Betriebssysteme 2 | BSys2

7usammenfassung

int *px = &x; // &x = Adresse des ints, * = Pointer-Bezeichne

4096	2048	1024	512	256	128	64	32	16	8	4	2	1
2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^3	2^2	2^1	20
10 00 _h	800 _h	4 00 _h	200 _h	1 00 _h	80 _h	40 _h	20 _h	10 _h	8 _h	4 _h	2 _h	1 _h
1'048'5	76	65′536		4'096		256			16		1	
16^{5}		16^{4}		16^{3}		16 ²			16 ¹		16^{0}	
10 00 0	n.	01 00 0	10.	00 10	00-	00.0	01 00 _h	- 1	00 00 10	۸.	00 00	01.

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

1 RETRIERSSYSTEM APT

Aufgaben: Abstraktion, Portabilität, Ressourcenmanagement & Isolation der Anwendungen, Benutzerverwaltung und Sicherheit.

Privilege Levels: Kernel-Mode (darf alles ausführen, Ring 0), User-Mode (darf nur beschränkte Mei struktionen ausführen, Ring 3)

Kornels: Microkernel (nur kritische Teile laufen im Kernel-Made) Manalithisch (meiste OS weniger Werhrel um niger Schutz), Unikernel (Kernel ist nur ein Programm

syscall veranlasst den Prozessor, in den Kernel Mode zu schalten. Jede OS-Kernel-Funktion hat einen Code, der einem Register übergeben werden muss. Jezit hat den Code 601

ABI: Application Binary Interface, Abstrakte Schnittstelle mit platformunabhängigen Aspekten API: Application Programming Interface, Konkrete Schnittstellen, Calling Convention, Abbildung von Datenstrukturen. Linux-Kernels sind API-, aber nicht ABI-kompatibel. (C-Wrapper-Funktie POSIX: Portable Operating System Interface, Sammlung von IEEE Standards, welche die Kombatibilität zwischen OS gewährleistet. Windows ist nicht POSIX-konform.

1.1. PROGRAMMARGUMENTE

clang -c abc.c -o abc.o. Die Shell teilt Programmargumente in Strings auf (Trennung durch Leerzeichen, sonst Quotes). Calling Convention: OS schreibt Argumente als null-terminierte Strings in den Speicherbereich des Programms. Zusätzlich legt das OS ein Array angv an, dessen Elemente jeweils auf das erste Zeichen eines Arguments zeigen. Die Art und Weise, wie das gehandhabt wird, ist die Calling Convention. Werden explizit angegeben, nützlich für Informationen, die bei jedem Aufruf anders sind

int main(int argc, char ** argv) f ... } // argv[8] = program path

1.2. UMGEBUNGSVARIABLEN

Strings, die mindestens ein Key=Value enthalten OPTER=1, PATH=/home/ost/bin. Der Key muss einzigartig sein. Unter POSIX verwaltet das OS die Umgebungsvariablen innerhalb jedes laufenden Prozesses. Werden initial festgelegt. Das OS legt die Variablen als ein null-terminiertes Array von Pointern auf null-terminierte Strings ab. Unter Czeigt die Variable extern ch Sollte nur über untenstehende Funktionen manipuliert werden. Werden implizit bereiteestellt. nützlich für Informationen, die bei jedem Aufruf gleich sind.

- Abfragen einer Umgebungsvariable: char *value = qetenv("PATH");
- Setzen einer Umgebungsvariable: int ret = setenv("HOME", "/usr/home", 1); Entfernen einer Umgebungsvariable: int ret = unsetenv("HOME");
- Hinzufügen einer Umgebungsvariable : int ret = putenv("HOME=/usr/home");

Grössere Konfigurationsinformationen sollten über Dateien übermittelt werden

2. DATEISYSTEM API

Applikationen dürfen nie annehmen, dass Daten gültig sind.

Arbeitsverzeichnis: Bezugspunkt für relative Pfade, ieder Prozess hat eines (getowd(), chdir(); nim String, fchdir(): nimmt File Deskriptor).

- Pfade: Absolut (beginnt mit /), Relativ (beginnt nicht mit /), Kanonisch (Absolut, ohne «.» und «..», real path())
- NAME_MAX: Maximale Länge eines Dateinamens (exklusive terminierender Null)
- PATH_MAX: Maximale Länge eines Pfads (inklusive terminierender Null) (beinholtet Wert von NAME_MAX)
- _POSIX_NAME_MAX: Minimaler Wert von NAME_MAX nach POSIX (14) - POSTX PATH MAX: Minimaler Wert von PATH MAX nach POSIX (256)

main (int argc, char ** argv) { char *wd = malloc(PATH_MAX); ge _MAX); printf("Current WD is %s", wd); free(wd); return 0; }

Zugriffsrechte: Je 3 Permission-Bits für Owner, Gruppe und andere Benutzer. Bits sind: read, write, everyte n=4 w=2 v=1 Reigniel: 8748 order nwy n== --- Owner hat alle Reg andere haben keine Rechte. POSIX: S_IRWXU = 0700, S_IWUSR = 0200, S_IRGRP = 0040, S_IXOTH = 0001. Werden mit I verknüpft.

POSIX-API: für direkten Zugriff, alle Dateien sind rohe Binärdaten. C-API: für direkten Zugriff auf Streams. POSIX FILE API: für direkten, unformatierten Zugriff auf Inhalt der Datei. Nur für Binärdaten verwenden. errno: Makro oder globale Variable vom typ int. Sollte direkt nach Auftreten eines Fehlers aufgerufen werden.

if (chdir("docs") < 0) { if (errno = EACCESS) { printf("Error: Denied"); }}

strenner eint die Adresse eines Strings zurück, der den Fehlercode code textuell beschreibt. perror schreibt text gefolgt von einem Doppelpunkt und vom Ergebnis von strerror (errno) auf

2.1. FILE-DESCRIPTOR (FD)

Files werden in der POSIX-API über FD's repräsentiert. Gilt nur innerhalb eines Prozesses. Returnt Index in Tabelle aller geöffneter Dateien im Prozess → Enthält Index in systemweite Tabelle → Enthält Daten zur Identifikation der Datei. STOTN ETLEND = 8: standard innut. STOOLT ETLEND = 1: standard output. STDERR FILENO = 2: standard error

int open (char *path, int flags, ...): öffnet eine Datei. Erzeugt FD auf Datei an path. flags gibt an, wie die Datei geöffnet werden soll, O RDONLY; nur lesen, O RDINR; lesen und schreiben. O_CREAT: Erzeuge Datei, wenn sie nicht existiert, O_APPEND: Setze Offset ans Ende der Datei vor jedem Schreibzugriff, 0_TRUNC: Setze Länge der Datei auf 0

int close (int fd): schliesst Datei bzw. dealloziert den FD. Kann dann wieder für andere Datei en verwendet werden. Wenn FD's nicht geschlossen werden, kann das FD-Limit erreicht werden, dann können keine weiteren Dateien mehr geöffnet werden. Wenn mehrere FDs die gleiche Datei öffnen, können sie sich gegenseitig Daten überschreiben.

int fd = open("myfile.dat", 0_RDONLY);
if (fd < 0) f /* error handling */ } /* read data: */ close(fd);</pre>

ssize t read(int fd. void * buffer. size t n):

kopiert die nächsten n Bytes am aktuellen Offset von fd in den Buffer

ssize t write(int fd. void * buffer. size t n): kopiert die nächsten n Byte vom buffer an den aktuellen Offset von fd

BSys2 | FS24 | Nina Grässli & Jannis Tschan

char buf[N] char spath[PATH_MAX]; // source path char dpath[PATH MAX]: // destination path

// ... gets paths from somewhere
int src = open(spath, O_RDONLY);
int dst = open(dpath, O_WRONLY | O_CREAT, S_IRWXU);

ssize t read bytes = read(src. buf. N): write(dst, buf, read_bytes); //if file gets closed early, use return value of

off_t lseek(int fd, off_t offset, int origin): Springen in einer Datei. Verschiebt den Offset und gibt den neuen Offset zurück, SEEK SET; Beginn der Datei, SEEK CUR; Aktueller Offset, SEEK_END: Ende der Datei. Lseek(fd, 0, SEEK_CUR) gibt aktuellen Offset zurück, Lseek(fd, 0, SEEK_END): gibt die Grösse der Datei zurück.

ssize t pread/pwrite(int fd. void * buffer, size t n. off t offset); Lesen und Schreiben ohne Offsetänderung. Wie nead bzw. write. Statt des Offsets von fd wird

der zusätzliche Parameter offset verwendet

2.1.1. Unterschiede Windows und POSIX

Bestandteile von Pfaden werden durch Backslash (\) getrennt, ein Wurzelverzeichnis pro Datenträger/Partition, andere File-Handling-Funktionen.

close(det)

Unabhängig vom Betriebssystem, Stream-basiert, gepuffert oder ungepuffert, hat einen eigenen File-Position-Indicator

Streams: FILE * enthält Informationen über einen Stream. Soll nicht direkt verwendet oder kopiert werden, sondern nur über von C-API erzeugte Pointer.

