Contents

1 Zentrale Beschreibegrößen **2**
- 1.0.1 Technischer Prozess . . . 2
- 1.0.2 Rechenprozesse . . . 3
- 1.0.3 System-software . . . 3

1.1 Realzeitbedingungen . . . 3
- 1.1.1 Auslastungsbedingung . . . 3
- 1.1.2 Rechtzeitigkeitsbedingung . . . 4
- 1.1.3 Harte und weiche Realzeit . . . 5

1.2 Systemaspekte . . . 5
- 1.2.1 Unterbrechbarkeit . . . 5
- 1.2.2 Prioritäten . . . 6
- 1.2.3 Ressourcenmanagment . . . 7

2 System-software **7**

2.1 Firmware . . . 7

2.2 RT-OS . . . 7
- 2.2.1 Systemcall-Interface . . . 8
- 2.2.2 Taskmanagment . . . 9
- 2.2.3 Scheduling . . . 13
- 2.2.4 Singlecore Scheduling . . . 14
- 2.2.5 Multicore Scheduling . . . 17
- 2.2.6 Memory Managment . . . 19
- 2.2.7 I/O Managment . . . 21

3 RT-Architekturen **24**

3.1 RT ohne Systemsoftware (deeplyembeeded system) . . . 24

3.2 RT basierend auf Standard OS (Linux) . . . 24

3.3 Threaded Interrupts . . . 24

3.4 Application to Kernel . . . 25

3.5 Multikernel Architektur . . . 25

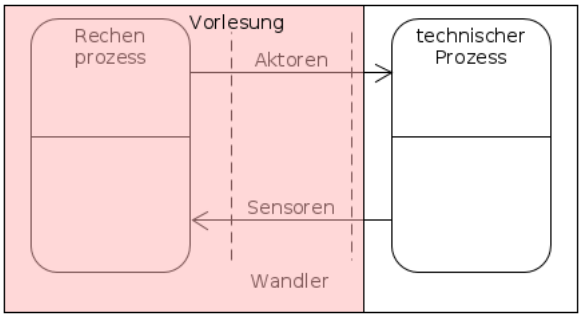
3.6 RT-Architektur auf Multicore-Basis . . . 25

3.7 RTOS . . . 26

1 Zentrale Beschreibegrößen

Definition: Realzeitsystem haben neben funktionalen Anforderungen auch zeitliche Anforderungen.

Ein Realzeitsystem besteht softwaretechnisch aus einer Reihe von Tasks und aus der System-Software.

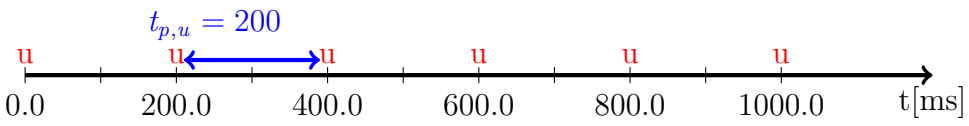


1.0.1 Technischer Prozess

Rechenzeitanforderung = Ereignis von technischen Prozess Releasetime = Zeitpunkt des Auftretens der RZ-Anforderung (RZ/RT = Realzeit)

Beispiel: periodisches Signal u alle 200ms

$t_{Release,u,1}$	=	0ms	$t_{Release,u,2}$	=	200ms
$t_{Release,u,3}$	=	400ms	$t_{Release,u,4}$	=	600ms



Prozesszeit = zeitlicher Abstand zwischen zwei RZ-Anforderungen gleichen Typs.

$t_{Pmin,i} = minimal => t_{max,i} = \frac{1}{t_{Pmin,i}}$

$t_{Pmax,i} = maximal$ <= uninteressant

$t_{Dmin,i}$ = minimal zulässige Reaktionszeit

$t_{Dmax,i}$ = maximal zulässige Reaktionszeit

Airbag:

t_{Dmax} = 50ms(Zeit bis zum Aufschlag) - 30ms(Zeit zum aufblasen) = 20ms

t_{Dmin} = 0ms

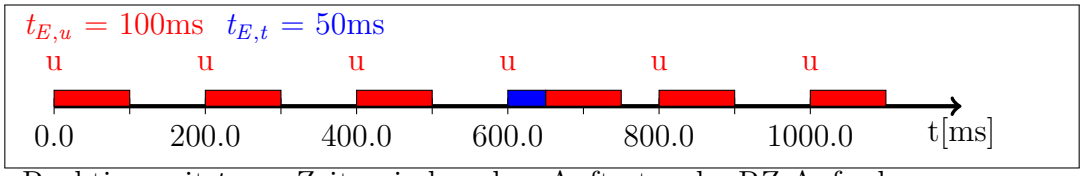
Phase = minimal Zeitlicher Abstand zwischen zwei unterschiedlicher RZ-Anforderungen $t_{Ph,i,j}$

1.0.2 Rechenprozesse

- Ausführuntgszeit (Executiontime) = Rechenzeit für eine RZ-Anforderung (ohne Warte oder Schlafzeiten)

  - WCET $t_{Emax,i}$ –> Erfahrung oder Messen Worstcase

  - BCET $t_{Emin,i} = 0$ Bestcase



- Reaktionszeit $t_{R,i}$ = Zeit zwischen dem Auftreten der RZ-Anforderungen i und dem Ende der Bearbeitung.

$T_{Rmax,i}$ = maximale Reaktionszeit
$T_{Rmin,i}$ = minimale Reaktionszeit
$T_{R,i} = t_{W,i} + t_{E,i}$ wobei $t_{W,i}$ Summe aller Wartezeiten

1.0.3 System-software

- Latenzzeit t_{L_i} = Zeit zwischen dem Auftreten einer RZ-Anforderung und dem Start der Bearbeitung - Interrup Latenzzeit - Tasklatenzzeit

1.1 Realzeitbedingungen

1.1.1 Auslastungsbedingung

$\rho_i = \frac{t_{E,i}}{t_{P,i}}$ Auslastung dur RZ-Anforderung i

$\rho_{max,i} = \frac{t_{Emax,i}}{t_{Pmin,i}}$ Worstcase, max. Auslastung

1. RT Bedingung

$$\rho_{max,ges} = \sum_{j=1}^{n} \frac{t_{Emax,j}}{t_{Pmin,j}} \leq c$$

j = für alle RZ-Anforderungen, c = Anzahl der Rechnerkerne

Beispiel: 2 RZ-Anforderungen A und B

$$\left.\begin{aligned} t_{Pmin,A} &= 2ms \\ t_{Emax,A} &= 0.8ms \end{aligned}\right\} \rho_{max,A} = \frac{0.8ms}{2ms} = 0.4ms$$

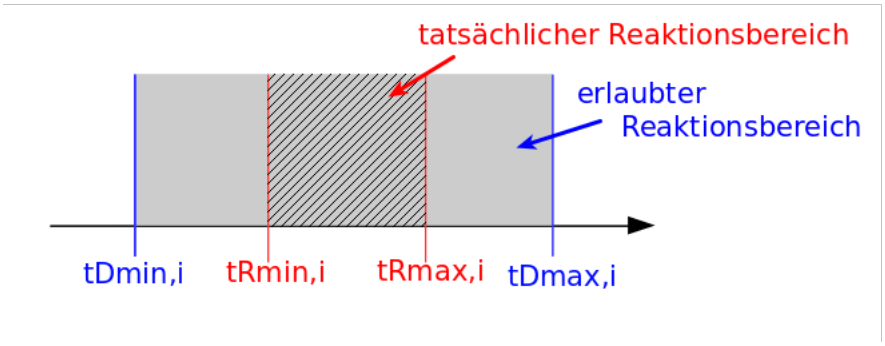
$$\left.\begin{aligned} t_{Pmin,B} &= 1ms \\ t_{Emax,B} &= 0.3ms \end{aligned}\right\} \rho_{max,B} = \frac{0.3ms}{1ms} = 0.3ms$$

