Echtzeitverarbeitung

R. Kaiser, K. Beckmann, R. Kröger

(HTTP: http://www.cs.hs-rm.de/~kaiser
EMail: robert.kaiser@hs-rm.de)

Sommersemester 2021

4. Echtzeitbetriebssysteme

https://school-time.co/

Inhalt

Hochschule RheinMain

4. Echtzeitbetriebssysteme
4.1 Einführung
4.2 Architektur von Echtzeitbetriebssystemen
4.3 Der Markt für Echtzeitbetriebssysteme
4.4 Beispiele von Echtzeitbetriebssystemen

Einführung

Hochschule RheinMain

Wiederholung Betriebssysteme

Definition (Betriebssystem)

Ein Betriebssystem ist ein Programm, das alle Betriebsmittel eines Rechensystems verwaltet und ihre Zuteilung kontrolliert und den Nutzern des Rechensystems eine virtuelle Maschine offeriert, die einfacher zu verstehen und zu programmieren ist als die unterlagerte Hardware.

- Betriebssystem als „Betriebsmittelverwalter“
- Betriebssystem als „virtuelle Maschine“
  - Hardware-Unabhängigkeit
  - Adäquate Abstraktionen
  - Langlebigkeit der Programmierschnittstelle

Aufgabenbereiche eines Betriebssystems

Wichtige Aufgabenbereiche

- Verwaltung der Schnittstelle zur unterliegenden Hardware
- Prozessverwaltung
- Speicherverwaltung
- Interprozesskommunikation
- Ein/Ausgabe
- Dateisysteme

Echtzeitbetriebssysteme ...

- ... bilden unterlagerte Software-Schicht für komplexe Echtzeit- / Embedded Control- Anwendungen
- ... haben prinzipiell die gleichen Aufgabenbereiche wie übliche Betriebssysteme

Aber: es kommen Randbedingungen hinzu ...

Aufgabenbereiche (2)

Hochschule RheinMain

... nämlich:

- bessere Unterstützung für die Vorhersagbarkeit des Systemverhaltens (→ Echtzeitcharakter) (s.u.)
- Konfigurierbarkeit / Skalierbarkeit
  - Auskommen mit minimalen Resourcen (Kostengründe in Massenprodukten)
  - nur die wirklich benötigten Komponenten sollten Bestandteil des aktuell verwendeten Betriebssystems sein
- Stark variierende Zielumgebungen
  - unterschiedlichste I/O-Konfigurationen
  - Boot aus ROM oder Betrieb aus ROM bei diskless targets (häufig!)
- Entwicklungsprozess
  - Häufig Cross-Entwicklungsumgebung notwendig (s. Kap. 3)
  - Debugging schwierig (auch wg. Verfälschung der Echtzeiteigenschaften)

Anforderungen bzgl. Vorhersagbarkeit

- Durchsetzen von Planungsentscheidungen (Scheduling):
  - Prioritäten-basierte Scheduler als Standardfall
  - Zuordnung von anwendungsspezifischen Prioritäten zu Prozessen
  - Verwendung spezifischer Scheduler
  - Formulierung und Überwachung von Zeitbedingungen
- Unterbrechungsbehandlung:
  - Auftreten externer Ereignisse muss gemäß der Dringlichkeit bearbeitet bzw. an das zugehörige Anwendungsprogramm weitergeleitet werden.
- Zeitdienste:
  - Operationen zum Realisieren von Verzögerungen, zeitgenauen Aktionen und das zeitgesteuerte Aktivieren und Deaktivieren.
- Unterbrechbarkeit des Betriebssystemkerns:
  - Die Bearbeitung eines Systemdienstes für einen Prozess darf die Bearbeitung einer zeitkritischen Aktion nicht behindern.
  - Problem: Unterbrechung der Bearbeitung eines system calls an beliebiger Stelle kann zu Inkonsistenzen führen

Anforderungen bzgl. Vorhersagbarkeit (2)

- Kontrolle über die Speicherverwaltung:
  Virtuelle Speicherverwaltung muss ausgesetzt oder vorhersagbar sein.
  - wie lange braucht ein `malloc()`?
- Vorhersagbarkeit des Dateizugriffsverhaltens.
  - wie lange dauert das Anlegen einer Datei?
- Vorhersagbarkeit des Verhaltens der Kommunikationskanäle.
  - wann und für wie lange ist der Bus frei?

→ Designentscheidungen in allen Ebenen

Anforderungen bzgl. Vorhersagbarkeit (2)

- Kontrolle über die Speicherverwaltung:
  Virtuelle Speicherverwaltung muss ausgesetzt oder vorhersagbar sein.
  - wie lange braucht ein `malloc()`?
- Vorhersagbarkeit des Dateizugriffsverhaltens.
  - wie lange dauert das Anlegen einer Datei?
- Vorhersagbarkeit des Verhaltens der Kommunikationskanäle.
  - wann und für wie lange ist der Bus frei?

→ Designentscheidungen in allen Ebenen

Klassifizierung

Hochschule RheinMain

Klassifizierung von BSen nach dem Funktionsumfang:

- Elementare Runtime-Systeme
  - einfaches Multitasking (eher Multithreading)
  - Betriebsmittel-optimiert
  - einfache (i.d.R. statische) Datenstrukturen (Bibliotheks-Charakter)
  - Beispiele:
    - Contiki (`www.sics.se/contiki`)
    - μC/OS-II (Micrium)
    - FreeRTOS (`www.freertos.org`)
    - OSEK-OS
- Multitasking-Kerne
  - Ein-Adressraum-Verwaltung (keine MMU-Nutzung)
  - dynamische Datenstrukturen (Tasks, Speichersegmente, ...)
  - moderater Betriebsmittelverbrauch
  - Steuerbarkeit des Echtzeitverhaltens
  - Beispiele:
    - VxWorks (`https://www.windriver.com`)
    - PXROS (HighTec, Saarbrücken)

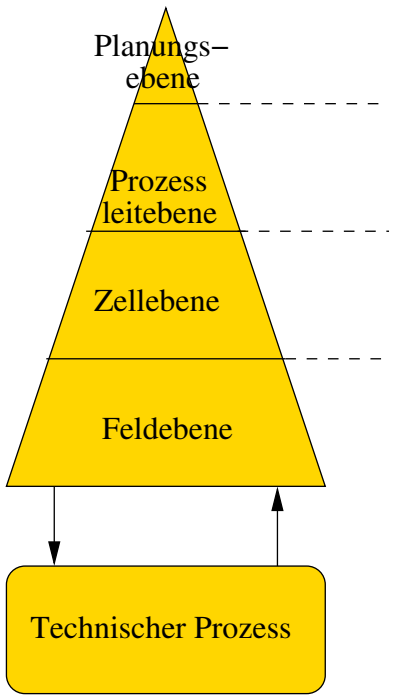
Klassifizierung (2)

- Vollständige Betriebssysteme incl. MMU-Unterstützung
  - Bereitstellung/Nutzung mehrerer Adressräume
  - Umfangreiches Dienstangebot
  - Verschiedene Dateisysteme
  - Netzwerke, Protokollstacks
  - i.d.R. Mittlere bis hohe Betriebsmittelanforderungen
  - Beispiele:
    - QNX-Neutrino
    - PikeOS
    - Linux (RTLinux, RTAI)
    - LynxOS
  - Übergänge z.T. fließend durch Konfigurierbarkeit des Kerns bzw. Erweiterbarkeit um entsprechende Subsysteme.

Beispiel: Anforderungen

Hochschule RheinMain

Automatisierungspyramide



Ebene	Charakteristische Anforderungen	Technologie
Planungsebene	Keine Echtzeitanorderungen, Mehrbenutzer-Umgebung, CAD, CAP	klassische EDV-Systeme
Prozessleitebene	Eingeschränkte Echtzeitfähigkeit, Graphische Bedienoberfläche, CAM, CAQ	Workstations, Visualisierungs-Software, Touchscreens
Zellebene	Harte Echtzeitbedingungen, kurze Reaktionszeiten, realisierung globaler konsistenter Zustände, Erkennung (globaler) kritischer Zustände, CNC, SPS, ...	Prozessrechner, Echtzeitbetriebssysteme, komplexe verteilte Steuerungsaufgaben
Feldebene	Harte Echtzeitbedingungen, kurze bis sehr kurze Reaktionszeiten	μController, Echtzeitkerne

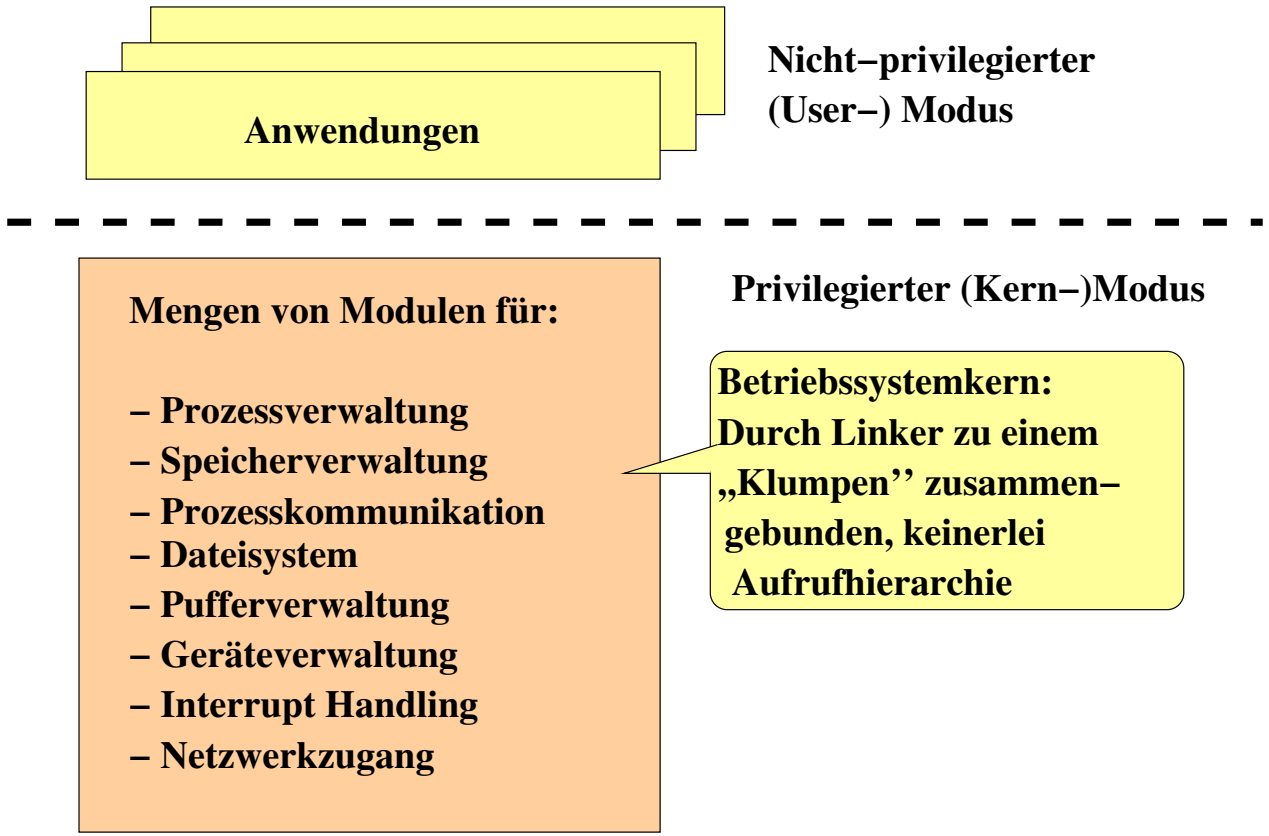
Architektur von Echtzeitbetriebssystemen

Organisationsformen von Betriebssystemen

- Klassifizierung der inneren Organisationsformen von Betriebssystemen
  1. Monolithische Systeme
  2. Geschichtete Systeme
  3. Kern-im-Kern Systeme
  4. Mikrokerne und virtuelle Maschinen

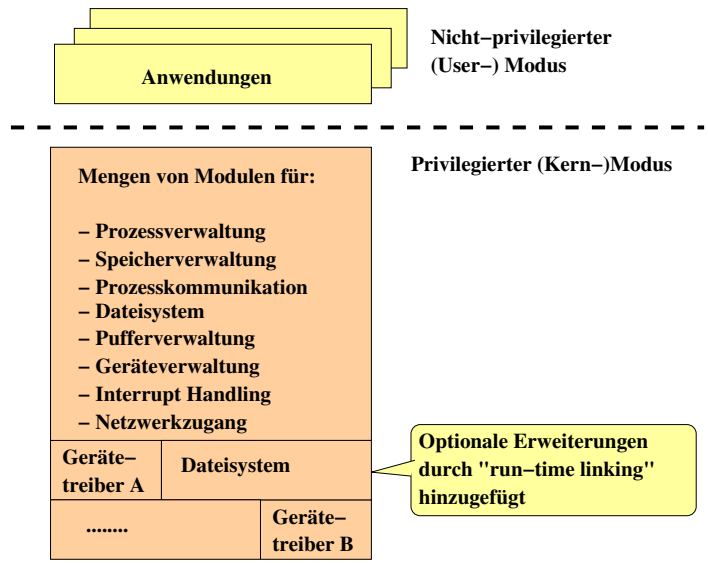
Monolitische Systeme

- Vorwiegende Struktur aller kommerziellen General Purpose Betriebssysteme: z.B. Windows, klassisches UNIX



Monolitische Systeme – Erweiterung

- Erweiterung eines monolithischen Kerns um ladbare Kernmodule
  - Ursprünglich in klassischen Betriebssystemen primär für Gerätetreiber gedacht
  - Sehr verbreitet bei eingebetteten Systemen, da so ein hohes Maß an Adaptierbarkeit an benötigte Kerndienste erreicht wird
  - Heute auch weit verbreitet bei üblichen Betriebssystemen (z.B. Linux)



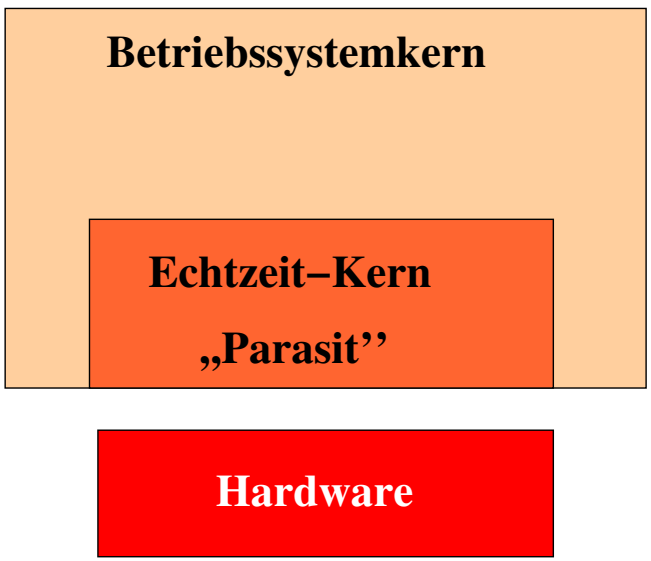
Geschichtete Systeme

Verallgemeinerung des monolithisches Ansatzes:

- Das BS wird als eine Hierarchie von Schichten (engl. layers) entworfen.
- Jede Schicht abstrahiert von gewissen Restriktionen der darunterliegenden Schicht. Die Implementierung einer Schicht benutzt die Dienste der darunterliegenden Schicht.
- Erstes System: THE (Techn. Hochschule Eindhoven, Dijkstra, 1968, einfaches Stapelverarbeitungssystem in Pascal).
- Weitere Verallgemeinerung in MULTICS: "konzentrische (Schutz-) Ringe", verbunden mit nach innen zunehmender Privilegierung, kontrollierter Aufruf zwischen den Ebenen zur Laufzeit.

Kern-im-Kern Systeme

- Insbesondere zum „Nachrüsten“ von Echtzeit-Funktionalität in Nicht-Echtzeit Betriebssysteme genutzt
- „Virtualisierung“ des Interrupt-Systems
- Der Echtzeit-Kern setzt sich als Kernmodul des Wirtskerns unter diesen und kontrolliert das reale Interrupt-System
- Der Wirtskern wird zum „Idle“-Prozess des Echtzeit-Kerns



Beispiele

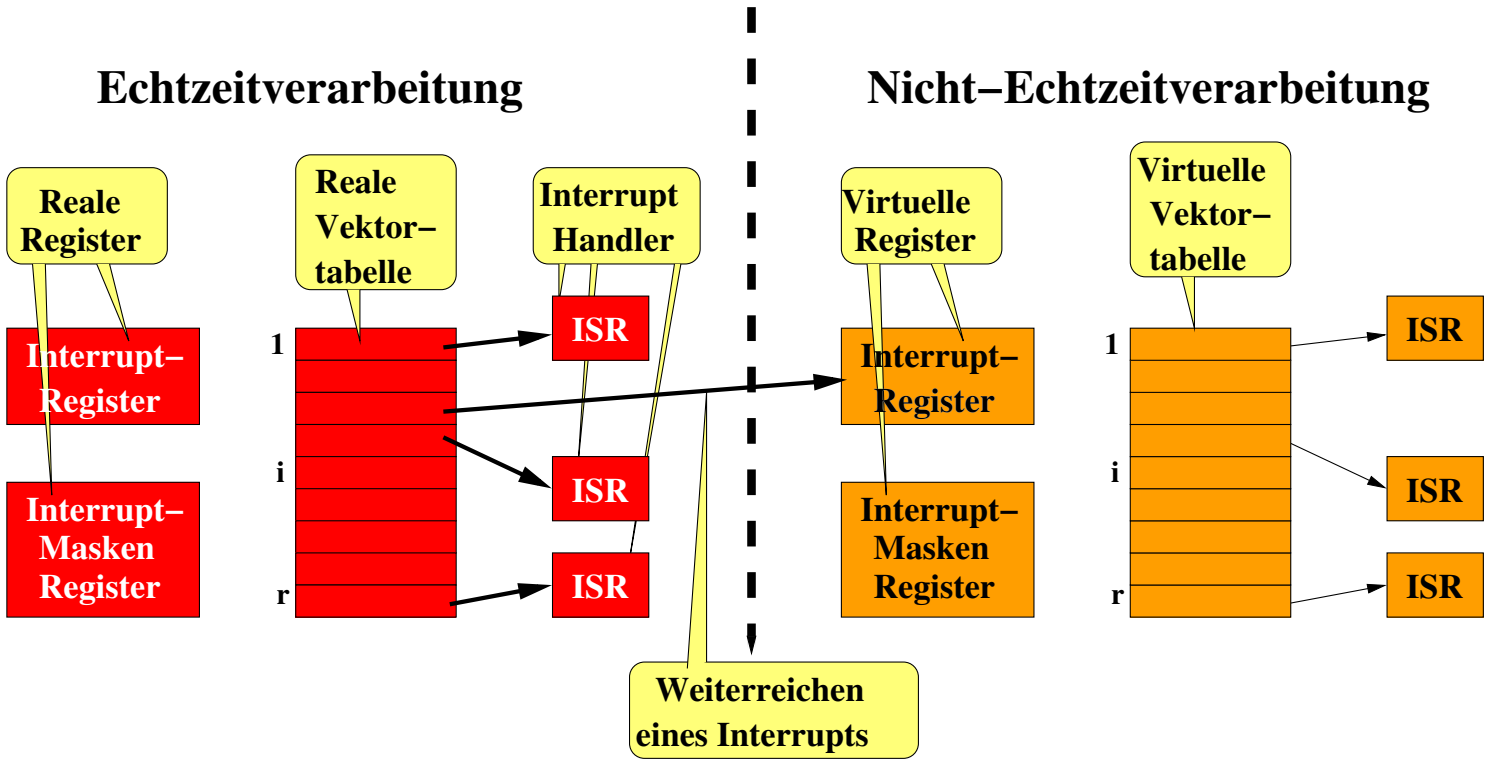
Wirtssystem	Linux	Linux	WinNT
Kern-im-Kern	RTLinux	RTAI	RTX

RTLinux – M. Barabanov, V. Yodaiken (New Mexico Inst. of Technology)
RTAI – Real-Time Application Interface, P. Mantegazza (DIAPM Mailand)
RTX – IntervalZero Inc.

