hochschule mannheim

Betriebssysteme (BS)

Vorlesung im Sommersemester 2024

Prof. Dr. Jens-Matthias Bohli

Fakultät für Informationstechnik

Hochschule Mannheim

March 1, 2024

Betriebssysteme im Sommersemester 2024

Montags 11:30
Vorlesung in S212
Themen:

- Einführung & Linux
- Prozesse und Threads
- Prozess-Scheduling
- Prozesssynchronisation und -kommunikation
- Arbeitsspeicherverwaltung
- Dateisysteme
- Ein-/Ausgabe

Donnerstag 9:45
Praktikum in S117

- Praktikumsaufgaben (Inhalte Prüfungsrelevant)
- Übungsaufgabenheft

Prüfungsleistung: Klausur 90 Min

- Verständnisfragen zu Themen der Vorlesung
- Algorithmische Aufgaben (ähnlich Übungsheft)
- Programmieraufgaben (ähnlich Praktiumsaufgaben)

Kapitel I

Grundlagen

Grundlagen

Motivation

Was ist ein Betriebssystem?

Betriebsarten

Systemaufrufe

Dateien

Motivation der Vorlesung

Verständnis für interne Rechnervorgänge

- Was passiert bei der Programmausführung?
- Welche Fehler können auftreten und wie kann man eingreifen?
- Welche Probleme gibt es bei paralleler Programmausführung?
- Zusammenhänge zwischen virtuellem und physikalischem Speicher?
- Wie sind Dateien auf dem Peripherspeicher organisiert?

Programmierung auf Linux-Systemen

- Kommandointerpreter
- Entwicklungssystem (Editoren, Compiler, Debugger)
- Systemaufrufe

Verbreitete Betriebssysteme

Worldwide device shipments by operating system

Source	Year	Android	iOS/macOS	Windows	Others
Gartner[20]	2015	1.3 billion (54%)	297 million (12.3%) macOS = 21 million	283 million (11.7%)	~520 million (21.6%)
Gartner[21]	2014	48.61%	11.04%	14.0%	26.34%
Gartner[22]	2013	38.51%	10.12%	13.98%	37.41%
Gartner[23]	2012	22.8%	9.6%	15.62%	51.98%

Grundlagen

Definitionen

Definition in Wikipedia

Ein Betriebssystem, auch OS (von englisch operating system) genannt, ist eine Zusammenstellung von Computerprogrammen, die die Systemressourcen eines Computers, wie Arbeitsspeicher, Festplatten, Ein- und Ausgabegeräte verwaltet und diese Anwendungsprogrammen zur Verfügung stellt. Das Betriebssystem bildet dadurch die Schnittstelle zwischen den Hardware-Komponenten und der Anwendungssoftware des Benutzers.

Deitel, Operating Systems

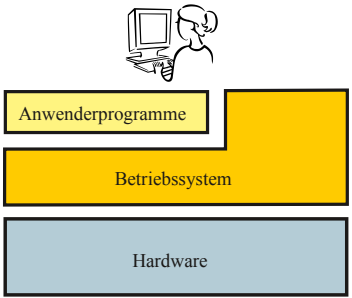
Operating systems are primilary resource managers; they manage hardware - including processors, memory, input/output devices and communication devices. They must also manage applications and other software abstractions that, unlike hardware, are not physical objects.

Rolle des Betriebssystems

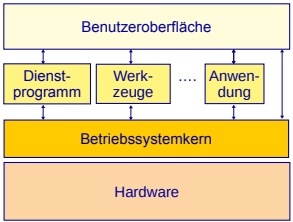
Betriebssystem

Ein Betriebssystem erfüllt also zwei wichtige Aufgaben:

- es bietet dem Anwendungs-Programmierer eine saubere abstrakte Schnittstelle auf die Hardware-Ressourcen.
- es verwaltet die Hardware-Ressourcen.



Betriebssystemaufbau



- Dienstprogramme, Werkzeuge: oft benutzte Programme wie Editor, ...
- Übersetzungsprogramme: Interpreter, Compiler, Translator, ...
- Organisationsprogramme: Speicher-, Prozessor-, Geräte-, Netzverwaltung.
- Benutzerschnittstelle: textuelle und graphische Interaktion mit dem Benutzer.

Kernfunktionalität

- Prozessverwaltung (process management)
  - Verwaltung der in Bearbeitung befindlichen Programme (Prozesse, Tasks)
  - Zuteilung des Prozessors an die rechenbereiten Prozesse.
  - Synchronisation von parallel oder quasiparallel ablaufenden Programmen
- Speicherverwaltung (memory management)
  - Vergabe des Arbeitsspeicherplatzes an die Prozesse
  - Abbildung des logischen Adressraums eines Prozesses auf die zugeteilten Bereiche des physikalischen Speichers.
- Dateiverwaltung (file management)
  - Organisation des Speicherplatzes auf peripheren Speichermedien
  - Funktionen zur Speicherung, Modifikation und Wiedergewinnung von Informationen auf Dateien
- Ein-/Ausgabesteuerung (io system management)
  - Verwaltung der Ein-/Ausgabegeräte und Netzwerk-Interfaces
  - Gerätetreiber mit einfachen Schnittstellen für die vielfältigen Gerätetypen

Erweiterte Funktionalität

- Netzdienste (networking)
  - Basisprotokolle für die Kommunikation in lokalen und öffentlichen Netzen.
  - Protokolle der Anwendungsschicht für den entfernten Zugriff auf Netzwerkressourcen (z.B. File-Server, Http-Server)
- Datensicherung und Datenschutz (protection)
  - Schutz vor Verlust von Daten (Datensicherung)
  - Schutz vor unerlaubten Zugriff auf Daten durch nichtautorisierte Benutzer (Datenschutz)
  - Benutzerschnittstelle (user interface)
- Grafische und alphanumerische Bedienoberflächen
- Kommandosprachen zur Ablaufsteuerung von Prozessen

Grundlagen

Betriebsarten

- Die Art der Nutzung eines Rechnerssystems (Betriebsart) bestimmt in wesentlichem Maße die Anforderungen an das Betriebssystem.
- Klassischerweise wird unterschieden zwischen
  - Batchverarbeitung (Stapelbetrieb),
  - Dialogverarbeitung und
  - Echtzeitverarbeitung
- In qualitativer Hinsicht unterscheiden sich Betriebsarten darin, ob sie Mehrprozessbetrieb unterstützen oder nicht.
- Die historische Entwicklung von Betriebssystemen ist in wesentlichem Maße durch die Anforderungen aus den jeweiligen Betriebsarten geprägt.

Stapelverarbeitung im Einprozessbetrieb

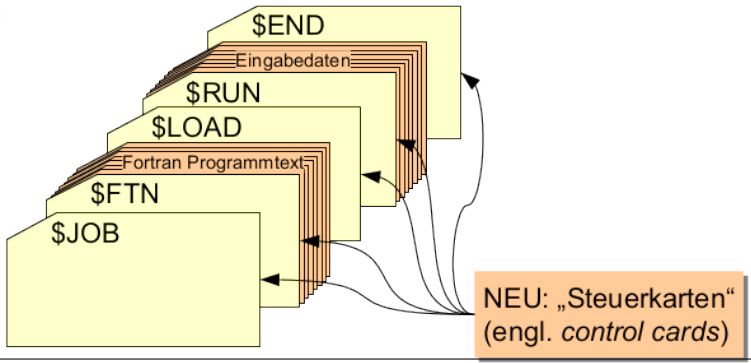
Anfänglich (ab 1945)

- Eingabe unmittelbar von Lochkartenstapel
- Ausgabe auf Drucker
- Rechnerzuteilung erfolgte manuell durch Papierterminkalender.
- Minimale Auslastung der CPU durch großzügige Rechnerzuteilung und langsame E/A-Geräte

IBM CPC (flickr: Seattle Municipal Archives)

Einfache Stapelsysteme ab 1955

- Ziel: Verringerung der manuellen Betriebseingriffe
- Erste Betriebssysteme: *residente Monitore*
  - Interpretation von Steuerbefehlen
  - Laden und Ausführen von Programmen
  - Geräteansteuerung



Einfache Stapelsysteme

Weitere Betriebssystemfunktionalität war nötig für den Fall fehlerhafter Anwendungen:

- Programm terminiert nicht
- Programm überschreibt Speicherbereich des residenten Monitors
- Programm greift direkt auf den Lochkartenleser zu und interpretiert die Steuerkarten als eigene Daten

Lösungen:

- Zeitgeber für Unterbrechungen (interrupts)
- Schutzregister für Speicher des Monitors
- Privilegierter Arbeitsmodus der CPU
  - Deaktivierung des Speicherschutzes
  - Ein-/Ausgabe

E/A Flaschenhals

CPU ist schneller als Kartenleser und Drucker. Der teure Prozessor wird zu selten genutzt
Erster Lösungsansatz: Spooling

- *Simultaneous Peripheral Operations Online*
- Gleichzeitig Rechnen und E/A-Operationen durchführen
- Kopieren der Lochkarten auf Bandlaufwerk und Drucken vom Bandlaufwerk auf separaten Maschinen.

Moderne Lösung: Direct Memory Access

Mehrprozessbetrieb ab 1965

CPU-Auslastung immer noch nicht optimal. Wenn Daten zum Weiterrechnen benötigt werden, muss CPU trotz Spooling warten.

Mehrprozessbetrieb ermöglicht eine effiziente Nutzung des Systems

- Mehrprogrammbetrieb: mehrere Teilnehmer und mehrere Aufgaben am Rechner bzw. Server-Betrieb im Netz
- Parallelbetrieb: unterschiedliche CPU vs.I/O-Nutzung parallel auszuführender Programme

Neue Anforderungen an das Betriebssystem (steigende Komplexität):

- Vollständige Kontrolle der Hardware-Ressourcen
- *Gerechte* Verteilung von CPU-Leistung und Arbeitsspeicherplatz
- Speicherschutz zwischen Benutzer-Prozessen
- Sicherheit und Abrechnung (accounting)

Dialogbetrieb ab 1970

Allgemeine Merkmale

- Der Benutzer hat mit Tastatur, Monitor, Maus unmittelbaren Zugang zum Rechnersystem und kann interaktiv in den Auftragsablauf eingreifen.
- Anzahl der gestarteten Prozesse kann nicht begrenzt werden.
- Betriebsziel sind kurze Antwortzeiten auf interaktiv eingegeben Befehle.

- Die CPU wird nur für ein kurzes Zeitquantum an einen Prozess vergeben.
- Nach Ablauf dieses Zeitquantums wechselt die CPU zum nächsten Prozess (Time-Sharing, Verdrängung laufender Prozesse).

Personal Computer

- Die Entwicklung auf dem Betriebssystemsektor wiederholt sich.
- Erste Betriebssysteme sind vergleichbar mit Batch-Systemen im Einprozessbetrieb (Beispiele: CPM, MS-DOS).
- Die rasch steigende Leistungsfähigkeit der PC's führt zum Einsatz von Mehrprozess-Betriebssystemen (Multitasking-Betriebssystemen) (Beispiele: Windows, UNIX, Linux).
- Aufgrund des Benutzerkreises bestehen erhöhte Anforderungen an die Einfachheit der Bedienung.
- Grafische Benutzeroberflächen tauchen erstmals in PC-Betriebssystemen auf.

Mobile Computing

- Unter Mobile Computing im engeren Sinn versteht man die Datenverarbeitung auf mobilen Endgeräten.
- Mobile Endgeräte können Laptops, Tablet-Computer, Smartphones, sowie kleine, in Gegenstände eingebettete Computer (embedded devices) sein.
- Charakteristisch ist der hohe Grad der Kommunikationsfähigkeit der Systeme über unterschiedliche Medien (WLan, Mobilfunknetz, Bluetooth, NFC) sowie die Einbeziehung von Sensoren und Standortbezug.
- Die Aspekte des Datenschutzes und der Datensicherheit haben eine ganz besondere Relevanz.
- Als Betriebssystem kommen derzeit überwiegend die Systeme iOS (Apple) und Android (Google) zum Einsatz.

Echtzeitbetrieb

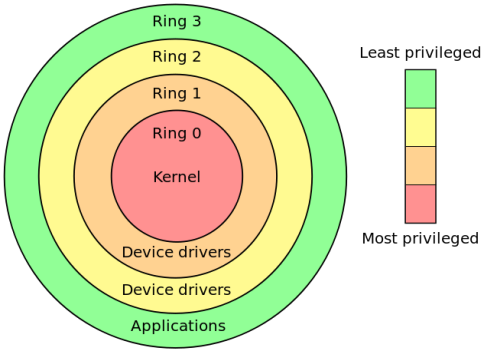
- Echtzeitsysteme nehmen Steuerungsaufgaben in einem dedizierten, meist technischen Umfeld wahr.
- Sie sind zentraler Bestandteil des Gesamtsystems und treten häufig in Form eines eingebetteten Systems (embedded system) auf.
- Auf bestimmte Ereignisse muss das Betriebssystem in einer definierten Zeitspanne reagieren und steuernd in das Prozessgeschehen eingreifen.
- Die Dringlichkeitsstufen einer Reaktion werden auf mehrere Interrupt- Ebenen und Prozessprioritäten abgebildet.
- Ein Ereignis mit kurzer Reaktionszeit löst einen hochprioren Interrupt aus. Aktivitäten mit niederer Dringlichkeit werden daraufhin zurück gestellt.
- Echtzeitbetrieb stellt besondere Anforderungen an das CPU-Scheduling.

Grundlagen

Ausführungsmodi des Prozessors

- Im Mehrprozessbetrieb ist der direkte Zugriff auf die Hardware nicht möglich. Eine Koordinierung durch das Betriebssystem ist notwendig.
- Hierzu stellt das Betriebssystem an der *Systemaufruf-Schnittstelle (system call interface)* die erforderlichen Dienste zur Verfügung.
- Im privilegierten *Betriebssystem-* oder *Kernel-Modus* steht der komplette Befehlssatz der CPU zu Verfügung.
- Benutzerprozesse laufen im unterprivilegierten *Benutzermodus*. Hier stehen weniger Befehle zur Verfügung und der Speicherzugriff ist auf die dem Prozess zugewiesenen Bereiche begrenzt.
- Möchte ein im Benutzermodus laufender Prozess eine Aufgabe erfüllen, die nur im Kernel-Modus möglich ist, muss er über einen *Systemaufruf (system call)* in den Betriebssystemkern wechseln.

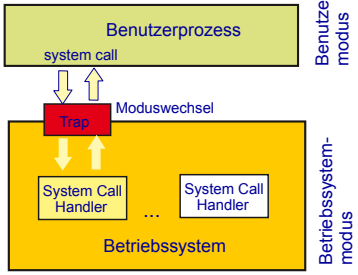
Schema der Ringe beim x86-System



- Linux und Windows nutzen nur User-Mode (Ring 3) und Kernel-Mode (Ring 0)
- Der Arbeitsspeicher ist nach Privilegierungsstufen aufgeteilt, zudem sind einige Assembler-Anweisungen nur in Ring 0 verfügbar (z.B. in, out, cli, lgdt, ltr).

Ablauf eines Systemaufrufs

- Ein Systemaufruf bewirkt einen Sprung in den Betriebssystemkern.
- Die Realisierung erfolgt mittels des Trap-Mechanismus.
- Der Trap-Mechanismus bewirkt einen Moduswechsel in den privilegierten Betriebssystemmodus.
- Der Benutzerprozess läuft unter der Kontrolle von Betriebssystem-Code und hat unbeschränkten Zugriff auf den Speicher und die übrige Hardware.



- Nach Abschluss des Systemaufrufs kehrt der Prozess unter erneutem Moduswechsel zu der unter- brochenen Stelle im gleichen Benutzerprogramm oder in ein anderes Benutzerprogramm zurück.

Unterbrechungen/Interrupts

- Ein analoges Konzept sind Unterbrechungen, die ebenfalls vom Benutzer- in den Systemmodus wechseln.
- Ursache für Unterbrechungen sind Fehlersituationen während der Programmausführung (z.B. Divisoin durch Null) oder externe Ereignisse, die von der Peripherie ausgelöst werden (z.B. die Eingabe eines Zeichens von der Tastatur).
- Die Hardware löst einen Moduswechsel und einen Sprung in die für die Unterbrechungsbehand- lung zuständige Betriebssystem-Routine (Interrrupt-Handler) aus.
- Nach Abschluss der Unter- brechungsbehandlung erfolgt ein Rücksprung zur unterbrochenen Stelle im Benutzerprogramm. Ggf. findet auch ein Prozesswechsel statt.

Systemaufrufe

Systemaufrufe, auch als System Calls oder Supervisor Calls (SVC) bezeichnet, dienen zum Aufruf von Betriebssystemdiensten.

- Ein Systemaufruf ähnelt einem Funktionsaufruf. Die Realisierung ist jedoch abweichend.
- Mit einem Systemaufruf ist ein Wechsel des Prozesses in den privilegierten Betriebssystemmodus verbunden. Bei der Rückkehr aus dem Betriebssystem muss diese Privilegierung wieder rückgängig gemacht werden.
- Realisiert werden Systemaufrufe mittels Software-Interrupts (Traps). Diese werden durch bestimmte Maschineninstruktionen ausgelöst und stehen dem Programmierer auf Assemblerebene zur Verfügung.
- Systemaufrufe enthalten einen Funktionscode sowie meist weitere Parameter. Die Parameterübergabe kann in Registern, in einem Parameterblock oder im Stack erfolgen.

Systemaufrufe in Linux

Sie finden eine Übersicht der Funktionscode der Linux-Systemaufrufe im Header <asm/unistd.h>. Hier für die 32-Bit Architektur:

```
#ifndef _ASM_X86_UNISTD_32_H
#define _ASM_X86_UNISTD_32_H 1

#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5
#define __NR_close 6
#define __NR_waitpid 7
#define __NR_creat 8
#define __NR_link 9
#define __NR_unlink 10
#define __NR_execve 11
#define __NR_chdir 12
#define __NR_time 13
#define __NR_mknod 14
#define __NR_chmod 15
#define __NR_lchown 16
#define __NR_break 17
#define __NR_oldstat 18
#define __NR_lseek 19
#define __NR_getpid 20
#define __NR_mount 21
#define __NR_umount 22
#define __NR_setuid 23
#define __NR_getuid 24
#define __NR_stime 25
#define __NR_ptrace 26
#define __NR_alarm 27
#define __NR_oldfstat 28
#define __NR_pause 29
#define __NR_utime 30
#define __NR_stty 31
```

Beispiel: *Hello World* Assembler-Programm

```
.section .data
hello: .ascii "Hello World!\n"
.section .text
.globl _start
_start:
   mov $4, %eax      # 4 fuer den Syscall 'write'
   mov $1, %ebx      # File Descriptor
   mov $hello, %ecx # Speicheradresse des Textes
   mov $13, %edx     # Laenge des Textes
   int $0x80         # und los
   mov $1, %eax      # das
   mov $0, %ebx      # uebliche
   int $0x80         # beenden
```

Systemaufrufe und die C-Standardbbliothek

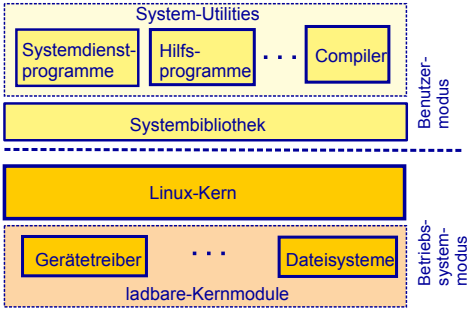
- Für höhere Programmiersprachen existieren Bibliotheksfunktionen, welche die Systemaufrufe in der Syntax der Sprache ermöglichen.
  - Im Falle der Programmiersprache C ist dieses die C-Standard-Bibliothek.
- Eine Teilmenge der Funktionen der C-Standardbibliothek sind so genannte *Wrapper-Funktionen*.
  - Wrapper-Funktionen bereiten den Funktionsaufruf in eine für den Trap-Mechanismus adäquate Form auf und führen die Trap-Instruktion aus.
  - Rückgabewerte des Systemaufrufs werden in einer C-konformen Weise an den Aufrufer der Wrapper-Funktion zurück geliefert.
- Neben den Wrapper-Funktionen enthält die C-Standard-Bibliothek Funktionen, welche die Systemaufrufe „veredeln“ (z.B, printf(), fread()).

Beispiel – Ausgabe

```
#include <unistd.h> // für POSIX-Systemaufrufe
#include <cstdlib>

int main() {
  const char* hello = "Hello World!\n";
  write(1,hello,13);
  exit(0); // kommentar
}
```

Linux und Unix

- Linux ist ein UNIX-kompatibles System
  - Linux und UNIX verfügen über die gleichen Systemaufrufe.
  - Programme aus höheren Programmiersprachen lassen sich durch eine Neukompilierung von einem System auf das andere übertragen.
  - Unix-Tools wurden in Linux reimplementiert. Unix-Benutzer finden die bekannten Kommandos im Linux-System wieder. (GNU Project)
- Die unterschiedlichen Konzepte und Strategien in den Betriebssystemkernen können in Einzelfällen zu unterschiedlichem Laufzeitverhalten führen.



Grundlagen

Dateien

Datei

Eine Datei ist eine benannte Ansammlung von zusammengehöriger Information auf einem Peripherspeicher. Im erweiterten, abstrakten Sinne kann eine Datei auch einen Ein- oder Ausgabestrom von bzw. zu einem Gerät beinhalten.

Dateinamen

- Der Name einer Datei besteht aus einer Folge von Buchstaben, Ziffern und bestimmten Sonderzeichen.
- MS-DOS und Windows unterscheiden nicht zwischen Groß- und Kleinschreibung, in UNIX- und Linux-Systemen ist Groß-/Kleinschreibung signifikant.
- Die Länge des Dateinamens ist systemabhängig.

  MS-DOS 12 Zeichen (mit Extension)

  Windows 255 Zeichen (mit Extension)

  Frühe UNIX-Systeme 14 Zeichen

  Heutige Linux-Systeme 255 Zeichen

Dateiattribute

Neben ihrem Namen hat eine Datei eine Reihe von Attribute wie z.B. die Dateilänge, Angaben zum Eigentümer, Schutzattribute und Zeitangaben.

ls -la

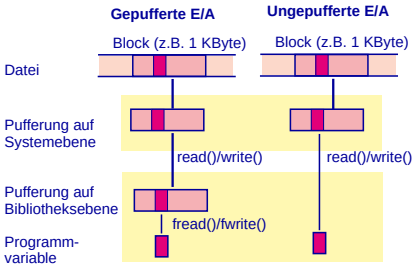
```
drwxr-xr-x  3 bjm users 4096 Okt  8 15:03 .
drwxr-xr-x 22 bjm users 4096 Okt  8 15:01 ..
-rw-r--r--  2 bjm users   12 Okt  8 15:01 data
-rw-r--r--  2 bjm users   12 Okt  8 15:01 data_hardlink
lrwxrwxrwx  1 bjm users    4 Okt  8 15:01 data_softlink -> data
-rwxr-xr-x  1 bjm users 8304 Okt  8 15:02 hello
-rw-r--r--  1 bjm users   57 Okt  8 15:03 hello.c
-rw-r--r--  1 bjm users   12 Okt  8 15:03 .hidden
drwxr-xr-x  2 bjm users 4096 Okt  8 15:01 mnt
||||||||||  | |   |     |    |           |
||||||||||  | |   group |    |           filename
||||||||||  | owner     |    date/time changed
||||||||||  links       length
|||||||||execute others
||||||||write others
|||||||read others
||||||execute group
|||||write group
||||read group
|||execute owner
||write owner
|read owner
type
```

Dateioperationen

- Neben den eigentlichen Schreib- und Leseoperationen existieren Operationen
  - zum Öffnen und Schließen von Dateien
  - zum Positionieren des Schreib-/Lesezeigers
  - zur Abfrage und zur Manipulation der Dateiattribute.
- Die genannten Operationen stehen als Systemaufrufe zur Verfügung. (z.B. read(), write())
- Weiterhin enthalten die Bibliotheken sprachabhängige Funktionsaufrufe, die sich bei ihrer Implementierung auf die Systemaufrufe stützen. (z.B. fprintf(), fscanf())
- Schließlich existieren Kommandos, welche die dateispezifischen System- funktionen auch auf der Kommandoebene zur Verfügung stellen (z.B. cat, chmod).

Gepufferte und ungepufferte Ein-/Ausgabe

- Für das geräteunabhängige E-/A-System von Linux besteht eine Datei aus einer Folge von logischen Blöcken einer festen Länge (z.B. 1024 Bytes).
- Beim Öffnen einer Datei legt die C-Standardbibliothek einen Dateipuffer in Blockgröße im Adressraum des Benutzers an.
- Beim Lesen/Schreiben wird immer ein ganzer Block transferiert.



Unix-Systemaufrufe zur Ein-/Ausgabe (1)

```
int open(char *path, int oflag, int mode);
```

- open() öffnet eine bestehende Datei oder erzeugt und öffnet eine neue Datei. Zurück geliefert wird ein Dateideskriptor (ganze Zahl) für die unter path angegebene Datei.
- path enthält einen gültigen Pfadnamen.
- oflag enthält einen Bitvektor, welcher den gewünschten Zugriffsmodus angibt. Die möglichen Werte (u.a. `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_CREAT`) sind in <fnctl.h> definiert. Falls oflag das Bit `O_CREAT` enthält, und die Datei noch nicht existiert, wird eine neue Datei erzeugt.
- Der optionale Parameter `mode` gibt die Zugriffsrechte für eine neu erzeugte Datei an, z.B. als Oktalzahl 0644 für die Rechte `-rw-r--r--`.

```
int close(int filedes);
```

- Datei filedes wird geschlossen. Der Dateideskriptor wird freigegeben.

Unix-Systemaufrufe zur Ein-/Ausgabe (1)

```
int read(int filedes, char *buf, int nbytes);
int write(int filedes, char *buf, int nbytes);
```

- read() liest ab der aktuellen Schreib-/Leseposition nbytes Bytes aus der Datei filedes und legt diese ab der Adresse buf ab. Die Schreib-/Leseposition wird um nbytes weitergesetzt.
- Die Anzahl der gelesenen Zeichen wird zurückgegeben. Bei Dateiende wird eine 0 zurück gegeben.
- write() überträgt nbytes Bytes aus buf in die Datei filedes. Geschrieben wird ab der aktuellen Schreib-/Leseposition. Die Schreib-/Leseposition wird um nbytes weitergesetzt.
- Zurückgegeben wird die Anzahl der übertragenen Bytes.

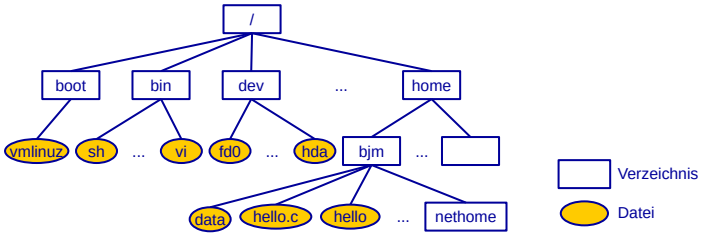
Zugriffsrechte

- Dateien haben als Owner einen Benutzer und eine Gruppe
- Die Berechtigungen zum Lesen, Schreiben und Ausführen (R, W, X) können getrennt für den Benutzer, die Gruppe und andere vergeben werden.
- Der Benutzer, der Owner der Datei ist, darf die Berechtigungen immer ändern.
- Darstellung der Zugriffsrechte als Oktalzahl (führende 0 in C)

Kürzel	Zugriffsrecht	Wert
r	lesen	4
w	schreiben	2
x	ausführen	1
-	keine	0

Also beispielsweise 0644 für `rw-r-r-`

Linux-Dateisysteme



- Der mit "/" bezeichnete Wurzelknoten ist das root-Verzeichnis. Die inneren Knoten stellen Dateiverzeichnisse (directories) mit einer Liste der im Verzeichnis enthaltenen Dateien und Unterverzeichnisse dar. Die Blattknoten enthalten
- gewöhnliche Dateien (ordinary files), welche beliebige Information (z.B. Texte, Binärprogramme, Daten) enthalten
- Spezialdateien (special files), für periphere Geräte oder FIFO-Dateien
- Symbolische Links mit einem Verweis auf einen anderen Knoten.

