# grundlagen

**Aufgaben eines OS:**

* Abstraktion (von Hardware/Protokollen/Software-Services) und damit Portabilität
* Resourcenmanagement und Isolation von Anwendungen voneineander
* Benutzerverwaltung und Sicherheit

**Grenzen der Portierbarkeit:** OS kann nicht entscheiden, was Applikation meint. (z.B Maus vs Touchscreen oder FullHD vs UltraHD). Applikation muss dafür sorgen, dass diese auf verschiedenen Geräten korrekt benutzbar ist.

**Grenzen der Isolierbarkeit:** Applikationen, die auf einem Bildschirm laufen, konkurrieren um Bildschirm und Tastatur (z.B Applikationen stehlen über Popup Eingabefokus)

**CPU Privilege Levels:** Moderne OS benötigen CPU mit mindestens 2 Privilege Levels. OS läuft im Kernel Mode und kann bestimmen, welche Software in welchem Modus läuft.

* Kernel-Mode: Darf jede Instruktion ausführen
* User-Mode: Darf nur eine beschränkte Menge an Instruktionen ausführen

|  |  |  |
| --- | --- | --- |
| Kernmodul 1 | OS-Modul A | Applikation a |
| Kernmodul 2 | OS-Modul B | Applikation B |
| … | … | … |
| Kernel |  |  |
|  | Betriebssystem |  |

**Microkernel:** Kernelfunktionalität reduziert auf ein Minimum. Selbst Gerätetreiber laufen im User-Mode. Nur kritische Teile des Kernels sind im Kernel-Mode. Vorteile sind Stabilität und Analysierbarkeit, dafür gibt es Performance-Einbussen durch häufige Mode-Wechsel.

**Monolithische Kernel:** Die meisten OS-Kernel sind monolithisch (viel Funktionalität, die auch im User-Mode laufen könnten). Vorteil ist bessere Performance, da weniger zwichen Modes gewechselt wird, dafür weniger Schutz vor Programmierfehlern (Kleine Fehler können über Modulgrenzen propagieren und nicht nachvollziehbare Auswirkungen haben)

**Unikernel:** Ein «normales» Programm als Kernel. Kernelfunktionalität ist in einer Library und es gibt keine Trennung zwischen Kernel- und User-Mode. Vorteile sind echte Minimalität und extrem kompakt, dafür Single Purpose, da nur ein Programm je laufen wird

**User-Mode zu Kernel-Mode:** Wenn man direkt vom User zum Kernel Mode wechseln könnte, könnte jede Applikation nach Belieben in den Kernel Mode wechseln. Stattdessen muss man auf Intel 64 Prozessoren im User Mode eine syscall Instruktion ausführen. Diese Instruktion schaltet in den Kernel Mode, damit im Kernel Mode immer Kernel-Code läuft.

**ABI vs API:**

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

**ABI-Binärkompatibilität verschiedener Linux Kernel:** Auf Linux gilt diese Calling Convention nur für jeweils eine spezifische Kernel-Version Linux-Kernels sind nicht binär-kompatibel zueinander, d.h. Applikationen müssen jeweils für einen Kernel kompiliert werden. Z.B Kernel A: exit hat Code 60, Kernel B: exit hat Code 55, etc.

**API-Kompatibilität der Linux-Kernel für Applikationen:** Applikationen sollen nicht direkt Syscalls aufrufen, sondern C-Wrapper-Funktionen. Generiert mit Compiler und C-Standard-Library immer den zum Kernel passende Binär-Code, ohne C-Code der Applikation anpassen zu müssen. Linux-Kernel sind API-kompatibel.

**POSIX:** Grundsätzlich hat jedes OS seine eigene API und ABI. POSIX ist eine Sammlung von IEEE Standard, welche die Kombatibilität zwischen OS gewährleistet.

**POSIX-Konformität:** Zahlreiche POSIX-konforme OS, aber nicht jedes OS muss POSIX-konform sein. macOS ist POSIX-konform, Linux mehr oder weniger und Windows nicht.

**Shell:** Die Shell ist ein Programm, das es erlaubt, über Texteingabe Betriebssystemfunktionen aufzurufen. Es gibt viele verschiedene Shells mit jeweils eigener Syntax. Die Shell benötigt keine besonderen Rechte oder spezielle Vorkehrungen. Die Shell benötigt lediglich einen Ausgabe-Stream und einen Eingabe-Stream.

## programmargumente

Wird ein Programm gestartet, kann es Programmargumente erhalten, z.B. clang -c abc.c -o abc.o. Shell. Shell teilt Programmargumente in Strings auf. Fast alle Shells verwenden Leerzeichen als Trennung zwischen Programmargumenten. Das OS interessiert sich nicht für den Inhalt der Argumente.

**Calling Convention:** Beim Start schreibt das OS diese Programmargumente als null-terminierte Strings in den Speicherbereich des Programms. Zusätzlich legt das OS ein Array an, dessen Elemente jeweils auf das erste Zeichen eines Programmarguments zeigen. Der Pointer auf dieses Array und die Anzahl der Elemente wird dem Progamm an einer vom OS definierten Stelle zur Verfügung gestellt, z.B. in Registern oder auf dem Stack

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| argv\ |  | argv[0] |  | Programmname |
|  |  | argv[1] |  | 1. Argument |
|  |  | argv[2] |  | 2. Argument |
|  |  | ... |  |  |
|  |  | argv[argc - 1] |  |  |
|  |  | 0 |  |  |

**Programmargumente in C:** In C wird dieser Umstand durch die beiden Parameter der Funktion main ausgedrückt. int argc enthält die Anzahl der Programmargumente (argument count) + 1. char \*\*argv enthält den Pointer auf das Array (argument values). Achtung: argv[0] ist der Programmname, die Argumente selbst folgen als argv[1] bis argv[argc - 1].

## umgebungsvariablen

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

PATH=/home/ost/bin:/home/ost/.local/bin

Teilstring vor = ist Key, danach ist Value. Jeden Key kann es höchstens einmal geben. Unter POSIX verwaltet das OS die Umgebungsvariablen innerhalb jedes laufenden Prozesses. Umgebungsvariablen werden initial vom erzeugenden Prozess festgelegt, also z.B. der Shell.

**Environ:** In C zeigt die Variable environ auf das Array von Umgebungsvariablen. Jedoch sollte environ nicht direkt verwendet werden, sondern über vordefinierte Funktionen.

**Getenv:**

* Gibt die Adresse des ersten Zeichens des entsprechenden Values zurück.
* Gibt 0 zurück, wenn keine Umgebungsvariable mit diesem Key vorhanden.

char \* getenv (const char \* key)

char \*value = getenv("PATH");

// value = "/home/ost/bin:/home/ost/.local/bin"

**Setenv:**

* Wenn key schon in einer Umgebungsvariable v enthalten ist und overwrite != 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, sonst Fehlercode in errno

int setenv(const char \*key, const char \*value, int ow);

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

**Unsetenv:**

* Entfernt die Umgebungsvariable mit dem Key key
* Gibt 0 zurück, wenn alles OK, sonst Fehlercode in errno

int unsetenv(const char \*key);

int ret = unsetenv("HOME");

**Putenv:**

* Fügt den Pointer kvp dem Array der Umgebungsvariablen hinzu. Der String, auf den kvp zeigt, wird nicht kopiert
* Wenn der Key schon vorhanden ist, wird der String gelöscht, auf den der existierende Pointer zeigt
* Gibt 0 zurück, wenn alles OK, sonst Fehlercode

int putenv (char \* kvp)

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

**Zweck von Programmargumente:** Programmargumente werden bei jedem Programmaufruf explizit angegeben. Das ist nützlich für Informationen, die bei jedem Aufruf typischerweise anders sind, z.B. die Datei, die kompiliert werden soll.

**Zweck von Umgebungsvariablen:** Umgebungsvariablen werden dem Programm implizit bereitgestellt. Das ist nützlich, wenn Informationen normalerweise über alle Aufrufe gleich sind, z.B. die Pfade, in denen das System nach Hilfsprogrammen und Bibliotheken suchen soll (PATH)

# dateisystem

**Daten:** Sequenz von Bytes, die als Ganzes im Dateisystem abgelegt ist. Inhalt oder interne Struktur ist Dateisystem egal. Anwendungen haben entweder keinen oder vollen Zugriff

**Metadaten:** Sammlung von Attributen (Informationen über die Datei). Teilweise sichtbar für Nutzer, z.B. Dateiname, Grösse, Verzeichnis, Datum des Erstellens. Teilweise nicht, z.B. Ablageort auf Datenträger, Verkettung von Blöcken auf Datenträger

**Dateitypen:** Dateiendungen sind die Zeichen nach dem letzten Punkt, jedoch für FS/OS fast keine Relevanz. Keine besondere Behandlung von Dateien in Abhängigkeit der Endung. Typ wird häufig durch Magic Numbers/Strings innerhalb der Datei gekennzeichnet.

**Falsche Datentypen:** Applikation muss den Dateityp richtig zu bestimmen und selbst gegen Fehlinterpretation schützen. Immer Daten validieren und auf Grenzverletzungen überprüfen

**Verzeichnis:** Liste von Dateien oder weiteren Verzeichnisse. Als Datei realisiert, die diese Liste enthält und hat einen Dateinamen.

**Verzeichnishierarchie:** Gesamtheit aller Verzeichnisse im System in Baum-Hierarchie. Jedes Verzeichnis (ausser Wurzelverzeichnis) hat genau ein Elternverzeichnis.

**Wurzelverzeichnis:** Oberstes Verzeichnis in der Hierarchie. Hat keinen Namen, wird aber oft mit / bezeichnet

**Besondere Verzeichnisse:** Jedes Verzeichnis enthält zwei implizite Referenzen auf Verzeichnisse: . Referenz auf sich selbst und .. Referenz auf das Elternverzeichnis.

**Pfade:** Ein Pfad spezifiziert eine Datei oder ein Verzeichnis in der Verzeichnishierarchie. Verzeichnisnamen werden durch / voneinander getrennt. Pfade können minimal 14 und maximal 256 Zeichen lang sein.

**Absolute vs Relative vs Kanonische Pfade:**

|  |  |  |
| --- | --- | --- |
| Absolut | Relativ | Kanonisch |
| Beginnt mit / | Beginnt nicht mit / | Pfad ohne . und .. |
| Bezug auf Wurzelverzeichnis | Bezug auf Arbeitsverzeichnis |  |

**Zugriffsrechte:** Jeder Datei und jedem Verzeichnis sind Zugriffsrechte zugeordnet und gehören genau einem Benutzer (Owner) und genau einer Gruppe. Je 3 Bits für Owner, Gruppe und alle anderen Benutzer.

## posix file api

POSIX-API bietet direkten Zugriff auf Daten als rohe Binärdaten (so wie sie in der Datei gespeichert sind). Erlaubt direkten, unformatierten Zugriff auf Inhalt der Datei. Sollte nur für Binärdateien verwendet werden (nicht für Verzeichnisse oder Textdateien).

**errno:** Makro oder globale Variable vom Typ int. Wird von vielen Funktionen gesetzt. Sollte unmittelbar nach Auftreten eines Fehlers aufgerufen werden.

