Programmiersprache C

Toolchain

Präprozessor: entfernt alle Kommentare, ersetzt alle Makros
Output: reine C-Datei/Translation-Unit
Compiler: übersetzt Translation-Unit nach Assembler
- erstellt Abstract Syntax Tree(AST) (=Programm) werden

Output: Assembly file(.s, mit Referenzen auf externe Variablen/Funktionen
Assembler: übersetzt Text-Assembler in Binärdatei (*Objekt-Datei .o*, Referenzen auf externe Variablen/Funktionen)
Linker: Auflösung von Referenzen
- Statische Bibliotheken: noch Referenzen auf externe Variablen/Funktionen
- Executables/dynamische: vollständig aufgelöst

Output: Bibliotheken (statisch/dynamisch), Executable

Sprache

Operatoren

& Adresse, * Wert an Adresse
Logisch AND, OR: &&, ||
Bitweises AND, OR, XOR, Negation: &, |, ∧, ∼
Zuweisungen eines Structs auf einen anderen → ganzer Inhalt kopiert(nicht nur Referenz)
`write();` schreibt sofort, `fflush(stdout)` Buffer leeren
%i `int`, %X `unsigned int` als Hex, %li `long`, %lli `long long`, %p `void*`, %c int schreibt char %s `char*`

OS API

Aufgaben OS

- Abstraktion/Portabilität von Hardware, Protokolle, Software-Services
- Resourcenmanagement/Isolation der Anwendungen voneinander (Rechenzeit, Hauptspeicherverwaltung, Sek. Speicher, Netzwerkbandbreite)
- Benutzerverwaltung/Sicherheit

Prozessor Privilege Level

mind. 2 Privilege Levels auf Prozessor: Kernel Mode, User Mode
Kernel bestimmt in welchem Modus ein Programm läuft (Entscheid somit softwareseitig)

Wechsel vom User Mode in Kernel Mode

syscall Instruktion notwendig → Prozessor schaltet in Kernel-Mode um → setzt Instruction Pointer auf System Call Handler
jede OS-Kernel-Funktion hat somit einen Code. Dieser wird in Register übergeben. Je nach Funktion in anderen Register weitere Infos.
Linux-Kernel nicht binärkompatibel wegen unterschiedlichen Calling Conventions (anderer syscall Code/anderes Register) → Appl. für jeden Kernel einzeln kompilieren, C-API (auf Quellcode, ABI = interface auf binary) verwenden

Programmargumente

Argumente vom OS in Speicherbereich des Programms als Array mit Pointern auf null-terminierte Strings
`main (int argc, char** argv)`: `argc` Anz. Argumente, `argv` Pointer auf Array mit Strings(`char*`), `argv[0]` Programmname!

Umgebungsvariablen

Umgebungsvar. vom OS in Speicherbereich des Programms kopiert als Array mit Pointern auf null-terminierte Strings (wie Programmarg.)
Gemäss POSIX jeder Prozess eigene Umgebungsvariablen
environ →environ[0]→ Key0=Value0
String: PATH=/home/hsr/bin, Key (unique) Value
Umgebungsvariablen initial vom erzeugenden Prozess festgelegt (z.B. shell)

API

nie direkt über environ!
`char * getenv (const char * key)` Adresse 1. Zeichens, 0 nichts gefunden
`int setenv(const char *key, const char *value, int overwrite);`
overwrite != 0 ganzer Wert wird mit neuem String übschrieben
`int unsetenv(const char *key);` entfernt Umgebungsvariable
`int putenv (char * kvp)` ersetzt mit Pointer, keine Kopie (wie set)!

Einfügen von neuen Umgebungsvar.→OS legt neuen grösseren Speicher an, kopiert Array dorthin

Prozesse

Monoprogrammierung: 2 SW-Akteure (OS, Programm), Programm kennt nur OS & sich selbst (ist isoliert) **Quasi-Parallel:** Programme gleichzeitig in Hauptspeicher, Ausführung nacheinander, für Isolation: jeder Prozess virtueller Adressraum **Prozess umfasst:** Abbild des Programms (text section), globale Var. (data section), Speicher für Heap (startet bei kleinster Nr.)↔Stack (startet bei grösster) **Eigenschaften Prozess:** eigener Adressraum, frei Registerbelegung, Isolation (gut für unabhängige Appl.) - gemeinsame Ressourcen schwierig, grosser Overhead für Prozesserzeugung, Realisierung Parallelisierung aufwändig

Process Control Block (PCB)

