



# Betriebssysteme

Prof. Dr. Thomas Fuchß  
Hochschule Karlsruhe – Technik und Wirtschaft  
Fakultät für Informatik und Wirtschaftsinformatik



# Inhaltsverzeichnis

- Grundlagen
- Virtualisierung (Teil I) – Virtualisierung der CPU
- Persistenz, Dateisysteme und I/O
- Virtualisierung (Teil II) – Virtualisierung des Hauptspeichers
- Concurrency: Prozesse und Threads

**„Betriebssysteme Übung“ ab dem ....  
Prof. Dr. Waldhorst**



# Literatur

**Die Vorlesung basiert im Wesentlichen auf folgendem Buch.**

- Arpaci-Dusseau, Remzi H.; Arpaci-Dusseau, Andrea C.  
**Operating Systems: Three Easy Pieces, (V. 0.90) Arpaci-Dusseau Books, 2015**  
(<http://pages.cs.wisc.edu/~remzi/OSTEP/>)

**Dies ist frei verfügbar und sollte gelesen werden.**



# Literatur

## Darüber hinaus empfehlenswerte Bücher:

- Tanenbaum, Andrew S.; Bos, H.  
Modern Operating Systems (4th Edition) – Pearson, 2014
- Stallings, W.  
Operating Systems: Internals and Design Principles (8th Edition) – Pearson, 2014
- Kerrisk, M.  
The Linux Programming Interface: A Linux and UNIX System Programming Handbook – No Starch, 2010
- Hailperin, Max.  
Operating Systems and Middleware: Supporting Controlled Interaction, Edition 1.2.1, 2016  
(Creative Commons Attribution-Share Alike 3.0 Unported License)  
<https://gustavus.edu/mcs/max/os-book/>
- Glatz, Eduard.  
Betriebssysteme: Grundlagen, Konzepte, Systemprogrammierung – dpunkt, 2010



# Literatur

## Die Programmbeispiele sind in der Regel in C:

- Bröckl, U.; Dausmann, M.; Goll, J.  
C als erste Programmiersprache (Auflage: 7) – Springer Vieweg, 2010
- Wolf , J.  
Grundkurs C: C-Programmierung verständlich erklärt – Galileo Computing, 2010
- Wolf , J.  
C von A bis Z: Das umfassende Handbuch (Auflage: 3) – Galileo Computing, 2009
- Kernighan, B. W.; Ritchie, D.  
The C Programming Language (2nd. Edition) – Prentice Hall, 1988
- `:> man man` Der Linux Manual Pager



# Literatur

## Papiere und Aufsätze:

- nach Bedarf

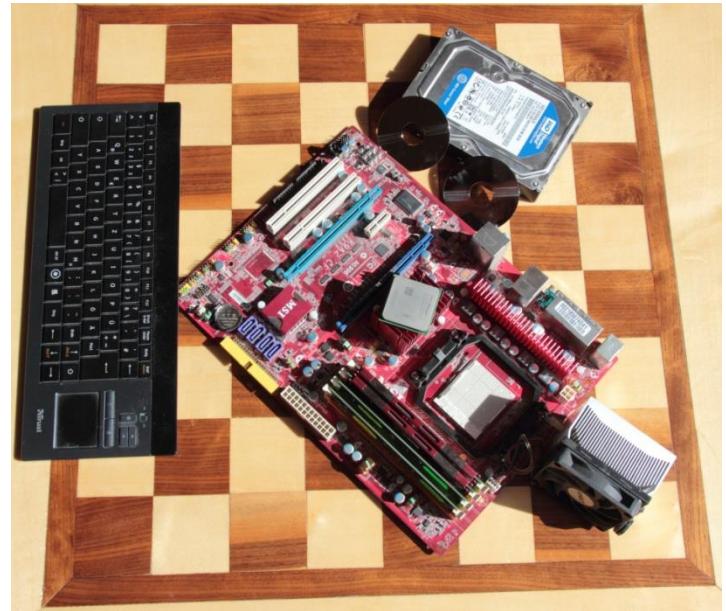


# Grundlagen



# Grundlagen

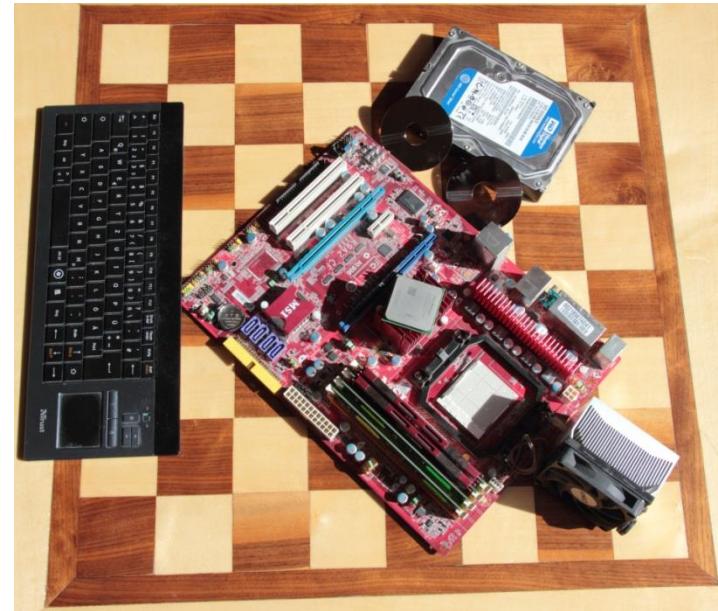
- Was ist ein Betriebssystem?
- Was sind die Herausforderungen?
- Was gilt es zu lernen?





# Was ist ein Betriebssystem?

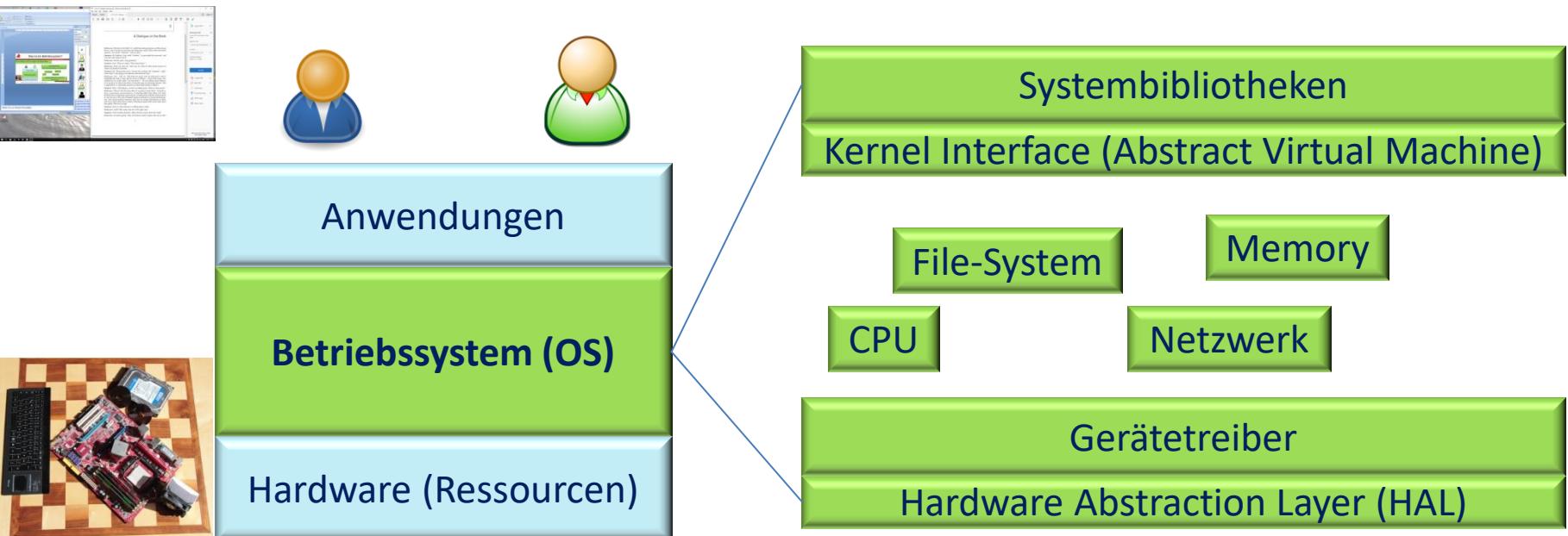
Eine Software, die die unterschiedlichen Hardware-Ressourcen eines Rechensystems managet und den Anwendungsprogrammen eine einheitliche Schnittstelle bietet.





# Was ist ein Betriebssystem?

Eine Software, die abstrahiert und managet.





# Was ist ein Betriebssystem?

Eine Software, die abstrahiert.

- Standardbibliotheken für Systemressourcen

Systembibliotheken

Kernel Interface (Abstract Virtual Machine)

- für die CPU Prozesse und Threads
- für den Speicher virtuelle Adressen
- für die Platte Dateien und Verzeichnisse
- für das Netzwerk Sockets

File-System

Memory

CPU

Netzwerk



# Was ist ein Betriebssystem?

- Vorteil:

Anwendungen müssen sich nicht um hardwarenahe Aufgaben kümmern, sondern bekommen diese einheitlich und geräteunabhängig angeboten.

- Problem:

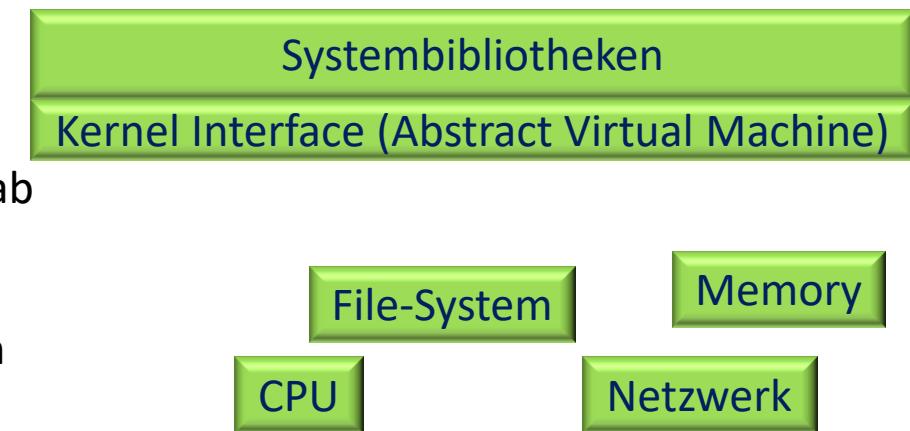
Was ist die richtige Abstraktionsebene?  
Welche Funktionalität ist erforderlich?  
Wie viel Hardware darf man sehen?



# Was ist ein Betriebssystem?

Eine Software, die managet.

- ist Moderator
  - teilt die Ressourcen zwischen den Anwendungen auf
  - schirmt die Anwendungen untereinander ab
- und Magier
  - Jede Anwendung hat die Maschine für sich
  - Speicher ist unendlich
  - Dateien können beliebig wachsen
  - ...





# Was ist ein Betriebssystem?

- Vorteil:

Anwendungen können sich nicht gegenseitig stören. Die Zuteilung von Ressourcen erfolgt fair und effektiv.

- Problem:

Wie kann Abschirmung erreicht werden?  
Wie kann Hardware virtualisiert werden?  
Was ist eine faire Zuteilung?



# Was sind die Herausforderungen?

- **Zuverlässigkeit und Verfügbarkeit**
  - Wie verhält sich das System im Fehlerfall?
  - Wie werden Fehler erkannt?
  - Wie werden Fehler korrigiert?
- **Sicherheit**
  - Wie hoch ist die Wahrscheinlichkeit, dass ein spezifischer Angriff stattfindet?
  - Wie hoch ist die Wahrscheinlichkeit, dass ein Sicherheitsmechanismus versagt?
- **Privacy**
  - Wie können Anwendungen und Nutzer voneinander abgeschirmt werden?



# Was sind die Herausforderungen?

- Virtualisierung

Wie kann erreicht werden, dass jede Anwendung glaubt:

**Sie besitzt die Maschine und ihre Hardware alleine!?**

**Die ihr zur Verfügung stehenden Ressourcen seien unbegrenzt!?**



# Was sind die Herausforderungen?

- **Concurrency (Nebenläufigkeit)**  
**Vieles geschieht gleichzeitig und Anwendungen müssen miteinander kooperieren.**

**Das Betriebssystem muss dies organisieren.**



- Auf Ereignisse reagieren
- Anwendungen trennen
- Anwendungen koordinieren (Locks, Semaphore, kritische Abschnitte, ...)



# Was sind die Herausforderungen?

- **Persistence (Datenspeicherung und Datenzugriff)**  
Daten müssen über die zeitlichen Grenzen einer Anwendung hinaus verfügbar gehalten werden.

Auch hierfür muss das Betriebssystem geradestehen und die entsprechenden Mechanismen bereithalten.



# Was sind die Herausforderungen?



- **Abstraktion:**  
Applikationen kennen lediglich Dateien und Verzeichnisse.
- **Performance:**  
Laufwerke sind langsam, der Zugriff muss aber schnell erfolgen.
- **Reliability:**  
Hardware ist fehleranfällig. Fehler müssen maskiert und korrigiert werden.



# Warum sollte man dies lernen?

- Betriebssysteme sind toll und wir alle werden ein eigenes bauen!
- Betriebssysteme sind toll und wir alle werden später an der Entwicklung von Android, iOS, Windows und Linux mitwirken!

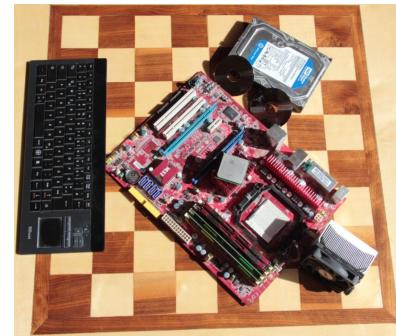
?

**Eher unwahrscheinlich, aber nicht ausgeschlossen!**



# Warum sollte man dies lernen?

**Das Betriebssystem beeinflusst maßgeblich das Verhalten der Maschine!**



- Wissen über das Betriebssystem ermöglicht die Entwicklung performanter Software.
- Wissen über das Betriebssystem ermöglicht es Entscheidungen zu treffen.  
(z.B.: bei Fragen zur Hardware-Beschaffung und Konfiguration)



# Warum sollte man dies lernen?

**Virtualisierung und Abstraktion sind  
fundamentale Techniken der Informatik.**

- Die vermittelten Datenstrukturen und Algorithmen sind von zentraler Bedeutung für die Entwicklung.
  - Verteilter Systeme
  - Mobiler Systeme
  - Datenbank Systeme
  - ...



# Prozesse benötigen Rechenzeit

## Teil I

# Virtualisierung der CPU oder die Illusion von der eigenen Maschine



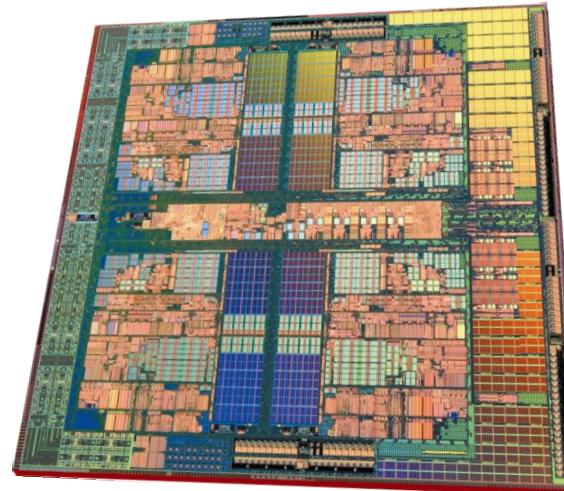
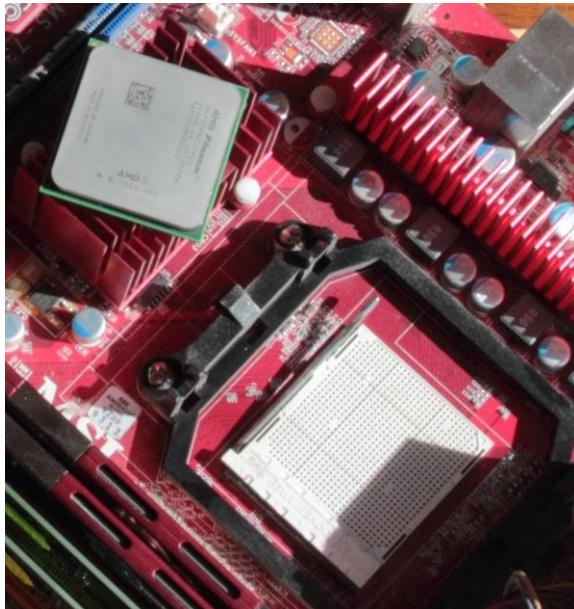
# Virtualisierung der CPU

- CPU-Virtualisierung (**Prozessorvirtualisierung**)
  - Rechnerarchitektur
  - Prozesse
  - Dispatcher und Scheduling



# Etwas Rechnerarchitektur

## Aufbau eines Prozessors

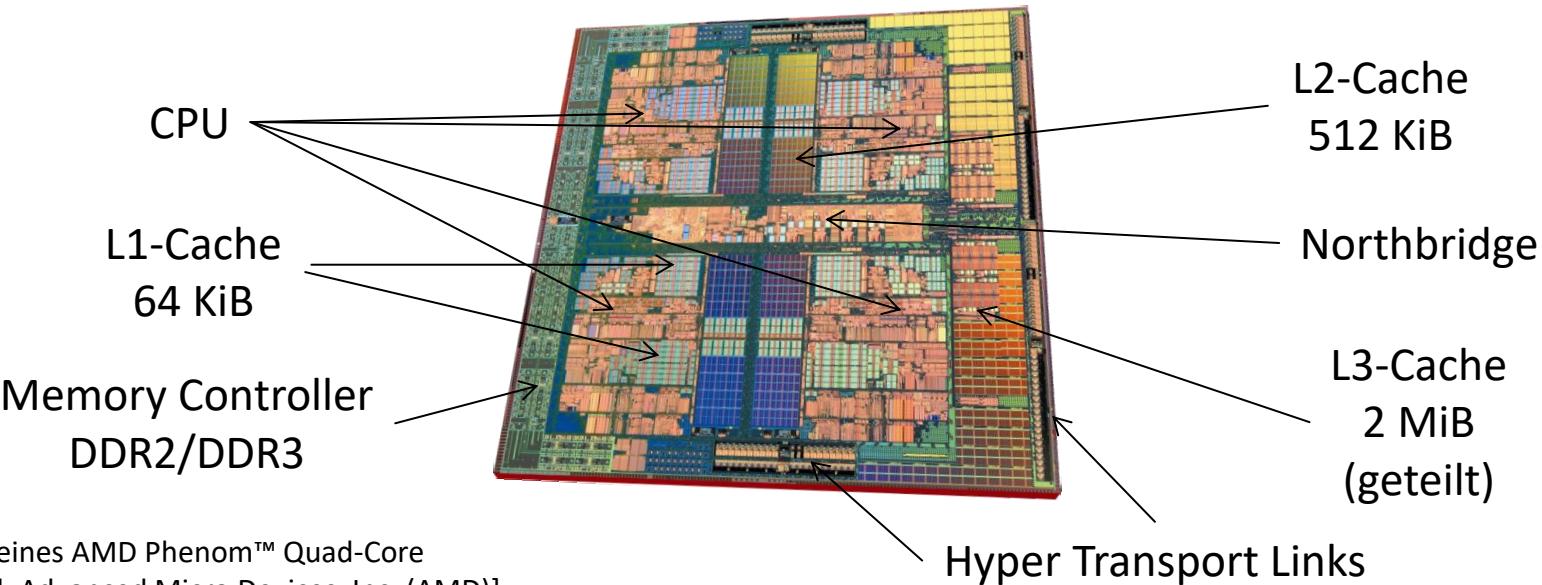


Entsprechendes Die eines AMD Phenom™ Quad-Core  
[Bild: Advanced Micro Devices, Inc. (AMD)]



# Etwas Rechnerarchitektur

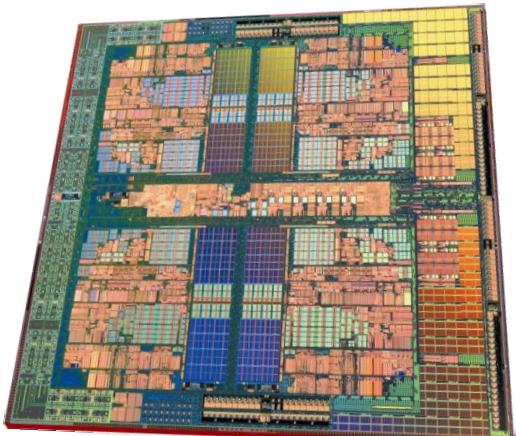
## Aufbau eines Prozessors



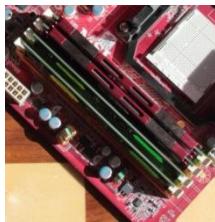
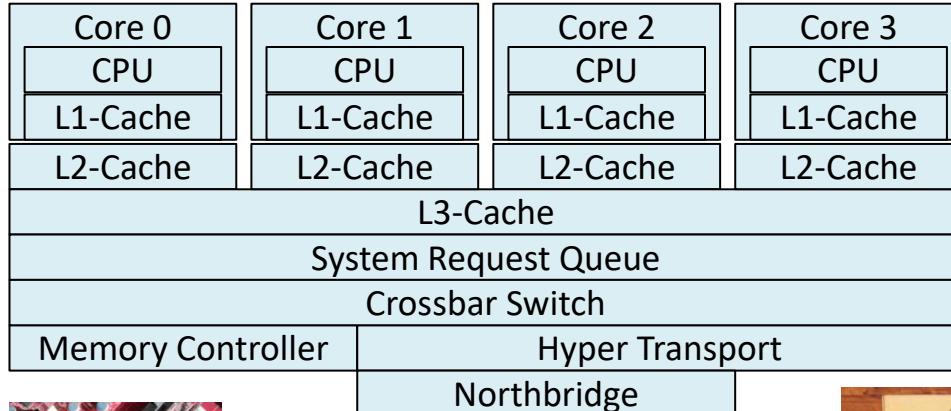
Die eines AMD Phenom™ Quad-Core  
[Bild: Advanced Micro Devices, Inc. (AMD)]



# Etwas Rechnerarchitektur



Die eines AMD Phenom™ Quad-Core  
[Bild: Advanced Micro Devices, Inc. (AMD)]

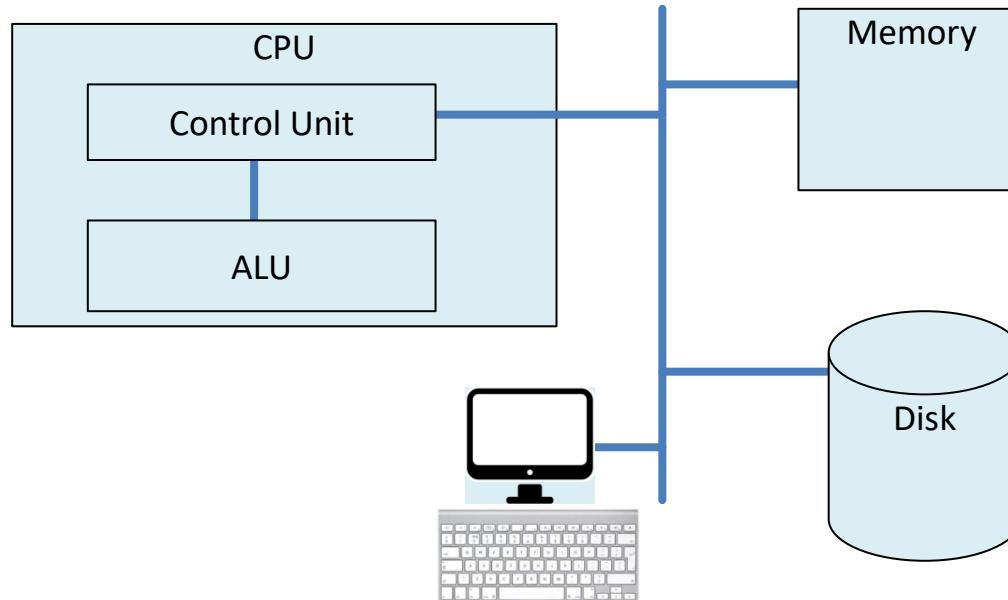




# Etwas Rechnerarchitektur

vereinfacht

Steuerwerk

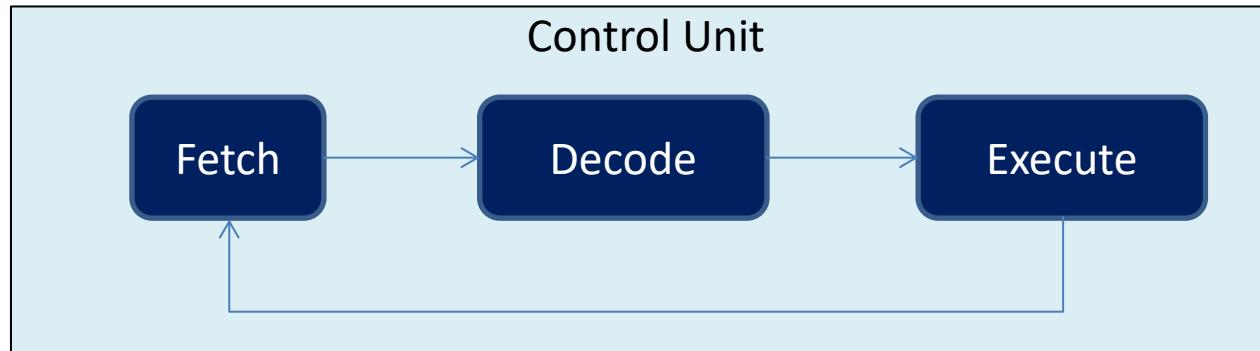




# Aufgaben der CPU

Ausführung eines Programms.

- **Befehle aus dem Speicher holen und diese ausführen!**  
Art und Anzahl der Befehle sind von der Architektur des Systems abhängig (ARM, x86, SPARC, ...)

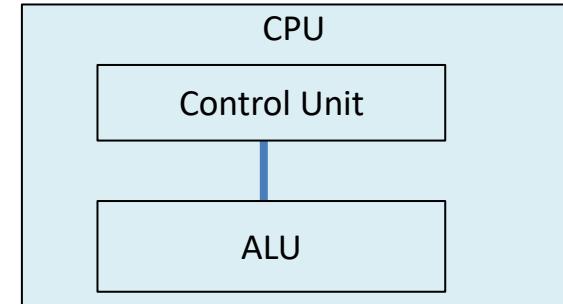




# Aufgaben der CPU

## Ausführung eines Programms.

- **CPU interne Register speichern Daten und Metadaten**
  - Universelle Register, Register für Gleitkommazahlen, ...
  - Befehlszeiger (Instruction Pointer), Stack Pointer
  - Statusregister (PSW)
    - Carry-Flag
    - Zero-Flag
    - Sign-Flag
    - ...

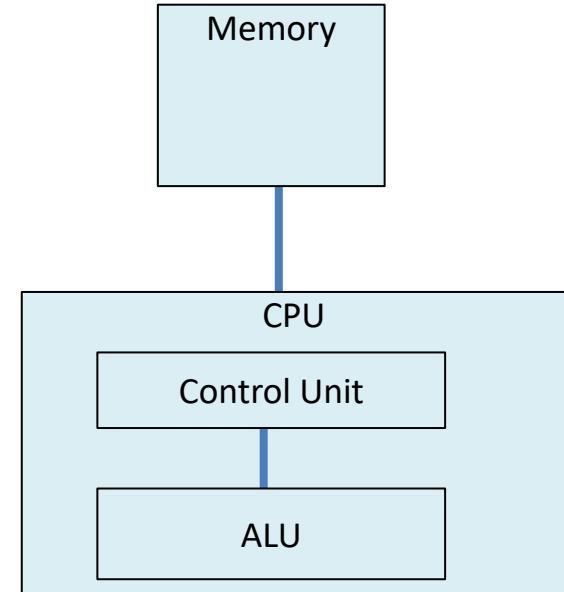




# Caches sorgen für Performance

**Die CPU ist schneller als jeder Speicher**

- Registerzugriff 1 CPU-Zyklus (<1ns)
- Cache-Zugriffszeiten (Bsp.: AMD Phenom – K10)
  - L1 Data Cache (64 KiB) 3 CPU-Zyklen ( $\approx$  1ns)
  - L1 Instruction Cache (64 KiB) 3 CPU-Zyklen
  - L2 Cache (512 KiB) 15 CPU-Zyklen
  - L3 Cache (6MiB)  $\approx$  50 CPU-Zyklen
  - RAM  $\approx$  100ns





# Ein Programm

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char** argv)
{
    const char* greetings = "Hello World";
    printf("%s\n", greetings);
    return EXIT_SUCCESS ;
}
```

```
::> gcc -Wall prog.c -o prog
::> ./prog
Hello World
::>
```

C-Programm in der Datei prog.c



# Von der CPU ausgeführt wird

```
:> objdump -d prog > prog.code  
:>
```

Der auszuführende Maschinen-Code

000000000040079a <main>:	
40079a:	55
40079b:	48 89 e5
40079e:	48 83 ec 20
4007a2:	89 7d ec
4007a5:	48 89 75 e0
4007a9:	48 c7 45 f8 17 08 40
4007b0:	00
4007b1:	48 8b 45 f8
4007b5:	48 89 c7
4007b8:	e8 c3 fc ff ff
4007bd:	b8 00 00 00 00
4007c2:	c9
4007c3:	c3
4007c4:	66 2e 0f 1f 84 00 00
4007cb:	00 00 00
4007ce:	66 90

push	%rbp
mov	%rsp,%rbp
sub	\$0x20,%rsp
mov	%edi,-0x14(%rbp)
mov	%rsi,-0x20(%rbp)
movq	\$0x400817,-0x8(%rbp)
mov	-0x8(%rbp),%rax
mov	%rax,%rdi
callq	400480 <puts@plt>
mov	\$0x0,%eax
leaveq	
retq	
nopw	%cs:0x0(%rax,%rax,1)
xchg	%ax,%ax



# Vom Programm zum Prozess

Als Prozess bezeichnet man die sequentielle Ausführung eines Programms.

- Ein Prozess ist eine Verwaltungseinheit bestehend aus:

- Programm
  - Daten (Heap / Stack)
  - Register
  - offen Dateien
  - Netzwerkverbindungen
- } Adressraum

All diese Informationen werden im Process Control Block (PCB) zusammen gefasst.



# Vom Programm zum Prozess

**Ein Prozess ist kein Programm!**

- Ein Programm ist etwas Statisches, bestehend aus Code und Daten.
- Ein Prozess ist die Ausführung des Programms, etwas Dynamisches.
- Zu einem Programm kann es viele Prozesse geben, die dieses Programm ausführen.
- Jeder Prozess hat einen eindeutigen Namen, **die Prozess ID (PID)**

```
:> ps -aux
USER      PID %CPU %MEM   VSZ   RSS TT STAT STARTED          TIME COMMAND
fuchss  1014  0.0  0.0   300   224  0 R    5:02PM  0:00.00 ./prog
fuchss  1015  0.0  0.0   300   224  0 R    5:02PM  0:00.00 ./prog
fuchss  1016  0.0  0.0     0     0  0 Z    5:02PM  0:00.01 <defunct>
```

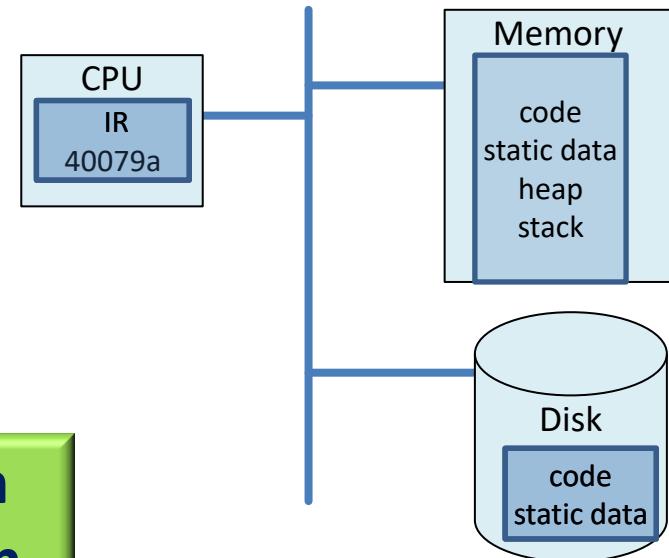


# Erste Ideen

## Wie kann ein Programm zur Ausführung gebracht werden?

- **Idee: Direkte Ausführung durch die CPU.**
  - Betriebssystem erzeugt lediglich den Prozess (lädt das Programm und reserviert den Adressraum)
  - übergibt die Kontrolle an „main()“ (setzt den Instruction Pointer)
- **Vorteil:**
  - schnell, einfach, simpel

**Das Betriebssystem ist lediglich  
eine Sammlung von Bibliotheken.**





# Probleme

- Der laufende Prozess kann jetzt alles tun!
  - beliebige Daten ändern und lesen
- Es läuft nur ein Prozess, alle andern müssen warten!
  - im schlimmsten Fall für immer
- Es wird Rechenzeit verschenkt!
  - ein Prozess kann nichts tun und warten:
    - auf eine User-Eingabe
    - auf Daten aus dem Speicher
    - ...



# Lösung: Schritt eins

- **Der laufende Prozess kann jetzt alles tun!**
  - beliebige Daten ändern und lesen
- Ab jetzt nicht mehr: Es gibt begrenzte Zugriffsrechte!
  - Prozesse laufen in unterschiedlichen Modi.
  - Der Zugriff auf bestimmte Register und Speicherbereiche ist nur in einem privilegierten Modus möglich.

**Limited Direct Execution:**  
**Das Betriebssystem bleibt weiter Herr der Lage!**



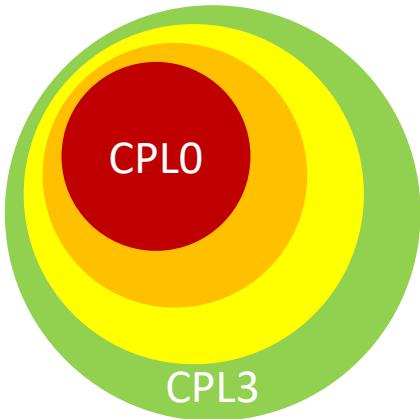
# Absicherung von Prozessen

- Normale Prozesse laufen im **User Mode** (restricted Mode)
- Das Betriebssystem läuft im **Kernel Mode** (privileged Mode)  
Dieser ist erforderlich für:
  - das Schreiben in Gerätereister
  - den Zugriff auf Memory/ Platte
  - das Anhalten der CPU
  - das Ändern des Modes
  - das Aktivieren/Deaktivieren von Interrupts

Weitere Abstufungen sind möglich, aber  
eher von geringer Bedeutung.



# Idee



## Protection Ring (Current Privilege Level)

- MULTICS            8 Ringe
- x86                4 Ringe
- **Linux und Windows nutzen 2**
- ARM                3 Ringe
  - application
  - operating system
  - hypervisor (-1)

**Problem:**  
Der Ring-Switch ist  
aufwändig –  
etwa 1000 Zyklen,  
100 für den Wechsel  
plus BS-Overhead.



# Idee

## Noch offene Punkte:

- Wie lassen sich diese Stufen realisieren?
- Wie kann man auf geschützte Funktionalität zugreifen?  
(Hardware, I/O, usw.)
- Was geschieht wenn man sich nicht an die Regeln hält?



# Umsetzung

## Wie lassen sich diese Stufen realisieren?

via Hardware (x86 2 Bits im CS-Register):

- Der Prozessor prüft, ob die notwendigen Berechtigungen vorliegen.
- Dies geschieht bei jeder Instruktion!

Beispiele für Level 0 Instruktionen

- HLT – Halt:  
Stoppt den Prozessor bis zum nächsten Interrupt.
- POPF – Pop Stack into FLAGS Register:  
Ändert das Statusregister und damit u.a. Berechtigungen für den I/O-Zugriff

Mehr Infos online im  
x86 Manual

“Combined Volume Set of Intel® 64 and IA-32  
Architectures Software Developer’s Manuals”  
(online [www.intel.com](http://www.intel.com))



# Umsetzung

**Wie kann man auf geschützte Funktionalität zugreifen?**

**Man muss dem Betriebssystem die Kontrolle wieder zurückgeben.**  
**In der Hoffnung, dass man sie später wieder zurückbekommt.**

- Man braucht eine definierte Falltür, um auf die Kernelebene abzusteigen.  
**(Eine Trap-Instruktion)**
- Man braucht ein geeignetes Interface, um die richtige Funktionalität auszuwählen!

```
mov    $0x4,%eax  
int    $0x80
```

Systemaufruf (4 ≡ Linux write)  
Wechsel in den Kernel Mode (Trap)



# Umsetzung

## Andere Architekturen andere Instruktionen:

Architektur	Instruktion	Call-Nr.	Return Value
alpha	callsys	v0	a0
arm64	svc #0	x8	x0
i386	int \$0x80	eax	eax
x86_64	syscall <b>(AMD)</b>	rax	rax
x86_64	sysenter <b>(Intel)</b>	rax	rax
ia64	break 0x100000	r15	r8
mips	syscall	v0	v0
			[syscall(2)]



# Umsetzung

## Linux Systemaufrufe (System Calls):

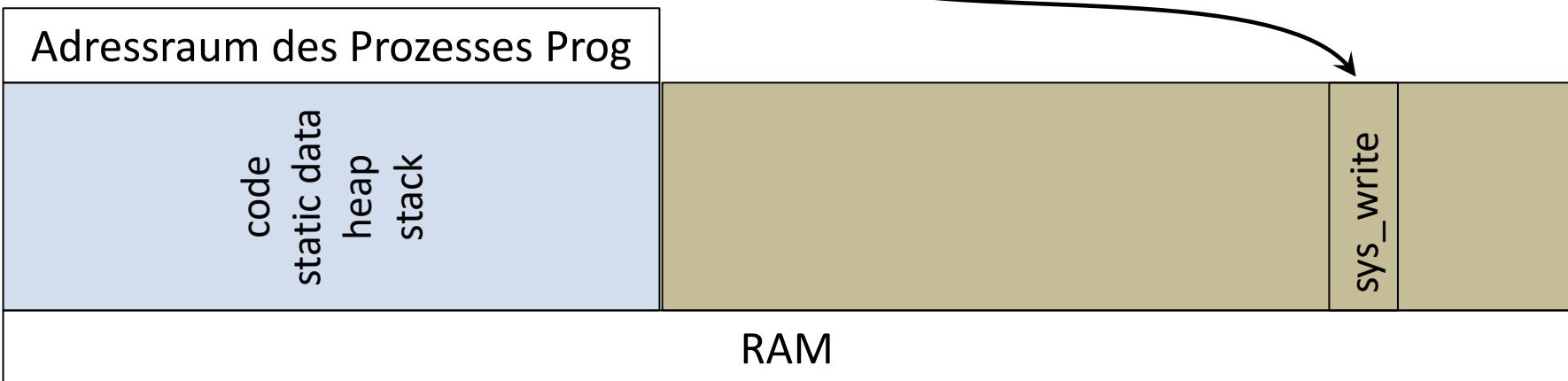
Nr.	Name	Entry Point	Implementierung
3	read	sys_read	fs/read_write.c
4	write	sys_write	fs/read_write.c
5	open	sys_open	fs/open.c
6	close	sys_close	fs/open.c
...			



# Umsetzung

```
int main(){  
    const char* c = "Hello World";  
    printf("%s\n", c);  
    return EXIT_SUCCESS ;  
}
```

Für die Ausgabe muss der Prozess Prog auf die Adresse SYS\_WRITE zugreifen.

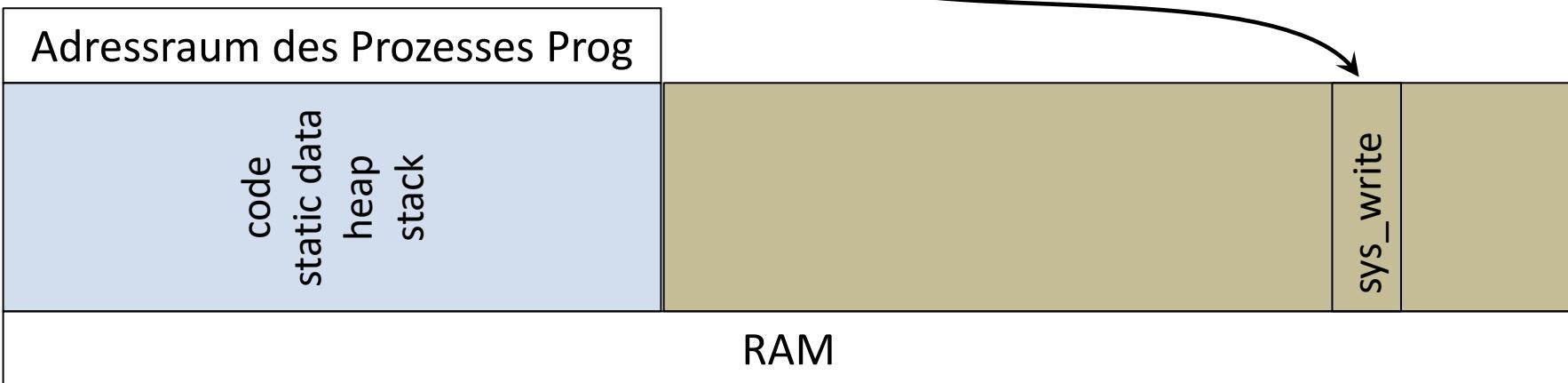




# Umsetzung

```
int main(){
    const char* c = "Hello World";
    printf("%s\n", c);
    return EXIT_SUCCESS ;
}
```

Diese ist außerhalb des eigenen Adressraums und damit ist der Zugriff unmöglich.  
(User Mode CPL3)

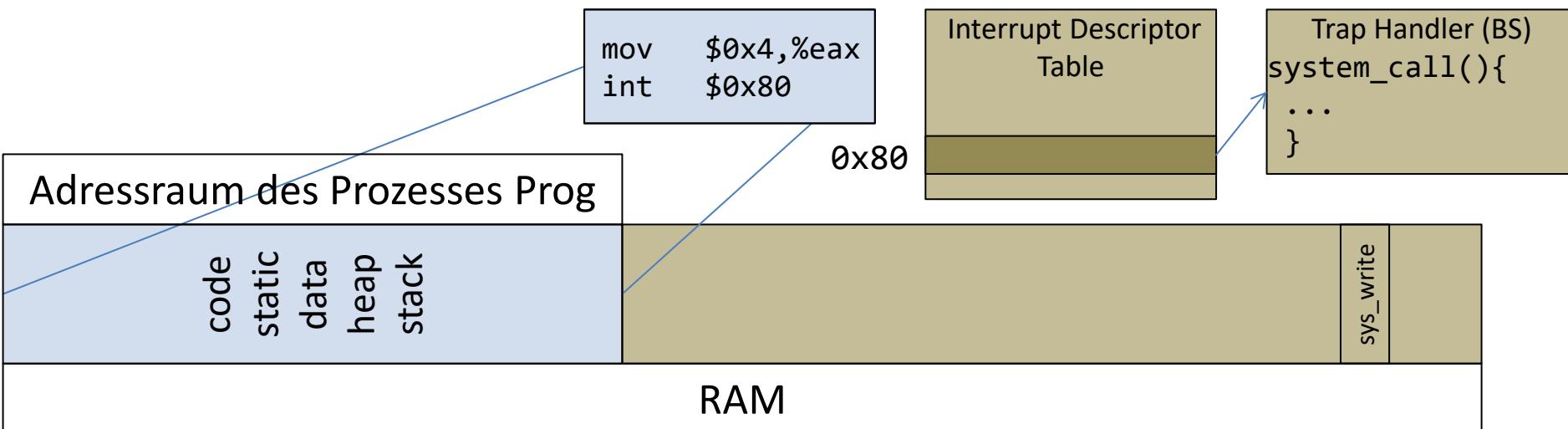




# Umsetzung (x86)

```
int main(){
    const char* c = "Hello World";
    printf("%s\n", c);
    return EXIT_SUCCESS ;
}
```

1. Kontrolle an den BS-spezifischen Trap Handler übergeben und in den Kernel Mode wechseln.





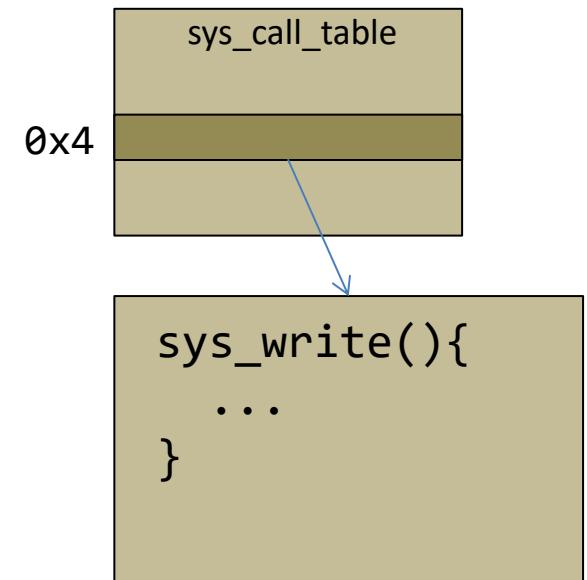
# Umsetzung (x86)

```
int main(){
    const char* c = "Hello World";
    printf("%s\n", c);
    return EXIT_SUCCESS ;
}
```

2. Trap Handler führt den entsprechenden Aufruf aus.

```
Trap Handler (BS)
240      # system call handler stub
241 ENTRY(system_call)
...
248      cmpl $(nr_syscalls), %eax
249      jae syscall_badsys
250 syscall_call:
251      call *sys_call_table(%eax,4)
252      movl %eax,EAX(%esp) # store the return value
```

Linux/arch/i386/kernel/entry.S





# Umsetzung (x86)

```
int main(){
    const char* c = "Hello World";
    printf("%s\n", c);
    return EXIT_SUCCESS ;
}
```

3. Trap Handler kehrt nach erfolgreichem Aufruf zurück, legt die Return-Werte an die richtige Stelle und führt die `return-from-trap` Instruktion `IRET` aus.
4. Die CPU wechselt zurück in den User Mode und das Programm wird weiter ausgeführt.

Trap Handler (BS)

```
125 #define RESTORE_ALL      \
126     RESTORE_REGS      \
127     addl $4, %esp; \
128 1:    iret;          \
```

[Linux/arch/i386/kernel/entry.S](#)



# Zusammenfassung

1. Das BS weist jedem Systemaufruf eine eindeutige Nummer zu und initialisiert die Sprungtabelle mit den korrekten Funktionen.
2. Das BS registriert den Trap Handler.  
(zur Boot-Zeit des Systems)

Zur Laufzeit des Anwendungsprogramms:

3. Der User-Prozess löst den Systemaufruf aus ( x86 via INT 0x80 ).



# Zusammenfassung

4. Die CPU wechselt in den Kernel Mode.
5. Die CPU übergibt dem Trap Handler die Kontrolle.
6. Der Trap Handler bestimmt den Systemaufruf mittels Tabelle und startet ihn.
7. Die hinterlegte Funktion realisiert die gestellte Aufgabe  
(und kopiert das Ergebnis an die gewünschte Stelle, z.B. Puffer im Speicher)
8. Der Trap Handler kehrt zurück (x86 via IRET).



# Zusammenfassung

9. Die CPU wechselt zurück in den User Mode.
10. Die CPU führt den anstehende Prozess weiter aus.



# Missachtung von Regeln

## Was geschieht wenn man sich nicht an die Regeln hält?

- Das Betriebssystem wird informiert
- und terminiert den Prozess.

```
:> gcc -Wall prog.c -o prog
:> ./prog
Segmentation fault (core dumped)
:>
```

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char** argv)
{
    const char* greetings = "Hello World";
    printf("%s\n", greetings - 10000);
    return EXIT_SUCCESS ;
}
```



# Probleme

- Der laufende Prozess kann jetzt alles tun!
    - beliebige Daten ändern und lesen
  - Es läuft nur ein Prozess, alle andern müssen warten!
    - im schlimmsten Fall für immer
  - Es wird Rechenzeit verschenkt!
    - ein Prozess kann nichts tun und warten:
      - auf eine User-Eingabe
      - auf Daten aus dem Speicher
- } erledigt
- } noch offen



# Lösung: Schritt zwei

- Es läuft nur ein Prozess, alle andern müssen warten!
  - im schlimmsten Fall für immer
- ...

**Ab jetzt nicht mehr:**

- Das BS muss in die Lage versetzt werden einen Prozess anzuhalten.
- Das BS muss in die Lage versetzt werden, zwischen Prozessen zu wechseln.
- Das BS muss entscheiden können, welcher Prozess als nächstes ausgeführt wird.

**Das Betriebssystem managet die Prozesse!**



# Lösung: Schritt zwei

**Man unterscheidet stets zwischen:**

## Mechanismus:

- Wie kann ein Prozess angehalten werden?
- Wie kann ein anderer gestartet werden?

}

Dispatcher

## Strategie:

- Wann wird ein Prozess angehalten?
- Mit welchem wird fortgefahrene?

}

Scheduler



# Der Dispatcher (Zuteiler):

- Dispatch Loop (der Mechanismus)

```
while (true) {  
  
    run process for a while;  
    stop process and save its state (context);  
    load context of another process;  
  
}
```

1. Wie erlangt der Dispatcher, nachdem ein Prozess läuft, wieder die Kontrolle?
2. Welche Information (Kontext) muss gesichert werden?



# Der kooperative Ansatz:

**Das Betriebssystem vertraut den Anwendungen!**

- Systemaufrufe finden sicher regelmäßig statt (I/O, ...).
- Für Langläufer wird ein spezieller Systemaufruf eingeführt.

**yield()**

**to yield to sb**

**jdm den Vortritt lassen**

[pons: de.pons.com]



# Der kooperative Ansatz:

**Problem: Vertrauen ist gut, Kontrolle ist besser!**

- Niemand ist gezwungen, `yield` aufzurufen.
- Auch bei gutem Willen kann es zu Fehlern kommen.

```
unsigned int x = 100;
while (x >= 0) {
    ...
    x--;
}
yield();
```



# Nicht kooperativ

**In modernen Chips lösen Hardware Timer (HPET) in regelmäßigen Abständen (ca. 1ms) Interrupts aus.**

- Damit wird garantiert, dass das Betriebssystem wieder zum Zuge kommt.

**Das Betriebssystem hat die Kontrolle zurück!**



# Aufgaben von Hardware und Software

Hardware  
Control Unit

## Timer Interrupt

Software  
Dispatcher

- Sichert die Informationen, um den Prozess wieder zu starten.
  1. wechselt in den Kernel Mode
  2. kopiert den Inhalt zentraler Register in den Kernel Stack
  3. startet den Handler
  8. restauriert den Registerinhalt und wechselt in den User Mode
  9. führt den neuen Prozess aus
- Führt nach Bedarf den Prozesswechsel durch.
  4. sichert weitere prozessspezifische Register und sonstige Information (im PCB)
  5. wählt einen anderen Prozess
  6. restauriert die Registerinhalte und setzt den Stack Pointer des Kernel Stacks entsprechend der gesicherten Information
  7. kehrt zurück **IRET**



# Der Process Control Block

```
1458 struct task_struct {  
1459     volatile long state;      /* -1 unrunnable, 0 runnable, >0 stopped */  
1460     void *stack;  
1461     atomic_t usage;  
1462     unsigned int flags;      /* per process flags, defined below */  
1463     unsigned int ptrace;  
  
1921 /* CPU-specific state of this task */  
1922     struct thread_struct thread;  
1923 /*  
1924  * WARNING: on x86, 'thread_struct' contains a variable-sized  
1925  * structure. It *MUST* be at the end of 'task_struct'.  
1926  *  
1927  * Do not put anything below here!  
1928 */  
1929 };
```

Ausschnitt aus  
include/linux/sched.h



# Probleme

- Der laufende Prozess kann jetzt alles tun!
    - beliebige Daten ändern und lesen
  - Es läuft nur ein Prozess, alle andern müssen warten!
    - im schlimmsten Fall für immer
  - **Es wird Rechenzeit verschenkt!**
    - ein Prozess kann nichts tun und warten:
      - auf eine User-Eingabe
      - auf Daten aus dem Speicher
      - ...
- 
- erledigt**
- noch offen**



# Lösung: Schritt drei

- **Es wird Rechenzeit verschenkt!**
  - ein Prozess kann nichts tun und warten:
    - auf eine User-Eingabe, ...

