Linux – Prinzipien und Programmierung

Dr. Klaus Höppner

Hochschule Darmstadt – Sommersemester 2014

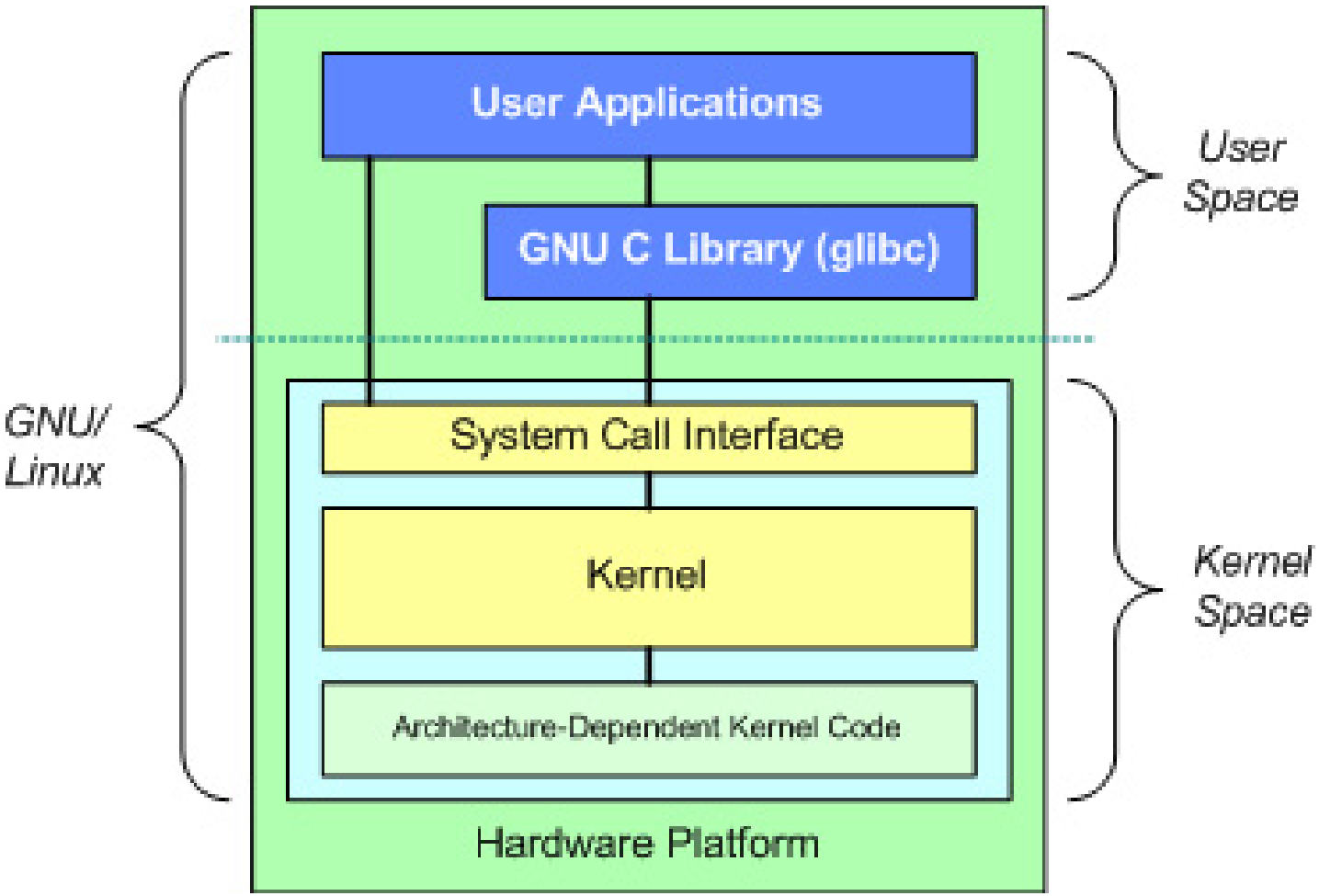
Aufgabe des Kernels

Herzstück des Betriebssystems

- Verwaltung und Bereitstellung von Hardware und Systemressourcen
- Typische Komponenten:
  Interrupt-Handler, Process Scheduler, Memory Management, System Services (Networking, IPC)
- Unterscheidung: User Space $\leftrightarrow$ Kernel Space

Grundlegendes Schema:
Eine Anwendung wird im User Space in einem Prozess ausgeführt, macht einen *System Call* (z. B. `printf`), dieser wird im Kernel Space im Prozess-Kontext ausgeführt.
Zusätzlich laufen im Kernel Space Aktionen ab, die nicht im Kontext eines Prozesses liegen, z. B. Behandlung von Interrupts.

