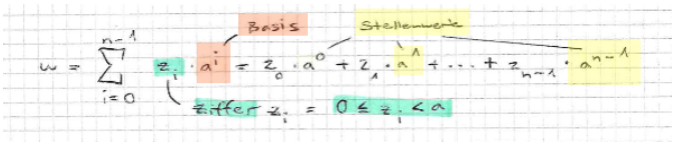
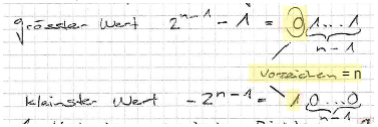
Stellenwertsystem



Nibble = 4 Bit, Oktett = 8 Bit, Byte = Oktett (für uns)

Minima/Maxima

unsigned/ohne Vorzeichen: $0...2^n - 1$
signed/mit Vorzeichen:



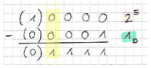
Zweierkomplement

Allgemein bei -1 alle Bits gesetzt: 1..1
Maximum $2^{n-1} - 1$ invertiert = 10..01
$N(0) = 0$, $2^{n-1} = N(2^{n-1}) = 100..$

bestimmt Vorzeichen, 0 = +, 1 = -, index = n

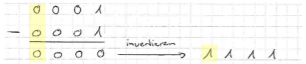
1. Variante: Subtraktionsverfahren

$N(b) = 2^n - b$, b = binäre Zahl



2. Variante: Inversionsverfahren

- 1, invertieren



3. Variante: invertieren + 1

Logik

Funktion, Parameter/Platzhalter, Argument
$f(x, y) := x \wedge g(x, y)$

Anzahl Argumentkombinationen 2^n, n = Anzahl Argumente
n-äre Funktion $f(x_0, .., x_{n-1}) = x_0 \wedge x_1 \wedge .. \wedge x_{n-1}$
nulläre Funktion f() = 1

Unäre Funktionen

Identität I(p): I(x) = x, I(1) = 1, I(0) = 0 oder einfach x
Negation N(p): N(1) = 0, N(0) = 1 oder einfach $\neg x$
F(p) immer 0 (eigentlich nullär)
T(p) immer 1 (eigentlich (nullär)

Disjunktion, Konjunktion

Disjunktion: $\vee$, **Konjunktion:** $\wedge$
Kanonische disjunktive Normalform (KDNF): Alle wahren Zeilen aus Wahrheitstabelle mit $\wedge$ verknüpft

Dualität

Dualität: 0 und 1, v und $\wedge$ sind dual zueinander Ersetzt man in einer korrekten Formel alle 0, 1, und durch ihr jeweils duales Element, ist auch diese duale Formel korrekt.

Regeln

Neutrales Element	$x \vee 0 = x$	$x \wedge 1 = x$
Idempotenz	$x \vee x = x$	$x \wedge x = x$
Komplement	$x \vee \bar{x} = 1$	$x \wedge \bar{x} = 0$
Extremum	$x \vee 1 = 1$	$x \wedge 0 = 0$
Kommutativität	$x \vee y = y \vee x$	$x \wedge y = y \wedge x$
Assoziativität	$x \vee (y \vee z) = (x \vee y) \vee z$	$x \wedge (y \wedge z) = (x \wedge y) \wedge z$
Distributivität	$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$	$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$
Absorption	$x \vee (x \wedge y) = x$	$x \wedge (x \vee y) = x$
De Morgan	$\overline{x \vee y} = \bar{x} \wedge \bar{y}$	$\overline{x \wedge y} = \bar{x} \vee \bar{y}$

Binäre Funktionen

Exklusives Oder

$x \oplus y$
Entweder x = 1 oder y = 1, nicht beides gleichzeitig
Negation der Äquivalenz $\neg(x \leftrightarrow y)$
Entspricht der Addition zweier Bits ohne Übertrag (Übertrag ist x y)

NAND, NOT AND

$x \mid y = \overline{x \wedge y} = \overline{xy}$ heisst NAND (NOT AND): Nur dann 0, wenn $x = 1$ und $y = 1$.

x	y	$x \wedge y$	$x \mid y$
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