**File-Descriptor:** Dateien werden in der POSIX-API über File-Deskriptoren (FD) repräsentiert. Gilt immer nur innerhalb eines Prozesses. FD ist zustandsbehaftet, d.h merkt sich aktuellen Offset. Index in eine Tabelle aller geöffneten Dateien im Prozess. Tabelleneintrag enthält Index in systemweite Tabelle aller geöffneten Dateien. Enthält Daten, um physische Datei (mit richtigem Treiber, Datenträger, etc) zu identifizieren. In jedem Prozess gibt es 3 vordefinierte File Deskriptoren.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | File Table of Process |  | Global File Table |
|  |  | standard in (0) |  |  |
| File descriptor |  | standard out (1) |  |  |
|  |  | standard error (2) |  |  |
|  |  |  |  |  |

**open:** Erzeugt einen FD auf die Datei, die an path liegt. Flags gibt an, wie Datei geöffnet werden soll (O\_RDONLY, O\_RDWR, O\_CREAT, O\_APPEND, O\_TRUNC). Permission (S\_IRWXU and many more)

int open (char \*path, int flags, int permission)  
open(path2, O\_WRONLY | O\_CREAT, S\_IRWXU);

**close:** Dealloziert FD, kann später von open für andere Datei verwendet werden (0 = Ok)

int close (int fd)

**read:** Versucht die nächsten n Byte am aktuellen Offset von fd in den buffer zu kopieren. Gibt Anzahl gelesener Bytes zurück oder -1 bei Fehler und erhöht Offset um Anzahl gelesener Bytes

ssize\_t read (int fd, void \*buffer, size\_t n)

**write:** Versucht die nächsten n Byte vom buffer an den aktuellen Offset von fd zu kopieren. Gibt Anzahl geschriebener Bytes zurück oder -1 bei Fehler und erhöht Offset um Anzahl geschriebener Bytes

ssize\_t write (int fd, void \*buffer, size\_t n)

**lseek:** Setzt den Offset von FD auf Offset. Origin gibt an, wozu Offset relativ ist. SEEK\_SET: Beginn der Datei, SEEK\_CUR: Aktueller Offset und SEEK\_END: Ende der Datei

off\_t lseek (int fd, off\_t offset, int origin)

lseek (fd, 0, SEEK\_CUR); // Gibt aktuellen Offset zurück

lseek (fd, 0, SEEK\_END); // Gibt Dateigrösse zurück

lseek (fd, n, SEEK\_END); // Hängt bei nachfolgendem write n Nullen an Datei

**Lesen/Schreiben ohne Offset:** Wie read/write, aber Funktionsaufruf mit pread und pwrite. Statt des Offsets von fd wird der zusätzliche Parameter offset verwendet. Der Offset von FD wird nicht verändert.

**Beispiel:**

int testdesc = open(path1, O\_RDONLY);

int testdesc2 = open(path2, O\_WRONLY | O\_CREAT, S\_IRWXU);

while (1) {

  ssize\_t bytes1 = read(testdesc, buf1, len);

  write(testdesc2, buf1, bytes1);

  if (bytes1 < len) {

    break;

  }

}

close(testdesc); close(testdesc2);

## c stream api

Unabhängig vom OS: für POSIX und Windows gleich. Kann gepuffert oder ungepuffert sein (Abhängig von der Implementierung). Hat einen eigenen File-Position-Indicator. Bei gepufferten Streams: bestimmt Position im Puffer (Standard bei Dateien). Bei ungepufferten Streams: entspricht dem Offset des File-Descriptors.

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

**fopen:** Erzeugt FILE-Objekt (und damit Stream) für Datei an path. Mode gibt Flags analog zu open als nullterminierten String an (r, w, a, r+, w+, a+). Gibt Pointer auf erzeugtes FILE-Objekt zurück oder 0 bei Fehler

FILE\* fopen (char const \*path, char const \*mode)

**fclose:** Schliesst den durch file bezeichneten Stream und entfernt file aus Speicher. Ruft fflush auf. Gibt 0 zurück, wenn OK, andernfalls EOF

int fclose (FILE \*file)

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

int fflush (FILE \*file)

**fdopen:** Wie fopen, aber statt Pfad wird direkt der File-Deskriptor übergeben

FILE \* fdopen (int fd, char const \*mode)

**fileno:** Gibt File-Descriptor zurück, auf den sich der Stream bezieht, oder -1 bei Fehler

int fileno (FILE \*stream)

**fgetc:** Liest das nächste Byte vom stream als unsigned char und gibt es als int zurück

int fgetc (FILE \*stream)

**fgets:** Liest bis zu n - 1 Zeichen aus stream, bis Newline oder EOF auftritt. Hängt eine 0 an, und erzeugt damit null-terminierten String

char \* fgets (char \*buf, int n, FILE \*stream)

**ungetc:** 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 (keine Schreiboperation)

int ungetc (int c, FILE \*stream)

**fputc:** Konvertiert c in unsigned char und schreibt diesen auf stream

int fputc (int c, FILE \*stream)

**fputs:** Schreibt die Zeichen vom String s bis zur terminierenden 0 in stream. Die terminierende 0 wird nicht mitgeschrieben

int fputs (char \*s, FILE \*stream)

**feof:** Gibt genau dann 0 zurück, wenn Dateiende noch nicht erreicht wurde

int feof (FILE \*stream)

**ferror:** Gibt genau dann 0 zurück, wenn kein Fehler auftrat

int ferror (FILE \*stream)

**ftell:** Gibt den gegenwärtigen File-Position-Indicator zurück. POSIX-Erweiterung von C: ftello mit Rückgabetyp off\_t

long ftell (FILE \*stream)

**fseek:** Setzt den File-Position-Indicator, analog zu lseek. POSIX-Erweiterung von C: fseeko mit off\_t als Typ für offset

int fseek (FILE \*stream, long offset, int origin)

**rewind:** Setzt den Stream zurück. Äquivalent zu fseek(stream, 0, SEEK\_SET) und Löschen des Fehlerzustands

int rewind (FILE \*stream)

**FILE-Datenstruktur:** Vordefinierte File-Deskriptoren (\*stdin, \*stdout, \*stderr)

fputs("go watch https://www.youtube.com/@NinomaeInanis ", stdout);

**Beispiel:** Read from f1 and write to f2 menacingly.

FILE \*f1 = fopen(path1, "r");

FILE \*f2 = fopen(path2, "w");

while (feof(f1) == 0) {

    char \*bytes = fgets(buf, len, f1);

    int bytes2 = fputs(buf, f2);

}

fclose(f1); fclose(f2);

# prozesse

**Prozesse:** Ein Prozess umfasst das Abbild eines Programms im Hauptspeicher (text section), die globalen Variablen des Programms (data section), Speicher für den Heap und Stack.

**Prozess vs Programm:** Programm ist passiv: beschreibt bestimmte Abläufe. Prozess ist aktiv: führt die Abläufe aus. Ein Programm kann mehrfach ausgeführt werden: verschiedene, voneinander unabhängige Prozesse.

**Process Control Block (PCB):** OS hält Daten über jeden Prozess in einem Process Control BlockSpeicher für alle Daten, die das OS benötigt, um die Ausführung des Prozesses ins Gesamtsystem zu integrieren, u.a Eigene/Parent ID und andere wichtige IDs, Speicher für Zustand des Prozessors (Kontext), Scheduling-Info etc.

**Interrupts:** Wenn ein Interrupt auftritt, muss der Kontext des aktuellen Prozesses im dazugehörigen PCB gesichert werden (context save). Register, Flags, Instruction Pointer, MMU-Konfiguration werden gespeichert. Nach dem Ende des Interrupt-Handlers wird der Kontext des Prozesses aus seinem PCB wiederhergestellt (context restore)

**Kontext-Wechsel:** Wenn Prozessor von Prozess A auf B umschalten will, muss das OS einen Kontext-Wechsel durchführen. OS sichert dazu den Kontext von Prozess A im PCB A (bspw. während eines Timer-Interrupts). Danach stellt das OS den Kontext von Prozess B aus dem PCB B wieder her. Nach Rücksprung aus dem Interrupt-Handler läuft somit Prozess B stattA

**Erzeugen eines Prozesses:** Um aus einem Programm einen Prozess zu machen, muss das OS einen Prozess erzeugen und ein Programm in diesen Prozess laden und in einen laufbereiten Zustand versetzen. In POSIX sind dies 2 Schritte, in Windows werden beide Schritte in einer Funktion ausgeführt.

**Prozess Hierarchie:** In POSIX hat jeder Prozess ausser Prozess 1 genau einen Parent-Prozess. Jeder Prozess kann beliebig viele Child-Prozesse haben (Baumstruktur).

**Fork:** Erzeugt eine exakte Kopie (Child C) des Prozesses (Parent P), aber hat eine eigene Prozess-ID und hat als Parent-ID die Prozess-ID von Parent. Führt in beiden Prozessen den Code an derselben Stelle fort: Am Rücksprung aus fork.

pid\_t fork (void)

**Example:**

pid\_t new\_pid = fork();

if (new\_pid > 0) { // Parent Code }

else if (new\_pid == 0) { // Child Code }

**Exit:** Entspricht dem gleichnamigen OS-Aufruf. Kann an jeder Stelle im Programm verwendet werden und bietet somit eine Alternative zum «Rücksprung» aus main.

void exit (int code)

**Wait:** Unterbricht aufrufenden Prozess, bis einer seiner Child-Prozesse beendet wurde. Gibt die Statusinformationen über den int zurück, auf den status zeigt (Out-Parameter)

* WIFEXITED(\*status): ̸= 0 («true»), wenn Child ordnungsgemäss beendet wurde
* WEXITSTATUS(\*status): Exit-Code von Child, etc.
* Gibt -1 bei Fehler zurück, Fehlercode in errno

pid\_t wait (int \*status)

**waitpid:** 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 und pid < -1 ermöglichen das Warten auf Prozesse einer bestimmten Prozessgruppe

pid\_t waitpid (pid\_t pid, int \*status, int options)

**Zusammenspiel von fork und wait:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | Parent: pid > 0 | |  |  |
| Parent | pid = fork(); |  | | wait(); | Parent |
|  |  | |  |  | |
|  | Child: pid == 0 | | exit | OS | |

**Beispiel:** Mehrere Worker-Prozesse

for (int i = 0; i < n; ++i)  {

    spawn\_worker (...);

}

// ... do something in parent process

do {

    pid = wait (0);

} while (pid > 0 || errno != ECHILD);

**Exec:** exec-Funktion ersetzen im gerade laufenden Prozess das Programmimage durch ein anderes Programmimage.Bei jeder exec-Funktion müssen die Argumente spezifiziert werden

* Bei den execl\*-Funktionen als Liste: execl(path, arg0, arg1, ...)
* Bei den execv\*-Funktionen als Vektor (Array): execv(path, argv)
* Die exec\*e-Funktionen erlauben die Angabe eines Arrays für die Umgebungsvariablen, in den anderen Versionen bleiben die Umgebungsvariablen
* Die exec\*p-Funktionen suchen den Dateinamen über die Umgebungsvariable PATH, die anderen verwenden absolute/relative Pfade

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Programm A | | | exec(“B”, …) | | | Programm B | | | |
| Prozess 1 | | | Prozess 1 | | | |
|  | |  | | | Programmargumente | | | |
| als Liste | | als Array | |
| Angabe des Pfads | | mit neuem Env | | | execle | | execve | |
| mit altem Env | | | execl | | execv | |
| Suche über PATH | | execlp | | execvp | |
|  |  | | Parent: pid > 0 | | |  | |  | |
| Parent | pid = fork(); | |  | | | wait(); | | Parent | |
|  |  | | |  | | OS | | | |
|  | Child: pid == 0 | | | exec\*() | | exit | |  | |

**Zombie Prozess:** Wenn ein Prozess C beendet wird, ist Parent-Prozess P verantwortlich dafür, auf jeden Fall wait aufzurufen. Das OS weiss nicht, wann das passieren wird und muss Statusinformationen von C solange vorhalten, bis P wait aufruft.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | Parent: pid > 0 | |  |  |
| Parent | pid = fork(); |  | | wait(); | Parent |
|  |  | |  |  | |
|  | Child: pid == 0 | | exit | Zombie bis Parent wait aufruft | |

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, denn er scheint C vergessen zu haben. Die Situation kann bereinigt werden, in dem P gestoppt wird

**Orphan Prozess:** Wird ein Prozess P beendet, haben seine Child-Prozesse C keinen Parent-Prozess mehr. Beim Ende eines Prozesses werden alle 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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | Parent |  | Prozess 1 | |  |
| Parent | pid = fork(); |  | exit |  | |  |
|  |  |  |  | wait(); | | Prozess 1 |
|  | Child |  |  |  | OS |  |
|  |  |  |  | exit | |  |

**Sleep:** Unterbricht die Ausführung, bis die Anzahl der Sekunden ungefähr verstrichen ist. Kann vom System auch unterbrochen werden (z.B. durch Signale). Gibt die Anzahl Sekunden zurück, die vom Schlaf noch verblieben sind

unsigned int sleep (unsigned int seconds)

**atexit:** Das OS kann viele Ressourcen automatisch nach Beendigung aufräumen, z.B. verwendeten Speicher oder offene Dateien**.** Aber nicht immer: z.B. Geräte in einen bestimmten Zustand bringen, andere Prozesse benachrichtigen, oder eine Remote-Session schliessen. Damit ein Programm trotzdem noch kurz vor dem wirklichen Ende letzte Aufräumarbeiten durchführen kann, kann es eine oder mehrere Funktionen mit atexit registrieren. Diese Funktionen werden dann nach einem Aufruf von exit (bzw. nach dem Rücksprung aus main) in umgekehrter Reihenfolge der Registrierung aufgerufen.

int atexit(void (\*function)(void))

**PID lesen:** Gibt PID bzw. Parent PID zurück

pid\_t getpid(void); pid\_t getppid(void);

# programme/bibliotheken

## pengos loader/elf

**Loader:** Loader lädt Executables und dynamische Bibliotheken (keine statische) in den Hauptspeicher. Executables und dynamische Bibliotheken müssen also alle Informationen enthalten, die der Loader benötigt.

