Binär

Begriffe

Bit: Eine Stelle einer Binärzahl
Bit = 1: Gesetztes Bit(*set bit*)
Bit = 0: Gelöschtes Bit(*cleared bit*)
LSB: Least Significant Bit, niederwertigstes Bit, Bit 0
MSB: Most Significant Bit, höchstwertiges Bit, Bit n-1
Nibble: Binärzahl mit *vier* Bit.
Byte / Oktett: Binärzahl mit *acht* Bit.
Carry Bit: Übertragsbit, wird bei einem Übertrag gesetzt

Formeln

Grösste Darstellbare Zahl(unsigned): $2^n - 1$
Anzahl Zahlen: 2^n
Bereich(unsigned): 0 bis $2^n - 1$
Grösste Darstellbare Zahl(signed): $2^{n-1} - 1$
Kleinste Negative Zahl(signed): -2^{n-1}
Bereich(signed): -2^{n-1} bis $2^{n-1} - 1$
MSB = 0: Dient als +
MSB = 1: Dient als -

Addition / Subtraktion:

Addition dezimal	Addition dual	Subtraktion dezimal	Subtraktion dual
168 + 37 205	10101000 + 00100101 11001101	168 - 37 131	10101000 - 00100101 10000011

Multiplikation

Multiplikation dezimal	Multiplikation dual
12 * 14 12 48 168	1100*1110 1100 1100 1100 0000 10101000

Zweierkomplement(übliche Darstellung negativer Binärzahlen)

1. **Invertieren**
2. **Plus 1 Addieren**

0000'0001 → 1111'1110 + 1 = 1111'1111
0000'1111 → 1111'0000 + 1 = 1111'0001
0101'1010 → 1010'0101 + 1 = 1010'0110

	signed	unsigned
4 Bit	−8 … 7	0 … 15
8 Bit	−128 … 127	0 … 255
16 Bit	−32K … 32K − 1	0 … 64K−1
32 Bit	−2G … 2G − 1	0 … 4G − 1

2^{10}	$1.024 \cdot 10^3$	K	Kilo	Ki	Kibi
2^{20}	$1.049 \cdot 10^6$	M	Mega	Mi	Mebi
2^{30}	$1.074 \cdot 10^9$	G	Giga	Gi	Gibi
2^{40}	$1.100 \cdot 10^{12}$	T	Tera	Ti	Tebi
2^{50}	$1.126 \cdot 10^{15}$	P	Peta	Pi	Pebi
2^{60}	$1.153 \cdot 10^{18}$	E	Exa	Ei	Exbi

2^0	1	1
2^1	2	10
2^2	4	100
2^3	8	1000
2^4	16	1'0000
2^5	32	10'0000
2^6	64	100'0000
2^7	128	1000'0000
2^8	256	1'0000'0000
2^9	512	10'0000'0000

Beispiel wie viele Bits benötigt für:

0-32T -1 → 2^5 * 2^40 = 45 bit
0-32T→46 bit
0-10^{12}_d → 40 bit (weil 2^39 = 0.550 * 10^12 was nicht reicht)

Hexadezimal

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Beispiele

2^{15}-1 → 8000_h-1 = $7FFF_h$
2^{16}-1 = FFFF
144 = 90
E8 = 232

Logik

Unäre Funktion Funktion mit einem Parameter (einstellig, *unary function*)
Beispiel: $f(x) = x$
Binäre Funktion Funktion mit zwei Parametern (zweistellig, *binary function*).
Beispiel: $f(x, y) = x \wedge y$
n-äre Funktion Funktion mit *n* Parametern (*n*-stellig, *n-ary function*).

Beispiel: $f(x_0, \ldots, x_{n-1}) = \bigwedge_{i=0}^{n} x_i$

Nulläre Funktion Funktion mit null Parametern (nullstellig, *nullary function*).
Beispiel: $f() = 1$

Nulläre Funktion
1 Möglichkeit: keine Argumente
Unäre Funktion
2 mögliche Argumente: x = 0 oder =1
Binäre Funktion
4 mögliche Argumente(Kartesisches Produkt): (0,0),(0,1),(1,0),(1,1)
n-äre Funktion
2^n Argumentkombinationen.

Regeln für Disjunktion und Konjunktion

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	$\overline{x \vee y} = \bar{x} \wedge \bar{y}$	$\overline{x \wedge y} = \bar{x} \vee \bar{y}$

x	y	$x \vee y$	
0	0	0	$\bar{x}\bar{y}$
0	1	1	$\bar{x}y$
1	0	1	$x\bar{y}$
1	1	1	xy

DNF KDNF
$\Rightarrow \overbrace{\bar{x} \vee y}^{\text{DNF}} = \overbrace{\bar{x}\bar{y} \vee x\bar{y} \vee xy}^{\text{KDNF}}$

KDNF beinhaltet jedes Element 1x

$x \oplus y$ heisst Exklusives Oder:
- Entweder $x = 1$ oder $y = 1$ aber nicht beide gleichzeitig.
- Negation der Äquivalenz: $x \oplus y = \overline{x \leftrightarrow y}$
- Entspricht der Addition zweier Bits ohne Übertrag (Übertrag ist $x \wedge y$)

x	y	$x + y$	$x \wedge y$	$x \oplus y$
0	0	00	0	0
0	1	01	0	1
1	0	01	0	1
1	1	10	1	0

Pro logische Funktion Rückgabewert ein Bit → Viele Bit = Viele Funktionen

Modellierung

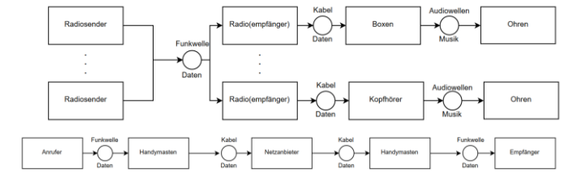
Akteur ist eine aktive Komponente, die Informationen erzeugt oder verarbeitet.

Speicher ist eine passive Komponente, die Informationen speichern kann.

Kanal ○ ist eine passive Komponente, die Informationen übertragen kann.

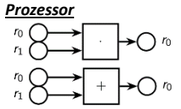
Merke: Nie folgt auf ein Element ein gleiches Element.

Beispiel Modellierung



Wie will man am Modell alle Anrufe in einem Netz abbilden? Es geht schlicht und einfach nicht!(Alle Modelle falsch, aber nützlich)

Prozessor



Register von Prozessor sind mit Bausteinen verknüpft, welche dieser ansprechen kann und somit rechnen kann.

Zyklus

1. Prozessor wählt aktiven Baustein aus
2. Aktiver Baustein liest aus Registern
3. Aktiver Baustein führt Berechnung aus
4. Aktiver Baustein schreibt in Register

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

- Assembler ein Programm, welches textuelle Befehle in Maschinencode übersetzt.
- Die Assemblersprache(assembly) ist die entsprechende Sprache
- Konvention für assembly hängt von Hersteller des Assemblers ab

Maschinencode

Darstellung von Operationen als Zahlen

Befehlssatz

-Menge aller Maschinencodes, welcher ein Prozessor kennt.
-Menge aller Befehle ist von Hersteller zu Hersteller oder sogar von Prozessorfamilie zu Prozessorfamilie unterschiedlich, deshalb ist es enorm wichtig, dass man weiss, mit welchem Prozessor man arbeitet/arbeiten will.
- Maschinencodes können je nach Befehlssatz gleich oder unterschiedlich lang sein

- 1 byte (8 bit): `byte` , `DB` , `RESB`
- 2 bytes (16 bit): `word` , `DW` , `RESW`
- 4 bytes (32 bit): `dword` , `DD` , `RESD`
- 8 bytes (64 bit): `qword` , `DQ` , `RESQ`

Merke: Beim Schreiben der unteren 32-bit(EAX, EBX, etc.) werden die oberen 32-bit auf 0 gesetzt.
Werden Änderungen im ersten 16-bit Block vorgenommen(bsp AX/AL) ändert das die oberen Bits nicht.

