffic participants are usually able to avoid a specific harm.	Less than 90% of all drivers or other traffic participants are usually able, or barely able, to avoid a specific harm.

- Beispiele für verschiedene Klassen:
  - C0: Unerwartete Lautstärkesteigerung des Radios
  - C1: Anfahren mit verriegeltem Lenkradschloss
  - C2: Ausfall des ABS während einer Notbremsung
  - C3: Fehlverhalten der Lenkung bei mittlerer oder hoher Geschwindigkeit
- Einstufung im laufenden Beispiel (kein Gewähr):
  - Kein Licht trotz Einschalten:
    - C1
  - Lichtausfall:
    - C2

ASIL-Einstufung – Kein Gewähr

		C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	E1	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	E1	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

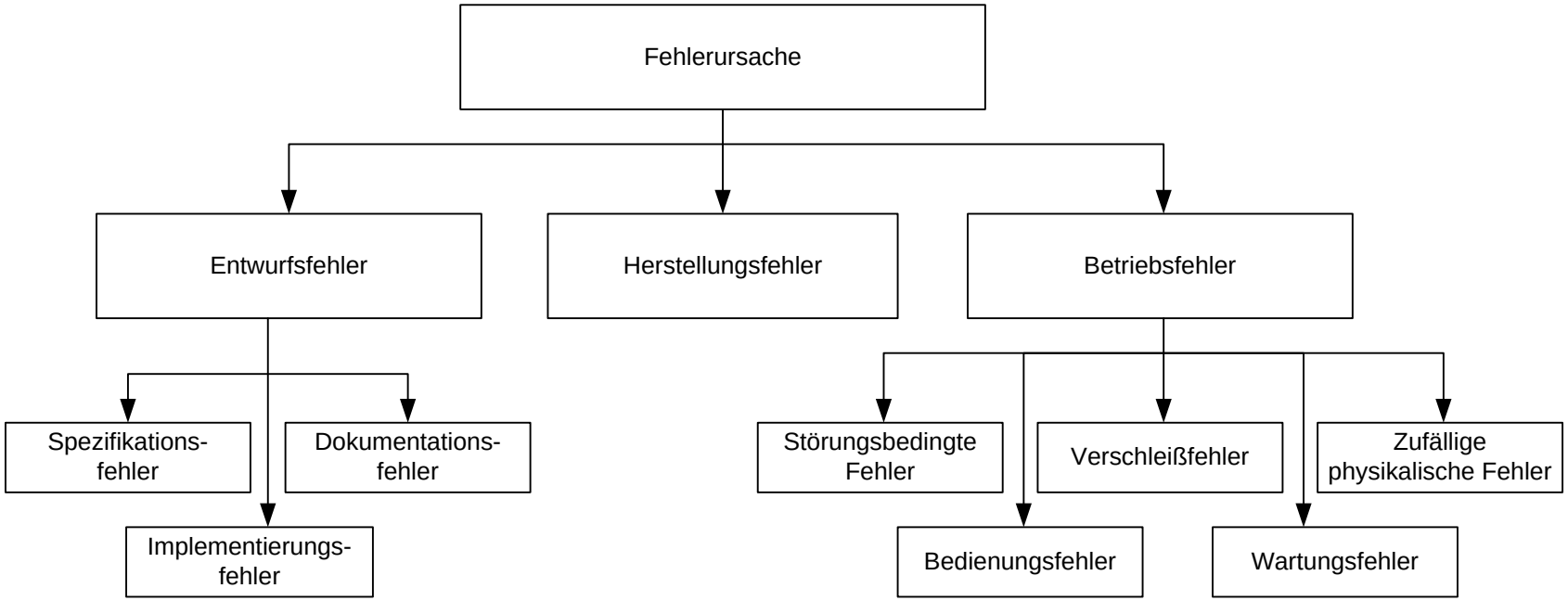
Kein Licht trotz Einschalten

Lichtausfall

Entwicklung sicherheitskritischer Systeme

Analyse der möglichen Fehler

Fehlerursachen



Klassifizierung von Fehlern

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

Beispiel Fehlerbilder Prozessoren und vorgeschlagene Erkennung

Table D.1 — Faults or failures to be analysed in the derivation of diagnostic coverage

Component	See Tables	Recommendations for diagnostic coverage		
		Low (60 %)	Medium (90 %)	High (99 %)
Processing units				
Register, internal RAM	D.4	Stuck-at (see footnote) for data and addresses	d.c. fault model (see foot note)for data and addresses	d.c. fault model for data and addresses Dynamic cross-over for memory cells No, wrong or multiple address-ing
Coding and execu-tion including flag register		Wrong coding or no execution	Wrong coding or wrong execution	No generic fault model avail-able. Detailed analysis neces-sary. Depends on CPU architec-ture
Address calculation		Stuck-at	d.c. fault model	No generic fault model avail-able. Detailed analysis neces-sary.
Interrupt handling		No or continuous inter-rupts	No or continuous inter-rupts Cross-over of interrupts	No or continuous interrupts Cross-over of interrupts

Table D.4 — Processing units

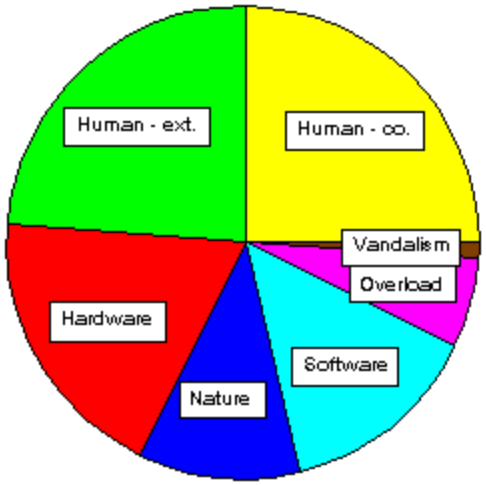
Diagnostic technique/measure	See overview of techniques	Maximum diagnostic coverage considered achievable	Notes
Self-test by software: limited num-ber of patterns (one channel)	D.2.3.1	Medium	Depends on the quality of the self test
Self-test by software cross ex-change between two independent units	D.2.3.5	Medium	Depends of the quality of the self test
Self-test by software: walking bit (one-channel)	D.2.3.2	Medium	Depends on the quality of the self test
Self-test supported by hardware (one-channel)	D.2.3.3	Medium	Depends on the quality of the self test
Coded processing (one-channel)	D.2.3.4	High	-

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.

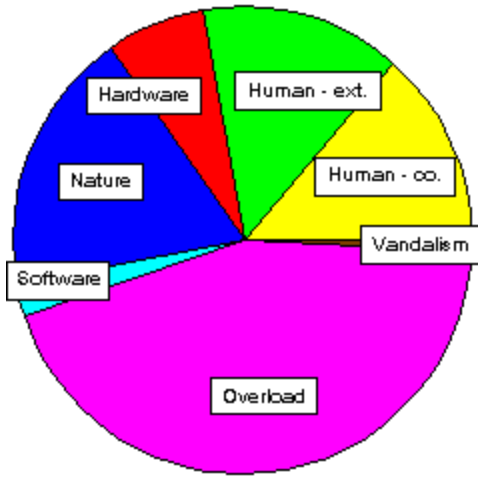
Ursachen und Wirkung

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 error - 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
  - $Fb_1 \cup ... \cup Fb_n \neq S$ (→ es gibt einen Perfektionskern $S \setminus (Fb_1 \cup ... \cup Fb_n)$
  - EXISTS $i,j \in \{1...n\}: Fb_i \cap Fb_j \neq \emptyset$ (→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, fail-silent)
  - Anhalte-Ausfall: sobald ein Fehler aufgetreten ist, gibt das System kein Ergebnis mehr aus (fail-stop): jedes ausgegebenen Ergebnis 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.

