Speicher / Caches

RAM-Aufbau In Zeilen und Spalten, es wird immer eine Zeile gelesen → bursty reads sind schneller

SRAM Statisch, mehr Bauteile → teurer

DRAM Dynamisch, weniger Bauteile (R/C), braucht alle 64ms refresh

Speicherhierarchie Volatile (register/cache/RAM), Non-Volatile (SSD, HDD, ...). ↑: CPU-Distanz/Kapazität/Fehlerraten; ↓: Geschwindigkeit, Preis pro Byte

Lokalität (Zeit) `s += a[i]` in einem loop: Wenn `s` jetzt gebraucht wird, vermutlich auch demnächst

Lokalität (Raum) lokale Variablen sind in der Nähe im RAM: Wenn `a` gebraucht wird, dann vermutlich auch `a + 1`, ...

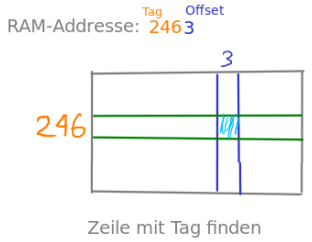
Timing

- T_c: Zugriffszeit auf Cache
- T_M: Zugriffszeit auf RAM ($T_M >> T_C$)
- p_c: Wahrscheinlichkeit von cache hit - Wegen Lokalität oft $90\% - 100\%$
- Durchschnittliche Zugriffszeit: $E(T) = p_c T_c + (1 - p_c) T_m$

Typen

Fully associative cache

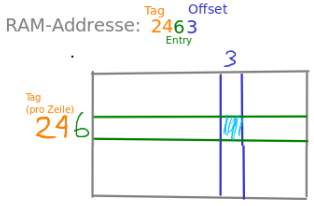
`tag(A) = A`; cache speichert volle Adresse. Müsste quasi eine Hashmap in Hardware implementieren, oder iterativ O(n). Daher oft $< 4\,\text{kB}$.



Zeile mit Tag finden

Direct mapped

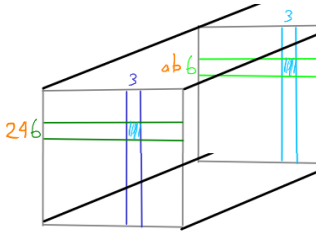
`tag(A) = A[m..n-1]`; wir maskieren die Adresse. Einfach und schnell, aber viele Kollisionen - wir brauchen also mehr Kapazität.



Zeile mit Entry finden,
Tag vergleichen

k-way set associative

Wir haben k direct mapped caches, d.h. wir können pro way/set-Paar ein Entry haben.



Wenn wir nun ab63 finden wollen, müssen wir vier Cache lines (→ 1 set) lesen, und die Tags vergleichen.

Speicherbedarf (k-way)

- z.B. 32 kB cache, 4-way; $l = 64\,\text{B}$ (Cachezeilenlänge)
- Cachezeilen $z = 32\,\text{kB}/64\,\text{B} = 2^{15}/2^6 = 2^9 = 512$
- (→ pro way: $z' = 128$)
- Offset (Spalte) braucht $\log_2(l)$ (→ 6)
- Set (Zeile) braucht $\log_2(z')$ (→ 7)
- Tag braucht den Rest, bei Adressbreite von 64bit z.B. $64 - 6 - 7 = 51$

Zugriff

R: Look-Aside Zugriff geht gleichzeitig auf Bus und Cache. Entweder legt der Cache den Wert auf den Bus (hit), oder der Cache liest den Bus (miss). Einfacher und günstiger.

R: Look-Through Zugriff geht direkt auf den Cache, Cache kommuniziert mit Bus (falls miss). Weniger Bus traffic.

W: Write-Through Synchron, CPU schreibt Daten zum Cache/RAM und wartet auf beide.

W: Write-Back Asynchron, CPU schreibt Daten zum Cache, Cache schreibt die Daten ins RAM. Problem: Cache-Koheränz bei Multiprozessor-Systemen.

Dynamischer Speicher

Stack/Heap Stack kann dynamisch Speicher allozieren, hat aber eine beschränkte Grösse.

double free Grösse des Blockes wird für `free` direkt vor dem Block gespeichert. Mit double-free wird aber beliebiger Wert als Grösse angenommen.

memory leak Speicher wird nie freigegeben (obwohl es keinen Pointer mehr darauf gibt) - problem bei expliziter Freigabe.

interne Fragmentierung Interne Block-Grösse ist grösser als angeforderter Bereich → waste

externe Fragmentierung Programm fordert unterschiedlich grosse Speicherbereiche an, gibt gewisse wieder frei, es entstehen Lücken.