Virtualisierung des Interruptsystems

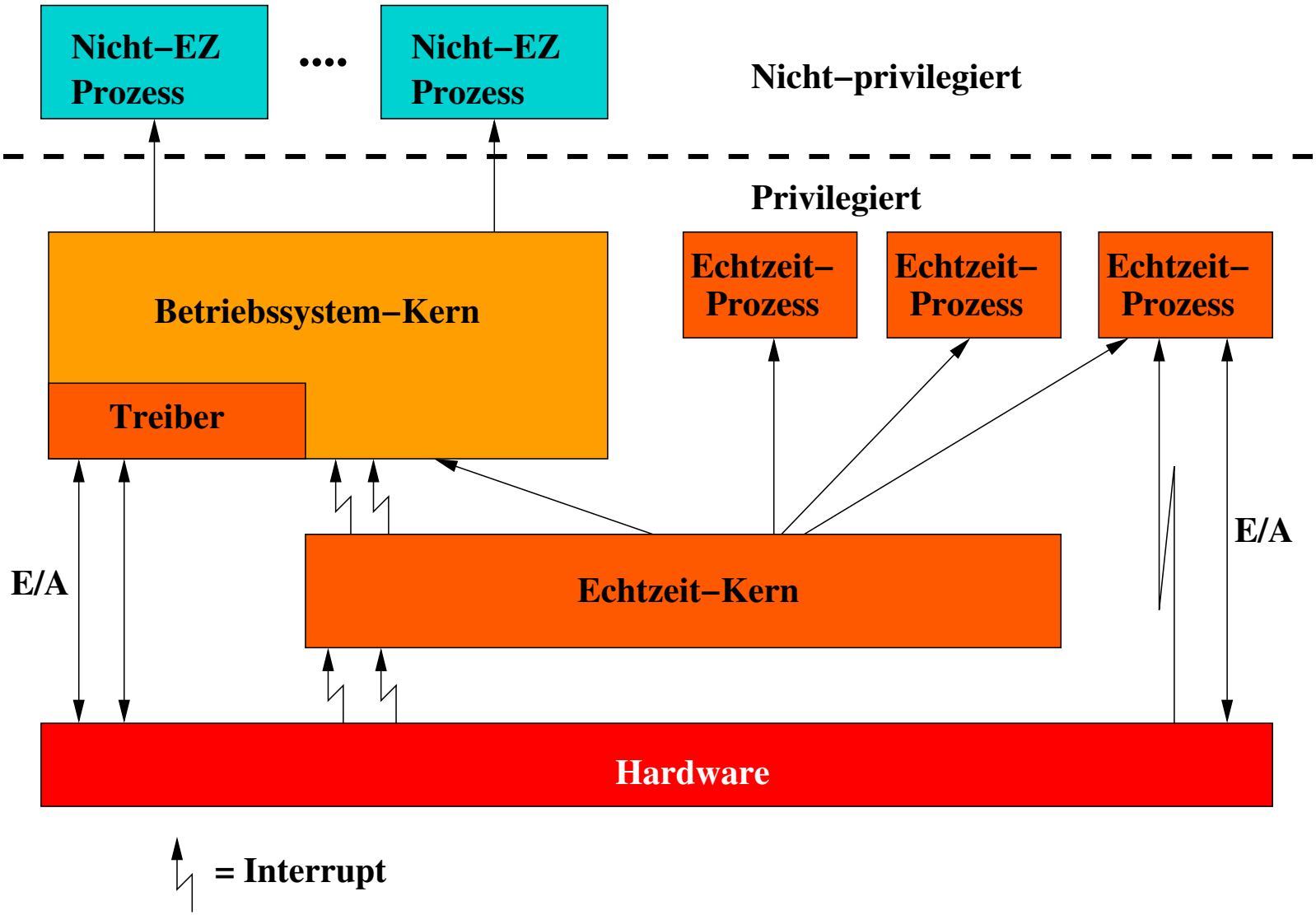
Hochschule RheinMain

- Software-Emulation des Interrupt-Systems
- Minimale Änderungen des Wirts-Kerns: Ersetzen aller Interrupt-bezogenen Operationsaufrufe (`cli`, `sti`, `iret`) im Wirts-Kern durch entsprechende emulierende Makros



Gesamtarchitektur

Hochschule RheinMain



Merkmale

Hochschule RheinMain

- Echtzeit-Kern: Ziel: Nutzung der Hardware durch Echtzeit-Prozesse mit minimaler Latenzzeit
- Echtzeit-Kern mit allen Komponenten und Echtzeit-Anwendungen laufen im privilegierten Modus

  → Geringe Prozesswechselzeiten für Echtzeit-Prozesse

  aber: Fehler in Echtzeit-Anwendung → Absturz des Wirtskerns!

  aber: Keine Systemdienste des Wirtskerns verfügbar!

  aber: Echtzeit-Anwendungen müssen für jede neue Version des Wirts-Kerns neu kompiliert werden
- Scheduling und Echtzeiteigenschaften von Echtzeit-Prozessen können nicht durch den Wirtskern beeinflußt werden
- Wirtskern: Funktionalität nicht eingeschränkt
- Unmerklich schlechtere Rechenleistung wegen Indirektion der Interrupt-Verarbeitung

Der Echtzeit-Kern als Parasit

- Der Echtzeit-Kern nutzt den Wirtskern zu seiner eigenen Konfiguration:
- Installieren von Komponenten des Echtzeit-Kerns als ladbare Module des Wirtskerns
- Beispiel: Linux + RTAI (oder RTLinux):
  - Linux `insmod` und `rmmod` zum Laden der RTAI-Module
  - `rtai` – RTAI framework, interrupt dispatching, timer support
  - `rtai_sched` – preemptiver, Prioritäten-basierter Scheduler
  - `rtai_fifos` – FIFOs, Semaphoren
  - `rtai_shm` – shared memory
  - `rtai_pthread` – POSIX threads
- Der RT-Kern nutzt die Funktionen des Wirtskerns soweit irgend möglich
- Beispiel: Geräteinitialisierung (ist nicht echtzeitkritisch)

Mikrokerne und virtuelle Maschinen

Hochschule RheinMain

- Bei den bisher vorgestellten Organisationsformen wird unnötig viel Funktionalität im privilegierten Modus realisiert
- Für Funktionen wie Netzwerkprotokolle, Dateisysteme, ja sogar Gerätetreiber ist das keineswegs zwingend erforderlich
- Da die Funktionen im Kern liegen, zählen sie zur „Trusted Code Base“
- (s.o.) Dadurch wird die Trusted Code Base –selbst für einfachste Anwendungen– so groß, dass eine umfassende Validierung (geschweige denn eine Verifikation) praktisch unmöglich ist

$\Rightarrow$ Mikrokern-Ansatz[1]: Funktionen nur dann im Kern, wenn sie ausseralb nicht realisierbar sind

  - Prozesse, Speicherschutz, Betriebsmittelzuteilung
  - Nur **ein** Dienst im Kern: Inter-Prozess-Kommunikation (IPC)

[1] Siehe J. Liedtke: „On μ-kernel construction“

Mikrokern-Prinzip

- Ein Mikrokern ist nicht einfach nur ein kleinerer Kern
- Konstruktionsprinzip: Trennung von Methode und Mechanismus (*separation of policy and mechanism*)
- Beispiel: Speicherverwaltung:
  - Methode/Policy: „Welcher Prozess darf auf welchen Speicher zugreifen?“
  - Mechanismus/Mechanism: „Wie programmiert man die Memory Management Unit des Prozessors?“
- Prinzip: Mechanismen gehören in den Mikrokern, Methoden nicht!

$\Rightarrow$ Kern wird klein, wenig komplex, kleine „Trusted Code Base“

$\Rightarrow$ Validierung (sogar: Verifikation) wird machbar

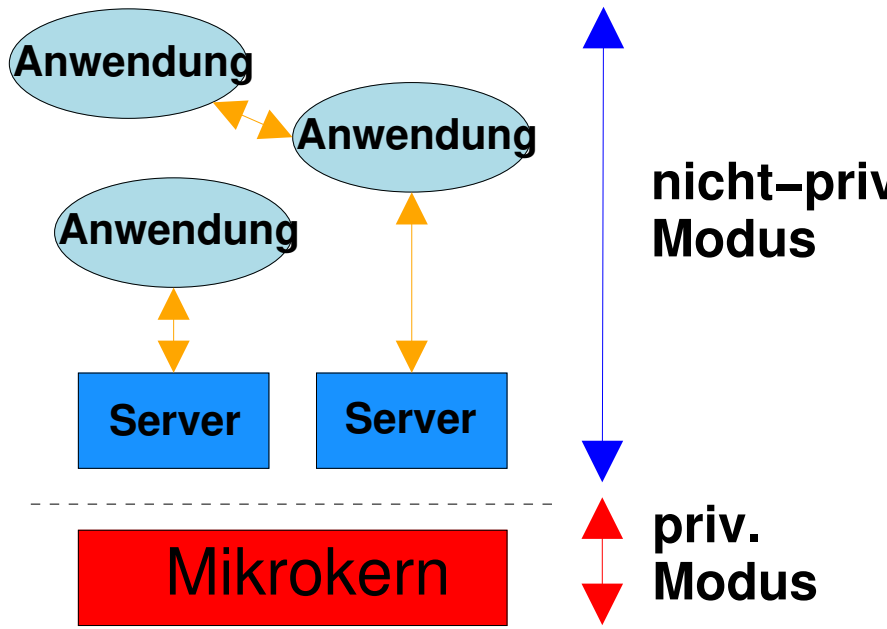
?!? „Aber wo werden dann die Betriebssystem-Dienste erbracht?“

Server/Client-Architektur

Hochschule RheinMain

- Dienste werden durch *Server* erbracht
- Prozesse, die im nicht-privilegierten Modus in geschütztem Adressräumen arbeiten
- Interprozesskommunikation ersetzt den Kernaufruf
- Verschiedene Server können verschiedene, alternative Dienstmengen anbieten

→ Mehrere Betriebssysteme gleichzeitig in einer Maschine



- Vorteil: Anwendungen müssen nur den Servern trauen, deren Dienste sie nutzen

Server/Client-Architektur

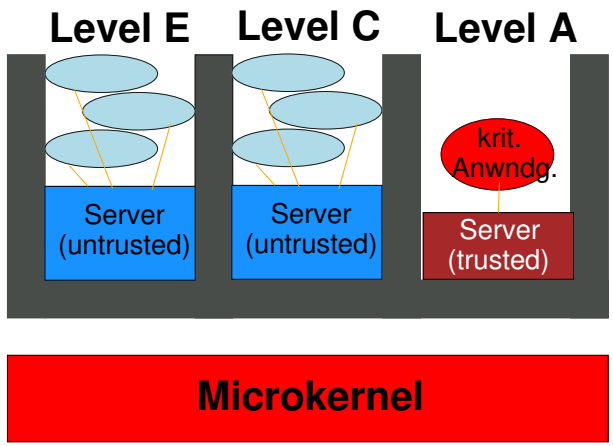
Hochschule RheinMain

Codecomplexität und Sicherheitsklassifizierung gegeneinander abwägen

- MILS-Architektur[2]
- Sonderfall: Mikrokern- Prozesse als „Container“ für Betriebssysteme

⇒ Virtualisierung (vgl. Xen, VMware, Virtualbox)

⇒ Virtual Machine Monitore, Hypervisor: Spezielle Formen von Mikrokernen



[2] Multiple Independent Levels of Safety / Security

Interne Architektur von BS-Kernen

Aufgaben eines Echtzeit-Betriebssystems

- Verwaltung von Rechenprozessen und Betriebsmitteln unter Erfüllung der Forderungen nach Rechtzeitigkeit, Gleichzeitigkeit und Effizienz

Betriebssystemfunktionen:

- Organisation des Ablaufs der Rechenprozesse (Scheduling)
- Organisation der Interruptverwaltung
- Organisation der Speicherverwaltung
- Organisation der Ein-/Ausgabe
- Organisation des Ablaufs bei irregulären Betriebszuständen und des (Wieder-) Anlaufs

Rechenprozess-Verwaltung

Arten von Rechenprozessen

- Anwenderprozesse
- Systemprozesse:
  - zentrale Protokollierung
  - Verwaltung von Speichermedien
  - Netzwerk-Protokollabwicklung
  - Idle-Prozess

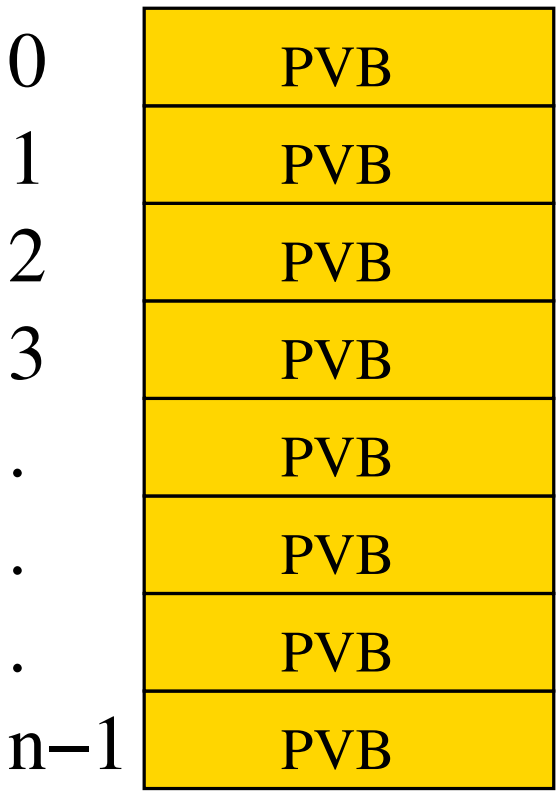
Aufgaben bei der Rechenprozess-Verwaltung

- Koordinierung des Ablaufs von Anwender- und Systemprozessen
- Parallelbetrieb möglichst vieler Betriebsmittel
- Abarbeitung von Warteschlangen bei Betriebsmitteln
- Synchronisierung von Anwender-Systemprozessen

Datenstrukturen zur Prozessverwaltung (1)

Hochschule RheinMain

Prozesstabelle



- Liste der existierenden Rechenprozesse
- Elemente der Liste:
  Prozessverwaltungsblock: PVB
- auch genannt:
  - PCB – *Process Control Block*
  - TCB – *Task Control Block/ Thead Control Block*

Datenstrukturen zur Prozessverwaltung (2)

Prozessverwaltungsblock – PVB

- Dient zum Speichern des gesamten Zustandes eines Prozesses

Prozessverwaltung	**Speicherverwaltung**	**evtl. Dateisystem**
Register	Zeiger auf Speichersegmente	Wurzelverzeichnis
Programmzähler	Belegtliste	aktuelles Verzeichnis
Programmstatuswort	Nachrichtenpuffer	offene Dateideskriptoren
Stack-Zeiger	Zugriffsrechte	Aufrufparameter
Prozesszustand	verschiedene Flags	verschiedene Flags
Prozessnummer		
Prozesserzeugungszeitpunkt		
Terminierungsstatus		
verbrauchte Prozessorzeit		
Alarm-Zeitpunkt		
Signalstatus		
Zeiger auf Nachrichten		
verschiedene Flags		

zusätzlich: Zeiger zur Verkettung in Warteschlangen

Datenstrukturen zur Prozessverwaltung (2)

Prozessverwaltungsblock – PVB

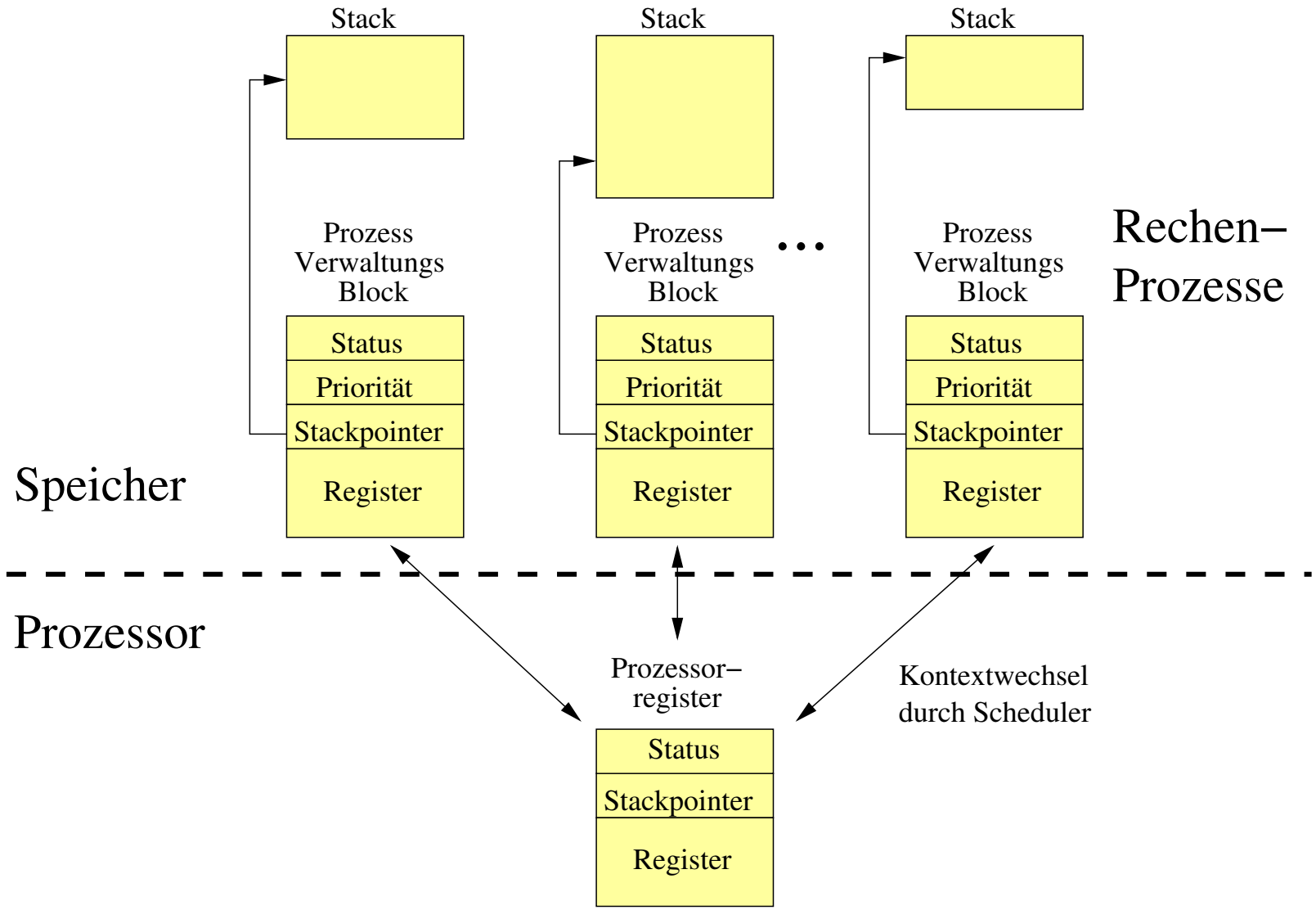
- Dient zum Speichern des gesamten Zustandes eines Prozesses

