Betriebssysteme 2 | BSys2

Zusammenfassung

INHALTSVERZEICHNIS

1. BETRIEBSSYSTEM API

Aufgaben eines Betriebssystems:
- ***Abstraktion*** und damit ***Portabilität*** *(von Hardware, Protokollen, Software-Services)*
- ***Ressourcenmanagement*** und ***Isolation*** der Anwendungen *(Rechenzeit, RAM- & Speicherverwendung etc.)*
- ***Benutzerverwaltung*** und Sicherheit

Grenzen der Portierbarkeit

Applikation muss auf allen Bildschirmgrössen, mit allen verschiedenen Bedienarten (Maus vs. Touchscreen) etc. funktionieren. Moderne Betriebssysteme bieten dafür Mechanismen an, es obliegt aber der Applikation, diese zu verwenden. Das OS kann nicht entscheiden, was die Applikation meint.

Grenzen der Isolierbarkeit

Applikationen, die auf einem Bildschirm laufen, konkurrieren zwangsläufig um Bildschirm und Tastatur. Häufiges Problem: Fokus-Diebstahl über Popups

Prozessor Privilege Level

Moderne OS benötigen Prozessor mit mindestens zwei Privilege Levels:
- ***Kernel-Mode:*** Darf jede Instruktion ausführen *(Ring 0)*
- ***User-Mode:*** Darf nur eine beschränkte Menge an Instruktionen ausführen *(Ring 3)*

Das OS läuft im Kernel-Mode und bestimmt über Software.

1.1. GRUNDAUFBAU EINES BETRIEBSSYSTEMS

OS werden typischerweise in einen ***Kern*** und einen ***Nicht-Kernbereich*** aufgeteilt. Kern umfasst die Komponenten, die im Kernel-Mode laufen müssen, alle anderen Komponenten sollten im User-Mode laufen.

1.1.1. Microkernel

Kernelfunktionalität ***reduziert*** auf ein Minimum. Selbst Gerätetreiber laufen im User-Mode, nur ***kritische Teile*** des Kernels laufen im Kernel-Mode. ***Stabil*** und Analysierbar, jedoch ***Performance-Einbussen***.

1.1.2. Monolithische Kernel

Die meisten OS-Kernel sind monolithisch. ***Vorteil:*** weniger Wechsel zwischen den Modi → bessere Performance. ***Nachteil:*** weniger Schutz vor Programmierfehlern, da weniger Isolierung.

1.1.3. Unikernel

Ein «normales» Programm als Kernel, die Kernelfunktionalität ist in einer Library. Keine Trennung zwischen Kernel- und User-Mode. ***Vorteil:*** Echte Minimalität, extrem kompakt. ***Nachteil:*** Single Purpose, Applikationsentwickler muss sich mit Hardware auseinandersetzen.

1.1.4. Wechsel des Privilege Levels vom User zum Kernel Mode

Die `syscall` Instruktion auf Intel x86 Prozessoren veranlasst den Prozessor, in den ***Kernel Mode zu schalten*** und den Instruction Pointer auf OS-Code (System Call Handler) umzusetzen. Dadurch ist gewährleistet, dass im Kernel Mode immer Kernel-Code läuft.

1.1.5. Zusammenspiel von Applikation und Kernel Code

Da es nur einen `syscall`-Befehl gibt, muss jede OS-Kernel-Funktion mit einem ***Code*** versehen werden. Dieser Code muss in einem Register übergeben werden. Zusätzlich müssen je nach Funktion ***Parameter*** in anderen Registern übergeben werden. Z.B. System Call `exit`: Hat den Code 60 und erwartet den Exit-Code des Programms in einem Register.

1.1.6. ABI vs. API

Application Binary Interface - ABI	*Application Programming Interface - API*
– Abstrakte Schnittstellen – Plattformunabhängige Aspekte – Kann für diverse OS gleich sein	– Konkrete Schnittstellen – Calling Convention – Abbildung von Datenstrukturen

Linux-Kernels sind API-, aber nicht ABI-kompatibel. Die API-Kompatibilität ist dadurch gegeben, dass Applikationen nicht direkt Syscalls aufrufen, sondern C-Wrapper-Funktionen verwenden. Diese verwenden zum Kernel passenden Binärcode.

1.1.7. POSIX (Portable Operating System Interface)

Jedes OS hat eigene API und ABI. Der OS-spezifische Teil der UNIX/C-API ist jedoch als POSIX standardisiert. macOS und Linux sind POSIX-konform, Windows nicht.

Man Pages

Man pages *(manual pages)* enthalten Dokumentation für die Programme auf einem POSIX-System. Liefert viele Informationen über ein POSIX-System. Ist in 9 Kapitel aufgeteilt, z.B. Kapitel 3 für Libraries.

Shell

Programm, das es erlaubt, über Texteingabe Betriebssystemfunktionen aufzurufen. Gibt viele verschiedene Shells mit unterschiedlichem Syntax. Benötigt keine besonderen Rechte oder spezielle Vorkehrungen. Benötigt nur Ausgabe- und Eingabe-Stream.

1.2. PROGRAMMARGUMENTE

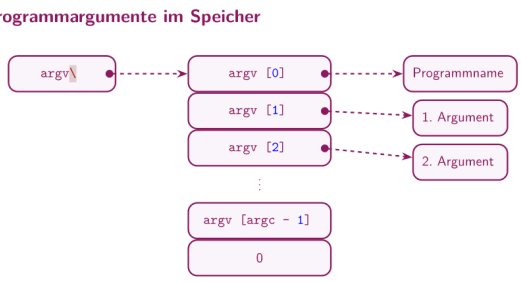
Wird ein Programm gestartet, kann es Programmargumente erhalten.
Beispiel: `clang -c abc.c -o abc.o`

1.2.1. Programmargumente aus der Shell

Shell teilt Programmargumente in ***Strings*** auf. Fast alle Shells verwenden ***Leerzeichen als Trennung*** zwischen Programmargumenten. Viele Shells erlauben Spaces in Programmargumenten durch die Verwendung von ***Quotes***. Das OS interessiert sich ***nicht*** für den ***Inhalt*** der Argumente.

1.2.2. Calling Convention

Beim Start schreibt das OS die Programmargumente als ***null-terminierte Strings*** in den Speicherbereich des Programms. Zusätzlich legt das OS ein ***Array*** `argv` an, dessen Elemente jeweils auf das ***erste Zeichen eines Programmarguments*** zeigen. Der Pointer auf dieses Array und die Anzahl der Elemente `argc` wird dem Programm an ***einer vom OS definierten Stelle*** zur Verfügung gestellt, z.B. in Registern oder auf dem Stack. Die Art und Weise wie/wo dies gehandhabt wird, ist die Calling Convention.



1.2.3. Programmargumente in C

In C wird dieser Umstand durch die beiden Parameter der Funktion `main()` ausgedrückt.
– `int argc` enthält die Anzahl der Programmargumente + 1.
– `char **argv` enthält den Pointer auf das Array.

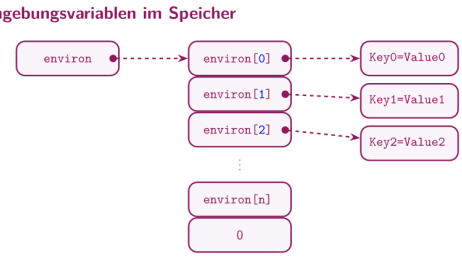
Achtung: `argv[0]` ist der Programmname, die Argumente selbst folgen als `argv[1]` bis `argv[argc - 1]`

```
int main(int argc, char ** argv) { ... }
```

1.3. UMGEBUNGSVARIABLEN

Die Umgebungsvariablen eines Programms sind eine Menge an Strings, die jeweils mindestens ein = enthalten, z.B.:

```
OPTER=1
OPTIND=1
OSTYPE=linux-gnu
PATH=/home/ost/bin:/home/ost/.local/bin
```



- Der Teilstring vor dem = wird als ***Key***, der nach dem = als ***Value*** bezeichnet
- Jeden Key kann es ***höchstens einmal*** geben

Unter POSIX verwaltet das OS die Umgebungsvariablen innerhalb jedes laufenden Prozesses. Sie werden ***initial*** vom erzeugenden Prozess festgelegt, also z.B. der Shell *(die Shell kopiert ihre Umgebungsvariablen in den Prozess)*. Das OS legt die Umgebungsvariablen als ein ***null-terminiertes Array von Pointern auf null-terminierte Strings*** ab. Unter C zeigt die Variable `extern char **environ` auf dieses Array. ***`environ`*** sollte nicht direkt verwendet werden, sondern nur über folgende Funktionen manipuliert werden: ***`getenv()`***, ***`putenv()`***, ***`setenv()`*** und ***`unsetenv()`***.

Abfragen einer Umgebungsvariable: `char * getenv (const char * key)`

Durchsucht die Umgebungsvariablen nach dem Key `key` und gibt die Adresse des ersten Zeichens des entsprechenden Values zurück falls vorhanden, ansonsten `0`.

```
char *value = getenv("PATH");
// value = "/home/ost/bin:/home/ost/.local/bin"
```

Setzen einer Umgebungsvariable: `int setenv (const char *key, const char *value, int overwrite)`

Wenn `key` schon in einer Umgebungsvariable `v` enthalten ist ***und*** `overwrite` $\neq$ `0`: überschreibt den Wert von `v` mit `value`. Wenn `key` noch nicht in einer Umgebungsvariable enthalten ist: fügt eine neue Umgebungsvariable hinzu und kopiert `key` und `value` dort hinein. Gibt `0` zurück wenn alles OK, ansonsten Fehlercode in `errno`.

```
int ret = setenv("HOME", "/usr/home", 1);
```

Entfernen einer Umgebungsvariable: `int unsetenv (const char *key)`

Entfernt die Umgebungsvariable mit dem Key `key`. Gibt `0` zurück wenn alles OK, ansonsten Fehlercode in `errno`.

```
int ret = unsetenv("HOME");
```

Hinzufügen einer Umgebungsvariable: `int putenv (char * kvp)`

fügt den Pointer `kvp` *(key-value-pair)* dem Array der Umgebungsvariablen hinzu. Der String, auf den `kvp` zeigt, wird ***nicht*** kopiert. Wird der String nach dem Setzen der Umgebungsvariabel geändert, wird diese ebenfalls geändert. Wenn der Key schon vorhanden ist, wird der String gelöscht, auf den der existierende Pointer zeigt. D.h. der Pointer verweist anschliessend auf eine leere Stelle. Gibt `0` zurück wenn alles OK, ansonsten Fehlercode in `errno`. ***Gefährliche Funktion!***

```
int ret = putenv("HOME=/usr/home");
```

1.4. ZWECK VON PROGRAMMARGUMENTEN UND UMGEBUNGSVARIABLEN

Programmargumente	***Umgebungsvariablen***
– werden explizit angegeben – nützlich für Informationen, die bei jedem Aufruf anders sind *(z.B. die Datei, die kompiliert werden soll)*	– werden implizit bereitgestellt – nützlich für Informationen, die bei jedem Aufruf gleich sind *(z.B. Pfade für Hilfsprogramme, Libraries)*

Grössere Konfigurationsinformationen sollten bevorzugt über ***Dateien*** übermittelt werden. Das ist häufig nötig wegen Beschränkungen der Zeilenlänge. Datenformat völlig in der Hand des Programms; keine Unterstützung durch das OS. Der Dateiname kann als Umgebungsvariable oder Programmargument übergeben werden.

Manche Betriebssysteme kennen noch andere Mechanismen, z.B. Windows Registry, die eher einer Datenbank gleicht.

2. DATEISYSTEM API

Dateiendungen sind die Zeichen nach dem letzten Punkt. Dateiendungen haben für File System (FS) und OS ***(fast) keine Relevanz***. Bestimmte Programme deuten Dateiendung als Typ. Häufig wird der Typ aber durch ***Magic Numbers*** oder Strings innerhalb der Datei gekennzeichnet.

2.1. DATEISYSTEM-GRUNDLAGEN

2.1.1. Schutz gegen falsche Datentypen

Es liegt an der Applikation, den Dateityp richtig zu bestimmen. Applikationen müssen sich gegen «Datenmüll» (bzw. Fehlinterpretation) schützen. Sie dürfen ***nie*** annehmen, dass Daten ***gültig*** sind, sondern müssen diese ***validieren*** und auf Grenzverletzungen überprüfen.

2.1.2. Begriffe

- ***Verzeichnis:*** Liste, die Dateien oder weitere Verzeichnisse enthalten kann. Als Datei realisiert, die diese Liste enthält. Hat einen Dateinamen.
- ***Verzeichnishierarchie:*** Gesamtheit aller Verzeichnisse im System. Jedes Verzeichnis (ausser Wurzelverzeichnis) hat genau ein Elternverzeichnis (Baum-Hierarchie).
- ***Wurzelverzeichnis:*** Oberstes Verzeichnis in der Hierarchie. Hat keinen Namen, wird aber oft mit / bezeichnet. *(Windows: Root pro Partition, Unix: Root pro OS)*

2.1.3. Besondere Verzeichnisse

Jedes Verzeichnis enthält zwei implizite Referenzen auf Verzeichnisse:

- `.` *(ein Punkt)*: Referenz auf sich selbst
- `..` *(zwei Punkte)*: Referenz auf das Elternverzeichnis

Jeder Prozess hat ein ***Arbeitsverzeichnis (working directory)***, welches den Bezugspunkt für relative Pfade darstellt. Dieses wird beim Prozessstart von aussen festgelegt. Wird mit `getcwd()` ermittelt und mit `chdir()` *(nimmt Pfad als String)* bzw. `fchdir()` *(nimmt file descriptor)* geändert.

2.1.4. Pfade

Ein Pfad spezifiziert eine Datei oder ein Verzeichnis in der Verzeichnishierarchie. Verzeichnisnamen werden durch / voneinander getrennt (Windows: \).

- ***Absoluter Pfad*** beginnt mit / *(vom Root-Verzeichnis aus)*
- ***Relativer Pfad*** beginnt ***nicht*** mit / *(vom Arbeitsverzeichnis aus)*
- ***Kanonische Pfade*** sind absolute Pfade ohne "`.`" und "`..`". Können mit `realpath()` ermittelt werden.

Längster Pfadname

Verschiedene Implementierungen von POSIX dienen unterschiedlichen Zwecken. Systeme können unterschiedliche Limits haben. Jedes POSIX-System definiert den Header `<limits.h>`:

- ***`NAME_MAX`:*** Maximale Länge eines Dateinamens (exklusive terminierender Null)
- ***`PATH_MAX`:*** Maximale Länge eines Pfads (inklusive terminierender Null) *(beinhaltet Wert von `NAME_MAX`)*
- ***`_POSIX_NAME_MAX`:*** Minimaler Wert von `NAME_MAX` nach POSIX *(14)*
- ***`_POSIX_PATH_MAX`:*** Minimaler Wert von `PATH_MAX` nach POSIX *(256)*

Beispiel - Arbeitsverzeichnis ausgeben:

```c
int main (int argc, char ** argv) {
  char *wd = malloc(PATH_MAX); // PATH_MAX = Maximale Länge des Pfades
  getcwd(wd, PATH_MAX);
  printf("Current working directory is %s", wd);
  free(wd);
  return 0;
}
```

2.1.5. Zugriffsrechte (Unix)

Jeder Datei und jedem Verzeichnis sind Zugriffsrechte zugeordnet. Gehört genau einem Benutzer (Owner) und genau einer Gruppe. Hat je 3 Permission-Bits für ***Owner***, ***Gruppe***, und ***andere Benutzer***.

- ***Read-Bits:*** Darf lesen
- ***Write-Bits:*** Darf schreiben
- ***Execute/Search-Bits:*** Darf ausführen (Datei) bzw. durchsuchen (Verzeichnis)

Es gibt eine feste Reihenfolge der 9 Permission Bits: `owner rwx - group rwx - other rwx`.
Schreibweise: `r=4, w=2, x=1`
`0740` oder `rwx r-- ---` bedeutet Owner hat alle Rechte, Gruppe kann nur lesen, andere haben keine Rechte.

POSIX

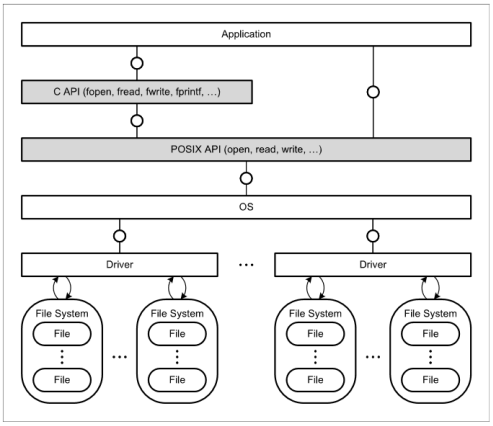
Die POSIX-API definiert die `STAT_INODE` Konstanten für die Zugriffsrechte in `<sys/stat.h>`. Beispiele:

- ***S_IRWXU*** = `0700` = `rwx------` *read, write & execute for user*
- ***S_IWUSR*** = `0200` = `-w-------` *write for user*
- ***S_IRGRP*** = `0040` = `---r-----` *read for group*
- ***S_IXOTH*** = `0001` = `--------x` *execute for other*

Können mit `|` verknüpft werden, z.B. `S_IRWXU | S_IRGRP`

2.2. ÜBERBLICK APIS

- ***POSIX-API:*** für direkten Zugriff, alle Dateien sind rohe Binärdaten *(so wie sie in der Datei gespeichert sind)*
- ***C-API:*** für direkten Zugriff auf Streams *(Textdateien)*, Abstraktion über Dateien, Pipes, etc. Für formatierte Ein- und Ausgabe, OS leitet alle Zugriffe an Treiber weiter.



2.3. POSIX FILE API

API für ***direkten, unformatierten Zugriff*** auf Inhalt der Datei. Sollte ***nur für Binärdaten*** verwendet werden. Funktionen sind deklariert in `<unistd.h>` *(Unix Standard API)* und `<fcntl.h>` *(File Control)* und geben im Fehlerfall `-1` zurück. Der Fehler-Code kann dann mit `errno` abgefragt werden.

2.3.1. `errno`

- ***Makro oder globale Variable vom Typ `int`*** *(verhält sich immer wie eine globale Variable)*
- Wird von vielen Funktionen gesetzt.
- Sollte ***unmittelbar nach Auftreten eines Fehlers*** aufgerufen werden *(damit Wert nicht von anderer Funktion überschrieben wird)*

```c
if (chdir("docs") < 0) {
  // hier nichts anderes machen, damit Fehlercode nicht überschrieben wird
  if (errno == EACCESS) {
    printf("Error: Access denied");
  }
}
```

`char * strerror (int code)`

`strerror` gibt die Adresse eines Strings zurück, der den Fehlercode `code` textuell beschreibt.

```c
if (chdir("docs") < 0) {
  printf("Error: %s\n", strerror (errno)); // e.g. Error: Permission denied
}
```

`void perror (const char *text)`

`perror` *(prefix error)* schreibt `text` gefolgt von einem Doppelpunkt und vom Ergebnis von `strerror(errno)` auf den Error-stream.

```c
if (chdir("docs") < 0) {
  perror("chdir"); // chdir: No such file or directory
}
```

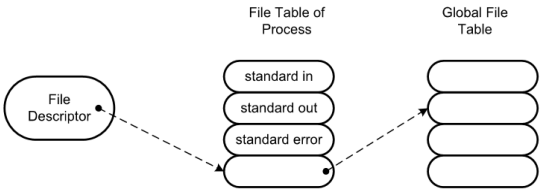
2.3.2. File-Descriptor

Files werden in der POSIX-API über ***File-Deskriptoren (FD)*** repräsentiert. Gilt immer nur innerhalb eines Prozesses. Ein neu erstellter FD returnt einen Index, über welchen auf ihn zugegriffen werden kann.

- Index in eine ***Tabelle aller geöffneten Dateien im Prozess***
- Tabelleneintrag enthält ***Index in systemweite Tabelle*** aller geöffneten Dateien
- Die systemweite Tabelle enthält Daten, um physische Datei zu identifizieren. Zustandsbehaftet: merkt sich aktuellen Offset (Offset des Bytes, das als nächstes gelesen werden wird)

In jedem Prozess sind drei Standard File-Deskriptoren definiert:

- ***STDIN_FILENO = 0:*** standard input
- ***STDOUT_FILENO = 1:*** standard output
- ***STDERR_FILENO = 2:*** standard error



2.3.3. Öffnen und Schliessen von Dateien: `int open (char *path, int flags, ...)`

Erzeugt einen File-Deskriptor auf die Datei, die an `path` liegt. `flags` gibt an, wie die Datei geöffnet werden soll.
(können über Pipe kombiniert werden. Sollen noch Berechtigungs-Flags verwendet werden, werden diese als eigener Parameter angegeben)

- ***O_RDONLY:*** nur lesen
- ***O_RDWR:*** lesen und schreiben
- ***O_CREAT:*** Erzeuge Datei, wenn sie nicht existiert; benötigt weiterer Parameter für Zugriffsrechte
- ***O_APPEND:*** Setze Offset ans Ende der Datei vor jedem Schreibzugriff *(ohne dieses Flag wird bei jedem Schreiben der Inhalt von Anfang an überschrieben)*
- ***O_TRUNC:*** Setze Länge der Datei auf 0 *(Inhalt löschen)*

`int close (int fd)` ***deallociert*** den ***File-Deskriptor*** `fd`. Dieser kann später von `open` für eine andere Datei verwendet werden *(gleiche File-Deskriptoren != gleiche Datei)*. Gibt `0` *(OK)* oder `-1` *(Fehler, z.B. FD existiert nicht)* zurück. Wird die Datei ***nicht geschlossen***, kann es sein, dass das ***FD-Limit*** des Prozesses erreicht wird und ***keine weiteren Dateien*** mehr geöffnet werden können. Es können auch ***mehrere FDs dieselbe Datei öffnen***, da diese aber ***verschiedene Offsets*** haben können, besteht die Gefahr, dass sie sich ***gegenseitig Daten überschreiben*** - nicht empfehlenswert.

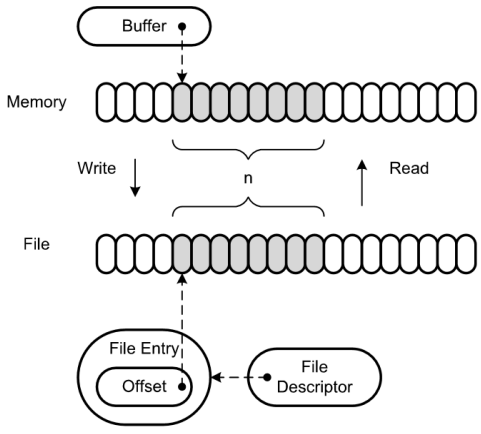
```c
int fd = open("myfile.dat", O_RDONLY);
if (fd < 0) {
  // error handling, -1 means error
}
// read data
close(fd); //gets written on the disk and the resources (and the file) can be used again
```

2.3.4. Lesen und Schreiben von Dateien: `ssize_t read(int fd, void * buffer, size_t n)`

Versucht, die nächsten n Byte am aktuellen Offset von `fd` in den `buffer` zu kopieren.

`ssize_t write(int fd, void * buffer, size_t n)` versucht, die nächsten n Byte vom `buffer` an den aktuellen Offset von `fd` zu kopieren.

Beide Funktionen geben die Anzahl der gelesenen / geschriebenen Bytes zurück oder −1 bei Fehler *(darum ist return type signed size)*. Blockieren normalerweise, bis n Bytes kopiert wurden, ein Fehler auftritt oder das Ende der Datei erreicht wurde. Erhöhen Offset von `fd` um Anzahl gelesener / geschriebener Bytes.

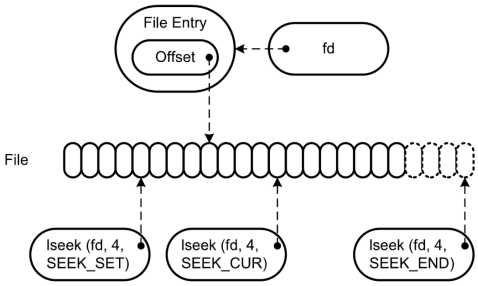


```c
#define N 32
char buf[N]
char spath[PATH_MAX]; // source path
char dpath[PATH_MAX]; // destination path
// ... gets paths from somewhere
int src = open(spath, O_RDONLY);
int dst = open(dpath, O_WRONLY | O_CREAT, S_IRWXU);
ssize_t read_bytes = read(src, buf, N);
write(dst, buf, read_bytes); // if file gets closed early, only write as many bytes
close(src);                  // as have been read from the file
close(dst);
```

2.3.5. Springen in einer Datei: `off_t lseek(int fd, off_t offset, int origin)`

Setzt den Offset von `fd` auf `offset`. `origin` gibt an, wozu `offset` relativ ist:

- ***`SEEK_SET:`*** Beginn der Datei *(absoluter Offset)*
- ***`SEEK_CUR:`*** Aktueller Offset *(relativer Offset)*
- ***`SEEK_END:`*** Ende der Datei *(Offset über Datei hinaus)*



Gibt neuen Offset zurück oder −1 bei Fehler.

Weitere Anwendungsmöglichkeiten:

- ***`lseek(fd, 0, SEEK_CUR):`*** gibt aktuellen Offset zurück
- ***`lseek(fd, 0, SEEK_END):`*** gibt die Grösse der Datei zurück
- ***`lseek(fd, n, SEEK_END):`*** hängt bei nachfolgendem `write` n Nullen an Datei *(Padding, um Datei auf bestimmte Grösse zu setzen)*.

2.3.6. Lesen und Schreiben ohne Offsetänderung:

```c
ssize_t pread(int fd, void * buffer, size_t n, off_t offset)
ssize_t pwrite(int fd, void * buffer, size_t n, off_t offset)
```

Wie `read` bzw. `write`. Statt des Offsets von `fd` wird der zusätzliche Parameter `offset` verwendet.
(`off_t` ⩾ `signed int`) Der Offset von `fd` wird ***nicht*** verändert.