Entwicklung sicherheitskritischer Systeme

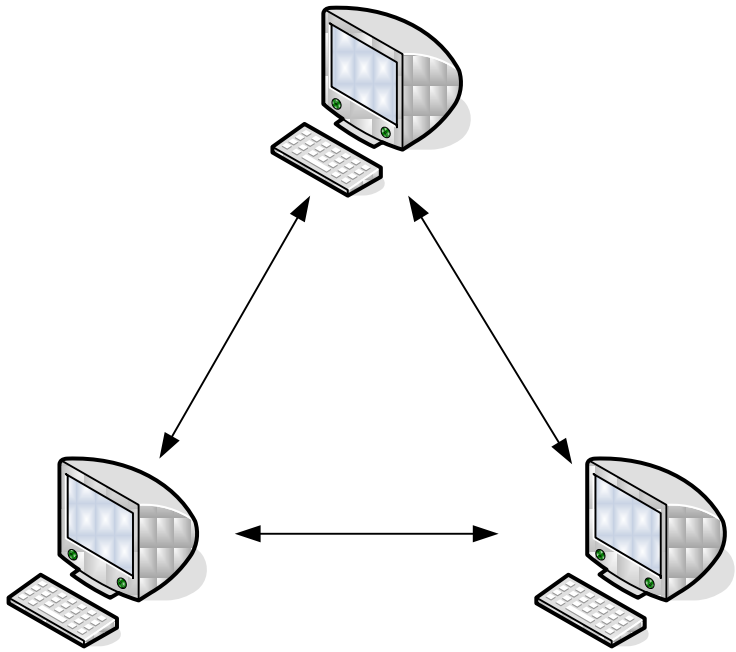
Fehlererkennung

Grundlage der Fehlererkennung: 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 redundancy) 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

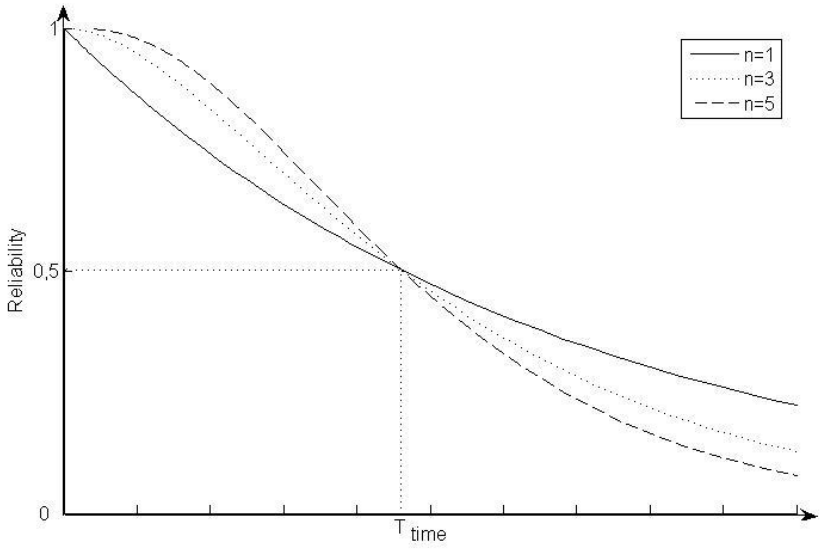
- Redundanz kann, muss aber nicht die Zuverlässigkeit verbessern:
- Beispiel: 2-von 3 System, stochastisch unabhängige Fehler, konstante Ausfallsrate λ, R_1: Zuverlässigkeit einer Komponte, R_3:Zuverlässigkeit des TMR-Systems

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

- Allgemeiner Fall m-von-n System:

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

→ 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.

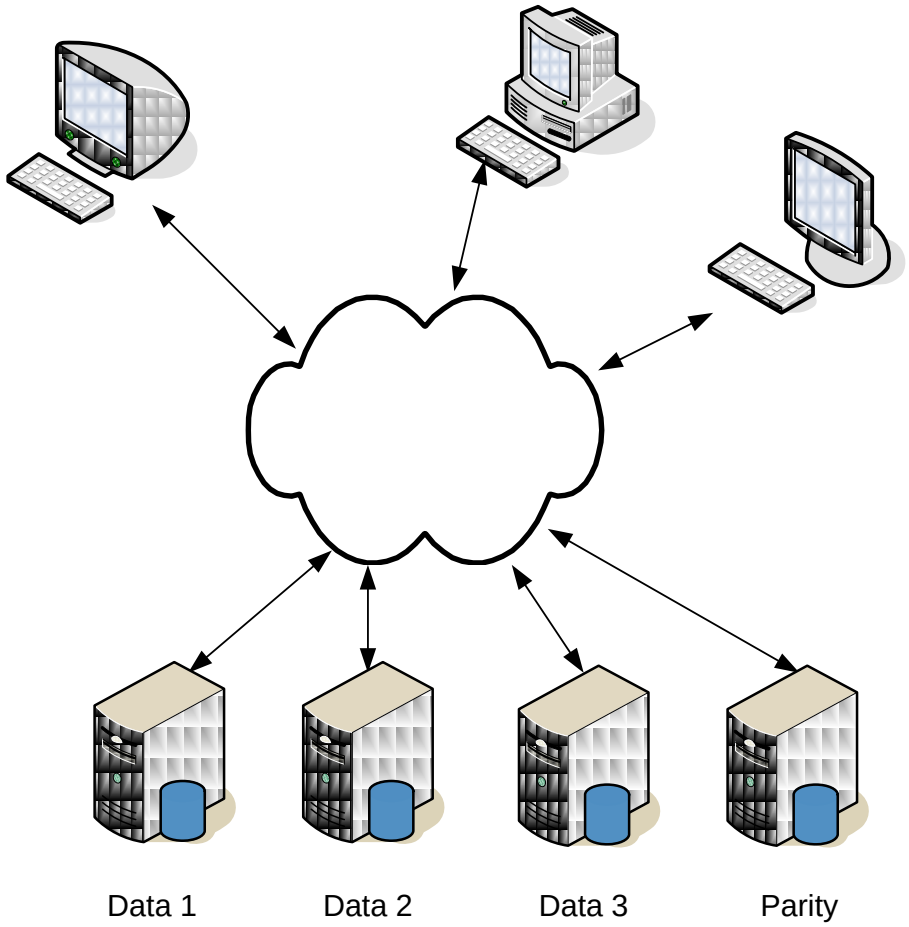
Strukturelle Redundanz

- 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:

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

$$1\ 0\ 0\ 1\ 0\ 0\ 0 \ / \ 1\ 0\ 1\ 1 = 1\ 0\ 1\ 0$$

$$\begin{array}{r}
1\ 0\ 1\ 1 \\
\hline
0\ 1\ 0\ 0 \\
0\ 0\ 0\ 0 \\
\hline
1\ 0\ 0\ 0 \\
1\ 0\ 1\ 1 \\
\hline
0\ 1\ 1\ 0 \\
0\ 0\ 0\ 0 \\
\hline
1\ 1\ 0 \quad \longleftarrow \text{Rest der Division}
\end{array}$$

