

# Betriebssysteme

## Grundlagen

Historie:

1. Generation: Bestand aus Vakuumröhren  
(zuse Z1, Z2) und Relais

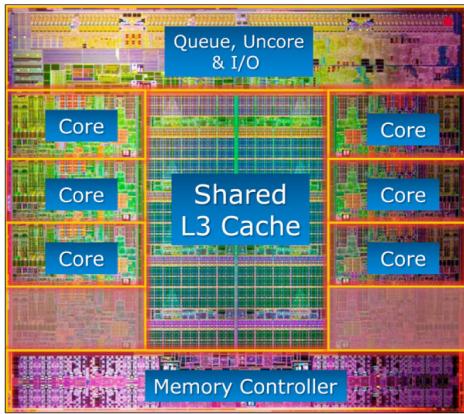
2. Generation: Erfindung der Transistoren ermöglicht Weiterentwicklung  
→ Elektrik  
→ günstiger, schneller, zuverlässiger

3. Generation: Erfindung von Integrated Chips  
Anzahl der Transistoren verdoppelt sich  
→ Anzahl der Transistoren steigt  
→ Größe der Transistoren sinkt

4. Generation: Hochintegrierte Schaltungen

# Cache:

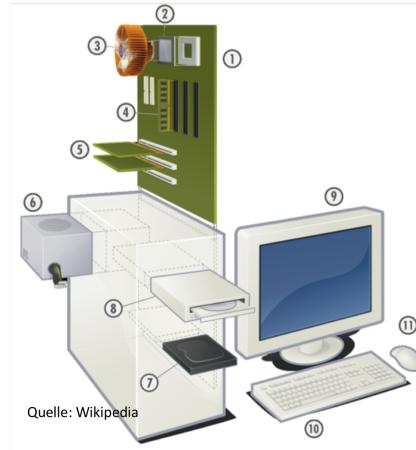
- Hier werden Sachen gespeichert, oft benutzt werden → Zugriff beschleunigen



## Komponenten eines PC

Hochschule Esslingen  
University of Applied Sciences

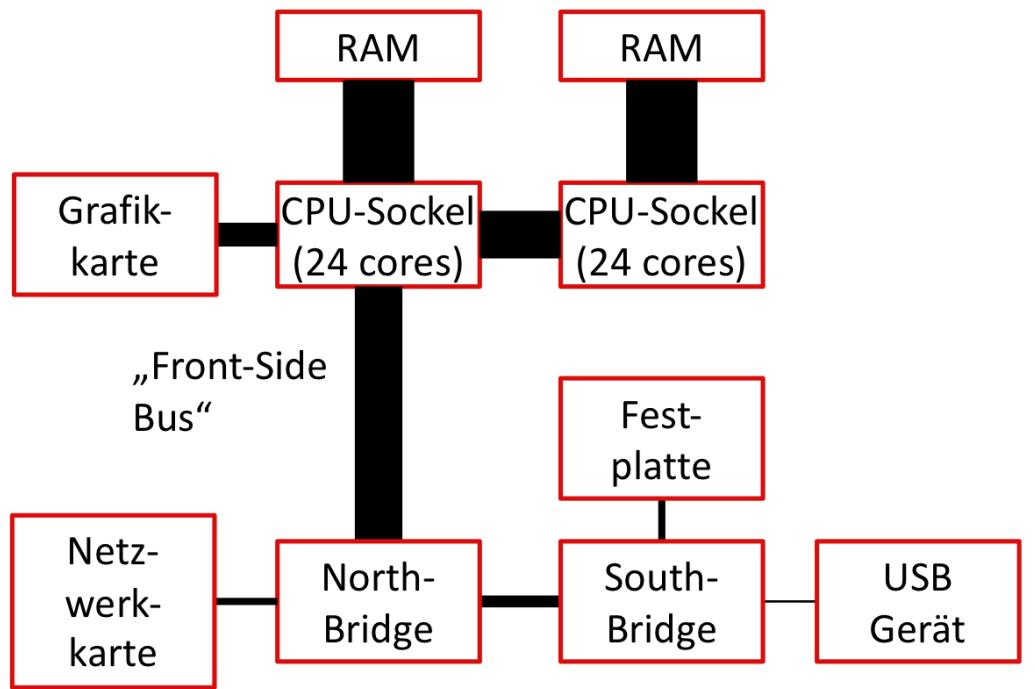
Ein Standard PC ist wie folgt aufgebaut



1. Motherboard / Hauptplatine
2. Prozessor
3. Kühlkörper mit Radiallüfter
4. Hauptspeicher (4 Slots)
5. Erweiterungskarten (3 Slots)
6. Netzteil
7. Festplatte (3½ Zoll)
8. CD-ROM/DVD Laufwerk
- Das ist die Hardware!  
Fehlt die Software!

- Riesiger L3-Cache
- L2 Cache
- 1 L1-Cache pro Core

# Hardware



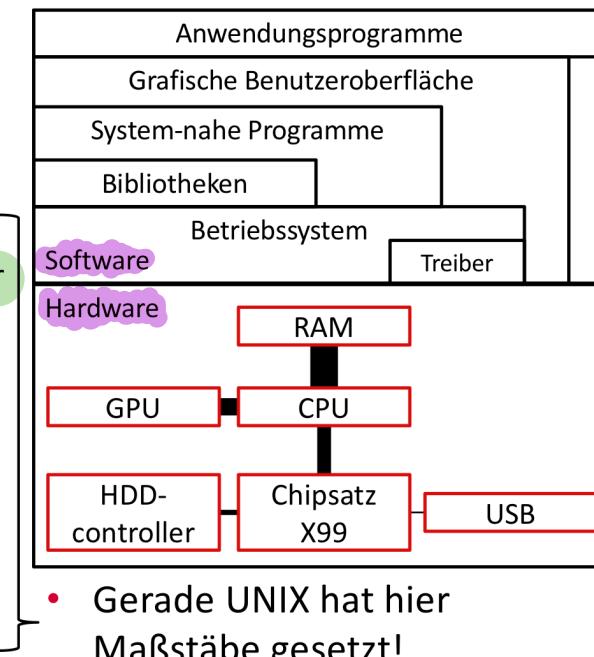
- Wichtig zu merken: Datentransfers vom Prozessor in den Speicher, oder von der Peripherie ist langsam (s. die „dünnen“ Verbindungslien)

# Betriebssystem

Eine Software, die auf effiziente Weise die Komplexität eines Computers vor dem Benutzer versteckt und einer Gruppe von Benutzern und Programmen gemeinsamen und sicheren Zugriff auf Rechen-Speicher- und Kommunikationsmittel zur Verfügung stellt

Was macht ein Betriebssystem ?

- Das Betriebssystem abstrahiert von der HW, es virtualisiert die Ressourcen.
- Es teilt CPU, Speicher, etc. den Programmen zu; und nimmt sie wieder weg!
- Es verwaltet die Dateien
- Es stellt eine einheitliche Programmierschnittstelle zur Verfügung  
(Application Programming Interface, API)



- Gerade UNIX hat hier Maßstäbe gesetzt!

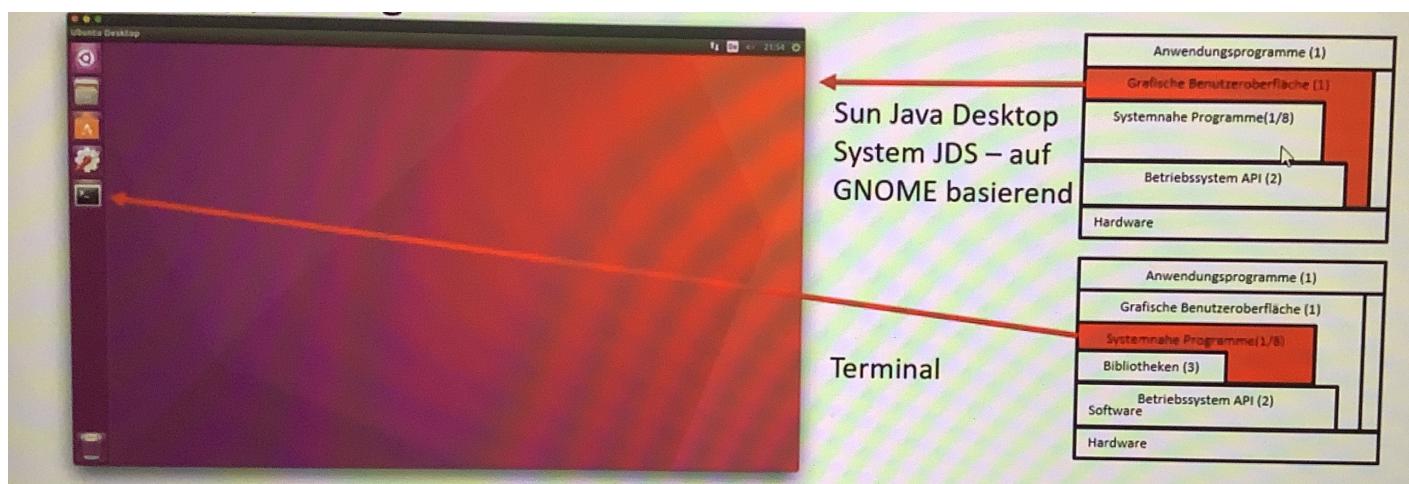
Das Betriebssystem verwaltet Ressourcen...  
Wenn das BS eine Anwendungsoftware wäre,  
was würde dann seine Ressourcen verwalten?  
→ Multitasking wäre nicht möglich  
Zum BS gehören nicht: Bios, Compiler, Editor

**Distribution:** Sie enthält Software-Pakete, strukturiert, besitzt eigene Paketverwaltung und eine grafische Benutzeroberfläche