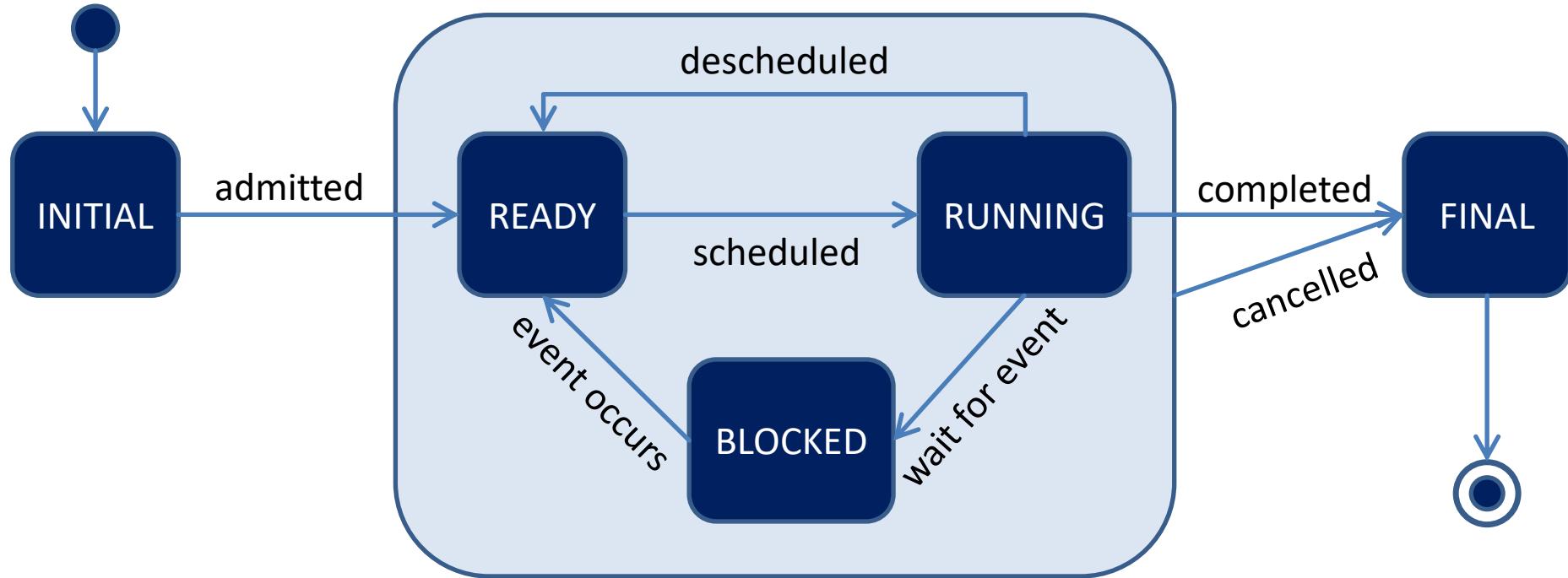
Auch hierfür gibt es ein adäquates Mittel:

- Das BS muss wissen, was jeder Prozess gerade macht.
- Wartet ein laufender Prozess auf ein Betriebsmittel, dann führt es am besten einen Kontextwechsel durch. (Neuer Prozess läuft)

**Das Betriebssystem managet und verwaltet die Prozesse!**



# Prozesse haben Zustände





# Prozesse haben Zustände

- **initial:** der Prozess wurde erzeugt
- **ready:** der Prozess ist in der Lage, ausgeführt zu werden
- **running:** der Prozess läuft (pro CPU ein Prozess)
- **blocked:** der Prozess wartet auf ein bestimmtes Ereignis (z.B.: I/O)
- **final:** der Prozess ist beendet, es wird aufgeräumt



# Prozesse haben Zustände

**Typischerweise unterhält das Betriebssystem hierfür diverse Queues in denen die PCBs verwaltet werden.**

- **Ready Queue:**

Enthält die PCBs der Prozesse, die zur Ausführung bereit wären.

- **Event Queue:**

Eine Queue für jede Art von Ereignis, auf das gewartet werden kann.  
(Plattenzugriff, Locks, usw.)



# Zusammenfassung CPU-Virtualisierung (Mechanismen)

- **Limited Direct Execution:**

Die Ausführung bleibt schnell und das Betriebssystem behält die Kontrolle.

- **Schnelle Kontextwechsel:**

Es entsteht der Eindruck als besäße jeder Prozess seine eigene CPU.

- **Hardware-Unterstützung:**

- Protection Ring
- Timer und Interrupts
- Sicherung und Wiederherstellung von Zustandsinformation (Register)



# Prozesse und ihre Erzeugung

- Zentrale Gedanken (SWE)

**Man programmiert auf eine Schnittstelle hin, nicht auf eine Implementierung.**

**Objekte konkreter Klassen werden durch Muster wie „Erbauer“ oder „abstrakte Fabrik“ erzeugt und nicht durch den Klient.**

[Gamma, E.; et. al. Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software. Addison-Wesley,2001]



# Prozesse und ihre Erzeugung

**Problem: Erzeugungsmuster benötigen in der Regel ein Objekt, das die Erzeugung steuert.**

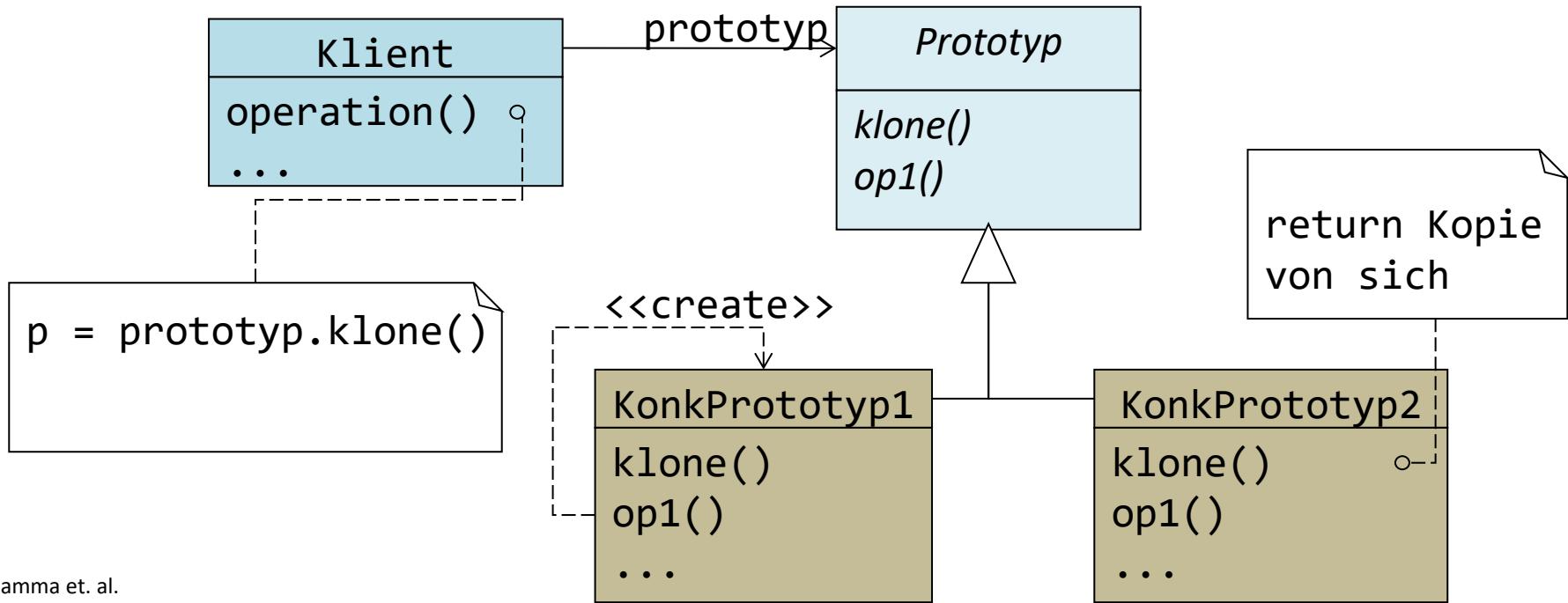
## Alternativen:

- **Fabrikmethode:** Man benötigt etwas, das sie implementiert und jemanden, der sie nach Wunsch parametriert.
- **Prototyp:** Definierte Methode, die ein bestehendes Objekt dupliziert bzw. klont!

**Vorteil: Man benötigt keine Parameter, keine Klassen, sondern nur etwas objektähnliches, das dupliziert werden kann – etwa ein Prozess.**



# Prototyp (nach Gama)



[Erich Gamma et. al.

Entwurfsmuster : Elemente wiederverwendbarer  
objektorientierter Software – Addison-Wesley, 2001.]



# Prozesse und ihre Erzeugung

Lässt man den abstrakten Prototypen weg, und macht man den Prototypen selbst zum Klienten, dann erhält man etwas, das sich selbst duplizieren kann.

## Linux:

- **fork:**

Dupliziert (klont) den aktuellen Prozess.

- **exec:**

Ersetzt den auszuführenden Code.



# Prozesse und ihre Erzeugung (Linux)

**fork:**

1. Unterbricht den aktuellen Prozess: State wechselt nach „BLOCKED“
2. Kopiert alles: PCB, Code, Daten und Stack

**Ein neuer Prozess existiert!**

3. Fügt die PCBs in die Ready Queue ein.

**Flexibel und einfach aber sehr  
aufwändig!**



# Beispiel (Linux)

```
pid_t pid, waitRes;  
int status;  
  
if ((pid = fork()) < 0) { /* Fehlerbehandlung: ... */}  
  
else if ( pid == 0) {/* Im Tochterprozess: Die Variable pid hat den Wert 0 */  
    execl(fileName, arg1, arg2, NULL);      // neues Programm ausführen  
}  
else {/* Vaterprozess: pid hat den Wert der PID des Tochterprozesses.*/  
    do {  
        if ((waitRes = wait(&status) < 0)  
            exit(EXIT_FAILURE);  
    } while (waitRes != pid);    // warten bis Tochter terminiert  
}
```



# Prozesse und ihre Erzeugung (Linux)

Mehr über die Linux-Process-API findet  
man in den Man Pages.

```
:> man 2 fork
```



# CPU-Virtualisierung (Strategien)

bisher:

- **Technik und Mechanismen (Dispatcher):**
  - führt den Kontextwechsel durch
  - sichert und restauriert die Register
  - verwaltet die PCBs
  - ...

**Typischerweise stehen viele Prozesse in der Ready Queue.  
Doch mit welchem soll der Kontextwechsel erfolgen?**



# CPU-Virtualisierung (Strategien)

**Diese strategische Entscheidung trifft der Scheduler!**

- **Ziele:**
  - optimale Auslastung des Prozessors und der übrigen Hardware,
  - gute Reaktionszeiten bei interaktiven Anwendungen,
  - hoher Durchsatz bei rechenintensiven Anwendungen,
  - gerechte Verteilung der Rechenzeit und Wartezeit,
  - ...



# Scheduling

Um Ziele zu erreichen, benötigt man Maße, um Werte zu messen und zu vergleichen!

- Metriken zur Bewertung der Performance:

- **Umlaufzeit (Turnaround Time)** (man möchte nicht unnötig warten)

$$T_{\text{Umlauf}} = T_{\text{Ende}} - T_{\text{Ankunft}}$$

- **Reaktionszeit (Response Time)** (Antworten sollten schnell erfolgen)

$$T_{\text{Reaktion}} = T_{\text{Start}} - T_{\text{Ankunft}}$$

- **Durchsatz = (erledigte Aufgaben) / Zeit** (Vieles muss gleichzeitig erledigt werden)



# Scheduling

Genauso wichtig ist Fairness!

- Berechnung des Fairness-Index nach Jain für n Prozesse:

$$F(x_1, x_2, \dots, x_n) = \frac{\left(\sum_{i=1}^n x_i\right)^2}{n \sum_{i=1}^n x_i^2}$$

[Jain, R.:The Art of Computer Systems Performance Analysis. Interscience, 1991]

Dabei ist  $x_i$  ein Maß für die Belegung der CPU durch den Prozess I.

Je näher der Fairness-Index bei 1 liegt, desto fairer ist die Zuteilung.



# Scheduling

## Ein sehr vereinfachtes Modell

- Alle Prozesse (kurz Jobs oder Aufgaben) benötigen dieselbe Zeit.
- Alle Jobs stehen zur gleichen Zeit zur Ausführung bereit.
- Einmal angefangen, läuft jeder Job bis zu seinem Ende durch.
- Alle Jobs verwenden nur die CPU, d.h., es gibt keine Hardware-Zugriffe.
- Die Zeit, die ein Job braucht, ist bekannt.



# Scheduling (Erste Idee)

Oftmals ist die simpelste Idee die  
beste Idee!

Die naheliegendste Idee:

- FIFO: First In, First Out

„Wer zuerst kommt, mahlt zuerst!“

[Repgow, et.al:Sassen Speyghel, 1234]

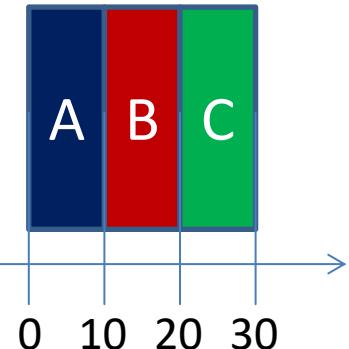
Aufgaben werden in der Reihenfolge bearbeitet,  
in der sie ankommen.



# Scheduling (FIFO)

## Performance-Betrachtung (Umlaufzeit):

- Gemäß idealisiertem Modell stehen alle Jobs zur gleichen Zeit bereit => Reihenfolge ist zufällig.
- Alle Jobs dauern gleich lang. (Laufzeit z.B.:10s)



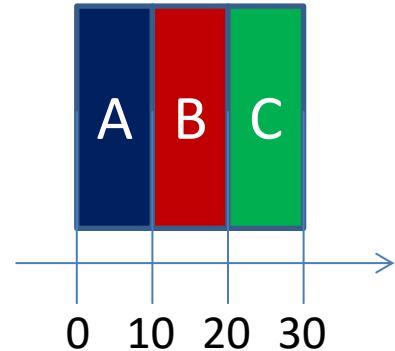
- **Umlaufzeit (A)** =  $T_{\text{Ende}}(A) - T_{\text{Ankunft}}(A) = 10\text{s} - 0\text{s} = 10\text{s}$
- **Umlaufzeit (B)** =  $T_{\text{Ende}}(B) - T_{\text{Ankunft}}(B) = 20\text{s} - 0\text{s} = 20\text{s}$
- **Umlaufzeit (C)** =  $T_{\text{Ende}}(C) - T_{\text{Ankunft}}(C) = 30\text{s} - 0\text{s} = 30\text{s}$



# Scheduling (FIFO)

## Performance-Betrachtung (Umlaufzeit):

- Gemäß idealisiertem Modell stehen alle Jobs zur gleichen Zeit bereit => Reihenfolge ist zufällig.
- Alle Jobs dauern gleich lang. (Laufzeit z.B.:10s)



durchschnittliche Umlaufzeit:

$$(10s + 20s + 30s) / 3 = 20s$$

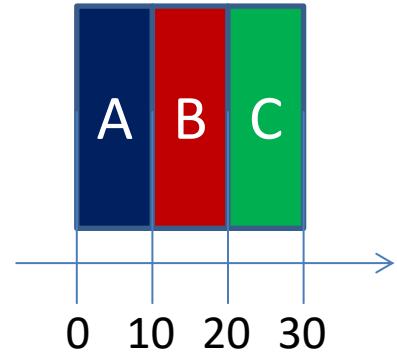
Aufgrund der Idealisierten Bedingungen ist dieser Wert offensichtlich unabhängig von der Ausführungsreihenfolge.



# Scheduling (FIFO)

## Performance-Betrachtung (Umlaufzeit):

Bei etwa gleichlanger Prozesslaufzeit und, falls die durchschnittliche Umlaufzeit das Maß aller Dinge ist, dann ist FIFO eine durchaus adäquate Strategie.





# Scheduling (FIFO)

**Was geschieht, wenn die Bearbeitungszeit variiert?**

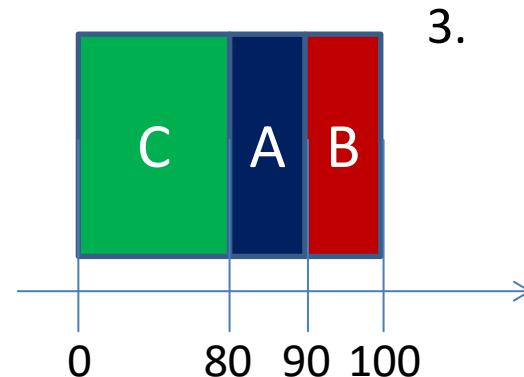
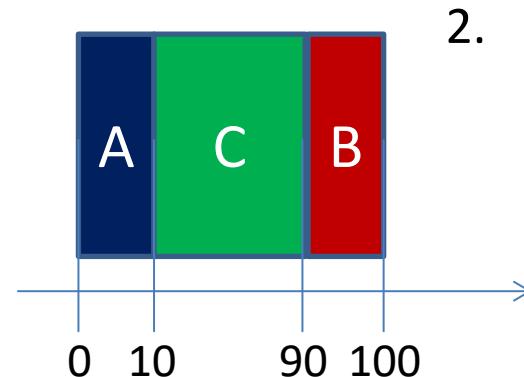
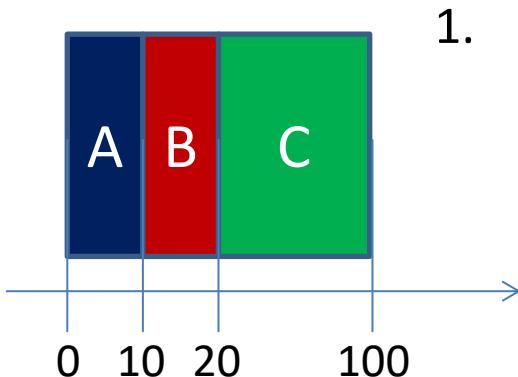
- Alle Prozesse (kurz Jobs oder Aufgaben) benötigen dieselbe Zeit.
- Alle Jobs stehen zur gleichen Zeit zur Ausführung bereit.
- Einmal angefangen, läuft jeder Job bis zu seinem Ende durch.
- Alle Jobs verwenden nur die CPU, d.h., es gibt keine Hardware-Zugriffe.
- Die Zeit, die ein Job braucht, ist bekannt.



# Scheduling (FIFO)

## Performance-Betrachtung (Umlaufzeit):

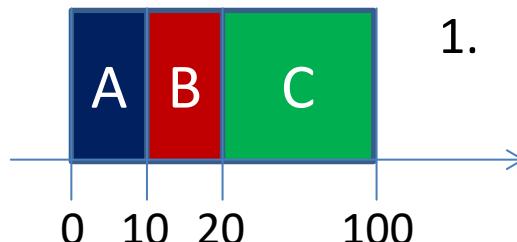
- Weiterhin stehen alle Jobs zur gleichen Zeit bereit => Reihenfolge ist zufällig
- Die Jobs dauern jedoch unterschiedlich lange. (Laufzeit A 10s, B 10s, C 80s)



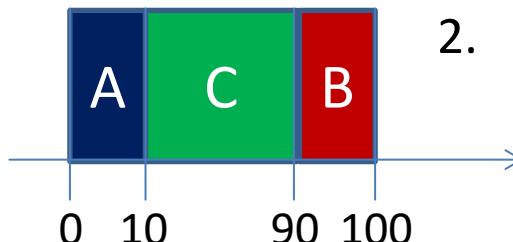


# Scheduling (FIFO)

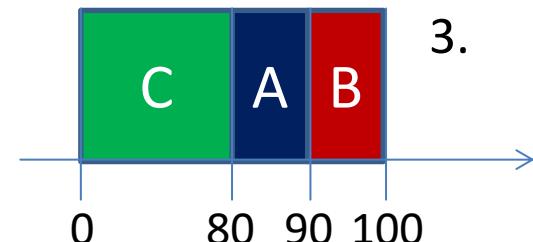
## Performance-Betrachtung (Umlaufzeit):



1.



2.



3.

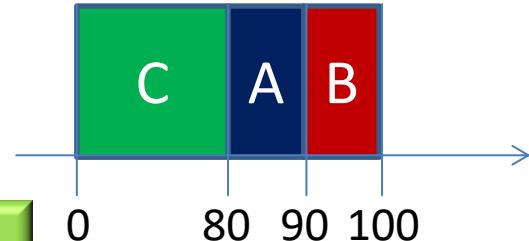
- Umlaufzeit (A): 10s – 0s = 10s
  - Umlaufzeit (B): 20s – 0s = 20s
  - Umlaufzeit (C): 100s – 0s = 100s
- $(10s + 20s + 100s) / 3 = 43,3s$
- Umlaufzeit (A): 10s – 0s = 10s
  - Umlaufzeit (B): 100s – 0s = 100s
  - Umlaufzeit (C): 90s – 0s = 90s
- $(10s + 100s + 90s) / 3 = 66,7s$
- Umlaufzeit (A): 90s – 0s = 90s
  - Umlaufzeit (B): 100s – 0s = 100s
  - Umlaufzeit (C): 80s – 0s = 80s
- $(90s + 100s + 80s) / 3 = 90s$



# Scheduling (FIFO)

Performance-Betrachtung (Umlaufzeit):

Problem:

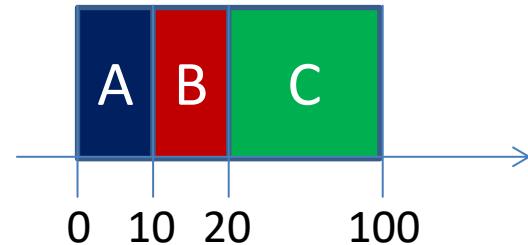


Die durchschnittliche Umlaufzeit steigt drastisch, wenn kurze Aufträge auf die Beendigung langer Aufträge warten müssen.



# Scheduling (Zweite Idee)

Die durchschnittliche Umlaufzeit ist minimal gdw., die Aufträge nach ihrer Dauer von kurz nach lang bearbeitet werden!



Die offensichtliche Idee:

- SJF (Shortest Job First)      „Die kürzeste Aufgabe kommt zuerst!“



# Scheduling (SJF)

**Was geschieht, wenn die Ankunftszeit der einzelnen Jobs variiert?**

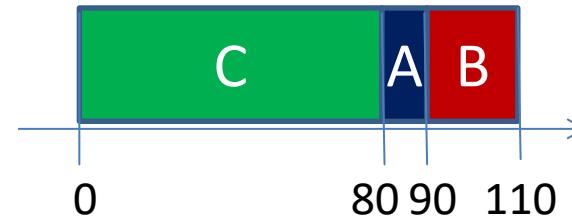
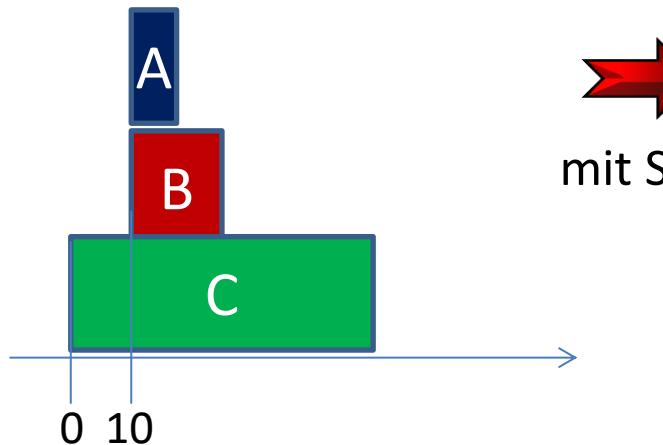
- Alle Jobs stehen zur gleichen Zeit zur Ausführung bereit.
- Einmal angefangen, läuft jeder Job bis zu seinem Ende durch.
- Alle Jobs verwenden nur die CPU, d.h., es gibt keine Hardware-Zugriffe.
- Die Zeit, die ein Job braucht, ist bekannt.



# Scheduling (SJF)

## Performance-Betrachtung (Umlaufzeit):

- Die einzelnen Jobs kommen zu beliebigen Zeiten an. (C vor B; A und B gleichzeitig)
- Die Jobs dauern unterschiedlich lange. (Laufzeit A 10s, B 20s, C 80s)



- **durchschnittliche Umlaufzeit**  
 $((90 - 10)s + (110 - 10)s + (80 - 0)s) / 3 = 86,7s$



# Scheduling (SJF)

Performance-Betrachtung (Umlaufzeit):

**SJF ist nur dann optimal, wenn die Aufgaben bereits vorliegen.**

doch!

**„Prognosen sind äußerst schwierig, vor allem, wenn sie die Zukunft betreffen.“**

[Mark Twain ?]



# Scheduling (SJF)

**Das Betriebssystem ist aber in der Lage, Prozesse anzuhalten und zu verdrängen! Könnte dies hilfreich sein?**

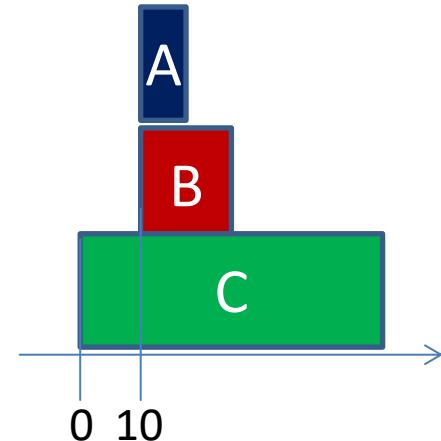
- 
- 
- Einmal angefangen, läuft jeder Job bis zu seinem Ende durch.
- Alle Jobs verwenden nur die CPU, d.h., es gibt keine Hardware-Zugriffe.
- Die Zeit, die ein Job braucht, ist bekannt.



# Scheduling (Dritte Idee)

„präemptiv“ (preemptive, vorsorglich)

Der Scheduler kann einen beliebigen Job unterbrechen, um mit einem anderen (ggf. neuen) fortzufahren.



Die offensichtliche Idee: SJF mit etwas „Vorsorge“

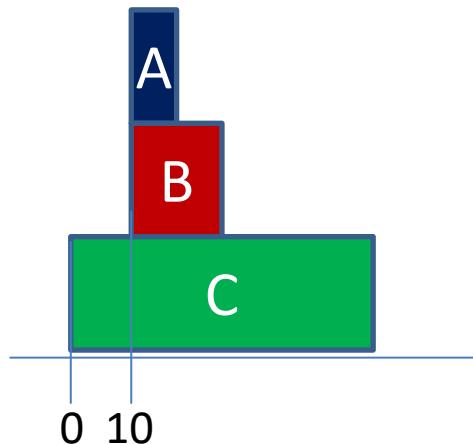
- STCF (Shortest Time to Completion First)  
„Die Aufgabe, die zuerst fertig wird, wird vorgezogen.“



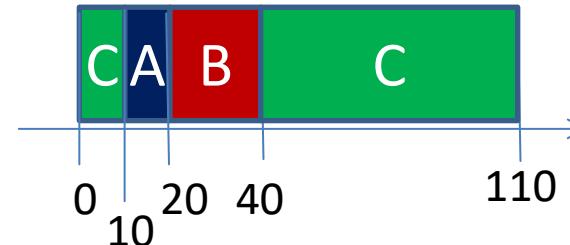
# Scheduling (STCF)

## Performance-Betrachtung (Umlaufzeit):

- Die einzelnen Jobs kommen zu beliebigen Zeiten an. (C vor B; A und B gleichzeitig)
- Die Jobs dauern unterschiedlich lange. (Laufzeit A 10s, B 20s, C 80s)



mit STCF



- **durchschnittliche Umlaufzeit**  
 $((20 - 10)s + (40 - 10)s + (110 - 0)s) / 3 = 50s$



# Scheduling (STCF)

## Performance-Betrachtung:

Ist die Dauer einer Aufgabe bekannt, dann ist die durchschnittliche Umlaufzeit minimal gdw. stets an der Aufgabe weitergearbeitet wird, die zuerst abgeschlossen werden kann.



# Scheduling (STCF)

## Performance-Betrachtung:

- Umlaufzeit ist nicht das Maß aller Dinge.
- Die wenigsten Programme arbeiten ohne Interaktion mit dem Anwender.



**Sowohl beim Handy als auch bei einer Workstation erwartet der Anwender stets kurze Reaktionszeiten.**

**Reaktionszeit (Response Time)**

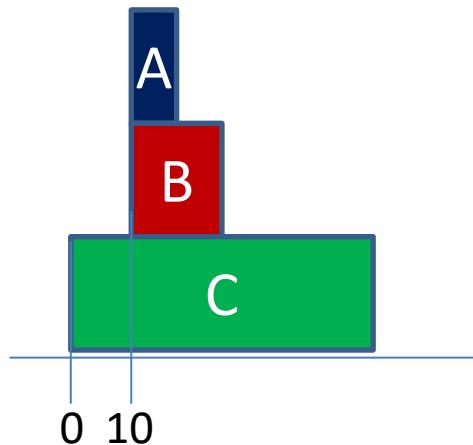
$$T_{\text{Reaktion}} = T_{\text{Start}} - T_{\text{Ankunft}}$$



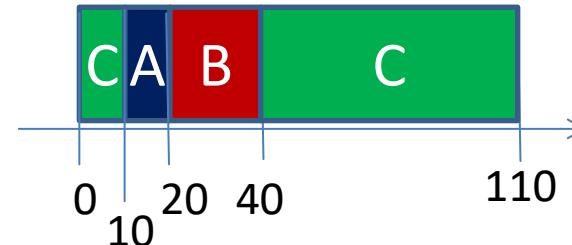
# Scheduling (STCF)

## Performance-Betrachtung (Reaktionszeit):

- Die einzelnen Jobs kommen zu beliebigen Zeiten an. (C vor B; A und B gleichzeitig)
- Die Jobs dauern unterschiedlich lange. (Laufzeit A 10s, B 20s, C 80s)



mit STCF

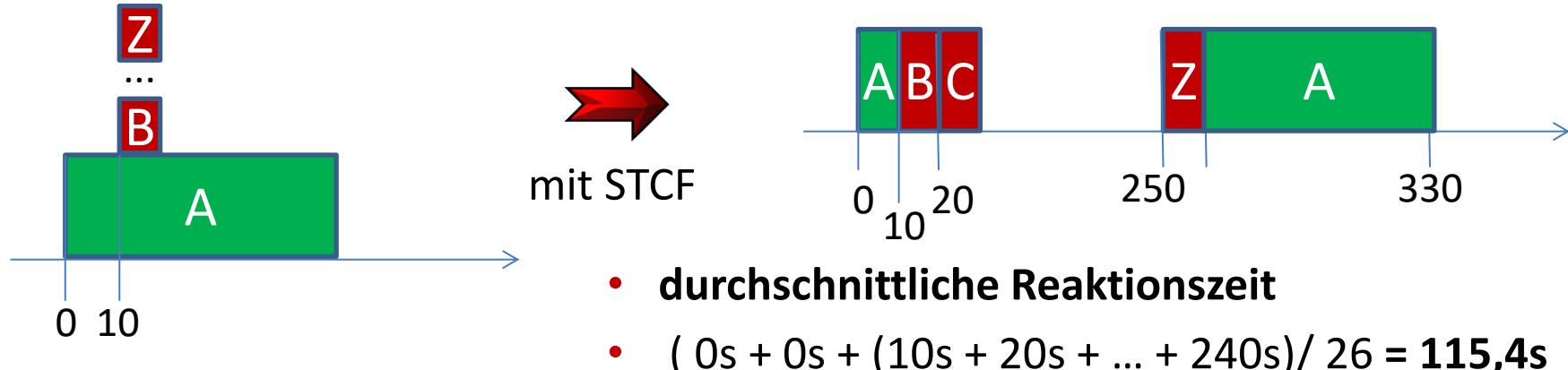


- **durchschnittliche Reaktionszeit**
- $(0s + (20 - 10)s + 0s) / 3 = 3,3s$



# Scheduling (STCF)

Performance-Betrachtung (Reaktionszeit):



**Die durchschnittliche Reaktionszeit steigt dramatisch,  
sobald mehrere Prozesse (Jobs) zu bearbeiten sind!**



# Scheduling (Round Robin)

**Warum sollte man das Bestreben haben, Dinge zu Ende zu bringen  
bevor man etwas Neues beginnt?  
Ab jetzt geschehen die Dinge „gleichzeitig“!**

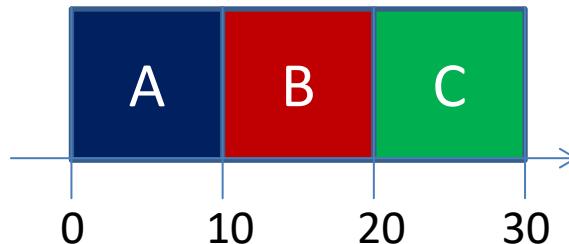
- Jeder Job bekommt eine Zeitscheibe zugewiesen.
- Die Zuweisung wiederholt sich periodisch.
- Je kleiner die Zeitscheiben, desto mehr entsteht der Eindruck von Gleichzeitigkeit.



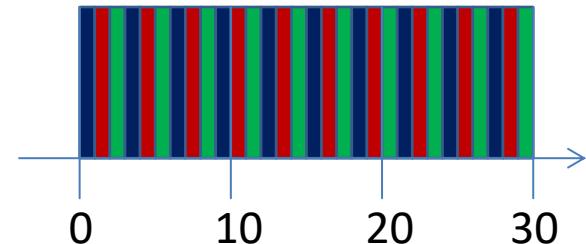
# Scheduling (RR)

## Performance-Betrachtung (Reaktionszeit):

- SJF
- STCF



- RR



- Reaktionszeit(A): 0s – 0s = 0s
- Reaktionszeit(B): 10s – 0s = 10s
- Reaktionszeit(C): 20s – 0s = 20s

$$(0s + 10s + 20s) / 3 = 10s$$

$$\begin{aligned} 0s - 0s &= 0s \\ 1s - 0s &= 1s \\ 2s - 0s &= 2s \end{aligned}$$

$$(0s + 1s + 2s) / 3 = 1s$$



# Scheduling (RR)

- Je kleiner die Zeitscheiben, desto kürzer die Reaktionszeit.
  - Je kleiner die Zeitscheiben, desto mehr entsteht der Eindruck von Gleichzeitigkeit.

## Achtung:

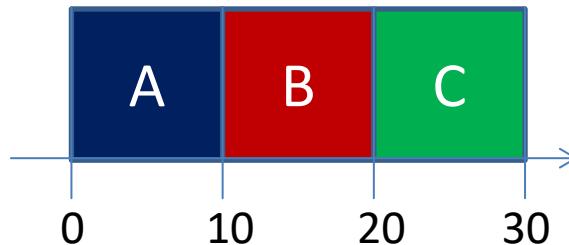
- Die Größe der Zeitscheiben ist begrenzt durch die Auflösung des HW-Timers ( $\text{HPET} \approx 1\text{ms}$ ).
- Auch der Kontextwechsel kostet Zeit ( $\approx 1\text{-}1000 \mu\text{s}$ ).



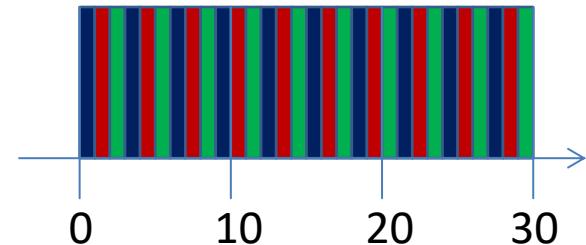
# Scheduling (RR)

## Performance-Betrachtung (Umlaufzeit):

- SJF
- STCF



- RR



- Umlaufzeit(A): 10s – 0s = 10s
- Umlaufzeit(B): 20s – 0s = 20s
- Umlaufzeit(C): 30s – 0s = 30s

$$(10s + 20s + 30s) / 3 = 20s$$

- 28s – 0s = 28s
- 29s – 0s = 29s
- 30s – 0s = 30s

$$(28s + 29s + 30s) / 3 = 29s$$



# Scheduling



**Was für die Reaktionszeit gut ist, ist für die Umlaufzeit schlecht und umgekehrt!**

- **Umlaufzeit**
  - SJF
  - STCF
- **Reaktionszeit**
  - RR



# Scheduling

**Etwas mehr Realität: Programme, die keine Ausgaben liefern und keine Eingaben nutzen, braucht man nicht!**

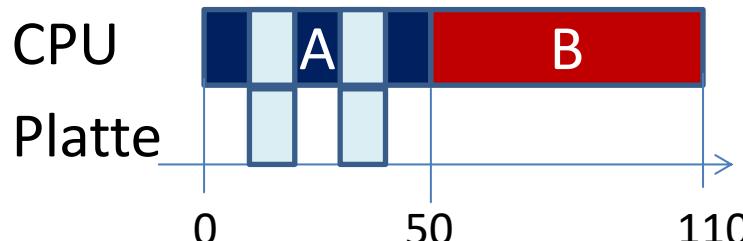
- 
- 
- 
- Alle Jobs verwenden nur die CPU, d.h., es gibt keine Hardware-Zugriffe.
- Die Zeit, die ein Job braucht, ist bekannt.



# Scheduling (SJF)

## Performance-Betrachtung (Ressourcenausnutzung):

- Zwei Jobs
  - A dauert 30ms (reine CPU-Zeit) plus zwei Plattenzugriffe à 10ms
  - B dauert 60ms (reine CPU-Zeit)



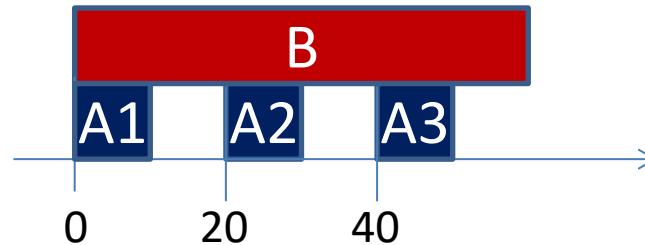
**Problem:**  
CPU-Zeit wird  
verschenkt!



# Scheduling ist Hardware-aware

## Performance-Betrachtung (Ressourcenausnutzung):

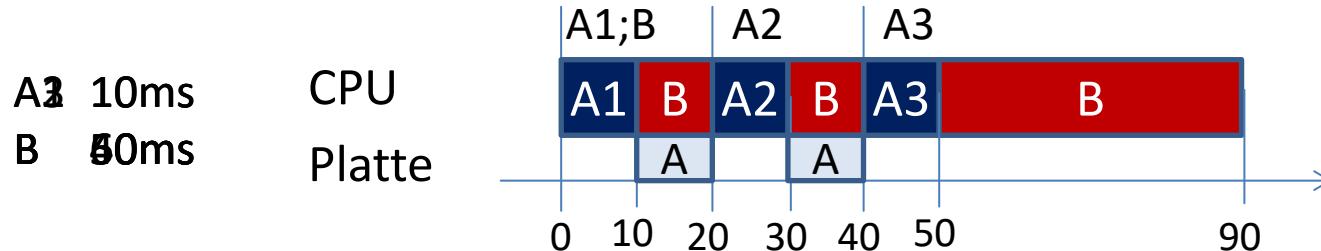
- Jede CPU-Nutzung wird als separater Job angesehen!
  - A zerfällt in 3 Jobs à 10ms, die zu unterschiedlichen Zeiten ankommen.  
A1 (0s), A2 10ms nach dem Ende von A1 und A3 10ms nach dem Ende von A2.
  - B dauert 60ms und kommt zusammen mit A1 an.





# Scheduling ist Hardware-aware

## Performance-Betrachtung (Ressourcenausnutzung):



- T 0: A1 und B kommen an; A1 ist am Zug (kürzer als B).
- T 10ms: A1 ist abgearbeitet; Plattenzugriff beginnt; B ist am Zug (kein weiterer Job)
- T 20ms: Plattenzugriff beendet; A2 kommt an und unterbricht B ( kürzer als der Rest von B)
- T 30ms: A2 ist abgearbeitet; Plattenzugriff beginnt; B wird fortgesetzt (kein weiterer Job)
- T 40ms: Plattenzugriff beendet; A3 kommt an und unterbricht B ( kürzer als der Rest von B)
- T 50ms: A3 ist abgearbeitet; B wird fortgesetzt (kein weiterer Job)



# Scheduling

**Ab jetzt wird es richtig schwierig. Wir verzichten auf unser  
a priori Wissen!**

- 
- 
- 
- 
- Die Zeit, die ein Job braucht, ist bekannt.



# Multi-Level Feedback Queue (MLFQ)

**Idee:** Wir nutzen das Wissen über die Vergangenheit, um Vorhersagen über die Zukunft zu treffen.

## Ziel:

- Minimierung der Umlaufzeit
- Minimierung der Reaktionszeit

**Das System soll schnell auf äußere Ereignisse reagieren und zügig die anstehenden Aufgaben bewältigen.**



# Multi-Level Feedback Queue (MLFQ)

## Schritt 1: Alle Jobs werden priorisiert.

hoch



niedrig

- Es gibt eine feste Anzahl von Queues.
- Jede Queue repräsentiert einen definierten Prioritätslevel.

### Regeln:

1.  $\forall A, B : \text{Run}(A) \Rightarrow \text{Prio}(A) \geq \text{Prio}(B)$
2.  $\forall A, B : \text{Run}(A) \wedge \text{Prio}(A) = \text{Prio}(B) \Rightarrow \diamond \text{Run}(B)$
3. Der Scheduler nutzt **Round Robin!**

Nur die Jobs mit der höchsten Priorität werden ausgeführt!



# Multi-Level Feedback Queue (MLFQ)

## Schritt 2: Prioritäten wechseln!



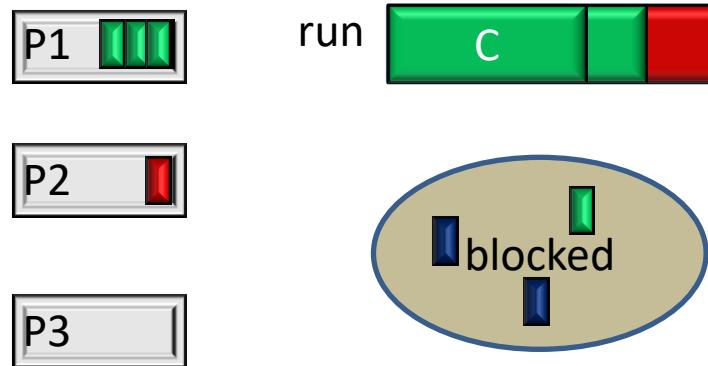
### Regeln:

4. Jeder neue Job beginnt auf der höchsten Prioritätsstufe.
- 5a. Jeder Job, der den gesamten Zeitschlitz nutzt, wird in der Priorität abgestuft.
- 5b. Jeder Job, der nicht den gesamten Zeitschlitz nutzt, behält seine Priorität.



# Multi-Level Feedback Queue (MLFQ)

- Drei Jobs **A**, **B** und **C** werden neu erzeugt.
  - **A** und **C** mit vielen Interaktionen
  - **B** rechenintensiv mit wenig Interaktionen





# Multi-Level Feedback Queue (MLFQ)

- Interaktive Prozesse verbrauchen den gesamten Zeitschlitz selten.
  - **Priorität bleibt hoch.  
Sie werden bevorzugt!**
- Rechenintensive Prozesse verbrauchen den gesamten Zeitschlitz oft.
  - **Priorität sinkt stetig. Sie kommen dann zum Zug, wenn alle anderen blockiert sind!**



# Multi-Level Feedback Queue (MLFQ)

## Problem:

- Bei vielen interaktiven Prozessen kommen die rechenintensiven nicht mehr zum Zug! (**Starvation**)
- Wer die Strategie kennt, kann mogeln!  
Ein Aufruf von `yield()` zur rechten Zeit wirkt Wunder!
- Einmal abgestuft immer abgestuft!



# Multi-Level Feedback Queue (MLFQ)

## Prioritäts-Boost

Regeln:

6. Alle Prozesse werden nach einer definierten Zeit S wieder in die höchste Queue eingereiht.



Auch rechenintensive Prozesse  
kommen wieder zum Zug.



# Multi-Level Feedback Queue (MLFQ)

**Buchhalten:**

**Regeln:**

**5.a** Jeder Job, der den gesamten Zeitschlitz nutzt, wird in der Priorität abgestuft.

**5.b** Jeder Job, der nicht den gesamten Zeitschlitz nutzt, behält seine Priorität.

**Werden ersetzt!**



# Multi-Level Feedback Queue (MLFQ)

**Buchhalten:**

**Regel:**

5. Sobald ein Job sein Zeitkontingent verbraucht hat, wird er abgestuft.



**Mogeln hilft nicht mehr. Wer zu viel Rechenzeit braucht, muss auch mal warten!**



# Multi-Level Feedback Queue (MLFQ)

## Regel:

1.  $\forall A, B : \text{Run}(A) \Rightarrow \text{Prio}(A) \geq \text{Prio}(B)$
2.  $\forall A, B : \text{Run}(A) \wedge \text{Prio}(A) = \text{Prio}(B) \Rightarrow \diamond \text{Run}(B)$
3. Der Scheduler nutzt **Round Robin!**
4. Jeder neue Job beginnt auf der höchste Prioritätsstufe.
5. Sobald ein Job sein Zeitkontingent verbraucht hat, wird er abgestuft.
6. Alle Prozesse werden nach einer definierten Zeit S wieder in die höchste Queue eingereiht.



# Lotterie-Scheduling

## Betrachtung (Fairness):

- Die Zuteilung eines Zeitschlitzes erfolgt über Lose.
- Wer das richtige Los hat, der gewinnt.
- Eine höhere Priorität bedeutet mehr Lose.

**einfach, simpel und fair!**



# Lotterie-Scheduling

```
public class Lottery<JOB> {  
  
    private Queue<AssessedJOB> queue = new LinkedList<>(); ← ausführbare Jobs  
    private int currentTickets = 0; ← Anzahl aktueller Lose  
    private Random randomGenerator = new Random(); ← Zufallszahlengenerator für die Lotterie  
  
    public synchronized void addJob(JOB job, int tickets){  
        if (tickets <= 0) throw new IllegalArgumentException("no tickets");  
        this.currentTickets += tickets;  
        this.queue.add(new AssessedJOB(job,tickets));  
    }  
  
    private class AssessedJOB{ ← Jobs mit Losen  
        private final int tickets;  
        private final JOB job;  
        private AssessedJOB(JOB job, int tickets){  
            this.tickets = tickets;  
            this.job = job;  
        }  
    }  
}
```

- ausführbare Jobs
- Anzahl aktueller Lose
- Zufallszahlengenerator für die Lotterie
- **Jobs mit Losen**



# Lotterie-Scheduling (die Lotterie)

```
public synchronized JOB next2run(){
    if (this.queue.size() == 0) return null;
    int ticket = randomGenerator.nextInt(currentTickets);
    Iterator<AssessedJOB> iter = this.queue.iterator();

    while(iter.hasNext()){
        AssessedJOB assessedJob = iter.next();
        if (assessedJob.tickets > ticket){
            iter.remove();
            this.currentTicket -= assessedJob.tickets;
            return assessedJob.job;
        }
        ticket -= assessedJob.tickets;
    }
    return null;
}
```

- gezogenes Los
- Gewinner ermitteln
- Gewinner entfernen und ausgeben



# Lotterie-Scheduling (wir wollten C!)

```
job_t* lottery_next2run(lottery_t* lottery) {
    if (lottery->queue == NULL) return NULL;
    ready_queue_t *iter = lottery->queue;
    job_t *job = NULL;
    int ticket = rand() % lottery->tickets;
    while (iter) {
        if (iter->tickets > ticket) {
            job = iter->job;
            lottery->tickets -= iter->tickets;
            lottery->queue = unlink(lottery->queue, iter);
            free(iter);
            return job;
        }
        ticket -= iter->tickets;
        iter = iter->next;
    }
    return NULL;
}
```

```
typedef struct lottery_t {
    ready_queue_t *queue;
    uint32_t tickets;
} lottery_t;
```

```
typedef struct ready_queue_t {
    job_t *job;
    uint32_t tickets;
    struct ready_queue_t *next;
    struct ready_queue_t *pred;
} ready_queue_t;
```



# Lotterie-Scheduling

- Ein echtes Problem:



**echter Zufall**



# Scheduling (Multicore-Prozessoren)

## Neue Fragen und Probleme!

- Zuteilung hat mehrere Dimensionen (Zeitschlitz u. Prozessor)
- Die Verlagerung eines Jobs von einem Kern zum nächsten ist sehr aufwendig.
- Prozessoren sollten gleichmäßig ausgelastet sein.
- ...

**Ziel: n-Prozessoren verhalten sich bei vielen Jobs so, wie ein Prozessor, der n-mal so schnell ist!**



# Scheduling (Multicore-Prozessoren)

**Lösungsideen!**

- Mehrere Queues (eine pro CPU)
- Job-Migration, CPUs die unterfordert sind, klauen Jobs bei anderen.
- ...

**Wichtig: Cache-Belegung beachten!**



# Teil II

# Persistenz, Dateisysteme und I/O



# Persistenz, Dateisysteme und I/O

- Übersicht
  - Ein generisches I/O-Device
  - Festplatten
  - (Raid-Systeme)
- File-Systeme
  - Dateien und Verzeichnisse
  - Ein simples File-System
  - ...



# Zentrale Gedanken

## Ein generisches I/O-Device



# I/O-Device

**Ohne Peripherie (Ein-/Ausgabegeräte) nutzt der beste Rechner nichts!**

- Daten muss man eingeben können (**Funktionen benötigen Parameter**)
  - Tastatur / Maus,
  - Kamera,
  - Festplatten, Memory-Sticks, ...
- Daten muss man ausgeben können (**Funktionen liefern Ergebnisse**)
  - Monitor, Drucker,
  - Lautsprecher,
  - Festplatten, Memory-Sticks, ...



# I/O-Device

**Wie kommen Daten vom Device zum Programm  
und vom Programm zum Device?**

## 1. Hardware:

Die Anbindung erfolgt über diverse Schnittstellen und Bussysteme

## 2. Software:

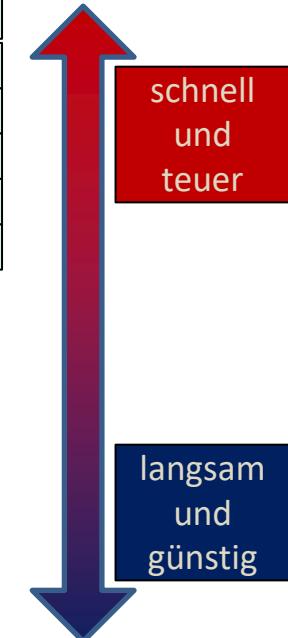
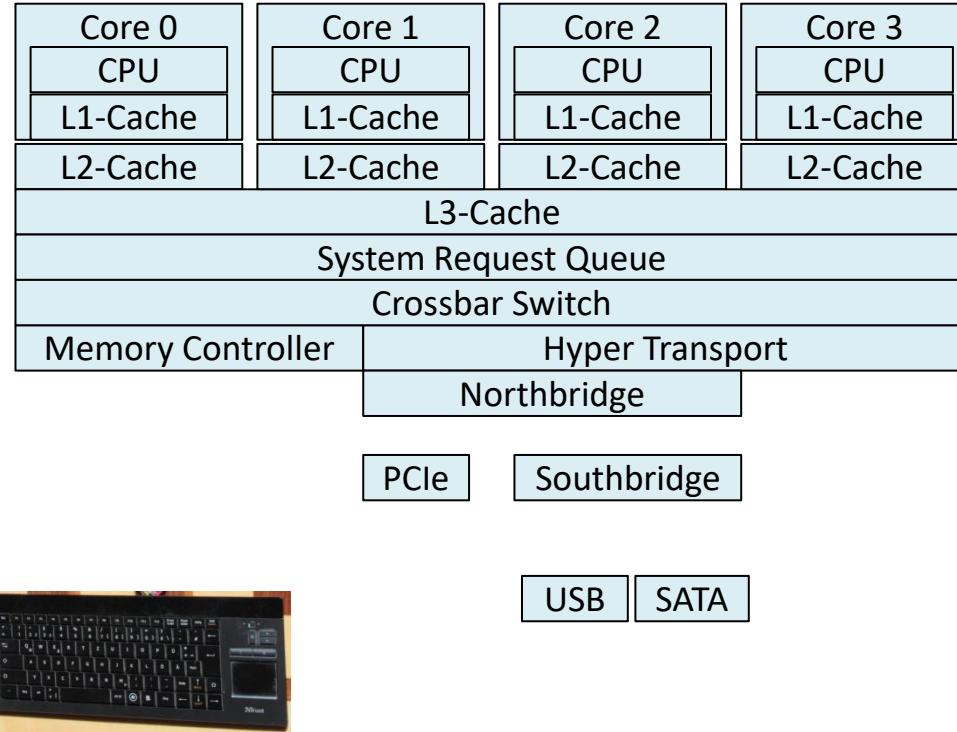
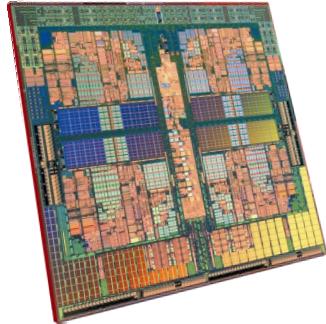
Das Betriebssystem managet den Zugriff!





