

Betriebssysteme

Silas Hoffmann, inf103088

November 25, 2019

Contents

1	Betriebssysteme	2
1.1	Einführung	2
2	Generelles	72

1 Betriebssysteme

1.1 Einführung

Richtig oder falsch: Ältere Betriebssysteme wie MS-DOS benutzten Basisfunktionen des BIOS auch nach dem Laden des eigentlichen BS.

Richtig. Neuere Varianten tun dies nicht mehr, sodass man gar den Chip des BIOS während des laufenden Betriebs entfernen kann.

Entwicklung des BIOS



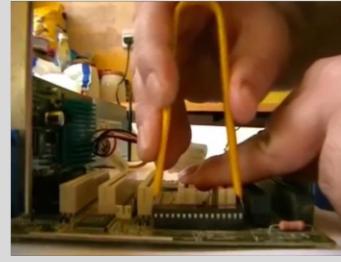
Früher vs. Heute

- Ältere Betriebssysteme wie MS-DOS benutzten Basisfunktionen des BIOS auch nach Laden des eigentlichen BS
- z.B. für Ausgabe von Text, Verarbeitung von Tastatureingaben, Zugriff auf Festplatte

"MS-DOS (PC DOS), which was the dominant PC operating system from the early 1980s until the mid 1990s, relied on BIOS services for disk, keyboard, and text display functions."

"MS Windows NT, Linux, and other protected mode operating systems in general ignore the abstraction layer provided by the BIOS and do not use it after loading, instead accessing the hardware components directly."

Entfernen des BIOS Chip nach Start des Betriebssystems



Quelle: <https://www.youtube.com/watch?v=3Rz8qutSsvY>

Erläutern Sie den Begriff des **UEFI** wofür steht diese Abkürzung, woher kommt es und welche Vorteile bietet es im Gegensatz zum BIOS? Erläutern Sie ebenfalls kurz was man unter dem Begriff *Firmware* versteht. Welche Nachteile bietet das Booten mittels UEFI?

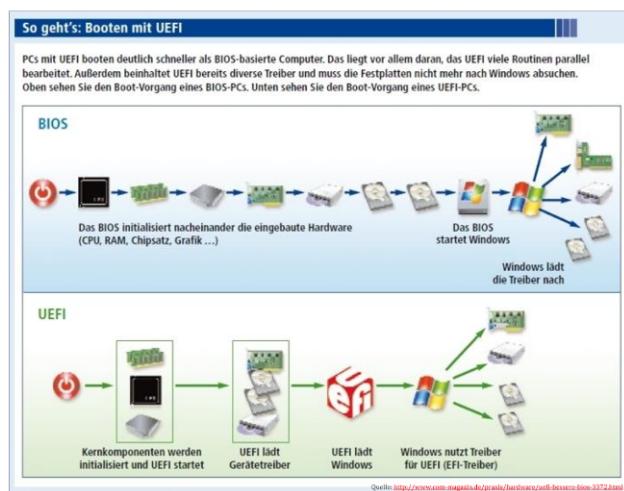
Unified Extensible Firmware Interface (UEFI)



Woher kommt es?	Vorteile
Unified Extensible Firmware Interface <i>Vereinheitlichte Erweiterbare Firmware*-Schnittstelle</i>  <ul style="list-style-type: none"> ▪ als Ersatz/Nachfolger des BIOS gedacht ▪ geht zurück auf eine Initiative von Intel, um Beschränkungen des alten BIOS zu überwinden <p>*Firmware = Software, die in Geräte eingebettet ist</p>	<ul style="list-style-type: none"> ▪ Unterstützung von Laufwerken mit mehr als 2.2 TB → erlaubt also z.B. Booten auch von einer 3TB Platte ▪ Schnelleres Booten <ul style="list-style-type: none"> ▪ Paralleles Laden der Gerätetreiber ▪ UEFI Treiber werden vom BS wiederverwendet ▪ Netzwerkzugriff schon zur Bootzeit → z.B. Suche nach neuen UEFI-BIOS Updates <small>↳ Nachteil: Hackeraufgriffe möglich</small> ▪ UEFI-Shell: Eigene Kommandozeile ▪ Bootmanager zur Auswahl des zu startenden BS ist schon integriert

Skizzieren Sie den **Bootvorgang** mittels UEFI und grenzen Sie diesen von dem mittels des klassischen BIOS ab.

Bootvorgang mit UEFI



Erläutern Sie die BS *Dienstleistung* der **Abstraktion** von konkreter Hardware.
Wieso ist so etwas sinnvoll und gibt es hierbei Ausnahmen?

2. Dienstleistung des Betriebssystem



Abstraktion von der konkreten Hardware

Ob Webcam, Maus, Drucker, Grafikkarte, etc. von Firma X oder Y ist aus Sicht der Programme egal



Somit: Gerätetreiber = Teil des Betriebssystems

Bsp: MS-DOS & fehlende Abstraktionsschicht für ...

... Sounderzeugung

Soundkarte bei MS-DOS = Teil der Hardware, die direkt von jedem Programm angesprochen werden musste



z.B. Soundblaster Karte 2.0 (1991)

Welche Dienstleistungen stellt ein Betriebssystem im Bereich der **Speichermedien** bereit?

3. Dienstleistung des Betriebssystem



Dateisysteme als Abstraktionsschicht vom Speichermedium

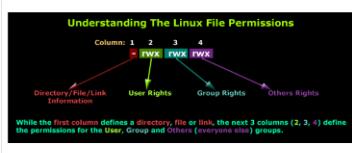
- Dateisysteme: FAT32, NTFS, ReFS (Resilient File System → Win8), ext2, ext3, ext4, ReiserFS, Reiser4, Tux3, etc.
- Egal ob Festplatte, CD, DVD, Blu-Ray, USB-Stick, SD-Card, Netzlaufwerk, etc. die Zugriffswise ist immer gleich



Bildquellen:
USB-Stick: <http://ecx.images-amazon.com/images/I/31V5cpn%2B0L3P266166LT.jpg>
CD: <http://www.wikiwand.com/wi/CD-ROM>
Blu-ray: <http://de.wikipedia.org/wiki/Blu-ray>

Dateisystem-Rechteverwaltung

Wer darf auf Datei X wie zugreifen?



Quelle: http://www.linuxlinks.co/magnuscarsten/lxweb/intro/intro_to_file_permissions.html

Welche Dienstleistungen stellt ein BS im Bereich der **Prozessverwaltung** bereit? Erläutern Sie ebenfalls diesen Begriff.

4. Dienstleistung des Betriebssystem



Prozessverwaltung

- Starten/Stoppen von Prozessen („Programme“)
- Hilfe bei Vermeidung möglicher Konflikte parallel arbeitenden Threads („Programmfäden“)
- Zuordnung: Rechenkern / Rechenzeit
- Zuteilung von Speicher und anderer beschränkter Betriebsmittel (z.B. gleichzeitiger Zugriff mehrerer Prozesse auf die gleiche Festplatte)
- Überwachen von Speicherzugriffsverletzungen

Beispiel: Speicherzugriffsverletzung

```
#include <iostream>

int main()
{
    int* a = new int[5];
    // a[0] = 2*0
    // a[1] = 2*1
    // a[2] = 2*2
    // a[3] = 2*3
    // a[4] = 2*4

    for (int i=0; i<50; i++)
    {
        a[i] = 2*i;
        std::cout << a[i] << std::endl;
    }
    return 0;
}
```

Access Violation Error

Prozess- und Betriebsmittelverwaltung

Abstraktion von der Hardware
Benutzeroberfläche Dateisysteme



Zwischenfazit: Betriebssystem als Dienstleister!

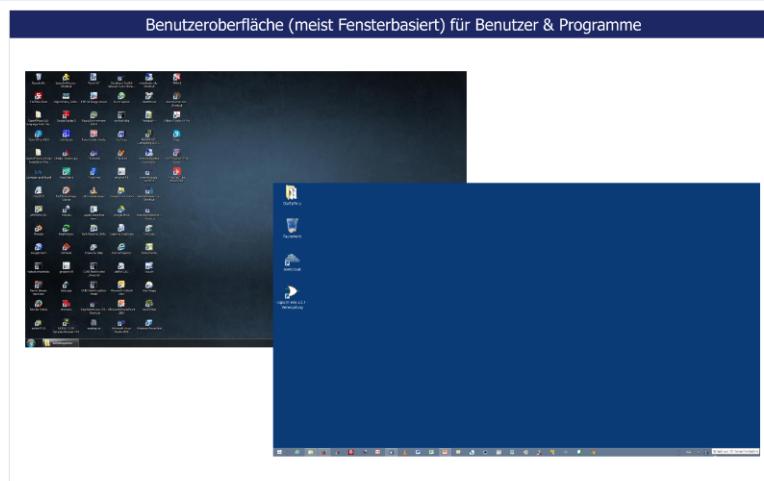
Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

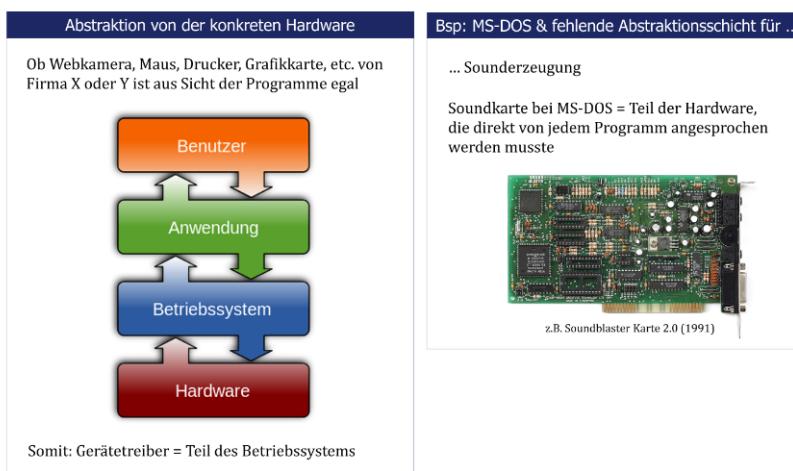
21

Nennen und beschreiben Sie alle wesentlichen Dienstleistungen die ein Betriebssystem zur Verfügung stellt.

1. Dienstleistung des Betriebssystem



2. Dienstleistung des Betriebssystem



3. Dienstleistung des Betriebssystem



Dateisysteme als Abstraktionsschicht vom Speichermedium

- Dateisysteme: FAT32, NTFS, ReFS (Resilient File System → Win8), ext2, ext3, ext4, ReiserFS, Reiser4, TuxFS, etc.
- Egal ob Festplatte, CD, DVD, Blu-Ray, USB-Stick, SD-Card, Netzlaufwerk, etc. die Zugriffsweise ist immer gleich



Bildquellen:
USB-Stick: http://csimages-studenten.cs.uni-frankfurt.de/images/f/1195c9f28d11_3216f1a66fT.jpg
CD: <http://de.wikipedia.org/w/index.php?title=CD&oldid=15800120>
Blu-ray: <http://de.wikipedia.org/w/index.php?title=Blu-ray>

Dateisystem-Rechteverwaltung

Wer darf auf Datei X wie zugreifen?



4. Dienstleistung des Betriebssystem



Prozessverwaltung

- Starten/Stoppen von Prozessen („Programme“)
- Hilfe bei Vermeidung möglicher Konflikte parallel arbeitenden Threads („Programmfäden“)
- Zuordnung: Rechenkern / Rechenzeit
- Zuteilung von Speicher und anderer beschränkter Betriebsmittel (z.B. gleichzeitiger Zugriff mehrerer Prozesse auf die gleiche Festplatte)
- Überwachen von Speicherzugriffsverletzungen

Beispiel: Speicherzugriffsverletzung

```
#include <iostream>
int main()
{
    int *a = new int[5];
    // a[0] = 2*0
    // a[1] = 2*1
    // a[2] = 2*2
    // a[3] = 2*3
    // a[4] = 2*4
    for (int i=0; i<50; i++)
    {
        a[i] = 2*i;
        std::cout << a[i] << std::endl;
    }
    return 0;
}
```

Access Violation Error

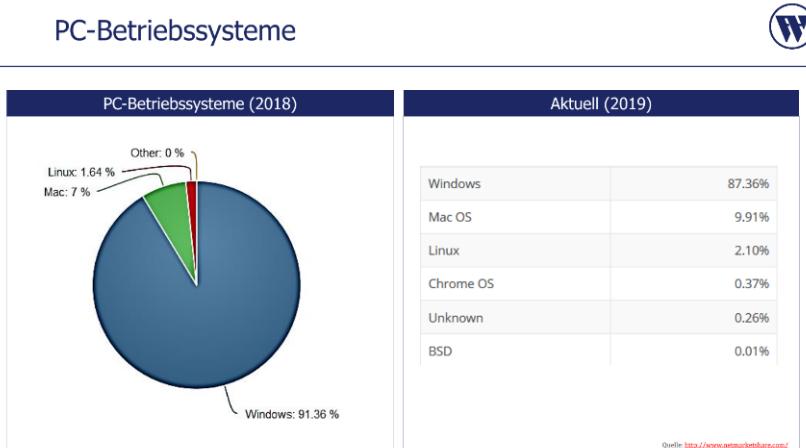
Prozess- und Betriebsmittelverwaltung

Abstraktion von der Hardware
Benutzeroberfläche

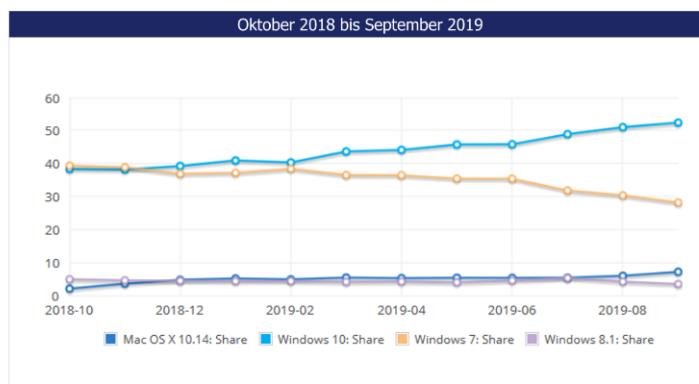


Zwischenfazit: Betriebssystem als Dienstleister!

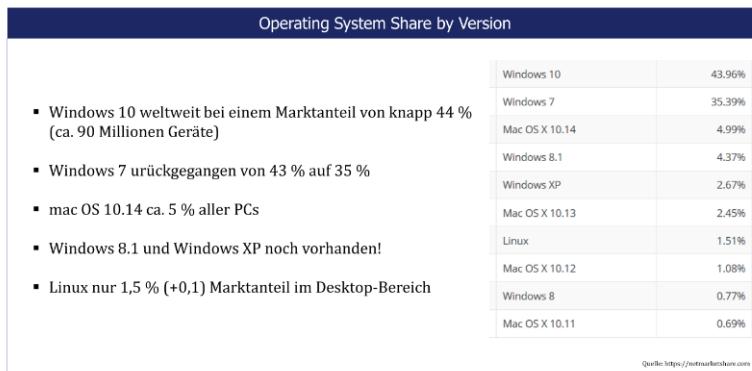
Beschreiben Sie grob die Marktanteile der PC-Betriebssysteme.



Marktanteile der führenden Betriebssysteme I

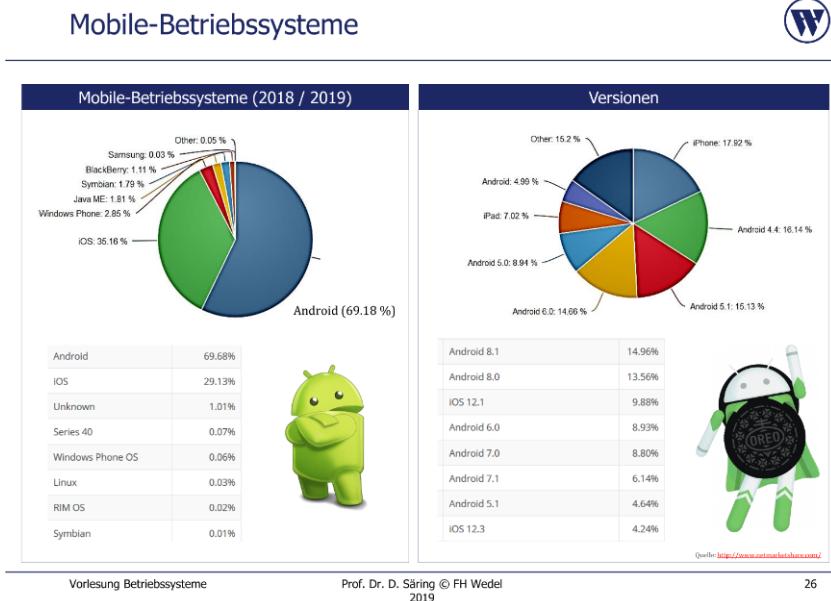


Marktanteile der führenden Betriebssysteme II



... und im mobilen Bereich?

Beschreiben Sie grob die Marktanteile der **mobilen-Betriebssysteme**.



Einführung und Überblick



Lernziele: Einführung und Überblick

- Wozu benötigt man ein Betriebssystem bzw. welche Aufgaben erfüllt ein Betriebssystem?
- Welche Arten von Betriebssystemen gibt es?
- Was ist der „Kernel“?



Nennen Sie die Besonderheiten von **Server-Betriebssystemen**.

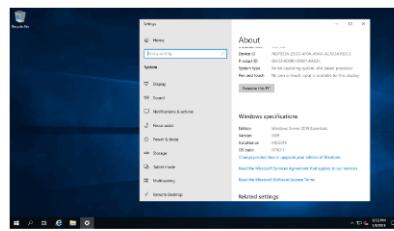
Server-Betriebssysteme



Bei Servern stehen andere Aufgaben im Vordergrund

- **Multi-User Fähigkeit:**
mehrere Benutzer sollen gleichzeitig den Rechner verwenden können
- Bereitstellung bestimmter **Dienste**:
 - **Fileserver**
 - **eMail-Server:** z.B. Microsofts „Exchange Server“ (Groupware) kann nur auf Microsoft Server BS installiert werden
 - **Rechteverwaltung beim Zugriff auf Ressourcen**
in einem Firmennetzwerk (Loginrechte auf PCs, Druckerzugriff) z.B. Microsofts „Active Directory“ in Windows Server BS
 - **Webserver** z.B. Microsofts „Internet Information Server (IIS)“

Beispiel: MS Server 2019



Quelle: <http://www.server-essential.com>

Nennen Sie die Besonderheiten von **Echtzeitbetriebssystemen**. Und nennen Sie einige Beispiele.