OS benötigt Daten für Integration des Prozesses im Gesamtsystem, PCB eines Prozesses beinhaltet: Eigene ID, Parent ID andere wichtige IDs — Speicher Zustand Prozessor — Scheduling-Infos — Daten für Sync/Kommunikation zwischen Prozessen — Filesystem-Infos — Security-Infos

Interrupts

Auftreten eines Interrupts, Ablauf:
1. context safe: Register, Flags, Instruction Pointer, MMU-Config(Page-Table-Pointer)
2. Aufruf Interrupt-Handler, kann Kontext überschreiben
3. context restore: Wiederherstellung des Prozesses aus PCB

Kontext-Wechsel sehr teuer (viele Register inv.), Cache hier als Nachteil→alles muss gewechselt werden

Prozesshierarchie

Tool: `pstree`, jeder Prozess: 1 Parent-Prozess, beliebige Anz. Child-Prozesse — sytemd (PID 1): init Prozess→Starten, Beenden & Überwachen von Prozessen

API

exakte Kopie des Prozesses, ausser: Child hat eigene/andere Prozess-ID

```
pid_t pid = fork();//Rücksprung beide
if (pid > 0){/*Parent Code*/}
else if (pid == 0){/*Child Code*/}
//-1 Fehler, errno
```

`pid_t wait (int * status)` unterbricht Prozess bis 1 Child-Prozess beendet — `status` Out-Parameter, Abfrage durch Makros
`pid_t waitpid (pid_t pid, int * status, int options)`
`pid == -1` auf irgendein Child-Prozess, wie `wait()`

gerade laufender Prozess: Programmimage ersetzt durch anderes

Suche des Programms	Umgebungsvariablen	Programmargumente als Liste	als Array
Angabe des absoluten /	mit neuem Environment	execle	execve
realtiven Datei-Pfads	mit altem Environment	execl	execv
Suche über PATH		execlp	execvp

`unsigned int sleep (unsigned int seconds)`
durch Signale unterbrochen, Anz. verbleibende Sek. zurück

`void exit (int code)`
`int atexit(void (*function)(void))`
Aufräumfunktion mitgeben, in umgekehrter Reihenfolge nach Exit ausgeführt

`pid_t getpid(void)`, `pid_t getppid(void)`

Zombieprozess

Child zwischen seinem Ende und Aufruf von `wait()` Zombie
Parent verantwortlich, OS behält Statusinfos bis zum Aufruf **Dauerhafter Zombie:** Parent ruft wait nicht auf (vermutlich Fehler), Lösung: Parent stoppen → Childs werden zu Orphants

Orphanprozess

Parent Prozess beendet → alle Child-Prozesse verwaisen, werden an Prozess Nr. 1 übergeben systemd: ruft wait in Endlosschleife auf

Threads

parallel ablaufende Aktivitäten innerhalb Prozess, Geteilte Ressourcen: text section, data section, Heap, geöffnete Dateien, MMU-Infos — jeder Thread eigener Stack + Kontext (da unterschiedliche Stadien, eigene Funktionsaufrufkette)→Thread-Control Block

Amdahls Regel

n Anzahl Prozessoren
T Ausführungszeit, wenn komplett seriell ausgeführt
T' Zeit, wenn max. parallelisiert ($T_s + \frac{T-T_s}{n}$)
T_s Zeit, der seriell ausgeführt werden muss
$T - T_s$ Zeit, die parallisiert werden kann
$\frac{T-T_s}{n}$ Parallel-Anteil verteilt auf n Prozessoren
$s = \frac{T_s}{T}$ serieller Anteil Algorithmus

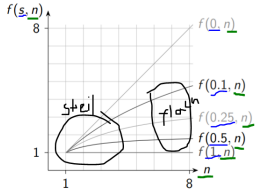
Speedup-Faktor

$$f \leq \frac{T}{T'} = \frac{T}{T_s + \frac{T-T_s}{n}} = \frac{T}{s \cdot T + \frac{T-s \cdot T}{n}} = \frac{T}{s \cdot T + \frac{1-s}{n} \cdot T} = \frac{1}{s + \frac{1-s}{n}}$$

parallele Variante max. f-mal schneller als serielle

Bedeutung

Abschätzung einer oberen Schranke/max. Geschwindigkeitsgewinn
Nur wenn alles parallelisierbar ist, ist Speedup proportional und maximal $f(0, n) = n$
Sonst Speedup mit höherem n geringer



Mit höherer Anz. Prozessoren nähert sich Speedup $\frac{1}{s}$ an: $lim_{n \to \infty} \frac{1}{s + \frac{1-s}{n}} = \frac{1}{s}$

