Toolchain

Präprozessor: entfernt Kommentare, ersetzt Makros *Output: reine C-Datei/Translation-Unit*
Compiler: übersetzt Translation-Unit nach Assemble *Output: Assembly file* **Assembler:** übersetzt Text-Assembler in Binärdatei **Linker:** Auflösung von Referenzen *Output: Bibliotheken (statisch/dynamisch), Executable* **Loader:** lädt Executables & dynamische Bibliotheken in Hauptspeicher (statische vorher mit Executable/dynamischer verknüpft)

OS API

Aufgaben OS

- Abstraktion/Portabilität
- Resourcenmanagement/Isolation
- Benutzerverwaltung/Sicherheit

Prozessor Privilege Level

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

Wechsel vom User Mode in Kernel Mode

syscall → *Kernel-Mode* → *Instr. Pointer auf Call Handler*
Linux-Kernel nicht binärkompatibel

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.)
POSIX jeder Prozess eigene Umgebungsvariablen
Key (unique) Value
Umgebungsvariablen initial vom erzeugenden Prozess festgelegt (z.B. shell)
`putenv` ersetzt mit Pointer, keine Kopie (wie set)!

Prozesse

Monoprogrammierung: 2 SW-Akteure (OS, Programm) **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 PCB: 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, Instr. Pointer, MMU-Config
2. Aufruf Interrupt-Handler, kann Kontext überschreiben
3. context restore: Wiederherstellung des Prozesses aus PCB

Kontext-Wechsel sehr teuer

Prozesshierarchie

jeder Prozess: 1 Parent-Prozess, bel. Anz. Child-Prozesse

API

Kopie des Prozesses, ausser: Child hat eigene Prozess-ID
`wait` unterbricht Prozess bis 1 Child-Prozess beendet | `status` Out-Parameter, Abfrage durch Makros

Zombieprozess

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

Orphanprozess

Parent Prozess beendet → alle Child-Prozesse verwaisen, werden an Prozess Nr. 1 übergeben → 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

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

parallele Variante max. f-mal schneller als serielle

Bedeutung

Nur wenn alles parallelisierbar ist, ist Speedup proportional und maximal $f(0, n) = n$
Mit höherer Anz. Prozessoren nähert sich Speedup $\frac{1}{s}$ an: Attribut angeben, Vorgehensweise: **Lebensdauer/Beendigung Thread:** springt aus `start_function` zurück, ruft `pthread_exit` auf, anderer Thread ruft `pthread_cancel` auf, Prozess wird beendet
`pthread_join` Wartet bis Thread beendet, Rückgabe wie create oder exit (0 keine)

Scheduling

1 Prozessor max. 1 Thread (= *running*), *ready* (alle in Ready-Queue), *waiting* **Powerdown-Modus:** Wenn kein Thread *ready*

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
atomare Instruktion: 1 Instruktion, vom Prozessor unterbrechungsfrei ausführbar
Race-Condition: Ergebnisse abhängig von Ausführungsreihenfolge einzelner Instruktionen
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

`test_and_set oder compare_and_swap`

(Liest Wert aus Hauptspeicher & überschreibt im Hauptspeicher, falls erwarteten Wert entspricht)

3. Semaphore

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

Priority Inversion

gemeinsam verwendete Ressource hat niedrigste Prio.

Priority Inheritance

Thread mit Ressouce = MAX(Aller Threads)

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, (*segmentation fault*)
Andere Prozesse: Ctrl-C

Signale behandeln

Ausser SIGKILL & SIGSTOP alle Handler überschreibbar

Wichtige Signale

SIGTERM bittet Programm zu beenden
SIGKILL killt Programm (SIGKILL/SIGSTOP nicht überschreibbar)

Message-Passing

Direkte Kommunikation

Sender muss Empfänger kennen
symmetrisches: kennt vs. Asymmetrisches: erhält ID

Indirekte Kommunikation

Mailbox/Port/Queue kennen
Queue gehört zu Prozess oder zu OS(Lösch/Erzeugmechanism.)