$\rho_{max,ges} = \rho_{max,A} + \rho_{max,B} = 0.7 = 70\%$

Annahme Singlecore c = 1 $\rho_{max,ges} \leq c$ => $0.7 \leq 1$
Auslastungsbedingung erfüllt

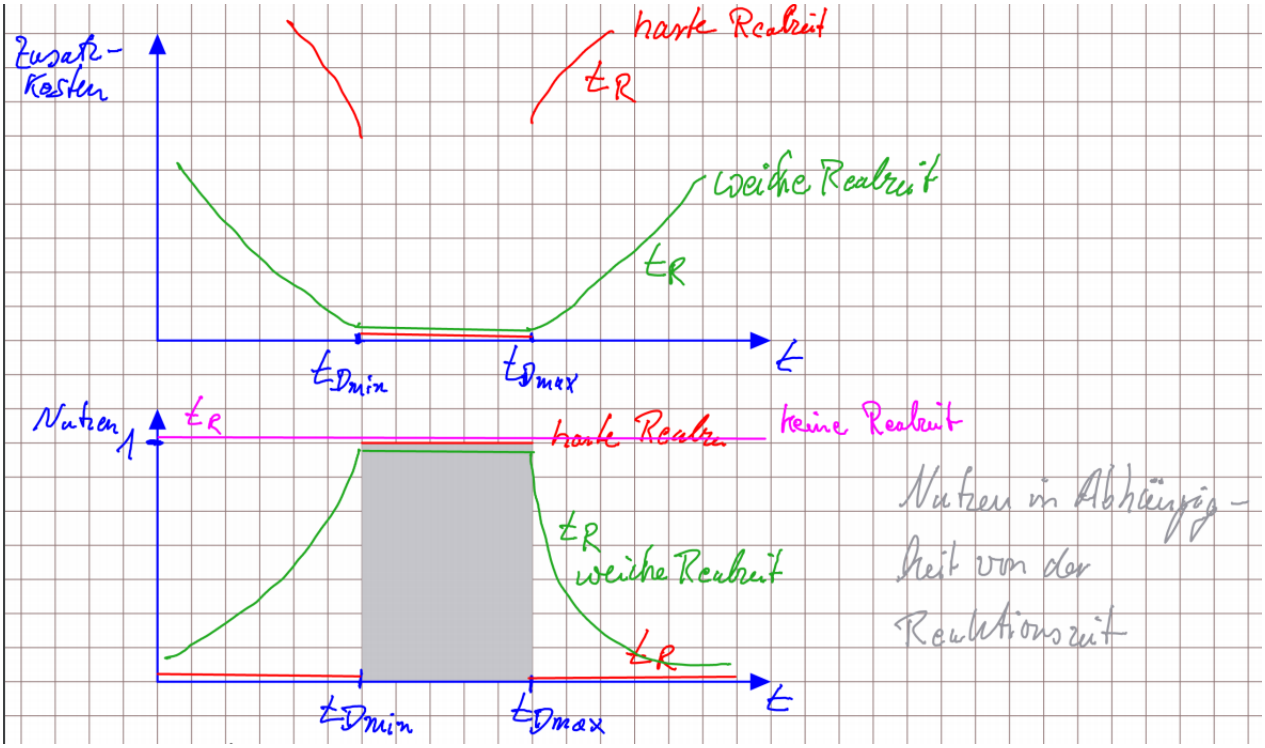
1.1.2 Rechtzeitigkeitsbedingung

Für den Realzeitbetrieb muss die tatsächliche Reaktion innerhalb des Zulässigen Reaktionsbereiches erfolgt sein.



2. RT Bedingung
Für alle RZ-Anforderungen j muss gelten:
$t_{Dmin,j} \leq t_{Rmin,j} \leq t_{Rmax,j} \leq t_{Dmax,j}$

1.1.3 Harte und weiche Realzeit



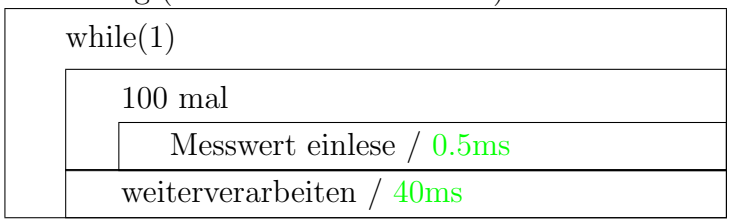
1.2 Systemaspekte

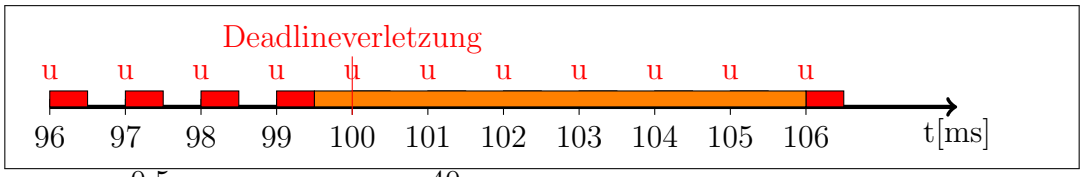
1.2.1 Unterbrechbarkeit

Forderung: Codesequenzen lassen sich in Teilsequenzen unterteilen, die in korrekter Reihenfolge aber unabhängig voneinander abgearbeitete werden können.
=> notwendig für den Realzeitbetrieb

Begründung: Ein Messwert soll kontinuierlich erfasst werden.
$t_{Emin,u} = t_{Emax,E} = 0.5ms$
Jeweils 100 Messwerte (alle 100ms) sollen weiterverarbeitet werden
$t_{Pmin,w} = 100ms$
$t_{Dmin,w} = 0ms$
$t_{Dmax,w} = 100ms$
$t_{Dmax,w} = 100ms$
$t_{Emin,w} = t_{Emax,w} = 40ms$

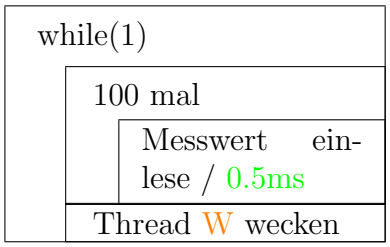
Lösung (ohne Unterbrechbarkeit):



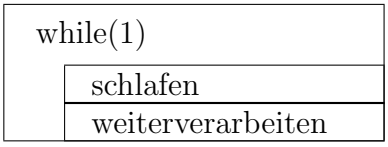


$\rho_{max,u} = \frac{0.5ms}{1ms} = 0.5$; $\rho_{max} = \frac{40ms}{100ms} = 0.4 => \rho_{max,ges} = 0.9$

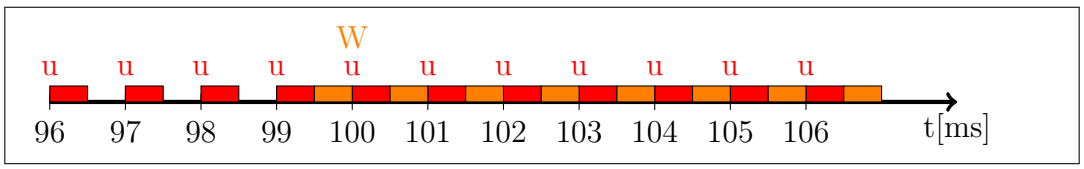
Lösung mit Unterbrechbarkeit:



Thread U



Thread W



Konsequenzen:

a) Inter-prozess-Kommunikation (IPC) (Sync, Datenaustausch)

b) Multithreading/Multitasking

1.2.2 Prioritäten

Forderung: Der Systemarchitekt muss einfluss auf die Abarbeitungsreihenfolge mehrerer Tasks nehmen können z.B. über Prioritäten.