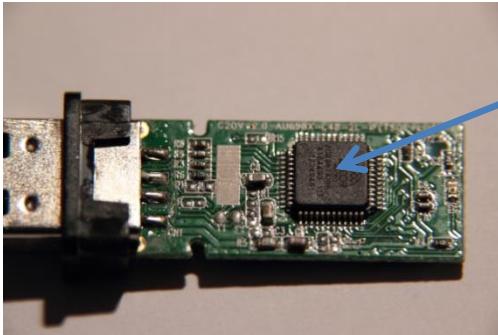
# Etwas Rechnerarchitektur

Die eines AMD Phenom™ Quad-Core  
[Bild: Advanced Micro Devices, Inc. (AMD)]

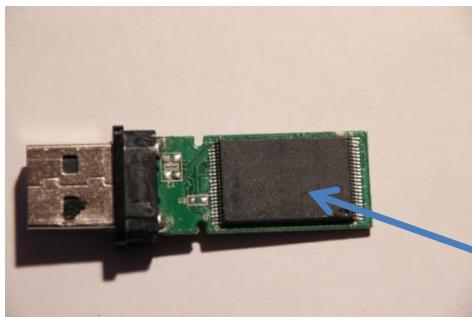




# Ein Peripheriegerät

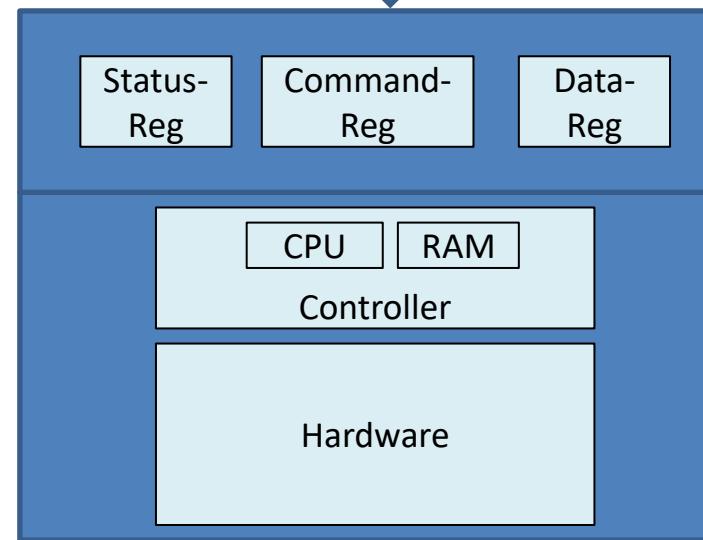


Controller  
CPU+RAM



Flash-Speicher

Treiber

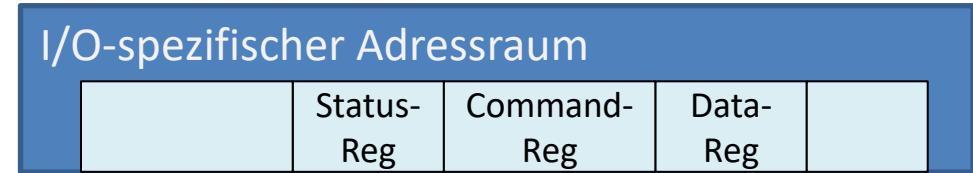
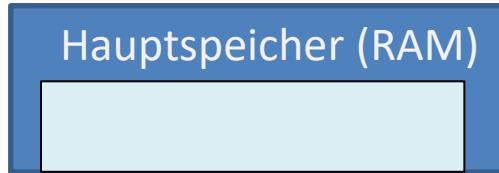


Interface  
(Hardware-  
Abstraktion)

Gerät



# Ein Peripheriegerät (die Schnittstelle)

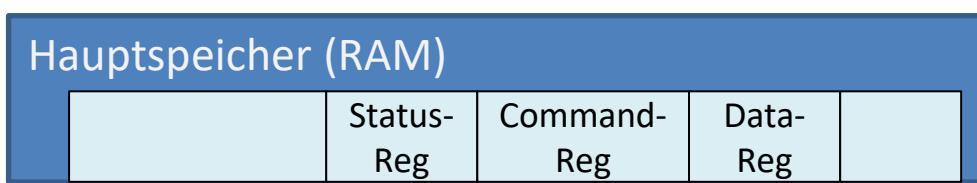


- Port-Mapped I/O:  
Hardware-Register (Ports) liegen in einem separaten Adressraum, der über spezifische I/O-Befehle angesprochen wird. Physisch docken hier die externen Bussysteme an.

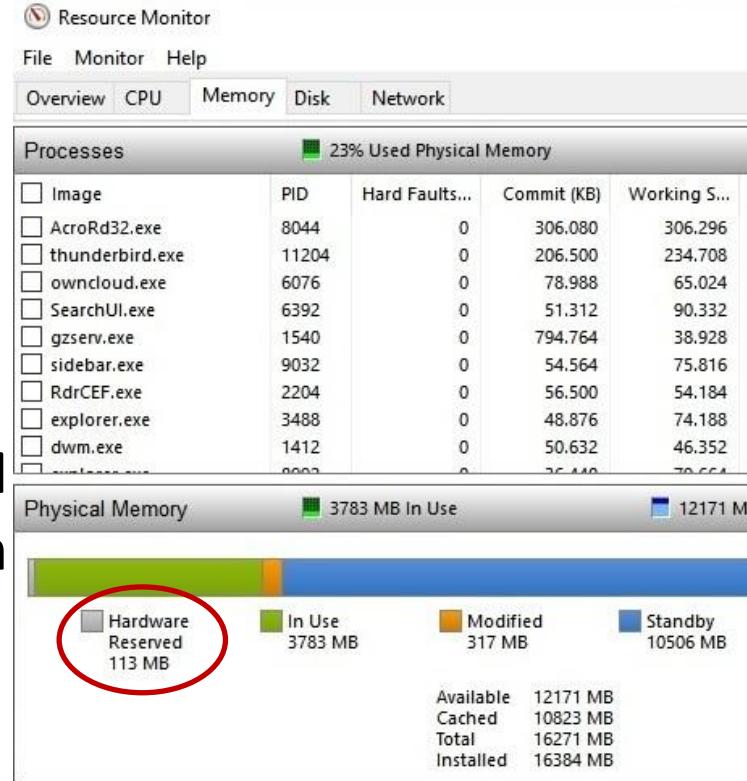
**Dies war früher relevant, als der Adressraum noch „klein“ war (16 Bit bzw. 32 Bit)**



# Ein Peripheriegerät (die Schnittstelle)



- Memory-Mapped I/O:  
Hardware-Register stellen aus Sicht des Betriebssystems Speicherbereiche im RAM dar und können mit den üblichen Befehlen angesprochen werden. (Ein Teil des RAM bleibt reserviert.)





# Ein erstes Protokoll

Hauptspeicher (RAM)

	Status-Reg	Command-Reg	Data-Reg	
--	------------	-------------	----------	--

```
while (STATUS != READY)
    wait for interrupt; // wait until device is ready
write data to DATA register
write command to COMMAND register // this starts the data transfer
                                    // to the physical device and
                                    // executes the command
while (STATUS == BUSY)
    wait for interrupt; // wait until device is done
```

Aufgabe von Hardware und Bussystemen.  
(Eine andere Vorlesung)



# Performance

Interrupts und damit verbundene Prozesswechsel benötigen Zeit.  
Eventuell mehr als das aktive Warten auf ein schnelles Device.



Ein bisschen  
Spinning und  
dann (yield)  
wait;

```
int i = a_little;
while (STATUS != READY)
    if (--i == 0){ i = a_little;
        wait for interrupt; // wait until device is ready
    };
write data to DATA register
write command to COMMAND register // this starts the data transfer
                                    // and executes the command
while (STATUS == BUSY)
    if (--i == 0){i = a_little;
        wait for interrupt; // wait until device is done
    };
```



# Performance

**Nicht nur schnelle Devices verursachen Probleme!**

Auch ein hoher Netzwerk-Traffic kann massiv viele Interrupts generieren und das System lahmlegen.



**Nicht immer auf Interrupts reagieren:**

- Interrupts ggf. ignorieren
- Interrupts ggf. sammeln



# Performance

## Wie erfolgt die Übergabe vieler Daten?

Hauptspeicher (RAM)

process managed (x)

kernel managed (y)

Status-  
Reg

Command-  
Reg

Data-  
Reg

```
char x[];  
FILE *f;  
...  
printf(f, "%s", x);  
...
```

```
...  
int len = strlen(x);  
memcpy(x,y,len);  
while (len > 0){  
    ...  
    len -= DATA_REG_SIZE;  
}
```

```
while (STATUS != READY)  
    if (--i == 0){ i = a_little;  
        wait for interrupt; };  
write data from y to DATA register  
write command to COMMAND register  
while (STATUS == BUSY)  
    if (--i == 0){i = a_little;  
    wait for interrupt; };
```



# Performance

Wie erfolgt die Übergabe vieler Daten?

Hauptspeicher (RAM)

process managed (x)

kernel managed (y)

Status-  
Reg

Command-  
Reg

Data-  
Reg

CPU



Platte

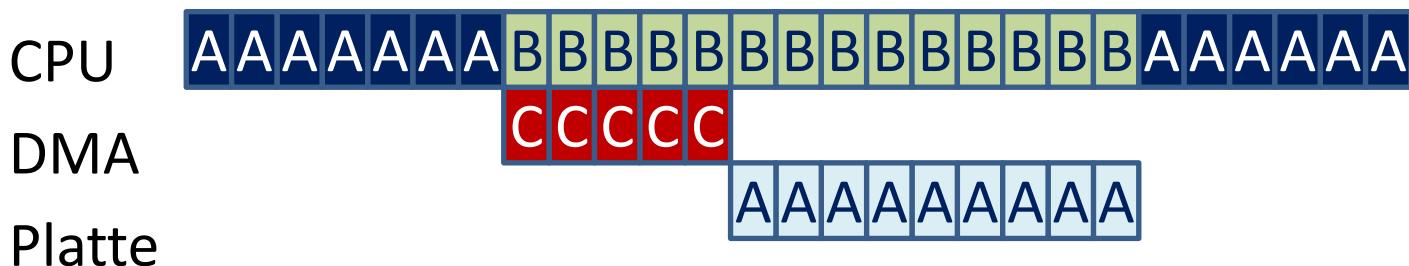


Aufwand für das Kopieren der Daten und deren Transport zum Device. (Kernel managed)



# Performance

Anstelle eines durch das Betriebssystem direkt kontrollierten Datentransfers (Programmed I/O kurz PIO) übernimmt spezielle Hardware, die DMA (Direct Memory Access), diese Aufgabe.



Das Betriebssystem programmiert lediglich den DMA-Controller (wie viel Daten von wo nach wo) setzt das Command ab und erwartet danach den Interrupt.



# Beispiel

## Lesen von Daten aus einem Peripheriegerät (z.B. Festplatte)

1. Das Betriebssystem sendet das Lese-Kommando an den Platten-Controller (damit ist die Aufgabe des Betriebssystems vorerst erledigt)
2. Der Platten-Controller initiiert den DMA-Transfer.
3. Der DMA-Controller überträgt die einzelnen Bytes.
4. Wurden alle Daten übertragen, setzt der DMA-Controller einen Interrupt ab.
5. Das OS ist wieder an der Reihe und kopiert die Daten an die gewünschte Stelle.

Wichtig: Der DMA-Controller muss entsprechend programmiert sein!

- Ziel- und Quelladressen sind definiert,
- Übertragungsgrößen festgelegt,
- Interrupts vereinbart.



# Gerätetreiber (Device Driver)

Peripheriegeräte gibt es wie Sand am Meer und jedes verfügt über mehr oder weniger individuelle Fähigkeiten.

Alle Varianten in ein Betriebssystem abzubilden ist unmöglich.

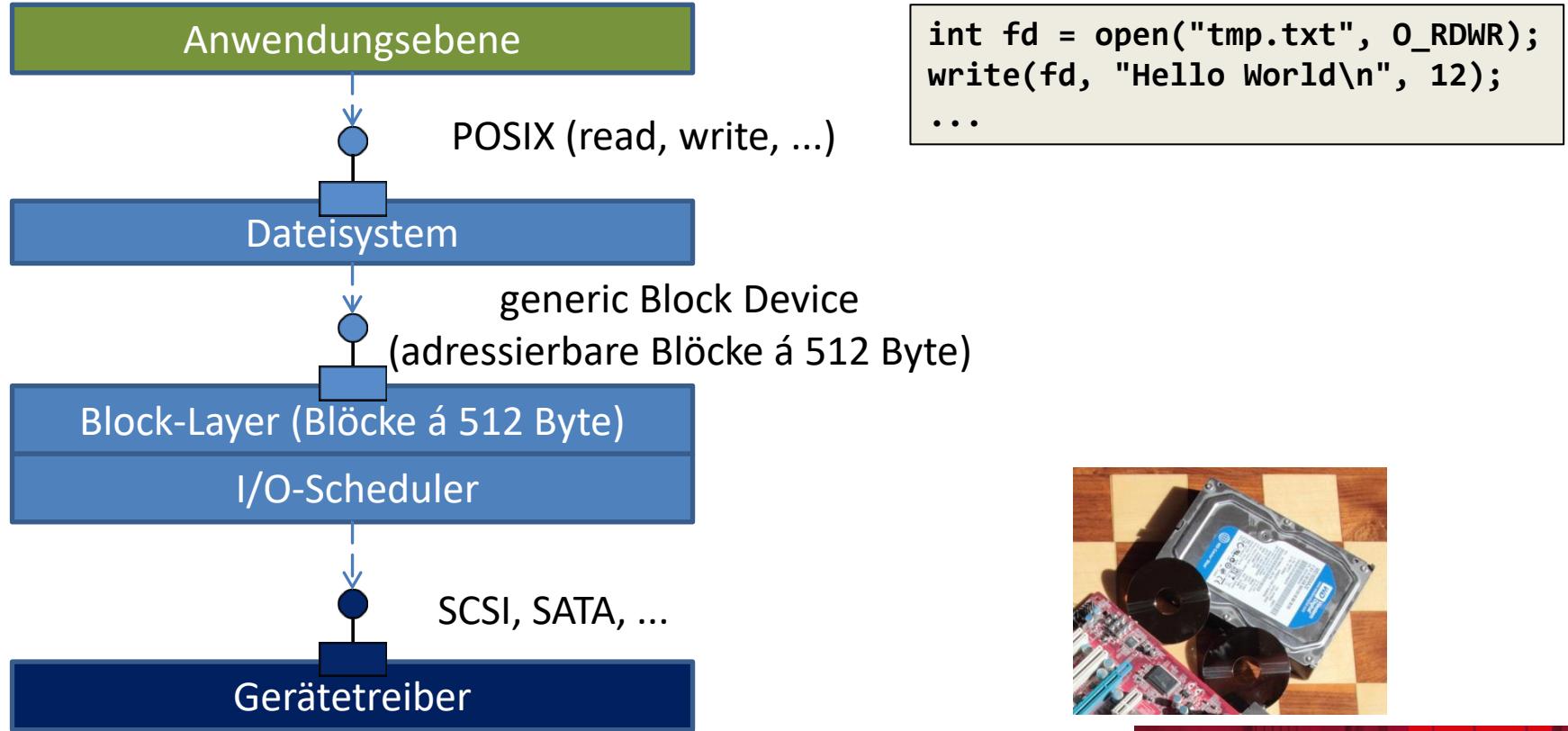
gerätespezifische Software (Treiber)

definierte Schnittstellen für das Betriebssystem, um Treiber anzusprechen

definierte Schnittstellen für Anwendungen, um via OS das Gerät zu nutzen



# Beispiel: File-Stack





# Block-Layer

**Der Block-Layer kapselt den Zugriff auf alle Block-Devices und bietet hierfür eine einheitliche Schnittstelle.  
(Jeder logische Block besteht aus 512 Byte)**

- Stellt Schreib- und Lese-Puffer bereit
  - Mapping in den Adressraum des Betriebssystems
  - Mapping in den Adressraum der Anwendung (falls gewünscht)
- Verwaltet logische Volumen  
=> unterschiedliche reale Plattenpartitionen können zu einer logischen zusammengefasst werden
- ...



# Festplatten



# Interface

**Festplatten sind in Sektoren aufgeteilt.**

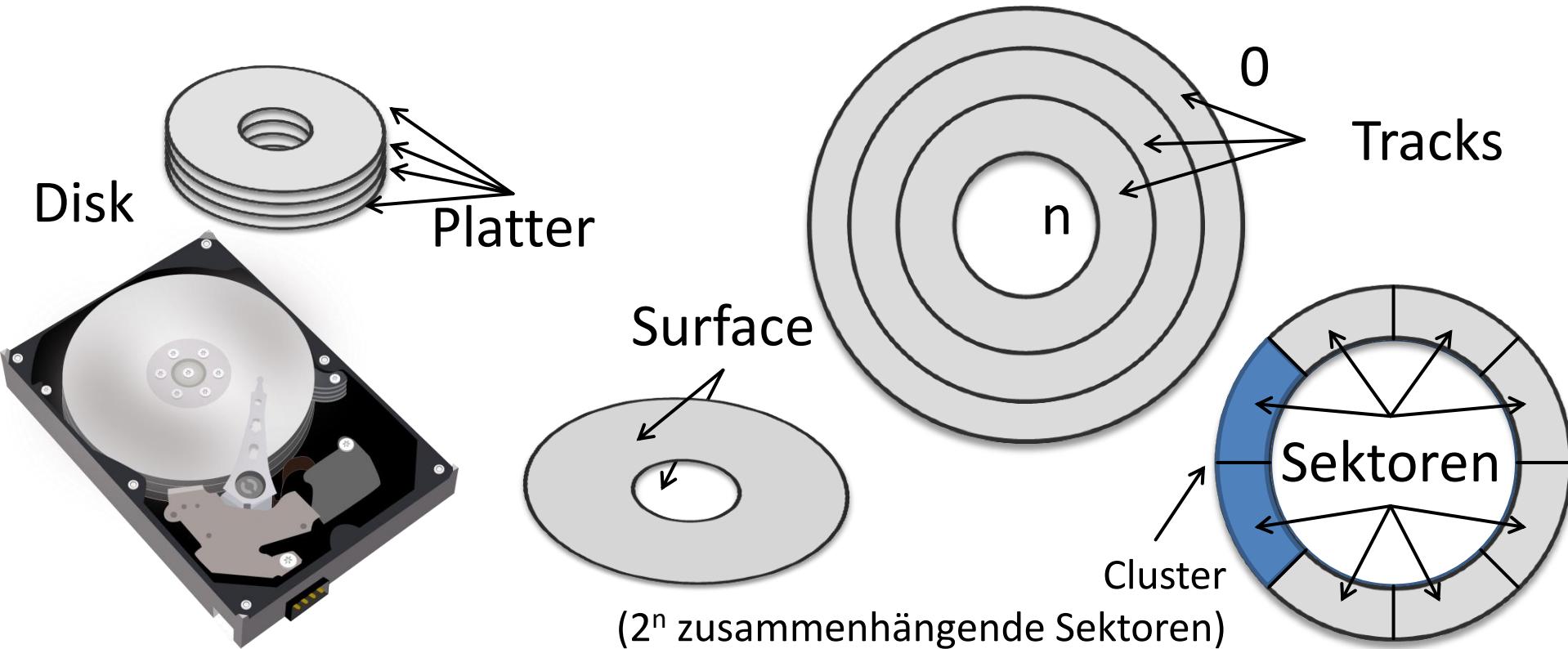
- Sektoren besitzen eine definierte Größe (512 Byte)  
(seit 2014 gibt es Platten mit 4096 Byte, unterstützt ab Windows 8 und Linux 2.6.36)
- Sektoren werden durchnummeriert (0 – n)
- Sektoren kann man atomar lesen und schreiben
- typischerweise wird auf mehrere (z.B.: 8) Sektoren gleichzeitig zugegriffen



**Festplatten sind Arrays von Sektoren.**



# Hardware

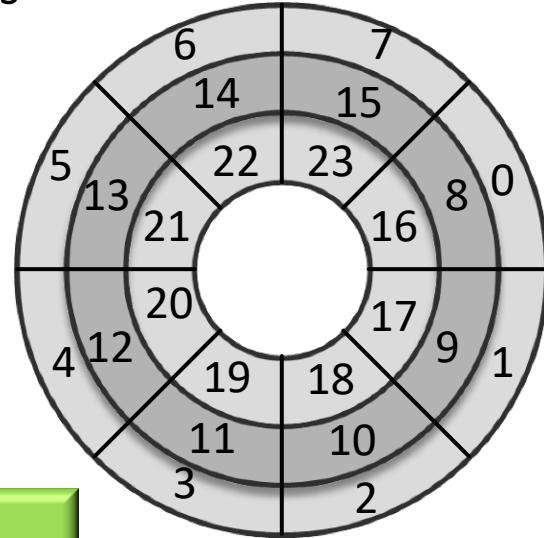
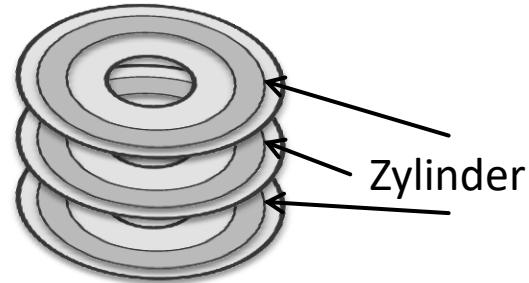




# Hardware



übereinanderliegende Tracks  
bilden einen Zylinder



**Offensichtlich kann auf Sektoren, die nahe beieinanderliegen oder sich im gleichen Zylinder befinden, schneller zugegriffen werden, als auf Sektoren die weit von einander entfernt sind!**



# Hardware: Lesen/Schreiben eines Sektors



1. Schreib/Lese-Kopf über dem richtigen Track positionieren (**Seek**)
2. Warten bis aufgrund der Rotation der Sektor sich unter dem Kopf positioniert hat (**Rotation**)
3. Daten übertragen (**Transfer**)

$$T_{I/O} = T_{\text{Seek}} + T_{\text{Rotation}} + T_{\text{Transfer}}$$



# Hardware: Seek-Time



- Zeit zum Anfahren (etwa 0,5 ms)
- + Zeit für den Weg zum richtigen Track?
- + Zeit zum Anhalten und Justieren  
(etwa 0,5 ms – 1ms)

$$T_{seek} = T_{acc} + T_{move} + T_{settle}$$

Oder wie viele Tracks müssen übersprungen werden?

**Faustregel:**  $\bar{T}_{move} \approx 1/3 T_{move\_max}$  (**Über die ganze Breite**)



# Hardware: Rotationsverzögerung



Durchschnittliche  
Rotationsverzögerung (Latenzzeit)  
beträgt bei n Sektoren pro Track

$$\bar{T}_{Rotation} = \frac{T_{1Rot}}{n} \left( \frac{1}{n} \sum_{i=1}^{n-1} i \right) = \frac{T_{1Rot}}{n} \left( \frac{1}{n} \cdot \frac{(n-1) \cdot n}{2} \right) = \frac{T_{1Rot}}{n} \cdot \frac{n-1}{2} \approx \frac{T_{1Rot}}{2}$$



# Hardware: Einige Größen



## Seagate Cheetah® 15K.7 High-End-Platten (300-600 GB)

- Seektime in ms (R/W)
  - Durchschnitt 3,4/3,9
  - single track 0,2/0,44
  - full stroke 6,6/7,4
- Latenzzeit 2ms
- Transferraten
  - intern 1,49 – 2,37 Gbits/s
  - maximal 204 MB/s
- Geometrie
  - Platters 4
  - Tracks pro Zoll 165000

Seagate: Cheetah 15K.7  
SAS Product Manual 100516226, Rev. H  
August 2015  
(<http://www.seagate.com/files/staticfiles/support/docs/manual/enterprise/cheetah/15K.7/SAS/100516226h.pdf>)



# Hardware: Einige Größen



## Seagate Savvio 15K v6 High-End-Platten (300-900 GB)

- Latenzzeit 2ms
- Transferrate maximal 315 MB/s
- Geometrie
  - Platters 3
  - Tracks pro Surface 138800
  - Tracks pro Zoll 335000
  - MBytes pro Surface 150.000
  - KBytes pro Track 1081
  - Bytes pro Sektor  
4k Mode 4160/4224

Seagate: Enterprise Performance 15K HDD v6 SAS  
Product Manual 100802255 - Rev. A September 2016  
(<http://www.seagate.com/files/www-content/product-content/enterprise-performance-savvio-fam/enterprise-performance-15k-hdd/ent-perf-15k-6/en-us/docs/100802255a.pdf>)



# Hardware: Einige Größen



## Seagate Barracuda 7200.11 Serial ATA (1,5 TB)

- Seektime in ms (R/W)
  - Durchschnitt 8,5/10,0
  - single track 0,8/1,0
- Latenzzeit 4ms
- Transferraten
  - intern 1,7 Gbits/s
  - maximal 135 MB/s
- Geometrie
  - Platters 4
  - Bytes pro Sektor 512
  - Sektoren pro Track 63
  - Zylinder 16383
  - Tracks pro Zoll 190000

Seagate: Product Manual Barracuda  
7200.11 Serial ATA 100507013 Rev. E  
December 2008  
(<http://www.seagate.com/files/staticfiles/support/disc/manuals/desktop/Barracuda%207200.11/100507013e.pdf>)



# Performance

$$T_{I/O} = T_{\text{Seek}} + T_{\text{Rotation}} + T_{\text{Transfer}}$$

- Seek langsam:  $\varnothing \approx 3 - 10 \text{ ms}$
- Rotation langsam:  $\varnothing \approx 2 - 5 \text{ ms}$
- Transfer richtig schnell:  $\varnothing \approx 100 - 300 \text{ MB/s} \Rightarrow 512 \text{ Byte in } 2 - 5 \mu\text{s}$

 Ein sequentieller Zugriff wäre optimal,  
zufälliges, verteiltes Zugreifen katastrophal.



# Performance

# Wie teuer ist es, 16KiB zufällig zu lesen?

- Consumer-Platte
    - Seek 8ms
    - Latenz 4ms
    - Transfer 100MB/s => 16KiB in 164µs
  - High-End-Platte
    - Seek 4ms
    - Latenz 2ms
    - Transfer 300MB/s => 16KiB in 55µs



12,16ms

6,05ms



# Performance: Durchsatz (Random-Read)

- Consumer-Platte (12,16ms)

$$\text{Durchsatz} = \frac{16KiB}{12,16ms} \times \frac{1MiB}{1024KiB} \times \frac{1000ms}{1s} \approx 1,3MiB/s$$

- High-End-Platte(6,05ms)

$$\text{Durchsatz} = \frac{16KiB}{6,05ms} \times \frac{1MiB}{1024KiB} \times \frac{1000ms}{1s} \approx 2,6MiB/s$$



# Performance: Durchsatz

- Consumer-Platte
  - Sequentiell: 100MB/s
  - Random: 1,3MB/s
- High-End-Platte
  - Sequentiell: 300MB/s
  - Random: 2,7MB/s



**Optimierung notwendig!**



**Ziel: Daten möglichst sequentiell lesen und schreiben!**



# Caching / Prefetching

**Um den Zugriff zu verbessern benötigt man zusätzlichen Speicher.**

Seagate: Enterprise Performance 15K HDD v6 SAS  
Product Manual 100802255 - Rev. A September 2016

- **Caching (lesen):**

Wenn der Cache aktiviert ist, werden die im Rahmen eines Lesebefehls angeforderten Daten möglichst aus dem Cache abgerufen, bevor ein Plattenzugriff initiiert wird. Falls nicht alle angeforderten Blöcke im Cache liegen, werden nur die fehlenden von der Platte geladen und in den Cache transferiert, so dass die Anfrage beantwortet werden kann. Bei deaktiviertem Cache wird immer auf die Platte zugegriffen, der Cache dient lediglich als Transportmedium.

Seagate Savvio 15K v6  
84MB volatile Cache (R/W)  
8MB non volatile Cache (W)



# Caching / Prefetching

- Caching (schreiben):

Alle zu schreibenden Daten landen zuerst im Cache und sind sofort wieder lesend zugreifbar.

- Es wird geprüft, ob vom Schreibvorgang aktuelle Daten des Caches betroffen sind. Diese werden gelöscht.
- Sobald die Daten im Cache sind, gelten sie logisch als geschrieben.
- **Die Firmware der Platte entscheidet letztlich wann genau die Daten auf die Platte geschrieben werden.**  
**(Durch einen expliziten „Sync“ kann dies jedoch erzwungen werden.)**



# Caching / Prefetching

- Caching (schreiben):

**Achtung: Stromausfall führt zum Datenverlust!**



**Hybride-Platten schützen vor diesem  
Datenverlust!**

“Hybrid 15K provides NVC-protected write caching over the portion of the DRAM used to coalesce writes. Write data only goes into NVC when there is an unexpected power loss to the drive. The NVC has 90-day data retention.”

Seagate: Enterprise Performance 15K HDD v6 SAS  
Product Manual 100802255 - Rev. A September 2016



# Caching / Prefetching

- Prefetching :  
Werden im Rahmen eines Datenzugriffes einzelne Segmente gelesen, dann werden typischerweise alle Segmente des Zylinders eingelesen (Read-Ahead).

**Um nicht unnötig viel Cache-Speicher zu verbrauchen,  
sind intelligente Algorithmen gefragt.**

“When prefetch (read look-ahead) is enabled, the drive enables prefetch of contiguous blocks from the disk when it senses that a prefetch hit will likely occur. The drive disables prefetch when it decides that a prefetch hit is not likely to occur.”

Seagate: Enterprise Performance 15K HDD v6 SAS  
Product Manual 100802255 - Rev. A September 2016



# I/O-Scheduling



# I/O-Scheduling

**In welcher Reihenfolge sollen die Aufträge abgearbeitet werden?**

- **Ziele:**
  - Umlaufzeit minimieren
  - Reaktionszeiten minimieren
  - Durchsatz maximieren
  - ...



# I/O-Scheduling



**Die relative Position der Daten ist wichtiger  
als die Größe der Daten!**



# Scheduling (Erste Idee)

Oftmals ist die simpelste Idee die  
beste Idee!

Die naheliegendste Idee:

- FIFO: First In, First Out
- FCFS: First Come, First Serve

„Wer zuerst kommt, mahlt zuerst!“

[Repgow, et.al:Sassen Speyghel, 1234]

Aufgaben werden in der Reihenfolge bearbeitet,  
in der sie ankommen.



# Scheduling (FCFS)

## Performance-Betrachtung (Umlaufzeit):

- Wir gehen wieder davon aus, dass alle Anfragen zur gleichen Zeit ankommen  
=> die Reihenfolge ist zufällig.
- Zu jeder Anfrage ist Surface, Zylinder und Sektor bekannt.

**0x083FFF3F**

- 0x3F => Sektor 63
- 0xFFFF => Zylinder 16383
- 0x08 => Platter 4 Seite 2

- Seek-Zeit plus Latenzzeit beträgt etwa 10ms bei einem Zufallszugriff.
- Die Transferzeit beträgt 5µs (vernachlässigbar)



# Scheduling (FCFS)

## Performance-Betrachtung (Umlaufzeit):

- Anfrage A    **0x083FFF3F => Zugriffszeit 10ms    (initialer Seek)**
- Anfrage B    **0x0814443F => Zugriffszeit 10ms    (neuer Seek)**
- Anfrage C    **0x083FFF3E => Zugriffszeit 10ms    (neuer Seek)**
- Anfrage D    **0x0814443E => Zugriffszeit 10ms    (neuer Seek)**
  - **Umlaufzeit (A) =  $T_{Ende}(A) - T_{Ankunft}(A) = 10ms - 0s = 10ms$**
  - **Umlaufzeit (B) =  $T_{Ende}(B) - T_{Ankunft}(B) = 20ms - 0s = 20ms$**
  - **Umlaufzeit (C) =  $T_{Ende}(C) - T_{Ankunft}(C) = 30ms - 0s = 30ms$**
  - **Umlaufzeit (D) =  $T_{Ende}(D) - T_{Ankunft}(D) = 40ms - 0s = 40ms$**

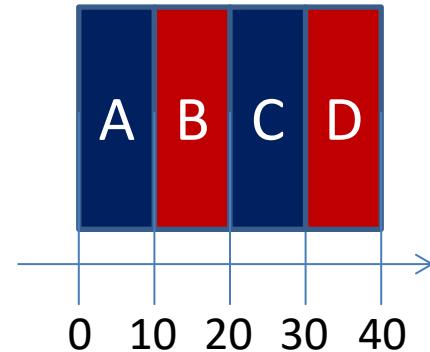


# Scheduling (FCFS)

Performance-Betrachtung (Umlaufzeit):

durchschnittliche Umlaufzeit:

$$(10\text{ms} + 20\text{ms} + 30\text{ms} + 40\text{ms}) / 4 = 25\text{ms}$$



Was geschieht, wenn sich die Reihenfolge ändert?



# Scheduling (FCFS)

## Performance-Betrachtung (Umlaufzeit):

- Anfrage A      **0x083FFF3F**      => Zugriffszeit 10ms      (initialer Seek)
  - Anfrage D      **0x0814443E**      => Zugriffszeit 10ms      (neuer Seek)
  - Anfrage B      **0x0814443F**      => entfällt      (kein neuer Seek)
  - Anfrage C      **0x083FFF3E**      => Zugriffszeit 10ms      (neuer Seek)
- 
- Anfrage D      **0x0814443E**      => Zugriffszeit 10ms      (initialer Seek)
  - Anfrage B      **0x0814443F**      => entfällt      (kein neuer Seek)
  - Anfrage C      **0x083FFF3E**      => Zugriffszeit 10ms      (neuer Seek)
  - Anfrage A      **0x083FFF3F**      => entfällt      (kein neuer Seek)



# Scheduling (FCFS)

Performance-Betrachtung (Umlaufzeit):

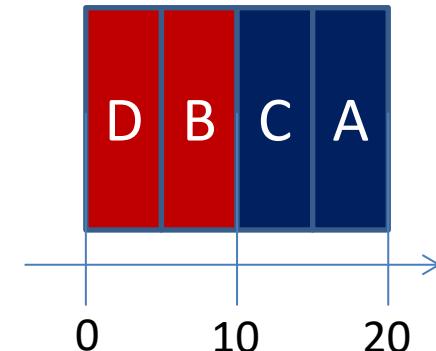
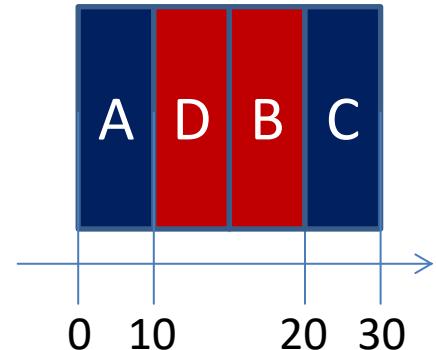
durchschnittliche Umlaufzeit:

$$(10\text{ms} + 20\text{ms} + 20\text{ms} + 30\text{ms}) / 4 = 20\text{ms}$$



$$(10\text{ms} + 10\text{ms} + 20\text{ms} + 20\text{ms}) / 4 = 15\text{ms}$$

Die Reihenfolge der Anfragen ist entscheidend!





# Scheduling

## Performance-Betrachtung (Umlaufzeit):

**Die durchschnittliche Umlaufzeit ist minimal  
gdw.,  
die Aufträge nach ihrer Dauer (Seek-Time + Latenzzeit)  
von kurz nach lang bearbeitet werden!**

## Die offensichtliche Idee:

- SPTF (Shortest Positioning Time First)  
„Die kürzeste Aufgabe kommt zuerst!“



# Scheduling (SPTF): Probleme

- Implementierung:  
Das Betriebssystem kann diese Strategie nur approximieren
  - Falls die Geometrie bekannt ist: **Shortest Seek Time First (SSTF)**



**Die Reihenfolge wird aufgrund der Zylindernummer festgelegt.**

- Falls die Geometrie nicht bekannt ist: **Nearest Block First (NBF)**



**Die Reihenfolge wird aufgrund der Blockadresse festgelegt.**



# Scheduling (SPTF): Probleme

- Starvation:

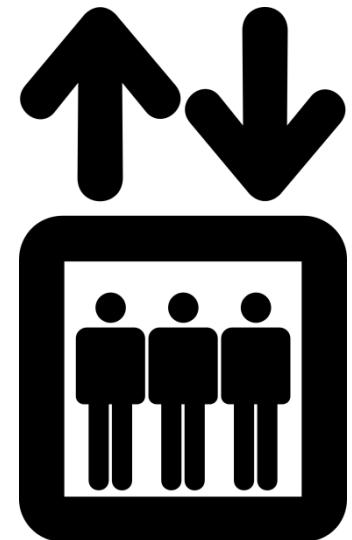
Bei vielen Anfragen, kommen weitentfernte Zugriffe zu kurz!



Fairness ist gefragt!



Eine einfache Erweiterung:  
„Fahrstuhl“-Algorithmen





# Scheduling (SCAN)

**Während sich der Schreib/Lese-Kopf von innen nach außen bzw. von außen nach innen bewegt, wird versucht, die unterschiedlichen Anfragen zu bedienen. Sobald eine Anfrage ein Segment adressiert, an dem der Schreib/Lese-Kopf sich bereits vorbeibewegt hat, muss diese Anfrage warten, bis der Schreib/Lese-Kopf seine Richtung wieder ändert. Der Kopf bewegt sich dabei stets von einem zum anderen Ende.**



# Scheduling (SCAN) Varianten

- **F-SCAN (Freeze)**

**In der Variante „Freeze“, stehen die Anfragen zu Beginn der Bewegung des Fahrstuhls fest. Alle Anfragen während der Bewegung werden gequeuet und erst auf dem Rückweg bearbeitet.**

- **C-SCAN (Circular)**

**In der Variante „Circular“, steht die Bewegungsrichtung fest. Anfragen werden nur in einer Richtung bearbeitet. Sobald das Ende erreicht ist, springt der Arm zurück und startet von neuem.**



# Scheduling (neue Idee)

**Zum Glück gehören alle diese Ideen der Vergangenheit an, in SSDs bewegt sich nichts mehr!**



# Hardware (SSD)

[micron: 7100 PCIE Family  
<http://www.micron.com>]



Auch SSDs (Solid-State Drives) unterstützen keinen echten Random-Access. Um Daten zu schreiben (Flash Page), müssen größere Bereiche gelöscht werden (Flash Block).

Dies ist aufwändig und führt längerfristig zur Zerstörung des Blocks!



# Hardware (SSD): Varianten

- **Single-Level Cell (SLC)**

In einer Zelle wird ein Bit gespeichert.

Viele Zellen wenige Bits



teuer



- **Multi-Level Cell (MLC)**

Zum Teil werden bis zu 3 Bits (8 unterschiedliche Werte) in einer Zelle kodiert.

Wenige Zellen viele Bits



günstig

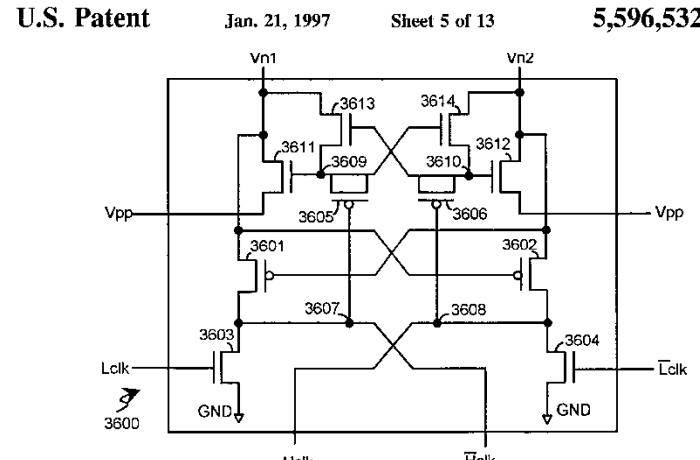


fig.4c  
prior art



# Flash: Speicherhierarchien

- **Page: (viele Zellen)**

Es kann immer nur eine komplette Page gelesen oder geschrieben werden (2KiB – 8KiB)

- **Block: (mehrere Pages)**

Es kann immer nur ein kompletter Block gelöscht werden (256 KiB – 1MiB)

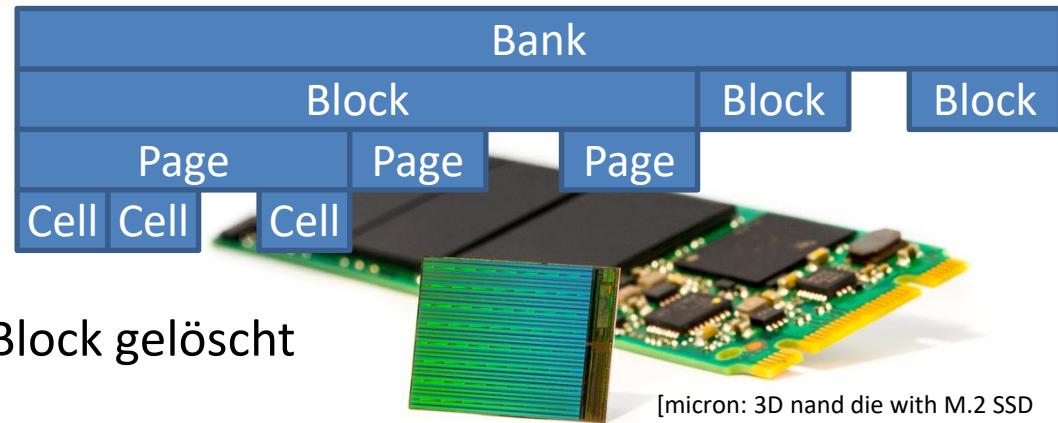
- **Bank: viele Blöcke,**

Über die Bank erfolgt der Zugriff auf die einzelnen Blöcke und ihre Pages (>> MiB)

- **Die (Wafer): mehrere Banks (meist 2),**

Auf die Banks eines Dies kann parallel zugegriffen werden.

- **Chip:** Vereinigt mehrere Dies in einem Bauteil



[micron: 3D nand die with M.2 SSD  
<http://www.micron.com>]



# Flash: Operationen

- **Lesen: (Read)**

Pages werden als Ganzes gelesen. Eine Positionierung ist nicht erforderlich. Lediglich die Page-Adresse muss bekannt sein.

**Zugriffszeit echt schnell: 0,1ms – 0,2ms ( $\approx$ 30 MiB/s)**

Seagate: Product Manual **1200.2 SSD**  
100773817, Rev. D. October 2016  
(<http://www.seagate.com>)

- **Schreiben: (Program)**

Pages können nicht mit beliebigen Werten beschrieben werden. Aufgrund von physikalischen Eigenschaften kann man lediglich „1“en in „0“en verwandeln und nicht umgekehrt. **Des Weiteren werden die Pages eines Blocks nur sequentiell geschrieben.**

**Aber auch dies geht verdammt schnell:**  
**Zugriffszeit : 0,025ms – 0,1ms ( $\approx$ 60MiB/s)**



# Flash: Operationen

- **Löschen: (Erase)**

Während Pages gelesen und „geschrieben“ (programmiert) werden können, ist ein Löschen nur blockweise möglich. Alle Bits werden auf „1“ gesetzt.

Für etwa 2ms muss eine entsprechende Spannung angelegt werden.

Löscht man den Speicher zu oft geht er kaputt. Etwa 100.000 Zyklen bei einer SLC-Platte und etwa 10.000 Zyklen bei einer MLC-Platte  
**(Dafür sind MLC-Platten etwa 10x günstiger)**



# Flash Operationen (Zusammenfassung)

Das „zufällige“ Lesen und Schreiben einer Page geht etwa 30 – 50mal schneller als bei herkömmlichen Festplatten, die aufgrund der Mechanik hier deutlich im Nachteil sind.

Problematisch wird es erst, wenn für einen Schreibvorgang Daten gelöscht, oder überschrieben werden müssen. Dies geht nur blockweise und dies ist teuer:

- Alle Pages des Blocks, die erhalten werden müssen, wegsteuern
- Block löschen
- Page schreiben

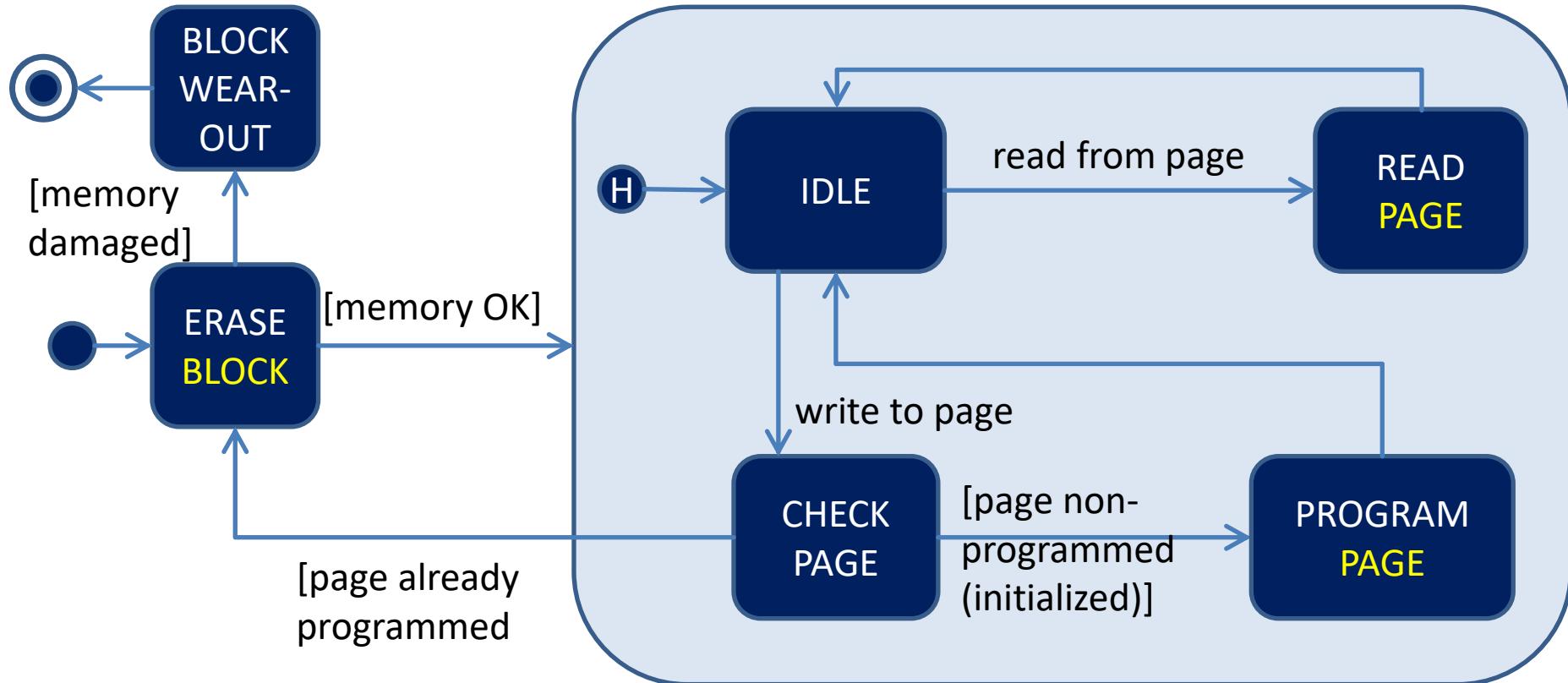
**Wird dies zu oft durchgeführt, wird der Block zerstört!**



**Management ist gefragt**



# Operationen (Lifecycle eines Blocks)





# Zustände einer Page

- **Clean:**  
Die Page wurde gelöscht und noch nicht beschrieben
- **Valid:**  
Die Page wurde programmiert (geschrieben). Der Inhalt ist gültig.
- **Invalid:**  
Im Rahmen eines Schreibvorgangs wurde der Inhalt dieser Seite geändert. (**Die neue gültige Seite befindet sich an einer anderen Stelle**)

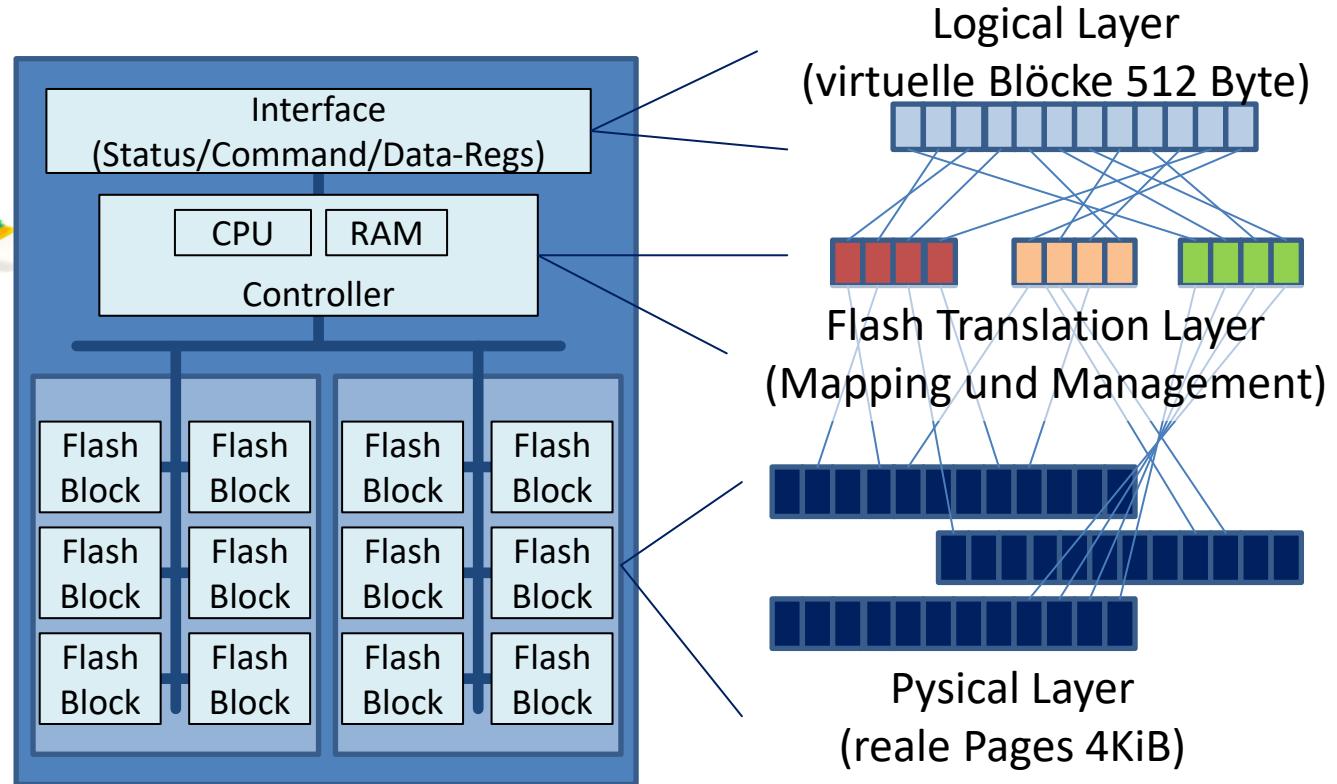
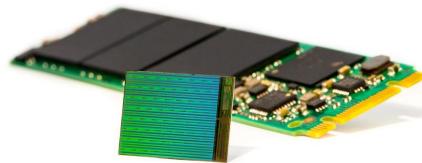
Flash Page	Flash Page	Flash Page	Flash Page
Flash Page	Flash Page	Flash Page	Flash Page
Flash Page	Flash Page	Flash Page	Flash Page
Flash Page	Flash Page	Flash Page	Flash Page
Flash Page	Flash Page	Flash Page	Flash Page

Achtung: Zum Teil müssen Blöcke gelöscht werden, die noch saubere Seiten enthalten => ungleichmäßiger Verschleiß innerhalb eines Blocks.



# Hardware (SSD)

[micron: 3D nand die with M.2 SSD  
<http://www.micron.com>]





# Flash Translation Layer

Übersetzt logische Adressen in physische Zellen (Pages).  
Verbirgt die realen Charakteristika des Speichermediums.



## zentrale Aufgabe:

Jimenez, X.; Novo,D. ; lenne, P. : Wear Unleveling – Improving NAND Flash Lifetime by Balancing Page Endurance. In FAST '14. 2014  
(<https://www.usenix.org/conference/fast14/technical-sessions/presentation/jimenez>)

## Garbage Collection:

Blöcke mit ungültigen Seiten werden gelöscht. Typischerweise werden Seiten aufgrund ihrer Update-Frequenz gruppiert.

- **Hot:** Änderungen stehen bevor. Hot-Blocks verfügen über viele ungültige Seiten.
- **Warm:** Änderungen sind zu erwarten. Warm-Blocks verfügen über wenige ungültige Seiten.
- **Cold:** Änderungen sind eher ausgeschlossen. Cold-Blocks verfügen über kaum ungültigen Seiten.



# Flash Translation Layer

**Mit der Qualität der Algorithmen des FTLs  
steht und fällt die Lebensdauer und die  
tatsächliche Performance der SSD!**



# Flash Translation Layer

**Übersetzen erfordert Platz (Page Tables)**

pro TiB etwa 1GiB (bei 4 Byte pro Page)

**Deutlich zu viel Speicherbedarf und deutlich zu heiß!  
Mehrstufige Verfahren sind gefragt!**

- Wenige Blöcke, die page-weise übersetzt werden.
- Viele Blöcke, die zusammenhängende Adressen beinhalten und blockweise übersetzt werden.



# Flash Translation Layer

**Inwieweit kann das Betriebssystem den Umgang mit einer SSD unterstützen?**



Scheduling der Schreibvorgänge

**Wer ganze Blöcke schreibt, der reduziert unnötiges Kopieren!**



# Flash Translation Layer

Rollins, Doug : **The Effect of Host Data Patterns on SSD Write Performance.**

*Technical Marketing Brief, Micron Technology (2012)*

([http://www.micron.com/-/media/documents/products/technical-marketing-brief/brief\\_ss\\_effect\\_data\\_placement\\_writes.pdf](http://www.micron.com/-/media/documents/products/technical-marketing-brief/brief_ss_effect_data_placement_writes.pdf))

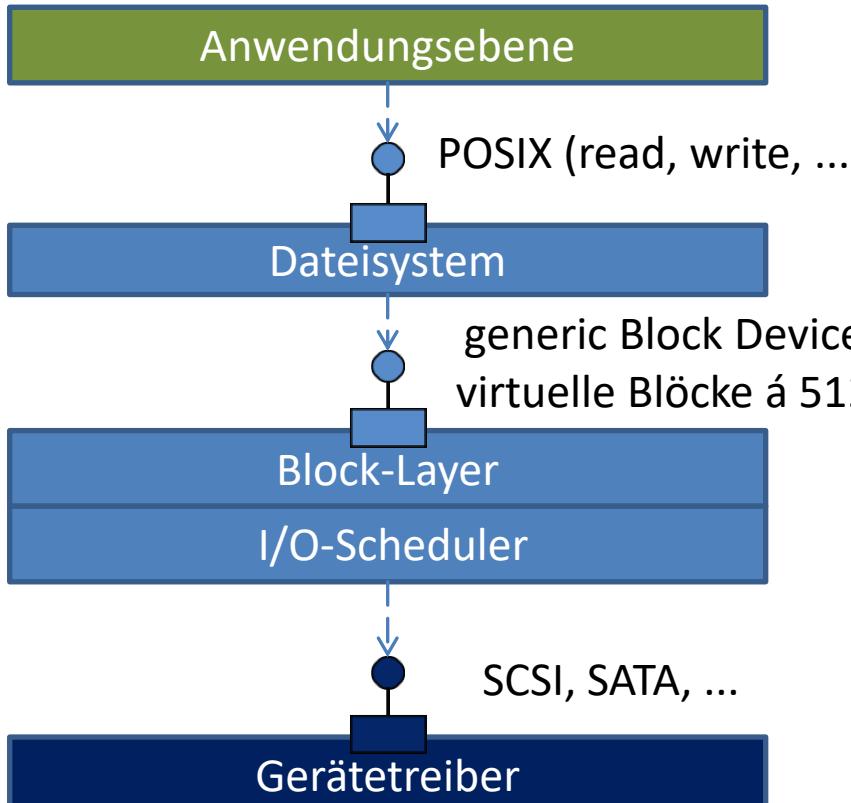
**Sequential Write vs. Random Write**  
*26.000 IOPs vs. 10.500 IOPs*



# File-Systeme



# File-Systeme



Wie verwaltet und organisiert man persistenten Speicher (, der aus einzelnen, adressierbaren Blöcken besteht)?