POSIX Thread API

```
int pthread_create (
pthread_t * thread_id, //Out-Parameter
pthread_attr_t const * attributes, //0 -> default
void * (* start_function ) ( void *) ,//1. Instruktion v. Thread
void * argument )//Argumente an Funktion, Pointer auf Heapobj.
```

Attribut angeben, Vorgehensweise: **Lebensdauer/Beendigung Threat:** springt aus `start_function` zurück, ruft `pthread_exit` auf, anderer Thread ruft `pthread_cancel` auf, Prozess wird beendet

```
void pthread_exit(void * return_value)//gleicher Wert wie start_function
int thread_cancel(pthread_t thread_id) 0 = existiert, ESRCH existiert
```

nicht, Funktion wartet nicht bis Thread tatsächlich beendet wurde

```
int pthread_detach(pthread_t thread_id)
```

entfernt Speicher, den Thread belegt hatte aber beendet nicht

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

Wartet bis Thread beendet, Rückgabe wie create oder exit (0 keine)

```
pthread_t pthread_self (void)
```

ID laufenden Threats

Scheduling

1 Prozessor max. 1 Thread (= *running*), *ready* (alle in Ready-Queue), *waiting*
Powerdown-Modus: Wenn kein Thread *ready*, Prozessor vom OS in Standby, Interrupt → wieder normale Operation

Laufzeit eines Threats

Umsetzung eines nebenläufigen Systems: *kooperativ* → *Thread entscheidet*, präemptiv → Scheduler entscheidet

Ausführungsarten

Parallel: Alle Threads gleichzeitig: für n Threads n Prozessoren, **Quasiparallel:** n Threads auf $< n$ Prozessoren abwechselnd (es entsteht der Eindruck es sei parallel), **Nebenläufig:** Oberbegriff für Parallel/Quasiparrallel

Scheduling-Scope

Process-Contention Scope: Alle Threads innerhalb des aktiven Prozesses berücksichtigt **System-Contention Scope:** Alle Threads des gesamten Systems berücksichtigt

Scheduling-Strategien

Anforderungen an Scheduler

Aus Sicht Applikation/Offene Systeme: Durchlaufzeit (Start &Ende Threat), Antwortzeit (Empfang Request bis Antwort), Wartezeit (Zeit in Ready-Queue)
Geschlossene Sys./Embedded/Server: Durchsatz (Anz. Threads pro Interall bearbeitet), Prozessorverwendung (% Verwendung gegenüber Nichtverwendung), Latenz (durchschnittliche Zeit Auftreten & Ereignis verarbeiten)

Prioritäten-basiertes Scheduling

Jeder Thread eine Nr., Threads mit gleicher Prio → FCFS
Risiko→Starvation, Thread mit niedriger Prio läuft unendlich lange nicht, Lösung: Aging (in best. Abständen Prio um 1 erhöht)

Multi-Level Scheduling

nach bestimmten Kriterien in verschiedene Level (z.B. Priorität, Prozesstyp, Hinter- oder Vordergrund), fürs jedes Level eigene Queue, jedes Level kann eigenes Verfahren haben, Queues können priorisiert werden

Multi-Level Scheduling mit Feedback

Je Priorität eine Ready-Queue, Threads aus Queue mit höherer Prio bevorzugt, Wenn mehr als Level-Zeit benötigt→ Prio -1 (Thread landet in Queue mit niedriger Prio) (wenn benötigte Zeit = Level-Zeit → bleibt auf altem Level), Queue mit niedriger Prio → länger, Threads mit kurzen Prozessor-Bursts werden bevorzugt

Synchronisation

Jeder Thread hat eigener Instruction Pointer, IPs werden unabhängig voneinander bewegt (auch bei Parallelisierung, z.B. wegen Speicherzugriffen)

Producer-Consumer-Problem

Threads arbeiten unterschiedlich schnell, Ring-Buffer begrenzt gross
Race-Conditions **atomare Instruktion:** 1 Instruktion, vom Prozessor unterbrechungsfrei ausführbar
`++counter` = 3 Instrk., `inc reg1` = 1 Instrk.
Race-Condition: Ergebnisse abhängig von Ausführungsreihenfolge einzelner Instruktionen
Nebenläufige Threads im Wettrennen um Hauptspeicher→Thread-Snych., ausschliessen von Threads notwendig

Critical Section

Critical Section: Code-Bereich der mit anderen Threads geteilt wird
Anforderungen: Gegenseitiger Ausschluss (nur 1 Thread in Sect.), Fortschritt (Welcher Thread ist nächster?),
Begrenztes Warten (Thread nur n-mal übergangen, n fix)
Computer-Arch. → keine Garantien: Instruktionen nicht atomar, Sequenzen werden umgeordnet