Echtzeitbetriebssysteme (Real Time Operating Systems)

Wieder andere Rahmenbedingungen & Anforderungen

- oft leistungsschwächere Hardware als im Desktop PC / Server - Bereich (weniger Speicher, weniger Rechenleistung, ...)
- Echtzeit-Betriebssystem muss kompakt sein / Ressourcen-Bedarf muss sehr gering sein
- Garantieren von
 - stabilem Betrieb
 - sofortiger Abarbeitung von Prozessen
 - vorhersagbaren Laufzeiten von Prozessen stehen im Vordergrund (Beispiel: Radarbasierter Notbremsassistent)
- geringere BS-Lizenzkosten für eingebettete Systeme

Fahrzeuge = Komplexe Rechensysteme



„Sie müssen das System neu starten, damit ihre Änderungen wirksam werden“ ... *

Quellen:
Auto: <http://www.radiotest.com/de/reviews/autotests/reviews/das-schweizer-auto-des-jahres-2014-der-peugeot-308-setzt-sich-frei-aus>
Flugzeug: <http://www.flyertalk.com/forum/airlines-airports/1021173-airbus-a380-prime-airlines-airbus-a380-wie-kommen-i.html>
* nach einer Idee von Friedhelm Stappert / Hochschule Darmstadt

Echtzeitbetriebssysteme (II)

Beispiele von Echtzeitbetriebssystemen



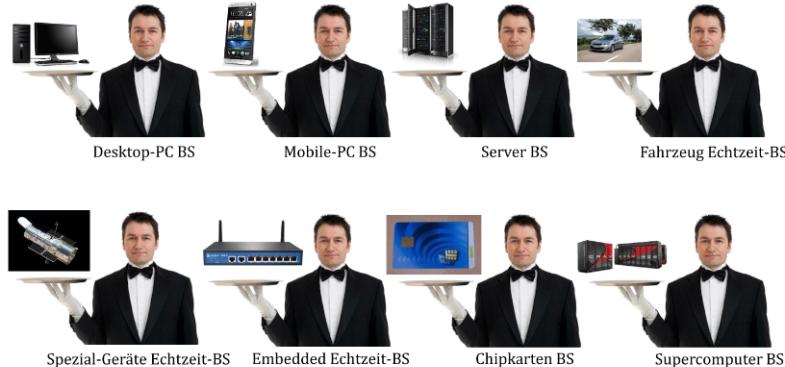
RTLinux

The screenshot shows the RTLinux project page on the Linux Foundation Wiki. It features a large image of a penguin holding a clock, the text "RTLinux", and a link to the official website. Below the image, there's a summary of what RTLinux is and how it can be used. There are sections for "Versions of PREEMPT_RT patches", "Community News", and "Communication". The "Community News" section includes a link to the Real-Time Summit 2010 in Berlin.



Fassen Sie die verschiedenen Arten von Betriebssystemen zusammen.

Vielfalt der Betriebssysteme



Bildergalerie
Desktop PC: <http://sofortkostenrechnung/2012/07/www-in-bezug-ublich-syntaxis-mit-hilfe-oder-desktop/>
Smartphone: <http://techcheck-test.de/de/marco/umfrage-vergleich-sentago/>
Server: <http://www.techcheck.com/blog/unreaktiv-server-jetzt-guenstig-bei-ebay/>

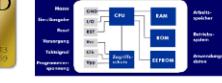
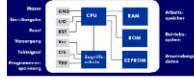
Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

37

Weitere Betriebssysteme



<p>Router-BS</p>  <p>Quelle: http://wellrounded.net/green.aspx/2011/12/29/vmware-forswings-juniper-sys-5-on-the-main-router/</p>	<p>SmartCard (Prozessorchipkarten vs. Speicherchipkarten) BS</p>  <p>BasicCard, CombOS, CardOS, JCOP, MTCOS, MultOS, SECOS, Sicrypt, STARCOS, TCOS, etc.</p> <p></p> <p>Zum Beispiel Speichern eines Private Keys auf der SmartCard und direkte Verschlüsselung von Daten auf der Karte mittels des Karten-Prozessors</p> <p>Quelle: http://www.katzen Infos.de/fachinfo/elektronik/Smartcard-smartcard.html</p>
<p>Supercomputer BS</p> <p>Cray Linux Environment (CLE), früher "UNICOS" skaliert auf bis zu mehr als 500.000 Prozessor-Kerne</p>  <p>Quelle: http://www.cray.com/Products/Computing/SC.aspx</p>	<p>Verteilte Betriebssysteme</p> <p>d.h. Virtualisierung von verteilten Ressourcen wie Speicher, Prozesse, etc. z.B. LinuxPMI = Linux Process Migration Infrastructure</p>  <p>Quelle: http://lxpmi.org/lxpmi/</p>

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

36

Was ist ein **Kernel** und woraus bestehen seine wesentlichen Aufgaben?

Zusammenfassung - Kernel: Software Kern des Betriebssystems, ist selbst ein Programm welches Dienste bereitstellt.

Aufgaben des BS-Kern (Kernel)



Typische BS-Kern Aufgaben		Weitere nicht BS-Kern Aufgaben	
<ul style="list-style-type: none"> ▪ Abstraktion von der Hardware ▪ Speicherverwaltung ▪ Prozessverwaltung insbesondere IPC = Inter-Prozess-Communication ▪ Geräteverwaltung <i>• Wichtig: "GUI/Desktop nicht Teil d. kernels."</i> ▪ Dateisystemzugriff 		<ul style="list-style-type: none"> ▪ Grafische Benutzeroberfläche z.B. in Linux: Fenstermanager KDE, Gnome, Unity (Ubuntu) ▪ Shell / Kommandozeile ▪ BS-Tools 	

Aber: Liste der Aufgaben die vom BS-Kern erfüllt werden hängt vom BS-Kern Typ ab ...

Nennen Sie die Hauptarten von BS-Kernel.

Klassen von Kernel



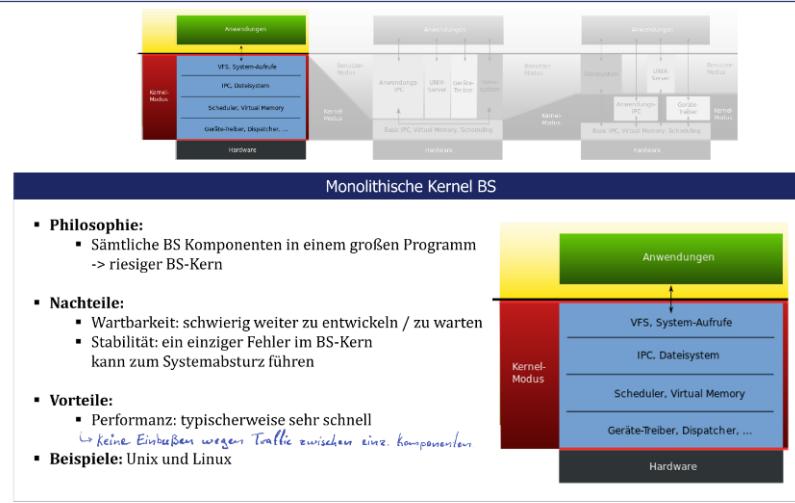
BS-Kerne lassen sich anscheinend klassifizieren					...
Kernel name	Used in	Creator	Executable format (also see section below)	Type	
Agnix [3] a	?	?	?	?	
Amiga Exec	AmigaOS	Commodore International	HUNK	Exokernel (atypical) ^[4]	
Amiga Exec SG (2nd Generation)	AmigaOS 4	Hyperion Entertainment	ELF/HUNK	Exokernel (atypical)	
DragonFly BSD kernel	DragonFly BSD	Matt Dillon	ELF, others - platform dependent	hybrid	
FreeBSD kernel	FreeBSD, Debian GNU/kFreeBSD, Gentoo/FreeBSD, Orbit OS	The FreeBSD Project	ELF, others - platform dependent	monolithic	
GNU/Hurd (Arch)	GNU				

SunOS kernel	SunOS	Sun Microsystems	a.out	monolithic
Solaris kernel	Solaris, OpenSolaris, GNUnix/OpenSolaris (Nexenta OS)	Sun Microsystems	ELF (32 bit only until Solaris 7 in 1998)	monolithic
Trix kernel	Trix, GNU/KTrix	Massachusetts Institute of Technology	a.out	monolithic
Windows NT kernel [12] a	Windows NT, 2000, XP, 2003, Vista, Windows 7, Windows 8	Microsoft	PE, others?	hybrid
XNU (Darwin kernel) [13] a	Mac OS X, iOS, OpenDarwin, PureDarwin GNU/Darwin	Apple Computer	Mach-O	hybrid
SPARTAN kernel [14] a	HelenOS	Jakub Jermar	ELF	microkernel

Quelle: http://en.wikipedia.org/wikia/comparison_of_operating_system_kernels

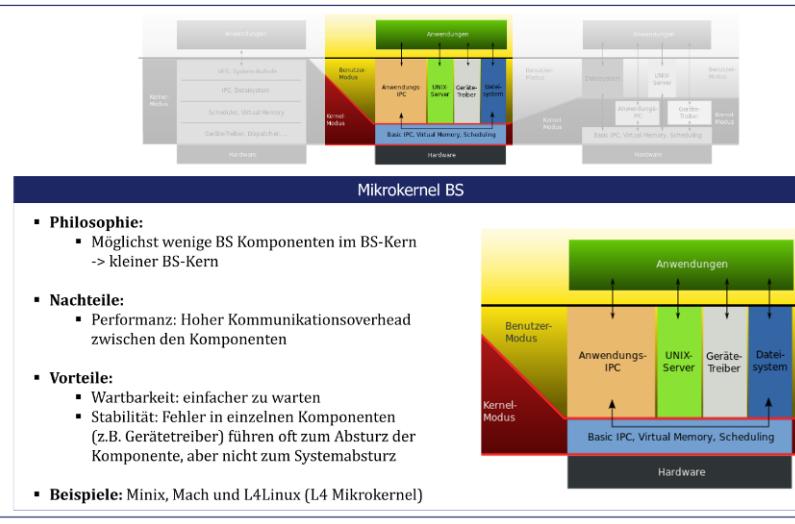
Beschreiben Sie die *Philosophie*, *Nachteile* und die *Vorteile* eines monolithischen Kernel BS .

Die 3 Hauptarten von BS-Kernel



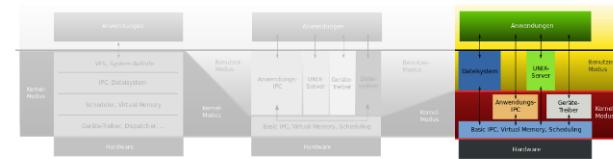
Beschreiben Sie die *Philosophie*, *Nachteile* und die *Vorteile* eines Mikrokernels BS .

Die 3 Hauptarten von BS-Kernel



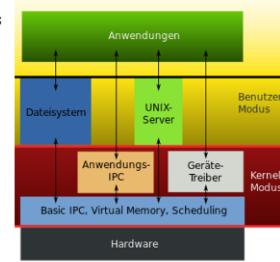
Beschreiben Sie die *Philosophie*, *Nachteile* und die *Vorteile* eines **hybridenBS Kernels**.

Die 3 Hauptarten von BS-Kernel



Hybride BS Kernel

- **Philosophie:**
 - Einige Dienste (z.B. Anwendungs-IPC) auch im BS-Kern aus Performanz-Gründen
- **Nach- und Vorteile:**
 - Kompromiss zwischen Performanz und Stabilität/Wartbarkeit
- **Beispiele:** Windows und OS X



Fassen Sie noch einmal die verschiedenen Klassen von Kernels zusammen und grenzen Sie diese voneinander ab.

Klassen von Kernel



BS-Kerne lassen sich anscheinend klassifizieren					...
Kernel name	Used in	Creator	Executable format (also see section below)	Type	
Agnix [3] ⓘ	?	?	?	?	
Amiga Exec	AmigaOS	Commodore International	HUNK	Exokernel (atypical) [4]	
Amiga Exec SG (2nd Generation)	AmigaOS 4	Hyperion Entertainment	ELF/HUNK	Exokernel (atypical)	
DragonFly BSD kernel	DragonFly BSD	Matt Dillon	ELF, others - platform dependent	hybrid	
FreeBSD kernel	FreeBSD, Debian, GNU/FreeBSD, Gentoo/FreeBSD, Orbis OS	The FreeBSD Project	ELF, others - platform dependent	monolithic	
GNU/Hurd (Arch)	GNU				

SunOS kernel	SunOS	Sun Microsystems	a.out	monolithic
Solaris kernel	Solaris, OpenSolaris, GNU/kOpenSolaris (Nexenta OS)	Sun Microsystems	ELF (32-bit only until Solaris 7 in 1998)	monolithic
Trix kernel	Trix, GNU/kTrix	Massachusetts Institute of Technology	a.out	monolithic
Windows NT kernel [12] ⓘ	Windows NT, 2000, XP, 2003, Vista, Windows 7, Windows 8	Microsoft	PE, others?	hybrid
XNU (Darwin kernel) [13] ⓘ	Mac OS X, iOS, OpenDarwin, PureDarwin, GNU/Darwin	Apple Computer	Mach-O	hybrid
SPARTAN kernel [14] ⓘ	HelenOS	Jakub Jermar	ELF	microkernel

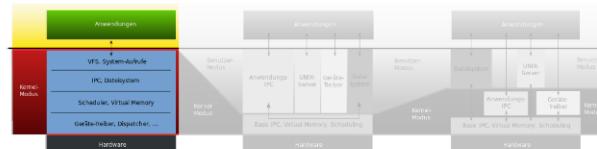
Quelle: http://en.wikipedia.org/wiki/Comparison_of_operating_system_kernels

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

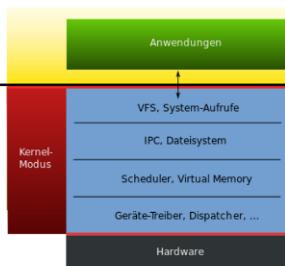
43

Die 3 Hauptarten von BS-Kernel



Monolithische Kernel BS

- **Philosophie:**
 - Sämtliche BS Komponenten in einem großen Programm -> riesiger BS-Kern
- **Nachteile:**
 - Wartbarkeit: schwierig weiter zu entwickeln / zu warten
 - Stabilität: ein einziger Fehler im BS-Kern kann zum Systemabsturz führen
- **Vorteile:**
 - Performanz: typischerweise sehr schnell
 - ↳ keine Einbußen wegen Trafik zwischen Komponenten
- **Beispiele:** Unix und Linux

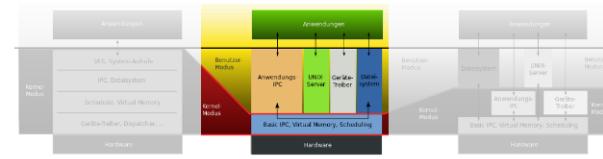


Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

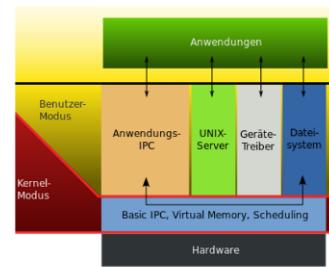
44

Die 3 Hauptarten von BS-Kernel

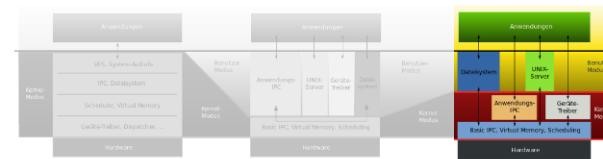


Mikrokernel BS

- **Philosophie:**
 - Möglichst wenige BS Komponenten im BS-Kern
-> kleiner BS-Kern
- **Nachteile:**
 - Performanz: Hoher Kommunikationsoverhead zwischen den Komponenten
- **Vorteile:**
 - Wartbarkeit: einfacher zu warten
 - Stabilität: Fehler in einzelnen Komponenten (z.B. Gerätetreiber) führen oft zum Absturz der Komponente, aber nicht zum Systemabsturz
- **Beispiele:** Minix, Mach und L4Linux (L4 Mikrokernel)

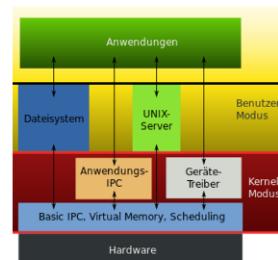


Die 3 Hauptarten von BS-Kernel



Hybride BS Kernel

- **Philosophie:**
 - Einige Dienste (z.B. Anwendungs-IPC) auch im BS-Kern aus Performanz-Gründen
- **Nach- und Vorteile:**
 - Kompromiss zwischen Performanz und Stabilität/Wartbarkeit
- **Beispiele:** Windows und OS X



Nennen Sie die wesentlichen Vorteile von **UEFI** gegenüber **BIOS** !
!missing IDENTIFIER!

Wiederholung (I)


Studienfach: http://www.ph-wedel-potsdam.de

Lernkontrolle

- Nennen Sie die wesentlichen Vorteile von UEFI gegenüber BIOS!
 - Unterstützung von Laufwerken mit mehr als 2.2 TB
 - Schnelleres Booten
 - Paralleles Laden der Gerätetreiber
 - UEFI Treiber werden vom BS wiederverwendet
 - Netzwerkzugriff schon zur Bootzeit
 - UEFI-Shell: Eigene Kommandozeile
 - Bootmanager zur Auswahl des zu startenden BS ist schon integriert

- Nennen Sie die wesentlichen „Dienstleistungen“ eines Betriebssystems!
 - Prozess- und Betriebsmittelverwaltung
 - Abstraktion der Hardware
 - Bereitstellung einer Benutzeroberfläche
 - Umsetzung von Dateisystemen



Nennen Sie die wesentlichen Dienstleistungen eines Betriebssystems

Wiederholung (I)


Studienfach: http://www.ph-wedel-potsdam.de

Lernkontrolle

- Nennen Sie die wesentlichen Vorteile von UEFI gegenüber BIOS!
 - Unterstützung von Laufwerken mit mehr als 2.2 TB
 - Schnelleres Booten
 - Paralleles Laden der Gerätetreiber
 - UEFI Treiber werden vom BS wiederverwendet
 - Netzwerkzugriff schon zur Bootzeit
 - UEFI-Shell: Eigene Kommandozeile
 - Bootmanager zur Auswahl des zu startenden BS ist schon integriert

- Nennen Sie die wesentlichen „Dienstleistungen“ eines Betriebssystems!
 - Prozess- und Betriebsmittelverwaltung
 - Abstraktion der Hardware
 - Bereitstellung einer Benutzeroberfläche
 - Umsetzung von Dateisystemen



Welche der Kernel Aufgaben gehört **NICHT** zu den Aufgaben eines BS-Kerns?

