

Betriebssysteme

SS 2015

Hans-Georg Eßer
Dipl.-Math., Dipl.-Inform.

Foliensatz A:

- Einführung
- Installation der Linux-VM
- Shell-Grundlagen, Dateiverwaltung

v1.1, 2015/02/01

Hilfreiche Vorkenntnisse:

- **Linux-Shell** – Benutzung der Standard-Shell *bash* unter Linux
→ Bash-Crashkurs
- **C** – Grundlagen der Programmierung in C (oder C++, C#, Java)
- **Rechnerarchitekturen** – grober Aufbau eines Computers (Prozessor, Hauptspeicher, Peripherie etc.)

Veranstaltung kombiniert:

Theorie und Praxis der Betriebssysteme

Service / Web-Seite: <http://fom.hgesser.de>

- Folien und Praktikumsaufgaben
- Vorlesungs-Videos („test, test“)
- Probeklausur gegen Semesterende
- Folien auch im Online-Campus

Praktikum:

- Systemadministration unter Linux
- Praktische Programmier-Beispiele zum Theorieteil in **C** umsetzen

Prüfung und Benotung:

1. Lernfortschrittskontrolle (LFK)
2. Klausur über 120 Minuten

Fragen:

- direkt in der Vorlesung (Handzeichen)
- oder danach oder per E-Mail

Hans-Georg Eßer

- Dipl.-Math. (RWTH Aachen, 1997)
Dipl.-Inform. (RWTH Aachen, 2005)
Fachjournalist (FJS Berlin, 2006)
- Chefredakteur Linux-Zeitschrift (seit 2000)
und Autor diverser Computerbücher
- LPI-zertifiziert (LPIC-1 und LPIC-2)
- seit 2006 Dozent (u. a. FOM, HS München, TH Nürnberg,
Univ. Erlangen-Nürnberg): Betriebssysteme, Rechner-
architektur, IT-Infrastruktur, Informatik-Grundlagen, System-
programmierung, Betriebssystem-Entwicklung, IT-Sicherheit
- Seit 2010 Doktorand (Univ. Erlangen-Nürnberg), Doktorarbeit
fertig, Verteidigung im März 2015



Einführung und Motivation

Linux-Administration

- Nutzen von Shell-Befehlen
- Standard-Datei- und -Verzeichnis-Operationen
- Editor vi
- Shell-Variablen, Unix-Filter-Programme
- Jobs und Prozesse
- Software-Verwaltung

LPI-Zertifizierung (1)



LPIC-1

LPI Certified Junior Level Linux Professional

- Work at the Linux command line
- Perform easy maintenance tasks: help out users, add users to a larger system, backup & restore, shutdown & reboot
- Install and configure a workstation (including X) and connect it to a LAN, or a stand-alone PC via modem to the Internet.



LPIC-2

LPI Certified Advanced Level Linux Professional

- Administer a small to medium-sized site
- Plan, implement, maintain, keep consistent, secure, and troubleshoot a small mixed (MS, Linux) network, including a LAN server (samba), Internet Gateway (firewall, proxy, mail, news), Internet Server (webserver, FTP server)
- Supervise assistants
- Advise management on automation and purchases

**LPI Certified Senior Level Linux Professional**

- hauptsächlich: LDAP
- dazu: verschiedene Spezialisierungen
 - LPI 302: Mixed Environments
 - LPI 303: Security
 - LPI 304: Virtualization and High Availability
- es kommen vielleicht noch weitere

Im Rahmen dieser
Vorlesung: teilweise (!)
Vorbereitung auf die
LPIC-1-Prüfungen
101 und 102



- keine grafischen Werkzeuge – auch wenn es welche gibt
- also: nicht YaST & Co., sondern Kommandozeilentools, Konfigurationsdateien, Shell-Skripte
- verstehen, was im Hintergrund abläuft

- Für die Zertifizierung: Stoff aus dieser Veranstaltung reicht nicht
- insbesondere: „passives Konsumieren“ reicht nicht
- Prüfungen sind so gestaltet, dass Administratoren sie leicht bestehen können
→ üben, üben, üben :)
- Kosten: 145 € (+ USt) pro Prüfung (LPIC-1 = 2 Prüfungen)

- Windows setzt auch bei der Administration überwiegend auf grafische Tools
- „GUI-Administration“ leichter (schneller) zu erlernen, bietet aber weniger Möglichkeiten, wenn etwas schief geht
- Windows ist im Rahmen dieser Vorlesung kein Thema

„Klicken Sie auf Schließen.“

- Im Theorieteil nicht: „Wie bediene ich ... ?“, sondern:
„Wie und warum funktioniert ... intern?“
 - Konsequenzen für Anwendungsentwickler
 - Sicherheitsprobleme
 - Auswahl eines geeigneten Betriebssystems
- ... und das Thema ist auch an sich spannend

- Schützt Hardware vor direkten Zugriffen
(→ defekte oder bösartige Software)
- Befreit Software vom Zwang, die Hardware im Detail zu kennen
- Zulassen mehrerer Benutzer und Abgrenzung (Multi-user)
- Parallelbetrieb mehrerer Anwendungen (Multi-tasking): faire Aufteilung der Ressourcen

- Abstraktionsschicht zwischen Hardware und Programmen (→ virtuelle Maschine)
- Verwaltung der vorhandenen Ressourcen
- Einheitlicher Zugriff auf Geräte einer groben Kategorie, z. B.:
 - *Datenträger* (Plattenpartition, CD, DVD, Diskette, USB-Stick, Netzwerk-Volume)
 - *Drucker* (PostScript-Laser, Etikettendrucker, Billig-Tintenstrahler, ...)

- Virtualisierung des Speichers
 - Anwendungen müssen nicht wissen, wo sie im Hauptspeicher liegen
 - Speicher über phys. RAM hinaus verfügbar (Swap etc.)

Desktop-PC – die Standardaufgabe, Intel & Co.

- Anwendungsprogramme (Office, Grafik, kaufmännische Software etc.)
- Internet-Zugang und Web-basierte Anwendungen (WWW, E-Mail, File Sharing, ...)
- Datenbank-Client
- Software-Entwicklung
- Multimedia



Server-PC

Häufig ähnliche Hardware wie Desktop-PC, aber ganz andere Einsatzgebiete:

- Web- / FTP- / Mail-Server (Internet oder Intranet)
- Datenbank-Server
- „Number Crunching“ bzw. High Performance Computing (oft: Cluster)

Industrieanwendungen

- Robotersteuerung
- automatische Navigation
- Temperaturregelung
- Motorenkontrolle
- Herzschrittmacher

→ **Echtzeit-Betriebssysteme**
(real time operating systems)



Bild: Wikipedia, KUKA Schweißanlagen

Embedded systems (ohne Echtzeit-Ansprüche)

- Mobiltelefone, PDAs, mobile MP3/Video-Player
- Fernseher, Videorekorder, DVD-Player
- DSL-WLAN-Router (mit Firewall etc.)
- Taschenrechner
- Videospiel-Konsolen
- Geldautomaten

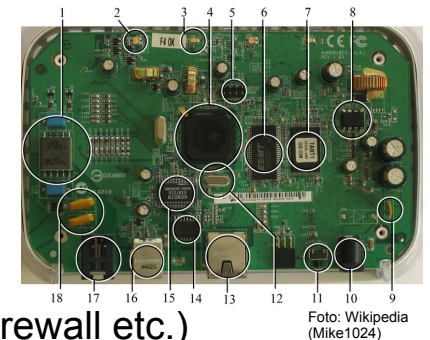


Foto: Wikipedia (Mike1024)

Beim Programmieren tauchen häufig Probleme in zwei Bereichen auf:

- **Zuverlässigkeit**

Software tut nicht das, was sie soll;
unerwartetes Verhalten;
mangelnde Fehlertoleranz

- **Sicherheit**

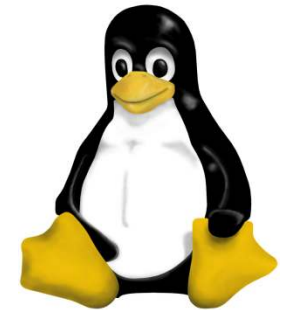
Software ist nicht geschützt vor Angriffen
durch Dritte

Funktionsweise des Betriebssystems nicht klar
→ fehlerhaft programmierte Anwendungen, z. B.

- Race Conditions
- Buffer Overflows

Darum verstehen und lernen, wie
Betriebssysteme intern arbeiten

- Etabliertes Standardsystem für sehr viele Plattformen (PC Desktop / Server, Embedded etc.)
- vor allem auf Servern weit verbreitet
- Offene Kernel-Quellen:
 - nachlesen, wie etwas geht
 - ändern, was nicht gefällt
- Image eines virtuellen Linux-PCs für VMware / VirtualBox

**Grundlagenbuch Linux**

Grundlagen, Techniken, Lösungen
(Eßer, Dölle)

Data Becker, 2007

ISBN: 3815829011

→ als PDF-Dokument im Campus-System

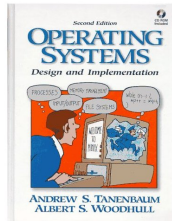
**Linux**

Das umfassende Handbuch
(Kofler)

Galileo Computing, 2013

ISBN: 3836225913

49,90 €

**Operating Systems**

Design and Implementation
(Tanenbaum, Woodhull)
Prentice Hall, 1997 (Klassiker)
ISBN: 0136386776
(englisch)

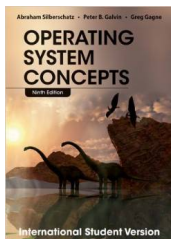
**Betriebssysteme**

Ein Lehrbuch mit Übungen zur System-
programmierung in Unix/Linux
(Ehse et al.)
ISBN 3-8273-7156-2
Pearson Studium, 2005, 30 Euro

Gliederung

**Operating Systems**

Internals and Design Principles
(Stallings)
Prentice Hall, 8. Auflage, 2014
ISBN: 1292061359
(englisch)

**Operating System Concepts**

(Silberschatz, Galvin, Gagne)
John Wiley & Sons, 9. Auflage, 2013
ISBN: 1118093755
(englisch)

BS Praxis

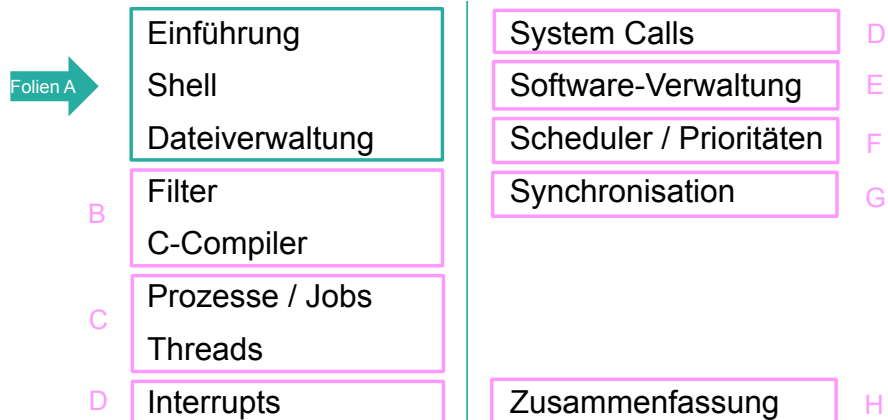
- **Arbeiten mit der Shell**
- Dateiverwaltung
- Filter
- Prozesse und Jobs
- Software-Verwaltung

BS Theorie

- Prozesse und Threads
- Polling und Interrupts
- System Calls
- Systemprogrammierung
- Scheduling
- Synchronisation

	Shell	Dateiverwaltung	Filter	C-Compiler	Prozesse / Jobs	Threads	Interrupts	System-Calls	Scheduler / Priorit.	Synchronisation	Speicherverw.	Software-Verw.	Partitionen	Auskunft	User, Gr., Rechte
BS Theorie					X	X	X	X	X	X	X				
Systemprog.				X	X	X		X	X						
Linux-Progr.				X	X	X		X	X						
BS Praxis	X	X	X	X	X			X	X			X	X	X	X

Debian Linux in der virtuellen Maschine



Praxisteil

- Installation VirtualBox auf Ihrem Notebook / Netbook
- Import einer virtuellen Maschine mit Debian Linux 6
- erste Experimente mit dem installierten Linux

Praxisteil (Demonstration)

- Arbeiten mit der Shell
- Verzeichnisnavigation, -Listings
- Dateien kopieren, umbenennen, verschieben
- Verzeichnisse erstellen, löschen etc.
- Dateien öffnen
- Der Editor „vi“

- Vor dem \$, >, # meist Hinweise auf Benutzer, Rechner, Arbeitsverzeichnis

```
[esser@macbookpro:BS] $
```

```
root@quad:~#
```

- esser, root: Benutzername; individuell
- macbookpro, quad: Rechnername
- BS, ~: Arbeitsverzeichnis, je nach Prompt-Einstellung auch in voller Länge (z. B. /home/esser/Daten/FOM/SS2015/BS)
- ~ = „Home-Verzeichnis“ des Benutzers

- Shell zeigt durch **Prompt** an, dass sie bereit ist, einen Befehl entgegen zu nehmen
- Prompts können verschieden aussehen:
 - ... \$ _ : Anwender-Prompt, nicht-privilegiert
 - ... > _ : Anwender-Prompt, nicht-privilegiert
 - ... # _ : Root-Prompt, für den Administrator

- Am Prompt Befehl eingeben und mit [Eingabe] abschicken
- Shell versucht, (in der Regel) erstes Wort als Kommandoname zu interpretieren:
 - Alias? (→ nicht in dieser Vorlesung)
 - Shell-interne Funktion? (→ nicht in dieser Vorles.)
 - eingebautes Shell-Kommando? (z. B. cd)
 - externes Programm? (Suche in Pfad)

- Beispiel: Aktuelles **Arbeitsverzeichnis** anzeigen (`pwd` = **p**rint **w**orking **d**irectory)

```
[esser@quad:~]$ pwd
/home/esser
[esser@quad:~]$ _
```

- Nach Abarbeiten des Befehls (oft: mit einer „Antwort“) erscheint wieder der Prompt – Shell ist bereit für nächstes Kommando

- **Inhaltsverzeichnis** anzeigen: `ls` (**l**ist)
- bezieht sich immer auf das aktuelle Arbeitsverzeichnis (Alternative: Ort als Parameter angeben)

```
[esser@quad:~]$ ls
bahn-2015-03-06.pdf      bh-win-04-kret.pdf
buch_kap08.pdf          bv-anleitung.pdf
bz2.pdf
[esser@quad:~]$ ls /tmp
cvcd  kde-esser  ksocket-esser  orbit-esser
ssh-vrUNLb1418  virt_1111
[esser@quad:~]$ _
```

- Mehrere Befehle auf einmal abschicken: mit Semikolon ; voneinander trennen

```
[esser@quad:~]$ pwd; pwd
/home/esser
/home/esser
[esser@quad:~]$ _
```

- Inhalt mit mehr Informationen: `ls -l`

```
[esser@quad:~]$ ls -l
-rw-r--r-- 1 esser users 29525 Nov 09 14:11 bahn-2015-03-06.pdf
-rw-r--r-- 1 esser users 74520 Apr 10 2004 bh-win-04-kret.pdf
-rw-r--r-- 1 esser users 856657 Oct 21 2005 buch_kap08.pdf
-rw-r--r-- 1 esser esser 738570 Mar 17 20:29 bv-anleitung.pdf
-rw-r--r-- 1 esser users 123032 Sep 22 2003 bz2.pdf
[esser@quad:~]$ _
```

- Ausgabe enthält zusätzlich:
 - Zugriffsrechte (`-rw-r--r--` etc.)
 - Dateibesitzer und Gruppe (`esser, users`)
 - Größe und Datum/Zeit der letzten Änderung

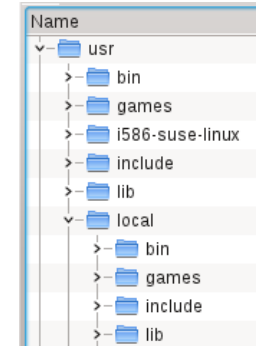
- Leere Datei erzeugen (für Experimente): `touch`

```
[esser@quad:~]$ touch Testdatei
[esser@quad:~]$ ls -l Testdatei
-rw-r--r-- 1 esser esser 0 Apr  7 13:58 Testdatei
[esser@quad:~]$ _
```

- Datei hat Größe 0

Grundlagen (1)

- Linux kennt keine „Laufwerksbuchstaben“ (C:, D: etc.)
- Wurzelverzeichnis heißt /
- Pfadtrenner: auch / – d. h.:
/usr/local/bin ist das Verzeichnis bin im Verzeichnis local im Verzeichnis usr.
(wie bei Webadressen)



- Fehlermeldungen: Unbekanntes Kommando

```
[esser@quad:~]$ fom
No command 'fom' found, did you mean:
Command 'fim' from package 'fim' (universe)
Command 'gom' from package 'gom' (universe)
Command 'fop' from package 'fop' (universe)
Command 'fdm' from package 'fdm' (universe)
Command 'fpm' from package 'fpm2' (universe)
[...]
fom: command not found
[esser@quad:~]$ _
```

- Meldung kann auch deutschsprachig sein

Grundlagen (2)

- Weitere Datenträger erscheinen in Unterordnern
 - Beispiel: DVD mit Dateien zum Kurs hat Volume-Name BS-ESSER
 - Datei `test.txt` auf oberster DVD-Verzeichnisebene ist als `/media/BS-ESSER/test.txt` erreichbar (Windows: `e:\test.txt`)
 - Datei `Software/index.html` der DVD entsprechend als `/media/BS-ESSER/Software/index.html` (Windows: `e:\Software\index.html`)

Grundlagen (3)

- Für private Nutzerdaten hat jeder Anwender ein eigenes **Home-Verzeichnis**, das i. d. R. unterhalb von `/home` liegt, z. B. `/home/esser`.
- Die Tilde `~` ist immer eine Abkürzung für das Home-Verzeichnis
 - funktioniert auch in zusammengesetzten Pfaden
 - `~/Daten/brief.txt` statt `/home/esser/Daten/brief.txt`

Grundlagen (5)

- Zwei Spezialverzeichnisse in jedem Ordner
 - `..` ist das Verzeichnis eine Ebene höher (von `/usr/local/bin` aus ist `..` also `/usr/local`)
 - `.` ist das aktuelle Verzeichnis
- Pfade kann man **absolut** und **relativ** zusammen bauen
 - absoluter Pfad beginnt mit `/`
 - relativer Pfad nicht; er gilt immer ab dem aktuellen Arbeitsverzeichnis

Grundlagen (4)

- Ausnahme: Das Home-Verzeichnis des Systemadministrators `root` ist nicht `/home/root`, sondern `/root`
- Der Trick mit der Tilde `~` funktioniert aber auch für `root`
- Warum? `/home` könnte auf einer separaten Partition liegen und bei einem Fehlstart nicht verfügbar sein

Verzeichnisnavigation

- Kommando `cd` (change directory) wechselt in ein anderes Verzeichnis
- Zielverzeichnis als Argument von `cd` angeben – wahlweis mit relativem oder absolutem Pfad

```
[esser@quad:~]$ pwd
/home/esser
[esser@quad:~]$ cd /home ; pwd
/home
[esser@quad:home]$ cd .. ; pwd
/
[esser@quad:/]$ _
```

Datei kopieren

- Kommando `cp` (**copy**) kopiert eine Datei
- Reihenfolge: `cp Original Kopie`

```
[esser@quad:tmp]$ ls -l
-rw-r--r-- 1 esser wheel 1501 Apr  5 11:37 test.dat
[esser@quad:tmp]$ cp test.dat kopie.dat
[esser@quad:tmp]$ ls -l
-rw-r--r-- 1 esser wheel 1501 Apr  8 12:17 kopie.dat
-rw-r--r-- 1 esser wheel 1501 Apr  5 11:37 test.dat
[esser@quad:tmp]$ _
```

! Kopie erhält aktuelles Datum/Zeit

Datei verschieben

- Kommando `mv` (**move**) verschiebt eine Datei
- Reihenfolge: `mv AltName NeuerOrdner/`

```
[esser@quad:tmp]$ ls -l
-rw-r--r-- 1 esser wheel 1501 Apr  5 11:37 test.dat
[esser@quad:tmp]$ mv test.dat /home/esser/
[esser@quad:tmp]$ ls -l
[esser@quad:tmp]$ ls -l /home/esser/
-rw-r--r-- 1 esser wheel 1501 Apr  5 11:37 test.dat
[...]
```

! Verschieben ändert Datum/Zeit nicht

Datei umbenennen

- Kommando `mv` (**move**) benennt eine Datei um
- Reihenfolge: `mv AltName NeuName`

```
[esser@quad:tmp]$ ls -l
-rw-r--r-- 1 esser wheel 1501 Apr  5 11:37 test.dat
[esser@quad:tmp]$ mv test.dat neu.dat
[esser@quad:tmp]$ ls -l
-rw-r--r-- 1 esser wheel 1501 Apr  5 11:37 neu.dat
[esser@quad:tmp]$ _
```

! Umbenennen ändert Datum/Zeit nicht

Datei löschen

- Kommando `rm` (**remove**) löscht eine Datei

```
[esser@quad:tmp]$ ls -l
-rw-r--r-- 1 esser wheel 1501 Apr  5 11:37 test.dat
[esser@quad:tmp]$ rm test.dat
[esser@quad:tmp]$ ls -l
[esser@quad:tmp]$ _
```

Mehrere Dateien

- Einige Befehle akzeptieren mehrere Argumente, z. B.

- mv (beim Verschieben in anderen Ordner)
- rm

- Beispiele:

```
[esser@quad:tmp]$ mv datei1.txt datei2.txt Ordner/  
[esser@quad:tmp]$ rm datei3.txt datei4.txt datei5.txt  
[esser@quad:tmp]$ _
```

- Löschbefehl mit Wildcards zu gewagt?
→ vorher mit echo testen:

```
[esser@quad:Downloads]$ echo rm *.zip  
rm Logo_a5_tif.zip Uebung1.zip c32dwenu.zip  
ct.90.01.200-209.zip ct.90.12.130-141.zip  
ct.91.02.285-293.zip ct.91.12.024-025-1.zip  
ct.91.12.024-025.zip ct.92.08.052-061.zip  
ix.94.03.010-011.zip ix.94.07.068-071.zip  
[esser@quad:Downloads]$ rm *.zip  
[esser@quad:Downloads]$ _
```

Wildcards (*, ?)

- Bei Befehlen, die mehrere Argumente akzeptieren, können Sie auch Wildcards verwenden:

- * steht für beliebig viele (auch 0) beliebige Zeichen
- ? steht für genau ein beliebiges Zeichen

- Beispiele:

```
[esser@quad:~]$ ls -l ??????.pdf  
-rw-r--r-- 1 esser staff 79737 Apr 2 01:18 RegA4.pdf  
-rw-r--r-- 1 esser staff 132246 Apr 4 18:02 paper.pdf  
[esser@quad:~]$ rm /tmp/*  
[esser@quad:~]$ _
```

- Das letzte Beispiel verrät etwas über das Auflösen der Wildcards
 - Wenn Sie `rm *.zip` eingeben, startet die Shell *nicht* `rm` mit dem Argument „*.zip“
 - Die Shell sucht im aktuellen Verzeichnis alle passenden Dateien und macht jeden Dateinamen zu einem Argument für den `rm`-Aufruf.
 - Es wird also
`rm Logo_a5_tif.zip Uebung1.zip c32dwenu.zip ct.90.01.200-209.zip ...`
aufgerufen.

Mit Verzeichnissen können Sie ähnliche Dinge tun wie mit Dateien

- Verzeichnis erstellen
- (leeres!) Verzeichnis löschen
- Verzeichnis umbenennen oder verschieben
- Verzeichnis rekursiv (mit allen enthaltenen Dateien und Unterordnern) löschen

Verzeichnis erstellen

- Kommando `mkdir` (**make directory**) erzeugt ein neues (leeres) Unterverzeichnis

```
[esser@quad:tmp]$ ls -l
[esser@quad:tmp]$ mkdir unter
[esser@quad:tmp]$ ls -l
drwxr-xr-x  2 esser  wheel  68 Apr  8 14:28 unter
[esser@quad:tmp]$ cd unter
[esser@quad:unter]$ ls -l
[esser@quad:unter]$ cd ..
[esser@quad:tmp]$ _
```

! Kurzform `md` für `mkdir` nicht immer vorhanden → vermeiden

Verzeichnis löschen

- Kommando `rmdir` (**remove directory**) löscht ein leeres (!) Unterverzeichnis

```
[esser@quad:tmp]$ touch unter/datei
[esser@quad:tmp]$ rmdir unter
rmdir: unter: Verzeichnis nicht leer
[esser@quad:tmp]$ rm unter/datei
[esser@quad:tmp]$ rmdir unter
[esser@quad:tmp]$ _
```

! Kurzform `rd` für `rmdir` nicht immer vorhanden → vermeiden

Verzeichnis umbenennen / verschieben

- funktioniert wie das Umbenennen / Verschieben von Dateien
- gleicher Befehl: `mv`, wieder zwei Varianten:
 - `mv Verzeichnis NeuerName`
 - `mv Verzeichnis AndererOrdner/`

Verzeichnis rekursiv löschen

- Kommando `rm` (**r**emove) hat eine Option `-r` zum rekursiven Löschen:

```
[esser@quad:tmp]$ mkdir a; mkdir a/b; mkdir a/b/c
[esser@quad:tmp]$ touch a/b/c/datei
[esser@quad:tmp]$ rmdir a
rmdir: a: Verzeichnis nicht leer
[esser@quad:tmp]$ rm -r a
[esser@quad:tmp]$ _
```

! Vorsicht beim rekursiven Löschen: „Was weg ist, ist weg“

- Undelete = Löschen rückgängig machen
 - gibt es unter Linux nicht
 - Wiederherstellung von gelöschten Dateien mit Profi-Tools möglich, wenn Computer nach dem Löschen sofort ausgeschaltet wurde
 - solche Tools stellen aber sehr viele Dateien wieder her → enormer Aufwand, anschließend die gesuchte Datei zu finden; u. a. sind die Dateinamen dauerhaft verloren

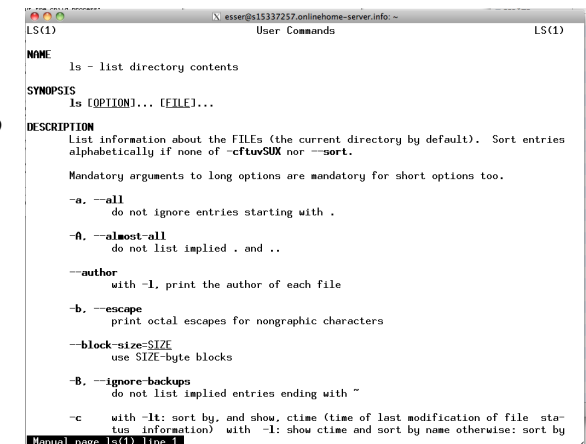
→ vor `rm -r ...` mehrfach prüfen ...