Die Basisoperationen können ausschliesslich aus | dargestellt werden

- $\bar{x} = \overline{x \wedge x} = x \mid x$
- $x \wedge y = \overline{x \mid y} = (x \mid y) \mid (x \mid y)$
- $x \vee y = \overline{\bar{x} \wedge \bar{y}} = (x \mid x) \mid (y \mid y)$

Prozessor-Zyklus

1. Prozessor fordert Wert von der Adresse an, die im Befehlszeiger steht. 2. Prozessor decodiert Instruktion aus Wert. 3. Prozessor wählt den zur Instruktion gehörenden Baustein aus. 4. Aktiver Baustein decodiert Parameter aus Wert. 5. Aktiver Baustein liest aus den Registern. 6. Aktiver Baustein führt Berechnung aus. 7. Aktiver Baustein schreibt in die Register. 8. Prozessor erhöht Befehlszeiger entsprechend der Länge der Instruktion.

Assembler

assembly = language, assembler, Compiler

Byte-Order

16 Bit:

	Big-Endian		Little-Endian	
Adressen	m_i	m_{i+1}	m_i	m_{i+1}
Bytes	b_1	b_0	b_0	b_1
Stellen	$s_3 s_2$	$s_1 s_0$	$s_1 s_0$	$s_3 s_2$
Beispiel:	CA	FE	FE	CA

32 Bit:

Beispiel 32-Bit-Zahl $b = 87654321$.

Adressen	m_i	m_{i+1}	m_{i+2}	m_{i+3}				
Big-Endian	b_3	b_2	b_1	b_0	87	65	43	21
Little-Endian	b_0	b_1	b_2	b_3	21	43	65	87

Zwei 16-Bit-Zahlen $b[0] = 4321_h$ und $b[1] = 8765_h$.

Adressen	m_i	m_{i+1}	m_{i+2}	m_{i+3}				
Big-Endian	$b[0]_1$	$b[0]_0$	$b[1]_1$	$b[1]_0$	43	21	87	65
Little-Endian	$b[0]_0$	$b[0]_1$	$b[1]_0$	$b[1]_1$	21	43	65	87

Byte 1 Byte / 8 Bit, db 0x35 Word 2 Byte / 16 Bit, dw 0x2135 ↔ db 0x35, 0x21
Doubleword 4 Byte / 32 Bit, dd 0x2135 ↔ db 0x35, 0x21, 0x00, 0x00
Quadword 8 Byte / 64 Bit, dq
Double Quadword 16 Byte / 128 Bit

Register

instruction pointer: ip in 16-bit, eip in 32-bit, rip in 64-bit

1	RAX	EAX	*AX*	AL
2	RBX	EBX	*BX*	BL
3	RCX	ECX	*CX*	CL
4	RDX	EDX	*DX*	DL
5	RSI	ESI	SI	SIL
6	RDI	EDI	DI	DIL
7	RSP	ESP	SP	SPL
8	RBP	EBP	BP	BPL
9	R8-R15	R8D-R15D	R8W-R15W	R8L-R15L

1. Accumulator, 2. Datenpointer, 3. Counter für Schleifen, Stringoperationen, 4. Pointer für I/O-Operationen, 5. Quelindizes für Stringoperationen, 6. Zielindizes für Stringoperationen, Exitcode, 7. Stackpointer, Adresse des allozierten Stacks, 8. Basepointer, Adresse innerhalb des Stacks, Basis des Rahmens der Funktion , 9. Zusätzliche Register

Obererer 8-Bit Teil kann separat verwendet werden: AH, BH, CH, DH
Operationen auf 8 Bit und 16-Bit-Registerteil ändern den Rest des Registers nicht, Bei Operationen auf 32-Bit Ebene wird der obere Teil auf 0 gesetzt

Adressierungsmodi

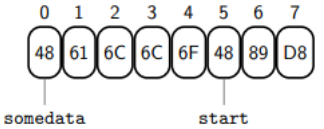
Die Adresse der Speicherstelle folgt unmittelbar(Displacement): mov rax, [0x1000]
Die Adresse der Speicherstelle steht in einem Register (Base): mov rax, [rcx]
(Index * Scale): Index ist ein Register und Scale ist 1, 2, 4, oder 8, mov rax, [rbx * 4]