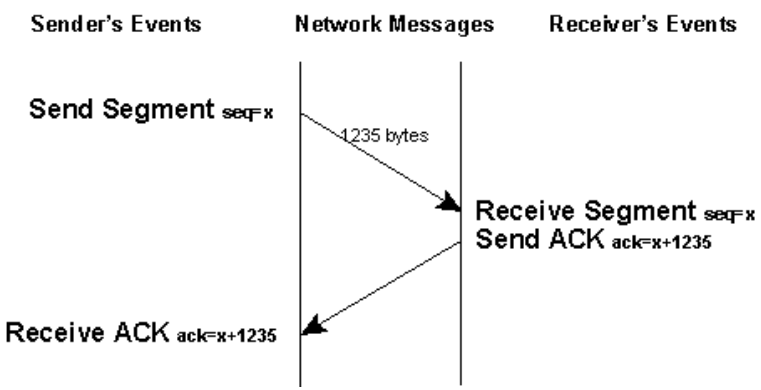
- Also Codewort: (**1001000**) – (**0000110**) = (**1001110**)

Zeitredundanz

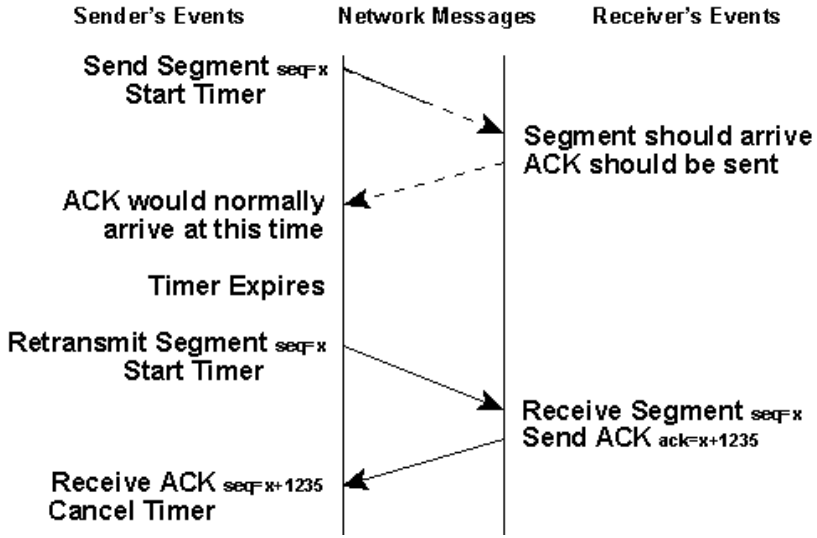
- Zeitredundanz bezeichnet über den Zeitbedarf des Normalbetriebs hinausgehende zusätzliche Zeit, die einem funktionell redundantem System zur Funktionsausführung zur Verfügung steht.
- Beispiele:
  - Wiederholungsbetrieb
  - Zeitbedarf für Konsistenzmechanismen in verteilten Dateisystemen

Beispiel: TCP

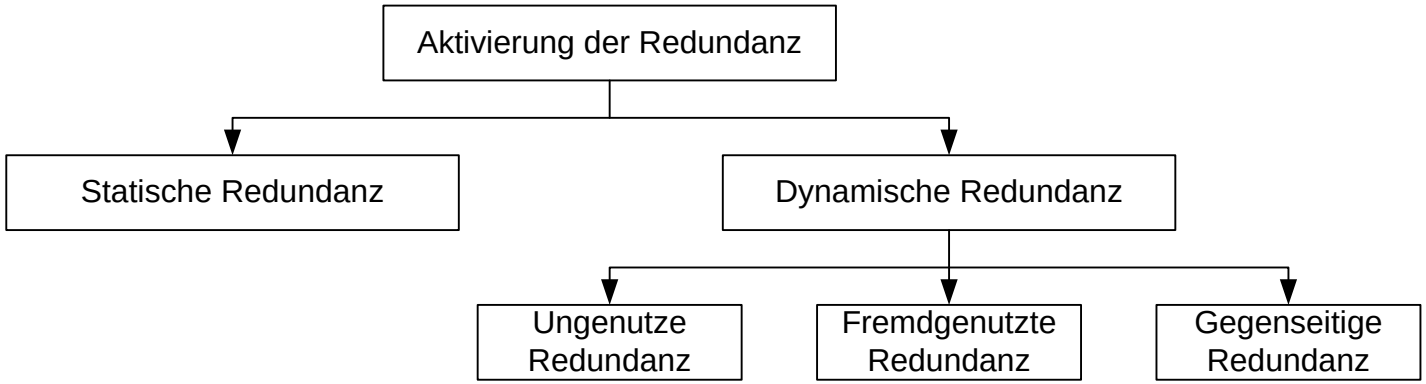
Fehlerfrei



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.

Entwicklung sicherheitskritischer Systeme

Fehlertoleranzmechanismen

Grundlage: Fehlererkennung

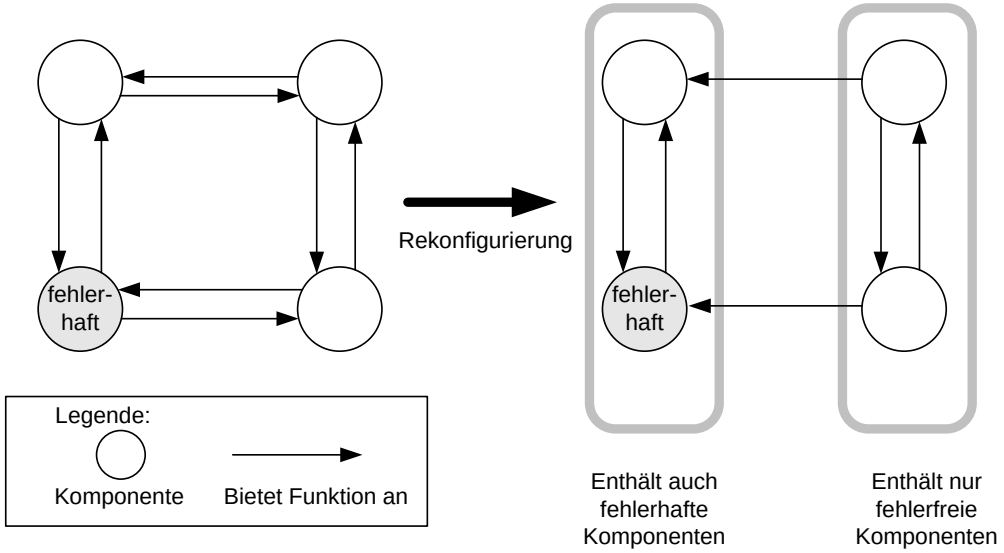
- Grundlage der Fehlertoleranzmechanismen ist die Fehlerdiagnose.
- Ziele der Fehlerdiagnose ist:
  - das Erkennen von Fehlern (im Nutzbetrieb)
  - die Lokalisierung von Fehlern (zumeist im Ausnahmebetrieb)
  - die Bestimmung des Behandlungsbereichs (zumeist im Ausnahmebetrieb)

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.

