Betriebssysteme I

Praktikum im SoSe 2020

Prof. Jens-Peter Redlich
Dorian Weber

Bei Fragen

E-Mail weber@informatik.hu-berlin.de

Büro RUD25, 3.326

Zoom virtueller Raum

Zeiten Wochentags gewöhnlich zwischen 11 und 17 Uhr online

Forum Moodle-Kurs beinhaltet ein Forum für eure Fragen

Themen

1. Schnittstellen zwischen Hard- und Software
   - Assembly, BIOS, Bootvorgang
2. Funktionsweise von Low-Level Diensten
   - Shell, Prozessverwaltung, Verzeichnisnavigation
   - Interprozesskommunikation, Netzwerkzugriff
3. Grundzüge paralleler Programmausführung
   - Exception, Generator, Koroutine, Thread, Scheduler
   - Deadlock, Race Condition, Lock, Mutex, Semaphore, Atomic
4. Speichermanagement
   - Speicheradressen, -virtualisierung, -segmentierung
   - geteilter Speicher, verteilter Speicher, Speicherverwaltung
5. Programmbibliotheken
   - Linker, statische/dynamische Bindung, C-ABI

Übungsaufgaben

- fünf Übungsaufgaben geplant
- zweiwöchige Bearbeitungsfrist, Abgabe am Abgabetag bis 23:59 Uhr möglich
- Abgabedateien in einem zip-Archiv; kommentierter und lesbarer Code
- Dokumentation, Kommentare und Bezeichner sind in deutscher oder englischer Sprache
- 50% aller Punkte müssen für die Zulassung zur Prüfung erreicht werden
- Zweiergruppen; höchstens eine Dreiergruppe
- Referenzumgebung in Form von VM-Image vorgegeben

Eigenschaften von C

- kompakte Syntax, imperativ, prozedural, maschinennah, statisch typisiert, portabel
- integrierter, makrobasierter Präprozessor
- Code wird vom Compiler in mnemonischen Maschinencode übersetzt
- C wird häufig als *portable assembler* charakterisiert
- Unterscheidung zwischen Header- und Quelldateien

  Headerdatei Deklaration globaler Bezeichner

  Quelldatei Implementation deklarierter Funktionen, Initialisierung globaler Variablen
- jedes Programm enthält genau eine `int main()`, die als Programmeinstiegspunkt fungiert

Programmbeispiel

```c
#include <stdio.h>

#if !defined(NDEBUG)
    #define TRACE(MSG) \
        fprintf(stderr, MSG)
#else
    #define TRACE(MSG)
#endif

int main(unsigned int argc, const char *argv[]) {
    TRACE("startup phase");
    printf("Hello %s!\n", argv[0]);
    TRACE("shutdown phase");
    return 0;
}
```

Schematische Headerdatei

```c
/* include guard */
#ifndef HEADER_H_INCLUDED
#define HEADER_H_INCLUDED

/* include directives */
#include <...> // standard library
#include "..." // program library

/* structure definitions */
typedef struct MyStruct {
    ...         // members
} MyStruct;

/* function declarations */
extern int  myStructInit(MyStruct *self);
extern void myStructRelease(MyStruct *self);

/* global variables */
extern int MY_STRUCT_I;
#endif
```

Schematische Quelldatei

```c
/* include directives */
#include "header.h" // our own header file
#include <...>
#include "..."

/* internal structures */
enum Enum {...};

/* internal variables */
static const float PI = 3.1415;

/* internal functions */
static enum Enum fun1(const MyStruct *self, ...);

/* function definitions */
int  myStructInit(MyStruct *self) {...}
void myStructRelease(MyStruct *self) {...}

/* variable initialization */
int MY_STRUCT_I = 42;
```