Pfadnamen und Links

- Die Knoten im Baum werden über Pfadnamen identifiziert.
- absoluter Pfadname ist ein vom Wurzelkoten ausgehender Pfadname. Er beginnt mit dem Zeichen "/" und enthält die Namen aller Unterverzeichnisse und ggf. den abschließenden Dateinamen. Die Namensbestandteile werden durch "/" getrennt. Beispiel: /home/bjm/data
- Ein relativer Pfadename ist ein Pfadname, der vom aktuellen Verzeichnis ausgeht. Beispiel: bjm/data (Annahme: /home ist das aktuelle Verzeichnis)
- Über mehrere Links kann ein Knoten mehrere Vaterknoten besitzen.
- Unix unterscheidet zwischen harten Links (hard link) und symbolischen Links (symbolic link).
- In beiden Fällen besteht die Möglichkeit, ein und denselben Knoten über mehrere Pfadnamen anzusprechen.
- Über mount-Vorgänge können Dateisysteme auf weiteren Datenträgern oder Partitionen in das root-Dateisystem eingehängt werden. (Beispiel: nethome)

Kontroll-Fragen

1. Geben Sie 2 Assemblerinstruktionen an, die in einem Mehrprozessbetriebssystem nur im privilegierten Modus ausgeführt werden können.
2. Warum sind Sie beim Erlernen einer Programmiersprache nicht auf das Thema der Systemaufrufe gestoßen?
3. Nennen Sie einen Grund, warum ein Systemaufruf nicht wie ein einfacher Funktionsaufruf funktionieren kann.
4. Warum lassen sich Programme ohne Veränderung von Unix nach Linux portieren?

Kapitel II

Prozesse

Prozesse

Vom Programm zum Prozess

Definition

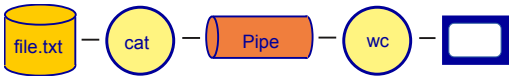
Ein *Prozess* oder eine *Task* ist ein in Ausführung befindliches Programm.

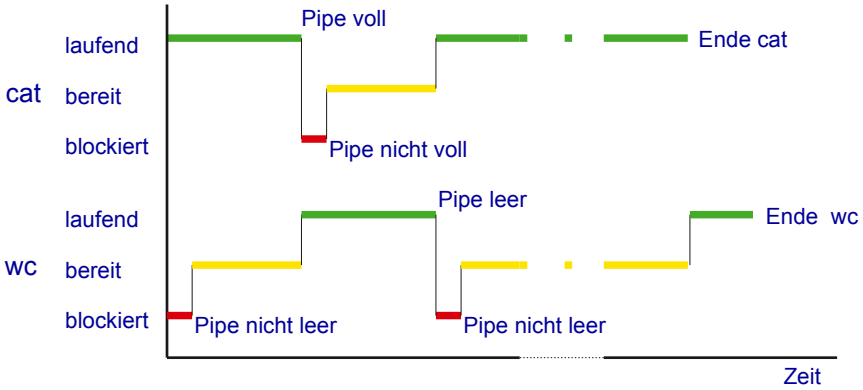
Vom Programm zum Prozess

1. Laden
   - Reservierung von Arbeitsspeicherplatz
   - Laden (eines Teils) des Maschinencodes und Einrichtung von Speicherbereichen für Programmdaten.
   - Einrichtung von systemspezifischen Datenstrukturen zur Verwaltung des Prozesses durch das Betriebssystem.
2. Dispatch (zur Ausführung bringen)
   - Initialisierung der allgemeinen Register und Systemregister (Die Initialisierung des Instruktionsregisters bewirkt einen Sprung in den Code des ausführungsbereiten Programms.)

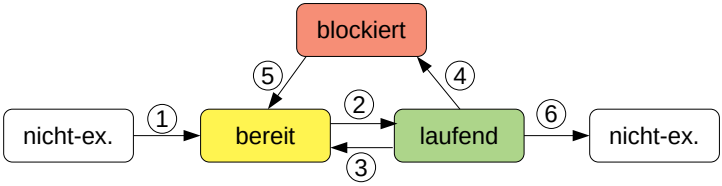
Beispiel Prozesszustände

```
~$ cat file.txt | wc
```





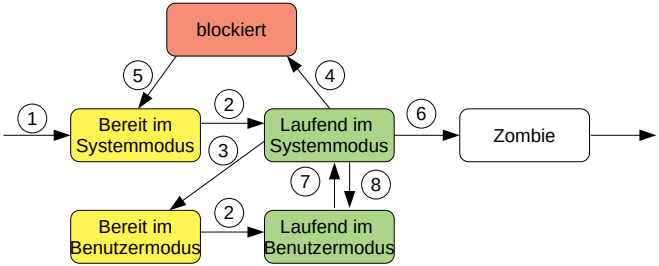
Prozesszustände



1. Prozesserzeugung
2. dispatch: Dem Prozess wird der Prozessor zugeteilt.
3. preempt: Dem Prozess wird der Prozessor entzogen. Er wird verdrängt
4. sleep: Der Prozess versetzt sich selbst in den blockiert-Zustand, weil die weitere Ausführung nicht möglich ist.
5. wakeup: Nach Eintreffen des Ereignisses wird der Prozess geweckt und in den bereit-Zustand überführt.
6. Der Prozess terminiert

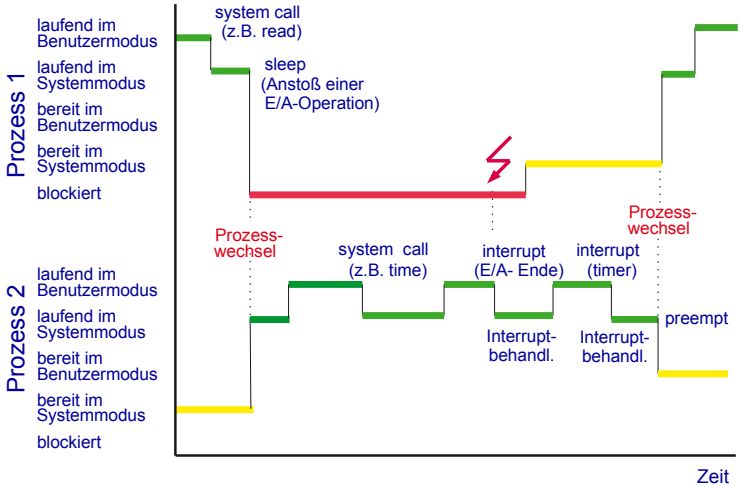
Prozesszustände in Linux/Unix

- Laufend im Benutzermodus: Der Prozess läuft unter Kontrolle von Benutzercode.
- Laufend im Systemmodus: Der Prozess läuft unter Kontrolle von BS-Code.
- Bereit im Systemmodus: Der Prozess wurde aus dem blockiert-Zustand geweckt.
- Bereit im Benutzermodus: Dem Prozess wurde der Prozessor entzogen.
- Zombie: Der Prozess wurde beendet, aber konnte seinen Exit-Status noch nicht übermitteln.



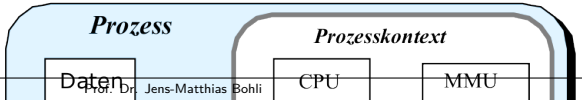
7. system call/interrupt:
   Der Prozess tritt in den Systemmodus ein.
8. return:
   Der Prozess verlässt den Systemmodus.

Beispiel Prozesszustände Unix/Linux



Prozesskontext

- Die Ablaufumgebung eines Prozesses heißt Prozesskontext. Er besteht aus dem Hardware-Kontext, dem Benutzerkontext und dem Systemkontext.
- Die Inhalte der allgemeinen Register und der prozessspezifischen Systemregister bilden den Hardware-Kontext.
- Der Benutzerkontext besteht aus den Speicherbereichen eines Prozesses Diese enthalten den Benutzercode und die Benutzerdaten.
- Der Systemkontext (Prozesskontrollblock) enthält prozessspezifische Kontroll- und Statusinformation zur Verwaltung des Prozesses durch das Betriebssystem.



System-Kontext

Typische Bestadteile des Systemkontextes

- die Prozessidentifikation (PID)
- den Prozesszustand
- Scheduling-Information (z.B. Priorität)
- den gesicherten Hardwarekontext
- Angaben zu den geöffneten Dateien
- Angaben zum Adressraum

Verwaltung der Prozesse in Listen

- Systemkontexte aller existierenden Prozesse werden in der Prozessliste verwaltet.
- Die ausführungsbereiten Prozesse befinden sich zusätzlich in der Run-List oder in der Ready-Queue.
- Wartenden Prozesse sind in eine ereignisspezifische Warteliste eingekettet.
- Der Prozesskontext des laufenden Prozesses wird über einen Pointer im Betriebssystemkern adressiert.

Kontextwechsel

Der Wechsel des Prozessors von einem Prozess zu einem anderen bezeichnet man als *Kontextwechsel*. Auslösende Ereignisse

- Ein Prozess beendet sich oder geht in den blockiert-Zustand über.
- Der laufende Prozess wird abgebrochen.
- Ein Interrupt überführt einen hochprioren Prozess in den bereit-Zustand.
- Nach einem Timer-Interrupt wird der laufende Prozess wegen Überschreitung seines Zeitquantums verdrängt.

Aktionen

1. Registerinhalte des alten Prozesses werden in einem Bereich des Systemkontextes gesichert. Die Register werden aus dem gesicherten Hardware-Kontext des neuen Prozesses geladen.
2. Diverse Informationen (Zustand, Wartebedingung) in den Systemkontexten der betroffenen Prozesse werden aktualisiert.

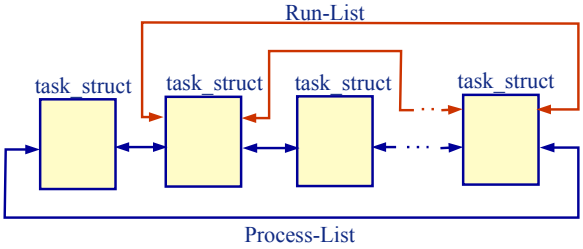
Prozessdeskriptor in Linux

```
struct task_struct {
  volatile long state;
  struct thread_info *thread_info;
  unsigned long flags;

  int prio, static_prio;
  struct list_head run_list;
  prio_array_t *array;
  unsigned long sleep_avg;
  unsigned long policy;
  unsigned int time_slice;

  struct list_head tasks;
  struct mm_struct *mm;
  pid_t pid, tgid;

  struct task_struct *parent;
  struct list_head children;
  struct list_head sibling;
```

```
  /* Fortsetzung */

  unsigned long rt_priority;
  cputime_t utime, stime;

  uid_t uid, euid;
  gid_t gid, egid;

  struct thread_struct thread;

  struct fs_struct *fs;

  struct files_struct *files;

  struct sighand_struct *sighand;

  /* ... */
};
```

Datenstrukturen zur Prozessverwaltung

- Die Prozessdeskriptoren werden in der Process-List verwaltet (verkettet über die `list_head`-Struktur tasks).
- Die Prozesse in den Zuständen „laufend“ und „bereit“ sind in der Run-List enthalten (verkettet über die `list_head`-Struktur `run_list`).
- Wartende Prozesse sind in einer von mehreren Wait-Queues eingekettet

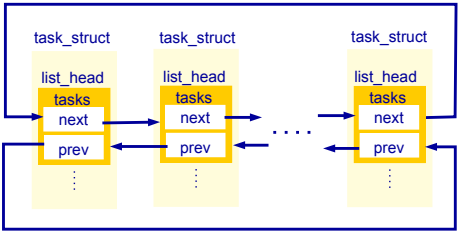


Verkettete Listen im Linux-Kernel

- Verketteten Listen sind unabhängig von Typ der Listenelemente mit Hilfe des rekursiven Strukturtyps `list_head` realisiert.

```
struct list_head {
   struct list_head *next, *prev;
};
```

- Die `list_head`-Strukturen sind als Unterstrukturen in den zu verkettenden Listenelementen enthalten.
- Der Linux-Kernel enthält eine Reihe von elementaren Funktionen und Markros zur Manipulation und zum Durchquerung von verketteten Listen.



Prozessarten

Benutzerprozesse

Benutzerprozesse sind mit einem Terminal verbunden und operieren unter der Kontrolle eines Benutzers.
Beispiele: bash, ps

Dämonenprozesse

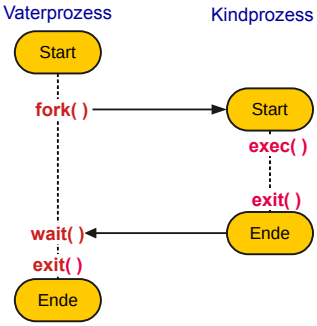
Die ebenfalls auf Benutzerebene ablaufenden Dämonenprozesse sind nicht mit einem Terminal verbunden und führen bestimmte Systemfunktionen aus.
Beispiele: init, cron, inetd

Systemprozesse

Systemprozesse, unter Linux als Kernel-Threads bezeichnet, verbleiben immer im Systemmodus. Sie operieren im residenten Speicherbereich des Kerns und benötigen keinen virtuellen Adressraum.
Beispiele: kswapd, bdflush

Prozesse

Prozessduplikation



- Der Systemaufruf **fork()** dupliziert den aufrufenden Prozess, indem er eine Kopie seines Prozesskontexts erzeugt.
- Der Vaterprozess kann mittels **wait()** auf die Beendigung eines Sohnprozesses warten. Hierbei wird ihm die Prozessnummer des beendeten Sohnprozesses und dessen Exit-Status mitgeteilt.
- Mit dem **exec()**-Systemaufrufs tauscht der Sohnprozess den Programmcode und die aktuellen Programmdaten durch die eines anderen Programms aus.
- Nach erfolgreicher Programmausführung oder im Fehlerfall beendet sich der Sohn- prozess unter Angabe eines Exit-Codes mit einem **exit()**-Systemaufruf.

fork()-Systemaufruf

Informieren Sie sich über den fork()-Systemaufruf:

```
man 2 fork
```

```
#include <unistd.h>
pid_t fork(void);
```

- fork() erzeugt einen neuen Prozess (Sohnprozess). Dieser ist eine exakte Kopie des aufrufenden Prozesses (Vaterprozess).
- Der Sohnprozess erbt vom Vaterprozess u.a. die Daten- und Codebereiche, den Zugriff auf die zum Zeitpunkt des Aufrufs geöffnete Dateien und die Priorität
- Unterschiede bestehen in der Prozessnummer (PID) und der Prozessnummer des Vaterprozesses (PPID)
- Rückgabewerte:
  - an den Sohnprozess: 0
  - an den Vaterprozess: im Erfolgsfall PID des Sohnprozesses, -1 sonst

wait()-Systemaufruf

```
man 2 wait
```

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *wstatus);

pid_t waitpid(pid_t pid, int *wstatus, int options);
```

- Der aufrufende Prozess wird blockiert bis einer seiner Sohnprozesse terminiert.
- `wstatus` enthält eine Adresse zur Ablage des vom beendeten Sohn übermittelten Exit-Status.
- Rückgabewert: Prozessnummer des beendeten Sohnprozesses

exec()-Systemaufruf

Bei dem Systemaufruf exec() handelt es sich um eine Familie von Systemaufrufen, die sich lediglich in der Parametrierung unterscheiden. Der eigentliche System-Call ist `int execve()`

```
man 3 exec
```

```
#include <unistd.h>
int execl(const char *path, const char *arg, ..., (char  *) NULL);
```

- execl() überlagert die Code- und Datenbereiche des aufrufenden Prozesses mit denen einer neuen Programmdatei. Der Systemkontext bleibt weitgehend unverändert. Der Zugriff zu den geöffneten Dateien bleibt erhalten.
- path enthält den Pfadnamen der neuen Programmdatei
- arg0 enthält den Dateinamen (letzter Bestandteil des Pfadnamens) der neuen Programmdatei
- arg1,...,argn enthalten mögliche Programmargumente
- Die variabel lange Argumentliste wird mit einem NULL-Zeiger abgeschlossen.
- Rückgabewert: 0, falls fehlerfrei, -1 sonst.

_exit()-Systemaufruf

Systemaufruf

```
#include <unistd.h>
void _exit(int status);
```

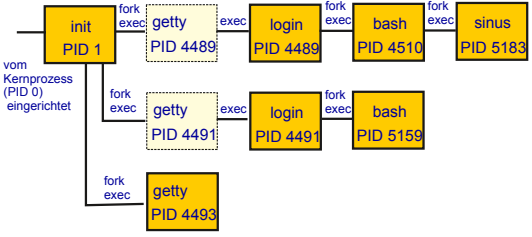
Bibliotheksfunktion

```
#include <stdlib.h>
void exit(int status);
```

- `_exit()` terminiert den aufrufenden Prozess. Alle offenen Dateien werden geschlossen.
- Über dem Parameter status kann der Exit-Status an den Vaterprozess übermittelt werden (im Erfolgsfall der Wert 0, im Fehlerfall ein Wert ungleich 0).
- `exit()` beendet alle Threads im Prozess und leert die stdio-Puffer.

Prozesshierarchie in Lnux

- Die Wurzel aller Prozesse ist ein Kernel-Thread mit PID 0 (Kernprozess).
- Der Kernprozess initialisiert eine Reihe von Datenstrukturen des Kerns, erzeugt einen weiteren Kernel-Thread mit Prozessnummer 1 und übernimmt danach die Funktion des Idle Task.
- Der Kernel-Thread mit PID 1 führt einen exec()-Systemaufruf aus und lädt das Programm init. Er wird so zu einem Dämonen-Prozess namens init.
- init ist für das weitere Hochfahren des Systems und für die Erzeugung aller weiteren Prozesse verantwortlich.



Ausnahmesituationen

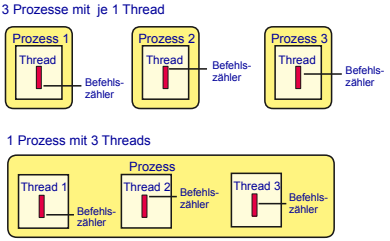
- Vaterprozess endet vor Sohnprozess
  - Falls der Vaterprozess ohne einen wait()-Aufrufvor dem Sohnprozess endet, wird der Sohnprozess von einem Systemprozess namens init “adoptiert”.
- Sohnprozess endet vor wait() des Vaterprozesses
  - Ein Sohnprozess, welcher endet, bevor der Vaterprozess einen wait()-Aufruf tätigen konnte, wird zu einem so genannten Zombie.
  - Alle geöffneten Dateien werden geschlossen, alle belegten Speicherbereiche werden freigegeben. Der Prozess verbleibt noch solange in der Prozessliste, bis der Vater-Prozess einen wait()-Aufruf tätigt oder seinerseits endet.
  - Im Falle von Linux enthält die Variable state im Prozessdeskriptor den Wert ZOMBIE.

ps-Kommando

```
F   UID   PID  PPID PRI  NI  VSZ  RSS WCHAN STAT TTY  TIME COMMAND
4     0     1     0  16   0  692  260 -       S  ?    0:00 init
.....................
1     0   420     5  20   0    0    0 pdflus  S  ?    0:00 pdflush
1     0   422     1  25   0    0    0 kswapd  S  ?    0:00 kswapd0
.................
5 65534  4396     1  25   0 1536  480 -       S  ?    0:00 portmap
5     4  4503     1  16   0 6724 3264 -       S  ?    0:00 cupsd
1     0  4611     1  16   0 1772  748 -       S  ?    0:00 cron
5     0  5106     1  25   0 4800 1912 -       S  ?    0:00 sshd
4     0  5148     1  16   0 2792 1216 wait    S  ?    0:00 login
4     0  5149     1  17   0 2792 1232 wait    S  ?    0:00 login
4     0  5150     1  17   0 2792 1216 wait    S  ?    0:00 login
4     0  5151     1  18   0 1928  604 -       S  tty4 0:00 getty
4     0  5152     1  17   0 1932  608 -       S  tty5 0:00 getty
4     0  5153     1  18   0 1932  604 -       S  tty6 0:00 getty
4  1000  5396  5148  16   0 4392 1872 wait    S  tty1 0:00 bash
4  1000  5422  5149  15   0 4516 1912 wait    S  tty2 0:00 bash
4  1000  5447  5150  16   0 4392 1872 wait    S  tty3 0:00 bash
0  1000  5472  5447  16   0 1344  252 -       S  tty3 0:00 ea
0  1000  5477  5422  25   0 1340  240 -       R  tty2 0:30 cpu
0  1000  5479  5396  17   0 2540  764 -       R  tty1 0:00 ps
```

Prozesse

Threads

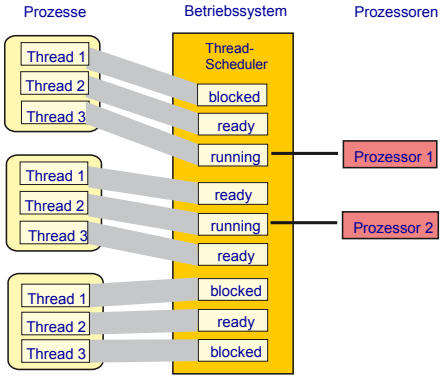


- Ein *Thread* ist ein eigenständiger "Ausführungsfaden" innerhalb eines Prozesses.
- Er verfügt über einen eigenen Programmzähler, einen eigenen Stack sowie einem Bereich zur Ablage des Hardware-Kontextes.
- Code- und Datenbereiche sowie die geöffneten Dateien werden von den innerhalb des Prozesskontexts ablaufenden Threads gemeinsam genutzt.
- Ein Thread benötigt weniger Kontextinformation als ein Prozess.
- Kontextwechsel zwischen Threads sind weniger aufwendig ist, als Kontext- wechsel zwischen Prozessen (leichtgewichtige Prozesse).

Threads auf Benutzerebene

Die Thread-Verwaltung, das Thread-Scheduling sowie die Kontextwechsel zwischen Threads werden von einer Bibliothek wahrgenommen.

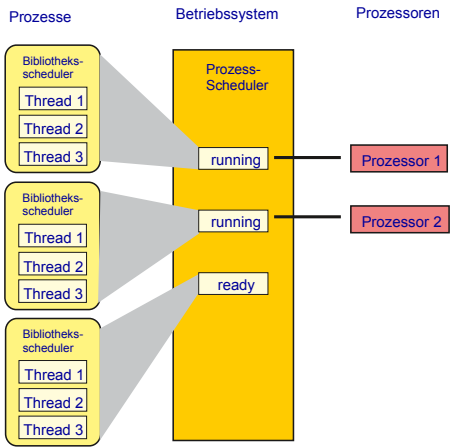
- Vorteile:
  - keine Belastung des BS
- Nachteile:
  - Blockade des gesamten Prozesses durch einen blockierten Thread
  - unflexibles Scheduling
  - keine echte Parallelität



Threads auf Betriebssystemebene

Das Betriebssystem verwaltet Threads (Kernel-Threads). Scheduling-Einheiten sind Threads (nicht Prozesse).

- Vorteile:
  - keine globale Blockade
  - faires Scheduling
  - echte Parallelität
- Nachteile:
  - Belastung des BS



Threads in Linux

- Bei der Implementierung unterscheidet Linux nicht zwischen Threads und Prozessen.
- Ein Thread ist in der Prozessliste durch einen Task-Deskriptor repräsentiert und nimmt wie Prozesse am Scheduling teil.
- Die zu einem Prozess gehörigen Threads teilen sich den virtuellen Adressraum und möglicherweise weitere Ressourcen.
- Zur Erzeugung von Threads verfügt Linux über den nicht POSIX-konformen Systemaufruf clone().

```
/* Prototype for the glibc wrapper function */
#include <sched.h>
int clone(int (*fn)(void *), void *child_stack, int flags,
    void *arg);
```

Threads in C: Pthreads

- Von der Posix-Gruppe wurde mit den Posix-Threads (kurz Pthreads) eine standardisierte Thread-Bibliothek vorgeschlagen
- Der Vorschlag spezifiziert ein API, bestehend aus einer Reihe von C-Datentypen und C-Funktionen, welche in der Header-Datei <pthread.h> deklariert sind.
- Abhängig von den Möglichkeiten des jeweiligen Betriebssystems sind die Posix- Threads auf Benutzerebene oder als Kernel-Threads implementiert.
- Pthread-Implementierungen existieren für Windows, Linux und Unix-Derivate
- Auswahl der wichtigsten Funktionen
  - `pthread_create()` zum Erzeugen eines Threads (vergl. fork())
  - `pthread_exit()` zum Beenden eines Threads (vergl exit()) und
  - `pthread_join()` zum Warten auf das Ende eines Threads (vergl. wait())

Threads in C++: <thread>

C++ bietet seit C++11 Multi-Threading in der Standardbibliothek im Header <thread>.

- Eine Instanz der Klasse `thread` repräsentiert einen Thread.
- Der Konstruktor erzeugt den Threads, der eine globale Funktion, eine Klassenmethode, ein Funktionsobjekt oder eine Lambda-Funktion ausführen kann.
- Die auszuführende Threadfunktion kann beliebig viele Parameter beliebigen Typs haben, die immer implizit kopiert werden, um eine ausreichend lange Lebensdauer zu gewährleisten.
- Der Destruktor der Thread-Klasse prüft, ob der zugehörige Laufzeitthread noch läuft, und wirft gegebenenfalls eine Exception, daher sollte
  - mithilfe der Methode `join()` auf das Ende des Threads gewartet werden, oder
  - der Laufzeitthread vom C++ Threadobjekt mithilfe der Methode detach() entkoppelt werden.
- Für die Rückgabe eines Wertes muss ein Objekt vom Typ `future<T>` oder die Abstaktion mittesl `async()` verwendet werden (siehe Beispiele).
- Einige Funktionen der pthread-Bibliothek, z.B. Threads mit unterschiedlichen Prioritäten, werden von der C++-Bibliothek nicht unterstützt.

Beispiel – C++-Thread und pthreads

```c
#include <pthread.h>
/* ... */
Buffer buf;
int main() {
  pthread_t tid;
  /* ... */
  printf("Enter two integer arguments: ");
  scanf("%d %d", &buf.a, &buf.b);
  pthread_create( &tid, NULL, add, &buf);
  pthread_join(tid,(void*)&erg );
  /* ... */
}

void *add(void *arg) {
  Buffer *buf = (Buffer*)arg;
  printf("%d", pthread_self());
  buf->erg = buf->a + buf->b;
}
```