Schaubild


User Applications
GNU C Library (glibc)
System Call Interface
Kernel
Architecture-Dependent Kernel Code
Hardware Platform
User Space
Kernel Space
GNU/ Linux

Monolithischer Kernel

Der Linux-Kernel ist ein monolithischer Kernel.
Bedeutet: Prinzipiell kann der Kernel ein einzelnes Executable sein, das in einem einzelnen Prozess mit globalem Adressraum läuft.

Vorteil:	einfacher zu implementieren, ressourcen-schonend (keine IPC nötig)
Nachteil:	Kein Schutzmechanismus, Kernel Code hat freien Zugriff auf den gesamten Adressraum

Besonderheit bei Linux: Zwar monolithisch, aber modular, d. h. der Kernel hat die Möglichkeit, Kernel Code dynamisch hinzu zu laden oder zu entfernen.

Microkernel

Bei Microkerneln werden die Aufgaben des Kernels von einzelnen Prozessen ausgeführt, so genannten Servern.

Vorteil: Jeder Server braucht nur die Rechte, die für seine Aufgabe notwendig sind; getrennte Adressräume

Nachteil: Mehr Aufwand bei Implementation; IPC notwendig, das kostet Zeit.

Beispiele: Minix, GNU/Hurd

Zwitter: Windows ab NT (Hybridkernel)

Versions-Schema

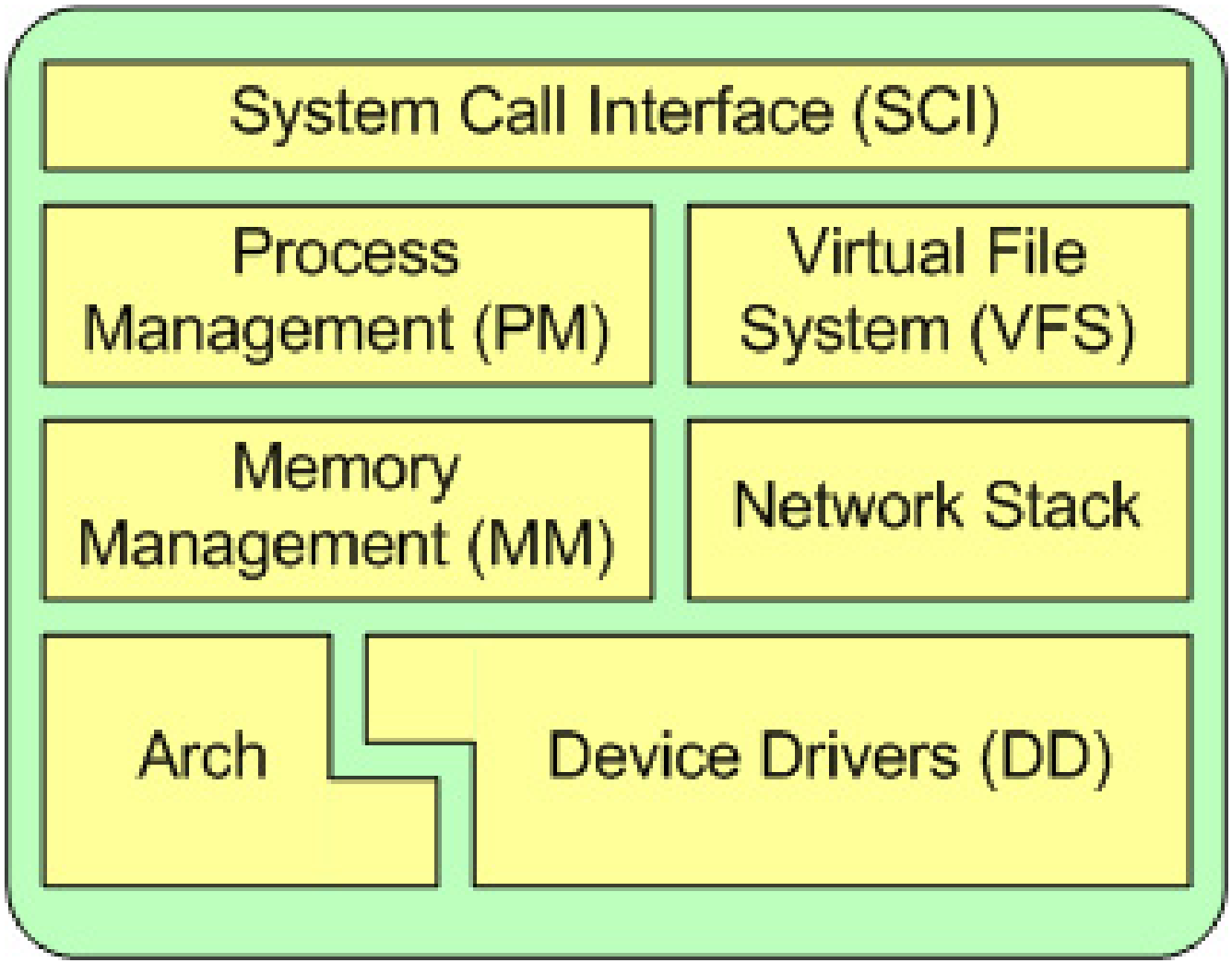
Jede Linux-Kernelversion hat eine Nummer, bestehend aus