	BIG-ENDIAN	LITTLE-ENDIAN
0xCAFEBABE will be stored as	**CA | FE | BA | BE**	**BE | BA | FE | CA**

RAX Accumulator, für einige Rechenoperationen das einzige mögliche Register
RBX Datenpointer
RCX Counter für Schleifen und Stringoperationen
RDX Pointer für I/O-Operationen
RSI, RDI Quell- und Zielindizes für Stringoperationen
RSP Stackpointer, Adresse des allozierten Stacks
RBP Basepointer, Adresse innerhalb des Stacks, Basis des Rahmens der Funktion
R8 – R15 Zusätzliche Register

Linksshift

Rechtsshift

EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D – R15D

AL, BL, CL, DL, DIL, SIL, BPL, SPL, R8L - R15L

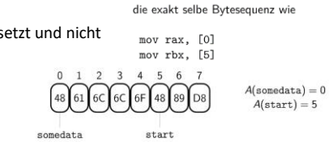
RAX = 64 Bit
EAX = 32 Bit
AX= 16 Bit
AL = 8 Bit
x - extended
l - low

Label
Am Anfang jeder Zeile kann ein sogenanntes Label stehen.
Das Label wird nicht in Bytecode übersetzt und nicht ausgegeben.
Intern assoziiert der Assembler Speicher für das Label

```
mov rax, [somedata]
mov rbx, [start]
```

die exakt selbe Bytesequenz wie

```
mov rax, [0]
mov rbx, [5]
```



db 48	schreibt ein Byte mit dem Wert 48d
db 0x35, 0h21, 049h	schreibt die drei Byte 35h, 21h, 49h
db 'a'	schreibt den ASCII-Code von a = 61h äquivalent zu: db 0x61
db 'Hallo'	schreibt die ASCII-Codes von H, a, l, l und o.

Sprünge
Ändern des RIP(RIP + x). JMP, JNZ, usw. Wir springen in Abhängigkeit von Flags!

cmp a, b jg label ; Jump if A > B	cmp a, b jl label ; Jump if A < B
Cmp a,b Jnz label; Jump if A ≠	Jmp label ; Jump always

Flags

CF(Carry Flag) → 1 wenn Add./Sub. Übertrag erfordert(unsigned)
OF(Overflow Flag) → Gibt Überlauf bei signed Operatioen
CF und OF immer gesetzt, da Prozessor k.P ob signed/unsigned
ZF(Zero Flag) → 1 wenn Resultat 0
SF(Sign Flag) → SF = MSB vom Resultat
PF(Parity Flag) → Gesetzt wenn unterstes Byte gerade Anzahl von gesetzten Bits(Einsen).

Vergleich (Bei signed SF ≠ OF)

Subtraktion a -b
a<b → CF = 1
a = b → ZF = 1
a>b ZF = 0 & CF = 0

bei unsigned «Below» (B) und «Above»(A)
bei signed «Less» (L) und «Greater»(G)

CMP

Vergleichsoperation → Setzt nur Flags ohne Subtraktion

Programm Start

Das Label **_start** repräsentiert bei uns der EntryPoint. Im Executable steht eine Adresse(für start) ab wo das Programm anfängt. Wieso nicht erstes Byte? Weil dort evtl. Data/Befehle

Programm Ende

Das Programm muss sagen, wann es beendet ist. Es macht den **Systemcall exit**(auf Linux rax = 60, rdi = 0 dann syscall)

Privilege Levels

0 – 3: 0 Alle Befehle, Prg 3 → Programme «keinen» Schaden

OS-Handler

Handler für Systemaufrufe. Was in Rax, welche Argumente in anderen Register(bsp rdi, rsi). Adresse des Handlers wird bei Bootvorgang in **IA32_LSTAR** geschrieben(nur bei PL 0 veränderbar)

System Aufruf

1. Programm Funktionscode in RAX(Argumente in andere Register)
2. Programm SYSCALL
3. RIP → RCX, RFLAGS → R11, PL 0, RIP→IA31_LSTAR
4. Ausführung OS-Handler(Syscall)

Beachte! Rückgabewert exit(RDI) ist nur unterstes Byte
Beispiel mov rdi, 300 → echo $? → 44 weil 300-256 = 44

Beispiel Hello World

```
bits 64
msg: db 'Hallo World!'
len: equ $-msg
global _start
_start:
mov rax, 1 ; sys_write
mov rdi, 1 ; stdout
mov rsi, msg ;address of the String
mov rdx, len ; length of the String
syscall ;system call--> OS do what
mov rax, 60 ;sys_exit
mov rdi, 0 ; with code 0
syscall
```

Beispiel Summe 0 + 1 … + n

```
bits 64
global _start
_start:
mov rdi, 6
mov rcx, 0
_sum:
add rcx, rdi
dec rdi
jnz _sum
mov rax, 60
mov rdi,rcx
syscall
```

Beispiel Anzahl Operationen bis gleichgross

```
bits 64
global _start
_start:
mov rdi, 20
mov rcx, 8
mov rbx, 0
_operate:
dec rdi
inc rcx
inc rbx
cmp rdi, rcx
jg _operate
mov rax, 60
mov rdi,rbx
syscall
```

Stack

Stapel von 8 Byte grossen Elemente, der zur Laufzeit Arraygrösse ändert.

Push/ Pop

Push legt ein 8-Byte-Wert oben auf den Stapel.
Mit **pop** kann man einen Wert von Stack nehmen.
Das oberste Element liegt an der niedrigsten Adresse!(Wächst von hohen zu niedrigen Adressen)

RSP - Stack Pointer

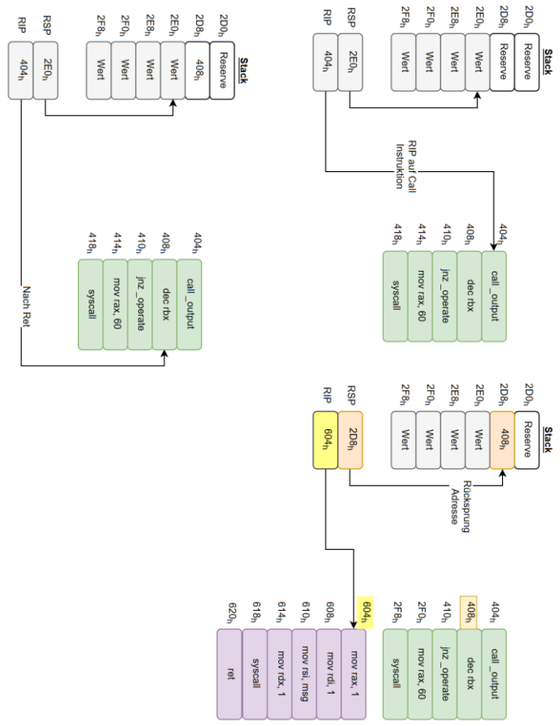
Der Stack Pointer(RSP) zeigt auf das aktuelle Element des Stacks. Wird nun Push/Pop ausgeführt, wird dieser RSP verändert.
Push rax = sub rsp, 8 & mov [rsp], rax
Pop rax, mov rax, [rsp] & add rsp, 8 // erhöhen →Stack kleiner

Call

Call **Speichert** die **Rückspring-Adresse** auf dem Stack und Springt dann. Das bedeutet die Adresse der nächsten Instruktion wird auf dem Stack abgelegt und dann wird «die Funktion» aufgerufen.

