1 Betriebssystem API

Aufgaben OS: Abstraktion von Hardware, Protokollen & Software-Services & damit Portabilität, Management von Ressourcen wie Rechenzeit, RAM-Verwaltung, sekundärem Speicher, Netzwerkbandbreite, usw., Isolation der Apps zueinander, Benutzerverwaltung, Security

Privilege Levels: Hardware-Mechanismus für Applikations-Isolierung, durch OS verwaltet, Kernelmodule des OS laufen im *Kernel Mode* & dürfen jede Instruktion ausführen, bei OS-Modulen & Apps im *User Mode* ist Menge Instruktionen beschränkt

Microkernel: nur kritische Kernel-Teile im Kernel Mode, alles inkl. Treiber im User Mode, stabil & analysierbar, Performance-Einbussen durch häufige Mode-Wechsel

Monolithische Kernel: viel Funktionalität inkl. Treiber im Kernel Mode, gute Performance wegen wenigen Mode-Wechseln, wenig Schutz bei Programmierfehlern

Unikernel: normales Programm als Kernel, Kernelfunktionalität in Library, keine Trennung von User / Kernel Mode, echt minimal & kompakt, Single Purpose

PL Wechsel: syscall Instruktion aus User Mode, Prozessor schaltet in Kernel Mode & setzt Instruction Pointer auf OS-Code (System Call Handler), so läuft im Kernel Mode immer Kernel-Code, jede OS-Kernel-Funktion ist mit einem Code versehen, welcher in einem Register übergeben wird, je nach Funktion müssen in anderen Registern noch andere Parameter übergeben werden

API: Application Programming Interface, abstrakte Schnittstelle, plattformunabhängig,gleich bei diversen OS

ABI: Application Binary Interface, konkrete Schnittstelle, Calling Convention, Abbildung von Datenstrukturen

Linux: Calling Convention gilt für spezifische Kernel-Version, nicht binärkompatibel, Apps pro Kernel kompilieren

API-Kompatibilität: Linux erfüllt, Apps sollten C-Wrapper-Funktionen statt syscalls aufrufen, dann kann der Compiler zum Kernel passenden Binär-Code generieren, ohne dass man den C-Code anpassen muss

POSIX: Standardisierung des OS-spezifischen Teils der UNIX/C-API, macOS konform, Linux mehr oder weniger, Windows nicht, viele Infos in man Pages

Shell: über Texteingabe OS-Funktionen aufrufen, keine besonderen Rechte, braucht Ausgabe-&Eingabe-Stream

Programmargumente: Aufteilung in Strings von Shell, Trennung mit Leerzeichen, OS legt null-terminiertes String-Array in Programm-Speicherbereich an, für Infos die bei jedem Aufruf anders sind, explizite Bereitstellung, argv[0]→Programmname, argv[1]→1. Argument, usw.

Umgebungsvariablen: Menge Strings eines Programms mit mind. einem =, Key=Value, Keys sind unique & case-sensitive, werden initial vom erzeugenden Prozess festgelegt (z.B. Shell), C-Variable *environ* zeigt auf null-terminiertes Array von Pointern auf null-terminierte Strings, für meist gleichbleibende Infos, implizite Bereitstellung, environ nicht direkt verwenden, sondern mit env-Fns

2 Dateisystem API

2 Bedeutungen Dateisystem: «Teil des OS, der für die Verwaltung der Datenträger zuständig ist» & «Die Strukturierung eines Datenträgers»

Datei: *Daten* (Byte-Sequenz) & *Metadaten* (Datei-Infos, sichtbar wie Grösse & nicht wie Datenträger-Ablageort)

Dateitypen: Dateiendungen haben keine Relevanz, durch Magic Numbers & Strings gekennzeichnet, Applikationen bestimmen & schützen gegen Fehlinterpretation

Verzeichnis: Liste mit Dateien & Unterverzeichnissen, "." Referenz auf sich selbst, ".." Ref. auf Elternverzeichnis

Verzeichnishierarchie: Gesamtheit aller Verzeichnisse

Wurzelverzeichnis: root, oberstes Verzeichnis, oft "/"

Arbeitsverzeichnis: hat jeder Prozess, Bezugspunkt für relative Pfade, externe Festlegung, Fn's: getcwd, (f)chdir

Absoluter Pfad beginnt mit /, ***Relativer Pfad*** beginnt nicht mit /, ***kanonischer Pfad***=absoluter Pfad ohne . & ..

NAME_MAX/PATH_MAX: max. Filename-/Pfad-Länge

Zugriffsrechte: jede Datei und jedes Verzeichnis haben je 3 Bits (0-7) Rechte für Owner, Gruppe & andere, 4=read, 2=write, 1=execute, addieren

errno: Makro oder globale Variable vom Typ int, wird von vielen aber nicht allen Funktionen gesetzt

POSIX-API: direkter/unformatierter Zugriff, nur für Binärdateien, Fehlerfall→Function-Return -1 & Code in errno, Zugriffsablauf: API → OS → Driver → File System → File

