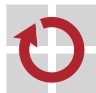
Übung zu Betriebssystemtechnik

Paging in StuBSmI

23. Mai 2019

Bernhard Heinloth, Andreas Ziegler,
Christian Eichler & Harald Böhm

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



Exkurs: Multiboot Specification
oder Wie stiefel ich meinen Kernel?

Überblick zur Multiboot Specification

- offener PC **Bootloader Standard**, ab 1995 entwickelt

Überblick zur Multiboot Specification

- offener PC **Bootloader Standard**, ab 1995 entwickelt
- Betriebssystem muss als **32bit ELF** oder **a.out** vorliegen

Überblick zur Multiboot Specification

- offener PC **Bootloader Standard**, ab 1995 entwickelt
- Betriebssystem muss als **32bit ELF** oder **a.out** vorliegen
- übernimmt die [hässliche] Initialisierung eines x86 PCs in einen **wohl definierten Zustand**
  - 32 bit Protected Mode
  - nur BSP (Bootstrap Processor)
  - A20 Gate aktiviert (> 1 MB nutzbar)
  - setzt optional auch Grafikmodus

Überblick zur Multiboot Specification

- offener PC **Bootloader Standard**, ab 1995 entwickelt
- Betriebssystem muss als **32bit ELF** oder **a.out** vorliegen
- übernimmt die [hässliche] Initialisierung eines x86 PCs in einen **wohl definierten Zustand**
  - 32 bit Protected Mode
  - nur BSP (Bootstrap Processor)
  - A20 Gate aktiviert (> 1 MB nutzbar)
  - setzt optional auch Grafikmodus
- übergibt dem BS **„vitale“ Informationen** über das System

Überblick zur Multiboot Specification

- offener PC **Bootloader Standard**, ab 1995 entwickelt
- Betriebssystem muss als **32bit ELF** oder **a.out** vorliegen
- übernimmt die [hässliche] Initialisierung eines x86 PCs in einen **wohl definierten Zustand**
  - 32 bit Protected Mode
  - nur BSP (Bootstrap Processor)
  - A20 Gate aktiviert (> 1 MB nutzbar)
  - setzt optional auch Grafikmodus
- übergibt dem BS **„vitale“ Informationen** über das System
- lädt ggf. auch die **initiale Ramdisk** in den Speicher

Überblick zur MULTIBOOT SPECIFICATION

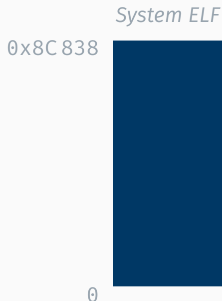
- offener PC **Bootloader Standard**, ab 1995 entwickelt
- Betriebssystem muss als **32bit ELF** oder **a.out** vorliegen
- übernimmt die [hässliche] Initialisierung eines x86 PCs in einen **wohl definierten Zustand**
  - 32 bit Protected Mode
  - nur BSP (Bootstrap Processor)
  - A20 Gate aktiviert (> 1 MB nutzbar)
  - setzt optional auch Grafikmodus
- übergibt dem BS **„vitale“ Informationen** über das System
- lädt ggf. auch die **initiale Ramdisk** in den Speicher
- wird u.a. von **GRUB** (Referenzimplementierung) und **PXELINUX** (Netzwerkboot) unterstützt

Überblick zur MULTIBOOT SPECIFICATION

- offener PC **Bootloader Standard**, ab 1995 entwickelt
- Betriebssystem muss als **32bit ELF** oder **a.out** vorliegen
- übernimmt die [hässliche] Initialisierung eines x86 PCs in einen **wohl definierten Zustand**
  - 32 bit Protected Mode
  - nur BSP (Bootstrap Processor)
  - A20 Gate aktiviert (> 1 MB nutzbar)
  - setzt optional auch Grafikmodus
- übergibt dem BS **„vitale“ Informationen** über das System
- lädt ggf. auch die **initiale Ramdisk** in den Speicher
- wird u.a. von **GRUB** (Referenzimplementierung) und **PXELINUX** (Netzwerkboot) unterstützt
- und wird für OO/MPSTUBS verwendet

Aufbau einer MULTIBOOT-kompatiblen Binärdatei

Beispiel:
`make` generiert eine 563K große `build/system` ELF.

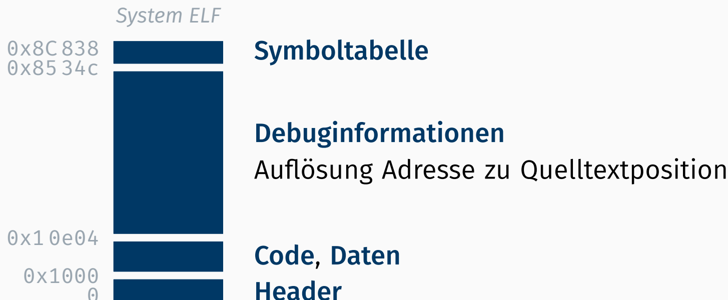


Aufbau einer MULTIBOOT-kompatiblen Binärdatei

Beispiel:

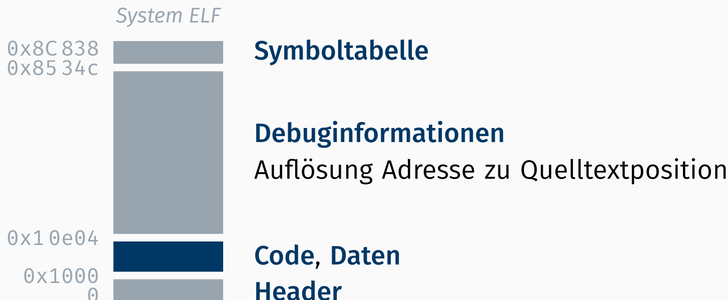
`make` generiert eine 563K große `build/system` ELF.

Analyse mittels `readelf` offenbart folgende Struktur



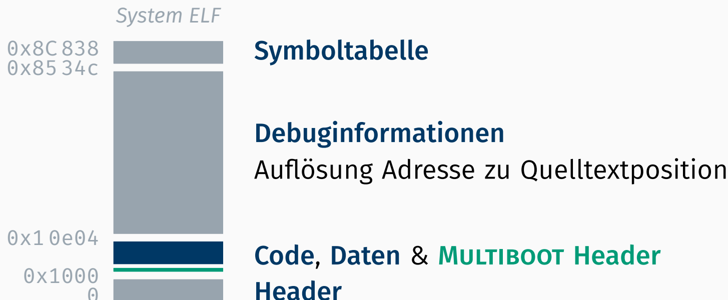
Aufbau einer MULTIBOOT-kompatiblen Binärdatei

Beispiel:
`make` generiert eine 563K große `build/system` ELF.
Analyse mittels `readelf` offenbart folgende Struktur,
für uns ist jedoch nur die (durch das Linkerskript) zusammengefasste Code- (`.text`) und Datensektion (`.[ro]data`) interessant



Aufbau einer MULTIBOOT-kompatiblen Binärdatei

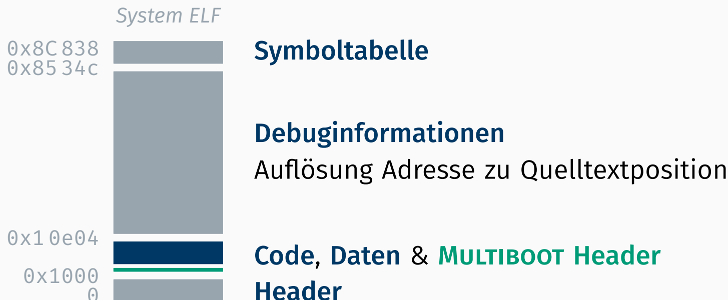
Beispiel:
make generiert eine 563K große build/system ELF.
Analyse mittels `readelf` offenbart folgende Struktur,
für uns ist jedoch nur die (durch das Linkerskript) zusammengefasste Code- (`.text`) und Datensektion (`.[ro]data`) interessant, in welcher auch der **MULTIBOOT Header** liegt



Aufbau einer Multiboot-kompatiblen Binärdatei

Multiboot Header

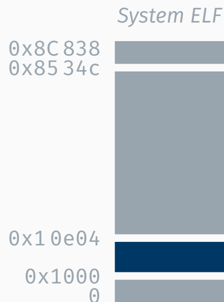
- Erkennung durch Wert `0x1badb002` (und Prüfsumme)
- muss in den ersten 8192 Bytes (der ELF) liegen
- bei uns in `startup.asm` definiert
- beinhaltet Konfiguration wie Einsprungsadresse

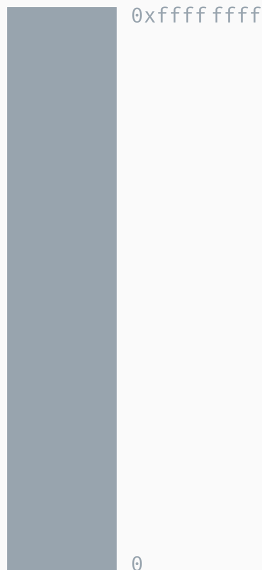


Laden einer MULTIBOOT-kompatiblen Binärdatei

Ablauf im Bootloader

1. liest *System ELF*

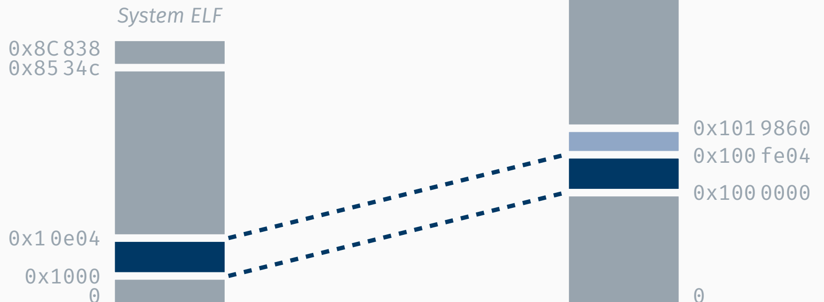




Laden einer MULTIBOOT-kompatiblen Binärdatei

Ablauf im Bootloader

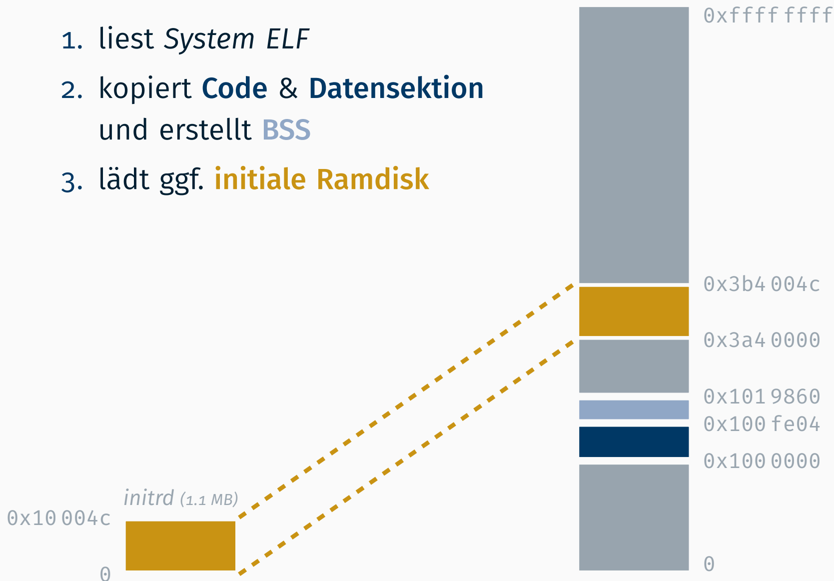
1. liest *System ELF*
2. kopiert **Code** & **Datensektion** und erstellt **BSS**