Ret

Die Gegenteilige Funktion von Call heisst Ret(return). Diese kann man in der «aufgerufenen Funktion» aufrufen. Dann **springt** man zur **Adresse zurück**, welche man vorhin mit Call auf den Stack gepusht hat. Dann verkleinert man den RSP wieder.



Beispiel Print Stars	_output:

bits 64
global _start
msg: db '*'
_start:
mov rbx, 6
_operate:
call _output
dec rbx
jnz _operate
mov rax, 60
mov rdi,0
syscall
	mov rax, 1 ; sys_write
	mov rdi, 1 ; stdout filedescriptor
	mov rsi, msg ;address of the String
	mov rdx, 1 ; length of the String
	syscall
	ret ;Back to _operate

Calling Convention

Da Call keine Parameter oder Rückgabewerte hat und keine Register, Stack oder Flags sichert braucht es Vereinbarungen.

Calling Convention sind Vereinbarungen zwischen Aufrufen(Caller) und aufgerufener Funktion(Callee): Wo Argumente, Wo Rückgabewerte, welche Register bearbeitet, etc.

Von OS zu OS, Compiler zu Compiler verschieden

Hauptgrund warum verschiedene Compiler/OS inkompatibel

Performance des Gesamtsystems abhängig wegen Tradeoff

Mehr Register -> weniger Register für Rechnungen

Mehr Stack -> mehr langsame Speicherzugriffe Kein Optimum

Lokale Variabeln / Globale Variabeln

Lokle Variabeln: liegen auf dem Stack

Globale Variabeln: fixe Adresse im Speicher(Label)

Quelle:

```
// Ein Einzeilen-Kommentar
/* Ein Block-Kommentar mit
   mehr als einer Zeile */

puts ("Hallo␣\
HSR");
```

Ergebnis:

```
puts ("Hallo␣HSR");
```

Beispiel Wörter Ausgabe	_iterieren:
	call _ausgabe
	inc r9
	cmp r9, [anzahl]
	jl _iterieren
	mov rax, 60
	mov rdi, 0
	syscall
	
	_ausgabe:
	mov rax, 1
	mov rdi, 1
	mov rsi, r8
	mov rdx, [len + r9 * 8]
	add r8, rdx
	syscall
	ret

bits 64
global _start
anzahl: dq 3
strings: db 'String1', 'plz ',
'Dellsperger'
len: dq 7, 4, 11
_start:
mov r8, strings ;erste Adresse von s
mov r9, 0 ;String counter	

Die Sprache C

Weniger Aufwand, Register nicht selber verwalten, einfach Schleifen, Verzeigungen, Sprünge, Nachvollzierbarkeit, keine Unterscheidung signed oder unsigned→ Evolution Assembler

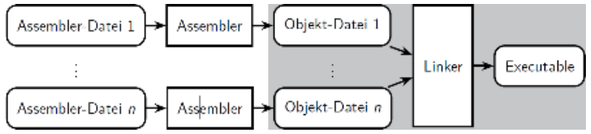
Toolchain(logisch gesehen)

Assembler: Macht aus (mehreren) Assembler Dateien (mehrere) Objekt Dateien.

Linker: Macht aus mehren Objekt-Dateien eine Executable.

Präprozessor: Entfernt Kommentare, Tokenization, Direktiven

Compiler: Kompiliert reinen C-Code in Assembler Dateien



Objekt Dateien(Binärrepresentation einer Assembler-Datei)

Enthält Binärsequenzen für Daten und Maschinenbefehle

Enthält Symbole(Labels) in der Symboltabelle

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

10 <f>:
10:  48 8b 04 25 00 00 00 00          mov     rax, [0x0]
18:  48 8b 1c 25 00 00 00 00          mov     rbx, [0x0]
20:  48 39 d8                         cmp     rax, rbx
23:  75 05                            jne     +5 -> 2A <no_call>
25:  e8 00 00 00 00                   call    +0 -> 2A <no_call>
2A <no_call>:
2A:  c3                               ret
```

```
10: 48 8b 04 25 00 00 00 00          mov     rax, [0x0]
18: 48 8b 1c 25 00 00 00 00          mov     rbx, [0x0]
```

Das nebenstehende Programm assemblieren wir mit:

```
nasm -f elf64 ex0600000.asm -o ex060000.o
```

Executable(Ausführbare Datei)

Symbole in den Objektdateien haben noch keine festgelegte Adresse..

Linker weist jedem Symbol eindeutige Adresse zu. Er erzeugt aus mehreren Objekt-Dateien eine einzige Binärsequenz, die Executable. Dann nicht mehr unterscheidbar, aus welcher Objekt-Datei welches Symbol.

