

Übung zu Betriebssystemtechnik

Aufgabe 4: Trennung von Kern & Anwendungen

14. Juni 2022

Bernhard Heinloth, Phillip Raffeck & Dustin Nguyen

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg



Lehrstuhl für Verteilte Systeme
und Betriebssysteme



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

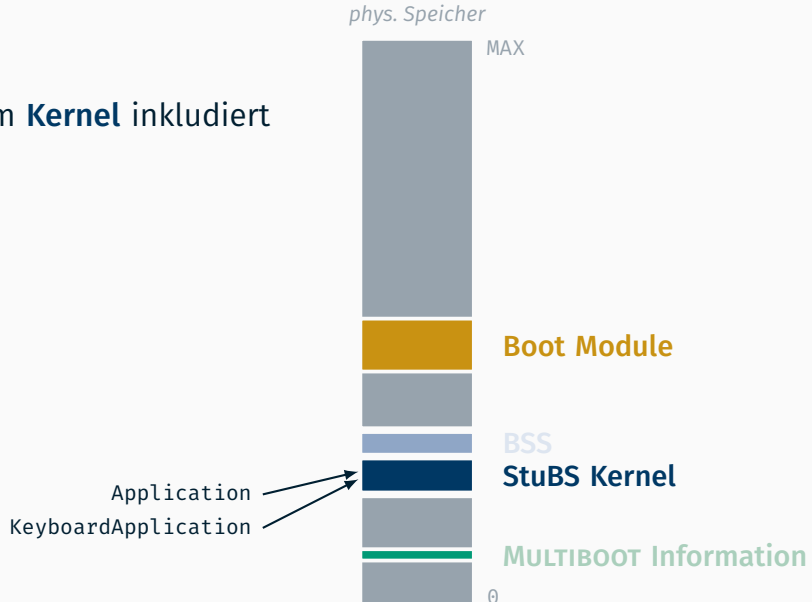
TECHNISCHE FAKULTÄT

Anwendungen sollen (sowohl bei Übersetzung als auch bei Ausführung) vom Kernel getrennt werden

Bestandsaufnahme

Bisher:

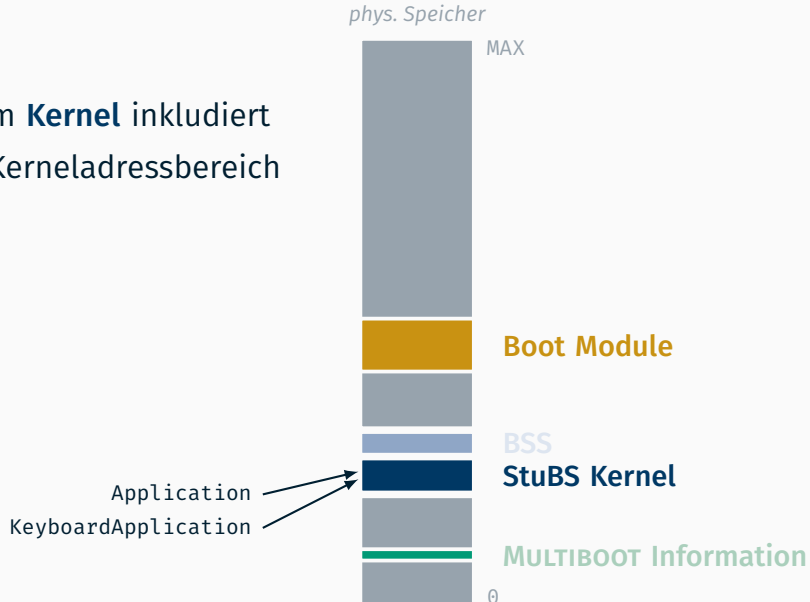
- Anwendungen im **Kernel** inkludiert



Bestandsaufnahme

Bisher:

- Anwendungen im **Kernel** inkludiert
- Ausführung im Kerneladressbereich



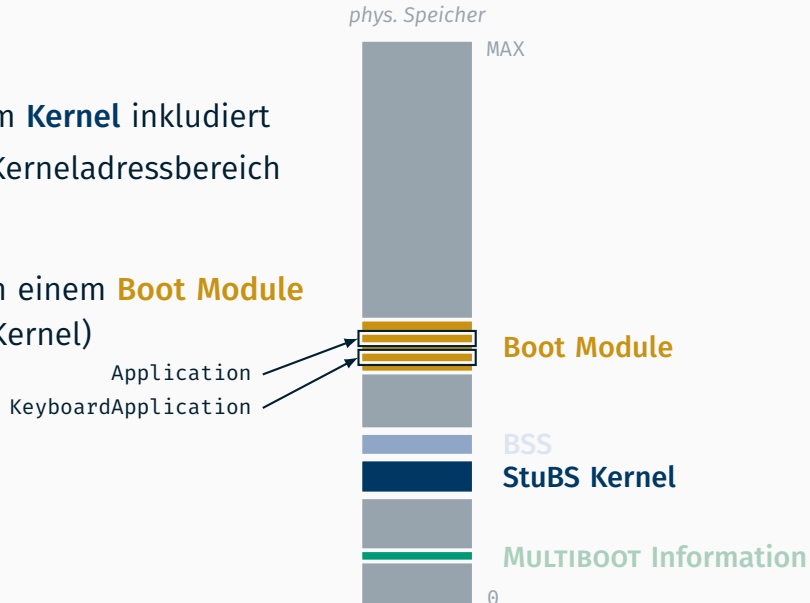
Ziel dieser Aufgabe

Bisher:

- Anwendungen im **Kernel** inkludiert
- Ausführung im Kerneladressbereich

Ab dieser Aufgabe:

- Anwendungen in einem **Boot Module**
(losgelöst vom Kernel)



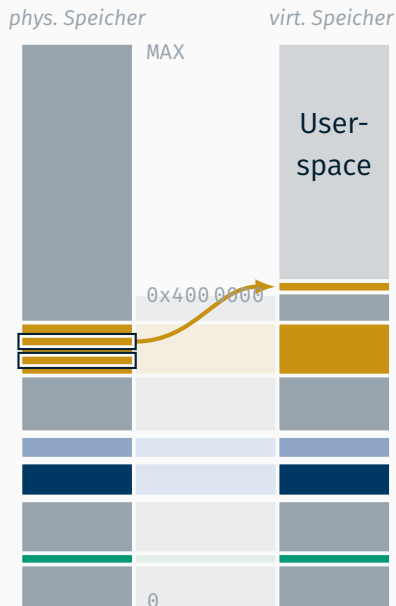
Ziel dieser Aufgabe

Bisher:

- Anwendungen im **Kernel** inkludiert
- Ausführung im Kerneladressbereich

Ab dieser Aufgabe:

- Anwendungen in einem **Boot Module** (losgelöst vom Kernel)
- Ausführung im **Userspace** ab fixer Adresse `0x400 0000`



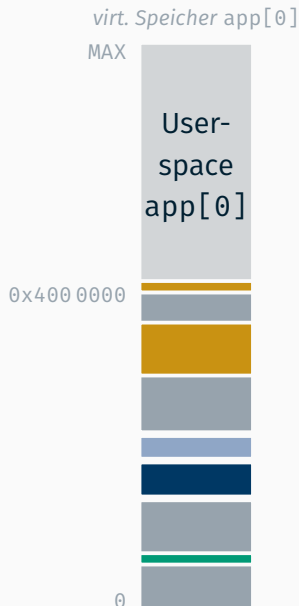
Ziel dieser Aufgabe

Bisher:

- Anwendungen im **Kernel** inkludiert
- Ausführung im Kerneladressbereich

Ab dieser Aufgabe:

- Anwendungen in einem **Boot Module** (losgelöst vom Kernel)
- Ausführung im **Userspace** ab fixer Adresse `0x400 0000` (im jeweils eigenen Adressraum)



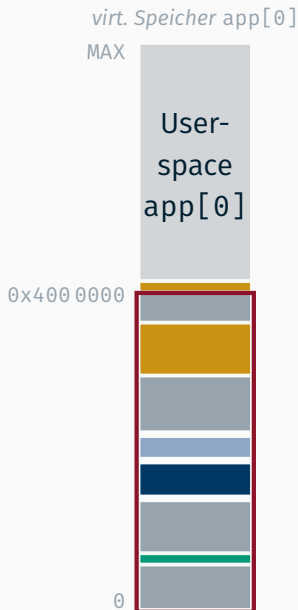
Ziel dieser Aufgabe

Bisher:

- Anwendungen im **Kernel** inkludiert
- Ausführung im Kerneladressbereich

Ab dieser Aufgabe:

- Anwendungen in einem **Boot Module** (losgelöst vom Kernel)
- Ausführung im **Userspace** ab fixer Adresse `0x400 0000` (im jeweils eigenen Adressraum)
- **Kernel vor Zugriff aus Ring 3 geschützt**



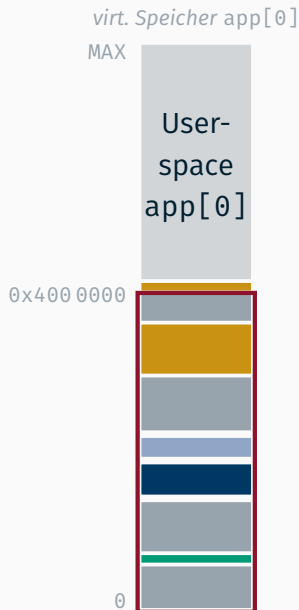
Ziel dieser Aufgabe

Bisher:

- Anwendungen im **Kernel** inkludiert
- Ausführung im Kerneladressbereich

Ab dieser Aufgabe:

- Anwendungen in einem **Boot Module** (losgelöst vom Kernel)
 - Ausführung im **Userspace** ab fixer Adresse `0x400 0000` (im jeweils eigenen Adressraum)
 - **Kernel vor Zugriff aus Ring 3 geschützt**
- **vollständige Isolation der Anwendungen**



Restrukturierung der Quellen

Organisation der Dateien

Bisher:

- Anwendungscode und Betriebssystemcode vermischt

Organisation der Dateien

Bisher:

- Anwendungscode und Betriebssystemcode vermischt
→ alles in einer großen `system[64]`-Datei

Organisation der Dateien

Bisher:

- Anwendungscode und Betriebssystemcode vermischt
→ alles in einer großen `system[64]`-Datei

Nun Aufteilung des Codes in unterschiedliche Verzeichnisse:

`kernel/` nur der Betriebssystemkern

`user/` enthält **mehrere** Anwendungen

Organisation der Dateien

Bisher:

- Anwendungscode und Betriebssystemcode vermischt
→ alles in einer großen `system[64]`-Datei

Nun Aufteilung des Codes in unterschiedliche Verzeichnisse:

`kernel/` nur der Betriebssystemkern

`libs/` Unterstützung für Anwendungen

`user/` enthält **mehrere** Anwendungen

Jede Anwendung wird gegen `libs` gelinkt
und zu einer eigenen Binärdatei kompiliert

Organisation der Dateien

Bisher:

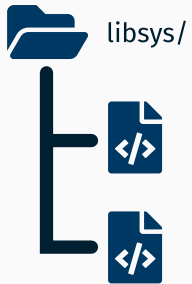
- Anwendungscode und Betriebssystemcode vermischt
→ alles in einer großen `system[64]`-Datei

