

1 Generelles

Charakterisieren Sie das BIOS und nennen Sie die wichtigsten Aufgaben.
Wofuer steht die Abkuerzung ueberhaupt?

Basisbetriebssystem



Ausgaben von einem 1. Betriebssystem, dem *BIOS*



Beispiel eines BIOS ROM Chips



Quelle: http://www.techbeamung.com/reviews/boards/Z77_Extreme11/1/MC

Aufgaben des BIOS (Basic Input Output System)

- POST = Power On Self Test → Suche nach und Zugriff auf wichtigsten Hardware-Komponenten (Tastatur, Arbeitsspeicher, Grafikkarte, Festplatten, CD-Laufwerk) und evtl. Test (Arbeitsspeicher)
- Sequentialle Überprüfung aller gefundener Laufwerke, ob Sie einen Bootsektor (MBR = Master Boot Record) enthalten
- Konfiguration der Hardware (BIOS-Menü), z.B. Prozessor- oder Arbeitsspeicher-Taktfrequenz

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



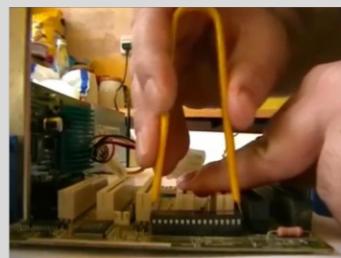
Früher vs. Heute

- 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=3k8kjgj0frY>

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?

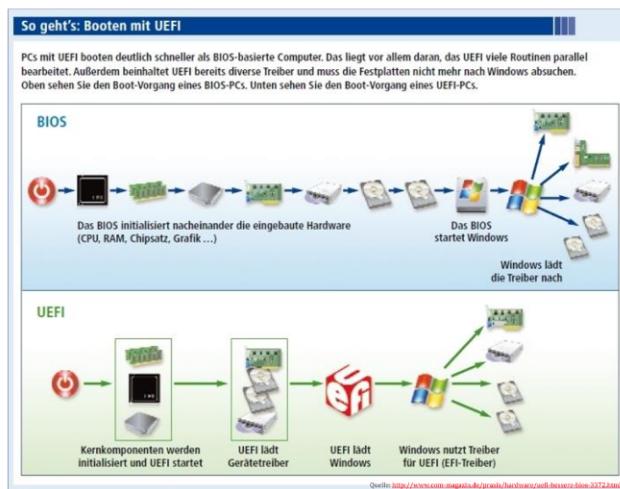
Unified Extensible Firmware Interface (UEFI)



Woher kommt es?	Vorteile
Unified Extensible Firmware Interface Vereinheitlichte Erweiterbare Firmware*-Schnittstelle  <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: Hackangriffe möglich</small> ▪ UEFI-Shell: Eigene Kommandozeile ▪ Bootmanager zur Auswahl des zu startenden BS ist schon integriert

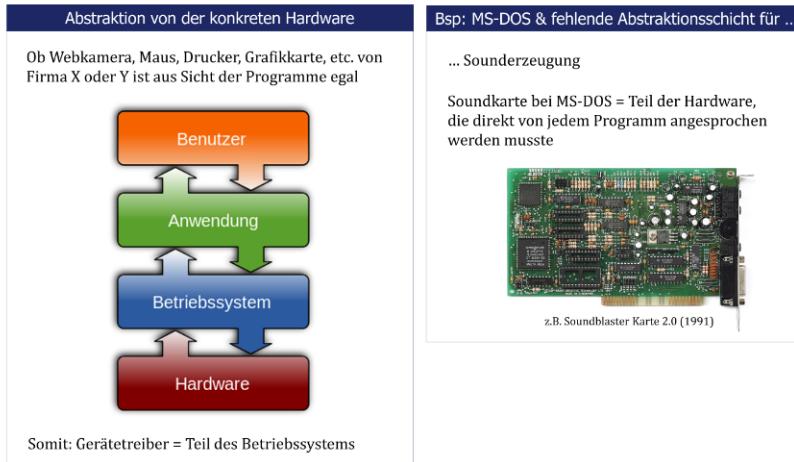
Skizzieren Sie den **Bootvorgang** mittels UEFI und grenzen Sie diesen von dem mittels des klassischen BIOS ab.

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



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

3. Dienstleistung des Betriebssystem



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!

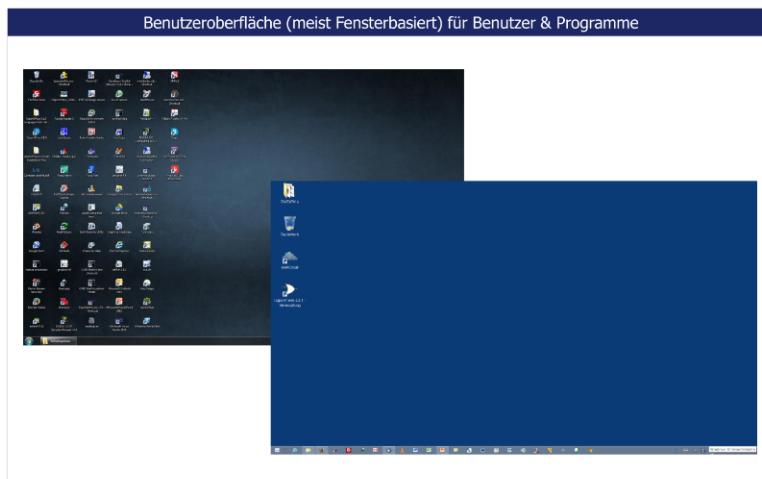
Vorlesung Betriebssysteme

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

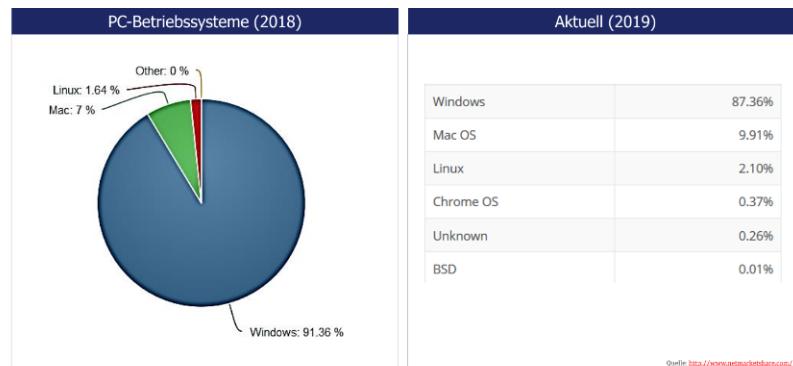


3. Dienstleistung des Betriebssystem

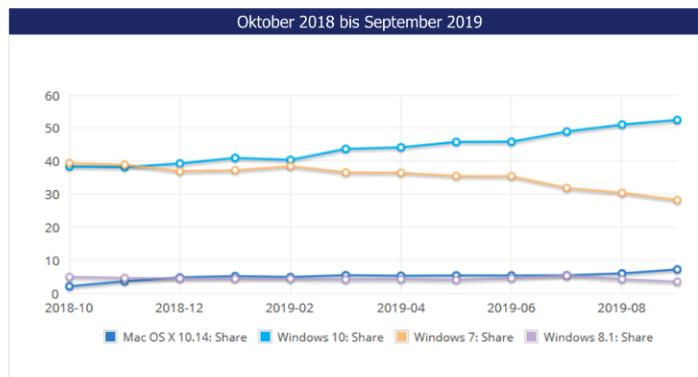


Beschreiben Sie grob die Marktanteile der PC-Betriebssysteme.

