Betriebssysteme 2 | BSys2

7usammenfassung

int v = *px: // *px = Wert einer int-Adresse, v = 5, * = Dereferenzoper

4096	2	048	1024	512	2	56	128	64	32	16	8	4	1	2	1
2^{12}	21	1	2^{10}	29	2	3	2^{7}	2^{6}	2^5	2^{4}	2 ³	22	2	21	2^{0}
10 00	h 8	00 _h	4 00 _h	2 00	h 1	00 _h	80 _h	40 _h	20 _h	10 _h	8 _h	4,	2	2 _h	1 _h
1'04	8′57	6	65′53	В	4'	096		256		\rightarrow	16		1		
16^{5}			16^{4}		16	3		16^{2}			16^{1}		16	0	
1000	00 _h		01 00 0	10 _h	00	1000		000	100 _h		00 00	10 _h	00	00 01	h
		. '													
0	1	2	3	4	5	6	7	8	9	\boldsymbol{A}	B	C	D	E	F
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
8888	0001	0016	0011	0100	0101	8118	0111	1000	1001	1010	1011	1100	1101	1110	111

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 Me Instruktionen ausführen, Ring 3)

Kornels: Microkernel (nur kritische Teile laufen im Kernel-Manel Manelithisch (meiste OS weniner Werbrel weniger 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 en, 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 exten Sollte nur über untenstehende Funktionen manipuliert werden. Werden implizit bereitgestellt. nützlich für Informationen, die bei jedem Aufruf gleich sind.

- Abfragen einer Umgebungsvariable: char *value = getenv("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 Konflaurationsinformationen sollten über Dateien übermittelt werden.

2. DATEISYSTEM API

Applikationen dürfen nie annehmen, dass Daten gültig sind

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

Pfade: Absolut (beginnt mit /), Relativ (beginnt nicht mit /), Kanonisch (Absolut, ohne «..» und «...», real path (1)

- NAME MAX: Maximale Länge eines Dateinamens (exklusive terminierender Null)
- PATH_MAX: Maximale Länge eines Pfads (inklusive terminierender Null) (beinhalt. 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)
- // Gibt Arbeitsverzeichnis aus
 int main (int argc, char ** argv) { char *wd = malloc(PATH_MAX);

getcwd(wd, PATH_MAX); printf("Current WD is %s", wd); free(wd); return 8; } Zugriffsrechte: Je 3 Permission-Bits für Owner, Gruppe und andere Benutzer. Bits sind: read, write,

everyte: n=4 w=2 v=1 Reisniel: 8748 order nwy n== === (Owner but alle Berbte Gr andere haben keine Rechte). POSIX: S_IRWXU = 0700, S_IWUSR = 0200, S_IRGRP = 0040, S_IXOTH = 0001. Worden mit I verknünft

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 FTI END = 8: standard innut. STDOUT FILENO = 1: standard output. STDERR FILENO = 2: standard erro

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,
- 0 RDWR: lesen und schreiben
- O_CREAT: Erzeuge Datei, wenn sie nicht existiert,
- 0_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 Dateien 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);

ng +/ } /* read data: +/ close(fd):

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 statische Bibliotheken) zu Executables oder dynamischen Bibliotheken. Loader lädt Executables und eventuelle dynamische Bibliotheken dieser in den Hauptspeicher

char spath[PATH_MAX]; // source path

char dpath[PATH_MAX]; // destination path

ssize_t read_bytes = read(src, buf, N);

SEEK END); gibt die Grösse der Datei zurück.

der zusätzliche Parameter offset verwendet.

2.1.1. Unterschiede Windows und POSIX

2.2. C STREAM API

File-Position-Indicator.

2 2 1 Dateiende und Fehler

virtueller Adressraum zugeordnet.

Stream zurück

3 PROZESSE

3.1 PROZESS-API

Prozessarunnen-IDI

Suche über PATH

3.1.1. Zombie- & Orphan-Prozesse

wait() in einer Endlosschleife aufruft.

4. PROGRAMME UND BIBLIOTHEKEN

träger/Partition, andere File-Handling-Funktionen.

kopiert werden, sondern nur über von C-API erzeugte Pointer

Nach API-Umwandlung vorherige nicht mehr verwenden.

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

int fgetc(FILE *stream): Liest das nächste Byte und erhöht FPI um 1.

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

en(spath, O_RDONLY);

int dst = open(dpath, 0_WRONLY | 0_CREAT, S_IRWXU);

write(dst, buf, read bytes): // if file closed early, use return value

ssize t pread/pwrite(int fd. void * buffer, size t n. off t offset);

off t lseek(int fd. off t offset, int origin); Springen in einer Datei, Verschieht 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,

Lesen und Schreiben ohne Offsetänderung. Wie nead bzw. wnite. Statt des Offsets von fid wird

Bestandteile von Pfaden werden durch Backslash (\) getrennt, ein Wurzelverzeichnis pro Daten-

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

Streams: FILE * enthält Informationen über einen Stream. Soll nicht direkt verwendet oder

FILE * fopen(char const *path, char const *mode): Öffnen eine Datei. Erzeugt FILE-Objekt

"a": (in neue oder bestehende Datei anfügen), "r+: (Datei lesen & schreiben), "w+" (neue oder geleerte bestehende

Datei lesen & überschreiben), "a+" (neue oder bestehende Datei lesen & an Datei anfügen). Gibt Pointer auf

erzeugtes FILE-Objekt zurück oder O bei Fehler. FILE * fdopen(int fd, char const * mode) ist

gleich, aber statt Pfad wird direkt der FD übergeben, int fileno (FILE *stream) gibt FD zurück.

auf, schliesst den Stream, entfernt fille aus Speicher und giht Olzurück wenn OK, andernfalls FOF.

fgets(char *buf, int n, FILE *stream) liest bis zu n-1 Zeichen aus stream

int fputc(int c, FILE *stream): Schreibt c in eine Datei. int fputs(char *s, FILE *stream)

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

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

Ein Prozess umfasst das Abbild eines Programms im Hauptspeicher (text section), die globaler

Variablen des Programms (data section). Speicher für den Heap und Speicher für den Stack, Der

Heap wird für globale Variablen genutzt und der Stack für variablen die nur während dem Funkti-

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, Daten zur Synchronisation, Security-Informationen etc.

