

3. System-Software

3.1 Echtzeitbetriebssysteme (RT-Betriebssysteme)

3.1.1 Anforderungen

Echtzeitfähigkeit:

RT-Betriebssystem muss auf externe und interne *Ereignisse (Events)* in vorbestimmten Zeitgrenzen reagieren (*deterministisches Zeitverhalten*). Zeitverhalten muss exakt vorhersagbar sein (Predictability).

Multi-Tasking/Multi-Threading:

Reaktion auf Ereignisse oder Gruppen von Ereignissen durch *Tasks* (Programme in Ausführung).

Ablauf mehrerer Tasks (quasi-) parallel.

In neueren Systemen auch nebenläufige Ausführungspfade (Threads) pro Task.

Prozessor-Verwaltung:

Dringendste Tasks/Threads müssen zuerst ausgeführt werden (*Scheduling*).

Einhalten von Zeitgrenzen im 'Worst-Case'.

Synchronisation und Kommunikation unter vollständiger Benutzerkontrolle.

Schutzmechanismen ursprünglich weniger bedeutend, aber zunehmend relevant.

Speicherverwaltung:

Virtueller Speicher mit seinem indeterministischen Verhalten nicht geeignet und nicht erforderlich. Dynamische Speicherverwaltung mit Garbage Collection ebenfalls problematisch.

⇒ Meist einfache *statische Allokation* von Speicherbereichen für Tasks (ggf. mit Overlays).

Ein/Ausgabe-Verwaltung:

Spezielle Prozessperipherie wird von Tasks selbst verwaltet.

Standard-E/A-Geräte wie Bildschirm, Tastatur, Plattenspeicher oft gar nicht oder nur rudimentär unterstützt.

Dateisystem:

Wenn vorhanden, ähnlich Standard-Dateisystemen

Entwicklungsumgebung:

Entwicklung auf Host (PC, Workstation),

Test nach Download ins Zielsystem.

Komfortable RT-Betriebssysteme erlauben Tests auf Host und 'abgespeckte Versionen' für Zielsysteme (i. allg. gleicher Prozessor).

Werkzeuge zur Fehlersuche (Debugging) und Leistungsmessung (Monitoring/Profiling)

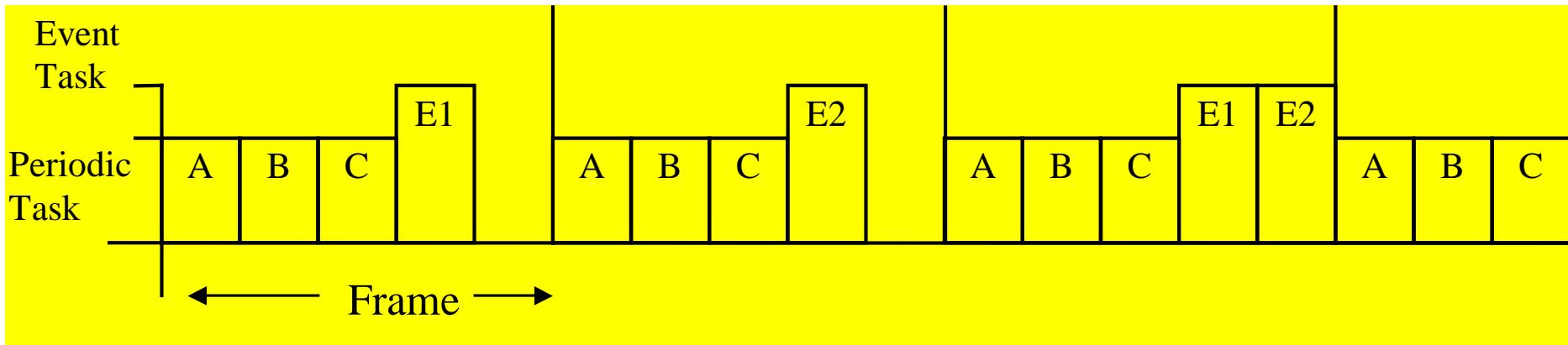
⇒ ***Spezielle Echtzeit-Betriebssysteme oder RT-Varianten universeller Betriebssysteme erforderlich***

3.1.2 Synchrone Task-Ausführung

Prinzip:

Zeitgesteuert: Periodische Ausführung von (Event) Tasks

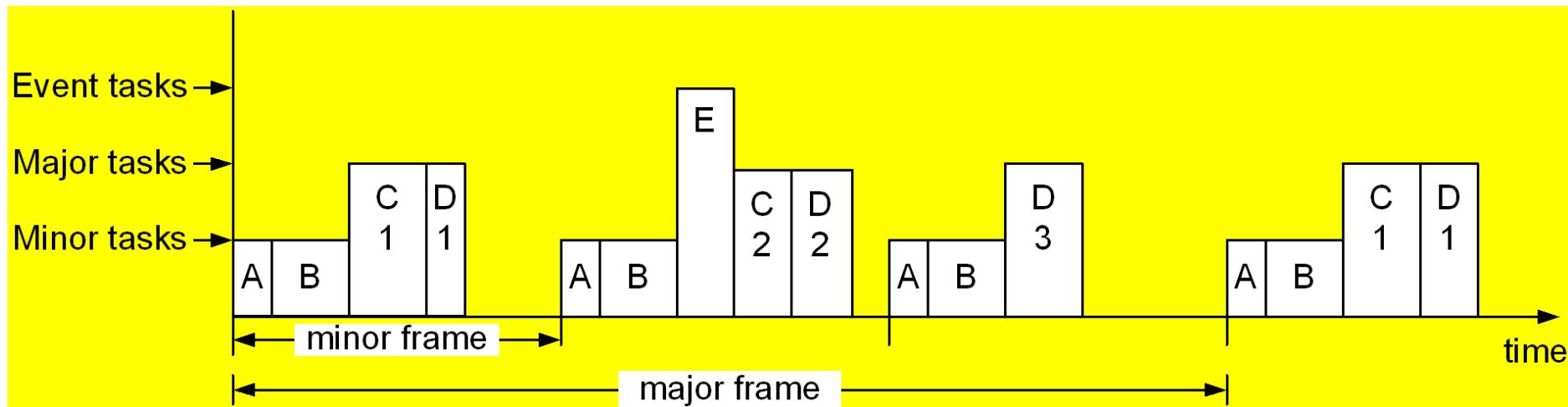
Tasks werden statisch und zyklisch angeordnet (statisches Scheduling).



- *Feste Framelänge*, Ausführung ausgelöst durch Timer Interrupt
 - Tasks laufen pro Frame komplett ab
 - *Feste Reihenfolge* der Tasks, damit einfache Intertaskkommunikation (keine explizite Synchronisation)
 - *Periodische Tasks* (A, B, C) werden in jedem Zyklus ausgeführt
 - *Event Tasks* (E1, E2) nur bei Vorliegen des zugehörigen Events (Interrupt oder Polling durch RT-Kernel)
 - Frame-Länge $\geq \Sigma$ period. Tasks + Σ Event Tasks
- Vgl.: Arbeitsweise von SPS

Realistischere Variante mit Multi-Level Time Frames

Unterstützung verschiedener Task-Ausführungs frequenzen durch *Minor und Major Frames*



Major Frames bestehen aus fester Anzahl von *Minor Frames*.

Tasks mit niedriger Frequenz werden nur einmal pro Major Frame ausgeführt, ggf. verteilt über mehrere Minor Frames (C, D).

Tasks mit höherer Frequenz werden in Minor Frames vollständig ausgeführt (A, B).

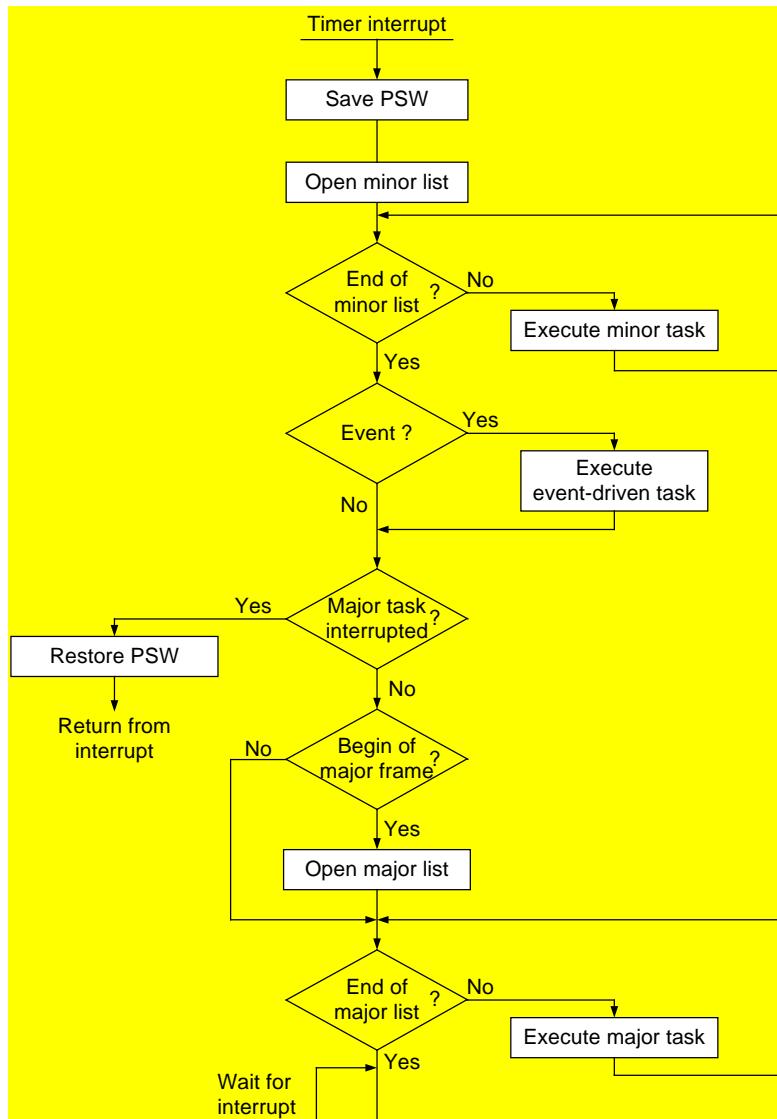
Zusätzlich Ausführung von Event Tasks am Ende der Minor Frames (E).

Zeitbedingungen:

$$\text{Minor Frame} \geq \text{Max}_i (\sum \text{Minor Tasks pro Minor Frame } i + \sum \text{Anteil Major Frame Tasks pro Minor Frame } i + \sum \text{Event Tasks pro Minor Frame } i)$$

$$\text{Major Frame} \geq \sum \text{Minor Tasks} + \sum \text{Major Tasks} + \sum \text{Event Tasks}$$

Synchrone Multi-Level-Taskverwaltung mit automatischer Aufteilung von Major Tasks auf mehrere Minor-Frames



PSW: Program Status Word (Major Task)

minor list: Liste der Minor Tasks für aktuelles Minor Frame

major list: Liste der Major Tasks für aktuelles Major Frame

3.1.3 Asynchrone Task-Ausführung

Nachteile der synchronen Task-Ausführung:

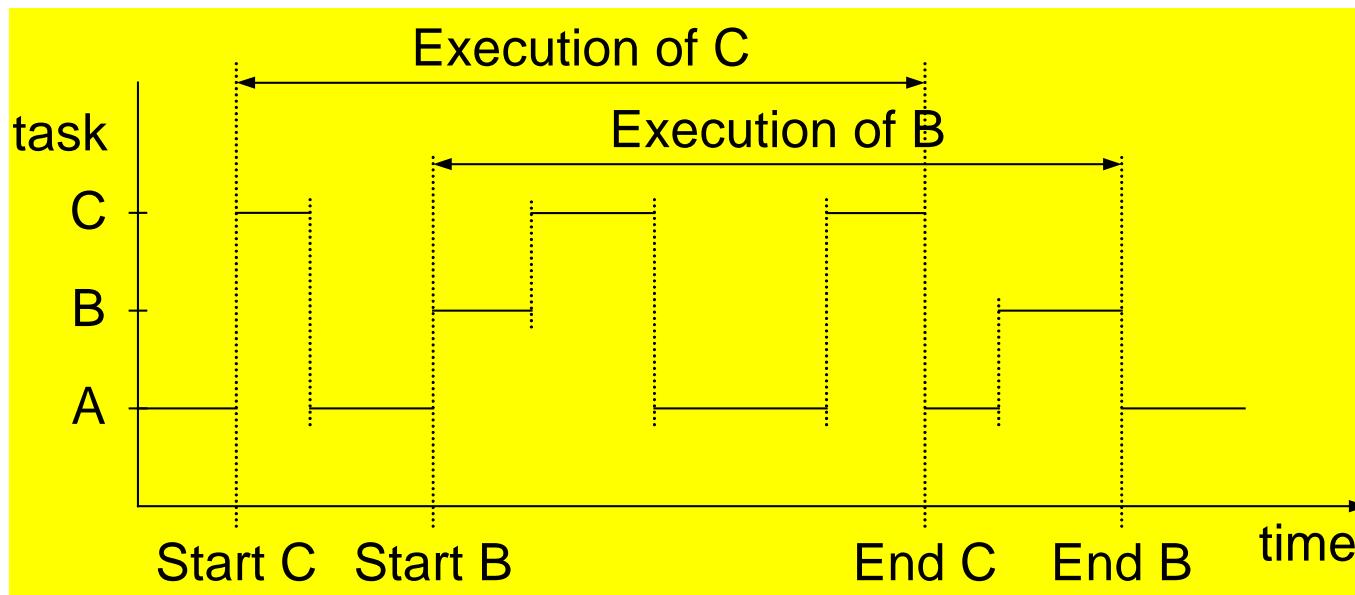
Starres Ausführungsschema

Verschwendung von Prozessorleistung bei sich dynamisch ändernden Umgebungen

Asynchrone Task-Ausführung erlaubt *ereignisgesteuerte* Ausführung von Tasks.

Dynamische Anordnung der Tasks.

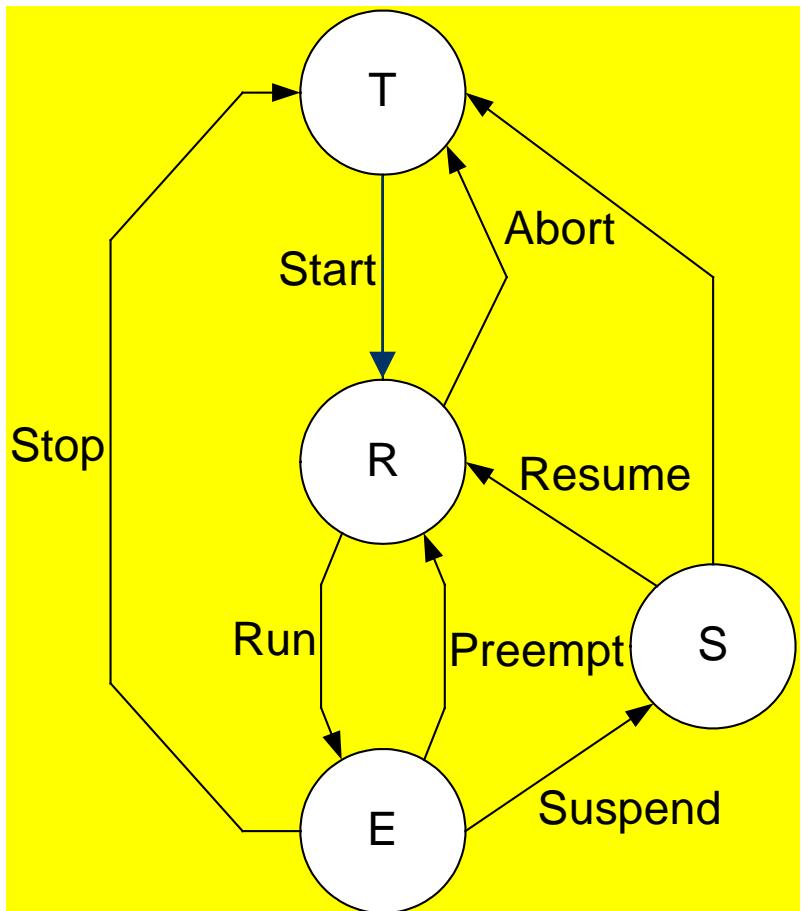
Quasi-parallele Ausführung (Zeitscheiben).



Meist wird auch dynamisches Starten und Terminieren von Tasks unterstützt.