Laden einer MULTIBOOT-kompatiblen Binärdatei

Ablauf im Bootloader

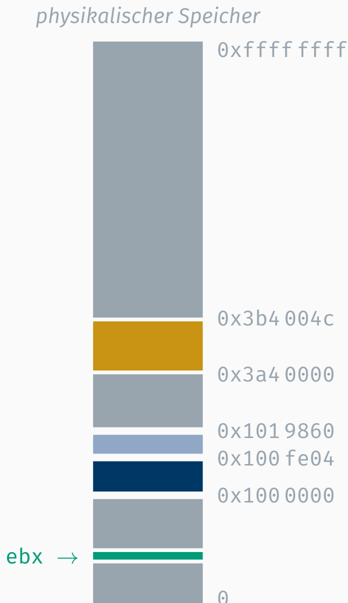
1. liest *System ELF*
2. kopiert **Code** & **Datensektion** und erstellt **BSS**
3. lädt ggf. **initiale Ramdisk**



Laden einer MULTIBOOT-kompatiblen Binärdatei

Ablauf im Bootloader

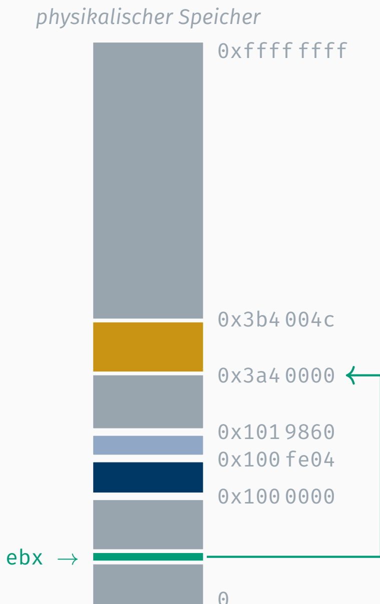
1. liest *System ELF*
2. kopiert **Code** & **Datensektion** und erstellt **BSS**
3. lädt ggf. **initiale Ramdisk**
4. setzt `%eax` auf `0x2BADB002` sowie `%ebx` als Zeiger auf Struktur mit **MULTIBOOT Information**



Laden einer MULTIBOOT-kompatiblen Binärdatei

Ablauf im Bootloader

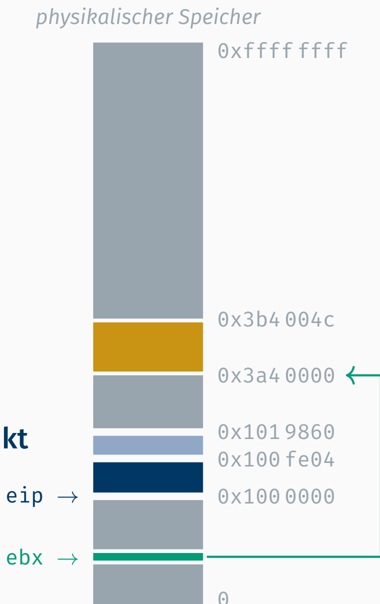
1. liest *System ELF*
2. kopiert **Code** & **Datensektion** und erstellt **BSS**
3. lädt ggf. **initiale Ramdisk**
4. setzt `%eax` auf `0x2BADB002` sowie `%ebx` als Zeiger auf Struktur mit **MULTIBOOT Information**



Laden einer MULTIBOOT-kompatiblen Binärdatei

Ablauf im Bootloader

1. liest *System ELF*
2. kopiert **Code** & **Datensektion** und erstellt **BSS**
3. lädt ggf. **initiale Ramdisk**
4. setzt `%eax` auf `0x2BADB002` sowie `%ebx` als Zeiger auf Struktur mit **MULTIBOOT Information**
5. Springt an den **Einsprungpunkt** (und übergibt somit an das Betriebssystem)



Ziel dieser Übung

Implementieren Sie ein Betriebssystem mit Speicherschutz.

Implementieren Sie ein Betriebssystem
mit Speicherschutz.

(ein unbekannter Manager, echte Welt)

Bestandsaufnahme

Kernel und **Anwendung**
ineinander verwoben
im gemeinsamen Speicher.

Anforderungen

0xffff ffff

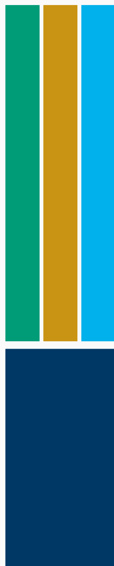
Anwendung
getrennt

0x200 0000

„lower-half“ **Kernel**
in den ersten 32 MB

0

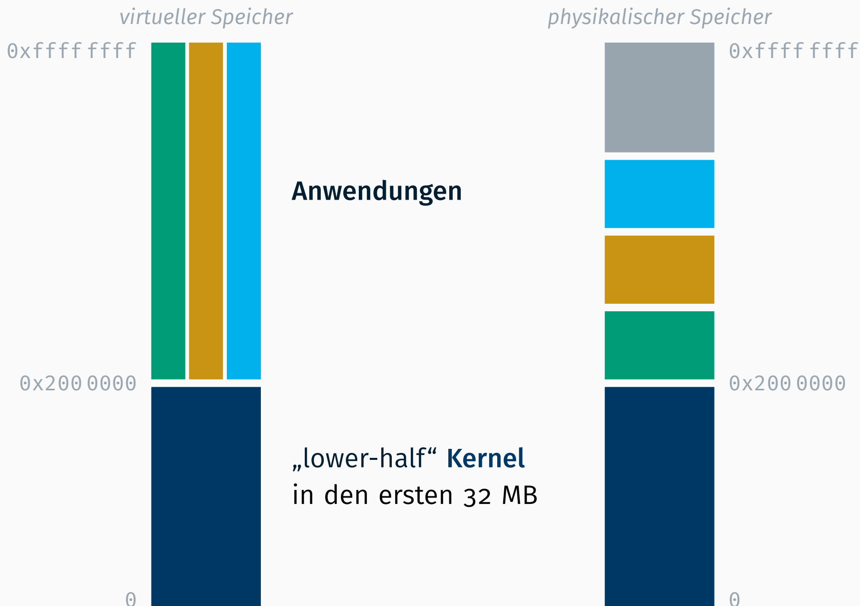
Anforderungen



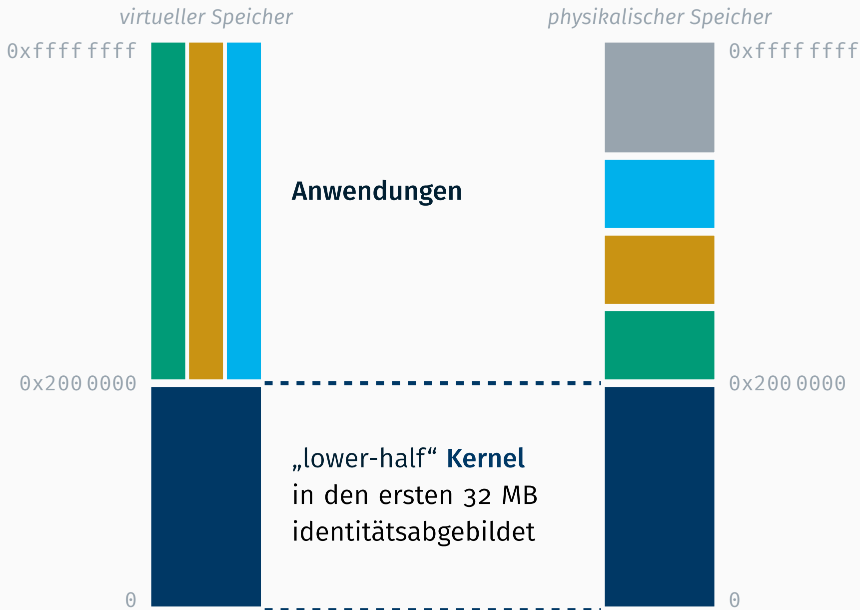
Anwendungen 1, 2, x auch untereinander getrennt

