# Betriebssystem API

## Grundlegende Aufgaben eines Betriebssystems

* Abstraktion / Portabilität von HW, Protokollen, SW-Services
* Resourcenmanagement / Anwendungsisolation|Prozesse, Speichertrennung,etc
* Benutzerverwaltung / Security

## Portierbarkeit + Isolation

Portierbarkeit: Hersteller zuständig, seine Applikation auf versch. Systemen läuft

Modere OS bieten Mechanismen an, obliegt Applikation diese zu verwenden

Mechanismen z.B Touchscreen, Bildschirme, Maus, Touchpad, Tastatur, etc.

OS entscheidet nicht was Applikation meint

Isolierbarkeit: Jede App will Fokus, Problem Rennen um Bildschirm & Tastatur

Problem bei verschiedenen Programmen, z.B Passwort-Safe fängt Tastenkombi ab

## Prozesser Priviledge Level

Isolation von Applikationen voneinander. Proz. benötigt mind. 2 Privlege Levels.

* Kernel Mode: alle Instruktionen,
* User-Mode: beschränkte Anz. Instruktionen(stark eingeschränkt)

OS im Kernel Mode entscheidet welche SW in welchem Mode.

OS muss Hardwarelevels softwaretechnisch verwalten 🡪 sichere App-Isolation

***Aufteilung OS***: Kernel(Komponenten im privilegierten Modus ) + Rest(User-Mode)  
Wechsel User 🡪 Kernel Mode auf Intel 64 Proz. via ***syscall***. Prozessor schaltet in Kernel Mode, setzt IP auf OS-Code, System Call Handler. Gewährleistet nur Kernel-Code im Kernel-Mode. Jede OS-Kernel-Funkt. Hat einen Code für syscall. Übergabe in Register, Parameter in anderen Registern. 🡪 ***ziemlich sicher***

## Kernel Varianten

|  |  |
| --- | --- |
| Microkernel | Monolithische Kernel |
| Kernel minimal, nur Kritischen in Kenel-Mode(sogar Treiber in User-Mode(Theoretisch nicht praktisch)) | Wie wir gewohnt sind: viele Funktionalitäten, können auch ausserhalb von Kernel-Mode laufen |
| + sehr stabil, fast keine Abstürze | + Perfomance(sehr schnell) durch minimale Anzahl Modi-Wechsel |
| + gut analysieren dank wenig Code |
| - Perfomance-Einbussen durch viele Modi-Wechsel(immer Umschalten) | - weniger Schutz durch Programmierfehler(Sicherheit, Abstürze, Nachvollziehbarkeit) |

## Syscall, Posix & Shell

***Syscall*** code in spezielles Register mit verschiedenen Parametern z.B rsi 60 --> exit

Keine Binärkompatibilität unter verschiedenen Linux-Kernel(gibt Bestreben dazu)

🡪 ***Wrapper Funktionen*** in C, sichergestellt, zum Kernel passende Binär-Code generiert ohne dass der C-Code geändert werden muss z.B exit(code) für alle Kern

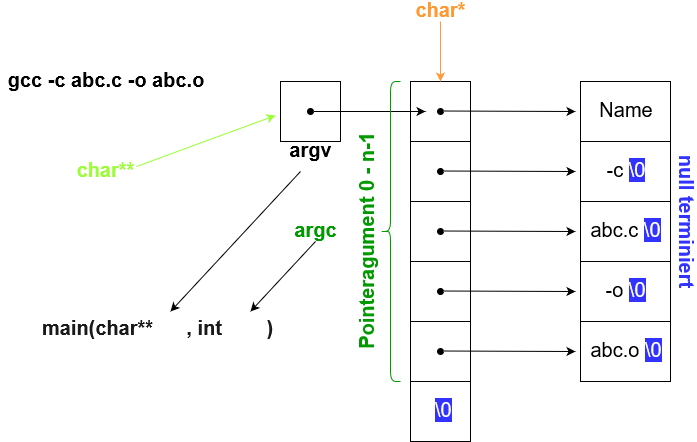
### Posix Standardisierte API, welch in Linux fast zu 100% eingesetzt

### Shell Programm, das erlaubt über Texteingabe Betriebssystemaufrufe aufzuführen

## Programmargumente & Calling Convention

Shell zuständig, Programmargumente in Strings aufzuteilen(z.B durch Leerzeichen)

Inhalt Argumente interessieren OS nicht sondern werden von App interpretiert



- Bevor main aufgerufen, OS legt Programmargumente in den Speicherbereich vom Prozess.

- Os legt Array für Argumente an- siehe links

## Umgebungsvariabeln

Menge an Strings, welche mindestens ein = enthalten. Key=value 🡪key unique

SHELL=/bin/bash oder USER=dese oder GJS\_DEBUG\_TOPICS=JS ERROR;JS LOG

Ein Bild, das Screenshot enthält.

Mit sehr hoher Zuverlässigkeit generierte BeschreibungUnter Posix verwaltet OS Umgebungsvariabeln innerhalb jedes laufenden Prozess

# Prozesse

## Grundlagen

Monoprogrammierung: Prozessor führt nur 1 Programm aus. Kommunikation über C Funktionen (2 Akteure: OS, Programm).

Quasi-Parallele Ausführung: viele Programme gleichzeitig laufen lassen 🡪 alle Programme gleichzeitig im Hauptspeicher, OS weisst App nacheinander Zeit auf dem Prozessor zu, Verwaltungseinheit für Programme = Prozess

### Man will das Prinzip «Monoprogrammierung» behalten:

* OS bietet seine Dienste für jede (einzelne) Applikation an
* OS isoliert verschiedene Programme voneinander(also wäre es Mono)
* OS kapselt Prozesse/Programme 🡪 jeder Prozess eigener virt. Adressraum

### Ein Bild, das Screenshot enthält. Mit sehr hoher Zuverlässigkeit generierte BeschreibungProzess allgemein

Abbild eines Programms im HS (text section), Globalen Variablen (data section),

Speicher für den Heap(von gross nach klein) und Stack (wachsen gegeneinander)

### Programm(passiv - Theaterstück) vs Prozess(aktiv - Theateraufführung)

Programm beschreibt Abläufe und Prozess führ diese aus. Programm mehrfach ausgeführt werden, es handelt sich dann um verschiedene, unabhängige Prozesse

## ***Process Control Block(PCB) & Interrupt & Kontext wechsel***

Grundprinzip: Repräsentation des Prozesses auf OS Ebene, Speicher für OS-benötigte Daten 🡪 grosse Menge von Metainfos

### Interrupt: Wenn ein Interupt auftritt werden folgende Dinge gemacht:

### Kontexsicherung(context save) in PCB[Register, Flags, IP, Page-Table-Config]

1. Interrupt Handler: Kann Kontext überschreiben.
2. Nach Ende Interrupt-Handlers: Kontext wiederhergestellt(context restore)

### Kontext-Wechsel: Wenn zwischen Prozessen gewechselt werden soll/muss

von A nach B 🡪 Interrupt Prozess A, Interrupt Handler, context-restore Prozess B

sind teuer: Register, Flags, IP & Floating Pointer, Cache laden(virt. Adressen)

## Erzeugen von Prozessen & Prozess-Hierarchie

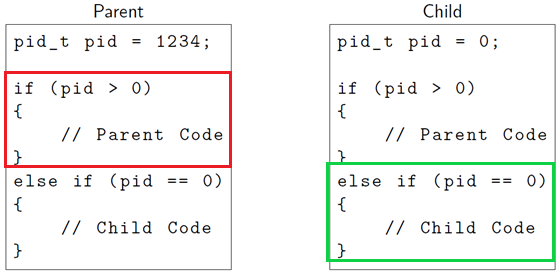
Erzeugen 1. Prozess erzeugen, 2. Image des Programms in diesen Prozess laden

Prozess-Hierarchie: Ein Parent-, beliebig viele Child-Prozesse, root = ***Quellprozess***

## Funktionen

### pid\_t fork(void) – Prozess wird kopiert(identisch, ausser Prozess- & Parent-ID)

Prozesse gleicher Instruction Pointer 🡪 laufen synchron

Rückgabe🡪In P bei Erfolgt Child-Prozess-ID(>0) oder Fehler(<0) bei C return = 0

Code gleich aber wegen PID(Rückgabe) andere Code-Teile ausgeführt

pid\_t wait (int \*status) - Unterbricht aufrufender Prozess bis irgendein Child-Prozess beendet wurde

### Ein Bild, das Screenshot enthält. Mit sehr hoher Zuverlässigkeit generierte Beschreibungpid\_t waitpid (pid\_t pid, int \*status, int options) – wie wait aber gibt an, auf welchen Child Prozess gewartet werden soll(pid > 0 🡪 Childnummer)

Der Parent ist solange stillgelegt, bis der Child-prozess den syscall exit aufruft

## Zombie

Prozess ist zwischen seinem Ende und Aufruf von wait durch Parent ein Zombieprozess (tot, aber noch nicht entfernt). Ist ein Prozess über längere Zeit ein Zombie so hat P wahrscheinlich einen Fehler. Lösung: P beenden 🡪 C an pid=1

## Orphan (waise)

Wird ein Prozess P vor seinen Childs beendet, haben diese keinen Parent mehr und verwaisen 🡪 sie werden zum ***Orphan*** ***Prozess***

Ein Bild, das Karte enthält.

Mit sehr hoher Zuverlässigkeit generierte BeschreibungDa nun P kein wait() mehr aufrufen kann würden die Orphan-Prozesse zum ewigen Zombie-Prozess. Deshalb hat man den Mechanismus eingebaut, dass beim beenden eines Parent-Prozesses, alle Childs an den Quellprozess(pid=1) zugewiesen werden. Dieser hat ein wait()-Endlosschleife.

# Threads

## Kooperatives Multitasking mit Beispiel Textverarbeitung

Anwendung ruft jede Aktivität periodisch auf, diese werden zu einem Teil augeführt. Anwendung kann Aktivität nicht unterbrechen(hofft dass diese nur kleine Teilschritte macht). Absturzgefahr(durch (Programmier)fehler) schlechte

Performance bei vielen Aktivitäten

## Prozess- und Threadmodell

### Nebenläufigkeit: Pro Prozessor genau ein Prozess 🡪 man möchte leichtgewichtige Prozesse(Thread) mit gemeinsamen Adressraum weil z.T gleiche Ressourcen benutzt werden(beispielsweise gleicher Text im Word)

|  |
| --- |
| ***Prozessmodell – pro Teilaktivität ein Prozess*** |
| Jeder Prozess virtuell den ganzen Rechner zur Verfügung – grosse Probleme, wenn verschiedene Prozesse auf gleiche Daten zugreifen müssen |
| + Gut für unabhängige Apps(Sicherheit) |
| - Parallele Abläufe aufwendig  - Overhead für Prozesserzeugung zu gross bei kleinen Teilaktivitäten  - Gem. Ressourcennutzung erschwert(Datenaustausch braucht OS -> teuer) |
| ***Threadmodell*** |
| parallel ablaufende Aktivitäten innerhalb eines Prozesses  Threads haben auf alle Ressourcen im Prozess gleichermassen Zugriff  Jeder Thread eigenen Kontext +eigene Stack(Eigene Funktions-Aufrufketten) |
| + Datenaustausch erleichtert(Zugriff auf alle Ressourcen im Prozess (Code, globale variablen, heap, open files, mmu-data))  + Performance |
| - weniger Schutz vor (Programmier)fehler  - Sicherheit(weil «keine» Zugriffschutz innerhalb Prozess zwischen Threads) |

## Lebensdauer eines Threads

* Er springt aus der Funktion start\_function zurück
* Er ruft pthread\_exit auf.
* Ein anderer Thread ruft pthread\_cancel auf.
* Sein Prozess wird beendet.