**Neue Abstraktionsebene:  
Dateien und Verzeichnisse**



# Dateien und Verzeichnisse

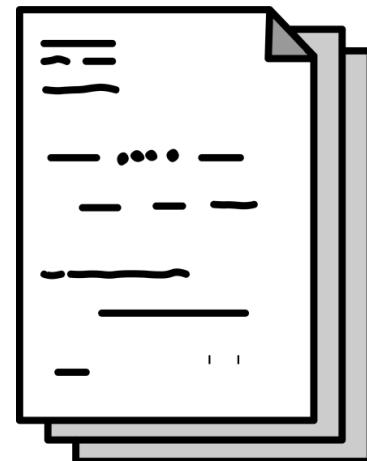


# Dateien

**Eine Datei ist eine zusammenhängende Folge von Bytes, die gelesen und geschrieben werden können.**

Dateien müssen für Benutzer und Systeme identifizierbar sein :

- **Inode-Number**  
Ein eindeutiger Index (pro File-System), der die Datei identifiziert.
- **Dateien haben einen Namen (Pfad):**  
Ein symbolischer Name, der eine Datei beginnend vom Wurzelverzeichnis („/“) aus identifiziert (/dir1/dir2/datei).
- **File Descriptor:**  
Eine Nummer, die eine geöffnete Datei aus Sicht eines Prozesses identifiziert. (Verwaltet im Wesentlichen die Inode-Nummer plus einen Offset)





# Dateien

Ein File-System verwaltet zu jeder Datei eine Menge an Informationen (Meta-Daten)

```
:> debugfs a_blockdevice
debugfs 1.42.12 (29-Aug-2014)
debugfs: stat Betriebssysteme-Teil2.pdf
Inode: 15  Type: regular    Mode:  0770    Flags: 0x0
Generation: 2267572304  Version: 0x00000001
User: 1000  Group: 1000   Size: 1529105
File ACL: 8484  Directory ACL: 0
Links: 1  Blockcount: 3004
Fragment: Address: 0  Number: 0  Size: 0
ctime: 0x5845889d -- Mon Dec  5 16:32:45 2016
atime: 0x5845885b -- Mon Dec  5 16:31:39 2016
mtime: 0x5845885b -- Mon Dec  5 16:31:39 2016
```

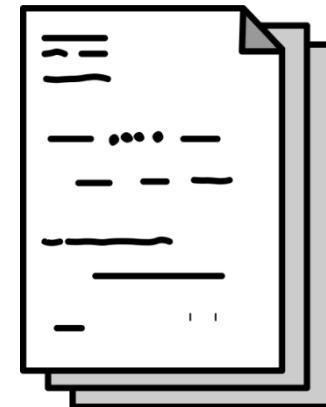
BLOCKS:  
(0-11):2097-2108,  
(IND):679,  
(12-15):2109-2112,  
(16-31):1681-1696,  
(32-63):1761-1792,  
(64-127):1953-2016,  
(128-255):4737-4864,  
(256-267):13441-13452,  
(DIND):680,  
(IND):901,  
(268-511):13453-13696,  
(512-523):13825-13836,  
(IND):902,  
(524-779):13837-14092,  
(IND):903,  
(780-1023):14093-14336



# Inodes

Meta-Daten werden typischerweise separat, in einem Index-Knoten (Inode) verwaltet.

```
:> ls -i
12 Aufgaben1.pdf
13 Aufgaben2.pdf
14 Betriebssysteme-Teil1.pdf
15 Betriebssysteme-Teil2.pdf
16 LoesungBlatt1.pdf
17 LoesungBlatt2.pdf
11 lost+found
1673 workspace
```



Wird eine Datei gelöscht, dann kann dieser Knoten und dessen Index wiederverwendet werden.



# Verzeichnisse

Wurzel „/“

Name	Inode-Number
.	2
..	2
Aufgaben1.pdf	12
Aufgaben2.pdf	13
Betriebssysteme-Teil1.pdf	14
Betriebssysteme-Teil2.pdf	15
LoesungBlatt1.pdf	16
LoesungBlatt2.pdf	17
lost+found	11
workspace	1673

Verzeichnisse sind Verwaltungsstrukturen.  
Jedes Verzeichnis verwaltet die Zuordnung  
der in ihm abgelegten Dateien auf deren  
Inodes.

„workspace“

Name	Inode-Number
.	1673
..	2
.metadata	1674



# Beispiel: Root-Verzeichnis

```
00000000 02 00 00 00 0c 00 01 02 2e 00 00 00 02 00 00 00  
00000020 0c 00 02 02 2e 2e 00 00 0b 00 00 00 24 00 0a 02  
00000040 6c 6f 73 74 2b 66 6f 75 6e 64 00 00 89 06 00 00  
00000060 10 00 05 02 4c 69 6e 75 78 00 00 00 0c 00 00 00  
0000100 18 00(0d)01 41 75 66 67 61 62 65 6e 31 2e 70 64  
0000120 66 00 00 00 0d 00 00 00 18 00(0d)01 41 75 66 67
```

0000140 **61 62 65 6e 32 2e 70 64 66 00 00 00 0f 00 00 00**

0000160 24 00 19 01 42 65 74 72 69 65 62 73 73 79 73 74

0000200 65 6d 65 2d 54 65 69 6c 32 2e 70 64 66 66 00 00

0000220 89 06 00 00 24 00 09 02 77 6f 72 6b 73 70 61 63

0000240 65 79 73 74 65 6d 65 2d 54 65 69 6c 20 32 2e 70

0000260 64 66 00 00 10 00 00 00 1c 00(11)01 4c 6f 65 73

0000300 **75 6e 67 42 6c 61 74 74 31 2e 70 64 66 00 00 00**

0000320 **11 00 00 00 1c 00(11)01 4c 6f 65 73 75 6e 67 42**

0000340 **6c 61 74 74 32 2e 70 64 66 00 00 00 0e 00 00 00**

0000360 14 03 19 01 42 65 74 72 69 65 62 73 73 79 73 74

0000400 65 6d 65 2d 54 65 69 6c 31 2e 70 64 66 00 00 00

0000420 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

\*

0002000

**df.txt**

**Aufgaben1.pdf**

**Aufgaben2.pdf**

**LoesungBlatt1.pdf**

**LoesungBlatt2.pdf**

**Wortlänge (13)(13)(17)(17)**

**Inode-No (12)(13)(16)(17)**

```
:> debugfs a_blockdevice
debugfs: dump . dumpfile
debugfs: quit
:> od -t x1 dumpfile > df.txt
:>
```



# Beispiel: Root-Verzeichnis

```
00000000 02 00 00 00 0c 00 01 02 2e 00 00 00 02 00 00 00  
0000020 0c 00 02 02 2e 2e 00 00 0b 00 00 00 24 00 0a 02  
0000040 6c 6f 73 74 2b 66 6f 75 6e 64 00 00 89 06 00 00  
0000060 10 00 05 02 4c 69 6e 75 78 00 00 00 0c 00 00 00  
0000100 18 00 0d 01 41 75 66 67 61 62 65 6e 31 2e 70 64  
0000120 66 00 00 00 0d 00 00 00 18 00 0d 01 41 75 66 67  
0000140 61 62 65 6e 32 2e 70 64 66 00 00 00 0f 00 00 00  
0000160 24 00 19 01 42 65 74 72 69 65 62 73 73 79 73 74  
0000200 65 6d 65 2d 54 65 69 6c 32 2e 70 64 66 66 00 00  
0000220 89 06 00 00 24 00 09 02 77 6f 72 6b 73 70 61 63  
0000240 65 79 73 74 65 6d 65 2d 54 65 69 6c 20 32 2e 70  
0000260 64 66 00 00 10 00 00 00 1c 00 11 01 4c 6f 65 73  
0000300 75 6e 67 42 6c 61 74 74 31 2e 70 64 66 00 00 00  
0000320 11 00 00 00 1c 00 11 01 4c 6f 65 73 75 6e 67 42  
0000340 6c 61 74 74 32 2e 70 64 66 00 00 00 0e 00 00 00  
0000360 14 03 19 01 42 65 74 72 69 65 62 73 73 79 73 74  
0000400 65 6d 65 2d 54 65 69 6c 31 2e 70 64 66 00 00 00  
0000420 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
*  
0002000
```

**df.txt**

**Was findet man noch heraus?  
Wie hieß das Verzeichnis  
„workspace“ bevor es  
umbenannt wurde?**



# Umgang mit Dateien (LINUX)

- **creat(path) und mkdir(path):**

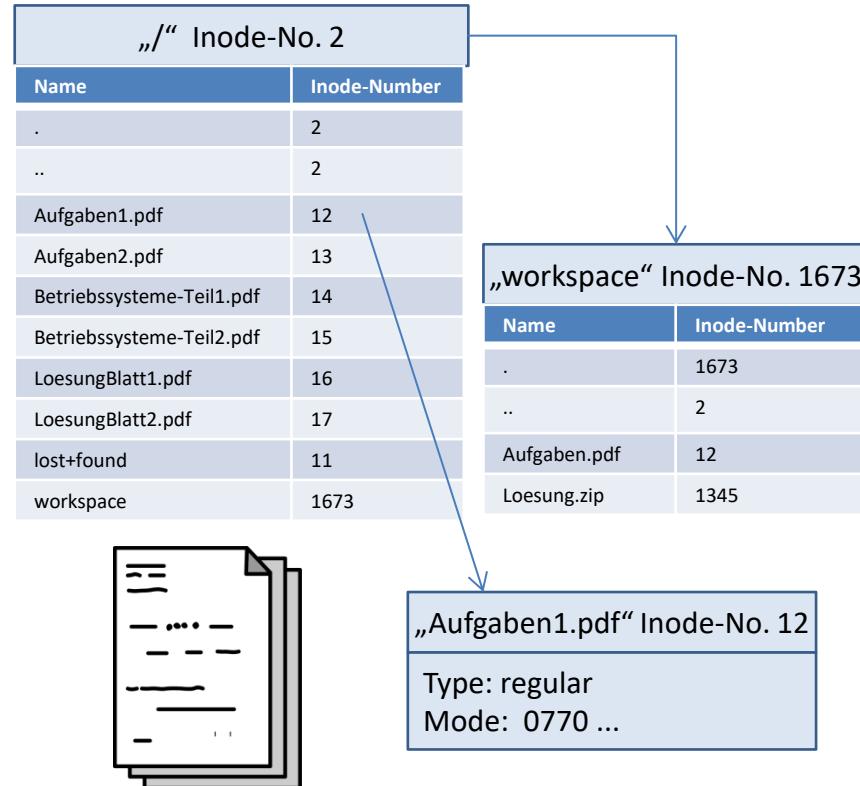
Erzeugt eine Inode für eine Datei bzw. ein Verzeichnis und verbindet sie mit einem symbolischen Namen.

- **link(name, newName)**

Erzeugt einen neuen Verweis. Dieselbe Inode ist nun über einen weiteren Namen adressiert.

- **unlink (name)**

Löscht einen Verweis auf eine Inode. Sobald eine Inode alle Verweise verloren hat, kann sie als gelöscht betrachtet werden und wird zusammen mit dem physischen Speicherplatz recycelt.





# Umgang mit Dateien (LINUX)

- **open(path):**

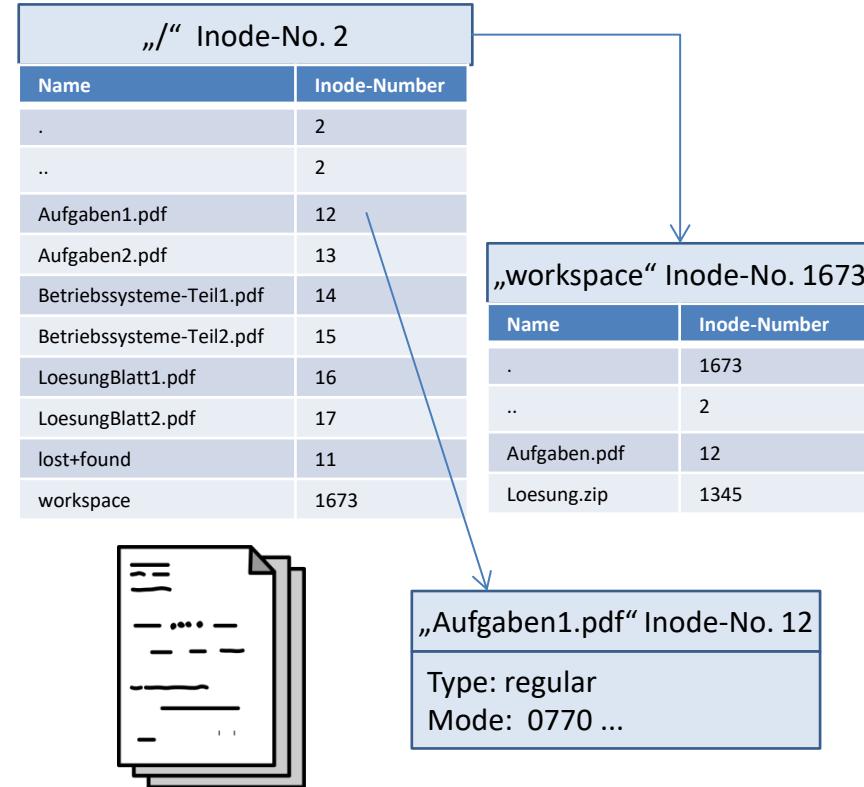
Ausgehend vom aktuellen Verzeichnis (ggf. Wurzelverzeichnis) und dessen Inode wird die Inode der entsprechenden Datei ermittelt.

- **read (path):**

Über die Inode kann auf die einzelnen Datenblöcke zugegriffen werden.

- **write(path, data):**

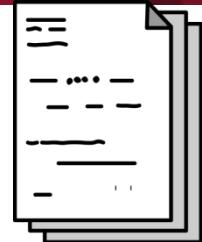
Abhängig vom Modus werden bestehende Blöcke überschrieben oder neue erzeugt und angehängt. **Damit verbunden sind typischerweise auch Änderungen an der Inode selbst.**



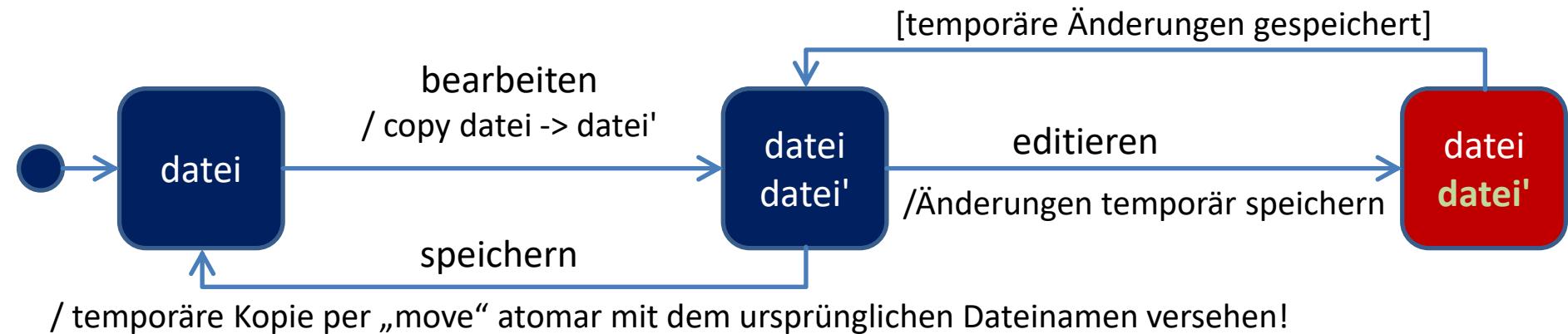


# Umgang mit Dateien (LINUX)

- **move (name):** Ändert den Namen einer Datei bzw. eines Verzeichnisses, den Verweis auf die Inode. Diese Änderung erfolgt atomar.



Änderungen an Dateien können so erfolgen, dass daraus stets ein konsistenter Zustand resultiert.





# Ein simples File-System



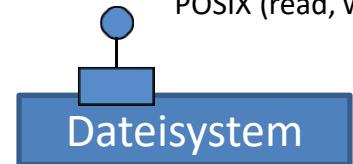
# Reine Software

Zum ersten Mal benötigen wir keine Hardware-Unterstützung.

- Dateisysteme sind Software-Lösungen
- Dateisysteme verfügen über eine definierte Schnittstelle



Ziel: „Optimale“ Implementierung dieser Schnittstelle.



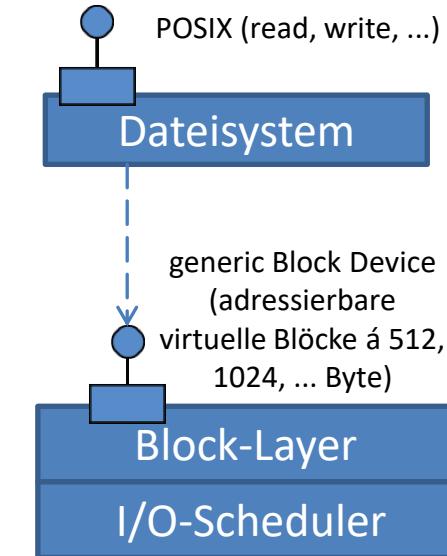


# 1. Schritt: Wahl geeigneter Strukturen

## Verwaltung von Daten und Meta-Daten.

- Welche Größe haben die virtuellen Blöcke?
  - ein Block pro Datei?
  - ein Block pro Inode?
- Wie sind die Blöcke organisiert?
  - linear (Array)
  - als Baum
  - ...

unrealistisch

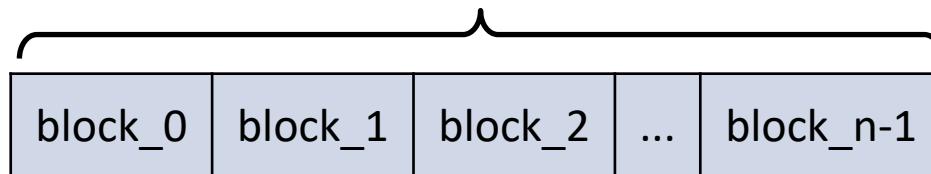




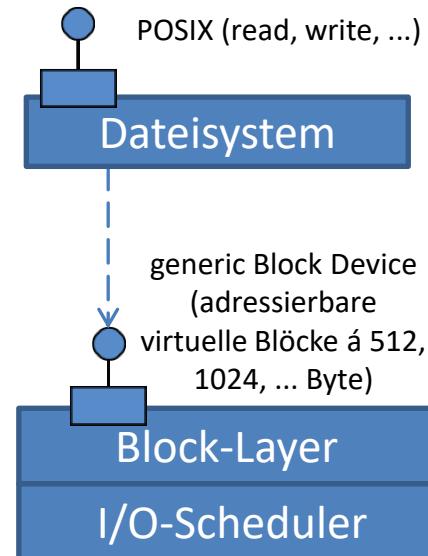
# Datenstrukturen

KISS: „Keep it simple, stupid“

- Blöcke erhalten eine feste Größe, 4KiB
- Blöcke werden als Array organisiert



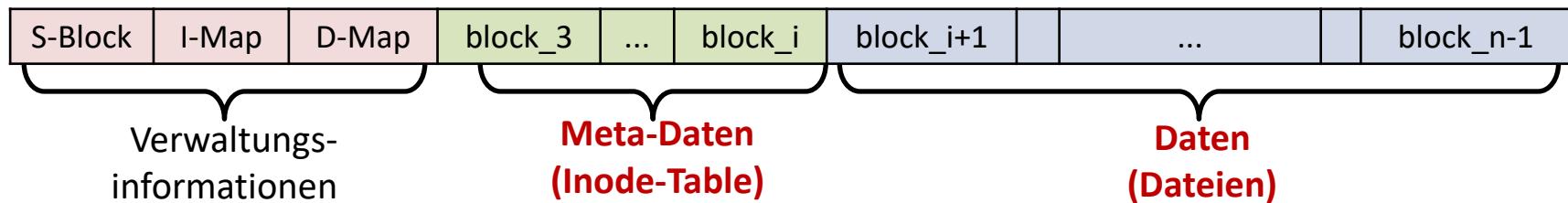
pro virtuellem Block 8 physische Segmente





# Layout

- Blöcke für Meta-Daten (Inodes) – wenige
- Blöcke für Daten (Verzeichnisse und Dateien) – viele
- Blöcke für allgemeine Informationen zum Dateisystem
  - Superblock (Informationen zum File-System, Anzahl Inode-/Daten-Blöcke, usw.)
  - Blöcke zur Verwaltung freier und belegter Inode/Daten-Blöcke (Free/Used List)



**Typischerweise liegen mehrere Inodes im selben Block und typischerweise werden mehrere Blöcke für eine Datei gebraucht!**



# Virtuelle Blöcke und physische Sektoren

## Zuordnung: Inode-Nummer => Block => Sektor

1. relativen Blockindex berechnen

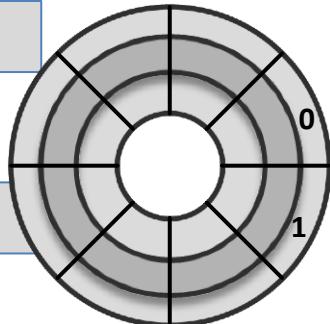
```
int rblockIdx = (inodeNumber * sizeof(inode_t)) / sizeof(block_t);
```

2. absolute Adresse bestimmen

```
inode_t *inodeAddr = (rblockIdx * sizeof(block_t))+ &inodeTable;
```

3. Sektor-Adresse berechnen

```
int sector = inodeAddr / sizeof(sector_t);
```





# Inode

Inode	
Meta-Daten	
Größe	6457
Zugriffsrechte	rwx
letzter Zugriff	1.1.2017
bestehende Links	42
...	
Block List	
0x013FFF30	
0x013FFF31	
...	
	0x023FFE48

**Wir haben ein Problem:**  
Eine einfache Abbildung virtueller Adressen (Blöcke) auf physische Adressen (Segmente) gelingt nur, wenn jede Inode die gleiche Größe hat.



**Kleine Dateien oder sehr  
große Inodes!**



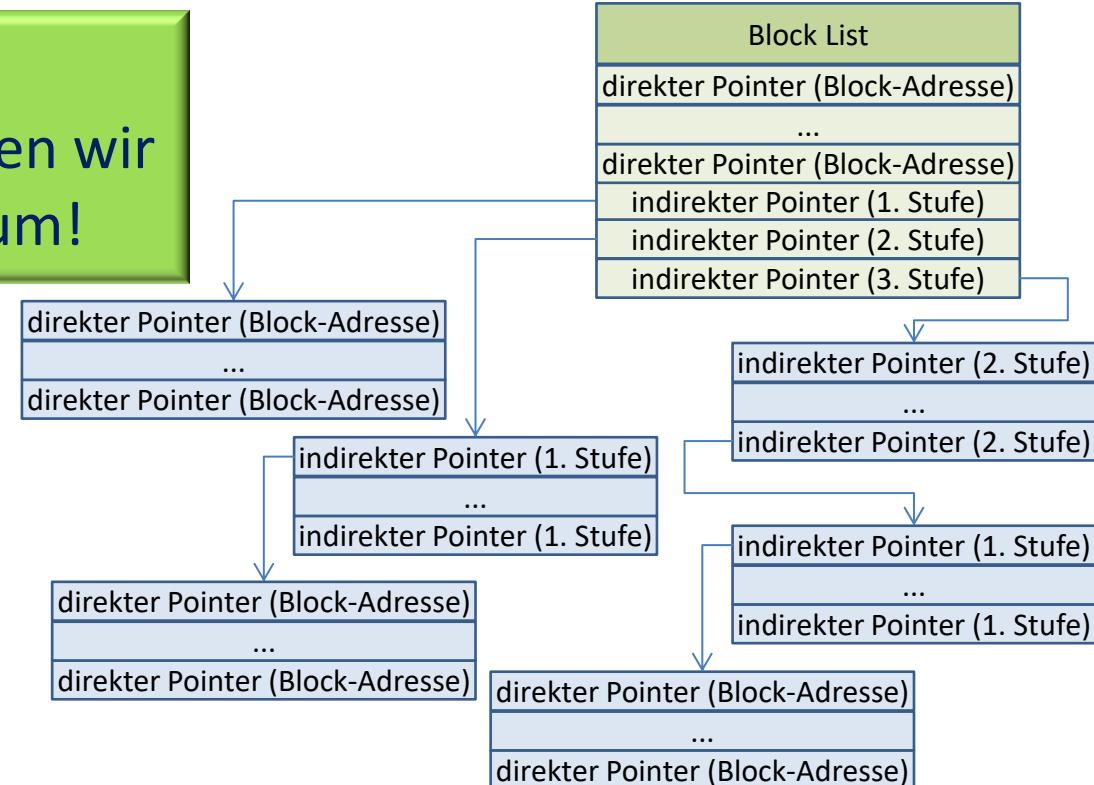
# Idee: hierarchische Baumstrukturen

Hierarchien:  
Statt in einer Tabelle verwalten wir  
die Adressen in einem Baum!

Multi-Level Page Tables



Multi-Level Block Lists



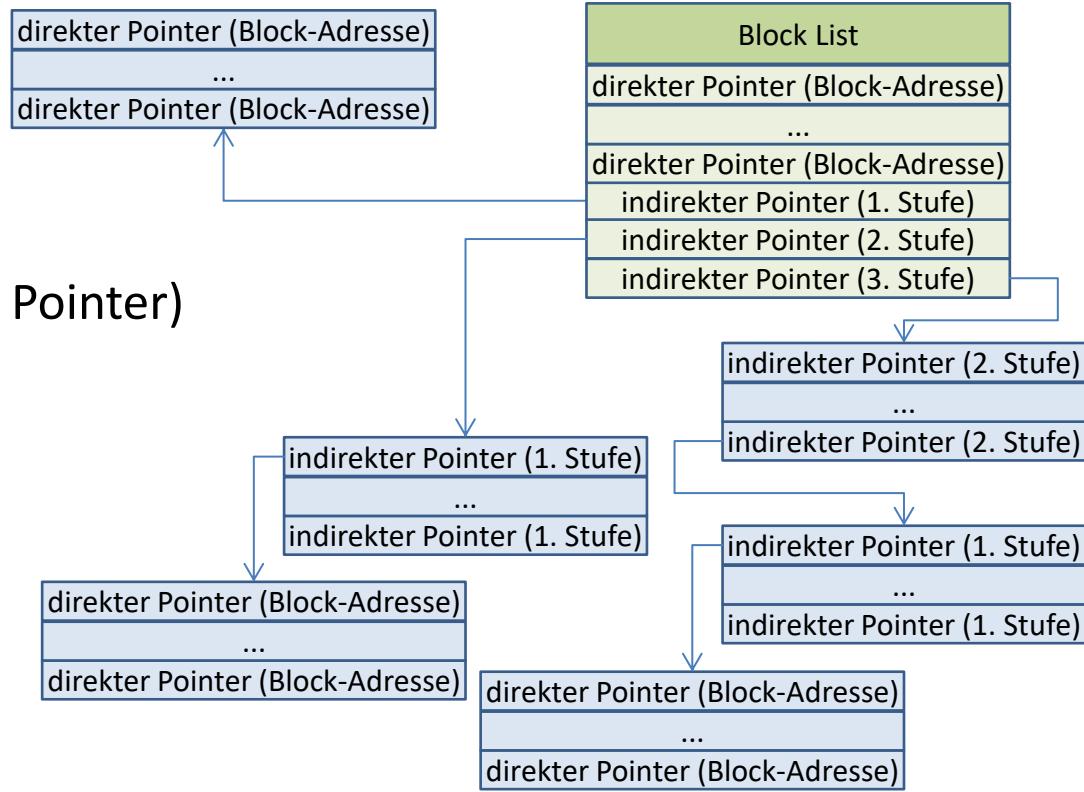


# Größenbetrachtung

- Inode 512 Byte
  - 12 direkte Pointer
  - 3 indirekte Pointer
- 1024 Pointer pro Block  
(Blockgröße 4KiB und 4 Byte pro Pointer)

## Unterstützte Dateigröße

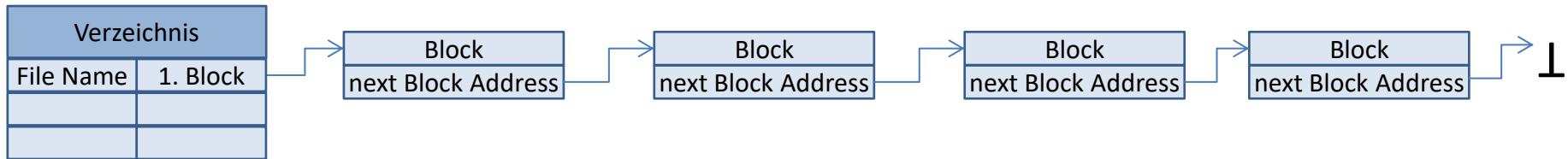
$$\begin{aligned} 12 \times 4\text{KiB} &= 48\text{KiB} \\ + 1024 \times 4\text{KiB} &= 4\text{MiB} \\ + 1024 \times 4\text{MiB} &= 4\text{GiB} \\ + 1024 \times 4\text{GiB} &= 4\text{TiB} \\ \hline &\approx 4,4\text{TB} \end{aligned}$$





# Alternative Idee

- Dateien sind Listen verketteter Blöcke
- Meta-Daten werden auf Verzeichnisebene verwaltet



Problem:



- Die Größe eines Blocks ist nicht mehr ein Vielfaches einer großen Zweierpotenz (der Pointer benötigt Platz)
- Der Zugriff ist langsam, die Blöcke müssen in der richtigen Reihenfolge gelesen werden!

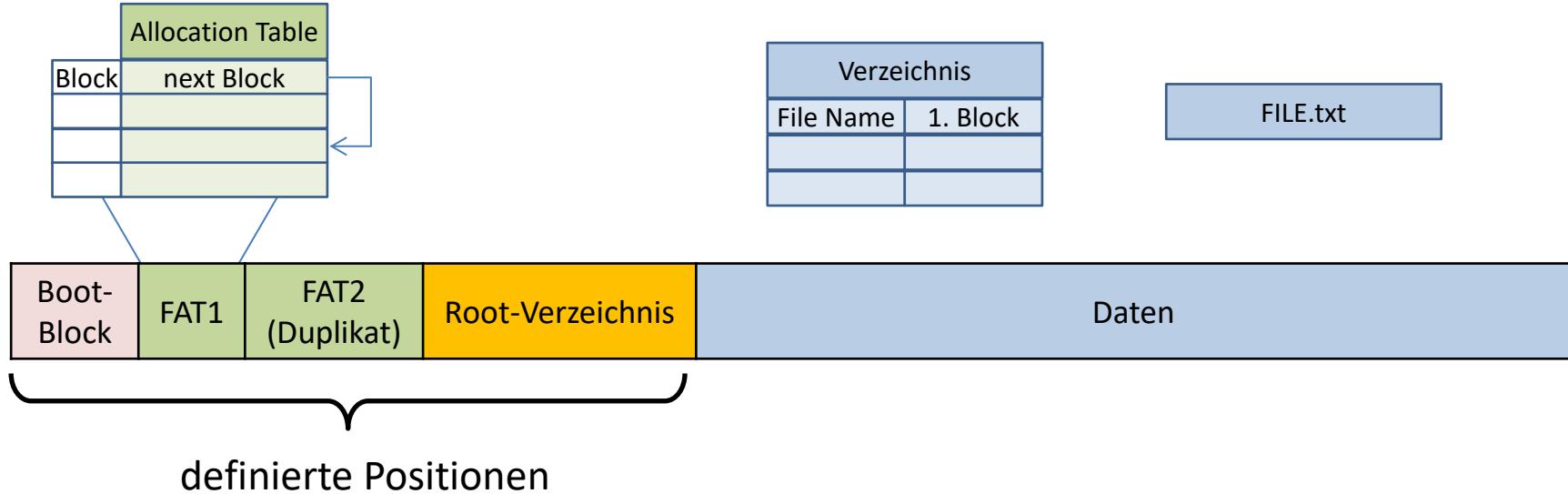


# Alternative Idee

Lösung:



Separate Tabelle für die Pointer



[Microsoft: Extensible Firmware Initiative FAT32 File System Specification,  
FAT: General Overview of On-Disk Format. Microsoft Corporation, 2000]



# 2. Schritt: Free-Space Management

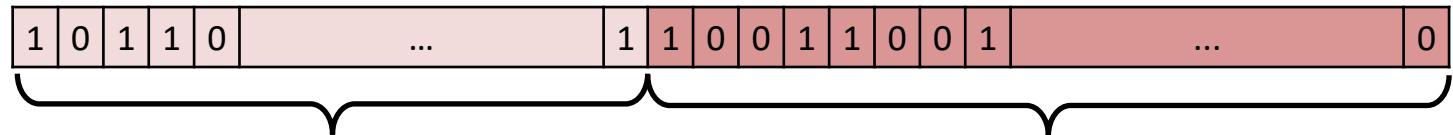
Will man Daten speichern, dann benötigt man Platz;  
diesen muss man finden.

- **Free-List:**

Nachteil, die Liste befindet sich auf der Platte und nicht im RAM => langsam

- **Bit-Maps:**

simpel und sparsam



- **Bäume:**

warum nicht

belegte (1) / freie (0) Inodes  
(I-Map)

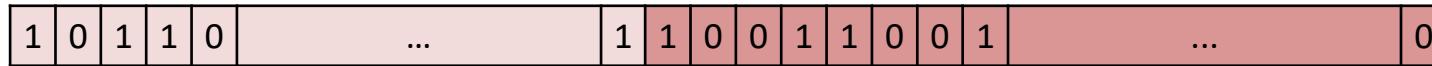
belegte (1) / freie (0) Datenblöcke  
(D-Map)

- ...



# Free-Space Management (Strategie)

## Welchen Block sollte man wählen?



- Best Fit
- Worst Fit
- First Fit
- Next Fit
- ...

Die Daten werden auf einem Medium gespeichert.  
Zusammenhängende Blöcke versprechen zusammenhängende  
Sektoren bzw. zusammenhängende SSD-Blöcke.

**Neue Dateien eher größer als kleiner dimensionieren.  
Nicht „einen“ Block, sondern „n“ Blöcke!**



# Ein generelles Problem: Performance

Ein Zugriff auf eine Datei bedeutet:

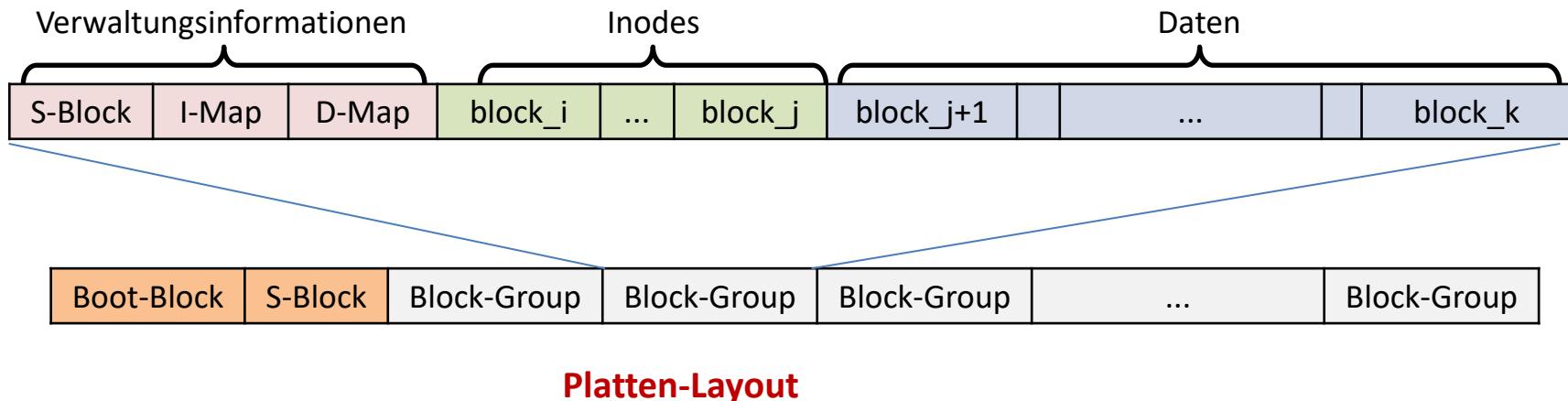
- Zugriff auf die Datenblöcke der Datei
- Zugriff auf die Inode der Datei, um die Datenblöcke zu lokalisieren
- Zugriff auf ein Verzeichnis, um die Inode der Datei zu lokalisieren
- Zugriff auf die Inode des Verzeichnisses, um die Datenblöcke des Verzeichnisses zu lokalisieren.
- ...

**Dies bedeutet exzessiver Random-Read und ggf. Random-Write!**



# Ein einfache Lösung: Mehr Lokalität

- Inode und Daten müssen näher zusammenrücken
- Dateien und Verzeichnisse müssen näher zusammenrücken





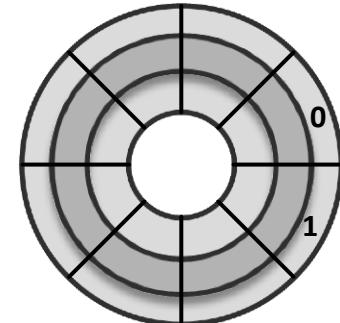
# Ein einfache Lösung: Mehr Lokalität

Es besteht die berechtigte Hoffnung, dass Sektor-Adressen, die nahe beieinanderliegen auch physisch benachbart sind.  
Idealerweise auf dem gleichen Track oder im gleichen Zylinder

- Verzeichnis-Daten, Verzeichnis-Inode, dessen File-Inodes und die darin direkt-adressierten Blöcke kommen möglichst in dieselbe Block-Group.  
**(Dateien in einem Verzeichnis hängen voneinander ab)**
- Große Dateien bekommen ihre eigenen Block-Groups.

Ziel:

Der Block mit den direkten Pointern und die darüber adressierten Blöcke liegen in derselben Block-Group!





# Ein konsistentes File-System



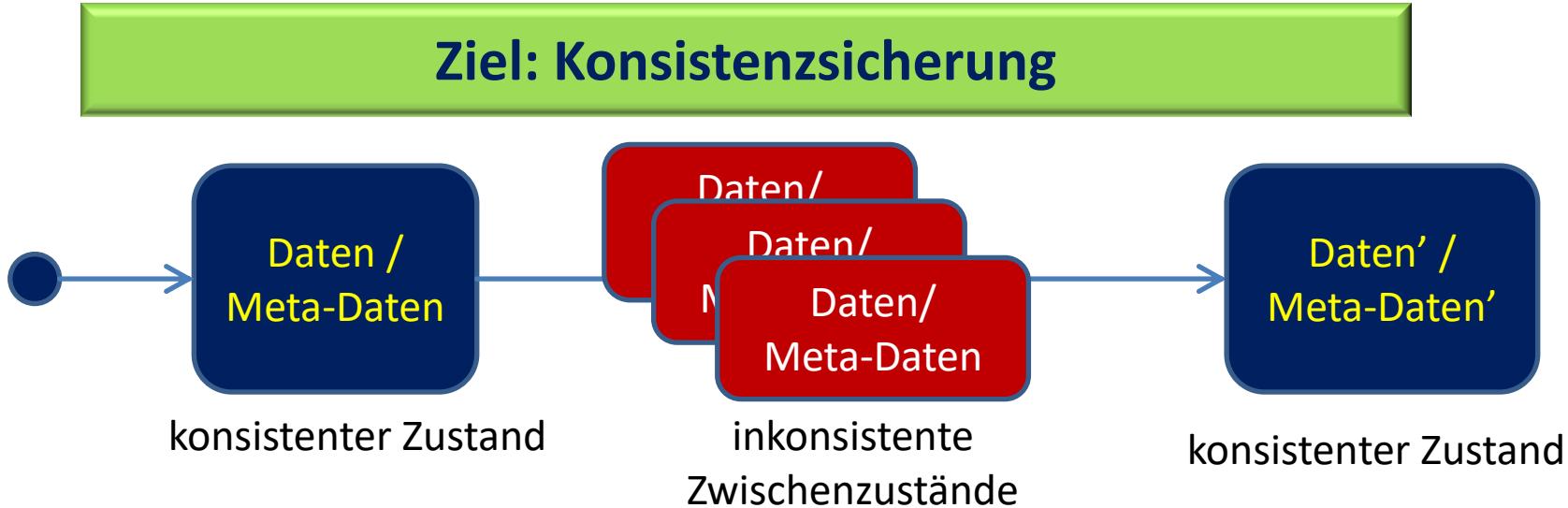
# 3. Schritt: Konsistenz

**Ein Datei-System macht nur Sinn, wenn garantiert werden kann, dass die Daten, die sich auf der Platte befinden, korrekt sind.**

**Das Ändern von Daten und Meta-Daten ist ein „kritischer Abschnitt“.**



# 3. Schritt: Konsistenz



**Auch im Falle eines Crashes (Stromausfall) darf sich das System nach dem Neustart nicht in einem inkonsistenten Zwischenzustand befinden.**

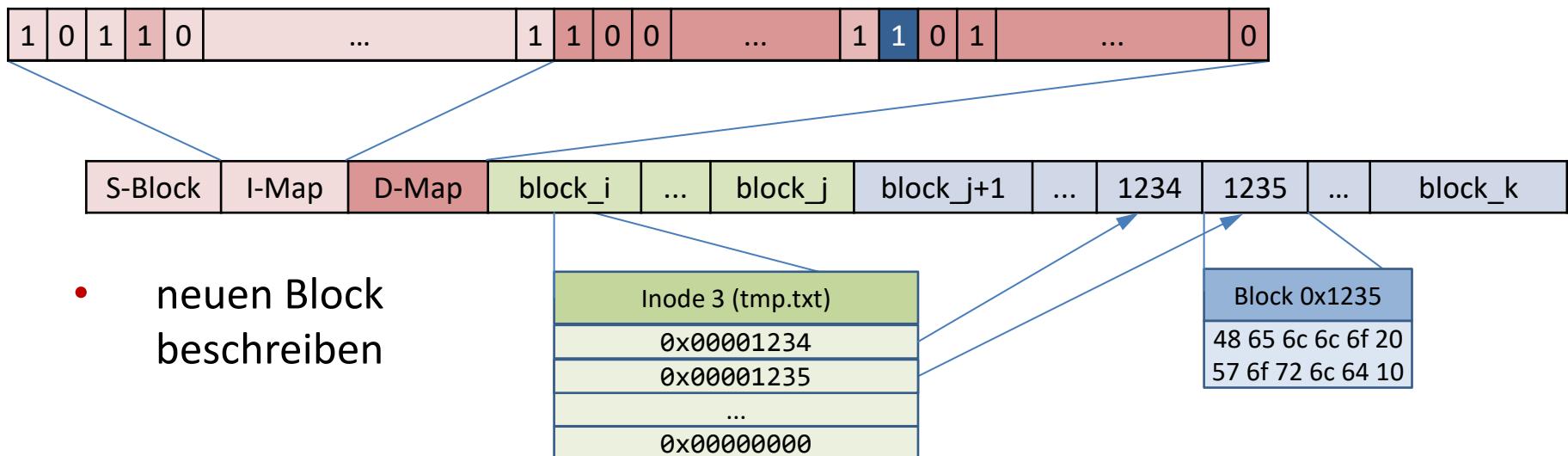


# Details

“Hello World” wird angehängt

- Inode ändern
- D-Map ändern

```
int fd = open("tmp.txt", O_RDWR | O_APPEND);
write(fd, "Hello World\n", 12);
...
```





# Details

## kritisch aber konsistent:

Lediglich der Block enthält Unsinn.

## kritisch und inkonsistent:

Keine Daten wurden geschrieben. Gemäß Inode sind aber Daten vorhanden und gemäß D-Map ist der Block frei.

## kritisch und inkonsistent:

Daten wurden geschrieben und die Inode verweist auf den richtigen Block. Dieser ist aber offiziell noch frei und könnte anderweitig vergeben werden.

## kritisch und inkonsistent:

Keine Daten wurden geschrieben. Gemäß D-Map ist der Block belegt, aber er wird in keiner Inode geführt.  
**Der Block ist verloren.**

## kritisch und inkonsistent:

Daten wurden geschrieben und in der D-Map vermerkt. Der Block wird aber in keiner Inode geführt.  
**Daten und Block sind verloren.**

## konsistent und (unkritisch):

Daten wurden geschrieben, aber niemand weiß davon.



# Lösung des Konsistenzproblems

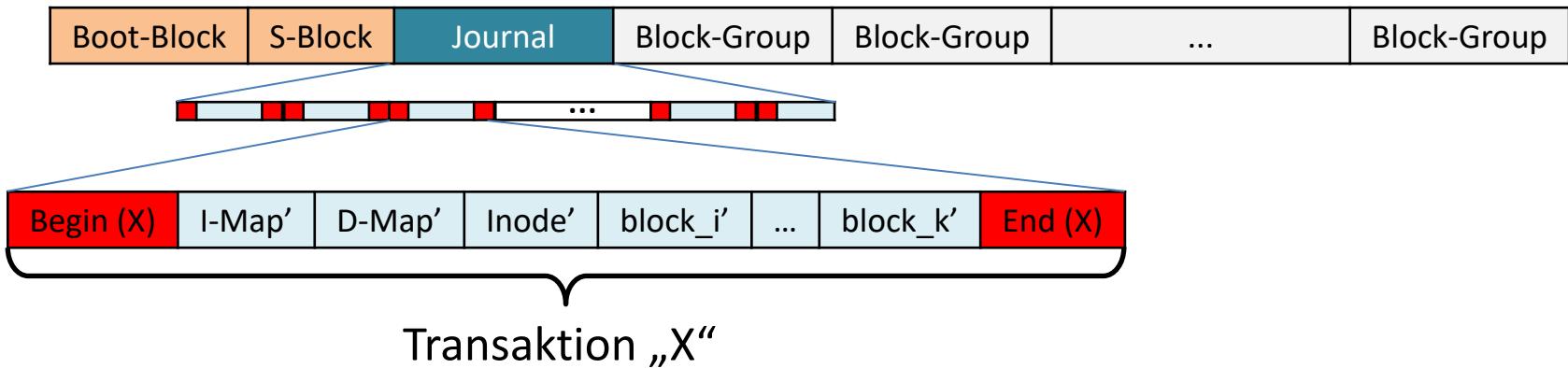
- Probleme lösen
  - **Check-and-Repair:** Inkonsistenzen werden systematisch gesucht, erkannt und beseitigt (fsck)

Bei heutigen Plattengrößen ist der dafür erforderliche Aufwand jedoch nicht mehr tragbar. (Linux ext2)
- Probleme vermeiden
  - **Write-Ahead (Journaling):** Das File-System erhält ein Log (Journal) in dem die bevorstehenden Änderungen in Form von Transaktionen protokolliert werden.
  - **Copy-on-Write:** Werden Blöcke geändert, werden Sie kopiert und zu einem definierten Zeitpunkt wird das File-System einfach „ausgetauscht“ (Bsp.: ZFS).



# Write-Ahead (Journaling)

Ein Journal protokolliert die bevorstehenden Änderungen!





# Write-Ahead (Journaling)

- **Änderungen protokollieren (Journal Write):**  
Alle Änderungen an Daten und Meta-Daten, die für einen Schreib/Create/Delete-Vorgang notwendig sind, werden im Journal abgelegt und mit einem Begin- und einem End-Block versehen.
- **Änderungen durchführen und Aufräumen (Checkpoint and Free):**  
Die realen Änderungen werden erst danach ausgeführt. Sobald diese durch den Platten-Controller bestätigt wurden, kann der Journal-Eintrag gelöscht werden.



# Write-Ahead (Journaling)

## Im Falle eines Crash

- Ein Crash während der Protokollphase ist unkritisch, da nichts passiert ist!
- Ein Crash während die eigentlichen Änderungen durchgeführt werden ist unkritisch, da das Journal existiert!
- Ein Crash nachdem die Änderungen durchgeführt wurden, aber bevor das Journal aufgeräumt wurde ist unkritisch, da die Daten jetzt zwar zweimal geschrieben werden, aber dies den Inhalt sicher nicht verändert.

**Wichtig ist, dass beim Schreiben eine Reihenfolge eingehalten wird.**



# Die Reihenfolge beim Protokollieren

**Beginn und Ende einer Transaktion werden getrennt.**

- Änderungen protokollieren (Journal Write):  
Alle Änderungen werden im Journal abgelegt und mit einem Begin-Block versehen.  
**Dieser Vorgang kann als Ganzes, in Teilen und in beliebiger Reihenfolge erfolgen.**

**Typischerweise definiert der Platten-Controller die genaue Reihenfolge!**



# Die Reihenfolge beim Protokollieren

- **Änderungen abschließen (Journal Commit):**

Sind die Änderungen protokolliert - **und nur dann** - wird der End-Block geschrieben.

**Entspricht die Größe des End-Blocks der Größe eines Sektors,  
erfolgt dieser Schreibvorgang aus Sicht der Platte „atomar“!**

- **Änderungen durchführen (Checkpoint):**

**Die tatsächlichen Änderungen können auch zu einem späteren  
Zeitpunkt erfolgen. Änderungen können gesammelt werden!**

- **Journal-Eintrag freigeben (Free):**

Zu einem beliebig späteren Zeitpunkt kann der Eintrag im Journal freigegeben werden.  
Lediglich die Reihenfolge muss bei der Freigabe beachtet werden.



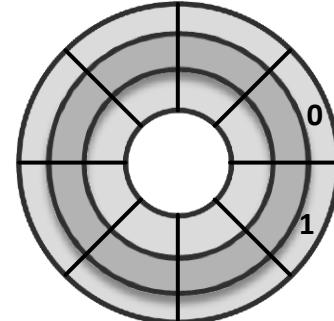
# Die Reihenfolge beim Protokollieren

**Begin- und End-Blöcke haben eine gewisse Größe und bieten Platz für reichlich Informationen.**

- **Check-Summen lösen das Reihenfolgeproblem.**

Versieht man die Begin- und End-Blöcke mit einer Check-Summe, dann kann geprüft werden, ob das Journal korrekt ist. Die Reihenfolge in der es geschrieben wird ist jetzt offensichtlich egal.

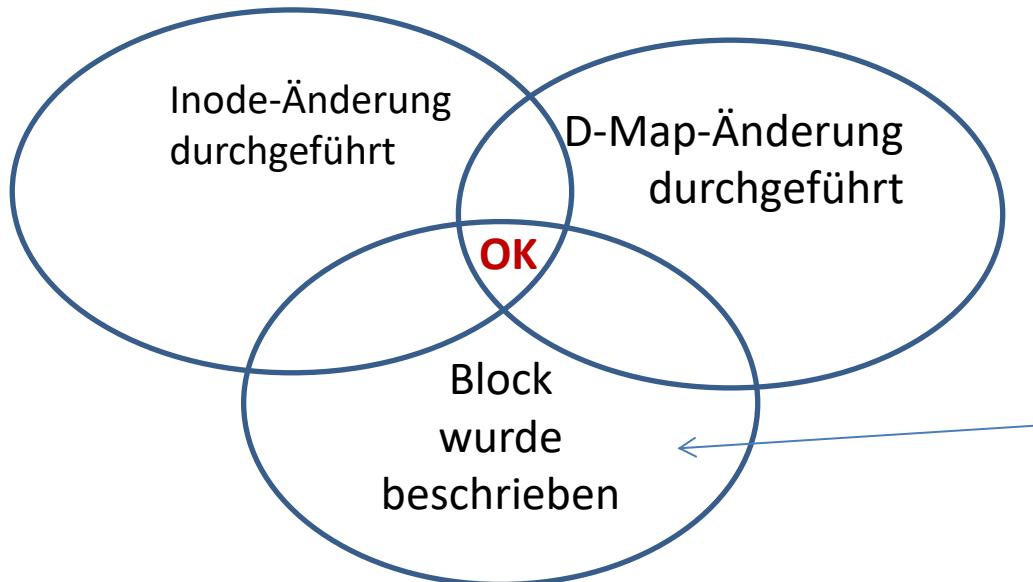
**Dies spart Zeit, eine ganze Umdrehung!  
(Linux ext4)**





# Meta-Data Journaling

Aus welchem Grund sollte man alle Daten zweimal schreiben?  
Schreiben ist kostspielig!