„lower-half" **Kernel**
in den ersten 32 MB

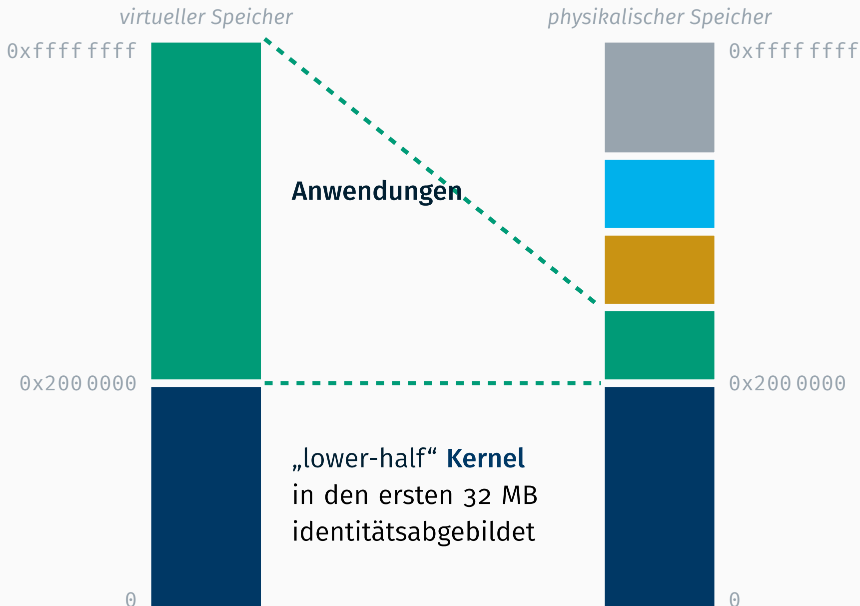
Umsetzung



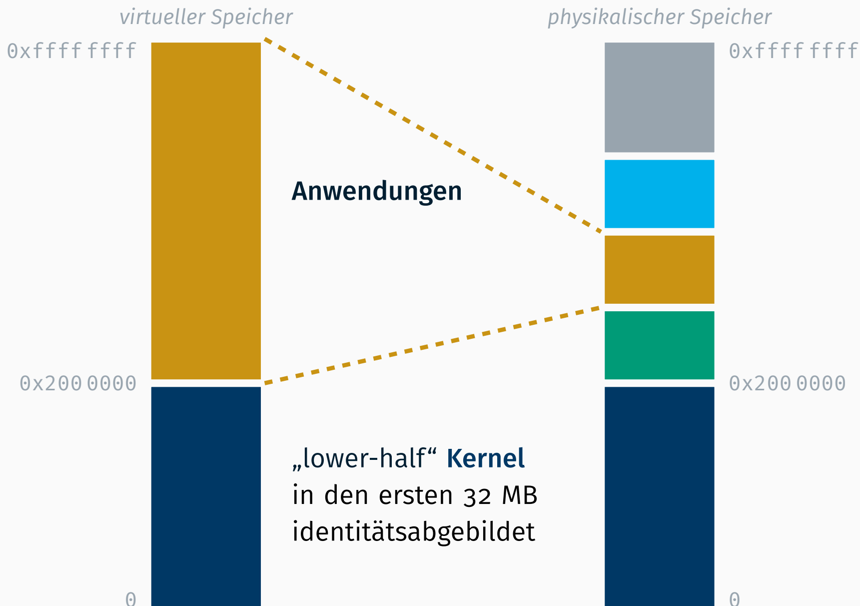
Umsetzung



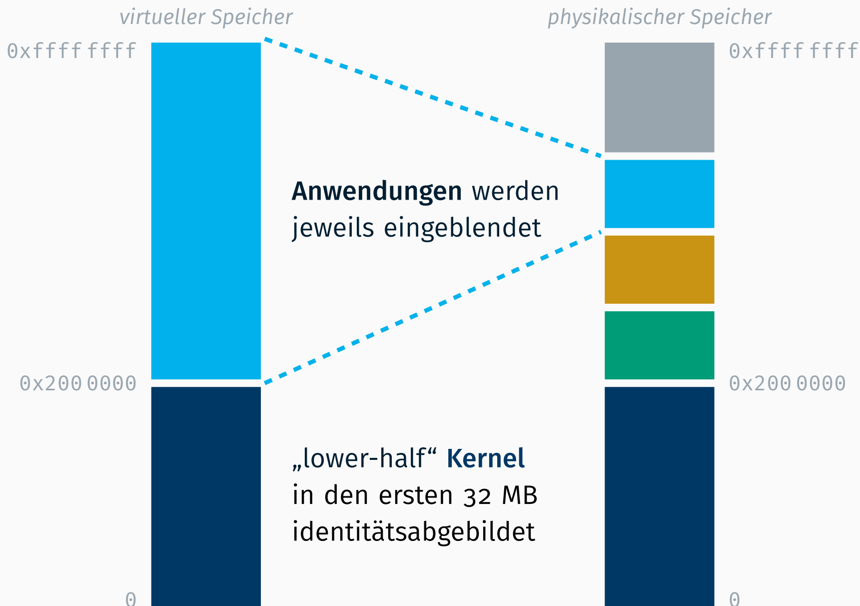
Umsetzung



Umsetzung



Umsetzung



Was ist zu tun?

Speicher verwalten

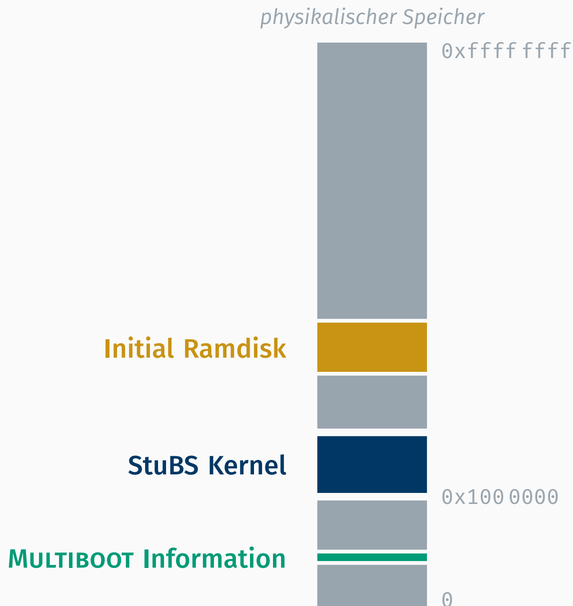
- Speicher erkennen
- Freien (Kern-/Anwendungs-)speicher verwalten
- Adressräume anlegen

Anwendungen auslagern

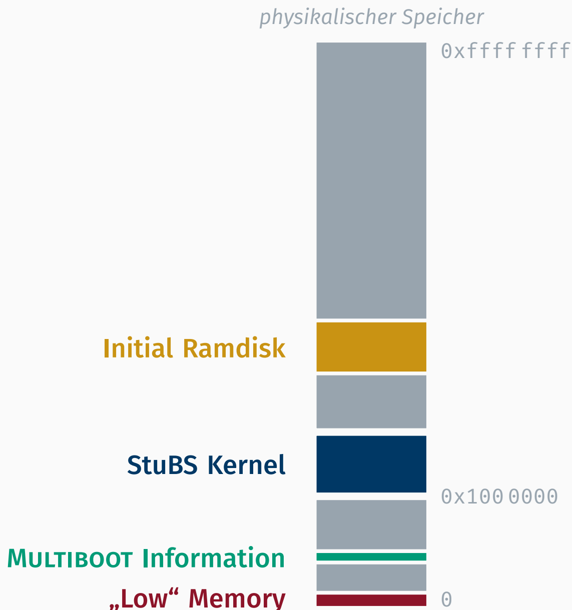
- Applikationen als eigene Anwendungen
- Erstellen eines initialen Speicherabbildes