- **Argumente:** z. B. Dateinamen; beziehen sich oft auf Objekte, die manipuliert werden sollen
- **Optionen:** verändern das Verhalten eines Befehls
 - bei den meisten Befehlen zwei Varianten:
 - kurze Optionen: `-a`, `-b`, `-c`, ...
→ lassen sich kombinieren: `-abc` = `-a -b -c`
 - lange Optionen: `--ignore`, `--force`, `--all` etc.
 - Beispiel: `-r` bei `rm`

- Zu den meisten Kommandos gibt es eine sog. Manpage, die Sie über

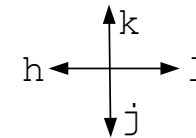
`man kommando`
abrufen

- Beispiel:
`man ls`



- Standard-Editor auf allen Unix-Systemen (und damit auch Linux): **vi** (visual editor)
- gewöhnungsbedürftige Bedienung
- zwei Betriebsarten
 - **Befehlsmodus** (nach Start aktiviert; Normalmodus)
 - **Bearbeitungsmodus**
- **vi** aus Versehen gestartet? Verlassen ohne Speichern von Änderungen mit **[Esc] :q!**

- Wechseln in den Bearbeitungsmodus: **i**, **I**, **a**, **A**
 - **i**: Text vor dem Cursor einfügen
 - **a**: Text nach dem Cursor einfügen
 - **I**: Text am Zeilenanfang einfügen
 - **A**: Text am Zeilenende einfügen
- Bearbeitungsmodus verlassen: **[Esc]**
- Navigieren im Text:
Cursortasten oder:



- Warum Umgang mit **vi** lernen?
 - auf jedem – noch so minimalistischen – Unix-System ist ein **vi** installiert (kleines Programm):

```
[esser@quad:~]$ ls -l /usr/bin/vi /usr/bin/emacs
-rwxr-xr-x 1 root root 5502096 Nov  9  2008 /usr/bin/emacs
-rwxr-xr-x 1 root root  630340 Oct 17  2008 /usr/bin/vi
```
 - läuft im Terminal → hilfreich bei Remote-Zugriff
 - Bei Problemen (Plattenfehler, nicht alle Dateisysteme verfügbar) sind andere Editoren evtl. nicht erreichbar, **vi** vielleicht doch → gilt leider nicht mehr für aktuelle Linux-Versionen
 - Thema ist LPI-prüfungsrelevant

- Zeichen / Text löschen:
 - im Bearbeitungsmodus mit **[Rückschritt]** und **[Entf]**, wie aus anderen Editoren bekannt
 - im Befehlsmodus mehrere Möglichkeiten:
 - **x** löscht Zeichen unter Cursor
 - **X** löscht Zeichen links von Cursor
 - **dw** löscht ab Cursor-Position bis Anfang des nächsten Worts
 - **dd** löscht aktuelle Zeile
 - **vorab Zahl**: Mehrfachausführung (**15dd**: 15 Zeilen)

- Speichern und beenden
 - Immer zuerst in den Befehlsmodus
→ im Zweifelsfall einmal [Esc] drücken
 - Speichern: `:w`
 - Speichern (erzwingen): `:w!`
 - Beenden (klappt nur, wenn Text seit letztem Speichern nicht verändert wurde): `:q`
 - Beenden erzwingen (ohne speichern): `:q!`
 - Speichern und beenden: `:wq` (oder: `ZZ` ohne „:“)

- Rückgängig machen / wiederherstellen
 - Letzte Änderung rückgängig machen: `u` (undo)
 - geht auch mehrfach: `u`, `u`, `u`, ...
 - ... und mit Mehrfachausführung: `3u` macht die letzten drei Änderungen rückgängig
 - Einen Undo-Schritt aufheben: `[Strg]+r` : redo
 - mehrfaches Redo: z. B. `3 [Strg]+r`

- Suche im Text
 - Vorwärtssuche: `/` und Suchbegriff, dann [Eingabe]
 - Sprung zum nächsten Treffer: `n` (next)
 - Rückwärtssuche: `?` und Suchbegriff, dann [Eingabe]
 - Sprung zum nächsten Treffer: `n`
 - Wechsel zwischen Vorwärts- und Rückwärtssuche:
einfach `/` bzw. `?`, dann Eingabe und mit `n` weiter
(in neuer Richtung) suchen

- Copy & Paste: Kopieren ...
 - `yw` (ab Cursorposition bis Wortende)
 - `y$` (ab Cursorposition bis Zeilenende)
 - `yy` (ganze Zeile)
 - `3yy` (drei Zeilen ab der aktuellen)
- ... und Einfügen
 - `P` (fügt Inhalt des Puffers an Cursorposition ein)
- Cut & Paste
 - Löschen mit `dd`, `dw` etc.; dann einfügen mit `P`

- Copy & Paste mit der Maus
 - Wenn Sie die grafische Oberfläche verwenden, geht es auch mit der Maus:
 - Kopieren: Mauszeiger auf 1. Zeichen, klicken (und gedrückt halten), zum letzten Zeichen ziehen, loslassen
 - Einfügen: Cursor zu Ziel bewegen, dann (im Einfügemodus!) die mittlere Maustaste drücken
 - Bei beiden Schritten muss man je nach `vi`-Version evtl. die [Umschalt]-Taste drücken

- Aufgabenblatt
 - Umgang mit Dateien und Verzeichnissen
 - Editor vi

- Datei im Editor öffnen:
`[esser@quad:~] vi Dateiname`
- zweite Datei an Cursorposition hinzuladen:
`:read Dateiname`
(im Befehlsmodus!)

Betriebssysteme

SS 2015

Hans-Georg Eßer
Dipl.-Math., Dipl.-Inform.

Foliensatz B:

v1.1, 2015/02/01

- Shell: Variablen, History, Filter-Programme, Pipelines
- C: Der GNU C Compiler (gcc)
- C: Crashkurs zur Programmiersprache C

Shell: Variablen, History und Filter

BS Praxis und BS Theorie (3)



Einführung
Shell
Dateiverwaltung
Filter
C-Compiler
Prozesse / Jobs
Threads
Interrupts

System Calls
Software-Verwaltung
Scheduler / Prioritäten
Synchronisation

Arbeiten in der Shell

Aus Einführung und Übungsblatt 1 bereits bekannt:

- pwd: aktuelles (Arbeits-) Verzeichnis anzeigen
- cd: Verzeichniswechsel
- .. : nächst höheres Verzeichnis
- ls: Verzeichnisinhalt anzeigen
- cp: Datei kopieren
- vi: Text-Editor
- mkdir: Verzeichnis erzeugen
- rmdir: Verzeichnis löschen
- rm: Datei löschen
- rm -r: Verzeichnis rekursiv löschen
- touch: Datei (leer) erzeugen; Zugriffsdatum aktualisieren
- less: Datei anzeigen
- grep: Suchen in Datei
- head, tail: Anfang und Ende einer Datei
- man: Hilfe anzeigen
- dmesg: Systemmeldungen ausgeben
- wc: word count
- shutdown: System runter fahren

- Die Shell (und auch andere Programme) nutzen **Umgebungsvariablen** (für Optionen, Einstellungen etc.)
- „set“ gibt eine Liste aller in dieser Shell gesetzten Variablen aus

```
$ set
BASH=/bin/bash
BASH_VERSION='3.2.48(1)-release'
COLUMNS=156
COMMAND_MODE=unix2003
DIRSTACK=()
DISPLAY=/tmp/launch-Lujw2L/org.x:0
EUID=501
GROUPS=()
HISTFILE=/home/esser/.bash_history
HISTFILESIZE=500
HISTSIZE=500
HOME=/home/esser
HOSTNAME=macbookpro.fritz.box
...
```

- Exportieren?
Wert einer Variablen gilt nur lokal in der laufenden Shell.
- Exportierte Variablen gelten auch in aus der Shell heraus gestarteten Programmen

```
$ A=eins; B=zwei; export A
$ echo "A=$A B=$B"
A=eins B=zwei
$ bash                # neue Shell starten; das ist ein neues Programm!
$ echo "A=$A B=$B"
! A=eins B=
$ exit                # diese zweite Shell verlassen, zurück zur ersten
$ echo "A=$A B=$B"
A=eins B=zwei
```

- Einzelne Variablen geben Sie mit „echo“ und einem Dollar-Zeichen (\$) vor dem Variablennamen aus

```
$ echo $SHELL
/bin/bash
$ _
```

- zum Ändern / Setzen schreiben Sie „var=wert“:

```
$ TESTVAR=fom
$ echo $TESTVAR
fom
$ set | grep TEST
TESTVAR=fom
$ _
```

- Sie können Variablen auch **exportieren**:

```
$ export TESTVAR
$ _
```

→ nächste Folie

- Liste aller exportierten Variablen gibt „export“ ohne Argument aus – allerdings in ungewöhnlicher Syntax

```
$ export
declare -x A="1"
declare -x Apple_PubSub_Socket_Render="/tmp/launch-CYfDhh/Render"
declare -x COMMAND_MODE="unix2003"
declare -x DISPLAY="/tmp/launch-Lujw2L/org.x:0"
declare -x HOME="/Users/esser"
declare -x INFOPATH="/sw/share/info:/usr/share/info"
declare -x LOGNAME="esser"
...
```

- (Hintergrund: „declare -x VAR“ exportiert ebenfalls die Variable VAR, ist also dasselbe wie „export VAR“)

- Shell merkt sich die eingegebenen Befehle („History“)
- Komplette Ausgabe mit „history“:

```
$ history
1 df -h
2 ll
3 /opt/seamonkey/seamonkey
4 dmesg|tail
5 ping hgeser.de
6 google-chrome
7 killall kded4
```

- Wie viele Einträge? Normal 500:

```
$ echo $HISTSIZE
500
```

- Idee beim Filter:
 - Standardeingabe in Standardausgabe verwandeln
 - Ketten aus Filtern zusammen bauen:
 - `prog1 | filter1 | filter2 | filter3 ...`
 - mit Eingabedatei:
`prog1 < eingabe | filter1 | ...`
- `cat`, `cut`, `expand`, `fmt`, `head`, `od`, `join`, `nl`, `paste`, `pr`, `sed`, `sort`, `split`, `tail`, `tr`, `unexpand`, `uniq`, `wc`

- Neben Ausgabe der kompletten History gibt es auch eine intelligente Suche nach alten Kommandos: [Strg-R]

```
$ # Suche nach dem letzten echo-Aufruf
$ ^R
(reverse-i-search)`ech': echo $HISTFILESIZE
```

- mit [Eingabe] ausführen
- weitere [Strg-R] liefern ältere Treffer
- Außerdem: Mit [Pfeil hoch], [Pfeil runter] durch alte Befehle blättern
- gefundenes Kommando kann übernommen und überarbeitet werden

- `cat` steht für **concatenate** (aneinanderfügen)
- gibt mehrere Dateien unmittelbar hintereinander aus
- auf Wunsch auch nur eine Datei
→ Mini-Dateibetrachter
- Spezialoptionen:
 - `-n` (Zeilennummern)
 - `-T` (Tabs als `^I` anzeigen)
 - ... und einige weitere (siehe: `man cat`)

- cut kann spaltenweise Text ausschneiden – Spalten sind wahlweise definierbar über
 - Zeichenpositionen
 - Trennzeichen (die logische Spalten voneinander trennen)

	c: character; ↓ zeichenbasiert	d: delimiter; ↓ Trennzeichen
<pre>\$ cat test.txt 1234 678901 234 abc def ghijklmn r2d2 12 99 1 2 3 Langer Testeintrag</pre>	<pre>\$ cut -c3-8 test.txt 34 678 c def d2 12 2 3 nger T</pre>	<pre>\$ cut -d" " -f2,3 test.txt 678901 234 def ghijklmn 12 99 2 3 Testeintrag</pre>

f: field (Feld)

- split kann große Dateien in mehrere Dateien mit angegebener Maximalgröße aufteilen
- (cat fügt diese anschließend wieder zusammen)

```
$ split ZM_ePaper_18_11.pdf -b1440k ZM_ePaper_18_11.pdf.
$ ls -l ZM*
-rw-r--r-- 1 esser esser 10551293 2011-04-29 06:58 ZM_ePaper_18_11.pdf
-rw-r--r-- 1 esser esser 1474560 2011-04-29 14:46 ZM_ePaper_18_11.pdf.aa
-rw-r--r-- 1 esser esser 1474560 2011-04-29 14:46 ZM_ePaper_18_11.pdf.ab
-rw-r--r-- 1 esser esser 1474560 2011-04-29 14:46 ZM_ePaper_18_11.pdf.ac
-rw-r--r-- 1 esser esser 1474560 2011-04-29 14:46 ZM_ePaper_18_11.pdf.ad
-rw-r--r-- 1 esser esser 1474560 2011-04-29 14:46 ZM_ePaper_18_11.pdf.ae
-rw-r--r-- 1 esser esser 1474560 2011-04-29 14:46 ZM_ePaper_18_11.pdf.af
-rw-r--r-- 1 esser esser 1474560 2011-04-29 14:46 ZM_ePaper_18_11.pdf.ag
-rw-r--r-- 1 esser esser 229373 2011-04-29 14:46 ZM_ePaper_18_11.pdf.ah
$ cat ZM_ePaper_18_11.pdf.* > ZM_Kopie.pdf
$ ls -l ZM_Kopie.pdf
-rw-r--r-- 1 esser esser 10551293 2011-04-29 14:48 ZM_Kopie.pdf
$ diff ZM_ePaper_18_11.pdf ZM_Kopie.pdf
$ _
```

- fmt (format) bricht Textdateien um

```
$ cat test.txt
Das ist mal ein Beispiel fuer einen Satz. Das ist mal ein Beispiel fue
r einen Satz. Das ist mal ein Beispiel fuer einen Satz. Das ist mal ei
n Beispiel fuer einen Satz. Das ist mal ein Beispiel fuer einen Satz.
Das ist mal ein Beispiel fuer einen Satz. Das ist mal ein Beispiel fue
r einen Satz. Das ist mal ein Beispiel fuer einen Satz. Das ist mal ei
n Beispiel fuer einen Satz. Das ist mal ein Beispiel fuer einen Satz.
```

```
$ fmt test.txt
Das ist mal ein Beispiel fuer einen Satz. Das ist mal ein Beispiel
fuer einen Satz. Das ist mal ein Beispiel fuer einen Satz. Das ist
mal ein Beispiel fuer einen Satz. Das ist mal ein Beispiel fuer
einen Satz. Das ist mal ein Beispiel fuer einen Satz. Das ist mal
ein Beispiel fuer einen Satz. Das ist mal ein Beispiel fuer einen
Satz. Das ist mal ein Beispiel fuer einen Satz. Das ist mal ein
Beispiel fuer einen Satz.
```

- Parameter -w75: Breite 75 (width)

keine
Umbrüche

Zeilen-
umbrüche

- sort ist ein komplexes Sortier-Tool, das
 - Sortierung nach *n*-ter Spalte
 - alphabetische und numerische Sortierung
- unterstützt
- Einfache Beispiele:

<pre>\$ cat test3.txt 13 Autos 5 LKW 24 Fahrraeder 2 Baeume Wohnung Haus Hotel Strasse Allee</pre>	<pre>\$ sort test3.txt 13 Autos 2 Baeume 24 Fahrraeder 5 LKW Allee Haus Hotel Strasse Wohnung</pre>	<pre>\$ sort -n test3.txt Allee Haus Hotel Strasse Wohnung 2 Baeume 5 LKW 13 Autos 24 Fahrraeder</pre>
--	---	--

- **uniq** (**u**nique, einmalig) fasst mehrere identische (aufeinander folgende) Zeilen zu einer zusammen; entfernt also Doppler
- Alternative: Beim Sortieren mit sort kann man über die Option -u (unique) direkt Doppler entfernen;
 - statt `sort datei | uniq` also besser `sort -u datei`

- **sed** (**S**tream **E**ditor) führt (u. a.) Suchen-/ Ersetzen-Funktionen in einem Text durch

```
$ cat test4.txt
Das Wort ist ein Wort, und mehrere
Woerter sind der Plural von Wort.
Ohne Woerter oder Worte gibt es
keinen Satz - wir sind wortlos.
```

```
$ sed 's/Wort/Bild/' test4.txt
Das Bild ist ein Wort, und mehrere
Woerter sind der Plural von Bild.
Ohne Woerter oder Bilde gibt es
keinen Satz - wir sind wortlos.
```

```
$ sed 's/Wort/FOM/g' test4.txt
Das FOM ist ein FOM, und mehrere
Woerter sind der Plural von FOM.
Ohne Woerter oder FOMe gibt es
keinen Satz - wir sind wortlos.
```

```
$ sed 's/Wort/FOM/gi' test4.txt
Das FOM ist ein FOM, und mehrere
Woerter sind der Plural von FOM.
Ohne Woerter oder FOMe gibt es
keinen Satz - wir sind FOMlos.
```

s: substitute (s/.../.../gi)
g: global (s/.../.../gi)
i: ignore case (s/.../.../gi)

Die i-Option gibt es nicht in jeder sed-Version!

- **grep** (**g**lobal/**r**egular **e**xpression/**p**rint) zeigt nur die Zeilen einer Datei, die einen Suchbegriff enthalten – oder nicht enthalten (Option -v)

```
$ wc -l /etc/passwd
57 /etc/passwd
$ grep esser /etc/passwd
esser:x:1000:1000:Hans-Georg Esser,,,:/home/esser:/bin/bash
$ grep /bin/bash /etc/passwd
root:x:0:0:root:/root:/bin/bash
esser:x:1000:1000:Hans-Georg Esser,,,:/home/esser:/bin/bash
$ grep -v /bin/bash /etc/passwd | head -n5
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
```

- **sed**-Optionen:
 - -i: in-place-editing, verändert die angegebene Datei; am besten mit Angabe eines Suffix für eine Backup-Datei:
z. B. `sed -i.bak 's/Wort/Bild/g' test4.txt` legt erst Sicherheitskopie test4.txt.bak an und verändert dann test4.txt
 - -e: zum Kombinieren mehrerer Ersetzungen; z. B. `sed -e 's/1/eins/g' -e 's/2/zwei/g' test.txt`
 - weitere Optionen → Manpage

- Idee: Allgemeinere Suchbegriffe, vergleichbar mit Wildcards (*, ?) bei Dateinamen
- Muster:
 - . – ein beliebiges Zeichen
 - [abcd] – eines der Zeichen a, b, c, d
 - [2-8] – eines der Zeichen 2, 3, 4, 5, 6, 7, 8
 - ^ – Zeilenanfang
 - \$ – Zeilenende
 - ? – vorheriger Ausdruck darf vorkommen, muss aber nicht
 - * – vorheriger Ausdruck kann beliebig oft (auch 0 mal) vorkommen

<pre>\$ cat test5.txt Haus Die Hotels Hotels am Wasser Bau-Haus-Objekt Diese Zeile nicht</pre>	<pre>\$ grep 'H.*s' test5.txt Haus Die Hotels Hotels am Wasser Bau-Haus-Objekt</pre>	<pre>\$ sed 's/H.*s/HAUS/g' test5.txt HAUS Die HAUS HAUSer Bau-HAUS-Objekt Diese Zeile nicht</pre>
--	--	--

C-Compiler

- Beispiele für reguläre Ausdrücke (live, in der Shell...)

- Quellcode-Datei `prog.c` im Editor erstellen
- In der Shell

```
gcc -o prog prog.c
```

eingeben, um zu kompilieren
- In der Shell

```
./prog
```

eingeben (wichtig: „./“ am Anfang), um das übersetzte Programm zu starten

- Vorab das wichtigste:
 - keine Klassen / Objekte
 - statt Objekten:
„structs“ (zusammengesetzte Datentypen)
 - statt Methoden nur Funktionen
 - zu bearbeitende Variablen immer als Argument übergeben
 - kein String-Datentyp (sondern Zeichen-Arrays)
 - häufiger Einsatz von Zeigern
 - `int main () {}` ist immer Hauptprogramm

- Im Anschluss an diese Vorlesung: erstes Übungsblatt mit C-Aufgaben
- Vorbereitend ein paar Informationen zu
 - **Structs** (Strukturen, zusammengesetzte Typen)
 - **Pointern**
 - **Quellcode- und Header-Dateien** (Prototypen)

- Ausführlichere Informationen fürs Selbststudium: <http://www.c-howto.de/>
 - auf der Webseite: ausführlichere Version mit erklärenden Kommentaren (Download: Zip-Archiv)
- auch als Buch für ca. 20 € erhältlich
- in der Vorlesung/Übung: Fokus auf Unterschiede zu C++/C#/Java
- keine komplexen C-Programme im Kurs

Structs

- Mehrere Möglichkeiten der Deklaration

```
struct {  
    int i;  
    char c;  
    float f;  
} variable;  
  
variable.i = 9;  
variable.c = 'a';  
variable.f = 0.123;
```

```
struct mystruct {  
    int i;  
    char c;  
    float f;  
};  
  
struct mystruct variable;  
  
variable.i = 9;  
variable.c = 'a';  
variable.f = 0.123;
```

```
typedef struct {  
    int i;  
    char c;  
    float f;  
} mystruct;  
  
mystruct variable;  
  
variable.i = 9;  
variable.c = 'a';  
variable.f = 0.123;
```

Pointer

- Deklaration mit *: `char *ch_ptr;`
- verwalten Speicheradressen (an welchem Ort befindet sich die Variable?)

- Operatoren

- & (Adresse von)
- * (Dereferenzieren)

```
char ch, ch2;
char *ch_ptr; char *ch_ptr2;

ch_ptr = &ch; // Adresse von ch?
ch2 = *ch_ptr; // Inhalt

ch_ptr2 = ch_ptr;
// kopiert nur Adresse
```

• Pointer-Typen

- `typ *ptr;`
→ `ptr` ist ein Zeiger auf etwas vom Typ `typ`
- `typ **pptr;`
→ `pptr` ist ein Zeiger auf einen Zeiger vom Typ `typ`
- `ptr` bzw. `pptr` sind Speicheradressen
- `*ptr` gibt den Wert zurück, der an der Speicherstelle abgelegt ist, auf die `ptr` zeigt
- analog: `**pptr` ist ein Wert, aber `*pptr` ein Zeiger

- Struct und Pointer kombiniert
- Oft bei verketteten Listen

```
struct liste {
    struct liste *next;
    struct liste *prev;
    int inhalt;
};

struct liste *anfang;
struct liste *p;

for (p=anfang; p != NULL; p=p->next) {
    use (p->inhalt);
}
```

• Pointer-Typen

- &-Operator erzeugt zu Variable einen Pointer
- Beispiele:

```
int i;
int *ip;
int **ipp;

i = 42;
ip = &i;           // ip = Adresse von i
ipp = &ip;         // ipp = Adresse von ip

printf (*ip);      // -> 42
printf (**ipp);    // -> auch 42
```

- Nicht-initialisierte Pointer: schlecht

- Beispiel:

```
int *ip;
int **ipp;

printf (ip);    // nicht-init. Adresse (0)
printf (*ip);   // illegal -> Abbruch

*ip = 42;       // auch illegal, schreibt an
                // nicht def. Adresse
```

- Programm- und Header-Dateien

- Header-Dateien (*.h) enthalten Funktionsprototypen und Makrodefinitionen (aber keinen normalen Code)
 - Programmdateien (*.c) enthalten den Code, können aber ebenfalls Prototypen und Makros enthalten (kein Zwang, eine .h-Datei zu erzeugen)

- Vorsicht bei `char* a,b,c;` etc.

```
[esser@macbookpro:tmp]$ cat t2.c
int main () {
    char* a,b;
    printf ("|a| = %d \n", sizeof(a));
    printf ("|b| = %d \n", sizeof(b));
}

[esser@macbookpro:tmp]$ gcc t2.c; ./a.out
|a| = 8
|b| = 1
```

- besser: `char *a, *b, *c;`

- Funktionsprototypen (in Header-Dateien)

- erlauben die Verwendung von Funktionen, deren Implementierung weiter unten im Programm (oder in einer anderen Datei) steht
 - Prototyp enthält nur Rückgabotyp, Name und Argumente, z. B.
`int summe (int x, int y);`

- Wie findet der Compiler die Header-Dateien?
→ Zwei Varianten:
 - `#include "pfad/zu/datei.h"`
Dateiname ist Pfad (relativ zu Verzeichnis mit der .c-Datei)
 - `#include <name.h>`
name.h wird in den Standard-Include-Verzeichnissen gesucht.
Welche sind das? Beim Bauen des gcc festgelegt...

- Standard-Include-Verzeichnisse

```
[esser@s15337257:~]$ cpp -v
Using built-in specs.
Target: i486-linux-gnu
[...]
#include "..." search starts here:
#include <...> search starts here:
  /usr/local/include
  /usr/lib/gcc/i486-linux-gnu/4.4.5/include
  /usr/lib/gcc/i486-linux-gnu/4.4.5/include-fixed
  /usr/include
End of search list.
(und deren Unterordner)
```

Betriebssysteme

SS 2015

Hans-Georg Eßer
Dipl.-Math., Dipl.-Inform.

Foliensatz C:

- Prozesse und Threads

v1.1, 2015/02/01

Prozesse (Theorie)

Übersicht: BS Praxis und BS Theorie

Einführung

Shell

Dateiverwaltung

Filter

C-Compiler

Prozesse / Jobs

Threads

Interrupts

System Calls

Software-Verwaltung

Scheduler / Prioritäten

Synchronisation

Folien C

Prozesse & Threads: Gliederung

Vorlesung:

- Theorie / Grundlagen
- Prozesse & Threads im Linux-Kernel

Praktikum:

- Prozesse auf der Linux-Shell
- Prozesse in C-Programmen
- Threads in C-Programmen

Single-Tasking / Multitasking:

Wie viele Programme laufen „gleichzeitig“?

- MS-DOS, CP/M: 1 Programm
- Windows, Linux, ...: Viele Programme

Single-Processing / Multi-Processing:

Hilft der Einsatz mehrerer CPUs?

- Windows 95/98/Me: 1 CPU
- Windows 2000, XP, Vista, 7, Linux, Mac OS, ...: Mehrere CPUs

Prozess:

- Konzept nötig, sobald >1 Programm läuft
- Programm, das der Rechner ausführen soll
- Eigene Daten
- von anderen Prozessen abgeschottet
- Zusätzliche Verwaltungsdaten

MS-DOS:

- Betriebssystem startet, aktiviert Shell
COMMAND.COM
- Anwender gibt Befehl ein
- Falls kein interner Befehl:
Programm laden und aktivieren
- Nach Programmende: Rücksprung zu
COMMAND.COM

Kein Wechsel zwischen mehreren Programmen

Prozessliste:

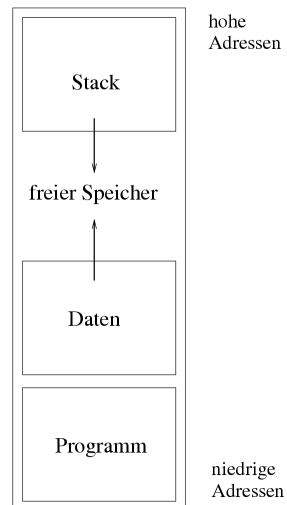
- Informationen über alle Prozesse und ihre Zustände
- Jeder Prozess hat dort einen **Process Control Block (PCB)**:
 - Identifier (PID)
 - Registerwerte inkl. Befehlszähler
 - Speicherbereich des Prozess
 - Liste offener Dateien und Sockets
 - Informationen wie Vater-PID, letzte Aktivität, Gesamtlaufzeit, Priorität, ...