```c
#include <thread>
/* ... */
Buffer buf;
int main() {
  thread t1;
  /* ... */
  cout << "Enter two integer arguments: ";
  cin >> buf.a >> buf.b;
  t1 = thread(add, &buf);
  t1.join();
  /* ... */
}

void add(Buffer* buf) {
  cout << this_thread::get_id();
  buf->erg = buf->a + buf->b;
}
```

Kontroll-Fragen

1. Welches sind die Auswirkungen, wenn die Kapazität der Pipe in Beispiel 2.2
   - unendlich groß ist.
   - nur einige Bytes beträgt.
2. Geben Sie je ein Ereignis an, welches die nachfolgend aufgeführten Zustandsübergänge auslöst.
   - laufend nach blockiert
   - blockiert nach bereit
   - laufend nach bereit
3. Warum lässt sich ein Prozesswechsel nicht vollständig durch die Hardware abwickeln?
4. Warum erfährt der login-Prozess im Linux-System davon, wenn die Shell mit einem exit-Kommando beendet wird?

Kontroll-Fragen

5. In Linux können Kommandos als Hintergrundprozesse abgewickelt werden; d.h. nach Eingabe eines Kommandos kann sofort ein weiteres Kommando eingegeben werden, ohne dass auf die Beendigung des ersten Kommandos gewartet werden muss. Beschreiben Sie die hierfür notwendige Modifikation an der in 2.6.1 beschriebenen Kommandointerpretation durch die Shell.
6. Warum können Vaterprozess und Sohnprozess unter Linux nicht über gemeinsame globale Variablen kommunizieren?
7. Was ist der Unterschied zwischen einem Dämonen-Prozess und einem Kernel-Thread?
8. Worin unterscheiden sich Threads in Linux von typischen Kernel-Threads?

Kapitel III

Scheduling

Scheduling

Ziele beim Prozess-Scheduling

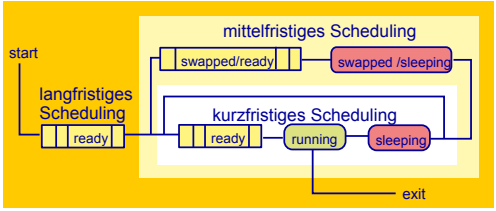
Nicht-präemptives Scheduling

Präemptives Scheduling

Scheduling in Linux

Prozess-Scheduling

- Scheduling beinhaltet allgemein die strategische Reihenfolgeplanung bei der Belegung eines meist knappen Betriebsmittels.
  - Scheduler: verantwortlich für die strategische Zuteilung des Prozessors.
  - Dispatcher: zuständig für die exekutive Durchführung von Prozesswechseln.
- Betriebssysteme treffen CPU-Zuteilungsentscheidungen auf drei Ebenen:
  - langfristiges Scheduling: Zulassung von Prozessen zum System
  - mittelfristiges Scheduling: Aus- und Einlagerung von Prozessen
  - kurzfristiges Scheduling: kurzfristige CPU-Zuteilung



- Hier betrachteten wir Verfahren für das *kurzfristige Scheduling*.

Scheduling-Ziele

- Benutzerorientiert
  - *Minimale Verweilzeit* – In Batch-Systemen ist die Zeit von der Eingabe des bis zur Fertigstellung eines Auftrags (Verweilzeit)zu minimieren.
  - *Minimale Antwortzeit* – In Dialogsystemen ist die Zeitspanne zwischen der Eingabe eines Dialogkommandos und der ersten Reaktion auf das Kommando (Antwortzeit) zu minimieren.
  - *Garantierte Reaktionszeit* – In Echtzeitsystemen ist die Zeitspanne, innerhalb der das Rechnersystem auf ein externes Ereignis reagieren muss (Reaktionszeit) zu garantieren.
- Systemorientiert
  - *Maximale CPU-Auslastung* – Das teure Betriebsmittel Prozessor soll möglichst gut ausgelastet werden.
  - *Fairness* – Jeder Prozess soll einen gerechten Anteil von der verfügbaren Prozessorzeit erhalten.
  - *Lastausgleich* – Auch E/A Geräte sollen gleichmäßig ausgelastet sein.

Scheduling

First Come First Served (FCFS)

- Bei First Come First Served (FCFS) werden die Prozesse in der Reihenfolge ihres Eintreffens in der Eingangswarteschlange bearbeitet.
- Ein einmal eingelagerter Prozess verbleibt danach bis zu seiner Terminierung im System.
- Das Verfahren minimiert die Zahl der Kontextwechsel und optimiert die CPU-Auslastung.
- Nachteil: Der *Konvoi-Effekt* verursacht hohe Antwortzeit und niedrigen E/A-Durchsatz.

Job	Eintrittszeit	Rechenzeit
J1	0	4
J2	2	20
J3	3	7

J1	J2	J3

Mittlere Verweilzeit: $(4+22+28)/3 = 18$

Shortest Job first (SJF)

- Unter den auf einen Platz im System wartenden Prozessen/Jobs wird der mit der minimal erforderlichen Rechenzeit auswählt.
- Das Verfahren ist bzgl. der mittleren Verweilzeit optimal.
- Praktisch schwer umzusetzen: Voraussetzung für die Anwendung der Verfahrens ist die Kenntnis über die zu erwartenden Bearbeitungszeit der Prozesse/Jobs.
- Nachteil: *Verhungern* Rechenintensiver Prozesse

Job	Eintrittszeit	Rechenzeit
J1	0	4
J2	2	20
J3	3	7

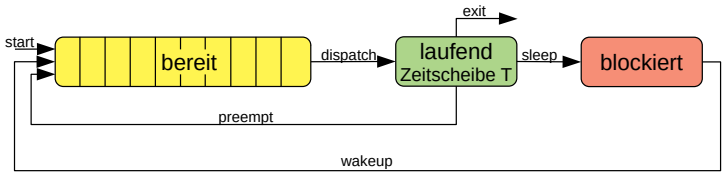
J1	J3	J2

Mittlere Verweilzeit: $(4+8+29)/3 = 13{,}666...$

Scheduling

Round Robin

- Jedem Prozess wird für ein bestimmtes Zeitintervall (Zeitquantum, Zeitscheibe) der Prozessor zugewiesen.
- Nach Ablauf der Zeitscheibe wird dem Prozess der Prozessor entzogen.
  - Der Prozess reiht sich erneut am Ende der *ready*-Warteschlange ein.
  - Der nächste Prozess wird gemäß FCFS aus der Warteschlange der bereiten Prozesse ausgewählt.
- Vorrausetzung für den Einsatz des Verfahrens ist ein Zeitgeber.
- Falls die Zeitscheibe nicht aufgebraucht wird, sondern der Prozess beispielsweise durch I/O blockiert, findet ein vorzeitiger Prozess-wechsel statt.
- Was passiert mit Prozessen, die häufig vor Ende ihrer Zeitscheibe blockieren?



Leistungsprobleme bei RR

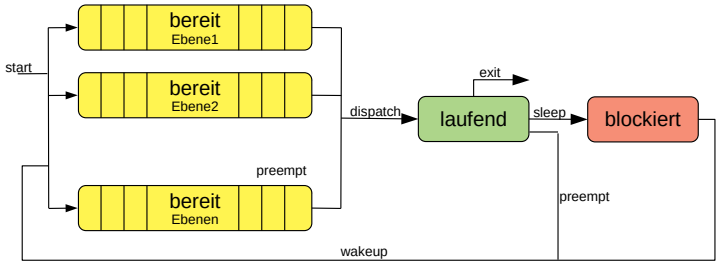
Leistungsprobleme bei RR

- E/A-lastige Prozesse beenden ihre kurze CPU-Phase innerhalb ihrer Zeitscheibe. Sie blockieren und kommen erst am Ende ihrer E/A-Phase in die Bereit-Warteschlange.
- CPU-lastige Prozesse schöpfen dagegen ihre Zeitscheibe voll aus und werden direkt wieder in die Warteschlange eingefügt.
- E/A-lastige Prozesse werden schlecht bedient und dadurch E/A-Geräte schlecht ausgelastet.
- die Antwortzeit E/A-lastiger Prozesse erhöht sich.

Idee: Bevorzugung von Prozessen, die ihre Zeitscheibe nicht ausgenutzt haben.

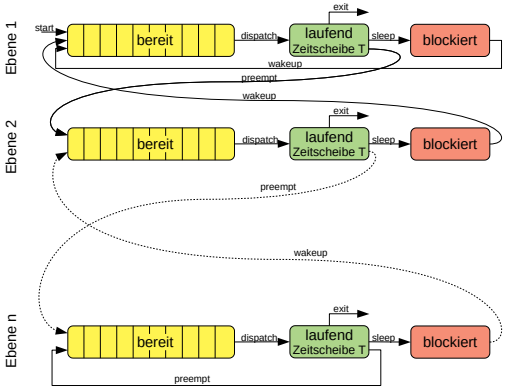
Prioritäts-basierte Verfahren

- Den Prozessen werden beim Start (von außen) feste Prioritäten zugeordnet. Diese Prioritäten orientieren sich an den jeweils geforderten Reaktionszeiten.
- Die Prozessorzuteilung erfolgt jeweils an den ersten Prozess aus der höchst-prioren, nichtleeren Warteschlange.
- Ein Prozess wird verdrängt, wenn ein Prozess mit höherer Priorität rechenbereit wird.



Dialog-Systeme: Multilevel-Feedback

- Nach Verbrauch der gesamten Zeitscheibe wird der Prozess in die Warteschlange der nächst tieferen Prioritätsebene verdrängt.
- Aus tieferen Prioritätsebenen kann er durch vorzeitige Abgabe des Prozessors wieder in die nächst höhere Prioritätsebene aufsteigen.
- Mehrere Prioritätsebenen mit einer eigenen *bereit*-Warteschlange.
- Zuteilung des Prozessors an den ersten Prozess aus der höchstprioren, nichtleeren Warteschlange.



Scheduling

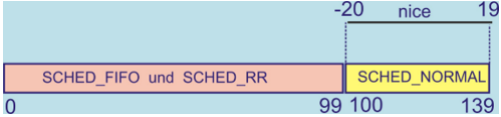
Scheduling-Klassen

Bezüglich des Scheduling unterscheidet Linux zwischen 3 Klassen von Prozessen:

- Real-Time FIFO (policy im Task-Deskriptor hat den Wert `SCHED_FIFO`).
- Real-Time Round Robin (policy im im Task-Deskriptor hat den Wert `SCHED_RR`)
- Timesharing (policy im Task-Deskriptor hat den Wert `SCHED_OTHER`)

Scheduling in Linux – Prozessprioritäten

- Die Prozessprioritäten reichen bei Linux von 0 bis 139.
- Niedrige Prioritätswerte bedeuten dabei eine hohe Dringlichkeit.



- Prioritäten für Echtzeitprozesse sind fest zugeordnet (externe Prioritäten).
- Die Prioritäten für normale Prozesse sind dynamische Prioritäten.
- Die über den nice-Wert festgelegte, statische Priorität wird, abhängig von der CPU-Nutzung, um maximal 5 Punkte inkrementiert bzw. dekrementiert.
- Maßgeblich hierfür ist die Variable `sleep_avg`, welche die Ausführungs- und Wartezeiten der Vergangenheit akkumuliert.
- Der Minimalwert von 100 und der Maximalwert von 139 werden durch die Korrektur nicht unter- bzw. überschritten.

nice()

```
int nice(int incr);
```

- Zulässige Werte für incr liegen zwischen -20 und 19.
- Nur Prozesse des root-Benutzers dürfen negative nice-Werte verwenden.
- Die statische Priorität ist in der Variablen `static_prio` des Prozessdeskriptors hinterlegt und hat standardmäßig den Wert 120.
- Dieser Standardwert kann durch einen nice()-Systemaufruf verändert werden.

Dynamische Priorität

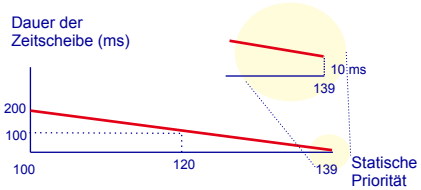
- Die statische Priorität für Nichtechtzeitprozesse liegt zwischen 100 und 139.
- Die für das Scheduling relevante dynamische Priorität ergibt sich durch Addition eines Korrekturwertes, der sich zwischen -5 (Bonus) und +5 (Malus) bewegt.
- Der Korrekturwert berechnet sich aus der Variablen `sleep_avg` im Prozessdeskriptor. `sleep_avg` enthält eine Maßzahl für die Interaktivität eines Prozesses, basierend auf seinen Ausführungs- und Wartezeiten in der Vergangenheit.
- Bei der Korrektur des Prioritätswertes werden der Minimalwert von 100 und der Maximalwert von 139 nicht unter- bzw. überschritten.
- Die dynamische Priorität wird in der Variablen prio des Prozessdeskriptors abgelegt.

ps-Kommando

```
F   UID   PID  PPID PRI  NI  VSZ  RSS WCHAN STAT TTY  TIME COMMAND
4     0     1     0  16   0  692  260 -       S  ?    0:00 init
.....................
1     0   420     5  20   0    0    0 pdflus  S  ?    0:00 pdflush
1     0   422     1  25   0    0    0 kswapd  S  ?    0:00 kswapd0
.................
5 65534  4396     1  25   0 1536  480 -       S  ?    0:00 portmap
5     4  4503     1  16   0 6724 3264 -       S  ?    0:00 cupsd
1     0  4611     1  16   0 1772  748 -       S  ?    0:00 cron
5     0  5106     1  25   0 4800 1912 -       S  ?    0:00 sshd
4     0  5148     1  16   0 2792 1216 wait    S  ?    0:00 login
4     0  5149     1  17   0 2792 1232 wait    S  ?    0:00 login
4     0  5150     1  17   0 2792 1216 wait    S  ?    0:00 login
4     0  5151     1  18   0 1928  604 -       S  tty4 0:00 mingetty
4     0  5152     1  17   0 1932  608 -       S  tty5 0:00 mingetty
4     0  5153     1  18   0 1932  604 -       S  tty6 0:00 mingetty
4  1000  5396  5148  16   0 4392 1872 wait    S  tty1 0:00 bash
4  1000  5422  5149  15   0 4516 1912 wait    S  tty2 0:00 bash
4  1000  5447  5150  16   0 4392 1872 wait    S  tty3 0:00 bash
0  1000  5472  5447  16   0 1344  252 -       S  tty3 0:00 ea
0  1000  5477  5422  25   0 1340  240 -       R  tty2 0:30 cpu
0  1000  5479  5396  17   0 2540  764 -       R  tty1 0:00 ps
```

Zeitscheibe

- In Linux variiert die Dauer der Zeitscheibe, abhängig von der statischen Priorität des Prozesses.

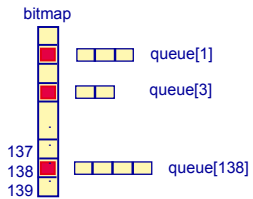


- Die Zeitscheibe ist nicht auf eine Rechenphase begrenzt. Falls ein Prozess verdrängt wird oder den Prozessor vorzeitig abgibt, behält er das Restquantum für spätere Rechenphasen.
- Wenn die Zeitscheibe aufgebraucht ist, wird der Prozess verdrängt und so lange nicht mehr an der Vergabe der CPU beteiligt, bis auch alle anderen rechen-bereiten Prozesse ihre Zeitscheibe aufgebraucht haben.
- In diesem Falle erhalten alle Prozesse eine neue Zeitscheibe zugeteilt.

Implementierung (1)

- Linux verfügt über einen so genannten O(1)-Scheduler; d.h. die Laufzeit von Scheduling-Entscheidugen ist unabhängig von der Anzahl der rechenbereiten Prozesse.
- Die Run-Queue enthält zwei Priority Arrays.
- Im aktiven Array (active array) werden die Prozesse verwaltet, welche noch über ein Restquantum verfügen.
- Im abgelaufenen Array (expired array) werden die Prozesse verwaltet, deren Zeitscheibe aufgebraucht ist.

```
struct prio_array {
  unsigned int nr_active;
  unsigned long bitmap[BITMAP_SIZE];
  struct list_head queue[MAX_PRIO];
};
```



Implementierung (2)

- Im Falle einer anstehenden Scheduling-Entscheidung wird der Bitvektor nach dem ersten gesetzten Bit durchsucht (höchstpriore, nichtleere Warteschlange).
- Falls ein Prozess verdrängt wird oder in den Wartezustand geht, wird sein Zeitquantum um die Zeit der abgeschlossene Rechenphase verringert.
- Falls die Zeitscheibe eines Prozesses abgelaufen ist wird der Prozess in die jeweilige Warteschlage im expired array umgekettet. Dabei erhält der Prozess ein neues Zeitquantum.
- Falls `nr_active` im active array den Wert 0 hat, falls also alle Prozesse ihr Zeitquantum aufgebraucht haben, werden active array und expired array ausgetauscht.

Realtime-Prozesse

- Aufgrund der hohen Prioritäten werden Echtzeitprozesse beim Scheduling in jedem Fall den normalen Prozesses vorgezogen.
- Ein Prozess der Klasse `SCHED_FIFO` behält in jedem Falle den Prozessor so lange, bis ein Prozess mit höherer Priorität rechenbereit wird.
- Prozesse der Klasse `SCHED_RR` mit gleichen Prioritäten werden nach dem Round-Robin-Prinzip behandelt. Nach Ablauf seiner Zeitscheibe wird ein `SCHED_RR`-Prozess wieder am Ende seiner Warteschlange eingekettet.

Kontroll-Fragen

1. Welche Eigenschaft muss ein Scheduling-Algorithmus im Realzeitbetrieb unbedingt haben?
2. Warum werden E-/A-intensive Prozesse in fast allen praktisch eingesetzten Scheduling-Strategien bevorzugt?
3. Nennen Sie einen Grund für die Wahl einer kurzen Zeitscheibe beim Round-Robin-Verfahren.
4. Nennen Sie drei Einflussfaktoren, welche bei der Wahl der Zeitscheibe zu beachten sind.

Kapitel IV

Prozesssynchronisation und -kommunikation

Prozesssynchronisation und -kommunikation

Prozesssynchronisation

Problembeschreibung

- Die Prozesse in Multiprozessanwendungen laufen in den wenigsten Fällen völlig unabhängig voneinander ab.
- Die Abhängigkeiten werden durch gemeinsam benutzte „Betriebsmittel“ (Daten oder Hardware-Betriebsmittel) verursacht.
- Die Aktivitäten der Prozesse müssen in vielen Fällen in einer bestimmten Reihenfolge ablaufen.
- Das Synchronisationsproblem taucht in analoger Weise bei Multi-Threading-Anwendungen auf.
- Besondere Bedeutung hat die Prozesssynchronisation für den Betriebssystemkern, da die Datenstrukturen des Betriebssystemkerns von allen im Systemmodus ablaufenden Prozessen gemeinsam benutzt werden.

Prozesssynchronisation und -kommunikation

Programmbeispiel

```
static const int NUM_THREADS = 50;
static const int LOOPSIZE = 10000;

int sum = 0;

void thread_sum(int loopSize) {
    for (int i = 0; i < loopSize; i++) {
        sum ++;
    }
    return;
}


int main() {
  // ...
  for (int i = 0; i < NUM_THREADS; i++) {
    cout << "In main: creating thread " << i << endl;
    threads[i] = thread(thread_sum, LOOPSIZE);
  }
  // ...
}
```

Race Conditions

Die Operation `sum++` ist keine atomare Operation: Die folgende parallele Ausführung führt zu einem falschen Ergebnis:

Thread A

```
movl    sum, %edx
addl    $1, %edx

movl    %edx, sum
```

Thread B

```
movl    sum, %edx
addl    $1, %edx
movl    %edx, sum
```

Ergebnis: `sum` ist nur um 1 inkrementiert. Die Operation von Thread B wurde überschrieben. Wechselseitiger Ausschluss bei Zugriff auf `sum` nötig.

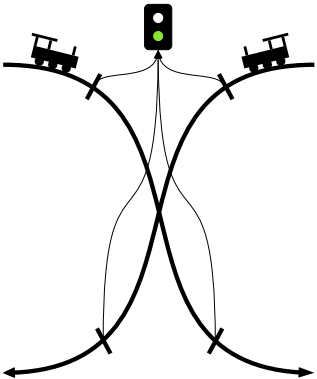
- Betriebsmittel, welche zu einem gewissen Zeitpunkt nur von einem Prozess genutzt werden können, heißen *kritische Betriebsmittel*.
- Eine Anweisungsfolge, in der auf ein kritisches Betriebsmittel zugegriffen wird, heißt *kritischer Abschnitt* (*critical section*).

Wechselseitiger Ausschluss: Forderungen

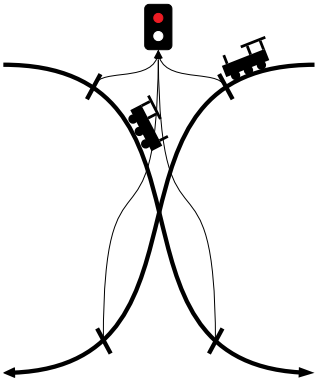
1. Zwei Prozesse dürfen nicht gleichzeitig in ihren kritischen Abschnitten sein (mutual exclusion).
2. Jeder Prozess, der am Eingang eines kritischen Abschnitts wartet, muss irgendwann den Abschnitt auch betreten dürfen: kein ewiges Warten darf möglich sein (fairness condition).
3. Ein Prozess darf außerhalb eines kritischen Abschnitts einen anderen Prozess nicht blockieren.
4. Es dürfen keine Annahmen über die Abarbeitungsgeschwindigkeit oder Anzahl der Prozesse bzw. Prozessoren gemacht werden.

(Dijkstra 1965)

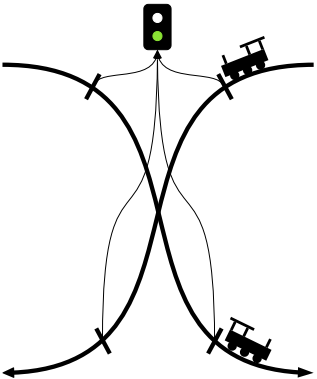
Semaphor



Kritischer Abschnitt ist frei.



Erster einfahrender Prozess sperrt den Abschnitt. Andere Prozesse warten.



Nach Ausfahrt wird der Abschnitt freigegeben.

Spin Locks

- Ein *Spin Lock* ist eine binäre Variable S, auf der die Operationen `spin_lock()` und `spin_unlock()` wie folgt definiert sind:

```
void spin_lock(int *S) {
  while (*S == 0);
  *S  = 0;
}

void spin_unlock(int *S) {
  *S = 1;
}
```

- In Einprozessor-Systemen machen Spin Locks wegen des aktiven Wartens (busy wait) keinen Sinn.
- In Multiprozessor-Systemen werden Spin Locks zur Synchronisation innerhalb des Betriebssystems eingesetzt (Beispiel Linux).

Programmbeispiel 1. Versuch

```
void spin_lock(int *S) {
  while (*S == 0);
  *S  = 0;
}

void spin_unlock(int *S) {
  *S = 1;
}

void thread_sum(int loopSize) {
    for (int i = 0; i < loopSize; i++) {
        spin_lock(&signal);
        sum ++;
        spin_unlock(&signal);
    }
    return;
}
```

Test-And-Set-Instruktionen (1)

Das kritische Betriebsmittel ist die Variable S.

Kritischer Abschnitt

```
if (S == 1)
   S = 0;
```

Assembler Sequenz

```
            movl S, %eax
            cmpl $1, %eax
            jne  WAIT
            movl $0, S
            jmp  KRITISCH
WAIT:       ;...
KRITISCH:   ;...
```

Falls es zwischen der cmpl-Anweisung und der jne-Anweisung zu einem Prozesswechsel kommt, leistet der Semaphor nicht den gewünschten wechselseitigen Ausschluss.

Test-And-Set-Instruktionen (2)

- Eine einfache Lösung des Problems besteht darin, während dieser kurzen Anweisungsfolge die Unterbrechungssperre zu setzten.
- Zur Realisierung von Semaphoren verfügen viele Prozessoren über eine nicht unterbrechbare Test-And-Set-Instruktion (TAS), welche den Wert einer Semaphorvariablen testet und ggf. dekrementiert.
- Intel-Prozessoren verfügen mit dem Exchange-Befehl über eine äquivalente Möglichkeit. Dieser tauscht den Inhalt einer Speicherzelle mit den Inhalt eines Registers.

Assembler Sequenz

```
          movl  $0, %eax
          xchgl S, %eax
          cmpl  $1, %eax
          jne   WAIT
          jmp   KRITISCH
WAIT:     ; ...
KRITISCH: ; ...
```

Programmbeispiel 2. Versuch

```
void spin_lock(int *S) {
    int val = 0;
    do {
      __asm__("xchg %0, %1" : "+q" (val), "+m" (*S));
      //val = __atomic_exchange_n(S, val, __ATOMIC_RELAXED);
    } while(val == 0);
}

void spin_unlock(int *S) {
  *S = 1;
}

void thread_sum(int loopSize) {
    for (int i = 0; i < loopSize; i++) {
        spin_lock(&signal);
        sum ++;
        spin_unlock(&signal);
    }
    return;
}
```

Binäre Semaphoren (Mutex)

- Statt aktiv zu warten und die CPU zu belegen, soll der Prozess blockieren.
- Hierzu bieten Betriebssysteme bzw. Thread-Bibliotheken sogenannte Semaphordienste an. In der einfachsten Form handelt es sich um *binäre Semaphoren* auch als *Mutex* bezeichnet.
- Ein binärer Semaphor ist eine zweiwertige Variable S, auf der die P- und V-Operationen wie folgt definiert sind.

```
void P(int *S) {
    if (*S == 0)
        < setze aufrufenden Prozess "wartend auf S" >
    *S = 0;
}

void V(int *S) {
    *S = 1;
    < setze einen wartenden Prozess "bereit" > ;
}
```

Conceptual Independence

```
void P(int *S) {
    while (*S == 0)
        < setze aufrufenden Prozess "wartend auf S" >
    *S = 0;
}

void V(int *S) {
    *S = 1;
    < setze einen wartenden Prozess "bereit" > ;
}
```

Blockiert und *bereit* sind Konzepte des Scheduling. Diese sollten nicht benutzt werden um Synchronisations-Information zu übermitteln. Also im *bereit*-Zustand erst nochmal das Semaphor prüfen!

Programmbeispiel: Mutex

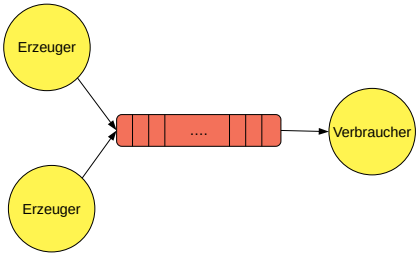
- Mit Hilfe eines `mutex`-Objekts aus der C++-Standardbibliothek lässt sich der konkurrierende Zugriff auf die Variable `sum` in der gewünschten Weise synchronisieren.
- Aktives Warten wie bei Spin Locks wird vermieden.

```
int sum = 0;
mutex mtx;

void thread_sum(int loopSize) {
    for (int i = 0; i < loopSize; i++) {
        mtx.lock();
        sum ++;
        mtx.unlock();
    }
    return;
}
```

Prozesssynchronisation und -kommunikation

Erzeuger/Verbraucherproblem

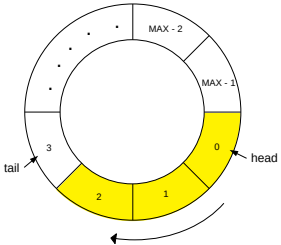
- Bei dem Erzeuger/Verbraucher-Problem handelt es sich um eine repräsentative, abstrakte Problemstellung zur Diskussion der Prozesssynchronisation und -kommunikation
- Ein oder mehrere Erzeugerprozesse erzeugen “Nachrichten” und schreiben diese in einen Nachrichtenpuffer von begrenzter Kapazität.
- Ein Verbraucherprozess liest die Nachrichten und leert hierdurch den Nachrichtenpuffer.



Programmbeispiel - Nachrichtenpuffer

Der Nachrichtenpuffer ist als ringförmig gespeicherte, lineare Liste organisiert und im gemeinsamen Speicher für alle beteiligten Prozesse bzw. Threads zugreifbar.

```
// Definition der FIFO-Queue
class Queue {
public:
// ...
  virtual void put(const int& ele);
private:
  int* _array;
  size_t _head;
  size_t _tail;
  size_t _size;
  size_t _cap;
};
```




```
void Queue::put(const int& ele) {
    if (_size == _cap) { // is_full
        throw std::runtime_error("Full");
    }
    _array[_tail] = ele;
    _tail = (_tail+1) % _cap;
    _size += 1;
}
```

Synchronisation mittels Semaphoren

- Die Funktion `Queue::put()` enthält einen kritischen Abschnitt und muss bei parallelem Zugriff wechselseitig geschützt werden. Zudem muss die Reihenfolge des Zugriffs auf die Queue koordiniert werden.
- Hierfür werden Semaphordienste mit erweiterter Funktionalitär benötigt.
- Die Systemschnittstelle von Linux enthält die von UNIX stammenden *System-V-Semaphordienste* welche *mehrwertige Semaphoren* zur Verfügung stellt.
- Die Posix-Groppe hat mit den sogenenannten *Posix-Semaphoren* ebenfalls eine etwas einfacher zu handhabende Bilbliothek für mehrwertuge Semaphoren spezifiziert, mit der sowohl Prozesse als auch Threads synchronisiert werden können.

Mehrwertige Semaphoren

Ein *mehrwertiger Semaphor* ist eine ganzzahlige Variable S, auf der die P- und V-Operationen wie folgt definiert sind:

```
void P(int *S) {
    while (*S < 1) {
        < setze aufrufenden Prozess "wartend auf S" >;
    }
    *S = *S-1;
}

void V(int *S) {
    *S = *S+1;
    < setze einen wartenden Prozess "bereit" >
}
```

POSIX Semaphore (1)

Die Posix-Bibliothek ist in der Header-Datei <semaphore.h> definiert. Hier eine Zusammenstellung der wichtigsten Funktionen:

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
```

- `sem_init()` erzeugt einen Semaphor vom Typ `sem_t`.
- *pshared* ist 0 falls die Semaphore zwischen Threads geteilt wird und $\neq 0$, wenn sie zwischen Prozessen geteilt werden soll.
- Der Semaphor muss in Speicher plaziert werden, den sich verschiedene Threads (thread-shared semaphore, z.B. als globale Variable) oder verschiedene Prozesse (process-shared semaphore, z.B. in Shared Memory) teilen.
- Der Initialwert ist durch *value* gegeben.
- `sem_destroy()` entfernt den Semaphor `sem`. Es muss sicher gestellt werden, dass zu dem Zeitpunkt kein anderer Prozess auf den Semaphor wartet.