FILE * fopen(char const *path, char const *mode); Offnen eine Datei, Erzeugt FILE-Objekt für Datei an nath, Flags für mode: "r" (Datei Jesen), "w" (In neue oder bestehende geleerte Datei schreiben), "g": (in neue oder hestehende Datei anfilaen). "P+1 (Datei lesen & schreiben). "W+" (neue oder geleette hestehende Datei lesen & überschreiben), "a+" (neue oder bestehende Datei lesen & an Datei anfügen). Gibt Pointer auf erzeugtes ETIE-Ohiekt zurück oder O bei Fehler. FILE * fdopen(int fd, char co aber statt Pfad wird direkt der FD übergeben, int fileno (FILE *stream) gibt FD zurück. Nach API-Umwandlung vorherige nicht mehr verwenden.

int fclose(FILE *file): Schliesst eine Datei, Ruft fflush() (schreibt Inhalt aus Speicher in die Datei) auf, schliesst den Stream, entfernt file aus Speicher und gibt 0 zurück wenn OK, andernfalls EOF int_fgetc(FTLE_*stream): Liest das nächste Byte und erhöht EPL um 1.

char * fqets(char *buf, int n, FILE *stream) liest bis zu <math>n-1 Zeichen aus stream int ungetc(int c, FILE *stream): Lesen rückgängig machen. Nutzt Unget-Stack int foutc(int c. FILE *stream): Schreibt cin eine Datei, int fouts(char *s. FILE *stream)

int feof(FILE *stream) gibt 0 zurück, wenn Dateiende noch nicht erreicht wurde

int ferror(FILE * stream) gibt 0 zurück, wenn kein Fehler auftrat.

schreibt die Zeichen vom String s bis zur terminierenden 0 in stream.

2.2.2. Manipulation des File-Position-Indicator (FPI):

long ftell(FILE *stream) gibt den gegenwärtigen FPI zurück, int fseek (FILE *stream, long offset, int origin) setzt den FPI, analog zu lseek, int rewind (FILE *stream) setzt den Stream zurück.

PROZESSE

Prozesse (aktiv) sind die Verwaltungseinheit des OS für Programme (passiv). Jedem Prozess ist ein virtueller Adressraum zugeordnet.

Fin Prozess umfasst das Abbild eines Programms im Hauntspeicher (text section), die globalen Variablen des Programms (data section), Speicher für den Heap und Speicher für den Stack. Process Control Block (PCB): Das Betriebssystem hält Daten über jeden Prozess in jeweils einem PCB vor. Speicher für alle Daten, die das OS benötigt, um die Ausführung des Prozesses ins Gesamtsystem zu integrieren, u.a.: Diverse IDs. Speicher für Zustand. Scheduling-Informationen. Da

ten zur Synchronisation. Security-Informationen etc. Interrupts: Kontext des aktuellen Prozesses muss im dazugehörigen PCB gespeichert werden (context save): Register, Flags, Instruction Pointer, MMU-Konfiguration. *Interrupt-Handler* überschreibt

den Kontext, Anschliessend wird Kontext aus PCB wiederhergestellt (context restore). Prozess-Erstellung: Das OS erzeugt den Prozess und lädt das Programm in den Prozess. Unter PO-SIX getrennt unter Windows eine einzige Funktion

Prozess-Hierarchie: Baumstruktur, startet bei Prozess 1

3.1. PROZESS-API

fork(void) erzeugt exakte Kopie (C) als Kind des Prozesses (P), mit eigener Prozess-ID (>0) Die Funktion führt in heiden Prozessen den Code an derselben Stelle fort

void exit(int code): Beendet das Programm und gibt code zurück. mid + wait (int +etatue): unterbricht Prozess his Child beendet wurde

pid t waitpid (pid t pid, int *status, int options); wie wait(), aber pid bestimmt, auf welchen Child-Prozess man warten will (> 0 = Prozess mit dieser ID, -1 = irgendeinen, 0 = alle C mit der gleichen Prozessgruppen-ID).

if (fork() = 0) { /* do something in worker process; */ exit(0); }

for (int i = 0: i < n: ++i) { spawn worker(...): }

do { pid = wait(0); } while (pid > 0 || errno ≠ ECHILD); // wait for all children

exec()-Funktionen: Jede davon ersetzt im gerade laufenden Prozess das Programmimage durch ein anderes, Programmargumente müssen spezifiziert werden, (... Lals Liste... v als Arrayi

		Programmargumente als Liste	Programmargumente als Array
Angabe des Pfads	mit neuem Environment	execle()	execve()
	mit altem Environment	execl()	execv()
Suche über PATH		execlp()	execvp()

3 1 1 Zombie- & Ornban-Prozesse

C ist zwischen seinem Ende und dem Aufruf von wait () durch P ein Zombie-Prozess, Dauerhafter Zombie-Prozess: P ruft wegen Fehler wait() nie auf. Orphan-Prozess: P wird vor C beendet. P kann somit nicht mehr auf C warten, was bei Beendung von C in einem dauerhaften Zombie resultiert. Wenn P beendet wird, werden deshalb alle C an Prozess mit pid=1 übertragen, der wait() in einer Endlosschleife aufruft.

ned int sleep (unsigned int seconds): unterbricht Ausführung, bis eine Anzahl Sekunden ungefähr verstrichen ist. Gibt vom Schlaf noch vorhandene Sekunden zurück.

int atexit (void (*function)(void)): Registriert Funktionen für Aufräumarbeiten vor Ende. pid t getpid()/getppid() geben die (Parent-)Prozess-ID zurück.

4. PROGRAMME UND BIBLIOTHEKEN

Assembler → Objekt-Datei → Linker → Executable

Präprozessor: Die Ausgabe des Präprozessors ist eine reine C-Datei (Translation-Unit) ohne Makros, Kommentare oder Präprozessor-Direktiven. Linker: Der Linker verknüpft Objekt-Dateien (und statische Ribliotheken) zu Executables oder dynamischen Ribliotheken. Londer lädt Executables und eventuelle dynamische Bibliotheken dieser in den Hauptspeicher

4.1. FLF (EXECUTABLE AND LINKING FORMAT)

Bingr-Format, das Kompilate spezifiziert. Besteht aus Linking View (wirhtig für Linker für Object-Files und Shared Objects) und Execution View (wichtig für Loader, für Programme und Shared Objects). Struktur: Besteht aus Header, Programm Header Table (execution view), Seamente (execution view), Section Header Ta ble (linking view), Sektionen (linking view)

4.2. SEGMENTE UND SEKTIONEN

Segmente und Sektionen sind eine andere Einteilung für die gleichen Speicherbereiche. View des Loaders sind die Segmente, view des Compilers die Sektionen, Definieren «gleichartige» Daten Der Linker vermittelt zwischen beiden Views

Header: Beschreibt den Aufbau der Datei: Typ. 32/64-bit. Encoding. Maschinenarchitektur. Entrypoint. Infos zu den Einträgen in PHT und SHT.

Program Header Table und Segmente: Tabelle mit n Einträgen, jeder Eintrag (je 32 Byte) beschreibt ein Segment (Tvp und Flags, Offset und Grösse, virtuelle Adresse und Grösse im Speiche schiedlich zur Dateigrösse sein). Ist Verbindung zwischen Segmenten im RAM und im File. Definiert, wo ein Segment liegt und wohin der Loader es im RAM laden soll. Segmente werden vom Loader dynamisch zur Laufzeit verwendet

Section Header Table und Sektionen: Tabelle mit m Einträgen ($\neq n$). Jeder Eintrag (je 40 Byte) beschreibt eine Sektion (Name, Section-Typ, Flags, Offset und Grösse, ...). Werden vom Linker verwendet: Verschmilzt Sektionen und erzeugt auführbares Executable

String-Tabelle: Bereich in der Datei, der nacheinander null-terminierte Strings enthält. Strings werden relativ zum Beginn der Tabelle referenziert. Symbole & Symboltabelle: Die Symboltabelle enthält jeweils einen Eintrag je Symbol (16 Byte: 48

Name 48 Wert 48 Grösse 48 Infol

4.3 RIRLIOTHEKEN

Statische Bibliotheken: Archive von Obiekt-Dateien. Name: lib<name>, a. referenziert wird nur <name> Linker hehandelt statische Bibliotheken wie mehrere Obiekt-Dateien. Ursprünglich gab es nur statische Bibliotheken (Einfach zu implementieren, aber Funktionalität fix).

Dynamische Bibliotheken: Linken erst zur Ladezeit bzw. Laufzeit des Programms. Höherer Aufwand, jedoch austauschbar, Executable enthält nur Referenz auf Bibliothek, Vorteile: Entkonnelter enszyklus. Schnellere Ladezeiten durch Lazy Loading. Flexibler Funktionsumfang.

POSIX SHARED OBJECTS AP

void * dlopen (char * filename, int mode): öffnet eine dynamische Bibliothek und gibt ein Handle darauf zurück. mode: RTLD_NOW: (Alle Symbole werden beim Laden 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 OD verwendet)

* dlsvm (void * handle, char * name); gibt die Adresse des Symbols name aus der mit handle bezeichneten Bibliothek zurück. Keine Typinformationen (Variabel? Funktion?)

// type "func_t" is a address of a function with a int param and int return type edef int (*func t)(int): handle = dlopen("libmylib.so", RTLD NOW): // open library

func_t f = dlsym(handle, "my_function"); // write "my_function" addr into a func_t int *i = dlsym(handle, "my_int"); // get address of "my_int" (*f)(*i): // call "my function" with "my int" as paramete

int dlclose (void * handle): schliesst das durch handle bezeichnete, zuvor geöffnete Objekt char * dlerror(): gibt Fehlermeldung als null-terminierten String zurück.