File-Descriptor: File-Repräsentation, gilt innnerhalb von Prozess, Index in eine Tabelle aller geöffneten Dateien im Prozess, Tabelleneintrag enthält Index in systemweite Tabelle aller geöffneten Dateien, Daten um physische Datei zu identifizieren, merkt sich aktuellen Offset, mit lseek kann Offset relativ zu Start, aktuellem Offset und Ende gesetzt werden, in jedem Prozess vorhanden: stdin_fileno=0, stdout_fileno=1, stderr_fileno=2

C-API: Stream-basiert, zeichen-orientiert, Abstraktion über Dateien, "\n"→OS-spezifische Zeilenenden, un/gepuffert (FD-Offset / eigener File-Position-Indicator), Funktionen heissen f*, greift auf POSIX API zu

3 Prozesse

Monoprogrammierung: nur OS & Programm laufen auf CPU, Kommunikation Programm→OS über C-Functions

Prozess: RAM-Abbild des Programms (text section), globale Variablen (data section), Heap- & Stack-Speicher, ist aktiv, Programm ist passiv & kann mehrfach in unabhängigen Prozessen ausgeführt werden

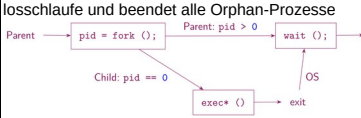
PCB: Process Control Block, vom OS, enthält Daten über einen Prozess: eigene ID, Parent ID, andere wichtige IDs, Speicher für Prozessor-Zustand, Scheduling-Infos, Daten für Sync & Com zwischen Prozessen, Filesystem relevante Infos, Security Infos

Interrupt: 1. Kontext (Register, Flags, Instruction Pointer, MMU-Config) in PCB sichern (context save), 2. Interrupt-Handler aufrufen (überschreibt Kontext evtl. komplett), 3. Kontext aus PCB wiederherstellen (context restore)

POSIX Prozess-Start: 1. Prozess erzeugen 2. Programm reinladen & in laufbereiten Zustand versetzen

Hierarchie: jeder Prozess ausser 1 hat genau einen Parent & beliebig viele Childs, Baum anzeigen: *pstree*

Zombie: Child zwischen seinem Ende & wait Aufruf von Parent (tot, aber noch nicht entfernt), ***Orphan***: Child ohne Parent, OS trägt Parent 1 ein, dieser hat wait-Endlosschlaufe und beendet alle Orphan-Prozesse



Weitere Funktionen: getpid, getppid, atexit (Fns für Ausführung bei exit registieren), sleep, waitpid

4 Programme und Bibliotheken

C-Toolchain: Headers/C-Datei → Präprozessor → Translation Unit → Compiler → Asm-Dateien → Assembler → Objekt-Dateien (auch Libs)→Linker → Executable

Loader: lädt Executables & dynamische Libs in RAM, *Linux*: exec* = syscall auf sys_execve, Datei suchen, x-Bit prüfen, öffnen, Args & Env-Vars zählen & kopieren, Request an alle reg. Binary Handler (z.B. ELF/a.out) übergeben, Handler versuchen nacheinander auszuführen

ELF: Executable & Linking Format, Linking View (Object-Files) & Execution View (Programme), dynamische Libs in beiden Views, *Bereiche*: **Header** (52B, Typ=Relozierbar|Ausführbar|Shared Object, 32/64bit, little/big endian, Maschine z.B. i386, Entrypoint: Programm-Startadresse, relative Adresse, Anzahl & Grösse von Program & Section Header Table), **Program/Segment Header Table** (32B Einträge: Segment-Typ, Flags, Offset/Grösse Datei, virtuelle Adresse, Grösse im Speicher), **Segments** (Loader-View, bestimmte werden geladen, weitere Segments für z.B. dynamisches Linken), **Section Header Table** (40/64B Einträge: Name: Referenz auf String Table, Typ, Flags, Offset/Grösse Datei, spezifische Infos je nach Section-Typ), **Sections** (andere Einteilung für gleiche Speicherbereiche wie Segments, jeweils gleichartige Daten, Compiler-View, Linker vermittelt, fügt Sektionen aller Object-Files zu Executable zusammen, verschmilzt gleichnamige Sections) ***Sektionstypen***: SHT_PROGBITS, symtab, strtab, rel(a), hash, dynamic, nobits (mehrere möglich), ***Sektionsattribute***: SHF_WRITE, alloc, execinstr, ***Spezialsektionen***: .bss, .data(1), .debug, .rodata(1), .text, .sym/strtab ***Symbol Table***: 16B Eintrag pro Symbol, je 4B: Name (strtab-Ref), Wert (z.B. Adresse), Grösse (z.B. Funktions-Länge), Info (Typ (Var, Array, Funktion, Sektion), Binding-Attribute (lokal, global, weak), Ref auf Section-Header, für die das Symbol gilt)

String Table: Dateibereich mit aneinandergereihten Strings, String-Referenz ist Offset ab Start der String Table, enthält Namen von Symbolen, keine String-Literale

Statische Libs: Object-File Archive, Linker behandelt sie wie mehrere Object-Files, einfache Verwendung & Implementierung, Programme müssen bei Änderungen in Libs neu erstellt werden, Funktionalität ist fix, keine Plugins

Dynamische Libs: auch Shared Objects genannt, linken erst zur Lade-/Laufzeit, mehr Aufwand für Coder/Compiler/Linker/OS, Exe hat nur Ref auf Lib, Lib-Update möglich ohne Exe-Änderung, Lib-Entwickler können direkt & unabhängig updaten, nur die Libs laden die gebraucht werden, schnellerer Start, Plugins möglich: Programm definiert bestimmte Fn-Signaturen, Plugin-Libs impl. diese, können flexibel anhand Namen aufgerufen werden