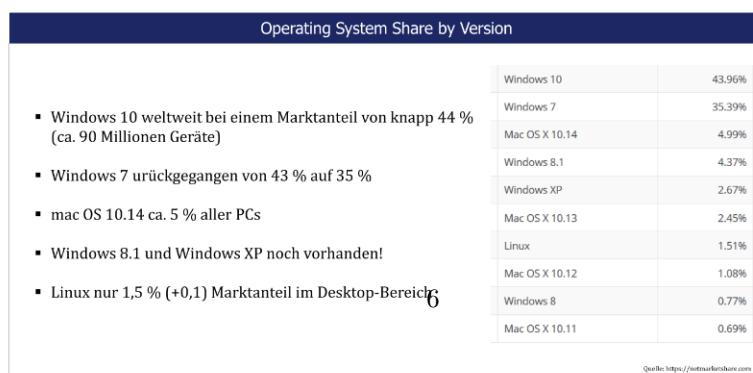
PC-Betriebssysteme



Marktanteile der führenden Betriebssysteme I

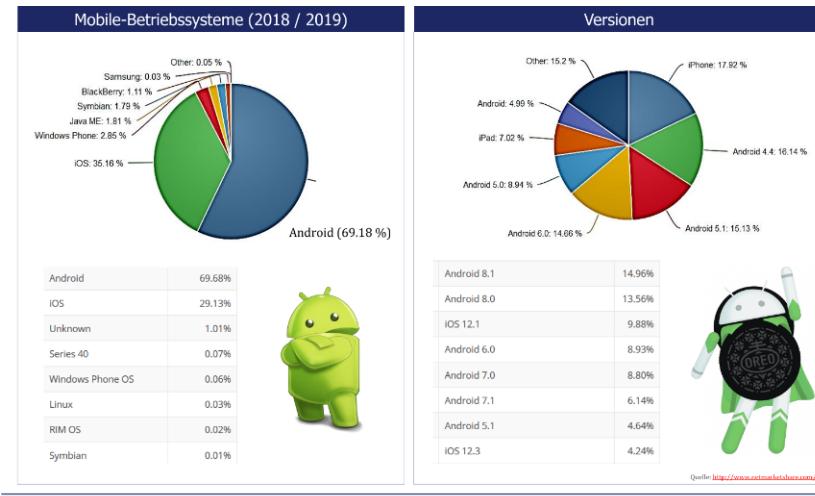


Marktanteile der führenden Betriebssysteme II



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

Mobile-Betriebssysteme



Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

26

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“?



Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

22

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

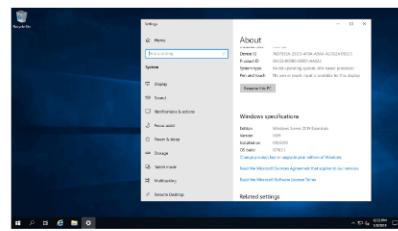
Server-Betriebssysteme



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: <https://www.server-magazin.de>

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 kompakter 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.autogesetz.de/medien/mediendatenbank/das-schweizer-autos-des-jahres-2011-dor-peugeot-308-tecnisch>
Flugzeug: http://de.wikipedia.org/w/index.php?title=Boeing_747&oldid=621179211
* nach einer Idee von Friedhelm Stappert / Hochschule Darmstadt

Echtzeitbetriebssysteme (II)

Beispiele von Echtzeitbetriebssystemen



RTLinux



Quellen:
RTLinux: <http://www.rtlinux.org/> (2010/12/getting-started-with-rtlinux)
QNX: <http://devkit.qnx.com/qnx/qnx.html>
Windows Embedded: <http://www.microsoft.com/windows/embedded/linx/en/developers.aspx>
Wie funktioniert RTLinux? → <http://www.opensourcefor.com/2010/12/getting-started-with-rtlinux/>

Fassen Sie die verschiedenen Arten von Betriebssystemen zusammen.

Vielfalt der Betriebssysteme



Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel

37

Weitere Betriebssysteme



Router-BS



Quelle: <http://wellimanded.webs.com/2011/12/29/you-can-fine-tune-a-computer-and-so-on-the-main-router/>

SmartCard (Prozessorkarten vs. Speicherchipkarten) BS

BasicCard, CombOS, CardOS, JCOP, MTCOS, MultOS, SECCOS, Sicrypt, STARCOS, TCOS, etc.



Zum Beispiel Speichern eines Private Keys auf der SmartCard und direkte Verschlüsselung von Daten auf der Karte mittels des Karten-Prozessors

Quelle: <http://www.kiosken.info/definition/technik/Smartcard-smartcard.html>

Supercomputer BS

Cray Linux Environment (CLE), früher "UNICOS" skaliert auf bis zu mehr als 500.000 Prozessor-Kerne



Quelle: <http://www.cray.com/products/Computing/XE.aspx>

Verteilte Betriebssysteme

d.h. Virtualisierung von verteilten Ressourcen wie Speicher, Prozesse, etc.
z.B. LinuxPMI = Linux Process Migration Infrastructure



Quelle: <http://lxswx.org/trac/>

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

Diagramm: Architekturdiagramm der Betriebssystemarchitektur. Es zeigt die hierarchische Struktur von Anwendungsprogrammen über die User Space bis zum Kernel Space, wo der Linux-Kernel (Prozesse, Speicher, Dateisystem, Gerätetreiber, Netzwerkstack...) sowie das System Call Interface liegt. Die User Space enthält unter anderem die C-Bibliothek (libc, eglibc, uclibc...) und andere Bibliotheken (OpenGL, Webkit...).

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

42

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	
Amiga Exec	AmigaOS	Commodore International	HUNK	Exokernel (atypical) ^[2]	
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, Orbis OS	The FreeBSD Project	ELF, others - platform dependent	monolithic	
GNU/Hurd (Arch)	GNU				

Diagramm: Klassifizierung von Betriebssystemkernen. Ein roter Pfeil weist auf den FreeBSD-Kernellinie hin, der als monolithisch klassifiziert wird. Ein anderer roter Pfeil weist auf den FreeBSD-Kernellinie hin, der als hybrid klassifiziert wird.

SunOS kernel	SunOS	Sun Microsystems	a.out	monolithic
Solaris kernel	Solaris, OpenSolaris, GNU/OpenSolaris (Nexenta OS)	Sun Microsystems	ELF (32-bit only until Solaris 7 in 1998)	monolithic
Trix kernel	Trix, GNU/Trix	Massachusetts Institute of Technology	a.out	monolithic
Windows NT kernel [12] ^[2]	Windows NT, 2000, XP, 2003, Vista, Windows 7, Windows 8	Microsoft	PE, others?	hybrid
XNU (Darwin kernel) [13] ^[2]	Mac OS X, iOS, OpenDarwin, PureDarwin GNU/Darwin	Apple Computer	Mach-O	hybrid
SPARTAN kernel [14] ^[2]	HelenOS	Jakub Jermar	ELF	microkernel

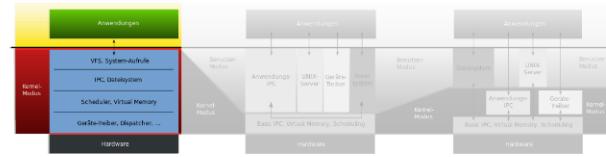
Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

43

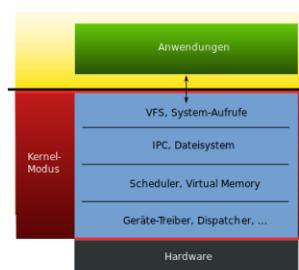
Beschreiben Sie die *Philosophie*, *Nachteile* und die *Vorteile* eines monolithischen Kernel BS .