Task-Zustandsübergangsdiagramm

Zustände



T (Terminated):

Task ist in Taskliste registriert, hat aber keine Ressourcen allokiert (Start- und Endzustand)

R (Ready):

Task ist ausführungsbereit, d. h. alle Ressourcen außer Prozessor sind allokiert

E (Executing):

Task wird auf Prozessor ausgeführt

S (Suspended):

Task wartet alternativ auf

- Allokation einer Ressource
- Eintreten eines Ereignisses
- Ablauf eines Zeitintervalls

Zustandsübergänge (Scheduler)

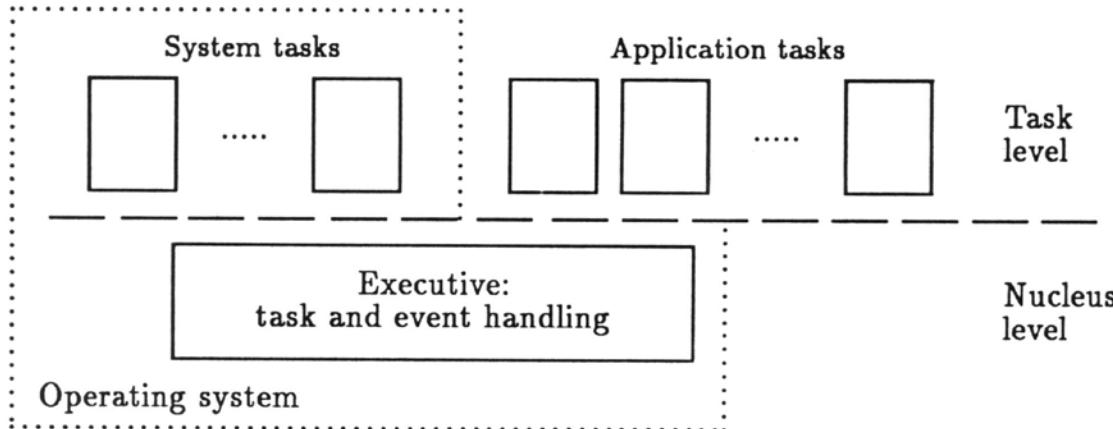
Ausführung von Tasks (Run): Betriebssystem wählt von ausführbereiten Tasks (Zustand **R**) den dringendsten aus, i. Allg. gesteuert über **Task-Prioritäten** und versetzt diesen in den Zustand **E**.

Verdrängung von Tasks (Preempt) aus Zustand **E** in **R** ist Entscheidung des Schedulers, nicht der Task.

Suspendieren einer Task (Suspend) aus Zustand **E** in **S** ist abhängig von Ereignissen (Events):

- | | |
|----------------------|---|
| Externe Ereignisse: | von peripheren Schnittstellen, i. Allg. signalisiert durch Interrupts oder periodisches Polling durch das Betriebssystem können indirekt zur Suspendierung führen |
| Timing-Ereignisse: | Ablauf von Zeitintervallen, signalisiert durch Timer-Interrupts (Hardware-Timer) |
| Interne Ereignisse: | Ausnahme- und Fehlermeldungen, generiert innerhalb des Rechners (Exceptions) |
| Programm-Ereignisse: | explizit ausgelöst vom laufenden Programm (Software-Interrupts, Traps) |

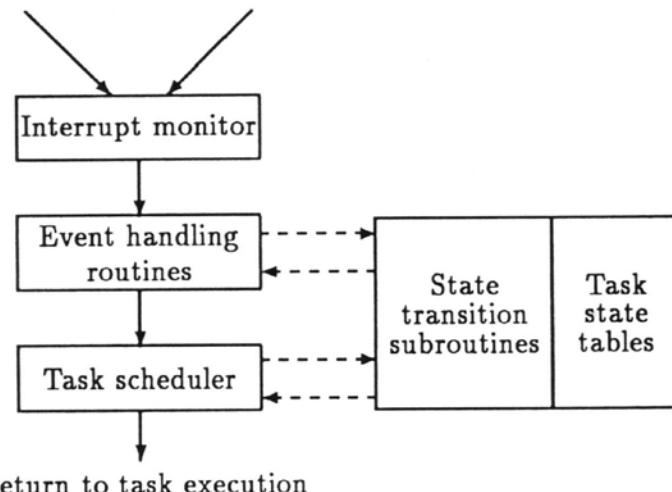
Aufbau eines RT-Multi-Tasking-Betriebssystems



Asynchrone Verwaltung der System- und Anwendungstasks durch Betriebssystemkern (Nucleus, Kernel)

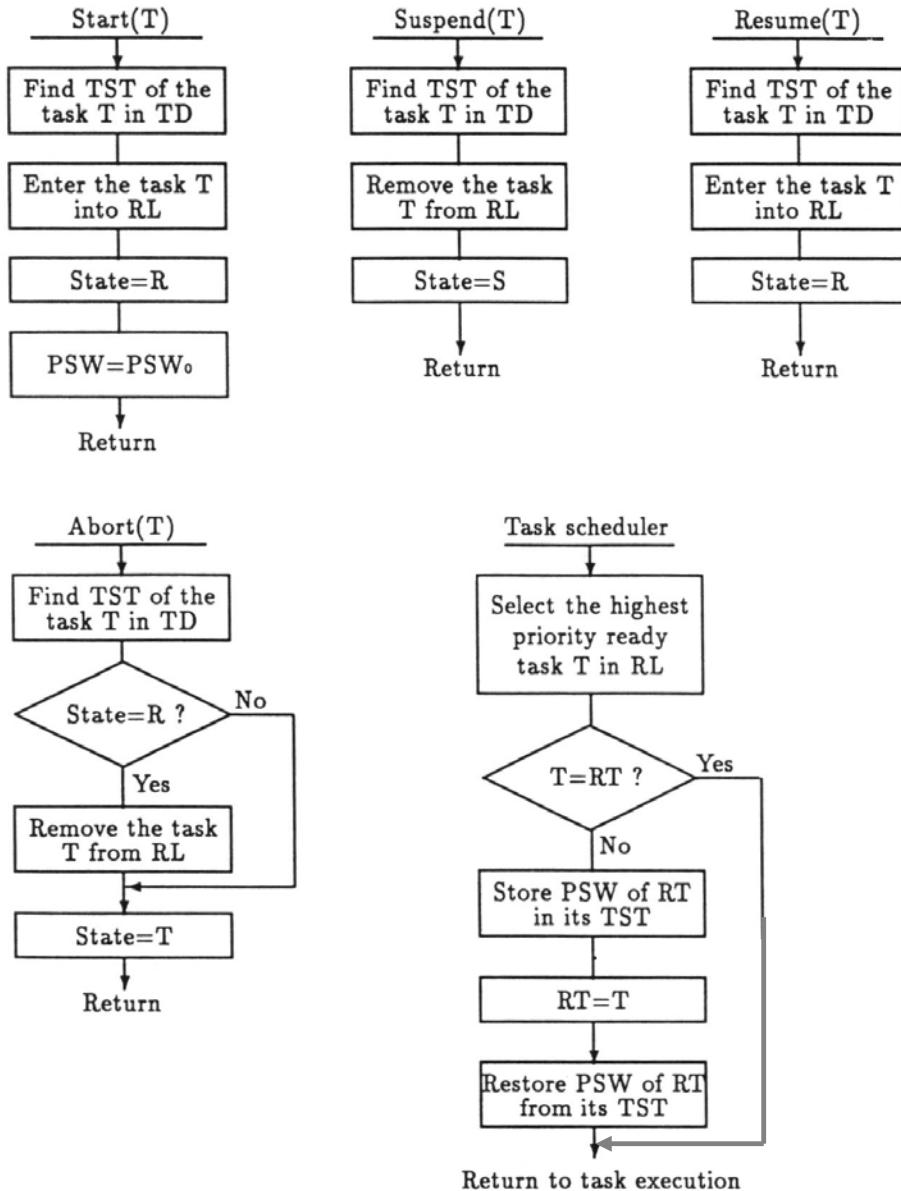
Prinzipielle Struktur des Multi-Tasking-Kernels

Interrupts and program traps



- Ereignisbehandlung (Interrupts)
- Task-Scheduling
- Verwaltung der Taskzustände (Task State Tables) und Übergänge (State Transition Subroutines)

Task-Transition Subroutines



TD:

Task Directory mit registrierten Tasks (Umsetzung Task-ID in Adresse von TST)

TST:

Task State Table enthält pro Task:
State, Priority, PSW, PSW₀

State: Task Zustand (in TST)

Priority: Priorität der Task

PSW: Program Status Word

PSW₀: initiales PSW

RL:

Ready List (ausführbare Tasks)

RT:

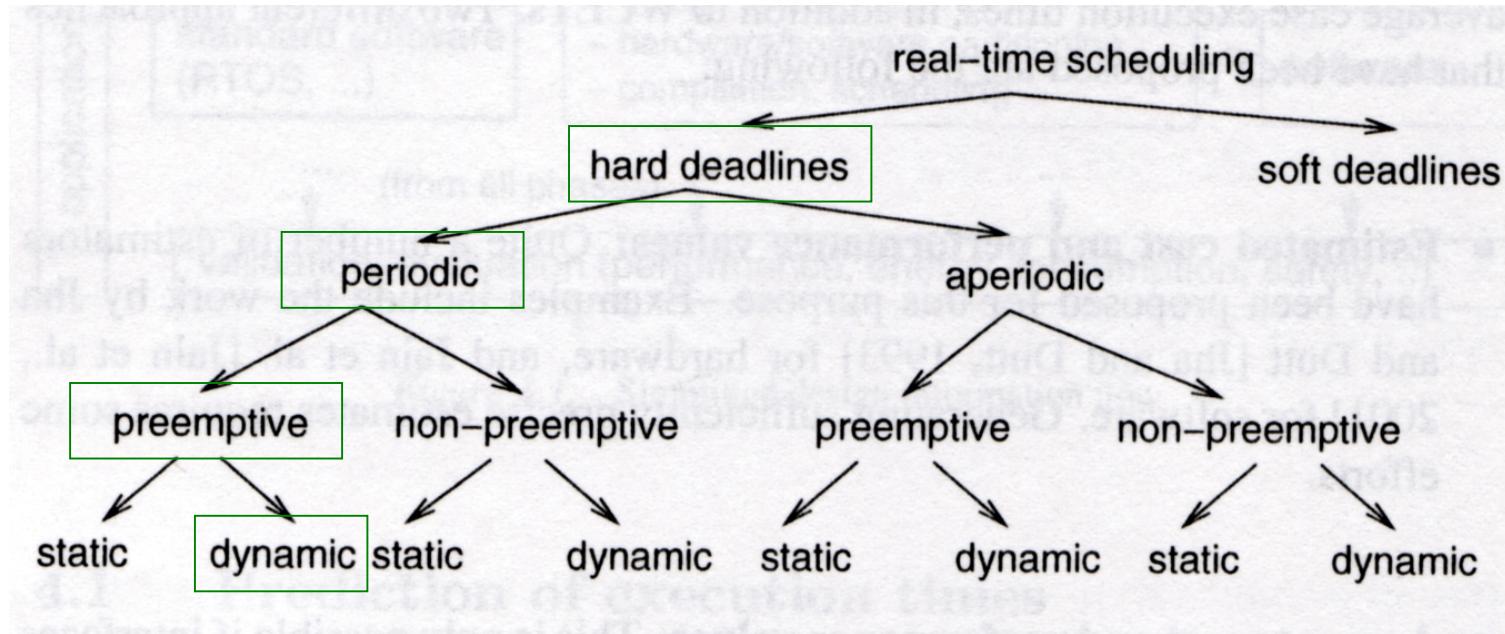
Running Task (laufende Task)

3.1.4 Real-Time Scheduling

Problemstellung

Prozessor soll periodischen bzw. aperiodischen Tasks so zugeteilt werden, dass harte bzw. weiche Zeitgrenzen (Deadlines) eingehalten werden.

Klassifikation



Verdrängend (preemptive): Laufende Task darf unterbrochen werden.

Statisch (static): Schedule wird vor dem Programmstart festgelegt.

Dynamisch (dynamic): Schedule wird erst zur Laufzeit ermittelt.

Modellannahmen

- Alle Tasks laufen *periodisch* auf einem Prozessorkern (Multitasking).
- Alle (harten) Deadlines sind am *Ende* ihrer Perioden.
- Die Ausführungszeiten der Tasks sind *konstant*.
- Zeiten für Kontextwechsel werden ignoriert.
- Alle Tasks sind *unabhängig*, d. h. haben keine Datenabhängigkeiten.
- Der jeweils *höchstpriore* bereite Task (Zustand: READY) wird ausgeführt und *verdrängt* ggf. niederpriore Tasks.
- Keine Interrupts

Notation:

$$\text{Task}_j = (C_j, T_j)$$

$$C_j:$$

$$T_j:$$

$$U_j = C_j/T_j:$$

Ausführungszeit (Worst-Case) von Task j

Periode von Task j

Prozessorauslastung durch Task j

$$U = \sum_{j=1}^n \frac{C_j}{T_j}$$

Gesamtauslastung

Beispiel 1:

$$\text{Task}_1 = (1, 2)$$

$$\text{Task}_2 = (2, 5)$$

$$\text{Task}_1 : C_1 = 1, T_1 = 2 \Rightarrow U_1 = 1/2$$

$$\text{Task}_2 : C_2 = 2, T_2 = 5 \Rightarrow U_2 = 2/5$$

Beispiel 2:

$$\text{Task}_1 = (1, 2)$$

$$\text{Task}_2 = (2, 3)$$

$$\text{Task}_1 : C_1 = 1, T_1 = 2 \Rightarrow U_1 = 1/2$$

$$\text{Task}_2 : C_2 = 2, T_2 = 3 \Rightarrow U_2 = 2/3$$

Rate-Monotonic Scheduling (RMS)

Schedulingstrategie

Prioritäten werden *statisch* (zur Design-Zeit) nach zunehmenden Taskperioden festgelegt, d. h. die Task mit der kürzesten Periode bekommt die höchste Priorität und die mit der längsten Periode die niedrigste Priorität.

Eigenschaften von RMS [Liu/Layland 73]

Optimales Verfahren mit statischen Prioritäten unter den Modellannahmen, d. h. liefert unter allen dynamischen Schedulingverfahren mit statischen Prioritäten eine optimale Prioritätenzuweisung.

Einfach implementierbar mit Komplexität $O(n)$ bei n Tasks (lineare Suche in Taskliste).

In der Praxis sehr häufig eingesetztes RT-Scheduling-Verfahren.

Beispiel 1:

$$\text{Task}_1 = (1, 3)$$

$$\text{Task}_2 = (1, 5)$$

$$\text{Task}_3 = (2, 9)$$

$$\text{Task}_1: \quad C_1 = 1, T_1 = 3 \Rightarrow U_1 = 1/3$$

$$\text{Task}_2: \quad C_2 = 1, T_2 = 5 \Rightarrow U_2 = 1/5$$

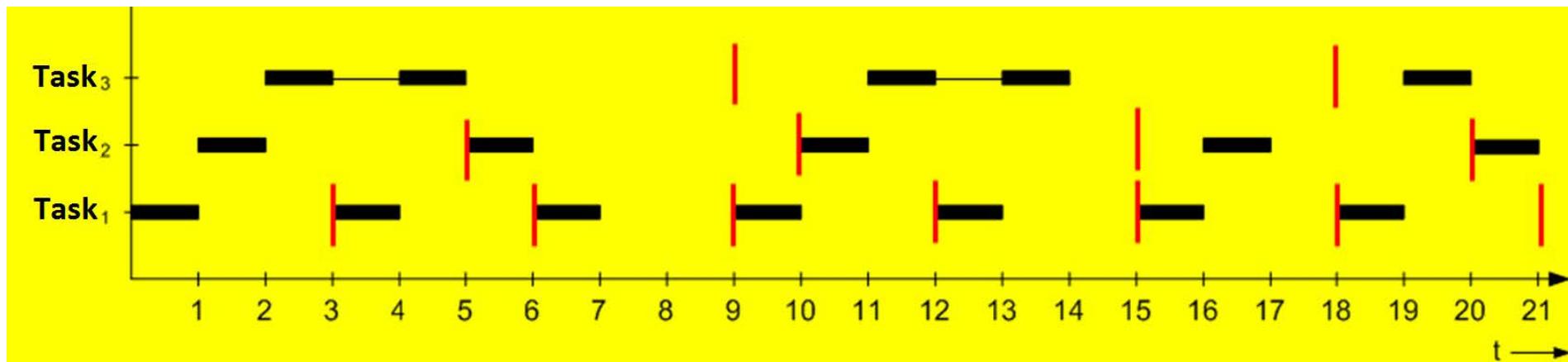
$$\text{Task}_3: \quad C_3 = 2, T_3 = 9 \Rightarrow U_3 = 2/9$$

Gesamtauslastung

$$U = 1/3 + 1/5 + 2/9 = 34/45 \approx 0,755 < 1$$

Prioritäten

$$T_1 < T_2 < T_3 \Rightarrow \text{Priority}(\text{Task}_1) > \text{Priority}(\text{Task}_2) > \text{Priority}(\text{Task}_3)$$



Schedulebarkeitsanalyse

Satz [Liu/Layland 73]:

Für eine Menge von n Tasks existiert immer ein RM-Schedule, wenn gilt:

$$U = \sum_{j=1}^n \frac{C_j}{T_j} \leq n \left(2^{\frac{1}{n}} - 1 \right) = U_g(n)$$

$$U_g(2) \approx 0.828$$

$$U_g(3) \approx 0.780$$

$$U_g(\infty) \approx 0.693$$

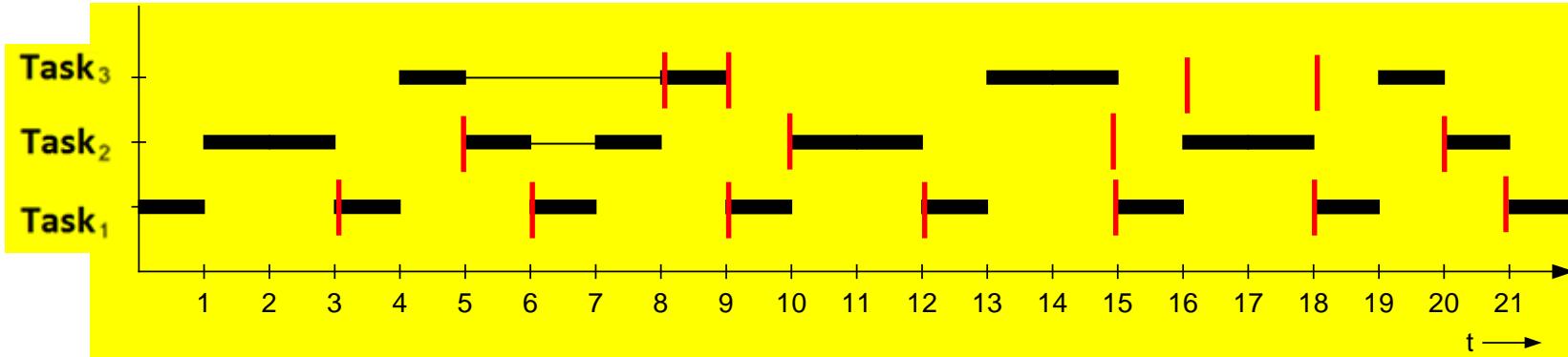
Beispiel 1: $U = 0.75 < U_g(3) = 0.78$
 \Rightarrow RM-Schedule existiert.