Synchronisation

blockierend (synchron)/nicht-blockierend(asynchron)
→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

Sockets: bind, listen, accept, recv, send, close

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

Verwendung von Pointern: Pointer relativ zu einer Anfangsadresse sein
→beide Varianten liegen bei Mehr-Prozessoren-Systemen gleichauf, Message-Passing vermutlich perfomanter in Zukunft

Dateisysteme-API

Referenzen

. → auf sich selbst, . . → auf Elternverzeichnis
Jeder Prozess Arbeitsverzeichnis. Aussen festgelegt.

Pfadarten

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

Zugriffsrechte

1 Oktal-Zahl/3 Bit-Stellen für Owner, Gruppe und Andere
r: 4, 100 w: 2, 010 x(execute): 1, 001
rwx------ → 0700

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)

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

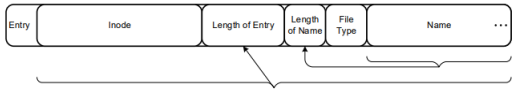
Blockgrösse: 1 KB, 2KB oder 4KB (Standard)
Block enthält nur Daten einer einzigen Datei
Inodes-Grösse: fixe Grösse pro Volume, 2er-Potenz, mind. 128 Byte, max. 1 Block

Anzahl referenzierter Blöcke

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

Verzeichnisse

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



Entries

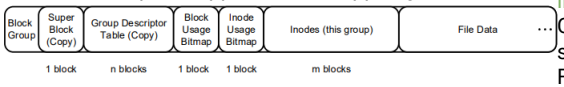
Länge variabel 8 - 263 Bytes, Vielfaches von 4 Bytes
4 Bytes Inode, 2 Byte Length of Entry, 1 Byte Length of Name, 1 Byte File Type (1=Datei, 2=Verzeichnis, 7=Symbolischer Link), 0-255 Byte Name (Ascii)

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: 32k Blöcke
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 (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

Lokalisierung eines Inodes

Alle Inodes gelten als eine grosse Tabelle
Inode-Nr. beginnen bei 1
Blockgruppe = (Inode - 1)/Anz. Inodes pro Gruppe
Index 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 + Grosse Dateien, Nur 1 Extent Speicher vs. Jede Blocknr.

Journaling

Ablauf bei Dateierweiterung: Allokation neuer Blöcke, Anpass. Inode, Anpassung Block-Usage-Bitmap/Counter freier Blöcke, Schreiben von Daten in Datei
System ohne Journaling: Muss alle Meta-Daten überprüfen **mit Journaling:** nur Metadaten, vom Journal
Journal Replay: Bei Systemneustart, Untersuch der Metadaten auf korrupte Werte anhand Journal **Journal:** Metadaten & Dateiinhalte ins Journal + maximale Datensicherheit, - Geschwindigkeit **Ordered:** 1. Metadaten ins Journal 2. File Content direkt an finale Position 3. Commit + Dateien nach Commit richtigen Inhalt - geringere Geschwindigkeit **Writeback:** dito Ordered aber Commit & Schreiben der Daten in beliebiger Reihenfolge +sehr schnell -Dateien evtl. Datenmüll

Programme

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 versch. 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`)

Bibliotheken

Lin.-Name:	lib + Biblio. + .so	libmylib.so
SO-Name:	Lin.-Name + . + V.nr.	libmylib.so.2
Real-Name:	SO-Name + . + SubV.nr.	libmylib.so.2.1

`/usr/lib`

X-Window/GUI

programm-gesteuert, **ereignis-gesteuert**(event-driven)
X Window System: Grundfnkt. der Fensterdarstellung
Desktop Manager: Hilfsmittel wie File-Manager, Papierkorb etc. **Display:** Rechner mit Tastatur + Zeigegerät + Bildschirme **Client:** Applikation, Display nutzen will **Server:** Teil von X Window System der Display ansteuert

Fensterverwaltung/Window Manager

Top-Lvl Win.: Kind Root-Win. → gehören Applikation

Atom

ID eines Strings, der für Meta-Zwecke benötigt
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

X-Protocol

Festlegung Formate für Nachrichten XClient↔Server
Events: z.B. Mausklicks, Maus traversiert Fenstergrenze
Für Requests: Nachrichtenbuffer auf Clientseite
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

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