## Thread Local Storage

### Fehlercodeauswertung: Funktion aufrufen, Rückgabewert auswerten, wenn dieser auf Fehler hinweist, errno auswerten

Problem bei Threads🡪 gleiche globale Varriabeln(errno eventuell modifiziert)

### TLS Mechanismus: globale Variablen per Thread zur Verfügung stellen

### Vorgehen

### Bevor Thread Erstellung

### Anlegen eines Keys, der die TLS-Variable identifiziert

### Speichern des Keys in einer globalen Variable

### Im Thread

### Auslesen des Keys aus der globale Variable

### Auslesen / Schreiben des Werts anhand des Keys über besondere Funktionen

# Scheduling

## Grundlegendes Modell & Zustände von Threads

**Verschiedene Status der Threads:**

* Running(der Thread läuft gerade auf einem Prozessor)
* Ready(Bereit um ausgeführt zu werden – in Warteschleife(Queue)
* Waiting(Kein Busy-Wait(Verschwendung) OS setzt in Status waiting)

Statusübergänge werden immer vom OS übernommen

### Ready Queue

### Powerdown-Modus: Wenn kein Thread laufbereit, schaltet OS CPU in Standby

### Arten von Threads

* I/O-lastig kommuniziert häufig mit I/O Geräten, rechnet relativ wenig
* CPU-lastig: kommuniziert kaum/gar nicht mit I/O-Geräten, rechnet fast ausschliesslich

### Nebenläufigkeit:

* Kooperativ: Jeder Threadentscheidet selbst, wann er Prozessor abgibt
* Präemtiv: Der Scheduler entscheidet, wann einem Thread der Prozessor entzogen wird(es passiert etwas: I/O-device waiting oder Interupt, etc.)

### Verschiedene Ausführungen:

* Parallel: Alle Threads laufen tatsächlich gleichzeitig(n Threads benötigt n CPU)
* Quasiparallel: Abwechselnde Ausführung von n Threads auf <n CPU🡪 erweckt den Anschein von Parallelität
* Nebebläufig: Parallel oder quasiparallel(meist die Sicht vom Programmierer)

### Bursts

* Prozessor-Burst – Intervall in dem Thread die CPU belegt(running – waiting)
* I/O-Burst – Intervall in dem Thread CPU nicht benötigt(waiting bis running)

## Scheduling-Strategien

Man strebt nach paralleler Ausführung die in Praxis fast unmöglich/unrealistisch ist

### Scheduler Grundlagen

Keinen optimalen Scheduler, gute Allgemein Lösung, abhängig von Use-Zweck

Optimierungen:

Durchlaufzeit: Zeit von Start bis Ende eines Threats

Antwortzeit: Zeit von Anfang bis Antwort kommt

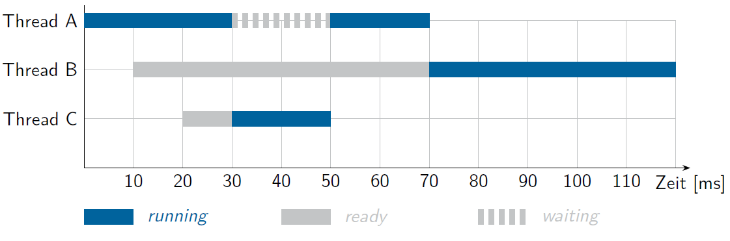
Wartezeit: Zeit, die ein Thread in der Ready Queue verbringt

Durchsatz: Anzahl Threadsm die in bestimmter Zeit verarbeitet werden können

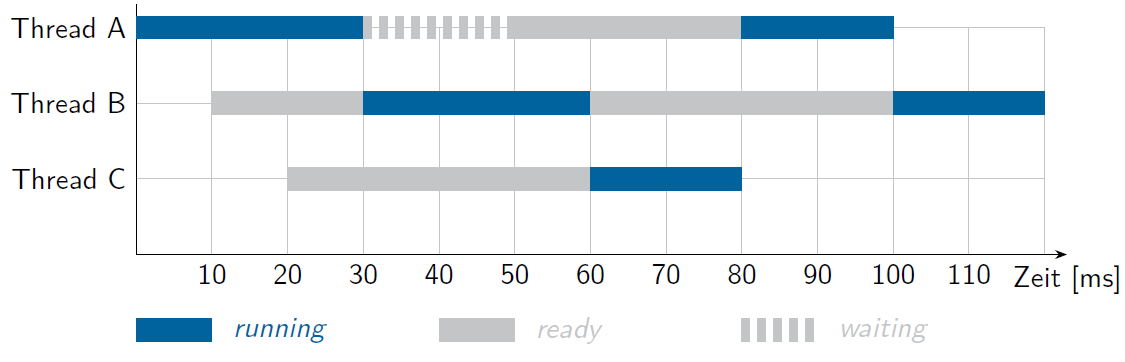
CPU Benutzung: Prozentsatz Verwendung(ggü. Nicht-Verwendung) CPU

### FCFS-Strategie(First Come First Served): 🡪 FIFO Prinzip, nicht präemtiv

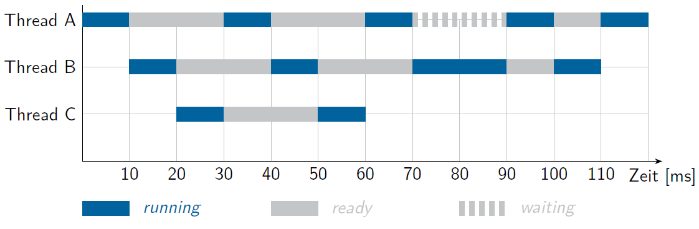
### SJF-Strategie (Shortest Job First): Thread mit kürzesten nächsten Prozessor-Burst, kooperativ oder präemtiv 🡪 optimale Wartezeit(kürzester blockt andere minimal)

Schwierigkeit ist nächste Prozessor-Burst Zeit vorauszusehen(Analyse hist. Daten)

***Round-Robin Scheduling***: FCFS mit Zeitscheibe 10 bis 100ms(in RQ nach x ms)

Braucht der Thread weniger als time-slice fängt anderer früher an!

Time Slice(oben 30ms | unten 10ms) beeinflusst Verhalten massiv:

Grösser: Latenz grösser, kleiner 🡪 mehr Kontextwechsel

### Prioritäten-basiertes Scheduling: Jeder Thread Priorität, wenn gleich 🡪 FCFS

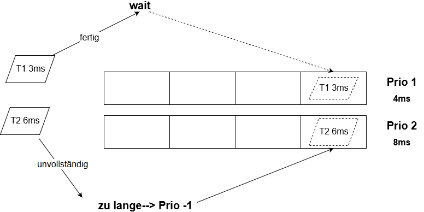
***Starvation(Verhungerung):*** Wenn ***Thread*** mit niederiger Prio unendlich nicht run

### Aging: Priorität in bestimmten Abständen erhöht(nützt gegen Starvation)

### Multi-Level-Scheduling: Thread kategorisiert und in Level aufgeteilt

### Jeder Level eigene Ready Queue + eigene Verfahren(L1 Prio, L2 Round Robin)

### Multi-Level Scheduling mit Feedback



Je Prio & Level eine Ready Queue + ein Time-Slice

Erschöpf Threads Zeitscheibe(prio-1)

Zeitscheiben gegen unten grösser

Kurze CPU-Burst bevorzugt

# Synchronisation

Jeder Thread eigenen Instruction-Pointer. Instruction-Pointer werden unabhängig voneinander bewegt🡪 führt zu Problemen. Implement. nur mit Hardware möglich!

## Producer-Consumer-Problem

Problem nebenläufiger Systeme – gegenseitiges Warten(Producer und Consumer),

weil Threads unterschiedlich schnell arbeiten-BUSY WAIT unnötige CPU-Belastung

Atomare Instruktionen: kann von CPU unterbrechungsfrei ausgeführt werden

**Race Condition**:

* Ergebnisse die von Ausführungsreihenfolge einzelner Instruktionen abhängen.
* Register werden bei Kontext-Wechsel gesichert, Hauptspeicher nicht.
* Threads müssen synchronisiert werden um Hauptspeicherveränderungen zu gewährleisten.(ansonsten Probleme und keine Garantie was passiert)
* Synchronisation🡪 Thread kann andere Threads Zugriff ausschliessen(im HS)

**Critical Section**: Code-Bereich, der Thread mit anderen Threads teilt

## Synchronisationsmechanismen

***Gegenseitiger* *Ausschluss***: Wenn ein Thread in seiner Critical Section ist, dürfen andere Threads nicht in ihre Critical Section.(Thread sperrt die Critical Section)

***Fortschritt***: Entscheiden wer als nächstes in Critical Section darf(in endlicher Zeit)

***Begrenztes* *Warten***: Thread wird nur n mal übergangen(von anderen Threads)

### Abschalten von Interrupts: Deaktivieren aller Interrupts wenn Critical Section betreten. Keine gute Lösung(bei parallel Threats). OS keine Threads unterbrechen

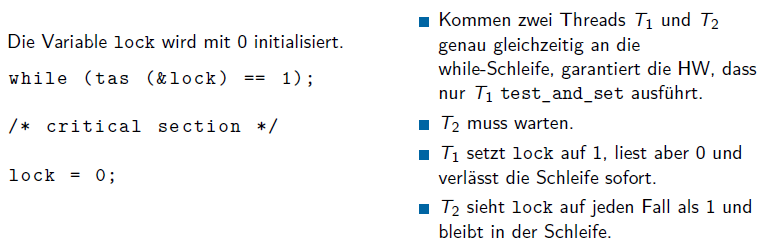
## Spezielle Instruktionen test-and-set, compare-and-swap

Zwei atomare Funktionen: ***test-and-set, compare-and-swap***

Hardware garantiert, dass nicht 2 Instruktionen gleichzeitig ausgeführt werden, auch über mehrere Prozessoren hinweg. Locks können implementiert werden.

Jedoch immer noch busy-wait!

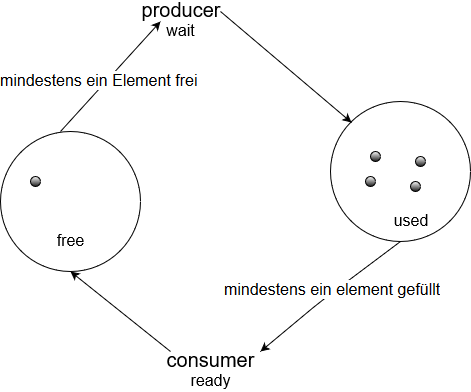
**Test-and-set:** Liest den Wert von einer Adresse (0 oder 1) und setzt ihn dann auf 1



**Compare-and-swap** Liest einen Wert aus dem Hauptspeicher und überschreibt ihn im Hauptspeicher, falls er einem erwarteten Wert entspricht

Beide Instruktionen lesen den Wert einer Adresse aus. TAS überschreibt diesen Wert aber in jedem Fall mit 1, während CAS das nur tut, wenn dieser Wert einem bestimmten anderen Wert entspricht.