  → nicht ausreichend für erfolgreiche Fehlerbehandlung

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

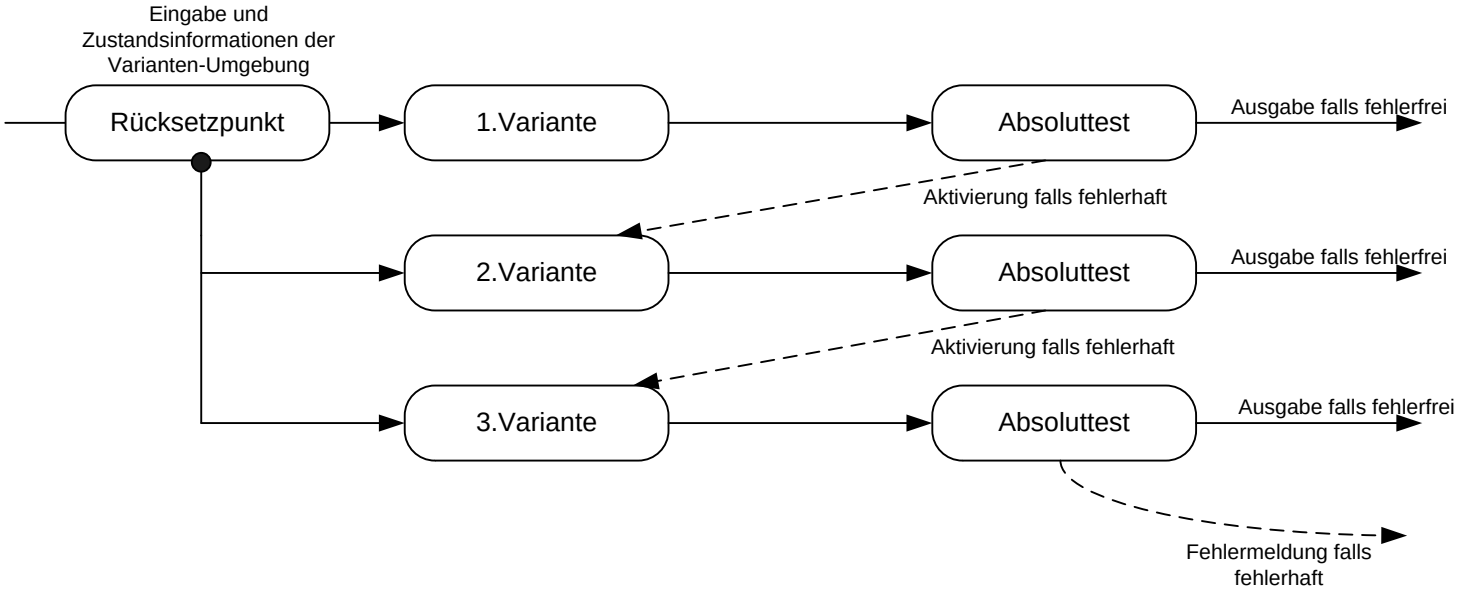
Rückwärtsbehebung (backward recovery)

- 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.

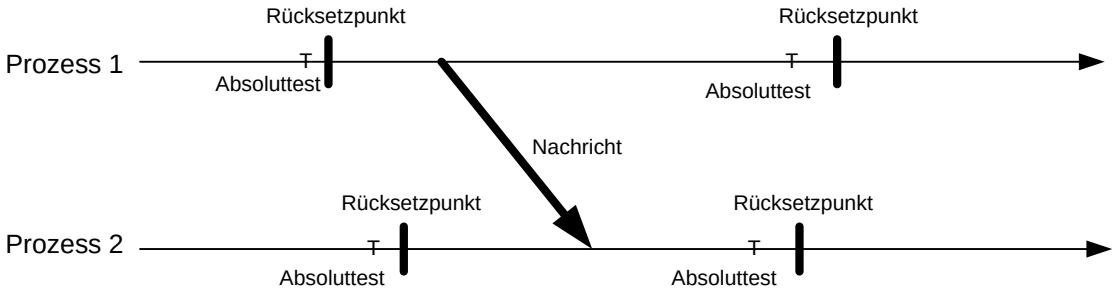
Rückwärtsbehebung diversitärer Systeme



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. ➔ schwierigere Abschätzung des Zeitverhaltens und damit eher schlecht für Echtzeitsysteme geeignet

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
- Minimale Verzögerung des Systems im Fehlerfall, da Vorwärtsbehebung meistens relativ einfache Algorithmen umfasst ➔ gute Eignung für Echtzeitsysteme
- Vorwärtsbehebung ist nicht transparent implementierbar, 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.

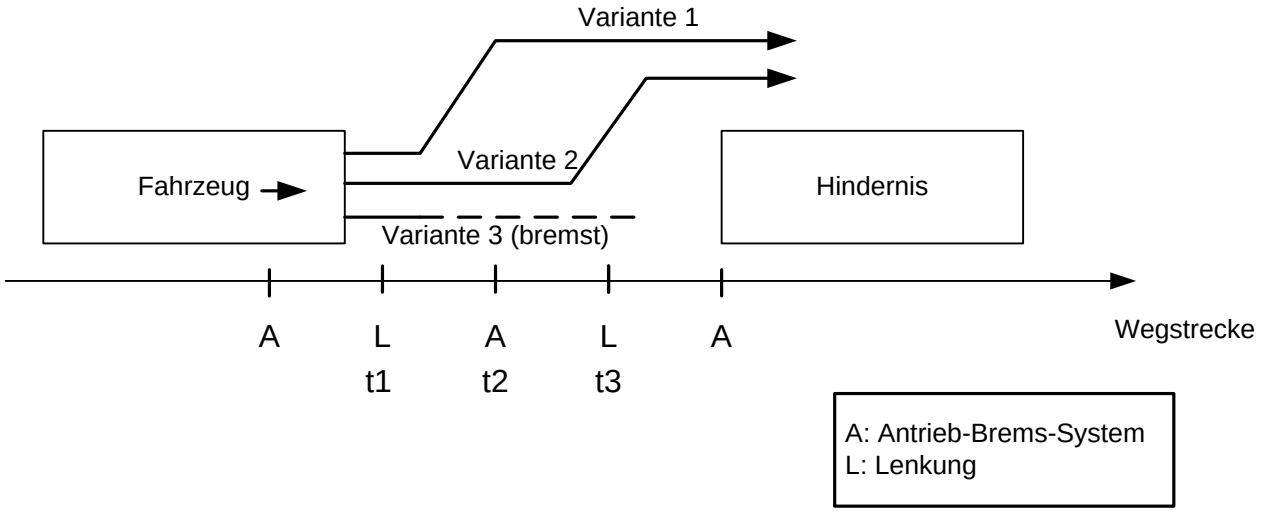
Maskierungsentscheidungen/Votierung

- 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 und Zeitverhalten unterscheidet sich nicht im Fehlerfall vom Normalfall ➔ besonders gut geeignet für harte Echtzeitsysteme
- 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.

  → 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)

  → 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. ➔ gut geeignet für Echtzeitsysteme
- 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.

Entwicklung sicherheitskritischer Systeme

Anforderungen der ISO 26262 an die Softwareentwicklung

Auswahl der Programmiersprache

Tabelle C.1 – Empfehlungen für bestimmte Programmiersprachen

	Programmiersprache	SIL1	SIL2	SIL3	SIL4
1	ADA	++	++	+	+
2	ADA mit Teilmenge	++	++	++	++
3	MODULA-2	++	++	+	+
4	MODULA-2 mit Teilmenge	++	++	++	++
5	PASCAL	++	++	+	+
6	PASCAL mit Teilmenge	++	++	++	++
7	FORTRAN 77	+	+	+	+
8	FORTRAN 77 mit Teilmenge	++	++	++	++
9	C	+	o	--	--
10	C mit Teilmenge und Codierrichtlinie und Verwendung von statischen Analysewerkzeugen	++	++	++	++
11	PL/M	+	o	--	--
12	PL/M mit Teilmenge und Codierrichtlinie	++	+	+	+

Klassische Entscheidung für laufendes Beispiel: Verwendung von C mit MISRA-Regeln und statischer Analyse zur Prüfung

