Betriebssysteme I

Praktikum im SoSe 2020

Prof. Jens-Peter Redlich
Dorian Weber

28. April 2020

Überblick

- 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

Einfache Beispiele

- einfache Anweisung: `asm("mov $0, %rax");`
- mehrere Anweisungen:

```
asm(
    "mov $0, %rax;"
    "mov $1, %rbx;"
);
```

- nicht zu optimierende Anweisungen mit Seiteneffekten:

```
asm volatile(
    "mov $60, %rax;" // exit()
    "mov $0, %rdi;"  // Rückgabe: Erfolg!
    "syscall;"       // Kontrolltransfer zum Kernel
);
```

Randbedingungen

r	universelles Register
a	`rax`, `eax`, `ax`, `al`
b	`rbx`, `ebx`, `bx`, `bl`
c	`rcx`, `ecx`, `cx`, `cl`
d	`rdx`, `edx`, `dx`, `dl`
S	`rsi`, `esi`, `si`
D	`rdi`, `edi`, `di`

Tabelle: Spezifikation der Belegung von Ein- und Ausgaberegistern

Erweiterte Beispiele I

- Angabe von Ein- und Ausgaberegistern mit Seiteneffekten:

```
long out, in = 10;

asm(
    "mov %1, %%rbx;"
    "mov %%rbx, %0;"
:   "=r"(out) // Ausgabe out zuweisen
:   "r" (in)  // Eingabe in zuweisen
:   "rbx"     // Seiteneffekt auf Register rbx
);
```

- (die zwei % sind nötig, um gcc die Unterscheidung zwischen Registern und Operanden zu erleichtern)

Erweiterte Beispiele II

- mehrere Eingaberegister:

```
const char* hello = "Hello World!";
asm(
    "syscall"
::  "a"(1), "D"(1),
    "S"(hello), "d"(strlen(hello))
:   "rax"
);
```

- mehrere Ein- und Ausgaberegister:

```
long x = 45, y = 13, div, mod;
asm(
    "div %%rbx"
:    "=a"(div), "=d"(mod)
:    "d"(0), "a"(x), "b"(y)
);
```

Erweiterte Beispiele III

Euklidischer Algorithmus

```
int gcd(int a, int b) {
    int result;

    asm(".cont:"
            "cmp $0, %%ebx;"
            "je .done;"
            "mov $0, %%edx;"
            "idiv %%ebx;"
            "mov %%ebx, %%eax;"
            "mov %%edx, %%ebx;"
            "jmp .cont;"
        ".done:"
    :   "=a"(result)
    :   "a"(a), "b"(b)
    :   "edx"
    );

    return result;
}
```

Übung

Inline Assembly

1. Implementieren Sie einen Algorithmus in inline Assembly zum Vertauschen zweier Zahlen.
2. Implementieren Sie eine Funktion `strlen()` mit inline Assembly zum Ermitteln der Länge einer Zeichenkette.
3. Die Assembly-Instruktion `popcnt src, dst` bestimmt die Anzahl der gesetzten Bits in einer Zahl. Implementieren Sie eine Funktion `bitCount()`, die die Anzahl gesetzter Bits in einem Integer-Array bekannter Größe berechnet.

Lösung

Erste Aufgabe

```
1 asm(""
2 :   "=a"(b), "=b"(a)
3 :   "a"(a), "b"(b)
4 );
```

Lösung

Zweite Aufgabe

```
1  int strlen(const char* str) {
2      int len;
3      asm("mov $0, %0;"
4          ".loop:"
5              "lodsb;"
6              "cmp $0, %%al;"
7              "je .done;"
8              "add $1, %0;"
9              "jmp .loop;"
10         ".done:"
11     :   "=r"(len)
12     :   "S"(str)
13     :   "ax"
14     );
15     return len;
16 }
```

Lösung

Dritte Aufgabe

```
1  int bitCount(const int* bitset, int size) {
2      int sum = 0, e;
3
4      for (int i = 0; i < size; ++i) {
5          asm("popcnt %1, %0"
6          :   "=r"(e)
7          :   "r"(bitset[i])
8          );
9
10         sum += e;
11     }
12
13     return sum;
14 }
```

Unterbrechungsanfrage

Überblick

- zwei Arten von Interrupts: Hardware- und Software-Interrupts
- Hardware-Interrupts werden durch die angeschlossene Peripherie asynchron ausgelöst
- Software-Interrupts werden mittels `int $code` vom Programm selbst ausgelöst
- Nutzung zur Interaktion mit BIOS:
  - Auslösen eines Interrupts durch Nutzerprogramm
  - Lokalisation der Unterbrechungsroutine (*interrupt service routine*) in einer Tabelle (*interrupt vector table*)
  - Ausführung der Unterbrechungsroutine
  - Rückgabe der Kontrolle an das Programm
- *interrupt vector table* wird initial aus dem ROM geladen und lässt sich beim Booten dann überschreiben