- Major Version
- Minor Version
- Revision
- optional Stable Release-Nummer (seit 2005)

Aktuelle Nummer: Ab 2012 wird die Major Version 3 verwendet, aktuell 3.15.1 (letzte stabile Version der Linie 2.6: 2.6.32.63

Hinweis: Die frühere Unterscheidung (gerade Minor-Nr. Produktionsversion, ungerade Entwicklerversion) wurde inzwischen aufgegeben.

Schaubild: Komponenten des Kernels



Komponenten des Kernels

Neben dem Interface für Systemaufrufe (SCI – *System Call Interface*) besteht der Linux-Kernel aus folgenden grundlegenden Subsystemen:

Prozess-Management Ausführung von Prozessen (aus Kernelsicht werden die *Threads* genannt). Applikationen werden über das SCI Funktionen zur Verfügung gestellt, wie `fork`, `exec`, `kill`, ... Bedeutender Teil ist natürlich das Hin- und Herschalten zwischen den Prozessen, *Scheduling*.

Speicher-Management Verwaltung von *Memory Pages* (üblich 4 KB-Blöcke bei 32bit-Systemen, 8KB bei 64bit). Kritisch, da innerhalb des Kernels jedweder Speicher ungeschützt ist.

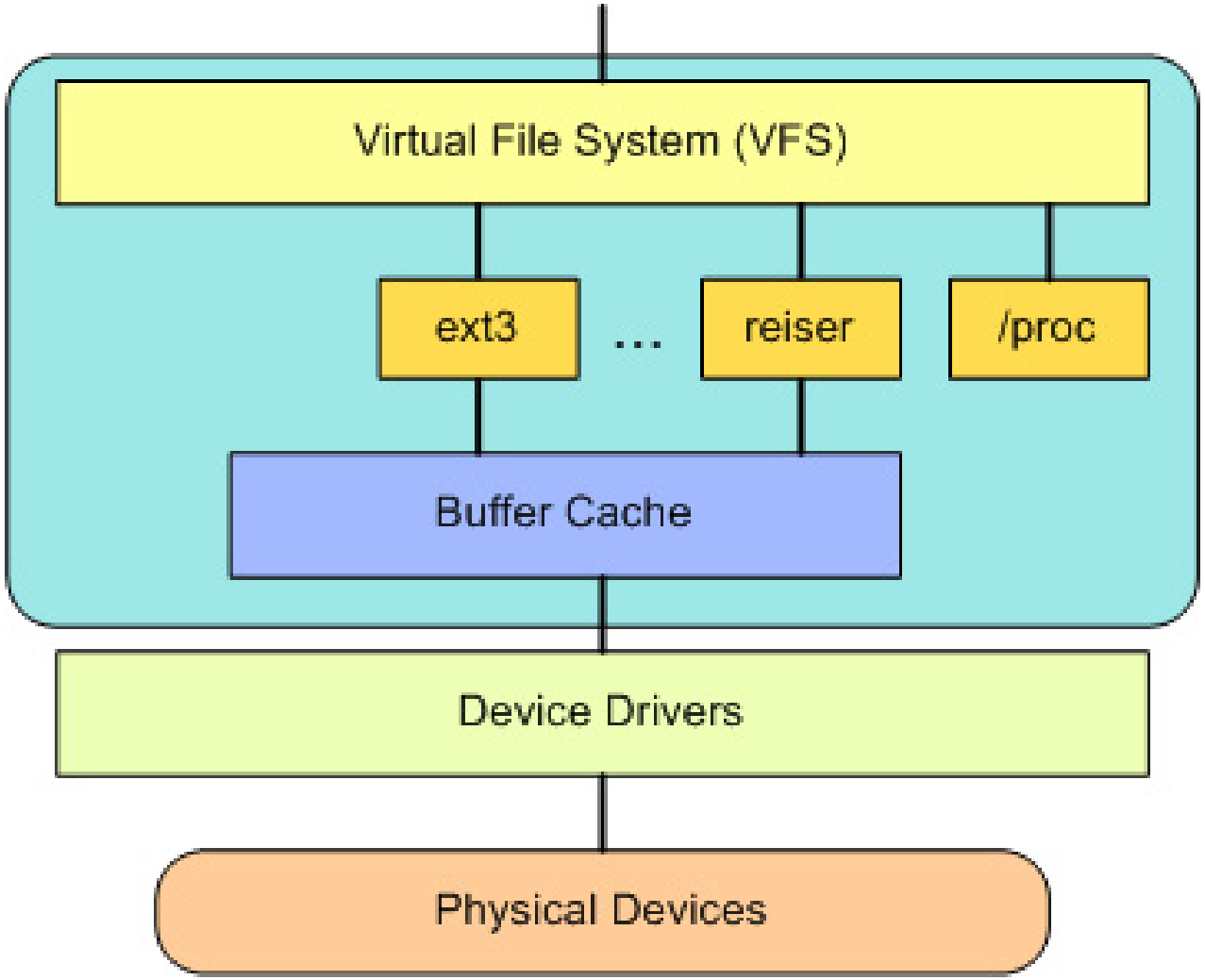
Komponenten des Kernels (Forts.)