1.2.3 Ressourcenmanagment

–> später

2 System-software

2.1 Firmware

Aufgabe:

- Basisinitialisierung der Hardware
- Diagnose
- Betriebsinitialisierung
- Laden + Aktivieren von Codes
- Runtime Services

Ausprägungen:

- BIOS
- UEFI
- Bootloader ("Das U-Boot")
- Monitor Software

2.2 RT-OS

Definition.: Bezeichnung für alle Software-Komponenten, die

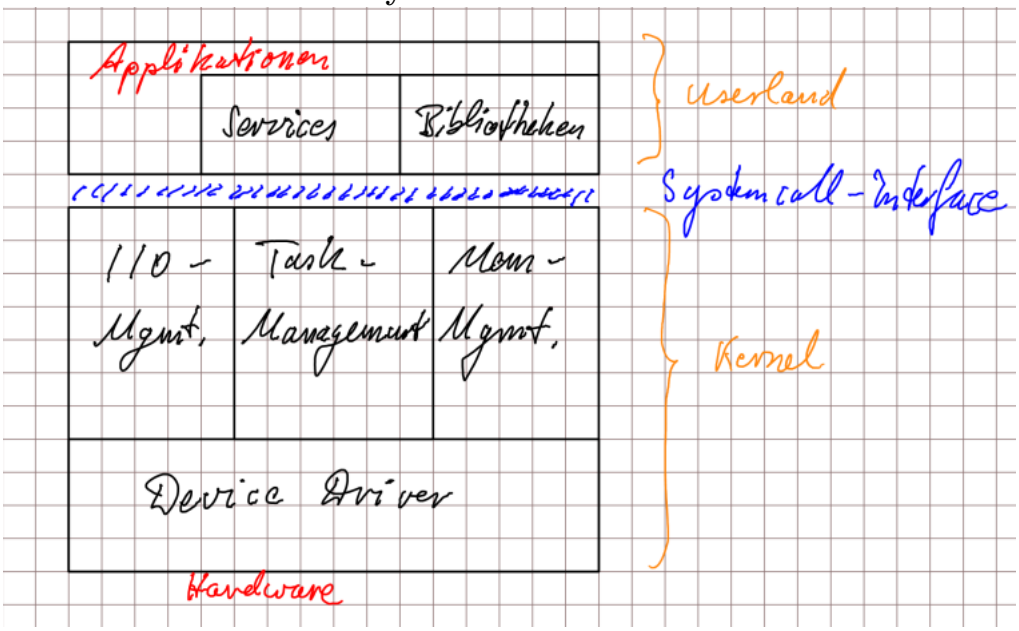
- die Ausführung der Applikationen und
- die Verteilung der Betriebsmittel (Memory, Files, CPU, Drucker, ...) ermöglichen, steuern und überwachen.

Anforderungen:

- Zeitverhalten
- Ressourcenverbrauch
- Zuverlässigkeit und Stabilität
- Sicherheit
- Flexibilität und Kompatibilität
- Portierbarkeit
- Skalierbarkeit

Beispiele: Sämtliche Betriebsysteme

Systemarchitektur



2.2.1 Systemcall-Interface

Systemcall = Dienst des Kernels –> 300-400 Dienste

Beispiele: open(), close(), read(), write(), exit(), fork(), clone(), clock_nanosleep(), kill(), adjtime(),...
Technische Realisierung: SW-Interrupt

Ablauf: ret = write(fd, "Hello World", 13);
↓ Systemcall "write" per SW-Interrupt
"int 0x80", "trap", "sysenter"
Systemcall mit EAR = 4 <– x86 Register

ISR (SW-Interrupt 0x80)
↓ EAX = 4 –> bedeutet write
vfs_write()
↓ wertet die übrigen CPU-Register aus
↓ fd –> entscheidet über den zu nutzenden Gerätetreiber
driver_write() –> gibt Hardwaretehcnisch die Daten aus

2.2.2 Taskmanagment

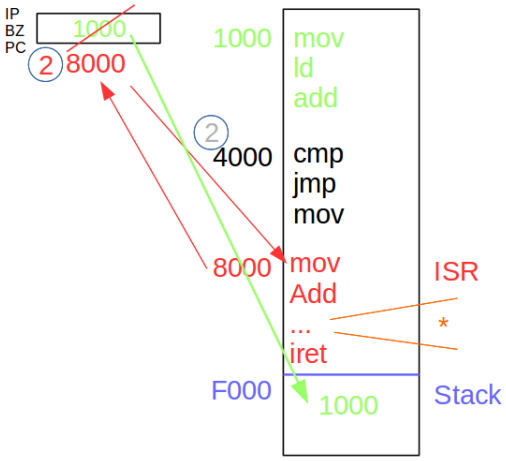
Aufgabe: Verwaltung der Ressource CPU

–> quasi parallele Verarbeitung auf einzelnen CPU-Kernen

–> real parallele Verarbeitung auf Multicore-Rechnern

Scheduling = Auswahl des als nächsten zu bearbeiten Jobs
Content Switch = Aktivierung eines Jobs

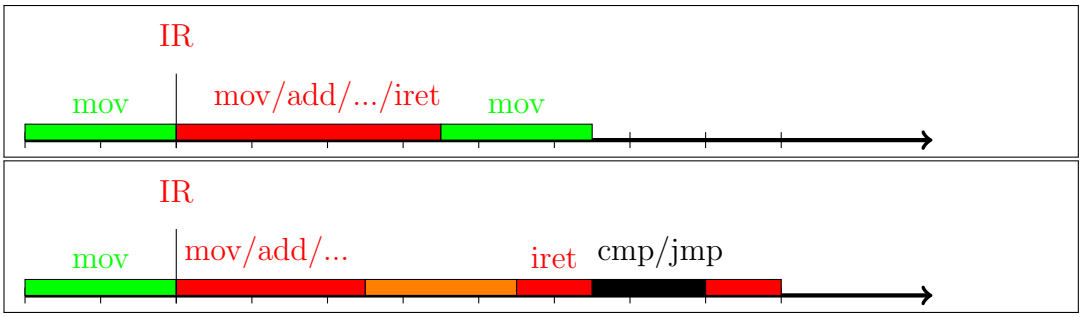
Singelcore-Scheduling Realisierung: Modifikation der Rücksprungadresse auf dem Stack beim Interrupt.



