# Übung zu Betriebssystemtechnik

## Aufgabe 4: Trennung von Kern & Anwendungen

06. Juni 2023

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

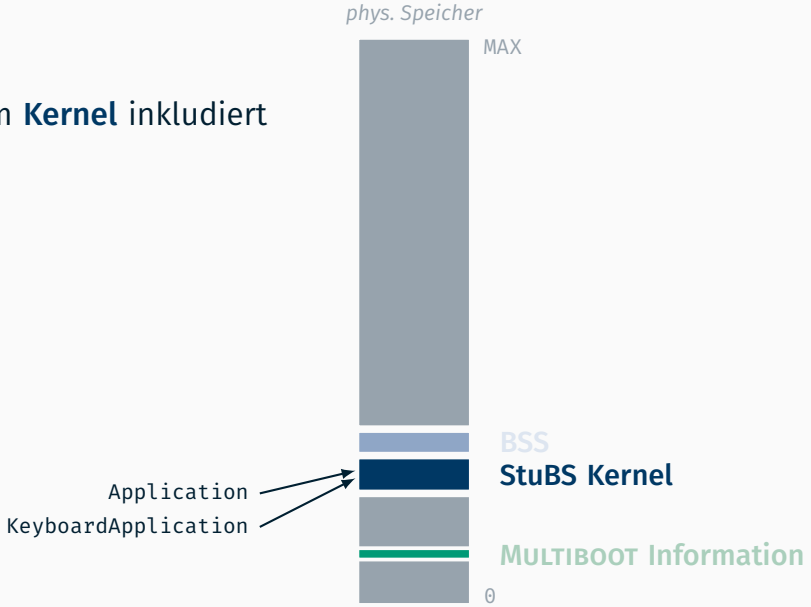
TECHNISCHE FAKULTÄT

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

# Bestandsaufnahme

**Bisher:**

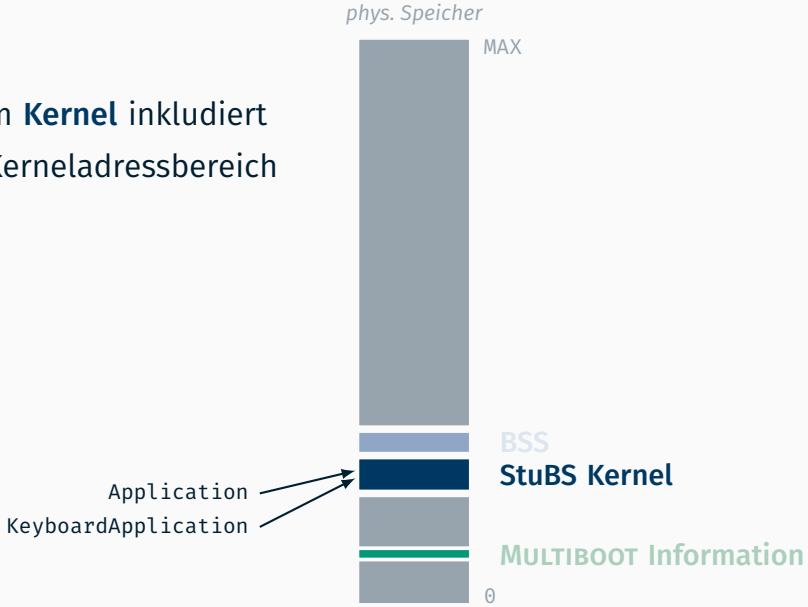
- Anwendungen im **Kernel** inkludiert



# Bestandsaufnahme

**Bisher:**

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



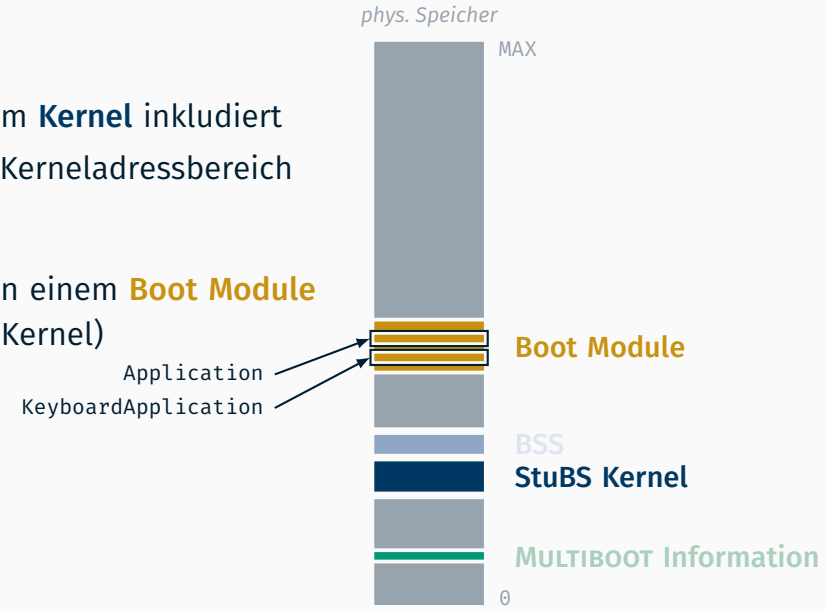
# Ziel dieser Aufgabe

**Bisher:**

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

**Ab dieser Aufgabe:**

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



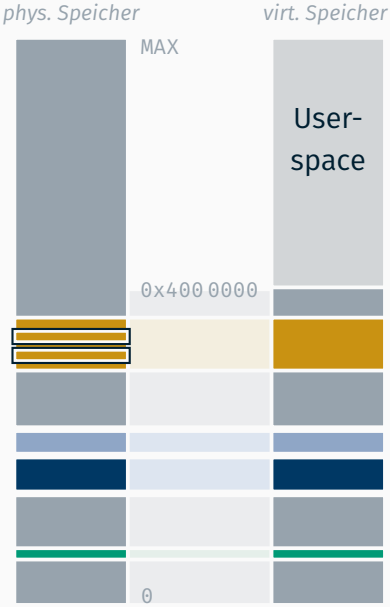
# Ziel dieser Aufgabe

**Bisher:**

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

**Ab dieser Aufgabe:**

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



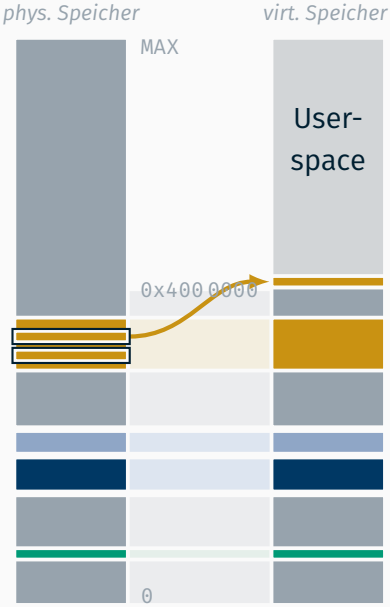
# Ziel dieser Aufgabe

**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`



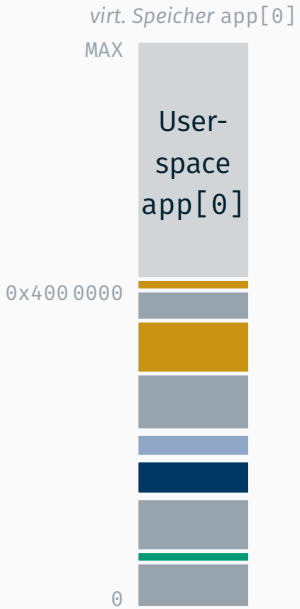
# Ziel dieser Aufgabe

**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)



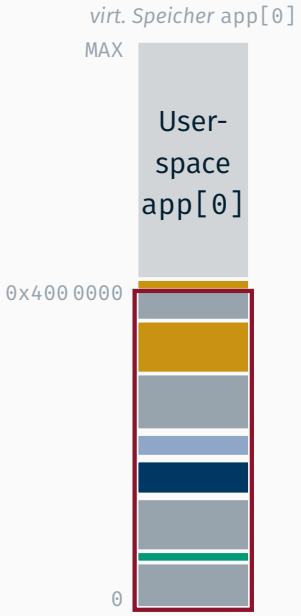
# Ziel dieser Aufgabe

**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

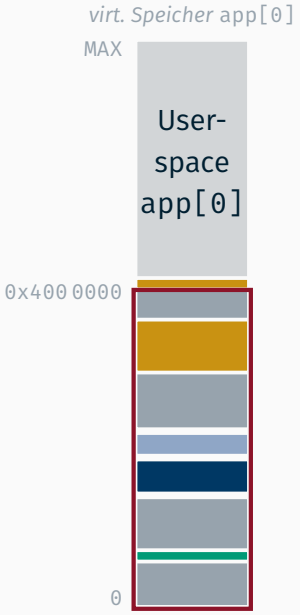
**Bisher:**

- 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

`libsys/` Unterstützung für Anwendungen

`user/` enthält **mehrere** Anwendungen
Jede Anwendung wird gegen `libsys` gelinkt