Prozess im Detail:

- Eigener Adressraum
- Ausführbares Programm
- Aktuelle Daten (Variableninhalte)
- Befehlszähler (Program Counter, PC)
- Stack und Stack-Pointer
- Inhalt der Hardware-Register (Prozess-Kontext)

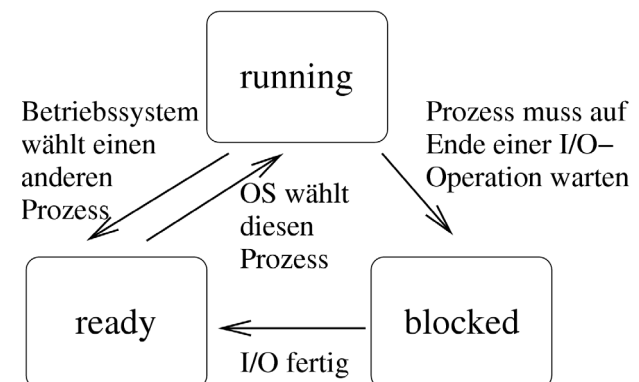
Zustände

- **laufend / running**: gerade aktiv
- **bereit / ready**: würde gerne laufen
- **blockiert / blocked / waiting**: wartet auf I/O
- **suspendiert**: vom Anwender unterbrochen
- **schlafend / sleeping**: wartet auf Signal (IPC)
- **ausgelagert / swapped**: Daten nicht im RAM

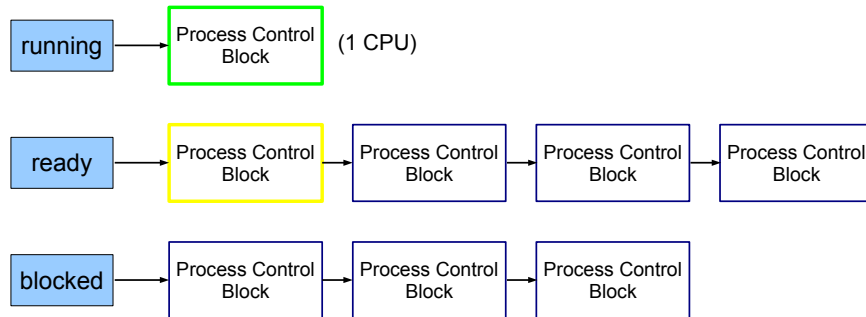


- Daten: dynamisch erzeugt
- Stack: Verwaltung der Funktionsaufrufe
- Details: siehe Kapitel Speicherverwaltung
- Stack und Daten „wachsen aufeinander zu“

Zustandsübergänge

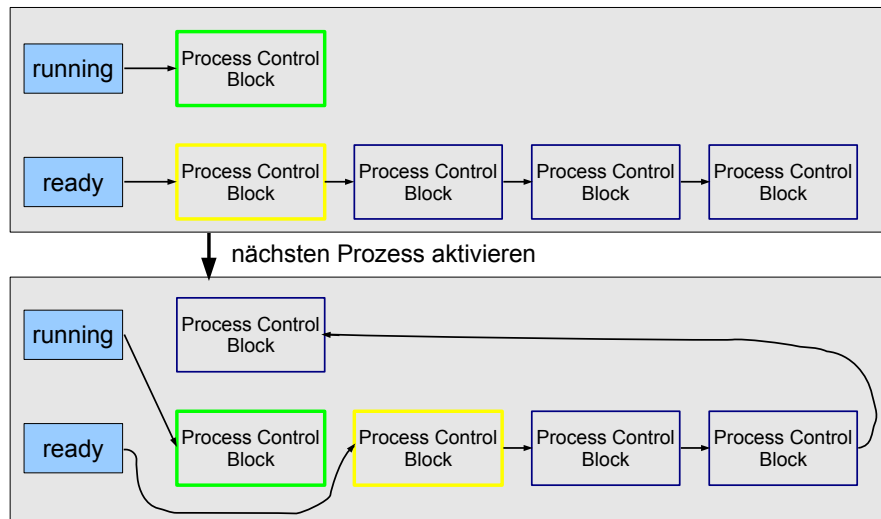


Prozesslisten



Hierarchien

- Prozesse erzeugen einander
- Erzeuger heißt Vaterprozess (parent process), der andere Kindprozess (child process)
- Kinder sind selbständig (also: eigener Adressraum, etc.)
- Nach Prozess-Ende: Rückgabewert an Vaterprozess



Prozesse (Praxis, Anwenderperspektive)

- In der Shell gestartete Anwendungen laufen standardmäßig im **Vordergrund** – d. h.,
 - die Shell ist blockiert, solange das Programm läuft,
 - und es nutzt das aktuelle Terminal (-Fenster) für Ein- und Ausgabe
- Alternativ kann ein Programm im **Hintergrund** laufen:
 - die Shell kann dann sofort weiter genutzt werden (weitere Kommando eingeben),
 - keine Eingabe möglich, aber Ausgabe (auch ins aktuelle Terminal; besser umleiten)

- Programm im Vordergrund starten:
einfach den Namen eingeben
Bsp.: `ls -l`
- Programm im Hintergrund starten:
kaufmännisches Und (&, ampersand) anhängen
Bsp.: `/usr/sbin/apache2 &`
- Wechsel von Vordergrund in Hintergrund:
 - Programm mit [Strg-Z] unterbrechen
 - Programm mit `bg` in den Hintergrund schicken
- Wechsel von Hinter- in Vordergrund: `fg`

- Typische Vordergrund-Programme
 - Kommandos, die eine Anfrage sofort beantworten
 - Text-Editoren
 - Compiler
- Typische Hintergrund-Programme
 - manuell gestartete Server (Dienste)
 - unter X Window: grafische Anwendungen (die kein Terminal brauchen, sondern ein eigenes Fenster öffnen)

- Programme, die aus einer laufenden Shell heraus gestartet wurden, heißen **Jobs** dieser Shell
- Anzeige mit: `jobs`

```
[esser@macbookpro:~]$ jobs
[esser@macbookpro:~]$ nedit &
[1] 77787
[esser@macbookpro:~]$ vi /tmp/test.txt
^Z
[2]+  Stopped                  vi /tmp/test.txt
[esser@macbookpro:~]$ find / > /tmp/ergebnisse.txt &
[3] 77792
[esser@macbookpro:~]$ jobs
[1]  Running                    nedit &
[2]+  Stopped                  vi /tmp/test.txt
[3]-  Running                  find / > /tmp/ergebnisse.txt &
[esser@macbookpro:~]$
```

Zustand des Jobs:

- running: aktiv / bereit
- stopped: mit ^Z oder kill -STOP angehalten
- terminated: beendet, wird nur 1x angezeigt

```
[esser@macbookpro:~]$ jobs
[1]    Running      nedit &
[2]+  Stopped      vi /tmp/test.txt
[3]-  Running      find / > ...&
```

Kommandos

1,2,3, ...: Job-Nummer

+ : „current job“ = letzter Job, der
 - im Vordergrund gestartet und dann unterbrochen
 - oder im Hintergrund gestartet wurde
 viele Kommandos (fg, bg, ...) ohne Parameter
 beziehen sich auf den current job
 - : „previous job“ = vorletzter Job mit obiger Eigenschaft

Kommandos zur Job-Verwaltung

- bg %n: in den Hintergrund bringen
- fg %n: in den Vordergrund bringen
- kill %n: beenden
- kill -*SIGNALNAME* %n: Signal schicken, siehe nächste Folie
- disown %n: Verbindung mit der Shell lösen; disown -a: für alle Jobs
- wait %n: Warten, bis Job beendet ist

- Jobs gezielt ansprechen: %n (mit n = Job-Nummer)

```
[esser@macbookpro:~]$ jobs
[1]    Running      nedit /tmp/1 &
[2]    Running      nedit /tmp/2 &
[3]    Running      nedit /tmp/3 &
[4]-  Running      nedit /tmp/4 &
[5]+  Running      nedit /tmp/5 &
[esser@macbookpro:~]$ kill %3
[esser@macbookpro:~]$ jobs
[1]    Running      nedit /tmp/1 &
[2]    Running      nedit /tmp/2 &
[3]    Terminated  nedit /tmp/3
[4]-  Running      nedit /tmp/4 &
[5]+  Running      nedit /tmp/5 &
[esser@macbookpro:~]$ jobs
[1]    Running      nedit /tmp/1 &
[2]    Running      nedit /tmp/2 &
[4]-  Running      nedit /tmp/4 &
[5]+  Running      nedit /tmp/5 &
```

Signale (mit Signalnummer)

- TERM, 15: terminieren, beenden (mit „Aufräumen“); Standardsignal
- KILL, 9: sofort abbrechen (ohne Aufräumen)
- STOP, 19: unterbrechen (entspricht ^Z)
- CONT, 18: continue, fortsetzen; hebt STOP auf
- HUP, 1: hang-up, bei vielen Server-Programmen: Konfiguration neu einlesen (traditionell: Verbindung zum Terminal unterbrochen)
- Liste aller Signale: kill -l

```
$ kill -l
 1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL
 5) SIGTRAP     6) SIGABRT    7) SIGBUS      8) SIGFPE
 9) SIGKILL    10) SIGUSR1   11) SIGSEGV    12) SIGUSR2
13) SIGPIPE    14) SIGALRM   15) SIGTERM    16) SIGSTKFLT
17) SIGCHLD    18) SIGCONT   19) SIGSTOP    20) SIGTSTP
21) SIGTTIN    22) SIGTTOU   23) SIGURG     24) SIGXCPU
25) SIGXFSZ    26) SIGVTALRM 27) SIGPROF    28) SIGWINCH
29) SIGIO      30) SIGPWR    31) SIGSYS     34) SIGRTMIN
35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3 38) SIGRTMIN+4
39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

- `ps` (ohne Optionen) zeigt alle Prozesse an, die zur aktuellen Shell-Sitzung gehören – das sind dieselben wie in der Ausgabe von `jobs`:

```
[esser@quadamd:~]$ jobs
[1]+  Angehalten    vi /tmp/test4

[esser@quadamd:~]$ ps
  PID TTY          TIME CMD
 27967 pts/0        00:00:00 bash
 28160 pts/0        00:00:00 vi
 28168 pts/0        00:00:00 ps
```

- über Optionen (ohne „-“) lässt sich die Ausgabe von `ps` anpassen, z. B. `ps auxw`:
 - a: alle Prozesse (die ein Terminal haben)
 - u: „user oriented format“
 - x: auch Prozesse ohne Terminal
 - w: „wide“: Befehlszeilen nicht abschneiden

- Die Bezeichnung **Job** bezieht sich immer auf die aktuelle Shell-Sitzung
- Jobs, die Sie in verschiedenen Shells starten, haben nichts miteinander zu tun
- Allgemeinerer Begriff: **Prozess**
- Tool für die Prozessanzeige: `ps`
- Die (Gesamt-) Prozessliste (`ps auxw`) enthält alle Prozesse auf dem Linux-System

Prozess-IDs in der Job-Liste

```
esser@sony:Folien> jobs
[1]-  Running                  xpdf -remote sk bs02.pdf &
[2]+  Running                  nedit kap02/index.tex &

esser@sony:Folien> jobs -l
[1]-  8103 Running              xpdf -remote sk bs02.pdf &
[2]+ 20568 Running              nedit kap02/index.tex &

esser@sony:Folien> ps w|grep 8103|grep -v grep
 8103 pts/15 S      5:27 xpdf -remote sk bs02.pdf
```

```
[esser@quadamd:~]$ ps aux
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root      1122  0.0  0.0   1872   580 tty4      Ss+  Apr17   0:00 /sbin/getty -8 38400 tty4
root      1127  0.0  0.0   1872   580 tty5      Ss+  Apr17   0:00 /sbin/getty -8 38400 tty5
root      1150  0.0  0.0   1872   576 tty2      Ss+  Apr17   0:00 /sbin/getty -8 38400 tty2
root      1151  0.0  0.0   1872   576 tty3      Ss+  Apr17   0:00 /sbin/getty -8 38400 tty3
root      1153  0.0  0.0   1872   580 tty6      Ss+  Apr17   0:00 /sbin/getty -8 38400 tty6
root      1776  0.0  0.0   5248  2156 tty1      Ss   Apr17   0:00 /bin/login --
esser     1941  0.0  0.1   9272  4816 tty1      S    Apr17   0:01 -bash
esser     2956  0.0  0.0   1912   540 tty1      S    Apr17   0:00 /bin/sh /usr/bin/startx
esser     2973  0.0  0.0   3128   708 tty1      S+   Apr17   0:00 xinit /home/esser/.xinitrc -- /etc/X11/xini
root      2974  0.0  1.8  85616 75712 tty8      Ss+  Apr17  14:56 /usr/bin/X -nolisten tcp :0 -auth /tmp/ser
esser     2977  0.0  0.0   1912   564 tty1      S    Apr17   0:00 /bin/sh /home/esser/.xinitrc
esser     3037  0.0  0.0   3456   560 tty1      S    Apr17   0:00 dbus-launch --sh-syntax --exit-with-session
esser     3045  0.0  0.0   1700    64 tty1      S    Apr17   0:00 /usr/lib/kde4/libexec/start_kdeinit +kcmi
esser     3174  0.0  0.0   1832   244 tty1      S    Apr17   0:00 kwrapper4 ksmserver
esser     3454  0.0  0.0   6552  2040 pts/2    Ss   Apr17   0:00 /bin/bash
root      3775  0.0  0.0   8560  2112 pts/2    S    Apr17   0:00 sudo su
root      3776  0.0  0.0   8316  1764 pts/2    S    Apr17   0:00 su
root      3784  0.0  0.0   6656  2172 pts/2    S+   Apr17   0:00 bash
esser     10674  1.4  7.0  443588 289940 pts/6    sl   Apr30  13:17 /usr/lib/opera/opera
esser     10694  0.0  0.1  21764   7552 pts/6    S    Apr30   0:08 /usr/lib/opera//operapluginwrapper 101 110
esser     10695  0.0  0.0   3040    548 pts/6    S    Apr30   0:02 /usr/lib/opera//operapluginwrapper 10674
esser     10699  1.3  0.0    0      0 pts/6    Z    Apr30  119:12 [gtk-gnash] <defunct>
esser     12198  0.0  0.0   6552  1828 pts/4    Ss   Apr17   0:00 /bin/bash
root      12583  0.0  0.0   8560  2116 pts/4    S    Apr17   0:00 sudo su
root      13077  0.0  0.0   8316  1768 pts/4    S    Apr17   0:00 su
root      13097  0.0  0.0   6612  2052 pts/4    S+   Apr17   0:00 bash
esser     13653  0.0  0.0   6768  2140 pts/3    Ss   Apr17   0:00 /bin/bash
esser     18905  0.0  0.1   9128  4712 pts/6    Ss+  Apr22   0:02 /bin/bash
esser     27587  0.0  0.0   5256  2144 pts/3    S+   Apr17  19:40 0:00 ssh backup
esser     28613  0.0  0.1  11744  7264 pts/5    Ss+  Apr17  22:45 0:00 -bash
esser     29091  0.0  0.1  12156  7704 pts/0    Ss   Apr17  22:58 0:00 -bash
esser     29307  0.0  0.0   5856  1172 pts/0    R+   Apr17  23:42 0:00 ps aux
```

- Signale an beliebige Prozesse schicken
 - wie vorher: Kommando `kill`
 - aber: nicht `kill %n` (n =Job-ID), sondern `kill p` (p = PID)
 - auch hier Angabe eines Signals möglich
- `killall Name`: alle Prozesse beenden, deren ausführbares Programm *Name* heißt
- mit `killall` auch (wie bei `kill`) andere Signale an alle Prozesse mit passendem Namen schicken

- Spalten in der Ausgabe von `ps aux`:
 - USER: Benutzer, dem der Prozess gehört
 - PID: Prozess-ID
 - %CPU: CPU-Nutzung in Prozent (Verhältnis Rechenzeit / Lebenszeit)
 - %MEM: RSS / RAM-Größe in Prozent
 - VSZ: Größe des virtuellen Speichers (in KByte)
 - RSS: Resident Set Size, aktuell genutzter Speicher (KByte)
 - TTY: Terminal
 - STAT: Prozess-Status
 - START: Startzeit des Prozesses (ggf. Datum)
 - TIME: bisherige Gesamtlaufzeit
 - COMMAND: Kommando (Aufruf)

- Darstellung der Prozessliste auch in Baumansicht möglich: `pstree`
- Jeder Prozess hat einen Vater-prozess
- identische Teilbäume nur 1x
- Option `-p`: Prozess-IDs anzeigen

```
[esser@quadamd:~]$ pstree
init--+-NetworkManager--+-dhclient
|
|_acpid
|_akonadi_control--+-2*[akonadi_contact]
|                   |_3*[akonadi_ical_re]
|                   |_akonadi_maildir
|                   |_akonadi_maildis
|                   |_akonadi_nepomuk
|                   |_akonadi_vcard_r
|                   |_akonadiserver--+-mysqld--23*[mysqld]
|                   |                   ^-15*[akonadiserver]}
|                   ^-3*[akonadi_control]}
|_atd
|_avahi-daemon--+-avahi-daemon
|_console-kit-dae--64*[console-kit-da)}
|_cron
|_cupsd
[...]
|_knotify4--6*[knotify4]}
|_konsole--2*[bash--sudo--su--bash]
|   |_bash--ssh
|   |_bash--opera--+-operapluginlea
|   |                 |_operapluginwrap--gtk-gnash
|   |                 ^-6*[opera]}
|   ^-2*[konsole]}
|_krunker--11*[krunker]}
|_kuiserver
|_kwalletd
|_login--bash--startx--xinit--+-xinitrc--kwrapper4
|                               ^-Xorg
|_upstart-socket-
|_upstart-udev-br
|_vpnaagentd
|_wpa_supplicant
```

- Wenn Sie sich in der Konsole abmelden (`exit`) oder unter X Window ein Terminalfenster schließen, erhalten alle in der Shell laufenden Jobs das HUP-Signal (Hang-up).
- Die Standardreaktion auf HUP ist: beenden
- Abmelden / Fenster schließen beendet also alle darin gestarteten Programme
- Auswege:
 - Programme mit `nohup` starten oder
 - Prozess mit `disown` von der Shell lösen

- Prozesse nach CPU-Auslastung sortiert anzeigen: `top`
- Anzeige wird regelmäßig aktualisiert

```
top - 00:07:30 up 19 days, 6:15, 7 users, load average: 0.00, 0.02, 0.05
Tasks: 194 total, 2 running, 191 sleeping, 0 stopped, 1 zombie
Cpu(s): 1.2%us, 0.7%sy, 0.0%ni, 98.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 4120180k total, 2353392k used, 1766788k free, 560756k buffers
Swap: 4191936k total, 0k used, 4191936k free, 566868k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
3177	esser	20	0	302m	45m	20m	S	2	1.1	260:37.29	knotify4
10674	esser	20	0	433m	283m	27m	S	2	7.0	131:47.89	opera
3432	esser	20	0	893m	155m	33m	S	1	3.9	253:02.73	seamonkey-bin
1093	messageb	20	0	4524	2664	840	R	0	0.1	16:55.65	dbus-daemon
1576	root	20	0	12072	7272	3408	S	0	0.2	8:50.93	python
2076	root	20	0	5560	972	692	S	0	0.0	5:07.52	udisks-daemon
3238	esser	20	0	367m	64m	35m	S	0	1.6	1:13.69	krunker
29348	esser	20	0	2632	1180	852	R	0	0.0	0:00.03	top
1	root	20	0	3028	1892	1236	S	0	0.0	0:02.06	init
2	root	20	0	0	0	0	S	0	0.0	0:00.73	kthreadd
3	root	20	0	0	0	0	S	0	0.0	1:36.46	ksftirgq/0
6	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/0
17	root	0	-20	0	0	0	S	0	0.0	0:00.00	cpuset
18	root	0	-20	0	0	0	S	0	0.0	0:00.00	khelper
19	root	0	-20	0	0	0	S	0	0.0	0:00.00	netns
21	root	20	0	0	0	0	S	0	0.0	0:02.28	sync_supers
22	root	20	0	0	0	0	S	0	0.0	0:00.05	bdi-default
23	root	0	-20	0	0	0	S	0	0.0	0:00.00	kintegrityd
24	root	0	-20	0	0	0	S	0	0.0	0:00.00	kblockd
25	root	0	-20	0	0	0	S	0	0.0	0:00.00	kacpid
26	root	0	-20	0	0	0	S	0	0.0	0:00.00	kacpi_notify
27	root	0	-20	0	0	0	S	0	0.0	0:00.00	kacpi_hotplug

- `nohup` hat zwei Funktionen:
 - der gestartete Prozess ignoriert HUP-Signale
 - Ausgaben des Prozesses (auf die Standardausgabe) erscheinen nicht im Terminal, sondern werden in die Datei `nohup.out` geschrieben

```
[esser@macbookpro:~]$ nedit /tmp/1 &
[1] 79142
[esser@macbookpro:~]$ nohup nedit /tmp/2 &
[2] 79144
appending output to nohup.out
```

- Sortierung in `top` anpassbar (Sortierspalte ändern mit `<` und `>`)
- Über der Prozessliste: Informationen zur Gesamtauslastung des Systems
- umschaltbar auf Anzeige/CPU bzw. /Kern: 1

```
top - 00:14:22 up 19 days, 6:22, 7 users, load average: 0.05, 0.03, 0.05
Tasks: 194 total, 2 running, 191 sleeping, 0 stopped, 1 zombie
Cpu0 : 0.7%us, 0.3%sy, 0.0%ni, 99.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu1 : 1.7%us, 1.0%sy, 0.0%ni, 97.3%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu2 : 0.0%us, 0.3%sy, 0.0%ni, 99.7%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu3 : 3.6%us, 0.7%sy, 0.0%ni, 95.8%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 4120180k total, 2353400k used, 1766780k free, 560948k buffers
Swap: 4191936k total, 0k used, 4191936k free, 566868k cached
```

- Weitere Systeminformationen:
 - `free` (freien Speicher anzeigen)

```
[esser@quadamd:~]$ free
              total        used        free      shared    buffers     cached
Mem:      4120180     2347264     1772916           0       561488     566896
-/+ buffers/cache:    1218880     2901300
Swap:      4191936           0       4191936
```

- `uptime` (wie lange läuft das System schon?)

```
[esser@quadamd:~]$ uptime
00:34:08 up 19 days, 6:42, 6 users, load average: 0.06, 0.07, 0.05
```

- `nice` mit Priorität (Option) und auszuführendem Kommando (folgende Argumente) aufrufen, z. B.

```
[esser@quadamd:~]$ nice -5 program &

[esser@quadamd:~]$ ps -eo user,pid,ni,cmd
USER      PID  NI  CMD
...
root      28299   0 [kworker/2:0]
root      28300   0 [kworker/0:1]
esser     28301   5 program
esser     28303   0 ps -eo user,pid,ni,cmd
```

- negative Nice-Werte kann nur Administrator `root` setzen:

```
[esser@quadamd:~]$ nice --10 vi
nice: kann Priorität nicht setzen: Keine Berechtigung
```

- Jeder Linux-Prozess hat eine **Priorität**. Diese bestimmt, welchen Anteil an Rechenzeit der Prozess erhält.
- Priorität ist ein Wert zwischen -20 und 19.
- Konvention: hohe Priorität = kleiner Wert (also: -20 = maximale Prior., 19 = minimale Prior.)
- unter Linux/Unix auch als **nice value** („Nettigkeit“) bezeichnet: 19 = extrem nett, -20 = gar nicht nett
- Bei Programmstart Priorität mit `nice` setzen

- Alternative Syntax für bessere Lesbarkeit:
`nice -n Wert` (statt `nice -Wert`)
- vor allem für negative Werte intuitiver:
`nice -n -10` (statt `nice --10`)

```
[esser@quadamd:~]$ su
Passwort:
root@quadamd:~# nice -n -10 program &
[1] 28373
root@quadamd:~# ps -eo user,pid,ni,cmd
USER      PID  NI  CMD
...
root      28311   0 su
root      28319   0 bash
root      28373  -10 program
root      28375   0 ps -eo user,pid,ni,cmd
```

- Genauer: Nice-Wert in `nice`-Aufruf ist relativ zum „aktuellen Nice-Level“ (Standard: 0)
- angegebener Wert wird zum Nice-Wert addiert:

```
[esser@quadamd:~]$ nice
0
[esser@quadamd:~]$ nice -n 5 bash
[esser@quadamd:~]$ nice
5
[esser@quadamd:~]$ nice -n 10 bash
[esser@quadamd:~]$ nice
15
[esser@quadamd:~]$ _
```

Threads (Theorie)

- Nice-Wert für laufendes Programm ändern: `renice`
- Wert <0 setzen darf nur *root*
- in alten Linux-Versionen galt auch: aktuellen Wert verringern darf nur *root*)

```
[esser@quadamd:~]$ program &
[5] 28937
[esser@quadamd:~]$ ps -eo user,pid,ni,cmd
USER      PID  NI  CMD
esser    28937   0  program
[esser@quadamd:~]$ renice 5 28937
28937: Alte Priorität: 0, neue Priorität: 5
[esser@quadamd:~]$ ps -eo user,pid,ni,cmd
USER      PID  NI  CMD
esser    28937   5  program
[esser@quadamd:~]$ renice 0 28937
28937: Alte Priorität: 5, neue Priorität: 0
[esser@quadamd:~]$ renice -10 28937
renice: 28937: setpriority: Keine Berechtigung
```

Was ist ein Thread?

- Aktivitätsstrang in einem Prozess
- einer von mehreren
- Gemeinsamer Zugriff auf Daten des Prozess
- aber: Stack, Befehlszähler, Stack Pointer, Hardware-Register separat pro Thread
- Prozess-Scheduler verwaltet Threads – oder nicht (Kernel- oder User-level-Threads)

Warum Threads?

- Multi-Prozessor-System: Mehrere Threads echt gleichzeitig aktiv
- Ist ein Thread durch I/O blockiert, arbeiten die anderen weiter
- Besteht Programm logisch aus parallelen Abläufen, ist die Programmierung mit Threads einfacher

Komplexe Berechnung mit Benutzeranfragen

Mit Threads:

```
T1:
while (1) {
    rechne_alles ();
}
```

```
T2:
while(1) {
    if benutzereingabe (x) {
        bearbeite_eingabe (x);
    }
}
```

Zwei unterschiedliche Aktivitätsstränge: Komplexe Berechnung mit Benutzeranfragen

Ohne Threads:

```
while (1) {
    rechne_ein_bisschen ();
    if benutzereingabe (x) {
        bearbeite_eingabe (x);
    }
}
```

Server-Prozess, der viele Anfragen bearbeitet

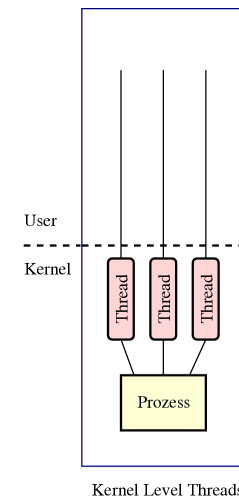
- Prozess öffnet Port
- Für jede eingehende Verbindung: Neuen Thread erzeugen, der diese Anfrage bearbeitet
- Nach Verbindungsabbruch Thread beenden
- Vorteil: Keine Prozess-Erzeugung (Betriebssystem!) nötig

Ein Prozess, neun Threads:

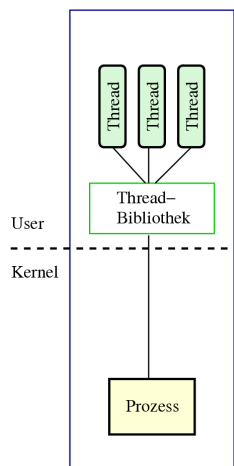
```
[esser:~]$ ps -elf | grep mysql
UID      PID  PPID  LWP  C  NLWP STIME TTY          TIME CMD
root      27833    1 27833  0   1 Jan04 ?        00:00:00 /bin/sh /usr/bin/mysql_safe
mysql     27870 27833 27870  0   9 Jan04 ?        00:00:00 /usr/sbin/mysqld --basedir=/usr
mysql     27870 27833 27872  0   9 Jan04 ?        00:00:00 /usr/sbin/mysqld --basedir=/usr
mysql     27870 27833 27873  0   9 Jan04 ?        00:00:00 /usr/sbin/mysqld --basedir=/usr
mysql     27870 27833 27874  0   9 Jan04 ?        00:00:00 /usr/sbin/mysqld --basedir=/usr
mysql     27870 27833 27875  0   9 Jan04 ?        00:00:00 /usr/sbin/mysqld --basedir=/usr
mysql     27870 27833 27876  0   9 Jan04 ?        00:00:00 /usr/sbin/mysqld --basedir=/usr
mysql     27870 27833 27877  0   9 Jan04 ?        00:00:00 /usr/sbin/mysqld --basedir=/usr
mysql     27870 27833 27878  0   9 Jan04 ?        00:00:00 /usr/sbin/mysqld --basedir=/usr
mysql     27870 27833 27879  0   9 Jan04 ?        00:00:00 /usr/sbin/mysqld --basedir=/usr

[esser:~]$
```

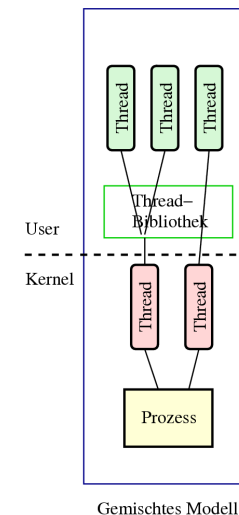
PID: Process ID
 PPID: Parent Process ID
 LWP: Light Weight Process ID (Thread-ID)
 NLWP: Number of Light Weight Processes



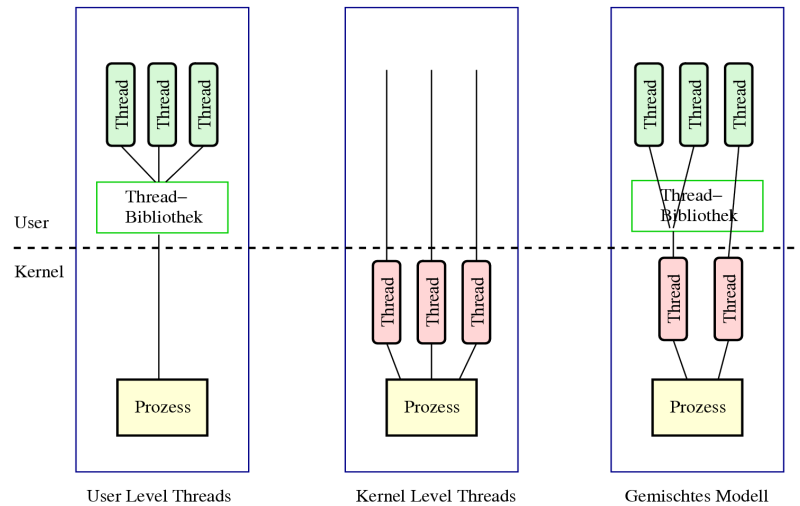
- BS kennt Threads
- BS verwaltet die Threads:
 - Erzeugen, Zerstören
 - Scheduling
- I/O eines Threads blockiert nicht die übrigen
- Aufwendig: Context Switch zwischen Threads ähnlich komplex wie bei Prozessen



- BS kennt kein Thread-Konzept, verwaltet nur Prozesse
- Programm bindet Thread-Bibliothek ein, zuständig für:
 - Erzeugen, Zerstören
 - Scheduling
- Wenn ein Thread wegen I/O wartet, dann der ganze Prozess
- Ansonsten sehr effizient



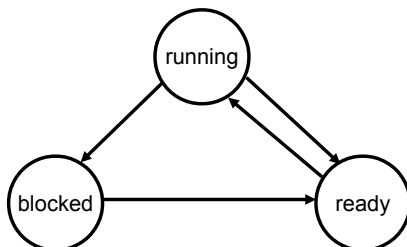
- Beide Ansätze kombinieren
- KL-Threads + UL-Threads
- Thread-Bibliothek verteilt UL-Threads auf die KL-Threads
- z.B. I/O-Anteile auf einem KL-Thread
- Vorteile beider Welten:
 - I/O blockiert nur einen KL-Thread
 - Wechsel zwischen UL-Threads ist effizient
- SMP: Mehrere CPUs benutzen



Prozesse und Threads (Praxis, Entwicklersicht)

Thread-Zustände

- Prozess-Zustände suspended, swapped und einige andere nicht auf Threads übertragbar (warum nicht?)
- Darum betrachten wir nur drei Thread-Zustände



Prozesse und Threads erzeugen (1/11)

Neuer Prozess: `fork ( )`

```

main() {
    int pid = fork(); /* Sohnprozess erzeugen */
    if (pid == 0) {
        printf("Ich bin der Sohn, meine PID ist %d.\n", getpid() );
    }
    else {
        printf("Ich bin der Vater, mein Sohn hat die PID %d.\n", pid);
    }
}
  
```

Anderes Programm starten: `fork` + `exec`

```
main() {
    int pid=fork();    /* Sohnprozess erzeugen */
    if (pid == 0) {
        /* Sohn startet externes Programm */
        execl( "/usr/bin/gedit", "/etc/fstab", (char *) 0 );
    }
    else {
        printf("Es sollte jetzt ein Editor starten...\n");
    }
}
```

Andere Betriebssysteme oft nur: „spawn“

```
main() {
    WinExec("notepad.exe", SW_NORMAL);    /* Sohn erzeugen */
}
```

Wirklich mehrere Prozesse:

Nach `fork` () zwei Prozesse in der Prozessliste

```
> pstree | grep simple
... -bash---simplefork---simplefork
```

```
> ps w | grep simple
25684 pts/16 S+      0:00 ./simplefork
25685 pts/16 S+      0:00 ./simplefork
```

Warten auf Sohn-Prozess: `wait` ()

```
#include <unistd.h>          /* sleep() */
main()
{
    int pid=fork();          /* Sohnprozess erzeugen */
    if (pid == 0)
    {
        sleep(2);            /* 2 sek. schlafen legen */
        printf("Ich bin der Sohn, meine PID ist %d\n", getpid() );
    }
    else
    {
        printf("Ich bin der Vater, mein Sohn hat die PID %d\n",
pid);
        wait();              /* auf Sohn warten */
    }
}
```

Linux: pthread-Bibliothek (POSIX Threads)

	Thread	Prozess
Erzeugen	<code>pthread_create()</code>	<code>fork()</code>
Auf Ende warten	<code>pthread_join()</code>	<code>wait()</code>

- Bibliothek einbinden:
`#include <pthread.h>`
- Kompilieren:
`gcc -lpthread -o prog prog.c`

- Neuer Thread:
`pthread_create()` erhält als Argument eine Funktion, die im neuen Thread läuft.
- Auf Thread-Ende warten:
`pthread_join()` wartet auf einen *bestimmten* Thread.