Konventionen: Shared Objects können automatisch bei Bedarf geladen werden. Der Linker verwendet den Linker-Namen, der Loader verwendet den SO-Namen. Linker-Name: lib + Bibliotheksname + .so /z.B. libmvlib.sol. SO-Name: Linker-Name + . + Versionsnummer /z.B. libmylib.so.2), Real-Name: SO-name + . + Unterversionsnummer (z.B. libmylib.so.2.1)

Shared Objects: Nahezu alle Executeables benötigen zwei Shared Objects: 1 thc. so: Standard C library, ld-linux.so: ELF Shared Object loader (Ladt Shared Objects und rekursiv alle Dependencies). Implementierung dynamischer Bibliotheken: Müssen verschiebbar sein, mehrere müssen in den gleichen Prozess geladen werden. Die Aufgabe des Linkers wird in den Loader bzw. Dynamic Linker

4.5. SHARED MEMORY

Dynamische Bibliotheken sollen Code zwischen Programmen teilen. Code soll nicht mehrfach im Speicher abgelegt werden. Mit Shared Memory kann jedes Programm eine eigene virtuelle Page für den Code definieren. Diese werden auf denselben Frame im RAM gemappt, Benötigt Position Independendent Code (Adressen nur relativ zum Instruction Pointer, Pro Relative Moves via Relative Calls: Mittels Hilfsfunktion wird Rücksprungadresse in Register abgelegt, somit kann relativ dazu gearheitet werden.

Global Offset Table (GOT): Pro dynamische Bibliothek & Executable vorhanden, enthält pro Symbol einen Eintrag. Der Loader füllt zur Laufzeit die Adressen in die GOT ein.

Procedure Linkage Table (PLT): Implementiert Lazy Binding. Enthält pro Funktion einen Eintrag,

dieser enthält Sprungbefehl an Adresse in GOT-Eintrag. Dieser zeigt auf eine Proxy-Funktion, welche den GOT-Eintrag überschreibt. Vorteil: erspart bedingten Sprung.

Jeder Prozess hat virtuell den ganzen Rechner für sich alleine. Prozesse sind gut geeignet für un abhängige Applikationen. Nachteile: Realisierung paralleler Abläufe innerhalb derselben Applikation ist aufwändig. Overhead zu gross falls nur kürzere Teilaktivitäten, gemeinsame Ressour

Threads: narallel ablaufende Aktivitäten innerhalb eines Prozesses, welche auf alle Ressourcen im Prozess gleichermassen Zugriff haben. Benötigen eigenen Kontext und eigenen Stack. Informationen werden in einem Thread-Control-Block abgelegt

5.1. AMDAHLS REGEL

Nur hestimmte Teile eines Algorithmus können narallelisiert werden.

T Ausführungszeit, wenn komplett seriell durchgeführt im Bild: $T = T_0 + T_1 + T_2 + T_3 + T_4$ Anzahl der Prozessoren

T' Ausführungszeit, wenn maximal parallelisiert gesuchte Grösse

 T_* Ausführungszeit für den Anteil, der seriell ausgeführt werden muss Im Bild: $T_s = T_0 + T_2 + T_4$ $T-T_s$ Ausführungszeit für den Anteil, der *parallel* ausgeführt werden *kann Im Bild*: T_1+T_3

 $\frac{T-T_s}{n}$ Parallel-Anteil verteilt auf alle n Prozessoren Im Bild: $\frac{T_1+T_3}{T_1+T_3}$ $T_s + \frac{T - T_s}{n}$ Serieller Teil + Paralleler Teil = T'

Die serielle Variante benötigt also höchstens f mal mehr Zeit als die narallele Variante

f heisst auch ${\it Speedup-Faktor}$, weil die parallele Variante max. f-mal schneller ist als die serielle. Definiert man $s = \frac{T_s}{s}$ also den seriellen Anteil am Algorithmus, dann ist $s \cdot T = T$. Dadurch erhält man f unabhängig von der Zeit:

4. PRODOKARNE UND BIBLIOHEREN

C-Quelle
$$\rightarrow$$
 Pröprozessor \rightarrow Bereinigte C-Quelle \rightarrow Compiler \rightarrow Assembler-Datei \rightarrow

$$f \leq \frac{T}{T_s + \frac{T-x}{1-x}} = \frac{T}{s \cdot T + \frac{T-x}{1-x}} = \frac{T}{s \cdot T + \frac{T-x}{1-x} \cdot T} \Rightarrow f \leq \frac{1}{s \cdot \frac{1-x}{1-x}}$$
Assembler \rightarrow Objekt-Datei \rightarrow Linker \rightarrow Executable

5.1.1. Bedeutung

- Abschätzung einer oberen Schranke für den maximalen Geschwindigkeitsgewinn
- Nur wenn alles parallelisierbar ist, ist der Speedup pro
- portional und maximal f(0,n) = nSonst ist der Speedup mit höherer Prozessor-Anzahl im
- mer geringer (Kurve flacht ab) f(1, n): rein seriell

Grenzwert:

Mit höherer Anzahl Prozessoren nähert sich der Speedup 1 an:

$$\lim_{n\to\infty} \frac{1-s}{n} = 0 \qquad \qquad \lim_{n\to\infty} s + \frac{1-s}{n} = s$$

5.2 POSIX THREAD API int othread create(

erzeugt einen Thread, die ID des neuen Threads wird im Out-Parameter thread, id zurückgege ben. attributes ist ein opakes Objekt, mit dem z.B. die Stack-Grösse spezifiziert werden kann. Die erste auszuführende Instruktion ist die Funktion in stant function, angument ist ein Pointe auf eine Datenstruktur auf dem Heap für die Argumente für start function.



Thread-Attribute

pthread attr t attr: // Variabel erstellen othread attr init (Sattr): // Variabel initialisierer pthread_attr_setstacksize (&attr, 1 < 16); // 64kb Stackgrösse pthread_create (..., &attr, ...); // Thread erstellen pthread_attr_destroy (&attr); // Attribute löschen

Lebensdauer: Lebt solange, bis er aus der Funktion start_function zurückspringt, er pthread exit oder ein anderer Thread pthread cancel aufruft oder sein Prozess beendet wird.

void pthread_exit (void *return_value): Beendet den Thread und gibt den return_value zurück. Das ist äquivalent zum Rücksprung aus start_function mit dem Rückgabewe int othered cancel (othered t thread id): Sendet eine Anforderung, dass der Thread mit thread_id beendet werden soll. Die Funktion wartet nicht, dass der Thread tatsächlich beendet wurde. Der Rückgabewert ist O. wenn der Thread existiert, bzw. ESRCH (error search), wenn nicht. int pthread_detach (pthread_t thread_id): Entfernt den Speicher, den ein Thread belegt hat, falls dieser bereits beendet wurde. Beendet den Thread aber nicht. (Erstellt Doemon Thre int pthread_join (pthread_t thread_id, void **return_value): Wartet solange, bis der Thread mit thread_id beendet wurde. Nimmt den Rückgabewert des Threads im Out-Paramete return_value entgegen. Dieser kann NULL sein, wenn nicht gewünscht. Ruft pthread_detach auf. pthread_t pthread_self (void): Gibt die ID des gerade laufenden Threads zurück.

5.3. THREAD-LOCAL STORAGE (TLS)

TLS ist ein Mechanismus, der globale Variablen per Thread zur Verfügung stellt. Dies benötigt mehrere explizite Einzelschritte: Bevor Threads erzeugt werden: Anlegen eines Keys, der die TLS-Variable identifiziert, Speichern des Keys in einer globalen Variable Im Thread: Aus aus der globalen Variable. Auslesen / Schreiben des Werts anhand des Kevs.

int pthread_key_create(pthread_key_t *key, void (*destructor) (void*)): Erzeugt einen neuen Key im Out-Parameter key. Opake Datenstruktur. Am Thread-Ende Call auf destructor. 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 ve* aufgerufen werden, wenn alle dazugehörende Threads beendet sind. int pthread_setspecific(pthread_key_t key, const void * value) void + athread getenerific(athread key + key) schreiht haw liest den Wert der mit dem

Key in diesem Thread assoziiert ist. Oft als *Pointer auf einen Speicherbereich* verwendet. typedef struct { void print_error (void) {
 error_t * e = pthread_getspecific(error); int code; char *message: printf("Error %d: %s\n e→code, e→message);} .nt +orce_error (void) {
 error_t * e = pthread_getspecific(error); void set_up_error (void) { // am Anfang des Threads aufgerufen e→code = 98; e→message = "file not found":

error, malloc(sizeof(error t)))} // Main und Thread void *thread function (void *) { setu_up_error();
if (force_error () = -1) { print_error (); } int main (int argc. char **argv) { pthread_key_create (&error, NULL); // Key erzeugen pthread_t tid; pthread_create (&tid, NULL, &thread_function, NULL); // Threads erzeugen pthread_join (tid, NULL);

6 SCHEDIII TNG

Auf einem Prozessor läuft zu einem Zeitpunkt immer höchstens ein Thread. Es gibt folgende Zustände: Running (der Thread der gerade läuft). Ready ready running (Threads die laufen können, es aber gerade nicht tun), Waisuspend ting: (Threads, die auf ein Ereignis warten, können nicht direkt in den Running State wechseln), Übergänge von eiwaiting nem Status zum anderen werden in vorgenommen. Dieser Teil vom OS heisst

Das OS registriert Threads auf ein Er-Waiting for Event A eignis und setzt sie in den Zustant «waiting». Tritt das Ereignis auf, ändert das OS den Zustand auf ready. (Es laufen nui E 0 0 int setpriority (int which, id_t who, int prio): setzt den Nice-Wert.