Bewertungen der Programmiersprachen in der IEC 61508

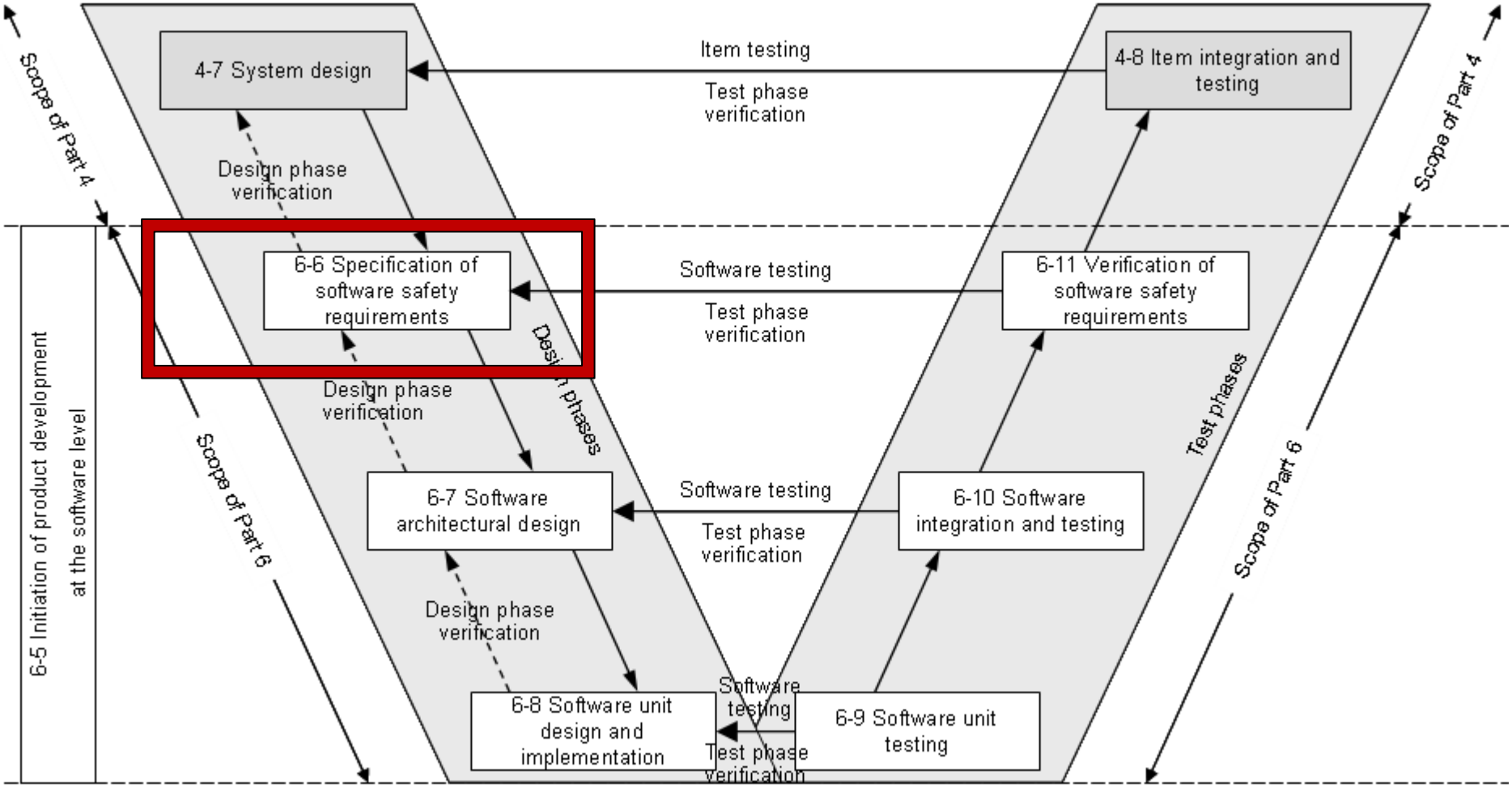
"++": Das Verfahren oder die Maßnahme wird für diesen Sicherheits-Integritätslevel besonders empfohlen. Wenn dieses Verfahren oder diese Maßnahme nicht verwendet wird, dann sollte der Grund während der Sicherheitsplanung ausführlich dargelegt und mit dem Gutachter abgestimmt werden.

"+": Das Verfahren oder die Maßnahme wird für diesen Sicherheits-Integritätslevel empfohlen. Es handelt sich um eine geringere Empfehlung als die mit "++" gekennzeichnete.

"o": Das Verfahren oder die Maßnahme hat keine Empfehlung für oder gegen die Verwendung.

"--": Das Verfahren oder die Maßnahme wird für diesen Sicherheits-Integritätslevel ausdrücklich nicht empfohlen. Wenn dieses Verfahren oder diese Maßnahme verwendet wird, dann sollte der Grund während der Sicherheitsplanung ausführlich dargelegt und mit dem Gutachter abgestimmt werden.

Vorgehensmodell bei der Softwareentwicklung



Spezifikation der Softwaresicherheitsanforderungen (SRS)

- Ziele:
  - Spezifikation aller Sicherheitsfunktionen der Software mit zugehörigen SIL
- Quelle der Sicherheitsanforderungen:
  - Systemsicherheitsanforderungsspezifikation
  - Hardwaresicherheitsanforderungsspezifikation
    - Beispiel: Erkennen des Ausfalls von Hardware (z.B. CPU- oder Speichertests)
- Regeln:
  - Präzise, quantitative Aussagen:
    - **Nicht**: Nach Betätigung des Lichtschalters wird das Ablendlicht eingeschalten.
    - **Sondern**: Maximal 1 Sekunde nach Betätigung des Lichtschalters wird das Licht eingeschalten. Die volle Lichtleistung wird nach maximal 5 Sekunden erreicht.
  - Präzise Benennung des Subjekts und der Verbindlichkeiten:
    - **Nicht**: Der Schalterstatus kann dem Fahrer signalisiert werden.
    - **Sondern**: Der Status des Schalters wird durch eine Schalterbeleuchtung angezeigt.

Methoden zur Definition und Verifikation der SRS

Tabelle A.1 – Spezifikation der Software-Sicherheitsanforderung (siehe 7.2)

	Verfahren/Maßnahme *	siehe	SIL1	SIL2	SIL3	SIL4
1	Rechnergestützte Spezifikationswerkzeuge	B.2.4	+	+	++	++
2a	Semi-formale Methoden	Tabelle B.7	+	+	++	++
2b	Formale Methoden mit z. B., CCS, CSP, HOL, LOTOS, OBJ, temporäre Logik, VDM und Z	C.2.4	o	+	+	++

Vorschlag zur Spezifikation der Softwaresicherheitsanforderungen zusätzlich zur prosaischen Beschreibung im IEC 61508-Standard.
Semi-Formale Methoden: z.B. Ablauf-, Datenfluss-, Zustandsübergangsdiagramme