## Semaphore

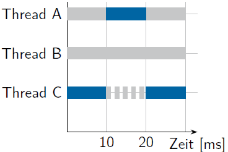
Mittels Zähler implementiert. Funktionen synchronisiert

* Post: z + 1
* Wait:
  + If z > 0 🡪{z – 1} setzt Ausführung fort
  + if z = 0 Thread in waiting bis anderer Thread z erhöht

Producer-Consumer-Problem**:** semaphore free = n und semaphore used = 0

### Priority Inversion: Vertauschen von Prioritäten , wegen gehaltener Ressource

Thread C braucht nach 10ms Semaphore s, welches Thread A mit niedrigster Prio hält. Das führt dazu, dass B, obwohl niedrigere Prio als C, die Ausführung nach 10ms beginnt, da C am Warten ist.



Lösung: Priority Inheritance: Priorität von Thread mit gehaltener Ressource wird erhöht und somit der Thread ausgeführt bis Semaphore freigegeben wird.

## Mutexe

Ein Mutexe hat einen binären Zustand z(0 oder 1). Es interessiert dabei nur, ob der Mutex in Verwendung oder nicht.(Semaphore Beschränkung z = 1)

* Acquire/Lock:
  + Wenn z = 0: Setze z auf 1 und fahre fort
  + Wenn z = 1: blockiere Thread solange bis z = 0
* Release/Unlock: setzt z = 0

# Interprozess-Kommunikation (IPC)

Prozesse müssen in verschiedenen Situation. miteinander kommunizieren können.

## Signale

Signale sind dafür da, einen Prozess von aussen zu unterbrechen🡪 OS verhält sich so als wenn ein Interrupt geschickt worden wäre.:

* Unterbrechen des gerade laufenden Prozesses (bzw. Threads)
* Auswahl der Signal-Handler-Funktion
* Ausführen der Signal-Handler-Funktion
* Fortsetzen des Prozesses (falls Signal-Handler Prozess nicht beendet)

Quellen: HW/OS (segm fault, ungültige. Instuktion /0) Andere Prozesse (Ctr+C, kill)

Jeder Prozess hat pro Signal einen Handler. Signal-Handler können überschrieben werden(ausser SIGKILL und SIGSTOP)

|  |  |
| --- | --- |
| SIGFPE | Fehler in arithmetischer Operation |
| SIGILL | Ungültige Instruktion |
| SIGSEGV | Ungültiger Speicherzugriff |
| SIGSYS | Ungültiger Systemaufruf |
| SIGTERM | Höfliche Anfrage Beenden |
| SIGINT | nachdrücklichere Aufforderung Beenden 🡪 Ctrl+C |
| SIGQUIT | =SIGINT aber abnormale Terminierung |
| SIGABRT | =SIGQUIT schickt sich Prozess selber(wenn Fehler bemerkt) |
| SIGKILL | Letzte Zuflucht: Kann nicht blockiert oder abgefangen werden |
| SIGTSTP | Zustand Stopped 🡪 Ctrl-Z |
| SIGSTOP | =SIGTSTP kann nicht ignorier oder abgefangen werden |
| SIGCONT | Prozess fortsetzen Kommando=fg oder bg |

kill 1234 5678 // SIGTERM an Prozesse 1234 5678

kill -KILL 1234 // SIGKILL an 1234

kill -l // lister alle Signale auf

## Message Passing

Triviale Implementierung des Producer-Consumer-Problems

Send Kopiert die Nachricht *aus* dem Prozess - send (message)

Receive Kopiert die Nachricht *in* den Prozess - receive (message)

### Direkte Kommunikation(Nur ein Empfänger pro Nachricht)

|  |  |
| --- | --- |
| Senden | P1 sendet Nachricht direkt an P2(Muss Empfänger kennen)  🡪 send(p1, message) |
| Empfangen | Symmetrisch: Empfänger muss Sender kennen  🡪 receive (P2, message) |
| Asymmetrisch**:** Sender nicht kennen, erhält ID in Out-Parameter  **🡪** receive(id, message**)** |

### Indirekte Kommunikation(Mehrere Empfänger pro Nachricht)

### Ein Empfänger

OS-Objekte Mailboxen, Ports oder Queues vorhanden

P1 sendet Nachricht an Queue/Maibox: send (Q, message)

P2 empfängt Nachricht aus Queue/Mailbox: receive (Q, message)

Beide Prozesse müssen Objekt kennen, gibt nur noch ein Sender/Emfänger --> Q

### Mehrere Empfänger

Möglichkeiten zu entscheiden, welcher Prozess die Nachricht empfängt:

Beschränke der Queue auf Sender und Empfänger, Beschränken des Aufrufs von receive auf nur einem Prozess, Zufällige Auswahl or Algorithmen(z.B round-robin)

### Lebensdauer Queue

Queue kann zu Prozess gehören(lebt solange wie der Prozess) oder kann dem OS gehören und lebt dann unabhängig vom Prozess.

### Synchronisation

|  |  |  |
| --- | --- | --- |
|  | blockieren (synchron) | nicht-blockieren (asynchron) |
| Senden | Sender blockiert bis die Nachricht vom Empfänger empfangen wurde. | Sender sendet Nachricht und macht weiter |
| Empfangen | Empfänger blockiert bis Nachricht verfügbar | Empfänger erhält Nachricht wenn verfügbar |

**Rendezvous**: Wenn Empfangen und Senden beide blockieren sind. OS kopiert dann direkt vom Sende in den Empfänger-Prozess

### Message Pufferung

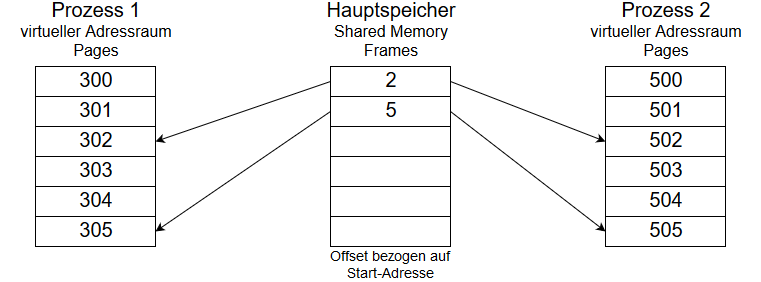
Keine: Keine Zwischenspeicherung 🡪 Sender muss blockieren

Beschränkte: n Nachrichten speichern 🡪 blockiert wenn voll   
Unbeschränkte: unbegrenzt Nachrichten speichern(theoretisch)-> nie blockieren

### Prioritäten: Nachrichten mit Prioritäten Empfänger zuerst MSG mit höchster Prio

## Shared Memory

Hauptspeicherbereich, der für mehrere Prozesse gleichzeitig sichtbar ist.

Pointer, welche auf Speicherzellen ausserhalb des Shared Memory zeigen, sind für andere Prozesse wertlos, da diese andere virtuelle Adressen(Pages) haben.

***Lösung***: Man arbeitet mit Pointer, welche relativ auf die Anfangsadresse zeigen:

Offset bezogen auf eine Start-Adresse

## Shared Memory vs. Message Passing

**Shared Memory**

|  |  |  |
| --- | --- | --- |
| * Schnell realisiert | * schnell existierende Applikation umschreiben weniger wartbar | * weniger stark modularisiert + Prozesse schlechter gegeinander geschützt |

**Message Passing**

|  |  |  |
| --- | --- | --- |
| * Mehr Engineering Aufwand | * Existierende App neu implementiert danach wartbar/erweiterbar | * sauber gekapselte Anwendung |

**Performance**: Bei modernen mehr-core Prozessoren eher Message Passing

# DATEISYSTEME-API

### Filesystem Zwei Bedeutungen: Teile des OS für die Verwaltung der Datenträger und Strukturierung eines Datenträgers.

Datenträgerstrukturierung interessiert Applikation nicht da API sehr abstrakt.

### Logische Datei: Anwendungssicht Einheiten von Bytes in Dateisystem, Inhalt oder interne Struktur egal für Dateisystem

Metadaten: Informationen zur logischen Datei gehören. Sichtbar für user(Grösse, Name, etc.) und z.T unsichtbar(Ablageort/Vekettung von Blöcken auf Datenträger)

Dateitypen: Alles nach dem letzten Punkt, keine Bedeutung für FS/OS nur für APP

Verzeichnis: Datei mit Liste von weiteren Dateien/Verzeichnisse, hat einen Namen

Verzeichnishierarchie: Gesamtheit aller Verzeichnisse im System(Baum Struktur)

Wurzelverzeichnis(root): Oberstes Verzeichnis in Hierarchie, keinen Namen!!!!

**.** Referenz auf sich selbst

**..** Referenz auf Elternverzeichnis

Pfad: Spezifiziert Verzeichnis oder Datei in Verzeichnishierarchie.

|  |  |
| --- | --- |
| Arbeitsverzeichnis | /home/dese |
| Absoluter Pfad | Beginnt mit Wurzelverzeichnis 🡪 /home/dese/docs |
| Relativer Pfad | Startet im Arbeitsverzeichnis, ohne / 🡪 docs/note.txt |
| Kanonischer Pfad | Tatsächlicher Pfad einer Datei /home/dese/docs/note.txt |

Wichtig: Für jede Datei(& Verzeichnis) gibt es beliebig viele relative und absolute Pfade ABER nur einen kanonischen(tatsächlichen) Pfad:

Realpath testdatei: /test/testdatei

Symlink: /home/dese/test/symlink-testdatei (symlink-testdatei -> /test/testdatei)

******realpath symlink-testdatei: /test/testdatei

***Hardlink***: gleicher Inode, verschiedene Pfade

***SymLink:*** Datei enthält Pfad anderer Datei

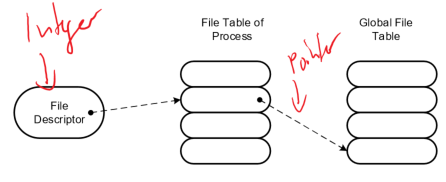
Wenn Pfad < 60 kein Datenblock gebraucht(weil 60 Bytes passen genau in einen Inode)

## Posix API

Posix: Direkten, unformatierten Zugriff auf Inhalt der Datei(rohe Binärdateien)

File-Descriptor: Ist ein Index(Integer) einer Tabelle aller im Prozess geöffneten Dateien. Die Tabelle beinhaltet Einträge(Pointer), welche auf die systemweite Tabelle aller geöffneten Datei zeigt.

***Ein Bild, das Screenshot enthält.

Mit sehr hoher Zuverlässigkeit generierte Beschreibung***Zustandbehaftet: Merkt sich aktuellen Offset(Offset des Bytes, welches als nächstes gelesen werden wird)

# Dateisysteme ext2 und ext4

### Partition: Ein Teil eines Datenträgers, wird wie ein Datenträger behandelt.

***Volume:*** Ein Datenträger oder eine Partition davon.

***Sektor:*** Kleinste logische Untereinheit von Volumens(Blocknummer \* Anzahl Sekt)

***Format***: Layout der logischen Strukturen auf dem Datenträger(vom DS festgelegt)

### Block: Mehrere aufeinanderfolgende Sektoren, normal 1,2 oder 4 KB

1. Sektor: Speziell: Blocknummern \* «Anzahl Sektoren je Block»

***Speicherallokation(read/write)***: In Form von Blöcken, ***Datentransfer*** als Sektoren

Logische Blocknummer: Blocknummer vom Anfang der Datei ausgesehen

***Physische Blocknummer***: Tatsächliche Blocknummer auf dem Volumen

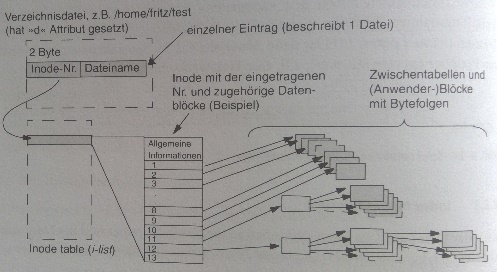
***Fragmentierung(v.a HDD)***: Wenn phy. Blocknummern grosse Lücken haben.