**konsistent und (unkritisch):**  
Daten wurden geschrieben,  
aber niemand weiß davon.



# Meta-Data Journaling (Linux ext3)

- Daten-Änderungen durchführen (diese werden nicht geloggt)
- Meta-Daten-Änderungen protokollieren
  - Journal Meta-Data Write
  - Journal Meta-Data Commit
- Meta-Daten-Änderungen durchführen (Checkpoint)
- Journal-Eintrag freigeben (Free)

**Achtung: Was geschieht, wenn Daten und  
Meta-Daten-Blöcke im gleichen  
Adressraum liegen?**



# Ein kleines Problem

1. Wir erzeugen eine „große“ Datei, so dass die Inode indirekte Block-Pointer benötigt.  
**Die Block-Pointer sind Meta-Daten und werden im Journal gesichert.**
2. Wir löschen diese Datei, die Blöcke werden wieder freigegeben.
3. Wir erzeugen eine „kleine“ Datei (:  
echo “Hello World” > tmp.txt)
4. Der zuvor freigegebenen Block (mit den Block-Pointern) könnte recycelt werden.
5. Die Daten (“Hello World”) werden geschrieben.  
**Dies sind keine Meta-Daten, sie werden nicht im Journal gesichert!**
6. **Wir ziehen nun vor dem eigentlich Free des Journals den Stecker!**
7. **Beim nächsten Hochfahren wird das File-System repariert.**

**Doch was steht jetzt in tmp.txt, eine Liste von Pointern?**



# Write-Ahead (Journaling)

**Das Journal muss erweitert werden!**



**Ein Revoke-Record sorgt dafür, dass nicht alle Änderungen im Falle eines Crashes wiederholt werden.**



# Konsistente Daten



# Prozesse benötigen Speicher

## **Teil III**

# **Virtualisierung des Hauptspeichers oder die Illusion vom eigenen RAM**



# Virtualisierung des RAM

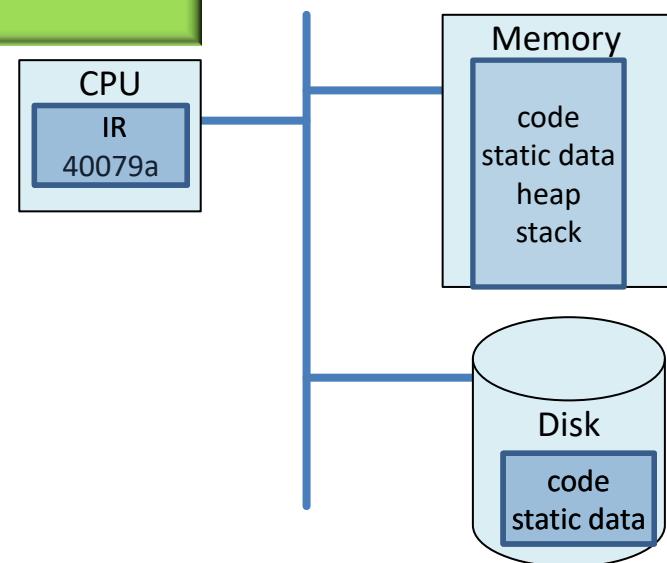
- Memory-Virtualisierung (**Spechervirtualisierung**)
  - Adressraum
  - Segmente und Page Frames
  - Free-Space Management
  - Swapping



# „Antike“

Damals, als das Betriebssystem lediglich eine Sammlung von Bibliotheken war!

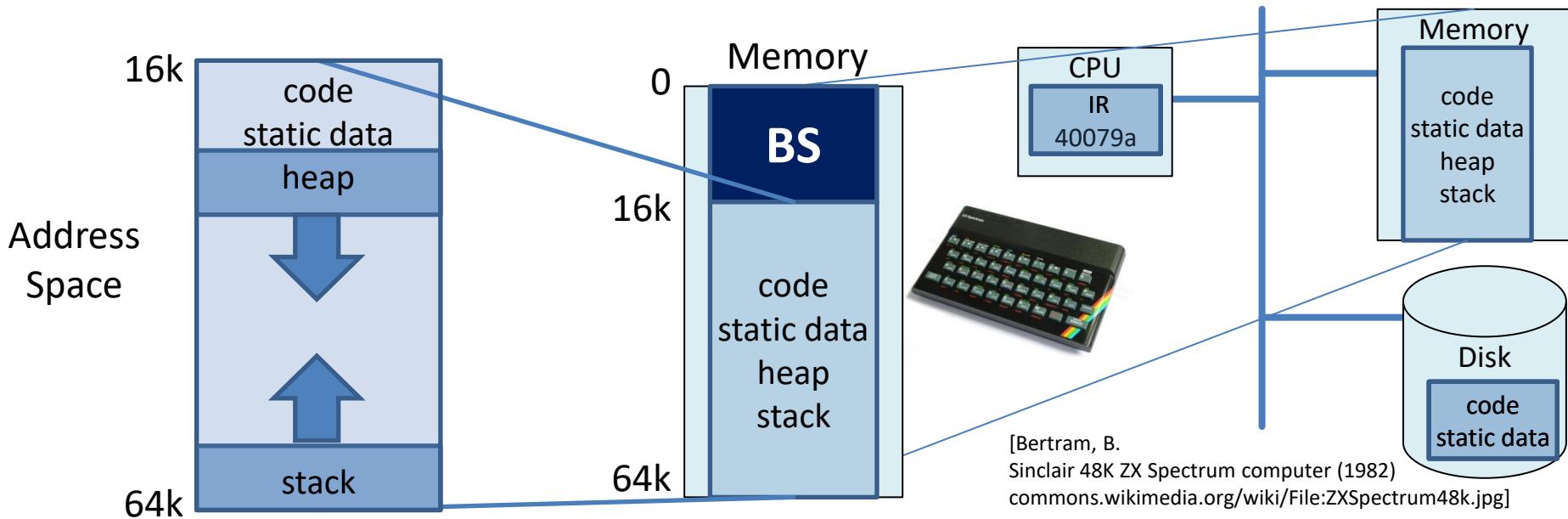
- Idee: Direkte Ausführung durch die CPU.
  - Betriebssystem erzeugt lediglich den Prozess (lädt das Programm und reserviert den Adressraum)
  - übergibt die Kontrolle an „main()“ (setzt den Instruction Pointer)





# „Antike“

Damals, als das Betriebssystem lediglich eine Sammlung von Bibliotheken war!





# Auf dem Weg in die Moderne

**Existiert mehr als ein Prozess, muss das Betriebssystem die Adressräume verwalten.**

## Ziele:

- Transparenz
- Schutz
- Effizienz

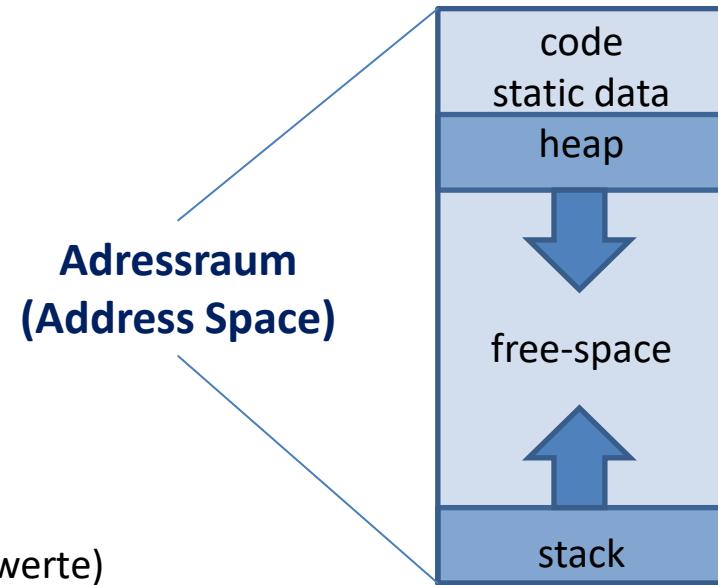
**Dies sind die zentralen Aufgaben einer Speichervirtualisierung!**



# Adressraum

Jeder Prozess bekommt seinen eigenen Adressraum.

- Programmcode
  - Funktionen, Methoden
- statische Daten
  - globale (static) Variablen, Konstanten
- dynamische Daten
  - **Heap:** (wächst von oben nach unten)  
Alles was **bewusst** erzeugt wird!  
(typischerweise durch new bzw. malloc)
  - **Stack:** (wächst von unten nach oben)  
Alles was **unbewusst** erzeugt wird!  
(lokale Variablen, Übergabeparameter, Rückgabewerte)





# Statische Daten, Heap und Stack

```
#include <stdio.h>
#include <stdlib.h>

static int on_static_data = 1;

int main(void) {
    int on_stack = 1;
    int *on_heap = (int*)malloc(sizeof(int));
    printf("Code beginnt bei: %p\n", (void*) main);
    printf("Data beginnt bei: %p\n", (void*) &on_static_data);
    printf("Heap beginnt bei: %p\n", (void*) on_heap);
    printf("Stack beginnt bei: %p\n", (void*) &on_stack);
    free(on_heap);
    return EXIT_SUCCESS;
}
```

- globale (static) Variablen, Konstanten
- Stack: lokale Variablen, Parameter, Rückgabewerte
- Heap: Alles was bewusst erzeugt wird!



# Statische Daten, Heap und Stack

```
#include <stdio.h>
#include <stdlib.h>

static int on_static_data = 1;

int main(void) {
    int on_stack = 1;
    int *on_heap = (int*)malloc(sizeof(int));
    printf("Code beginnt bei: %p\n", (void*) main);
    printf("Data beginnt bei: %p\n", (void*) &on_s
    printf("Heap beginnt bei: %p\n", (void*) on_he
    printf("Stack beginnt bei: %p\n", (void*) &on_
    free(on_heap);
    return EXIT_SUCCESS;
}
```

```
0000000000400586 <main>:
```

400586: 55	push	%rbp
400587: 48 89 e5	mov	%rsp,%rbp
40058a: 48 83 ec 10	sub	\$0x10,%rsp
40058e: c7 45 f4 01 00 00 00	movl	\$0x1,-0xc(%rbp)
400595: bf 04 00 00 00	mov	\$0x4,%edi

```
:> ./FindAddress
```

Code beginnt bei: 0x400586
Data beginnt bei: 0x60103c
Heap beginnt bei: 0x1c32010
Stack beginnt bei: 0x7ffc402203f4



# Adressraum

Wie groß kann der Adressraums werden?

- Adressgröße bis 32 Bit (4 Byte)  
=> Werte bis  $2^{32}-1$

$$1 \text{ GiB} = 1024^3 = 2^{30}$$



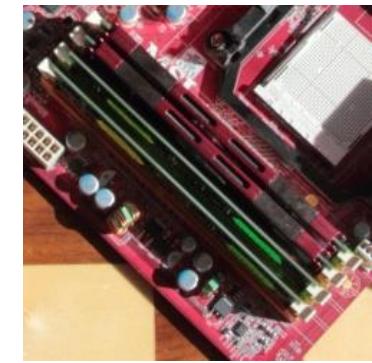
- Adressgröße bis 64 Bit (8 Byte)  
=> Werte bis  $2^{64}-1$



# Adressraum

Wie viel ist  $2^{64}$  ?

$$4 \text{ GiB} = 2^{32}$$

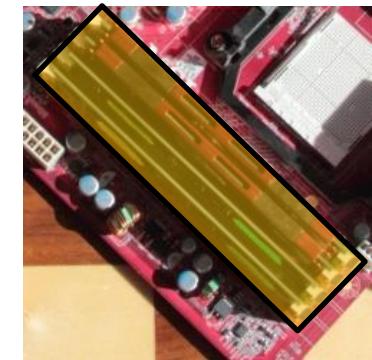


etwas mehr als  
1 Mrd  
16-GiB-Riegel



# Adressraum

Größe des Boards  
etwa  
 $1\text{km}^2$



$\approx 5000 \text{ mm}^2$



Der Adressraum ist groß genug, auch  
für die Zukunft!



# Virtualisierung des Adressraums

```
#include <stdio.h>
#include <stdlib.h>

static int on_static_data = 1;

int main(void) {
    int on_stack = 1;
    int *on_heap = (int*)malloc(sizeof(int));
    printf("Code beginnt bei: %p\n", (void*) main);
    printf("Data beginnt bei: %p\n", (void*) &on_stat);
    printf("Heap beginnt bei: %p\n", (void*) on_heap);
    printf("Stack beginnt bei: %p\n", (void*) &on_stack);
    free(on_heap);
    return EXIT_SU
}
```

```
:> ./FindAddress
Code beginnt bei: 0x400586
Data beginnt bei: 0x60103c
Heap beginnt bei: 0x1c32010
Stack beginnt bei: 0x7ffc402203f4
:>
```

## Oder, warum liegt der Stack soweit hinten?

```
0000000000400586 <main>:
400586: 55                      push  %rbp
400587: 48 89 e5                mov   %rsp,%rbp
40058a: 48 83 ec 10              sub   $0x10,%rsp
40058e: c7 45 f4 01 00 00 00 00  movl  $0x1,-0xc(%rbp)
400595: bf 04 00 00 00          mov   $0x4,%edi
```

Stack beginnt bei: 0x7ffc402203f4

$2^{48} = 0x800000000000$



# Virtualisierung des Adressraums: Erste Ideen

## Time-Sharing: CPU-Virtualisierung als Vorbild

- **Jeder Prozess bekommt das (komplette) RAM als Adressraum**
  - Kommt es zum Prozesswechsel, dann wird
    - der Adressraum des einen Prozesses auf die Festplatte gesichert und
    - der Adressraum des anderen Prozesses aus der Festplatte ins Ram geladen.



# Virtualisierung des Adressraums: Erste Ideen

**Leider keine gute Idee!**

- **Der Adressraum ist zu groß und Plattenzugriffe sind zu langsam**
  - Kontextwechsel würden „ewig“ dauern!
- **Die Speicheradressen stehen in der Binärdatei (im Binary)**
  - Jedes Programm müsste für die spezifische Hardware-Konfiguration des Anwenders kompiliert werden.



# Virtualisierung des Adressraums

## Space-Sharing:

- Jeder Prozess bekommt seinen eigenen Anteil am RAM.
- Während des Kontextwechsels verbleibt der Adressraum im RAM.
- Die Adressen, die ein Prozess nutzt, sind keine „physischen“ Adressen, sondern „**virtuelle**“ Adressen.

**Aufgabe des Betriebssystems ist es,  
dies zu organisieren.**



# Virtualisierung des Adressraums

## Ziele:

- Transparenz
  - Ein Prozess darf von dieser Abbildung nichts mitbekommen.
- Schutz
  - Die Abbildung muss dafür sorgen, dass keine virtuelle Adresse in den Adressraum eines anderen Prozesses verweist.
  - Kein Prozess darf auf die Abbildung Einfluss nehmen.
- Effizienz
  - Es muss schnell gehen! (**Limited Direct Execution**)



# Virtualisierung des Adressraums unter Limited Direct Execution

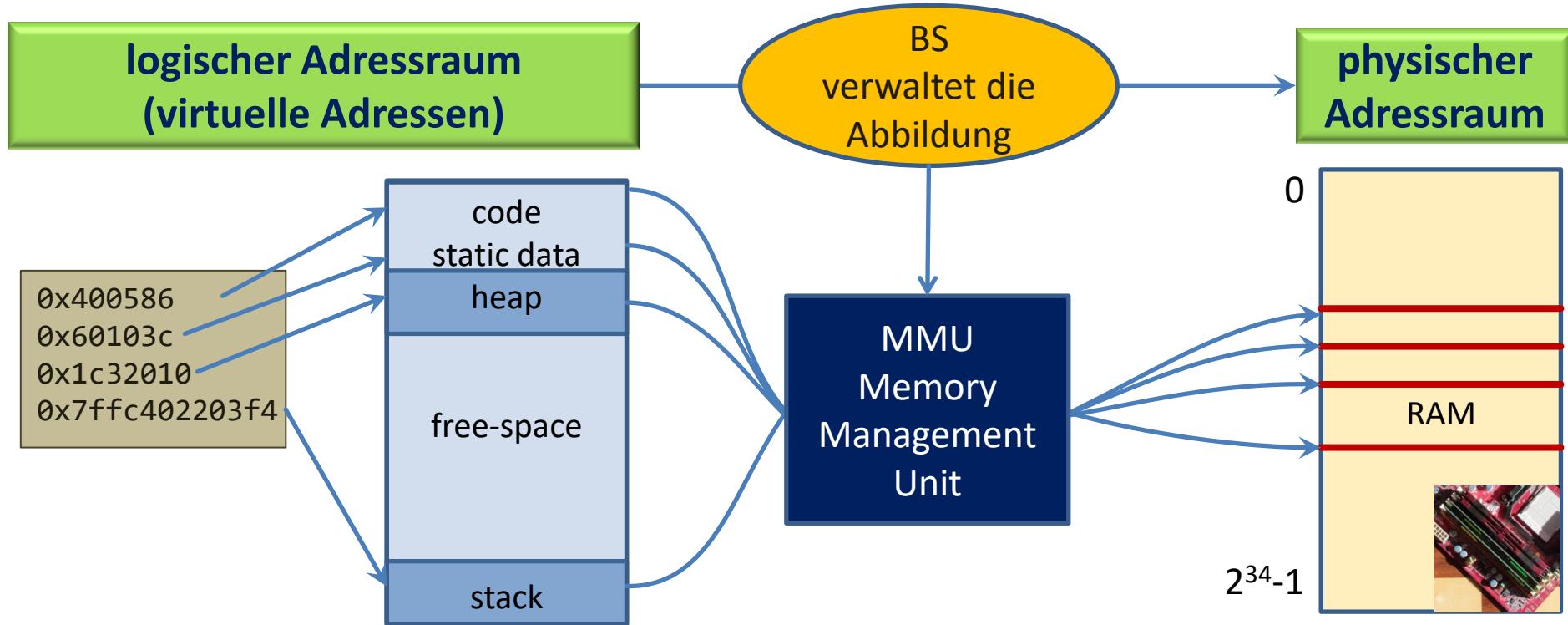
**MMU:**  
**Memory Management Unit**  
**Hardware-Baugruppe in der CPU (Register)**

- Kernel Mode:
  - Betriebssystem kann auf jede Adresse zugreifen
  - Betriebssystem kann Register in der MMU ändern
- User Mode:
  - Jeder Speicherzugriff geht durch die MMU

**Aufgabe des BS:**  
MMU Register so setzen,  
dass kein Zugriff auf fremde  
Speicherstellen möglich  
wird!



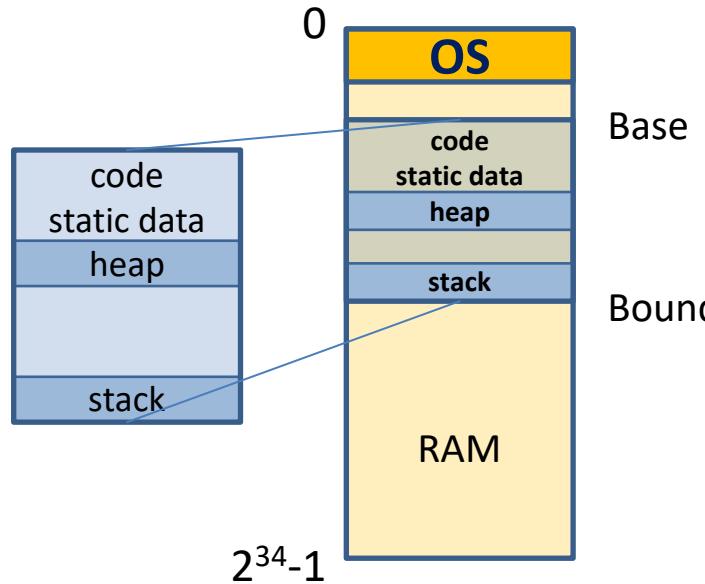
# Virtualisierung des Adressraums





# Memory Management Unit: Erste Idee

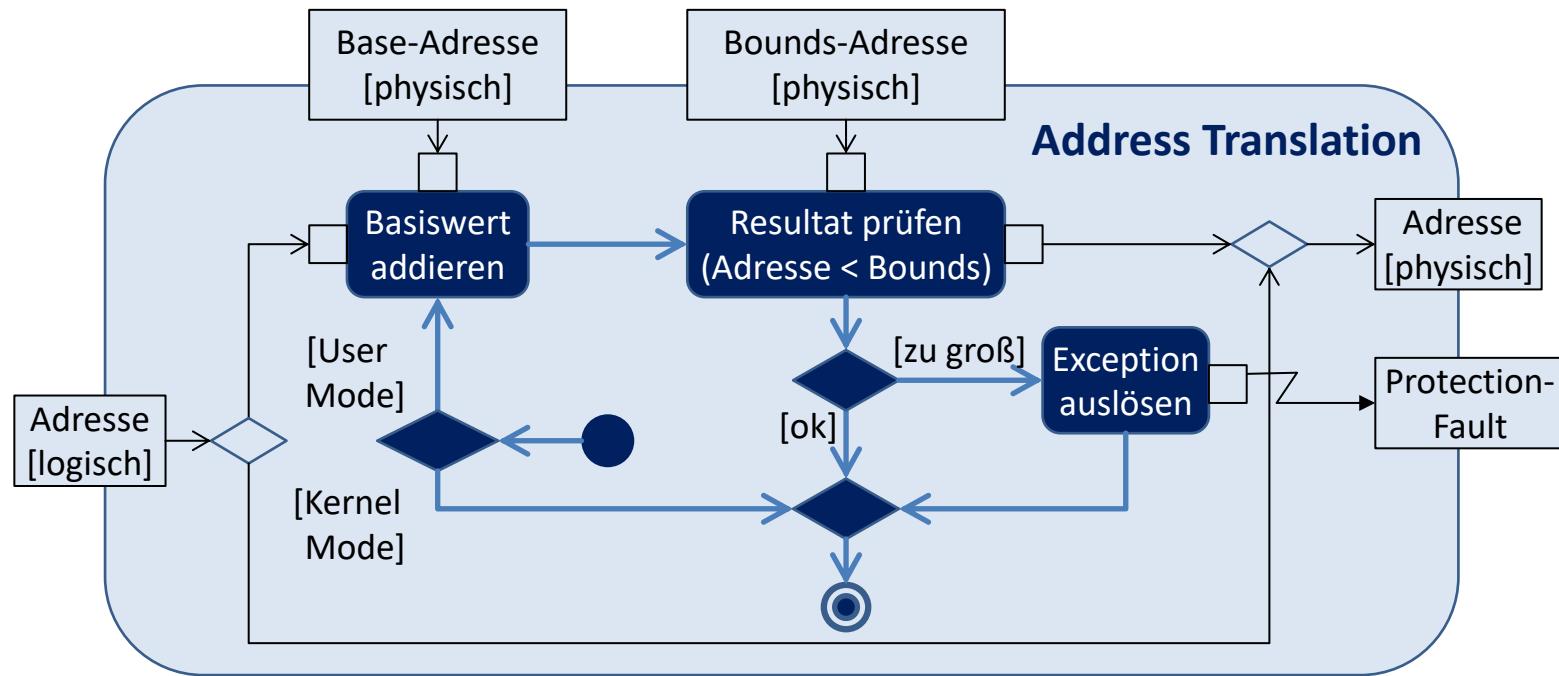
## Base-and-Bounds



- Die Basisadresse des physischen Adressraums wird im Base-Register gespeichert (**erste gültige Adresse**).
- Die Größe des physischen Adressraums wird im Bounds-Register gespeichert (**z.B.: erste ungültige Adresse**).



# Base-and-Bounds:Hardware-basierte Adressumrechnung





Hardware  
Control Unit

## Timer Interrupt

Software  
Dispatcher

- Sichert die Informationen, um den Prozess wieder zu starten.
  1. wechselt in den Kernel Mode
  2. kopiert den Inhalt zentraler Register in den Kernel Stack
  3. startet den Handler
  8. restauriert den Registerinhalt und wechselt in den User Mode
  9. führt den neuen Prozess aus
- Führt nach Bedarf den Prozesswechsel durch.
  4. sichert weitere prozessspezifische Register **Base-Register** und **Bounds-Register** (im PCB)
  5. wählt einen anderen Prozess
  6. restauriert die Registerinhalte insbesondere **Base** und **Bounds** und setzt den Stack Pointer des Kernel Stacks entsprechend
  7. kehrt zurück **IRET**



# Base-and-Bounds

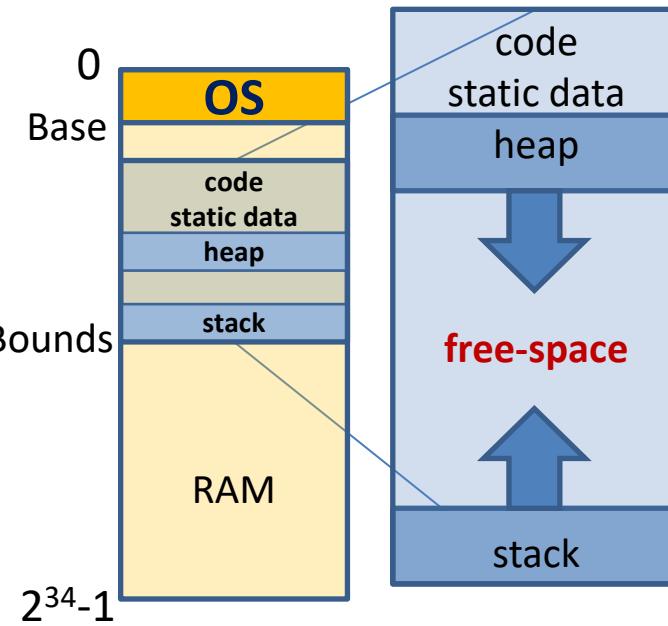
## Vorteile:

- Transparenz:
  - Adressumrechnung erfolgt durch die Hardware während des Zugriffs!
- Schutz:
  - Hardware stellt sicher, dass bei Missachtung das Betriebssystem informiert wird, dieses terminiert üblicherweise den Prozess
  - das Setzen der Base- und Bound-Register sind privilegierte Operationen
- Effizienz
  - lediglich zwei weitere Register
  - sehr schnell (Addieren und Vergleichen) lässt sich sogar parallelisieren



# Base-and-Bounds

## Nachteile:



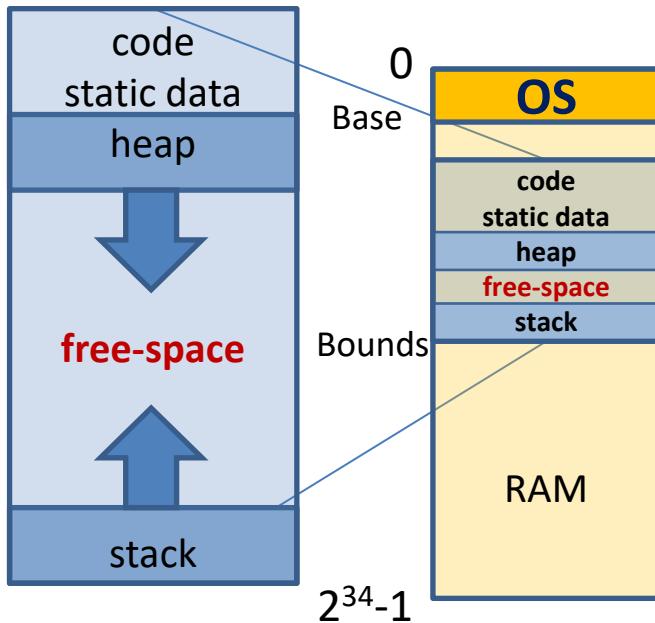
### Ein immenser Speicherverbrauch!

- Der komplette Adressraum liegt immer im Hauptspeicher.
- Das meiste davon wird fast nie genutzt.
- Läuft dasselbe Programm parallel, dann gibt es mehrere Ausprägungen des gleichen Adressraums.



# Memory Management Unit: Zweite Idee

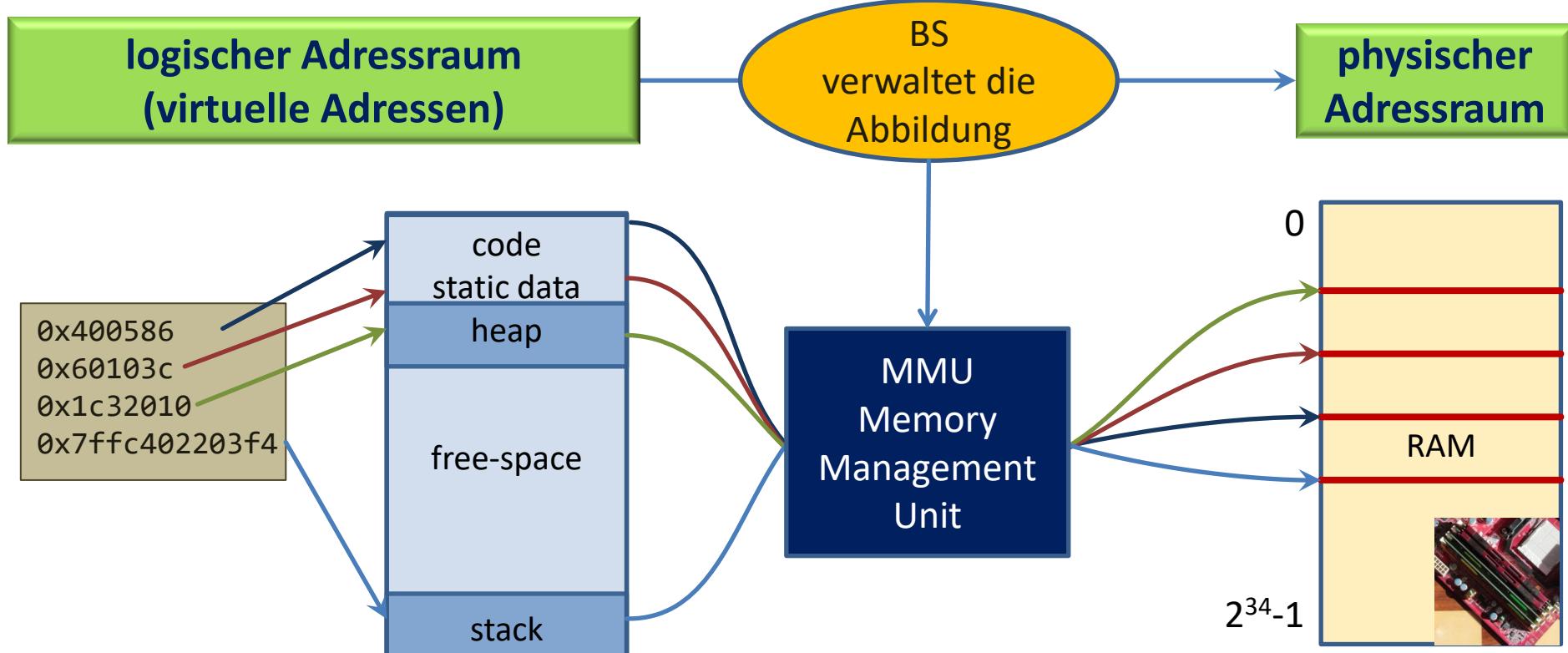
## Segmentation



- Warum sollte der physische Adressraum ein Abbild des logischen Adressraums sein?
- **Jedes Segment bekommt seine eigene Abbildung mit Base- und Bounds-Register.**
  - Code, Static Data, Heap und Stack, eventuell auch mehr
- Segmente sollte man sharen können!
  - Etwa gemeinsames Code-Segment bei parallelen Prozessen, die auf demselben Programm basieren.



# Virtualisierung des Adressraums





# Segmentation (Ziele)

**Bessere Speicherausnutzung!**

- Dynamische Anpassung der Größe von Heap und Stack!
  - Heap über speziellen System-Call (falls kein Speicher mehr vorhanden)
  - Stack implizit, durch überwachten Interrupt.
- Sharing von Segmenten bei Aufrechterhaltung des Schutzes:
  - Um eine gemeinsamen Zugriff auf Segmente zu ermöglichen, werden Protection-Bits für die Segmente eingeführt: **read, write, execute**



# Segmentation (Realisierung)

**Wie übersetzt die MMU eine segmentierte virtuelle Adresse in eine physische Adresse?**

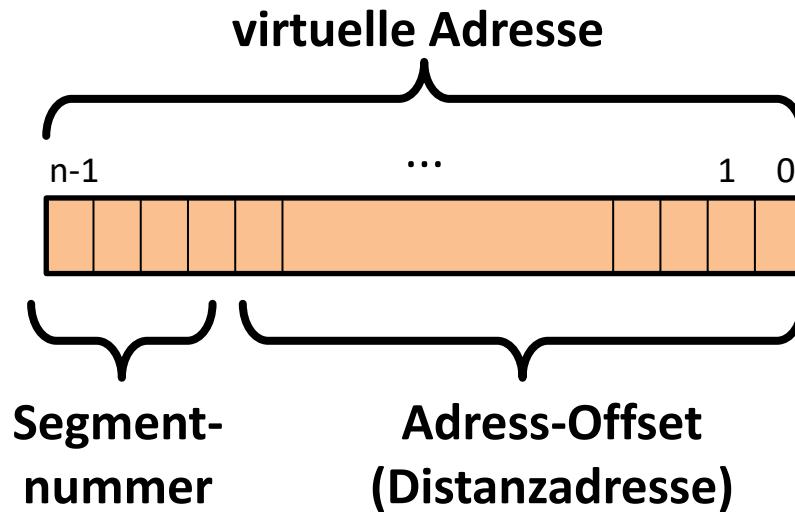
- Segment identifizieren.
- Base addieren
- Bounds prüfen

**Zusätzlich muss vor der Ausführung des Befehls noch die Berechtigung geprüft werden.**



# Erster Schritt: Segment identifizieren

Idee: Das Segment wird durch die virtuelle Adresse identifiziert!



Für 4 Segmente benötigt man 2 Bits.

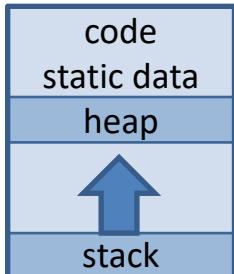


# Zweiter Schritt: Base addieren

Fast so simpel wie in der ursprünglichen Variante von  
Base-and-Bounds.

Zu beachten ist lediglich, dass nicht jedes Segment in  
die gleiche Richtung „wächst“!

- Der Stack wächst „negativ“.



Auch diese Information (Wuchsrichtung)  
muss in der MMU verwaltet werden.



# Segmentation die MMU

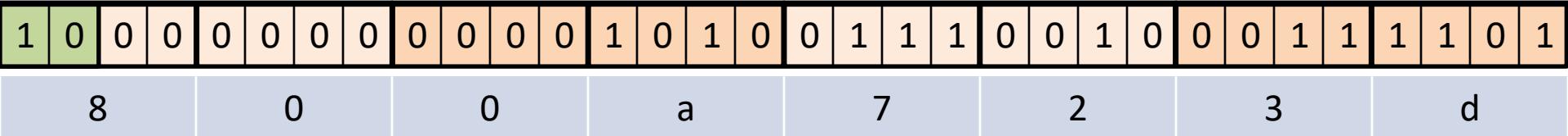
Die MMU beinhaltet für jeden Prozess eine eigene Segmenttabelle

	Segment	Base (Adresse)	Bounds (Adresse)	wächst positiv	R W X
Code	0 (256 KiB)	0x10 10 00 00	0x10 14 00 00	1	1 0 1
static Data	1 (256 KiB)	0x40 a0 00 00	0x40 a4 00 00	1	1 1 0
Heap	2 ( 1 MiB)	0x0f 00 00 00	0x0f 10 00 00	1	1 1 0
Stack	3 ( 8 MiB)	0x00 ef ff ff	0x00 6f ff ff	0	1 1 0

Jede Zeile stellt einen Segmentdeskriptor dar!



# Beispiel einer Adressumrechnung



- virtuelle Adresse 0x800a723d

- Segmentnummer 2

- Adress-Offset 0xa723d

1. 0x0f000000 (Base)

+ 0x000a723d (Offset)

= 0x0f0a723d (physische Adresse)

Segment	Base (Adresse)	Bounds (Adresse)	wächst positiv
2	0x0f000000	0x0f100000	1

Segmentdeskriptor

2. 0x0f0a723d < 0x0f100000 => Adresse im Range ✓



# Beispiel einer Adressumrechnung



- virtuelle Adresse 0xffffe723d
- Segmentnummer 3
- Adress-Offset 0x3ffe723d

**Achtung:** Der Stack wächst von unten nach oben. Wir sind also am Abstand zwischen der virtuellen Adressen und der größten gültigen virtuellen Adresse in diesem Segment interessiert.



# Beispiel einer Adressumrechnung



1.  $0x3fffffff$  (virtuelle Adressgröße  $2^{30}$  d.h. zwischen  $0..2^{30}-1$ )

-  $0x3ffe723d$  (Adress-Offset)

=  $0x00018dc2$  (Delta)

2.  $0x00efffff$  (Base)

-  $0x00018dc2$  (Delta)

=  $0x00ee723d$  (physische Adresse)

Segmentdeskriptor

Segment	Base (Adresse)	Bounds (Adresse)	wächst positiv
3	$0x00efffff$	$0x006fffff$	0

2.  $0xee723d > 0x6fffff \Rightarrow$  Adresse im Range ✓



# Segmentation

## Vorteile:

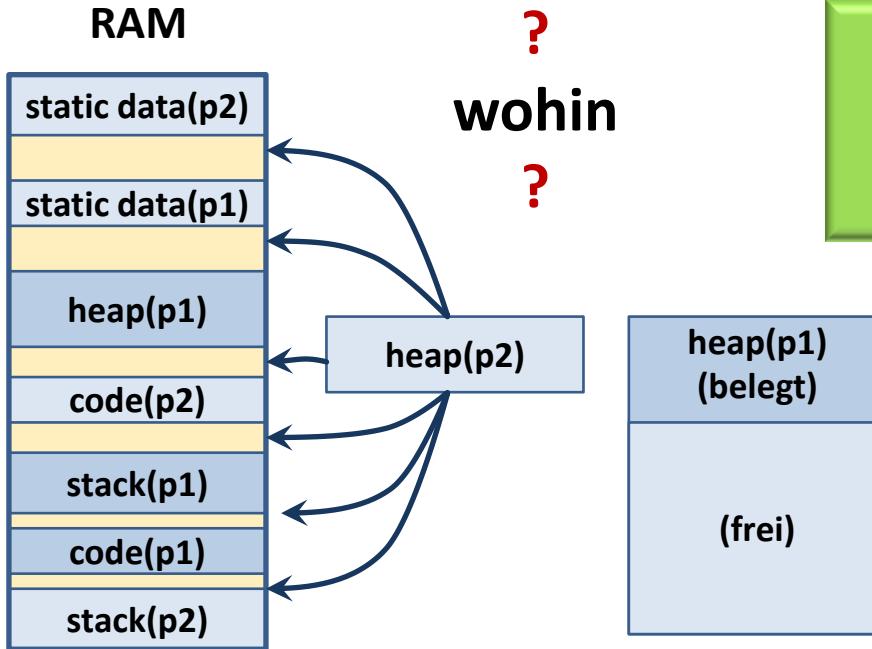
- deutlich geringerer Verbrauch an realem Speicher.
- die reale Speichergröße kann bei Bedarf angepasst werden.
- individueller Schutz der Segmente (rwx).
- Segmente können geteilt werden.
- Segmente können verschoben werden.

## Nachteil:

**Noch immer befindet sich das ganze Segment als eine zusammenhängende (eventuell sehr große) Einheit im Hauptspeicher.**



# Zusammenhängende große Einheiten



**Problem: Große Mengen an nicht nutzbarem aber freiem Speicher!**

## Fragmentierung

- extern: außerhalb eines „Segments“
- intern: innerhalb eines „Segments“



# Memory Management Unit: Dritte Idee

**Weder Base- noch Bounds-Register noch Segmente!**

RAM

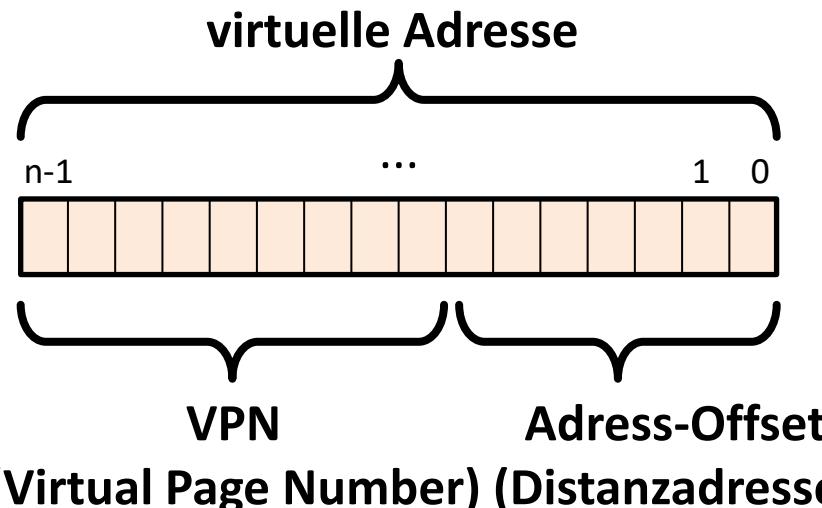
Page Frame 0	
Page Frame 1	
Page Frame 2	
Page Frame 3	
Page Frame 4	
Page Frame n-1	
Page Frame n	

Statt dessen, viele kleine, gleich große Schnipsel, die den gesamten Speicher füllen,  
**„Page Frames“**



# Page Frames

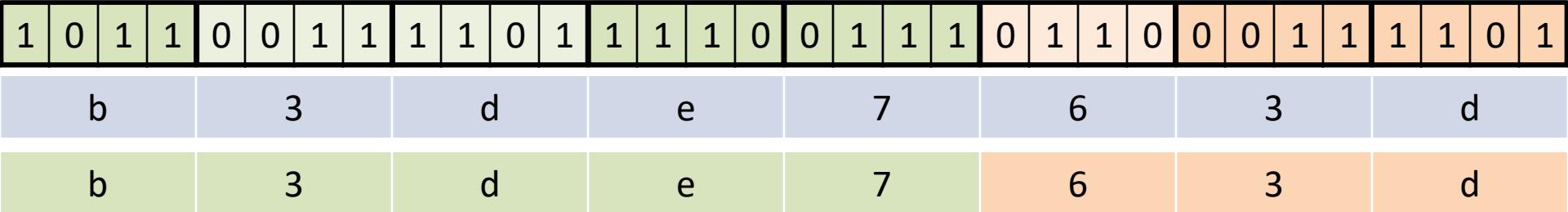
Wie zuvor das Segment, wird jetzt das Frame durch die virtuelle Adresse identifiziert!



Das Verhältnis hat sich geändert.  
**kleiner Offset**  
**dafür**  
**viele Frames**



# Adressumrechnung durch die MMU

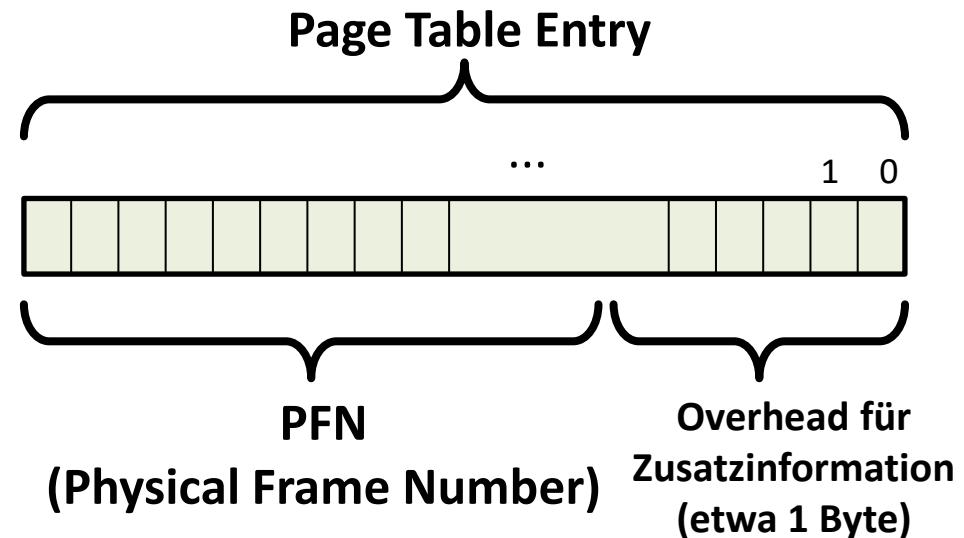
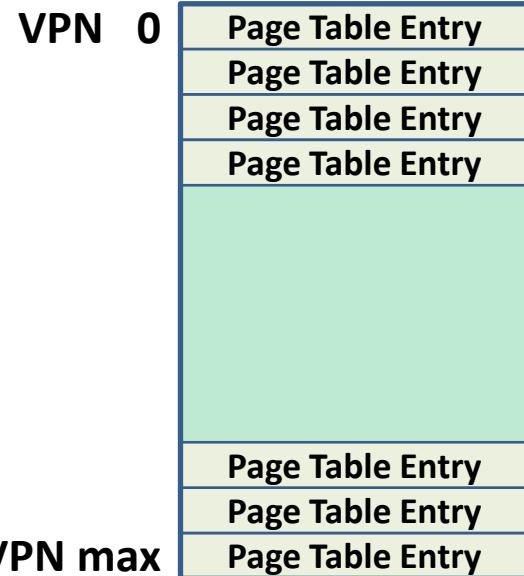


- virtuelle Adresse: 0xb3de763d
  - 20 bit VPN und 12 bit Offset
    - VPN: 0xb3de7  0x4fd6 PFN (Physical Frame Number)
    - Offset : 0x63d 0x63d Offset
- 0x04fd663d** physische Adresse



# Page Frames: Transformation

Naheliegend und simpel! Ein Array (Page Table) pro Prozess





# Page Table Entry

## Typische Zusatzinformationen

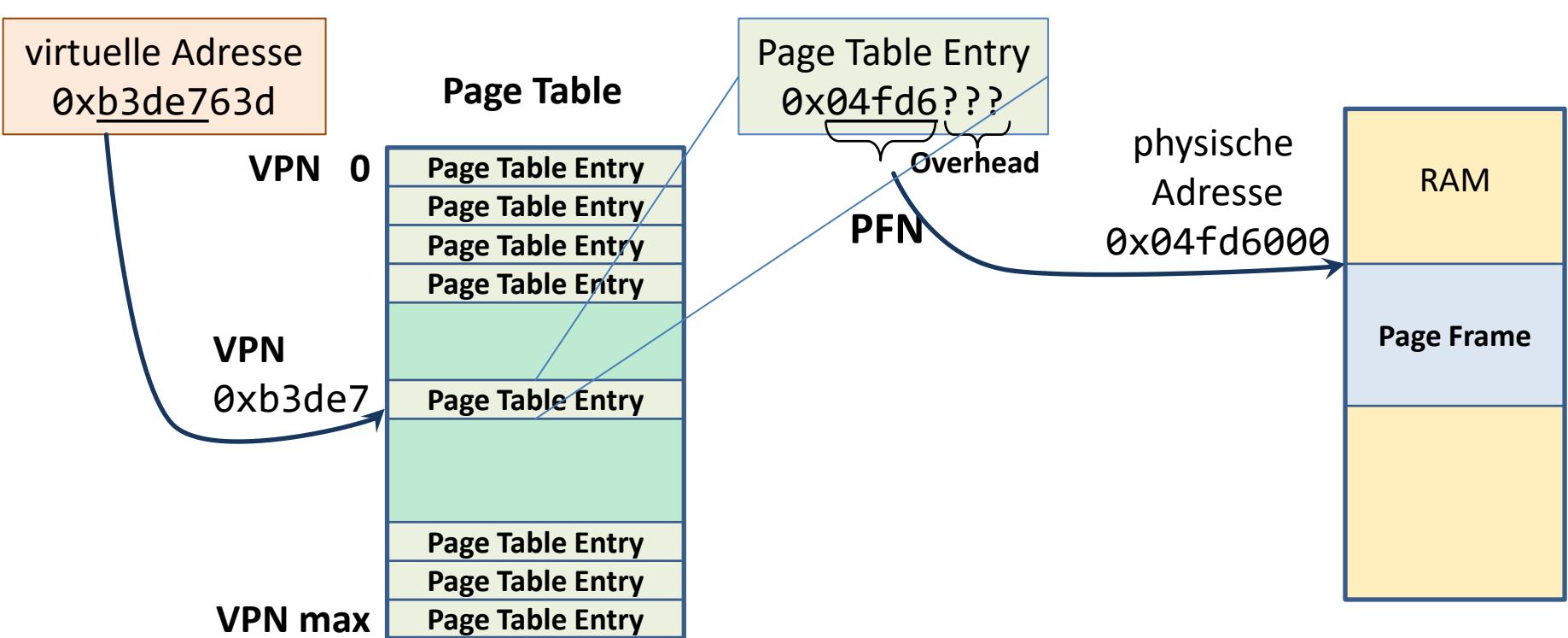
- ein Valid-Bit
  - „true“ die Seite ist zugewiesen
  - „false“ die Seite ist nicht zugewiesen
- Protection-Bits (rwx)
  - auf die Seite darf lesend oder schreibend zugegriffen werden
  - die Seite enthält ausführbaren Code



**Jeder widerrechtliche Zugriff, bzw. Zugriff auf eine nicht zugewiesene Seite führt zu einem Interrupt!**



# Page Frames: Transformation





# Page Frames: Transformation

**Achtung: Limited Direct Execution!**

**Die Adresstransformation muss mittels Hardware realisiert werden.**

- Hardware und Betriebssystem müssen aufeinander abgestimmt sein.
- Die Größe der Frames wird von der Hardware vorgegeben.
- Die Tabellen müssen gefunden werden, d.h. die Startadresse der Tabelle liegt in einem Register. (Unter x86 ist dies das CR3-Register)
- **Dies muss im Kontextwechsel berücksichtigt werden!**



# Page Frames: Größenbetrachtungen

- x86 und x86-64 (32/64-bit): ein Frame hat 4 KiB = 4096 Byte  
$$4096 = 4 \times 2^{10} = 2^{12}$$
  - => Adress-Offset von 12 bit
  - => VPN 20/52-bit
  - $2^{20} = 1.048.576$  adressierbare Seiten bei einem 32-bit Betriebssystem
  - $2^{52} = 4.503.599.627.370.496$  adressierbare Seiten bei einem 64-bit Betriebssystem
- [real x86-64] die Größe einer virtuellen Adresse ist begrenzt auf 48 Bit**
- $2^{36} = 68.719.476.736$  adressierbare Seiten.**



# Page Frames: Größenbetrachtungen

**Speicherbedarf für eine vollständige Zuordnung (VPN -> PFN)  
pro Prozess**

=

**Anzahl der möglichen VPNs (adressierbare Seiten) x  
Speicherbedarf pro PTE**



# Page Frames: Größenbetrachtungen

Kodierungsgröße einer PFN (Bytes pro PFN)

=

kleinste ganze Zahl größer-gleich  $\log_2(\text{Anzahl der Frames}) / 8$

- 32-bit-Architektur und 4 GiB RAM  $\Rightarrow (4 \times 2^{30}) / 2^{12} = 2^{20}$  Frames

$$\text{Bytes pro PFN} = \left\lceil \frac{\log_2(2^{20})}{8} \right\rceil = \left\lceil \frac{20}{8} \right\rceil = 3$$

- 64-bit-Architektur und 128 GiB RAM  $\Rightarrow (128 \times 2^{30}) / 2^{12} = 2^{25}$  Frames

$$\text{Bytes pro PFN} = \left\lceil \frac{\log_2(2^{25})}{8} \right\rceil = \left\lceil \frac{25}{8} \right\rceil = 4$$



# Page Frames: Größenbetrachtungen

Speicherbedarf für eine vollständige Zuordnung pro Prozess  
bei einem Overhead von 1 Byte.

- 32-bit-Architektur und 4 GiB RAM (4 Byte pro PTE)  
 $2^{20} \times (3+1) \Rightarrow 4 \text{ MiB}$
- 64-bit-Architektur und 128 GiB RAM (5 Byte pro PTE)  
 $2^{52} \times (4+1) = 20 \times 2^{50} \Rightarrow 20 \text{ PiB}$
- **real (x86-64) 8 Byte pro PTE aber nur  $2^{48}$  virtuelle Adressen**  
 $2^{36} \times 8 = 2^9 \times 2^{30} = 512 \text{ GiB}$

[Ntdebugging Blog: Understanding !PTE , Part 1: Let's get physical, 2010  
[blogs.msdn.microsoft.com/ntdebugging/2010/02/05/understanding-pte-part-1-lets-get-physical/](http://blogs.msdn.microsoft.com/ntdebugging/2010/02/05/understanding-pte-part-1-lets-get-physical/)]



# Page Frames: Größenbetrachtungen

- **32-bit-Architektur und 4 GiB RAM**
  - Arrays der Größe 4 MiB sind problemlos zu verarbeiten
  - Die Page Table eines Prozesses kann problemlos im RAM abgelegt werden
  - Beim Prozesswechsel muss lediglich die Startadresse des aktuellen PT aus dem CR3-Register gesichert und die Startadresse des neuen Prozesses aus dessen PCB geladen werden.
- **64-bit-Architektur und 128 GiB RAM**
  - **Arrays der Größe 20 PiB sind alles andere als eine gute Idee!**
  - Auch 512 GiB ist keine gute Idee!



# Page Frames

## Vorteile:

- keine Fragmentierung des RAMs:  
der physische Adressraum ist ein Vielfaches der Größe eines Page Frames.
- sehr flexibel:  
die Zuordnung von Speicher erfolgt in relativ kleinen Einheiten,  
diese lassen sich einfach allokieren und wieder freigeben.

**Das Betriebssystem verwaltet eine Liste aller Page Frames!  
Und für jeden Prozess die Zuordnung der Page Frames!**





# Page Frames

## Nachteil:

- weiterhin interne Fragmentierung:

```
char *text = (char*) malloc(4097); // benötigt 2 Frames
```

- deutlich aufwendiger im Adresszugriff:

- eine zusätzliche Indirektionsstufe in der Berechnung und
  - ein zweiter Speicherzugriff für die Ermittlung des PTE

- Größe:

- Page Tables können verdammt groß werden und brauchen Platz.

**Das Konzept ist noch ausbaufähig!**



# Ein kurzer Interrupt!

## Nachteil:

- weiterhin interne Fragmentierung:

```
char *text = (char*) malloc(4097); // benötigt 2 Frames
```

**Wer kümmert sich um die interne Fragmentierung?  
Man braucht eher selten eine ganze Seite!**

**Was verbirgt sich hinter malloc und free?**



# Ein kurzer Interrupt!

Was verbirgt sich hinter malloc und free?

Bibliotheken, die den Heap des aktuellen Prozesses managen und bei Bedarf vom Betriebssystem neuen Speicher anfordern, und freien Speicher wieder zurückgeben.