Wiederholung (II)



Bildquelle: <http://www.phoebes-puzzle.de>

Lernkontrolle

- Welche der folgenden Aufgaben gehört **NICHT** zu den Aufgaben eines BS-Kerns?
 - Abstraktion von der Hardware
 - Dateisystemzugriff
 - Grafische Benutzeroberfläche
 - Geräteverwaltung

- Beschreiben Sie die drei Hauptarten von BS-Kernel und gehen Sie dabei auf die Vor- und Nachteile ein!
 - Monolithische Kernel BS
 - (+) Performanz
 - (-) Wartbarkeit und Stabilität
 - Mikrokern BS
 - (-) Wartbarkeit und Stabilität
 - (-) Performanz
 - Hybride BS Kernel
 - Kompromiss zwischen Performanz und Stabilität/Wartbarkeit



Beschreiben Sie die drei Hauptarten von BS-Kernel und gehen Sie dabei auf die Vor- und Nachteile ein!

Wiederholung (II)



Bildquelle: <http://www.phoebes-puzzle.de>

Lernkontrolle

- Welche der folgenden Aufgaben gehört **NICHT** zu den Aufgaben eines BS-Kerns?
 - Abstraktion von der Hardware
 - Dateisystemzugriff
 - Grafische Benutzeroberfläche
 - Geräteverwaltung

- Beschreiben Sie die drei Hauptarten von BS-Kernel und gehen Sie dabei auf die Vor- und Nachteile ein!
 - Monolithische Kernel BS
 - (+) Performanz
 - (-) Wartbarkeit und Stabilität
 - Mikrokern BS
 - (-) Wartbarkeit und Stabilität
 - (-) Performanz
 - Hybride BS Kernel
 - Kompromiss zwischen Performanz und Stabilität/Wartbarkeit



Definieren Sie den Begriff des **Prozesses**. Was versteht man in diesem Zusammenhang unter der der *Quasiparallelität*?missing IDENTIFIER;

Prozesstabelle

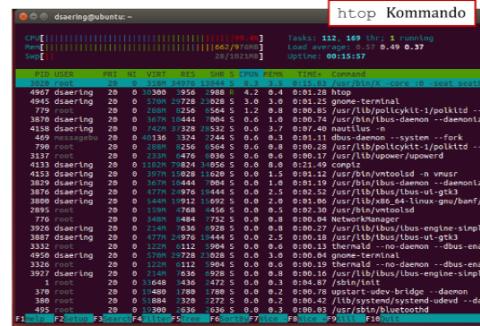


Prozesstabelle		Prozesskontrollblock (PCB)
speichert PCBs aller Prozesse		Der Prozesskontrollblock (Prozesskontext oder Task Control Block) ist eine Datenstruktur , die vom BS verwendet wird, um die assoziierten Daten für einen Prozess zu speichern
Assozierte Daten		
Metadaten , die BS mit Prg assoziiert:		
<ul style="list-style-type: none"> ▪ Prozessnummer (PID) ▪ Prozessbesitzer ▪ Priorität ▪ Prozessrechte ▪ Befehlszähler ▪ Scheduling-Informationen ▪ ... 		
Reservierte Ressourcen		
<ul style="list-style-type: none"> ▪ Speicher <ul style="list-style-type: none"> ▪ Heap (Datenspeicher) ▪ „Call Stack“ (Liste der aufgerufenen Funktionen) ▪ File Handles ▪ CPU Register 		

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

55



Prozessmodell



Aspekt der Laufzeit		WER ZEIT HAT: HIER ANSTEHEN	WER KEINE ZEIT HAT: HIER ANSTEHEN
<ul style="list-style-type: none"> ▪ Prozesse wechseln sich mit der Nutzung der CPU ab (Beispiel: Warteschlange) ▪ Geschwindigkeit des Programmablaufs ist daher nicht einheitlich und selten reproduzierbar (Beispiel: Audioprozess mit zeitlicher Verzögerung) <ul style="list-style-type: none"> ▪ Spezialfall: Echtzeitbetriebssysteme 			
Allgemein		<p>... mal schauen wie ein Prozessmodell umgesetzt wird!</p>	
<ul style="list-style-type: none"> ▪ Prozesse sind eine Aktivität, das auf einem Programm basiert, Ein- und Ausgaben sowie einen Zustand besitzt ▪ Prozesse teilen sich die CPU, wobei die Nutzung durch eine Schedulingstrategie bestimmt wird ▪ <i>Anmerkung:</i> Ein Programm, das zweimal ausgeführt wird läuft in unterschiedlichen Prozessen ab 			

Vorlesung Betriebssysteme

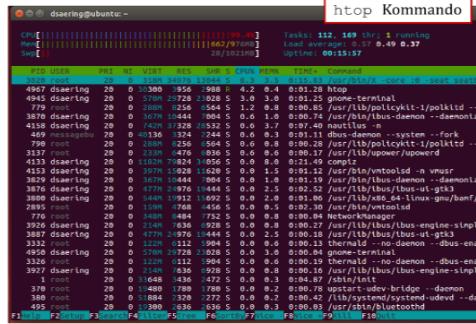
Prof. Dr. D. Säring © FH Wedel
2019

54

Was versteht man unter dem sog. **Prozessmodell**?
 Erläutern Sie in diesem Zusammenhang grob, was man unter einer *Schedulingstrategie* versteht.

Prozesstabelle



Prozesstabelle	Prozesskontrollblock (PCB)
speichert PCBs aller Prozesse	Der Prozesskontrollblock (Prozesskontext oder Task Control Block) ist eine Datenstruktur , die vom BS verwendet wird, um die assoziierten Daten für einen Prozess zu speichern
Assoziierte Daten	
Metadaten , die BS mit Prg assoziiert: <ul style="list-style-type: none"> ▪ Prozessnummer (PID) ▪ Prozessbesitzer ▪ Priorität ▪ Prozessrechte ▪ Befehlszähler ▪ Scheduling-Informationen ▪ ... Reservierte Ressourcen <ul style="list-style-type: none"> ▪ Speicher <ul style="list-style-type: none"> ▪ Heap (Datenspeicher) ▪ „Call Stack“ (Liste der aufgerufenen Funktionen) ▪ File Handles ▪ CPU Register 	 <p>htop Kommando</p>

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

55

Prozessmodell



Aspekt der Laufzeit
<ul style="list-style-type: none"> ▪ Prozesse wechseln sich mit der Nutzung der CPU ab (Beispiel: Warteschlange) ▪ Geschwindigkeit des Programmablaufs ist daher nicht einheitlich und selten reproduzierbar (Beispiel: Audioprozess mit zeitlicher Verzögerung) <ul style="list-style-type: none"> ▪ Spezialfall: Echtzeitbetriebssysteme



Quelle: die-besten-100.de

Allgemein
<ul style="list-style-type: none"> ▪ Prozesse sind eine Aktivität, das auf einem Programm basiert, Ein- und Ausgaben sowie einen Zustand ▪ Prozesse teilen sich die CPU, wobei die Nutzung durch eine Schedulingstrategie bestimmt wird ▪ <i>Anmerkung:</i> Ein Programm, das zweimal ausgeführt wird läuft in unterschiedlichen Prozessen ab

... mal schauen wie ein Prozessmodell umgesetzt wird!

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

54

Beschreiben Sie den Begriff der **Prozesstabellen**. Welche Daten werden hierbei festgehalten?



Prozesstabellen

Prozesstabellen		Prozesskontrollblock (PCB)
speichert PCBs aller Prozesse		Der Prozesskontrollblock (Prozesskontext oder Task Control Block) ist eine Datenstruktur , die vom BS verwendet wird, um die assoziierten Daten für einen Prozess zu speichern
Assoziierte Daten		
<p>Metadaten, die BS mit Prg assoziiert:</p> <ul style="list-style-type: none"> ▪ Prozessnummer (PID) ▪ Prozessbesitzer ▪ Priorität ▪ Prozessrechte ▪ Befehlszähler ▪ Scheduling-Informationen ▪ ... <p>Reservierte Ressourcen</p> <ul style="list-style-type: none"> ▪ Speicher <ul style="list-style-type: none"> ▪ Heap (Datenspeicher) ▪ „Call Stack“ (Liste der aufgerufenen Funktionen) ▪ File Handles ▪ CPU Register 	<p>htop Kommando</p>	

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel

2019 55

Was versteht man unter einem *Prozesskontrollblock*?missing IDENTIFIER_i

Prozesstabelle



Prozesstabelle	Prozesskontrollblock (PCB)
<p>speichert PCBs aller Prozesse</p> <p>Assozierte Daten</p> <p>Metadaten, die BS mit Prg assoziiert:</p> <ul style="list-style-type: none"> ▪ Prozessnummer (PID) ▪ Prozessbesitzer ▪ Priorität ▪ Prozessrechte ▪ Befehlszähler ▪ Scheduling-Informationen ▪ ... <p>Reservierte Ressourcen</p> <ul style="list-style-type: none"> ▪ Speicher <ul style="list-style-type: none"> ▪ Heap (Datenspeicher) ▪ „Call Stack“ (Liste der aufgerufenen Funktionen) ▪ File Handles ▪ CPU Register 	<p>Der Prozesskontrollblock (Prozesskontext oder Task Control Block) ist eine Datenstruktur, die vom BS verwendet wird, um die assoziierten Daten für einen Prozess zu speichern</p>

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel

55

Prozesskontrollblock (PCB) in Linux



Anfang der Datenstruktur: Zeile 1282 in sched.h	Ende der Datenstruktur: Zeile 1712 in sched.h
<pre> 1280 } 1281 struct task_struct { 1282 volatile long state; /* -1 = unreachable, 0 runnable, > 0 stopped */ 1283 void *stack; 1284 atomic_t user; 1285 unsigned int flags; /* per process flags, defined below */ 1286 unsigned int priority; 1287 unsigned int static_prio, normal_prio; 1288 int on_rq; 1289 unsigned int preempt_count; 1290 const struct sched_class *which_class; 1291 struct sched_entity entity; 1292 struct list_head rt_rq; 1293 unsigned long wake_ip; 1294 unsigned long wake_ip_decr; 1295 unsigned long wake_ip_rq; 1296 int wake_ip_rq; 1297 int on_rq_rq; 1298 int on_rq_rq_rq; 1299 int prio, static_prio, normal_prio; 1300 unsigned int rt_rq_prio; 1301 const struct sched_group *which_group; 1302 struct sched_entity *entity_rq; 1303 struct list_head preempt_notifier; 1304 struct list_head preempt_ip_notifier; 1305 struct list_head preempt_ip_rq_notifier; 1306 struct task_group *sched_task_group; 1307 struct { 1308 struct sched_entity entity; 1309 struct list_head preempt_notifier; 1310 struct list_head preempt_ip_notifier; 1311 /* list of struct preempt_notifier */ 1312 struct list_head preempt_ip_rq_notifier; 1313 } preempt; 1314 #ifdef CONFIG_SCHED_HRT_TIMER 1315 unsigned int hrt_clock_seq; 1316 #endif 1317 #ifdef CONFIG_PREEMPT_NOTIFIERS 1318 unsigned int preempt_ip_rq; 1319 unsigned int preempt_ip_rq_rq; 1320 int w_ip_rq_rq_rq; 1321 cpumask_t cpus_allowed; 1322 </pre>	<pre> 1675 int curr_ret_stack; 1676 /* Return stack addresses for return function tracing */ 1677 struct ftrace_ret_stack *ret_stack; 1678 /* time stamp of the last trace timestamp */ 1679 unsigned long trace_timestamp; 1680 /* Number of functions that haven't been traced 1681 * because of depth overrun. 1682 */ 1683 atomic_t trace_overruns; 1684 /* Pauses for the tracing */ 1685 atomic_t trace_stopped; 1686 atomic_t trace_stopped_wq; 1687 #endif 1688 #ifdef CONFIG_SCHED_TRACING 1689 /* state flags for use by tracers */ 1690 unsigned long trace; 1691 /* number of levels of trace recursion */ 1692 unsigned long trace_recursion; 1693 #endif 1694 #ifdef CONFIG_NCLS 1695 struct memcontrol *memcontrol; 1696 /* Group memcontrol */ 1697 gfp_t gfp_mask; 1698 int order; 1699 unsigned int may_oom; 1700 #endif 1701 #endif 1702 #ifdef CONFIG_UPRIBES 1703 /* Task migration task_struct */ 1704 #endif 1705 #if defined(CONFIG_BCACHE) defined(CONFIG_BCACHE_MODULE) 1706 unsigned int sequential_io; 1707 unsigned int sequential_io_avg; 1708 #endif 1709 #ifdef CONFIG_DEBUG_ATOMIC_SLEEP 1710 unsigned long task_state_change; 1711 #endif 1712 }; 1713 /* Future-safe accessor for struct task_struct's cpus_allowed. */ 1714 #define task_cpus_allowed(task) ((task)->cpus_allowed) 1715 #define TIF_MIGRATED 0x01 1716 </pre>

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

56

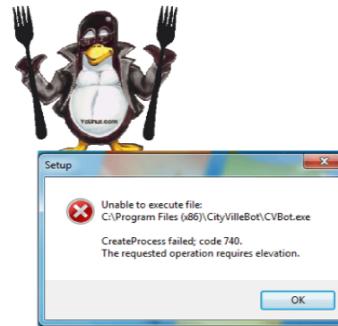
Erläutern Sie den Begriff der **Prozesserzeugung**. Wann ist so etwas sinnvoll, welche Möglichkeiten der Erzeugung gibt es, welche Befehle werden verwendet?

Prozesserzeugung



Prozesserzeugung

- Erzeugung von Prozessen können verursacht werden durch:
 - Initialisierung des Systems
 - Systemaufruf zum Erzeugen eines Prozesses durch einen anderen Prozess
 - Benutzeranfrage, einen neuen Prozess zu erzeugen
 - Initiierung einer Stapelverarbeitung
- Prozesse werden unterschieden in **Vordergrund** und **Hintergrundprozesse** (u.a. Daemons)
- Prozesserzeugung ist immer dann hilfreich, wenn Arbeit auf mehrere in Verbindung stehende aber unabhängig voneinander interagierende Prozesse aufteilbar ist! (Beispiel: BigData Processing)
- Ein Prozess wird immer durch einen anderen Prozess mit Hilfe eines Systemaufrufs erzeugt (`fork / createProcess`)



... und wie beenden wir Prozesse?

Wie kann ein Prozess beendet werden? Welche unterschiedlichen Arten gibt es, erläutern Sie dies am Beispiel vom BS Linux.

Prozessbeendigung



Prozessbeendigung

- Ein Prozess **terminiert** aufgrund einer der folgenden Bedingungen:
 - Normales Beenden (freiwillig)
 - Beenden durch Fehler (freiwillig)
 - Beenden durch schwerwiegenden Fehler (unfreiwillig)
 - Beenden durch anderen Prozess (unfreiwillig) (z.B. `KILL` da selbst ein Programm)
- **Freiwilliges Beenden** z.B. nach vollständiger Erfüllung der Aufgabe erfolgt durch einen Systemaufruf (`exit / exitProcess`)
- **Unfreiwilliges Beenden** z.B. durch Programmierfehler die durch das System erkannt werden oder durch einen Befehl eines anderen Prozesses (`kill / terminateProcess`)

HANDLING NON-RESPONDING & FROZEN APPLICATIONS



Erläutern Sie die unterschiedlichen Prozesshierarchien von Windows und Unix-Maschinen. Wie können solche Prozesse erzeugt werden und wie sieht deren Beziehung untereinander aus.

Prozesshierarchien: Eltern- / Kindprozesse

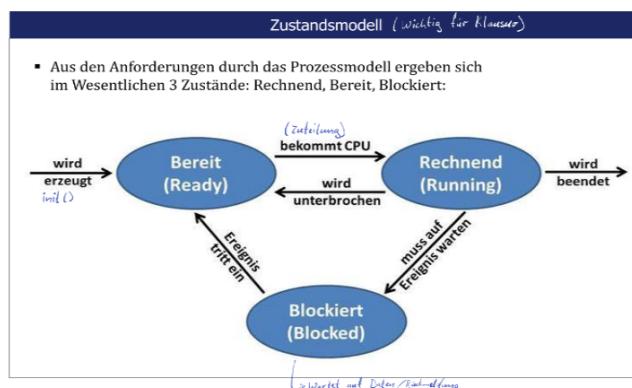


Adressraum	Beziehung
<ul style="list-style-type: none"> ▪ UNIX: Bei der Erzeugung eines Prozesses mit <code>fork</code> wird eine exakte Kopie des Prozesses angelegt, dabei hat der Kindprozess <ul style="list-style-type: none"> ▪ das gleiche Speicherabbild ▪ die gleichen Umgebungsvariablen und ▪ die gleiche geöffneten Dateien ▪ u.a. durch <code>execve</code> kann nach der Erzeugung das Speicherabbild gem. der neuen Aufgabe gewechselt werden ▪ Windows: Mit <code>createProcess</code> werden beide o.g. Schritte auf einmal durchgeführt ▪ Generell: Eltern- und Kindprozess besitzen nach der Erzeugung einen eigenen getrennten Adressraum -> es wird kein beschreibbarer Speicher gemeinsam benutzt 	<ul style="list-style-type: none"> ▪ UNIX: Bei der Erzeugung eines Prozesses durch einen Kindprozess entsteht eine Prozesshierarchie. ▪ Alle Kinder eines Elternprozesses bilden eine Prozessfamilie ▪ Signale werden an alle in Verbindung stehenden Familienmitglieder gesendet bzw. weitergeleitet (Beispiel: Prozess beenden) ▪ Prozesse sind zwar unabhängige Einheiten, müssen aber auch mit anderen Prozessen kommunizieren (... IPC etwas später) ▪ „Der“ Elternprozess: <code>init</code> oder <code>systemd</code> ▪ Windows: Kein Konzept für eine Hierarchie, alle Prozesse sind gleichwertig.

... welchen Zustand können Prozesse haben?

Erläutern Sie das grundsätzliche Zustandsmodell der möglichen Prozesszustände anhand einer Skizze.

Prozesszustand (Standardschema)



Erläutern Sie anhand eines einfachen Beispiels die Sinnhaftigkeit von Threads.

