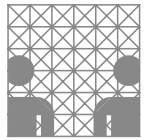


64-040 Modul InfB-RS: Rechnerstrukturen

https://tams.informatik.uni-hamburg.de/lectures/2016ws/vorlesung/rs

– Kapitel 15 –

Andreas Mäder

Universität Hamburg
Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik

Technische Aspekte Multimodaler Systeme

Wintersemester 2016/2017

Kapitel 15

Assembler-Programmierung

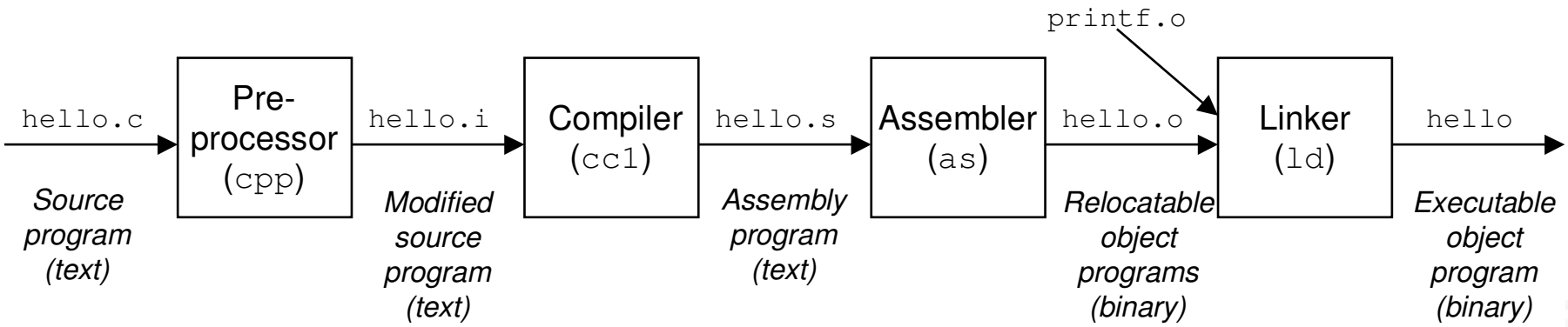
- Motivation
- Grundlagen der Assemblerebene
- x86 Assemblerprogrammierung
  - Elementare Befehle und Adressierungsarten
  - Operationen
  - Kontrollfluss
  - Sprungbefehle und Schleifen
  - Mehrfachverzweigung (Switch)
  - Funktionsaufrufe und Stack
- Speicherverwaltung
  - Elementare Datentypen
  - Arrays
  - Strukturen
  - Objektorientierte Konzepte
- Linker und Loader
- Dynamische Speicherverwaltung

Kapitel 15 (cont.)

Puffer-Überläufe

Literatur

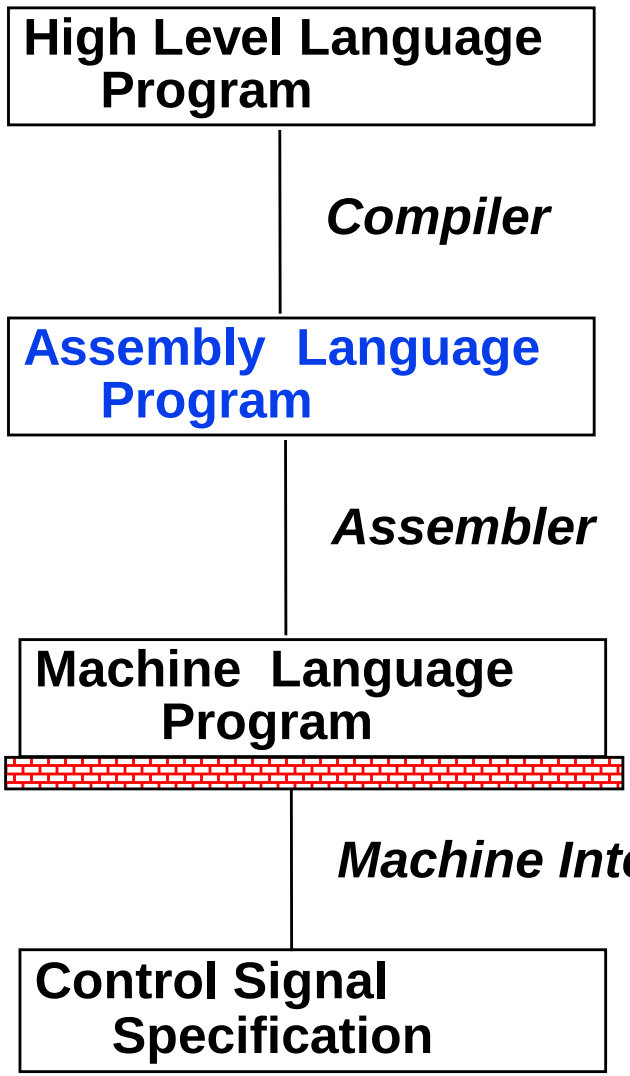
Wiederholung: Kompilierungssystem



[BO15]

- verschiedene Repräsentationen des Programms
  - Hochsprache
  - Assembler
  - Maschinensprache
- Ausführung der Maschinensprache
  - von-Neumann Zyklus: Befehl holen, decodieren, ausführen
  - reale oder virtuelle Maschine

Wiederholung: Kompilierungssystem (cont.)




```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw  $15,   0($2)
lw  $16,   4($2)
sw $16,   0($2)
sw $15,   4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

ALUOP[0:3] <= InstReg[9:11] & MASK

[PH16b]

Warum Assembler?

Programme werden nur noch selten in Assembler geschrieben

- Programmentwicklung in Hochsprachen weit produktiver
- Compiler/Tools oft besser als handcodierter Assembler

aber Grundwissen bleibt trotzdem unverzichtbar

- Verständnis des Ausführungsmodells auf der Maschinenebene
- Programmverhalten bei Fehlern / Debugging
  - das High-Level Sprachmodell ist dort nicht anwendbar
- Programmleistung verstärken
  - Ursachen für Programm-Ineffizienz verstehen
  - effiziente „maschinengerechte“ Datenstrukturen / Algorithmen
- Systemsoftware implementieren
  - Compilerbau: Maschinencode als Ziel
  - Betriebssysteme implementieren (Prozesszustände verwalten)
  - Gerätetreiber schreiben

Assembler: Lernziele

- Grundverständnis der Programmausführung
  - Umsetzung arithmetisch/logischer Operationen
  - Umsetzung der gängigen Kontrollstrukturen:
    (mehrfach) Verzweigungen, (bedingte) Sprünge, Schleifen
  - Datenstrukturen
  - ein- und mehrdimensionale Arrays

- Funktionsaufrufe — Stack
  - Funktionsparameter — by-value, by-reference
  - lokale Variablen
  - rekursive Funktionen

- Grundlagen dynamischer Speicherverwaltung — Heap
- Funktionsbibliotheken — Linker
- Umsetzung objektorientierter Konzepte im Rechner — vtable

Assembler: Speicherverwaltung

- Speicher aufgeteilt in mehrere Regionen
  - Programmcode
  - Funktionsbibliotheken, Linker und Loader
  - Stack mit Funktionsaufrufen und lokalen Variablen
  - statisch allozierte Daten und globale Variablen
  - dynamisch allozierte Daten
  - Umsetzung objektorientierter Konzepte
  - Interrupts, Exceptions, System-Calls

- Programmierfehler und Sicherheitslücken
  - aktuelle Rechner bieten keinen/kaum Speicherschutz
  - geschützte Systeme (“capabilities”) bisher am Markt gescheitert

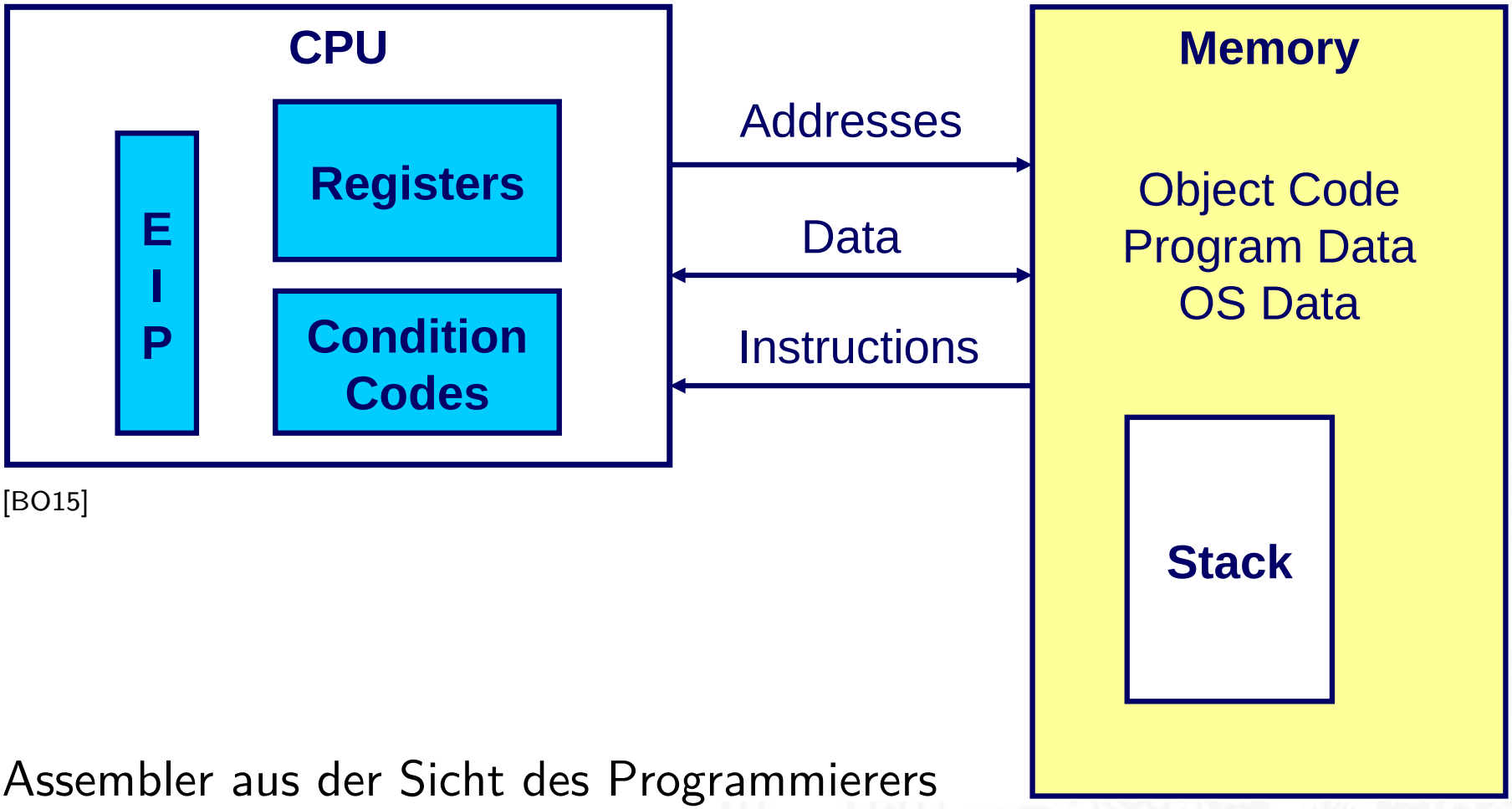
  - fehlerhafte dynamische Speicherverwaltung
  - Pufferüberläufe, Stack-allocated Daten
  - Ausnutzen durch bösartigen Code

Assembler in RS

- Beschränkung auf wesentliche Konzepte
  - GNU Assembler für x86 (32-bit)
  - nur ein Datentyp: 32-bit Integer (long)
  - nur kleiner Subset des gesamten Befehlssatzes

- diverse nicht behandelte Themen
  - Makros
  - Implementierung eines Assemblers (2-pass)
  - Tipps für effizientes Programmieren
  - Befehle für die Systemprogrammierung (supervisor mode)
  - x86 Gleitkommabefehle
  - ...

Assembler-Programmierung



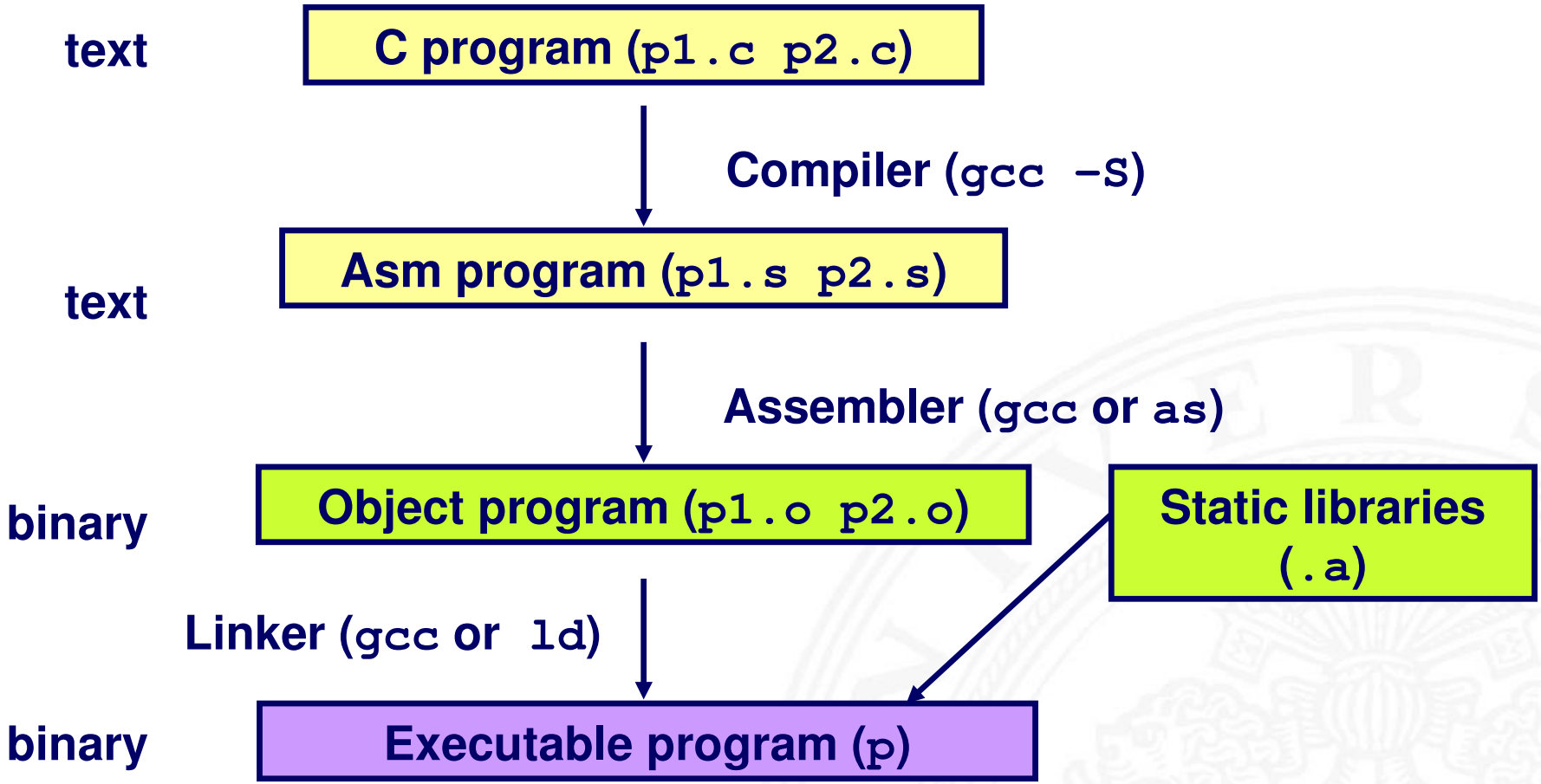
[BO15]

Assembler aus der Sicht des Programmierers

Beobachtbare Zustände (Assemblersicht)

- Programmzähler (*Instruction Pointer*) x86 `eip` Register
  - Adresse der nächsten Anweisung
- Registerbank `eax`...`ebp` Register
  - häufig benutzte Programmdaten
- Zustandscodes EFLAGS Register
  - gespeicherte Statusinformationen über die letzte arithmetische Operation
  - für bedingte Sprünge benötigt (*Conditional Branch*)
- Speicher
  - byteweise adressierbares Array
  - Code, Nutzerdaten, (einige) OS Daten
  - beinhaltet Kellerspeicher zur Unterstützung von Abläufen

Umwandlung von C in Objektcode



[BO15]

Kompilieren zu Assemblercode: Funktion sum()

code.c

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

[BO15]

- Befehl `gcc -0 -S code.c`
- Erzeugt `code.s`

code.s

```
_sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

Assembler: Charakteristika

- hardwarenahe Programmierung: Zugriff auf kompletten Befehlssatz und alle Register einer Maschine

- je ein Befehl pro Zeile
  - Mnemonics für die einzelnen Maschinenbefehle
  - Konstanten als Dezimalwerte oder Hex-Werte
  - eingängige Namen für alle Register
  - Addressen für alle verfügbaren Adressierungsarten
  - Konvention bei gcc/gas x86: Ziel einer Operation steht rechts

- symbolische Label für Sprungadressen
  - Verwendung in Sprungbefehlen
  - globale Label definieren Einsprungpunkte für den Linker/Loader

Assembler: Datentypen

- nur die von der Maschine unterstützten „primitiven“ Daten
- keine Aggregattypen wie Arrays, Strukturen, oder Objekte
  - nur fortlaufend adressierbare Bytes im Speicher

- Ganzzahl-Daten, z.B. 1, 2, 4, oder 8 Bytes — 8...64 bits
  - Datenwerte für Variablen — int/long/long long
  - positiv oder vorzeichenbehaftet — signed/unsigned
  - Textzeichen (ASCII, Unicode) — char
- Gleitkomma-Daten mit 4 oder 8 Bytes — float/double
- Adressen bzw. „Pointer“ — untypisierte Adressenverweise

Assembler: Befehle/Operationen

- arithmetische/logische Funktionen auf Registern und Speicher
  - Addition/Subtraktion, Multiplikation, usw.
  - bitweise logische und Schiebe-Operationen
- Datentransfer zwischen Speicher und Registern
  - Daten aus Speicher in Register laden
  - Registerdaten im Speicher ablegen
  - ggf. auch Zugriff auf Spezial-/OS-register
- Kontrolltransfer
  - unbedingte / bedingte Sprünge
  - Unterprogrammaufrufe: Sprünge zu/von Prozeduren
  - Interrupts, Exceptions, System-Calls

- Makros: Folge von Assemblerbefehlen

Objektcode: Funktion sum()

- 13 Bytes Programmcode
- x86-Instruktionen mit 1-, 2- oder 3 Bytes Erklärung s.u.

- Startadresse: `0x401040`
- vom Compiler/Assembler gewählt

```
0x401040 <sum>:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

Assembler und Linker

Assembler

- übersetzt `.s` zu `.o`
- binäre Codierung jeder Anweisung
- (fast) vollständiges Bild des ausführbaren Codes
- Verknüpfungen zwischen Code in verschiedenen Dateien fehlen

Linker / Binder

- löst Referenzen zwischen Dateien auf
- kombiniert mit statischen Laufzeit-Bibliotheken
  - z.B. Code für `malloc, printf`
- manche Bibliotheken sind *dynamisch* verknüpft
  - Verknüpfung wird zur Laufzeit erstellt

Beispiel: Maschinenbefehl für Addition

- C-Code
  - addiert zwei Ganzzahlen mit Vorzeichen

```
int t = x+y;
```

- Assembler
  - Addiere zwei 4-Byte Integer
    - `long` Wörter (für gcc)
    - keine signed/unsigned Unterscheidung
  - Operanden

    `x:` Register `%eax`
    `y:` Speicher `M[%ebp+8]`
    `t:` Register `%eax`
    Ergebnis in `%eax`

```
addl 8(%ebp),%eax
```

Similar to expression
`x += y`

- Objektcode (x86-Befehlssatz)
  - 3-Byte Befehl
  - Speicheradresse `0x401046`

```
0x401046:    03 45 08
```

Objektcode Disassembler: objdump

```
00401040 <_sum>:
   0:      55                   push   %ebp
   1:      89 e5                mov    %esp,%ebp
   3:      8b 45 0c             mov    0xc(%ebp),%eax
   6:      03 45 08             add    0x8(%ebp),%eax
   9:      89 ec                mov    %ebp,%esp
   b:      5d                   pop    %ebp
   c:      c3                   ret
   d:      8d 76 00             lea    0x0(%esi),%esi
```

[BO15]

- objdump -d ...
  - Werkzeug zur Untersuchung des Objektcodes
  - rekonstruiert aus Binärcode den Assemblercode
  - kann auf vollständigem, ausführbaren Programm (a.out) oder einer .o Datei ausgeführt werden

Alternativer Disassembler: gdb

Object

```
0x401040:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

Disassembled

```
0x401040 <sum>:      push   %ebp
0x401041 <sum+1>:    mov    %esp,%ebp
0x401043 <sum+3>:    mov    0xc(%ebp),%eax
0x401046 <sum+6>:    add    0x8(%ebp),%eax
0x401049 <sum+9>:    mov    %ebp,%esp
0x40104b <sum+11>:   pop    %ebp
0x40104c <sum+12>:   ret
0x40104d <sum+13>:   lea    0x0(%esi),%esi
```

gdb Debugger

```
gdb p
disassemble sum
```

- Disassemble procedure

```
x/13b sum
```

- Examine the 13 bytes starting at `sum`

[BO15]

Was kann „disassembliert“ werden?


```
% objdump -d WINWORD.EXE

WINWORD.EXE:     file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:  55                 push   %ebp
30001001:  8b ec              mov    %esp,%ebp
30001003:  6a ff              push   $0xffffffff
30001005:  68 90 10 00 30     push   $0x30001090
3000100a:  68 91 dc 4c 30     push   $0x304cdc91
```

[BO15]

- alles, was als ausführbarer Code interpretiert werden kann
- Disassembler untersucht Bytes und rekonstruiert Assemblerquelle (soweit wie möglich)

x86 Assemblerprogrammierung

- Adressierungsarten
- arithmetische Operationen
- Statusregister
- Umsetzung von Programmstrukturen

Einschränkungen

- Beispiele nutzen nur die 32-bit (long) Datentypen
  - x86 wird wie 8-Register 32-bit Maschine benutzt (=RISC)
  - CISC Komplexität und Tricks bewusst vermieden
- Beispiele nutzen gcc/gas Syntax (vs. Microsoft, Intel)

Grafiken und Beispiele dieses Abschnitts sind aus R.E. Bryant, D.R. O'Hallaron: *Computer systems – A programmers perspective* [BO15], bzw. dem zugehörigen Foliensatz

Datentransfer „move“

- Format: `movl` ⟨*src*⟩`,` ⟨*dst*⟩
- transferiert ein 4-Byte „long“ Wort
- sehr häufige Instruktion

- Typ der Operanden
  - Immediate: Konstante, ganzzahlig
    - wie C-Konstante, aber mit dem Präfix `$`
    - z.B.: `$0x400`, `$-533`
    - codiert mit 1, 2 oder 4 Bytes
  - Register: 8 Ganzzahl-Register
    - `%esp` und `%ebp` für spezielle Aufgaben reserviert
    - z.T. Spezialregister für andere Anweisungen
  - Speicher: 4 konsekutive Speicherbytes
    - zahlreiche Adressmodi

movl Operanden-Kombinationen

	Source	Destination		C Analogon
movl	Imm	Reg	`movl $0x4,%eax`	`temp = 0x4;`
		Mem	`movl $-147,(%eax)`	`*p = -147;`
	Reg	Reg	`movl %eax,%edx`	`temp2 = temp1;`
		Mem	`movl %eax,(%edx)`	`*p = temp;`
	Mem	Reg	`movl (%eax),%edx`	`temp = *p;`

movl: Operanden/Adressierungsarten

- Immediate: `$x` → `x`
  - positiver (oder negativer) Integerwert
- Register: `%R` → `Reg[R]`
  - Inhalt eines der 8 Universalregister `eax`...`ebp`
- Normal: `(R)` → `Mem[Reg[R]]`
  - Register `R` spezifiziert die Speicheradresse
  - Beispiel: `movl (%ecx), %eax`
- Displacement: `D(R)` → `Mem[Reg[R]+D]`
  - Register `R`
  - Konstantes „Displacement“ D spezifiziert den „offset“
  - Beispiel: `movl 8(%ebp), %edx`

Beispiel: Funktion swap()


```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

Nutzung der Register: ecx: yp
edx: xp
eax: t1
ebx: t0

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx

    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Set Up

Body

Finish

Indizierte Adressierung

- allgemeine Form
  - `Imm(Rb,Ri,S)` → `Mem[Reg[Rb]+S*Reg[Ri]+Imm]`
    - $\langle Imm \rangle$ Offset
    - $\langle Rb \rangle$ Basisregister: eines der 8 Integer-Register
    - $\langle Ri \rangle$ Indexregister: jedes außer %esp
      %ebp grundsätzlich möglich, jedoch unwahrscheinlich
    - $\langle S \rangle$ Skalierungsfaktor 1, 2, 4 oder 8

- gebräuchlichste Fälle
  - `(Rb)` → `Mem[Reg[Rb]]`
  - `Imm(Rb)` → `Mem[Reg[Rb] + Imm]`
  - `(Rb,Ri)` → `Mem[Reg[Rb] + Reg[Ri]]`
  - `Imm(Rb,Ri)` → `Mem[Reg[Rb] + Reg[Ri] + Imm]`
  - `(Rb,Ri,S)` → `Mem[Reg[Rb] + S*Reg[Ri]]`

Beispiel: Adressberechnung

%edx	0xf000
%ecx	0x100

Expression	Computation	Address
0x8(%edx)	0xf000 + 0x8	0xf008
(%edx,%ecx)	0xf000 + 0x100	0xf100
(%edx,%ecx,4)	0xf000 + 4*0x100	0xf400
0x80(,%edx,2)	2*0xf000 + 0x80	0x1e080

Arithmetische Operationen

- binäre Operatoren

Format		Computation	
`addl`	*Src,Dest*	*Dest* = *Dest* + *Src*	
`subl`	*Src,Dest*	*Dest* = *Dest* – *Src*	
`imull`	*Src,Dest*	*Dest* = *Dest* * *Src*	
`sall`	*Src,Dest*	*Dest* = *Dest* << *Src*	also called `shll`
`sarl`	*Src,Dest*	*Dest* = *Dest* >> *Src*	Arithmetic
`shrl`	*Src,Dest*	*Dest* = *Dest* >> *Src*	Logical
`xorl`	*Src,Dest*	*Dest* = *Dest* ^ *Src*	
`andl`	*Src,Dest*	*Dest* = *Dest* & *Src*	
`orl`	*Src,Dest*	*Dest* = *Dest* | *Src*	

Arithmetische Operationen (cont.)

- unäre Operatoren

Format	Computation
`incl` ***Dest***	***Dest*** `=` ***Dest*** `+ 1`
`decl` ***Dest***	***Dest*** `=` ***Dest*** `- 1`
`negl` ***Dest***	***Dest*** `= -` ***Dest***
`notl` ***Dest***	***Dest*** `= ~` ***Dest***

- `leal`-Befehl: *load effective address*
  - Adressberechnung für (späteren) Ladebefehl
  - Speichert die Adresse in Register:
    `Imm(Rb,Ri,S)` $\rightarrow$ `Reg[Rb]+S*Reg[Ri]+Imm`
  - wird oft von Compilern für arithmetische Berechnung genutzt
    s. Beispiele

Beispiel: arithmetische Operationen


```
int arith
   (int x, int y, int z)
{
   int t1 = x+y;
   int t2 = z+t1;
   int t3 = x+4;
   int t4 = y * 48;
   int t5 = t3 + t4;
   int rval = t2 * t5;
   return rval;
}
```

```
arith:
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%eax
    movl 12(%ebp),%edx
    leal (%edx,%eax),%ecx
    leal (%edx,%edx,2),%edx
    sall $4,%edx
    addl 16(%ebp),%ecx
    leal 4(%edx,%eax),%eax
    imull %ecx,%eax

    movl %ebp,%esp
    popl %ebp
    ret
```

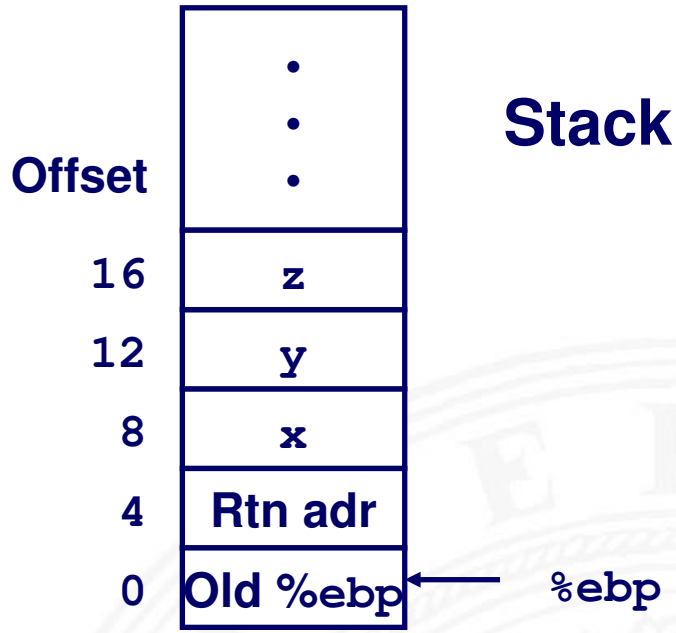
Set Up

Body

Finish

Beispiel: arithmetische Operationen (cont.)