(which: PRIO_PROCESS, PRIO_PSRP oder PRIO_USER, who: ID des Processes, der Gruppe oder des Users) Ready-Queue: In der Ready-Queue (kann auch ein Tree sein) befinden sich alle Threads, die bereit sind zu laufen. Powerdown-Modus: Wenn kein Th road laufhoreit ist schaltet das OS don Prozessor in Standby und wird durch

Arten von Threads: I/O-lastig (Wenig rechnen, viel I/O-Geräte-Kommunikation), Prozessor-lastig (Viel rech-

Arten der Nebenläufigkeit: Kooperativ (Threads entscheiden selbst über Abgabe des Prozessors), Präempi (Scheduler entscheidet, wann einem Thread der Prozessor entzogen wird,

Präemptives Multithreading: Thread läuft, bis er auf etwas zu warten beginnt, Prozessor yielded, ein System-Timer-Interrupt auftritt oder ein bevorzugter Thread erzeugt oder ready wird. Parallele, quasiparallele und nebenläufige Ausführung: Parallel (Tatsächliche Gleichzeitigkeit, n Prozes soren für n Threads). Quasiparallel (n Threads auf < n Prozessoren abwechseind). Nebenläufia (überbegriff für

Bursts: Prozessor-Burst (Thread belegt Thread A Prozessor voll), I/O-Burst (Thread belegt 10 20 30 40 50 60 70 80 90 Zeit Ims Prozessor nicht). Jeder Thread kann als Abfolge von Prozessor-Bursts und I/O-Bursts hetrachtet werden

6.1. SCHEDULING-STRATEGIEN

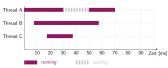
Anforderungen an einen Scheduler können vielfältig sein. Geschlossene Systeme (Hersteller kennt Anwendungen und ihre Beziehungen) VS. Offene Systeme (Hersteller muss von typischen Anwendungen ausgehen)

Anwandungssicht Minimiarung von: Durchlaufzeit (Zeit vom Starten der Thrende bir zu reinem Ende) Antewortzeit (Zeit vom Empfang eines Requests bis die Antwort zur Verfügung steht), Wartezeit (Zeit, die ein Thread in

Aus Systemsicht, Maximierung von: Durchsatz (Anzahl Threads, die pro Intervall bearbeitet werden), Prowendung (Prozentsatz der Verwendung des Prozessors gegenüber der Nichtverwendung)

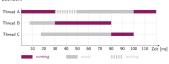
Latenz ist die durchschnittliche Zeit zwischen Auftreten und Verarheiten eines Ereignisses. Im schlimmsten Fall tritt das Ereignis dann auf, wenn der Thread gerade vom Prozessor entfernt wurde. Um die Antwortzeit zu verringern, muss jeder Thread öfters ausgeführt werden. was iedoch zu mehr Thread-Wechsel und somit zu mehr Overhead führt. Die Utilization nimmt also ab. wenn die Antwortzeit verringert wird.

Idealfall: Parallele Ausführung (Dient als idealisierte Schranke)



FCFS-Strategie (First Come, First Served)

Nicht präemptiv: Threads geben den Prozessor nur ab, wenn sie auf waiting wechseln oder sich

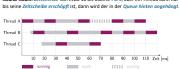


SJF-Strategie (Shortest Job First)

Scheduler wählt den Thread aus, der den kürzesten Prozessor-Burst hat. Bei gleicher Länge wird nach FCFS ausgewählt. Kann kooperativ oder präemptiv sein. Ergibt optimale Wartezeit, kann jedoch nur korrekt implementiert werden, wenn die Länge der Bursts bekannt sind.



Round-Robin: Zeitscheibe von etwa 10 bis 100ms ECES aber ein Thread kann nur solange laufen



Prioritäten-basiert: Jeder Thread erhält eine Nummer, seine Priorität. Threads mit höherer Priorität werden vor Threads mit niedriger Priorität ausgewählt. Threads mit gleicher Priorität werde

nach ECES ausgewählt. Prioritäten ie nach OS unterschiedlich Starvation: Thread mit niedriger Priorität wird immer übergangen und kann nie laufen. Abhilfe

z R mit Aging: in hestimmten Abständen wird die Priorität um 1 erhöht Multi-Level Scheduling: Threads werden in Level aufgeteilt (Priorität, Prozesstvo, Hinter-/Vordergrund) jedes Level hat eigene Ready-Queue und kann individuell geschedulet werden. (z.B. Times) Multi-Level Scheduling mit Foodback: Erschönft ein Thread seine Zeitscheibe wird seine Priorität um 1 verringert. Typischerweise werden die Zeitscheiben mit niedrigerer Priorität grösser und Threads mit kurzen Prozessor-Bursts bevorzugt. Threads in tiefen Queues dürfen zum Ausgleich



6.2 PRIORITÄTEN IN POSIX

Nice-Wert: Jeder Prozess hat einen Nice-Wert von -20 (soll bevorzugt werden) his +19 (nicht bevorzugt) nice [-n increment] utility [argument...]: Nice-Wert beim Start erhöhen oder verringern int nice (int i): Nice-Wert im Prozess erhöhen oder verringern. (Addiert i zum Wert dazu. int getpriority (int which, id t who); gibt den Nice-Wert von p zurück

Priorität hei Thread-Frzeugung setzen

Funktionen ohne attr bevor Thread gestartet wird:

ead_getschedparam(pthread_t thread, int * policy, struct sched_param : param) bzw. int pthread setschedparam(othread t thread, int policy, const struct sched_param * param) nthroad attr t a

MUTEXE UND SEMAPHORE

Jeder Thread hat seinen eigenen Instruction-Pointer und Stack-Pointer Wenn Freehnisse von der Ausführungsreihenfolge einzelner Instruktionen abhängen, spricht man von einer Race Condition. Threads müssen synchronisiert werden, damit keine Race Condition entsteht. Critical Section: Code-Bereich, in dem Daten mit anderen Threads geteilt werden. Muss unbe-

Atomare Instruktionen: Eine atomare Instruktion kann vom Prozessor unterbrechungsfrei ausge-

führt werden. Achtung: Selbst einzelne Assembly-Instruktionen nicht immer atoma Anforderungen an Synchronisations-Mechanismen: Gegenseitiger Ausschluss (Nur ein Thread darf in

Critical Section sein), Fortschritt (Entscheidung, wer in die Critical Section darf, muss in endlicher Zeit getroffen werden), Begrenztes Warten (Thread wird nur n mal übergangen, bevor er in die Critical Section darf).

Implementierung: Nur mit HW-Unterstützung möglich. Es gibt zwei atomare Instruktionen, Test-And-Set (Setzt einen int auf 1 und returnt den vorherigen Wert: test_and_set(int * target) {int value : *target: *target = 1: return value:} und Compare-and-Swap (Überschreibt einen int mit einem spezifizierten Wert, wenn dieser dem erwarteten Wert entspricht: compane_and_swap (int *a, int expected, int new_a) fint value = *a: if (value = expected) { *a = new a: } return value: }.

7.1. SEMAPHORE

Enthält Zähler $z \ge 0$. Wird nur über Post(v) (Erhöht z um 1) und Wait(v) zugegriffen (Wenn z > 0, verringert z um 1 und fährt fort. Wenn z = 0, setzt den Thread in waiting, bis anderer Thread z erhäht, int sem init (sem t *sem, int pshared, unsigned int value): Initialisiert den Semanhor.

typischerweise als globale Variable. pshared = 1: Verwendung über mehrere Prozesse: sen_t sem: int main (int argc, char ** argv) { sem init (&sem. 0. 4): } oderals Parameter für den Thread (Speicher auf dem Stack oder Heap): struct T { sem_t *sem; ... };

int sem_wait (sem_t *sem); int sem_post (sem_t *sem): implementieren Post und Wait. int sem trywait (sem t *sem): int sem timedwait (sem t *sem, const struct timespec *abs_timeout): Sind wie sem_wait, aber brechen ab, falls Dekrement nicht durchgeführt werden kann, sem trywait bricht sofort ab. sem timedwait nach der angegebenen Zeitdauer. int sem_destroy (sem_t *sem): Entfernt Speicher, den das OS mit sem assoziiert ha

```
semanhore free - n:
while (1) {
                                                       while (1) {
  // Warte, falls Customer zu langsam
WAIT (free); // Hat es Platz in Queue?
WAIT (used); // Hat es Elemente in Queue?
produce_item (&buffer[w], ...);
consume (&buffer[r]);
  POST (used): // 1 Floment mehr in Oueue POST (free): // 1
  w = (w+1) % BUFFER_SIZE;
                                                         r = (r+1) % BUFFER_SIZE;
```