Die 3 Hauptarten von BS-Kernel



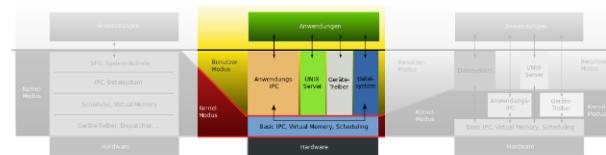
Monolithische Kernel BS

- **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 Einschränkungen wegen Trafik zwischen einz. Komponenten
- **Beispiele:** Unix und Linux



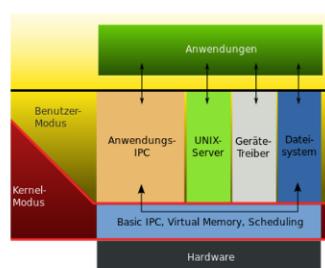
Beschreiben Sie die *Philosophie*, *Nachteile* und die *Vorteile* eines Mikrokernels BS .

Die 3 Hauptarten von BS-Kernel



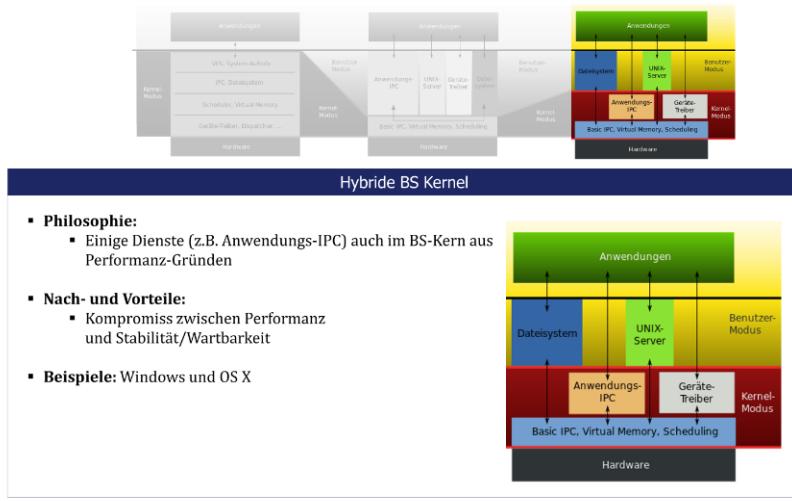
Mikrokernels BS

- **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 Mikrokern)



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

Die 3 Hauptarten von BS-Kernel



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) [2]	
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, GentooFreeBSD, Orbis OS	The FreeBSD Project	ELF, others - platform dependent	monolithic	
GNU/Hurd (Arch)	GNU				

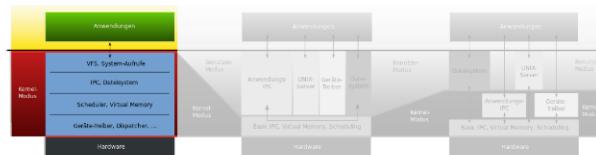
Quelle: http://mankelpa.org/wikia/Comparisons_of_operating_system_kernels

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

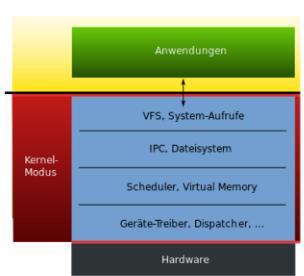
43

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 Traffic zwischen einer Komponenten
- **Beispiele:** Unix und Linux

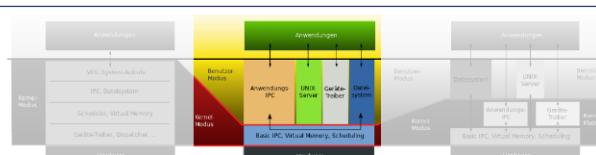


Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

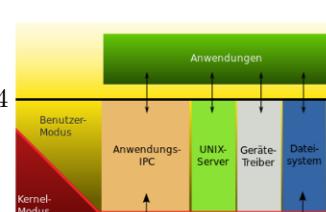
44

Die 3 Hauptarten von BS-Kernel



Mikrokernell 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



14

Kernell Modus

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

Wiederholung (I)



Bildquelle: <http://www.okdokumente.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



Betriebssysteme::Lernkontrolle

Nennen Sie die wesentlichen Dienstleistungen eines Betriebssystems

Wiederholung (I)



Bildquelle: <http://www.okdokumente.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)



Wikipedia: <http://www.wikipedia-potsdam.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
 - Mikrokernel 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)



Wikipedia: <http://www.wikipedia-potsdam.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
 - Mikrokernel 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;



Prozessmodell

Aspekt der Laufzeit

- 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)
 - Spezialfall: Echtzeitbetriebssysteme



Quelle: die-besten-100.de

Allgemein

- 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
- *Anmerkung:* Ein Programm, das zweimal ausgeführt wird läuft in unterschiedlichen Prozessen ab

... mal schauen wie ein Prozessmodel umgesetzt wird!

Prozessmodell

Aspekt der Laufzeit

- 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)
 - Spezialfall: Echtzeitbetriebssysteme



Quelle: die-besten-100.de

Allgemein

- 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
- *Anmerkung:* Ein Programm, das zweimal ausgeführt wird läuft in unterschiedlichen Prozessen ab

... mal schauen wie ein Prozessmodel umgesetzt wird!

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

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
Allgemein
<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



Quelle: die-besten-100.de

... mal schauen wie ein Prozessmodell umgesetzt wird!

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

54

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
Allgemein
<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



Quelle: die-besten-100.de

... 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)
<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> <p>htop Kommando</p> <pre>Tasks: 122, 169 thr: 5 running CPU: [██████████] 100% user [██████████] 100% system Mem: 11.8GB total, 10.9GB used, 0.9GB free Swap: 1.0GB total, 0.0GB used, 1.0GB free PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ Command 3296 dseiring 20 0 318M 3456 1008 S 0.3 0.1 0:00.01 /usr/bin/python -m cgitb --help 4947 dseiring 20 0 30360 29728 21288 S 3.0 3.4 0:01.25 gnome-terminal 779 root 20 0 288M 6564 6564 S 1.2 0.8 0:00.45 /usr/lib/policykit-1/polkitd --no-daemon 3432 dseiring 20 0 274M 274M 274M S 0.0 0.0 0:00.00 /usr/libexec/gdm-xserver-demon -daemonize 4518 dseiring 20 0 742M 37328 28532 S 0.0 3.7 0:07.40 nautilus -n 469 messenger 20 0 40136 3724 2744 S 0.0 0.3 0:01.11 dbus-daemon -system --fork 796 udev 20 0 100M 100M 100M S 0.0 0.0 0:00.00 udevd -d -n -k -u -b /var/run/udev/mktd -no-daemon 3137 root 20 0 233M 6476 6396 S 0.0 0.6 0:00.17 /usr/lib/powerd /upowerd 4133 dseiring 20 0 1182M 79824 34956 S 0.0 0.0 0:21.49 compiz 4125 dseiring 20 0 1182M 11108 10984 S 0.0 0.0 0:00.00 /usr/libexec/gdm-xserver-dmd -n vncr 3829 dseiring 20 0 347M 10444 10444 S 0.0 1.0 0:00.119 /usr/bin/dbus-daemon -daemonize 3876 dseiring 20 0 477M 24976 19444 S 0.0 2.5 0:00.52 /usr/lib/dbus/dbus-ut-gtk3 3888 dseiring 20 0 139M 4768 4556 S 0.0 0.5 0:00.23 /usr/bin/vtcsd 776 root 20 0 348M 8484 7752 S 0.0 0.5 0:00.00 NetworkManager 3352 dseiring 20 0 122M 2812 2812 S 0.0 0.0 0:00.00 /usr/libexec/gdm-xserver-dbus-engine-simple 3887 dseiring 20 0 477M 24976 19444 S 0.0 2.5 0:00.18 /usr/lib/dbus/dbus-ut-gtk3 3352 dseiring 20 0 122M 6112 5904 S 0.0 0.0 0:00.01 thermald -no-daemon --dbus-enable 4526 dseiring 20 0 122M 7048 6936 S 0.0 0.0 0:00.00 /usr/libexec/gdm-xserver-dbus-engine-simple 3326 root 20 0 122M 6112 5904 S 0.0 0.0 0:00.19 thermald -no-daemon --dbus-enable 3927 dseiring 20 0 13648 4146 4146 S 0.0 0.0 0:00.00 /sbin/init 1 root 20 0 13648 4146 4146 S 0.0 0.0 0:00.00 /sbin/init 378 root 20 0 19480 1788 1788 S 0.0 0.2 0:00.78 upstart-udev-bridge - daemon 588 root 20 0 10380 2636 2636 S 0.0 0.3 0:00.03 /usr/lib/systemd/udevd -d -a 495 root 20 0 10380 2636 2636 S 0.0 0.3 0:00.03 /usr/lib/systemd/bluetoothd F1 1111 F2 1111 F3 1111 F4 1111 F5 1111 F6 1111 F7 1111 F8 1111 F9 1111 F10 1111</pre>