## Inode

dient zur Verwaltung einzelner Dateien

* Zu jeder Datei im Verzeichnis ein Dateieintrag(siehe unten für mehr Infos)
* Inode-Nummer ist Index in Liste der verfügbaren Verwaltungsblöcke(i-list).
* Jeder Inode beschreibt genau eine Datei.
* beinhaltet allgemeine Infos(Metadaten[Eigentümerinfos, Permission Bits, Grosse, Zeitinfos) wie auch sämtliche Angaben um alle Datenblöcke der Datei zu adressieren.
* Es gibt direkte Verweise auf Datenblöcke aber auch (1-, 2-, 3-fach) indirekte Verweise(Siehe Bild). Diese Multi-level Verweise werden wieder mit Listen in Datenblöcken realisiert.
* ***Blockliste***: 60 Byte Inode aufgeteilt in 4 Byte grosse Blocknummern.

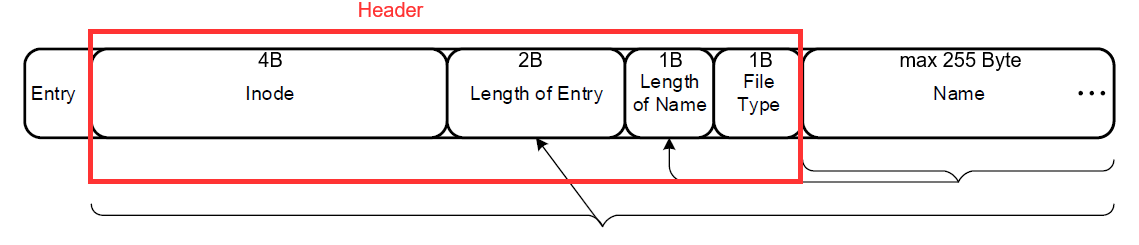
Die ersten 12 Blöcke verweisen auf die ersten 12 Datenblöcke der Datei.

******Die 13. Ist ein indirekter Block, 14 + 15 mehrfach indirekter Block:

### File Holes: Bereiche in der Datei, in der nur Nullen stehen(Referenz auf 0)

Spezielle Inodes: Wurzelverzeichnis Inode 2, schlechte Datensätze Inode 1, . referenziert auf eigenen Inode, .. Inode des Elternverzeichnisses

## Dateieintrag und Blockgruppen

Dateieintrag: Ein Dateieintrag in einem Verzeichnis ist so aufgebaut:

Blockgruppe: Volumen wird in Blockgruppen aufgeteilt: Ein Bild, das Screenshot enthält.

Mit sehr hoher Zuverlässigkeit generierte Beschreibung

Superblock: Enthält Metadaten(Anzahlen, Grössen, Zeitpunkte, etc.) über Volumen, beginnt an Byte 1024(davor eventuell boot-relevante Daten)

Gruppendeskriptortabelle: Ein Gruppendeskriptor je Gruppe

Gruppendiskriptor: Beschreibung(Infos) einer Blockgruppe

Sparse Superblocks: Reduktion der Superblocks -> reine Potenzen von 3, 5 oder 7;

Inode table: Grosse Tabelle von Inodes der jeweiligen Blockgruppe(BG = Inode -1)

FAT: Verzeichnisse haben alle Daten über Dateien, keine hard-links möglich

EXT2: Dateien durch Inodes beschrieben, Link von Verzeichnis zu Inode

NTFS: Dateien durch File-Records beschrieben, Verzeichnis hat Namen und Link auf Datei, Link und Name sind ein Datei-Attribut

## Ext4

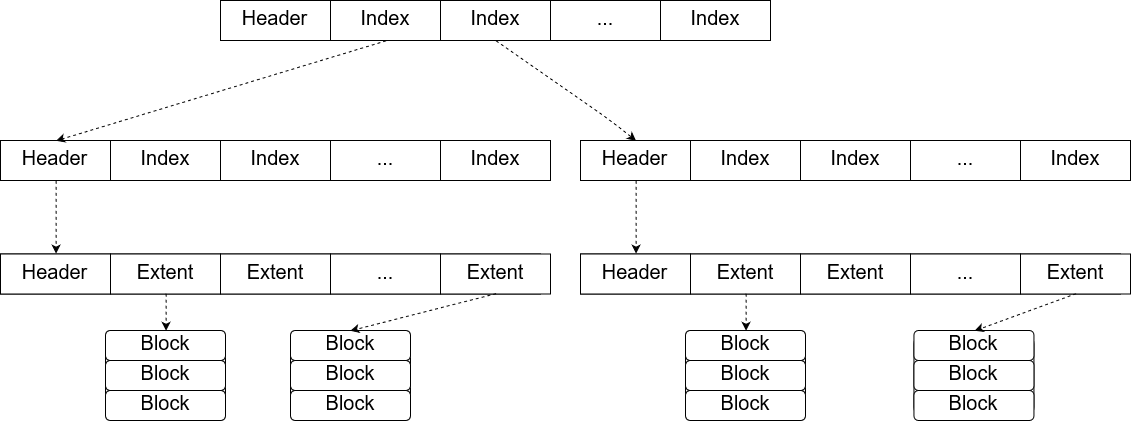
Wichtige Datenstrukturvergrösserungen (Inodes von 128 auf 256B, Grp-Diskreptoren von 32 auf 64B, Blockgrösse bis 64KB),

Extend Trees: Bestehen aus 12Byte grossen Datenstrukturen(insgesamt also 60 Byte) mit folgende Elementen:

Header(ID des Index-Blocks) – welches Level und wie viele weitere Elemente

Index-Knoten(innere Knoten des Baumes) – verweist auf Baumelemente

Extents -Verweisen auf die Blöcke

Sobald man 5 Extends verwendet, braucht man Indexe. Bei grossen Dateien sinnvoll(spart sich enorm viel Metadaten). Denn Blocklisten müssen für jeden Block dessen Nummer(4B) in Metadaten speichern. Extend Trees eventuell nur ein Extend.

### Journaling: Mechanismus um Inkonsistenz-Prüfung zu verringern + Performance

Im Journaling geht es darum, einen Speicherbereich auf der Festplatte zu reservieren, auf dem Daten schnell zwischengespeichert werden könne 🡪 das Journal.

Es ist viel schneller, Daten in einen Ringpuffer zu schreiben als an ihre tatsächliche Adresse. Es wird wie folgt vorgegangen:

1. Daten(Änderungen) werden zuerst als Transaktions ins Journal geschrieben

2. Daten werden geändert(committing)🡪 auf endgültige Position geschrieben

3. Nach Commit werden Daten aus dem Journal entfernt.

Wenn das System crasht, kann anhand der Journal-Daten zumindest entschieden

werden, ob das System noch konsistent ist.

Journal Replay: Startet das System neu, so kann es anhand der Journal-Einträge die Meta-Daten untersuchen welche eventuell korrupt sind. So können Änderungen vorgenommen werden-> Führt zu fast fehlerfreiem Arbeiten.

Modi:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Journal | Ordered | Writeback |
|  | Metdaten und Datei-Inhalt ins Journal | Nur Metadaten ins Journal – Dateiinhalte vor Commit geschrieben | Nur Metadaten  Commit und Schreiben in beliebiger Reihenfolge |
| + | Max. Datensicherheit | Daten nach Commit richtigen Inhalt | Sehr schnell, keine Sync von commit und write nötig |
| - | Grosse Geschwindigkeiteinbussen | Etwas geringere Geschwindigkeit | Daten könne Datenmüll enthalten |

# Programme

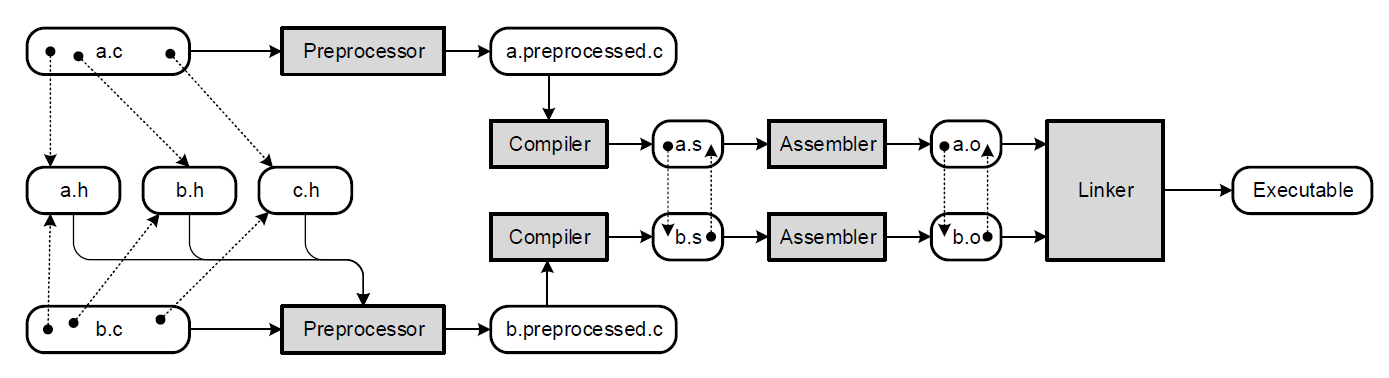
## C-Toolchain

**Präprozessor:**  Parst mehrmals: Entfernt Kommentare, Tokenization, Direktiven

Macht aus Mehreren Files(includes, etc.) ein pures C File.

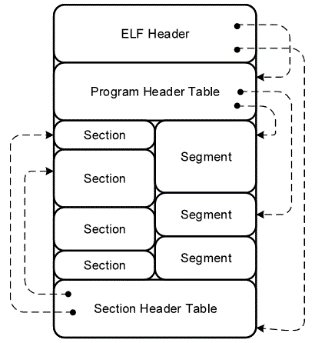
**Compiler**: Kompiliert reinen C-Code in Assembler Dateien.(+Optimierungen)

**Assembler:** Macht aus (mehreren) Assembler Dateien (mehrere) Objekt Dateien.

**Linker:** Macht aus mehren Objekt-Dateien eine Executable. Oder verknüpft Objektdateien (und statische Bibliotheken) zu dyn. Bibliotheken oder Executables

## ELF – Executable and Linking Format

Der ***loader*** lädt Executables und dynamische Bibliotheken in den Hauptspeicher.