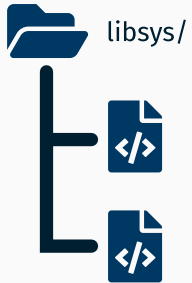
Nun Aufteilung des Codes in unterschiedliche Verzeichnisse:

`kernel/` nur der Betriebssystemkern
→ `system[64]`

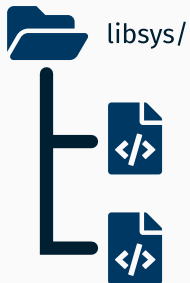
`libsys/` Unterstützung für Anwendungen
→ statische Bibliothek `libsys.a`

`user/` enthält **mehrere** Anwendungen
Jede Anwendung wird gegen `libsys` gelinkt
und zu einer eigenen Binärdatei kompiliert
→ Archiv `initrd.img` mit den einzelnen Binärdateien

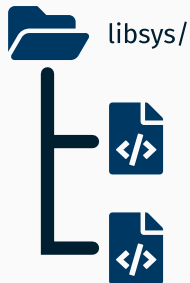




- Systemaufrufstümpfe (aus Aufgabe 2)

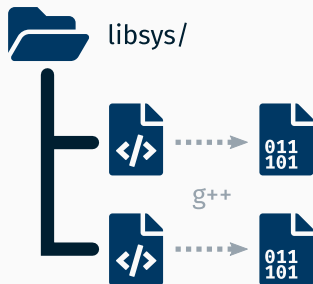


- Systemaufrufstümpfe (aus Aufgabe 2)
- C Funktionsprologe (siehe `compiler/crti.asm` und `compiler/crtn.asm`)



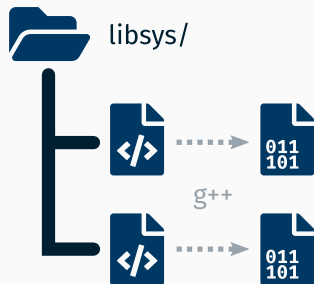
- Systemaufrufstümpfe (aus Aufgabe 2)
- C Funktionsprologe (siehe `compiler/crti.asm` und `compiler/crtn.asm`)
- *optional beliebig erweiterbar*
 - komfortabler Ausgabestrom (siehe `object/outputstream.cc`)
 - dynamischer Allokator → Untertützung für `new` (siehe `utils/alloc.cc`)

```
$ g++ -fno-builtin -nodefaultlibs -nostdlib -nostdinc ...
```



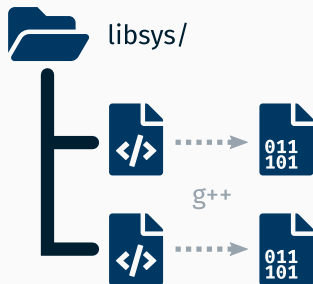
- Systemaufrufstümpfe (aus Aufgabe 2)
- C Funktionsprologe (siehe `compiler/crti.asm` und `compiler/crtn.asm`)
- *optional beliebig erweiterbar*
 - komfortabler Ausgabestrom (siehe `object/outputstream.cc`)
 - dynamischer Allokator → Untertützung für `new` (siehe `utils/alloc.cc`)

→ \$CXXFLAGS aus tools/build.mk berücksichtigen



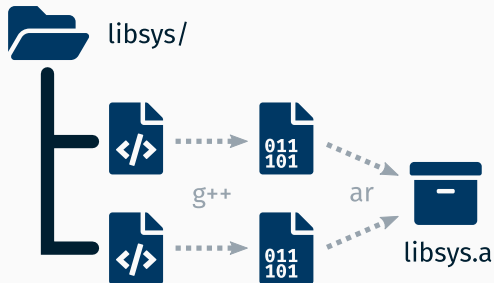
- Systemaufrufstümpfe (aus Aufgabe 2)
- C Funktionsprologe (siehe `compiler/crti.asm` und `compiler/crtn.asm`)
- *optional beliebig erweiterbar*
 - komfortabler Ausgabestrom (siehe `object/outputstream.cc`)
 - dynamischer Allokator → Untertützung für `new` (siehe `utils/alloc.cc`)

→ oder gleich `tools/common.mk` in Makefile einbinden



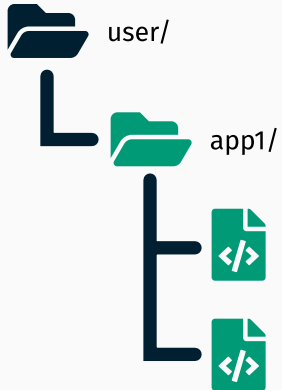
- Systemaufrufstümpfe (aus Aufgabe 2)
- C Funktionsprologe (siehe `compiler/crti.asm` und `compiler/crtn.asm`)
- *optional beliebig erweiterbar*
 - komfortabler Ausgabestrom (siehe `object/outputstream.cc`)
 - dynamischer Allokator → Untertützung für `new` (siehe `utils/alloc.cc`)

```
$ ar rcs libsys.a libsys/*.o
```

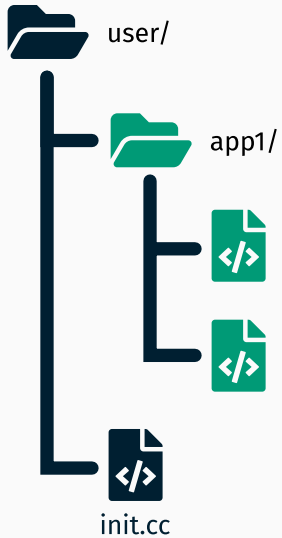


- Systemaufrufstümpfe (aus Aufgabe 2)
- C Funktionsprologe (siehe `compiler/crti.asm` und `compiler/crtn.asm`)
- *optional beliebig erweiterbar*
 - komfortabler Ausgabestrom (siehe `object/outputstream.cc`)
 - dynamischer Allokator → Untertützung für `new` (siehe `utils/alloc.cc`)

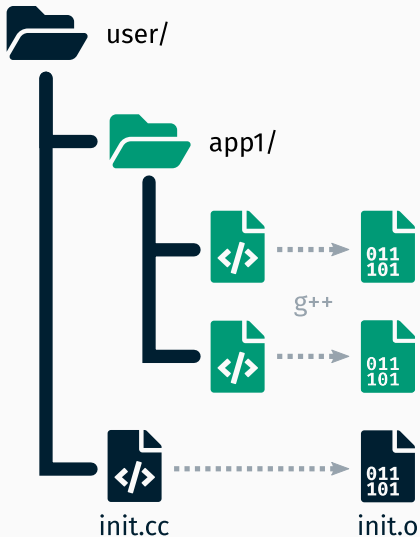
Anwendung



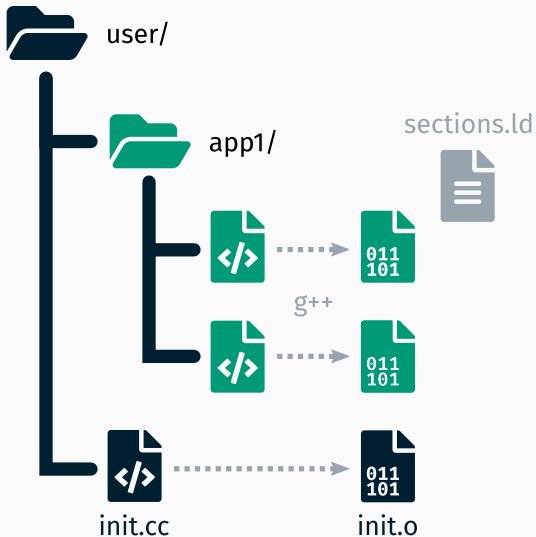
Anwendung

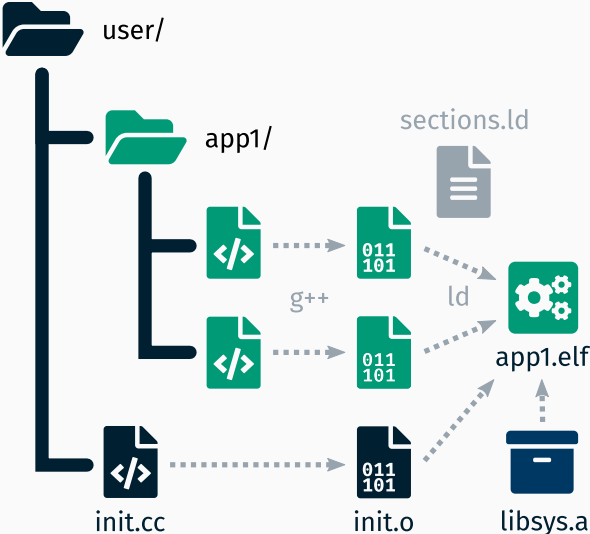


Linkerskript `user/sections.ld` mit `0x400 0000` (64 MiB) als Einsprungpunkt

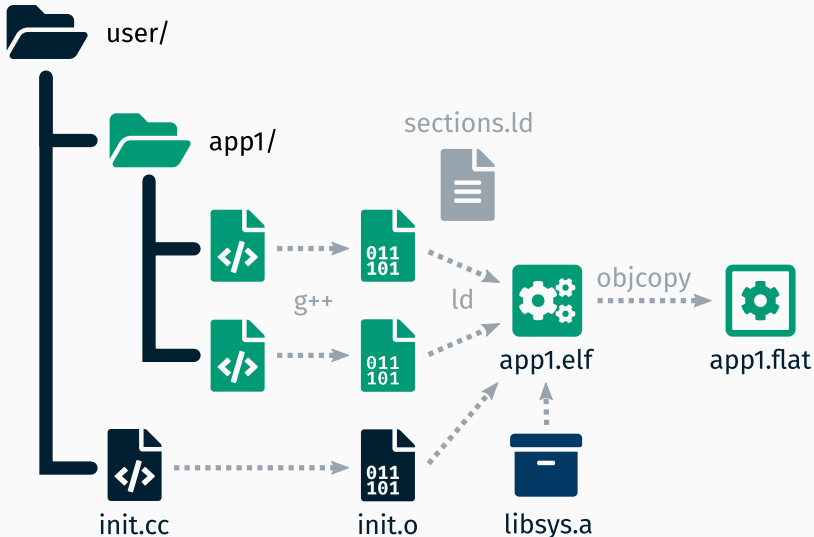


```
$ g++ $CXXFLAGS -Wl,T sections.ld -o app1.elf $LDFLAGS init.o [...]
```