**Program Loader:** Funktionen exec\* enthalten syscall auf sys\_execve. Sucht Datei, prüft Rechte (x-Bits) und öffnet die spezifizierte Datei. Zählt und kopiert die Argumente und Umgebungsvariablen. Übergibt den Request an jeden registrierten «Binary Handler». Binary Handler versuchen nacheinander jeweils die Datei zu laden und zu interpretieren.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Storage |  |  |  | Main Memory |
| Executable |  | Loader |  | Process Image |

**Executable and Linking Format (ELF):** Binär-Format, das Kompilate spezifiziert (a.out für Unix und .exe für Windows). Eigentlich 2 Formate Linking und Execution View. Object Files verwenden Linking View und Programme Execution Views. Shared Objects beide.

**ELF Struktur:** Besteht aus Header, Program Header Table (nur in EV erforderlich), Segmente (nur in EV erforderlich), Section Header Table (nur in LV erforderlich) und Sektionen (nur in LV erforderlich)

|  |  |  |  |
| --- | --- | --- | --- |
| ELF Header | Program Header Table | Section and Segments | Section Header Table |

* ELF Header (52B): Beschreibt die Struktur der Datei. Relozierbar, Ausführbar, Shared Object. 32-bit oder 64-bit.
* Program Header Table: Tabelle mit n Einträgen. Jeder Eintrag (32Bit) und beschreibt ein Segment (Segment-Typ und Flags, Offset und Grösse der Datei, virtuelle Adresse und Grösse im Speicher)
* Section Header Table: Tabelle mit m Einträgen. Jeder Eintrag (40B) beschreibt eine Sektion (Name, Typ, Flags, Offset und Grösse in der Datei, Spezifische Informationen je nach Sektions-Typ)

**Segmente/Sektionen:** Jeweils eine andere Einteilung für die gleichen Speicherbereiche. View des Loaders sind die Segmente: Segmente definieren die Portionen, die in den Hauptspeicher geladen werden. View des Compilers sind die Sektionen: Sektionen definieren «gleichartige» Daten. Linker vermittelt zwischen beiden Views. Kombiniert gleichnamige Sektionen aus unterschiedlichen Objekt-Dateien und definiert Segmente.

|  |  |
| --- | --- |
| Section Types | Special Sections |
| SHT\_PROGBITS: Data defined by program, linker doesn’t interpret this  SHT\_SYMTAB: Symbol Table  SHT\_STRTAB: String Table  SHT\_REL/RELA: Relocation Information  SHT\_HASH: Hashtable for symbols  SHT\_DYNAMIC: Information for dynamic Linking  SHT\_NOBITS: Section without data in file | .bss: uninitialized data  .data: initialized data  .data1: initialized data  .debug: debug information  rodata: read only data  .rodata1: read only data  .text: executable instructions  .symtab: Symbol-Table  .strtab: String-Table |

**String Tabelle:** Bereich in der Datei, der nacheinander null-terminierte Strings enthält. Strings werden relativ zum Beginn der Tabelle referenziert. Enthält keine String-Literale aus Programmen (z.B. «Hello World»); diese sind typischerweise in .rodata.

**Symbole:** Symboltabelle enthält jeweils einen Eintrag je Symbol. Ein Symbol ist 16 Byte gross, aufgeteilt in 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) und 4 Byte Info (Typ, Binding Attribute, Referenz auf Sektions Header)

## bibliotheken

**Statische Bibliotheken:** Sind Archive von Objekt-Dateien. Archive sind Dateien, die andere Dateien enthalten (wie ein primitives Dateisystem)

clang -c f1 . c -o f1 . o // Kompilieren

clang -c f2 . c -o f2 . o

ar r libmylib . a f1 . o f2 . o // Archiv erstellen

**Statische Bibliotheken mit Program:** 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.

**Vorteil:** Einfach zu verwenden und zu implementieren. Programmierer/Linker haben geringen Mehraufwand. OS/Compiler haben keinen Mehraufwand.

**Nachteil:** Programme müssen bei Änderungen in Bibliotheken neu erstellt werden und Funktionalität ist fix (Plugins nicht möglich).

Dynamische Bibliotheken

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

clang - fPIC -c f1 . c -o f1 . o // Kompilieren

clang - fPIC -c f2 . c -o f2 . o

clang - shared -Wl , - soname , libmylib . so .2 -o libmylib . so .2.1 f1 . o f2 . o – lc // create image -> . so file

**Vorteil:** Programm kann Updates erhalten, ohne das Binary zu ändern. Funktionalität von kann unabhängig voneinander upgedatet werden. Bugfixes können von Bibliotheks-Entwickler direkt zur Anwenderin gebracht werden. Programm muss nur die Bibliotheken laden, die es braucht. Somit startet das Programm schnell mit kleiner, häufig benötigter Basis-Funktionalität

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Executable |  |  |  | Executable |  |  |
| prog.o |  | libmylib.so.1.1 |  | prog.o |  | libmylib.so.1.1 |
|  |  |  |  |  |
|  |  | libmylib.so.1.2 |  |  |  | libmylib.so.1.2 |

**Position-Dependent Code:** Code hängt normalerweise von absoluten Adressen ab (Funktionsaufrufe, globale Variablen etc.). Wird solcher Code im Speicher verschoben, stimmen die Referenzen nicht mehr. Code muss also nach dem Verschieben

reloziert werden

**Position-Independent Code:** Position-Independent Code hängt nicht von seiner Adresse ab. Verwendet keine absoluten, sondern nur Adressen relativ zum Instruction Pointer. Dadurch sieht der Code immer gleich aus, egal wo im Speicher er liegt.

**Global Offset Table:** In jedem Programm/Bibliothek haben wir eine globale Offset-Tabelle, die die relativen Abstände zu den Bibliotheken usw. anzeigt, die Sie benötigen/brauchen. Auf diese Weise kann man dieselbe dynamische Bibliothek verwenden, die in den Speicher geladen ist, auch wenn ein anderer Prozess sie ebenfalls verwendet.

**Process Linkage Table:** Implementiert "Lazy Binding"! Eine Funktion pro Eintrag. PLT-Eintrag hat Aufrufanweisung im GOT-Eintrag. GOT zeigt auf Proxy-Funktion. Die Proxy-Funktion findet den Link zur richtigen Funktion und überschreibt den eigenen Eintrag. Der Vorteil ist, dass die Prüfung, ob die Funktion geladen ist, nach dem ersten Laden entfällt (teuer!!).

**Dlopen:** Öffnet eine dynamische Bibliothek und gibt ein Handle darauf zurück. mode gibt Art an, wie mit der Bibliothek umgegangen wird. (RTLD\_NOW, RTLD\_LAZY, RTLD\_GLOBAL, RTLD\_LOCAL)

void\* dlopen (char \* filename, int mode)

**Dlsym:** Gibt die Adresse des Symbols name aus der mit handle bezeichneten Bibliothek zurück. Keine Typeninformation übertragen, nur Adresse. Unklar ob Funktion oder Variable. Gibt 0 zurück wenn no worky.

void\* dlsym (void \* handle, char \* name)

**Dlclose:** Schliesst das durch handle bezeichnete, zuvor von dlopen geöffnete Objekt. Gibt 0 zurück, wenn erfolgreich

int dlclose (void \* handle)

**Dlerror:** Gibt Fehlermeldung als null-terminierten String zurück, wenn ein Fehler aufgetreten war oder null wenn fehlerfrei

char\* dlerror ()

**Beispiel:**

#include <dlfcn.h> #include <stdio.h>

int main() {

  typedef void (\*func\_t)(int \*);

  void \*handle = dlopen("./libinc.so", RTLD\_NOW); // open dynamic library

  func\_t f = dlsym(handle, "increment");

  int a = 5;

  printf("before: %d\n", a);

  (\*f)(&a);

  printf("after: %d\n", a);

  dlclose(handle);

}

**Automatisches Laden von Shared Objects (SO):** Können automatisch bei Bedarf geladen werden Im Executable muss eine Referenz auf das SO (ELF) hinterlegt sein (Dependency). OS sucht automatisch beim Programmstart die richtigen Bibliotheken

**SO Benennungsschema:** Linker-Name (lib + Bibliotheksname + .so),

SO-Name: (Linker-Name + . + Version Nr) und Real-Name: SO-name + . + Subversionsnr

**Real Name:** Wird beim Erstellen des SO verwendet. Versions Nr wird erhöht, wenn sich die Schnittstelle ändert. Unterversion Nr wird erhöht, wenn die Schnittstelle gleichbleibt.

**Shared Object Facts:** Referenced Shared Objects are stored inside the executables (readelf -d shows the content of dynamic section, type of the entries is NEEDED).ldd shows all shared objects, even indrectly used ones (executes executables and instructs loader to show trace. Should only be used on trusted executables!!). Pretty much all Executables (C) need at least two shared objects (libc.so -> standard C library, ld-linux.so -> ELF Shared Object Loader)

# threads

**Motivation:** Anwendung ruft Aktivität periodisch auf, welcher arbeitet und dann den Kontrollfluss zurückgibt. Anwendung kann Aktivität nicht abbrechen, d.h mögliches Blockieren der Anwendung.

**Prozessmodell:** Jeder Prozess hat virtuell den ganzen PC für sich allein zur Verfügung (eigener Adressraum, freie Registerbelegung, Isolation). Prozesse sind gut geeignet für unabhängige Applikationen. Jedoch ist Realisierung paralleler Abläufe innerhalb derselben Applikation aufwändig, Overhead für Prozesserzeugung und gemeinsame Ressourcennutzung ist erschwert.

**Threadmodell:** Threads sind parallel ablaufende Aktivitäten innerhalb eines Prozesses. Threads haben auf alle Ressourcen. Prozess gleichermassen Zugriff (Code, Globale Variablen, Heap, Geöffnete Dateien, MMU).

**Thread als Stack:** Jeder Thread benötigt aber einen eigenen Kontext und ein eigenes Stack, weil er eine eigene Funktions-Aufrufkette hat. Diese Informationen werden häufig in einem Thread-Control-Block abgelegt

**Threads vs. Prozess:** Prozesse umfassen Threads, das heisst, ein Prozess kann mehrere Threads beinhalten. Beim Forking wird der Adressraum dupliziert, aber Threads teilen sich denselben Adressraum.