(context save): Register, Flags, Instruction Pointer, MMU-Konfiguration, Interrupt-Handler über-

pid t fork(void) erzeugt exakte Kopie (C) als Kind des Prozesses (P), mit eigener Prozess-ID

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 ID = alle C mit der gleichen

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

exec()-Funktionen: Jede dayon ersetzt im gerade laufenden Prozess das Programmimage durch

als Liste

execl()

execlp()

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 hei Beendung von C in einem dauerhaften Zombie

resultiert. Wenn P beendet wird, werden deshalb alle C an Prozess mit pid=1 übertragen, der

int atexit (void (*function)(void)); Registriert Funktionen für Aufräumarbeiten vor Ende.

C-Ouelle → Präprozessor → Bereiniate C-Ouelle → Compiler → Assembler-Datei →

Präprozessor: Die Ausgabe des Präprozessors ist eine reine C-Datei (Translation-Unit) ohne Ma

kros Kommentare oder Prängozessor-Direktiven Linker: Der Linker verknünft Obiekt-Dateien (und

Sekunden ungefähr verstrichen ist. Gibt vom Schlaf noch vorhandene Sekunden zurück.

pid t getpid()/getppid() geben die (Parent-)Prozess-ID zurück.

Assembler → Objekt-Datei → Linker → Executable

ed int sleep (unsigned int seconds): unterbricht Ausführung, bis eine Anzahl

als Array

execve()

execv()

execvp()

Interrupts: Kontext des aktuellen Prozesses muss im dazugehörigen PCB gespeichert werden

schreibt den Kontext. Anschliessend wird Kontext aus PCB wiederhergestellt (context restore).

Prozess-Erstellung: Das OS erzeugt den Prozess und lädt das Programm in den Prozess.

(> 0). Die Funktion führt in beiden Prozessen den Code an derselben Stelle fort.

old spawn_worker (...) {

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

ein anderes. Programmargumente müssen spezifiziert werden. (... Lals Liste... v als Arrayi

/oid exit(int code): Beendet das Programm und gibt code zurück.

pid t wait(int *status); unterbricht Prozess, bis Child beendet wurde

Unter POSIX getrennt, unter Windows eine einzige Funktion.

oss-Hiorarchio: Raumstruktur startet hei Prozess 1

for (int i = 0; i < n; ++i) { spawn_worker(...); }

Angabe des Pfads | mit neuem Environment | execle()

int ungetc (int c. FILE *stream): Lesen rückgängig machen, Nutzt Unget-Stack.

int_feof(FTLE_*stream) gibt 0 zurück, wenn Dateiende noch nicht erreicht wurde

pron(FILE * stream) gibt 0 zurück, wenn kein Fehler auftrat

fclose(FILE *file): Schliesst eine Datei. Ruft fflush() (schreibt Inhalt aus Speicher in die Datei

für Datei an path. Flags für mode: "r" (Datei lesen), "w" (in neue oder bestehende geleerte Datei schreiben),

4.1. FLF (EXECUTABLE AND LINKING FORMAT)

Verschmilzt Sektionen und erzeugt auführbares Executable.

Binär-Format, das Kompilate spezifiziert. Besteht aus Linking View (wichtig 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), Segmente (execution view), ion Header Table (linking view), Sektionen (linking view)

4.2. SEGMENTE UND SEKTIONEN

mente und Sektionen sind eine andere Einteilung für die gleichen Speicherbereiche. View des lers sind die Segmente, view des Compilers die Sektionen. Laden Daten in den Speicher und definieren «gleichartige» Daten. Der Linker vermittelt zwischen beiden Views.

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

ent/Program Header Table und Seamente: Tabelle mit n Einträgen, ieder Eintrag (ie 32 Byte) beschreibt ein Segment (Tva und Flaas, Offset und Grösse, virtuelle Adresse und chiedlich 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 and Sektionen: Tabelle mit m Finträgen (± n), leder Fintrag (ir 40 Byte) beschreibt eine Sektion (Name, Section-Typ, Flags, Offset und Grösse, ...). Werden vom Linker verwendet:

String-Tabelle: Bereich in der Datei, der nacheinander null-terminierte Strings enthält. Strings erden relativ zum Beginn der Tabelle referenziert.