und zu einer eigenen Binärdatei kompiliert

# Organisation der Dateien

**Bisher:**

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

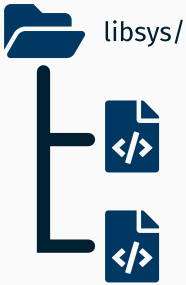
**Nun Aufteilung des Codes in unterschiedliche Verzeichnisse:**

`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

# libsys



# libsys


- Systemaufrufstümpfe (aus Aufgabe 2)

# libsys

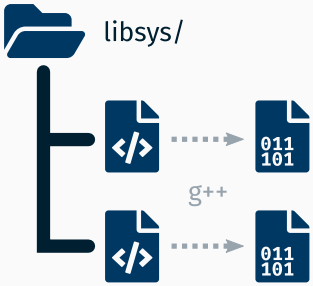

- Systemaufrufstümpfe (aus Aufgabe 2)
- C Funktionsprologe (siehe `compiler/crti.asm` und `compiler/crtn.asm`)

# libsys


- 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`)

# libsys

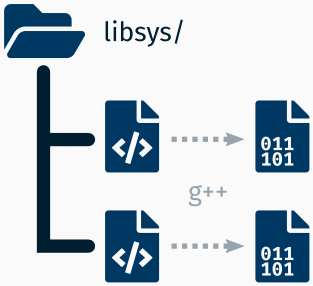
```
$ g++ -fno-builtin -nodefaultlibs -nostdlib -nostdinc …
```




- 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`)

# libsys

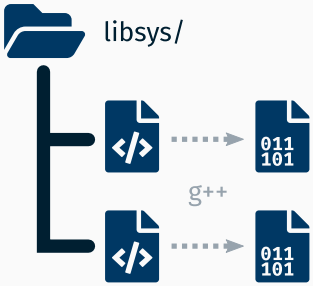
→ `$CXXFLAGS` aus `tools/build.mk` berücksichtigen




- 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`)

# libsys

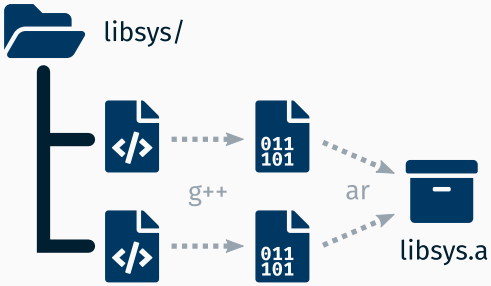
→ oder gleich `tools/common.mk` in `Makefile` einbinden




- 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`)

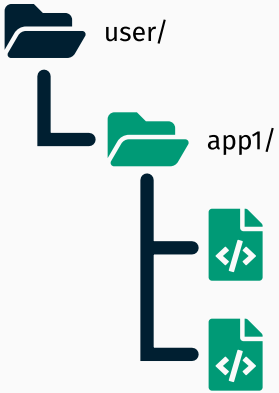
# libsys

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

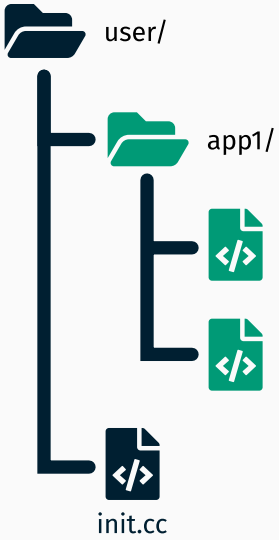



- Systemaufrufstümpfe (aus Aufgabe 2)
- C Funktionsprologe (siehe `compiler/crti.asm` und `compiler/crtn.asm`)
- *optional beliebig erweiterbar*
  - komfortabler Ausgabestrom (siehe `object/outputstream.cc`)
  - dynamischer Allokator → Untertützung für new (siehe `utils/alloc.cc`)