```
ld -o syscall syscall.o
```

C Sprachebenen

Präprozessor → Bearbeitet includes und defines

Basiskonstrukte → Grundgerüst vom Programm, Variabeln, Schleifen, etc

Standardbibliotheken → stellen Funktionen und Typen zur Verfügung, welche Basis-Funktionalität beinhalten.

Globale Variabeln(Typ + Bezeichner [Optional initalisiert])

Werden Exportiert→ wie globale Elemente in Assembler

Extern → Compiler nimmt an, dass Variable in anderer Datei ist

Ohne Extern: Variabel wird in Objekt-Datei erstellt

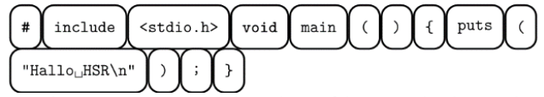
Aller ausführbarer Code muss in Funktion stehen.

Int main () erste aufgerufenen Funktion

Präprozessor

1.Durchlauf: Entfernen von Kommentaren und Zeilenumbrüchen bei fortgesetzten Zeilen:

2.Durchlauf: Zusammenführen Header + Tokenbildung



Bezeichner: Beginnt mit Buchstaben oder _ gefolgt von Buchstaben, _ oder Ziffern.

a, _, _a, a0, _0, _a0, _0_0, _____, int, for, size_t

Zahl: Beginnt mit Ziffer(oder Punkt Ziffer) gefolt von Ziffern, Buchstaben, _ oder . oder Exponenten.

0, 1, 0123, 0x1234, .05, 0_.0 (nicht gültig in C), 0xE+12

String und Character: alles zwischen «» und ‘’

"Hallo␣HSR", "Hallo␣'HSR'", "Hallo␣\"HSR\"", 'H', '\'', '"'

Punctuator(achtung greedy → grösstmögliche Token):

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

3.Durchlauf Direktiven bearbeiten(include, define, if, else, endif)

#include → Kopiert tokenisierten Inhalt von Include-Datei

Präprozessor macht aus mehreren Dateien eine Datei(Translation Unit) danach geht diese abgearbeitete Datei zum Compiler

#define → Ersetzt jeden passenden Token mit neuem Wert

Der Präprozerssor arbeitet ein define nach dem anderen ab, bis alle durch sind:

BSP. #define XYZ 123, #define 123 700

1. → int y = XYZ; → int y = 123;
2. → int y = 123 → int y = 700

Nach Ersetzungen geht er nochmals defines von oben durch

datei.c	x.h
#include "x.h" #define z w int y = x;	#define x z

Lösung y = w → Zuerst y=z → dann y = w

Objekte & Wert

Objekt: zusammenhängender Speicherbereich

Wert: Bedeutung des Inhalts eines Objektes

Jede globale Variabel Objekt, aber nicht jedes Objekt Variabel

Typen

Basistypen(int, float), Abgeleitete Typen(arrays, etc), ENUMS

Typen beschreiben Objekt bzw. Eigenschaften, bsp sizeof(T)

Vorteil: auto. Grösse, richtige Operationen(signed / unsigned), Fehler melden(beispiel Typen nicht kompatibel)

Auf Maschinencode-Ebene nur Adressen, keine Variabeln

Basistypen

- `signed char` hat mindestens 8 Bit und die kleinsten Objekte mit eigener Adresse
- `short int` hat mindestens 16 Bit
- `int` hat mindestens 16 Bit und ≥ `short int`, soll der natürlichen Grösse der Architektur der Ausführungsumgebung entsprechen.
- `long int` hat mindestens 32 Bit und ≥ `int`
- `long long int` mindestens 64 Bit und ≥ `long long`

Ebenfalls unsigned char, unsigned bool und void

Abgeleitete Typen

- Array-Typen: Mehrere Elemente vom gleichen Typ nacheinander
- Struct-Typen: Mehrere Elemente möglicherweise verschiedener Typen
  - Pointer-Typen: Adresse eines Objekts
  - Funktions-Typen: Adresse einer Funktion

Pointer

Adresse, welche auf ein Objekt zeigt.

Definition: void* oder int* oder auch int**

Ausdrücke

Ausdrücke sind Kombination von Operanden und Operatoren

Besteht aus verschiedenen Teil ausdrücken und hat einen Typ

Jeder Ausdruck ausser void hat einen Wert

Conditionals(if)

Ausdrücke, welche interpretiert werden, dass **0 falsch** ist und **jeder andere Wert wahr**.

Referenz- und Dereferenzoperator

Der Operator «**&**» erzeugt die **Adresse** eines Audruckes

Pointer(*)werden jeweils Adressen(&) zugewiesen, sprich der eigentliche Wert von T* ist &a

*& heben sich gegeneinander auf» → *&a = a = &*a

Achtung bei &*a muss a Pointer sein, weil Adresse verlangt

Arrays

Int b [5] [= {0,1,2}], verbleibende Elemente mit 0 initialisiert

Wird ein Array mit einem Bezeichner angesprochen, ist immer die erste Adresse des Arrays gemeint

→ Es können auch Pointer verwendet werden:

```
int a = 1;
int *pointer = &a;
int **pointer2 = &pointer;
printf("%i", *pointer); //1
printf("%i", **pointer2); //1
```

```
int a [5] = {1,2,3};
int *p1 = a;
int *p2 = &a[1];
printf("%i und %i", *p1, *p2);
```

```
char a [5] = "Hallo";
char *s1 = a;
char *s2 = &a[1];
printf("%s", a); //Hallo
printf("%c", *s1); //H
printf("%c", *s2); //a
```

```
char *a = "Hallo";
char *s1 = a;
char *s2 = &a[1];
printf("%s", a); //Hallo
printf("%c", *s1); //H
printf("%c", *s2); //a
```

Beispiel extern

```
int fd = 1;//filediscreptor 1 = stdout
int anzahl = 3;
char *s = "Hallo" "und" "Ciao";
int len[3] = { 5, 3, 4 };
for (int i = 0; i < anzahl; i++)
  {
    write (fd, s, len[i]);
    s = s + len[i];
  }
```

```
void
main ()
{
  int len = 0;
  while (*s != '\0')
    {
      int len = strlaenge (s);
      write (fd, s, len);
      s = s + len + 1;
    }
}
```

```
int strlaenge (char *str)
{
  int counter = 0;

  while (str[counter] != '\0')
    {
      counter++;
    }

  return counter;
}
```

```
#include "strlaenge.h"
extern int strlaenge (char *str);
```

Funktion wurde in andere Datei verschoben.

Strukturierte Variabeln / Structs

```
struct
{
  int x;
  int y;
} t;

t.x = 1;

printf ("%i", t.x);
```

Eine Strukturierte Variabel beinhaltet verschiedene Member, welche eizeln angesprochen werden können:

Ein Struct braucht gleich viel Speicher, wie beide einzelnen Variabeln.

Jedes Member hat eigene Adresse.

Structs können auch ***Variabeln*** sein:

```
struct database
{
  char * name;
  int tel;
};

struct database person1;
person1.name = "Hans";
person1.tel = 123;
printf ("%s, %i", person1.name, person1.tel);
```

```
struct database
{
  char * name;
  int tel;
};

struct database person1 = {"Hans", 123};
printf ("%s, %i", person1.name, person1.tel);
```

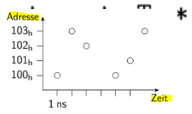
Pointer auf Struct:

```
struct T
{
  int x;
  int y;
};

struct T *t;
t->x = 1;
t->y = t->x;
printf ("%i", t->y);
```

Mit dem «->» Operator können auf die Adressen der Member zugegriffen werden.

Memberwise Copy

```
struct T t, u;
t = u;

struct T *pt = &t;
      T *pu = &u;
```



Complete / Incomplete

Vollständig(complete) wenn Compiler genug Infos, um Grösse dieses Typs zu bestimmen, andernfalls unvollständig

Forward-Deklarationen

```
struct Folder;      // Forward-Deklaration
struct File
{
    struct Folder *parent;      // Poi
    char name[256];             // Gro
};                              // Typ
struct Folder
{
    struct File * file[256];    // Arr
};
```

Unions

Unions werden in Structs definiert. Ihre Member beginnen alle mit Adresse der Union. In einer Union hat es mehrere Structs.

Mit Unions lassen sich Casts vermeiden

```
union U {
    int kind;
    struct { int kind; char value[256]; } str;
    struct { int kind; long long value; } ll;
};
```

Funktionen(Call by Value)

```
void f ();   ← Argumente ohne Wirkung
void g (void);  ← keine Argumente

int f (int *x)
{
    *x = *x + 1;
    return *x * 4;
}
int p = 7;
int q = f (&p);      // p wird 8, q wird 32
```

Main Funktion

Die Main Funktion hat ein Argument Counter und ein Char-Array von Argumenten, welche der Funktion mitgegeben werden können. Da ein Array immer nur als Pointer übergeben werden kann, muss die Anzahl(argc) einzeln übergeben werden.

1. OS übergibt Programm Kommandozeilenargumente
2. Argv ist ein Pointer auf das erste Element eines Arrays von String Pointern
3. Agrv[0] beinhaltet den Pointer auf den Programmnamen
4. Agrv[1] – [argc-1] enthlt Pointer auf Argumente

Lokale Variabeln

