

BS

Volkmar Sieh

WS 2019/20

Inhaltsverzeichnis

1	Einstieg in die Betriebssystementwicklung	2
1.1	Übersetzen und Linken	2
1.2	Booten	2
1.3	Debugging	3
2	Unterbrechungen	5
2.1	Hardware	5
2.1.1	Grundlegende Fragestellungen	5
2.1.2	Hardware-Architekturen	7
2.2	Software	8
2.2.1	Begriffe und Grundannahmen	8
2.2.2	Zustandssicherung	10
2.2.3	Zustandsänderungen	11
2.3	SoftIRQ	11
2.3.1	Beispiele	11
2.3.2	Aufteilung Interrupt-Bearbeitung	12
2.3.3	SoftIRQs	12
2.3.4	SoftIRQ-Beispiel	13
2.3.5	Verwandte Konzepte	13
2.4	Synchronisation	14
2.4.1	Prioritätsebenenmodell	14
2.4.2	Harte Synchronisation	15
2.4.3	Weiche Synchronisation	15
2.4.4	Prolog/Epilog-Modell	16
3	IA-32: Die 32-Bit-Intel-Architektur	18
3.1	Urvater: Der 8086	18
3.2	Die 32-Bit Intel-Architektur	19
3.3	Protectect Mode	20
3.4	Multitasking	21
3.5	AMD64	21
4	Koroutinen und Programmfäden	22
4.1	Grundbegriffe	22
4.2	Implementierung	23
5	Scheduling	24
5.1	Betriebssystemfäden	24
5.2	Kooperativer Fadenwechsel	24
5.3	Präemptiver Fadenwechsel	24
5.4	Ablaufplanung	24
5.4.1	Grundbegriffe und Klassifizierung	24
5.4.2	unter Windows	26
5.4.3	unter Linux	26
6	Betriebssystem-Architekturen	27
6.1	Paradigmen der Betriebssystementwicklung	27
6.1.1	Bibliotheks-Betriebssystemen	27
6.1.2	Monolithen	28
6.1.3	Mikrokerne	29
6.1.4	Exokerne und Virtualisierung	30
6.1.5	Fazit	32
7	Fadensynchronisation	32
7.1	Prioritätsebenenmodell mit Fäden	32
7.2	Mechanismen	33
7.2.1	Randbedingungen	33
7.2.2	Mutex, Implementierungsvarianten	33
7.2.3	Passives Warten	34
7.2.4	Semaphore	34
7.3	Beispiel: Windows	35
8	Gerätetreiber	36
8.1	Anforderungen an das BS	36
8.1.1	Einheitlicher Zugriff	36
8.1.2	Spezifischer Zugriff	37
8.2	Struktur des E/A-Systems	37
9	Interprozesskommunikation	38
9.1	IPC über Speicher	39
9.2	IPC über Nachrichten	40
9.3	Basisabstraktionen	40
9.4	Trennung der Belange mit AOP	41
9.5	Fazit	42

1 Einstieg in die Betriebssystementwicklung

1.1 Übersetzen und Linken

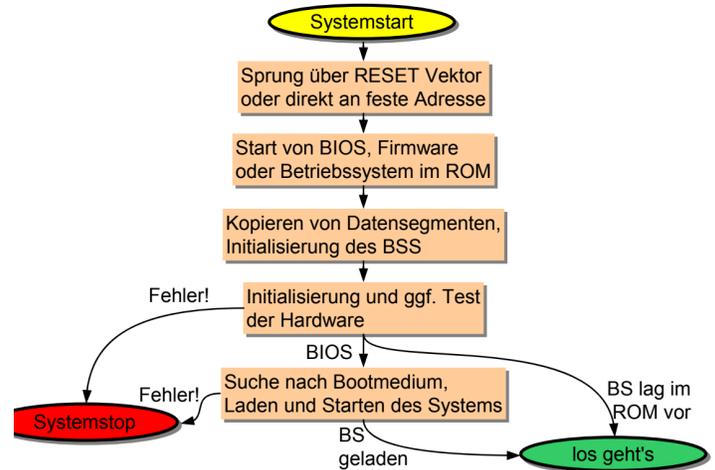
- kein dynamischer Binder vorhanden
 - alle nötigen **Bibliotheken statisch einbinden**.
- libstdc++ und libc benutzen Linux Systemaufrufe (insbesondere write)
 - die normalen **C/C++ Laufzeitbibliotheken können nicht benutzt werden**. Andere haben wir (meistens) nicht.
- generierte Adressen beziehen sich auf virtuellen Speicher! ("nm hello — grep main" liefert "0804846c T main")
 - die Standardeinstellungen des Binders können nicht benutzt werden. **Man benötigt eine eigene Binderkonfiguration**.
- der Hochsprachencode stellt Anforderungen (*Registerbelegung, Adressabbildung, Laufzeitumgebung, Stapel, ...*)
 - ein eigener **Startup-Code** (in Assembler erstellt) muss die Ausführung des Hochsprachencodes vorbereiten

1.2 Booten

Bootstrapping „Bootstrapping (englisches Wort für Stiefelschleife) bezeichnet einen Vorgang bei dem ein einfaches System ein komplexeres System startet. Der Name des Verfahrens kommt von der **Münchhausen-Methode**.“

Münchhausen-Methode „Die Münchhausen-Methode bezeichnet allgemein, dass ein System sich selbst in Gang setzt. Die Bezeichnung spielt auf die deutsche Legende von **Baron Münchhausen** an, der sich an seinen eigenen Haaren aus einem Sumpf gezogen haben soll. In der amerikanischen Fassung benutzte er seine Stiefelschlaufen, was die englische Bezeichnung **Bootstrapping** für diese Methode begründete.“

Bootvorgang



- Bootsektor
 - das PC BIOS lädt den 1. Block (512 Bytes) des Bootlaufwerks an die Adresse 0x7c00 und springt dorthin (blind!)
 - Aufbau des „Bootsektors“:

OOSTuBS

Offset	Inhalt
0x0000	<code>jmp boot;</code>
0x0004	Anzahl der Spuren
0x0006	Anzahl der Köpfe
0x0008	Anzahl der Sektoren
0x000a	reservierte Sektoren (Setup-Code)
0x000c	reservierte Sektoren (System)
0x000e	BIOS Gerätecode
0x000f	Startspur der Diskette/Partition
0x0010	Startkopf der Diskette/Partition
0x0011	Startsektor der Diskette/Partition
0x0010	<code>boot:</code>
	...
0x01fe	<code>0xaa55</code>

Wichtig ist eigentlich nur der Start und die „**Signatur**“ (**0xaa55**) am Ende. Alles weitere benutzt der **Boot-Loader**, um das eigentliche System zu laden.

- Boot Loader
 - einfache, systemspezifische Boot Loader
 - * Herstellung eines definierten Startzustands der Hard- und Software
 - * ggf. Laden weiterer Blöcke mit Boot Loader Code
 - * Lokalisierung des eigentlichen Systems auf dem Boot-Medium
 - * Laden des Systems (mittels Funktionen des BIOS)
 - * Sprung in das geladene System

- "Boot Loader auf nicht boot-fähigen Disketten
 - * Ausgabe einer Fehlermeldung und Neustart
- Boot Loader mit Auswahlmöglichkeit
 - * (z.B. im Master Boot Record einer Festplatte)
 - * Darstellung eines Auswahlmenüs
 - * Nachbildung des BIOS beim Booten des ausgewählten Systems
 - Laden des jeweiligen Bootblocks nach 0x7c00 und Start
- Speicher, Geräte (selbst Sound- und Netzwerkkarte)
- selbst Windows und Linux Systeme laufen in Bochs
- * implementiert in C++
- * Entwicklungsunterstützung
 - Protokollinformationen, insbesondere beim Absturz
 - eingebauter Debugger (GDB-Stub)

1.3 Debugging

• „printf – Debugging“

- gar nicht so einfach, da es printf() per se nicht gibt!
 - * oftmals gibt es nicht mal einen Bildschirm
- printf() ändert oft auch das Verhalten des debuggee
 - * mit printf() tritt der Fehler nicht plötzlich nicht mehr / anders auf
 - * das gilt gerade auch bei der Betriebssystementwicklung
- Strohhalm
 - * eine blinkende LED
 - * eine serielle Schnittstelle

• (Software-)Emulatoren

- ahmen reale Hardware in Software nach
 - * einfacheres Debugging, da die Emulationssoftware in der Regel kommunikativer als die reale Hardware ist
 - * kürzere Entwicklungszyklen
- Vorsicht: am Ende muss das System auf realer Hardware laufen!
 - * in Details können sich Emulator und reale Hardware unterscheiden!
 - * im fertigen System sind Fehler schwerer zu finden als in einem inkrementell entwickelten System
- übrigens: "virtuelle Maschinen und Emulatoren sind **nicht** gleichbedeutend
 - * in VMware wird z.B. kein x86 Prozessor emuliert, sondern ein vorhandener Prozessor führt Maschinencode in der VM direkt aus
- **Beispiel "Bochs"**
 - * emuliert i386, ..., Pentium, AMD64 (Interpreter)
 - optional MMX, SSE, SSE2 und 3DNow! Instruktionen
 - Multiprozessoremulaton
 - * emuliert kompletten PC

• Debugging

- ein Debugger dient dem Auffinden von Softwarefehlern durch Ablaufverfolgung
 - * in Einzelschritten (single step mode)
 - * zwischen definierten **Haltepunkten** (breakpoints), z.B. bei
 - Erreichen einer bestimmten **Instruktion**
 - Zugriff auf ein bestimmtes **Datenelement**
- Vorsicht: manchmal dauert die Fehlersuche mit einem Debugger länger als nötig
 - * wer **gründlich nachdenkt kommt oft schneller zum Ziel**
 - Einzelschritte kosten viel Zeit
 - kein Zurück bei versehentlichem Verpassen der interessanten Stelle
 - * beim printf-Debugging können Ausgaben besser aufbereitet werden
 - * Fehler im Bereich der Synchronisation nebenläufiger Aktivitäten sind interaktiv mit dem Debugger praktisch nicht zu finden
- praktisch: Analyse von "core dumps"
 - * beim Betriebssystembau allerdings weniger relevant

• Funktionsweise

- praktisch alle CPUs unterstützen das Debugging
- Beispiel: Intels x86 CPUs
 - * die **INT3** Instruktion löst "breakpoint interrupt" aus (ein TRAP)
 - wird gezielt durch den Debugger im Code platziert
 - der TRAP-Handler leitet den Kontrollfluss in den Debugger
 - * durch Setzen des **Trap Flags (TF)** im Statusregister (EFLAGS) wird nach jeder Instruktion ein "debug interrupt" ausgelöst
 - kann für die Implementierung des Einzelschrittmodus genutzt werden
 - der TRAP-Handler wird nicht im Einzelschrittmodus ausgeführt

- * mit Hilfe der **Debug Register DR0-DR7** (ab i386) können bis zu vier Haltepunkte überwacht werden, ohne den Code manipulieren zu müssen

- erheblicher Vorteil bei Code im ROM/FLASH oder nicht-schreibbaren Speichersegmenten

- besonders effektiv wird Debugging, wenn das Programm im Quelltext visualisiert wird (**source-level debugging**)

- * erfordert Zugriff auf den Quellcode und Debug-Informationen
- * muss durch den Übersetzer unterstützt werden (z.B. -g Flag)

• Remote Debugging

- bietet die Möglichkeit Programme auf Plattformen zu debuggen, die (noch) kein interaktives Arbeiten erlauben

- * setzt eine Kommunikationsverbindung voraus (seriell, Ethernet, ...)
- * erfordert einen Gerätetreiber
- * der Zielrechner kann auch ein Emulator sein (z.B. Bochs)

- die Debugging-Komponente auf dem Zielsystem (stub) sollte möglichst einfach sein

- Beispiel gdb

- * das **Kommunikationsprotokoll** (*"GDB Remote Serial Protocol RSP"*)
 - spiegelt die Anforderungen an den gdb stub wieder
 - basiert auf der Übertragung von ASCII Zeichenketten
 - Nachrichtenformat:

\$ < Kommando oder Antwort > # < Prüfsumme >

- Nachrichten werden unmittelbar mit + (OK) oder - (Fehler) beantwortet

- das Kommunikationsprotokoll – **kompletter Umfang**

- * Register- und Speicherbefehle
- * Steuerung der Programmausführung
 - letzte Unterbrechungsursache abfragen
 - Einzelschritt
 - mit Ausführung fortfahren
- * Sonstiges
 - Ausgabe auf der Debug Konsole
 - Fehlernachrichten

• Debugging Deluxe

- viele Prozessorhersteller integrieren heute Hardwareunterstützung für Debugging auf ihren Chips (**OCDS – On Chip Debug System**)

- * BDM, OnCE, MPD, JTAG

- i.d.R. einfaches serielles Protokoll zwischen DebuggingEinheit und externem Debugger (Pins sparen!)

- Vorteile:

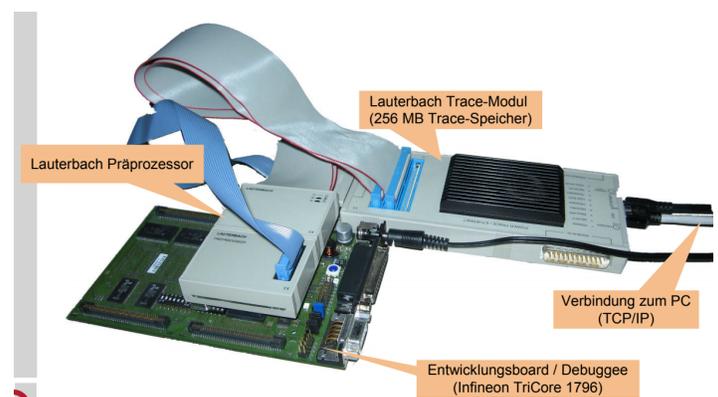
- + der Debug Monitor (z.B. gdb stub) belegt keinen Speicher
- + Implementierung eines Debug Monitors entfällt
- + Haltepunkte im ROM/FLASH durch Hardware-Breakpoints
- + Nebenläufiger Zugriff auf Speicher und CPU Register
- + mittels Zusatzhardware ist zum Teil auch das Aufzeichnen des Kontrollflusses zwecks nachträglicher Analyse möglich

- Beispiel BDM

- * "Background Debug Mode eine on-chip debug Lösung von Motorola
- * serielle Kommunikation über drei Leitungen (*DSI, DSO, DSCLK*)
- * BDM Kommandos der 68k und ColdFire Prozessoren
 - RAREG/RDREG – Read Register
 - WAREG/WDREG – Write Register
 - READ/WRITE – Read Memory/Write Memory
 - DUMP/FILL – Dump Memory/Fill Memory
 - BGND/GO – Enter BDM/Resume

- Hardware-Lösung

- * Lauterbach Hardware-Debugger



- * Alle Hardwareregister im Zugriff (auch Peripherie)
- * Flexible bedingte Breakpoints
- * Umfangreiche Skriptsprache
- * Trace-Einheit
 - Taktgenaue Zeitmessungen von beliebigem Code

- Exakte Abbildung des Kontrollflusses auch bei Unterbrechung / Ausnahme / Taskwechsel
- „Rückwärts“ steppen
- ...

2 Unterbrechungen

2.1 Hardware

- Ein Blick zurück in die Historie von Betriebssystemen...
- Überlappte Ein-/Ausgabe:
 - Eingaben: Verschwendung von anderweitig nutzbaren Prozessorzyklen bei (oft nicht vorhersagbar langem) aktivem Warten
 - Ausgaben: selbständiges Agieren der E/A-Geräte (z.B. durch DMA) entlastet die CPU
- Timesharing Betrieb:
 - Zeitgeber-Unterbrechungen geben dem Betriebssystem die Möglichkeit
 - * zur Verdrängung von Prozessen
 - * Aktivitäten zeitgesteuert zu starten

2.1.1 Grundlegende Fragestellungen

☞ Priorisierung

- **Problem:**
 - * Mehrere Unterbrechungsanforderungen können gleichzeitig signalisiert werden. Welche ist wichtiger?
 - * Während die CPU auf die wichtigste Anforderung reagiert, können weitere Anforderungen signalisiert werden.
- **Lösung:** ein **Priorisierungsmechanismus** ...
 - * **in Software:** die CPU hat nur einen IRQ (interrupt request) Eingang und fragt die Geräte in bestimmter Reihenfolge ab
 - * **in Hardware:** eine Priorisierungsschaltung ordnet Geräten eine Priorität zu und leitet immer nur die dringendste Anforderung zur Behandlung weiter
 - * **mit festen Prioritäten:** jedem Gerät wird statisch eine Priorität zugeordnet
 - * **mit variablen Prioritäten:** Prioritäten sind dynamisch änderbar oder wechseln zum Beispiel zyklisch

☞ Verlust von IRQs

- **Problem:**
 - * während der Behandlung oder Sperrung von Unterbrechungen, kann die CPU keine neuen Unterbrechungen behandeln

- * die Speicherkapazität für Unterbrechungsanforderungen ist endlich. (*i.d.R. ein Bit pro Unterbrechungseingang*)
- **Lösung:** in Software
 - * die Unterbrechungsbehandlungsroutine sollte möglichst kurz sein (zeitlich!), um die Wahrscheinlichkeit von Verlusten zu minimieren
 - * Unterbrechungen sollten nicht unnötig lange gesperrt werden
 - * jeder Gerätetreiber sollte davon ausgehen, dass **eine** Unterbrechung **mehr als eine** abgeschlossene E/A Operation anzeigen kann

☞ Zuordnung einer Behandlungsroutine

- **Problem:**
 - * die Software soll mit möglichst wenig Aufwand herausfinden können, welches Gerät die Unterbrechung ausgelöst hat. (*eine sequentielle Abfrage der Geräte kostet nicht nur Zeit, sondern verändert die Zustände von E/A Bussen und unbeteiligten Geräten*)
- **Lösung:**
 - * jeder Unterbrechung wird eine Nummer zugeordnet, die als Index in eine Vektortabelle verwendet wird
 - die Vektornummer hat nicht zwangsläufig etwas mit der Priorität zu tun
 - es kommt in der Praxis leider vor, dass Geräte sich eine Vektornummer teilen müssen (interrupt sharing)
 - * der Aufbau der Vektortabelle variiert je nach Prozessortyp
 - meist enthält sie Zeiger auf Funktionen
 - seltener sind die Einträge selbst bereits Instruktionen

die vorherige Priorität wird auf einem Stapel abgelegt
Zustandssicherung

- **Problem:**
 - * nach der Ausführung der Behandlungsroutine soll zum normalen Kontext zurückgekehrt werden können
 - * die Behandlung soll quasi unbemerkt ablaufen (transparency)
- **Lösung:**
 - * Zustandssicherung durch Hardware
 - nur das Notwendigste: z.B. Rücksprungadresse u. Prozessorstatuswort
 - Wiederherstellung durch speziellen Befehl, z.B. iret, rte, ...
 - * Zustandssicherung durch Software

- da Unterbrechungen jederzeit auftreten können, muss auch die Behandlungsroutine Zustände sichern und wiederherstellen

☞ Geschachtelte Behandlung

– Problem:

- * um auf sehr wichtige Ereignisse schnell reagieren zu können, soll auch eine Unterbrechungsbehandlung unterbrechbar sein
- * eine unbegrenzte Schachtelungstiefe muss aber vermieden werden

– Lösung:

- * die CPU erlaubt immer nur Unterbrechungen mit höherer Priorität
- * die aktuelle Priorität wird im Prozessorstatuswort gespeichert
- * die vorherige Priorität wird auf einem Stapel abgelegt

☞ Interrupt-Ablauf

HW E/A-Gerät signalisiert Interrupt an Interrupt-Controller

HW Interrupt-Controller signalisiert Interrupt an CPU

HW CPU sichert einige Register (PC, IE, ...)