Implementierung

Fixe Blockgrösse

Speicher wird in fixe Blöcke unterteilt. Variable Zuordnungsgrösse: Spezialfall mit Blockgrösse 1. Blockgrösse ist ein Tradeoff zwischen wenig Fragmentierung (kleine Blöcke) und bessere Performance bzw. weniger Metadaten (grosse Blöcke).

Object pools

Ein Block (bzw. Page) wird in kleinere Blöcke (für je 1 bestimmtes Objekt) mit fixer Grösse aufgeteilt. Wenig/keine Verschwendung; Rekombination entfällt. Freiliste mit freien Objekten.

Buddy-System

Ähnlich wie beim *quick fit*-Algorithmus gibt es verschiedene Blockgrössen, jeweils 2^k. Blöcke werden in Zweiergruppen geteilt:

```
[               ] (32, n=6) 00000 (requested: 3B)
[       ][ 16 f ] (n=5)     00000 / 10000
[   ][8f][      ] (n=4)     00000, 01000  / 10000, 11000
[][][   ][      ] (n=3)     00000, 00100  / 01000, 01100 / ...
4!4f  (4 used, 4 free)      ^- buddies -^ (nur bit n anders)

buddies = a ^ b == 1 << level;
```

Reservation: Suche passenden Block (angeforderte Grösse auf 2^k aufgerundet). Falls nicht vorhanden: Finde nächstgrösseren Block und zerteile ihn.

Freigabe: Kombiniere (nur) *mit Buddy-Block* falls möglich (rekursiv).

Metadaten

Dezentral

Metadaten direkt vor/nach Speicherblöcken. Schnelle Rekombination, aber kein Schutz vor z.B. array overflows (ausser mit magic numbers).

Zentral

Eigene Datenstruktur für Metadaten. Entweder fixe Grösse oder mehr Komplexität, aber Schutz vor Überläufen.

Bitliste 1 bit pro Speicherblock, used/free. Suche nach Speicherbereich: Suchen nach genug langer 0-Kette (kann lange dauern).

Linked list Liste mit `(is_free, *start, size)` Elementen, sortiert nach Adresse. Einfaches Suchen; einfaches Zusammenführen von freien Nachbarn nach `free`.

Suchalgorithmen

First Fit Nimmt erste passende Lücke - Ansammlung von vielen kleinen Lücken am Anfang.

Next Fit Nimmt erste passende Lücke nach zuletzt reserviertem Bereich. Kleine Lücken überall, daher schlechter als FF.

Best Fit Durchsucht ganze Liste nach bester Lücke. Sehr langsam, mehr Verschwendung da verbleibende Lücken zu klein sind, um genutzt zu werden.

Worst Fit Wie BF, sucht aber nach grösster Lücke. Langsam, keine guten Ergebnisse.

Quick Fit Verschiedene Grössenklassen (z.B. $1..n$ oder 2^n) mit eigenen linked lists. Allozierung wählt nächstes freies Element der passenden Klasse. Sehr schnell, aber Rekombination ist aufwändig (Nachbarn sind schwer zu finden).

Virtueller Speicher

Realer Speicher Programm kriegt echte Addressen, ggf. zur Laufzeit umgeschrieben für Relokation. Speicherschutz möglich (Code-Register das nur vom OS geschrieben werden kann mit Code pro Prozess, oder Limit-Register für relative Addressen).

Translation Lookaside Buffer Speichert häufig genutzte page mappings (Lokalität)

Memory mapped file Echte Datei statt swap-file via RAM

Virtueller Speicher OS konfiguriert MMU, Prozess kennt nur virtuelle MMU-Addressen.

Ungültiger Zugriff MMU signalisiert Fault Interrupt zum OS.

Swapping OS lädt Page von Sekundärspeicher ins RAM und passt mapping table an (MMU kennt nur RAM!).

Pages Virtueller Speicher basiert auf *pages* (meist 4 kB)

Page Frames Hauptspeicher wird in *page frames* unterteilt (jeder Prozess braucht ≥ 1 page frame). Bei einer Addresse wie `0xAB 78 9C CC` ist dann `0xAB789` die page frame number, und `CCC` (12 LSB) der Offset.