Speicherverwaltung

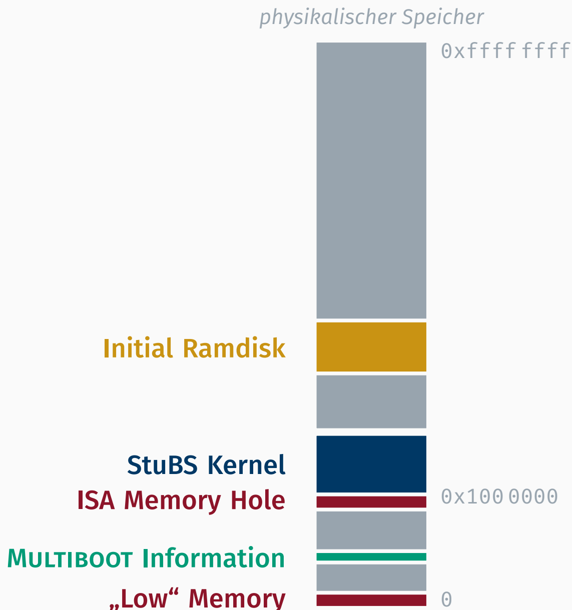
Freien Speicher finden



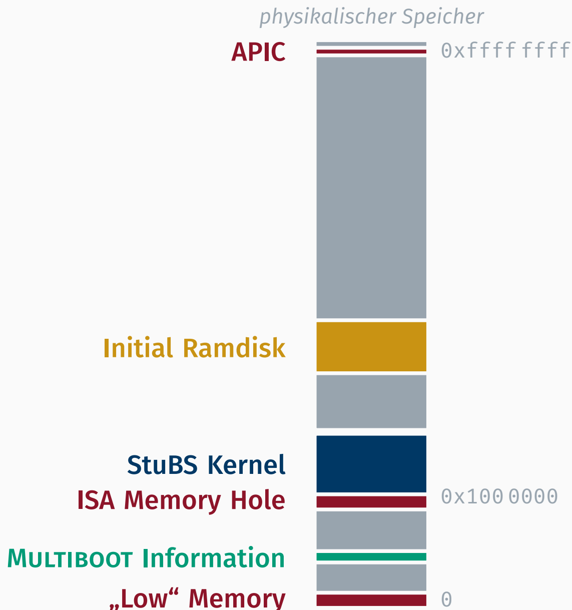
Freien Speicher finden



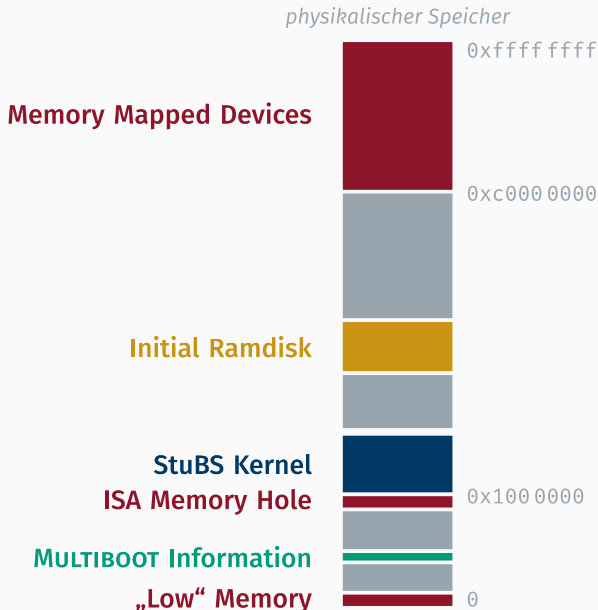
Freien Speicher finden



Freien Speicher finden

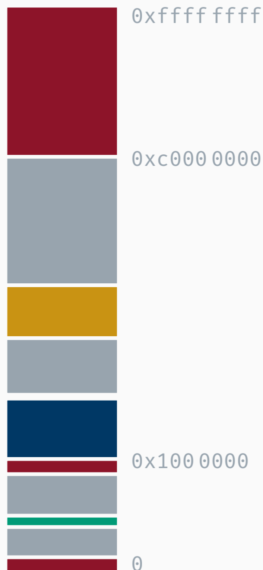


Freien Speicher finden



Freien Speicher finden

Abfrage der **Memory Map** über BIOS

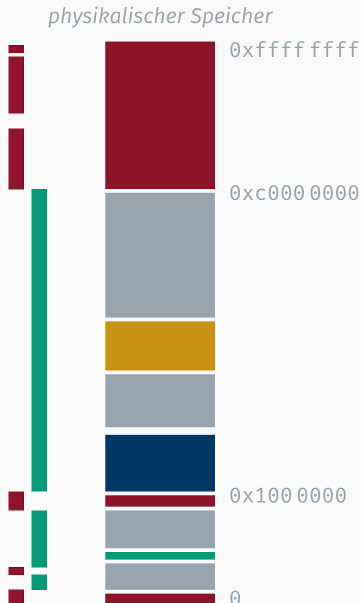


Freien Speicher finden

Abfrage der **Memory Map** über BIOS

Ergebnis in MULTIBOOT Information

- freier und belegter Speicher

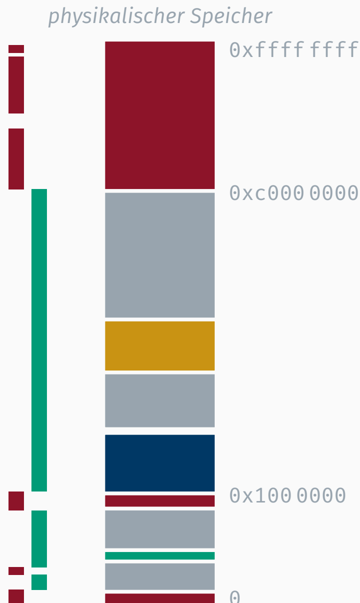


Freien Speicher finden

Abfrage der **Memory Map** über BIOS

Ergebnis in Multiboot Information

- freier und belegter Speicher
- ignoriert aber Kernel, `initrd` und Multiboot Information

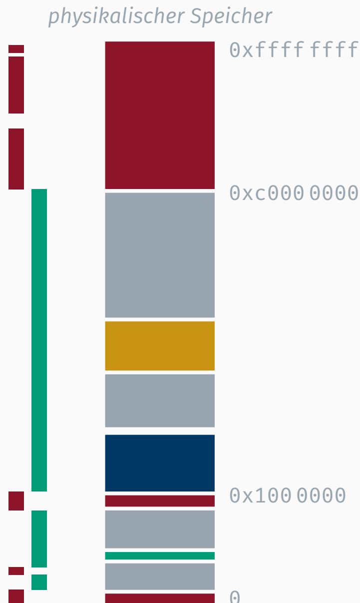


Freien Speicher finden

Abfrage der **Memory Map** über BIOS

Ergebnis in MULTIBOOT Information

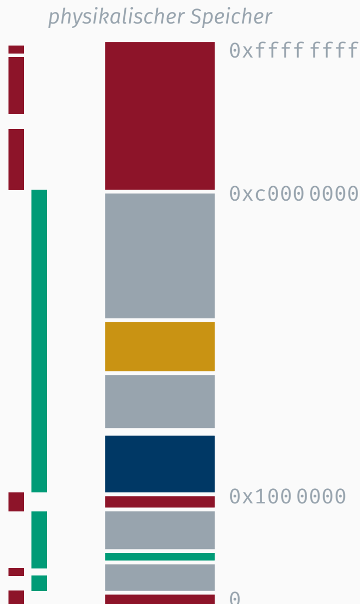
- freier und belegter Speicher
- ignoriert aber Kernel, `initrd` und MULTIBOOT Information
- besser defensiv auswerten
  (überlappende/widersprüchliche Bereiche!)



Freien Speicher finden

Abfrage der **Memory Map** über BIOS
Ergebnis in MULTIBOOT Information

- freier und belegter Speicher
- ignoriert aber Kernel, `initrd` und MULTIBOOT Information
- besser defensiv auswerten
  (überlappende/widersprüchliche Bereiche!)
- Adressen sind 64 Bit!

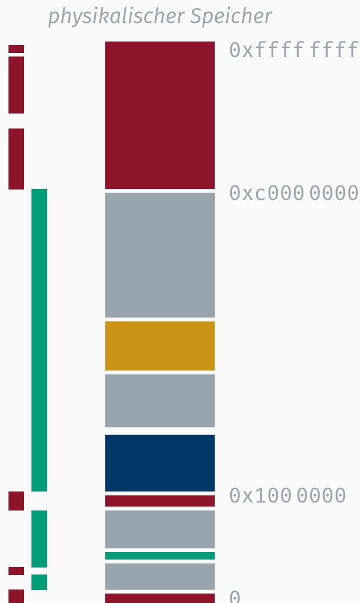


Freien Speicher finden

Abfrage der **Memory Map** über BIOS

Ergebnis in MULTIBOOT Information

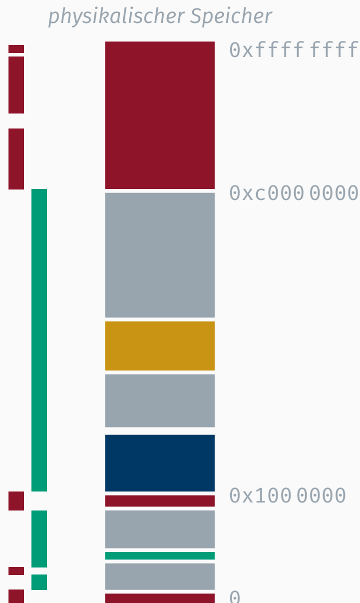
- freier und belegter Speicher
- ignoriert aber Kernel, `initrd` und MULTIBOOT Information
- besser defensiv auswerten (überlappende/widersprüchliche Bereiche!)
- Adressen sind 64 Bit!
- Speicherung in geeigneter Verwaltungsstruktur



Freien Speicher finden

Abfrage der **Memory Map** über BIOS
Ergebnis in MULTIBOOT Information

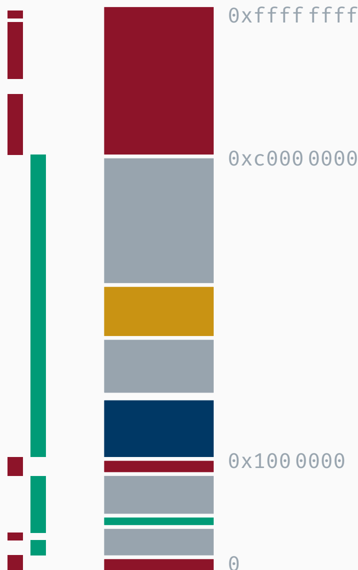
- freier und belegter Speicher
- ignoriert aber Kernel, `initrd` und MULTIBOOT Information
- besser defensiv auswerten
  (überlappende/widersprüchliche Bereiche!)
- Adressen sind 64 Bit!
- Speicherung in geeigneter Verwaltungsstruktur
  - Bitmap, ein Bit pro 4 KiB Page
    $\Rightarrow$ 131 072 Bytes für 4 GiB



Freien Speicher finden

Abfrage der **Memory Map** über BIOS
Ergebnis in MULTIBOOT Information

- freier und belegter Speicher
- ignoriert aber Kernel, `initrd` und MULTIBOOT Information
- besser defensiv auswerten (überlappende/widersprüchliche Bereiche!)
- Adressen sind 64 Bit!
- Speicherung in geeigneter Verwaltungsstruktur
  - Bitmap, ein Bit pro 4 KiB Page $\Rightarrow$ 131 072 Bytes für 4 GiB
  - erst freie Pages markieren
  - danach alle irgendwie belegten Pages ausknipsen



Paging

Adressumsetzung am Beispiel
oder: *Wo muss ich hin?*

Virtuelle Adresse: `0xdeadbabe`

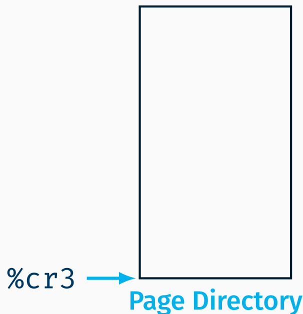
Adressumsetzung am Beispiel

oder: *Wo muss ich hin?*

Virtuelle Adresse: `0xdeadbabe`

`%cr3`

Virtuelle Adresse: `0xdeadbabe`

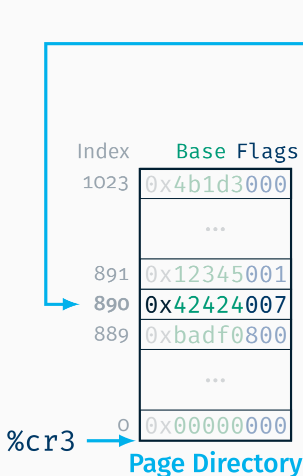


Virtuelle Adresse: `0xdeadbabe`

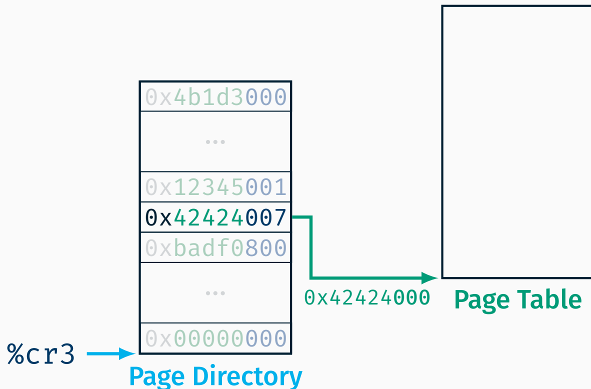
Index	Base Flags
1023	0x4b1d3000
	...
891	0x12345001
890	0x42424007
889	0xbadf0800
	...
0	0x00000000