HW CPU disabled Interrupt-Eingang (IE = 0)

HW CPU holt sich von Interrupt-Controller IRQ-Nummer

HW CPU lädt Vektor aus Interrupt-Vektor-Tabelle in PC

SW Interrupt-Handler sichert weitere Register

SW Interrupt-Handler behandelt Interrupt

SW Interrupt-Handler restauriert Register

SW Interrupt-Handler ruft iret auf

HW CPU restauriert PC, IE, ... Rücksprung in das unterbrochene Haupt-Programm, Interrupt-Eingang wieder enabled

☞ Multiprozessorsysteme

– Problem:

- * Unterbrechungen können immer nur von einer CPU behandelt werden. Aber welche?
- * es gibt eine weitere Kategorie von Unterbrechungen: die Interprozessor-Unterbrechungen

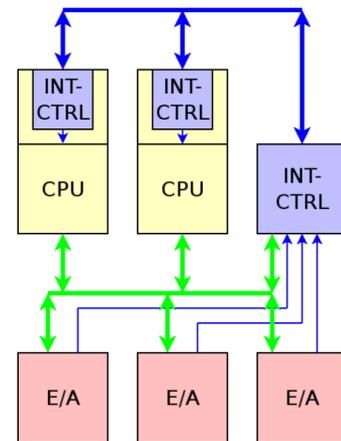
– **Lösung:** die Hardware zur Unterbrechungsbehandlung auf Multiprozessorsystemen muss komplexer ausgelegt sein. Es gibt viele Entwurfsvarianten ...

- * feste Zuordnung
- * zufällige Zuordnung
- * programmierbare Zuordnung

- * Zuordnung unter Berücksichtigung der Prozessorlast

... und Kombinationen davon.

Interrupt-Hardware (SMP)



- Interrupt-Controller sind selbständige, konfigurierbare Einheiten...

☞ Gefahr: „unechte Unterbrechungen“ („spurious interrupts“)

– **Problem:** ein technischer Mechanismus zur Unterbrechungsbehandlung birgt die Gefahr von fehlerhaften Unterbrechungsanforderungen, z.B. durch ...

- * Hardwarefehler
- * fehlerhaft programmierte Geräte

– Lösung:

- * Hardware- und Softwarefehler vermeiden
- * Betriebssystem „defensiv“ programmieren
 - mit unechten Unterbrechungen rechnen

☞ Gefahr: „Unterbrechungstürme“ („interrupt storms“)

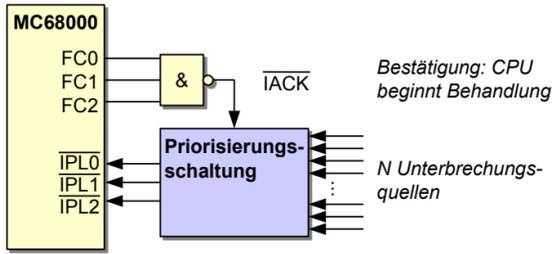
– Problem:

- * hochfrequente Unterbrechungsanforderungen können einen Rechner lahm legen
- * es handelt sich entweder um unechte Unterbrechungen oder der Rechner ist mit der E/A Last überfordert
- * kann leicht mit Seitenflattern (thrashing) verwechselt werden

– Lösung:

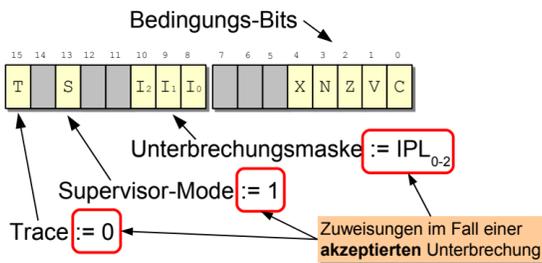
- * Unterbrechungstürme erkennen
- * das verursachende Gerät deaktivieren

2.1.2 Hardware-Architekturen



Motorola/Freescale 68k

- Das Statusregister (SR) des MC68000
 - enthält u.A. die aktuelle Unterbrechungsmaske
 - bei einer Unterbrechung wird geprüft, ob $IPL_{0-2} > I_{0-2}$ ist. Wenn nein, wird der Anforderung (noch) nicht stattgegeben.
 - eine Unterbrechung mit $IPL_{0-2} = 7$ wird aber immer bearbeitet (NMI)



Autovektorielle Unterbrechungen

- über VPA signalisiert die externe Schaltung, dass die CPU die Vektornummer automatisch berechnen soll:

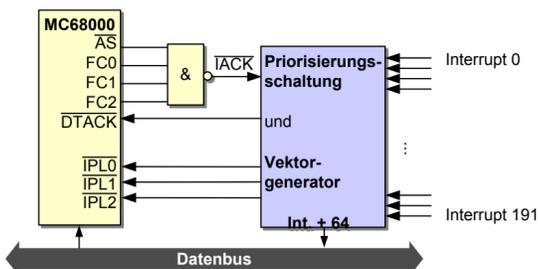
$$Index = 24 + IPL_{0-2}$$

Problem Es stehen nur 6 Vektoren für Geräte bereit. Bei mehr Geräten ist „sharing“ nicht zu vermeiden.

Nicht-autovektorielle Unterbrechungen

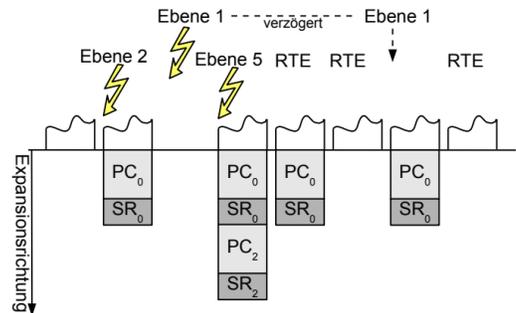
- über DTACK signalisiert die externe Schaltung, dass die CPU die Vektornummer über den Datenbus lesen soll.

$$Index = 64 + D_{0-7}$$



Zustandssicherung beim MC68000

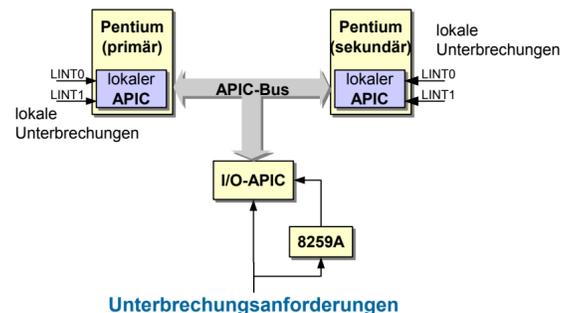
- der vorherige SR Inhalt und der PC werden bei einer Unterbrechung auf dem Supervisor-Stapel gesichert
- der RTE Befehl macht den Vorgang rückgängig



Intel x86

- bis einschließlich i486 hatten x86 CPUs nur einen IRQ und einen NMI Eingang
- externe Hardware sorgte für die Priorisierung und Vektornummergenerierung
 - durch einen Chip namens **PIC 8259A**
 - 8 Interrupt-Eingänge
 - 15 Eingänge bei Kaskadierung von zwei PICs
 - keine Multiprozessorunterstützung
- heutige x86 CPUs enthalten den weit leistungsfähigeren „Advanced Programmable Interrupt Controller“ (APIC)
 - notwendig für **Multiprozessorsysteme**
 - inzwischen aber auch in allen Einprozessorsystemen aktiv
 - natürlich gibt es den PIC 8259A noch immer

Die APIC Architektur



Der I/O APIC

- heute typischerweise in der Southbridge von PC Chipsätzen integriert

- normalerweise 24 Interrupt-Eingänge
 - * zyklische Abfrage (Round-Robin Priorisierung)
- für jeden Eingang gibt es einen 64 Bit Eintrag in der **Interrupt Redirection Table**
 - * beschreibt das Unterbrechungssignal
 - * dient der Generierung der APIC-Bus Nachricht

I/O APIC

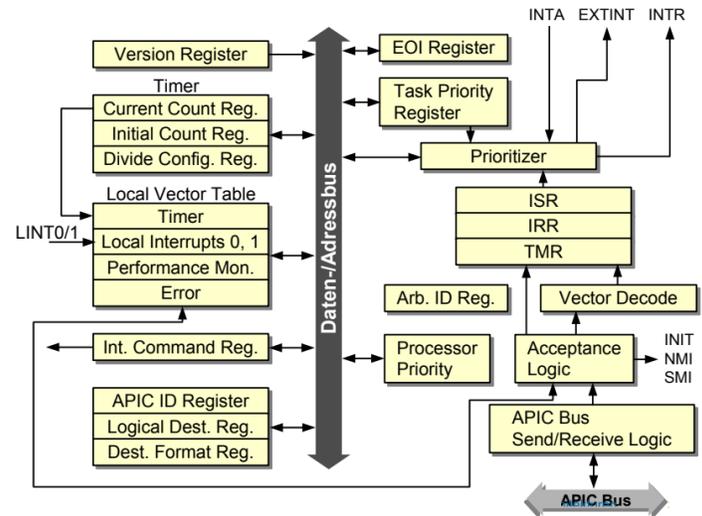
Aufbau (Bits) eines Eintrags in der Interrupt Redirection Table

63:56	Destination Field	- R/W. 8 Bit Zieladresse. je nach Bit 11: APIC ID der CPU (<i>Physical Mode</i>) oder CPU Gruppe (<i>Logical Mode</i>)
55:17	<reserviert>	
16	Interrupt-Mask	- R/W. Unterbrechungssperre.
15	Trigger Mode	- R/W. <i>Edge-</i> oder <i>Level-Triggered</i>
14	Remote IRR	- RO. Art der erhaltenen Bestätigung
13	Interrupt Pin Polarity	- R/W. Signalpolarität
12	Delivery Status	- RO. Interrupt-Nachricht unterwegs?
11	Destination Mode	- R/W. <i>Logical Mode</i> oder <i>Physical Mode</i>
10:8	Delivery Mode	- R/W. Wirkung bei Ziel-APIC
	000 - <i>Fixed</i> :	Signal an alle Zielprozessoren ausliefern
	001 - <i>Lowest Priority</i> :	Liefere an CPU mit aktuell niedrigster Prio.
	010 - <i>SMI</i> :	<i>System Management Interrupt</i>
	100 - <i>NMI</i> :	<i>Non-Maskable Interrupt</i>
	101 - <i>INIT</i> :	Ziel-CPU's initialisieren (Reset)
	111 - <i>ExtINT</i> :	Antwort an PIC 8259A
7:0	Interrupt Vector	- R/W. 8 Bit Vektornummer (16 - 254)

Local APICs

- empfangen Unterbrechungsanforderungen vom APIC Bus
- führen die Auswahl und Priorisierung durch
- können zwei lokale Unterbrechungen direkt verarbeiten
- enthalten weitere Funktionseinheiten
 - * Eingebauten Timer, Performance Counter
 - * Command-Register
 - um selber APIC-Nachrichten zu verschicken
 - insbesondere Inter-Prozessor-Interrupt (IPI)
- programmierbar über 32 Bit Register ab 0xf0000000
 - * memory mapped (ohne externe Buszyklen)
 - * jede CPU programmiert „ihren“ Local APIC

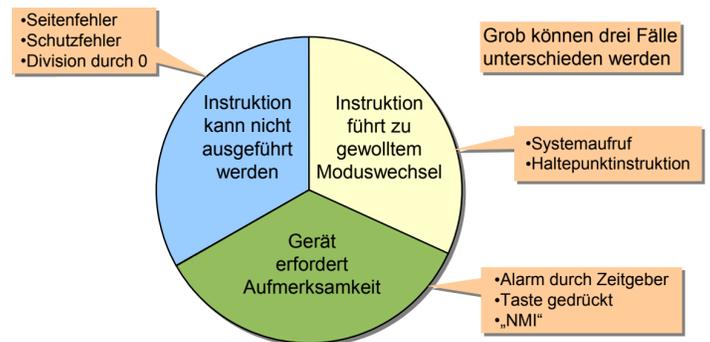
LAPIC Register



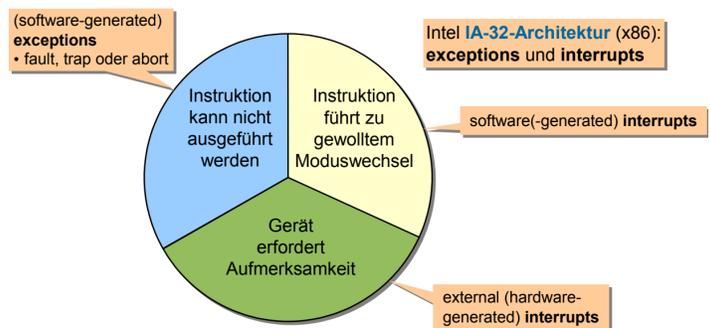
2.2 Software

2.2.1 Begriffe und Grundannahmen

Begriffe



Begriffe: Intel IA-32



Begriffe: Unser Verständnis in BS

- „Trap“



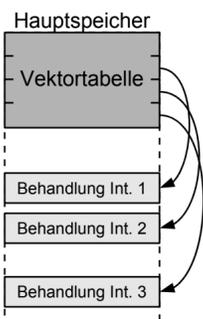
- durch **Instruktion** ausgelöst
 - * auch die „trap“ oder „int“ Instruktion für Systemaufrufe
 - * nicht definiertes Ergebnis (z.B. Division durch 0)
 - * Hardware-Problem (z.B. Busfehler)
 - * Betriebssystem muss eingreifen (z.B. Seitenfehler)
 - * ungültige Instruktion (z.B. bei Programmfehler)
- Eigenschaften
 - * oft vorhersagbar, oft reproduzierbar
 - * Wiederaufnahme **oder Abbruch** der auslösenden Aktivität
- „**Unterbrechung**“ (engl. Interrupt):



- durch **Hardware** ausgelöst
 - * Hardware verlangt die Aufmerksamkeit der Software (Zeitgeber, Tastatursteuereinheit, Festplattensteuereinheit, ...)
- Eigenschaften:
 - * nicht vorhersagbar, nicht reproduzierbar
 - * in der Regel Wiederaufnahme der unterbrochenen Aktivität

Grundannahmen

1. Die **CPU startet** die **Behandlungsroutine** automatisch.
 - erfordert die Zuordnung einer Behandlungsroutine
 - Ermittlung der Unterbrechungsursache nötig



Varianten:

- Register enthält Startadresse der Vektortabelle
- Tabelleneinträge enthalten Code
- Programmierbarer „Event Controller“ behandelt die Unterbrechung in Hardware
- Tabelle enthält Deskriptoren
- Behandlungsroutine hat eigenen Prozesskontext

2. Die Unterbrechungsbehandlung **erfolgt im Systemmodus**.

- Unterbrechungen sind der einzige Mechanismus, um nichtkooperativen Anwendungen die CPU zu entziehen
- nur das BS darf uneingeschränkt auf Geräte zugreifen
- die CPU schaltet daher vor der Unterbrechungsbehandlung in den privilegierten Systemmodus

Varianten:

- bei **16-Bit-CPU**s ist eine Aufteilung in Benutzer-/Systemmodus eher die Ausnahme
- bei **8-Bit-CPU**s (oder kleiner) gibt es diese Aufteilung nicht

3. **Das unterbrochene Programm kann fortgesetzt werden.**

- notwendiger Zustand wird automatisch gesichert
- ggf. auch geschachtelt, erfordert Stapel

4. **Die Maschineninstruktionen verhalten sich atomar.**

- definierter CPU-Zustand zu Beginn der Behandlungsroutine
- Wiederherstellbarkeit des Zustands
- ☞ Trotz der Schwierigkeiten liefern die meisten CPUs „**präzise Unterbrechungen**“

5. **Die Unterbrechungsbehandlung kann unterdrückt werden.**

- Beispiele:
 - Motorola 680x0: entsprechend der Priorität
 - Intel x86: global mit sti, cli (*set/clear Interrupt Flag*)
 - Interrupt Controller: jede Quelle einzeln
- automatische Unterdrückung erfolgt auch durch die CPU vor Betreten der Behandlungsroutine
 - Unterbrechungen nicht vorhersagbar, theoretisch beliebig häufig
 - Stapelüberlauf könnte nicht ausgeschlossen werden
- unterdrückt wird (durch die Hardware) die Behandlung...
 - pauschal aller Unterbrechungen (sehr restriktiv)
 - Unterbrechungen niedriger oder gleicher Priorität (weniger restriktiv)
 - * bestimmte Geräte werden bevorzugt
- bessere Modelle mit Hilfe von Software (z.B. in Linux):
 - Unterbrechungen, die bereits behandelt werden, werden unterdrückt
 - * hohe Reaktivität ohne Bevorzugung einzelner Geräte

2.2.2 Zustandssicherung

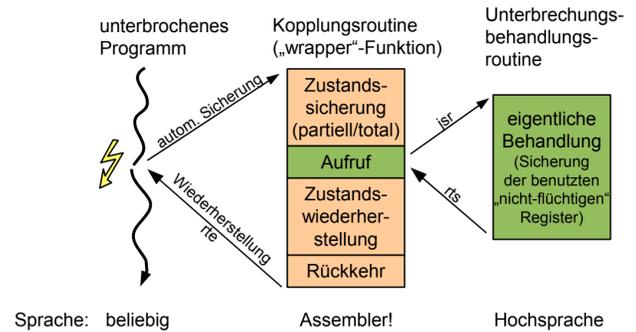
- der Zustand eines Rechners ist enorm groß
 - alle Prozessorregister
 - * Instruktionszeiger, Stapelzeiger, Vielzweckregister, Statusregister, ...
 - der komplette Hauptspeicherinhalt, Caches
 - der Inhalt von E/A-Registern bzw. Ports, Festplatteninhalte, ...
- jeglicher benutzter Zustand, dessen asynchrone Änderung das unterbrochene Programm nicht erwartet, ...
 - darf während der Unterbrechungsbehandlung nicht modifiziert werden
 - muss gesichert und später wiederhergestellt werden
- die CPU sichert (je nach Typ) automatisch ...
 - minimal wenige Bytes (nur Instruktionszeiger und Statusregister)
 - alle Register

Zustandssicherungskonzepte

- **totale Sicherung**
 - die Behandlungsroutine sichert alle Register, die nicht automatisch gesichert wurden
 - **Nachteil:** eventuell wird zu viel gesichert
 - + **Vorteil:** gesicherter Zustand leicht „zugreifbar“
- **partielle Sicherung**
 - die Behandlungsroutine sichert nur die Register, die im weiteren Verlauf geändert werden bzw. nicht gesichert und wieder hergestellt werden
 - machbar, wenn die eigentliche Behandlung in einer Hochsprache wie C oder C++ implementiert ist
 - + **Vorteile:**
 - + nur veränderter Zustand wird auch gesichert
 - + evtl. weniger Instruktionen zum Sichern und Wiederherstellen nötig
 - **Nachteil:** gesicherter Zustand „verstreut“

Übergang auf die Hochsprachenebene

- nicht-portabler Maschinencode sollte minimiert werden
- die eigentliche Unterbrechungsbehandlung erfolgt in einer Hochsprachenfunktion



Zustandsänderungen

- eine Aufteilung, die **der (C/C++) Übersetzer** vornimmt
 - **nicht-flüchtig**
 - * der Übersetzer garantiert, dass der Wert dieser Register über Funktionsaufrufe hinweg erhalten bleibt
 - * ggf. in der aufgerufenen Funktion gesichert und wiederhergestellt
 - **flüchtig** (engl. scratch registers)
 - * wenn die aufrufende Funktion den Wert auch nach dem Aufruf noch benötigt, muss das Register selbst (beim Aufrufer) gesichert werden
 - * normalerweise für Zwischenergebnisse verwendet
- üblicherweise gibt es jedoch einen Standard
 - an den sich alle Übersetzer halten
 - Beispiel x86:
 - * `eax`, `ecx`, `edx` und `eflags` gelten als flüchtig

Wiederherstellung

- die Kopplungsroutine muss alle gesicherten Registerinhalte am Ende wieder laden
 - ... und dann nicht mehr verändern!
- mit einer speziellen Instruktion (z.B. `rte` oder `iret`) wird der vorherige Zustand wiederhergestellt
 - Lesen des automatisch gesicherten Zustands von Supervisor-Stack
 - Setzen des gesicherten Arbeitsmodus (Benutzer-/Systemmodus) und Sprung an die gesicherte Adresse

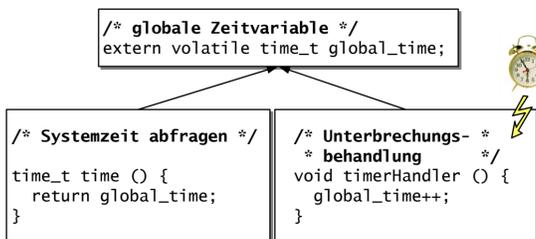
Das BS kann den Zustand auch vor dem `rte/iret` ändern. Dies wird gerne ausgenutzt, um BS-Code im Benutzermodus auszuführen.