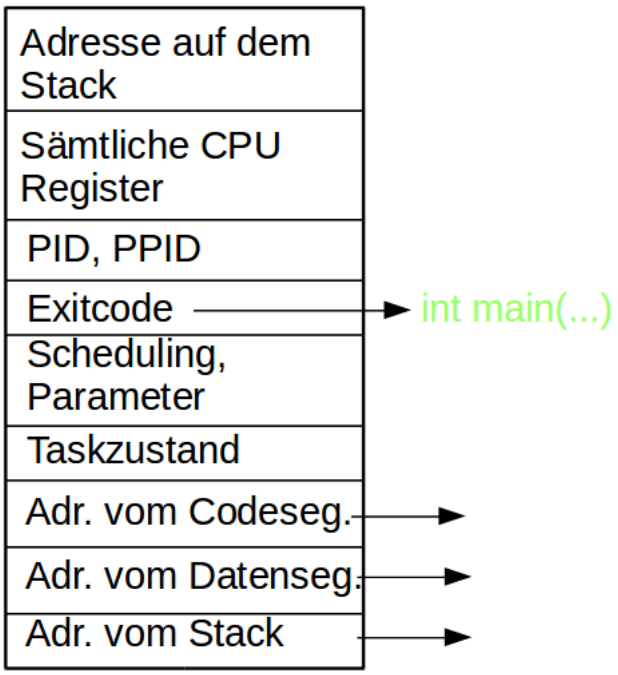
1. Code an der im IP stehenen Adresse wird abgearbeitete
2. IR tritt auf
   - Inhalt vom IP wird auf den Stack gelegt
   - IP wird auf die Adressee der ISR gelegt (CPU arbeitet die ISR ab)
3. Bei iret wird die auf dem stack hinterlegte Adressee zurück auf den IP geladen

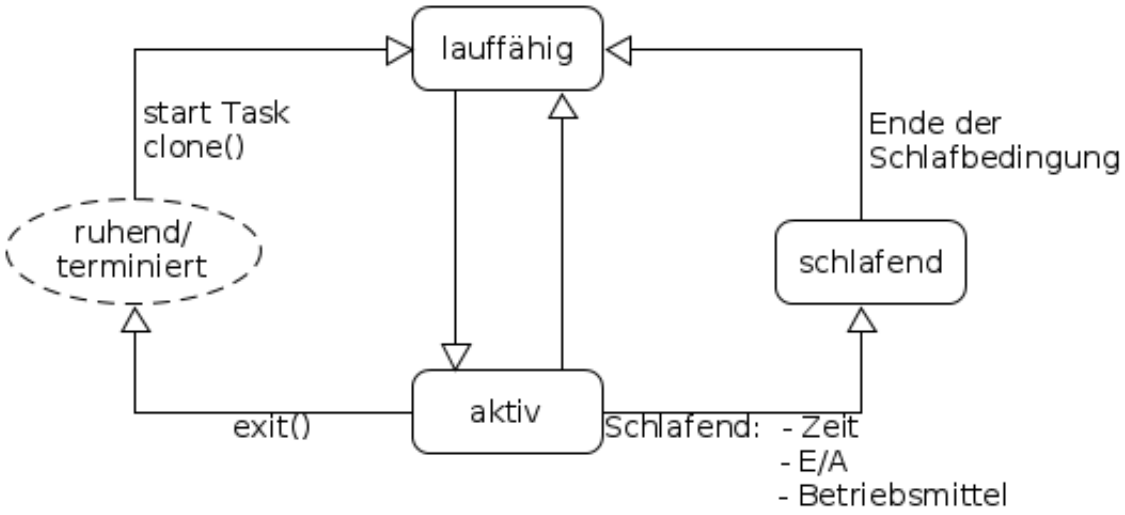
–> normale Verarbeitung wird fortgesetzt

* zusätlicher Code der die auf dem Stack liegende Adresse ändert
z.B. die 1000 wird mit 4000 überschrieben



Eine Datenstruktur wird benötigt, um alle Informationen zu einer Codesequenzen speichern(Job, Rechenprozess): Task Controll Block (TCB)





Erzeugen von Rechenprozessen Neue Jobs werden durch Kopieren von

- TCB

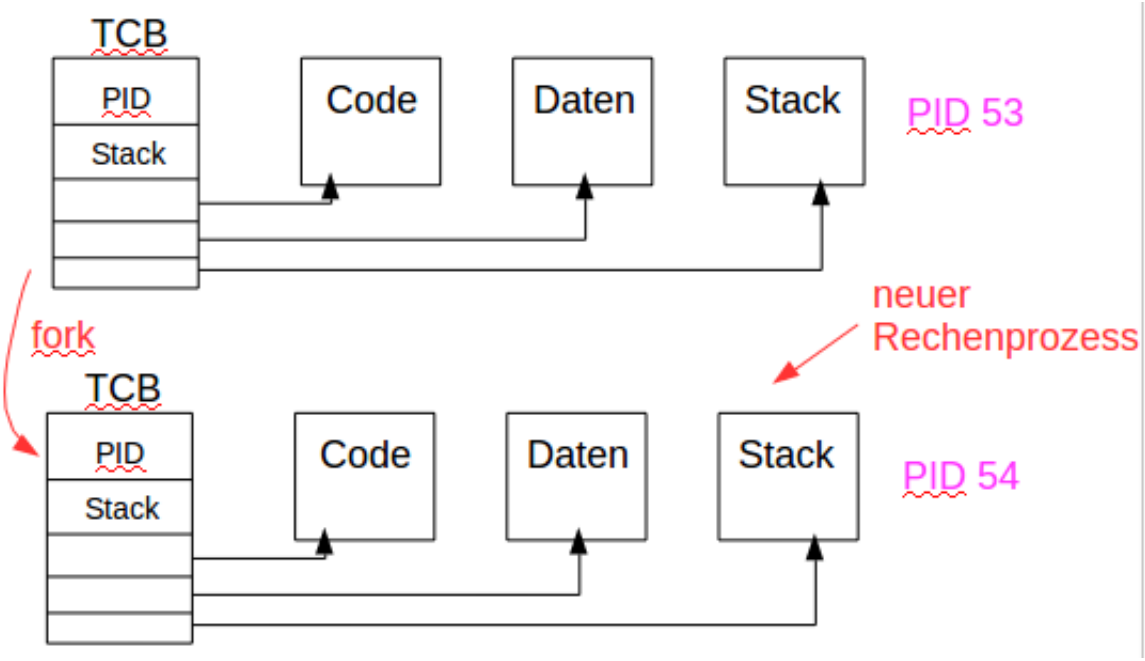
- Stack-Segment

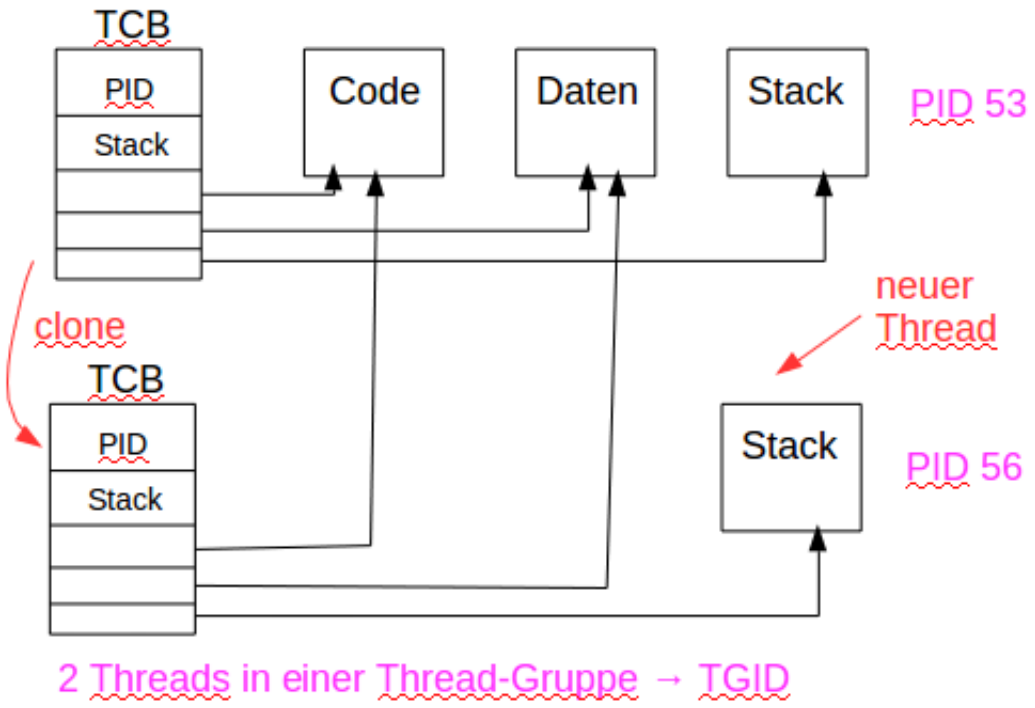
- Code-Segment

- Daten Segment

- Neue PID vergeben

erzeugt





Threads einer Threadgruppe teilen sich Code- und Datensegment Vorteil:

- Speicherplatzersparnis

- schnell erzeugt

- einfache IPC (Inter Process Communication)

Nachteil:

- Safety: Ein amok laufender Thread bringt die gesamte Threadgruppe in einen inkonsistenten Zustand.