Prozessverwaltung	**Speicherverwaltung**	**evtl. Dateisystem**
Register	Zeiger auf Speichersegmente	Wurzelverzeichnis
Programmzähler	Belegtliste	aktuelles Verzeichnis
Programmstatuswort	Nachrichtenpuffer	offene Dateideskriptoren
Stack-Zeiger	Zugriffsrechte	Aufrufparameter
Prozesszustand	verschiedene Flags	verschiedene Flags
Prozessnummer		
Prozesserzeugungszeitpunkt		
Terminierungsstatus		
verbrauchte Prozessorzeit		
Alarm-Zeitpunkt		
Signalstatus		
Zeiger auf Nachrichten		
verschiedene Flags		

zusätzlich: Zeiger zur Verkettung in Warteschlangen

Mehrprozessbetrieb

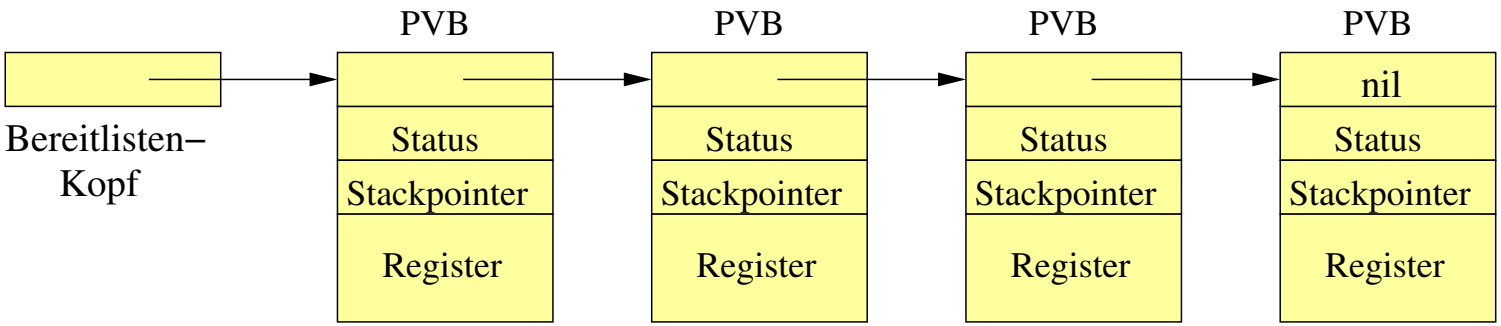
Hochschule RheinMain



Warteschlangenstruktur (1)

Hochschule RheinMain

- Einfache Struktur der Liste der rechenwilligen Prozesse (Bereit-Liste oder *Ready Queue*):



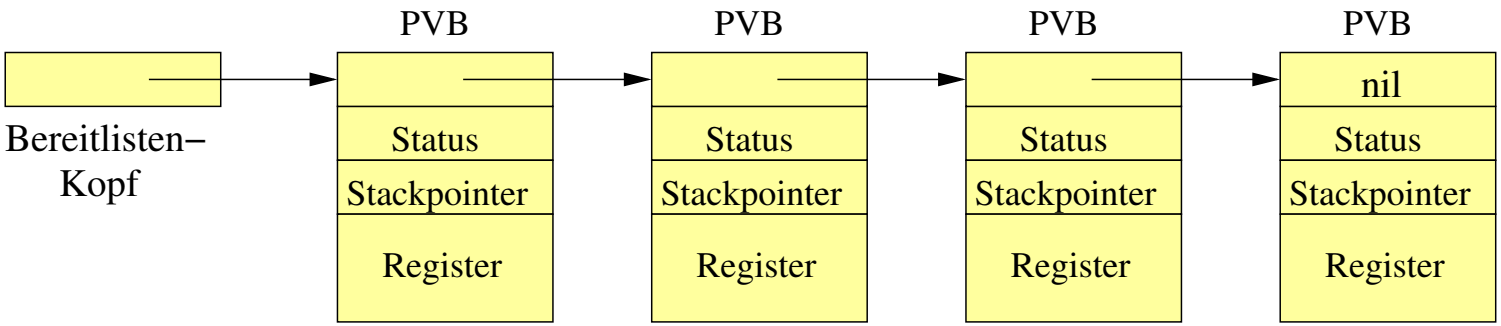
- Scheduler entnimmt vordersten Rechenprozess und teilt ihm den Prozessor zu
- Ankommende Rechenprozesse werden am Ende angefügt

$\Rightarrow$ FIFO-Scheduler

Warteschlangenstruktur (2)

Hochschule RheinMain

- Einfache Struktur der Liste der rechenwilligen Prozesse (Bereit-Liste oder *Ready Queue*):



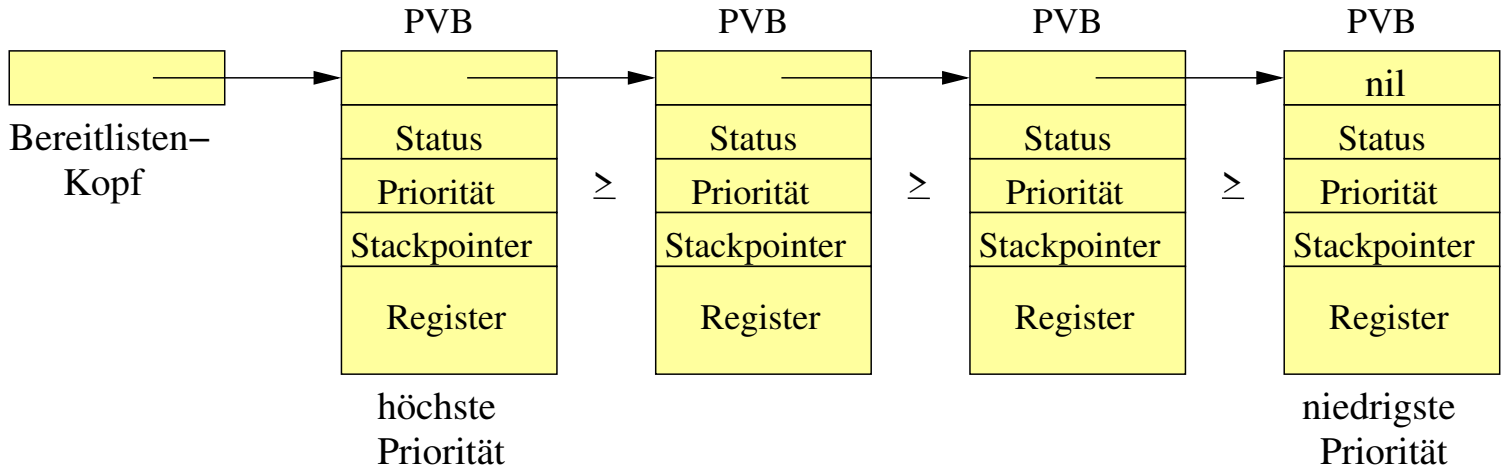
- Nach Ablauf einer Zeitscheibe wird der laufende Rechenprozess unterbrochen und am Ende angefügt
- Ansonsten unverändert

$\Rightarrow$ Round-Robin-Scheduler

Warteschlangenstruktur (3)

Hochschule RheinMain

- Einfache Struktur der Liste der rechenwilligen Prozesse (Bereit-Liste oder *Ready Queue*):



- Ankommende Rechenprozesse werden anhand ihrer Priorität in die Liste eingefügt (Bei gleicher Priorität: FIFO-Reihenfolge)

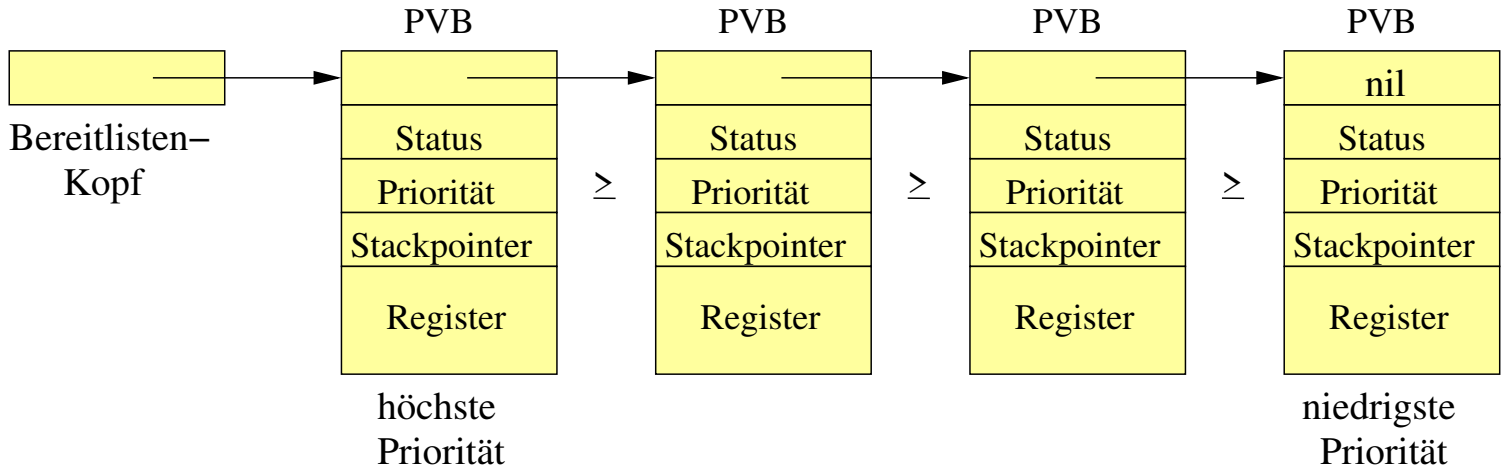
⇒ Prioritäts-Scheduler

- Problem: Laufzeitaufwand beim einsortieren ist nicht konstant (hängt von der Länge der Liste ab) → Komplexität O(n)

Warteschlangenstruktur (3)

Hochschule RheinMain

- Einfache Struktur der Liste der rechenwilligen Prozesse (Bereit-Liste oder *Ready Queue*):



- Ankommende Rechenprozesse werden anhand ihrer Priorität in die Liste eingefügt (Bei gleicher Priorität: FIFO-Reihenfolge)

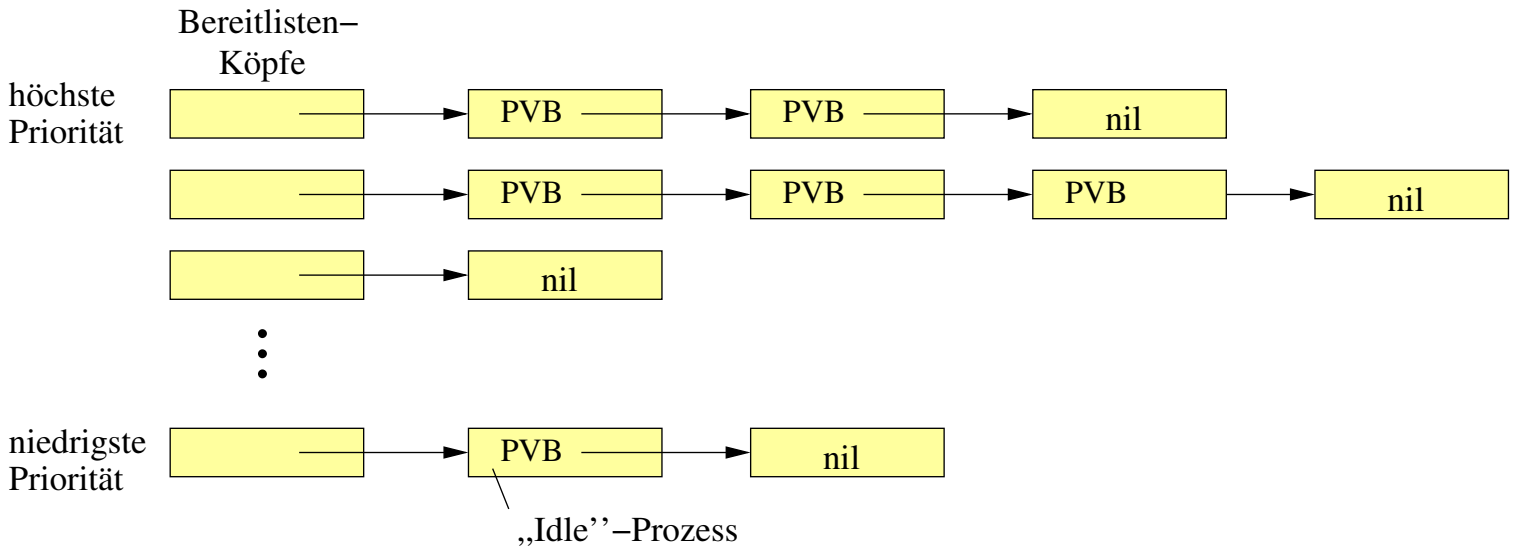
⇒ Prioritäts-Scheduler

- Problem: Laufzeitaufwand beim einsortieren ist nicht konstant (hängt von der Länge der Liste ab) → Komplexität O(n)

Warteschlangenstruktur (4)

Hochschule RheinMain

- (Für Echtzeitbetriebssysteme) typische Struktur der Bereit-Liste:

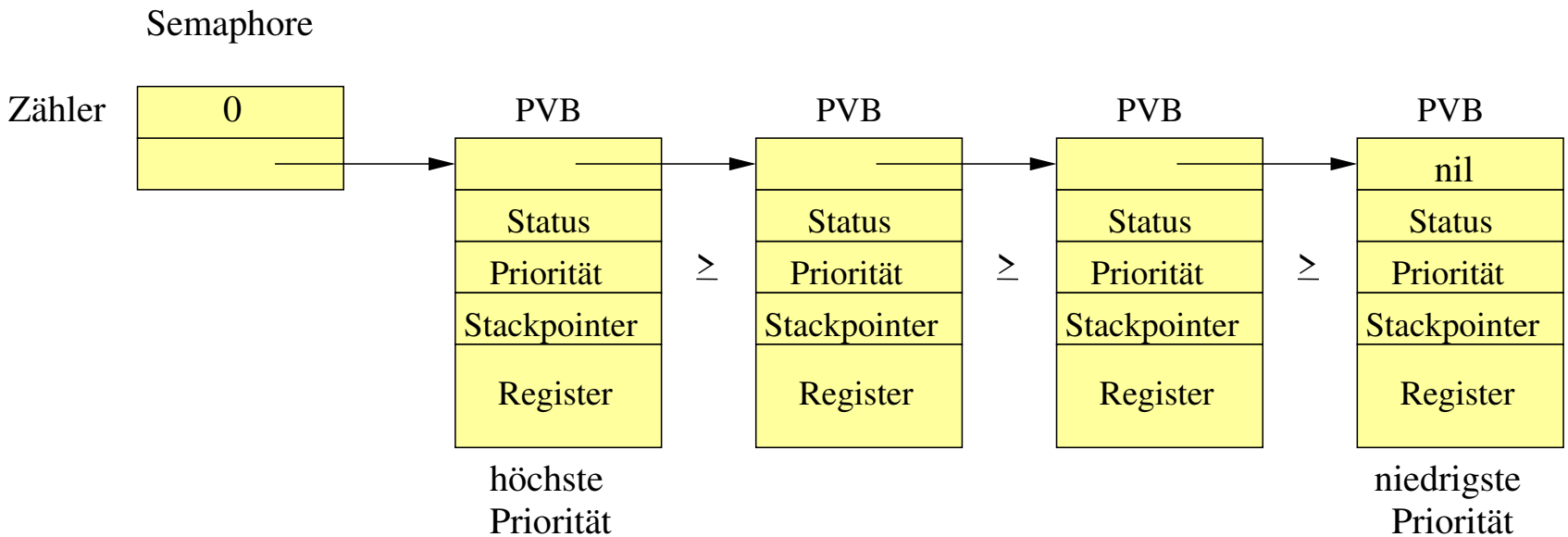


- Eine Bereitliste pro Prioritätsstufe
- Vorteil: Konstanter Laufzeitaufwand beim Einreihen („O(1)-Scheduler“)
- Nachteil: Feste Anzahl möglicher Prioritäten (typisch: 32-256)

Verwaltung blockierter Rechenprozesse (1)

Hochschule RheinMain

- z.B. Warten auf einen Semaphor
- Semaphor besteht aus Zähler und Wartelistenkopf



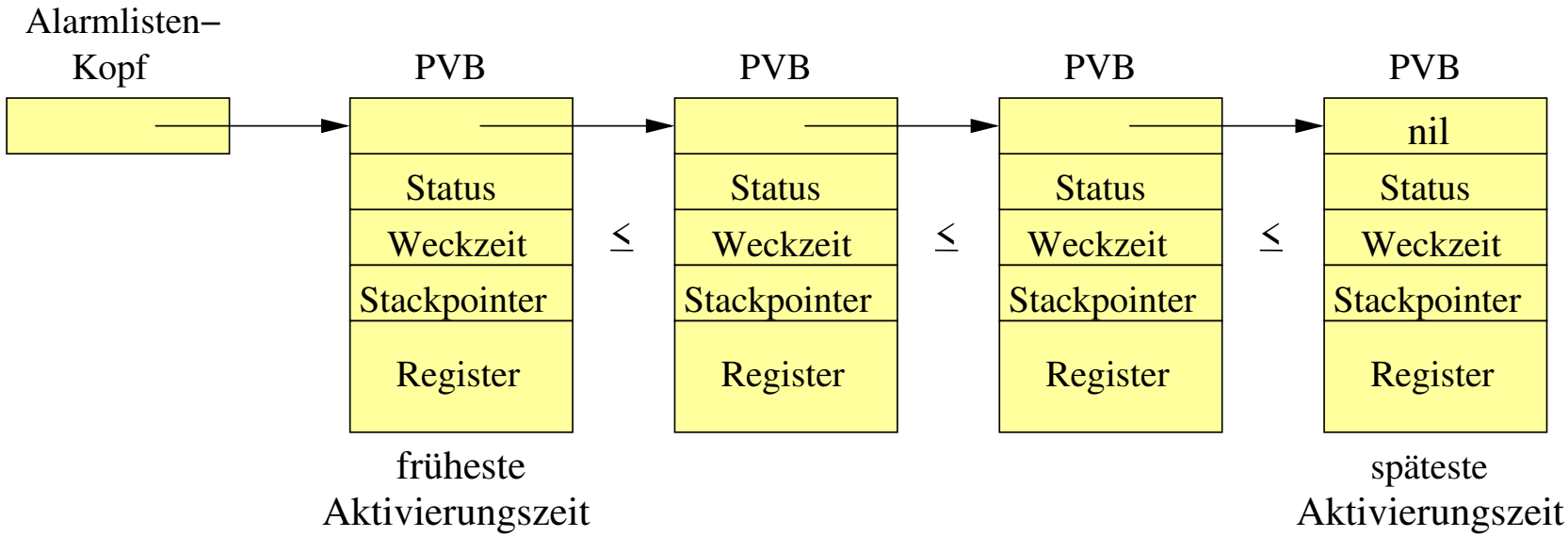
- Rechenprozesse sind entweder rechenwillig oder blockiert

→ Können in gleicher Weise wie in Bereitliste verkettet werden

- Einreihen nach Priorität oder FIFO-Reihenfolge möglich

Verwaltung blockierter Rechenprozesse (2)

- z.B. Zeitbegrenztes Warten (*timed sleep*)



- Einreihen nach wachsender (absoluter) Weckzeit
- Interrupt-Service Routine der Hardware-Uhr muss stets nur den vordersten Eintrag prüfen (O(1)!)