Methoden zur Verifikation (ISO 26262) – gilt ähnlich für die nächste Phase

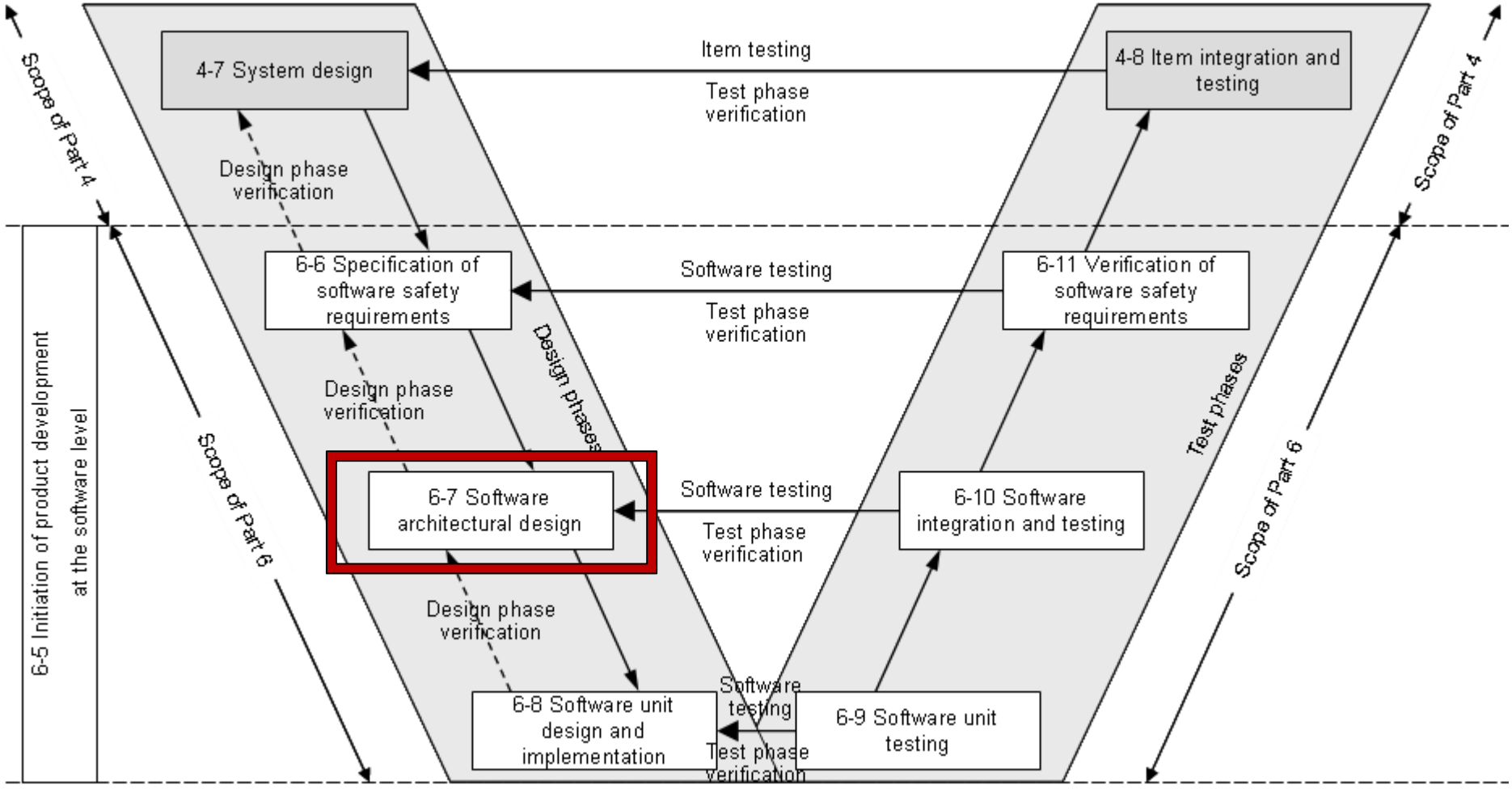
Table 2 — Methods for the verification of requirements

	Methods	ASIL A	ASIL B	ASIL C	ASIL D
1a	Informal verification by walkthrough	++	+	o	o
1b	Informal verification by inspection	+	++	++	++
1c	Semi-formal verification[a]	+	+	++	++
1d	Formal verification	o	+	+	+

[a] Method 1c can be supported by executable models.

**Interpretation der Tabellen: Alternative Methoden sind durch Buchstaben gekennzeichnet. Für die IEC 61508 reicht die Anwendung einer Methode aus, bei der ISO 26262 muss der Entwickler eine Auswahl anwenden und die Auswahl begründen (es sei denn alle höchstgewerteten Methoden sind gewählt).*

Vorgehensmodell bei der Softwareentwicklung



Allgemeine Anforderungen an die Softwarearchitektur

- Die Architektur beschreibt
  - Die Identifikation der Softwarekomponenten, der Struktur und der Schnittstellen
  - Die Interaktion zwischen den Softwarekomponenten, aber auch den Hardwarekomponenten durch Spezifikation der Datenstrukturen, der Kommunikation, Ablaufsequenzen, etc.
  - Das Systemumfeld (Ein-/Ausgangsdaten, interne Datenstruktur, Kommunikation, Schnittstellen, Wartungsdaten)
  - Datensicherungsmethoden
  - Resultierende Testfälle
- Die Anforderungen der Software-SRS müssen durch die Architektur erfüllt werden
  - Insbesondere gilt: Bei mehreren Software-Sicherheitsfunktionen mit unterschiedlichen ASIL ist das Gesamtsystem stets nach dem höchsten SIL zu entwickeln
    - Ausnahme: die Unabhängigkeit (Rückwirkungsfreiheit) kann nachgewiesen werden
  - Bei Verwendung von Standard- bzw. Wiederverwendbaren Komponenten:
    - Nachweis durch Betriebsbewährtheit (Beachtung der Randbedingungen – siehe ARIANE 5 - Unfall) oder entsprechende ASIL bezogene Implementierung (Safety-Element-out-of-Context)

Anforderungen an die Softwarearchitektur (ISO 26262)

Table 4 — Principles for software architectural design

Methods		ASIL			
		A	B	C	D
1a	Hierarchical structure of software components	++	++	++	++
1b	Restricted size of software components[a]	++	++	++	++
1c	Restricted size of interfaces[a]	+	+	+	+
1d	High cohesion within each software component[b]	+	++	++	++
1e	Restricted coupling between software components[a, b, c]	+	++	++	++
1f	Appropriate scheduling properties	++	++	++	++
1g	Restricted use of interrupts[a, d]	+	+	+	++

[a] In methods 1b, 1c, 1e and 1g "restricted" means to minimise in balance with other design considerations.

[b] Methods 1d and 1e can, for example, be achieved by separation of concerns which refers to the ability to identify, encapsulate, and manipulate those parts of software that are relevant to a particular concept, goal, task, or purpose.

[c] Method 1e addresses the limitation of the external coupling of software components.

[d] Any interrupts used have to be priority-based.

Zu verwendende Mechanismen (ISO 26262)

- Fehlererkennung:

Table 5 — Mechanisms for error detection at the software architectural level

	Methods	ASIL A	ASIL B	ASIL C	ASIL D
1a	Plausibility check[a]	++	++	++	++
1b	Detection of data errors[b]	+	+	+	+
1c	External monitoring facility	o	+	+	++
1d	Control flow monitoring	o	+	++	++
1e	Diverse software design[c]	o	o	+	++

[a] Plausibility checks include assertion checks. Complex plausibility checks can be realised by using a reference model of the desired behaviour.

[b] Types of methods that may be used to detect data errors include error detecting codes and multiple data storage.

[c] Diverse software design is not intended to imply n-version programming.