2.2.3 Scheduling

Def.: Auswahl der Task, der arbeiten/rechnen darf
Content Switch: Aktivierung der ausgewählten Task
Statisches Scheduling: Bedarfsituation ist bekannt
↳ Reihenfolge (Plan) kann im vorhinein festgelegt werden (SPS)
Dynamisches Scheduling: Die Auswahl erfolgt auf Basis der aktuellen Bedarfsituation (z.B. PC, Smartphone)
Singlecore Scheduling: Quasiparallele Verarbeitung auf einem CPU-Kern
Multicore Scheduling: Realparallele Verarbeitung auf mehrerern CPU-Kernen

↳ Verteilungsaufgabe Preemption Point: Auftreten einer RF-Anforderung –> Interrup Serivce Routine(ISR) wird aktiv –> Scheduling

2.2.4 Singlecore Scheduling

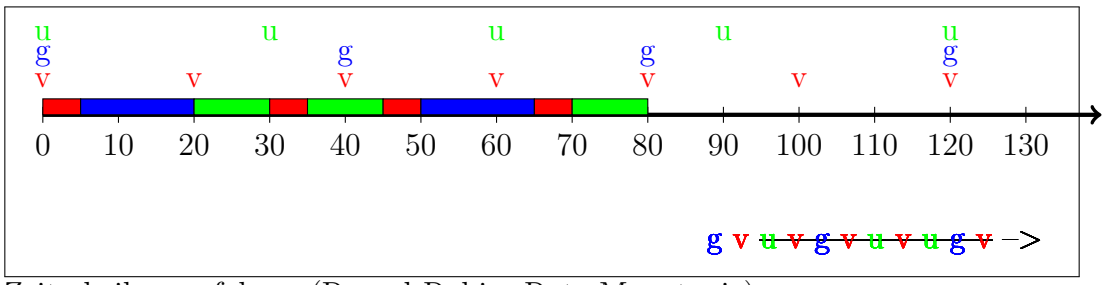
Table 1: Beispieldaten:

RZ-Anf	t_{Pmin}	t_{Dmin}	t_{Dmax}	t_{Emin}	t_{Emax}
v	20ms	0ms	20ms	1ms	5ms
g	40ms	0ms	40ms	1ms	15ms
u	30ms	0ms	30ms	1ms	10ms

First come First Serve (FCFS, FIFO)

- Lauffähige Jobs werden gemäß Auftrittszeitpunkt in eine Queue eingeführt
- Der erste Job in der Liste wird ausgewählt
- Er darf solange die CPU benutzen (rechnen) bis
    a) er sich schlafen legt
    b) er sich beendet

Ein "aufgeweckter" Job wird an den Anfang der Queue gestelt und unterbricht den laufenden Job nicht.



Zeitscheibenverfahren (Round Robin, Rate Monotonic)

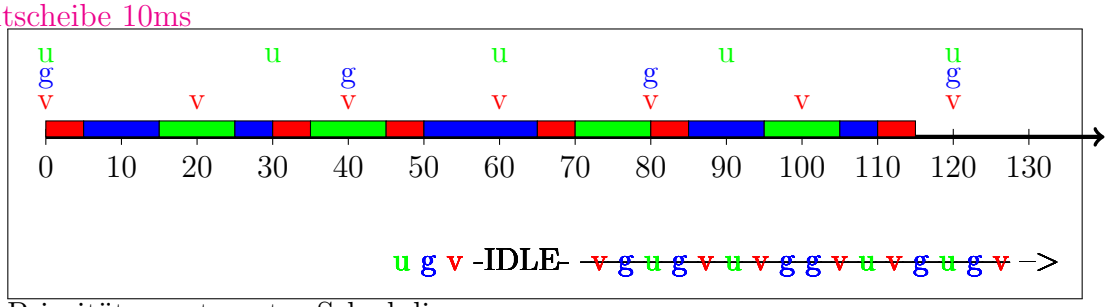
- Lauffähige Jobs werden gemäß Auftrittszeitpunkt in eine Queue eingehängt
- Der erste Job in der Liste wird ausgewählt
- Er darf die CPU benutzen bis
    a) er sich schlafen left

b) er sich beendet

c) seine Zeitscheibe (Quantum) aufgebraucht ist

- Ein aufgeweckter Job wird ans Ende der Queue angehängt

Zeitscheibe 10ms



Prioritätengesteuertes Scheduling

- Jedem Job wird eine Priorität zugewiesen

- Der Lauffähige Job mit der höchsten Priorität wird ausgewählt

- Er darf rechnen bis

a) er sich schlafen legt

b) er sich beendet

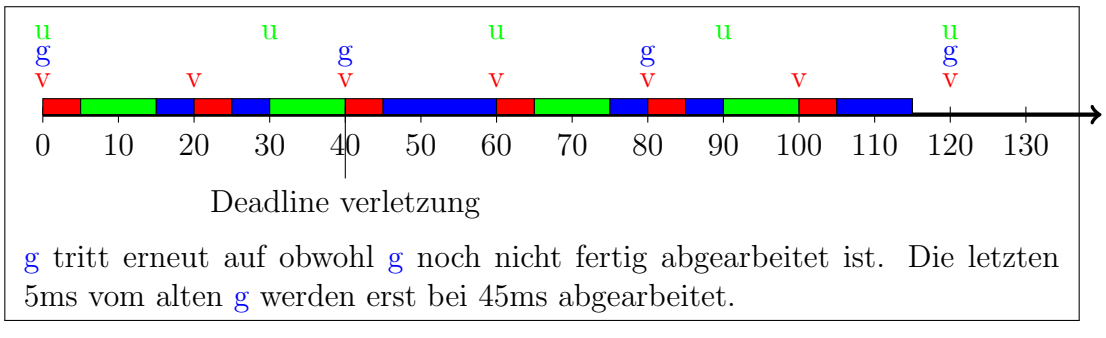
c) ein Job mit höherer Prio Lauffähig wird

Problem: Verteilung der Prioritäten

kurze t_{Emax} und kurze t_{Pmin} –> hohe Prio

Gibt es zwischen $t_{E,i}$ und $t_{P,i}$ keine Korrelation hilft bei der Prioritätenverteilung nur ausprobieren!

Beispiel: V = 1 (höchste Prio)
G = 3 (niedrigste Prio)
U = 2 (mittlere Prio)



g tritt erneut auf obwohl g noch nicht fertig abgearbeitet ist. Die letzten 5ms vom alten g werden erst bei 45ms abgearbeitet.