```
int arith
   (int x, int y, int z)
{
   int t1 = x+y;
   int t2 = z+t1;
   int t3 = x+4;
   int t4 = y * 48;
   int t5 = t3 + t4;
   int rval = t2 * t5;
   return rval;
}
```




```
movl 8(%ebp),%eax          # eax = x
movl 12(%ebp),%edx         # edx = y
leal (%edx,%eax),%ecx      # ecx = x+y   (t1)
leal (%edx,%edx,2),%edx    # edx = 3*y
sall $4,%edx               # edx = 48*y (t4)
addl 16(%ebp),%ecx         # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax     # eax = 4+t4+x (t5)
imull %ecx,%eax            # eax = t5*t2 (rval)
```

Beispiel: logische Operationen


```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

$2^{13} = 8192, 2^{13} - 7 = 8185$

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    movl %ebp,%esp
    popl %ebp
    ret
```

Set Up

Body

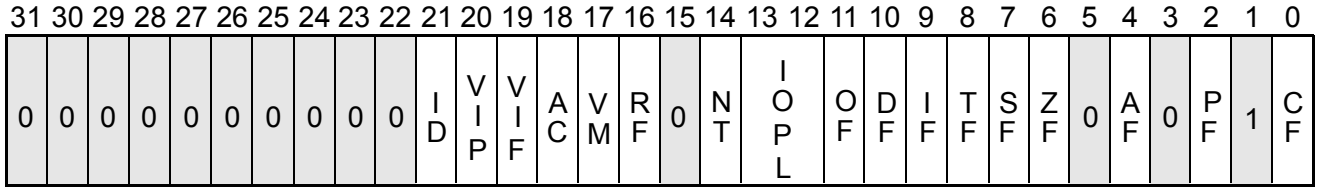
Finish

```
movl 8(%ebp),%eax      eax = x
xorl 12(%ebp),%eax     eax = x^y     (t1)
sarl $17,%eax          eax = t1>>17  (t2)
andl $8185,%eax        eax = t2 & 8185
```

Kontrollfluss / Programmstrukturen

- Zustandscodes
  - Setzen
  - Testen
- Ablaufsteuerung
  - Verzweigungen: „If-then-else“
  - Schleifen: „Loop“-Varianten
  - Mehrfachverzweigungen: „Switch“

x86: EFLAGS Register



X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check / Access Control (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)

S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag

Reserved bit positions. DO NOT USE.
Always set to values previously read.

[IA64]

Zustandscodes

- vier relevante „Flags“ im Statusregister EFLAGS
  - `CF` Carry Flag
  - `SF` Sign Flag
  - `ZF` Zero Flag
  - `OF` Overflow Flag

1. implizite Aktualisierung durch arithmetische Operationen
   - Beispiel: `addl` $\langle src \rangle$, $\langle dst \rangle$ in C: `t=a+b`
   - `CF` höchstwertiges Bit generiert Übertrag: Unsigned-Überlauf
   - `ZF` wenn $t = 0$
   - `SF` wenn $t < 0$
   - `OF` wenn das Zweierkomplement überläuft
     $(a > 0 \;\&\&\; b > 0 \;\&\&\; t < 0) \;||\; (a < 0 \;\&\&\; b < 0 \;\&\&\; t \geq 0)$

Zustandscodes (cont.)

2. explizites Setzen durch Vergleichsoperation
   - Beispiel: `cmpl` $\langle src2 \rangle$, $\langle src1 \rangle$
     wie Berechnung von $\langle src1 \rangle$ - $\langle src2 \rangle$ (`subl` $\langle src2 \rangle$, $\langle src1 \rangle$)
     jedoch ohne Abspeichern des Resultats

   - `CF` höchstwertiges Bit generiert Übertrag
   - `ZF` setzen wenn $src1 = src2$
   - `SF` setzen wenn $(src1 - src2) < 0$
   - `OF` setzen wenn das Zweierkomplement überläuft
     $(a > 0 \;\&\&\; b < 0 \;\&\&\; (a - b) < 0) \;||$
     $(a < 0 \;\&\&\; b > 0 \;\&\&\; (a - b) \geq 0)$

Zustandscodes (cont.)

3. explizites Setzen durch Testanweisung
   - Beispiel: `testl` $\langle src2 \rangle$, $\langle src1 \rangle$
     wie Berechnung von $\langle src1 \rangle \& \langle src2 \rangle$ (`andl` $\langle src2 \rangle$, $\langle src1 \rangle$)
     jedoch ohne Abspeichern des Resultats

   $\Rightarrow$ hilfreich, wenn einer der Operanden eine Bitmaske ist

   - ZF setzen wenn $src1 \& src2 = 0$
   - SF setzen wenn $src1 \& src2 < 0$

Zustandscodes lesen: set..-Befehle

- Befehle setzen ein einzelnes Byte (LSB) in Universalregister

SetX	Condition	Description
sete	ZF	Equal / Zero
setne	~ZF	Not Equal / Not Zero
sets	SF	Negative
setns	~SF	Nonnegative
setg	~(SF^OF)&~ZF	Greater (Signed)
setge	~(SF^OF)	Greater or Equal (Signed)
setl	(SF^OF)	Less (Signed)
setle	(SF^OF)|ZF	Less or Equal (Signed)
seta	~CF&~ZF	Above (unsigned)
setb	CF	Below (unsigned)

Beispiel: Zustandscodes lesen

- ein-Byte Zieloperand (Register, Speicher)
- meist kombiniert mit `movzbl`
  move with zero-extend byte to long
  also Löschen der Bits 31...8


```
int gt (int x, int y)
{
  return x > y;
}
```

```
movl 12(%ebp),%eax   # eax = y
cmpl %eax,8(%ebp)    # Compare x : y
setg %al             # al = x > y
movzbl %al,%eax      # Zero rest of %eax
```

Sprünge („Jump“): j..-Befehle

- unbedingter- / bedingter Sprung (abhängig von Zustandscode)

jX	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

Assembler: Label

- Assemblercode enthält je einen Maschinenbefehl pro Zeile
- normale Programmausführung ist sequentiell
- Befehle beginnen an eindeutig bestimmten Speicheradressen

- Label: symbolische Namen für bestimmte Adressen
  - am Beginn einer Zeile, oder vor einem Befehl
  - vom Programmierer / Compiler vergeben
  - als symbolische Adressen für Sprünge verwendet

  - `_max`: global, Beginn der Funktion `max()`
  - `L9:` lokal, nur vom Assembler verwendete interne Adresse

  - Label müssen in einem Programm eindeutig sein

Beispiel: bedingter Sprung


```
int max(int x, int y)
{
  if (x > y)
    return x;
  else
    return y;
}
```

```
_max:
        pushl %ebp
        movl %esp,%ebp

        movl 8(%ebp),%edx
        movl 12(%ebp),%eax
        cmpl %eax,%edx
        jle L9
        movl %edx,%eax
L9:

        movl %ebp,%esp
        popl %ebp
        ret
```

Set Up

Body

Finish

Beispiel: bedingter Sprung (cont.)


```
int goto_max(int x, int y)
{
  int rval = y;
  int ok = (x <= y);
  if (ok)
    goto done;
  rval = x;
done:
  return rval;
}
```

- C-Code mit goto
- entspricht mehr dem Assemblerprogramm
- schlechter Programmierstil

```
    movl 8(%ebp),%edx    # edx = x
    movl 12(%ebp),%eax   # eax = y
    cmpl %eax,%edx       # x : y
    jle L9               # if <= goto L9
    movl %edx,%eax       # eax = x   } Skipped when x ≤ y
L9:                      # Done:
```

Beispiel: „Do-While“ Schleife

- C Code

```
int fact_do
  (int x)
{
  int result = 1;
  do {
    result *= x;
    x = x-1;
  } while (x > 1);
  return result;
}
```

goto Version

```
int fact_goto(int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

- Rückwärtssprung setzt Schleife fort
- wird nur ausgeführt, wenn „while“ Bedingung gilt

Beispiel: „Do-While“ Schleife (cont.)


```
int fact_goto
  (int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

Register

%edx x

%eax result

```
_fact_goto:
  pushl %ebp          # Setup
  movl %esp,%ebp      # Setup
  movl $1,%eax        # eax = 1
  movl 8(%ebp),%edx   # edx = x

L11:
  imull %edx,%eax     # result *= x
  decl %edx           # x--
  cmpl $1,%edx        # Compare x : 1
  jg L11              # if > goto loop

  movl %ebp,%esp      # Finish
  popl %ebp           # Finish
  ret                 # Finish
```

„Do-While“ Übersetzung

C Code

```
do
  Body
  while (Test);
```

Goto Version

```
loop:
  Body
  if (Test)
    goto loop
```

- beliebige Folge von C Anweisungen als Schleifenkörper
- Abbruchbedingung ist zurückgelieferter Integer Wert
  - $= 0$ entspricht Falsch: Schleife verlassen
  - $\neq 0$ –"– Wahr: nächste Iteration

„While“ Übersetzung

C Code

```
while (Test)
  Body
```

Do-While Version

```
  if (!Test)
    goto done;
  do
    Body
    while(Test);
done:
```

Goto Version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

„For“ Übersetzung

For Version

```
for (Init; Test; Update)
    Body
```

While Version

```
Init;
while (Test) {
    Body
    Update ;
}
```

Do-While Version

```
  Init;
  if (!Test)
    goto done;
  do {
    Body
    Update ;
  } while (Test)
done:
```

Goto Version

```
  Init;
  if (!Test)
    goto done;
loop:
  Body
  Update ;
  if (Test)
    goto loop;
done:
```

Mehrfachverzweigungen „Switch“

- Implementierungsoptionen
  1. Folge von „If-Then-Else“
     - \+ gut bei wenigen Alternativen
     - – langsam bei vielen Fällen
  2. Sprungtabelle „Jump Table“
     - Vermeidet einzelne Abfragen
     - möglich falls Alternativen kleine ganzzahlige Konstanten sind
  - Compiler (gcc) wählt eine der beiden Varianten entsprechend der Fallstruktur

Anmerkung: im Beispielcode fehlt „Default“

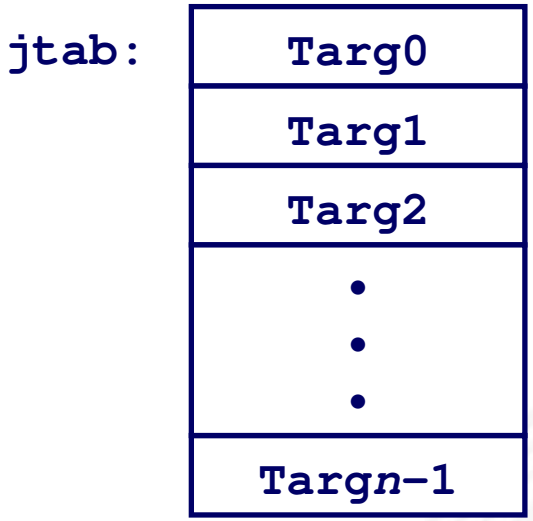
```
typedef enum
 {ADD, MULT, MINUS, DIV, MOD, BAD}
    op_type;

char unparse_symbol(op_type op)
{
  switch (op) {
  case ADD :
    return '+';
  case MULT:
    return '*';
  case MINUS:
    return '-';
  case DIV:
    return '/';
  case MOD:
    return '%';
  case BAD:
    return '?';
  }
}
```

Sprungtabelle

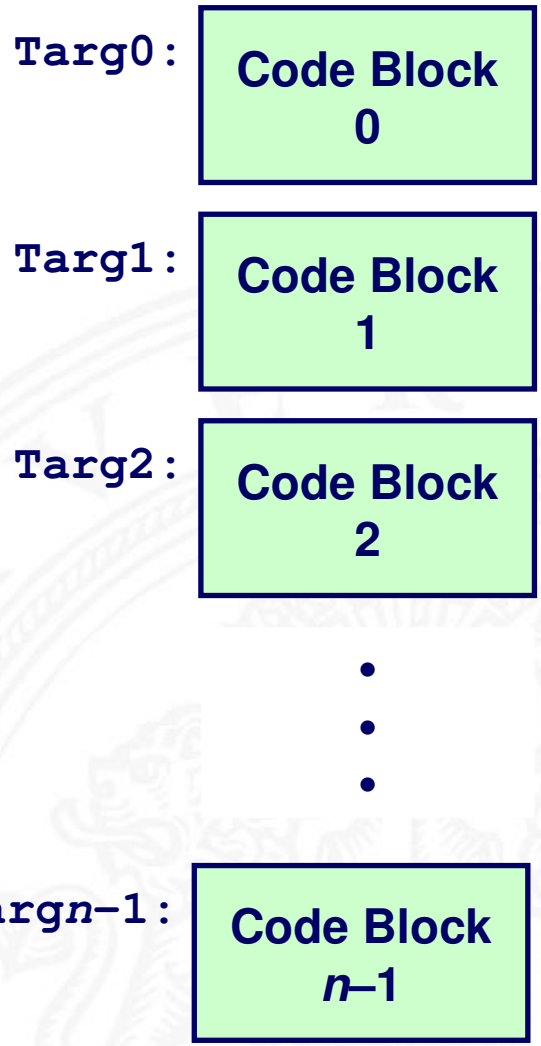
Switch Form

```
switch(op) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

Jump Table



Jump Targets



Approx. Translation

```
target = JTab[op];
goto *target;
```

- Vorteil: k-fach Verzweigung in $\mathcal{O}(1)$ Operationen

Beispiel: „Switch“

Branching Possibilities

```
typedef enum
  {ADD, MULT, MINUS, DIV, MOD, BAD}
    op_type;

char unparse_symbol(op_type op)
{
  switch (op) {
    • • •
  }
}
```

Enumerated Values

```
ADD   0
MULT  1
MINUS 2
DIV   3
MOD   4
BAD   5
```

Setup:

```
unparse_symbol:
   pushl %ebp              # Setup
   movl %esp,%ebp          # Setup
   movl 8(%ebp),%eax       # eax = op
   cmpl $5,%eax            # Compare op : 5
   ja .L49                 # If > goto done
   jmp *.L57(,%eax,4) # goto Table[op]
```

Beispiel: „Switch“ (cont.)

- Compiler erzeugt Code für jeden `case` Zweig
  - je ein Label am Start der Zweige, `.L51`...`.L56`
  - werden dann vom Assembler/Linker in Adressen umgesetzt
- Tabellenstruktur
  - jedes Ziel benötigt 4 Bytes
  - Basisadresse bei `.L57`
- Sprünge
  - `jmp *.L57(,%eax, 4)`
    - Sprungtabelle ist mit Label `.L57` gekennzeichnet
    - Register `%eax` speichert *op*
    - Skalierungsfaktor 4 für Tabellenoffset
    - Sprungziel: effektive Adresse `.L57` $+ \; op \times 4$
  - `jmp .L49` markiert das Ende der Switch-Anweisung

Beispiel: „Switch“ (cont.)

Table Contents

```
.section .rodata
    .align 4
.L57:
  .long .L51 #Op = 0
  .long .L52 #Op = 1
  .long .L53 #Op = 2
  .long .L54 #Op = 3
  .long .L55 #Op = 4
  .long .L56 #Op = 5
```

Enumerated Values

```
ADD     0
MULT    1
MINUS   2
DIV     3
MOD     4
BAD     5
```

Targets & Completion

```
.L51:
    movl $43,%eax # '+'
    jmp .L49
.L52:
    movl $42,%eax # '*'
    jmp .L49
.L53:
    movl $45,%eax # '-'
    jmp .L49
.L54:
    movl $47,%eax # '/'
    jmp .L49
.L55:
    movl $37,%eax # '%'
    jmp .L49
.L56:
    movl $63,%eax # '?'
    # Fall Through to .L49
```

Sprungtabelle aus Binärcode Extrahieren


```
Contents of section .rodata:
 8048bc0 30870408 37870408 40870408 47870408  0...7...@...G...
 8048bd0 50870408 57870408 46616374 28256429  P...W...Fact(%d)
 8048be0 203d2025 6c640a00 43686172 203d2025   = %ld..Char = %
 …
```

- im read-only Datensegment gespeichert (`.rodata`)
  - dort liegen konstante Werte des Codes
- kann mit `objdump` untersucht werden
  `objdump code-examples -s --section=.rodata`
  - zeigt alles im angegebenen Segment
  - schwer zu lesen (!)
  - Einträge der Sprungtabelle in umgekehrter Byte-Anordnung
    z.B: `30870408` ist eigentlich `0x08048730`

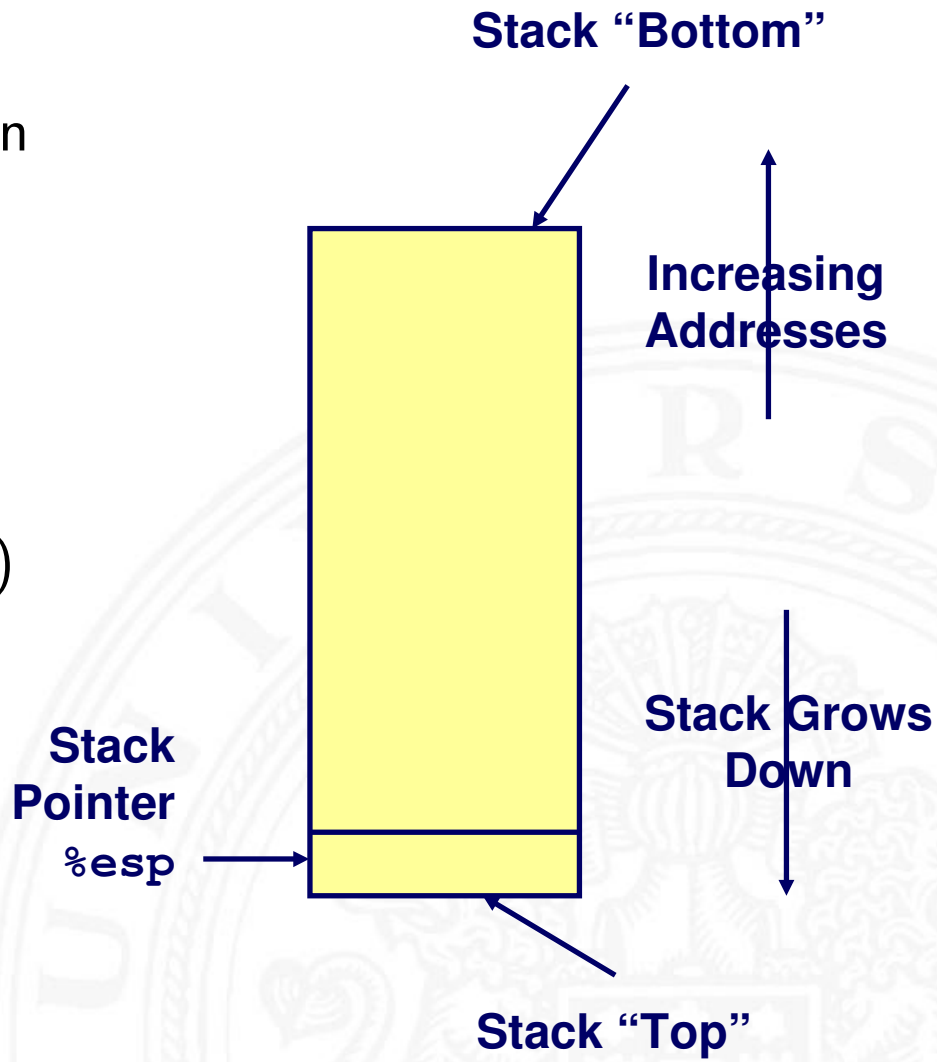
Zusammenfassung – Assembler

- Primitive Operationen und Adressierung
- C Kontrollstrukturen
  - „if-then-else“
  - „do-while“, „while“, „for“
  - „switch“
- Assembler Kontrollstrukturen
  - „Jump“
  - „Conditional Jump“
- Compiler
  - erzeugt Assembler Code für komplexere C Kontrollstrukturen
  - alle Schleifen in „do-while“ / „goto“ Form konvertieren
  - Sprungtabellen für Mehrfachverzweigungen „case“

Stack (Kellerspeicher)

- Speicherregion
- Startadresse vom OS vorgegeben
- Zugriff mit Stackoperationen
- wächst in Richtung niedrigerer Adressen

- Register `%esp` („Stack-Pointer“)
  - aktuelle Stack-Adresse
  - oberstes Element



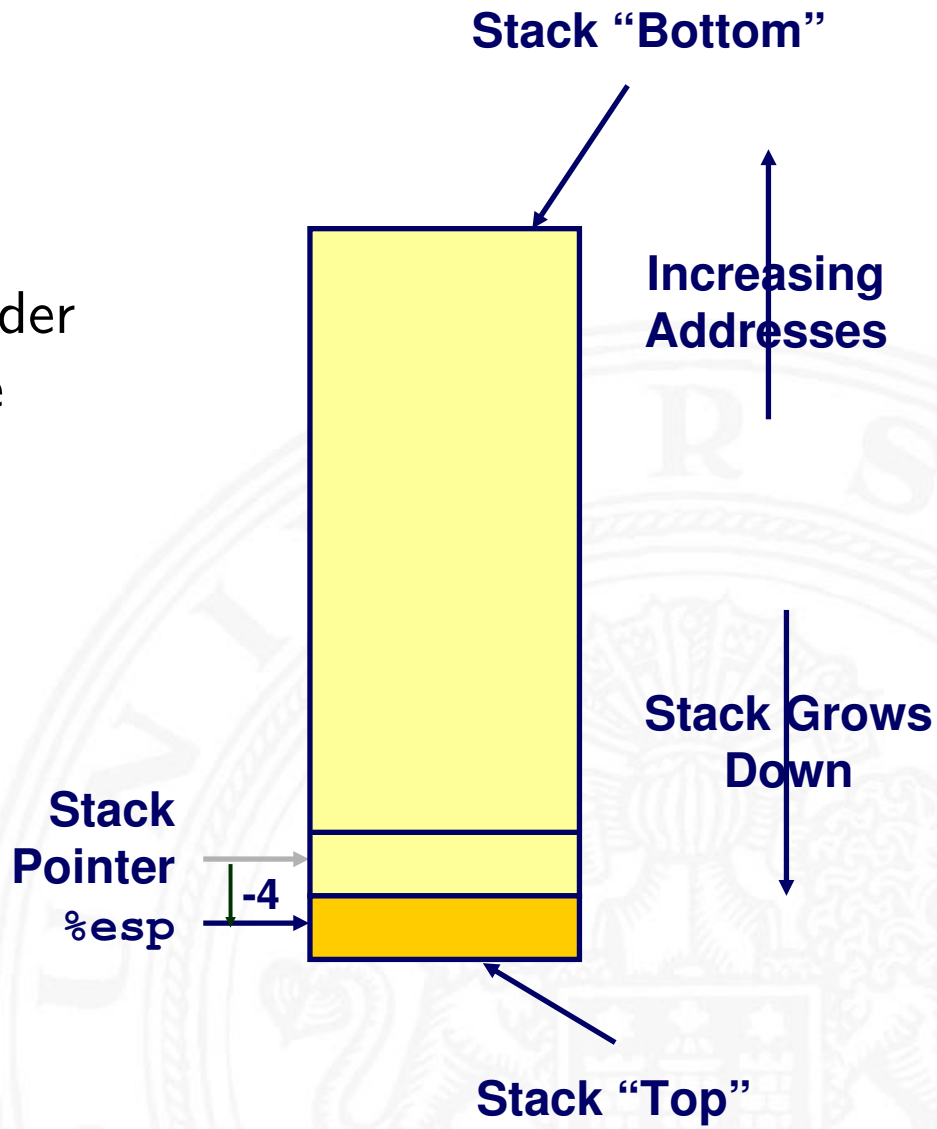
Stack (Kellerspeicher) (cont.)

- Implementierung von Funktionen/Prozeduren
  - Speicherplatz für Aufruf-Parameter
  - Speicherplatz für lokale Variablen
  - Rückgabe der Funktionswerte
  - auch für rekursive Funktionen (!)
- mehrere Varianten/Konventionen
  - Parameterübergabe in Registern
  - „Caller-Save“
  - „Callee-Save“
  - Kombinationen davon
  - Aufruf einer Funktion muss deren Konvention berücksichtigen

Stack: Push

`pushl` ⟨*src*⟩

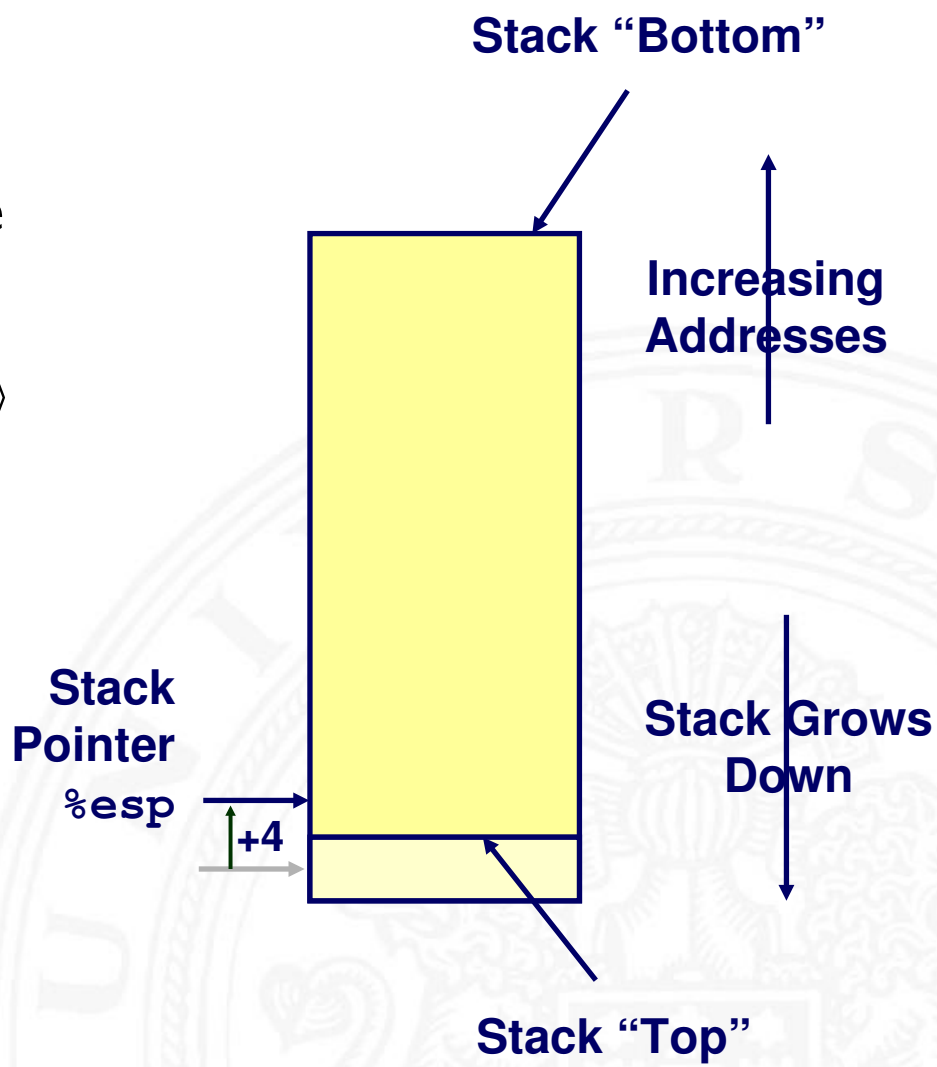
- holt Operanden aus ⟨*src*⟩
- dekrementiert `%esp` um 4
- speichert den Operanden unter der von `%esp` vorgegebenen Adresse



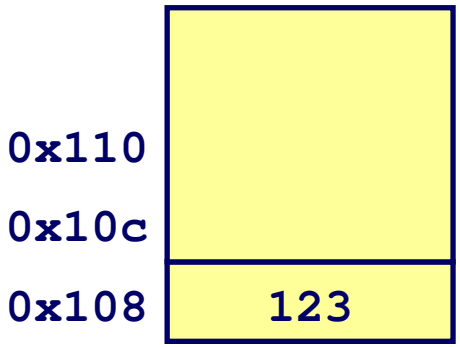
Stack: Pop

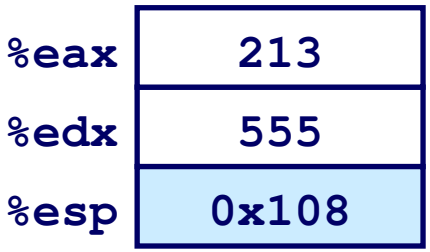
`popl` ⟨*dst*⟩

- liest den Operanden unter der von `%esp` vorgegebenen Adresse
- inkrementiert `%esp` um 4
- schreibt gelesenen Wert in ⟨*dst*⟩

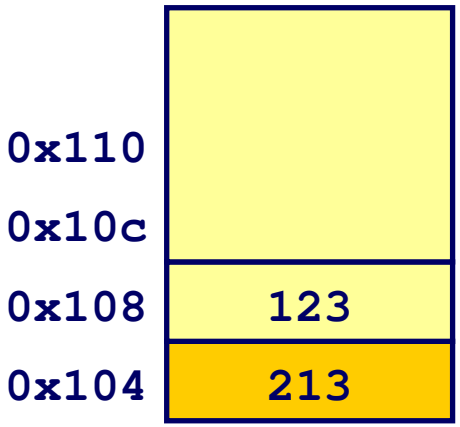


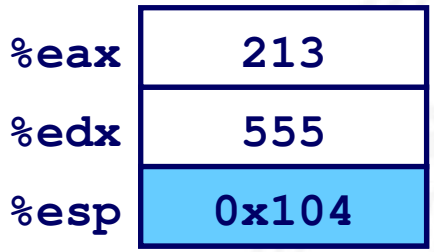
Beispiele: Stack-Operationen



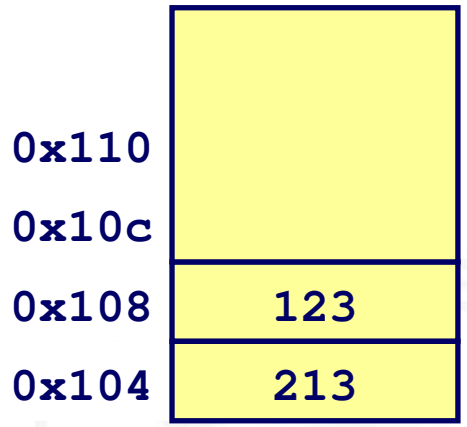


pushl %eax





popl %edx



Funktions-/Prozeduraufruf

- x86 ist CISC: spezielle Maschinenbefehle für Funktionsaufruf
  - `call` zum Aufruf einer Funktion
  - `ret` zum Rücksprung aus der Funktion
  - beide Funktionen ähnlich `jmp`: `eip` wird modifiziert
  - Stack wird zur Parameterübergabe verwendet
- zwei Register mit Spezialaufgaben
  - `%esp` „stack-pointer“: Speicheradresse des top-of-stack
  - `%ebp` „base-pointer“: Speicheradresse der aktuellen Funktion

Funktions-/Prozeduraufruf (cont.)