%cr3 → **Page Directory**

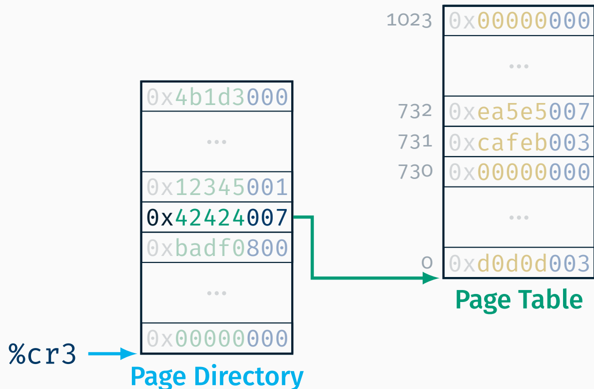
Virtuelle Adresse: `0xdeadbabe`



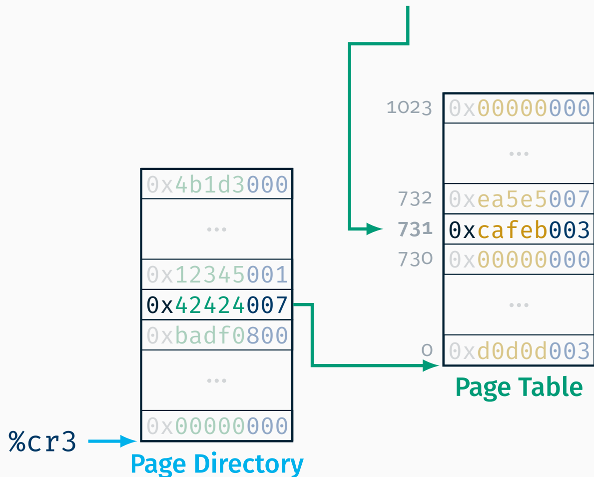
Virtuelle Adresse: `0xdeadbabe`



Virtuelle Adresse: `0xdeadbabe`

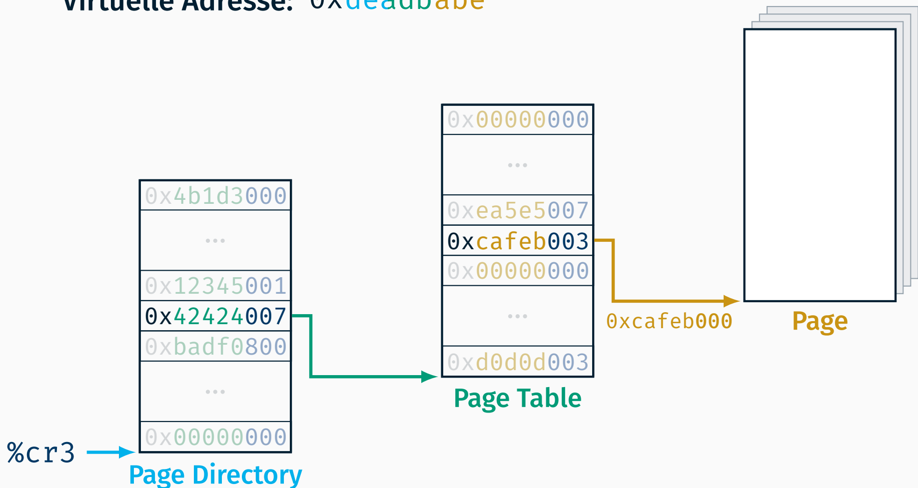


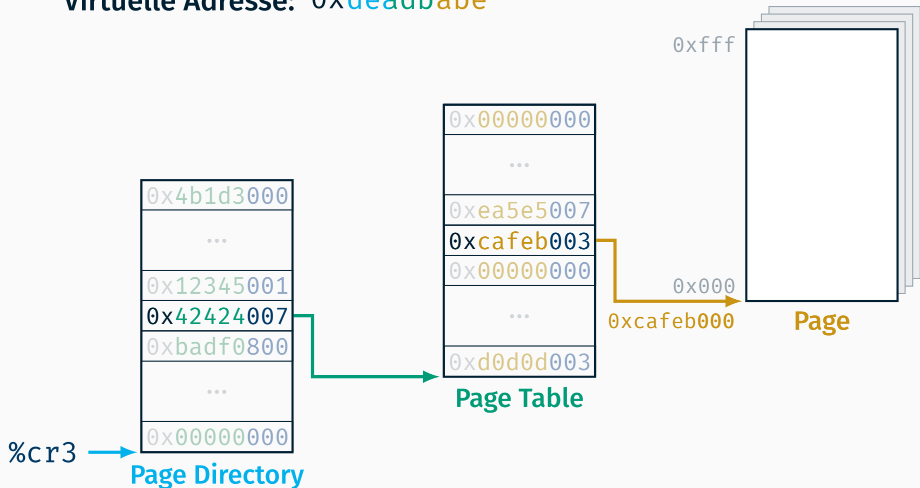
Virtuelle Adresse: 0xdeadbabe



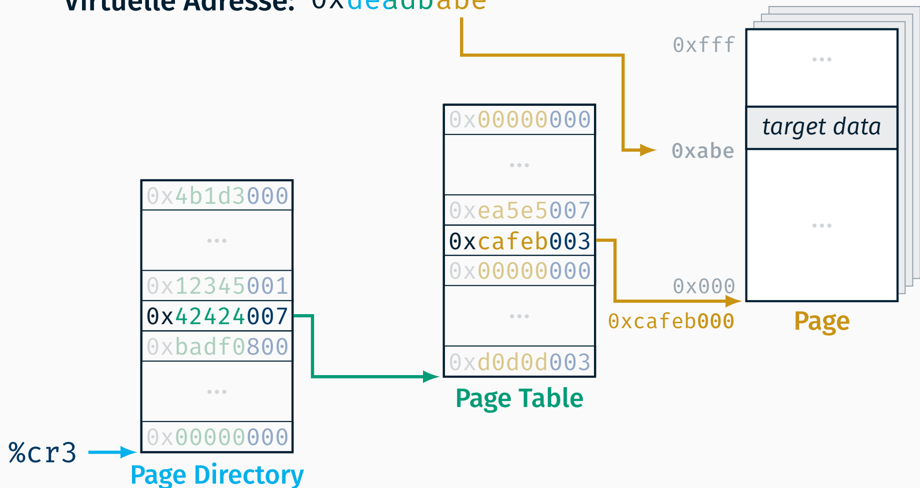
Adressumsetzung am Beispiel

oder: *Wo muss ich hin?*




Virtuelle Adresse: 0xdeadbabe
0xfff
0x4b1d3000
...
0x12345001
0x42424007
0xbadf0800
...
0x00000000
%cr3
Page Directory
0x00000000
...
0xea5e5007
0xcafeb003
0x00000000
...
0xd0d0d003
Page Table
0x000
0xcafeb000
Page

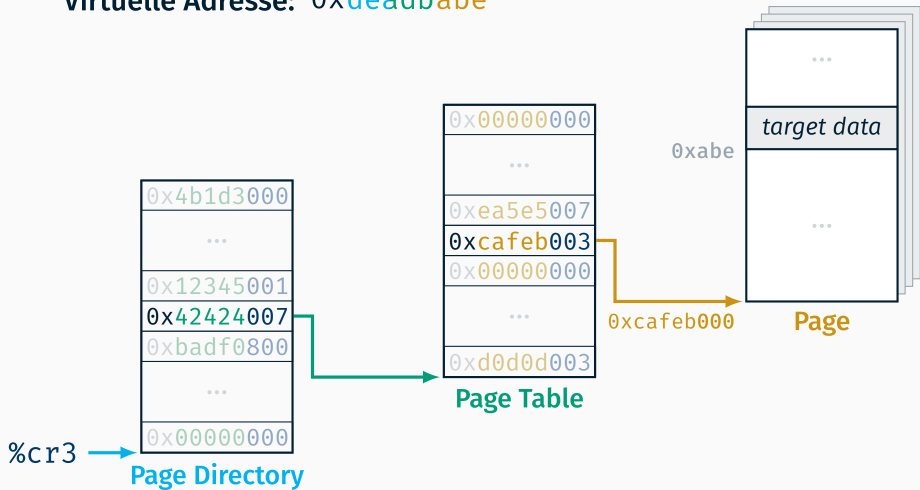
Adressumsetzung am Beispiel
oder: *Wo muss ich hin?*



Adressumsetzung am Beispiel

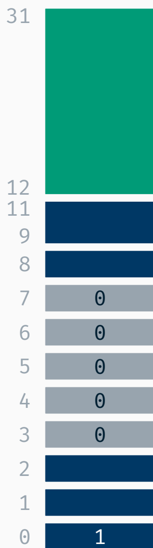
oder: *Wo muss ich hin?*

Virtuelle Adresse: `0xdeadbabe`



Physikalische Adresse: `0xcafebabe`

Eintrag im Seitenverzeichnis (Page-Directory Entry)



Adresse der Seitentabelle, welche an einer 4 KiB-Grenze ausgerichtet sein muss

Available – zur freien Verwendung

Global: Bei 4 KiB Seiten ignoriert (freie Verwendung)

Page Size: 4 KiB (0) oder 4 MiB (1) Seiten

ignoriert

Accessed: (1) falls die Zielseite verwendet wurde

Page-Level Cache Disable: (1) deaktiviert Caching

Page-Level Write Through: (1) aktiviert WT Caching

User Mode: (1) erlaubt Zugriff aus Ring 3

Writeable: (1) erlaubt schreiben (sonst nur lesen)

Present: Eintrag aktiv (1) oder inaktiv (0)

Eintrag im Seitenverzeichnis (Page-Directory Entry)

Bit	Wert	Bedeutung
31–12		**Adresse der Seitentabelle**, welche an einer 4 KiB-Grenze ausgerichtet sein muss
11–9		**Available** – zur freien Verwendung
8		**Global:** Bei 4 KiB Seiten ignoriert (freie Verwendung)
7	0	**Page Size:** 4 KiB (0) oder 4 MiB (1) Seiten *ignoriert*
6	0	
5	0	**Accessed:** (1) falls die Zielseite verwendet wurde
4	0	**Page-Level Cache Disable:** (1) deaktiviert Caching
3	0	**Page-Level Write Through:** (1) aktiviert WT Caching
2		**User Mode:** (1) erlaubt Zugriff aus Ring 3
1		**Writeable:** (1) erlaubt schreiben (sonst nur lesen)
0	1	**Present:** Eintrag aktiv (1) oder inaktiv (0)

Auch das Seitenverzeichnis selbst muss an 4 KiB ausgerichtet sein!