Referenz = Wert der gespeicherten Adresse wird eingesetzt
$2E0_h = 2$
$rax = 2E0_h$
$movrbx, [rax] \leftrightarrow$ mov rbx, 2

Label

somedata: db 'Hallo'
start: mov rax, rbx

Label wird nicht übersetzt, wird mit dem Offset A(*label*) des nachfolgenden Befehls assoziiert



$A(somedata) = 0$
$A(start) = 5$

Arrays

Startadresse, Offset $O_a(index) = Index \cdot$ Variablengrösse in Byte (Byteadressierung)

```
move rax, 0
add rax, [a + 0 * 8]
add rax, [a + 1 * 8]
add rax, [a + 2 * 8]
move [r], rax
```

Arithmetische und logische Operatoren

Befehle können unterschiedlich lang sein → Sequenz muss Befehl für Befehl durchgegangen werden

`add z, q`	$z \leftarrow z + q$	
`sub z, q`	$z \leftarrow z - q$	
`neg z`	$z \leftarrow 0 - z$	Zweierkomplement
`inc z`	$z \leftarrow z + 1$	Inkrement
`dec z`	$z \leftarrow z - 1$	Dekrement
`and z, q`	$z_i \leftarrow z_i \wedge q_i$	Für jedes Bit i
`or z, q`	$z_i \leftarrow z_i \vee q_i$	
`xor z, q`	$z_i \leftarrow z_i \oplus q_i$	
`not z`	$z_i \leftarrow \bar{z}_i$	Bitweise Negation

Shifting

Links-Shift

Multiplikation einer Binärzahl mit 2^m
um m-Stellen nach links verschieben + mit m-Nullen auffüllen
1111 Linksshift um 2^1 = 11110

Rechts-Shift

Division einer Binärzahl um 2^m
um m-Bits nach rechts verschoben + links mit 0 auffüllen
1111 um 2^1 = 0111

Sign-Extension

Statt 0 wird überall 1 reinkopiert
Rechtsshift mit Signextension = Arithmetischer Rechts-Shift

Rotate

Rotates füllen mit den ursprünglichen Bits auf

Befehle

i = Konstante oder Register CL

`shl z, i`	$z \leftarrow z \cdot 2^i$	
`shr z, i`	$z \leftarrow z \cdot 2^{-i}$	z unsigned
`sar z, i`	$z \leftarrow z \cdot 2^{-i}$	z signed
`rol z, i`	Left-Rotate i Bits	
`ror z, i`	Right-Rotate i Bits	

Multiplikation

unsigned

Ergebnis einer Multiplikation benötigt doppelt so viele Bits wie einzelne Operanden
Also 2n Bit

signed

Vorzeichen explizit berücksichtigen!

Befehle

`mul z`	RDX:RAX ← RAX · z	unsigned, z 64-Bit-Operand
`mul z`	EDX:EAX ← EAX · z	unsigned, z 32-Bit-Operand
`mul z`	DX:AX ← AX · z	unsigned, z 16-Bit-Operand
`mul z`	AX ← AL · z	unsigned, z 8-Bit-Operand
`imul z`	RDX:RAX ← RAX · z	signed, z 64-Bit-Operand
⋮	⋮	⋮
`imul r, z`	$r \leftarrow r \cdot z$	signed, r 64-Bit- Register, z 64-Bit-Operand
`imul r, z`	$r \leftarrow r \cdot z$	signed, r 32-Bit- Register, z 32-Bit-Operand
`imul r, z`	$r \leftarrow r \cdot z$	signed, r 16-Bit- Register, z 16-Bit-Operand

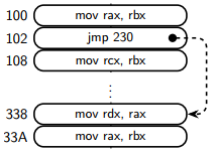
Division

sehr langsam → für Zweierpotenzen durch Rechts-Shift ersetzen

`idiv z`	RAX ← RDX:RAX/z RDX ← RDX:RAX mod z	signed, z 64-Bit-Operand
⋮	⋮	⋮
`div z`	RAX ← RDX:RAX/z RDX ← RDX:RAX mod z	unsigned, z 64-Bit-Operand
⋮	⋮	⋮