Page Tables

Die MMU muss wissen, ob eine page gültig ist, und in welchem page frame sie ist. Das OS muss wissen, wo die page ist (Haupt-/Sekundärspeicher), ob sie verwendet wird, und ob sie seit dem letzten Einlagern verwendet wurde.

Single level

Array mit `(frame_num, status_bits)` für jede page, mit page number als index.

Wenn Prozess auf `0x8765431` zugreift, ermittelt die MMU die page number `0x8765`, schaut in der page table da nach, findet `0xABCD0` → Addresse ist also `0xABCD0321`. Wird sehr schnell gross, Beispiel:

- 4 kB page size (2^{12}B); 32-bit Addressen
  → $\log_2(4\,\text{kB}) = 12$ bits Offset, 32 - 12 = 20 bit page number
- 4 GB RAM (2^{32}B)/4 kB = 2^{20} (1M) page frames
  → 20 bit frame number
- 4 GB virtueller Speicher → 1M Einträge
- 32 bit pro Eintrag (page nr + statusbits)
- Page table ist 4 MB pro Prozess!

Two level

Page number wird in *directory index* und *page table index* aufgeteilt. Es gibt mehrere page tables. Das page directory (für jede page table) sowie die page table (für jede page) haben ein used-bit.

```
page_table = page_directory[directory_index]
page = page_table[page_table_index]
```

Beispiel:

- Prozess will `0x87654321`
- Directory index: `0x21D`
- Page number: `0x254`
- MMU schaut bei Index `0x21D` nach, kriegt pointer auf secondary level page table (`0x2000`)
- MMU schaut in SLPT an Index `0x254`, kriegt frame number `0xABCD0`
- Reale Addresse: `0xABCD0321`

Mit 2^{10} Einträgen pro Page haben wir so 2^{20} Pages pro Prozess!

IPC

Shared memory Prozesse teilen sich Speicher. Verschiedene Pages (in den zwei Prozessen) für denselben Page Frame.

Message passing OS kopiert Daten zwischen zwei unabhängigen pages/frames. Mehr Isolation, aber Overhead.

Page tables auf x86

Jede Page hat ein `P` (present) bit. Mit `P = 1`: MMU-spezifisches Layout mit Page Frame und status bits (u.a. "Accessed" für jeden Zugriff, "Dirty" bei Schreibzugriff). Mit `P = 0`: MMU wirft interrupt, OS kann beliebige Infos (Ort im Sekundärspeicher) in der table speichern. OS wirft segfault bei ungültigen Infos.

Paging-Strategien

Ladestrategien

Welche Page wird wann geladen? Beeinflusst Häufigkeit von Page-Faults.

Demand Paging Paging werden auf Anfrage bei Interrupt geladen. Minimaler Aufwand, aber lange Wartezeiten für den Prozess bei jedem page fault.

Prepaging System versucht heuristisch zu ermitteln, welche Pages gebraucht werden.

Demand Paging mit Prepaging Lokalität: Wird eine Page gebraucht, werden vermutlich auch Nachbarn gebraucht und spekulativ mitgeladen.

Entladestrategien

Wann werden im RAM modifizierte Pages zurück geschrieben? Beeinflusst Verzögerung bei Page Fault.

Demand cleaning Page wird zurück geschrieben wenn Frame wiederverwendet werden soll. Minimaler Aufwand, erhöhte Wartezeit für Prozess der neue Page braucht

Precleaning Page wird frühzeitig in Sekundärspeicher geschrieben. Reduzierte Wartezeit, aber wenn Page nach Schreibvorgang nochmals verändert wird schlecht.

Page Buffering OS hat Liste von un-/modifizierten pages. Unmodifizierte werden zuerst ersetzt, modifizierte werden als batch geschrieben und in die andere Liste verschoben. Prozess kriegt schnell neue Page.

Verdrängungsstrategien

Welche Page wird aus dem Hauptspeicher entfernt, wenn Speicher knapp ist?