```
int * f (int **x)
{
    int y = 123;
    *x = &y;
    return &y;   → Blödsinn beachte Lebensdauer
}
```

Liegen auf dem Stack, innerhalb von Funktion definiert.

Printf

%s → String
%i → int
%p → pointer
%c → Char

Speicher

Grundsatz

Es gibt kein System, das optimal für alle Anwendungszwecke eingesetzt werden kann. Jede Optimierung geschieht auf einen oder mehrere Anwendungsfällem hin. Ein perfektes System gibt es nicht!

Software-Basis-Modell

CPU lädt Befehlssequenzen aus Hauptspeicher und führt die aus

Ein Befehl kann, Daten zwischen Hauptspeicher und Registern verschieben, Auf Registern rechnen oder den Befehlspointer verändern

Compiler generiert Befehlssequenzen, CPU führt diese aus

Speicher-Programmier-Modell

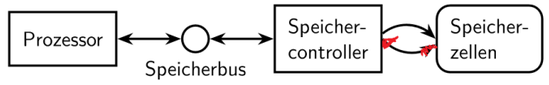
Früher einfachen Aufbau(**weil Proz gleichschnell wie Hauptspeicher**):

Aus diesem einfachen Prinzip haben sich Compiler und OS gebildet. **Compiler→ Generiere Code**, um Werte zwischen Hauptspeicher und Prozessort zu Transportieren

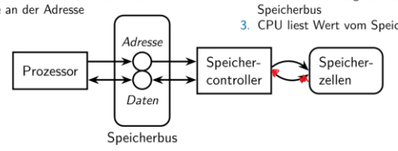
OS: **Weise** den **Programmen Speicher**(Bereiche) zu

Hauptspeicher(RAM / ROM)

Speicher mit Speicherzellen, welche Wert speichern können. Zugriff über Speichercontroller, Kommunikation zwischen Speichercontroller und Prozessor geht über den Speicherbus



- Wert an Adresse speichern:
  1. CPU legt Adresse und Wert auf den Speicherbus
  2. Speichercontroller schreibt Wert in Zelle an der Adresse
- Wert von Adresse lesen:
  1. CPU legt Adresse auf den Speicherbus
  2. Speichercontroller liest Wert aus Zelle an Adresse und legt sie auf den Speicherbus
  3. CPU liest Wert vom Speicherbus



Speicherhiearchie

Moderne Computer brauchen aus physikalischen, logischen und ökonomischen Gründen eine Speicherhierarchie, in der verschiedene Sorten Speicher eingesetzt ist.

Physikalisch: Licht, Hitze(mehr Transistoren desto heisser)

Logisch(Adressierung ist logarithmisch, mehr = mehr Zeit)

Ökonomisch: Grundsatz desto CPUnäher Speicher desto teurer, komplexer, schneller, weniger fehleranfällig

- **Flüchtiger Speicher**(ohne Strom weg)
  - CPU-Register
  - CPU Cache(Level)
  - Hauptspeicher(RAM / ROM)
- **Nichtflüchtiger Speicher**(behält ohne Strom)
  - Festplatte(HDD/SSD)
  - Externe Datenträger(CD, Flashdrive, etc)
  - Netzwerkspeicher/Cloudspeicher
  - Archiv/Backup(Tape)

Vorteile höherwertiger Speicher

- Näher an CPU
- Höhere Zugriffs und Transfergeschwindigkeit
- Weniger Fehler(anfälligkeit)
- Geringere Sicherheitsanforderungen(bei Zugriff verloren)

Vorteile niederwertiger Speicher

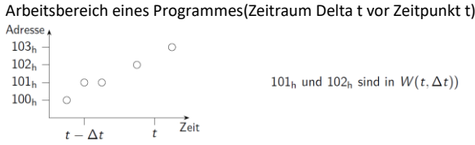
- Niederiger Preis pro Byte
- Höhere Kapazität
- Grössere Transfereinheiten

Beispiel

Es gibt beispielsweise Cache, Hauptspeicher und Sekundärer Speicher. Programme und Daten liegen im Hauptspeicher und werden vom Cache-Controller automatisch in den Cache geladen, um den Zugriff durch die CPU zu beschleunigen. Festplatten werden zum einen dafür verwendet Programme und Daten dauerhaft zu speichern und zum anderen als Auslagerung für Hauptspeicher, um diesen zu erweitern. Das wird vom Betriebssystem gesteuert. Grundsätzlich kann man sagen, je weiter weg ein Speicher von der CPU ist, desto billiger, fehleranfälliger und langsamer ist er. Vergleiche dazu die HDD.

Lokalitätsprinzip

Arbeitsbereich eines Programmes(Zeitraum Delta t vor Zeitpunkt t):



Lokalitätsdiagramm

Im Lokalitätsdiagramm werden die Zugriffe aus Speicherstellen im zeitlichen verlauf aufgezeigt.

Räumliche Lokalität: Wird auf eine bestimmte Adressse im Hauptspeicher zugegriffen, so ist die Wahrscheinlichkeit recht hoch, dass der folgende Zugriffe auf eine Adresse in der Nachbarschaft erfolgt. Im Speicher wird dies genutzt in dem man immer Datenblöcke verschiebt.

Zeitliche Lokalität: Wird auf eine bestimmte Adresse im Hauptspeicher zugegriffen, so ist die Wahrscheinlichkeit hoch, dass in naher Ukunft wieder darauf zugegriffen wird. Im Speichersystem will man also die zuletzt zugegriffenen Daten auf der schnellsten Stufe der Hierarchie halten.

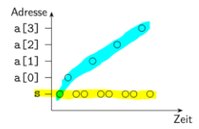
➔ Wenn ich den Arbeitsbereich kenne, kann ich auf zukünftigen Arbeisbereich schliessen

➔ Ohne dieses Prinzip, bräuchte man nur die langsamste Speicherstufe

Berechnung des Durchschnitts:

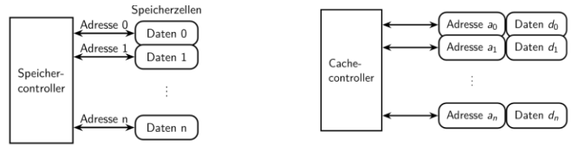
```
for (i = 0; i < n; ++i)
{
    s = s + a [i];
}
m = s / n;
```

- Zeitliche Lokalität auf der Variablen s
- Räumliche Lokalität auf dem Array a



Cache

- Zwischenspeicher, welcher kleiner als Hautpspeicher ist und näher an CPU→schnellere Zugriffszeiten.
- **Neben Daten auch Teil der Adresse der Zellen speichern**
- Lokälitätsprinzip→ Daten in Cache(schnelle, viele Zugriffe)
- Cache zwischen Prozessor und Hauptspeicher: Ohne Sie wären Computersysteme praktisch unbrauchbar(weil langsam).



➔ Cache aufwendig und teuer

Cache Hit: Gesuchte Adresse ist im Cache.

Chache Miss: Gesuchte Adresse ist nicht im Cache.