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

void *thread_function1(void *arg) {
    int i;
    for ( i=0; i<10; i++ ) {
        printf("Thread 1 sagt Hi!\n");
        sleep(1);
    }
    return NULL;
}

void *thread_function2(void *arg) {
    int i;
    for ( i=0; i<10; i++ ) {
        printf("Thread 2 sagt Hallo!\n");
        sleep(1);
    }
    return NULL;
}

int main(void) {
    pthread_t mythread1;
    pthread_t mythread2;

    if ( pthread_create( &mythread1, NULL,
        thread_function1, NULL) ) {
        printf("Fehler bei Thread-Erzeugung.");
        abort();
    }

    sleep(5);

    if ( pthread_create( &mythread2, NULL,
        thread_function2, NULL) ) {
        printf("Fehler bei Thread-Erzeugung.");
        abort();
    }

    sleep(5);

    printf("bin noch hier...\n");

    if ( pthread_join ( mythread1, NULL ) ) {
        printf("Fehler beim Join.");
        abort();
    }

    printf("Thread 1 ist weg\n");

    if ( pthread_join ( mythread2, NULL ) ) {
        printf("Fehler beim Join.");
        abort();
    }

    printf("Thread 2 ist weg\n");

    exit(0);
}
```

1. Thread-Funktion definieren:

```
void *thread_funktion(void *arg) {
    ...
    return ...;
}
```

2. Thread erzeugen:

```
pthread_t thread;

if ( pthread_create( &thread, NULL,
    thread_funktion, NULL) ) {
    printf("Fehler bei Thread-Erzeugung.\n");
    abort();
}
```

Keine „Vater-“ oder „Kind-Threads“

- POSIX-Threads kennen keine Verwandtschaft wie Prozesse (Vater- und Sohnprozess)
- Zum Warten auf einen Thread ist Thread-Variable nötig: `pthread_join (thread, ..)`

Prozess mit mehreren Threads:

- Nur ein Eintrag in normaler Prozessliste
- Status: „l“, multi-threaded
- Über `ps -eLf` Thread-Informationen
 - NLWP: Number of light weight processes
 - LWP: Thread ID

```
> ps auxw | grep thread
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
esser    12022  0.0  0.0  17976   436 pts/15    S1+  22:58   0:00  ./thread

> ps -eLf | grep thread
UID      PID  PPID  LWP  C  NLWP  STIME TTY          TIME CMD
esser    12166  4031  12166  0    3  23:01 pts/15    00:00:00  ./thread1
esser    12166  4031  12167  0    3  23:01 pts/15    00:00:00  ./thread1
esser    12166  4031  12177  0    3  23:01 pts/15    00:00:00  ./thread1
```

Kernel unterscheidet nicht zwischen Prozessen und Threads.

- Doppelt verkettete, ringförmige Liste
- Jeder Eintrag vom Typ `struct task_struct`
- Typ definiert in `include/linux/sched.h`
- Enthält alle Informationen, die Kernel benötigt
- `task_struct`-Definition 132 Zeilen lang!
- Maximale PID: 32767 (short int)

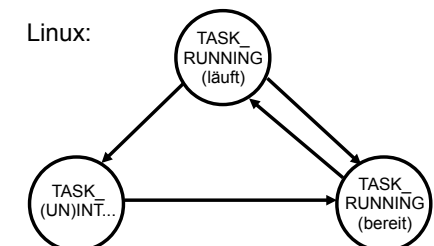
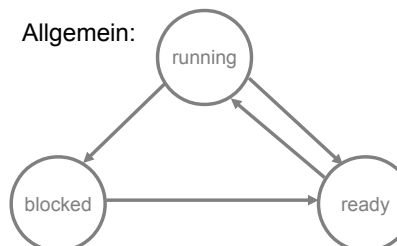
Unterschiedliche Semantik:

- Prozess erzeugen mit `fork ( )`
 - erzeugt zwei (fast) identische Prozesse,
 - beide Prozesse setzen Ausführung an gleicher Stelle fort (nach Rückkehr aus `fork`-Aufruf)
- Thread erzeugen mit `pthread_create (..., funktion, ...)`
 - erzeugt neuen Thread, der in die angeg. Funktion springt
 - erzeugender Prozess setzt Ausführung hinter `pthread_create`-Aufruf fort

Auszug aus `include/linux/sched.h`:

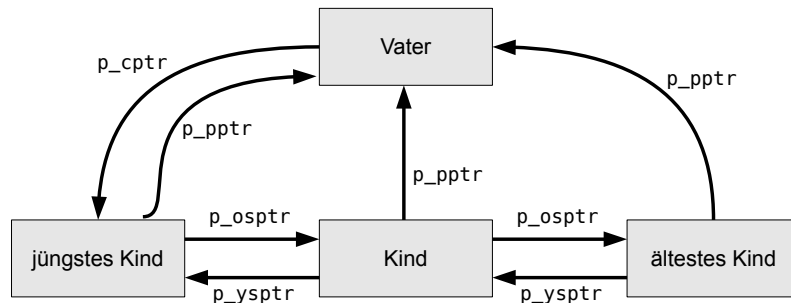
```
#define TASK_RUNNING 0
#define TASK_INTERRUPTIBLE 1
#define TASK_UNINTERRUPTIBLE 2
#define TASK_STOPPED 4
#define TASK_TRACED 8
/* in tsk->exit state */
#define EXIT_ZOMBIE 16
#define EXIT_DEAD 32
/* in tsk->state again */
#define TASK_NONINTERACTIVE 64
#define TASK_DEAD 128
```

- `TASK_RUNNING`: ready oder running
- `TASK_INTERRUPTIBLE`: entspricht blocked
- `TASK_UNINTERRUPTIBLE`: auch blocked
- `TASK_STOPPED`: angehalten (z. B. von einem Debugger)
- `TASK_ZOMBIE`: beendet, aber Vater hat Rückgabewert nicht gelesen



Verwandtschaftsverhältnisse (alte Linux-Version)

```
struct task_struct {
    [...]
    struct task_struct *p_opptr, *p_pptr, *p_cpctr, *p_ysptr, *p_osptr;
```



Prozessgruppen und Sessions

```
struct task_struct {
    [...]
    struct task_struct *group_leader;
    /* threadgroup leader */
    [...]
    /* signal handlers */
    struct signal_struct *signal;
```

```
struct signal_struct {
    /* job control IDs */
    pid_t pgrp;      Process Group ID
    pid_t tty_old_pgrp;
    pid_t session;   Session ID
    /* boolean value for session
    group leader */
    int leader;
```

- Jeder Prozess Mitglied einer Prozessgruppe
- Process Group ID (PGID) – `ps j`
- `current->signal->pgrp`

Verwandtschaftsverhältnisse (neue Linux-Version)

```
struct task_struct {
    [...]
    struct task_struct *parent; /* parent process */
    struct list_head children; /* list of my children */
    struct list_head sibling; /* linkage in my parent's children list */
```

Zugriff auf alle Kinder:

```
list_for_each(list, &current->children) {
    task = list_entry(list, struct task_struct, sibling);
    /* task zeigt jetzt auf eines der Kinder */
}
```

Vom aktuellen Pfad durch den Prozessbaum bis zu `init`:

```
for (task = current; task != &init_task; task = task->parent) {
    ...
}
```

Prozessgruppen

- Signale an alle Mitglieder einer Prozessgruppe:
`killpg(pgrp, sig);`
- Warten auf Kinder aus der eigenen Prozessgruppe:
`waitpid(0, &status, ...);`
- oder einer speziellen Prozessgruppe:
`waitpid(-pgrp, &status, ...);`

Sessions

- Meist beim Starten einer Login-Shell neu erzeugt
- Alle Prozesse, die aus dieser Shell gestartet werden, gehören zur Session
- Gemeinsames „kontrollierendes TTY“

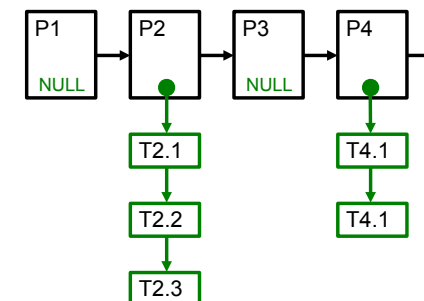
- Linux verwendet für Threads und Prozesse die gleichen Verwaltungsstrukturen (task list)
- Thread: Prozess, der sich mit anderen Prozessen bestimmte Ressourcen teilt, z. B.
 - virtueller Speicher
 - offene Dateien
- Jeder Thread hat `task_struct` und sieht für den Kernel wie ein normaler Prozess aus

```
> ps j
PPID  PID  PGID  SID  TTY      TPGID STAT  UID    TIME COMMAND
19287  7628  7628  19287 pts/8    19287 S      500    0:00 /bin/sh /usr/bin/mozilla -mail
7628   7637  7628  19287 pts/8    19287 Sl     500    20:50 /opt/moz/lib/mozilla-bin -mail
9634   10095 10095  10095 tty1     10114 Ss     500    0:00 -bash
10095  10114 10114  10095 tty1     10114 S+     500    0:00 /bin/sh /usr/X11R6/bin/startx
10095  10115 10114  10095 tty1     10114 S+     500    0:00 tee /home/esser/.X.err
10114  10135 10114  10095 tty1     10114 S+     500    0:00 xinit /home/esser/.xinitrc
10135  10151 10151  10095 tty1     10114 S      500    0:00 /bin/sh /usr/X11R6/bin/kde
10151  10238 10151  10095 tty1     10114 S      500    0:00 kwrapper ksmsserver
10258  10270 10270  10270 pts/2    10270 Ss+    500    0:00 bash
10276  10278 10278  10278 pts/4    10278 Ss+    500    0:00 bash
10260  10284 10284  10284 pts/5    10284 Ss+    500    0:00 bash
10275  10292 10292  10292 pts/6    10989 Ss     500    0:00 bash
10259  10263 10263  10263 pts/1    10263 Ss+    500    0:00 bash
10263  28869 28869  10263 pts/1    10263 S      500    0:16 konqueror /media/usbdisk/dcim
10263  28872 28872  10263 pts/1    10263 S      500    0:13 konqueror /home/esser
29201  29203 29203  29203 pts/7    29203 Ss+    500    0:00 bash
4822   4823 4823  4823 pts/14   4823 Ss+    500    0:00 -bash
4823  31118 31118  4823 pts/14   4823 S      500    0:00 nedit kernel/sched.c
4823  31297 31297  4823 pts/14   4823 S      500    0:00 nedit kernel/fork.c
23115  32703 32703  23115 pts/13   32703 R+     500    0:00 ps j
```

- Fundamental anders als z. B. Windows und Solaris

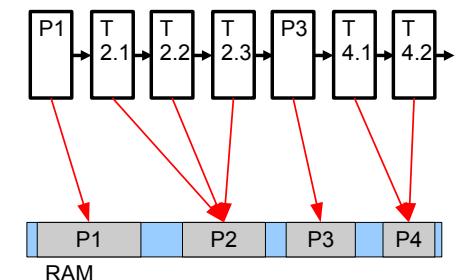
Modell 1:

reine Prozesslisten



Modell 2 (Linux):

Prozesse + Threads gemischt



Betriebssysteme

SS 2015

Hans-Georg Eßer
Dipl.-Math., Dipl.-Inform.

Foliensatz D:

- Interrupts
- System Calls

v1.1, 2015/02/01

Interrupts

Übersicht: BS Praxis und BS Theorie

Einführung
Shell
Dateiverwaltung
Filter
C-Compiler
Prozesse / Jobs
Threads

System Calls
Software-Verwaltung
Scheduler / Prioritäten
Synchronisation

Folien D

Folien D

Interrupts

Motivation (1)

- Festplattenzugriff ca. um Faktor 1.000.000 langsamer als Ausführen einer CPU-Anweisung

- Naiver Ansatz für Plattenzugriff:

```
naiv () {  
    rechne (500 ZE);  
    sende_anfrage_an (disk);  
    antwort = false;  
    while ( ! antwort ) {  
        /* diese Schleife rechnet 1.000.000 ZE lang */  
        antwort = test_ob_fertig (disk);  
    }  
    rechne (500 ZE);  
    return 0;  
}
```

Motivation (2)

- Naiver Ansatz heißt „Pollen“: in Dauerschleife ständig wiederholte Geräteabfrage
- Pollen verbraucht sehr viel Rechenzeit:



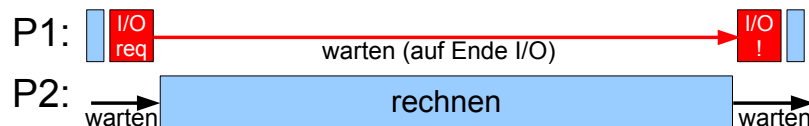
- Besser wäre es, in der Wartezeit etwas anderes zu tun
- Auch bei Parallelbearbeitung mehrerer Prozesse: Polling immer noch ungünstig

Motivation (4)

- Lösung: Interrupts – bestimmte Ereignisse können den „normalen“ Ablauf unterbrechen
- Nach jeder ausgeführten CPU-Anweisung prüfen, ob es einen Interrupt gibt (gab)

Motivation (3)

- Idee: Prozess, der I/O-Anfrage gestartet hat, solange schlafen legen, bis die Anfrage bearbeitet ist – in der Zwischenzeit was anderes tun



- Woher weiß das System,
 - wann die Anfrage bearbeitet ist, also
 - wann der Prozess weiterarbeiten kann?

Interrupt-Klassen

- **I/O (Eingabe/Ausgabe, asynchrone Interrupts)**
Meldung vom I/O-Controller: „Aktion ist abgeschlossen“
- **Timer**
- **Hardware-Fehler**
Stromausfall, RAM-Paritätsfehler
- **Software-Interrupts (Exceptions, Traps, synchrone Interrupts)**
Falscher Speicherzugriff, Division durch 0, unbekannte CPU-Instruktion, ...

Vorteile

- **Effizienz**

I/O-Zugriff sehr langsam → sehr lange Wartezeiten, wenn Prozesse warten, bis I/O abgeschlossen ist

- **Programmierlogik**

Nicht immer wieder Gerätestatus abfragen (Polling), sondern schlafen, bis passender Interrupt kommt

Nachteile

- **Mehraufwand**

Kommunikation mit Hardware wird komplexer, Instruction Cycle erhält zusätzlichen Schritt

Grundsätzlich

- Interrupt tritt auf
- Laufender Prozess wird (nach aktuellem Befehl) unterbrochen, BS übernimmt Kontrolle
- BS speichert Daten des Prozesses (wie bei Prozesswechsel → Scheduler)
- BS ruft Interrupt-Handler auf
- Danach (evtl.): Scheduler wählt Prozess aus, der weiterarbeiten darf (z. B. den unterbrochenen)

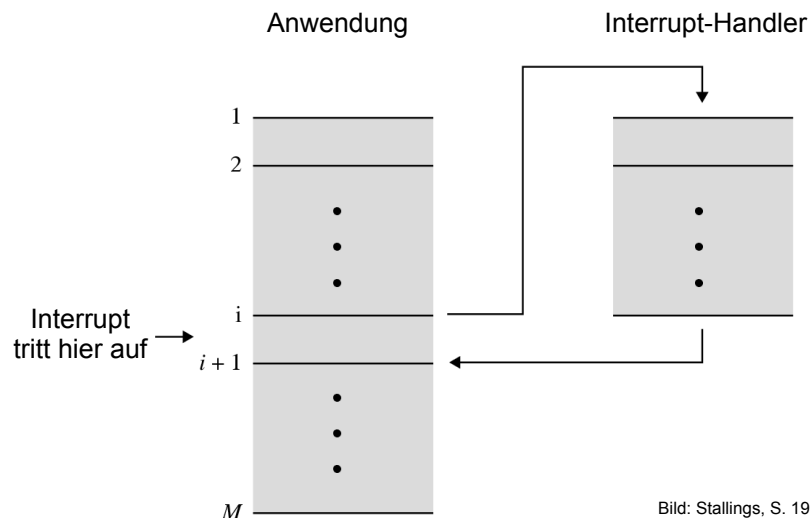


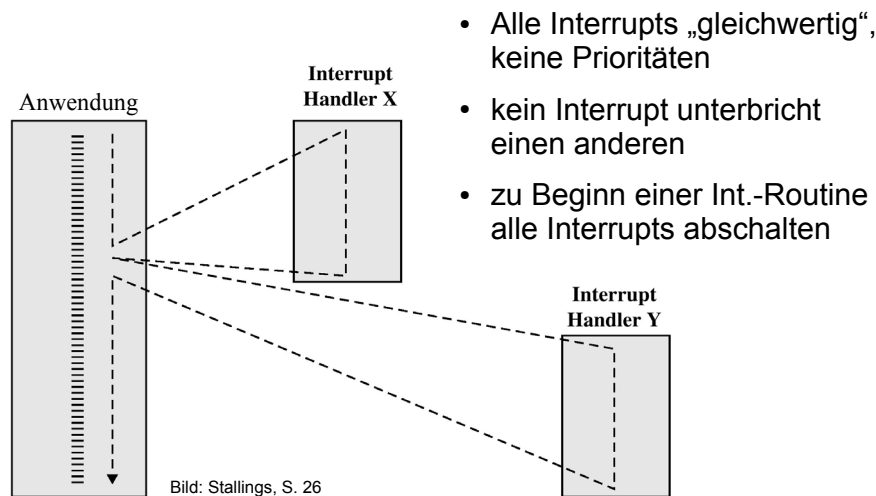
Bild: Stallings, S. 19

Was tun bei Mehrfach-Interrupts?

Drei Möglichkeiten

- Während Abarbeitung eines Interrupts alle weiteren ausschließen (DI, disable interrupts) → Interrupt-Warteschlange
- Während Abarbeitung andere Interrupts zulassen
- Interrupt-Prioritäten: Nur Interrupts mit höherer Priorität unterbrechen solche mit niedrigerer

Mehrfach-Interrupts (1)



I/O-lastig vs. CPU-lastig (1)

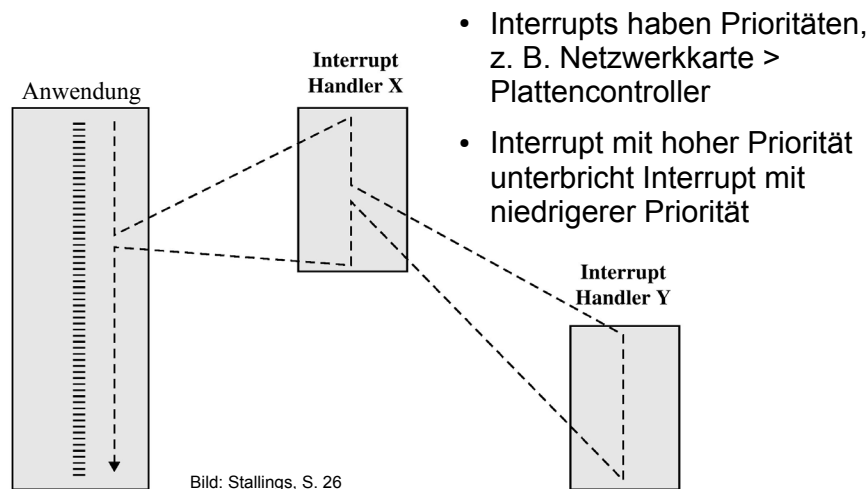
• CPU-lastiger Prozess

- Prozess benötigt überwiegend CPU-Rechenzeit und vergleichsweise wenig I/O-Operationen
- Längere Rechenphasen werden nur gelegentlich durch I/O-Wartezeiten unterbrochen

• I/O-lastiger Prozess

- Prozess führt viele I/O-Operationen durch und benötigt vergleichsweise wenig Rechenzeit
- Sehr kurze Rechenphasen wechseln sich mit häufigen Wartezeiten auf I/O ab

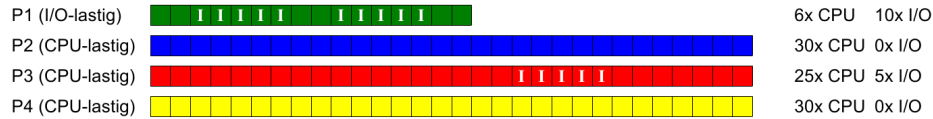
Mehrfach-Interrupts (2)



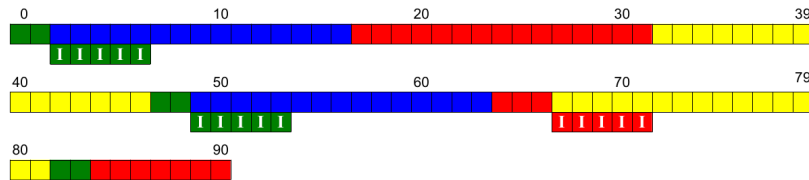
I/O-lastig vs. CPU-lastig (2)

Multitasking und Interrupts

- Multitasking verbessert CPU-Nutzung:
 - I/O-lastiger Prozess wartet auf I/O-Events,
 - CPU-lastiger Prozess rechnet währenddessen weiter
- Prozess stößt I/O-Operation an und blockiert (wartet darauf, dass das BS ihn wieder auf „bereit“ setzt und irgendwann fortsetzt)
- optimale Performance: gute Mischung I/O- und CPU-lastiger Prozesse



Ausführreihenfolge mit Round Robin, Zeitquantum 15:



Prozess	CPU-Zeit	I/O-Zeit	Summe	Laufzeit	Wartezeit *)
P1	6	10	16	84	68
P2	30	0	30	64	34
P3	25	5	30	91	61
P4	30	0	30	82	52

Praxis: Interrupts unter Linux

```
[esser@server ~]$ cat /proc/interrupts
CPU0
0: 3353946487      XT-PIC  timer
2:                0      XT-PIC  cascade
3:                4663    XT-PIC  NVidia CK804
5: 159275991       XT-PIC  ohci1394, nvidia
7:  971775         XT-PIC  hsfpcibasic2
8:                2      XT-PIC  rtc
9:                0      XT-PIC  acpi
10:               31052   XT-PIC  libata, ohci_hcd
11: 197906977      XT-PIC  libata, ehci_hcd
12: 16904921       XT-PIC  eth0
14: 60349322       XT-PIC  ide0
NMI:                0
LOC:                0
ERR:                0
MIS:                0
```

```
[esser@quad:~]$ cat /proc/interrupts
CPU0      CPU1      CPU2      CPU3
0: 5224      3         1         1      IO-APIC-edge  timer
1: 298114    774        793       793     IO-APIC-edge  i8042
3: 9         8         6         9      IO-APIC-edge
4: 8         9         8         6      IO-APIC-edge
8: 0         0         0         1      IO-APIC-edge  rtc0
9: 0         0         0         0      IO-APIC-fasteoi acpi
12: 3070145  16539      16542     16485   IO-APIC-edge  i8042
16: 2760924  881        904       886     IO-APIC-fasteoi uhci_hcd:usb1, nvidia
18: 24122388  6538      6698      6647     IO-APIC-fasteoi ehci_hcd:usb6, uhci_hcd:usb7
19: 281       28         27        10      IO-APIC-fasteoi uhci_hcd:usb3, uhci_hcd:usb5
21: 22790     0          0         0      IO-APIC-fasteoi uhci_hcd:usb2
22: 7786588  10464141  8251870   8439964  IO-APIC-fasteoi HDA Intel
23: 899       0          1         1      IO-APIC-fasteoi uhci_hcd:usb4, ehci_hcd:usb8
221: 9519152  10751650  9745810   10326363 PCI-MSI-edge  eth0
222: 14462926  38205     38095     38178   PCI-MSI-edge  ahci
NMI: 0          0          0          0      Non-maskable interrupts
LOC: 724999305 786034088 748693018 748218173 Local timer interrupts
RES: 5334382  3576152  3464671  3357556  Rescheduling interrupts
CAL: 2111668  4233550  4067655  3871450  function call interrupts
TLB: 101757  113319  88752  107777  TLB shootdowns
TRM: 0          0          0          0      Thermal event interrupts
SPU: 0          0          0          0      Spurious interrupts
ERR: 0
MIS: 0
```

Für jedes Gerät:

- Interrupt Request (IRQ) Line
- Interrupt Handler (Interrupt Service Routine, ISR) → Teil des Gerätetreibers
- C-Funktion
- läuft in speziellem Context (Interrupt Context)
- „top half“ und „bottom half“

Tasklet (bottom half)

- Tasklets führen längere Berechnungen durch, die zur Interrupt-Verarbeitung gehören – dabei sind Interrupts zugelassen
- Tasklet ist kein Prozess (struct tasklet_struct), läuft direkt im Kernel; im Interrupt-Context
- Zwei Prioritäten:
 - *tasklet_hi_schedule*: startet direkt nach ISR
 - *tasklet_schedule*: startet erst, wenn kein anderer Soft IRQ mehr anliegt

„top half“ und „bottom half“**top half**

- Interrupt handler
- startet sofort, erledigt zeitkritische Dinge
- bestätigt (der Hardware) den Erhalt des Interrupts, setzt Gerät zurück etc.
- erzeugt bottom half
- Alles andere → bottom half

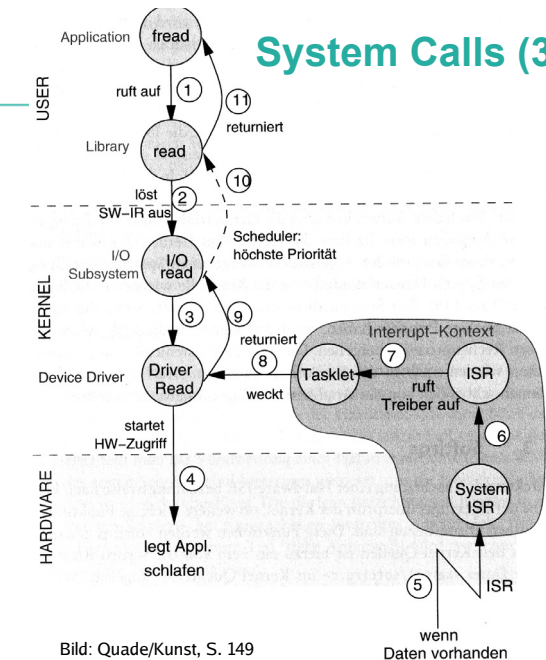
Mehr Informationen:

- [1] Linux Kernel 2.4 Internals, Kapitel 2,
http://www.faqs.org/docs/kernel_2_4/lki-2.html
- [2] J. Quade, E.-K. Kunst: „Linux-Treiber entwickeln“, dpunkt-Verlag,
<http://ezs.kr.hsnr.de/TreiberBuch/html/>

- **System Call:** Mechanismus, über den ein Anwendungsprogramm Dienstleistungen des BS nutzt.
- Führt eine Anwendung einen System Call aus, schaltet das BS in den **Kernel-Modus** („privilegierten Modus“) um.
- Für viele Aufgaben (etwa Zugriff auf Geräte oder Kommunikation mit anderen Prozessen) sind Rechte nötig, die normale Anwendungen nicht besitzen (User mode vs. Kernel mode). Das geht dann nur mit System Calls.
- Oft implementiert über **Software Interrupt (Trap)**. Nach Interrupt Wechsel in den Kernel-Modus.
- System-Call-Nummer in ein Register eintragen und den Software-Interrupt auslösen

Beispiel für einen System Call:

Library-Funktion
fread()



/usr/include/asm/unistd_32.h: Über 300 System Calls

```

/*
 * This file contains the system call
 * numbers.
 */

#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5
#define __NR_close 6
#define __NR_waitpid 7
#define __NR_creat 8
#define __NR_link 9
#define __NR_unlink 10
#define __NR_execve 11
#define __NR_chdir 12
#define __NR_time 13
#define __NR_mknod 14
#define __NR_chmod 15
#define __NR_lchown 16