Deadline-Scheduling (Earliest Deadline First)

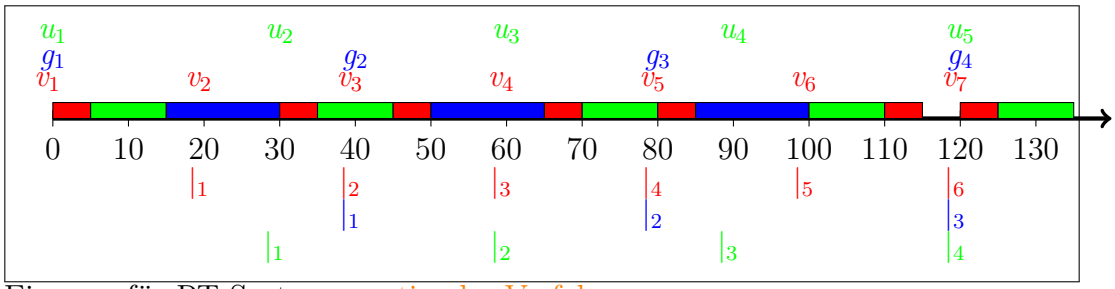
- Der Lauffähige Job, der als erstes fertig sein muss (t_{Dmax}) darf rechnen.

- Er darf arbeiten bis

  a) er sich schlafen legt

  b) er sich beendet

  c) ein Job Lauffähig wird, der früher abgeschlossen sein muss.



Eignung für RT-Systeme: optimales Verfahren

Kombinierte Schedulingverfahren

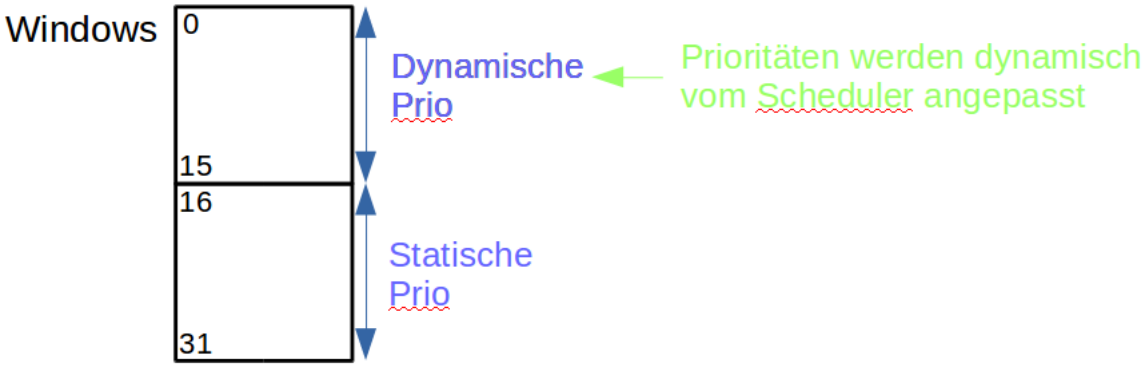
- Prioritätengesteuertes Scheduling –> Prioritätsebene

- Jobs mit gleiche Priorität (=auf gleicher Prioritätsebene)

$$\left.\begin{array}{l} werden \;\; gemaess \;\; a) RoundRobin \\ \qquad\qquad\quad\;\; b) FCFS (FIFO) \end{array}\right\} \text{Posix-Scheduling}$$

gescheduled => Linux,

$$\underbrace{VxWorks, Windows}_{\text{Round Robin}}$$



Linux: Prioritäten per Konsole vergeben: chrt 69 ./carrera // werte von 0..99
Konzept der Scheduling Klassen:

0 Stop-Sched-Class

1 Prioritäten gesteuertes Scheduling

2 EDF (Earliest Deadline first)

3 CFS (Completly Fair Scheduling)

4 Idle Shed. Class

2.2.5 Multicore Scheduling

Aufgabe: Verteilung der Jobs auf die CPU-Kerne, so dass unsere Zeitbedingunen eingehalten werden und das System (eneregie-)effizient arbeitet.

Lösungen:

1. Partitioniert Scheduling
   Tasks werden auf die CPU-Kerne Statisch verteilt.(per Hand)

2. Semipartioniertes Scheduling
   tastk werden in Gruppen auf die CPU-Kerne verteilt.

3. Globales Scheduling
   Scheduler verteilt die Tasks auf Basis der aktuellen Lastsituation.

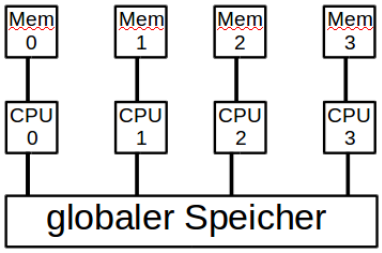
Taskmigration: Verschiebung von Tasks auf andere CPU-Kerne
Problemstellung:

- Kosten der Taskmigration sind abhängig von der eingesetzten Hardware

- Nutzen ist abhängig von der eingesetzten Hardware

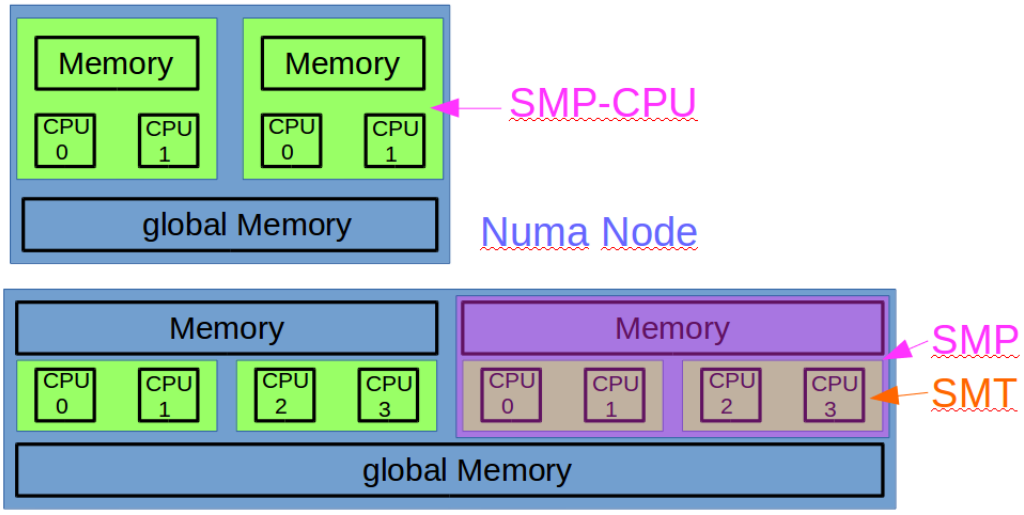
Hardware-Architekturen SMP: Symmetirc Multi Processing (= alle CPU Kerne sind gleich)
Kosten: mittel
Nutzen: mittel
SMT: Symmetric Multi Threading (Hyperthreading, = Verdopplung der Instruction Pipeline)
Kosten: niedrig
Nutzen: niedrig

NUMA: Non Uniform memory Architecture
Kosten: hoch
Nutzen: hoch

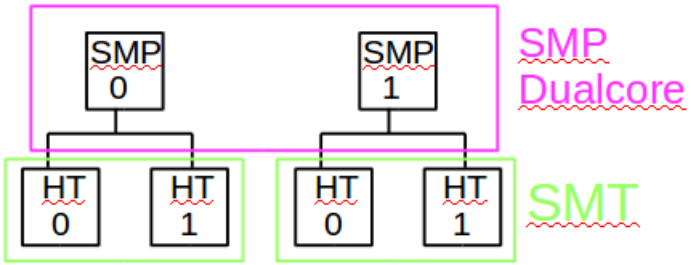


bigLITTLE-Architektur: (z.B. 4(starke) + 4(schwache) Kerne)
Optimierungsziel: Energieeffizienz
Kosten(in hinblick auf Leistung) = SMP(mittel)
Nutzen = SMP(mittel)

AMP: Asymmetirc Multiprocessing (unterschiedliche CPU-Kerne)
Kosten: hoch
Nutzen: je nach Anwendung
=> wird in der Praxis über Partitioniertes Scheduling genutzt.