Virtuelles Dateisystem Das VFS (*Virtual File System*) ist eine Abstraktionsschicht über den realen Dateisystemen (wie `ext2`, `ext3`, ...) und macht diese über eine allgemeine API zugänglich.

Netzwerk-Stack Der Kernel stellt die in den letzten beiden Vorlesungen beschriebenen Protokollschichten (IP – Netzwerkschicht; TCP, UDP – Transportschicht) bereit.

Gerätetreiber (*Device Drivers*) für z. B. Schnittstellen, Netzwerkkarten, Sound usw.
Arch Architektur-spezifischer Code, z. B. für i386, PowerPC, M68k usw.

Schaubild: VFS



Herunterladen der Kernel-Sources

Zum Runterladen der Quellen stehen zwei Wege zur Verfügung:

- Archiv einer bestimmten Version runterladen, z. B.: https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.15.1.tar.bz2 (oder von einem der Mirrors, s. http://www.kernel.org/mirrors/)
- Sich eine lokale Kopie des Code-Repositories (git) erstellen:

  ```
  git clone
  git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
  ```

  (in einer Zeile)
  Letzteres ist natürlich bequemer, wenn Änderungen am Kernel jeweils nachvollzogen werden sollen.

Der Source-Tree

arch	Architektur-spezifische Quelldateien
block	Block I/O
crypto	Crypto API
drivers	Gerätetreiber
firmware	für bestimmte Geräte benötigt
fs	Dateisystem
include	Kernel-Headerdateien
init	Boot- und Initialisierungs-Code
ipc	Code für Interprozess-Kommunikation
kernel	Kern, z. B. Scheduler
lib	Hilfsbibliotheken
mm	Speichermanagement
net	Netzwerk
samples	Beispiele
scripts	zum Kompilieren benötigte Skripte
security	Sicherheit
sound	Audio
tools	Entwicklungstools
usr	für initramfs benutzt
virt	Infrastruktur für Virtualisierung

Kernel kompilieren

Bevor der Kernel kompiliert werden kann, muss er (passend) konfiguriert werden.
Hierfür stehen mehrere Wege zur Verfügung:

- `make defconfig` Voreinstellung für vorhandene CPU und
- ggfs. danach `make menuconfig` zum Anpassen der Konfiguration oder
- `make oldconfig` zum neuen Einlesen der (ggfs. manuell geänderten) Konfigurationsdatei.

Danach wird mit `make` das komprimierte Kernel-Image `arch/cpu/boot/bzImage` erzeugt.

Module werden mit `make modules_install` installiert.

Der Scheduler

Im Laufe der Entwicklung des Kernels gab es verschiedene Änderungen am *Scheduling*, die jeweils zu einer grundlegenden Neuimplementierung des *Schedulers* führten.

bis Linux 2.4 Scheduler mit $O(n)$ (bei jedem Task-Switch wurde die gesamte Prozessliste durchsucht, welcher als nächster drankommen soll).

Linux 2.5, frühes Linux 2.6 $O(1)$-Scheduler, sortierung der Prozesse in Runqueues, Zeit zum Auswahl des nächsten Prozesses unabhängig von Gesamtzahl der Prozesse, teilweise mit heuristischen Elementen bei der Bewertung von Prozessen.

