

gcc -Os

Know your binary

Knowing my binary - WTF?

- Erstellen eines C-Programmes, das NUR *return* aufruft
- Überprüfen und Optimieren der Größe des Binarys
- Warum?
 - Weil es geht! ... und weil es interessant ist! ...
 - Weil Verschwendung auch in der Zeit unendlicher Mengen von Speicher kein guter Stil ist?
 - Weil auch die höchste Sprache immer schlechten Low-Level-Code verwendet und es deshalb nie verkehrt ist, diesen zu kennen

Metainformationen (1)

- **gcc** ist die Compiler Collection des GNU-Projektes
 - Option -o definiert den Namen der Ausgabedatei (Binary)
 - Option -O definiert verschiedene Optimierungsstufen
 - Aktiviert verschiedene -f Optionen
 - Option -s verwendet das UNIX-Programm *strip* zum entfernen unnötiger Symbole aus einer Objektdatetei
- Rückgabewerte werden in der Variable **\$?** gespeichert
 - Ausgabe mit echo \$?
- Größe kann mit **wc -c** angezeigt werden
 - Zählt die Zeichen, in der Datei (Zeichen = Byte)

Ergebnis: C inkl. Optimierung

- Trotz Compileroptimierungen ist die Binärdatei „RIESIG“
- Wie verkleinert man ein 1-zeiliges Programm?
- C scheint Overhead beim Erstellen von Objektdaten zu erzeugen
- Wo kommt dieser Overhead her?
- Nutzung von Assembler
 - Bei diesem Mini-Programm kein Problem
 - Akkumulator eax auf 42 setzen und return aufrufen

Metainformationen (2)

- Aufbau der Assemblerdatei
 - Header
 - Codesektionen (.text)
 - Datensektionen (.data)
- NASM – Netwide Assembler
 - Option -o gibt die Ausgabedatei an
 - Option -f für das Ausgabeformat
 - Option -l zum Erstellen eines Listfiles
- Die Objektdatei wird anschließend mit Hilfe von GCC zur Executable (Stichwort: *ld*)

```
;ein Kommentar  
BITS 32  
EXTERN <name>  
GLOBAL main
```

```
SECTION .text  
main:  
    mov eax, 42  
    call blubb  
    ret
```

```
blubb:  
    ret
```

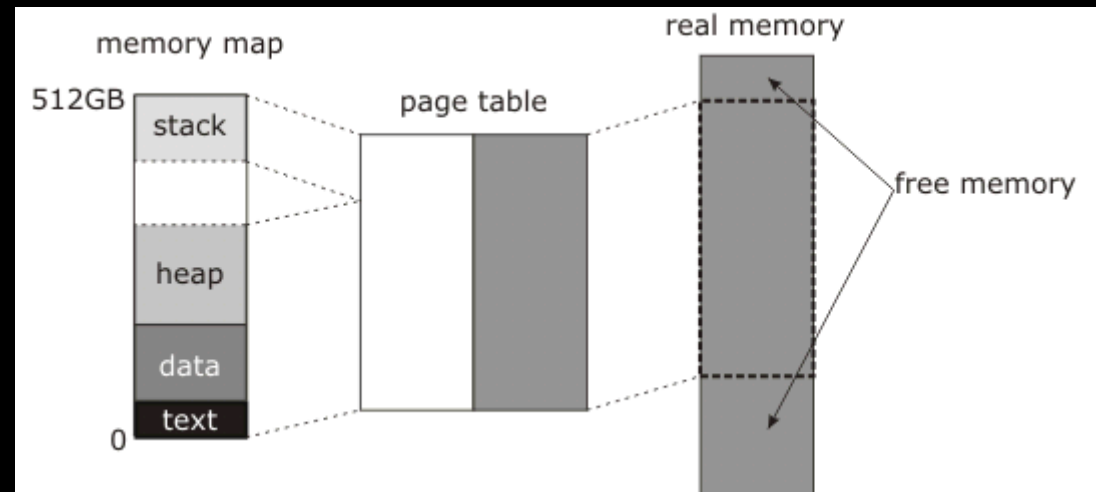
```
SECTION .data  
msg db      „Hello“, 0xa  
len equ    $ - msg
```