POSIX Shared Objects API: dlopen (dynamische Lib öffnen), dlsym (Adresse Symbol holen), dlclose, dlerror

Versionierung: alles unter /usr/lib, Linker-Name libx.so →Soft-Link (Linker folgt)→SO-Name libx.so.2→Soft-Link (Loader folgt)→Real-Name libx.so.2.1

readelf -d: Inhalt dynamic Section auslesen, dynamic Libs, die linked mit -l<x> wurden, haben NEEDED-Entry

ldd: führt Exe aus und gibt Loader-Trace mit allen (auch indirekt) benötigten Libs aus

ld-linux.so: wird vom OS indirekt aufgerufen, findet & lädt alle benötigten shared objects (rekursive), auch CLI

Implementierung: müssen verschiebbar sein, Loader / Dynamic Linker macht Linker-Aufgabe, dynamische Libs liegen in shared memory (dynamic lib page frame), Prozesse haben virtuelle page (zeigt auf dlpf), so wird der RAM nur 1x gebraucht, pages haben andere Adressen

Position-Independent Code: normalerweise hängt Code von absoluten Adressen ab, das funktioniert aber nicht bei verschiebbarem Code, daher werden Adressen relativ zum Instruction Pointer verwendet, relative Calls: x86_32/64, relatives mov nur 64

GOT: Global Offset Table, 1x pro shared obj, Eintrag pro Symbol, das von anderen shared obj benötigt wird, relative Adressen, Loader füllt Adressen zur Laufzeit ein

PLT: Procedure Linkage Table, für Lazy Binding, Funktionen erst binden wenn benötigt, Eintrag pro Funktion, enthält Sprungbefehl an Adresse in GOT-Eintrag, zeigt auf Proxy-Funktion, welche Link zur richtigen Funktion sucht und dann den GOT-Eintrag überschreibt

5 Threads

Nachteile Prozess: Datenaustausch benötigt OS-Eingriff, aufwändiges Umschalten, Isolation/Security oft unnötig, Realisierung paralleler Abläufe aufwändig, Overhead der Prozesserzeugung zu gross für kürzere Teilaktivitäten, gemeinsame Ressourcennutzung erschwert

Thread: parallel ablaufende Aktivitäten innerhalb Prozess, haben auf alle Ressourcen (Code, globale Variablen, Heap, geöffnete Files, MMU-Daten) gleichermassen Zugriff, eigener Kontext & Stack, Ablage in Thread-Control-Block (Linux: PCB-Kopie mit eigenem Kontext)

Amdahls Regel: *T*=Serial Ausführungszeit, *n*=Anz. CPU

$$f \le \frac{T}{T_s + \frac{T-T_s}{n}} = \frac{T}{s \cdot T + \frac{T-s \cdot T}{n}} = \frac{T}{s \cdot T + \frac{1-s}{n} \cdot T}$$

$$= \frac{1}{s + \frac{1-s}{n}}$$

T'=Ausführungszeit max. parallelisiert
T_s=*s*T*=Ausführungszeit serieller Anteil
T-T_s=Ausführungszeit paralleler Anteil
f=obere Schranke für max. Speedup, *f(s,n)→f(0,n)=n*

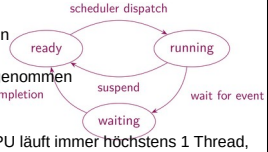
Lebensdauer: mystart returned, "void pthread_exit (void *return_value)" wird aufgerufen, anderer Thread ruft "int pthread_cancel (pthread_t thread_id)" auf

POSIX Thread API: pthread_create, pthread_exit, pthread_cancel, pthread_detach (entfernt Thread-Speicher, falls beendet), pthread_join, pthread_self (return id)

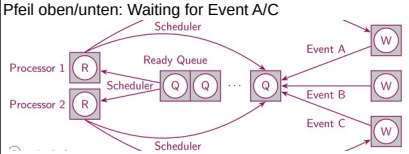
Thread Local Storage: errno kann nicht als globale Var impl. werden, weil mehrere Threads sie gegenseitig überschreiben könnten, TLS stellt globale Vars per Thread zur Verfügung, *bevor Threads erzeugt werden*: Key anlegen, der TLS-Var identifiziert & Key in globaler Var speichern (pthread_key_create), *im Thread*: Key aus globaler Var auslesen & Value auslesen / schreiben anhand des Keys (pthread_(s|g)etspecific)

6 Scheduling

Thread-Zustände:
waiting z.B. nach join
Übergänge werden immer vom OS vorgenommen



Scheduling: pro CPU läuft immer höchstens 1 Thread, Scheduler entscheidet, welcher als nächster laufen soll, Pfeil oben/unten: Waiting for Event A/C



Powerdown: wenn kein Thread running ist, geht die CPU in den Standby, das spart erheblich Energie