Berechnung Mittlere Zugriffszeit

Tc Zugriffszeit auf den Cache.

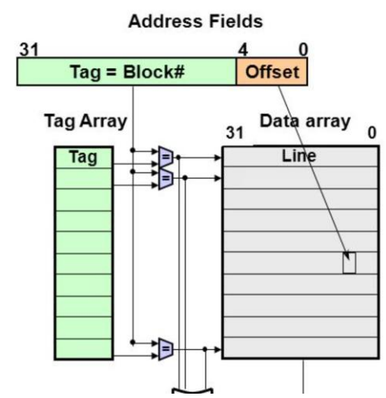
Tm Zugriffszeit auf den Hauptspeicher.

pc Wahrscheinlichkeit eines Cache Hits.

Mittlere Zugriffszeit: Erwartungswert E(T) der Zugriffszeit T:

$$E(T) = p_C \cdot T_C + (1 - p_C) \cdot T_M$$

Fully Associative Cache(FAC)



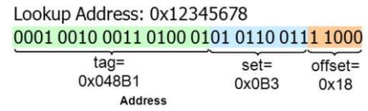
Adressen sind aufgeteilt in Tags und Offset.

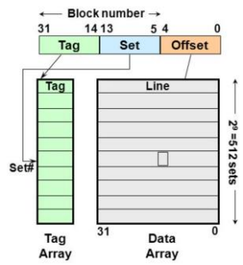
Man sucht die Cachezeile mit dem richtigen Tag und in diesem dann den richtigen Offset.

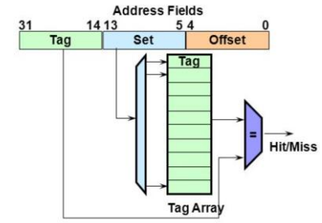
Beste Cache-Leistung, aber aufwendige Hardware(teuer).

Viel Vergleichsbausteine

Direct Mapped Cache

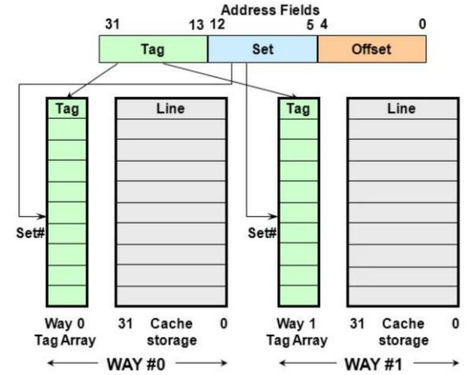






Einfachere Implementierung, viele Kollisionen, schnell

k-Way Set Associative Cache:



k-verschiedene DMCs, Kompromiss zwischen FAC und DMC→ Weniger komplex als FAC, weniger Kollisionen als DMC aber genau so schnell wie DMC

Berechnungen

Gegeben 2 Level Cache:

a) 64KB 4-way set associative
b) 4 MB FAC

Cache: 64Byte Cachezeilenlänge und 64Bit-Adressen. 16 GB Hauptspeicher

1. Wie viele Cachezeilen Pro Stufe und Pro Way
   - I. DMC
     - i. $(Gesamt\ Cache) : (Cachezeilenläge) \rightarrow 2^{16} : 2^6 = 2^{10}$(pro Stufe 1)
     - ii. $(Cachezeilen\ insgesamt) : (Anzahl\ Ways) \rightarrow 2^{10} : 2^2 = 2^8$
   - II. FAC
     - i. $(Gesamt\ Cache) : (Cachezeilenläge) \rightarrow 2^{22} : 2^6 = 2^{16}$
2. Wie viele Bits benötigt ein Tag in jeder Stufe:
   - I. DMC
     - i. $Adresse\ 64bit = 2^6 = 6\ bit\ Offset$
     - ii. $Anzahl\ Cache\ Zeilen\ pro\ Way\ 2^8 = 8\ bit\ Set$
     - iii. $Bits\ pro\ Tag = Adressgrösse - Offset - Set$ →**64 – 6 – 8 = 50 Bit**
   - II. FAC
     - i. $Bits\ pro\ Tag = Adressgrösse - Offset$ → **64 – 6 = 58Bit**
     - ii. Im FAC wird des gesamte Tag gespeichert, weil jede Zeile in jedem Eintrag gespeichert werden kann
3. Overhead/zusätzlicher Speicher für Tags im Cache nötig?
   - I. DMC
     - i. $Overhead = Anzahl\ Bits\ pro\ Tag * Cachezeilen\ pro\ Stufe \rightarrow 2^{10} * 50Bit$
   - II. FAC
     - i. $Overhead = Anzahl\ Bits\ pro\ Tag * Cachezeilen\ pro\ Stufe \rightarrow 2^{16} * 58Bit$
4. Speicherstellen des Hauptspeichers auf dieselben Cacheeinträge?
   - I. DMC
     - i. $Hauptspeicher * Way - Anzahl : Grösse\ Cache = 2^{34} * 2^2 : 2^{16} = 2^{20}$
     - ii. 2 MB Hauptspeicher teilen sich 4 Stellen(anzahl ways) im Cache
   - II. FAC
     - i. Keine Stellen geteilt!

Gegeben 8 MB Hauptspeicher, 8KB FAC, 64B Zeilenlänge
1 Anzahl Cachezeilen = 8KB / 64 = 128 Cachezeilen
2 minimale Anzahl an Bits je Tag:
Hauptspeicher ergibt Grösse Adresse: 8 MB → 23 Bit Adressen
Bit pro Tag: Adresse – Offset = 23 – 6 = 17 Bit
3 Zusätzlicher Speicher für Tags
17 Bit * 128 = 2176 Bit = 272 Byte

Heap

- Speicherbereich für **dynamischen Speicher**
- Vom **OS verwaltet**
- Beliebige Grösse(der Speicherblöcke)
- Beliebiger Zeitpunkt Reservation/Freigabe

Speicherfreigabe

Explizit: Programmierer, wann Speicher freigegeben wird
Implizit: Speicher auto. Freigegeben, wenn nicht benötigt
OS nur explizite Speicherverwaltung, implizit App(JVM, Py, …)
Malloc(s): allozier Speicherblock grösse S
Free(*p): gibt Speicherblock frei, der an Adresse p beginnt
Problem expliziter Freigabe: Speicherlacks wenn nicht mehr gebrauchter Speicher nie freigegeben wird.
→malloc & free gehören zusammen wie Klammern, sodass es «keine» Speicherlacks gibt
→Programmierer zuständig für Freigabe/kein OS Compiler Hilfe

Interne Fragmentierung

Heap reserviert grösseren Speicherblock als angefragt wurde, der zusätzliche Speicherbereich wird vom Programm nicht verwendet(Verschwendung)

Externe Fragmentierung

Programm reserviert immer wieder Speicher und gibt ihn unregelmässig wieder frei.

- → Über Längere Zeit entstehen kleine Löcher, grössere Speicherreservierungen können nicht bedient werden obwohl die Summe der Löcher genug gross wäre

Heap Implementierungen(geheim, Einsatzgebiet, Forschung)

- Variable Zuordnungsgrösse(Blockgrösse = 1)
- Mehrfache fester Blockgrösse
-Grössenklassen(verschiedene Listen von 2^x freien Blöcken)
- Buddy-System
- Kombinationen der gennanten Basis-Varianten

Mehrfache fester Blockgrösse

Blockgrösse x und Mehrfache von x → je kleiner, desto weniger interne Fragmentierung aber mehr Aufwand/Metadaten
Metadaten dezentral(bei Speicherblöcken, Überlaufschutz mit magic Numbers, kein Schutz vor Programmierfehler)
Metadaten zentral(jenseits von Speicherblöcken) Kein Überlauf, aber zusätzlicher Speicherplatz für Metadaten

Speicherblock-Verwaltung

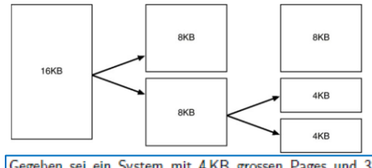
Bitliste(Ein Bit pro Block 0 = Block frei, 1 Block verwendet), suche linear und langsam, he weiter hinten Platz frei ist, desto länger
Verkettete Liste(frei,32 -> besetzt, 32 -> frei 64), sehr schnelles zusammenführen, wenn a(32) frei und b(32) frei -> ab(64)

Suchalgorithmen(eingefügte Blöcke werden aufgeteilt!)

