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}	2	11	2^{10}	29	2	В	2^{7}	2^{6}	2^5	2^4	2^3	22	:	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 _h	1 _h	
1'048'576 65'536				4'	4'096			256			16			1		
16^{5}			16^{4}		16	16^{3}		16^{2}			16^{1}		16	0		
10 00 00 _h 01 00 00 _h		00	00 10 00 _h		000	00 01 00 _h		00 00 10 _h		00	00 00 01 _h					
		. '						:								
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	0016	0011	0100	0101	9119	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 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) { ... } // 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 C zeigt die Variable extern Sollte nur über untenstehende Funktionen manipuliert werden. Werden implizit bereiteestellt. 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()) - 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 0; } Zugriffsrechte: Je 3 Permission-Bits für Owner, Gruppe und andere Benutzer. Bits sind: read, write,

everyte: n=4 w=2 v=1 Reisniel: 97//9 oder nwv n== == /Owner hat alle Perhte Gr andere haben keine Rechte). POSIX: S_IRWXU = 0700, S_IWUSR = 0200, S_IRGRP = 0040, S_IXOTH = 0001. Werden 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); if (fd < A) { /* error hand ing +/ } /* 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

BSys2 | FS24 | Nina Grässli & Jannis Tschan

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

kopiert die nächsten n Byte vom buffer an den aktuellen Offset von fd

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 (ex Section Header Table (linking view), Sektionen (linking view)

4.2. SEGMENTE UND SEKTIONEN

Segmente und Sektionen sind eine andere Einteilung für die gleichen Speicherbereiche. View des Londers sind die Segmente, view des Compilers die Sektionen, 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.

Program Header Table und Segmente: Tabelle mit n Einträgen, jeder Eintrag (je 32 Byte) beschreibt ein Segment (Typ und Flags, Offset und Grösse, virtuelle Adresse und Grösse im Spe unterschiedlich 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

Streams: FILE * enthält Informationen über einen Stream. Soll nicht direkt verwendet oder 4.3 RIRI IOTHEKEN

Name 48 Wert 48 Grösse 48 Infol

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 Ribliothek, Vorteile: Entkonnelter 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 * dlsym (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 typedef int (+func t)(int):

typeoprant (*Tone_J(lint);

handle = dlopen("libsyllo,so", RTLD_NOW); // open library

func_t f = dlsym(handle, "my_function"); // write my_function addr. into a func_t

int *1 = dlsym(handle, "my_int"); // get address of "my_int"

int *1 = 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 Redarf geladen werden. Der Linker verwendet den Linker-Namen, der Loader verwendet den SO-Namen. - Linker-Name: lib + Bibliotheksname + .so (z.B. libmylib.so),

- 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 Evecuteables benötigen zwei Shared Objects: 11bc, en: Standard C library, ld-linux.so: ELF Shared Object loader (Lädt 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 verschoben (Load Time Relocation).

4.5 SHARED MEMORY

Dynamische Ribliotheken 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, Prozessor muss relative Instruktionen anbieten). Relative Moves via Relative Calls: Mittels Hilfsfunktion wird Rücksprungadresse in Register abgelegt, 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 unabhängige Applikationen. Nachteile: Realisierung paralleler Abläufe innerhalb den Applikation ist aufwändig. Overhead zu gross falls nur kürzere Teilaktivitäten, gemeinsame Res-

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 AMDAHIS REGEL

Nur bestimmte Teile eines Algorithmus können parallelisiert werden.

 $\boldsymbol{T} \quad \text{Ausführungszeit, wenn } \textit{komplett seriell} \; \text{durchgeführt (Im Bild:} \; T = T_0 + T_1 + T_2 + T_3 + T_4)$ n Anzahl der Prozessoren

T' Ausführungszeit, wenn maximal narallelisiert gesuchte Grösse

 T_s Ausführungszeit für den Anteil, der seriell ausgeführt werden muss (im Bild: $T_s = T_0 + T_2 + T_4$) T_a Ausführungszeit für den Anteil, der parallel ausgeführt werden kann (Im Bild: T₁ + T₂)

 $(T-T_{\circ})/n$ Parallel-Anteil verteilt auf alle n Prozessoren. (Im Bild: $(T_1 + T_3)/n$) $T_a + \frac{T - T_a}{T}$ Serieller Teil + Paralleler Teil = T'

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

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

erhält man f unabhängig von der Zeit:

$$f \leq \frac{T}{T_s + \frac{T - T_s}{n}} = \frac{T}{s \cdot T + \frac{T - s \cdot T}{n}} = \frac{T}{s \cdot T + \frac{1 - s}{n} \cdot T} \quad \Rightarrow \quad f \leq \frac{1}{s + \frac{1 - s}{n}}$$

5.1.1. Bedeutung

- 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 aerinaer (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 pthread_create(

othread t * thread id. othread 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 stant ifunction, argument ist ein Pointer auf eine Datenstruktur auf dem Heap für die Argumente für start_function.