Relative Sprünge

JMP zahl = RIP←RIP + zahl
JMP label = RIP←label



Flags

gemeinsames Register RFLAGS
Carry Flag - CF Überlauf unsigned
Overflow Flag - OF Überlauf signed
werden immer beide bestimmt
Zero Flag - ZF Resultat = 0
Sign Flag - SF MSB des Resultats
Parity Flag - PF niederwertigste Byte = gerade Anzahl 1

Condition Codes

CC	Name	Flags
A	Above	CF = 0 und ZF = 0
AE	Above or Equal	CF = 0
B	Below	CF = 1
BE	Below or Equal	CF = 1 oder ZF = 1
E	Equal	ZF = 1
G	Greater	ZF = 0 and SF = OF
GE	Greater or Equal	SF = OF
L	Less	SF ≠ OF
LE	Less or Equal	ZF = 1 und SF ≠ OF
PE	Parity Even	PF = 1
PO	Parity Odd	PF = 0

einbuchstabigen CC: C ↔ CF = 1
negierter CC: NC ↔ CF = 0

cmp

cmp rax, rbx berechnet RAX – RBX, verwirft Ergebnis, setzt Flags

Bedingte Anweisungen

CMOVcc: Conditional MOV, Jcc: Conditional JMP, SETcc: Schreibt 1 ins 8-Bit grosse Ziel, wenn CC erfüllt, sonst 0

Programmstart/-ende

Entrypoint = _start:
Intel Prozessoren haben Privilege-Levels: 0 = OS, 3 = Programme, deshalb:

```
mov rax, 60; Funktionscode
mov rdi, 0; weitere Argumente, hier Exitcode 0
syscall; Handler für Syscalls -> Unter-Handler für 60
```

Stack, Funktionen, Variablen

call x legt die Rucksprungadresse auf den Stack und springt dann zu x
ret nimmt die Rucksprungadresse vom Stack und springt dahin
Parameter und lokale Variablen werden je nach Calling Convention auf dem Stack oder in Registern abgelegt
Parameter sind lokale Variablen, die vom Caller initialisert werden
Globale Variablen verhindern rekursive (allg. re-entrante) Funktionen, sind fix im Speicher mit eigener Adresse

---

C Toolchain

1. Präprozessor
2. Compiler
3. Assembler Assemblerdatei .asm→Objektdatei/Binärsequenz .o
4. Linker mehrere Objektdateien/Binärsequenzen →Executable

Objekt-Datei

Enthält Binärsequenzen
Symboltabelle Bestandteil von Obj-Datei::

```
0000        *UND*   y
0000        *UND*   g
0008    g   .text   x
0010    g   .text   f
```

Offset Exportierte Symbole: global deklarierte Label x, f Importierte Symbole: extern deklarierte Label = *UND*
In Objektdatei, Stellen mit Symbolen 0x0, da Adresse noch nicht bekannt

Executable

Jedes Symbol erhält einen eigenen, festen Platz im Executable

Präprozessor

1. Durchlauf

Entfernen aller Kommentare, fortgesetzte Zeile → in eine Zeile

```
puts ("Hallo␣\
HSR");          puts ("Hallo␣HSR");
```

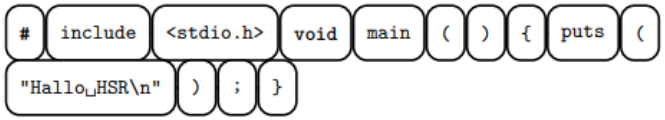
2. Durchlauf, Tokenization

Bezeichner
~~Sonderzeichen~~, Beginn mit _/A-Za-z
Präprozessor-Zahlen:
beginnt mit Ziffer, Bsp: 0, 1, 0123, 0x1234, .05, `0_.0*(nicht gueltig in C)*`, `0xE+12*`(wird nicht als 0xE + 12 interpretiert!)*, `p+`, `P+`, `p-`, `P-`, ungültige Zahlen → vom Compiler entdeckt
String- und Character-Literale
immer zusammengefasst, " escapen: \",' escapen: \'
Operatoren und Satzzeichen (punctuators)

```
( ) [ ] { } . , : ; ? ... -> # ##
= + - * / % & | ^ ~ ! << >> == != < > <= >= && ||
*= /= %= += -= ++ -- <<= >>= &= |= ^=
```