**Speedup Factor:**

|  |  |
| --- | --- |
| **Amdahl’s Law Formula (T = time)** | **Parallelization using N processors** |
|  |  |
| **SpeedUp** | **Efficiency** |
|  |  |

## posix thread api

**Create Thread:** 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
* start\_function ist die Adresse der ersten zu aufrufenden Funktion.
* argument ist typischerweise ein Pointer auf eine Datenstruktur auf dem Heap

int pthread\_create (

    pthread\_t \*thread\_id,

    pthread\_attr\_t const \*attributes,

    void \* (\*start\_function) (void \*),

    void \*argument)

Um Attribute anzugeben, muss man nach folgendem Muster verfahren. Der Grund für diesen Ansatz ist, dass pthread\_attr\_t je nach Implementation weiteren Speicher benötigen kann.

pthread\_attr\_t attr;

pthread\_attr\_init (&attr);

pthread\_attr\_setstacksize (&attr, 1 << 16);

pthread\_create (..., &attr, ...);

pthread\_attr\_destroy (&attr);

**Lebensdauer eines Threads:** Ein Thread lebt solange, bis er entweder aus der Funktion start\_function zurückspringt, pthread\_exit aufruft, ein anderer Thread pthread\_cancel aufruft oder sein Prozess beendet wird

**Exit Thread:** Beendet den Thread und gibt den return\_value zurück. Das ist äquivalent zum Rücksprung aus start\_function mit dem Rückgabewert.

void pthread\_exit (void \*return\_value)

**Cancel Thread:** 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, wenn nicht.

int pthread\_cancel (pthread\_t thread\_id)

**Detach Thread:** Entfernt den Speicher, den ein Thread belegt hat, falls dieser bereits beendet wurde. Beendet den Thread aber nicht, falls er noch nicht beendet wurde.

int pthread\_detach (pthread\_t thread\_id)

**Join Thread:** Wartet solange, bis der Thread mit thread\_id beendet wurde. Nimmt den Rückgabewert des Threads (von pthread\_exit bzw. Rückgabewert der start\_function) im Out-Parameter return\_value entgegen. return\_value kann NULL sein, wenn Rückgabewert nicht gewünscht. Ruft pthread\_detach auf.

int pthread\_join (pthread\_t thread\_id, void \*\*return\_value)

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

pthread\_t pthread\_self (void)

## thread local storage (tls)

**Motivation**: 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 bei mehreren Threads unerwartetes Verhalten aufweisen

**Thread-Local Storage:** Mechanismus, der globale Variablen per Thread zur Verfügung stellt. In höheren Programmiersprachen wird dafür häufig nur ein einfaches Attribut angegeben. Auf OS-Ebene braucht man dafür 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

**Key erzeugen:** 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. 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 falls Ok

int pthread\_key\_create(pthread\_key\_t \*key,void \*destructor)(void\*))

**Key löschen:** Entfernt den Key und die entsprechenden Values aus allen Threads. Der Key darf nach diesem Aufruf nicht mehr verwendet werden. Das Programm muss dafür sorgen, sämtlichen Speicher freizugeben, der eventuell zusätzlich alloziert worden war.

int pthread\_key\_delete(pthread\_key\_t key)

**Key holen/setzen:** Schreibt bzw. liest den Wert, der mit dem Key in diesem Thread assoziiert ist. Typischerweise verwendet man den Wert als Pointer auf einen Speicherbereich

int pthread\_setspecific(pthread\_key\_t key, const void \* value)

void \* pthread\_getspecific(pthread\_key\_t key)

**Beispiel:**

typedef struct {

    int code; char \*message;

} error\_t;

pthread\_key\_t error;

void set\_up\_error (void) {

    pthread\_setspecific (error, malloc (sizeof(error\_t)));}

void print\_error (void) {

    error\_t \* e = pthread\_getspecific (error);

    printf ("Error %d: %s\n", e->code, e->message);}

int force\_error(void) {

    error\_t \* e = pthread\_getspecific (error);

    e->code = 98;

    e->message = "file not found";

    return -1;}

void\* thread\_function (void\*) {

    set\_up\_error ();

    if (force\_error () == -1) {

        print\_error ();

    }}

int main (int argc, char \*\*argv) {

    pthread\_key\_create (&error, NULL);

    pthread\_t tid;

    pthread\_create (&tid, NULL, &thread\_function, NULL);

    pthread\_join (tid, NULL);}

# scheduling

**Zustände eines Threads:**

* Running: Auf dem CPU laufender Thread. Maximal 1 Thread im running
* Ready: Können laufen, warten, bis CPU frei ist
* Waiting: Warten auf ein Ereignis (join oder Eingabe von Tastatur)

A close-up of a computer code

Description automatically generatedA diagram of a scheduler

Description automatically generated

**Wartende Threads:** Threads, die auf Ereignisse warten, müssen das nicht in einer Endlosschleife tun (Busy-Wait). Stattdessen registriert das OS sie auf das entsprechende Ereignis. Tritt das Ereignis ein, ändert das OS den Zustand auf ready.

**Ready Queue:** In der Ready-Queue befinden sich alle Threads, die bereit sind zu laufen. Je nach Scheduling-Algorithmus kann Queue auf ein Red-Black Tree sein.

**Powerdown-Modus:** Wenn kein Thread laufbereit ist, schaltet das OS den Prozessor in einem Standby, um Strom zu sparen. Wird mit dem nächsten Interrupt geweckt und fährt mit seiner normalen Operation fort. Im Normalfall verbringt CPU sehr viel Zeit im Standby.

**Arten von Threads:** Die Unterscheidung ist fliessend, aber gut entworfene Systeme trennen rechen-intensive von interaktiven Aktivitäten

* I/O-lastig: Kommuniziert sehr häufig mit I/O-Geräten und rechnet relativ wenig
* CPU-lastig: Kommuniziert kaum oder gar nicht mit I/O-Geräten und rechnet fast ausschliesslich

**Arten der Nebenläufigkeit:**

* Kooperativ: Jeder Thread entscheidet selbst, wann er den Prozessor abgibt
* Präemptiv: Scheduler entscheidet, wann einem Thread der Prozessor entzogen wird

**Präemptives Multithreading:** Aktueller Thread läuft so lange weiter, bis er auf Ein-/Ausgabedaten oder anderen Thread wartet, freiwillig auf CPU verzichtet, etc.

**Parallel vs Quasiparallel vs Nebenläufig:**

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

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

* 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 benötig, also vom Eintritt in waiting bis zum nächsten running

## scheduling strategien

**Parallele Ausführung:** Ideale Strategie. n Threads auf n Prozessoren. Jeder Thread kann seinen Prozessor immer dann verwenden, wenn er ihn braucht. In Praxis unrealistisch, da es immer mehr Threads als Prozessoren gibt.

**Anforderungen an Scheduler:**

* Geschlossene Systeme: Der Hersteller des Systems kennt alle Anwendungen und weiss, in welcher Beziehung sie zueinanderstehen (z.B. embed. Systeme, Server)
* Offene Systeme: Der Hersteller des OS muss von typischen Anwendungen ausgehen und dahin gehend optimieren (z.B. Consumer Devices)

**Anforderungen aus Anwendungssicht:**

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

**Anforderungen aus Systemperspektive:**

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

**Unvereinbarkeit:** Grundsätzlich können Scheduler nicht auf alle Anforderungen gleichzeitig optimiert werden. Wahl des Schedulers hängt vom Einsatzzweck des Schedulers ab. Ausserdem bieten viele OS die Möglichkeit, den Scheduler zu beeinflussen.

**First Come, First Served:** Threads werden in der Reihenfolge geschedulet, in der sie der Ready-Queue (FIFO) hinzugefügt wurden. Nicht präemptiv da Threads den Prozessor nur abgeben, wenn sie auf waiting wechseln oder sich beenden. Durchschnittliche Wartezeit hängt von der Reihenfolge des Eintreffens der Threads ab

Chart

Description automatically generated

**Shortest Job First:** Wählt den Thread aus, der den kürzesten nächsten Prozessor-Burst hat. Bei gleicher Prozessor-Burst-Länge wählt der Scheduler nach FCFS aus. Kann kooperativ oder präemptiv eingesetzt werden. Ergibt optimale Wartezeit da der kürzeste Prozessor-Burst die anderen Threads minimal blockiert.

Chart, bar chart

Description automatically generated

**Abschätzung der Zeit in SJF:** Kann nur korrekt implementiert werden, wenn die Längen der Prozessor-Bursts bekannt sind. Die Schwierigkeit in anderen Systemen besteht darin zu wissen, wie lang der nächste Prozessor-Burst sein wird. Kann dann nur annähernd implementiert werden, wenn man eine Abschätzung anhand historischer Daten vornimmt

**Round Robin:** Scheduler definiert eine Zeitscheibe (time slice) von etwa 10 bis 100ms. Grundprinzip folgt FCFS, aber ein Thread kann nur solange laufen, bis seine Zeitscheibe erschöpft ist. Dann wird er wieder in der Ready-Queue hinten angehängt.

Chart

Description automatically generated

**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.

**Starvation:** Problem welches bei prioritäten-basierten Systemen auftreten kann. 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, sodass der Thread erreicht irgendwann eine genügend hohe Priorität, um zu laufen.

**Multi-Level:** 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. Queues können z.B. gegeneinander priorisiert werden: Threads in Queues mit höherer Priorität werden immer bevorzugt.

**Multi-Level 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. Thread landet in Ready-Queue mit niedrigerer Priorität. Typischerweise werden die Zeitscheiben mit niedrigerer Priorität grösser. Threads mit kurzen Prozessor-Bursts werden bevorzugt.

**Nice-Wert:** Jeder Prozess p hat einen Nice-Wert. Soll p bevorzugen, wenn p kleiner (p ist weniger nett). Soll p weniger oft laufen lassen, wenn np grösser (p ist netter). Auf Linux liegt der Nice-Wert im Bereich von -20 bis +19. nice(i) addiert i zum nice Wert des Prozess

int nice (int i)

**Getpriority/Setpriority:** Kann Priorität des Prozesses, Gruppe oder User ändern.

int getpriority (int wich, id\_t who)

int setpriority (int wich, id\_t who, int prio)

# mutexe und semaphore

Jeder Thread hat seinen eigenen Instruction-Pointer. Die Instruction-Pointer aller Threads werden unabhängig voneinander bewegt. Bei parallelen Threads völlig unabhängig und nicht synchron, selbst bei identischem Code (z.B. wegen Warten auf Speicherzugriffe)

**Producer-Consumer Problem:** Producer Thread erzeugt Items. Consumer Thread verarbeitet diese Items. Beide Threads arbeiten unterschiedlich schnell. Items werden über einen begrenzt grossen Ring-Puffer übermittelt

**Inkrementierung in C:** Keine einzelne Instruktion, sondern typischerweise 3. CPU kann zwischen diesen Instruktionen jeweils unterbrechen und bspw. den Thread wechseln

**Race Condition:** Wenn Ergebnisse von der Ausführungsreihenfolge einzelner Instruktionen abhängen, spricht man von einer Race Condition

**Critical Section:** Jeder kooperierende Thread hat einen Code-Bereich, in dem er Daten mit anderen Threads teilt, die Critical Section. Die Critical Section kann für jeden Thread anderer, aber auch gleicher Code sein

**Anforderungen an Synchronisationsmechanismen:**

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

**Atomare Instruktionen:** Eine atomare Instruktion kann vom Prozessor unterbrechungsfrei ausgeführt werden. Achtung: selbst einzelne Instruktionen können unter Umständen nicht atomar durchgeführt werden, z.B.:

mov [2000], rax; //atomar, weil aligned (durch 8 teilbarer Adresse)

mov [2009], rax; //nicht atomar, weil nicht aligned

Moderne Computer-Architekturen geben kaum Garantien bezüglich der Ausführung von Instruktionen. Instruktionen müssen nicht atomar sein und Sequenzen können umgeordnet werden, wenn sie funktional noch äquivalent sind.

**Abschalten von Interrupts:** Alle Interrupts werden abgeschaltet, wenn eine Critical Section betreten werden soll. Auf Systemen mit einem Prozessor effektiv: Es kann zu keinem Kontext-Wechsel kommen. Für Systeme mit mehreren Prozessoren nicht praktikabel da parallele Threads. Generell gefährlich da OS keine Threads nicht unterbrechen kann solange die Interrupts ausgeschaltet sind

**Verwendung spezieller Instruktionen:** Moderne Prozessoren stellen in irgendeiner Form eine von zwei atomaren Instruktionen zur Verfügung (Test-And-Set/Compare-And-Swap). Die Hardware garantiert, dass keine zwei dieser Instruktionen gleichzeitig ausgeführt werden, auch über mehrere Prozessoren hinweg.

**Test-And-Set:** Kommen zwei Threads T1 und T2 genau gleichzeitig an die while-Schleife, garantiert die HW, dass nur T1 test\_and\_set ausführt. T2 muss warten. T1 setzt lock auf 1, liest aber 0 und verlässt die Schleife sofort. T2 sieht lock auf jeden Fall als 1 und bleibt in der Schleife.

int test\_and\_set (int \*target) {

    int value = \*target;

    \*target = 1;

    return value;

}

while (tas (&lock) == 1) {}

/\* critical section \*/

lock = 0;

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

int compare\_and\_swap (int \*a, int expected, int new\_a) {

    int value = \*a;

    if (value == expected) {

        \*a = new\_a;

    }

    return value;

}

while (cas (&lock, 0, 1) == 1) {}

/\* critical section \*/

lock = 0;

## semaphore

Ein Semaphor enthält einen Zähler z ≥ 0. Auf den Semaphor wird nur über spezielle Funktionen zugegriffen.