Vorlesung Betriebssysteme

Prof. Dr. D. Sören © FH Wedel
2019

55

Was versteht man unter einem *Prozesskontrollblock*?missing IDENTIFIER_i

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 <ul style="list-style-type: none"> 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 aufgerufenenFunktionen) ▪ File Handles ▪ CPU Register 	

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 #include <linux/types.h> 1281 #include <linux/sched.h> 1282 struct task_struct { 1283 volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */ 1284 void *stack; 1285 atomic_t nr_wakeups; 1286 unsigned long flags; /* per process flags, defined below */ 1287 unsigned int priority; 1288 }; 1289 #ifdef CONFIG_SMP 1290 struct list_head wake_wake_q; 1291 int on_rq; 1292 atomic_t nr_wakeups; 1293 unsigned long wake_flip; 1294 unsigned long wake_flip_decay_ts; 1295 int wake_cqpu; 1296 int on_rq_prio; 1297 int preemptible; 1298 int prio; 1299 static_prio, normal_prio; 1300 unsigned int nr_migrate醒; 1301 const struct schedClass *class; 1302 struct sched_entity se; 1303 struct list_head runq; 1304 struct list_head sleepq; 1305 #endif CONFIG_CGROUP_SCHED 1306 struct task_group *task_group; 1307 struct sched_task_group *sched_task_group; 1308 struct sched_dl_entity dl; 1309 #ifdef CONFIG_PREEMPT_NOTIFIERS 1310 /* list of preemptible task_group_notifiers */ 1311 struct list_head preempt_notifiers; 1312 struct list_head preempt_notify_list; 1313 #endif 1314 #ifdef CONFIG_RCU_BOOST_THREADS 1315 unsigned int brtrace_seq; 1316 unsigned int brtrace_ip; 1317 #endif 1318 unsigned int polling; 1319 int nr_wake_allowed; 1320 spinlock_t cgroup_allowed; 1321 }</pre>	<pre>1675 int curr_ret_stack; 1676 /* Return stack addresses for return function tracing */ 1677 struct trace_stack *ret_stack; 1678 /* time stamp for the scheduler */ 1679 unsigned long long frametime; 1680 /* Number of functions that haven't been traced 1681 * because of depth overrun, 1682 */ 1683 atomic_t trace_overrun; 1684 /* Pause for the tracing */ 1685 atomic_t trace_stopped; 1686 /* Trace graph pause */ 1687 #endif 1688 #endif /* CONFIG_SCHED_DEBUG */ 1689 /* state flags for use by tracer */ 1690 #define _SCHED_TRACE(flags) ((flags) & ~_SCHED_TRACE) 1691 #define _SCHED_TRACE_RECURSION(flags) ((flags) & ~_SCHED_TRACE_RECURSION) 1692 #define _SCHED_TRACE_DEPTH(flags) ((flags) & ~_SCHED_TRACE_DEPTH) 1693 #endif /* CONFIG_SCHED_DEBUG */ 1694 #ifndef CONFIG_HZ_1000 1695 struct seq_struct_info { 1696 struct seq_group *seqg; 1697 gfp_t gfp_mask; 1698 unsigned int max_order; 1699 } memcg_oom; 1700 #endif 1701 #endif 1702 #endif /* CONFIG_UPROBES */ 1703 #endif 1704 #endif 1705 #if defined(CONFIG_BORCHIE) 1706 unsigned int sequential_lo; 1707 unsigned int sequential_lo_wq; 1708 #endif 1709 #endif /* CONFIG_BORCHIE */ 1710 #endif /* CONFIG_BORCHIE_MODULE */ 1711 #endif 1712 #endif 1713 /* Future-safe accessor for struct task_struct's cgroup_allowed. */ 1714 #define task_cgroup_allowed(tsk) ((tsk)->cgroup_allowed) 1715 #define task_cgroup_allowedtksk(tsk) ((tsk)->cgroup_allowed) 1716 1717 #define TM_HIGH_PRIORITY 0x01</pre>

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel

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?

Vorlesung Betriebssysteme

Prof. Dr. D. Siring © FH Wedel
2019

57

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

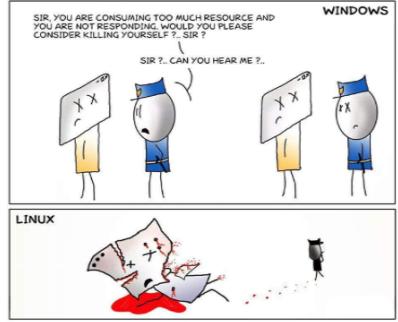
Prozessbeendigung



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. wenn da selbst ein Programm]
- **Freiwilliges Beenden** z.B. nach vollständiger Erledigung 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 anderer Prozesses (`kill / terminateProcess`)

HANDLING NON-RESPONDING & FROZEN APPLICATIONS



Vorlesung Betriebssysteme

Prof. Dr. D. Siring © FH Wedel
2019

58

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 einen 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>system</code> ▪ Windows: Kein Konzept für eine Hierarchie, alle Prozesse sind gleichwertig.

... welchen Zustand können Prozesse haben?

Vorlesung Betriebssysteme

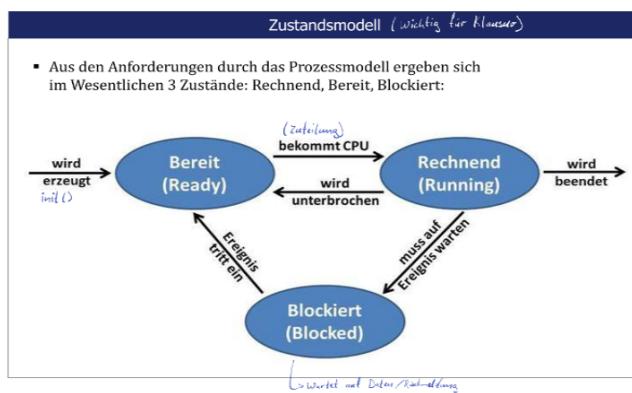
Prof. Dr. D. Säring © FH Wedel

2019

59

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

Prozesszustand (Standardschema)



Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

60

Erläutern Sie anhand eines einfachen Beispiels die Sinnhaftigkeit von Threads.