2.2.3 Zustandsänderungen

- sind Sinn und Zweck der Unterbrechungsbehandlung
 - Gerätetreiber müssen über den Abschluss einer E/A Operation informiert werden
 - der Scheduler muss erfahren, dass eine Zeitscheibe abgelaufen ist
- müssen mit Vorsicht durchgeführt werden
 - Unterbrechungen können zu jeder Zeit auftreten
 - kritisch sind Daten/Datenstrukturen, die der normale Kontrollfluss und die Unterbrechungsbehandlung sich teilen

Beispiele

- **Beispiel 1: Systemzeit**
 - per Zeitgeberunterbrechung wird die globale Systemzeit inkrementiert
 - * z.B. einmal pro Sekunde
 - mit Hilfe einer Betriebssystemfunktion `time()` kann die Systemzeit abgefragt werden



- hier schlummert möglicherweise ein Fehler ...
 - * das Lesen von `global_time` erfolgt nicht notwendigerweise atomar!

32-Bit-CPU: <code>mov global_time, %eax</code>	16-Bit-CPU (little endian): <code>mov global_time, %r0; lo mov global_time+2, %r1; hi</code>
---	---

- ⚠ Alle 18,2 Stunden kann die Systemzeit (kurz) um etwa die gleiche Zeit vorgehen. **Leider ist das Problem nicht verlässlich reproduzierbar.**

- **Beispiel 2: Ringpuffer**
 - Unterbrechungen wurden eingeführt, damit das System **nicht aktiv** auf Eingaben warten muss
 - * während gerechnet wird, kann die Unterbrechungsbehandlung Eingaben in einem Puffer ablegen
 - ⚠ Wenn die Eingabe nicht schnell genug verarbeitet werden, kann der Puffer voll werden. Die Behandlungsroutine kann die Eingabe dann nicht im Puffer ablegen. In diesem Fall geht die Eingabe verloren.
 - ⚠ auch die Pufferimplementierung ist kritisch ...
 - kann zu inkonsistenten Zuständen führen

Problemanalyse

- selbst einzelne Zuweisungen müssen nicht atomar sein
 - Abhängigkeit vom CPU-Typ, Übersetzer und Codeoptimierung
- Pufferspeicher ist endlich
 - Behandlungsroutine kann nicht warten
 - Daten können verloren gehen
- Pufferdatenstruktur kann kaputt gehen aufgrund von ...
 - inkonsistenten Zwischenzuständen bei Änderungen durch den normalen Kontrollfluss
 - Zustandsänderungen während des Lesens (inkonsistente Kopie!)
 - Änderungen mit Hilfe einer Kopie, die nicht mehr dem Original entspricht
- das Problem ist nicht symmetrisch
 - der normale Kontrollfluss „unterbricht“ nicht die Unterbrechungsbehandlung
 - kann ausgenutzt werden!

„Harte“ Synchronisation Durch Unterdrückung von Unterbrechungen können **race conditions** vermieden werden:

```
char consume() {
    disable_interrupts();
    int elements = occupied;
    if (elements == 0) {
        enable_interrupts();
        return 0;
    }
    char result = buf[nextout];
    nextout++; nextout %= SIZE;
    occupied = elements - 1;
    enable_interrupts();
    return result;
}
```

- Probleme:
 - Gefahr des Verlusts von Unterbrechungsanforderungen
 - hohe und schwer vorherzusagende „**Unterbrechungslatenzen**“

2.3 SoftIRQ

2.3.1 Beispiele

- Beispiel Console
 1. Benutzer tippt Taste.
 2. Keyboard-Controller signalisiert Interrupt.

3. CPU liest Key-Code aus.
⇒ **Hardware-IRQ abgearbeitet!**
4. CPU übersetzt Key-Code in ASCII-Zeichen.
5. ASCII-Zeichen wird im Eingabe-Puffer abgelegt.
6. Wenn „ECHO“-Modus eingeschaltet ist, wird Zeichen auf Bildschirm dargestellt (evntl. Scrollen, Cursor verschieben, Pixel gemäß Zeichensatz setzen/löschen).
7. Die CPU weckt ggf. einen wartenden Prozess auf.

- Hardware-Unterbrechungen

- Probleme:
 - Während der ganzen Unterbrechungslaufzeit sind alle weiteren bzw. alle niederprioren Unterbrechungen gesperrt.
 - * Gefahr von großer Interrupt-Verzögerung
 - * Gefahr von Datenverlusten
 - Unterbrechungsbehandlung kann nicht passiv warten.
 - * Interrupts sind gesperrt

2.3.2 Aufteilung Interrupt-Bearbeitung

- Consolen-Bsp: **Nach Punkt 3 könnten IRQs wieder erlaubt werden!**

☞ Unterbrechungsbehandlung zweigeteilt:

1. Teil, der Zeichen/Pakete/... bei der Hardware ausliest und in einen Puffer kopiert.
 - Interagiert nur minimal mit dem Rest des Systems.
 - Kann (fast) immer ablaufen.
2. Teil, der Zeichen/Pakete/... aus Puffer ausliest und weiterverarbeitet.
 - Ist weitgehend Hardware-unabhängig.
 - Kann (fast) immer unterbrochen werden.

- 1. Teil ggf. leer → Beispiel: Timer/Uhr-Interrupt (*Typisch: alle „Edge-Triggered“-Interrupts*)
- Multiprozessoren:
 - 1. und 2. Teil können auf verschiedenen Kernen ausgeführt werden (Beispiel: E/A-Interrupts; Last-Verteilung).
 - 1. und 2. Teil können auf gleichem Kern ausgeführt werden (Beispiel: Schedule-Interrupt).

```
void keyboard_hard_irq() {
    uint8_t code =
        read_code_from_keyboard_controller();
    if (count < sizeof(buffer)) {
        buffer[head] = code;
        head = (head + 1) % sizeof(buffer);
    }
}
```

```
        count++;
    }
}
void keyboard_soft_irq() {
    asm volatile ("cli\n");
    uint8_t code = buffer[tail];
    tail = (tail + 1) % sizeof(buffer);
    count--;
    asm volatile ("sti\n");
    ...
}
void keyboard_irq() {
    keyboard_hard_irq();
    asm volatile ("sti\n");
    keyboard_soft_irq();
}
```

- Problem: nach **sti** kann Stack überlaufen!

→ Daher (ähnlich Linux):

→ **selbstdefinierte Interrupt-Prioritäten**

```
void keyboard_irq() {
    keyboard_hard_irq();
    apic_disable_keyboard_irq();
    asm volatile ("sti\n");
    keyboard_soft_irq();
    asm volatile ("cli\n");
    apic_enable_keyboard_irq();
}
```

– Nachteile:

- Schreiben in / Lesen aus Puffer kostet Zeit.
- Disable/Enable von Interrupts kostet Zeit.

+ Vorteil:

- + Andere(!), ggf. wichtige(!), Interrupts werden früher bearbeitet.

- Im Falle von „shared interrupts“ sollten

- erst von allen beteiligten Geräten ggf. Daten ausgelesen und zwischengespeichert,
- dann Interrupts wieder zugelassen,
- dann von allen beteiligten Geräten die zwischengespeicherten Daten weiterverarbeitet werden

2.3.3 SoftIRQs

- Hinweis: keyboard_soft_irq muss ggf. gar nicht aufgerufen werden:
- Eine Taste wurde losgelassen.
- Eine Shift/Alt/Ctrl/...-Taste wurde gedrückt.
Daher könnte keyboard_hard_irq-Methode Bit setzen, wenn keyboard_soft_irq-Methode ausgeführt werden soll.

⇒ Entspricht einem Interrupt-Mechanismus
(diesmal in Software)
sogenannte „**SoftIRQs**“

- Typisches SoftIRQ-Interface (pro SoftIRQ):

configure: Registriere Behandlungsfunktion, die beim Auftreten des SoftIRQs aufgerufen werden soll.
trigger: Wenn SoftIRQ disabled: setze Pending-Bit, sonst: führe Behandlungsfunktion aus.
disable: Disable SoftIRQ.
enable: Enable SoftIRQ. Wenn Pending-Bit gesetzt, führe Behandlungsfunktion aus.

⚠ Vorsicht! Race-Condition:

```
void disable() {
    enabled = false;
}
void enable() {
    while (pending) {
        pending = false;
        handler();
    }
    <= trigger hat keinen Effekt!
    enabled = true;
}
void trigger() {
    if (enabled) {
        asm volatile ("sti\n");
        handler();
        asm volatile ("cli\n");
    } else {
        pending = true;
    }
}
```

☞ trigger wird nur im Interrupt-Handler aufgerufen. Kann daher mit cli/sti hart synchronisiert werden:

```
void enable() {
    asm volatile("cli\n");
    while (pending) {
        pending = false;
        asm volatile("sti\n");
        handler();
        asm volatile("cli\n");
    }
    enabled = true;
    asm volatile("sti\n");
}
```

- Idee: *BSD: trigger wird nur im Interrupt-Handler aufgerufen und kennt damit gesicherten Instruktion-Pointer (IP)

→ Prüfen ob Race-Condition vorliegt.

2.3.4 SoftIRQ-Beispiel

```
void keyboard_hard_irq() {
    uint8_t code = read_code_from_keyboard_con
    if (count < sizeof(buffer)) {
        buffer[head] = code;
        head = (head + 1) % sizeof(buffer);
        count++;
    }
    apic_disable_keyboard_irq();
    trigger();
    apic_enable_keyboard_irq();
}
```

```
void keyboard_soft_irq() {
    asm volatile ("cli\n");
    while (0 < count) {
        uint8_t code = buffer[tail];
        tail = (tail + 1) % sizeof(buffer);
        count--;
        asm volatile ("sti\n");
        print_to_screen(code);
        asm volatile ("cli\n");
    }
    asm volatile ("sti\n");
}
```

```
void console_write(char code) {
    disable();
    print_to_screen(code);
    enable();
}
```

2.3.5 Verwandte Konzepte

- **Top/Bottom Halves**

- Interrupt-Handler liest Daten aus Gerät in Puffer und **setzt Bit**
- vor **Rückkehr in den User-Mode** werden Puffer ausgelesen und die Daten verarbeitet, **wenn Bit gesetzt** (Linux)

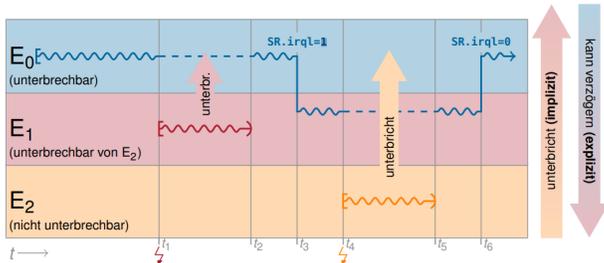
- **ksoftirqd**

- while (pending) { pending = false; handler(); }-Schleife wird nur begrenzte Anzahl von Malen durchlaufen.
- Danach wird „**ksoftirqd**“ (**Kernel-Thread**) aufgeweckt, der SoftIRQ-Abarbeitung übernimmt. Vorteil: Interrupts können CPUs nicht monopolisieren. (Linux)

- **tasklets**

- Hard-Interrupt-Handler triggert nicht (per Nummer) die Ausführung einer bestimmten Methode, sondern fügt eine **beliebige Methode** in eine **Liste** ein. Liste wird dann vor **Rückkehr in User-Mode** abgearbeitet. (Linux)

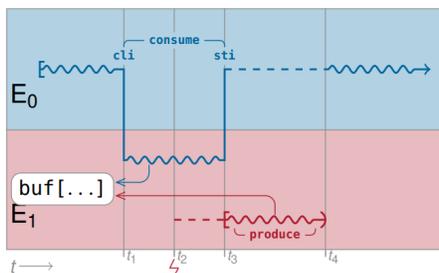
2. **nie unterbrochen** durch Kontrollflüsse von E_k (für $k \leq l$)
 3. **sequentialisiert** mit weiteren Kontrollflüssen von E_l
- Kontrollflüsse können die Ebene **wechseln**
- * durch spezielle Operationen (hier: Modifizieren des Statusregisters)



☞ Konsistenzsicherung

- Jede Zustandsvariable ist (logisch) genau einer Ebene E_l zugeordnet
 - * Zugriffe aus E_l sind implizit konsistent (\leftarrow *Sequentialisierung*)
 - * Konsistenz bei Zugriff aus höheren / tieferen Ebenen muss explizit sichergestellt werden
- Maßnahmen zur Konsistenzsicherung bei Zugriffen:
 - * „von oben“ (aus E_k mit $k < l$) durch **harte Synchronisation**
 - explizit die Ebene auf E_l wechseln beim Zugriff (Verzögerung)
 - damit erfolgt der Zugriff aus derselben Ebene (\leftarrow *Sequentialisierung*)
 - * „von unten“ (aus E_m mit $m > l$) durch **weiche Synchronisation**
 - algorithmisch sicherstellen, dass Unterbrechungen nicht stören
 - erfordert unterbrechungstransparente Algorithmen

2.4.2 Harte Synchronisation



- Zugriff „von oben“ wird hart synchronisiert: Für die Ausführung von `consume()` wechselt der Kontrollfluss auf E_1

+ Vorteile

- + Konsistenz ist sicher gestellt
 - * auch bei komplexen Datenstrukturen und Zugriffsmustern
 - * unabhängig davon, was der Compiler macht
- + einfach anzuwenden, „funktioniert immer“
 - * im Zweifelsfall legt man einfach sämtlichen Zustand auf die höchstprioräre Ebene

– Nachteile

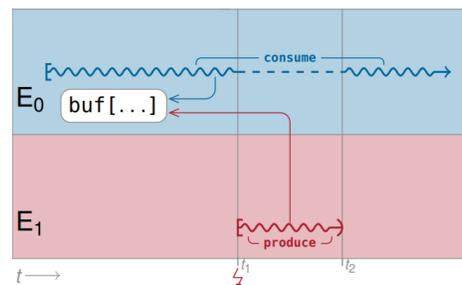
- Breitbandwirkung
 - * Es werden pauschal alle Unterbrechungsbehandlungen (Kontrollflüsse) auf und unterhalb der Zustandsebene verzögert
- Prioritätsverletzung
 - * Es werden Kontrollflüsse höherer Priorität verzögert
- prophylaktisches Verfahren
 - * Nachteile werden in Kauf genommen, obwohl die Wahrscheinlichkeit, dass tatsächlich eine relevante Unterbrechung eintrifft, sehr klein ist.

☞ Bewertung

- Ob die Nachteile erheblich sind, hängt ab von
 - * Häufigkeit,
 - * durchschnittlicher Dauer,
 - * maximaler Dauer der Verzögerung.
- Kritisch ist vor allem die maximale Dauer
 - * hat direkten Einfluss auf die anzunehmende Latenz
 - * Wird die Latenz zu hoch, können Daten verloren gehen
 - edge-triggered Unterbrechungen gehen verloren
 - Daten werden zu langsam von EA-Gerät abgeholt

Fazit Harte Synchronisation ist eher **ungeeignet** für die Konsistenzsicherung **komplexer Datenstrukturen**

2.4.3 Weiche Synchronisation



- Zugriff „von unten“ wird weich synchronisiert: consume() liefert ein korrektes Ergebnis, auch wenn während der Abarbeitung produce() ausgeführt wurde.
- **Konsistenzbedingung**
 - Ergebnis einer unterbrochenen Ausführung soll äquivalent sein zu dem einer sequentiellen Ausführung der Operation
 - * entweder consume() vor produce() oder consume() nach produce()
- **Annahmen**
 - produce() unterbricht consume()
 - * alle anderen Kombinationen kommen nicht vor
 - produce() läuft immer durch (run-to-completion)

+ Vorteile

- + Konsistenz ist sichergestellt (durch Unterbrechungstransparenz)
- + Priorität wird nie verletzt
 - * Kontrollflüsse der höherpriorären Ebenen kommen immer durch
- + Kosten entstehen entweder gar nicht oder nur im Konfliktfall
 - * gar nicht → Beispiel Bounded Buffer
 - * im Konfliktfall → optimistische Verfahren, Beispiel Systemzeit (zusätzliche Kosten durch Wiederaufsetzen)

– Nachteile

- Lösungen häufig sehr komplex
 - * Wenn man überhaupt eine Lösung findet, ist diese in der Regel schwer zu verstehen – und noch schwieriger zu verifizieren
- Lösungen häufig sehr fragil (bezüglich Randbedingungen)
 - * Kleinste Änderungen können die Konsistenzgarantie zerstören
 - * Codegenerierung des Compilers ist zu beachten
- Bei größeren Datenmengen steigen die Wiederaufsetzkosten

☞ Fazit

- Weiche Synchronisation durch Unterbrechungstransparenz ist **grundsätzlich erstrebenswert!**
- Es handelt sich bei den Algorithmen jedoch immer um **Speziallösungen** für **Spezialfälle**.
- Als allgemein verwendbares Mittel für die Sicherung **beliebiger Datenstrukturen** ist sie **nicht geeignet**.

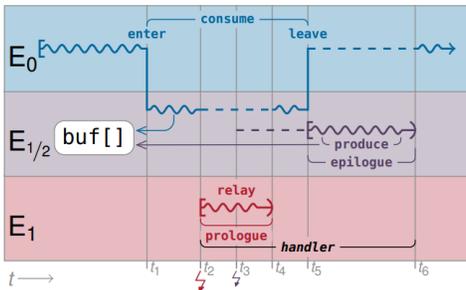
2.4.4 Prolog/Epilog-Modell

- Ansatz: Latenzverbergung durch zusätzliche Ebene
 - Wir fügen eine weitere **logische Ebene** ein: $E_{1/2}$
 - * $E_{1/2}$ liegt zwischen der Anwendungsebene E_0 und den UB-Ebenen $E_{1..n}$
 - Unterbrechungsbehandlung wird **zweigeteilt** in **Prolog** und **Epilog**
 - * **Prolog** arbeitet auf Unterbrechungsebene $E_{1..n}$
 - * **Epilog** arbeitet auf der neuen (Software-)Ebene $E_{1/2}$ (Epilogebeine)
 - Zustand liegt (so weit wie möglich) auf der Epilogebeine
 - * eigentliche Unterbrechungsbehandlung wird nur noch kurz gesperrt
 - Unterbrechungsbehandlungsroutinen werden zweigeteilt
 - * beginnen im Prolog (immer)
 - * werden fortgesetzt im Epilog (bei Bedarf)
 - **Prolog** (→ Hardwareunterbrechung)
 - * läuft auf Hardwareunterbrechungsebene
 - hat damit Priorität über Anwendungsebene und Epilogebeine
 - * ist **kurz**, fasst wenig oder gar keinen Zustand an
 - Üblicherweise wird nur der Hardware-Zustand gesichert und bestätigt
 - Unterbrechungen bleiben nur kurz gesperrt (→ Latenzminimierung)
 - kann bei Bedarf einen Epilog für die weitere Verarbeitung anfordern
 - **Epilog** (→ Softwareunterbrechung)
 - * läuft auf Epilogebeine $E_{1/2}$ (zusätzliche Kontrollflüsseebene)
 - Ausführung erfolgt verzögert zum Prolog
 - erledigt die eigentliche Arbeit (→ Latenzverbergung)
 - * hat Zugriff auf größten Teil des Zustands
 - Zustand wird auf Epilogebeine synchronisiert
- **Epilogebeine**
 - Die Epilogebeine wird (ganz oder teilweise) in **Software** implementiert
 - * trotzdem handelt es sich um eine ganz normale Prioritätsebene des Ebenenmodells
 - * es müssen daher auch dieselben Gesetzmäßigkeiten gelten
 - Es gilt: Kontrollflüsse auf der Epilogebeine $E_{1/2}$ werden

1. **jederzeit unterbrochen** durch Kontrollflüsse der Ebenen $E_{1...n}$
 - * Prologe (Unterbrechungen) haben Priorität über Epiloge
2. **nie unterbrochen** durch Kontrollflüsse der Ebene E_0
 - * Epiloge haben Priorität über Anwendungskontrollflüsse
3. **sequentialisiert** mit anderen Kontrollflüssen von $E_{1/2}$
 - * Anhängige Epiloge werden nacheinander abgearbeitet.
 - * Bei Rückkehr zur Anwendungsebene sind alle Epiloge abgearbeitet.