Mögliche Synchmechanismen mit Hardwaresupport

1. Interrupts abschalten

Alle Interrupts abgeschaltet, wenn in Critical Section
System mit 1 Prozi: effektiv, kommt zu keinem Kontext-Wechsel
Mit mehreren: Problem: parallele Threads, geht nicht!!
Generell: OS kann Thread nicht unterbrechen

2. Verwendung von Instruktionen

3. Semaphore

Zähler z, `post`: `z++`, `wait`: `z--` falls $z > 0$ sonst Thread→waiting
Bsp. für Producer/Consumer, kein Busy-wait mehr

API

```
sem_t sem; //globale Variable
main{
sem_init(&sem, 0/*nur innerh. Proz. verwendet*/, 4/*init z*/)}

int sem_wait(sem_t *sem);//beide: 0->ok, -1 + errno->Fehler
int sem_post(sem_t *sem);//Fehler-> Semaphore bleibt gleich
```

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

```
int sem_destroy ( sem_t * sem );
sem_getvalue (sem_t * sem, int * out_param);
```

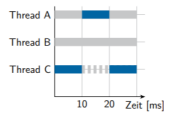
Priority Inversion

Grund: gemeinsam verwendete Ressource hat niedrigste Prio.
Voraussetzungen:

- Ein hoch-priorisierter Thread wartet auf eine Ressource, die von einem niedriger priorisierten Thread gehalten wird
- Ein Thread mit Priorität zwischen diesen beiden Threads erhält den Prozessor

Auswirkung: zugewiesene Prios ≠ effektive Prios → Inversion

Priority Inheritance



Temporäre Erhöhung der Prio:
Thread A hat niedrige Prio und hält einen Mutex M, Thread B mittel, Thread C hohe und läuft gerade, Nach 10ms benötigt C M, Prio von A temporär auf Prio von C/B nicht ausgeführt, A läuft, bis Freigabe von M, Dann läuft wieder C

4. Mutexe

Acquire/Lock: Wenn z = 0: `z = 1`, fahre fort — wenn z = 1: blockiere Thread bis z = 0 **Release/Unlock:** setzt z = 0

Interprozess-Kommunikation (IPC)

Signale

ermöglichen Unterbruch eines Prozesses von aussen
wird vom OS wie ein Interrupt behandelt

Quelle von Signalen

Hardware/OS: Ungültige Instruktion, Zugriff auf ungültigen Speicherbereich (*segmentation fault*), Division durch 0 **Andere Prozesse:** Ctrl-C, `kill`-Kommando

Signale behandeln

Jeder Prozess pro Signal 1 Handler (bei Prozessbeginn Default-Handler)

Ignore-Handler	ignoriert Signal
Terminate-Handler	beendet Programm
Abnormal-Terminate-Handler	beendet + Core Dump(Speicherauszug)

Ausser SIGKILL & SIGSTOP alle Handler überschreibbar

Wichtige Signale

Programmfehler→Abnormal-Terminate-Handler
SIGFPE Fehler in arithmetischer Operation SIGILL Ungültige Instruktion SIGSEGV Ungültiger Speicherzugriff SIGSYS Ungültiger Systemaufruf
Prozesse abbrechen→Terminate-Handler
SIGTERM normale Beendigungsanfrage `kill 1234` SIGINT nachdrücklichere Aufforderung `Ctrl-C` SIGQUIT anormale Terminierung `Ctrl-\` (`Ctrl-Alt Gr-<`) SIGABRT anormale Terminierung (vom Prozess selber bei Programmierfehler) SIGKILL letzte Möglichkeit, kann nicht blockiert/ignoriert/abgefangen werden
Stop und Continue
SIGTSTP versetzt in Zustand *stopped*, ähnlich zu *waiting* `Ctrl-Z` SIGSTOP wie SIGTSTP, kann nicht abefangen/ignoriert werden SIGCONT setzt Prozess fort

Signalhandler ändern

```
int sigaction (//um Handler für Signal anzumelden
int signal, //Nr. des Signals, welches man handeln will
struct sigaction * new,
struct sigaction * old)//0 wenn unerwünscht

struct sigaction{
void (* sa_handler )( int );//Handlerfunktion
sigset_t sa_mask;//alle zu blockierende Signale
int sa_flags;};
//sigset = Menge aller zu blockierender Signale
```

Message-Passing

Direkte Kommunikation

Sender muss Empfänger kennen `send (receiver, message)`
symmetrisches Empfangen: Empfänger muss Sender kennen Asymmetrisches Empfangen: Empfänger erhält ID in Out-Parameter (kennt Sender nicht)

Indirekte Kommunikation

Beide Teilnehmer müssen gleiche Mailbox/Port/Queue kennen
Mehrere Mailboxen zwischen Sender/Receiver möglich
Queue gehört zu Prozess oder zu OS(Lösch/Erzeugmechanism.)