First-Fit → erste passende Lücke am Anfang
Next Fit: → erste passene Lücke nach zuletzt reserviertem Bereich
Best-Fit→Durchsucht alle Lücken, wählt kleinste passende aus
Worst-Fit → Durchsucht alle Lücken, nimmmt grösste Lücke

Buddy System

Speicheranfragen → nächste 2er Potenz, wenn zu klein → Buddy



Bsp: Gesamt 16 KB
Braucht 5 KB → zwei Buddys mit jeweils 8 KBs
Dann nochmal 3 KB

→ Nochmals 2 mit 4 KB. Freigabe umgekehrt!

Gegeben sei ein System mit 4 KB grossen Pages und 32 KB virtuellem Adressraum. Der Hauptspeicher betrage 64 KB.

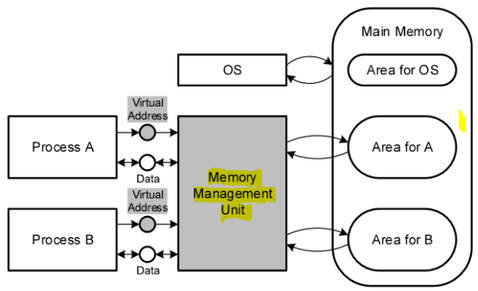
1.Wie viele Bits benötigt eine virtuelle Adresse?
32KB virtueller Adressraum = 2^{15} = 15 Bits (Länge) es gibt 2^{15} Adressen.
2.Wie viele Bits benötigt eine reale Adresse?
64KB Hauptspeicher = 2^{16} = 16 Bits.
3.Wie viele Bits davon sind Offset?
Page ist 4KB gross = 2^{12} = 12Bit Offset. →gleiche Adresse wie Page.
4. Was sind die kleinste und grösste virtuelle Adresse?
Kleinste = 0 / Grösste = 2^{15}-1 → 8000_h-1 = $7FFF_h$ (mit 32KB rechnen)
5.Was sind die kleinste und grösste reale Adresse?
Kleinste = 0 / Grösste = $FFFF_h$
6.Wie viele Pages gibt es?
Grösse Adressraum = 32KB «geteilt» Page Grösse 4KB = 32/4=8 Pages.
7.Wie viele Page Frame gibt es?
Hauptspeicher 64KB «geteilt» Page Grösse 4KB = 64/4 = 16 Page Frames.
8.Wie viele Bits benötigt eien Frame Nummer?
Es gibt 16 Frames = 2^4 = 4 Bits braucht es. (Frames 2er Potenz rechnen)
9.Wie gross ist die Page Table pro Prozess ohne Statusbits mindestens?
In PageTable sind x Pages. Hier 8 Pages. FrameNr 4 Bits→ 8*4Bits = 32Bits
10. Wie gross ist sie wenn Haupspeicher vertoppelt wird. Also 128KB?
128KB = 2^{17} = 17Bits. 128KB/4KB=32 Frames = 2^5 →5Bits.
8*5Bits=40Bits.
11.Wie gross ist sie wenn virtuelle Adressraum verdoppelt wird (64KB)?
64KB virtueller Speicher/4KB(PageGrösse) = 16 Pages.
16*4Bits(FrameNr)=64Bits.

Ein System habe 16 MB Hauptspeicher, 32-Bit breite virtuelle Adressen und 4 KB grosse Pages. Es soll eine zweistufige Page-Table verwendet werden.

1.Wie viele Page Tables kann es in diesem System pro Prozess geben?.
Zeilen = Grosse Directory : Breite Eintrag 2^{12} : 2^2(32 bit) = 2^{10} Page Tables
2.Table soll gleich gross wie Page sein mit 32Bit Einträge. Wie viele Pages?
Jede Page Table hat 1K Einträge. Pro Eintrag eine Page(1K). == 1K*1K=1M
3.Wie viele virtuelle Adressen kann es in einem Prozess geben?
Jede Page hat 4K virt. Adressen über alles → 4K*1M(Pages) = 4GB v.A.

Virtueller Speicher

- → ***Prozesse***
- → Prozesse bekommen von OS Speicher zur Verfügung¨
- → OS zuständig, dass Prozesse sich nicht stören, Daten austauschen, etc.
- → Lösung: Prozess kennt nur virtuelle Adresse, MMU übersetzt virtuelle Adresse in physische Adresse, OS konfiguriert MMU pro Prozess:



Pages

Virtueller Adressraum besteht aus Pages, Page jedoch kein Speicher, benötigt dazu Speicherort(Hauptspeicher/Sekundärspeicher)

(Page)Frames

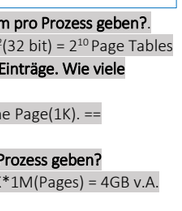
Hauptspeicher besteht aus Frames. In ein Frame passt genau eine Page. Daraus folgt ein Frame = Page.

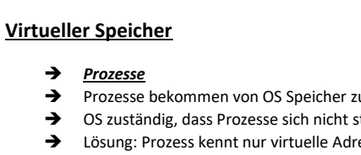
Virtueller Adressraum / Page Table

- Pro Prozess ein virtueller Adressraum
- Pro Prozess ein Pagetable(Mapping Tabelle)
- OS verwaltet, welche Pages wann wo liegen müssen, MMU kennt nur Hauptspeicher→ MMU kann nur sagen ob Page, resp. Zu Page gehörendes Frame im Hauptspeicher

MMU

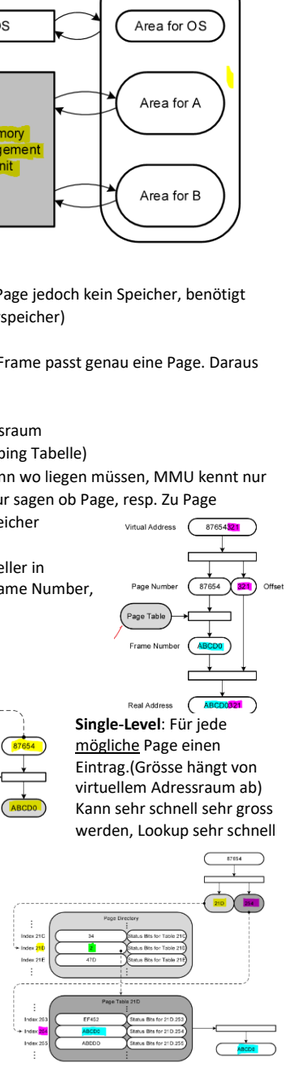
Generelle Aufgabe: Übersetzten von virtueller in physikalische Adresse. Page Number → Frame Number, Offset bleibt gleich.





Single-Level: Für jede mögliche Page einen Eintrag.(Grösse hängt von virtuellem Adressraum ab) Kann sehr schnell sehr gross werden, Lookup sehr schnell

Two Level: Page Number wird aufgeteilt: in Directory Index & Page Table Index
1)Page Table in Directory finden
2) Table Index in Page Table finden
Viele Page Tables, Page Directory zeigt auf Page Tables



Status-Bit Used(P Bit)
Gibt an, ob Page überhaupt benutzt(Used = 1 oder 0)

Interprozesskommunikation(IPC)

Shared Memory: Prozesse teilen Speicher, kein Schutz
Message Passing: OS kopiert Daten, Sicher, Overhead

Berechnungen

Speicher Two Level :(t + 1) *4KB 1 Table pro 1KPages
Speicher Single Level (Adressbereich : Pagegrösse * Zeilengrösse)

Paging

Paging nennt man die Verwaltung von pageorniertierem Speicher(ist der zugehörige Frame im HS oder SS), Laden von Pages, etc.

Dreschen

Dreschen nennt man den Vorgang, wenn die ganze Zeit Pages aus dem Sekundärspeicher geholt werden müssen. Gründe dafür sind i.d.R Hauptspeicher zu klein oder zu viele Prozesse parallel laufen

Paging-Strategien

Da das OS beim Paging ständig Pages in der Speicherhierarchie verschiebt gibt es verschiedene Strategien:
Ladestrategie: Wann und welche Pages laden
Entladestrategie: Wann modefizierte Pages zurückschreiben
Veränderungsstrategie: Welche Page entfernen, wenn HS voll

Ladestrategien