Achtung: Die Umkehrung des Satzes gilt nicht, d. h. auch wenn $U_g < U \leq 1$ ist, kann durchaus ein RM-Schedule existieren!

Beispiel 2:

wie Beispiel 1, aber $C_2 = 2$, $T_2 = 5 \Rightarrow U_2 = 2/5$
 $U = 1/3 + 2/5 + 2/9 \approx 0,95 > 0,78 = U_g(3)$

RM-Schedule



Trotz Verletzung der Bedingung aus Satz realisierbar!

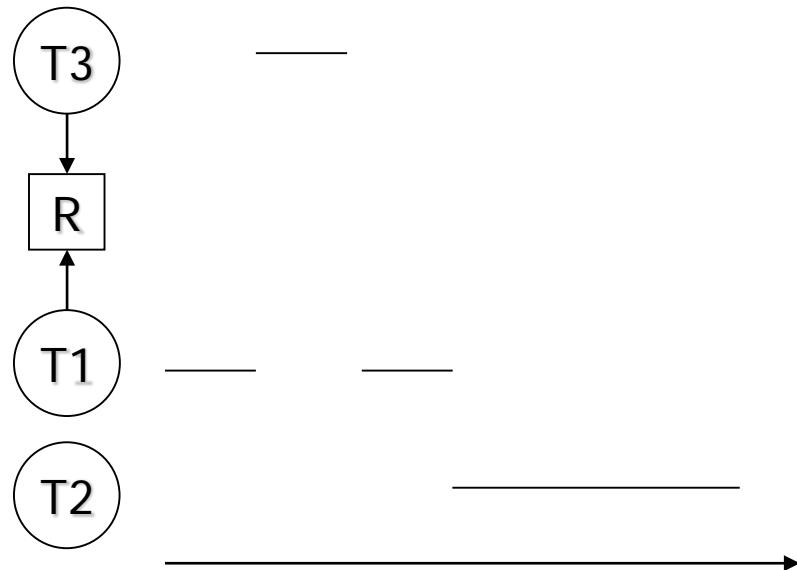
Beispiel 3:

wie Beispiel 2, aber $C_3 = 2$, $T_3 = 8 \Rightarrow U_3 = 2/8$
 $U = 1/3 + 2/5 + 2/8 \approx 0,98 < 1$

RM-Schedule nicht mehr realisierbar, da Prozessor bis $t = 8$ bereits voll mit T_1 und T_2 ausgelastet ist, d. h. T_3 verpasst seine Deadline, obwohl Gesamtauslastung noch unter 100% liegt!

Unbegrenzte Prioritätsinvertierung

Modellannahmen: wie bei RMS, aber Abhängigkeiten der Tasks durch Nutzung gemeinsamer Ressourcen.



Beispiel: RMS-Schedule (T1<T2<T3)

Obwohl sie die höchste Priorität hat, wird T3 durch niedriger priorisierte T2 blockiert, da diese T1 an der Freigabe der Ressource R hindert.

Mögliche Gegenmaßnahme: Ressource bekommt ebenfalls eine Priorität, die höher als die der höchspriorisierten sie nutzenden Task ist. Nutzt eine Task die Ressource, nimmt sie temporär diese höhere Priorität an.

RM-Schedulebarkeitsanalyse mit Blockierung

$$\sum_{j=1}^n \frac{C_j}{T_j} + \max\left(\frac{B_1}{T_1}, \frac{B_2}{T_2}, \dots, \frac{B_{n-1}}{T_{n-1}}\right) \leq n \left(2^{\frac{1}{n}} - 1\right)$$

B_j : Blockierungszeiten von Task j



Mars
Pathfinder
Mission

Was ist
passiert?

Weitere RT-Schedulingverfahren

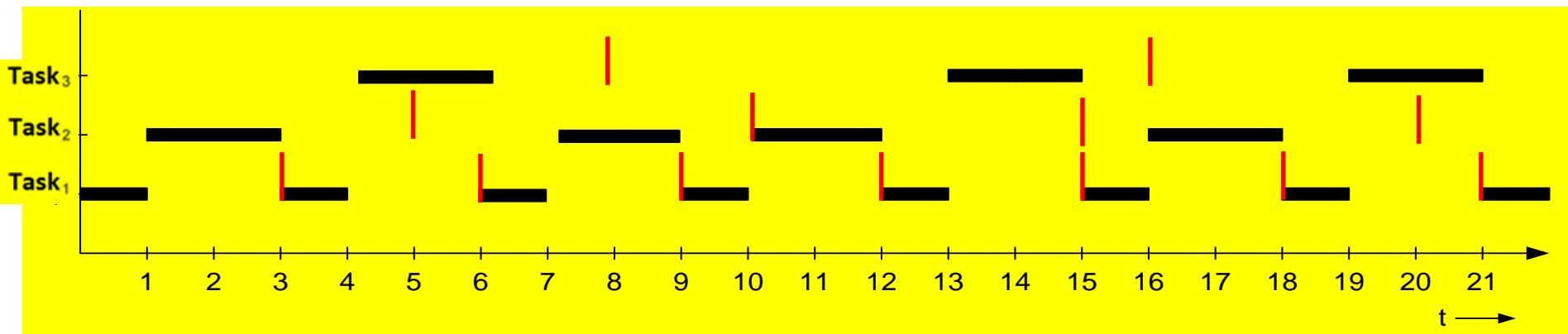
EDF: Earliest Deadline First

Schedulingverfahren mit *dynamischen Prioritäten*, d.h. Prioritäten werden den Tasks erst während der Laufzeit zugeordnet und ändern sich:
Hier: in Reihenfolge der Deadlines mit der nächsten Deadline zuerst.

Garantiert unter den idealisierten Modellannahmen von RMS 100% Prozessorauslastung, aber wesentlich aufwendiger zu implementieren (damit höhere Kontextwechselzeiten).

Beispiel3:

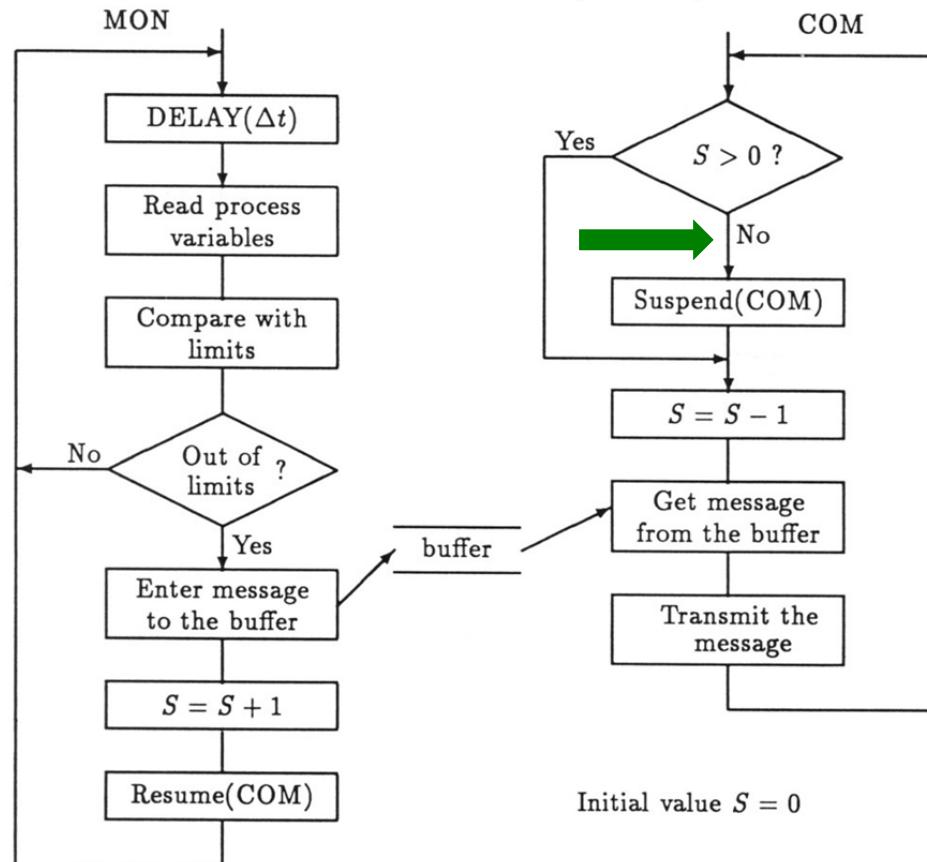
Task1: $C_1=1$, $T_1=3$; Task2: $C_2=2$, $T_2=5$; Task3: $C_3=2$, $T_3=8$; *Jetzt schedulebar!*



Noch eine Reihe von weiteren Schedulingverfahren bekannt und von RT-OS unterstützt, RMS und EDF aber praktisch am bedeutendsten.

3.1.5 Task-Synchronisation und -kommunikation

Beispiel: Inkorrekte Ereignissynchronisation



MON: Monitor-Task, die Daten einliest und bei Überschreiten von Grenzwerten Nachricht in Puffer schreibt

COM: Kommunikations-Task, die gepufferte Nachrichten an Ausgabegerät schickt

Annahmen:

COM wesentlich langsamer als MON

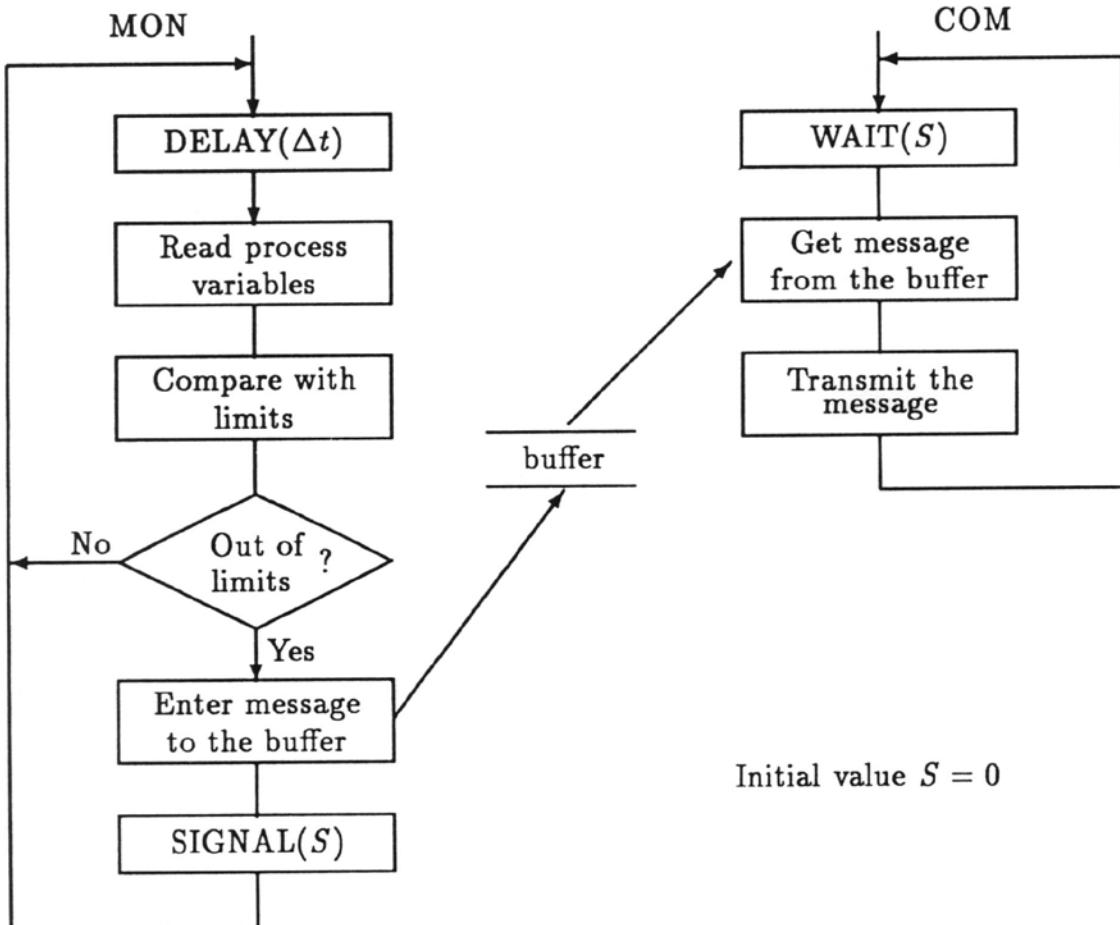
Grenzüberschreitungen relativ selten (kein Pufferüberlauf)

Fehler: gepufferte Nachricht wird evtl. nicht weitergeleitet!

Korrekte Ereignis-Synchronisation mit Semaphoren

Semaphore

Synchronisationsvariable in gemeinsamem Speicher (Mono- oder Multiprozessoren)



Semaphor-Operationen (atomar):

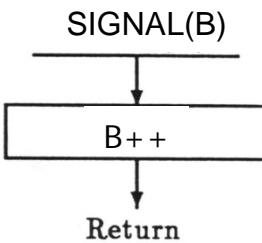
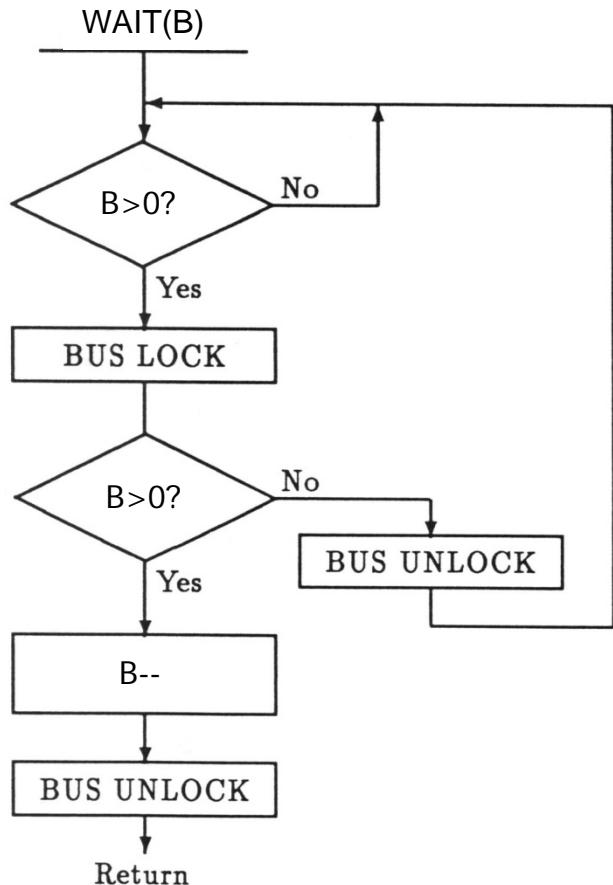
WAIT (S):

if $S > 0$ then $S = S - 1$ else
suspend running task

SIGNAL (S):

if no task is suspended on S then
 $S = S + 1$ else resume one
(highest priority) task

Semaphore mit aktivem Warten (Spinlocks)



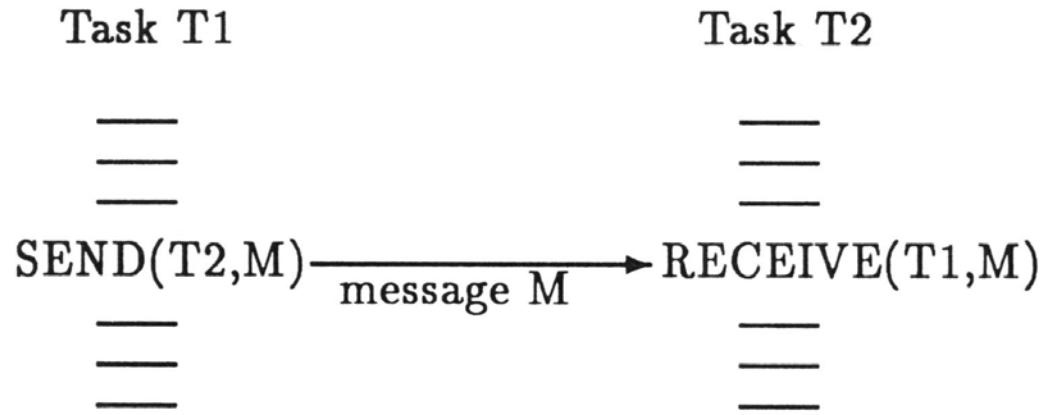
BUS LOCK / UNLOCK stellt Atomizität bei Multiprozessoren sicher. Während der atomaren Phase darf kein Taskwechsel stattfinden (Interruptsperre).