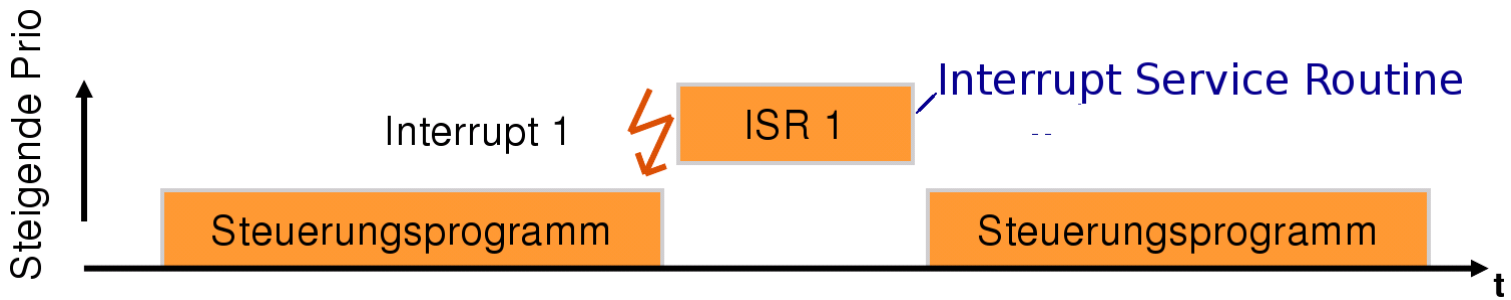
Interrupt-Verwaltung

Hochschule RheinMain

- Unterbrechung des geplanten Programmablaufs
- Beauftragung einer Behandlungsroutine (ISR)

Geplanter Programmablauf: (ohne Interrupt)

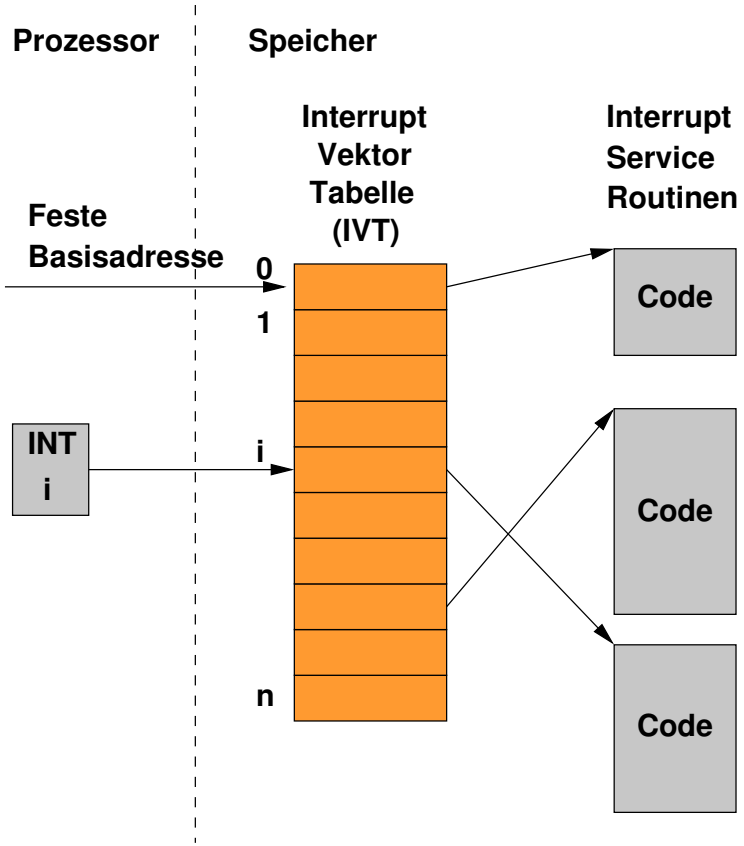
Tatsächlicher Ablauf: (mit Interrupt)



Bestimmung der ISR

Hochschule RheinMain

Typisch:



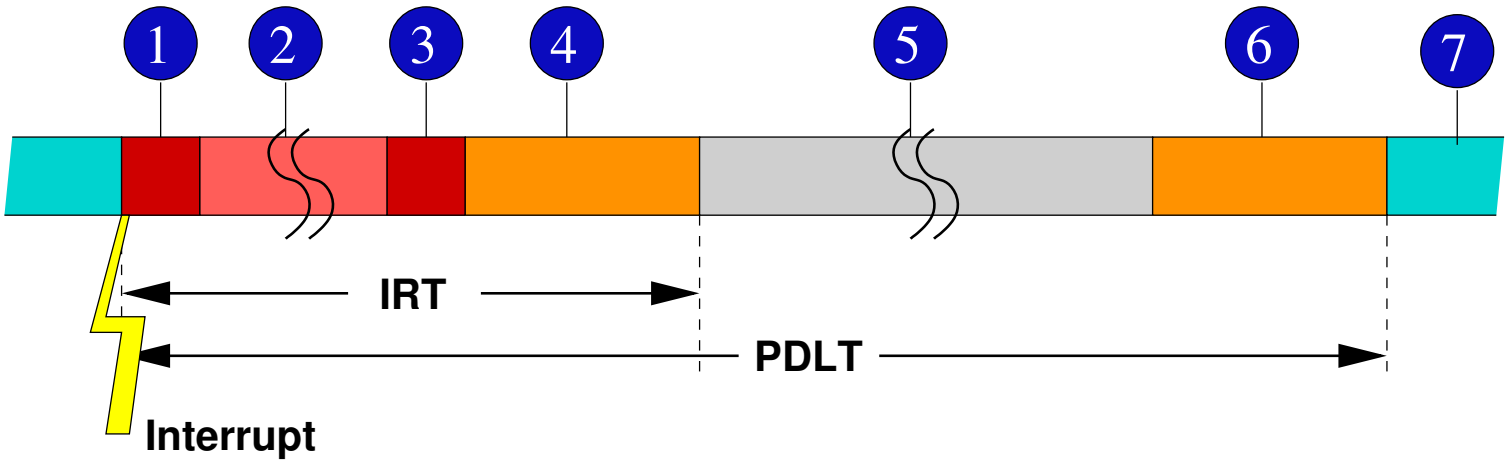
- Interrupt Service Routine (ISR): Gerätespezifische Routine zur Behandlung von Interrupts
- Zuordnung von Interrupt(-Nummer) zu ISR über *Interrupt-Vektor-Tabelle* (IVT)

Echtzeitbezogene Kenngrößen

Wichtige Kenngrößen zur Beurteilung der Echtzeitfähigkeit:

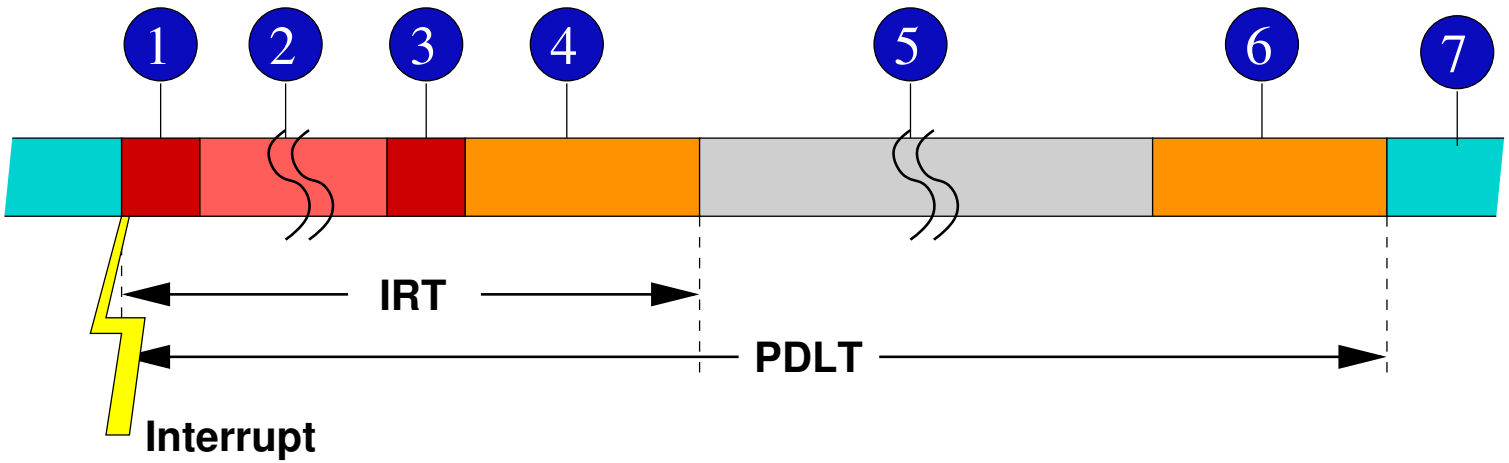
- Interrupt Latency Time
- Interrupt Response Time
- Interrupt Cycle Time
- Process Dispatch Latency Time

Detaillierter Ablauf der Interruptbehandlung (1)



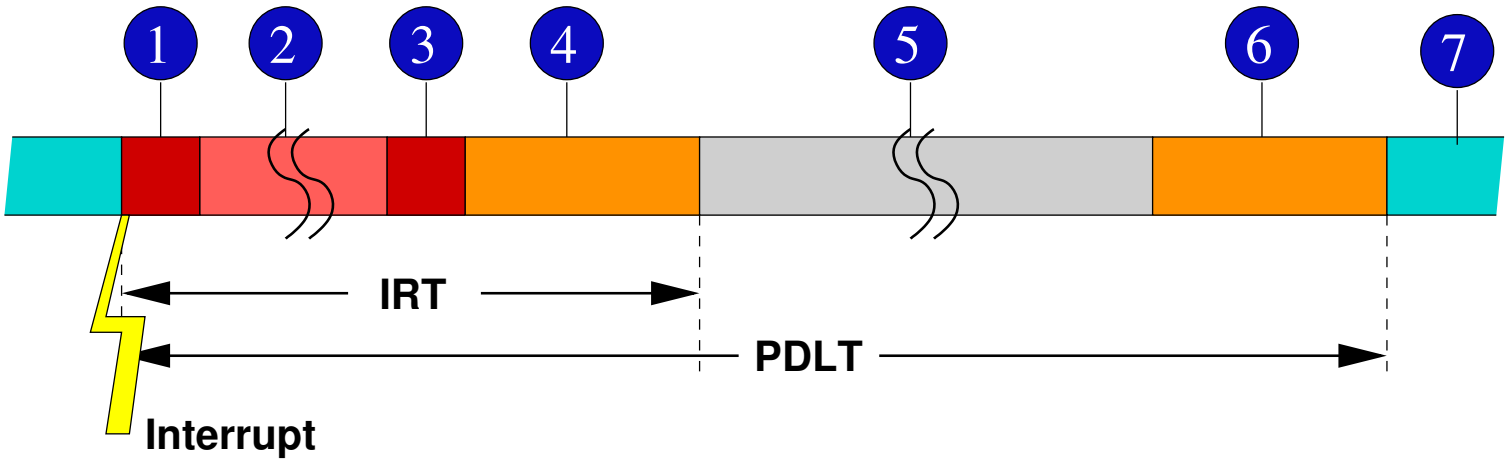
1. Setzen der Interrupt-Anforderung durch die Hardware (HW-Verzögerung).
2. Warten auf Freigabe einer ev. Interruptsperre, ev. Abwarten der Bearbeitung aller höher prioren Interrupts, Arbitrierung bei Anliegen mehrerer Interrupts.
3. Beenden der Ausführung des aktuellen Befehls.

Detaillierter Ablauf der Interruptbehandlung (2)



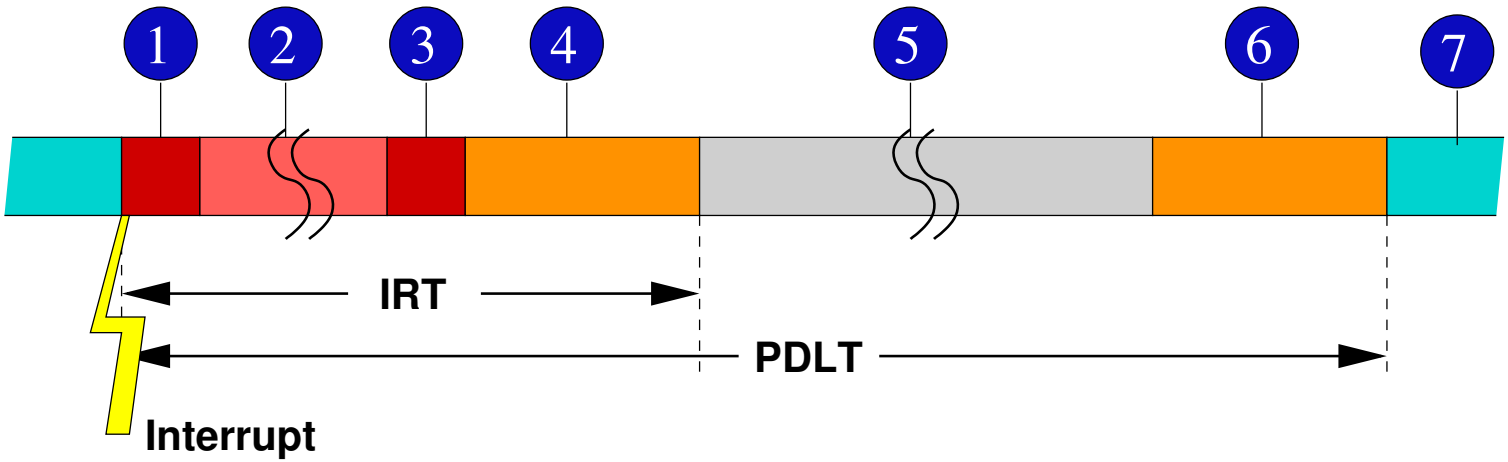
4. Kontextwechsel
   - Retten Befehlszähler und Statusregister
   - Bestimmung der zugehörigen Interrupt Service Routine (ISR).
   - Verzweigen in die Interrupt Service Routine

Detaillierter Ablauf der Interruptbehandlung (3)



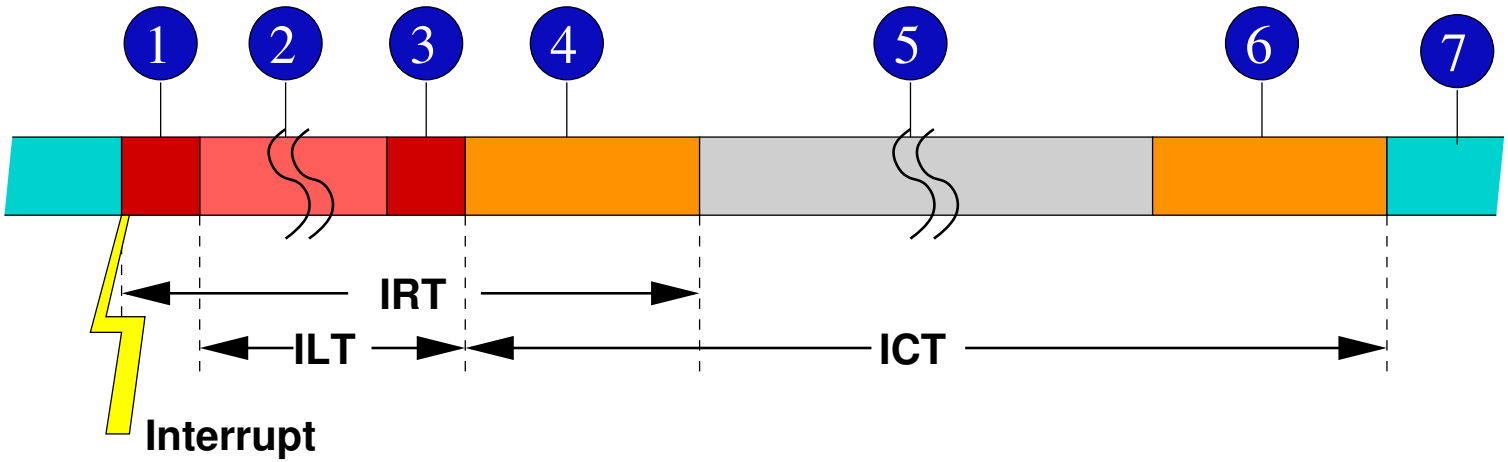
5. Ausführen der Interrupt Service Routine (Interrupt Handler)
   - Retten der Umgebung der unterbrochenen Prozesses, soweit dessen Betriebsmittel benötigt werden (häufig vollständiger Registersatz)
   - ev. Aufbau einer für das Betriebssystem notwendigen Interrupt-Bearbeitungsumgebung
   - Eigentlichen ISR-Code ausführen.
   - Wiederherstellen der Umgebung des unterbrochenen Prozesses.