Symbole & Symboltabelle: Die Symboltabelle enthält ieweils einen Eintrag ie Symbol (16 Byte: 4B

4.3 RIBLIOTHEKEN

Statische Bibliotheken: Archive von Obiekt-Dateien, Name: Lib<name>, a, referenziert wird nur <name>. Linker behandelt statische Bibliotheken wie mehrere Objekt-Dateien. Ursprünglich gab es nur statische Bibliotheken (Einfach zu imp

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

4.4. POSIX SHARED OBJECTS API

void * dlopen (char * filename, int mode): öffnet eine dynamische Bibliothek und gibt ein Handle darauf zurück. mode ist einer der folgenden Werte:

- RTLD_NOW: Alle Symbole werden beim Laden gebunden
- RTLD_LAZY: Symbole werden bei Bedarf gebunden
- RTLD GLOBAL: Symbole können beim Binden anderer Obiektdateien verwendet werden
- RTLD LOCAL: Symbole werden nicht für andere Obiektdateien verwendet

void * dlsvm (void * handle, char * name); gibt die Adresse des Symbols name aus der mit handle bezeichneten Bibliothek zurück. Keine Typinformationen (Variabel? Funktion Wenn name nicht gefunden werden kann, wird ein nullpointer zurückgegeben. Kann zu undefined

behaviour führen address of a function with a int param and int return type

typedef bool (*func_t)(bool);
handle = dlopen("licoollib.so", RTLD_NOW); // open library

func_t function = dlsym(handle, "myboolfunction"); // write myboolfunction addr L *b = dlsym(handle, "mycoolbool"); // get address of "mycoolboo

(*function)(*b): // call "myboolfunction" with "mycoolbool" as parameter 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 ver wendet den Linker-Namen, der Loader verwendet den SO-Namen.

- Linker-Name: lih + Rihliotheksname + .sn (z 8 lihmvlih sa).
- SO-Name: Linker-Name + . + Versionsnummer (z.B. libmylib.so.2)
- Real-Name: S0-name + . + Unterversionsnummer (z.B. libmylib.so.2.1)

Shared Objects: Nahezu alle Executeables benötigen zwei Shared Objects: Libc. so: Standard C library, ld-linux.so: ELF Shared Object loader (Läde Shared Objects und rekursiv alle Dependencies). Implementierung dynamischer Ribliotheken: Müssen verschiehhar sein, mehrere müssen in den gleichen Prozess geladen werden. Die Aufgabe des Linkers wird in den Loader bzw. Dynamic Linker prschohon // and Time Relocation)

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 denselhen Frame im RAM gemannt. Benötigt Position Relative Moves via Relative Calls: Mittels Hilfsfunktion wird Rücksprungadresse in Register abge legt, somit kann relativ dazu gearbeitet 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 unghhängige Annlikationen Nachteile: Realisierung naralleler Ahläufe innerhalb de Applikation ist aufwändig. Overhead zu gross falls nur kürzere Teilaktivitäten, gemeinsame Resrcennutzuna ist erschwert.

Threads: parallel 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 AMDAHIS REGEL

Nur bestimmte Teile eines Algorithmus können parallelisiert werden. T Ausführungszeit, wenn komplett seriell durchgeführt (Im Bild: $T = T_0 + T_1 + T_2 + T_3 + T_4 + T_4$)

- Anzahl der Prozessoren
- T' Ausführungszeit, wenn maximal parallelisiert gesuchte Grösse
- T_a Ausführungszeit für den Anteil, der seriell ausgeführt werden muss (Im Bild: $T_a = T_0 + T_2 + T_3$) $T-T_s$ Ausführungszeit für den Anteil, der parallel ausgeführt werden kann (Im Bild: T_1+T_3)
- $(T-T_{-})/n$ Parallel-Anteil verteilt auf alle n Pro zessoren. (Im Bild: $(T_1 + T_2)/n$) $T_s + \frac{T - T_s}{n}$ Serieller Teil + Paralleler Teil = T'

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



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

$$f \le \frac{T}{T_{s} + \frac{T - T_{s}}{s}} = \frac{T}{s \cdot T + \frac{T - s}{s}T} = \frac{T}{s \cdot T + \frac{1 - s}{s} \cdot T} \implies f \le \frac{1}{s + \frac{1 - s}{s}}$$

- Abschätzung einer oberen Schranke für den maximalen Geschwindigkeitsgewinn
- Nur wenn alles parallelisierbar ist, ist der Speedup

- proportional und maximal f(0, n) = n- Sonst ist der Speedup mit höherer Prozessor-Anzahl
- immer geringer (Kurve flacht ab) - f(1, n); rein seriell

Grenzwert

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

5.2. POSIX THREAD API

pthread_t * thread_id, pthread_attr_t const *attributes,

void * (*start_function) (void *), void *argument)

erzeugt einen Thread, die ID des neuen Threads wird im Out-Parameter thread_id zurückgegeben. attributes ist ein opakes Objekt, mit dem z.B. die Stack-Grösse spezifiziert werden kann Die erste auszuführende Instruktion ist die Funktion in start function, argument ist ein Pointer auf eine Datenstruktur auf dem Heap für die Argumente für start function



Thread-Attribute

pthread_attr_t attr; // Variabel enstellen
pthread_attr_init (&attr); // Variabel initialisieren
pthread_attr_setstacksize (&attr, 1 << 16); // 64kb Stackgrösse</pre> pthread_create (..., &attr, ...); // Thread erstellen pthread_attr_destroy (&attr); // Attribute löschen

Lebensdauer: Lebt solange, bis er aus der Funktion start function zurückspringt, ei pthread_exit oder ein anderer Thread pthread_cancel aufruft oder sein Prozess beendet wird

void athread exit (void treturn value): Reendet den Thread und gibt den neturn value zurück. Das ist äquivalent zum Rücksprung aus stant_function mit dem Rückgabewert. int pthread_cancel (pthread_t thread_id): Sendet eine Anforderung, dass der Thread mit thread, id beendet werden soll. Die Funktion wartet nicht, dass der Thread tatsächlich beende wurde. Der Rückgabewert ist 0, wenn der Thread existiert, bzw. ESRCH (error_search), wenn nicht. int othread detach (othread t thread id): Entfernt den Speicher, den ein Thread belegt hat.