Nachteil: Pollen von B kostet Prozessorzyklen

Vorteil: schnelle Reaktion bei Multiprozessoren

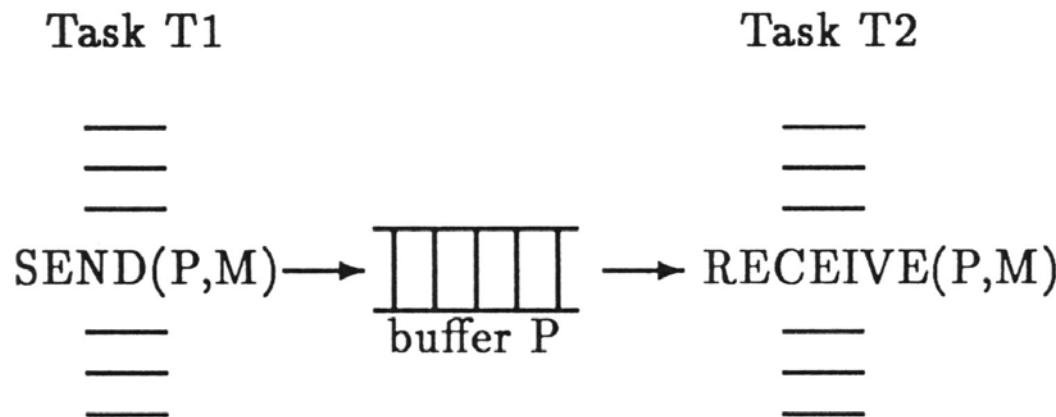
Nachrichtenkommunikation (Message Passing)

Synchrone Nachrichtenkommunikation



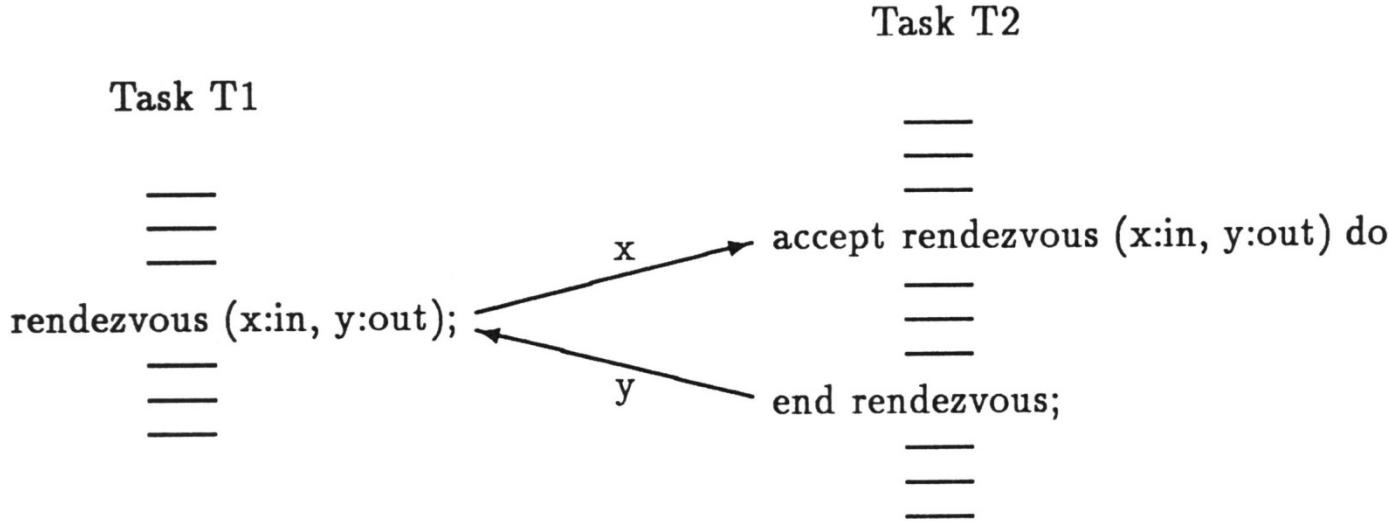
Nachricht wird erst übertragen,
wenn beide Tasks SEND bzw.
RECEIVE aufgerufen haben
(schnellere muss warten).
Kein Puffer erforderlich!

Asynchrone Nachrichtenkommunikation (Nowait Send)



Sender schreibt Nachricht in
Puffer (Mailbox) und läuft sofort
weiter.
Empfänger holt sie später ab
oder wird an Mailbox blockiert,
falls zu früh.

(Erweitertes) Rendezvous



Task T1 spezifiziert Eingabe- und Ausgabeparameter x bzw. y.

Task T2 akzeptiert Rendezvous, berechnet Ausgabedaten und schickt sie an T1 zurück (end rendezvous).

T1 wird bis zum Ende des Rendezvous suspendiert.

Keine Pufferung.

***Alle Synchronisations- und Kommunikationsprimitive logisch äquivalent.
Message-Passing-Primitive auch für verteilte Systeme mit lokalen Speichern geeignet.***

Zeitverwaltung bei asynchroner Task-Ausführung

Betriebssystem unterhält Liste von Tasks, die suspendiert sind und auf das Ende eines Zeitintervalls warten (üblicherweise nach Ablaufzeiten geordnet).

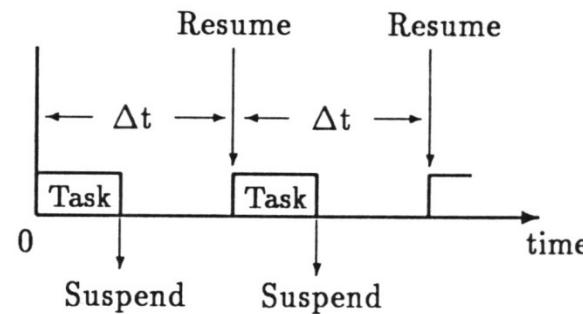
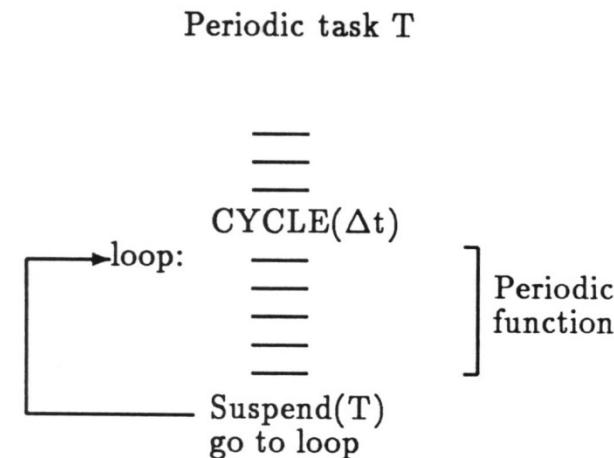
Hardwaretimer liefert Zeitbasis, die regelmäßig mit Ablaufzeit verglichen wird (hardware- oder softwaremäßig implementiert).

Ist die Zeit abgelaufen, wird die suspendierte Task wieder in die Ready-Liste eingetragen (aktiviert).

Typische Timer-Operationen:

DELAY (ΔT): Task wird nach Zeit ΔT wieder aktiviert

CYCLE (ΔT): Task wird zyklisch nach Zeit ΔT aktiviert



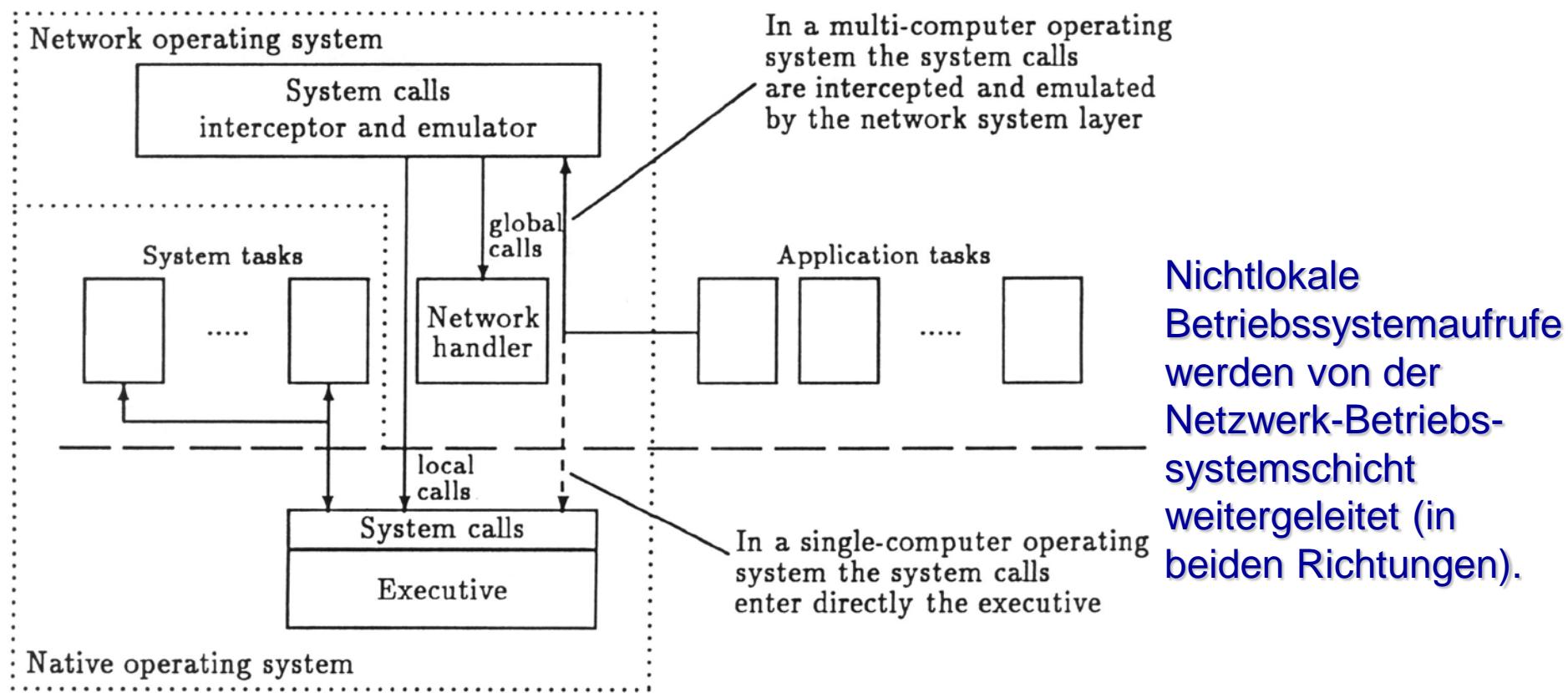
Time out:

Tritt Ereignis nach Ablauf des Time-out-Intervalls nicht ein, wird suspendierter Task reaktiviert oder terminiert (Fehlermeldung!)

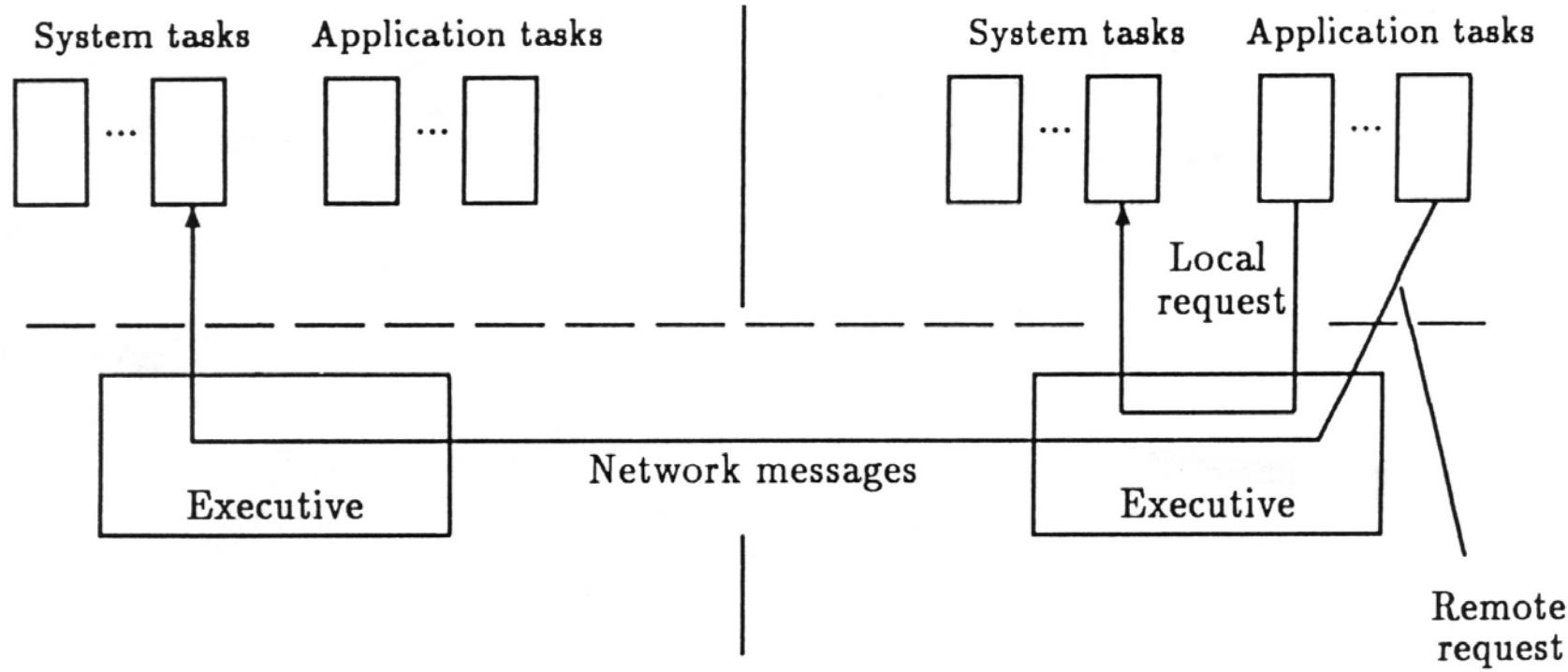
3.1.6 Verteilte Echtzeit-Betriebssysteme

Systeme aus über ein (lokales) Netzwerk verbundenen Knotenrechnern mit jeweils eigenen lokalen Betriebssystemkernen.

Klassische Lösung: Netzwerk-Layer oberhalb des lokalen Betriebssystems



Client-Server-Modell: Applikationstasks (Clients) können auch auf entfernte Systemtasks (Server) zugreifen



Server-Tasks können im ganzen System verteilt sein.