Thread-Arten: I/O-lastig/bound & Prozessor-lastig/bound

Kooperativ: Thread entscheidet CPU-Abgabe selbst

Präemptiv: Scheduler entscheidet, wann dem Thread die CPU entzogen wird, Gründe für nächsten Thread: Thread wartet auf I/O / Ressource (blockiert), verzichtet freiwillig, System-Timer-Interrupt, anderer Thread wird ready und bevorzugt, neuer Prozess wird bevorzugt

Ausführung: parallel (alle n Threads gleichzeitig auf n CPUs, ideal), quasiparallel (n Threads abwechselnd auf <n CPUs), nebenläufig (parallel oder quasiparallel)

Prozessor-Burst: Intervall, in dem ein Thread den Prozessor voll belegt, also vom Eintritt in running zu waiting

I/O-Burst: Intervall, in dem ein Thread den Prozessor nicht benötigt, also vom Eintritt in waiting zu running

Scheduler-Anforderungen: *Durchlaufzeit* (turnaround time, Zeit Thread-Start bis Ende), *Antwortzeit* (response time, Zeit Request-Empfang bis Antwort, slice kl.), *Wartezeit* (waiting time, Zeit in der ready-Queue), *Durchsatz* (throughput, Anz. Threads pro Intervall, slice gr.), *Prozessor-Verwendung* (processor utilization, % CPU-Verwendung ggü. Nichtverwendung), optimal in allem geht nicht

First Come First Serve: k, Scheduling in Reihenfolge, wie in Ready-Queue added (FIFO)

Shortest Job First: k or p, optimal, wählt Thread mit kürzestem nächsten Prozessor-Burst aus

Round Robin: p, Zeitscheibe (time slice) von 10-100ms, Thread läuft so lange & wird bei Ready-Queue appended

Prio-basiert: p, Threads haben Prio und werden danach ausgewählt, gleiche Prio→FCFS, SJF: Prio=CPU-Burst

Starvation: Threads mit niedriger Prio laufen nie, weil immer Threads mit grösserer Prio in der Queue sind, Abhilfe z.B. mit Aging (Prio wird regelmässig erhöht)

Multi Level: p, Aufteilung in Level nach Kriterien (z.B. Prio/Prozesstyp/Back-/Foreground), Queue mit eigenem Verfahren pro Level, Queues gegeneinander priorisiert

mit Feedback: Queue pro Prio & Zeitscheiben, Thread erschöpft Zeitscheibe→tiefere Prio & landet damit in schlechterer Queue, Zeitscheiben werden grösser je tiefer die Prio, kurze CPU-Bursts bevorzugt

Niceness: hat jeder Thread unter Linux, von -20 (höchste Prio) bis 19 (tiefste Prio), Shell "nice -n 0 util"
Thread-Attribute enthalten *struct sched_param* mit Member *sched_priority*, kann vom Thread abgefragt werden

7 Mutexe & Semaphore

Producer-Consumer-Problem: Producer erzeugt Items, Consumer verarbeitet sie, sind nicht gleich schnell, Übermittlung mit begrenzt grossem Ring-Puffer, warten aufeinander mit Busy-Wait, 1 Abstand zwischen Elementen

Race Condition: Ergebnisse hängen von Ausführungsreihenfolge einzelner Instruktionen ab, z.B. sind "var++" 3 Instruktionen, da kann es Interrupts & Thread-Wechsel geben, Register/Kontext werden zwar gesichert, RAM-Bereiche jedoch nicht, nebenläufige Threads können Bereiche gegenseitig überschreiben, ***Critical Section***: Code-Bereich, in dem Daten mit anderen Threads geteilt werden, Protokoll benötigt, anhand dessen Threads den Zugriff zu ihren Critical Sections synchronisieren können

Anforderungen Sync-Mechanismen: *Mutual Exclusion* (nur 1 Thread in CS), *Fortschritt* (Entscheidung in endlicher Zeit, wer in CS darf), *begrenztes Warten* (Thread darf max. *n*-mal übergangen werden, wenn er in CS will)

Atomare Instruktion: kann von CPU unterbrechungsfrei ausgeführt werden, teilweise sind einzelne Asm-Instruktionen nicht atomar: "mov [2009], rax" (2009/8 unmöglich)

Sync-Mechanismen: brauchen HW-Support, weil Sequenzen funktional äquivalent umgeordnet werden können, Interrupts nicht abschalten während CS, Test&Set / Compare&Swap werden garantiert auch über mehrere Prozessoren nie gleichzeitig ausgeführt, Basis von Locks

```
int test_and_set (int *target) { int value = *target;
*target = 1; return value; }
while (test_and_set (&lock) == 1) {} /* CS */ lock = 0;
int compare_and_swap (int *a, int expected, int new_a) {
int value = *a; if (value == expected) { *a = new_a; }
return value; }
while (c_a_s (&lock, 0, 1) == 1) {} /* CS */ lock = 0;
```

Semaphor: enhält Zähler $z \geq 0$, *post* erhöht z um 1, *wait* verringert z um 1 und fährt fort oder versetzt Thread in *waiting*, bis ein anderer Thread *post* ausführt, synced
Weitere Methoden: sem_(trywait|timedwait|destroy)
Mutex: binärer Semaphor, also *z=0|1*, *acquire/lock*: *z=1* & fortfahren wenn *z=0*, Thread blockieren bis *z=0* wenn *z=1*, *release/unlock*: *z=0* setzen, dazwischen CS

```
sem_t free, used; sem_init(&free, 0, n);
sem_init(&used, 0, 0);
/* producer */ while (1) { sem_wait(&free);
produce_item (&buffer[w], ...); sem_post (&used);
w = (w+1) % BUFFER_SIZE; }
/* consumer */ while (1) { sem_wait(&used);
consume (&buffer[r]); sem_post (&free);
r = (r+1) % BUFFER_SIZE; }
```