2.3.7. Unterschiede Windows und POSIX

- Bestandteile von Pfaden werden durch ***Backslash*** (\) getrennt.
  (müssen darum in C-Strings doppelt geschrieben werden, da \ Escape-Character ist)
- Ein ***Wurzelverzeichnis pro*** Datenträger/Partition
- Andere File-Handling-Funktionen *(CreateFile, ReadFile, WriteFile, SetFilePointer, CloseHandle)*

2.4. C STREAM API

- ***Unabhängig vom Betriebssystem:*** für POSIX und Windows gleich
- ***Stream-basiert:*** zeichen-orientiert *(Ist dafür da, mit Text zu arbeiten)*
- Kann ***gepuffert*** oder ***ungepuffert*** sein. Für Dateien im Normalfall ***gepuffert***. Transferiert selbstständig grössere Daten-Blöcke zwischen Datei und Puffer.
- Hat einen eigenen ***File-Position-Indicator***: Bei gepufferten Streams bestimmte Position im Puffer, bei ungepufferten Streams entspricht dieser dem Offset des File-Descriptors.

2.4.1. Streams

Datenstruktur `FILE` enthält ***Informationen über einen Stream***. Soll ***nicht direkt verwendet werden***, sondern nur über von C-API erzeugte Pointer ***`(FILE *)`***. Soll nicht kopiert werden, Pointer an sich kann von API als ID verwendet werden.

Drei definierte Standard-Streams analog zu den Standard-FDs:
`FILE *stdin`, `FILE *stdout`, `FILE *stderr`

2.4.2. Öffnen einer Datei: `FILE * fopen(char const *path, char const *mode)`

erzeugt `FILE`-Objekt (und damit Stream) für Datei an `path`. `mode` gibt Flags analog zu `open` als nullterminierten String an:

- ***"r":*** wie `O_RDONLY` *(Datei lesen)*
- ***"w":*** wie `O_WRONLY | O_CREAT | O_TRUNC` *(in neue oder bestehende geleerte Datei schreiben)*
- ***"a":*** wie `O_WRONLY | O_CREAT | O_APPEND` *(in neue oder bestehende Datei anfügen)*
- ***"r+":*** wie `O_RDWR` *(Datei lesen & schreiben)*
- ***"w+":*** wie `O_RDWR | O_CREAT | O_TRUNC` *(neue oder geleerte bestehende Datei lesen & überschreiben)*
- ***"a+":*** wie `O_RDWR | O_CREAT | O_APPEND` *(neue oder bestehende Datei lesen & an Datei anfügen)*

Gibt Pointer auf erzeugtes `FILE`-Objekt zurück oder 0 bei Fehler.

`FILE * fdopen(int fd, char const * mode)` ist wie `fopen()`, aber statt Pfad wird direkt der File-Deskriptor übergeben.

`int fileno (FILE *stream)` gibt File-Deskriptor zurück, auf den sich der Stream bezieht, oder −1 bei Fehler.

Da die POSIX- & Stream-API ***unterschiedliche Offsets*** haben, sollte man nach dem Umwandeln mit den obigen Funktionen die «vorherige» API ***nicht mehr verwenden***, da es wie bei mehreren FDs auf dieselbe Datei zu ***Konflikten*** kommen kann.

2.4.3. Schliessen einer Datei: `int fclose(FILE *file)`

Ruft `fflush()` auf, schliesst den durch `file` bezeichneten Stream, entfernt `file` aus Speicher und gibt 0 zurück wenn OK, andernfalls `EOF`.

2.4.4. Flushen einer Datei: `int fflush(FILE *file)`

Schreibt eventuell zu schreibenden Inhalt aus dem Hauptspeicher in die Datei. Wird automatisch aufgerufen, wenn der Puffer voll ist oder die Datei geschlossen wird. Gibt 0 zurück wenn OK, andernfalls `EOF`.

2.4.5. Lesen aus einer Datei: `int fgetc(FILE *stream)`

Liest das nächste Byte vom `stream` als ***unsigned char*** und gibt es als ***int*** zurück *(weil man den nächstgrösseren Dateityp `int` benötigt, um Fehlercodes abzubilden)*. Erhöht den ***File-Position-Indicator*** um 1.

`char * fgets(char *buf, int n, FILE *stream)` liest bis zu $n-1$ Zeichen aus `stream`, bis Newline oder `EOF` auftritt. Hängt eine 0 an, und erzeugt damit null-terminierten String. Gibt `buf` zurück, oder 0 wenn ein Fehler auftrat. Erhöht den File-Position-Indicator entsprechend der gelesenen Zeichen.

Lesen rückgängig machen: `int ungetc(int c, FILE *stream)`

Schiebt `c` zurück in den `stream` auf den ***Unget-Stack***. `fgetc` bevorzugt immer den Unget-Stack: `c` wird bei der nächsten Leseoperation so zurückgegeben, als ob es an der Stelle gestanden hätte. Die Datei selbst wird ***nicht*** verändert. Der Unget-Stack hat ***mindestens Grösse 1: Funktioniert mindestens einmal***. Gibt `c` zurück, oder `EOF` im Fehlerfall.

2.4.6. Schreiben in eine Datei: `int fputc(int c, FILE *stream)`

Konvertiert `c` in ***unsigned char*** und schreibt diesen auf `stream`. Gibt entweder `c` zurück oder `EOF`. Erhöht den File-Position-Indicator um 1.

`int fputs(char *s, FILE *stream)` schreibt die Zeichen vom String `s` bis zur terminierenden 0 in `stream`. Die terminierende 0 wird ***nicht*** mitgeschrieben. Gibt im Fehlerfall `EOF` zurück.

2.4.7. Dateiende und Fehler:

- `int feof(FILE *stream)` gibt 0 zurück, wenn Dateiende ***noch nicht*** erreicht wurde
- `int ferror(FILE * stream)` gibt 0 zurück, wenn ***kein*** Fehler auftrat

```c
int return_value = fgetc (stream);
if (return_value == EOF) {
  if (feof(stream) != 0) {
    // EOF reached
  } else if (ferror(stream) != 0) {
    // Error occurred, check errno
  } // feof() and ferror() need to be checked separately
}
```

2.4.8. Manipulation des File-Position-Indicator (FPI)

- **`long ftell(FILE *stream)`** gibt den gegenwärtigen FPI zurück.
  (POSIX-Erweiterung von C: `ftello` mit Rückgabetyp `off_t`)
- **`int fseek (FILE *stream, long offset, int origin)`** setzt den FPI, analog zu `lseek`.
  (POSIX-Erweiterung von C: `fseeko` mit `off_t` als Typ für `offset`)
- **`int rewind (FILE *stream)`** setzt den Stream zurück.
  Äquivalent zu `fseek(stream, 0, SEEK_SET)` und Löschen des Fehlerzustands.

3. PROZESSE

Wenn ein Prozessor nur ein einziges Programm ausführt, laufen auf ihm ***nur zwei Software-Akteure:*** Das ***Programm*** und das ***Betriebssystem***.

Dieses System nennt man Monoprogrammierung: ***Kommunikation*** vom Programm zum OS auf SW-Ebene über ***C-Funktionsaufrufe***. Das Programm ***kennt nur sich selbst*** und das OS.

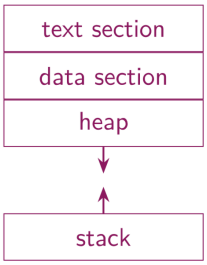
Moderne Prozessoren bieten ***genügend Rechenleistung***, um ***viele Programme*** gleichzeitig ausführen zu können. All diese Programme müssen ***gleichzeitig*** im Hauptspeicher sein. OS muss jedem Programm ***nacheinander*** (nicht gleichzeitig) Zeit auf dem Prozessor zuweisen. Das OS benötigt eine ***Verwaltungseinheit*** für Programme, die laufen sollen: ***den Prozess***.

Die ***Monoprogrammierung*** soll jedoch erhalten bleiben. Aufgabe des OS ist es, Programme voneinander zu ***isolieren***. Jedem Prozess ist ein ***virtueller Adressraum*** zugeordnet.

3.1. GRUNDLAGEN

Ein Prozess umfasst:

- Das ***Abbild eines Programms*** im Hauptspeicher, die ***text section***
- die ***globalen Variablen des Programms***, die ***data section***
- Speicher für den ***Heap***
- Speicher für den ***Stack***



3.1.1. Prozess vs Programm

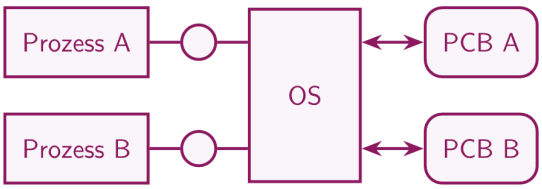
- Ein ***Programm*** ist ***passiv***: beschreibt bestimmte Abläufe *(wie ein Rezept)*
- Ein ***Prozess*** ist ***aktiv***: führt Abläufe aus *(das Kochen des Rezeptes)*

Ein Programm kann als verschiedene, voneinander unabhängige Prozesse ***mehrfach*** ausgeführt werden. Unter POSIX kann ein Prozess mehrere Programme ***nacheinander*** ausführen.

3.2. BETRIEBSSYSTEMSICHT

Das Betriebssystem hält Daten über jeden Prozess in jeweils einem ***Process Control Block (PCB)*** vor.

3.2.1. Process Control Block (PCB)



Speicher für alle Daten, die das OS benötigt, um die Ausführung des Prozesses ins Gesamtsystem zu integrieren, u.a.:

- Eigene ***Process ID***, Parent ID und andere wichtige IDs
- Speicher für den ***Zustand*** des Prozessors *(Prozesskontext)*
- ***Scheduling-Informationen*** *(welcher Prozess ist wann an der Reihe)*
- Daten zur ***Synchronisation*** und ***Kommunikation*** zwischen Prozessen
- ***Dateisystem-relevante*** Informationen *(z.B. offene Dateien)*
- ***Security-Informationen*** *(Prozess selber sieht diese nicht)*

3.2.2. Interrupts und Prozesse

Wenn ein Interrupt auftritt, muss der ***Kontext*** des aktuellen Prozesses im dazugehörigen PCB gespeichert werden *(context save)*: Register, Flags, Instruction Pointer, MMU-Konfiguration

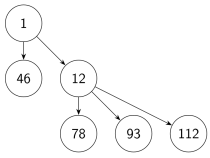
Dann wird der ***Interrupt-Handler*** aufgerufen, der je nach Bedarf den Kontext ***komplett überschreiben*** kann. Nach dem Ende des Interrupt-Handlers wird der Kontext des Prozesses aus seinem PCB ***wiederhergestellt*** *(context restore)*.

Ablauf eines Kontext-Wechsels: OS sichert Kontext von Prozess A im PCB A und stellt den Kontext von Prozess B aus dem PCB B wieder her. Nach Rücksprung aus dem Interrupt-Handler läuft somit Prozess B statt A.

3.2.3. Prozess-Erstellung

Um aus einem Programm einen Prozess zu machen, muss das OS ***einen Prozess erzeugen*** und ***ein Programm in diesen Prozess laden***. Unter POSIX sind beide Schritte getrennt, unter Windows finden beide in einer einzigen Funktion statt.

3.2.4. Prozess-Hierarchie



In POSIX hat jeder Prozess ausser Prozess 1 genau ***einen*** Parent-Prozess. Jeder Prozess kann ***beliebig viele*** Child-Prozesse haben. Dadurch wird eine ***Baum-Struktur*** definiert: Die ***Prozess-Hierarchie***. Diese kann mit dem Tool `pstree` angezeigt werden.

3.3. PROZESS API

3.3.1. Die Funktion `fork()`

`pid_t fork(void)` erzeugt eine ***exakte Kopie*** (Child C) des Prozesses (Parent P), aber: C hat eine ***eigene Prozess-ID*** und als ***Parent-Prozess-ID*** die ID von P. Die Funktion führt in ***beiden*** Prozessen den Code an derselben Stelle fort: Am Rücksprung aus `fork`.

- In P bei ***Erfolg***: Gibt die Prozess-ID von C zurück (> 0)
- In P bei ***Misserfolg***: Gibt −1 zurück und Fehlercode in `errno`
- In C: Gibt 0 zurück

```c
pid_t new_pid = fork();
if (new_pid > 0) {
  // code running in parent
} else if (new_pid == 0) {
  // code running in child
}
```

3.3.2. Die Funktion `exit()`

`void exit(int code)` entspricht dem gleichnamigen Betriebssystem-Aufruf. Kann an jeder Stelle im Programm verwendet werden und bietet somit eine Alternative zum «Rücksprung» aus `main()`. Springt nie zurück, sondern ***beendet das Programm***. `code` ist der Code, der am Ende des Prozesses zurückgegeben wird *(return/exit value des Programms)*.

3.3.3. Die Funktion `wait()`

`pid_t wait(int *status)` unterbricht den Prozess, bis einer seiner Child-Prozesse beendet wurde. Gibt die Statusinformationen über den `int` zurück, auf den `status` zeigt *(Out-Parameter)*. Der Status wird durch Macros aus dem Header `<sys/wait.h>` abgefragt:

- ***`WIFEXITED(*status)`:*** $\neq$ `0`, wenn Child ordnungsgemäss beendet wurde. *(wait-if)*
- ***`WEXITSTATUS(*status)`:*** Exit-Code von Child

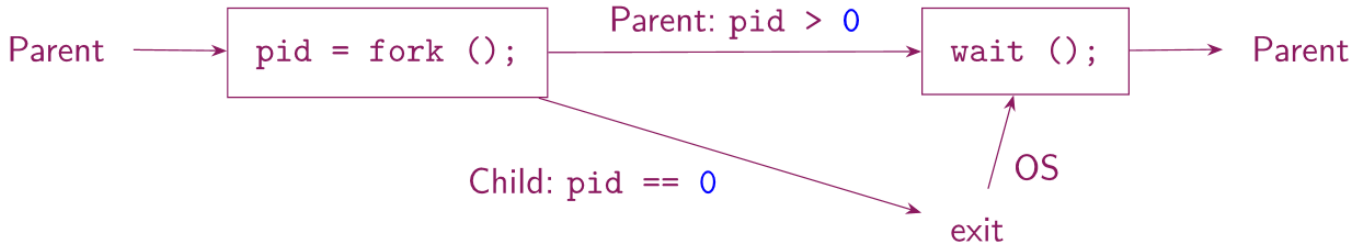
Gibt −1 zurück, wenn ein Fehler auftritt, Fehlercode in `errno`. Mögliche Fehler:
- `ECHILD`: Hat kein Child mehr, um darauf zu warten.
- `EINTR`: Wurde von einem Signal unterbrochen.

3.3.4. Die Funktion `waitpid()`

`pid_t waitpid (pid_t pid, int *status, int options)` ist wie `wait()`, aber `pid` bestimmt, auf welchen Child-Prozess man warten will.
- ***`pid > 0`:*** Wartet nur auf den Child-Prozess mit dieser `pid`
- ***`pid == -1`:*** Wartet auf irgendeinen Child-Prozess (= `wait()`)
- ***`pid == 0`:*** wartet auf alle Child-Prozesse welche dieselbe Prozessgruppen-ID wie der Parent haben
- ***`pid < -1`:*** wartet auf alle Child-Prozesse welche dieselbe Prozessgruppen-ID wie der absolute `pid`-Wert haben

Gibt −1 zurück, wenn ein Fehler auftritt, Fehlercode in `errno`. Hat diesselben Fehler wie `wait()`.

3.3.5. Zusammenspiel von `fork()` und `wait()`




```c
void spawn_worker (...) {
  if (fork() == 0) {
    // ... do something in worker process
    exit(0); // exit from worker process
  } // Parent process does nothing in functions
}

for (int i = 0; i < n; ++i) {
  spawn_worker(...);
}
// ... do something in parent process
do { pid = wait(0); } while (pid > 0 || errno != ECHILD); // wait for all children
```

3.3.6. `exec()`-Funktionen

Es gibt 6 `exec()`-Funktionen: `execl()`, `execle()`, `execlp()`, `execv()`, `execve()`, `execvp()`. Jede `exec`-Funktion ***ersetzt*** im gerade laufenden Prozess das Programmimage ***durch ein anderes Programmimage***.

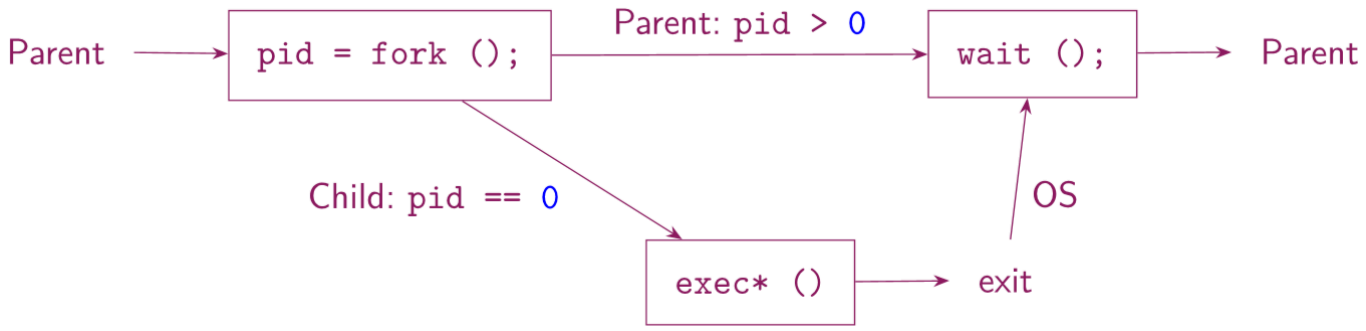
Bei jeder `exec`-Funktion müssen die ***Programmargumente spezifiziert*** werden.
- ***Bei den `execl*`-Funktionen als Liste*** *(`l` für Liste von Argumenten)*: `execl(path, arg0, arg1, ...)`
- ***Bei den `execv*`-Funktionen als Array*** *(`v` für Vektor/Array)*: `execv(path, argv)`

Die `exec*e`-Funktionen erlauben die ***Angabe eines Arrays*** für die ***Umgebungsvariabeln***, in den anderen Versionen bleiben die Umgebungsvariablen gleich.

Die `exec*p`-Funktionen suchen den ***Dateinamen*** über die Umgebungsvariable ***`PATH`***, die anderen verwenden absolute/relative Pfade.

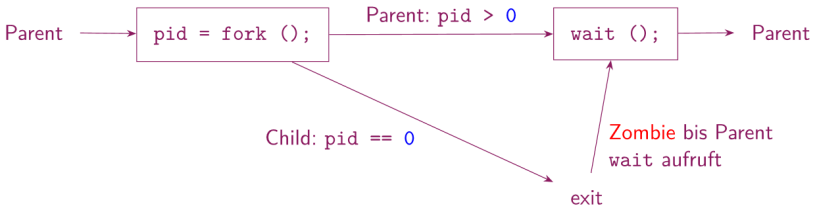
Aufrufen der Executable	*Zustand der Environment-Variabeln*	*Programmargumente als Liste*	*Programmargumente als Array*
Angabe des Pfads	mit neuem Environment	`execle()`	`execve()`
	mit altem Environment	`execl()`	`execv()`
Suche über `PATH`		`execlp()`	`execvp()`

3.3.7. Zusammenspiel von `fork()`, `exec()` und `wait()`



3.3.8. Zombie-Prozess

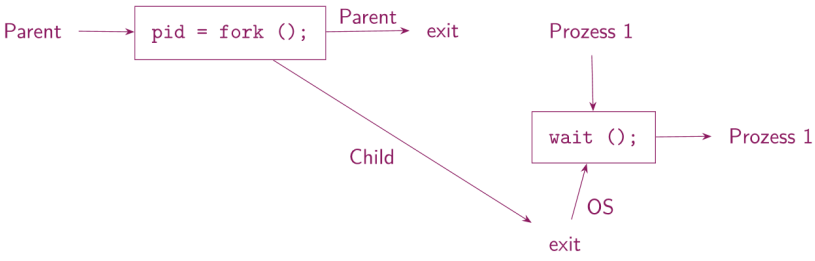
Wenn ein Prozess C ***beendet*** wird, ist sein Parent-Prozess P verantwortlich dafür, ***auf jeden Fall*** `wait()` aufzurufen. Das OS weiss nicht, ***wann*** das passieren wird. Das OS muss die Statusinformationen von C solange vorhalten, bis P `wait()` aufruft. ***C ist zwischen seinem Ende und dem Aufruf von `wait()` durch P ein Zombie-Prozess*** *(tot, aber noch nicht entfernt)*.



Dauerhafter Zombie-Prozess: Bleibt ein Prozess C längere Zeit ein Zombie, bedeutet das, dass sein Parent P `wait()` längere Zeit nicht aufruft. Vermutlich hat P einen Fehler. Die Situation kann bereinigt werden, indem P gestoppt wird und C somit an Prozess 1 übertragen wird.

3.3.9. Orphan-Prozess

Wird ein Prozess P ***beendet***, haben seine Child-Prozesse C keinen Parent-Prozess mehr. Sie ***verwaisen*** und werden zu ***Orphan-Prozessen***. P kann nicht mehr seiner Verantwortung nachkommen und auf C warten. C würden bei ihrem Ende zu ***dauerhaften Zombie-Prozessen*** und würden nie entfernt werden.



Damit das nicht passiert, werden beim Ende eines Prozesses P all seine Child-Prozesse an den Prozess mit der `pid=1` ***übertragen***. Dieser Prozess ruft in einer ***Endlosschleife*** `wait()` auf und beendet somit alle ihm übertragenen Orphan-Prozesse.

3.3.10. Die Funktion `sleep()`

`unsigned int sleep (unsigned int seconds)` unterbricht die Ausführung, bis die Anzahl der Sekunden ***ungefähr*** verstrichen ist. Kann vom ***System auch unterbrochen werden***. Gibt die Anzahl Sekunden zurück, die vom Schlaf noch verblieben sind.

3.3.11. Die Funktion `atexit()`

`int atexit (void (*function)(void))` dient dazu, dass ein Programm kurz vor seinem Ende letzte ***Aufräumarbeiten*** durchführen kann. Diese Aufräum-Funktionen werden dann nach einem Aufruf von `exit` in ***umgekehrter Reihenfolge der Registrierung*** aufgerufen *(Funktionen werden also von unten nach oben ausgeführt)*.

3.3.12. Funktionen zum Lesen von PIDs

`pid_t getpid(void)` und `pid_t getppid(void)` geben die Prozess-ID des aufrufenden Prozesses bzw. seines Parent-Prozesses zurück.

```c
int main() {
  pid_t my_pid = getpid();
  pid_t my_parent_pid = getppid();
  printf("I am %d, my parent is %d\n", my_pid, my_parent_pid);
}
```

4. PROGRAMME UND BIBLIOTHEKEN

4.1. C TOOLCHAIN

`C-Quelle → Präprozessor → Bereinigte C-Quelle → Compiler → Assembler-Datei → Assembler → Objekt-Datei → Linker → Executable`

- ***Präprozessor:*** Die Ausgabe des Präprozessors ist eine reine C-Datei (Translation-Unit) ohne Makros, Kommentare oder Präprozessor-Direktiven.
- ***Linker:*** Der Linker verknüpft Objekt-Dateien (und statische Bibliotheken) zu Executables oder dynamischen Bibliotheken. Löst Referenzen der Objekt-Dateien untereinander soweit wie möglich auf. Executable und dynamische Bibliotheken müssen vollständig aufgelöst sein.

4.2. LOADER UND DAS ELF

Der ***Loader*** lädt Executables und eventuelle ***dynamische*** Bibliotheken dieser in den Hauptspeicher. Executable und dynamische Bibliotheken müssen also ***alle Informationen*** enthalten, die der Loader benötigt. Lädt ***keine statische Bibliotheken***, diese werden bereits vorher im Kompilationsprozess vom Linker mit Executable oder dynamischer Bibliothek verknüpft.

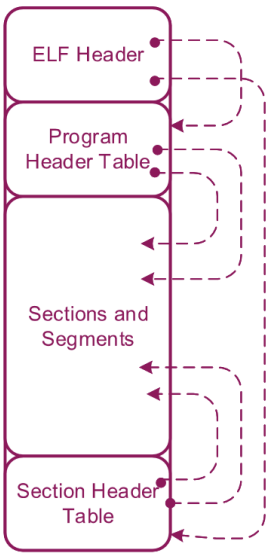
4.2.1. Linux Loader

Eine Funktion der `exec*`-Familie erhält `syscall`. Diese wird auf `sys_execve` übersetzt. ***Sucht*** Datei, ***prüft*** Rechte (x-Bits) und ***öffnet*** die spezifizierte Datei. ***Zählt und kopiert*** die Argumente und Umgebungsvariablen *(weil execve)*. ***Übergibt den Request*** an jeden registrierten «Binary Handler» *(für verschiedene Dateiformate: ELF, a.out etc.)*. Diese versuchen nacheinander jeweils die ***Datei zu laden*** und zu interpretieren → ***Ausführung des neuen Programms***.

4.2.2. ELF (Executable and Linking Format)

Binär-Format, das Kompilate spezifiziert. Eigentlich zwei Formate / Views, werden aber oft beide benötigt: ***Linking View*** *(wichtig für Linker)* und ***Execution View*** *(wichtig für Loader)*. Verwendung:

- ***Object-Files:*** Linking View
- ***Programme:*** Execution View
- ***Shared Objects (Dynamische Bibliotheken):*** Linking View und Execution View

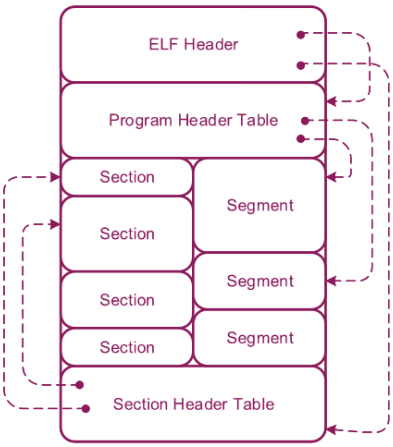


4.2.3. Struktur des ELF

- ***Header***
- ***Programm Header Table*** *(nur in Execution View erforderlich)*
- ***Segmente*** *(nur in Execution View erforderlich)*
- ***Section Header Table*** *(nur in Linking View erforderlich)*
- ***Sektionen*** *(nur in Linking View erforderlich)*

Segmente und Sektionen

Segmente und Sektionen sind jeweils eine ***andere Einteilung für die gleichen Speicherbereiche***. Die ***View des Loaders*** sind die ***Segmente***: Diese definieren die Portionen, die in den Hauptspeicher geladen werden. Die ***View des Compilers*** sind die ***Sektionen***: Diese definieren «gleichartige» Daten *(z.B. `.data`, `.text`)*. Der ***Linker*** vermittelt zwischen beiden Views: kombiniert gleichnamige Sektionen aus unterschiedlichen Objekt-Dateien und definiert Segmente.