#define __NR_break 17
#define __NR_oldstat 18
#define __NR_lseek 19
#define __NR_getpid 20
#define __NR_mount 21
#define __NR_umount 22
#define __NR_setuid 23
#define __NR_getuid 24
#define __NR_stime 25
#define __NR_ptrace 26
#define __NR_alarm 27
#define __NR_oldfstat 28
#define __NR_pause 29
#define __NR_ftime 30
#define __NR_utime 31
#define __NR_stty 32
#define __NR_gtty 33
#define __NR_access 34
#define __NR_nice 35
#define __NR_ftime 36
#define __NR_sync 37
#define __NR_kill 38

```

**System Calls
für Programmierer:**

Standardfunktionen in C

open()

Daten zum Lesen/Schreiben öffnen

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
```

Rückgabewert: File Descriptor

man 2 open

Beispiel:

```
fd = open("/tmp/datei.txt", O_RDONLY);
```

Beispiel: read() und open()

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
int main (void) {
    int len; int bufsiz=128; char line[bufsiz+1];
    line[bufsiz] = '\0';
    int fd = open( "/etc/fstab", O_RDONLY );
    while ( (len = read ( fd, line, bufsiz )) > 0 ) {
        if ( len < bufsiz ) { line[len]='\0'; }
        printf ("%s", line );
    }
    close(fd);
    return 0;
}
```

read()

Daten aus Datei (File Descriptor) lesen

```
ssize_t read(int fd, void *buf, size_t count);
```

Rückgabewert: Anzahl gelesene Bytes

man 2 read

Beispiel:

```
int bufsiz=128; char line[bufsiz+1];
int fd = open( "/etc/fstab", O_RDONLY );
int len = read ( fd, line, bufsiz );
```

write()

Daten in Datei (File Descriptor) schreiben

```
ssize_t write(int fd, void *buf, size_t count);
```

Rückgabewert: Anzahl geschriebene Bytes

man 2 write

Beispiel:

```
main() {
    char message[] = "Hello world\n";
    int fd = open( "/tmp/datei.txt",
        O_CREAT | O_WRONLY, S_IRUSR | S_IWUSR );
    write ( fd, message, sizeof(message) );
    close(fd);
    exit(0);
}
```

`close( )` Datei (File Descriptor) schließen

```
int close(int fd);
```

Rückgabewert: 0 bei Erfolg, sonst -1 (errno enthält dann Grund)

man 2 close

Beispiel:

```
close(fd);
```

`fork( )` neuen Prozess starten

```
pid_t fork(void);
```

Rückgabewert: Child-PID (im Vaterprozess); 0 (im Sohnprozess); -1 (im Fehlerfall)

man fork

Beispiel:

```
pid=fork( )
```

`exit( )` Programm beenden

```
void exit(int status);
```

Kein Rückgabewert, aber *status* wird an aufrufenden Prozess weitergegeben.

man 3 exit

Beispiel:

```
exit(0);
```

`exec( )` Anderes Programm im Prozess laden

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char * const envp[]);
int execlp(const char *path, char *const argv[]);
int execlp(const char *file, char *const argv[]);
```

Rückgabewert: keiner (Funktion kehrt nicht zurück)
Parameter arg0 (Name), arg1, ...; letztes Argument: NULL-Zeiger

man 3 exec

Beispiele:

```
execl ("/usr/bin/vi", "", "/etc/fstab", (char *) NULL);
execlp ("vi", "", "/etc/fstab", (char *) NULL);
```

Am Anfang jedes C-Programms:

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
```

sys/stat.h enthält z. B. S_IRUSR, S_IWUSR
fcntl.h enthält z. B. O_CREAT, O_WRONLY

Betriebssysteme

SS 2015

Hans-Georg Eßer
Dipl.-Math., Dipl.-Inform.

Foliensatz E:

- Paketverwaltung

v1.1, 2014/11/21

Software-Verwaltung

Übersicht: BS Praxis und BS Theorie

Einführung
Shell
Dateiverwaltung
Filter
C-Compiler
Prozesse / Jobs
Threads
Interrupts

System Calls
Software-Verwaltung
Scheduler / Prioritäten
Synchronisation

Folien E

Grundlagen Paketverwaltung (1)

- Verschiedene Linux-Distributionen verwenden unterschiedliche Software-Paketformate:
 - **.deb* – Debian-Paketformat (z. B. Debian, Ubuntu, Knoppix)
 - **.rpm* – Red-Hat-Paketformat (z. B. OpenSuse, Fedora / Red Hat, Mandriva)
 - weitere (weniger verbreitete Formate), z. B. **.txz* (Slackware, xz-komprimiertes *tar*-Archiv)
- Pakete enthalten die eigentliche Software und „Anleitungen“ für (De-) Installation

- Die beiden „großen“ Paketformate sind
 - das Debian-Format (*.deb) und
 - das RPM-Format (*.rpm, Red Hat Package Manager)
- Software-Verwaltung mit *deb/rpm*-Paketen findet immer auf zwei Ebenen statt:
 - auf Ebene einzelner Pakete; Tools:
 - dpkg (Debian Package Manager)
 - rpm (Red Hat Package Manager)
 - auf Ebene von Repositories; Tools:
 - APT (Advanced Packaging Tool / Debian)
 - YUM (Yellowdog Updater, Modified / Red Hat, Fedora)



- Funktionen auf Repository-Ebene (1)
 - Ein Repository (Repo) ist eine Paketquelle, z. B.
 - Installations-CD/DVD
 - Verzeichnis auf einem Web- oder FTP-Server
 - lokales Verzeichnis
 - Repos haben ein Inhaltsverzeichnis mit den wichtigsten Metadaten zu allen vorhandenen Paketen
 - Repos für jede Distributions-Version separat

- Funktionen auf Paketebene
 - Paket installieren / deinstallieren / aktualisieren
 - Paketinhalt anzeigen
 - erkennt → Abhängigkeiten und → Konflikte, verweigert evtl. (De-) Installation oder Update
 - zu installierter Datei das zugehörige Paket finden
 - Installation / Update benötigt immer eine konkrete Paketdatei

- Funktionen auf Repository-Ebene (2)
 - Anlegen / Aktualisieren / Entfernen von Repos (→ parallele Nutzung mehrerer Repos möglich)
 - Suche nach Paketen in allen eingetragenen Repos
 - Installation mit automatischem Auflösen von Abhängigkeiten
 - Komplett-Upgrade auf neue Version einer Linux-Distribution
 - Drittanbieter-Repos (z. B. für Multimedia-Pakete, Treiber vom Hardware-Hersteller oder proprietäre Software)

- Paket installieren:



```
[root@redhat:~]# ls
testpaket-1.2.3.i386.rpm
[root@redhat:~]# rpm -ivh testpaket-1.2.3.i386.rpm
Preparing... ##### [100%]
 1:testpaket      ##### [100%]
[root@redhat:~]# _
```

Optionen -vh erzeugen
die Ausgabe



```
[root@debian:~]# ls
joe_3.5-1.1_amd64.deb
[root@debian:~]# dpkg -i joe_3.5-1.1_amd64.deb
Wähle vormals abgewähltes Paket joe. (Lese Datenbank ...
113154 Dateien und Verzeichnisse sind derzeit installiert.)
Entpacke joe (aus ../archives/joe_3.5-1.1_amd64.deb) ...
Richte joe ein (3.5-1.1) ...
[root@debian:~]# _
```

- Paket löschen: benötigt nur Paketnamen, nicht die Versionsnummer



```
[root@redhat:~]# rpm -e testpaket
```

e = erase



```
[root@debian:~]# dpkg -r joe
(Lese Datenbank ... 113175 Dateien und
Verzeichnisse sind derzeit installiert.)
Entferne joe ...
```

r = remove

oder

```
[root@debian:~]# dpkg -P joe
(Lese Datenbank ... 113175 Dateien und
Verzeichnisse sind derzeit installiert.)
Entferne joe ...
Lösche Konfigurationsdateien von joe ...
```

P = purge

- Paket aktualisieren (Update):



```
[root@redhat:~]# rpm -ivh dhcpd-3.2.3-66.69.1.i586.rpm
Preparing... ##### [100%]
file /sbin/dhcpd from install of dhcpd-3.2.3-
66.69.1.i586 conflicts with file from package dhcpd-
3.2.3-65.1.i586
```

```
[root@redhat:~]# rpm -Uvh dhcpd-3.2.3-66.69.1.i586.rpm
Preparing... ##### [100%]
 1:dhcpd      ##### [100%]
```



```
[root@debian:~]# dpkg -i joe_3.5-1.1_amd64.deb
```

(Update bei Debian auch über Installationsoption -i)

- Paketinhalte:



```
[root@redhat:~]# rpm -ql rpm
/bin/rpm
/etc/init.d/rpmconfigcheck
/etc/rpm
/usr/bin/gendiff
/usr/bin/rpm2cpio
/usr/bin/rpmbuild
/usr/bin/rpmdb
...
```

ql = query,
list



```
[root@debian:~]# dpkg -L dpkg
/.
/var
/var/lib
/var/lib/dpkg
/var/lib/dpkg/updates
/var/lib/dpkg/parts
/var/lib/dpkg/info
joe ...
```

L = list

• Paketinformationen ():

```
[root@redhat:~]# rpm -qi rpm
Name           : rpm                      Relocations: (not relocatable)
Version        : 4.8.0                  Vendor: openSUSE
Release        : 27.1                   Build Date: Do 17 Feb 2011 19:09:21 CET
Install Date:  Di 15 Mär 2011 11:01:20 CET  Build Host: build24
Group          : System/Packages         Source RPM: rpm-4.8.0-27.1.src.rpm
Size           : 3674658                  License: GPLv2+
Signature      : RSA/8, Do 17 Feb 2011 19:09:42 CET, Key ID b88b2fd43dbdc284
Packager       : http://bugs.opensuse.org
Summary        : The RPM Package Manager
Description    :
RPM Package Manager is the main tool for managing the software packages
of the SuSE Linux distribution.
```

RPM can be used to install and remove software packages. With rpm, it is easy to update packages. RPM keeps track of all these manipulations in a central database. This way it is possible to get an overview of all installed packages. RPM also supports database queries.

qi = query,
info

Paketkonvertierung

• alien wandelt RPM- und DEB-Archiv ineinander um

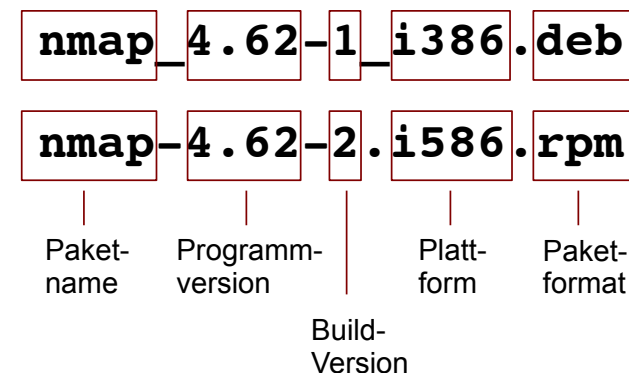
```
[root@debian:~]# alien --to-rpm nmap_4.62-1_i386.deb
Warning: Skipping conversion of scripts in package nmap: postinst premm
Warning: Use the --scripts parameter to include the scripts.
nmap-4.62-2.i386.rpm generated
[root@debian:~]# ls -l
-rw-r--r-- 1 root root 1054002 May 13 20:55 nmap-4.62-2.i386.rpm
-rw-r--r-- 1 root root 1048536 May 13 20:54 nmap_4.62-1_i386.deb
[root@debian:~]# rpm -qip nmap-4.62-2.i386.rpm
Name           : nmap                      Relocations: (not relocatable)
Version        : 4.62                  Vendor: (none)
Release        : 2                     Build Date: Fri May 13 20:55:33 2011
Install Date:  (not installed)         Build Host: hgesser.com
Group          : Converted/net         Source RPM: nmap-4.62-2.src.rpm
Size           : 3461471                License: see /usr/share/doc/nmap/copyri
Signature      : (none)
Summary        : The Network Mapper
Description    :
Nmap is a utility for network exploration or security auditing. It
supports ping scanning (determine which hosts are up), many port
...
```

• Paketinformationen ():

```
[root@debian:~]# dpkg -s dpkg
Package: dpkg
Essential: yes
Status: install ok installed
Priority: required
Section: admin
Installed-Size: 7276
Origin: debian
Maintainer: Dpkg Developers <debian-dpkg@lists.debian.org>
Architecture: i386
Version: 1.14.31
Replaces: manpages-de (<= 0.4-3), manpages-pl (<= 20051117-1)
Pre-Depends: libc6 (>= 2.7-1), coreutils (>= 5.93-1), lzma
Suggests: apt
Conflicts: apt (< 0.7.7), aptitude (< 0.4.7-1), dpkg-dev (< 1.14.16)
Conffiles:
/etc/logrotate.d/dpkg 501f8c90b83c7ea180868ca82e1e82d1
/etc/dpkg/origins/debian 731423fa8ba067262f8ef37882d1e742
/etc/dpkg/dpkg.cfg f4413ffb515f8f753624ae3bb365b81b
Description: Debian package management system
This package provides the low-level infrastructure for handling the
installation and removal of Debian software packages.
.
For Debian package development tools, install dpkg-dev.
Homepage: http://wiki.debian.org/Teams/Dpkg
```

s = status

Aufbau der Paketnamen



Plattformen:

i386: Intel i386
i486, i586, i686: ...
sparc: Sun SPARC
ppc: IBM, Apple
ppc64: IBM PowerPC
alpha: DEC Alpha
ia64: Intel Itanium
x86_64: PC, 64 Bit
noarch: übergreifend

```
# dpkg-architecture -I
armel
lpia
i386
ia64
alpha
amd64
armeb
arm
...
```

Vollständige Paketliste:

```
# rpm -qa
gpg-pubkey-3dbdc284-4be1884d
bundle-lang-gnome-en-11.4-5.13.1.noarch
translation-update-11.3-7.1.noarch
yast2-trans-stats-2.19.0-4.1.noarch
openSUSE-release-ftp-11.4-1.9.i586
apparmor-docs-2.5.1.r1445-52.55.1.i586
lsscsi-0.23-6.1.i586
libspeexl-1.1.999_1.2rc1-9.1.i586
...
-qa: query, all

# dpkg -l
Desired=Unknown/Install/Remove/Purge/Hold
Status=Not/Inst/Cfg-files/Unpacked/Failed-cfg/Half-inst/trig-aWait/Trig-pend
/ Err?=(none)/Hold/Reinst-required/X=both-problems (Status,Err: uppercase=bad)
||/ Name Version Description
++-----+-----+-----+
ii adduser 3.110 add and remove users and groups
ii alien 8.72 convert and install rpm and other package
ii apache2 2.2.9-10+lenny9 Apache HTTP Server metapackage
ii apache2-mpm-prefork 2.2.9-10+lenny9 Apache HTTP Server - traditional non-thre
ii apache2-utils 2.2.9-10+lenny9 utility programs for webserver
ii apache2.2-common 2.2.9-10+lenny9 Apache HTTP Server common files
ii apt 0.7.20.2+lenny2 Advanced front-end for dpkg
...
-l: list
```

- Ein Paket hat **Abhängigkeiten** (engl. **dependencies**), wenn weitere Pakete oder bestimmte Dateien zwingend installiert sein müssen, damit es funktioniert
- Diese zusätzlich benötigten Pakete / Dateien nennt man die Abhängigkeiten
- Beispiele:
 - Paket mc (Midnight Commander) benötigt die Bibliothek libc.so.6
 - Paket openssh-askpass benötigt Paket openssh

- rpm und dpkg werden heute nur noch selten direkt benutzt
- stattdessen Paketverwaltung mit „höheren“ Tools wie YUM und APT

- Abhängigkeit von Paket meist (relativ) leicht lösbar:
 - Paket suchen und installieren
 - richtige Versionsnummer, Linux-Distribution (und -Version), Plattform beachten!
- Abhängigkeit von Datei (i.d.R. Bibliothek) oft schwieriger zu beheben
 - „Welches Paket enthält die Datei x.y.so.1?“

- Installation bei **nicht erfüllten Abhängigkeiten** lässt sich trotzdem erzwingen:



```
[root@redhat:~]# rpm -i --nodeps paket.rpm
```



```
[root@debian:~]# dpkg -i --force-depends paket.deb
```

- Ratsam ist das aber nicht; Software wird meist nicht (oder nicht korrekt) arbeiten

- Zwei Pakete können miteinander in **Konflikt** stehen, d. h.: Es darf nur eines von beiden installiert sein.
- Konflikt tritt meist auf, wenn man versucht, ein neues Paket zu installieren, und bereits ein damit in Konflikt stehendes installiert ist.
- Beispiele:
 - zwei Mail-Server, zwei DHCP-Server (feste Ports)

- Informationen zu Abhängigkeiten:



```
[root@redhat:~]# rpm -qp --requires mttools-4.0.15-5.6.1.i586.rpm -qp: query package
/bin/sh
rpmLib(PayloadFilesHavePrefix) <= 4.0-1
rpmLib(CompressedFileNames) <= 3.0.4-1
libc.so.6
libc.so.6(GLIBC_2.0)
rpmLib(PayloadIsLzma) <= 4.4.6-1
...
```



```
[root@debian:~]# dpkg -f nmap_4.62-1_i386.deb depends
libc6 (>= 2.7-1), libgcc1 (>= 1:4.1.1),
libpcap0.8 (>= 0.9.3-1), libpcre3 (>= 7.4), -f: field
libssl0.9.8 (>= 0.9.8f-5), libstdc++6 (>= 4.2.1)
```

Achtung Syntax: Nach -f erst Paketname, dann Feldname depends

- Installation bei **Konflikten** lässt sich trotzdem erzwingen:



```
[root@redhat:~]# rpm -i --force paket.rpm
```



```
[root@debian:~]# dpkg -i --force-conflicts paket.deb
```

- Auch das Ignorieren von Konflikten ist nicht ratsam.

- Repository:
 - Sammlung von Software-Paketen
 - für genau eine Distributionsversion (z. B. OpenSuse 11.4 oder Ubuntu 10.10)
 - mit Inhaltsbeschreibungen aller Pakete
 - und Abhängigkeitsinformationen zu allen Paketen
- Installations-Tool kann Abhängigkeiten damit automatisch auflösen, nötige Zusatzpakete auswählen und diese gleich mit-installieren



- Debian: APT
 - Advanced Packaging Tool
 - neben Debian auch von „Debian-basierten“ Distributionen genutzt, z. B. Knoppix, Ubuntu
 - Repository-Konfiguration in `/etc/apt/sources.list`
 - Paketverwaltung mit den Befehlen `apt-get` und `apt-cache`

- Paketverwaltung mit Repos:
 - Verwaltung der Repos
 - Eintragen neuer Repos und Löschen von Repos
 - Aktualisieren vorhandener Repos
 - „eigentliche“ Paketverwaltung
 - Installation, Update, Deinstallation von Paketen über die Repo-Tools



- Beispiel-Konfigurationsdatei `/etc/apt/sources.list`:

```
deb cdrom:[Debian GNU/Linux 4.0]/ etch contrib main
deb http://ftp.gwdg.de/pub/linux/debian/debian/ etch main non-free
deb-src http://ftp.gwdg.de/pub/linux/debian/debian/ etch main
```
- Jede Zeile besteht aus vier Teilen:
 - `deb` oder `deb-src`: Handelt es sich um eine Binär- oder Source-Paketquelle?
 - URI zur Quelle, z. B. mit `http://`, `ftp://` oder `cdrom:[Volume-Name]`
 - Distributionsversion (`etch`, `sarge`, `squeeze` etc.; alternativ: `stable`, `unstable`, `testing`)
 - einzubindende (Unter-) Repositories: `main`, `contrib`, `non-free` etc.



- Neues Repo anlegen (Web):
 - Zusätzliche Zeile in `sources.list` eintragen
 - `apt-get update` ausführen

```
server:~ # apt-get update
Hole:1 http://security.debian.org etch/updates Release.gpg [189B]
Hole:2 http://security.debian.org etch/updates Release [22,5kB]
Hole:3 http://http.us.debian.org etch Release.gpg [378B]
Hole:4 http://http.us.debian.org etch Release [58,2kB]
Hole:5 http://security.debian.org etch/updates/main Packages [91,5kB]
Hole:6 http://http.us.debian.org etch/main Packages [4223kB]
Hole:7 http://security.debian.org etch/updates/contrib Packages [14B]
Hole:8 http://security.debian.org etch/updates/non-free Packages [14B]
Hole:9 http://http.us.debian.org etch/contrib Packages [50,6kB]
Hole:10 http://http.us.debian.org etch/non-free Packages [63,9kB]
Es wurden 4510kB in 15s geholt (291kB/s)
Paketlisten werden gelesen... Fertig
```



- Neues Repo (CD/DVD) anlegen: `apt-cdrom`

```
server:~ # apt-cdrom add
Verwendeter CD-ROM-Einbindungspunkt: /media/cdrom0/
Identifizieren ... [e1947a0c703f32d960fbfce6c7961521-2]
Durchsuchen des Mediums nach Index-Dateien ...
1 Paketindizes, 0 Quellindizes, 0 Übersetzungsindizes und 0
Signaturen gefunden
Dieses Medium heißt:
»Debian GNU/Linux 6.0.1a _Squeeze_ - Official i386 xfce+lxde-CD
Binary-1 20110322-15:11«
Reading Package Indexes... Fertig
Schreiben der neuen Quellliste
Quelllisteneinträge für dieses Medium sind:
deb cdrom:[Debian GNU/Linux 6.0.1a _Squeeze_ - Official i386
xfce+lxde-CD Binary-1 20110322-15:11]/ squeeze main
Wiederholen Sie dieses Prozedere für die restlichen Disks Ihres
Satzes.
```

→ erzeugt auch neuen Eintrag in `/etc/apt/sources.list`



- Pakete suchen: `apt-cache search`

```
server:~ # apt-cache search kmenu
deskmenu - A root menu for X11 window managers
kmenuedit - menu editor for KDE
oroborus - A lightweight themeable windowmanager for X
tastymenu - replacement K-menu for KDE/Kicker
```

- Pakete installieren: `apt-get install`

```
server:~ # apt-get install lynx
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  lynx-cur
The following NEW packages will be installed:
  lynx lynx-cur
0 upgraded, 2 newly installed, 0 to remove and 6 not upgraded.
Need to get 2040kB of archives.
After this operation, 4518kB of additional disk space will be used.
Do you want to continue [Y/n]? Y
```



Abhängigkeiten

```
root@debian:/home/esser# apt-get install k3b
Paketlisten werden gelesen... Fertig
Abhängigkeitsbaum wird aufgebaut
Statusinformationen werden eingelesen... Fertig
Die folgenden zusätzlichen Pakete werden installiert:
  cdparanoia cdrdao dvd+rw-tools exiv2 genisoimage k3b-data kaboom kbase-runtime kbase-runtime-data kdelibs-bin
  kdelibs5-data kdelibs5-plugins kdoc2tools libao-common libao4 libasynctns libattica0 libclucene0ldbl libeggdbus-1-0
  libexiv2-9 libflac+6 libgomp1 libgraphicsmagick3 liblmbase6 libiodbc2 libjack-jackd2-0 libk3b6 libk3b6-extracodecs
  libk3b6-extracodecs4 libk3b6-support4 libkdecore5 libkdeu5 libkdeui5 libkdnssd4 libkfile4 libkhtml5 libkio5 libkjsapi4
  libkjsembed4 libkmediaplayer4 libknewstuff2-4 libknewstuff3-4 libknotifyconfig4 libkntlm4 libkparts4 libkpty4
  libkrosscore4 libktexteditor4 libkutil4 libmusicalbrainz4c2a libmysqldclient16 libnepomuk4 libnepomukquery4a libopenexr6
  libphonon4 libplasma3 libpolkit-agent-1-0 libpolkit-gobject-1-0 libpolkit-qt-1-0 libpulse-mainloop-glib0 libpulse0
  libqca2 libqt4-dbus libqt4-designer libqt4-network libqt4-opengl libqt4-qtsupport libqt4-script libqt4-sql libqt4-sql-
  mysql libqt4-svg libqt4-webkit libqt4-xml libqt4-xmlpatterns libreadline5 libsamplerate0 libsndfile1 libsolid4
  lib soprano4 libssh-4 libstreamanalyzer0 libstreams0 libtheadweaver4 libvirtodbc0 libvorbisfile3 libwmf0.2-7 libxcb-
  shape0 libxine1 libxine1-bin libxine1-console libxine1-ffmpeg libxine1-misc-plugins libxine1-plugins libxine1-x
  libxml2-utils libxslt1 mysql-common odbinst odbinst-debian2 oxygen-icon-theme phonon phonon-backend-xine plasma-
  scriptengine-javascript shared-desktop-ontologies soprano-daemon ttf-dejavu ttf-dejavu-extra vcdimager virtuos-o-minimal
  virtuos-o-opensource-6.1-bin virtuos-o-opensource-6.1-common wodim xdg-utils
Die folgenden NEUEN Pakete werden installiert:
  cdparanoia cdrdao dvd+rw-tools exiv2 genisoimage k3b k3b-data kaboom kbase-runtime kbase-runtime-data kdelibs-bin
  kdelibs5-data kdelibs5-plugins kdoc2tools libao-common libao4 libasynctns libattica0 libclucene0ldbl libeggdbus-1-0
  libexiv2-9 libflac+6 libgomp1 libgraphicsmagick3 liblmbase6 libiodbc2 libjack-jackd2-0 libk3b6 libk3b6-extracodecs
  libk3b6-extracodecs4 libk3b6-support4 libkdecore5 libkdeu5 libkdeui5 libkdnssd4 libkfile4 libkhtml5 libkio5 libkjsapi4
  libkjsembed4 libkmediaplayer4 libknewstuff2-4 libknewstuff3-4 libknotifyconfig4 libkntlm4 libkparts4 libkpty4
  libkrosscore4 libktexteditor4 libkutil4 libmusicalbrainz4c2a libmysqldclient16 libnepomuk4 libnepomukquery4a libopenexr6
  libphonon4 libplasma3 libpolkit-agent-1-0 libpolkit-gobject-1-0 libpolkit-qt-1-0 libpulse-mainloop-glib0 libpulse0
  libqca2 libqt4-dbus libqt4-designer libqt4-network libqt4-opengl libqt4-qtsupport libqt4-script libqt4-sql libqt4-sql-
  mysql libqt4-svg libqt4-webkit libqt4-xml libqt4-xmlpatterns libreadline5 libsamplerate0 libsndfile1 libsolid4
  lib soprano4 libssh-4 libstreamanalyzer0 libstreams0 libtheadweaver4 libvirtodbc0 libvorbisfile3 libwmf0.2-7 libxcb-
  shape0 libxine1 libxine1-bin libxine1-console libxine1-ffmpeg libxine1-misc-plugins libxine1-plugins libxine1-x
  libxml2-utils libxslt1 mysql-common odbinst odbinst-debian2 oxygen-icon-theme phonon phonon-backend-xine plasma-
  scriptengine-javascript shared-desktop-ontologies soprano-daemon ttf-dejavu ttf-dejavu-extra vcdimager virtuos-o-minimal
  virtuos-o-opensource-6.1-bin virtuos-o-opensource-6.1-common wodim xdg-utils
0 aktualisiert, 114 neu installiert, 0 zu entfernen und 27 nicht aktualisiert.
Es müssen noch 96,2 MB von 99,6 MB an Archiven heruntergeladen werden.
Nach dieser Operation werden 238 MB Plattenplatz zusätzlich benutzt.
Möchten Sie fortfahren [Y/n]? Y
```



- Pakete entfernen

```
server:~ # apt-get remove mc
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages will be REMOVED:
 mc
0 upgraded, 0 newly installed, 1 to remove and 6 not upgraded.
After this operation, 6402kB disk space will be freed.
Do you want to continue [Y/n]?
```

→ remove: ohne Entfernen der Konfigurationsdateien (wie dpkg -r)

```
server:~ # apt-get purge paketname
```

→ purge: mit Entfernen der Konfigurationsdateien (wie dpkg -P)



- Alle Pakete aktualisieren: apt-get upgrade