falls dieser bereits beendet wurde. Beendet den Thread aber nicht. (Erstellt Doemon Thread) 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-Parameter 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: Auslesen des Keys aus der globalen Variable, Auslesen / Schreiben des Werts anhand des Keys.

int pthread key create(pthread key t *key, void (*destructor) (void*)); Erzeugteinen neuen Key im Out-Parameter key. Opake Datenstruktur. Am Thread-Ende Call auf destructor. int othread key delete(othread key t key): Entfernt den Key und die entsprechenden Values aus allen Threads. Der Key darf nach diesem Aufruf nicht mehr verwendet werden. Sollte erst aufgerufen werden, wenn alle dazugehörende Threads beendet sind. int pthread setspecific(pthread key t key, const void * value)

roid * pthread getspecific(pthread key t key) schreibt bzw. liest den Wert, der mit dem Key in diesem Thread assoziiert ist. Oft als Pointer auf einen Speicherhereich verwendet

// Main und Thread void *thread_function (void *) { setu_up_error(); if (force error () = -1) { print error (): }

t main (int argc, char **argv) { pthread_key_create (&error, NULL); // Key erzeugen pthread t tid: pthread create (&tid, NULL, &thread function, NULL): // Threads erzeuger pthread_join (tid, NULL);

SCHEDULING

Auf einem Prozessor läuft zu einem Zeitnunk immer höchstens ein Thread. Es eibt folgende

- Running (der Thread, der gerade läuft) - Ready (Threads die laufen können, es aber gerade nicht
- Waiting: (Threads, die auf ein Ereignis warten, können nicht direkt in den Running State wechseln.

Übergänge von einem Status zum anderen werden immer vom OS vorgenommen. Dieser Teil vom OS hoiset Schodulo Arten von Threads: I/O-lastia (Wenia rechnen, viel I/O-Geräte-Kommunikation), Prozessor-lastia (Viel rech-

Arten der Nebenläufigkeit: Kooperativ (Threads entscheiden selbst über Abgabe des Prozessors), Präemptiv einem Thread der Prozerror 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 (Totsächliche Gleichzeitigkeit, n Prozessoren für n Threads), Quasiparallel (n Threads auf < n Prozessoren abwechselnd), Nebenläufig (Überbegriff für parallel oder quasiparallel



Powerdown-Modus: Wenn kein Thread laufbereit ist, schaltet das OS den Prozessor in Standby and wird durch Interrunt wieder geweckt

Bursts: Prozessor-Burst (Thread beleat The Prozessor volll. I/O-Burst (Thread beleat 10 20 30 40 50 60 70 80 90 Zeit [mc Prozessor nicht). Jeder Thread kann als Abfolge von Prozessor-Bursts und I/O-Bursts hetrachtet werden

6.1 SCHEDULING-STRATEGIEN

f(0.5, n)

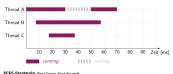
Anforderungen an einen Scheduler können vielfältig sein, Geschlossene Systeme (Hersteller kennt Anwendungen und ihre Beziehungen) VS. Offene Systeme (Hersteller muss von typischen Anwendungen ausgehen,

Anwendungssicht, Minimierung von: Durchlaufzeit (Zeit vom Starten des Threads bis zu seinem Ende), Antwortzeit (Zeit vom Empfang eines Requests bis die Antwort zur Verfügung steht), Wartezeit (Zeit, die ein Thread

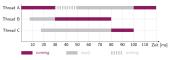
Aus Systemsicht, Maximierung von: Durchsatz (Anzahl Threads, die pro Intervall bearbeitet werden), Prondung (Prozentsatz der Verwendung des Prozessors gegenüber der Nic

Latenz ist die durchschnittliche Zeit zwischen Auftreten und Verarbeiten eines Freignisses. 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 jedoch zu mehr Thread-Wechsel und somit zu mehr Overhead führt. Die Utilization nimmt also ab. wenn die Antwortzeit verrinaert wird.

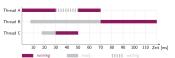
Idealfall: Parallele Ausführung (Dient als Idealisierte Schranke, berret gebt nicht)



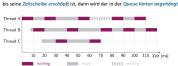
Nicht präemptiv: Threads geben den Prozessor nur ab, wenn sie auf waiting wechseln oder sich



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



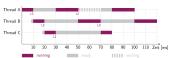
Round-Robin: Zeitscheibe von etwa 10 bis 100ms. FCFS, aber ein Thread kann nur solange laufen,



Prioritäten-basiert: Jeder Thread erhält eine Nummer, seine Priorität. Threads mit höherer Priorität werden vor Threads mit niedriger Priorität ausgewählt. Threads mit gleicher Priorität werden nach ECES ausgewählt. Prioritäten je nach OS unterschiedlich

Starvation: Thread mit niedriger Priorität wird immer übergangen und kann nie laufen. Abhilfe z B. mit Aging: in bestimmten Abständen wird die Priorität um 1 erhöht.

Multi-Level Scheduling: Threads werden in Level aufgeteilt (Priorität, Prozesstyp, Hinter-/Vordergrund) jedes Level hat eigene Ready-Queue und kann individuell geschedulet werden. (zB. Timeslice/C Multi-Level Scheduling mit Feedback: Erschöpft 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 länger am Stück laufen.