Der Scheduler (Forts.)

seit Linux 2.6.23 *Completely Fair Scheduler*, CFS Sortierung der Prozesse in einem balancierten binären Baum (red-black tree) anhand der Abweichung zwischen der real erhaltenen CPU-Zeit und der fairen CPU-Zeit, die dem Prozess aufgrund seiner Priorität zugestanden hätte.
Autor: Ingo Molnár

Prinzipielle Betrachtung des Scheduling

Wenn ein Scheduler an n Prozesse eine CPU-Zeit t_{ges} verteilen soll, so ist die faire Zeit pro Prozess

$$t_i = t_{\text{ges}} \cdot \frac{w_i}{\sum_i w_i}$$

In einem idealen Prozessor würde diese Formel für jede (beliebig kleine) Zeit t_{ges} gelten. Für einen realen Prozessor ist dies natürlich nicht realisierbar, jeder Prozesswechsel dauert eine gewisse Zeit $\rightarrow$ bei zu kleiner Zeitscheibe (*timeslice*) CPU-Verbrauch für Scheduling statt für die eigentlichen Prozesse.
Konflikt: Timeslice groß, effizient aber interaktive Prozesse scheinen zu „hängen“ vs. Timeslice klein, also ineffizient.
Granularität: Mindestzeit zwischen zwei Taskswitchen (üblich 1 ms)

Unterschiedliches Vorgehen beim CFS-Scheduler

Der CFS-Scheduler benutzt weder Timeslices noch Runqueues, sondern pro Prozess wird (auf ns genau) die Abweichung zwischen realer CPU-Zeit und fairer Zeit protokolliert.
Nachdem der Scheduler einen Prozess zum Ausführen ausgewählt hat, wird dieser so lange ausgeführt, bis ein anderer Prozess nach der Abweichung zwischen realer und fairer Zeit „dringender“ wird.
Bei Linux ergibt sich das Gewicht eines Prozesses für die CPU-Zeitverteilung aus dem so genannten Nice-Level.

Das Virtuelle File System

Linux unterstützt eine Vielzahl von Dateisystemen:

- Physische Dateisysteme, wie `ext2`, `ext3`, `reiserfs`, `iso9660`, `(v)fat`,
- Netzwerk-Dateisysteme, insbes. `NFS`,
- Virtuelle Dateisysteme, wie das `proc`-Filesystem.

Merksatz: *Everything is a file*

Als Abstraktionsschicht stellt der Kernel das Virtuelle Dateisystem `VFS` zur Verfügung.

Wie kommen File Systems in den Kernel?

Für jedes Dateisystem gibt es unterhalb von `fs` ein Unterverzeichnis mit dem Namen des Dateisystems, also z. B. `ext2`.

Die Datei `fs/ext2/super.c` definiert nun eine Funktion, die das Dateisystem registriert:

```
int init_ext2_fs(void)
{
    return register_filesystem(&ext2_fs_type);
}
```

mit

```
static struct file_system_type ext2_fs_type = {
    ext2_read_super, "ext2", 1, NULL
};
```

Die Struktur `file_system_type`

Die Struktur `file_system_type` enthält:

- einen Zeiger auf eine Funktion zum Lesen des *Superblocks* des Dateisystems.
  Der Superblock enthält Informationen wie Name des Dateisystems, einen Zeiger auf das Device (bei physikalischen FS), Blockgröße, Status, ...
- Name des Dateisystems (so wie er dann beim Befehl `mount` bei der Option `-t` angegeben wird),
- Die Angabe, ob das Dateisystem auf einem physikalischen Device aufsetzt (dann `1`), so wie z. B. `ext2` ein Device (z. B. `/dev/hda1`) braucht.
- Einen Nullpointer (warum?)

Durch das `register_filesystem` wird das Dateisystem einer *Linked List* von bekannten Dateisystem hinzugefügt (beim Anhängen eines neuen Eintrags in die Liste wird der NULL-Pointer des bisherigen letzten FS dann überschrieben).

Mounten eines Dateisystems

Beim Mounten eines Dateisystems laufen folgende Schritte ab:

1. In der Linked-List der bekannten Dateisystem wird das entsprechende Dateisystem gesucht.
2. Aus der Struktur `file_system_type` wird die Funktion zum Lesen des Superblocks aufgerufen.

Soll nun in dem Dateisystem eine Datei gefunden werden, so werden die Pfadkomponenten beginnend vom ersten Inode aus rekursiv gesucht. Dabei unterstützt jeder Inode eine Menge von Inode-Operationen, die in einem Zeiger auf eine Struktur des Typs `inode_operations` beschrieben sind (und die sich z. B. zwischen Inodes, die Dateien oder Verzeichnisse bezeichnen, unterscheiden können),