- Fehlerbehebung:

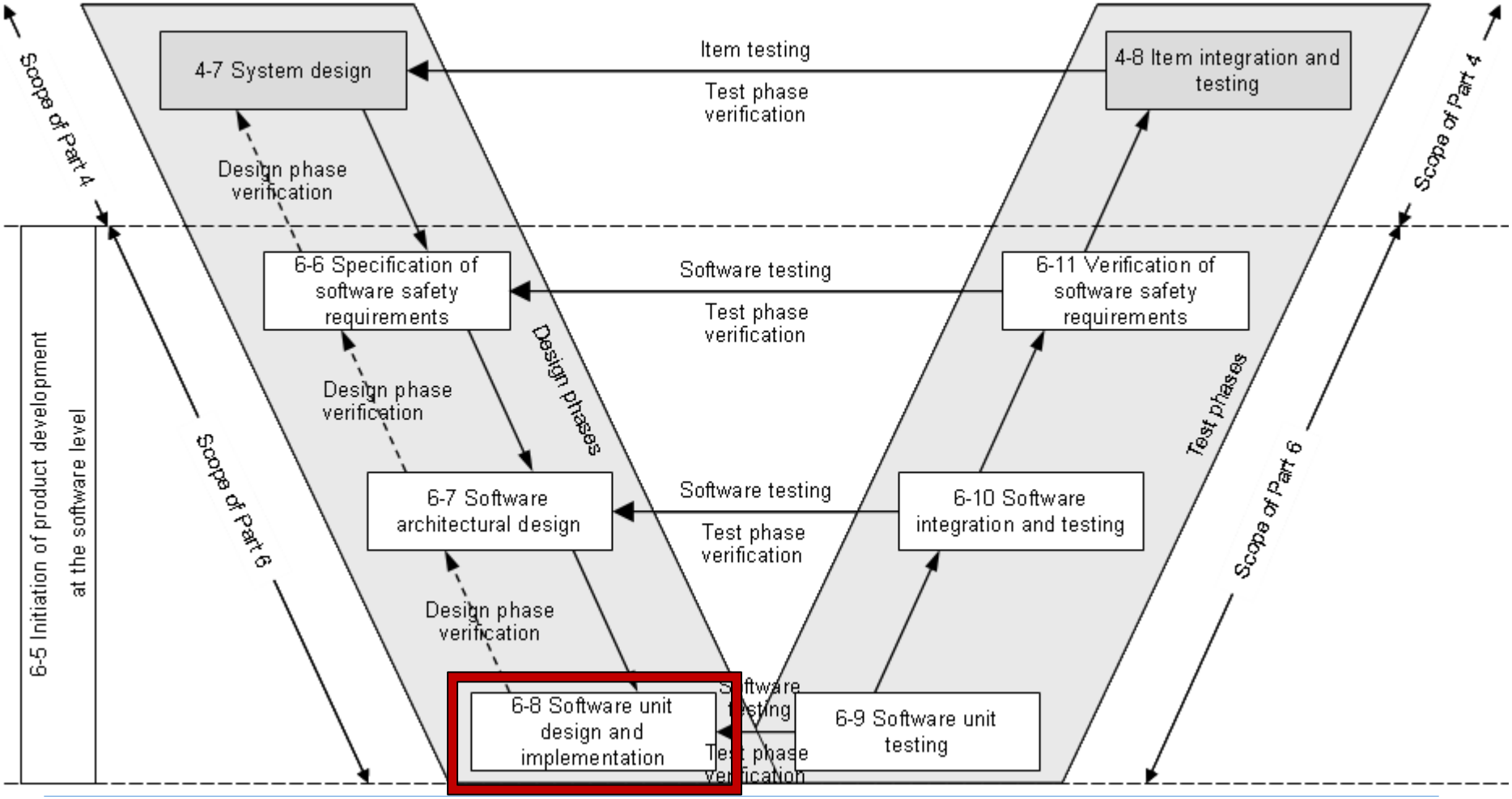
	Methods	ASIL A	ASIL B	ASIL C	ASIL D
1a	Static recovery mechanism[a]	+	+	+	+
1b	Graceful degradation[b]	+	+	++	++
1c	Independent parallel redundancy[c]	o	o	+	++
1d	Correcting codes for data	+	+	+	+

[a] Static recovery mechanisms can be realised by recovery blocks, backward recovery, forward recovery and recovery through repetition.

[b] Graceful degradation at the software level refers to prioritising functions to minimise the adverse effects of potential failures on functional safety.

[c] For parallel redundancy to be independent there has to be dissimilar software in each parallel path.

Vorgehensmodell bei der Softwareentwicklung



Anforderungen an die Implementierung

Beschreibung des Designs:

Table 8 — Notations for software unit design

Methods		ASIL A	ASIL B	ASIL C	ASIL D
1a	Documentation of the software unit design in natural language	++	++	++	++
1b	Informal notations	++	++	+	+
1c	Semi-formal notations	+	++	++	++
1d	Formal notations	+	+	+	+

Designprinzipien:

Table 9 — Design principles for software unit design and implementation

Methods		ASIL A	ASIL B	ASIL C	ASIL D
1a	One entry and one exit point in subprograms and functions[a]	++	++	++	++
1b	No dynamic objects or variables, or else online test during their creation[a, b]	+	++	++	++
1c	Initialisation of variables	++	++	++	++
1d	No multiple use of variable names[a]	+	++	++	++
1e	Avoid global variables or else justify their usage[a]	+	+	++	++
1f	Limited use of pointers[a]	o	+	+	++
1g	No implicit type conversions[a, c]	+	++	++	++
1h	No hidden data flow or control flow[b, d]	+	++	++	++
1i	No unconditional jumps[a, c, d]	++	++	++	++
1j	No recursions	+	+	++	++

[a] Methods 1a, 1b, 1c, 1d, 1e, 1f, 1g and 1i may not be applicable for graphical modelling notations used in model-based development.

[b] If these compiler features are "tool qualified" in accordance with ISO 26262-8:—, Clause 10, Method 1b need not be applied if a compiler is used which ensures that there will be enough program storage allocated for all dynamic variables and objects before run-time or which inserts run-time tests for correct online-allocation of program storage, i.e. stack bounds checking.

[c] Methods 1g and 1i are not applicable in assembler programming.

[d] Methods 1h and 1i reduce the potential for modelling data flow and control flow through jumps or global variables.

Verifikation der Implementierung

Table 10 — Methods for the verification of software unit design and implementation

Methods		ASIL			
		A	B	C	D
1a	Informal verification	See Table 11			
1b	Semi-formal verification[a]	+	+	++	++
1c	Formal verification	o	o	+	+
1d	Control flow analysis[b, c]	+	+	++	++
1e	Data flow analysis[b, c]	+	+	++	++
1f	Static code analysis	+	++	++	++
1g	Semantic code analysis[d]	+	+	+	+