Eintrag in der Seitentabelle (Page-Table Entry)

Bits	Wert	Bedeutung
31–12		**Physikalische Adresse** der durch diesen Eintrag referenzierten 4 KiB Seite
11–9		**Available** – zur freien Verwendung
8	0	**Global**: Verhindert TLB Aktualisierung
7	0	falls **PAT** aktiv: Speichertyp
6	0	**Dirty**: (1) falls auf die Zielseite geschrieben wurde
5	0	**Accessed**: (1) falls die Zielseite verwendet wurde
4	0	**Page-Level Cache Disable**: (1) deaktiviert Caching
3	0	**Page-Level Write Through**: (1) aktiviert WT Caching
2		**User Mode:** (1) erlaubt Zugriff aus Ring 3
1		**Writeable:** (1) erlaubt schreiben (sonst nur lesen)
0	1	**Present:** Eintrag aktiv (1) oder inaktiv (0)

Details im Intel Software Developer's Manual Vol. 3A / 4.3 32-Bit Paging

Kontrollregister 0 (`%cr0`)

Bit	Wert	Bedeutung
31	1	**Paging** aktiv (1) oder inaktiv (0)
30	0	**Cache Disable**: (1) deaktiviert Caching
29	0	**Not Write Through**: (1) deaktiviert WT Caching
28–19		**Reserviert**
18	0	**Alignment Mask**: (1) aktiviert Prüfung der Ausrichtung
17		**Reserviert**
16	0 / 1	**Write Protect:** (0) erlaubt Schreiben von `ro`-Seiten in Ring 0
15–6		**Reserviert**
5	0	**Numeric Error**: (1) aktiviert FPU Ausnahmebehandlung
4	0	**Extension Type**: für Koprozessor (Modelabhängig)
3	0	**Task Switched** } für FPU Kontextsicherung
2	0	**Emulation** } für FPU Kontextsicherung
1	0	**Monitor Coprocessor** } für FPU Kontextsicherung
0	1	**Protection Enable:** Real (0) oder Protected (1) Mode

Kontrollregister 0 (`%cr0`)

Bit	Wert	Bedeutung
31	1	**Paging** aktiv (1) oder inaktiv (0)
30	0	**Cache Disable**: (1) deaktiviert Caching
29	0	**Not Write Through**: (1) deaktiviert WT Caching
28–19		**Reserviert**
18	0	**Alignment Mask**: (1) aktiviert Prüfung der Ausrichtung
17		**Reserviert**
16	0 / 1	**Write Protect:** (0) erlaubt Schreiben von `ro`-Seiten in Ring 0
15–6		**Reserviert**
5	0	**Numeric Error**: (1) aktiviert FPU Ausnahmebehandlung
4	0	**Extension Type**: für Koprozessor (Modelabhängig)
3	0	**Task Switched** – für FPU Kontextsicherung
2	0	**Emulation** – für FPU Kontextsicherung
1	0	**Monitor Coprocessor** – für FPU Kontextsicherung
0	1	**Protection Enable:** Real (0) oder Protected (1) Mode

Details im Intel Software Developer's Manual Vol. 3A / 2.5 Control Registers

Kontrollregister 1 (`%cr1`)

31

Reserviert. Für was auch immer.

0

Kontrollregister 2 (%cr2)

Page-Fault Linear Address
Beinhaltet bei einem Seitenfehler die virtuelle Adresse, die den Fehler verursacht hat.

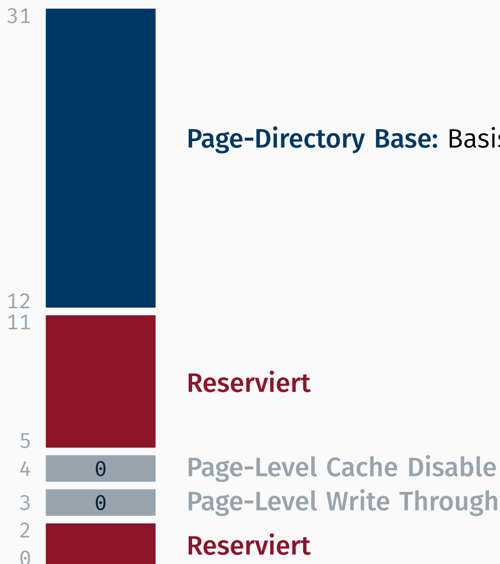
Kontrollregister 2 (`%cr2`)

Page-Fault Linear Address

Beinhaltet bei einem Seitenfehler die virtuelle Adresse, die den Fehler verursacht hat.

(Noch) nicht notwendig in dieser Übung,
aber kann das Entkäfern deutlich vereinfachen!

Kontrollregister 3 (%cr3)



Page-Directory Base: Basisadresse des Seitenverzeichnisses

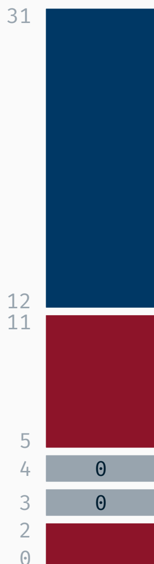
Reserviert

Page-Level Cache Disable

Page-Level Write Through

Reserviert

Kontrollregister 3 (`%cr3`)



Page-Directory Base: Basisadresse des Seitenverzeichnisses
(deshalb Seitenverzeichnis auf 4 KiB ausgerichtet!)

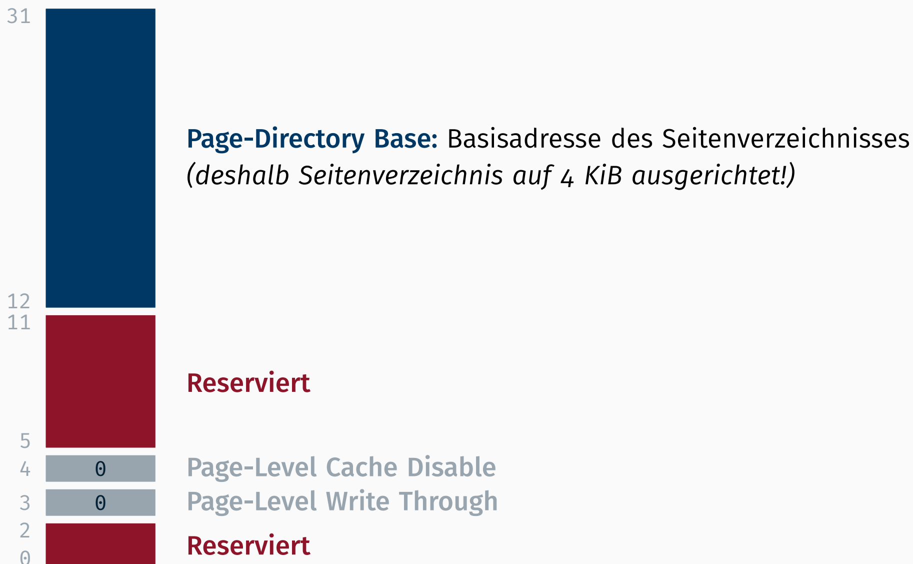
Reserviert

Page-Level Cache Disable

Page-Level Write Through

Reserviert

Kontrollregister 3 (`%cr3`)



Hinweis: Mit Schreiben von `%cr3` wird TLB gespült

weitere Kontrollregister

`%cr4` Steuerung von architekturabhängigen Erweiterungen wie **Page Size Extension** (4 MiB große Seiten) oder **Physical Address Extension** (erlaubt mehr als 4 GiB Speicher unter 32 Bit).

`%cr5` *reserviert*

`%cr6` *reserviert*

`%cr7` *reserviert*

`%cr8` nur für Long Mode (64 bit), steuert Zugriff auf *Task Priority Register*

weitere Kontrollregister

`%cr4` Steuerung von architekturabhängigen Erweiterungen wie **Page Size Extension** (4 MiB große Seiten) oder **Physical Address Extension** (erlaubt mehr als 4 GiB Speicher unter 32 Bit).

`%cr5` *reserviert*

`%cr6` *reserviert*

`%cr7` *reserviert*

`%cr8` nur für Long Mode (64 bit), steuert Zugriff auf *Task Priority Register*

Aber: Nicht wichtig für uns, wir ignorieren diese in der Übung.

Trennung von Anwendungen und Kernel

Anwendungen in eigenen Binärdateien

Bisher:

- Anwendungscode und Daten sind mit Kernelobjekten vermischt
- Alles in einer großen *System ELF*-Datei

Anwendungen in eigenen Binärdateien

Bisher:

- Anwendungscode und Daten sind mit Kernelobjekten vermischt
- Alles in einer großen *System ELF*-Datei

Ab heute: Initiales Speicherabbild (`initrd`)

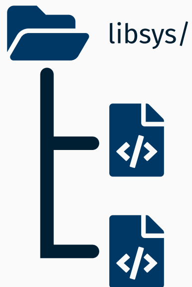
Aufteilung des Codes in unterschiedliche Verzeichnisse:

`kernel/` liefert weiterhin *System ELF*

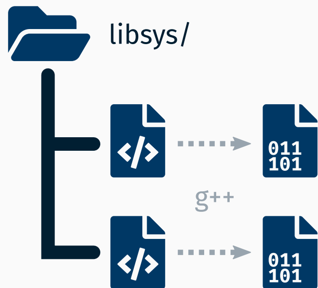
`libsys/` enthält Funktionen zur Anwendungsunterstützung (Systemaufrufstümpfe)