MMU setzt R-bit (referenced/accessed) bei jedem Lese-/Schreibzugriff, OS löscht es z.B. periodisch. MMU setzt M-bit (modified/dirty) bei Schreibzugriff, OS löscht es nachdem page zurückgeschrieben wurde. Pages mit M=0 werden bevorzugt ersetzt (zurückschreiben nicht nötig).

optimal Ersetze Page, die am spätestens in der Zukunft gebraucht werden wird. Unrealistisch in der Praxis (ausser bei genau bekanntem Verhalten), aber Referenz als Grenze des Machbaren.

FIFO Braucht keine Status bits, ersetze älteste Seite nach Ladezeitpunkt. Alte, häufig benutzte Seiten werden immer wieder entfernt und geladen; mehr RAM kann zu mehr faults führen (*Beladys Anomalie*)

Second chance Wie FIFO, aber R-bit der letzten Page (erstes LL-Element) wird geprüft. R=0: Wurde nicht benutzt → weg; R=1: `R := 0`, page an das Ende der LL ⇒ entfernt älteste nicht verwendete Page. Neue Page: `R := 1`

Clock Effizientere Implementierung mit circular linked list und next-pointer. e HW-Aufwand; grössere Page-Einträge.

NFU (not frequently used) Pro Page ein counter, periodischer Interrupt inkrementiert counter falls benutzt, ersetze Page mit niedrigstem counter. Alte Pages (früher oft verwendet) haben aber einen hohen counter selbst wenn sie nicht mehr genutzt werden.

NFU mit aging counter ist Verlauf: `cnt = (cnt >> 1) + (R << n)`

Working Set Behalte Pages im "Arbeitsbereich" mit Intervall T. Jede Page hat ein timestamp t. Bei Fault interrupt werden alle pages gescannt; wenn `R = 1` wird t aktualisiert, wenn `R = 0` wird entfernt falls $\Delta t < T$. Fallback: älteste page.

WSClock Circular linked lists mit zusammenhängenden Pages

Adressen

absolut direkt ohne Abstraktion, wie Speichercontroller

absolut indirekt virtueller Speicher, Relok., Prozesse schützen

relativ direkt Relokation, Limit-Register

I/O

Memory mapped Gerät hat Adressbereich der wie Speicher benutzt wird. Diese Adressen können aber nicht für RAM benutzt werden. Einfach

Port mapped Eigener IO Bus und Addressraum. CPU braucht zusätzliche Logik (und Instruktionen), System braucht zusätzlichen Bus.

- **via Speicherbus** RAM-Bus hat zusätzliches selector bit, CPU hat I/O-Instruktionen. Wie Port mapped, aber Systemdesign ist einfacher.

Polling Normalerweise schlecht, aber (bei schnelleren Geräten) bessere (und deterministischere) Performance.

Interrupt Controller Hängt an allen Devices die einen Interrupt senden wollen. Signalisiert Interrupt der CPU und gibt Interrupt-Nummer an.

DMA Eigener Controller - CPU programmiert DMA und gibt Bus frei, DMA lässt das Gerät direkt die Daten auf den Bus kopieren und setzt Interrupt.

- **Einzeltransfer** DMA nutzt Bus nur für einen Transfer (periodisch wenige Daten)
- **Blocktransfer** DMA belegt Bus länger, wenn Gerät viele Daten liefern kann (z.B. HDD, NIC)

Treiberarchitektur

- **Ohne** Programme kommunizieren direkt mit der Hardware. Sicherheit/Stabilität ist ein Problem, und Programme sind komplexer. Nur ein Prozess kann auf die Hardware zugreifen.
- **Mit** OS stellt API für die Hardware bereit; OS hat einen Treiber (generisch, vom Hersteller, ...). Treiber laufen meist im Kernel-Mode.
- **Microkernel** Treiber sind Prozesse im User-Mode, generischer Interrupt-Handler der Treiber-Prozesse anstösst. Stabiler aber langsamer.

GUIs

Baumstruktur Bildschirm ist das root window, Fenster haben Unterfenster (widgets).
Fenster Rechteckiger Bereich des Bildschirms