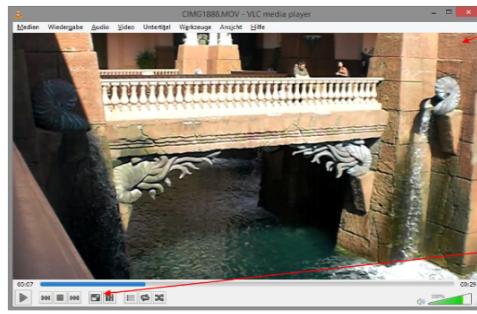


Threads

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!

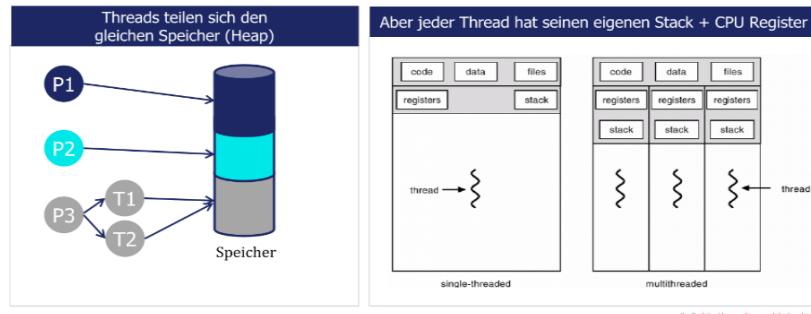


1. Video abspielen

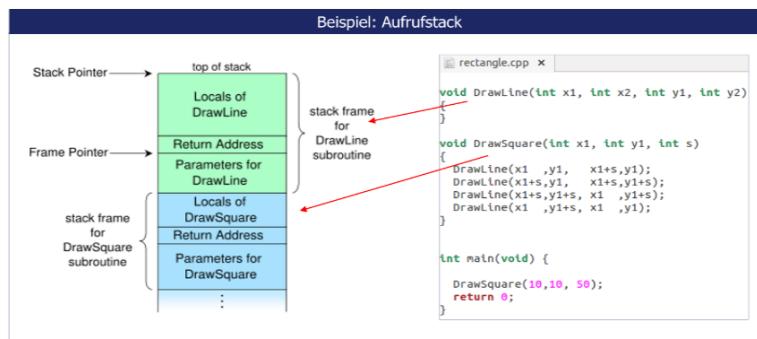
2. Auf Steuerelemente reagieren

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

Unterschied Prozesse vs. Threads („leichtgewichtige Prozesse“)



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	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

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

67

pstree bei fork()



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

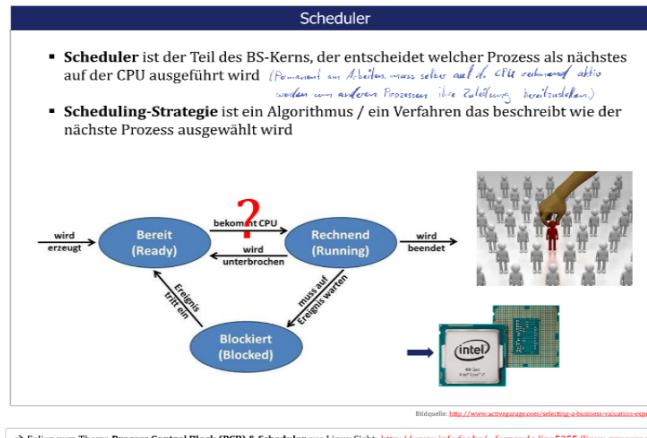
10

Prof. Dr. D. Säring © FH Wedel

1

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

Prozessverwaltung mittels Scheduler



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.

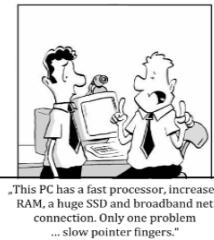
Auf welchen Systemen ist ein **Scheduler** von besonderer Bedeutung? Gehen Sie hierbei insbesondere auf die Gründe ein.

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.)
 - Speicherzuordnungstabellen aktualisieren
 - Neuen Prozess auswählen
 - MMU aktualisieren mit neuer Zuordnungstabelle
 - Prozess starten
 - Cache eventuell ungültig -> erneutes Laden



Vorlesung Betriebssysteme

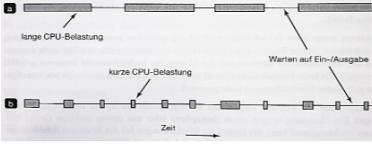
Prof. Dr. D. Säring © FH Wedel
2019

73

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

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> <ul style="list-style-type: none"> ▪ Ansatz: Verwendung eines Hardwaretimers, der zyklisch eine Entscheidung des Schedulers anfordert. Man unterscheidet hier in <ul style="list-style-type: none"> ▪ 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> <ul style="list-style-type: none"> ▪ Ansatz: Verwendung eines Hardwartertimers, der zyklisch eine Entscheidung des Schedulers anfordert. Man unterscheidet hier in <ul style="list-style-type: none"> ▪ Nicht-unterbrechende Strategie und ▪ Unterbrechende Strategie

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> <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 <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>

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



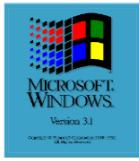
unterbrechende vs. nicht-unterbrechende Strategien

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.guidebookgallery.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 Elternprozessein neuer Prozess wird mit der BS Funktion <code>fork</code> erzeugtein 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

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

67

pstree bei fork()



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

1

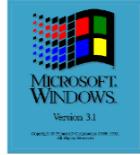
Prof. Dr. D. Säring © FH Wedel
2018

1

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

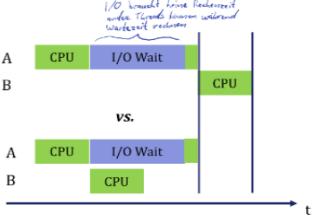
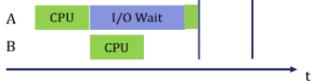
unterbrechende vs. nicht-unterbrechende Strategien



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;">  <p>Microsoft Windows Version 3.1 © Microsoft Corporation 1990</p> </div> <p>Bildquelle: http://www.gutenberg-galerie.org/screenshots/windows/</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



<p>1. Fairness</p> <ul style="list-style-type: none"> Kein Prozess/Thread sollte „unverhältnismäßig“ lange warten müssen sonst im Grenzfall: Nicht-unterbrechendes Scheduling! 	<p>3. Interaktivität</p> <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;">  </div>
<p>2. Hohe Ressourcen-Auslastung</p> <ul style="list-style-type: none"> Systemressourcen wie CPU, Festplatte, etc. sollten möglichst gleichzeitig genutzt werden Bsp: <div style="text-align: center;">  <p>vs.</p>  </div> 	