- Implementierung

- Benötigt werden Operationen, um
 1. explizit die Epiloge Ebene zu betreten: **enter()**
 - * entspricht dem cli bei der harten Synchronisation
 2. explizit die Epiloge Ebene zu verlassen: **leave()**
 - * entspricht dem sti bei der harten Synchronisation
 3. einen Epilog anzufordern: **relay()**
 - * entspricht dem Hochziehen der IRQ-Leitung beim PIC



⚠ Wann müssen anhängige Epiloge abgearbeitet werden?

Immer unmittelbar, bevor die CPU auf E_0 zurückkehrt!

1. bei explizitem Verlassen der Epiloge Ebene mit `leave()`
 - während der Anwendungskontrollfluss auf $E_{1/2}$ gearbeitet hat könnten Epiloge aufgelaufen sein (\leftarrow *Sequentialisierung*).
2. nach Abarbeitung des letzten Epilogs
 - während der Epilogabarbeitung könnten weitere Epiloge aufgelaufen sein (\leftarrow *Sequentialisierung*).
3. wenn der **letzte** Unterbrechungsbehandler terminiert

- während der Abarbeitung von $E_{1...n}$ -Kontrollflüssen könnten Epiloge aufgelaufen sein (\leftarrow *Priorisierung*).

☞ Implementierungsvarianten

- rein softwarebasiert (\rightarrow *Übung*)
- mit Hardwareunterstützung durch einen AST (\rightarrow [2, 3])
- Ein **AST (asynchronous system trap)** ist eine Unterbrechung, die (nur) durch Software angefordert werden kann.
 - z. B. durch Setzen eines Bits in einem bestimmten Register
 - ansonsten technisch vergleichbar mit einer Hardware-Unterbrechung
 - * AST wird (im Gegensatz zu Traps/Exceptions) **asynchron** abgearbeitet
 - * AST läuft auf eigener Unterbrechungsebene zwischen der
 - * Anwendungsebene und den Hardware-UBs (\rightarrow unsere $E_{1/2}$) Gesetzmäßigkeiten des Ebenenmodells gelten (*AST-Ausführung ist verzögerbar, wird automatisch aktiviert, ...*)
- Sicherstellung der Epilogabarbeitung wird damit sehr einfach!
 - Abarbeitung der Epiloge erfolgt im AST
 - \rightarrow und damit automatisch, bevor die CPU auf E_0 zurückkehrt
 - bleibt nur noch die Verwaltung der anhängigen Epiloge
- Beispiel TriCore: Implementierung mit AST
 - AST hier als Unterbrechung der E_1 konfiguriert (\Leftrightarrow unsere $E_{1/2}$)
 - Geräteunterbrechungen laufen auf $E_{2...n}$

```
void enter() {
    CPU::setIRQL(1); // betrete E1, verzögere AST
}
void leave() {
    CPU::setIRQL(0); // erlaube AST (anhängiger
} // AST wurde jetzt abgearbeitet)
void relay(<Epilog>) {
    <haenge Epilog an queue an>
    CPU_SRC1::trigger(); // aktiviere Level-1 IRQ (AST)
}
void __attribute__((interrupt_handler)) irq1Handler() {
    while(<Epilog in queue>) {
        <entferne Epilog aus queue>
        <arbeite Epilog ab>
    }
}
```

- Ziel erreicht?
 - Kernzustand kann jetzt auf Epiloge Ebene verwaltet und synchronisiert werden.

- * Hardware-UBs müssen nicht (mehr) gesperrt werden!
- Ein Problem bleibt noch: Die Epilog-Warteschlange
 - * Zugriff erfolgt aus Prologen und der Epiloge-bene
 - muss also entweder hart synchronisiert werden (im Bild)
 - oder man sucht eine Speziallösung mit weicher Synchronisation
 - ☞ Harte Synchronisation erscheint hier **akzeptabel**, da die Sperrzeit (\Leftrightarrow Ausführungszeit von `dequeue()`) **kurz** und **deterministisch** ist.
 - * Eine Lösung mit **weicher Synchronisation** (z. B. [7]) wäre natürlich schöner!

+ Vorteile

- + Konsistenz ist sichergestellt (durch Synchronisation auf Epiloge-bene)
- + Programmiermodell entspricht dem (einfach verständlichen) Modell der harten Synchronisation
- + Auch komplexer Zustand kann synchronisiert werden
 - * ohne das dabei Unterbrechungsanforderungen verloren gehen
 - * ermöglicht es, den gesamte BS-Kern auf Epiloge-bene zu schützen

- Nachteile

- Zusätzliche Ebene führt zu zusätzlichem Overhead
 - * Epilogaktivierung könnte länger dauern als direkte Behandlung
 - * Komplexität für den BS-Entwickler wird erhöht
- Unterbrechungsperrern lassen sich nicht vollständig vermeiden
 - * Gemeinsamer Zustand von Pro- und Epilog muss weiter hart oder weich synchronisiert werden

• Fazit

- Das Prolog/Epilog-Modell ist ein **guter Kompromiss** für die Synchronisation des Kernzustands.
- Es ist auch für die Konsistenzsicherung **komplexer Datenstrukturen geeignet**

• Verwandte Konzepte

- UNIX: top/bottom half
 - * Aktivitäten der bottom half ($\rightarrow E_1$) sind asynchron zu den Aktivitäten der top half ($\rightarrow E_{1/2}$) und dürfen keine Systemfunktionen aufrufen

- Windows: ISRs / deferred procedure calls (DPCs)
 - * Unterbrechungsbehandler (\rightarrow Prologe) können DPCs (\rightarrow Epiloge) in eine Warteschlange einhängen. Diese wird verzögert abgearbeitet, bevor die CPU auf Faden-Ebene zurückkehrt
- Linux: top halves / bottom halves, tasklets, irq threads
 - * Klassisch: Unterbrechungsbehandler (ISR) setzt Bit, durch das eine verzögerte bottom half (BH \rightarrow Epilog) angefordert werden kann.
 - * Aktuell: BH \rightarrow Softirqs, dazu kommen tasklets (vgl. mit Windows DPCs) und interrupt threads.
- eCos: ISRs / deferred service routines (DSRs)
 - ☞ Nahezu alle Betriebssysteme, die Unterbrechungsbehandlung verwenden, bieten auch eine „Epiloge-bene“.

⚠ Nachtrag: Mehrkernsysteme

- Techniken funktionieren (so) nur bei echter Unterbrechungsemantik: A und UB werden auf **demselben** Prozessor ausgeführt
- Wird die UB „echt parallel“ (auf einem weiteren Prozessor) ausgeführt, kommt es zu Problemen
 - * Annahmen des Prioritätsebenenmodells gelten nicht mehr! (Sequentialisierung, Priorisierung, run-to-completion)
 - * Asymmetrie (UB unterbricht A) ist nicht länger gegeben (weiche Synchronisation wird dadurch viel schwieriger)
- Zusätzlich erforderlich: **Interprozessor-Synchronisation**
 - * „hart“ \rightarrow zweiseitig blockierend, z. B. mit Spin-Locks \rightarrow Übung
 - * „weich“ \rightarrow algorithmisch nichtblockierend (**schwer!**) \rightarrow [CS]

3 IA-32: Die 32-Bit-Intel-Architektur

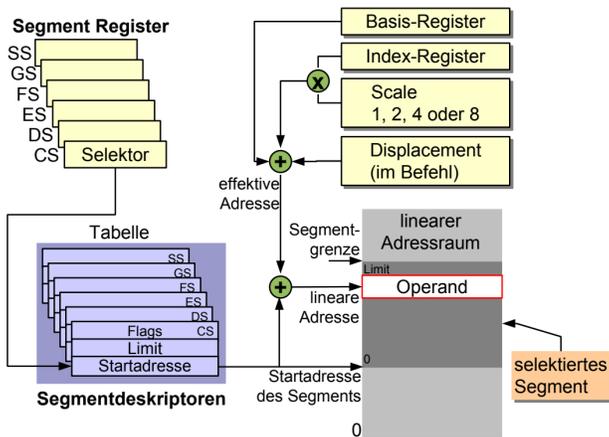
3.1 Urvater: Der 8086

- Programmiermodell
 - 16-Bit Architektur, little-endian
 - 20-Bit Adressbus, d.h. maximal 1 MiB Hauptspeicher
 - wenige Register
 - * (jedenfalls aus heutiger Sicht)
 - 123 Befehle
 - kein orthogonaler Befehlssatz

3.3 Protect Mode

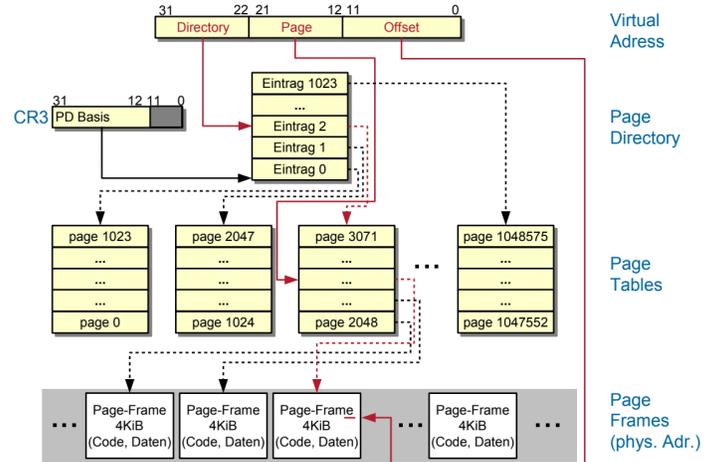
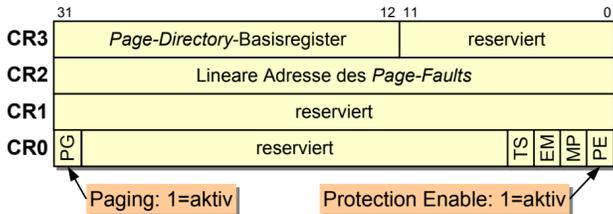
☞ Segmente

- ein Programm (in Ausführung) besteht aus mehreren Speichersegmenten
 - * mindestens CODE, DATEN und STACK
 - * Segmentselektoren beschreiben (indirekt) Adresse und Länge
- „Lineare Adresse“ ist Segmentstartadresse + EA
 - * Segmente dürfen sich im linearen Adressraum überlappen, z.B. dürfen die Segmentstartadressen bei 0 liegen. Dadurch wird ein „flacher“ Adressraum nachgebildet.
 - * „Lineare Adresse“ entspricht der physikalischen Adresse, falls die Paging Unit nicht eingeschaltet ist.



☞ MMU

- Ein- und Auslagerung von Speicher (zwecks virtuellem Speicher) ist bei Segmentierung aufwändig. Daher bieten viele andere CPUs lediglich eine seitenbasierte Speicherverwaltung.
- ab dem 80386 kann eine **Paging Unit (PU)** optional hinzugeschaltet werden.
- die wichtigsten Verwaltungsinformationen stehen in den CRx Steuerregistern:



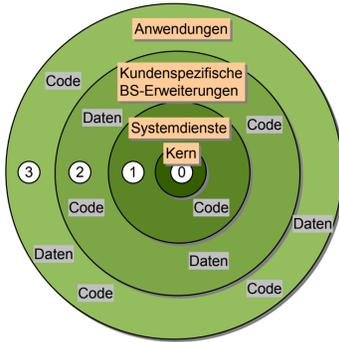
☞ TLB

- Problem: bei aktiver Paging Unit wäre eine IA-32 CPU erheblich langsamer, wenn bei jedem Speicherzugriff das Page Directory und die Page Table gelesen werden müssten
- Lösung: der **Translation Lookaside Buffer (TLB)**:
 - * vollassoziativer Cache
 - Tag: 20 Bit Wert aus Page Directory und Page Table Index
 - Daten: Page Frame Adresse
 - Größe beim 80386: 32 Einträge
 - * bei normalen Anwendungen erreicht der TLB eine Trefferrate von etwa 98%
 - * Schreiben in das CR3 Register invalidiert den TLB

☞ Schutz

- Allgemein
 - * die wichtigste Eigenschaft des IA-32 Protected Mode ist das Schutzkonzept
 - * **Ziel:** fehlerhaften oder nicht vertrauenswürdigen Code isolieren
 - Schutz vor Systemabstürzen
 - Schutz vor unberechtigten Datenzugriffen
 - keine unberechtigten Operationen, z.B. I/O Port Zugriffe
 - * Voraussetzungen: Code und Daten ...
 - werden hinsichtlich der Vertrauenswürdigkeit kategorisiert
 - bekommen einen Besitzer (siehe "Multi-tasking")

☞ Schutzringe und Gates



- * Durch einen 2 Bit Eintrag im Segmentdeskriptor wird jedes Segment einer Privileg-Ebene zugeordnet
 - Code eines inneren Rings kann nur durch "Gates" aufgerufen werden
 - Zugriff auf Daten eines inneren Rings ist verboten
 - der Aufruf von Code eines äußeren Rings ist verboten!

☞ Segmentdeskriptoren

- * weitere Informationen über die Schutzanforderungen der Segmente enthalten die Deskriptoren
 - jede Verletzung führt zum Auslösen einer Ausnahme

⚠ die meisten PC Betriebssysteme nutzen die Segmentierung nicht.

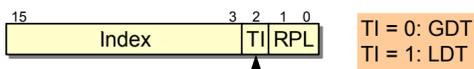
- 32 Bit Offset der logischen Adresse = lineare Adresse
- trotzdem müssen zwei Segmentdeskriptoren angelegt werden

3.4 Multitasking

- neben dem Schutz vor unberechtigten "vertikalen" Zugriffen zwischen Segmenten unterschiedlicher Ebenen unterstützt IA-32 auch ein Task-Konzept ("horizontale Trennung")
- die Zuordnung von Segmenten zu Tasks erfolgt über "Lokale Deskriptortabellen" (LDTs)

☞ Lokale Segmentdeskriptortabellen (LDTs)

- ... sind nötig, wenn der Segmentselektor (z.B. aus einem Segmentregister) sich auf die LDT bezieht:



- ... werden mit Hilfe des LDTR gefunden, das bei jedem Taskwechsel ausgetauscht wird:

Speicherverwaltungsregister				
	15	0 31	0 19	0
TR	TSS-Sel.		TSS-Basisadresse	TSS-Limit
LDTR	LDT-Sel.		LDT-Basisadresse	LDT-Limit
IDTR			IDT-Basisadresse	IDT-Limit
GDTR			GDT-Basisadresse	GDT-Limit

☞ Der Task-Zustand: TSS Segmente

- das Task-Register TR verweist auf eine Datenstruktur, die den kompletten TaskZustand aufnimmt
- bei einem Task-Wechsel wird der komplette Zustand gesichert und der Zustand des ZielTasks geladen
 - * alles in Hardware!

☞ Task-Wechsel

- für einen Task-Wechsel benötigt man entweder ...
 - * ein Task-Gate in der GDT, einer LDT oder der IDT (Task-Wechsel bei Unterbrechungen!)
 - * oder einfach nur einen TSS Deskriptor in der GDT
- ausgelöst werden kann ein Wechsel durch ...
 - * eine JMP Instruktion
 - * eine CALL Instruktion
 - * eine Unterbrechung
 - * eine IRET Instruktion
- Nested Tasks
 - * bei Unterbrechungen und CALLs wird das NT Flag im EFLAGS Register und der "Prev. Task Link" im TSS gesetzt.
 - * Wenn dies der Fall ist, springt IRET zum vorherigen Task zurück.
- Ein-/Ausgaben im Protected Mode
 - * Zugriffe auf Geräte im Speicher (memory-mapped I/O) können über Speicherschutz abgefangen werden
 - * Zugriffe auf I/O Ports werden eingeschränkt:
 - die I/O Privilege Level Bits im EFLAGS Register erlauben Ein- und Ausgaben auf bestimmten Schutzringen
 - auf den anderen Ebenen regelt die I/O Permission Bitmap für jeden Task und Port der Zugriff

3.5 AMD64

- 64-Bit-Erweiterung von x86, propagiert von AMD
 - offizielle Intel-Bezeichnung: x86-64
 - Achtung: IA-64 bezeichnet Intels (tote?) Itanium-Architektur
- Viele IA-32-Features sind weggefallen!

- Segmentierung (außer FS und GS Register)
 - MMU muss benutzt werden
- Task-Modell
- Ring-Modell (außer -1, 0, 3)
- Virtual-X86-Modus

4 Koroutinen und Programmfäden

- C/C++ bietet keine Bordmittel für „verschränkte“ Ausführung
- Wir brauchen Systemunterstützung, um Kontrollflüsse „während der Ausführung“ verlassen und wieder betreten zu können
- Anmerkung: Aus Systemsicht („von unten“) würde der PC reichen!
 - (PC) ↔ minimaler Kontrollflusszustand
 - alles weitere ist letztlich eine Entwurfsentscheidung des Compilers → [UE1]
 - wird in der Praxis jedoch durch Hardwarehersteller nahegelegt (ISA, ABI)

4.1 Grundbegriffe

- Routine: eine endliche Sequenz von Anweisungen
 - z. B. die Funktion f
 - Sprachmittel fast aller Programmiersprachen
 - wird ausgeführt durch **(Routinen-)Kontrollfluss**
- **(Routinen-)Kontrollfluss**: eine Routine in Ausführung
 - Ausführung und Kontrollfluss sind synonyme Begriffe
 - z. B. die Ausführung $\langle f \rangle$ der Funktion f
 - * beginnt bei Aktivierung mit der ersten Anweisung von f
 - Zwischen Routinen und Ausführungen besteht eine Schema-Instanz Relation.
 - * $\langle f \rangle$, $\langle f' \rangle$, $\langle f'' \rangle$ sind Ausführungen von f
 - Routinen-Kontrollflüsse werden erzeugt, gesteuert, und zerstört mit speziellen Elementaroperationen
 - * $\langle f \rangle$ call g (Ausführung $\langle f \rangle$ erreicht Anweisung call g)
 1. erzeugt neue Ausführung $\langle g \rangle$ von g
 2. suspendiert die Ausführung $\langle f \rangle$
 3. aktiviert die Ausführung $\langle g \rangle$ (→ erste Anweisung wird ausgeführt)
 - * $\langle g \rangle$ ret (Ausführung $\langle g \rangle$ erreicht Anweisung ret)

1. zerstört die Ausführung $\langle g \rangle$
2. reaktiviert die Ausführung des Vater-Kontrollflusses (z. B. $\langle f \rangle$)

☞ asymmetrisches Fortsetzungsmodell

- * Routinen-Kontrollflüsse bilden eine **Fortsetzungshierarchie**
 - Vater-Kind Relation zwischen Erzeuger und Erzeugtem
- * Aktivierte Kontrollflüsse werden nach **LIFO** fortgesetzt
 - Der zuletzt aktivierte Kontrollfluss terminiert immer zuerst
 - Vater wird erst fortgesetzt, wenn Kind terminiert
- * Das gilt auch bei Unterbrechungen
 - $\langle f \rangle$ $\not\leftarrow$ irq ist wie call, nur implizit
- * Unterbrechungen können als implizit erzeugte und aktivierte Routinen-Ausführungen verstanden werden