- Prozeduraufruf: `call` ⟨*label*⟩
  - Rücksprungadresse auf Stack („Push“)
  - Sprung zu ⟨*label*⟩
- Wert der Rücksprungadresse
  - Adresse der auf den `call` folgenden Anweisung
  - Beispiel:

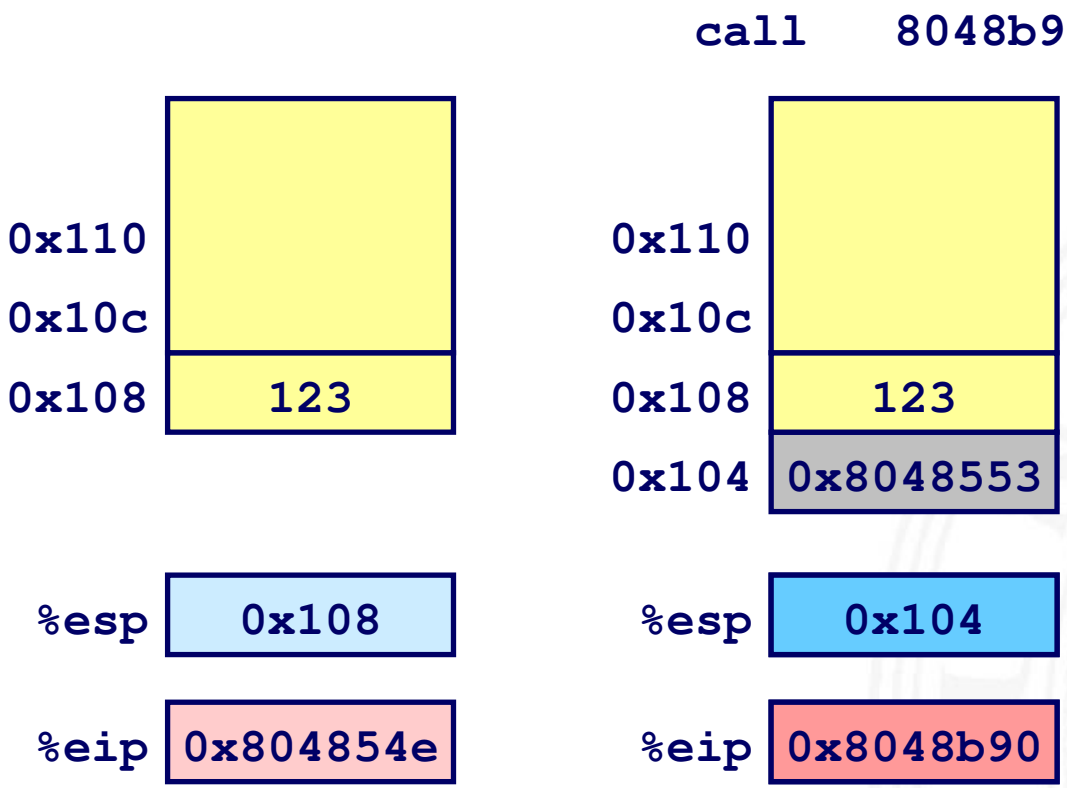
```
804854e:   e8 3d 06 00 00   ;call 8048b90
8048553:   50               ;pushl %eax
⟨main⟩     ...              ;...
8048b90:                    ;Prozedureinsprung
⟨proc⟩     ...              ;...
...        ret              ;Rücksprung
```

  - Rücksprungadresse `0x8048553`
- Rücksprung `ret`
  - Rücksprungadresse vom Stack („Pop“)
  - Sprung zu dieser Adresse

Beispiel: Prozeduraufruf

- Prozeduraufruf `call`

```
804854e:  e8 3d 06 00 00      call   8048b90 <main>
8048553:  50                  pushl  %eax
```

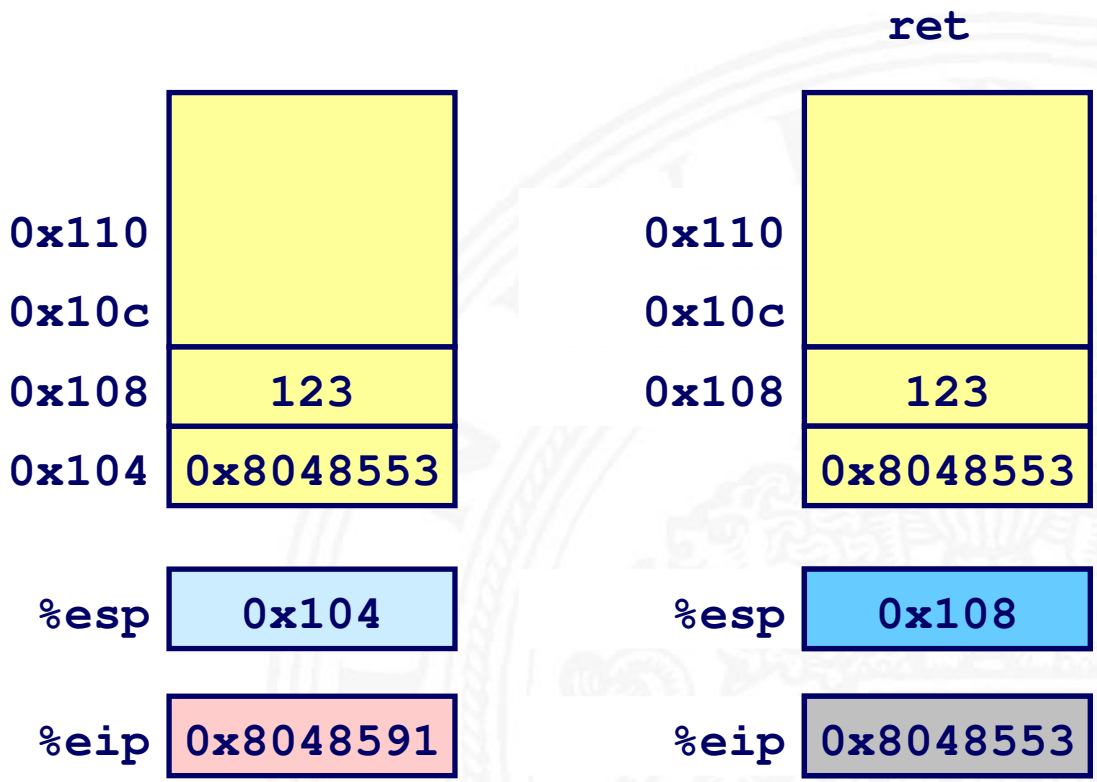


`%eip` is program counter

Beispiel: Prozeduraufruf (cont.)

- Prozedurrücksprung `ret`

```
8048591:   c3                    ret
```



`%eip` is program counter

Stack-basierende Programmierung

- für alle Programmiersprachen, die Rekursion unterstützen
  - C, Pascal, Java, Lisp, usw.
    - Code muss „reentrant“ sein
    - erlaubt mehrfache, simultane Instanziierungen einer Prozedur
  - benötigt Platz, um den Zustand jeder Instanziierung zu speichern
    - Argumente
    - lokale Variable(n)
    - Rücksprungadresse
- Stack-„Prinzip“
  - dynamischer Zustandsspeicher für Aufrufe
  - zeitlich limitiert: vom Aufruf (`call`) bis zum Rücksprung (`ret`)
  - aufgerufenes Unterprogramm („Callee“) wird vor dem aufrufendem Programm („Caller“) beendet
- Stack-„Frame“
  - der Bereich/Zustand einer einzelnen Prozedur-Instanziierung

Stack-Frame

- „Closure“: alle Daten für einen Funktionsaufruf
- Daten
  - Aufruf-Parameter der Funktion/Prozedur
  - Rücksprungadresse
  - lokale Variablen
  - temporäre Daten
- Verwaltung
  - beim Aufruf wird Speicherbereich zugeteilt „Setup“ Code
  - beim Return –"– freigegeben „Finish“ Code
- Adressenverweise („Pointer“)
  - Stackpointer `%esp` gibt das obere Ende des Stacks an
  - Framepointer `%ebp` gibt den Anfang des aktuellen Frame an

Beispiel: Stack-Frame

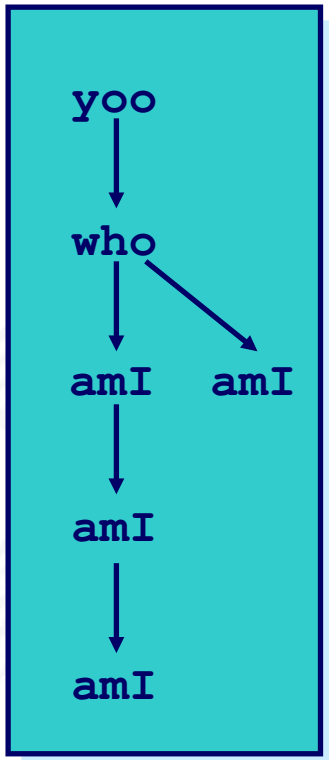
Code Structure

```
yoo(…)
{
    •
    •
    who();
    •
    •
}
```

```
who(…)
{
    • • •
    amI();
    • • •
    amI();
    • • •
}
```

```
amI(…)
{
    •
    •
    amI();
    •
    •
}
```

Call Chain

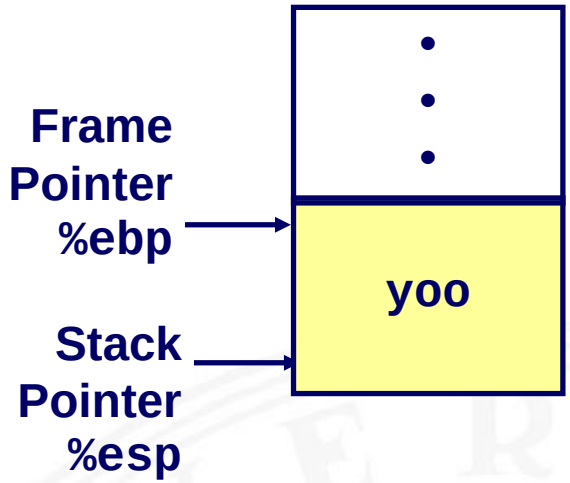


Beispiel: Stack-Frame (cont.)


```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

Call Chain

yoo



Beispiel: Stack-Frame (cont.)

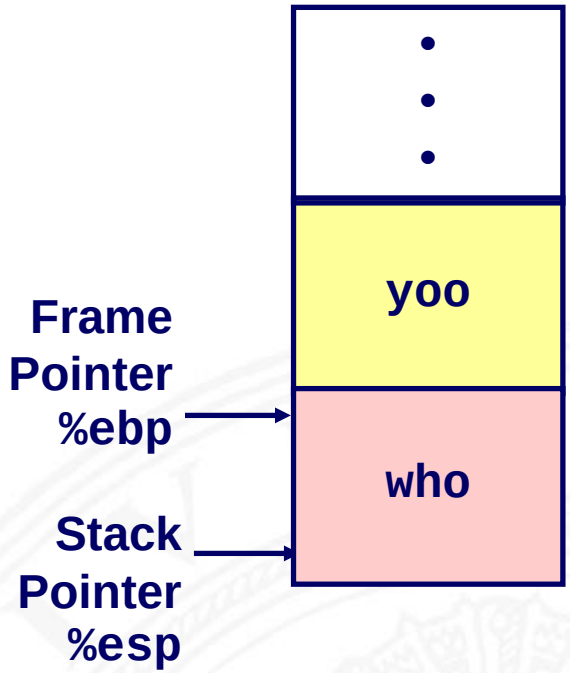

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

Call Chain

yoo

↓

who

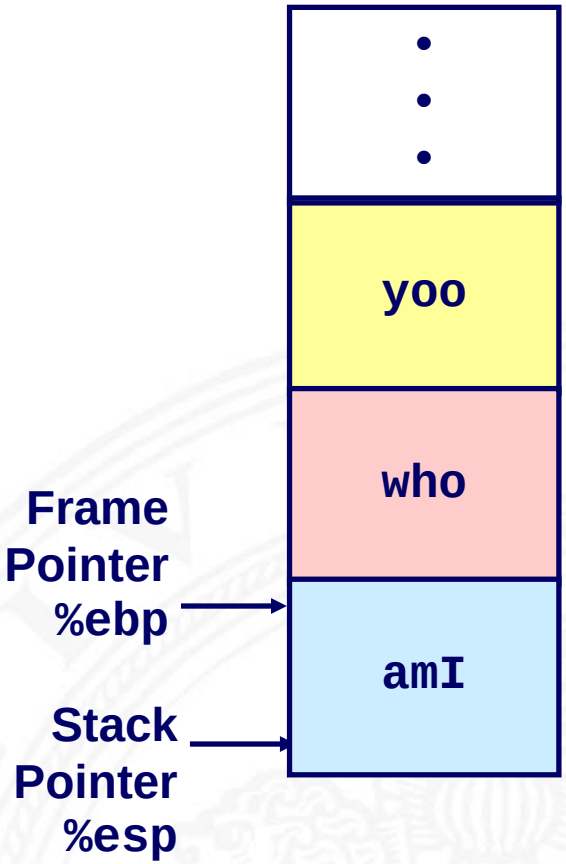


Beispiel: Stack-Frame (cont.)


```
amI(…)
{
    •
    •
    amI();
    •
    •
}
```

Call Chain

yoo
↓
who
↓
amI

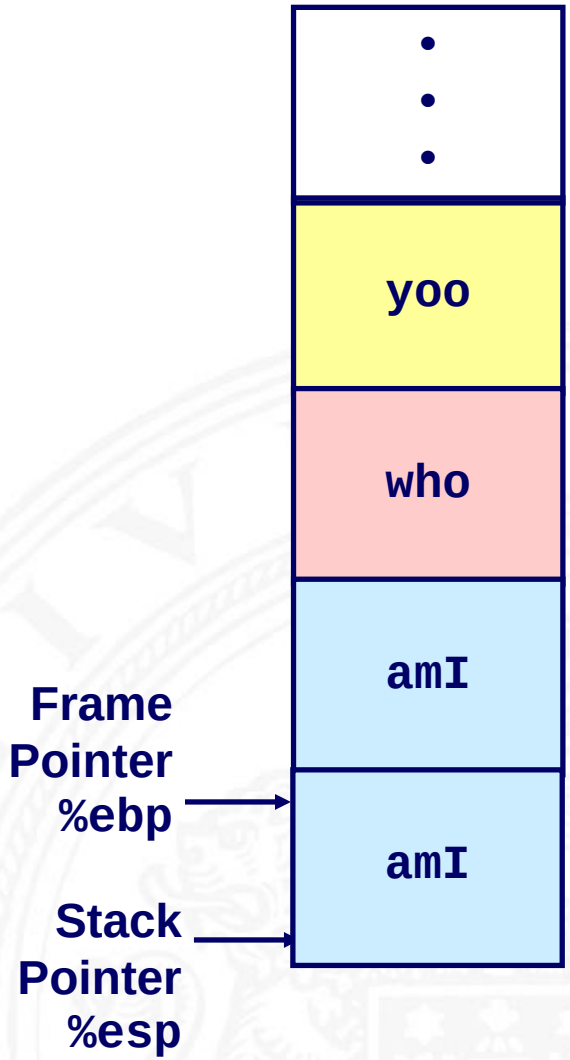


Beispiel: Stack-Frame (cont.)


```
amI(…)
{
    •
    •
    amI();
    •
    •
}
```

Call Chain

yoo
↓
who
↓
amI
↓
amI

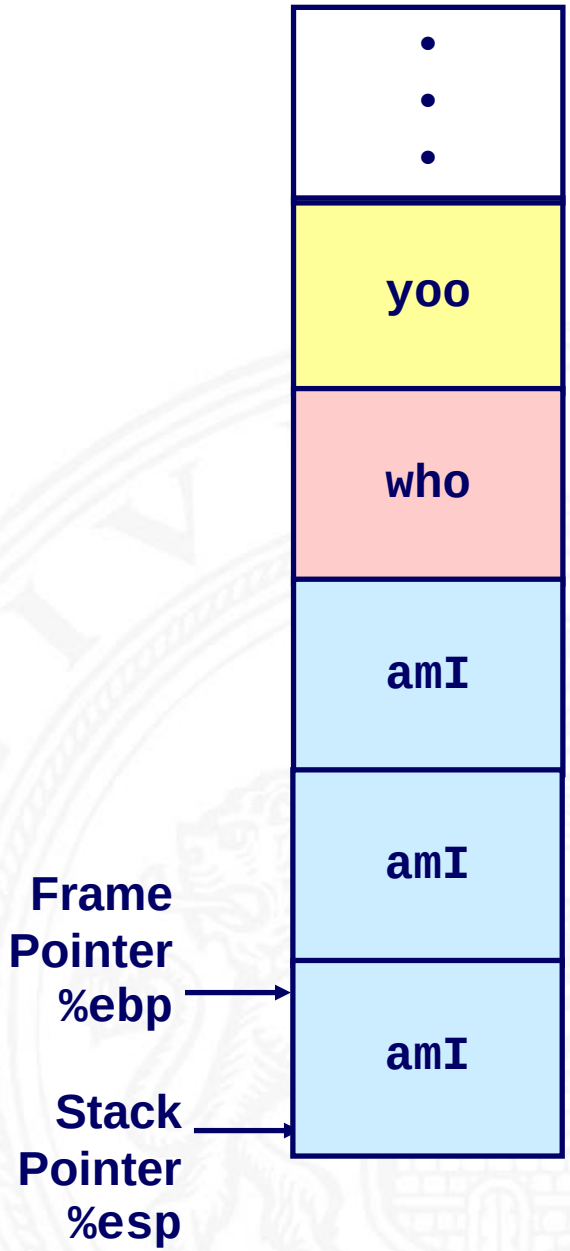


Beispiel: Stack-Frame (cont.)

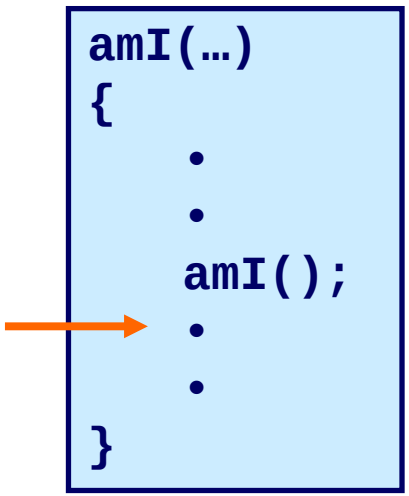

```
amI(…)
{
    •
    •
    amI();
    •
    •
}
```

Call Chain

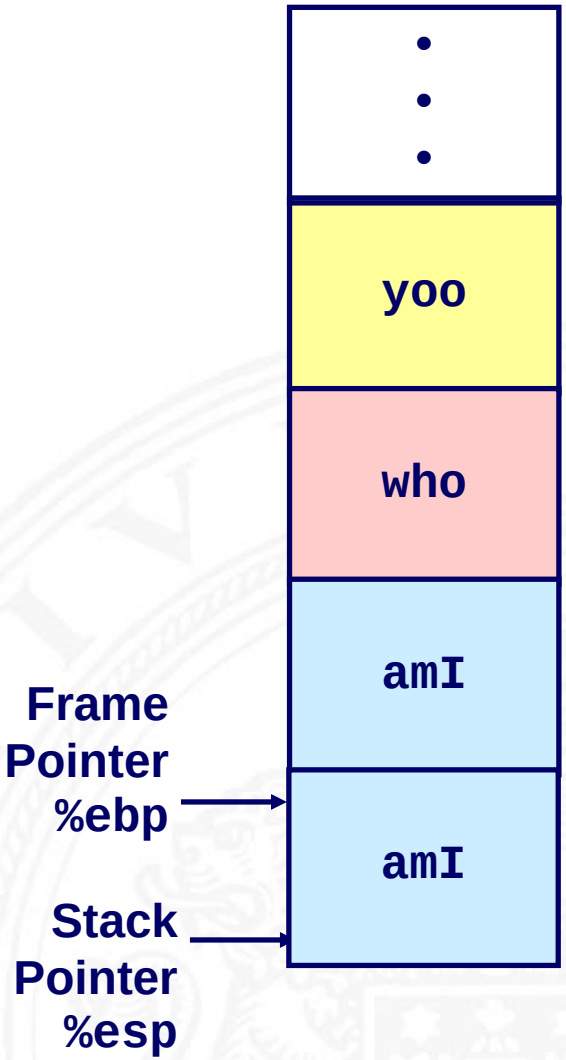
yoo → who → amI → amI → amI



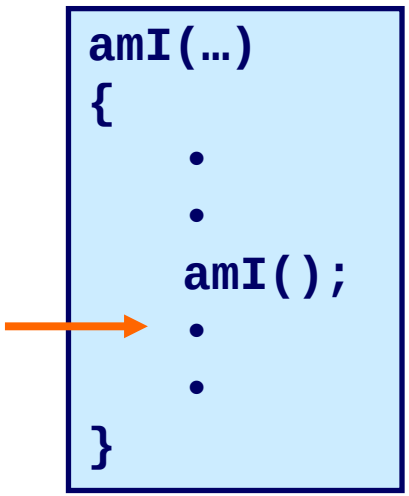
Beispiel: Stack-Frame (cont.)



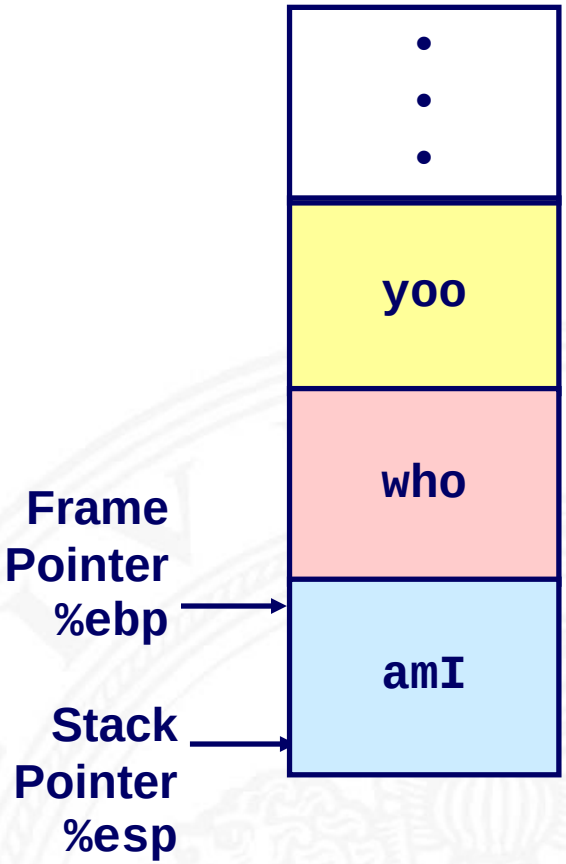
Call Chain



Beispiel: Stack-Frame (cont.)



Call Chain

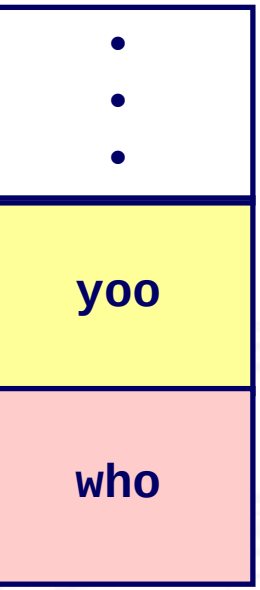


Beispiel: Stack-Frame (cont.)

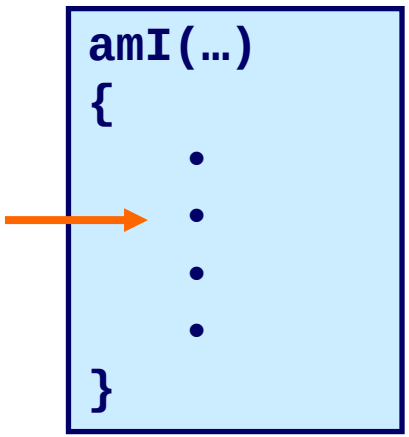

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

Call Chain

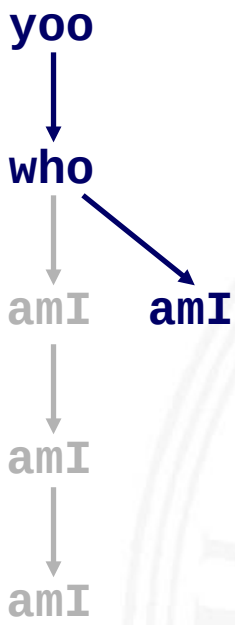
yoo
↓
who
↓
amI
↓
amI
↓
amI

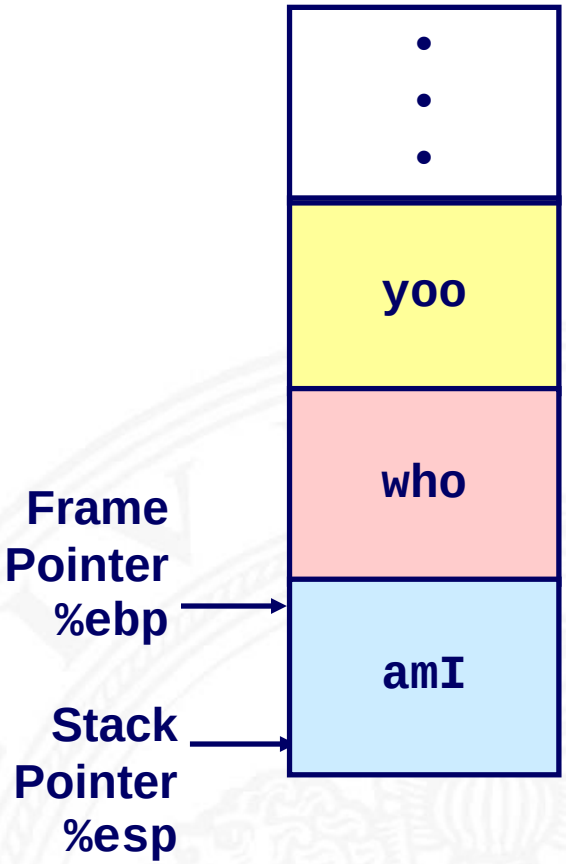


Beispiel: Stack-Frame (cont.)



Call Chain

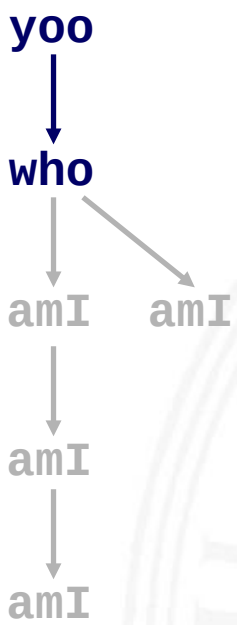


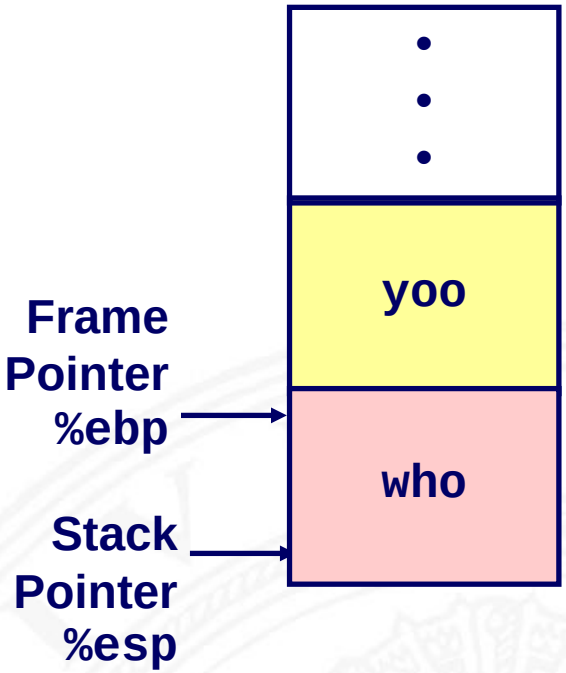


Beispiel: Stack-Frame (cont.)