⇒ *Flexible Konfigurierbarkeit und Lastverteilung, Hardware-Architektur für Programmierer nicht sichtbar (Single System Image)*

3.1.7 Beispiel-Echtzeitbetriebssystem: QNX

Zielplattformen

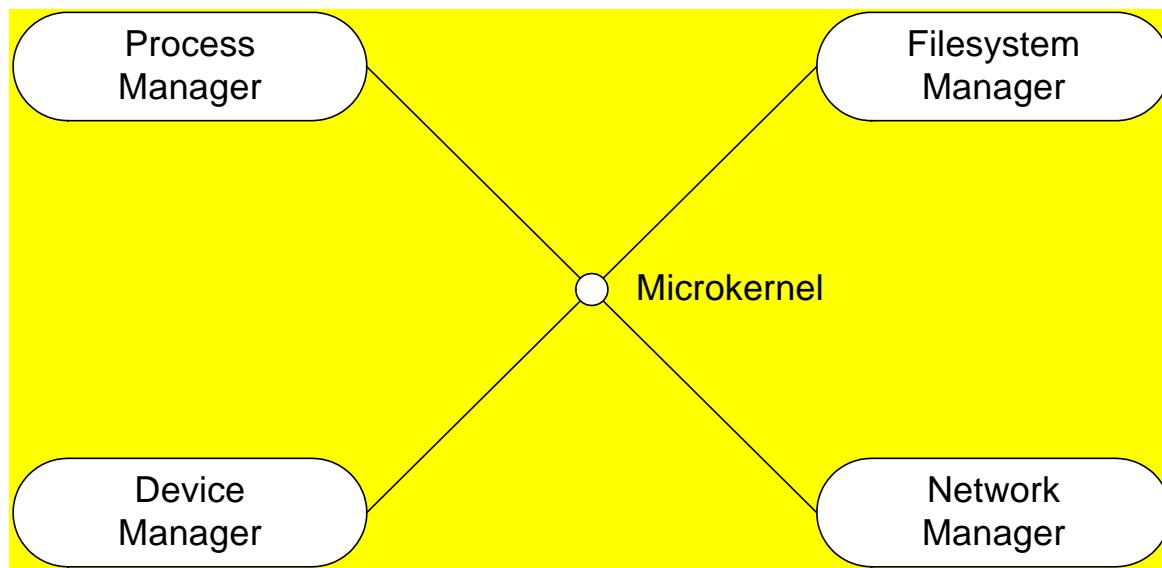
ARM, MIPS, PowerPC, SH-4, Xscale, x86, auch als Multiprozessorsysteme (SMPs) oder Verteilte Systeme

Programmiersprachen

C, C++, Java, Assembler

Architektur

Microkernel-Architektur (QNX Neutrino RTOS)



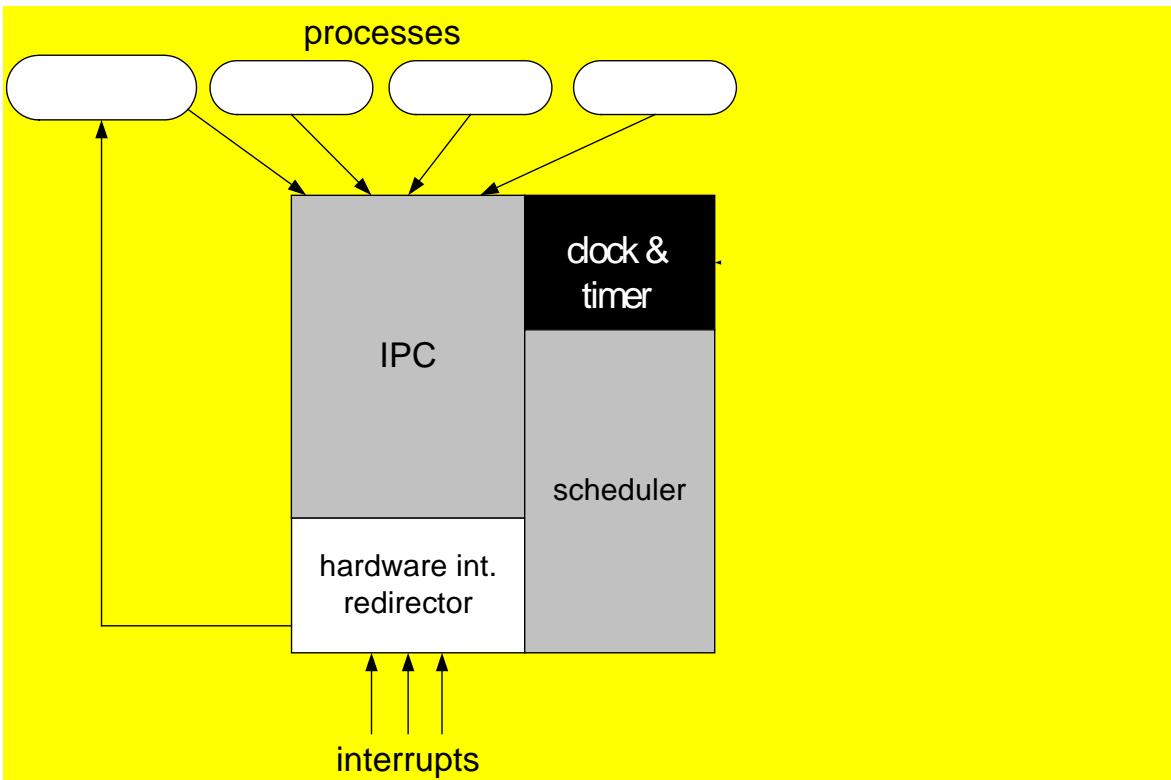
Client-Server-Modell, d. h.
Prozesse (Manager) für
wichtige Systemdienste

Vorteile:

- flexibel erweiterbar
(zusätzliche Manager)
- leicht verteilbar
(mehrere Knotenrechner)
- skalierbar

Microkernel

- Inter-Prozess-Kommunikation (IPC) über Nachrichten (Message Passing)
- Zeitverwaltung (Timer, Realtime-Clock)
- Prozessorzuteilung (Scheduling)
- hardwarenahe Unterbrechungsverarbeitung (Interrupts)



Sehr klein, nur wichtigste Grundfunktionen. Übrige Teile des Betriebssystems als Server-Prozesse, die wie Applikations-Prozesse vom Kernel verwaltet werden und über Nachrichten kommunizieren.

⇒ *skalierbar von einzelnen Mikrocontrollern bis zu großen verteilten Systemen*

System-Prozesse (Auswahl):

- *Process Manager*
verwaltet System- und Applikations-Tasks (Kreierung, Terminierung, Prioritäten etc.). Mehrere Threads pro Prozess möglich (vgl. LINUX).
- *Filesystem Manager*
verwaltet das Dateisystem (Anlegen, Öffnen, Schließen von Dateien, Schreiben/Lesen etc.) basierend auf Standard-Dateisystemen (z. B. DOS, Linux, NFS).
- *Device Manager*
enthält die benötigten Gerätetreiber (z. B. Netzwerktreiber, Plattentreiber, Bildschirmtreiber), die dadurch vom geräteunabhängigen Rest des Systems separiert werden.
- *Netzwerk Manager*
unterstützt die netzwerkweite Kommunikation zwischen mehreren Knotenrechnern und zu externen Netzen (z.B. TCP/IP).

Applikations-Prozesse laufen auf gleicher Ebene wie System-Prozesse!
Alle Prozesse besitzen geschützten Adressraum (Memory Protection).

Anwender-Programmierschnittstelle (Application Programming Interface – API)

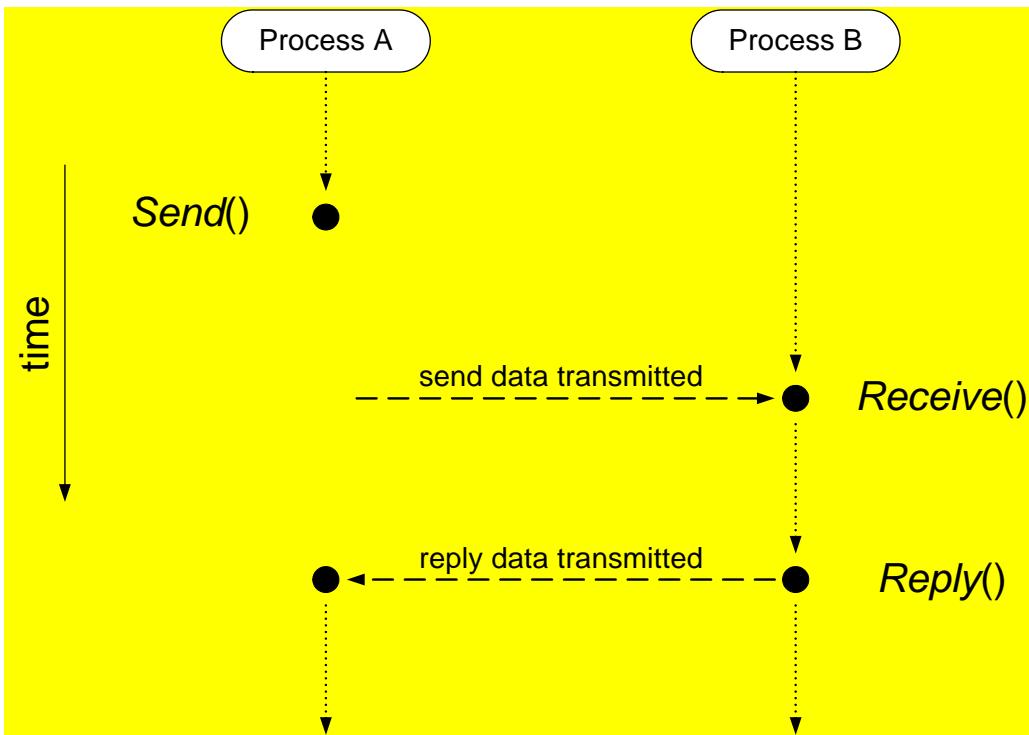
POSIX-kompatibel incl. POSIX Realtime-Erweiterungen

⇒ Programmierung aus Sicht des Anwenders ähnlich wie (Embedded) LINUX,
aber andere Implementierung als LINUX-Kernel, damit *(hart) echtzeitfähig*.

Systemdienste des Microkernels (POSIX-Schnittstelle)

- Threads
- Message Passing
- Signals
- Clocks
- Timers
- Interrupt Handlers
- Semaphores
- Mutual Exclusion Locks (Mutexes)
- Conditional Variables (Condvars)
- Barriers

Asynchrone Nachrichtenkommunikation (Rendezvous-Konzept) zwischen Prozessen

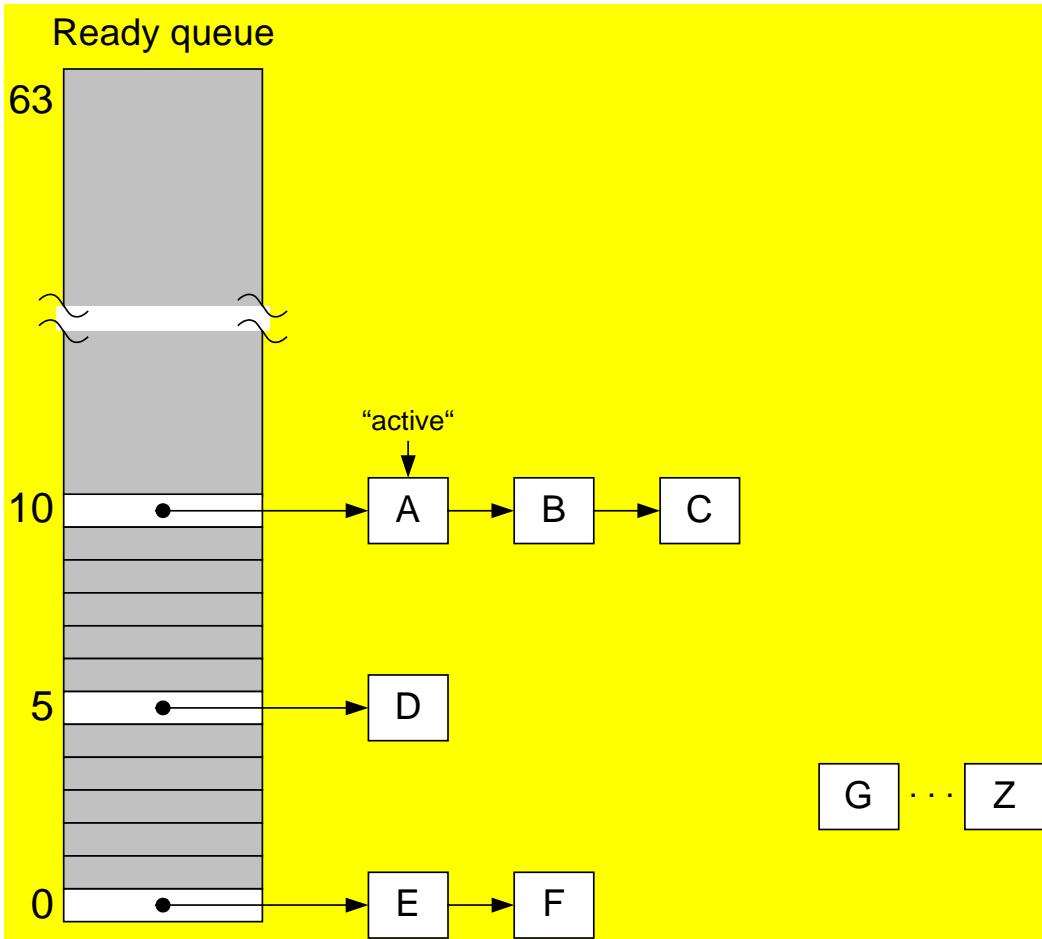


- Send ():** Schickt Message an kooperierenden Prozess und suspendiert rufenden Prozess bis zur Antwort (blockierendes Send)
- Receive ():** Empfängt Message und bereitet Antwort vor (blockierend bei Aufruf vor korrespondierendem Send)
- Reply ():** Schickt Ergebnis an sendenden Prozess zurück und deblockiert ihn.

Rendezvous-Konzept besonders geeignet für Client-Server-Kommunikation, insbesondere zwischen Anwender- und System-Prozessen.

Thread-Scheduling

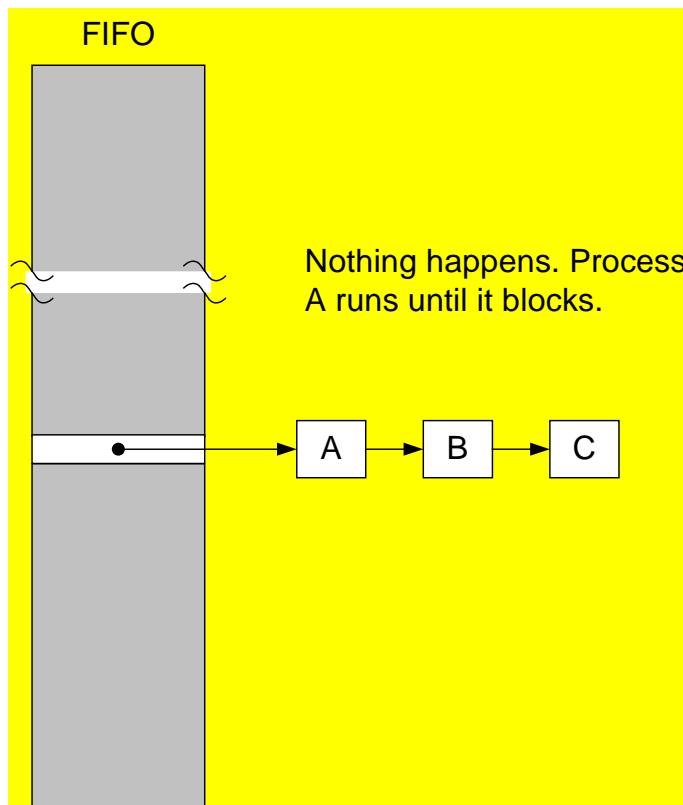
Ausführbereiter Thread wird nach *Priorität* (64 Prioritätsklassen) dem Prozessor zugeteilt.



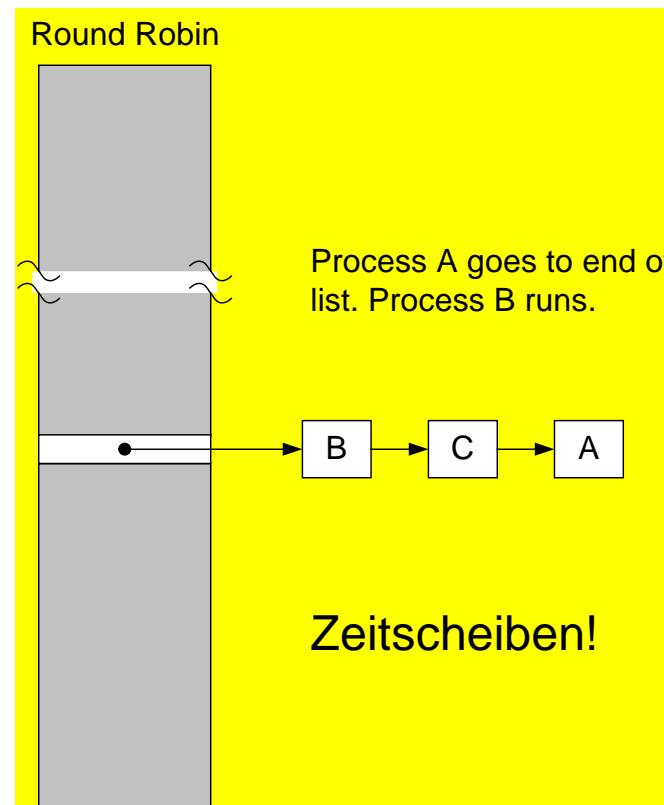
Höherpriore Threads verdrängen niederpriore (Preemption).

Innerhalb einer Prioritätsklasse verschiedene Strategien: FIFO, Round Robin oder Sporadic (Adaptive).