Synchronisation

blockierend (synchron)/nicht-blockierend(asynchron)
synchrones Senden Sender blockiert bis Nachricht empfangen
synchrones Empfangen Empfänger blockiert bis Nachricht verfügbar
→Alle Kombinationen möglich (z.B. synchroner Sender/asynchroner Receiver)

Rendezvous

Sender & Empfänger blockierend
OS kann direkt vom Sende- in Empfängerprozess kopieren (meistens ungepuffert) (implizite Synch, Impl. Producer/Consumer-Problem)

POSIX API

- Message-Queues vom OS
- variable Nachrichtenlänge, Maximum pro Queue einstellbar
- synchrone/asynchrone Verwendung
- Prioritäten

```
mqd_t //Message-Queue-Descriptor
mqd_t mq_open (const char * name , int flags,
mode_t mode , struct mq_attr * attr);//attr=0 -> Default-Attr.
//Flags = O_RDONLY, O_CREAT, O_NONBLOCK
int mq_close(mqd_t queue);//bleibt im OS bis Entfernung
int mq_unlink(const char * name);//wird entfernt, wenn keine Proz.
int mq_send ( mqd_t queue , const char * msg ,
size_t length , unsigned int priority);//blockiert wenn Queue voll
int mq_receive ( mqd_t queue , const char * msg,
size_t length , unsigned int * priority);//blockiert, wenn leer
//length mind. solange wie max. Grösse Nachricht
```

Shared Memory

Frames des Hauptspeichers werden zwei Prozessen freigegeben:

- In P1 wird Page V1 auf einen Frame F abgebildet
- In P2 wird Page V2 auf denselben Frame F abgebildet

Beide Prozesse können beliebig daraufzugreifen

Verwendung von Pointern: Sollen Pointer verwendet werden, müssen diese relativ zu einer Anfangsadresse sein (Offset auf Startadresse)

→beide Varianten liegen bei Mehr-Prozessoren-Systemen gleichauf, Message-Passing vermutlich perfomanter in Zukunft

Dateisysteme-API

Referenzen

. → auf sich selbst, .. → auf Elternverzeichnis
Jeder Prozess hat Arbeitsverzeichnis. Bezugspunkt für relative Pfade. Wird von aussen festgelegt.

Pfadarten

Absolut: beginnt bei Root (/) **Relativ:** beginnt mit Arbeitsverzeichnis **Kanonisch:** ohne `.` oder `..`, Ermittlung mit `realpath`

Zugriffsrechte

jede Datei/jedes Verzeichnis gehört einer Gruppe & einem Benutzer (Owner)
1 Oktal-Zahl/3 Bit-Stellen für Owner, Gruppe und Andere
r: 4, 100 w: 2, 010 x(execute): 1, 001
rwx------ → 0700, 111

API

File-Descriptor: gilt nur innerhalb Prozess, Index auf Filedeskriptor-Tabelle, integer
File-Descriptor-Table of Process: Element enthält Index in die systemweite Tabelle, Zustandsdaten (Offset)
Global Descriptor Table: enthält Daten um physische Datei zu indentifizieren (richtiger Treiber, Datenträger etc.)

POSIX API

alle Daten sind rohe Binärdaten (wie abgespeichert)

```
lseek (fd, offset, origin) //return status
//Offset des FDs auf offset setzen
pread(..)/pwrite(..)
//mit Offset-Angabe, verändern FD nicht
```

C API

formatierte Ein- und Ausgabe (via Streams(= `FILE`)), **File-Position-Indicator:** gepuffert → bestimmt Position im Puffer, ungepuffert → Offset des File-Descriptors

Dateisysteme EXT2 und EXT4

Partition: Teil eines Datenträgers, wird selbst wie ein Datenträger behandelt **Volume:** Datenträger oder Partition **Sektor:** kleinste logische Untereinheit eines Volumens, Daten als Sektoren transferiert, Grösse durch HW bestimmt, enthält Header, Daten und Error-Correction-Codes **Format:** Lyout der logischen Strukturen, vom Dateisystem definiert