# Free-Space Management

- **void \*malloc(size\_t size)**
  - Besorgt size-Bytes zusammenhängenden virtuellen Speicher und liefert dessen Anfangsadresse zurück.
- **void free(void \*ptr)**
  - Gibt den zuvor allokierten Speicher wieder frei. Eine Größenangabe ist nicht erforderlich!

**Offensichtlich benötigt man hierfür Verwaltungsinformation!**



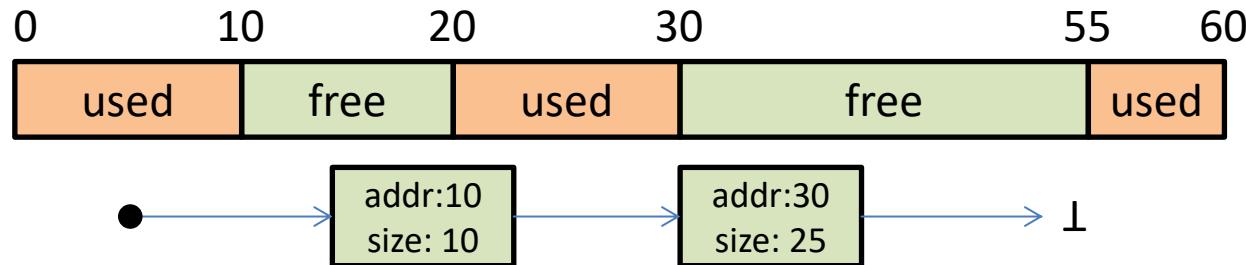
## Typischen Informationen:

- die Größe der zu verwaltenden Speichers,
  - die Mindestgröße für allozierbare Einheiten,
  - eine Liste aller allozierten und freien Einheiten (**Free-List**),
  - Informationen darüber, welche Einheiten zusammen gehören.
- **Zentrale Rahmenbedingungen:**
- **Der zu verwaltende Speicher ist zusammenhängend.**
  - **Einmal allozierter Speicher kann nicht verschoben werden.**



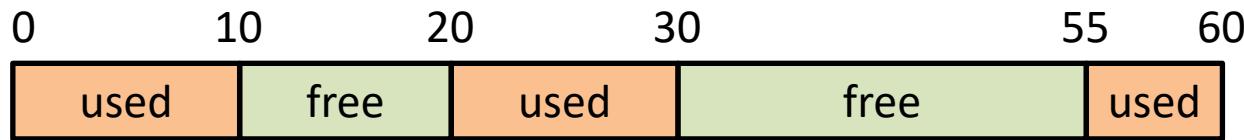
# Free-List

- eine Liste aller allokierten und freien Einheiten
  - **Variante1:** Die Liste besteht aus Knoten, die den verbrauchten Teil des Speichers beschreiben.
  - **Variante2:** Die Liste besteht aus Knoten, die den zusammenhängende Teil des freien Speichers beschreiben.  
**(Vorteil: Die Knoten können im freien Teil gespeichert werden)**





# Free-List



```
char *text = (char*) malloc(20);
```

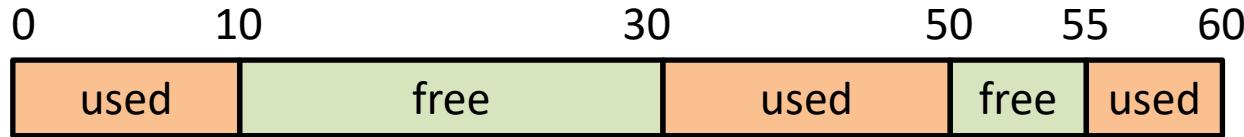




# Free-List



free(20);





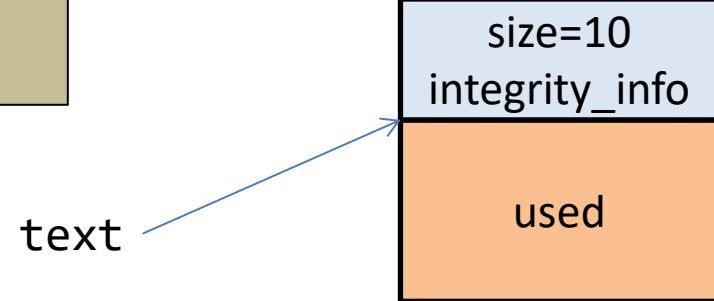
# Free-List

- Informationen darüber, welche Einheiten zusammen gehören

free(20) hat nur 10 Byte und nicht 30 freigegeben

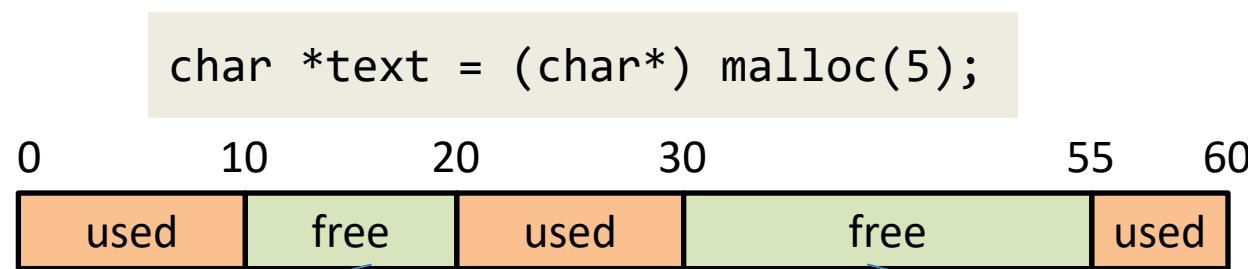
→ Jedes Stück allokiert Speicher bekommt einen eigenen Header, in dem diese Informationen gespeichert werden!

```
char *text = (char*) malloc(10);
```





# Free-Space Management (Strategie)



- Best Fit
- Worst Fit
- First Fit
- Next Fit
- ...



# Free-Space Management (Binary Buddy)

[Evans, J. A Scalable Concurrent malloc(3) Implementation for FreeBSD. 2006 (jemalloc)]

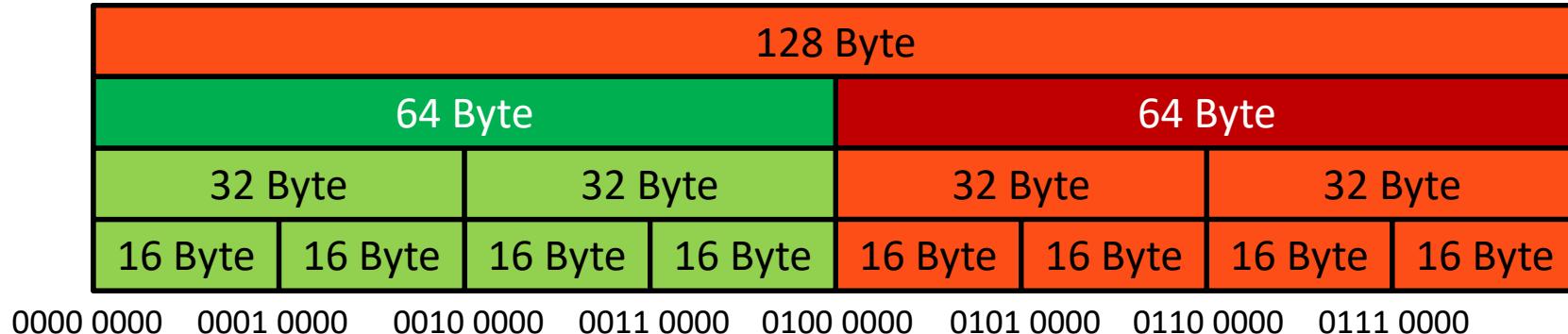
Kategorie	Uterkategorie	Größe
small	tiny	2-8 Byte
	quantum	16-512 Byte
	sub-page	1-2 KiB
large		4 KiB - 1 MiB
huge		2 MiB -

- Vom Betriebssystem angefordert werden stets 2 MiB oder mehr.
- Die kleinste Einheit, die allokiert wird ist ein „Quantum“ à 16 Byte

- Werden n-Bytes benötigt, dann wird ein freier Block der Größe  $2^k$  gewählt, so dass  $2^{k-1} < n \leq 2^k$



# Free-Space Management (Binary Buddy)



- allokiert werden:
  - 10 Byte (0000 0000)
  - 20 Byte (0010 0000)
  - 9 Byte (0001 0000)
  - 33 Byte (0100 0000)
- freigegeben werden werden:
  - 9 Byte (0001 0000)
  - 20 Byte (0010 0000)
  - 10 Byte (0000 0000)



# Free-Space Management (Binary Buddy)

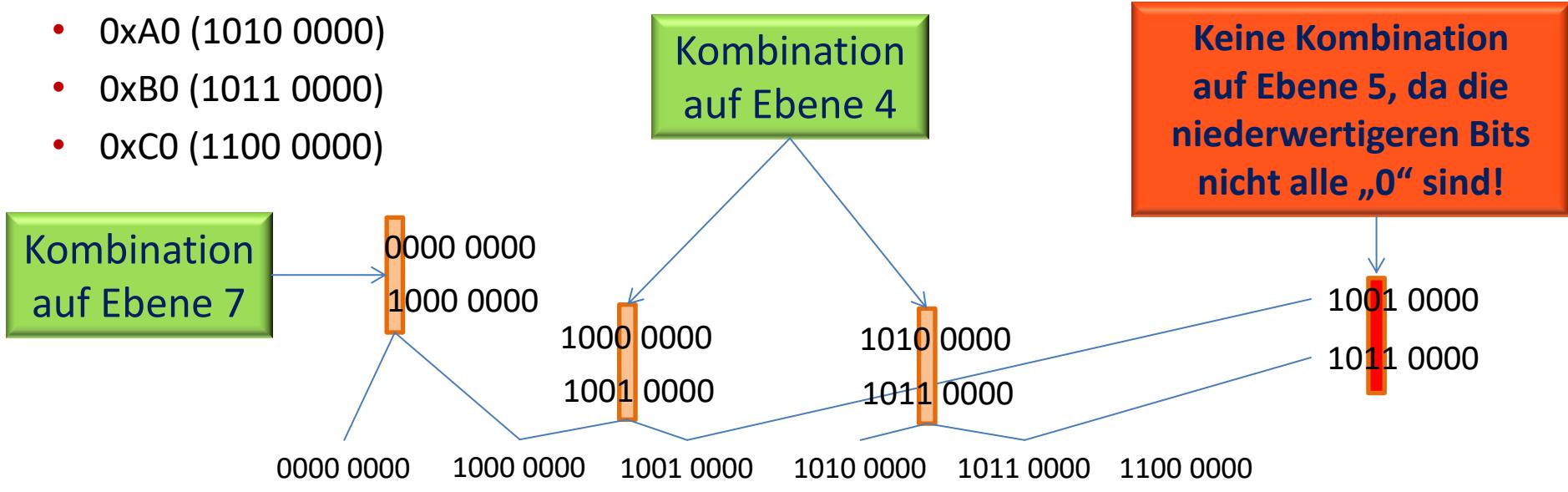
Offensichtlich gilt:

- Eine Adresse repräsentiert einen Buddy der Ebene n, wenn die n niedrigwertigsten Bits „0“ sind!
- Zwei Buddies der Ebene n können zusammengefasst werden, gdw. sie sich nur im n-ten Bit unterscheiden! Sie bilden dann einen Buddy der Ebene n+1. Dieser wird durch die kleinere der beiden Adressen repräsentiert.
- Ein Buddy der Ebenen n repräsentiert einen Speicherbereich von  $2^n$  Byte.



# Free-Space Management (Binary Buddy)

- 0x00 (0000 0000)
- 0x80 (1000 0000)
- 0x90 (1001 0000)
- 0xA0 (1010 0000)
- 0xB0 (1011 0000)
- 0xC0 (1100 0000)





# Page Frames (zurück vom Interrupt)

## Nachteil:

- Weiterhin interne Fragmentierung:

```
char *text = (char*) malloc(4097); // benötigt 2 Frames
```

- Deutlich aufwendiger im Adresszugriff:

- eine zusätzliche Indirektionsstufe in der Berechnung und
  - **ein zweiter Speicherzugriff für die Ermittlung des PTE**

- Größe:

- Page Tables können verdammt groß werden und brauchen Platz.

**Das Konzept ist noch ausbaufähig!**



# Page Frames: TLBs

Adresstransformationen müssen schnell gehen!

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int count = 1000;
    int *x = (int*) malloc(count*sizeof(int));
    for (int i = 0; i < count; i++)
        x[i] = i;
    ...
}
```

- 4000 Byte im selben Segment
- 1000 Speicherzugriffe
  - 1000 Transformation
  - **nochmals 1000 Speicherzugriffe für die Ermittlung ein und desselben PTE**



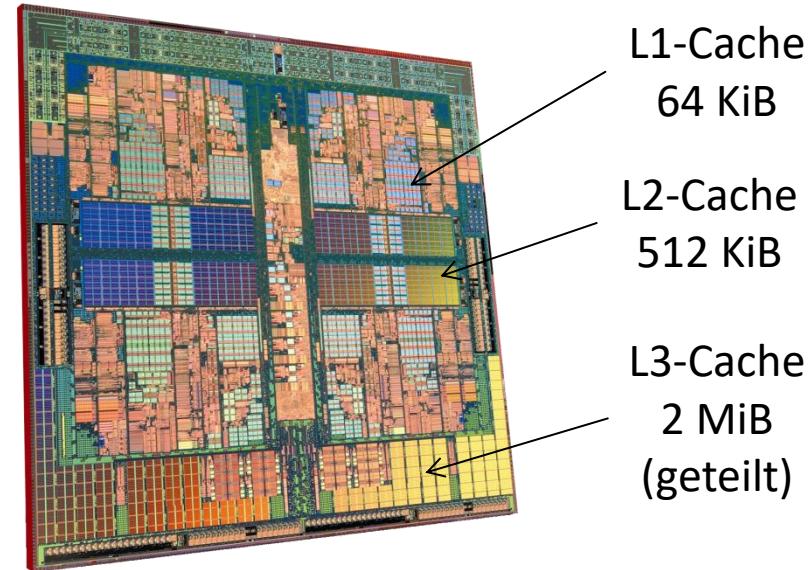
# Page Frames: TLBs

[www.7-cpu.com/cpu/K10.html]

- TLB L1 48 Einträge
- TLB L2 512 Einträge

## Translation Lookaside Buffer

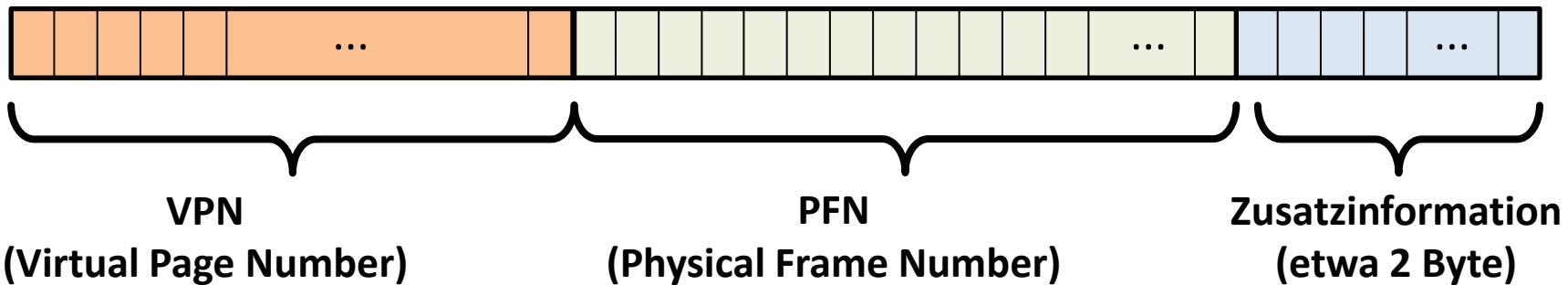
**Zusätzlicher Hardware Cache,  
in dem die wichtigsten  
Übersetzungen gespeichert  
werden!**



Die eines AMD Phenom™ Quad-Core  
[Bild: Advanced Micro Devices, Inc.  
(AMD)]



# TLB: Einträge

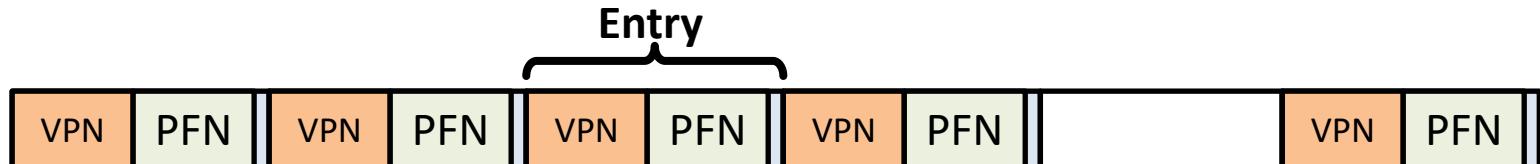


- Beinhaltet VPN und PFN
- Plus Zusatzinformationen
  - sowohl für die Übersetzung von VPN -> PFN
  - als auch die inverse Operation PFN ->VPN



# TLB: Zugriff

TLB



- Die CPU ist in der Lage parallel auf die Einträge zuzugreifen.
  - sowohl für die Übersetzung von VPN -> PFN
  - als auch die inverse Operation PFN ->VPN

**Bei 48 Einträgen im L1 Cache bedeutet dies. Für diese Pages entfällt faktisch der Overhead beim Speicherzugriff. Für 512 weitere Seiten ist er minimal. Erst danach macht er sich bemerkbar.**



# TLB: Zugriff



**Berechnungen sind schnell, wenn sie sich auf zeitlich und räumlich nah beieinanderliegende Daten beziehen.**

- Speicheradressen liegen in der gleichen virtuellen Page (gleiche VPN).
- Der TLB-Eintrag wurde noch nicht verdrängt.



# TLB: Kontextwechsel

- VPNs sind prozessspezifisch => TLB löschen.
- Alternative:
  - TLB-Eintrag anreichern und individualisieren



**ASID: Address Space Identifier**



# TLB: Aufgabenverteilung

## TLB-Miss

- Hardware
  - Strukturen sind prozessorspezifisch
  - Page Tables haben eine definierte Struktur und liegen an definierten Stellen (Register CR3 bei X86 Prozessoren)
- Betriebssystem
  - löscht und organisiert den TLB
  - lediglich die Entry-Strukturen sind definiert



# Page Frames: Kleinere Page Tables

**Page Tables müssen kleiner werden!**

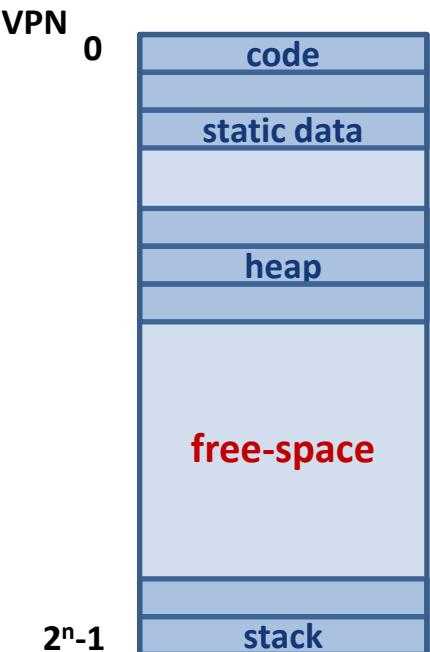
- 64-bit-Architektur und 128 GiB RAM
  - Arrays der Größe 20 PiB sind alles andere als eine gute Idee!

**Da ein Prozess „selten“ den gesamten, verfügbaren, virtuellen Adressraum belegt sind viele Einträge ungültig und damit verschwendeter Platz.**



# Kleinere Page Tables

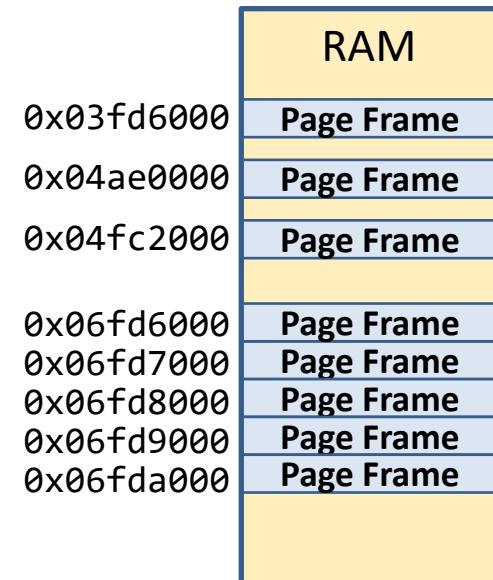
logischer Adressraum



MMU

Page Table Entry			
PFN	...	valid	rwx
0x06fd6	...	1	r-x
0x06fd7	...	1	r-x
0x06fd8	...	1	rw-
		0	
ungültig		0	
		0	
...	...	0	
0x04ae0	...	1	rw-
0x04fc2	...	1	rw-
0x03fd6	...	1	rw-
		0	
ungültig		0	
		0	
...	...	0	
0x06fda	...	1	rw-
0x06fd9	...	1	rw-

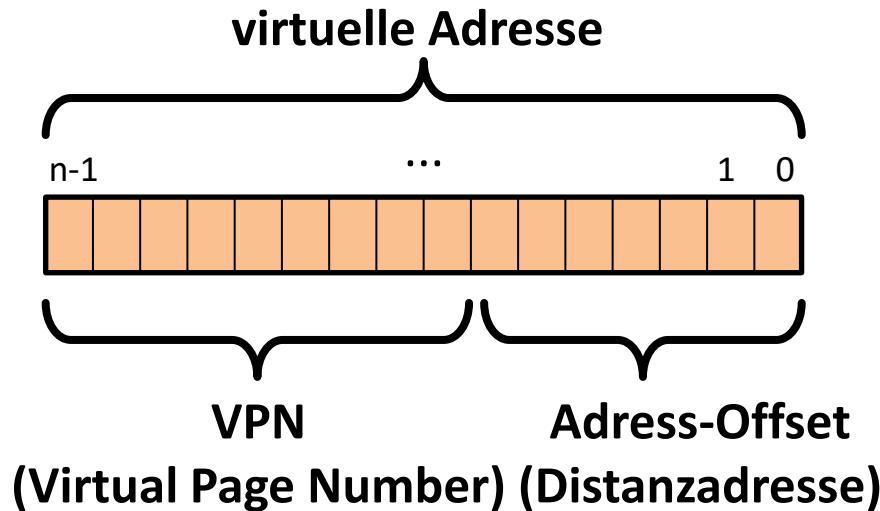
physischer Adressraum





# Kleinere Page Tables: Erste Idee

## Größere Page Frames



- mehr Bits für den Adress-Offset
- weniger Bits für die VPN



kleinere Tabellen aber wieder mehr interne Fragmentierung



# Kleinere Page Tables: Erste Idee

**Variable Adress-Offsets  
müssen von der Hardware unterstützt werden!**



**Je größer die Variabilität, desto komplexer die Hardware.**

**Moderne Hardware unterstützt typischerweise wenige unterschiedliche Frame-Größen. Der Vorteil liegt aber weniger in der Größenreduktion der Page-Tabelle als vielmehr in der Vermeidung von TLB-Zugriffsfehlern bei großen Datenstrukturen.**



# Kleinere Page Tables: Neue Ideen

**Komplexere Datenstrukturen!**



- weniger komplex: Hardware-Realisierung möglich
- sonst: Software-Lösung im Betriebssystem
  - TLB-Miss => Interrupt
  - BS findet PFN schnell und genial
  - TLB Update mit neuer Zuordnung (VPN -> PFN)



# Kleinere Page Tables: Zweite Idee

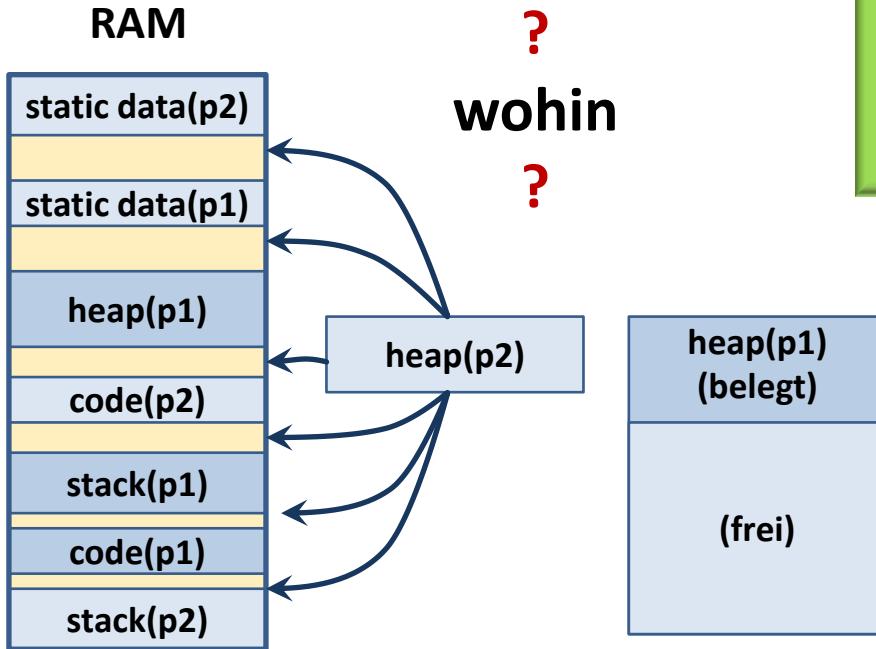
**Was war den so schlecht an der Segmentierung des Speichers!**

Nachteil: (siehe Folie 271)

**Noch immer befindet sich das ganze Segment als eine zusammenhängende (eventuell sehr große) Einheit im Hauptspeicher.**



# Rückblick: Segmentierung (Folie 272)



**Problem: Große Mengen an nicht nutzbarem aber freiem Speicher!**

## Fragmentierung

- extern: außerhalb eines „Segments“
- intern: innerhalb eines „Segments“



# Rückblick: Segmentierung

## Vorteil: (siehe Folie 271)

- deutlich geringerer Verbrauch an realem Speicher.
- die reale Speichergröße kann bei Bedarf angepasst werden.
- individueller Schutz der Segmente (rwx).
- Segmente können geteilt werden.
- Segmente können verschoben werden.

Hat man zwei gute Ansätze, dann führt deren Kombination vielleicht zu einem noch besseren!  
**Vorteile nutzen, Mängel kompensieren!**



# Kleinere Page Tables: Zweite Idee

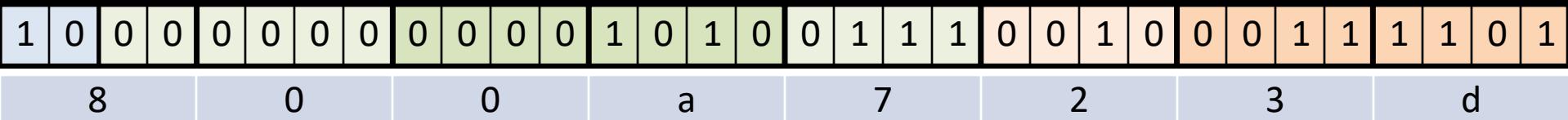
## Segmentierte Page Frames

- nicht eine Page Table pro Prozess,
- sondern eine Page Table pro Prozess und Segment

→ Page Table enthält nur valide Einträge,  
dafür sorgt Base-And-Bound.



# Beispiel einer Adressumrechnung



- virtuelle Adresse 0x800a723d

Segmentdeskriptor

- Segmentnummer 2

Segment	Base (Adresse PT)	Bounds (Adresse)	wächst positiv
2	0x00000fa0	0x00001260	1

- Virtual Page Number 0xa7

- Adress-Offset 0x23d

0	0x06fd6	...	1	rw-	0x0000fa0
	0x06fd7	...	1	rw-	
		1			
	gültig			1	
		1			
175	0x04ae0	...	1	rw-	0x0000125c

Adresse des ersten und  
letzten PTEs (4 Byte)



# Beispiel einer Adressumrechnung



- virtuelle Adresse 0x800a723d Segmentdeskriptor

Segment	Base (Adresse PT)	Bounds (Adresse)	wächst positiv
2	0x00000fa0	0x00001260	1

1.  $0x00000fa0$  (Base)

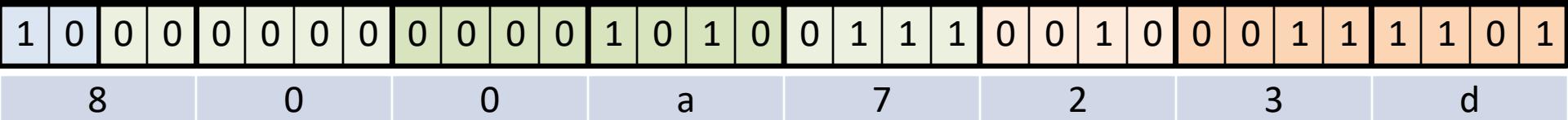
$$+ 0x29c \text{ (VPN } \times 4\text{)}$$

$$= 0x0000123c$$

2.  $0x0000123c < 0x00001260 \Rightarrow \text{PTE-Adresse im Range} \checkmark$



# Beispiel einer Adressumrechnung



- virtuelle Adresse 0x800a723d Segmentdeskriptor

• Segmentnummer	2	<b>Segment</b>	<b>Base (Adresse PT)</b>	<b>Bounds (Adresse)</b>	<b>wächst positiv</b>
• Virtual Page Number	0xa7	2	0x00000fa0	0x00001260	1

3. 0x0000123c

4. 0x05a6d000 (Adresse des Page Frames)

+ 0x23d (Offset)

= 0x05a6d23d (physische Adresse)

PTE

0x0000123c 0x05a6d ... 1 rw-



# Segmentierte Page Frames

## Vorteil:

- **Segmente:**
  - garantieren geringen Verbrauch an realem Speicher.
  - wachsen bei Bedarf => Page Table beinhaltet nur gültige Seiten.
- **Page Frames:**
  - garantieren einen nicht fragmentierten physischen Speicher.
  - unterstützen nicht zusammenhängende Segmente  
=> problemloses Wachstum.



# Segmentierte Page Frames

Nachteil:

Große Segmente haben weiterhin  
große, zusammenhängende  
Page Tables!



# Kleinere Page Tables: Dritte Idee

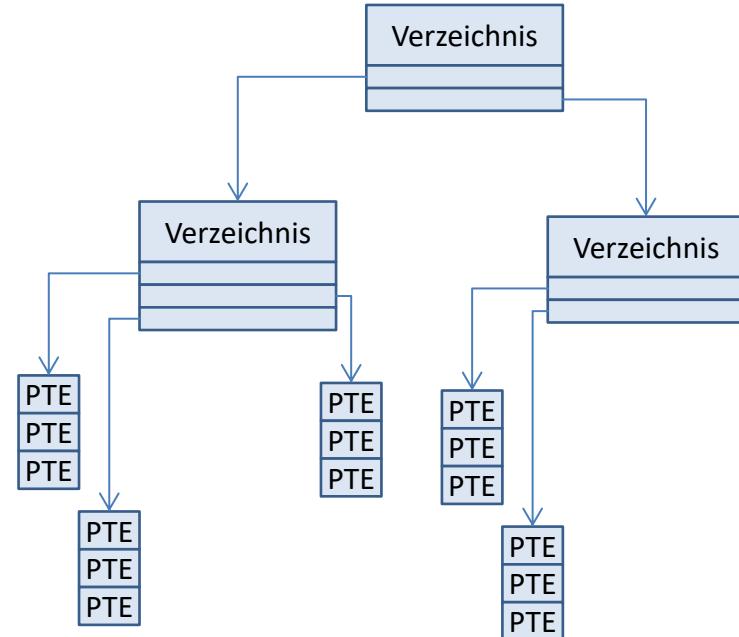
Hierarchien:  
Statt in einer Tabelle verwalten wir  
die PTEs in einem Baum!

## Vorteil:

- Der Speicherverbrauch sinkt logarithmisch

## Nachteil:

- Es wird komplexer!





# Multi-Level Page Tables

Ziel:

Kleine, nicht zusammenhängende Page Tables  
mit möglichst wenigen invaliden Einträgen.

**Frage:**

- Wie groß sollte ein zusammenhängendes Stück der Page Table maximal sein?

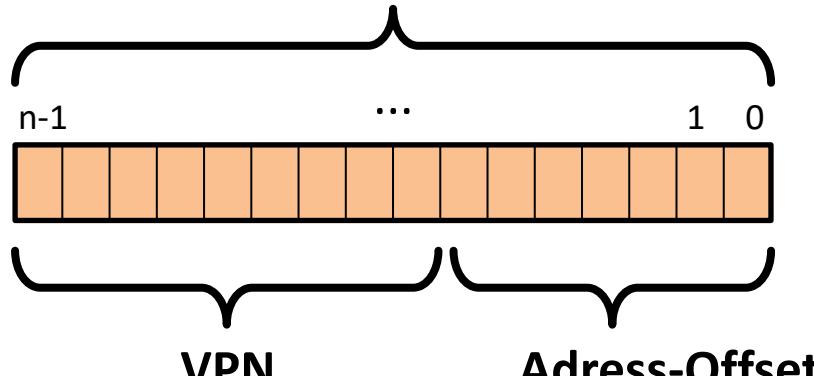
**Antwort:**

- Nicht größer als die Größe eines Page Frames!



# Multi-Level Page Tables

virtuelle Adresse (n Bit)



(Virtual Page Number) (Distanzadresse)

- virtuelle Adresse (n Bit)
- Adress-Offset (k Bit)
- Größe eines Page Frames von  $2^k$  Byte
- bietet Platz für  $2^k / 4$  4-Byte-Einträge
- bietet Platz für  $2^k / 8$  8-Byte-Einträge

[Ntdebugging Blog: Understanding !PTE , Part 1: Let's get physical, 2010  
[blogs.msdn.microsoft.com/ntdebugging/2010/02/05/understanding-pte-part-1-lets-get-physical/](http://blogs.msdn.microsoft.com/ntdebugging/2010/02/05/understanding-pte-part-1-lets-get-physical/)]



# Multi-Level Page Tables: Hierarchien

Der Bedarf an Hierarchieebenen (HE) entspricht der Größe einer VPN

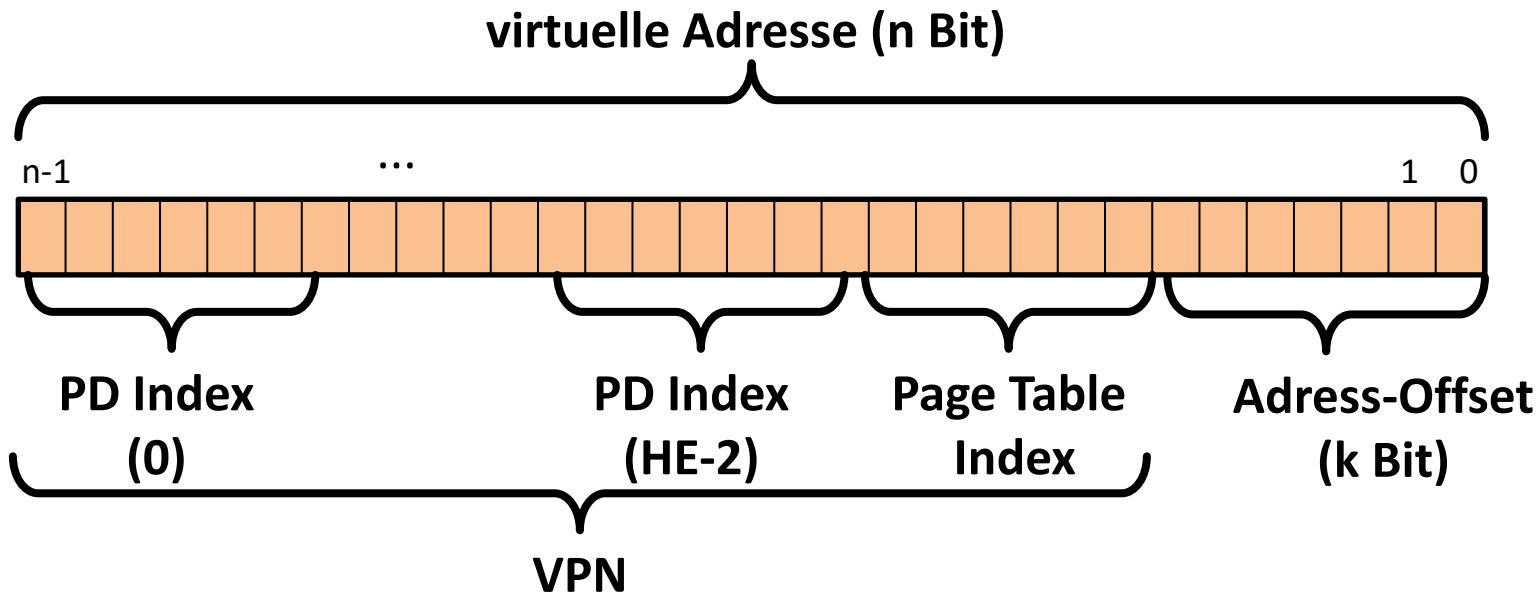
geteilt durch

$$HE = \left\lceil \frac{(n-k)}{\log_2 \left( \frac{2^k}{\left( \frac{\text{size}(PTE)}{8} \right)} \right)} \right\rceil = \left\lceil \frac{n-k}{k+3-\log_2(\text{size}(PTE))} \right\rceil$$

die Anzahl der Bits, die notwendig sind alle Einträge einer Seite zu identifizieren.



# Multi-Level Page Tables: Hierarchien





# Multi-Level Page Tables: Hierarchien

## x86-64

- PTE = 64 Bit
- n = 48 Bit
- k = 12 Bit

$$\left\lceil \frac{n-k}{\log_2\left(\frac{2^k \times 8}{size(PTE)}\right)} \right\rceil = \left\lceil \frac{n-k}{k+3-\log_2(64)} \right\rceil = \left\lceil \frac{36}{12+3-6} \right\rceil = \left\lceil \frac{36}{9} \right\rceil = 4$$

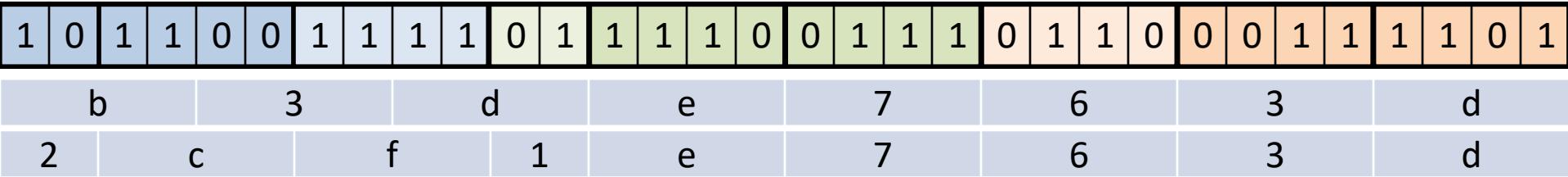
## x86-32

- PTE = 32 Bit
- n = 32 Bit
- k = 12 Bit

$$\left\lceil \frac{n-k}{\log_2\left(\frac{2^k \times 8}{size(PTE)}\right)} \right\rceil = \left\lceil \frac{n-k}{k+3-\log_2(32)} \right\rceil = \left\lceil \frac{20}{12+3-5} \right\rceil = \left\lceil \frac{20}{10} \right\rceil = 2$$



# Adressumrechnung



- virtuelle Adresse:  $0xb3de763d$
- 20 Bit VPN und 12 Bit Offset
- 10 Bits zum Identifizieren der Einträge
- 2 Ebenen (Page Directory + Page Table)
  - Adress-Offset:  $0x63d$
  - Page Table Index:  $0x1e7$
  - Page Directory Index:  $0x2cf$

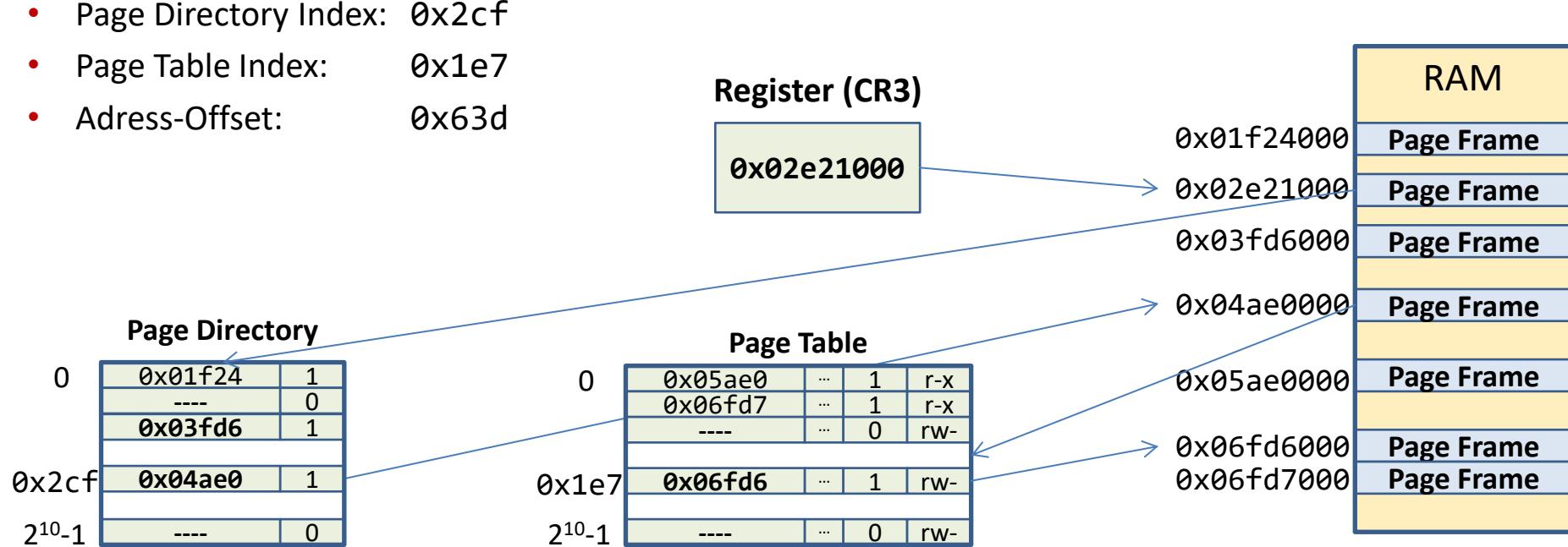


# Adressumrechnung

- virtuelle Adresse: 0xb3de763d
- Page Directory Index: 0x2cf
- Page Table Index: 0x1e7
- Adress-Offset: 0x63d

Transformation

physische Adresse: 0x06fd663d





# Page Frame: Zusammenfassung

- Der TLB ermöglicht die schnelle Transformation von virtuellen in physische Adressen.
- Bei einem TLB-Miss ist „Köpfchen“ gefragt! Page Tables gilt es zu organisieren:
  - Hardware-Lösungen:  
Eher simple Algorithmen mit festem Schema.
    - Multi-Level Page Tables / Segmentierte Page Frames
  - Software-Lösungen:  
Auch komplexe Algorithmen könnten von Vorteil sein.
    - Hash-Tabellen
    - Bäume
    - ...



# Swapping

Was tun, wenn die physischen Seiten ausgehen, wenn mehr Speicher gebraucht wird als RAM vorhanden ist?

Wie kann für jeden Prozess die Illusion eines eigenen, „beliebig großen“ Adressraums aufrechterhalten werden?



# Swapping

Lösung: mehr Speicher muss her!

- Auf der Festplatte ist meist reichlich Platz.
- Nicht gebrauchte Frames könnten ausgelagert werden!

Ein Teil der Festplatte wird hierfür vom Betriebssystem reserviert!

- Swap Space
- pagefile.sys (Windows)



# Swapping: Umsetzung

Wo befindet sich welche Seite?

- Der Ort (Memory/Disk) muss im PTE identifiziert werden!



Ein weiteres Bit im PTE  
valid/ rwx / present



Statt der physischen Adresse (PFN) wird  
der Block im Pagefile hinterlegt



# Swapping: Page Fault

- **TLB-Miss**
  - **Hardware-managed:** PTE wird lokalisiert
    - Present-Bit „true“ => TLB wird aktualisiert
    - Present-Bit „false“ => Interrupt
  - **Software-Managed (BS)** => Interrupt
- **Das Betriebssystem lagert (ggf.) nicht gebrauchte Seiten aus (Page out), das Betriebssystem lädt die angeforderte Seite von der Platte (Page in), das Betriebssystem aktualisiert die PTEs und kehrt vom Interrupt zurück.**
- **TLB-Miss kann jetzt aufgelöst werden (Present-Bit „true“)**



# Swapping: Strategie

**Ziel:**

**Page Faults vermeiden, denn Page Faults kosten  
richtig Zeit (mehrere ms)!**



## Zwei zentrale Fragen:

- Welche Seiten sollen (wann) geladen werden? (Page in)
- Welche Seiten sollen (wann) verdrängt werden? (Page out)



# Swapping: AMAT

## Average Memory Access Time (Durchschnittliche Zugriffszeit)

$$AMAT = (P_{HIT} \cdot T_{MEM}) + (P_{MISS} \cdot T_{DISK})$$

- $T_{MEM} \approx 100 \text{ ns}$  (im Cache nochmals deutlich schneller)
- $T_{DISK} \approx 10 \text{ ms}$
- $P_{HIT} = 90\% \quad \Rightarrow AMAT \approx (0,9 \quad \times 100\text{ns}) + (0,1 \quad \times 10\text{ms}) \approx 1.000.100 \text{ ns}$
- $P_{HIT} = 99\% \quad \Rightarrow AMAT \approx (0,99 \times 100\text{ns}) + (0,01 \times 10\text{ms}) \approx 100.100 \text{ ns}$
- $P_{HIT} = 99,9\% \quad \Rightarrow AMAT \approx (0,999 \times 100\text{ns}) + (0,001 \times 10\text{ms}) \approx 10.100 \text{ ns}$



# Swapping: Page in

## Wann sollte eine Seite geladen werden?

- **On Demand:** Man lädt die Seite wenn man sie braucht!
  - Simpel und naheliegend.
  - Jeder Zugriff auf eine neue Seite führt zu einem Fehler.
- **Im Voraus (Prefetching):** Man lädt die Seite bevor man sie braucht!
  - Das Betriebssystem fungiert als Orakel für die Zukunft.
  - Die richtige Strategie ist gefragt.
- **Mit Anwendungsunterstützung:** Die Applikation gibt Tipps!
  - `man 2 madvise()`



# Swapping: Page out

**Wann und welche Seite sollte verdrängt werden?**

- **Wann?**

**Trivial, wenn für die neue Seite kein Platz vorhanden ist.**

- **Welche?**

**Viel schwieriger, die richtige Strategie ist gefragt.**



# Swapping: Die optimale Strategie (MIN)

**Man verdrängt die Seite, die man nie wieder oder ganz zuletzt benötigen wird. (am weitesten in der Zukunft)**

[van Roy, B. A Short Proof of Optimality for the MIN Cache Replacement Algorithm. 2010]



**Probleme wiederholen sich:**

**Vorhersagen mussten schon beim Scheduling getroffen werden! Wieder ist eine Approximation gefragt!**



# Swapping-Strategien

**Generell sollte man räumliche und zeitliche Nähe betrachten.  
(Arrays und Schleifen)**

- **FIFO: First In First Out (simpel)**
  - Was man vor langer Zeit gebraucht hat, ist nicht mehr aktuell!
- **LRU: Least Recently Used (nicht so einfach)**
  - Was man schon lange nicht mehr gebraucht hat, kann weg!
- **LFU: Least Frequently Used (komplex)**
  - Was man am wenigsten braucht, kann weg!



# Swapping-Strategien (Beispiel)

	ABC	A	C	D	D	A	E	E	C	A	A	A	→
MIN	A B C	A B C	A B C	A B C	A D C	A D C	A D C	A E C	A E C	A E C	A E C	A E C	2 Fehler
FIFO	A B C	A B C	A B C	A B C	D B C	D B C	D A C	D A E	D A E	C A E	C A E	C A E	4 Fehler
LRU	A B C	A B C	A B C	A B C	A D C	A D C	A D C	A D E	A D E	A C E	A C E	A C E	3 Fehler
LFU	A B C	A B C	A B C	A B C	A D C	A D C	A D C	A D E	A D E	A C E	A C E	A C E	3 Fehler



# Swapping-Strategien: Implementierung

**Problem: Es gibt verdammt viele Page Frames.  
Diese müssen verwaltet werden! (pro 4 GiB RAM eine Million)**

- **FIFO: klar Queue**
  - Achtung: Prozesse geben Speicher auch wieder frei. Die Queue verliert permanent Elemente.
- **LRU: PTEs benötigen Zeitstempel**
  - Achtung: Diese müssen beim Auslagern einer Seite alle überprüft werden oder die Listen werden bei jedem Zugriff aktualisiert!

**untauglich!**



# Swapping-Strategien: Implementierung

Oftmals ist eine Approximation ausreichend!  
Oder warum sollte man eine Strategie, die lediglich eine  
Approximation darstellt, perfekt implementieren!

- **Approximation von LRU:**

- Nicht der letzte Eintrag wird ermittelt, sondern ein „nicht neuer“.



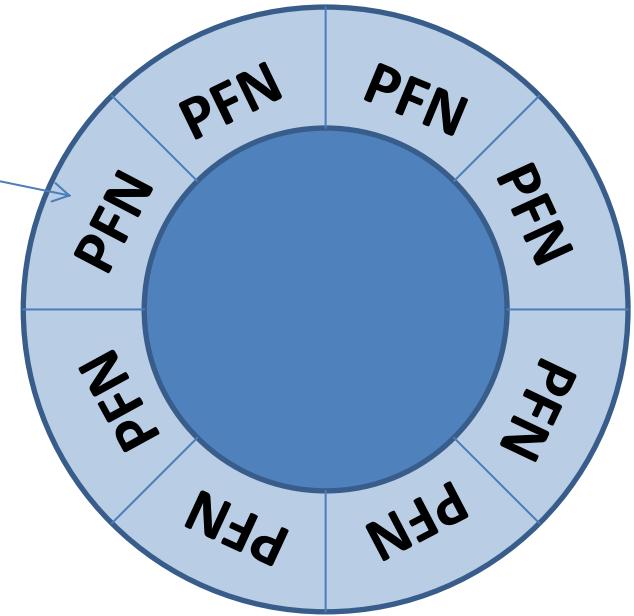
ein weiteres Bit im PTE  
valid/ rwx / present / **used**

- Dieses wird immer dann gesetzt, wenn auf die Seite zugegriffen wird.



# LRU durch Clocking

```
typedef struct clock_t{  
    pfn_t      current;  
} clock_t;  
  
static clock_t clock;  
  
pfn_t LRU(){  
    while (in_use(clock.current)){  
        reset_used(clock.current)  
        clock.current++;  
        clock.current %= PAGE_COUNT;  
    }  
    return clock.current;  
}
```





# LRU durch Clocking

Sieht simpel aus, ist es aber nicht.

- **Das Used-Bit wird im PTE verwaltet nicht in der PFN.**

Man muss von der PFN zum PT (prozessspezifisch) und weiter zum PTE kommen!

→ (Inverted Page Table) Zu welchem Prozess gehört dieser Frame und unter welcher VPN wird er geführt. (**vgl. Folie 285**)

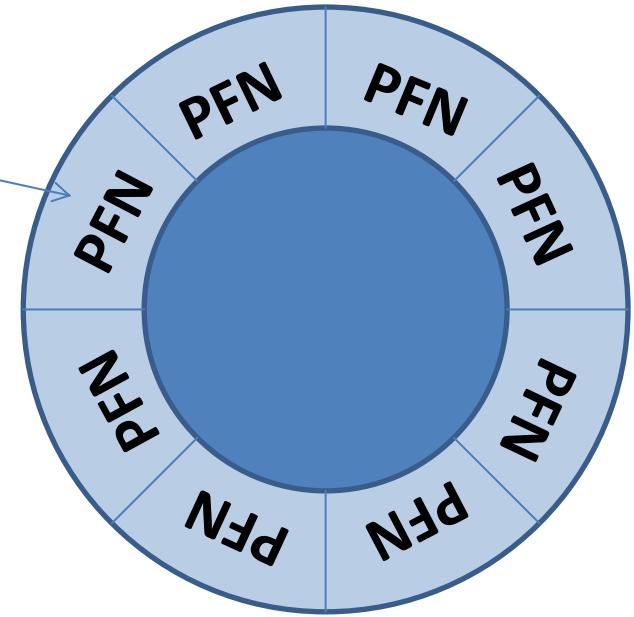
- **Bedarf existiert nur, wenn keine verfügbaren Frames vorhanden sind.**

→ Eine Free-List aller verfügbaren Frames wäre nicht schlecht.



# LRU durch Clocking