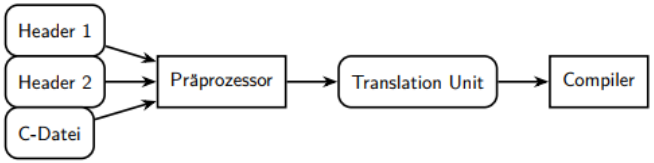
Der Präprozessor ist greedy:
ist `a+++++b` nicht `a++ + ++b` sondern `a++ ++ +b` (ungültiger Ausdruck)

Whitespace (Leerschlag, Zeilenumbruch, Tabulator) trennt Token voneinander, aber nicht alle Token mussen durch Whitespace getrennt werden, z.B. a+b↔[a][+][b].



3. Durchlauf
Präprozessor-Direktiven ausführen, Makros durch Expansion ersetzen
nächste Token als direktive
Direktiven: `include header`, `define makro`, `if`, `else`, `endif`

`#include`:
`#include <file.h>` sucht nur in den Systemverzeichnissen
`#include "file.h"` erst im aktuellen Verzeichnis und dann in den Systemverzeichnissen
1. Präprozessor fuhrt Durchläufe 1 bis 3 für file.h durch
2. Präprozessor setzt Arbeit nach der Direktive in Orignaldatei fort
Objektartige Makros: `#define XYZ 123`



Variablen

globale Variable analog Assembler, Speicher wird fix reserviert, Grösse durch Typ bestimmt
Bezeichner = Label auf Assembler-Ebene
Intialisierung:
`int x = 15;` ↔ x : dd 15
Ohne Initialwert→mit 0 initialisiert
Globale Variablen werden standardmässig exportiert (wie Assembler)

Globale Variablen aus anderen Dateien: `extern int c;`

Objekt

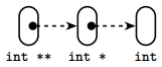
zusammenhängender Speicherbereich, Inhalt kann als Wert interpretiert werden **Objektgrösse bestimmen:** `sizeof(T)`

Basistypen

`char`	mind. 8 Bit (nach Impl. signed/unsigned)
`[signed] short [int]`	mind. 16 Bit
`[signed] int`	mind. 16 Bit
`[signed] long [int]`	mind. 32 Bit
`[signed] long long [int]`	mind. 64 Bit

Somit default-mässig signed
unsigned muss sonst strikt angegeben werden

Pointer-Typen



Referenzoperator

`&a` = Adresse der Variable `a`
`T*` ist der Typ des Ausdrucks `&a`

Dereferenzoperator

`*` wandelt Adress-Ausdruck in Ausdruck bezogen auf Inhalt um

*&a ergibt wieder a und &*b ergibt wieder b (wenn b eine Pointervariable ist)

Testen auf Null-Pointer: `if (px == 0)`

Funktionspointer:

```
int f (int x, int y);
int (*p) (int, int) = &f;
p = g; //andere Funktion zuweisen
int i = (*p) (1,2);
int j = p (3,4); //Alternativer Aufruf
```

Arrays

Bezeichner eines Arrays → Pointervariable
Jeder Pointer kann im Array verwendet werden: Mit Elementtyp T ist `a[index]` äquivalent zu `a + sizeof(T) * index`

Structs

```
struct T //Tag für Wiederverwendung
{int x; int y;};

struct T t, u;
```

belegt gleichen Speicherplatz wie `int x; int y;`einzeln
Zugriff: `t.x = t.y`
`x` gleiche Adresse wie Struct
Member müssen im Speicher nicht dicht liegen (Padding möglich)
Pointer auf Structs: `struct T* z; t->x = t->y;`