☞ Koroutine

- verallgemeinerte Routine
 - * erlaubt zusätzlich: expliziten Austritt und Wiedereintritt
 - * Sprachmittel einiger Programmiersprachen
 - z. B. Modula-2, Simula-67, Stackless Python
 - * wird ausgeführt durch **Koroutinen-Kontrollfluss**
- **Koroutinen-Kontrollfluss**: eine Koroutine in Ausführung
 - * Kontrollfluss mit eigenem, unabhängigen Zustand
 - mindestens Programmzähler (PC)
 - zusätzlich je nach (zu unterstützendem) **Compiler / ABI / ISA**: weitere Register, Stapel, ...
 - Im Prinzip ein eigenständiger Faden (engl. Thread)
- Koroutinen-Kontrollflüsse werden erzeugt, gesteuert, und zerstört über zusätzliche Elementaroperationen
 - * create g
 1. erzeugt neue Koroutinen-Ausführung $\langle g \rangle$ von g
 - * $\langle f \rangle$ resume $\langle g \rangle$
 1. suspendiert die Koroutinen-Ausführung $\langle f \rangle$
 2. (re-)aktiviert die Koroutinen-Ausführung $\langle g \rangle$
 - * destroy $\langle g \rangle$
 - zerstört die Koroutinen-Ausführung $\langle g \rangle$

- ☞ symmetrisches Fortsetzungsmodell
 - * Koroutinen-Kontrollflüsse bilden eine **Fortsetzungsfolge**
 - Koroutinenzustand bleibt über Ein-/Austritte hinweg erhalten
 - * Alle Koroutinen-Kontrollflüsse sind **gleichberechtigt**
 - kooperatives Multitasking
 - Fortsetzungsreihenfolge ist beliebig

☞ Koroutinen und Programmfäden

- * Koroutinen-Kontrollflüsse werden oft auch bezeichnet als
 - kooperative **Fäden** (engl. cooperative **Threads**)
 - **Fasern** (engl. **Fibers**)
- * Das ist im Prinzip richtig, die Begriffe entstammen jedoch aus verschiedenen Welten
 - Koroutinen-Unterstützung ist historisch (eher) ein **Sprachmerkmal**
 - Mehrfädigkeit ist historisch (eher) ein **Betriebssystemmerkmal**
 - Die Grenzen sind fließend (Sprachfunktion — (Laufzeit-)Bibliothekfunktion — Betriebssystemfunktion)
- * Wir verstehen Koroutinen als **technisches Konzept**
 - um Mehrfädigkeit im BS zu implementieren
 - insbesondere später auch nicht-kooperative Fäden

4.2 Implementierung

- Fortsetzung (engl. Continuation): Rest einer Ausführung
 - Eine Fortsetzung ist ein Objekt, das einen suspendierten Kontrollfluss repräsentiert.
 - * Programmzähler, Register, lokale Variablen, ...
 - * kurz: gesamter Kontrollflusszustand
 - wird benötigt, um den Kontrollfluss zu reaktivieren
- Routinen-Fortsetzungen werden i. a. auf einem Stapel instantiiert
 - in Form von **Stapel-Rahmen**, erzeugt und zerstört durch
 - * **Compiler** (explizit) und **CPU** (implizit) bei **call, ret**
 - * **Kopplungsfunktion** (explizit) und **CPU** (implizit) bei **↵, iret**
 - Der Compiler verwendet dafür i. a. den CPU-Stapel

- * call, ret, push, pop, ... verwenden implizit den CPU-Stapel

☞ Koroutinen → symmetrisches Fortsetzungsmodell

- Ansatz: Koroutinen-Fortsetzungen durch **Routinen-Fortsetzungen** implementieren
 - * Ein resume-Aufruf sieht für den Compiler wie die Erzeugung und Aktivierung eines ganz normalen Routinen-Kontrollflusses aus.
 - * Vor dem ret wird in resume jedoch intern der Koroutinen-Kontrollfluss gewechselt.
- Folge: Technisch gesehen, müssen wir das Routinen-Fortsetzungsmodell **des Compilers** bereitstellen
 - * Registerverwendung → nichtflüchtige Register über Wechsel erhalten
 - * Fortsetzungs-Stapel → eigener Stapel für jede Koroutinen-Instanz

⚠ Problem: nicht-flüchtige Register

- * Der Stapel-Rahmen enthält keine **nicht-flüchtigen Register**, da der Aufrufer davon ausgeht, dass diese nicht verändert werden.
- * Wir springen jedoch in einen **anderen Aufrufer** zurück!
- Implementierung: create
 - **Aufgabe:** Koroutinen-Kontrollfluss < start > erzeugen
 - * Gebraucht wird dafür
 1. **Stapelspeicher** (irgendwo, global) static void* stack_start[256];
 2. **Stapelzeiger** SP sp_start = &stack_start[256];
 3. **Startfunktion** void start(void* param)
 - ...
 4. **Parameter** für die Startfunktion
 - * Koroutinen-Kontrollfluss wird suspendiert erzeugt
 - **Ansatz: create** erzeugt zwei Stapel-Rahmen
 - * so als hätte < start > bereits **resume** als Routine aufgerufen
 1. Rahmen der Startfunktion selber (erzeugt vom „virtuellen Aufrufer“)
 2. Rahmen von **resume** (enthält Fortsetzung in < start >)
 - * erstes **resume** macht „Rücksprung“ an den Beginn von start

5 Scheduling

5.1 Betriebssystemfäden

- Ansatz: Anwendungen „unbemerkt“ als eigenständige Fäden ausführen
 - eine BS-Koroutine pro Anwendung
 - Aktivierung der Anwendung erfolgt durch Aufruf
 - Koroutinenwechsel erfolgt indirekt durch Systemaufruf
- + Vorteile
 - + unabhängige Anwendungsentwicklung
 - + Ablaufplanung (Scheduling) wird zentral implementiert
 - + bei E/A kann eine Anwendung einfach vom BS „blockiert“ und später wieder „geweckt“ werden
 - + zusätzlicher Entzugsmechanismus (preemption mechanism) kann die Monopolisierung der CPU verhindern

5.2 Kooperativer Fadenwechsel

- Die Koroutine **kickoff**
 - wird für jeden Applikationsfaden einmal instantiiert
 - aktiviert die jeweilige Applikation durch einen **up-call**
- **resume** Systemaufrufe
 - der Mechanismus für Anwendungen, um den Prozessor freiwillig abzugeben
 - ggf. verbunden mit einem Moduswechsel der CPU (in diesem Fall wird noch ein Wrapper benötigt)

5.3 Präemptiver Fadenwechsel

- CPU-Entzug durch Zeitgeberunterbrechung
 - die Unterbrechung ist „nur“ ein impliziter Aufruf
 - Behandlungsroutine kann resume aufrufen

⚠ **Achtung:** So geht es normalerweise nicht, denn resume trifft eine **Scheduling-Entscheidung**. Bei den notwendigen Datenstrukturen ist **Unterbrechungssynchronisation** zu beachten!

5.4 Ablaufplanung

5.4.1 Grundbegriffe und Klassifizierung

☞ Scheduler

- trifft strategische Entscheidungen zur Ablaufplanung

- betrachtet wird immer eine Menge lauffähiger Fäden
 - * die Fäden sind allgemein in einer CPU-Warteschlange aufgereiht
 - * die Sortierung erfolgt entsprechend der Scheduling-Strategie
- laufende Faden ist immer von der Entscheidung mit betroffen
 - * dazu muss der laufende Faden jederzeit „greifbar“ sein
 - * vor der Umschaltung ist der laufende Faden zu vermerken
- ein ausgewählter neuer Faden wird dem Dispatcher übergeben

☞ Dispatcher

- setzt die Entscheidungen durch und schaltet Fäden (mit Hilfe von resume) um

Einteilung

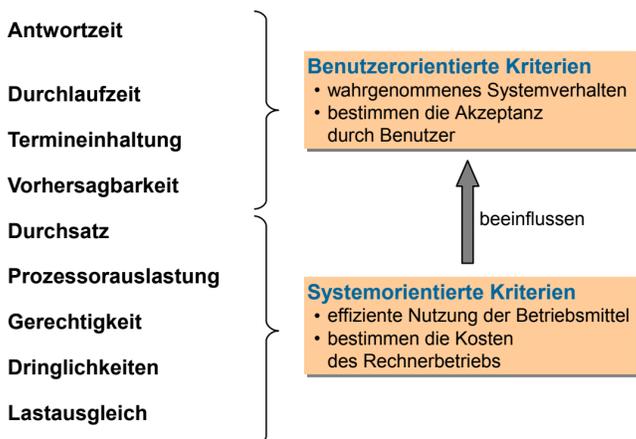
- nach der **Betriebsmittelart** der zeitweilig belegten Hardware-Ressourcen
 - **CPU scheduling** des Betriebsmittels "CPU"
 - * die Prozessanzahl zu einem Zeitpunkt ist höher als die Prozessoranzahl
 - * ein Prozessor ist zwischen mehreren Prozessen zu multiplexen
 - * Prozesse werden dem Prozessor über eine Warteschlange zugeteilt
 - **I/O scheduling** des Betriebsmittels "Gerät", speziell: "Platte"
 - * gerätespezifische Einplanung der von Prozessen abgesetzten E/A-Aufträge
 - * disk scheduling, z.B., berücksichtigt typischerweise drei Faktoren:
 - (1) Positionszeit, (2) Rotationszeit, (3) Transferzeit
 - * Geräteparameter und Gerätezustand bestimmen die nächste E/A-Aktion
 - * die getroffenen Entscheidungen sind ggf. nicht konform zum CPU scheduling
- nach der **Betriebsart** des zu bedienenden/steuernden Rechnersystems
 - **batch scheduling** interaktionsloser bzw. unabhängiger Programme
 - * nicht-verdrängende bzw. verdrängende Verfahren mit langen Zeitscheiben
 - * Minimierung der Kontextwechsellanzahl
 - **interactive scheduling** interaktionsreicher bzw. abhängiger Programme

- * ereignisgesteuerte, verdrängende Verfahren mit kurzen Zeitscheiben
- * Antwortzeitminimierung durch Optimierung der Systemaufrufe
- **real-time scheduling** zeitkritischer bzw. abhängiger Programme
 - * ereignis- oder zeitgesteuerte **deterministische** Verfahren
 - * Garantie der Einhaltung umgebungsbedingter Zeitvorgaben
 - * Rechtzeitigkeit ist entscheidend und nicht Geschwindigkeit
- nach dem **Zeitpunkt** der Erstellung des Ablaufplans
 - **online scheduling** dynamisch, **während** der eigentlichen Ausführung
 - * interaktive und Stapelsysteme, aber auch weiche Echtzeitsysteme
 - **offline scheduling** statisch, **vor** der eigentlichen Ausführung
 - * wenn die Komplexität eine Ablaufplanung im laufenden Betrieb verbietet
 - Einhaltung aller Zeitvorgaben garantieren: ein NP-vollständiges Problem
 - kritisch, wenn auf jede abfangbare katastrophale Situation zu reagieren ist
 - * Ergebnis der Vorbereitung ist ein vollständiger Ablaufplan (in Tabellenform)
 - (semi-) automatisch erstellt per Quelltextanalyse spezieller "Übersetzer"
 - oft zeitgesteuert abgearbeitet/ausgeführt als Teil der Prozessabfertigung
 - * die Verfahren sind zumeist beschränkt auf strikte Echtzeitsysteme
- nach der **Vorhersagbarkeit** von Zeitpunkt und Dauer von Prozessabläufen
 - **deterministic scheduling** bekannter, exakt vorberechneter Prozesse
 - * Prozesslaufzeiten/-termine sind bekannt, sie wurden ggf. "offline" berechnet
 - * die genaue Vorhersage der CPU-Auslastung ist möglich
 - * das System garantiert die Einhaltung der Prozesslaufzeiten/-termine
 - * die Zeitgarantien gelten unabhängig von der jeweiligen Systemlast
 - **probabilistic scheduling** unbekannter Prozesse
 - * Prozesslaufzeiten/-termine bleiben unbestimmt
 - * die (wahrscheinliche) CPU-Auslastung kann lediglich abgeschätzt werden
- * das System kann Zeitgarantien nicht geben und auch nicht einhalten
- * Zeitgarantien sind durch Anwendungsmaßnahmen bedingt erreichbar
- nach dem **Kooperationsverhalten** der (Benutzer/System-) Programme
 - **cooperative scheduling** voneinander abhängiger Prozesse
 - * Prozesse müssen die CPU freiwillig abgeben, zugunsten anderer Prozesse
 - * die Programmausführung muss (direkt/indirekt) Systemaufrufe bewirken
 - * die Systemaufrufe müssen (direkt/indirekt) den Scheduler aktivieren
 - **preemptive scheduling** voneinander unabhängiger Prozesse
 - * Prozessen wird die CPU entzogen, zugunsten anderer Prozesse
 - * **Ereignisse** können die Verdrängung des laufenden Prozesses bewirken
 - * die Ereignisverarbeitung aktiviert (direkt/indirekt) den Scheduler
- nach der **Rechnerarchitektur** des Systems
 - **uni-processor scheduling** in Mehrprogramm,prozesssystemen
 - * die Verarbeitung von Prozessen kann nur pseudo-parallel erfolgen
 - **multi-processor scheduling** in Systemen mit gemeinsamen Speicher
 - * die parallele Verarbeitung von Prozessen wird ermöglicht
 - alle Prozessoren arbeiten eine globale Warteschlange ab
 - jeder Prozessor arbeitet seine lokale Warteschlange ab
- nach der **Ebene der Entscheidungsfindung** bei der Betriebsmittelvergabe
 - **long-term scheduling** [s – min] kontrolliert den Grad an Mehrprogrammbetrieb
 - * Benutzer Systemzugang gewähren, Programme zur Ausführung zulassen
 - * Prozesse dem medium- bzw. short-term scheduling zuführen
 - **medium-term scheduling** [ms – s] als Teil der Ein-/Auslagerungsfunktion
 - * Programme zwischen Vorder- und Hintergrundspeicher hin- und herbewegen
 - * swapping: auslagern (swap-out), einlagern (swap-in)

- **short-term scheduling** [us – ms] regelt die Prozessorzuteilung an die Prozesse
 - * ereignisgesteuerte Ablaufplanung: Unterbrechungen, Systemaufrufe, Signale
 - * Blockierung bzw. Verdrängung des laufenden Prozesses

Scheduling-Kriterien

- **Antwortzeit** Minimierung der Zeitdauer von der Auslösung einer Systemanforderung bis zur Entgegennahme der Rückantwort, bei gleichzeitiger Maximierung der Anzahl interaktiver Prozesse.
- **Durchlaufzeit** Minimierung der Zeitdauer vom Starten eines Prozesses bis zu seiner Beendigung, d.h., der effektiven Prozesslaufzeit und aller Prozesswartezeiten.
- **Termineinhaltung** Starten und/oder Beendigung eines Prozesses zu einem fest vorgegebenen Zeitpunkt.
- **Vorhersagbarkeit** Deterministische Ausführung des Prozesses unabhängig von der jeweils vorliegenden Systemlast.
- **Durchsatz** Maximierung der Anzahl vollendeter Prozesse pro vorgegebener Zeiteinheit. Liefert ein Maß für die geleistete Arbeit im System.
- **Prozessorauslastung** Maximierung des Prozentsatzes der Zeit, während der die CPU Prozesse ausführt, d.h., "sinnvolle" Arbeit leistet.
- **Gerechtigkeit** Gleichbehandlung der auszuführenden Prozesse und Zusicherung, den Prozessen innerhalb gewisser Zeiträume die CPU zuzuteilen.
- **Dringlichkeiten** Bevorzugte Verarbeitung des Prozesses mit der höchsten (statisch/dynamisch zugeordneten) Priorität.
- **Lastausgleich** Gleichmäßige Betriebsmittelauslastung bzw. bevorzugte Verarbeitung der Prozesse, die stark belastete Betriebsmittel eher selten belegen.



Kriterien bei typischen Betriebsarten

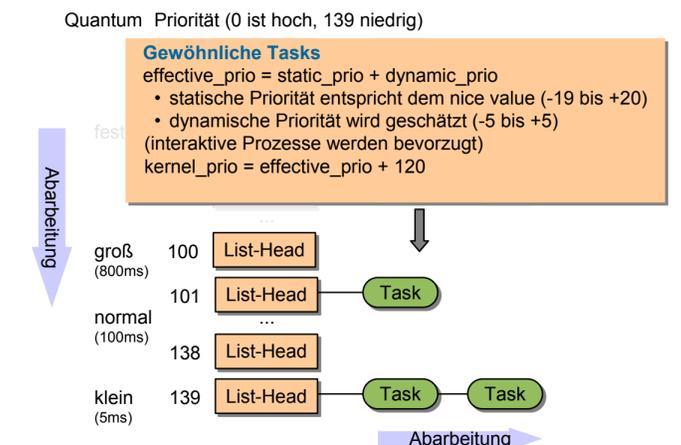
- allgemein (unabhängig von der Betriebsart)
 - Gerechtigkeit
 - Lastausgleich
- **Stapelsysteme**
 - Durchsatz
 - Durchlaufzeit
 - Prozessorauslastung
- **interaktive Systeme**
 - Antwortzeit (Proportionalität – Bearbeitungsdauer entspricht Erwartung)
- **Echtzeitsysteme**
 - Dringlichkeit
 - Termineinhaltung
 - Vorhersagbarkeit

5.4.2 unter Windows

- interactive, probabilistic, online, preemptive, multiprocessor CPU scheduling
- Prioritätenmodell erlaubt feine Zuteilung der Prozesszeit
 - Dynamische Anpassungen beachten
 - Usermode-Threads mit hohen Echtzeitprioritäten haben Vorrang vor allen System-Threads!
 - Executive ist im allgemeinen unterbrechbar
- Interaktive Threads können bevorzugt werden
 - Insbesondere GUI/Multimedia-zentrierte Threads
- Weitere Verbesserungen für SMP in Windows 2003

5.4.3 unter Linux

- **O(1) Scheduler: Multi-Level Queues**



- „interactive, probabilistic, online, preemptive, multiprocessor CPU scheduling“
- Bevorzugung interaktiver Prozesse
 - * schnelle Reaktion auf Eingaben
 - * gleichzeitig Fortschrittgarantie für CPU-lastige Prozesse
- $O(1)$ bei allen Operationen des Scheduler
 - * Einfügen, Entfernen, Scheduling-Entscheidung
- Mehrprozessorunterstützung
 - * Mehrere Bereit-Listen: Parallele Scheduler Ausführung
 - * Keine Idle-Phasen (war ein Problem beim alten Linux Scheduler)
 - * CPU-Lastausgleich

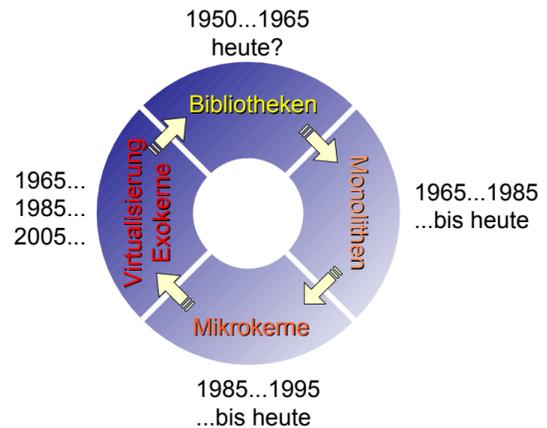
• Completely Fair Scheduler (CFS)

- Ansatz: Ablaufbereite Tasks bekommen die Rechenzeit gleichmäßig ("fair") zugeteilt
 - * bei n Tasks jeweils $1/n$ -tel der CPU-Leistung
 - * hierarchische Zuteilung durch scheduling groups
- CFS läuft nur bei SCHED_NORMAL
 - * Echtzeittask (SCHED_RR und SCHED_FIFO) wie bisher
 - * ansonsten: Task mit geringster CPU-Zeit hat höchste Priorität
- Scheduling-Kriterium ist die bislang zugeteilte CPU-Zeit
 - * Ready-Liste als Rot-Schwarz-Baum, sortiert nach der Zeit
 - * Komplexität $O(\log N)$ (in der Praxis trotzdem effizienter als alter $O(1)$ -Scheduler)
 - * Prioritäten (im Sinne von nice) werden durch schnellere/langsamere Uhren abgebildet
- „interactive, probabilistic, online, preemptive, multiprocessor CPU scheduling“
- **Keine** direkte Bevorzugung interaktiver Prozesse
 - * Fortschrittgarantie für alle Prozesse
 - * Im Vergleich zum $O(1)$ -Scheduler: Besserstellung von Hintergrundprozessen
- $O(\log n)$ bei den meisten Operationen des Scheduler
 - * Einfügen, Scheduling-Entscheidung
- Hierarchie durch scheduling groups
 - * z.B. Fairness zwischen Benutzern: Eine Gruppe pro Benutzer
 - * innerhalb der Gruppe: Fairness zwischen allen Mitgliedern
 - * Benutzer kann weitere Gruppen erstellen, um seinen Anteil aufzuteilen

6 Betriebssystem-Architekturen

- Anwendungsorientierte Kriterien
 - **Portabilität** - Wie unabhängig ist man von der Hardware?
 - **Erweiterbarkeit** - Wie leicht lässt sich das System erweitern (z. B. um neue Gerätetreiber)?
 - **Robustheit** - Wie stark wirken sich Fehler in Einzelteilen auf das Gesamtsystem aus?
 - **Leistung** - Wie gut ist die Hardware durch die Anwendung auslastbar?
- Technische Kriterien (Architektureigenschaften)
 - **Isolationsmechanismus** - Wie werden Anwendungen / BS-Komponenten isoliert?
 - **Interaktionsmechanismus** - Wie kommunizieren Anwendungen / BS-Komponenten miteinander?
 - **Unterbrechungsmechanismus** - Wie werden Unterbrechungen zugestellt und bearbeitet?