```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

Call Chain

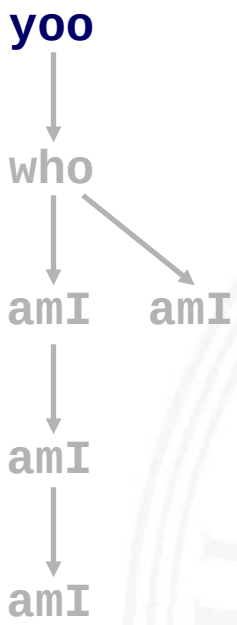


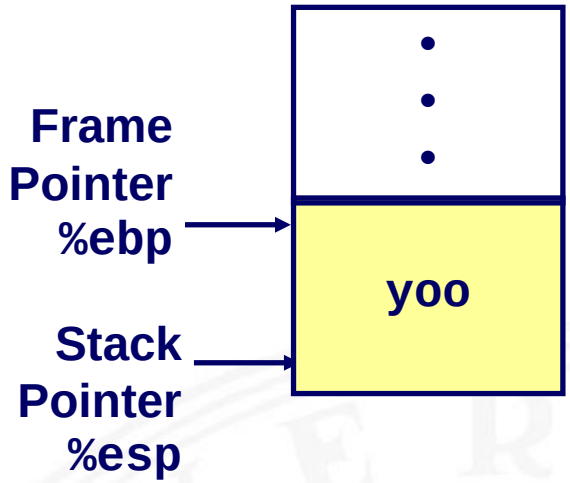


Beispiel: Stack-Frame (cont.)


```
yoo(…)
{
    •
    •
    who();
    •
    •
}
```

Call Chain





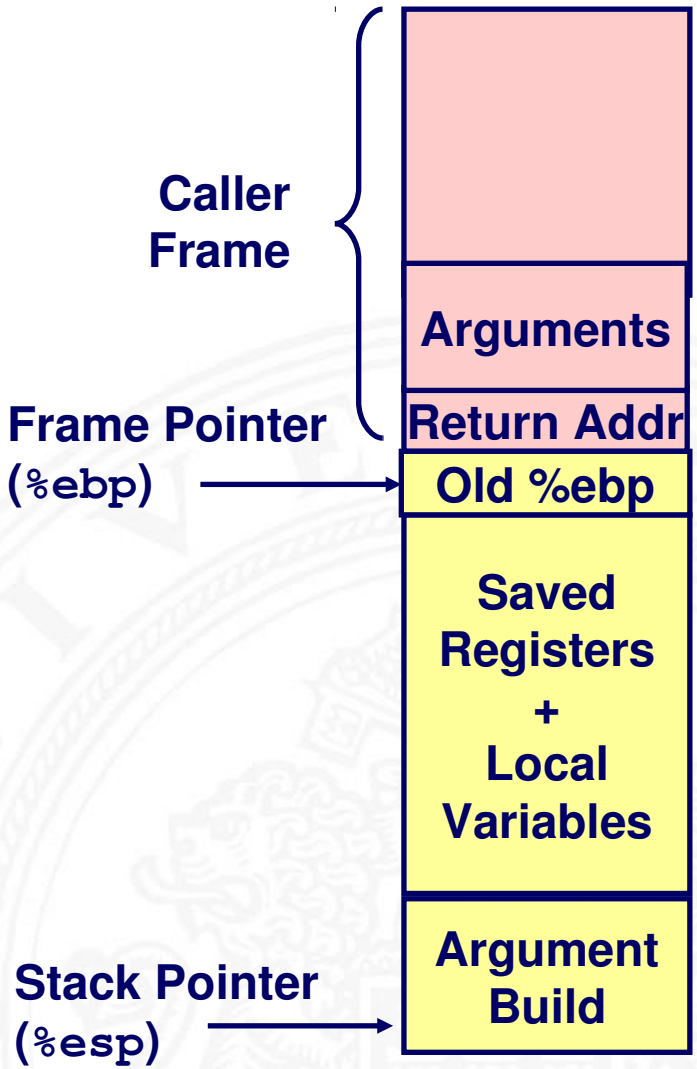
x86/Linux Stack-Frame

aktueller Stack-Frame / „Callee“

- von oben nach unten organisiert „Top“. . . „Bottom“
- Parameter für weitere Funktion die aufgerufen wird `call`
- lokale Variablen
  - wenn sie nicht in Registern gehalten werden können
- gespeicherter Registerkontext
- Zeiger auf vorherigen Frame

„Caller“ Stack-Frame

- Rücksprungadresse
  - von `call`-Anweisung erzeugt
- Argumente für aktuellen Aufruf



Register Sicherungskonventionen

- yoo („Caller“) ruft Prozedur `who` („Callee“) auf

```
yoo:
    • • •
    movl $15213, %edx
    call who
    addl %edx, %eax
    • • •
    ret
```

```
who:
    • • •
    movl 8(%ebp), %edx
    addl $91125, %edx
    • • •
    ret
```

- kann `who` Register für vorübergehende Speicherung benutzen?
  - Inhalt von `%edx` wird von `who` überschrieben

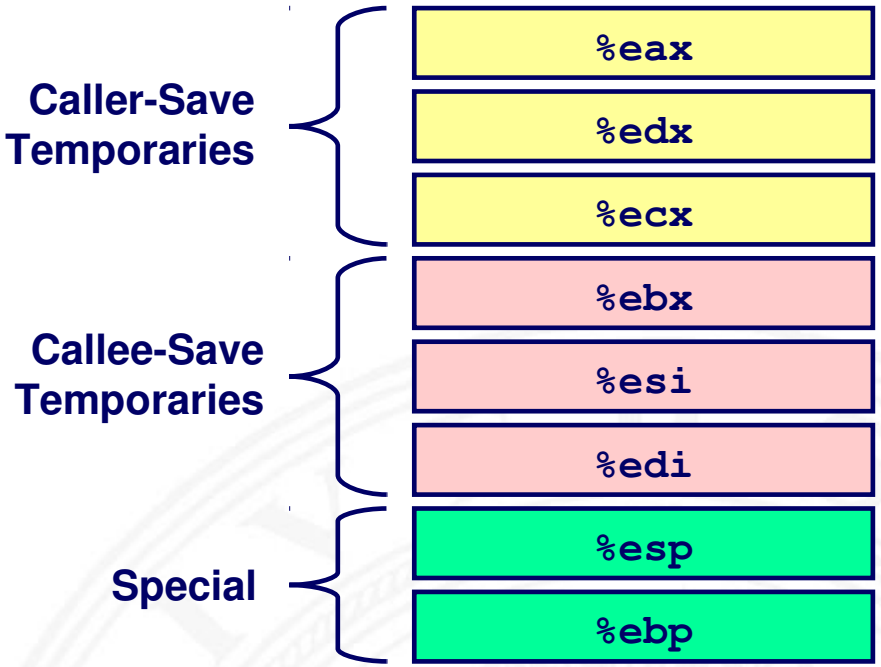
$\Rightarrow$ zwei mögliche Konventionen

- „Caller-Save“
  yoo speichert in seinen Frame vor Prozeduraufruf
- „Callee-Save“
  `who` speichert in seinen Frame vor Benutzung

x86/Linux Register Verwendung

Integer Register

- zwei spezielle Register
  - %ebp, %esp
- „Callee-Save“ Register
  - %ebx, %esi, %edi
  - vor Benutzung werden „alte“ Werte auf dem Stack gesichert
- „Caller-Save“ Register
  - %eax, %edx, %ecx
  - “Caller” sichert diese Register
- Register %eax speichert auch den zurückgelieferten Wert



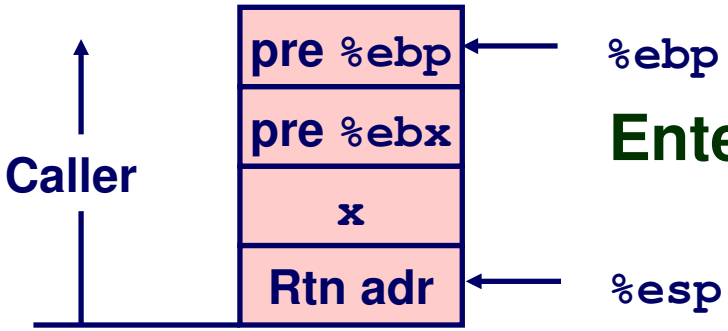
Beispiel: Rekursive Fakultät


```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1);
  return rval * x;
}
```

- %eax
  - benutzt ohne vorheriges Speichern
- %ebx
  - am Anfang speichern
  - am Ende zurückschreiben

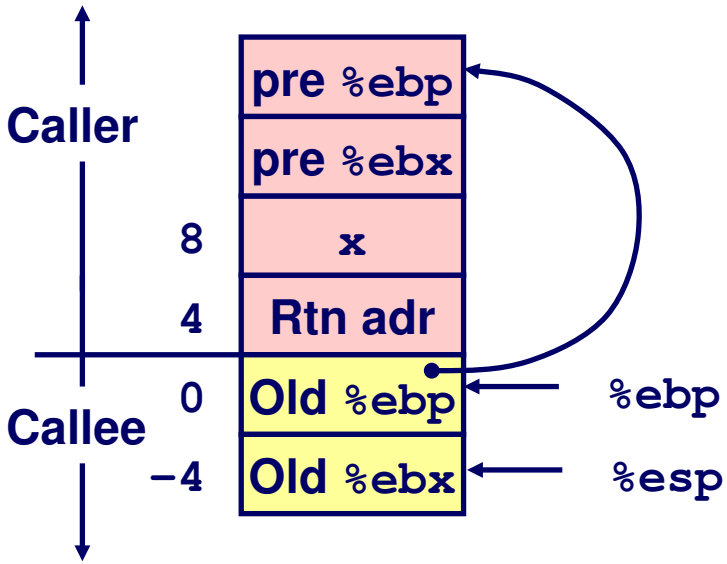
```
.globl rfact
    .type
rfact,@function
rfact:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 8(%ebp),%ebx
    cmpl $1,%ebx
    jle .L78
    leal -1(%ebx),%eax
    pushl %eax
    call rfact
    imull %ebx,%eax
    jmp .L79
    .align 4
.L78:
    movl $1,%eax
.L79:
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Beispiel: `rfact` – Stack „Setup“



Entering Stack

```
rfact:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```



Beispiel: `rfact` – Rekursiver Aufruf

Recursion

```
  movl 8(%ebp),%ebx    # ebx = x
  cmpl $1,%ebx         # Compare x : 1
  jle .L78             # If <= goto Term
  leal -1(%ebx),%eax   # eax = x-1
  pushl %eax           # Push x-1
  call rfact           # rfact(x-1)
  imull %ebx,%eax      # rval * x
  jmp .L79             # Goto done
.L78:                # Term:
  movl $1,%eax         # return val = 1
.L79:                # Done:
```

```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1) ;
  return rval * x;
}
```

Registers

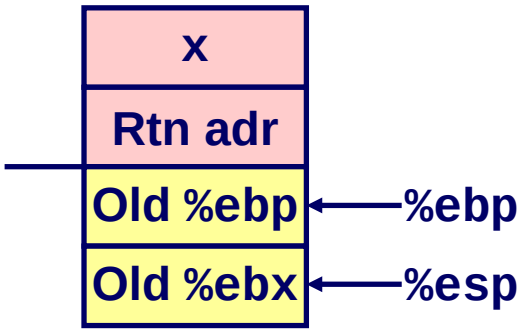
`%ebx` Stored value of x

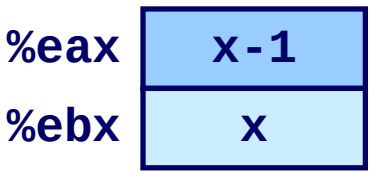
`%eax`

- Temporary value of `x-1`
- Returned value from `rfact(x-1)`
- Returned value from this call

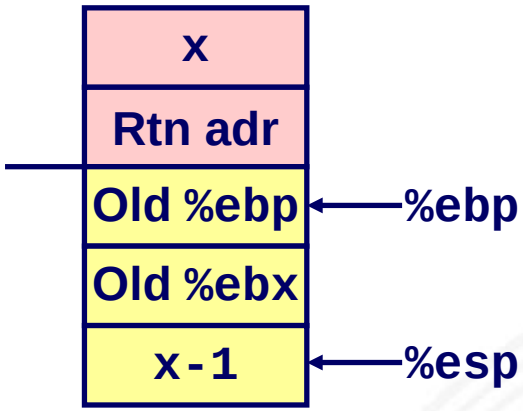
Beispiel: `rfact` – Rekursion

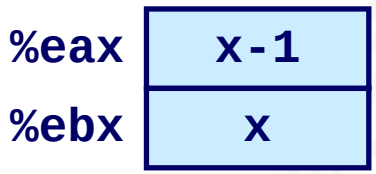

```
leal -1(%ebx),%eax
```



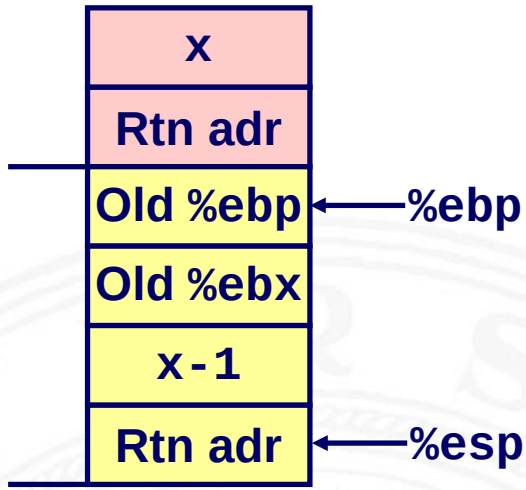


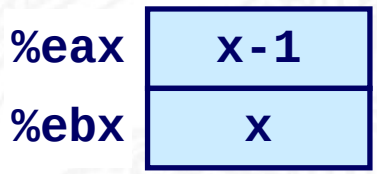

```
pushl %eax
```





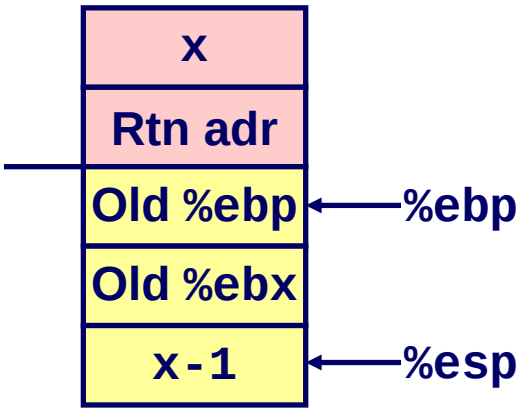

```
call rfact
```



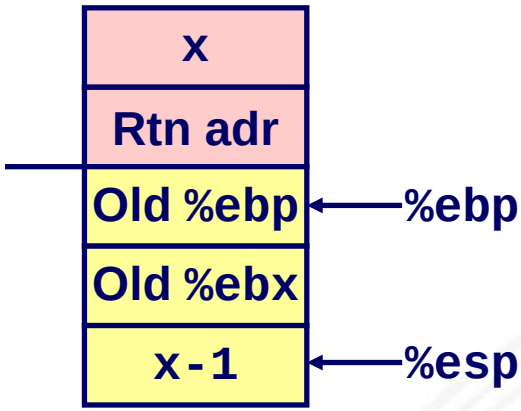


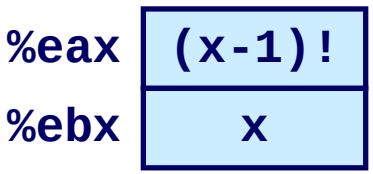
Beispiel: `rfact` – Ergebnisübergabe

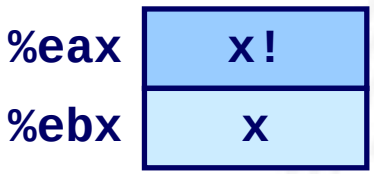
Return from Call




```
imull %ebx,%eax
```



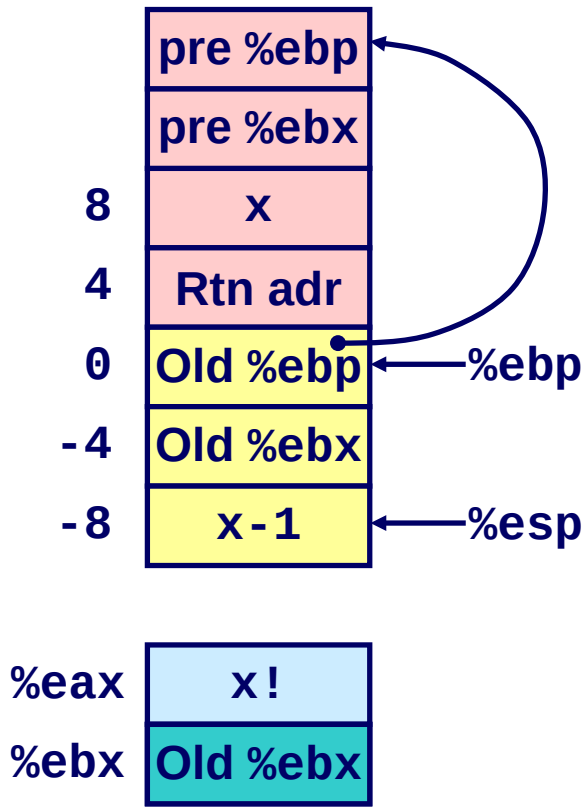


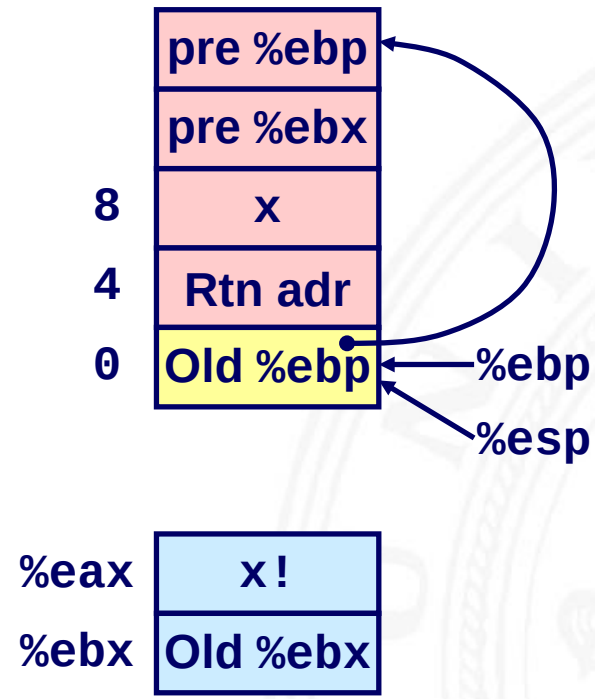


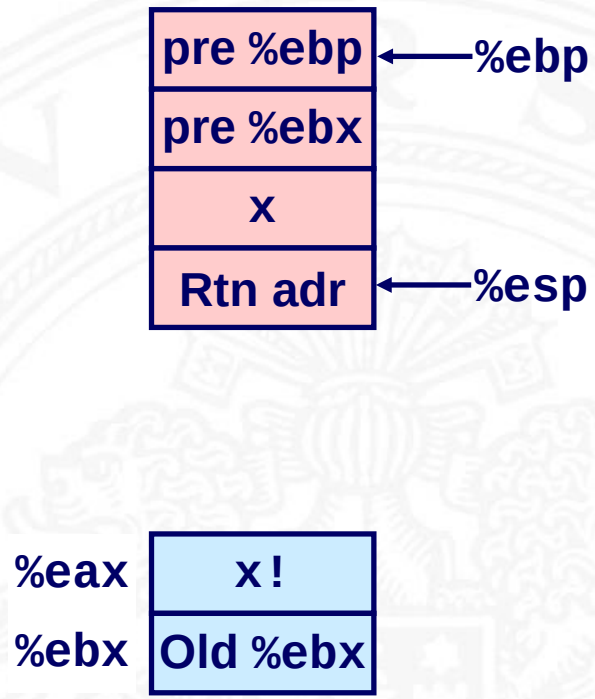
Assume that `rfact(x-1)` returns `(x-1)!` in register `%eax`

Beispiel: `rfact` – Stack „Finish“


```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```







Zeiger auf Adresse / *call by reference*

- Variable der aufrufenden Funktion soll modifiziert werden

$\Rightarrow$ Adressenverweis (*call by reference*)

- Beispiel: `sfact`

Recursive Procedure

```
void s_helper
  (int x, int *accum)
{
  if (x <= 1)
    return;
  else {
    int z = *accum * x;
    *accum = z;
    s_helper (x-1,accum);
  }
}
```

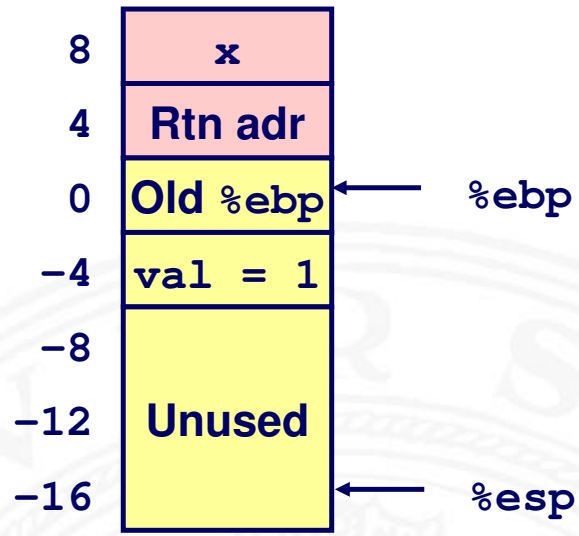
Top-Level Call

```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

Beispiel: sfact

Initial part of sfact

```
_sfact:
   pushl %ebp          # Save %ebp
   movl %esp,%ebp      # Set %ebp
   subl $16,%esp       # Add 16 bytes
   movl 8(%ebp),%edx   # edx = x
   movl $1,-4(%ebp)    # val = 1
```



- lokale Variable val auf Stack speichern
  - Pointer auf val
  - berechnen als -4(%ebp)
- Push val auf Stack
  - zweites Argument
  - movl $1, -4(%ebp)

```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

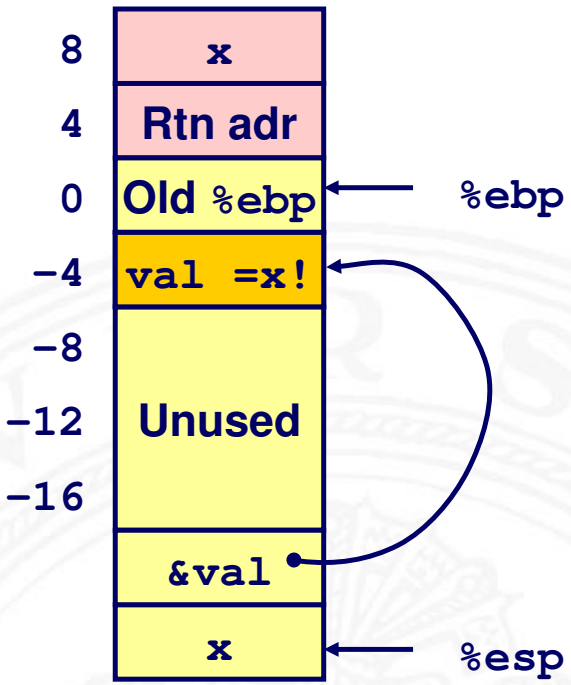
Beispiel: sfact – Pointerübergabe bei Aufruf

Calling s_helper from sfact

```
leal -4(%ebp),%eax # Compute &val
pushl %eax         # Push on stack
pushl %edx         # Push x
call s_helper      # call
movl -4(%ebp),%eax # Return val
• • •              # Finish
```

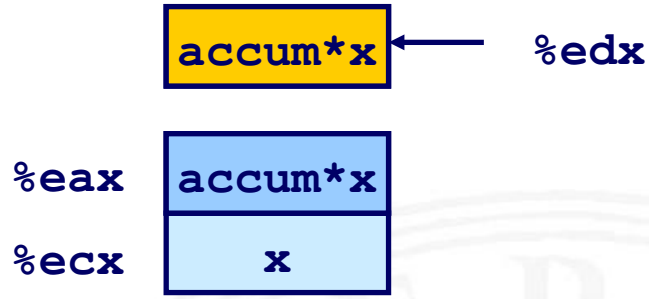
```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

Stack at time of call



Beispiel: sfact – Benutzung des Pointers


```
void s_helper
   (int x, int *accum)
{
   • • •
     int z = *accum * x;
     *accum = z;
   • • •
}
```




```
   • • •
   movl %ecx,%eax     # z = x
   imull (%edx),%eax  # z *= *accum
   movl %eax,(%edx)   # *accum = z
   • • •
```

- Register %ecx speichert x
- Register %edx mit Zeiger auf accum

Zusammenfassung – Stack

- Stack ermöglicht Funktionsaufrufe und Rekursion
  - lokaler Speicher für jeden Prozeduraufruf („call“)
    - Instanziierungen beeinflussen sich nicht
    - Adressierung lokaler Variablen und Argumente ist relativ zur Stackposition (Framepointer)
  - grundlegendes (Stack-) Prinzip
    - Prozeduren terminieren in umgekehrter Reihenfolge der Aufrufe

- x86 Prozeduren sind Kombination von Anweisungen und Konventionen
  - `call`- und `ret`-Befehle
  - Konventionen zur Registerverwendung
    - „Caller-Save“ / „Callee-Save“
    - `%ebp` und `%esp`
  - festgelegte Organisation des Stack-Frame

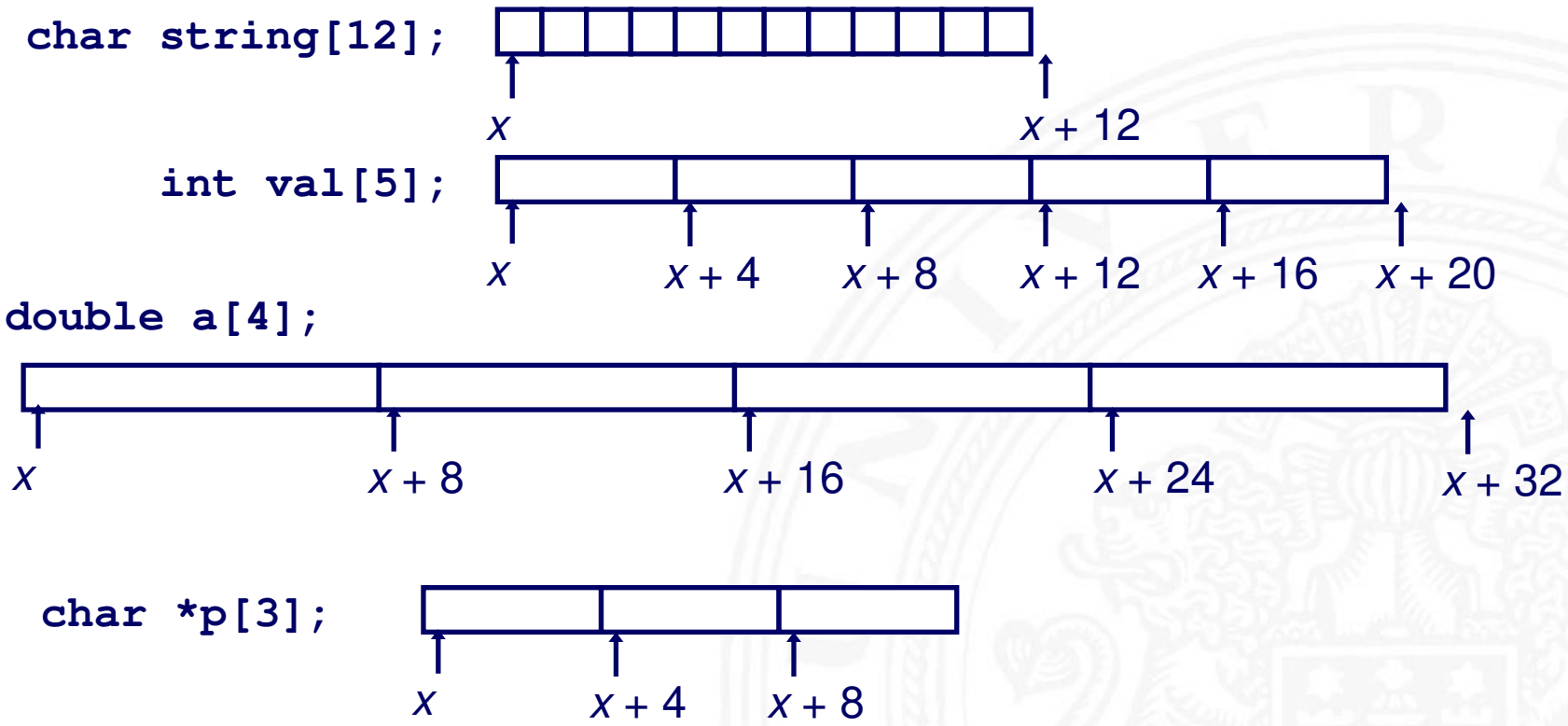
Elementare Datentypen

- Ganzzahl (Integer)
  - wird in allgemeinen Registern gespeichert
  - abhängig von den Anweisungen: *signed*/*unsigned*
Intel	gas	Bytes	C	gas: *Gnu ASsembler*
byte	b	1	[unsigned] char
word	w	2	[unsigned] short
double word	l	4	[unsigned] int

- Gleitkomma (Floating Point)
  - wird in Gleitkomma-Registern gespeichert
Intel	gas	Bytes	C	gas: *Gnu ASsembler*
Single	s	4	float
Double	l	8	double
Extended	t	10/12	long double

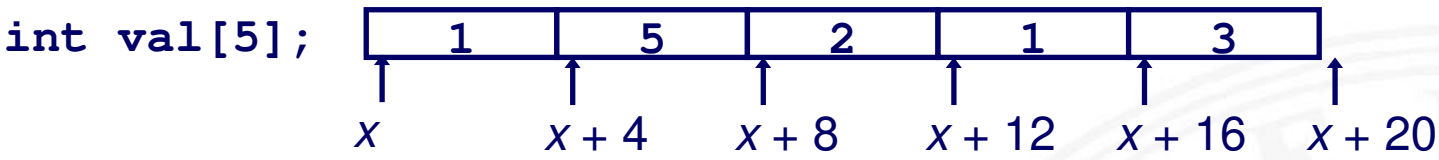
Array: Allokation / Speicherung

- `T A[N];`
  - Array `A` mit Daten von Typ `T` und `N` Elementen
  - fortlaufender Speicherbereich von `N` $\times$ sizeof(`T`) Bytes



Array: Zugriffskonvention

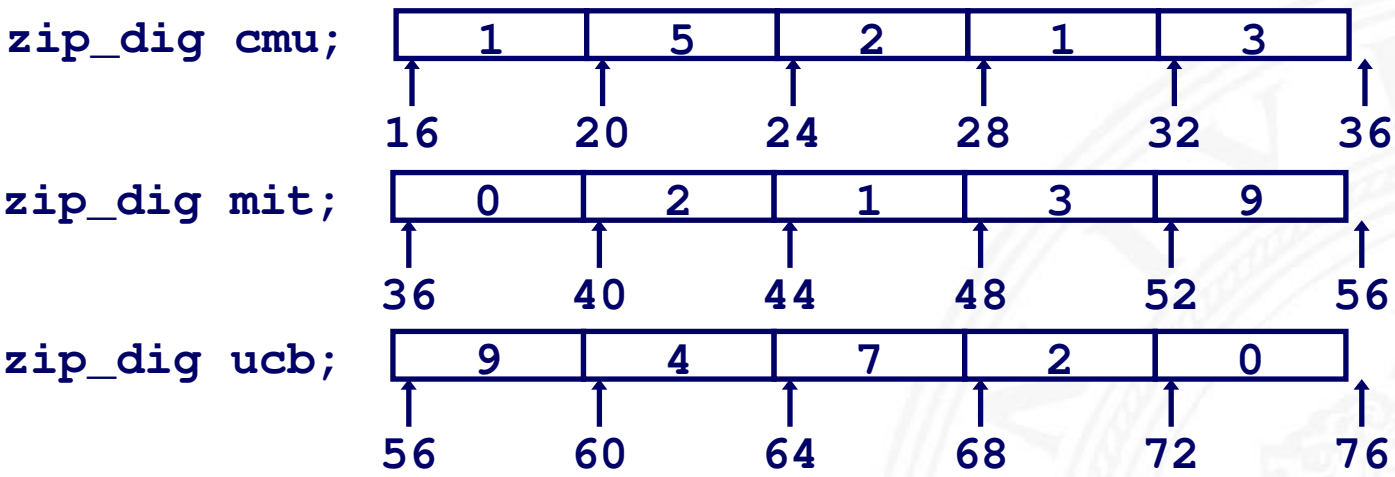
- `T A[N];`
  - Array `A` mit Daten von Typ `T` und `N` Elementen
  - Bezeichner `A` zeigt auf erstes Element des Arrays: Element 0