Die Struktur inode_operations

```
struct inode_operations {
    struct file_operations * default_file_ops;
    int (*create) (struct inode *,const char *,int,int,struct inode **);
    int (*lookup) (struct inode *,const char *,int,struct inode **);
    int (*link) (struct inode *,struct inode *,const char *,int);
    int (*unlink) (struct inode *,const char *,int);
    int (*symlink) (struct inode *,const char *,int,const char *);
    int (*mkdir) (struct inode *,const char *,int,int);
    int (*rmdir) (struct inode *,const char *,int);
    int (*mknod) (struct inode *,const char *,int,int,int);
    int (*rename) (struct inode *,const char *,int,struct inode *,const ch
    int (*readlink) (struct inode *,char *,int);
    int (*follow_link) (struct inode *,struct inode *,int,int,struct inode
    int (*readpage) (struct inode *, struct page *);
    int (*writepage) (struct inode *, struct page *);
    int (*bmap) (struct inode *,int);
    void (*truncate) (struct inode *);
    int (*permission) (struct inode *, int);
    int (*smap) (struct inode *,int);
};
```

Memory Management

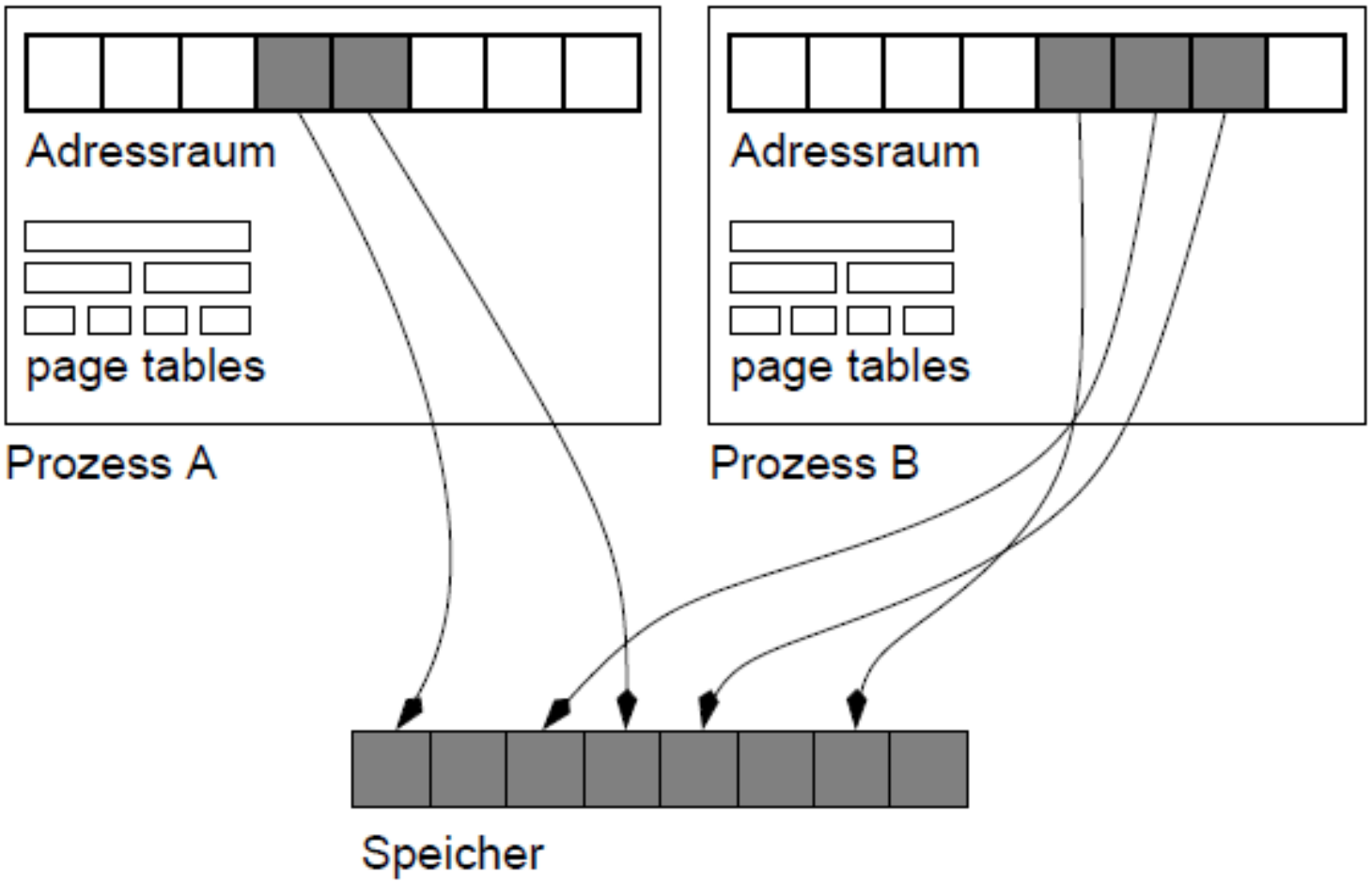
Der Linux-Kernel unterteilt den Arbeitsspeicher in Blöcke (*Memory Pages*) von 4 bzw. 8 KB.
Hierbei erhält jeder Prozess (im Userspace) einen eigenen, geschützten Speicherbereich für

- Code
- Daten (Stack und Heap)

Naiver Ansatz: Jeder Prozess bekommt ein zusammenhängendes Segment im Speicher. Simpel, aber schlecht skalierbar (dynamische Speicherallokation?)

