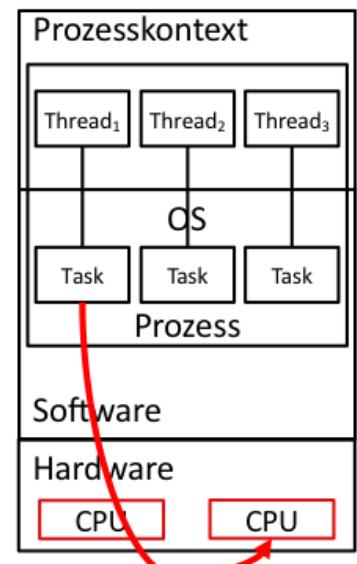


- Verwaltung der Prozesse in sortiertem Baum (Red Black-Tree) Elemente links im Baum haben geringere vruntime.
- Aufwand für Einfügen in Baum $O(\log(n))$
- Wähle stets das Element, welches ganz links im Baum steht (Zeiger auf dieses Element verringert Aufwand beim finden des Elements auf $O(1)$)

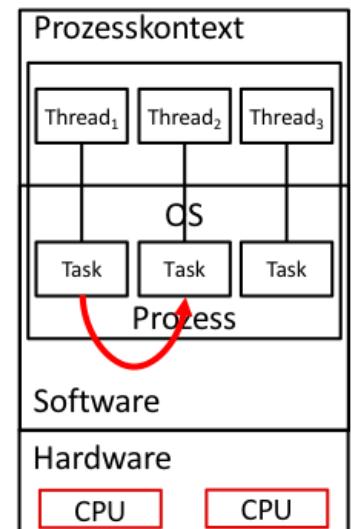


- Die Migration von einer CPU auf andere ist teuer, mittels
`/proc/sys/kernel/sched_migration_cost`
 kann die Migration erschwert / verhindert werden.

Linux Kontextwechsel

- Ist die vruntime eines Tasks abgelaufen, wird ein neuer Prozess ausgewählt: es wird der linkeste Task (höchste Priorität) aus dem RB-Baum selektiert:

- » Der `scheduler()` ruft `context_switch()` in (`kernel/sched/core.c`)
- » Ruft `switch_mm()` (architekturabhängig), um Virtual Mapping auf neuen Prozess zu wechseln
- » Und `switch_to()` (architekturabhängig), um den HW-Kontext zu wechseln – speichern aller Register (CPU, FPU), Flags, Stack-pointer, Segmente, etc.



- » `CreateThread (. . .)`
- » `ExitThread (. . .)`
- » `TerminateThread (. . .)`

- Mit der Entwicklung von Threads musste ein großer Teil von Software untersucht und neu geschrieben werden!!!
- Ein Beispiel `errno`: global definierte `int`-Variable gibt Fehler von Posix Fkt. wider, z.B. Fehlerwert `ENOMEM`.
- Kommentar aus glibc' /usr/include/errno.h:

```
/* Declare the `errno' variable, unless it's defined as a macro by bits/errno.h. This is the case in
GNU, where it is a per-thread variable...*/
```

```
#ifndef errno
extern int errno;
#endif
```

```
/* und in /usr/include/i386-linux-gnu/bits/errno.h */
extern int * __errno_location(void) __attribute__((__const__));
#if defined __LIBC_REENTRANT
    /* When using threads, errno is a per-thread value.  */
#define errno (*__errno_location ())
#endif
```

- Für jeden möglichen Codepfad ist sicherzustellen, daß keine gemeinsame Ressourcen von mehreren Threads verändert werden. Ein Beispiel:

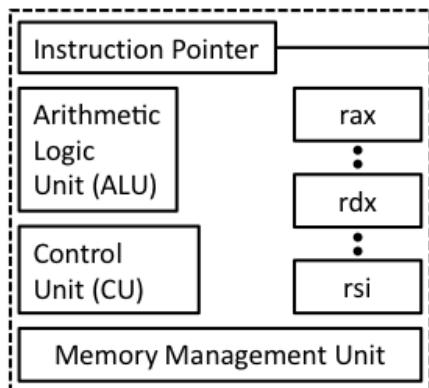
```
konto_update(int nr, int betrag) {
    kontos[nr] += betrag;
}
```

The diagram shows the assembly code for the `konto_update` function:

```
movslq %edi, %rax
leaq _kontos(%rip), %rcx
addl %esi, (%rcx,%rax,4)
ret
```

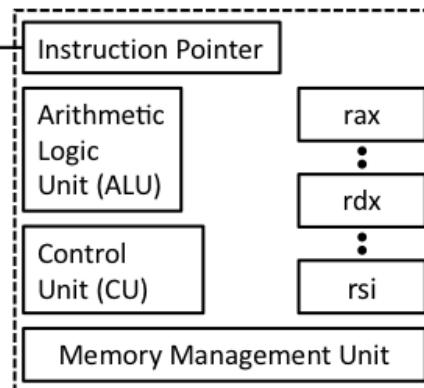
Two arrows point from the `kontos[nr]` access in the assembly code to the `kontos` array in the memory models of Thread 1 and Thread 2.

CPU von Thread 1:



Zugriff auf `kontos` ist nicht geschützt, (keine Locks, nicht atomar).

CPU von Thread 2:



Übersetzen Sie folgenden Code und führen ihn aus:

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

int z = 0;

/* this function is run by the second thread */
void *worker(void *arg_ptr)
{
    /* increment x to 100 */
    int* x_ptr = (int *) arg_ptr;
    for(int i = 1; i < 100; i++){
        printf("THREAD: x: %d\n", *x_ptr);
        *x_ptr = *x_ptr +1;
        z++;
    }
    printf("THREAD: x increment finished\n");

    /* the function must return something - NULL will do */
    return NULL;
}

} ...
```

Posix Threads in C ausprobieren (2)

```
... int main() {
    int x = 0, y = 0;

    /* this variable is our reference to the second thread */
    pthread_t worker_thread;

    /* create a second thread which executes worker*/
    if(pthread_create(&worker_thread, NULL, worker, &x)) {
        fprintf(stderr, "Error creating thread\n");
        return 1;
    }

    /* increment y to 100 in the first thread */
    for(int i = 0; i < 100; i++){
        y++;
        z++;
        printf("MAIN: x: %d, y: %d z: %d\n", x, y, z);
    }
    printf("MAIN: y increment finished\n");
    /* wait for the second thread to finish */
    if(pthread_join(worker_thread, NULL)) {
        fprintf(stderr, "Error joining thread\n");
        return 2;
    }
    printf("x: %d, y: %d, z: %d\n", x, y, z);
    return 0;
}
```

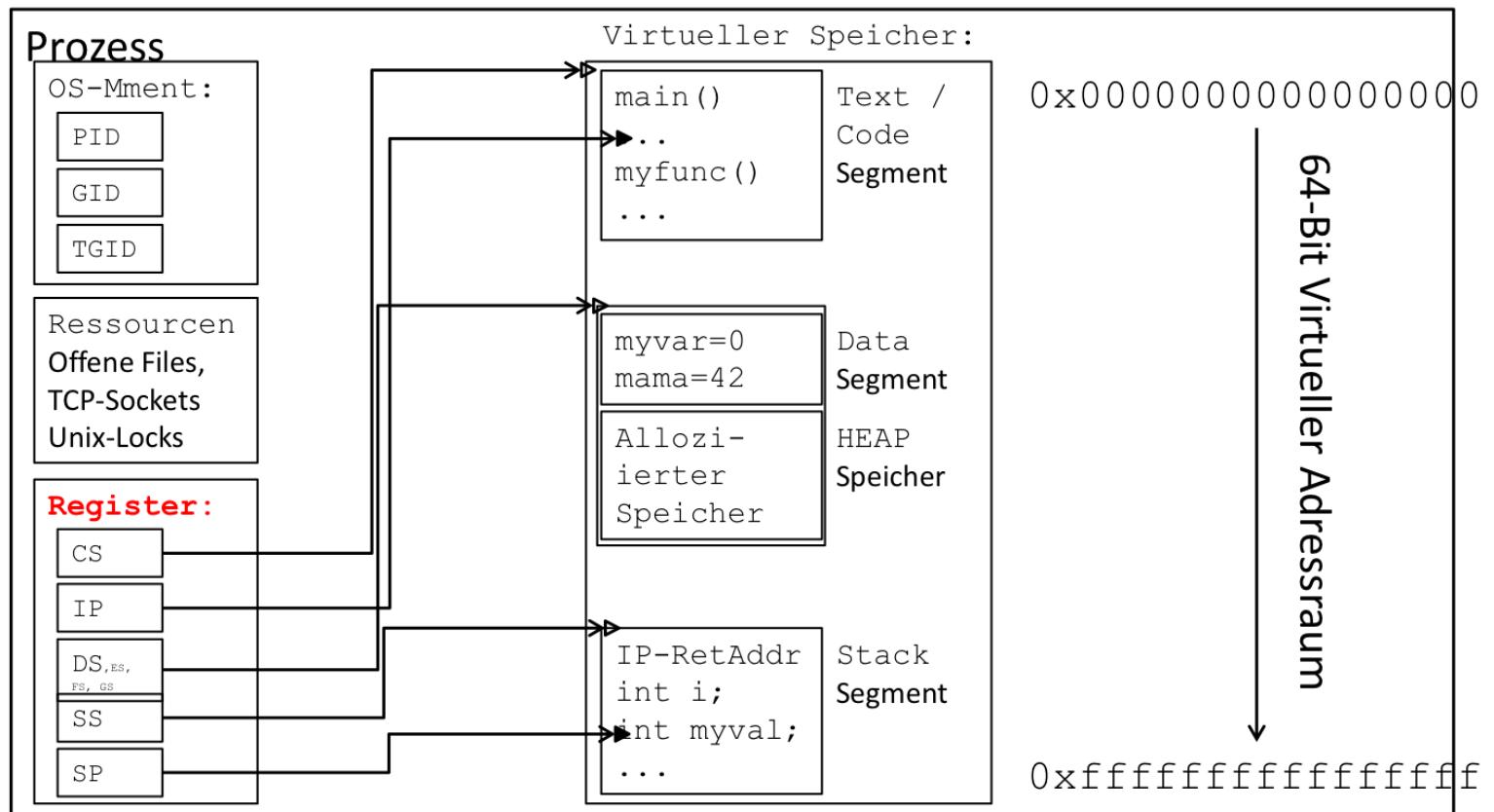
- Mutex
 - Mutual exclusion
 - Code wird gesperrt sobald ein Thread den mit `pthread_mutex_lock(&mutexname);` markieren Codeteil betritt
 - Threads die `mutex_lock` aufrufen während ein anderer Thread den Lock gesetzt hat werden blockiert
 - Der Lock wird mit `pthread_mutex_unlock(&konto_mutex);` wieder freigegeben. Wartende Threads werden nicht mehr blockiert (bis der nächste Prozess den Lock aktiviert).
- Weitere Synchronisationsmöglichkeiten
 - Semaphore
 - Condition Variablen
 - → Kommen später!

Lösung: Thread-Safety

Lösung: Exklusiver Zugriff mit Mutex

```
int kontos[2]={0, 0};  
int aktuelles_konto = 0;  
pthread_mutex_t konto_mutex= PTHREAD_MUTEX_INITIALIZER;  
  
void konto_update(int nr, int betrag) {  
  
    pthread_mutex_lock( &konto_mutex );  
    aktuelles_konto = nr;  
    kontos[aktuelles_konto] += betrag;  
    pthread_mutex_unlock( &konto_mutex );  
  