6.2. PRIORITÄTEN IN POSIX

running

ready

suspend

Nice-Wert: Jeder Prozess hat einen Nice-Wert von -20 (soll beworzugt 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 int setpriority (int which, id_t who, int prio): setzt den Nice-Wert.

who: ID des Prozesses, der Gruppe oder des Users)

- which: PRIO PROCESS, PRIO PGRP oder PRIO USER

Funktionen ohne attr bevor Thread gestartet wird:

- ead_getschedparam(pthread_t thread, int * policy, struct sched_param - int pthread_setschedparam(pthread_t thread, int policy, const struct sched_param

pthread attr t a: pthread attr init (&a): // initialize attributes

pthread_attr_getschedparam (&a, &p); // read parameter

// set p.sched_priority here... (code cut for brevity)

pthread_attr_setschedparam (&a, &p); // write modified parameters pthread_create (&id, &a, thread_function, argument);
pthread_attr_destroy (&a); // destroy attributes

MUTEXE UND SEMAPHORE

Jeder Thread hat seinen eigenen Instruction-Pointer und Stack-Pointer. Wenn Ergebnisse 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 unbedingt

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

führt werden. Achtung: Selbst einzelne Assembly-Instruktionen nicht immer atomar! Anforderungen an Synchronisations-Mechanismen: Gegenseitiger Ausschluss (Nur ein Thread darf in Critical Section sein). Fortschrift (Entscheidung wer in die Critical Section darf muss in endlicher Zeit getraffen 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: compare_and_swap (int *a, int expected, int new_a) {int 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 erhöht).

int sem_init (sem_t *sem, int pshared, unsigned int value): Initialisiert den Semaphor, typischerweise als globale Variable. pshared = 1: Verwendung über mehrere Prozesse: sen_t sem; int main (int argc, char ** argv) { sem_init (&sem, 0, 4); } oder als 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 hat

```
semaphore free = n: semaphore used = 0:
while (1) {
                                                  while (1) {
                                                                  falls Producer zu lan
 WAIT (free); // Hat es Platz in Queue? WAIT (used); //
produce_item (&buffer[w], ...); consume (&buffer
                                                     WAIT (used); // Hat es Elem. in Queue?
consume (&buffer[r]);
                                                                                ent weniger in O
 POST (used): // 1 Flu
                              ent make in Augus PAST (free): /
                                                    r = (r+1) % BUFFER_SIZE;
 w = (w+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)