Reference	Type	Value
`val[4]`	`int`	`3`
`val`	`int *`	x
`val+1`	`int *`	$x + 4$
`&val[2]`	`int *`	$x + 8$
`val[5]`	`int`	`??`
`*(val+1)`	`int`	`5`
`val + i`	`int *`	$x + 4\,i$

Beispiel: einfacher Arrayzugriff


```
typedef int zip_dig[5];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



Beispiel: einfacher Arrayzugriff (cont.)

- Register `%edx` : Array Startadresse
  `%eax` : Array Index
- Adressieren von $4 \times$ `%eax` $+$ `%edx`

$\Rightarrow$ Speicheradresse `(%edx,%eax,4)`

```
int get_digit
  (zip_dig z, int dig)
{
  return z[dig];
}
```

Memory Reference Code

```
  # %edx = z
  # %eax = dig
   movl (%edx,%eax,4),%eax # z[dig]
```

- keine Bereichsüberprüfung („*bounds checking*“)
- Verhalten außerhalb des Indexbereichs ist Implementierungsabhängig

Beispiel: Arrayzugriff mit Schleife

- Originalcode

```
int zd2int(zip_dig z)
{
  int i;
  int zi = 0;
  for (i = 0; i < 5; i++) {
    zi = 10 * zi + z[i];
  }
  return zi;
}
```

- transformierte Version: gcc
  - Laufvariable `i` eliminiert
  - aus Array-Code wird Pointer-Code
  - in „do-while“ Form
  - Test bei Schleifeneintritt unnötig

```
int zd2int(zip_dig z)
{
  int zi = 0;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while(z <= zend);
  return zi;
}
```

Beispiel: Arrayzugriff mit Schleife (cont.)

- Register %ecx : z
  %eax : zi
  %ebx : zend
- *z + 2*(zi+4*zi)
  ersetzt 10*zi + *z
- z++ Inkrement: +4

```
int zd2int(zip_dig z)
{
  int zi = 0;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while(z <= zend);
  return zi;
}
```

```
  # %ecx = z
  xorl %eax,%eax          # zi = 0
  leal 16(%ecx),%ebx      # zend  = z+4
.L59:
  leal (%eax,%eax,4),%edx # 5*zi
  movl (%ecx),%eax        # *z
  addl $4,%ecx            # z++
  leal (%eax,%edx,2),%eax # zi = *z + 2*(5*zi)
  cmpl %ebx,%ecx          # z : zend
  jle .L59                # if <= goto loop
```

Zwei- und mehrdimensionale Arrays

$(N \times M)$ Matrizen? drei grundsätzliche Möglichkeiten

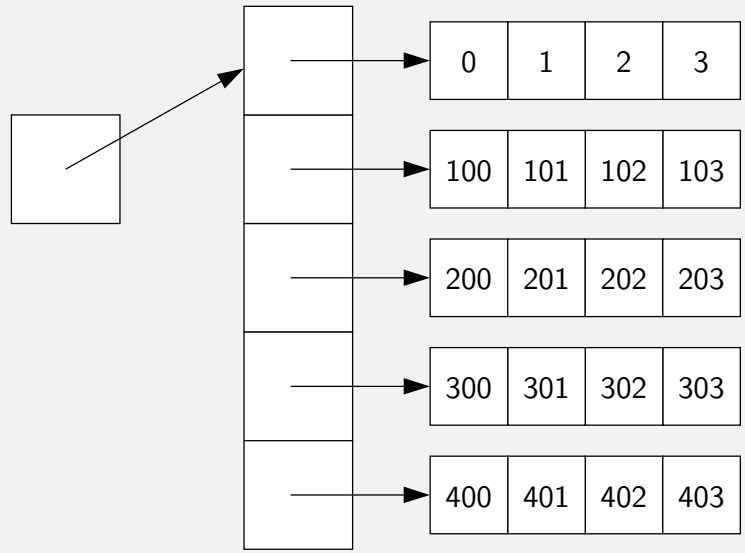
1. Array von Pointern auf Zeilen-Arrays von Elementen — Java
   - sehr flexibel, auch für nicht-rechteckige Layouts
   - Sharing/Aliasing von Zeilen möglich

- Array von $N \times M$ Elementen und passende Adressierung
  2. row-major Anordnung — C, C++
  3. column-major Anordnung — Matlab, FORTRAN

- bei Verwendung/Mischung von Bibliotheksfunktionen aus anderen Sprachen unbedingt berücksichtigen

Java: Array von Pointern auf Arrays von Elementen


```
class MatrixDemo {
  int matrix[][]; // matrix[i]->

  public MatrixDemo( int NROWS, int NCOLS ) {
    matrix = new int[NROWS][NCOLS];
    for( int r=0; r < matrix.length; r++ ) {
      for( int c =0; c < matrix[r].length; c++ ) {
        matrix[r][c] = 100*r + c;
      }
    }
    // int[] row0 = matrix[0];
    // int    m23 = matrix[2][3];
  }
  public int get( int r, int c ) {
    return matrix[r][c];
  }
}
```



Zweidimensionale Arrays in C


```
int n_rows = 4; int n_cols = 5;
int matrix[4][5];  // 00 01 02 03 04  10 11 12 13 14 .. 34
int schach[8][8] = { 0,1,2,3,4,5,6,7, 10,11,12,13,.. 77 };

int m00 = matrix[0][0]; // *(matrix[0] + 0);
int m01 = matrix[0][1]; // *(matrix[0] + 1);
int m20 = matrix[2][0]; // *(matrix[2] + 0);
int m34 = matrix[3][4]; // *(matrix[3] + 4);

int *elem = &(matrix[2][2]);
elem++;          // nächste Spalte (bzw. Wraparound);
elem+= n_cols; // nächste Zeile
```

- Arrayelemente in „*row-major*“ Anordnung, Spalten fortlaufend
- „*column-major*“ ist transponiert: 00 10 20 . . . 01 11 21 . . . 34

Mehrdimensionale Arrays: entsprechend

- d-dimensionales $N_1 \times N_2 \times \cdots \times N_d$ Array
  - Element addressiert mit Tupel $(n_1, n_2, \ldots, n_d)$, mit d (zero-offset) Indizes $n_k \in [0, N - K - 1]$

- row-major Anordnung: letzte Dimension ist fortlaufend

$$n_d + N_d \cdot (n_{d-1} + N_{d-1} \cdot (n_{d-2} + N_{d-2} \cdot (\cdots))) = \sum_{k=1}^{d} \left(\prod_{\ell=k+1}^{d} N_\ell \right) n_k$$

- column-major Anordnung: erste Dimension ist fortlaufend

$$n_1 + N_1 \cdot (n_2 + N_2 \cdot (n_3 + N_3 \cdot (\cdots + N_{d-1} n_d) \cdots)) = \sum_{k=1}^{d} \left(\prod_{\ell=1}^{k-1} N_\ell \right) n_k$$

- oder Arrays von Arrays von Arrays auf Arrays auf Elemente

Strukturen (*Records*)

- Allokation eines zusammenhängenden Speicherbereichs
- Elemente der Struktur über Bezeichner referenziert
- verschiedene Typen der Elemente sind möglich

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

Memory Layout

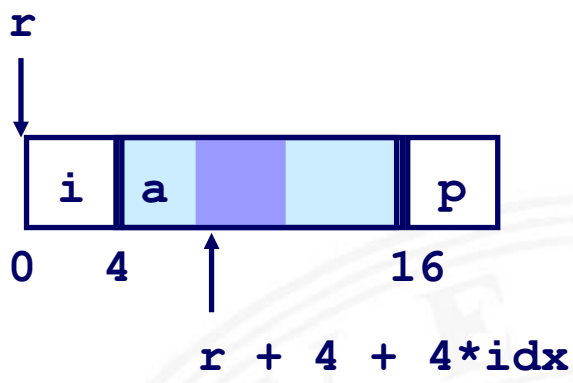

```
void
set_i(struct rec *r,
      int val)
{
  r->i = val;
}
```

Assembly

```
# %eax = val
# %edx = r
movl %eax,(%edx)    # Mem[r] = val
```

Strukturen: Zugriffskonventionen

- Zeiger `r` auf Byte-Array für Zugriff auf Struktur(element)
- Compiler bestimmt Offset für jedes Element




```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

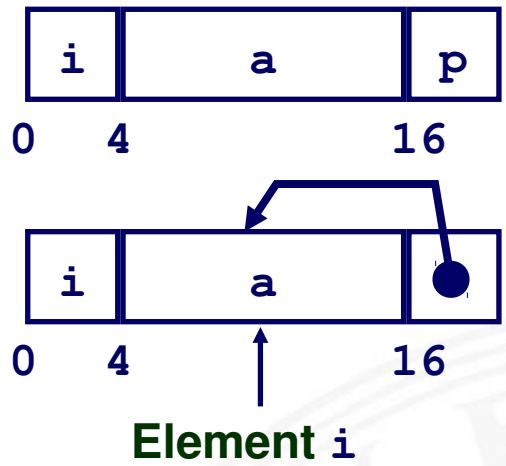
```
int *
find_a
 (struct rec *r, int idx)
{
  return &r->a[idx];
}
```

```
 # %ecx = idx
 # %edx = r
 leal 0(,%ecx,4),%eax   # 4*idx
 leal 4(%eax,%edx),%eax # r+4*idx+4
```

Beispiel: Strukturreferenzierung


```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

```
void
set_p(struct rec *r)
{
  r->p =
   &r->a[r->i];
}
```




```
# %edx = r
movl (%edx),%ecx          # r->i
leal 0(,%ecx,4),%eax      # 4*(r->i)
leal 4(%edx,%eax),%eax    # r+4+4*(r->i)
movl %eax,16(%edx)        # Update r->p
```

Ausrichtung der Datenstrukturen (*Alignment*)

- Datenstrukturen an Wortgrenzen ausrichten
  double- / quad-word
- sonst Problem
  - ineffizienter Zugriff über Wortgrenzen hinweg
  - virtueller Speicher und Caching

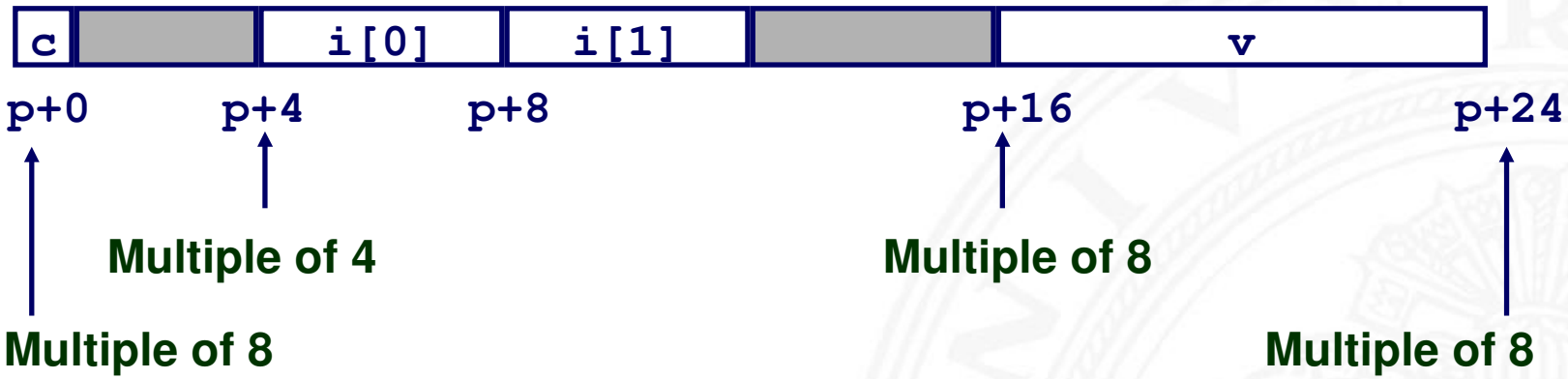
⇒ Compiler erzeugt „Lücken“ zur richtigen Ausrichtung

- typisches Alignment (IA32)

Länge	Typ	Windows	Linux
1 Byte	`char`	keine speziellen Verfahren	
2 Byte	`short`	Adressbits: ...0	...0
4 Byte	`int, float, char *`	–"– ...00	...00
8 Byte	`double`	–"– ...000	...00
12 Byte	`long double`	–"– –	...00

Beispiel: Structure Alignment


```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



Umsetzung von Objektorientierung?

- Klassen/Objekte verbinden Daten und Methoden
  - polymorphe Funktionen — name-mangling
  - Metadaten, run-time type-information — rtti
  - Vererbung und dynamischer Funktionsaufruf — vtable

- Grundidee
  - Datenstrukturen wie in Assembler/C
  - Schema zur Erzeugung eindeutiger Namen
  - zusätzliche Pointer auf Typ/Klassen-Information
  - zusätzliche Pointer auf Funktionstabelle(n)
  - Methodenaufrufe bekommen `this`-Pointer als Argument

  - gute Performance erfordert effiziente Implementierung
  - Details normalerweise vor dem Programmierer verborgen

Objekte: Exemplare von Klassen

- kombinieren Daten mit den zugehörigen Methoden

- Datenelemente wie C/Assembler Strukturen angeordnet
- ein Pointer auf die rtti-Datenstruktur
  - Debug-Infos: Name der Klasse, Datenelemente
  - Pointer auf Basisklasse(n)
  - Interfaces und Vererbungsinformation
- ein Pointer auf die vtable-Tabelle
  - Array mit allen Methoden der Klasse
  - Name-Mangling erhält Typ-Infos der Parameter
- aus Effizienzgründen diese Pointer ggf. mit negativem Offset
  - Speicherverwaltung berücksichtigt dies

Polymorphe Funktionen: Name-Mangling

- Programmierer arbeitet mit Klassen und deren Methoden
- polymorphe Funktionen, abhängig vom Typ der Parameter

```
class polymorph { public:
  float f( int i )   { return 2.0f*i; }
  float f( float f ) { return 1.5f*f; } ...
}
```

- aber: Assembler und Linker erwarten globale Funktionen

⇒ Name-Mangling („name decoration“) im Compiler

- Funktionsname gebildet aus Prefix + Name + Typkennung
- Prefix bildet Klassennamen/Namespace ab
- Typkennung zur eindeutigen Unterscheidung der Argumente
  `_ZN9polymorph1fEi _ZN9polymorph1fEf`
- Java: siehe Java Native Interface und `javah`-Tool

Methodenaufruf: this-Pointer

- bisher: Funktionen/Code vollkommen separat von Daten
- woher weiss eine Methode, zu welchem Objekt sie gehört?
- wie kommt eine Methode an Exemplarvariablen heran?

- Trick: Compiler übergibt `this` als erstes Argument
  - implizit, muss normalerweise nicht geschrieben werden
  - Pointer auf das aktuelle Objekt
  - Referenz auf Daten über `this->x`
  - Referenz auf Methoden über `this->vtable[offset]`
  - zusätzliche Funktionsparameter anschließend wie gewohnt

  - `Point3D.f( int i, int j )` wird intern zu
    `Point3D.f( Point3D *this, int i, int j )`

Methodenaufruf: this-Pointer

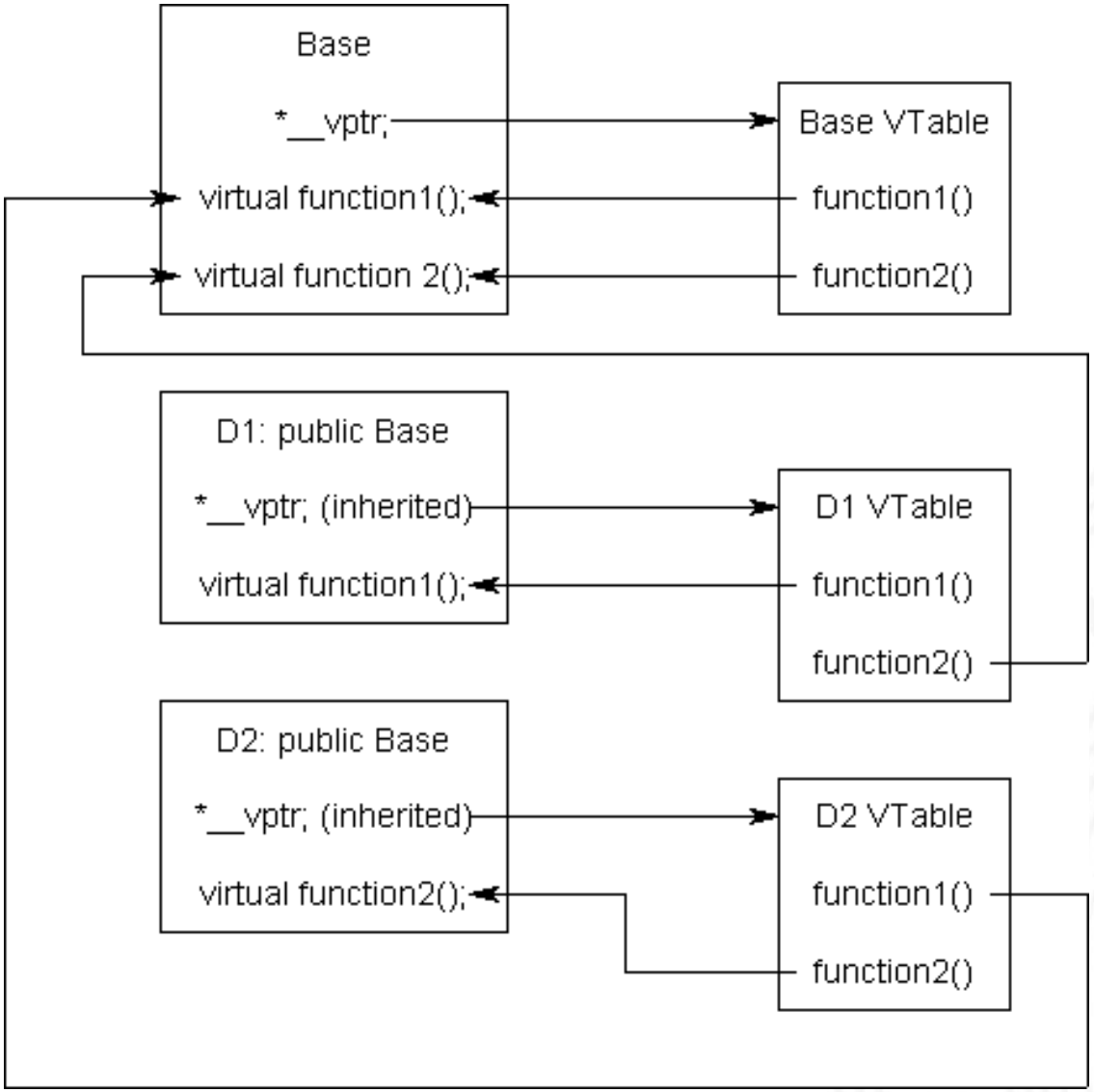

```
#include <stdio.h>

class Point3D {
  private: int x; int y; int z;
  public:
    Point3D( int _x, int _y, int _z ) { x = _x; y = _y; z = _z; }
    int getX() { return x; }
};

int main( int argc, char** argv ) {
  Point3D p( 42, 2, 3 );
  printf( "%d\n", p.getX() );
}
```

```
08048454 <_ZN7Point3D4getXEv>:
 8048454:       55                      push   %ebp
 8048455:       89 e5                   mov    %esp,%ebp
 8048457:       8b 45 08                mov    0x8(%ebp),%eax
 804845a:       8b 00                   mov    (%eax),%eax
 804845c:       5d                      pop    %ebp
 804845d:       c3                      ret
 804845e:       90                      nop
 804845f:       90                      nop
```

Virtual Table: Dynamischer Methodenaufruf

- Compiler kennt und sammelt alle Methoden einer Klasse
  - inklusive aller Methoden der Basisklassen
- erzeugt vtable Array mit Pointer auf die Funktionen
  - Aufruf der Funktionen als `*((this->vtable)+offset)()` wobei der Offset die jeweilige Methode auswählt
    - wieder `this`-Pointer als erster Parameter
    - weitere Parameter anschließend auf dem Stack
    - ein zusätzlicher Speicherzugriff (vergl. mit direktem Aufruf)
  - vererbte Methoden zeigen auf Code der Basisklasse
  - überschriebene Methoden zeigen auf Code der Unterklasse
  - `super.f()` durch Zugriff auf vtable der Basisklasse

Virtual Table: Vererbung



LearnCpp.Com: 12.5 the virtual table

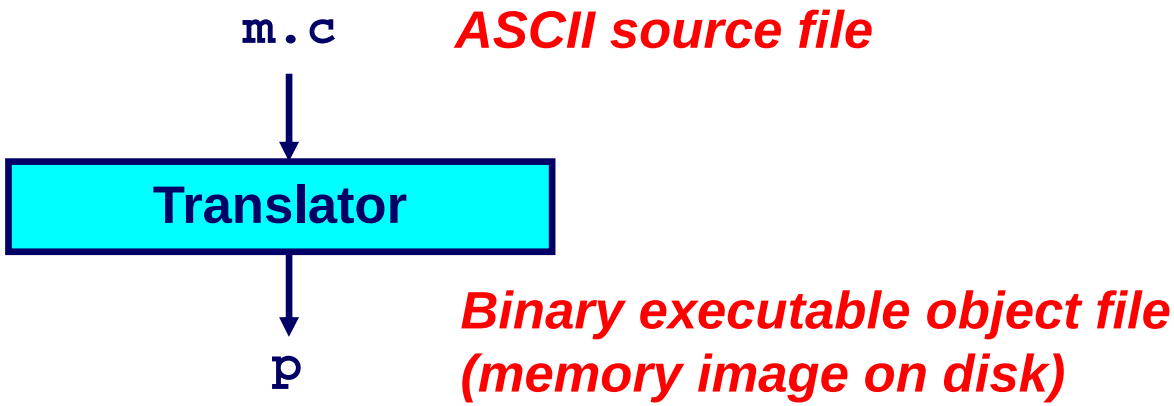
Zusammenfassung – Datentypen

- Arrays
  - fortlaufend zugeteilter Speicher
  - Adressverweis auf das erste Element
  - keine Bereichsüberprüfung (*Bounds Checking*)
- Compileroptimierungen
  - Compiler wandelt Array-Code in Pointer-Code um
  - verwendet Adressierungsmodi um Arrayindizes zu skalieren
  - viele Tricks, um die Array-Indizierung in Schleifen zu verbessern
- Strukturen
  - Bytes werden in der ausgewiesenen Reihenfolge zugeteilt
  - ggf. Leerbytes, um die richtige Ausrichtung zu erreichen
- Objekte
  - wie Strukturen, zwei extra Pointer auf Typ-Infos und vtable
  - Methodenaufruf über vtable mit this-Pointer

Linker und Loader

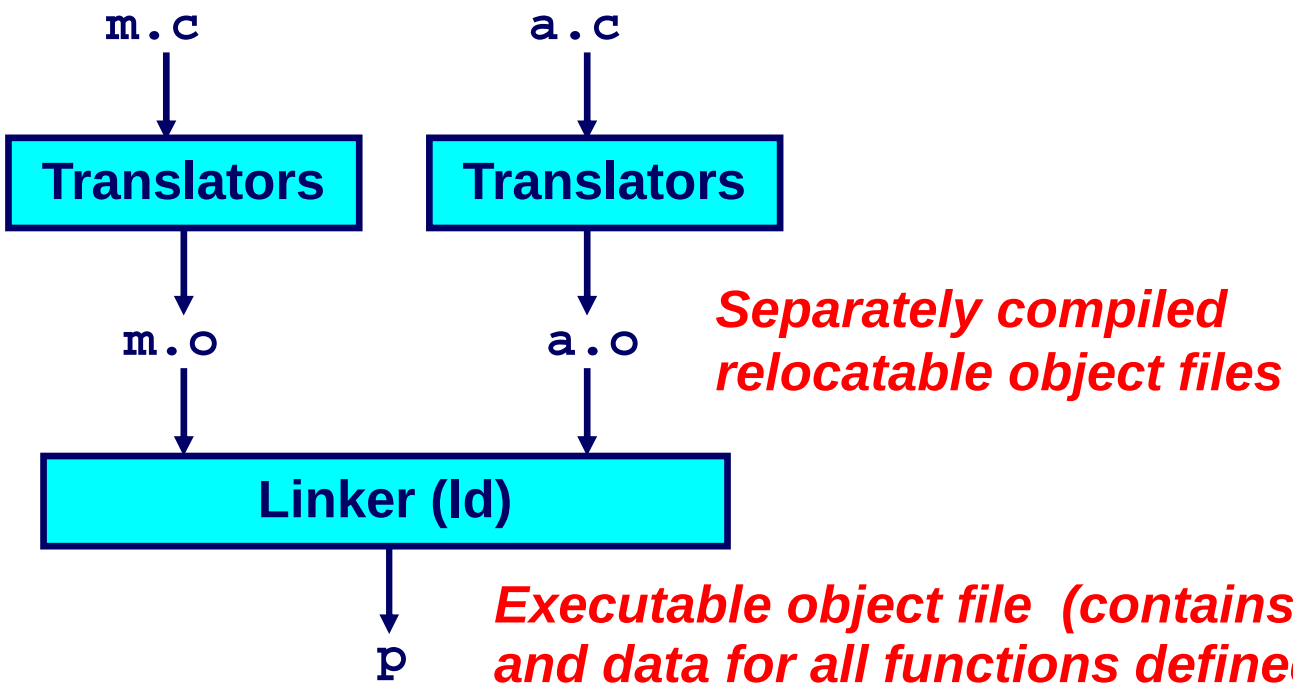
- Statisches Linken
- Object-Dateien (ELF)
- Statische Funktionsbibliotheken
- Loading
- Dynamische Funktionsbibliotheken (shared libraries)

Bisher: gesamtes Programm in einer Quelldatei



- Probleme
  - schlechte Effizienz: jede kleine Änderung erfordert volle Neu-Compilierung des Programms
  - keine Modularisierung: wie können wichtige Funktionen wiederverwendet werden? (z.B. `malloc`, `printf`)
- Lösung
  - Statisches Binden („static linking“)

Static Linking: Konzept



- Quelltext auf mehrere Dateien aufgeteilt
- einzeln in verschiebbaren Objektcode compiliert *position-independent code* (PIC)
- Linker baut daraus eine ausführbare Datei

Static Linking: Aufgaben des Linkers

- Zusammenführen der einzelnen (.o) Objektdateien in eine vollständige kombinierte Objektdatei
- Suchen der referenzierten Funktionen — external references
- Relozieren aller Speicherreferenzen — relocate symbols
  - für Daten — `int *xp=&x;`
  - und Funktionen — `printf();`
  - nicht aufgerufene Funktionen werden eliminiert
- Compiler(-driver) kümmert sich um Aufruf der einzelnen Tools
  - Präprozessor (`cpp`), Compiler (`cc1`), Assembler (`gas`) und Linker (`ld`)
  - „Finetuning“ und Reihenfolge über Kommandozeilen-Parameter

Static Linking: Vorteile

- Programm aus übersichtlichen Modulen zusammengesetzt
- erlaubt den Aufbau von Funktionsbibliotheken, z.B. mathematische Funktionen, Standard C-Library, Datenstrukturen, TCP/IP, Grafik ...