}
```

Speicherverwaltung



Jeder Speicherzugriff muss schnell sein!

Temporale Lokalität

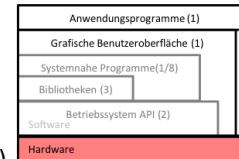
Speicherzugriffe tendieren dazu, in kurzer Zeit wieder auf die gleiche oder ähnliche Speicherbereiche zuzugreifen

Spatiale Lokalität:

Speicherzugriffe tendieren dazu, räumlich wieder auf gleiche oder ähnliche Speicherbereiche zuzugreifen

→ Essenziell für Caches

- Intel i386 Prozessor Mechanismen:
 - » Erster 32 Bit Prozessor (bis zu 4GB Speicher)
 - » 4 Ebenen mit (Speicher)schutzmechanismen, wir wissen:
 - » Level 0 darf alles/jede Instruktion ausführen (Betriebssystem)
 - » Level 3 darf am wenigsten (Applikationsebene)
(Linux verwendet nur diese 2 Ebenen)
 - » Virtueller Speicher: nicht der gesamte 4 GB Adressraum muss physikalisch vorhanden sein:
 - » Auslagern von Speicher auf Festplatte (Paging/Swapping)
 - » Damit Segmentierung des Speicherbereichs

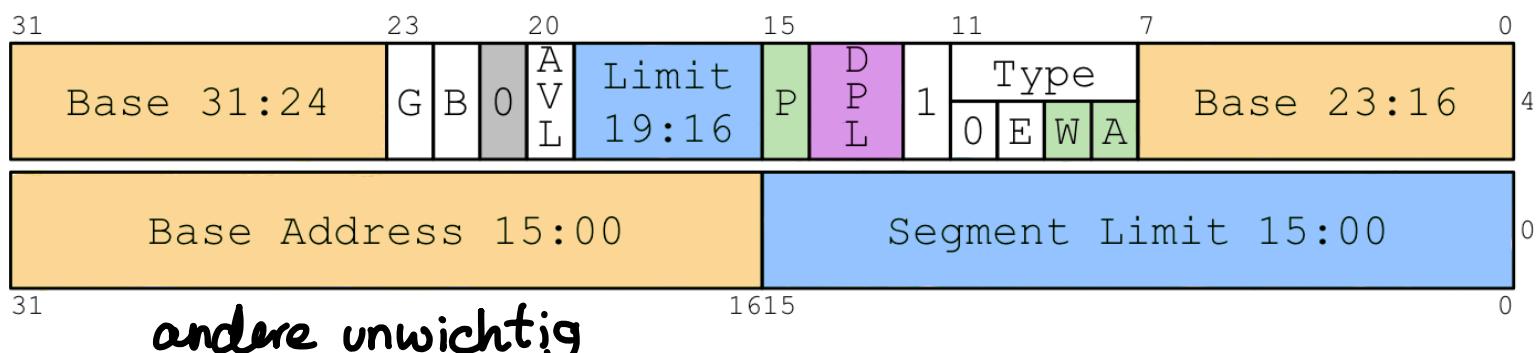


Physischer Speicher wird in 4KB Pages unterteilt

→ Jede Page hat einen Deskriptor, den der Prozessor benutzt, um den Zugriff zu kontrollieren!

→ Oder eine Exception auszulösen den das OS auffängt!

→ Data Segment Descriptor (32 Bit)



Base Adress: 32 Bit Start-Adresse des Segments

Segment Limit: 20-Bit Länge in 4KB Pages

DPL: 2-Bit Descriptor Privilege Level

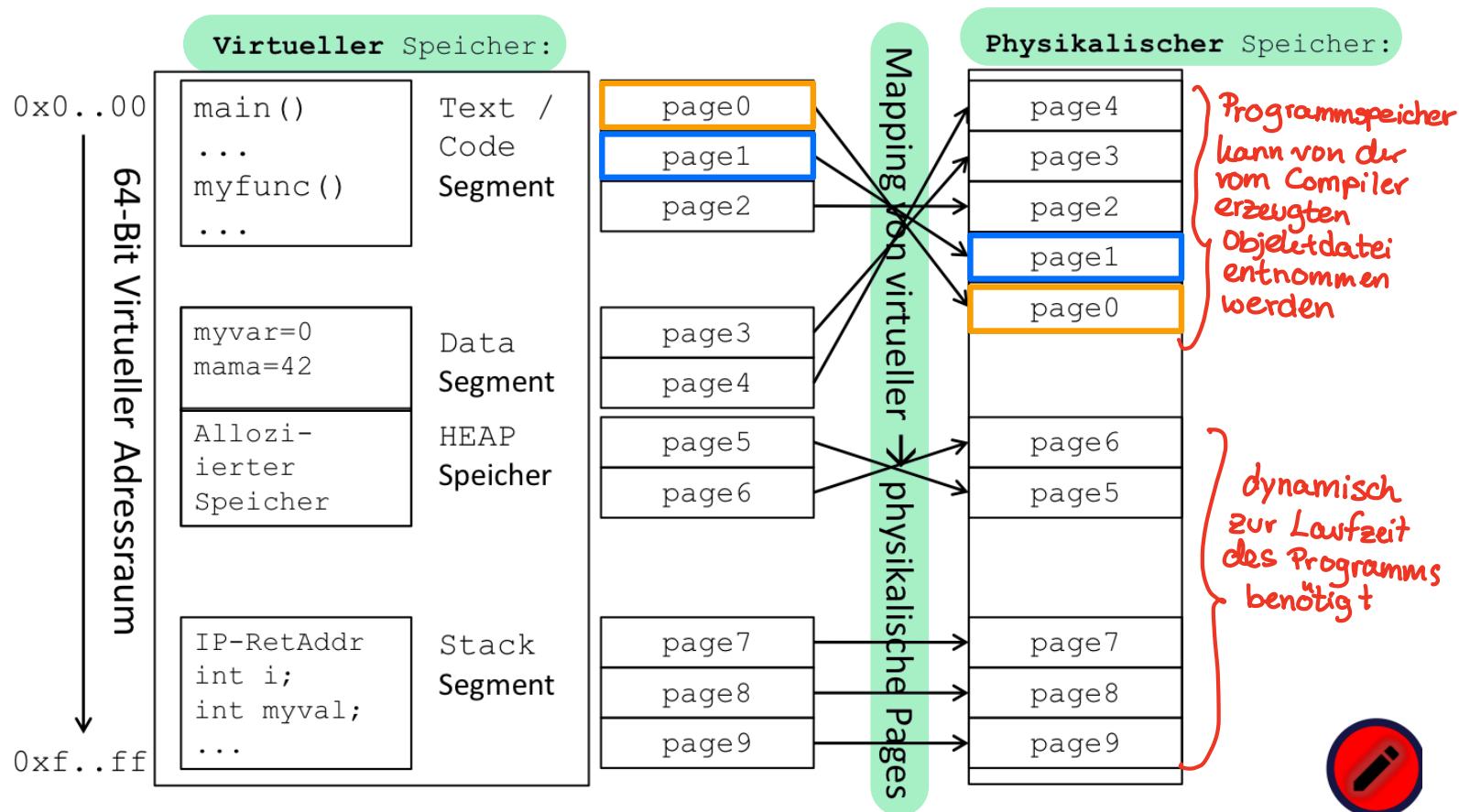
→ LVL.0 → OS , LVL.3 User Space

W: Writable Bit = Speichersegment ist beschreibbar

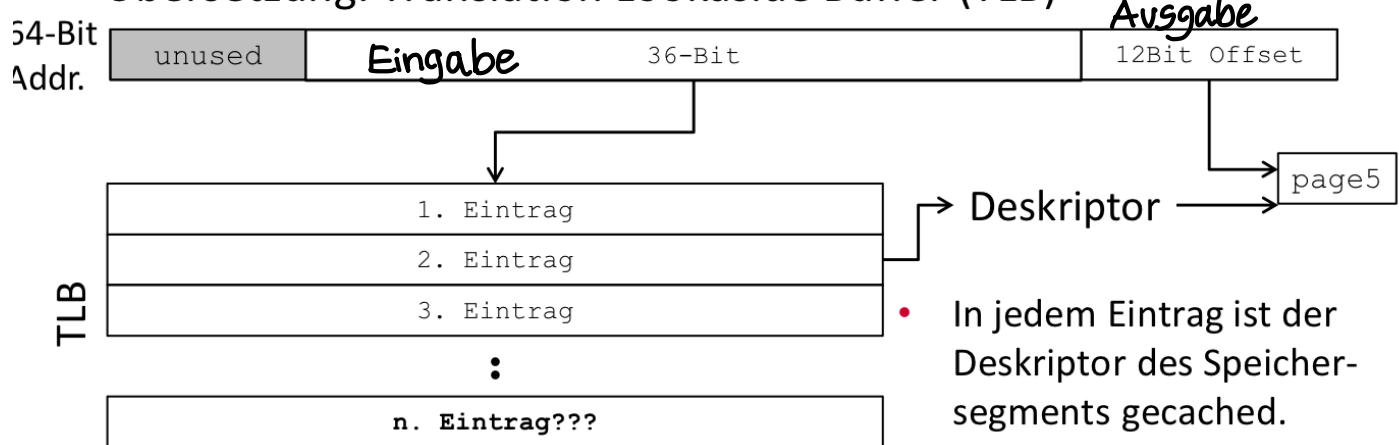
A: Access Bit = Speichersegment wurde angefasst

P: Present Bit = Segment ist im Hauptspeicher

Ein 64-Bit Adressraum ist heute (noch nicht) notwendig (Von den 64-Bit führt die CPU „nur“ 48-Bit Adressleitungen nach außen...
→ 32 Bit auf 33 Bit wäre Verdoppelung des Speichers!



- Diese Umwandlung der 64-Bit virtuellen auf die 64-Bit physikalische Adresse macht der Prozessor...
- Jeder** Speicherzugriff wird so überprüft!!!
- Damit der Zugriff schnell erfolgt, gibt es einen Cache zur Übersetzung: Translation Lookaside Buffer (TLB)



- Das OS muss effizient verwalten, in Bezug auf:

- » Speicher (möglichst wenig Speicher für Verwaltungsdaten)
- » Zeit (möglichst wenig Instruktionen zur Verwaltung, wenige Speicherverweise auf nicht-gecachte Daten!)
- » Abbildung von VM Speicherseiten auf Auslagerungsdatei/Swap-Partition etc. auf der Festplatte
(Eventuell müssen auch Speicherseite der Verwaltungsdaten auf Festplatte auslagerbar sein?!)

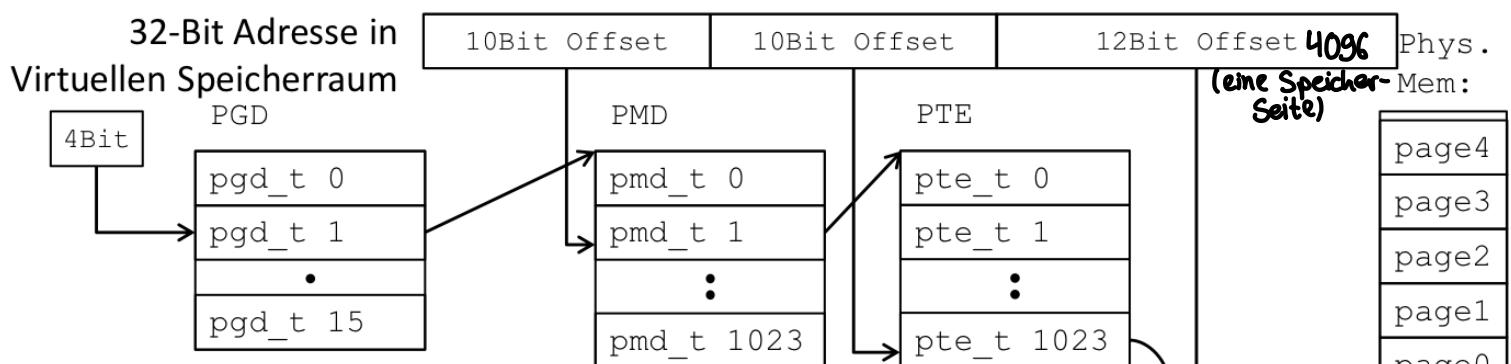
- Also: Effiziente Umwandlung von Virt. → Phys. Address

Für jeden TLB Miss muss das OS die Pages einen Prozess durchsuchen (Page-Walk) TLB = Translation Lookaside Buffer

Virtueller Speicher @ 32 Bit: 4kB Pages



Hochschule Esslingen
University of Applied Sciences



Die 32-Bit Adresse wird aufgespalten in einzelne Teile: als Index in je 4kB Pages mit Zeigern auf PTE/page:

PGD = Page Global Directory

PMD = Page Middle Directory

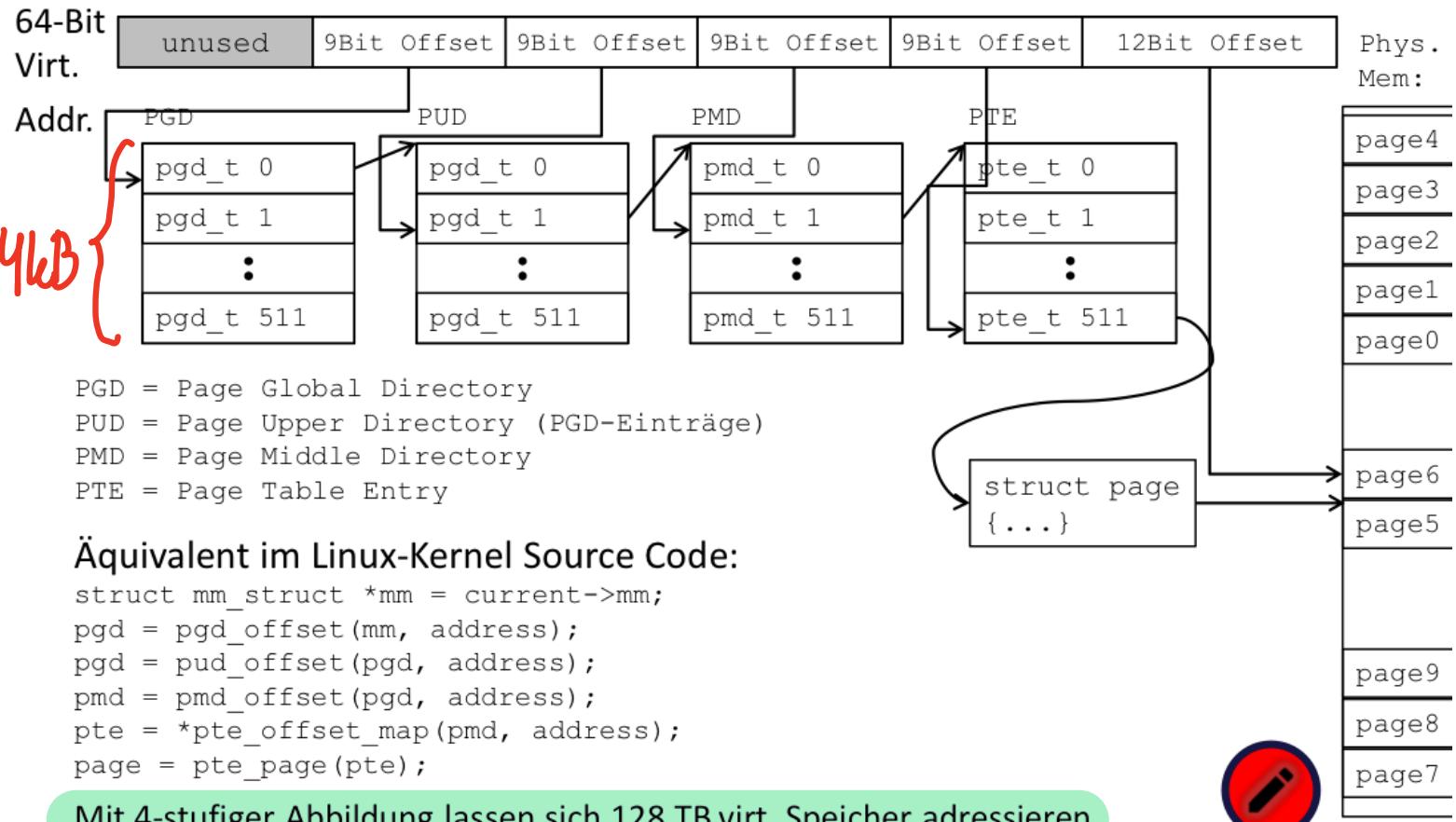
PTE = Page Table Entry

Im Kernel (current=task_struct des ausgeführten Tasks):