Detaillierter Ablauf der Interruptbehandlung (4)



6. Rückkehr aus der Interruptbehandlung RETI (Befehlszähler, PSW laden)
7. Fortsetzung des unterbrochenen Rechenprozesses

Kenngrößen bei der Interruptbehandlung



- IRT (*interrupt response time*): Zeit vom Eintreten des Interrupts bis zu Beginn der Ausführung der ISR
- ILT (*interrupt latency time*): Zeit für das Warten auf Interruptfreigabe (2) und Beenden des aktuellen Befehls (3)
- ICT (*interrupt cycle time*): Gesamtdauer der Bearbeitung des Interrupts

Interrupt Response Time (IRT)

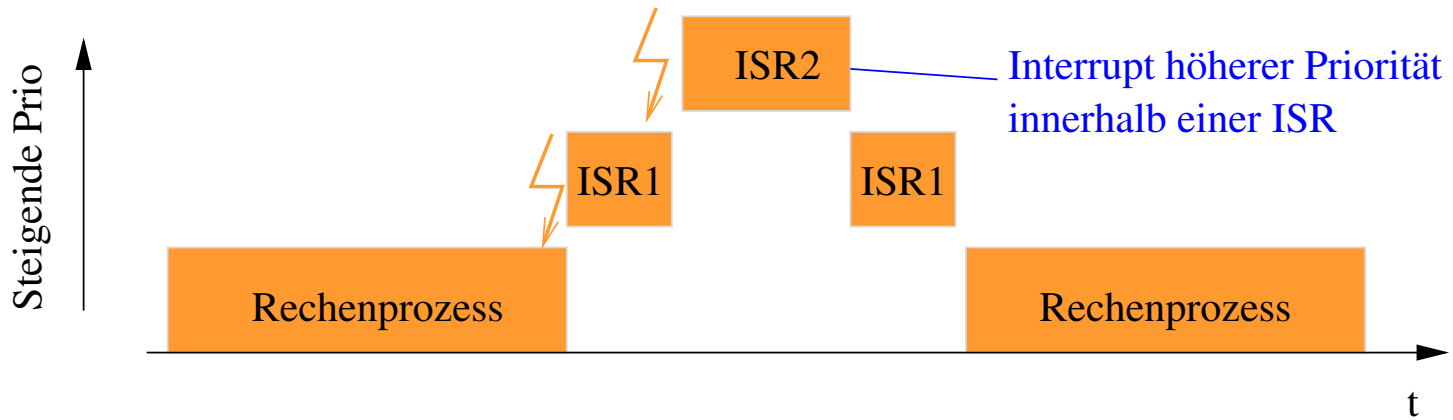
- Typisches Merkmal zur Beschreibung der Reaktionsfähigkeit
- Typisch 1 – 100 μs,
- hängt stark von Prozessor-Hardware ab
- Vgl. Tabelle Zusammenfassung Produkte

Priorisierte Interrupts

Hochschule RheinMain

Interrupts sind unterschiedlich priorisiert

→ Möglicher Ablauf:

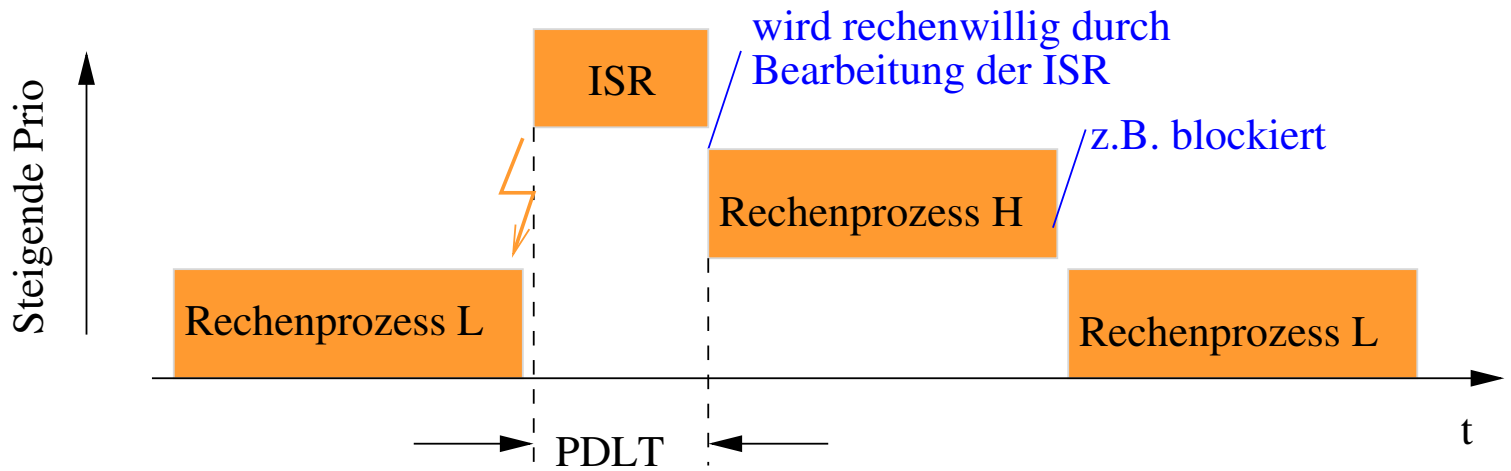


- Maximale Verschachtelungstiefe = Anzahl der Interrupts

Höherpriore Rechenprozesse

Hochschule RheinMain

ISR setzt höherprioren, zuvor blockierten Prozess rechenwillig



- Rechenprozess wird unmittelbar nach der ISR gestartet
- Scheduler-Aufruf am Ende der ISR

Weitere wichtige Kenngröße:

- PDLT (*process dispatch latency time*): Zeit vom Eintreten des Interrupts bis zu Beginn des aktivierten Rechenprozesses

Auswirkung von Interrupt-Sperren

- Interrupt-Anforderungen können jederzeit auftreten.
- Interrupt-Sperren werden durch spezielle Maschinenbefehle realisiert.
- Interrupt-Sperren dienen der Vermeidung von Inkonsistenzen von Daten, die gemeinsam von unterbrochener Aktivität und Interrupt-Handler bearbeitet werden.
- Nachteile von Interrupt-Sperren:
  - Interrupt-Antwortzeit (IRT) wächst!
    (worst case: um die Dauer der längsten Interrupt-Sperre).
  - Dadurch sinkt schnelle Reaktionsfähigkeit.
  - Interrupt-Sperren können periodische Aktivierung von Tasks bzw. Handlern verzögern (z.B. Störung von Regelalgorithmen).

Auslagern der Interrupt-Bearbeitung

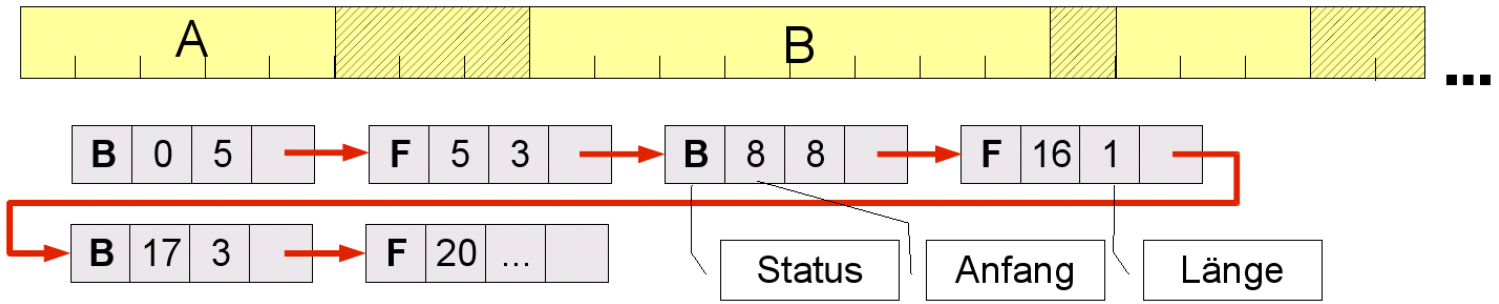
- Für die Dauer der Interrupt-Bearbeitung werden i.d.R. Interrupts gleicher oder niedrigerer Priorität nicht sichtbar (bleiben maskiert).
- Um entsprechende Ereignisse möglichst früh wahrnehmen zu können, ist es wünschenswert, die Interrupt-Bearbeitung durch Auslagern wesentlicher Aktivitäten zu verkürzen.
- Vorgehensweise:
- In der ISR nur das unbedingt Notwendige tun.
  - Aktivität außerhalb der ISR einleiten
  - (z.B. Auftragserteilung an einen Thread (ISR Thread, Kernel Thread, „Deferred Procedure call“ (DPC)),
  - Event oder Message an Anwenderprozess zustellen, o.ä.).

Speicherverwaltung

Speicher: Je schneller desto teurer

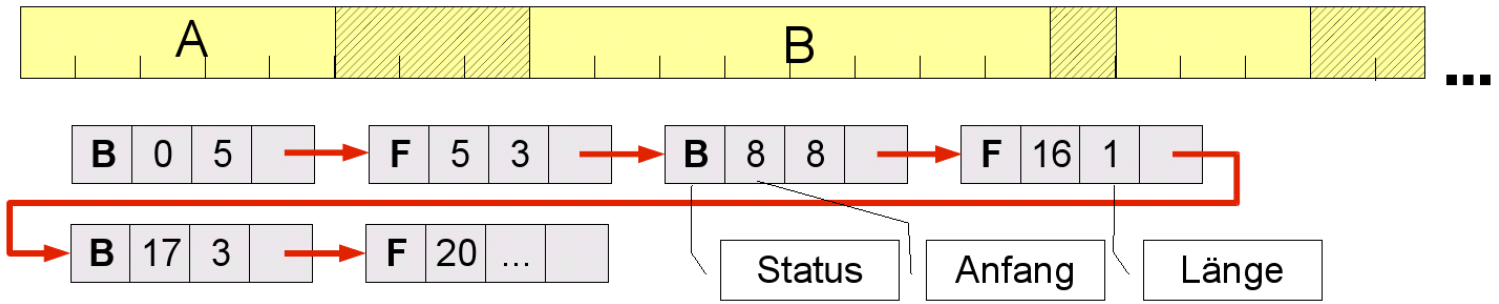
- Speicherhierarchie-Ebenen
  - Cache-Speicher (besonders schneller Halbleiterspeicher)
  - Arbeitsspeicher
  - Plattenspeicher
  - Backup-Speicher (z.B. Magnetband)
- Aufgaben der Speicherverwaltung
  - Optimale Ausnutzung der „schnellen“ Speicher
  - Koordinierung des gemeinsamen Zugriffs auf einen Speicherbereich
  - Schutz des Speicherbereichs verschiedener Rechenprozesse gegen Fehlzugriffe
  - Zuweisung von physikalischen Speicheradressen für die logischen Namen in Anwenderprogrammen

Einfache Speicherverwaltung – `malloc()` (1)



- Jedem belegten und jedem freien Speicherbereich wird ein Listenelement zugeordnet.
- Segmente dürfen variabel lang sein.
- Jedes Listenelement enthält Startadresse und Länge des Segments, sowie den Status (B=belegt, F=frei).
- Gefundenes freies Segment wird (falls zu groß) aufgespalten.
- Freigegebenes Segment wird ggf. mit ebenfalls freien Nachbarsegmenten „verschmolzen“

Einfache Speicherverwaltung – `malloc()` (2)



- Die freien Segmente können auch in separater Liste geführt werden („Freiliste“)
- Die Freiliste kann in sich selbst gehalten werden, d.h. in den verwalteten freien Bereichen (→ kein weiterer Speicher nötig).
- Die Segmentliste kann nach Anfangsadressen geordnet sein.
  Vorteil: freiwerdendes Segment kann mit benachbartem freien Bereich zu einem freien Segment „verschmolzen“ werden.
- Freiliste kann alternativ nach der Größe des freien Bereichs geordnet sein.
  Vorteil: Vereinfachung beim Suchen nach einem freien Bereich bestimmter Länge.

Ein-/Ausgabesteuerung

Hochschule RheinMain

Verschiedenartige Typen von Ein-/Ausgabegeräten

- Unterscheidung in Geschwindigkeit
- Unterscheidung in Datenformaten

Realisierung der Ein-/Ausgabesteuerung

- Hardwareunabhängige Ebene für die Datenverwaltung und den Datentransport
- Hardwareabhängige Ebene, die alle gerätespezifischen Eigenschaften berücksichtigt (Treiber-Programme)

Behandlung irregulärer Betriebszustände (1)

Klassifizierung von Fehlern

- fehlerhafte Benutzereingaben: nicht zulässige Eingaben müssen mit Fehlerhinweisen abgelehnt werden.
  Siehe „Sunk by Windows NT"[3]
  „The source of the problem on the USS Yorktown was that bad data was fed into an application running on one of the 16 computers on the LAN. The data contained a zero where it shouldn't have, and when the software attempted to divide by zero, a buffer overrun occurred – crashing the entire network and causing the ship to lose control of its propulsion system"
- fehlerhafte Anwenderprogramme: Gewährleistung, dass ein fehlerhaftes Anwenderprogramm keine Auswirkungen auf andere Programme hat

[3] `http://www.wired.com/science/discoveries/news/1998/07/13987`

Behandlung irregulärer Betriebszustände (1)

Klassifizierung von Fehlern (Forts.)

- Hardwarefehler/-ausfälle:
  - Erkennung von Hardwarefehlern bzw. -ausfällen
  - Rekonfigurierung ohne die fehlerhaften Teile
  - Abschaltsequenzen bei Stromausfällen
- Deadlocks aufgrund dynamischer Konstellationen
  - sichere Vermeidung von Deadlocks ist nicht immer möglich

Der Markt für Echtzeitbetriebssysteme

Kriterien bei der Auswahl von Echtzeit-Betriebssystemen

- Entwicklungs- und Zielumgebung
- Modularität und Kernelgröße
- Leistungsdaten
  - Anzahl von Tasks
  - Prioritätsstufen
  - Taskwechselzeiten
  - Interruptlatenzzeit
- Anpassung an spezielle Zielumgebungen
  - z.B. Betrieb ohne Festplatte
- Allgemeine Eigenschaften
  - Schedulingverfahren
  - Interprozesskommunikation
  - Netzwerkkommunikation
  - Gestaltung Benutzungsoberfläche

Welches Betriebssystem?

- Aufgrund der vielfältigen Anforderungen gibt es nicht **das** Echtzeitbetriebssystem
- Gerade „so viel Betriebssystem, wie nötig“
- Oft ist bereits bei der Konzeption bekannt, wieviel das konkret ist

→ In diesem Fall sollte das Betriebssystem statisch skalierbar sein

- Andererseits muss es auch kostengünstig sein (Auch Software-Lizenzen kosten Geld)

→ So stellt u.U. ein eigentlich zu umfangreiches, nur bedingt echtzeitfähiges, dabei aber sehr kostengünstiges Betriebssystem (z.B. Linux) den wirtschaftlich besseren Kompromiss dar

Echtzeitunterstützung für Linux

Linux RT_PREEMPT patch: www.osadl.org

General road map of the main patch components

Architecture / Feature	x86	x86/64	powerpc	arm	mips	68knommu
Deterministic Scheduler	●	●	●	●	●	●
Preemption Support	●	●	●	●	●	●
PI Mutexes	●	●	●	●	●	●3
High-Resolution Timer	●	●1	●1	●1	●1	●
Preemptive Read-Copy Update	●2	●2	●2	●2	●2	●2
IRQ Threads	●4	●4	●4	●4	●4	●3,4,5
Raw Spinlock Annotation	●6	●6	●6	●6	●6	●6
Forced IRQ Threads	●7	●7	●7	●7	●7	●7
R/W Semaphore Cleanup	●7	●7	●7	●7	●7	●7
Full Realtime Preemption Support	●	●	●	●	●	●3

● Available in mainline Linux

● Available when Realtime Preempt patches applied

https://www.osadl.org/Realtime-Linux.projects-realtime-linux.0.html

Marktübersicht

Hochschule RheinMain

breites Angebot am Markt für alle Klassen:

- es gibt nicht „den“ dominierenden Anbieter
  (stark unterschiedliche Ziel-Hardware und Entwicklungsplattformen bieten außerdem zahlreiche Nischen)
- Auch: breites Preisspektrum
  - Unterscheidung Entwicklungslizenzen
  - Runtime-Lizenzen für Target-Systeme
- Nach einer Umfrage aus 2006[4] wurden 30% der eingebetteten Systeme ohne Betriebssystem realisiert

[4] `http://www.embedded.com/columns/showArticle.jhtml?articleID=187203732`.

Echtzeit-BS Marktübersicht (iX 4/2012)

Hochschule RheinMain

Anbieter von Echtzeitbetriebssystemen

Betriebssystem	Hersteller	Webseite	Echtzeitverhalten[1]	Lizenzmodell
µC/OS-II; µC/OS-III; µC/OS-OSEK; µC/OS-MMU	Micrium (Distributor: Embedded Office)	www.embedded-office.de, www.micrium.com	Hard-RT / 100 bis 120 µs	proprietär, Source-Code wird mitgeliefert
EB tresos	Elektrobit	www.elektrobit.com	k. A.	proprietär
eCosPro	eCosCentric	www.ecoscentric.com	Hard-RT / 0,67 µs	modifizierte GNU GPL
ElinOS	Sysgo AG	www.sysgo.com	wie andere Linux-Systeme	GNU GPL
emBOS	Segger Microcontroller	www.segger.com	Hard-RT / 1 µs (ARM, 200 MHz)	proprietär, Source-Code erhältlich
Euros	Euros – Embedded Systems	www.euros-embedded.com	Hard-RT / 10 µs	proprietär
Integrity	Greenhills Software	www.ghs.com	Hard-RT / je nach Architektur	proprietär
LynxOS, LynxOS-178, LynxOS-SE	LynuxWorks	www.lynuxworks.com	k. A.	proprietä
Microsar	Vector Informatik	www.vector.com	Hard-RT / k. A.	proprietär
Microware OS-9	Radisys (Distributor: Microsys)	www.radisys.com/germany	k. A.	proprietär, Source-Code erhältlich
Monta Vista Linux	Montavista	www.mvista.com	k. A.	GNU GPL
Neutrino	QNX Software Systems GmbH & Co. KG	www.qnx.com/company/germany	Hard-RT / 0,5 bis 2,6 µs z.B. ARM Cortex: 0,5 µs)	proprietär, Teile des Source-Codes sind offen
Nucleus	Mentor Graphics Deutschland GmbH	www.mentor.com/germany	Hard-RT/ ja	proprietär
OSE, OSEck	Enea	www.enea.de	k. A.	proprietär
PikeOS	Sysgo AG	www.sysgo.com	Hard-RT / < 1 µs	proprietär
Realtime Linux (OSADL recommends 2.6.33.7.2-rt30)	OSADL	www.osadl.org/Realtime-Linux.projects-realtime-linux.0.html	Hard-RT / max. 100 000 Taktzyklen (z. B. 1-GHz-CPU: 100 µs)	GNU GPL v2
Red Hat Enterprise MRG	Red Hat	www.redhat.com/mrg/	Soft-RT / 8 µs	GNU GPL
RMOS3	Siemens AG	www.siemens.de/rmos3	Hard-RT / 10 µs	proprietär
RTA-OSECK3, RTA-OS3.0	ETAS	www.etas.com/de	Hard-RT / Autosar-4-0-konform	proprietär
RTOS-32	On Time Software	www.on-time.com	Hard-RT / < 5 µs	proprietär, Source-Code erhältlich
RTOS-UH	IEP	www.iep.de	Hard-RT / 1 µs (1 GHz Power-PC)	proprietär
SMX RTOS	Micro Digital (Distributor: Embedded Tools)	www.smxrtos.com, www.embedded-tools.de	Hard-RT / 78 Taktzyklen (z.B. 0,4 µs auf ARM 200 MHz)	proprietär
SUSE Linux Enterprise Real Time	SUSE Linux GmbH	www.suse.com/de-de/products/realtime/	Hard-RT / 10 bis 100 µs	GNU GPL
Symobi; µnOS	Miray Software	www.miray.de, www.symobi.com	Hard-RT / ARM PXA-320: 0,2 ms	proprietär
Thread X	Express Logic	www.expresslogic.de, www.rtos.com	k. A.	proprietär, Source-Code wird mitgeliefert
VxWorks	Wind River Systems	www.windriver.com/de	Hard-RT / < 10µs	proprietär
Wind River Linux	Wind River Systems	www.windriver.com/de	wie andere Linux-Systeme	k. A.
Windows Embedded Compact 7	Microsoft	www.microsoft.com/windowsembedded	k. A.	proprietär

Alle Daten sind Herstellerangaben, [1]Definition Hard- und Soft-RT siehe Text, dahinter minimale garantierte Antwortzeiten. Die tatsächlich erreichbaren Werte hängen stark von der Kombination Hard- und Software ab.