Linux bildet beim Booten eine Scheduling Domain



Der Multicore Scheduler balanziert die last innerhalb einer Scheduling Domain –> sorgt für ausgeglichene Lastverhältnisse.
Die einzelnen Cores verwenden einen Singlecore Scheduler. Unter Linux ist der name der Rechenprozesse, die für Multicore-Scheduling zuständig sind "migration".

Der Multicore Scheduler wird aktiv:

- exit()
- pthread_create(), clone(), fork()
- clock_nanosleep()
- zeitgesteuert

2.2.6 Memory Managment

Aufgaben:

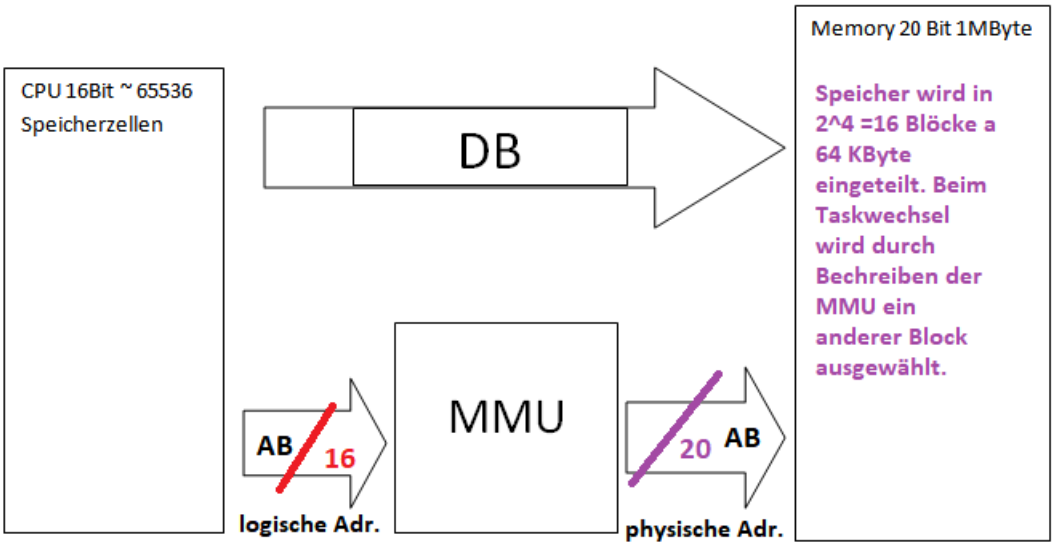
- Speicherschutz
- Adressumsetzung
- virtuellen Speicher zur verfügung stellen
- erweiterten Speicher zur verfügung stellen

Technologien:

1. Segmentierung
2. Paging (Seitenorientierung)

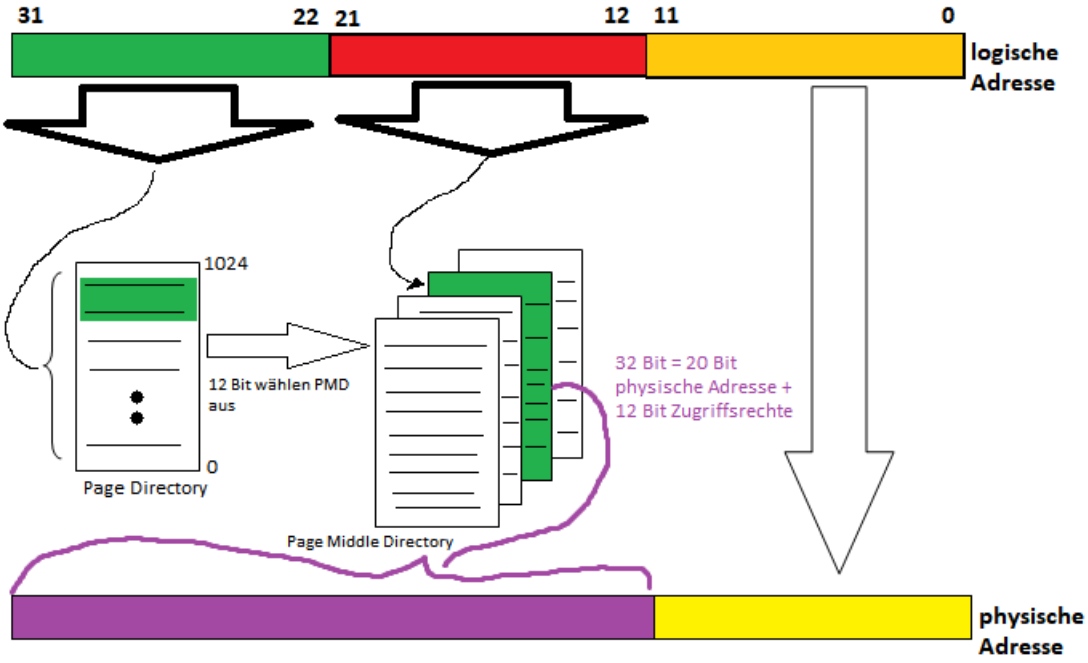
Auf 32 Bit Systemen: Two Level Paging
Auf 64 Bit Systemen: Three Level paging

Segmentierung



Konsequenz: Um eine Variable aus dem Hauptspeicher zu lesen, sind auf einem 32 Bit System 3 Hauptspeicherzugriffe notwendig.
=> zur Optimierung TCB (Cache)
Außerdem: In den Page Directories sind die obersten Einträge (auf 32Bit 15 Byte) für den Kernelspace reserviert.

Paging



Auf 32 Bit Systemen: Two Level Paging
Auf 64 Bit Systemen: Three Level Paging

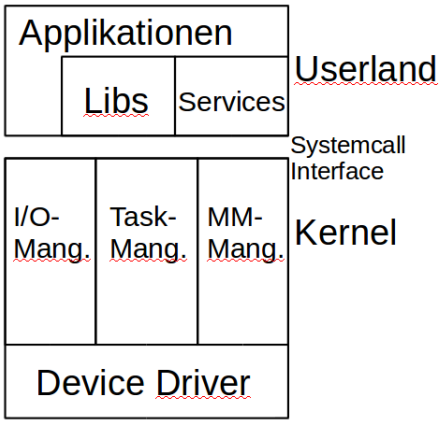
2.2.7 I/O Managment

Aufgabe:

a) Einheitliche API für den HW Zugriff

b) Systemkonforme Integration von Hardware über Gerätetreiber

c) Strukturierter Zugriff auf Daten (Filesystem)



API: open, cloes(), read(), write(), ioctl(), fcntl(), seek()

Hintergrund: Direct I/O - Bufferd I/O
printf("\n Hallo"); <– Buffered
printf("Hallo \n");
Buffered I/O => Daten werden aus Performance-Gründen zwischengespeichert. Reale Ausgabe erfolgt, wenn der Zwischenspeicher (Buffer) voll ist oder wenn ein "\n" kommt.

fopen(), fclose(), fprintf(), fwrite(), fread(), fflush()

Direct I/O => Ein-/Ausgabe-Aufrufe werden direkt ausgeführt.