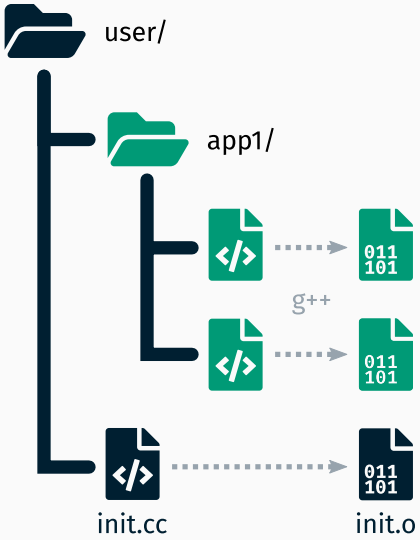
# Anwendung



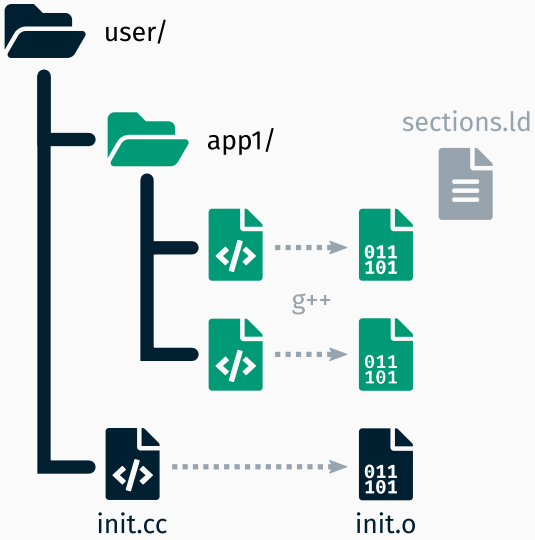
# Anwendung



# Anwendung

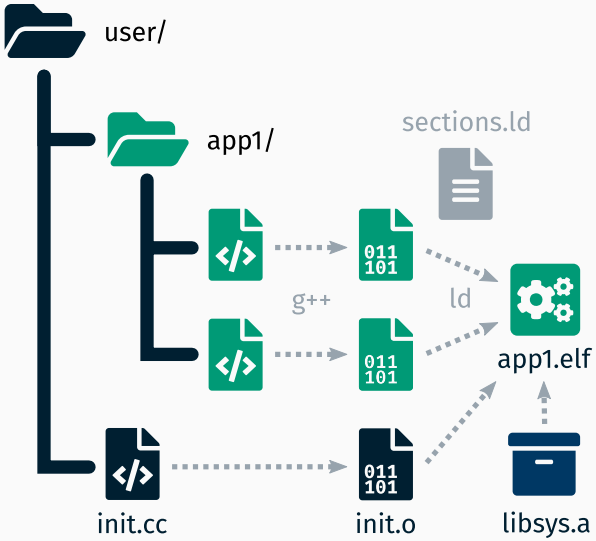


Linkerskript `user/sections.ld` mit `0x400 0000` (64 MiB) als Einsprungpunkt



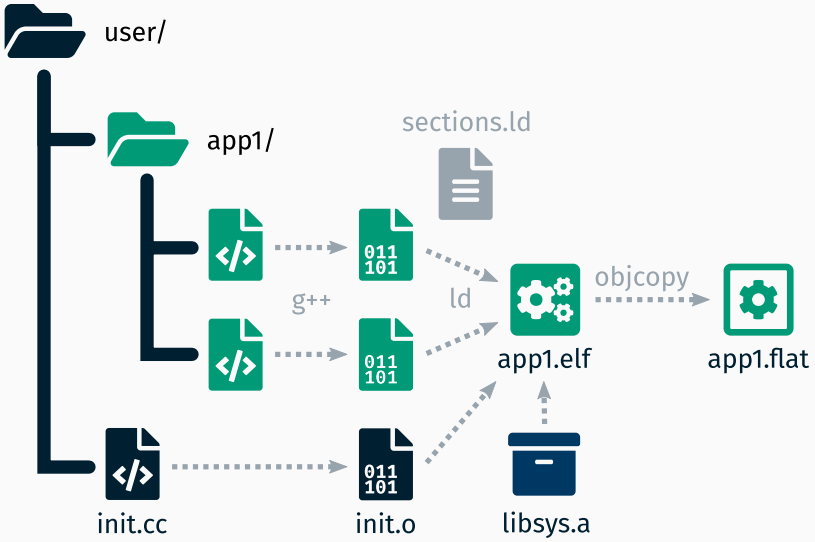
# Anwendung

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



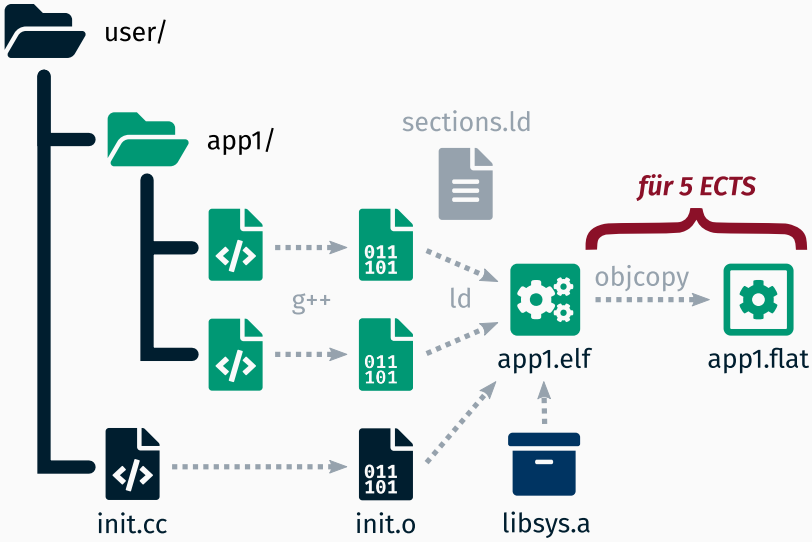
# Anwendung

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



# Anwendung

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



# 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`)

# 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 Initialisierungssroutinen
     (`__preinit_array`, `_init()` und `__init_array`, siehe `compiler/libc.cc`)
  3. Übergabe an `main` der App
  4. bei Rückkehr Deinitialisierung
     (`__fini_array`, `_fini()` und Endlossschleife)
  - Bereitstellung der für C++ benötigten Funktionen
    (Dummies für `__cxa_pure_virtual()` und `__cxa_atexit()`, siehe `compiler/libcxx.cc`)

# C/C++ Laufzeitumgebung

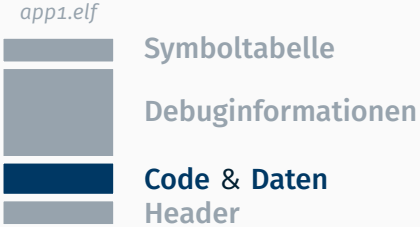
- `crti.o` und `crtn.o` für C Funktionsprologe
  - für Initialisierungs- (`_init`) und Beendigungsroutine (`_fini`)
    (siehe `compiler/crti.asm` sowie `compiler/crtn.asm`)
- `init.o` setzt die Laufzeitumgebung auf
  1. Einsprungspunkt ist `void start()`
  2. Ausführung der C Startup Initialisierungssroutinen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