**Shell:** Ist eine Kommandozeile mit der man Programme startet und ist die niedrigste, systemnahe Software, mit der ein User interagiert. (Schnittstelle zum BS)

## Bash

- Linux ist Open Source (Man kann selbst Änderungen vornehmen)
- Distributionen: Ubuntu, Debian, Yocto, Gentoo



- Mit der Shell startet man Programme
- Die Shell ist die niedrigste systemnahe Software, mit der ein User interagiert

# Funktionen der Bash:

- Systemnahe Programme starten

Syntax: \$ Programm [Param1] [P2|P3] -p pid

optionale Parameter    Wenn { von beiden Parametern benutzt wird    extra Parameter

- i ignore case (Groß-/Kleinschreibung egal)
- h human readable (einfach lesbar)
- p PID Process ID wird als Argument angegeben

## Umgebungsvariablen:

HOME: Heimatverzeichnis

PWD: derzeitiges Verzeichnis

OLDPWD: vorherige aktive Verzeichnis

USER: Benutzername

MEINE\_VAR=1 (Zuweisung)

echo \$\$HOME (gibt Wert von HOME aus)

cd \$\$OLDPWD (Wechsel in vorheriges Verzeichnis) = cd .. /

mkdir test (Verzeichnis "test" wird erstellt)

mkdir -p test/src (Verzeichnisbaum wird angelegt)

ls Inhalte werden angezeigt

CTRL-C killt den laufenden Prozess

CTRL-Z stoppt Prozess (verbraucht keine CPU mehr / schläft)

↳ **bg** Im Hintergrund starten ↳ **fg** im Vordergrund starten  
**Programmname &** = **bg** Programmname  
 → Ausgabe: [1] 8631 (Prozess ID)  
**kill 8631** killt den Prozess  
**cd /Verzeichnisname** Wechselt in gewünschtes Verz.  
**ls -al**

- gibt alle Dateien im long-Format aus (akt. Verz.)
- gibt Rechte, Größe, Besitzer... an

**touch Name** Datei mit Größe 0 anlegen  
**echo "moin" > Name** "moin" wird in Datei "Name" geschrieben  
**cat Name** Zeigt den Inhalt von Dateien an

## Unix Hilfen

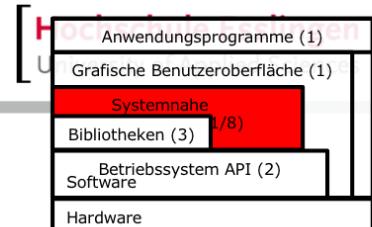
Zu allen Kommandos finden Sie unter Hilfe:

\$ man *kommando* (für manual), bspw.  
 \$ man bash (Hilfeseite zur Bash)

Sucht, formatiert und stellt diese mit less dar (q=beendet)

Die „Man-Pages“ sind eingruppiert in Bereiche 1-8 & n:

\$ man 1 bash (1=System & Benutzerprogramme)  
 \$ man 2 gettimeofday (2=Systemaufrufe &  
 Kernelparameter)  
 \$ man 3 fprintf (3=Bibliotheksaufrufe)  
 \$ man 4 kmem (4=Gerätetreiber & Netz.-  
 protokoll)  
 \$ man 5 ssh\_config (5=Misc. Config.-Dateien)  
 \$ man 7 tcp (7=Gerätetreiber und Protokolle)  
 \$ man 8 netstat (8=Systemadmin-Tools)



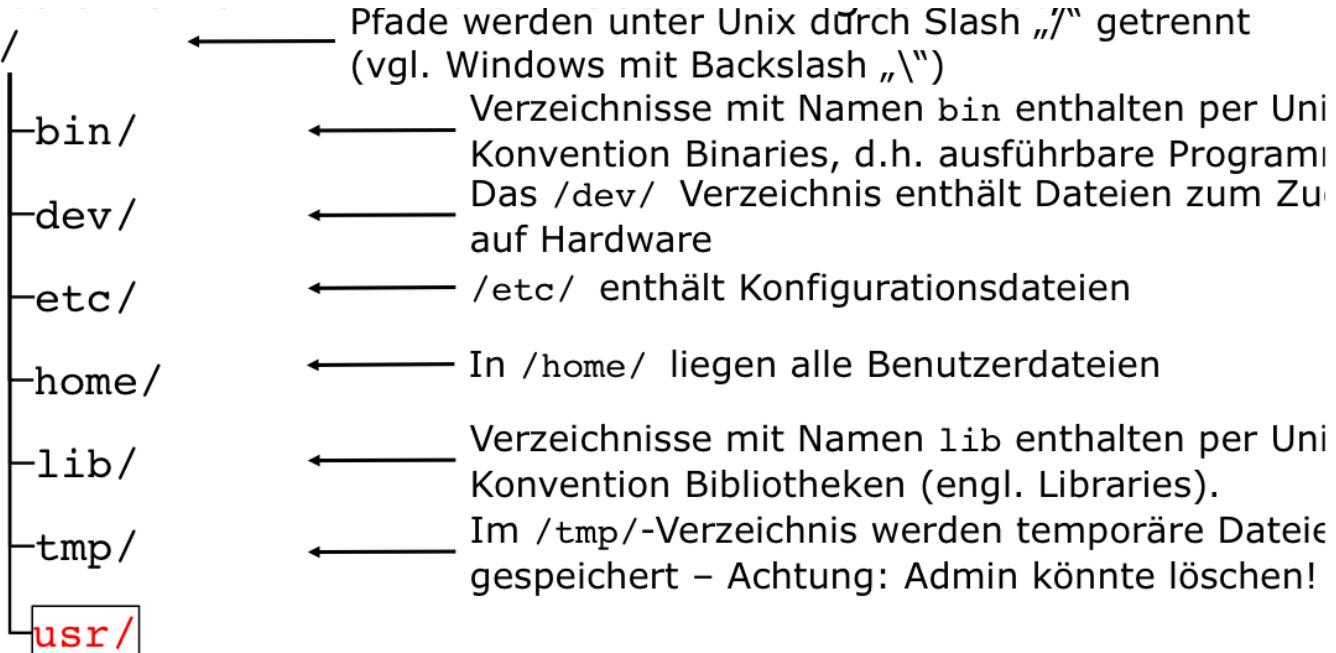
## Bash Skripte

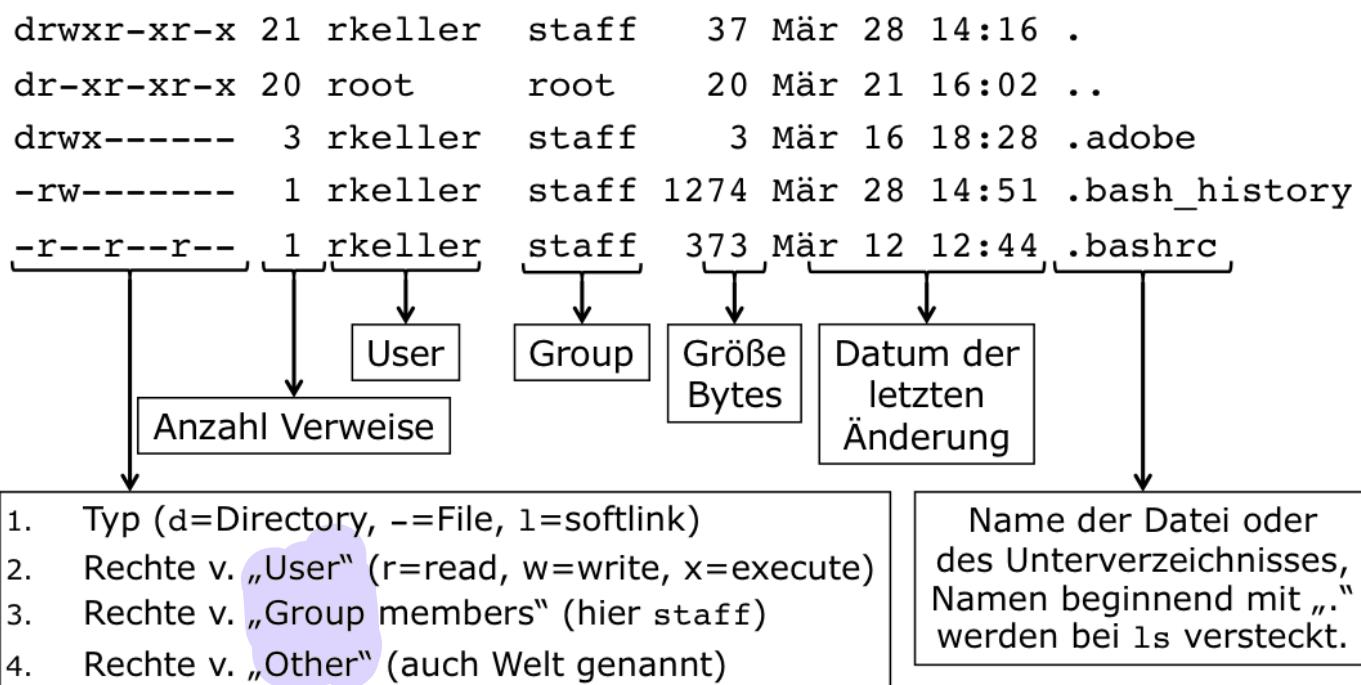
**#!/bin/bash**

`./mein-script` Programme ausführen  
→ "Pipe" → Die Ausgabe von dem, was davor steht,  
als Eingabe für das, was danach steht sein