Header

Der Header *(52 Byte)* beschreibt ***den Aufbau*** der Datei:

- ***Typ:*** Relozierbar *(beliebig verschiebbar im Speicher)*, Ausführbar, Shared Object
- ***32-bit oder 64-bit***
- ***Encoding:*** little-endian oder big-endian
- ***Maschinenarchitektur:*** z.B. i386, Motorola 68k
- ***Entrypoint:*** Adresse, an der das Programm starten soll *(Default: `_start`)*
- ***Infos zu den Einträgen in der Program Header Table:*** Relative Adresse, Anzahl und Grösse
- ***Infos zu den Einträgen in der Section Header Table:*** Relative Adresse, Anzahl und Grösse

Segment/Program Header Table und Segmente

Die Segment Header Table *(SHT/PHT)* ist eine Tabelle mit n Einträgen. Jeder Eintrag (je 32 Byte) ***beschreibt ein Segment:***

- Segment-Typ und Flags
- Offset und Grösse in der Datei
- Virtuelle Adresse und Grösse im Speicher *(möglich zusätzlich auch physische Adresse)*

Die SHT ist die ***Verbindung zwischen den Segmenten im RAM und im File***. Die PHT definiert, wo ein Segment in der Datei liegt und wohin der Loader das Segment in den RAM laden soll.

Achtung: Grösse der Datei und Grösse im Speicher können unterschiedlich sein. Es kann auch Speicher reserviert werden. Deswegen kann Dateigrösse 0 sein, aber Speicher z.B. 5MB.
Segmente werden vom Loader ***zur Laufzeit*** verwendet: Der ***Loader*** lädt bestimmte Segmente in den Speicher und kann weitere Segmente für andere Informationen verwenden *(dynamisches Linken oder Meta-Informationen)*.

Section Header Table und Sektionen

Die Section Header Table (auch SHT) ist eine Tabelle mit m Einträgen (m meist $\neq n$). Jeder Eintrag (je 40 Byte) ***beschreibt eine Sektion***:

- ***Name:*** Referenz auf String Table
- ***Section-Typ*** und Flags
- ***Offset*** und ***Grösse*** in der Datei
- ***Spezifische Informationen*** je nach Sektions-Typ

Sektionen werden vom ***Linker*** verwendet: Sammelt alle Sektionen aus allen Object-Files zusammen. ***Verschmilzt*** Sektionen ***gleichen Namens*** aus verschiedenen Object-Files und ***erzeugt ausführbares Executable***.

Sektionstypen *(Auswahl)*

- ***`SHT_PROGBITS`:*** Daten definiert vom Programm, Linker interpretiert Inhalt nicht
- ***`SHT_SYMTAB`:*** Symbol-Tabelle
- ***`SHT_STRTAB`:*** String-Tabelle
- ***`SHT_REL/RELA`:*** Relokations-Informationen
- ***`SHT_HASH`:*** Hashtabelle für Symbole
- ***`SHT_DYNAMIC`:*** Informationen für dynamisches Linken
- ***`SHT_NOBITS`:*** Sektionen ohne Daten in der Datei

Sektionsattribute

- ***`SHF_WRITE`:*** Daten dieser Sektion sollen bei Ausführung ***schreibbar*** sein *(SHF für Flag)*
- ***`SHF_ALLOC`:*** Daten dieser Sektion sollen bei Ausführung ***im Speicher liegen***
- ***`SHF_EXECINSTR`:*** Daten dieser Sektion stellen ***Maschinencode*** dar

Spezielle Sektionen *(Auswahl)*

- ***.bss:*** Uninitialisierte Daten *(SHT_NOBITS, SHF_ALLOC, SHF_WRITE)*
- ***.data / data1:*** Initialisierte Daten *(SHT_PROGBITS, SHF_ALLOC, SHF_WRITE) - data1 ist historisch*
- ***.debug:*** Debug-Informationen *(SHT_PROGBITS)*
- ***.rodata / .rodata1:*** Read-Only Daten *(SHT_PROGBITS, SHF_ALLOC)*
- ***.text:*** Ausführbare Instruktionen *(SHF_PROGBITS, SHF_ALLOC, SHF_EXECINSTR)*
- ***.symtab:*** Symbol-Tabelle *(SHT_SYMTAB)*
- ***.strtab:*** String-Tabelle *(SHT_STRTAB)*

String-Tabelle

Bereich in der Datei, der nacheinander ***null-terminierte Strings enthält***. Strings werden ***relativ zum Beginn der Tabelle*** referenziert *(z.B. Tabelle beginnt bei Dateioffset 1234, String bei 1238 → String-Referenz = 4)*.
Enthält typischerweise ***Namen von Symbolen*** und ***keine String-Literale*** *(z.B. «Hello World» - diese sind typischerweise in `.rodata`)*

Symbole & Symboltabelle

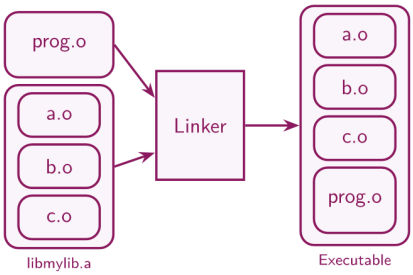
Die Symboltabelle enthält jeweils ***einen Eintrag je Symbol***. Ein Symbol hat ***16 Byte***.

- 4 Byte ***Name***: Referenz in String-Tabelle
- 4 Byte ***Wert***: Je nach Symboltyp, kann z.B. Adresse sein
- 4 Byte ***Grösse***: Grösse des Symbols *(z.B. Länge einer Funktion)*
- 4 Byte ***Info***: Typ *(Objekt, Funktion, Sektion…)*, Binding-Attribute, Referenz auf Sektions-Header

4.3. BIBLIOTHEKEN

4.3.1. Statische Bibliotheken

Statische Bibliotheken sind ***Archive von Objekt-Dateien***. Archive sind Dateien, die andere Dateien enthalten *(wie ein ZIP ohne Kompression)*, werden mit dem Tool «`ar`» erzeugt. Per Konvention folgen Bibliotheksnamen dem Muster `lib<name>.a`. Referenziert wird dann nur `<name>`: `clang -lmylib ...`.

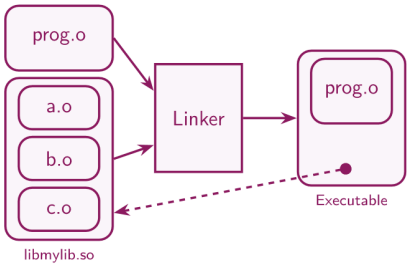


Der Linker behandelt statische Bibliotheken wie ***mehrere Objekt-Dateien***. Alle gelinkten statischen Bibliotheken werden vom Linker im Programm-Image ***verteilt***, alle Variablen und Funktionen werden auf absolute Adressen ***fixiert***.

Ursprünglich gab es ***nur statische Bibliotheken***. Das war ***einfach*** zu implementieren, jedoch müssen Programme bei Änderungen in Bibliotheken ***neu erstellt*** werden und die ***Funktionalität ist fix*** *(Keine Plugins möglich)*.

4.3.2. Dynamische Bibliotheken

Dynamische Bibliotheken linken erst zur ***Ladezeit*** bzw. Laufzeit des Programms. ***Höherer Aufwand*** für Programmierer, Compiler, Linker und OS. Das Executable enthält nur noch Referenz auf Bibliothek. Vorteile davon sind:



Entkoppelter Lebenszyklus: Das Programm kann ***Updates erhalten***, ohne das Binary zu ändern. Funktionalität kann unabhängig voneinander ***geupdated*** werden. Bugfixes können ***direkt*** zur Anwenderin gebracht werden.

Verzögertes Laden: Das Programm muss ***nur die Bibliotheken laden***, die es minimal braucht. Führt zu schnelleren Ladezeiten.

Flexibler Funktionsumfang: Programme können um Funktionalitäten ***ergänzt*** werden, die beim Schreiben nicht vorgesehen war. ***Ablauf:*** Programm definiert API für Plugin-Bibliotheken und enthält Mechanismus, anhand des Modulnamens Bibliotheken zu finden.

4.3.3. POSIX: Shared Objects API

`void * dlopen (char * filename, int mode)` ***öffnet eine dynamische Bibliothek*** und gibt ein ***Handle*** darauf zurück. `mode` gibt Art an, wie mit der Bibliothek umgegangen wird:
- ***`RTLD_NOW`:*** Alle Symbole werden beim Laden der Bibliothek gebunden
- ***`RTLD_LAZY`:*** Symbole werden bei Bedarf gebunden
- ***`RTLD_GLOBAL`:*** Symbole können beim Binden anderer Objekt-Dateien verwendet werden *(damit andere Libs diese benutzen können)*
- ***`RTLD_LOCAL`:*** Symbole werden nicht für andere Objekt-Dateien verwendet

`void * dlsym (void * handle, char * name)` gibt die ***Adresse des Symbols*** `name` aus der mit `handle` bezeichneten ***Bibliothek*** zurück. Keine Typinformationen, nur Adresse. Es ist also weder klar, ob es sich um Funktionen oder Variablen handelt, noch welche Signatur bzw. welchen Typ diese haben.

```
// type "func_t" is an address of a function with a int param and int return type
typedef int (*func_t)(int);
handle = dlopen("libmylib.so", RTLD_NOW); // open library
func_t f = dlsym(handle, "my_function"); // write address of "my_function" into a func_t
int *i = dlsym(handle, "my_int"); // get address of "my_int"
(*f)(*i); // call "my_function" with "my_int" as parameter
```

Dabei ist `f` die Adresse der Funktion namens `my_function` in `libmylib.so.1` und `i` die Adresse der globalen Variable namens `my_int`. Beides sind Symbole, die von der Library exportiert werden.

`int dlclose (void * handle)` schliesst das durch `handle` bezeichnete, zuvor von `dlopen` geöffnete Objekt. Gibt 0 zurück, wenn erfolgreich. ***Aufgepasst vor offenen Pointer auf Library-Symbole!***

`char * dlerror()` gibt die Fehlermeldung als null-terminierten String zurück, wenn ein Fehler aufgetreten war.

4.3.4. Shared-Object Konventionen

Automatisches Laden von Shared Objects

Shared Objects können ***automatisch*** bei Bedarf geladen werden. Im Executable (ELF) muss eine Referenz auf das Shared Object (ELF) hinterlegt sein (Dependency). Das OS sucht automatisch beim Programmstart die richtigen Bibliotheken.

Shared Objects Benennungsschema
- ***Linker-Name:*** `lib + Bibliotheksname + .so` *(z.B. libmylib.so)*
- ***SO-Name:*** `Linker-Name + . + Versionsnummer` *(z.B. libmylib.so.2)*
- ***Real-Name:*** `SO-name + . + Unterversionsnummer` *(z.B. libmylib.so.2.1)*

Sind in POSIX meist gelinkt im Dateisystem: Linker → SO → Real *(libmylib.so → libmylib.so.2 → libmylib.so.2.1)*
Real-Name wird beim ***Erstellen des Shared Objects*** verwendet. Die ***Versionsnummer*** wird erhöht, wenn sich die Schnittstelle ***ändert***. Die ***Unterversionsnummer*** wird erhöht, wenn die Schnittstelle gleich bleibt *(Bugfixes)*. Der Linker verwendet den Linker-Namen, der Loader verwendet den SO-Namen.

Shared Object-Koexistenz verschiedener Versionen

Alle Versionen und Unterversionen können gleichzeitig existieren und verwendet werden. Programme können bei Bedarf die Unterversion ganz präzise angeben *(`libmylib.so.2` zeigt auf die neuste 2-er Version)*.

4.3.5. Erstellen von Bibliotheken

Erstellen von statischen Bibliotheken mit dem `clang`

Zuerst kompilieren mit `clang -c f1.c -o f1.o; clang -c f2.c -o f2.o`, dann zusammenfügen zu einem Archiv: `ar r libmylib.a f1.o f2.o` *(`r` fügt Dateien hinzu oder überschreibt existierende)*

Dynamische Bibliotheken mit `clang` kompilieren

Falls zusätzlich zu den Befehlen oben `-fPIC` für Position-Independent Code verwendet wird, kann ein spezielles Image erzeugt werden mit `clang -shared -Wl, -soname, libmylib.so.2 -o libmylib.so.2.1 f1.o f2.o -lc`, wobei ***`-shared`*** = Erzeugen eines Shared Objects, ***`-Wl,`*** = Weitergeben der folgenden Option an den Linker, ***`-soname`*** = spezifizieren des SO-Namens `libmylib.so.2`, ***`-lc`*** = Einbinden der Standard C-Bibliothek (`libc.so`).

4.3.6. Verwenden von Bibliotheken

– ***Statische Bibliothek*** **(Link-time Library)*:*** `clang main.c -o main -L. -lmylib`
  (`-L.` fügt «.» zum Suchpfad hinzu, `-lmylib` bezieht sich auf die Bibliothek `mylib`, nach Konvention also auf Datei `libmylib.a`)
– ***Dynamische Bibliothek, die mit Programm geladen werden soll*** **(Load-time Library)*:***
  `clang main.c -o main -lmylib`
  (`-lmylib` bezieht sich auf `libmylib.so`, ohne `-L.` muss `libmylib.so` im OS installiert sein)
– ***Dynamische Bibliothek, die mit `dlopen` geladen werden soll*** **(Run-time Library)*:***
  `clang main.c -o main -ldl`
  (`-ldl` linkt auf (dynamische) Bibliothek `libdl.so`, die `dlopen` etc zur Verfügung stellt)

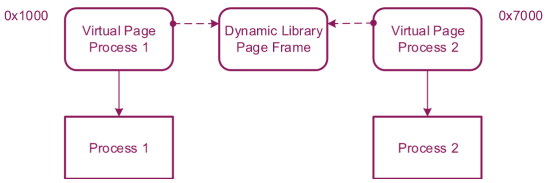
Shared Objects: Referenzierte Shared Objects sind im ***Executable*** abgelegt. ***`readelf -d`*** zeigt den ***Inhalt der dynamischen Sektion***. Typ der entsprechenden Einträge ist ***`NEEDED`***. Das Tool ***`ldd`*** zeigt ***alle, auch indirekt*** benötigten Shared Objects an. Dazu führt es die Executable aus, sollte deshalb nur auf ***vertrauenswürdigen Executables*** ausgeführt werden. Nahezu alle Executables benötigen ***zwei Shared Objects***:

– ***`libc.so`:*** Standard C library
– ***`ld-linux.so`:*** ELF Shared Object loader. Wird indirekt vom OS aufgerufen, wenn ein Shared Object geladen werden soll. Findet und lädt nacheinander alle benötigten Shared Objects, danach rekursiv die Dependencies der geladenen Shared Objects.

4.3.7. Implementierung von dynamischen Bibliotheken

Dynamische Bibliotheken müssen ***verschiebbar*** sein und mehrere Bibliotheken müssen in den ***gleichen Prozess*** geladen werden können. Die Aufgabe des Linkers wird in den Loader / Dynamic Linker verschoben *(Load Time Relocation)*.

Dynamische Bibliotheken sollen ***Code zwischen Programmen teilen***. Code soll ***nicht mehrfach*** im Speicher abgelegt werden, auch wenn mehrere Programme die Bibliothek verwenden. Das kann durch ***Shared Memory*** gelöst werden. Jedes Programm kann eine ***eigene virtuelle Page*** für den Code definieren. Diese werden auf denselben Frame im RAM gemappt, so belegt der Code den Hauptspeicher nur einmal.



Code-Sharing erlaubt jedoch keinen ***Position-Dependent Code***. Wenn zwei Prozesse unterschiedliche virtuelle Seiten verwenden, an welchen Prozess werden die Adressen angepasst? Deshalb muss mit dynamischen Bibliotheken mit ***Position-Independent Code*** gearbeitet werden. Dieser verwendet ***keine absoluten Adressen***, sondern nur Adressen ***relativ zum Instruction Pointer***.

4.3.8. Position-Independent Code (PIC)

Für Position-Independent Code (PIC) muss der ***Prozessor relative Instruktionen anbieten.*** `x86_64` *(64-bit Prozessoren)* hat relative Funktionsaufrufe und Move-Instruktionen, `x86_32` *(32-bit Prozessoren)* nur relative Calls. Relative Moves können aber über relative Calls emuliert werden.

Relative Moves via Relative Calls

CPU legt bei einem Call die ***Rücksprungadresse auf den Stack***. Funktion `f` will relativen Move ausführen und ruft Hilfsfunktion `h` auf. `h` kopiert ***Rücksprungadresse*** vom Stack in ein ***Register*** und springt zurück. `f` hat nun die ***Rücksprungadresse = Instruction Pointer*** im Register und kann relativ dazu arbeiten, weil `f` nun weiss, wo sie im Speicher liegt.

Global Offset Table (GOT)

Existiert einmal pro dynamischer Bibliothek und Executable. Enthält ***pro Symbol***, welches von anderen Libs benötigt wird, ***einen Eintrag***. Im Code werden relative Adressen in die GOT verwendet. Der Loader füllt zur Laufzeit die Adressen in die GOT ein.

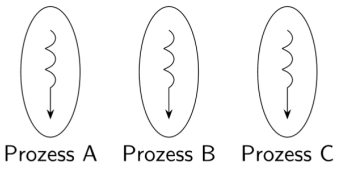
Procedure Linkage Table (PLT)

Implementiert ***Lazy Binding*** *(erst binden, wenn benötigt)*. Enthält ***pro Funktion einen Eintrag***. PLT-Eintrag enthält einen ***Sprungbefehl*** an Adresse in GOT-Eintrag. Der GOT-Eintrag zeigt zunächst auf eine ***Proxy-Funktion***, welche dann den Link zur richtigen Funktion sucht und den eigenen GOT-Eintrag überschreibt. ***Vorteil:*** Erspart bedingten Sprung *(teuer)*.

5. THREADS

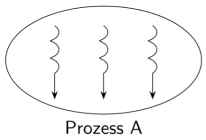
5.1. PROZESSMODELL

Jeder Prozess hat virtuell den ***ganzen Rechner*** für ***sich alleine***. ***Prozesse*** sind gut geeignet für ***unabhängige Applikationen***.
Nachteile: Realisierung ***paralleler Abläufe*** innerhalb derselben Applikation ist ***aufwändig***. ***Overhead*** zu gross falls nur kürzere Teilaktivitäten, ***gemeinsame Ressourcennutzung*** ist ***erschwert***.



5.2. THREADMODELL

Threads sind ***parallel ablaufende Aktivitäten innerhalb eines Prozesses***, welche auf ***alle*** Ressourcen im Prozess gleichermassen Zugriff haben *(Code, globale Variablen, Heap, geöffnete Dateien, MMU-Daten)*

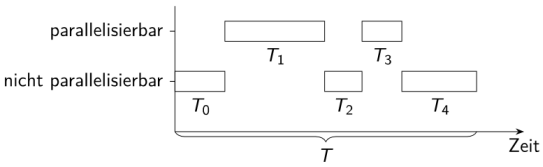


5.2.1. Thread als Stack + Kontext

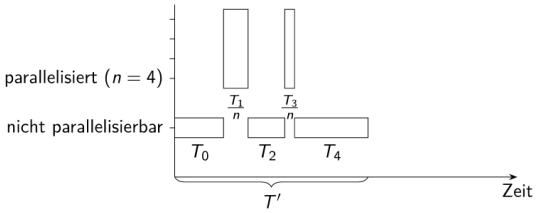
Jeder Thread benötigt einen ***eigenen Kontext*** und einen ***eigenen Stack***, weil er eine eigene Funktions-Aufrufkette hat. Diese Informationen werden häufig in einem ***Thread-Control-Block*** abgelegt *(Linux: Kopie des PCB mit eigenem Kontext)*.

5.3. AMDAHLS REGEL

Bestimmte Teile eines Algorithmus können ***nicht*** parallelisiert werden, weil sie ***voneinander abhängen***. Man kann für jeden Teil eines Algorithmus angeben, ob dieser ***parallelisiert*** werden kann oder nicht.



- T Ausführungszeit, wenn ***komplett seriell*** durchgeführt
  (Im Bild: $T = T_0 + T_1 + T_2 + T_3 + T_4$)
- n Anzahl der Prozessoren
- T' Ausführungszeit, wenn ***maximal parallelisiert*** *(gesuchte Grösse)*
- T_s Ausführungszeit für den Anteil, der ***seriell*** ausgeführt werden ***muss***
  (Im Bild: $T_s = T_0 + T_2 + T_4$)
- $T - T_s$ Ausführungszeit für den Anteil, der ***parallel*** ausgeführt werden ***kann*** *(Im Bild: $T - T_s = T_1 + T_3$)*
- $\frac{T-T_s}{n}$ Parallel-Anteil verteilt auf alle n Prozessoren *(Im Bild: $(T_1 + T_3)/n$)*
- $T_s + \frac{T-T_s}{n}$ Serieller Teil + Paralleler Teil *($= T'$)*



Die ***serielle Variante*** benötigt also höchstens f ***mal mehr Zeit*** als die ***parallele Variante*** *(wegen Overhead nur $\leq$)*:

$$f \leq \frac{T}{T'} = \frac{T}{T_s + \frac{T-T_s}{n}}$$

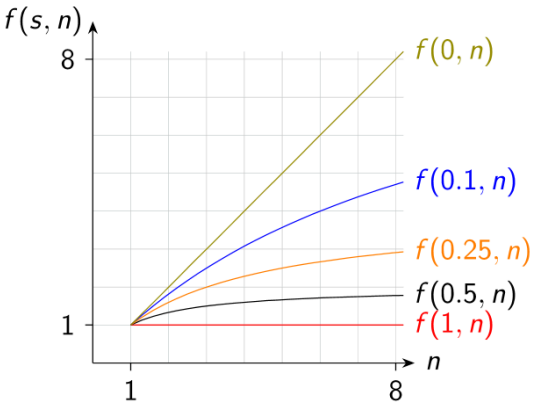
f heisst auch ***Speedup-Faktor***, weil man sagen kann, dass die parallele Variante maximal f-mal schneller ist als die serielle.

Definiert man $s = T_s/T$, also den seriellen Anteil am Algorithmus, dann ist $s \cdot T = T_s$. Dadurch erhält man f unabhängig von der Zeit:

$$f \leq \frac{T}{T_s + \frac{T-T_s}{n}} = \frac{T}{s \cdot T + \frac{T-s \cdot T}{n}} = \frac{T}{s \cdot T + \frac{1-s}{n} \cdot T} \quad \Rightarrow \quad f \leq \frac{1}{s + \frac{1-s}{n}}$$

5.3.1. Bedeutung

- Abschätzung einer ***oberen Schranke*** für den maximalen Geschwindigkeitsgewinn
- Nur wenn ***alles*** parallelisierbar ist, ist der Speedup ***proportional*** und ***maximal*** $f(0, n) = n$
- Sonst ist der Speedup mit ***höherer Prozessor-Anzahl*** immer ***geringer*** *(Kurve flacht ab)*
- $f(1, n)$: rein seriell



5.3.2. Grenzwert

Mit höherer Anzahl Prozessoren nähert sich der Speedup $1/s$ an:

$$\lim_{n\to\infty} \frac{1-s}{n} = 0 \qquad \lim_{n\to\infty} s + \frac{1-s}{n} = s \qquad \lim_{n\to\infty} \frac{1}{s + \frac{1-s}{n}} = \frac{1}{s}$$

5.4. POSIX THREAD API

5.4.1. `pthread_create()`

```
int pthread_create(
    pthread_t *thread_id, pthread_attr_t const *attributes,
    void * (*start_function) (void *), void *argument
)
```

Erzeugt einen Thread und gibt bei ***Erfolg 0*** zurück, sonst einen Fehlercode. Die ***ID*** des neuen Threads wird im ***Out-Parameter*** `thread_id` zurückgegeben. `attributes` ist ein ***opakes Objekt***, mit dem z.B. die ***Stack-Grösse*** spezifiziert werden kann.

Die ***erste Instruktion***, die der neue Thread ausführen soll, ist ein ***Aufruf der Funktion***, deren Adresse in `start_function` übergeben wird. Diese Funktion muss ebendiese Signatur haben. Zusätzlich übergibt der Thread das Argument `argument` an diese Funktion. Dies ist typischerweise ein Pointer auf eine Datenstruktur auf dem Heap. *(**Achtung:** Wird diese Struktur auf dem Stack angelegt, muss sichergestellt werden, dass während der Lebensdauer des Threads der Stack nicht abgebaut wird.)*

```
// Erstellung
struct T { // params of function
  int value;
};
void * my_start (void * arg) {
  struct T * p = arg;
  printf ("%d\n", p→value);
  free (arg);
  return 0:
}
```

```
// Verwendung
void start_my_thread (void) {
  struct T * t = malloc (sizeof (struct T));
  t→value = 109; // set argument
  pthread_t tid;
  pthread_create (
    &tid,
    0, // default attributes
    &my_start,
    t
  );
}
```

Thread-Attribute

Um Attribute ***anzugeben***, muss man nach folgendem ***Muster*** verfahren, da `pthread_attr_t` je nach Implementation ***weiteren Speicher*** benötigen kann:

```
pthread_attr_t attr; // Variabel erstellen
pthread_attr_init (&attr); // Variabel initialisieren
pthread_attr_setstacksize (&attr, 1 << 16); // 64kb Stackgrösse
pthread_create (..., &attr, ...); // Thread erstellen
pthread_attr_destroy (&attr); // Attribute löschen
```

5.4.2. Lebensdauer eines Threads

Ein Thread ***lebt*** solange, bis eine der folgenden Bedingungen eintritt:
- Er springt aus der Funktion ***`start_function`*** zurück
- Er ruft ***`pthread_exit`*** auf *(Normales `exit` terminiert Prozess)*
- Ein ***anderer Thread*** ruft ***`pthread_cancel`*** auf
- Sein ***Prozess*** wird ***beendet***.

5.4.3. `void pthread_exit (void *return_value)`

Beendet den Thread und gibt den `return_value` zurück. Das ist äquivalent zum ***Rücksprung aus `start_function` mit dem Rückgabewert***.