```
server:~ # apt-get upgrade
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages will be upgraded:
 bind9-host dhcp3-client dhcp3-common dnstools libbind9-50
 libdns58 libisc50 libisccc50 libiscfg50 liblwres50
 libmozjs1d libnss3-ld libtiff4 mysql-common xulrunner-1.9
15 upgraded, 0 newly installed, 0 to remove and 6 not
upgraded.
Need to get 9825kB of archives.
After this operation, 24.6kB of additional disk space will
be used.
Do you want to continue [Y/n]? Y
Get:1 http://update.onlinehome-server.info lenny/updates/main
dhcp3-client 3.1.1-6+lenny5 [225kB]
Get:2 http://update.onlinehome-server.info lenny/updates/main
dhcp3-common 3.1.1-6+lenny5 [291kB]
[...]
```



- Umstieg auf neue Distributionsversion:

- Repository-Einträge in sources.list auf neue Version umstellen
- Dann apt-get dist-upgrade ausführen
- Dabei werden evtl. einige Pakete entfernt, für die es in der neuen Version keinen Ersatz gibt



Übung:

- Tragen Sie die Zeile
`deb http://ftp.debian.org/debian/ squeeze main`
in die APT-Konfigurationsdatei ein (falls noch nicht vorhanden – nicht mit einem ähnlich aussehenden Eintrag für Updates verwechseln!)
- Aktualisieren Sie die Paketlisten
- Installieren Sie über die APT-Tools den grafischen Editor nedit

Hinweis zum Netzwerk: Wenn das Netzwerk nicht funktioniert, geben Sie in der Shell `su` (danach das root-Passwort), `killall dhclient` und `dhclient eth0` ein – danach sollte es gehen. Evtl. ist eine Anmeldung des Rechners im Browser (fom.de) nötig.

- In der „RPM-Welt“ gibt es kein einheitliches APT-ähnliches Tool für die Verwaltung von Repositories:
 - Fedora / Red Hat: YUM
 - Mandriva: URPMI
 - Suse: Zypper
- Wir betrachten hier nur YUM
- Prinzipien sind dieselben wie bei APT; aber Eintragen der Repos funktioniert anders

- Beispiel-Konfigurationsdatei für ein Repo:

```
[livna]
name=rpm.livna.org for $releasever - $basearch
#baseurl=http://rpm.livna.org/repo/$releasever/$basearch/ http://ftp-
stud.fht-esslingen.de/pub/Mirrors/rpm.livna.org/repo/$releasever/
$basearch/
mirrorlist=http://rpm.livna.org/mirrorlist
failovermethod=roundrobin
enabled=1
gpgcheck=1
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-livna
```

- anders als bei APT: diese Dateien nicht ändern
- einfach in den Ordner `/etc/yum.repos.d/` kopieren

- Red Hat / Fedora: YUM
 - Yellow Dog Updater, Modified
 - Yellow Dog Linux ist/war eine Linux-Distribution für PowerPC-Prozessoren, z. B. die alten Apple-Macs vor der Intel-Umstellung)
 - YUM war zunächst deren Paketmanager
 - Repository-Konfiguration: einzelne Dateien in `/etc/yum.repos.d/`
 - Paketverwaltung mit dem Befehl `yum`

- Neues Repo anlegen:
 - Repo-Anbieter stellt spezielle RPM-Datei zur Verfügung, welche das Repo automatisch einrichtet
 - Installation z. B. mit


```
rpm -i http://server/pfad/repopakete.rpm
```

 möglich
- Repos anzeigen: `yum repolist all`

```
server:~ # yum repolist all
Loading "installonlyn" plugin
repo id      repo name      status
development  Fedora - Development  disabled
development-debuginfo Fedora - Development - Debug  disabled
development-source Fedora - Development - Source  disabled
fedora        Fedora 7 - i386      enabled
fedora-debuginfo Fedora 7 - i386 - Debug  disabled
```

- Pakete suchen: `yum search`

```
server:~ # yum search mplayer
[...]  
mplayer.i386 1.0-0.75.20070513svn.1 livna  
Matched from:  
mplayer  
MPlayer is a movie player that plays most MPEG, VOB, AVI, OGG/OGM,  
VIVO, ASF/WMA/WMV, QT/MOV/MP4, FLI, RM, [...]
```

- Pakete installieren: `yum install`

```
server:~ # yum install w3m  
Setting up Install Process  
Parsing package install arguments  
Resolving Dependencies  
--> Running transaction check  
---> Package w3m.i386 0:0.5.2-1.fc7 set to be updated  
--> Processing Dependency: libgc.so.1 for package: w3m  
--> Restarting Dependency Resolution with new changes.  
[...]
```

- Alle Pakete aktualisieren: `yum update`
(vgl.: `apt-get upgrade`)
- gleicher Effekt wie bei APT
- Distributions-Upgrade:

- RPM-Pakete mit Repo-Informationen der neuen Version einspielen, z. B.:

```
rpm -Uvh ftp://download.fedora.redhat.com/pub/fedora/linux/releas  
es/8/Fedora/i386/os/Packages/fedora-release-8-3.noarch.rpm ftp://  
download.fedora.redhat.com/pub/fedora/linux/releases/8/Fedora/i38  
6/os/Packages/fedora-release-notes-8.0.0-3.noarch.rpm
```

- dann `yum upgrade` ausführen

- Pakete entfernen

```
server:~ # yum remove glibc  
[...]  
Transaction Summary  
=====
```

Install	0 Package(s)
Update	0 Package(s)
Remove	989 Package(s)

```
Is this ok [y/N]? n  
  
Exiting on user command.  
Complete!
```

- YUM schlägt vor, auch alle abhängigen Pakete automatisch mit zu entfernen

Betriebssysteme

SS 2015

Hans-Georg Eßer
Dipl.-Math., Dipl.-Inform.

Foliensatz F:

- Scheduling
- Prioritäten

v1.3, 2015/02/01

Scheduler und Prioritäten

Übersicht: BS Praxis und BS Theorie

Einführung
Shell
Dateiverwaltung
Filter
C-Compiler
Prozesse / Jobs
Threads
Interrupts

System Calls
Software-Verwaltung
Scheduler / Prioritäten
Synchronisation

Folien F

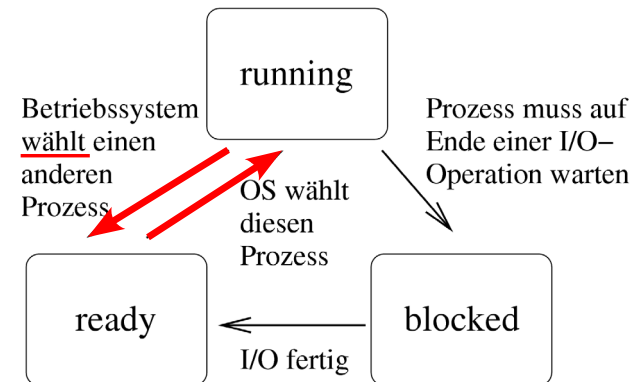
Gliederung

- Was ist Scheduling? Motivation
- Kooperatives / präemptives Scheduling
- CPU- und I/O-lastige Prozesse
- Ziele des Scheduling
(abhängig vom BS-Typ)
- Standard-Scheduling-Verfahren

Einführung

Scheduling: Prozess auswählen

Zustandsübergänge



Scheduling – worum geht es?

- Multitasking: Mehrere Prozesse konkurrieren um Betriebsmittel
- Betriebssystem verwaltet die Betriebsmittel
- Rechenzeit auf dem Prozessor
- Scheduler entscheidet: Welchen Prozess wann ausführen?
- Ausführreihenfolge entscheidend für Gesamt-Performance des Betriebssystems

Wann wird Scheduler aktiv?

- Neuer Prozess entsteht (fork)
- Aktiver Prozess blockiert wegen I/O-Zugriff
- Blockierter Prozess wird bereit
- Aktiver Prozess endet (exit)
- Prozess rechnet schon zu lange
- Interrupt tritt auf

Prozess-Unterbrechung möglich?

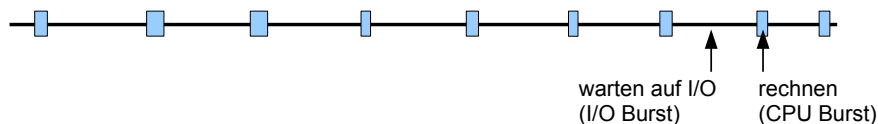
- Kooperatives Scheduling:
 - Prozess rechnet so lange, wie er will; bis zum nächsten I/O-Aufruf oder bis exit()
 - Scheduler wird nur bei Prozess-Blockieren oder freiwilliger CPU-Aufgabe aktiv
- Präemptives (unterbrechendes) Scheduling:
 - Timer aktiviert regelmäßig Scheduler, der neu entscheiden kann, „wo es weiter geht“

Faktoren

- **Zeit für Kontext-Switch:** Scheduler benötigt Zeit, um Prozesszustand zu sichern
→ verlorene Rechenzeit
- **Wartezeit der Prozesse:** Häufigere Wechsel erzeugen stärkeren Eindruck von Gleichzeitigkeit

I/O-lastig:

- Prozess hat zwischen I/O-Phasen nur kurze Berechnungsphasen (CPU)



CPU-lastig:

- Prozess hat zwischen I/O-Phasen lange Berechnungsphasen



Aus Anwendersicht

- **[A1] Ausführdauer:** Wie lange läuft der Prozess insgesamt?
- **[A2] Reaktionszeit:** Wie schnell reagiert der Prozess auf Benutzerinteraktion?
- **[A3] Deadlines** einhalten
- **[A4] Vorhersehbarkeit:** Gleichartige Prozesse sollten sich auch gleichartig verhalten, was obige Punkte angeht
- **[A5] Proportionalität:** „Einfaches“ geht schnell

Aus Systemsicht

- **[S1] Durchsatz:** Anzahl der Prozesse, die pro Zeit fertig werden
- **[S2] Prozessorauslastung:** Zeit (in %), die der Prozessor aktiv war
- **[S3] Fairness:** Prozesse gleich behandeln, keiner darf „verhungern“
- **[S4] Prioritäten** beachten
- **[S5] Ressourcen** gleichmäßig einsetzen

Wie schnell reagiert das System auf Benutzereingaben?

- Benutzer drückt Taste, klickt mit Maus etc. und wartet auf eine Reaktion
- Wie lang ist die Zeit zwischen Auslösen des Interrupts und Aktivierung des Prozesses, der die Eingabe auswertet?
- Toleranz bei langen Wartezeiten gering; schon 2–4 Sekunden kritisch, darüber inakzeptabel

Wie viel Zeit vergeht vom Programmstart bis zu seinem Ende?

- n Prozesse p_1 bis p_n starten zum Zeitpunkt t_0 und sind zu den Zeitpunkten t_1 bis t_n fertig
- Durchschnittliche Ausführdauer:
 $1/n \cdot \sum_i (t_i - t_0)$
- Abhängig von konkreten Prozessen;
Berechnung nur für Vergleich verschiedener Scheduling-Verfahren sinnvoll

Hält das System Deadlines ein?

- Realtime-Systeme: besondere Ansprüche
- Prozesse müssen in vorgegebener Zeit ihre Aufgaben erledigen, also ausreichend und rechtzeitig Rechenzeit erhalten
- Wie oft werden Deadlines nicht eingehalten?
- Optimierte (prozentuale) Anteil der eingehaltenen Deadlines

Ähnliches Verhalten ähnlicher Prozesse?

- Intuitiv: Gleichartige Prozesse sollten sich auch gleichartig verhalten, d. h.
 - Ausführdauer und Reaktionszeit immer ähnlich
 - Unabhängig vom sonstigen Zustand des Systems
- Schwierig, wenn das System beliebig viele Prozesse zulässt → Beschränkungen?

Terminierende Prozesse

- Anzahl der Prozesse, die pro Zeiteinheit (z. B. pro Stunde) fertig werden, sollte hoch sein
- Misst, wie viel Arbeit erledigt wird
- Abhängig von konkreten Prozessen; Berechnung nur für Vergleich verschiedener Scheduling-Verfahren sinnvoll

Vorgänge, die „einfach“ sind, werden schnell erledigt

- Es geht um das (evtl. falsche) Bild, das Anwender sich von technischen Abläufen machen
- Benutzer akzeptiert Wartezeit eher, wenn er den zugrunde liegenden Vorgang als komplex einschätzt

CPU immer gut beschäftigt halten

- Anteil der Taktzyklen, in denen die CPU nicht „idle“ war
- Interessanter Faktor, wenn Rechenzeit sehr wertvoll ist (kommerzielles Rechenzentrum)

Alle Prozesse haben gleiche Chancen

- Jeder Prozess sollte mal drankommen (kein „Verhungern“, engl. *process starvation*)
- Keine großen Abweichungen bei den Wartezeiten und Ausführdauern
- Falls Prozess-Prioritäten:
→ „manche sind gleicher“

„BS verwaltet die Betriebsmittel...“

- Grundidee des BS: alle Ressourcen gleichmäßig verteilen und gut auslasten
- CPU-Scheduler hat auch Einfluss auf (un)gleichmäßige Auslastung der I/O-Geräte
- Prozesse bevorzugen, die wenig ausgelastete Ressourcen nutzen wollen

Verschieden wichtige Prozesse auch verschieden behandeln

- Prioritätsklassen: Prozesse mit hoher Priorität bevorzugt behandeln
- Dabei verhindern, dass nur noch Prozesse mit hoher Priorität laufen (und alles andere steht)

Drei Kategorien

- Stapelverarbeitung
- Interaktives System
- Echtzeitsystem

Immer wichtig:

- S3 Fairness
- S4 Prioritäteneinsatz
- S5 Ressourcen-Balance

Stapelverarbeitung

- S3 Fairness
- S4 Prioritäteneinsatz
- S5 Ressourcen-Balance
- S1 Durchsatz
- A1 Ausführdauer
- S2 Prozessor-Auslastung

Echtzeitsystem

- S3 Fairness
- S4 Prioritäteneinsatz
- S5 Ressourcen-Balance
- A3 Deadlines
- A4 Vorhersehbarkeit

Interaktives System

- S3 Fairness
- S4 Prioritäteneinsatz
- S5 Ressourcen-Balance
- A2 Reaktionszeit
- A5 Proportionalität

Scheduler für Stapelverarbeitung (Batch-Systeme)

Eigenschaften der Stapelverarbeitung

- Nicht interaktives System (keine normalen Benutzerprozesse)
- Jobs werden über Job-Verwaltung abgesetzt; System informiert über Fertigstellung
- Typische Aufgaben: Lange Berechnungen, Kompilervorgänge

Moderne Batch-Systeme

- Normale Rechner (mit Platten, Netzwerk etc.)
- Kein interaktiver Betrieb (kein Login etc.)
- Job-Management-Tool nimmt Jobs an
- Long term scheduler entscheidet, wann ein Job gestartet wird – evtl. basierend auf Informationen über Ressourcenverbrauch und erwartete Laufzeit des Programms

Historisch:

- Batch-Betrieb mit Lochkarten
- Programm-Code und Daten auf Lochkarten
- Keine I/O (keine Geräte außer Kartenleser, Drucker)
- Kartenstapel (engl.: batch) legt Reihenfolge fest
 - Programm-Code von Karten lesen
 - Daten von Karten lesen
 - Berechnung durchführen
 - Ergebnis auf Karten stanzen oder drucken
 - Nächster Job

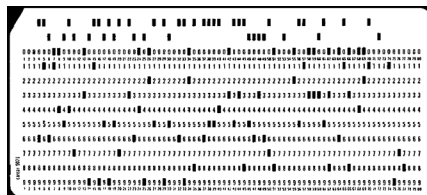


Bild: <http://www.fao.org/docrep/X5738E/x5738e0h.htm>

Scheduling-Verfahren für Batch-Betrieb

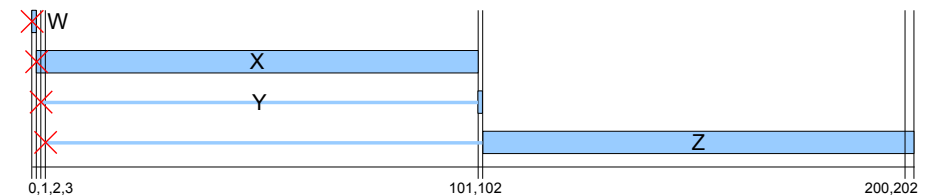
- First Come, First Served (FCFS)
- Shortest Job First (SJF)
- Shortest Remaining Time Next

Einfache Warteschlange

- Neue Prozesse reihen sich in Warteschlange ein
- Scheduler wählt jeweils nächsten Prozess in der Warteschlange
- Prozess arbeitet, bis er fertig ist (kooperatives Scheduling)

- FCFS bevorzugt lang laufende Prozesse
- Beispiel: 4 Prozesse W, X, Y, Z

Prozess	Ankunftszeit	Service Time T_s (Rechenzeit)	Startzeit	Endzeit	Turnaround T_r (Endzeit- Ankunftszeit)	T_r/T_s
W	0	1	0	1	1	1,00
X	1	100	1	101	100	1,00
Y	2	1	101	102	100	100,00
Z	3	100	102	202	199	1,99



Drei Prozesse mit
Rechendauern

T1: 15 Takte

T2: 4 Takte

T3: 3 Takte

Durchschnittliche
Ausführdauer:

$$a) (15+19+22) / 3 = 18,67$$

$$b) (3+7+22) / 3 = 10,67$$

$$c) (3+18+22) / 3 = 14,33$$



FCFS bevorzugt CPU-lastige Prozesse

- Während CPU-lastiger Prozess läuft, müssen alle anderen Prozesse warten
- I/O-lastiger Prozess kommt irgendwann dran, läuft nur sehr kurz und muss sich dann wieder hinten anstellen
- Ineffiziente Nutzung sowohl der CPU als auch der I/O-Geräte

- Keine Unterbrechungen (wie FCFS)
- Nächste Rechendauer (Burst) aller Prozesse bekannt oder wird geschätzt
- Strategie: Führe zunächst den Prozess aus, der am kürzesten laufen wird
- Minimiert die durchschnittliche Laufzeit aller Prozesse
- Prinzip war schon in FCFS-Beispiel erkennbar

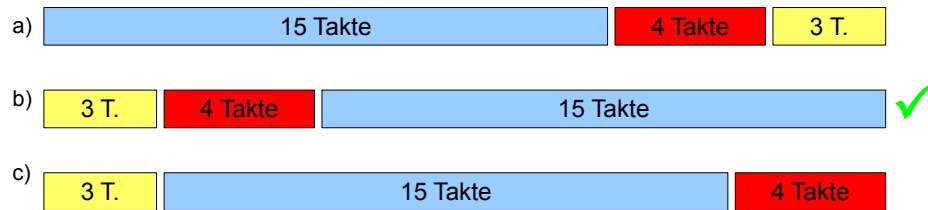
Generelles Problem:

Woher wissen, wie lange die Prozesse laufen?

- Batch-System; Programmierer muss Laufzeit schätzen
→ Bei grober Fehleinschätzung: Job abbrechen
- System, auf dem immer die gleichen / ähnliche Jobs laufen → Statistiken führen
- Interaktive Prozesse: Durchschnitt der bisherigen Burst-Längen berechnen

Ohne diese Information ist dieses Scheduling-Verfahren nur ein theoretisches

Im Beispiel von der FCFS-Folie:
Ausführreihenfolge b) entspricht SJF



Einfachste Variante: Mittelwert

$$S_{n+1} = \frac{1}{n} \sum_{i=1}^n T_i = \frac{1}{n} T_n + \frac{n-1}{n} S_n$$

mit:

T_i : Dauer des i -ten CPU-Burst des Prozess

S_i : Vorausgesagte Dauer des i -ten CPU-Burst

S_1 : Vorausgesagte Dauer des 1. CPU-Burst
(nicht berechnet)

Exponentieller Durchschnitt

$$S_{n+1} = \alpha T_n + (1 - \alpha) S_n \quad \alpha: \text{Gewicht zwischen 0 und 1}$$

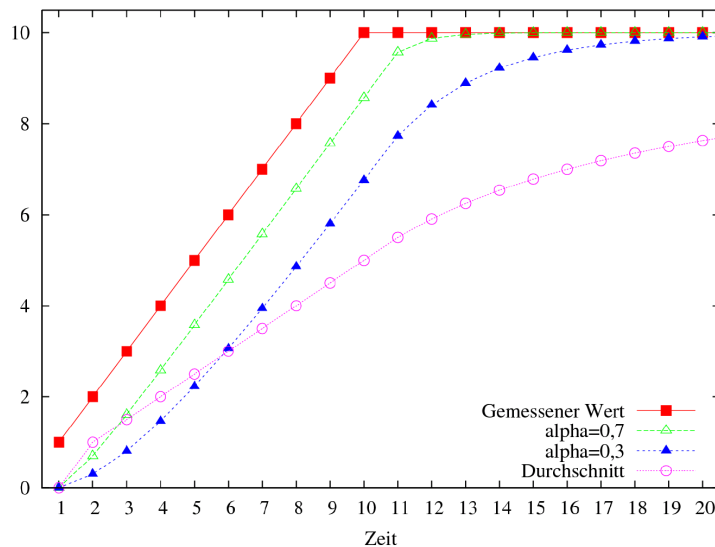
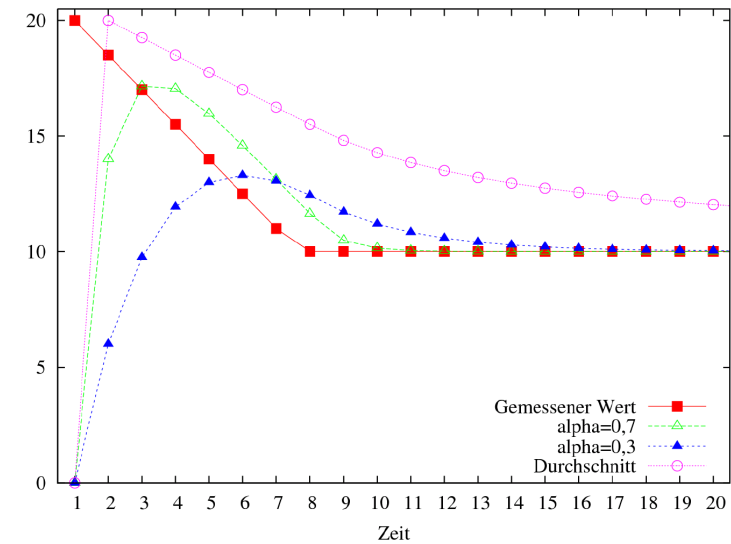
Beispiel: $\alpha = 0,8$:

$$S_2 = 0,8 T_1 + 0,2 S_1$$

$$S_3 = 0,8 T_2 + 0,2 S_2 = 0,8 T_2 + 0,2 (0,8 T_1 + 0,2 S_1)$$

$$\dots = 0,8 T_2 + 0,16 T_1 + 0,04 S_1$$

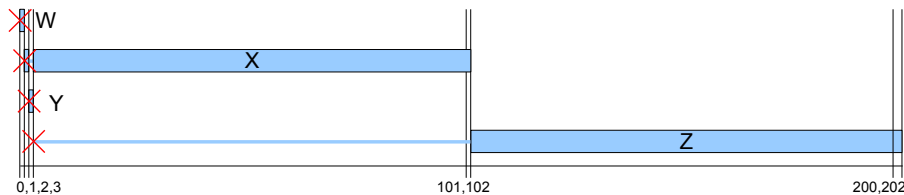
$$S_{n+1} = \sum_{i=0}^n (1 - \alpha)^{n-i} \alpha T_i \quad \text{mit} \quad T_0 := S_1$$



- Ähneln SJF, aber:
- präemptiv (mit Unterbrechungen)
- Regelmäßige Neuberechnung, wie viel Restzeit die Prozesse noch benötigen werden
- Scheduler prüft Reihenfolge immer, wenn ein neuer Job erzeugt wird
- Für kürzeren (auch neuen) Job wird der aktive unterbrochen
- Wie bei SJF gute Laufzeitprognose nötig

Altes FCFS-Beispiel: SRT unterbricht jetzt X:
Denn Y kommt zwar später, ist aber kürzer

Prozess	Ankunftszeit	Service Time T_s (Rechenzeit)	Startzeit	Endzeit	Turnaround T_r (Endzeit- Ankunftszeit)	T_r/T_s
W	0	1	0	1	1	1,00
X (1)	1	100	1	2 (*)		
Y	2	1	2	3	1	1,00
X (2)			3	102	102-1=101	1,01
Z	3	100	102	202	199	1,99



Scheduler für interaktive Systeme

SJF vs. SRT

- Vorsicht (auch für Prüfung): SJF und SRT werden oft verwechselt.
- „SJF < SRT“
 - (SJF ist i.d.R. schlechter als SRT, weil es keine Unterbrechungen kennt)
 - dieser Vergleich gilt auch lexikalisch (Eselsbrücke)

Interaktive Systeme

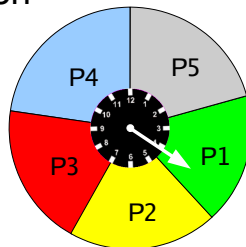
- Typisch: Interaktive und Hintergrund-Prozesse
- Desktop- und Server-PCs
- Eventuell mehrere / zahlreiche Benutzer, die sich die Rechenkapazität teilen
- Scheduler für interaktive Systeme prinzipiell auch für Batch-Systeme brauchbar (aber nicht umgekehrt)

Scheduling-Verfahren für interaktive Systeme

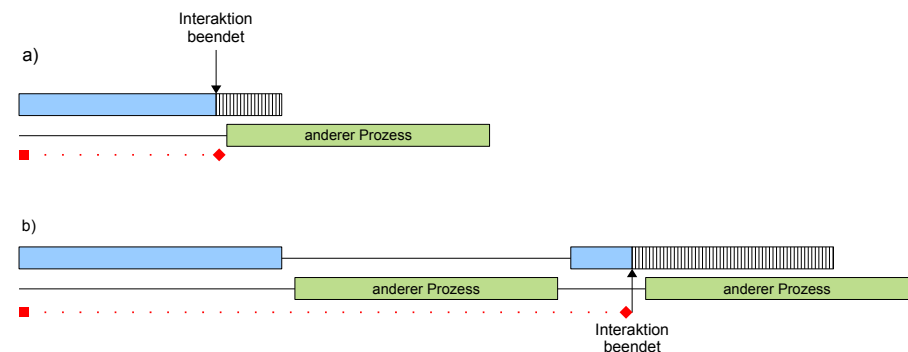
- Round Robin
- Prioritäten-Scheduler
- Lotterie-Scheduler

- Blockierten Prozess, der wieder bereit wird, hinten in Warteschlange einreihen
- Kriterien für Wahl des Quantums:
 - Größe muss in Verhältnis zur Dauer eines Context Switch stehen
 - Großes Quantum: evtl. lange Verzögerungen
 - Kleines Quantum: kurze Antwortzeiten, aber Overhead durch häufigen Context Switch

- Wie FCFS – aber mit Unterbrechungen
- Alle bereiten Prozesse in einer Warteschlange
- Jedem Thread eine Zeitscheibe (quantum, time slice) zuordnen
- Ist Prozess bei Ablauf der Zeitscheibe noch aktiv, dann:
 - Prozess verdrängen (preemption), also in den Zustand „bereit“ versetzen
 - Prozess ans Ende der Warteschlange hängen
 - Nächsten Prozess aus Warteschlange aktivieren

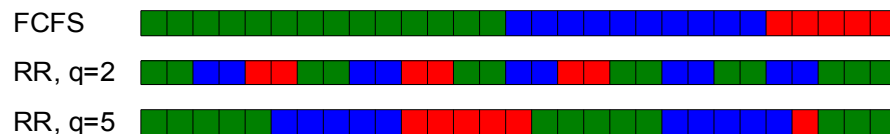


- Oft: Quantum q etwas größer als typische Zeit, die das Bearbeiten einer Interaktion benötigt



Szenario: Drei Prozesse

- FCFS (einfache Warteschlange, keine Unterbrechung)
- Round Robin mit Quantum 2
- Round Robin mit Quantum 5



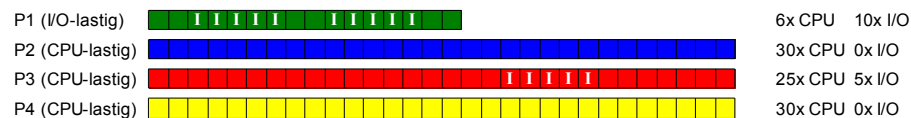
Beobachtung:

- Round Robin unfair gegenüber I/O-lastigen Prozessen:
 - CPU-lastige nutzen ganzes Quantum,
 - I/O-lastige nur einen Bruchteil

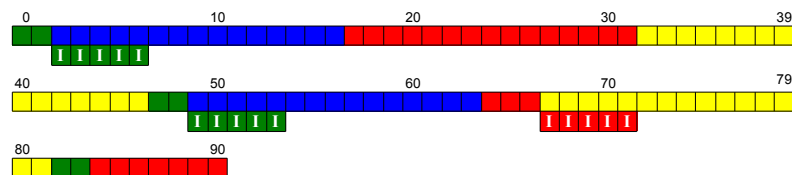
Lösungsvorschlag:

- Idee: Nicht verbrauchten Quantum-Teil als „Guthaben“ des Prozesses merken
- Sobald blockierter Prozess wieder bereit ist (I/O-Ergebnis da): Restguthaben sofort aufbrauchen

Idealer Verlauf (wenn jeder Prozess exklusiv läuft)



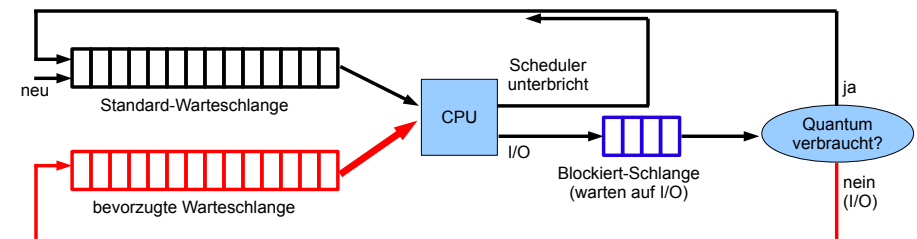
Ausführreihenfolge mit Round Robin, Zeitquantum 15:



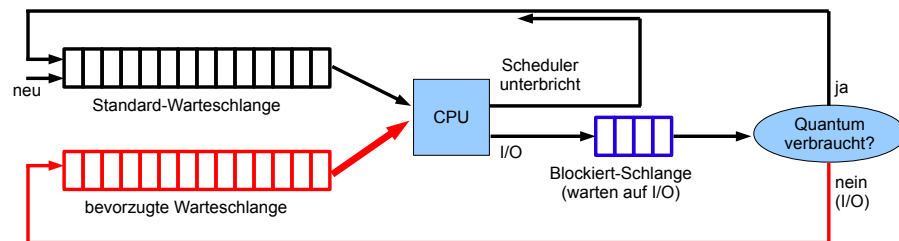
Prozess	CPU-Zeit	I/O-Zeit	Summe	Laufzeit	Wartezeit *)
P1	6	10	16	84	68
P2	30	0	30	64	34
P3	25	5	30	91	61
P4	30	0	30	82	52

*) im Zustand
bereit, nicht
blockiert!

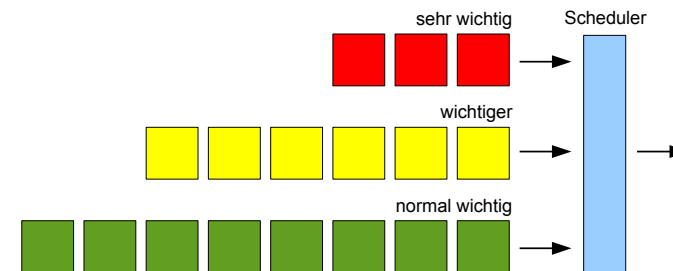
- Prozesse, die Zeitquantum verbrauchen, wie bei normalem Round Robin behandeln:
zurück in Warteschlange
- Prozesse, die wegen I/O blockieren und nur Zeit $u < q$ ihres Quantums verbraucht haben, bei Blockieren in Zusatzwarteschlange stecken



- Scheduler bevorzugt Prozesse in Zusatzschlange
- Quantum für diesen Prozess: $q-u$
(kriegt nur das, was ihm „zusteht“, was er beim letzten Mal nicht verbraucht hat)



- a) Mehrere Warteschlangen für Prioritätsklassen



- b) Scheduler sucht Prozess mit höchster Priorität *)



- Idee:
 - a) Prozesse in Prioritätsklassen einteilen oder
 - b) jedem Prozess einen Prioritätswert zuordnen
- Scheduler bevorzugt Prozesse mit hoher Prior.
- Priorität
 - bei Prozesserzeugung fest vergeben
 - oder vom Scheduler regelmäßig neu berechnen lassen
- Scheduling kooperativ oder präemptiv

Mehrere Warteschlangen

- Prozesse verschiedenen Prioritätsklassen zuordnen und in jeweilige Warteschlangen einreihen
- Scheduler aktiviert nur Prozesse aus der höchsten nicht-leeren Warteschlange
- Präemptiv: Prozesse nach Zeitquantum unterbrechen
- Innerhalb der Warteschlangen: Round Robin

Keine Hierarchien, sondern individuelle Prozess-Prioritäten

- Alle Prozesse stehen in einer Prozessliste
- Scheduler wählt stets Prozess mit der höchsten Priorität
- Falls mehrere Prozesse gleiche (höchste) Priorität haben, diese nach Round Robin verarbeiten

Aging:

- Priorität eines Prozesses, der bereit ist und auf die CPU wartet, wird regelmäßig erhöht
- Priorität des aktiven Prozesses und aller nicht-bereiten (blockierten) Prozesse bleibt gleich
- Ergebnis: Lange wartender Prozess erreicht irgendwann ausreichend hohe Priorität, um aktiv zu werden

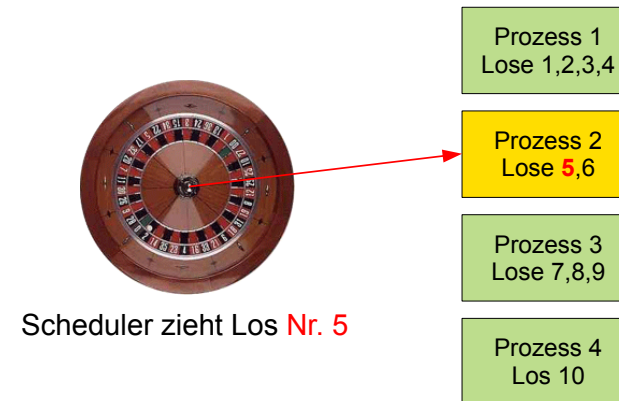
Prozesse können „verhungern“ → Aging**Prioritätsinversion:**

- Prozess hoher Priorität ist blockiert (benötigt ein Betriebsmittel)
- Prozess niedriger Priorität besitzt dieses Betriebsmittel, wird aber vom Scheduler nicht aufgerufen (weil es höher-prioritäre Prozesse gibt)
- Beide Prozesse kommen nie dran, weil immer Prozesse mittlerer Priorität laufen
- Ausweg: Aging

Verschiedene Quantenlängen

- Mehrere Prioritätsklassen:
 1. Priorität = 1 Quantum, 2. Priorität = 2 Quanten, 3. Priorität = 4 Quanten, 4. Priorität = 8 Quanten
- Prozesse mit hoher Priorität erhalten kleines Quantum.
- Geben sie die CPU vor Ablauf des Quantums zurück, behalten sie hohe Priorität
- Verbrauchen sie Quantum, verdoppelt Scheduler die Quantenlänge und stuft die Priorität runter – solange, bis Prozess sein Quantum nicht mehr aufbraucht

- Linux-Praxis: nice, renice
siehe Folien C-38 bis C-42
(aus Foliensatz zu Prozessen)



- Idee: Prozesse erhalten „Lotterie-Lose“ für die Verlosung von Ressourcen
- Scheduler zieht ein Los und lässt den Prozess rechnen, der das Los besitzt
- Priorisierung: Einige Prozesse erhalten mehr Lose als andere

- Gruppenbildung und Los-Austausch:
 - Zusammenarbeit Client / Server
 - Client stellt Anfrage an Server, gibt ihm seine Lose und blockiert
 - Nach Bearbeitung gibt Server die Lose an den Client zurück und weckt ihn auf
 - Keine Clients vorhanden?
→ Server erhält keine Lose, rechnet nie

- Aufteilung der Rechenzeit nur statistisch korrekt
- In konkreten Situationen verschieden lange Wartezeiten möglich
- Je länger mehrere Prozesse laufen, desto besser ist erwartete CPU-Aufteilung

- Multitasking auf einzelnen CPUs (oder nicht?)
- CPUs gleich-behandeln oder Master/Slaves?
- Zuordnung Prozess ↔ CPU: fest/variabel?
- BS-Instanz auf jeder CPU (was passiert, wenn zwei Scheduler denselben Prozess auswählen?)
- Gang Scheduling
- Dynamisches Scheduling

Literatur: William Stallings, „Operating Systems – Internals and Design Principles“, Kapitel 10

Betriebssysteme

SS 2015

Hans-Georg Eßer

Dipl.-Math., Dipl.-Inform.

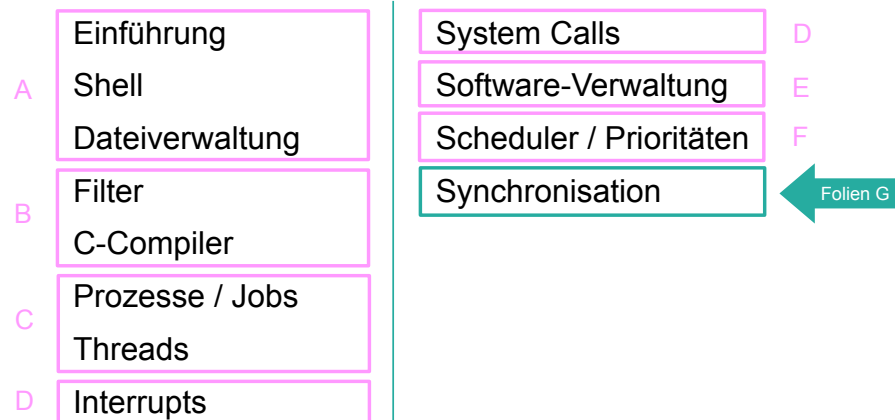
Foliensatz G:

- Synchronisation
- Deadlocks

v1.0, 2015/01/01

Synchronisation

Übersicht: BS Praxis und BS Theorie



Einführung (1)

- Es gibt Prozesse (oder Threads oder Kernel-Funktionen) mit gemeinsamem Zugriff auf bestimmte Daten, z. B.
 - Threads des gleichen Prozesses: gemeinsamer Speicher
 - Prozesse / Threads öffnen die gleiche Datei zum Lesen / Schreiben
 - Mehr-Prozessor-System: Scheduler (je einer pro CPU) greifen auf gleiche Prozesslisten / Warteschlangen zu

- Synchronisation: Probleme mit „gleichzeitigem“ Zugriff auf Datenstrukturen
- Beispiel: Zwei Threads erhöhen einen Zähler