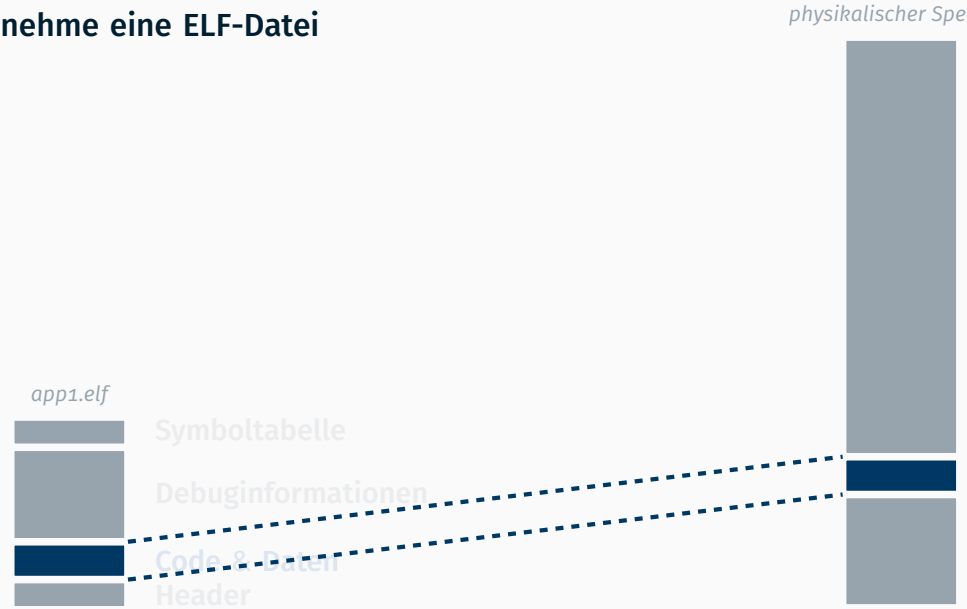
# Erstellung einer flachen Binärdatei

**Man nehme eine ELF-Datei**



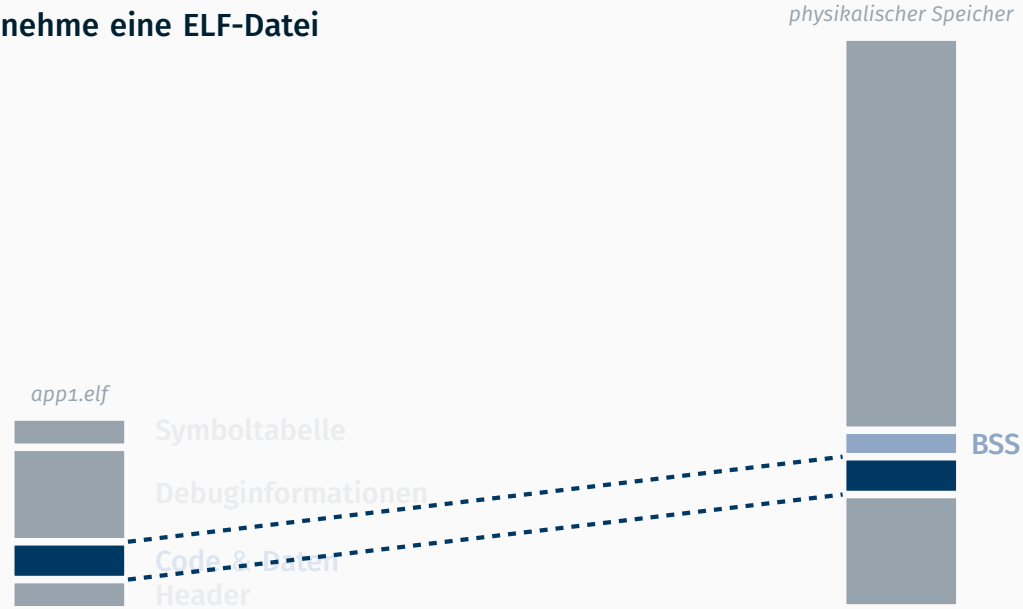
# Erstellung einer flachen Binärdatei

## Man nehme eine ELF-Datei



# Erstellung einer flachen Binärdatei

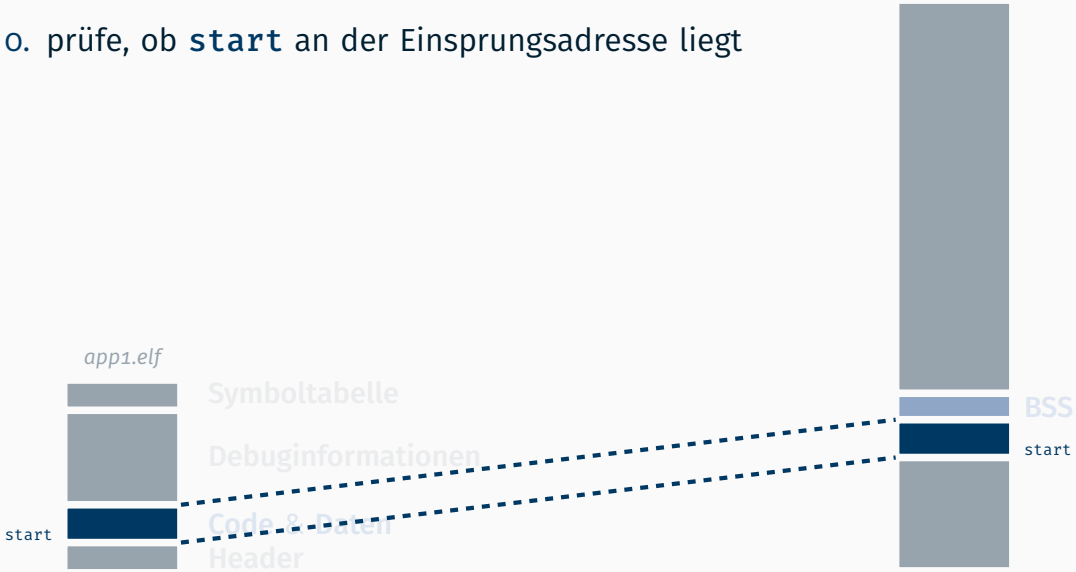
## Man nehme eine ELF-Datei



# Erstellung einer flachen Binärdatei

**Man nehme eine ELF-Datei**

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

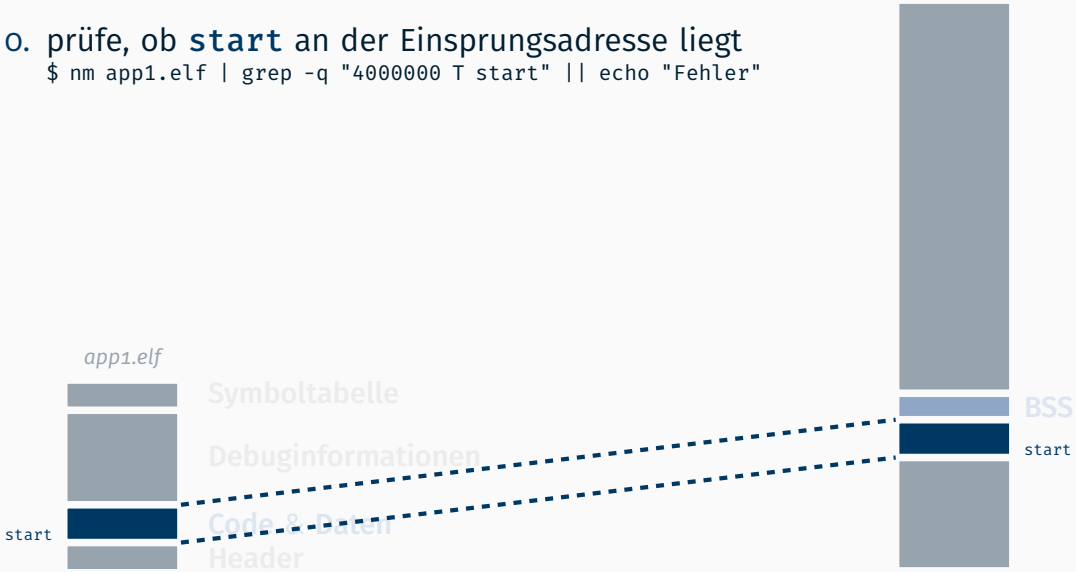


# Erstellung einer flachen Binärdatei

## Man nehme eine ELF-Datei

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

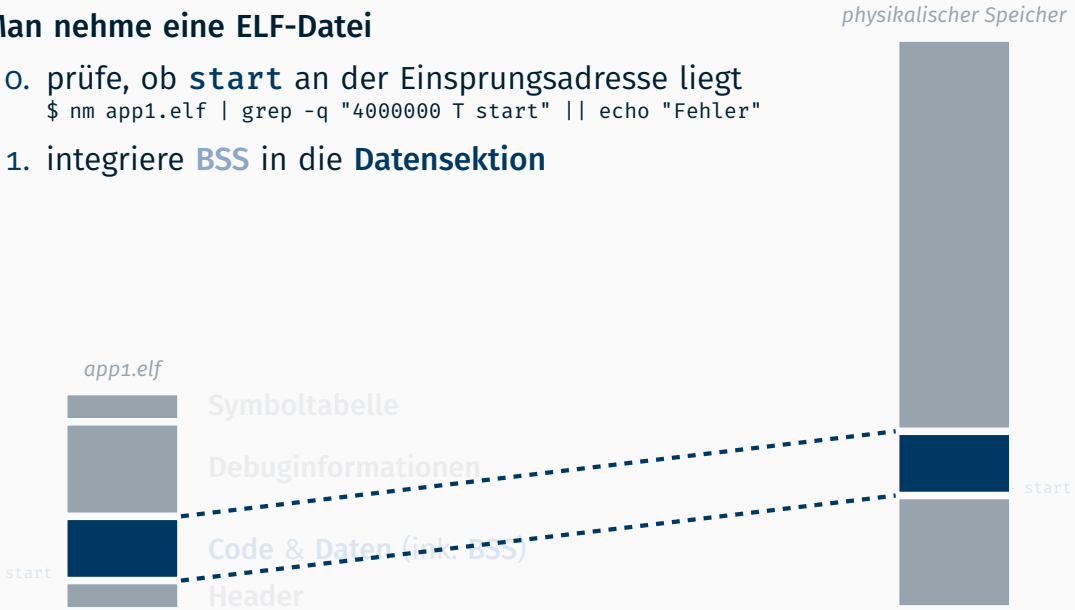
```
$ nm app1.elf | grep -q "4000000 T start" || echo "Fehler"
```