Thread-Attribute

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

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 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 bei wurde. Der Rückgabewert ist O. wenn der Thread existiert, bzw. ESRCH (error search), wenn nicht. read detach (othread t thread id): Entfernt den Speicher, den ein Thread belegt hat. falls disser hereits heendet wurde Roendet 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)

TIS ist ein Mechanismus, der globale Variablen ner 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 othread key create(othread key t *key, void (*destructor) (void*)); Erzeugteiner 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 verwendet werden. Sollte erst aufgerufen werden, wenn alle dazugehörende Threads beendet sind. int pthread_setspecific(pthread_key_t key, const void * value)
void * 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 Speicherbereich verwendet



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 erzeuger

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 (Threads die laufen können, es aber gerade nicht

pthread_join (tid, NULL);

waiting - Waiting: (Threads, die auf ein Ereignis warten, könner Übergänge von einem Status zum anderen werden immer vom OS vorgenommen. Dieser Teil vom OS heisst Scheduler

Das OS registriert Threads auf ein Fr. eignis und setzt sie in den Zustand waiting». Tritt das Ereignis auf, ändert das OS den Zustand auf ready.

Ready-Queue: In der Ready-Queue (kann auch ein Tree sein) befinden sich alle Threads, die bereit sind zu laufen

Powerdown-Modus: Wenn kein Thread laufbereit ist, schaltet das OS der Prozessor in Standby und wird durch Interrunt wieder gewerkt.

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), Präempt (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 (Totsächliche Gleichzeitigkeit, n Prozes soren für n Threads). Quasiparallel (n Threads auf < n Prozessoren abwechselnd). Nebenläufia (überbegriff für

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

6.1. SCHEDULING-STRATEGIEN

ab. wenn die Antwortzeit verringert wird.

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 Minimigrung von: Durchlaufzeit (Zeit vom Storten der Threndr hir zu reinem Endel Antwortzeit (Zeit vom Empfana eines Requests bis die Antwort zur Verfügung steht). Wartezeit (Zeit, die ein Thread

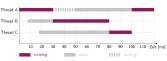
Aus Systemsicht, Maximierung von: Durchsatz (Anzahl Threads, die pro Intervall bearbeitet werden), Prowendung (Prozentsatz der Verwendung des Prozessors gegenüber der Nichtverwendung) Latenz ist die durchschnittliche Zeit zwischen Auftreten und Verarheiten eines Ereignisses. Im schlimmsten Fall tritt das Ereignis dann auf, wenn der Thread gerade vom Prozessor entfernt wurde. Um die Antwortzeit zu verringern, muss ieder Thread öfters ausgeführt werden, was iedoch zu mehr Thread-Wechsel und somit zu mehr Overhead führt. Die Utilization nimmt also

Idealfall: Parallele Ausführung (Dient als idealisierte Schranke, besser geht nicht)



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. (28. 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 länger am Stück laufen.



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

int setpriority (int which, id t who, int prio); setzt den Nice-Wert O Front B

- which PRIN PRINCESS PRIN PERP oder PRIN USER - who: ID des Prozesses, der Gruppe oder des Users)

Priorität bei Thread-Erzeugung setzen

Funktionen ohne attr bevor Thread gestartet wird:

- int pthread_getschedparam(pthread_t thread, int * policy, struct sched_param - int othread setschedoaram(othread t thread, int policy, const struct sched param

running

00

char spath[PATH_MAX]; // source path

int src = open(spath, 0_RDONLY); int dst = open(dpath, O_WRONLY | O_CREAT, S_IRWXU);

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

void spawn worker (...) {

3.1.1. Zombie- & Orphan-Prozesse

wait() in einer Endlosschleife aufruft.

4 PROGRAMME JIND RTRI TOTHEKEN

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):

Daten zur Synchronisation, Security-Informationen etc.