```
pthread_mutex_t mutex; // global
int main() {
    pthread_mutex_t mutex; // global
                                                   while (running) {
   pthread_mutex_init (&mutex, 8);
 // run threads & wait for them to finish __nthread mutex lock (&mutex):
       nread_mutex_destroy (&mutex); }
                                                     pthread_mutex_unlock (&mutex); ...}}
```

Priority Inversion: Ein hoch-priorisierter Thread C wartet auf eine Ressource, die von einem niedriger priorisierten Thread A gehalten wird. Ein Thread mit Prioriät zwischen diesen beiden Threads erhält den Prozessor. Kann mit **Priority Inheritance** gelöst werden: Die Priorität von A ${\rm wird\ tempor \ddot{a}r\ auf\ die\ Priorit\ddot{a}t\ von\ } \ C\ {\rm gesetzt,\ damit\ der\ Mutex\ schnell\ wieder\ freigegeben\ wird.}$

8. SIGNALE, PIPES UND SOCKETS

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

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

Programmfehler-Signale: SIGFPE (Fehler in arithmetischen Operation), SIGILL (Ungültige Instruktion), SIGSEGV (Ungültiger Speicherzugriff), SIGSYS (Ungültiger Systemaufruf)
Prozesse abbrechen: SIGTERM (Normale Anfrage an den Prozess, sich zu beenden), SIGINT (No

Aufforderung an den Prozess, sich zu beenden), SIGOUIT (Wie SIGINT, aber anormale Terminierung), SIGABRT (Wie SIGOUIT, 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

STRTSTP, aber kann nicht ignoriert oder abgefangen werden). STRCONT (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 $\neq 0$. (Finene Signal-Handler definiert via signation struct) mask: Blockierte Signale während Ausführung, bearbeitet nur durch sig*set()-Funktionen: sigemptyset, sigfillset, sigaddset, sigdelset, sigismember)

8.2. PIPES

Eine geöffnete Datei entspricht einem Eintrag in der File-Descriptor-Tabelle (FDT) im Prozess. Zugriff über File-API (open, close, read, write, ...). Das OS speichert je Eintrag der Prozess-FDT einen Verweis auf die alobale FDT. Bei fonk() wird die FDT auch kopiert.

int dup (int source_fd); int dup2 (int source_fd, int destination_fd): Duplizieren 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 dun2(fd. 1): // dunlicate fd for log.txt as standard output // e.g. load new image with exec*, fd's re
} else { /* parent */ close (fd); }

Eine Pipe ist eine «Datei» (Eine Datei muss nur open, close etc. unterstützen) im Hauptspeicher, 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.

int fd[2]; // 0 = read, 1 = write // don't use write end pipe (fd); int id = fork(): char buffer [BSIZE]:

Pipe lebt nur so lange, wie mind. ein Ende int n = read (fd[0], buffer, BSIZE); geöffnet ist. Alle Read-Ends geschlossen ightarrow } else { geöffnet ist. Alle Read-Ends geschlossen -> } else { // Parent thread
SIGPIPE an Write-End. Mehrere Writes können
char * text = "1 < 3 segfaults";</pre> zusammengefasst werden. Lesen mehrere Prozesse dieselbe Pipe, ist unklar, wer die Daten 1

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, ie nach System auch über Reboots hinweg. Muss explizit mit unlink gelöscht werden.

write (fd[1], text, strlen(text) + 1):

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; (IP: IP-Adresse & Portnr.)

t 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 (SOCK_DGRAM, SOCK_STREAM), protocol (System-spezifisch, 0 = Default-Protocol)

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

von Verbindungsanfragen von Clients), accept (Erzeugen einer Verbindung auf Anfrage von Client), pecy / pead (Emafangen von Daten, 0 - ∞ mail, Send / write (Senden von Daten, 0 - ∞ mail, close (Schliessen der Verbindung)

struct sockaddr_in ip_addr;
ip addr.sin port = htons (443): // default HTTPS por net_pton (AF_INET, "192.168.0.1", &ip_addr.sin_addr.s_addr);

/ port in memory: 0x01 0x88 // addr in memory: 0x00 0x88 0x80 0x81

htons() konvertiert 16 Bit von Host-Byte-order (LE) zu Network-Byte-Order (BE), htonl() 32 Bit. ntohs() und ntohl() sind Gegenstücke. inet_pton() konvertiert protokoll-spezifische Adresse von String zu Network-BO, inet_nton() ist das Gegenstück (network-to-presentation)

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 von

Verbindungen». Erzeugt eine Warteschlange, die so viele Verbindungsanfragen aufnehmen kann, wie backlog angibt. int accept (int socket, struct sockaddr *remote_address, socklen_t address_len): Wartet, bis 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 geschrie-

8.3.1. Typisches Muster für Server

t server_fd = socket (...); bind (server_fd, ...); listen (server_fd, ...); int client_fd = accept (server_fd, 0, 0); delegate to worker thread (client fd): // will call close(client fd)

ben. Der neue Socket kann keine weiteren Verbindungen annehmen, der bestehende schon.

send (fd, buf, len, 8) = 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.

ose (int socket): Schliesst den Socket für den aufrufenden Prozess. Hat ein andere 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-Zugriffe mehr), SHUT_RDWR (Keine Lese- oder Schreib-Zugriffe mehr)

9. MESSAGE PASSING UND SHARED MEMORY

Prozesse sind voneinander isoliert, müssen jedoch trotzdem miteinander interagieren. Message Passing ist ein Mechanismus mit zwei Operationen: Send (Koniert die Nochricht aus dem Prozess: send (message)), Receive: (Kopiert die Nachricht in den Prozess: receive (message)). Dabei können Implemen tierungen nach verschiedenen Kriterien unterschieden werden (Feste oder Variable Nachrichtenaröss) 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 Empfänger kennen. Es gibt symmetrische direkte Kommunikation (Empfänger muss Sender auch kennen) und metrische direkte Kommunikation (Empfänger muss Sender nicht kennen)