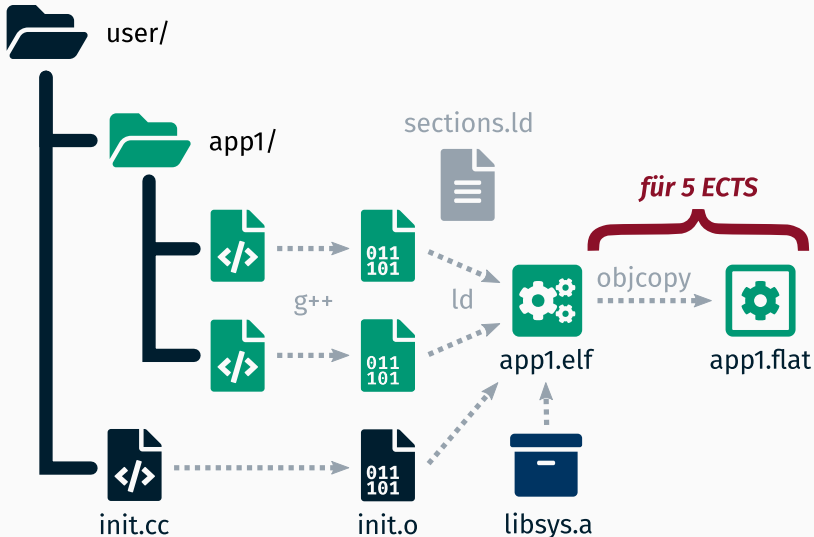




```
$ objcopy -O binary --set-section-flags \
    .bss=alloc,load,contents app1.elf app1.flat
```



```
$ objcopy -O binary --set-section-flags \
    .bss=alloc,load,contents app1.elf app1.flat
```



- `crti.o` und `crtn.o` für C Funktionsprologe
 - für Initialisierungs- (`_init`) und Beendigungsroutine (`_fini`)
(siehe `compiler/crti.asm` sowie `compiler/crtn.asm`)

C/C++ Laufzeitumgebung

- `crti.o` und `crtn.o` für C Funktionsprologe
 - für Initialisierungs- (`_init`) und Beendigungsroutine (`_fini`)
(siehe `compiler/crti.asm` sowie `compiler/crtn.asm`)
- `init.o` setzt die Laufzeitumgebung auf
 1. Einsprungspunkt ist `void start()`
 2. Ausführung der C Startup Initialisierungsroutinen
(`__preinit_array`, `_init()` und `__init_array`, siehe `compiler/libc.cc`)
 3. Übergabe an `main` der **App**
 4. bei Rückkehr Deinitialisierung
(`__fini_array`, `_fini()` und Endlosschleife)
 - Bereitstellung der für C++ benötigten Funktionen
(Dummies für `__cxa_pure_virtual()` und `__cxa_atexit()`, siehe `compiler/libcxx.cc`)

C/C++ Laufzeitumgebung

- `crti.o` und `crtn.o` für C Funktionsprologe
 - für Initialisierungs- (`_init`) und Beendigungsroutine (`_fini`)
(siehe `compiler/crti.asm` sowie `compiler/crtn.asm`)
- `init.o` setzt die Laufzeitumgebung auf
 1. Einsprungspunkt ist `void start()`
 2. Ausführung der C Startup Initialisierungsroutinen
(`__preinit_array`, `_init()` und `__init_array`, siehe `compiler/libc.cc`)
 3. Übergabe an `main` der App
 4. bei Rückkehr Deinitialisierung
(`__fini_array`, `_fini()` und Endlosschleife)
 - Bereitstellung der für C++ benötigten Funktionen
(Dummies für `__cxa_pure_virtual()` und `__cxa_atexit()`, siehe `compiler/libcxx.cc`)

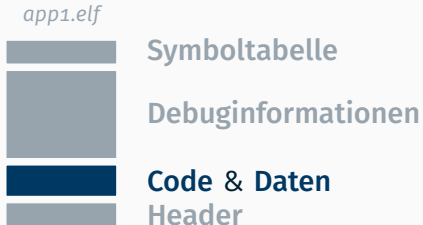


Beim Linken muss `init.o` die erste Objektdatei sein, damit `start()` die erste Funktion (am Einsprungspunkt an Adresse `0x400 0000`) ist!

Flat Binaries und Image Builder *(für 5 ECTS)*

Erstellung einer flachen Binärdatei

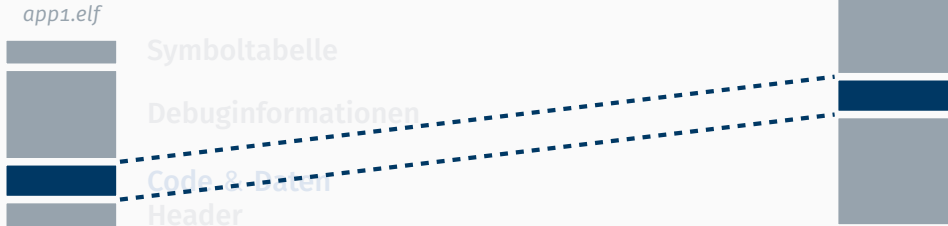
Man nehme eine ELF-Datei



Erstellung einer flachen Binärdatei

Man nehme eine ELF-Datei

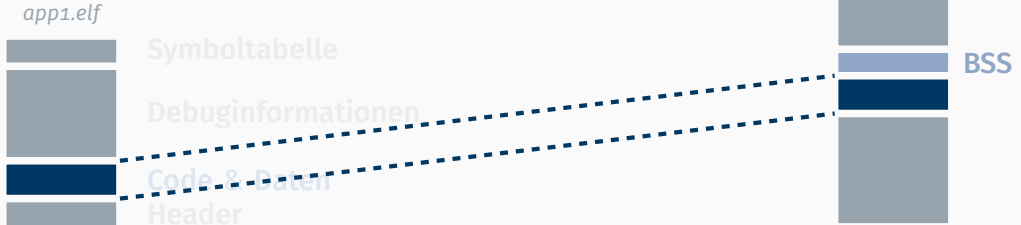
physikalischer Speicher



Erstellung einer flachen Binärdatei

Man nehme eine ELF-Datei

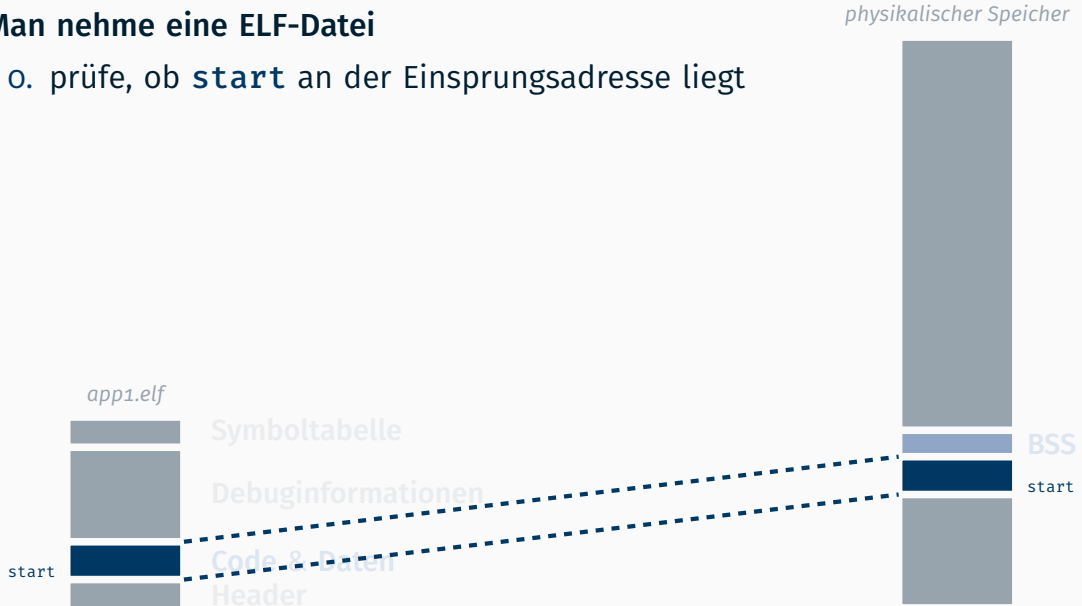
physikalischer Speicher



Erstellung einer flachen Binärdatei

Man nehme eine ELF-Datei

o. prüfe, ob **start** an der Einsprungsadresse liegt

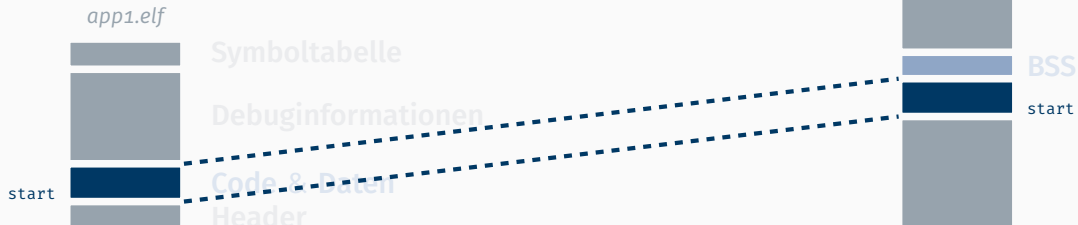


Erstellung einer flachen Binärdatei

Man nehme eine ELF-Datei

- o. prüfe, ob **start** an der Einsprungsadresse liegt

```
$ nm app1.elf | grep -q "4000000 T start" || echo "Fehler"
```

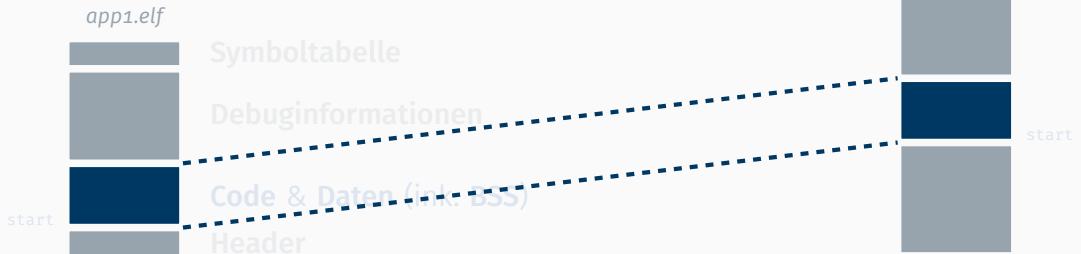


Erstellung einer flachen Binärdatei

Man nehme eine ELF-Datei

0. prüfe, ob **start** an der Einsprungsadresse liegt

```
$ nm app1.elf | grep -q "4000000 T start" || echo "Fehler"
```
1. integriere **BSS** in die **Datensektion**



Erstellung einer flachen Binärdatei

Man nehme eine ELF-Datei

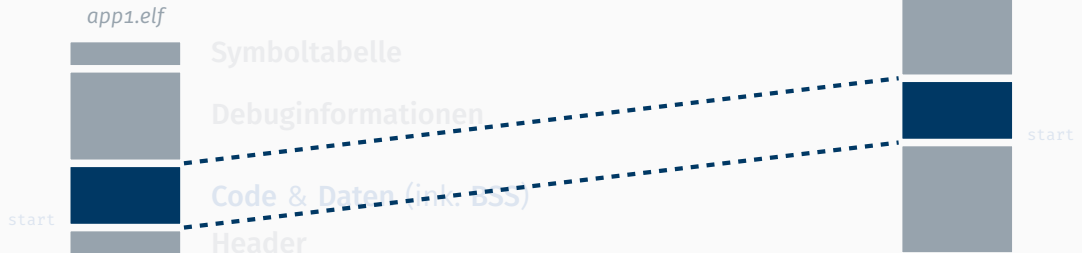
0. prüfe, ob **start** an der Einsprungsadresse liegt

```
$ nm app1.elf | grep -q "4000000 T start" || echo "Fehler"
```