Unter POSIX getrennt, unter Windows eine einzige Funktion

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

Angabe des Pfads | mit neuem Environment | execle()

mit altem Environment

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

und eventuelle dynamische Ribliotheken dieser in den Hauntsneicher

Assembler → Objekt-Datei → Linker → Executable

Prozess-Hierarchie: Raumstruktur, startet hei Prozess 1.

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.

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 lacek(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 fd wird

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

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

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

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

erzeugtes FILE-Objekt zurück oder 0 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.

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

auf, schliesst den Stream, entfernt fille aus Speicher und eint Olzurück wenn OK, andernfalls FOE

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 globalen

Process Control Block (PCB): Das Betriebssystem hält Daten über ieden Prozess in ieweils 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.

(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 = iroendeinen. 0 = alle C mit der gleichen

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

C ist zwischen seinem Ende und dem Aufruf von wait() durch P ein Zombie-Prozess. Dauerhafte

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

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

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

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

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

statische Bibliotheken) zu Executables oder dynamischen Bibliotheken. Loader lädt Executables

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

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

als Liste

exect()

execlp()

als Array

execve()

execv()

execvp()

anderes. Programmargumente müssen spezifiziert werden. (.. l als Liste, .. v als Array)

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

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.

void exit(int code): Beendet das Programm und eibt code zurück.

pid_t wait(int *status): unterbricht Prozess, bis Child beendet wurde

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

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

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

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

7. 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 synchronisiert werden.

Atomare Instruktionen: Eine 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 Thre 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: 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 > 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 com init (sem t teem int nebered uncinned int value): Initialisiert den Semanhor typischerweise als alobale 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 sen destroy (sem t *sem): Entfernt Speicher, den das OS mit sem assoziiert hat semaphore free = n: semaphore used = 0:

```
while (1) {
                                                                      while (1) {
  WAIT (free); // Hat es Platz in queue? WAIT (used); // Hat es Elem. in queue? produce_item (&buffer(M), ...); consume (&buffer(M), // Element mehr in queue POST (maed): // I Element mehr in queue
  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)

```
// Rejected Initialisis
                                       // Reigniel Verwendung in Thre
            x_t mutex; // global
int main() {
                                         while (running) { .
  pthread_mutex_init (&mutex, 0);
// run threads & wait for t
                                o finish pthread_mutex_lock (&mutex);
  pthread_mutex_destroy (&mutex); }
```

Priority Inversion: Fin hock-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 Awird temporär auf die Priorität von C gesetzt, damit der Mutex schnell wieder freigegeben wird.

8 STGNALE 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 Sianal-Handler-Funktionen, Fortsetzen des Prozesses. Werden über ungültige Instruktionen oder Abbruch auf Seitens Benutzer ausgelöst. Jeder Prozess hat pro 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 Instruktion), SIGSEGV (Ungültiger Speicherzugriff), SIGSYS (Ungültiger Systemaufruf)

Prozesse abbrechen: SIGTERM (Normale Anfrage an den Prozess, sich zu beenden), SIGINT (Nachdrücklichere 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 ignoriert oder abgefangen werden), SIGCONT (Setzt den Prozess fort)

Signale von der Shell senden: kill 1234 5678 sendet SIGTERM an Prozesse 1234 und 5678 nt 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)

pipe (fd):

Fine geöffnete Datei entspricht einem Fintrag 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 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 fd[2]: // 0 = read. 1 = write

```
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*,
} else { /* parent */ close (fd); }
```

Fine Pine ist eine #Datein /Eine Ontei must auf onen close 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