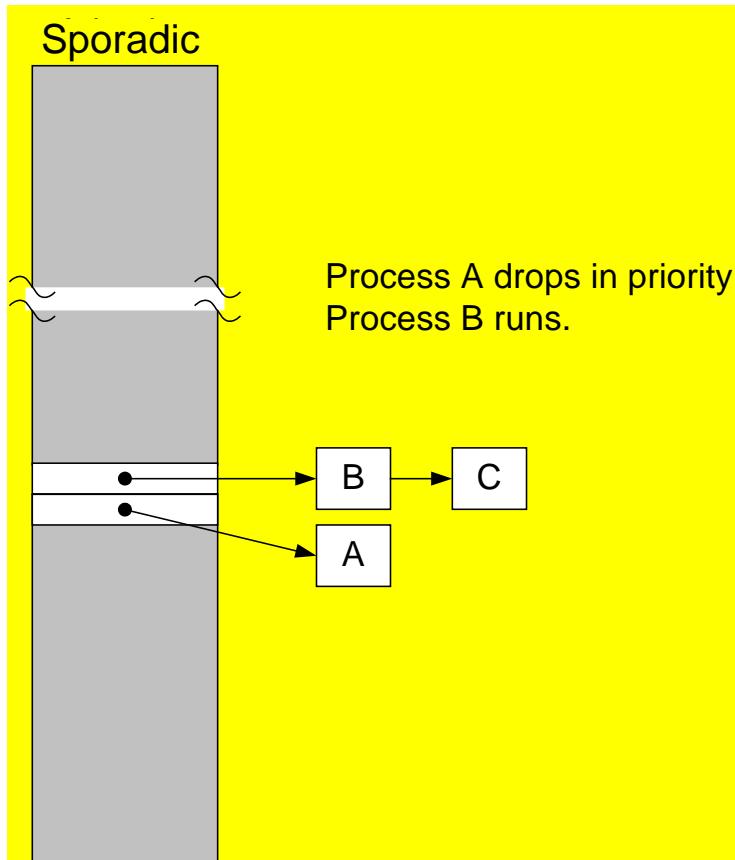
Scheduling-Strategien



Threads laufen in
Reihenfolge der Liste bis
sie jeweils den Prozessor
selbst abgeben.



Threads werden reihum in
Zeitscheiben ausgeführt.



Threads bekommen ein Zeit-Budget pro Periode (z.B. 10 ms je 40 ms).

Sobald ihr Zeitbudget abgelaufen ist, sinkt ihre Priorität (siehe Thread A).

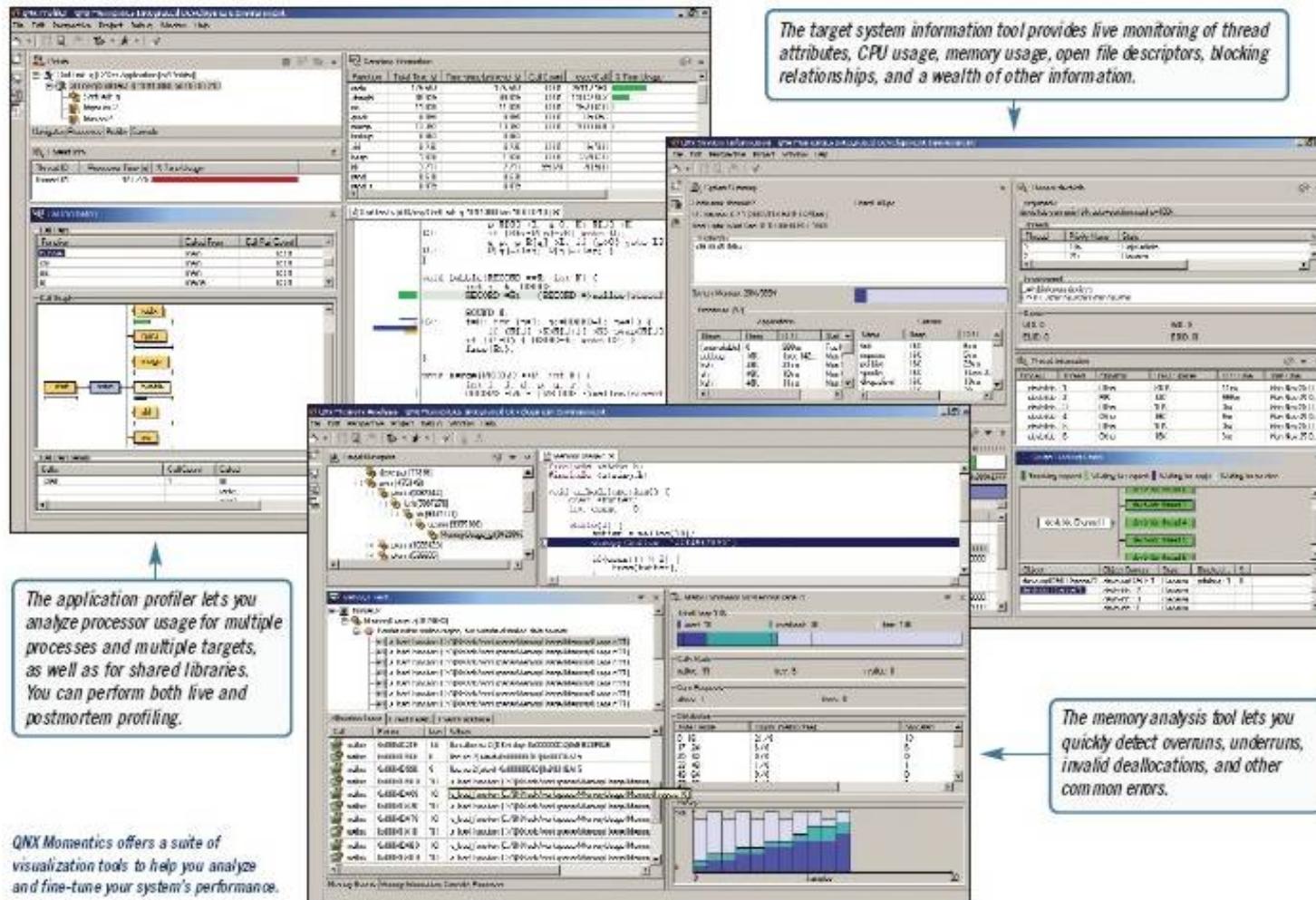
Zu Beginn der nächsten Periode erhalten sie wieder ein neues Zeitbudget mit der ursprünglichen höheren Priorität.

=> Adaptives Verfahren, das verhindert, dass ein Thread eine zu hohe Prozessorlast generiert.

Weitere Taskverwaltung (fork, exec etc.) durch Process Manager.

QNX Momentics Integrated Development Suite

Editoren, Debugger, Profiler, Bibliotheken etc. zur Programmentwicklung auf Host-PC (Windows, Linux) oder Solaris Workstations.



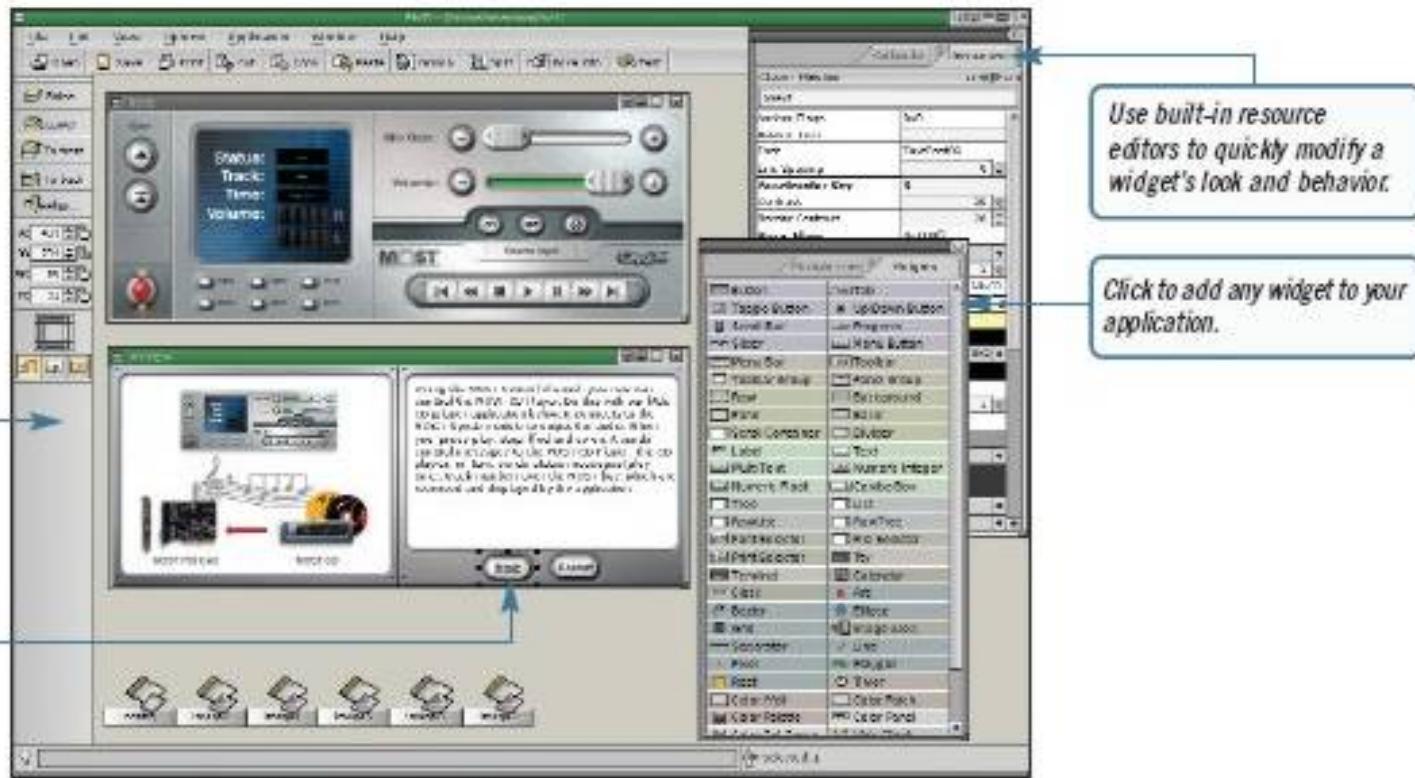
Basiert auf
Eclipse

QNX PHOTON Application Builder mit MicroGUI

Creating applications for the Photon microGUI is a snap when you use PhAB, the Photon Application Builder. Just point and click to add or customize buttons, windows, menus, gauges, and other interface components.

Translate your UI into multiple languages, with full Unicode support.

Attach callbacks that will automatically open windows, dialogs, and menus – no need to write the “glue” that ties your interface together.



Applikationsspezifische graphische Benutzeroberflchen

Weitere Infos und Downloads: www.qnx.com

3.1.8 Realtime Linux und Embedded Linux

Viele verschiedene Linux-Varianten, die ständig weiterentwickelt werden.

Linux zunächst nicht für Echtzeit-Anwendungen entwickelt, aber unter der Bezeichnung *Realtime-Linux* (hart) echtzeitfähige Varianten verfügbar.

Standard-Linux ab *Kernel 2.6* auch für (weiche) Echtzeit geeignet.

Spezielle Variante *uClinux*, die für Mikrocontroller ohne MMU entwickelt wurde. Allerdings eingeschränkte Funktionalität und damit Kompatibilitätsprobleme mit Standard-Linux.

uClinux heute weniger bedeutend, da selbst kleine Systeme mit MMU verfügbar sind und ab *Kernel 2.6* Standard-Linux auch ohne MMU konfigurierbar ist.

Teile von *uClinux* wie die *uClibc* und die *Busybox*, die besonders platzsparend sind, wurden mittlerweile in Standard-Linux übernommen.

Zielplattformen

32-Bit-Mikroprozessoren (z. B. ARM, x86, Power PC, MIPS, M68000)
mit mindestens 2 MB (Flash)ROM und 4 MB RAM

Beispiel: Axis ETRAX 100LX Controller



Anwendungsbeispiele:

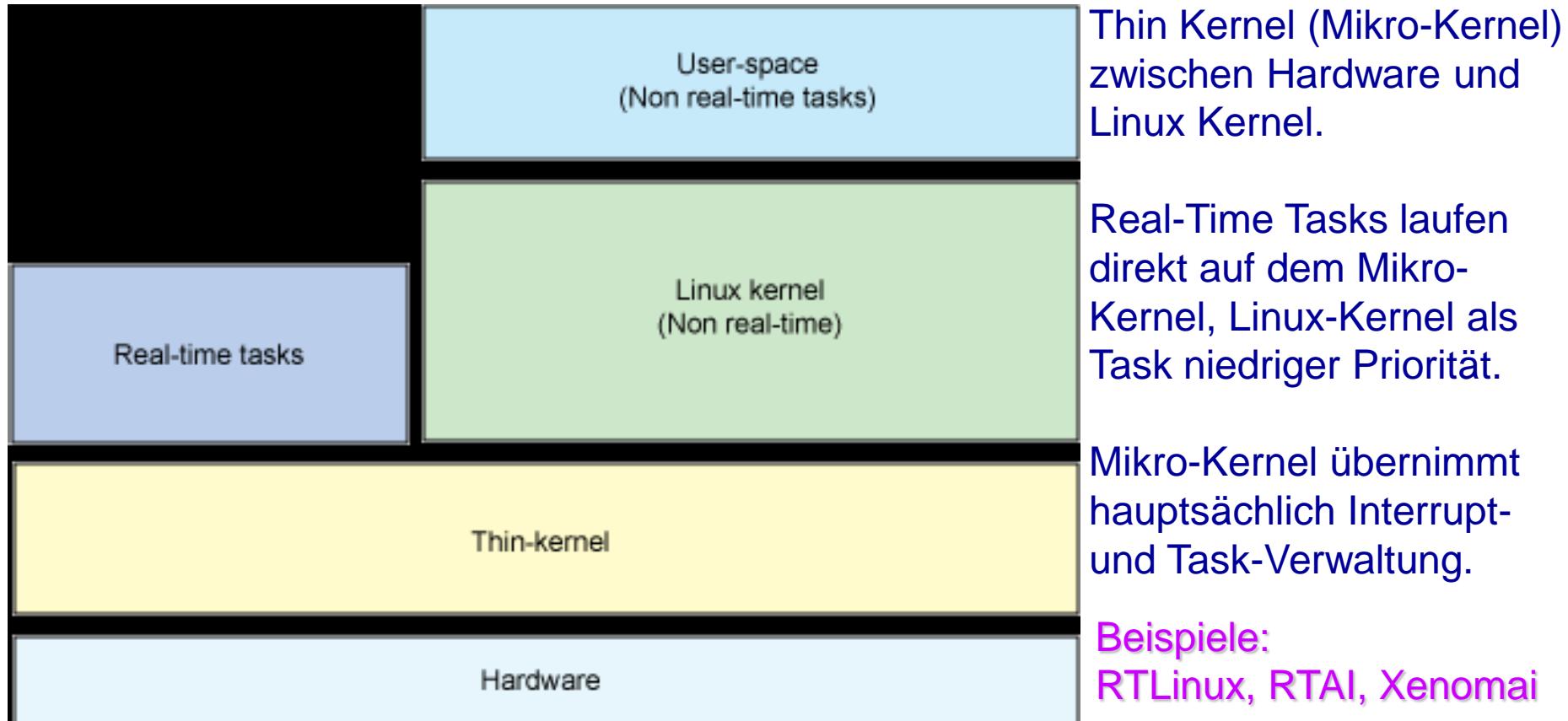
Mini-Webserver (z. B. Webcams)
Network Devices (z. B. Router)
Ersatz für einfache Mikrocontroller

32-Bit-RISC-CPU ETRAX 100LX, 100MHz
Takt, MMU, 8KB Cache
8 MB Flash ROM, 16/32 MB SDRAM
10/100 Mbps Ethernet Port (RJ45-Buchse)
2x USB 1.1 (Full-Speed Host)
RS232-Port (TTL-Pegel) und I2C
IDE, SCSI oder Wide-SCSI
zwei Stiftleisten mit je 2x20 Pins, binäre
Ports
3,3V-IOs, 5V-tolerant
Versorgung: 5V=, ca. 280mA
Abmessungen: 66mm x 72mm
Ready-to-run Embedded Linux System
(Linux Kernel 2.6)
Standardanwendungen: HTTP (Web-
Server), FTP, Telnet, DHCP, SSH, PPP...