# 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

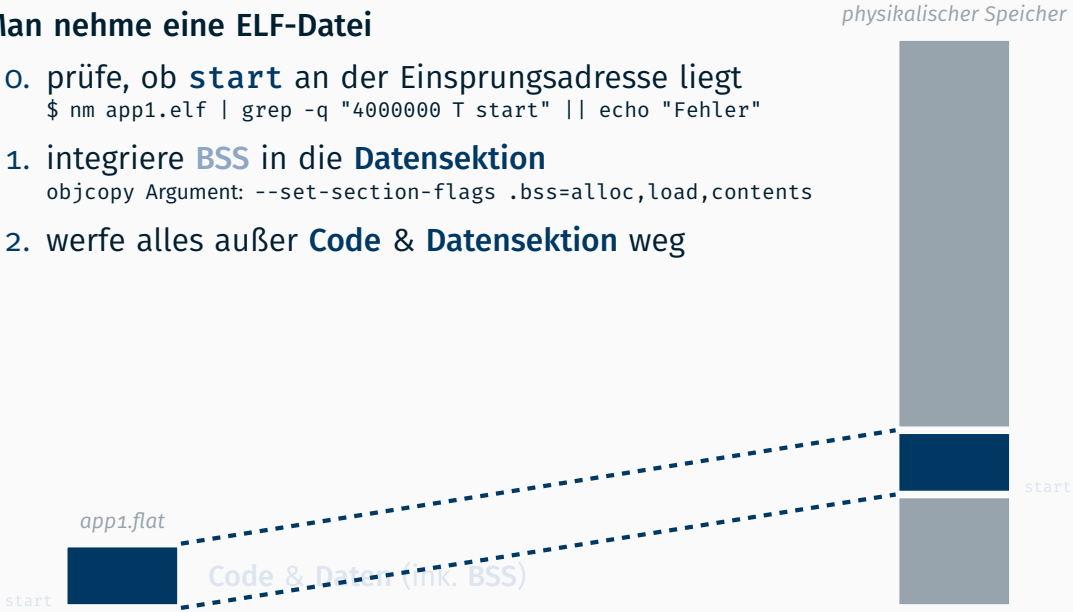
## Man nehme eine ELF-Datei

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`

# 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



# 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`

# 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

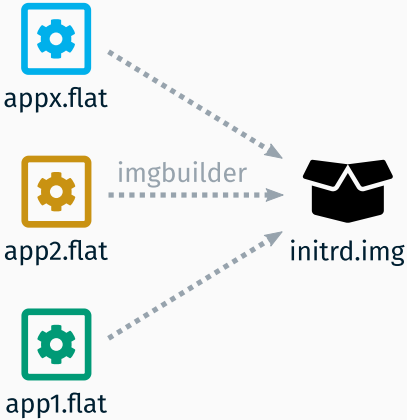
appx.flat

app2.flat

app1.flat

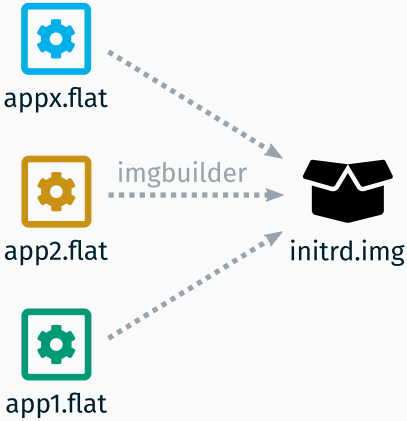
# Anwendungsabbildarchiv als Boot Module

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



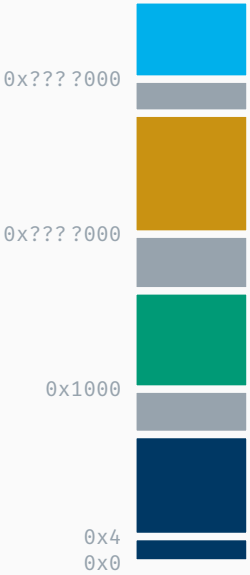
# Anwendungsabbildarchiv als Boot Module

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




➔ Der Quelltext des *Image Builder*s ist in der Aufgabenstellung verlinkt.