`netstat -nat` Gibt Internet-Ports aus (numerisch, Alle ,TCP)  
`grep "LISTEN"` sucht nach "LISTEN"  
`cut -f1 -d ":"` Der Teil vor dem ":" wird herausgeschnitten und verwendet  
`cut -f2 -d ":"` Der Teil nach dem ":" wird herausgeschnitten und verwendet  
`sort -n` sortiert numerisch

Dinge in Skripten in Backticks (` `) werden ausgeführt





$$r=4 \quad w=2 \quad x=1$$

Permissions setzen: chmod 600 (Read+Write für Owner)

Permissions anschauen: ls -al ~/.bashrc

Oder chmod u+x Dateiname (Bei Dateien)

u = user      g = group      o = other

chmod gu=rw filename (group+user read+write)

chmod o+x filename (Other werden Ausführungsrechte hinzugefügt)

chmod g-w filename (group werden Schreibrechte entzogen)

Hardlink erstellen:

In testdatei linkdatei

(was man mit der einen Datei macht,  
passiert auch mit der anderen Datei)

# Softlinks erstellen:

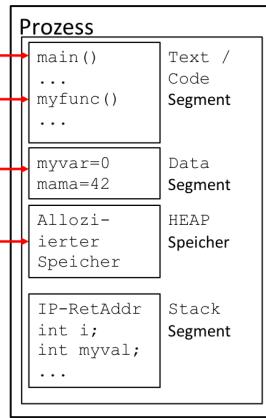
ln -s testdatei linkname (es wird lediglich auf den Dateinamen verwiesen)

mount -t fs-type device mount-point (Ein Dateisystem im Verzeichnisbaum einhängen)  
fsck device (Dateisystem auf Fehler überprüfen)

Ihr Programm app.c:  
static int myvar=0;  
static int mama=42;  
void myfunc(void) {  
 mama=4711;  
}  
int main (int argc, char\* argv[]){  
 myfunc();  
 char \* mem = malloc(1024);  
 printf("mama:%d mem:%p\n", mama, mem);  
 return EXIT\_SUCCESS;  
}

Compiler übersetzt das C Programm:  
gcc -Wall -O2 -o ./app app.c

Das OS startet für ./app einen Prozess, darin wird Speicher angelegt für Text & Data Segment und springt „in main“...

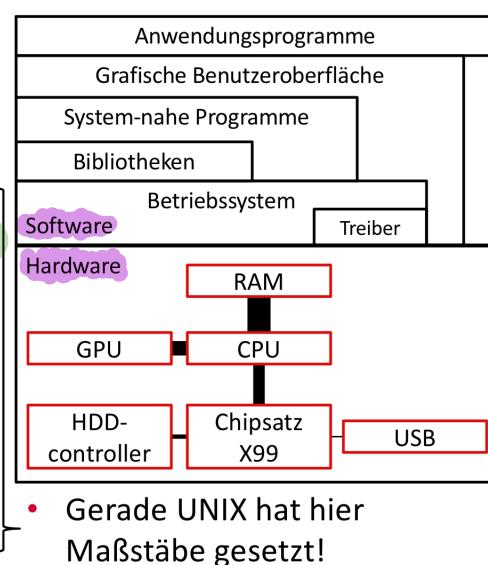


Jedes Programm besteht aus mind. 1 Prozess, dieser aus mind. 1 Thread.

Für jeden Prozess wird CPU, Speicher(-bereich), Tastatur, andere Ressourcen zugeteilt.

Was macht ein Betriebssystem?

- Das Betriebssystem abstrahiert von der HW, es virtualisiert die Ressourcen.
- Es teilt CPU, Speicher, etc. den Programmen zu; und nimmt sie wieder weg!
- Es verwaltet die Dateien
- Es stellt eine einheitliche Programmierschnittstelle zur Verfügung (Application Programming Interface, API)

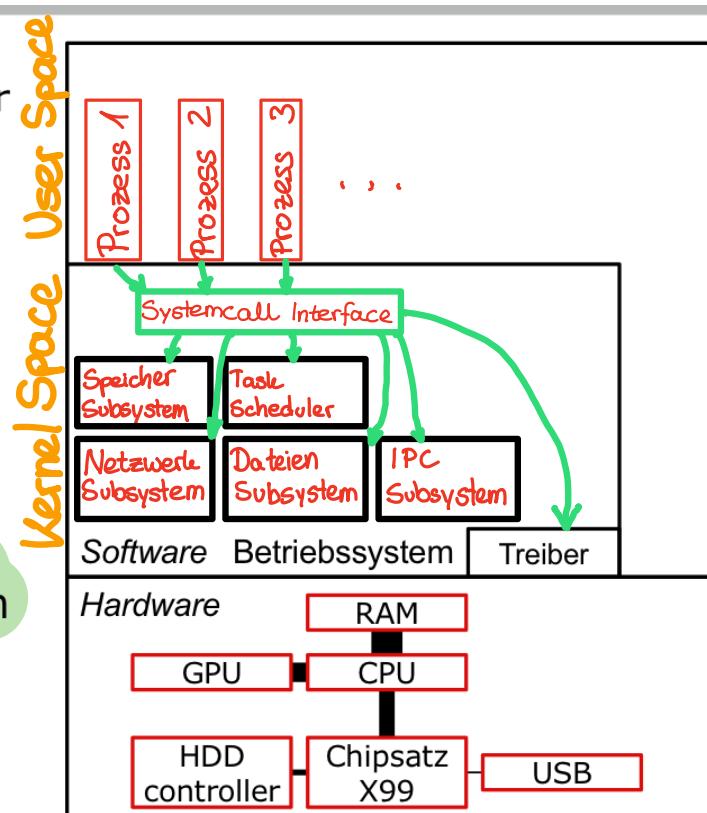


Ressourcen können entweder Hardware- oder Softwareressourcen, entziehbar oder nicht entziehbar und exklusiv oder gemeinsam nutzbar sein.

(z.B. CPU, Hauptspeicher, Drucker)

Das BS regelt den konfliktfreien Zugriff auf die Ressourcen (mithilfe verschiedener Algorithmen in verschiedenen Subsystemen des Kernels)

- Für das Betriebssystem ist jede Anwendung ein einzelner **Prozess**.
- Diese Prozesse laufen unabhängig voneinander
- Das Betriebssystem läuft im **Kernel-Space**
- Die Prozesse laufen im **User-Space**
- Nur **das** erlaubt perfekten Schutz zwischen Prozessen
- Dieser Schutz funktioniert **nur** mittels CPU Unterstützung



Jeder Prozess kann über Systemcall die Funktionen des BS aufrufen

**open** Öffnen einer Datei

**read** Lesen aus einem Dateideskriptor

**clone** Eine Kopie des Prozesses wird erstellt

**gettimeofday** Tageszeit

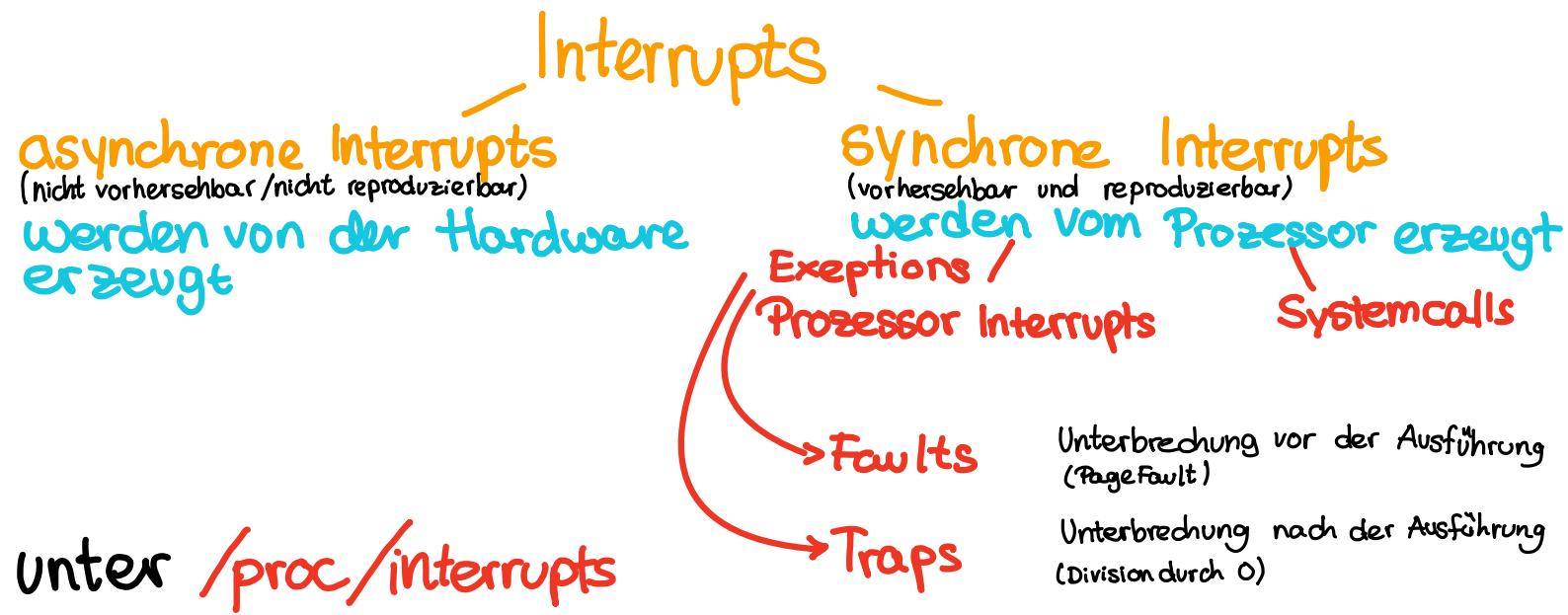
insgesamt ca. 350-400

Prozess ID herausfinden:

PID = getpid(); oder

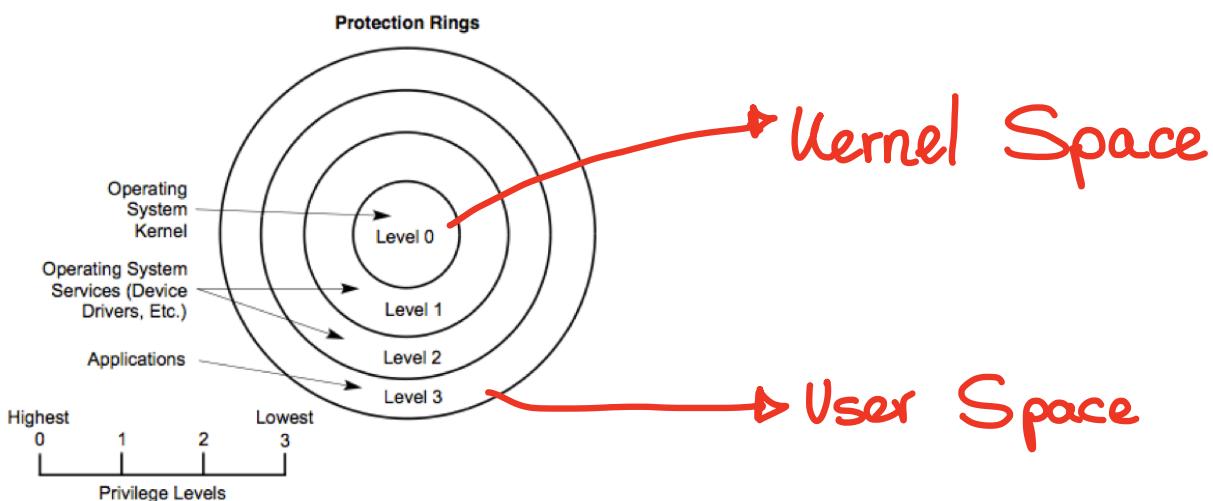
PID = syscall(SYS\_getpid);

- Für den Wechsel vom User- in den Kernel-Space verwendet Linux **Software-Interrupts**
- Dieser Wechsel heißt **Trap**
- Interrupts unterbrechen den Prozessor



1. Ein Gerät meldet Interrupt über Interrupt-Leitung
2. Der Mikroprozessor wird unterbrochen
3. Die ausgeführte Anwendung merkt davon nichts (Außer Verzögerung)
4. Eine Interrupt Behandlungsroutine liest Daten
5. BS springt in die unterbrochene Anwendung zurück
6. Periodisch muss der BS-interne Puffer kopiert werden
7. Wartet die Anwendung auf Daten (bspw. von der Festplatte) können diese gelesen werden

- Die CPU kann Interrupts "maskieren" (deaktivieren) um nicht bei der Abarbeitung des derzeitigen Interrupts gestört zu werden  
**(Eine Rekursion wäre fatal)**
- Bestimmte HW-Interrupts sind nicht maskierbar (z.B. Memory Errors)
- Interrupts können eine Priorität besitzen



```

Operation
IF (CS.L ≠ 1) or (IA32_EFER.LMA ≠ 1) or (IA32_EFER.SCE ≠ 1)
(* Not in 64-Bit Mode or SYSCALL/SYSRET not enabled in IA32_EFER *)
THEN #UD;
Fl;

```

```

RCX ← RIP; (* Will contain address of next instruction *)
RIP ← IA32_LSTAR;
R11 ← RFLAGS;
RFLAGS ← RFLAGS AND NOT(IA32_FMASK);

```

```

CS.Selector ← IA32_STAR[47:32] AND FFFCH (* Operating system provides CS; RPL forced to 0 *)
(* Set rest of CS to a fixed value *)
CS.Base ← 0; (* Flat segment *)
CS.Limit ← FFFFH; (* With 4-KByte granularity, implies a 4-GByte limit *)
CS.Type ← 11; (* Execute/read code, accessed *)
CS.S ← 1;
CS.DPL ← 0;
CS.P ← 1;
CS.L ← 1;
CS.D ← 0;
CS.C ← 1;
CPL ← 0;

```

```

SS.Selector ← IA32_STAR[47:32] + 8; (* SS just above CS *)
(* Set rest of SS to a fixed value *)
SS.Base ← 0; (* Flat segment *)
SS.Limit ← FFFFH; (* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type ← 3; (* Read/write data, accessed *)
SS.S ← 1;
SS.DPL ← 0;
SS.P ← 1;
SS.B ← 1;
SS.C ← 1;

```

} Fallunterscheidung, ob eine Instruktion in diesem Prozessor-Modus nicht unterstützt wird

} Das Code Segment Register wird initialisiert: vom OS bereitzustellen

} Current Privilege LVL. auf 0

} Das Stack Segment Register wird initialisiert

```

int main(int argc, char * argv[]) {
    unsigned long long int tsc, pid;
    double t_syscall, t_libc, t_int0x80;
    tsc = getrdtsc ();
    for (int i = 0; i < NUM_TIMES; i++)
        pid = getpid();
    t_libc = (double)(getrdtsc() - tsc) / NUM_TIMES;

    tsc = getrdtsc ();
    for (int i = 0; i < NUM_TIMES; i++)
        pid = syscall(SYS_getpid);
    t_syscall = (double)(getrdtsc() - tsc) / NUM_TIMES;

    tsc = getrdtsc ();
    for (int i = 0; i < NUM_TIMES; i++)
        pid = my_getpid_int0x80();
    t_int0x80 = (double)(getrdtsc() - tsc) / NUM_TIMES;

    printf ("Timing: libc:%f syscall:%f int0x80:%f\n",
           t_libc, t_syscall, t_int0x80);
    // Mess with the compiler's optimisations
    return (argc * pid);
}

```

Messen von getpid() aus der Bibliothek libc

Messen vom Syscall Aufruf

Messen mit eigener Routine

sicherstellen, dass der Compiler nicht wegoptimiert

Anfänge von Linux: Linus Torvalds wollte mithilfe eines Bootloaders die Schutzmechanismen eines Prozessors testen.  
Braucht dafür: Compiler, Host Betriebssystem

Gründe für Erfolg von Linux:

- Vertraute und Konkurrenten werden akzeptiert
- Andere sollen auch an der Weiterentwicklung teilhaben können
- Hohen Wert auf Sicherheit und Individualität gelegt

1991: ca. 10k Lines of Code

2019: ca. 25mio LOC

Steigerungen durch: \*neue Architekturen (im arch-Verzeichnis)  
\*Treiber für neue Hardware (in drivers Verzeichnis)

- Die Struktur des Kernels ist von je her gleich:

Documentation/	(Alles über Linux)
arch/	(Architektur-abhängige Dateien, z.B. x86)
drivers/	(Alle Low-Level Treiber für Hardware!!!)
fs/	(Schnittstelle und Code für Filesysteme!)
include/	(Alle C-Header Dateien)
init/	(Einsprung nach dem Booten: start_kernel)
kernel/	(Architektur-unabhängige Schnittstellen)
mm/	(Memory Management Schnittstelle)
net/	(Netzwerk Schnittstelle - ohne Treiber)

Was braucht man, um das BS / den Kernel auszutauschen?:

- Einen C Compiler (gcc)
- GNU Make
- Diverse Unix-Tools (Shell, mkdir, wc)

## Linux Kernel übersetzen

[ Hochschule Esslingen  
University of Applied Sciences ]

- Eine erste Variante den Kernel zu übersetzen:

make help<sup>4</sup> (Zeigt Hilfe zu Make-Parametern an!)

make config<sup>4</sup> (Abfrage **aller** Parameter, erzeugt .config)

make oldconfig<sup>4</sup> (basierend auf einer .config, neue abfragen)

make listnewconfig<sup>4</sup> (zeigt neue Parameter an)

make all<sup>4</sup> (Baut Kernel-Datei/Image und alle Module)

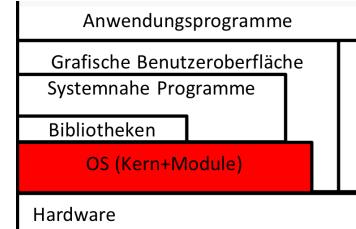
make install<sup>4</sup> (Installiert Kernel-Image unter /boot)

make modules\_install<sup>4</sup> (Installiert Kernel-Module unter /lib)

make firmware\_install<sup>4</sup> (Das selbe evtl. mit neuerer Firmware)

# Kernel Module

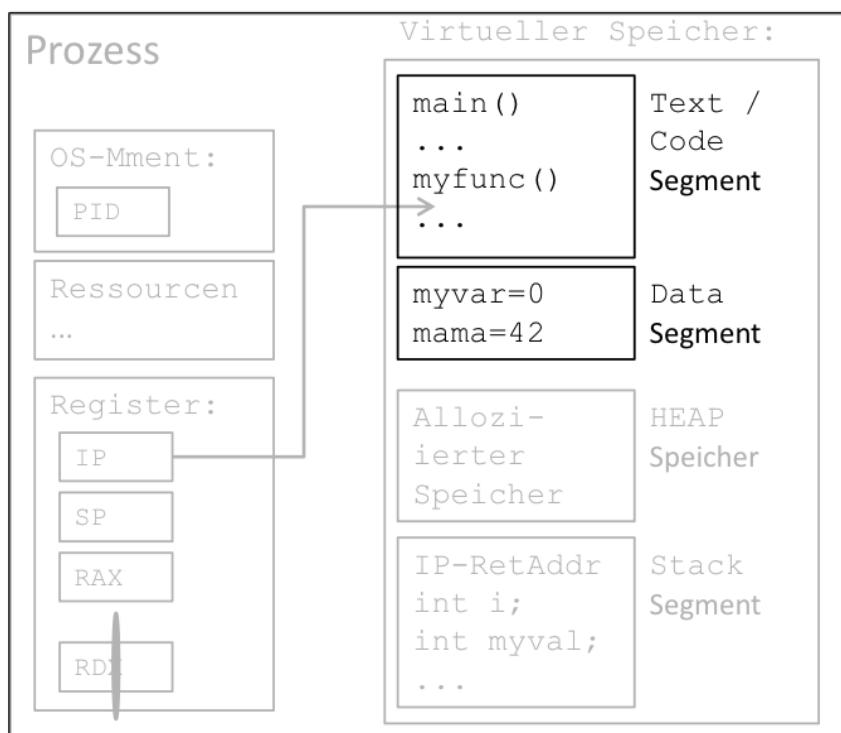
- erlauben Zugriff auf Hardware (Grafikkarten, Netzwerkkarten)
- Datenträger
- Funktionalitäten (Verschlüsselungsmethoden) → direkt im Kern
- befinden sich in /lib/modules/"Versionsnummer"