```
int id = fork():
Pipe lebt nur so lange, wie mind. ein Ende int n = read (ffel0), buffer, BSIZE);
penffnet ist. Alle Read Ende annaben.
} else { // Parent thread
geöffnet ist. Alle Read-Ends geschlossen →
STRPTPF an Write-End, Mehrere Writes können | char * text = "T <3 segfaults"
zusammengefasst werden. Lesen mehrere Pro- write (fd[1], text, strlen(text) + 1);
zesse dieselbe Pipe, ist unklar, wer die Daten
```

if (id = 0) { // Child thread close (fd [1]): // don't use write end char buffer [BSIZE]: else { // Parent thread close (fd[0]); // don't use read end

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

Fin Socket repräsentiert einen Endnunkt 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: (ID: ID-Adverse & Porter)

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 (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 onfragen 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)

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 0x8B addr in memory: 0x00 0x88 0x00 0x01

htons() konvertiert 16 Bit von Host-Ryte-order (IE) zu Network-Byte-Order (BE), hton1() 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-presentat 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 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 geschriehen. Der neue Socket kann keine weiteren Verhindungen annehmen, der hestehende 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. A) = write (fd. buf. len): recv (fd, buf, len, 8) = 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 andere Prozess den Socket noch geöffnet, bleibt die Verbindung bestehen. Die Gegenseite wird nich

benachrichtigt. int shutdown (int socket, int mode): Schliesst den Socket für alle Prozesse und haut die entsprechende Verbindung ab. mode: SHUT_RD (Keine Lese-Zugriffe mehr), SHUT_WR (Keine Schreib-Zugriffe mehr). SHIIT ROWR (Keine Lese- oder Schreih-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 (Kopiert die Nachricht aus dem Prozess: send (message)), Receive: (Kopiert die Nachricht 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 Empfänger kennen. Es gibt symmetrische direkte Kommunikation (Empfänger muss Sender auch kennen) und asymmetrische 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. szyklus 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 empangen 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 verfünhar oder (1)

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