5.4.4. `int pthread_cancel (pthread_t thread_id)`

Sendet eine ***Anforderung***, dass der Thread mit `thread_id` ***beendet*** werden soll. Die Funktion ***wartet nicht***, dass der Thread ***tatsächlich beendet*** wurde. Der Rückgabewert ist 0, wenn der Thread existiert, bzw. `ESRCH` *(error_search)*, wenn nicht.

5.4.5. `int pthread_detach (pthread_t thread_id)`

Entfernt den Speicher, den ein Thread belegt hat, falls dieser ***bereits beendet*** wurde. Beendet den Thread aber ***nicht*** *(Erstellt Daemon Thread)*.

5.4.6. `int pthread_join (pthread_t thread_id, void **return_value)`

Wartet solange, bis der Thread mit `thread_id` ***beendet*** wurde. Nimmt den ***Rückgabewert*** des Threads im Out-Parameter ***`return_value`*** entgegen. Dieser kann ***`NULL`*** sein, wenn nicht gewünscht. Ruft ***`pthread_detach`*** auf.

5.4.7. `pthread_t pthread_self (void)`

Gibt die ***ID*** des ***gerade laufenden*** Threads zurück.

5.5. THREAD-LOCAL STORAGE (TLS)

In C geben viele System-Funktionen den Fehlercode nicht direkt zurück, sondern über `errno`, z.B. die `exec`-Funktionen. Wäre `errno` eine ***globale Variable***, würde folgender Code bei mehreren Threads ***unerwartetes Verhalten*** aufweisen:

```c
void f (void) {
  int result = execl (...);
  if (result == -1) {
    int error = errno; // Kann auch von einem anderen Thread sein
    printf ("Error %d\n", error);
  }
}
```

Thread-Local Storage (TLS) ist ein Mechanismus, der ***globale Variablen per Thread*** zur Verfügung stellt. Dies benötigt mehrere explizite Einzelschritte:

Bevor Threads erzeugt werden:
- Anlegen eines ***Keys***, der die TLS-Variable ***identifiziert***
- ***Speichern*** des Keys in einer ***globalen Variable***

Im Thread:
- ***Auslesen*** des Keys aus der globalen Variable
- ***Auslesen / Schreiben*** des Werts anhand des Keys über besondere Funktionen

5.5.1. `int pthread_key_create( pthread_key_t *key, void (*destructor) (void*) )`

Erzeugt einen ***neuen Key*** im Out-Parameter `key`. ***`pthread_key_t`*** ist eine ***opake Datenstruktur***. Für jeden Thread und jeden Key hält das OS einen Wert vom Typ ***`void *`*** vor. Dieser Wert wird immer mit ***`NULL`*** initialisiert. Das OS ruft den ***`destructor`*** am Ende des Threads mit dem jeweiligen ***thread-spezifischen Wert*** auf, wenn dieser dann nicht `NULL` ist. Gibt 0 zurück wenn alles OK, sonst Fehlercode.

5.5.2. `int pthread_key_delete( pthread_key_t key)`

Entfernt den Key und die entsprechenden Values aus allen Threads. Der Key darf nach diesem Aufruf ***nicht mehr verwendet*** werden. Sollte erst aufgerufen werden, wenn alle dazugehörende Threads beendet sind. Das Programm muss dafür sorgen, ***sämtlichen Speicher freizugeben***, der eventuell zusätzlich alloziert worden war. Gibt 0 zurück wenn alles OK, sonst Fehlercode.

5.5.3. `pthread_setspecific` und `pthread_getspecific`

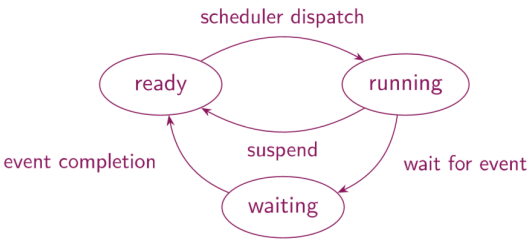
`int pthread_setspecific( pthread_key_t key, const void * value )`
`void * pthread_getspecific( pthread_key_t key )` ***schreibt*** bzw. ***liest*** den Wert, der mit dem Key in diesem Thread assoziiert ist. Typischerweise verwendet man den Wert als ***Pointer auf einen Speicherbereich***, bspw.:

```c
// Setup
typedef struct {
  int code;
  char *message;
} error_t;
pthread_key_t error; // globale Variabel

void set_up_error (void) { // wird am Anfang des Threads aufgerufen
  pthread_setspecific( error, malloc( sizeof( error_t ))) // speichert error_t-Pointer in error
}

// Lesen und Schreiben im Thread
void print_error (void) {
  error_t * e = pthread_getspecific (error); // error_t-Pointer aus TLS Key lesen
  printf("Error %d: %s\n", e→code, e→message); // error_t-Werte aus Pointer auslesen
}
int force_error (void) {
  error_t * e = pthread_getspecific (error); // error_t-Pointer aus TLS Key lesen
  // Durch Pointer Werte in error_t schreiben
  e→code = 98;
  e→message = "file not found";
  return -1;
}

// Main und Thread
void *thread_function (void *) {
  set_up_error();
  if (force_error () == -1) { print_error (); }
}
int main (int argc, char **argv) {
  pthread_key_create (&error, NULL); // Key erzeugen
  pthread_t tid;
  pthread_create (&tid, NULL, &thread_function, NULL); // Threads erzeugen
  pthread_join (tid, NULL);
}
```

6. SCHEDULING

Auf einem Prozessor läuft zu einem Zeitpunkt immer ***höchstens ein Thread***. Es gibt folgende ***Zustände:***

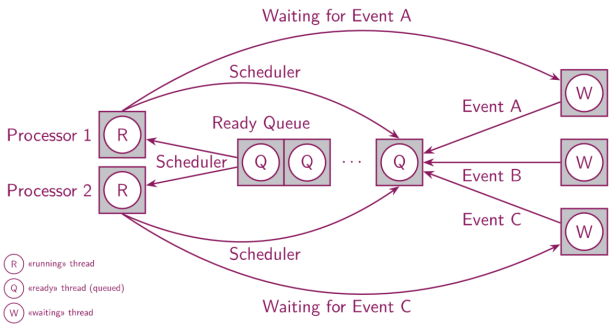
- ***Running:*** der Thread, der gerade läuft
- ***Ready:*** Threads die laufen können, es aber gerade nicht tun
- ***Waiting:*** Threads, die auf ein Ereignis warten *(können nicht direkt in den Status running wechseln, müssen neu gescheduled werden)*



Übergänge von einem Status zum anderen werden ***immer vom OS*** vorgenommen. Dieser Teil vom OS heisst ***Scheduler***.

6.1. GRUNDMODELL

Threads, die auf Ereignisse ***warten***, müssen das ***nicht*** in einer ***Endlosschleife*** tun *(Busy-Wait)*. Stattdessen registriert das OS sie auf das entsprechende Ereignis und setzt sie in den Zustand ***waiting***. Tritt das Ereignis auf, ändert das OS den Zustand auf ***ready***. Es laufen nur Threads auf dem Prozessor, die ***nicht warten***.



6.1.1. Ready-Queue

In der Ready-Queue *(kann auch ein Tree sein)* befinden sich alle Threads, die ***bereit sind zu laufen***. Neue Threads kommen typischerweise direkt in die Ready-Queue *(Einige OS stellen neue Threads auf waiting)*.

6.1.2. Powerdown-Modus

Wenn kein Thread ***laufbereit*** ist, schaltet das OS den Prozessor in ***Standby***. Der Prozessor wird dann durch den nächsten ***Interrupt*** wieder geweckt. So wird erheblich ***Energie gespart***. ***Busy-Waits*** sind verpönt, weil sie das Umschalten ins Standby verhindern.

6.1.3. Arten von Threads

- ***I/O-lastig:*** Kommuniziert sehr häufig mit I/O-Geräten und rechnet relativ wenig *(USB, Tastatur, Speicher)*. Priorisieren ***kurze Latenz***.
- ***Prozessor-lastig:*** Kommuniziert kaum oder gar nicht mit I/O-Geräten und rechnet fast ausschliesslich. Priorisieren ***mehr CPU-Zeit***.

Der Unterschied ist fliessend, aber gute Systeme ***trennen rechen-intensive von interaktiven Aktivitäten*** *(I/O-Thread, UI-Thread etc.)*.

6.1.4. Arten der Nebenläufigkeit

- ***Kooperativ:*** Jeder Thread entscheidet selbst, wann er den Prozessor abgibt *(Auch non-preemptive genannt)*
- ***Präemptiv:*** Der Scheduler entscheidet, wann einem Thread der Prozessor entzogen wird *(besseres System)*

Präemptives Multithreading: Der Thread läuft immer so lange weiter, bis er:

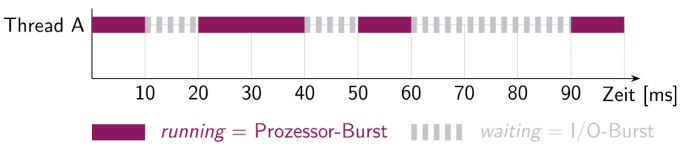
- Auf Ein-/Ausgabedaten, einen anderen Thread oder eine Ressource zu ***warten*** beginnt, d.h. blockiert
- ***freiwillig*** auf den Prozessor ***verzichtet*** *(yield)*
- ein ***System-Timer-Interrupt*** auftritt
- ein anderer Thread ***ready*** wird, der auf einen Event gewartet hat und ***bevorzugt werden soll***
- ein neuer Prozess ***erzeugt*** wird und ***bevorzugt werden soll***

6.1.5. Parallele, quasiparallele und nebenläufige Ausführung

- ***Parallel:*** Alle Threads laufen tatsächlich gleichzeitig, für n Threads werden n Prozessoren benötigt.
- ***Quasiparallel:*** n Threads werden auf $< n$ Prozessoren ***abwechselnd*** ausgeführt, sodass der Eindruck entsteht, dass sie parallel laufen würden.
- ***Nebenläufig:*** Überbegriff für parallel oder quasiparallel; aus Sicht des Programmierers sind thread-basierte Programme meist nebenläufig.

6.1.6. Bursts

- ***Prozessor-Burst:*** Intervall, in dem ein Thread den Prozessor in einem parallelen System ***voll belegt***, also vom Eintritt in ***running*** bis zum nächsten ***waiting***.
- ***I/O-Burst:*** Intervall, in dem ein Thread den Prozessor ***nicht*** benötigt, also vom Eintritt in ***waiting*** bis zum nächsten ***running***.



Jeder Thread kann als ***Abfolge*** von ***Prozessor-Bursts*** und ***I/O-Bursts*** betrachtet werden.

6.2. SCHEDULING-STRATEGIEN

Anforderungen an einen Scheduler können vielfältig sein. ***Geschlossene Systeme:*** Der Hersteller kennt alle Anwendungen und weiss, in welcher Beziehung sie zueinander stehen *(Router, TV Box)*. ***Offene Systeme:*** Der Hersteller des OS muss von typischen Anwendungen ausgehen und dahin gehend optimieren.

Anforderungen aus Sicht der Anwendung sind z.B. die Minimierung von:

- ***Durchlaufzeit (turnaround time):*** Zeit vom Starten des Threads bis zu seinem Ende
- ***Antwortzeit (respond time):*** Zeit vom Empfang eines Requests bis die Antwort zur Verfügung steht
- ***Wartezeit (waiting time):*** Zeit, die ein Thread in der Ready-Queue verbringt

Anforderungen aus Sicht des Systems sind z.B. die Maximierung von:

- ***Durchsatz (throughput):*** Anzahl Threads, die pro Intervall bearbeitet werden
- ***Prozessor-Verwendung (processor utilization):*** Prozentsatz der Verwendung des Prozessors gegenüber der Nichtverwendung

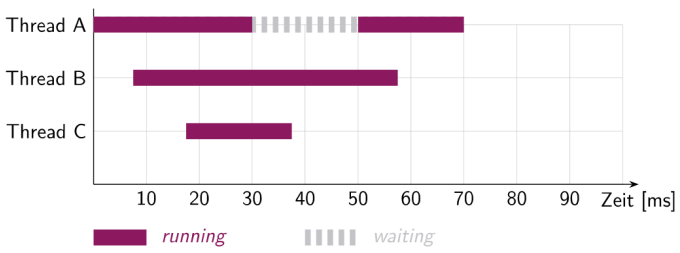
Grundsätzlich können Scheduler ***nicht*** auf ***alle Anforderungen gleichzeitig optimiert*** werden. Es gibt ***keinen optimalen Scheduler*** für ***alle*** Systeme. Die Wahl des Schedulers hängt vom Einsatzzweck ab.

6.2.1. Beispiel Utilization und Antwortzeit

Latenz ist die durchschnittliche Zeit zwischen Auftreten und vollständigem Verarbeiten eines Ereignisses. Im schlimmsten Fall tritt das Ereignis dann auf, wenn der Thread gerade vom Prozessor entfernt wurde. Um die Antwortzeit zu verringern, muss jeder Thread öfters ausgeführt werden, was jedoch zu mehr Thread-Wechsel und somit zu mehr Overhead führt. ***Die Utilization nimmt also ab, wenn die Antwortzeit verringert wird.***

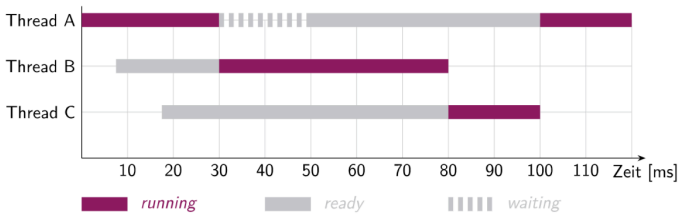
6.2.2. Idealfall: Parallele Ausführung *(n Threads auf n Prozessoren)*

Jeder Thread kann seinen Prozessor immer dann verwenden, wenn er ihn braucht. In der Praxis ***unrealistisch***, es gibt immer mehr Threads als Prozessoren. Dient als ***idealisierte Schranke*** für andere Scheduling-Strategien.



6.2.3. FCFS-Strategie *(First Come, First Served)*

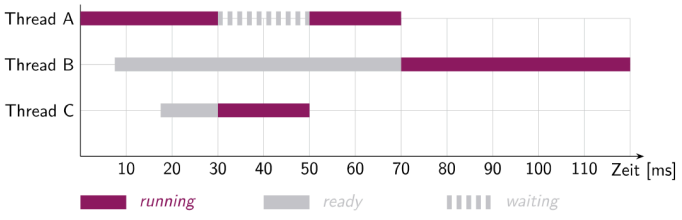
Threads werden in der Reihenfolge gescheduled, in der sie der Ready-Queue hinzugefügt werden. ***Nicht präemptiv:*** Threads geben den Prozessor nur ab, wenn sie auf «waiting» wechseln oder sich beenden. Die durchschnittliche Wartezeit hängt von der Reihenfolge des Eintreffens der Threads ab. Wird der längste Prozessor-Burst zuerst bearbeitet, warten die kürzeren Threads länger.



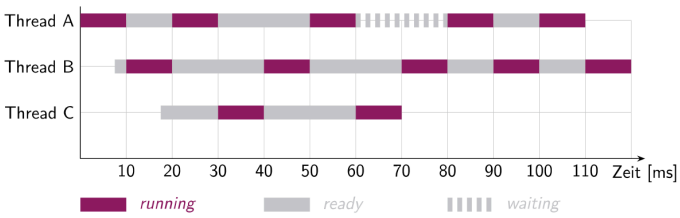
6.2.4. SJF-Strategie *(Shortest Job First)*

Scheduler wählt den Thread aus, der den ***kürzesten*** Prozessor-Burst hat. Bei gleicher Länge wird nach FCFS ausgewählt. Kann ***kooperativ*** oder ***präemptiv*** sein. Ergibt ***optimale Wartezeit:*** Der kürzeste Prozessor-Burst blockiert die anderen Threads minimal.

Kann nur ***korrekt implementiert*** werden, wenn die Länge der Bursts ***bekannt*** sind. Kann sonst nur mit einer ***Abschätzung historischer Daten annähernd*** implementiert werden.



6.2.5. Round-Robin-Scheduling



Der Scheduler definiert eine ***Zeitscheibe*** von etwa 10 bis 100ms. Das Grundprinzip folgt ***FCFS***, aber ein Thread kann nur solange laufen, bis seine ***Zeitscheibe erschöpft*** ist, dann wird der in der ***Ready-Queue hinten angehängt***. Benötigt er nicht den gesamten Time-Slice, beginnt die Zeitscheibe des nächsten Threads entsprechend früher. Die ***Wahl der Zeitscheibe beeinflusst das Verhalten*** massiv.

6.2.6. Prioritäten-basiertes Scheduling

Jeder Thread erhält ***eine Nummer***, seine ***Priorität***. Threads mit höherer Priorität werden vor Threads mit niedriger Priorität ausgewählt. Threads mit gleicher Priorität werden nach FCFS ausgewählt. SJF ist ein Spezialfall davon: kurzer nächster Prozessor-Burst entspricht hoher Priorität. Prioritäten je nach OS z.B. von 0 bis 7 oder von 0 bis 4096. Auf manchen OS ist 0 die höchste, auf anderen die niedrigste Priorität. Chaos ensues.

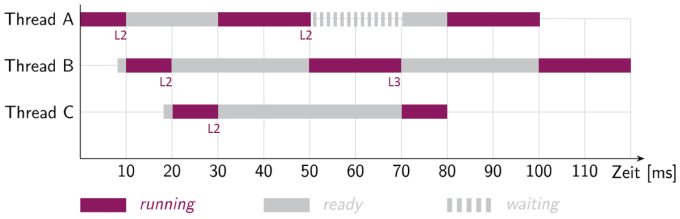
Starvation

Ein Thread mit ***niedriger Priorität*** kann ***unendlich lange nicht laufen***, weil immer Threads mit ***höherer Priorität*** laufen. Abhilfe z.B. mit ***Aging:*** in bestimmten Abständen wird die Priorität um 1 erhöht.

6.2.7. Multi-Level Scheduling

Threads werden nach bestimmten Kriterien in verschiedene ***Level*** aufgeteilt, z.B. Priorität, Prozesstyp, Hintergrund- oder Vordergrund. Für jedes Level gibt es eine ***eigene Ready-Queue***. Jedes Level kann nach einem eigenen Verfahren geschedulet werden, z.B. Queues gegeneinander priorisieren *(Threads in Queues mit* **höherer Priorität** *werden* **immer** *bevorzugt)* oder Time-Slices pro Queue *(80% für UI-Queue mit Round-Robin, 20% für Hintergrund-Queue mit FCFS)*.

6.2.8. Multi-Level Scheduling mit Feedback



Je Priorität eine Ready-Queue. Threads aus Ready-Queues mit ***höherer Priorität*** werden ***immer*** bevorzugt. Erschöpft ein Thread seine Zeitscheibe, wird seine Priorität um 1 verringert. Typischerweise werden die Zeitscheiben mit ***niedrigerer Priorität grösser*** und Threads mit ***kurzen Prozessor-Bursts bevorzugt***. Threads in tiefen Queues dürfen zum Ausgleich länger am Stück laufen.

6.3. PRIORITÄTEN IN POSIX

6.3.1. Der Nice-Wert

Jeder Prozess p hat einen Nice-Wert n_p *(in Linux jeder Thread)*. Dieser geht von -20 bis zu $+19$ und ist ein Hinweis ans System:

- Soll p ***bevorzugen***, wenn n_p ***kleiner*** ist *(p ist weniger nett)*
- Soll p ***weniger*** oft laufen lassen, wenn n_p ***grösser*** *(p ist netter)*

Nice-Wert beim Start erhöhen oder verringern: `nice [-n increment] utility [argument...]`

Startet `utility` *(u, mit den optionalen Argumenten)* mit möglicherweise ***anderem Nice-Value*** als der aufrufende Prozess p. Wenn kein `increment` angegeben: $n_u \geq n_p$. Wenn `increment` (i) angegeben: $n_u = n_p + i$

Nice-Wert im Prozess erhöhen oder verringern: `int nice (int i)`

Addiert `i` zum Nice-Wert des aufrufenden Prozesses p. Gibt n_p zurück oder -1, wenn Fehler. Da -1 aber auch ein gültiger Nice-Wert ist, muss man den Fehler wie folgt abfragen:

```
errno = 0; // reset errno
if (nice(i) == -1 && errno != 0) { /* Error */ } else { /* -1 is nice value */ }
```

Nice-Wert im Prozess abfragen oder setzen: `int getpriority` / `int setpriority`

`int getpriority (int which, id_t who)` gibt den Nice-Wert von p zurück

`int setpriority (int which, id_t who, int prio)` setzt den Nice-Wert von p auf n. Gibt 0 zurück wenn OK, sonst −1 und Fehlercode in `errno`.

Spezifiziert Priorität für einzelnen Prozess, Prozessgruppe oder alle Prozesse eines Users.
- ***which:*** `PRIO_PROCESS`, `PRIO_PGRP` oder `PRIO_USER`
- ***who:*** ID des Prozesses, der Gruppe oder des Users

Priorität bei Threads setzen: `...schedparam`

```
int pthread_getschedparam(pthread_t thread, int * policy, struct sched_param * param)
int pthread_setschedparam(pthread_t thread, int policy, const struct sched_param * param)
int pthread_attr_getschedparam(const pthread_attr_t * attr, struct sched_param * param)
int pthread_attr_setschedparam(pthread_attr_t * attr, const struct sched_param * param)
```

Die Priorität kann während der Thread läuft mit den regulären Funktionen oder vor dem Threadstart mit den `attr`-Funktionen gesetzt werden. Attribute eines Threads enthalten ein ***`struct sched_param`***. Dieser kann vom Thread oder seinen Attributen ***abgefragt*** werden. Enthält ein Member `sched_priority`, das die ***Priorität*** bestimmt.

Priorität bei Thread-Erzeugung setzen

```
pthread_attr_t a;
pthread_attr_init (&a); // initialize attributes

struct sched_param p;
pthread_attr_getschedparam ( &a, &p ); // read parameter
// set p.sched_priority here... (code cut for brevity)
pthread_attr_setschedparam ( &a, &p ); // write modified parameters
pthread_create ( &id, &a, thread_function, argument );
pthread_attr_destroy ( &a ); // destroy attributes
```

7. MUTEXE UND SEMAPHORE

Jeder Thread hat seinen ***eigenen*** Instruction-Pointer und Stack-Pointer. Die IPs aller Threads werden ***unabhängig*** voneinander bewegt. Bei parallelen Threads völlig unabhängig und ***nicht synchron***, selbst bei identischem Code. Bei nebenläufigen Threads auf dem selben Prozessor immer in Bursts bis zum nächsten Thread-Wechsel.

7.1. SYNCHRONISATIONS-MECHANISMEN GRUNDLAGEN

Ein Thread erzeugt Items: ***der Producer***. Ein anderer Thread verarbeitet diese: ***der Consumer***. Beide Threads arbeiten ***unterschiedlich schnell***. Items werden über einen ***begrenzt grossen Ring-Puffer*** übermittelt. Falls der Puffer voll ist, muss der Producer warten, bevor er wieder etwas auf den Puffer legen kann. Gleichermassen muss der Consumer warten, falls der Puffer leer ist.

Da C für ***keine noch so kleine Operation garantiert***, dass sie in eine ***einzige Instruktion*** übersetzt wird *(non-atomic)*, wird dieses Problem eine ***Race-Condition*** auslösen.

7.1.1. Race-Condition

Wenn Ergebnisse von der ***Ausführungsreihenfolge*** einzelner Instruktionen abhängen, spricht man von einer ***Race Condition***. Register werden beim Kontext-Wechsel gesichert, der Counter aber nicht. Greifen ***nebenläufige Threads*** schreibend und lesend auf den ***gleichen Hauptspeicherbereich*** zu, gibt es ***keine Garantien***, was passieren wird. Wenn die Änderung nicht schnell genug erfolgt, bekommt sie der andere Thread nicht mit, während er selbst Änderungen vornimmt. Ein Thread muss andere Threads vom Zugriff ausschliessen können - ***Threads müssen synchronisiert werden***.

7.1.2. Critical Sections

Jeder kooperierende Thread hat einen Code-Bereich, in dem er Daten mit anderen Threads teilt, die ***Critical Section***. Es wird ein ***Protokoll*** benötigt, anhand dessen Threads den Zugang zu ihren Critical Sections ***synchronisieren*** können.

7.1.3. Atomare Instruktionen

Eine atomare Instruktion kann vom Prozessor ***unterbrechungsfrei*** ausgeführt werden.

*(**Achtung:** Selbst einzelne Assembly-Instruktionen können unter Umständen nicht atomar durchgeführt werden, z.B. non-aligned Memory Access)*

7.1.4. Anforderungen an Synchronisations-Mechanismen

- ***Gegenseitiger Ausschluss:*** Wenn ein Thread in seiner Critical Section ist, dürfen alle anderen Threads ihre Critical Section nicht betreten. *(mutual exclusion, mutex)*
- ***Fortschritt:*** Wenn kein Thread in seiner Critical Section ist und irgendein Thread in seine Critical Section möchte, muss in endlicher Zeit eine Entscheidung getroffen werden, wer als nächstes in die Critical Section darf.
- ***Begrenztes Warten:*** Es gibt eine feste Zahl n, sodass gilt: Wenn ein Thread seine Critical Section betreten will, wird er nur n-mal übergangen.

7.1.5. Implementierung von Synchronisations-Mechanismen

Moderne Computer-Architekturen geben ***kaum Garantien*** bezüglich der Ausführung von Instruktionen. Instruktionen müssen ***nicht atomar*** sein, Sequenzen können äquivalent ***umgeordnet*** werden. Synchronisations-Mechanismen können auf modernen Computern ***nur mit Hardwareunterstützung*** implementiert werden.

Konzept: Abschaltung von Interrupts

Alle Interrupts werden abgeschaltet, wenn eine ***Critical Section betreten*** werden soll. Auf Systemen mit ***einem Prozessor effektiv***, weil es zu keinem Kontext-Wechsel kommen kann. Für Systeme mit ***mehreren Prozessoren*** jedoch ***nicht praktikabel***, da Interrupts für alle Threads ausgeschaltet werden müssten. ***Generell gefährlich:*** Solange die Interrupts ausgeschaltet sind, kann das OS den Thread nicht unterbrechen.