**find -name \*.ko | wc**

findet Dateien mit Endung .ko und liest Wörter aus

Wiederholung:



liegen in der Datei bereits als .text und .data Sections

Bei Modul Dateien genauso.  
Zusätzliche Section, die Infos über das Modul enthält  
→ Anzeigen: objdump



- C- Programme werden in Unix als Prozesse gestartet

## Kernel Module:

- besitzen keine main() Funktion
- werden nicht sequenziell ausgeführt
- können während der Ausführung unterbrochen werden
- haben höhere Ausführungsriorität und mehr Rechte bestimmte Instruktionen auszuführen
- Dürfen **keine** Fließkommaoperationen ausführen  
**(Aus Performance- und Speichergründen)**

lsmod	# Liste der derzeit geladenen Module	OS (Kern+Module)
depmod	# Dependencies (Abhängigkeiten) erkennen	Hardware
modprobe	# Module & Dependencies aus /lib/modules laden	
insmod	# Ein bestimmtes Modul (auch im CWD!) laden	
rmmod	# Ein Modul wieder entfernen	
modinfo	# Modulinformationen wie z.B. Funktion, Parameter (allesamt aus Section .modinfo!) anzeigen.	

---

Informationen zu einem Modul gibt modinfo:

```
modinfo rfcomm # Zeige alle Module-infos zu rfcomm
filename: /lib/modules/3.5.0-18-
generic/kernel/net/bluetooth/rfcomm/rfcomm.ko wo liegt das Modul?
alias: btproto-3 Wie heißt es sonst
```

```
license: GPL Lizenz
version: 1.11 Version
description: Bluetooth RFCOMM ver 1.11 Kurzbeschreibung
author: Marcel Holtmann <marcel@holtmann.org>
srcversion: CC35386D02177A222DBB63D eindeutige Versionsnummer
depends: bluetooth Abhängigkeiten
intree: Y Ist es Teil des Linux-Baums?
vermagic: 3.5.0-18-generic SMP mod_unload modversions 686
parm: disable_cfc:Disable credit based flow cntrl (bool)m
```

Mit **depmod** werden Abhängigkeiten zwischen Modulen ermittelt und in der Datenbank abgespeichert

**depmod** muss also immer ausgeführt werden, wenn ein neuer Kernel installiert wird oder Module mit Abhängigkeiten geändert werden

Das macht **modules\_install** automatisch

**modprobe**:

- kann Module mit allen Abhängigkeiten laden
- kann Aliase umwandeln
- kann Abhängigkeiten anzeigen lassen
- kann Module wieder entladen
- Ist ein systemnahe Programm

- Weniger komfortabel, dafür für unsere Anforderungen:

```
insmod modul_name # Einfügen von modul_name
```

- Will man ein Modul einfügen, welches im derzeitigen Verzeichnis (engl. CWD) liegt, einfach mit:

```
insmod ./modul.ko # Einfügen aus diesem Verzeichnis
```

- Um Module wieder aus dem Kernel zu entfernen:

```
rmmod modul_name # Entlädt modul_name
```

- Das systemnahe Programm strace zeigt alle System-Calls (mit Parametern und Rückgabe) an:

## MODUL:

```
#include <linux/kernel.h>
#include <linux/module.h>
MODULE_LICENSE("GPL"); Lizenz
--init int init_module(void){
    printk(KERN_DEBUG "Hello\n"); Hello wird ausgegeben
    return 0;
}
--exit void cleanup_module(void) Module entladen
{} ohne Ausgabe
```

## MAKEFILE:

```
obj-m= linux_module.o
KVERSION= $(shell uname -r)
```

## modules all:

```
make -C /lib/modules/$(KVERSION)/build M=$(PWD) modules
```

## clean:

```
make -C /lib/modules/$(KVERSION)/build M=$(PWD) clean
```

Wenn das Modul gebaut wird, wird die Endung .ko

→ Modul laden: sudo insmod modname

→ Ausgabe: dmesg

→ Entfernen: sudo rmmod modname

Modul suchen (mit "linux"): lsmod | grep linux

- Das einfachste Modul ist:

```
#include <linux/kernel.h>
#include <linux/module.h>
MODULE_LICENSE("GPL");
__init int init_module(void) {
    printk(KERN_DEBUG "Hello\n");
    return 0;
}
__exit void cleanup_module(void)
{
// Eventuell auch diese Macros:
module_init(my_init_module);
module_exit(my_cleanup_module);
```

} Header mit Definitionen zum Kernel und für Module

→ Lizenz festlegen

} wird beim Laden des Moduls aufgerufen

\_\_init zeigt Lader, dass der Speicher hinterher entfernt werden kann

\_\_exit wird beim Entladen aufgerufen

obj-m = linux\_module.o Name der Objektdatei des Moduls

KVERSION = \$(shell uname -r) Kernelversion

modules all: → baut alle Module im derzeitigen Verzeichnis

<TAB> make -C /lib/modules/\$ (KVERSION) /build M=\$ (PWD) modules  
clean:

<TAB> make -C /lib/modules/\$ (KVERSION) /build M=\$ (PWD) clean

```
hpcraink@ubuntu:~$ lsmod | grep linux_module
linux_module           12366  0
```

Warum so groß? → Es werden mehrere Memory-Pages für die Segmente verwendet

· Code · Data · Stack

- Dem `printf()` kann man den Log-Level mitgeben:
- |                           |     |                                       |
|---------------------------|-----|---------------------------------------|
| <code>KERN_DEBUG</code>   | "7" | wird typischerweise ausgefiltert      |
| <code>KERN_INFO</code>    | "6" | Einfache Information                  |
| <code>KERN_NOTICE</code>  | "5" | Nachricht an Benutzer                 |
| <code>KERN_WARNING</code> | "4" | Warnung                               |
| <code>KERN_ERR</code>     | "3" | Fehler, sollte nicht ignoriert werden |
| <code>KERN_CRIT</code>    | "2" | Kritischer Zustand!                   |
| <code>KERN_ALERT</code>   | "1" | Problem, Benachrichtigung d. Users!   |
| <code>KERN_EMERG</code>   | ""  | Notfall, System wohl tot...           |

- Diese Strings werden dem Text vorn ange stellt (**kein Komma**):  
`printf(KERN_DEBUG "Just read Byte:%d\n", byte);`
- Das Programm `sysctl` mit `kernel.printk` kann das filtern!

- `MODULE_LICENSE`:

Gibt die Source-Code Lizenz für dieses Module an. Sollte kein Open Source unter GPL-Code gesetzt sein, wird der Kernel nach dem Laden auf "Tainted" gesetzt:

```
[ 4605.592821] linux_module_2: module license 'unspecified' taints kernel.
[ 4605.592824] Disabling lock debugging due to kernel taint
```

Ein Kernel-Oops (Linux-Äquivalent zu einem Blue-Screen of Death) würde dieses TAINT beinhalten...

- `MODULE_AUTHOR`:

Der / Die Autoren des Modules

- `MODULE_DESCRIPTION`:

Die Kurzbeschreibung des Modules

- `MODULE_VERSION`:

Die Versionsnummer als String

```

#include <linux/kernel.h>
#include <linux/module.h>
MODULE_LICENSE("GPL");
MODULE_INFO(author, "Tobias Heer");
MODULE_INFO(intree, "N");
MODULE_DESCRIPTION("This module is just for demonstration");

static char *var="My linux module!"; Variable für String
//char pointer
//S_IRUGO: READ for USER GROUP OTHER
module_param(var, charp, S_IRUGO); Char Pointer (String)
MODULE_PARM_DESC(var, "The variable to pass"); → soll in var
Bedeutung → var übergeben geschrieben werden
_init int my_init_module(void) {
    printk(KERN_DEBUG "Hello %s\n", var);
    return 0;
}
_exit void my_cleanup_module(void)

// Wir benoetigen nun diese Macros:
module_init(my_init_module);
module_exit(my_cleanup_module);

~"linux_module3.c" 26L, 600C

```

Es soll ein String mitübergeben werden

S\_IRUGO → Zugriffsrechte (readonly)  
User Group Other

Modul mit Variable laden:  
sudo insmod modname var = "variable"

## Scheduler

Virtualisierung des Prozessors:

Single-Tasking: Nur ein Prozess kann gestartet werden

Multi-Tasking: Mehrere Prozesse können gleichzeitig ausgeführt werden

- kooperatives MT: Prozesse können CPU abgeben, müssen aber nicht

- Preemptives MT: BS teilt CPU zu (Time-Slicing)