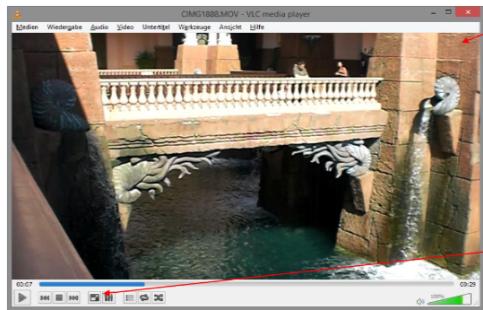
Threads



Motivation: ein Programm und zwei Aufgaben

- wir benötigen 2 Codestränge, die gleichzeitig ausgeführt werden
- aber beide sollten Zugriff auf die gleichen Daten haben

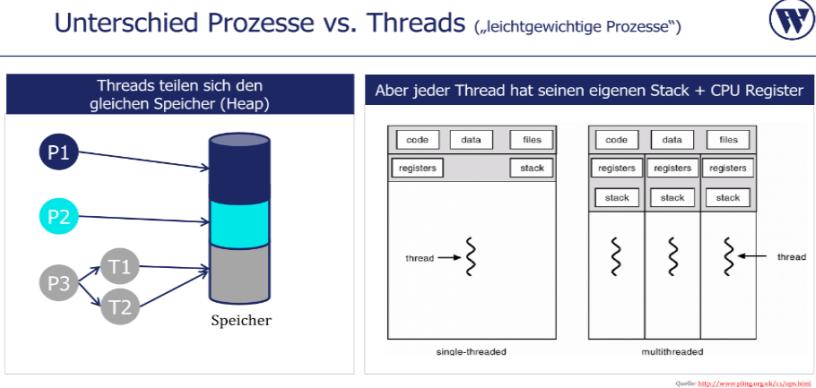
Lösung: Threads!



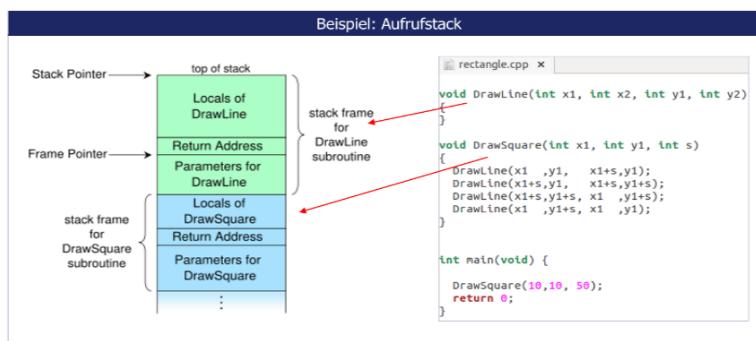
1. Video abspielen

2. Auf Steuerelemente reagieren

Worin genau liegt der Unterschied zwischen *Prozessen* und *Threads*? missing IDENTIFIER_i



Unterschied Prozesse vs. Threads („leichtgewichtige Prozesse“)



Nennen Sie die beiden Möglichkeiten der Thread-erzeugung unter Linux. Worin liegt der Unterschied zur Erzeugung von Prozessen?

Threads unter Linux



Realisierung von Threads

- Threads werden unter Linux im Wesentlichen genauso realisiert wie Prozesse!
- Thread = Prozess, der den gleichen Adressraum besitzt wie sein Elternprozess
- ein neuer Prozess wird mit der BS Funktion `fork()` erzeugt
- ein neuer Thread kann mit der BS Funktion `clone()` erzeugt werden



Software Demo: Threads unter Linux



- man fork**
 - `fork()` Source Code Beispiel
(neuer Prozess wird erzeugt)
- man init**
 - `ps-tree`
- man clone**
 - `clone()` Source Code Beispiel
(neuer Thread wird erzeugt)
- pthread Library Beispiel**
 - `g++ -w pthread_test.cpp -lpthread -o pthread_test`

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

66

fork()



- gabelt vom aktuellen Prozess einen neuen Prozess ab -> Kindprozess

```

Player [Ubuntu - VMware Workstation 12 Player (Non-commercial use only)]
Terminal: Datei Geblättert Ansicht Suchen Terminal Hilfe
[Ubuntu] dwesing@dwesing:~/Desktop/material/Codes$ gedit
[Ubuntu] dwesing@dwesing:~/Desktop/material/Codes$ gedit forktest.cpp
#include <iostream>
#include <sys/types.h>
#include <sys/wait.h>

int counter = 0;

int main()
{
    int pid = fork();
    if(pid == 0)
    {
        // Kindprozess, wenn fork() == 0
        // Pid ist > 0
        counter = counter + 1;
        std::cout << "Elternprozess: counter = " << pid << endl;
        sleep(3);
    }
    else
    {
        if(pid > 0)
        {
            // Elternprozess, wenn fork() > 0
            // Pid ist > 0
            // getpid() gibt die eigene PID zurück
            // pid ist > 0
            std::cout << "Elternprozess: counter = " << pid << endl;
            counter = counter + 1;
            std::cout << "Elternprozess: counter = " << pid << endl;
            sleep(3);
        }
    }
    return 0;
}

```

Vorlesung Betriebssysteme

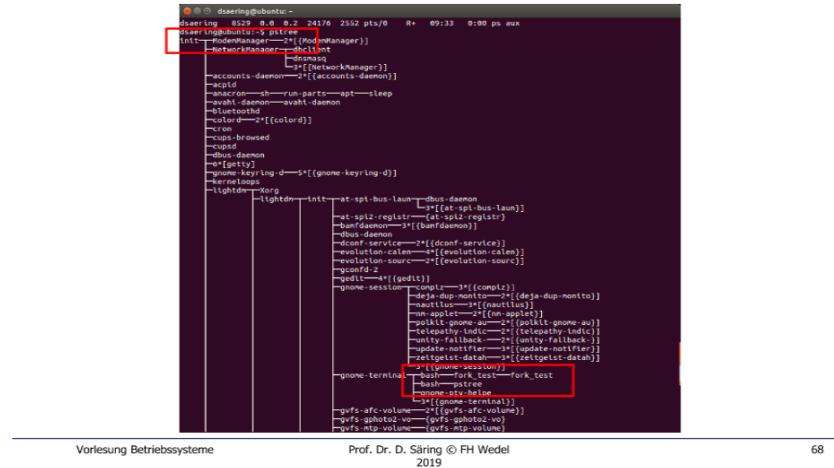
Prof. Dr. D. Säring © FH Wedel
2019

67

pstree bei fork()



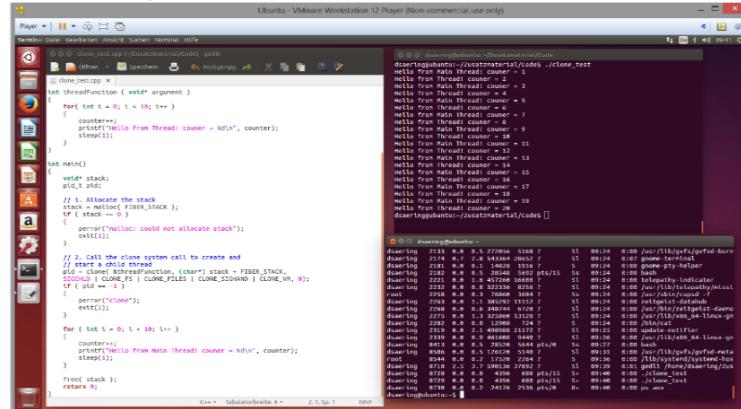
- Alle Prozesse stammen von einem ersten initialen Prozess ab: init



clone()



- gabelt vom aktuellen Prozess einen neuen Prozess ab -> Kindprozess mit teilweise gleichen Ressourcen



pthread



- Bibliothek: POSIX Standardschnittstelle zur Erzeugung und Steuerung von Threads

The screenshot shows a Linux desktop environment with several windows open. On the left, a terminal window displays the output of a C++ program that creates four threads, each printing "Hello World". The code in the terminal is as follows:

```
#include <iostream>
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/thread.h>
#include <sys/malloc.h>
```

using namespace std;

```
void PrintHello(void* thread)
{
    long tld = (long)thread;
    for (int i = 0; i < 10; i++)
        cout<<"Hello World from thread "<<tld<<endl;
    sleep(1);
    pthread_exit(NULL);
}
```

```
int main()
{
    for (int i = 0; i < 4; i++)
        pthread_create(i, NULL, PrintHello, (void *)i);
    for (int i = 0; i < 4; i++)
        cout<<"CRRRR unable to create thread "<<i<<endl;
    exit(1);
}
pthread_exit(NULL);
```

On the right, a code editor window shows the source code for the program. The terminal window has a red box around its output area, and the code editor window has a red box around its title bar.

POSIX = Portable Operating System Interface =
Betriebssystemschnittstellenstandard bestehend aus C-
Systemaufrufsfunktionen, Kommandozeileninterpreter
und Hilfsprogrammen für „unixoide“ Betriebssysteme

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

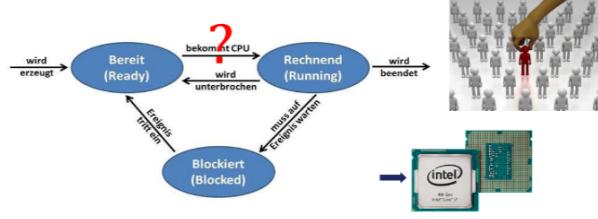
70

Was versteht man unter einem sog. **Scheduler** ?

Prozessverwaltung mittels Scheduler



- **Scheduler** ist der Teil des BS-Kerns, der entscheidet welcher Prozess als nächstes auf der CPU ausgeführt wird (Permanenter Aktionen muss selber auf d. CPU vorhanden aktiv werden um anderen Prozessen ihre Zeitteilung heranzutragen)
 - **Scheduling-Strategie** ist ein Algorithmus / ein Verfahren das beschreibt wie der nächste Prozess ausgewählt wird



Source: <http://www.activeedge.com/selecting-a-business-valuation-method>

→ Folien zum Thema **Process Control Block (PCB)** & Scheduler aus Linux Sicht: <http://www.inf.ufsc.br/~fernando/ine5355/linux-process.pdf>

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

72

Das *Kleingedruckte* : Permanent am Arbeiten, muss selber bei der CPU rechnend aktiv werden um anderen Prozessen ihre Zuteilung bereitzustellen.

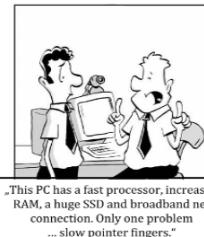
Auf welchen Systemen ist ein **Scheduler** von besonderer Bedeutung? Gehen Sie hierbei insbesondere auf die Gründe ein.

Scheduler (I)



Allgemein

- CPU-Zeit ist heute nur noch selten eine knappe Ressource
 - Programme warten eher auf User-Interaktion
 - CPU rechnet sehr schnell
 - Rechenintensive Anwendung (z.B. Rendern von Videos) bilden eher die Ausnahme
- > Schedulingstrategie auf einfachen PCs nicht sehr wichtig
- Höherer Stellenwert bei Servern und Architekturen mit optimierten Stromverbrauch (z.B. Smartphones)
- **Grund:** Prozesswechsel auf einer CPU ist sehr teuer
 - Wechsel vom Benutzer- in den Kernmodus
 - Sicherung des Zustands (inkl. speichern der Reg.)
 - Speicherzuordnungstabelle aktualisieren
 - Neuen Prozess auswählen
 - MMU aktualisieren mit neuer Zuordnungstabelle
 - Prozess starten
 - Cache eventuell ungültig -> erneutes Laden



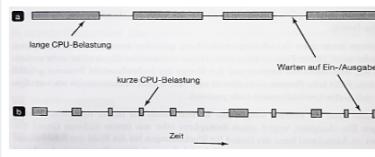
Erläutern Sie den Begriff des **Prozessverhaltens** im Zusammenhang mit dem *Scheduler*.

Scheduler (II)



Prozessverhalten

- Prozesse bestehen häufig aus einem Wechsel zwischen Rechenlast und Ein-/Auszube-anforderung, die aber häufig auch CPU-Last erzeugen (z.B. Darstellung von Bilddaten auf dem Monitor)
- Unterschiedliche Klassen von Prozessen
 - CPU-intensiv (rechenintensiv) Prozesse
 - E/A-intensiv Prozesse



Zeitpunkt

- Kernfrage: Wann soll das Scheduling beginnen?
- Fall 1: Erzeugung eines neuen Prozesses (Eltern- oder Kindprozess laufen lassen?)
- Fall 2: Prozess wird beendet (Ist ein Prozess rechenbereit? sonst `idle`)
- Fall 3: Prozess wird blockiert (Abhängigkeiten bei wartenden Prozessen)
- Fall 4: E-/A-Interrupt (Wechsel zu dem wartenden Prozess?)
- Ansatz: Verwendung eines Hardwaretimers, der zyklisch eine Entscheidung des Schedulers anfordert. Man unterscheidet hier in
 - Nicht-unterbrechende Strategie und
 - Unterbrechende Strategie

Welche Möglichkeiten gibt es *wann* das Scheduling beginnen sollte?

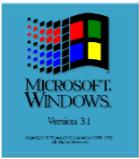
Scheduler (II)



Prozessverhalten	Zeitpunkt
<ul style="list-style-type: none"> ▪ Prozesse bestehen häufig aus einem Wechsel zwischen Rechenlast und Ein-/Ausgabe-anforderung, die aber häufig auch CPU-Last erzeugen (z.B. Darstellung von Bilddaten auf dem Monitor) ▪ Unterschiedliche Klassen von Prozessen <ul style="list-style-type: none"> ▪ CPU-intensive (rechenintensive) Prozesse ▪ E/A-intensive Prozesse 	<ul style="list-style-type: none"> ▪ Kernfrage: Wann soll das Scheduling beginnen? <p><i>Fall 1:</i> Erzeugung eines neuen Prozesses (Eltern- oder Kindprozess laufen lassen?)</p> <p><i>Fall 2:</i> Prozess wird beendet (Ist ein Prozess rechenbereit? sonst <code>idle</code>)</p> <p><i>Fall 3:</i> Prozess wird blockiert (Abhängigkeiten bei wartenden Prozessen)</p> <p><i>Fall 4:</i> E/A-Interrupt (Wechsel zu dem wartenden Prozess?)</p> <p>Ansatz: Verwendung eines Hardwaredimers, der zyklisch eine Entscheidung des Schedulers anfordert. Man unterscheidet hier in <ul style="list-style-type: none"> ▪ Nicht-unterbrechende Strategie und ▪ Unterbrechende Strategie </p>

unterbrechende vs. nicht-unterbrechende Strategien



Nicht-unterbrechendes Scheduling	Unterbrechendes Scheduling
<ul style="list-style-type: none"> ▪ Prozess wird so lange ausgeführt, bis er die CPU freiwillig wieder frei gibt (oder fertig ist) ▪ Andere Bezeichnung: <ul style="list-style-type: none"> ▪ Kooperatives Scheduling ▪ Engl.: „Run-to-completion“ → Problem: ein Prozess kann das ganze Betriebssystem lahm legen! → kaum praxisfähige Strategie (verwendet in Windows 3.1)  <p>Bildquelle: http://www.gutenberg.org/screenshots/win31</p>	<ul style="list-style-type: none"> ▪ Prozess kann vom Scheduler unterbrochen werden ▪ Andere Bezeichnung: <ul style="list-style-type: none"> ▪ Präemptives Scheduling ▪ Engl.: Preemptive Scheduling Strategie, die eigentlich von fast allen aktuellen Betriebssystemen verwendet wird <p>Wie soll ein Scheduler (Stück Software A) einen Prozess (Stück Software B) unterbrechen, wenn dieser Prozess (B) gerade auf der CPU ausgeführt wird? ... A läuft ja gerade gar nicht!</p> <p>→ Wir brauchen eine Hardware-Lösung! Hardware-Interrupts (Timer-Interrupts) → alle x ms wird ein Timer-Interrupt ausgelöst und zum Scheduler gewechselt (Kontextwechsel kostet Overhead)</p>

Grenzen Sie die *unterbrechbaren* von den *nicht-unterbrechbaren* Strategien ab.

unterbrechende vs. nicht-unterbrechende Strategien



Nicht-unterbrechendes Scheduling

- Prozess wird so lange ausgeführt, bis er die CPU freiwillig wieder frei gibt (oder fertig ist)
 - Andere Bezeichnung:
 - Kooperatives Scheduling
 - Engl.: „Run-to-completion“
- Problem: ein Prozess kann das ganze Betriebssystem lahm legen!
- kaum praxisfähige Strategie (verwendet in Windows 3.1)

Bildquelle: <http://www.guiscreensgallery.org/screenshots/win31>



Unterbrechendes Scheduling

- Prozess kann vom Scheduler unterbrochen werden
- Andere Bezeichnung:
 - Präemptives Scheduling
 - Engl.: Preemptive Scheduling

Strategie, die eigentlich von fast allen aktuellen Betriebssystemen verwendet wird

Wie soll ein Scheduler (Stück Software A) einen Prozess (Stück Software B) unterbrechen, wenn dieser Prozess (B) gerade auf der CPU ausgeführt wird? ... A läuft ja gerade gar nicht!

- Wir brauchen eine Hardware-Lösung!
Hardware-Interrupts (Timer-Interrupts)
- alle x ms wird ein Timer-Interrupt ausgelöst und zum Scheduler gewechselt
(Kontextwechsel: kostet Overhead)

Nennen Sie die beiden Möglichkeiten der Thread-Erzeugung unter Linux.
Worin liegt der Unterschied zur Erzeugung von Prozessen?