Verwendung spezieller Instruktionen

Moderne Prozessoren stellen eine von zwei ***atomaren*** Instruktionen zur Verfügung, mit denen ***Locks*** implementiert werden können:

- ***Test-And-Set:*** Setzt einen `int` auf 1 und returnt den vorherigen Wert
- ***Compare-And-Swap:*** Das gleiche, aber in Fancy: überschreibt einen `int` mit einem spezifizierten Wert, wenn dieser dem erwarteten Wert entspricht.

Test-And-Set: Liest den Wert von einer Adresse (0 oder 1) und setzt ihn dann auf 1.

```
tas (int * target) { int value = *target; *target = 1; return value; }

int lock = 0;
// T1: sets lock = 1 & reads 0, T2 sets lock = 1, but reads 1
while (tas (&lock) == 1) { /* busy loop */ }
/* critical section */
lock = 0;
```

Compare-And-Swap: ***Liest*** einen Wert aus dem Hauptspeicher und ***überschreibt*** ihn im Hauptspeicher, falls er einem ***erwarteten Wert*** entspricht.

```
cas (int *a, int expected, int new_a) {
  int value = *a;
  if (value == expected) { *a = new_a; }
  return value;
}

while (cas (&lock, 0, 1) == 1) { /* busy loop */ }
/* critical section */
lock = 0;
```

Kommen zwei Threads T_1 und T_2 genau gleichzeitig an die `while`-Schleife, garantiert die Hardware, dass ***nur*** T_1 ***`test_and_set`*** bzw. ***`compare_and_swap`*** ausführt. T_1 setzt `lock` auf 1, liest aber 0 und ***verlässt*** die Schleife sofort. T_2 sieht `lock` auf jeden Fall als 1 und ***bleibt*** in der Schleife.

7.2. SEMAPHORE

Ein Semaphore enthält einen ***Zähler*** $z \geq 0$. Auf den Semaphor wird nur über spezielle Funktionen zugegriffen:

- ***Post (v):*** Erhöht z um 1
- ***Wait (p):*** Wenn $z > 0$, verringert z um 1 und setzt Ausführung fort. Wenn $z = 0$, versetzt den Thread in waiting, bis ein anderer Thread z erhöht.

7.2.1. Producer-Consumer-Problem mit Semaphoren

Der ***Producer*** wartet darauf, dass mindestens ein Element ***frei*** ist. Der ***Consumer*** wartet darauf, dass mindestens ein Element ***gefüllt*** ist. Dafür verwenden wir ***zwei Semaphore***. Die Consumer und Producer geben sich diese gegenseitig frei.

```
semaphore free = n;
semaphore used = 0;
```

```
// Producer
int w = 0; // Index auf zuletzt geschr. Elem.
while (1) {
  // Warte, falls Customer zu langsam
  WAIT (free); // Hat es Platz in Queue?
  produce_item (&buffer[w], ...);
  POST (used); // 1 Element mehr in Queue
  w = (w+1) % BUFFER_SIZE;
}
```

```
// Consumer
int r = 0; // Index auf zu lesendes Elem.
while (1) {
  // Warte, falls Producer zu langsam
  WAIT (used); // Hat es Elemente in Queue?
  consume (&buffer[r]);
  POST (free); // 1 Element weniger in Queue
  r = (r+1) % BUFFER_SIZE;
}
```

7.2.2. `int sem_init (sem_t *sem, int pshared, unsigned int value);`

Initialisiert den Semaphor `sem`, sodass er `value` Marken enthält *(max. Grösse der Queue)*. Ist `pshared = 0`, kann `sem` nur innerhalb eines Prozesses verwendet werden, ansonsten über mehrere.

Anwendung als globale Variable

Typischerweise legt man im Programm eine ***globale Variable*** `sem` vom Typ `sem_t` an. ***Bevor*** der erste Thread gestartet wird, der `sem` verwenden soll, wird ***`sem_init`*** aufgerufen, z.B. im `main()`.

```
sem_t sem;
int main ( int argc, char ** argv ) { sem_init (&sem, 0, 4); }
```

Anwendung als Parameter für den Thread

Alternativ definiert man im Struct, das dem Thread übergeben wird, einen Member `sem` vom Typ `sem_t *`.
Der Speicher für den Semaphor wird dann entweder auf dem Stack oder auf dem Heap alloziert.

```
struct T { sem_t *sem; ... };
int main ( int argc, char ** argv ) {
  sem_t sem;
  sem_init (&sem, 0, 4);
  struct T t = { &sem, ... };
}
```

7.2.3. `sem_wait` und `sem_post`

`int sem_wait (sem_t *sem); int sem_post (sem_t *sem);` implementieren ***Post*** und ***Wait***. Geben 0 zurück, wenn Aufruf OK, sonst −1 und Fehlercode in `errno`. Im ***Fehlerfall*** wird der Semaphor ***nicht verändert***.

7.2.4. `sem_trywait` und `sem_timedwait`

```
int sem_trywait (sem_t *sem);
int sem_timedwait (sem_t *sem, const struct timespec *abs_timeout);
```

Sind wie `sem_wait`, aber ***brechen ab***, falls Dekrement ***nicht*** durchgeführt werden kann. `sem_trywait` bricht sofort ab, `sem_timedwait` nach der angegebenen Zeitdauer. Es gibt kein `sem_trypost`.

7.2.5. `int sem_destroy (sem_t *sem);`

Entfernt möglichen zusätzlichen ***Speicher***, den das OS mit `sem` ***assoziiert*** hat.

7.3. MUTEXE

Ein Mutex hat einen ***binären Zustand*** z, der nur durch zwei Funktionen verändert werden kann:
- ***Acquire:*** Wenn $z = 0$, setze z auf 1 und fahre fort. Wenn $z = 1$, blockiere den Thread , bis $z = 0$
- ***Release:*** Setzt $z = 0$

Kann durch einen Semaphor mit ***Beschränkung*** von z auf 1 realisiert werden. Acquire und Release heissen auch ***Lock*** bzw. ***Unlock***.

Ein Mutex ist die ***einfachste Form*** der Synchronisierung. Acquire und Release müssen immer paarweise durchgeführt werden. `ACQUIRE(mutex); ++counter; RELEASE(mutex);`

7.3.1. POSIX Thread Mutex API

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

Initialisiert die opake Daten-Struktur ***`pthread_mutex_t`***. Attribute sind ***optional***, Verwendung analog zu `pthread`-Attributen mit `pthread_mutexattr_init, ..._destroy`.

Attribute:
- `protocol`: z.B. `PTHREAD_PRIO_INHERIT`: Mutex verwendet Priority-Inheritance,
- `pshared`: Mutex kann von anderen Prozessen verwendet werden,
- `type`: Mutex kann beliebig oft vom selben Thread aquiriert werden,
- `prioceiling`: Minimale Priorität der Threads, die den Mutex halten.

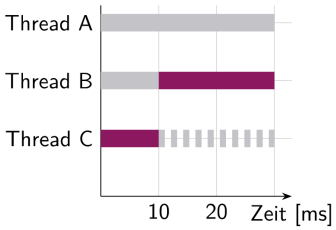
```
int pthread_mutex_lock (pthread_mutex_t *mutex);     // acquire (blocking)
int pthread_mutex_trylock (pthread_mutex_t *mutex); // attempt to acquire (non-blocking)
int pthread_mutex_unlock (pthread_mutex_t *mutex);  // release
int pthread_mutex_destroy (pthread_mutex_t *mutex)  // cleanup
```

```
// Beispiel Initialisierung
pthread_mutex_t mutex; // global variable
int main() {
  // 0 = default Attribute
  pthread_mutex_init (&mutex, 0);
  // run threads and wait for them to finish
  pthread_mutex_destroy (&mutex);
}
```

```
// Beispiel Verwendung in Threads
void * thread_function (void * args) {
  while (running) {
    ...
    // Enter critical section
    pthread_mutex_lock (&mutex);
    // Perform atomic action, e.g. ++counter
    // Leave critical section
    pthread_mutex_unlock (&mutex);
    ...
  }
}
```

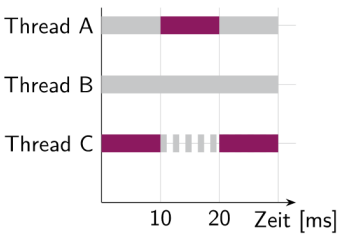
7.3.2. Priority Inversion und Priority Inheritance

- Thread A hat ***niedrige Priorität*** und hält einen Mutex M
- Thread B hat ***mittlere Priorität***
- Thread C hat ***hohe Priorität*** und läuft gerade. Nach 10ms benötigt Thread C den Mutex M.



Ein ***hoch-priorisierter*** Thread wartet auf eine Ressource, die von einem ***niedriger priorisierten*** Thread ***gehalten*** wird. Ein Thread mit Priorität zwischen diesen beiden Threads erhält den Prozessor. Die effektiven Prioritäten des hoch-priorisierten und des mittel-priorisierten Threads sind ***invertiert*** gegenüber den zugewiesenen Prioritäten. ***Gemeinsam verwendete Ressourcen werden bei Priority Inversion im schlimmsten Fall mit der niedrigsten Priorität aller beteiligten Threads gehalten.***

Um dieses Problem zu lösen, wird bei Priority Inheritance die ***Priorität von A temporär auf die Priorität von C gesetzt***, damit der Mutex schnell wieder freigegeben wird. A läuft, bis er den Mutex M freigibt, danach erhält A wieder die vorherige Priorität und C läuft weiter.



8. SIGNALE, PIPES UND SOCKETS

8.1. SIGNALE

Signale ermöglichen es, einen Prozess ***von aussen*** zu unterbrechen. Wird ein Signal an einen Prozess geschickt, verhält sich das OS, als ob ein ***Interrupt*** geschickt wurde *(quasi Software-Interrupts)*:
- ***Unterbrechen*** des gerade laufenden Prozesses/Threads
- Auswahl und Ausführen der ***Signal-Handler-Funktionen***
- ***Fortsetzen*** des Prozesses

8.1.1. Quelle von Signalen

- ***Hardware / OS*** *(ungültige Instruktion, Zugriff auf ungültige Speicheradresse, Division durch 0)*
- ***Andere Prozesse*** *(Abbruch des Benutzerprogramms über Ctrl-C, Aufruf des Kommandos `kill`)*

8.1.2. Signale behandeln

Jeder Prozess hat ***pro Signal einen Handler***. Bei Prozessbeginn gibt es für jedes Signal einen von ***drei Default-Handlern***:
- ***Ignore-Handler:*** ignoriert das Signal
- ***Terminate-Handler:*** beendet das Programm
- ***Abnormal-Terminate-Handler:*** beendet das Programm und erzeugt Core Dump *(Snapshot des Programms)*

Fast alle Signal-Handler können ***überschrieben*** werden, ***ausser `SIGKILL` und `SIGSTOP`.***

8.1.3. Wichtige Signale

Programmfehler: Diese Signale werden vom OS erzeugt und nutzen standardmässig den Abnormal-Termination-Handler:
- ***`SIGFPE`:*** Fehler in arithmetischen Operation *(floating point error)*
- ***`SIGILL`:*** Ungültige Instruktion *(illegal instruction)*
- ***`SIGSEGV`:*** Ungültiger Speicherzugriff *(segmentation violation)*
- ***`SIGSYS`:*** Ungültiger Systemaufruf

Prozesse abbrechen:
- ***`SIGTERM`:*** Normale Anfrage an den Prozess, sich zu beenden *(terminate)*
- ***`SIGINT`:*** Nachdrücklichere Aufforderung an den Prozess, sich zu beenden *(interrupt, Ctrl-C)*
- ***`SIGQUIT`:*** Wie `SIGINT`, aber anormale Terminierung *(Ctrl-\)*
- ***`SIGABRT`:*** Wie `SIGQUIT`, aber vom Prozess an sich selber *(abort, bei Programmfehler z.B.)*
- ***`SIGKILL`:*** Prozess wird «abgewürgt», kann vom Prozess nicht verhindert werden

Stop and Continue:
- ***`SIGTSTP`:*** Versetzt den Prozess in den Zustand ***stopped***, ähnlich wie ***waiting*** *(terminal stop, Ctrl-Z)*
- ***`SIGSTOP`:*** Wie `SIGTSTP`, aber kann nicht ignoriert oder abgefangen werden
- ***`SIGCONT`:*** Setzt den Prozess fort *(Auf shell mit `fg` / `bg` = foreground / background)*

8.1.4. Signale von der Shell senden

Das Kommando `kill` sendet ein Signal an einen oder mehrere Prozesse *(ohne Angabe eines Signals wird `SIGTERM` gesendet)*
- ***`kill 1234 5678`*** sendet `SIGTERM` an Prozesse 1234 und 5678
- ***`kill -KILL 1234`*** sendet `SIGKILL` an Prozess 1234
- ***`kill -l`*** listet alle möglichen Signale auf

8.1.5. Signal-Handler im Programm ändern: `sigaction`

`int sigaction (int signal, struct sigaction *new, struct sigaction *old)`

`signal` ist die ***Nummer des Signals*** *(`SIGKILL` oder `SIGSTOP` nicht erlaubt)*. ***Definiert*** Signal-Handler für `signal`, wenn `new` $\neq 0$. ***Gibt*** den ***bestehenden*** Signal-Handler für `signal` ***zurück***, wenn `old` $\neq 0$

`struct sigaction { void (*sa_handler)(int); sigset_t sa_mask; int sa_flags;}`

`sa_handler` ist die ***Adresse der Funktion***, die aufgerufen wird, wenn das Signal auftritt. *`sa_mask`* gibt an, welche Signale während Handler-Ausführung blockiert werden, das eigene Signal wird immer blockiert. *`sa_flags`* ermöglicht verschiedene zusätzliche Eigenschaften.

8.1.6. Signale spezifizieren

`sigset_t` wird nur mit folgenden Funktionen verwendet:
- *`int sigemptyset (sigset_t *set):`* Kein Signal ausgewählt
- *`int sigfillset (sigset_t *set):`* Alle Signale ausgewählt
- *`int sigaddset (sigset_t *set, int signal):`* Fügt `signal` der Menge hinzu
- *`int sigdelset (sigset_t *set, int signal):`* Entfernt `signal` aus der Menge
- *`int sigismember (const sigset_t *set, int signal):`* Gibt 1 zurück, wenn `signal` in der Menge enthalten ist.

8.2. PIPES

Eine geöffnete Datei entspricht einem ***Eintrag in der File-Descriptor-Tabelle (FDT)*** im Prozess. Zugriff über ***File-API*** *(`open`, `close`, `read`, `write`, …)*. Prozess weiss ***nicht***, was eine Datei ist und wie das OS damit umgeht. Das OS speichert ***je Eintrag der Prozess-FDT*** einen ***Verweis auf die globale FDT***. Wenn ein Prozess mit `fork` neu erzeugt wird, wird auch die ***FDT*** des Parents in das Child ***kopiert***.

8.2.1. `int dup (int source_fd); int dup2 (int source_fd, int destination_fd);`

Duplizieren den File-Descriptor `source_fd` und geben den neuen File-Descriptor zurück. *`dup`* alloziert einen ***neuen FD***, *`dup2`* ***überschreibt*** `destination_fd`.

8.2.2. Umleiten des Ausgabestreams

```c
int fd = open ("log.txt", ...);
int id = fork ();
if (id == 0) { // child process
  dup2 (fd, 1); // duplicate fd for log.txt as standard output
                // e.g. load new image with exec*, fd's remain
} else { // parent process
  close (fd);
}
```

8.2.3. Abstrakte Dateien

Die ***Konsole*** ist ***keine Datei*** auf einem Datenträger, aber trotzdem ***Standard-Ausgabestream***. Die Abstraktion ***«Datei»*** sagt ***nichts über die Infrastruktur*** aus. Eine «Datei» muss nur `open`, `close` etc. unterstützen. ***«In POSIX, everything is a file».***

Eine Pipe ist eine «Datei» im Hauptspeicher, die über zwei File-Deskriptoren verwendet wird:
- ***read end*** zum Lesen aus der Pipe
- ***write end*** zum Schreiben in die Pipe

Daten, die in ***write end*** geschrieben werden, können aus ***read end*** genau ***einmal*** und als ***FIFO*** gelesen werden. Pipes unterstützen ***kein*** `lseek`, erlauben aber ***Kommunikation über Prozess-Grenzen hinweg***.

8.2.4. `int pipe (int fd[2]) // equivalent to int pipe(int *fd)`

Erzeugt eine Pipe und zwei FD's *(0 = read, 1 = write)*, die in `fd` abgelegt werden. Pipe lebt solange, wie eines der beiden Enden in einem Prozess geöffnet ist. Rückgabewert 0, wenn OK, sonst −1 und Fehlercode in `errno`. Unter Linux ***Default-Pipe-Grösse: 16 Pages*** *(mit 4 KB-Pages = 64 KB)*. Kann mit `close`, `read` und `write` ***wie Datei*** verwendet werden.

8.2.5. Daten von Parent zu Child

```
int fd [2];
pipe (fd);
int id = fork();

if (id == 0) { // Child process
  close (fd [1]); // don't use write end
  char buffer [BSIZE];
  int n = read (fd[0], buffer, BSIZE);
} else { // Parent process
  close (fd[0]); // don't use read end
  char * text = "Die Zemmefassig isch viel z lang";
  write (fd [1], text, strlen(text) + 1);
}
```

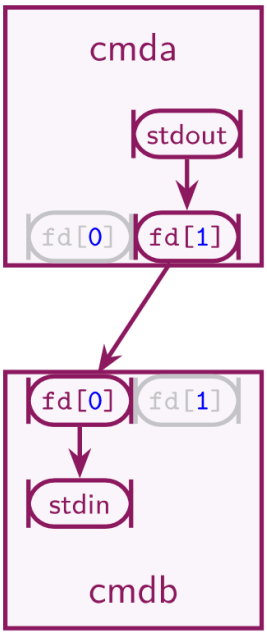
8.2.6. Lesen aus einer Pipe

Aus einer Pipe kann mit `read` gelesen werden, als ob sie eine ***Datei*** wäre. Sind ***keine Daten*** in der Pipe, ***blockiert*** `read`, bis Daten hineingeschrieben werden. Gibt es zusätzlich ***kein geöffnetes Write-End*** mehr, gibt `read` 0 zurück ***(EOF)***. ***Lesender Prozess*** muss deshalb sein ***Write-End schliessen***, damit schreibender Prozess über das Schliessen seines Write-Ends das ***Ende der Kommunikation mitteilen*** kann.

8.2.7. Standard-Ausgabe mit -Eingabe verknüpfen

Beispiel-Befehl in Shell: `cmda | cmdb`

```
int fd [2];
pipe (fd);
int id1 = fork();

if (id1 == 0){ // child (cmda)
  close (fd [0]); // don't use read end
  dup2 (fd [1], 1); // define pipe write end as stdout
  exec ("cmda", ...);
} else { // parent (shell)
  int id2 = fork();
  if (id2 == 0) { // child (cmdb)
    close (fd[1]); // don't use write end
    dup2 (fd [0], 0);
    exec ("cmdb", ...);
  } else { // parent (shell)
    wait (0);
    wait (0);
  }
}
```



Pipes sind ***unidirektional:*** es ist nicht spezifiziert, was beim Schreiben ins ***read end*** oder Lesen vom ***write end*** passiert. Sind ***alle read ends*** geschlossen, erhält Prozess mit ***write end*** ein ***`SIGPIPE`*** *(Broken Pipe)*.

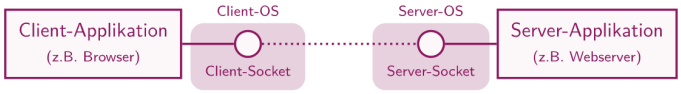
Wann der Transport erfolgt, ist implementierungsabhängig. Mehrere `writes` können bspw. zusammengefasst werden. Ein Rückgabewert $< n$ von `read(...,n)` bedeutet ***nicht***, dass später nicht noch mehr Daten kommen können. Lesen mehrere Prozesse die selbe Pipe, ist unklar, welcher die Daten erhält.

8.2.8. `int mkfifo (const char *path, mode_t mode);`

Erzeugt eine Pipe ***mit Namen und Pfad*** im Dateisystem. Hat via `mode` ***permission bits*** wie eine normale Datei. Lebt ***unabhängig vom erzeugenden Prozess***, je nach System auch über Reboots hinweg. Muss explizit mit ***`unlink` gelöscht*** werden.

8.3. SOCKETS

Berkeley Sockets sind eine Abstraktion über Kommunikationsmechanismen. Beispiele: UDP, TCP über IP sowie Unix-Domain-Sockets. Ein Socket ***repräsentiert einen Endpunkt auf einer Maschine***. Kommunikation findet im Regelfall zwischen zwei Sockets statt. Sockets benötigen für Kommunikation einen Namen: *(Beispiel IP: IP-Adresse, Portnummer)*



8.3.1. `int socket(int domain, int type, int protocol);`

Erzeugt einen neuen Socket als «Datei». Socket sind nach Erzeugung zunächst ***unbenannt***. Alle Operationen blockieren per default. Gibt FD zurück (≥ 0) bzw. −1 bei Fehler mit Fehlercode in `errno`.

- ***domain:*** Adress-Domäne *(`AF_UNIX`: Innerhalb einer Maschine, `AF_INET`: Internet-Kommunikation über IPv4, Adressen sind IP-Adressen plus Ports, `AF_INET6`: Internet-Kommunikation über IPv6)*
- ***type:*** Art der Kommunikation *(`SOCK_DGRAM`: Datagram-Socket wie UDP, `SOCK_STREAM`: Byte-Stream Socket wie TCP)*
- ***protocol:*** System-spezifisch, 0 = Default-Protocol

Ein Client verwendet einen Socket in folgender Reihenfolge:

1. ***connect:*** Verbindung unter Angabe einer Adresse aufbauen *(Socket erhält damit Namen)*
2. ***send / write:*** Senden von Daten, $0 - \infty$ mal *(z.B. eine Anfrage)*
3. ***recv / read:*** Empfangen von Daten, $0 - \infty$ mal *(z.B. eine Antwort)*
4. ***close:*** Schliessen der Verbindung

Ein Server verwendet einen Socket in folgender Reihenfolge:

1. ***bind:*** Festlegen einer nach aussen sichtbaren Adresse *(z.B. zuweisen von IP/Port)*
2. ***listen:*** Bereitstellen einer Queue zum Sammeln von Verbindungsanfragen von Clients
3. ***accept:*** Erzeugen einer Verbindung auf Anfrage von Client *(erzeugt neuen Socket)*
4. ***recv / read:*** Empfangen von Daten, $0 - \infty$ mal *(z.B. eine Anfrage)*
5. ***send / write:*** Senden von Daten, $0 - \infty$ mal *(z.B. eine Antwort)*
6. ***close:*** Schliessen der Verbindung

8.3.2. Beispiel Angabe IP-Adresse

```
struct sockaddr_in ip_addr;
ip_addr.sin_port = htons (443); // default HTTPS port
inet_pton (AF_INET, "192.168.0.1", &ip_addr.sin_addr.s_addr);
// IP address in memory: 0xC0 0xA8 0x00 0x01
// Port in memory: 0x01 0xBB
```

`htons` konvertiert 16 Bit von Host-Byte-order *(LE)* zu Network-Byte-Order *(BE)*, ***`htonl`*** 32 Bit. ***`ntohs`*** und ***`ntohl`*** sind Gegenstücke. ***`inet_pton`*** konvertiert protokoll-spezifische Adresse von String zu Network-BO. ***`inet_ntop`*** ist das Gegenstück *(network-to-presentation)*.

8.3.3. `int bind (int socket, const struct sockaddr *local_address, socklen_t addr_len);`

Bindet den Socket an die ***angegebene***, unbenutzte ***lokale Adresse***, wenn noch nicht gebunden. ***Blockiert***, bis der Vorgang abgeschlossen ist. Gibt 0 zurück, wenn alles OK, sonst −1 und Fehlercode in `errno`.

8.3.4. `int connect (int socket, const struct sockaddr *remote_addr, socklen_t addr_len);`

Aufbau einer Verbindung. ***Bindet*** den Socket an eine ***neue***, unbenutzte ***lokale Adresse***, wenn noch nicht gebunden. ***Blockiert***, bis Verbindung steht oder ein Timeout eintritt. Gibt 0 zurück, wenn alles OK, sonst −1 und Fehlercode in `errno`.

8.3.5. `int listen (int socket, int backlog);`

Markiert den Socket als ***«bereit zum Empfang von Verbindungen»***. Erzeugt eine ***Warteschlange***, die so viele Verbindungsanfragen aufnehmen kann, wie `backlog` angibt. Gibt 0 zurück, wenn alles OK, sonst −1 und Fehlercode in `errno`.

8.3.6. `int accept (int socket, struct sockaddr *remote_address, socklen_t address_len);`

Wartet bis eine ***Verbindungsanfrage*** in der Warteschlange ***eintrifft***. Erzeugt einen neuen Socket und bindet ihn an eine neue lokale Adresse. Die Adresse des Clients wird in ***`remote_address`*** geschrieben. Der neue Socket kann keine weiteren Verbindungen annehmen, der bestehende hingegen schon. Gibt FD des neuen Sockets zurück, wenn alles OK, sonst −1 und Fehlercode in `errno`.

8.3.7. Typisches Muster für Server

```
int server_fd = socket ( ... );
bind (server_fd, ...);
listen (server_fd, ...);
while (running) {
  int client_fd = accept (server_fd, 0, 0);
  delegate_to_worker_thread (client_fd); // will call close(client_fd)
}
```

8.3.8. `send` und `recv`

```
ssize_t send (int socket, const void *buffer, size_t length, int flags);
ssize_t recv (int socket, void *buffer, size_t length, int flags);
```

Senden und Empfangen von Daten. Erweitern `read()` bzw. `write()` um Socket-Funktionalitäten durch den `flags`-Parameter. Puffern der Daten ist Aufgabe des Netzwerkstacks.

```
send (fd, buf, len, 0) == write (fd, buf, len);
recv (fd, buf, len, 0) == read (fd, buf, len)
```

8.3.9. `int close (int socket);`

Schliesst den Socket für den ***aufrufenden*** Prozess. Hat ein anderer Prozess den Socket noch geöffnet, bleibt die Verbindung bestehen. Die Gegenseite wird ***nicht*** benachrichtigt *(Schlechte Idee, besser `shutdown`)*.