```
typedef struct clock_t{  
    inv_pte_t inv_pt[PAGE_COUNT];  
    pfn_t     current;  
} clock_t;  
  
static clock_t clock;  
  
pfn_t LRU(){  
    while (in_use(clock.current)){  
        reset_used(clock.current)  
        clock.current++;  
        clock.current %= PAGE_COUNT;  
    }  
    return clock.current;  
}
```





# LRU durch Clocking

Sieht simpel aus, ist es aber nicht.

- Sollten wirklich alle Seiten gleich behandelt werden?
  - Nicht jede Seite erzeugt den gleichen Aufwand. Haben keine Änderungen stattgefunden, dann muss man sie nicht sichern.



**Dirty-Bit im PTE**

- Was ist mit Prozessen die warten?
- ...



**Der Grundstein ist gelegt! Die Illusion ist komplett!**

**Jeder Prozess hat seinen eigenen Prozessor und verfügt  
über das ganze adressierbare RAM!**

**Und vor allem, das Betriebssystem hat alles im Griff!**



# Teil III

# Concurrency:

# Prozesse und Threads



# Concurrency: Prozesse und Threads

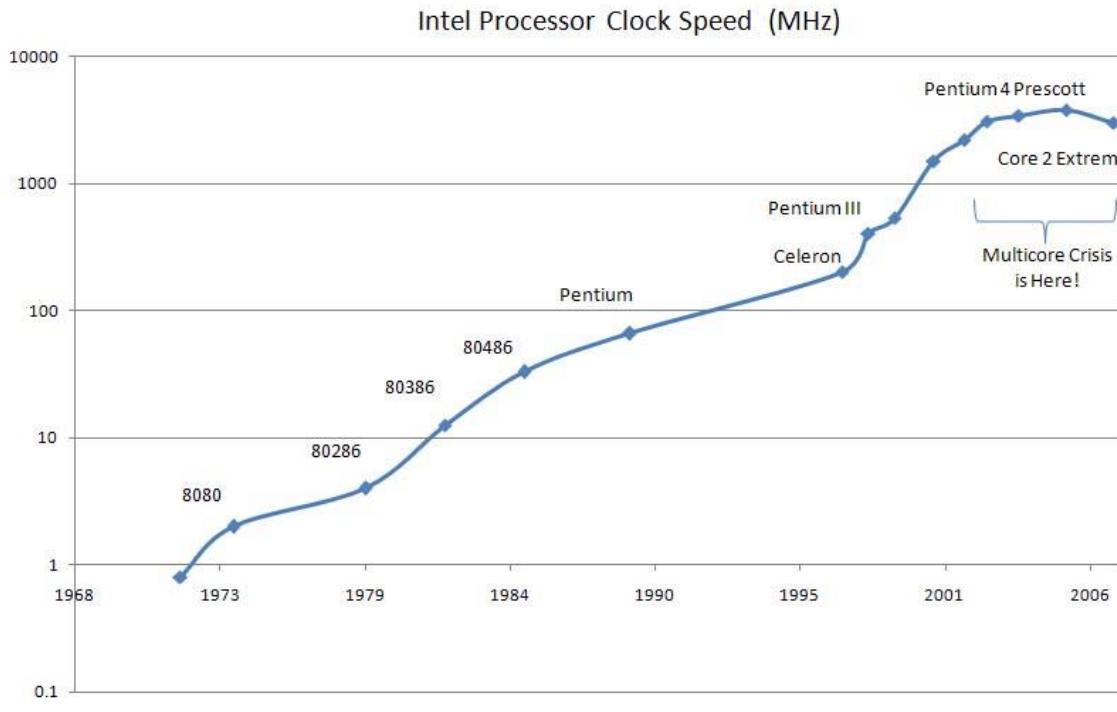
- Threads
- Mutexes, Locks und Condition Variables
- Semaphore
- Monitore
- Deadlocks



- **Patterns**
  - Producer / Consumer
  - Reader / Writer



# Die Multicore-Krise



Seit etwa 2005 hat sich die Prozessorgeschwindigkeit nicht mehr signifikant erhöht!



Seither erhöht sich die Zahl der Kerne.

[Warfield, B.: A Picture of the Multicore Crisis. Posted in SmoothSpan Blog owned by Bob Warfield on September 6, 2007]

<https://smoothspan.wordpress.com/2007/09/06/a-picture-of-the-multicore-crisis/>



# Die Multicore-Krise

Anwendungen müssen alle Kerne gleichmäßig nutzen.

## Idee:

- Anwendungen bestehen aus mehreren Prozessen
- Prozesse kommunizieren nur bei Bedarf

## Vorteil:

- Teile sind isoliert und abgesichert

## Nachteil:

- Kontextswitch und Prozesserzeugung ist aufwendig für das System
- Interprozesskommunikation ist aufwendig für den Entwickler und das System



Skaliert für Anwendungen im Kleinen gut und im Großen schlecht!

Bsp.: Chromium

Ein Prozess pro Instanz  
(Tab oder Seite)

[Reis, Ch.; Gibble,S. D.: Isolating Web Programs in Modern Browser Architectures. In Proceedings of EuroSys'09. Nürnberg: ACM, April 2009]



# Die Multicore-Krise

## Nebenläufigkeit innerhalb eines Prozesses

Threads (Fäden) sind leichtgewichtige Prozesse, die sich den gleichen Adressraum teilen.

- Jeder Thread besitzt seinen eigenen Stack
- Jeder Thread hat einen eigenen Instruction Pointer
- Jeder Thread hat eine eigenen ThreadID.



- Es gibt weiterhin ein Kontextswitch.  
(Register muss man sichern aber **nicht die Page Table**)
- Man braucht für jeden Thread einen Control Block (TCP) als Teil des PCB.



# Die Multicore-Krise

## Nebenläufigkeit innerhalb eines Prozesses

**Threads (Fäden) sind leichtgewichtige Prozesse, die sich den gleichen Adressraum teilen.**

- Alle Threads eines Prozesses nutzen gemeinsam:

- den gleichen Adressraum
- die gleiche ProcessID
- die gleichen File-Deskriptoren
- ...



**Environment**

**Skaliert auch im Großen!**



# Die Multicore-Krise

**Anwendungen müssen alle Kerne gleichmäßig nutzen.**  
**Ziel n-Kerne sind n-mal so schnell wie ein Kern!**

- Neue Aufgaben für den Entwickler: **(Parallele Programmierung)**
  - Teilaufgaben müssen verteilt werden und möglichst autonom bearbeitet.  
(Objekte als selbständig agierende Einheiten)
  - verlässliche „Kommunikation“ innerhalb des Prozesses

**Das Betriebssystem muss hierfür die Mittel bereitstellen.**



# Parallel Programmierung

- **Producer Consumer**
  - Objekte haben unterschiedliche Aufgaben. Es gibt Objekte, die etwas „erzeugen“, das andere „verbrauchen“ (**man wartet aufeinander**)
- **Monitore**
  - Spezielle Objekte übernehmen die Absicherung kritischer Funktionalität.
- **Worker Queues**
  - Aufgaben werden als Objekte verstanden, die von autonomen Einheiten (Worker) abgeholt und bearbeitet werden.
- **Pipelines (Fließband)**
  - Aufgaben zerfallen in Teile, die unabhängig und nacheinander bearbeitet werden.



# Prozesse benötigen Rechenzeit

## Threads



# Threads und das Betriebssystem

## User-Level Threads:

- User-Level Threads werden von der Anwendung verwaltet (Bibliothek, virtuelle Maschine).
- Das Betriebssystem sieht nur den Prozess aber nicht dessen Threads.

### Vorteil:

- **keine zusätzliche Betriebssystemunterstützung => weniger Overhead**
- **eigene Scheduling-Strategien**

### Nachteil:

- **Das Betriebssystem sieht nur einen Prozess => nur ein Kern wird genutzt.**
- **Blockiert ein Thread (I/O-Zugriffe) blockieren alle.**

**Nur geeignet für Single-Core-Prozessoren etwa für  
eingebettete Systeme!**



# Threads und das Betriebssystem

## Kernel-Level Threads:

- Werden vom Betriebssystem geplant und verwaltet.
- Erzeugung, Scheduling, Synchronisierung

**Linux statt fork, clone**

[man 2 clone]

## Vorteil

- Kernel-Level Threads laufen parallel und unabhängig (blockierende Threads behindern sich nicht).
- Unterschiedliche Threads können unterschiedlichen Kernen zugewiesen werden.



**leichtgewichtige Prozesse**

## Nachteile

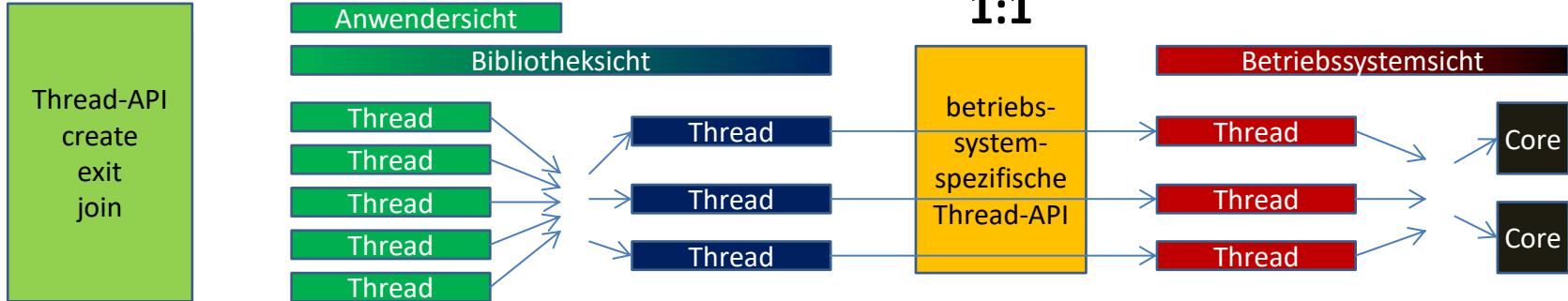
- Durch das Betriebssystem entsteht Overhead, und man muss in der Lage sein, mit Massen umzugehen!



# Threads und das Betriebssystem

## Vom User-Level Thread zum Kernel-Level Thread

Aufgabe einer Thread-Bibliothek oder eines Interpreters ist es, User-Level Threads auf Kernel-Level Thread abzubilden.





# Threads nutzen denselben Speicher

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

static volatile int magic = 0;

void *the_thread(void *args){
    for (int i = 0; i < 100000; i++)
        magic++;
    return NULL;
}
```

```
int main(void)
{
    pthread_t th1, th2;
    pthread_create(&th1, NULL, the_thread, NULL);
    pthread_create(&th2, NULL, the_thread, NULL);

    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    printf("A surprise %d\n", magic);
    return EXIT_SUCCESS ;
}
```

```
:> gcc -Wall thread.c -lpthread -o thread
:> ./thread
A surprise 200000
:> ./thread
A surprise 151785
:>
```

?



# Threads nutzen denselben Speicher

```
:> gcc -Wall thread.c -lpthread -o thread  
:> objdump -d thread
```

0000000004006a6 <the\_thread>:

```
4006a6: 55                      push   %rbp
4006a7: 48 89 e5                mov    %rsp,%rbp
4006aa: 48 89 7d e8              mov    %rdi,-0x18(%rbp)
4006ae: c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
4006b5: eb 13                  jmp    4006ca <the_thread+0x24>
4006b7: 8b 05 8b 09 20 00      mov    0x20098b(%rip),%eax
4006bd: 83 c0 01                add    $0x1,%eax
4006c0: 89 05 82 09 20 00      mov    %eax,0x200982(%rip)
4006c6: 83 45 fc 01              addl   $0x1,-0x4(%rbp)
4006ca: 81 7d fc 9f 86 01 00    cmpl   $0x1869f,-0x4(%rbp)
4006d1: 7e e4                  jle    4006b7 <the_thread+0x11>
4006d3: b8 00 00 00 00          mov    $0x0,%eax
4006d8: 5d                      pop    %rbp
4006d9: c3                      retq
```

magic++;

# 601048 <\_\_TMC\_END\_\_>  
# 601048 <\_\_TMC\_END\_\_>



# Threads nutzen denselben Speicher

Thread 1

- `mov ($601048),%eax`
- `add $0x1,%eax`
  - timer interrupt
- `mov %eax,($601048)`
  - weiter

```
mov    ($601048),%eax  
add    $0x1,%eax  
mov    %eax,($601048)
```

richtig wäre `magic + 2`

0x601048 magic + 1

EAX magic +1

gesichert  
EAX  
magic +1

Thread 2

- `mov ($601048),%eax`
- `add $0x1,%eax`
- `mov %eax,($601048)`
  - timer interrupt



# Threads „fundamentale“ Begriffe

- **Critical Section (kritischer Abschnitt):**

Ein **kritischer Abschnitt** ist ein Stück Code, in dem eine gemeinsame Ressource (Speicher, Datei, ...) genutzt wird!

- **Race Condition (Wettlaufsituation):**

Eine **Race Condition** tritt ein, wenn sich mehrere Threads innerhalb einer Critical Section befinden und versuchen die gemeinsame Ressource zu verändern.

- **Indeterminism (Indeterminismus):**

Ein Programm verhält sich **indeterministisch**, wenn sich Threads während einer Race Condition von „Mal zu Mal“ unterschiedlich verhalten.



# Threads



**Lösung:** Synchronisation und wechselseitiger Ausschluss aus dem „kritischen Abschnitt“!

```
mov    ($601048),%eax  
add    $0x1,%eax  
mov    %eax,($601048)
```



unteilbare Einheiten



**Unterstützung durch das Betriebssystem!**



# Synchronisation von Threads

**Mutexes, Locks  
und  
Condition Variables**



# Erster Baustein: Locks (Mutex)

Ein Mechanismus der den „wechselseitigen Ausschluss“ (mutual exclusion) aus dem kritischen Abschnitt garantiert!

## Mutex anfordern

```
mov    ($601048),%eax  
add    $0x1,%eax  
mov    %eax,($601048)
```

} unteilbare Einheiten

## Mutex freigeben

Nur der (Thread), der das Mutex besitzt, kann den kritischen Abschnitt durchlaufen. Alle anderen müssen warten.



# Erster Baustein: Locks (Mutex)

- Lock (Mutex) erzeugen
  - `pthread_mutex_init()`
  - `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`
- kritischen Abschnitt betreten (für andere sperren)
  - `pthread_mutex_lock(& mutex);`
  - `if (pthread_mutex_trylock(& mutex) == 0)`
- kritischen Abschnitt verlassen (für andere wieder freigeben)
  - `pthread_mutex_unlock(& mutex);`
- Datenstruktur wieder freigeben
  - `pthread_mutex_destroy(& mutex )`

Linux

```
:> man 3 pthread
```



# Erster Baustein: Locks (Mutex)

- Lock (Mutex) erzeugen
  - **Mutex mutex = new Mutex();**
- kritischen Abschnitt betreten (für andere sperren)
  - **bool allowed = mutex.WaitOne();**
  - **bool allowed = mutex.WaitOne(1000); // eine Sekunde warten**
- kritischen Abschnitt verlassen (für andere wieder freigeben)
  - **mutex.ReleaseMutex();**
- Datenstruktur wieder freigeben
  - **mutex.Dispose();**

C#

msdn



# Erster Baustein: Locks (Mutex)

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

static volatile int magic = 0;
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *the_thread(void *args){
    for (int i = 0; i < 100000; i++){
        pthread_mutex_lock(&mutex);
        magic++;
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```

```
int main(void)
{
    pthread_t th1, th2;
    pthread_create(&th1,NULL,the_thread,NULL);
    pthread_create(&th2,NULL,the_thread,NULL);

    pthread_join(th1,NULL);
    pthread_join(th2,NULL);
    printf("A surprise %d\n",magic);
    return EXIT_SUCCESS ;
}
```

```
:> gcc -Wall thread.c -lpthread -o thread
:> ./thread
A surprise 200000
:> ./thread
A surprise 200000
:>
```

!tut!



# Erster Baustein: Locks (Mutex)

**Wie realisiert man (effizient)  
Locks?**

- Welche Aufgabe übernimmt die Hardware?
- Welche Aufgabe übernimmt das Betriebssystem?



# Locks (Mutex): Ziele

- **Wechselseitiger Ausschluss!**

**Weitere Ziele?**

- **Korrektheit**

- **Deadlock-Freiheit**

Ist ein Lock verfügbar, dann kann es auch in Besitz genommen werden.

- **Starvation-Freiheit**

Wer auf ein Lock wartet, das freigegeben wird, der hat auch die Chance es zu bekommen.



# Locks (Mutex): Ziele

- **Fairness:**  
Jeder, der wartet, hat die gleiche Chance „an die Reihe zu kommen“.
- **Performance:**  
Der Einsatz von Locks sollte den Ablauf nicht (merklich) verzögern.



# Locks (Mutex): Realisierung erste Idee

Interrupts ausschalten!



<code>lock();</code>	<code>mov (\$601048),%eax</code>
<code>magic++;</code>	<code>add \$0x1,%eax</code>
<code>unlock();</code>	<code>mov %eax,(\$601048)</code>

Die drei Anweisungen bilden trivialerweise eine Einheit,  
da kein Interrupt auftritt, der sie unterbrechen kann.

Diese Idee wurde aber bereits früher verworfen!



# Der kooperative Ansatz:

**Das Betriebssystem vertraut den Anwendungen!**

- Systemaufrufe
- Für Langläufer

to yield

```
int main(void)
{
    lock();
    ...
    unlock();
    return EXIT_SUCCESS;
}
```

0, ...).  
eingeführt.

lassen

[pons: de.pons.com]



# Locks (Mutex): Realisierung zweite Idee

## Ausnutzen atomarer Hardware-Operationen!

- **Test-And-Set (atomic exchange)**  
realisiert einen  
atomaren Wechsel  
(x86 xchg)
- **Compare-And-Set**  
realisiert einen  
bedingten atomaren Wechsel  
(x86 cmpxchg)

```
int t_a_s(int* ptr, int new_val) {  
    int old_val = *ptr;  
    *ptr = new_val;  
    return old_val;  
}
```

Pseudocode

```
int c_a_s(int* ptr, int exp_val int new_val) {  
    int actual_val = *ptr;  
    if (actual_val == exp_val)  
        *ptr = new_val;  
    return actual_val;  
}
```

Pseudocode



# Locks (Mutex): Test-And-Set

```
typedef struct mutex_t {  
    int lock;  
} mutex_t;  
  
void init_mutex(mutex_t *mtx) {  
    mtx->lock = 0; // mutex is free  
}  
  
void mutex_lock(mutex_t *mtx) {  
    while(t_a_s(&(mtx->lock), 1)==1)  
        ; // try again  
}  
  
void mutex_unlock(mutex_t *mtx) {  
    mtx->lock = 0;  
}
```

Setzt den Wert des Wächters in einer atomaren Operation auf 1 und gibt den alten Wert zurück.

Zwei mögliche Resultate:

- 1. der alte Wert war 1**, d.h., das Mutex ist bereits gelockt => der Test „== 1“ evaluiert zu „true“ und wir versuchen es erneut.
- 2. der alte Wert war 0**, d.h., das Mutex ist nicht gelockt => der Test „==1“ evaluiert zu „false“ die Schleife bricht ab und wir haben den kritischen Abschnitt betreten.



# Locks (Mutex): Compare-And-Set

```
typedef struct mutex_t {  
    int lock;  
} mutex_t;  
  
void init_mutex(mutex_t *mtx) {  
    mtx->lock = 0; // mutex is free  
}  
  
void mutex_lock(mutex_t *mtx) {  
    while(c_a_s(&(mtx->lock), 0, 1)==1)  
        ; // try again  
}  
  
void mutex_unlock(mutex_t *mtx) {  
    mtx->lock = 0;  
}
```

Prüft den Wert des Locks in einer atomaren Operation auf 0, um ihn dann ggf. zu ändern.

Zwei mögliche Resultate:

- 1. der alte Wert war 1**, (das Mutex ist bereits gelockt)  
=> der Vergleich auf 0 schlägt fehl und es wird 1 zurückgegeben. Der Test „== 1“ evaluiert zu „true“ und wir versuchen es erneut.
- 2. der alte Wert war 0**, (das Mutex ist nicht gelockt)  
=> der Vergleich auf 0 gelingt, der Wert wird auf 1 gesetzt und der alte Wert (0) wird zurückgegeben. Der Test „==1“ evaluiert zu „false“ die Schleife bricht ab und wir haben den kritischen Abschnitt betreten.



# Locks (Mutex): Probleme

```
while(t_a_s(&(mtx->lock), 1) == 1)  
    ; // try again
```

```
while(c_a_s(&(mtx->lock), 0, 1) == 1)  
    ; // try again
```

„spinning Lock“ „aktives Warten“

- **Der wartende Thread verbraucht unnötig CPU-Zeit!**



Nutzt der Scheduler Round Robin! wird ein ganzer Zeitschlitz mit „Nichtstun“ verschwendet!

- **Der wartende Thread kommt vielleicht nie an die Reihe!**



Wer garantiert, dass das Lock nicht immer zur falschen Zeit belegt ist!



# Locks (Mutex): Probleme

```
while(t_a_s(&(mtx->lock), 1) == 1)  
    pthread_yield(); // try again later
```

```
while(c_a_s(&(mtx->lock), 0, 1) == 1)  
    pthread_yield(); // try again later
```

yield(), den anderen den Vortritt geben

- Der wartende Thread verbraucht keine Zeit, er gibt auf
- Dafür viele Kontextwechsel bei vielen wartenden Threads, bevor der Thread im kritischen Abschnitt wieder an die Reihe kommt.



etwas besser, aber immer noch unfair!



# Locks (Mutex): Ticket Locks (sind fair)

## Ausnutzen atomarer Hardware-Operationen!

- Fetch-And-Add (atomic increment) realisiert die atomare Addition des Wertes in einer Speicherzelle (x86 xadd)

```
int f_a_a(int* ptr) {  
    int old_val = *ptr;  
    ++*ptr;  
    return old_val;  
}
```

Pseudocode

Jeder Thread wartet auf eine besondere Ausprägung des Locks, sein „Ticket“. Erst dann ist er an der Reihe.



# Locks (Mutex): Ticket Locks (sind fair)

```
typedef struct mutex_t {
    int lock, ticket;
} mutex_t;

void init_mutex(mutex_t *mtx) { // no ticket
    mtx->lock = mtx->ticket = 0; // in use
}

void mutex_lock(mutex_t *mtx) {
    int ticket = f_a_a(&(mtx->ticket));
    while(mtx->lock != ticket)
        pthread_yield(); // try again later;
}

void mutex_unlock(mutex_t *mtx) {
    mtx->lock++; // next please!
}
```



# Locks (Mutex): Mehr Performance

yield() ist gut, aber nicht gut genug!

- **Viele wartende Threads, d.h., viele Kontextwechsel bevor der Thread im kritischen Abschnitt wieder an die Reihe kommt.**



Offensichtlich braucht man etwas Management!



# Locks: Queues, Park und Unpark

```
typedef struct mutex_t {  
    int lock, guard; // a lock for the lock  
    queue_t *threads2wait; // a queue for  
} mutex_t; // waiting threads  
  
void init_mutex(mutex_t *mtx) {  
    mtx->lock = mtx->guard = 0;  
    queue_init(mtx->threads2wait);  
}  
void mutex_unlock(mutex_t *mtx) {  
    while(t_a_s(&(mtx->guard), 1) == 1);  
    if (mt_queue(mtx->threads2wait))  
        mtx->lock = 0;  
    else  
        unpark(dq_queue(mtx->threads2wait))  
    mtx->guard = 0;  
}
```

```
void mutex_lock(mutex_t *mtx) {  
    while(t_a_s(&(mtx->guard), 1) == 1);  
    if (mtx->lock == 0){// we are the first  
        mtx->lock = 1;  
        mtx->guard = 0;  
    } else {  
        q_enqueue(mtx->threads2wait, gettid());  
        setpark(); // we make an announcement  
        mtx->guard = 0;  
        park();  
    }  
}
```

eine  
Race Condition



# Locks (Mutex): Noch mehr Performance

Bei einem Multicore-Prozessor muss der aktiv wartende Thread (spinning) eventuell nicht den ganzen Zeitschlitz warten, wenn der aktive Thread auf einem anderen Kern läuft.

- **Spinnig ist gut für kurze Wartezeiten**
- **yield() ist gut für lange Wartezeiten**



Ein bisschen Spinning und dann yield();



# Locks (Mutex): Noch mehr Performance



Ein bisschen Spinning und dann yield();

```
void mutex_lock(mutex_t *mtx) {
    int i = 10000;
    while(t_a_s(&(mtx->guard), 1) == 1)
        if (--i == 0){
            i = 10000;
            yield(); // give up and try again later
        }; // try again;
    ...
}
```



# Locks (Mutex): Zusammenfassung

## Locks ermöglichen den wechselseitigen Ausschluss

Für die Realisierung sind Konzepte auf Anwendungsebene, Betriebssystemebene und Hardware-Ebene gefragt.

- **Anwendungsebene:**  
Threads und Locks müssen verwaltet werden (Queues,...).
- **Betriebssystemebene:**  
Funktion zum Anhalten und Starten von Threads müssen bereitgestellt werden.
- **Hardware-Ebene:**  
Komplexe unteilbare Operationen garantieren eindeutige Zustände.



# Beispiel: Dekorierer der synchronisiert

```
...
TableModel model = new SyncTableModel(data, columnNames);
final JTable table = new JTable(model);
...

public class SyncTableModel extends DefaultTableModel {

    private static final long serialVersionUID = 1L;
    private Object[] locks;

    public SyncTableModel(Object[][][] data, String[] columnNames) {
        super(data, columnNames);
        this.locks = new Object[this.getRowCount()];
        for (int i = 0; i < this.locks.length; i++)
            this.locks[i] = new Object();
    }
}
```



# Beispiel: Dekorierer der synchronisiert

```
public Object getValueAt(int rowIndex, int columnIndex) {  
    synchronized (this.locks[rowIndex]) {  
        return super.getValueAt(rowIndex, columnIndex);  
    }  
}  
  
public void setValueAt(Object aValue, int rowIndex, int columnIndex) {  
    synchronized (this.locks[rowIndex]) {  
        super.setValueAt(aValue, rowIndex, columnIndex);  
    }  
}  
}  
  
...  
TableModel model = new SyncTableModel(data, columnNames);  
final JTable table = new JTable(model);  
table.setPreferredScrollableViewportSize(new Dimension(500, 70));  
table.setFillsViewportHeight(true);  
...
```



# Threads agieren unkoordiniert

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

static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *the_thread(void *args){
    for (int i = 0; i < 100; i++){
        pthread_mutex_lock(&mutex);
        printf("I'm %s\n", (char*) args);
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```

```
:> gcc -Wall thread.c -lpthread -o thread
:> ./thread
I'm A
I'm A
I'm A
I'm A
I'm A
I'm A
```

```
int main(void)
{
    pthread_t th1, th2;
    pthread_create(&th1, NULL, the_thread, "A");
    pthread_create(&th2, NULL, the_thread, "B");

    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    return EXIT_SUCCESS ;
}
```

**Manchmal wäre etwas Koordination besser!**



# Threads



Lösung klar! Bedingung definieren, prüfen und warten, falls sie nicht erfüllt ist.

```
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;  
  
static int volatile state;  
  
typedef struct info_t{  
    char* name;  
    int target; // target state  
    int next;   // next state  
} info_t;
```



# Threads

```
void *the_thread(void *args){  
    info_t* info = (info_t*) args;  
    for (int i = 0; i < 100; i++){  
        while (1){  
            pthread_mutex_lock(&mutex);  
            if (state == info->target)  
                break; // my turn  
            pthread_mutex_unlock(&mutex);  
        }  
        printf("I'm %s\n", info->name);  
        state = info->next; // change  
        pthread_mutex_unlock(&mutex);  
    }  
    return NULL;  
}
```

```
int main(void)  
{  
    pthread_t th1, th2;  
    info_t info1, info2;  
    info1.name = "A";  
    info2.name = "B";  
    info1.target = info2.next = 0;  
    info2.target = info1.next = 1;  
    state= 0;  
  
    pthread_create(&th1,NULL, the_thread, &info1);  
    pthread_create(&th2,NULL, the_thread, &info2);  
  
    pthread_join(th1, NULL);  
    pthread_join(th2, NULL);  
    return EXIT_SUCCESS ;  
}
```

```
:> gcc -Wall thread.c -lpthread -o thread  
:> ./thread
```

I'm A

I'm B

I'm A

I'm B



# Threads



Spinning ist aber keine gute Idee:  
Ineffizient und gefährlich!

```
while (1){  
    pthread_mutex_lock(& mutex);  
    if (condition)  
        break; // my turn  
    pthread_mutex_unlock(& mutex);  
}  
// do action  
  
pthread_mutex_unlock(& mutex);
```

Gefragt ist konfigurierbare Funktionalität  
nach dem Vorbild von  
park(), unpark() und setpark(),  
damit gezielt gewartet werden kann!



# Threads



**Bedingte Synchronisation im kritischen Abschnitt.  
(Condition Variable)**

Zwei wichtige Aufgaben:

- **Warten**, bis eine Vorgabe (Condition) erfüllt ist.
- **Signalisieren**, dass eine Vorgabe eingetreten ist.



# Zweiter Baustein: Condition Variables

```
void *the_thread1(void *args, cond1, cond2){  
    for (int i = 0; i < 100; i++){  
        pthread_mutex_lock(& mutex );  
        // wait until condition is true  
  
        printf("I'm %s\n", (char*) args);  
  
        // signal condition changed  
        pthread_mutex_unlock(& mutex );  
  
    }  
    return NULL;  
}
```

warten bis man darf

Was geschieht mit dem Lock,  
das vom wartenden Prozess  
gehalten wird?



Implizit freigeben und  
nach dem Aufwecken  
wieder erwerben.

jetzt darf der andere



# Thread: Condition Variables

- Condition Variable erzeugen
  - `pthread_cond_init()`
  - `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`
- **warten** auf eine Condition und derweil das Lock freigeben
  - `pthread_cond_wait(&cond,&mutex);`
  - `if ( pthread_cond_timedwait(&cond, &mutex, &timespec) == 0);`
- **signalisieren**, dass eine Bedingung eingetreten ist;
  - `pthread_cond_signal(&cond);`
- Datenstruktur wieder freigeben
  - `pthread_cond_destroy(& cond)`

Linux

```
:> man 3 pthread
```



# Threads: Condition Variables



Lösung klar! Bedingung definieren, prüfen und richtig warten, falls sie nicht erfüllt ist.

```
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

static int volatile state;

typedef struct info_t{
    char* name;
    int target; // target state
    int next;   // next state
} info_t;
```



# Threads: Condition Variables

```
void *the_thread(void *args){  
    info_t* info = (info_t*) args;  
    for (int i = 0; i < 100; i++){  
  
        pthread_mutex_lock(&mutex);  
        if (state != info->target)  
            // other turn wait  
            pthread_cond_wait(&cond,&mutex);  
  
        printf("I'm %s\n", (info->name));  
        state = info->next; // changed  
        pthread_cond_signal(&cond);  
        pthread_mutex_unlock(&mutex);  
    }  
    return NULL;  
}
```

```
int main(void)  
{  
    pthread_t th1, th2;  
    info_t info1, info2;  
    info1.name = "A";  
    info2.name = "B";  
    info1.target = info2.next = 0;  
    info2.target = info1.next = 1;  
    state = 0;  
  
    pthread_create(&th1,NULL, the_thread, &info1);  
    pthread_create(&th2,NULL, the_thread, &info2);  
  
    pthread_join(th1, NULL);  
    pthread_join(th2, NULL);  
    return EXIT_SUCCESS ;  
}
```

```
:> gcc -Wall thread.c -lpthread -o thread  
:> ./thread  
I'm A  
I'm B  
I'm A  
I'm B
```



# Threads: Condition Variables

```
int main(void)
{
    pthread_t th1, th2, th3;
    info_t info1, info2, info3;
    info1.name = "A";
    info2.name = "B";
    info3.name = "C";
    info1.target = info2.next = info3.next = 0;
    info2.target = info3.target = info1.next = 1;
    state = 0;

    pthread_create(&th1, NULL, the_thread, &info1);
    pthread_create(&th2, NULL, the_thread, &info2);
    pthread_create(&th3, NULL, the_thread, &info3);

    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    pthread_join(th3, NULL);
    return EXIT_SUCCESS ;
}
```

```
::> gcc -Wall thread.c -lpthread -o thread
::> ./thread
I'm A
I'm B
I'm C
I'm A
```

Was geschieht,  
wenn zwei Threads  
das gleiche tun?

Offensichtlich weckt „B“ statt  
„A“ jetzt „C“!?



# Threads: Condition Variables

```
void *the_thread(void *args){  
    info_t* info = (info_t*) args;  
    for (int i = 0; i < 100; i++){  
  
        pthread_mutex_lock(&mutex);  
        while (state != info->target) ←  
            // other turn wait  
            pthread_cond_wait(&cond,&mutex);  
  
        printf("I'm %s\n", info->name);  
        state = info->next; // changed  
        pthread_cond_signal(&cond);  
        pthread_mutex_unlock(& mutex);  
    }  
    return NULL;  
}
```

Stets die Bedingung neu prüfen.  
Man wird nur geweckt, um den Rest muss man sich selbst kümmern!



# Threads: Condition Variables

```
:> gcc -Wall thread.c -lpthread -o thread  
:> ./thread  
I'm A  
I'm B  
I'm A  
I'm B  
■
```

Offensichtlich haben  
wir einen Bug!  
Warum kam C nicht  
zum Zug?

```
int main(void)  
{  
    pthread_t th1, th2, th3;  
    info_t info1, info2, info3;  
    info1.name = "A";  
    info2.name = "B";  
    info3.name = "C";  
    info1.target = info2.next = info3.next = 0;  
    info2.target = info3.target = info1.next = 1;  
    state = 0;  
  
    pthread_create(&th1, NULL, the_thread, &info1);  
    pthread_create(&th2, NULL, the_thread, &info2);  
    pthread_create(&th3, NULL, the_thread, &info3);  
  
    pthread_join(th1, NULL);  
    pthread_join(th2, NULL);  
    pthread_join(th3, NULL);  
    return EXIT_SUCCESS ;  
}
```



# Threads: Condition Variables

## Überlegung:

1. A, B und C laufen los, jedoch unklar wann und wer wie viel CPU-Zeit bekommt.
2. Sicher ist, dass A nicht warten muss. Falls die anderen schneller sind, warten sie und geben das Lock wieder frei.  
=> A besitzt das Lock, schreibt, ändert den Zustand und signalisiert die Änderung.
3. Falls B und/oder C bereits warten, wird einer geweckt. Falls nicht, wartet einer oder beide auf das Lock oder sie pausieren noch gemäß Scheduling.
4. Sicher ist, dass B oder C als nächste schreiben dürfen, und A warten muss (Lock oder Cond)

**Dies geht solange weiter, bis B oder C den falschen wecken!**  
**Dieser legt sich dann sofort wieder schlafen und keiner verschickt mehr Signale.**



# Threads: Condition Variables

```
void *the_thread(void *args){  
    info_t* info = (info_t*) args;  
    for (int i = 0; i < 100; i++){  
  
        pthread_mutex_lock(&mutex);  
        while (state != info->target)  
            // other turn wait  
            pthread_cond_wait(&cond,&mutex);  
  
        printf("I'm %s\n", info->name);  
        state = info->next; // changed  
        pthread_cond_broadcast(&cond);  
        pthread_mutex_unlock(&mutex);  
    }  
    return NULL;  
}
```

Manchmal muss man auch alle wecken!



# Threads: Condition Variables

```
:> gcc -Wall thread.c -lpthread -o thread
:> ./thread
I'm A
I'm B
I'm A
I'm B
I'm A
I'm C
```

Offensichtlich machte das Programm jetzt das gewünschte (fast).  
Nach einem A folgt ein B oder C und dies in mehr oder weniger  
zufälligem Wechsel.

Leider werden aber zu wenige A produziert, und die beiden  
Threads bleiben immer noch hängen!



# Condition Variables: typische Probleme

- **Signal kommt zu früh und weckt niemanden.**
  - Immer sicherstellen, dass jemand wartet. (Ebenfalls Zustandsinformation)
- **Race Conditions:**

**Noch bevor man wartet, ändert sich der Zustand und die Signalisierung verpufft.**

  - Prüfen-und-Warten, sowie Ändern-und-Signalisieren sind kritische Abschnitte die mit identischen Locks zu versehen sind.
- **Signal weckt den Falschen und dieser legt sich wieder schlafen.**
  - **Signal an alle schicken**
  - **Signale weiterschicken**



# Thread: Condition Variables

- **`pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mtx)`**
  - die Ausführung des Threads wird pausiert
  - wird der Thread geweckt, dann wird das Lock wieder belegt
  - ist das Lock nicht belegt, ist das Verhalten undefiniert
- **`pthread_cond_signal(pthread_cond_t *cond)`**
  - **einer der wartenden** Threads wird geweckt
  - wartet keiner, dann verpufft das Signal
- **`pthread_cond_broadcast(pthread_cond_t *cond)`**
  - **alle wartenden** Threads werden geweckt
  - wartet keiner, dann verpufft das Signal



# Eine typische Aufgabe

In vielen Anwendung fallen Aufgaben an, die nebenläufig bearbeitet werden können und damit Optimierungsmöglichkeiten bieten.

## Beispiel:

Das Schreiben sowie das Lesen von Daten ist sehr zeitintensiv. D.h., Daten werden nie direkt gelesen oder geschrieben, sondern immer gepuffert

- **Lesen:**

Um eine große Datei einzulesen, wird immer nur ein bestimmtes Kontingent von der Platte geladen und in einem Array gespeichert. Ist dieses verbraucht, wird ein neues angefordert. Idealerweise wurde das nächste Kontingent bereits im Voraus **gelesen** und **bereitgehalten**.

- **Schreiben (analog):**

Ist ein Array mit Daten gefüllt, die gespeichert werden sollen, dann wird an einen Writer das **volle Array übergeben** und durch ein **neues leeres Array ersetzt**. Der Writer schreibt die vollen Arrays dann sukzessive auf die Platte und fügt sie in das Kontingent der leeren wieder ein.



# Eine typische Aufgabe: Teil1 „Lesen“

Idee, man braucht diverse Datenstrukturen und Verantwortliche:

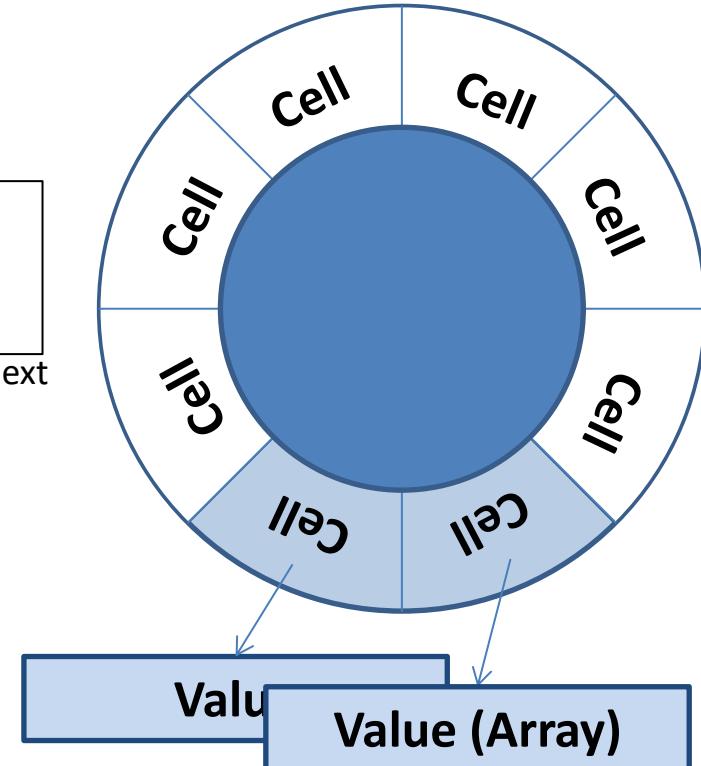
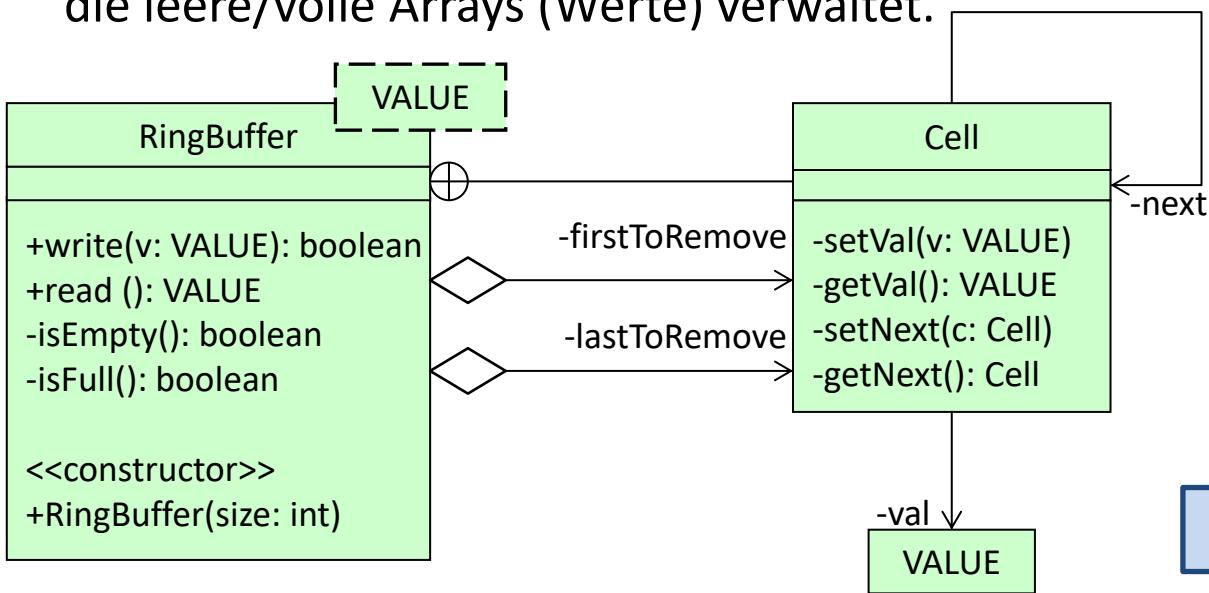
- Eine **Datenstruktur**,
  - die leere Arrays verwaltet.
  - die volle Arrays verwaltet.
- Einen **Verantwortlichen**,
  - der leere Arrays mit Daten aus einer Datei füllt.
  - der volle Arrays verarbeitet und abgearbeitete wieder freigibt.



# Eine typische Aufgabe: Teil1 „Lesen“

## Lösung:

- Eine **Datenstruktur**, die leere/volle Arrays (Werte) verwaltet.





# RingBuffer

**Ein Ringpuffer ermöglicht ein stetiges Schreiben und Lesen!  
(im Uhrzeigersinn)**

- Sobald ein Wert gelesen wurde, rückt der firstToRemove-Zeiger eine Zelle weiter.
- Soll ein Wert geschrieben werden, rückt der lastToRemove-Zeiger eine Zelle weiter, und dann wird an dieser Position der neue Wert eingetragen.

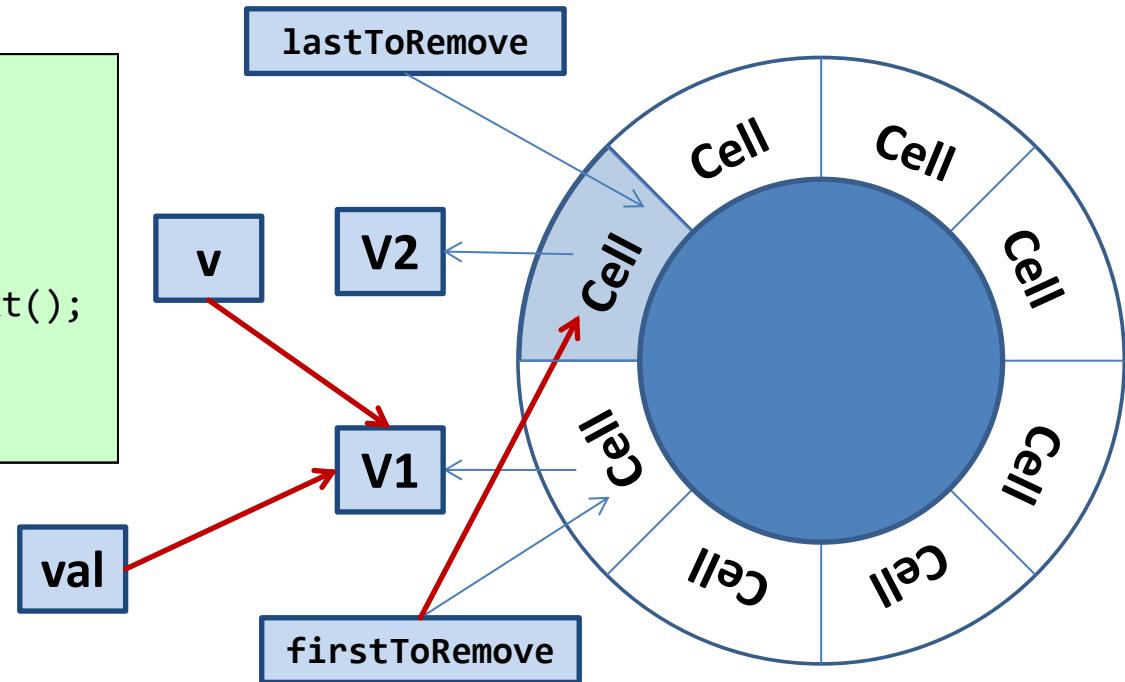


# RingBuffer

- Sobald ein Wert gelesen wurde, rückt der `firstToRemove`-Zeiger eine Zelle weiter.

```
public VALUE read() {  
    if (this.isEmpty())  
        return null;  
    VALUE v = firstToRemove.getVal();  
    firstToRemove = firstToRemove.getNext();  
    return v;  
}
```

```
VALUE val = ringBuffer.read();
```



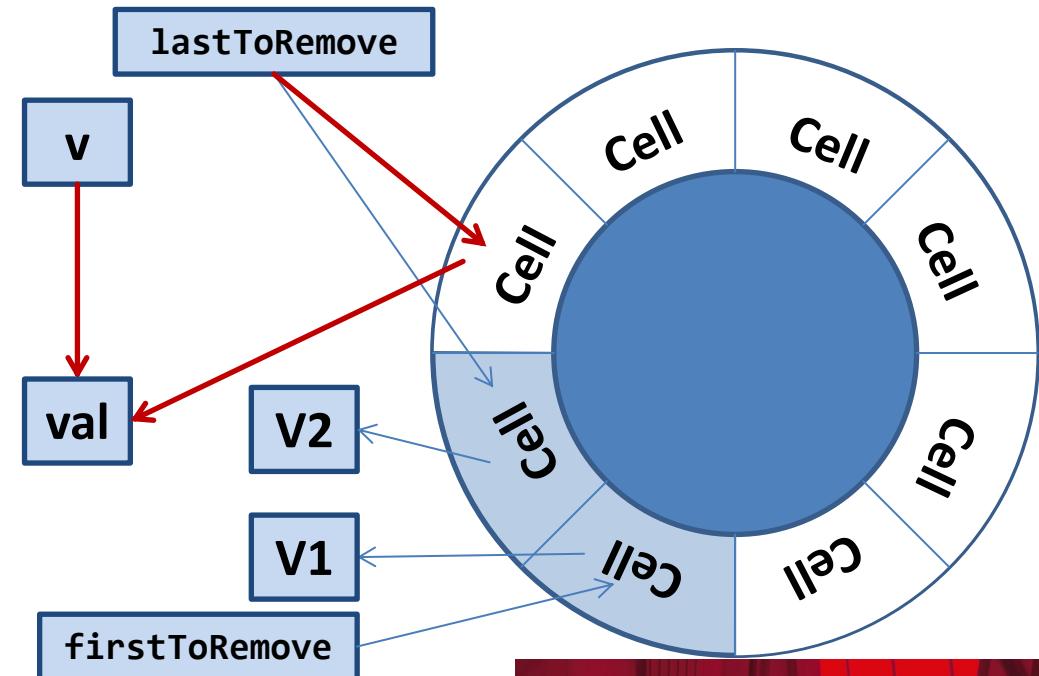


# RingBuffer

- Soll ein Wert geschrieben werden, rückt der `lastToRemove`-Zeiger eine Zelle weiter, und dann wird an dieser Position der neue Wert eingetragen.

```
public boolean write(VALUE v) {  
    if (this.isFull())  
        return false;  
    lastToRemove = lastToRemove.getNext();  
lastToRemove.setVal(v);  
    return true;  
}
```

```
ringBuffer.write(val);
```





# RingBuffer

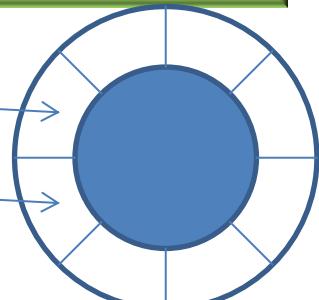
Um einen vollen von einem leeren Puffer zu unterscheiden,  
verfügt der Puffer über eine extra Zelle.

- Ein Puffer ist leer falls:

**lastToRemove.next = firstToRemove**

firstToRemove

lastToRemove

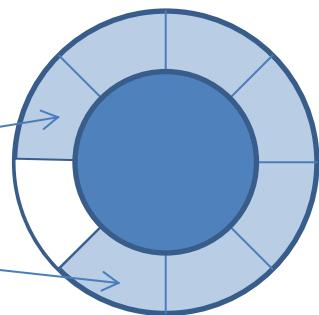


- Ein Puffer ist voll falls

**lastToRemove.next.next = firstToRemove**

firstToRemove

lastToRemove





# RingBuffer

```
typedef struct ring_buffer_t {  
    int size;  
    void** cell;  
    int first_to_remove;  
    int last_to_remove;  
} ring_buffer_t;
```

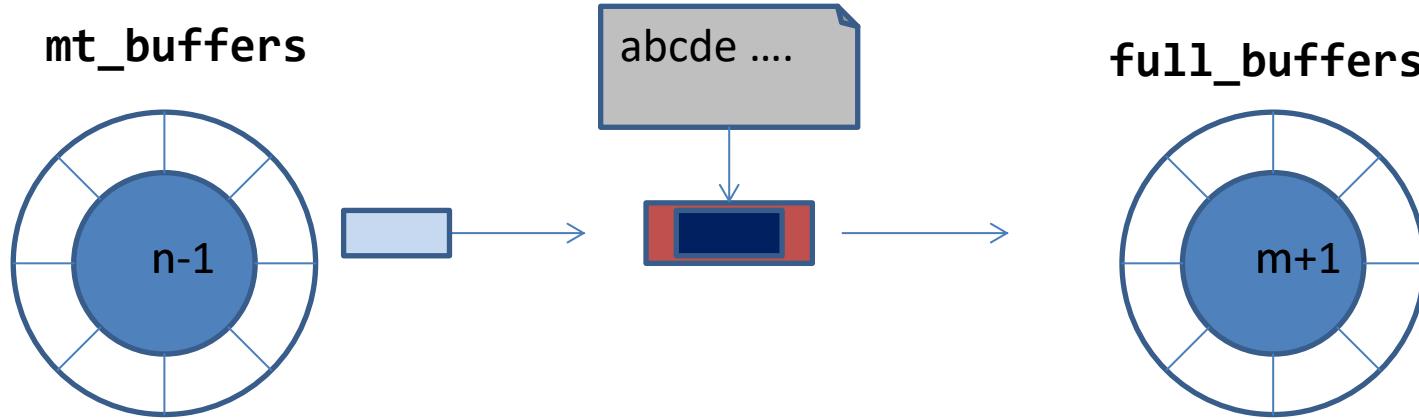
```
int write_rb(ring_buffer_t *buf, void *val) {  
    if (full_rb(buf))  
        return 0;  
    buf->last_to_remove = (buf->last_to_remove + 1) % buf->size;  
    buf->cell[buf->last_to_remove] = val;  
    return 1;  
}
```

```
void *read_rb(ring_buffer_t *buf) {  
    if (mt_rb(buf))  
        return NULL;  
    void* val = buf->cell[buf->first_to_remove];  
    buf->first_to_remove = (buf->first_to_remove + 1) % buf->size;  
    return val;  
}
```



# Eine typische Aufgabe: Teil 1 „Lesen“

- Einen Verantwortlichen, der leere Arrays mit Daten aus einer Datei füllt.





# Eine typische Aufgabe: Teil 1 „Lesen“

- Einen Verantwortlichen, der leere Arrays mit Daten aus einer Datei füllt.

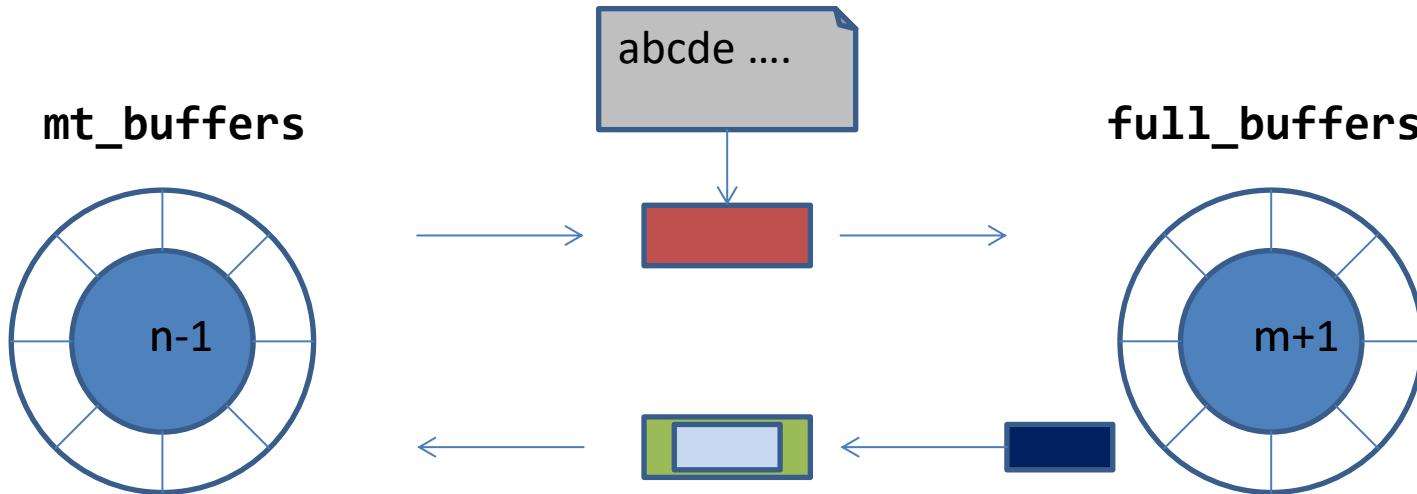
```
void fill_buffer(ring_buffer_t *mt_buffers, ring_buffer_t *full_buffers, int fd) {  
    char_buffer_t* bp = NULL;  
  
    bp = (char_buffer_t *) read_rb(mt_buffers);  
  
    bp->size = read(fd, &bp->cbuf, MAX_BUFFER_SIZE);  
  
    write_rb(full_buffers, bp);  
}
```

```
#define MAX_BUFFER_SIZE 512  
  
typedef struct char_buffer_t {  
    int size;  
    char cbuf[MAX_BUFFER_SIZE];  
} char_buffer_t;
```



# Eine typische Aufgabe: Teil 1 „Lesen“

- Einen Verantwortlichen, der leere Arrays mit Daten aus einer Datei füllt.
- Einen weiteren, der volle Arrays verarbeitet und abgearbeitete wieder freigibt.