Mögliche Kriterien für eine gute (präemptive) Scheduling-Strategie



<p>1. Fairness</p> <p>2. Hohe Ressourcen-Auslastung</p> <p>3. Interaktivität</p>	<p>4. Hoher Durchsatz an Prozessen</p> <ul style="list-style-type: none"> Prozesse sollten in einer Reihenfolge bzw. so zu Unterbrechungszeitpunkten abgearbeitet werden, dass die resultierende Gesamtzeitdauer minimal ist 	<p>6. Garantierte Laufzeiten</p> <ul style="list-style-type: none"> Echtzeitbetriebssysteme benötigen garantierte (maximale) Laufzeiten Bsp: Prozess der entscheidet, ob Notbremse ausgelöst werden soll <div style="text-align: center;">  </div>
<p>5. Kurze Durchlaufzeit für 1 Prozess</p> <ul style="list-style-type: none"> Ein Prozess sollte nicht „ewig“ dauern, nur weil die resultierende Gesamtzeitdauer dann minimal ist 	<p>Problem: viele dieser Kriterien können im Gegensatz stehen, z.B.</p> <p>(4) vs. (5): ein Prozess wird sehr oft unterbrochen um hohen Durchsatz zu erreichen vs. wird stark „zerstückelt“; braucht lange bis zur kompletten Abarbeitung</p> <p>(1) vs. (6): ein Prozess wird immer wieder hinten angestellt</p> <p>→ Scheduler müssen sich für eine echte Teilmenge an zu optimierenden Kriterien entscheiden oder einen Kompromiss realisieren</p>	
<p>Problem: viele dieser Kriterien können im Gegensatz stehen, z.B.</p> <p>(4) vs. (5): ein Prozess wird sehr oft unterbrochen um hohen Durchsatz zu erreichen vs. wird stark „zerstückelt“; braucht lange bis zur kompletten Abarbeitung</p> <p>(1) vs. (6): ein Prozess wird immer wieder hinten angestellt</p> <p>→ Scheduler müssen sich für eine echte Teilmenge an zu optimierenden Kriterien entscheiden oder einen Kompromiss realisieren</p>		

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

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

2. Hohe Ressourcen-Auslastung

3. Interaktivität

6. Garantierte Laufzeiten

- Echtzeitbetriebssysteme benötigen garantierte (maximale) Laufzeiten
- Bsp: Prozess der entscheidet, ob Notbremsung 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

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

80

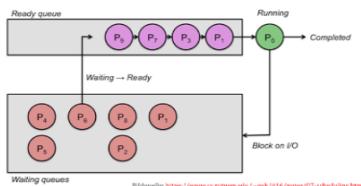
Erläutern Sie die Grundidee sowie die Vor- / Nachteile der **FCFS** Scheduling-Strategie. In welche Kategorie lässt sich diese einordnen? Wofür steht die Abkürzung ?missing IDENTIFIER? Nennen Sie eine Situation aus dem realen Leben die mit dieser Strategie gut verglichen werden kann.



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

Idee

- Prozesse werden in Warteschlange eingereiht
- Prozess vom Kopf der Schlange entnehmen
- Ausführen bis zu kompletter Abarbeitung
- Nächster Prozess



Eigenschaften

Vorteile:

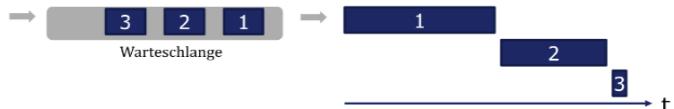
- Kein großer Scheduling-Overhead durch häufiges Umschalten zwischen Prozessen
- „fair“ („wer zuerst kommt wird zuerst bearbeitet“)
- minimal mögliche Durchlaufzeit für einen Prozess (Prozess wird nie unterbrochen)

Nachteile:

- typischerweise geringe Ressourcenauslastung
- typischerweise nicht interaktiv
- typischerweise geringer Durchsatz an Prozessen

→ Situation wie an der Supermarktkasse

Beispiel



Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

81

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 	<p>Vorteile:</p> <ul style="list-style-type: none"> Kaum Scheduling-Overhead „fair“ („wer schnell durch ist, darf als Nächstes ran“) <p>Nachteile:</p> <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) → Situation wie an der Supermarktkasse mit freundlichem Vorlassen
Beispiel	

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 <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

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

Strategie #3: Zeitscheibenverfahren → Ablaufbeispiel



Gegeben			Resultierende Prozessor zu CPU Zuteilung beim Zeitscheibenverfahren																							
<table border="1"> <thead> <tr> <th>Process Name</th> <th>Arrival Time</th> <th>Execute Time</th> </tr> </thead> <tbody> <tr><td>P0</td><td>0</td><td>250</td></tr> <tr><td>P1</td><td>50</td><td>170</td></tr> <tr><td>P2</td><td>130</td><td>75</td></tr> <tr><td>P3</td><td>190</td><td>100</td></tr> <tr><td>P4</td><td>210</td><td>130</td></tr> <tr><td>P5</td><td>350</td><td>50</td></tr> </tbody> </table>			Process Name	Arrival Time	Execute Time	P0	0	250	P1	50	170	P2	130	75	P3	190	100	P4	210	130	P5	350	50	<p>Quantum: 100ms</p> <p>Execute Time</p> <p>Round Robin Scheduling</p>		
Process Name	Arrival Time	Execute Time																								
P0	0	250																								
P1	50	170																								
P2	130	75																								
P3	190	100																								
P4	210	130																								
P5	350	50																								

Strategie #3: Zeitscheibenverfahren → Ablaufbeispiel



Gegeben			Resultierende Prozessor zu CPU Zuteilung beim Zeitscheibenverfahren																							
<table border="1"> <thead> <tr> <th>Process Name</th> <th>Arrival Time</th> <th>Execute Time</th> </tr> </thead> <tbody> <tr><td>P0</td><td>0</td><td>250</td></tr> <tr><td>P1</td><td>50</td><td>170</td></tr> <tr><td>P2</td><td>130</td><td>75</td></tr> <tr><td>P3</td><td>190</td><td>100</td></tr> <tr><td>P4</td><td>210</td><td>130</td></tr> <tr><td>P5</td><td>350</td><td>50</td></tr> </tbody> </table>			Process Name	Arrival Time	Execute Time	P0	0	250	P1	50	170	P2	130	75	P3	190	100	P4	210	130	P5	350	50	<p>Quantum: 100ms</p> <p>Execute Time</p> <p>Round Robin Scheduling</p>		
Process Name	Arrival Time	Execute Time																								
P0	0	250																								
P1	50	170																								
P2	130	75																								
P3	190	100																								
P4	210	130																								
P5	350	50																								

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



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>„hohe Priorität-Meld. Zeitabwicklungsfolgen“</i></p> <p>Bildquelle: http://www.cs.utexas.edu/~psk/314/notes/07_scheduling.html</p>

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>(Fehlende Priorität führt zu Verhungern)</i></p> <p>Bildquelle: http://www.cs.utexas.edu/~osk/314/notes/07-scheduling.html</p>

Multilevel Queue Scheduling (MQS)



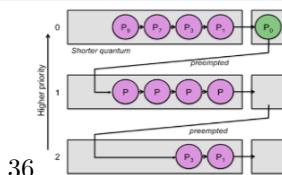
Idee	Lösung #2: Time-Slicing der Queues
<ul style="list-style-type: none"> Es gibt nicht den besten Scheduler! (der alle Kriterien erfüllt), daher teile die Prozesse in verschiedene Kategorien ein, z.B. <ul style="list-style-type: none"> interaktive Prozesse mit Benutzeroberfläche (Vordergrundprozesse) 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> <p><i>(Warteschlangen in Gruppen einteilen & darauf die Zeitscheibenverfahren anwenden)</i></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



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

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

90

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
$S_{1,1} x_1 \text{ ms}$	$S_{1,1} x_1 \text{ ms}$
$S_{2,1} x_2 \text{ ms}$	$S_{2,1} x_2 \text{ ms}$
$S_{3,1} x_3 \text{ ms}$	$S_{3,1} x_3 \text{ ms}$
	$S_{4,1} x_4 \text{ ms}$
	$S_{5,1} x_5 \text{ ms}$
	$S_{6,1} x_6 \text{ ms}$
	$S_{7,1} x_7 \text{ ms}$
	$S_{8,1} x_8 \text{ ms}$
	$S_{9,1} x_9 \text{ ms}$

Scheduling Strategien:

- <http://www.cs.rutgers.edu/~psk/4116/notes/07-scheduling.html>
- <http://www.cs.unmontana.edu/~chandrima.sarker/AdvancedOS/SchedulingOS/>
- <http://www.pjwing.org.uk/cs/ops.html>

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

91

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



Completely Fair Scheduler (CFS) I

Ausgangssituation & Kernidee	Prozess-Prioritäten
<ul style="list-style-type: none"> 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: <ul style="list-style-type: none"> Jeder Prozess sollte gleiche Rechenzeit erhalten Sobald ein oder mehrere Prozesse hinsichtlich Rechenzeit stark benachteiligt sind, sollten diese die CPU wieder zugeteilt bekommen Zentrale Variable vruntime (virtuelle Laufzeit), diese misst, wie lange ein Prozess bisher gelaufen ist <p><i>Bsp: Priorität geht in Form eines Gewichtes P_i mit ein. Es wird mit der Prozessdauer multipliziert.</i></p> <p><i>P₀ = 100ms P₁ = 100ms VRKT P₀(D,5) = 50ms P₁(D) = 200ms Priorität</i></p>	<ul style="list-style-type: none"> 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. Siring © 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	Prozess-Prioritäten
<ul style="list-style-type: none"> 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: <ul style="list-style-type: none"> Jeder Prozess sollte gleiche Rechenzeit erhalten Sobald ein oder mehrere Prozesse hinsichtlich Rechenzeit stark benachteiligt sind, sollten diese die CPU wieder zugeteilt bekommen Zentrale Variable vruntime (virtuelle Laufzeit), diese misst, wie lange ein Prozess bisher gelaufen ist <p><i>Bsp: Priorität geht in Form eines Gewichtes P_i mit ein. Es wird mit der Prozessdauer multipliziert.</i></p> <p><i>P₀ = 100ms P₁ = 100ms VRKT P₀(D,5) = 50ms P₁(D) = 200ms Priorität</i></p>	<ul style="list-style-type: none"> 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/>

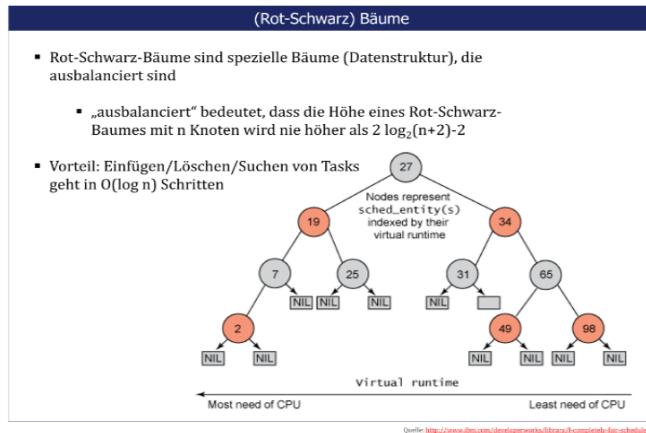
Vorlesung Betriebssysteme

Prof. Dr. D. Siring © FH Wedel
2019

94

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

Completely Fair Scheduler (CFS) II



Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

95

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

Completely Fair Scheduler (CFS) III



Task Groups

- Task Group:
mehrere Prozesse können zu einer Gruppe zusammengefasst werden und CPU Zeitzuweisung kann auf Gruppenzuordnung vom Scheduler vorgenommen werden
- Früher: $O(1)$ Scheduler:
Benutzer A = 2 Prozesse A_1, A_2
Benutzer B = 1 Prozess B_1
 - A_1, A_2, B_1 erhielten die gleiche Rechenzeit
 - Benutzer A bekam doppelt so viel Rechenzeit wie Benutzer B
- CFS:
 $A_1 + A_2 = \text{Task Group } \#1, B_1 = \text{Task Group } \#2$
→ $A_1 = 25\%, A_2 = 25\%, B_1 = 50\% \text{ CPU Zeit}$

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

96

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://cukkelink.org/patches/bfz/info-faq.txt>

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

→ Details auf Codebase zu einem Linux Scheduler: http://criticallinux.com/news/what-came-up-with/2013/12/linux_scheduler_ticks.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



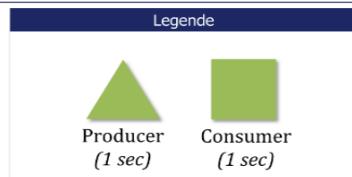
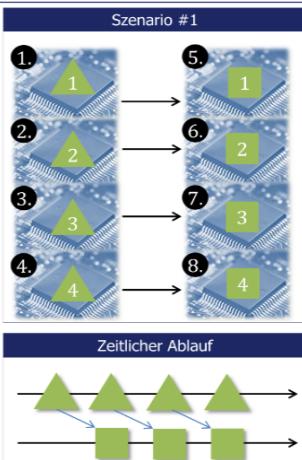
Alles gut?

- Prozesse und Threads können auf Mehrkern-CPPUs 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**

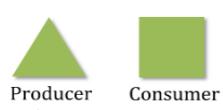
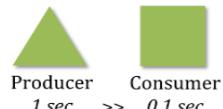
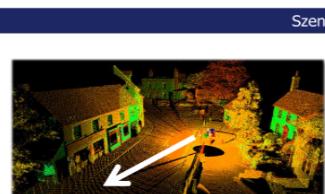
Das Nebenläufigkeitsproblem tritt in der Praxis der Softwareentwicklung z.B. beim *Producer/Consumer Pattern* auf

Verarbeitung der Sensordaten („Heile-Welt Szenario“)



Zeitlicher Ablauf

Verarbeitung der Sensordaten (Realistische Situation)



Nennen Sie die beiden Möglichkeiten der Thread-Erzeugung und 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 Elternprozessein neuer Prozess wird mit der BS Funktion <code>fork</code> erzeugtein 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

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

67

pstree bei fork()



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

10

Prof. Dr. D. Säring © FH Wedel

1

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

Lösungsvorschläge

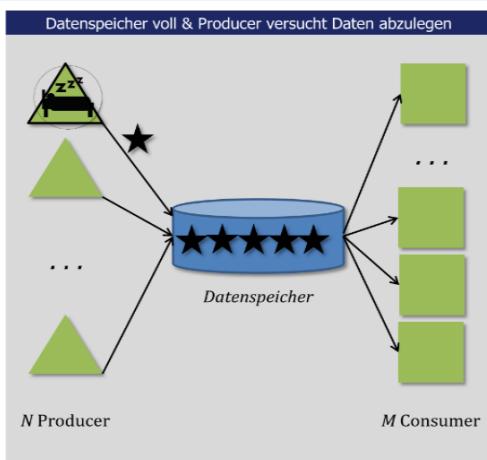


Neue Probleme!	
1	Datenspeicher voll & Producer versucht Daten abzulegen
2	Datenspeicher leer & Consumer versucht Daten zu lesen
3	Gleichzeitiger Zugriff auf den Datenspeicher: Nebenläufigkeits- bzw. Prozess/Thread-synchronisationsproblem



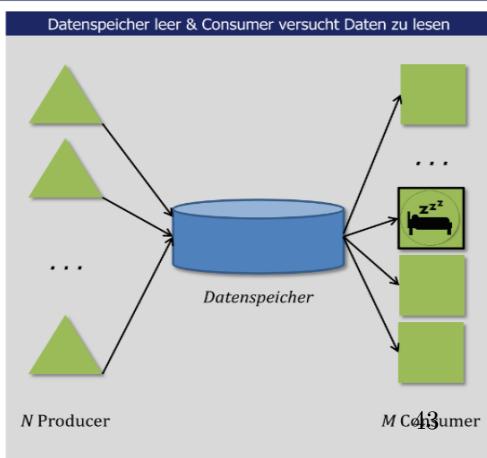
Quelle: http://www.agila.de/images/Wie_junge_Hunde_schlafen.png