* Post: Erhöht z um 1
* Wait: Wenn z > 0: Verringert z um 1 und setzt Ausführung fort, sonst versetzt Thread in waiting, bis ein anderer Thread z erhöht

Die Funktionen sind synchronisiert, so dass z immer korrekt gesetzt wird

**Sem\_init:** Initialisiert den Semaphor sem, so dass er value Marken enthält. Ist pshared = 0, kann sem nur innerhalb eines Prozesses verwendet werden

int sem\_init (sem\_t \*sem, int pshared, unsigned int value);

Typischerweise legt man 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. in main

sem\_t sem; int main (int argc, char \*\*argv) {

    sem\_init (&sem, 0, 4);

    return 0;

}

**Sem\_Wait/Sem\_Post:** Geben 0 zurück, wenn Aufruf OK, sonst -1 und Fehlercode in errno. Im Fehlerfall wird der Semaphor nicht verändert

int sem\_wait (sem\_t \*sem);

int sem\_post (sem\_t \*sem);

**Sem\_trywait/Sem\_timedwait:** trywait gleich wie sem\_wait, aber bricht ab anstatt blockt falls dekrementieren nicht möglich ist. Timedwait wartet entweder bis timeout erreicht wird oder wait erfolgreich war.

int sem\_trywait (sem\_t\* sem);

int sem\_timedwait (sem\_t\* sem, const struct timespec\* abs\_timeout);

**Sem\_Destroy:** Entfernt möglichen zusätzlichen Speicher, den das OS mit sem assoziiert hat

int sem\_destroy (sem\_t \*sem);

**Priority Inversion:** Priority Inversion entsteht wenn ein hoch-priorisierter Thread wartet auf eine Ressource, die von einem niedriger priorisierten Thread gehalten wird .

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

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Thread A |  |  | |  | |  | |  | |
|  |  |  | |  | |  | |  | |
| Thread B |  |  | |  | |  | |  | |
|  |  |  | |  | |  | |  | |
| Thread C |  | waiting | | | |  | |  | |
|  | 10 | | 20 | | Zeit(ms) | |  | |  |

**Priority Inheritance:** Gemeinsam verwendete Ressourcen werden im schlimmsten Fall mit der niedrigsten Priorität aller beteiligten Threads gehalten. Bei Priority Inheritance erhält der Thread, der eine Ressource hält, die höchste Priorität aller Threads, die auf diese Ressource warten.

**Beispiel:** Thread A hat niedrige Priorität und hält einen Mutex M. Thread B hat mittlere Priorität. Thread C hat hohe Priorität und läuft gerade. Nach 10 ms benötigt Thread C den Mutex M. Die Priorität von A wird temporär auf die Priorität von C gesetzt. A läuft, bis er den Mutex M freigibt. Dann läuft wieder C

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Thread A |  |  | |  | | |  | |
|  |  |  | |  | | |  | |
| Thread B |  |  | |  | | |  | |
|  |  |  | |  | | |  | |
| Thread C |  | waiting | | |  | |  | |
|  | 10 | | 20 | | | Zeit(ms) | |

## mutex

Ein Mutex hat einen binären Zustand z, der durch zwei Funktionen verändert werden kann:

* Acquire: Wenn z=0: Setze z auf 1 und fahre fort, ansonsten blockiere Thread bis z=0
* Release: Setzt z=0

**Anwendung:** Einfachste Form der Synchronisierung. Acquire und Release müssen immer paarweise durchgeführt werden

acquire (mutex); ++counter; release (mutex);

**POSIX Thread Mutex API:** Initialisiert die opake Daten-Struktur pthread\_mutex\_t. Attribute sind optional, Verwendung analog zu pthread-Attributen mit pthread\_mutexattr\_init, ...\_destroy, etc.

int pthread\_mutex\_init(pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \*attr);

// acquire (blocking)

int pthread\_mutex\_lock (pthread\_mutex\_t \*mutex);

// attempt to acquire (non-blocking)

int pthread\_mutex\_trylock (pthread\_mutex\_t \*mutex);

// release

int pthread\_mutex\_unlock (pthread\_mutex\_t \*mutex);

// clean up

int pthread\_mutex\_destroy (pthread\_mutex\_t \*mutex); 50

pthread\_mutex\_t mutex; // globale Variable

**Beispiel:**

int main () {

    pthread\_mutex\_init (&mutex, 0);

    // run threads and wait for them to finish

    pthread\_mutex\_destroy (&mutex);

}

void \* thread\_function (void \* args)  {

    while (running) {

        // Do something, then Enter critical section pthread\_mutex\_lock (&mutex);

        // Perform crit. section, e.g. ++counter

// Leave critical section

        pthread\_mutex\_unlock (&mutex); // Do something more

    }

}

# signale, pipes und sockets

## interrupts signals

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.

**Quellen von Signalen:** Hardware/OS (Ungültige Instruktion, Segmentation fault, Division durch 0) oder andere Prozesse (Abbruch über Ctrl-C oder Aufruf des Kommandos kill)

**Signale behandeln:** Jeder Prozess hat pro Signal einen Handler. Bei Prozessbeginn gibt es für jedes Signal einen von drei Default-Handlern. Fast alle Signal-Handler können überschrieben werden (ausser SIGKILL und SIGSTOP)

* Ignore-Handler: ignoriert das Signal
* Terminate-Handler: beendet das Programm
* Abnormal-Terminate-Handler: beendet das Programm und erzeugt Core Dump

**Wichtige Signale:**

* SIGFPE: Fehler in arithmetischer Operation (Floating Point Error)
* SIGILL: Ungültige Instruktion
* SIGSEGV: Ungültiger Speicherzugriff
* SIGSYS: Ungültiger Systemaufruf
* SIGTERM: Normale Anfrage an den Prozess, sich zu beenden
* SIGINT: Nachdrücklichere Aufforderung. Wird durch Benutzer ausgeführt (Ctrl-C )
* SIGQUIT: Wie SIGINT, aber anormale Terminierung. Wird generiert, wenn der Benutzer Ctrl-\\ (Ctrl-AltGr-<) drückt
* SIGABRT: Wie SIGQUIT (anormale Terminierung). Wird bevorzugt vom Prozess an sich selbst geschickt, z.B. wenn er selbst einen Programmierfehler bemerkt
* SIGKILL: Letzte Zuflucht: kann der Prozess nicht blockieren, ignorieren oder abfangen
* SIGTSTP: Versetzt einen Prozess in den Zustand stopped, ähnlich wie waiting Kann von der Shell mit Ctrl-Z ausgelöst werden
* SIGSTOP: Wie SIGTSTP, aber kann nicht ignoriert oder abgefangen werden
* SIGCONT: Setzt den Prozess fort. Wird auf der Shell mit den Kommandos fg bzw. bg erzeugt

**Signale von 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

**Signal-Handler im Programm ändern:** signal ist die Nummer des Signals (SIGKILL oder SIGSTOP nicht erlaubt), definiert Signal-Handler für signal, wenn new != 0 und gibt den bestehenden Signal-Handler für signal zurück, wenn old != 0

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

**Mehr Funktionen für Signal:**

int sigemptyset (sigset\_t \*set) // no signal chosen

int sigfillset (sigset\_t \*set) // all signals chosen

int sigaddset (sigset\_t \*set, int signal) // adds a signal to the set

int sigdelset (sigset\_t \*set, int signal) // removes a signal from the set

int sigismember (const sigset\_t \*set, int signal) // returns 1 when signal is inside the set

**Beispiel:**

void hand(int something) { printf("henlo from handler %d\n", something); exit(0); }

struct sigaction sig;

int main() {

  sigset\_t set;

  sigaddset(&set, SIGINT); // defines that SIGINT is ignored during handler execution

  sig.sa\_handler = &hand; // address of function which is executed when signal is received

  sig.sa\_mask = set; // deﬁnes signals that’ll be blocked when the handler runs (own signal is always blocked)

  sig.sa\_flags = 0; // allows multiple additional properties

  sigaction(SIGINT, &sig,0); // enable handler for SIGINT

  while(1){}

}

## pipes

**Duplizieren eines File-Descriptors:** Beide Funktionen duplizieren den File-Descriptor source\_fd und geben den neuen File-Descriptor zurück. dup alloziert einen neuen File-Descriptor. dup2 überschreibt den Eintrag destination\_fd.

int dup (int source\_fd); 2

int dup2 (int source\_fd, int destination\_fd);

**Abstrakte Dateien:** Konsole/Terminal ist offensichtlich keine Datei auf einem Datenträger. Dennoch ist der Standard-Ausgabestream oft die Ausgabe auf der Konsole. Eine «Datei» muss nur open und close sowie read und/oder write unterstützen.

**Pipes:** Pipe ist eine «Datei» im Hauptspeicher, die über zwei File-Deskriptoren verwendet wird. Daten, die in write end geschrieben werden, können aus read end genau einmal und in FIFO-Manier gelesen werden. Pipes unterstützen kein lseek. Pipes ermöglichen Kommunikation über Prozess-Grenzen hinweg.

**Pipe erzeugen:** Erzeugt eine Pipe und zwei File-Deskriptoren, die in fd abgelegt werden. Pipe lebt solange, wie sie in einem Prozess geöffnet bleibt

int pipe (int fd [2]); // equivalent to int pipe (int \*fd)

Diagram, text

Description automatically generated**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 geschrieben werden. Sind keine Daten in der Pipe und gibt es kein geöffnetes Write-End, gibt read 0 zurück (EOF)

**Pipes Beispiel:** Eingabe in der Shell: cmda | cmdb

int fd [2];

pipe (fd);

int id1 = fork ();

if (id1 == 0) { // child (cmda)

    // don't use read end

    close (fd [0]);

    // define pipe write end as stdout

    dup2 (fd [1], 1);

    exec ("cmda", ...);

} else { // parent (shell)

    int id2 = fork ();

    if (id2 == 0) { // child (cmdb)

        // don't use write end

        close (fd [1]);

        // define pipe read end as stdin

        dup2 (fd [0], 0);

        exec ("cmdb", ...);

    } else { wait (0); }

}

**Pipes Anmerkung:** Pipes sind unidirektional: nicht spezifiziert, was beim Schreiben ins read end oder Lesen vom write end passiert. Lesender Prozess sollte write end schliessen, schreibender Prozess read end. Sind alle read ends geschlossen, erhält Prozess mit write end ein SIGPIPE. Wann der Transport erfolgt, ist implementierungsabhängig, d.h mehrere writes können zu einem zusammengefasst werden, z.B. wenn sie unmittelbar hintereinander erfolgen.

**Named Pipes:** Erzeugt eine Pipe mit Namen und Pfad im Dateisystem. Hat Permission Bits wie eine normale Datei (von mode). Lebt unabhängig vom erzeugenden Prozess, je nach System auch über Reboots hinweg. Muss (wie Datei) explizit mit unlink gelöscht werden

int mkfifo(const char \*path, mode\_t mode);

## sockets (berkeley)

**Socket erzeugen:** Erzeugt einen neuen Socket als «Datei». Sockets sind nach Erzeugung zunächst unbenannt. Alle Operationen auf Sockets blockieren per default (kann aber umkonfiguriert werden). Domain entweder AF\_UNIX (interner Socket für Maschine), AF\_INET (IPv4), AF\_INET6 (IPv6). Type kann SOCK\_DGRAM (UDP) oder SOCK\_STREAM (TCP) sein.

int socket(int domain, int type, int protocol);

**Angaben von Adressen/IP-Adressen:**

struct sockaddr { // general interface

    sa\_family\_t sa\_family; // address family (integer)

    char sa\_data[]; // address (variable-length data)

}

struct in\_addr {

    in\_addr\_t s\_addr; // 32-bit unsigned integer

}

struct sockaddr\_in { // Internet (IP) address

    sa\_family\_t sin\_family; // = AF\_INET

    in\_port\_t sin\_port; // 16-bit unsigned integer

    struct in\_addr sin\_addr;

}

**Beispiel Angabe IP-Adresse:** htons konvertiert 16 Bit von Host-Byte-Order (Intel: Little-Endian) zu Network-Byte-Order (Big-Endian), htonl 32 Bit.

struct sockaddr\_in ip\_addr;

ip\_addr.sin\_port = htons (443);

inet\_pton (AF\_INET, "192.168.0.1", &ip\_addr.sin\_addr.s\_addr);