## Virtualisierung des Hauptspeichers:

- Speicher wird (mit HW-Unterstützung) unterteilt (engl. Page)
- Diese Pages können auf der Festplatte ausgelagert werden
- Wenn die Anwendung Speicher anfordert, kann das BS Speicher einer anderen Anwendung auslagern

## Threads:

Ein Thread ist einer von mehreren Ausführungspfaden innerhalb eines Prozesses

Die Threads eines Prozesses teilen sich:

- Offene File-Deskriptoren (FDs), Dateien, TCP-Sockets
- Speicher (alles was alloziert wurde)
- Signal-Handler

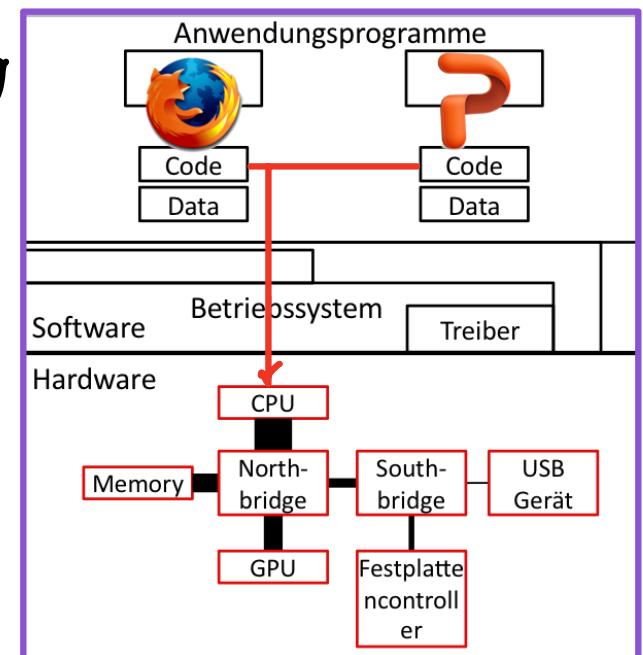
Jeder Thread eines Prozesses hat seine(n) eigene(n):

- Registersatz
- Stack
- Thread-ID, Priority
- Thread-lokale Fehlervariable errno (Fehler lokalisieren)

# Ein preemptives Multi-Tasking BS virtualisiert die CPU

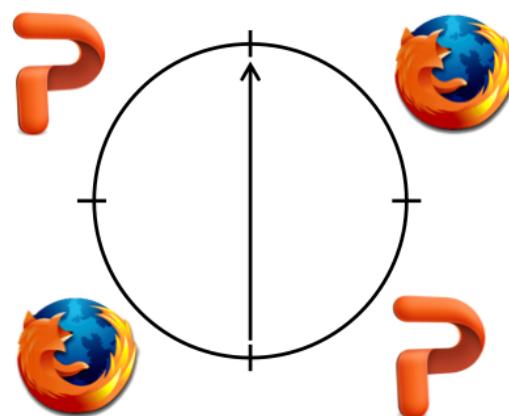
→ Jedes Programm denkt, es hat die CPU für sich allein

Aber das BS kann die CPU jederzeit wegnehmen

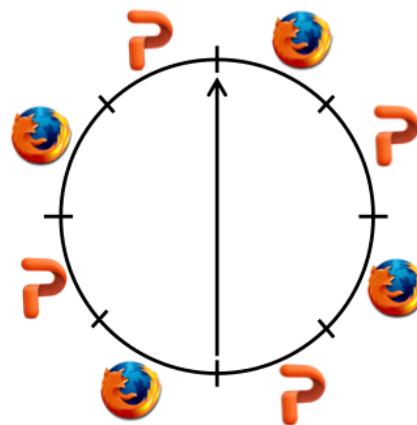


- Dazu braucht es die Hilfe der Hardware (CPU und Timer)
- Time Slicing & Time Scheduling ; Scheduler (Algorithmus)

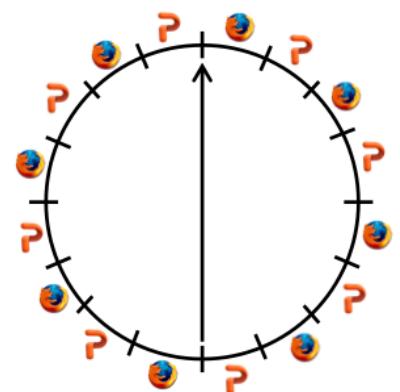
4 Mal pro 5 Sekunden



8 Mal pro 5 Sekunden



16 Mal pro 5 Sekunden



Die Summe der CPU-Zeit ist über den Zeitraum fast gleich verteilt (Die Schlüsse benötigen auch Zeit (Prozesswechsel))  
→ Das Zuordnen: Scheduling  
→ Der Algorithmus: Scheduler

**100 Hz** Desktop-System → guter Kompromiss für viele Anwendungen

**1000 Hz** Server Systeme mit vielen Tausend Prozessen  
→ Auch spezielle Systeme zur Verarbeitung von Audiodateien

**Variabel** Laptop / Systeme zur numerischen Datenverarbeitung  
→ Wenn wenige / nur eine Anwendung läuft

Das BS unterbricht (pre-emptiv) alle laufenden Anwendungen (**Tasks**) und teilt die CPU zu

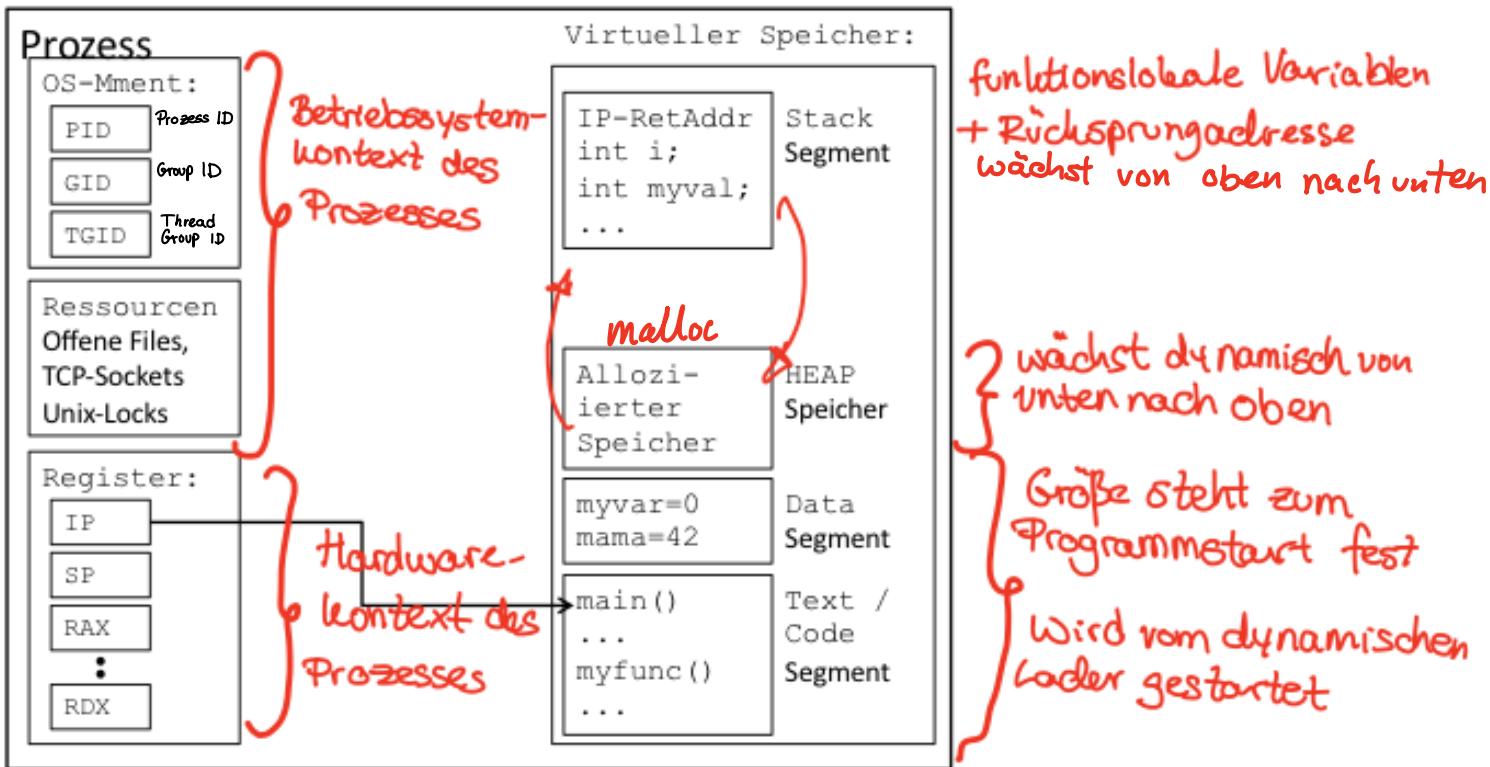
→ Damit das OS entscheiden kann, welcher Prozess auszuführen ist, werden Parameter entwickelt:

- \* Zu diesem Zeitpunkt ein rechenintensiver Prozess?
- \* Zu diesem Zeitpunkt ein I/O-intensiver Prozess?
- \* Welche Priorität hat ein Prozess?