Demand Paging: Laden auf Anfrage(Minimaler Aufwand, Lange Wartezeit)
Prepaging: vor Verwendung geladen(«vorhersehen» →Analyse)
Demand Paging mit Prepaging: Pages + Nachbarn geladen

Entladestrategien

Demand Cleaning: Page zurückgeschrieben, wenn Frame wiederverwendet werden soll. Minimaler Aufwand, Lange Wartezeit(Page muss vor Gebrauch noch geschrieben werden)
Precleaning: Modifizierte Pages frühzeitig in sekundär Speicher→ Schreiben evtl. sinnlos, wenn wiederverwendet, kurze Wartezeit
Page Buffering: Zwischenspeichern von veränderten und unveränderten P. (+/- wie Precleaning, +schelle Auswahl)

Verdrängungstrategien(massiver Einfluss auf Performance)

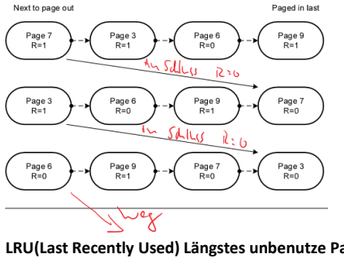
Page Fault, wenn Page nicht in Hauptspeicher
Optimal ersetzte Pages, die am längsten nicht kommt

Schritt	1	2	3	4	5	6	7	8	9	10	11
Referenz auf	2	5	6	2	1	2	5	3	2	7	2
Hauptspeicher	2	2	2	2	2	5	2	2	2	2	2
		5	5	5	5	2	5	3	3	3	3
			6	6	1	1	1	5	5	7	7
Sekundärer Speicher					6	6	6	1	1	5	5
								6	6	1	1
										6	6
Page-Fault	F	F	F		F			F		F	

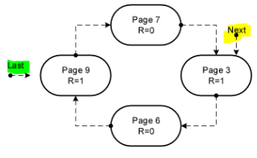
FIFO(First in First Out)

Schritt	1	2	3	4	5	6	7	8	9	10	11
Referenz auf	2	5	6	2	1	2	5	3	2	7	2
Hauptspeicher	2	5	6	6	1	2	5	3	3	7	2
		2	5	5	6	1	2	5	5	3	7
			2	2	5	6	1	2	2	5	3
Sekundärer Speicher					2	5	6	1	1	2	5
								6	6	1	1
										6	6
Page-Fault	F	F	F		F	F	F	F		F	F

Second Chance(2. Chance[siehe R])



Clock(Next Spring)



LRU(Last Recently Used) Längstes unbenutze Page[fast Optima]
NFU(Not Frequently Used)

Schritt	1	2	3	4	5	6	7	8	9	10	11
Referenz auf	2	5	6	2	1	2	5	3	2	7	2
Hauptspeicher	2	5	6	2	1	2	5	3	2	7	2
		2	5	6	2	1	2	5	3	2	7
			2	5	6	6	1	2	5	3	3
Sekundärer Speicher					5	5	6	1	1	5	5
								6	6	1	1
										6	6
Page-Fault	F	F	F		F		F	F		F	

1. Not Frequently Used

Page 1 Zugriffe	Page 1 Counter	Page 2 Zugriffe	Page 2 Counter	Zu entfernende Page
30	7	1	1	Implementierungsfehler
20	2	0	1	Page 2
0	2	2	2	
45	3	3	2	
66	5	2	4	
0	1	1	3	Page 1
78	1	2	1	Page 1
1	1	1	1	

Zeitpunkt	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57
Page 1 R	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Page 1 Zeitstempel	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57
Page 1 Alter	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56
Page 2 R	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Page 2 Zeitstempel	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57
Page 2 Alter	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56
Zu entfernende Page																																																																	

2. Not Frequently Used with Aging

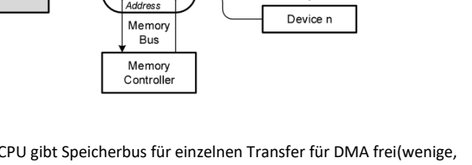
Page 1 Zugriffe	Page 1 Counter	Page 2 Zugriffe	Page 2 Counter	Zu entfernende Page
30	1000	1	1000	Page 2
20	1100	0	0100	Page 1
0	0110	2	1010	Page 1
45	1011	3	1101	Page 1
66	1100	2	1110	Page 1
0	0110	1	1011	Page 1
78	1001	2	1101	Page 1
1	1100	1	1110	Page 1

Ein- & Ausgabe

Eingabegerät: physikalische Sachverhalten→ Daten
Ausgabegerät: Daten→ physikalische Vorgänge
Einige Geräte auch beides(SSD, Festplatten, Touchscreen)

Datenkommunikation

Zwischen CPU und Device wird eine Schnittstelle eingefügt, welches Register in Adressen oder Daten übersetzt.



***Memory*-mapped I/O**

Pro Gerät wird ein eigenes Adressbereich reseviert, welcher nur für das Gerät verwendet werden kann.
+ Einfach(CPU weiss nicht, was Geräte, keine I/O-Logik)
– Adressraum des Speichers nicht für Speicher ausgenutzt

***Ported*-mapped IO**

Geräte haben eigenen IO Bus, zwei Adressräume, Speicher und Geräte
+ Speicher kann gesamten Adressraum verwenden
- Komplexität → CPU zusätzliche Logik, System zusätzlichen Bus

***Ported*-mapped I/O via Speicherbus¨**

Bus hat zusätzlicher Bitleitung(Selector Bit), welche mitteilt, ob Speicher oder Device. Verbindet Vorteile beider Verfahren.

Pollingsmechanismen(Wann/Wie werden Daten geholt)

Programmgesteuert: Ständige(oder Zyklische) Abfragung, ob Status Register gesetzt.
+keine Verzögerung -beansprucht CPU

Interruptgesteuert: Gerät sendet Unterbruchssignal wenn bereit

DMA(Direct Memory Access)

Eine weitere Möglichkeit, um Wartezeiten auf die Peripherie nutzungsbringend einzusetzten. Ziel ist es, dass der Datentransfer nicht über die CPU sondern direkt abläuft. Dabei gibt es einen weiteren Baustein, den DMA-Controller, welcher die Übertragung zwischen Daten aus dem Speicher und dem Device handelt:

Vorgehensweise(vereinfacht)

1. CPU programmiert DMA für Datentransfer
   a. CPU sieht DMA als eigentliches Gerät
   b. Quelle, Ziel, Menge, Betriebsart gesetzt
2. CPU gibt Speicherbus an DMA frei
   a. DMA gibt Angaben am Gerät, sodass dieses direkt den Transfer ausführen kann
3. Nach Beendigung sendet DMA an CPU einen Interrupt

Betriebsarten

Einzeltransfer: CPU gibt Speicherbus für einzelnen Transfer für DMA frei(wenige, periodische Daten Modem, Diskette)
Blockfransfer: CPU tritt Speicherbus an DMA ab, bis alle Daten übertragen sind(Fesplatte, Netzwerkkarte, etc)