Windows

Handle Pointer auf ein Objekt (repräsentiert OS-Ressource), opaque
MSG struct Enthält Fenster-Handle, event type, params, Zeitstempel, Hotspot (Pixel im Mauscursor). Für alle Messages gleich gross.
Keyboard messages `TranslateMessage()` macht `WM_CHAR` message mit char-Code aus `WM_KEYDOWN` mit scancode.
Window procedure Callback fürs OS, welches Messages akzeptiert. Wird via window class mit Message-Typ assoziiert. `DefWindowProc()` ist die default window procedure, muss für alle Messages die von `WndProc` nicht bearbeitet wurden aufgerufen werden.
Window class Definiert window procedure, style, Menü, etc. für ein Fenster. Es gibt vordefinierte Klassen, oder es können benutzerdefinierte via `RegisterClass` registriert werden. Wird über Name (String) referenziert.
Message Queue FIFO von Messages. Einige Typen (`WM_PAINT`/`_QUIT`/`_TIMER`) werden erst am Schluss abgearbeitet, `WM_PAINT` wird zusammengefasst. Auch Dinge wie "Text setzen" passieren via messages (an eigenes Window). Nur GUI-threads haben Queue.
PostMessage() fügt Message am Ende ein (non-blocking für caller). Gibt `bool` zurück (queuing OK). Mit `NULL` als handle in eigene Queue.
SendMessage() ruft direkt window procedure auf (blocking für caller). Gibt `LRESULT` zurück, kann bei mehreren Threads zu Deadlocks führen.
GetMessage() Wartet blockierend, gibt 0 bei `WM_QUIT` zurück. Hat Filter für Window und Type (min/max).
Ressourcen Windows hat einen resourcen-Compiler der einige C-Präprozessor-Direktivne kann. Darin sind ressourcen-Definitionen, die dann zu `.res` kompiliert werden, was der Linker versteht..
Device-Context Abstraktion von Gerät (z.B. Bildschirm, Fenster, Drucker), das z.B. auch Font und "Framebuffer" für das Gerät enthält. Kann geklont werden (z.B. um vorzuzeichnen).

```
int WinMain(...) {
  RegisterClass(...);
  CreateWindow(...);
  while(GetMessage(...)) {
    TranslateMessage(...);
    DispatchMessage(...);
  }
}

LRESULT WndProc(....) { ... }
```

Unicode

UTF-32 Ein Codepoint als 4 byte direkt (big/little endian)
UTF-8 Superset von ASCII. Keine Endianness.
UTF-16 LE/BE, 2 byte.
Windows Intern UTF-16, diverse API die doppelt als `A` und `W` Variante existiert.
Code Unit 8/16/32-bit unit in UTF-8/16/32

X

Puffer Puffer existiert auf Client und auf Serverseite (möglichst wenig Traffic!)
Window classes InputOutput und InputOnly.
Ressourcen Infos mit IDs, z.B. Pixmap/font/...
Grafikfunktionen Rastergrafik mit Farbtabellen, primitives, Font. Brauchen graphics context für Liniendike, Farben, etc.
Events Typ `XEvent`, union über alle Event-Typen (also so gross wie der grösste Typ). Erster Member ist ein `type` int, Code wählt damit den passenden Union-Member.
Fehlerbehandlung Rückgabewerte, `XSetErrorHandler()` (Protokoll), `XSetIOErrorHandler` (fatal)
Display Rechner mit Tastatur/Pointing device/screen

```
while (true) {
  XNextEvent(display, &event);
  switch (event.type)
  {
  case Expose:
    ...
    break;
  ...
  }
}
```

Dateisysteme

File-Deskriptor Index in die Tabelle der geöffneten Dateien (pro Prozess)
Volume Datenträger oder Partition
Hardlink Mehrere Pfade oder Inode-links für dieselbe Datei

FAT32

Bootsektor Enthält Sektorengrösse, Anzahl Sektoren pro Cluster, Anzahl FATs und Grösse, Volumegrösse, Label, etc.
Cluster Fässt Sektoren von Dateien zusammen.
FAT Enthält Clusterketten-Eintrag pro Cluster (erster Cluster einer Datei). Der Eintrag zu diesem Cluster hat dann die Addresse des nächsten Clusters (oder Ende-Zeichen). Es gibt ein Backup.
Verzeichnis Eintrag der auf weitere Directory/File Entries zeigt.
Eintrag 8.3 filename, Clusternummer, Flags, Size, Timestamps

NTFS

Konzept "Alles ist eine Datei" (auch Metadaten)
Bootsektor Enthält nur Dinge wie Sektorengrösse, Sektoren pro Cluster, Cluster der MFT, etc.
MFT Beschreibt Dateien, selber auch eine Datei. Mehrere Kopien.
Records Liste von Attributen (aka Streams)+ Endmarker.
Attribute Die meisten Attribute sind optional. Entweder Name oder Typ muss sich unterscheiden. Entweder resident (komplett im record, z.B. Dateiname) oder non-resident (wird ausserhalb der MFT fortgesetzt). Selbst `$DATA` kann für kleine Dateien resident sein.
Runlist Sequenz von Runs. Ein Run sind mehrere aufeinanderfolgende Cluster.
Komprimierung NTFS hat sparse files, Run ohne Adresse in Runlist. Kann aber LZ77 wenn damit Cluster eingespart werden, setzt als placeholder dann Sparse-Run ein.