1. integriere **BSS** in die **Datensektion**

```
objcopy Argument: --set-section-flags .bss=alloc,load,contents
```

physikalischer Speicher

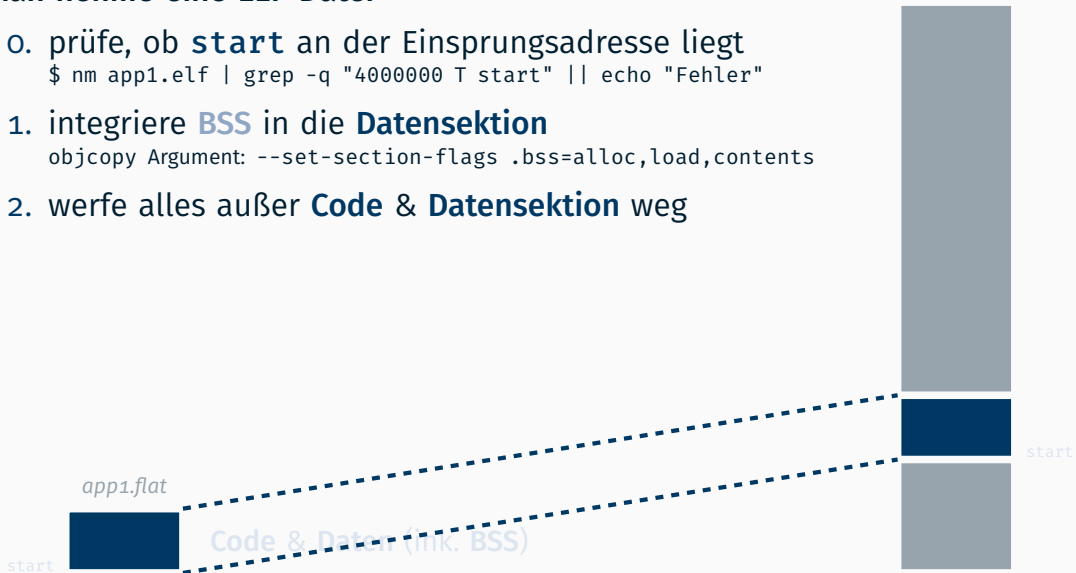


Erstellung einer flachen Binärdatei

Man nehme eine ELF-Datei

0. prüfe, ob **start** an der Einsprungsadresse liegt
`$ nm app1.elf | grep -q "4000000 T start" || echo "Fehler"`
1. integriere **BSS** in die **Datensektion**
objcopy Argument: `--set-section-flags .bss=alloc,load,contents`
2. werfe alles außer **Code & Datensektion** weg

physikalischer Speicher



Erstellung einer flachen Binärdatei

Man nehme eine ELF-Datei

0. prüfe, ob **start** an der Einsprungsadresse liegt

```
$ nm app1.elf | grep -q "4000000 T start" || echo "Fehler"
```

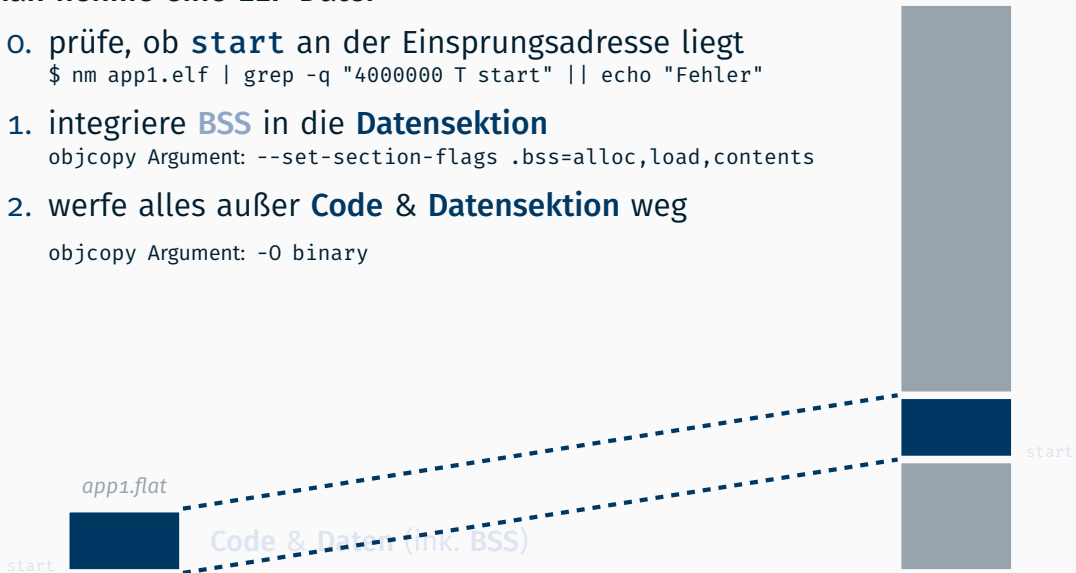
1. integriere **BSS** in die **Datensektion**

```
objcopy Argument: --set-section-flags .bss=alloc,load,contents
```

2. werfe alles außer **Code & Datensektion** weg

```
objcopy Argument: -O binary
```

physikalischer Speicher



Erstellung einer flachen Binärdatei

Man nehme eine ELF-Datei

0. prüfe, ob **start** an der Einsprungsadresse liegt

```
$ nm app1.elf | grep -q "4000000 T start" || echo "Fehler"
```

1. integriere **BSS** in die **Datensektion**

```
objcopy Argument: --set-section-flags .bss=alloc,load,contents
```

2. werfe alles außer **Code & Datensektion** weg

```
objcopy Argument: -O binary
```

Zack fertig: **Flat Binary**



Erstellung einer flachen Binärdatei

Man nehme eine ELF-Datei

0. prüfe, ob **start** an der Einsprungsadresse liegt

```
$ nm app1.elf | grep -q "4000000 T start" || echo "Fehler"
```

1. integriere **BSS** in die **Datensektion**

```
objcopy Argument: --set-section-flags .bss=alloc,load,contents
```

2. werfe alles außer **Code & Datensektion** weg

```
objcopy Argument: -O binary
```

Zack fertig: **Flat Binary**

→ initiales Speicherabbild einer Anwendung ohne Metainformationen



Anwendungsabbildarchiv als Boot Module



appx.flat



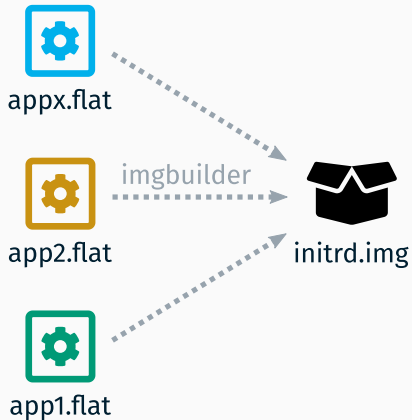
app2.flat



app1.flat

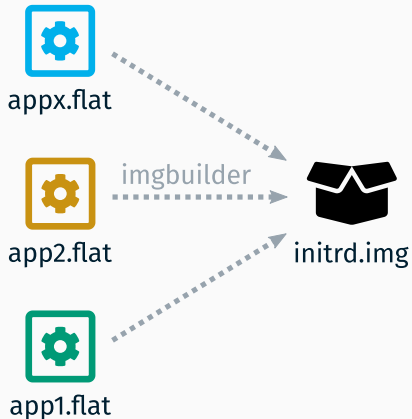
Anwendungsabbildarchiv als Boot Module

```
$ ./imgbuilder app1.flat app2.flat appx.flat > initrd.img
```



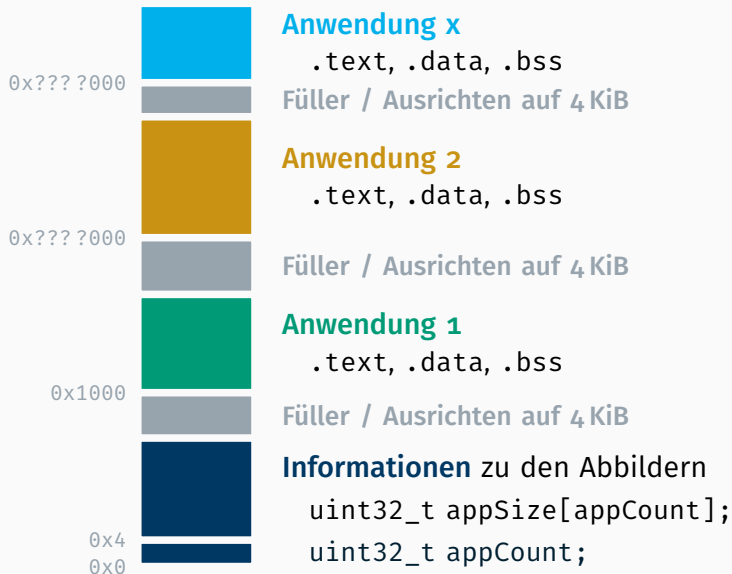
Anwendungsabbildarchiv als Boot Module

```
$ ./imgbuilder app1.flat app2.flat appx.flat > initrd.img
```



→ Der Quelltext des *Image Builders* ist in der Aufgabenstellung verlinkt.

Image Builder Format



ELF und TAR (*für 7.5 ECTS*)

Executable and Linking Format

- 1993 vom *Tool Interface Standard (TIS) Committee* spezifiziert

Executable and Linking Format

- 1993 vom *Tool Interface Standard (TIS)* Committee) spezifiziert
- Nachfolger von *a.out* (*Assembler output*, ab 1971)

Executable and Linking Format

- 1993 vom *Tool Interface Standard (TIS)* Committee) spezifiziert
- Nachfolger von *a.out* (*Assembler output*, ab 1971) und *coff* (*Common Object File Format*, ab 1983)

Executable and Linking Format

- 1993 vom *Tool Interface Standard (TIS)* Committee) spezifiziert
- Nachfolger von *a.out* (*Assembler output*, ab 1971) und *coff* (*Common Object File Format*, ab 1983)
- Standardformat auf unixoiden Systemen
- inzwischen plattformübergreifender Einsatz

Executable and Linking Format

- 1993 vom *Tool Interface Standard (TIS)* Committee) spezifiziert
- Nachfolger von *a.out* (*Assembler output*, ab 1971) und *coff* (*Common Object File Format*, ab 1983)
- Standardformat auf unixoiden Systemen
- inzwischen plattformübergreifender Einsatz
- vielfältige Einsatzgebiete
 - relocatable** Objektdateien (.o)
 - executable** (statische) Programme
 - dynamic** dynamische Programme und [geteilte] Bibliotheken (.so)

Executable and Linking Format

- 1993 vom *Tool Interface Standard (TIS)* Committee) spezifiziert
- Nachfolger von *a.out* (*Assembler output*, ab 1971) und *coff* (*Common Object File Format*, ab 1983)
- Standardformat auf unixoiden Systemen
- inzwischen plattformübergreifender Einsatz
- vielfältige Einsatzgebiete
 - relocatable** Objektdateien (.o)
 - executable** (statische) Programme
 - dynamic** dynamische Programme und [geteilte] Bibliotheken (.so)
- durch Sektionen flexibel erweiterbar