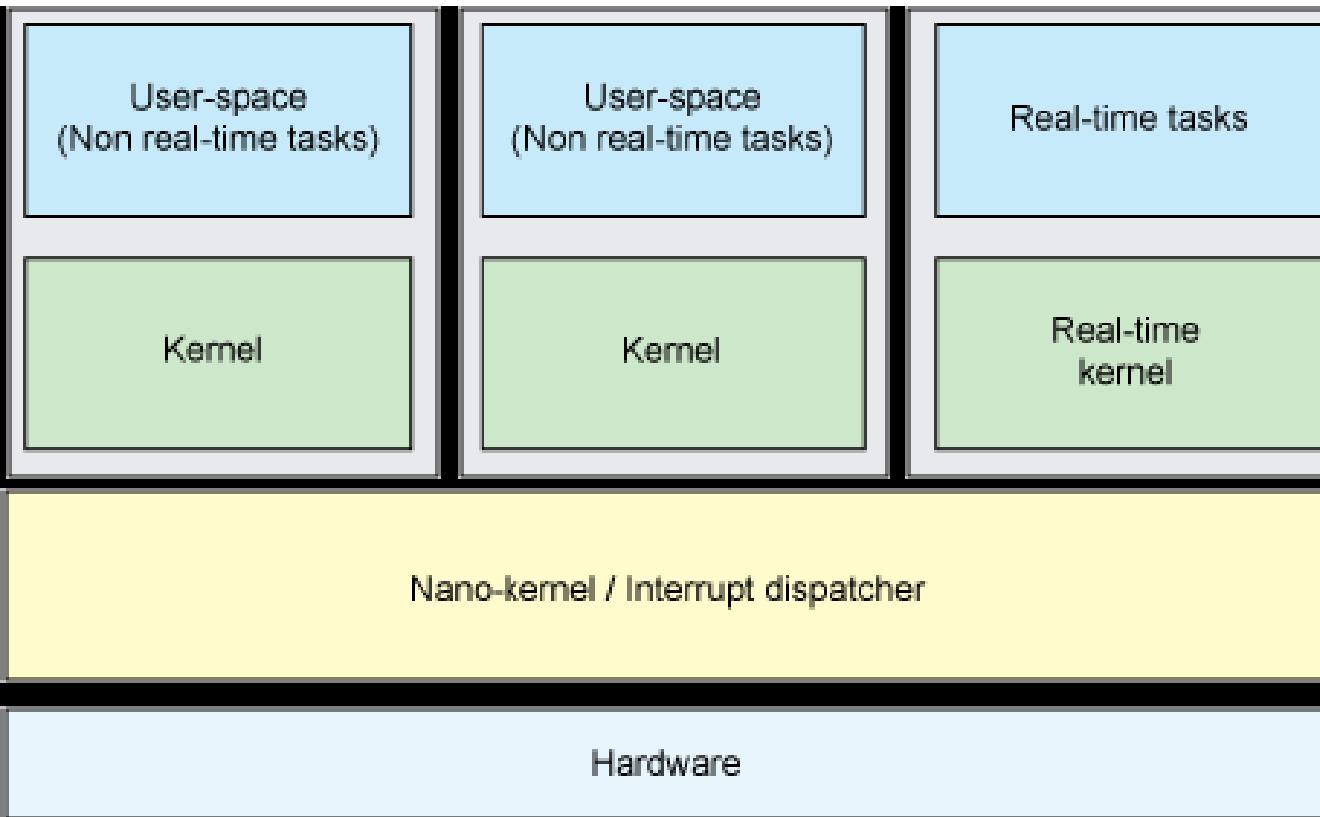
Realtime Linux (RT-Linux)

Da der Standard Linux-Kernel nicht hart echtzeitfähig ist, sind spezielle Kernel-Implementierungen erforderlich [Jones 2008]

Thin-Kernel Ansatz



Nano-Kernel Ansatz

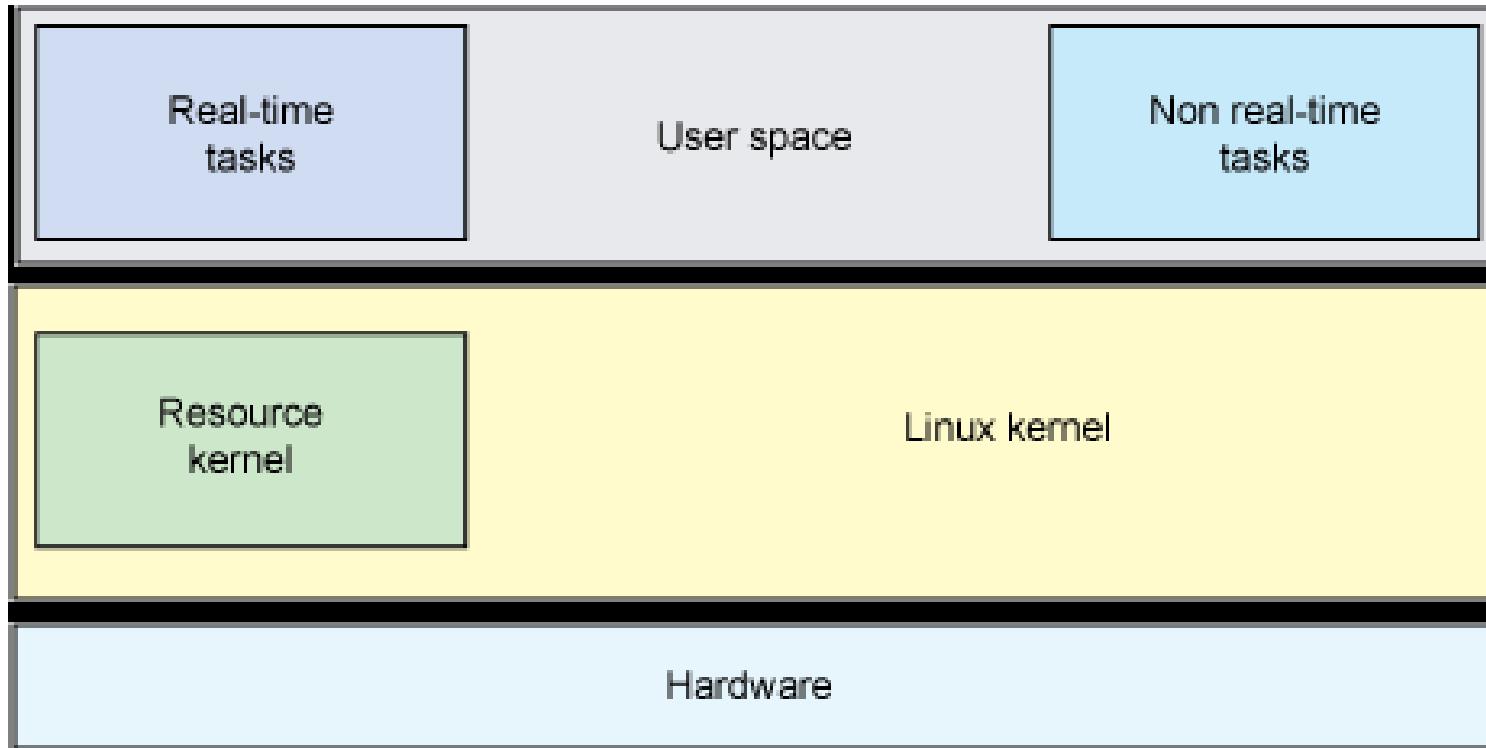


Noch kleinerer Kernel als Mikro-Kernel, der nur noch einen Hardware Abstraction Layer (HAL) enthält. Auf diesem können dann die Kerne mehrerer Betriebssysteme (inkl. RT-BS) aufsetzen.

Ähnlicher Ansatz wie Virtualisierung

Beispiel: ADEOS

Resource-Kernel Approach

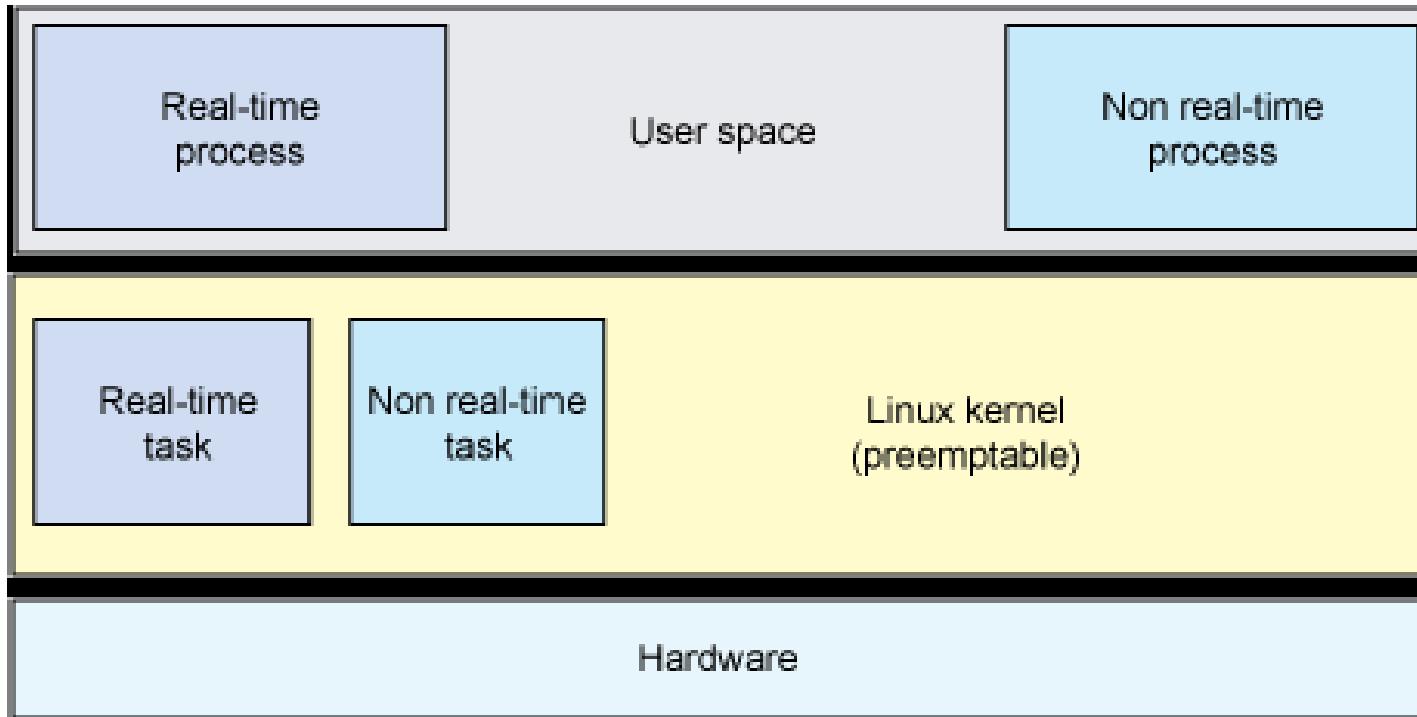


Modul innerhalb des Kernels, der die Ressourcen verwaltet (CPU, Netzwerk, Sekundärspeicher etc.).

Resource Kernel kann damit mittels geeignetem Schedulig (z. B. EDF) harte Echtzeit für RT-Tasks garantieren.

Beispiel: TimeSys Linux/RT

Standard Off-the-Shelf Linux Kernel 2.6



Ab Kernel 2.6
auch
Unterstützung
von (weicher)
Echtzeit!

Kernel Preemption: Optional kann eine niederpriore Task auch dann verdrängt werden, wenn sie gerade den Kernel benutzt (bei früheren Versionen nicht möglich).

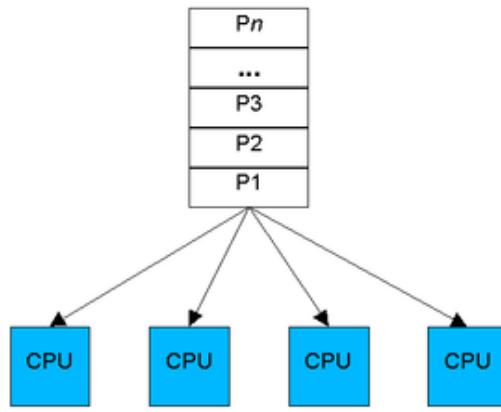
Schneller **O(1)-Scheduler** mit Priorisierung von RT-Tasks

Hochauflösende Timer (bis 1µs bei entsprechender Hardware-Unterstützung)

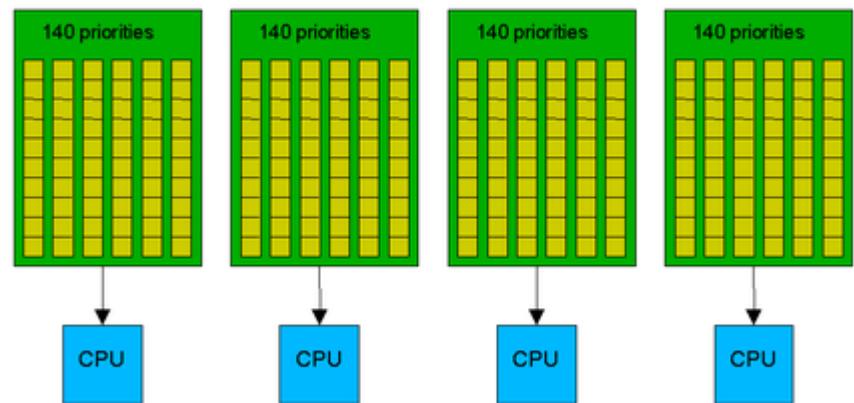
Linux Scheduler

Unterstützt Multiprozessoren mit gemeinsamem Speicher

Kernel 2.4



Kernel 2.6



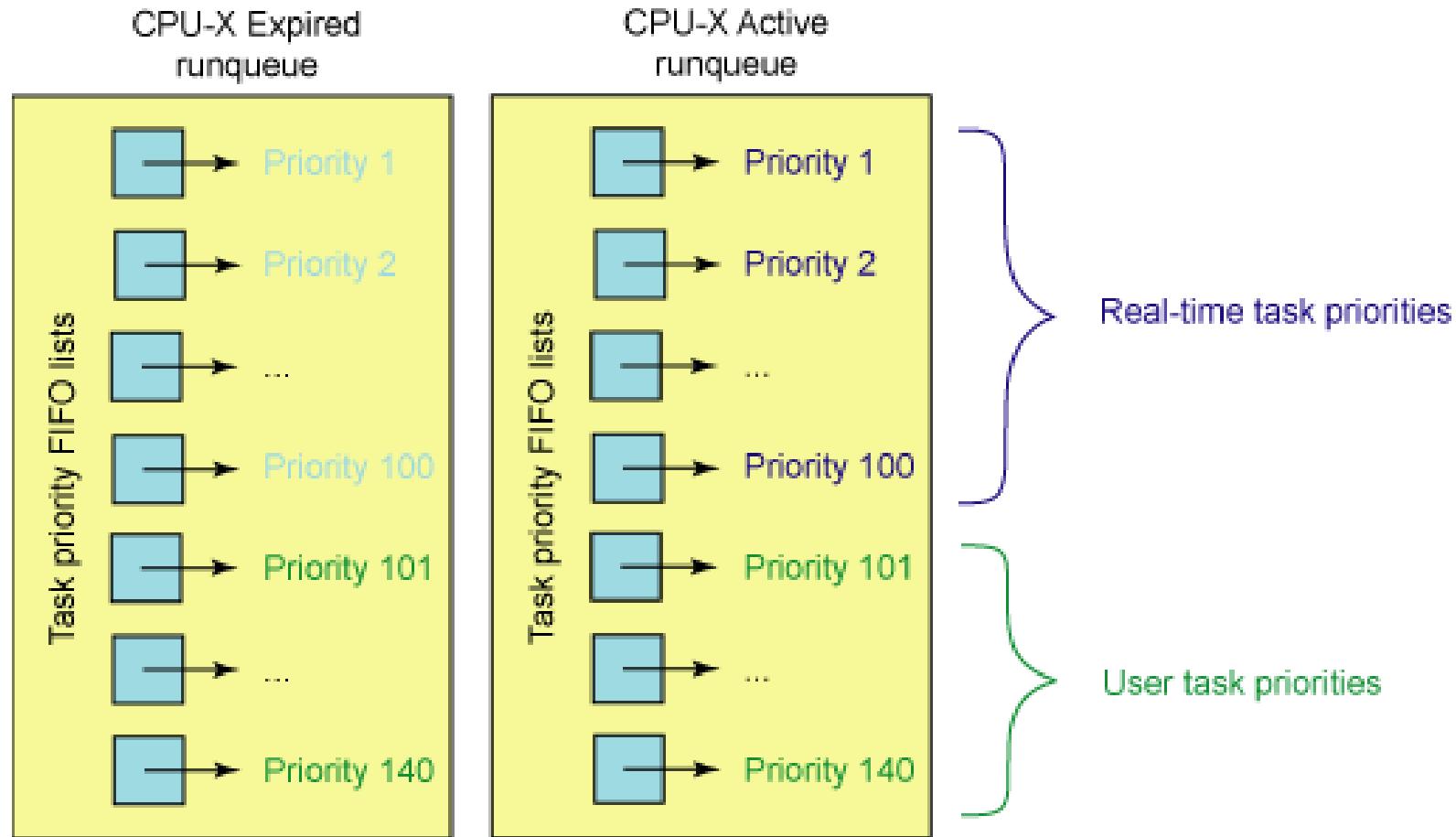
Eine **zentrale** Warteschlange, aus der der Scheduler die geeignete Task auswählt ($O(n)$)

Weder skalierbar noch echtzeitfähig!