`user/` enthält **mehrere** Anwendungen
Jede Anwendung wird gegen `libsys` gelinkt und zu eigenem ELF kompiliert

libsys

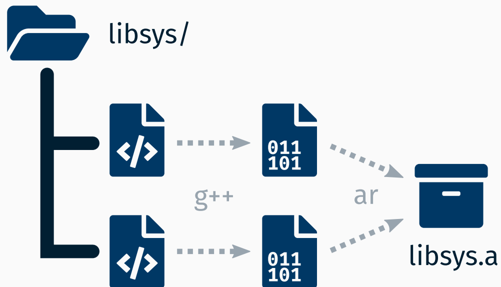


libsys

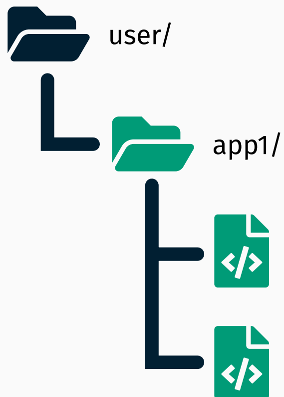


libsys

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

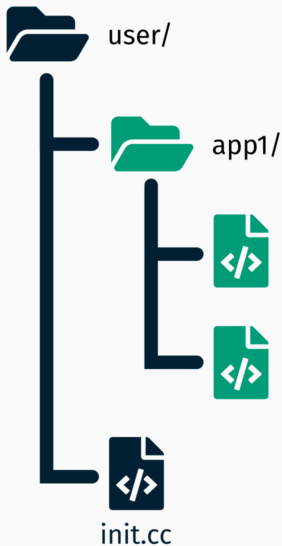


Anwendung

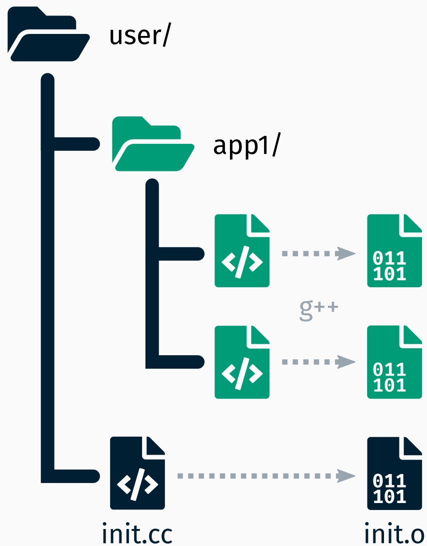


Anwendung

Initialisierung globaler Objekte durch `user/init.cc`

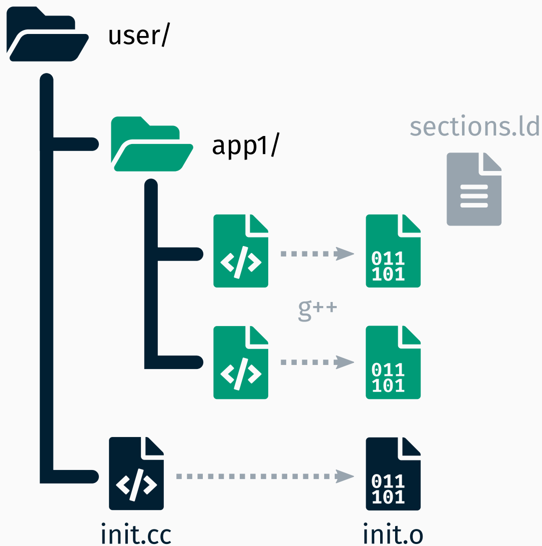


Anwendung



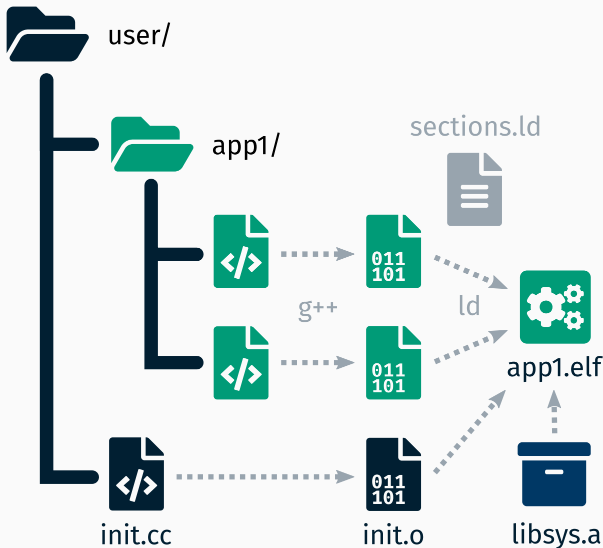
Anwendung

Linkerskript `user/sections.ld` mit `0x2000000` als Einsprungpunkt



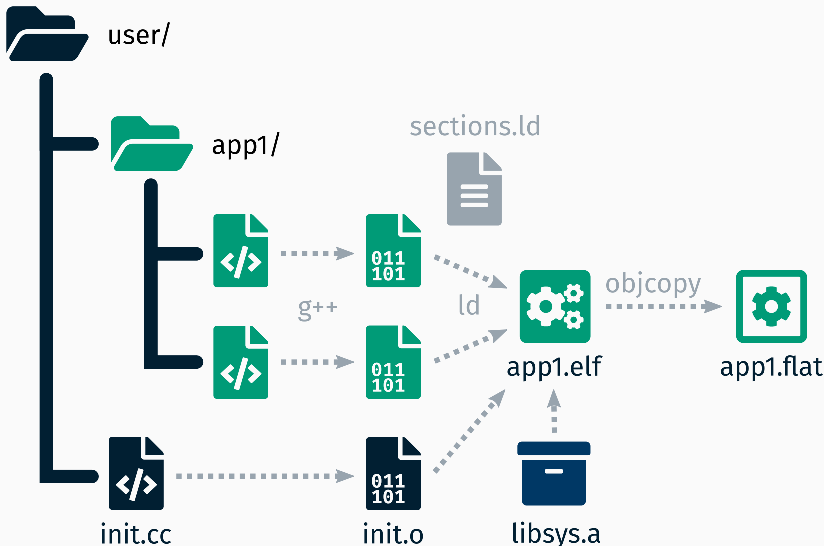
Anwendung

`$ ld -T sections.ld -o app1.elf $LDFLAGS init.o [...]`



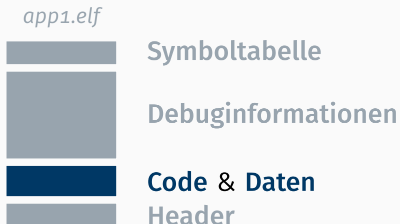
Anwendung

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



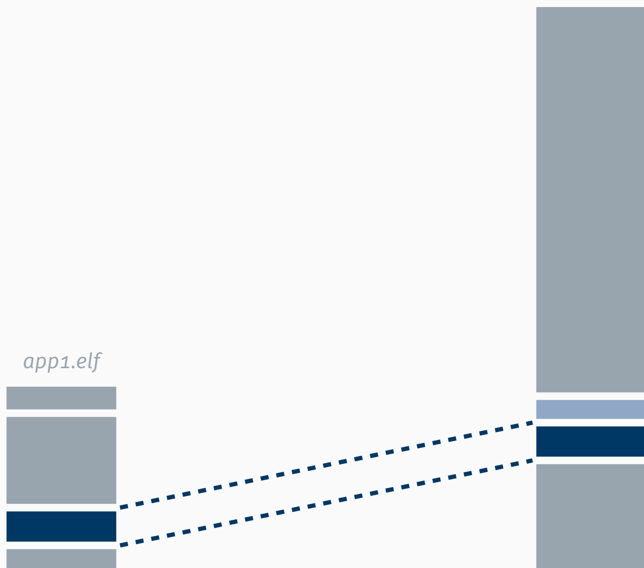
Erstellung einer flachen Binärdatei

Man nehme eine ELF-Datei



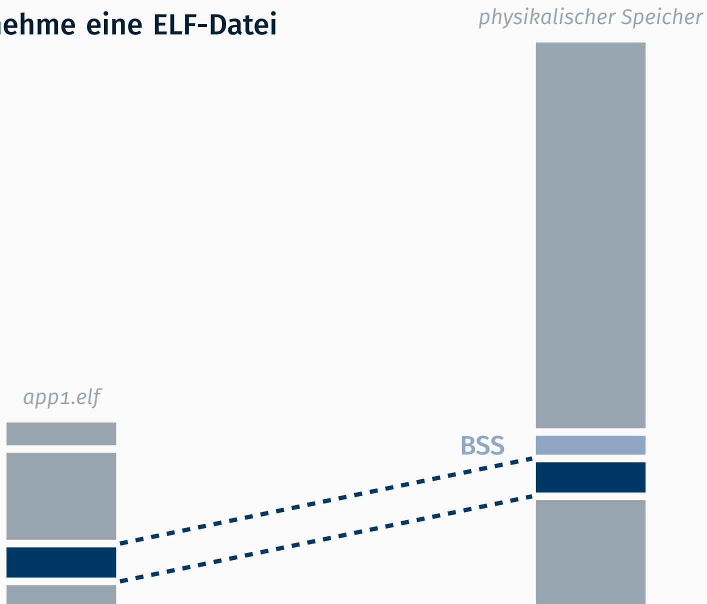
Erstellung einer flachen Binärdatei

Man nehme eine ELF-Datei



Erstellung einer flachen Binärdatei

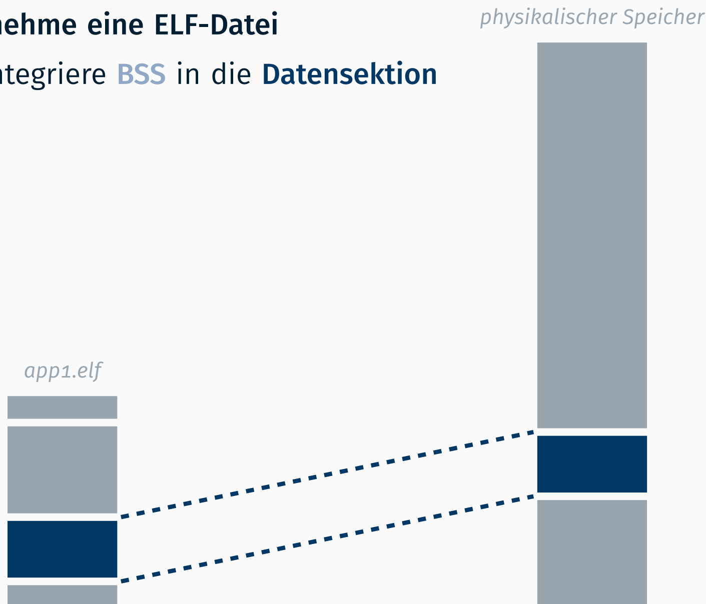
Man nehme eine ELF-Datei



Erstellung einer flachen Binärdatei

Man nehme eine ELF-Datei

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



Erstellung einer flachen Binärdatei

Man nehme eine ELF-Datei

1. integriere **BSS** in die **Datensektion**
   (`--set-section-flags .bss=alloc,load,contents`)