Executable and Linking Format

- 1993 vom *Tool Interface Standard (TIS)* Committee) spezifiziert
- Nachfolger von *a.out* (*Assembler output*, ab 1971) und *coff* (*Common Object File Format*, ab 1983)
- Standardformat auf unixoiden Systemen
- inzwischen plattformübergreifender Einsatz
- vielfältige Einsatzgebiete
 - relocatable** Objektdateien (.o)
 - executable** (statische) Programme
 - dynamic** dynamische Programme und [geteilte] Bibliotheken (.so)
- durch Sektionen flexibel erweiterbar
- bei Bedarf Einbettung von *DWARF* Debuginformationen

app1.elf



Symboltabelle



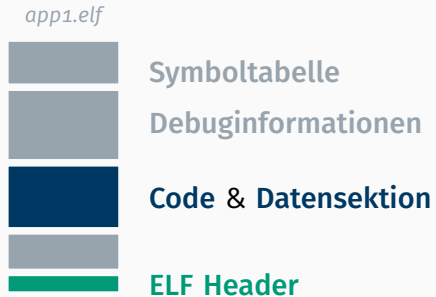
Debuginformationen



Code & Datensektion



Header



ELF Header

- Erkennung durch Wert { 0x7f, 'E', 'L', 'F' }
- gibt u.a. Architektur, ABI und Typ (*Executable*, *Relocatable*, *Shared*) an

app1.elf



Symboltabelle



Debuginformationen



Code & Datensektion



ELF Header



ELF Header

- Erkennung durch Wert { 0x7f, 'E', 'L', 'F' }
- gibt u.a. Architektur, ABI und Typ (**Executable**, *Relocatable*, *Shared*) an
- beinhaltet Einsprungsadresse und Position sowie Größe von **Program** & *Section Header Table*



Program Header Table

- Beschreibt die einzelnen Programmsegmente



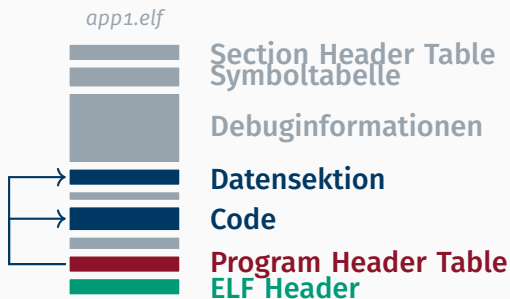
Program Header Table

- Beschreibt die einzelnen Programmsegmente
- jeweils mit Position & Größe in ELF-Datei und im Zielspeicher



Program Header Table

- Beschreibt die einzelnen Programmsegmente
- jeweils mit Position & Größe in ELF-Datei und im Zielspeicher
- verschiedene Typen: **LOAD**, **DYNAMIC**, **INTERP**, ...
- und Attribute wie les-, schreib- und ausführbar



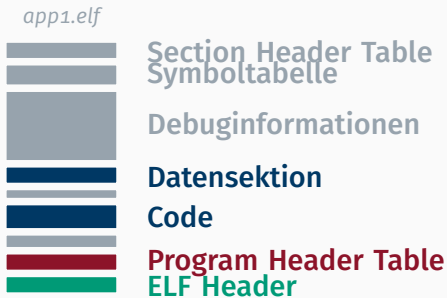
```
$ readelf -l -W user/app1/.build/app1.elf
```

```
Elf file type is EXEC (Executable file)
```

```
Entry point 0x4000000
```

```
There are 3 program headers, starting at offset 64
```

Type	Offset	VirtAdr	PhysAddr	FileSiz	MemSiz	Flg	Align
LOAD	0x001000	0x04000000	0x04000000	0x000aa0	0x000aa0	R E	0x1000
LOAD	0x002000	0x04001000	0x04001000	0x003720	0x006369	RW	0x1000
GNU_STACK	0x000000	0x00000000	0x00000000	0x000000	0x000000	RWE	0x10



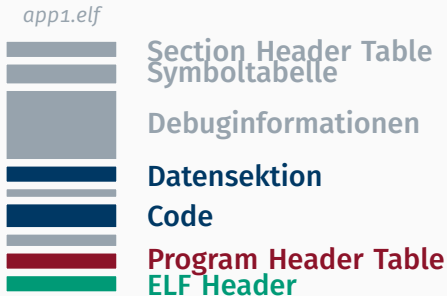
```
$ readelf -l -W user/app1/.build/app1.elf
```

```
Elf file type is EXEC (Executable file)
```

```
Entry point 0x4000000
```

```
There are 3 program headers, starting at offset 64
```

Type	Offset	VirtAdr	PhysAddr	FileSiz	MemSiz	Flg	Align
LOAD	0x001000	0x04000000	0x04000000	0x000aa0	0x000aa0	R E	0x1000
LOAD	0x002000	0x04001000	0x04001000	0x003720	0x006369	RW	0x1000
GNU_STACK	0x000000	0x00000000	0x00000000	0x000000	0x000000	RWE	0x10



```
$ readelf -l -W user/app1/.build/app1.elf
```

```
Elf file type is EXEC (Executable file)
```

```
Entry point 0x4000000
```

```
There are 3 program headers, starting at offset 64
```

Type	Offset	VirtAdr	PhysAddr	FileSiz	MemSiz	Flg	Align
LOAD	0x001000	0x04000000	0x04000000	0x000aa0	0x000aa0	R E	0x1000
LOAD	0x002000	0x04001000	0x04001000	0x003720	0x006369	RW	0x1000
GNU_STACK	0x000000	0x00000000	0x00000000	0x000000	0x000000	RWE	0x10



Die Vorgabe enthält unter `utils/` bereits das Grundgerüst (Parser) für einen einfachen Loader.



Die Vorgabe enthält unter `utils/` bereits das Grundgerüst (Parser) für einen einfachen Loader. Dieser soll

- nur LOAD von statische Binärdateien (ohne Relok.) unterstützen



Die Vorgabe enthält unter `utils/` bereits das Grundgerüst (Parser) für einen einfachen Loader. Dieser soll

- nur LOAD von statische Binärdateien (ohne Relok.) unterstützen
- Berechtigungen (Writeable / Executable) berücksichtigen



Die Vorgabe enthält unter `utils/` bereits das Grundgerüst (Parser) für einen einfachen Loader. Dieser soll

- nur LOAD von statische Binärdateien (ohne Relok.) unterstützen
- Berechtigungen (Writeable / Executable) berücksichtigen
- (ELF) Quelldaten in neue (Userspace) Zielseiten **kopieren**



Die Vorgabe enthält unter `utils/` bereits das Grundgerüst (Parser) für einen einfachen Loader. Dieser soll

- nur LOAD von statische Binärdateien (ohne Relok.) unterstützen
- Berechtigungen (Writeable / Executable) berücksichtigen
- (ELF) Quelldaten in neue (Userspace) Zielseiten **kopieren**
 - *optional* unter bestimmten Bedingungen auch Quellseite **einblenden**



Bandarchivierer (TAPE ARCHIVER)

- 1979 eingeführte Archivdatei

Bandarchivierer (TAPE ARCHIVER)

- 1979 eingeführte Archivdatei
 - *klebt* Dateien (unkomprimiert/unmodifiziert) aneinander
 - ausgerichtet an 512 Byte Grenzen
 - jeweils mit Kopfeintrag (beinhaltet Dateiname, Größe, Berechtigungen)

Bandarchivierer (TAPE ARCHIVER)

- 1979 eingeführte Archivdatei
 - *klebt* Dateien (unkomprimiert/unmodifiziert) aneinander
 - ausgerichtet an 512 Byte Grenzen
 - jeweils mit Kopfeintrag (beinhaltet Dateiname, Größe, Berechtigungen)
 - aber kein Index über alle Inhalte!

Bandarchivierer (TAPE ARCHIVER)

- 1979 eingeführte Archivdatei
 - *klebt* Dateien (unkomprimiert/unmodifiziert) aneinander
 - ausgerichtet an 512 Byte Grenzen
 - jeweils mit Kopfeintrag (beinhaltet Dateiname, Größe, Berechtigungen)
 - aber kein Index über alle Inhalte!
- für Bandlaufwerke (= sequentielle Ein-/Ausgabe) gedacht

Bandarchivierer (TAPE ARCHIVER)

- 1979 eingeführte Archivdatei
 - *klebt* Dateien (unkomprimiert/unmodifiziert) aneinander
 - ausgerichtet an 512 Byte Grenzen
 - jeweils mit Kopfeintrag (beinhaltet Dateiname, Größe, Berechtigungen)
 - aber kein Index über alle Inhalte!
- für Bandlaufwerke (= sequentielle Ein-/Ausgabe) gedacht
- 1988 POSIX-Standardisierung und Einführung der *UStar* (= *Unix Standard*) Erweiterung (mehr Attribute, längere Dateinamen)
 - zusätzliche Informationen im Kopfeintrag
 - Format bleibt aber weiterhin kompatibel

Bandarchivierer (TAPE ARCHIVER)

- 1979 eingeführte Archivdatei
 - *klebt* Dateien (unkomprimiert/unmodifiziert) aneinander
 - ausgerichtet an 512 Byte Grenzen
 - jeweils mit Kopfeintrag (beinhaltet Dateiname, Größe, Berechtigungen)
 - aber kein Index über alle Inhalte!
- für Bandlaufwerke (= sequentielle Ein-/Ausgabe) gedacht
- 1988 POSIX-Standardisierung und Einführung der *UStar* (= *Unix Standard*) Erweiterung (mehr Attribute, längere Dateinamen)
 - zusätzliche Informationen im Kopfeintrag
 - Format bleibt aber weiterhin kompatibel
- 1997 *pax* (= *portable archive formats*) Erweiterung
 - in großen Teilen kompatibel zu *UStar*
 - zur Befriedung der *Tar-Wars* (mit dem konkurrierenden *cpio*)