POSIX Semaphor (2)

```
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
```

- `int sem_wait()` realisiert eine P-Operation auf dem Semaphor `sem`. D.h. der aufrufende Prozess wird blockiert, falls der Semaphorwert kleiner als 1 ist; ansonsten wird der Semaphorwert dekrementiert.
- `int sem_post()` realisiert eine V-Operation auf dem Semaphor `sem`. D.h. der Semaphor wird inkrementiert und ein wartender Prozess wird bereit gesetzt.
- Im Erfolgsfall liefern `int sem_wait()` und `int sem_post()` den Wert 0, im Fehlerfall den Wert -1 zurück.

Programmbeispiel 1 - synchronisiert

- Mittels Posix-Semaphoren lässt sich der Zugriff auf die Queue-Datei synchronisieren.
- Im Konstruktor der Klasse `Queue_sync` wird der Semaphor mittes `sem_init()` mit dem Wert 1 initialisiert.
- Die Funktion `Queue_sync::put()` schützt den kritischen Abschnitt in der Funktion `Queue::put()` durch Einschluss in die Semaphorfunktionen `sem_wait()` und `sem_post()`.

```
class Queue_sync : public Queue {
//...
private:
  sem_t mutex;
};

void Queue_sync::put(const int& ele) {
  sem_wait(&mutex);
  try {
    Queue::put(ele);
  } catch(...) {
    sem_post(&mutex);
    throw;
  }
  sem_post(&mutex);
}
```

Prozesskooperation

- Im Falle der Prozesskooperation arbeiten mehrere parallel ablaufende Prozesse kooperativ an einer gemeinsamen Aufgabe.
- Beim Zugriff auf den Nachrichtenpuffer ist wechselseitiger Ausschluss erforderlich.
- Eine korrekte Nachrichtenübermittlung erfordert zudem die Einhaltung der folgenden Zugriffsreihenfolge:

1. Der Verbraucher kann erst dann eine Nachricht aus dem Puffer entnehmen, wenn ein Erzeuger vorher dort eine Nachricht hinterlegt hat.
2. Bei vollem Puffer kann ein Erzeuger erst dann eine Nachricht in der Puffer schreiben, wenn der Verbraucher vorher eine Nachricht entnommen hat:

Semaphoren bei Prozesskooperation

- Lösung mit Hilfe von zwei mehrwertigen Semaphoren:
- Ein Semaphor namens `voll` zählt die mit Nachrichten belegten Pufferplätze. Er wird mit 0 initialisiert.
- Ein Semaphor namens `leer` zählt die freien Pufferplätze. Er wird mit einem Wert `MAX` (Pufferkapazität) initialisiert.
- Beim Zugriff auf den Nachrichtenpuffer führen Erzeuger und Verbraucher überkreuzt P- und V-Operationen auf den beiden Semaphoren durch.

Erzeuger:

```
P(leer);
<Nachricht in Puffer schreiben>
V(voll);
```

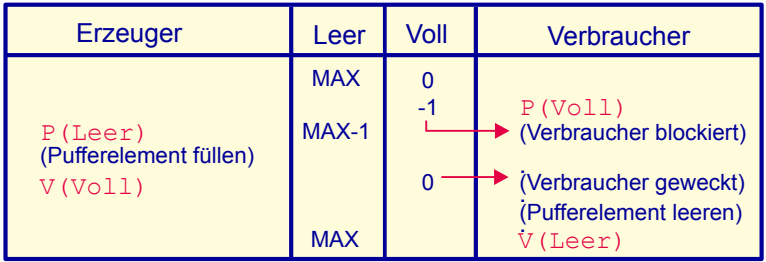
Verbraucher:

```
P(voll);
<Nachricht aus Puffer lesen>
V(leer);
```

Programmbeispiel 2 - blockierende Queue

```
class Queue_block : public Queue {
//...
private:
  sem_t mutex;
  sem_t empty;
  sem_t full;
};

void Queue_block::put(const int& ele) {
  sem_wait(&empty);
  sem_wait(&mutex);
  Queue::put(ele);
  sem_post(&mutex);
  sem_post(&full);
}
```

Ablauf Queue Leer



Ablauf Queue Voll

Erzeuger	Leer	Voll	Verbraucher
	MAX	0	
`P(Leer)`	MAX-1		
(Pufferelement füllen)			
`V(Voll)`		1	
`P(Leer)`	MAX-2		
(Pufferelement füllen)			
`V(Voll)`		2	
`P(Leer)`	0		
(Pufferelement füllen)			
`V(Voll)`		MAX	
`P(Leer)`	-1		
(Erzeuger blockiert) ←		MAX-1	`P(Voll)`
			(Pufferelement leeren)
(Erzeuger geweckt) ←	0		`V(Leer)`
(Pufferelement füllen)			
`V(Voll)`		MAX	

Prozesssynchronisation und -kommunikation

Wechselseitiger Ausschluss

Aufgabe

Füge ein Semaphor hinzu um wechselseitigen Ausschluss für den Zugriff auf die gemeinsame Variable `count` zu erreichen.

Thread A

```
cout = count + 1
```

Thread B

```
cout = count + 1
```

Signalisierung

Aufgabe

Thread A soll die Task `a1` ausgeführt haben, bevor Thread B die Task `b2` ausführt. Dazu muss Thread A die Erledigung der Task an Thread B signalisieren. Fügen Sie entsprechende Semaphor-Operationen ein:

```
statement a1

statement a2
```

```
statement b1

statement b2
```

Rendezvous

Aufgabe

Das Pattern aus der letzten Aufgabe soll verallgemeinert werden, so dass es in beide Richtungen funktioniert. Also

- a1 wird garantiert vor b2 ausgeführt
- b1 wird garantiert vor a2 ausgeführt

Thread A

```
statement a1


statement a2
```

Thread B

```
statement b1


statement b2
```

Barrier

Aufgabe

Verallgemeinere die Lösung des Rendevous-Problems auf n Threads. Jeder Thread hat Zugriff auf die Nummer der Threads n.
Bedingung: Kein Thread erreicht `critical point` bevor alle Threads `rendezvous` ausgeführt haben.

Verwenden Sie einen Zähler `count` der Zählt wieviele Threads `rendevous` bereits erreicht haben.

Thread i
```
rendezvous

Critical point
```

Prozesssynchronisation und -kommunikation

Mutex und Bedingungsvariable

Die C++-Standard-Bibliothek enthält keine mehrwertigen Semaphore, jedoch Mutex und Bedingungsvariablen.

Mutex

- Wechselseitiger Ausschluss ist durch eine Mutex-Variable möglich.
- Ein Mutex muss jedoch immer vom selben Thread belegt und wieder freigegeben werden, eignet sich also nicht zur Signalisierung.

Bedingungsvariable

- Eine Bedingungsvariable (condition variable) ist eine Sperrvariable, welche mit einer bestimmten Wartebedingung assoziiert ist. Ein Thread kann auf das Zutreffen der Bedingung warten.
- Ein weiterer Thread kann das Zutreffen der Bedingung signalisieren und so den wartenden Thread wecken.

Mutex

- C++ bietet im Header `<mutex>` eine binären Semaphor vom Typ `std::mutex`.
  - Die Methode `void lock()` sperrt den Mutex und blockiert, falls der Mutex bereits gesperrt ist.
  - Die Methode `bool try_lock()` blockiert nicht, sonder gibt `true` zurück, falls der Mutex erworben werden konnte, und `false` wenn der Mutex bereits gesperrt war.
  - Die Methode `void unlock()` gibt einen Mutex, den der Thread hält, wieder frei.
- Ein Mutex kann nicht zugewiesen oder kopiert werden.
- Ein Mutex muss von demselben Thread gesperrt und freigegeben werden. Für die Signalisierung zwischen Threads kann ein Mutex nicht verwendet werden. Dazu werden Bedingungsvariablen benötigt.

Benutzen von Mutex

- C++ verfügt über Wrapper-Klassen, die das Arbeiten mit Mutex-Locks vereinfachen. Dabei wird das RAII-Prinzip (Resource acquisition is initialization) angewendet.
- Eine Wrapper-Klasse um einen Mutex ruft `lock()` im Konstruktor auf und `unlock()` im Destruktor.
  - `std::lock_guard` ist die einfachste Klasse dazu. Verwendung:

```
mutex m;
void critical() {
   lock_guard<mutex> lock(m);
   // ...
} // Destruktor von lock_guard ruft m.unlock() auf
```

  - `std::unique_lock` hat eine äquivalente Funktionalität, bietet aber zusätzlich einen Durchgriff auf `lock()` und `unlock()` und sollte nur verwendet werden, wenn es benötigt wird.

Bedingungsvariable

- Eine Bedingungsvariable ist eine Sperrvariable, welche mit einer bestimmten Wartebedingung assoziiert ist. Ein Thread kann auf das Zutreffen der Bedingung warten. Ein weiterer Thread kann das Zutreffen der Bedingung signalisieren und so den wartenden Thread wecken.
- Damit lassen sich also `<setze aufrufenden Prozess "wartend auf S">` und `<setze einen wartenden Prozess "bereit">` in Folie 4.21 realisieren.
- C++ bietet im Header `<condition_variable>` eine Bedingungsvariable vom Typ `std::condition_variable`.
  - `void wait (unique_lock<mutex>& lck)` blockiert den aktuellen Thread, bis ein Aufweck-Signal für die Bedingungsvariable gesendet wird. Als Parameter wird ein Mutex erwartet (in einem `unique_lock`), der vor dem Blockieren freigegeben wird und vor dem Aufwachen wieder erworben wird.
  - `void cv.notify_one()` ermöglicht einem Thread einen (typischer Fall) auf die Bedingung wartenden Thread zu benachrichtigen. `void cv.notify_all()` benachrichtigt alle wartenden Threads.

Zusammenspiel von Mutex und Bedingungsvariable

- Vor einem Aufruf von `wait()` muss ein Mutex belegt werden. `wait()` gibt den Mutex frei, und belegt ihn erneut wenn der Thread wieder aufwacht.
- Bei der Rückkehr aus `wait()` ist nicht in jedem Falle gewährleistet, dass die Wartebedingung tatsächlich zutrifft ($\rightarrow$ spurious wakeup). Deshalb muss der Thread die Gültigkeit der Wartebedingung erneut prüfen und sich ggf. erneut blockieren.

Objekte:

```
mutex mtx;
condition_variable cv;
bool ready = false;
```

Thread 1

```
{
  unique_lock<mutex> lock(mtx);
  while (! ready)
        cv.wait(lock);
  // do anything ...
} // automatc mtx.unlock(),
```

Thread 2

```
{
  unique_lock<mutex> lock(mtx);
  ready = true;
  cv.notify_one();
} // automatic mtx.unlock(),
```

Programmbeispiel: Mutex und Bedingungsvariable

```
#include "queue.h"
#include <mutex>
#include <condition_variable>

class Queue_cv : public Queue {
public:
  Queue_cv(size_t cap);
  void put(const int& ele);
  int get();
private:
  std::mutex m;
  std::condition_variable notempty;
  std::condition_variable notfull;
};
```

```
void Queue_cv::put(const int& ele) {
  unique_lock<mutex> ul(m);
  while(is_full()) {
    notfull.wait(ul);
  }
  Queue::put(ele);
  notempty.notify_one();
}

int Queue_cv::get() {
  unique_lock<mutex> ul(m);
  while(is_empty()) {
    notempty.wait(ul);
  }
  int ret = Queue::get();
  notfull.notify_one();
  return ret;
}
```

Prozesssynchronisation und -kommunikation

Deadlocks

Eine Menge von Prozessen befindet sich in einem Deadlock-Zustand, falls jeder Prozess auf ein Ereignis wartet, dass nur ein anderer Prozess der Menge auslösen kann.

```
sem_t sem1, sem2;
// ...
sem_init(&sem1, 1);
sem_init(&sem2, 1);
sem_wait(&sem1);
sem_wait(&sem2);

/* kritischer Abschnitt */

sem_post(&sem1);
sem_post(&sem2);
//...
```

```
sem_t sem1, sem2;
// ...
sem_init(&sem1, 1);
sem_init(&sem2, 1);
sem_wait(&sem2);
sem_wait(&sem1);


/* kritischer Abschnitt */

sem_post(&sem2);
sem_post(&sem1);
//...
```

Bedingungen für ein Deadlock

Mutual Exclusion Der Zugriff auf die Betriebsmittel ist exklusiv.

Hold and Wait Die Prozesse fordern Betriebsmittel an, behalten aber zugleich den Zugriff auf andere.

No Preemption Die Betriebsmittel werden ausschließlich durch die Prozesse freigegeben (Da Ressourcenzugriff eines Prozesses nicht unterbrochen werden kann.)

Circular Wait Mindestens zwei Prozesse besitzen bezüglich der Betriebsmittel eine zirkuläre Abhängigkeit.

Deadlockverhinderung

- Preemption durchführen: Einem Prozess werden Betriebsmittel entzogen, um sie einem anderen zuzuteilen.
- Hold and Wait verhindern: Jeder Prozess gibt zu Beginn an, welche Betriebsmittel er benötigt. Falls alle benötigten Betriebsmittel gleichzeitig frei sind, bekommt sie ein Prozess auf einmal zugeteilt.
- Mutual Exclusion beseitigen: Die benötigten Betriebsmittel für alle Prozesse zugänglich zu machen, indem man den exklusiven Zugriff auflöst. Alternativ Spooling (Beispiel: Drucker) oder Virtualisierung von Betriebsmitteln (Beispiel: CPU).
- Circular Wait ausschließen: Betriebsmittel werden linear geordnet und in aufsteigender Reihenfolge vergeben.

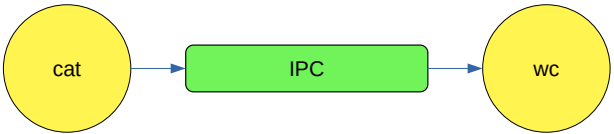
Prozesssynchronisation und -kommunikation

Interprozesskommunikation

- Häufig arbeiten verschiedene Prozesse zusammen, um gemeinsam eine Aufgabe zu lösen oder gemeinsam eine Ressource zu Nutzen.
- Häufige Situation: Aus der Shell werden zwei Prozesse gestartet

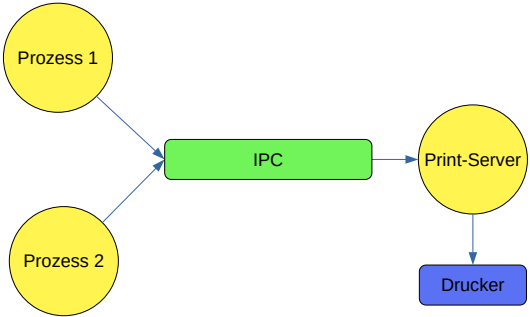
  ```
  cat file.txt | wc
  ```

- Die Prozesse brauchen Interprozesskommunikation, um die Daten auszutauschen.



Beispiel Print-Server

- Der Drucker darf nur exklusiv von einem Prozess genutzt werden.
- Lösung durch Mutex: Ein Prozess druckt, die anderen Prozesse müssen warten.
- Bessere Lösung durch Interprozesskommunikation:
  - Ein Durckprozess besitzt den Drucker und nimmt die Daten der anderen Prozesse entgegen.
  - Die Prozesse dürfen nicht selbst drucken, können aber weiterarbeiten.



Interprozesskommunikation - Varianten

- Shared Memory gestattet eine schnelle Übertragung großer Datenmengen zwischen Prozessen. Der Zugriff auf den gemeinsamen Speicher muss über Semaphoren synchronisiert werden.
- Message Queues ermöglichen einen adressierten, bidirektionalen Nachrichtenverkehr zwischen mehreren Prozessen. Sie eignen sich insbesondere zur Realisierung von Client-Server-Anwendungen.
- FIFO-Dateien gestatten einen einfach zu realisierenden, unidirektionalen Datenaustausch zwischen Prozessen auf Dateiebene. Sie können in zwei Varianten (named pipe und unnamed pipe) verwendet werden.

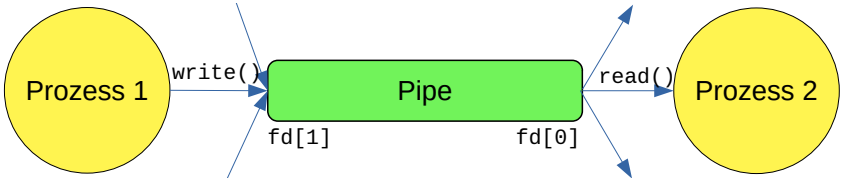
Interprozesskommunikation in Linux

In Linux-Systemen gibt es zahlreiche Möglichkeiten zur Interprozesskommunikation, z.B.

- Pipes
- FIFO-Dateien
- Shared Memory
- Memory Mapped Files
- Message Queues
- Signale
- Sockets

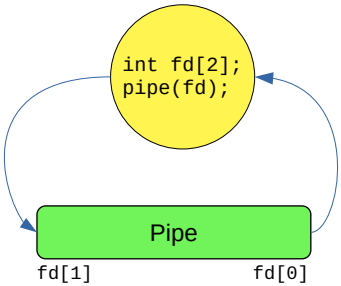
Unnamed Pipes

- FIFO-Buffer mit fester Größe zum Austausch eines Bytestroms.
- Schreiben (`write()`) in eine volle Pipe blockiert den schreibenden Prozess, bis ein anderer Prozess Daten gelesen hat.
- Lesen (`read()`) aus einer leeren Pipe blockiert den lesenden Prozess, bis ein anderer Prozess in die Pipe geschrieben hat.
- Einrichten einer Pipe durch den Systemaufruf `pipe()`.
- Eine Pipe lässt sich nur zum Datenaustausch zwischen verwandten Prozessen verwenden.

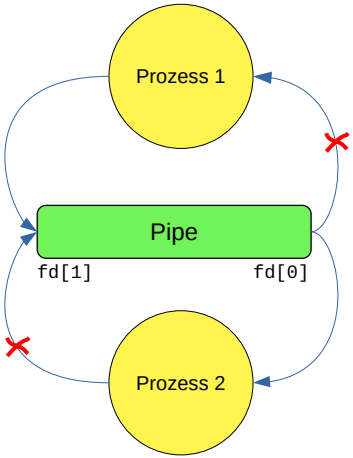


Pipes einrichten

```
int pipe(int fd[2]);
```



Nach dem Aufruf von `pipe()` ist der Prozess mit dem Lese- und Schreibende der Pipe verbunden und erhält ein Paar von Datei-Deskriptoren.



Nach `fork()` sind beide Prozesse mit der Pipe verbunden und können das ungenutzte Ende schließen.

FIFO-Dateien erzeugen

- FIFO-Dateien sind named Pipes.
- Dadurch können auch nicht verwandte Prozesse miteinander Kommunizieren.
- Erzeugen einer FIFO-Datei im Dateisystem ist über das Kommando `mknod`:

  `mknod -m <mode> <name> p`

  - mode gibt den Zugriffsmodus der zu erzeugenden FIFO-Datei an.
  - name gibt einen Pfadnamen für die zu erzeugende FIFO-Datei an.
  - Der Parameter p gibt an, dass es sich bei der zu erzeugenden Datei um eine FIFO-Datei (pipe) handelt.
- Erzeugen einer FIFO-Datei über den Systemaufruf mknod()

```
int mknod(char *path, int mode, int dev);
```

  - path gibt den Pfadnamen der Spezialdatei an.
  - mode enthält im Falle einer FIFO-Datei die Oderverknüpfung von Zugriffsmodus und dem Wert `S_FIFO`. `S_FIFO` ist in `<stat.h>` definiert ist.
  - dev hat im Falle der FIFO-Datei keine Bedeutung.

FIFO-Dateien öffnen

```
int open(char *path, int oflag);
int close(int filedes);
```

- FIFO-Dateien werden mit `open()` geöffnet, der Rückgabewert ist der vergebene Filedeskriptor.
- Als Flags sind insbesondere die Werte `O_RDONLY` (Lesen) und `O_WRONLY` (Schreiben) von Interesse.
- Das Öffnen einer FIFO-Datei zum Lesen blockiert den Prozess solange bis ein weiterer Prozess die FIFO-Datei zum Schreiben geöffnet hat. Umgekehrt wartet ein schreibender Prozess im `open()`-Systemaufruf bis ein weiterer Prozess die FIFO-Datei zum Lesen öffnet.
- Schließen einer FIFO-Datei ist mittels `close()` moglich. Einziger Parameter ist der von `open()` gelieferte Dateideskriptor.
- Wird die FIFO-Datei vom letzten schreibenden Prozess geschlossen, so erhalten alle lesenden Prozesse ein End-of-File.

FIFO-Dateien lesen/schreiben

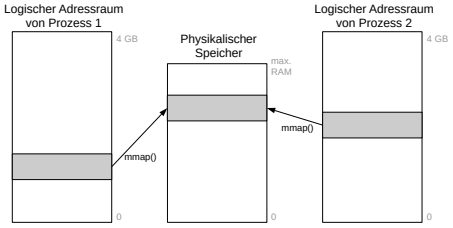
```
int write(int filedes, char *buf, unsigned nbyte);
int read(int filedes, char *buf, unsigned nbyte);
```

- `write()` schreibt `nbyte` Bytes aus dem Puffer `buf` in die FIFO-Datei mit Dateideskriptor `filedes`. Als Funktionswert liefert `write()` die Anzahl der übertragenen Bytes zurück.
- Beim Schreiben in eine volle FIFO-Datei blockiert der Systemaufruf.
- `read()` liest `nbyte` Bytes aus der Datei `filedes` und legt diese im Puffer `buf` ab. Als Funktionswert liefert `read()` die Anzahl der tatsächlich gelesenen Bytes zurück.
- Die Daten werden mit dem Lesen entfernt.
- `read()` blockiert bei leerer FIFO-Datei.
- Hat der letzte Schreiber die FIFO-Datei geschlossen, endet der `read()`-Aufruf mit Rückgabewert 0.

Shared Memory

- Die Adressräume von Prozessen sind strikt gegeneinander abgeschottet, sodass ohne besondere Vorkehrungen kein Datenaustauch möglich ist.
- Das Betriebssystem Linux bietet hierfür die Möglichkeit an, gemeinsame Speicherbereiche *(Shared Memory)* anzulegen.

- Shared Memory ist die schnellste Form der Interprozesskommunikation.
- Die Synchronisation muss jedoch explizit von den Anwendungen erfolgen.
- Vorsicht bei der Verwendung von Zeigern, da der Shared Memory in den Prozessen an unterschiedlichen Adressen liegen kann.



Shared Memory (POSIX)

- Ein POSIX Shared-Memory-Objekt erstellen oder öffnen:
  `int shm_open (const char *name, int oflag, mod_t mode);` Das Shared-Memory-Objekt wird im Filesystem im Ordner `/dev/shm` angezeigt.
- Größe des Shared-Memory anpassen:
  `ftruncate (smd, size_t len);`
- Mapping des Shared-Memory-Objekts in den Adressraum des Prozesses:
  `void* mmap (void* addr, size_t len, [...], smd, [...]);`
- Shared-Memory-Objekts aus dem Arbeitsspeicher entfernen:
  `int munmap (void* addr, size_t len);`
- Shared-Memory-Objekts löschen:
  `int shm_unlink (const char *name);`

Message-Queues (POSIX)

Message Queues ermöglichen einen Nachrichtenaustausch zwischen mehreren Prozessen. Dabei werden keinerlei Vorgaben betreffend Format und Bedeutung der auszutauschenden Nachrichten gemacht. Ähnlich einem Protokoll zur Kommunikation über Datennetze müssen die beteiligten Prozesse Absprachen hinsichtlich Format, Bedeutung und Abfolge von Nachrichten treffen.

- Der Systemaufruf `mq_open()` kann eine Message Queue erzeugen. Rückgabewert ist eine eine ganzahlige ID, die die Message Queue identifiziert.
- Ein weiterer Prozess kann die erzeugte Message Queue mitbenutzen, indem `mq_open()` unter Angabe eines identischen Namens aufgerufen wird.
- Die eigentliche Kommunikation erfolgt über die Systemaufrufe `mq_send()` (Senden einer Nachricht) und `mq_receive()` (Empfangen einer Nachricht).
- Eine Nachricht kann eine Priorität haben. Es wird immer die Nachricht mit der höchsten Priorität zuerst empfangen. Anders als bei der Kommunikation über Shared Memory, ist eine Synchronisierung bei Message-Queues direkt eingebaut.

Signale

- Ein Signal unterbricht ein laufendes Programm (ähnlich wie ein Interrupt) und kommuniziert dadurch mit ihm.
- Das eintreffen eines Signals führt zu einer Default-Aktion, typischerweise Programmabbruch.
- Jedes Signal wird durch einen numerischen Wert dargestellt, häufige Signale sind:

Nr.	Name	Beschreibung	Standard-Reaktion
2	SIGINT	wird an den Prozess gesandt, wenn die Abbruchtaste (Ctrl-C) gedrückt wird.	Programmabbruch
9	SIGKILL	Nicht abfangbares Signal zum Abbruch eines Prozesses durch das Betriebssystem	Programmabbruch
10	SIGUSR1	benutzerdefiniertes Signal	Programmabbruch
14	SIGALRM	wird nach Ablauf eines durch alarm() gesetzten Timers gesendet	Programmabbruch
15	SIGTERM	wird standardmäßig beim Aufruf des kill-Kommandos gesendet.	Programmabbruch
17	SIGCHLD	wird beim Ende eines Sohnprozesses gesendet	wird ignoriert

Signale senden

- Das Kommando kill sendet ein Signal an einen Prozess.

Beispiel

`kill [-signo ] pid`

Der Prozess mit ID `pid` erhält das Signal signo (int-Wert zwischen 1 und 30). Bei fehlender Angabe von signo wird das Signal 15 (SIGTERM) gesandt.

- Aus einem C-Programm heraus, kann der Systemaufruf `kill()` verwendet werden, um anderen Prozessen Signale zu senden.
- Ein Prozess kann Signale ignorieren oder durch einen Signal-Handler behandeln. Ausnahmen sind die Signale SIGKILL und SIGSTOP.
- Im Praktikum werden wir Signale abfangen (s. Folie 8.60).

Allgemeine Fragen

1. Warum umfasst das Problem der Prozesskooperation, das Problem des wechselseitigen Ausschlusses?
2. Warum laufen die P- und V-Operation typischerweise im privilegierten Systemmodus ab?
3. Nennen Sie eine Alternative zu einem Test-And-Set-Befehl.
4. Wie viele Semaphoren sind zur Lösung eines Erzeuger-/Verbraucherproblem notwendig?
5. Nennen Sie einen Unterschied zwischen einer FIFO-Datei und einer normalen Plattendatei.

Kapitel V

Speicherverwaltung

Speicherverwaltung

Motivation

Partitionierung

Virtueller Speicher

Betriebsmittel

- Aufgaben des Betriebssystems
  - Verwaltung der Betriebsmittel des Rechners
  - Schaffung von Abstraktionen, die Anwendungen einen einfachen und effizienten Umgang mit Betriebsmitteln erlauben
- Bisher: Prozesse als Abstraktion von der realen CPU
- Dieses Kapitel: Verwaltung von Haupt- und Hintergrundspeicher

Speicherverwaltung

Aufgaben der Speicherverwaltung

- Zuteilung von physikalischem Speicherplatz am die im System befindlichen Prozesse.
- Abbildung des logischen Adressraums in den physikalischen Adressraum.
- Schutz des von einem Prozess belegten Speicherbereichs vor unerlaubten Zugriffen durch andere Prozesse.