Priority Inversion: Problem bei Threads mit unterschiedlichen Prios, Low-Prio Thread *L* hat Ressource, High-Prio Thread *H* wartet auf R, Mid-Prio Thread *M* erhält Prozessor, weil *H* waiting ist, wenn *M* fertig ist, läuft wieder *L* bis fertig und erst dann *H*, die Prios von *L*&*H* sind vertauscht
Priority Inheritance: Lösung für Prio Inv, *L* erbt Prio von *H* (wenn *H* die locked Ressource anfragt) & kann somit nicht mehr von *M* unterbrochen werden, sobald *L* die R. nicht mehr benötigt, erhalten *L*&*H* wieder die alte Prio

8 Signal, Pipes & Sockets

Signal: unterbricht Prozess von aussen, wie Interrupt, Signal-Handler Funktion wird ausgewählt & ausgeführt, Prozess fährt fort, Quellen: Hardware (z.B. ungültige Instruktion), OS (z.B. Segfault), andere Prozesse (z.B. kill), 3 Default Handler für jedes Signal bei Prozessbeginn (Ignore, Terminate, Abnormal-Terminate), alle können überschrieben werden ausser sigkill & sigstop
Wichtige Signale: **OS**: *fpe* (Fehler in artithmentischer Operation), *ill* (ungültige Instruktion), *segv* (Segfault), *sys* (ungültiger syscall), **Abbruch**: *term* (normal beenden), *int* (Ctrl+C), *quit* (int aber abnormal), *abrt* (quit aber von Prozess selber), *kill* (letzte Zuflucht), **Stop/Continue**: *tstp* (Ctrl+Z, *stopped* Zustand), *stop* (force tstp), *cont*

```
kill -l && kill 123 && kill -KILL 123 # List sigs, term, kill
```

Pipe: abstrakte Datei (mit Eintrag in FDT) im RAM mit den 2 File-Descriptors read end und write end, Kommunikation zwischen Prozessen, geschriebene Daten können 1x in FIFO-Manier gelesen werden, kein lseek, read blockiert bis Daten kommen, bei write-close EOF von read, bei read-close SIGPIPE an Prozess mit write end

```
int fd [2]; pipe (fd); if (fork () == 0) { close (fd [1]); char
buffer [BSIZE]; int n = read (fd[0], buffer, BSIZE); }
else { close (fd [0]); char * text = "Hallo OST";
write (fd [1], text, strlen(text) + 1); }
dup2 (fd [0/1], 0/1); // read/write end als stdin/stdout
int mkfifo(const char *path, mode_t mode); /* erzeugt
named Pipe, Pfad & Permissions, Prozess-unabh. */
```

Socket: Endpunkt auf Maschine, IP & Port, Kommunikation zwischen Sockets, **Client**: *connect* (bindet Socket an neue, unbenutzte lokale Adresse, wenn noch nicht gebunden), *send/write* (optional, mehrmals möglich), *recv/read* (optional, mehrmals möglich), *close*, **Server**: *bind* (bindet Socket an angegebene, unbenutzte lokale Adresse, wenn noch nicht gebunden), *listen* (markiert Socket als empfangsbereit, stellt Connection-Queue bereit), *accept* (erzeugt neuen Socket, der kann keine neuen Verbindungen annehmen, bindet ihn an neue lokale Adresse), *recv/read, send/write, close*

9 Message Passing & Shared Memory

Verfahren Kommunikation zwischen Prozessen: Pipes, Sockets, Message Passing, Shared Memory, Disk-Files, Memory-mapped Files
Message Passing: Mechanismus mit 2 Operationen: *send* kopiert Msg aus Proc & *receive* kopiert Msg in Proc, bidirektional, Vorteile: leichte Erweiterung auf verteiltes System, schneller bei Multi-Core, Nachteile: grosse SW-Teile müssen neu impl. werden
Feste / variable Nachrichtengrösse: komplizierte Verwendung / komplizierte Implementierung
Direkte / indirekte Kommunikation: Sender adressiert an Empfänger, symmetrisch: Empfänger kennt Sender, asymmetrisch: Empfänger erhält Sender-ID in Out-Param / braucht Mailboxen, Ports oder Queues, die beide kennen, Lebenszyklus verwaltet Prozess oder OS?
Synchron / Asynchron: blockierendes / nicht-blockierendes Senden & Empfangen, alle Kombis möglich
Rendezvous: send & receive blockierend, triviale Producer-Consumer Implementierung, impliziter Sync
Buffering: keines, beschränkt, unbeschränkt, ***Message Priorität***: mit/ohne, bestimmt was zuerst received wird
POSIX: indirekt mit OS-Queues, variable Nachrichtengrösse mit Maximum pro Queue, synchron & asynchron möglich, 32 – 32K Prioritäten, Methoden: mq_open, mq_close, mq_unlink, mq_send, mq_receive
Shared Memory: Page Frames vom RAM werden zwei oder mehr Prozessen zugänglich gemacht, Prozesse haben eigene Pages mit eigenen Adressbereichen damit verknüpft & können beliebig zugreifen, Pointer müssen relative Offsets sein, OS benötigt spezielles Objekt S, das Infos über den Shared Memory verwaltet und Objekt M_i pro Prozess P_i, das die Mappings speichert, Vorteile: schnell realisiert, schneller bei Single-Core, Nachteile: schwer wartbare Programme, nebenläufige statt parallel Resultate wegen impliziten Abhängigkeiten, weniger Modularisierung & Isolation, ***POSIX API***: shm_open (Shared Memory mit systemwide /name öffnen), ftruncate (Grösse setzen), close, shm_unlink (SM löschen, wenn ihn alle Prozesse closed haben), mmap (mapped SM & returned 1. Adresse), munmap