Ergebnis: Assembler Programm

- Trotz Assembler ist die Binärdatei „RIESIG“
 - Was passiert während der Umsetzung des Assembler in Binärcode?
- Was hat es eigentlich mit `main()` aufsich?
 - Der Linker fügt ein Interface zum OS hinzu, das dann für den Aufruf von `main()` verantwortlich ist
 - Der eigentliche Einstiegspunkt des Linkers ist `_start`
 - GCC fügt `_start` automatisch hinzu

Aktueller Inhalt des Stack

- Segfault? - Sowas passiert mir in Java nicht!
- Wir verzichten auf das *main()* Interface
 - Wie sieht unser Stack überhaupt aus?
 - Wie endet *_start* eigentlich?
- Verzicht auf *ret* Anweisung
- Stattdessen direkter Aufruf von *_exit*
- → *man exit(2)*



Ergebnis: Assembler (2)

- Trotz Verzicht auf *main()*-Interface ist die Binärdatei „RIESIG“
 - Langsam wird es nervig ... wo kommt der Overhead her?
 - Interessanter Hinweis auf der GCC-Homepage

`-nostartfiles`
Do not use the standard system startup files when linking. The standard system libraries are used normally, unless `-nostdlib` or `-nodefaultlibs` is used.

`-nodefaultlibs`
Do not use the standard system libraries when linking. Only the libraries you specify will be passed to the linker, options specifying linkage of the system libraries, such as `-static-libgcc` or `-shared-libgcc`, will be ignored. The standard startup files are used normally, unless `-nostartfiles` is used. The compiler may generate calls to `memcpy`, `memset`, `memcpy` and `memmove`. These entries are usually resolved by entries in `libc`. These entry points should be supplied through some other mechanism when this option is specified.

`-nostdlib`
Do not use the standard system startup files or libraries when linking. No startup files and only the libraries you specify will be passed to the linker, options specifying linkage of the system libraries, such as `-static-libgcc` or `-shared-libgcc`, will be ignored. The compiler may generate calls to `memcpy`, `memset`, `memcpy` and `memmove`. These entries are usually resolved by entries in `libc`. These entry points should be supplied through some other mechanism when this option is specified.

- Die Standard-C-Bibliothek (glibc)?!
 - Die brauchen wir doch nicht, nur um ein Programm zu beenden

Metainformationen (3)

- Ok, wir brauchen die Stdlib um ein Programm zu beenden!
 - Die Routine `_exit` ist hier definiert
- Aber: Man muss ein Programm ohne Stdlib beenden können ...
 - Gab es da nicht Int.. Int.. Softwareinterrupts?
 - Aus der Signal-Manpage

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process

- Überarbeitung des Programmcodes
 - Setzen des Registers ebx auf 42 (= Rückgabewert)
 - Setzen des Registers eax auf 1 (= Signal)
 - Auslösen eines Softwareinterrupts (= 0x80)

Ergebnis: Ohne glibc

- Die Binärdatei hat endlich eine *akzeptable* Größe
- Kann man noch mehr optimieren?
 - Ja: Kürzere Assembler-Befehle
 - ld statt gcc zum Linken verwenden
- Wir haben das Binary auf 1/24 der Ursprungsgröße gebracht
- Mehr geht wirklich nichtmehr
 - Oder?
 - Eigentlich haben wir nur 7 Byte Programmcode ...

ELF (1)

- Weitere Optimierung erfordert ein Verständnis des Binärformates
- In unserem Falle das ELF
 - Verschiedene Header
 - ELF Header (52 Byte)
 - Program Header (32 Byte)
 - Section Header
 - Symboltabelle
 - Programmsektionen
 - Datensektionen

Linking View	Execution View
ELF header	ELF header
Program header table <i>optional</i>	Program header table
Section 1	Segment 1
...	
Section <i>n</i>	Segment 2
...	
...	...
Section header table	Section header table <i>optional</i>

ELF (2)