***ELF*** ist ein Dateiformat, das sowohl für Linker als auch Loader Informationen halten kann.

***Struktur/Aufbau***:

ELF Header – beschreibt Aufbau der Datei

Execution View(Laden in HS) – loader sieht nur diese Dinge:

- Programm Header Table(beschr. Segmente)

- Segmente

Linking View(Verknüpfen von Object-Files):

- Section Header Table(beschr. Sektionen)

- Section

Vorgehen:

- Compiler erzeugt Sektionen

- Linker kombiniert gleiche Sektionen und definiert Segmente

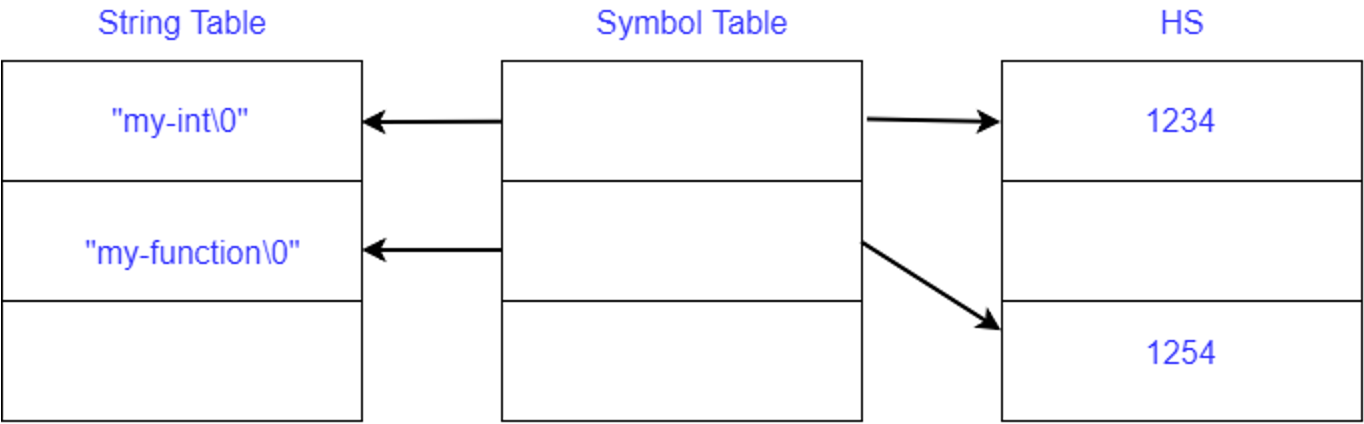
Segmente: Werden zur Laufzeit vom OS verwendet, Loader lädt bestimmte Segmente(in Executables) in den Hauptspeicher, wenn diese gebraucht werden.

Sektionen: Referenz auf String-Tabelle, Sektion werden vom Linker verwendet. Durchsucht alle Objekt-Dateien 🡪 Verschmilzt Sektionen mit gleichen Namen

* Es wird ein Executable erstellt

***String Tabelle:*** Bereich in Datei mit null-terminierten Meta-Daten String.

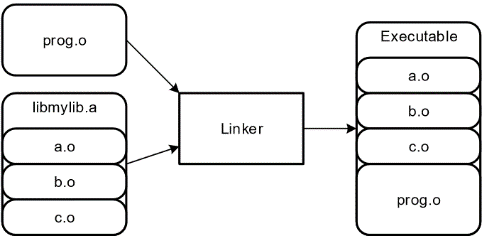
***Symboltabelle***: Beinhaltet Symbole mit Namen, Wert, Grösse und Info.

 Konzept wunderbar auf einander abgestimmt, ELF vielseitig eingesetzt!

# Bibliotheken lib<name>.a

## Statische Bibliotheken

Sammlungen/Archive von Objekt-Dateien. Archive sind Dateien, welche andere Dateien enthalten.

Der Linker behandelt statische Bibliotheken wie mehrere Objekt-Dateien.

+ Einfach zu verwenden und zu implementieren

- Programme müssen bei Änderungen in Bibliothek neu erstellt werden

- Funktionalität ist fix, keine Plugins möglich

## Dynamische Bibliotheken-(shared objects)

Werden erst zur Ladezeit oder Laufzeit des Programms gelinkt 🡪 Executable enthält nur noch die Referenz auf die Bibliothek.

+ Programm kann Lebenszyklus (teilweise) von Bibliothek entkoppeln

+ Programm wird flexibel 🡪 Updates/Features ohne Binaryänderungen

+ Funktionalität von verschiedenen Herstellen unabhängig updaten(Plug-Ins)

+ Bugfixes können direkt und ohne Unterbruch zum Anwender gebracht werden

+ Programm nur benötigte Bibliotheken laden 🡪 startet schneller, ist schmaler

- Höherer Aufwand für Programmieren, Compiler, Linker und OS

## Verwendung(Shared Objects)

***Bennenungsschema***

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

Tool ldconfig setzt Bibliotheksnamen korrekt auf! /usr/lib/ Standardordner

Alle Versionen und Unterversionen können gleichzeitig existieren und verwendet werden!

Real-Name: bei Erstellung verwendet, V + 1 bei Schnittst-Ände, UV +1 bei Bugfixes

Linker-Name: von Linker verwendet, Soft-Link auf SO-Name, neuste V. verwenden

SO-Name: von Loader verwendet, Soft-Link auf Real-Name, aktuellsten Bugfix ver.

## Verwendung von Bibliotheken

### Erstellen von Statischen Bibliotheken mit dem GCC & Tool ar

gcc -c f1.c -o f1.o | gcc -c f2.c -o f2.o 🡪 ar crs libmylib.a f1.o f2.o

### Erstellen von Dynamischen Bibliotheken mit dem GCC

gcc –fPIC –c f1.c –o f1.o | gcc –fPIC –c f2.c –o f2.o

gcc –shared –Wl,-soname,libmylib.so.2 –o libmylib.so.2.1 f1.o f2.o –lc

-lc 🡪 Einbinden der Standardbibliothek libc.so

### Verwendung von Bibliotheken

Statische Bibliothek: gcc main.c –o main –L. –lmylib

Dynamische Bibliothek mir Programm geladen: gcc main.c –o main –lmylib

Dynamische Bibliothek mit dlopen geladen: gcc main.c –o main –ldl

### Wichtige Tools / Shared Objects

readelf –d Inhalt der dynamischen Sektion, ldd alle indirekt benötigte Shared Objetct. Wichtigste: libc.so(Standard C) und ld-linux.so(ELF Shared Object loader)

## Implementierung Dynamische Bibliotheken – mit shared memory

* Müssen verschiebbar sein, mehrere Libaries in gleichen Prozess(nur ein virt. Adressraum) 🡪 Aufgabe des Linkers zum Loader(dynam. Linker) verschoben
* Sollen Code zwischen Programmen teilen. 🡪 Shared Memory Code in einem Frame(HS) aber in mehren Pages(virtueller Arbeitspeicher pro Prozess)

### Position-independent code Hängt nicht von seiner Adresse ab → Adressen relativ zum Instruction-Pointer. Dadurch sieht Code immer gleich aus(relative Adressen)

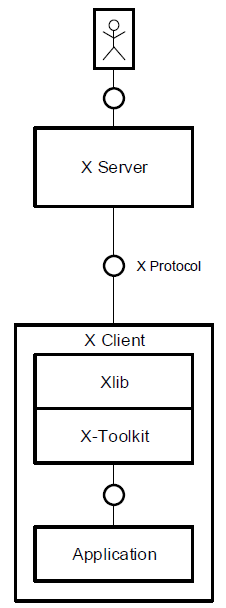
Globale Offset Table(GOT): 1x pro dyn. Biblio und exe. Enthält pro Symbol(dass Biblio/Exe verwendet) einen Eintrag., Beinhaltet relative Adressen 🡪 position Independent 🡪 Loader füllt zur Laufzeit die Adressen in GOT ein.

Procedure Linkage Table (PLT): Lazy-Binding für Funktionen; pro Funktion ein Eintrag Proxy-Funktion: Ersetzt sich selbst in GOT mit Link zur richtigen Funktion → erspart bedingten Sprung

Relative Move via Relative Calls**:** Funktion f will relative Move ausführen Hilfsfunktion h aufrufen. h kopiert Rücksprungadresse vom Stack in Register rücksprung. f hat Rücksprungadresse = Instruction Pointer

# X Windows System

## Grundlagen und Vorteile

Das GUI mittels X Windows System setzt auf modernem Unix-Systemkern auf und realisiert die GUI-Basisinfrastruktur:

+ Auf Unix-Kern aufgesetzt und damit austauschbar

+ Installierbar wenn tatsächlich benötigt

+ Netzwerktransparenz, Clients weiss nicht, wo Client läuft

+ Plattformunabhängig z.B auch Windows, Solaris, etc.

+ X legt Gestaltung der Bedienoberfläche nicht fest

## Komponenten

### Fenster: Rechteckiger Bereich des Bildschirms des beliebig weitere (Unter-)Fenster beinhalten kann. Dabei ist der Bildschirm die Wurzel(hat kein Elternfenster) mit 0 – n Unterfenster.

Maus: Physisches Gerät, das 2D-Bewegungen in Daten übersetzt(delta x und delta y).

Mauszeiger(Cursor): Grafik auf Bildschirm mit genauer Position(Hotspot/Pixel)

Funktionsweise: Das OS bewegt den Cursor analog der physischen Bewegung der Maus. Klickt man auf eine Maustaste, so soll ein Ereignis in dem unter dem Cursor liegenden Fenster ausgelöst werden. Z.B Klick auf Kreuz oben rechts 🡪 Schliesse das Fenster

## Programmiermodelle

### Programmgesteuert: Programmierer definiert Ablauf, Benutzer reagiert auf Prg.

Was wollen Sie kaufen?<Input> Wie viel wollen Sie kaufen?<Input> Noch mehr?...

Ereignisgesteuert: Benutzer wann welches Ereignis, Programm reagiert auf user.

Input-Felder mit Geburtstag eingeben und dann absenden. Programm wertet aus.

## Ein Bild, das Screenshot enthält. Mit sehr hoher Zuverlässigkeit generierte BeschreibungVerknüpfung von Maus und Fenster

- Maustreiber erzeugt Nachricht 🡪 Linke Maustasten-Klick an Position x,y

- OS verteilt Nachricht 🡪 Fenster x,y ermitteln 🡪 Nachricht an Fenster Besitzer

- Applikation verarbeitet Nachricht und führt das definierte Event aus(Reaktion)

- Asynchronität: Komponenten warten nicht aufeinander 🡪 sonst Cursorfreeze

- Gewisse Instruktionen werden direkt vom OS übernommen: Bewegen des Cursor

## GUI Architektur

Das GUI braucht folgende Komponenten:

X Window System: Grundfunktionen der Fensterdarstellung

Window Manager: Client-Applikation, welche die sichtbaren Fenster verwaltet und platziert. Es «dekoriert» Top-level-Windows von Applikationen mit Umrandungen, Knöpfe, Icons oder Aktionen: verschieben max-, minimieren, schliessen

Fensterhierarchie: Root-Window: bei Start von X Windows 🡪 ganzer Bildschirm

Dann Kinder und Kindes-Kinder 🡪 Werden vom Windows Manager verwaltet

Desktop Manager(Desktop): Desktop-Hilfsmittel, wie Taskleiste, Dateimanager, Papierkorb, etc.

Wichtig: X ist unabhängig von Window Manager oder Desktop. Viele verschiedene Implementierungen von Windows Manager und Desktop existieren.

Xlib: C Interface für X-Protokoll 🡪 gcc -lX11

X-Toolkits: Software-Schicht über Xlib 🡪 komfortabler 🡪 Bietet Widgets an

## Ein Bild, das Screenshot enthält. Mit sehr hoher Zuverlässigkeit generierte BeschreibungÜberblick

Display: Rechner mit Komponenten(Keyboard, Zeigeräte, Bildschirme, etc.)

Display in C = Objekt mit Informationen über Verbindung(Connection)

Client: Applikation, die ein Display nutzen will.