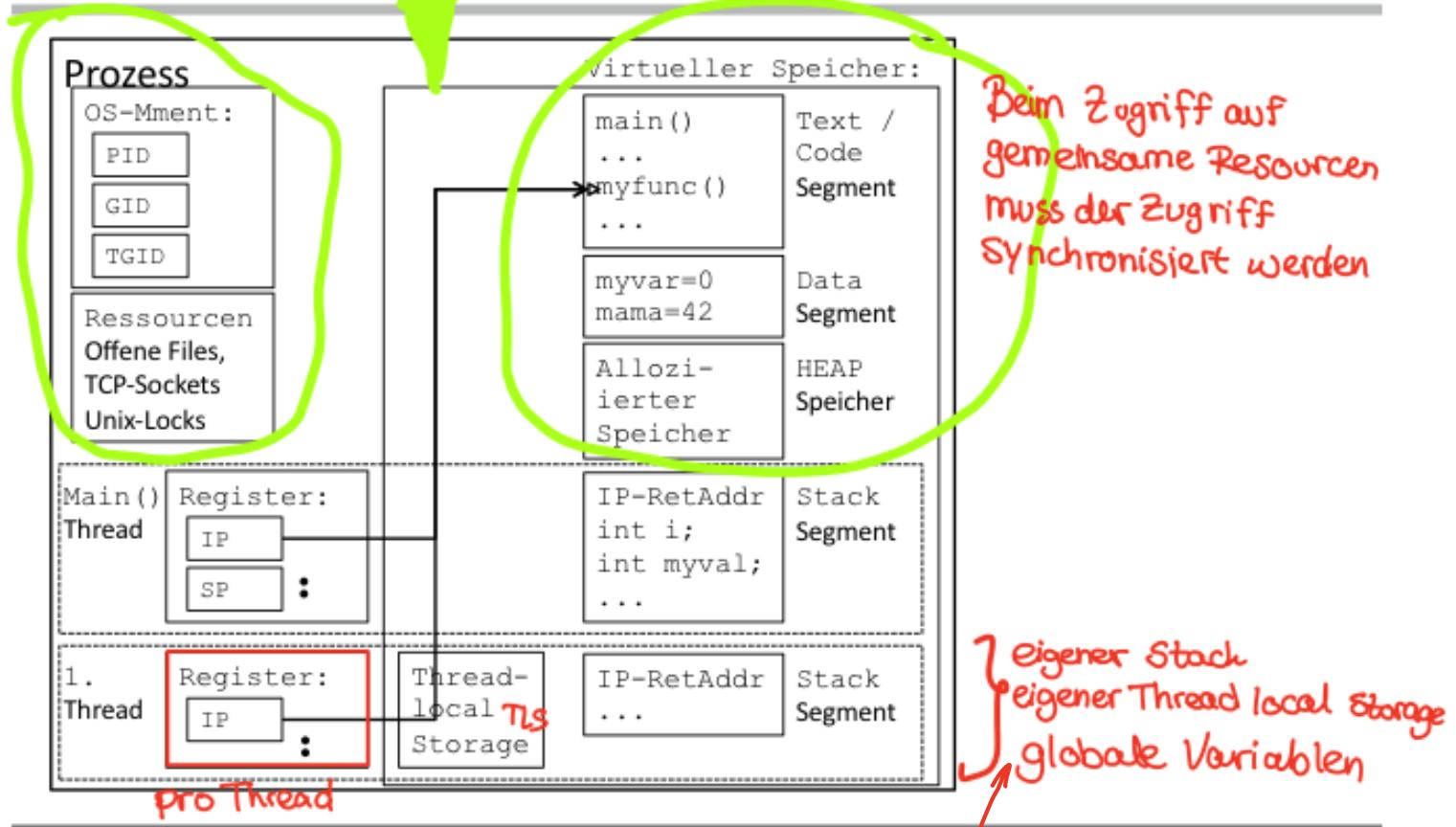
**Kriterien:** Effizienz, Fairness, Balance

- First-Come-First-Serve (FCFS), First-In-First-Out (FIFO):
  - » Prozesse werden in der Reihenfolge ihrer Ankunft **ohne Unterbrechung** bis zum Ende bzw. Blockierung bedient
  - + das einfachste Verfahren
  - + Implementierung über einfache Warteschlange
  - + im Allgemeinen gute Auslastung der Ressource (z.B. CPU)
  - u. U. große Wartezeiten und damit schlechte Antwortzeiten
  - kann zu langen Warteschlangen mit I/O-intensiven Prozessen führen, wenn ein rechenintensiver Prozess mit wenig I/O-Operationen immer wieder die CPU blockiert
- First-In-First-Out (FIFO) mit Prioritätenwarteschlange:
  - » Prozesse werden in der Reihenfolge ihrer Ankunft **mit Unterbrechung** bis zum Ende bzw. Blockierung bedient, aber rechenintensive Prozesse erhalten eine geringe Priorität und I/O-intensive eine hohe Priorität. Prozesse derselben Prioritätsklasse werden in der Reihenfolge ihrer Ankunft ohne Unterbrechung bis zu ihrem Ende bzw. ihrer Blockierung bedient. Prozesse können durch Prozesse höherer Priorität unterbrochen werden. Der Scheduler kann die Prioritäten der Prozesse ändern.
  - + das einfachste Verfahren
  - + Implementierung über einfache Warteschlange
  - + im Allgemeinen gute Auslastung der CPU
  - + Je nach Priorisierung faire Verteilung und
  - + Gutes Antwortverhalten (für I/O-intensive Prozesse)
- Shortest Job First:
  - » Der Prozess mit der kürzesten Rechenzeit wird als nächster **ohne Unterbrechung** bedient; Kenntnis der Rechenzeit ist erforderlich; Prozesse mit gleicher Rechenzeit werden nach FCFS bedient
  - + Verfahren ist optimal bezogen auf die mittlere Wartezeit für eine vorgegebene Menge von Prozessen
  - Voraussichtliche Rechenzeit muss bekannt sein
  - » nicht geeignet für kurzfristiges *Scheduling* innerhalb des OS, da die Rechenzeiten i.A. nicht bekannt sind,
  - » Wird eingesetzt für langfristiges *Scheduling* im Batchbetrieb (Das Zeitlimit wird vom Benutzer angegeben / geschätzt...).

- Round-Robin Scheduling:
  - » Der Prozess darf den Prozessor für eine **Time-Slice** benutzen; falls er in dieser Zeit nicht fertig wird, wird er **unterbrochen** und am Ende der Warteschlange eingereiht; es handelt sich hier um ein zyklisches Verfahren ohne Prioritäten.
  - + gleichmäßige Aufteilung der CPU-Zeit auf die Prozesse
  - Größe der Zeitscheibe ist stark Situationsabhängig:
    - zu kleine Zeitscheibe
      - ⇒ hoher Verwaltungsaufwand durch Prozesswechsel
      - ⇒ Effizienz niedrig, da Cache immer „cold“
    - zu große Zeitscheibe
      - ⇒ nähert sich dem FCFS-Verfahren, da mit zunehmender Zeitscheibe blockierende Systemaufrufe wahrscheinlicher werden
      - ⇒ mittlere Wartezeiten und Antwortzeiten werden größer
- Fair-Share Scheduling:
  - » Bindung eines Anteils einer CPU an den Process-Owner bzw. User, anstelle des Prozesses selbst. Einer User, der mehrere Prozesse laufen lässt, erhält für die Summe aller seiner Prozesse den gleichen Anteil an CPU Ressourcen wie ein anderer User mit nur einem Prozess.
  - + Benutzer die das System vollmachen werden bestraft.
  - Verwaltung der Information ist ziemlich aufwändig.



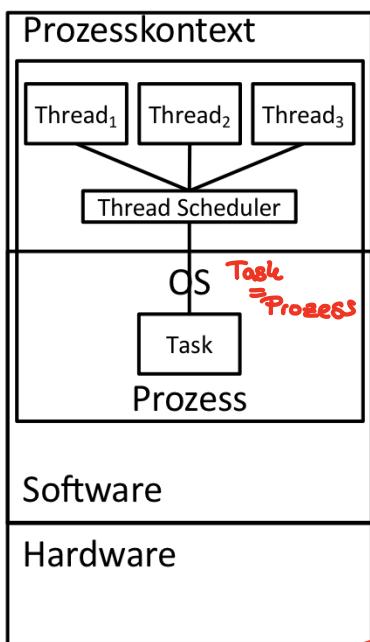
## Struktureller Aufbau nun mit Threads



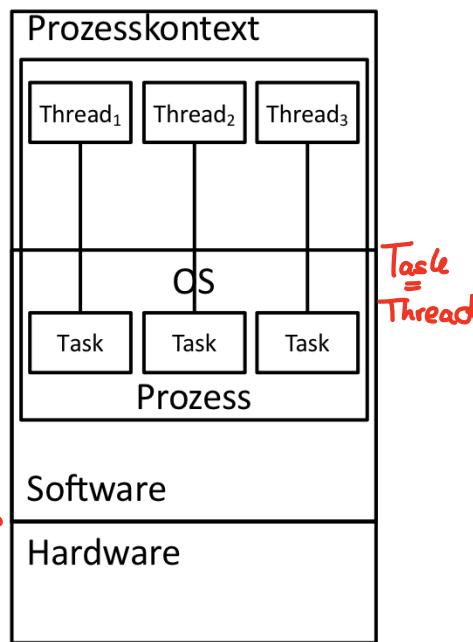
# Linux Prozesskontext:

- » Als Benutzer-Kontext werden die Daten des Prozesses und all seiner Threads im zugewiesenen Adressraum bezeichnet.
- » Als System-Kontext bezeichnet man Information, die das Betriebssystem aus seiner Sicht, über den Prozess speichert, beispielsweise die Prozessnummer, die vom Prozess geöffneten Dateien, Information über Eltern- oder Kindprozesse, Prioritäten, in Anspruch genommene Zeit usw.
- » Als Hardware-Kontext betrachtet man die Inhalte der CPU-Register zum Zeitpunkt der Ausführung des Prozesses, sowie ergänzend die Seitentabelle. Diese Information muss gespeichert werden, wenn im Rahmen eines Multitaskings ein Prozess durch einen anderen Prozess unterbrochen wird.