Ein Mutex hat einen binären Zustand z, der nur durch zwei Funktionen verändert werden kann: Acquire (Wenn z = 0, setze z auf 1 und fahre fort. Wenn z = 1, blockiere den Thread, bis z = 0), Release (Setzt z =0). Auch als non-blocking-Funktion: int pthread_mutex_trylock (pthread_mutex_t *mutex)

```
// Rejeniel Initialicierung
                                             // Rejected Verwendung in Thread
pthread_mutex_t mutex; // global
int main() {
                                               while (running) {
  pthread_mutex_init (&mutex, 8);
   // run threads & wait for
                                them to finish othered mutex lock (Smutex):
  pthread_mutex_destroy (&mutex);
                                                 // Leave critical section:
pthread_mutex_unlock (&mutex);...}}
```

 $\textbf{Priority Inversion:} \ \textbf{Ein} \ \textbf{\textit{hoch-priorisierter}} \ \textbf{Thread} \ C \ \text{wartet auf eine Ressource, die von einem } \textbf{\textit{nied-priorisierter}} \ \textbf{\textit{Thread}} \ C$ riger priorisierten Thread A gehalten wird. Ein Thread mit Prioriät zwischen diesen heiden Threads erhält den Prozessor. Kann mit **Priority Inheritance** gelöst werden: Die Priorität von A wird ${\it tempor\"ar} \ {\it auf} \ {\it die} \ {\it Priorit\"at} \ {\it von} \ {\it C} \ {\it gesetzt}, \ {\it damit} \ {\it der} \ {\it Mutex} \ {\it schnell} \ {\it wieder} \ {\it freigegeben} \ {\it wird}.$

8. SIGNALE, PIPES UND SOCKETS

Signale ermöglichen es, einen Prozess von aussen zu unterbrechen, wie ein Interrunt. Unterbrechen des gerade laufenden Prozesses/Threads, Auswahl und Ausführen der Signal-Handler-Funktionen, Fortsetzen des Prozesses. Werden über ungültige Instruktionen oder Abbruch auf Seitens Benutzer ausgelöst. Jeder Prozess hat nro Signal einen Handler.

Handler: Ignore-Handler (ignoriert das Signal), Terminate-Handler (beendet das Programm), Abnormal Terminate-Handler (beendet Programm und erzeugt Core-Dump). Fast alle Signale ausser SIGKILL und STRSTOP können überschrieben werden.

Programmfehler-Signale: SIGFPE (Fehler in arithmetischen Operation), SIGILL (Ungültige Instru SIGSEGV (Unaültiger Speicherzugriff), SIGSYS (Unaültiger Systemaufruf)

Prozesse abbrechen: SIGTERM (Normale Anfrage an den Prozess, sich zu beenden), SIGINT (Nachdrück Aufforderung an den Prozess, sich zu beenden), SIGQUIT (Wie SIGINT, aber anormale Terminierung), SIGABRT (Wie SIGQUIT, aber vom Prozess an sich selber), SIGKILL (Prozess wird «abgewürgt», kann nicht verhindert werden) Stop and Continue: SIGTSTP (Versetzt den Prozess in den Zustand stopped, ähnlich wie waiting), SIGSTOP (Wie SIGTSTP, aber kann nicht ianoriert oder abgefangen werden), SIGCONT (Setzt den Prozess fort)

Signale von der Shell senden: kill 1234 5678 sendet SIGTERM an Prozesse 1234 und 5678 t sigaction (int signal, struct sigaction *new, struct sigaction *old): Definiert Signal-Handler für signal, wenn new $\neq 0$. (Eigene Signal-Handler definiert via signation struct sa_handler: Zu callende Funktion, sa_mask: Blockierte Signale während Ausführung, bearbeitet nur durch sig*set()-

Funktionen: sigemptyset, sigfillset, sigaddset, sigdelset, sigismember)

Fine geöffnete Datei entspricht einem Fintrag in der File-Descriptor-Tabelle (FDT) im Prozess 71griff über File-API (open, close, read, write, ...). Das OS speichert je Eintrag der Prozess-FDT einen Verweis auf die globale FDT. Bei fork() wird die FDT auch kopiert.

int dun (int source fd): int dun? (int source fd, int destination fd): Dunlizieren den File-Descriptor source_fd. dup alloziert einen neuen FD, dup2 überschreibt destination_fd.

8.2.1. Umleiten des Ausgabestreams

```
int fd = open ("log.txt", ...);
int id = fork ();
if (id = 0) { // child
dup2 (fd, 1); // duplicate fd for log.txt as standard output
// e.g. load new image with exec*, fd's remain
} else { /* parent */ close (fd); }
```

Fine Pine ist eine «Datei» (Fine Datei muss auf onen ich ose etc. unterstützen) im Hauntsneicher, die über zwei File-Deskriptoren verwendet wird: read end und write end. Daten, die in write end geschrieben werden, können aus read end genau einmal und als FIFO gelesen werden. Pipes erlauben Kommunikation über Prozess-Grenzen hinweg, Ist unidirektional.

```
if (id = 0) { // Child
int fd [2]; // 8 = read, 1 = write
                                             close (fd [1]); // don't use write end
pipe (fd);
int id = fork():
                                             char buffer [BST7F]:
                                              nt n = read (fd[0], buffer, BSIZE);
Pipe lebt nur so lange, wie mind, ein Ende geöff-
net ist. Alle Read-Ends geschlossen → SIGPIPE an
                                             close (fd[0]); // don't use read end
Write-End. Mehrere Writes können zusammen- char * text = "La li lu":
gefasst werden. Lesen mehrere Prozesse diesel- write (fd [1], text, strlen(text) + 1);
he Pine, ist unklar, wer die Daten erhält.
```

int mkfifo (const char *path, mode_t mode): Erzeugt eine Pipe *mit Namen und Pfad* im Dateisystem. Hat via mode permission bits wie normale Datei. Lebt unabhängig vom erzeugenden Prozess, je nach System auch über Reboots hinweg. Muss explizit mit unlink gelöscht werden.

8.3. SOCKETS

Ein Socket repräsentiert einen Endpunkt auf einer Maschine. Kommunikation findet im Regelfall zwischen zwei Sockets statt (UDP, TCP über IP sowie Unix-Domain-Sockets). Sockets benötigen für Kommunikation einen Namen: //P- /P-Adresse Postor I

int socket(int domain, int type, int protocol); Erzeugt einen neuen Socket als «Datei». Socket sind nach Erzeugung zunächst unbenannt. Alle Operationen blockieren per default. Domain (AF_UNIX, AF_INET), type (SDCK_DGRAM, SDCK_STREAM), protocol (System-spezifisch, 0 = Default-Protocol)

Client: connect (Verbindung unter Angabe einer Adresse aufbauen), send / write (Senden von Daten, $0-\infty$ mal), ecv / read (Emplangen von Daten, $0 - \infty$ mal), close (Schliessen der Verbindung) Server: bind (Festlegen einer nach aussen sichtbaren Adresse), Listen (Bereitstellen einer Queue zum Sammeln von

unfragen von Clients), accept (Erzeugen einer Verbindung auf Anfrage von Client), recv / read (Empfangen von Daten. $0-\infty$ mall. Send / write (Senden von Daten. $0-\infty$ mall. close (Schliessen der Verbindung)

```
ct sockaddr_in ip_addr;
ip_addr.sin_port = htons (443); // default HTTPS port
inet_pton (AF_INET, "192.168.0.1", &ip_addr.sin_addr.s_addr);
      port in memory: 0x01 0xBB
addr in memory: 0xC0 0xA8 0x00 0x01
```

htons konvertiert 16 Bit von Host-Ryte-order ((E) zu Network-Ryte-Order (8E), hton3 32 Bit, ntohs und ntohl sind Gegenstücke inet_pton konvertiert protokoll-spezifische Adresse von String zu Network-BO. inet_ntop ist das Gegenstück (network-to-p

int bind (int socket, const struct sockaddr *local address, socklen t addr len) Bindet den Socket an die angegebene, unbenutze lokale Adresse, wenn noch nicht gebunden. Blockiert, bis der Vorgang abgeschlossen ist.

int connect (int socket, const struct sockaddr *remote_addr, socklen_t addr_len): Aufbau einer Verbindung. Bindet den Socket an eine neue, unbenutzte lokale Addresse, wenn noch nicht gebunden. Blockiert, bis Verbindung steht oder ein Timeout eintritt. int listen (int socket, int backlog): Markiert den Socket als «bereit zum Empfang vor

Verbindungen». Erzeugt eine Warteschlange, die so viele Verbindungsanfragen aufnehmen kann. wie backlog angibt. int accent (int socket struct sockedde tremote address socklen t address len):

Wartet bis eine Verbindungsanfrage in der Warteschlange eintrifft. Erzeugt einen neuen Socket und bindet ihn an eine neue lokale Adresse. Die Adresse des Clients wird in remote_address geschrieben. Der neue Socket kann keine weiteren Verbindungen annehmen, der bestehende schon.