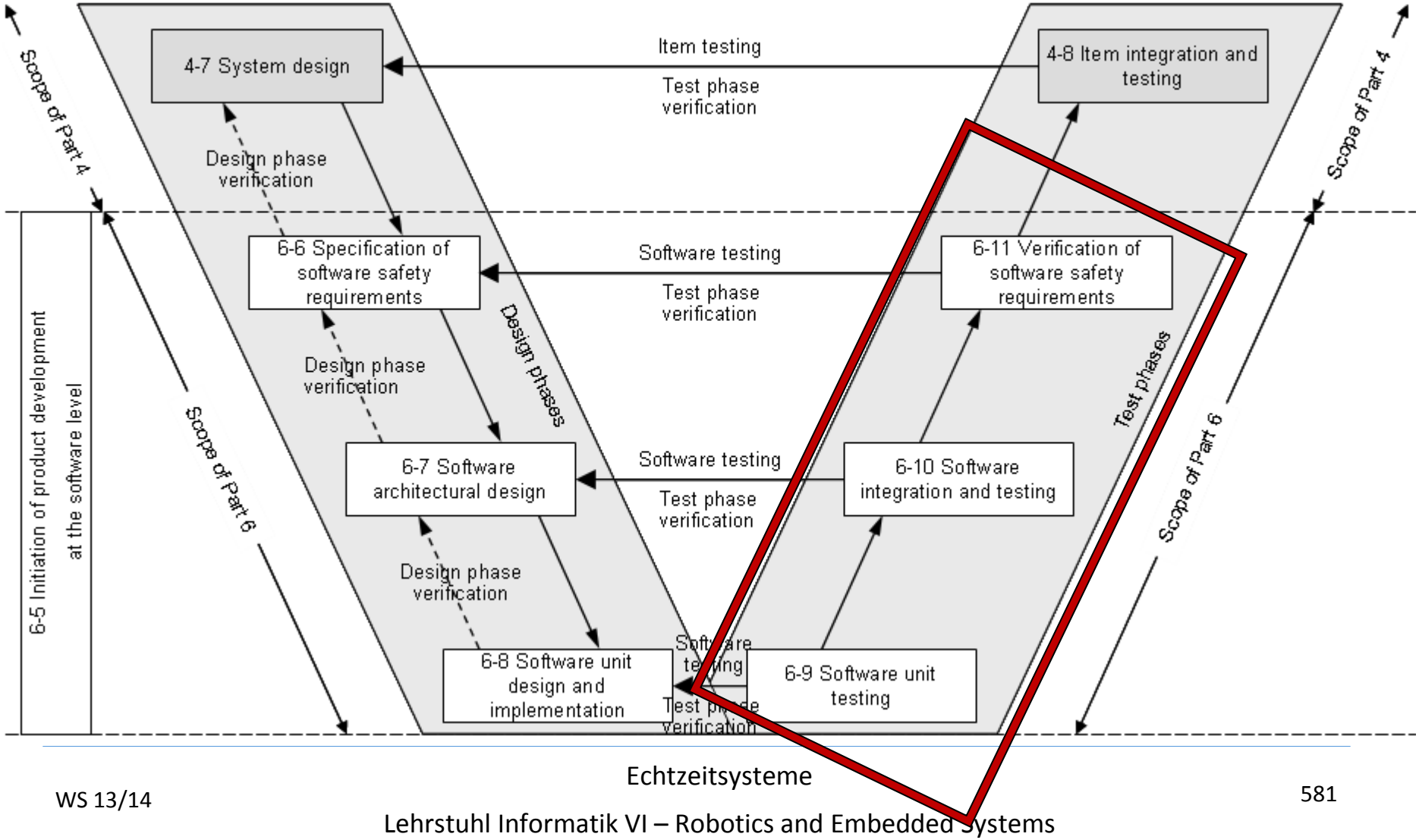
[a] Method 1b requires an executable design or implementation model of the unit to be verified.

[b] Methods 1d and 1e should be applied at the source code level. These methods are applicable both to manual code development and to model-based development.

[c] Methods 1d and 1e can be part of methods 1c or 1g.

[d] Method 1g is used for mathematical analysis of source code by use of an abstract representation of possible values for the variables. For this it is not necessary to translate and execute the source code.

Vorgehensmodell bei der Softwareentwicklung



Anforderungen an die Testarten

Auf Komponenten- & Architekturebene:

Table 12 — Methods for software unit testing

Methods		ASIL A	ASIL B	ASIL C	ASIL D
1a	Requirement-based test	++	++	++	++
1b	Interface test	++	++	++	++
1c	Fault injection test[a]	+	+	+	++
1d	Resource usage test[b]	+	+	+	++
1e	Back-to-back test between model and code, if applicable[c]	+	+	++	++

[a] This includes injection of arbitrary faults in order to test safety mechanisms (e.g. by corrupting values of variables)

[b] Some aspects of the resource usage test can only by evaluated properly when the software unit tests are executed on the target hardware or if the emulator for the target processor supports resource usage tests.

[c] This method requires a model that can simulate the functionality of the software units. Here, the model and code are stimulated in the same way and results compared with each other.

Auf Systemebene.

Table 18 — Test environments for conducting the software safety requirements verification

Methods		ASIL A	ASIL B	ASIL C	ASIL D
1a	Hardware-in-the-loop	+	+	++	++
1b	Electronic control unit network environments[a]	++	++	++	++
1c	Vehicles	++	++	++	++

[a] Examples are "lab-cars", "rest of the bus" simulations or test benches partially or fully integrating the electrical systems of a vehicle.

Testabdeckung in den einzelnen Phasen

Auf Komponentenebene:

Table 14 — Structural coverage metrics at the software unit level

Methods			ASIL A	ASIL B	ASIL C	ASIL D
1a	Statement coverage		++	++	+	+
1b	Branch coverage		+	++	++	++
1c	MC/DC (Modified Condition/Decision Coverage)		+	+	+	++

Auf Architekturebene:

Table 17 — Structural coverage metrics at the software architectural level

Methods		ASIL A	ASIL B	ASIL C	ASIL D
1a	Function coverage[a]	+	+	++	++
1b	Call coverage[b]	+	+	++	++

[a] The degree of coverage claimed in method 1a requires at least one execution of every software function. This evidence can be achieved by an appropriate software integration strategy.

[b] The degree of coverage claimed in method 1b requires at least one execution of every software function call.

Zusammenfassung

- Was sollen Sie aus diesem Kapitel mitgenommen haben?
  - Verständnis für Zweck und Aufbau von Zertifizierungsstandards
    - Produkthaftung
  - Kenntnisse der Sicherheitslevel und Einstufung einer Anwendung
    - 3 Kriterien: Schadensmaß, Aufenthaltsdauer, Kontrollierbarkeit
  - 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
  - Anforderungen an die Softwareentwicklung entlang des Entwicklungsprozesses
    - Phasen des V-Modells und die entsprechenden Anforderungen
    - Dokumentationsaufwand
    - Anforderungen an die Entwicklungsmethodik

Klausur WS 08/09 – 10 Punkte

- Gegeben seien folgende Systeme zu Erlangung von Fehlertoleranz:
  1. 3-aus-5-Rechnersystem
  2. Verteiltes System mit TTP als Kommunikationsprotokoll
  3. System mit periodischem Anlegen von Checkpoints und Rückwärtsbehebung im Fehlerfall
  4. Hot-Stand-By-System bestehend aus zwei Komponenten
- Ordnen Sie die Systeme eineindeutig den folgenden Anwendungen zu und begründen Sie kurz Ihre Antwort in Bezug auf die Echtzeitfähigkeit, sowie die tolerierbaren Fehler.
  a. In einem Atomkraftwerk muss die Steuerung fehlertolerant realisiert werden. Es muss mindestens ein beliebiger Rechnerausfall (fehlerhaftes Ergebnis bzw. Komplettausfall) toleriert werden können, zudem muss ein korrektes Ergebnis innerhalb von festen Zeitschranken bestimmt werden.
  b. Ein echtzeitkritisches Steuerungssystem soll zur Erhöhung der Verfügbarkeit fehlertolerant gestaltet werden. Aufgrund von intensiven Selbsttests kann davon ausgegangen werden, dass sich ein einzelner Rechner im Fehlerfall selbst abschaltet und kein fehlerhaftes Ergebnis ausgibt.
  c. Zur Verwaltung des Lagerbestandes wird in einem Logistiksystem ein fehlertolerantes System benötigt, das temporäre Fehler tolerieren muss.
  d. In einem Satelliten soll ein verteiltes System möglichst fehlertolerant gegenüber Strahlungseinflüssen entwickelt werden. Dabei werden speziell für den Weltraumeinsatz entwickelte Rechner mit Strahlenschutz eingesetzt.
- Lösung: 1a, 2d, 3c, 4b

Ausblick

Ausblick

- Vorlesung
  - Cyber-Physical Systems mit Schwerpunkt auf dem Thema Verifikation
  - Verteilte rekonfigurierbare Steuerungssysteme
  - Einführung in die digitale Signalverarbeitung
  - Kognitive Systeme
  - Bewegungsplanung in der Robotik
- Seminare:
  - Real-time Communication Systems
  - Machine Learning II
- Praktikum:
  - Applied Computer Vision for Robotics
  - RACE
  - HW/SW co-design with a LEGO Car
- Studienarbeiten (BA, MA)
- Studentische Hilfskräfte / Promotion (Lehrstuhl bzw. fortiss)