10 Unicode

ASCII: 7 Bit, 128 definierte Zeichen (Zahlen, Klein- / Grossbuchstaben, Interpunktionszeichen), viele unabhängige Erweiterungen auf 8 Bit, heissen Codepages & definieren Zeichen von 80_h bis FF_h, muss bekannt sein zur korrekten Textdarstellung, sonst unleserlich
Unicode: unique Code für jedes vorhandene Zeichen definieren, auch ausgestorbene Sprachen & Emojis, 149'813 von 17*64K−2K=1' 112' 064 verwendet, WIP
CP: Code Point, Nummer eines Zeichens
CU: Code Unit, Einheit um Zeichen encodet darzustellen
P_i: i-tes Bit CP, ***U_i***: i-tes Bit CU, ***B_i***: i-tes Byte CP
UTF-32: jede CU hat 32 Bit, jeder CP braucht 1 CU, obere 11 Bits können zweckentfremdet werden, Achtung Endianness, U20..0=P20..0, U0 & P0 ganz rechts bei BE
UTF-16: jede CU hat 16 Bit, CP braucht 1 - 2 CUs, Achtung Endianness, *2 Byte (CP in [0, FFFF] ohne [D800, DFFF]→P20..16=0)*: U15..0=P15..0, *4 Byte (CP in [1'0000, 10'FFFF])*: U9..0=P9..0, U15..10=110111, U25..16=P19..10, U31..26=110110, P20 verschwindet
UTF-8: jede CU hat 8 Bit, CP braucht 1 - 4 CUs, *1 Byte (CP in [0, 7F])*: U6..0=P6..0, *2 Byte (CP in [80, 7FF])*: U5..0=P5..0, U7..6=10, U12..8=P10..6, U15..13=110, *3 Byte (CP in [800, FFFF])*: U5..0=P5..0, U7..6=10, U13..8 =P11..6, U15..14=10, U19..16=P15..12, U23..20=1110, *4 Byte (CP in [1'0000, 10'FFFF])*: U5..0=P5..0, U7..6=10, U13..8=P11..6, U15..14=10, U21..16=P17..12, U23..22=10, U26..24=P20..18, U31..27=11110

11 Ext2

Partition: Datenträger-Teil, wird wie einer behandelt
Volume: Datenträger oder Partition davon
Sektor: kleinste logische Volume-Untereinheit, Datentransfer als Sektoren, Grösse HW-definiert, enthält Header, Daten & Error Correction Codes
Format: logisches Datenträger-Struktur-Layout, vom FS
Block: mehrere aufeinanderfolgende Sektoren, meist 4KB, Volume ist in Blöcke aufgeteilt & Speicher wird nur in Form von Blöcken alloziert, 1. Block-Sektor = Block-Nr. * "Anz. Sektoren je Block", logische Block-Nr: von Anfang der Datei aus gesehen, wenn Datei ununterbrochene Abfolge von Blöcken wäre (App-Sicht), physische Block-Nr: tatsächliche Block-Nr auf dem Volume
Inode: Index Node, Dateibeschreibung mit allen Metadaten (Grösse, Anz. Blöcke, Create- / Access- / Modification- / Delete-Time, UID, GID, Flags, Permissions) ausser Name & Pfad, fixe Grösse je Volume, ***i_block Array***: 15 4 Byte Einträge: *12x Block-Nr*, *einfach* (Blockgrösse 1KB→ref. Blöcke 12-267, 4KB→ref. Blöcke 12-1035) / *doppelt* (ref. indirekte Blöcke, 1KB→256^2 Blöcke, 4KB→1024^2 Blöcke) / *dreifach* indirekter Block (ref. doppelt indirekte Blöcke, 1KB→256^3 Blöcke, 4KB→1024^3 Blöcke), Max. Filesize: 4TB (i_block), aber 2TB (4 Byte Counter)
Blockgruppe: Blockgrösse 4KB→bis 32K Blöcke in Gruppe (x8), Gruppengrösse gleich für alle Gruppen