Ext

Block Mehrere Sektoren, entspricht Clustern. Es ist aber das ganze Volume aufgeteilt in Blöcke.
Inode 128-bit Beschreibung einer Datei - alle Metadaten ausser Name/Pfad. Verweist auf Blöcke, entweder direkt (¡= 12), oder 1/2/3-fach indirekt.
File Holes Wenn Block-Eintrag 0 ist, enthält der Block nur Nullen; wie sparse files.
Verzeichnis Inode mit Dateieinträgen im Datenbereich.
Dateieintrag Variable Länge mit Inode, Typ, Dateiname
Blockgruppen Volume wird aufgeteilt in Gruppen von aufeinanderfolgenden Blöcken. Gruppendeskriptoren beschreiben Blockgruppen.
Blockgruppe 0 Lage ist abhängig von Blockgrösse (erst ab Block 1 bei ¡= 1024). Enhält Superblock.
Sparse Superblocks Falls aktiviert, werden Superblocks nur an einigen Orten gehalten.
Extent Trees (ext4) Baum ähnlich wie NTFS Runs. Knoten zeigen auf Kinder oder auf Extents. Extent spezifiziert eine Anzahl fortlaufender Blöcke.

Journaling

Idee Inkonsistenzen schneller überprüfen, da nicht alle Metadaten verifiziert werden müssen.
mode Journal Daten werden komplett, ins Journal geschrieben, dann richtig, dann aus dem Journal entfernt. Maximale Datensicherkeit, schlechte Performance.
mode Ordered Nur Metadaten im Journal. Transaktion → Dateiinhalte → Commit. Dateien haben sicher richtigen Inhalt nach Commit, aber nicht optimale Performance.
mode Writeback Nur Metadaten im Journal. Commit und Dateiinhalte schreiben in beliebiger Reihenfolge. Sehr schnell, aber Daten sind ggf. nicht korrekt auf der Platte.

Programme

Unix environment `char *env[] = { "FOO=bar", (char*)0 }`
exec-Varianten `v`: vector, `l`: list (varargs), `p`: in `$PATH`, `e`: mit environment
System calls Programm/libc schreibt syscall Nummer nach `eax` und ggf. weitere Argumente und macht dann `int 0x80`. Interrupt handler läuft im Kernel Mode und ruft die entsprechende Service routine (`sys_*`) auf.
C toolchain Präprozessor → Compiler → Assembler → Linker

ELF

Enthält Informationen für Linker und Loader. Enthält ELF Header, Program Header Table (beschreibt Segmente die zur Laufzeit genutzt werden), Section Header Table (beschreibt Sektionen für den Linker, meist nicht vorhanden bei Objekt-Dateien) und Daten.

Segmente und Sektionen überlappen sch.

Sektionen

.bss Uninitialisierte Daten
.data Initialisierte Date
.debug Debug info
.rodata Read-Only Daten
.text Ausführbare Instruktionen
.symtab Symbol-Tabelle. Spezifiziert Symbole mit Name (via Stringtabelle), z.B. Addresse, Grösse, Info
.strtab String-Tabelle. Werden relativ zum Start der Tabelle referenziert, enthält z.B. Name von Symbolen.

Dynamische Bibliotheken

Idee: Austausch von Bibliothek ohne Executable zu verändern. Müssen verschiebbar sein, der Loader hat also mehr zu tun, der Linker weniger. Es kann RAM gespart werden, weil Bibliothek nur 1x im RAM ist (mit virtuellem Speicher)

Dazu braucht es PIC (position independent code), weil sich die Library nicht z.B. zwei Prozesse anpassen kann. Alle Referenzen zu Symbolen müssen relativ zu `eip` sein.

x86_64 hat relative call/move Instruktionen, x86_32 aber nur call. Dies kann umgangen werden mit einer Hilfsfunktion, die Rücksprungaddresse vom Stack in ein Register kopiert.