# Eine typische Aufgabe: Teil 1 „Lesen“

- Einen Verantwortlichen, der leere Arrays mit Daten aus einer Datei füllt.
- Einen weiteren, der volle Arrays verarbeitet und abgearbeitete wieder freigibt.

```
void work_on_buffer(ring_buffer_t *mt_buffers, ring_buffer_t *full_buffers) {  
    char_buffer_t* bp = NULL;  
  
    bp = (char_buffer_t *) read_rb(full_buffers);  
  
    // do something  
  
    write_rb(mt_buffers, bp);  
}
```



# Eine typische Aufgabe: Teil 1 „Lesen“

## Probleme:

- Was ist wenn die Ringpuffer leer bzw. voll sind?
  - eine Write- / Read-Operation nicht durchgeführt werden kann.
- Die beiden Aufgaben („fill“ und „work on“) laufen parallel ab und greifen auf dieselben Datenstrukturen zu!

```
if (mt_rb(buf)) return NULL;
```

```
if (full_rb(buf)) return 0;
```



**Lösung klar! Absichern und warten.**



# RingBuffer mit Mutex und Condition Variables

```
typedef struct ring_buffer_t {  
    int size;  
    void** cell;  
    int first_to_remove;  
    int last_to_remove;  
} ring_buffer_t;
```



```
typedef struct ring_buffer_t {  
    pthread_cond_t *cond_new_cell_written;  
    pthread_cond_t *cond_some_cell_removed;  
    pthread_mutex_t *lock;  
    int size;  
    void** cell;  
    int first_to_remove;  
    int last_to_remove;  
} ring_buffer_t;
```



# Eine typische Aufgabe: Teil 1 „Lesen“

- Einen Verantwortlichen, der leere Arrays mit Daten aus einer Datei füllt.

```
void fill_buffer(ring_buffer_t *mt_buffers, ring_buffer_t *full_buffers, int fd) {  
    char_buffer_t* bp = NULL;  
    while (1){  
        pthread_mutex_lock(mt_buffers->lock);  
        while ((bp = (char_buffer_t *) read_rb(mt_buffers)) == NULL)  
            pthread_cond_wait(mt_buffers->cond_new_cell_written, mt_buffers->lock);  
        pthread_mutex_unlock(mt_buffers->lock);  
  
        bp->size = read(fd, &bp->cbuf, MAX_BUFFER_SIZE);  
  
        pthread_mutex_lock(full_buffers->lock);  
        while (!write_rb(full_buffers, bp))  
            pthread_cond_wait(full_buffers->cond_some_cell_removed, full_buffers->lock);  
        pthread_mutex_unlock(full_buffers->lock);  
    }  
}
```



# Eine typische Aufgabe: Teil 1 „Lesen“

- Einen Verantwortlichen, der leere Arrays mit Daten aus einer Datei füllt.
- Einen weiteren, der volle Arrays verarbeitet und abgearbeitete wieder freigibt.

```
void work_on_buffer(ring_buffer_t *mt_buffers, ring_buffer_t *full_buffers) {  
    char_buffer_t* bp = NULL;  
    while (1){  
        pthread_mutex_lock(full_buffers->lock);  
        while ((bp = (char_buffer_t *) read_rb(full_buffers)) == NULL)  
            pthread_cond_wait(full_buffers->cond_new_cell_written, full_buffers->lock);  
        pthread_mutex_unlock(full_buffers->lock);  
        // do something  
        pthread_mutex_lock(mt_buffers->lock);  
        while (!write_rb(mt_buffers, bp))  
            pthread_cond_wait(mt_buffers->cond_some_cell_removed, mt_buffers->lock);  
        pthread_mutex_unlock(mt_buffers->lock);  
    }  
}
```



# Eine typische Aufgabe: Teil 1 „Lesen“

Probleme: Wer sendet die Signale?

```
void fill_buffer(ring_buffer_t *mt_buffers, ring_buffer_t *full_buffers, int fd) {  
    pthread_cond_wait(mt_buffers->cond_new_cell_written, mt_buffers->lock);  
    pthread_cond_wait(full_buffers->cond_some_cell_removed, full_buffers->lock);  
}
```

```
void work_on_buffer(ring_buffer_t *mt_buffers, ring_buffer_t *full_buffers) {  
    pthread_cond_wait(full_buffers->cond_new_cell_written, full_buffers->lock);  
    pthread_cond_wait(mt_buffers->cond_some_cell_removed, mt_buffers->lock);  
}
```



**Lösung: Dies erfolgt wechselseitig.**



# Eine typische Aufgabe: Teil 1 „Lesen“

```
void fill_buffer(ring_buffer_t *mt_buffers, ring_buffer_t *full_buffers, int fd) {
    char_buffer_t* bp = NULL;
    while (1){
        pthread_mutex_lock(mt_buffers->lock);
        while ((bp = (char_buffer_t *) read_rb(mt_buffers)) == NULL)
            pthread_cond_wait(mt_buffers->cond_new_cell_written, mt_buffers->lock);
        pthread_cond_signal(mt_buffers->cond_some_cell_removed);
        pthread_mutex_unlock(mt_buffers->lock);

        bp->size = read(fd, &bp->cbuf, MAX_BUFFER_SIZE);

        pthread_mutex_lock(full_buffers->lock);
        while (!write_rb(full_buffers, bp))
            pthread_cond_wait(full_buffers->cond_some_cell_removed, full_buffers->lock);
        pthread_cond_signal(full_buffers->cond_new_cell_written);
        pthread_mutex_unlock(full_buffers->lock);
    }
}
```



# Eine typische Aufgabe: Teil 1 „Lesen“

```
void work_on_buffer(ring_buffer_t *mt_buffers, ring_buffer_t *full_buffers) {
    char_buffer_t* bp = NULL;
    while (1){
        pthread_mutex_lock(full_buffers->lock);
        while ((bp = (char_buffer_t *) read_rb(full_buffers)) == NULL)
            pthread_cond_wait(full_buffers->cond_new_cell_written, full_buffers->lock);
        pthread_cond_signal(full_buffers->cond_some_cell_removed);
        pthread_mutex_unlock(full_buffers->lock);

        // do something

        pthread_mutex_lock(mt_buffers->lock);
        while (!write_rb(mt_buffers, bp))
            pthread_cond_wait(mt_buffers->cond_some_cell_removed, mt_buffers->lock);
        pthread_cond_signal(mt_buffers->cond_new_cell_written);
        pthread_mutex_unlock(mt_buffers->lock);
    }
}
```



# Eine typische Aufgabe: Teil 2 „Schreiben“

Idee: Man braucht diverse Datenstrukturen und Verantwortliche!

- Eine **Datenstruktur**,
  - die leere Arrays verwaltet.
  - die volle Arrays verwaltet.
- Einen **Verantwortlichen**,
  - der leere Arrays verarbeitet und mit sinnvollen Daten füllt.
  - der volle Arrays mit Daten in eine Datei schreibt und wieder freigibt.

Offensichtlich  
die analoge  
Aufgabe.



# Ein Konzept: Producer Consumer

**Typische Situation:**

**Man hat zwei parallel ablaufende Threads, der eine Thread liefert etwas, das der andere Thread verbraucht.**

- **Producer (Erzeuger):**

Der Producer erzeugt ein Datum und informiert den Consumer, wenn dieses bereitsteht. Danach wartet er, bis er ein neues erzeugen muss.

- **Consumer (Verbraucher):**

Der Consumer nutzt das vom Producer erzeugte Datum und informiert diesen sobald er es verarbeitet (verbraucht) hat. Danach wartet er auf ein neues Datum.



# Producer Consumer: Producer Thread

```
void* the_producer(void *args){  
    product_collection_t *products = (product_collection_t *) args;  
  
    while (1){  
        pthread_mutex_lock(products->lock);  
        while (products->full)  
            pthread_cond_wait(products->cond_consumed, products->lock);  
  
        add_some_item(products);  
  
        pthread_cond_signal(products->cond_filled);  
        pthread_mutex_unlock(products->lock);  
    }  
    return NULL;  
}
```

```
typedef struct prod_collection_t {  
    pthread_cond_t *cond_consumed;  
    pthread_cond_t *cond_filled;  
    pthread_mutex_t *lock;  
    int full;  
    int empty;  
    ...  
} product_collection_t;
```



# Producer Consumer: Consumer Thread

```
void* the_consumer(void *args){  
    product_collection_t *products = (product_collection_t *) args;  
  
    while (1){  
        pthread_mutex_lock(products->lock);  
        while (products->empty)  
            pthread_cond_wait(products->cond_filled, products->lock);  
  
        consume_some_item(products);  
  
        pthread_cond_signal(products->cond_consumed);  
        pthread_mutex_unlock(products->lock);  
    }  
    return NULL;  
}
```

```
typedef struct prod_collection_t {  
    pthread_cond_t *cond_consumed;  
    pthread_cond_t *cond_filled;  
    pthread_mutex_t *lock;  
    int full;  
    int empty;  
    ...  
} product_collection_t;
```



# Ein Konzept: Producer Consumer

**Achtung: Oftmals existieren mehrere Consumer und Producer gleichzeitig**

- Zustände immer mittels einer Schleife überprüfen.
  - Bei mehreren Threads kann man auch zwischen dem Empfang des Signals und dem Wiederbesitzen des Mutex unterbrochen werden.
  - Der Empfänger kann nicht unterscheiden, ob er durch ein cond\_signal oder cond\_broadcast geweckt wurde.

```
while (state)
    pthread_cond_wait(&cond, &lock);
```

**<<korrekt>>**

```
if (state)
    pthread_cond_wait(&cond, &lock);
```

**<<falsch>>**



# Ein Konzept: Producer Consumer

**Achtung: Oftmals existieren mehrere Consumer und Producer gleichzeitig**

- Vor dem Senden eines Signals immer das entsprechende Lock erwerben.

- Sonst könnte das Signal verloren gehen.
- Sonst könnte der Empfänger ewig warten.

```
pthread_mutex_lock(&lock);
while (state)
    pthread_cond_wait(&cond, &lock);
```

```
pthread_cond_signal(&cond);
pthread_mutex_unlock(&lock);
```

<<korrekt>>

```
pthread_mutex_unlock(&lock);
pthread_cond_signal(&cond);
```

<<falsch>>



# Ein Konzept: Producer Consumer

Die möglichen Zustände, die die Datenstruktur annehmen kann und die Condition Variables müssen aufeinander abgestimmt sein.

```
change_state(data); // s0->s1  
  
pthread_cond_signal(&state1);  
pthread_mutex_unlock(&lock);
```

```
pthread_mutex_lock(&lock);  
while (!data->state == s1)  
    pthread_cond_wait(&state1,&lock);
```



# Aufgabe

**Schreiben Sie ein C-Programm, das den Inhalt einer Datei in eine andere kopiert.**

- Nutzen Sie die zuvor definierten, Datenstrukturen und Algorithmen.
- Überlegen Sie sich wie und wann Sie die Threads terminieren.
- Was tun Sie im Fehlerfall?

**Infos zum Lesen und Schreiben von  
und in Dateien findet man unter:  
[man 2 read] [man 2 write]**



# Aufgabe

```
int main(int argc, char *argv[])
{
    pthread_t th1, th2;
    info_t info1, info2;
    char* source = argv[1];
    char* target = argv[2];

    // initialize data structures here

    pthread_create(&th1, NULL, the_thread, &info1);
    pthread_create(&th2, NULL, the_thread, &info2);

    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    return EXIT_SUCCESS ;
}
```

```
:> gcc -Wall copy.c -lpthread -o copy
:> ./copy source.txt target.txt
:>
```



# Synchronisation von Threads

## Semaphore



# Semaphore

**Mutex und Condition Variable sind aus Sicht einer Anwendung zustandslos.**

- Ein Mutex kann man erwerben und freigeben.
- Ein Condition Variable ermöglicht nur das Warten auf ein Ereignis.



**Darüber hinaus gehende Informationen müssen separat verwaltet werden.**



# Semaphore

Ein alternatives Synchronisationsprimitiv  
zustandsbasierte Locks  
„Semaphore“.

[Dijkstra, E.W.: Cooperating sequential processes.1968]

- jedes Semaphor verfügt über einen definierten Wert ( $\text{Integer } \geq 0$ )
- spezielle Operationen können diesen Wert erhöhen oder erniedrigen
  - **P()** : „probeer te verlagen“ „versuchen zu senken“
  - **V()**: „verhoog“ „erhöhen“
- Wird der Wert negativ, blockiert die P-Operation



# Semaphore

Linux (POSIX)

- Semaphor anlegen
  - `sem_t *sem_open(char *name, int oflag)`
  - `sem_init(sem_t *sem, int pshared, unsigned int value)`
- P-Operation durchführen
  - `sem_wait(sem_t *sem)`
  - `sem_trywait(sem_t *sem)`
  - `sem_timedwait(sem_t *sem, timespec *timeout)`
- V-Operation
  - `sem_post(sem_t *sem)`
- Datenstruktur wieder freigeben
  - `sem_close(sem_t *sem);`



# Semaphore

- Semaphor anlegen
  - `new Semaphore(int permits)`
- P-Operation durchführen
  - `acquire();`
  - `acquireUninterruptibly();`
  - `tryAcquire();`
- V-Operation
  - `release();`
- Datenstruktur wieder freigeben
  - **Garbage Collector**

Java



# Semaphore

Ein äquivalentes  
Konzept.

Mutexes  
und  
Condition Variables



Semaphore



# Ein äquivalentes Konzept

Mutexes und Condition Variables



Semaphore

- mit Hilfe von Mutexes und Condition Variables lassen sich Semaphore realisieren

```
typedef struct sem_t {  
    ...  
} sem_t ;  
  
void sem_init(sem_t *sem, unsigned int val)    {...}  
void sem_wait(sem_t *sem) {...}  
void sem_post(sem_t *sem) {...}
```



# Ein äquivalentes Konzept

```
typedef struct sem_t {  
    unsigned int val;  
    mutex_t mtx;  
    cond_t cond;  
} sem_t ;  
  
void sem_init(sem_t *sem, unsigned int val) {  
    mutex_init(&sem->mtx);  
    cond_init(&sem->cond);  
    sem->val = val;  
}
```

```
void sem_wait(sem_t *sem) {  
    mutex_lock(&sem->mtx);  
    while (sem->val == 0)  
        cond_wait(&sem->cond, &sem->mtx);  
    sem->val--;  
    mutex_unlock(&sem->mtx);  
}  
  
void sem_post(sem_t *sem) {  
    mutex_lock(&sem->mtx);  
    sem->val++;  
    cond_signal(&sem->cond);  
    mutex_unlock(&sem->mtx);  
}
```



# Ein äquivalentes Konzept

Semaphore



Mutexes und Condition Variables

1. mit Hilfe von Semaphoren werden Mutexes realisiert
2. mit Hilfe von Semaphoren werden Condition Variables realisiert

```
typedef struct mutex_t {  
...  
} mutex_t;  
  
void mutex_init(mutex_t *mtx)    {...}  
void mutex_lock(mutex_t *mtx)   {...}  
void mutex_unlock(mutex_t *mtx) {...}
```

```
typedef struct cond_t {  
...  
} cond_t;  
  
void cond_signal(cond_t *cnd) {...}  
void cond_wait(cond_t *cnd, mutex_t *lock){...}
```



# Ein äquivalentes Konzept

Semaphore



Mutexes

```
typedef struct mutex_t {  
    sem_t sem;  
} mutex_t;  
  
void mutex_init(mutex_t *mtx) {  
    sem_init(&mtx->sem, 0, 1); // at begin the mutex isn't locked  
}  
  
void mutex_lock(mutex_t *mtx) {  
    sem_wait(&mtx->sem);  
}  
void mutex_unlock(mutex_t *mtx) {  
    sem_post(&mtx->sem);  
}
```



# Ein äquivalentes Konzept

Semaphore



Condition Variables

```
typedef struct cond_t {  
    mutex_t mtx; // protecting operations  
    sem_t sem;  
    int waiting  
} cond_t;  
  
void cond_init(cond_t *cnd) {  
    sem_init(&cnd->sem, 0, 0); // one has to wait  
    cnd->waiting = 0;          // nobody waits  
    mutex_init(&cnd->mtx)  
}
```



# Ein äquivalentes Konzept

Semaphore



Condition Variables

```
void cond_signal(cond_t *cnd) {  
    mutex_lock(&cnd->mtx);  
    if (cnd->waiting > 0){  
        cnd->waiting--;  
        sem_post(cnd->sem); // wake-up  
    }  
    mutex_unlock(&cnd->mtx);  
}
```

```
void cond_wait(cond_t *cnd, mutex_t *lock) {  
    mutex_unlock(lock); // release the lock  
    mutex_lock(&cnd->mtx); // protect sem_op  
    cnd->waiting++;  
    mutex_unlock(&cnd->mtx);  
    sem_wait(&cnd->sem); // wait  
    mutex_lock(lock); // acquire the lock again  
}
```



# Ein Konzept: Producer Consumer

**Typische Situation:**

**Man hat zwei parallel ablaufende Threads, der eine Thread liefert etwas, das der andere Thread verbraucht.**

- **Producer (Erzeuger):**

Der Producer erzeugt ein Datum und informiert den Consumer, wenn dieses bereitsteht. Danach wartet er, bis er ein neues erzeugen muss.

- **Consumer (Verbraucher):**

Der Consumer nutzt das vom Producer erzeugte Datum und informiert diesen sobald er es verarbeitet (verbraucht) hat. Danach wartet er auf ein neues Datum.



# Ein Konzept: Producer Consumer

- **Producer (Erzeuger):**

Der Producer erzeugt ein Datum und informiert den Consumer, wenn dieses bereitsteht. Danach wartet er, bis er ein neues erzeugen muss.

- **Consumer (Verbraucher):**

Der Consumer nutzt das vom Producer erzeugte Datum und informiert diesen sobald er es verarbeitet (verbraucht) hat. Danach wartet er auf ein neues Datum.

```
while(1){  
    P(products->empty);  
    add_some_item(products);  
    V(products->full);  
}
```

```
while(1){  
    P(products->full);  
    consume_some_item(products);  
    V(products->empty);  
}
```



# Ein Konzept: Producer Consumer

Achtung: Falls der Container mehrere Daten aufnehmen kann, dann müssen zusätzlich die Operationen „Einfügen“ und „Entnahme“ abgesichert werden.

- Producer (Erzeuger):

```
while(1){  
    P(products->empty);  
    P(products->lock);  
    add_some_item(products);  
    V(products->lock);  
    V(products->full);  
}
```

- Consumer (Verbraucher):

```
while(1){  
    P(products->full);  
    P(products->lock);  
    consume_some_item(products);  
    V(products->lock);  
    V(products->empty);  
}
```



# Producer Consumer: Producer Thread

```
void* the_producer(void *args){  
    product_collection_t *products = (product_collection_t *) args;  
  
    while (1){  
        sem_wait(&products->empty);  
  
        sem_wait(&products->lock);  
        add_some_item(products);  
        sem_post(&products->lock);  
  
        sem_post(&products->full);  
    }  
    return NULL;  
}
```

```
typedef struct prod_collection_t {  
    pthread_sem_t full;  
    pthread_sem_t empty;  
    pthread_sem_t lock;  
    ...  
} product_collection_t;
```



# Producer Consumer: Consumer Thread

```
void* the_consumer(void *args){  
    product_collection_t *products = (product_collection_t *) args;  
  
    while (1){  
        sem_wait(&products->full);  
  
        sem_wait(&products->lock);  
        consume_some_item(products);  
        sem_post(&products->lock);  
  
        sem_post(&products->empty);  
    }  
    return NULL;  
}
```

```
typedef struct prod_collection_t {  
    pthread_sem_t full;  
    pthread_sem_t empty;  
    pthread_sem_t lock;  
    ...  
} product_collection_t;
```



# Ein Konzept: Producer Consumer

**Achtung: Die Zustände des Containers werden über die Semaphore identifiziert!**

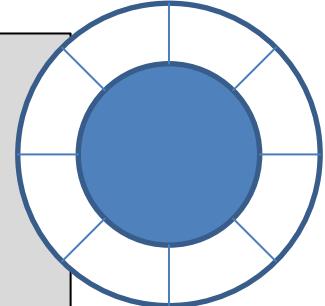


**Dies muss bei der Initialisierung der Datenstruktur beachtet werden.**



# Beispiel: RingBuffer

```
ring_buffer_t *create_rb(int size){  
    if (size < 1)  
        return NULL;  
    ring_buffer_t *rb = (ring_buffer_t *) malloc(sizeof(ring_buffer_t));  
    rb->size = size + 1;  
    rb->cell = (void**) malloc(sizeof(void*)*rb->size)  
    rb->first_to_remove = 1;  
    rb->last_to_remove = 0;  
  
    sem_init(&rb->full, 0, 0);  
    sem_init(&rb->empty, 0, size);  
    sem_init(&rb->lock, 0, 1);  
  
    return rb;  
}
```



```
typedef struct ring_buffer_t {  
    pthread_sem_t empty;  
    pthread_sem_t full;  
    pthread_sem_t lock;  
    int size;  
    void** cell;  
    int first_to_remove;  
    int last_to_remove;  
} ring_buffer_t;
```



# Semaphore

**Semaphore sind äquivalent zu Mutexes in Verbindung mit Condition Variables**

- Beide ermöglichen den „wechselseitigen Ausschluss“ und die Organisation von konkurrierenden Abläufen.
- Semaphore verfügen über einen Zustand, die Initialisierung definiert die Aufgabe
  - 0 => um auf ein Ereignis zu warten
  - 1 => um als binäres Lock zu fungieren
  - n => um verfügbare Ressourcen zu definieren



# Aufgabe

**Schreiben Sie ein zweites C-Programm, das den Inhalt einer Datei in eine andere kopiert.**

- Nutzen Sie Semaphore statt Mutexes und Condition Variables.
- Orientieren Sie sich an Ihrer vorherigen Lösung.



# Synchronisation von Threads

## Monitore



# Objektorientierung

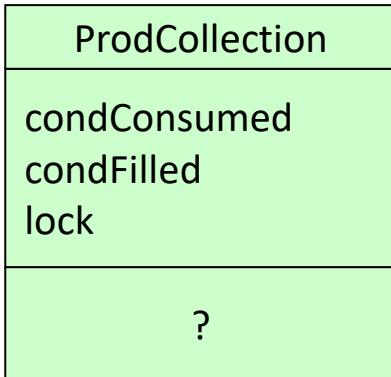
**Moderne Software ist objektorientiert!**

- **ganzheitliche Vorgehensweise** – beschrieben werden:
    - Daten,
    - Funktion und
    - deren Zusammenhänge
  - **intuitiv und kommunikativ** – das Problem wird gelöst durch:
    - Modellieren und
    - Simulieren
- } **das Programm als Abbild der realen Welt**



# Objektorientierung

Objekte verbinden Daten und Funktionalität in einer Einheit,  
die in Form einer Klasse definiert wird.



```
typedef struct prod_collection_t {  
    pthread_cond_t *cond_consumed;  
    pthread_cond_t *cond_filled;  
    pthread_mutex_t *lock;  
    int full;  
    int empty;  
    ...  
} product_collection_t;
```



# Objektorientierung

**Bisher folgt weder die Verwendung von Mutex und Condition Variable noch die von Semaphoren dem objektorientierten Paradigma!**

bisher:

- Synchronisation und Koordination von Threads
- Gegenseitiger Ausschluss aus einem kritischen Abschnitt



**nicht Objektorientierung**



# Idee

**Zusammenfassung der innerhalb der kritischen Bereiche liegenden Daten und Zugriffsfunktionen zu einer abgeschlossenen Einheit.**



**ein Monitor**



# Monitor

- bisher

```
typedef struct data_t {  
    ...  
} data_t;
```

```
data_t *data;  
...  
sem_wait(&lock);  
read_or_write(data);  
sem_post(&lock);
```

```
data_t *data;  
...  
sem_wait(&lock);  
read_or_write(data);  
sem_post(&lock);
```

- Monitor

MonitoredCollection
- cond: CV - data: Data
-wait(); ... + read(); +write();

[Hoare, C.A.R. Monitors: An Operating System Structuring Concept. Communications of the ACM, 17(10), 1974.]



# Monitor

## Elementare (private) Operationen nach Hoare

- **wait(condition):**  
den aktuellen Prozess (Thread) in den Zustand „warten“ versetzen.
- **signal (condition):**  
einen Prozess (Thread), der wartet wecken.
- **queue(condition):**  
prüfen, ob es Prozesse (Threads) gibt, die warten.

[Hoare, C.A.R. Monitors: An Operating System Structuring Concept.  
Communications of the ACM, 17(10), 1974.]



# Monitor: Besonderheiten

**Locks sind implizit!**

- Jeder Prozess kann öffentliche Methode des Monitors aufrufen.
- Nur ein Prozess kann sie jeweils ausführen. Alle anderen werden blockiert,
  - bis die Methode beendet ist oder
  - bis `wait()` aufgerufen wird.
- `signal()` weckt einen wartenden Prozess.

**Wird ein Prozess (Thread) per „`signal()`“ geweckt,  
dann bekommt er garantiert das Lock!**



# Monitor: Beispiel

## Readers and Writers!

[Hoare, C.A.R. Monitors: An Operating System Structuring Concept.  
Communications of the ACM, 17(10), 1974.]

- **Problem:**  
Viele unterschiedliche Akteure wollen auf einen gemeinsamen Datensatz zugreifen.
- **Lesende Zugriffe sind unkritisch:**  
Solange niemand schreibt, können beliebig viele Akteure (auch parallel) auf den Datensatz zugreifen.
- **Schreibende Zugriff sind kritisch:**  
Es darf immer nur ein Akteur schreiben. Insbesondere darf in dieser Phase auch kein anderer lesen! Jeder Prozess (Thread) kann eine Methode des Monitors aufrufen.
- **Fairness ist gefragt!**



# Reader and Writer (Fairness-Regeln)

- Wenn viele Leser existieren, dann muss sichergestellt sein, dass ein Writer die Daten verändern kann.

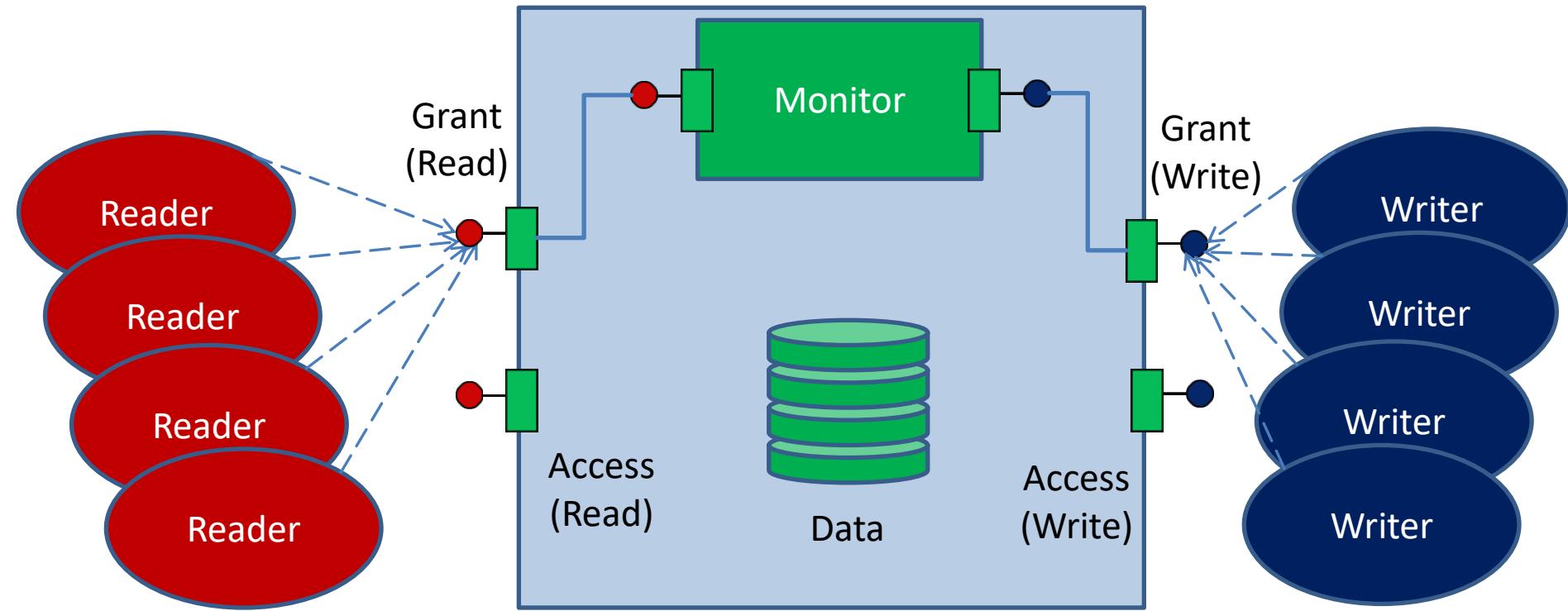
**Wartet ein Writer, dann müssen sich  
neue Reader hinter ihm anstellen.**

- Wenn viele Writer existieren, dann muss sichergestellt sein, dass Daten weiterhin gelesen werden können.

**Müssen Reader während eines Write-Vorgangs warten,  
erhalten Sie Vorrang vor allen anderen wartenden Writern.**



# Reader and Writer





# Reader and Writer (Probleme)

- Das Monitor-Konzept sieht vor, dass immer nur eine Prozess aktiv ist.
- Die Aufgabe erfordert es aber, dass Reader parallel (gleichzeitig) auf den Daten arbeiten.



**Nicht die Daten, sondern Verwaltungsinformationen werden durch den Monitor abgesichert.**



# Reader and Writer (abgesicherte Daten und Operation)

## Operationen:

- startWrite()
- endWrite()
- startRead()
- endRead()

## abgesicherte Daten:

- readerCount
- busy

ReaderWriterMonitor
- ok2write: CV - ok2read: CV - readerCount: int - busy: boolean
+ startWrite() + endWrite() + startRead() + endRead()  -wait(cond: CV) -signal(cond: CV) -queue(cond:CV) : boolean

## Condition Variables:

- ok2read
- ok2write



# Reader and Writer (allgemeine Lösung)

- Initialisierung

```
class ReaderWriterMonitor {  
    private CondVariable ok2read;  
    private CondVariable ok2write;  
    private int readerCount = 0;  
    private boolean busy = false; // nobody is writing  
  
    ReaderWriterMonitor (){  
        this.ok2read = new CondVariable();  
        this.ok2write = new CondVariable();  
    }  
    ...  
}
```



# Reader and Writer (allgemeine Lösung)

- **write()**

```
class ReaderWriterMonitor {  
    private CondVariable ok2read;  
    private CondVariable ok2write;  
    private int readerCount = 0;  
    private boolean busy = false; // nobody is writing
```

```
ReaderWriterMonitor (){  
    this.ok2read = new CondVariable();  
    this.ok2write = new CondVariable();  
}  
...  
}
```

```
public startWrite(){  
    if (this.busy || this.readerCount != 0)  
        this.wait(this.ok2write);  
    busy = true;  
}
```

```
public endWrite(){  
    this.busy = false;  
    if (this.queue(this.ok2read))  
        this.signal(this.ok2read);  
    else  
        this.signal(this.ok2write);  
}
```



# Reader and Writer (allgemeine Lösung)

- **read()**

```
class ReaderWriterMonitor {  
    private CondVariable ok2read;  
    private CondVariable ok2write;  
    private int readerCount = 0;  
    private boolean busy = false; // nobo
```

```
ReaderWriterMonitor (){  
    this.ok2read = new CondVariable();  
    this.ok2write = new CondVariable();  
}  
...  
}
```

```
public startRead(){  
    if (this.busy || this.queue(this.ok2write))  
        this.wait(this.ok2read);  
    readerCount++;  
    // once one can read they all can read  
    this.signal(this.ok2read);  
}
```

```
public endRead(){  
    this.readerCount--;  
    if (this.readerCount == 0)  
        this.signal(this.ok2write);  
}
```



# Monitor

## Vorteile:

- die Synchronisationsfunktionalität ist in einem Objekt gekapselt
- die Operationen können frei definiert werden
- die Programmstruktur ist übersichtlicher



**Weniger fehleranfällig.**



# Monitor

## Nachteile:

- keine echte Objektorientierung
- implizite Locks beschränken die tatsächliche Funktionalität
- die Signal-Semantik nach Hoare ist unrealistisch



**Keine reale Implementierung, lediglich  
ein erfolgreiches Konzept!**



# Monitor: Ideal und Realität

Monitore nach Hoare und die Semantik von signal() (**das Ideal**)

- startet einen wartenden Prozess sofort
- blockiert den sendenden Prozess sofort

**Der Sender muss sicherstellen, dass die Warte-Bedingung nicht mehr erfüllt ist!**

```
public startWrite(){  
    if (this.busy || this.readerCount != 0)  
        this.wait(this.ok2write);  
    busy = true;  
}
```



# Monitor: Ideal und Realität

## MESA-Semantik für signal() (die Realität)

- der wartende Prozess läuft erst los,
  - wenn der Sender explizit blockiert (wait) oder,
  - die Sender-Methode terminiert.
- der Empfang eines Signal ist lediglich ein Hinweis, dass sich etwas geändert hat und **keine Garantie**.



**Der Empfänger muss sicherstellen, dass die Warte-Bedingung nicht mehr erfüllt ist!**

[B.W. Lampson and Redell, D.R.  
Experience with Processes and Monitors in Mesa.  
Communications of the ACM, 23(2), 1980.]



# Monitor: Ideal und Realität

```
public startWrite(){
    while (this.busy || this.readerCount != 0)
        this.wait(this.ok2write);
    busy = true;
}
```



# Reader and Writer in Java

## ReaderWriterMonitor

- ok2read: CV  
- ok2write: CV  
- readerCount: int  
- busy: boolean

+ startRead()  
+ endRead()  
+ startWrite()  
+ endWrite()

-wait(cond: CV)  
-signal(cond: CV)  
-queue(cond:CV) : boolean

## Problem: (vor Java 5)

- keine Condition Variables
- lediglich
  - wait,
  - synchronized und
  - unspezifische Weck-Operationen
    - notify() bzw.
    - notifyAll()

Zum Glück hat sich dies geändert!



# Reader and Writer in Java

## ReaderWriterMonitor

- lock: ReentrantLock
- ok2read: Condition
- ok2write: Condition
- readerCount: int
- busy: boolean

- + startRead()
- + endRead()
- + startWrite()
- + endWrite()

```
class ReaderWriterMonitor {  
  
    private ReentrantLock lock = new ReentrantLock();  
    private Condition ok2read = lock.newCondition();  
    private Condition ok2write = lock.newCondition();  
  
    private int readerCount = 0;  
    private boolean busy = false; // nobody is writing  
  
    ...  
}
```



# Reader and Writer in Java

```
public void startWrite() throws InterruptedException {  
    this.lock.lock();  
    try {  
        while (this.busy || this.readerCount != 0)  
            this.ok2write.await();  
        busy = true;  
    } finally { this.lock.unlock(); }  
}
```

Auch hier, MESA-Semantik!

```
public void endWrite() {  
    this.lock.lock();  
    try {  
        this.busy = false;  
        if (this.lock.hasWaiters(this.ok2read))  
            this.ok2read.signal();  
        else  
            this.ok2write.signal();  
    } finally {this.lock.unlock();}  
}
```



# Reader and Writer in Java (MESA)

```
public startRead(){
    if (this.busy || this.queue(this.ok2write))
        this.wait(this.ok2read);
    readerCount++;
    // once one can read they all can read
    this.signal(this.ok2read);
}
```

```
public endWrite(){
    this.busy = false;
    if (this.queue(this.ok2read))
        this.signal(this.ok2read);
    else
        this.signal(this.ok2write);
}
```

## Problem: Fairness

Wartet ein Writer, dann müssen sich neue Reader hinter ihm anstellen.

Müssen Reader während eines Write-Vorgangs warten, erhalten Sie **Vorrang** vor allen anderen wartenden Writern.



# Reader and Writer in Java

Was bei Hoare so einfach war, wird jetzt schwierig!

```
public startRead(){  
    while (this.busy || this.queue(this.ok2write))  
        this.wait(this.ok2read);  
    readerCount++;  
    // once one can read they all can read  
    this.signal(this.ok2read);  
}
```

Dies wäre falsch, denn jetzt haben die Writer Priorität!



# Reader and Writer in Java

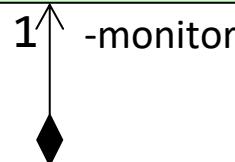
```
void startRead() throws  
InterruptedException {  
    this.lock.lock();  
    try {if (this.busy  
        || this.lock.hasWaiters(this.ok2write))  
        do  
            this.ok2read.await();  
        while (this.busy);  
    this.readerCount++;  
    // once one can read they all can read  
    if (this.lock.hasWaiters(this.ok2read))  
        this.ok2read.signal();  
    } finally { this.lock.unlock();}  
}
```

```
public void endRead() {  
    this.lock.lock();  
    try {this.readerCount--;  
        if (this.readerCount == 0  
            && this.lock.hasWaiters(this.ok2write))  
            this.ok2write.signal();  
    } finally {this.lock.unlock();}  
}
```



# Reader and Writer: ein Gesamtkonzept

ReaderWriterMonitor



Storage

- data: Data

+ read(): Data

+ write(data: Data)

```
class Storage {  
    private Data data;  
    private ReaderWriterMonitor monitor =  
        new ReaderWriterMonitor();  
  
    public Data read() throws InterruptedException {  
        Data data;  
        this.monitor.startRead();  
        //read data;  
        this.monitor.endRead();  
        return data;  
    }  
  
    public void write(Data data) throws InterruptedException {  
        this.monitor.startWrite();  
        // write data  
        this.monitor.endWrite();  
    }  
}
```



# Reader and Writer: ein Gesamtkonzept

```
class Reader implements Runnable {  
    private Storage storage;  
    private volatile Thread thread;  
  
    public Reader(Storage storage) {  
        this.storage = storage;  
        this.thread = new Thread(this);  
        this.thread.start();  
    }  
    ...  
}
```

```
@Override  
public void run() {  
    Thread mySelf = Thread.currentThread();  
    while (mySelf == this.thread) {  
        try {  
            Data data = this.storage.read();  
            // do something with data  
        } catch (InterruptedException e) {}  
    }  
}
```

```
public void stop() {  
    Thread mySelf = this.thread;  
    this.thread = null;  
    if (mySelf != null)  
        mySelf.interrupt();  
}
```



# Reader and Writer: Semaphoren statt CV

ReaderWriterMonitor
<ul style="list-style-type: none"><li>- ok2write: CV</li><li>- ok2read: CV</li><li>- readerCount: int</li><li>- busy: boolean</li></ul> <ul style="list-style-type: none"><li>+ startWrite()</li><li>+ endWrite()</li><li>+ startRead()</li><li>+ endRead()</li></ul> <ul style="list-style-type: none"><li>-wait(cond: CV)</li><li>-signal(cond: CV)</li><li>-queue(cond:CV) : boolean</li></ul>

- Semaphore verfügen über einen Zustand, die Initialisierung definiert die Aufgabe
  - 0 => um auf ein Ereignis zu warten
  - 1 => um als binäres Lock zu fungieren
  - n => um verfügbare Ressourcen zu definieren
- Ein Lock für die Monitorabsicherung (initialisiert mit 1)
  - **lock = new Semaphore(1);**
- Zwei Condition Variables (initialisiert mit 0)
  - **ok2read = new Semaphore(0);**
  - **ok2write = new Semaphore(0);**



# Reader and Writer mit Semaphoren in Java

## ReaderWriterMonitor

- lock: Semaphore
- ok2read: Semaphore
- ok2write: Semaphore
- readerCount: int
- busy: boolean

- + startRead()
- + endRead()
- + startWrite()
- + endWrite()

```
class ReaderWriterMonitor {  
  
    private Semaphore lock = new Semaphore(1);  
    private Semaphore ok2read = new Semaphore(0);  
    private Semaphore ok2write = new Semaphore(0);  
  
    private int readerCount = 0;  
    private boolean busy = false; // nobody is writing  
  
    ...  
}
```



# Reader and Writer mit Semaphoren in Java

```
public void startWrite() throws InterruptedException {  
    this.lock.acquire();  
    try {  
        while (this.busy || this.readerCount != 0) {  
            this.lock.release();  
            this.ok2write.acquire();  
            this.lock.acquire();  
        }  
        busy = true;  
    } finally {this.lock.release();}  
}
```

- P <=> acquire()
- V <=> release()

```
public void endWrite() throws InterruptedException{  
    this.lock.acquire();  
    try {  
        this.busy = false;  
        if (this.ok2read.hasQueuedThreads())  
            this.ok2read.release();  
        else{  
            if (this.ok2write.hasQueuedThreads())  
                this.ok2write.release();  
        }  
    } finally {this.lock.release();}  
}
```



# Reader and Writer mit Semaphoren in Java

```
void startRead() throws  
InterruptedException {  
    this.lock.acquire();  
    try {if (this.busy  
        || this.ok2write.hasQueuedThreads())  
        do {  
            this.lock.release();  
            this.ok2read.acquire();  
            this.lock.acquire();}  
        while (this.busy);  
    this.readerCount++;  
    // once one can read they all can read  
    if (this.ok2read.hasQueuedThreads())  
        this.ok2read.release();  
} finally { this.lock.release();}  
}
```

```
public void endRead() throws InterruptedException {  
    this.lock.acquire();  
    try {this.readerCount--;  
        if (this.readerCount == 0  
            && this.ok2write.hasQueuedThreads())  
            this.ok2write.release();  
    } finally {this.lock.release();}  
}
```



# Concurrency: Prozesse und Threads

**Lösung:** Synchronisation und wechselseitiger Ausschluss aus dem „kritischen Abschnitt“!



**Echt schwierig und verdammt fehleranfällig!**

```
class ReaderWriterMonitor {  
  
    private ReentrantReadWriteLock rwl = new ReentrantReadWriteLock()  
  
    ...  
}
```



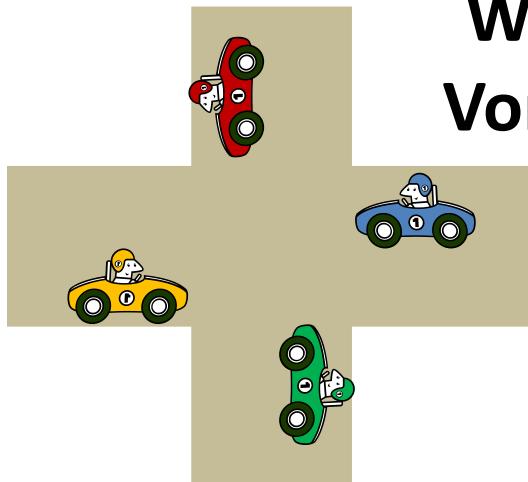
# Probleme

## Deadlocks



# Deadlock (Verklemmung)

Ein Stillstand, der dadurch entsteht, dass jeder darauf wartet, dass der andere zuerst etwas macht!



## Wer hat Vorfahrt?

### § 11 StVO

(3) Auch wer sonst nach den Verkehrsregeln weiterfahren darf oder anderweitig Vorrang hat, muss darauf verzichten, wenn die Verkehrslage es erfordert; auf einen Verzicht darf man nur vertrauen, wenn man sich mit dem oder der Verzichtenden verständigt hat.



# Deadlock: Zirkuläre Verklemmung

```
void* job_A(void *args){  
    ...  
  
    sem_wait(&lockA); ←  
  
    sem_wait(&lockB); ←  
  
    ...  
}
```

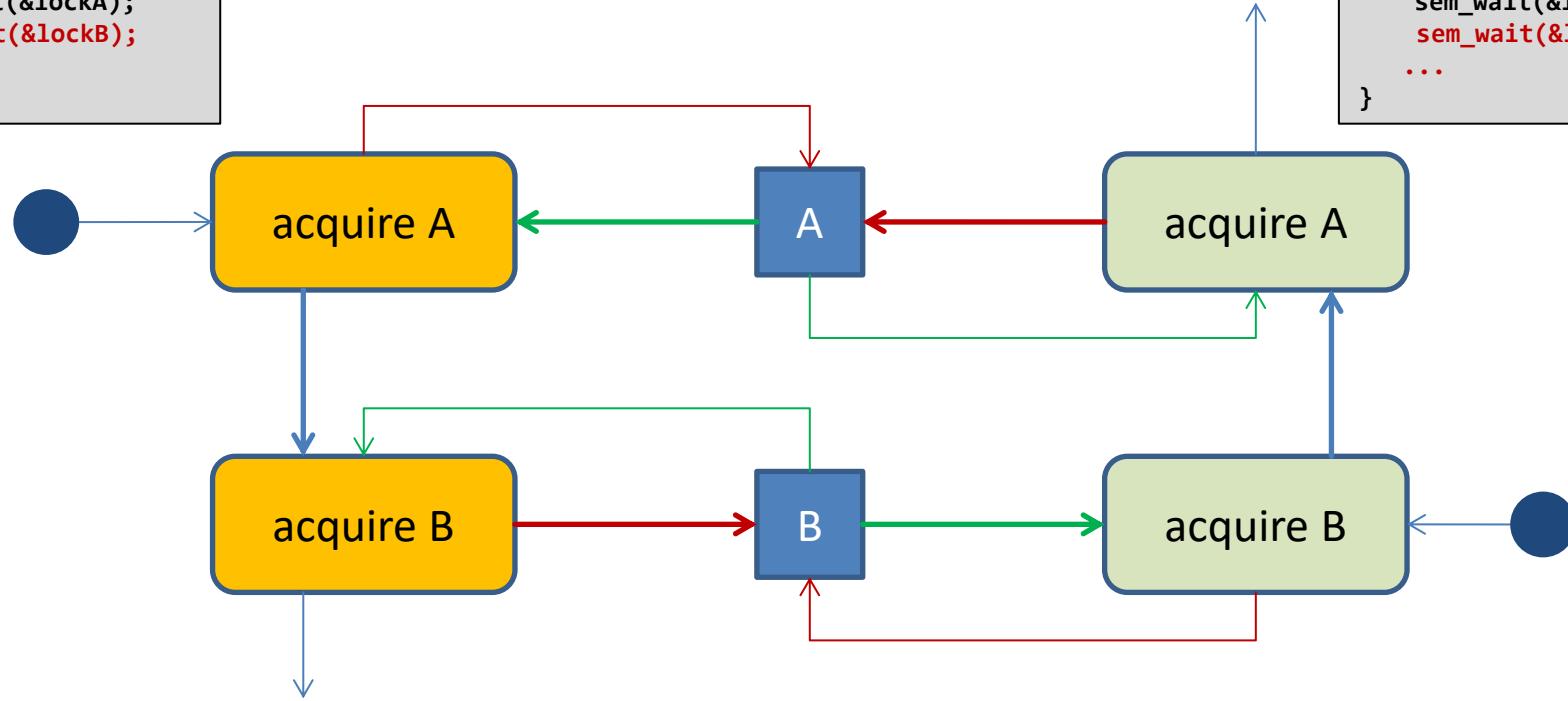
```
void* job_B(void *args){  
    ...  
  
    sem_wait(&lockB); →  
  
    sem_wait(&lockA); →  
  
    ...  
}
```



# Deadlock: Zirkuläre Verklemmung

```
void* job_A(void *args){  
    sem_wait(&lockA);  
    sem_wait(&lockB);  
    ...  
}
```

```
void* job_B(void *args){  
    sem_wait(&lockB);  
    sem_wait(&lockA);  
    ...  
}
```





# Deadlock: Zirkuläre Verklemmung

## Lösung

```
void* job_A(void *args){  
    ...  
  
    sem_wait(&lockA); ←  
  
    sem_wait(&lockB); ←  
  
    ...  
}
```

```
void* job_B(void *args){  
    ...  
  
    sem_wait(&lockA); →  
  
    sem_wait(&lockB); →  
  
    ...  
}
```

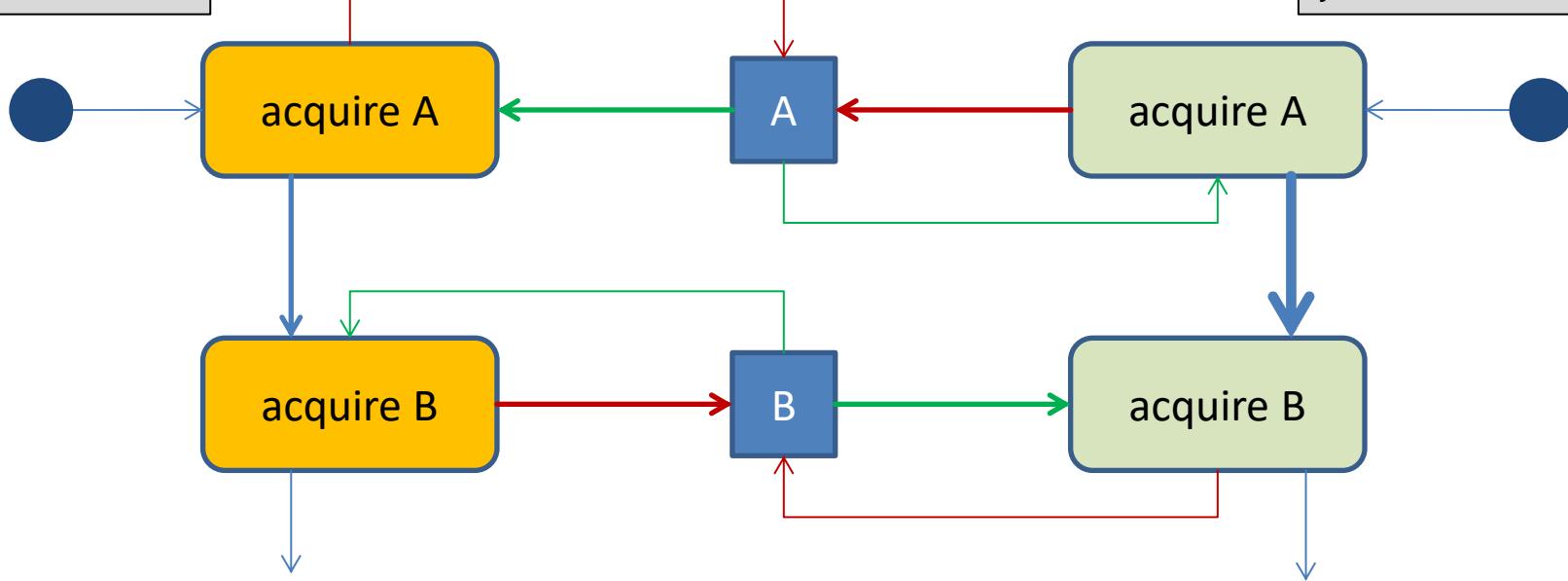


# Deadlock: Zirkuläre Verklemmung

```
void* job_A(void *args){  
    sem_wait(&lockA);  
    sem_wait(&lockB);  
    ...  
}
```

Lösung

```
void* job_B(void *args){  
    sem_wait(&lockA);  
    sem_wait(&lockB);  
    ...  
}
```





# Deadlock: Zirkuläre Verklemmung

**Problem: Welches ist die richtige Reihenfolge?**

```
void* job_A(void *args){  
    ...  
    sem_wait(&lockA);  
    sem_wait(&lockB);  
    ...  
}
```

```
void* job_B(void *args){  
    ...  
    sem_wait(&lockA);  
    sem_wait(&lockB);  
    ...  
}
```

```
void* job_A(void *args){  
    ...  
    sem_wait(&lockB);  
    sem_wait(&lockA);  
    ...  
}
```

```
void* job_B(void *args){  
    ...  
    sem_wait(&lockB);  
    sem_wait(&lockA);  
    ...  
}
```

Die Reihenfolge ist egal,  
sie muss nur  
eingehalten werden!



**sortieren**  
(auch Adressen sind vergleichbar)



# Deadlock: Alternative Lösungen

- Den Erwerb von Locks absichern.

```
void* job_A(void *args){  
    ...  
    sem_wait(&lock_lock);  
    sem_wait(&lockA);  
    sem_wait(&lockB);  
    sem_post(&lock_lock)  
    ...}
```

- Dafür sorgen, dass stets alle Locks oder keine Locks erworben werden!

Achtung!  
Livelock Gefahr

```
void* job_A(void *args){  
    while (1){  
        sem_wait(&lockA);  
        if (sem_trywait(&lockB) == 0)  
            break;  
        ...  
    }  
    ...  
}
```



# Deadlock:

- Sicher oder gefährlich?

```
int greater_than(vector_t *v1, vector_t *v2){  
    int res = 0;  
    pthread_mutex_lock(v1->lock);  
    pthread_mutex_lock(v2->lock);  
    res = ((v1->x*v1->x) + (v1->y*v1->y))  
        > ((v2->x*v2->x) + (v2->y*v2->y));  
    pthread_mutex_unlock(v2->lock);  
    pthread_mutex_unlock(v1->lock);  
    return res;  
}
```

```
typedef struct vector_t {  
    pthread_mutex_t *lock;  
    int x;  
    int y;  
} elem_t;
```

```
vector_t a;  
vector_t b;  
...
```

```
if (greater_than(&b,&a))
```

```
if (greater_than(&a,&b))
```



# Concurrency

- Concurrency zu beherrschen ist schwierig.
- Mehr und dedizierte Locks sind oftmals besser zu verstehen als wenige geniale.
- Locks zu ordnen ist meist eine gute Idee.
- Monitore konzentrieren Probleme auf überschaubare Codeabschnitte.