Bandarchivierer (TAPE ARCHIVER) Verwendung



appx.elf



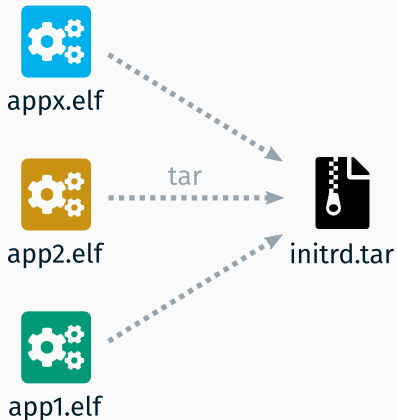
app2.elf



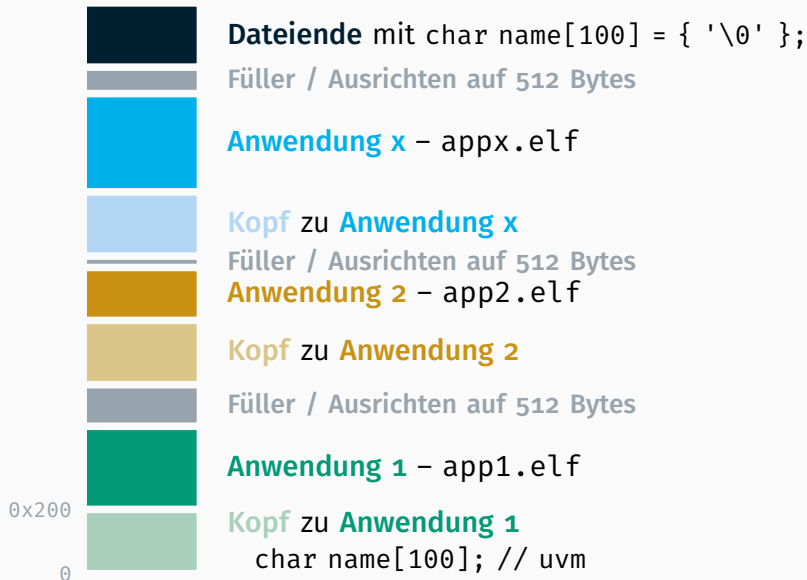
app1.elf

Bandarchivierer (TAPE ARCHIVER) Verwendung

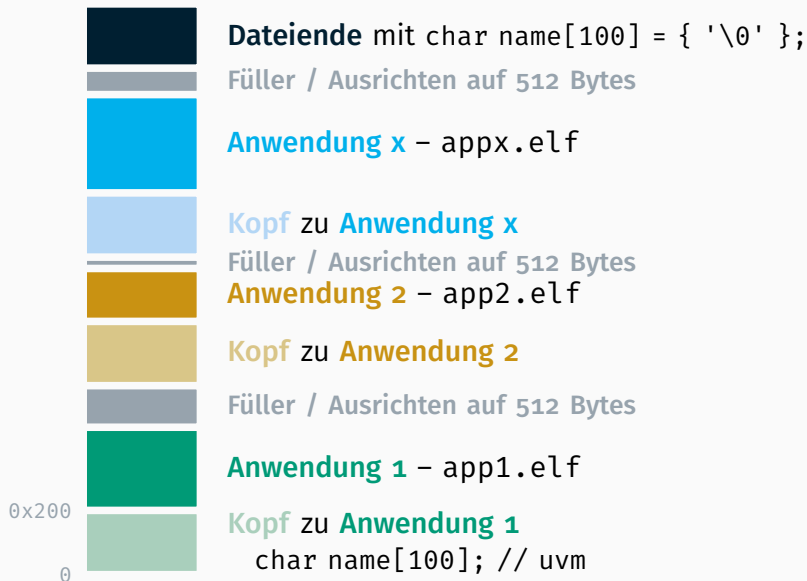
```
$ tar -cf initrd.tar app1.elf app2.elf appx.elf
```



TAR Format



TAR Format



→ (rudimentärer) TAR-Parser bereits in Vorgabe (unter `utils/`) enthalten!

TAPE ARCHIVE



ANGE ALBERTINI

<http://www.corkami.com>



```
$ tar -xOf hello.tar hello.txt
Hello World!
```

```
  0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000: .h .e .l .l .o . . .t .x .t
0060:      .0 .0 .0 .0 .6 .4 .4 00 .0 .0 .0 .0
0070: .7 .6 .4 00 .0 .0 .0 .1 .0 .4 .0 00 .0 .0 .0 .0
0080: .0 .0 .0 .0 .0 .1 .5 00 .1 .2 .4 .2 .0 .0 .1 .0
0090: .5 .3 .2 00 .0 .1 .4 .6 .3 .6 00 20 .0
0100:  .u .s .t .a .r 00 .0 .0 .A .n .g .e
0120:                                     .A .d .m .i .n .i .s
0030: .t .r .a .t .o .r .s
-----
0200: .H .e .l .l .o 20 .W .o .r .l .d .! 0A
2800: ]
```

FILE HEADER

CONTENTS

FIELDS

VALUES

file name	hello.txt
file mode	0000644
owner user ID	0000764
group user ID	0001040
file size	0000013
timestamp	2014-10-16 20:41
checksum	014636 \0\x20
type flag	00 REGTYPE
magic	ustar\x00
version	"00"
owner user name	Ange
owner group name	Administrators
contents	Hello World!\n

TAR WAS INITIALLY DESIGNED FOR TAPE DRIVES, IN 1979:

- NO COMPRESSION, BLOCK ALIGNED

- NUMERIC VALUES ARE STORED IN OCTAL, ENCODED IN ASCII

TAR IS OFTEN COMBINED WITH GZIP, BZIP2 OR LZMA.

THE TAR FORMAT EVOLVED:

THIS EXAMPLE IS A "USTAR" FILE, AS DEFINED IN 1988

Unsere Makefile unterstützt bereits ein Bootmodul, dessen Pfad in der Variable INITRD definiert werden muss.

Unsere Makefile unterstützt bereits ein Bootmodul, dessen Pfad in der Variable INITRD definiert werden muss.

Beim Target netboot wird diese Datei nach /proj/i4stubs/student/LOGIN/initrd.img kopiert und automatisch vom Bootloader geladen

(d.h. die Angabe in Multiboot::Module::getCommandLine() ist nicht sonderlich aussagekräftig, sondern beim Benutzer immer gleich)

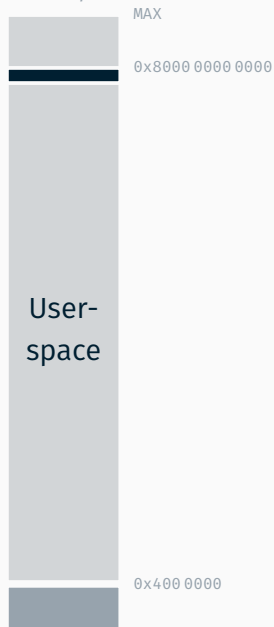
Dynamisch wachsender Stapel

Dynamisch wachsender Anwendungsstapel

Ablauf:

- initial z.B. 4 KiB Userspace Stack

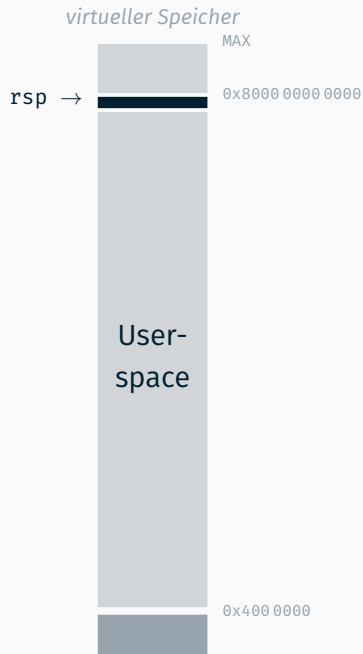
virtueller Speicher



Dynamisch wachsender Anwendungsstapel

Ablauf:

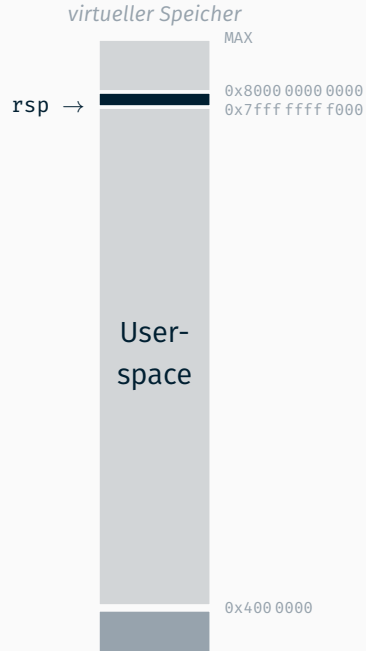
- initial z.B. 4 KiB Userspace Stack



Dynamisch wachsender Anwendungsstapel

Ablauf:

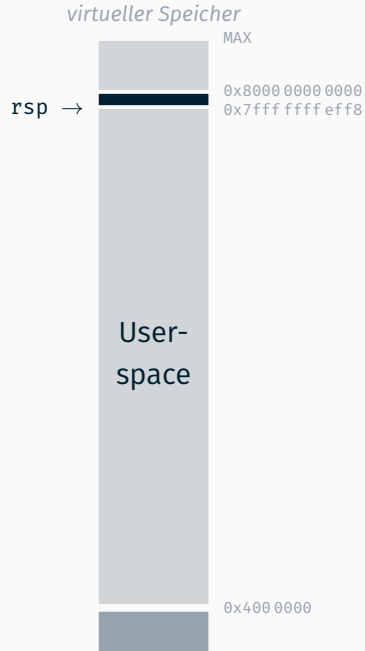
- initial z.B. 4 KiB Userspace Stack
- sobald Stack voll



Dynamisch wachsender Anwendungsstapel

Ablauf:

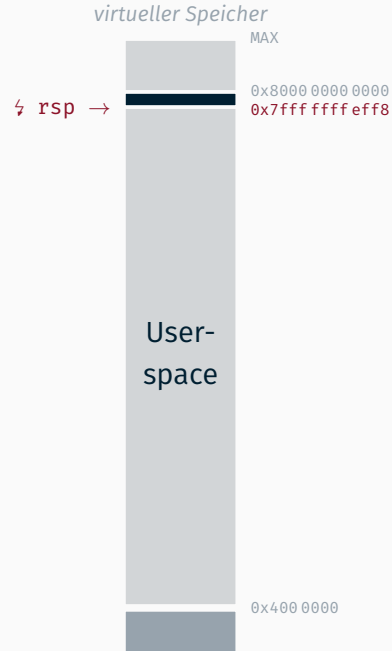
- initial z.B. 4 KiB Userspace Stack
- sobald Stack voll



Dynamisch wachsender Anwendungsstapel

Ablauf:

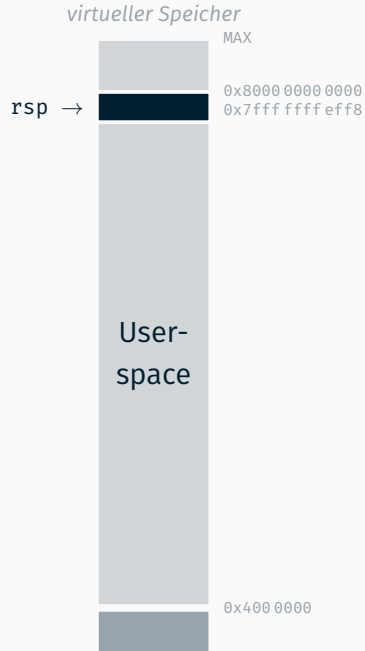
- initial z.B. 4 KiB Userspace Stack
- sobald Stack voll → **Seitenfehler**



Dynamisch wachsender Anwendungsstapel

Ablauf:

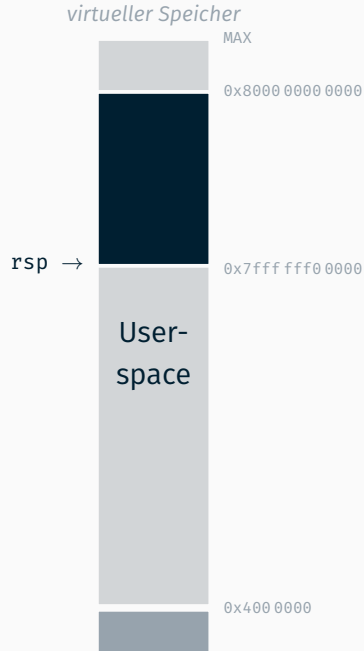
- initial z.B. 4 KiB Userspace Stack
- sobald Stack voll → **Seitenfehler**
- Behandlungsroutine allokiert mehr Speicher (und setzt Ausführung fort)



Dynamisch wachsender Anwendungsstapel

Ablauf:

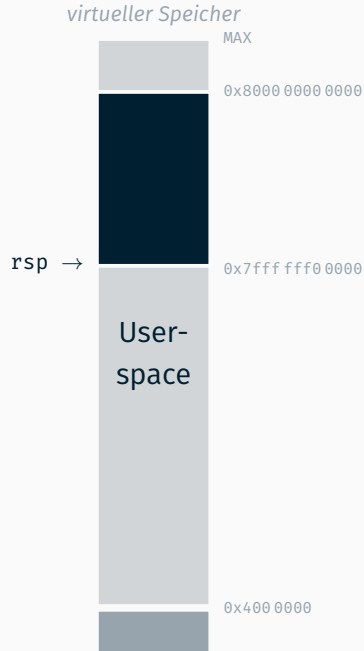
- initial z.B. 4 KiB Userspace Stack
- sobald Stack voll → **Seitenfehler**
- Behandlungsroutine allokiert mehr Speicher (und setzt Ausführung fort)
- bis zu einem vorgegebenen Maximum



Ablauf:

- initial z.B. 4 KiB Userspace Stack
- sobald Stack voll → **Seitenfehler**
- Behandlungsroutine allokiert mehr Speicher (und setzt Ausführung fort)
- bis zu einem vorgegebenen Maximum

Umsetzung mittels neuem `pagefault_handler`

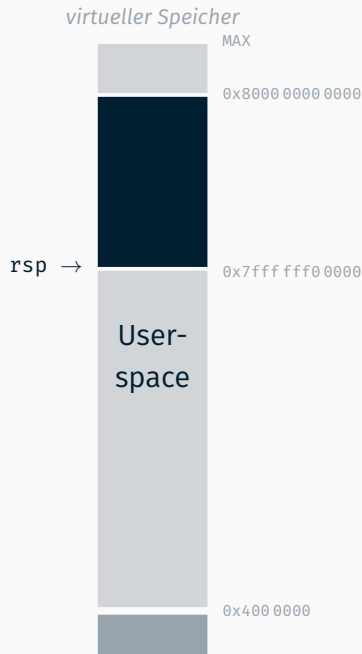


Ablauf:

- initial z.B. 4 KiB Userspace Stack
- sobald Stack voll → **Seitenfehler**
- Behandlungsroutine allokiert mehr Speicher (und setzt Ausführung fort)
- bis zu einem vorgegebenen Maximum

Umsetzung mittels neuem pagefault_handler

- Verwendung von `cr2` und `error_code`



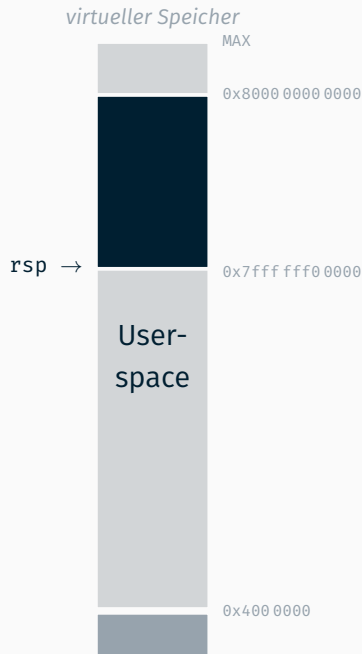
Dynamisch wachsender Anwendungsstapel

Ablauf:

- initial z.B. 4 KiB Userspace Stack
- sobald Stack voll → **Seitenfehler**
- Behandlungsroutine allokiert mehr Speicher (und setzt Ausführung fort)
- bis zu einem vorgegebenen Maximum

Umsetzung mittels neuem pagefault_handler

- Verwendung von `cr2` und `error_code`
- bei Änderung des aktiven Mappings an das Spülen des TLBs (z.B. via `invlpg`) denken!



Dynamisch wachsender Anwendungsstapel

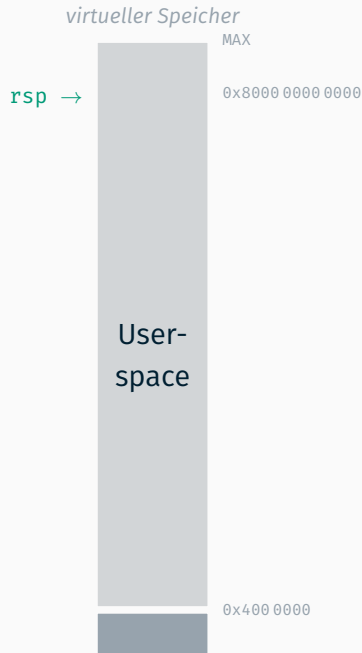
Ablauf:

- initial z.B. 4 KiB Userspace Stack
- sobald Stack voll → **Seitenfehler**
- Behandlungsroutine allokiert mehr Speicher (und setzt Ausführung fort)
- bis zu einem vorgegebenen Maximum

Umsetzung mittels neuem pagefault_handler

- Verwendung von `cr2` und `error_code`
- bei Änderung des aktiven Mappings an das Spülen des TLBs (z.B. via `invlpg`) denken!

Protipp: Initial gar keinen Stack allokierten!





Der Stack muss bei Funktionsaufrufen (d.h. vor `call`) an 16 Byte ausgerichtet sein!



Der Stack muss bei Funktionsaufrufen (d.h. vor `call`) an 16 Byte ausgerichtet sein!

- gemäß System V ABI 3.2.2



Der Stack muss bei Funktionsaufrufen (d.h. vor `call`) an 16 Byte ausgerichtet sein!

- gemäß System V ABI 3.2.2
- für FPU & SSE Instruktionen wichtig



Der Stack muss bei Funktionsaufrufen (d.h. vor `call`) an 16 Byte ausgerichtet sein!

- gemäß System V ABI 3.2.2
- für FPU & SSE Instruktionen wichtig
- einfache Überprüfung mittels `cmpxchg16b` Instruktion:

```
__int128 x = 0;  
__sync_val_compare_and_swap(&x, 0, 1);
```

und mit zusätzlichem GCC Parameter `-mcx16` übersetzen
→ *Protection Fault* bei unausgerichtetem Stack!

Optional: Fließkommazahlen

Geschichte der Fließkommazahlen auf dem x86

Gleitkomma-Operationen beim **x86**

Geschichte der Fließkommazahlen auf dem x86

Gleitkomma-Operationen beim **x86** entweder in Software

Geschichte der Fließkommazahlen auf dem x86

Gleitkomma-Operationen beim **x86** entweder in Software oder

- Anfangs Koprozessor (*Intel 8087* → **x87**)

Geschichte der Fließkommazahlen auf dem x86

Gleitkomma-Operationen beim **x86** entweder in Software oder

- Anfangs Koprozessor (*Intel 8087* → **x87**)
- ab 486er Gleitkommaeinheit integriert in CPU

Geschichte der Fließkommazahlen auf dem x86

Gleitkomma-Operationen beim **x86** entweder in Software oder

- Anfangs Koprozessor (*Intel 8087* → **x87**)
- ab 486er Gleitkommaeinheit integriert in CPU
 - acht 80 bit Datenregister (16 bit Exponent + 64 bit Mantisse)
→ organisiert als Stack (st0 – st7)
 - zudem weitere Kontrollregister
 - Befehle für Stack- und Rechenoperationen
→ Parallele Ausführung, Fehler als Exceptions

Geschichte der Fließkommazahlen auf dem x86

Gleitkomma-Operationen beim **x86** entweder in Software oder

- Anfangs Koprozessor (*Intel 8087* → **x87**)
- ab 486er Gleitkommaeinheit integriert in CPU
 - acht 80 bit Datenregister (16 bit Exponent + 64 bit Mantisse)
→ organisiert als Stack (st0 – st7)
 - zudem weitere Kontrollregister
 - Befehle für Stack- und Rechenoperationen
→ Parallele Ausführung, Fehler als Exceptions
- später Erweiterungen *Multi Media Extension (MMX)*

Geschichte der Fließkommazahlen auf dem x86

Gleitkomma-Operationen beim **x86** entweder in Software oder

- Anfangs Koprozessor (*Intel 8087* → **x87**)
- ab 486er Gleitkommaeinheit integriert in CPU
 - acht 80 bit Datenregister (16 bit Exponent + 64 bit Mantisse)
→ organisiert als Stack (st0 – st7)
 - zudem weitere Kontrollregister
 - Befehle für Stack- und Rechenoperationen
→ Parallele Ausführung, Fehler als Exceptions
- später Erweiterungen *Multi Media Extension (MMX)*, *Streaming SIMD Extensions (SSE)* [1...4]

Geschichte der Fließkommazahlen auf dem x86

Gleitkomma-Operationen beim **x86** entweder in Software oder

- Anfangs Koprozessor (*Intel 8087* → **x87**)
- ab 486er Gleitkommaeinheit integriert in CPU
 - acht 80 bit Datenregister (16 bit Exponent + 64 bit Mantisse)
→ organisiert als Stack (st0 – st7)
 - zudem weitere Kontrollregister
 - Befehle für Stack- und Rechenoperationen
→ Parallele Ausführung, Fehler als Exceptions
- später Erweiterungen *Multi Media Extension (MMX)*, *Streaming SIMD Extensions (SSE)* [1...4] und *Advanced Vector Extensions (AVX)*

Geschichte der Fließkommazahlen auf dem x86

Gleitkomma-Operationen beim **x86** entweder in Software oder

- Anfangs Koprozessor (*Intel 8087* → **x87**)
- ab 486er Gleitkommaeinheit integriert in CPU
 - acht 80 bit Datenregister (16 bit Exponent + 64 bit Mantisse)
→ organisiert als Stack (st0 – st7)
 - zudem weitere Kontrollregister
 - Befehle für Stack- und Rechenoperationen
→ Parallele Ausführung, Fehler als Exceptions
- später Erweiterungen *Multi Media Extension (MMX)*, *Streaming SIMD Extensions (SSE)* [1...4] und *Advanced Vector Extensions (AVX)*
 - x64 unterstützen immer (mindestens) SSE2

Geschichte der Fließkommazahlen auf dem x86

Gleitkomma-Operationen beim **x86** entweder in Software oder

- Anfangs Koprozessor (*Intel 8087* → **x87**)
- ab 486er Gleitkommaeinheit integriert in CPU
 - acht 80 bit Datenregister (16 bit Exponent + 64 bit Mantisse)
→ organisiert als Stack (st0 – st7)
 - zudem weitere Kontrollregister
 - Befehle für Stack- und Rechenoperationen
→ Parallele Ausführung, Fehler als Exceptions
- später Erweiterungen *Multi Media Extension (MMX)*, *Streaming SIMD Extensions (SSE)* [1...4] und *Advanced Vector Extensions (AVX)*
 - x64 unterstützen immer (mindestens) SSE2
 - min. sechzehn 128 bit Register (xmm0 – xmm15)

Geschichte der Fließkommazahlen auf dem x86

Gleitkomma-Operationen beim **x86** entweder in Software oder

- Anfangs Koprozessor (*Intel 8087* → **x87**)
- ab 486er Gleitkommaeinheit integriert in CPU
 - acht 80 bit Datenregister (16 bit Exponent + 64 bit Mantisse)
→ organisiert als Stack (st0 – st7)
 - zudem weitere Kontrollregister
 - Befehle für Stack- und Rechenoperationen
→ Parallele Ausführung, Fehler als Exceptions
- später Erweiterungen *Multi Media Extension (MMX)*, *Streaming SIMD Extensions (SSE)* [1...4] und *Advanced Vector Extensions (AVX)*
 - x64 unterstützen immer (mindestens) SSE2
 - min. sechzehn 128 bit Register (xmm0 – xmm15)
 - Nutzung (nach System V ABI 3.2.3) auch für Parameterübergabe

Geschichte der Fließkommazahlen auf dem x86

Gleitkomma-Operationen beim **x86** entweder in Software oder

- Anfangs Koprozessor (*Intel 8087* → **x87**)
- ab 486er Gleitkommaeinheit integriert in CPU
 - acht 80 bit Datenregister (16 bit Exponent + 64 bit Mantisse)
→ organisiert als Stack (st0 – st7)
 - zudem weitere Kontrollregister
 - Befehle für Stack- und Rechenoperationen
→ Parallele Ausführung, Fehler als Exceptions
- später Erweiterungen *Multi Media Extension (MMX)*, *Streaming SIMD Extensions (SSE)* [1...4] und *Advanced Vector Extensions (AVX)*
 - x64 unterstützen immer (mindestens) SSE2
 - min. sechzehn 128 bit Register (xmm0 – xmm15)
 - Nutzung (nach System V ABI 3.2.3) auch für Parameterübergabe

→ Zustandssicherung beim Kontextwechsel notwendig!