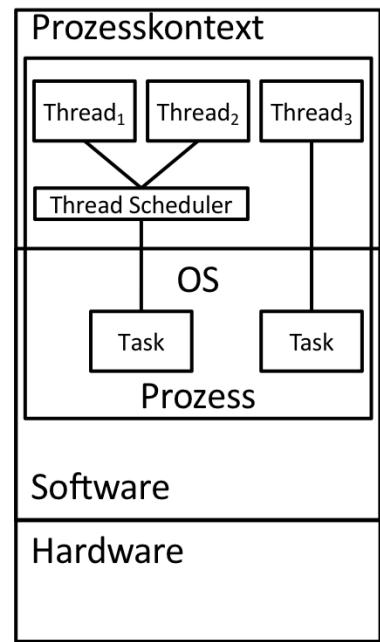
1:n Threads



1:1 Threads



m:n Threads

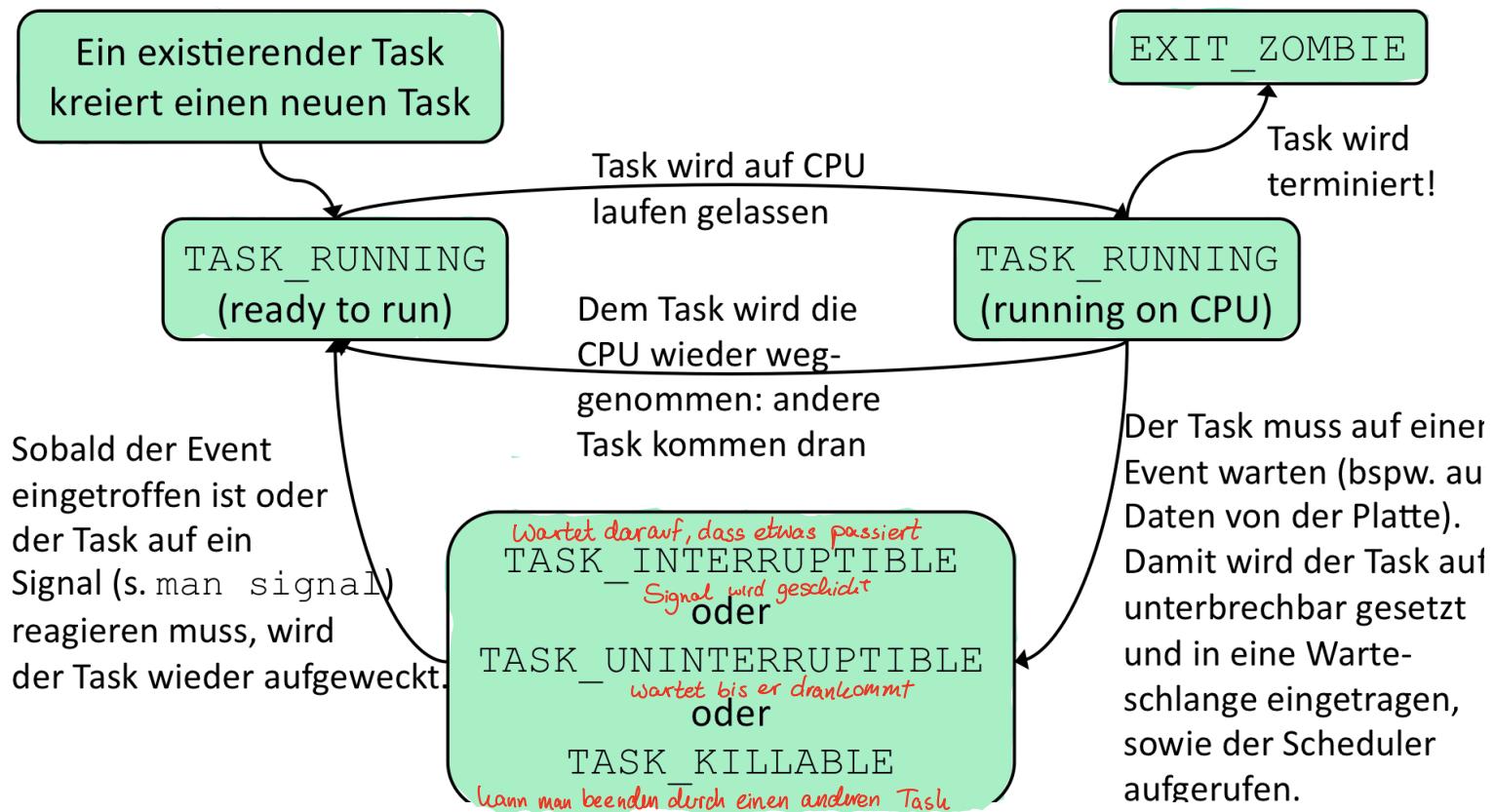


→ Vorteilhaft: Die Taskus können einzeln abgeschaltet werden

Linux verwaltet für jeden Thread jedes Prozesses eine Task-Struktur als `struct task_struct { ... }` `include/linux/sched.h`

- Dort sind Informationen über:
  - PID & TID
  - Status (runnable, stopped)
  - Zeiger auf den Stack
  - Prozess-Flags, Prioritäten
  - CPU-Maske, auf der der Task laufen kann

- Jeder Task kann folgende Zustände einnehmen:



- Zombie Tasks?

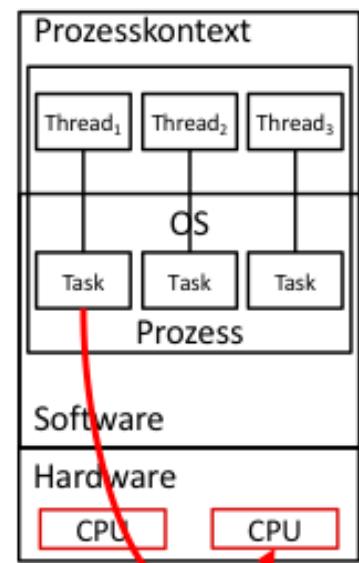
- Child Task, welcher seinen Rückgabewert noch nicht an den Parent Prozess zurückgeben konnte
- Kann nicht geschlossen werden, da man nicht weiß, ob der Wert später noch benötigt wird

- Zombies verhindern:

1. Rückgabewerte auslesen nach Beendigung des Child mit `waitpid`
2. System mitteilen, dass kein Rückgabewert benötigt wird mit `signal( SIGCHLD, SIG_IGN ) ;`

- Er ist ein „Completely Fair Scheduler“ (CFS), d.h. er modelliert eine CPU, die jeden der `nr_running` Prozesse mit  $1/nr\_running$  CPU-Zeit parallel ausführt.
- Jeder Task ist in einer Scheduling-Klasse:
  - » `SCHED_NORMAL`: Der Default (früher `SCHED_OTHER`)
  - » `SCHED_FIFO`: FIFO-Klasse (RT scheduling)
  - » `SCHED_RR`: Round-Robin Klasse (RT scheduling)
  - » `SCHED_BATCH`: CPU-intensive Programme
  - » `SCHED_DEADLINE`: Implementiert Earliest Deadline First Alg., Kernel 3.14
- Jeder Task erhält Zeitkontingent (virtual runtime `vruntime` in ns), von dem er zehrt – und ist genau in einer RunQueue `rq`!

- Konstanter Aufwand ( $O(1)$ )
- Zeitscheiben:
  - Jeder Task erhält eine feste Zeitscheibe nach der er unterbrochen (preempted) wird
  - Die Zeitscheibe wird basierend auf der Priorität des Prozesses festgelegt (140 Stufen)
    - Normale Priorität 100 ms (Level 100)
    - Höchste Priorität 10 ms (Level 0)
    - Niedrigste Priorität 200 ms (Level 140)
- Faktoren, die die Priorität beeinflussen:
  - Prioritätsbonus für interaktive Prozesse (Prozesse die viel auf externe Eingaben warten). Bonus: +/-5 Level
  - Priorität kann mit Befehl `nice` erhöht und verringert werden



- Scheduler verwendet zwei Arrays:
  - Active, expired
- Jedes Array hat pro Priorität eine verzeigerte Liste mit Prozessen
- Ablauf:
  - Prozesse, die ihre Zeitscheibe aufgebraucht haben werden von Active in Expired verschoben
  - Sobald alle Prozesse gelaufen sind befinden sich alle Prozesse im Array Expired, das Array Active ist leer
  - Pointer auf Active und Expired werden getauscht
- Jeder Teil des Algorithmus läuft in konstanter Zeit (unabhängig von der Anzahl der Tasks)
- Problem: Heuristiken zur Einordnung der Tasks in Prioritäten ist aufwändig und fehleranfällig
- Immer wenn ein Kontext-wechsel ansteht: Wähle stets task mit geringster vruntime
  - Erhöhe vruntime um Laufzeit wann immer ein Prozess läuft
  - Keine unterschiedlichen Listen/Strukturen sondern eine einzige Baumstruktur für alle Prioritäten. Jedoch Gewichtung der Erhöhung von vruntime → vruntime von niederprioren Prozessen wird stärker erhöht als von hochprioren Prozessen
    - $vruntime += t * weight$
    - Geringere Prioritätszahl: vruntime erhöht sich langsamer / Zeit vergeht langsamer für diesen Prozess
  - Interaktive Prozesse mit viel Wartezeit auf I/O verbrauchen weniger Zeit → vruntime bleibt geringer
  - → sie kommen automatisch häufiger dran.
  - → Vorteil: es müssen keine aufwändigen Heuristiken analysiert werden, um festzustellen welche Prozesse interaktiv sind (Vgl. O(1) Scheduler)