8.3.1. Typisches Muster für Server

```
int server_fd = socket ( ... ); bind (server_fd, ...); listen (server_fd, ...);
while (running) {
  int client_fd = accept (server_fd, 0, 0);
delegate_to_worker_thread (client_fd); // will call close(client_fd)
```

send (fd. buf. len. 0) = write (fd, buf, len); recv (fd, buf, len, 0) = read (fd, buf, len): Senden und Empfangen von Daten. Puffern der Daten ist Aufgabe des Netzwerkstacks. int close (int socket): Schliesst den Socket für den aufrufenden Prozess. Hat ein anderer Prozess den Socket noch geöffnet, bleibt die Verbindung bestehen. Die Gegenseite wird nicht benachrichtigt. int shutdown (int socket, int mode): Schliesst den Socket für alle Prozesse und baut die entsprechende Verbindung ab. mode: SHUT_RD (Keine Lese-Zugriffe mehr), SHUT_WR (Keine Schreib-Zuariffe mehr), SHUT_RDWR (Keine Lese- oder Schreib-Zugriffe mehr)

9. MESSAGE PASSING UND SHARED MEMORY

Prozesse sind voneinander isoliert, müssen iedoch trotzdem miteinander interagieren. Message Passing ist ein Mechanismus mit zwei Operationen: Send (Koniert die Nochricht aus dem Prozess: se (message)), Receive: (Kopiert die Nochricht in den Prozess: receive (message)). Dabei können Implementierungen nach verschiedenen Kriterien unterschieden werden (Feste oder Variable Nachrichtengrösse, direkte oder indirekte / synchrone oder asynchrone Kommunikation, puffering, mit oder ohne Prioriäten für Nachrichten)

Feste oder variable Nachrichtengrösse: feste Nachrichtengrösse ist einfacher zu implementieren, aber umständlicher zu verwenden als variable Nachrichtengrösse. Direkte Kommunikation: Kommunikation nur zwischen genau zwei Prozessen, Sender muss Emp-

fänger kennen. Es gibt symmetrische direkte Kommunikation (Emafänger muss Sender auch kennen) und asymmetrische direkte Kommunikation (Empfänger muss Sender nicht kennen).

Indirekte Kommunikation: Prozess sendet Nachricht an Mailhoxen, Ports oder Queues, Empfänger empfängt aus diesem Objekt. Beide Teilnehmer müssen die gleiche(n) Mailbox(en) kennen. Lebenszyklus Queue: Wenn diese Queue einem Prozess gehört, lebt sie solange wie der Prozess. Wenn sie dem OS gehört, muss das OS das Löschen übernehmen.

Synchronisation: Blockierendes Senden (Sender wird solange blockiert, bis die Nachr angen wurde), Nicht-blockierendes Senden (Sender sendet Nachricht und fährt sofort weiter). Blockierendes Empfangen (Empfanger wird blockiert, bis Nachricht verfügbar), Nicht-blockierendes Empfangen (Empfänger erhält Nachricht, wenn verfügbar, oder 0)

Rendezvous: Sind Empfang und Versand beide blockierend, kommt es zum Rendezvous, sobald heide Seiten ihren Aufruf getätigt haben. Impliziter Synchronisatio

// Producer	// Consumer
message msg;	message msg;
open (Q);	open (Q);
while (1) {	while (1) {
produce_next (&msg);	receive (Q, &msg); // blocked until rec.
send (Q, &msq); // blocked until ser	t consume_next (&msq);
1	1

Pufferung: Keine (Queue-Länge ist 0, Sender muss blockieren), Beschränkte (Maximal n Nachrichten, Sender blo nn Queue voll ist.), Unbeschränkte (Beliebig viele Nachrichten, Sender blackiert nie).

Prioriäten: In manchen Systemen können Nachrichten mit Prioritäten versehen werden. Der Empänger holt die Nachricht mit der höchsten Priorität zuerst aus der Queue

9.1.1. POSIX Message-Passing

OS-Message-Queues mit variabler Länge, haben mind. 32 Prioritäten und können synchron und asynchron verwendet werden.

mqd_t mq_open (const char *name, int flags, mode_t mode, struct mq_attr *attr): Offnet eine Message-Queue mit systemweitem name, returnt Message-Queue-Descriptor. (name mit / beginnen, flags & mode wie bei Dateien, mq_attr: Div. Konfigs & Queue-Status, R/W mit mp_getattr/mq_setattr) int mq_close (mqd_t queue): Schliesst die Queue mit dem Descriptor queue für diesen Prozess. int mq_unlink (const char *name): Entfernt die Queue mit dem Namen name aus dem System. Name wird sofort entfernt und Queue kann anschliessend nicht mehr geöffnet werden. int mq_send (mqd_t queue, const char *msg, size_t length, unsigned int priority):

Sendet die Nachricht, die an Adresse msg beginnt und length Bytes lang ist, in die queue. int mq_receive (mqd_t queue, const char *msg, size_t length, unsigned int *priority):

*Kopiert die nächste Nachricht aus der Queue in den Puffer, der an Adresse msg beginnt und length Bytes lang ist. Blockiert, wenn die Queue leer ist.

9.2 SHARED MEMORY

Frames des Hauntspeichers werden zwei (oder mehr) Prozessen Ps und Ps zugänglich gemacht. In P_1 wird Page V_1 auf einen Frame F abgebildet. In P_2 wird Page V_2 auf $\emph{denselben}$ Frame Fabgebildet. Beide Prozesse können beliebig auf dieselben Daten zugreifen. Im Shared Memory müssen relative Adressen verwendet werden.

9 2 1 POSIX API

Das OS benötigt eine «Datei» S. das Informationen über den gemeinsamen Speicher verwaltet und eine Mapping Table je Prozess. int fd = shu onen ("/mysharedmemory", 0 RDWR | 0 CREAT, S TRUSR | S TWUSR):

Erzeugt falls nötig und öffnet Shared Memory /mysharedmemory zum Lesen und Schreiben int ftruncate (int fd. offset t length); Setzt Grösse der «Datei». Muss zwingend nach SM- Erstellung gesetzt werden, um entsprechend viele Frames zu allozieren. Wird für Shared Memory mit ganzzahligen Vielfachen der Page-/Framegrösse verwendet.

int close (int fd): Schliesst «Datei». Shared Memory bleibt aber im System

int shm unlink (const char * name): Löscht das Shared Memory mit dem name. (bleit

int munmap (void *address, size t length); Entfernt das Mapping.

size of shared memory. // size t length (same as used in ftruncate) PROT_READ | PROT_WRITE, / // int file descripto // off_t offset (start map from first byte)

9.3. VERGLEICH MESSAGE-PASSING UND SHARED MEMORY

Shared Memory ist schneller zu realisieren, aber schwer wartbar. Message-Passing erfordert mehr Engineering-Aufwand, schlussendlich aber in Mehr-Prozessor-Systemen bald performanter

9.4. VERGLEICH MESSAGE-QUEUES UND PIPES

Message-Queues	Pipes
- bidirektional	- unidirektional
- Daten sind in einzelnen Messages organisiert	- übermittelt Bytestrom an Daten
	- FIFO-Zugriff
- Haben immer einen Namen	- Müssen keinen Namen haben

10. UNICODE

10.1. ASCII - AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE Hat 128 definierte Zeichen (erste Herzahl = Zeile zweite Herzahl = Spalte d.h. 41. = 4).

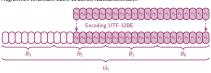
	0	1	2	3	4	5	6	7	8	9	A	В	С	D	Е	F
0	NUL							BEL	88	TAB	LF		FF	CR		SI
1									CAN	EM	58				RS	US
2	ш	1		#	\$	%	&	,	()	*	+	,	-		/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	0	A	В	С	D	Е	F	G	Н	I	J	K	L	М	N	0
5	P	Q	R	S	Т	U	V	W	Х	Y	Z	[١]	^	
6	- (a	ъ	С	d	е	f	g	h	i	i	k	1	m	n	0
7	р	q	r	s	t	u	v	w	х	у	z	{	1	}	~	DEL

Codepages: unabhängige Erweiterungen auf 8 Bit. Jede ist unterschiedlich und nicht erkennbar Unicode: Hat zum Ziel, einen eindeutigen Code für jedes vorhandene Zeichen zu definieren. D8 00h bis DF FFh sind wegen UTF-16 keine gültigen Code-Points. Code-Points (CP): Nummer eines Zeichen - «welches Zeichen?»

Code-Unit (CU): Einheit, um Zeichen in einem Encoding darzustellen (bietet den Speicherplatz. i-tes Bit des unkodierten CPs, $U_i = i$ -tes Code-Unit des kodierten CPs, $B_i = i$ -tes Byte des kodierten CPs

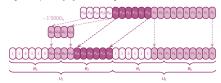
10.2 LITE-32

Jede CU umfasst 32 Bit. jeder CP kann mit einer CU dargestellt werden. Direkte Kopie der Bits in die CU bei Big Endian, bei Little Endian werden P_0 bis P_7 in B_3 kopiert usw. Wird häufig intern in enwendet Ohere 11 Rits oft «zweckentfremdet



10.3. UTF-16

lede CII umfasst 16 Bit ein CP henötigt 1 oder 2 CUs Encoding muss Endianness herücksichtigen Die 2 CUs werden Surrogate Pair genannt. Up: high surrogate. Up: low surrogate. Bei 2 Bytes (1 CU) wird direkt gemappt und vorne mit Nullen aufgefüllt. Bei 4 Bytes sind D8 00h bis DF FFh (Bits 17-21) wegen dem Separator ungültig und müssen «umgerechnet» werder



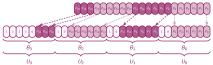
Encoding von U+10'437 (₩) 00 0100

- 1. Code-Point P minus 1 00 00₆ rechnen und in Binär umwandlen
- P = 1.0437, Q = 1.0437, -1.0000, = 0437, = 00.0000001.0000110111,
- 3. Oberer Wert mit D8 00, und unterer Wert mit DC 00, addieren, um Code-Units zu erhalten $U_1 = 00.01_h + D8.00_h = D8.01_h, U_2 = 01.37_h + DC.00_h = DD.37_h$
- Zu BE/LE zusammensetzen
- $BE = D801DD37_h, LE = 01D837DD_h$

10.4 LITE-8

Jede CU umfasst 8 Bit, ein CP benötigt 1 bis 4 CUs. Encoding muss Endianness nicht berücksichti gen. Standard für Webpages. Echte Erweiterung von ASCII.