```
struct mm_struct *mm = current->mm;
pgd = pgd_offset(mm, address);
pmd = pmd_offset(pgd, address);
pte = *pte_offset_map(pmd, address);
page = pte_page(pte);
```

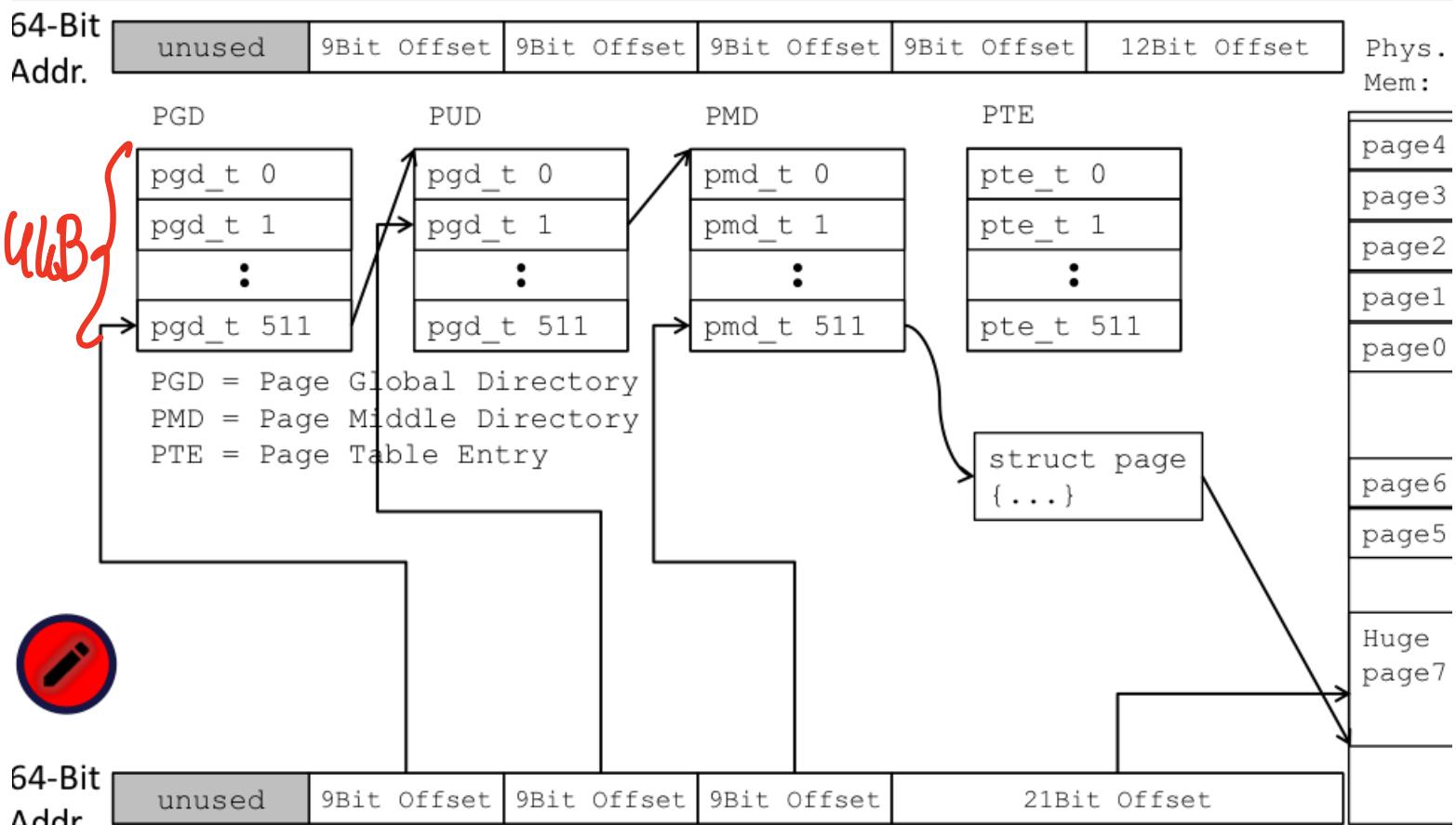
Preisfrage: wieviele Pages kann man mit diesem Konzept ansprechen?

Und wieviel Speicher?

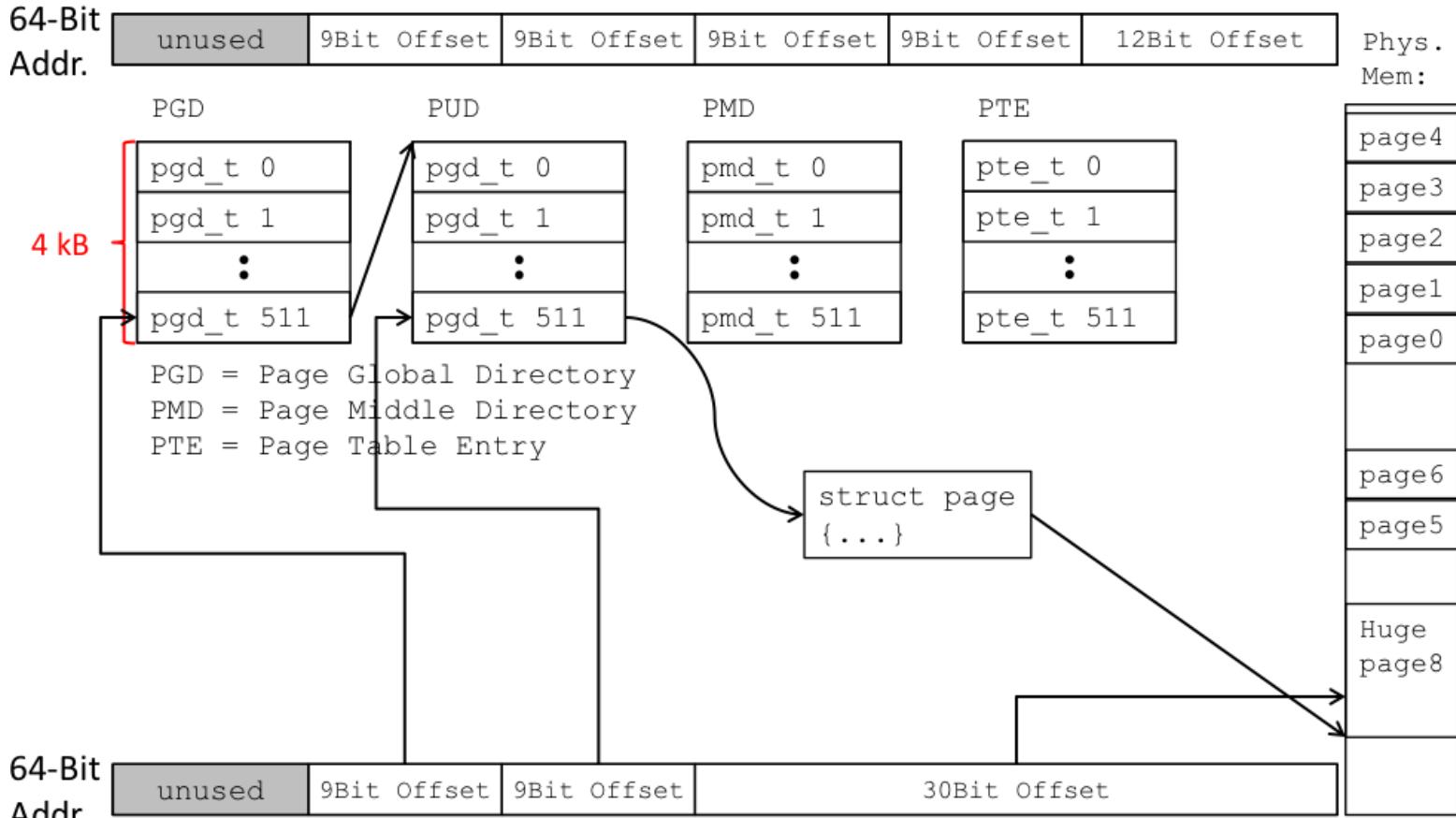


Mit 4-stufiger Abbildung lassen sich 128 TB virt. Speicher adressieren

Virtueller Speicher @ 64 Bit: 2MB Huge Pages



Virtueller Speicher @ 64 Bit: 1GB Huge Pages



- Ein Prozess hat Listen der ihm zugeordneten Pages. Wird ein PageFault ausgelöst, kann Linux rausfinden, ob der vom Prozess adressierte Speicher wirklich alloziert war, ob die Page vielleicht ausgelagert war...

- 2^n aufeinanderfolgende, freie Pages werden zusammengefasst
- Das erlaubt sehr einfache Verwaltung von freien Pages:



Physikalische Speicherseiten:

page15
page14
page13
page12
page11
page10
page9
page8
page7
page6
page5
page4
page3
4 kB
page2
4 kB
page1
4 kB
page0
4 kB

Buddy 1. Ordnung

buddy ₁ 7
buddy ₁ 6
buddy ₁ 5
buddy ₁ 4
buddy ₁ 3
buddy ₁ 2
buddy ₁ 1
buddy ₁ 0

8 kB

Buddy 2. Ordnung

buddy ₂ 3
buddy ₂ 2
buddy ₂ 1
buddy ₂ 0

16 kB

Buddy 3. Ordnung

buddy ₃ 1
buddy ₃ 0

32 kB

Frei

Prozess A : 32

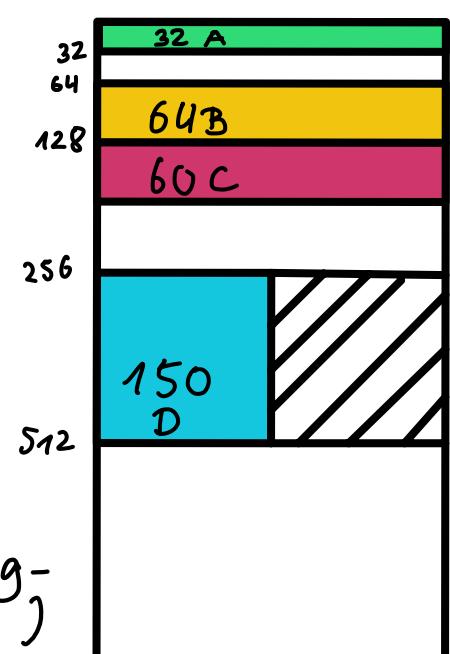
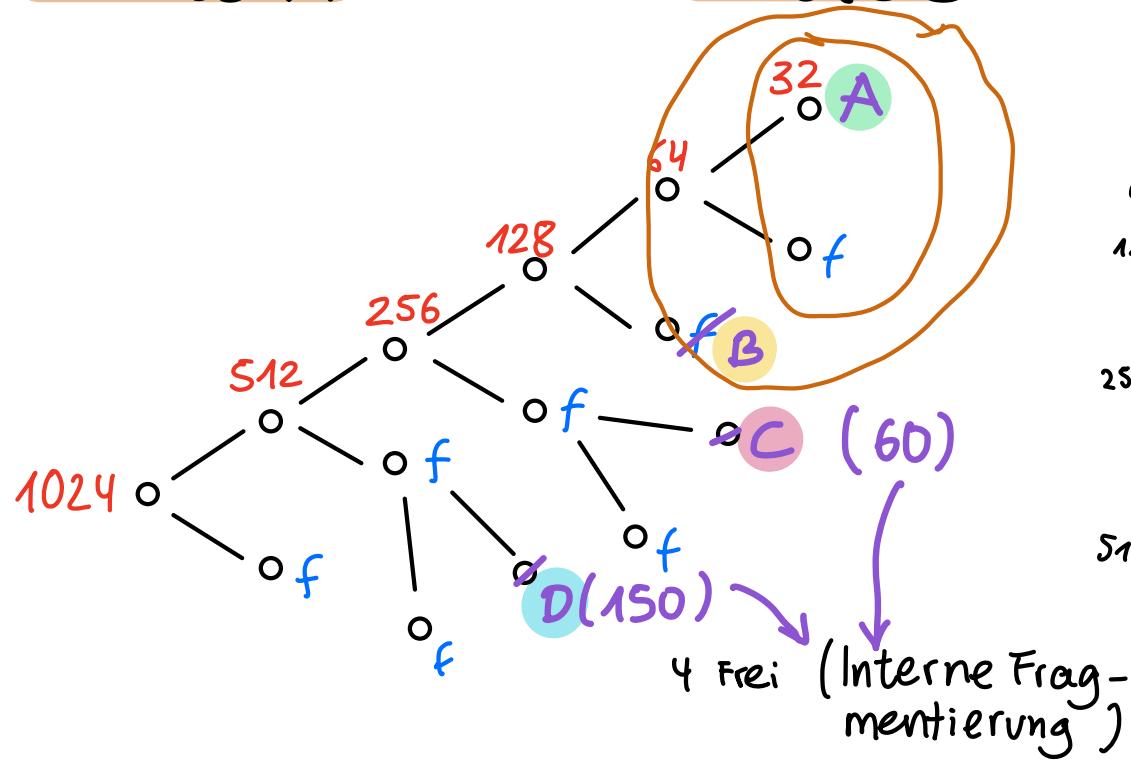
Prozess C : 60

Release A

Prozess B: 64

Prozess D: 150

Release B



Algorithmen um Pages auszulagern

* Optimaler Algorithmus zur Seitenersetzung
nicht realisierbar!

* Not-Recently-Used Algorithmus (NRU)

- Access Bit wird bei Speicherzugriff auf 1 gesetzt
- Access Bit muss periodisch gelöscht werden (Zeiteinheit)
- Speicherseite mit Access Bit = 0 wird auf die Festplatte ausgelagert um anderen Speicherseiten Platz zu schaffen

Sehr grobe Annäherung an den LRU

Nicht gut, da auch genutzte Seiten ausgelagert werden

* First-In-First-Out Algorithmus (FIFO)

- Erster Eintrag der Liste wird ausgelagert
- Entfernt evtl. auch wichtige Seiten

* Second Chance Algorithmus

- Gleiches Verfahren wie bei FIFO, mit einem Access Bit
- Seite wird nicht entfernt bei Access Bit = 1
- Seite wird ans Ende der Warteschlange gesetzt
- Access Bit wird auf 0 gesetzt

Enorme Verbesserung zum FIFO Algorithmus

*Clock Algorithmus

- Gleiches Verfahren wie beim Second Chance Algorithmus
- Kreisförmiger Ablauf (Uhr)

Realistische Annahmen über veraltete Pages

* Least-Recently-Used Algorithmus (LRU)

- Benutzte Speicherseiten werden nach hinten gestellt
- ältere Seiten rücken vor

Teure Implementierung

Liste wird oft umsortiert

- Während Sie den Firefox Browser öffnen und wieder schließen, überprüfen Sie den Speicherverbrauch mit:
- top:

```
hpcraink@ubuntu: ~
top - 13:19:59 up 2:20, 3 users, load average: 0.00, 0.01, 0.05
Tasks: 155 total, 1 running, 154 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.3 us, 0.3 sy, 0.0 ni, 99.3 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem: 1026616 total, 939508 used, 87108 free, 135876 buffers
KiB Swap: 1046524 total, 4240 used, 1042284 free, 374352 cached