Server: Softwareteil des X-Window-System, der ein Display ansteuert

X Protocol: Protokoll für Nachrichten zwischen X Client und X Server:

- Requests: Dienstanforderungen | Client 🡪 Server(«zeiche Linie»)

- Replies: Antworten auf bestimmte Request | Client 🡨 Server

- Events: spontane Ereignismeldungen | Client 🡨 Server(«Mausklick»)

- Errors: Fehlermeldungen auf vorangegangene Requests | Client 🡨 Server

Nachrichtenpufferung: Request werden auf der Client-Seite gepuffert. 🡪 Übertragung an Server nur, wenn sinnvoll oder nötig 🡪 Serverentlastung

Ereignisse werden doppelt gepuffert(Server und Client). Serverseitig beachtet Netzwerkverfügbarkeit, Clientseitig, Events gequeuet bis Client abgeholt.

X Ressourcen: Server-seitige Daten zur Reduktion des Netzwerkverkehrs

Halten Infos im Auftrag von Client: Fenstereigenschaften, Colormap, Font, etc.

### Grafikfunktionen und Graphics Context: Grafikfunktionen z.B DrawRectangle braucht immer ein Graphic Context GC(Grafiktelementeigenschaften).

## Window Manager Events(Fenster schliessen mit Kreuz oben rechts)

Protokoll zwischen Window Manager und Applikation. Windows Manager senden eine ClientMessage Event an die Applikation. Event muss ein seinem data Teil die ID eines Properties WM\_DELETE\_MESSAGE enthalten.

Atom: ID eines String, der für Meta-Zwecke benötigt wird.

Properties: Gehören zu einem Fenster, vom WM gelesen, über Atom ermittelt

***WM\_PROTOCOLS***: Ist eine Liste von Protokollen. Ein Client kann/muss sich für Protokolle registrieren(Atom in Propery WM\_Protocols hinzufügen). Dazu muss er die Funktion XSetWMProtocols aufrufen.(Siehe unten für weitere Infos).

## Farben

Jeder Pixel besteht aus drei Subpixel: Rot Grün, Blau. Jeder Farbteil braucht 8 Bit 🡪 Farbe = 24Bit 🡪 224 Farben = 16 M Farben(Mensch nur 10M wahrnehmen)

### Bilddarstellung: Mittels Rastergrafik und Farbtabelle Schwarz/Weiss = 1 Bit pro Bildpunkt Farben/Grautöne = Mehrere Bits pro Bildpunkt

***Vorteil* *Farbtabelle*:** Anstatt 2n nur noch 2m Farben gleichzeitig.

***Colormap***: Ist eine x Ressource. Wird über ID ColorMap referenziert

***RGB***: rgb:rrrr/gggg/bbbb pro Farbe 4 Hex Stellen ***rgb***:1abc/0/e4

***RGBi (i=intensity):*** Dezimale Kommazahl rgbi:.5/0/.25

***Fensterattribute***: XSetWindowsAttributes (Hintergrundfarbe, Randfarbe, Gravity, Mauscursor) Ist eine Struktur keine Funktion!

***Fenster neuzeichnen***: Buffering umgehen mit Expose-Nachrichten oder beliebig

# Unicode

KeyCode: Jede Taste auf Tastatur hat einen Tastencode 🡪 KeyCode wird vom Server mitgesendet(event.keycode)

KeySym: Clients benötigen den mit dem Tastencode assoziierten Symbolcode. Diese haben jeweils Index 🡪 z.B «a» und «A» anderer Index.

***ASCII***: American Standard Code for Information Interchange 128 definierte Zeichen – 7Bit: 00 bis 7F

**UTF-32**: 32 Bit, jeder CP kann mit einer CU dargestellt werden

**UTF-32LE**: U0 = B0B1B2B3, **UTF-32BE**: U0 = B3B2B1B0

Litle-Endian-Format: niederwertigstes Bit zuerst

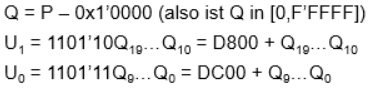
Big-Endian-Format: höchstwertigste Bit zuerst

**LE : 0xCAFEBABE = BE | BA | FE | CA**

**BE : 0xCAFEBABE = CA| FE | BA| BE**

**UTF-16:**

1 CU:

2 CU: 

P = 10330

Q = 10330 – 10000 = 0330 = 00 0000 0000 11 0011 0000

Q19…Q10 = 0 🡪 Q9…Q0 = 330

U1 = D800 + 0 = D800

U0 = DC00 + 330 = DF30

BE: D8 00 DF 30 🡪 LE: 00 D8 30 DF

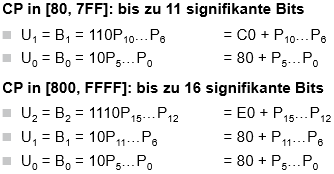
**UTF-16BE**: CP mit 1 CU: U0 = B1B0; 2 CUs: U1U0 = B3B2B1B0

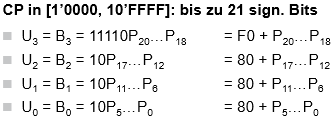
**UTF-16LE**: CP mit 1 CU: U0 = B0B1 ; 2 CUs: U1U0 = B2B3B0B1

**UTF-8**: 8 Bit, ein CP benötigt 1 bis 4 CUs, HTML, XML

Ohne D800 – DFFF, da UTF-16 diese nicht darstellen kann.

CP bis 7 signifi. Bits: U0 = B0 = 0P6…P0





$ = 20 AC = 0010 0000 1010 1100 🡪 14 signifikate Bits

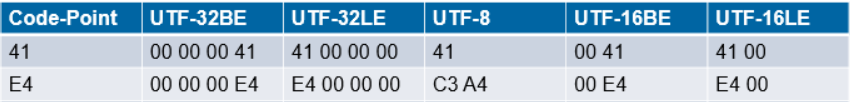
P15..P12 = 02 🡪 P11..P6 = 02 🡪 P5..P0 = 2C

U2 = B2 = **1110**P15..P12 = **E0** + P15..P12 = E2

U1 = B1 = **10**P11..P6 = **80** + P11..P6 = 82

U0 = B0 = **10**P5..P0 = **80** + P5..P0 = AC

* $ = E2 82 AC





|  |  |  |
| --- | --- | --- |
| **Hex ⬄ Binär** | **Hex ⬄ Dezimal** | **Dez/Hex** |
| 4er Blöcke  Hex: A 0 2  Bin: 1010 0000 0010 | AF -> Dez  15 \* 16^0 + 10 \* 16^1 = 15 + 160 =240 | Dez - Bin  Bin-Hex |

# Meltdown

## Vorbereitung

Aus Perfomance-Gründen(keine Kontext-Wechsel nötig) und unter der Annahme das die Page-Table so konfiguriert ist, dass nur das OS Zugriff darauf hat, mappt der OS-Kernel den gesamten physischen Hauptspeicher in jeden virtuellen Adressraum.

Meltdown ermöglicht es den gesamten Hauptspeicher auszulesen. Es wird möglich dass ein Prozess alle geheime Informationen der anderen Prozesse lesen kann.

Der Speicherschutz schmilzt dahin.

## Out-of-Order Execution & Spekulation für mehr Geschwindigkeit

Irgendwann vor einigen Jahren sind die Prozessoren so schnell geworden, dass dieser immer wieder lange Wartezeiten hatte, da die anderen Speicher relativ langsam waren. Man hat nach Mechanismen gesucht, diese Wartezeit zu überbrücken:  
Out-of-Order Execution: Ein CPU darf die Reihenfolge der auszuführenden Instruktionen ändern, wenn keine Kontextverletzung stattfindet. Ebenfalls agieren moderne CPUs heute spekulativ. Er führt gewisse Instruktionen spekulativ aus, um dadurch Zeit zu gewinnen. Auch wenn diese eigentlich gar nicht ausgeführt werden hätten dürfen.

## Meltdown Vorgehen

**Seiteneffekte von O3E:**

Nun kann es sein, dass gewisse Daten spekulativ in den Cache geladen worden sind. Man kann den Cache zwar nicht auslesen, aber anhand der Zugriffszeit entscheiden, ob gewisse Daten/Zeilen sich im Cache befinden oder nicht.

Genau bei diesem Massnahme kommt Meltdown ins Spiel.

## Daten auslesen anhand der Zugriffszeit

Man möchte nun Daten auslesen auf die man eigentlich gar kein Zugriff haben dürfte. Man muss nur erreichen, dass diese im Cache landen und können nachher anhand der Zugriffszeit rausfinden, welche Daten es sind:

Man legt ein grosses Array an und definiert den zu erratenden Wert X als Index im Array.

Bringe die CPU dazu den Wert Array[X] in den Cache zu laden(via O3E)

Dann iteriert man über das Array und misst die Zugriffszeiten

Der Index mit dem kürzesten Zugriff entspricht dem gesuchten Wert X

## Eindeutigkeit 🡪 Flash & Reload

Damit man sicher sein kann, dass es nur ein X mit kürzestem Zugriff gibt will man davor aber den Cache noch «bereinigen»:

Clflush p 🡪 Entfernt alle Zeilen mit Adresse p.

Man iteriert über das Array und ruft für jedes Element clflush auf. Somit stellt man sicher, dass kein Element des Arrays sich im Cache befindet.

Problem Cachezeilen

Nun hat man das Problem, dass der Cache immer ganze Cachezeilen ladet und zum Teil sogar noch benachbarte Cachezeilen(Lokalitätsprinzip 🡪 Spekulation für Performance). Sicher ist, dass der Cache nicht mehr als eine Page laden kann. Somit macht man jedes Element im Array so gross wie eine Page.

Hat man den Cache bereinigt und den die Elementgrösse im Array auf eine Pagegrösse abgestimmt, kann man Meltdown verwenden. Wie vorhin beschrieben.

## Allgemein + was hilft dagegen

* Hauptsächlich Intel CPUs betroffen.
* 500KB/s 0.02% Fehlerrate 🡪 Schnell und wenig Fehler
* Hinerlässt praktisch keine Spuren
* Passwörter auslesen z.B von Browser aus Passwort Manager
* Von VM auf andere VM zugreifen(Cloud!!!)
* **Lösung**: Kernel page-table isolation 🡪 Verschieden Page Tables für Kernel bzw. User mode

# Code

## Environment

char \* getenv (const char \* key)//Durchsucht Umgebungsvariabel nach key;   
return Pointer auf erstes Zeichen oder 0, wenn nicht gefunden

int unsetenv(const char \*key***)***//Entfernt Umgebungsvariabele mit dem key

int setenv(const char \*key, const char \*value, int overwrite) ***KEY*** ***gefunden*** + Overwrite != 0: überschreibt value (löscht alten Eintrag)

***KEY*** ***nicht*** ***gefunden***: neue Umgebungsvariable + Kopiert key & value

Return 0🡪 all OK ansonsten Fehlercode in errno

int putenv (char \* kvp) //Fügt Pointer kvp dem Array environment hinzu

***key noch nicht vorhanden***: Pointer environment[x] setzen

***key bereits vorhanden***: String löschen & Pointer environment[x] umgehängt

0 wenn alles OK sonst errno

|  |  |
| --- | --- |
| Externe + Globale Deklarationenextern char \*\*environ;char sevenv[] ="severin=king";***Print-Anwendung***Print\_all();Print\_value(key);***print-env-Anwendung***print\_env();} erheblicher Unterschied zwischen char \*a und char a[]!  Ersteres Pointer auf Strin-Literal(Read-Only-Speicher). Zweiteres Array, dass mit String-Literalen initialisiert wird! | print\_value-Methode***void print\_value(const char \* key){***char \*value = getenv(key);if(value == 0){printf("key not found\n", key);}else{printf("key=value: ’%s’='%s'\n", key, value);}}Print-all()-Methode***void print\_all(){***int i = 0;while(environ[i] != NULL){printf("%s\n", environ[i++]);}} |
| setenv-Anwendung – Kopieren!!char\* key = "SHELL";int overwrite = 1;char\* value = "/bin/sh";int result = ***setenv*** (key, value, overwrite)// SHELL=/bin/sh | putenv-Anwendung – Pointer!!char sevenv[] ="severin=king";***putenv***(sevenv); //Pointer setzenfor(int i = 8; i < sizeof(sevenv) -1; i++){sevenv[i] = '?';}//severin=???? |