# *Image Builder* Format




**Anwendung x**

`.text`, `.data`, `.bss`

Füller / Ausrichten auf 4 KiB

**Anwendung 2**

`.text`, `.data`, `.bss`

Füller / Ausrichten auf 4 KiB

**Anwendung 1**

`.text`, `.data`, `.bss`

Füller / Ausrichten auf 4 KiB

**Informationen** zu den Abbildern

```
uint32_t appSize[appCount];
uint32_t appCount;
```

# 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

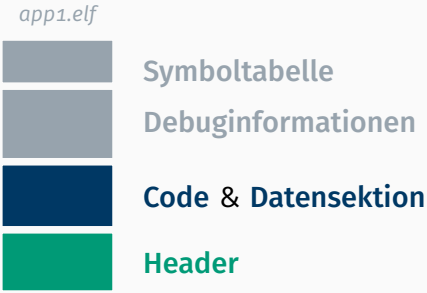
  **relocatable** Objektdateien (`.o`)

  **executable** (statische) Programme

  **dynamic** dynamische Programme und [geteilte] Bibliotheken (`.so`)
- durch Sektionen flexibel erweiterbar
- bei Bedarf Einbettung von *DWARF* Debuginformationen

# Executable and Linking Format: Details





# Executable and Linking Format: Details



# Executable and Linking Format: Details



## ELF Header

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

# Executable and Linking Format: Details



## 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

# Executable and Linking Format: Details



```
$ 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
```

# Executable and Linking Format: Details



```
$ 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
```

# Executable and Linking Format: Details



```
$ 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**

# ELF 101 — a Linux executable walkthrough

ANGE ALBERTINI
CORKAMI.COM

## DISSECTED FILE

```
$ uname -m
x86_64
$ ./simple64.elf
Hello World!
```

SIMPLE64.ELF

HEADER 1/2
TECHNICAL DETAILS FOR IDENTIFICATION AND EXECUTION

ELF HEADER
IDENTIFY AS AN ELF FILE + SPECIFY THE ARCHITECTURE

PROGRAM HEADER TABLE
EXECUTION INFORMATION

SECTIONS
CONTENTS OF THE EXECUTABLE

CODE
EXECUTABLE INFORMATION

DATA
INFORMATION USED BY THE CODE

SECTIONS' NAMES
.shstrtab..text
.rodata

HEADER 2/2
TECHNICAL DETAILS FOR LINKING AND/OR FOR EXECUTION

SECTION HEADER TABLE
LINKING AND/OR PROGRAM DISPLAY INFORMATION

HEXADECIMAL DUMP / ASCII DUMP

```
7F 45 4C 46 02 01 01 00 00 00 00 00 00 00 00 00  .ELF............
02 00 3E 00 01 00 00 00 80 00 00 00 00 00 00 00  ..>.............
40 00 00 00 00 00 00 00 F0 00 00 00 00 00 00 00  @...............
00 00 00 00 40 00 38 00 01 00 40 00 04 00 03 00  ....@.8...@.....
```

```
01 00 00 00 05 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 80 00 00 00 00 00 00 00  ................
D0 00 00 00 00 00 00 00 D0 00 00 00 00 00 00 00  ................
08 00 00 00 00 00 00 00                          ........
```

```
48 B8 01 00 00 00 00 00 00 00 48 C7 C6 C0 00 00  H.........H.....
10 48 C7 C7 01 00 00 00 48 C7 C0 01 00 00 00 0F  .H......H.......
05 48 C7 C7 01 00 00 00 48 C7 C0 3C 00 00 00 0F  .H......H..<....
05                                               .
```

```
48 65 6C 6C 6F 20 57 6F 72 6C 64 21 0A 00        Hello World!..
```

```
00 2E 73 68 73 74 72 74 61 62 00 2E 74 65 78 74  ..shstrtab..text
00 2E 72 6F 64 61 74 61 00                       ..rodata.
```

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
31 00 00 00 01 00 00 00 06 00 00 00 00 00 00 00  1...............
00 00 00 00 00 00 00 00 78 00 00 00 00 00 00 00  ........x.......
00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00  ................
11 00 00 00 01 00 00 00 02 00 00 00 00 00 00 00  ................
C0 00 00 00 00 00 00 00 C0 00 00 00 00 00 00 00  ................
0D 00 00 00 00 00 00 00 08 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
01 00 00 00 03 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 D0 00 00 00 00 00 00 00  ................
19 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

FIELDS / VALUES / EXPLANATION

1 e_ident

| FIELDS | VALUES | EXPLANATION |
|---|---|---|
| EI_MAG | 0x7F, "ELF" | CONSTANT SIGNATURE |
| EI_CLASS, EI_DATA | 2, 1 | 64-BITS, LITTLE-ENDIAN |
| EI_VERSION | 1 | ALWAYS 1 |
| e_type | 2 | EXECUTABLE |
| e_machine | 0x3E | AMD64 (AMD LATER) |
| e_version | 1 | ALWAYS 1 |
| e_entry | 0x10000000 | 3 ADDRESS WHERE EXECUTION STARTS |
| e_phoff | 0x40 | PROGRAM HEADERS OFFSET |
| e_shoff | 0xF0 | SECTION HEADERS OFFSET |
| e_ehsize | 0x40 | ELF HEADERS SIZE |
| e_phentsize | 0x38 | SIZE OF A SINGLE PROGRAM HEADER |
| e_phnum | 1 | COUNT OF PROGRAM HEADERS |
| e_shentsize | 0x40 | SIZE OF A SINGLE SECTION HEADER |
| e_shnum | 4 | COUNT OF SECTION HEADERS |
| e_shstrndx | 3 | INDEX OF THE NAMES SECTION IN THE TABLE |

2

| | | |
|---|---|---|
| p_type | 1 | THE SEGMENT SHOULD BE LOADED IN MEMORY |
| p_flags | 5 | READABLE AND EXECUTABLE |
| p_offset | 0 | OFFSET WHERE IT SHOULD BE READ |
| p_vaddr | 0x10000000 | VIRTUAL ADDRESS WHERE IT SHOULD BE LOADED |
| p_paddr | 0x10000000 | PHYSICAL ADDRESS WHERE IT SHOULD BE LOADED |
| p_filesz | 0xD0 | SIZE ON FILE |
| p_memsz | 0xD0 | SIZE IN MEMORY |

X64 ASSEMBLY / EQUIVALENT C CODE

3