Block/Inodes

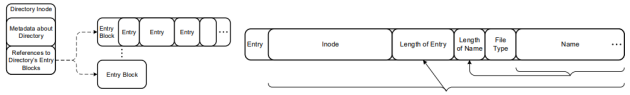
Blockgrösse: 1 KB, 2KB oder 4KB (Standard)
Block enthält nur Daten einer einzigen Datei
Inodes-Grösse: fixe Grösse pro Volume, 2er-Potenz, mind. 128 Byte, max. 1 Block

Anzahl referenzierter Blöcke

Blockliste (60 Byte): 15 Blocknr. à 32 Bit
Anzahl abhängig von der Blockgrösse:
Index 0-12 **Indirekter Block:** Blockgrösse in Bits/32 Bit
Index 13 **Doppelt indirekter Block:** (Blockgrösse in Bits/32 Bit)^2
Index 14 **Dreifach indirekter Block:** (Blockgrösse in Bits/32 Bit)^3

Verzeichnisse

Inode, dessen Datenbereich Entries enthält
automatisch angelegte Entries: `.` — eigener Inode gespeichert, `..` — Inode des Elternverzeichnisses



Entries

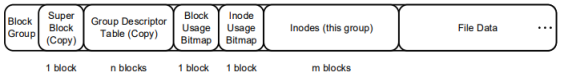
Länge variabel 8 - 263 Bytes, aber immer Vielfaches von 4 Bytes
4 Bytes Inode, 2 Byte Length of Entry, 1 Byte Length of Name, 1 Byte File Type (1=Datei, 2=Verzeichnis, 7=Symbolischer Link), 0-255 Byte Name (Ascii)

Links

Hardlink: Inode ist gleich, Pfade sind verschieden **Symbolischer Link:** wie Datei, die Pfad auf andere Datei enthält, (Pfad ¡ 60 Zeichen: Pfad direkt in Array gespeichert, ohne Blockallokation, sonst Bockallokation)

Blockgruppe

Volume wird in Blockgruppen unterteilt
Gruppengrösse bis zu Faktor 8 der Anzahl Bytes pro Block
z.B. Blockgrösse 4 KB → Gruppegrösse: $2^2 KB \cdot 2^3 = 2^5 K$ Blöcke pro Gruppe
Anzahl Blöcke pro Gruppe für alle Gruppen gleich



Superblock

enthält alle Meta-Daten übers Volume (Anzahlen, Bytes pro Block etc., verschiedene Zeitpunkte, verschiedene Statusbytes, erster Inode, Feature-Flags) startet immer an Byte 1024 (wegen evtl. Boot-Daten davor)

Gruppendeskriptor

32 Bytes, Beschreibung einer Blockgruppe (Blocknummern Bitmaps/Inode-Tabelle, Anzahl freier Inodes/Blöcke, Anzahl Verzeichnisse pro Gruppe)

Sparse Superblocks

Die Kopien des Superblocks & Group Descriptor Table werden nur noch in Blockgruppe 0 & 1, sowie in allen reinen Potenzen von 3/5/7 gehalten

Lage des Superblocks

Blockgruppe 0 enthält immer Superblock
Blockgrösse 1024: Block 0 kommt vor Blockgruppe 0, Block 1 ist in Blockgruppe 0, Superblock in Block 1 Blockgrösse >1024: Block 0 in Blockgruppe 0, Superblock in Block 0

Lokalisierung eines Inodes

Alle Inodes gelten als eine grosse Tabelle
Inode-Nr. beginnen bei 1
Blockgruppe = (Inode - 1)/Anz. Inodes pro Gruppe
Indes des Inodes in Gruppe = (Inode - 1) % Anz. Inodes pro Gruppe
Sektor und Offset anhand Superblock

Ext4

Inodes 256 Bytes statt 128, Gruppendeskrip. 64 Bytes statt 32, Blockgrösse bis 64 KB

Extent Trees

Tree (60 Byte): 5 Elemente à 12 Byte, max. Tiefe 5

Journaling

Ablauf bei Dateierweiterung: Allokation neuer Blöcke, Anpassung der Inode, Anpassung Block-Usage-Bitmap/Counter freier Blöcke, Schreiben von Daten in Datei
System ohne Journaling: Muss alle Meta-Daten auf Inkonsistenzen überprüfen **mit Journaling:** nur Metadaten, welche im Journal sind
Journal Replay: Bei Systemneustart, Untersuch der Metadaten auf korrupte Werte anhand Journal