6.1 Paradigmen der Betriebssystementwicklung



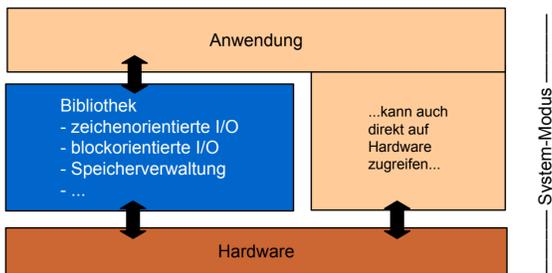
6.1.1 Bibliotheks-Betriebssystemen

- Entstehung
 - Erste Rechnersysteme besaßen keinerlei Systemsoftware
 - * Jedes Programm musste die gesamte Hardware selbst ansteuern
 - * Systeme liefen Operator-gesteuert im Stapelbetrieb
 - single tasking, Lochkarten
 - * Peripherie war vergleichsweise einfach
 - Seriell angesteuerter Lochkartenleser und -schreiber, Drucker, Bandlaufwerk

- Code zur Geräteansteuerung wurde in jedem Anwendungsprogramm repliziert
 - * Die Folge war eine massive Verschwendung von
 - Entwicklungszeit (**teuer!**)
 - Übersetzungszeit (**sehr teuer!**)
 - Speicherplatz (**teuer!**)
 - * außerdem eine hohe Fehleranfälligkeit

☞ Logische Folge: **Bibliotheks-Betriebssysteme**

- * Zusammenfassung von häufig benutzen Funktionen zur Ansteuerung von Geräten in **Software-Bibliotheken** (Libraries)
 - Systemfunktionen als „normale“ Subroutinen
- * Funktionen der Bibliothek waren dokumentiert und getestet
 - verringerte Entwicklungszeit (von Anwendungen)
 - verringerte Übersetzungszeit (von Anwendungen)
- * Bibliotheken konnten resident im Speicher des Rechners bleiben
 - verringerter Speicherbedarf (der Anwendungen)
 - verringerte Ladezeit (der Anwendungen)
- * Fehler konnten von Experten zentral behoben werden
 - verbesserte Zuverlässigkeit



• Bewertung

- Anwendungsorientierte Kriterien
 - * **Portabilität** ← **gering**
 - keine Standards, eigene Bibliotheken für jede Architektur
 - * **Erweiterbarkeit** ← **mäßig**
 - theoretisch gut, in der Praxis oft „Spaghetti-Code“
 - * **Robustheit** ← **sehr hoch**
 - single tasking, Kosten für Programmwechsel sehr hoch
 - * **Leistung** ← **sehr hoch**
 - direktes Operieren auf der Hardware, keine Privilegien

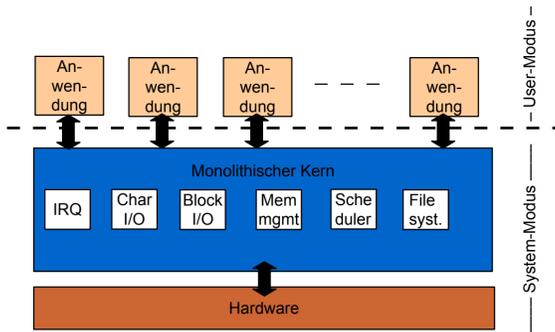
- Technische Kriterien (Architektureigenschaften)
 - * **Isolationsmechanismus** ← **nicht erforderlich**
 - Anwendung ≡ System
 - * **Interaktionsmechanismus** ← **Funktionsaufrufe**
 - Betriebssystem ≡ Bibliothek
 - * **Unterbrechungsmechanismus** ← **oft nicht vorhanden**
 - Kommunikation mit Geräten über polling

⚠ Probleme

- * Teure Hardware wird nicht optimal ausgelastet
 - Hoher Zeitaufwand beim Wechseln der Anwendung
 - Warten auf Ein-/Ausgabe verschwendet unnötig CPU-Zeit
- * Organisatorische Abläufe sehr langwierig
 - Stapelbetrieb, Warteschlangen
 - von der Abgabe eines Programms bis zum Erhalt der Ergebnisse vergehen oft Tage
 - um dann festzustellen, dass das Programm in der ersten Zeile einen Fehler hatte...
- * Keine Interaktivität möglich
 - Betrieb durch Operatoren, kein direkter Zugang zur Hardware
 - Programmabläufe nicht zur Laufzeit parametrierbar

6.1.2 Monolithen

- Motivation: Mehrprogrammbetrieb
- Problem: Isolation
- Ansatz
 - BS als Super-Programm, Kontrollinstanz
 - * Programme laufen unter der Kontrolle des Betriebssystems
 - * Dadurch erstmals (sinnvoll) Mehrprozess-Systeme realisierbar
 - Einführung eines Privilegiensystems
 - * Systemmodus ↔ Anwendungsmodus
 - * Direkter Hardwarezugriff nur im Systemmodus
 - Gerätetreiber gehören zum System
 - Einführung neuer Hard- und Software-Mechanismen
 - * Traps in den Kern
 - * Kontextumschaltung und -sicherung
 - * Scheduling der Betriebsmittel



- Monolithische Systeme: Bell Labs/AT&T UNIX
 - Neu: Portabilität durch Hochsprache
 - * C als domänenspezifische Sprache für Systemsoftware
 - * UNIX wurde mit den Jahren auf nahezu jede Plattform portiert
 - Weitere richtungweisende Konzepte:
 - * alles ist eine Datei, dargestellt als ein Strom von Bytes
 - * komplexe Prozesse werden aus einfachen Programmen komponiert

☞ Bewertung

- Anwendungsorientierte Kriterien
 - * **Portabilität** ← hoch
 - dank „C“ kann und konnte UNIX einfach portiert werden
 - * **Erweiterbarkeit** ← mäßig
 - von Neukompilierung → Modulkonzept
 - * **Robustheit** ← mäßig
 - Anwendungen isoliert, nicht jedoch BS-Komponenten (Treiber!)
 - * **Leistung** ← hoch
 - Nur Betreten / Verlassen des Kerns ist teuer
- Technische Kriterien (Architektureigenschaften)
 - * **Isolationsmechanismus** ← Privilegebenen, Adressräume
 - Pro Anwendung ein Adressraum, Kern läuft auf Systemebene
 - * **Interaktionsmechanismus** ← Funktionsaufrufe, Traps
 - Anwendung → Kern durch Traps, innerhalb des Kerns durch call / ret
 - * **Unterbrechungsmechanismus** ← Bearbeitung im Kern
 - interne Unterteilung in UNIX: bottom half, top half

⚠ Probleme: Betriebssystem-Monolithen

- * Monolithen sind schwer handhabbar

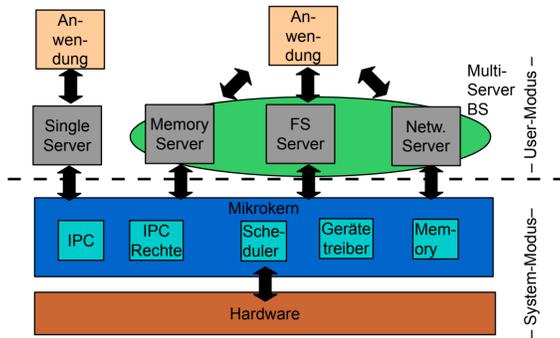
- Hinzufügen oder Abändern von Funktionalität betrifft oft mehr Module, als der Entwickler vorhergesehen hat
- * Eingeschränkte Synchronisationsmechanismen
 - Oft nur ein „Big Kernel Lock“, d. h. nur ein Prozess kann zur selben Zeit im Kernmodus ausgeführt werden, alle anderen warten
 - Insbesondere bei Mehrprozessor-Systemen leistungsreduzierend
- * Gemeinsamer Adressraum aller Kernkomponenten
 - Sicherheitsprobleme in einer Komponente (z.B. buffer overflow) führen zur Kompromittierung des gesamten Systems
 - Viele Komponenten laufen überflüssigerweise im Systemmodus
 - Komplexität und Anzahl von Treibern hat extrem zugenommen

6.1.3 Mikrokerne

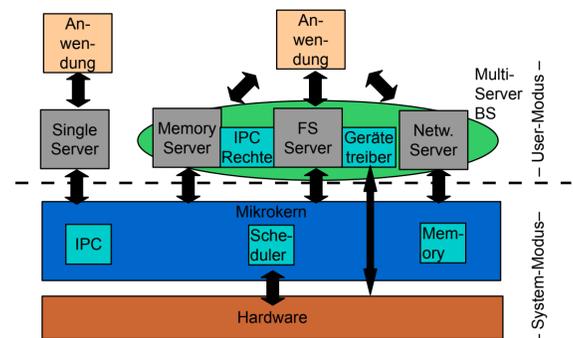
- Motivation

- Reduktion der Trusted Computing Base (TCB)
 - * Minimierung der im privilegierten Modus ablaufenden Funktionalität
 - * BS-Komponenten als Server-Prozesse im nichtprivilegierten Modus
 - * Interaktion über Nachrichten (IPC, Inter Process Communication)
- Prinzip des geringsten Privilegs
 - * Systemkomponenten müssen nur so viele Privilegien besitzen, wie zur Ausführung ihrer Aufgabe erforderlich sind
 - z.B. Treiber: Zugriff auf spezielle IO-Register, nicht auf die gesamte HW
 - * Nur der Mikrokern läuft im Systemmodus
- Geringere Codegröße
 - * L4: 10 kloc C++ ↔ Linux: 1 Mloc C (ohne Treiber)
 - * Ermöglicht Ansätze zur formalen Verifikation des Mikrokerns [6]

Mikrokerne erster Generation



Mikrokerne zweiter Generation



⚠ Probleme

- hoher Overhead für IPC-Operationen
 - * Systemaufrufe Faktor 10 langsamer gegenüber monolithischem Kern
 - Immer noch viel zu große Code-Basis
 - * Gerätetreiber und Rechteverwaltung für IPC im Mikrokern
- die eigentlichen Probleme nicht gelöst!

🔍 Mikrokerne zweiter Generation

- **Ziel:** Mikrokern, diesmal aber richtig!
 - * Verzicht auf Sekundärziele: Portabilität, Netzwerktransparenz, ...
- **Ansatz:** Reduktion auf das Notwendigste
 - * Ein Konzept wird nur dann innerhalb des Mikrokerns toleriert, wenn seine Auslagerung die Implementierung verhindern würde.
 - * synchroner IPC, Kontextwechsel, CPU Scheduler, Adressräume
- Ansatz: Gezielte Beschleunigung
 - * fast IPCs durch Parameterübergabe in Registern
 - * Gezielte Reduktion der Cache-Load (durch sehr kleinen Kern)
- Viele von Mikrokernen der 1. Generation noch im Systemmodus implementierte Funktionalität ausgelagert
 - * z. B. Überprüfung von IPC-Kommunikationsrechten
 - * vor allem aber: Treiber

🔍 Bewertung

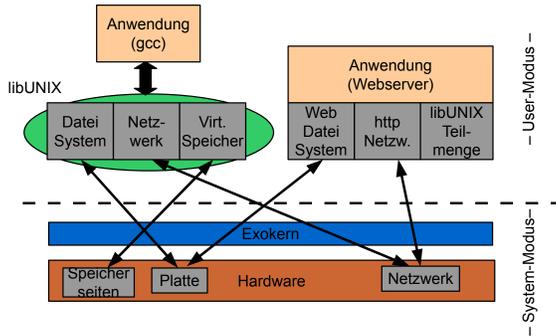
- **Anwendungsorientierte Kriterien**
 - * **Portabilität** ← **mäßig**
 - ursprünglich rein in Assembler, aktuell in C++ entwickelt
 - * **Erweiterbarkeit** ← **sehr hoch**
 - durch neue Server im Benutzermodus, auch zur Laufzeit
 - * **Robustheit** ← **sehr hoch**
 - durch strikte Isolierung
 - * **Leistung** ← **mäßig - gut**
 - IPC-Performance ist **der** kritische Faktor
- **Technische Kriterien (Architektureigenschaften)**
 - * **Isolationsmechanismus** ← **Adressräume**
 - Ein Adressraum pro Anwendung, ein Adressraum pro Systemkomponente
 - * **Interaktionsmechanismus** ← **IPC**
 - Anwendungen und Systemkomponenten interagieren über Nachrichten
 - * **Unterbrechungsmechanismus** ← **IPC an Server-Prozess**
 - Unterbrechungsbehandlung erfolgt durch Faden im Benutzermodus

6.1.4 Exokerne und Virtualisierung

Exokerne

- Motivation
 - Leistungsverbesserung durch Reduktion
 - * Entfernung von Abstraktionsebenen
 - * Implementierung von Strategien (z.B. Scheduling) in der Anwendung
 - Extrem kleiner Kern, dieser implementiert nur
 - * Schutz
 - * Multiplexing von Ressourcen (CPU, Speicher, Disk-Blöcke, ...)

- Trennung von Schutz und Verwaltung der Ressourcen!
- * Keine Implementierung von IPC-Mechanismen (Mikrokern) oder weiterer Abstraktionen (Monolithen)
- * Anwendungen können die für sie idealen Abstraktionen, Komponenten und Strategien verwenden



Bewertung

- **Anwendungsorientierte Kriterien**
 - * **Portabilität** ← sehr hoch
 - Exokerne sind sehr klein
 - * **Erweiterbarkeit** ← sehr hoch
 - aber auch erforderlich! - der Exokern stellt kaum Funktionalität bereit
 - * **Robustheit** ← gut
 - Schutz wird durch den Exokern bereitgestellt
 - * **Leistung** ← sehr gut
 - Anwendungen operieren nahe an der Hardware, wenige Abstraktionsebenen
- **Technische Kriterien (Architektureigenschaften)**
 - * **Isolationsmechanismus** ← **Addressräume**
 - Ein Adressraum pro Anwendung
 - + von ihr gebrauchter Systemkomponenten
 - * **Interaktionsmechanismus** ← **nicht vorgegeben**
 - wird von der Anwendung bestimmt
 - * **Unterbrechungsmechanismus** ← **nicht vorgegeben**
 - Exokern verhindert nur die Monopolisierung der CPU

⚠ Probleme

- * Exokernel sind nicht als Basis für die Verwendung mit beliebigen „legacy“-Anwendungen geeignet
- * Anwendungen haben volle Kontrolle über Abstraktionen

- müssen diese aber auch implementieren
- hohe Anforderungen an Anwendungsentwickler
- * Definition von Exokern-Schnittstellen ist schwierig
 - Bereitstellung adäquater Schnittstellen zur System-Hardware
 - Genaue Abwägung zwischen Mächtigkeit, Minimalismus und ausreichendem Schutz
- * Bisher kein Einsatz in Produktionssystemen
 - Es existieren lediglich einige proof-of-concept-Systeme
 - Viele Fragen der Entwicklung von BS-Bibliotheken noch offen

Virtualisierung

• Motivation

- Ziel: Isolation und Multiplexing **unterhalb** der Systemebene
- Ansatz: Virtual Machine Monitor (VMM) / Hypervisor
 - * Softwarekomponente, läuft direkt auf der Hardware
 - * stellt Abstraktion Virtual Maschine (VM) zur Verfügung
- VM simuliert die gesamten Hardware-Ressourcen
 - * Prozessoren, Speicher, Festplatten, Netzwerkarten, ...
 - * Container für beliebige Betriebssysteme zusammen mit Anwendungen
- Vergleich zu Exokernen
 - * größere Granularität der zugeteilten Ressourcen
 - z.B. gesamte Festplattenpartition vs. einzelne Blöcke
 - * „brute force“ Ansatz
 - Multiplexen ganzer Rechner statt einzelner Betriebsmittel
 - * Anwendungen (und BS) brauchen nicht angepasst werden

• Beispiel VMWare, Xen

- Problem: IA-32 ist eigentlich nicht virtualisierbar
 - * Virtualisierungskriterien von Popek und Goldberg [8] sind nicht erfüllt
 - * Insbesondere: Äquivalenzanforderung - nicht alle Ring 0 Befehle trappen bei Ausführung auf Ring 3
- Ansatz: Paravirtualisierung
 - * „kritische Befehle“ werden ersetzt

- entweder zur Übersetzungszeit (Xen) oder zur Laufzeit (VMWare)
 - * VMs laufen in Ring 3, Ringmodell durch Adressräume nachgebildet
 - Die meisten BS verwenden eh nur Ring 0 und Ring 3
 - Neue IA-32 CPUs unterstützen Virtualisierung in HW (→ VL 6)
 - * Paravirtualisierung in der Praxis oft noch performanter
- Bewertung
 - **Anwendungsorientierte Kriterien**
 - * **Portabilität gering**
 - sehr hardware-spezifisch, Paravirtualisierung ist aufwändig
 - * **Erweiterbarkeit keine**
 - in den üblichen VMMs nicht vorgesehen
 - * **Robustheit gut**
 - grobgranular auf der Ebene von VMs
 - * **Leistung mäßig - gut**
 - stark abhängig vom Einsatzszenario (CPU-lastig, IO-lastig, . . .)
 - **Technische Kriterien** (Architektureigenschaften)
 - * **Isolationsmechanismus VM, Paravirtualisierung**
 - Jede Instanz bekommt einen eigenen Satz an Hardwaregeräten
 - * **Interaktionsmechanismus nicht vorgesehen**
 - Anwendungen in den VMS kommunizieren miteinander über TCP/IP
 - * **Unterbrechungsmechanismus Weiterleitung an VM**
 - VMM simuliert Unterbrechungen in den einzelnen VMs

6.1.5 Fazit

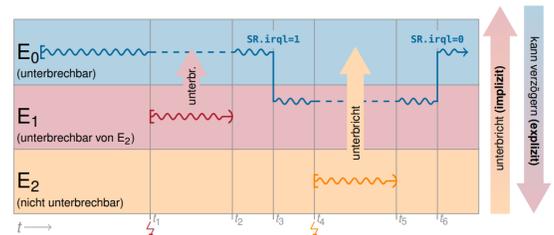
- Betriebssysteme sind ein unendliches Forschungsthema
 - „alte“ Technologien (wie Virtualisierung oder Bibliotheken) finden immer wieder neue Einsatzgebiete
 - Hardwaretechnologie treibt die weitere Entwicklung
- Revolutionäre Neuerungen sind schwer durchzusetzen
 - Kompatibilität ist ein hohes Gut
 - * Auf Anwendungsebene durch Personalities erreichbar
 - * Neue Systeme scheitern jedoch meistens an fehlenden Treibern

- Virtualisierte Hardware als Kompatibilitätsebene
- Die „ideale“ Architektur ist letztlich eine Frage der Anwendung!
 - Sensornetze, tief eingebettete Systeme, Desktoprechner, Server, ...
 - Architektur → nichtfunktionale Eigenschaft des Betriebssystems

7 Fadensynchronisation

- **Bisher:** Konsistenzsicherung bei Zugriffen von Kontrollflüssen aus **verschiedenen** Ebenen
 - Zustand wurde auf einer Ebene „platziert“
 - Sicherung entweder „von oben“ (hart) oder „von unten“ (weich)
 - Innerhalb einer Ebene wurde implizit sequenzialisiert
- **Nun:** Konsistenzsicherung bei Zugriffen von Kontrollflüssen aus **derselben** Ebene
 - Fäden können jederzeit durch andere Fäden verdrängt werden
 - Fäden können echt parallel arbeiten (bei mehreren CPUs)

7.1 Prioritätsebenenmodell mit Fäden



- Kontrollflüsse auf E_l werden
 1. **jederzeit unterbrochen** durch Kontrollflüsse von E_m (für $m > l$)
 2. **nie unterbrochen** durch Kontrollflüsse von E_k (für $k \leq l$)
 3. **sequenzialisiert** mit weiteren Kontrollflüssen von E_l

⚠ Das ist der Knackpunkt!