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
1023 mysql 20 0 319m 33m 5784 S 0.3 3.3 0:04.47 mysqld
2265 hpcraink 20 0 259m 76m 32m S 0.3 7.7 1:38.94 compiz
2663 hpcraink 20 0 97540 17m 12m S 0.3 1.8 0:09.29 gnome-terminal
  1 root 20 0 3628 1928 1284 S 0.0 0.2 0:00.92 init
  2 root 20 0 0 0 0 S 0.0 0.0 0:00.00 kthreadd
  3 root 20 0 0 0 0 S 0.0 0.0 0:00.46 ksoftirqd/0
  6 root rt 0 0 0 0 S 0.0 0.0 0:00.00 migration/0
  7 root rt 0 0 0 0 S 0.0 0.0 0:00.83 watchdog/0
  8 root 0 -20 0 0 0 S 0.0 0.0 0:00.00 cpuset
QtOctave 0 -20 0 0 0 S 0.0 0.0 0:00.00 khelper
  10 root 20 0 0 0 0 S 0.0 0.0 0:00.00 kdevtmpfs
  11 root 0 -20 0 0 0 S 0.0 0.0 0:00.00 netns
  12 root 20 0 0 0 0 S 0.0 0.0 0:00.04 sync_supers
  13 root 20 0 0 0 0 S 0.0 0.0 0:00.00 bdi-default
  14 root 0 -20 0 0 0 S 0.0 0.0 0:00.00 kintegrityd
  15 root 0 -20 0 0 0 S 0.0 0.0 0:00.00 kblockd
  16 root 0 -20 0 0 0 S 0.0 0.0 0:00.00 ata_sff
```

- Während Sie den Firefox Browser öffnen und wieder schließen, überprüfen Sie den Speicherverbrauch mit:
- vmstat fasst die Werte für einen Zeitraum zusammen:

```

hpcraink@ubuntu: ~
4 0 6716 128624 51192 372208 0 0 0 0 130 772 6 4 90 0
refox Web Browser
hpcraink@ubuntu:~$ vmstat 1
procs -----memory----- --swap-- -----io---- -system-- ----cpu-----
r b swpd free buff cache si so bi bo in cs us sy id wa
2 0 6716 173092 51248 372080 0 1 86 22 86 328 1 1 98 0
0 0 6716 173000 51248 372080 0 0 0 0 129 269 1 2 97 0
2 0 6716 143456 51288 372192 0 0 16 292 300 4038 33 19 48 0
2 0 6716 126264 51296 372224 0 0 0 88 391 3981 51 21 28 0
0 0 6716 125644 51296 372224 0 0 0 0 245 1003 22 14 64 0
1 0 6716 102704 51304 372224 0 0 0 80 326 380 66 27 7 0
0 0 6716 122048 51304 372224 0 0 0 8 145 188 40 7 53 0
0 0 6716 121924 51304 372224 0 0 0 0 112 193 1 1 98 0
0 0 6716 127380 51304 372224 0 0 0 0 99 210 3 1 96 0
0 0 6716 126752 51304 372228 0 0 0 0 58 182 1 1 98 0
0 0 6716 126884 51312 372216 0 0 0 64 54 199 1 0 99 0
0 0 6716 126884 51312 372224 0 0 0 0 42 132 1 1 98 0
0 0 6716 126884 51312 372224 0 0 0 0 53 167 2 0 98 0
0 0 6716 126876 51312 372224 0 0 0 0 52 194 0 1 99 0
0 0 6716 126628 51312 372224 0 0 0 0 45 138 0 0 100 0
1 0 6716 126628 51312 372224 0 0 0 0 174 1517 8 5 87 0
0 0 6716 128140 51320 372224 0 0 0 20 321 469 4 2 94 0
0 0 6716 128140 51320 372224 0 0 0 0 152 256 2 1 97 0

```

- Während Sie den Firefox Browser öffnen und wieder schließen, überprüfen Sie den Speicherverbrauch mit:
- Mit pmap lässt sich /proc/PID/maps interpretieren:

```

hpcraink@ubuntu: ~
refox Web Browser
hpcraink@ubuntu:~$ pmap 6716
b7732000    4K r---- /lib/i386-linux-gnu/libdl-2.15.so
b7734000    4K r--s- /run/user/hpcraink/dconf/user
b7735000    4K r---- /usr/lib/locale/locale-archive
b7736000    4K r-x-- /usr/lib/i386-linux-gnu/libgthread-2.0.so.0.3400.0
b7737000    4K r---- /usr/lib/i386-linux-gnu/libgthread-2.0.so.0.3400.0
b7738000    4K rw--- /usr/lib/i386-linux-gnu/libgthread-2.0.so.0.3400.0
b7739000   32K r-x-- /usr/lib/i386-linux-gnu/libXrender.so.1.3.0
b7741000    4K r---- /usr/lib/i386-linux-gnu/libXrender.so.1.3.0
b7742000    4K rw--- /usr/lib/i386-linux-gnu/libXrender.so.1.3.0
b7743000    8K r-x-- /usr/lib/firefox/libmozalloc.so
b7745000    4K r---- /usr/lib/firefox/libmozalloc.so
b7746000    4K rw--- /usr/lib/firefox/libmozalloc.so
b7747000    8K rw--- [ anon ]
b7749000    4K r-x-- [ anon ]
b774a000  128K r-x-- /lib/i386-linux-gnu/ld-2.15.so
b776a000    4K r---- /lib/i386-linux-gnu/ld-2.15.so
b776b000    4K rw--- /lib/i386-linux-gnu/ld-2.15.so
b776c000    72K r-x-- /usr/lib/firefox/firefox
b777e000    4K r---- /usr/lib/firefox/firefox
b777f000    4K rw--- /usr/lib/firefox/firefox
bff4e000  136K rw--- [ stack ]
total 374052K
(END)

```

Speicherüberwachung:

- * cat /proc/meminfo
- * cat /proc/\$\$/status
- * cat /proc/\$\$/smaps

Jeder Prozess hat Soft- und Hardlimits:

- * ulimit -a
- * cat /proc/\$\$/limits

Speicherverbrauch

- Was macht das folgende Programm? Variieren Sie die Größen und beobachten die Ausgabe! (Achtung gekürzt)

```
#define SIZE_START 4096
#define SIZE_DELTA 4096
#define SLEEP_MS    (1000*1000)
#define MEMSET_CHAR '\0'

int main (int argc, char * argv[]) {
    char * p;
    size_t size = SIZE_START;
    p = malloc (size);
    while (NULL != p) {
        usleep (SLEEP_MS);
        memset (p, MEMSET_CHAR, size);
        size += SIZE_DELTA;
        p = realloc (p, size);
    }
}
```

Bitte aus Moodle
herunterladen!

Dateisysteme:

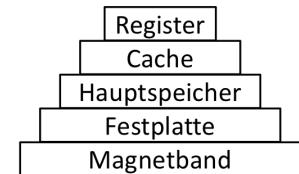
- mount → Dateisystem wird an Datei-Baum eingehängt