Threads unter Linux



Realisierung von Threads	Software Demo: Threads unter Linux
<ul style="list-style-type: none"> Threads werden unter Linux im Wesentlichen genauso realisiert wie Prozesse! Thread = Prozess, der den gleichen Adressraum besitzt wie sein Elternprozess ein neuer Prozess wird mit der BS Funktion <code>fork()</code> erzeugt ein neuer Thread kann mit der BS Funktion <code>clone()</code> erzeugt werden 	<ul style="list-style-type: none"> man fork <ul style="list-style-type: none"> <code>fork()</code> Source Code Beispiel (neuer Prozess wird erzeugt) man init <ul style="list-style-type: none"> <code>psTree</code> man clone <ul style="list-style-type: none"> <code>clone()</code> Source Code Beispiel (neuer Thread wird erzeugt) pthread Library Beispiel <ul style="list-style-type: none"> <code>g++ -w pthread_test.cpp -lpthread -o pthread_test</code>

Vorlesung Betriebssysteme

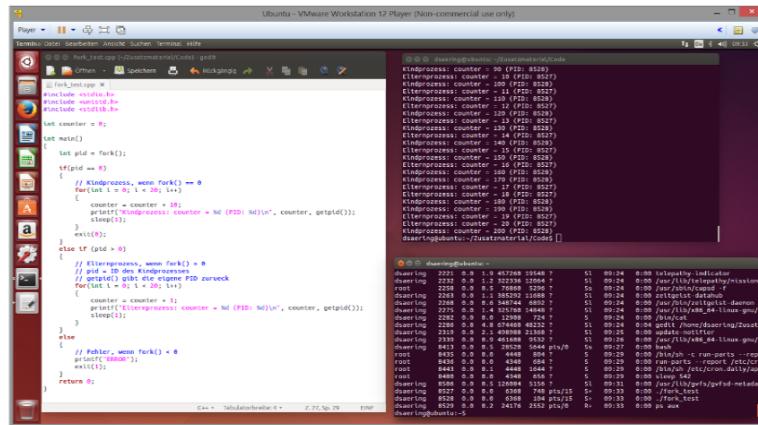
Prof. Dr. D. Säring © FH Wedel
2019

66

fork()



- gabelt vom aktuellen Prozess einen neuen Prozess ab -> Kindprozess



```

#include <iostream>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>

int counter = 0;

int main()
{
    pid_t pid = fork();
    if(pid == -1)
    {
        // Fehler, wenn Fork() == -1
        // Rufen Sie die Fehlerbehandlung
        perror("Fehler: fork()");
        exit(1);
    }
    else if(pid == 0)
    {
        // Elternteil, wenn Fork() == 0
        // Rufen Sie die Fuktionsroutine auf
        // getpid() gibt die eigene PID zurück
        printf("Elternteil: counter = %d (%d)\n", counter, getpid());
        sleep(1);
        counter++;
        printf("Elternteil: counter = %d (%d)\n", counter, getpid());
        sleep(1);
    }
    else
    {
        // Fehler, wenn Fork() > 0
        printf("Fehler: wenn Fork() > 0\n");
        exit(1);
    }
    return 0;
}

```

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

67

pstree bei fork()



- Alle Prozesse stammen von einem ersten initialen Prozess ab: init

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

68

`clone()`



- gabelt vom aktuellen Prozess einen neuen Prozess ab -> Kindprozess mit teilweise gleichen Ressourcen

Verlagung: Retrichanystems

Prof. Dr. D. Säring © FH Wedel
2010

60

pthread



- Bibliothek: POSIX Standardschnittstelle zur Erzeugung und Steuerung von Threads

The screenshot shows a Linux desktop environment with several windows:

- A terminal window titled "Ubuntu - VMware Workstation 12 Player (Non-commercial use only)" displaying the output of a "pthread" program, which prints "Hello world from thread #n" for n from 0 to 10.
- A file explorer window showing a C source file named "pthread.c". The code creates multiple threads and prints their IDs and the message "Hello world from thread #n". It includes necessary headers like `<csyslibs.h>`, `<sys/types.h>`, `<sys/conf.h>`, `<sys/thread.h>`, and `<sys/thread.h>`.
- A desktop environment window titled "dockingdock" showing various icons.

Below the screenshots, there is a note:

POSIX = Portable Operating System Interface =
Betriebssystemschnittstellenstandard bestehend aus C-Systemaufzfunktionen, Kommandozeileninterpreter und Hilfsprogrammen für „unixoide“ Betriebssysteme

Erläutern Sie die möglichen Kriterien für ein gutes **präemptives** Scheduling. Grenzen Sie dabei noch einmal die beiden wesentlichen *Scheduling-Strategien* voneinander ab.

unterbrechende vs. nicht-unterbrechende Strategien

Nicht-unterbrechendes Scheduling	Unterbrechendes Scheduling
<ul style="list-style-type: none"> ▪ Prozess wird so lange ausgeführt, bis er die CPU freiwillig wieder frei gibt (oder fertig ist) ▪ Andere Bezeichnung: <ul style="list-style-type: none"> ▪ Kooperatives Scheduling ▪ Engl.: „Run-to-completion“ <p>→ Problem: ein Prozess kann das ganze Betriebssystem lahm legen!</p> <p>→ kaum praxisfähige Strategie (verwendet in Windows 3.1)</p> <div style="text-align: center; margin-top: 10px;"> <p>Microsoft Windows Version 3.1</p> <p><small>© 1990 Microsoft Corporation. All rights reserved.</small></p> </div> <p><small>Bildquelle: http://www.gutenberg.org/screenshots/windows/</small></p>	<ul style="list-style-type: none"> ▪ Prozess kann vom Scheduler unterbrochen werden ▪ Andere Bezeichnung: <ul style="list-style-type: none"> ▪ Präemptives Scheduling ▪ Engl.: Preemptive Scheduling <p>Strategie, die eigentlich von fast allen aktuellen Betriebssystemen verwendet wird</p> <p>Wie soll ein Scheduler (Stück Software A) einen Prozess (Stück Software B) unterbrechen, wenn dieser Prozess (B) gerade auf der CPU ausgeführt wird? ... A läuft ja gerade gar nicht!</p> <p>→ Wir brauchen eine Hardware-Lösung! Hardware-Interrupts (Timer-Interrupts) → alle x ms wird ein Timer-Interrupt ausgelöst und zum Scheduler gewechselt (Kontextwechsel: kostet Overhead)</p>

Mögliche Kriterien für eine gute (präemptive) Scheduling-Strategie

<div style="background-color: #002060; color: white; padding: 5px;"> 1. Fairness </div> <ul style="list-style-type: none"> ▪ Kein Prozess/Thread sollte „unverhältnismäßig“ lange warten müssen ▪ sonst im Grenzfall: Nicht-unterbrechendes Scheduling! 	<div style="background-color: #002060; color: white; padding: 5px;"> 3. Interaktivität </div> <ul style="list-style-type: none"> ▪ Benutzeroberflächen sollten kurze Reaktionszeiten aufweisen ▪ Bsp: Videoplayer, der erst nach 5 Sek. auf Knopfdruck reagiert <div style="text-align: center; margin-top: 10px;"> </div>
<div style="background-color: #002060; color: white; padding: 5px;"> 2. Hohe Ressourcen-Auslastung </div> <ul style="list-style-type: none"> ▪ Systemressourcen wie CPU, Festplatte, etc. sollten möglichst gleichzeitig genutzt werden ▪ Bsp: <div style="text-align: center; margin-top: 10px;"> <p style="text-align: center;">vs.</p> </div>	

Mögliche Kriterien für eine gute (präemptive) Scheduling-Strategie



1. Fairness

2. Hohe Ressourcen-Auslastung

3. Interaktivität

4. Hoher Durchsatz an Prozessen

- Prozesse sollten in einer Reihenfolge bzw. so zu Unterbrechungszeitpunkten abgearbeitet werden, dass die resultierende Gesamtzeitdauer minimal ist

5. Kurze Durchlaufzeit für 1 Prozess

- Ein Prozess sollte nicht „ewig“ dauern, nur weil die resultierende Gesamtzeitdauer dann minimal ist

6. Garantierte Laufzeiten

- Echtzeitbetriebssysteme benötigen garantierte (maximale) Laufzeiten
- Bsp: Prozess der entscheidet, ob Notbremse ausgelöst werden soll



Problem: viele dieser Kriterien können im Gegensatz stehen, z.B.

(4) vs. (5): ein Prozess wird sehr oft unterbrochen um hohen Durchsatz zu erreichen
vs. wird stark „zerstückelt“; braucht lange bis zur kompletten Abarbeitung
(1) vs. (6): ein Prozess wird immer wieder hinten angestellt

→ Scheduler müssen sich für eine echte Teilmenge an zu optimierenden Kriterien entscheiden oder einen Kompromiss realisieren

Welche der Kriterien für ein gutes **präemptives** Scheduling schliessen sich gegenseitig aus? Nennen Sie zwei Beispiele.

Mögliche Kriterien für eine gute (präemptive) Scheduling-Strategie



1. Fairness

2. Hohe Ressourcen-Auslastung

3. Interaktivität

4. Hoher Durchsatz an Prozessen

- Prozesse sollten in einer Reihenfolge bzw. so zu Unterbrechungszeitpunkten abgearbeitet werden, dass die resultierende Gesamtzeitdauer minimal ist

5. Kurze Durchlaufzeit für 1 Prozess

- Ein Prozess sollte nicht „ewig“ dauern, nur weil die resultierende Gesamtzeitdauer dann minimal ist

6. Garantierte Laufzeiten

- Echtzeitbetriebssysteme benötigen garantierte (maximale) Laufzeiten
- Bsp: Prozess der entscheidet, ob Notbremse ausgelöst werden soll



Problem: viele dieser Kriterien können im Gegensatz stehen, z.B.

(4) vs. (5): ein Prozess wird sehr oft unterbrochen um hohen Durchsatz zu erreichen
vs. wird stark „zerstückelt“; braucht lange bis zur kompletten Abarbeitung
(1) vs. (6): ein Prozess wird immer wieder hinten angestellt

→ Scheduler müssen sich für eine echte Teilmenge an zu optimierenden Kriterien entscheiden oder einen Kompromiss realisieren

Erläutern Sie die Grundidee sowie die Vor- / Nachteile der **FCFS** Scheduling-Strategie. In welche Kategorie lässt sich diese einordnen? Wofür steht die Abkürzung ?missing IDENTIFIER? Nennen Sie eine Situation aus dem realen Leben die mit dieser Strategie gut verglichen werden kann.

Strategie #1 (nicht präemptiv): First Come First Served (**FCFS**)



Idee	Eigenschaften
<ul style="list-style-type: none"> Prozesse werden in Warteschlange eingereiht Prozess vom Kopf der Schlange entnehmen → Ausführen bis zu kompletter Abarbeitung → Nächster Prozess 	Vorteile: <ul style="list-style-type: none"> Kein großer Scheduling-Overhead durch häufiges Umschalten zwischen Prozessen „fair“ („wer zuerst kommt wird zuerst bearbeitet“) minimal mögliche Durchlaufzeit für einen Prozess (Prozess wird nie unterbrochen) Nachteile: <ul style="list-style-type: none"> typischerweise geringe Ressourcenauslastung typischerweise nicht interaktiv typischerweise geringer Durchsatz an Prozessen <p>→ Situation wie an der Supermarktkasse</p>

Erläutern Sie die Grundidee sowie die Vor- / Nachteile der **SJF** Scheduling-Strategie. In welche Kategorie lässt sich diese einordnen? Wofür steht die Abkürzung ?missing IDENTIFIER? Nennen Sie eine Situation aus dem realen Leben die mit dieser Strategie gut verglichen werden kann.

Strategie #2 (nicht präemptiv): Shortest Job Next (**SJN**), Shortest Job First (**SJF**)



Idee	Eigenschaften
<ul style="list-style-type: none"> Voraussetzung ist, dass es irgend eine Möglichkeit gibt, die Laufzeit eines Prozesses grob abzuschätzen (z.B. aus Erfahrungswerten) der Prozess mit der vorhergesagten kürzesten Laufzeit wird zuerst ausgeführt 	Vorteile: <ul style="list-style-type: none"> Kaum Scheduling-Overhead „fair“ („wer schnell durch ist, darf als Nächstes ran“) Nachteile: <ul style="list-style-type: none"> „unfair“ („wer zuerst kommt ist <i>nicht</i> zuerst dran“) Es kann passieren, dass ein langer Prozess nie zum Zuge kommt! (kürzere Prozesse kommen immer wieder in die Warteschlange dazu) <p>* Gewartet schwierig Prozessausdauer abzuschätzen → Situation wie an der Supermarktkasse mit freundlichem Vorlassen</p>

Vorlesung Betriebssysteme Prof. Dr. D. Säring © FH Wedel 2019 82

Erläutern Sie die Grundidee sowie die Vor- / Nachteile des **Zeitschreibeverfahrens**. In welche Kategorie lässt sich diese Strategie einordnen? Nennen Sie eine Situation aus dem realen Leben die mit dieser Strategie gut verglichen werden kann. Unter welchem *Namen* ist diese Strategie allgemeinhin noch bekannt? Nennen Sie ein Beispiel einer **Variation** dieses Verfahrens.

Strategie #3 (präemptiv): Zeitscheibenverfahren



Idee	Eigenschaften
<ul style="list-style-type: none"> ▪ Engl. Round Robin Scheduling: Spielrundenstrategie ▪ Einteilung der CPU-Zeit in kleine Zeitscheiben fester Größe / „Quantum“ (z.B. 100ms) + Warteschlange ▪ Prozess vorne in der Warteschlange darf 100ms rechnen, muss sich dann in der Warteschlange hinten wieder anreihen ▪ Variante: Gewichtete Zeitscheibengrößen (z.B. gewichtet durch Prozessprioritäten) / Engl. „Weighted Round Robin“ 	<p>Vorteile:</p> <ul style="list-style-type: none"> ▪ „fair“ („jeder darf ne Runde auf die CPU“) ▪ Weitestgehend interaktiv (aber nicht interaktiv bei z.B. 10 Rechenprozessen & 1 GUI Prozess) <p>Nachteile:</p> <ul style="list-style-type: none"> ▪ Zu kleine Zeitscheibe → viel Overhead durch viele Prozesswechsel ▪ Zu große Zeitscheibe → Interaktivität zu gering ▪ i.d.R. viel Overhead, da Zeitscheiben klein gewählt → Situation wie bei Fahrgeschäften im Freizeitpark (eine Runde fahren, dann wieder hinten in der Warteschlange anstellen)



Was versteht man im Zusammenhang des **Zeitschreibeverfahrens** unter einem *Quantum* ?missing IDENTIFIER?

Strategie #3 (präemptiv): Zeitscheibenverfahren



Idee	Eigenschaften
<ul style="list-style-type: none"> ▪ Engl. Round Robin Scheduling: Spielrundenstrategie ▪ Einteilung der CPU-Zeit in kleine Zeitscheiben fester Größe / „Quantum“ (z.B. 100ms) + Warteschlange ▪ Prozess vorne in der Warteschlange darf 100ms rechnen, muss sich dann in der Warteschlange hinten wieder anreihen ▪ Variante: Gewichtete Zeitscheibengrößen (z.B. gewichtet durch Prozessprioritäten) / Engl. „Weighted Round Robin“ 	<p>Vorteile:</p> <ul style="list-style-type: none"> ▪ „fair“ („jeder darf ne Runde auf die CPU“) ▪ Weitestgehend interaktiv (aber nicht interaktiv bei z.B. 10 Rechenprozessen & 1 GUI Prozess) <p>Nachteile:</p> <ul style="list-style-type: none"> ▪ Zu kleine Zeitscheibe → viel Overhead durch viele Prozesswechsel ▪ Zu große Zeitscheibe → Interaktivität zu gering ▪ i.d.R. viel Overhead, da Zeitscheiben klein gewählt <p>→ Situation wie bei Fahrgeschäften im Freizeitpark (eine Runde fahren, dann wieder hinten in der Warteschlange anstellen)</p>

Beispiel

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

83

Strategie #3: Zeitscheibenverfahren → Ablaufbeispiel



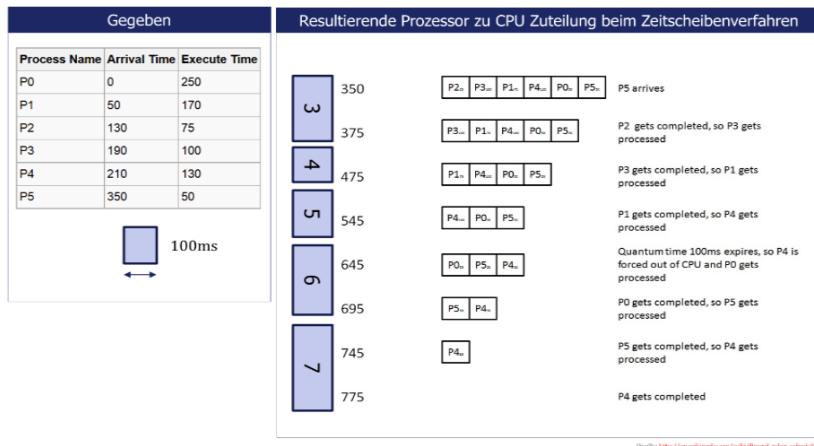
Gegeben			Resultierende Prozessor zu CPU Zuteilung beim Zeitscheibenverfahren		
<i>Quellen</i>			<i>Quellen</i>		
Process Name	Arrival Time	Execute Time	Execute Time	Round Robin Scheduling	
P0	0	250	0	P0 _{..}	P0 arrives and the gets processed
P1	50	170	50	P0 _{..} , P1 _{..}	P1 arrives and waits for quantum to expire
P2	130	75	100	P1 _{..} , P0 _{..}	Quantum time 100ms expires, so P0 is forced out of CPU and P1 gets processed
P3	190	100	130	P1 _{..} , P0 _{..} , P2 _{..}	P2 arrives
P4	210	130	190	P1 _{..} , P0 _{..} , P2 _{..} , P3 _{..}	P3 arrives
P5	350	50	200	P0 _{..} , P2 _{..} , P3 _{..} , P1 _{..}	Next 100ms expires, so P1 is forced out of CPU and P0 gets processed
100ms			210	P0 _{..} , P2 _{..} , P3 _{..} , P1 _{..} , P4 _{..}	P4 arrives
100ms			300	P2 _{..} , P3 _{..} , P1 _{..} , P4 _{..} , P0 _{..}	Next 100ms expires, so P0 is forced out of CPU and P2 gets processed

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

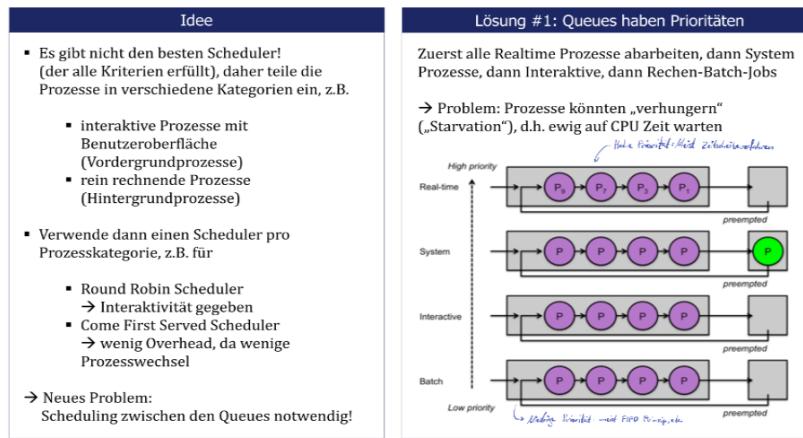
84

Strategie #3: Zeitscheibenverfahren → Ablaufbeispiel



Erläutern Sie den Begriff sowie die **Grundidee** des **MQS**, wofür steht diese Abkürzung? |missing IDENTIFIER|

Multilevel Queue Scheduling (MQS)



Welche Rolle spielt der Begriff **Feedback** in der **Multilevel Feedback Queue Scheduling** Strategie?