Es gibt eine global offset table (GOT) mit allen Adressen von sichtbaren Symbolen, Loader füllt die Addresse ein. Der Code nutzt relative Adressen in die GOT. Addressen finden ist teuer, deshalb lazy binding.

Um bedingte Sprünge (beim Funktionsaufruf mit lazy binding) zu vermeiden gibt es eine Procedure Linkage Table (PLT), die in GOT springt. Dieser zeigt auf eine Proxy-Funktion die lazy binding implementiert - sie sucht die Funktion und überschreibt GOT-Eintrag.

Manuelles Laden

```c
void *lib = dlopen("libfoo.so", RTLD_LAZY);
// handle lib == NULL, e.g. with dlerror()
typedef char *(*GetMessage)(void);
GetMessage get_message =
   (GetMessage)dlsym(lib, "get_message");
// handle get_message == NULL
get_message();
int fail = dlclose(lib);
// handle fail
```

`RTLD_NOW` / `RTLD_LAZY`: early/late binding
`RTLD_GLOBAL` / `RTLD_LOCAL`: Mit global können Symbole für andere Objekt-Dateien verwendet werden.

Automatisches Laden

linker name libz.so (symlink auf soname). Vom Linker verwendet.
soname libz.so.1 (symlink auf real name). Steht im Executable, wird vom Loader verwendet (da die Library mit der gleichen Major-Version ABI-kompatibel sein sollte)
real name libz.so.1.2.11

utils.c

Zweierpotenzen / binary

2^0	1	0	0000
2^1	2	1	0001
2^2	4	2	0010
2^3	8	3	0011
2^4	16	4	0100
2^5	32	5	0101
2^6	64	6	0110
2^7	128	7	0111
2^8	256	8	1000
2^9	512	9	1001
2^{10}	1024	A	1010
2^{11}	2048	B	1011
2^{12}	4096	C	1100
2^{13}	8192	D	1101
2^{14}	16386	E	1110
2^{15}	32768	F	1111
2^{16}	65536		

2^{10}	kB
2^{20}	MB
2^{30}	GB

$\Rightarrow 32\,\text{kB} = 2^5 \cdot 2^{10} = 2^{15}$

Encodings

UTF-8

First	Last	1	2	3	4
U+0000	U+007F	0xxxxxxx			
U+0080	U+07FF	110xxxxx	10xxxxxx		
U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxxx	
U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxxx	10xxxxxx

Beispiel

U+20AC ist 0010 0000 1010 1100
Encodiert: **1110** 0010 **10**00 0010 **10**10 1100

UTF-16

U+0000 - U+D7FF Direkt in 16-bit.
U+D800 - U+DFFF Reserved (surrogates)
U+E000 - U+FFFF Direkt in 16-bit.
U+10000 - U+10FFFF Als surrogate: `0x10000` abziehen, in 10 high/low bits aufteilen.
High value: `0xD800` + $b_{19} \dots b_{10}$ (`1101 10`)
Low value: `0xDC00` + $b_9 \dots b_0$ (`1101 11`)

Surrogate-Beispiel: U+10437 ist 0001 0000 0100 0011 0111
Daraus wird **1101 10**00 0000 0001 **1101 11**00 0011 0111
Als code units: D801 DC37
UTF16-BE: D8 01 DC 37
UTF16-LE: 01 D8 37 DC

Filesystem API

POSIX API

`open(path, flags) -> fd`, `close(fd) -> status`,
`read/write(fd, buffer, n) -> status`,
`pread` und `pwrite` mit Offset als Argument
`lseek(fd, offset, whence) -> status`,

C API

`fdopen(fd, char *mode) -> FILE`
`fopen(path, mode) -> FILE`
`fileno(FILE *stream) -> fd`
`fflush/fclose(stream) -> status`
`feof/ferror(stream) -> int` (bool-ish)
`fgetc(stream) -> int`
`fgets(char *outs, n, stream) -> int`
`fputs(int c, stream) -> status`
`fputc(text, stream) -> status`
`fungetc(c, stream) -> status`
`ftell(stream) -> offset`
`fseek(stream, offset, whence) -> status`
`rewind(stream)`: `fseek(stream, 0, SEEK_SET)`, clear errno

cheatsheet –version

August 18, 2017