Freie Produkte (iX 4/2012)

Hochschule RheinMain

Auswahl freier Echtzeitprojekte

Projekt	Webseite	Lizenz
Atomthreads	atomthreads.com	BSD-Lizenz
ecos	ecos.sourceware.org	modified GNU GPL
EmboX	code.google.com/p/embox/	BSD
FreeOSEK	opensek.sourceforge.net	GPLv3
freeRTOS	www.freertos.org	modified GNU GPL
Realtime for Debian	debian.pengutronix.de	GNU GPL
RTLinuxFree	www.rtlinuxfree.com	GNU GPL
TinyOS	www.tinyos.net	BSD
Ubuntu RealTime-Erweiterung	wiki.ubuntu.com/RealTime	GNU GPL
Xenomai	www.xenomai.org	GPLv2

Beispiele von Echtzeitbetriebssystemen

Gliederung

1. AUTOSAR/OSEK
2. POSIX

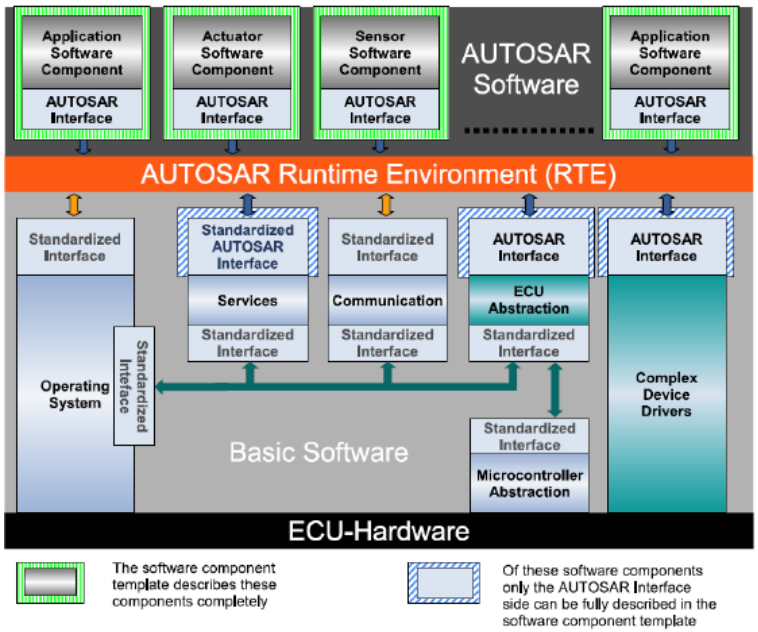
AUTOSAR

- AUTOSAR: AUTomotive Open System ARchitecture
- Weltweiter Zusammenschluss von Automobilherstellern und -zulieferern
- Ziel: Standardisierung einer Software-Architektur für Automotive-Systeme
- (Mittlerweile) zwei Plattformen:
  - *Classic:* Statische Laufzeitumgebung (Runtime Environment, RTE) auf Basis OSEK (s.u.) für „klassische“ Steuerungs- und Regelungsanwendungen
  - *Adaptive:* Dynamische Laufzeitumgebung auf Basis des POSIX-Standards (s.u.) für „neuere“ Anwendungen (z.B. autonomes Fahren)
- Konsortium beschliesst Standards, Softwarehersteller sind aufgefordert, diese zu implementieren

AUTOSAR Architektur

Hochschule RheinMain

- AUTOSAR RTE („Run Time Environment“): Schnittstelle für (ggf. verteilte) Applikationen
- Ortstransparenz durch Sicht als: „Virtual Function Bus“

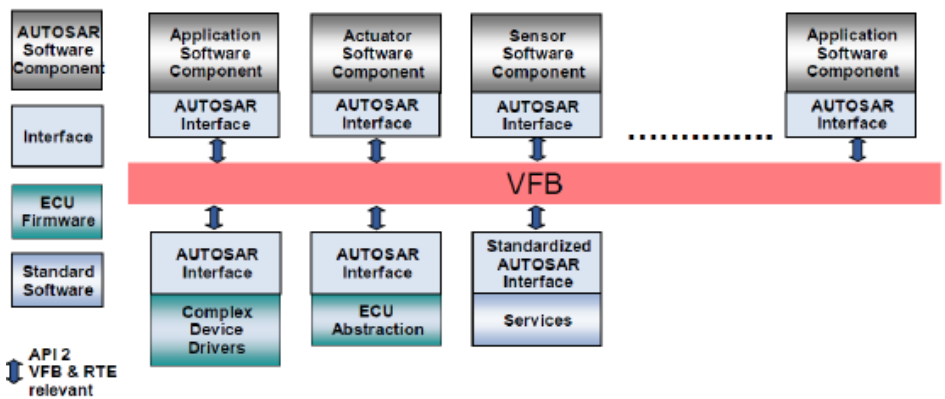


Quelle: Nico Naumann: „AUTOSAR Runtime Environment and Virtual Function Bus“

https://hpi.de/fileadmin/hpi/FG...AUTOSAR0809/NicoNaumann_RTE_VFB.pdf

AUTOSAR Architektur

- AUTOSAR RTE („Run Time Environment“): Schnittstelle für (ggf. verteilte) Applikationen
- Ortstransparenz durch Sicht als: „Virtual Function Bus“



Quelle: Nico Naumann: „AUTOSAR Runtime Environment and Virtual Function Bus“

https://hpi.de/fileadmin/hpi/FG...AUTOSAR0809/NicoNaumann_RTE_VFB.pdf

OSEK-OS

- OSEK: Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug
- Ursprung: Franz. VDX-Initiative 1988, verschmolzen mit deutschem OSEK-Konsortium in 1994
- OSEK OS: Spezifikation für Echtzeitbetriebssysteme, i.w. für Bereich Automotive
- Offener Standard seit 1997, Internationaler Standard ISO 17356-3 seit 2005
- Es existiert eine Vielzahl an Produkten und auch freien Implementierungen nach OSEK-Standard:
  - Arctic Core `https://www.arccore.com/`
  - Erika Enterprise `http://erika.tuxfamily.org/drupal/`
  - FreeOSEK `https://github.com/ciaa/Firmware/`
  - nxtOSEK (für Lego Mindstorms) `https://en.wikipedia.org/wiki/NxtOSEK`
  - ....
- OSEK-Spezifikation 2003 vom AUTOSAR-Konsortium übernommen (s.o.)

Grundidee: statisches System

Hochschule RheinMain

- Dynamisches Erzeugen/Verwerfen von Objekten ist nicht deterministisch

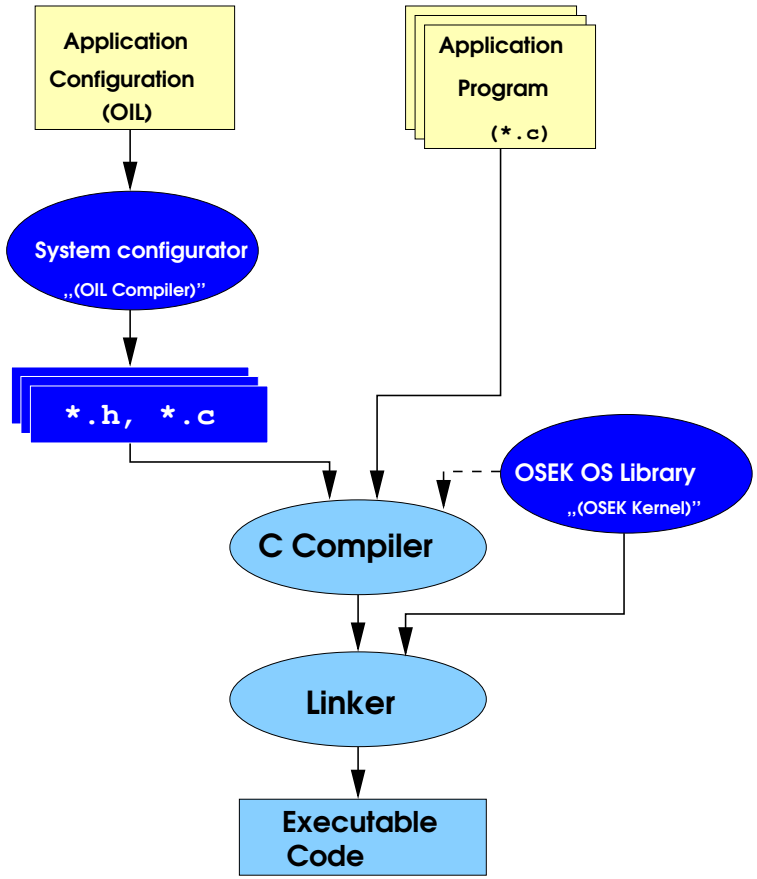
→ **Keine** Funktionen zum Allokieren bzw. Verwerfen von Objekten zur Laufzeit

- Dynamisches Ändern von Taskprioritäten in Verbindung mit Resource Locking kann zu Deadlocks führen

→ **Kein** Ändern von Prioritäten zur Laufzeit

→ Alle Objekte (Tasks, Events, Timers, etc.) und deren Parameter müssen zur Konfigurationszeit deklariert werden

- Spezielle Sprache dazu: *OIL*[5])



[5] OSEK Implementation Language

Skalierbarkeit (1)

- OSEK muss auf einem weiten Bereich von Plattformen einsetzbar sein (8-bit - 64-bit)

→ Der Standard definiert vier „Konformanzklassen“ (Untermengen der Scheduler-Funktionalität)

- Validierung von Funktionsargumenten zur Laufzeit kostet Rechenleistung, ist aber während Entwicklung/Test unverzichtbar

→ Der Standard definiert zwei „Error Checking Levels“

Einige OSEK Implementierungen erreichen weniger als 1kB Codegröße.

Skalierbarkeit (1)

- OSEK muss auf einem weiten Bereich von Plattformen einsetzbar sein (8-bit - 64-bit)

→ Der Standard definiert vier „Konformanzklassen“ (Untermengen der Scheduler-Funktionalität)

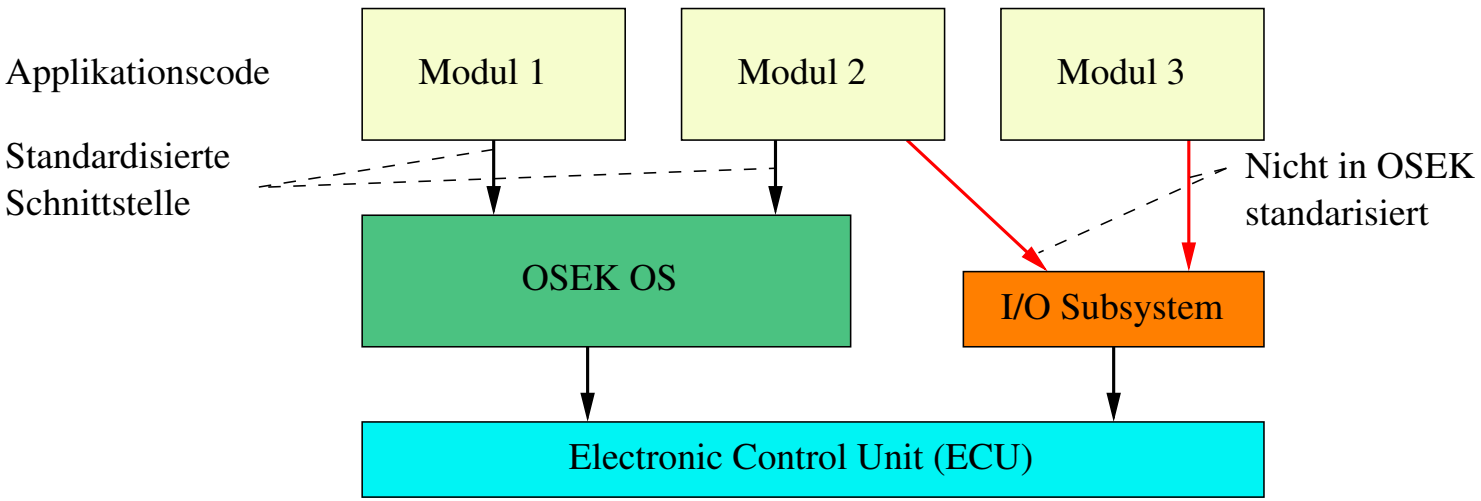
- Validierung von Funktionsargumenten zur Laufzeit kostet Rechenleistung, ist aber während Entwicklung/Test unverzichtbar

→ Der Standard definiert zwei „Error Checking Levels“

Einige OSEK Implementierungen erreichen weniger als 1kB Codegröße.

Skalierbarkeit (2)

- Plattformspezifische Dinge (z.B.) I/O sind nicht im OSEK OS Standard enthalten
  (Classic AUTOSAR definiert „Complex Device Drivers“)

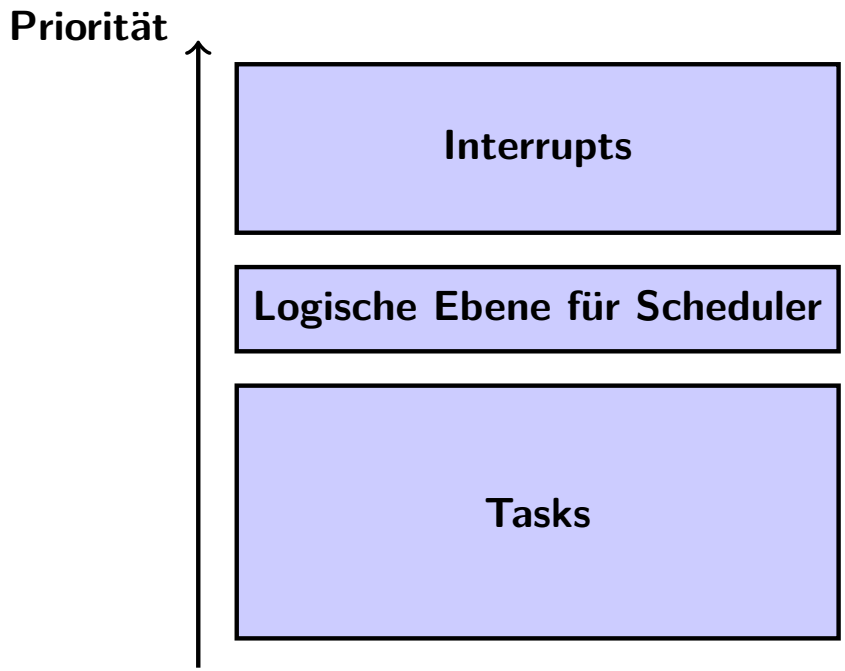


OSEK OS Processing Levels

Hochschule RheinMain

OSEK definiert drei Ebenen:

1. Interrupt
2. Scheduler
3. Tasks



OSEK OS Funktionsgruppen

Taskverwaltung

- Aktivierung/Terminierung von Tasks
- Taskstatus Verwaltung, Taskumschaltung

Synchronisation

- Ressourcenverwaltung
- Eventsteuerung

Interrupts

- Dienste zur Interruptverwaltung

Alarme

- relative und absolute Alarme

Intra-Prozessor Messages

- Datenaustausch zwischen Tasks
- (Inter-Prozessor Messages: OSEK COM)
- In AUTOSAR RTE enthalten

Fehlerbehandlung

- „Hook“-Funktionen

OSEK OS Task Konzept (1)

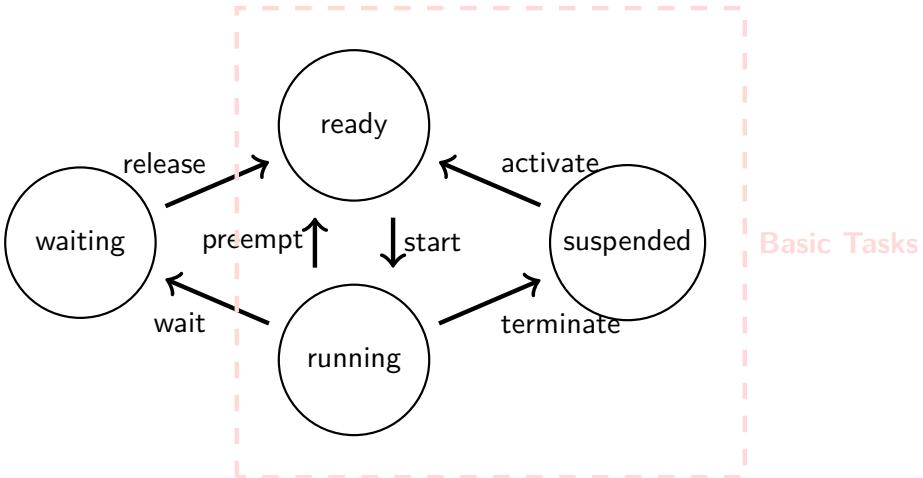
Hochschule RheinMain

Basic Tasks: geben die Kontrolle nur ab, wenn

- sie terminieren
- eine höherpriore Task rechenwillig wird
- ein Interrupt auftritt (→ Interrupthandler wird aktiviert)

Extended Tasks:

- kennen zusätzlich den Zustand „warten“ (auf ein Event)



OSEK OS Task Konzept (1)

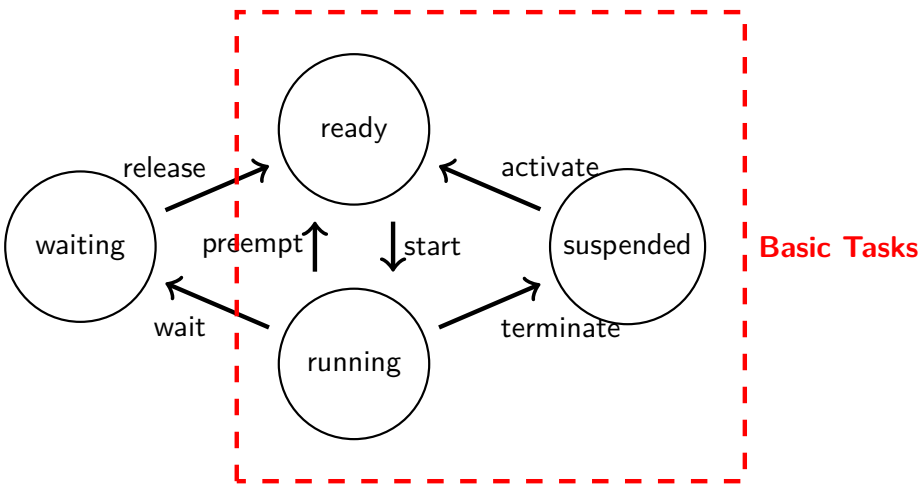
Hochschule RheinMain

Basic Tasks: geben die Kontrolle nur ab, wenn

- sie terminieren
- eine höherpriore Task rechenwillig wird
- ein Interrupt auftritt (→ Interrupthandler wird aktiviert)

Extended Tasks:

- kennen zusätzlich den Zustand „warten“ (auf ein Event)



OSEK OS Task Konzept (2)

Aktivierung einer Task mit `ActivateTask()` oder `ChainTask()`

- Je nach Konformanzklasse sind Mehrfach-Aktivierungen zulässig (werden nach Priorität bearbeitet)

Der Scheduler wird als Ressource betrachtet

- Tasks können durch Reservieren der Scheduler-Ressource verhindern, dass sie von anderen Tasks verdrängt werden

Prioritätenbasiertes Scheduling

- 0 = niedrigste Priorität
- Je nach Konformanzklasse eine oder mehrere Tasks je Prioritätsstufe

OSEK OS Task Konzept (3)

Non-preemptive Scheduling: Taskwechsel nur möglich, wenn:

- eine Task beendet wird (mit `TerminateTask()`)
- eine Task beendet wird und eine Nachfolgetask aktiviert (mit `ChainTask()`)
- der Scheduler explizit aufgerufen wird (mit `Schedule()`)
- in den Wartezustand übergegangen wird (mit `WaitEvent()`)

Full-preemptive Scheduling: Verdrängung einer Task durch eine andere (höherpriore) ist jederzeit möglich

- Ausnahme: Task hält die Scheduler-Ressource

Mixed-preemptive Scheduling: Koexistenz von Non-preemptive und Full-preemptive

OSEK OS Interruptbearbeitung

Drei Interrupt-Kategorien:

- Kategorie 1: Interrupthandler verwendet keine OSEK OS Systemdienste → geringster Overhead
- Kategorie 2: Interrupthandler verwendet eine Teilmenge der OSEK OS Systemdienste
- Kategorie 3 (Implementierung optional): wie Kategorie 1, jedoch können nach Aufruf von `EnterISR()` auch OSEK OS Systemdienste verwendet werden wie in Kategorie 2. Dann muss der Handler mit `LeaveISR()` beendet werden.