8.3.10. `int shutdown (int socket, int mode);`

Schliesst den Socket für ***alle*** Prozesse und baut die entsprechende Verbindung ab.
- ***`mode = SHUT_RD:`*** Keine Lese-Zugriffe mehr
- ***`mode = SHUT_WR:`*** Keine Schreib-Zugriffe mehr
- ***`mode = SHUT_RDWR:`*** Keine Lese- oder Schreib-Zugriffe mehr

---

9. MESSAGE PASSING UND SHARED MEMORY

Prozesse sind voneinander ***isoliert***, müssen jedoch trotzdem miteinander ***interagieren***.
Beispiel Chrome Browser: Jede Seite wird durch einen eigenen Prozess gerendert, Seiten können sich ***nicht*** gegenseitig ***beeinflussen***. Ein weiterer Prozess stellt ***GUI*** sowie Zugriffe auf Filesystem und Netzwerk zur Verfügung. Dies ***reduziert*** Auswirkungen von ***Security-Exploits***.

9.1. MESSAGE-PASSING / MESSAGE QUEUEING

Message-Passing ist ein Mechanismus mit zwei Operationen:
- ***Send:*** Kopiert die Nachricht ***aus*** dem Prozess: `send (message)`
- ***Receive:*** Kopiert die Nachricht ***in*** den Prozess: `receive (message)`

Dabei können Implementierungen nach verschiedenen Kriterien unterschieden werden:
- ***Feste*** oder ***Variable*** Nachrichtengrösse
- ***Direkte*** oder ***indirekte*** Kommunikation
- ***Synchrone*** oder ***asynchrone*** Kommunikation
- ***Pufferung***
- Mit oder ohne ***Prioritäten*** für Nachrichten

9.1.1. Feste oder variable Nachrichtengrösse

Message-Passing mit ***fester Nachrichtengrösse*** ist ***umständlicher zu verwenden***. Benutzer muss bei ***Überschreiten*** der Nachrichtengrösse selbst ***Vorsorge treffen***, um die Nachricht in kleinere Teilnachrichten aufzutrennen. Bei ***Unterschreiten*** wird Speicher ***verschwendet***.
Message-Passing mit ***variabler Nachrichtengrösse*** ist ***aufwändiger zu implementieren***.

9.1.2. Direkte Kommunikation - Senden

Bei direkter Kommunikation werden Nachrichten von einem Prozess P_1 an einen Prozess P_2 adressiert. P_1 muss den Empfänger einer Nachricht kennen. Kommunikation ***nur zwischen genau zwei Prozessen***: `send(P2, message)`.

9.1.3. Direkte Kommunikation - Empfangen

- ***Symmetrische direkte Kommunikation*** `receive(P1, message)`: P_2 muss den Sender seiner Nachricht ***kennen***
- ***Asymmetrische direkte Kommunikation*** `receive(id, message)`: P_2 muss den Sender seiner Nachricht ***nicht*** kennen, sondern erhält die ID in einem Out-Parameter `id`:

9.1.4. Indirekte Kommunikation

Bei indirekter Kommunikation existieren spezifische OS-Objekte: ***Mailboxen, Ports oder Queues***.
- `send(Q, message)`: Prozess P_1 ***sendet*** Nachrichten ***an eine Queue*** Q
- `receive(Q, message)`: Prozess P_2 ***empfängt*** Nachrichten ***aus einer Queue*** Q

Kommunikation erfordert, dass beide Teilnehmer die ***gleiche Mailbox kennen***. Es kann ***mehr als eine*** Mailbox zwischen zwei Teilnehmern geben.

Mehr als zwei Teilnehmer

Bei mehr als zwei Teilnehmer müssen Regeln definiert werden, welcher Prozess die Nachricht empfängt, wie ***Beschränkung*** der Queue auf nur einen Sender und Empfänger, Beschränkung des Aufrufs von `receive()` auf nur einen Prozess, ***zufällige Auswahl*** oder Auswahl nach ***Algorithmus***.

Lebenszyklus der Queue

- ***Queue gehört einem Prozess:*** Queue lebt solange wie der Prozess
- ***Queue gehört dem Betriebssystem:*** Queue existiert unabhängig von einem Prozess. OS muss Mechanismus zum Erzeugen und Löschen der Queue zur Verfügung stellen.

9.1.5. Synchronisation

Message-Passing kann ***blockierend (synchron)*** oder ***nicht-blockierend (asynchron)*** sein, alle Kombinationen sind möglich:
- ***Blockierendes Senden:*** Sender wird solange blockiert, bis die Nachricht vom Empfänger empfangen wurde
- ***Nicht-blockierendes Senden:*** Sender sendet Nachricht und fährt sofort weiter
- ***Blockierendes Empfangen:*** Empfänger wird blockiert, bis Nachricht verfügbar
- ***Nicht-blockierendes Empfangen:*** Empfänger erhält Nachricht, wenn verfügbar, oder 0

Rendezvous

Sind Empfang und Versand ***beide blockierend***, kommt es zum ***Rendezvous***, sobald beide Seiten ihren Aufruf getätigt haben *(Sender weiss, dass Empfänger empfangen hat)*. OS kann eine ***Kopieroperation sparen*** und ***direkt*** vom Sende- in den Empfänger-Prozess kopieren. ***Impliziter Synchronisationsmechanismus***.

```
// Producer
message msg;
open (Q);
while (1) {
  produce_next (&msg);
  send (Q, &msg); // blocked until sent
}
```

```
// Consumer
message msg;
open (Q);
while (1) {
  receive (Q, &msg); // blocked until received
  consume_next (&msg);
}
```

9.1.6. Pufferung

Je nach Nachrichten-Kapazität der Queue kann man drei Arten des Pufferung unterscheiden:

- ***Keine:*** Queue-Länge ist 0, keine Nachrichten können gespeichert werden, Sender muss blockieren
- ***Beschränkte:*** Maximal n Nachrichten können gespeichert werden. Sender blockiert erst, wenn Queue voll ist.
- ***Unbeschränkte:*** Beliebig viele Nachrichten passen in die Queue, Sender blockiert nie

9.1.7. Prioritäten

In manchen Systemen können Nachrichten mit ***Prioritäten*** versehen werden. Der Empfänger holt die Nachricht mit der ***höchsten Priorität zuerst*** aus der Queue.

9.1.8. POSIX Message-Passing

OS-Message-Queues mit ***variabler Länge***, haben mindestens 32 Prioritäten und können ***synchron oder asynchron*** verwendet werden.

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

Öffnet eine Message-Queue mit systemweitem `name` und gibt einen ***Message-Queue-Descriptor*** zurück.

- ***`name`*** sollte immer mit «/» beginnen und sonst keine «/» enthalten
- ***`flags:`*** `O_RDONLY` für read-only, `O_RDWR` für lesen und schreiben, `O_CREAT` erzeugt Queue, falls sie nicht existiert und `O_NONBLOCK` definiert, dass `send` und `receive` nicht blockieren
- ***`mode:`*** Legt Zugriffsberechtigungen fest: `S_IRUSR | S_IWUSR`
- ***`struct mq_attr:`*** Beinhaltet Flags, maximale Anzahl Nachrichten in Queue, maximale Nachrichtengrösse und Anzahl der Nachrichten, die aktuell in der Queue sind. Lesen/Schreiben der Attribute mit `mq_getatrr()`/`mq_setattr()`.

```
int mq_close (mqd_t queue);
```

Schliesst die Queue mit dem Descriptor `queue` für diesen Prozess. Sie ***bleibt*** aber noch ***im System***, bis sie entfernt wird.

```
int mq_unlink (const char *name);
```

Entfernt die Queue mit dem Namen `name` aus dem System. ***Name*** wird ***sofort entfernt*** und Queue kann anschliessend ***nicht mehr geöffnet*** werden. ***Queue*** selber wird ***entfernt***, sobald ***alle Prozesse sie geschlossen*** haben.

```
int mq_send (mqd_t queue, const char *msg, size_t length, unsigned int priority);
```

Sendet die Nachricht, die an Adresse `msg` beginnt und `length` Bytes lang ist, in die `queue`. ***Blockiert*** erst, wenn die ***Queue voll*** ist *(ausser mit `O_NONBLOCK`, returnt dann –1)*. Gibt es auch als Variante mit ***Timeout***: `mq_timedsend()`.

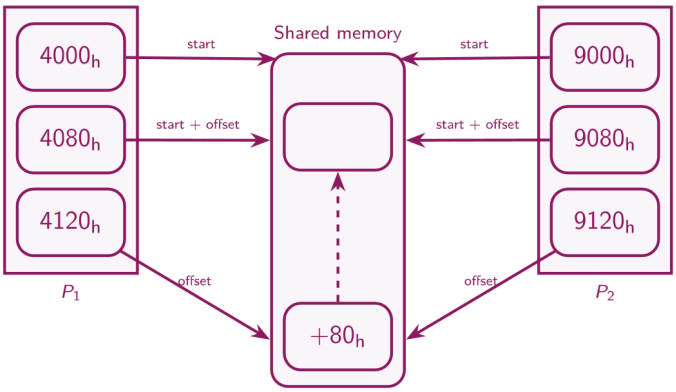
```
int mq_receive (mqd_t queue, const char *msg, size_t length, unsigned int *priority);
```

Kopiert die nächste Nachricht aus der Queue in den Puffer, der an Adresse `msg` beginnt und `length` Bytes lang ist *(sollte $\geq$ der maximalen Nachrichtengrösse der Queue sein)*. ***Blockiert***, wenn die Queue ***leer*** ist. Gibt es auch als Variante mit ***Timeout***: `mq_timedreceive()`. `priority` ist ein Out-Parameter für die Priorität der empfangenen Nachricht. Gibt ***Grösse*** der empfangenen Nachricht zurück.

9.2. SHARED MEMORY

Frames des Hauptspeichers werden ***zwei (oder mehr) Prozessen*** P_1 und P_2 ***zugänglich*** gemacht. In P_1 wird Page V_1 auf einen Frame F abgebildet. In P_2 wird Page V_2 auf ***denselben*** Frame F abgebildet. Beide Prozesse können ***beliebig*** auf dieselben Daten zugreifen.

Eine Adresse in V_1 ergibt nur für P_1 Sinn, dieselbe Adresse gehört für P_2 zu einer ***völlig anderen*** Speicherstelle. ***Keine absoluten Adressen/Pointer verwenden!*** Im Shared Memory müssen ***relative Adressen*** verwendet werden. *(Pointer müssen relativ zur Anfangs-Adresse sein, z.B. als Offset bezogen auf Start-Adresse)*



9.2.1. POSIX API

Das OS benötigt ein ***spezielles Objekt*** S, das Informationen über den gemeinsamen Speicher verwaltet. S wird in POSIX wie eine Datei verwendet.
Ausserdem benötigt das OS ein ***Objekt*** M_i, je Prozess P_i, der diesen Speicher verwenden möchte, um die spezifischen Mappings zu speichern *(Mapping Table)*.

```
int shm_open (const char *name, int flags, mode_t mode);
```

Öffnet ein Shared Memory mit system-weitem `name` und gibt FD zurück.
- ***name*** sollte immer mit «/» beginnen und sonst keine «/» enthalten
- ***flags:***
  - `O_RDONLY` für read-only
  - `O_RDWR` für lesen und schreiben
  - `O_CREAT` erzeugt Shared Memory, falls es nicht existiert
- ***mode:*** Legt Zugriffsberechtigungen fest: `S_IRUSR | S_IWUSR`

```
// Erzeugt (falls nötig) und öffnet Shared Memory /mysharedmemory zum Lesen und Schreiben
int fd = shm_open ("/mysharedmemory", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
```

```
int ftruncate (int fd, offset_t length);
```

Setzt Grösse der «Datei». Gibt 0 zurück wenn alles OK, sonst −1 und Fehlercode in `errno`. Muss ***zwingend*** nach Shared Memory-Erstellung gesetzt werden, um entsprechend viele Frames zu allozieren. Wird für Shared Memory ***mit ganzzahligen Vielfachen*** der Page-/Framegrösse verwendet.

```
int close (int fd);
```

Schliesst «Datei». Gibt 0 zurück wenn alles OK, sonst −1 und Fehlercode in `errno`. Shared Memory ***bleibt aber im System***, auch wenn kein Prozess das Shared Memory mehr offen hält.

```
int shm_unlink (const char * name);
```

Löscht das Shared Memory mit dem `name` aus dem System. Dies kann danach ***nicht mehr geöffnet*** werden, bleibt aber ***vorhanden***, bis es kein Prozess mehr geöffnet hält.

```
void * mmap(void *hint_address, size_t length, int protection, int flags,
int file_descriptor, off_t offset)
```

Mapped das Shared Memory, das mit `fd` geöffnet wurde, in den virtuellen Adressraum des laufenden Prozesses und gibt die (virtuelle) Adresse des ersten Bytes zurück.

```
void * address = mmap(
  0,                      // void *hint_address (0 because nobody cares)
  size_of_shared_memory,  // size_t length (same length as used in ftruncate)
  PROT_READ | PROT_WRITE, // int protection (never use execute!)
  MAP_SHARED,             // int flags
  fd,                     // int file_descriptor
  0                       // off_t offset (start map from first byte)
);
```

```
int munmap (void *address, size_t length);
```

Entfernt das Mapping wieder aus dem virtuellen Adressraum.

9.3. VERGLEICH MESSAGE-PASSING & SHARED MEMORY

Shared Memory ist oft der ***schneller zu realisierende Ansatz***. Existierende Applikationen können relativ schnell auf mehrere Prozesse mit Shared Memory umgeschrieben werden. Oft aber ***schwer wartbar***. Das System ist ***weniger stark modularisiert*** und Prozesse sind schlechter gegeneinander ***geschützt***.

Message-Passing erfordert ***mehr Engineering-Aufwand***. Existierende Applikationen müssen in grossen Teilen neu implementiert werden. Bei ***sauber gekapselten*** Anwendungen viel geringeres Problem. Lassen sich leicht als ***verteilte Systeme*** erweitern. Bei einigen OS sogar schon implementiert, z.B. QNX

9.3.1. Performance

– ***Einzel-Prozessor-System:*** Im Normalfall Shared-Memory wegen entfallenden Kopieroperationen
– ***Mehr-Prozessor-System:*** Shared-Memory benötigt zusätzlichen Aufwand aufgrund Cache-Synchronisation

Message-Passing-Systeme liegen auf Mehr-Prozessor-Systemen häufig ***gleichauf*** und werden in Zukunft vermutlich sogar ***performanter*** sein als Shared-Memory-Systeme.

9.4. VERGLEICH MESSAGE-QUEUES & PIPES

Message-Queues	***Pipes***
– bidirektional	– unidirektional
– Daten sind in einzelnen Messages organisiert	– übermittelt Bytestrom an Daten
– beliebiger Zugriff	– FIFO-Zugriff
– Haben immer einen Namen	– Müssen keinen Namen haben

10. UNICODE

10.1. ASCII

ASCII *(American Standard Code for Information Interchange)* hat ***128 definierte Zeichen*** *(7 Bit, 00_h bis $7F_h$)*. 33 Kontrollzeichen, *(viele davon obsolet)*, 10 Ziffern, 33 Interpunktionszeichen, 26 Grossbuchstaben und 26 Kleinbuchstaben.
(erste Hexzahl = Zeile, zweite Hexzahl = Spalte, d.h. 41_h = A)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SB	ESC	FS	GS	RS	US
2	␣	!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

10.1.1. Codepages

Codepages sind ***unabhängige Erweiterungen*** auf 8 Bit. Es gibt viele verschiedene, jede ist anders *(z.B. für andere Sprachen)*. Sie definieren jeweils 128 Zeichen von 80_h bis FF_h. Die Codierung ist ***nicht inhärent erkennbar***, Programme müssen wissen, welche Codepage verwendet wird, sonst wird der Text unleserlich.

10.2. UNICODE

Unicode hat zum Ziel, einen eindeutigen Code für ***jedes vorhandene Zeichen*** zu definieren. Hat Platz für 1’112’064 Code-Points *(21 bits)*, davon 149’813 verwendet. Benötigt also maximal 21 Bits, um alle Unicode-Zeichen darzustellen. Der Bereich von $D8\,00_h$ bis $DF\,FF_h$ enthält aufgrund von UTF-16 keine gültigen Code-Points.

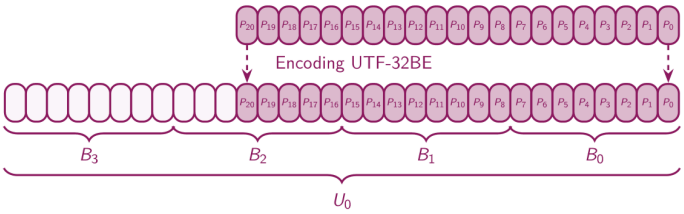
10.2.1. Verschiedene Encodings

Man unterscheidet ***Code-Points*** *(Nummer eines Zeichen - «welches Zeichen»)* und ***Code-Units*** *(Einheit, um Zeichen in einem Encoding darzustellen - bietet den Speicherplatz für das Zeichen)*.
$P_i = i$-tes Bit des unkodierten CPs, $U_i = i$-tes Code-Unit des kodierten CPs, $B_i = i$-tes Byte des kodierten CPs

– ***UTF-32:*** Jede CU umfasst ***32 Bit***, jeder CP kann mit ***einer CU*** dargestellt werden.
– ***UTF-16:*** Jede CU umfasst ***16 Bit***, ein CP benötigt ***1 oder 2 CUs***
– ***UTF-8:*** Jede CU umfasst ***8 Bit***, ein CP benötigt ***1 bis 4 CUs***

10.2.2. UTF-32

Direkte Kopie der Bits in die CU bei Big Endian, bei Little Endian werden P_0 bis P_7 in B_3 kopiert usw. Wird häufig intern in Programmen verwendet. Wurde aufgrund Memory Alignment auf 32 Bits erweitert, obwohl 24 Bits *(3 CUs)* genügen würden *(Bessere Performance)*. Obere 11 Bits werden oft «zweckentfremdet».



10.2.3. UTF-16

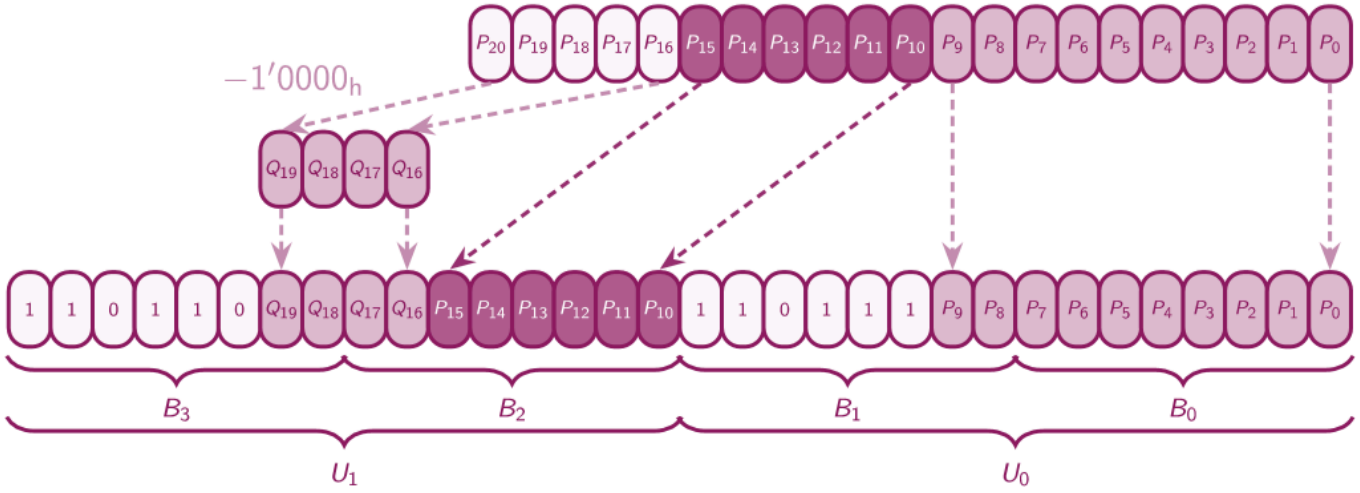
Encoding muss Endianness berücksichtigen. Die 2 CUs werden ***Surrogate Pair*** genannt.
U_0: high surrogate, U_1: low surrogate.

- ***UTF-16BE:*** Big Endian, CP mit 1 CU: $U_0 = B_1 B_0$, CP mit 2 CUs: $U_1 U_0 = B_3 B_2 B_1 B_0$
- ***UTF-16LE:*** Little Endian, CP mit 1 CU: $U_0 = B_0 B_1$, CP mit 2 CUs: $U_1 U_0 = B_2 B_3 B_0 B_1$

Bei ***2 Bytes*** *(1 CU)* wird direkt gemappt und vorne mit Nullen aufgefüllt.

Bei ***4 Bytes***, wenn CP in $1\,00\,00_\text{h}$ bis $10\,\text{FF}\,\text{FF}_\text{h}$ sind, ist der Bereich von $\text{D8}\,00_\text{h}$ bis $\text{DF}\,\text{FF}_\text{h}$ *(Bits 17-21)* wegen dem Separator ***ungültig*** und muss «umgerechnet» werden:

- $Q = P - 1\,00\,00_\text{h}$, also ist Q in 0_h bis $\text{F}\,\text{FF}\,\text{FF}_\text{h}$
- $U_1 = 110\,110\text{x}\,\text{xxxx}_\text{b} + \text{D8}\,00_\text{h}$, $U_0 = 1101\,11\text{xx}\,\text{xxxx}\,\text{xxxx}_\text{b} + \text{DC}\,00_\text{h}$



Beispiel

Encoding von U+10′437 (𐐷) $00\,0100\,0001\;00\,0011\,0111_\text{b}$:

1. Code-Point P minus $1\,00\,00_\text{h}$ rechnen und in Binär umwandeln
   $P = 1\,04\,37_\text{h}$, $Q = 1\,04\,37_\text{h} - 1\,00\,00_\text{h} = 04\,37_\text{h} = 00\,0000\,0001\;00\,0011\,0111_\text{b}$
2. Obere & untere 10 Bits in Hex umwandeln
   $00\,01\;01\,37_\text{h}$
3. Oberer Wert mit $\text{D8}\,00_\text{h}$ und unterer Wert mit $\text{DC}\,00_\text{h}$ addieren, um Code-Units zu erhalten
   $U_1 = 00\,01_\text{h} + \text{D8}\,00_\text{h} = \text{D8}\,01_\text{h}$, $U_2 = 01\,37_\text{h} + \text{DC}\,00_\text{h} = \text{DD}\,37_\text{h}$
4. Zu BE/LE zusammensetzen
   $\text{BE} = \underline{\text{D8}\,01\,\text{DD}\,37_\text{h}}$, $\text{LE} = \underline{01\,\text{D8}\,37\,\text{DD}_\text{h}}$

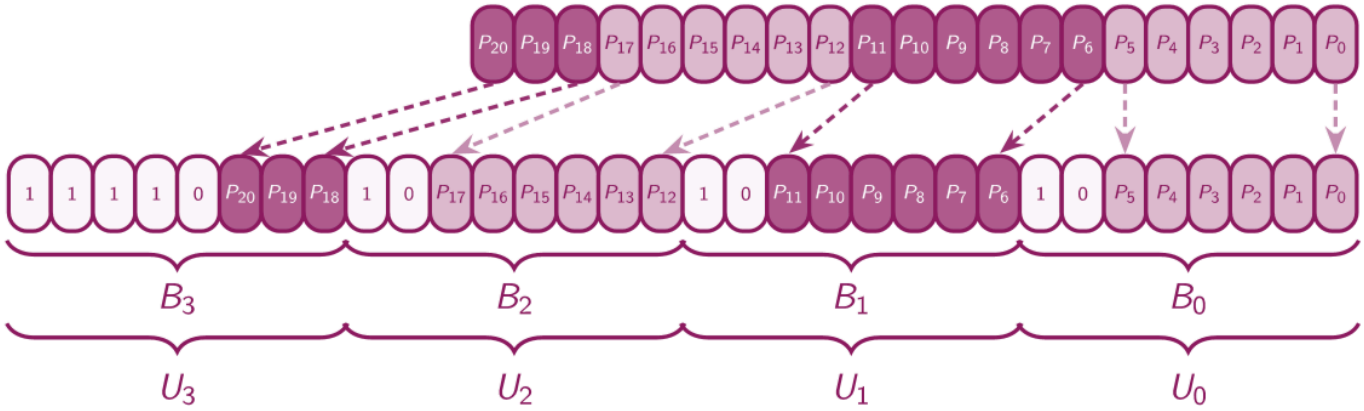
10.2.4. UTF-8

Encoding muss Endianness ***nicht*** berücksichtigen. Standard für Webpages. Echte Erweiterung von ASCII.

Code-Point in	U_3	U_2	U_1	U_0	*signifikant*
0_h - $7F_h$				0xxx xxxx$_b$	7 bits
80_h - 7 FF$_h$			110x xxxx$_b$	10xx xxxx$_b$	11 bits
8 00_h - FF FF$_h$		1110 xxxx$_b$	10xx xxxx$_b$	10xx xxxx$_b$	16 bits
1 00 00_h - 10 FF FF$_h$	1111 0xxx$_b$	10xx xxxx$_b$	10xx xxxx$_b$	10xx xxxx$_b$	21 bits

Die most significant Bits einer CU werden als Delimiter verwendet: 0_b = nur 1 CU, 10_b = es folgt mindestens 1 CU, 110_b = 2. und letzte CU, 1110_b = 3. und letzte CU, 1 1110_b = 4. und letzte CU.

In den CUs haben die Bytes 0_h - 7F$_h$ *(7 signifikante Bits)*, 80_h - 7 FF$_h$ *(11 Bits)*, 8 00_h - FF FF$_h$ *(16 bits)* bzw. 1 00 00_h - 10 FF FF$_h$ *(21 bits)* Platz.