```
mov rax, 1
mov rsi, 0x10000040
mov rdi, 1
mov rax, 1
syscall
```

```
write(STDOUT_FILENO, "Hello World!\n", len("Hello World!\n"));
```

```
mov rdi, 1
mov rax, 0x3c
syscall
```

```
exit(1);
```

STRINGS

```
"Hello World!\n", 0
```

SECTION NAMES

```
"" .shrtrtab .text .rodata
```

SECTION HEADER TABLE

| INDEX | NAME | TYPE | FLAGS | ADDRESS | OFFSET | SIZE |
|---|---|---|---|---|---|---|
| 0 | <null> | 0x0 | | 0x0 | 0x0 | 0x0 |
| 1 | .text | 1 PROGBITS | 6 ALLOC, EXECINSTR | 0x10000040 | 0x40 | 0x11 |
| 2 | .rodata | 1 PROGBITS | 2 ALLOC | 0x10000050 | 0x50 | 0x0D |
| 3 | .shrtrtab | 3 STRTAB | 0 | 0x0 | 0xD0 | 0x19 |

POSITIVE OFFSETS TO ELEMENT START(S)

THESE THE WHOLE FILE. HOWEVER, MOST ELF FILES CONTAIN MANY MORE ELEMENTS.
EXPLANATIONS ARE SIMPLIFIED FOR CONCISENESS.

# 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

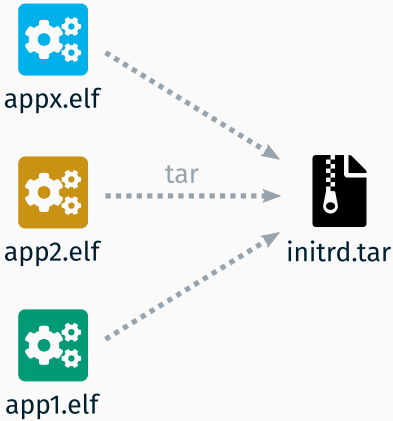
appx.elf

app2.elf

app1.elf

# Bandarchivierer (TAPE ARCHIVER) Verwendung

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



# TAR Format

**Dateiende** mit `char name[100] = { '\0' };`

Füller / Ausrichten auf 512 Bytes

**Anwendung x** – `appx.elf`

Kopf zu **Anwendung x**

Füller / Ausrichten auf 512 Bytes

**Anwendung 2** – `app2.elf`

Kopf zu **Anwendung 2**

Füller / Ausrichten auf 512 Bytes

**Anwendung 1** – `app1.elf`

0x200

Kopf zu **Anwendung 1**

```
char name[100]; // uvm
```

0

# TAR Format

**Dateiende** mit `char name[100] = { '\0' };`

Füller / Ausrichten auf 512 Bytes

**Anwendung x** – `appx.elf`

Kopf zu **Anwendung x**

Füller / Ausrichten auf 512 Bytes

**Anwendung 2** – `app2.elf`

Kopf zu **Anwendung 2**

Füller / Ausrichten auf 512 Bytes

**Anwendung 1** – `app1.elf`

0x200

Kopf zu **Anwendung 1**