// port in memory: 0x01 0xBB

// addr in memory: 0xC0 0xA8 0x00 0x01

**Beispiel:**

#include <errno.h>, <stdio.h>, <string.h>, <sys/socket.h>, <unistd.h>

int main(int argc, char \*\*argv) { // Server code

  void \*buff[100];

  int sock = socket(AF\_UNIX, SOCK\_STREAM, 0);

  struct sockaddr addr;

  addr.sa\_family = AF\_UNIX;

  strncpy(addr.sa\_data, argv[1], sizeof(addr.sa\_data));

  int b = bind(sock, &addr, sizeof(addr));

  if (b != 0) { perror(strerror(errno)); return -1;  }

  listen(sock, 5);

  while (1) {

    int clientsock = accept(sock, 0, 0);

    ssize\_t rec = recv(clientsock, buff, 20, 0);

    printf("%s\n", (char \*)buff);

    close(clientsock);

  }

  close(sock);

  unlink(argv[1]); // Wichtig zu unlinken, sonst bleibt eine Datei bestehen im FS

}

int main(int argc, char \*\*argv) { // Client code

  int sock = socket(AF\_UNIX, SOCK\_STREAM, 0);

  struct sockaddr addr;

  addr.sa\_family = AF\_UNIX;

  strncpy(addr.sa\_data, argv[1], sizeof(addr.sa\_data));

  int c = connect(sock, &addr, sizeof(addr));

  if (c != 0) { perror(strerror(errno)); }

  ssize\_t rec = send(sock, argv[2], 20, 0);

  close(sock); }

**Binden einer lokalen Adresse:** Bindet den Socket an die angegeben, unbenutzte lokale Adresse, wenn noch nicht gebunden. Blockiert, bis der Vorgang abgeschlossen ist

int bind (int socket,const struct sockaddr \*local\_address, socklen\_t address\_len);

**Aufbau einer Verbindung:** Bindet den Socket an eine neue, unbenutzte lokale Adresse, wenn noch nicht gebunden. Versucht eine Verbindung zur angegebenen Remote-Adresse aufzubauen. Blockiert, bis Verbindung steht oder ein (unspezifizierter) Timeout eintritt

int connect (int socket, const struct sockaddr \*remote\_address, socklen\_t address\_len);

**Vorbereitung zum Empfangen von Verbindungen:** Markiert den Socket als «bereit zum Empfang von Verbindungen». Erzeugt eine Warteschlange, die so viele Verbindungsanfragen aufnehmen kann, wie backlog angibt (oder die maximal mögliche Grösse)

int listen (int socket, int backlog);

**Erzeugen neuer Sockets je Verbindungsanfrage:** 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

int accept (int socket, struct sockaddr \*remote\_address, socklen\_t address\_len);

S**enden und Empfangen von Daten:** flags für eher spezielle Anwendungsfälle, die wir nicht weiter betrachten. Puffern der Daten ist Aufgabe des Netzwerkstacks

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);

**Schliessen eines Sockets:** Schliesst den Socket für den aufrufenden Prozess. Hat ein anderer Prozess den Socket noch geöffnet, bleibt die Verbindung bestehen

int close (int socket);

**Shutdown eines Sockets:** Schliesst den Socket für alle Prozesse und baut die entsprechende Verbindung ab. Mode kann SHUT\_RD (kein Read bis Shutdown), SHUT\_WR (kein Write bis Shutdown) oder SHUT\_RDWR (Beides nicht) sein.

int shutdown (int socket, int mode);

# message passing/shared memory

**Motivation:** Prozesse sind voneinander isoliert. Oft müssen Prozesse miteinander interagieren. Entweder als Teil einer einzigen Applikation oder als Teil eines komplexen Systems interagierender Applikationen

## message-passing

Mechanismus mit Operationen Send (Kopiert Nachricht aus dem Prozess) und Receive (Kopiert Nachricht in den Prozess). Dabei können Implementierungen nach feste/variable Nachrichtengrösse, direkte/indirekte und synchrone/asynchrone Kommunikation, Puffer.

**Nachrichtengrösse:** Message-Passing mit fester Nachrichtengrösse ist umständlicher zu verwenden, da User bei Überschreitung selbst Vorsorge treffen muss, um die Nachricht in kleine Teile aufzutrennen. Bei Unterschreiten wird Speicher verschwendet. Message-Passing mit variabler Nachrichtengrösse ist aufwändiger zu implementieren

**Direkte Kommunikation Send:** Bei direkter Kommunikation werden Nachrichten von einem Prozess P1 an einen Prozess P2 adressiert. P1 muss den Empfänger seiner Nachrichten kennen (oder einen Alias). Kommunikation nur zwischen jeweils genau 2 CPU

Direkte Kommunikation Receive: Bei symmetrischer direkter Kommunikation muss P2 den Sender seiner Nachrichten kennen oder einen Alias (receive (P1, message)). Bei asymmetrischer direkter Kommunikation muss P2 den Sender seiner Nachrichten nicht kennen, sondern erhält die ID in einem Out-Parameter id (receive (id, message))

**Indirekte Kommunikation:** Bei indirekter Kommunikation existieren spezifische OS-Objekte: Mailboxen, Ports oder Queues. P1 sendet Nachrichten an Queue q (send(Q, message)). P2 empfängt Nachrichten aus Q (receive(Q, message)). Kommunikation erfordert, dass beide Teilnehmer die gleiche Mailbox kennen. Kann mehr als eine Mailbox zwischen zwei Teilnehmern geben.

**Synchronisation:** 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. OS kann eine Kopieroperation sparen und direkt vom Sende- in den Empfänger-Prozess kopieren. Triviale Implementierung des Producer-Consumer-Problems

|  |  |
| --- | --- |
| Producer | Consumer |
| message msg;  open (Q);  while (1) {      produce\_next (&msg);      send (Q, &msg); } | message msg;  open (Q);  while (1) {      receive (Q, &msg);      consume\_next (&msg); } |

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

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

**Prioritäten:** In manchen Systemen können Nachrichten mit Prioritäten versehen werden. Der Empfänger holt dann immer die Nachricht zuerst aus der Queue, die die höchste Priorität aufweist.

**POSIX Message-Passing:** POSIX bietet Message-Passing mit Indirekte Kommunikation über Message-Queues die im OS existieren, Variable Nachrichtenlänge, deren Maximum je Queue einstellbar ist, Synchrone/Asynchrone Verwendung und Prioritäten

**Mq\_open:** Öffnet einen Message queue mit Systemnamen.Gibt einen Message Queue Descriptor zurück. Flags können 0\_RDONLY (read only), 0\_RDWR (read and write), 0\_CREAT (creates queue if not exist), 0\_NONBLOCK (send/receive don’t block)

mqd\_t mq\_open (const char \*name, int flags, mode\_t mode, struct mq\_attr \*attr);

**Mq\_close:** Schliesst Queue mit Descriptor Queue für diesen Prozess

int mq\_close (mqd\_t queue);

**Mq\_unlink:** Entfernt Queue mit dem Namen im System. Name wird soft entfernt während Queue geschlossen wird sobald alle Prozesse den Queue geschlossen haben

int mq\_unlink (const char \*name);

**Mq\_send:** Sendet Nachricht an Adresse mit Länge in die Queue. Blockt nur wenn Queue voll ist (auch mit 0\_NONBLOCK)

int mq\_send (mqd\_t queue, const char \*msg, size\_t length, unsigned int priority);

**Mq\_receive:** Kopiert nächste Nachricht von Queue in den Buffer welcher in der Adressmessage existiert.

int mq\_receive (mqd\_t queue, const char \*msg, size\_t length, unsigned int \*priority);

**Beispiel:**

#include <errno.h>, <mqueue.h>, <stdio.h>, <string.h>

int main(int argc, char\*\* argv) { // Server code

mqd\_t queue = mq\_open(argv[1], O\_RDONLY | O\_CREAT, S\_IRWXU, 0);

  if(queue == -1) { perror(strerror(errno)); }

  unsigned int prio;

  unsigned int prio2;

  struct mq\_attr def;

  mq\_getattr(queue, &def);

  char buf1[def.mq\_msgsize];

  char buf2[def.mq\_msgsize];

  mq\_receive(queue, buf1, def.mq\_msgsize, &prio);

  mq\_receive(queue, buf2, def.mq\_msgsize, &prio2);

  if (prio < prio2) {

    printf("Message: %s\n with prio: %d\n", buf1, prio);

    printf("Message: %s\n with prio: %d\n", buf2, prio2);

  } else {

    printf("Message: %s\n with prio: %d\n", buf2, prio2);

    printf("Message: %s\n with prio: %d\n", buf1, prio);

  }

  mq\_close(queue);

  mq\_unlink("mqueue");

}

int main(int argc, char\*\* argv) { // Client code

  mqd\_t queue = mq\_open(argv[1], O\_CREAT | O\_WRONLY , S\_IRWXU, 0 );

  if(queue == -1) { perror(strerror(errno)); }

  char msg[] = "thisbemessage";

  char msg2[] = "higherprio";

  mq\_send(queue,msg, sizeof(msg), 32);

  mq\_send(queue,msg2, sizeof(msg2), 1);

  mq\_close(queue);

  mq\_unlink("mqueue"); }

## shared memory

Frames des Hauptspeichers werden zwei (oder mehr) Prozessen P1 und P2 zugänglich gemacht. In P1 wird Page V1 auf einen Frame F abgebildet und In P2 wird Page V2 auf denselben Frame F abgebildet. Beide Prozesse können beliebig auf dieselben Daten zugreifen. Bequem, aber im Normalfall zusätzliche Synchronisation nötig.

**Pointer:** Pointer, die Speicherzellen ausserhalb von Shared Memory referenzieren, sind für andere Prozesse wertlos. Nur Adressen innerhalb von Shared Memory verwenden

**Relative Adressen:** Sollen Pointer verwendet werden, müssen diese relativ zu einer Anfangsadresse sein, z.B. als Offset bezogen auf eine Start-Adresse oder Index in ein Array.

**POSIX API:** Das OS benötigt ein spezielles Objekt S, das Informationen über den gemeinsamen Speicher verwaltet. S wird in POSIX wie eine Datei verwendet.

**Shm\_open:** Öffnet Shared Memory mit einem Systemnamen. Gibt einen File Deskriptor zurück. Flags kann 0\_RDONLY (read only), 0\_RDWR (read write) oder 0\_CREAT (create shared memory if it doesn’t exist) sein

int shm\_open (const char \*name, int flags, mode\_t mode);

**ftruncate:** Setzt Grösse eine Datei des File Deskriptors.

int ftruncate (int fd, offset\_t length);

**close:** Schliesst file. Shared memory bleibt im System auch wenn kein Prozess den Shared Memory verwendet.

int close (int fd);

**shm\_unlink:** Setzt Flag zum Löschen von Shared Memory. Wird erst gelöscht wenn kein Prozess den mehr verwendet.

int shm\_unlink (const char \*name);

**mmap:** Mappt den Shared memory, mit dem der Fle-Deskriptor geöffnet wurde, in den virtuellen Adressraum des aktuell

laufenden Prozesses ab und gibt die virtuelle Adresse des ersten Bytes

void \* mmap (void \*hint\_address, size\_t length, int protection, int flags, int file\_descriptor, off\_t offset);

void \* address = mmap (0, size\_of\_shared\_memory, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd, 0);

**munmap:** Entfernt das vorherig gesetze Mapping

int munmap (void \*address, size\_t length);

**Vergleich Message Parsing/Shared Memory:**

|  |  |
| --- | --- |
| Shared Memory | Message Parsing |
| Shared Memory ist oft der schnelle zu realisierende Ansatz | Message-Passing erfordert mehr Engineering-Aufwand |
| Existierende Applikationen mit einem Prozess können relativ schnell auf mehrere Prozesse mit Shared Memory umgeschrieben werden | Existierende Anwendungen müssen häufig in grossen Teilen neu implementiert werden |
| Oft werden solche Programme aber schwer wartbar | Bei sauber gekapselten Anwendungen ist das aber ein sehr viel geringeres Problem |
| Oft besteht (bestand) die Gefahr, dass Resultate wegen impliziter Abhängigkeiten zwar nebenläufig aber nicht echt parallel sind | Anwendungen mit Message-Passing lassen sich sehr leicht so erweitern, dass sie als verteilte Systeme arbeiten |
| Das System ist weniger stark modularisiert und Prozesse sind schlechter gegeneinander geschützt | Bei einigen OS ist das sogar schon integriert, z.B. QNX |

**Welche Variante performanter:** Auf Einzel-Prozessor-Systemen im Normalfall Shared-Memory wegen der entfallenden Kopieroperationen. Auf Mehr-Prozessor-Systemen benötigen Shared-Memory-Systeme zusätzlichen Aufwand in der Hardware. Message-Passing-Systeme liegen auf Mehr-Prozessor-Systemen häufig gleichauf und werden in Zukunft vermutlich sogar performanter als Shared-Memory-Systeme sein.