```
message msg;
                                          message msg;
                                          open(Q);
while(1) {
open(Q);
while(1) {
            ext(&msq);
                                            receive(Q, &msq); // blocked until rec
  send(0, &msq): // blocked until sent consume next(&msq):
```

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

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 synchron 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, mg_attr: Div. Konfigs & Queue-Status, R/W mit mp_getattr/mg_setattr) int nq_close (nqd_t queue): Schliesst die Queue mit dem Descriptor queue für diesen Prozess int mg 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 Hauntsneichers 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 Fabgebildet. 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 und eine Mappina Table ie Prozess.

int fd = shm_open ("/mysharedmemory", O_RDWR | O_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 bleibt aber im System int shm_unlink (const char * name): Löscht das Shared Memory mit dem name. (bleibt vorhanden,

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

void * address = mmap(// maps shared memory into virt. address space of process // void *hint_address (0 because nobody cares) size of shared memory. // size t length (same as used in ftruncate) PROT_READ | PROT_WRITE, int protection (never use execute) // off_t offset (start map from first byte)

9.2. 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 performantei

9.3. VERGLEICH MESSAGE-OUEUES UND PIPES

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

10. UNICODE

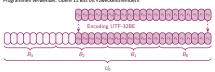
10.1. ASCII - AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE Hat 128 definierte Zeichen (erste Herzohl = Zeile zweite Herzohl = Soolte id.h. 41, = 4)

	0	1	2	3	4	5	6	7	8	9	A	В	С	D	Е	F
0	NUL				EOT	ENQ		BEL	88	TAB	LF		FF	CR.		SI
1						NAK		ETB	CAN	EH	88				RS	US
2	U	1		#	\$	%	&	,	()	*	+	,	-		/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	-	>	?
4	0	A	В	С	D	Е	F	G	Н	1	J	K	L	М	N	0
5	P	Q	R	S	Т	U	V	W	Х	Y	Z	[1	1	^	
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?»

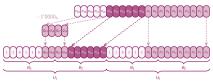
Code-Unit (CU): Einheit, um Zeichen in einem Encoding darzustellen (bietet den Speicherplatz) $P_i = i$ -tes Bit des unkodierten CPs, $U_i = i$ -tes Code-Unit des kodierten CPs, $B_i = i$ -tes Byte des kodierten CPs

Jede CU umfasst 32 Bit, ieder 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 Programmen verwendet. Obere 11 Bits 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, Un: high surrogate, Un: 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 Senarator ungültig und müssen «umgerechnet» werden



Encoding von U+10'437 (*) 00 0100 0001 00 0011 01111; 1 Code-Point P minus 1 00 00, rechnen und in Rinär umwandlen

P = 1.0437, Q = 1.0437, -1.0000, = 0437, = 00.0000001.000110111,

2. Obere & untere 10 Bits in Hex umwandlen

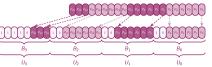
3. Oberer Wert mit D8 00s und unterer Wert mit DC 00s addieren, um Code-Units zu erhalten $= 0001_h + D800_h = D801_h, U_2 = 0137_h + DC00_h = DD37_h$ A 711 RF/LF 71152mmensetzen

 $BE = D801DD37_h$, $LE = O1D837DD_h$

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	signifikar
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

- $-\ddot{a}$: $P = E4_b = 0.00111001100$
- $\Rightarrow P_{10}...P_{6} = 0.0011_{b} = 03_{h}, P_{5}...P_{0} = 10.0100_{b} = 24_{h}$
- $\Rightarrow U_1 = {\rm CO_h}\; (= {\rm 1100\,0000_b}) + {\rm O3_h} = {\rm C3_h}, \quad U_0 = {\rm 80_h}\; (= {\rm 1000\,0000_b}) + {\rm 24_h} = {\rm A4_h}$
- $\Rightarrow \bar{a} = C3 \text{ A4}$
- \ddot{q} : $P = 1EB7_b = 0001 11 1010 11 0111_b$
- $\Rightarrow P_{15}...P_{12} = 01_h, \quad P_{11}...P_6 = 3A_h, \quad P_5...P_0 = 37_h$
- $\Rightarrow U_2 = EO_h \ (= 1110\,0000_b) + O1_h = E1_h, \quad U_1 = 80_h + 3A_h = BA_h, \quad U_0 = 80_h + 37_h = B7_h$ $\Rightarrow \breve{a} = \underline{\text{E1 BA B7}}_{b}$

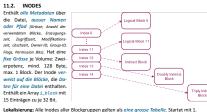
10.5. ENCODING-BEISPIELE

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	4100 _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	
ā	1E B7 _h	00 00 1EB7 _h	B7 1E 00 00 _h	E1 BA B7 _h	1EB7 _h	B7 1E _h	
	1 03 30 _h	00 01 03 30 _h	30 03 01 00 _h	F0 90 8C B0h	D8 00 DF 30h	00 D8 30 DF _h	

11. EXT2-DATEISYSTEM

Partition (Ein Teil eines Datenträgers, wird selbst wie ein Datenträger behandelt.), Volume (Ein Datenträger oder ein Partition davon.), Sektor (Kleinste logische 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.)

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 (Blocknum Anfana der Datei aus gesehen, wenn Datei eine ununterbrochene Abfolge von Blöcken wäre) UND Physische Blocknummern (Tatsächliche Blocknummer auf dem Volume)



Erzeugung: Neue Verzeichnisse werden in der Blockgruppe angelegt, die von allen Blockgrupper mit überdurchschnittlich vielen freien Inodes die meisten Blöcke frei hat. Dateien in der Blockgruppe des Verzeichnis oder nahen Gruppen. Bestimmung anhand Inode-Usage-Bitmap.

File-Holes: Bereiche in der Datei, in der nur Nullen stehen. Ein solcher Block wird nicht alloziert.

11.3. BLOCKGRUPPE

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