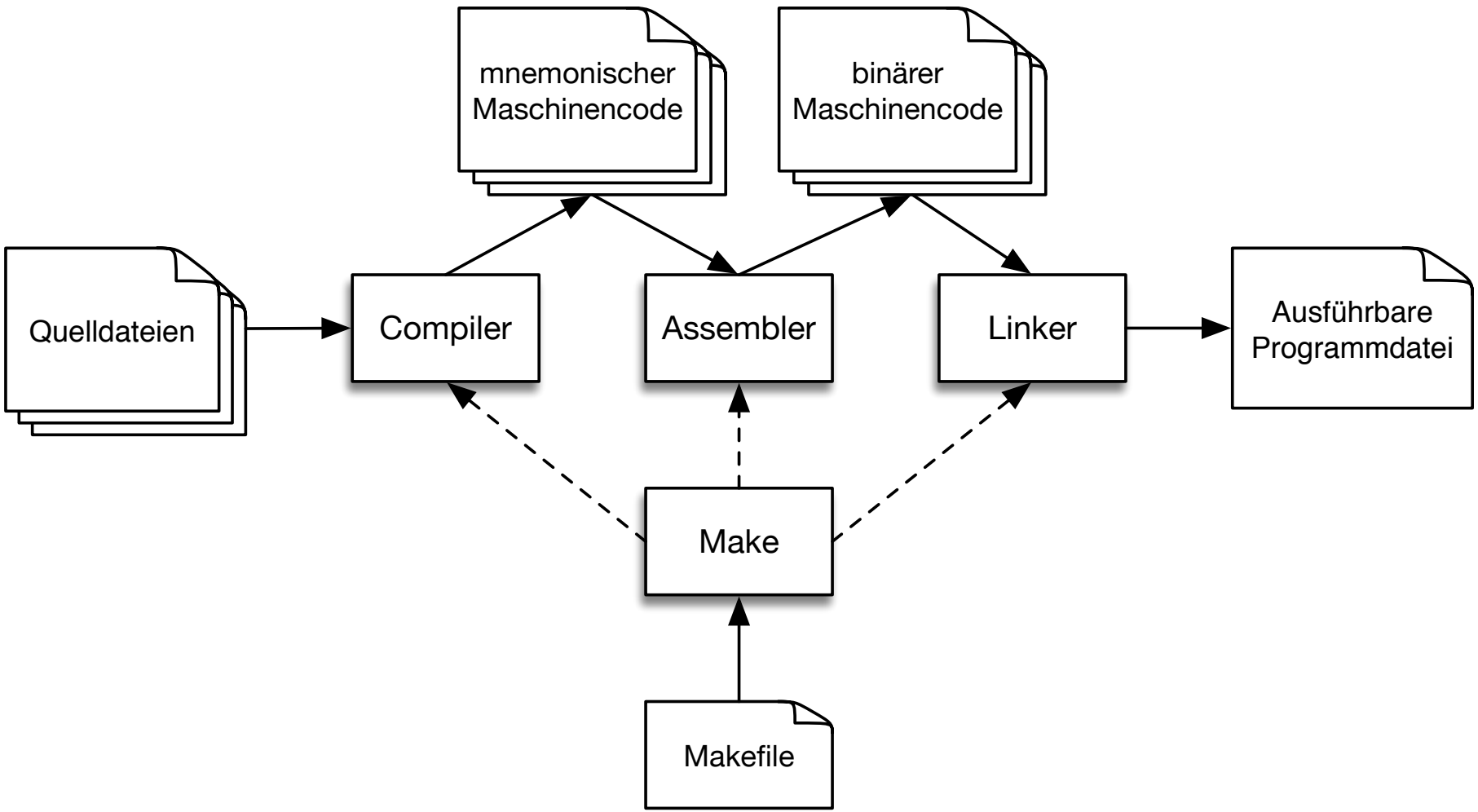
Übersetzung

Begriffe

Quelldatei	Programmcode einer Hochsprache
Objektdatei	Programmcode einer Maschinensprache
Bibliothek	Kollektion verwandter Objektdateien
Compiler	überführt High-Level- in Low-Level-Code
Assembler	konvertiert mnemonischen Maschinensprachencode in binären Maschinencode
Linker	löst symbolische Referenzen zwischen Objektdateien auf, erzeugt eine ausführbare Datei
Makefile	Eingabe für das Programm `make`, die alle relevanten Abhängigkeiten im Übersetzungsprozess spezifiziert

Übersetzung

Zusammenhang



Inline-Assembly

- C-Code kann Assembly-Code beinhalten
- allgemeines Format für gcc

```
asm("<AT&T Assembly>"
:   Ausgabeoperanden /* optional */
:   Eingabeoperanden /* optional */
:   Seiteneffekte    /* optional */
)
```

- kann prinzipiell überall stehen, aber kommt gewöhnlich nur in Funktionskörpern zur Ausführung

Eigenschaften von Assembly

- Maschinencode, untypisiert, registerbasiert, auf den Befehlssatz einer konkreten Maschine zugeschnitten
- maximal effizient, maximal flexibel, minimal portabel
- Assembler führt 1:1-Übersetzung in binären Maschinencode durch, sammelt globale Labels in einer Relokationstabelle und ersetzt lokale Labels durch konkrete Adressen

Begriffe

Anweisung Befehl, der durch die CPU ausgeführt wird; symbolisch notiert

Register hart auf der CPU verdrahteter Variablenslot; Quelle und Ziel von Anweisungen