```
hpcraink@ubuntu:/usr/src/linux-3.11.5/fs$ mount
/dev/sda1 on / type ext4 (rw,errors=remount-ro) ← Das Wurzelverzeichnis "Root" liegt auf der ersten Partition der ersten Platte!
proc on /proc type proc (rw,noexec,nosuid,nodev) ← Hier als EXT4 formatiert, das virtuelle Dateisystem proc erlaubt Zugriff auf Infos des laufenden Kernels
sysfs on /sys type sysfs (rw,noexec,nosuid,nodev)
none on /sys/fs/cgroup type tmpfs (rw)
none on /sys/fs/fuse/connections type fusectl (rw)
none on /sys/kernel/debug type debugfs (rw)
none on /sys/kernel/security type securityfs (rw)
udev on /dev type devtmpfs (rw,mode=0755)
devpts on /dev/pts type devpts (rw,noexec,nosuid,gid=5,mode=0620)
tmpfs on /run type tmpfs (rw,noexec,nosuid,size=10%,mode=0755)
none on /run/lock type tmpfs (rw,noexec,nosuid,nodev,size=5242880)
none on /run/shm type tmpfs (rw,nosuid,nodev)
none on /run/user type tmpfs (rw,noexec,nosuid,nodev,size=104857600,mode=0755)
none on /sys/fs/pstore type pstore (rw)
binfmt_misc on /proc/sys/fs/binfmt_misc type binfmt_misc (rw,noexec,nosuid,nodev)
systemd on /sys/fs/cgroup/systemd type cgroup (rw,noexec,nosuid,nodev,none,name=systemd)
gvfsd-fuse on /run/user/1000/gvfs type fuse.gvfsd-fuse (rw,nosuid,nodev,user=hpcraink)
:host:/ on /mnt/hgfs type vmhgfs (rw,ttl=1)
vmware-vmblock on /run/vmblock-fuse type fuse.vmware-vmblock (rw,nosuid,nodev,default_permissions,allow_other)
```

Das vmhgfs erlaubt Zugriff der virtuellen Maschine auf gesharete Verzeichnisse des Hosts ↑

Wiederholung Speicherhierarchie

- CPU-Instruktionen arbeiten am schnellsten mit Register
- Register bekommen Daten aus Hauptspeicher (L_1, L_2, L_3 Caches beschleunigen diesen Zugriff)
- Daten im Hauptspeicher werden auf Festplatte persistiert



Hauptspeicher (RAM) (Stand 2018)	Festplatte (HardDisk) (Stand 2018)
Volatile (geht bei Stromausfall verloren)	Persistent (Dateien bleiben erhalten)
Byte und Wort-granularer Zugriff	Block-basierter Zugriff (512 Bytes - 4kB) (Dateien)
Zugriff relativ schnell (<50ns) (wenn auch 1000x langsamer als Register)	Zugriff sehr langsam (4-8 ms) (also nochmals 1000x langsamer als RAM!)
Bandbreite relativ schnell (~150 GB/s)	Bandbreite gering (200 MB/s)
„Teuer“ (10 €/GB)	Billig (0,04€/GB) (hier 150x billiger!)
Typische Größe: 16 GB bei PCs	Bis zu 14 TB Festplatte



Abstand Schreib- und Lesekopf : 3nm
Umdrehungen: 10000 U/min
Zugriffszeit: 4ms
Bandbreite: 250 MB/s
Speicherkapazität: heute ca. 20TB

- Zum Auslesen von Daten schickt der Prozessor Kommandos an den Festplattencontroller
- Daten sind auf drehenden Festplatte organisiert als Spuren und Sektoren
- Dateisysteme abstrahieren diese Information von Spur/Sektoren

nicht flüchtiger Speicher



5 1/4 zoll



3 1/2 zoll Disc



LTO Magnetband



Hard disk Drive HDD



Solid State Drive SSD

Floppy Disc

Unix: EFS, XFS

Linux: Ext2, GFS

MacOs: APFS, HFS+

Windows: NTFS, FAT32

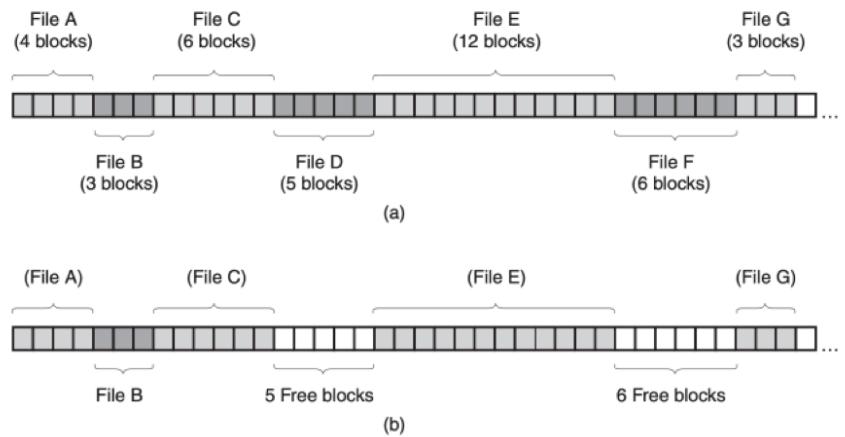
Unterstützte Größen:

- * Dateinamenlänge: 8 Zeichen für Dateinamen und 3 Zeichen für die Endung
 - heute max. 255 UTF-8 Zeichen
- * Anzahl der Dateien pro Verzeichnis: $2^{64}-1$
- * Größe einer Datei heute: von 4GiB (FAT32) bis 16GiB (ZFS)
- * Größe des Dateisystems heute:
 - 2TiB (FAT32) bis 512 YiB (GPFS)
- * Verschiedene Verzeichnisstrukturen
- * Metadaten
 - Zeiten (letzter Zugriff,...)
 - Versionierung (auf alte Versionen zugreifbar?)
 - Rechte
- * Datenkorrektheit & Datenschutz
 - Journaling
 - Interne Prüfsummen um Daten und Metadaten zu sichern
 - Verschlüsselung zum Schutz vor unbefugtem Zugriff
 - Erweiterte Zugriffsbeschränkungen
- * Datenstrukturen

Speicherung von Dateien

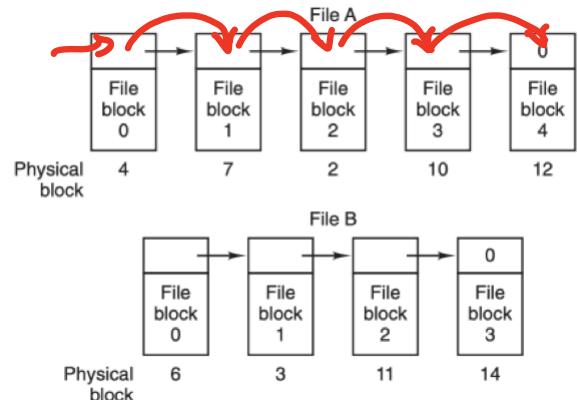
Kontinuierliche Allokation

- Vorteile:
 - » Einfach zu implementieren
 - » Sehr gute Lese-Performance
- Nachteile:
 - » Löschen einer Datei hinterlässt Platz der ggf. nicht ohne signifikanten Zusatzaufwand verwendet werden kann.
 - » Fragmentierung (leere nicht zusammenhängende Räume)
 - » Probleme mit wachsenden Dateien Reorganisation durch Kopieren
- Wird für generelle R/W Dateisysteme nicht verwendet
- Sinnvoll für read-only FS

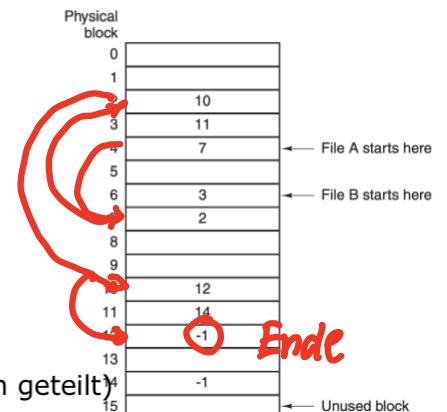


- Verkettete Listen
 - » Statt kontinuierlichen Dateien werden Blöcke allokiert die durch Zeiger verbunden werden
- Vorteile:
 - » Freigegebene Blöcke haben eine einheitliche Größe und können von anderen Dateien verwendet werden
 - » Dateien können besser wachsen und schrumpfen
- Nachteile:
 - » Schlechtere Lese-Performance
 - » Zeiger reduziert Blockgröße auf ungünstigen Wert

Verkettete Liste:



File Allocation Table:



→ **File allocation Table** (muss aber im Speicher sein, da von allen Dateien geteilt)

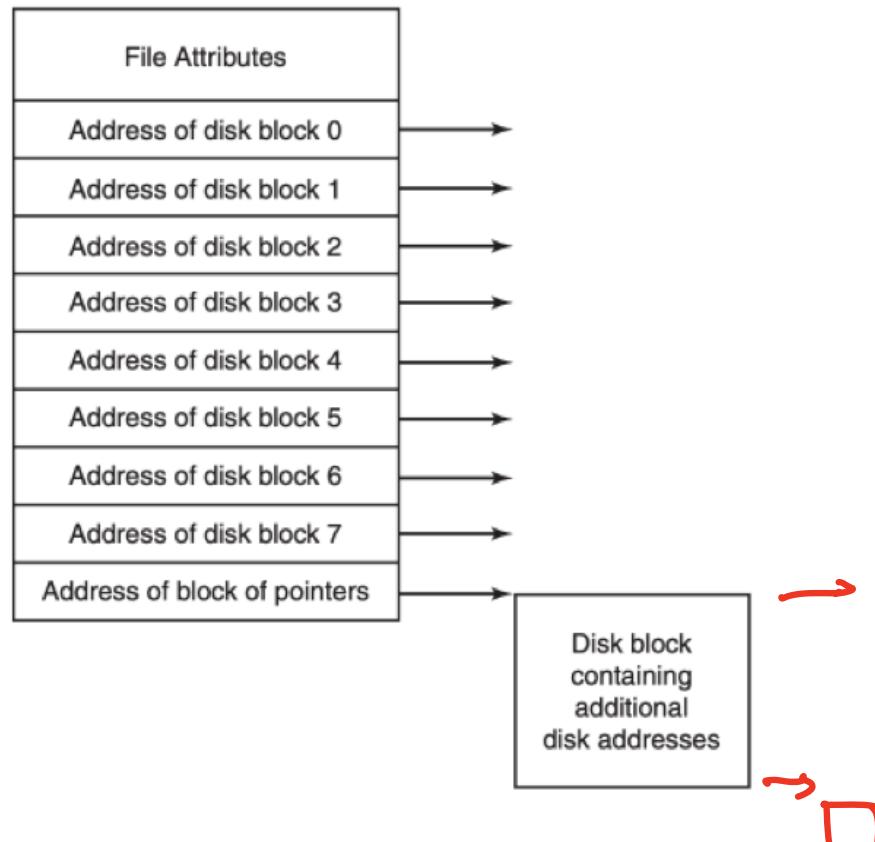
Index Nodes (inodes)

- **Vorteile:**

- » Vorteile wie FAT aber pro Datei eine einzelne Tabelle
- » Kann gut an physische Größen (Blöcke auf Disk) angepasst werden
- » Kann dynamisch wachsen und schrumpfen

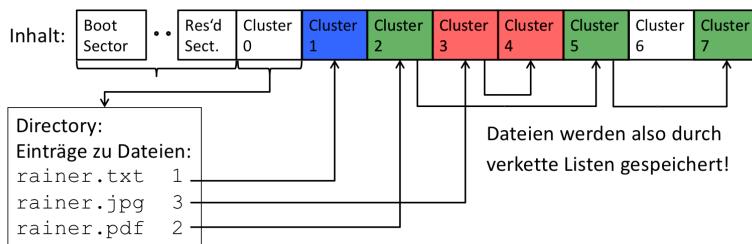
- **Nachteile:**

- » Inode Größe limitiert Größe von Datei → weiterer Verweis auf Inode zweiter Ebene...



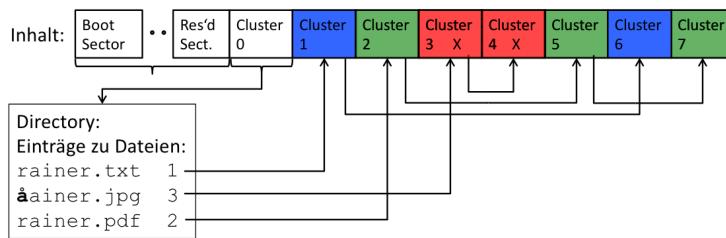
FS	Max.Länge Dateinamen	Max. Dateigröße	Max. Dateisystemgröße	Access Control Lists	Creation Time	Disk Quota
FAT32	8.3, bzw. mit LFN 255 Unicode	4 GB	2 TB	Nein	Nein	Nein
ReFS WinServer 2012	255 Unicode	16 EB	256ZB	Ja	Ja	Nein
Lustre	255	64 PB (auf Basis v. ext4)	1 YB	Ja	Z. T. nicht im Client	Nein
GPFS	255	512 YB	512 YB	Ja	Ja	Ja
EXT4	255	16 TB	64 ZiB	Ja	Ja	Ja

FAT (File Allocation Table) ist das am häufigsten verwendete File System, da es schon immer von Windows unterstützt wurde (Windows weit verbreitet), ein einfaches Dateisystem ist und geringe RAM und CPU Anforderung hat



Annahme:

- Ein Cluster besteht aus 8 Sektoren á 512 Bytes
- Datei `rainer.txt` ist 4095 Bytes lang,
- Datei `rainer.jpg` ist 4097 Bytes lang,
- Datei `rainer.pdf` ist 12288 Bytes lang.
- Dateien werden mittels verketteter Cluster gespeichert:

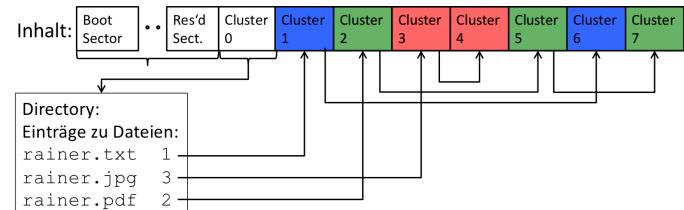


Annahme:

- Die Datei `rainer.jpg` wird gelöscht!

Faktisch wird nur der Name aus dem Verzeichnis „entfernt“ indem der 1. Buchstabe im Namen 0xE5 (â) ersetzt wird – das Cluster ist „gelöscht“.

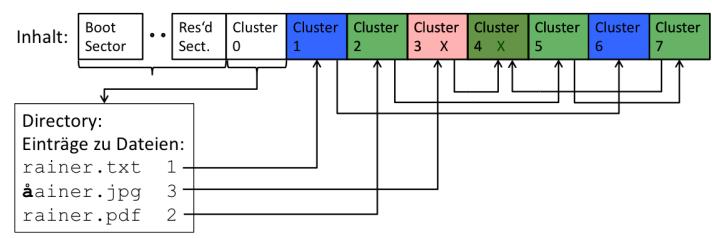
- Dateien werden mittels verketteter Cluster gespeichert:



Annahme:

- Die Datei `rainer.txt` wächst nun auf 4097 Bytes!

- Dateien werden mittels verketteter Cluster gespeichert:



Annahme:

- Die Datei `rainer.jpg` wurde gelöscht!

Wächst nun eine Datei wie `rainer.pdf` werden die „freigegebenen“ Datenbereiche überschrieben.

NTFS:

- hierarchisches Dateisystem
 - 1 Wurzel Verzeichnis
 - Unterverzeichnisse mit bis zu 2^{32} (ca. 4 Mrd.) Dateien erstellen
- Lange (255 Zeichen) und kurze (8.3 Zeichen) Dateinamen
- Max. Dateisystemgröße: 256 TiB - 64 kB à 64 kB Cluster
- Max. Dateigröße : 256 TiB - 64 kB

- Verschlüsselung des gesamten Dateisystems
- Journaling Dateisystem:
 - Änderungen werden in das Journal-Log geschrieben, Änderung wird vorgenommen Journal-Log wird bereinigt
- Hard Links:
 - mehrere Namen können auf eine Datei zeigen
- Dateistreams:
 - Eine Datei kann mehrere Streams beinhalten
 - Datei.txt liefert andere Daten als Datei.txt.stream1
- File Compression:
 - Dateien werden blockweise à 16 Cluster komprimiert (4kB Cluster → 64kB) (nur Cluster \leq 4kB)
- nutzt B+-Bäume zur effizienten Speicherung und Sortierung von Daten

ReFS:

- 4kB oder 64kB Clustergröße nicht so empfehlenswert, da wenn man nur 1 Byte bearbeiten will, man 6kB auf die Festplatte runterschreiben muss
- B+-Bäume
- Datei- & Verzeichnisgröße 2^{64}
- Mehrere Streams
- Bitlocker Verschlüsselung
- Keine Komprimierung, kein Disc Quota

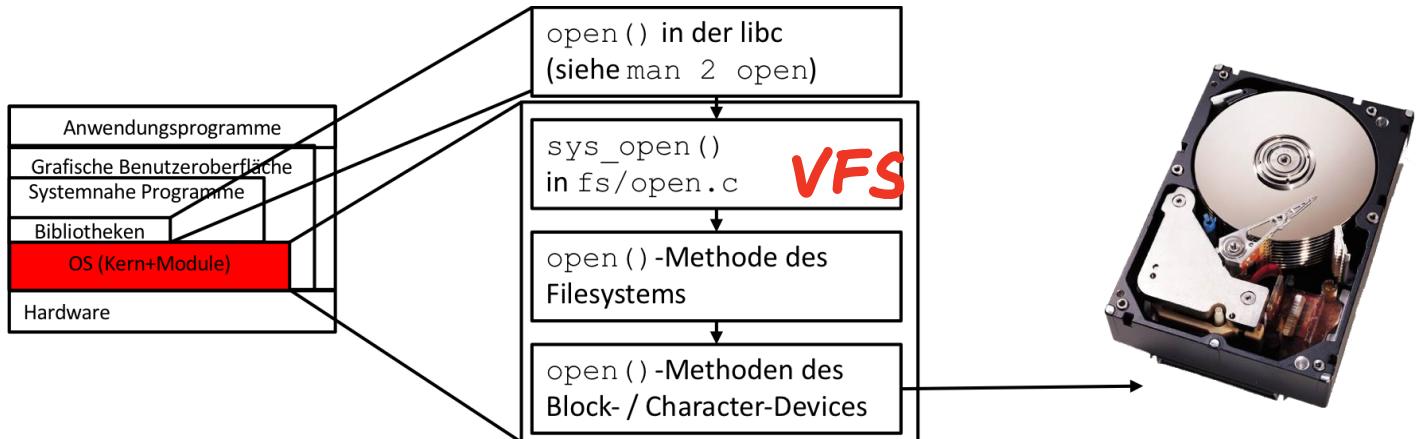
EXT4:

- Erweiterung von EXT 2 und EXT3
- Journaling
- Vorwärts- und rückwärtskompatibel
- Dank altem Fundament sehr stabil
- hierarchisches Dateisystem
- 255 lange Dateinamen
- Max. Dateisystemgröße: 1EB (à 4kB Clustergrößen)
- Max. Dateigrößen: 16TB (")
- Journaling in 3 Modi:
 - Performance
 - Metadatenjournal
 - höchste Datensicherheit
- B⁺- Bäume um Dateien schnell zu finden
 - Bis zu 4 Extents à 128 MB (ohne jeden 4kB Block speichern zu müssen)
- Schnelleres Erstellen und Prüfen

Repräsentation im Kernel

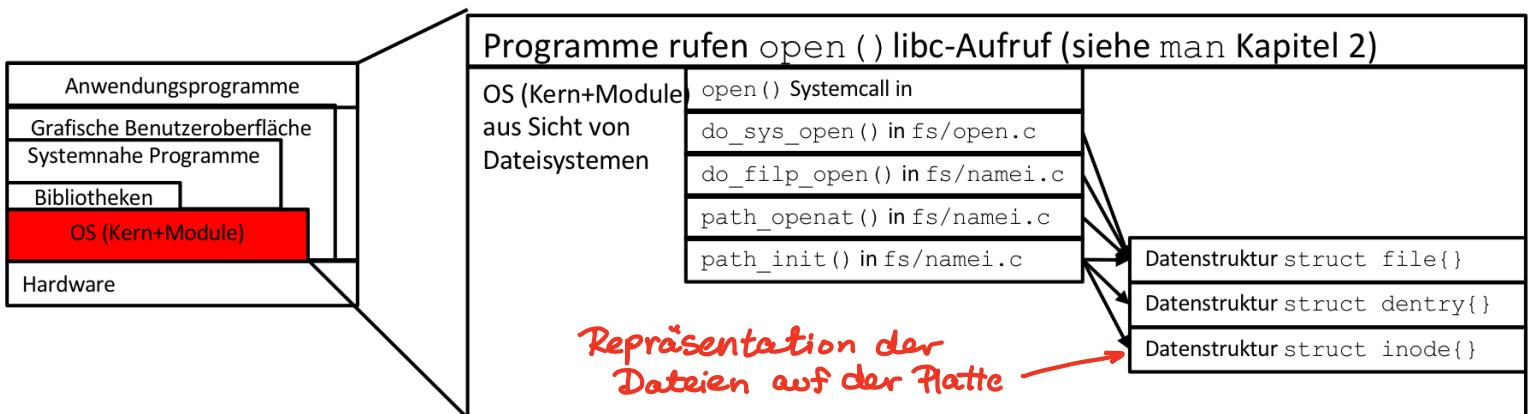
- Linux hat für die Darstellung von beliebigen FS ein genial einfaches Konstrukt implementiert
→ **Virtual Filesystem Switch (VFS)**

- Alle Dateisysteme müssen bestimmte Funktionen ("Methoden") implementieren
- Dann ist es egal, auf welchem Dateisystem die Datei liegt



Die wichtigsten Strukturen:

- struct file {} → Datei
- struct dentry {} → „directory entry“ → Verzeichnis
- struct inode {}



- Jede Datei wird durch struct file {} beschrieben
- Jeder Verzeichniseintrag ist ein struct dentry {}
- Jeder Block auf der Festplatte ist ein struct inode {}

- Struktur struct file{} in linux/fs.h:

```
struct file{
    struct path;                                // Diese struct enthält dentry
                                                // und Infos zum Mount Point.
    struct file_operations * f_op; // Methoden des FS!
    atomic_long_t f_count;                      // Reference-/Usage-Counter
    unsigned int flags;                         // Flags beim Öffnen
    fmode_t f_mode;                            // Modus (rwx) beim Öffnen
    loff_t f_pos;                             // Dateiposition
    unsigned int f_uid;                         // User's UID
    unsigned int f_gid;                         // User's GID
};

};
```

Die Struktur inode {} enthält alle Informationen, die der Kernel benötigt um ein File oder Directory zu ändern

```
const struct inode_operations *i_op; // Methoden auf inodes
struct super_block      *i_sb;     // Superblock Information
struct address_space     *i_mapping;
dev_t                   i_rdev;    // Device: HDD, FD, usw.
loff_t                  i_size;    // Größe d. Inodes (loff!)
                                // Zugriff mit i_size_read()
struct timespec          i_atime;   // Zeit des letzten Zugriffs
struct timespec          i_mtime;   // Zeit schreibender Zugriff
struct timespec          i_ctime;   // Zeit der Erstellung
unsigned short           i_bytes;   // \
unsigned int              i_blkbits; // Größe aus Blocks&Bytes!
blkcnt_t                 i_blocks;  // /
unsigned long             dirtied_when; // jiffies wann geändert
...
...
```

- Dateisysteme unterscheiden sich nach ihrem primären Anwendungszweck (Festplatte, SSD, USB-Stick)
 - Moderne Dateisysteme bringen komplexe Funktionen mit bezüglich Fehlertoleranz, Deduplication, Metadaten, Indexierung, Verteilung

`sudo tune2fs -l /dev/sdal` features, die unser gemountetes EXT4 besitzt

- !! letzte Eingabe wird wiederholt
- mkfs neues Filesystem anlegen (default: ext2)
- mkfs.ext4 EXT4 Filesystem anlegen
- Sudo mkfs -q /dev/ram1 8192 Beispiel um ein Device im Ram anzulegen

ls -lah Inhalte des Dir ausgeben (informativ → human readable)
sudo mount -t auto /dev/ram1 mein_dir/
/dev/ram1 wird gemountet in mein_dir
→ "auto" Dateisystem ist egal (default: EXT2)

`dd if=/dev/zero of=ext4.img bs=1024 count=$((128 * 1024))`
→ kopiere (dd) den Inhalt von /dev/zero (if: input file) in ext4.img (of: output file). Wie viel soll kopiert werden?
→ bs = 1024 (bs = block size) für die Größe $128 \cdot 1024$ Blöcke

sudo umount /dev/ram1 /dev/ram1 unmorten

Dateisystem vergrößern

sudo resize2fs /dev/ram1 64M Größe auf 64 MB ändern

du -h ext4.img Größe ausgeben

→ angezeigte Größe bspw. 4.4MB

- eigentliche Größe 128MB (durch Nullen (sparse files), die nicht abgespeichert werden müssen)

dd if=/dev/zero of=ext4.img bs=1024 count \$((128*1024)) seek
((128*1204)) Inhalt wird dem Ende vom of eingefügt

Dateisysteme selbst anlegen

- Im gemounteten Dateisystem eine Text Datei anlegen und editieren
- Die Datei bzw. ihren Inhalt im Dateisystem finden
- Datei unmounten und vergrößern
 - Danach das EXT4-FS mit resize2fs vergrößern

Funktioniert auch bei NTFS und FAT (anders)

Interprozesskommunikation:

Beispiel: Lochkarten (1930)

Kommunikation kann durch Datenaustausch mittels:

- Dateien
- Netzwerk
- Gemeinsamen Speicher durchgeführt werden

ISO/OSI 7 Schichten Modell

ISO/OSI-Schicht:	Funktion:	ISO/OSI-Schicht Computer 1:	ISO/OSI-Schicht Computer 2:
7. Anwendung (Application)	Netzwerkanwendungsschicht, die Daten verarbeitet...	7. Anwendung (Application)	7. Anwendung (Application)
6. Darstellung (Representation)	Datenrepräsentation, Verschlüsselung, Umwandlung in maschineneunabhängige Daten	6. Darstellung (Representation)	6. Darstellung (Representation)
5. Sitzung (Session)	Verwaltung von Sessions zwischen Anwendungen	5. Sitzung (Session)	5. Sitzung (Session)
4. Transport (Transport)	End-to-End Verbindung, Zuverlässigkeit, Flow Control	4. Transport (Transport)	4. Transport (Transport)
3. Vermittlung (Network)	Wahl des Pfads und Logische Adressierung	3. Vermittlung (Network)	3. Vermittlung (Network)
2. Sicherung (Data Link)	Physikalische Adressierung <i>Sichert Bits, Framing</i>	2. Sicherung (Data Link)	2. Sicherung (Data Link)
1. Bitübertragung (Physical Layer)	Signalverarbeitung (<i>z.B. Antennen / Kabel!</i>)	1. Bitübertragung (Physical Layer)	1. Bitübertragung (Physical Layer)