Sperren/Erlauben von Interrupts:

- bezogen auf die Interruptquelle
- global

OSEK OS Eventmechanismus

Ein „Event“ ist

- ein Mittel zur Task-Synchronisation
- fest einer Extended Task zugeordnet („Eigentümer“)
- bewirkt den Übergang des Eigentümers in den oder aus dem Wartezustand
- eine Extended Task kann Eigentümer mehrerer Events sein
- Basic Tasks kennen keinen Wartezustand → Basic Tasks können nicht Eigentümer eines Events sein

OSEK OS Ressourcenverwaltung (1)

Koordination gleichzeitiger Zugriffe mehrerer Tasks auf gemeinsame Ressourcen

- gegenseitiger Ausschluss: eine Ressource kann immer nur von einer Task belegt sein
- OSEK-Ressourcen verhindern Prioritätsinversion und Deadlocks
- Zugriff auf Ressourcen führt niemals zu einem Wartezustand
- Der Scheduler wird in OSEK als Ressource behandelt: durch Belegen der Scheduler-Ressource kann eine Task ihre Verdrängung (Preemption) durch andere Tasks verhindern

OSEK OS Ressourcenverwaltung (2)

Priority Ceiling Protokoll

- Bei der Systemkonfiguration ist die Gesamtheit der Tasks, die auf diese Ressource zugreifen (und deren Priorität), bekannt

$\Rightarrow$ *Ceiling*-Priorität:

  - $\geq$ höchste aller Prioritäten der Tasks, die auf die Ressource zugreifen
  - $<$ niedrigste aller Prioritäten der Tasks, die nicht auf die Ressource zugreifen, und deren Priorität höher liegt, als die höchstpriore der Tasks, die darauf zugreifen.

- Wenn eine Task eine Ressource beansprucht, und ihre Priorität unter der Ceiling-Priorität liegt, so wird ihre Priorität vorübergehend auf die Ceiling-Priorität angehoben
- Wenn eine Task die Ressource freigibt, fällt ihre Priorität auf den ursprünglichen Wert zurück

OSEK OS Alarme

Alarme dienen zur Behandlung wiederkehrender Ereignisse

- z.B. periodische Timer-Interrupts, Winkelgeber an Kurbelwellen oder Nockenwellen, etc.
- Kopplung der Ereignisquellen an Zähler (Counters) wird vorausgesetzt (nicht im OSEK-Standard spezifiziert)
- In jeder Implementierung existiert mindestens ein Counter (`OS_Counter`), alle weiteren Counter sind implementierungsspezifisch
- Mehrere Alarme können einem gemeinsamen Counter zugeordnet werden
- Jedem Alarm wird statisch (im OIL-File) ein Counter und eine Task zugewiesen

OSEK OS Alarme

Alarme dienen zur Behandlung wiederkehrender Ereignisse

- OSEK OS bietet Dienste zum Aktivieren von Tasks, wenn ein Alarm abläuft, (d.h. wenn ein vorgegebener Zählerwert erreicht wird)
- Es gibt relative und absolute Alarme
- Es gibt einzelne oder zyklische Alarme
- Der Ablauf eines Alarms bewirkt wahlweise das Setzen eines Events oder die Aktivierung einer Task

OSEK OS Messages

Nur lokale (Intra-Prozessor) Messages

- nicht-lokale Messages sind in OSEK COM spezifiziert (AUTOSAR: RTE beinhaltet OSEK OS und OSEK COM)
- *Static length Messages:* Größe in der Systemkonfiguration („OIL-File“) festgelegt
- *Dynamic length Messages:* Größe wird zur Laufzeit festgelegt, Maximalgröße in der Systemkonfiguration
- *Unqueued Messages:* neue Nachrichten überschreiben alte
- *Queued Messages:* Nachrichten werden in FIFO-Reihenfolge abgearbeitet; bei Überlauf geht die zuletzt geschriebene Nachricht verloren

OSEK OS Fehlerbehandlung und Debugging (1)

„Hook“-Routinen

- Durch den Anwender spezifizierbare Routinen, die vom System bei bestimmten Ereignissen aufgerufen werden
- Ausführung erfolgt als Teil des Betriebssystems, mit höherer Priorität als alle Tasks, nicht unterbrechbar durch Interrupts der Kategorien 2 und 3
- Aufrufsyntax und -parameter standardisiert, nicht jedoch die Funktion ($\rightarrow$ i.A. nicht portabel)
- Nur eine Teilmenge der OSEK OS Funktionen darf aus einer Hook-Funktion aufgerufen werden

OSEK OS Fehlerbehandlung und Debugging (2)

„Hook“-Routinen

- `StartupHook()`: Wird beim Start des Systems aufgerufen
- `ShutdownHook()`: Wird beim „Herunterfahren“ des Systems aufgerufen
- `ErrorHook()`: Wird bei Fehlern aufgerufen,
  Unterscheidung zwischen:
  - Applikationsfehler: Angeforderter Systemdienst konnte nicht ausgeführt werden, System ist intakt. Die Hook-Routine muss entscheiden, was zu tun ist (z.B. Shutdown oder Weitermachen)
  - Fatale Fehler: System ist nicht intakt ($\rightarrow$ Shutdown)
- `PreTaskHook()`, `PostTaskHook()`: Werden vor bzw. nach jedem Taskwechsel aufgerufen

POSIX

Hochschule RheinMain

POSIX: Portable Operating System Interface (for unIX)

- Familie internationaler Standards ISO/IEC 9945 ursprünglich spezifiziert durch IEEE Computer Society als IEEE 1003
- Zusammenfassung vieler Teile ab 2008
- Aktuell POSIX.1-2008 = IEEE Std 1003.1-2008, Issue 7, 2016 Edition. `http://pubs.opengroup.org/onlinepubs/9699919799/`
- Üblicherweise API-Spezifikationen für „C“
- Kompatibilität auf Quellcode-Ebene (kein ABI)
- Funktionalität: POSIX Base Definitions, System Interfaces, and Commands and Utilities (which include POSIX.1, extensions for POSIX.1, Real-time Services, Threads Interface, Real-time Extensions, Security Interface, Network File Access and Network Process-to-Process Communications, User Portability Extensions, Corrections and Extensions, Protection and Control Utilities and Batch System Utilities.

Portables Programmieren mit POSIX (1)

- Headerdateien definieren Funktions-Prototypen, Konstanten und Makros
- Maschinenabhängigkeiten verstecken durch konsequente Verwendung der POSIX-Headerdateien
- Beispiel: Prozess IDs:

älteres UNIX

```
short int pid;

pid = getpid();
```

neueres UNIX

```
long int pid;

pid = getpid();
```

POSIX

```
#include <sys/types.h>
#include <unistd.h>

pid_t pid;
pid = getpid();
```

Portables Programmieren mit POSIX (2)

- Headerdateien definieren Funktions-Prototypen, Konstanten und Makros
- Maschinenabhängigkeiten verstecken durch konsequente Verwendung der POSIX-Headerdateien
- Beispiel: Manipulation der Signalmaske:

UNIX

```
#include <signal.h>

int mask;

mask = 0;
mask |= 1 << (SIGALRM-1)
```

POSIX

```
#include <signal.h>

sigset_t mask;

sigemptyset(&mask);
sigaddset(&mask, SIGALRM);
```

POSIX 1003.1b - Echtzeiterweiterungen

Funktionsgruppen

1. Prioritätengesteuertes Scheduling
2. Echtzeit-Signale
3. Clocks und Timer
4. Semaphore
5. Messages
6. Memory Mapped Files und Shared Memory
7. Asynchrone Ein/Ausgabe
8. Synchrone Ein/Ausgabe
9. Memory Locking

POSIX 1003.1b - Scheduling

Scheduling Parameter

- Scheduling policy
  - `SCHED_FIFO`: Prioritätenbasiert, Preemptiv
  - `SCHED_RR`: wie `SCHED_FIFO`, jedoch mit Quantum
  - `SCHED_OTHER`: nicht näher spezifiziert
- Priorität
  - Prioritäten sind fix
  - Eine Ready-Liste pro Prioritätsstufe
  - Die älteste Task auf der höchsten Prioritätsstufe wird jeweils ausgeführt
  - Zustandsänderung „blockiert“ → „rechenwillig“: Task an das Ende der Ready-Liste (ihrer Prioritätsstufe)
  - Preemption: Task an den Anfang der Ready-Liste
- Funktionen:

`sched_setscheduler()`	Scheduling Policy / Parameter setzen
`sched_getscheduler()`	Scheduling Policy ermitteln
`sched_setparam()`	Scheduling Parameter setzen
`sched_getparam()`	Scheduling Parameter ermitteln
`sched_yield()`	Prozess ans Ende der Ready-Liste
`sched_get_priority_min()`	Minimale Priorität ermitteln
`sched_get_priority_max()`	Maximale Priorität ermitteln

POSIX 1003.1b - Signale

- Signale: vergleichbar mit Interrupts:
  - Es gibt eine begrenzte Anzahl von Signalen
  - Eine Task kann einen Handler für ein Signal installieren
  - Eine Task kann Signale selektiv maskieren
  - Falls kein Handler installiert ist, gibt es einen Default-Handler für jedes Signal
  - Signale werden i. A. nicht gepuffert
- POSIX 1003.1b Erweiterung: Echtzeitsignale
  - Rückwärtskompatibel mit POSIX 1003.1
  - Mindestens 8 neue Signale (`SIGRTMIN` ... `SIGRTMAX`)
  - EZ-Signale können gepuffert werden
- Funktionen:

`sigaction()`	Signal Handler setzen
`sigprocmask()`	Signale maskieren / freigeben
`sig`*xxx*`set()`	Signalmaske manipulieren (*xxx*=`add`, `del`, `fill`)
`sigsuspend()`	Blockieren, bis Signal eintritt
`sigwaitinfo()`	Warten aif Signal (ohne Handleraufruf)
`sigtimedwait()`	Dito, mit Timeout

POSIX 1003.1b - Clocks und Timer

- Clocks
  - Verschiedene „Uhren“ (Taktquellen) möglich
  - Mindestens eine Uhr (`CLOCK_REALTIME`)
  - API erlaubt Auflösung bis zu einer Nanosekunde
- Timer
  - Senden eines Signals an eine Task nach Ablauf eines vorgegebenen Zeitintervalls
  - Zeitquelle wählbar (`CLOCK_REALTIME`, ..)
  - Dynamisch zu erzeugen: Bis zu 32 Timer pro Prozess
  - Absolute und Relative Verzögerung möglich
  - Overrun-Erkennung
- Funktionen:

`clock_settime()`	Uhr setzen
`clock_gettime()`	Uhrzeit ermitteln
`clock_getres()`	Uhrenauuflösung ermitteln
`timer_create()`	Timer erzeugen
`timer_settime()`	Ablaufzeit / -Intervall für Timer setzen
`timer_gettime()`	Zeit bis Timer-Ablauf ermitteln
`timer_getoverrun()`	Anzahl verpasster Timer-Zyklen ermitteln
`timer_delete()`	Timer löschen
`nanosleep()`	Hochauflösendes Blockieren

POSIX 1003.1b - Semaphore

- Zählsemaphore
  - Queueing nach Prozesspriorität
  - Name-based: Identifikation über (Datei-)Namen
  - Memory-based: Identifikation über Speicheradresse
- Funktionen:

`sem_init()`	Memory-based Semaphore erzeugen
`sem_destroy()`	Memory-based Semaphore verwerfen
`sem_open()`	Name-based Semaphore erzeugen
`sem_close()`	Name-based Semaphore verwerfen
`sem_unlink()`	Name-based Semaphore löschen
`sem_wait()`	Semaphore dekrementieren („P-Operation“)
`sem_trywait()`	Nichtblockierendes dekrementieren
`sem_post()`	Semaphore inkrementieren („V-Operation“)
`sem_getvalue()`	Zählerstand ermitteln

POSIX 1003.1b - Message Queues

- Medium für ungerichtete Inter-Prozess-Kommunikation
  - Queueing nach Prozesspriorität
  - Name-based: Identifikation über (Datei-)Namen
- Funktionen:

`mq_open()`	Message Queue erzeugen
`mq_close()`	Message Queue verwerfen
`mq_unlink()`	Message Queue löschen
`mq_send()`	Nachricht senden
`mq_receive()`	Nachricht empfangen
`mq_getattr()`	Message Queue Attribute ermitteln
`mq_setattr()`	(Teilmenge der) Message Queue Attribute setzen
`mq_notify()`	Signal senden, wenn Nachricht eintrifft

POSIX 1003.1b - Mapped Files und Shared Memory

- Memory Mapped Files
  - Abbildung von Dateien in den Adressraum (mit `mmap()`)
- Shared Memory
  - Implementiert als Spezialfall von Memory Mapped Files
  - Identifier für Objekte sind Dateinamen
  - Trotzdem ohne Dateisystem implementierbar
    (→ Restriktionen bei der Namenswahl um Portabilität zu sichern)
  - Die Dateioperationen `ftruncate()` und `close()` sind auch auf Shared Memory Objekte anwendbar
- Funktionen:

`mmap()`	Shared Memory oder Datei in Adressraum abbilden
`munmap()`	Adressraum-Abbildung verwerfen
`shm_open()`	File Descriptor für shared Memory erzeugenn
`shm_close()`	File Descriptor für shared Memory verwerfen
`shm_unlink()`	Shared Memory Segment löschen
`ftruncate()`	Größe eines shared Memory Segemntes festlegen
`mprotect()`	Zugriffsattribute für shared Memory Segment ändern

POSIX 1003.1b - Asynchrone Ein/Ausgabe

- Daten Lesen/Schreiben „im Hintergrund“:
  - Standard (POSIX 1003.1) Funktionen `read()` und `write()` blockieren
  - Es ist nicht sichergestellt, dass der I/O Vorgang nach Rückkehr von `read()`, bzw. `write()` tatsächlich beendet ist
  - `aio_read()` und `aio_write()` blockieren nicht.
  - Jeder `aio_`*xxx*`()`-Auftrag beinhaltet:
    - ★ Datei-Position
    - ★ Signal, das bei Beendigung geschickt wird
    - ★ Priorität (relativ zu anderen `aio_`*xxx*`()`-Anforderungen)
- Funktionen:

`aio_read()`	Asynchron lesen
`aio_write()`	Asynchron schreiben
`aio_listio()`	Mehrere I/O-Anforderungen absetzen
`aio_cancel()`	Asynchronen I/O-Vorgang abbrechen
`aio_suspend()`	Warten auf Beendigung des asynchronen I/O
`aio_return()`	Ergebniswert von asynchronem I/O ermitteln
`aio_error()`	Fehlercode von asynchronem I/O ermitteln

POSIX 1003.1b - Synchrone Ein/Ausgabe

- Sicherstellen, dass Daten und Dateiinhalt übereinstimmen
  - Nach (POSIX 1003.1) stellen `read()` und `write()` nicht sicher, dass die Daten wirklich gelesen / geschrieben wurden (Buffer Cache!)
  - Entweder explizites Synchronisieren über Funktionsaufrufe, oder optionale Flags bei open() angeben:
    - ★ `O_DSYNC` - Nur Daten synchronisieren
    - ★ `O_SYNC` - Daten und Metadaten synchronisieren
    - ★ `O_RSYNC` - Auch Lesezugriffe synchronisieren
- Funktionen:

`fsync()`	Daten und Metadaten synchronisieren
`fdatasync()`	Nur Daten synchronisieren

POSIX 1003.1b - Memory Locking

- Temporäres Auslagern von Speicherseiten („paging“) führt bei Echtzeitprogrammen zu unvorhersagbaren Verzögerungen
  - „Festnageln“ der für die Echtzeitverarbeitung erforderlichen Ressourcen im physikalischen Speicher
- Funktionen:

`mlock()`	Adressbereich „festnageln“
`munlock()`	Adressbereich wieder auslagerbar machen
`mlockall()`	Alle Ressourcen eines Prozesses „festnageln“
`munlockall()`	Ressourcen eines Prozesses wieder auslagerbar machen

POSIX 1003.1c - Threads

Funktionsgruppen

1. Thread Erzeugen/Löschen
2. Thread Attribute
3. Thread Synchronisation
4. Signalbehandlung

POSIX 1003.1c Threads Erzeugen/Löschen (1)

- Thread Erzeugen/Löschen: Beispiel

```c
#include <stdio.h>
#include <pthread.h>

void my_thread(int *param);

main(int argc, char *argv[])
{
        pthread_t thread;
        int arg = atoi(argv[1]);

        pthread_create(&thread,
                       NULL,
                       (void*)my_thread,
                       (void*)&arg);
        .....

        pthread_join(thread, NULL);
        return 0;
}
```

```c
void my_thread(int* pcount)
{
        int i;
        for(i = 0; i < *pcount; i++)
                do_whatever();
}
```

- Funktionen:

pthread_create()	Thread erzeugen
pthread_cancel()	Anderen Thread beenden
pthread_exit()	Eigenen Thread beenden
pthread_join()	Auf Beendigung von Thread warten
pthread_setcancelstate()	Thread cancel verhindern / zulassen
pthread_setcanceltype()	Cancel zu jeder Zeit verhindern / zulassen
pthread_testcancel()	Anstehende Cancels ausliefern
pthread_equal()	Vergleich zweier Thread Handles
pthread_self()	Eigenes Thread Handle ermitteln
pthread_detach()	Thread „abhängen“

POSIX 1003.1c Threads Erzeugen/Löschen (1)

- Thread Erzeugen/Löschen: Beispiel

```c
#include <stdio.h>
#include <pthread.h>

void my_thread(int *param);

main(int argc, char *argv[])
{
        pthread_t thread;
        int arg = atoi(argv[1]);

        pthread_create(&thread,
                       NULL,
                       (void*)my_thread,
                       (void*)&arg);
        .....

        pthread_join(thread, NULL);
        return 0;
}
```