Label symbolische Programmadresse; wird durch den Assembler durch die konkrete Adresse ersetzt

Flag bei der Ausführung bestimmter Instruktionen automatisch gesetztes Bit mit spezieller Semantik

Interrupt Unterbrechung des normalen Programmablaufs zur Behandlung eines asynchronen Ereignisses

Sektion benannter Bereich in einer Assembly-Datei

BIOS Schnittstelle zur geräteunabhängigen Ein- und Ausgabe; Initialisierung von Hardware und Betriebssystem

Registersatz

- Unterscheidung zwischen frei verwendbaren und Spezialregistern
- freie Register: Ausführung ohne Seiteneffekte
  - *Accumulator* `a`, *Base* `b`, *Count* `c`, *Data* `d`
  - 16-Bit Architektur: `ax`, `bx`, `cx`, `dx`
  - 32-Bit Architektur: `eax`, `ebx`, `ecx`, `edx`
  - 64-Bit Architektur: `rax`, `rbx`, `rcx`, `rdx`
- Spezialregister: Schnittstelle zu internem Prozessorzustand
  - `ip`: *Instruction Pointer*; wird nach jeder Instruktion erhöht
  - `bp`, `sp`: *Base Pointer, Stack Pointer*; zeigen auf Anfang und Ende des aktuellen Stack-Frames
  - `si`, `di`: *Source Index*, *Destination Index*; Iteration von Arrays
  - `of` (*Overflow*), `if` (*Interrupt*), `tf` (*Trap*), `sf` (*Sign*), `zf` (*Zero*), `cf` (*Carry*): einzelnes Kontrollregister; Flags, die während der Programmabarbeitung durch die CPU gesetzt werden

Registerstruktur

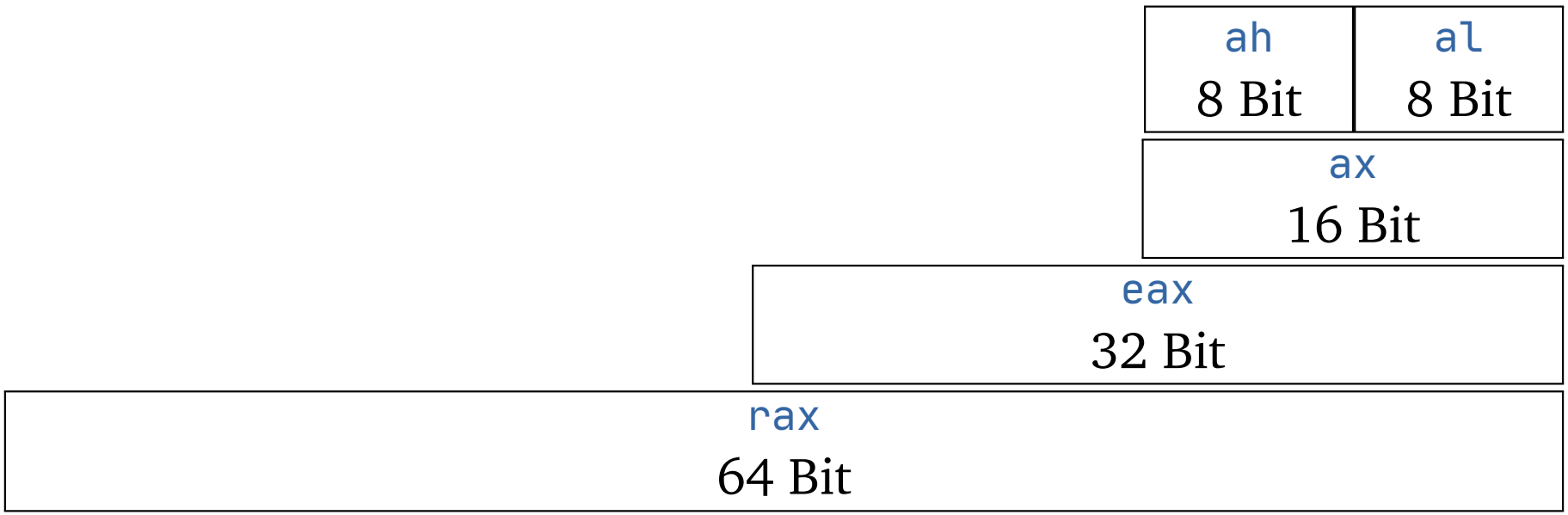


Abbildung: Interner Registeraufbau

Syntax

- grundsätzliches Format:
  ```
  mnemonic src,dst
  ```
- Registern wird `%` vorangestellt:
  ```
  mov %eax,%ebx
  ```
- Literalen wird `$` vorangestellt:
  ```
  mov $100,%ebx
  ```
- Labels sind Literale, deren konkrete Adressen bei der Assemblierung substituiert werden:
  ```
  mov $hello,%ebx
  ```
- Sprünge benötigen für Labels kein vorgestelltes `$`