Zuweisungen eines Structs auf einen anderen → ganzer Inhalt kopiert(nicht nur Referenz)

Union

zur Verhinderung von Casts, ähnlich wie Structs, Start der Adressen der Elemente = Start der Adresse der Union Grösse der Union = Grösse des grössten Elements `union U{//Elemente}`

Funktionen

```
void f();
f(1,2,3); // OK
```

Parameter abhängig von Calling Convention in Registern/Stack übergeben
Immer **Call by value**

Pointer: `int f (int* f)`, Call by Reference emuliert

Arrays: Arrays werden als Pointer intepretiert/übergeben `f(int x[], int len)` = `f(int* x, int len)`
Structs: ganzer Struct als Argument kopiert, mit Struct Pointer →analog Java
Type-Definitionen anderer Bezeichner für denselben Typen definieren
`typedef struct {my_int x; } T;`

`printf`

indirekte Ausgabe über stdout, Ausgabe erst wenn Buffer voll
`write` ist direkte Ausgabe
sofortige Ausgabe: `fflush(stdout)`

vom Stack eingelesen:
%i `sizeof(int)`Bytes
%X `sizeof(int)`Bytes als Hex
%li `sizeof(long)`Bytes
%lli `sizeof(long long)`Bytes

als Pointer:
%p `sizeof(void *)`Bytes
%s `sizeof(char *)`Bytes auf null-terminierten String

Cache

Mittlere Zugriffszeit

Cache-Hit Gesuchte Adresse ist im Cache
Cache-Miss Gesuchte Adresse ist nicht im Cache
T_C Zugriffszeit auf den Cache
T_M Zugriffszeit auf den Hauptspeicher
p_C Wahrscheinlichkeit eines Cache-Hits (wegen Lokalitätseffekt oft > 0.9)

Erwartungswert E(T) der Cache-Zugriffszeit T:
$E(T) = p_C \cdot T_C + (1 - p_C) \cdot T_M$

Fully Associative Cache (FAC)

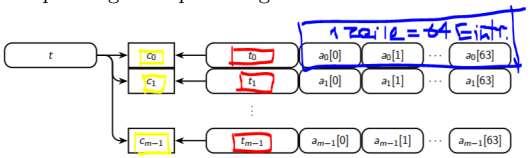
Eintrag i besteht jeweils aus Adresse a_i und und Datenbyte $d_i = [a_i]$

Cachezeilen à 64 Byte

nur die Adresse des ersten Bytes wird gespeichert
Total Einträge: Grösse des Caches(b) / 64 Einträge
$64 = 2^6$
n-Bit Adresse: t_i: oberen n-6 Bit + Offset(untere 6 Bit)
Overhead: $2^{n-6} \cdot b$

Lookup

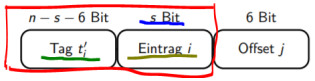
Zu jeder Zeile ein Hardware-Baustein c_i, der überprüft $t = t_i$
Überprüfung wird parallel gemacht



\+ Lokalitätsprinzip bestmöglich ausgenutzt
\- Lookups benötigen viel Hardware (c_i) und sind teuer

Direct-Mapped Cache (DMC)

Anzahl Einträge/Zeilen = 2^s
eine Cachezeile nur an einem einzigen Ort möglich
Eintrag wird durch die untersten s Bit des Tags t bestimmt
$t_i = t_i' \cdot 2^s + i$
gespeichert wird nur das reduzierte Tag t_i', i wird abgeleitet

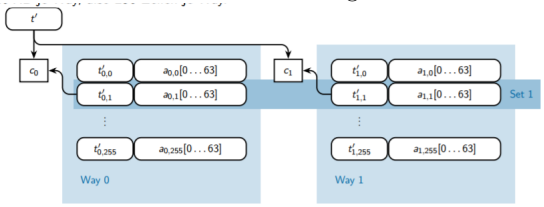


Lookup

ein einziger Vergleichsbaustein
Überprüfung t' = t'_i
Vollständige Adresse: $(t'_i \cdot 2^s + i) \cdot 2^6 + j$
\+ einfach zu implementieren + sehr schneller Lookup - viele Kollisionen (1234_h, $AB34_h$ bei s=8 gleicher Eintrag)