```

erhoehe_zaeher() {
  w=read(Adresse);
  w=w+1;
  write(Adresse,w);
}

```

Ausgangssituation: w=10

T1: w=read(Adresse); // 10 w=w+1; // 11 ----- write(Adresse,w); // 11 !!	T2: w=read(Adresse); // 10 w=w+1; // 11 write(Adresse,w); // 11
--	--

Ergebnis nach T1, T2: w=11 – nicht 12!

- Ursache: `erhoehe_zaeher()` arbeitet nicht **atomar**:
 - Scheduler kann die Funktion unterbrechen
 - Funktion kann auf mehreren CPUs gleichzeitig laufen
- Lösung: Finde alle Code-Teile, die auf gemeinsame Daten zugreifen, und stelle sicher, dass immer nur ein Prozess auf diese Daten zugreift (gegenseitiger Ausschluss, mutual exclusion)

- Gewünscht wäre eine der folgenden Reihenfolgen:

Ausgangssituation: w=10 P1: w=read(Adr); // 10 w=w+1; // 11 write(Adr,w); // 11 ----- P2: w=read(Adr); // 11 w=w+1; // 12 write(Adr,w); // 12 Ergebnis nach P1, P2: w=12	Ausgangssituation: w=10 P1: w=read(Adr); // 10 w=w+1; // 11 write(Adr,w); // 11 ----- P2: w=read(Adr); // 11 w=w+1; // 12 write(Adr,w); // 12 Ergebnis nach P1, P2: w=12
---	---

- Analoges Problem bei Datenbanken:

```

exec sql CONNECT ...
exec sql SELECT kontostand INTO $var FROM Konto
      WHERE kontonummer = $knr
$var = $var - abhebung
exec sql UPDATE Konto SET kontostand = $var
      WHERE kontonummer = $knr
exec sql DISCONNECT

```

Bei parallelem Zugriff auf gleichen Datensatz kann es zu Fehlern kommen

→ Lösung über (Datenbank-) **Transaktionen**, die u. a. **atomar und isoliert (ACID-Prinzip)** erfolgen müssen

Race Condition:

- Mehrere parallele Threads / Prozesse nutzen eine gemeinsame Ressource
- Zustand hängt von Reihenfolge der Ausführung ab
- Race: die Threads liefern sich „ein Rennen“ um den ersten / schnellsten Zugriff

- Idee: Zugriff via Lock auf einen Prozess (Thread, ...) beschränken:

```
int Lock = UNLOCKED;           // global
erhoehe_zaebler( ) {
    while (Lock == LOCKED) { } ; // warten
    Lock = LOCKED;
    // Anfang Datennutzung
    w=read(Adresse);
    w=w+1;
    write(Adresse,w);
    // Ende Datennutzung
    Lock = UNLOCKED;
}
```

- Problem: Lock-Variable nicht geschützt

Warum Race Conditions vermeiden?

- Ergebnisse von parallelen Berechnungen sind nicht eindeutig (d. h. potenziell falsch)
- Bei Programmtests könnte (durch Zufall) immer eine „korrekte“ Ausführreihenfolge auftreten; später beim Praxiseinsatz dann aber gelegentlich eine „falsche“.
- Race Conditions sind auch Sicherheitslücken

- Nicht alle Zugriffe sind problematisch:
 - Gleichzeitiges Lesen von Daten stört nicht
 - Prozesse, die „disjunkt“ sind (d. h.: die keine gemeinsamen Daten haben) können ohne Schutz zugreifen
- Sobald mehrere Prozesse/Threads/... gemeinsam auf ein Objekt zugreifen – und mindestens einer davon schreibend –, ist das Verhalten des Gesamtsystems **unvorhersehbar** und **nicht reproduzierbar**.

- Einführung, Race Conditions
- Kritische Abschnitte und gegenseitiger Ausschluss
- Synchronisationsmethoden, Standard-Primitive:
 - Mutexe
 - Semaphore

- Bestimmen des kritischen Abschnitts nicht ganz eindeutig:

```
void test () {  
    z = global[i];  
    z = z + 1;  
    global[i] = z;  
    // was anderes tun  
    z = global[j];  
    z = z - 1;  
    global[j] = z;  
}
```

- zwei kritische Abschnitte oder nur einer?

- Programmteil, der auf gemeinsame Daten zugreift
 - Müssen nicht verschiedene Funktionen sein: auch mehrere Instanzen (z. B. Threads), welche dieselbe Funktion ausführen
- Block zwischen erstem und letztem Zugriff
- Nicht „den Code schützen“, sondern die Daten
- Formulierung: kritischen Abschnitt „betreten“ und „verlassen“ (enter / leave critical section)

- Anforderung an parallele Threads:
 - Es darf maximal ein Thread gleichzeitig im kritischen Abschnitt sein
 - Kein Thread, der außerhalb kritischer Abschnitte ist, darf einen anderen blockieren
 - Kein Thread soll ewig auf das Betreten eines kritischen Abschnitts warten
 - Deadlocks sollen vermieden werden (z. B.: zwei Prozesse sind in verschiedenen kritischen Abschnitten und blockieren sich gegenseitig)

- Tritt nie mehr als ein Thread gleichzeitig in den kritischen Abschnitt ein, heißt das „**gegenseitiger Ausschluss**“ (englisch: **mutual exclusion**, kurz: mutex)
- Es ist Aufgabe der Programmierer, diese Bedingung zu garantieren
- Das Betriebssystem bietet Hilfsmittel, mit denen gegenseitiger Ausschluss durchgesetzt werden kann, schützt aber nicht vor Programmierfehlern

- **TSL** muss zwei Dinge leisten:
 - Interrupts ausschalten, damit der Test-und-Setzen-Vorgang nicht durch einen anderen Prozess unterbrochen wird
 - Im Falle mehrerer CPUs den Speicherbus sperren, damit kein Prozess auf einer anderen CPU (deren Interrupts nicht gesperrt sind!) auf die gleiche Variable zugreifen kann

- **Maschineninstruktion** (z. B. mit dem Namen **TSL = Test and Set Lock**), die **atomar** eine Lock-Variable liest und setzt, also ohne dazwischen unterbrochen werden zu können.

```
enter:
    tsl register, flag ; Variablenwert in Register kopieren und
                        ; dann Variable auf 1 setzen
    cmp register, 0    ; War die Variable 0?
    jnz enter         ; Nicht 0: Lock war gesetzt, also Schleife
    ret

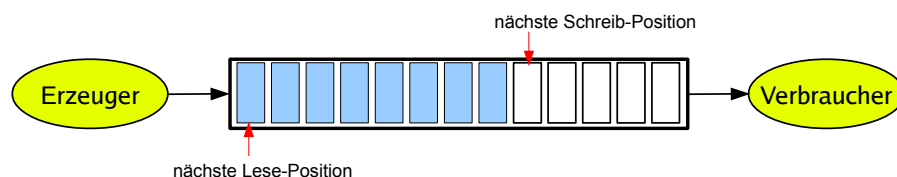
leave:
    mov flag, 0        ; 0 in flag speichern: Lock freigeben
    ret
```

- **Aktives Warten (busy waiting)**:
 - Ausführen einer Schleife, bis eine Variable einen bestimmten Wert annimmt.
 - Der **Thread ist bereit** und **belegt die CPU**.
 - Die Variable muss von einem anderen Thread gesetzt werden.
 - (Großes) Problem, wenn der andere Thread endet.
 - (Großes) Problem, wenn der andere Thread – z. B. wegen niedriger Priorität – nicht dazu kommt, die Variable zu setzen.

- **Passives Warten (sleep and wake):**
 - Ein Thread **blockiert** und wartet auf ein Ereignis, das ihn wieder in den Zustand „bereit“ versetzt.
 - Blockierter Thread **verschwendet keine CPU-Zeit**.
 - Ein anderer Thread muss das Eintreten des Ereignisses bewirken.
 - (Kleines) Problem, wenn der andere Thread endet.
 - Bei Eintreten des Ereignisses muss der blockierte Thread geweckt werden, z. B.
 - explizit durch einen anderen Thread,
 - durch Mechanismen des Betriebssystems.

- **Synchronisation**
 - **Puffer nicht überfüllen:**
Wenn der Puffer voll ist, muss der Erzeuger warten, bis der Verbraucher eine Information aus dem Puffer abgeholt hat, und erst dann weiter arbeiten.
 - **Nicht aus leerem Puffer lesen:**
Wenn der Puffer leer ist, muss der Verbraucher warten, bis der Erzeuger eine Information im Puffer abgelegt hat, und erst dann weiter arbeiten.

- Beim **Erzeuger-Verbraucher-Problem** (producer consumer problem, bounded buffer problem) gibt es zwei kooperierende Threads:
 - Der Erzeuger speichert Informationen in einem **beschränkten Puffer**.
 - Der Verbraucher liest Informationen aus diesem Puffer.



- Realisierung mit passivem Warten:
 - Eine gemeinsam benutzte Variable „count“ zählt die belegten Positionen im Puffer.
 - Wenn der Erzeuger eine Information einstellt und der Puffer leer war ($\text{count} == 0$), weckt er den Verbraucher;
bei vollem Puffer blockiert er.
 - Wenn der Verbraucher eine Information abholt und der Puffer voll war ($\text{count} == \text{max}$), weckt er den Erzeuger;
bei leerem Puffer blockiert er.

```
#define N 100                // Anzahl der Plätze im Puffer
int count = 0;              // Anzahl der belegten Plätze im Puffer

producer () {
    while (TRUE) {          // Endlosschleife
        produce_item (item); // Erzeuge etwas für den Puffer
        if (count == N) sleep(); // Wenn Puffer voll: schlafen legen
        enter_item (item);   // In den Puffer einstellen
        count = count + 1;    // Zahl der belegten Plätze inkrementieren
        if (count == 1) wake(consumer); // war der Puffer vorher leer?
    }
}

consumer () {
    while (TRUE) {          // Endlosschleife
        if (count == 0) sleep(); // Wenn Puffer leer: schlafen legen
        remove_item (item); // Etwas aus dem Puffer entnehmen
        count = count - 1;    // Zahl der belegten Plätze dekrementieren
        if (count == N-1) wake(producer); // war der Puffer vorher voll?
        consume_item (item); // Verarbeiten
    }
}
```

- **Problemursache:**
Wakeup-Signal für einen – noch nicht – schlafenden Prozess wird ignoriert
- Falsche Reihenfolge
- Weckruf „irgendwie“ für spätere Verwendung aufbewahren...

VERBRAUCHER	ERZEUGER
n=read(count);	..
..	produce_item();
..	n=read(count);
..	/* n=0 */
..	n=n+1;
..	write(n,count);
..	wake(VERBRAUCHER);
/* n=0 */	..
sleep();	..

- Das Programm enthält eine *race condition*, die zu einem *Deadlock* führen kann, z. B. wie folgt:
 - Verbraucher liest Variable count, die den Wert 0 hat.
 - Kontextwechsel zum Erzeuger.
 - Erzeuger stellt etwas in den Puffer ein, erhöht count und weckt den Verbraucher, da count vorher 0 war.
 - Verbraucher legt sich schlafen, da er für count noch den Wert 0 gespeichert hat (der zwischenzeitlich erhöht wurde).
 - Erzeuger schreibt den Puffer voll und legt sich dann auch schlafen.

- Lösungsmöglichkeit: Systemaufrufe *sleep* und *wake* verwenden ein „**wakeup pending bit**“:
 - Bei *wake()* für einen nicht schlafenden Thread dessen wakeup pending bit setzen.
 - Bei *sleep()* das wakeup pending bit des Threads überprüfen – wenn es gesetzt ist, den Thread nicht schlafen legen.
- Aber: Lösung lässt sich nicht verallgemeinern (mehrere zu synchronisierende Prozesse benötigen evtl. zusätzliche solche Bits)

Ein **Semaphor** ist eine Integer- (Zähler-) Variable, die man wie folgt verwendet:

- Semaphor hat festgelegten Anfangswert N („Anzahl der verfügbaren Ressourcen“).
- Beim **Anfordern** eines Semaphors (P- oder **Wait**-Operation): P = (niederl.) probeer
 - Semaphor-Wert um 1 erniedrigen, falls er >0 ist,
 - Thread blockieren und in eine Warteschlange einreihen, wenn der Semaphor-Wert 0 ist.

- Bei **Freigabe** eines Semaphors (V- oder **Signal**-Operation): V = (niederl.) vrijgeven
 - einen Thread aus der Warteschlange wecken, falls diese nicht leer ist,
 - Semaphor-Wert um 1 erhöhen (wenn es keinen auf den Semaphor wartenden Thread gibt)
- Code sieht dann immer so aus:


```
wait (&sem);
/* Code, der die Ressource nutzt */
signal (&sem);
```
- in vielen Büchern: **P**(&sem), **V**(&sem)

- Pseudo-Code für Semaphor-Operationen

<pre>wait (sem) { if (sem>0) sem--; else { ADD_CALLER_TO (QUEUE(sem)); SLEEP; } }</pre>	<pre>signal (sem) { if (P in QUEUE(sem)) { WAKEUP (P); REMOVE (P, QUEUE); } else sem++; }</pre>
--	---

- **Mutex**: boolesche Variable (true/false), die den Zugriff auf gemeinsam genutzte Daten synchronisiert
 - true: Zugang erlaubt
 - false: Zugang verboten
- **blockierend**: Ein Thread, der sich Zugang verschaffen will, während ein anderer Thread Zugang hat, blockiert → Warteschlange
- Bei Freigabe:
 - Warteschlange enthält Threads → einen wecken
 - Warteschlange leer: Mutex auf true setzen

- **Mutex (mutual exclusion) = binärer Semaphore**, also ein Semaphore, der nur die Werte 0 / 1 annehmen kann. Pseudo-Code:

```
wait (mutex) {
    if (mutex==1)
        mutex=0;
    else {
        ADD_CALLER_TO (QUEUE(mutex));
        SLEEP
    }
}

signal (mutex) {
    if (P in QUEUE(mutex)) {
        WAKEUP (P);
        REMOVE (P, QUEUE);
    }
    else
        mutex=1;
}
```

- Neue Interpretation: wait → lock
signal → unlock
- Mutexe für exklusiven Zugriff (kritische Abschnitte)

- Bei Mutexen / Semaphoren müssen die beiden Operationen wait() und signal() **atomar** implementiert sein:

Während der Ausführung von wait() / signal() darf kein anderer Prozess an die Reihe kommen

- Betriebssysteme können Mutexe und Semaphore **blockierend** oder **nicht-blockierend** implementieren
- blockierend:
wenn der Versuch, den Zähler zu erniedrigen, scheitert
→ warten
- nicht blockierend:
wenn der Versuch scheitert
→ vielleicht etwas anderes tun

- Mutexe / Semaphore verwalten Warteschlangen (der Prozesse, die schlafen gelegt wurden)
- Beim Aufruf von signal() muss evtl. ein Prozess geweckt werden
- Auswahl des zu weckenden Prozesses ist ein ähnliches Problem wie die Prozess-Auswahl im Scheduler
 - FIFO: **starker** Semaphore / Mutex
 - zufällig: **schwacher** Semaphore / Mutex

Erzeuger-Verbraucher-Problem mit Semaphoren und Mutexen

```
typedef int semaphore;
semaphore mutex = 1;      // kontrolliert Zugriff auf Puffer
semaphore empty = N;      // zählt freie Plätze im Puffer
semaphore full = 0;       // zählt belegte Plätze im Puffer

producer() {
    while (TRUE) {        // Endlosschleife
        produce_item(item); // erzeuge etwas für den Puffer
        wait (empty);      // leere Plätze dekrementieren bzw. blockieren
        wait (mutex);      // Eintritt in den kritischen Abschnitt
        enter_item (item); // in den Puffer einstellen
        signal (mutex);    // kritischen Abschnitt verlassen
        signal (full);     // belegte Plätze erhöhen, evtl. consumer wecken
    }
}

consumer() {
    while (TRUE) {        // Endlosschleife
        wait (full);       // belegte Plätze dekrementieren bzw. blockieren
        wait (mutex);      // Eintritt in den kritischen Abschnitt
        remove_item(item); // aus dem Puffer entnehmen
        signal (mutex);    // kritischen Abschnitt verlassen
        signal (empty);    // freie Plätze erhöhen, evtl. producer wecken
        consume_entry (item); // verbrauchen
    }
}
```

Deadlocks – Gliederung

- Einführung
- Ressourcen-Typen
- Hinreichende und notwendige Deadlock-Bedingungen
- Deadlock-Erkennung und -Behebung
- Deadlock-Vermeidung (avoidance): Banker-Algorithmus
- Deadlock-Verhinderung (prevention)

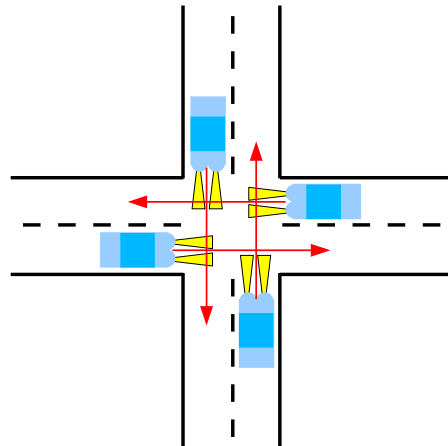
Deadlocks

Was ist ein Deadlock?

- Eine Menge von Prozessen befindet sich in einer **Deadlock-Situation**, wenn:
 - jeder Prozess auf eine Ressource wartet, die von einem anderen Prozess blockiert wird
 - keine der Ressourcen freigegeben werden kann, weil der haltende Prozess (indem er selbst wartet) blockiert ist
- In einer Deadlock-Situation werden also die Prozesse dauerhaft verharren
- Deadlocks sind unbedingt zu vermeiden

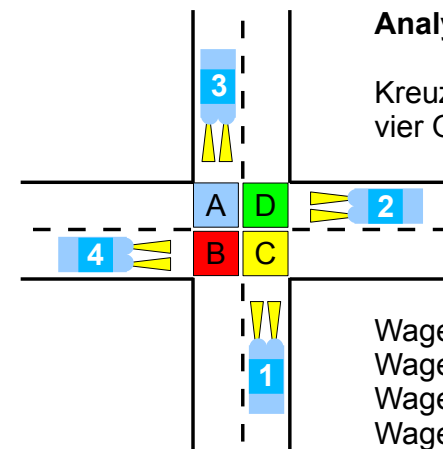
Deadlock: Rechts vor Links (1)

- Der Klassiker: Rechts-vor-Links-Kreuzung



Wer darf fahren?
Potenzieller Deadlock

Deadlock: Rechts vor Links (3)

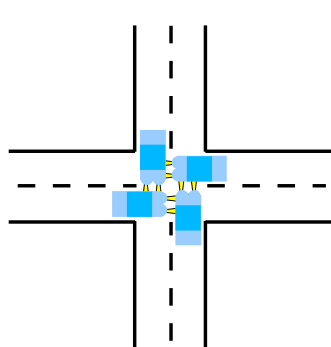


Analyse:

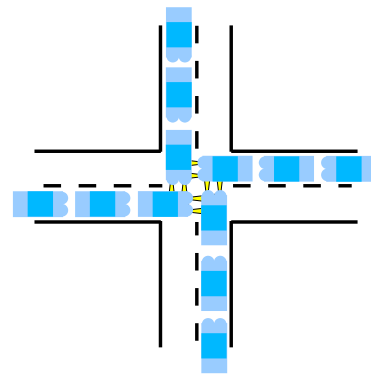
Kreuzungsbereich besteht aus vier Quadranten A, B, C, D

Wagen 1 benötigt C, D
Wagen 2 benötigt D, A
Wagen 3 benötigt A, B
Wagen 4 benötigt B, C

Deadlock: Rechts vor Links (2)

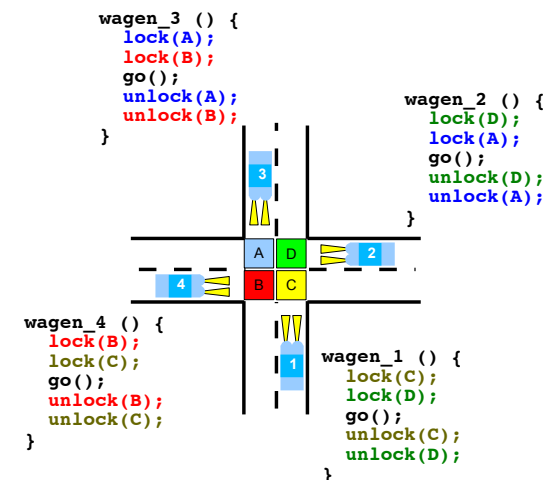


Deadlock, aber behebbar:
eines oder mehrere Autos
können zurücksetzen



Deadlock, nicht behebbar:
beteiligte Autos können nicht
zurücksetzen

Deadlock: Rechts vor Links (4)



**Problematische
Reihenfolge:**

w1: lock(C)
w2: lock(D)
w3: lock(A)
w4: lock(B)
w1: lock(D) ← blockiert
w2: lock(A) ← blockiert
w3: lock(B) ← blockiert
w4: lock(C) ← blockiert

Deadlock: kleinstes Beispiel (1)

- Zwei Locks A und B
 - z. B. A = Scanner, B = Drucker,
 - Prozesse P, Q wollen beide eine Kopie erstellen
- Locking in verschiedenen Reihenfolgen

Prozess P