Modi

Journal: Metadaten & Dateiinhalte ins Journal + maximale Datensicherheit, - Geschwindigkeit **Ordered:** 1. Metadaten ins Journal 2. File Content direkt an endgültige Position 3. Commit + Dateien nach Commit richtigen Inhalt - geringere Geschwindigkeit **Writeback:** dito Ordered aber Commit und Schreiben der Daten in beliebiger Reihenfolge +sehr schnell -Dateien enthalten evtl. Datenmüll

Programme

Loader: lädt Executables & dynamische Bibliotheken in Hauptspeicher (statische vorher mit Executable/dynamischer verknüpft)

Systemcall `sys_execve`

sucht und öffnet spezifizierte Datei
zählt und kopiert Argumente/Umgebungsvariablen
Request an jeden Binary Handler
Binary Handler versucht Datei zu laden & interpretieren, wenn erfolgreich → Programm ausführen

Executable and Linking Format (ELF)

Binärformat, das Kompilate spezifiziert
Object-Files: Linking View, Programme: Execution View
Shared Objects (dynamische Bibliotheken): Linking/Execution View
Compiler erzeugt Sektionen, Linker Segmente (verschmilzt Sektionen gleicher Namens aus verschiedenen Object-Files)
Loader sieht nur Segmente
Header (52 Byte): Typ, 32-bit/64-bit, endianess, maschine, entrypoint (zeigt, wo Programm gestartet werden muss), relative Adresse/Anzahl/Grösse Einträge der Tables **Program Header Table:** Einträge zu 32 Byte; Einträge: Segment-Typ/Flags, Offset/Grösse der Datei, Virtuelle Adresse/Grösse im Speicher
Section Header Table: Einträge zu 40 Byte; Einträge: Name(Referenz auf String Table), Typ/Flags, Offset/Grösse der Datei, Infos spezifisch für Typ
String-Tabelle: Namen von Symbolen, keine String-Literale aus Programm(in `.rodata`)
Symbol-Tabelle: Einträge zu 16 Byte; Einträge: Name (4 Byte, Referenz in String-table), Wert (4 Byte, z.B. Adresse), Grösse (4 Byte, Grösse des Symbols), Info (4 Byte, z.B. Var/Arr/Funktion, lokal/global, Referenz Section-Header)

Bibliotheken

Benennungsschema

Linker-Name:	lib + Bibliotheksname + .so	libmylib.so
Shared Object-Name:	Linker-Name + . + Vers.nr.	libmylib.so.2
Real-Name:	SO-Name + . + Unterversnr.	libmylib.so.2.1

Shared Object-Name für Loader
`/usr/lib` + Linker-Name, Softlink auf → `/usr/lib` + SO-Name, Softlink auf → absoluter Speicherort `/usr/lib` + Real-Name
Versionnr. erhöhen, wenn Schnittstelle ändert
Unterversionsnr. erhöhen, wenn Schnittstelle bleibt (Bugfixes)

Implementierung

Dynamische Bibliotheken müssen verschiebbar sein
Code zwischen Programmen soll geteilt werden: nur einmal im Hauptspeicher → Shared Memory
Anwendung: Virtuelle Pages der Prozesse werden auf denselben Frame im RAM gemappt → Adressen müssen relativ sein! (position-independent)

Global Offset Table (GOT)

eine pro dynamischer Bibliothek/Executable
pro Symbol, das von anderer dynamischer Bibliothek benötigt wird, ein Eintrag
Im Code werden relative Adressen in die GOT verwendet.
Loader füllt zur Laufzeit "echte" Adresse in GOT ein.

Procedure Linkage Table (PLT)

implementiert Lazy Binding(Funktionen werden erst gebunden, wenn benötigt)
pro Funktion ein Eintrag
PLT-Eintrag enthält Sprungbefehl auf Stelle in GOT
GOT-Eintrag zunächst Proxy-Funktion
Proxy-Funktion sucht Link zu richtiger Funktion, überschreibt dann eigener GOT-Eintrag
Vorteil: Erspart bedingten Sprung

X-Window/GUI

programm-gesteuert, **ereignis-gesteuert**(event-driven)
X Window System: Grundfunktionen der Fensterdarstellung
Desktop Manager: Hilfsmittel wie File-Manager, Papierkorb etc.