Set-Associative Cache

parallele Verwendung von k Direct-Mapped Caches
Jede Cachezeile kann in k Einträgen gespeichert werden
Anzahl Set = k
Jeder DMC = WAY
Set-Nummer = Nummer des Eintrags i



Wie viele Speicherstellen des Hauptspeichers werden jeweils auf dieselben Cacheeinträge abgebildet? $\frac{GrösseRAM \cdot Anz.Ways}{Gesamtgr.Cache}$ = Anteil Hauptspeicher à Anz.Ways Stellen
weniger komplex als FAC, weiniger Kollisionen als DMC, Genau so schnell wie FAC und DMC

Random Access Memory (RAM)

Heap/Stack

Heap: Speicherbereich für vollständig dynamischen Speicher
Nur explizite Speicherfreigabe durch OS vorgesehen:
Reservieren: `void * malloc (unsigned int s)`
Reserviert Speicherblock der Grösse s
gibt Adresse des allozierten Speicherblocks zurück
freigeben: `void free (void* p)`
Grösse wird häufig direkt vor Block angegeben: `*(p - x)`

Interne Fragmentierung: Heap reserviert grösserer Speicherblock als angefragt **Externe Fragmentierung:** Programm reserviert immer wieder Speicher, gibt unregelmässig frei

Feste Blockgrösse

Dezentrale Speicherung: Überläufe **Zentrale Speicherung:** Speicherplatz muss extra reserviert werden **Bitlisten:** 0 Block ist frei, 1 verwendet **Verkettete Listen:** Status(frei?), Start(Adresse erster Block), Size(Anzahl Blöcke), Next(Pointer nächstes Listenelement) — freie Elemente→zusammengeführt

Suchalgorithmen

First Fit: Wählt erste passende Lücke am Anfang **Next Fit:** Wählt erste passende Lücke nach zuletzt reserviertem Bereich **Best Fit:** Durchsucht alle Lücken und wählt die kleinste passende aus **Worst Fit:** Durchsucht alle Lücken und wählt die grösste aus

Grössenklassen/Quickfit

Bereiche nur in bestimmten Grössen zur Verfügung (Zahlen, Zweierpotenzen..), freie Bereiche in Liste, **Quickfit:** wählt kleinstpassenden aus Liste

Buddy-System

Wird ein 2^k-Bereich in zwei 2^{k-1}-Buddys geteilt, müssen deren Startadressen die untersten k-1 Bits = 0
Die Startadresse des einen Buddys ist a
Die Startadresse des anderen Buddys ist a mit einer 1 an Bit k
Bsp. k=8 →Buddies 1. Bereich: 0000'0000...0111'1111,
2. Bereich: 1000'0000...1111'1111
wenn gleich gross & einziger Unterschied bei k → Buddies

Objekt-Pools

Speicherbereich fester Grösse(Page) in kleinere Bereiche mit gleicher Grösse unterteilt(Objekte), keine Rekombi bei Rückgabe, Mehr Objekte benötigt?→neue Page, Freie Objekte in Freiliste

Memory Management Unit (MMU)

Pro Prozess eine Page-Table

Single-Level Page-Table

ein Eintrag pro Page + Lookup sehr schnell → Index = PageNumber

Two-Level Page-Table

Page Number wird aufgeteilt: Directory Index (nur eine!), Page Table Index (zweidimensionales Array)

Translation Lookaside Buffer: Cache für häufig benötigte Mappings **Inverted Page-Table:** Pro Frame ein Eintrag, schlimmenstenfalls alle Einträge durchsuchen **Hashed Page-Table:** Page Number als Key, gleicher Hash?→Linked List

Interprozesskommunikation (IPC)

Shared Memory: Prozesse teilen sich Speicher, + Kaum Overhead, - Aufhebung des Schutzes (Bufferoverflow) **Message Passing:** OS kopiert Daten zwischen Prozessen, + Prozesse können sich nicht direkt beeinflussen - Overhead