```

graph TD
    subgraph Computer1 [ISO/OSI-Schicht Computer 1]
        7A[7. Anwendung (Application)] <--> 6A[6. Darstellung (Representation)]
        6A <--> 5A[5. Sitzung (Session)]
        5A <--> 4A[4. Transport (Transport)]
        4A <--> 3A[3. Vermittlung (Network)]
        3A <--> 2A[2. Sicherung (Data Link)]
        2A <--> 1A[1. Bitübertragung (Physical Layer)]
    end
    subgraph Computer2 [ISO/OSI-Schicht Computer 2]
        7B[7. Anwendung (Application)] <--> 6B[6. Darstellung (Representation)]
        6B <--> 5B[5. Sitzung (Session)]
        5B <--> 4B[4. Transport (Transport)]
        4B <--> 3B[3. Vermittlung (Network)]
        3B <--> 2B[2. Sicherung (Data Link)]
        2B <--> 1B[1. Bitübertragung (Physical Layer)]
    end
    1A --> 1B
    1B --> 1A
    2A --> 2B
    2B --> 2A
    3A --> 3B
    3B --> 3A
    4A --> 4B
    4B --> 4A
    5A --> 5B
    5B --> 5A
    6A --> 6B
    6B --> 6A
    7A --> 7B
    7B --> 7A

```

Ein Protokoll ist eine exakte Vereinbarung, nach der Daten zwischen Computern bzw. Prozessen ausgetauscht werden, die (durch ein Netz) miteinander verbunden sind

Diese Komm.-eigenschaften sind zu unterscheiden:

- Verbindungsorientierte- / Verbindungslose Kom. **TCP** vs. **UDP**
- Punkt-zu-Punkt- / Mehrpunktverbindung **IP** vs. **IGMP**
- Uni-/Bidirektionale Verbindung **Unix Pipes** vs. **TCP Sockets**
- Zwei-Seitige / 1-seitige Kommunikation **TCP** vs. **RDMA**
- Übertragungsgranularität **Byte-** vs. **Paketorientierte** Verbindung
- Reihenfolgentreu / Reihenfolge nicht garantiert
- Korrektheitsgarantie **Prüfsummengesichert** / **Ungesicherte** Kommunikation
- Qualitätsgarantien **Definiertes Quality of Service** vs. **Best-Effort**
- Kontrollfluss-Steuerung? **Zur Vermeidung von Überlast**
- Erkennung und Behandlung von Kollisionen? **Bei Kollisionen anderes Handeln**

Uriterien zur Evaluation von Kommunikation:

* Bandbreite

- wie viele Daten können pro Sekunde übertragen werden?

* Latenzzeit:

- Verzögerung bis die ersten Daten beim Empfänger angelkommen sind / bi-directionale Latenz, also bis der Sender weiß, dass die Daten beim Empfänger angelkommen sind

* Message Rate

- wie viele Pakete können pro Sekunde eingespeist werden

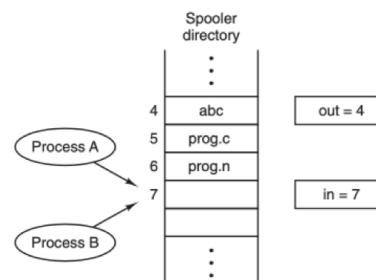
Interprozesskommunikation & Synchronisation

- Dateien
- Signale
- Sockets
- Message Queues
- Pipe
- Named Pipe
- Semaphore
- Memory Mapping
- Message Passing

Lost Conditions

Beispiel:

- » Printer Spooler hat eine Liste von zu druckenden Dateien
 - » Zwei Prozesse möchten eine Datei zeitgleich drucken
 - » Prozess A liest 7 als nächsten freien Platz und merkt sich 7 und wird schlafen gelegt
 - » Prozess B liest 7 als nächsten freien Platz, merkt sich 7 und schreibt seine zu druckende Datei an 7
 - » Prozess A wird aufgeweckt und schreibt seine zu druckende Datei an die zuvor gemerkte 7
- Lost Update: „Race Condition“ hängt von Scheduling ab. Nicht-deterministische Fehler.



Kommunikation über gemeinsame Ressourcen

Es muss verhindert werden dass durch gleichzeitigen Zugriff Inkonsistenzen in Daten entstehen
→ Mutexe / Semaphoren und Locks (gegenseitiger Ausschluss)

Gemeinsame Nutzung beschränkter Mittel (z.B. Peripherie-Geräte) → Mutex - Verfahren
→ Meistens werden komplexere Scheduling - Methoden benötigt

Übergabe von Daten / Nachrichten von einem Prozess an einen anderen → Interprozesskommunikation

Steuerung von Unterprozessen durch Signale / Abbrechen von Prozessen oder warten auf Terminierung

Semaphore: eine Zählvariable speichert Zustand
→ Operationen "Passiere" & "Verlasse"
belgegen Semaphor und geben frei

Monitore: Bündelung von Daten und Mutex zu einem Objekt, zusammen mit Monitor Operationen

Mutex (binärer Semaphor):

→ Read / Write Mutex (parallel mehrere Reads erlaubt)

Auf BS-Ebene

Spinlocks: Die CPU versucht ständig aktiv den Lock zu nehmen (while-Schleife, die prüft bis Bedingung erfüllt ist (busy waiting))

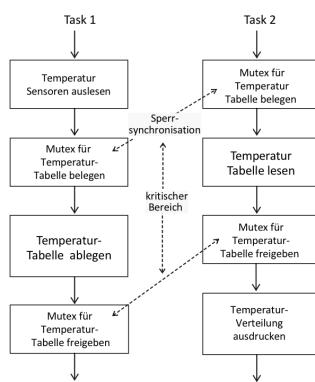
Atomic Operations: Unterstützt durch f/w
→ CPU bietet Intrusionen

Synchronisation – Variante 1

- Die Sperrsynchronisation

Wichtig hierbei:

- Kritischer Bereich sollte so kurz wie möglich gehalten werden
- Verschiedene Möglichkeiten Sperren zu implementieren (atomics, spinlocks, mutex)

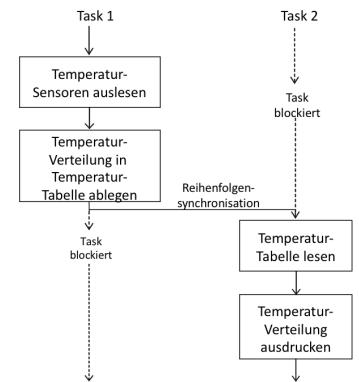


Synchronisation – Variante 2

- Reihenfolgensynchronisation

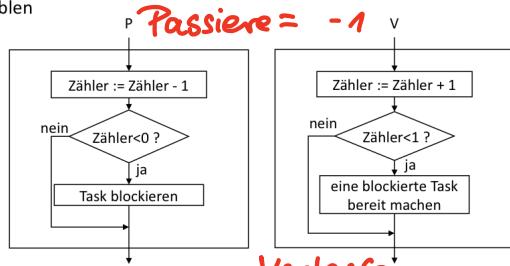
Wichtig hierbei:

- Implementation und Nutzung komplizierter!
- Bei großer Anzahl von Tasks/Threads nicht besonders skalierend...



Ein Semaphore besteht aus:

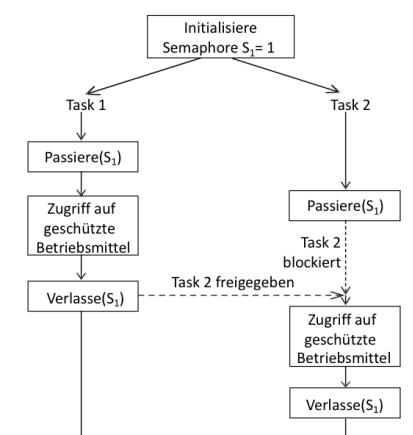
- zwei nicht unterbrechbaren Operationen P (Passieren) und V (Verlassen)
- einer Zählvariablen



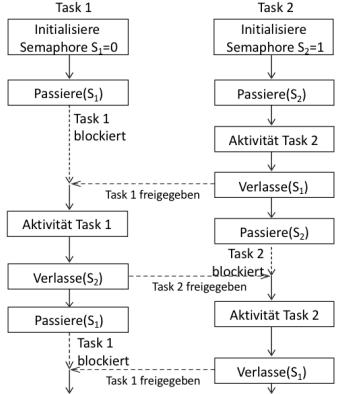
- Bei Standardbetriebssystemen: Bereitmachen eines beliebigen, wartenden Tasks bei V
- Bei Echtzeitbetriebssystemen: Bereitmachen des höchsprioren, wartenden Tasks bei V

- Sperrsynchronisation mit einem Semaphore

- Diese Form des Einsatzes von Semaphoren heißt auch Mutex (Mutual Exclusion)



- Reihenfolgensynchronisation mit zwei Semaphoren
- Zu beachten:
Benutzung von S1 vs. S2 in den einzelnen Tasks!



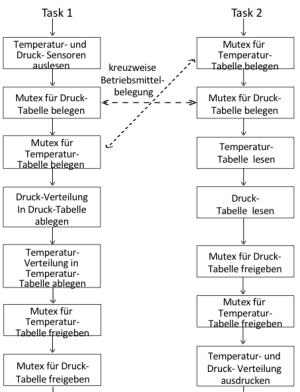
- Deadlock (Verklemmung): mehrere Tasks warten auf die Freigabe von Betriebsmitteln, die sich gegenseitig blockieren.

- Beispiel:
kreuzweise Betriebsmittelbelegung

Gegenmaßnahmen:

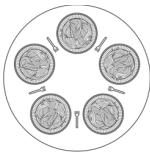
- Abhängigkeitsanalyse
- Vergabe der Betriebsmittel in immer gleicher Reihenfolge
- Timeout & Backoff

Livelock (Starvation): Tasks werden andauernd durch andere Tasks von der Ausführung gehindert.



Dining Philosophers

5 Philosophen, 5 Gabeln, glitschige Spaghetti → Jeder Philosoph braucht zum Essen zwei Gabeln. Ablauf: denken, essen, ...



Beispiele für mögliche Algorithmen:

Alg 1 (Deadlock): Philosph nimmt Gabel und wartet auf andere Gabel. **Alle nehmen rechte Gabel gleichzeitig.**

Alg 2 (Livelock) : Philosph nimmt Gabel, gibt sie wieder ab, wenn die andere Gabel nicht frei ist. **Alle nehmen Gabel gleichzeitig, geben Gabel gleichzeitig wieder her.**

Alg 3: Philosophen verwenden Mutex furs Essen.
Langsam, da nur einer ist.

Alg 4: Philosophen verwenden Mutexe und Array von Semaphoren (eine für jeden Philosoph)

Signale werden von der Shell versandt

* **CTRL-C** → Tastatur hw sendet Interrupt an CPU

→ Aktiver Prozess wird ermittelt und OS sendet SIGINT Signal (Interrupt)

* **CTRL-Z** → Tastatur hw sendet Interrupt an CPU

→ Aktiver Prozess wird ermittelt und OS sendet SIGTSTP Signal

= **\$ kill -z 'pidof firefox'**

- Vordefinierte Signale:

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

- Die default Aktion je Signal ist entweder:

- TERM Prozess terminiert oder
- CORE Prozess liefert einen Core-Dump

2) SIGINT : Interrupt from Keyboard

3) SIGQUIT: Quit from Keyboard

9) SIGKILL : Kill Signal

11) SIGSEGV: Invalid memory reference

14) SIGALRM: Timer Signal from alarm

15) SIGTERM: Termination Signal

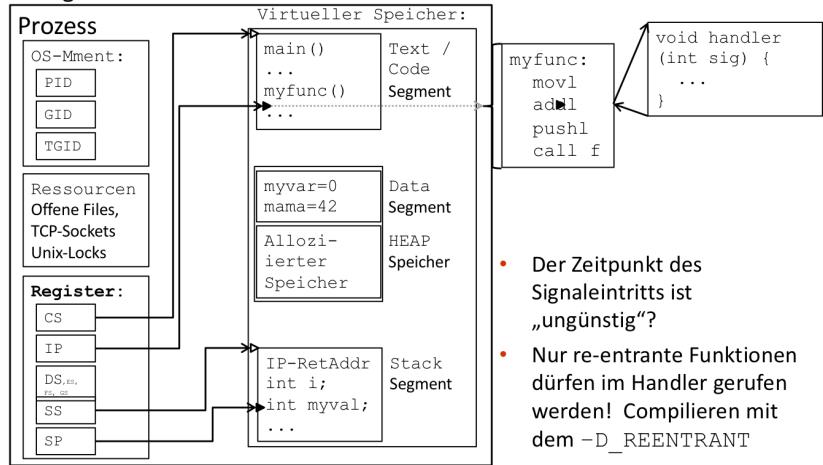
18) SIGCONT: Continue if stopped

19) SIGSTOP: Stop process

20) SIGTSTP: Stop typed at terminal

27) SIGPROF: Profiling timer expired

Das größte Problem ist:



- Der Zeitpunkt des Signaleintritts ist „ungünstig“?
- Nur re-entrant Funktionen dürfen im Handler gerufen werden! Compilieren mit dem `-D REENTRANT`

Pipes

→ Daten zwischen Prozessen austauschen

`./program1 | program2 # Pipes stdout to stdin`

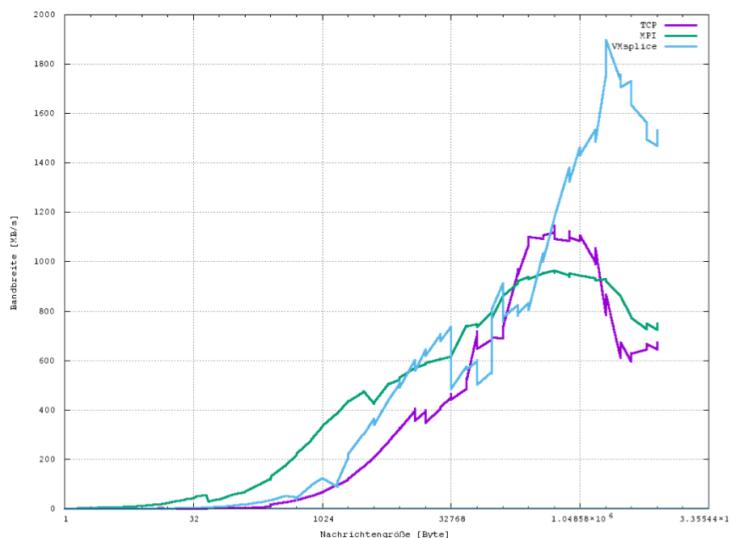
Pipe verbindet Standardausgabe von P1 mit der Standardeingabe von P2

- unidirektional

→ Eingabe vorne, Ausgabe hinten

Messungen auf der LIDA Maschine

(4-socket AMD mit 12 Kernen, Netpipe-3.7.2, OpenMPI-1.8)



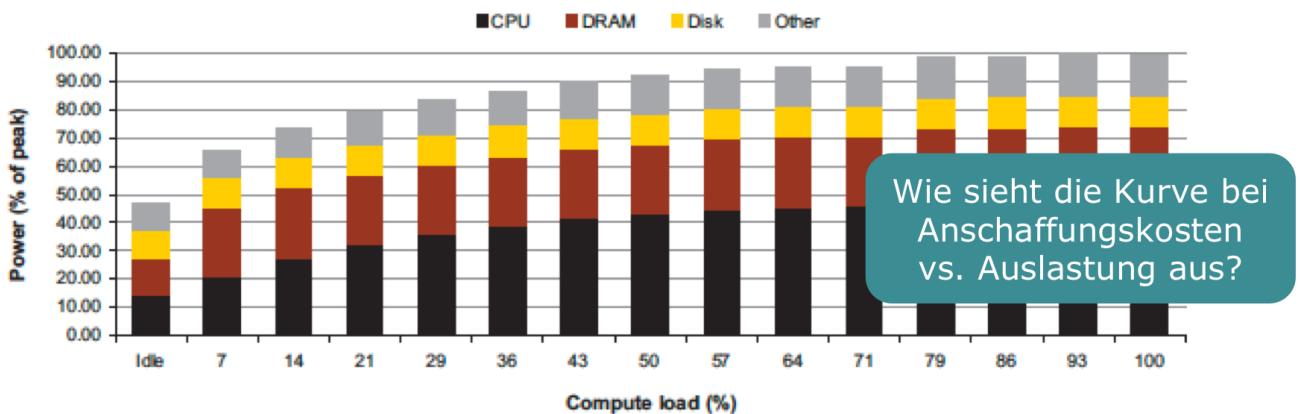
Netpipe: Performance tool

OpenMPI verwendet mmap als Zwischenspeicher zwischen den Prozessen

Vmsplice: selbst entwickelte Erweiterung, sodass Netpipe Vmplice verwenden kann

TCP: Kommunikation über Netzwerkprotokoll über localhost

Virtualisierung



Server power usage as load varies from idle to 100%

- » 50% peak power when idle
- » 70% peak power when 10% utilized
- » 90% peak power when 50% utilized

Most servers in common data centers are utilized 10% to 50%

Multicomputer: Mehrere Computer arbeiten zusammen
Vorteil: Performance, Security und Zuverlässigkeit
Mehrere Betriebssysteme

Ein VMM (Virtual Memory Monitor) erzeugt die Illusion von mehreren virtuellen Rechnern auf der gleichen physischen Hardware. **VMM = Hypervisor**

→ 1 Computer kann mehrere VMs hosten, wobei jeder Computer ein völlig anderes BS ausführt

Vorteile von Virtualisierung:

- * Software Crashes beeinflussen die anderen Systeme nicht
- * Parallel mehrere virtuelle Betriebssysteme
→ mehrere Anwendungen
- * Kontrolle über Gast-Systeme
- * Neue Ideen können ausprobiert werden
- * Einfache Verlagerung von Funktionen
- * Checkpointing und Migration
- * Verwenden von Legacy Anwendungen
- * Vereinfachung der Entwicklung für verschiedene Systeme
- * Durch Isolation: Unterschiedliche Kunden auf einer Hardware (Salami Beispiel)

Anfänge der Virtualisierung:
→ 1964 Mainframes

IT-Kosten werden gespart durch:

- Down-Sizing *Mit weniger, mehr erreichen: Last auf Server verteilen*
- Centralizing *Anstatt Infrastruktur mehrfach an versch. Orten → einmal optimal*
- Virtualizing *Beides zusammen durch virtuelle Maschinen auf wenigen Servern*

Anforderungen an Virtualisierung:

- Die Software, die auf dem Hypervisor läuft, muss sich wie auf der echten Hardware verhalten
- Virtualisierte Ressourcen muss der Hypervisor selbst in der Hand haben
- Schnelle Ausführung → Wenn zu einem guten Anteil trotzdem die Maschineninstruktionen ausgeführt werden können

Sensitive instructions : Instruktionen, die sich anders verhalten, wenn sie im Kernelmodus ausgeführt werden, als wenn sie im Benutzermodus ausgeführt werden. Dazu gehören Instruktionen, die E/A verwenden, die MMU-Einstellungen ändern usw.

Privileged instructions: Es gibt auch eine Reihe von Anweisungen, die einen Trap verursachen, wenn sie im Benutzermodus ausgeführt werden.

Eine Maschine ist nur dann virtualisierbar, wenn die sensiblen Instruktionen eine Teilmenge der privilegierten Instruktionen sind.

→ Wenn man etwas im User-Mode versucht, was man nicht tun sollte, sollte ein Trap aufgerufen werden

Control sensitive instructions:

- verändern die Konfiguration

Behavior sensitive instructions:

- Sind von der Konfiguration abhängig

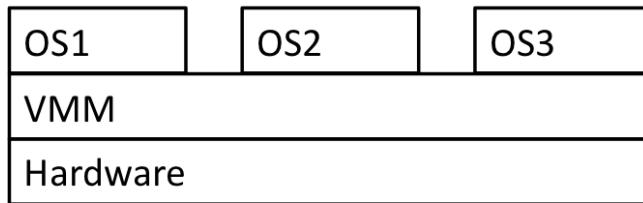
- Virtualisierung: eine Methode um Ressourcen zu entkoppeln und auf ausführende Einheiten zu verteilen.

- Methoden mit denen Ressourcen entkoppelt werden:
 - » Hardware und Software Partitionierung *
 - » **Zeitscheibenverfahren (time Sharing) ***