Unterbrechungsanfrage

Ablauf

1. Prozessor wartet, bis die Ausführung des aktuellen Maschinenbefehls abgeschlossen ist, bevor die Unterbrechung behandelt wird
2. Interruptzyklus wird ausgeführt:
   - Befehlszähler (Instruction Pointer `ip` und Codesegment `cs`) wird auf dem Stack gesichert
   - Unterbrechungsroutine wird in einer Tabelle von Interruptvektoren lokalisiert
   - Programm fährt dort fort
3. Unterbrechungsroutine sichert alle durch sie genutzten Register auf dem Stack und führt die Anfrage aus
4. ursprünglicher Programmzustand wird wiederhergestellt

BIOS

- Problem: Computer sind modular aufgebaut und können viele Komponenten unterschiedlicher Hersteller enthalten
- Lösung: standardisierte Schnittstellen für alle Geräte und einheitliches Programm zur Initialisierung
- Programm des *Basic Input/Output System* liegt auf dem Mainboard in EEPROM und läuft bei Rechnerstart automatisch
- Aufgabe ist Identifikation, Test und Initialisierung angeschlossener Peripherie (*Power-On Self-Test*)
- initialisiert *interrupt vector table*, sucht nach bootbaren Medien
- stellt Funktionen zum Hardwarezugriff zur Verfügung
  - Festplattenzugriff
  - Tastatureingabe
  - Bildschirmausgabe
  - Druckerzugriff

BIOS

Beispiele

- Ausgabe des ASCII-Zeichens `'H'` auf dem Bildschirm:

```
mov $0xE48, %eax
mov $0x7, %ebx
int $0x10
```

- Einlesen eines Zeichens von der Tastatur:

```
mov $0, %eax
int $0x16
```

- Ausgabe eines eingelesenen Zeichens:

```
mov $0, %eax
int $0x16
mov $0x7, %ebx
mov $0xE, %ah
int $0x10
```

- Wikipedia hat eine Liste möglicher *interrupt calls*

Master Boot Record

- befindet sich im ersten Sektor auf einem bootbaren Medium
- 512 Bytes groß, wird von Boot-Firmware an RAM Adresse `0x7C00` geladen
- Aufgabe: Betriebssystemkern laden und Kontrolle abgeben

Adresse	Funktion	Größe
`0x7C00`	Programmcode	446 Bytes
`0x7DBE`	Partitionstabelle	64 Bytes
`0x7DFE`	Bootsektor-Signatur (0x55AA)	2 Bytes