Code-Point in	U_3	U_2	U_1	U_0	signifikant
0 _h - 7F _h				Oxxx xxxx _b	7 bits
80 _h - 7 FF _h			110x xxxx _b	10xx xxxx _b	11 bits
8 00 _h - FF FF _h		1110 xxxx _b	10xx xxxx _b	10xx xxxx _b	16 bits
1 00 00 _h - 10 FF FF _h	11110xxx _b	10xx xxxx _b	10xx xxxx _b	10xx xxxx _b	21 bits



Beispiele

- \ddot{a} : $P = E4_h = 0.0011 10.011$
- $\Rightarrow P_{10}...P_6 = 0.0011_b = 03_b, P_5...P_0 = 10.0100_b = 24_b$ $\Rightarrow U_1 = C0_h (= 11000000_h) + 03_h = C3_h, U_0 = 80_h (= 10000000_h) + 24_h = A4_h$
- $\Rightarrow \bar{a} = \underline{C3A4}_{b}$
- φ: P = 1E B7_h = 0001 11 1010 11 0111_h
- $\begin{array}{l} \vdots \\ P_{15}...P_{12} = 01_{b}, P_{11}...P_{6} = 3A_{b}, P_{5}...P_{0} = 37_{b} \\ \Rightarrow U_{2} = \mathrm{EO}_{b} \ (= 1110\,0000_{b}) + 01_{b} = \mathrm{E1}_{b}, U_{1} = 80_{b} + 3A_{b} = \mathrm{BA}_{b}, U_{0} = 80_{b} + 37_{b} = \mathrm{B7}_{b} \end{array}$ $\Rightarrow \breve{a} = E1 BA B7$

10.5. ENCODING-BEISPIELE

Zeichen	Code-Point	UTF-32BE	UTF-32LE	UTF-8	UTF-16BE	UTF-16LE
A	41 _h	00 00 00 41 _h	41 00 00 00 _h	41 _h	00 41 _h	41 00 _h
ā	E4 _h	00 00 00 E4 _h	E4 00 00 00 _h	C3 A4 _h	00 E4 _h	E4 00 _h
α	3 B1 _h	00 00 03 B1 _h	B1 03 00 00 _h	CE B1 _h	03 B1 _h	B1 03 _h
ă	1E B7 _h	00 00 1E B7 _h	B7 1E 00 00 _h	E1 BA B7 _h	1E B7 _h	B7 1E _h
, V	1 03 30 _h	00 01 03 30 _h	30 03 01 00 _h	F0 90 8C B0 _h	D8 00 DF 30 _h	00 D8 30 DF _h

Rei LE / RE werden aur die Zeichen innerhalb eines Code-Points vertauscht, nicht die Code-Points an sich

11. EXT2-DATEISYSTEM

Partition (Ein Teil eines Datenträgers, wird selbst wie ein Datenträger behandelt.). Volume (Ein Datenträger oder ein Partition davon.), Sektor (Kleinste lagische Untereinheit eines Volumes. Daten werden als Sektoren transferiert. Grösse ist von HW definiert. Enthält Header, Daten und Error-Correction-Codes.), Format (Layout der logischen Strukturen auf dem Datenträger, wird vom Dateisystem definiert.)

11 1 BLOCK

Ein Block besteht aus mehreren aufeinanderfolgenden Sektoren (1 KB, 2 KB oder 4 KB (normal)). Das gesamte Volume ist in Blöcke aufgeteilt und Speicher wird nur in Form von Blöcken alloziert. Ein Block enthält nur Daten einer einzigen Datei. Es gibt Logische Blocknummern (Blockn Anfang der Datei aus gesehen, wenn Datei eine ununterbrochene Abfolge von Blöcken wäre) und Physische Block-

11.2 INODES



Lokalisierung: Alle Inodes aller Blockgruppen gelten als eine grosse Tabelle. Startet mit 1. Erzeugung: Neue Verzeichnisse werden in der Blockgruppe angelegt, die von allen Blockgruppen mit überdurchschnittlich vielen freien Inodes die meisten Blöcke frei hat. Dateien in der Blockgruppe des Verzeichnis oder nahen Gruppen. Bestimmung anhand Inode-Usage-Bitmaps File-Holes: Bereiche in der Datei, in der nur Nullen stehen. Ein solcher Block wird nicht alloziert

11 3 RIOCKGRUPPE

Eine Blockgruppe besteht aus mehreren aufeinanderfolgenden Blöcken bis zu 8 mal der Anzahl Bytes in einem Block.

Layout: Block 0 (Kopie des Superblocks), Block 1 bis n (Kopie der Gruppendeskriptorentabelle), Block n+1(Block-Usage-Bitmap mit einem Bit je Block der Gruppe), $Block \ n+2$ (Inade-Usage-Bitmap mit einem Bit je Inade der Gruppe), Block n+3 bis n+m+2 (Tabelle aller Inodes in dieser Gruppe), Block n+m+3 bis Ende der Gruppe (Blöcke der eigentlichen Daten)

Superblock: Enthält alle Metadaten über das Volume (Anzahlen, Zeitpunkte, Statusbits, Erster Inode, ...) immer an Byte 1024, wegen möglicher Bootdaten vorher.

Sparse Superblock: Kopien des Superblocks werden nur in Blockgruppe 0, 1 und allen reinen Poenzen von 3, 5 oder 7 gehalten (Sehr hoher Wiederherstellungsgrad, aber deutlich weniger Platzverbrauch). Gruppendeskriptor: 32 Byte Beschreibung einer Blockgruppe, (Blockgrupper des Block-Usage-Bitmans Gruppe, Anzahl der Verzeichnisse in der Gruppe)

Gruppendeskriptortabelle: Tabelle mit Gruppendeskriptor pro Blockgruppe im Volume. Folgt direkt auf Superblock(-kopie). $32 \cdot n$ Bytes gross. Anzahl Sektoren = $\frac{32 \cdot n}{1244 \cdot n^2}$

Verzeichnisse: Enthält *Dateieinträge* mit variabler Länge von 8 - 263 Byte (48 In 18 Dateinamenlänge, 18 Dateityp, 0 - 2558 Dateiname aligned auf 48). Defaulteinträge: «.» und «..» Links: Es gibt Hard-Links (gleicher Inode, verschiedene Pfade: Wird von verschiedenen Dateieinträgen referenziert)

und Symbolische Links (Wie eine Datei, Datei enthält Pfad anderer Datei). 11.4. VERGLEICH FAT. NTFS. EXT2

FAT	Ext2	NTFS
Verzeichnis enthält alle Da- ten über die Datei Datei ist in einem einzigen Verzeichnis Keine Hard-Links möglich	Dateien werden durch In- odes beschrieben Kein Link von der Datei zu- rück zum Verzeischnis Hard-Links möglich	Dateien werden durch File-Records beschrieben Verzeichnis enthält Namen und Link auf Datei Link zum Verzeichnis und Name sind in einem Attribi Hard-Links möglich

Vergrössert die wichtigen Datenstrukturen, besser für grosse Dateien, erlaubt höhere maximale Dateigrösse. Blöcke werden mit Extent Trees verwaltet, Journaling wird eingeführt.

12.1. EXTENTS

Beschreiben ein Intervall physisch konsekutiver Blöcke. Ist 12 Byte gross (48 Jogische Blo physische Blocknummer, 2B Anzahl Blöcke). Positive Zahlen = Block initialisiert. Negativ = Block voralloziert. Im Inode hat es in den 60 Byte für direkte und indirekte Block-Adressierung Platz für 4 Extents und einen Header

Extent Trees Index-Knoten (Innerer Knoten des Baums, besteht aus Index-Eintrag und Index-Block), Index-Ein traa (Enthält Nummer des physischen Index-Blocks und kleinste loaische Blocknummer aller Kindknoten). Index-Block (Enthält eigenen Tree-Header und Referenz auf Kindknoten)

Extent Tree Header: Renotiet ab 4 Extents, weil zusätzlicher Block, Magic Number E3 0A. (28), Anzahl Einträge, die direkt auf den Header folgen (28), Anzahl Einträge, die maximal auf den Header folgen können (28), Tiefe des Baums (28) - (0: Einträge sind Extents, ≥1: Einträge sind Index Nodes), Reserviert

Index Node: Spezifiziert einen Block der Extents enthält. Besteht aus einem Header und den Ext. ents (max. 340 bei 4 KB Blockgrösse). Ab 1360 Extents zusätzlicher Block mit Index Nodes nötig.