Prozesse arbeiten mit *Virtuellem Speicher*, der auf die physikalischen Pages gemappt wird.

Schaubild


Adressraum
page tables
Prozess A
Adressraum
page tables
Prozess B
Speicher

Analyse

Vorteile:

- Beliebige Zuordnung von Memory Pages zu Prozessen
- Adressen prozessweit, nicht mehr global (perfekter Schutz)
- Prozesse sehen keine Speicherfragmentierung

Wichtige Begriffe

Page fault/Segmentation fault Nicht gestatteter Zugriff auf Speicher, z. B. Schreibzugriff auf RO-Section. Führt dazu, dass der Kernel dem Prozess das Signal SIGSEV schickt (Folge i. A. Core Dump)

Bus Error Zugriff auf Speicher, der physikalisch nicht zugänglich ist (Beispiel: Shared Memory über VME-Bus, wenn im adressierten Slot gar keine Karte steckt).

Copy on Write Wichtige Maßnahme des Linux Kernels beim Fork: Der virtuelle Speicher wird dupliziert, verweist aber noch auf den originalen physikalischen Speicher. Erst wenn Eltern- oder Kindprozesse in Speicherblöcken schreiben, werden diese vorher kopiert.

Laden von Kernelmodulen

Linux unterstützt die Auslagerung von Kernel Code in Module. Diese können mit `insmod` dynamisch hinzugeladen werden.

Zum Arbeiten mit Modulen existieren folgende Befehle:

- `insmod` zum Laden eines Moduls in den Kernel,
- `rmmod` zum Entladen eines Moduls,
- `lsmod` zum Auflisten der geladenen Module,
- `modinfo` zum Anzeigen von Infos zu einem Modul.

Hinweis: Beim Laden eines Moduls, das nicht erklärt, unter GPL zu stehen, wird der Kernel als *tainted* markiert (damit gibt es keinen Support seitens der Kernel-Community, weiterhin können Funktionen des Kernels ausschließlich GPL-Modulen zugänglich gemacht werden).

Einfaches Modul

Ein Modul muss mindestens zwei Funktionen aufweisen: je eine, die beim Laden bzw. Entladen ausgeführt wird:

```
/*
 *  hello.c - Demo module
 */
#include <linux/module.h>   /* Needed by all modules */
#include <linux/kernel.h>   /* Needed for KERN_INFO */
#include <linux/init.h>     /* Needed for the macros */

static int __init init_hello(void)
{
    printk(KERN_INFO "Hello, world\n");
    return 0;
}

static void __exit cleanup_hello(void)
{
    printk(KERN_INFO "Goodbye, world\n");
}

module_init(init_hello);
module_exit(cleanup_hello);
```

Analyse

Im vorliegenden Beispiel werden die Funktionen `init_hello` und `cleanup_hello` über die Makros `module_init` und `module_exit` als Routinen definiert, die beim Laden bzw. Entladen des Moduls aufgerufen werden.

Hier tun diese beiden Funktionen nichts außer dem Aufruf von `printk`. Dieses Makro (!) ist analog zu `printf`, nur dass die Meldungen dem Kernel-Logger übergeben werden, der sie je nach Schweregrad (hier `KERN_INFO`) nach `/var/log/messages` und/oder auf die Konsole schreibt (oder je nach Konfiguration direkt wegschmeißt).
Achtung: `printk(KERN_INFO "Dies ist ein Text")` ist kein Tippfehler!

Devices

Selbstgeschriebene Kernel-Module sind meist Device-Treiber, die mit einem Device-Knoten unterhalb von `/dev` Ein- und Ausgabeoperationen durchführen.

Ein solcher Device-Knoten wird mit dem Befehl
`mknod` *`name typ major minor`*
wobei der Typ entweder `b` (block device), `c` oder `u` (character device) oder `p` (FIFO) ist. Die Major-Nummer dient zur Identifikation des Kernelmoduls, das zur Ein- bzw. Ausgabe über das Device zuständig ist, die Minor-Nummer wird beim `open` übergeben und kann bei Bedarf vom Kernelmodul verwendet werden.