- Programm Header
 - ELF ID (7F 45 4C 46)
 - Typ der Datei
 - Systemarchitektur
 - Version des EFL-Standard
 - Abstand des Programm- und Section Header vom Beginn der Datei
- Section Header
 - Optional bei Executables - Aber vorhanden!
 - Name und Typ der Sektion
 - Abstand des ersten Byte der Sektion vom Beginn der Datei

```
#define EI_NIDENT      16

typedef struct {
    unsigned char    e_ident[EI_NIDENT];
    Elf32_Half       e_type;
    Elf32_Half       e_machine;
    Elf32_Word       e_version;
    Elf32_Addr       e_entry;
    Elf32_Off        e_phoff;
    Elf32_Off        e_shoff;
    Elf32_Word       e_flags;
    Elf32_Half       e_ehsize;
    Elf32_Half       e_phentsize;
    Elf32_Half       e_phnum;
    Elf32_Half       e_shentsize;
    Elf32_Half       e_shnum;
    Elf32_Half       e_shstrndx;
} Elf32_Ehdr;
```

```
typedef struct {
    Elf32_Word       sh_name;
    Elf32_Word       sh_type;
    Elf32_Word       sh_flags;
    Elf32_Addr       sh_addr;
    Elf32_Off        sh_offset;
    Elf32_Word       sh_size;
    Elf32_Word       sh_link;
    Elf32_Word       sh_info;
    Elf32_Word       sh_addralign;
    Elf32_Word       sh_entsize;
} Elf32_Shdr;
```

ELF (3)

- Verschiedene Teile des Programmes können sich an beliebigen Positionen in der Binärdatei befinden ... und überlappen!
- Verschiedene Parameter der Header werden nicht verwendet
 - Das Feld *Identification* ist 16 Byte lang
 - Benutzt werden jedoch nur 8 Byte
 - Wir benötigen nur 7 Byte für unser Programm :)
- Unser Programm ist jetzt exakt so lang, wie
 - 1 ELF Header (52 Byte) und
 - 1 Programm Header (32 Byte)
- Doubleplusgood!

ELF (4)

- Nochmal:
 - Verschiedene Teile des Programmes können sich an beliebigen Positionen in der Binärdatei befinden ... **und überlappen!**
- Das Ende des ELF-Header und der Begin des Program-Header sind zum Teil *identisch*
 - Ein paar kleine Modifikationen an den Positions- und Längenangaben erlauben und, die beiden Header *zusammenzuschieben*

```

        dw      ehdrsize      ; e_ehsize
        dw      phdrsize      ; e_phentsize
        dw      1             ; e_phnum
        dw      0             ; e_shentsize
        dw      0             ; e_shnum
        dw      0             ; e_shstrndx

ehdrsize equ    $ - ehdr

phdr:
        dd      1             ; Elf32_Phdr
        dd      0             ; p_type
        dd      $$            ; p_offset
        dd      $$            ; p_vaddr
        dd      $$            ; p_paddr
        dd      filesize      ; p_filesz

```

ELF (5)

- Welche Werte der Header werden tatsächlich benötigt?

unsigned char	e_ident[EI_NIDENT];	7F 45 4C 46
Elf32_Half	e_type;	2 (Executable)
Elf32_Half	e_machine;	3 (i386)
Elf32_Word	e_version;	ELF-Standard
Elf32_Addr	e_entry;	Startadresse
Elf32_Off	e_phoff;	Program Header Offset
Elf32_Off	e_shoff;	Section Header Offset
Elf32_Word	e_flags;	(unbenutzt)
Elf32_Half	e_ehsize;	Verifikation der ELF Header Größe
Elf32_Half	e_phentsize;	Verifikation der Program Header Größe
Elf32_Half	e_phnum;	Größe des Program Header
Elf32_Half	e_shentsize;	Verifikation der Section Header Größe
Elf32_Half	e_shnum;	Größe des Section Header
Elf32_Half	e_shstrndx;	Index des Section Header Table
Elf32_Word	p_type;	1
Elf32_Off	p_offset;	korrektes Offset
Elf32_Addr	p_vaddr;	Adresse zum laden des Programms
Elf32_Addr	p_paddr;	(unbenutzt)
Elf32_Word	p_filesz;	zu ladende Byte
Elf32_Word	p_memsz;	zu reservierender Speicher
Elf32_Word	p_flags;	Speicher-Flags (RX)
Elf32_Word	p_align;	Alignment-Infos (u.a. Relocation)