Thread erzeugen und starten

```c
void my_thread(int* pcount)
{
        int i;
        for(i = 0; i < *pcount; i++)
                do_whatever();
}
```

- Funktionen:

pthread_create()	Thread erzeugen
pthread_cancel()	Anderen Thread beenden
pthread_exit()	Eigenen Thread beenden
pthread_join()	Auf Beendigung von Thread warten
pthread_setcancelstate()	Thread cancel verhindern / zulassen
pthread_setcanceltype()	Cancel zu jeder Zeit verhindern / zulassen
pthread_testcancel()	Anstehende Cancels ausliefern
pthread_equal()	Vergleich zweier Thread Handles
pthread_self()	Eigenes Thread Handle ermitteln
pthread_detach()	Thread „abhängen“

POSIX 1003.1c Threads Erzeugen/Löschen (1)

- Thread Erzeugen/Löschen: Beispiel

```c
#include <stdio.h>
#include <pthread.h>

void my_thread(int *param);

main(int argc, char *argv[])
{
        pthread_t thread;
        int arg = atoi(argv[1]);

        pthread_create(&thread,
                       NULL,
                       (void*)my_thread,
                       (void*)&arg);

        .....

        pthread_join(thread, NULL);
        return 0;
}
```

Thread erzeugen und starten

Attribute, default falls NULL

```c
void my_thread(int* pcount)
{
        int i;
        for(i = 0; i < *pcount; i++)
                do_whatever();
}
```

- Funktionen:

pthread_create()	Thread erzeugen
pthread_cancel()	Anderen Thread beenden
pthread_exit()	Eigenen Thread beenden
pthread_join()	Auf Beendigung von Thread warten
pthread_setcancelstate()	Thread cancel verhindern / zulassen
pthread_setcanceltype()	Cancel zu jeder Zeit verhindern / zulassen
pthread_testcancel()	Anstehende Cancels ausliefern
pthread_equal()	Vergleich zweier Thread Handles
pthread_self()	Eigenes Thread Handle ermitteln
pthread_detach()	Thread „abhängen“

POSIX 1003.1c Threads Erzeugen/Löschen (1)

- Thread Erzeugen/Löschen: Beispiel

```
#include <stdio.h>
#include <pthread.h>

void my_thread(int *param);

main(int argc, char *argv[])
{
        pthread_t thread;
        int arg = atoi(argv[1]);

        pthread_create(&thread,
                       NULL,
                       (void*)my_thread,
                       (void*)&arg);

        .....

        pthread_join(thread, NULL);
        return 0;
}
```

```
void my_thread(int* pcount)
{
        int i;
        for(i = 0; i < *pcount; i++)
                do_whatever();
}
```

Thread erzeugen und starten

Attribute, default falls NULL

Zeiger auf Thread-Code

- Funktionen:

pthread_create()	Thread erzeugen
pthread_cancel()	Anderen Thread beenden
pthread_exit()	Eigenen Thread beenden
pthread_join()	Auf Beendigung von Thread warten
pthread_setcancelstate()	Thread cancel verhindern / zulassen
pthread_setcanceltype()	Cancel zu jeder Zeit verhindern / zulassen
pthread_testcancel()	Anstehende Cancels ausliefern
pthread_equal()	Vergleich zweier Thread Handles
pthread_self()	Eigenes Thread Handle ermitteln
pthread_detach()	Thread „abhängen“

POSIX 1003.1c Threads Erzeugen/Löschen (1)

- Thread Erzeugen/Löschen: Beispiel

```
#include <stdio.h>
#include <pthread.h>

void my_thread(int *param);

main(int argc, char *argv[])
{
        pthread_t thread;
        int arg = atoi(argv[1]);

        pthread_create(&thread,
                       NULL,
                       (void*)my_thread,
                       (void*)&arg);

        .....

        pthread_join(thread, NULL);
        return 0;
}
```

```
Zeiger auf
Thread-Code
(int* pcount)

    int i;
    for(i = 0; i < *pcount; i++)
        do_whatever();
}
```

Thread erzeugen und starten

Attribute, default falls NULL

Argument (beliebiger Zeiger)

- Funktionen:

pthread_create()	Thread erzeugen
pthread_cancel()	Anderen Thread beenden
pthread_exit()	Eigenen Thread beenden
pthread_join()	Auf Beendigung von Thread warten
pthread_setcancelstate()	Thread cancel verhindern / zulassen
pthread_setcanceltype()	Cancel zu jeder Zeit verhindern / zulassen
pthread_testcancel()	Anstehende Cancels ausliefern
pthread_equal()	Vergleich zweier Thread Handles
pthread_self()	Eigenes Thread Handle ermitteln
pthread_detach()	Thread „abhängen“

POSIX 1003.1c Threads Erzeugen/Löschen (1)

- Thread Erzeugen/Löschen: Beispiel

```
#include <stdio.h>
#include <pthread.h>

void my_thread(int *param);

main(int argc, char *argv[])
{
        pthread_t thread;
        int arg = atoi(argv[1]);

        pthread_create(&thread,
                       NULL,
                       (void*)my_thread,
                       (void*)&arg);

        .....

        pthread_join(thread, NULL);
        return 0;
}
```

```
Zeiger auf
my_thread(int* pcount)
Thread-Code

int i;
for(i = 0; i < *pcount; i++)
    do_whatever();
}
```

Thread erzeugen und starten

Attribute, default falls NULL

Argument (beliebiger Zeiger)

Warten auf Ende des Thread

- Funktionen:

pthread_create()	Thread erzeugen
pthread_cancel()	Anderen Thread beenden
pthread_exit()	Eigenen Thread beenden
pthread_join()	Auf Beendigung von Thread warten
pthread_setcancelstate()	Thread cancel verhindern / zulassen
pthread_setcanceltype()	Cancel zu jeder Zeit verhindern / zulassen
pthread_testcancel()	Anstehende Cancels ausliefern
pthread_equal()	Vergleich zweier Thread Handles
pthread_self()	Eigenes Thread Handle ermitteln
pthread_detach()	Thread „abhängen“

POSIX 1003.1c Threads Erzeugen/Löschen (1)

- Thread Erzeugen/Löschen: Beispiel

```
#include <stdio.h>
#include <pthread.h>

void my_thread(int *param);

main(int argc, char *argv[])
{
        pthread_t thread;
        int arg = atoi(argv[1]);

        pthread_create(&thread,
                       NULL,
                       (void*)&my_thread,
                       (void*)&arg);

        .....

        pthread_join(thread, NULL);
        return 0;
}
```

```
(int* pcount)

for(i = 0; i < *pcount; i++)
        do_whatever();
```

Thread erzeugen und starten

Attribute, default falls NULL

Zeiger auf Speicher für Returnwert (beliebiger Zeiger)

Warten auf Ende des Thread

(NULL falls nicht erwünscht)

- Funktionen:

pthread_create()	Thread erzeugen
pthread_cancel()	Anderen Thread beenden
pthread_exit()	Eigenen Thread beenden
pthread_join()	Auf Beendigung von Thread warten
pthread_setcancelstate()	Thread cancel verhindern / zulassen
pthread_setcanceltype()	Cancel zu jeder Zeit verhindern / zulassen
pthread_testcancel()	Anstehende Cancels ausliefern
pthread_equal()	Vergleich zweier Thread Handles
pthread_self()	Eigenes Thread Handle ermitteln
pthread_detach()	Thread „abhängen“

POSIX 1003.1c Threads Erzeugen/Löschen (2)

Cleanup Stack

- Liste von Routinen (dynamisch erstellt), die bei Terminierung eines Thread aufgerufen werden
- Funktionen:

`pthread_cleanup_push()`	Neuer Eintrag auf Cleanup Stack
`pthread_cleanup_pop()`	Obersten Eintrag vom Cleanup Stack entfernen

POSIX 1003.1c Thread Attribute (1)

Thread Attribute

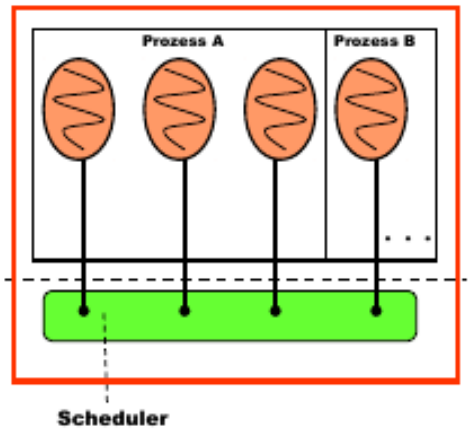
- Das System gibt sinnvolle Default Attribute vor
- Detached/Joinable
  - Detached: Thread läuft eigenständig (weniger Ressourcen)
  - Joinable: Andere Threads können `pthread_join()` aufrufen
- Scheduling Parameter
  - Vgl. POSIX 1003.1b
- Vererbbarkeit von Scheduling-Parametern
- Scheduling scope
- Stackposition und -Größe
  - Vorsicht: Stackmanipulationen sind nicht portabel!

POSIX 1003.1c Thread Attribute (2)

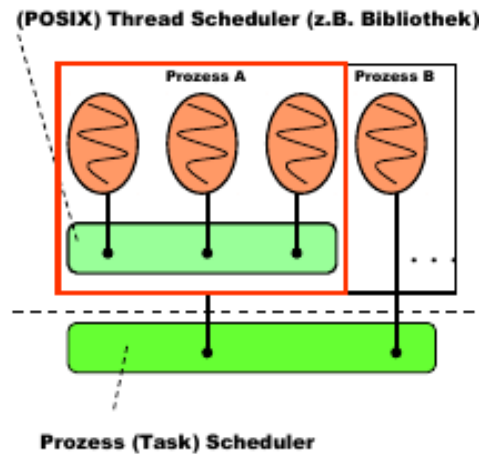
Hochschule RheinMain

- Scheduling Scope
  - *System scope*: Threads konkurrieren systemweit mit anderen Threads/Prozessen
    → „Threads sind Objekte des System-Schedulers“
  - *Process scope*: Threads konkurrieren nur mit anderen Threads desselben Prozesses
    - ★ z.B. Thread-Bibliothek
  - Vielfach nur Untermenge implementiert
    - ★ z.B. Linux: nur System Scope
- Funktionen:

`pthread_attr_init()`	Attribute default-initialisieren
`pthread_attr_destroy()`	Attributstruktur verwerfenen
`pthread_attr_get`*xxx*`()`	Diverse Attribute in Struktur ermitteln
`pthread_attr_set`*xxx*`()`	Diverse Attribute in Struktur setzen
`pthread_getschedparam()`	Scheduling Parameter ermitteln
`pthread_setschedparam()`	Scheduling Parameter setzen



System Scope



Process Scope

POSIX 1003.1c Thread Synchronisation (1)

Mutex (Mutual Exclusion = Wechselseitiger Ausschluss)

- Attribut: Prioritätsprotokoll:
  - `PTHREAD_PRIO_NONE`: keines
  - `PTHREAD_PRIO_PROTECT`: priority ceiling
  - `PTHREAD_PRIO_INHERIT`: priority inheritance
- Zuteilung nach Priorität geordnet
- Funktionen:

`pthread_mutexattr_init()`	Mutex-Attribute default-initialisieren
`pthread_mutexattr_destroy()`	Mutex-Attribute verwerfenen
`pthread_mutexattr_get`*xxx*`()`	Diverse Mutex-Attribute ermitteln
`pthread_mutexattr_set`*xxx*`()`	Diverse Mutex-Attribute setzen
`pthread_mutex_init()`	Mutex initialisieren
`pthread_mutex_destroy()`	Mutex verwerfen
`pthread_mutex_lock()`	Mutex acquirieren
`pthread_mutex_trylock()`	Mutex acquirieren ohne blockieren
`pthread_mutex_unlock()`	Mutex freigeben

`pthread_once()`-Funktion

- Sicherstellen, dass gegebene Funktion genau einmal ausgeführt wird
- z.B. Anlegen einer von mehreren Threads benötigten Ressource

POSIX 1003.1c Thread Synchronisation (2)

Condition Variablen

- ähnlich Zählsemaphore (signal- und wait-Operationen)
- signalisieren/erwarten eines Zustandes
- immer im Zusammenhang mit einem Mutex

Beispiel:

```
void inc_count(void)
{ /*
   * z.B. mehrfach-Aufruf
   * aus verschiedenen Threads
   */
  int i;
  for(i = 0; i < TCOUNT; i++) {
    pthread_mutex_lock(&count_mutex);
    ++count;
    if(count == WATCH_COUNT)
      pthread_cond_signal(&count_cond);
    pthread_mutex_unlock(&count_mutex);
  }
}
```

```
int count = 0;
pthread_mutex_t count_mutex =
            PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t count_cond =
              PTHREAD_COND_INITIALZER;

void watch_count(void)
{
  pthread_mutex_lock(&count_mutex)
  while(count <= WATCH_COUNT) {
    pthread_cond_wait(&count_cond,
      &count_mutex);
  }
  pthread_mutex_unlock(&count_mutex);
  printf("watch_count␣reached\n");
}
```

POSIX 1003.1c Thread Synchronisation (3)

„Process shared“ Attribut:

- Bedeutung: Objekt ist über Prozessgrenzen hinweg nutzbar
- Anwendbar auf Mutexe & Condition Variablen
- Objekt (Mutex bzw. Condition Variable) muss in einem shared Memory Segment liegen
  (ist nicht automatisch gegeben!)
- Nur in Verbindung mit „System“ Scheduling scope
- Alternativ: POSIX 1003.1b Semaphore (`sem_xxx()`, s.o.)

Condition Funktionen

`pthread_condattr_init()`	Condition Attribute default-initialisieren
`pthread_condattr_destroy()`	Condition Attribute verwerfen
`pthread_condattr_getpshared()`	Condition Attribut „process shared“ ermitteln
`pthread_condattr_setpshared()`	Condition Attribut „process shared“ setzen
`pthread_cond_init()`	Condition initialisieren
`pthread_cond_destroy()`	Condition verwerfenen
`pthread_cond_signal()`	Condition signalisieren
`pthread_cond_broadcast()`	Condition an alle signalisieren
`pthread_cond_wait()`	Auf Condition warten
`pthread_cond_timedwait()`	Auf Condition warten mit timeout

POSIX 1003.1c Thread Spezifische Daten (1)

Threadbezogene statische Daten

- z.B. errno

Beispiel:

```
static int Path;

int OpenFile(void)
{
  int x;
  x = open(FILENAME, O_RDONLY);
  if(x >= 0) {
    Path = x;
    return(OK);
  }
  else
    return(ERROR);
}

int ReadFile(int count)
{
  return(read(Path, count));
}
```

```
static pthread_key_t Path;

int OpenFile(void)
{
  int *x = (int*)malloc(sizeof(int));
  pthread_key_create(&Path, NULL);
  *x = open(FILENAME, O_RDONLY);
  if(*x >= 0) {
    pthread_setspecific(Path, (void*)x);
    return(OK);
  }
  else
    return(ERROR);
}

int ReadFile(int count)
{
  int *x = (int*)pthread_getspecific(Path);
  return(read(*x, count));
}
```

Funktionen

pthread_key_create()	Key erzeugen
pthread_key_delete()	Key verwerfen
pthread_getspecific()	Threadspezifisches Datum ermitteln
pthread_setspecific()	Threadspezifisches Datum setzen

POSIX 1003.1c Thread Spezifische Daten (1)

Threadbezogene statische Daten

- z.B. errno

Beispiel

Nicht Thread-safe

```
static int Path;

int OpenFile(void)
{
  int x;
  x = open(FILENAME, O_RDONLY);
  if(x >= 0) {
    Path = x;
    return(OK);
  }
  else
    return(ERROR);
}

int ReadFile(int count)
{
  return(read(Path, count));
}
```

```
static pthread_key_t Path;

int OpenFile(void)
{
  int *x = (int*)malloc(sizeof(int));
  pthread_key_create(&Path, NULL);
  *x = open(FILENAME, O_RDONLY);
  if(*x >= 0) {
    pthread_setspecific(Path, (void*)x);
    return(OK);
  }
  else
    return(ERROR);
}

int ReadFile(int count)
{
  int *x = (int*)pthread_getspecific(Path);
  return(read(*x, count));
}
```

Funktionen

pthread_key_create()	Key erzeugen
pthread_key_delete()	Key verwerfen
pthread_getspecific()	Threadspezifisches Datum ermitteln
pthread_setspecific()	Threadspezifisches Datum setzen

POSIX 1003.1c Thread Spezifische Daten (1)

Threadbezogene statische Daten

- z.B. errno

Beispiel

Nicht Thread-safe

```
static int Path;

int OpenFile(void)
{
  int x;
  x = open(FILENAME, O_RDONLY);
  if(x >= 0) {
    Path = x;
    return(OK);
  }
  else
    return(ERROR);
}

int ReadFile(int count)
{
  return(read(Path, count));
}
```

Thread-safe

```
static pthread_key_t Path;

int OpenFile(void)
{
  int *x = (int*)malloc(sizeof(int));
  pthread_key_create(&Path, NULL);
  *x = open(FILENAME, O_RDONLY);
  if(*x >= 0) {
    pthread_setspecific(Path, (void*)x);
    return(OK);
  }
  else
    return(ERROR);
}

int ReadFile(int count)
{
  int *x = (int*)pthread_getspecific(Path);
  return(read(*x, count));
}
```

Funktionen

pthread_key_create()	Key erzeugen
pthread_key_delete()	Key verwerfen
pthread_getspecific()	Threadspezifisches Datum ermitteln
pthread_setspecific()	Threadspezifisches Datum setzen

POSIX 1003.1c Thread Signalbehandlung (1)

Auswahl des Thread, der ein Signal erhält

Signalart	Ursache	Ziel des Signals	Auswahl
Synchron	Exception (z.B) Division durch Null	Ein bestimmter Thread	Verursacher-Thread
Synchron	Anderer Thread ruft `pthread_kill()`	Ein bestimmter Thread	Ziel-Thread
Asynchron	Externer Prozess ruft `kill()`	Ganzer Prozess	Jeder Thread des Prozesses

POSIX 1003.1c Thread Signalbehandlung (1)

Auswahl des Thread, der ein Signal erhält

Signalart	Ursache	Ziel des Signals	Auswahl
Synchron	Exception (z.B) Division durch Null	Ein bestimmter Thread	Verursacher-Thread
Synchron	Anderer Thread ruft `pthread_kill()`	Ein bestimmter Thread	Ziel-Thread
Asynchron	Externer Prozess ruft `kill()`	Ganzer Prozess	Jeder Thread des Prozesses

Steuerbar durch individuelle (Per-Thread) Signalmaske

POSIX 1003.1c Thread Signalbehandlung (2)

Zustellung asynchroner Signale

- Threads haben individuelle Signalmasken

→ Möglichkeit der Zuordnung bestimmter Signale an bestimmte Threads

- Falls ein Signal bei mehreren Threads nicht maskiert ist: Zustellung an (irgend)einen der Threads (!)

Signalbehandlung

- Signal Handler Tabelle ist gemeinsam für alle Threads
- Nur eine Untermenge der POSIX Funktionsaufrufe sind zulässig (insbesondere keine Pthread Synchronisations- funktionen)
- Aber: POSIX 1003.1b Funktionen (`sem_xx`) sind zulässig