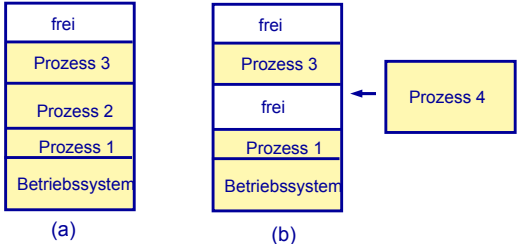
Die wichtigsten Verfahren zur Speicherverwaltung

- Partitionierung - in einfachen Rechnersystemen ohne besondere Hardware-Unterstützung zur Speicherverwaltung
- Virtuelle Speichertechnik - in leistungsfähigeren Rechnersystemen mit geeigneter Hardware-Unterstützung zur Speicherverwaltung.

Speicherverwaltung

Partitionierung

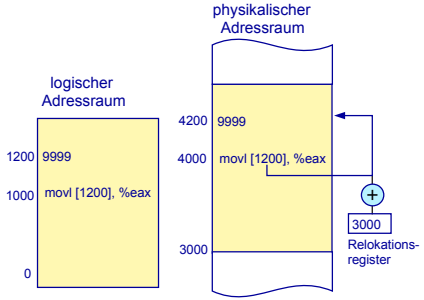
- Jedem Prozess wird über dessen gesamte Laufzeit ein zusammenhängender Speicherbereich (Partition) zugeteilt.
- Die jeweils benötigte Länge entnimmt das Betriebssystem aus den vom Compiler generierten Angaben in der ladefähigen Programmdatei.
- Das Phänomen der Zerstückelung des Arbeitsspeichers bezeichnet man als externe Fragmentierung.



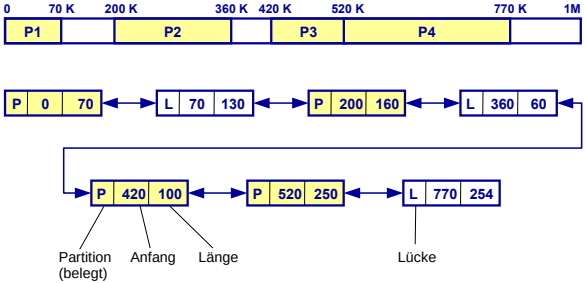
Relokation und Speicherschutz

Die Abbildung des logischen Adressraums an die physikalische Lage der Partition bezeichnet man als Relokation.

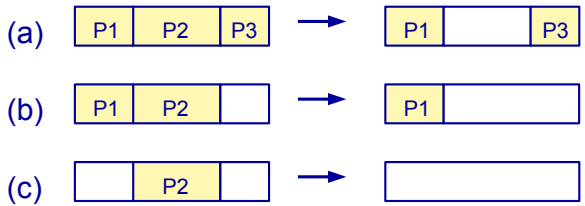
- Ein Reloktionsregister enthält die Anfangsadresse der Partition des aktuell ausgeführten Prozesses.
- Beim jedem Speicherzugriff wird die logische Adresse um den Inhalt des Relokationsregisters inkrementiert.
- Das Programmlängenregister enthält die maximale logische Programmadresse.
- Bei jedem Speicherzugriff wird die logische Adresse mit dem Wert des Programm-längenregisters verglichen.



Speicherverwaltung mit verketteten Listen



Freigabe einer Partition



Belegung einer Partition

- Die Belegungsstrategie First Fit durchsucht die Liste in aufsteigender Reihenfolge, bis eine ausreichend große Lücke gefunden wird.
- Die Lücke wird in zwei Teile aufgespalten. Der Prozess wird in der ersten Teil eingelagert

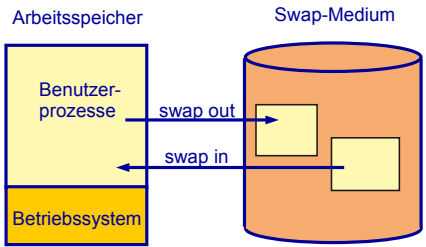
- Bei der Variante Next Fit wird die Suche an der Listenposition fortgesetzt, welche auf den Eintrag der zuvor belegten Partition folgt.
- Next Fit strebt eine gleichmäßige Ausnutzung aller Adressbereiche an.
- Best Fit durchsucht die gesamte Liste und belegt die kleinste Lücke, welche größer ausreichend groß ist. Ziel von Best Fit ist es, die Lücke im Speicher zu finden, in die der Prozess "am besten" passt.
- Analog dazu sucht Worst Fit nach der Lücke, bei der der größtmögliche Rest bleibt.
- Best Fit oder Worst Fit führen bei größerem Aufwand nicht zu einer verbesserten Speicherausnutzung als First Fit.

Partitionierung mit Swapping

- In Dialogsystemen kann die Anzahl der Prozesse so anwachsen, dass der benötigte logische Adressraum den physikalischen Speicherplatz übersteigt.
- Beim Swapping-Verfahren kann ein Prozess im Zustand "blockiert" auf Peripherspeicher ausgelagert werden (swap out).
- Nach einer bestimmten Zeit wird versucht, ausgelagerte Prozesse, welche sich im Zustand "bereit" befinden, wieder einzulagern (swap in).

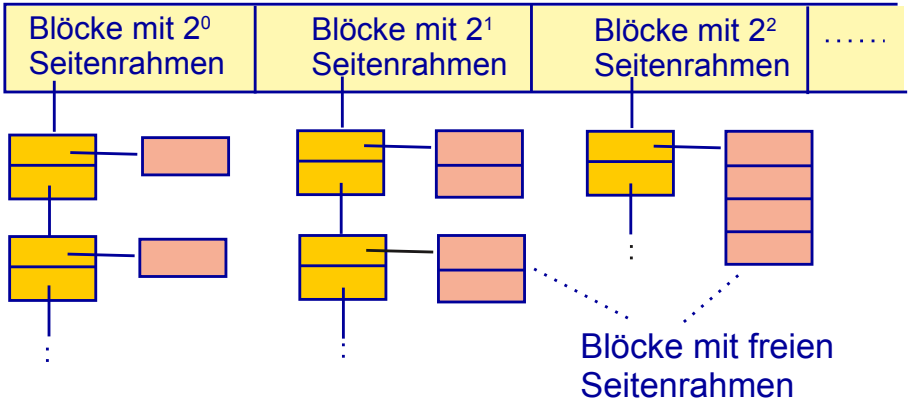
Organisation des Swap-Bereichs

1. Der Swap- Bereich besteht aus einer speziell hierfür vorgesehenen Plattendatei (Swap File).
2. Der Swap-Bereich liegt in einer eigenen Partition einer Festplatte (Swap-Partition).



Buddy-Verfahren

- Linux: Die freien Speicherbereiche werden in dem `free_area`-Vektor verwaltet. Der Vektor enthält verkettete Listen, welche jeweils zusammenhängende, freie Speicherblöcke der Größe 2^0 Seitenrahmen, 2^1 Seitenrahmen, 2^2 Seitenrahmen usw. verwalten.



Linux – Belegung und Freigabe von Speicher

Belegen und Freigabe von Speicherblöcken

- Die Belegung von physikalischem Speicherplatz geschieht immer in zusammenhängenden Blöcken mit einer 2er-Potenz der Seitenrahmengröße.
- Wird kein freier Block der gewünschten Größe gefunden, so wird ein Block der nächst höheren Größe in zwei Blöcke aufgeteilt.
- Bei der Rückgabe eines Blocks wird geprüft, ob dieser mit einem benachbarten, gleich großen Block zu einem doppelt so großen Block verschmolzen werden kann. Auf diese Weise wird die externe Fragmentierung verringert.
- Die beschriebene Art und Weise der Belegung und Freigabe wird als Buddy-Algorithmus bezeichnet.

Slab-Allokatoren

- Häufig benötigte Datenstrukturen des Kerns (z.B. task_struct, inode) haben nicht die Größe eines reservierbaren Speicherblocks
- Zur Vermeidung von interner Fragmentierung) gibt es für diese Datenstrukturen spezielle Slab-Allokatoren, welche einen größeren Speicherblock reservieren und diesen selbst verwalten.

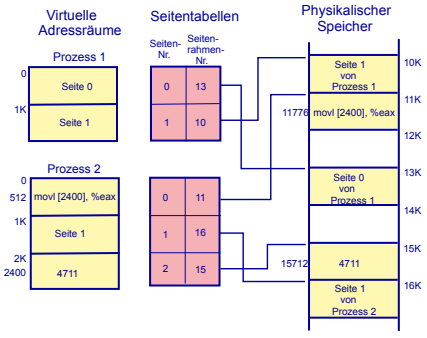
Partitionierung

- Das Verfahren der Partitionierung enthält eine Reihe von Unzulänglichkeiten:
  1. Das Problem der Fragmentierung ist nur unbefriedigend gelöst.
  2. Der logische Adressraum eines Prozesses wird vollständig im Speicher gehalten.
  3. Die Größe des logischen Adressraums eines Prozesses ist durch die Größe des vorhandenen Speichers begrenzt.
- Bei virtuellem Speicher ist die Größe des logischen Adressraums nur durch die Adressbreite, nicht durch die Größe des physikalischen Speichers begrenzt. Der logische Adressraum wird in diesem Fall als virtueller Adressraum bezeichnet.

Speicherverwaltung

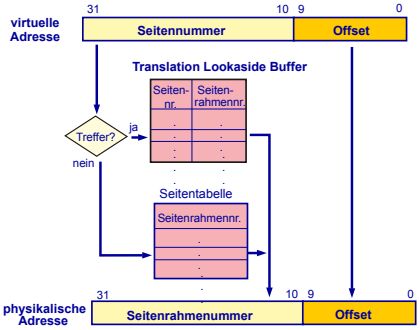
Adressabbildung bei virtuellem Speicher

- Die Adressabbildung bei virtuellem Speicher wird als Paging bezeichnet.
- Der virtuelle Adressraum eines Programms wird in Seiten (Pages) aufgeteilt.
- Der physikalische Speicher ist in Seitenrahmen (Page Frames) aufgeteilt.
- Über eine Seitentabelle (Page Table) wird die virtuelle Anfangsadresse einer Seite auf die physika-lische Anfangsadresse eines Seitenrahmens abgebildet.
- Seitengröße: 512 Byte bis 16K



Memory Management Unit

- Die Adressabbildung bei Paging wird durch eine Hardware-Einrichtung namens Memory Management Unit realisiert.
- Zur Beschleunigung enthält die Memory Management Unit einen Translation Lookaside Buffer (TLB).
- Der TLB ist ein assoziativer Cache-Speicher mit den Adresspaaren der zuletzt durchgeführten Adress-umrechnungen.
- Im Falle eines Treffers ist kein Zugriff auf den Arbeits-speicher erforderlich (Trefferrate: bis zu 90 %).



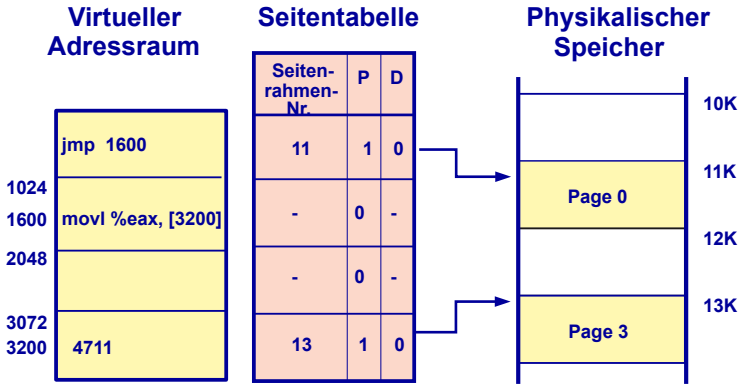
Demand Paging

- Zur Ausführungszeit muss sich nicht der gesamte Adressraum im Speicher befinden.
- Ein vollständiges Abbild des virtuellen Programmadressraums befindet sich auf dem Peripherspeicher.
- Die benötigten Seiten werden bei Bedarf (on demand) in den Arbeitsspeicher nachgeladen.
- Der Zugriff auf eine nicht im Speicher befindliche Seite löst eine Seitenfehler-Exception (Page Fault) aus.
- Die fehlende Seite wird vom Peripherspeicher in einem freien Seitenrahmen nachgeladen.
- Nach Behebung des Seitenfehlers kann die Ausführung fortgesetzt werden.
- Falls kein freier Seitenrahmen existiert, wird nach einer bestimmten Ersetzungsstrategie (Replacement Policy) ein Seitenrahmen ausgewählt und dessen Inhalt mit der fehlenden Seite überschrieben.
- Falls der Inhalt des zu überlagernden Seitenrahmens verändert wurde, wird er vor der Überlagerung auf den Peripherspeicher zurück geschrieben.

Seitenfehler beim Demand Paging

Demand Paging benötigt zwei Steuerbits in den Seitentabelleneinträgen:

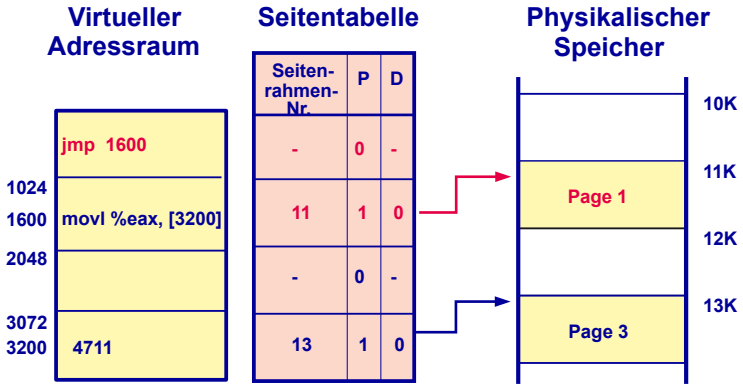
- Das Present-Bit (P-Bit) gibt an, ob sich die Seite im Speicher befindet.
- Das Dirty-Bit (D-Bit) zeigt an, ob die Seite verändert wurde.



Seitenfehler beim Demand Paging

Demand Paging benötigt zwei Steuerbits in den Seitentabelleneinträgen:

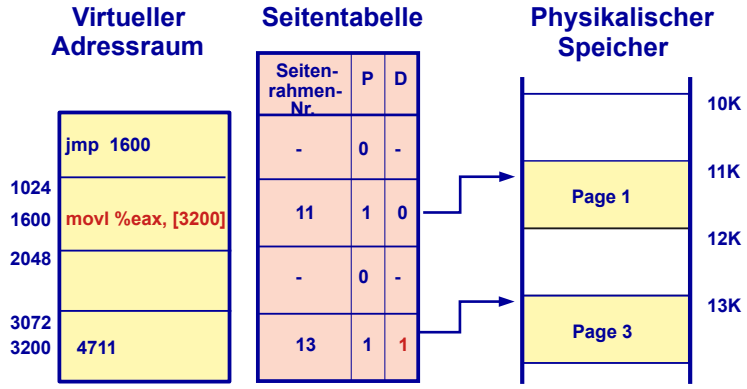
- Das Present-Bit (P-Bit) gibt an, ob sich die Seite im Speicher befindet.
- Das Dirty-Bit (D-Bit) zeigt an, ob die Seite verändert wurde.



Seitenfehler beim Demand Paging

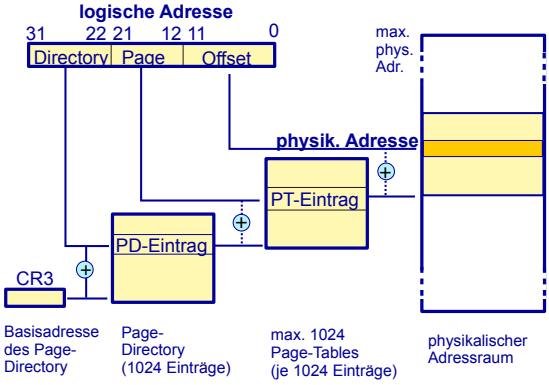
Demand Paging benötigt zwei Steuerbits in den Seitentabelleneinträgen:

- Das Present-Bit (P-Bit) gibt an, ob sich die Seite im Speicher befindet.
- Das Dirty-Bit (D-Bit) zeigt an, ob die Seite verändert wurde.



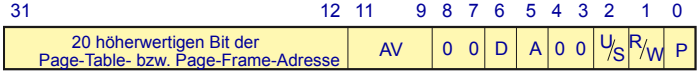
Paging in der IA-32-Architektur

- Der virtuellen Adressraum von 4G Bytes wird in 4K Bytes große Seiten eingeteilt.
- Ein Seitentabelleneintrag ist 4 Bytes lang.
- Bei einstufiger Adressabbildung würde die Seitentabelle 4M Bytes beanspruchen.
- Die Adressumsetzung erfolgt aus diesem Grund zweistufig.



Tabelleneinträge in der IA-32-Architektur

- Page-Directory- und Page-Table-Einträge haben einen identischen Aufbau.
- Seitentabellen können selbst mit Demand Paging nachgeladen werden.



Erläuterung:

AV - frei verfügbar für BS
D - Dirty
A - Accessed
U/S - User/Supervisor
R/W - Read/Write
P - Present

falls P=0 , sind die Bits 31-1 frei verfügbar

- Das Present-Bit (P-Bit) kennzeichnet, ob die Page Table bzw. die Seite sich augenblicklich im Speicher befindet.
- Das Dirty-Bit (D-Bit) wird bei jedem Schreibzugriff von der Hardware gesetzt.
- Das Accessed-Bit (A-Bit) wird bei jedem Zugriff auf die Page Table bzw. die Seite von der Hardware gesetzt.
- Das User/Supervisor-Bit (U/S-Bit) definiert in Verbindung mit dem Read/Write-Bit (R/W-Bit) die Schreib-/Leserechte für ein Benutzerprogramm und das Betriebssystem.
- Die AV-Bits sind frei verfügbar für das Betriebssystem.

Speicherschutz bei virtuellem Speicher

- Da eine Umgehung der Adressumsetzung nicht möglich ist, hat der Benutzer keine Möglichkeiten, gezielt auf bestimmte physikalische Adressen zuzugreifen.
- Die Seitentabellen werden durch das Betriebssystem eingerichtet.
- Eine Manipulation der Seitentabellen ist nicht möglich, da sich diese im Systemadressraum befinden.
- Eine Überadressierung der Seitentabelle (ungültige Seiten-Nr.) muss mit Hilfe eines Seitentabellen-Längenregisters, welches die Anzahl der Einträge enthält, ausgeschlossen werden.

Speicherschutz bei der IA-32-Architektur

- Im Falle der IA-32-Archttektur haben Page-Directory und Page-Tabellen eine feste Länge (1024 Einträge).
- Ungültige Einträge sind durch ein nicht gesetztes P-Bit in Verbindung mit einer speziellen Bitkombination in den AV-Bits als solche gekennzeichnet.
- Eine Adressierung über einen ungültigen Tabelleneintrag führt zu einer Page-Fault-Exception.

Seitentabellen und Seitenrahmentabellen

- Seitentabellen dienen zur Adressabbildung.
- Struktur und Inhalt sind prozessorspezifisch.
- Für die Verwaltung des Arbeitsspeichers unterhält das Betriebssystem eine Seitenrahmentabelle zur Verwaltung der belegten und der freien Seitenrahmen.
- Die Seitenrahmentabelle ist quasi eine invertierte Seitentabelle.

Seitentabellen

Prozess 1

Seiten-Nr.	Rahm.-Nr.
0	3
1	-
2	2
3	5

Prozess 2

Seiten-Nr.	Rahm.-Nr.
0	4
1	-

Prozess 3

Seiten-Nr.	Rahm.-Nr.
0	7
1	0
2	-

Invertierte Seitentabelle

Rahm.-Nr.	Prozess	Seiten-Nr.
0	3	1
1	-	-
2	1	2
3	1	0
4	2	0
5	1	3
6	-	-
7	3	0

Belegungsstrategien

Lokale Belegungsstrategie

- Jedem Prozess wird ein bestimmtes Kontingent an Seitenrahmen zugebilligt. Die Größe des Kontingents ist abhängig von der Größe des virtuellen Adressraums.
- Ist das Kontingent erschöpft, so muss der Prozess im Falle eines Seitenfehlers einen prozesseigenen Seitenrahmen überlagern.

Globale Belegungsstrategie

- Die Anzahl der Seitenrahmen eines Prozesses ist grundsätzlich nicht limitiert.
- Im Falle eines Seitenfehlers werden alle Seitenrahmen für eine Überlagerung in Betracht gezogen. Ein Prozess kann einem anderen Prozess eingelagerte Seiten entziehen.
- Ein Speicherengpass kann im Extremfall in einen Thrashing-Zustand führen, in dem das Rechnersystem ausschließlich mit der Ein-und Auslagerung von Seiten beschäftigt ist.
- Eine ständige Überwachung des Thrashing-Zustands, z.B. durch eine Überwachung der Seitenfehlerrate, ist erforderlich.

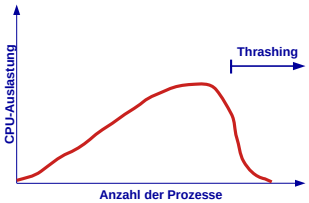
Thrashing

Thrashing

Ausgelagerte Seite wird gleich wieder angesprochen. Die Prozesse verbringen mehr Zeit mit dem Warten auf das Beheben von Seitenfehlern als mit der eigentlichen Ausführung.

Ursachen

- Prozess ist nahe am Seitenminimum
- Zu viele Prozesse gleichzeitig im System
- Schlechte Ersetzungsstrategie



Lösungen

- Lokale Belegungsstrategie (kein Thrashing zwischen Prozessen)
- Überwachung der Seitenfehlerrate (Seitenfehler pro Zeiteinheit)
- Begrenzung der Prozessanzahl

Ersetzungsstrategien

Optimale Ersetzungsstrategie:

- Lagere die Seite aus, deren nächster Zugriff am weitesten in der Zukunft liegt.
- Implementierung ist praktisch unmöglich. Die optimale Seitenersetzung kann jedoch zum Vergleich bzw. zur Bewertung einer Ersetzungsstrategie verwendet werden.

Gefordert sind daher Ersetzungsstrategien, welche das Idealverhalten möglichst gut approximieren und gleichzeitig implementierungsfähig sind. Für die nahe Zukunft wird das gleiche Verhalten wie in der nahen Vergangenheit angenommen.

Lokalität von Programmen

Innerhalb eines vorgegebenen Zeitintervalls wird nur eine bestimmte Teilmenge der Seiten eines Programms angesprochen.

- Instruktionen werden meist stückweise sequentiell ausgeführt.
- Programmschleifen erstrecken sich meist über wenige Seiten.
- Referenzierte Datenstrukturen sind häufig zusammenhängend.

Seitenersetzung bei globaler Belegungsstrategie

- Bei globaler Belegungsstrategie wird die zu ersetzende Seite unter der Gesamtheit der eingelagerten Seiten aller Prozesse ausgewählt.
- Der Aufwand zum Durchsuchen der Seitentabellen aller Prozesse wäre zu aufwendig.
- Sehr viel effizienter ist es, in der Seitenrahmentabelle nach einem Ersetzungskandidaten zu suchen.
- Wenn die Seitenrahmentabelle invertiert angelegt ist, ist der Zugriff auf die von der Hardware unterstützten Reference-Bits möglich.

FIFO / Round-Robin

Einfachstes Verfahren, die älteste Seite im Hauptspeicher wird ersetzt. Das Verfahren berücksichtigt nicht die Lokalität von Programmen.

Seitenzugriff	1	2	3	4	1	2	5	1	2	3	4	5
Speicher inhalt												
page fault?												

FIFO-Anomalie (Bélády's Anomalie, 1969): Zähle die Page Faults der gegebenen Seitenzugriffsfolge jeweils mit einem Hauptspeicher der Größe 3 und 4 Seiten.

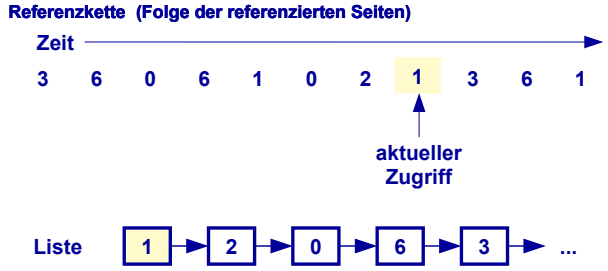
Least-Recently-Used LRU

- Es wird die Seite zur Überlagerung ausgewählt, auf die am längsten nicht mehr zugegriffen wurde. (Lokalität von Programmen)
- LRU ist ein *Stack-Algorithmus*, d.h. bei k Seitenrahmen sind die eingelagerten Seiten immer eine Untermenge der Seiten, die bei k+1 Seitenrahmen zum gleichen Zeitpunkt eingelagert wären! Bei solchen Algorithmen kann keine Anomalie auftreten.

Seitenzugriff	1	2	3	4	1	2	5	1	2	3	4	5
Speicher inhalt												
page fault?												

Least-Recently-Used LRU

Typische Implementierung mit verketteter Liste



Problem

- Es muss jeder Speicherzugriff berücksichtigt werden
- Hoher Speicherplatzbedarf, viele zusätzliche Speicherzugriffe
- Implementierung von LRU nicht ohne Hardwareunterstützung möglich

Second-chance / Clock-Algorithmus

Ziel der Second-Chance-Strategie ist es LRU anzunähern und dabei die Einfachheit von FIFO beizubehalten.
Einsatz von Reference-Bits

- Der Second-Chance-Strategie benötigt ein Reference-Bit in den Seitentabelleneinträgen, welches bei jedem Speicherzugriff von der Hardware gesetzt wird.
- Moderne Prozessoren bzw. MMUs unterstützen Reference-Bits (bei Intel: Accessed-Bit)

Strategie

- Bei einer frisch eingelagerten Seite wird das Referenzbit zunächst auf 1 gesetzt.
- Wenn die Seite ersetzt werden soll (z.B. durch FIFO) wird das Referenzbit abgefragt
  - R = 1 die Seite übersprungen und das Reference-Bit zurückgesetzt.
  - R = 0 die Seite wird ersetzt
- Der *Clock-Algorithmus* ist eine Effiziente implementierung der Second-Chance-Strategie: Ein Zeiger in einem Ringpuffer zeigt auf die älteste Seite

Second Chance (Beispiel)

Seitenzugriff	1	2	3	4	1	2	5	1	2	3	4	5
Speicher-	$\rightarrow 1_1$	1_1										
inhalt,	$-_0$	$\rightarrow 2_1$										
Umlaufzeiger,	$-_0$	$-_0$										
Referenzbit												
page fault?												

Bemerkung:
Bei Second Chance kann es auch zur FIFO Anomalie kommen: Wenn alle Referenzbits gleich 1, wird nach FIFO entschieden

Kapitel VI

Dateisysteme

Dateistruktur

- Für Windows- und Linux-Betriebssysteme besteht eine Dateien aus einer unstrukturierten Folge von Bytes.
- Die byteorientierte Dateistruktur passt sowohl auf die Information einer Plattendateien als auch auf die Zeichenfolgen in Ein-/Ausgabeströmen von und nach zeichenorientierten Geräten

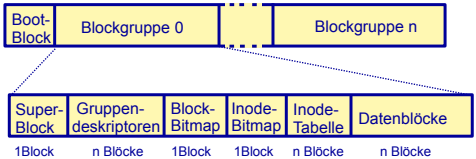
Dateisysteme

Das Linux-Dateisystem ext3

Das virtuelle Dateisystem in Linux

Das `proc`-Dateisystem in Linux

Grundaufbau von ext3



- Eine Partition ist in Blöcke gleicher Länge (wahlweise 1K, 2K oder 4K) aufgeteilt.
- Der erste Block ist reserviert für den Boot-Block.
- An den Boot-Block schließt sich eine Folge von Blockgruppen an. Die Anzahl ist abhängig von der Größe der Partition.
- Die erste Blockgruppe enthält den primären Superblock. Aus Redundanzgründen gibt es Kopien in weiteren Blockgruppen.
- Die Gruppendeskriptortabelle enthält die Deskriptoren aller Blockgruppen.
- Zwei nachfolgende Bitvektoren geben die Belegung der Inode-Liste bzw. die Belegung der Datenblöcke an.
- Die Inode-Liste enthält Einträge (Inodes) mit Verwaltungsdaten für jede in der Blockgruppe befindliche Datei.