Fensterverwaltung/Window Manager

Top-Level Window: Kind des Root-Window, gehören zu Applikation
Close-Button:

```
Atom atom = XInternAtom(display, "WM_DELETE_WINDOW", False);
XSetWMProtocols(display, window, &atom, 1);
```

Atom

ID eines Strings, der für Meta-Zwecke benötigt
`Atom XInternAtom (Display*, char*, Bool only_if_exists)`
Übersetzt String in Atom auf angegebenen Display

Properties

WM liest/setzt Properties auf Fenster
Property über Atom identifiziert
Zu jedem Property gehören Daten wie Liste von Atomen, ein/mehrere Strings

Protokolle Client↔WM

Client-Registrierung: im Property `WM_PROTOCOLS` Liste der Atome der Protokollnamen speichern
`XSetWMProtocols(Display*, Window, Atom* first_arrayelem, int array_l)`

X-Protocol

Festlegung Formate für Nachrichten XClient↔Server
Events: z.B. Mausklicks, Maus traversiert Fenstergrenze
Für Requests: Nachrichtenbuffer auf Clientseite
Pufferleerung: wenn client auf Server wartet, Client-Request Reply benötigt, `XFlush()` Für Events: doppelte Bufferung bei Server (checkt Netzwerk)/Client(nur selektierte Typen)

Encoding

CP in [D800, DFFF] für alle UTF-Codierungen nicht erlaubt

Unicode

Coderaum/Codepoints: 17 Ebenen à 2^{16} Punkte = 1'114'112 Punkte
Codepoint: Nummer eines Zeichens
Code-Unit: Einheit um Zeichen in Encoding darzustellen
CU-Länge: 8-Bit, 16-Bit, 32-Bit

UTF-8

Endianess egal!

Code-Point in	1	2	3	4
[0, 7F]	0xxx'xxxx			
[80, 7FF]	110x'xxxx	10xx'xxxx		
[800, FFFF]	1110'xxxx	10xx'xxxx	10xx'xxxx	
[1'0000, 10'FFFF]	1111'0xxx	10xx'xxxx	10xx'xxxx	10xx'xxxx

UTF-16

Code-Point in	
[0, FFFF]	Code-Unit = Code-Point
[D800, DFFF]	reserved (surrogate)
[1'0000, 10'FFFF]	1101'10([P_{20}, P_{16}]-1)[P_{15}, P_{10}]1101'11[P_9, P_0] in CU werden nur [P_{19}, P_0] geschrieben

2 CU's resultierend → Surrogate-Pairs

Meltdown

Ausgangslage

- Jeder Prozess hat eigenen virtuellen Adressraum
- Fordert ein Prozess einen OS-Service an, müsste man eigentlich den Kontext wechseln
- Macht er nicht, bleibt in aktuellem Kontext des Prozesses — Performancegründe!
  - OS mappt alle Kernel-Daten in Adressraum des Prozesses
  - Table-Config: nur OS kann daraufzugreifen
- OS muss auf alle Prozesse zugreifen können
  - Der OS-Kernel mappt den gesamten physischen Hauptspeicher in jeden virtuellen Adressraum

→Ganzer Hauptspeicher kann ausgelesen werden

Out-Of-Order Execution

Moderne CPUs optimieren → Reihenfolge der Ausführung kann sich ändern
O3E möglich, auch wenn Befehl später nicht ausgeführt wird:
mov rax, [0x1234]
jz label
mov rbx, 9 *Ergebnis wird verworfen, wenn übersprungen*
label:

Seiteneffekte O3E

```
char d = 65;
int * p = 0;
*p = 1; //Exceptiom kann nicht an Adresse 0 schreiben
char c = array [d]; //wird spekulativ ausgeführt
```

Cache kann man nicht auslesen, aber Zugriffszeitmessung möglich, kurz→Zeile war im Cache

Eigentlicher Angriff

Array als Hilfsmittel
Array in Assembler: [Startadresse + index * Variablengrösse in Byte]

1. clflush p für jedes Array-Element → Elemente werden aus Cache entfernt
2. via O3E auf Array zugreifen, kurze Zugriffszeit →Adresse mit verw. Daten gefunden

Array-Element-Grösse: Array-Elemente auf verschiedene Pages verteilen `array[i * 4096]`, da Chache optimiert sein könnte (mehrere Zeilen miteinander lädt)

Gegenmassnahmen

Kernel page-table isolation: Verschiedene Page-Tables für Kernel/User-Mode → negative Ausw. auf Perfomance

Spectre

Ziel wie Meltdown. Weitere Eigenschaft wird ausgenutzt:

Branch Prediction

Prozis lernen ob Sprung erfolgt oder nicht. Muss Prozi auf Sprungbedingung warten→Ausführung wahrscheinlicher Zweig

Angriffsflächen

- Alle Prozesse, die auf gleichem Prozi, haben die gleichen Vorhersagen→Vorhersagen können für andere Prozesse trainiert werden
- Opfer-Prozess muss zur Kooperation gezwungen werden →Ziel im verworfenen Branch auf Speicher zugreifen