Beispiel eines Character-Devices

```
/*
 *  chardev.c: Creates a read-only char device that says how many times
 *  you've read from the dev file
 */

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h>    /* for put_user */

/*
 *  Prototypes - this would normally go in a .h file
 */
int init_module(void);
void cleanup_module(void);
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *,
       size_t, loff_t *);
```

Beispiel eines Character-Devices (Forts.)

```
#define SUCCESS 0
#define DEVICE_NAME "chardev"
/* Dev name as it appears in /proc/devices   */
#define BUF_LEN 80      /* Max length of the message from the device */

/*
 * Global variables are declared as static, so are global within the file.
 */

static int Major;       /* Major number assigned to our device driver */
static int Device_Open = 0; /* Is device open?
                 * Used to prevent multiple access to device */
static char msg[BUF_LEN];   /* The msg the device will give when asked */
static char *msg_Ptr;

static struct file_operations fops = {
    .read = device_read,
    .write = device_write,
    .open = device_open,
    .release = device_release
};
```

Beispiel eines Character-Devices (Forts.)

```
/*
 * This function is called when the module is loaded
 */
int init_module(void)
{
  Major = register_chrdev(0, DEVICE_NAME, &fops);

  if (Major < 0) {
    printk(KERN_ALERT "Registering char device failed with %d\n", Major);
    return Major;
  }

  printk(KERN_INFO "I was assigned major number %d. To talk to\n", Major);
  printk(KERN_INFO "the driver, create a dev file with\n");
  printk(KERN_INFO "'mknod /dev/%s c %d 0'.\n", DEVICE_NAME, Major);
  printk(KERN_INFO "Try various minor numbers. Try to cat and echo to\n");
  printk(KERN_INFO "the device file.\n");
  printk(KERN_INFO "Remove the device file and module when done.\n");

  return SUCCESS;
}
```

Beispiel eines Character-Devices (Forts.)

```
/*
 * This function is called when the module is unloaded
 */
void cleanup_module(void)
{
  /*
   * Unregister the device
   */
  unregister_chrdev(Major, DEVICE_NAME);
}

/*
 * Called when a process tries to open the device file, like
 * "cat /dev/mycharfile"
 */
static int device_open(struct inode *inode, struct file *file)
{
  static int counter = 0;

  if (Device_Open)
    return -EBUSY;
```

Beispiel eines Character-Devices (Forts.)

```
  Device_Open++;
  sprintf(msg, "I already told you %d times Hello world!\n", counter++);
  msg_Ptr = msg;
  try_module_get(THIS_MODULE);

  return SUCCESS;
}

/*
 * Called when a process closes the device file.
 */
static int device_release(struct inode *inode, struct file *file)
{
  Device_Open--;    /* We're now ready for our next caller */

  /*
   * Decrement the usage count, or else once you opened the file, you'll
   * never get get rid of the module.
   */
  module_put(THIS_MODULE);

  return 0;
}
```

Beispiel eines Character-Devices (Forts.)

```
/*
 * Called when a process, which already opened the dev file, attempts to
 * read from it.
 */
static ssize_t device_read(struct file *filp,
        char *buffer, size_t length, loff_t * offset)
{
  /*
   * Number of bytes actually written to the buffer
   */
  int bytes_read = 0;

  /*
   * If we're at the end of the message,
   * return 0 signifying end of file
   */
  if (*msg_Ptr == 0)
    return 0;
```

Beispiel eines Character-Devices (Forts.)

```
  /*
   * Actually put the data into the buffer
   */
  while (length && *msg_Ptr) {

    /*
     * The buffer is in the user data segment, not the kernel
     * segment so "*" assignment won't work.  We have to use
     * put_user which copies data from the kernel data segment to
     * the user data segment.
     */
    put_user(*(msg_Ptr++), buffer++);

    length--;
    bytes_read++;
  }

  /*
   * Most read functions return the number of bytes put into the buffer
   */
  return bytes_read;
}
```

Beispiel eines Character-Devices (Forts.)

```
/*
 * Called when a process writes to dev file: echo "hi" > /dev/hello
 */
static ssize_t
device_write(struct file *filp,
             const char *buff, size_t len, loff_t * off)
{
  printk(KERN_ALERT "Sorry, this operation isn't supported.\n");
  return -EINVAL;
}
```

Nach dem Laden muss natürlich noch das Device angelegt werden mit

`mknod /dev/chardev c xxx 0`

(für die Major-No. s. `/var/log/messages`)

Testen des Character-Devices

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>

int main() {
    int fd = open("/dev/chardev",O_RDONLY);
    if (fd<0) {
        printf("Fehler %d, %s\n", errno, strerror(errno));
        return(1);
    }
    char msg[80];
    int size = read(fd, msg, 79);
    if (size>=0) msg[size] = '\0';
    printf("%d\n%s\n",size,msg);
    size = read(fd, msg, 79);
    if (size>=0) msg[size] = '\0';
    printf("%d\n%s\n",size,msg);
    close(fd);
    return(0);
}
```