Superblock

- Der Superblock enthält wichtige Kenngrößen des Dateisystems.
- Hierzu gehören die Blockgröße, die Anzahl der Inodes, die Anzahl der Datenblöcke, der Status des Dateisystems, die Anzahl der mount-Operationen seit der letzten Überprüfung.

dumpe2fs /dev/sda2

```
Filesystem volume name:   <none>
Last mounted on:          /boot
Filesystem UUID:          95963c54-5f06-4c34-a007-926787d7ffa9
Filesystem magic number:  0xEF53
Filesystem revision #:    1 (dynamic)
Filesystem features:      has_journal ext_attr resize_inode
↪  dir_index filetype needs_recovery extent 64bit flex_bg
↪  sparse_super large_file huge_file dir_nlink extra_isize
↪  metadata_csum
Filesystem flags:         signed_directory_hash
Default mount options:    user_xattr acl
Filesystem state:         clean
Errors behavior:          Continue
Filesystem OS type:       Linux
Inode count:              124928
Block count:              499712
Reserved block count:     24985
Free blocks:              97780
Free inodes:              124596
First block:              1
Block size:               1024
Fragment size:            1024
Group descriptor size:    64
Reserved GDT blocks:      256
Blocks per group:         8192
Fragments per group:      8192
Inodes per group:         2048
Inode blocks per group:   256
Flex block group size:    16
Filesystem created:       Wed May 31 19:49:17 2017
Last mount time:          Sun Oct  7 17:29:57 2018
Last write time:          Sun Oct  7 17:29:57 2018
Mount count:              113
Maximum mount count:      -1
Last checked:             Wed May 31 19:49:17 2017
Check interval:           0 (<none>)
Lifetime writes:          14 GB
Reserved blocks uid:      0 (user root)
Reserved blocks gid:      0 (group root)
First inode:              11
Inode size:                  128
Journal inode:            8
Journal size:             8M
...
```

Gruppendeskriptoren

- Die Gruppendeskriptoren beinhalten Informationen
  - zur relativen Lage der Bitmaps und der Inode-Tabelle innerhalb einer Gruppe
  - zur Anzahl der freien Blöcke und der freien Einträge in der Inode-Liste
  - zur Anzahl der in der Gruppe befindlichen Verzeichnisse.
- Die Nummerierung der Inodes und Blöcke ist innerhalb eines Dateisystems eindeutig.

dumpe2fs /dev/sda2 (cont.)

```
Group 0: (Blocks 1-8192) csum 0xc7ad [ITABLE_ZEROED]
  Primary superblock at 1, Group descriptors at 2-5
  Reserved GDT blocks at 6-261
  Block bitmap at 262 (+261), csum 0x4f0afd26
  Inode bitmap at 278 (+277), csum 0x728a8b12
  Inode table at 294-549 (+293)
  3787 free blocks, 2003 free inodes, 2 directories, 1997 unused inodes
  Free blocks: 4406-8192
  Free inodes: 39, 41-44, 51-2048
Group 1: (Blocks 8193-16384) csum 0x2ee3 [INODE_UNINIT, ITABLE_ZEROED]
  Backup superblock at 8193, Group descriptors at 8194-8197
  Reserved GDT blocks at 8198-8453
  Block bitmap at 263 (bg #0 + 262), csum 0xa5596884
  Inode bitmap at 279 (bg #0 + 278), csum 0x00000000
  Inode table at 550-805 (bg #0 + 549)
  3595 free blocks, 2048 free inodes, 0 directories, 2048 unused inodes
  Free blocks: 8454, 8457-8704, 8708-10240, 12289-13698, 14240-14336, 16079-16384
  Free inodes: 2049-4096
...
```

Inode-Liste und Datenblöcke

- Die Inode-Liste (kurz: Ilist) hat eine feste Länge. Diese Länge wird bei der Anlage des Dateisystems festgelegt.
- Die Länge der Inode-Liste bestimmt die maximale Anzahl der in der Gruppe gespeicherten Dateien.
- Jeder Listeneintrag ist in der Inode-Bitmap positionsbedingt durch ein Bit vertreten, welches den Belegungszustand des Listeneintrags kennzeichnet.
- Der Bitvektor wird für die Suche nach einem freien Eintrag in der Inode-Liste verwendet.
- Die Belegung der Datenblöcke wird durch das Block-Bitmap verwaltet.
- Die Beschränkung des Block-Bitmap auf einen Block bestimmt die maximale Länge einer Blockgruppe (bei 1K-Blöcken bis zu 8192 Datenblöcke).
- Die maximale Größe einer Blockgruppe bestimmt die Anzahl der Blockgruppen auf der Partition.

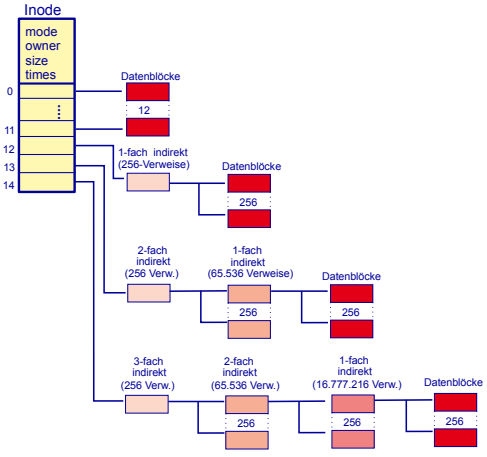
Inodes

Die Verwaltungsinformation zu einer Datei wird in einem Inode gespeichert.

```
struct ext3_inode {
  __u16   i_mode;                /* File mode */
  __u16   i_uid;                /* Low 16 bits of Owner Uid */
  __u32   i_size;                /* Size in bytes */
  __u32   i_atime;                /* Access time */
  __u32   i_ctime;                /* Creation time */
  __u32   i_mtime;                /* Modification time */
  __u32   i_dtime;                /* Deletion Time */
  __u16   i_gid;                /* Low 16 bits of Group Id */
  __u16   i_links_count;        /* Links count */
  __u32   i_blocks;                /* Blocks count */
  __u32   i_flags;                /* File flags */
  union   osd1;                /* OS dependent 1 */
  __u32   i_block[EXT3_N_BLOCKS];        /* Pointers to blocks */

  //...
};
```

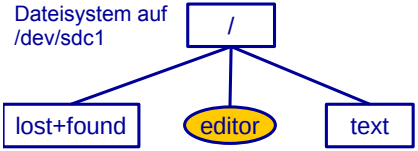
Blockverweise in Inodes

- Die Strukturkomponente `i_block` enthält einen 15-elementigen Array mit Verweisen auf Datenblöcke.
- 12 Einträge mit direkten Adressen von Datenblöcken.
- Der 13. Eintrag verweist auf einen Block mit indirekten Verweisen.
- Der 14. Eintrag verweist auf einen Block mit 2-fach indirekten Verweisen.
- Der 15. Eintrag verweist auf einen Block mit 3-fach indirekten Verweisen.



Dateiverzeichnisse

- Ein Verzeichnis (Directory) ist eine spezielle Datei.
- Es enthält eine Liste mit Einträgen für die in dem Verzeichnis befindlichen Dateien und Unterverzeichnisse.
- Ein Eintrag besteht aus dem Dateinamen sowie der zugehörigen Inumber.




```
struct ext4_direntry {
        uint32_t inode;
        uint16_t entry_length;
        uint8_t name_length;
        uint8_t inode_type;
        uint8_t name[255];
};
```

Byte	*inode*	*rec_len*	*name_len*	*name*												
0	2	12	1	.	\0	\0	\0									
12	2	12	2	.	.	\0	\0									
24	11	20	10	l	o	s	t	+	f	o	u	n	d	\0	\0	
44	12	16	6	e	d	i	t	o	r	\0	\0					
60	28449	12	4	t	e	x	t									

Links

Harte Links

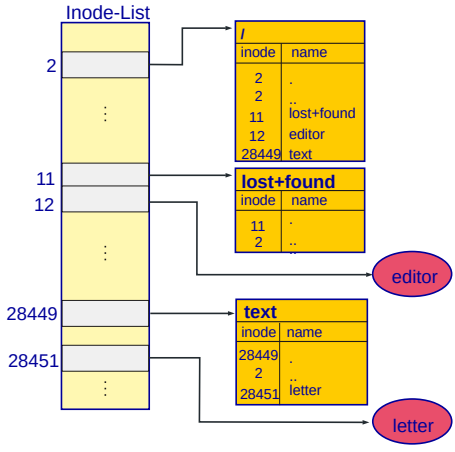
- Im Falle eines harten Links enthalten zwei Verzeichniseinträge den gleichen Inode-Eintrag.
- Original und Kopie sind nicht zu unterscheiden.
- Beim Löschen eines harten Link wird lediglich der Link-Zähler im zugehörigen Inode dekrementiert.
- Die Datei wird erst gelöscht, wenn der Link-Zähler den Wert Null erreicht hat.

Symbolische Links

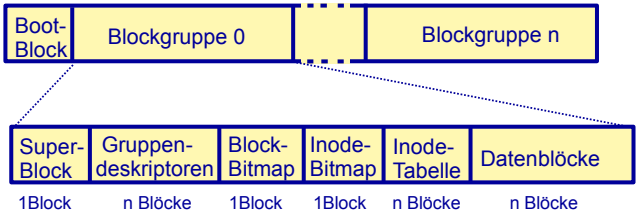
- Symbolische Links sind spezielle Datei mit einem eigenen Inode.
- Die „Linkdatei“ enthält den Pfadnamen des Links.
- Falls das Ziel des Links gelöscht wird, verweist der Links ins Leere.
- Im Gegensatz zu harten Links können symbolische Links auch über Dateisystemgrenzen eingesetzt werden.

Lookup eines Pfadnamens

- Beim Öffnen einer Datei taucht das Problem auf, den Inode zu einem gegebenen Pfadnamen zu ermitteln.
- Hierzu muss die Verzeichnishierarchie schrittweise durchsucht werden.



Dateitransaktionen in ext3



- Die Erzeugung einer neuen Datei oder die Veränderung an einer bestehenden Datei tangiert eine Reihe von Blöcken des ext3-Dateisystems
  - Es muss ggf. ein freier Inode gefunden werden (Inode-Bitmap)
  - Der Inode wird angelegt (Inode-Tabelle)
  - Es müssen freie Blöcke gefunden werden (Block-Bitmap)
  - Daten werden geschrieben (Datenblöcke)
  - Dies alles bewirkt Veränderungen im Superblock
- Die Transaktion ist erst erfolgreich, wenn alle Veränderungen umgesetzt wurden.
- Im Falle eines Systemabsturzes während einer Transaktion ist das Dateisystem inkonsistent.

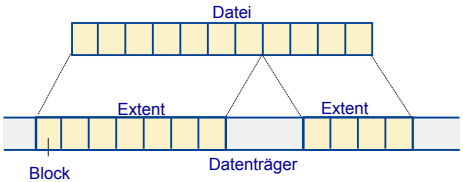
Journalfunktion in ext3

Ein Journal ermöglicht es, ein Dateisystem nach einem Systemausfall schneller in einem konsistenten Zustand zurück zu führen.

- Veränderte Blöcke werden zunächst in Form einer Transaktion in das Journal geschrieben. Nach Abschluss des Datentransfers werden die Daten als *committed* markiert.
- Anschließend findet die Übertragung des Blocks in das eigentliche Dateisystem statt. Nachdem auch dieser Transfer erfolgreich beendet wurde, wird der Journaleintrag gelöscht.
- Beim Hochlauf des Systems nach einem Absturz wird zunächst das Journal gelesen. Committete Transaktionen werden jetzt in das Dateisystem übertragen.
- Nicht committete Transaktionen werden verworfen. In diesem Fall sind die Daten im Dateisystem noch konsistent. Die Veränderungen gehen jedoch verloren.

Extends in ext4

- ext4 benutzt 48 Bit große Blocknummern (ext3 hatte 32 Bit) und unterstützt so größere Partitionen oder Volumes.
- ext4 verwendet ein gegenüber ext3 verändertes Schema zur Verwaltung der Datenblöcke einer Datei.



- Mehrere Datenblöcke werden in zusammenhängenden Bereichen, den sogenannten Extents platziert.
  - Die Position und die Länge der Extents in der Datei sowie deren Blockadressen auf dem Datenträger sind im Inode vermerkt.
  - Voraussetzung für die gebündelte Speicherung in Extents ist eine verzögerte Allokation der Datenblöcke (delayed allocation). Inhaltliche Änderungen an einer Datei werden im Arbeitsspeicher gecached und später in Form eines Extents auf das Speichermedium übertragen

Dateisysteme

Das Linux-Dateisystem ext3

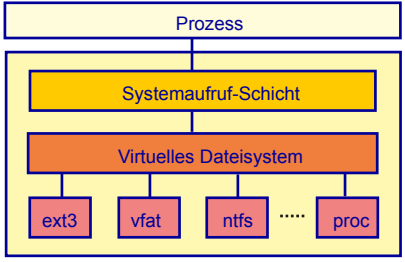
Das virtuelle Dateisystem in Linux

Das `proc`-Dateisystem in Linux

Das virtuelle Dateisystem (VFS)

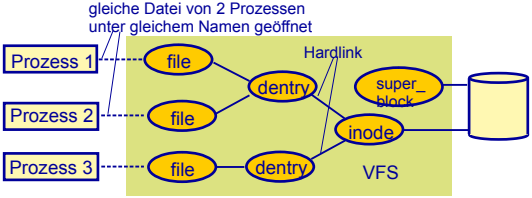
Linux unterstützt mehrere Dateisystem-Typen (z.B. FAT- oder ntfs-Dateisysteme).

```
mount -t vfat /dev/sdc1 /mnt/flash
cp /mnt/flash/file.txt /home/bjm/file.txt
```



Objekte des VFS (1)

- VFS weist eine „objektorientierte“ Architektur auf.
- VFS-Objekte spiegeln den Grundaufbau eines Unix-typischen Dateisystemen wieder.
- Jedes dieser Objekte enthält ein so genanntes Operations-Objekt. Dieses ist eine Struktur mit Zeigern auf Funktionen, die von den konkreten Datei-systemen implementiert werden müssen.



Objekte des VFS (2)

Superblock-Objekt

- Für jedes gemountete Dateisystem existiert ein Superblock-Objekt (Typ: `struct super_block`).
- Es enthält Angaben, welche vergleichbar sind mit den Superblock-Daten eines konkreten Dateisystems (z.B. Blockgröße, Mount-Verzeichnis).
- Beim mount eines Dateisystems wird ein Superblock-Objekt erzeugt und mit Information aus dem Superblock des realen Dateisystems gefüllt

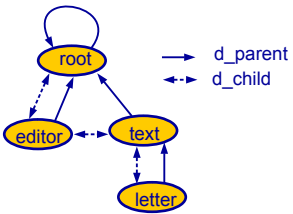
Inode-Objekt

- Das Inode-Objekt (Typ: struct inode) enthält die Informationen zur Handhabung von Dateien oder Verzeichnissen.
- Im Falle von UNIX-typischen Dateisystemen entstammt die Information aus dem entsprechenden Platten-Inode.
- Inode-Objekte werden bei Bedarf (Öffnen einer Datei, Durchsuchen eines Verzeichnisses) im Speicher erzeugt.
- Falls das Inode-Objekt ein Verzeichnis beschreibt, enthält es eine Liste von Dentry-Objekten, welche die Directory-Einträge repräsentieren.

Objekte des VFS (3)

Dentry-Objekt

- Ein Dentry-Objekt (Typ: struct dentry) repräsentiert einen Directory-Eintrag.
- Ein Pfadname /text/letter wird durch drei Dentry-Objekte beschrieben.
- Beim Lookup des Pfadnamens werden die Objekte im Speicher angelegt. Ein Dentry-Objekt enthält u.a. den Namen des Pfadbestandteils, sowie einen Zeiger auf das zugehörige Inode-Objekt.



Objekte des VFS (4)

File-Objekt

- Ein File-Objekt (Typ: struct file) repräsentiert eine geöffnete Datei.
- Es wird beim Öffnen einer Datei durch einen Prozess erzeugt und spiegelt die individuelle Sicht (z.B. Zugriffsmodus, Schreib-/Lesezeiger) des Prozesses auf die Datei wieder.
- Die Strukturkomponenten `f_mode` und `f_pos` enthalten die betreffenden Angaben.

Das zugehörige Operationsobjekt enthält u.a. die Operationen zum Lesen und Schreiben.

```
ssize_t (*read) (struct file*, char *, size_t, loff_t*);
ssize_t (*write) (struct file*, const char *,
                                        size_t, loff_t*);
```

Prozess-spezifische Datenstrukturen

- Der Zeiger files im Prozessdeskriptor verweist auf eine Struktur vom Typ struct `files_struct`, welche die Verbindung des Prozesses zu den geöffneten Dateien repräsentiert.
- Die Struktur enthält den 32-elementigen Array `fd_array` mit Zeigern zu den dem Prozess zugeordneten File-Objekten.

```
struct files_struct {
  atomic_t count;
  spinlock_t file_lock; /* Protects all the
                           below members */
  int max_fds;
  int max_fdset;
  int next_fd;
  struct file ** fd;   /* current fd array */
  fd_set *close_on_exec;
  fd_set *open_fds;
  fd_set close_on_exec_init;
  fd_set open_fds_init;
  struct file *fd_array[NR_OPEN_DEFAULT];
};
```

Dateisysteme

Das Linux-Dateisystem ext3

Das virtuelle Dateisystem in Linux

Das `proc`-Dateisystem in Linux

Das /proc-Dateisystem

- Das /proc-Dateisystem gibt Auskunft über Parameter des Kernels, Attribute des Rechners sowie den Status der Prozesse.
- Aus Sicht des Benutzers sieht das /proc-Dateisystem wie jedes andere Dateisystem aus. Man kann sich mit cd darin bewegen, Verzeichnisinhalte mit ls anzeigen lassen und den Inhalt von Dateien mit cat betrachten.
- Der Kernel fängt die Zugriffe auf das /proc-Dateisystem ab und erzeugt die diesbezüglichen Verzeichnis- und Dateiinhalte durch Auslesen von Kernel- Parametern.

```
ls /proc
```

```
.       2834    4337    5322    cpuinfo     iomem    net
..      2993    4340    5330    devices     ioports  partitions
1       4063    4348    5522    dma         irq      pci
10      4095    4350    5534    driver      kcore    scsi
172     4140    4353    5560
                         ..............

2677    4250    5070    bus     filesystems modules  sys
2910    4251    5148    cmdline interrupts  mounts   version
```

Implementierung des /proc-Dateisystems

- Kern der Implementierung des /proc-Dateisystems ist die Definition einer statischen Verzeichnisstruktur und die Zuordnung von festen I-Nummern zu den Knoten des Dateisystems.
- Der Kernel verwendet diese I-Nummer als Schlüssel für die zu generierende Information.
- Im Falle von prozessspezifischen Verzeichnissen enthält die 32 Bit lange Inumber in den höherwertigen Bitstellen die 16-Bit lange Prozessnummer. Die niederwertigen 16 Bit identifizieren die untergeordneten Knoten des Prozessverzeichnisses.
- Die Inumber von systemspezifischen Verzeichnissen hat in den höherwertigen Bitstellen den Wert 0. Die niederwertigen Bits kennzeichnen den betreffenden Dateisystemknoten.
- Die Implementierung der /proc-spezifischen VFS-Operationen sammeln die gewünschte Information in den Kernel-Datenstrukturen, wandeln sie in Textform um und platzieren den Text in dem Eingabepuffer des Prozesses.

Kontroll-Fragen

1. Worin liegt der Nutzen von mehreren Blockgruppen im ext3-Dateisystem?
2. Ist ein Dateisystem bei Verlust der Block-Bitmap noch zu retten? Begründen Sie Ihre Aussage.
3. Wie viele Plattenzugriffe sind notwendig für den Lookup des Inodes von `/home/bjm/data`? Gehen Sie davon aus, dass sich der Inode des root-Verzeichnisses im RAM befindet und dass jedes Verzeichnis in einen Block passt.
4. Wo befindet sich das Journal eines Journaling-Dateisystems? Im Arbeitsspeicher oder auf Peripherspeicher?
5. Welchen Nutzen hat das VFS?
6. Können Sie sich vorstellen, warum die Objektstruktur von VFS sehr auf die Struktur eines UNIX-typischen Dateisystems ausgerichtet ist?

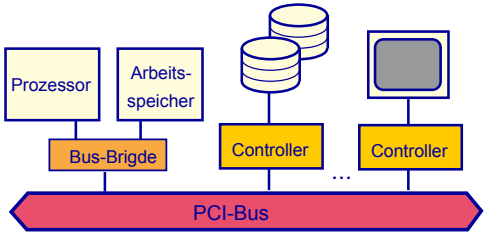
Kapitel VII

Ein-/Ausgabe

Aufgaben

- Bereitstellung von “Datenwegen” zwischen Arbeitsspeicher und den E/A- Geräten
  - exklusiv von einem Prozess genutzt
  - im Multiplex-Verfahren von mehreren Prozessen genutzt
- Anpassung von
  - Geschwindigkeit und
  - Datenformat zwischen Prozessor/Hauptspeicher und diversen E/A-Geräten
- Bereitstellung einer
  - einfach zu bedienenden
  - möglichst für alle Geräte einheitliche E/A-Schnittstelle.

Ein-/Ausgabe



Ein-/Ausgabegeräte werden über Controller an den Systembus angebunden. Die Programmierung erfolgt über E/A-Register auf den Controllern.

Geräteklassen

Die wesentlichen Klassen sind zeichenorientierte Geräte (character devices) und Blockorientierte Geräte (block devices).

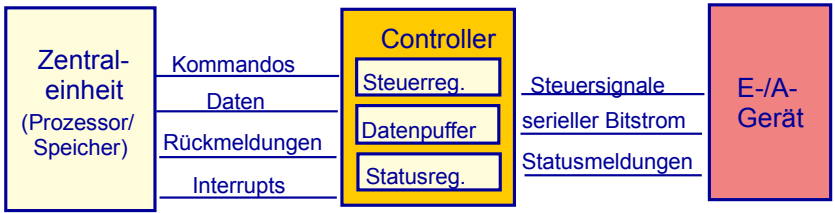
- Zeichenorientierte Geräte
  - Dienen zur Kommunikation zwischen Benutzer und Rechner sowie zur Datenkommunikation zwischen Rechnern.
  - Zeichenorientierte Geräte akzeptieren oder erzeugen einen Zeichenstrom; d.h. die elementare, übertragbare Informationseinheit ist ein Zeichen im Sinne eines bestimmten Codes (z.B. ASCII).
  - Meist rein sequentieller Zugriff, selten wahlfreie Positionierung
  - Tastatur, Drucker, Modem, Maus, ...
- Blockorientierte Geräte
  - Dienen zur langfristigen Speicherung von Information.
  - Die Information ist in Sektoren gespeichert (Größe 512 Bytes bis 4 K Bytes). Beim Lesen bzw. Schreiben wird jeweils ein ganzer Sektor übertragen.
  - Meist wahlfreier blockweiser Zugriff (random access)
  - Festplatte, Diskette, CD-ROM, DVD, Bandlaufwerke, ...

Grafikkarten und Netzwerkkarten passen weniger in dieses Schema.

Geräte-Controller

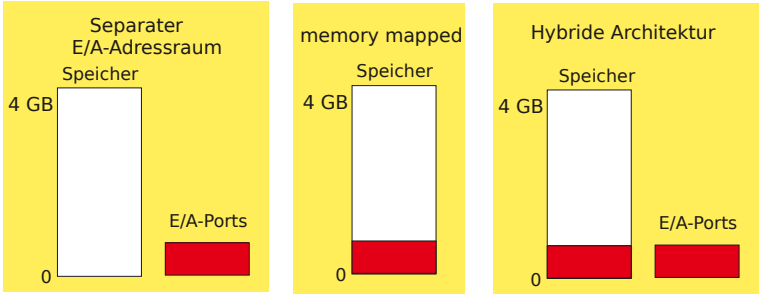
- Geräte-Controller sind eigenständige Hardware-Einheiten, welche Daten zwischen einem (oder mehreren) E/A-Geräten und dem Prozessor bzw. Arbeitsspeicher transferieren.
- Je nach Gerätetyp befindet sich der Controller auf einer Schnittstellenkarte oder im E-/A-Gerät selbst.

Informationsfluss zwischen Zentraleinheit, Controller und E-/A-Gerät



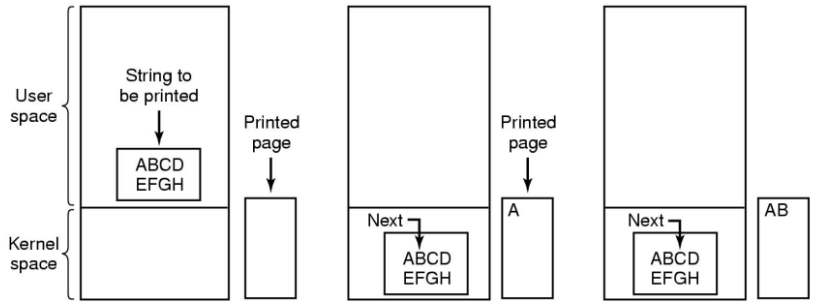
E-/A-Adressräume

- Der Gerätetreiber kommuniziert mit dem Controller über Steuer- und Datenregister.
- Die Adressen dieser Register (E/A-Ports) können entweder Teil des physikalischen Adressraums sein (memory mapped I/O) oder einen speziellen E/A-Adressraum bilden.
- Bei einem eigenen E-/A-Adressraum sind spezielle E/A-Befehle erforderlich.
- Auch hybride Architekturen sind möglich (siehe IA-32).



Arbeitsweise von Gerätetreibern

- Je nach Fähigkeiten des Geräts erfolgt E/A mittels
  - Polling (oder „Programmierte E/A“),
  - Unterbrechungen oder
  - DMA
- Beispiel: Drucken eines Strings



Quelle: Tanenbaum, „Modern Operating Systems“

Polling-Modus

Im Polling-Modus fragt der Prozessor wiederholt die Statusregister des Controllers ab. Bei erfolgreichem Transfer überträgt er die Daten aus dem Datenregister an die gewünschte Stelle im Arbeitsspeicher.

Pseudocode einer Betriebssystemsfunktion

```
/* Zeichen in Puffer kopieren */
copy_from_user (buffer, p, count);
/* Schleife über alle alle Zeichen */
for(i=0;i<count;i++) {
    /* Warte bis Drucker bereit */
    while (*printer_status_reg!= READY);
    /* Ein Zeichen ausgeben */
    *printer_data_reg = p[i];
}
return_to_user();
```

Interrupt-Modus

Im Interrupt-Modus löst der Controller beim Abschluss eines E-/A-Vorgangs einen Interrupt aus, woraufhin die Daten aus dem Datenregister gelesen werden können.

Initiieren der E/A Operation

```
copy_from_user(buffer, p, count);
/* Druckerunterbrechungen erlauben */
enable_interrupts();
/* Warte bis Drucker bereit */
while(*printer_status_reg!=READY);
/* Erstes Zeichen ausgeben */
*printer_data_reg = p[0];
i++;
scheduler();
return_to_user();
```

Unterbrechungs-Behandlungs-Routine

```
if (count> 0) {
  *printer_data_reg = p[i];
  count--;
  i++;
}
else {
  unblock_user();
}
acknowledge_interrupt();
return_from_interrupt();
```

DMA-Modus

Im DMA-Modus (Direct Memory Access) transferiert der Controller die Daten eigenständig an die im E-/A-Auftrag spezifizierte Adresse im Arbeitsspeicher und löst abschießend einen Interrupt aus.