Indirekte Kommunikation: Prozess sendet Nachricht an Mailboxen, 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 Nachricht vom Empfänger emp fangen wurde), Nicht-blockierendes Senden (Sender sendet Nachricht und fährt sofort weiter), Blockierendes Empfangen (Empfänger 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 beide Seiten ihren Aufruf getätigt haben. Impliziter Synchronisationsmechanismus

message msg; message msg; onen(f): onen(fi) while(1) { while(1) { /e(Q, &msg); // blocked until rec send(Q, &msg); // blocked until sent consume_next(&msg);

blackiert wenn Overe vall ist 1. Unbeschränkte (Beliebig viele Nachrichten, Sender blackiert niel.)

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

9.0.1. 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): Öffnet 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 un close (nad t queue): Schliesst die Queue mit dem Descriptor queue für diesen Prozess Lnt 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 mg send (mgd t gueue, 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.

9.1. SHARED MEMORY

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

9 1 1 POSIX API

Das OS benötigt eine «Datei» S. das Informationen über den gemeinsamen Speicher verwaltet

int fd = shm_open ("/mysharedmemory", 0_RDWR | 0_CREAT, S_IRUSR | S_IWUSR): 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 bleiht aber im System

int shm_unlink (const char * name): Löscht das Shared Memory mit dem name. (bleibt vorhander solange noch von Prozess geöffnet)

int munmap (void *address, size_t length): Entfernt das Mapping.

void * address = mmap(// maps shared memory into virt. address space of process size t length (same as used in ftruncate) PROT_READ | PROT_WRITE, // int protection (never use execute) MAP SHARED. // off_t offset (start map from first byte)

9.2 VERGLEICH MESSAGE-PASSING LIND SHARED MEMORY

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

9.3. VERGLEICH MESSAGE-QUEUES UND PIPES

Message-Queues	Pipes			
- bidirektional	- unidirektional			
- Daten sind in einzelnen Messages organisiert				
 beliebiger Zugriff 	- 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 Hexzahl = Zeile, zweite Hexzahl = Spalte, d.h. 41h = A).

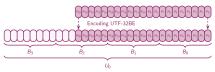
	0	1	2	3	4	5	6	7	8	9	A	В	C	D	Ε	F
0	NUL				EOT	ENG		BEL	88	TAB	LF		FF	CR	80	
1	DEE				D04	NAK		ETB	CAN	EH	58				RS	
2	ш	!	"	#	\$	%	&	,	()	*	+	,	-		/
3	0	1	2	3	4	5	6	7	8	9	1	;	<	=	>	?
4	0	A	В	С	D	Е	F	G	Н	I	J	K	L	М	N	0
5	P	Q	R	S	Т	U	V	W	Х	Y	Z	[1]	^	_
6	¢	a	b	С	d	е	f	g	h	i	j	k	1	m	n	0
7	n	a	r	s	t.	11	v	w	x	v	7.	{		}	~	

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 Speicherplot $= i \cdot \text{tes Bit des unkodierten CPs, } U_i = i \cdot \text{tes Code-Unit des kodierten CPs, } B_i = i \cdot \text{tes Byte des kodierten CPs UTF baut}$ auf ASCII auf und ist ein superset davon.

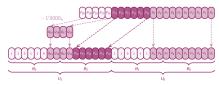
10.2. UTF-32

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_a bis P_a in B_a kopiert usw. Wird häufig intern in Programmen verwendet. Obere 11 Bits oft «zweckentfremdet».



10.3 LITE-16

Jede CU umfasst 16 Bit, ein CP benötigt 1 oder 2 CUs. Encoding muss Endianness berücksichtigen Die 2 CUs werden Surrogate Pair genannt, Un: high surrogate, U1: low surrogate. Bei 2 Bytes (1 CU) wird direkt gemannt und vorne mit Nullen aufgefüllt. Bei 4 Bytes sind D8 00. bis DE FF. /Bits 17-211 wegen dem Separator ungültig und müssen «umgerechnet» werde



Encoding von U+10'437 (♥) 00 0100 0001 00 0011 0111:

1. Code-Point P minus 1 00 00₆ rechnen und in Binär umwandlen

 $P = 1.04.37_h$, $Q = 1.04.37_h - 1.00.00_h = 04.37_h = 00.0000.0001.00.0011.0111_h$ 2. Obere & untere 10 Bits in Hex umwandlen

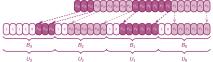
3. Oberer Wert mit D8 00h und unterer Wert mit DC 00h addieren, um Code-Units zu erhalter $U_1 = 00\,01_{\rm h} + {\rm D8}\,00_{\rm h} = {\rm D8}\,01_{\rm h}, \quad U_2 = {\rm O1}\,37_{\rm h} + {\rm DC}\,00_{\rm h} = {\rm DD}\,37_{\rm h}$ 4. Zu BE/LE zusammensetzen

$BE = 08010037_h$, $LE = 01083700_h$ 10.4 LITE-8

Jede CU umfasst 8 Bit, ein CP benötigt 1 bis 4 CUs. Encoding muss Endianness nicht berücksichtigen. 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

Code-Point in	U_3	U_2	U_1	U_0	signifikant
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

 $-\ddot{a}$: $P = E4_h = 0.0011 10.0110$

 $\Rightarrow P_{10} P_{2} = 0.0011_{10} = 0.36_{10} P_{2} P_{3} = 10.0100_{10} = 246_{10}$

 $\Rightarrow U_1 = CO_h (= 11000000_h) + OO_h = CO_h, U_0 = 80_h (= 10000000_h) + 24_h = A4_h$ $\Rightarrow \bar{a} = C3 A4_h$

 $- \check{a}$: P = 1EB7 = 0001 11 1010 11 0111.

 $\Rightarrow P_{15}...P_{12} = 01_b$, $P_{1}...P_{6} = 3A_b$, $P_{5}...P_{9} = 37_b$

 $\Rightarrow U_2 = \mathrm{EO_h} \; (= 1110\,0000_b) + \mathrm{O1_h} = \mathrm{E1_h}, \quad U_1 = 80_b + \frac{3\mathrm{A_h}}{20_b} = \mathrm{BA_h}, \; U_0 = 80_b + 37_b = \mathrm{B7_h}$ $\Rightarrow \breve{a} = \underline{\text{E1 BA B7}}_{h}$

10.5. ENCODING-BEISPIELE

Zeichen	Code-Point	UTF-32BE	UTF-32LE	UTF-8	UTF-16BE	UTF-16LE
А	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
ã	1E B7 _h	00 00 1EB7 _h	B7 1E 00 00 _h	E1 BA B7 _h	1EB7 _h	B7 1E _h
J/	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

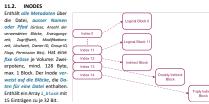
Bei LE / RE werden our die Zeichen innerhalb eines Code-Points vertauscht, nicht die Code-Points an sich

11 FYT2-DATETSYSTEM