## Prozesse

pid\_t fork(void)// dupliziert den Prozess🡪 erstellt Childprozess gibt   
 Rückgabe🡪In P bei Erfolgt Child-Prozess-ID(>0) oder Fehler(<0) bei C return = 0

pid\_t wait(int \*status) // Unterbricht aufrufender Prozess bis irgendein Child-Prozess beendet wird

pid\_t waitpid (pid\_t pid, int \*status, int options) //wie wait aber gibt an, auf welchen Child Prozess gewartet werden soll(pid > 0 🡪 Childnummer)

unsigned int sleep(unsigned int seconds) //Unterbricht die Ausführung für die Anzahl Sekunden(ungefähr), gibt Anzahl verbleibenden Sekunden vom Schlaf zurück

void exit(int code) // Entspricht dem gleichnamigen OS-Auruf, Rücksprung main

int atexit(void (\*function)(void)) – aufräumen nach dem exit vom main

void exit(int code) // Entspricht dem gleichnamigen OS-Auruf, Rücksprung main

pid\_t getpid(void); //Gibt die Prozess-ID zurück

pid\_t getppid(void); // //Gibt die Parent Prozess-ID zurück

|  |  |
| --- | --- |
| ***Print Information Methode***void print\_pinfos(){printf("Process-ID: %i\nParent-ID: %i\n", getpid(), getppid());}***Print Parent Methode***void print\_parent(){printf("I'm the parent: \n");print\_pinfos();}***Print Child Methode***void print\_child(){printf("I'm the child, parent wait until I exit: \n");print\_pinfos();sleep(10);printf("Exit\n");} | ***Main Test Method***pid\_t pid = fork();if(pid > 0){pid\_t waitreturn = wait(0);print\_parent();}else if(pid == 0){print\_child();}else{printf("Error");} ***atexit (&handle\_at\_exit);*** |

## Adressen

Computergenerierter Alternativtext:
global var: 6295592 
local in main: 2109309116 
Add ress 
Add ress 
Add ress 
Add ress 
variable 
function 
variable 
function 
fl 
fl 
f2 
f2 
2109309084 
4195543 
2109309084 
4195596 Computergenerierter Alternativtext:
global var: 6295592 
local in main: 1427575644 
Add ress 
Add ress 
Add ress 
Add ress 
variable 
function 
variable 
function 
fl 
fl 
f2 
f2 
1427575612 
4195543 
1427575612 
4195596 

lokale Variabeln auf dem Stack deshalb gleich(Var push dann pop dann var2 push)

Funktionen im virtuellen Adressraum(Heap) deshalb kleinere Nummern

🡪 Deaktivieren von ***Address Space Layout Randomization*** führt dazu, dass Adressen nicht mehr «unvorhersehbar» sind sondern immer gleich bleiben

## Threads

### Int pthread\_create (pthread\_t \* thread\_id , pthread\_attr\_t const \* attributes , void \* (\* start\_function ) ( void \*), void \* argument )

Erzeugt einen Thread und gibt bei Erfolg 0 zurück, sonst einen Fehlercode

Die ID des neuen Threads wird im Out-Parameter thread\_id zurückgegeben

### void pthread\_exit ( void \* exit\_code )

Beendet den Thread und gibt den exit\_code zurück = Rücksprung start\_function

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

Sendet Anforderung, dass Thread mit thread\_id beendet werden soll. Wartet nicht auf tatsächliche Beendigung. Rückgabewert = 0, wenn Thread existiert

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

Entfernt Speicher, den Thread belegt hat, falls dieser beendet(keine Beendigung)

### int pthread\_join ( pthread\_t thread\_id )

Wartet solange, bis Thread(thread\_id) beendet wurde. Ruft pthread\_detach auf.

|  |  |
| --- | --- |
| int globvar;void print\_infos(){int locvar;printf("Address locvar: %p\n", &locvar);printf("Address globvar: %p\n", &globvar);printf("Thread-ID: %lu\n", pthread\_self());printf("Prozess-ID: %i\n", getpid());}void \* execute\_thread(void \* data){print\_infos(); //print thread infospthread\_exit(NULL); //thread beenden}void main(){***pthread\_t tid1;***pthread\_t tid2;***pthread\_create(&tid1,0, &execute\_thread, NULL);***pthread\_create(&tid2,0, &execute\_thread, NULL);pthread\_join(tid1, NULL);pthread\_join(tid2, NULL);} | Session Log:  Address locvar: 0x7fadb5393ecc  Address globvar: 0x601050  Thread-ID: 140384046499584  Prozess-ID: 6238  Address locvar: 0x7fadb5b94ecc  Address globvar: 0x601050  Thread-ID: 140384054892288  Prozess-ID: 6238  Threads haben andere lokale Variable 🡪 Jeder eignener Stack  Gleiche globale Variabeln:  Liegen auf Heap |

## Amdahls Regel

Serielle Algorithmen können parallelisiert werden(zumindest Teil davon)

Einige Teile nicht parallelisierbar wegen Abhängigkeiten, etc.

T: Ausführungszeit wenn seriell

n: Anzahl Prozessoren

T’: Ausführungszeit, wenn max. parallelisiert

Ts: Ausführungszeit für serieller Anteil

T – Ts: Ausführungszeit für paralleler Anteil

s = : serieller Anteil im Algorithmus,

Speed-up-Faktor: , unabhängig von der Zeit:

Grenzwert:

## Thread Local Storage

int pthread\_key\_create ( pthread\_key\_t \*key , void (\* destructor )  
Erzeugt einen neuen Key im Out-Parameter key  
Gib 0 zurück, wenn alles OK war, sonst einen Fehlercode.  
Das OS ruft den destructor am Ende des Threads mit dem jeweiligen  
thread-spezifischen Wert auf, wenn dieser dann nicht 0 ist.

int pthread\_key\_delete ( pthread\_key\_t key)  
Entfernt den Key und die entsprechenden Values aus allen Threads.  
Der Key darf nach diesem Aufruf nicht mehr verwendet werden.  
Das Programm muss dafür sorgen, sämtlichen Speicher freizugeben  
Gib 0 zurück, wenn alles OK war, sonst einen Fehlercode.

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

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

Schreibt bzw. liest den Wert, der mit dem Key in diesem Thread assoziiert ist.

### long pthread\_self () //Gibt die ID des Threads als Long zurück

# Synchronisation

#### Semaphore

***Int sem\_init(sem\_t \*sem, int pshared, unsigned int value)***   
initialisiert semaphore sem, enthält value Zahl, pshared = 0 sem nur innerhalb gleicher Prozess

Int sem\_wait(sem\_t \*sem) – implementiert wait   
If z > 0 🡪{z – 1} setzt Ausführung fort  
if z = 0 Thread in waiting bis anderer Thread z erhöht

Int sem\_post(sem\_t \*sem) – implementiert post z + 1

Int sem\_destroy(sem\_t \*sem) – entfernt möglichen zus. Speicher

#### Mutexe

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

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

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

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

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

|  |  |
| --- | --- |
| #define JOBS (100 \* 100)#define PERSONS 100#define MAX 10struct printer{unsigned const max\_jobs; ***sem\_t jobs\_in\_queue*** ;};void printing\_document(struct printer \* my\_printer){printf("Printing \n"); ***sem\_wait(&my\_printer->jobs\_in\_queue);}***void finish\_printing(struct printer \* my\_printer){printf("Finishing \n");***sem\_post(&my\_printer->jobs\_in\_queue);***}void print\_status(struct printer \* my\_printer){int job\_amount;sem\_getvalue (&my\_printer->jobs\_in\_queue, &job\_amount);printf("There are %i/%i print jobs active!\n", my\_printer->max\_jobs - job\_amount, my\_printer->max\_jobs);} | void add\_printjobs( struct printer \* my\_printer , int amount ){for(int i = 0; i < amount; i++){printing\_document(my\_printer);print\_status(my\_printer);finish\_printing(my\_printer);}}void \* execute\_print\_job(void \* data){struct printer \*my\_printer = data;add\_printjobs(my\_printer, JOBS);return 0;}void main(){struct printer my\_printer = {MAX, 0};sem\_init (&my\_printer.jobs\_in\_queue, 0, MAX);print\_status(&my\_printer);pthread\_t tid[PERSONS];for(int i = 0; i < PERSONS; i++){pthread\_create (&tid[i], NULL, &execute\_print\_job, &my\_printer);}for(int j = 0; j < PERSONS; j++) {pthread\_join(tid[j], NULL); }***sem\_destroy (&my\_printer.jobs\_in\_queue);***} |

# Interprozess-Kommunikation (IPC)

## Signale

struct sigaction { void (\*sa\_handler)(int); sigset\_t sa\_mask; int sa\_flags; };  
Argumente: Handler Funktion, zu blockierende Signale, zusätzliche Eigenschaften

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

Überschreibt die Aktion vom Signal signal

Ein Bild, das Text enthält.

Mit hoher Zuverlässigkeit generierte Beschreibungsignal=Nummer des Signals, gibt bestehenden Signal Handler zurück

### SIGINT(CTRL + C) & SIGTSZP(CTRL + Z)überschreiben!

|  |  |
| --- | --- |
| void print\_after\_ctrl\_c(int signal){if(signal == 2){ printf("SIGINT Signal-ID: %i\n", signal);} //ID=2else{printf("SIGTSTP SIGNAL-ID %i\n", signal); //ID=20}} | void main (int argc, char \*\*argv){struct sigaction sa = {.sa\_handler = &print\_after\_ctrl\_c,.sa\_flags = SA\_RESETHAND };sigemptyset(&sa.sa\_mask);sigaction (SIGINT, &sa, 0);sigaction (SIGTSTP, &sa, 0);sleep(10);} |

## Message Passing

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

***Flags:*** O\_RDONLY, O\_RDWR, O\_CREAT(Creat if not exist), O\_NONBLOCK

***Mode:*** S\_IRUSR | S\_IWUSR

***Attr = 0***🡪 Default-Attribute

***int mq\_close (mqd\_t queue);*** //Schliesst Queue bleibt aber noch im System

***int mq\_unlink (const char \*name);*** //Entfernt Queue aus dem System

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

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

#### mqd\_t q1 = mq\_open (QUEUE1, O\_WRONLY | O\_CREAT, S\_IRUSR | S\_IWUSR, 0);

# File System

## Posix API

***open(path, flags) -> fd*** //Erzeugt einen Filediscriptor, return fd oder -1

***Flags:*** O\_RDONLY, O\_RDWR, O\_CREAT(Creat if not exist), O\_NONBLOCK

O\_APPEND: Setze Offset ans Ende der Datei vor jedem Schreibzugriff

O\_TRUNC: Setze Länge der Datei auf 0

***close (fd) -> status*** //Entfernt(Dealloziert) den File-Discriptor fd, return 0/-1

***read (fd, buffer, n) -> status*** //Lesen von nächsten n Bytes; return n oder -1(errno!!)

***write (fd, buffer, n) -> status*** //Kopiert n byte von Buffer an aktuelle Offset, analog read

***lseek (fd, offset, origin) -> status*** //Offset von fd auf offset setzten, retur. new-offset/-1

***origin:*** gib an wozu offset relativ: SEEK\_SET(Beginn der Datei),

SEEK\_CUR(Aktueller Offset) SEEK\_END(Ende der Datei)