Syntax

- Arbeitsspeicher lässt sich nach folgendem Schema adressieren: `seg:off(base,index,scale)`
  - `mov %cx,100` — `*100 = cx`
  - `mov %cx,(%eax)` — `*eax = cx`
  - `mov %cx,(%eax,%ebx)` — `*(eax+ebx) = cx`
  - `mov %cx,(%eax,%ebx,2)` — `*(eax+ebx*2) = cx`
  - `mov %cx,(,%ebx,2)` — `*(ebx*2) = cx`
  - `mov %cx,-10(%eax)` — `*(eax-10) = cx`
- wenn die Größe sich nicht aus Quelle oder Ziel ergibt, sollte sie explizit via Suffix angegeben werden: `b` (1 Byte), `w` (2 Bytes), `l` (4 Bytes) oder `q` (8 Bytes)
  - `movb (%ebx),(%eax)` kopiert ein Byte von der Speicheradresse `ebx` an die Speicheradresse `eax`

Befehle

Ausdrücke & Sprünge

- `mov src,dst` — `dst = src`
- `add op,dst` — `dst += op`
- `sub`, `imul`, `div`, `mod`, `and`, `or`, `xor` — `-`, `*`, `/`, `%`, `&`, `|`, `^`
- `lea off(base,i,s),dst` — `dst = base+i*s+off`
- `int code` — führt ein Prozessorinterrupt aus
- `lods` — `a = *si++`
- `stos` — `*si++ = a`
- `cmp lhs,rhs` — Vergleich, Ergebnis in Kontrollregister
- `jmp loc` — Sprung an `loc`
- `je loc` — Sprung an `loc`, falls `zf` gesetzt
- `jg`, `jge`, `jl`, `jle`, `jne` — Sprung falls $>$, $\geqslant$, $<$, $\leqslant$, $\neq$

Befehle

Funktionsstack

`push op` Sicherung von `op` auf dem Stack

`push %eax` ist äquivalent zu

```
sub $4, %rsp
mov %eax, (%rsp)
```

`pop dst` Wiederherstellung eines Stackwertes

`pop %ax` ist äquivalent zu

```
mov (%rsp), %ax
add $2, %rsp
```

`call adr` Ruf einer Funktion

`call print` ist äquivalent zu

```
push %rip
jmp print
```

`ret` Rücksprung an Aufrufadresse

`ret` ist äquivalent zu

```
pop %rip
```

„Hello World“ in Assembly

```
1  .text
2  .globl _start
3  _start:
4      mov $hello, %rsi
5      mov $0xE, %ah
6  .loop:
7      lodsb
8      cmp $0, %al
9      je .done
10     int $0x10
11     jmp .loop
12 .done:
13     ret
14 .data
15 hello:
16     .asciz "Hello World!"
```

Quadrieren

Schreiben Sie eine Funktion `int square(int a)`, die das Integer-Argument quadriert und rufen Sie sie wie `square(2)`.

```
1  _square:
2      imul %rax, %rax
3      ret
4
5  _main:
6      mov $2, %rax
7      call _square
8      ret
```

Maximum

Schreiben Sie eine Funktion `int max(int a, int b)`, die von zwei Argumenten in den Registern `rax` und `rbx` das größere in `rax` zurückgibt.

```
1 _max:
2     cmp %rax, %rbx
3     jg .done
4     mov %rbx, %rax
5 .done:
6     ret
```

Größter gemeinsamer Teiler

Implementieren Sie den euklidischen Algorithmus zur Berechnung des ggT. Argumente werden in `rax` und `rbx` übergeben.

```
1 int gcd(int a, int b) {
2     while (b ≠ 0) {
3         int c = a % b;
4         a = b;
5         b = c;
6     }
7     return a;
8 }
```

- `idiv` teilt einen 128-Bit Dividenten durch einen 64-Bit Divisor
- Divident wird in zwei Registern gehalten: `rdx` und `rax`
- Ergebnis der Division landet in Register `rax`
- Divisionsrest landet in Register `rdx`

Größter gemeinsamer Teiler

Lösung

```
1  _gcd:
2  .cont:
3      cmp  $0, %rbx
4      je   .done
5      mov  $0, %rdx
6      idiv %rbx
7      mov  %rbx, %rax
8      mov  %rdx, %rbx
9      jmp  .cont
10 .done:
11     ret
```