Initiieren der E/A Operation

```
copy_from_user(buffer, p, count);
set_up_DMA_controller (p, count);
scheduler();
return_to_user();
```

Unterbrechungs-Behandlungs-Routine

```
acknowledge_interrupt();
unblock_user();
return_from_interrupt();
```

Unterbrechungen

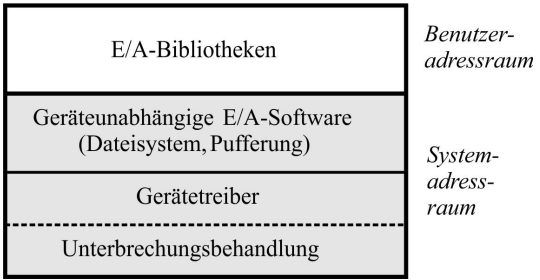
- Kontextsicherung
  - Wird teilweise von der CPU selbst erledigt, z.B. Statusregister und Rücksprungsadresse, aber nur das Minimum.
  - Alle veränderten Register müssen gesichert und am Ende der Behandlung wiederhergestellt werden.
- Unterbrechungssynchronisation
  - Während der Unterbrechungsbehandlung müssen weitere Unterbrechungen unterdrückt werden.
    - x86: sti, cli
    - Es droht der Verlust von Unterbrechungen.
  - Durch mehrstufige Behandlungen wird die Zeit für das harte Sperren von Unterbrechungen minimiert.
  - Top Half, Bottom Half: In einem typischen Szenario speichert die obere Hälfte die Daten vom Gerät möglichst schnell in einen gerätespezifischen Puffer, trägt ihre untere Hälfte in den Scheduler ein und endet. Das geht sehr schnell und nur für diesen Zeitraum müssen Unterbrechungen gesperrt werden.

Diskussion: DMA

- Caches
  - Heutige Prozessoren arbeiten mit Daten-Caches; DMA läuft am Cache vorbei!
  - Vor dem Aufsetzen eines DMA-Vorgangs muss der Cache-Inhalt in den Hauptspeicher zurückgeschrieben und invalidiert werden bzw. der Cache darf für die entsprechende Speicherregion nicht eingesetzt werden.
- Speicherschutz
  - Heutige Prozessoren verwenden eine MMU zur Isolation von Prozessen und zum Schutz des Betriebssystems; DMA läuft am Speicherschutz vorbei!
  - Fehler beim Aufsetzen von DMA-Vorgängen sind extrem kritisch.
  - Anwendungsprozesse dürfen DMA-Controller nie direkt programmieren!

E/A-Software

- Die Ein-/Ausgabe-Software ist in Form eines Schichtenmodells organisiert.
- Zur geräteunabhängige Schicht gehören das Dateisystem sowie Funktionen zur Pufferung von Ein-/Ausgabedaten auf Systemebene.
- Die geräteabhängige Schicht enthält einen Gerätetreiber für jeden unterstützten Gerätetyp.
- Ein zum Treiber gehörender Interrupt-Handler behandelt die vom Controller ausgelösten Interrupts.



Gerätetreiber

- Charakteristika von E/A-Geräten
  - Die Ein-Ausgabe findet blockweise oder zeichenweise statt.
  - Das Gerät unterstützt nur Lese-, nur Schreib- oder Schreib- und Lesezugriffe.
  - Die Schreib-/Lesezugriffe finden sequentiell oder random statt.
  - Die Transfergeschwindigkeit reicht von mehren Bytes pro Sekunde bis zu mehreren MByte pro Sekunde.
- Aufgaben von Gerätetreibern
  - Initialisierung des Geräts beim Systemstart
  - Umsetzung von geräteunabh. E/A-Aufträgen in gerätespez. E-/A-Aufträge
  - Steuerung des Geräts über den Geräte-Controller
  - Behandlung von E/A-Unterbrechungen
  - Handhabung von Gerätefehlern
- Einbinden von Gerätetreibern
  - Treiber für Nichtstandard-Schnittstellen müssen dynamisch in den Kern eingebunden werden.
  - Hierzu definieren die Betriebssysteme eine Standardschnittstelle, die jeder Treiber implementieren muss.

Linux-Geräteabstraktionen

Periphere Geräte werden als Spezialdateien repräsentiert

- Der Inode enthält eine eindeutige Identifikation des Gerätes durch ein 3-Tupel:
  - Geräteklasse (Block oder Character-Device)
  - Major-Devive-Number (Gerätetyp)
  - Minor-Device-Number (Instanz eines Gerätetyps)
- Geräte können bei gegeber Berechtigung wie Dateien mit Lese- und Schreiboperationen angesprochen werden
- Öffnen der Spezialdateien schafft eine Verbindung zum Gerät, die durch einen Treiber hergestellt wird
- Direkter Durchgriff vom Anwender auf den Treiber über dei Systemaufrufe read() bzw. write()

Gerätedateien

```
ls -l /dev
```

```
crw--w----  1 root     root    5,  1 Jan 17 16:54 console
brw-rw-rw-  1 root     disk    2,  0 Mai 23 2004  fd0
brw-rw----  1 root     disk    1,  0 Mai 23 2004  hda
brw-rw----  1 root     disk    1,  1 Mai 23 2004  hda1
crw-rw-rw-  1 root     lp      6,  0 Mai 23 2004  lp0
brw-rw----  1 root     disk    8,  0 Mai 23 2004  sda
brw-rw----  1 root     disk    8,  1 Mai 23 2004  sda1
crw--w----  1 root     tty     4,  1 Jan 17 16:54 tty01
crw--w----  1 root     tty     4,  2 Jan 17 16:54 tty02
|                              |   |              |
Geräteklasse                   |   |              Gerätename
                               |   Minor-Device-Number
                               Major-Device-Number
```

- Die Geräteklasse unterscheidet zwischen block- bzw. zeichenorientierten Geräten.
- Die Major-Device-Number identifiziert den Gerätetyp und damit den Gerätetreiber. Sie kann Werte zwischen 1 und 254 annehmen.
- Die Minor-Device-Number unterscheidet mehrere Geräte gleichen Typs.

Linux: Zugriffsprimitive

- `int open(const char *devname, int flags)` Öffnen“ eines Geräts. Liefert Dateideskriptor als Rückgabewert.
- `off_t lseek(int fd, off_t offset, int whence)` Positioniert den Schreib-/Lesezeiger – natürlich nur bei Geräten mit wahlfreiem Zugriff.
- `ssize_t read(int fd, void *buf, size_t count)` Einlesen von max. count Bytes in Puffer buf von Deskriptor fd.
- `ssize_t write(int fd, const void *buf, size_t count)` Schreiben von count Bytes aus Puffer buf auf Deskriptor fd
- `int close(int fd)` Schließen eines Geräts. Dateideskriptor fd kann danach nicht mehr benutzt werden.

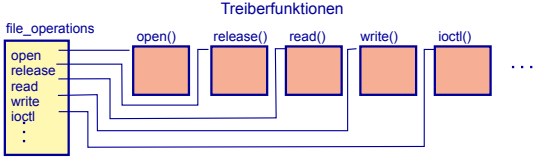
Um spezielle Geräteeigenschaften anzuspechen wird `int ioctl(int d, int request,...);` verwendet.

Registrierung eines Treibers

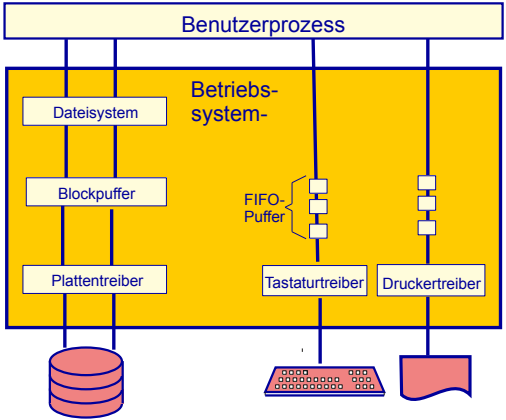
- Beim Laden eines Treibermoduls meldet sich dieser mit Hilfe der Funktion `register_chrdev()` bzw. `register_blkdev()` beim Kern an.

```
int register_chrdev(unsigned int major, const char *name,
                                        struct file_operations *fops);
```

  - major enthält die Major-Device-Number.
  - name gibt den Namen des Treiber an.
  - fops enthält eine Struktur vom Typ `struct file_operations` (siehe VFS) mit den Funktionszeigern auf die Treiberfunktionen.
- Beim Öffnen einer Gerätedatei wird ein VFS-FILE-Objekt erzeugt.
- Die File-Operationen werden durch die bei der Registrierung angegebenen Treiberfunktionen ersetzt.



Pufferung (1)

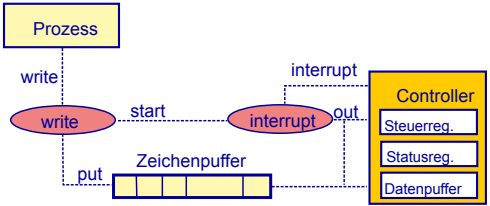

Benutzerprozess
Betriebs-
system-
Dateisystem
Blockpuffer
Plattentreiber
FIFO-
Puffer
Tastaturtreiber
Druckertreiber

Pufferung (2)

- Pufferung bei zeichenweiser E-/A
  - Ein Puffer ermöglicht die Anpassung der Ein-/Ausgabegeschwindigkeit der Anwenderprozesses an die Geschwindigkeit des jeweiligen Geräts.
- Pufferung bei blockweiser E-/A
  - Systemaufrufe zum Lesen und Schreiben betreffen häufig nur wenige Bytes. Physikalisch wird immer ein Block (meist mehrere Sektoren) gelesen.
  - Der Speicherbereich mit der Zieladresse eines über DMA abgewickelten E-/A-Vorgangs darf nicht ausgelagert werden.
  - Ein Depot von Puffern fungiert als Platten-Cache.
  - Ein einmal gelesener Puffer behält zunächst seinen Inhalt. Nachfolgende Lese-Zugriffe auf den im Puffer befindlichen Block erfordern keinen erneuten Zugriff auf das Gerät.
  - Schreibzugriffe auf einen Block bewirken zunächst lediglich eine Veränderung des Pufferinhalts.
  - Das Pufferdepot wird nach einer LRU-Strategie verwaltet.

Zeichenweise Ein-/Ausgabe

- Im Falle der zeichenorientierten Ein-Ausgabe leitet der Kern die Systemaufrufe an die Treiber-Routinen weiter.
- Der Treiber passt die Geschwindigkeit der Aus- bzw. Eingabe durch den Prozessor an die Geschwindigkeit des Geräts an.



- Wenn der Puffer eine maximale Füllhöhe erreicht hat, blockiert die write-Funktion den Prozess.
- Wenn die interrupt-Funktion den Puffer geleert hat, setzt sie den Anwenderprozess wieder bereit.

Blockpuffer-Cache

- Das Pufferdepot wird durch eine Menge von reservierten Seitenrahmen im Datenbereich des Kerns gebildet.
- Ein Puffer hat die Größe einen Blocks; d.h. bei einer Blockgröße von 1024 Bytes enthält ein Seitenrahmen 4 Puffer.
- Ein Puffer wird mit Hilfe einer Datenstruktur vom Typ `struct buffer_head` verwaltet. Dieser Puffer-Header enthält
  - die Adresse des Puffers
  - Major- und Minor-Device-Number des Geräts, dem der Puffer zugeordnet ist
  - die logische Blocknummer des Pufferinhalts
  - die Sektoradresse auf dem Datenträger
  - die Inumber der Datei, zu der der Pufferinhalt gehört
  - diverse Statusflaggen
- Puffer, für die ein E-/A-Vorgang anhängig ist, sind in eine Request-List eingekettet.
- Puffer, für die kein E-/A-Vorgang anhängig ist, sind in einer Free-List enthalten.

Blockweise Ein-/Ausgabe

- Falls ein Block gelesen oder geschrieben werden soll, wird zunächst über ein Hash-Verfahren nach einem Puffer mit dem Blockinhalt gesucht.
- Falls sich der Block im Pufferdepot befindet, kann der E-/A-Vorgang unmittelbar durch Lesen oder Verändern des Puffers erledigt werden. Bei Veränderung des Puffers erhält er den Status *dirty*.
- Andernfalls wird ein Puffer aus der Free-List entnommen und ein E-/A-Vorgang initiiert. Der Puffer wird hierzu in die Request-List eingekettet.
- Die Kernel-Threads bdflush und kupdate sorgen dafür, dass veränderte Puffer auf den Plattenspeicher zurück geschrieben werden.
- Mit Hilfe der Systemaufrufe sync() und fsync() kann ein Prozess den Transfer von veränderten Puffern veranlassen.

```
void sync(void);
int fsync(int fildes);
```

- sync() bewirkt den Transfer von allen veränderten Puffern.
- fsync() veranlasst den Transfer allen veränderten Puffern der Datei filedes.

Kontroll-Fragen

1. Worin besteht der Unterschied zwischen port-basiertem und memory-mapped I/O?
2. Warum werden Gerätetreiber nicht statisch in den Kernel-Code gelinkt?
3. Welche Bedeutung hat die Minor-Device-Number?

Kapitel VIII

Praktikum

Praktikum

Praktikum 1: Linux Shell

Praktikum 2+3: Software-Entwicklung unter Linux

Praktikum 4: Parallele Programme mit Threads

Praktikum 5: Semaphor-Bibliothek

Praktikum 6: MMAP

Praktikum 7: Message Queues

UNIX-Shell

- *Schale*, die den Betriebssystemkern umgibt.
  - Textbasierte Schnittstelle zum Starten von Kommandos.
  - Jedes ausgeführte Kommando ist ein Kindprozess
  - Normalerweise blockiert die Shell bis das Kommando terminiert
  - Kommandos können jedoch auch gestoppt, fortgesetzt, oder im Hintergrund ausgeführt werden (*job control*)
- Unix-Philosophie
  - Jedes Kommando erledigt nur eine einzelne überschaubare Aufgabe.
  - Ein Kommando liest die Eingabedaten von der Konsoleneingabe und schreibt die Ausgabe in die Konsole.
  - Komplexere Aufgaben können gelöst werden, indem die einzelnen Kommandos mittels Pipes kombiniert werden.
- Beispiel
  - `head -n` gibt die ersten n Zeilen der Eingabe/Datei aus.
  - `tail -n` gibt die letzten n Zeilen der Eingabe/Datei aus.
  - `head -17 datei | tail -1` gibt Zeile 17 in der Datei `datei` aus.

Shell-Kommandos

Ein Shell-Kommando besteht syntaktisch aus

- Kommandonamen
- Kommando-Optionen (typischerweise mit vorangestelltem Minuszeichen, im Beispiel `-l`)
- Parameter (z.B. Dateien, Programme)

Die Bestandteile eines Kommandos sind durch Leerzeichen getrennt.

Example

ls	Ausgabe einer Liste aller Dateien im aktuellen Verzeichnis.
ls -l	Langform-Ausgabe der Dateiliste (mit zusätzlichen Angaben, z.B. Zugriff, Größe, Datum des letzten Zugriffs)
ls -l /home/bjm	Langform-Ausgabe der Dateiliste im Verzeichnis /home/bjm

Weitere Kommandos sind in den Folien zum Praktikum aufgeführt (Folie 8.9).
Hier wird nur sehr unvollständig auf die möglichen Optionen und Parameter eingegangen.
Nähere Auskunft gibt das Online-Manual.

Online-Manual

Das Online-Manual (man pages) enthält zu jedem Kommando oder Systemaufruf einen umfassenden Hilfetext. Das Manual ist in Sektionen eingeteilt. Sektion 1 enthält Beschreibung der Kommandos, Sektion 2 die Beschreibung der Systemaufrufe. Die Online-Hilfe kann über das Kommando man aufgerufen werden. Als Argument wird der Kommandos bzw. Des Systemaufrufs übergeben.

Example

```
man kill
```

man durchsucht standardmäßig zuerst die Sektion 1 und nachfolgend die Sektion 2. Möchte man gezielt eine bestimmte Sektion durchsuchen, muss die Sektionsnummer als 1. Argument angegeben werden. Dies ist insbesondere dann notwendig, wenn die Online-Hilfe zu einem Systemaufruf benötigt wird, zu dem ein gleichnamiges Kommando existiert.

Example

```
man 2 kill
```

Wildcards und reguläre Ausdrücke

Mit *Wildcards* als Platzhalter können Ausdrücke erstellt werden, die für eine Vielzahl von Dateien gelten.
In der Linux-Shell können Sie sogar auf das erweiterte Konzept der *regulären Ausdrücke* zurückzugreifen. Ein kurzer Auszug aus der Liste der Möglichkeiten bietet folgende Tabelle.

Ausdruck	Bedeutung
`*`	Eine beliebige Zeichenfolge.
`?`	Ein beliebiges Zeichen.
`[a-z]`	Genau ein Zeichen von a bis z.
`[!Bb]`	Ein Zeichen außer B und b.
`{info, hinweis, hilfe}.txt`	Eine der drei Dateien info.txt, hinweis.txt oder hilfe.txt.
`b*|info*`	Alle Dateien, die mit b oder info beginnen.

Wildcards und reguläre Ausdrücke: Beispiele

Example

Beispiele

`ls *.c`	Ausgabe aller Dateinamen im Arbeitsverzeichnis, deren Namen mit `.c` enden.
`ls programm.?`	Ausgabe aller Dateinamen, die mit der Zeichenkette `programm.` beginnen, und danach genau ein weiteres, beliebiges Zeichen haben.
`cp *.txt test`	Kopieren aller Dateien mit der Endung txt in den Ordner test (Voraussetzung: Der Order `test` ist vorhanden.).
`mv *test* test`	Verschiebt alle Dateien, in deren Dateinamen das Wort test vorkommt in ein Verzeichnis `test`.

Wichtige Arbeitserleichterung

History

Die bash speichert eine History ihrer eingegebenen Kommandos. Mit Hilfe der Cursor-Up und Cursor-Down-Tasten kann in der History-List zurück bzw. vorwärts geblättert werden, um ein Kommando erneut auszuführen. Mittels CTRL-R können Sie nach einem zuvor eingegebenen Kommando suchen.

Dateinamenergänzung

Für die Eingabe von Dateinamen verfügt die bash über eine automatische Namensergänzung. Sobald die eingegebenen Zeichen den Namensbestandteil in eindeutiger Weise identifizieren, wird der Name durch Betätigung der TAB-Taste automatisch vervollständigt.

Verzeichnis-Operationen

- `cd [directory]` Wechsel ins HOME-Verzeichnis bzw. nach directory
- `ls [-l] [directory]` Ausgabe des directory-Inhalts (bei fehlender directory-Angabe: aktuelles Verzeichnis) -l Ausgabe der Langform
- `mkdir directory` Erzeugen von directory
- `rmdir directory` Löschen von directory (muss leer sein)
- `pwd` Ausgabe des aktuellen Verzeichnis

```
bjm@Yoga:~$ ls -l
insgesamt 12
-rw-rw-r-- 1 bjm bjm 171 Sep  5 17:55 hello.cpp
-rw-rw-r-- 1 bjm bjm  78 Sep  5 17:55 hello_io.cpp
-rw-rw-r-- 1 bjm bjm 167 Sep  5 19:49 Makefile
bjm@Yoga:~$ mkdir subdir
bjm@Yoga:~$ ls -l
insgesamt 16
-rw-rw-r-- 1 bjm bjm  171 Sep  5 17:55 hello.cpp
-rw-rw-r-- 1 bjm bjm   78 Sep  5 17:55 hello_io.cpp
-rw-rw-r-- 1 bjm bjm  167 Sep  5 19:49 Makefile
drwxrwxr-x 2 bjm bjm 4096 Feb  8 14:10 subdir
bjm@Yoga:~$ pwd
/home/bjm
bjm@Yoga:~$ cd subdir/
bjm@Yoga:~/subdir$ ls
bjm@Yoga:~/subdir$ echo "hello" > hello.txt
bjm@Yoga:~/subdir$ ls
hello.txt
bjm@Yoga:~/subdir$ cd ..
bjm@Yoga:~$ rmdir subdir/
rmdir: konnte 'subdir/' nicht entfernen:
Das Verzeichnis ist nicht leer
bjm@Yoga:~$ rm subdir/hello.txt
bjm@Yoga:~$ rmdir subdir
bjm@Yoga:~$
```

Datei-Operationen

Kommando	Bedeutung
cat datei	Bildschirmausgabe von datei
less datei	Bildschirmausgabe von datei mit Pager-Funktion
chmod mod datei	Zugriffsrechte an datei gemäß mod ändern
chown owner[:group] datei	Eigentümer und Gruppenzugehörigkeit von datei ändern
cp datei1 datei2	Kopieren von datei1 nach datei2
ln [-s] ziel link-name	Einrichten eines (symbolischen) Link auf ziel namens link-name
mv datei1 datei2	Umbenennen von datei1 nach datei2
rm datei	Löschen von datei
touch datei	Leere datei anlegen bzw. den Zeitstempel einer bestehenden Datei aktualisieren

Prozess-Management:

Kommando	Bedeutung
kill pidProzess	mit Nr pid beenden
nice -no pid	Priorität von Prozess mit Nr. pid um no erniedrigen
ps	Information zu Prozessen anzeigen

Weitere Kommandos

Kommando	Bedeutung
date	gibt Datum und Uhrzeit aus
echo text	gibt text auf der Konsole aus
exit	Logout aus der aktuellen Shell-Sitzung
find path file	Sucht file ausgehend von path (viele darüber hinaus gehende Suchfuntionen)
mount device directory	Hängt device an der Stelle directory in den Verzeichnisbaum ein
passwd	Passwort ändern
who	Zeigt die aktuell angemeldeten Benutzer an

Unix Standard E/A Kanäle

Normalerweise mit dem Terminal verbunden in dem sie Shell läuft.

- Standard-Eingabe (Tastatur),
- Standard-Ausgabe (Teminalfenster),
- Standard-Fehlerausgabe: Separater Kanal für Fehlermeldungen (Terminalfenster)

Die Shell bietet eine einfache Syntax um die Standard E/A Kanäle umzuleiten (z.B. in Dateien, Pipes).

- Umeitung der Standard-Ausgabe mit >
- Umleitung der Stadnard-Eingabe mit <
- Pipe | verbindet die Standardausgabe der linken Seite mit der Standardeingabe der rechten Seite

```
bjm@Yoga-14:~$ ls -l > l1
bjm@Yoga-14:~$ grep "2017" < l1 > l2
bjm@Yoga-14:~$ wc < l2
  7  63 393
```

Mit Pipes geht es kompakter:

```
bjm@Yoga-14:~$ ls -l | grep "2017" | wc
  7  63 393
```

Praktikum

Praktikum 1: Linux Shell

Praktikum 2+3: Software-Entwicklung unter Linux

Praktikum 4: Parallele Programme mit Threads

Praktikum 5: Semaphor-Bibliothek

Praktikum 6: MMAP

Praktikum 7: Message Queues

Der Editor `vi`

Unter den UNIX-Editoren hat sich der `vi` zum bekanntesten und meistverwendeten Editor entwickelt.
Der vi unterscheidet zwischen den beiden Arbeitsmodi:

- Im *Kommandomodus* können Kommandos eingegeben werden (z.B. Cursor Bewegen, Text Löschen, Sichern der Textdatei, Wechsel in den Eingabemodus.
- Im *Eingabemodus* werden alle Tastatureingaben an der aktuellen Cursorposition (im Einfügemodus) in die Textdatei übernommen.

Der Aufruf des Editors erfolgt durch Eingabe des Kommandos `vi filename`

- Der Editor befindet sich zunächst im Kommando-Modus.
- Zur Texteingabe muss in den Eingabemodus gewechselt werden. Dies ist mittels folgender Kommandos möglich:
  - **i** Einfügen von Text vor der Cursorposition
  - **a** Einfügen von Text nach der Cursorposition
- Durch Betätigung der *ESC-Taste* kommt man zurück in den Kommandomodus.

Kommandos im vi

Folgende Dateioperationen sind nützlich:
`:w [filename]` Sichern der Arbeitsdatei unter dem aktuellen Dateinamen bzw. auf der Datei filename
`:r filename` Einfügen des Datei-Inhalts von filename an der Cursorposition
Das Verlassen des vi kann mit oder ohne Sicherung der Arbeitsdatei erfolgen:
`:x` Verlassen von vi mit Sicherung der Arbeitsdatei
`:q!` Verlassen von vi ohne Sicherung der Arbeitsdatei
Zum Löschen von Text muss sich der Editor im Kommandomodus befinden. Text kann mittels folgenden Kommandos gelöscht werden:
x Zeichen an Cursorposition löschen
dw Teil eines Wortes ab Cursorposition löschen
dd Zeile an Cursorposition löschen
u Letztes Kommando rückgängig machen

Weitere Kommandos im vi

Kopieren und Bewegen von Textteilen kann mittels folgender Kommandos erfolgen:
`:n1[,n2]con3` Kopieren der Zeile(n) n1 (bis n2) hinter Zeile n3
`:n1[,n2]mn3` Bewegen der Zeile(n) n1 (bis n2) hinter Zeile n3

Example

```
:3,4co5
:3,4m5
```

Hierbei ist es nützlich die Zeilen-Numerierung auf dem Bildschirm einzuschalten. Ein- und Abschalten der Zeilen-Numerierung geschieht mittels der Kommandos:
`:set number` Anzeigen der Zeilen-Nummern
`:set nonumber` Entfernen der Zeilen-Nummern

Die GNU Compiler Collection gcc

Der C++-Compiler der GNU Compiler Collection wird mittels g++ aufgerufen, der C-Compiler mittels gcc.
Der Compiler erwartet als einziges Argument den Dateinamen des zu übersetzenden Quellprogramms.

`g++ source [, source ]...`

`source` muss mit dem Suffix `.cpp` oder `.c` enden, also beispielsweise

Example

```
gcc myprog.c
g++ myprog.cpp
```

Bei fehlerfreier Übersetzung bewirkt gcc - wenn nicht explizit ausgeschlossen - automatisch einen Aufruf des Linkers ld. Bei fehlerfreiem Bindelauf wird eine ausführbare Programmdatei namens `a.out` erzeugt.

Optionen des Compilers

-c	erzeugt nur bindefähige Objektmodule (Suffix `.o`), unterdrückt den Aufruf des Linkers.
-o filename	ausführbare Datei wird unter dem Namen filename abgelegt.
-g	fügt Debuginformation für den GDB Debugger ein.

Example

```
g++ myprog.cpp -o myprog
g++ myprog.cpp -g -o myprog
```

Der Debugger gdb

gdb

Der gdb ist ein Kommandozeilen-Debugger. Das bedeutet, dass Sie eine Eingabeaufforderung erhalten, an welcher Stelle Sie einen Befehl eingeben sollten. Der gdb wird typischerweise mit dem Ziel-Programm als Argument gestartet.

- Ein Debugger ermöglicht es Ihnen, durch das Programm zu laufen und den aktuellen Zustand des Prozesses zu jedem Zeitpunkt zu überprüfen.
- Debugger sind unverzichtbare Werkzeuge für jeden ernsthaften Programmierer. Sie sollten sich daher mit einen Debugger auskennen.
- Ein wichtiger Debugger auf Linux-Systemen ist der gdb den Sie in dieser Veranstaltung nutzen sollen.

Vorbereiten des Debuggings

Compilieren mit Debugsymbolen

Damit der Debugger den Programmablauf dem Quellcode zuordnen kann, müssen bereits beim Compilieren die Debugsymbole eingefügt werden. Das geschieht mit der gcc Option -g, z.B.

```
gcc -g program.cpp -o program
```

Anschließend starten sie den gdb für Ihr Programm:

Beispielaufruf

```
gdb ./program
```

Die folgenden Abschnitte geben Ihnen einen ersten Überblick über den gdb. Eine vollstandige Dokumentation finden Sie hier: *https://sourceware.org/gdb/current/onlinedocs/gdb/*.

gdb-Kommandozeile

Sobald der Debugger gestartet wurde landen sie in der Kommandozeile des gdb.

- Sie können einige Vorbereitungen treffen, wie z.B. das Setzen von Breakpoints mit `break`.
- Wenn Sie bereit sind, die Ausführung Ihres Programms zu starten, geben Sie den Befehl **`run`** ein. Dabei können auch die Kommandozeilenparameter für Ihr Programm angegeben werden.
- Sie können Ihr Programm jederzeit mittels **`Ctrl+C`** unterbrechen und landen wieder in der gdb Eingabeaufforderung.
- Durch das **`print`**-Kommando können Sie nun den aktuellen Zustand Ihres Programms ansehen.
- Um die Programmausführung fortzusetzen, geben Sie den Befehl **`continue`** ein.