Beispiele

- ***ä***: $P = \texttt{E4}_h = 0\,0011\,10\,0110_b$
  $\Rightarrow P_{10}...P_6 = 0\,0011_b = 03_h$, $P_5...P_0 = 10\,0100_b = 24_h$
  $\Rightarrow U_1 = \texttt{C0}_h\ (= 1100\,0000_b) + 03_h = \texttt{C3}_h$, $U_0 = 80_h\ (= 1000\,0000_b) + 24_h = \texttt{A4}_h$
  $\Rightarrow$ ä $= \underline{\texttt{C3 A4}_h}$
- ***ặ***: $P = \texttt{1E B7}_h = 0001\,11\,1010\,11\,0111_b$
  $\Rightarrow P_{15}...P_{12} = 01_h$, $P_{11}...P_6 = 3A_h$, $P_5...P_0 = 37_h$
  $\Rightarrow U_2 = \texttt{E0}_h\ (= 1110\,0000_b) + 01_h = \texttt{E1}_h$, $U_1 = 80_h + 3A_h = \texttt{BA}_h$, $U_0 = 80_h + 37_h = \texttt{B7}_h$
  $\Rightarrow$ ặ $= \underline{\texttt{E1 BA B7}_h}$

10.2.5. Encoding-Beispiele

Zeichen	*Code-Point*	*UTF-32BE*	*UTF-32LE*	*UTF-8*	*UTF-16BE*	*UTF-16LE*
A	41_h	00 00 00 41_h	41 00 00 00_h	41_h	00 41_h	41 00_h
ä	$E4_h$	00 00 00 $E4_h$	E4 00 00 00_h	C3 $A4_h$	00 $E4_h$	E4 00_h
α	3 $B1_h$	00 00 03 $B1_h$	B1 03 00 00_h	CE $B1_h$	03 $B1_h$	B1 03_h
ặ	1E $B7_h$	00 00 1E $B7_h$	B7 1E 00 00_h	E1 BA $B7_h$	1E $B7_h$	B7 $1E_h$
𝄞	1 03 30_h	00 01 03 30_h	30 03 01 00_h	F0 90 8C $B0_h$	D8 00 DF 30_h	00 D8 30 DF_h

Bei LE / BE werden nur die Zeichen **innerhalb** *eines Code-Points vertauscht, nicht die Code-Points an sich.*

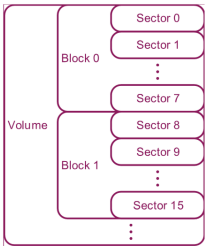
11. EXT2-DATEISYSTEM

11.1. DATENTRÄGER-GRUNDBEGRIFFE

- ***Partition:*** Ein Teil eines Datenträgers, wird selbst wie ein Datenträger behandelt.
- ***Volume:*** Ein Datenträger oder eine Partition davon.
- ***Sektor:*** Kleinste logische Untereinheit eines Volumes. Daten werden als Sektoren transferiert. Grösse ist von HW definiert *(z.B. 512 Bytes oder 4KB)*. Enthält Header, Daten und Error-Correction-Codes.
- ***Format:*** Layout der logischen Strukturen auf dem Datenträger, wird vom Dateisystem definiert.

11.2. BLOCK

Ein Block besteht aus ***mehreren aufeinanderfolgenden Sektoren*** *(1 KB, 2 KB oder 4 KB (normal))*.
Das gesamte Volume ist in ***Blöcke aufgeteilt*** und Speicher wird ***nur in Form von Blöcken*** alloziert.
Ein Block enthält nur Daten einer ***einzigen Datei***.



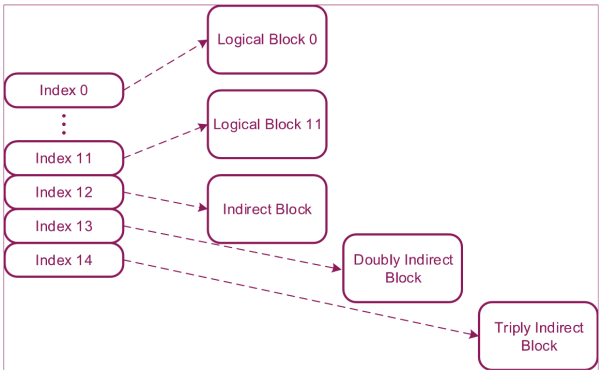
- ***Logische Blocknummer:*** Blocknummer vom Anfang der Datei aus gesehen, wenn Datei eine ununterbrochene Abfolge von Blöcken wäre *(innerhalb Datei)*
- ***Physische Blocknummer:*** Tatsächliche Blocknummer auf dem Volume *(auf dem Datenträger)*

11.3. INODES

Achtung: ≠ Index Node. Beschreibung einer Datei. Enthält ***alle Metadaten*** über die Datei, ***ausser Namen oder Pfad*** *(Grösse, Anzahl der verwendeten Blöcke, Erzeugungszeit, Zugriffszeit, Modifikationszeit, Löschzeit, Owner-ID, Group-ID, Flags, Permission Bits)*.
Hat eine ***fixe Grösse*** je Volume: Zweierpotenz, mind. 128 Byte, max 1 Block.

Der Inode ***verweist auf die Blöcke***, die ***Daten für eine Datei*** enthalten. Enthält ein Array `i_block` mit 15 Einträgen zu je 32 Bit:

- 12 Blocknummern für die ***ersten 12 Blöcke*** einer Datei
- 1 Blocknummer des ***indirekten Blocks***, der wiederum bei 1024 Byte Blockgrösse auf 256 oder bei 4096 Byte auf 1024 Blöcke verweist.
- 1 Blocknummer des ***doppelt indirekten Blocks***, welcher Nummern von indirekten Blöcken enthält. Bei Blockgrösse 1024 auf $256 \cdot 256 = 65536$ Blöcke, bei 4096 auf $1024 \cdot 1024 = 1\text{M}$ Blöcke
- 1 Blocknummer des ***dreifach indirekten Blocks***
  $256 \cdot 256 \cdot 256 = 16\text{M}$ *bzw.* $1024 \cdot 1024 \cdot 1024 = 1\text{G}$



Jeder verwendete Block einer Datei hat einen direkten oder indirekten ***Verweis***.

11.3.1. Lokalisierung

Alle Inodes aller Blockgruppen gelten als ***eine grosse Tabelle***. Zählung der Inodes startet mit 1.

- Blockgruppe $= (\text{Inode} - 1) / \text{ Anzahl Inodes pro Gruppe}$
- Index des Inodes in Blockgruppe $= (\text{Inode} - 1) \bmod \text{Anzahl Inodes pro Gruppe}$
- Sektor und Offset können anhand der Daten aus dem Superblock bestimmt werden.

11.3.2. Erzeugung

Neue Verzeichnisse werden bevorzugt in der Blockgruppe angelegt, die von allen Blockgruppen mit ***überdurchschnittlich vielen freien Inodes*** die ***meisten Blöcke frei*** hat. Dateien werden möglichst in der Blockgruppe des Verzeichnisses oder in nahen Gruppen angelegt.
Bestimmung des ersten freien Inodes in der Gruppe anhand des ***Inode-Usage-Bitmaps***. Bit wird entsprechend auf 1 gesetzt und die Anzahl freier Inodes in Gruppendeskriptor und Superblock angepasst.

11.3.3. File-Holes

Bereiche in der Datei, in der ***nur Nullen*** stehen. Wird ein Eintrag auf einen Block auf 0 gesetzt, heisst das, dass der Block nur Nullen enthält. Ein solcher Block wird ***nicht alloziert***. Darum kann Grösse & Anzahl der verwendeten Blöcke voneinander abweichen.

11.4. BLOCKGRUPPE

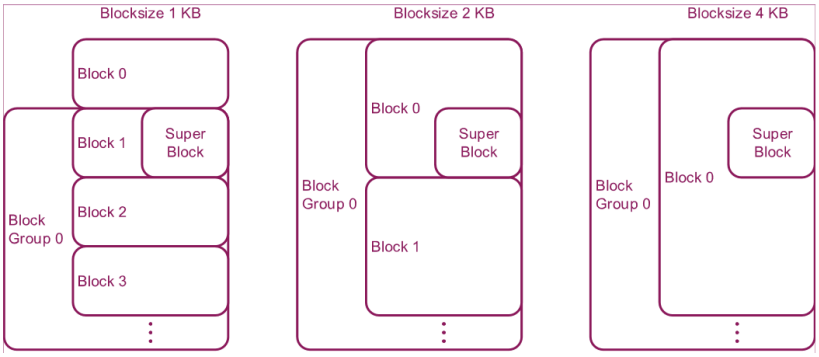
Ein Volume wird in ***Blockgruppen*** unterteilt. Eine Blockgruppe besteht aus ***mehreren aufeinanderfolgenden Blöcken*** bis zu 8 mal der Anzahl Bytes in einem Block *(Bsp. Blockgrösse 4KB sind bis zu 32K Blöcke in einer Gruppe)*. Anzahl Blöcke je Gruppe ist gleich für alle Gruppen.

11.4.1. Lage der Blockgruppen

Die Lage der Blockgruppe 0 ist ***abhängig von der Blockgrösse***. Blockgruppe 0 ist definiert als ***die Gruppe, deren erster Block den Superblock enthält***.

Blockgrösse $\leq$ 1024: Block 0 kommt vor Blockgruppe 0 → Block 1 ist der erste Block und beinhaltet Superblock.

Blockgrösse $>$ 1024: Block 0 in Blockgruppe 0 → Superblock ist in Block 0.



11.4.2. Layout

- ***Block 0:*** Kopie des Superblocks
- ***Block 1 bis n:*** Kopie der Gruppendeskriptorentabelle
- ***Block $n+1$:*** Block-Usage-Bitmap mit einem Bit je Block der Gruppe *(Welche Blöcke werden gerade verwendet)*
- ***Block $n+2$:*** Inode-Usage-Bitmap mit einem Bit je Inode der Gruppe *(Welche Inodes werden verwendet)*
- ***Block $n+3$ bis $n+m+2$:*** Tabelle aller Inodes in dieser Gruppe
- ***Block $n+m+3$ bis Ende der Gruppe:*** Blöcke der eigentlichen Daten

11.4.3. Superblock

Startet immer an ***Byte 1024*** *(Wegen eventuellen Bootdaten im Bereich vorher)* und enthält ***alle Metadaten*** über das Volume:

- ***Anzahlen:*** Inodes frei und gesamt, Blöcke frei und gesamt, reservierte Blöcke, Bytes je Block, Bytes je Inode, Blöcke je Gruppe, Inodes je Gruppe
- ***Zeitpunkte:*** Mountzeit, Schreibzeit, Zeitpunkt des letzten Checks
- ***Statusbits:*** Um Fehler zu erkennen
- ***Erster Inode***, der von Applikationen verwendet werden kann
- ***Feature-Flags:*** Zeigen an welche Features das Volume verwendet.

Sparse Superblocks

Feature, dass die ***Anzahl der Superblocks*** stark ***reduziert***. Wird über ein bestimmtes Flag aktiviert. Wenn aktiv, dann werden Kopien des Superblocks nur noch in Blockgruppe 0 und 1 sowie allen reinen Potenzen von 3, 5 oder 7 gehalten *(0, 1, 3, 5, 7, 9, 25, 27, 49, 81, 125, 243, 343, ...)*. Dadurch ist immer noch ein ***sehr hoher Wiederherstellungsgrad*** möglich, obwohl deutlich weniger Platz verwendet wird.

11.4.4. Gruppendeskriptor

32 Byte ***Beschreibung einer Blockgruppe***. Beinhaltet:

- Blocknummer des ***Block-Usage-Bitmaps***
- Blocknummer des ***Inode-Usage-Bitmaps***
- Nummer des ersten Blocks der ***Inode-Tabelle***
- Anzahl ***freier Blöcke*** und ***Inodes*** in der Gruppe
- Anzahl der ***Verzeichnisse*** in der Gruppe

Gruppendeskriptortabelle

Eine ***Tabelle mit n Gruppendeskriptoren*** für alle n Blockgruppen im Volume. Benötigt selbst $32 \cdot n$ Anzahl Bytes; $\text{Anzahl Sektoren} = (32 \cdot n)/\text{Sektorgrösse}$. Folgt ***direkt*** auf Superblock. Kopie der Tabelle direkt nach jeder Kopie des Superblocks.

11.5. VERZEICHNISSE

Ein Verzeichnis enthält die ***Dateieinträge*** bzw. den Inode, dessen Datenbereich Dateieinträge enthält. Es gibt ***zwei automatisch angelegte*** Einträge: `"."` ist der Dateieintrag mit eigenem Inode, `".."` ist der Dateieintrag mit dem Inode des Elternverzeichnis. Das ***Wurzelverzeichnis*** ist der Inode Nummer 2.

Ein ***Dateieintrag*** hat eine variable Länge von 8 - 263 Bytes:
- 4 Byte ***Inode***
- 2 Byte ***Länge*** des ***Eintrags***
- 1 Byte ***Länge*** des ***Dateinamens***
- 1 Byte ***Dateityp*** *(1: Datei, 2: Verzeichnis, 7: Symbolischer Link)*
- 0 - 255 Byte ***Dateiname*** *(ASCII, nicht null-terminiert)*
- Länge wird aus Effizienzgründen immer auf 4 Byte aligned *(Maschinenwort)*

11.6. LINKS

- ***Hard-Link:*** gleicher Inode, verschiedene Pfade *(Inode wird von verschiedenen Dateieinträgen referenziert)*
- ***Symbolischer Link:*** Wie eine Datei, Datei enthält Pfad anderer Datei *(Pfad $<$ 60 Zeichen: Wird in Blockreferenzen-Array gespeichert, Pfad $\geq$ 60: Pfad wird in eigenem Block gespeichert)*

11.7. VERGLEICH FAT, NTFS, EXT2

FAT	***Ext2***	***NTFS***
– Verzeichnis enthält alle Daten über die Datei – Datei ist in einem einzigen Verzeichnis – Keine Hard-Links möglich	– Dateien werden durch Inodes beschrieben – Kein Link von der Datei zurück zum Verzeichnis – Hard-Links möglich *(Mehrere Links zum Inode möglich)*	– Dateien werden durch File-Records beschrieben – Verzeichnis enthält Namen und Link auf Datei – Link zum Verzeichnis und Name sind in einem Attribut – Hard-Links möglich *(Attribut kann mehrfach vorkommen)*

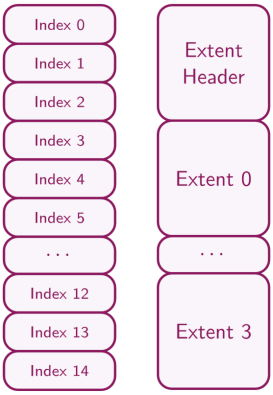
12. EXT4

In Ext4 sind die wichtigen Datenstrukturen vergrössert *(**Inodes** haben 256 Byte statt 128, **Gruppendeskriptoren** 64 Byte statt 32, **Blockgrösse** bis 64 KB)*. Grosse Blöcke sind besser für viele grosse Dateien, da weniger Metadaten benötigt werden. Erlaubt höhere maximale Dateigrösse. Zudem werden Blöcke von den Inodes mit ***Extent Trees*** verwaltet und ***Journaling*** wird verwendet.

12.1. EXTENTS

Ein ***Extent*** beschreibt ein ***Intervall physisch konsekutiver Blöcke***. Ist 12 Byte gross *(4 Byte logische Blocknummer, 6 Byte physische Blocknummer, 2 Byte Anzahl Blöcke)*.
Positive Zahlen = Block initialisiert, Negativ = Block voralloziert.

Da eine Einschränkung auf ausschliesslich konsekutive Dateien nicht praktikabel ist, muss eine Datei ***mehr als einen Extent umfassen*** können. Im Inode hat es in den 60 Byte für direkte und indirekte Block-Adressierung Platz für 4 Extents und einen Header.



12.2. EXTENT TREES

Index-Knoten *(Innerer Knoten des Baums, besteht aus Index-Eintrag und Index-Block)*
Index-Eintrag *(Enthält Nummer des physischen Index-Blocks und kleinste logische Blocknummer aller Kindknoten)*

12.2.1. Extent Tree Header

Für mehr als 4 Extents braucht man einen zusätzlichen Block. Deshalb sind die ersten 12 Byte im Inode kein Extent, sondern der Extent Tree Header:

- 2 Byte ***Magic Number*** $\texttt{F3 0A}_\text{h}$
- 2 Byte ***Anzahl Einträge***, die ***direkt*** auf den Header folgen *(Wie viele Extents folgen dem Header?)*
- 2 Byte ***Anzahl Einträge***, die ***maximal*** auf den Header folgen können
- 2 Byte ***Tiefe des Baums*** *(0: Einträge sind Extents, $\geq$1: Einträge sind Index Nodes)*
- 4 Byte ***reserviert***

12.2.2. Index Nodes

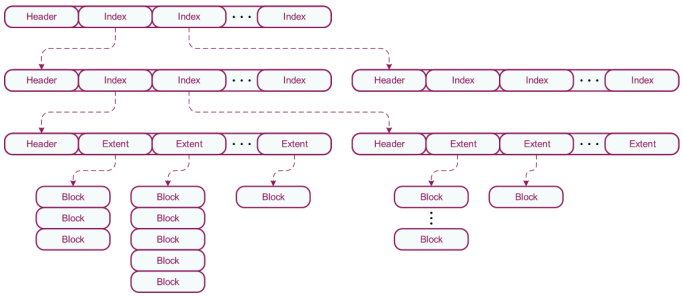
Ein Index-Node spezifiziert ***einen Block, der Extents enthält***. Der Block enthält am Anfang einen ***Header*** und danach die Extents *(max. 340 bei 4 KB Blockgrösse)*.

- 4 Byte ***kleinste logische Blocknummer*** aller Kind-Extents
- 6 Byte ***physische Blocknummer*** des Blocks, auf den der Index-Node verweist
- 2 Byte ***unbenutzt***

Werden mehr als $4 \cdot 340 = 1360$ ***Extents*** *($5\,50_h$)* benötigt, muss man Blöcke mit Index-Nodes einführen. Statt Extents stehen dann ***Index Nodes im Block***. Die ***Tiefe*** im Inode wird auf ***2*** gesetzt, in den Index-Node-Blöcken auf 1. Die ***kleinste logische Blocknummer*** aller Kind-Extents ***propagiert*** dann bis in den jeweils obersten Index-Node. Benötigt man dann ***noch mehr Extents***, kann die ***Tiefe*** im Inode bis auf ***5*** gesetzt werden *(Dann wird das Maximum von $2^{32} = 4\text{G}$ Blöcken pro Datei erreicht)*.

12.2.3. Index-Block

Ein Index-Block enthält einen eigenen ***Tree-Header***, Tiefe ist um 1 kleiner als beim übergeordneten Knoten. Enthält ***Referenz auf die Kind-Knoten***: je nach Tiefe entweder Index-Einträge oder Extents. `i_block[0...14]` kann als (sehr kleiner) Index-Block aufgefasst werden.



12.2.4. Notation

(in)direkte Adressierung	***Extent-Trees***
direkte Blöcke: <Index> $\mapsto$ <Blocknummer>	***Indexknoten:*** <Index> $\mapsto$ (<Kindblocknummer>, <kleinste Nummer der 1. logischen Blöcke aller Kinder>)
indirekte Blöcke: <indirekter Block>.<Index> $\mapsto$ <direkter Block>	***Blattknoten:*** $\mapsto$ (<1. logischer Block>, <1. physischer Block>, <Anzahl Blöcke>)
	Header: <Index> $\mapsto$ (<Anz. Einträge>, <Tiefe>)

Beispiel-Berechnung: 4MB grosse, konsekutiv gespeicherte Datei, 4KB Blöcke ab Block $10\,00_\text{h}$

(In-)direkte Block-Adressierung

4 MB = 2^{22}B, 4 KB = 2^{12}B, $2^{22-12} = 2^{10} = 4\,00_\text{h}$ Blöcke von $10\,00_\text{h}$ bis $13\,\text{FF}_\text{h}$

$0 \mapsto 10\,00_\text{h}$, $1 \mapsto 10\,02_\text{h}$, ..., $\text{B}_\text{h} \mapsto 10\,0\text{B}_\text{h}$, $\text{C}_\text{h} \mapsto 14\,00_\text{h}$ *(indirekter Block, $4\,00_\text{h}$ nach Startblock)*

$14\,00_\text{h}.0_\text{h} \mapsto 10\,0\text{C}_\text{h}$, $14\,00_\text{h}.1_\text{h} \mapsto 10\,0\text{D}_\text{h}$, ..., $14\,00_\text{h}.3\,\text{F3}_\text{h} \mapsto 13\,\text{FF}_\text{h}$

Extent Trees

Header: $0 \mapsto (1, 0)$

Extent: $1 \mapsto (0, 10\,00_\text{h}, 4\,00_\text{h})$

12.3. JOURNALING

Wird eine Datei ***erweitert***, passiert folgendes:
- ***Neue Blöcke*** werden für die Daten ***alloziert***
- Der ***Inode*** der Datei wird ***angepasst***, um die Blöcke zu referenzieren
- Die ***Block-Usage-Bitmaps*** werden ***angepasst***
- Die ***Counter*** freier und benutzter Blöcke werden ***angepasst***
- Die ***Daten*** werden in die Datei geschrieben

Wenn das Dateisystem dabei ***unterbrochen*** wird, kann es zu ***Inkonsistenzen*** kommen. Ein System ***ohne*** Journaling kann sehr lange brauchen, um ein Dateisystem auf Inkonsistenzen zu prüfen, da ***alle Metadaten*** überprüft werden müssen. ***Journaling verringert diese Prüfung erheblich***. Dateisystem muss nur die Metadaten überprüfen, die noch im Journal referenziert sind.

12.3.1. Journal

Das Journal ist eine ***reservierte Datei***, in die Daten relativ ***schnell geschrieben*** werden können. Besteht aus ***wenigen sehr grossen Extents*** oder bestenfalls aus ***einem einzigen Extent*** *(Typischerweise Inode 8, 128MB)*.

Eine ***Transaktion*** ist eine Folge von Einzelschritten, die das Dateisystem gesamtheitlich vornehmen soll.

12.3.2. Journaling und Committing

Daten werden zuerst als ***Transaktion ins Journal*** geschrieben *(Journaling)*. Daten werden erst ***danach*** an ihre ***endgültige Position*** geschrieben *(Committing)*. Daten werden nach dem Commit aus dem Journal ***entfernt***.

Journaling ist ***schneller***, weil alle Daten in ***konsekutive*** Blöcke geschrieben werden. Committing muss u.U. ***viele verschiedene*** Blöcke modifizieren.

12.3.3. Journal Replay

Startet das System neu, kann es ***anhand der Journal-Einträge*** die Metadaten untersuchen, die ***potenziell korrupt*** sein könnten. Alle Transaktionen, die noch im Journal sind, wurden ***noch nicht durchgeführt*** und werden mit Journal Replay (noch einmal) ausgeführt oder auf Fehler überprüft. ***Im Gegensatz zu ext2 muss nicht der gesamte Datenträger auf Fehler untersucht werden.***

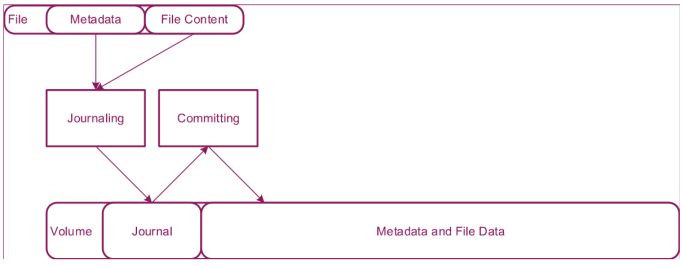
12.3.4. Journaling Modi

Es gibt 3 Modi: (Full) Journal, Ordered und Writeback. Die Modi ***Ordered*** und ***Writeback*** schreiben nur ***Metadaten***, ***Journal*** schreibt auch ***Datei-Inhalte*** ins Journal.

(Full) Journal

Metadaten und Datei-Inhalte kommen ins Journal. Grosse Änderungen werden in mehrere Transaktionen gesplittet.

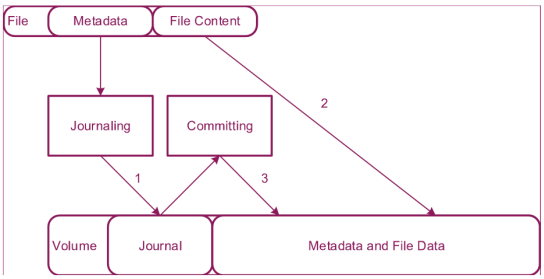
Vorteil: maximale Datensicherheit
Nachteil: grosse Geschwindigkeitseinbussen.



Ordered

Nur Metadaten kommen ins Journal. Dateiinhalte werden immer ***vor*** dem Commit geschrieben:
1. Transaktion ins Journal
2. Dateiinhalte an endgültige Position schreiben
3. Commit ausführen

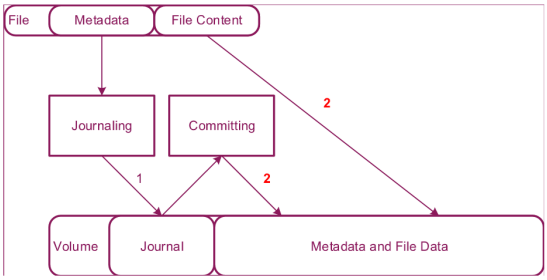
Vorteil: Dateien enthalten nach dem Commit den richtigen Inhalt
Nachteil: Etwas geringere Geschwindigkeit als Writeback.
(In Linux gibt es einen lost+found Ordner im Root-Verzeichnis, Dateien mit unvollständigen Transaktionen bei z.B. Absturz werden da deponiert)



Writeback

Nur Metadaten kommen ins Journal. Commit und Schreiben der Dateiinhalte werden in ***beliebiger Reihenfolge*** ausgeführt.

Vorteil: Sehr schnell, keine Synchronisation von Commit und Datenschreiben nötig.
Nachteil: Dateien können Datenmüll enthalten.



12.4. VERGLEICH EXT2 & EXT4

Ext2	*Ext4*
– schlank und leistungsfähig – einfach zu implementieren – mächtiger als FAT, weniger mächtig als NTFS	– fügt wichtige Features hinzu – Journaling – Effizientere Verwaltung grosser Verzeichnisse und Dateien

13. X WINDOW SYSTEM

13.1. GUI BASISKONZEPTE

Frühere Unix-Systeme waren rein ***textorientierte*** Bedienschnittstellen und basierten auf ***programmgesteuerten Interaktionen***. Moderne Unix-Systeme verwenden ein ***GUI mittels des X Window System*** oder mittels anderen Technologien *(Google Android, Apple Aqua, Canonical Mir Display Server, Wayland).* Diese sind ***Ereignisgesteuerte Interaktionen:*** Benutzer entscheidet, wann welches Ereignis ausgelöst wird, Programm reagiert auf Benutzer.