Erstellung einer flachen Binärdatei

Man nehme eine ELF-Datei

1. integriere **BSS** in die **Datensektion**
   (`--set-section-flags .bss=alloc,load,contents`)
2. werfe alles außer **Code** & **Datensektion** weg

app1.flat

Erstellung einer flachen Binärdatei

Man nehme eine ELF-Datei

1. integriere **BSS** in die **Datensektion**
   (`--set-section-flags .bss=alloc,load,contents`)
2. werfe alles außer **Code** & **Datensektion** weg

Zack fertig: **Flat Binary**

app1.flat

Initiales Speicherabbild

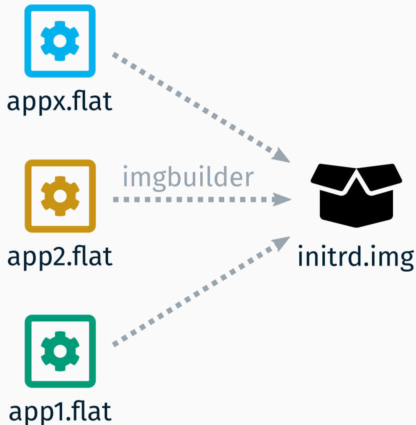
appx.flat

app2.flat

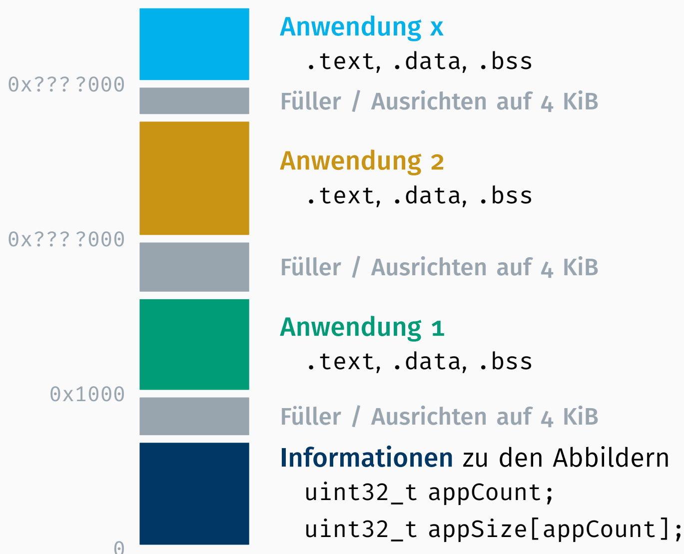
app1.flat

Initiales Speicherabbild

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



imgbuilder Format



We are not done yet – Erweiterte Speicherverwaltung für 7.5 ECTS

Zusätzliche Systemaufrufe

Implementierung der Systemaufrufe

- `void* map(size_t size, void* addr)`
  blendet Seiten der Größe `size` im aktuellen Adressraum an `addr` ein (oder an geeigneter Lücke falls `NULL`)
- `void exit()`
  Beendet aktuellen Prozess und räumt komplett auf

Optional: ELF und TAR

Executable and Linking Format

app1.elf

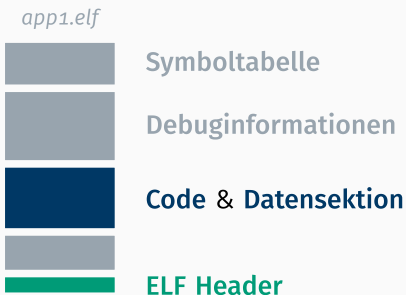
Symboltabelle

Debuginformationen

Code & Datensektion

Header

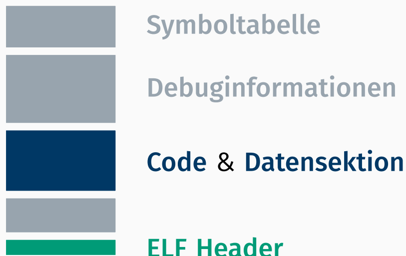
Executable and Linking Format



Executable and Linking Format

ELF Header

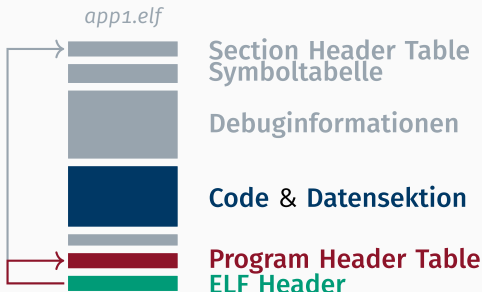
- Erkennung durch Wert `{ 0x7f, 'E', 'L', 'F' }`
- gibt u.a. Architektur, ABI und Typ (***Executable***, *Relocatable*, *Shared*) an



Executable and Linking Format

ELF Header

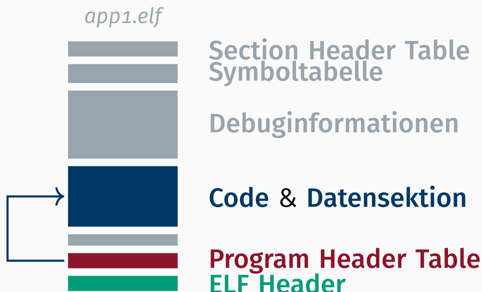
- Erkennung durch Wert `{ 0x7f, 'E', 'L', 'F' }`
- gibt u.a. Architektur, ABI und Typ (***Executable***, *Relocatable*, *Shared*) an
- beinhaltet Einsprungsadresse und Position sowie Größe von ***Program*** & *Section Header Table*



Executable and Linking Format

Program Header Table

- Beschreibt die einzelnen Programmsegmente



Executable and Linking Format

Program Header Table

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

Executable and Linking Format

Program Header Table

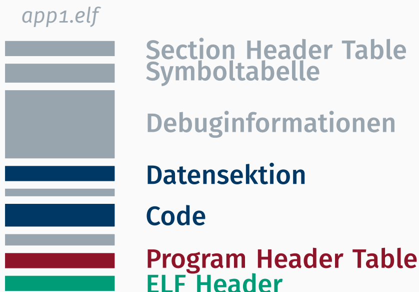
- Beschreibt die einzelnen Programmsegmente
- jeweils mit Position & Größe in ELF-Datei und im Zielspeicher
- verschiedene Typen: ***LOAD***, *DYNAMIC*, *INTERP*, ...
- und Attribute wie les-, schreib- und ausführbar

Executable and Linking Format

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

Elf-Datei-Typ ist EXEC (ausführbare Datei)
Entry point 0x2000000
There are 3 program headers, starting at offset 52

Typ          Offset     VirtAdr      DateiGr   SpeiGr    Flg   Ausr.
LOAD         0x001000   0x02000000   0x0120c   0x0120c   RWE   0x1000
LOAD         0x003000   0x02002000   0x00600   0x03000   RW    0x1000
GNU_STACK    0x000000   0x00000000   0x00000   0x00000   RW    0x10
```

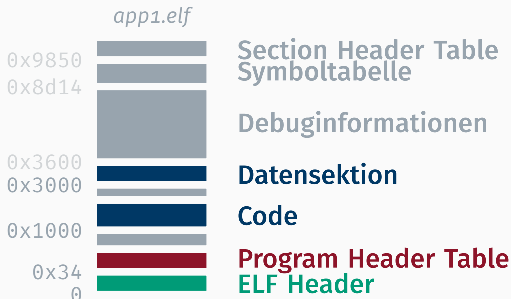


Executable and Linking Format

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

Elf-Datei-Typ ist EXEC (ausführbare Datei)
Entry point 0x2000000
There are 3 program headers, starting at offset 52

Typ          Offset     VirtAdr      DateiGr   SpeiGr    Flg   Ausr.
LOAD         0x001000   0x02000000   0x0120c   0x0120c   RWE   0x1000
LOAD         0x003000   0x02002000   0x00600   0x03000   RW    0x1000
GNU_STACK    0x000000   0x00000000   0x00000   0x00000   RW    0x10
```



Bandarchivierer (Tape ARchiver)

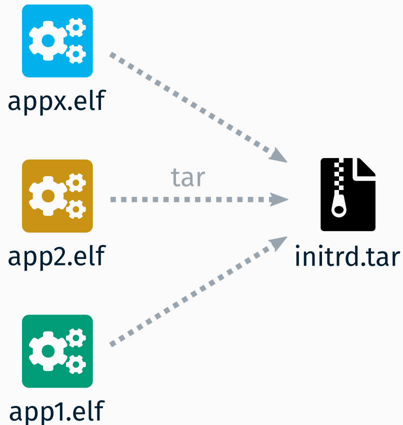
appx.elf

app2.elf

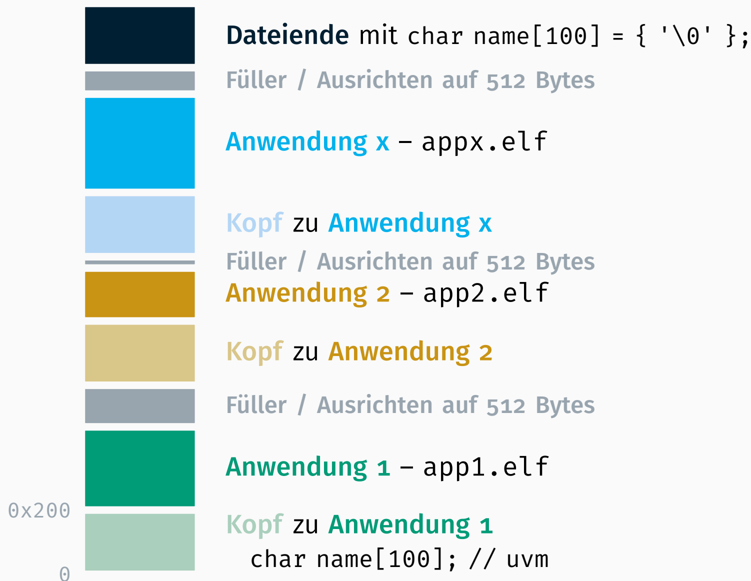
app1.elf

Bandarchivierer (Tape ARchiver)

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



TAR Format



Fragen?

Fragen?

Lust auf einen *Meltdown* & *Fallout* Vortrag übernächste Woche?