Paging

MMU setzt A-Bit/Accessed—R/Referenced bei jedem Zugriff auf Page, D-Bit/Dirty—M/Modified bei jedem Schreibzugriff auf Page→zurückschreiben in HDD

Dreschen/Häufiges Pagen

Hauptspeicher viel zu klein/zu viele Prozesse→mehr Hauptspeicher, Beschränkung Anz. Prozesse, Verminderung Paging-Strategien

Ladestrategien RAM←HDD

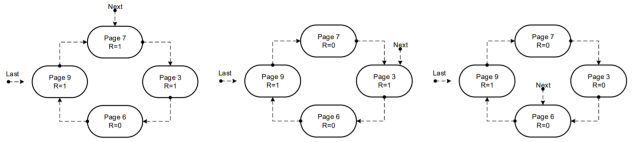
Demand Pagaing: Laden auf Anfrage, + min. Aufwand, - lange Wartezeiten **Prepaging:** Seiten frühzeitig geladen **Demand Paging mit Prepaging:** wie Demand + benachbarte Pages(Lokalitätpr.), + weniger Page-Faults, + Blocktransfer, - zu viel Aufwand

Entladestrategien RAM→HDD

Demand Cleaning: nur geschrieben wenn nötig, + min. Aufwand, -erhöhte Wartezeit **Precleaning:** Vorausschauendes Schreiben, + reduzierte Wartezeit, - wenn Pages nach Schreibvorgang nochmals geändert **Page Buffering:** 2 Listen: mit unveränderten Pages (zuerst ersetzt), veränderten Pages, +/- Precleaning, + schnelle Auswahl

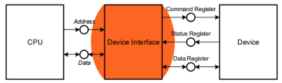
Verdrängungsstrategien

Beladys Anomalie: mehr Hauptspeicher→mehr Page Faults **Optimal:** erste Page, spätesten in Zukunft gebraucht **FIFO:** Problem: alte, häufig benutzte Pages, Belad. **Second Chance:** FIFO + prüft Referenced-Bit, 0=weg, 1=hinten + 0 **Least Recently Used:** ersetzt längste unbenutzte Page, notiert Zeitpunkt in Page-Table, + sehr nahe am Optimum - grosser Aufwand in HW, Page-Einträge grösser **Clock:**

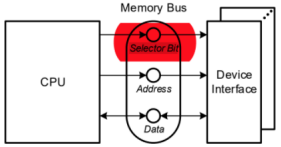


Not Frequently Used: Pro Page-Table Eintr. 1 Counter, ersetzt Page min. Counter, - alte, viel gebrauchte Pages bleiben erhalten **NFU mit Aging:** 4 Bit Counter = Zugr., kein Zugr., Zugr., kein Zugr. 1000→0100→1010→0101 **Working Set:** R=1: t gesetzt,R=0: aktuelleZeit - t<Working Set T→Nein: Page entfernt, Ja: im Working Set, behalten

In-/Output

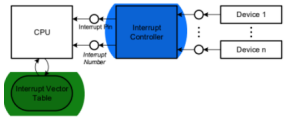


Memmory-mapped I/O: pro Gerät 1 Adressbereich, -1 für Memory, + Einfachheit **Port-mapped I/O:** separater Bus, zwei Adressräume (Speicher und Geräte), zwei Instruktionssets, - Komplexität **Port-mapped I/O via Speicherbus:** gemeinsamer Bus, zusätzliche Bitleitung für Trennung Speicher—I/O, Adressraum wird um 1 Bit erweitert



Kommunikationsmechanismen

Programmgesteuert/Polling (Busy Wait = ständig)
Interruptgesteuert: Interrupt-Vektor-Tabelle, Tabelle von Pointern auf Funktionen(von CPU aufgerufen)



Direct Memory Access (DMA)

DMA-Controller steuert Speicherbus anstelle CPU, 1. CPU programmiert DMA für Transfer, 2. DMA lässt Gerät direkt auf Speicherbus kopieren, 3. nach Beendigung setzt DMA Interrupt