Verteilte Warteschlangen mit **Prioritäten**, aus denen die CPU die höchspriore Task in konstanter Zeit ermitteln kann ($O(1)$).

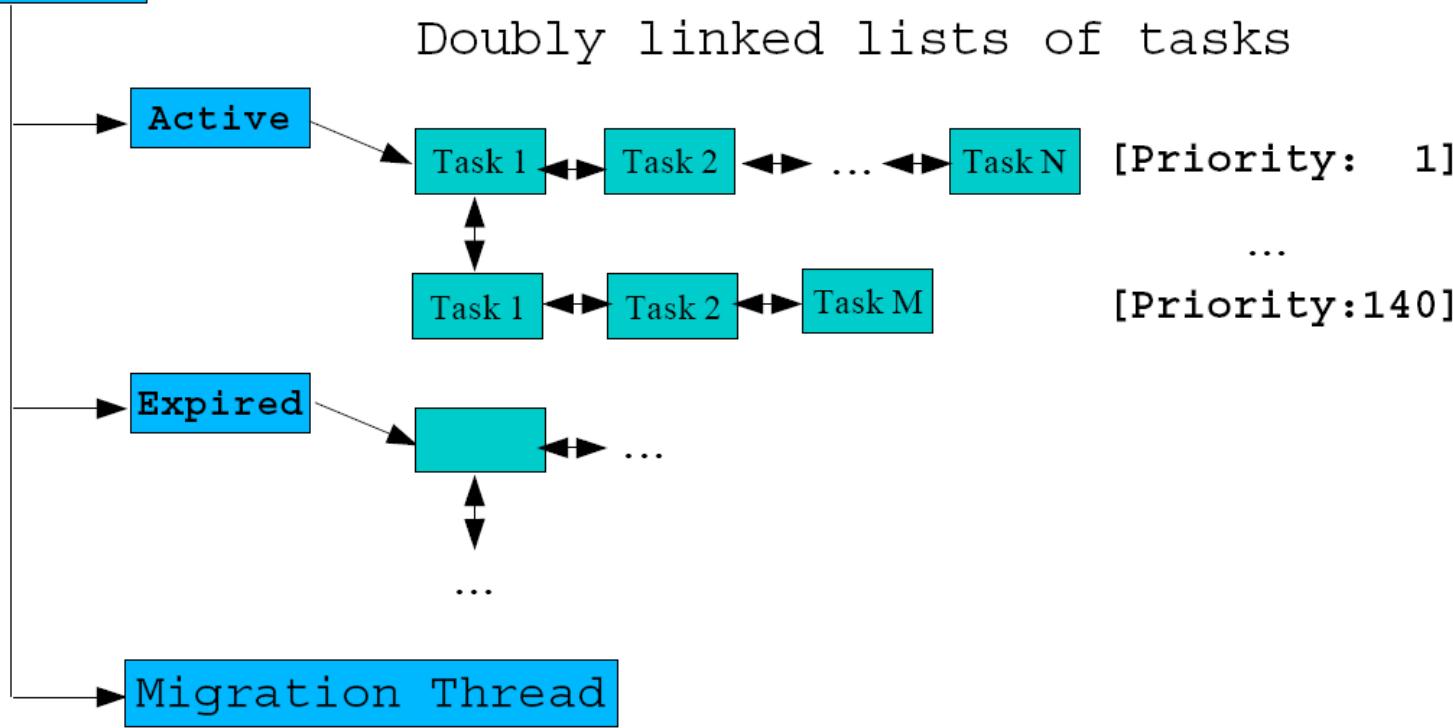
Skalierbar und echtzeitfähig!

Linux 2.6 Scheduler



Runqueue mit 140 Prioritäten, davon 100 für RT-Tasks
Expired Queue für Tasks, deren Zeitscheiben erschöpft sind

RunQueue



Pro Priorität doppelt verkettete Liste von aktiven Tasks

Ermittlung der höchspriorenen Liste mittels Bitoperationen (Position des höchstwertigen Bits, bei vielen Prozessoren eigener Befehl). Innerhalb der Liste Task 1

Lastausgleich zwischen Prozessoren durch Thread Migration

=> konstante Schedulingzeit, d. h. O(1)

Scheduling Policies

RT-Tasks

- statische Prioritäten (RMS)
- innerhalb einer Prioritätsklasse entweder ohne Verdrängung, bis die Task den Prozessor freigibt, oder Round-Robin mit Zeitscheiben

Nicht RT-Tasks

- dynamische Prioritäten
- interaktive Tasks bekommen dynamisch höhere Priorität als rechenintensive (Art der Tasks wird automatisch während der Laufzeit ermittelt)
- Wechsel der Prioritätsklasse nur bei Übergang in Expired Queue
- Umschalten auf Expired Queue, wenn Active Queue abgearbeitet, welche dann zur Expired Queue wird etc.

Lastausgleich zwischen Prozessoren durch eigenen Migration Thread, der regelmäßig aktiviert wird und Threads migriert.

3.1.8 Aktuelle Trends bei Echtzeitbetriebssystemen

- Einfache (8-Bit)-Mikrocontroller oft ohne eigenes Betriebssystem, d. h. Programmierung der „nackten“ Hardware, ggf. unterstützt durch Bibliotheken oder einfache OS-Kerne (meist nur Taskverwaltung).
- Spezialisierte Echtzeitbetriebssysteme (Embedded OS) (z. B. QNX, VxWorks, RTOS-UH, OSE, OS-9) für komplexere (16, 32-Bit)-Mikrocontroller, Einplatinen-Computer oder Industrie-PCs.
- Echtzeitversionen bekannter Desktop-Betriebssysteme (Windows CE, XP Embedded, Embedded Linux) immer weiter verbreitet, aber noch problematisch bzgl. harter Echtzeit und Zuverlässigkeit (nicht für kritische Anwendungen, aber Handies, PDAs etc.).
- Alternativ Realtime-Aufsätze für Windows (z. B. RTX): Echtzeittasks laufen direkt auf PC-Hardware (hart echtzeitfähig), Windows dazu nebenläufig (niedrigere Priorität).
- Unterstützung von Multiprozessoren (SMPs, heterogene MPs).
- Verteilte Echtzeitsysteme (z. B. vernetzt mit CAN-Bus, Ethernet).
- Standard-Schnittstellen wie POSIX mit Realtime Extensions

3.2 Echtzeit-Programmiersprachen

3.2.1 Anforderungen und Sprachtypen

Spezielle Anforderungen

- Formulierbarkeit von Zeitbedingungen
- Verwaltung paralleler Tasks/Threads
- Mechanismen zur Inter-Task-Kommunikation/ Synchronisation
- Unterbrechungs-Verarbeitung
- Hardwarenaher E/A-Zugriff (z. B. auf E/A-Register)
- Ausnahme- und Fehlerbehandlung

Dazu übliche Anforderungen an Programmiersprachen wie

- Modularität
- Unterstützung strukturierter Programmierung
- Objektorientierung

etc.

Sprachtypen

Assemblersprachen

Für kleine, zeitkritische Anwendungen (Mikrocontroller, Signalprozessoren) immer noch verbreitet.

Einbetten von Assemblerroutinen an zeitkritischen Stellen in Programme in höheren Programmiersprachen.

Da 'low-level' und prozessorabhängig, möglichst zu vermeiden!

Universelle höhere Programmiersprachen

Beispiele: C/C++, Basic, Java, ...

Erfüllen nicht die speziellen Anforderungen der Echtzeitprogrammierung.

⇒ Echtzeiterweiterungen erforderlich z. B. über Bibliotheken bzw. spezielle Varianten (z. B. Embedded Java).

C/C++ mit entsprechenden Bibliotheken heute in der Praxis dominierend.

Höhere Echtzeit-Programmiersprachen

Beispiele: ADA, PEARL, PL/M, PL/1, FORTH, OCCAM, Euclid, ...

Enthalten Konstrukte, die die speziellen Echtzeit-Anforderungen (mehr oder weniger) erfüllen.

Aber: Weniger komfortable Programmierumgebungen,
Akzeptanzprobleme ('exotisch').

Einsatz auf spezielle Anwendungsgebiete beschränkt (z. B. Militär, Prozessautomatisierung).

Anwendungsorientierte Programmiersprachen

Beispiele: KOP, AWL, FUP, AS, MatLab/Simulink, VHDL, LabView, ...

Zugeschnitten auf spezielle Anwendungsprobleme (z. B. SPS-Steuerungen, Signalverarbeitung, Hardwareentwurf, Prozessvisualisierung).

Problemorientierte, oft graphische Benutzeroberflächen.

Teilweise graphische Programmierung (KOP, FUP, AS, LabView etc.).
Auch von 'Nicht-Programmierern' leicht erlernbar und bedienbar.

In ihren jeweiligen Anwendungsbereichen oft weit verbreitet.

3.2.2 Echtzeitprogrammierung in C

C aufgrund seiner Hardwarenahe für Echtzeitprogrammierung gut geeignet.

Spezielle Bibliotheken/Erweiterungen für E/A-Schnittstellen (z. B. digitale E/A, Analog/Digitalwandler), Timer etc.

Insbesondere für einfache Mikrocontroller kompatible C-Entwicklungsumgebungen verfügbar (z. B. CodeWarrior für 68HC08, ICCAVR oder GNU für ATMEGA, C-Builder für x86).

Schnittstellen zu RT-Betriebssystemen (z.B. POSIX)

Programmierung an SPS-Arbeitsweise orientiert

Zyklische Arbeitsweise durch Timerinterrupt

- Netzwerke werden durch **IF/THEN-Regeln** dargestellt:

IF Prämissen THEN Konklusion

Prämissen: log. Verknüpfung von Eingaben und Variablen

Konklusion: Setzen von Variablen, Betätigen von Aktoren,
Ausgabe von Meldungen etc.

Schrittsteuerung über *Schrittmerker* und Gliederung in

- Ablaufebene
- Ausgabeebene
- Meldeebene

analog zur strukturierten SPS-Programmierung.

- *Techniken der SPS-Programmierung auch in C nachbildbar*
- *Hartes Echtzeitverhalten von SPSen auch in C für Mikrocontroller realisierbar.*

Beispielhaftes C-Programm

Objekt Timer, Eigenschaft auf 10ms Zykluszeit eingestellt.

Logale Variable deklarieren

Einlesen den Prozessabbildes

Ausmaskieren der Eingänge E0..E7

Grundstellung (GS) ermitteln

Ablaufstruktur bearbeiten

In der Ausgabeebene wird entsprechend der Schritte das Prozessabbild der Ausgänge (PAA) zugewiesen

In der Meldeebebene wird je nach Zustand der Eingänge bzw.Schritte ein Meldetext der Variablen Meldung zugewiesen.

Hier wird die Meldung an das Objekt Meldetext gesendet

```
graph TD; GS --> S1[S=1]; S1 --> S2[S=2]; S2 --> S0[S=0];
```

```
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{ int PAE,PAA;
  char Meldung[50];
  PAE=255->inportb(0x314);           //Prozessabbild einlesen
  E0=PAE&1; E1=PAE&2; E2=PAE&4; E3=PAE&8; E4=PAE&16; E7=PAE&128;
  // Teil 1 : Ablaufebene
  GS=E3 && !E4;                      // Berechnung des Grundschrittes
  if( GS && E0 && (S==0) ) S=1;//Schritt 1
  if( E4 && !E3 && (S==1) ) S=2; //Schritt 2
  if( GS && (S==2) ) S=0;
  if( !E1 )                           S=0; //Ausschaltbedingung
  if( !E7 )                           S=0; //Ausschaltbedingung
  //Teil 2 : Ausgabeebene
  if(S==0) PAA=0;
  if(S==1) PAA=64;                   //Kolben ausfahren
  if(S==2) PAA=128;                 //Kolben einfahren
  if( E2 && E1 && !S ) PAA=128;    //Richten
  outportb(0x318,PAA);              //Prozessabbild ausgeben
  //Teil 3 : Meldeebebene
  if ( E3 && E4 ) strcpy(Meldung,"Alarm,Sensor defekt");
  if (!E3 && !E4 && !S) strcpy(Meldung,"Kolben nicht in Endlage");
  if ( S==1 )                         strcpy(Meldung,"Kolben fährt aus");
  if ( S==2 )                         strcpy(Meldung,"Kolben fährt ein");
  if ( !E7 )                          strcpy(Meldung,"NOT-AUS eingerastet");
  Meldetext->Text=Meldung;
}
```

3.2.3 Echtzeitprogrammiersprache: PEARL (Process and Experiment Automation Realtime Language)

Entwickelt in Deutschland in den 70er Jahren.

In Europa bei der Prozessautomatisierung gewisse Verbreitung.

Erfüllt die wesentlichen speziellen Echtzeitanforderungen (parallele Tasks, Zeitverwaltung etc.).

Beispiele:

AFTER 10 SEC ALL 5 SEC DURING 70 MIN
ACTIVATE schütz PRIORITY 7;

Aktiviert Task 'schütz' nach 10s für einen Zeitraum von 70 min alle 5 s mit der Priorität 7.

AT 12:00:00 ALL 60 MIN UNTIL 24:00:00
ACTIVATE Protokoll

Task 'Protokoll' wird zwischen 12 Uhr und 24 Uhr jede Stunde gestartet.

Programmbeispiel: Bohrersteuerung

```
MODULE (Bohren);
SYSTEM;                                     /* Systemteil */
/*Beschreibung der Hardware-Struktur*/
PROBLEM;                                     /* Problemteil */
SPC (DrehzahlZumBohren, DrehzahlZumRuecklauf) INV FIXED;
SPC (MaximaleDrehzahl, Drehzahl_0) INV FIXED;
SPC (GalgengeschwZumBohren, GalgengeschwZumRuecklauf) INV FIXED;
SPC (Galgengeschw_0) INV FIXED;
SPC (GewuenschteBohrtiefe, Bohrtiefenbegrenzung) INV FIXED;
SPC (Bohrzeit, Ruecklaufzeit) DURATION GLOBAL;
SPC (AktuelleBohrtiefe, AktuelleDrehzahl) FIXED GLOBAL;
SPC BohrerInAusgangsposition: PROC GLOBAL;
SPC Bohren:PROC (Drehzahl FIXED, Geschwindigkeit FIXED) GLOBAL;

DCL (Maschinenfehler, Bohrtiefe_OK) INV FIXED GLOBAL INIT (-1,0);
DCL BohrErgebnis FIXED, GLOBAL INIT (Bohrtiefe_OK);
DCL FertigMitBohren SEMA GLOBAL PRESET (0);
DCL NORMAL INV FIXED(1) GLOBAL INIT(0);
DCL Eine_Sekunde INV DURATION INIT (1 SEC);

BohrSteuerung: TASK PRIO 50 GLOBAL;
CALL BohrerInAusgangsposition;
ALL 2 SEC ACTIVATE Ueberwachen;

CALL Bohren (DrehzahlZumBohren, GalgengeschwZumBohren);
While (AktuelleBohrtiefe LT GewuenschteBohrtiefe)
REPEAT
    Bohrzeit := ((GewuenschteBohrtiefe - AktuelleBohrtiefe) /
                  GalgengeschwZumBohren) * Eine_Sekunde;
    AFTER Bohrzeit RESUME;
END;

CALL Bohren (Drehzahl_0, Galgengeschw_0);

CALL Bohren (DrehzahlZumRuecklauf, GalgengeschwZumRuecklauf);
Ruecklaufzeit
    := (AktuelleBohrtiefe / GalgengeschwZumRuecklauf) * Eine_Sekunde;
AFTER Ruecklaufzeit RESUME;
CALL Bohren (Drehzahl_0, Galgengeschw_0);

PREVENT Ueberwachen;
RELEASE FertigMitBohren;
END;                                         /*BohrSteuerung*/
```

Ueberwachen: TASK PRIO 10;

```
IF ( (AktuelleBohrtiefe GT Bohrtiefenbegrenzung) OR  
(AktuelleDrehzahl GT MaximaleDrehzahl)  
)  
THEN  
TERMINATE BohrSteuerung;  
CALL Bohren (Drehzahl_0, Galgengeschw_0);  
BohrErgebnis:= Maschinenfehler;  
PREVENT; /* Ueberwachen plant sich selbst aus*/  
RELEASE FertigMitBohren;  
FIN; /* Ueberwachen*/  
END;  
MODEND;
```

Parallele Tasks:
BohrSteuerung,
Ueberwachen

Task **Bohr-Steuerung:**

- Fährt Bohrer in Ausgangsposition
- Startet Task **Ueberwachen** periodisch alle 2 s
- Startet Bohrer und suspendiert sich bis zum Ende der berechneten Bohrzeit
- Stoppt Bohrung
- Startet Bohrung zurückfahren und suspendiert sich für berechnete Rückfahrzeit
- Stoppt Bohrer
- Beendet Task **Ueberwachen**
- Meldet erfolgreiche Fertigstellung über Semaphore **FertigmitBohren**