 - » Partielle und komplett Maschinensimulation *
 - » Quality of Service *
 - » Emulation

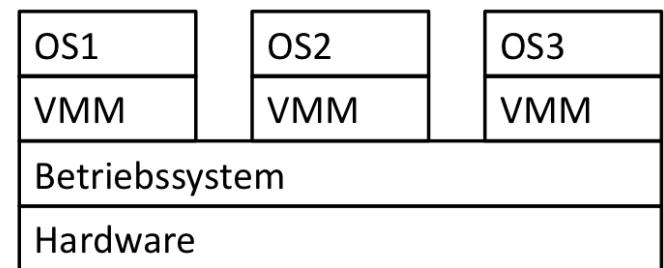
Methoden mit * brauchen Hardware-Unterstützung um die Entkoppelung durchsetzen zu können.

3 Schichten:

- Hardware-Level Virtualisierung
- Betriebssystem-Level Virtualisierung
- Hochsprachen Virtualisierung



Type 1 (Native / BareMetal)

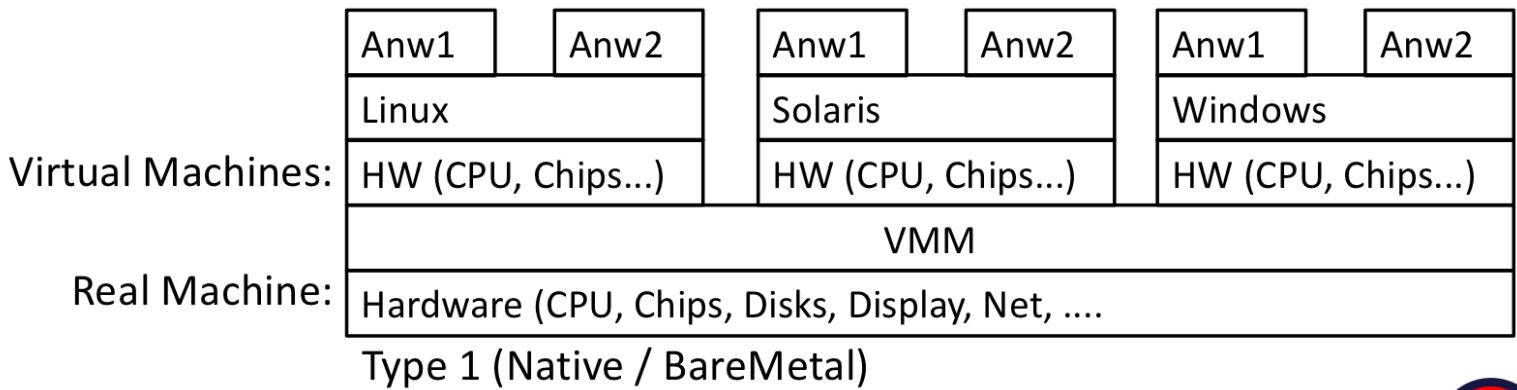


Type 2 („Hosted Hypervisor“)

HW-Level Virtualisierung

BS-Level Virtualisierung

HW-Level Virtualisierung im Detail:



Häufig von Web-Hostern mit vielen VMs pro Maschine

BS-Level Virtualisierung im Detail:

Eine feinere Unterteilung ist bei OS-Level Virt. notwendig

- Fully Virtualized:

Das Gast-OS ist nicht anzupassen! Alle Funktionen von Speicherverwaltung, über HW-Zugriffe (bspw. Grafikkarte) werden virtualisiert -- und teilweise nach Abfangen emuliert!

- ParaVirtualized:

Das Gast-OS **ist** anzupassen! Funktionen wie Speicherverwaltung werden beim Host-OS angefragt! Dies erlaubt weitaus effizientere Virtualisierung!

- HW-Assisted Virtualization:

Die Hardware unterstützt durch einzelne CPU-Instruktionen die Ausführung von OS als Gast in VMs.

Das ist mittlerweile auch auf x86 effizient (s. Artikel von VMware in ASPLOS).

- Partial Virtualized:

Der Host wird partitioniert, meist bezüglich des Speicheradressbereichs.

Die darin laufenden Gast-OS müssen nicht verändert werden!

Fully Virtualized im Detail:

- Alle CPU-Modi, die ein modernes Gast-OS erwartet, müssen abgefangen werden
 - Also Instruktionen, die nur in Ring-0 ausgeführt werden dürfen, sogenannte „trap-and-emulate“
- die sensiblen Funktionen sollen einen Trap aufrufen, dann geht es zum Hypervisor, welcher die Funktion der eigentlichen HW emuliert

Lasse Betriebssystem in ~~nicht~~ privilegiertem Kontext laufen (nicht Ring 0 sondern Ring 1 oder 2).

- Schütze tatsächliche Hardware vor Betriebssystem

„Trap and Emulate“

- Zugriff auf privilegierte Ressourcen löst Trap aus, wenn sie in unprivilegiertem Kontext (Ring 1 oder 2) ausgeführt wird → Hypervisor (Ring 0) wird aufgerufen

„Shadow Structures“

- Das Host System legt für wichtige Verwaltungsstrukturen (z.B. Page Table) eigene „Shadow Structures“ an
- Zugriff des Betriebssystems auf den eigenen Speicher löst keine Traps aus. Wann wird die Originalstruktur verändert? → Memory traces. Hypervisor bekommt nichts mit

„Tracing“, „Trace fault“, „Memory Traces“

- Speicherseiten mit Originalstruktur des Betriebssystems auf „Write-only“ setzen: Hypervisor wird jedes Mal aufgerufen wenn Schreibzugriff erfolgt.

- Was passiert, wenn das Gastbetriebssystem (das sich im Kernelmodus befindet) eine Anweisung ausführt, die nur erlaubt ist, wenn sich die CPU wirklich im Kernelmodus befindet?

- Bei CPUs ohne VT schlägt die Anweisung fehl und das Betriebssystem stürzt ab.
- Bei CPUs mit VT TRAP und Wechsel zu Hypervisor. Der HV prüft, ob Instruktion aus Anwendung oder BS des Gastsystems kommt. → dementsprechende Behandlung

VT Support:

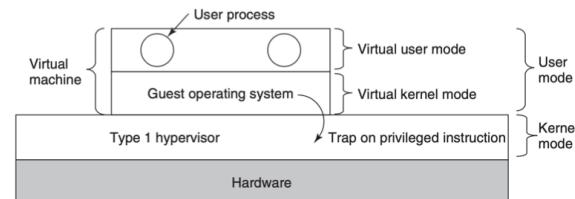
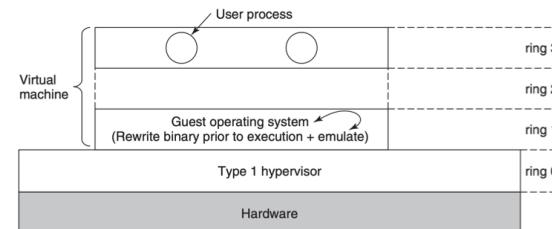


Figure 7-3. When the operating system in a virtual machine executes a kernel-only instruction, it traps to the hypervisor if virtualization technology is present.

Kein VT Support:



Binary Translation:

Ein „**basic Block**“ ist eine kurze, lineare Folge von Anweisungen, die mit einem Branch/Sprung endet. Per Definition enthält ein basic Block keinen Sprung, Aufruf, Trap, Return oder andere Anweisung, die den Kontrollfluss ändert.

Hypervisor scannt Basic Block vor Ausführung und ersetzt sensitive Instruktionen mit Call von Hypervisor-Prozedur.

Dynamische Übersetzung und Emulation klingen teuer, sind es aber in der Regel nicht.

- Übersetzte Blöcke werden zwischengespeichert.
- Die meisten Codeblöcke keine sensitiven oder privilegierten Anweisungen und können nativ ausgeführt werden.
- Der binäre Übersetzer kann alle Benutzerprozesse ignorieren. Diese werden ohnehin im nicht privilegierten Modus ausgeführt.

Virtualized Memory Extended Page Tables

Mapping von guest virtual addresses nach guest physical addresses und host physical addresses (machine physical addresses) nötig

- Hoher Aufwand bei Speicherzugriffen

Problem: Virt. Gastadresse wird nicht zwischengespeichert
→ Pagewalk

- Jede Ebene in der Paging-Hierarchie des Gasts führt zu einer weiteren Suche in den verschachtelten Seitentabellen des Hosts → Anzahl der Speicherverweise wächst quadratisch

Manche Hypervisor erlauben es mehr Speicher zu vergeben als physisch vorhanden ist

→ Paging und Swapping möglich

Swapping von Gast-Memory durch den Hypervisor kann sehr ineffizient sein:

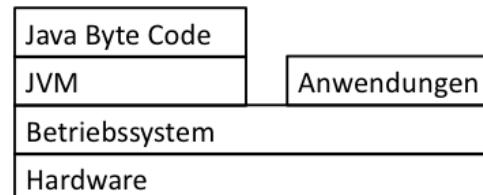
- Hypervisor swappt wichtige Seiten aus
- Hypervisor lagert aus bevor Gastsystem Seiten auslagert → Speicher wird durch HW ausgelagert, muss wieder eingelagert werden, damit er durch das Gastsystem ausgelagert werden kann → Hoher Aufwand

→ Lösung: Ballooning

- Balloon Modul im Gast verbraucht künstlich Speicher, um Gastsystem direkt zum Auslagern zu bewegen.
- Speicher wird verfügbar

Hochsprachen Virtualisierung

- Die Anwendung einer Hochsprache wird in einer VM ausgeführt:
- Die Virtualisierungsschicht selbst ist eine „Anwendung“ für das OS.



- **Vorteil:** Anwendungen sind nur abhängig von der VM-schicht.
- **Nachteile:**
 - » Durch Virtualisierung geht Performance verloren
 - » HW kann nicht durch mehrere VMs verwendet werden (bspw. GPU)
- Diese VM kann auf beliebiger Hardware / OS laufen, z.B.:
 - » Die Java VM ist standardisiert; viele Sprachen führen darin aus.

Technologie	Host OS Typ != Guest OS	Vollwertige Virtuelle Maschine	Guest OS funktioniert ohne Modifikationen	Unterstützt andere Architekturen	Beispiele
Virtualisierung auf Betriebssystem Ebene	Bedingt: -Solaris BrandZ unterstützt Linux -Open VZ unterstützt andere Distributionen	Bedingt: -Angepasstes OS -Je nach Technologie keine komplette Ressourcen Trennung	Nein -Devices müssen angepasste werden -Nicht alle Funktionen des OS nutzbar	Nein	-Solaris Zones -OpenVZ -LXC
Hardware Virtualisierung	Ja	Ja	Ja -Angepasste Treiber für Performance	Nein	KVM, XEN
Para-Virtualisierung	Bedingt: -Guest OS muss die Virtualisierung unterstützen	Bedingt: Angepasstes OS	Nein -Devices, libraries, kernel Anpassungen notwendig	Nein <i>da auf der CPU arbeitet</i>	XEN, VMware
Hardware Emulation	Ja	Ja	Ja	Ja	-Qemu (z.B. Android Emulator)

Quelle: Inovex

Wenn eine wichtige Maschine (die 24/7 läuft und nicht abgeschaltet werden darf) verlagert werden muss:

- Hypervisors entkoppeln die VM von der physischen HW
- Verschieben von Speicher Festplatte (RAM), Netzwerk (IP) im Betrieb
- **Live Migration**: Verlagerung der laufenden VM auf andere HW

Pre-copy memory migration: Speicherseiten des Systems kopieren, während es läuft und Anfragen bearbeitet

- Da viele Speicherseiten nicht viel beschrieben werden, ist das Kopieren sicher

- Seiten, die sich nach dem Kopieren verändern, werden als "dirty" markiert und erneut kopiert
- Nachdem die meisten Speicherseite kopiert wurden, werden die restlichen Seiten kopiert während das System kurzzeitig pausiert wird

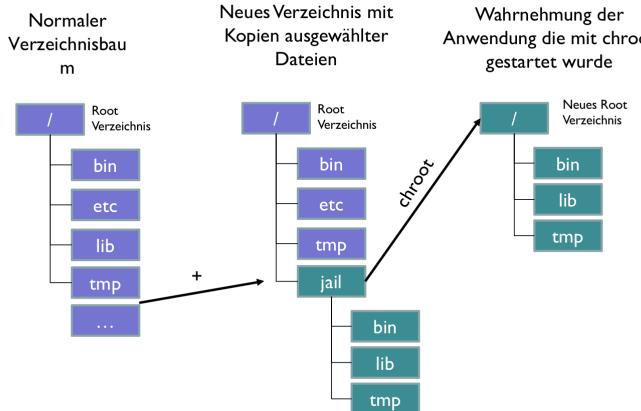
Checkpointing: Sichern des Zustands der laufenden Maschine, später wieder zurücksetzen auf Checkpoint
 → Sicherung des Zustands der pausierten Maschine auf der Festplatte

Einfache Lsg: Einfache Kopie

Schnelle und effiziente Lsg: Copy on write

- Virtualisierung spart, verschlingt aber dennoch Ressourcen:
 - Speicher für die Installation eines gesamten Virt. OS mit Distribution, etc.
 - Prozessor-Overhead um VMs zu verwalten, Hypervisor auszuführen
 - Human-Ressourcen um die VMs zu administrieren!
- Der letzte Schrei sind Container / Containervirtualisierung:
 - Unter Unix gibt es seit langem chroot-Umgebungen:
 Ein Programm wird darin eingesperrt (so als ob / nun /tmp/service wäre); diese Programm kann nicht mehr daraus ausbrechen (und bspw. /etc/passwd aufmachen).
 - Linux Container basieren auf Control-Groups (Cgroups)
 Docker ist eine Implementierung davon.
 - Unter anderen Unix:
 Jails, OpenVZ, Solaris Zones, Linux Containers (LXC)

Root - Jail im chroot



Schritte zum Erstellen eines chroot-environment

A) Identifikation der nötigen Programme

» Beispiel: nur ls und bash werden benötigt

B) Identifikation der Abhängigkeiten der Programme

» Welche Libraries werden verwendet?
» \$> ldd /bin/ls und \$> ldd /bin/bash nennen alle Abhängigkeiten

C) Reproduktion der nötigen Umgebung der Programme

» Anlegen von /usr/lib, /usr/bin, /lib64, ... im neuen chroot Pfad
» Selektives Füllen der Verzeichnisse mit den benötigten libraries (aus ldd /bin/ls, ... bekannt)

D) Ändern des Stammverzeichnisses (Root) auf das neue chroot jail und ausführen des Programms im chroot jail

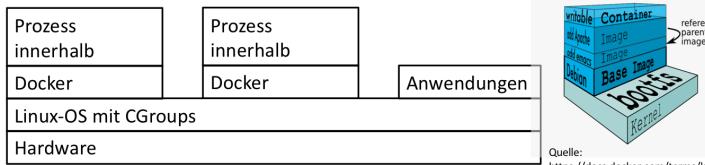
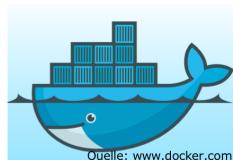
» sudo chroot /.../chroot/ /bin/bash
» Nun sieht das Programm nur den selektiv für das Programm vorbereiteten Teil des Dateisystems.

docker [naut.]: der Dockarbeiter, der Hafenarbeiter

40' Container haben Welthandel revolutioniert!

Container in Virtualisierung:

- Provide a uniformed wrapper around a software package: «Build, Ship and Run Any App, Anywhere» [\[www.docker.com\]](http://www.docker.com)



Vorteile:

- Weniger physikalische Hardware (statt 5% Auslastung mehrerer Servern, dann 60% auf nur einem) – oder gar keine HW mehr
- Kapselung von Anwendern/Services/Code! (Spectre...)
- Tools um viele Server (viele VMs) schnell zu warten
- Bessere Ausfallsicherheit durch Migration von VMs
- Einfachere Reinstallation von einem Snapshot/Image

Nachteile:

- Die VMs (OS darin!) muss gepflegt werden
- Am Ende mehr VMs als es ursprünglich physikalische Rechner...
- Erhöhte Kosten durch VM Software (Lizenzen)
- Virtualisierung kostet Performance (von 4% - 20%).

- Datenaustausch zwischen Gast & Host ist essentiell
- Einige Aspekte der Integration sind:
 - Direkter Zugriff auf HW: Bspw. Gast nutzt GPU
 - Mehrere Möglichkeiten Gast als Netzwerk-Peer einzubinden
 - Zugriff auf gemeinsame Dateien: mount des Hosts im Gast
 - Copy-Paste: Selektion und Kopieren von Daten über OS-Grenze hinaus.
 - Drag-and-Drop: Auswahl und Verwendung der gezogenen Daten sind von VM-SW für Host und Gast gut umgesetzt.
- Am Besten man braucht keine „Full-Blown“ VM: Container sind leichtgewichtig, kaum Overhead, volle Trennung von Prozessen mittels Cgroups in CPU-Sets, Memory-Sets und sogar in separate Netze möglich

Applikation

- Für den Benutzer sichtbar
- A) Zugreifbar über Web
- B) Zugreifbar über App

Plattform

- Building-Block für Anwendungen
- Plattformdienste erleichtern Anwendungserstellung

Infrastruktur

- Verwendete Hardware