```
lock (A);
lock (B);

/* krit. Bereich */

unlock (A);
unlock (B);
```

Prozess Q

```
lock (B);
lock (A);

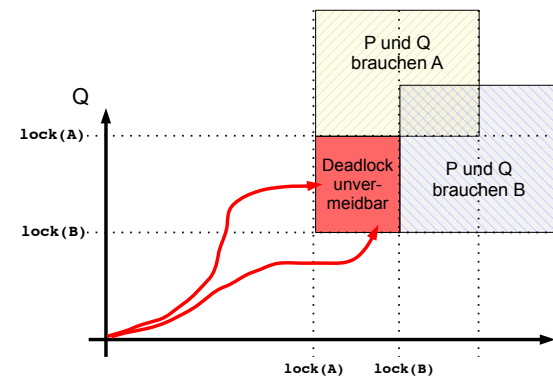
/* krit. Bereich */

unlock (B);
unlock (A);
```

Problematische Reihenfolge:

```
P: lock(A)
Q: lock(B)
P: lock(B) ← blockiert
Q: lock(A) ← blockiert
```

Deadlock: kleinstes Beispiel (3)



Programmverzahnungen, die zwangsläufig in den Deadlock führen:

oberer roter Weg:

```
Q: lock(B)
P: lock(A)
```

unterer roter Weg:

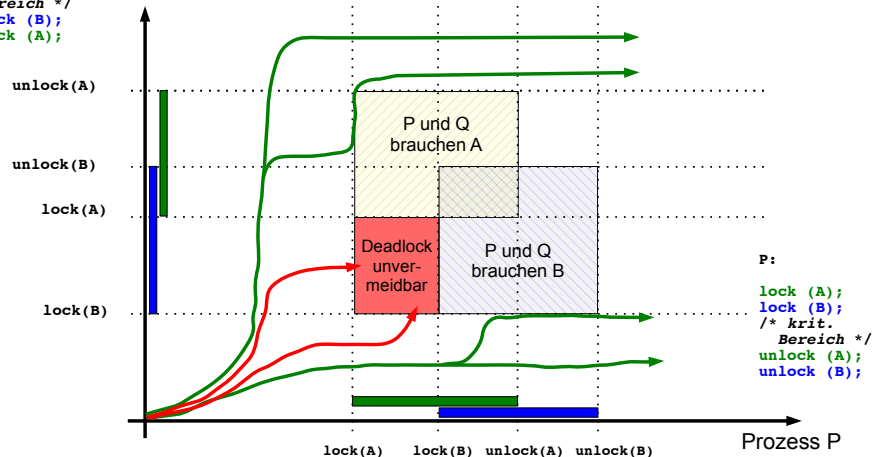
```
P: lock(A)
Q: lock(B)
```

Deadlock: kleinstes Beispiel (2)

Q:

```
lock (B);
lock (A);
/* krit. Bereich */
unlock (B);
unlock (A);
```

Prozess Q



Deadlock: kleinstes Beispiel (4)

- Problem beheben:
P benötigt die Locks nicht gleichzeitig

Prozess P

```
lock (A);
/* krit. Bereich */
unlock (A);

lock (B);
/* krit. Bereich */
unlock (B);
```

Prozess Q

```
lock (B);
lock (A);
/* krit. Bereich */
unlock (B);
unlock (A);
```

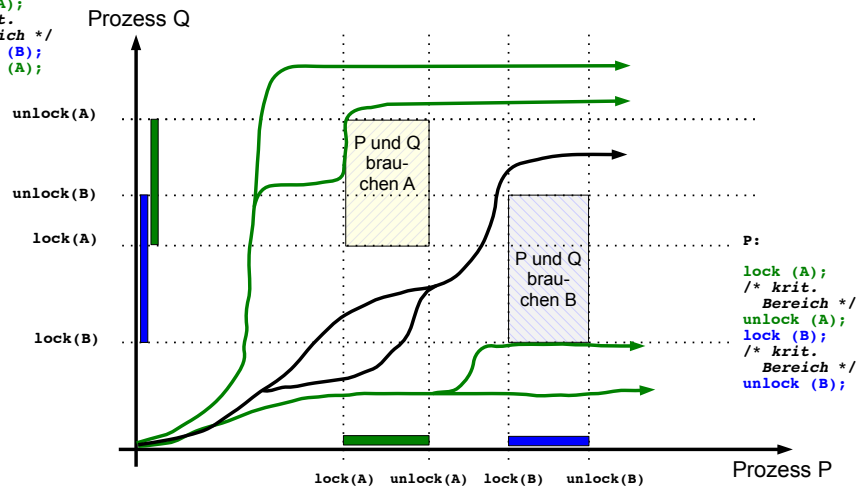
Jetzt kann kein Deadlock mehr auftreten

- Andere Lösung: P und Q fordern A, B in gleicher Reihenfolge an

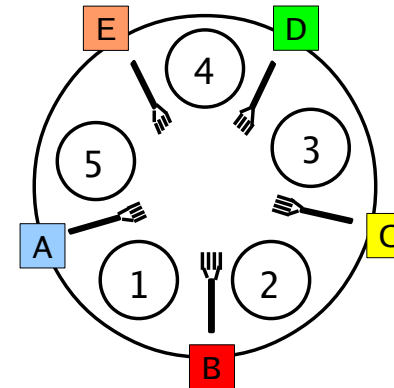
Deadlock: kleinstes Beispiel (5)

Q:

```
lock (B);
lock (A);
/* krit.
Bereich */
unlock (B);
unlock (A);
```



Fünf-Philosophen-Problem

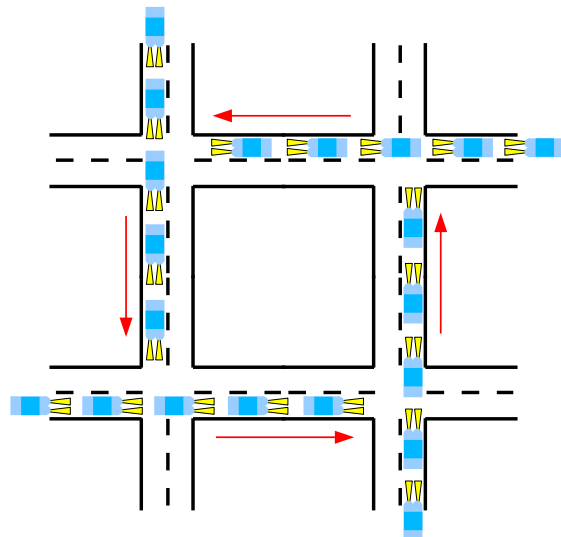


Philosoph 1 braucht Gabeln A, B
 Philosoph 2 braucht Gabeln B, C
 Philosoph 3 braucht Gabeln C, D
 Philosoph 4 braucht Gabeln D, E
 Philosoph 5 braucht Gabeln E, A

Problematische Reihenfolge:

p1: lock (B)
 p2: lock (C)
 p3: lock (D)
 p4: lock (E)
 p5: lock (A)
 p1: lock (A) ← blockiert
 p2: lock (B) ← blockiert
 p3: lock (C) ← blockiert
 p4: lock (D) ← blockiert
 p5: lock (E) ← blockiert

Grid Lock



Ressourcen-Typen (1)

Zwei Kategorien von Ressourcen: unterbrechbar / nicht unterbrechbar

- unterbrechbare Ressourcen
 - Betriebssystem kann einem Prozess solche Ressourcen wieder entziehen
 - Beispiele:
 - CPU (Scheduler)
 - Hauptspeicher (Speicherverwaltung)
 - das kann Deadlocks vermeiden

- nicht unterbrechbare Ressourcen
 - Betriebssystem kann Ressource nicht (ohne fehlerhaften Abbruch) entziehen – Prozess muss diese freiwillig zurückgeben
 - Beispiele:
 - DVD-Brenner (Entzug → zerstörter Rohling)
 - Tape-Streamer (Entzug → sinnlose Daten auf Band oder Abbruch der Bandsicherung wegen Timeout)
- Nur die *nicht* unterbrechbaren sind interessant, weil sie Deadlocks verursachen können

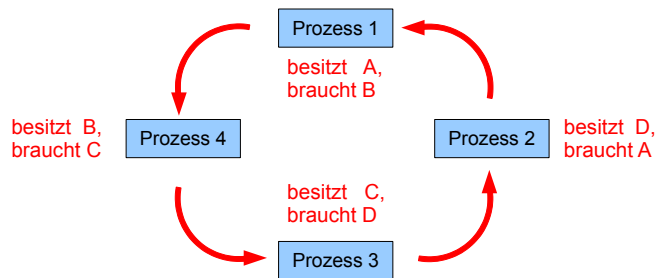
1. **Gegenseitiger Ausschluss (mutual exclusion)**
 - Ressource ist exklusiv: Es kann stets nur ein Prozess darauf zugreifen
2. **Hold and Wait (besitzen und warten)**
 - Ein Prozess ist bereits im Besitz einer oder mehrerer Ressourcen, und
 - er kann noch weitere anfordern
3. **Ununterbrechbarkeit der Ressourcen**
 - Die Ressource kann nicht durch das Betriebssystem entzogen werden

- wiederverwendbare vs. konsumierbare Ressourcen
 - **wiederverwendbar:** Zugriff auf Ressource zwar exklusiv, aber nach Freigabe wieder durch anderen Prozess nutzbar (Platte, RAM, CPU, ...)
 - **konsumierbar:** von einem Prozess erzeugt und von einem anderen Prozess konsumiert (Nachrichten, Interrupts, Signale, ...)

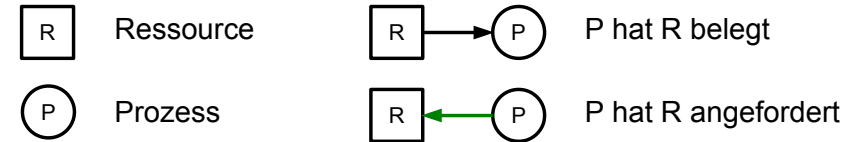
- (1) bis (3) sind **notwendige** Bedingungen für einen Deadlock
- (1) bis (3) sind aber auch „wünschenswerte“ Eigenschaften eines Betriebssystems, denn:
 - gegenseitiger Ausschluss ist nötig für korrekte Synchronisation
 - Hold & Wait ist nötig, wenn Prozesse exklusiven Zugriff auf mehrere Ressourcen benötigen
 - Bei manchen Betriebsmitteln ist eine Präemption prinzipiell nicht sinnvoll (z. B. DVD-Brenner, Streamer)

4. Zyklisches Warten

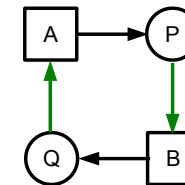
- Man kann die Prozesse in einem Kreis anordnen, in dem jeder Prozess eine Ressource benötigt, die der folgende Prozess im Kreis belegt hat



- Belegung und (noch unerfüllte) Anforderung grafisch darstellen:



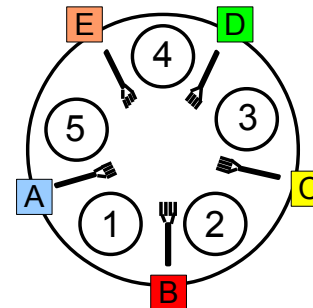
- P, Q aus Minimalbeispiel:
- Deadlock = Kreis im Graph



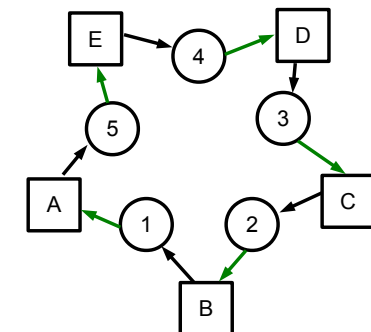
- Gegenseitiger Ausschluss
- Hold and Wait
- Ununterbrechbarkeit der Ressourcen
- Zyklisches Warten

- (1) bis (4) sind **notwendige und hinreichende** Bedingungen für einen Deadlock
- Das zyklische Warten (4) (und dessen Unauflösbarkeit) sind Konsequenzen aus (1) bis (3)
- (4) ist der erfolgversprechendste Ansatzpunkt, um Deadlocks aus dem Weg zu gehen

Philosophen-Beispiel



Situation, nachdem alle Philosophen ihre rechte Gabel aufgenommen haben

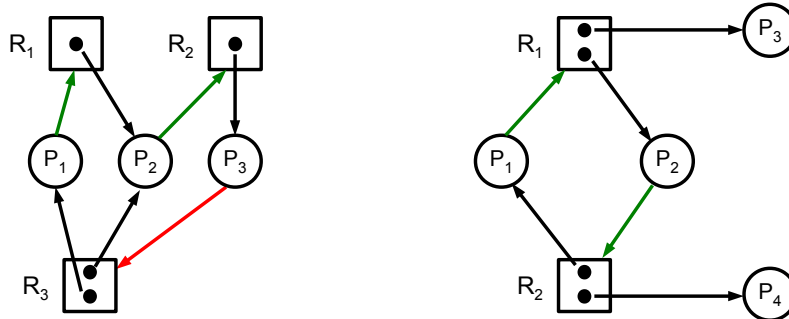


- Variante für Ressourcen, die mehrfach vorkommen können



- Idee: Deadlocks zunächst zulassen
- System regelmäßig auf Vorhandensein von Deadlocks überprüfen und diese dann abstellen
- Nutzt drei Datenstrukturen:
 - Belegungsmatrix
 - Ressourcenrestvektor
 - Anforderungsmatrix

- Beispiele mit mehreren Instanzen



Mit roter Kante ($P_3 \rightarrow R_3$) gibt es einen Deadlock (ohne nicht)

Kreis, aber kein Deadlock – Bedingung ist nur **notwendig**, nicht hinreichend!

- n Prozesse $P_1, \dots, P_n$
- m Ressourcentypen $R_1, \dots, R_m$
 Vom Typ R_i gibt es E_i Ressourcen-Instanzen ($i=1, \dots, m$)
 $\rightarrow$ **Ressourcenvektor** $E = (E_1 \ E_2 \ \dots \ E_m)$
- Ressourcenrestvektor** A (wie viele sind noch frei?)
- Belegungsmatrix** C
 C_{ij} = Anzahl Ressourcen vom Typ j , die von Prozess i belegt sind
- Anforderungsmatrix** R
 R_{ij} = Anzahl Ressourcen vom Typ j , die Prozess i noch benötigt

• Beispiel:

	ISDN-Karte	Streamer	Scanner	DVD-Brenner		ISDN-Karte	Streamer	Scanner	DVD-Brenner	
$E = (1\ 3\ 2\ 4)$					Ressourcenvektor	$A = (0\ 0\ 1\ 2)$				Ressourcenrestvektor
$C = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 2 \\ 0 & 2 & 1 & 0 \end{pmatrix}$					Belegungsmatrix	$R = \begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 2 \end{pmatrix}$				Anforderungsmatrix
										Prozess 1
										Prozess 2
										Prozess 3

- Alle Prozesse, die nach diesem Algorithmus nicht markiert sind, sind an einem Deadlock beteiligt

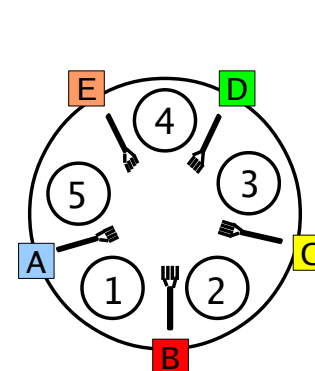
• Beispiel

$E = (1\ 3\ 2\ 4)$	$A = (0\ 0\ 1\ 2)$	$E = (1\ 3\ 2\ 4)$	$A = (1\ 2\ 2\ 4)$
$C = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 2 \\ 0 & 2 & 1 & 0 \end{pmatrix}$	$R = \begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 2 \end{pmatrix}$	$C = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 2 \\ 0 & 2 & 1 & 0 \end{pmatrix}$	$R = \begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 2 \end{pmatrix}$
$E = (1\ 3\ 2\ 4)$	$A = (0\ 2\ 2\ 2)$	$E = (1\ 3\ 2\ 4)$	$A = (1\ 3\ 2\ 4)$
$C = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 2 \\ 0 & 2 & 1 & 0 \end{pmatrix}$	$R = \begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 2 \end{pmatrix}$	$C = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 2 \\ 0 & 2 & 1 & 0 \end{pmatrix}$	$R = \begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 2 \end{pmatrix}$

Algorithmus

1. Suche einen unmarkierten Prozess P_i , dessen verbleibende Anforderungen vollständig erfüllbar sind, also $R_{ij} \leq A_j$ für alle j
2. Gibt es keinen solchen Prozess, beende Algorithmus
3. Ein solcher Prozess könnte erfolgreich abgearbeitet werden. Simuliere die Rückgabe aller belegten Ressourcen:
 $A := A + C_i$ (i -te Zeile von C)
 Markiere den Prozess – er ist nicht Teil eines Deadlocks
4. Weiter mit Schritt 1

Beispiel (5 Philosophen)



	A B C D E	A B C D E
$E = (1\ 1\ 1\ 1\ 1)$	$A = (0\ 0\ 0\ 0\ 0)$	
$C = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$	$R = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$	

- Algorithmus bricht direkt ab
- alle Prozesse sind Teil eines Deadlocks

Deadlock-Behebung:

Was tun, wenn ein Deadlock erkannt wurde?

- **Entziehen** einer Ressource?
In den Fällen, die wir betrachten, unmöglich
(ununterbrechbare Ressourcen)
- **Abbruch** eines Prozesses, der am Deadlock beteiligt ist
- **Rücksetzen** eines Prozesses in einen früheren Prozesszustand, zu dem die Ressource noch nicht gehalten wurde
 - erfordert regelmäßiges Sichern der Prozesszustände

Sichere vs. unsichere Zustände

- Ein Zustand heißt **sicher**, wenn es eine Ausführreihenfolge der Prozesse gibt, die auch dann keinen Deadlock verursacht, wenn alle Prozesse sofort ihre maximalen Ressourcenforderungen stellen.
- Ein Zustand heißt **unsicher**, wenn er nicht sicher ist.
- Unsicher bedeutet nicht zwangsläufig Deadlock!

Deadlock Avoidance (Vermeidung)

- **Idee:** BS erfüllt Ressourcenanforderung nur dann, wenn dadurch auf keinen Fall ein Deadlock entstehen kann
- Das funktioniert nur, wenn man die **Maximalforderungen aller Prozesse** kennt
 - Prozesse registrieren **beim Start** für alle denkbaren Ressourcen ihren Maximalbedarf
 - für die Praxis i. d. R. irrelevant
 - nur in wenigen Spezialfällen nützlich

Banker-Algorithmus (1)

- Idee: Liquidität im Kreditgeschäft
 - Kunden haben eine Kreditlinie (maximaler Kreditbetrag)
 - Kunden können ihren Kredit in Teilbeträgen in Anspruch nehmen, bis die Kreditlinie ausgeschöpft ist – dann zahlen sie den kompletten Kreditbetrag zurück
 - Prüfe bei Kreditanforderung, ob diese die Bank in einem **sicheren** Zustand lässt, was die Liquidität angeht – wird der Zustand unsicher, lehnt die Bank die Auszahlung ab

Banker-Algorithmus (2) – Beispiel

Bank: 1200 €,
900 € verliehen, 300 € Cash



	Max.	Aktueller Kredit
Kunde 1	1000 €	500 €
Kunde 2	400 €	200 €
Kunde 3	900 €	200 €

sicher, denn es gibt folgende Auszahlungs-/Rückzahlungsreihenfolge:

	(Bank)
K2: leiht	200 € (100 €)
K2: rückz.	400 € (500 €)
K1: leiht	500 € (0 €)
K1: rückz.	1000 € (1000 €)
K3: leiht	700 € (300 €)
K3: rückz.	900 € (1200 €)

Bank: 1200 €,
1000 € verliehen, 200 € Cash



	Max.	Aktueller Kredit
Kunde 1	1000 €	500 €
Kunde 2	400 €	200 €
Kunde 3	900 €	300 €

unsicher, weil es keine mögliche Auszahlungsreihenfolge gibt, die die Bank bedienen kann:

	(Bank)
K2: leiht	200 € (0 €)
K2: rückz.	400 € (400 €)
K1: leiht	500 € (-100 €)
K3: leiht	600 € (-200 €)
	(letzte zwei unmöglich)

Banker-Algorithmus (4)

- Datenstrukturen wie bei Deadlock-Erkennung:
 - n Prozesse $P_1 \dots P_n$, m Ressourcentypen $R_1 \dots R_m$ mit je E_i Ressourcen-Instanzen ($i=1, \dots, m$)
→ **Ressourcenvektor** $E = (E_1 E_2 \dots E_m)$
 - Ressourcenrestvektor** A (wie viele sind noch frei?)
 - Belegungsmatrix** C
 C_{ij} = Anzahl Ressourcen vom Typ j , die Prozess i belegt
 - Maximalbelegung** Max :
 Max_{ij} = max. Bedarf, den Prozess i an Ressource j hat
 - Maximale zukünftige Anforderungen**: $R = Max - C$,
 R_{ij} = Anzahl Ressourcen vom Typ j , die Prozess i noch maximal anfordern kann

Banker-Algorithmus (3) – Beispiel

Bank: 1200 €,
900 € verliehen, 300 € Cash



	Max.	Aktueller Kredit
Kunde 1	1000 €	500 €
Kunde 2	400 €	200 €
Kunde 3	900 €	200 €

→ Kunde 3 fordert 100 € an

Übergang sicher → unsicher nicht zulassen!

Bank: 1200 €,
1000 € verliehen, 200 € Cash



	Max.	Aktueller Kredit
Kunde 1	1000 €	500 €
Kunde 2	400 €	200 €
Kunde 3	900 €	300 €

↓ Kunde 2 fordert 200 € an und zahlt alles zurück

Bank:	400 € Cash
Kunde 1	1000 € 500 €
Kunde 2	400 € 0 €
Kunde 3	900 € 300 €



Banker-Algorithmus (5)

Anforderung zulassen, falls

- Anforderung bleibt im Limit des Prozesses
- Zustand nach Gewähren der Anforderung ist sicher

Feststellen, ob ein Zustand sicher ist

=

Annehmen, dass alle Prozesse sofort ihre Maximalforderungen stellen, und dies auf Deadlocks überprüfen
(siehe Algorithmus auf Folie G-67)

Banker-Algorithmus (6) – Beispiel

$$C = \begin{pmatrix} 0 & 1 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \rightarrow E = (10 \ 5 \ 7) \quad A = \begin{pmatrix} 3 & 3 & 2 \end{pmatrix} \quad \text{Max} = \begin{pmatrix} 7 & 5 & 3 \\ 3 & 2 & 2 \\ 9 & 0 & 2 \\ 2 & 2 & 2 \\ 4 & 3 & 3 \end{pmatrix} \quad R = \text{Max} - C = \begin{pmatrix} 7 & 4 & 3 \\ 1 & 2 & 2 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix}$$

Anforderung (1 0 2) durch Prozess P2 – ok?

1. $(1 \ 0 \ 2) < (1 \ 2 \ 2)$, also erste Bedingung erfüllt
2. Auszahlung simulieren

$$C' = \begin{pmatrix} 0 & 1 & 0 \\ 3 & 0 & 2 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \rightarrow E = (10 \ 5 \ 7) \quad A' = \begin{pmatrix} 2 & 3 & 0 \end{pmatrix} \quad R' = \begin{pmatrix} 7 & 4 & 3 \\ 0 & 2 & 0 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix} \quad \text{Jetzt Deadlock-Erkennung durchführen}$$

Deadlock-Verhinderung (prevention): Vorbeugendes Verhindern

- mache mindestens eine der vier Deadlock-Bedingungen unerfüllbar
 1. gegenseitiger Ausschluss
 2. Hold and Wait
 3. Ununterbrechbarkeit der Ressourcen
 4. Zyklisches Warten
- dann sind keine Deadlocks mehr möglich (denn die vier Bedingungen sind notwendig)

Deadlock-Vermeidung (avoidance) (9)

① $E = (10 \ 5 \ 7) \quad A' = (2 \ 3 \ 0)$ $C' = \begin{pmatrix} 0 & 1 & 0 \\ 3 & 0 & 2 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \quad R' = \begin{pmatrix} 7 & 4 & 3 \\ 0 & 2 & 0 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix}$	④ $E = (10 \ 5 \ 7) \quad A' = (7 \ 5 \ 3)$ $C' = \begin{pmatrix} 0 & 1 & 0 \\ 3 & 0 & 2 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \quad R' = \begin{pmatrix} 7 & 4 & 3 \\ 0 & 2 & 0 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix}$
② $E = (10 \ 5 \ 7) \quad A' = (5 \ 3 \ 2)$ $C' = \begin{pmatrix} 0 & 1 & 0 \\ 3 & 0 & 2 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \quad R' = \begin{pmatrix} 7 & 4 & 3 \\ 0 & 2 & 0 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix}$	⑤ $E = (10 \ 5 \ 7) \quad A' = (10 \ 5 \ 5)$ $C' = \begin{pmatrix} 0 & 1 & 0 \\ 3 & 0 & 2 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \quad R' = \begin{pmatrix} 7 & 4 & 3 \\ 0 & 2 & 0 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix}$
③ $E = (10 \ 5 \ 7) \quad A' = (7 \ 4 \ 3)$ $C' = \begin{pmatrix} 0 & 1 & 0 \\ 3 & 0 & 2 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \quad R' = \begin{pmatrix} 7 & 4 & 3 \\ 0 & 2 & 0 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix}$	⑥ $E = (10 \ 5 \ 7) \quad A' = (10 \ 5 \ 7)$ $C' = \begin{pmatrix} 0 & 1 & 0 \\ 3 & 0 & 2 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \quad R' = \begin{pmatrix} 7 & 4 & 3 \\ 0 & 2 & 0 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix}$ OK!

1. Gegenseitiger Ausschluss

- Ressourcen nur dann exklusiv Prozessen zuteilen, wenn es keine Alternative dazu gibt
- Beispiel: Statt mehrerer konkurrierender Prozesse, die einen gemeinsamen Drucker verwenden wollen, einen Drucker-Spooler einführen
 - keine Konflikte mehr bei Zugriff auf Drucker (Spooler-Prozess ist der einzige, der direkten Zugriff erhalten kann)
 - aber: Problem evtl. nur verschoben (Größe des Spool-Bereichs bei vielen Druckjobs begrenzt?)

2. Hold and Wait

- Alle Prozesse müssen die benötigten Ressourcen gleich beim Prozessstart anfordern (und blockieren)
- hat verschiedene Nachteile:
 - Ressourcen-Bedarf entsteht oft dynamisch (ist also beim Start des Prozesses nicht bekannt)
 - verschlechtert Parallelität (Prozess hält Ressourcen über einen längeren Zeitraum)
- Datenbanksysteme: **Two Phase Locking**
 - Sperrphase: Alle Ressourcen erwerben (wenn das nicht klappt → alle sofort wieder freigeben)
 - Zugriffsphase (anschließend Freigabe)

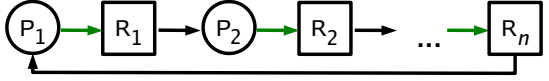
4. Zyklisches Warten (1)

- Ressourcen durchnummerieren
 - $ord: R = \{R_1, \dots, R_n\} \rightarrow \mathbb{N}$, $ord(R_i) \neq ord(R_j)$ für $i \neq j$
- Prozess darf Ressourcen nur in der durch ord vorgegebenen Reihenfolge anfordern
 - Wenn $ord(R) < ord(S)$, dann ist die Sequenz
`lock (S);`
`lock (R);`
 ungültig
- Das macht Deadlocks unmöglich

3. Ununterbrechbarkeit der Ressourcen

- Ressourcen entziehen?
- siehe Deadlock-Behebung (Abbruch / Rücksetzen)

4. Zyklisches Warten (2)

- Annahme: Es gibt einen Zykel 

Für jedes i gilt: $ord(R_i) < ord(R_{i+1})$ und wegen des Zyklus auch $ord(R_n) < ord(R_1)$,
daraus folgt $ord(R_1) < ord(R_1)$: Widerspruch

- Problem: Gibt es eine feste Reihenfolge der Ressourcenbelegung, die für alle Prozesse geeignet ist?
- reduziert Parallelität (Ressourcen zu früh belegt)