Multilevel Queue Scheduling (MQS)



Idee	Lösung #1: Queues haben Prioritäten
<ul style="list-style-type: none"> Es gibt nicht den besten Scheduler! (der alle Kriterien erfüllt), daher teile die Prozesse in verschiedene Kategorien ein, z.B. <ul style="list-style-type: none"> interaktive Prozesse mit Benutzeroberfläche (Vordergrundprozesse) rein rechnende Prozesse (Hintergrundprozesse) Verwende dann einen Scheduler pro Prozesskategorie, z.B. für <ul style="list-style-type: none"> Round Robin Scheduler → Interaktivität gegeben Come First Served Scheduler → wenig Overhead, da wenige Prozesswechsel <p>→ Neues Problem: Scheduling zwischen den Queues notwendig!</p>	<p>Zuerst alle Realtime Prozesse abarbeiten, dann System Prozesse, dann Interaktive, dann Rechen-Batch-Jobs</p> <p>→ Problem: Prozesse könnten „verhungern“ („Starvation“), d.h. ewig auf CPU Zeit warten</p> <p><i>Hier Priorität nach Reihenfolge aufnehmen</i></p> <p>High priority Real-time System Interactive Batch Low priority</p> <p><i>↳ Mäßige Priorität und Faireteile möglich</i></p> <p>Widelpurze: https://www.cs.vt.edu/~mik/316/notes/07_scheduling.html</p>

Multilevel Queue Scheduling (MQS)



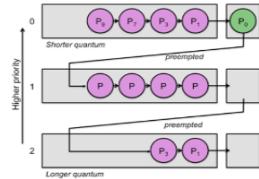
Idee	Lösung #2: Time-Slicing der Queues
<ul style="list-style-type: none"> Es gibt nicht den besten Scheduler! (der alle Kriterien erfüllt), daher teile die Prozesse in verschiedene Kategorien ein, z.B. <ul style="list-style-type: none"> interaktive Prozesse mit Benutzeroberfläche (Vordergrundprozesse) rein rechnende Prozesse (Hintergrundprozesse) Verwende dann einen Scheduler pro Prozesskategorie, z.B. für <ul style="list-style-type: none"> Round Robin Scheduler → Interaktivität gegeben Come First Served Scheduler → wenig Overhead, da wenige Prozesswechsel <p>→ Neues Problem: Scheduling zwischen den Queues notwendig!</p>	<p>Löst das „Verhungern von Prozessen“ (Starvation) Problem</p>

Multilevel **Feedback** Queue Scheduling (MLFQ)?



Idee	Wer darf als nächster in die CPU?
<ul style="list-style-type: none"> Multilevel Queue Scheduling erlaubt es nicht den Prozessoren die Warteschlange zu wechseln Multilevel Feedback Queue Scheduling schon! Prozess „frisst“ sehr viel CPU Zeit <ul style="list-style-type: none"> → er wird in eine Warteschlange mit niedrigerer Priorität geschickt → lässt interaktive Prozesse oder Prozesse die nur auf I/O warten in eine Warteschlange mit hoher Priorität Prozess der schon sehr „alt“ ist darf in eine Warteschlange höherer Priorität wechseln (verhindert das „Verhungern“ von Prozessen), d.h. Prozessalter muss verwaltet werden <p><i>↳ Prozesse können Queue aufgrund ihres Alters wechseln.</i></p>	<ul style="list-style-type: none"> Zunächst die Prozesse, die in der obersten Queue sind Ansonsten jeweils Prozesse aus der nächst tieferen Queue

MLFQ Beispiel mit 3 Queues und Zeitscheibenverfahren

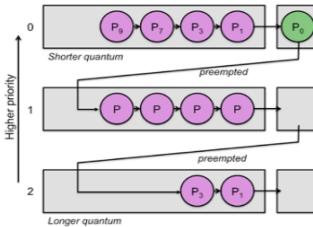


Bildquelle: http://www.cs.rug.ac.za/~omk/416/notes/07_scheduling.html

Multilevel **Feedback** Queue Scheduling (MLFQ)?



MLFQ Beispiel mit 3 Queues und Zeitscheibenverfahren
<ol style="list-style-type: none"> Alle Prozesse starten in der obersten Queue Kann der Prozess innerhalb 8ms nicht abgearbeitet werden wird er in die mittlere Queue geschoben <p>Gibt die CPU innerhalb der 8ms wieder frei, verbleibt er in der Queue (z.B. Warten auf Benutzereingabe)</p> <p>→ z.B. interaktive Prozesse die keine CPU Auslastung verursachen</p> Schafft er es auch nicht in der mittleren Queue innerhalb von 16ms abgearbeitet zu werden, wird er in die unterste Queue verschoben. <p>Gibt die CPU innerhalb von 16ms wieder frei, verbleibt er in der Queue</p> In der untersten Queue wird oft ein FCFS Scheduling angewendet <p><i>Festland & Laufzeit d. Prozesses.</i></p>



Bildquelle: http://www.cs.rug.ac.za/~omk/416/notes/07_scheduling.html

Erläutern Sie die **MLFQ**-Strategie anhand von 3 Queues und dem Zeitscheibenverfahren.

Multilevel **Feedback** Queue Scheduling (MLFQ)?



MLFQ Beispiel mit 3 Queues und Zeitscheibenverfahren

- Alle Prozesse starten in der obersten Queue
- Kann der Prozess innerhalb 8ms nicht abgearbeitet werden wird er in die mittlere Queue geschoben
Gibt er die CPU innerhalb der 8ms wieder frei, verbleibt er in der Queue (z.B. Warten auf Benutzereingabe)
→ z.B. interaktive Prozesse die keine CPU Auslastung verursachen
- Schafft er es auch nicht in der mittleren Queue innerhalb von 16ms abgearbeitet zu werden, wird er in die unterste Queue verschoben.
Gibt er die CPU innerhalb von 16ms wieder frei, verbleibt er in der Queue
- In der untersten Queue wird oft ein FCFS Scheduling angewendet

Feedback = Laufzeit d. Prozesses

Bildquelle: <http://www.cs.rutgers.edu/~pk/416/notes/07-scheduling.html>

Nennen Sie die *wichtigsten* Parameter der **MLFQ**-Strategie.

Multilevel **Feedback** Queue Scheduling: viele Parameter



Wichtige Parameter
▪ Anzahl der Queues?
▪ Scheduling Algorithmus pro Queue?
▪ Länge einer Zeitscheibe?
▪ Wann soll ein Prozess hochgestuft werden?
▪ Wann soll ein Prozess runtergestuft werden?
▪ Was soll mit Prozessen passieren, die z.B. auf I/O warten?

Quelle: Silberschatz et al., Operating System Concepts, S. 276

Beispiel: Scheduler mit 3 vs. 9 Stufen	
Prozess-Scheduler #1	Prozess-Scheduler #2
$S_1, x_1 \text{ ms}$	$S_1, x_1 \text{ ms}$
$S_2, x_2 \text{ ms}$	$S_2, x_2 \text{ ms}$
$S_3, x_3 \text{ ms}$	$S_3, x_3 \text{ ms}$
	$S_4, x_4 \text{ ms}$
	$S_5, x_5 \text{ ms}$
	$S_6, x_6 \text{ ms}$
	$S_7, x_7 \text{ ms}$
	$S_8, x_8 \text{ ms}$
	$S_9, x_9 \text{ ms}$

Scheduling Strategien:
 → <http://www.cs.rutgers.edu/~pk/416/notes/07-scheduling.html>
 → <http://www.cs.cornell.edu/~chandrima.sarkar/AdvancedOS/SchedulingOS/>
 → <http://www.csplib.org.uk/cs/ops.html>

Beschreiben Sie die **Ausgangssituation** sowie die *Kernidee* eines CFS .
Wofür steht die Abkürzung?

Completely Fair Scheduler (CFS) I



Ausgangssituation & Kernidee

- Probleme mit altem O(1) Scheduler:
viele Heuristiken und unübersichtlicher Code, um festzustellen, ob Prozess eher interaktiv („I/O bound“) oder rechnend („CPU bound“) ist
- Kernidee ist Fairness bzgl. Rechenzeit:
 - Jeder Prozess sollte gleiche Rechenzeit erhalten
 - Sobald ein oder mehrere Prozesse hinsichtlich Rechenzeit stark beteiligt sind, sollten diese die CPU wieder zugeteilt bekommen
- Zentrale Variable **vruntime** (virtuelle Laufzeit), diese misst, wie lange ein Prozess bisher gelaufen ist

Bsp: Priorität geht in Form eines Gewichtes auf mit einer λ wird mit der Prozesslauer multipliziert.
 $P_0 = 100\text{ms}$ $P_1 = 100\text{ms}$
 $\lambda = 1$ $P_0(1,5) = 50\text{ms}$ $P_1(2) = 200\text{ms}$
 Priorität

Prozess-Prioritäten

- Prozess-Prioritäten werden dann direkt auf **vruntime** über „decay“ Faktoren gerechnet (**vruntime** = tatsächliche Laufzeit * decay)
 - normale Prozesspriorität: decay = 1
-> **vruntime** = tatsächliche Laufzeit auf CPU
 - niedrige Prozesspriorität: decay > 1
-> **vruntime** > tatsächliche Laufzeit auf CPU (z.B. 300ms vs. 200ms)
 - hohe Prozesspriorität: decay < 1
-> **vruntime** < tatsächliche Laufzeit auf CPU (z.B. 200ms vs. 300 ms)
- Schließlich wählt der Scheduler den nächsten Prozess mit der kleinsten **vruntime** (löst auch das Verhungern/Starvation – Problem)

Quelle: <http://www.ibm.com/developerworks/library/l-completely-fair-scheduler/>

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

94

Wie berechnet sich die *virtuelle* Laufzeit eines CFS und wo liegt der Zusammenhang zur *Priorität* eines Prozesses?

Completely Fair Scheduler (CFS) I



Ausgangssituation & Kernidee

- Probleme mit altem O(1) Scheduler:
viele Heuristiken und unübersichtlicher Code, um festzustellen, ob Prozess eher interaktiv („I/O bound“) oder rechnend („CPU bound“) ist
- Kernidee ist Fairness bzgl. Rechenzeit:
 - Jeder Prozess sollte gleiche Rechenzeit erhalten
 - Sobald ein oder mehrere Prozesse hinsichtlich Rechenzeit stark beteiligt sind, sollten diese die CPU wieder zugeteilt bekommen
- Zentrale Variable **vruntime** (virtuelle Laufzeit), diese misst, wie lange ein Prozess bisher gelaufen ist

Bsp: Priorität geht in Form eines Gewichtes auf mit einer λ wird mit der Prozesslauer multipliziert.
 $P_0 = 100\text{ms}$ $P_1 = 100\text{ms}$
 $\lambda = 1$ $P_0(1,5) = 50\text{ms}$ $P_1(2) = 200\text{ms}$
 Priorität

Prozess-Prioritäten

- Prozess-Prioritäten werden dann direkt auf **vruntime** über „decay“ Faktoren gerechnet (**vruntime** = tatsächliche Laufzeit * decay)
 - normale Prozesspriorität: decay = 1
-> **vruntime** = tatsächliche Laufzeit auf CPU
 - niedrige Prozesspriorität: decay > 1
-> **vruntime** > tatsächliche Laufzeit auf CPU (z.B. 300ms vs. 200ms)
 - hohe Prozesspriorität: decay < 1
-> **vruntime** < tatsächliche Laufzeit auf CPU (z.B. 200ms vs. 300 ms)
- Schließlich wählt der Scheduler den nächsten Prozess mit der kleinsten **vruntime** (löst auch das Verhungern/Starvation – Problem)

Quelle: <http://www.ibm.com/developerworks/library/l-completely-fair-scheduler/>

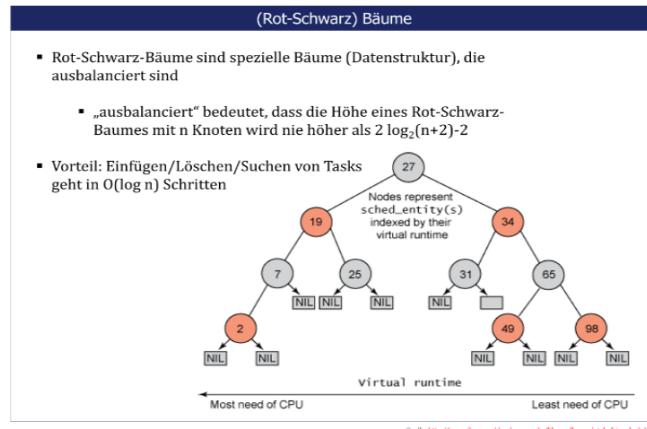
Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

94

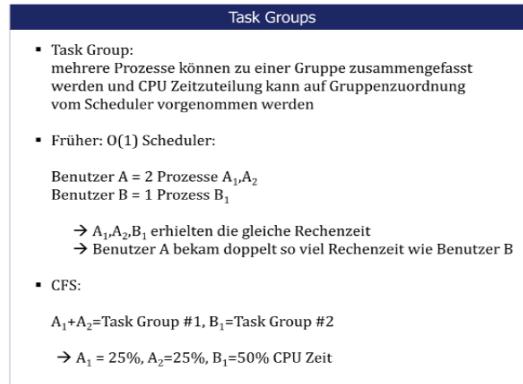
Welche **Datenstruktur** verwendet ein **CFS** um seine Daten zu speichern?

Completely Fair Scheduler (CFS) II



Erläutern Sie das Konzept einer **Task Group** im Zusammenhang mit dem **CFS**.

Completely Fair Scheduler (CFS) III



Geben Sie einen *kurzen* Überblick über den **BFS**.

BFS im Überblick



Ausgangssituation & Hauptziel

- Unzufriedenheit des ehemaligen Linux Kernel Entwicklers Con Kolivas mit dem CFS
- Beobachtungen von Kolivas:
Das CFS ist ein „Codemonster“ geworden und daher schwer zu verstehen. CFS skaliert zwar auf riesige Systeme, bietet aber schlechte Performance für kleine Desktopsysteme mit 2-4 Kernen
- Hauptziel des Schedulers: *Extrem hohe Interaktivität* für Desktopsysteme und mobile Systeme
- Wichtigster **Unterschied** zum CFS:
CFS: Ruhepause eines Prozesses mit hoher Priorität (z.B. interaktiver Prozess) beeinflusst die Größe seiner Gesamtaufzeit. Bleibt ein Prozess lange „ruhig“ (wartet auf Eingabe), bleibt die Gesamtaufzeit lange klein und er darf häufiger rechnen
BFS: entfernt solche Berechnungen und vereinfacht dadurch den Scheduler

...

- Nicht im Hauptentwicklungszweig von Linux gelandet, wird aber von einigen wenigen Linux-Distributionen als Standard Scheduler verwendet:

- Zenwalk
- PC Linux OS
- Kanotix

Quelle: <http://ckkolivas.org/patches/bfs/info.txt>

Beispiel eines Spezialkernels für Anwender, die hohen Wert auf Interaktivität / Reaktionsgeschwindigkeit von Benutzerschnittstellen legen

→ Details auf GitHubseite zu einem Linux Scheduler: http://cgit.github.com/torvalds/linux/tree/scheduler/2013/12/linux_scheduler_notes.html.pdf

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

98

Beschreiben Sie grob was man unter dem sog. **Nebenläufigkeitsproblem** versteht. Erläutern Sie dies anhand eines Beispiels.

Moderne BS ermöglichen echte und quasi-Parallelität



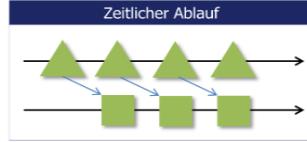
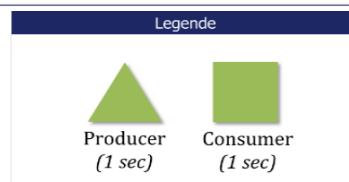
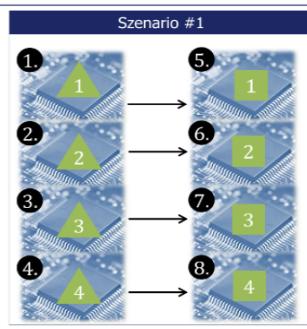
Alles gut?

- Prozesse und Threads können auf Mehrkern-CPUs gleichzeitig laufen (*Multiprocessing*) und/oder quasi-gleichzeitig (*Multitasking*) über das Zeitscheibenverfahren
- Moderne Betriebssysteme nutzen beides! (sonst bei reinem Multiprocessing: Anzahl parallel laufender Prozesse auf Anzahl der Kerne beschränkt)
 - eine Datei?
 - Variablen?

Man spricht vom **Nebenläufigkeitsproblem**

Das Nebenläufigkeitsproblem tritt in der Praxis der Softwareentwicklung z.B. beim *Producer/Consumer Pattern* auf

Verarbeitung der Sensordaten („Heile-Welt Szenario“)

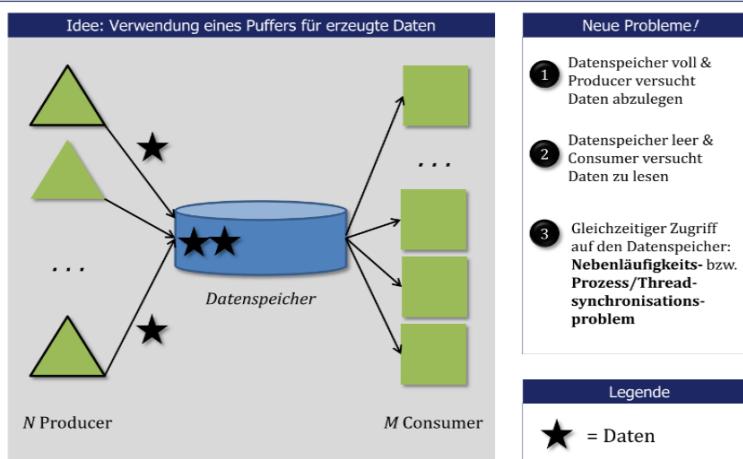


Verarbeitung der Sensordaten (Realistische Situation)



Lösung:

Keine direkte Kopplung zwischen Producer → Consumer



pstree bei fork()



- Alle Prozesse stammen von einem ersten initialen Prozess ab: init

A screenshot of a Linux terminal window showing the output of the command 'ps aux'. The tree diagram illustrates the ancestry of various processes. The root process is 'init' (PID 1). Other processes shown include 'gnome-session' (which has a red box around it), 'gnome-terminal' (another node in the tree), 'bash', 'fork', and 'test'. The entire tree is highlighted with a red box.

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel

68

clone()



- gabelt vom aktuellen Prozess einen neuen Prozess ab -> Kindprozess mit teilweise gleichen Ressourcen

A screenshot of a Linux terminal window showing the output of the command 'ps aux'. It displays a list of processes including 'doxygen', 'dhclient', 'gdm', 'gnome-terminal', 'libpulse', 'pulseaudio', 'qemu-system-i386', 'qemu-gui', 'qemu-kvm', 'qemu-user', 'qemu-x86_64', and 'root'. A red box highlights the 'root' process. Below the terminal, there is a code editor showing C++ code for a 'clone' function.