Lösung für Teilproblem 1



Lösung	
→ Producer schläft bis wieder Platz im Datenspeicher ist	

Lösung für Teilproblem 2



Lösung	
→ Consumer schläft bis wieder mind. 1 Datensatz im Speicher ist	

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



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

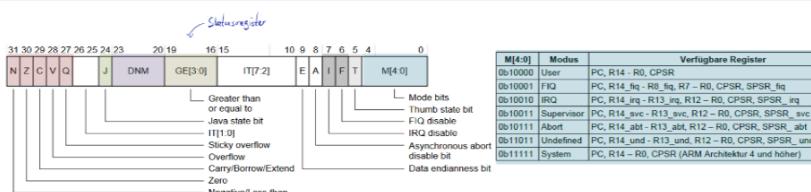
- Einfachste Lösung auf einem Einprozessorsystem: **Ausschalten der Interrupts**, nach dem Eintreten in einen kritischen Bereich
 - Kein CPU Wechsel mehr möglich -> kein anderer Prozess kann während Speicheränderungen Zugriff erhalten
 - Kein guter Ansatz, der Prozess könnte das System dauerhaft blockieren
 - 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 \geq C

[7..6] Execute Interrupts

I: Ermöglicht das Unterbinden von IRQs

F: Ermöglicht das Unterbinden von FIRQs

[24] Jazelle

J: Flag für die Verwendung von Java Code

[5] Thumb mode

T: Thumb- (=1) oder ARM-Instruktionen (=0)

[23...20] DNM

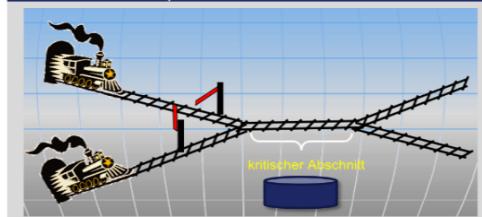
Do Not Modify!

[4...0] Processor mode

Wechselseitiger Ausschluss: Naiver Ansatz (I)



Metapher für kritischen Abschnitt



So klappt es nicht!

Thread A ↳ Schleifer könnte nun teilweise Mann unterbrechen.

```
→ ...
if (Sperre == 0)
{
  Sperre = 1;
  kritischer_Abschnitt();
  Sperre = 0;
} else warte();
...
```

Thread B

```
→ ...
if (Sperre == 0)
{
  Sperre = 1;
  kritischer_Abschnitt();
  Sperre = 0;
} else warte();
...
```

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 petersonsloesung;

public class MyThread extends Thread {

    static int turn;
    static boolean[] interested = new boolean[2];
    static int kritischerDatenbereich;
    int id;

    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 Vorrang 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();
}
```

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-Set-Lock Operationen

- 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

- ...
- Was wenn Thread A@CPU1 und Thread B@CPU2 *gleichzeitig* 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-Set-Lock Operation jetzt zur Thread-Synchronisation verwendet werden?

enter_region: TSL REGISTER,LOCK CMP REGISTER,#0 JNE enter_region RET	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
leave_region: MOVE LOCK,#0 RET	j store a 0 in lock j return to caller

- 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 j was lock zero? j if it was non zero, lock was set, so loop j return to caller; critical region entered
leave_region: MOVE LOCK,#0 RET	store a 0 in lock j 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 darf jetzt betreten werden und setzt Ausführungen im kritischen Abschnitt fort

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

Wechselseitiger Ausschluss: Lösungsmöglichkeiten (IV)



(5) Semaphor	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 (<code>down</code> und <code>up</code>) 	<ul style="list-style-type: none"> up-Operation erhöht den Wert des Semaphors um eins <ul style="list-style-type: none"> 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 	

Vorlesung Betriebssysteme

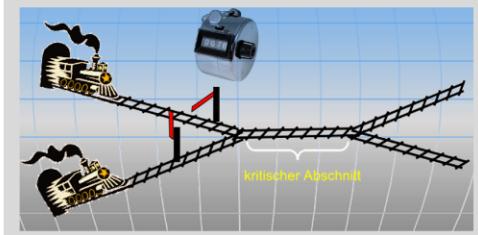
Prof. Dr. D. Säring © FH Wedel
2019

137

Realisierung des wechselseitigen Ausschlusses: Semaphore



Metapher für kritischen Abschnitt + Semaphore



Semaphor



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

Vorlesung Betriebssysteme

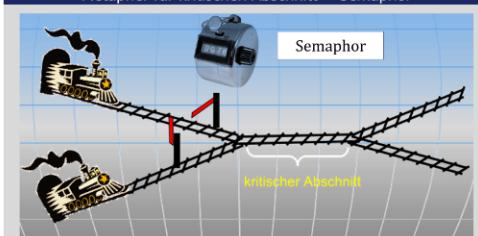
Prof. Dr. D. Säring © FH Wedel
2019

138

Realisierung des wechselseitigen Ausschlusses: Semaphore



Metapher für kritischen Abschnitt + Semaphore



Wechselseitiger Ausschluss: jetzt klappt es!

Thread A

Thread B

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

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

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);  
...
```

CPU 1
Thread A

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);  
...
```

CPU 1
Thread A

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

CPU 2
Thread B

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

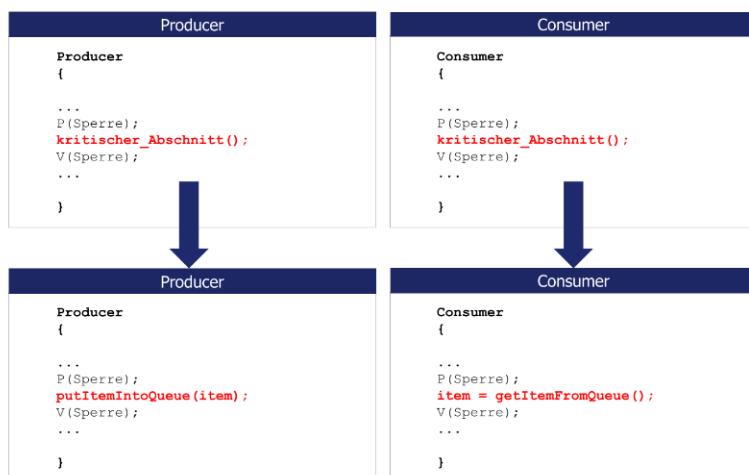
Beschreiben Sie den zeitlichen Ablauf des *wechselseitigen Ausschlusses* unter Verwendung einer **Semaphore** anhand eines Beispiels.

Wechselseitiger Ausschluss: Lösungsmöglichkeiten (IV)



(5) Semaphor	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 (<code>down</code> und <code>up</code>) 	<ul style="list-style-type: none"> up-Operation erhöht den Wert des Semaphors um eins <ul style="list-style-type: none"> 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. Semaphore)



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 im Speicher ein Objekt	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	<ul style="list-style-type: none"> ▪ 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	
<pre>Jedes Objekt in Java hat Monitor-Fähigkeiten! public class MeinDatenspeicher { ... public synchronized void addItem(...) { ... } public synchronized int getItem(...) { ... } }</pre>	

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

150

Schlussfolgerung

Worin liegt das Problem?	Wie könnte man es besser machen?
<ul style="list-style-type: none"> ▪ 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 Programmiersprachenkonzept 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 	<ul style="list-style-type: none"> ▪ 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 <ul style="list-style-type: none"> ▪ send(destination, &message) ▪ receive(source, &message)

Vorlesung Betriebssysteme

Prof. Dr. D. Säring © FH Wedel
2019

151