Kontrollfluss (Wann?, Unter welchen Umständen?)

Zugriffsarten:

- Blockierend

- nicht Blockierend

- Asynchron

a) Blockierend
Beispiel "read": Job schläft, bis die angeforderten Daten zur Verfügungn stehen –> Carrear Bahn, eigene Spur

b) Nicht Blockierend fd = open("dev/carrera", O_RDWR|O_NONBLOCK)
Beispiel "read": Job bekommt die angeforderten Daten oder die Information, dass zu Zeit des Aufrufes keine Daten zur Verfügung stehen.
–> Job wird nicht schlafen gelegt.

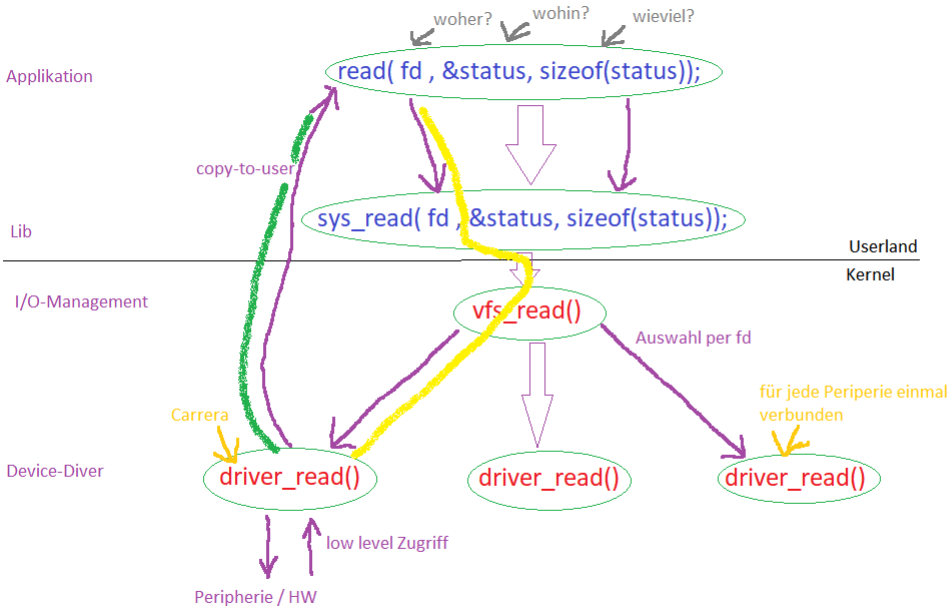
c) Asynchron
Aufträge werden dem Kernel übergeben und das Resultat zu einem späteren zeitpunkt abgeholt.

Systemkonforme Integration von Hardware über Gerätetreiber

Idee: open, close, read, write, ... bieten einen einheitlichen Zugriff auf unterschiedliche Peripherie

Außerdem: Applikationen sollen keinesfalls direkt auf Peripherie zugreifen können (Safety)

Und: Applikationen sollen keine Details der Hardware (z.B. Adressen, Bitmaskierungen, Gerätetreiber) kennen müssen.



Filesysteme Strukturen zur Ablage von Daten auf Hintergrundspeicher in hierarchisch organisierten Dateien.

Beispiel: Ext4, NTFS, FAT32, exFat, JFFS2, ISO9660, ...

Merkmale:

- maximale FileSystem-Größe
- Anzahl Dateien
- maximale Datei Größe
- Attribute
- Zugriffszeit

3 RT-Architekturen

3.1 RT ohne Systemsoftware (deeplyembeeded system)

- Keine Systemsoftawre (evtl. Monitorsoftware)
- Single Threaded
- Parallelität über ISR (Interrupt Service Routine)
- Preiswerte HW -> 8Bit/16Bit
- Programmierung: Assembler/C

3.2 RT basierend auf Standard OS (Linux)

- Linux, Windows -> Embedded Variante
- Vorteil: Bekannte Schnittstellen für Entwickler und Anwender
- Nachteil: (Kosten), HW-Anforderungen, 32Bit+MMU, Sicherheit/Safety, RT-Fähigkeit, Haltedauer

3.3 Threaded Interrupts

- Idee: "Alles ist ein Thread" => auch ISR
- Vorteil:
  - Priorisierung aller Komponenten
  - ISR kann schlafen
  - übersichtliche Systemarchitektur
- Nachteil: häufige Kontextwechsel
- Funktionsweise: Kurzse ISR weckt den Schlafenden "Interrupt"-Thread auf

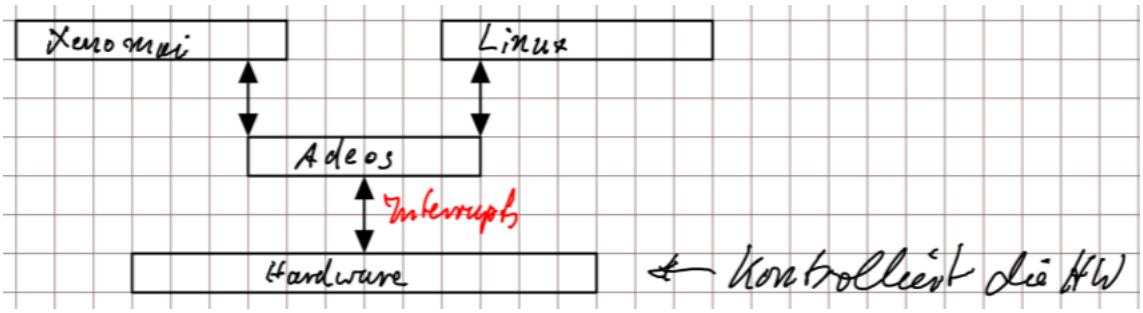
3.4 Application to Kernel

- Zeitliche Teile der Applikation werden in den Kernel verlagert
  ↳ Kernelmodul, Gerätetreiber
- Nachteil: Verquickung von System und Anwendung
  ↳ hochgradigk unsauber!
  ↳ Securtiy
  ↳ Safety
- Vorteil: gute RT-Eigentschaften
- Beispiel: Raspberry Pi als digitales Speicheroszilloskop

3.5 Multikernel Architektur

Basistechnologie Virtualisierung mehrerer Betriebsysteme auf 1 Hardware.
RTOS + Standard-OS

RT-Hypervisor mit Standard-OS als IdleTask



- Vorteil: gute RT-Eigentschaften
- Nachteil: 2 Systeme + Hypervisor
  Einarbeitungs und Pflegeaufwand

3.6 RT-Architektur auf Multicore-Basis

Einzelne CPU-Kerne eines Multicore-Rechners werden exklusiv für Realzeitaufgaben reserviert.
Technologien: - Affinität
- Isolation

Affinität: Jobs werden fest Prozessorkernen zugeordnet.
Isolation: Der CPU-kern steht nur den zugeordneten Threads zur Verfügung

Jobs werden von Hand auf die Kerne verteilt (partitionieren)
↳ Jeder Kern wird als Singelcore-System betrachtet
↳ für jeden Kern ist ein RT-Nachweis durchzuführen

3.7 RTOS

z.B. VxWorks, Free RTOS

- Vorteil:
  - sehr schlank (z.B. 16kByte)
  - häufig Zertifiziert (Safety)
  - gute RT-Eigentschaften
- Nachteil:
  - proprietär
  - evtl. teuer (VxWorks)