12.1.1. NOLATION			
(in)direkte Addressierung	Extent-Trees		
$direkte\ Bl\"{o}cke$: Index \mapsto Blocknr.	Indexknoten: Index → (Kindblocknr, kleinste Nummer der 1. logischen Blöcke aller Kinder)		
indirekte Blöcke: indirekter Block.Index → direkter Block	Blattknoten: Index → (1. logisch. Block, 1. phy. Block, Anz Blöcke)		
	Header: Index → (Anz. Fintrage. Tiefe)		

Beispiel Berechnung 2MB grosse, konsekutiv gespeicherte Datei, 2KB Blöcke ab Block 2000h (In-)direkte Block-Adressierun

 $2 \text{ MB} = 2^{21} \text{ B}$, $2 \text{ KB} = 2^{11} \text{ B}$, $2^{21-11} = 2^{10} = 400 \text{ b}$ Blöcke von 2000 b bis 23 FF_b

 $0 \mapsto 20 \ 00_h, 1 \mapsto 20 \ 02_h, ..., B_h \mapsto 20 \ 0B_h \ C_h \mapsto 24 \ 00_h \ (indirekter \ Block)$ $14\,00_{b}, 0_{b} \mapsto 20\,0C_{b}, 14\,00_{b}, 1_{b} \mapsto 20\,0D_{b}, ..., 14\,00_{b}, 3\,F3_{b} \mapsto 23\,FF_{b}$

Extent Trees Header: $0 \mapsto (1, 0)$ Extent: $1 \mapsto (0.2000, 400)$

12.2. JOURNALING

Wenn Dateisystem beim Erweitern einer Datei unterbrochen wird, kann es zu Inkonsiste kommen, Journaling verringert Zeit für Überprüfung von Inkonsistenzen erheblich.

Journal: Datei, in die Daten schnell geschrieben werden können. Bestenfalls 1 Extent.

Transaktion: Folge von Einzelschritten, die gesamtheitlich vorgenommen werden sollten Journaling: Daten als Transaktion ins Journal, dann an finale Position schreiben (committing), Trans-

aktion aus dem Journal entfernen. Journal Renlay: Transaktionen im Journal werden nach Neustart noch einmal ausgeführt

Journal Modi: (Full) Journal (Metadaten und Datei-Inhalte ins Journal, sehr sicher aber Janasam), Ordered (Nu Metadaten ins Journal, Dateiinhalte werden immer vor Commit geschrieben), Writeback (Nur Metadaten ins Journal beliebige Reihenfolge, nicht sehr sicher aber schnell).

13. X WINDOW SYSTEM

Setzt Grundfunktionen der Fensterdarstellung. Ist austauschbar, realisiert Netzwerktransparenz Plattformunabhängig, legt die GUI-Gestaltung nicht fest.

Programmgesteuerte Interaktion: Benutzer reagiert auf Programm.

Ereignisgesteuerte Interaktion: Programm reagiert auf Benutzer.

Fenster: Rechteckiger Bereich des Bildschirms. Es gibt eine Baumstruktur aller Fenster, der Bildschirm ist die Wurzel (z.B. Dialogbax, Scrollbar, Buttan...).

Display: Rechner mit Tastatur, Zeigegerät und 1..m Bildschirme

X Client: Applikation, die einen Display nutzen will. Kann lokal oder entfernt laufen

X Server: Softwareteil des X Window System, der ein Display ansteuert. Beim Nutzer.

Nicht nur X Window System, sondern auch Window Manager (Verwaltung der sichtbaren Fenster, Umrandung, Knöpfe. Läuft im Client und realisiert Window Layout Policy) und Desktop Manager (Desktop-Hilfsmittel wie Taskleiste, Dateimanager, Papierkorb etc.).

13.2 YIIR

13.1. GUI ARCHITEKTUR

Ist das C Interface für das X Protocol. Wird meist nicht direkt verwendet.

Funktionen: XOpenDisplay() öffnet Verbindung zum Display, NULL = Wert von DISPLAY Umgebungsvariabel), XCloseDisplay() schliesst Verbindung, XCreateSimpleWindow() erzeugt ein Fenster, XDestroyWindow() entfernt ein Fenster & Unterfenster, XMapWindow() bestimmt, dass ein Fenster angezeigt werden soll (unhide), XMapRaised() bringt Fenster in den Vordergrund, XMapSubwindows() zeigt alle Unterfenster an, Expose Event, XUnnapWindow() versteckt Fenster, XUnmapSubwindows() versteckt Unterfenster, UnmapNotify Event

X Protocol: Legt die Formate für Nachrichten zwischen X Client und Server fest. Requests (Diensta forderungen, Client → Server), Replies (Antworten auf Requests, Client ← Server), Events (Ereignismeldungen, Client ← Server), Errors (Fehlermeldungen auf vorangegangene Requests, Client ← Server) Request Buffer: Nachrichtenpufferung auf der Client Seite. Für Effizienz, Pufferung bei Freignis-

sen: Werden beim X Server und beim Client gepuffert. Server-Seitig berücksichtigt Netzw fügharkeit Client-Seitige hält Events hereit X Event Handling: Ereignisse werden vom Client verarbeitet oder weitergeleitet. Muss festlegen,

velche Typen er empfangen will. XSelectInput() legt fest, welche Events via Event-Masken emfpa den, z.B. ExposureMask, XNextEvent() kopiert den nächsten Event aus dem Buffer. 13.3 ZEICHNEN

Ressourcen: Server-seitige Datenhaltung zur Reduktion des Netzwerkverkehrs. Halten Informationen im Auftrag von Clients, Diese identifizieren Informationen mit IDs. Kein Hin- und Herkopieren komplexer Datenstrukturen nötig. (z.B. Window, Pixmap, Colormap, Font, Graphics-Context) Pufferung verdeckter Fensterinhalte: Minimal (keine Pufferung) oder Optional (Hintergrundspeicher zum

Pixmap: Server-Seitiger *Grafikspeicher*, wird immer gecached. X Grafikfunktionen: Bilddarstellung mittels Rastergrafik und Farbtabelle. Erlauben das Zeichnen von Figuren, Strings und Texten, Ziele für das Zeichnen können Fenster oder Pixmap sein Graphics Context: Leet diverse Eigenschaften fest, die Systemaufrufe nicht direkt unterstützen

(z.B. Linjendicke, Farben, Füllmuster), Client kann mehrere GCs gleichzeitig nutzen

14 MELTDOWN

13.4. FENSTER SCHLIESSEN Schaltfläche wird vom Window Manager erzeugt. X weiss nichts über spezielle Bedeutung der Schaltfläche, der Window Manager schliesst das Fenster. Es gibt ein Protokoll zwischen Window Manager und Applikation. ClientMessage Event mit WM_DELETE_MESSAGE.

Atoms: ID eines Strings, der für Meta-Zwecke benötigt wird. Erspart Parsen der Strings. Properties: Werden mit iedem Fenster assoziiert. Generischer Kommunikations-Mecha

zwischen Applikation und Window Manager. WM_PROTOCOLS: Von X Standard definierte Anzahl an Protokollen, die der Window Manager verstehen soll. Ein Client kann sich für Protokolle *registrieren*.

WM_DELETE_WINDOW: Wird beim Drücken des «x» vom Window Manager an den Client geschickt.

Meltdown ist eine HW-Sicherheitslücke, die es ermöglicht, den gesamten physischen Hauptspeicher auszulesen. Ein Prozess kann dadurch geheime Informationen anderer Prozesse lesen.

Der Prozessor muss dazu gebracht werden können:

1. aus dem geschützten Speicher an Adresse a das Byte m_a zu lesen2. die Information m_a in irgendeiner Form f_a zwischenze binäre Fragen der Form «f_a ≟ i» zu beantworten

Von i = 0 bis i = 255 iterieren: $f_a \stackrel{?}{=} i$ 5. Über alle a iterieren

bzw. User-Mode. Nachteil: System wieder langsam.

14.1 PERFORMANCE-OPTIMIERLINGEN Mapping des Speichers in jeden virtuellen Adressraum, Out-of-Order Execution (03E), Spekulative

Seiteneffekte O3E: Cache weiss nicht, ob Wert spekulativ angefordert wurde und speichert alles. Da Wert als Teil des Tags gespeichert und die Zeit gemessen werden kann, die ein Speicherzugriff benötigt, kann man herausfinden, ob etwas im Cache ist oder nicht (Timing Side Channel Attack)

Tests: Verschiedene CPUs (Intel, einige ARMs, keine AMDs) und verschiedene OS (Linux, Windows 10) sind betroffen. Geschwindigkeit bis zu 500 KB pro Sekunde bei 0.02% Fehlerrate. Einsatz: Auslesen von Passwörtern, Zugriff auf andere Dockerimages, Nachweis schwierig. Gegenmassnahmen: Kernel page-table isolation «KAISER»: verschiedene Page Tables für Kernel

Spectre: Gleiches Ziel, verwendet jedoch Branch Prediction mit spekulativer Ausführung. Branch Prediction wird nicht per Prozess unterschieden. Alle Prozesse, die auf dem selben Prozessor laufen, verwenden die selben Vorbersagen. Ein Angreifer kann damit den Branch Predictor für einen anderen Prozess «trainieren». Der Opfer-Prozess muss zur Kooperation «g indem im verworfenen Branch auf Speicher zugegriffen wird. Nicht leicht zu fassen, aber auch nicht leicht zu implementieren.

Seite 2

BSys2 | FS24 | Nina Grässli & Jannis Tschan