Partition (Ein Teil eines Datenträgers, wird selbst wie ein Datenträger behandelt.), Volume (Ein Datenträger oder eine Partition davon.), Sektor (Kleinste logische Untereinheit eines Volumes. Daten werden als Sektoren transferiert. Gröss ist von HW definiert. Enthält Header, Daten und Error-Correction-Codes 1. Format (Layout der Jonischen Strukturen auf

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 (Blocknummern Anfang der Datei aus gesehen, wenn Datei eine ununterbrochene Abfolge von Blöcken wäre) und Physische Blocknummern (Tatsächliche Blocknummer auf dem Volume).



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-Bitn 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 Rytes 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 Gruppel Block $n \pm 3$ his $n \pm m \pm 2$ (Tabella offer Inodes in dieser Gruppel) Block $n \pm m \pm 3$ his 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 reiner Potenzen von 3, 5 oder 7 gehalten (Sehr hoher Wiederherstellungsgrad, aber deutlich weniger Platzverbrauch). **Gruppendeskriptor:** 32 Byte Beschreibung einer Blockgruppe. (Blocknummer des Block-Usage-Bitmaps Blocknummer des Inode-Usage-Bitmaps, Nummer des ersten Blocks der Inode-Tabelle, Anzahl freier Blöcke und Inodes in der 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 $= (32 \cdot n)/\mathrm{Sektorgr\ddot{o}sse}$ Verzeichnisse: Enthält Dateieinträge mit variabler Länge von 8 - 263 Byte (48 Inode, 28 Eintraglänge menlänge, 18 Dateityp, 0 - 2558 Dateiname aligned auf 48). Defaulteinträge: «.» und «..» Links: Es gibt Hard-Links (gleicher Inode, verschiedene Pfade: Wird von versch

und Symbolische Links (Wie eine Datei, Datei enthält Pfad anderer Datei).

11.4. VERGLEICH FAT. NTFS. FXT2

FAT	Ext2	NTFS		
Daten über die Datei – Datei ist in einem einzigen Verzeichnis	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 enthälk Namen und Link auf Datei Link zum Verzeichnis und Name sind in einem Attribut Hard-Links möglich		

12. EXT4

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

12.1. EXTENTS

Beschreiben ein Intervall physisch konsekutiver Blöcke. Ist 12 Byte gross (48 logische Blocknummer, 68 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 Extent Trees: Index-Knoten (Innerer Vector der Roums, hertebt aus Index-Eintrag und Index-Block)

Index-Eintrag (Enthält Nummer des physischen Index-Blocks und kleinste logische Blocknummer aller Kindknoten), Index-Block (Enthält eigenen Tree-Header und Referenz auf Kindkn Extent Tree Header: Benötigt ab 4 Extents, weil zusätzlicher Block, Magic Number F3 0A, (28).

Anzahl Einträge, die direkt auf den Header folgen (28), Anzahl Einträge, die maximal auf der Header folgen können (28), Tiefe des Baums (28) - (0: Einträge sind Extents. >1: Einträge sind Index Nodes)

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

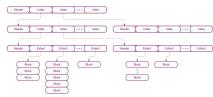
(in)direkte Addressierung	Extent-Trees
	Indexknoten: Index → (Kindblocknr, kleinste Nummer der 1. logischen Blöcke aller Kinder)
${\it indirekte Bl\"{o}cke}: {\it indirekter Block.} {\it Index} \mapsto {\it direkter Block}$	$Blottknoten$: Index \mapsto (1. logisch. Block, 1. phy. Block, Anz. Blöcke)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

Beispiel Berechnung 2MB grosse, konsekutiv gespeicherte Datei, 2KB Blöcke ab Block 2000h

2 MB = 2²¹B. 2 KB = 2¹¹B. 2²¹⁻¹¹ = 2¹⁰ = 400 Blöcke you 2000 his 23 FF 0 → 20 00s. 1 → 20 02s. Bs → 20 0Bs. Cs → 24 00s (indirekter Block) 14 00₆,0₆ → 20 0C₆, 14 00₆,1₆ → 20 0D₆, ..., 14 00₆,3 F3₆ → 23 FF

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

Extent: $1 \mapsto (0.2000_b, 400_b)$



12.2. JOURNALING

Wenn Dateisystem beim Erweitern einer Datei unterbrochen wird, kann es zu Inkonsistenze 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 sollter Journaling: Daten als Transaktion ins Journal, dann an finale Position schreiben (committing),

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. MELTDOWN

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 $\mathit{gesch\"{u}tzten Speicher}$ an Adresse a das Byte m_a zu lesen

2. die Information m. in irgendeiner Form f. zwischen

3. binäre Fragen der Form « $f_a \stackrel{\circ}{=} i$ » zu beantworten

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

13.1. PERFORMANCE-OPTIMIERUNGEN

hhΑ

Mapping des Speichers in jeden virtuellen Adressraum, Out-of-Order Execution (03E), Spekulative Ausführung.

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 henötiet, kann man herausfinden, oh 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 hetroffen. Geschwindigkeit his zu 500 KB pro Sekunde hei 0.02% Fehlerrate

Einsatz: Auslesen von Passwörtern, Zugriff auf andere Dockerimages, Nachweis schwierig. Gegenmassnahmen: Kernel page-table isolation «KAISER»: verschiedene Page Tables für Kernel-

hzw. User-Mode. Nachteil: System wieder langsam. 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 Vorhersagen. Ein Angreifer kann damit den Branch Predictor für einen anderen Prozess «trainieren». Der Opfer-Prozess muss zur Kooperation «gezwung werden, indem im verworfenen Branch auf Speicher zugegriffen wird. Nicht leicht zu fassen, aber

Partitition Root

Boot Rec Group 1 Group 2 Group x Rec. Super Group Block Inode Inode Data Bitmap Bitmap block Desc. Table Table Block 1 BI. 1 Bl. 1 Bl. 1 Bl. 214 Bl. 7974 (1 Inode Table Entry) BÌ. Inode Info (Mode Optional i block (15x4 b) Size, Times, Flags, Fields

Block

1 BI

Block

Block

All Pointers each 4 bytes Direct Block Indirect Doubly RP Pointers (12 x) (points to) Direct Block Indirect Block Pointer (4 B) Pointer (4 B)

Directory Entry (located in Data Blocks)

1 BI

inode	Record lenght	Name length	name	to next 4B)
4 B	2 B	2 B	0-255 B	0-4 B

BSys2 | FS24 | Nina Grässli & Jannis Tschan