Zustandssicherung

Sicherung der zusätzlichen Register beim Kontextwechsel ist teuer

Zustandssicherung

Sicherung der zusätzlichen Register beim Kontextwechsel ist teuer, aber viele Programme brauchen eigentlich gar keine FPU...

Zustandssicherung

Sicherung der zusätzlichen Register beim Kontextwechsel ist teuer, aber viele Programme brauchen eigentlich gar keine FPU...

- OS schaltet FPU standardmäßig aus und sichert beim Kontextwechsel nur die üblichen *general purpose* Register

Zustandssicherung

Sicherung der zusätzlichen Register beim Kontextwechsel ist teuer, aber viele Programme brauchen eigentlich gar keine FPU...

- OS schaltet FPU standardmäßig aus und sichert beim Kontextwechsel nur die üblichen *general purpose* Register
- falls ein Prozess die FPU benutzt, gibt es eine Exception

Sicherung der zusätzlichen Register beim Kontextwechsel ist teuer, aber viele Programme brauchen eigentlich gar keine FPU...

- OS schaltet FPU standardmäßig aus und sichert beim Kontextwechsel nur die üblichen *general purpose* Register
- falls ein Prozess die FPU benutzt, gibt es eine Exception
- OS aktiviert FPU, führt die Instruktion nochmal aus und kümmert sich bei diesem Programm von nun an auch um die FPU Register

Früher:

Sicherung der zusätzlichen Register beim Kontextwechsel ist teuer, aber viele Programme brauchen eigentlich gar keine FPU...

- OS schaltet FPU standardmäßig aus und sichert beim Kontextwechsel nur die üblichen *general purpose* Register
- falls ein Prozess die FPU benutzt, gibt es eine Exception
- OS aktiviert FPU, führt die Instruktion nochmal aus und kümmert sich bei diesem Programm von nun an auch um die FPU Register

Früher:

Sicherung der zusätzlichen Register beim Kontextwechsel ist teuer, aber viele Programme brauchen eigentlich gar keine FPU...

- OS schaltet FPU standardmäßig aus und sichert beim Kontextwechsel nur die üblichen *general purpose* Register
- falls ein Prozess die FPU benutzt, gibt es eine Exception
- OS aktiviert FPU, führt die Instruktion nochmal aus und kümmert sich bei diesem Programm von nun an auch um die FPU Register

Heute:

- Übersetzer nutzen standardmäßig auch xmm-Register

Früher:

Sicherung der zusätzlichen Register beim Kontextwechsel ist teuer, aber viele Programme brauchen eigentlich gar keine FPU...

- OS schaltet FPU standardmäßig aus und sichert beim Kontextwechsel nur die üblichen *general purpose* Register
- falls ein Prozess die FPU benutzt, gibt es eine Exception
- OS aktiviert FPU, führt die Instruktion nochmal aus und kümmert sich bei diesem Programm von nun an auch um die FPU Register

Heute:

- Übersetzer nutzen standardmäßig auch xmm-Register
- schnelle Spezialbefehle (wie `fxsave` & `fxrstor`)
 - hochoptimiert (84 & 44 Zyklen)
 - für alle FPU / MMX / SSE Register (512 bytes)

Zustandssicherung

Früher:

Sicherung der zusätzlichen Register beim Kontextwechsel ist teuer, aber viele Programme brauchen eigentlich gar keine FPU...

- OS schaltet FPU standardmäßig aus und sichert beim Kontextwechsel nur die üblichen *general purpose* Register
- falls ein Prozess die FPU benutzt, gibt es eine Exception
- OS aktiviert FPU, führt die Instruktion nochmal aus und kümmert sich bei diesem Programm von nun an auch um die FPU Register

Heute:

- Übersetzer nutzen standardmäßig auch xmm-Register
- schnelle Spezialbefehle (wie `fxsave` & `fxrstor`)
 - hochoptimiert (84 & 44 Zyklen)
 - für alle FPU / MMX / SSE Register (512 bytes)

→ standardmäßig im Userspace aktiviert

FPU-Umsetzung in STUBSml

- FPU / MMX / SSE bei Übersetzung von Userspace Apps aktivieren
(→ CXXFLAGS_NOFPU)

FPU-Umsetzung in STUBSml

- FPU / MMX / SSE bei Übersetzung von Userspace Apps aktivieren
(→ CXXFLAGS_NOFPU) – aber Kernel soll weiterhin ohne FPU laufen!

FPU-Umsetzung in STUBSml

- FPU / MMX / SSE bei Übersetzung von Userspace Apps aktivieren
(→ `CXXFLAGS_NOFPU`) – aber Kernel soll weiterhin ohne FPU laufen!
- beim Systemstart pro Kern FPU initialisieren (→ `FPU::init()`)

FPU-Umsetzung in STUBSml

- FPU / MMX / SSE bei Übersetzung von Userspace Apps aktivieren (→ CXXFLAGS_NOFPU) – aber Kernel soll weiterhin ohne FPU laufen!
- beim Systemstart pro Kern FPU initialisieren (→ FPU::init())
 1. Bits für *Software Emulation* (CR0_EM) und *Task Switched* (CR0_TS) in Kontrollregister 0 (cr0) löschen, *Monitor coprocessor* (CR0_MP) setzen

FPU-Umsetzung in STUBSML

- FPU / MMX / SSE bei Übersetzung von Userspace Apps aktivieren (→ CXXFLAGS_NOFPU) – aber Kernel soll weiterhin ohne FPU laufen!
- beim Systemstart pro Kern FPU initialisieren (→ `FPU::init()`)
 1. Bits für *Software Emulation* (CR0_EM) und *Task Switched* (CR0_TS) in Kontrollregister 0 (cr0) löschen, *Monitor coprocessor* (CR0_MP) setzen
 2. FPU selbst auf Standardwerte initialisieren (fninit), Statuswort (muss 0 sein) sowie Kontrollwort prüfen (muss 0x37f sein)

FPU-Umsetzung in STUBSML

- FPU / MMX / SSE bei Übersetzung von Userspace Apps aktivieren (→ CXXFLAGS_NOFPU) – aber Kernel soll weiterhin ohne FPU laufen!
- beim Systemstart pro Kern FPU initialisieren (→ FPU::init())
 1. Bits für *Software Emulation* (CR0_EM) und *Task Switched* (CR0_TS) in Kontrollregister 0 (cr0) löschen, *Monitor coprocessor* (CR0_MP) setzen
 2. FPU selbst auf Standardwerte initialisieren (fninit), Statuswort (muss 0 sein) sowie Kontrollwort prüfen (muss 0x37f sein)
 3. Bits für *Enable SSE Instructions* (CR4_OSFXSR) und *Enable SSE Exceptions* (CR4_OSXMMEXCPT) in Kontrollregister 4 (cr4) setzen

FPU-Umsetzung in STUBSML

- FPU / MMX / SSE bei Übersetzung von Userspace Apps aktivieren (→ CXXFLAGS_NOFPU) – aber Kernel soll weiterhin ohne FPU laufen!
- beim Systemstart pro Kern FPU initialisieren (→ FPU::init())
 1. Bits für *Software Emulation* (CR0_EM) und *Task Switched* (CR0_TS) in Kontrollregister 0 (cr0) löschen, *Monitor coprocessor* (CR0_MP) setzen
 2. FPU selbst auf Standardwerte initialisieren (fninit), Statuswort (muss 0 sein) sowie Kontrollwort prüfen (muss 0x37f sein)
 3. Bits für *Enable SSE Instructions* (CR4_OSFXSR) und *Enable SSE Exceptions* (CR4_OSXMMEXCPT) in Kontrollregister 4 (cr4) setzen
- pro Thread Speicher für FPU-Zustand reservieren (→ FPU::State)

FPU-Umsetzung in STUBSML

- FPU / MMX / SSE bei Übersetzung von Userspace Apps aktivieren (→ `CXXFLAGS_NOFPU`) – aber Kernel soll weiterhin ohne FPU laufen!
- beim Systemstart pro Kern FPU initialisieren (→ `FPU::init()`)
 1. Bits für *Software Emulation* (`CR0_EM`) und *Task Switched* (`CR0_TS`) in Kontrollregister 0 (`cr0`) löschen, *Monitor coprocessor* (`CR0_MP`) setzen
 2. FPU selbst auf Standardwerte initialisieren (`fninit`), Statuswort (muss 0 sein) sowie Kontrollwort prüfen (muss `0x37f` sein)
 3. Bits für *Enable SSE Instructions* (`CR4_OSFCSR`) und *Enable SSE Exceptions* (`CR4_OSXMMEXCPT`) in Kontrollregister 4 (`cr4`) setzen
- pro Thread Speicher für FPU-Zustand reservieren (→ `FPU::State`)
- beim Kontextwechsel aktuellen Zustand sichern (→ `FPU::save()`) und neuen laden (→ `FPU::State::restore()`)

FPU-Umsetzung in STUBSML

- FPU / MMX / SSE bei Übersetzung von Userspace Apps aktivieren (→ `CXXFLAGS_NOFPU`) – aber Kernel soll weiterhin ohne FPU laufen!
- beim Systemstart pro Kern FPU initialisieren (→ `FPU::init()`)
 1. Bits für *Software Emulation* (`CR0_EM`) und *Task Switched* (`CR0_TS`) in Kontrollregister 0 (`cr0`) löschen, *Monitor coprocessor* (`CR0_MP`) setzen
 2. FPU selbst auf Standardwerte initialisieren (`fninit`), Statuswort (muss 0 sein) sowie Kontrollwort prüfen (muss `0x37f` sein)
 3. Bits für *Enable SSE Instructions* (`CR4_OSFCSR`) und *Enable SSE Exceptions* (`CR4_OSXMMEXCPT`) in Kontrollregister 4 (`cr4`) setzen
- pro Thread Speicher für FPU-Zustand reservieren (→ `FPU::State`)
- beim Kontextwechsel aktuellen Zustand sichern (→ `FPU::save()`) und neuen laden (→ `FPU::State::restore()`)
- optional auch Unterstützung für `float` und `double` im Ausgabestrom der `libsys`

Fragen?