$\Rightarrow$ schnellere Entwicklung: nur geänderte Quelltexte müssen neu compiliert werden, Linken ist viel schneller als Compilieren

$\Rightarrow$ kompakte Programme: das ausführbare Programm enthält nur die tatsächlich benutzten Funktionen aus den Bibliotheken

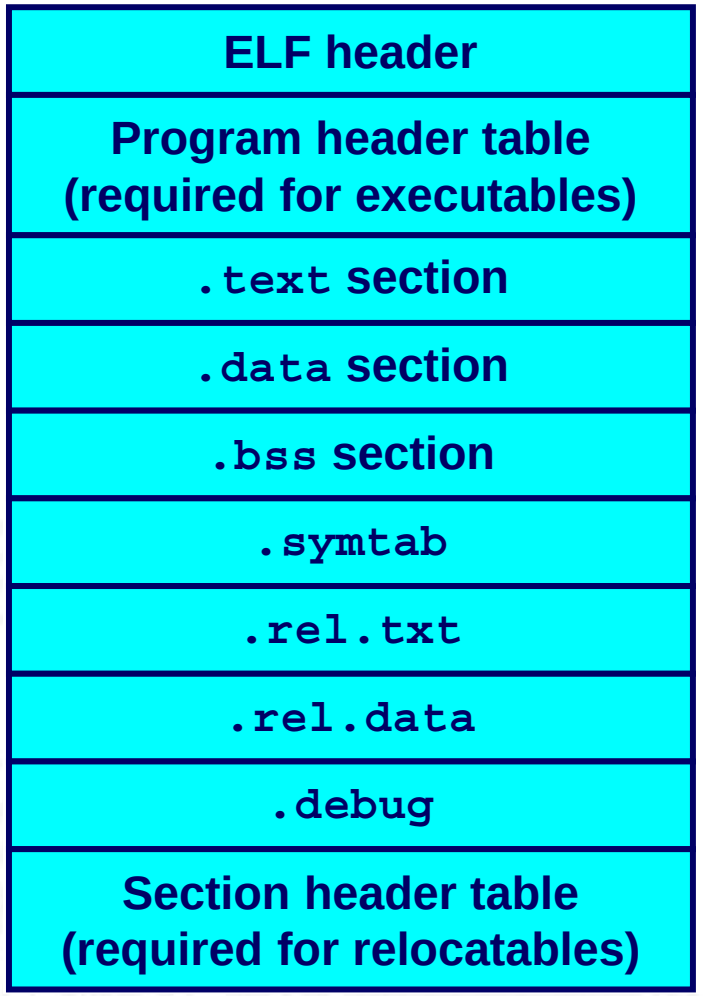
Unix: Executable and Linkable Format (ELF)

- Unix/Linux Standard für Objektdateien
- einheitliches Dateiformat für
  - relozierbare Objektdateien `.o`
  - ausführbare Objektdateien „`.exe`“
  - „*shared*“ Objektdateien `.so`
- ELF im Prinzip prozessor-/architektur-unabhängig
- aber gegebene Objektdatei ist natürlich architektur-spezifisch
  - enthält Maschinenbefehle für Zielarchitektur
  - Infos sind im Header codiert

- Microsoft nutzt COFF/PE („portable executable“) `.exe` `.dll`
- Java Class-Format `.class`

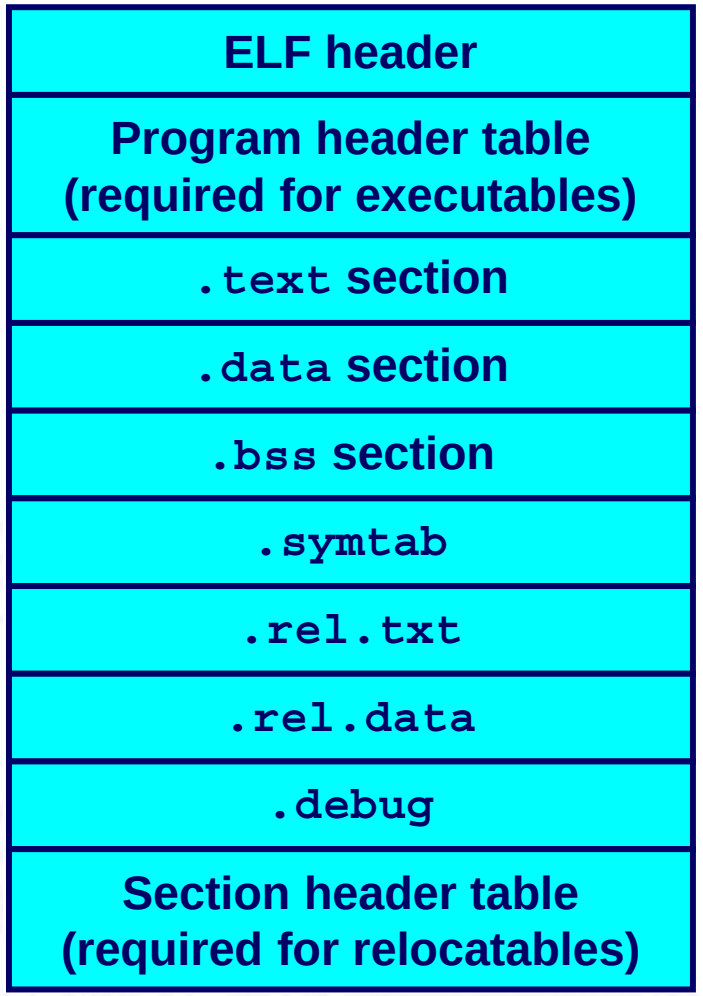
ELF Object File Format

- ELF header
  - magic number, Typ (.o, .so, .exe), Maschine, Byte-Order, usw.
- Program Header Tabelle
- `.text` Programmcode
- `.data` Statische Variablen
  - initiale Werte
- `.bss` Daten
  - unitialisierte statische Daten
  - „block started by symbol“
  - „better save space“



ELF Object File Format (cont.)

- `.symtab` Symboltabelle
  - Namen aller Funktionen und statischen
  - Variablen, Sektionsnamen und Offets
- `.rel.text` Relocation-Infos
  - alle Maschinenbefehle, die beim Linken angepasst werden müssen
  - Adressen aller (Sprung-) Befehle, die beim Linken angepasst werden müssen
- `.rel.data` Relocation-Infos
  - Adressen aller Pointer, die beim Linken angepasst werden müssen
- `.debug`
  - Hilfsinformationen fürs Debugging



Beispiel-Quellcode

m.c

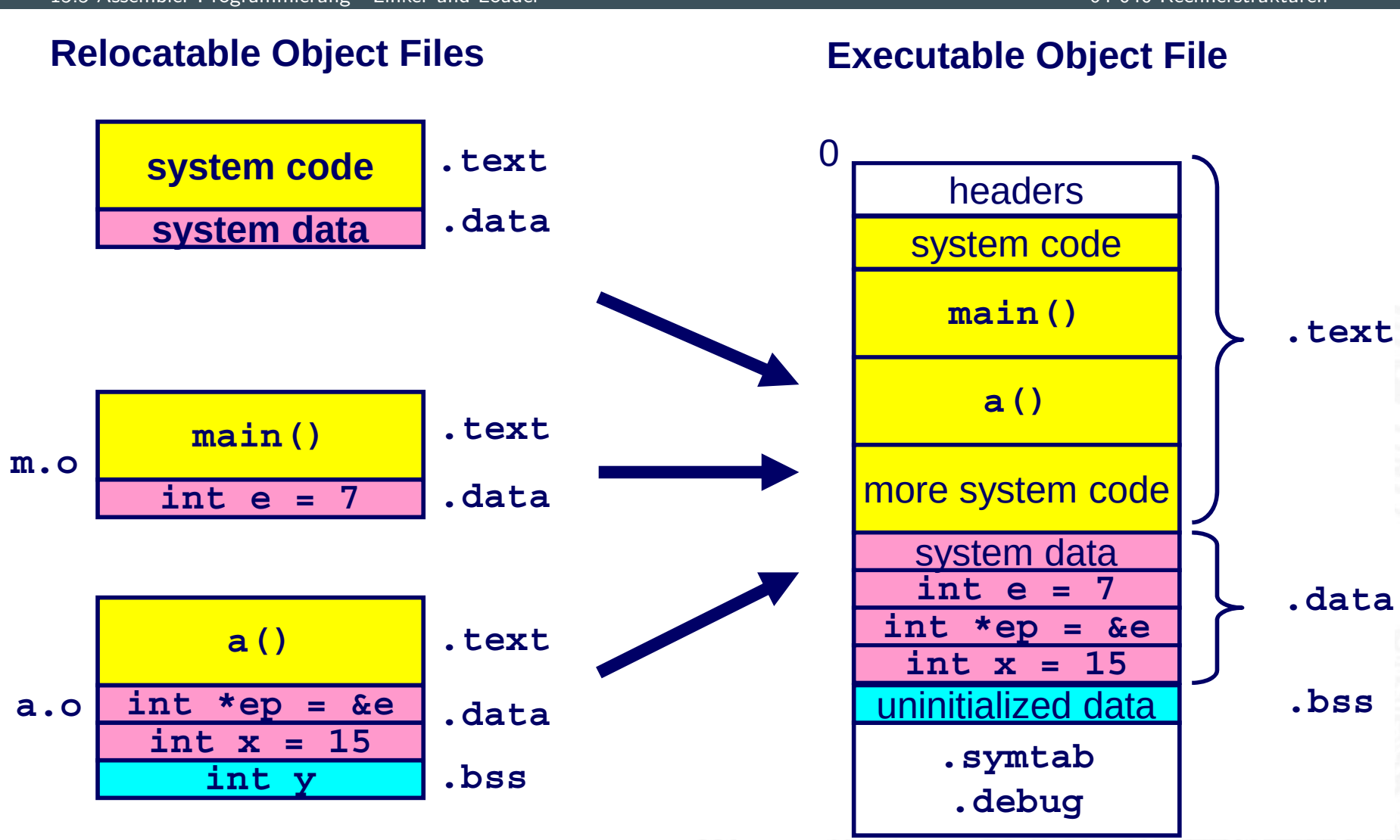
```
int e=7;

int main() {
  int r = a();
  exit(0);
}
```

a.c

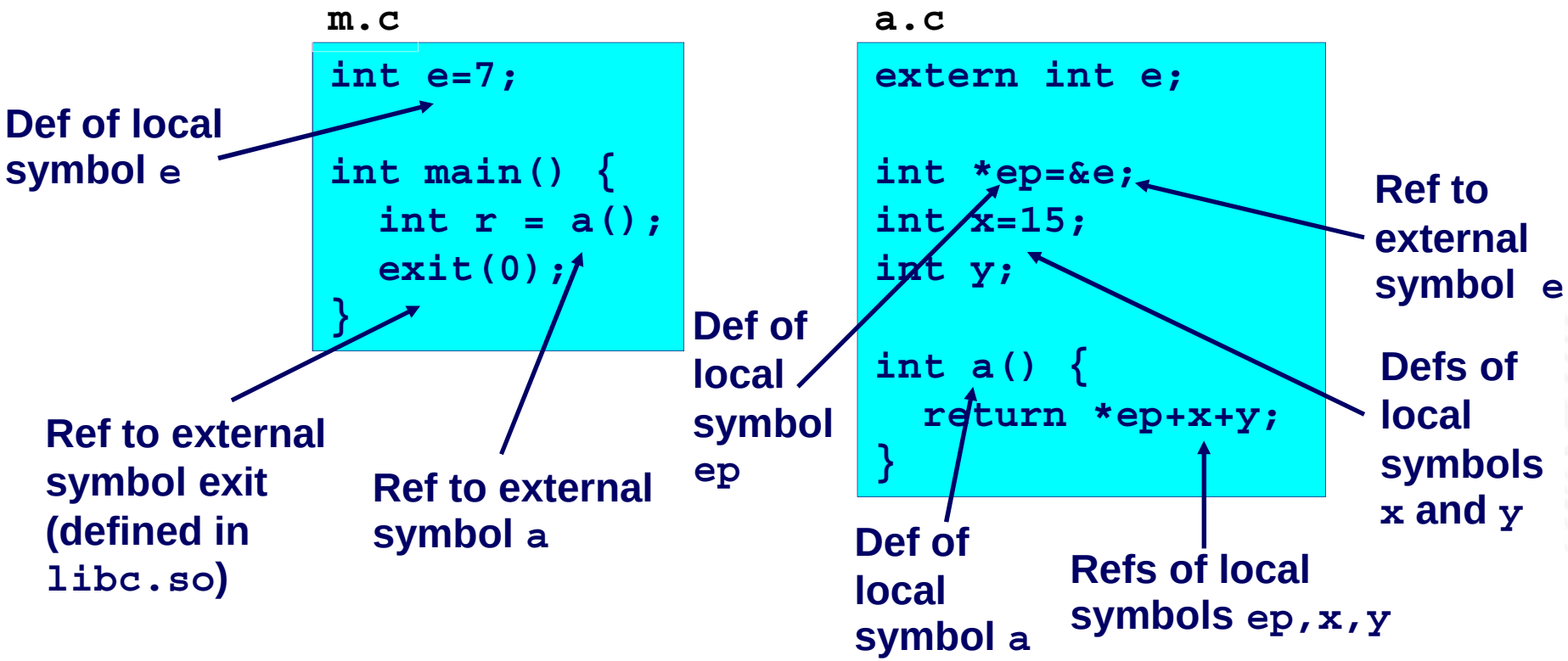
```
extern int e;

int *ep=&e;
int x=15;
int y;

int a() {
  return *ep+x+y;
}
```

- zwei Funktionen: `main()`, `a()`
- zusätzlicher System-Code, Initialisierung und `exit()`
- vier globale Variablen: `e`, `*ep`, `x`, `y`

Erzeugte ELF-Datei



Zuordnung der externen Referenzen



- Beispiel: `int e=7;` definiert und initialisiert Symbol e
  `int *ep=&e;` definiert Symbol ep und initialisiert mit der Adresse von e

m.o Relocation-Infos

m.c

```
int e=7;

int main() {
  int r = a();
  exit(0);
}
```

```
Disassembly of section .text:

00000000 <main>: 00000000 <main>:
   0:   55                   pushl  %ebp
   1:   89 e5                movl   %esp,%ebp
   3:   e8 fc ff ff ff       call   4 <main+0x4>
                             4: R_386_PC32   a
   8:   6a 00                pushl  $0x0
   a:   e8 fc ff ff ff       call   b <main+0xb>
                             b: R_386_PC32   exit
   f:   90                   nop
```

```
Disassembly of section .data:

00000000 <e>:
   0:   07 00 00 00
```

a.o Relocation-Infos für .text

a.c

```
extern int e;

int *ep=&e;
int x=15;
int y;

int a() {
  return *ep+x+y;
}
```

```
Disassembly of section .text:

00000000 <a>:
   0:   55                      pushl  %ebp
   1:   8b 15 00 00 00          movl   0x0,%edx
   6:   00
                                3: R_386_32     ep
   7:   a1 00 00 00 00          movl   0x0,%eax
                                8: R_386_32     x
   c:   89 e5                   movl   %esp,%ebp
   e:   03 02                   addl   (%edx),%eax
  10:   89 ec                   movl   %ebp,%esp
  12:   03 05 00 00 00          addl   0x0,%eax
  17:   00
                                14: R_386_32    y
  18:   5d                      popl   %ebp
  19:   c3                      ret
```

a.o Relocation-Infos für .data

a.c

```
extern int e;

int *ep=&e;
int x=15;
int y;

int a() {
  return *ep+x+y;
}
```

```
Disassembly of section .data:

00000000 <ep>:
   0:   00 00 00 00
                        0: R_386_32     e
 00000004 <x>:
   4:   0f 00 00 00
```

Erzeugtes ausführbares Programm .text

```
08048530 <main>:
 8048530:       55                   pushl  %ebp
 8048531:       89 e5                movl   %esp,%ebp
 8048533:       e8 08 00 00 00       call   8048540 <a>
 8048538:       6a 00                pushl  $0x0
 804853a:       e8 35 ff ff ff       call   8048474 <_init+0x94>
 804853f:       90                   nop

08048540 <a>:
 8048540:       55                   pushl  %ebp
 8048541:       8b 15 1c a0 04       movl   0x804a01c,%edx
 8048546:       08
 8048547:       a1 20 a0 04 08       movl   0x804a020,%eax
 804854c:       89 e5                movl   %esp,%ebp
 804854e:       03 02                addl   (%edx),%eax
 8048550:       89 ec                movl   %ebp,%esp
 8048552:       03 05 d0 a3 04       addl   0x804a3d0,%eax
 8048557:       08
 8048558:       5d                   popl   %ebp
 8048559:       c3                   ret
```

Erzeugtes ausführbares Programm .data

m.c

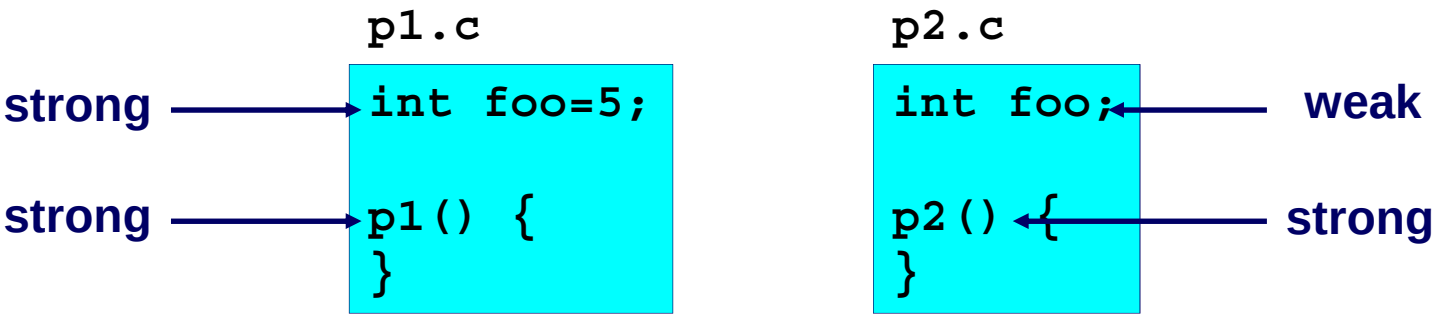
```
int e=7;

int main() {
  int r = a();
  exit(0);
}
```

a.c

```
extern int e;

int *ep=&e;
int x=15;
int y;

int a() {
  return *ep+x+y;
}
```

```
Disassembly of section .data:

0804a018 <e>:
 804a018:       07 00 00 00

0804a01c <ep>:
 804a01c:       18 a0 04 08

0804a020 <x>:
 804a020:       0f 00 00 00
```

Starke und schwache Symbole



- strong: alle Prozeduren und initialisierte globale Daten
- weak: nicht-initialisierte globale Daten

1. jedes starke Symbol darf nur einmal auftreten
2. ein schwaches Symbol wird einem starken Symbol zugewiesen
3. der Linker kann sich eines von mehreren Schwachen aussuchen

Linker-Quiz: Separate Quelldateien (C)


```
int x;
p1() {}
```

```
p1() {}
```

Link time error: two strong symbols (`p1`)

```
int x;
p1() {}
```

```
int x;
p2() {}
```

References to `x` will refer to the same uninitialized int. Is this what you really want?

```
int x;
int y;
p1() {}
```

```
double x;
p2() {}
```

Writes to `x` in `p2` might overwrite `y`! Evil!

```
int x=7;
int y=5;
p1() {}
```

```
double x;
p2() {}
```

Writes to `x` in `p2` will overwrite `y`! Nasty!

```
int x=7;
p1() {}
```

```
int x;
p2() {}
```

References to `x` will refer to the same initialized variable.

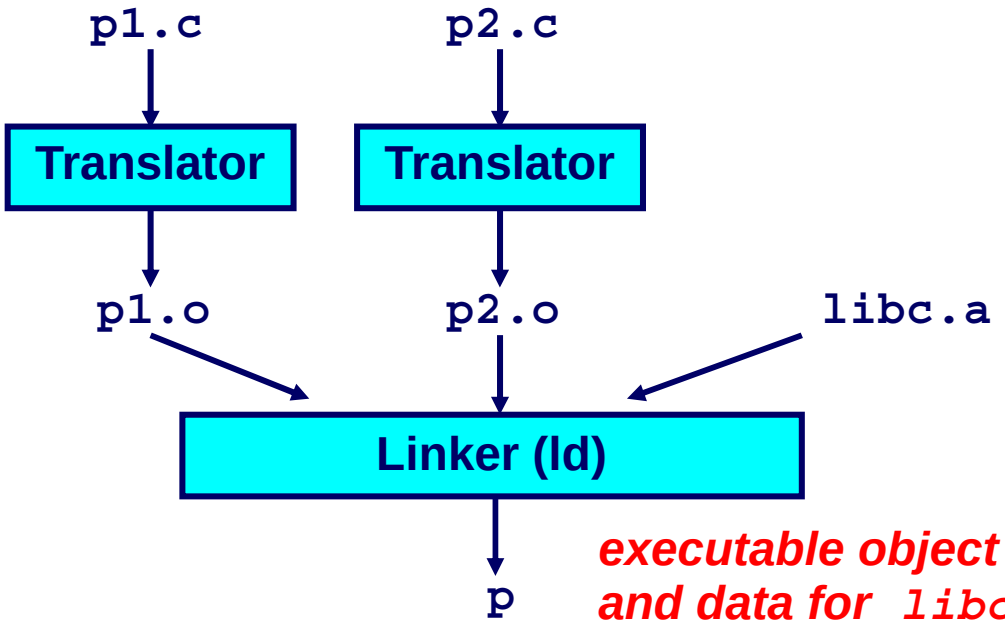
Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.

Funktionsbibliotheken

- Zugriff auf häufig benötigte Funktionen?
  - Math, Strings, I/O, Threads, Speicherverwaltung, usw.
  - alle Funktionen in einer Quelldatei ist keine Lösung
  - jede Funktion in separater Quelldatei ist sehr mühsam

- statische Funktionsbibliotheken (.a Archiv-Dateien)
  - Sammlung von compilierten Funktionen mit Index
  - Linker sucht (strong) Symbole im Index
  - gefundene Funktionen und Daten werden ins Programm eingebunden

Funktionsbibliotheken (cont.)



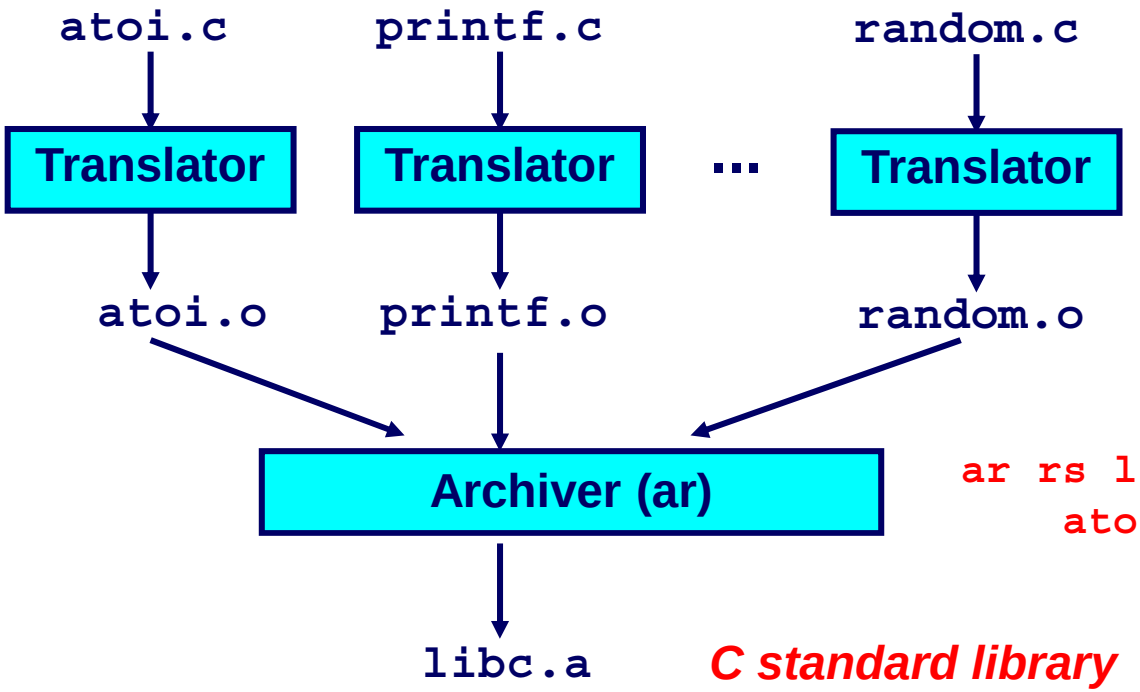
static library (archive) of relocatable object files concatenated into one file.

executable object file (only contains code and data for `libc` functions that are called from `p1.c` and `p2.c`)

Ausführbares Programm gebaut aus

- relozierbaren Modulen (`.o`), compiliert aus den Quelltexten (`.c`)
- vordefinierten Funktionsbibliotheken (`.a`)
- nur die verwendeten Funktionen landen im Programm

Statische Funktionsbibliotheken zusammenbauen




```
ar rs libc.a \
    atoi.o printf.o … random.o
```

- alle Funktionen der Bibliothek einzeln compilieren
- Archiver (`ar`) erzeugt den benötigten Index
- erzeugte ELF Datei (`.a`) mit Objektcode für alle Funktionen
- inkrementelles Update möglich (einzelne `.c` nach `.o` compilieren)

Wichtige Bibliotheken

- `libc.a`: die C „Standard-Bibliothek“
  - 900 Funktionen, ca. 8 MByte
  - I/O, Speicherverwaltung, Strings, Datum und Zeit, Zufallszahlen, Integer-Arithmetik, Signale
- `libm.a`: die C „Mathematik-Bibliothek“
  - 226 Funktionen, ca 1 MByte
  - Gleitkommafunktionen (sin, cos, tan, log, exp, sqrt, ...)
- Funktionen anzeigen
  - `ar -t /usr/lib/libm.a | sort` (32-bit)
  - `ar -t /usr/lib64/libm.a | sort` (64-bit)

    `ar -t /usr/lib/x86_64-linux-gnu/libm.a | sort`
- Java/Python/usw. benutzen eigene Bibliotheken, die wiederum auf `libc`/`libm` aufbauen

Funktionsbibliotheken verwenden

- Linker bekommt Liste der `.o` und `.a` Dateien vom Compiler
- alle Dateien werden nach fehlenden Referenzen durchsucht
- gefundene Referenzen werden sofort gelinkt („reloziert“)
- jede fehlende Referenz führt zum Abbruch

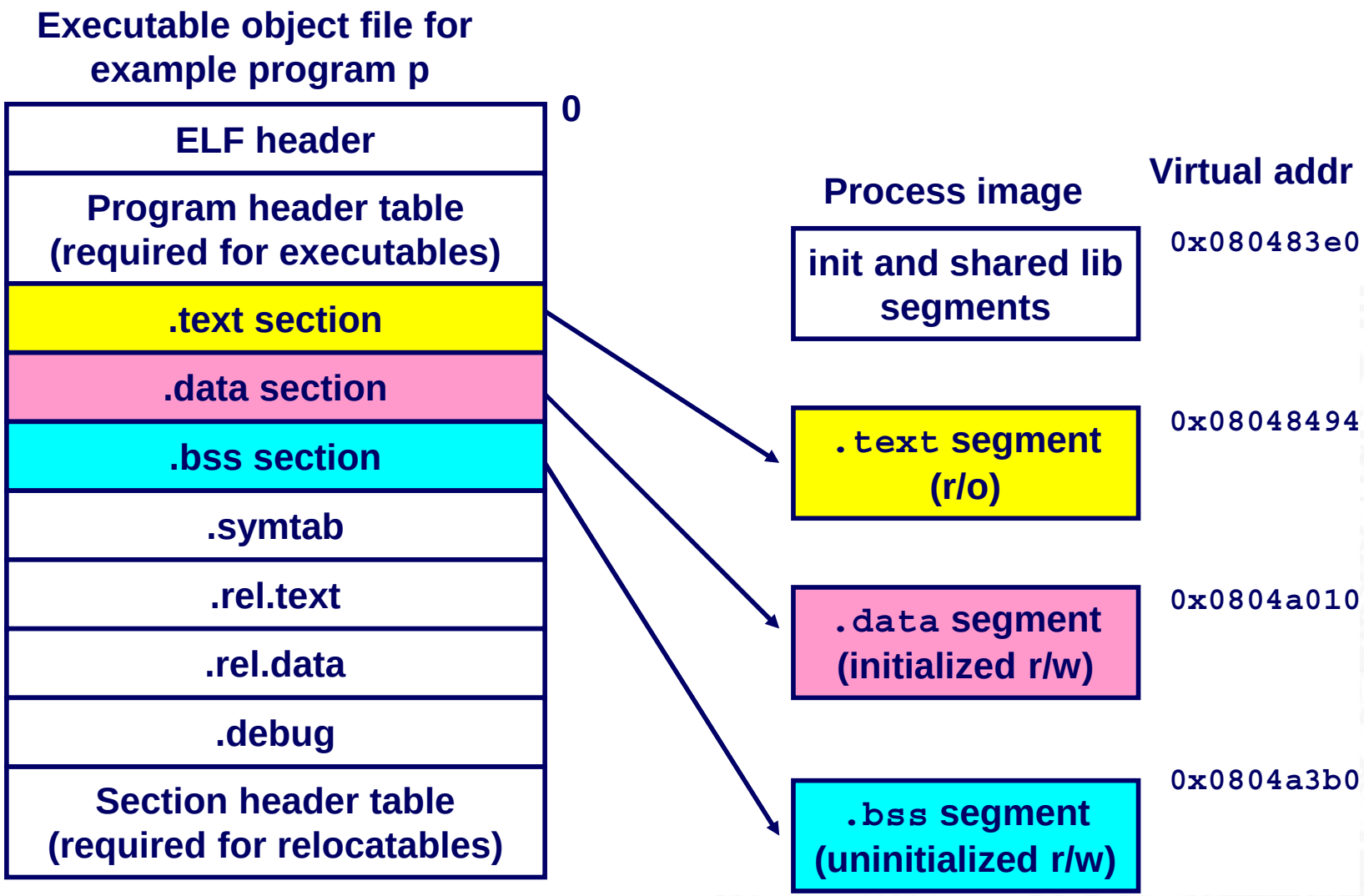
$\Rightarrow$ Reihenfolge der Module/Bibliotheken ist wichtig

$\Rightarrow$ Bibliotheken gehören ans Ende der Kommandozeile

- Unix-Konvention
  - Bibliotheken heissen `libXYZ.a`
  - Linker-Kommandozeile ohne „`lib`“, sondern nur `-lXYZ`
  - Suchverzeichnisse mit `-L <dir>` Option angeben

  - `gcc a.c b.c c.o d.o -L . -lbluetooth -lpthread -lm -lc`

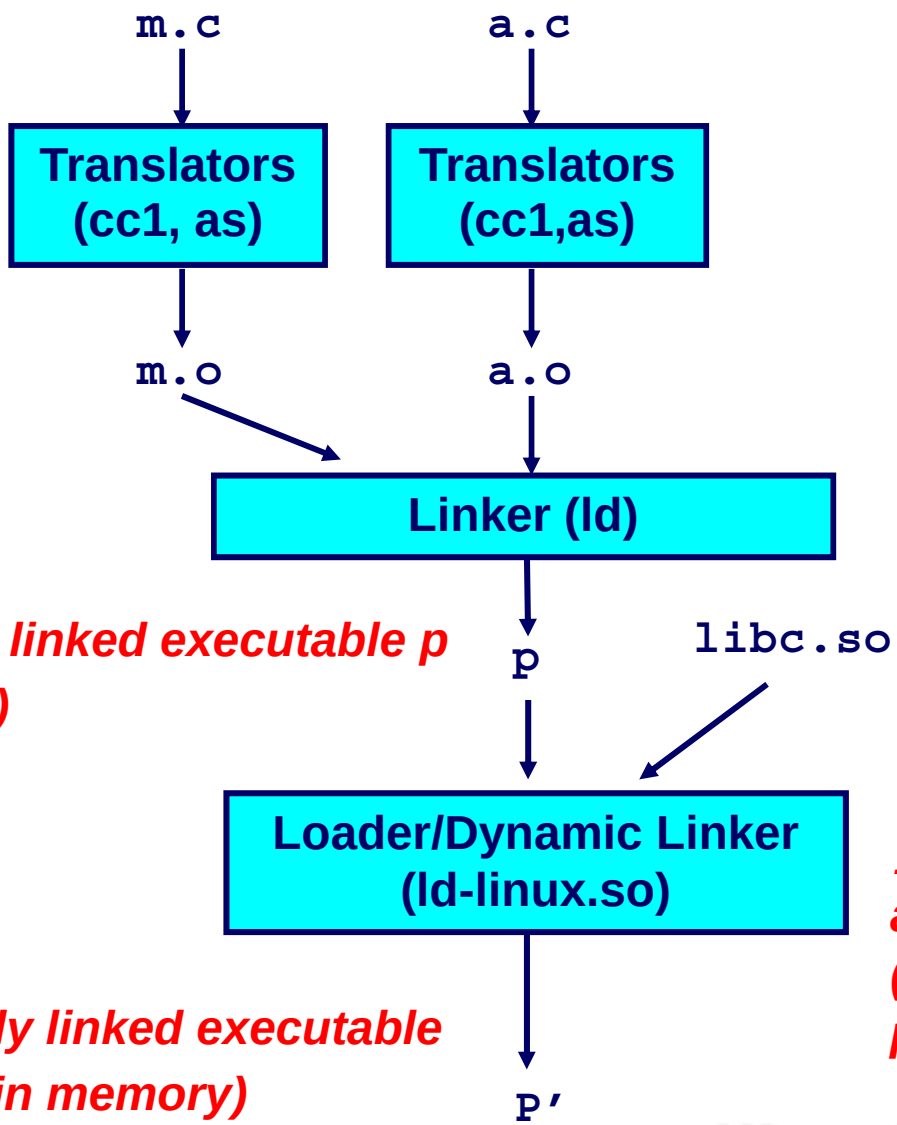
Loader: ELF-Module/Programme laden und ausführen