Tabelle: Struktur des *klassischen* MBR

Master Boot Record

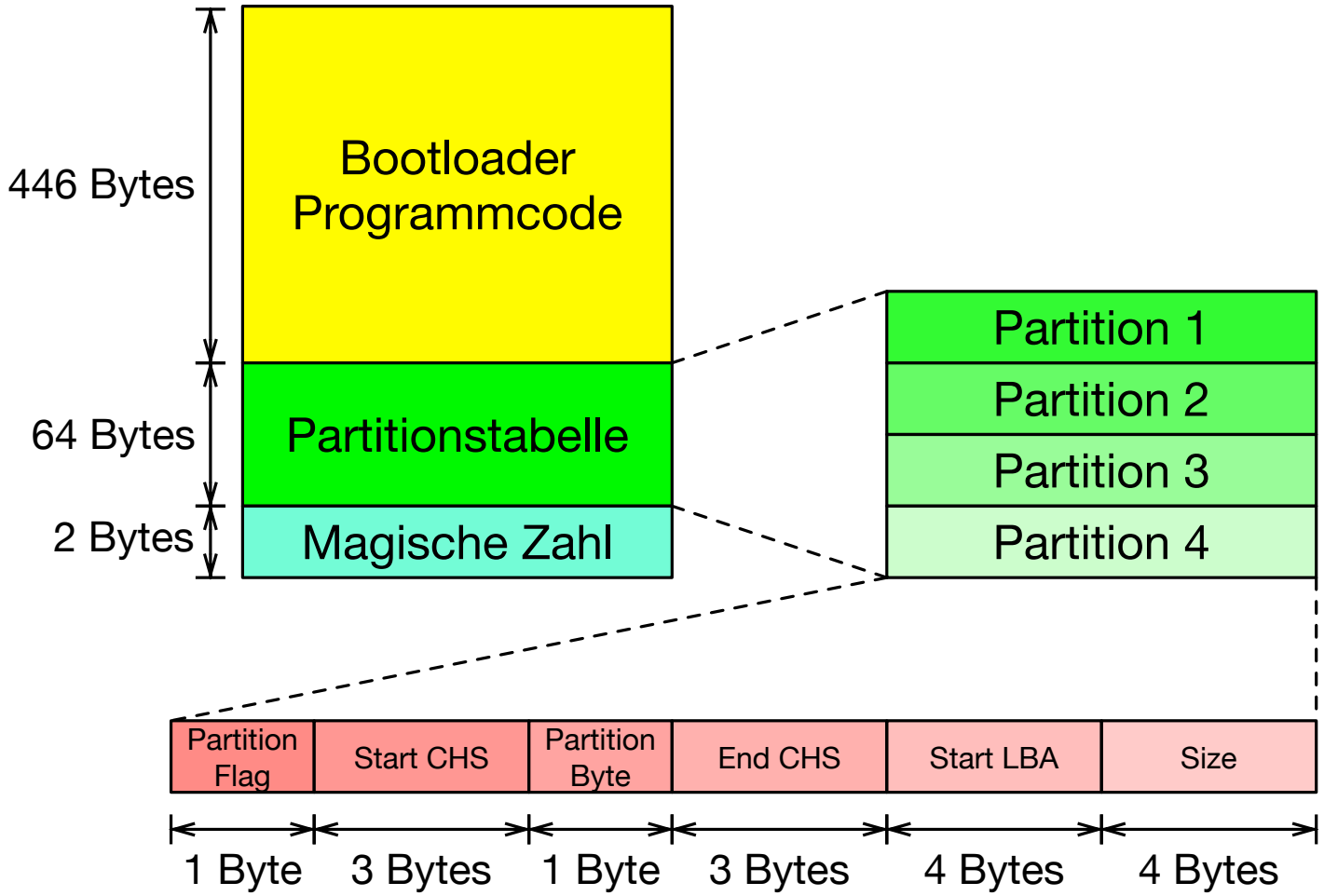

Abbildung: Grafische Darstellung der klassischen MBR Struktur

Bootloader

Aufgabenstellung

- Implementation eines Bootloaders
- Vorgabe läuft während des Bootvorgangs in eine Endlosschleife
- folgende Dinge müssen erweitert werden:
  - Ausgabe einer Zeichenkette auf dem Bildschirm
  - verdeckte Eingabe einer Zeichenkette, Ausgabe von Punkten
  - Ausgabe nach Drücken der Enter-Taste
  - Neustart des Systems, falls Eingabe leer
- Bonuspunkte für weitere Funktionalität
  - kleines Spiel
  - kleine Grafikdemo
  - Nachladen von Code
- Makefile darf modifiziert werden, aber keine Änderungen an Compilerflags erlaubt

Vorgegebene Quelldatei

```
1  asm(".code16gcc\njmp $0, $main");
2
3  void main(void) {
4      asm(
5          "mov $0x007, %%ebx;"
6          "mov $0xE4E, %%eax; int $0x10;"
7          "mov $0xE69, %%eax; int $0x10;"
8          "mov $0xE63, %%eax; int $0x10;"
9          "mov $0xE65, %%eax; int $0x10;"
10         "mov $0xE20, %%eax; int $0x10;"
11         "mov $0xE42, %%eax; int $0x10;"
12         "mov $0xE6F, %%eax; int $0x10;"
13         "mov $0xE6F, %%eax; int $0x10;"
14         "mov $0xE74, %%eax; int $0x10;"
15         "mov $0xE73, %%eax; int $0x10;"
16         "jmp .;"
17         ::: "eax", "ebx"
18     );
19 }
```

Vorgegebene Linkerdatei

```
ENTRY(main);
SECTIONS
{
    . = 0x7C00;
    .text : AT(0x7C00)
    {
        _text = .;
        *(.text);
        _text_end = .;
    }
    .data :
    {
        _data = .;
        *(.bss);
        *(.bss*);
        *(.data);
        *(.rodata*);
        *(COMMON)
        _data_end = .;
    }
    .sig : AT(0x7DFE)
    {
        SHORT(0xAA55);
    }
    /DISCARD/ :
    {
        *(.note*);
        *(.iplt*);
        *(.igot*);
        *(.rel*);
        *(.comment);
        *(.eh_frame);
    }
}
```

Vorgegebenes Makefile

```
SRC = bootloader.c
TAR = bootloader.bin
PCK = lab-1.zip
CFLAGS = -m32 -c -Os -march=i686 -ffreestanding -Wall -Werror
LFLAGS = -m elf_i386 -static -Tlinker.ld -nostdlib --nmagic

%.o: %.c
    $(CC) $(CFLAGS)  $^ -o $@
%.elf: %.o
    $(LD) $(LFLAGS) -o $@ $^
%.bin: %.elf
    objcopy -O binary $^ $@

all: $(TAR)
run: $(TAR)
    qemu-system-x86_64 -drive format=raw,file=$^
pack:
    zip $(PCK) Makefile *.c *.h *.s
clean:
    $(RM) $(RMFILES) $(TAR) $(PCK)
```