- Mit der Unterstützung **präemptiver Fäden** können wir **Annahme 3** nicht länger aufrechterhalten:
 - * keine run-to-completion-Semantik mehr
 - * Zugriff auf geteilten Zustand nicht mehr implizit sequenzialisiert

- Dies gilt für alle Ebenen, die Verdrängung (Preemption) oder echte Parallelität von Kontrollflüssen erlauben, also insbesondere die Anwendungsebene E_0 .

☞ Erweitertes Prioritätsebenenmodell - Kontrollflüsse auf E_l werden

1. **jederzeit unterbrochen** durch Kontrollflüsse von E_m (für $m > l$)
2. **nie unterbrochen** durch Kontrollflüsse von E_k (für $k \leq l$)
3. **jederzeit verdrängt** durch Kontrollflüsse von E_l (für $l = 0$)

☞ Kontrollflüsse der E_0 (Fadenebene) sind verdrängbar. Für die Konsistenzsicherung auf dieser Ebene brauchen wir zusätzliche Mechanismen zur Fadensynchronisation.

7.2 Mechanismen

7.2.1 Randbedingungen

- Fäden können **unvorhersehbar** verdrängt werden
 - zu jedem beliebigen Zeitpunkt
 - von beliebigen anderen Fäden höherer, gleicher, oder niedrigerer Priorität (\leftarrow Fortschrittsgarantie)
- Annahmen typisch für Arbeitsplatzrechner
 - probabilistic, interactive, preemptive, online CPU scheduling
 - andere Arten des Scheduling werden im Folgenden nicht betrachtet

☞ Problematisch ist hier die **Fortschrittsgarantie**

Bei rein **prioritätsgesteuertem Scheduling** (Fäden innerhalb einer Prioritätsstufe werden sequentiell abgearbeitet) könnten wir das Ebenenmodell der Unterbrechungsbehandlung einfach auf Fadenprioritäten ausdehnen und vergleichbaren Mechanismen (expliziter Ebenenwechsel, algorithmisch unter der Annahme von run-to-completion) synchronisieren.

- typisch für ereignisgesteuerte Echtzeitsysteme
- in Windows/Linux: Bereich der Echtzeitprioritäten
- bei mehreren Kernen bleibt das Problem der echten Parallelität!

☞ Ziel: aus Anwendersicht **Koordinierung** und **Interaktion**

- Koordinierung des exklusiven Zugriffs auf wiederverwendbare Betriebsmittel (gegenseitiger Ausschluss) \rightarrow **Mutex**

- Interaktion / Koordinierung von konsumierbaren Betriebsmitteln (Synchronisation) \rightarrow **Semaphore**

☞ **Implementierungsansatz:** für den BS-Entwickler Steuerung der CPU-Zuteilung an **Fäden**

- Fäden werden zeitweise von der Zuteilung ausgenommen
- „**Warten**“ als BS-Konzept

7.2.2 Mutex, Implementierungsvarianten

- **Mutex** \rightarrow Kurzform von **mutual exclusion**
 - Ursprung: Bezeichnername eines zweiwertigen Semaphor, eingesetzt für gegenseitigen Ausschluss
 - allgemein: Algorithmus für die Sicherstellung von gegenseitigem Ausschluss in einem kritischen Gebiet
 - hier: Systemabstraktion class Mutex
- Korrektheitsbedingung
 - Es befindet sich maximal ein Faden im kritischen Gebiet

Mutex mit aktivem Warten (\rightarrow spin lock)

- Implementierung rein auf der Benutzerebene
 - markiere Belegung in boolescher Variable (0 \rightarrow frei, 1 \rightarrow belegt)
 - warte in lock() aktiv, bis Variable 0 wird

+ Vorteile

- + Konsistenz ist sichergestellt, Korrektheitsbedingung wird erfüllt
 - * unter der Voraussetzung von Fortschrittsgarantie für alle Fäden
- + Synchronisation erfolgt ohne Beteiligung des Betriebssystems
 - * keine Systemaufrufe erforderlich

- Nachteile

- aktives Warten verschwendet viel CPU-Zeit
 - * mindestens bis die Zeitscheibe abgelaufen ist
 - * bei Zeitscheiben von 10-800 msec ganz erheblich!
 - * Faden wird eventuell vom Scheduler „bestraft“

⚠ **Fazit:**

Aktives Warten ist - wenn überhaupt - nur auf **Multiprozessormaschinen** eine Alternative.

Mutex mit harter Synchronisation

- Implementierung mit „harter Fadensynchronisation“
 - deaktiviere Verdrängbarkeit vor Betreten des kritischen Gebiets
 - * neue Systemoperation: forbid()
 - reaktiviere Verdrängbarkeit nach Verlassen des kritischen Gebiets
 - * neue Systemoperation: permit()
 - ☞ In der Welt der Echtzeitsysteme steht dieses Verfahren hinter dem non-preemptive critical section (NPCS) protocol
 - Implementierung ganz einfach durch Betreten der Epilogebeine
 - Fadenumschaltung ist üblicherweise auf der Epilogebeine angesiedelt
 - * so lange ein Faden auf der Epilogebeine ist kann er nicht verdrängt werden
 - * Voraussetzung: Kontrollflüsse der Epilogebeine werden sequenzialisiert
- **Sequentialisierung auch mit Epilogen!**

+ Vorteile

- + Konsistenz ist sichergestellt, Korrektheitsbedingung wird erfüllt
- + einfach zu implementieren

- Nachteile

- Breitbandwirkung
 - * alle Fäden (und ggfs. sogar Epiloge!) werden pauschal verzögert
- Prioritätsverletzung
 - * „unbeteiligte“ Kontrollflüsse mit höherer Priorität werden verzögert
- prophylaktisches Verfahren
 - * Nachteile werden in Kauf genommen, auch wenn die Wahrscheinlichkeit einer tatsächlichen Kollision sehr klein ist.

⚠ Fazit

Fadensynchronisation auf **Epilogebeine** hat viele Nachteile. Sie ist nur auf Einprozessorsystemen für kurze, selten betretene kritische Gebiete geeignet - oder wenn sowieso mit Epilogen synchronisiert werden muss.

7.2.3 Passives Warten

- Bisherige Mutex-Implementierungen sind nicht ideal
 - Mutex mit aktivem Warten
 - Verschwendung von CPU-Zeit
 - Mutex mit harter Synchronisation

→ grobgranular, prioritätsverletzend

☞ Besserer Ansatz: Faden so lange **von der CPU-Zuteilung ausschließen**, wie der Mutex belegt ist.

- Erfordert neues BS-Konzept: **passives Warten**
 - * Fäden können auf ein Ereignis „passiv warten“
 - passiv warten → von CPU-Zuteilung ausgeschlossen sein
 - (Neuer) Fadenzustand: wartend (auf Ereignis)
 - * Eintreffen des Ereignisses bewirkt Verlassen des Wartezustands
 - Faden wird in CPU-Zuteilung eingeschlossen
 - Anschließender Fadenzustand: bereit
- Implementierung
 - Erforderliche Abstraktionen:
 - * Operationen: block(), wakeup()
 - Betreten bzw. Verlassen des Wartezustands
 - * Warteobjekt: Waitingroom
 - repräsentiert das Ereignis auf das gewartet wird
 - enthält üblicherweise eine Warteschlange der wartenden Fäden
 - ☞ Die Warteschlange sollte sinnvollerweise mit **derselben Priorisierungsstrategie** wie die Bereitliste des Schedulers verwaltet werden!
 - lock() und unlock() bilden ein eigenes kritisches Gebiet
 - müssen auch sequenzialisiert werden!
 - z.B. über epiloge-ebene
 - Fazit
 - Mutex-Zustand liegt nun **im Kern** auf der Epilogebeine
 - * genauer: auf derselben Ebene wie der **Scheduler Zustand**

7.2.4 Semaphore

- Semaphore ist das klassische Synchronisationsobjekt
 - Edgar W. Dijkstra, 1963 [3]
 - In vielen BS: Grundlage für alle Warte-/Synchronisationsobjekte
 - Für uns: Semaphore → Warteobjekt + Zähler
- Zwei Standardoperationen (mit jeweils diversen Namen [2-4])
 - prolaag(), P(), wait(), down(), acquire(), pend()
 - * wenn zähler > 0 vermindere Zähler

- * wenn $\text{zähler} \leq 0$ warte bis $\text{Zähler} > 0$ und probiere es noch einmal
- `verhoog()`, `V()`, `signal()`, `up()`, `release()`, `post()`
 - * erhöhe Zähler
 - * wenn $\text{Zähler} = 1$ wecke gegebenenfalls wartenden Faden
- Es gibt vielfältigste Varianten

Verwendung

- Semantik der Semaphore eignet sich besonders für die Implementierung von Erzeuger/Verbraucher-Szenarien
 - Also für den geordneten Zugriff auf **konsumierbare Betriebsmittel**
 - * Zeichen von der Tastatur
 - * Signale, die auf Fadenebene weiterverarbeitet werden sollen
 - * ...
 - Interner Zähler repräsentiert die Anzahl der Ressourcen
 - * Erzeuger ruft `V()` auf für jedes erzeugte Element.
 - * Verbraucher ruft `P()` auf, um ein Element zu konsumieren
 - wartet gegebenenfalls.

☞ Beachte!

- `P()` kann auf Fadenebene blockieren, `V()` blockiert jedoch nie!
- Als **Erzeuger** kommt daher auch ein Kontrollfluss auf Epilogebe oder Unterbrechungsebene in Frage. (Entsprechende Synchronisation des internen Semaphorzustands vorausgesetzt.)

Semaphore vs. Mutex: Einordnung

- Mutex wird „klassisch“ als binärer Semaphor bezeichnet
 - Mutex → Semaphore mit initialem Zählerwert 1
 - `lock()` → `P()`, `unlock()` → `V()`
- Die Semantik ist (heute) jedoch i. a. deutlich strenger:
 - Ein belegter Mutex hat (implizit oder explizit) einen **Besitzer**.
 - * Nur dieser Besitzer darf `unlock()` aufrufen.
 - * Muteximplementierungen in z. B. Linux oder Windows überprüfen dies.
 - Ein Mutex kann (üblicherweise) auch **rekursiv** belegt werden
 - * Interner Zähler: Derselbe Faden kann mehrfach `lock()` aufrufen; nach der entsprechenden Anzahl von `unlock()`-Aufrufen ist der Mutex frei

- * Eine Semaphore kann hingegen von jedem Faden verändert werden.

☞ Semaphore als Basis aller Dinge?

In vielen BS ist Semaphore die **Grundabstraktion** für Fadensynchronisation. Sie wird deshalb in der Literatur oft als (notwendige) **Implementierungsbasis** für Mutex, Bedingungsvariable, Leser-Schreiber-Sperre etc. angesehen.

7.3 Beispiel: Windows

- Windows treibt die Idee der Warteobjekte sehr weit
 - **Jedes** Kernobjekt ist (auch) ein Synchronisationsobjekt!
 - * explizite Synchronisationsobjekte: Event, Mutex, Timer, Semaphore, . . .
 - * implizite Synchronisationsobjekte: File, Socket, Thread, Prozess, . . .
 - Semantik des Wartens hängt vom Objekt ab
 - * Faden wartet auf „signalisiert“-Zustand
 - * Zustand wird gegebenenfalls durch erfolgreiches Warten geändert

☞ Kosten der Fadensynchronisation

- Synchronisationsobjekte werden im Kern verwaltet
 - * interne Datenstrukturen (Scheduler) → Schutz
 - * interne Synchronisation → Konsistenz
- Das kann ihre Verwendung sehr teuer machen
 - * für jede Zustandsänderung muss in den Kern gewechselt werden
 - * Benutzer-/Kernmodus-Transitionen sind sehr aufwändig
 - * Bei IA32 kommen schnell einige tausend Takte zusammen!
- Bei kurzen kritischen Gebieten mit geringer Wettstreitigkeit (Contention) schlägt dies besonders ins Gewicht
 - * Die benötigte Zeit, um den Mutex zu akquirieren und freizugeben ist oft ein Vielfaches der Zeit, die das kritische Gebiet belegt ist.
 - * Eine tatsächliche Konkurrenzsituation (Faden will in ein bereits belegtes kritisches Gebiet) tritt nur selten auf.
- Ansatz: Mutex soweit wie möglich im **Benutzermodus** verwalten
 - * Minimieren der Kosten im Normalfall
 - Normalfall → kritisches Gebiet ist frei
 - Spezialfall → kritisches Gebiet ist belegt
- Einführen eines fast path für den Normalfall

- * Test, Belegung, und Freigabe erfolgt im Benutzermodus
 - Konsistenz wird algorithmisch / durch atomare CPU-Befehle sichergestellt
- * Warten erfolgt im Kernmodus
 - für den Übergang in den passiven Wartezustand wird der Kern benötigt
- * weitere Optimierung für Multiprozessormaschinen
 - vor dem passiven Warten für begrenzte Zeit aktiv warten → hohe Wahrscheinlichkeit, dass das kritische Gebiet vorher frei wird

Windows: CRITICAL_SECTION

- * Struktur für einen fast mutex im Benutzermodus [8, 9]
 - verwendet intern ein Event (Kernobjekt), falls gewartet werden muss
 - Event wird „lazy“ (erst bei Bedarf) erzeugt

Unter Linux gibt es ab Kernel 2.6 mit Futexes (Fast user-mode mutexes) ein vergleichbares, noch deutlich mächtigeres Konzept.

- schnell arbeiten
- Energie sparen
- Speicher, Ports und Interrupt-Vektoren sparen
- Aktivierung und Deaktivierung zur Laufzeit
- Generische Power Management Schnittstelle
- Einheitlicher Zugriffsmechanismus
 - **minimaler Satz von Operationen** für verschiedene Gerätetypen
 - **mächtige Operationen** für vielfältige Typen von Anwendungen
- auch gerätespezifische Zugriffsfunktionen

Linux

- Geräte sind über Namen im Dateisystem ansprechbar


```
echo „Hallo, Welt“ > /dev/ttyS0
```

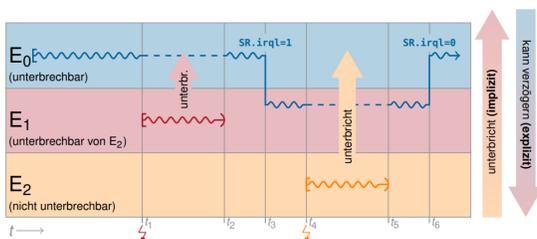
+ Vorteile:

- + Systemaufrufe für Dateizugriff (open, read, write, close) können auch für sonstige E/A verwendet werden
- + Zugriffsrechte können über die Mechanismen des Dateisystems gesteuert werden
- + Anwendungen sehen keinen Unterschied zwischen Dateien und „Gerätedateien“

– Probleme:

- blockorientierte Geräte müssen in Byte-Strom verwandelt werden
- manche Geräte lassen sich nur schwer in dieses Schema pressen
 - * Beispiel: 3D Graphikkarte

8 Gerätetreiber



- Treiberunterstützung ist für die Akzeptanz eines Betriebssystems ein **entscheidender Faktor**
 - warum sonst wäre Linux weiter verbreitet als andere freie UNIXe?
- in Gerätetreibern steckt eine erhebliche Arbeitsleistung
- der Entwurf des E/A Subsystems erfordert viel Geschick
 - möglichst viele wiederverwendbare Funktionen in eine **Treiber-Infrastruktur** verlagern
 - klare Vorgaben bzgl. Treiberstruktur, -verhalten und -schnittstellen, d.h. ein **Treibermodell**

8.1 Anforderungen an das BS

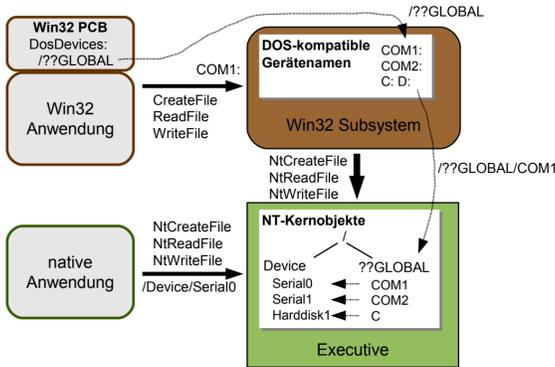
8.1.1 Einheitlicher Zugriff

- Ressourcenschonender Umgang mit Geräten

- blockierende Ein-/Ausgabe (Normalfall)
 - read: Prozess blockiert bis die angeforderten Daten da sind
 - write: Prozess blockiert bis Schreiben möglich ist
- nicht-blockierende Ein-/Ausgabe
 - open/read/write mit dem Zusatz-Flag `O_NONBLOCK`
 - statt zu blockieren kehren read und write so mit `-EAGAIN` zurück
 - der Aufrufer kann/muss die Operation später wiederholen
- nebenläufige Ein-/Ausgabe
 - neu: `aio_(read|write|...)` (POSIX 1003.1-2003)
 - indirekt mittels Kindprozess (fork/join)
 - select, poll Systemaufrufe

Windows

- Geräte sind Kern-Objekte der Executive

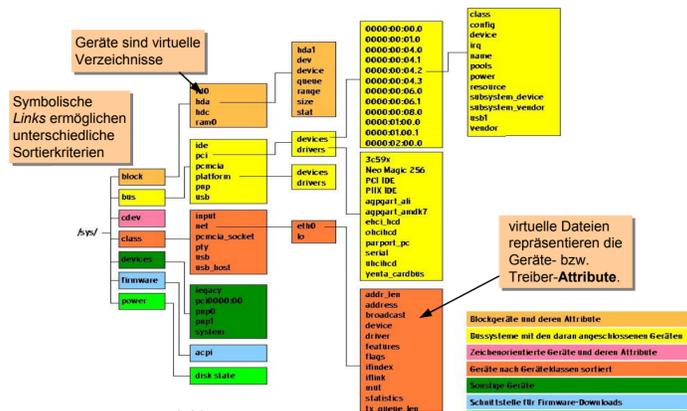


- synchrone oder asynchrone Ein-/Ausgabe (bool flag)
- weitere Möglichkeiten:
 - E/A mit Timeout
 - WaitForMultipleObjects – warten auf 1-N Kern-objekte
 - * Datei-Handles, Semaphore, Mutex, Thread-Handle, ...
 - I/O Completion Ports
 - * Aktivierung eines wartenden Threads nach I/O Operation

8.1.2 Spezifischer Zugriff

Linux

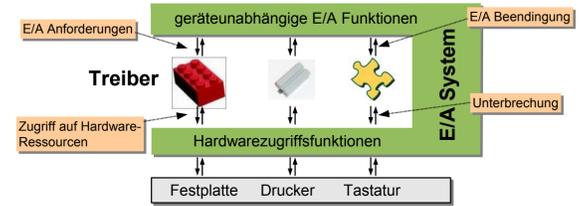
- spezielle Geräteeigenschaften werden (klassisch) über **ioctl** angesprochen
- Schnittstelle generisch und Semantik gerätespezifisch



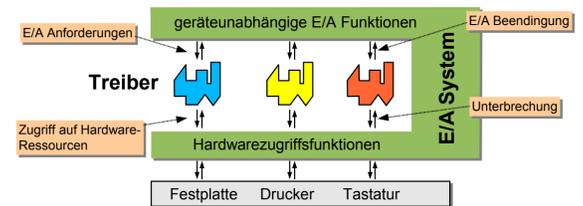
Das Gerätemodell erlaubt Kern- und Anwendungsfunktionen, die Rechnerhardware zu erforschen. Beispielsweise kann eine Power-Management-Funktion abhängige Geräte in der richtigen Reihenfolge stoppen und starten.