13.1.1. Vorteile

Das X Window System ist auf dem Unix-Kern aufgesetzt und damit ***austauschbar***. Installierbar, wenn ***tatsächlich benötigt***. Realisiert ***Netzwerktransparenz***, ***Plattform-unabhängig***. X legt die ***GUI-Gestaltung nicht*** fest.

13.1.2. Fenster

Rechteckiger Bereich des Bildschirms. Kann beliebig viele weitere Fenster enthalten *(z.B. Dialogbox, Scrollbar, Button…)*, es gibt eine ***Baumstruktur*** aller Fenster. Der ***Bildschirm*** ist die Wurzel.

13.1.3. Maus und Mauszeiger

Die ***Maus*** ist ein physisches Gerät, das 2D-Bewegungen in Daten übersetzt. Der ***Mauszeiger*** ist eine Rastergrafik, die auf dem Bildschirm angezeigt wird.

Das OS bewegt den Mauszeiger ***analog*** zur physischen Bewegung der Maus. Die ***Maustasten lösen Ereignisse*** aus. Das Ereignis soll für ***das Fenster*** gelten, über dem sich ***der Hotspot*** befindet.

Der ***Maustreiber erzeugt Nachrichten***, das ***OS verteilt*** diese an die zuständige ***Applikation***, welche die Nachricht ***verarbeitet***. Dieser Prozess ist ***asynchron***.

13.1.4. GUI Architektur

Das GUI braucht mehr als X Window System.
– ***X Window System:*** Grundfunktionen der Fensterdarstellung *(Events von Kernel erhalten & Fenster zuordnen)*
– ***Window Manager:*** Verwaltung der sichtbaren Fenster, Umrandung, Knöpfe
– ***Desktop Manager:*** Desktop-Hilfsmittel wie Taskleiste, Dateimanager, Papierkorb etc.

X selbst ist ***unabhängig*** von einem bestimmten Window Manager oder Desktop. Es existieren ***viele verschiedene Implementierungen*** des Window Manager und Desktops. Die ***Gestaltung der Bedienoberfläche*** und Bedienphilosophie bleiben damit ***frei***.

13.1.5. Window-Manager

Läuft im Client und realisiert eine ***Window Layout Policy***. Platziert Client-Fenster auf dem Bildschirm. Ist (nur) eine ***Client-Applikation mit Sonderrechten*** zur Fensterverwaltung.

Typische Dienste des Window Manager für den Benutzer:
- Applikationsfenster mit ***Titelleiste, Umrandung und zusätzlichen Knöpfen*** versehen
- Fenster ***verschieben***, ***Grösse*** ändern, minimieren, maximieren
- Neue Applikationen ***starten***
- ***Darstellungsreihenfolge*** *(stacking order)* überlappender Fenster ändern

13.1.6. Fensterverwaltung

Fensterhierarchie: Root-Window ist zuoberst, bedeckt den ganzen Bildschirm. Kinder des Root-Windows sind Top-Level Windows der Applikationen. Die übrigen Kindfenster sind zur Anzeige von Menüs, BUTTons, usw. in Applikationen.
Kindfenster können Elternfenster teilweise ***überlappen***, jedoch overflow hidden. Eingaben werden nur im Überlappungsbereich empfangen. Es gibt zwei unterstützte ***Fensterklassen:*** `InputOutput` *(kann Ein- und Ausgaben verarbeiten)* und `InputOnly` *(Kann nur Eingaben verarbeiten)*.

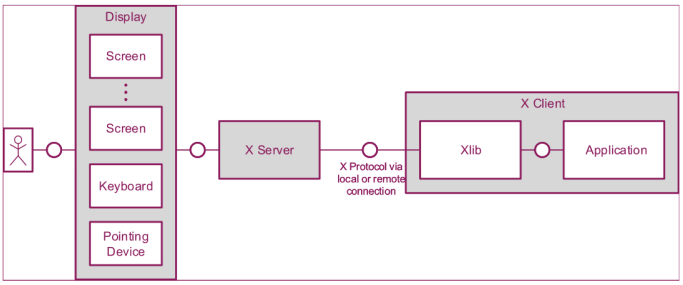
Der Window-Manager ***dekoriert*** Top-level Windows von Applikationen mit Buttons, Scrollbars, Titelleiste usw., indem er hinter jedes Top-level Window ein ***Extrafenster*** legt.

13.1.7. Beispiele Grafischer Desktops basierend auf X

- ***GNOME*** **(Default auf Ubuntu)*:*** verwendet GTK+ *(Fenster-Elemente und Widgets)*, GDK *(Wrapper für Grafikfunktionen)* und GLib *(Allgemeine Datenstrukturen und Algorithmen)*
- ***KDE*** verwendet Qt Toolkit *(Fensterelemente, Widgets, Algorithmen)*

13.2. BASISKONZEPTE DES X WINDOW SYSTEM

- ***Display:*** Rechner mit Tastatur, Zeigegerät und $1..m$ Bildschirme
- ***X Client:*** Applikation, die einen Display nutzen will. Kann lokal oder entfernt laufen.
- ***X Server:*** Softwareteil des X Window System, der ein Display ansteuert. Läuft stets auf dem Rechner, auf dem die GUI-Ein-/Ausgaben anfallen.



13.2.1. Xlib

Ist das ***C Interface*** für das X Protocol. Header wird in C Files eingebunden über `#include <X11/Xlib.h>`. Kompiliertes Executable muss mit X11 Library gelinkt werden: `-lX11`. Hat zahlreiche Funktionen und Datentypen, wird aber meist ***nicht direkt verwendet***, sondern über X-Toolkits, welche eine ***Software-Schicht oberhalb der Xlib*** darstellen. Diese stellen ***Standardbedienelemente*** fertig zur Verfügung, z.B. Command Buttons, Labels und Menüs. *(Xt Toolkit, Tk Toolkit, Motif Toolkit, Open Look Toolkit, GTK+ Toolkit, Qt Toolkit)*

Verbindung zum Display

Um einen Display zu verwenden, muss eine ***Verbindung*** zu diesem bestehen. Die Verbindung wird im ***Datentyp*** `Display` gespeichert.

`Display * XOpenDisplay (char *display_name)` öffnet eine Verbindung zum lokalen oder entfernten Display namens `display_name`. Falls `NULL`, wird der Wert der Umgebungsvariable `DISPLAY` verwendet.

`void XCloseDisplay (Display *display)` schliesst die Verbindung und entfernt die Ressourcen.
Es gibt Funktionen, um bestimmte Eigenschaften des Displays anzuzeigen *(`XDisplayHeight, XDisplayWidth, XRootWindow`)*

Erzeugen von Fenstern

`XCreateSimpleWindow` ist eine einfachere Variante von ***`XCreateWindow`*** mit folgenden Parametern: Display, Parent Window, Koordinaten der oberen linken Ecke, Breite und Höhe, Breite des Rands, Stil des Rands, Stil des Fensterhintergrunds. ***`XDestroyWindow`*** entfernt es und alle seine Unterfenster.

Anzeigen von Fenstern

- `XMapWindow (Display *, Window)` bestimmt, dass ein Fenster auf dem Display angezeigt werden soll. Wird nur angezeigt, wenn ***Elternfenster auch angezeigt*** wird. Teile des Fensters, die von anderen Fenstern ***überdeckt*** werden, werden ***nicht angezeigt***.
- `XMapRaised (Display *, Window)` bringt das Fenster in den Vordergrund.
- `XMapSubwindows (Display *, Window)` zeigt alle Unterfenster an.

Für jedes Fenster, das tatsächlich angezeigt wird, wird ein `Expose` Event erzeugt.

Verstecken von Fenstern

- `XUnmapWindow (Display *, Window)` versteckt ein Fenster und all seine Unterfenster.
- `XUnmapSubwindows (Display *, Window)` versteckt alle Unterfenster eines Fensters.

Für jedes Fenster, das versteckt wird, wird ein `UnmapNotify` Event erzeugt.

13.3. EVENT HANDLING

13.3.1. X Protocol

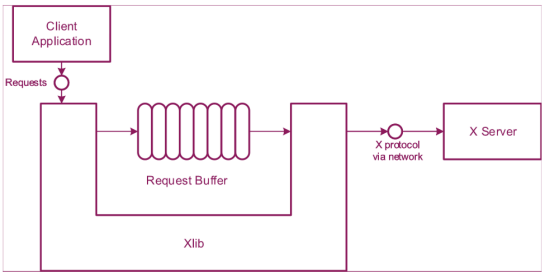
Legt die ***Formate für Nachrichten*** zwischen X Client und Server fest. Es gibt 4 Typen von Nachrichten:

- ***Requests:*** Dienstanforderungen, Client → Server *(«Zeichne eine Linie», «liefere aktuelle Fensterposition»)*
- ***Replies:*** Antworten auf bestimmte Requests, Client ← Server
- ***Events:*** spontane Ereignismeldungen, Client ← Server *(«Mausklick», «Fenstergrösse wurde verändert»)*
- ***Errors:*** Fehlermeldungen auf vorangegangene Requests, Client ← Server

13.3.2. Nachrichtenpufferung bei Requests

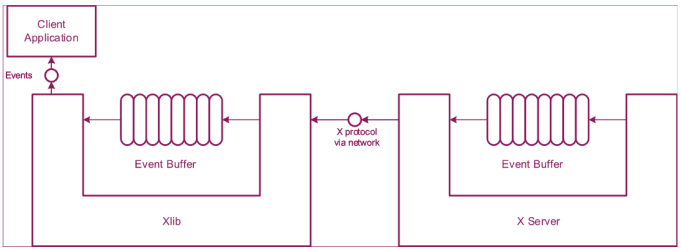
Für ***Requests*** gibt es einen ***Nachrichtenpuffer auf der Client-Seite*** *(Request Buffer)*. ***Ziel:*** möglichst wenige Anforderungsübertragungen an X Server. Gruppierung von Anforderungen für bessere ***Kommunikationseffizienz***.

Übertragung an Server nur, wenn ***sinnvoll oder zwingend nötig***
(Client beginnt auf Event zu warten und blockiert, Client-Request Reply des Servers wird benötigt, Client verlangt explizit Pufferleerung)



13.3.3. Nachrichtenpufferung bei Ereignissen

Ereignisse werden ***doppelt gepuffert***: beim X Server und beim Client. Die ***Server-seitige Pufferung*** berücksichtigt ***Netzwerkverfügbarkeit***, die ***Client-Seitige*** hält ***Events bereit***, bis vom Client abgeholt. Der Client liest Messages im Message-Loop mittels Funktion `XNextEvent()`.



13.3.4. X Event Handling

Ereignisse werden vom ***Client verarbeitet oder weitergeleitet***. Der Client muss vorher festlegen, ***welche Ereignistypen*** er empfangen will *(`XSelectInput()`)*. Die Selektion ist pro Fenster individuell, nur ***selektierte*** Ereignistypen werden dem Client ***zugestellt***. Default: leer. Vom Fenster nicht gewünschte Ereignistypen gehen an das übergeordnete Fenster.

`XSelectInput (Display *display, Window w, long event_mask)` legt als Maske fest, welche Events ausgewählt werden. Masken sind vordefiniert, z.B. `ExposureMask` für `Expose`-Events.

`XNextEvent (Display *display, Event *event)` kopiert den nächsten Event aus dem Buffer in `event`. Die Identifikation des betroffenen Displays und Fenster ist Teil des Events. *(`event.display, event.window`)*

Der Client entscheidet, ***wann*** er ein Event entgegennimmt. Die Verarbeitung der Events erfolgt in einer Programmschleife in der Form:

```
while(1) {
  XNextEvent(display, &event);
  switch (event.type) {
    case Expose: // Typisches Event: verlangt Neuzeichnen des Fensters
      ...
      break;
    case KeyPress: // Event: Taste wurde gedrückt
      ...
      if(...) exit(0);
      break;
  }
}
```

Events sind vom Typ ***`XEvent`***. Dieser Typ ist eine C-Union über alle Event-Typen, d.h. er ist so gross wie der ***grösste Event-Typ***. Es gibt ***33 verschiedene Event-Typen*** unterschiedlicher Grösse. Jeder Event-Typ ist ein ***`struct`***, der als erstes den ***`int type`*** enthält. Der Programm-Code soll anhand von `type` den richtigen ***Union-Member*** verwenden.

13.4. ZEICHNEN

13.4.1. Ressourcen

X Ressourcen sind ***Server-seitige Datenhaltung zur Reduktion des Netzwerkverkehrs***. Sie halten Informationen im Auftrag von Clients. Clients identifizieren Informationen mit zugeordneten ***Nummern*** *(IDs)*. Damit ist ***kein Hin- und Herkopieren*** komplexer Datenstrukturen nötig.

Beispiele von X Ressourcen

- ***Window:*** beschreibt Fenstereigenschaften
- ***Pixmap:*** Rastergrafik *(Verwendung z.B. für Icons, schnelles Neuzeichnen)*
- ***Colormap:*** Farbtabelle *(Setzt Farbindizes in konkrete Farben um)*
- ***Font:*** Beschreibung einer Schriftart
- ***Graphics-Context (GC):*** Grafikelementeigenschaften *(Liniendicke, Farbe, Füllmuster)*. Gleicher GC kann für verschiedene Grafikelemente benutzt werden.

13.4.2. Sichtbarkeit und Aktualisierung von Fensterinhalten

Pufferung verdeckter Fensterinhalte

- ***Minimal:*** keine Pufferung durch X Server, Client muss bei Sichtbarwerden neu zeichnen
- ***Optional:*** X Server hat Hintergrundspeicher zum Sichern der Inhalte abgedeckter Fenster
- Abfragbar von Applikation ob vorhanden mittels `XDoesBackingStore()`

X-Ressource Pixmap

Server-seitiger Grafikspeicher, von Client privat anleg- und nutzbar. ***Anwendung:*** z.B. komplizierte Inhalte in pixmap schreiben, bei Bedarf pixmap in Fenster kopieren *(pixmap wird immer gecached)*.

13.4.3. X Grafikfunktionen

Bilddarstellung mittels ***Rastergrafik*** und ***Farbtabelle*** *(Heute weniger als Tabelle, weil zu viele Farben)*.

- ***Schwarz/Weiss:*** genau ein Bit pro Bildpunkt
- ***Farben oder Grautöne:*** Mehrere Bits pro Bildpunkt. Keine direkte Farbzuordnungen zu Binärwerten, sondern Index in einer Farbtabelle *(color lookup table, colormap)*. Jedes Fenster kann theoretisch eine eigene Farbtabelle benutzen. In der Praxis oft nur eine einzige Farbtabelle für alle Applikationen.

Vorteil Tabelle: Reduktion der Bits pro Farbe von n *(Anzahl Bits pro absolut darstellbarer Farbe)* auf m *(Anzahl Bits pro gleichzeitig darstellbarer Farbe)*. Es sind also statt 2^n nur noch 2^m Farben gleichzeitig darstellbar *(Gilt nur für normale Fensterelemente, Bilder & Videos können ganzen Farbraum nutzen)*.

Grafikgrundfunktionen erlauben das Zeichnen von Geometrischen Figuren, Strings und Texten. ***Ziele*** für das Zeichnen können Fenster oder Pixmap sein.

13.4.4. Graphics Context

Grafikgrundfunktionen ***benötigen einen Graphics Context*** *(X Ressource)*. Legt diverse ***Eigenschaften*** fest, die Systemaufrufe nicht direkt unterstützen *(z.B. Liniendicke, Farben, Füllmuster)*. Client muss erst ***GC anlegen*** vor Aufruf einer Zeichenfunktion. Client kann ***mehrere GCs gleichzeitig*** nutzen. `XCreateGC` legt neuen GC an, `XCopyGC` kopiert GC, `XFreeGC` zerstört GC. `XDefaultGC` gibt den Standard GC für den angegebenen Screen zurück.

13.4.5. Grafik-Primitive (Auswahl)

- ***Einfache Formen:*** `XDrawPoint(s)`, `XDrawLine(s)`, `XDrawSegments`, `XDrawRectangle(s)`, `XFillRectangle(s)`, `XDrawArc(s)`, `XFillArcs`
- ***Text:*** `XDrawString`, `XDrawText`
- ***Bilder:*** `XPutImage`

13.5. FENSTER SCHLIESSEN

Die Schaltfläche zum Schliessen eines Fensters wird vom ***Window Manager*** erzeugt. X weiss ***nichts*** über die spezielle Bedeutung dieser Schaltfläche, der ***Window Manager schliesst*** das Fenster. Damit die Applikation davon erfährt, gibt es ein ***Protokoll*** zwischen Window Manager und der Applikation. Der Window Manager sendet ein ***`ClientMessage` Event*** an die Applikation. Dieses Event muss in seinem `data`-Teil die ID eines Properties `WM_DELETE_MESSAGE` enthalten.

13.5.1. Atoms

Ein Atom ist die ***ID eines Strings***, der für ***Meta-Zwecke*** benötigt wird. Anstelle der Strings verwendet man stattdessen die entsprechenden Atoms. Das ***erspart*** das ständige Parsen der Strings *(z.B. statt `WM_DELETE_MESSAGE` verwendet man intern Atom 5)*.

`Atom XInternAtom (Display *, char *, Bool only_if_exists)` ***übersetzt*** den String in ein Atom auf dem angegeben Display. `only_if_exists` gibt an, ob das Atom erzeugt werden soll, wenn es nicht existiert *(`only_if_exists = false`)*.

13.5.2. Properties

Mit jedem Fenster können ***Properties*** assoziiert werden. Der Window Manager ***liest und/oder setzt*** diese Properties. ***Generischer Kommunikations-Mechanismus*** zwischen Applikation und Window Manager. Eine Property wird über ein Atom ***identifiziert***. Zu jedem Property gehören ***spezifische Daten*** *(z.B. ein oder mehrere Strings oder eine Liste von Atomen).*

`WM_PROTOCOLS`

Der X Standard definiert eine ***Anzahl an Protokollen***, die der Window Manager verstehen soll. Ein Client kann sich für Protokolle ***registrieren***. Dazu muss er im Property `WM_PROTOCOLS` die Liste der Atome der Protokollnamen speichern. Das geschieht mit der Funktion `XSetWMProtocols`.

`XSetWMProtocols (Display *, Window, Atom *first_proto, int count)` speichert im Property `WM_PROTOCOLS` die Atome aus dem Array, das an `first_proto` beginnt und `count` Elemente enthält.

`WM_DELETE_WINDOW`

Der Window Manager schickt ein Event an den Client, wenn man auf den Close-Button drückt. Der Event-Typ ist `ClientMessage`, wird immer vom Client verarbeitet. Im Datenteil des Events steht das Atom von `WM_DELETE_WINDOW`.

Registrierung des Clients	*Verarbeiten des Events*

`Atom atom = XInternAtom (display,`
`  "WM_DELETE_WINDOW",`
`  /* only_if_exists = */ false);`
`XSetWMProtocols (`
`  display, window, &atom, 1);`
	`switch (event.type) {`
	`  case ClientMessage:`
	`  if (event.client→data.l[0] == atom) { ... }`
	`  break;`
	`}`

14. MELTDOWN

Meltdown ist eine ***Hardware-Sicherheitslücke***, mit der der ***gesamte physische Hauptspeicher ausgelesen*** werden kann. Insbesondere kann damit ein Prozess alle geheimen Informationen ***anderer*** Prozesse lesen.

Folgende ***Eigenschaften*** müssen für diese Sicherheitslücke gegeben sein:

Der Prozessor muss dazu gebracht werden können:
1. aus dem ***geschützten Speicher*** an Adresse a das Byte m_a zu ***lesen***
2. die Information m_a in irgendeiner Form f_a ***zwischenzuspeichern***
3. ***binäre Fragen*** der Form «$f_a \stackrel{?}{=} i$» zu beantworten *(Liegt f_a an Index i?)*
4. Von $i = 0$ bis $i = 255$ ***iterieren***: $f_a \stackrel{?}{=} i$
5. Über alle a ***iterieren***

14.1. PERFORMANCE-OPTIMIERUNGEN IN REALEN SYSTEMEN

Moderne HW und OS verwenden zahlreiche und nicht immer intuitive «Tricks» für Performance-Optimierung: ***Caches***, ***Out-of-Order Execution***, ***Spekulative Ausführung***, ***Mapping*** des gesamten physischen Speichers in jeden virtuellen Adressraum.

14.1.1. Mapping des Speichers in jeden virtuellen Adressraum

Virtueller Speicher soll Prozesse gegeneinander schützen. Deshalb hat jeder Prozess seinen eigenen virtuellen Adressraum. ***Kontext-Wechsel*** sind jedoch relativ ***teuer***. Deshalb arbeitet das OS aus ***Performance-Gründen*** im ***Kontext des Prozesses***. Der OS-Kernel wechselt den Kontext ***nicht***, sondern ***mappt alle Kernel-Daten*** in den Adressraum.
Die Page-Table ist so konfiguriert, dass ***nur das OS*** auf diese Teile ***zugreifen*** darf.

Da das OS auf ***alle Prozesse*** zugreifen können muss, mappt der OS-Kernel den ***gesamten physischen Hauptspeicher*** in jeden virtuellen Adressraum.

14.1.2. Out-of-Order Execution (O3E)

Moderne Prozessoren führen Befehle aus, wenn alle benötigten Daten zur Verfügung stehen *(Solange das Endergebnis dadurch nicht beeinträchtigt wird)*. Dadurch kann sich ***die Reihenfolge der Befehle ändern***.

Spekulative Ausführung

Out-of-Order Execution wird auch dann ***vorgenommen***, wenn der Befehl später ***gar nicht ausgeführt*** wird. Erfordert prozessor-internen Zustand neben den Registern - ***wesentliche Voraussetzung für Geschwindigkeit*** moderner Prozessoren. Wenn der Wert dann nicht gebraucht wird, wird er wieder ***verworfen*** *(z.B. Befehle nach conditional jumps)*.
Beim Zugriff auf eine Adresse, für die keine Berechtigung besteht, liest das OS den Wert zwar, gibt ihn aber nicht an den Prozess weiter. Schritt 1 ist damit erfüllt, da durch spekulative Ausführung auf jeglichen Speicher zugegriffen werden kann.

Seiteneffekte von Out-of-Order Execution

Der ***Cache weiss nicht***, ob ein Wert ***spekulativ*** angefordert wurde. Wird also trotzdem in den Cache geschrieben. Dieser kann vom Prozess nicht ausgelesen werden, da die MMU entscheidet, ob der Cache die Daten an ihn weitergeben darf.

```
char dummy_array[4069 * 256]; // page size * char size
char * sec_addr = 0x1234;
char sec_data = *sec_addr; // Exception, no permission
char dummy = dummy_array[sec_data]; // speculative
```

Schritt 2 ist damit erfüllt, da im Cache m_a als Teil des Tags gespeichert wird.
(`dummy_array` + `sec_data` → `sec_data` = Tag - `dummy_array`).

14.1.3. Cache auslesen

Jedoch kann ***die Zeit gemessen*** werden, die ein Speicherzugriff benötigt: ***Lange Zugriffszeit:*** Adresse war nicht im Cache, ***Kurze Zugriffszeit:*** Adresse war im Cache. Dies nennt man auch ***Timing Side Channel Attack***. Mit der Assembly-Instruktion `clflush p` werden alle Zeilen, die die Adresse `p` enthalten gelöscht. Das ermöglicht «Flush & Reload»: Über das gesamte Array iterieren und `clflush` ausführen, damit wird sichergestellt dass das komplette Array nicht im Cache ist. Schritt 3 ist somit auch erfüllt: Die Zugriffszeit verrät, ob f_a im Cache.

14.2. TESTS VON MELTDOWN

Verschiedene CPUs *(Intel, einige ARMs, keine AMDs)* und verschiedene OS *(Linux, Windows 10)* sind betroffen. ***Geschwindigkeit*** bis zu 500 KB pro Sekunde bei 0.02% Fehlerrate. So schnell können «sichere» Daten ausgelesen werden. *(1 GB in 35min mit 210KB Fehlern)*

14.3. EINSATZ VON MELTDOWN

Meltdown kann zum Beispiel zum ***Auslesen von Passwörtern*** aus dem Browser via Malware oder für ***Zugriff auf andere Dockerimages*** auf dem gleichen Host verwendet werden. Kann jedoch ***nicht*** aus einer VM heraus auf den Host oder auf geschlossene Systeme zugreifen.

Nachweis des Einsatzes ist sehr schwierig, die Attacke hinterlässt quasi ***keine Spuren***. ***AMD und ARM sind nicht betroffen***, vermutlich weil sie die Zugriff-Checks anders durchführen.

14.4. GEGENMASSNAHMEN

Kernel page-table isolation «KAISER»: verschiedene Page Tables für Kernel- bzw. User-Mode. Der ***Impact auf die Performance*** ist auf Linux-Systemen sehr unterschiedlich, kaum messbar bei Computerspielen, 5% beim Kompilieren, bis zu 20% bei Datenbanken und ***30% bei weiteren Anwendungen***.

14.5. SPECTRE

Angriff, der das gleiche Ziel hat wie Meltdown, nämlich Speicherbereiche anderer Prozesse zu lesen. Verwendet jedoch einen anderen Mechanismus: ***Branch Prediction mit spekulativer Ausführung***.

Moderne Prozessoren ***lernen*** über die Zeit, ***ob ein bedingter Sprung erfolgt*** oder nicht. Muss der Prozessor noch auf die Sprungbedingung warten, kann er schon ***spekulativ den Zweig ausführen, den er für wahrscheinlicher hält***.

14.5.1. Angriffsfläche von Branch Prediction

Branch Prediction wird nicht per Prozess ***unterschieden***. Alle Prozesse, die auf dem selben Prozessor laufen, verwenden die ***selben Vorhersagen***. Ein Angreifer kann damit den Branch Predictor ***für einen anderen Prozess «trainieren»***. Der ***Opfer-Prozess*** muss zur Kooperation ***«gezwungen»*** werden, indem im verworfenen Branch auf Speicher zugegriffen wird.

Spectre ist ***nicht besonders leicht zu fassen***, aber auch ***nicht besonders leicht zu implementieren***.

14.5.2. Fazit

HW-Probleme können teilweise durch SW ***kompensiert*** werden, Designentscheidungen können ***weitreichende Konsequenzen*** haben.