Superblock: enthält alle Metadaten über das Volume: Inodes frei/gesamt, Blöcke frei/gesamt/reserviert, Bytes pro Block/Inode, Blöcke/Inodes pro Gruppe, Mount-/Schreib-/Last-Check-Zeitpunkt, Statusbytes, 1. Inode für Apps, Feature Flags, startet immer an Byte 1024 (Bootdaten vorher), ***Sparse***: aktivieren mit Flag, Superblock-Kopien nur in Blockgruppe 0, 1 & Potenzen von 3/5/7
Group Descriptor Table: 32 Byte Eintrag pro Blockgruppe (Block-Nr Block/Inode Usage Bitmap/1. Block Inode-Tabelle, Anz. freie Blöcke/Inodes, Anz. Verzeichnisse)
Inode lokalisieren: Zählstart 1, Blockgruppe=(Inode - 1) / (% für Index darin) "Anz. Inodes pro Gruppe"
Inode erzeugen: Verzeichnisse in Gruppen mit vielen freien Blöcken & Inodes, Dateien in Gruppe mit Verzeichnis, 1. freies Inode mit Usage Bitmap ermitteln
File-Holes: wenn Block nur Nullen enthält, wird der Eintrag dafür auf 0 gesetzt, der Block wird nicht alloziert
Verzeichnis: Enthält Refs auf Dateieinträge, . & .. automatisch angelegt, ***Dateieintrag***: 4 Byte Inode, 2 Byte Eintrag-Länge, 1 Byte Dateinamen-Länge, 1 Byte Dateityp (Datei|Dir|Soft Link), 0-255 Byte Dateiname (ASCII)
Hard Link: Inode wird von versch. Dateieinträgen referenziert, ***Soft/Symbolischer Link***: wie eine Datei

12 Ext4

Unterschiede zu ext2: Inodes haben 256 statt 128 Byte, Gruppendeskriptoren haben 64 statt 32 Byte, Blockgrösse bis 64KB, Inodes verwalten Blöcke mit Extent Trees, Journaling für Konsistenz auch bei Stromausfällen
Extents: grosse Dateien werden in der Praxis oft physisch aufeinanderfolgend geschrieben, in ext2 braucht es trotzdem sehr viel Platz, um alle Blöcke eines grossen Files im Inode zu referenzieren, effizienter sind Extents, welche ein Intervall konsekutiver Blöcke beschreiben: ab logischer/physischer Blocknummer (4 Byte / 6 Byte), Anzahl Blöcke (2B signed, positiv=Blöcke initialisiert, negativ=Blöcke voralloziert), Datei kann >1 Extent umfassen, Inode hat Platz für Extent Tree Header + 4 Extents
Extent Tree Header: für mehr als 4 Extents pro Inode braucht es einen zusätzlichen Block, daher wird dieser Header gebraucht, 2 Byte grosse Felder: Magic Number, Anz. direkt dem Header folgende Einträge, max. Anz. Einträge die auf den Header folgen können, Tiefe (0→Einträge = Extents, 1 bis 5→Einträge = Index Nodes), am Schluss 4 Byte reserviert
Index Node: spezifiziert Block, der Extents enthält, besteht aus: Extent Tree Header & max. 340 Extents bei 4KB Blocksize (4 Byte kleinste logische Block-Nr aller Kind-Extents, 6 Byte physische Block-Nr des Blocks, auf den der Index Node verweist, 2 Byte unused)
Index Node Blöcke: braucht es, wenn man mehr als 4 * 340 Extents benötigt, Tiefe Inode = 2 (max. 5), Tiefe Index Node = 1, statt Extents sind Index Nodes im Block, kleinste logische Block-Nr wird bis ganz oben propagiert
Extent Tree: Baum mit Index Blocks als Root & innere Knoten, Extents als Leafs, Index Blocks haben eigenen Extent Tree Header und haben Refs auf Index Nodes oder Extents je nach Tiefe, welche immer um 1 kleiner als der übergeordnete Knoten sind, i_block[0...14] kann als sehr kleiner Index Block aufgefasst werden
Ablauf Dateierweiterung: neue Blöcke allozieren, Inode anpassen um Blöcke zu referenzieren, Block Usage Bitmap anpassen, Counter freie/benutzte Blöcke anpassen, Daten in Datei schreiben, Unterbrüche können zu Inkonsistenzen führen, dann müssen alle Metadaten auf Inkonsistenzen überprüft werden, was sehr lange dauert
Journaling: verringert Dauer von Inkonsistenz-Prüfung erheblich, Journal ist reservierte Datei, in die Daten relativ schnell geschrieben werden können, typischerweise Inode 8 & 128MB, besteht aus wenigen grossen, am besten einem Extents, darin befinden sich ***Transaktionen***: Folge von Einzelschritten, die das FS atomar vornehmen soll, Daten zuerst als Transaktion ins Journal schreiben (*Journaling*), dann an endgültige Position schreiben (*Committing*), dann Daten aus dem Journal entfernen, bei einem Unterbruch kann die Prüfung auf die Metadaten beschränkt werden, welche gemäss Journal betroffen sein könnten, *Journal Replay* führt die Transaktionen nochmals aus, ***Journal Modi***: *full* (Metadaten & Datei-Inhalte, maximale Sicherheit, grosse Geschwindigkeitseinbussen), *ordered* (Metadaten, Datei-Inhalt vor commit schreiben, etwas langsamer als write-back), *writeback* (Metadaten, commit & Datei-Inhalt schreiben in bel. Reihenfolge, schnell, Datenmüll mögl.)