- **DeviceIoControl** entspricht in Windows dem UNIX **ioctl**

8.2 Struktur des E/A-Systems



- Treiber mit unterschiedlicher Schnittstelle ...
 - erlauben die volle Ausnutzung aller Geräteeigenschaften
 - erfordern eine Erweiterung des E/A Systems für jeden Treiber
 - * enormer Aufwand bei der heutigen Gerätevielfalt
 - * unrealistisch, da erst das BS und dann die Treiber entstehen



- Treiber mit uniformer Schnittstelle ...
 - ermöglichen ein (dynamisch) erweiterbares E/A System
 - erlauben flexibles „SS Stapeln“ von Gerätetreibern
 - * virtuelle Geräte
 - * Filter
- Das Treibermodell umfasst ...
 - die Liste der erwarteten Treiber-Funktionen
 - Festlegung optionaler und obligatorischer Funktionen
 - die Funktionen, die ein Treiber nutzen darf
 - Interaktionsprotokolle
 - Synchronisationsschema und Funktionen
 - Festlegung von **Treiberklassen** falls mehrere Schnittstellentypen unvermeidbar sind

Anforderungen an Gerätetreiber

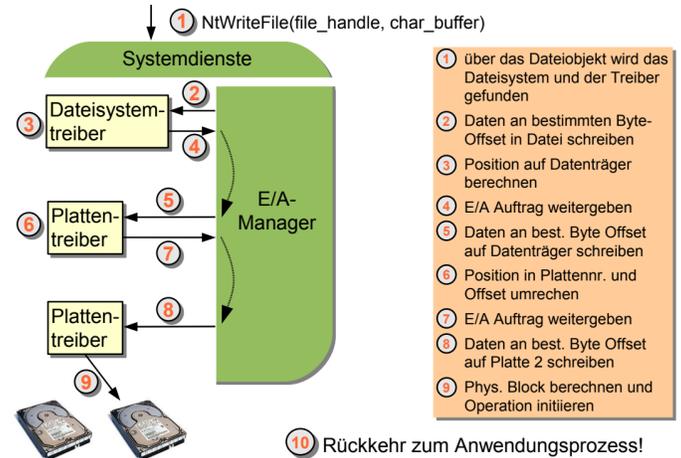
- Zuordnung zu Gerätedateien erlauben
- Verwaltung mehrerer Geräteinstanzen
- Operationen:
 - * Hardware-Erkennung

- * Initialisierung und Beendigung
 - * Lesen und Schreiben von Daten
 - ggf. auch Scatter/Gather
 - * Steueroperationen und Gerätestatus
 - z.B. über ioctl oder virtuelles Dateisystem
 - * Energieverwaltung
- intern zu bewältigen:
- * Synchronisation
 - * Pufferung
 - * Anforderung benötigter Systemressourcen

• Linux - Treiber-Infrastruktur

- Ressourcen reservieren
 - * Speicher, Ports, IRQ-Vektoren, DMA Kanäle
- Hardwarezugriff
 - * Ports und Speicherblöcke lesen und schreiben
- Speicher dynamisch anfordern
- Blockieren und Wecken von Prozessen im Treiber
 - * waitqueue
- Interrupt-Handler anbinden
 - * low-level
 - * Tasklets für länger dauernde Aktivitäten
- Spezielle APIs für verschiedene Treiberklassen
 - * Zeichenorientierte Geräte, Blockgeräte, USB-Geräte, Netzwerktreiber
- Einbindung in das proc oder sys Dateisystem

- * "Verteileroutinen"
 - Öffnen, Schließen, Lesen, Schreiben und gerätespezifische Oper.
- * Interrupt Service Routine
 - wird von der zentralen Interrupt-Verteilungsroutine aufgerufen
- * DPC-Routine
 - "Epilog" der Unterbrechungsbehandlung
- * E/A-Komplettierungs- und -Abbruchroutine
 - Informationen über den Ausgang weitergeleiteter E/A-Aufträge

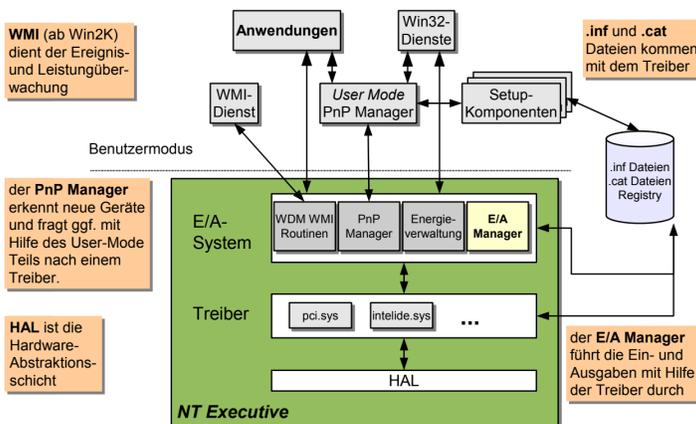


9 Interprozesskommunikation

- Kommunikation und Synchronisation

- ... sind durch das Kausalprinzip immer verbunden:
 - * Wenn **A** eine Information von **B** benötigt, um weiterzuarbeiten, muss **A** solange warten, bis **B** die Information bereitstellt.
- nachrichtenbasierte Kommunikation impliziert Synchronisation (z.B. bei send() und receive())
- Synchronisationsprimitiven eignen sich als Basis für die Implementierung von Kommunikationsprimitiven (z.B. Semaphore)

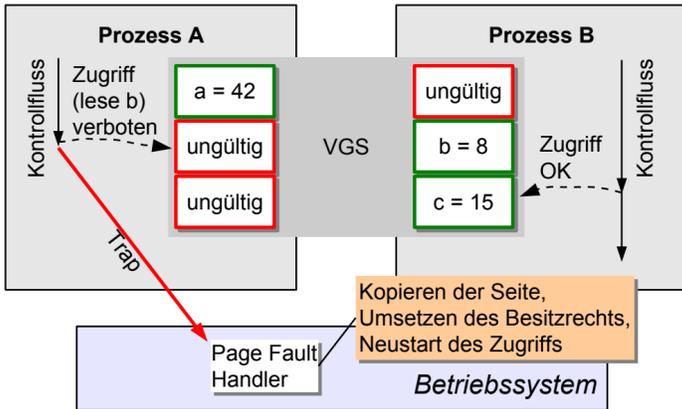
Windows – Treiberstruktur



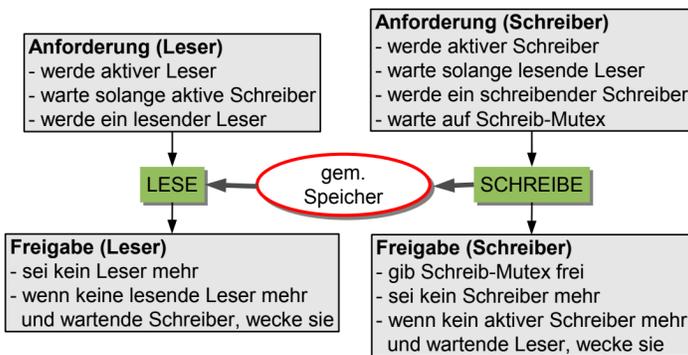
- Das E/A-System steuert den Treiber mit Hilfe der ...
 - * Initialisierungsroutine/Entladeroutine
 - wird nach/vor dem Laden/Entladen des Treibers ausgeführt
 - * Routine zum Hinzufügen von Geräten
 - PnP Manager hat ein neues Gerät für den Treiber

9.1 IPC über Speicher

Virtueller gemeinsamer Speicher



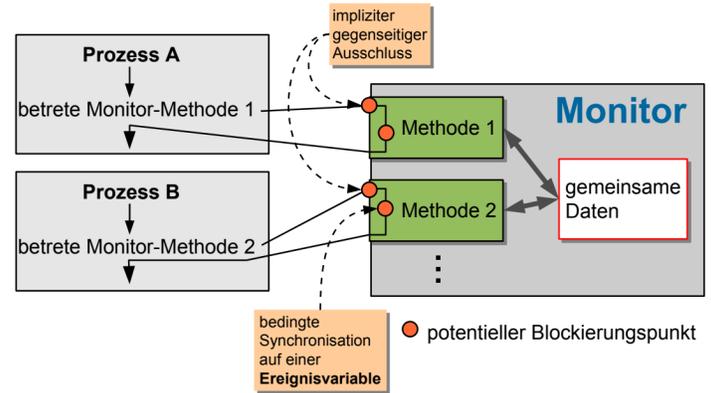
- Semaphore – einfache Interaktionen
 - gegenseitiger Ausschluss
 - einseitige Synchronisation
 - betriebsmittelorientierte Synchronisation
- komplexere Interaktionen
 - Leser/Schreiber-Problem
 - * Schreiber benötigen den Speicher exklusiv
 - * mehrere Leser können gleichzeitig arbeiten



- Erweiterungen
 - * nicht-blockierendes `p()`
 - * Timeout
 - * Felder von Zählern
 - Fehlerquellen
 - * Semaphorbenutzung wird nicht erzwungen
 - * Abhängigkeit kooperierender Prozesse
 - jeder muss die Protokolle exakt einhalten
 - * Aufwand bei der Implementierung
- Unterstützung durch die Programmiersprache
- * Korrekte Synchronisation wird erzwungen

Monitore

- Ansatz: Abstrakte Datentypen werden mit Synchronisationseigenschaften gekoppelt



- Einschränkung der Nebenläufigkeit auf vollständigen gegenseitigen Ausschluss.
 - in Java daher 'synchronized' auch für einzelne Methoden
- Kopplung von logischer Struktur und Synchronisation ist jedoch nicht immer natürlich.
 - siehe Leser/Schreiber Beispiel
 - gleiches Problem wie beim Semaphor: Programmierer müssen ein Protokoll einhalten

→ Die Synchronisation sollte von der Organisation der Daten und Methoden besser getrennt werden.

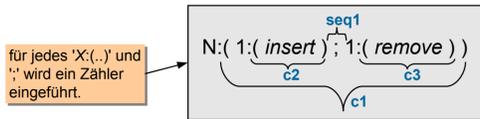
Pfadausdrücke

- Idee: flexible Ausdrücke beschreiben erlaubte Reihenfolgen und den Grad der Nebenläufigkeit.
- **path** name1, name2, name3 **end**
 - bel. Reihenfolge und bel. nebenläufige Ausführung von name1-3
- **path** name1; name2 **end**
 - vor jeder Ausführung von name2 mindestens einmal name1
- **path** name1 + name2 **end**
 - alternative Ausführung: entweder name1 oder name2
- **path** 2:(Pfadausdruck) **end**
 - max. 2 Kontrollflüsse dürfen gleichzeitig im Pfadausdruck sein
- **path** N:(1:(insert); 1:(remove)) **end**
 - z.B. Synchronisation eines N-elementigen Puffers

- * gegenseitiger Ausschluss während insert und remove
- * vor jedem remove muss mindestens ein insert erfolgt sein
- * nie mehr als N abgeschlossene insert-Operationen

- Implementierung

- Transformation in Zustandsautomaten
 - * Zustandsänderung bei Ein-/Austritt in die/aus der Operation
- Beispiel:



- + Vorteile

- + komplexere Interaktionsmuster als mit Monitoren möglich
 - * read + 1: write
- + Einhaltung der Interaktionsprotokolle wird erzwungen
 - * weniger Fehler!

- Nachteile

- Synchronisationsverhalten kann nicht von Zustandsvariablen oder Parametern abhängen
 - * Erweiterung: Pfadausdrücke mit Prädikaten
- Synchronisation des Zustandsautomaten kann Flaschenhals werden
- keine Unterstützung für Pfadausdrücke in gebräuchlichen Programmiersprachen

9.2 IPC über Nachrichten

- Anwendungsfälle/Voraussetzungen
 - IPC über Rechnergrenzen
 - Interaktion isolierter Prozesse
- positive Eigenschaften:
 - einheitliches Paradigma für IPC mit lokalen und entfernten Prozessen
 - ggf. Pufferung und Synchronisation
 - Indirektion erlaubt transparente Protokollerweiterungen
 - * Verschlüsselung, Fehlerkorrektur, ...
 - Hochsprachenmechanismen wie OO-Nachrichten oder Prozeduraufrufe lassen sich gut auf IPC über Nachrichten abbilden (RPC, RMI)

- Bekannt (aus SOS): Variationen von send() und receive()
 - synchron/asynchron (blockierend/nicht blockierend)
 - gepuffert/ungepuffert
 - direkt/indirekt
 - feste Nachrichtengröße/variable Größe
 - symmetrische/asymmetrische Kommunikation
 - mit/ohne Timeout
 - Broadcast/Multicast

9.3 Basisabstraktionen

- Welche IPC Basisabstraktionen bieten Betriebssysteme?
 - UNIX-Systeme: Sockets, System V Semaphore, Messages, Shared Memory
 - Windows NT/2000/XP: Shared Memory, Events, Semaphore, Mutant (Mutex), Sockets, Pipes, Named Pipes, Mailslots, ...
 - Mach: Nachrichten an Ports und Shared Memory (mit Copy on Write)
- Welche Abstraktionen nutzen die Systeme i.d.R. intern?
 - Semaphore erlauben gegenseitigen Ausschluss und einseitige Synchronisation, also sehr häufige Anwendungsfälle
 - * werden praktisch immer benutzt
 - Mikrokerne und verteilte Betriebssysteme: Nachrichten
 - Monolithische Systeme: Semaphore und gemeinsamen Speicher

☞ Dualität – Nachrichten in gemeinsamem Speicher

- auf Basis von Semaphoren und gemeinsamem Speicher lässt sich leicht eine Mailbox-Abstraktion realisieren:

```
class Mailbox : public List {
    Semaphore mutex; // (1)
    Semaphore has_elem; // (0)
public:
    Mailbox() : mutex(1), has_elem(0) {}

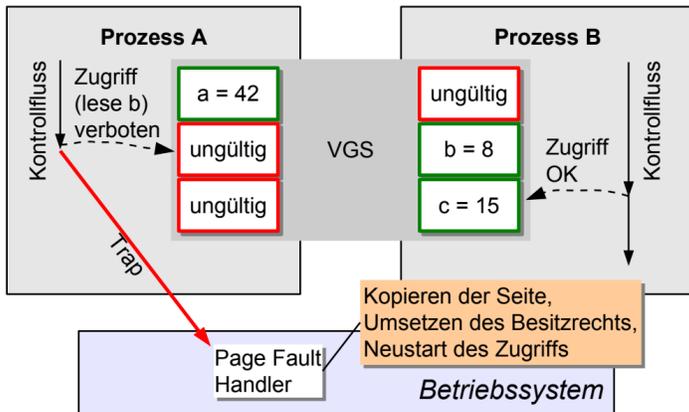
    void send(Message *msg) {
        mutex.p();
        enqueue(msg); // aus List
        mutex.v();
        has_elem.v();
    }
    Message *receive() {
        has_elem.p();
```

```

mutex.p();
Message *result = dequeue (); // List
mutex.v();
return result;
}
};

```

- Nachrichten werden nicht kopiert
 - * Sender sorgt für Speicher
- receive blockiert ggf.
- Mailbox-Abstraktion erlaubt M:N IPC



• VGS Diskussion

- Verteilter virtueller gemeinsamer Speicher ermöglicht...
 - * das Programmiermodell von Multiprozessoren auf Mehrrechnersystemen zu nutzen
 - * IPC über (virtuellen) gemeinsamen Speicher trotz getrennter Adressräume
- Probleme:
 - * Latenzen der Kommunikation und Trap-Behandlung
 - * „false sharing“ - Seitengröße entspricht nicht Objektgröße
- Lösungsansätze:
 - * schwache Konsistenzmodelle, z.B.:
 - nicht jeder Zugriff führt zu einem Trap, veraltete Werte werden in Kauf genommen
 - * Änderungen asynchron per Broad-/Multicast verbreiten

☞ Dualität – Aktive Objekte

```

class Server : public ActiveObject {
    Msg msg; // Nachrichtenpuffer
public:
    ...
    // Objekt mit Kontrollfluss!
    void action() {
        while (true) {

```

```

receive(ANY, msg); // empfangen Nachricht
switch (msg.type()) {
    case DO_THIS: doThis(); break;
    case DO_THAT: doThat(); break;
    default: handleError();
}
reply(msg);
}
}

```

```

}
}
void client1() {
    Message msg(DO_THIS);
    send(srv, msg);
}
void client2() {
    Message msg(DO_THAT);
    send(srv, msg);
}
}

```

- Objekte mit Kontrollfluss
- gut geeignet zur Zugriffssynchronisation in Systemen mit nachrichtenbasierter IPC
- ☞ Gegenseitiger Ausschluss durch die Verarbeitungsschleife wird garantiert. Durch das synchrone send() blockiert ein Client solange der Server noch beschäftigt ist. → genau wie ein Monitor

☞ Dualität – Diskussion

- Gibt es einen fundamentalen Unterschied zwischen IPC über gem. Speicher und IPC über Nachrichten?
 - * zugespitzt: sind oder prozedurorientierte BS (Monolithen) oder prozessorientierte BS (Mikrokerne) besser?
- Beispiel: Leser/Schreiber Monitor vs. Server:
 - * Monitor: 2 potentielle Wartepunkte
 - Client wird verzögert für gegenseitigen Ausschluss.
 - Client wird ggf. wegen einer Ereignisvariablen weiter verzögert.
 - * Server: 2 potentielle Wartepunkte
 - Reply wird verzögert, da der Server noch andere Requests bearbeitet.
 - Reply wird ggf. weiter verzögert, wenn der Request in eine Warteschlange gehängt werden muss.
- Fazit: Dualität in Synchronisation und Nebenläufigkeit [4]

9.4 Trennung der Belange mit AOP

- „Aspektorientierte Programmierung“ erlaubt die modulare Implementierung „querschneidender“ Belange
- Beispiel in AspectC++:

```

// Festlegung der Monitore des Systems
pointcut monitors() = "FileTable"||"BufferCache";

// Synchronisation per Aspekt
aspect MonitorSynch {
  advice monitors() : slice struct {
    Semaphore _mutex;
  };
  advice construction(monitors()) : before() {
    tjp->that()->_mutex.init(1);
  }
  advice execution(monitors()) : around() {
    tjp->that()->_mutex.p(); // Monitor sperren
    tjp->proceed(); // Fkt. ausführen
    tjp->that()->_mutex.v(); // Monitor freigeben
  }
};

```

9.5 Fazit

- Mechanismen beider Klassen sind in realen Betriebssystemen anzutreffen
 - Sprachmechanismen wie Monitore und Pfadausdrücke können bei der BS-Entwicklung allerdings i.d.R. nicht verwendet werden
- Bzgl. des Synchronisationsverhaltens und dem Grad der Nebenläufigkeit zeichnet sich keine Klasse besonders aus
 - Vor- und Nachteile liegen woanders
 - Ausblick: mit AOP Techniken könnte man von den konkreten Kommunikations- und Synchronisationsmechanismen abstrahieren