**Vergleich Message Queues/Pipes:**

|  |  |
| --- | --- |
| **Message Queues** | **Pipes** |
| Message-Queues sind bidirektional | Pipes sind unidirektional |
| Daten in Message-Queues sind in einzelnen Messages organisiert | Pipes übermitteln einen Bytestrom an Daten |
| Message-Queues beliebigen Zugriff (random access) | Pipes erlauben nur FIFO-Zugriff |
| Message-Queues haben immer einen Namen. | Pipes müssen nicht zwingend einen Namen im Dateisystem haben (es gibt aber named pipes) |

# unicode / ext2 / ext4

## Unicode

**Unicode:** Definiere eindeutigen Code für jedes vorhandene Zeichen, auch ausgestorbene Zeichen und Emoji. Maximal 1.1M Code-Points. D800 bis DFFF ist für keine Codierung gültige Code Points, da diese in UTF-16 nicht gültig sind.

**Verschiedene Encodings:** Unterscheidet zwischen Code-Points (Nummer eines Zeichens) und Code-Units (Einheit, um Zeichen in einem Encoding darzustellen).

**UTF-32:** 32 Bit pro CU. Jeder CP kann in einer CU dargestellt werden. Häufig intern in Programmen verwendet. Bei byteweisen Übertragungen muss Endianness beachtet werden. Little-Endian: U0 = B0B1B2B3. Big-Endian: U0 = B3B2B1B0

**UTF-8:** 8 Bit pro CU. Encoding muss Endianness nicht berücksichtigen. CP benötigt bis zu 4 CU. Standard für Webpages. Echte Erweiterung von ASCII.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Bits | Last code point | Byte 1 | Byte 2 | Byte 3 | Byte 4 |
| 7 | U+007F (127) | 0xxxxxxx |  |  |  |
| 11 | U+07FF (2’047) | 110xxxxx | 10xxxxxx |  |  |
| 16 | U+FFFF (65’535) | 1110xxxx | 10xxxxxx | 10xxxxxx |  |
| 21 | U+1FFFFF (2’097’151) | 111110xx | 10xxxxxx | 10xxxxxx | 10xxxxxx |

**UTF-16:** 16 Bit pro CU. Encoding muss Endianness berücksichtigen

1. **Get Codepoint:** (this is a tako)🐙= 128’025 (0x1F419)
2. **Subtract 65'536 (0x10000):** 0x1F419 - 0x10000= 0x0F419
3. **Convert to binary and fill left with 0 till 20 digits:** 0000 1111 0100 0001 1001
4. **Split in half and convert back to hex:** 0000 0011 1101 (3D) and 0000 0001 1001 (19)
5. **Add surrogate pairs to both:** D800 + 3D = D83D and DC00 + A9 = DC19
6. **Convert back to binary:** D83D = 1101100000111101 and DC19 = 1101110000011001

**Encoding Beispiele:**

A picture containing text, font, screenshot, number

Description automatically generated

## ext2-dateisystem

**Datenträger Begriffe:**

* Partition: Teil eines Datenträgers, welcher als solches behandelt wird
* Volume: Ein Datenträger oder eine Partition davon
* Sektor: Kleinste logische Untereinheit eines Volumes; Daten werden als Sektoren transferiert. Enthält Header, Daten und Error-Correction-Codes
* Format: Layout der logischen Strukturen auf dem Datenträger. Wird vom Dateisystem definiert

**Block:** Mehrere aufeinanderfolgendeSektoren (normal 4KB). Das gesamte Volume ist in Blöcke aufgeteilt. 1.Sektor eines Blocks = Blocknummer. Speicher wird nur in Form von Blöcken alloziert. Block enthält nur Daten einer einzigen Datei.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Volume | Block 0 | Sektor 0 |  | File | | Block 0 | |  | | Volume | | … |
| Sektor 1 |  | Block 1 | |  | | Block 10 |
| … |  | Block 2 | |  | | … |
| Sektor 7 |  | Block 3 | |  | | Block 21 |
| Block 1 | Sektor 8 |  | … | |  | | Block 22 |
| Sektor 9 |  |  | |  | |  | | … |
| … |  |  | |  | |  | | Block 31 |
| Sektor 15 |  |  | |  | |  | | … |
| … | |  | |  | |  | |  | |

Logische Blocknummer ist die Nummer vom Anfang der Datei ausgesehen, wenn Datei eine ununterbrochene Abfolge von Blöcken wäre (= aus Sicht der Applikation). Physische Blocknummer ist die tatsächliche auf dem Volume.

**Inodes (Index Nodes):** Enthält alle Metadaten über die Datei, ausser Namen oder Pfad. Grösse, Anzahl der verwendeten Blöcke, Erzeugungs-, Zugriffs-, Modifikations- und Löschzeit. Fixe Grösse je Volume (Zweierpotenz, mind. 128 Byte und max. 1 Block)

|  |  |  |
| --- | --- | --- |
| **Inode** |  |  |
| Metadata about File |  | Data Block |
| References to File’s Data Blocks |  | … |
|  |  | Data Block |

Inodes verweist auf die Blöcke, die Daten für die Datei enthalten. Enthält ein Array i\_block mit 15 Einträgen je 32 Bit. 12 BlockNr für die ersten 12 Blöcke, 1 BlockNr des indirekten Blocks, 1 BlockNr des doppelt ind. Blocks und 1 BlockNr des dreifachen ind. Blocks

**Indirekter Block:** Enthält 32-Bit Nummern der Blöcke der Datei ab Block 12. Anzahl der referenzierten Blöcke hängt von Blockgrösse ab.

Blockgrösse 1024 Bytes = 256 BlockNr: Block 12 bis 267.

Blockgrösse 4096 Bytes = 1024 BlockNr: Block 12 bis 1035

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | Indirect Block | Index 0 |  | LB 12 | LB = | Logical Block |
| Index 12 |  | Index 1 |  |  | LB 13 |  |
|  |  | Index 2 |  |  |  |  |
|  |  | … |  |  |  | LB 14 |

**Doppelt indirekter Block:** Enthält Nummern von indirekten Blöcken. Anzahl dadurch referenzierter Blöcke hängt vom Quadrat der Blockgrösse ab.

Blockgrösse 1024 = 256 Einträge je Block, 256 Blöcke = 65536 Blöcke referenziert

Blockgrösse 4096 = 1024 Einträge je Block, 1024 Blöcke = 1M Blöcke referenziert

**Dreifach indirekter Block:** Enthält Nummern von doppelt indirekten Blöcken.

**Blockgruppe:** Volume wird in Blockgruppen unterteilt. Eine Blockgruppe sind mehrere aufeinanderfolgende Blöcke (bis zu 8x der Anzahl Bytes in einem Block). Blöcke je Gruppe gleich für alle Gruppen. Bsp: Bei Blockgrösse 4 KB sind bis zu 32 K Blöcke in einer Gruppe.

**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
* Block n + 2: Inode-Usage-Bitmap mit einem Bit je Inode der Gruppe
* 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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Block Group | | | | | | |
| Super Block (Copy) | Group Descriptor Table (Copy) | Block Usage Bitmap | Inode Usage Bitmap | Inodes | File Data | … |

**Superblock:** Enthält alle Metadaten über das Volume u.a Anzahl Inodes frei/gesamt, Blöcke frei/gesamt, Bytes je Block/Inode/Gruppe, Inodes je Gruppe, Verschiedene Zeitpunkte, Verschiedene Statusbytes für Fehlerkennung, Erster Inodes, Feature-Flags. Startet immer an Byte 1024 (wegen eventuellen Bootdaten im Bereich vorher)

**Gruppendeskriptor:** 32 Byte Beschreibung einer Blockgruppe. BlockNr des Block-Usage Bitmaps und Inode-Usage-Bitmaps. Nummer des ersten Blocks der Inode-Tabelle. Anzahl freier Blöcke/Inodes/Verzeichnisse in der Gruppe

**Gruppendeskriptortabelle:** Ein Gruppendeskriptor je Gruppe. Tabelle mit n Gruppendeskriptoren für alle n Blockgruppen im Volume. Benötigt selbst 32 x n Anzahl Bytes; Anzahl Sektoren = 32 · n / Sektorgrösse. Folgt direkt auf Superblock. Kopie der Tabelle direkt nach jeder Kopie des Superblocks

**Sparse Superblocks:** Feature, das die Anzahl der Superblocks stark reduziert. Wird über ein bestimmtes Flag im Superblock 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. Dadurch immer noch sehr hohe Wiederherstellungsgrad möglich.

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

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Blocksize 1 KB | |  |  | Blocksize 2 KB | |  |  | Blocksize 4 KB | |
|  | Block 0 | |  | Block Group 0 | Block 0 |  |  | Block Group 0 | Block 0 | |
| Block Group 0 | Block 1 | Super Block |  | Super Block |  |  | Super Block |
| Bock 2 | |  | Block 1 | |  |  |  |
| Block 3 | |  |  |  | |
| … | |  | … | |  | … | |

**Inodes:** Lokalisierung eines Inodes. Alle Inodes aller Blockgruppen gelten als eine grosse Tabelle. Zählung der Inodes startet mit 1. Blockgruppe = (Inode - 1) / «Anzahl Inodes pro Gruppe»

A picture containing text, font, screenshot, number

Description automatically generated

**Erzeugen von Inodes:** Finden einer geeigneten Blockgruppe (optimiert). Neue Verzeichnisse werden bevorzugt in der Blockgruppe angelegt, die von allen Blockgruppen mit überdurchschnittlich vielen freien Inodes die meisten freien Blöcke hat. Dateien werden möglichst in der Blockgruppe des Verzeichnisses angelegt, oder in nahen Gruppen. Bestimmen des ersten freien Inodes in der Gruppe anhand des Inode-Usage-Bitmaps.

**File-Inodes:** 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. Analog zu NTFS Sparse Files

**Verzeichnisse:** Verzeichnis enthält Dateieinträge. Genauer: Inode, dessen Datenbereich Dateieinträge enthält. Zwei automatisch angelegte Einträge: «.»: Dateieintrag mit eigenem Inode (Verweis auf selbst). «..»: Dateieintrag mit Inode des Elternverzeichnisses (Verweis auf Elternverzeichnis). Wurzelverzeichnis ist Inode Nummer 2

A diagram of entry block

Description automatically generated with low confidenceA picture containing text, screenshot, font, line

Description automatically generated

**Links:** Hard-Link: gleicher Inode, verschiedene Pfade. Inode wird von verschiedenen Dateieinträgen (auch in verschiedenen Verzeichnissen) referenziert. Symbolischer Link: wie eine Datei. Datei enthält Pfad anderer Datei. Wenn Pfad < 60 Zeichen, dann wird der Pfad direkt in dem 15-Einträge grossen Array der Blockreferenzen gespeichert und keine Blöcke werden alloziert. Ansonsten wird genau ein Block alloziert (Pfad-Länge < Blocklänge)

## EXT-4 DATEISYSTEM

**Datenstrukturen vergrössert:**

* Inodes haben mind. 256 Byte statt 128 und verwalten Blöcke mit Extent Trees statt Array
* Gruppendeskriptoren haben 64B statt 32
* Blockgrösse bis 64KB statt 4KB

Daten werden erst ins Journal an dann an tatsächliche Position geschrieben.

A diagram of a physical block

Description automatically generated with medium confidence**Extent:** Ein Extent beschreibt ein Intervall physisch konsekutiver Blöcke. Besteht aus 12 Bytes aus 4B logische BlockNr, 6B physische BlockNr und 2B Anzahl Blöcke (signed). Positive Zahlen bedeuten initialisierte Blöcke. Negative Zahlen bedeuten vorallozierte Blöcke.

Eine Einschränkung auf ausschliesslich konsekutive Dateien ist nicht praktikabel. Eine Datei muss mehr als einen Extent umfassen können. Im Inode in den 15 · 4 = 60 Byte für direkte und indirekte Block-Adressierung Platz für 4 Extents und einen Header.