```
char name[100]; // uvm
```

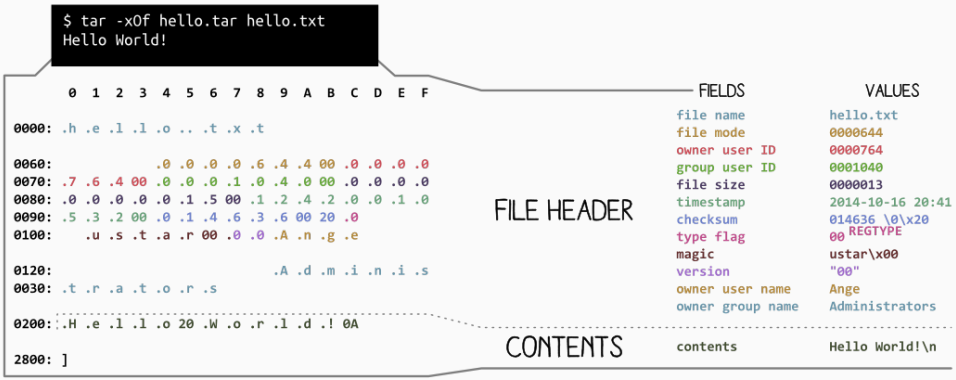
0

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

# TAPE ARCHIVE



ANGE ALBERTINI
http://www.corkami.com




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

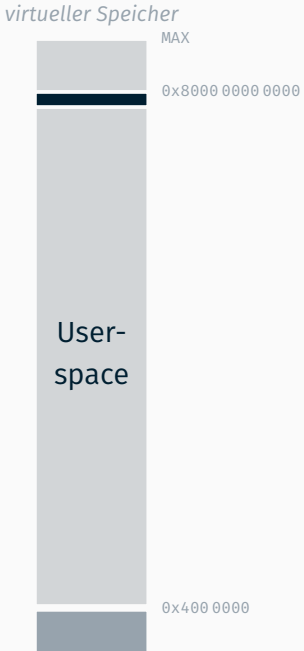
(d.h. die Angabe in `Multiboot::Module::getCommandLine()` ist nicht sonderlich aussagekräftig, sondern beim Benutzer immer gleich)

# Dynamisch wachsender Stapel

# Dynamisch wachsender Anwendungsstapel

**Ablauf:**

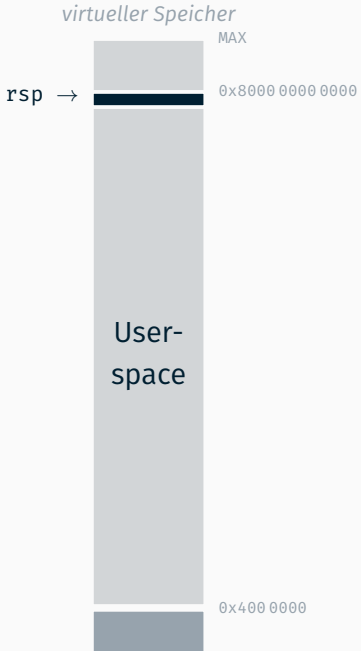
- initial z.B. 4 KiB Userspace Stack



# Dynamisch wachsender Anwendungsstapel

**Ablauf:**

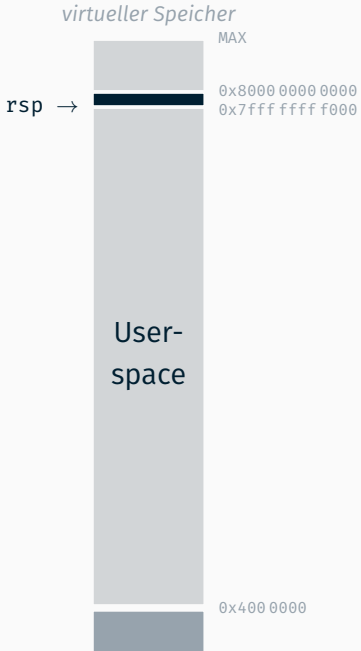
- initial z.B. 4 KiB Userspace Stack



# Dynamisch wachsender Anwendungsstapel

**Ablauf:**

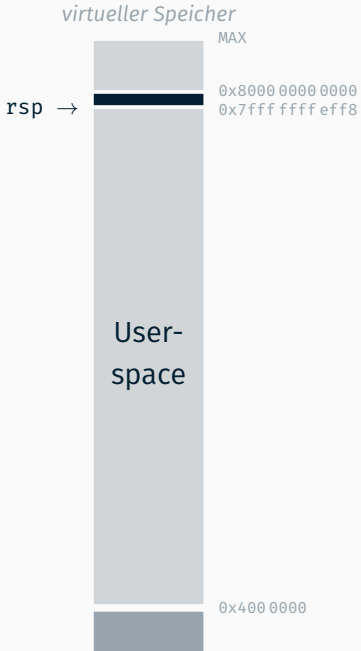
- initial z.B. 4 KiB Userspace Stack
- sobald Stack voll



# Dynamisch wachsender Anwendungsstapel

**Ablauf:**

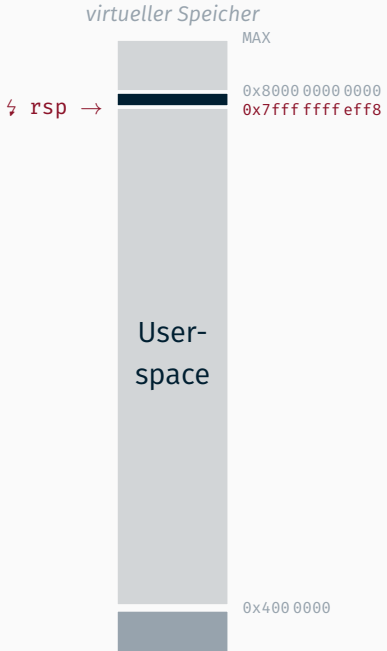
- initial z.B. 4 KiB Userspace Stack
- sobald Stack voll



# Dynamisch wachsender Anwendungsstapel

**Ablauf:**

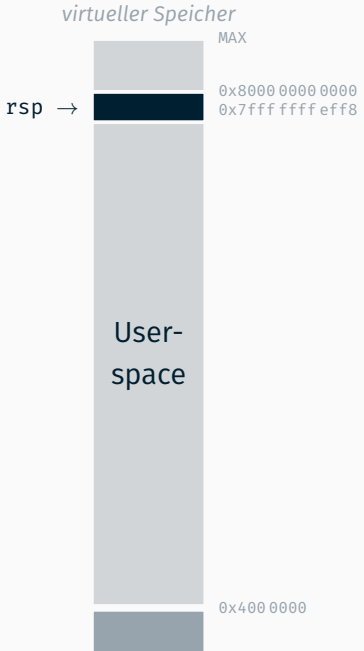
- initial z.B. 4 KiB Userspace Stack
- sobald Stack voll ➔ Seitenfehler



# Dynamisch wachsender Anwendungsstapel

**Ablauf:**

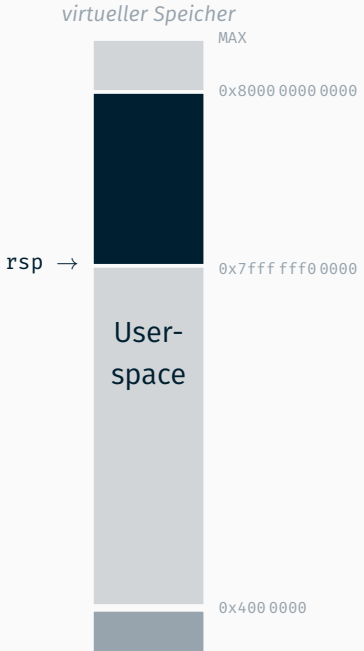
- initial z.B. 4 KiB Userspace Stack
- sobald Stack voll → Seitenfehler
- Behandlungsroutine allokiert mehr Speicher (und setzt Ausführung fort)



# 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

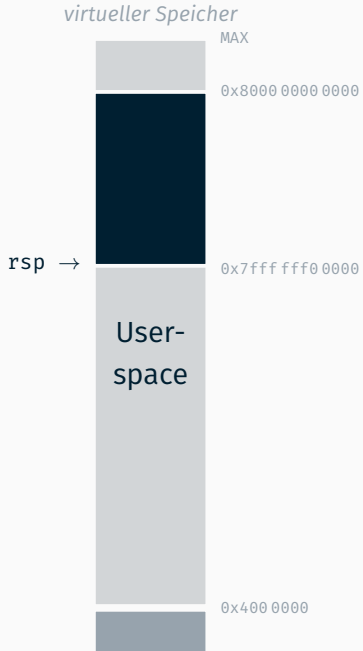


# Dynamisch wachsender Anwendungsstapel

**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`



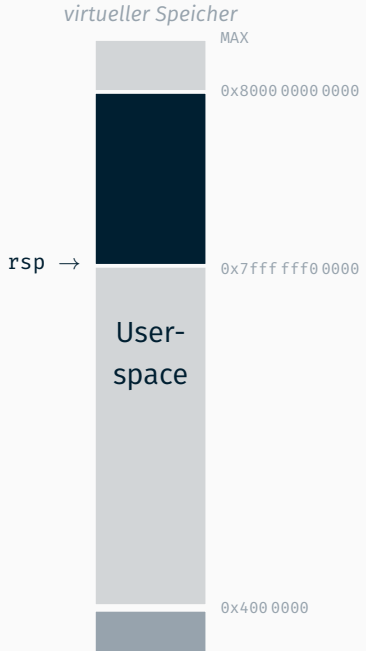
# Dynamisch wachsender Anwendungsstapel

**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`



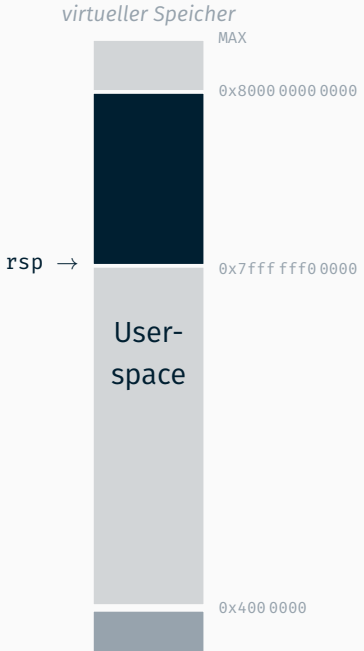
# Dynamisch wachsender Anwendungsstapel

**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

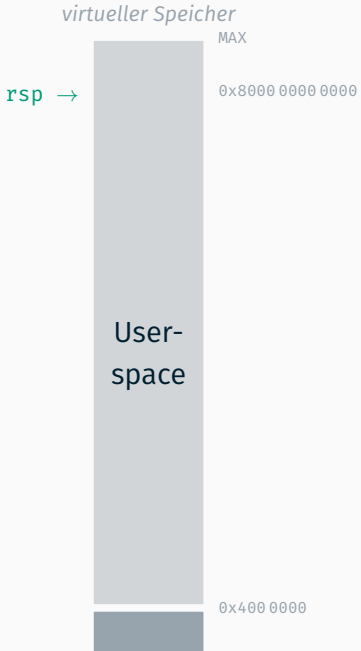
**Ablauf:**

- 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 allokieren!




**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 (`xmmm0` – `xmmm15`)

# 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 (`xmmm0` – `xmmm15`)
  - 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 (`xmmm0` – `xmmm15`)
  - 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

# 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
- OS aktiviert FPU, führt die Instruktion nochmal aus und kümmert sich bei diesem Programm von nun an auch um die FPU Register

# 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

# 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

# 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)

# 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 StuBSmi

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

# FPU-Umsetzung in StuBSmi

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

# FPU-Umsetzung in StuBSmi

- 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 StuBSmi

- 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 StuBSmi

- 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 StuBSmi

- 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 StuBSmi

- 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 StuBSmi

- 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`)
- beim Kontextwechsel aktuellen Zustand sichern (→ `FPU::State::save()`) und neuen laden (→ `FPU::State::restore()`)

# FPU-Umsetzung in StuBSmi

- 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`)
- beim Kontextwechsel aktuellen Zustand sichern (→ `FPU::State::save()`) und neuen laden (→ `FPU::State::restore()`)
- optional auch Unterstützung für `float` und `double` im Ausgabestrom der **`libsys`**

# Fragen?