13 X Window System

Programmgesteuerte Interaktion: Programm fordert Input via OS vom User an und erhält ihn vom User via OS
Ereignisgesteuerte Interaktion: User gibt Input via OS an Programm, welches via OS mit Output antwortet
Fenster: rechteckiger Bildschirmbereich, kann beliebig viele weitere Fenster enthalten, Baumstruktur, z.B. Hauptfenster, Dialogbox, Scrollbar, Textlabel, Button
Maus: physisches Gerät, das 2D-Bewegungen in Daten übersetzt (x-/y-Achse), Mauszeiger ist kleine Rastergrafik mit Hotspot, welcher exakte Zeiger-Position definiert, Tasten lösen Events aus, OS muss Fenster an Position ermitteln und Event an Fenster-Owner senden, asynchron, OS wartet nich auf App, um nicht zu freezen
Window Manager: verwaltet sichtbare Fenster, platziert Client-Fenster ansehnlich, akzeptiert / modifiziert / ignoriert Anzeige-Hinweise der Clients, Umrandung, Knöpfe
Desktop Manager: Desktop-Hilfsmittel wie Taskleiste
Fensterhierarchie: *Root Window*: bei X Start erzeugt, bedeckt ganzen Screen, Kinder davon sind *Top-Level Windows* der Apps, Kinder davon sind z.B. Buttons, können Eltern überlappen, aber nur innerhalb der Eltern angezeigt werden & Input empfangen
Fensterklassen: InputOutput / InputOnly
X Window System: Fensterdarstellungs-Grundfunktion, unabhängig von Window/Desktop Manager, Gestaltung der Bedienoberfläche & -philosophie bleiben frei
Display: 1..n Screens, Tastatur, Zeigegerät, *X Server* auf gleichem Gerät wie Display, *X protocol* lokal oder remote zum *X Client* (besteht aus *Xlib* & App)
X Toolkits: direkte Xlib Programmierung ist aufwändig, Toolkits vereinfachen & stellen Standardbedienelemente (widgets) zur Verfügung, z.B. GTK+ (gnome) / Qt (KDE)
Nachrichten zwischen X Client & Server: Request→ (Clt buffered, möglichst wenige Übertragungen an Server, z.B. ändere Farbe), Reply←, Event← (Clt & Srv buffered, Clt muss vorher festlegen, welche er empfangen will, 33 Typen, z.B. Mausklick, Systemereignis wie Fenstergrösse geändert), Error←
Ressourcen: server-seitig gehalten, um NW-Traffic zu reduzieren, haben IDs, optionale Pufferung verdeckter Fensterinhalte, z.B. Window, Pixmap (Rastergrafik, z.B. Icon), Colormap, Font, Graphics Context (GC, Grafikeigensch. wie Füllmuster, Liniendicke, Farbe)
Grafikfunktionen: Bilddarstellung mit Rastergrafik & Farbtabelle, genau ein Bit pro Bildpunkt bei S/W, mehrere Bits pro Bildpunkt bei Farben, Index in Farbtabelle, Tabelle reduziert Bits pro Farbe von n (absolut darstellbare Farben) auf m (gleichzeitig darstellbare Farben) Bits
Grafikgrundfunktionen: geometrische Figuren/Strings/Texte zeichnen in Fenstern/Pixmaps, brauchen GC
Atom: ID eines Strings, der für Meta-Zwecke benötigt wird, z.B. WM_DELETE_MESSAGE = 5
Property: mit Fenster assoziiert, WM liest/setzt sie, generischer Komm-Mechanismus zwischen App & WM, über Atom identifiziert, spezifische Daten gehören dazu
Fenster schliessen: Window Manager sendet Client-Message Event mit Property Atom 5 an Applikation, diese verarbeitet dann das Event

14 Meltdown

Moderne HW-/OS-Tricks: *Caches*, *Out-of-Order Execution* (O3E, Instruktions-Reihenfolge kann verändert werden, wenn Endergebnis nicht beeinträchtigt wird), *spekulative Ausführung* (O3E auch wenn Instruktion später gar nicht ausgeführt wird), *Mapping des gesamten physischen Speichers in jeden Adressraum* (man spart sich den teuren Kontext-Wechsel, wenn ein Prozess einen OS-Service anfordert, OS muss auf alle Prozesse zugreifen können, Page-Table begrenzt Zugriffsbereiche)

```
char dummy_array [256]; char * sec_addr=0x12345678;
char sec_data = *sec_addr; // Exception
char dummy = dummy_array [sec_data]; // Spekulativ
```

Meltdown: spekulative Ausführung ermöglicht Auslesen des Bytes m_a an Adresse a, konzeptionell sicher da Prozess den Wert nie sieht, m_a wird als Teil des Tags (Form f_a) im Cache zwischengespeichert, konzeptionell sicher, da MMU entscheidet, ob Prozess darauf zugreifen darf, jedoch ist Side-Channel Attacke möglich, bei binären Vergleichen mit f_a verrät die Zugriffszeit, ob es cached ist
Massnahmen: Kernel Address Isolation to have Side-channels efficiently removed (KAISER), verschiedene Page-Tables für Kernel & User Mode
Spectre: Ziel wie Meltdown, nutzt Branch Predicition mit spekulativer Ausführung (Prozessoren lernen, welche bedingten Sprünge erfolgen oder nicht), Angreifer kann Branch Predictor für anderen Prozess trainieren