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

69

pthread



- Bibliothek: POSIX Standardschnittstelle zur Erzeugung und Steuerung von Threads

The screenshot shows a Linux desktop environment with a terminal window and a file browser window.

The terminal window displays the output of a "Hello World" program using threads, showing multiple threads (0 to 6) printing "Hello world from thread #<threadid>".

```
Ubuntu - VMware Workstation 12 Player (Non-commercial use only)
Terminal: /usr/bin/ls -l /home/dsaebring/Desktop/ -F
Terminal: ./pthread_testapp -L/home/dsaebring/Desktop/Code5 -o
Hello world From thread #0
Hello world From thread #1
Hello world From thread #2
Hello world From thread #3
Hello world From thread #4
Hello world From thread #5
Hello world From thread #6
Hello world From thread #0
Hello world From thread #1
Hello world From thread #2
Hello world From thread #3
Hello world From thread #4
Hello world From thread #5
Hello world From thread #6
Hello world From thread #0
Hello world From thread #1
Hello world From thread #2
Hello world From thread #3
Hello world From thread #4
Hello world From thread #5
Hello world From thread #6
Hello world From thread #0
Hello world From thread #1
Hello world From thread #2
Hello world From thread #3
Hello world From thread #4
Hello world From thread #5
Hello world From thread #6
```

The file browser window shows the source code for "pthread_testapp.c". The code uses the `pthread` library to create multiple threads and print "Hello world" from each thread.

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/thread.h>
#include <sys/error.h>
#include <sys/malloc.h>
#include <sys/malloc.h>
#include <sys/thread.h>
#include <sys/error.h>
#include <sys/malloc.h>
```

Below the terminal window, a note reads:

POSIX = Portable Operating System Interface =
Betriebssystemschnittstellenstandard bestehend aus C-Systemaufzugsfunktionen, Kommandozeilenelementen und Hilfsprogrammen für „unixoide“ Betriebssysteme

Charakterisieren Sie die drei wesentlichen Probleme der **Nebenläufigkeit**.

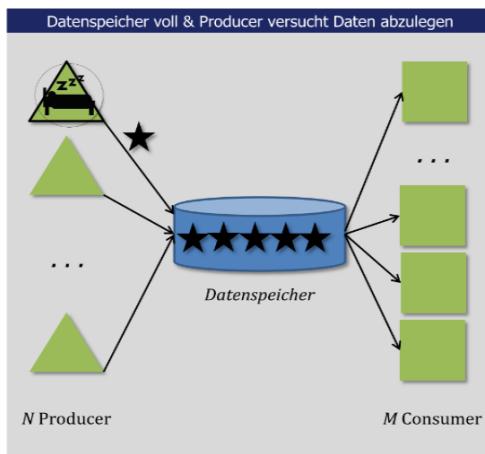
Lösungsvorschläge



Neue Probleme!	
1	Datenspeicher voll & Producer versucht Daten abzulegen
2	Datenspeicher leer & Consumer versucht Daten zu lesen
3	Gleichzeitiger Zugriff auf den Datenspeicher: Nebenläufigkeits- bzw. Prozess/Thread-synchronisationsproblem

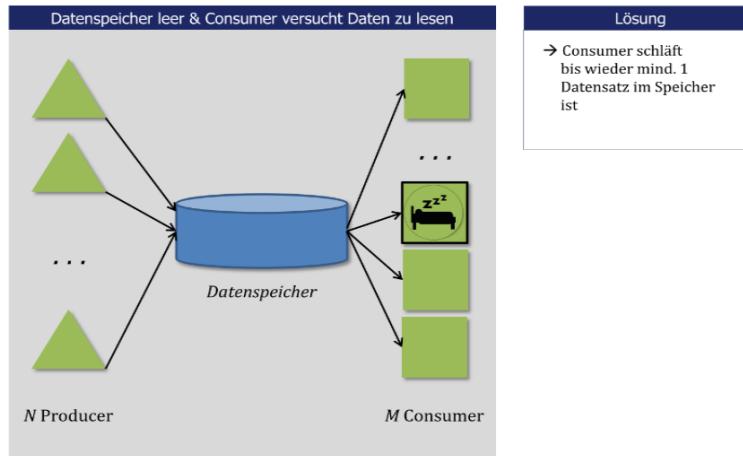


Lösung für Teilproblem 1



Lösung	
→ Producer schläft bis wieder Platz im Datenspeicher ist	

Lösung für Teilproblem 2

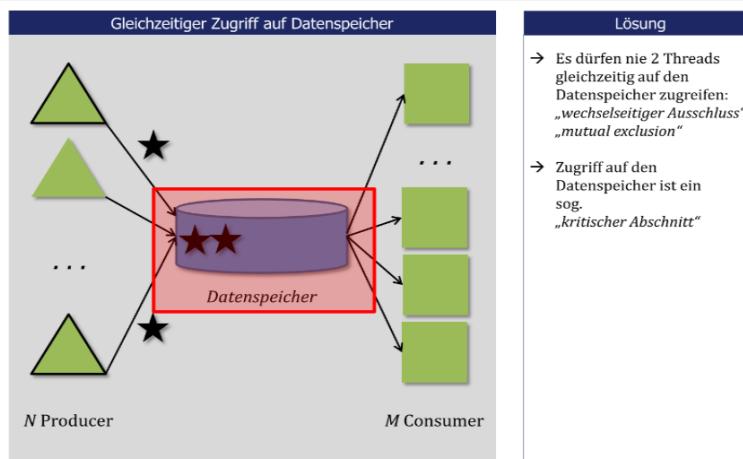


Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

115

Lösung für Teilproblem 3



Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

116

Erläutern Sie was man unter einem sog. *kritischem Abschnitt* versteht. Nennen Sie den *Fachbegriff* des Lösungsansatzes und beschreiben Sie die 4 **Bedingungen** für eine *gute* Lösung.

Kritischer Abschnitt



Einführung

- Ein **kritischer Abschnitt** (kritische Region) sind gemeinsam genutzte Bereiche (z.B. Speicher oder Dateien)
- Es muss vermieden werden, dass diese Bereiche von zwei Prozessen gleichzeitig darauf zugreifen (Beispiel: Druckerspooler)
- *Lösung:* Es muss ein **wechselseitiger Ausschluss** (mutual exclusion) für Prozesse umgesetzt werden
- Für eine gute Lösung müssen 4 Bedingungen eingehalten werden:
 - Keine zwei Prozesse dürfen gleichzeitig in ihrem kritischen Regionen sein
 - Es dürfen keine Annahmen über Geschwindigkeit und Anzahl der CPUs gemacht werden
 - Kein Prozess, der außerhalb seiner kritischen Region läuft, darf einen anderen Prozess blockieren
 - Kein Prozess sollte ewig darauf warten müssen, in seine kritische Region einzutreten

• Threads teilen sich Speicher

• Prozesse nicht



© 2010 Kenny Smith - FLURRITASDUCK.NET

Erläutern Sie wieso es das **abschalten** von Interrupts zum Steuern eines *kritischen Abschnitts* nicht als optimaler Lösungsansatz angesehen wird. Wo kann dieser Ansatz jedoch sinnvoll sein?

Wechselseitiger Ausschluss: Lösungsmöglichkeiten (I)



(1) Interrupts ausschalten

- Einfachste Lösung auf einem Einprozessorsystem: **Ausschalten der Interrupts**, nach dem Eintreten in einen kritischen Bereich
 - Kein CPU Wechsel mehr möglich
-> kein anderer Prozess kann während Speicheränderungen Zugriff erhalten
 - Kein guter Ansatz, der Prozess könnte das System dauerhaft blockieren
 - Im Multiprozessorsystem würde der 2. Prozessor noch arbeiten und unerwünschter Speicherzugriff wäre möglich
 - Generell aber eine nützliche Technik, wenn u.a. Listen aktualisiert werden sollen -> allerdings wenig geeignet für wechselseitigen Ausschluss

(2) Sperren über Variablen

- Softwarelösung: Nutzung von gemeinsamen Sperrvariablen -> if(Sperre == 0)
 - Ein Prozess kann nur Eintreten, wenn die Sperre == 0 ist, nach dem Eintreten setzt dieser Sperre auf 1
 - Problem: Was passiert bei einem Interrupt zum falschen Zeitpunkt?

Ein Blick auf unser Beispiel ...

Erläutern Sie die *Strategie des Sperrens von Variablen* zum regeln des Zugriffs auf einen kritischen Abschnitt.

Wechselseitiger Ausschluss: Lösungsmöglichkeiten (I)



(1) Interrupts ausschalten

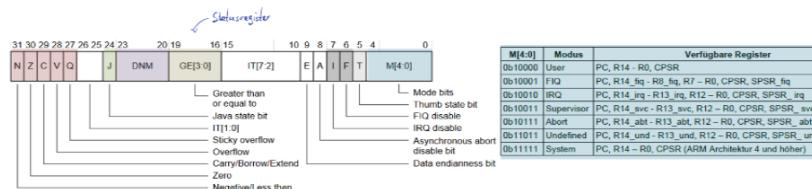
- Einfachste Lösung auf einem Einprozessorsystem: **Ausschalten der Interrupts**, nach dem Eintreten in einen kritischen Bereich
 - Kein CPU Wechsel mehr möglich
-> kein anderer Prozess kann während Speicheränderungen Zugriff erhalten
 - Kein guter Ansatz, der Prozess könnte das System dauerhaft blockieren
 - Im Multiprozessorsystem würde der 2. Prozessor noch arbeiten und unerwünschter Speicherzugriff wäre möglich
 - Generell aber eine nützliche Technik, wenn u.a. Listen aktualisiert werden sollen -> allerdings wenig geeignet für wechselseitigen Ausschluss

(2) Sperren über Variablen

- Softwarelösung: Nutzung von gemeinsamen Sperrvariablen -> if(Sperre == 0)
 - Ein Prozess kann nur Eintreten, wenn die Sperre == 0 ist, nach dem Eintreten setzt dieser Sperre auf 1
 - Problem: Was passiert bei einem Interrupt zum falschen Zeitpunkt?

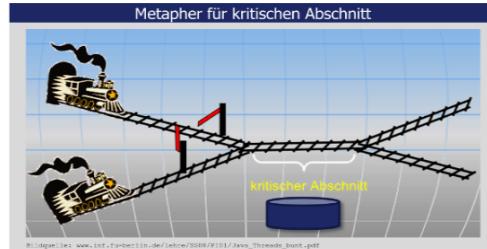
Ein Blick auf unser Beispiel ...

Interrupts deaktivieren: Program Status Register (ARM Cortex-A)



[31...27] Condition code flags N: Ergebnis einer Operation ist negativ Z: Ergebnis einer Operation ist 0 C: Überlauf bei Addition oder Subtraktion & Zuletzt geschobener Wert bei Schiebeoperation V: „overflow“ Q: „sticky overflow“ (nur bei E-Variante, DSP)	[19...16] SIMD instructions GE[3:0]: A op B ≥ C
[24] Jazelle J: Flag für die Verwendung von Java Code	[7...6] Execute Interrupts I: Ermöglicht das Unterbinden von IRQs F: Ermöglicht das Unterbinden von FIIRQs
[23...20] DNM Do Not Modify!	[5] Thumb mode T: Thumb- (=1) oder ARM-Instruktionen (=0)
	[4...0] Processor mode

Wechselseitiger Ausschluss: Naiver Ansatz (I)

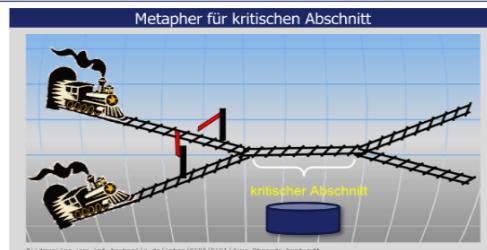


So klappt es nicht!

Thread A <i>(Scheduler könnte im falschen Moment unterbrechen.)</i> <pre>if (Sperre == 0) { Sperre = 1; kritischer_Abschnitt(); Sperre = 0; } else warte(); ...</pre>	Thread B <pre>if (Sperre == 0) { Sperre = 1; kritischer_Abschnitt(); Sperre = 0; } else warte(); ...</pre>
--	--

Das *Kleingedruckte* : Scheduler könnte im falschen Moment unterbrechen.

Wechselseitiger Ausschluss: Naiver Ansatz (I)



So klappt es nicht!

Thread A <i>(Scheduler könnte im falschen Moment unterbrechen.)</i> <pre>if (Sperre == 0) { Sperre = 1; kritischer_Abschnitt(); Sperre = 0; } else warte(); ...</pre>	Thread B <pre>if (Sperre == 0) { Sperre = 1; kritischer_Abschnitt(); Sperre = 0; } else warte(); ...</pre>
--	--

Beschreiben Sie die *Lösung von Peterson* zum regeln des Zugriffs auf einen kritischen Abschnitt. Auf wie viele *Threads* kann dieser Ansatz angewendet werden? Beschreiben Sie neben den Vorteilen auch die Nachteile dieses Ansatzes.

Wechselseitiger Ausschluss: Lösungsmöglichkeiten (II)



(3) Lösung von Peterson

- Peterson entwickelte 1981 einen einfachen Weg, um wechselseitigen Ausschluss zu realisieren
- Der Algorithmus besteht aus zwei Prozeduren
 - `enter_region(procID)`
 - `leave_region(procID)`
- Jeder Prozess ruft vor dem Betreten eines kritischen Bereichs `enter_region()` mit seiner Prozessnummer auf
- Der Prozess ruft `leave_region(procID)` mit seiner Prozessnummer auf, sobald er den Bereich verlässt

Was ist bei N>2 Threads?

- Man kann den Lösungsansatz von Peterson für 2 Threads verallgemeinern
- Lösung für N>2 Threads wird „Lamports Bäckerei Algorithmus“ genannt

Was ist der Nachteil von Petersons Lösung?

Busy Waiting!

Schauen wir uns hierfür einmal eine Implementierung in JAVA an ...

Demo: Petersons Lösung (I)



Java Source Code zur Synchronisierung von 2 Threads

```
package petersonslosung;

public class MyThread extends Thread {

    static int        turn;          // gibt an, wer gerade an der Reihe ist
    static boolean[] interested = new boolean[2]; // spezifiziert, ob ein Thread Interesse am
                                                // kritischen Abschnitt hat
    static int kritischerDatenbereich; // da sollten nicht 2 Threads gleichzeitig
                                       // schreiben
    int id;                         // ID des Threads

    public MyThread(int _id) {
        id = _id;
    }

    public void enter_region(int _id) {
        int other = 1 - _id; // Nummer des anderen Threads
        interested[_id] = true; // Interesse am kritischen Abschnitt bekunden
        turn = other; // Höflich dem anderen Thread den Vortritt lassen
        while (interested[other] == true && turn == other) { // Warten
            try { Thread.sleep( 10 ); }
            catch (Exception e) { System.out.println(e); }
        }
    }
}
```

Demo: Petersons Lösung (II)



Java Source Code zur Synchronisierung von 2 Threads

```

public void leave_region(int _id) {
    interested[_id] = false;
}

@Override
public void run() {
    for (int i=1; i<=100; i++) {
        enter_region( id );
        System.out.println( "\nThread " + id + " betritt kritischen Abschnitt");
        System.out.println( "Thread " + id + " verlässt kritischen Abschnitt\n");
        leave_region( id );
    }
}

public static void main(String[] args) {
    kritischerDatenbereich = 0;
    MyThread t0 = new MyThread( 0 );
    MyThread t1 = new MyThread( 1 );
    t0.start();
    t1.start();
}
}

```

Wieso klappt der wechselseitige Ausschluss?



Zentrale Rolle spielt die Methode enter_region

```

public void enter_region(int _id) {
    int other = 1 - _id; // Nummer des anderen Threads
    interested[_id] = true; // Interesse am kritischen Abschnitt bekunden
    turn = other; // Höflich dem anderen Thread den Vortritt lassen.
    while (interested[other] == true && turn == other) { // Warten
        try { Thread.sleep( 10 ); }
        catch (Exception e) { System.out.println(e); }
    }
}

```

1. Fall: ein Thread möchte kritischen Abschnitt betreten

- z.B. Thread #0
 - enter_region(0) wird aufgerufen, enter_region(1) nicht
 - other = 1-0 = 1
 - interested[0] = true
 - turn = other = 1
 - die While-Schleifenbedingung ist nicht erfüllt, da interested[1] = false
- Thread #0 darf enter_region() Methode wieder verlassen und im kritischen Abschnitt weitermachen



Wieso klappt der wechselseitige Ausschluss?

Zentrale Rolle spielt die Methode `enter_region`