Dynamische Bibliotheken „*Shared Libraries*“

- Programm wird zur Objektdatei compiliert
- Bibliotheken werden erst beim Laden dazugelinkt
- die Bibliotheken können von mehreren Prozessen gleichzeitig benutzt werden, liegen aber (maximal) einmal im Speicher
- signifikant effizienter als separat statische gelinkte Programme
- Symbole werden entweder sofort (wie beim statischen Binden) oder „lazy“ referenziert (erst beim ersten Aufruf)
- Versionierung: unter Unix/Linux ist es möglich, mehrere Versionen einer Bibliothek zu verwenden, `libopencv_core.so.2.4.8`

Linker und Loader – Shared Libraries



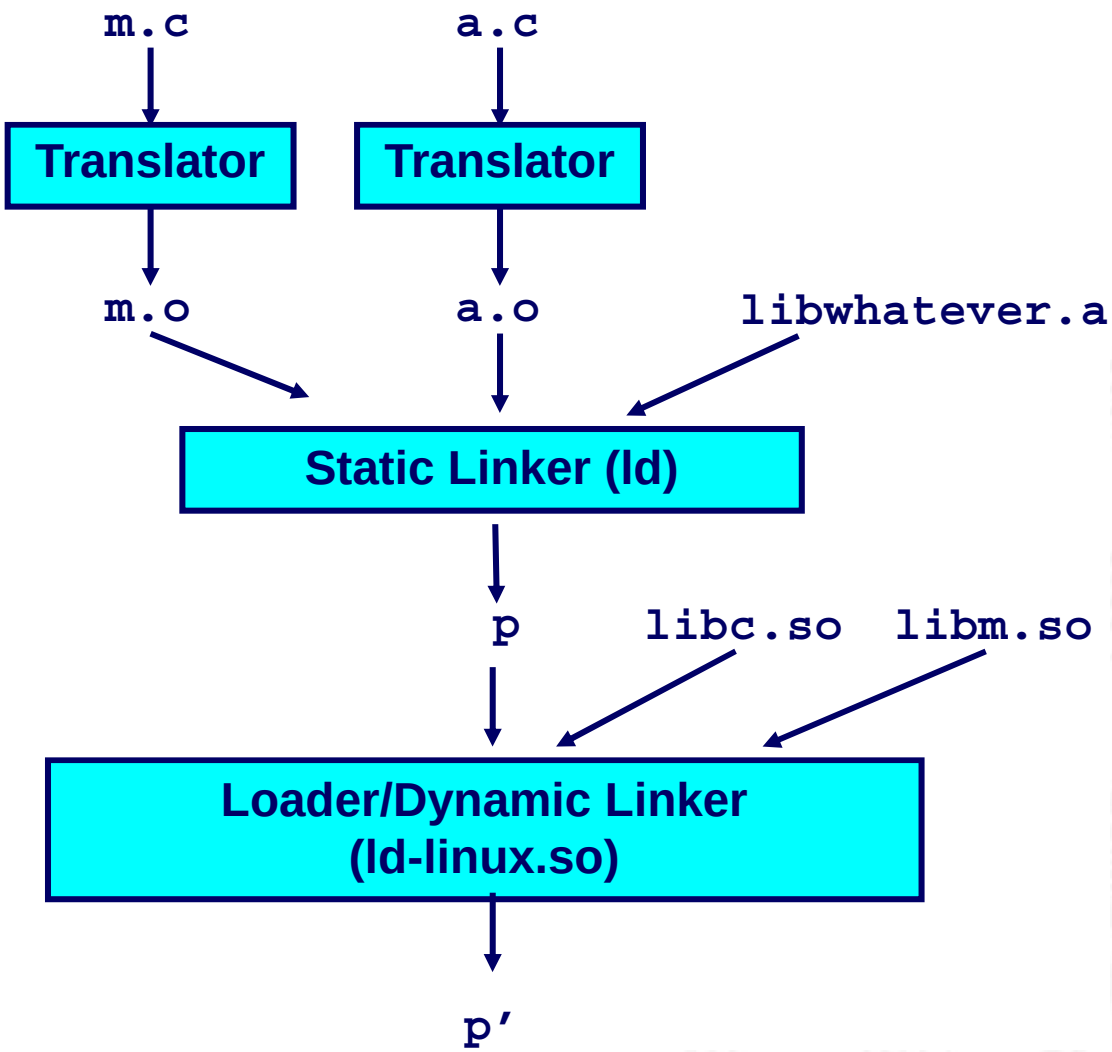
Partially linked executable p (on disk)

Shared library of dynamically relocatable object files

`libc.so` functions called by `m.c` and `a.c` are loaded, linked, and (potentially) shared among processes.

Fully linked executable p' (in memory)

Linker und Loader – Gesamtsystem



ELF: Speicheraufteilung in Regionen

- Header: Meta-Informationen
- Stack: Funktionsaufrufe
- Heap: dynamische angeforderte Daten
- statische (globale) Daten
- Code-Bereiche
- Debug- und Relocation-Infos

- bisher noch nicht erklärt: wie funktioniert die dynamische Speicherverwaltung im Heap?

Dynamic Memory Allocation

- nicht alle Daten können statisch alloziert werden
  - Speicher ist begrenzt
  - viele Daten/Arrays werden nur zeitweise benötigt
  - viele Algorithmen basieren auf dynamischen Bäumen/Graphen
  - usw.

- Datenstrukturen dynamisch anlegen
  - erst wenn die Daten benötigt werden
  - Speicher nach Benutzung wieder freigeben
  - Assembler, C/C++ benutzen die `malloc`-Bibliotheksfunktionen
  - Ursache für viele Programmierfehler

  - moderne Sprachen (Java, C# usw.) bieten automatische Heap-Verwaltung mit einem „garbage-collector“
  - bequem, aber oft auch langsamer, weniger Kontrolle

Bryant: „Harsh Reality: Memory Matters“

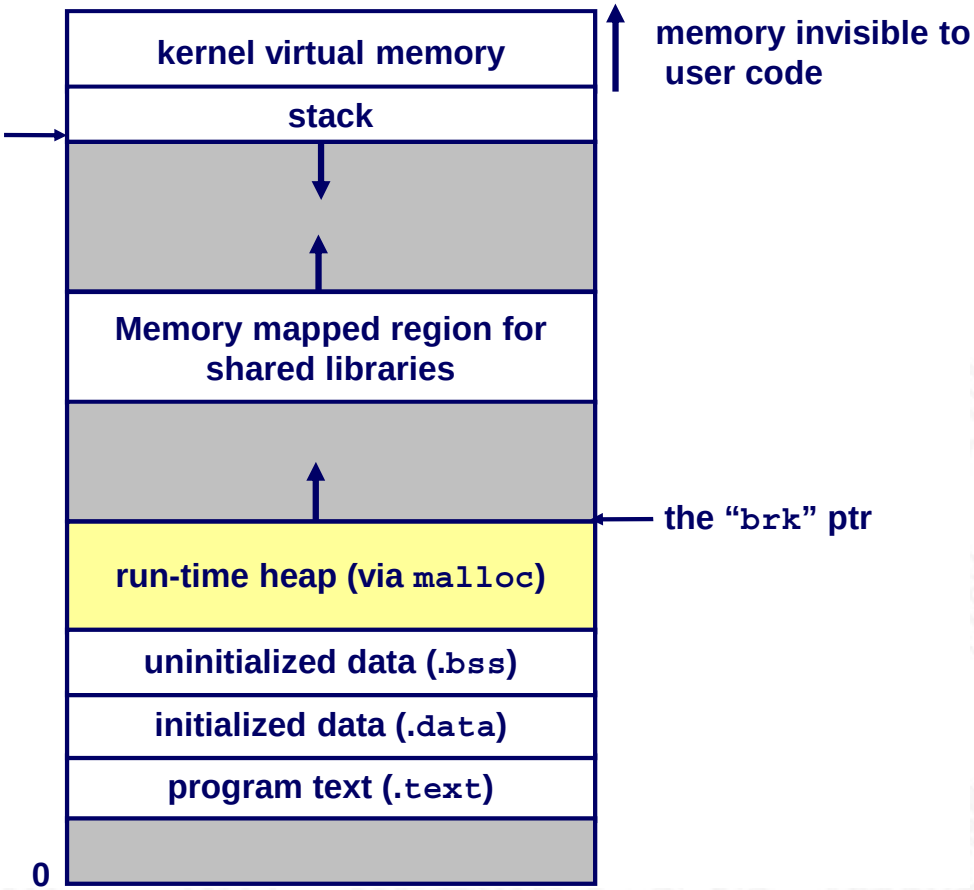
- viele Applikationen sind durch den verfügbaren Speicher begrenzt, z.B. komplexe Graph-Algorithmen
- Programmierfehler im Umgang mit dynamisch angefordertem Speicher sind häufig und schwer zu beseitigen
  - Effekt wird häufig erst spät und weit entfernt bemerkt
  - siehe wöchentliche Linux/Windows/Application Updates
- Performance eines Programms hängt entscheidend von effektivem Umgang mit dem Speicher ab
  - Cache und Virtual Memory empfindlich gegen falsche Datenstrukturen und Zugriffsmuster
  - effiziente Programmierung kann Wunder wirken

Linux: Speicherbereiche für ein Programm

- Kernel bei höchsten Adressen
- Stack wächst nach unten
- Shared-Bibliotheken mittig

Allocators request additional heap memory from the operating system using the `sbrk` function.

- Heap (dynamische Daten)
- globale statische Daten
- Programmcode
- Startup-Code ab Adresse 0



dynamischer Speicher: Funktionen

- `void* malloc( size_t size )`
  - liefert Pointer auf Speicherbereich mit mindestens `size` Bytes, ausgerichtet an 8-Byte Adressen
  - Aufruf mit `size == 0` liefert `NULL`
  - liefert `NULL`, wenn nicht erfolgreich

- `void free( void *p )`
  - gibt den Speicherbereich `*p` ans Betriebssystem zurück
  - Pointer `p` von vorherigem Aufruf von `malloc` oder `realloc`

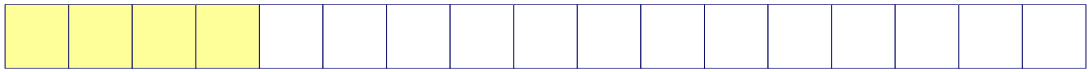
- `void* realloc( void *p, size_t size)`
  - ändert die Größe des Speicherbereichs `*p`
  - wenn erfolgreich, bleibt der Inhalt des Speicherbereichs unverändert, bis zum Minimum der alten und neuen Größe

dynamischer Speicher: Beispielcode

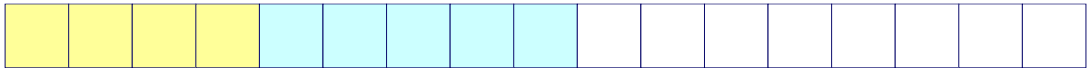
```
void foo(int n, int m) {
  int i, *p;

  /* allocate a block of n ints */
  if ((p = (int *) malloc(n * sizeof(int))) == NULL) {
    perror("malloc");
    exit(0);
  }
  for (i=0; i<n; i++)
    p[i] = i;

  /* add m bytes to end of p block */
  if ((p = (int *) realloc(p, (n+m) * sizeof(int))) == NULL) {
    perror("realloc");
    exit(0);
  }
  for (i=n; i < n+m; i++)
    p[i] = i;

  /* print new array */
  for (i=0; i<n+m; i++)
    printf("%d\n", p[i]);

  free(p); /* return p to available memory pool */
}
```

dynamischer Speicher: Memory Layout


```
p1 = malloc(4)
```

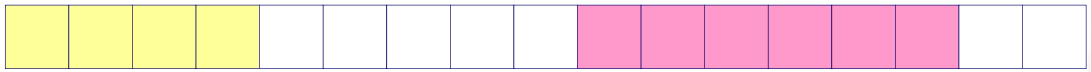


```
p2 = malloc(5)
```

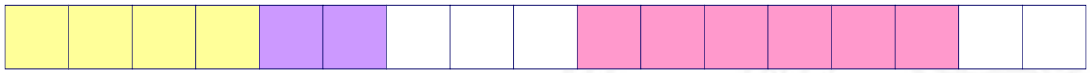


```
p3 = malloc(6)
```

```
free(p2)
```



```
p4 = malloc(2)
```



dynamischer Speicher: Anforderungen

- Programme können jederzeit `malloc` und `free` aufrufen
- die Anzahl oder Größe der angeforderten Blöcke kann nicht von der Speicherverwaltung beeinflusst werden
- Anfragen müssen sofort und möglichst schnell erfüllt werden
- dies erfordert ausreichende freie Speicherbereiche
- einmal allozierte Blöcke stehen für weitere Anfragen nicht mehr zur Verfügung, es sei denn, sie werden mit `free()` wieder freigegeben
- Vertiefung: eigenes `malloc` implementieren und testen :-)

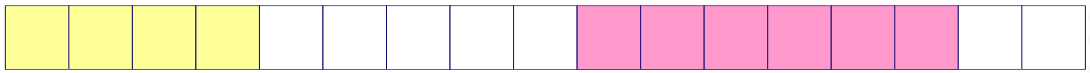
Problem: Fragmentierung


```
p1 = malloc(4)
```

```
p2 = malloc(5)
```

```
p3 = malloc(6)
```

```
free(p2)
```

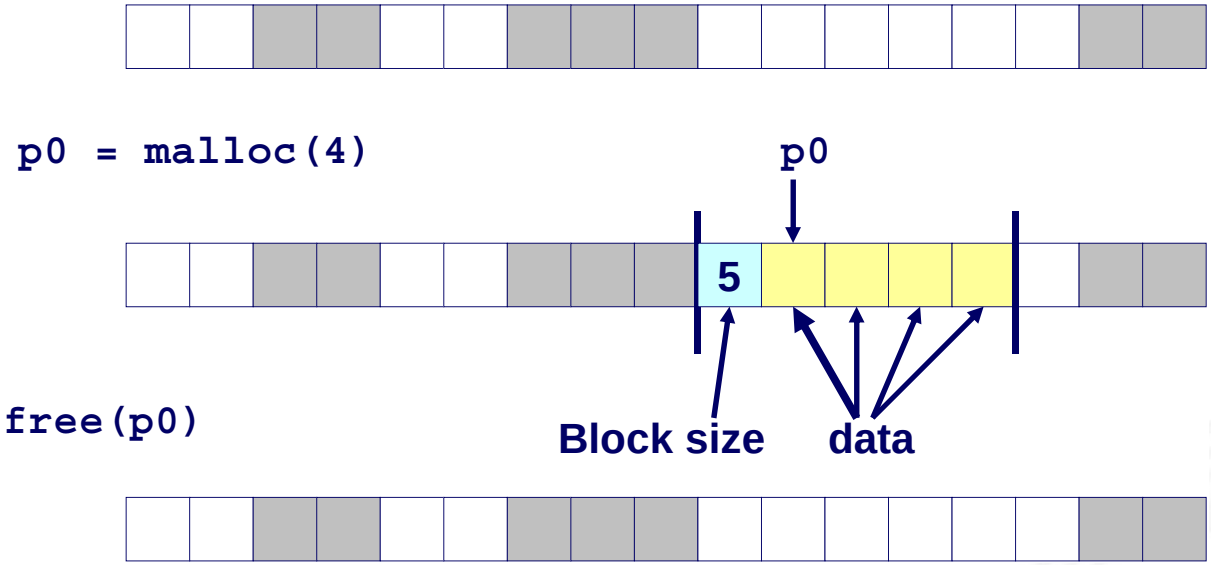


```
p4 = malloc(6)
```

oops!

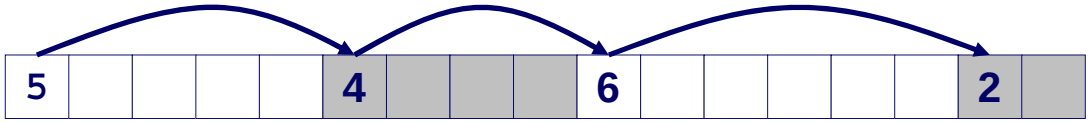
- Wir haben nur Platz für höchstens `malloc(5)`.

Idee zur Implementierung von `free()`



- Länge eines Blocks im Header gespeichert
- mindestens ein extra-Wort pro Block

Speicherblöcke verwalten



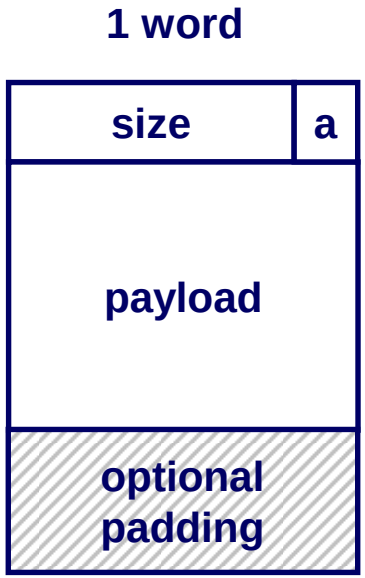
- Länge eines Blocks im Header gespeichert
  - Zusatz-/Verwaltungsdaten außerhalb des angeforderten Blocks
  - `malloc` und `free` kennen das Speicherlayout, und können Blöcke suchen bzw. zurückgeben

  - doppelte verkettete Listen (vorwärts/rückwärts) und Graphen sind effizienter als die gezeigte einfache Liste
  - Details: Bryant, O'Hallaron [BO15]

Speicherblöcke verwalten: Datenstruktur

- wie erkennt man, ob ein Block belegt ist?

Format of allocated and free blocks



a = 1: allocated block
a = 0: free block

size: block size

payload: application data (allocated blocks only)

- erfordert 1-bit extra,
- z.B. das niederwertigste Bit im `size` Feld bei wortweiser Allozierung (32-bit) und byte-weiser Adressierung

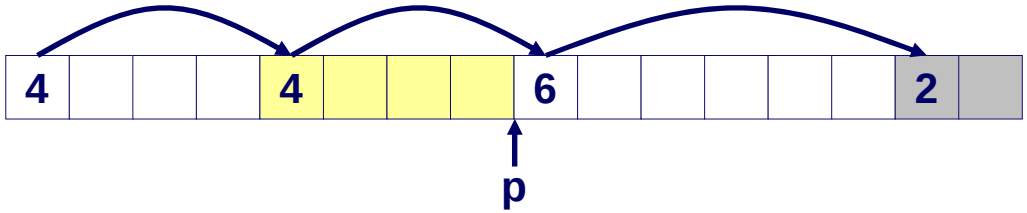
Freie Blöcke finden

- first fit: Liste vom Anfang an durchsuchen, erster passender Block wird zurückgeliefert. Linearer Zeitbedarf

```
p = start;
while ((p < end) ||     \\ not passed end
       (*p & 1) ||      \\ already allocated
       (*p <= len));    \\ too small
```

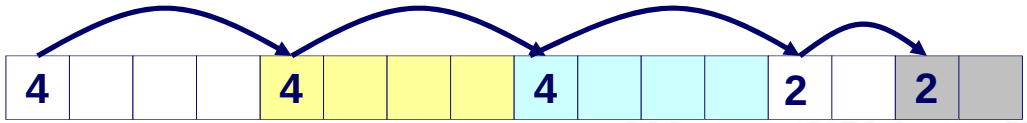
- next fit: startet die Suche vom zuletzt gefundenen Block. Fragmentierung häufig schlechter als bei first-fit.
- best fit: gesamte Liste durchsuchen, Block mit kleinstem Verschnitt zurückliefern. Weniger Fragmentierung, aber langsamer als first-fit

Freie Blöcke finden (cont.)

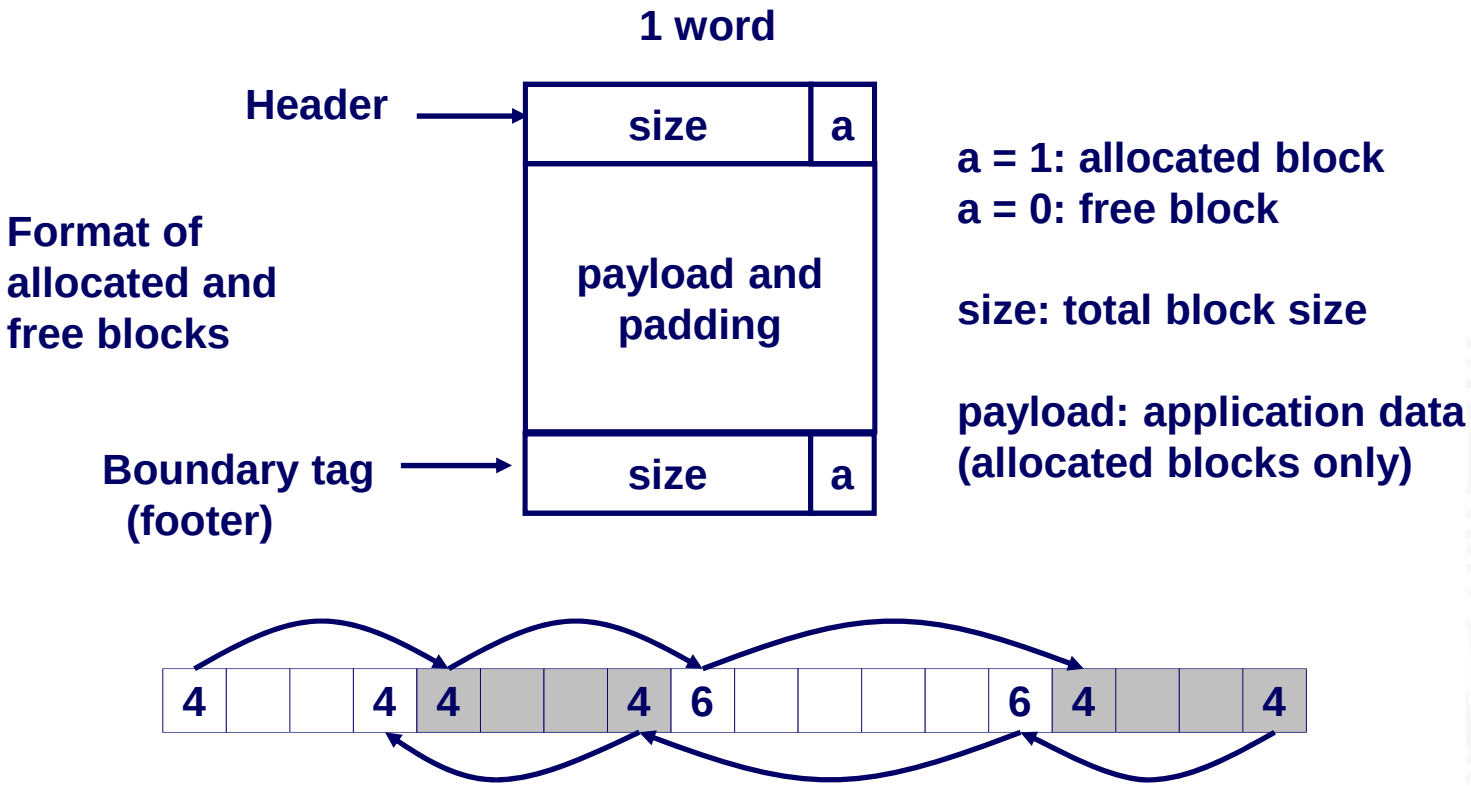



```
void addblock(ptr p, int len) {
  int newsize = ((len + 1) >> 1) << 1;  // add 1 and round up
  int oldsize = *p & -2;                // mask out low bit
  *p = newsize | 1;                     // set new length
  if (newsize < oldsize)
    *(p+newsize) = oldsize - newsize;   // set length in remaining
}                                       //   part of block
```

`addblock(p, 2)`

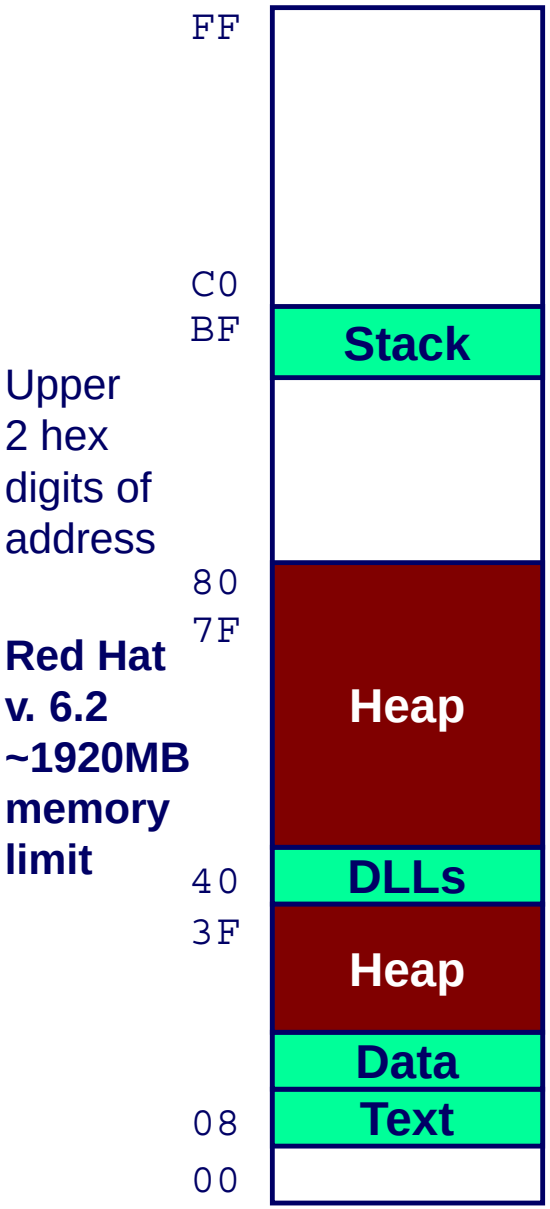


Doppelt verkettete Listen (*Bidirectional Coalescing*)



- size/allocated-Infos doppelt am Beginn und Ende des Nutzdaten-Blocks. Liste kann vorwärts und rückwärts schnell durchlaufen werden.
- schnelles Verschmelzen benachbarter freier Blöcke

Linux: Speicherlayout



Stack

- Runtime stack (8MB limit)

Heap

- Dynamically allocated storage
- When call `malloc`, `calloc`, `new`

DLLs

- Dynamically Linked Libraries
- Library routines (e.g., `printf`, `malloc`)
- Linked into object code when first executed