Anstatt das Programm bis zum nächsten Breakpoint oder zur manuellen Unterbrechung fortzusetzen, können Sie auch Schrittweise durch ein Programm laufen.

- Das Kommando **`next`** führt die Anweisungenin der nachfolgenden Quellcode-Zeile aus. Wenn die Zeile einen Funktionsaufruf enthält, wird dieser gesamte Funktionsaufruf ausgeführt.
- Falls Sie das Debugging innerhalb der Funktion zeilenweise fortsetzen wollen, verwenden Sie den Befehl **`step`** statt `next`.

Breakpoints

Breakpoints

Arbeit mit Breakpoints ist ein wesentlicher Bestandteil des Debuggens. Mit Breakpoints können Sie im Programm festlegen, an welcher Stelle die Ausführung unterbrochen werden muss. Wenn die Ausführung unterbrochen wird, können Sie den Programmzustand mit der Eingabeaufforderung des gdb überprüfen.

Breakpoints werden mit `b` gesetzt. Die Rückgabe ist die Nummer des Breakpoints.

- Als Argument kann der Name einer von Ihrem Programm definierten Funktion angegeben werden, oder
- ein Speicherort für den Quellcode, z.B. `file.c:123` für Zeile 123 in `file.c`.

Breakpoints können bedingt sein: In diesem Fall wird die Ausführung nur unterbrochen, wenn die Bedingung zutrifft. Das ist insbesondere bei Schleifen hilfreich.

- Eine Bedingung kann mit dem Befehl `condition` gesetzt werden.
- Z.B.:`cond 3 a > 103` setzt die Bedingung `a > 103` für Breakpoint 3.

Programmzustand betrachten

Mit GDB können Sie den aktuellen Zustand eines Programms in der Ausführung überprüfen.

- Mit dem Befehl `print` können Sie Werte von Variablen überprüfen und Ausdrücke auswerten. Man kann angeben wie der Wert ausgegeben werden soll, z.B.
  - `/x` für hexadezimale Ausgabe
  - `/t` für Binärausgabe.
- Sie können auch direkt den zum Prozess gehörenden Speicher überprüfen. Dies geschieht mit dem Befehl `x`. Auch hier können Formatierungsangaben verwendet werden und angegeben werden, wie viele Elemente gedruckt werden sollen.

Befehlsübersicht

print, p	zeigt das Ergebnis des angegebenen Ausdrucks (z.B. den Namen einer Variablen).
x	zeigt Inhalte an der angegebenen Speicheradresse.
run, r	führt das geladene Programm aus. Optional können Argumente für das auszuführende Programm angegeben werden.
continue, c	Setzt die Programmausführung fort.
next, n	Führt eine Zeile des Quellcodes aus. Funktionsaufrufe werden komplett ausgeführt.
step, s	Führt eine Zeile des Quellcodes aus. Bei Funktionsaufrufen wird in die Funktion gewechselt und ein Schritt ausgeführt.
finish	Führe das Programm bis zum Ende der aktuellen Funktion aus.
until	Führe das Programm bis zum Ende der aktuellen Schleife aus.
list, l	list source code.
backtrace, b	zeigt eine Spur aller Stack-Traces (eine Liste aller aktuellen Funktionsaufrufe).
frame, f	zeigt den aktuell ausgewählten Stack-Frame an, oder wählt einen anderen Stack-Frame nach Nummer.
break, b	einen Breakpoint setzen.
enable; disable	einen Breakpoint nach Nummer aktivieren oder deaktivieren.
delete, del	einen Breakpoint nach Nummer löschen.
info	Halteliste aktuell definierte Breakpointe.
help(h)	help

Speicherdebugging

Bugs, die durch Beschädigung des Programmspeichers ausgelöst werden, sind auch mit dem GDB schwer aufzuspüren. Um Speicherfehler zu finden sollten Sie daher das bereits aus OOP bekannte Tool `valgrind` verwenden.

Aufgabe Praktikum 3: System-Aufrufe

Schreiben Sie ein C++-Programm namens `zeiten` mit dessen Hilfe sich die Laufzeit und die CPU-Zeit eines beliebigen Programms messen lässt. Bei der CPU-Zeit soll zudem unterschieden werden zwischen CPU-Zeit im Benutzermodus und CPU-Zeit im Systemmodus. Der Aufruf des zu messenden Programms soll als Kommandozeilenargument des Mess-Programms angegeben werden.

Beispiel

Das Programm `zeiten` startet nun den `vi` und misst die dabei anfallenden Zeiten. Nach Beendigung des `vi` gibt `zeiten` die gemessenen Werte wie folgt aus:

```
bjm@Yoga:~$ ./zeiten ./cpu
Kommando: ./cpu
Laufzeit: 3.450000 sek
User-Zeit: 3.310000 sek
System-Zeit: 0.010000 sek
```

Bemerkung: Das Messprogramm wird im Laufe der weiteren Praktikumsaufgaben noch einige Male verwendet.

Praktikum

Praktikum 1: Linux Shell

Praktikum 2+3: Software-Entwicklung unter Linux

Praktikum 4: Parallele Programme mit Threads

Praktikum 5: Semaphor-Bibliothek

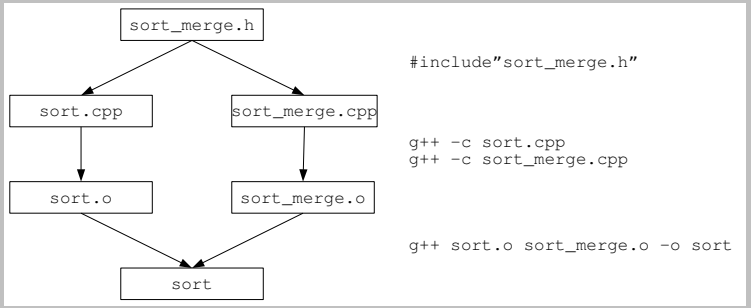
Praktikum 6: MMAP

Praktikum 7: Message Queues

Das Programm make

Bei Programmen, die aus mehreren Quelldateien bestehen und/oder weitere Bibliotheken einbinden, muss zwischen dem *Compilieren der Quelldateien* und dem *Linken zu einem ausführbaren Binary* unterschieden werden. Das Programm `make` erleichtert die Erstellung von solchen Programmen, welche aus mehreren Quelldateien und/oder Bibliotheken bestehen.

Beispiel



Makefile

Eingabe von `make` ist eine Datei (Makefile), welche die Abhängigkeiten zwischen Programmdateien und die erforderlichen Transaktionen (z.B. Übersetzungsschritte) beschreibt. Basierend auf den Zeitstempeln der beteiligten Dateien entscheidet make, welche Transaktionen durchzuführen sind.

Beispiel der vorherigen Folie

Die Abhängigkeiten werden in einer Datei namens Makefile wie folgt spezifiziert:

```
# Makefile für sort
sort : sort.o sort_merge.o
       g++ sort.o sort_merge.o -pthread -o sort

sort.o : sort.cpp sort_merge.h
       g++ -c -pthread sort.cpp

sort_merge.o : sort_merge.cpp sort_merge.h
       g++ -c sort_merge.cpp
```

Makefile Details

- Die Syntax einer zweizeiligen Abhängigkeitsspezifikation lautet generell wie folgt:
  ```
  target1 [ target2 ...] :   [ dependent1 ...]
  [<tab>  command ]
  ...
  ```
- Das Kommando muss mit einem Tabulator eingerückt werden!
- Makros
  - Makros in Makefiles sind vergleichbar mit *#define*-Direktiven in einem C-Programm.
    `bezeichner = text`
  - Der text hinter dem =-Zeichen wird mit dem Makrobezeichner gleichgesetzt. Der Aufruf des Makros erfolgt über den $-Operator, wobei der bezeichner in Klammern eingeschlossen wird.
- Kommentare werden durch das Zeichen # eingeleitet und gelten bis zum Zeilenende.

Beispiel Makefile

```
CC = gcc                    # Makrodefinition
#...
        $(CC) -c proc.c             # Makroaufruf
```

Aufruf des Makefiles

Syntax: make [-f makefile] [targets]

- Über die Option -f kann der Name des Makefile angegeben werde. Diese Angabe ist nicht nötig, wenn das Makefile unter dem Dateinamen `Makefile` abgelegt ist.
- targets enthält eine Liste von Zielen, welche aktualisiert werden sollen. Bei fehlender Angabe von targets wird das erstgenannte Ziel im Makefile aktualisiert.

Beispiele

```
make
make all
make prog.o
```

Praktikum

Praktikum 1: Linux Shell

Praktikum 2+3: Software-Entwicklung unter Linux

Praktikum 4: Parallele Programme mit Threads

Praktikum 5: Semaphor-Bibliothek

Praktikum 6: MMAP

Praktikum 7: Message Queues

Praktikum: Semaphor-Bibliothek

In diesem Praktikum soll in C++ eine statische Bibliothek für mehrwerteige Semaphoren auf Basis der Pthreads realisiert werden.

Lernziele sind:

- Threads aus der pthread-Bibliothek
- Verwenden von Mutexen und Bedingungsvariablen.
- Erstellung und Nutzung von statischen Bibliotheken.

Die Pthread-Bibliothek kennt selbst keine mehrwertuge Semaphoren. Mit Hilfe der binären Mutexte und den Bedingungsvariablen soll eine C++-Klasse realisert werden, die eine mehrwertugigen Semaphor implemntiet.
Die Funktionen der Klasse sollen in Form einer statischen Bibliothek genutzt werden können.

Threads

Syntax

```
int pthread_create(pthread_t *tid, const pthread_attr_t *attr,
                   void*(func)(void*), void* arg);
```

`pthread_create()` erzeugt einen neuen Thread. Die Thread-Id vom Typ `pthread_t` wird im Parameter *tid* zurück geliefert. Der Thread beginnt seine Ausführung in Startfunktion *func*. Für den Parameter *attr* kann im Normalfall NULL angegeben werden. Mittels *arg* können Argumente an die Startfunktion übergeben werden. Der Thread terminiert mit dem Ende der Startfunktion oder durch Aufruf der Funktion `pthread_exit()`.

Syntax

```
int pthread_join(pthread_t tid, void **staus);
```

Mittels `pthread_join()` kann ein Thread auf das Ende eines von ihm erzeugten Kind-Threads mit dem Id *tid* warten. Über den Paramter *status* kann ein Rückgabewert vom terminierten Thread empfangen werden.

Mutex

Ein Mutex ist eine Variable vom Typ `pthread_mutex_t`.

Syntax

```
int pthread_mutex_init(pthread_mutex_t *mutex,pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- `pthread_mutex_init()` initialisiert den Semaphor *mutex*. Bei einer Standardinitialisierung kann als für *attr* NULL angegeben werden.
- `pthread_mutex_destroy()` deaktiviert den Semaphor *mutex*. Dabei werden alle noch auf den Semaphor wartenden Threads bereit gesetzt.

Syntax

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- `pthread_mutex_lock()` führt eine P-Operation auf dem Semaphor *mutex* durch.
- `pthread_mutex_unlock()` führt eine V-Operation auf dem Semaphor *mutex* durch.

Bedingungsvariable

Eine Bedingungsvariable ist eine Variable vom Typ `pthread\_cond\_t`.

Syntax

```
int pthread_cond_init(pthread_cond_t *cond,  pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
```

- `pthread_con_init()` initialisiert die Bedingungsvariable *cond*. Bei einer Standardinitialisierung kann als für *attr* NULL angegeben werden.
- `pthread_cond_destroy()` deaktiviert die Bedingungsvariable *cond*.

Bedingungsvariable (Forts.)

Syntax

```
int pthread_cond_wait(pthread_cond_t *cond,  pthread_mutex_t *mutex);
int pthread_cond_signal (pthread_cond_t *cond);
```

- `pthread_cond_wait()}` suspendiert den aufrufenden Thread bis ein weiterer Thread die Bedingungsvariable signalisiert (siehe `pthread_cond_signal()`).
- Eine Bedingungsvariable ist immer mit einem Semaphor verknüpft. Ein Aufruf von `pthread_cond_wait()` ist nur im Besitz des Semaphors möglich.
- `pthread_cond_wait()` entsperrt zunächst den Mutex. Hierdurch werden andere Threads während der Wartezeit nicht unnötig behindert. Vor einer Rückkehr aus der Funktion (nach einer Signalisierung) belegt `pthread_cond_wait()` den Semaphor erneut.
- `pthread_cond_signal()` signalisiert die Bedingungsvariable *cond*. Falls vorhanden, wird ein auf *cond* wartender Thread geweckt.

Statische Bibliotheken: Archive

Die Erstellung und Verwaltung von Archiven erfolgt mit dem Programm `ar`. Im folgenden werden die wichtigsten Funktionen von ar beschrieben. Für eine weitergehende Beschreibung wird auf das Online-Manual verwiesen.

Syntax

Syntax: `ar -options archive [ files ]`

- *options* können in jeder beliebigen Reihenfolge kombiniert werden. Sie enthalten den Code für die aufzuführende(n) Operation(en). Folgende Angaben sind möglich:
  - c Das Archiv archive wird erzeugt.
  - t Eine Liste der in archive enthaltenen Dateien wird ausgegeben.
  - r Die in *files* angegebenen Dateien werden in archive eingefügt/ersetzt.
  - x Die in *files* angegebenen Dateien werden aus archive extrahiert.
  - d Die in *files* angegebenen Dateien werden aus archive entfernt.
  - v Protokollausgabe über jede eingefügte/extrahierte/gelöschte Datei.
- *archive* gibt den Namen eines Archives an. Üblicherweise haben Archive einen Namen welcher mit `lib` beginnt und mit `.a` endet (Beispiel: `libsem.a`).

Beispiele zum `ar` Kommando

Example

`ar -rv libfuncs.a func.o`

Die Datei `func.o` wird unter Protokollierung in das Archiv `libfuncs.a` eingetragen.

`ar -t libfuncs.a`

Ein Inhaltsverzeichnis von libfuncs.a wird ausgegeben.

Verwenden Statischer Bibliotheken

Werden beim Binden eines Programms neben der Standardbibliothek weitere Bibliotheken benötigt, muss dies im g++-Kommando mittels der Option `-l` angegeben werden. Die Option `-l`**name** durchsucht die Standardverzeichnisse nach einer Bibliothek mit Namen `lib`**name**`.a`.

Example

```
g++ prog.c -lm -o prog
```

Der Linker verwendet die Bibliothek `libm.a` (enthält die mathematischen Funktionen).

Falls sich die einzubindende Bibliothek nicht in einem Standardverzeichnis befinden, muss das Verzeichnis über die Option `-L` angegeben werden:

Example

```
gcc prog.c -lsem -L /home/bjm/lib -o prog
```

Die Bibliothek `libsem.a` wird eingebunden. Neben den Standardverzeichnissen sucht der Linker auch im Verzeichnis `/home/bjm/lib` nach Bibliotheken.

Praktikum

Praktikum 1: Linux Shell

Praktikum 2+3: Software-Entwicklung unter Linux

Praktikum 4: Parallele Programme mit Threads

Praktikum 5: Semaphor-Bibliothek

Praktikum 6: MMAP

Praktikum 7: Message Queues

Aufgabe

Sie sollen eine Datei im Filesystem nur durch die Verwendung von Speicheroperationen kopieren. Dazu wird für jedes Byte ein Zeiger dereferenziert und die Daten in den reservierten Ziel-Speicherbereich kopiert.

```
*ziel = *start;
```

Der Linux `mmap` Systemaufruf bietet die Möglichkeit ein Byte-für-Byte-Mapping einer Datei im Dateisystem in den logischen Adressraum eines Prozesses anzulegen. Zugriff auf den Dateiinhalt kann dann durch Zeigeroperationen im logischen Adressraum. Das Betriebssystem kümmert sich darum den Inhalt der Datei im Dateisystem zu aktualisieren.

Systemaufrufe zum Dateizugriff: open

Öffnen einer Datei.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

- `open` öffnet die durch `pathname` angegebene Datei.
- `flags` Bitmaske für weitere Argumente, z.B.
  - Eine der folgenden Zugriffsmodi muss gewählt werden: `O_RDONLY`, `O_WRONLY`, `O_RDWR`. Die Datei wird read- only, write-only, oder mit read/write Rechten geöffnet.
  - `O_CREAT` erzeugt eine neue Datei, falls `pathname` nicht existiert. Zusätzlich `O_EXCL` erzwingt dass die Datei erzeugt wird und liefert einen Fehler, falls die Datei bereits existiert.
  - Das `mode` spezifiziert die Bits für die Zugriffsrechte der Datei, falls `O_CREAT` gesetzt wurde.
- Der Rückgabewert ist ein File-Deskriptor, der in Nachfolgenden Systemaufrufen (z.B. read, write, mmap) verwendet wird.

Systemaufrufe zum Dateizugriff: fstat

Der `fstat` Aufruf liefert Informationen über eine Datei.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int fstat(int fd, struct stat *statbuf);
```

- `fd` ist ein File-Deskriptor.
- `statbuf` ist ein Zeiger auf einen `struct stat`, in dem die Informationen über die Datei abgelegt werden.

Darin enthalten ist beispielsweise die Größe der Datei.

```
struct stat {
   off_t     st_size;        /* Total size, in bytes */
   // ...
}
```

Systemaufrufe zum Dateizugriff: ftruncate

Die Funktion `ftruncate` setzt die Größe der durch den Deskriptor `fd` gegebenen Datei auf genau `length` Byte. Falls die Datei zuvor größer war, sind die Daten verloren, falls die Datei kürzer war, wird sie durch Nullbytes (`'\0'`) verlängert.

```
#include <unistd.h>
#include <sys/types.h>
int ftruncate(int fd, off_t length);
```

Im Erfolgsfall wird 0 zurückgegeben, im Fehlerfall -1 und `errno` beschreibt den Fehler.

Der mmap Systemaufruf (II)

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

- addr sollte NULL sein, dann wählt der Kernel eine geeignete Startadresse für das Mapping, sonst versucht der Kernel das Mapping an der angegebenen Adresse einzurichten.
- length ist Größe des Mappings in byte.
- prot ist die Bitmaske für Speicherschutz, entweder PROT_NONE (kein Zugriff erlaubt), oder ein bitweises oder eines oder mehrere der Flags PROT_READ, PROT_WRITE, PROT_EXEC.
- flags , Bitmaske für weitere Argumente z.B. MAP_SHARED andere Prozesse sehen die vorgenommenen Änderungen in dem Speicherbereich, MAP_ANONYMOUS anonymes Mapping, d.h. ein Speicherbereich der nicht im Filesystem liegt.
- fd Der zu mappende File Descriptor oder -1 bei MAP_ANONYMOUS.
- offset Offset im File (Vielfaches der Page-Größe) oder 0

Fehlerbehandlung in Linux

Fast alle Systemaufrufe im Fehlerfall den Wert -1 zurück. Die nähere Fehlerursache wird in einer globalen Variablen namens `errno` abgelegt.
Durch Inklusion der Header-Datei `<errno.h>` wird der Zugriff auf die globale Variable errno ermöglicht (<errno.h> enthält eine extern-Deklaration der Variablen errno).
Die Funktion perror() aus der C-Standard-Bibliothek dient zur Fehlerausgabe.

```
# include <stdio.h>
void perror(const char* msg);
```

perror

```
# include <stdio.h>
void perror(const char* msg);
```

perror() gibt den Text msg gefolgt von einer mit dem aktuellen Wert von errno korrespondierenden Fehlermeldung aus.

Beispiel

```
errno = EACCES;
perror("Fehler");          /* Ausgabe: Fehler: Permission denied */
```

Praktikum

Praktikum 1: Linux Shell

Praktikum 2+3: Software-Entwicklung unter Linux

Praktikum 4: Parallele Programme mit Threads

Praktikum 5: Semaphor-Bibliothek

Praktikum 6: MMAP

Praktikum 7: Message Queues

Aufgabe

Zu realisieren ist eine einfache Client-Server-Anwendung mittels Message Queues. Ein Client-Prozess wickelt zyklisch den nachfolgenden Dialog mit dem Benutzer ab.

Example

Ausgabe: Auftrag eingeben:

```
Eingabe: + 10 20
Ausgabe: 30
...
```

Der Client leitet den Auftrag zur Durchführung der eigentlichen Berechnung an den Server weiter. Nach Erhalt des Ergebnisses wird dieses vom Client auf dem Bildschirm ausgegeben. Die eigentliche Berechnung führt ein Server-Prozess durch. Dieser empfängt den Rechenauftrag vom Client, interpretiert ihn und sendet das Ergebnis an den Client zurück. Ihre Aufgabe ist es den entsprechenden Server zu implementieren.

Client-Server-Modell

Viele Anwendungen sind heute nach dem so genannten Client-Server-Modell realisiert. Ein Server-Prozess wartet auf die Anforderung eines bestimmten Dienstes durch einen oder mehrere Client-Prozesse. Ein typischer Ablauf sieht wie folgt aus.

1. Auf einem Rechner wird ein Server-Prozess gestartet. Dieser öffnet einen Kommunikationskanal und geht anschließend in einen Wartezustand, um auf Client-Anfragen zu warten.
2. Der oder die Client-Prozesse werden entweder auf dem Server-Rechner selbst oder auf entfernten Rechnern, welche mit dem Server-Rechner über ein Netz verbunden sind, gestartet. Anschließend sendet der Client-Prozess seine Anforderung an den Server-Prozess. Typische Anforderungen sind:
   - Lesen/Schreiben von/auf Dateien, die sich auf dem System des Servers befinden (Remote File Service)
   - Laden von Webseiten von einem http-Server
3. Nachdem der Server-Prozess seine Aufgabe für den Client-Prozess durchgeführt hat, geht er zurück in den Wartezustand um auf die nächste Client-Anfrage zu warten.

Systemaufrufe

Bei der folgenden Beschreibung der Systemaufrufe zur Handhabung von Message Queues handelt es sich um einen Auszug aus der Online-Dokumentation.

```
#include <fcntl.h>           /* For O_* constants */
#include <mqueue.h>
mqd_t mq_open(const char *name, int oflag);
mqd_t mq_open(const char *name, int oflag, mode_t mode,
              struct mq_attr *attr);
```

`mq_open()` erstellt oder öffnet die mit *name* assoziierte POSIX Message Queue und gibt einen Identifier zurück. Der Bitvektor *oflag* steuert die Ausführung von `mq_open()`. *oflag* muss ein Zugriffsrecht aus `O_RDONLY`, `O_WRONLY` oder `O_RDWR` enthalten. Dazu sind die folgenden Flags möglich:

- `O_CREAT` erstellt die Message Queue falls sie noch nicht existiert (nächste Folie), zusätzlich `O_EXCL` liefert einen Fehler, falls die Queue bereits existiert.
- `O_NONBLOCK` öffnet die Message Queue im non-blocking Modus.

Message-Queue erzeugen

Wenn `O_CREAT` spezifiziert wird müssen zwei weitere Argumente angegeben werden.

```
mqd_t mq_open(const char *name, int oflag, mode_t mode,
        struct mq_attr *attr);
```

mode definiert die Zugriffsrechte zu der erzeugten Message Queue im bekannten Format, *attr* ist ein Zeiger auf ein `struct mq_attr`, das Kapazität und Nachrichtengröße der Message Queue festlegt.

```
struct mq_attr {
   long mq_flags;       /* Flags (ignored for mq_open()) */
   long mq_maxmsg;      /* Max. # of messages on queue */
   long mq_msgsize;     /* Max. message size (bytes) */
   long mq_curmsgs;     /* # of messages currently in queue
                           (ignored for mq_open()) */
};
```

Beispiel

Example

```
// Set attributes of server queue
attr.mq_flags = 0;
attr.mq_maxmsg = 10;
attr.mq_msgsize = MSG_SIZE;
attr.mq_curmsgs = 0;

// Create the server queue
mq_server = mq_open(sqname_server, O_CREAT | O_RDONLY, 0644, &attr);
```

Senden einer Nachricht

msg_send fügt die Nachricht, auf die `msg_ptr` zeigt, der Message Queue `mqdes` hinzu. Das Argument `msg_len` spezifiziert die Länge der Nachricht, der wert muss kleiner oder gleich der `mq_msgsize` sein. Dert Wert `msg_prio` ist eine nicht-negativer Integer, der die Priorität beschreibt. Nachrichten werden in Reihenfolge absteigender Priorität ausgeliefert.
Falls die Message Queue voll ist, blockiert `mq_send()` typischerweise. Falls das Flag `O_NONBLOCK` für die Message Queue gesetzt ist, liefert der Aufrag stattdessen den Fehler `EAGAIN`.
Bei Erfolg ist der Rückgabewert von `mq_send()` 0, bei einem Fehler -1.

```
#include <mqueue.h>

int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
            unsigned int msg_prio);


// Link with -lrt.
```

Empfangen einer Nachricht

msg_receive entfernt die älteste der Nachrichten mit höchster Priorität aus der Message Queue und speichert sie in dem Puffer auf den `msg_ptr` zeigt. Die Länge des Puffers wird in `msg_len` angegeben und muss mindestens so groß sein wie das Attribut `mq_msgsize` der Message Queue. Falls `msg_prio` nicht NULL ist, wird die Priorität der ausgelieferten Nachricht in die referenzierte Variable gespeichert.
Falls die Message Queue leer ist, blockiert `mq_receive()` typischerweise. Falls das Flag `O_NONBLOCK` für die Message Queue gesetzt ist, liefert der Aufrag stattdessen den Fehler `EAGAIN`.
Der Rückgabewert gibt die Zahl der gelesenen Bytes zurück oder -1 bei Auftreten eines Fehlers.

```
#include <mqueue.h>

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
                   unsigned int *msg_prio);

// Link with -lrt.
```

Beispiel

Example

Die Zeichenkette in `message` wird in die Message Queue `mq_server` geschrieben. Anschließend wird eine Nachricht aus der Message Queue `mq_client` in den String `message` gelesen.

```
#define MSG_SIZE 100
char message[MSGSIZE];
strcpy(message, "Nachrichtentext");
if (mq_send(mq_server, message, MSG_SIZE, 0) == -1) {
   perror("CLIENT");
   return -1;
}
if (bytes_read = mq_receive(mq_client, message, MSG_SIZE, NULL) == -1) {
   perror("CLIENT");
   return -1;
}
```

Schließen der Message-Queue

```
#include <mqueue.h>
int mq_close(mqd_t mqdes);
```

mq_close() schließt die Message Queue `mqdes`. Im Fehlerfall liefert *mq_close()* den Wert -1 zurück.

Example

```
if (mq_close(mq_server) == -1) {
   perror("mq_close");
   exit(1);
}
```

Eine geöffnete Message-Queue finden Sie als Dateieintrag im Verzeichnis `/dev/mqueue/`. Dort kann die Message-Queue auch durch Dateioperationen gelöscht werden.

Signale

Folgendes Problem muss noch gelöst werden:

- Ein Serverprozess wird in der Regel als Hintergrundprozess gestartet und nach Abschluss der Anwendung mittels des Kommandos kill beendet.
- Der Abbruch darf dabei nicht unmittelbar erfolgen, da der Server vorher die Message Queue schließen sollte.

Dieses Problem lässt sich mit Hilfe von so genannten Signalen realisieren. Signale sind softwarebasierte Unterbrechungen, welche der Betriebssystemkern anlässlich verschiedener Ereignisse an den Prozess sendet.

Abfangen von Signalen im Programm

Außer SIGKILL und SIGSTOP können alle Signale ignoriert oder abgefangen und zur Behandlung an eine als Signal-Handler bezeichnete Funktion weitergeleitet werden.
Der Systemaufrufs signal() installiert einen Signal-Handler:

Syntax

```
#include <signal.h>
void (*signal(int signo, void (*func) (int)))(int);
```

- *signo* gibt den Namen des Signals an (siehe Tabelle 4.1).
- *func* kann die Adresse einer Funktion (Signal-Handler) enthalten, welche beim Empfang des Signals aufgerufen werden soll. Ein Signal-Handler erhält als Parameter ein `int`, die Signal-Nummer.
- `signal()` liefert selbst einen Funktionszeiger auf die vorher eingestellten Signal-Behandlung zurück.

Beispielaufruf

Example

```
void cleanUp(int sig);      /* Signal Handler */

signal(SIGTERM, cleanUp);
```