ELF (5.1)

- Welche Werte der Header werden tatsächlich benötigt?

unsigned char	e_ident[EI_NIDENT];	7F 45 4C 46
Elf32_Half	e_type;	2 (Executable)
Elf32_Half	e_machine;	3 (i386)
Elf32_Word	e_version;	ELF-Standard
Elf32_Addr	e_entry;	Startadresse
Elf32_Off	e_phoff;	Program Header Offset
Elf32_Off	e_shoff;	Section Header Offset
Elf32_Word	e_flags;	(unbenutzt)
Elf32_Half	e_ehsize;	Verifikation der ELF Header Größe
Elf32_Half	e_phentsize;	Verifikation der Program Header Größe
Elf32_Half	e_phnum;	Größe des Program Header
Elf32_Half	e_shentsize;	Verifikation der Section Header Größe
Elf32_Half	e_shnum;	Größe des Section Header
Elf32_Half	e_shstrndx;	Index des Section Header Table
Elf32_Word	p_type;	1
Elf32_Off	p_offset;	korrektes Offset
Elf32_Addr	p_vaddr;	Adresse zum laden des Programms
Elf32_Addr	p_paddr;	(unbenutzt)
Elf32_Word	p_filesz;	zu ladende Byte
Elf32_Word	p_memsz;	zu reservierender Speicher
Elf32_Word	p_flags;	Speicher-Flags (RX)
Elf32_Word	p_align;	Alignment-Infos (u.a. Relocation)

ELF (6)

- Was wäre, wenn wir den Program Header noch weitere 12 Byte nach oben schieben?
 - p_flags muss geändert werden, um mit e_phoff überlappen zu können
 - Das eigentliche Programm wird verschoben
 - Beginnt nun im e_shoff Feld und Endet im e_flags Feld
 - Die Load-Adresse wird verringert
 - Dies beeinflusst den Wert von e_entry und p_memsz
 - p_filesz macht Probleme!
 - Unser Programm überlappt mit e_flags → e_flags ist als RO definiert
 - p_memsz darf deshalb nicht größer sein, als e_filesz
 - Also definieren wir eine e_filesz, die zwar größer ist, als das Programm, aber dafür mit p_memsz übereinstimmt ;)

ELF (7)

- Das Programm ist jetzt exakt so groß, wie ein ELF-Header!
- Kleiner kann nichtmehr gehen, da wir sonst keinen kompletten Header mehr haben ...
- Unglaublich aber wahr
 - Wenn ein ELF File eine geringere Länge hat, als die des ELF Headers, füllt Linux die fehlenden Bytes mit Nullen auf!

```
dw      0x20          ; e_phentsize
dw      1             ; e_phnum
dw      0             ; e_shentsize
dw      0             ; e_shnum
dw      0             ; e_shstrndx

filesize    equ    $ - $$
```

Ergebnis

- Die Hälfte der Werte in der Datei verstößt gegen den ELF Standard
- Aber: Das Binary hat nun weniger als 1/147 der Ursprungsgröße
- Worauf will ich hinaus?
 - Kein normaler Mensch *kann* / *will* eine solche Optimierung betreiben ...
 - Trotzdem ist es oft interessant, Details zu hinterfragen ...
 - Das Verständnis von Details hilft beim Verständnis des ganzen ...
- Worauf noch?
 - Vorlesungen besuchen alleine, macht noch keinen guten Informatiker aus ...
 - Der CCC Fulda, den wir später gründen, soll Raum für genau solche Experimente und vorallem für Wissensaustausch bieten ...
 - Jedoch ist Eigeninitiative gefragt ...

Endlich ist er weg

- Danke, dass Ihr versucht habt wach zu bleiben ;)
- Fragen, Anregungen und vorallem Tipps, wie ich das Ding auf 42 Byte kriege nehme ich jederzeit entgegen
- sven.reissmann@informatik.hs-fulda.de