Layout: Black 0 (Kopie des Superblacks). Black 1 his n (Kopie der Gruppendeskriptgrentabelle). Black n+1(Block-Usage-Bitmap mit einem Bit je Block der Gruppe), $Block \ n+2$ (Inode-Usage-Bitmap mit einem Bit je Inode der Gruppe), Block n+3 bis n+m+2 (Tabelle aller Inodes in dieser Gruppe), Block n+m+3 bis Ende der Grunne (Bläcke der eigentlichen Daten)

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

Sparse Superblock: Kopien des Superblocks werden nur in Blockgruppe 0, 1 und allen reinen Potenzen von 3, 5 oder 7 gehalten (Sehr hoher Wiederherstellungsgrad, aber deutlich weniger Platzverbrauch). Grunnendeskrinter: 32 Byte Reschreihung einer Blackgrunne (Blackgrunner der Black-Licone-Bitman 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 = $(32 \cdot n)$ /Sektorgrösse Verzeichnisse: Enthält Dateieinträge mit variabler Länge von 8 - 263 Byte (48 Inode, 28 Eintraglänge

amenlänge, 18 Dateityp, 0 - 2558 Dateiname aligned auf 48). Defaulteinträge: «.» und «..» Links: Es gibt Hard-Links (gleicher Inode, verschiedene Pfade: Wird von verschiedenen Dateieinträgen referenziert, bolische Links (Wie eine Datei, Datei enthält Pfad anderer Datei).

11.4 VERGLEICH FAT NTES FXT2

FAT	Ext2	NTFS
Verzeichnis enthält alle Daten 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 Attribut 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

Reschreiben ein Intervall nhysisch konsekutiver Rläcke, ist 12 Byte gross (48 knigste Blos 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-Eintrag (Enthält Nummer des physischen Index-Blocks und kleinste logische Blocknummer aller Kindknoten) Index-Block (Enthält eigenen Tree-Header und Referenz auf Kindknoten)

Fxtent Tree Header: Renötigt ab 4 Extents, weil zusätzlicher Block, Magic Number F3 0As (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 (49)

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.

12.1.1. Notation	
(in)direkte Addressierung	Extent-Trees
	Indexknoten: Index → (Kindblocknr, kleinste Nummer der 1. logischen Blöcke aller Kinder)
indirekte Blöcke: indirekter Block.Index → direkter Block	$Blattknoten$: Index \mapsto (1. logisch. Block, 1. phy. Block, Anz. Blöcke)
	Header: Index → (Anz. Finträge. Tiefe)

Beispiel Berechnung 2MB grosse, konsekutiv gespeicherte Datei, 2KB Blöcke ab Block 20 00_h (In-)direkte Block-Adressieru 2 MB = 2^{21} B, 2 KB = 2^{11} B, $2^{21-11} = 2^{10} = 400$ _h Blöcke von 20 00_h bis 23 FF_h

 $1\mapsto 20\,02_h,\ ...,\ B_h\mapsto 20\,0B_h,\ C_h\mapsto 24\,00_h \text{ (indirekter Block)}$ $14.00_h.0_h \mapsto 20.00_h$, $14.00_h.1_h \mapsto 20.00_h$, ..., $14.00_h.3 F3_h \mapsto 23 FF_h$

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) Transaktion 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 Bild-

schirm ist die Wurzel (z.B. Dialogbax, Scrollbar, Button...). 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.

13.1 GUI ARCHITEKTUR

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

13.2 YIIR

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

Funktionen: XOpenDisplay() öffnet Verbindung zum Display, NULL = Wert von DISPLAY Umgebu XCloseDisplay() schliesst Verbindung, XCreateSimpleWindow() erzeugt ein Fenster, XDestrovWindow() 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, XUnmapWindow() versteckt Fenster, XUnmapSubwindows() versteckt Unterfenster, UnmapNotify Event

X Protocol: Legt die Formate für Nachrichten zwischen X Client und Server fest. Requests (Dienst anforderungen, Client → Server), Replies (Antworten auf Requests, Client ← Server), Events (Erejanismeldungen, Client

 \leftarrow Server), Errors (Fehlermeldungen auf vorangegangene Requests, Client \leftarrow Server) Request Buffer: Nachrichtennufferung auf der Client Seite. Für Effizienz Pufferung bei Ereignissen: Werden beim X Server und beim Client gepuffert. Server-Seitig berück-

sichtigt Notzworkvorfügharkeit Client-Seitige hält Events hereit X Event Handling: Ereignisse werden vom Client verarbeitet oder weitergeleitet. Muss festlegen, welche Typen er empfangen will. XSelectInput() legt fest, welche Events via Event-Masken emfpi werden, z.B. ExposureMask, XNextEvent () kopiert den nächsten Event aus dem Buffer

12.2 TEICHNEN

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 (Hintergrun

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: Legt diverse Eigenschaften fest, die Systemaufrufe nicht direkt unterstützen

endicke, Farben, Füllmuster). Client kann mehrere GCs gleichzeitig nutzen.

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 jedem Fenster assoziiert. Generischer Kommunikations-Mechan

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 14 MELTDOWN

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

Der Prozessor muss dazu gebracht werden können:

aus dem geschützten Speicher an Adresse a das Byte m_a zu lesen2. die Information m_a in irgendeiner Form f_a zwische

 binäre Fragen der Form «f_a ≟ i» zu beantworten Von i = 0 bis i = 255 iterieren: f_n = i

5. Über alle a iterieren 14.1 DEDECORMANCE OPTIMIED INCEN

Manning des Speichers in jeden virtuellen Adressraum, Out-of-Order Execution (O3E), 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 Attac 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 Kernelbzw. 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 «gezwungen: werden, indem im verworfenen Branch auf Speicher zugegriffen wird. Nicht leicht zu fassen, aber auch nicht leicht zu implementieren.

BSys2 | FS24 | Nina Grässli & Jannis Tschan