```
public void enter_region(int _id) {
    int other = 1 - _id; // Nummer des anderen Threads
    interested[_id] = true; // Interesse am kritischen Abschnitt bekunden
    turn = other; // Höflich dem anderen Thread den Vortritt lassen.
    // Diese Variable gibt es nur 1x! Da steht 0 ODER 1 drin
    while (interested[other] == true && turn == other) { // Warten
        try { Thread.sleep( 10 ); }
        catch (Exception e) { System.out.println(e); }
    }
}
```

2. Fall: beide Threads möchten kritischen Abschnitt betreten

- `enter_region(0)` wird aufgerufen, `enter_region(1)` auch
- z.B. Thread #1
- `other = 1-1 = 0`
- `interested[1] = true`
- `turn = other = 0`
- wenn Thread #1 zuletzt `turn = 0` setzt, wartet Thread #1 bis Thread #0 fertig ist (signalisiert durch `interested[0] = false`)

Die Thread-Kommunikation ist hier also durch gemeinsam genutzten Speicher (`interested[2]` sowie `turn` Variable) realisiert

Richtig oder Falsch: Die Abfolge von *Programmabläufen* unterschiedlicher Prozesse kann immer durch deren Reihenfolge im Source-Code *reproduzierbar* festgelegt werden?

Falsch: Durch die Zuordnung der Prozesse zur CPU und Prozesse mit höherer Priorität sind zeitliche Programmabläufe selten reproduzierbar.

Beschreiben Sie die Aufgaben eines **Prozesskontrollblocks** (PCB) und geben sie an wo dieser abgespeichert wird.

PCB ist eine Datenstruktur, die genutzt wird, um assoziierte Daten für einen Prozess zu speichern. PCBs werden in der Prozesstabelle des BS gespeichert.

Nennen und beschreiben Sie die drei Zustände eines Prozesses gemäss dem Standardschema für das Zustandsmodell.

- Bereit - Rechnend - Blockiert

Richtig oder Falsch: Ein Thread hat einen eigenen Stack und eigene CPU Register, Threads eines Prozesses teilen sich aber den gleichen Speicher innerhalb dieses Prozesses.

Richtig!

Was wird mit dem UNIX-Befehl `fork` erzeugt?

Eine exakte Kopie des Elternprozesses.

Was versteht man im Zusammenhang mit der Verarbeitung eines **kritischen Abschnitts** unter *TSL-Operationen*?

Wechselseitiger Ausschluss: Lösungsmöglichkeiten (III)



(4) Test-and-Lock Operationen	...
<ul style="list-style-type: none"> ▪ haben den Aufbau TSL REGISTER, LOCK ▪ liest den Inhalt der Speicheradresse LOCK in ein Register (für die weitere Überprüfung) ▪ speichert an der Speicheradresse LOCK den Wert 1 ▪ das Lesen der Speicheradresse LOCK und das Schreiben passiert hier atomar! ▪ dazu sperrt die CPU den Speicherbus für kurze Zeit, bis sie diese beiden Operationen ausgeführt hat 	<ul style="list-style-type: none"> ▪ Was wenn Thread A@CPU1 und Thread B@CPU2 <i>gleichzeitig</i> versuchen das TSL Kommando auszuführen? Geht nicht, immer nur eine CPU bzw. Kern erhält Zugriff auf den Speicherbus (der Speicherbus ist auch eine gemeinsam genutzte Ressource) ▪ Wichtiger Unterschied zur „Unterbrechungssperre“: die Unterbrechungssperre auf CPU1 würde nicht dafür sorgen, dass Threads auf CPU2 auch versuchen auf den Inhalt der Speicheradresse LOCK zuzugreifen

Atomare Lese/Schreib-Operationen auf Hardware Ebene



Wie kann so eine Test-and-Lock Operation jetzt zur Thread-Synchronisation verwendet werden?

<pre> enter_region: TSL REGiSTER,LOCK CMP REGiSTER,#0 JNE enter_region RET </pre>	<pre> j copy lock to register and set lock to 1 j was lock zero? j if it was nonzero, lock was set, so loop j return to caller; critical region entered </pre>
<pre> leave_region: MOVE LOCK,#0 RET </pre>	<pre> j store a 0 in lock j return to caller </pre>

- jeder Thread, der einen kritischen Abschnitt betreten möchte, ruft `enter_region` auf
- wenn `LOCK = 0` → der Thread darf zurückkehren und den kritischen Abschnitt betreten
- wenn `LOCK = 1` → der Thread muss warten, d.h. `LOCK` erneut überprüfen (Rücksprung zu `enter_region`)
- wenn der Thread, der den kritischen Abschnitt besitzt fertig ist, setzt er `LOCK` einfach auf 0 → signalisiert, dass der kritische Abschnitt jetzt wieder frei ist

Atomare Lese/Schreib-Operationen auf Hardware Ebene



Bei Intel x86 Prozessoren: XCHG Kommando

```
enter_region:
    MOVE REGISTERS
    XCHG REGISTER,LOCK
    CMP REGISTER,#0
    JNE enter.region
    RET
                                | put a 1 in the register
                                | swap the contents of the register and lock variable
                                | was lock zero?
                                | if it was non zero, lock was set, so loop
                                | return to caller; critical region entered
```

```
leave_region:
    MOVE LOCK,#0
    RET
                                | store a 0 in lock
                                | return to caller
```

- Hier: XCHG Kommando ist atomar
→ d.h. atomarer Vertausch der Werte in einem Register und einer Speicherzelle
- Ansonsten bleibt die Lösung gleich:
- Überprüfung ob $LOCK = 0$, d.h. kritischer Abschnitt noch frei
→ Thread verlässt **enter_region** und setzt Ausführung im kritischen Abschnitt fort
- $LOCK = 1$ → Thread muss warten bis $LOCK = 0$

Erläutern Sie die Lösungsmöglichkeit für den *wechselseitigen Ausschluss* mittels einer **Semaphore**. Welche *Operationen* stellt dieses Verfahren zur Verfügung?

Wechselseitiger Ausschluss: Lösungsmöglichkeiten (IV)



(5) Semaphor

- Dijkstra schlug 1965 vor eine ganzzahlige Variable zu benutzen, den sogenannten **Semaphor**
- Der Semaphor gibt an, ob aktuell Weckrufe (und wie viele) gespeichert sind
- Änderungen des Wertes sind über zwei Operationen möglich (`down` und `up`)

`up`-Operation

- `up`-Operation erhöht den Wert des Semaphors um eins
- Ist `semaphor == n` wird der Prozess sofort schlafen gelegt und `up` nicht vollständig ausgeführt
- Überprüfung, Veränderung und „Schlafenlegen“ müssen als atomare Aktionen umgesetzt werden

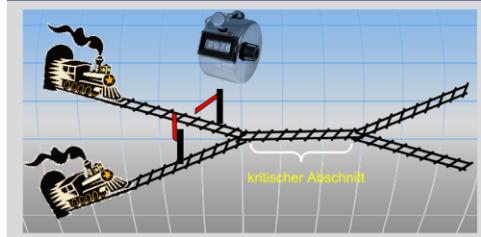
`down`-Operation

- `down`-Operation prüft auf `semaphor > 0` und vermindert den Wert um eins
 - Ist `semaphor == 0` wird der Prozess sofort schlafen gelegt und `down` nicht vollständig ausgeführt
 - Überprüfung, Veränderung und „Schlafenlegen“ müssen als atomare Aktionen umgesetzt werden

Realisierung des wechselseitigen Ausschlusses: Semaphore



Metapher für kritischen Abschnitt + Semaphor

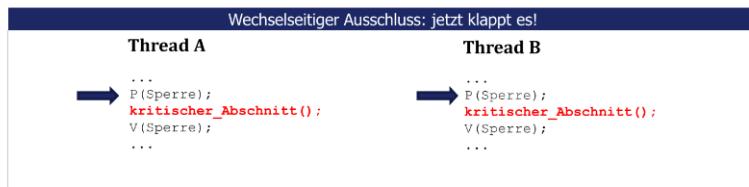
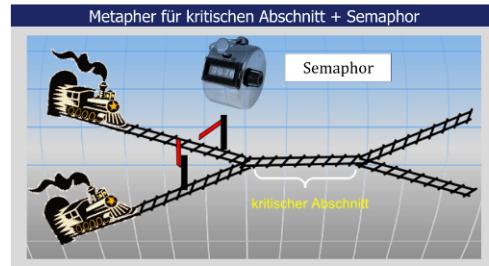


Semaphor

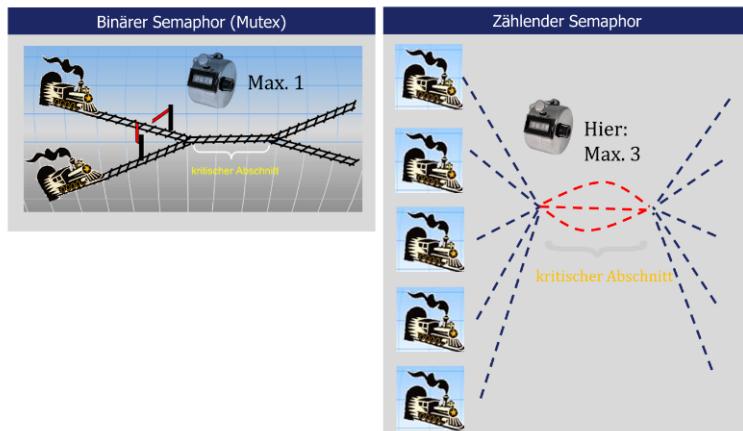


- Zähler: S
- Atomare Probieren / Reservieren Operation: `P()`, `down()`
 $S = S - 1;$
- Atomare Freigeben Operation: `V()`, `up()`
 $S = S + 1;$
- Warteschlange

Realisierung des wechselseitigen Ausschlusses: Semaphore



Binäre und zählende Semaphoren



Erläutern Sie die *verschiedenen* Möglichkeiten zur Realisierung von **atomaren** Operationen. Beschreiben Sie in diesem Zusammenhang noch kurz den Unterschied zwischen der sog. *Quasi-Parallelität* sowie der *echten Parallelität*.

Möglichkeiten der Realisierung von atomaren Operationen



1. Fall: nur ein Rechenkern (Quasi-Parallelität)

```
→ ...  
P(S1) ;  
kritischer_Abschnitt() ;  
V(S1) ;  
...
```



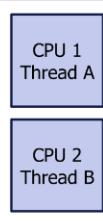
Wie realisiert das Betriebssystem, dass die Operationen P() und V() nicht unterbrochen werden, d.h. **atomar** sind?

Lösung: Unterbrechungssperre

Unteilbarkeit von P() und V() kann durch Sperren aller Unterbrechungen während der Ausführung von P() und V() erreicht werden. Zulässig, da nur wenige Maschineninstruktionen zur Implementierung nötig sind.

2. Fall: >=2 Rechenkerne (echte Parallelität)

```
→ ...  
P(S1) ;  
kritischer_Abschnitt() ;  
V(S1) ;  
...
```



Wie realisiert das Betriebssystem, dass die Operationen P() und V() nicht unterbrochen werden, d.h. **atomar** sind?

Hierzu gibt es **auf den Prozessoren** spezielle Befehle, die eine Variable im Speicher in einem nicht unterbrechbaren Schritt

- in ein Register lesen
- und gleichzeitig auf einen best. Wert setzen können

sog. **Test-and-Set-Lock** Operationen

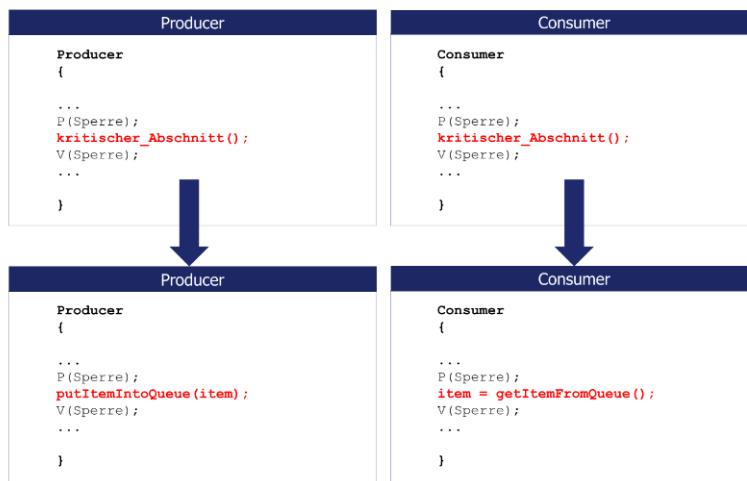
Beschreiben Sie den zeitlichen Ablauf des *wechselseiten Ausschlusses* unter Verwendung einer **Semaphore** anhand eines Beispiels.

Wechselseitiger Ausschluss: Lösungsmöglichkeiten (IV)



(5) Semaphore	up-Operation
<ul style="list-style-type: none"> Dijkstra schlug 1965 vor eine ganzzahlige Variable zu benutzen, den sogenannten Semaphor Der Semaphor gibt an, ob aktuell Weckrufe (und wie viele) gespeichert sind Änderungen des Wertes sind über zwei Operationen möglich (down und up) 	<ul style="list-style-type: none"> up-Operation erhöht den Wert des Semaphors um eins Ist <code>semaphor == n</code> wird der Prozess sofort schlafen gelegt und up nicht vollständig ausgeführt Überprüfung, Veränderung und „Schlafenlegen“ müssen als atomare Aktionen umgesetzt werden
down-Operation	
<ul style="list-style-type: none"> down-Operation prüft auf <code>semaphor > 0</code> und vermindert den Wert um eins <ul style="list-style-type: none"> Ist <code>semaphor == 0</code> wird der Prozess sofort schlafen gelegt und down nicht vollständig ausgeführt Überprüfung, Veränderung und „Schlafenlegen“ müssen als atomare Aktionen umgesetzt werden 	

Producer/Consumer Pattern mit Semaphoren



Producer/Consumer Pattern mit Semaphoren (3. Semaphor)

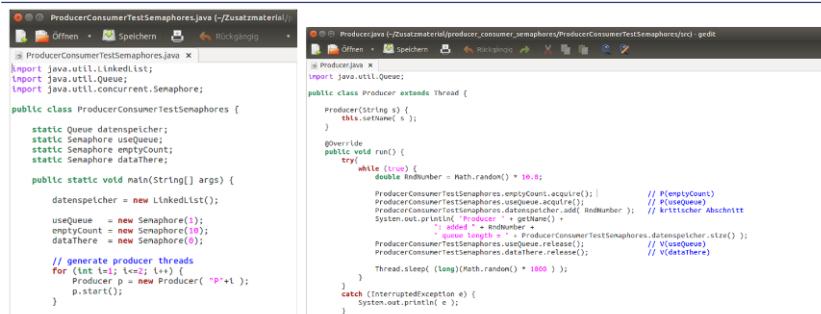


Producer	Consumer
<pre> Producer { ... P(emptyCount); P(useQueue); putItemIntoQueue(item); V(useQueue); V(dataThere); ... } </pre>	<pre> Consumer { ... P(dataThere); P(useQueue); item = getItemFromQueue(); V(useQueue); V(emptyCount); ... } </pre>

Benötigte Semaphoren			
Semaphor-Name	Start-wert	Bedeutung	Lösung für
useQueue	1	Lässt Producer/Consumer nur dann weiter, wenn kein anderer auf Speicher zugreift	3
emptyCount	N	Lässt Producer nur dann weiter, wenn noch Platz im Speicher ist	1
dataThere	0	Lässt Consumer nur dann weiter, wenn mind. 1 Datenitem im Speicher ist	2

Demo: Threadsynchronisation mittels Semaphore in Java



			
Semaphor-Name	Start-wert	Bedeutung	Lösung
useQueue	1	Lässt Producer/Consumer nur dann weiter, wenn kein anderer auf Speicher zugreift	3
emptyCount	N	Lässt Producer nur dann weiter, wenn noch Platz im Speicher ist	1
dataThere	0	Lässt Consumer nur dann weiter, wenn mind. 1 Datenitem im Speicher ist	2

Demo: Threadsynchronisation mittels Semaphore in Java



```

  Consumer.java (~/Zusatzmaterial/producer_consumer_semaphores/ProducerConsumerTestSemaphores/src-) -edit
  ☐ Offnen ☐ Speichern ☐ Rückgängig ☐
  ☐ Consumer.java x
  import java.util.Queue;
  public class Consumer extends Thread {
    Consumer(String s) {
      this.s = s;
    }
    @Override
    public void run() {
      try {
        while (true) {
          ProducerConsumerTestSemaphores.dataThere.acquire(); // P(dataThere)
          ProducerConsumerTestSemaphores.useQueue.acquire(); // P(useQueue)
          double d = (double) (Math.random() * 100);
          ProducerConsumerTestSemaphores.datenspeicher.remove(); // kritischer Abschnitt
          System.out.println("Consumer " + getUsername() +
            " : get " + d);
          ProducerConsumerTestSemaphores.useQueue.release(); // V(useQueue)
          ProducerConsumerTestSemaphores.emptyCount.release(); // V(emptyCount)
          Thread.sleep((Long)(Math.random() * 1000));
        }
      } catch (InterruptedException e) {
        System.out.println(e);
      }
    }
  }

```

P() und V() müssen atomare Funktionen sein, d.h. diese dürfen nicht vom Betriebssystem unterbrochen werden, bevor sie nicht vollständig abgeschlossen wurde.

Semaphor-Name	Start-wert	Bedeutung	Lösung
useQueue	1	Lässt Producer/Consumer nur dann weiter, wenn kein anderer auf Speicher zugreift	3
emptyCount	N	Lässt Producer nur dann weiter, wenn noch Platz im Speicher ist	1
dataThere	0	Lässt Consumer nur dann weiter, wenn mind. 1 Datenitem im Speicher ist	2

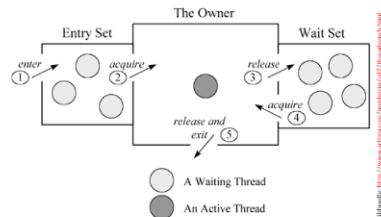
Beschreiben Sie das Konzept des sog. **Monitors**. Wofür ist es geeignet und welches bekannte Verfahren soll es ersetzen? Gehen Sie auch auf die Nachteile von Monitoren ein.

Alternative zu Semaphoren: Monitore



Monitor allgemein

- Programmiersprachenkonzept zur Prozesssynchronisierung (bereits zur Compilierung bekannt) -> Compiler realisiert intern den wechselseitigen Ausschluss
- Monitor ist eine Sammlung von Prozeduren, Variablen und Datenstrukturen
- Häufig wird ein Mutex für den Monitor verwendet, so dass garantiert nur ein Prozess im Monitor aktiv ist -> nur ein Prozess im kritischen Abschnitt
- Vorteil für Programmierer: Interne Realisierung wird vom Compiler übernommen, man muss lediglich sicherstellen kritische Regionen mit Monitorprozeduren zu belegen
- Zudem Hilfe für die Threadkooperation mit Zustandsvariablen:
`wait()`: Prozess wird schlafend gelegt und ein anderer Prozess darf den Monitor betreten
`signal()`: weckt einen schlafenden Prozess



Monitor in Java

Jedes Objekt in Java hat Monitor-Fähigkeiten!

```
public class MeinDatenspeicher {
    public synchronized void addItem(...) {
        ...
    }

    public synchronized int getItem(...) {
        ...
    }
}
```

Schlussfolgerung



Worin liegt das Problem?

- Semaphore sind sehr systemnah, da diese bei Multi-Prozessorsystemen auf gemeinsam genutzten Speicher und die Verwendung von TSL-Befehlen (bzw. XCHG-Befehlen) aufbauen
- **Problem:** Verteilte Systeme mit privatem Speicher und verbunden über ein Netzwerk
- Monitore sind als Programmierkonzept in vielen Programmiersprachen nicht einsetzbar (C oder C++), da der Compiler nicht die nötigen Funktionen zur Verfügung hat einen Monitor zu realisieren
- **Problem:** Nicht in allen Programmiersprachen umsetz- und einsetzbar

Wie könnte man es besser machen?

- Prozesse könnten etwas von der Existenz anderer Prozesse wissen
- Prozesse könnten „kommunizieren“, d.h. gegenseitig darüber informieren, ob der jeweilige Prozess überhaupt gerade interessiert ist, den kritischen Abschnitt zu betreten
- bei gleichzeitigem Interesse Absprache treffen, welcher Prozess den kritischen Abschnitt betreten darf
- Lösung:
Nachrichtenaustausch (messaging passing) unter Verwendung von zwei Primitiven
 - `send(destination, &message)`
 - `receive(source, &message)`

2 Generelles

Betriebssysteme::Lernkontrolle