***Pread/pwrite->*** //wie read/write aber mit Offset als Argument(verändert Offset nicht)

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

### Konstanten für Zugriffsrechte(mit | verknüpfen):

User rwx: S\_IRWXU

Group r--: S\_IRGRP 🡪 read

## C API

C-API: Zugriff auf Streams(Strom von Bytes)

File: Datenstruktur, welche Informationen über den Stream beinhaltet, soll über C-API erzeugte Pointer verwendet werden.(Pointer (FILE \*))

***fdopen(fd, char \*mode) -> FILE*** // Erzeugt File aus fd, return FILE oder null(errno!)

***fopen(path, mode) -> FILE*** // Erzeugt FILE aus path, return FILE oder null(errno!)

***fileno(FILE \*stream) -> fd*** //Gibt fd zurück auf den sich der Stream bezieht oder -1

***fflush (stream) -> status*** //Leer Buffer in Datei, return 0 oder EOF

***fclose(stream) -> status*** //Schliesst den Stream auf eine Datei, run fflush, return 0/EOF

***feof (stream) -> int*** //return 0 wenn End-Of\_File nicht erreicht wurde

***ferror(stream) -> status*** //return 0 wenn kein Fehler aufgetreten ist

***fgetc(stream) -> int*** //liest nächste byte und konvertiert es in einen int!

***fgets(string, n, stream) -> int*** //liest solange Bytes bin Ende oder n-1 Byzes gelesen, hängt eine 0 an letzte gelesende Byte an🡪 erzeugt 0 terminierter Stream

***fputs(int c, stream) -> status***//Konvertiert int in char schreibt in stream, return c/EOF

***fputc(text, stream) -> status***//Schreibt alle Bytes von text in stream, bis 0 kommt

***fungetc(c, stream) -> status***//Schiebt c zurück in den Stream

***ftell(stream) -> offset***//Gibt Index zurück

***fseek(stream, offset, origin) -> status***//analog lseek in POSIX

***rewind(stream): fseek(stream, 0, SEEK\_SET), clear errno***//reset Stream

File einleisen und Text verdoppeln - POSIX

int fd1 = open("test.txt", O\_RDWR);

void \* buffer = malloc(4096);

read(fd1, buffer, 4096);

write(fd1, buffer, sizeof(buffer));

A – Z in File schreiben- POSIX

int fd = open(argv[1], O\_RDWR | O\_CREAT);

for (char c = 'A'; c <= 'Z'; ++c)

{

write (fd , &c, sizeof c);

}

File copy- POSIX

int copy\_file(fd1, int fd2, char \*buffer, int size)

{

int ret = 1;

do

{

ret = read(fd1, buffer, size);

write(fd2, buffer, ret);

}while(ret > 0);

File copy- C

FILE \* eingabefile = fopen ("test.txt", "r");

FILE \* ausgabefile = fopen ("test2.txt", "w");

int buffer;

do

{

buffer = fgetc (src);

if (buffer == EOF)

{

break;

}

fputc (c, dst);

}

while (1);

fclose (src);

fclose (dst);

## Inode Berechnungen

1024 in hex = 400 & s\_log\_block\_size 0x18:

00000410: 45fe 0000 0000 0000 **0200 0000** 0200 0000

= 0000’ 0020 🡪daraus folgt Blockgrösse 2^12

0x58         \_\_le16         s\_inode\_size         Size of inode structure, in bytes.

00000450: 0000 0000 0b00 0000 **0001** 0000 3c00 0000

0100 --> 256 Byte

**Indirekter Block** beinhaltet 32 bit Einträge ab Block 12, daraus ergibt sich:

Blockgrösse 1024 Byte(213b) / 32b = 256 Blocknummern 🡪 Block 12 bis 267

Blockgrösse 4096 Byte(215b) / 32b = 1024 Blocknummern 🡪Block 12 bis 1035

**Doppelt** **Indirekter** **Block**

Blockgrösse 1024 (213b) 🡪256 Einträge je Block, 256 Blöcke = 2562 = 65536 Blöcke 🡪Blöcke 268 bis 65803

Blockgrösse 4096 (215b) 🡪1024 Einträge je Block, 1024 Blöcke = 10242 = 1M Blöcke 🡪 Blöcke 1036 bis 1’049’611

**Dreifach indirekter Block**: enthält Nummern von doppelt

Blockgrösse 1024 → 256 \* 256 \* 256 = 16M Blöcke referen-

ziert → Blöcke 65804 – 16843019

Max Dateigrösse dann: 1024 \* 16M = 16GB

Blockgrösse 4096 🡪 4096 \* 10243 = 4,4 TB

**Aufgaben**

Eine Datei hat 4MB, wie viele Blöcke werden gebraucht? Blockgrösse 1024Byte

4 MB = 222B / 210B = 212Blöcke

Eine Datei hat 2 KB, gibt es indirekte Blöcke? Wenn ja, wie viele?

8 KB = 213B / 210  = 8 Blöcke 🡪 es gibt keine indirekten Blöcke

Eine Datei hat 1 MB, gibt es doppelt Indexierung(bei Blockgrösse 4096B)

1MB = 220 / 212 = 28 = 256 Blöcke 🡪 Es gibt einen Doppelten indirekten Block

## Shared Objects

***void \* dlopen (char \* filename, int mode)*** //Öffnet dyn. Bibliothek, gibt Handle zurück

***mode:*** RTLD\_NOW: Alle Symbole werden beim Laden der Bibliothek gebunden

RTLD\_LAZY: Symbole werden bei Bedarf gebunden RTLD\_GLOBAL: Symbole können beim Binden anderer Objekt-Dateien verwendet werden RTLD\_LOCAL: Symbole werden nicht für andere Objekt-Dateien verwendet

***void \* dlsym (void \* handle, char \* name)*** //Gibt die Adresse des Symbols «name» aus der mit «handle» bezeichneten Bibliothek zurück. Man weiss nicht, ob es sich um Funktion oder Variable handelt, da man nur Adresse erhält!

**typedef int (\*func\_t)(int);** / Erstellen Typ -> Pointer auf eine Funktion mit int als Argument und int als Rückgabewert

**func\_t f = dlsym(handle, "my\_function");** //Speichert in f die Adresse des Symbols «my\_function» in der mit handle bezeichneten dynamischen Bibliothek.

**int \*i = (int \*) dlsym(handle, "my\_int");** // Speichere Adresse von «my\_int»

**(\*f)(\*i);** // Funktionsaufruf – Es wird die Funktion aufgerufen, auf welche f zeigt, als Argument wird der Wert der Adresse mitgegeben, auf die i zeigt.

***int dlclose (void \* handle)*** //Schliess das durch handle bezeichnete Objekt

***char \* dlerror ()*** //return Fehlermeldung(nullterminierter String), wenn Fehler war

## X Window

***Display \* XOpenDisplay (char \*display\_name)*** //Öffnet Verbindung zum Display mit display\_name, ist dieser Name = NULL wird die Umgebungsvariable DISPLAY ausgelesen

***void XCloseDisplay (Display \*display)*** //Schliesst Verbindung, entfernt ass. Ressourcen

***XCreateSimpleWindow*** //erzeugt ein(simples) Fenster auf einem Display

***Parameter***: Display, Parent Window, Koordinaten der oberen linken Ecke,Breite und Höhe, Breite des Rands (in Pixeln), Stil des Rands, Stil des Fensterhintergrunds

***XDestroyWindow***//entfernt Fenster mit allen Unterfenster

***XMapWindow (Display \*, Window)*** //Bestimmt, dass ein Fenster auf Display angezeigt wird. Fenster erst angezeigt, wenn Parent Fenster angezeigt wird.

***XMapRaised (Display \*, Window)*** //Wie XMapWindow aber bringt Fenster vor alle

***XMapSubwindows (Display \*, Window)*** //Zeigt alle Unterfenster an

***XUnmapWindow (Display \*, Window)*** //Versteckt Fenster und alle seine Unterfenster

***XUnmapSubwindows (Display \*, Window)*** //Versteckt alle Unterfensters

***XSelectInput (Display \*display, Window w, long event\_mask)*** //Legt als Maske fest, welche Events ausgewählt werden(mehrere möglich) – Masken vordefiniert. Pro Fenster individuell, nicht gewünschte Fehlertypen gehen ans übergeordnete Fenster

***XNextEvent (Display \*display, XEvent \*event)*** //Kopiert das nächste Event aus dem Buffer in event 🡪 Normalerweise in Endlosschleife 🡪 while(1)

Jedes Element in einem XEvent ist so gross wie grösster Union-Member 🡪Felxibilität

***XCreateGC(Display \*, Window, 0, Null)*** //Erstellt neuer Grafik Kontext

***XFreeGC (Display \*, GC gc);*** //Entfernt Grafik Kontext

***XDrawRectangle(s), XDrawString, XDrawText***

***XSetWMProtocols (Display \*, Window, Atom \*first\_protocol, int count)*** //Speichert die Anzahl count Atome in WM\_Procolls.

### Atom erstellen und hinzufügen

#### Atom atom = XInternAtom (display, "WM\_DELETE\_WINDOW", False);

Es wird die ID des Atoms WM\_DELETE\_WINDOW vom System abgefragt und im

gespeichert.

#### XSetWMProtocols (display, window, &atom, 1); //Properte hinzufühen zu WMProtocols

### Event auslesen/verarbeiten

#### switch (event.type) {

#### case ClientMessage:

#### if (event.client->data.l[0] == atom) { ... }

#### break;

#### }

### String erstellen, analog dazu Rectange

#### XDrawString

#### (

#### event->display,

#### event->window,

#### this->gc,

#### 10, 10, // top left corner

#### text,

#### sizeof text - 1 // length of string (without terminating 0)

#### );

Display Pointer weil Zeiger auf Datenstruktur(struct) von Bildschrim 🡪 Verbindungsinformationen

## Funktionen

***Colormap DefaultColormap(Display \*, Screen)*** //Gibt die Default Color Map zurück

***XParseColor (Display \*, Colormap, char \* specification, XColor \*color)*** //Parst den String Spezifikation(RGB)

***XAllocColor (Display \*, Colormap, XColor \*color)*** //Legt die Farbe in der ColorMap an

***XChangeWindowAttributes (Display \*, Window, unsigned long mask, XSetWindowAttributes \*)*** //Ändern der Attribute, welche in der Mask angegeben wurden

***XSetWindowBackground(Display \*, Window, unsigned long index)*** //Fenster

***XSetWindowBorder(Display \*, Window, unsigned long index)*** //Rahmen Fenster

***XSetForeground(Display \*, GC, unsigned long index)*** //Vordergrund GC

***XSetBackground(Display \*, GC, unsigned long index)***// Hintergrund GC(z.B Strich)

***XClearWindow (Display \*d, Window w)*** //Lösch Fenster

### Zuerst Farbe Parsen und dann anlegen!