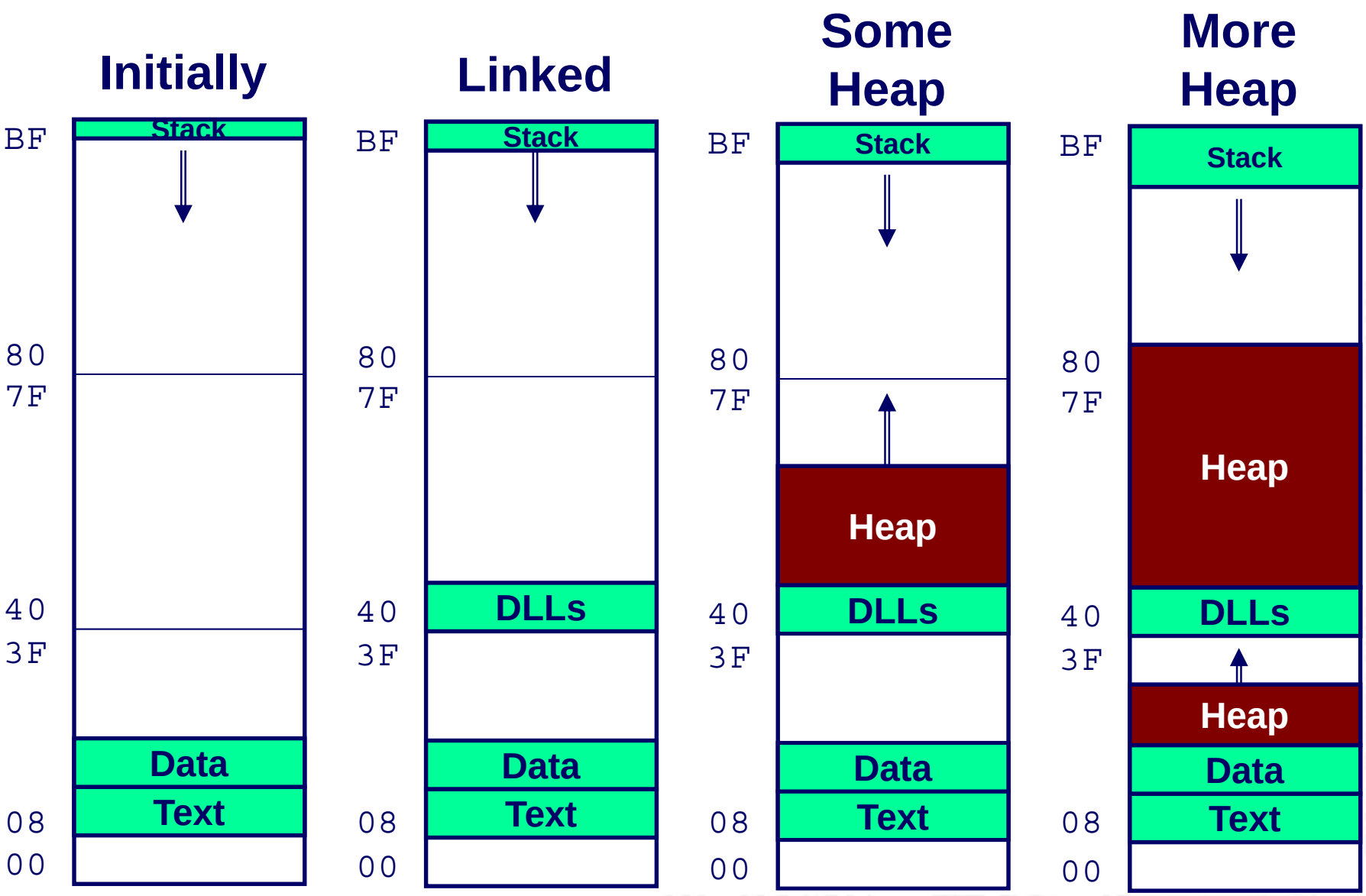
Data

- Statically allocated data
- E.g., arrays & strings declared in code

Text

- Executable machine instructions
- Read-only

Linux: Speicherverwaltung



Linux: Beispiel für Text und Stack

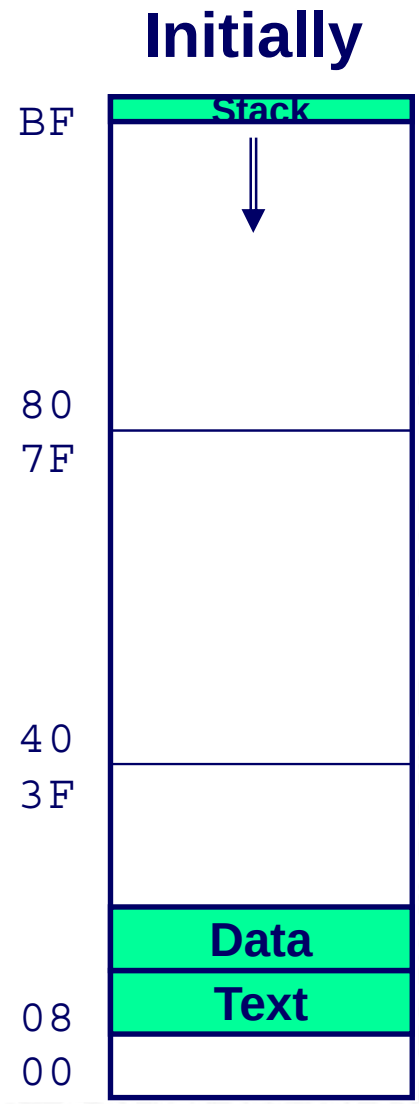

```
(gdb) break main
(gdb) run
  Breakpoint 1, 0x804856f in main ()
(gdb) print $esp
  $3 = (void *) 0xbffffc78
```

Main

- Address `0x804856f` should be read `0x08804856f`

Stack

- Address `0xbffffc78`

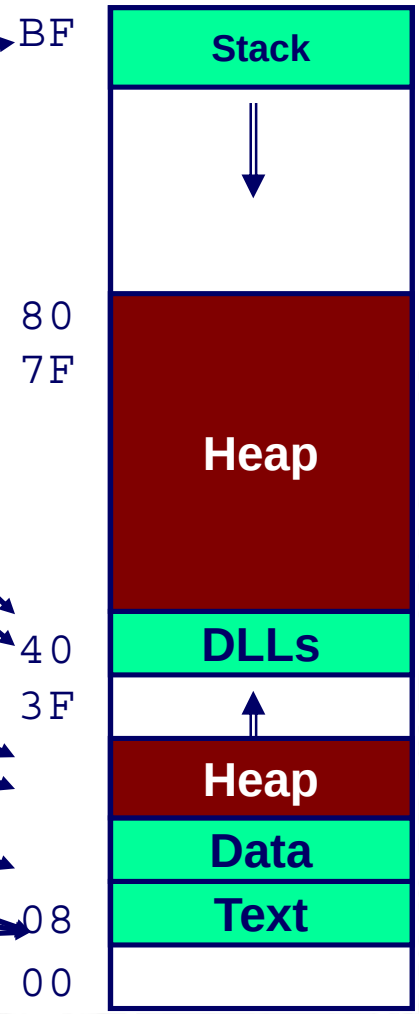


Beispiel: malloc


```
char big_array[1<<24];  /*  16 MB */
char huge_array[1<<28]; /* 256 MB */

int beyond;
char *p1, *p2, *p3, *p4;

int useless() {  return 0; }

int main()
{
 p1 = malloc(1 <<28);  /* 256 MB */
 p2 = malloc(1 << 8);  /* 256 B  */
 p3 = malloc(1 <<28);  /* 256 MB */
 p4 = malloc(1 << 8);  /* 256 B  */
 /* Some print statements ... */
}
```

Beispiel: Speicherbereiche

$esp	0xbffffc78
p3	0x500b5008
p1	0x400b4008
Final malloc	0x40006240
p4	0x1904a640
p2	0x1904a538
beyond	0x1904a524
big_array	0x1804a520
huge_array	0x0804a510
main()	0x0804856f
useless()	0x08048560
Initial malloc	0x08048454



Gruselkabinett: Memory-related Bugs

- ungültige Pointer dereferenzieren
- nicht existierende Variablen referenzieren
- nicht-initialisierten Speicher lesen
- Speicherbereiche überschreiben
- freie Blöcke referenzieren
- Blöcke mehrfach freigeben
- Blöcke nicht freigeben: Speicherlecks

- Details: Bryant, O'Hallaron [BO15]
- Java: die meisten (dieser) Fehler sind unmöglich

Ungültige Pointer dereferenzieren

- der „klassische“ `scanf`-Bug

```
scanf("%d", val);
```

- lokale Variablen „verschwinden“ nach dem Rücksprung:

```
int *foo () {
   int val;
   return &val;
}
```

- tückisch: direkt nach dem Rücksprung liegen die Daten noch auf dem Stack, werden aber von späteren Funktionsaufrufen überschrieben

Nicht initialisierten Speicher lesen

- per `malloc` allozierter Speicher ist nicht initialisiert

```
/* return y = Ax */
int *matvec(int **A, int *x) {
   int *y = malloc(N*sizeof(int));
   int i, j;

   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         y[i] += A[i][j]*x[j];
   return y;
}
```

$\Rightarrow$ `calloc` aufrufen oder Bereich explizit initialisieren

Speicherbereiche überschreiben

- versehentlich falsche Größe beim `malloc`

```
int **p;

p = malloc(N*sizeof(int));

for (i=0; i<N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```

- off-by-one Fehler

```
int **p;

p = malloc(N*sizeof(int *));

for (i=0; i<=N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```

Speicherbereiche überschreiben (cont.)

- Maximalgröße von Puffern nicht beachtet

```
char s[8];
int i;

gets(s);  /* reads "123456789" from stdin */
```

  - sehr häufiger Fehler
  - Einfallstor für Schadsoftware

- Missverständnis der Pointerarithmetik

```
int *search(int *p, int val) {

   while (*p && *p != val)
      p += sizeof(int);

   return p;
}
```

Speicherbereiche mehrfach freigeben


```
x = malloc(N*sizeof(int));
<manipulate x>
free(x);

y = malloc(M*sizeof(int));
<manipulate y>
free(x);
```

- Zugriff auf freigegebenen Speicher

```
x = malloc(N*sizeof(int));
<manipulate x>
free(x);
...
y = malloc(M*sizeof(int));
for (i=0; i<M; i++)
   y[i] = x[i]++;
```

Speicherlecks

- Speicherbereiche nicht freigeben

```
foo() {
   int *x = malloc(N*sizeof(int));
   ...
   return;
}
```

  - nach dem Rücksprung bleibt der Speicher belegt, aber es gibt keinen (gültigen) Pointer mehr

Speicherlecks (cont.)

- Speicherbereiche nur teilweise freigeben

```
struct list {
   int val;
   struct list *next;
};

foo() {
   struct list *head =
                malloc(sizeof(struct list));
   head->val = 0;
   head->next = NULL;
   <create and manipulate the rest of the list>
   ...
   free(head);
   return;
}
```

gets aus Standard C Library

- Puffer wird übergeben, aber Anzahl der gelesenen Zeichen ist nicht limitiert

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getc();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getc();
    }
    *p = '\0';
    return dest;
}
```

Verwundbarer Code

- Puffer liegt auf dem Stack, ist viel zu klein

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
int main()
{
  printf("Type a string:");
  echo();
  return 0;
}
```

Verwundbarer Code (cont.)

- Verhalten

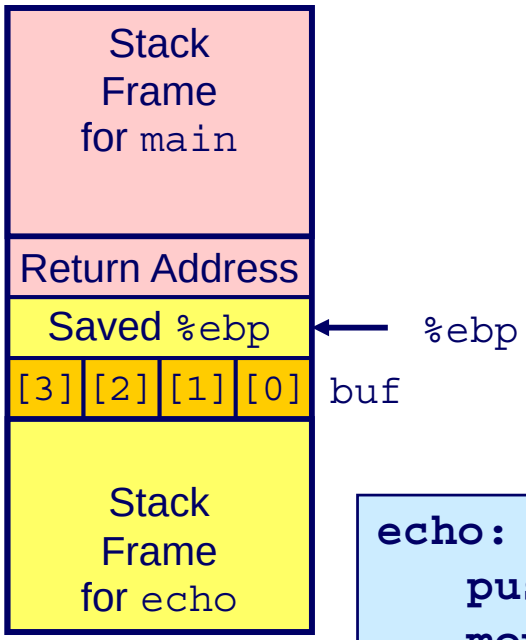
```
unix>./bufdemo
Type a string:123
123
```

```
unix>./bufdemo
Type a string:12345
Segmentation Fault
```

```
unix>./bufdemo
Type a string:12345678
Segmentation Fault
```

⇒ Array überschreibt den Stack

Verwundbarer Code: Stack

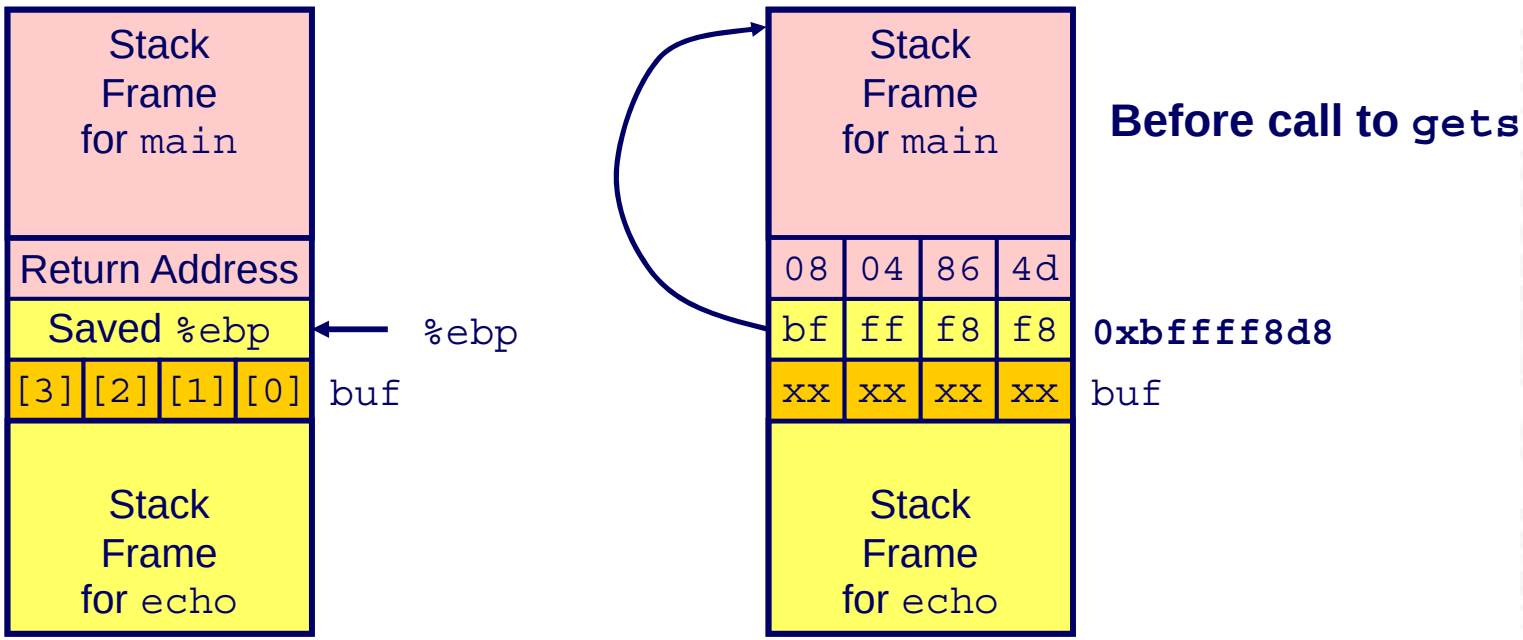



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    pushl %ebp              # Save %ebp on stack
    movl %esp,%ebp
    subl $20,%esp           # Allocate space on stack
    pushl %ebx              # Save %ebx
    addl $-12,%esp          # Allocate space on stack
    leal -4(%ebp),%ebx      # Compute buf as %ebp-4
    pushl %ebx              # Push buf on stack
    call gets               # Call gets
    . . .
```

Verwundbarer Code: Stack (cont.)

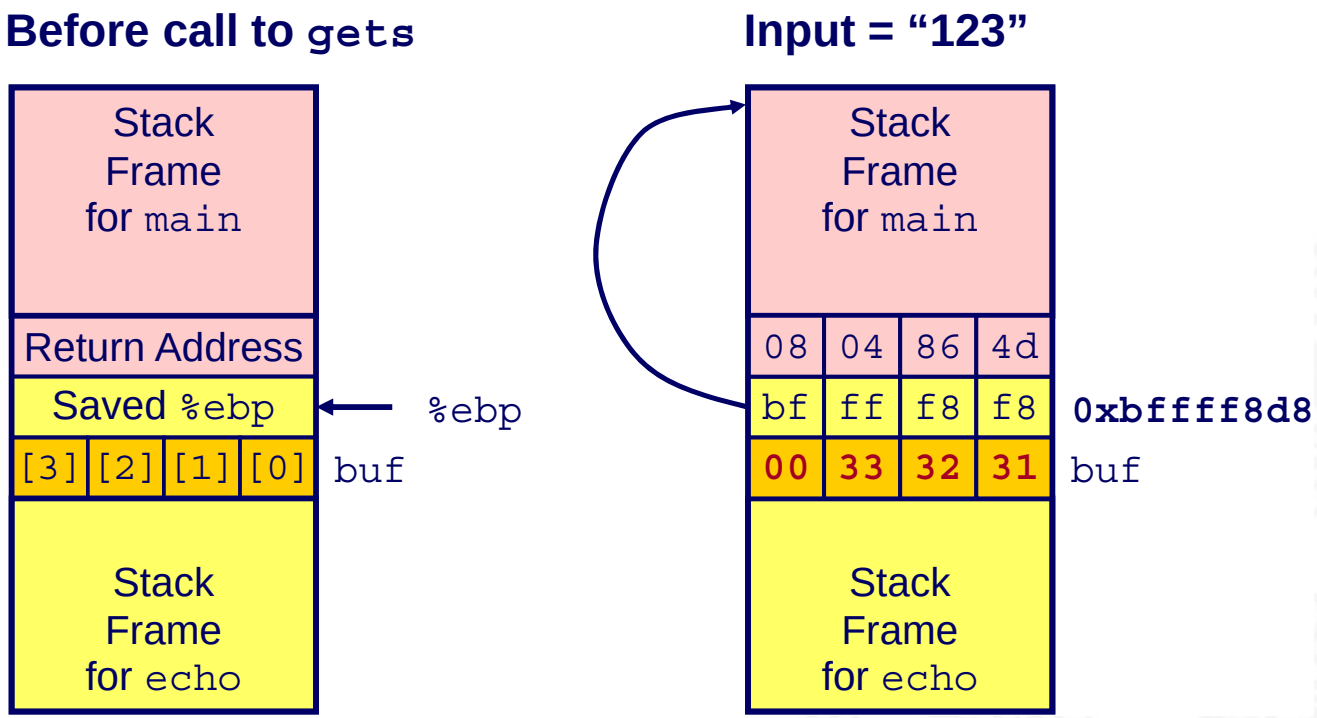

```
unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x8048583
(gdb) run
Breakpoint 1, 0x8048583 in echo ()
(gdb) print /x *(unsigned *)$ebp
$1 = 0xbffff8f8
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x804864d
```




```
8048648:  call 804857c <echo>
804864d:  mov  0xffffffe8(%ebp),%ebx # Return Point
```

Verwundbarer Code: Beispiele

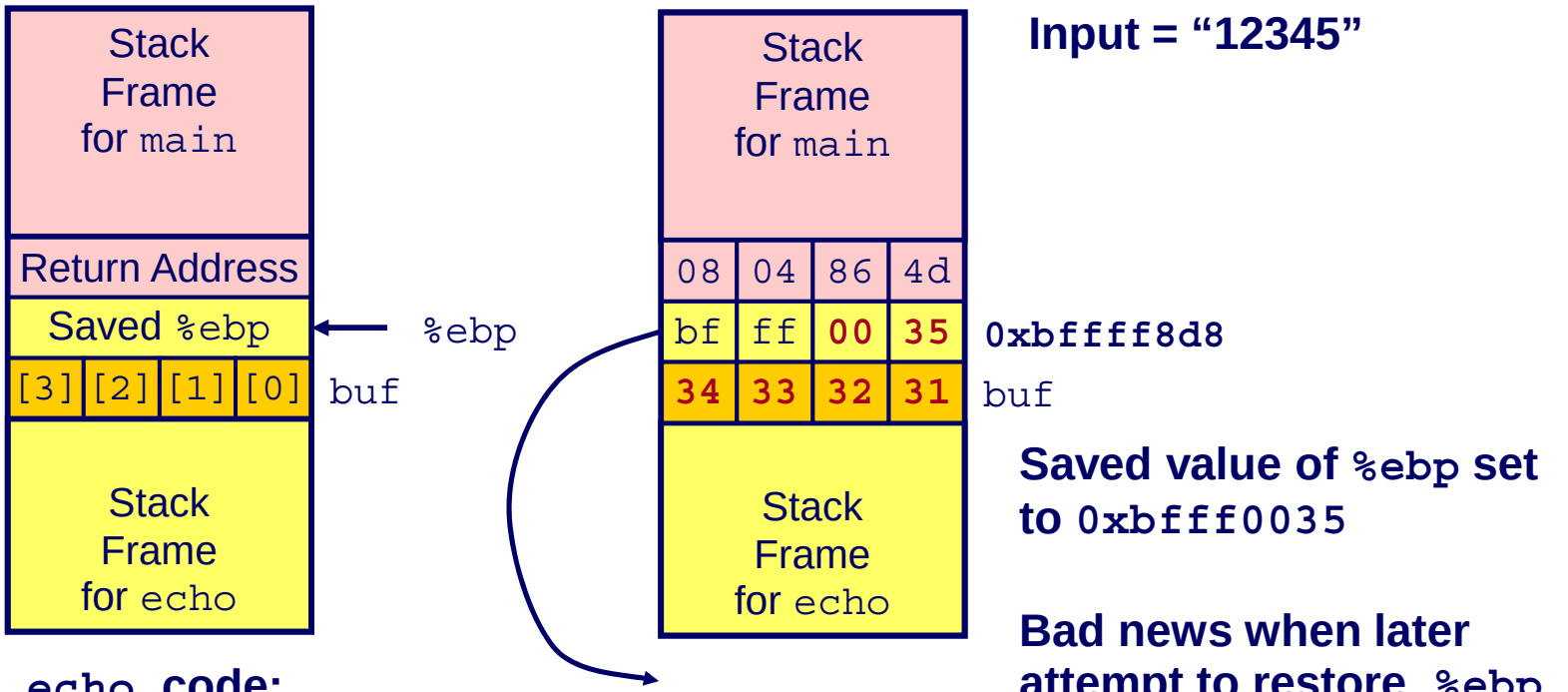
- Eingabe "123"



$\Rightarrow$ alles OK

Verwundbarer Code: Beispiele (cont.)

- Eingabe "12345"



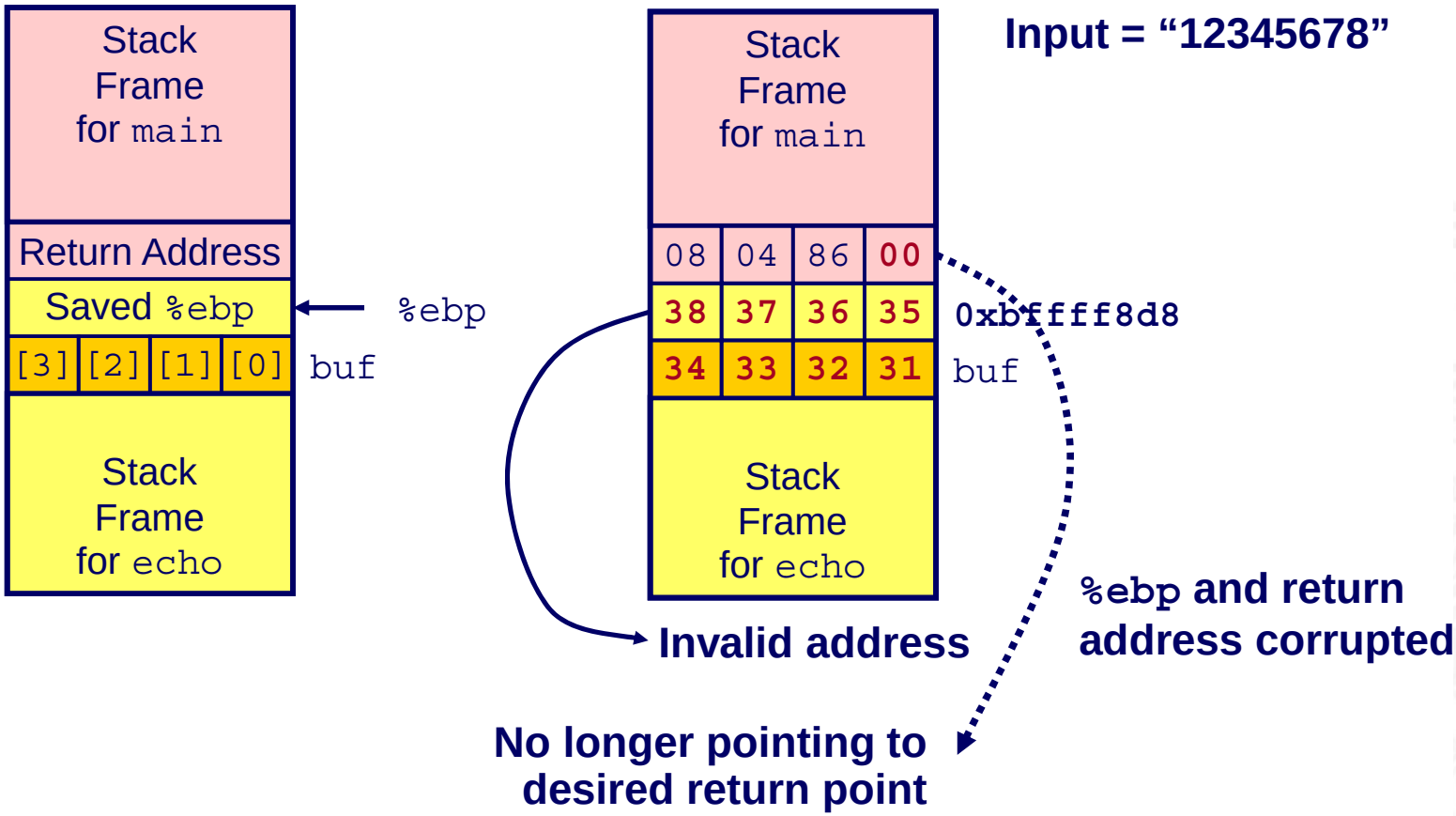
echo code:

```
8048592:  push   %ebx
8048593:  call   80483e4 <_init+0x50>  # gets
8048598:  mov    0xfffffff8(%ebp),%ebx
804859b:  mov    %ebp,%esp
804859d:  pop    %ebp     # %ebp gets set to invalid value
804859e:  ret
```

⇒ Array überschreibt den Stack

Verwundbarer Code: Beispiele (cont.)

- Eingabe "12345678"




```
8048648:  call 804857c <echo>
804864d:  mov  0xffffffe8(%ebp),%ebx # Return Point
```

⇒ Return Adresse überschrieben

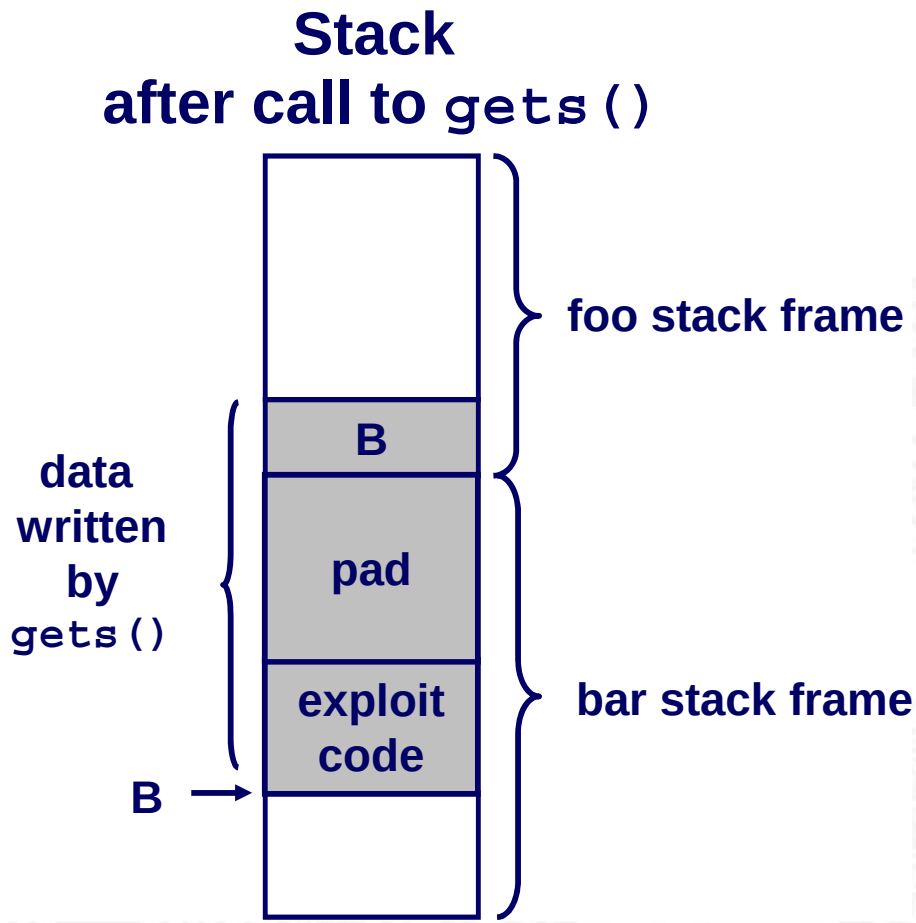
Verwundbarer Code: Beispiele (cont.)

⇒ Rücksprung in Schad-Code!

return address A

```
void foo(){
  bar();
  ...
}
```

```
void bar() {
  char buf[64];
  gets(buf);
  ...
}
```



Puffer-Überläufe: Sicherheitslücken

- Buffer-Overflow Bugs erlauben es Angreifern, beliebigen Code auf den angegriffenen Rechnern auszuführen!
- der Code wird vom Angreifer mitgeliefert

```
GET
  /default.ida?NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN....NNNN
  NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN%u9090%u6858%ucbd3%u780
  1%u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801%u9090%u909
  0%u8190%u00c3%u0003%u8b00%u531b%u53ff%u0078%u0000%u00=a
HTTP/1.0" 400 325 "-" "-"
```

$\Rightarrow$ sorgfältig programmieren

$\Rightarrow$ verwundbare Bibliotheksfunktionen vermeiden

Assemblerebene – Zusammenfassung

- Umsetzung von Programmen mit Kontrollstrukturen
- Funktionsaufrufe, Parameter, lokale Variablen

- Speicherlayout von strukturierten Daten und Arrays
- Umsetzung objektorienter Konzepte

- ELF-Dateiformat und statisches Linking
- Programmcode, Stack, Heap, statische Variablen
- Funktionsbibliotheken

- Dynamische Speicherverwaltung im Heap
- Puffer-Überläufe

Literatur

[BO15] R.E. Bryant, D.R. O'Hallaron:
Computer systems – A programmers perspective.
3rd global ed., Pearson Education Ltd., 2015.
ISBN 978–1–292–10176–7. `csapp.cs.cmu.edu`

[TA14] A.S. Tanenbaum, T. Austin: *Rechnerarchitektur – Von der digitalen Logik zum Parallelrechner*.
6. Auflage, Pearson Deutschland GmbH, 2014.
ISBN 978–3–86894–238–5

[IA64] *Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 1: Basic Architecture*.
Intel Corp.; Santa Clara, CA.
`www.intel.de/content/www/de/de/processors/architectures-software-developer-manuals.html`

Literatur (cont.)

[PH16a] D.A. Patterson, J.L. Hennessy: *Computer Organization and Design – The Hardware Software Interface: ARM Edition*. Morgan Kaufmann Publishers Inc., 2016. ISBN 978–0–12–801733–3

[PH16b] D.A. Patterson, J.L. Hennessy: *Rechnerorganisation und Rechnerentwurf – Die Hardware/Software-Schnittstelle*. 5. Auflage, Oldenbourg, 2016. ISBN 978–3–11–044605–0

[Hyd10] R. Hyde: *The Art of Assembly Language Programming*. 2nd edition, No Starch Press, 2010. ISBN 978–1–59327–207–4. `www.plantation-productions.com/Webster/www.artofasm.com`