**Extent Tree Header:** Für mehr als 4 Extents braucht man einen zusätzlichen Block. Dieser befindet sich in den ersten 12B. 2B Magic Number, 2B Anzahl Einträge die nach Header folgen, 2B Anzahl Einträge die maximal auf den Header folgen können, 2B Tiefe des Extent Tree (0: Einträge sind Extents, 1: Einträge sind Index Nodes) und 4B sind reserviert

**Index Nodes:** Spezifiziert einen Block, der Extents enthält. 4B gibt kleinste logische BlockNr aller Kind-Extents an, 6B physische BlockNr des Blocks, auf den der Index-Node verweist und 2B unbenutzt. Der Block enthält am Anfang einen Header (wie im Inode, aber Tiefe=0), danach Extents (max 340 bei 4KB Blockgrösse)

**Blöcke mit Index Nodes:** Benötigt man mehr als 4 · 340 = 1360 Extents, 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 BlockNr 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

**Extent Trees:** Index-Knoten (innerer Knoten des Baums). Wird in allen Ebenen ausser der untersten verwendet. Besteht aus Index-Eintrag (12 Bytes) und Index-Block. Index-Eintrag. Enthält Nummer des physischen Index-Blocks. Enthält kleinste logische Blocknummer aller Kindknoten. Vereinfacht die Suche nach logischen Blocknummern im Baum.

A picture containing text, screenshot, font, line

Description automatically generated

## journaling

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

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

**Journal:** Reservierte Datei, in die Daten schnell geschrieben werden können. Folge von Einzelschritten, die das Dateisystem gesamtheitlich vornehmen soll.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Volume | Journal | Metadata and File Data | | | |
|  |  |  | | | |
| Journal | Transaction | Transaction | Transaction | Transaction | … |

**Journaling/Committing:** Daten werden zuerst als Transaktion ins Journal geschrieben (Journaling), danach erst an ihre endgültige Position geschrieben (Committing). Daten werden nach dem Commit aus dem Journal entfernt. Journaling schneller, weil alle Daten in konsekutive Blöcke geschrieben werden. Committing muss viele verschiedene Blöcke modifizieren

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Journaling | |  | Committing |  |
|  | |  |  |  |
| Volume | Journal | | Metadata and File Data | |

**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. Journal Replay führt diese Transaktionen aus oder stellt Fehler fest.

**Journaling Modi:** Modi Ordered und Writeback schreiben nur Metadaten. Modus Journal schreibt auch Datei-Inhalte ins Journal.

**Modus Journal:** Metadaten und Datei-Inhalte kommen ins Journal. Maximale Datensicherheit, jedoch grosse Geschwindigkeitseinbussen

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| File | Metadata | File Content | | |  | |
|  |  | |  | |  | |
|  | Journaling | |  | Committing | |  |
|  |  | |  | |  | |
| Volume | Journal | | Metadata and File Data | | | |

**Modus Ordered:** Nur Metadaten kommen ins Journal. Dateiinhalte werden immer vor dem Commit geschrieben. Daten enthalten nach dem Commit den richtigen Inhalt, jedoch etwas geringere Geschwindigkeit als Writeback und sehr lange Blockaden bei viel Transaktionen.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| File | Metadata | File Content | | |  | |
|  |  | |  | |  | |
|  | Journaling | |  | Committing | | 2 |
|  | 1 | |  | | 3 3 | |
| Volume | Journal | | Metadata and File Data | | | |

**Modus Writeback:** Nur Metadaten kommen ins Journal. Commit und Schreiben der Dateiinhalte werden in beliebiger Reihenfolge ausgeführt. Sehr schnell da keine Synchronisation zwischen Commit/Datenschreiben nötig, jedoch können Dateien Datenmüll enthalten

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| File | Metadata | File Content | | |  | |
|  |  | |  | |  | |
|  | Journaling | |  | Committing | | 2 |
|  | 1 | |  | | 3 2 | |
| Volume | Journal | | Metadata and File Data | | | |

# x window system

## gui basiskonzepte

**Moderne Unix Systeme:** GUI mittels des X Window System. Setzt auf Unix-Systemkern auf. Alternativ auch Google Android, Apple Aqua, Canonical Mir Display Server, Wayland

**Ereignisgesteuerte Interaktion:** Programmierer legt Reaktionen fest, wartet auf Ereignisse und reagiert darauf, Benutzer entscheidet wann welches Ereignis ausgelöst wird.

**Vorteile:** Auf Unix-Kern aufgesetzt und damit austauschbar. Installierbar wenn tatsächlich benötigt. Realisiert Netzwerktransparenz, d.h. Clients «sehen» nicht, wo ihr GUI läuft.

**Entstehung:** In den 80er Jahren für Athena-Projekt (von MIT und DEC) entwickelt. Ziel ist Hardware-Unabhängigkeit, Portabilität und geeignet für mittels Netzwerke (TCP/IP) verbundene verteilte Systeme. Verwandtes Project ist VNC

**Fenster:** Rechteckiger Bereich des Bildschirms und kann beliebig viele weiter Fenster enthalten. Baumstruktur aller Fenster. Jedes Fenster hat 0..n Unterfenster und genau 1 Parent. Bildschirm ist die Wurzel und hat kein Elternfenster.

**Maus:** Physisches Gerät das 2D-Bewegung in Daten übersetzt. Kleine Rastergrafik, die auf dem Bildschirm angezeigt wird. Maustasten lösen Ereignisse aus.

**Verknüpfung von Maus/Fenster:** Maustreiber erzeugt Nachrichten (z.B Klick an Pos x, y). OS verteilt Nachrichten, muss aber das Fenster an Position x, y ermitteln. Asynchron

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Mouse Driver |  | OS |  | Window Owner |

**GUI-Architektur:** GUI braucht mehr als X Window System

* X Window System: Grundfunktionen der Fensterdarstellung
* 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. Viele verschiedene Implentationen von Window- und Desktop Manager existieren.

**Window-Manager:** Läuft im Client und realisiert eine Window Layout Policy. Platziert passend Client-Fenster auf dem Bildschirm. Clients geben Window Manager Hinweise, wie sie angezeigt werden sollen. WM kann diese akzeptieren, modifizieren oder ignorieren.

Zusätzliche Dienste des Window Manager für den Benutzer sind Applikationsfenster sind Fenster verschieben, Grösse ändern, minimieren/maximieren, neue App. starten etc.

**Fensterverwaltung:** Fensterhierarchie ist Root-Window zuoberst, bedeckt ganzen Bildschirm. Kinder des Root-Window sind Top-level Windows der Applikation. Übrige Fenster sind zur Anzeige von Menü, Button usw. in Applikationen. Kindfenster können Elternfenster teilweise überlappen, jedoch Anzeige des Kinds nur innerhalb des Elternfenster. Eingaben werden nur im Überlappungsbereich empfangen.

## x window system basiskonzept

**Terminologie:**

* Display: Rechner mit Keyboard, Zeigegerät und 1..n Screens. Bild kann über mehrere Screens verteilt sein
* X Server: Softwareteil des X Window System, der ein Display ansteuert
* X Client: Applikation, die ein Display nutzen will

Clients können lokal oder entfernt laufen. X-Server läuft stets auf dem Rechner.

**Xlib:** C Interface für das X Protocol. Zahlreiche Funktionen/Datentypen welche meist nicht verwendet werden.

**X-Toolkits:** Direkt Xlib verwenden ist aufwendig da Low-Level-Programming.X Toolkits stellen eine Software-Schicht oberhalb der Xlib dar (komfortabler). Stellt Widgets fertig zur Verfügung, z.B. Command Buttons, Labels und Menüs. Beliebte Toolkits sind Qt, GTK+

**Verbindung zum Display:** Um ein Display zu verwenden, muss eine Verbindung zu diesem bestehen. Die Verbindung zum Display wird im Datentyp Display gespeichert

## event handling

**X Protocol:** Legt die Formate für Nachrichten zwischen X Client und Server fest

* Requests: Dienstanforderungen, Client → Server
* Replies: Antworten auf bestimmte Requests (viele Requests ohne Replies), Client ← Server
* Events: spontane Ereignismeldungen, Client ← Server
* Errors: Fehlermeldungen auf vorangegangene Requests, Client ← Server

**Nachrichtenpufferung Request:** Für Requests gibt es einen Nachrichtenpuffer auf **Client-Seite** (Request Buffer). Spart Kommunikationsoverhead, wenn einige Request gesammelt werden. Pufferleerung wenn Client auf Event wartet, Client-Request einen Replay des Server benötigt oder wenn Client dies explizit verlangt (z.B Aufruf XFlush())

**Nachrichtenpufferung Events:** Ereignisse (Event) werden doppelt gepuffert (**Server und Client**). Server-seitige Pufferung berücksichtigt Netzwerkverfügbarkeit, Client-seitige Pufferung hält Events bereit, bis von Client abgeholt, Client liest Messages in Message-Loop mittels Funktion XNextEvent().

A cartoon gnome with a red hat and white beard

Description automatically generated**X Event Handling:** Ereignisse werden von Client verarbeitet oder weitergeleitet. Client muss vorher festlegen, welche Ereignistypen er empfangen will (XSelectInput()).Selektion pro Fenster individuell. Nur selektierte Ereignistypen werden dem Client zugestellt.

while (1) {

    XNextEvent(display, &event); switch (event.type) {

        case Expose: // Event: verlangt Neuzeichnen des Fensters

        ...

        break;

    case KeyPress: // Event: Taste wurde gedrückt

GNOME or something idk,

I use Hyprland

        ...

        if (...) exit(0);

    break;

    }

}

Events sind vom Typ XEvent. C-Union über alle Event-Typen, d.h. so gross wie der grösste Event-Typ. 33 verschiedene Event-Typen unterschiedlicher Grösse. Jeder Event-Typ ist ein struct, der als erstes den int «type» enthält.

typedef union \_XEvent {

    int type; // must not be changed; first element

    XAnyEvent xany; XKeyEvent xkey;

    XButtonEvent xbutton; /\* ... \*/

} XEvent;

## zeichnen

A diagram of a server

Description automatically generated with low confidence**Ressourcen:** X Ressourcen sind Server-seitige Datenhaltung zur Reduktion des Netzwerkverkehr. Halten Informationen im Auftrag von Clients. Clients identifizieren Informationen mit zugeordneten Nummern (IDs). X Ressourcen können Windows, Pixmap, Colormap, Font und Graphics-Context sein.

**X Grafikfunktionen:** Grafikgrundfunktionen erlauben das Zeichnen von Geometrischen Figuren, Strings und Texten

* Bilddarstellung mittels Rastergrafik und Farbtabelle
* Schwarz/weiss: Genau ein Bit pro Bildpunkt
* Farben oder Grautöne: Mehrere Bits pro Bildpunkt. Keine direkte Farbzuordnung zu Binärwerten. Index in eine Farbtabellen
* Vorteil: Reduktion der Bits pro Farbe von n auf m (n ist die Anzahl Bits pro absolut darstellbare Farbe und m ist die Anzahl Bits pro gleichzeitig darstellbare Farbe)

**Graphic Context:** Grafikgrundfunktionen benötigen stets einen Graphics Context (X Ressource). Legt div. Eigenschaften fest, die Systemaufrufe nicht direkt unterstützen (z.B. Liniendicke, Farben, Füllmuster). Client muss erst GC anlegen vor Aufruf einer Zeichenfunktion und es können mehrere GC gleichzeitig genutzt werden.

## fenster schliessen

**Window Manager Events:** Die Schaltfläche zum Schliessen eines Fensters wird vom Window Manager erzeugt. X weiss nichts über die spezielle Bedeutung dieser Schaltfläche. Man hat dazu ein Protokoll zwischen Window Manager und Applikation definiert. Window Manager sendet ein ClientMessage Event an Applikation. Dieses Event muss in seinem data-Teil die ID eines Properties WM\_DELETE\_MESSAGE enthalten.

**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

**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. Ein Property wird über ein Atom identifiziert und das Atom wird jeweils über einen wohl-definierten String ermittelt.

**Property WM\_PROTOCOLS:** Der X Standard definiert eine Anzahl an Protokollen, die der Window Manager verstehen soll. Ein Client kann sich für Protokolle registrieren.

A cartoon of a child with long hair and ears

Description automatically generated with low confidence**A picture containing clipart, graphics, illustration, graphic design

Description automatically generatedProtokoll WM\_DELETE\_WINDOW:** Window Manager schickt Event an Client, wenn man auf den Close-Button drückt. Event-Typ ist ClientMessage, wird immer vom Client verarbeitet. Im Datenteil des Events steht das Atom von WM\_DELETE\_WINDOW

I was forced to put this in

**We chose to be lost. It’s our lifestyle.**