Übung zu Betriebssystembau

(Ur)Laden des x86er

17. Dezember 2024

Alexander Krause

Arbeitsgruppe Systemsoftware
Technische Universität Dortmund

(Mit Material vom Lehrstuhl 4 der FAU)

WARNUNG

Die folgende Präsentation zeigt die Untiefen sowie historisch bedingten Grausamkeiten der *Intel x86*-Architektur und deren Vorgänger auf.
Unter Berücksichtigung der Weisungen von Experten ist jedoch heutzutage ein weitestgehend gefahrloser Einstieg in die Betriebssystementwicklung möglich.
Sollte jedoch dennoch eigenmächtig versucht werden, die nachfolgend dargestellten Programmierstunts nachzuimplementieren, so
übernimmt die Arbeitsgruppe Systemsoftware keine Haftung für
dadurch ausgelöste psychische Belastungen oder Erkrankungen.

Es war einmal ein kleiner Chip ...

SPI Flash Memory (ROM)

Quelle: allegro.pl

Es war einmal ein kleiner Chip ...

BIOS (Basic Input/Output System)

- von ROM geladen
- automatischer Start
- Power-On Self-Test
- Hardwareinitialisierung
  - u.a. Bootmedium

Es war einmal ein kleiner Chip ...

BIOS (Basic Input/Output System)

- von ROM geladen
- automatischer Start
- Power-On Self-Test
- Hardwareinitialisierung
  - u.a. Bootmedium

oder UEFI (Unified Extensible Firmware Interface)

Es war einmal ein kleiner Chip ...

BIOS lädt **Bootsektor** von Bootmedium

Es war einmal ein kleiner Chip …

BIOS lädt **Bootsektor** von Bootmedium

- 1. Sektor von **Floppy**, der *VBR (Volume Boot Record)*
- 1. Sektor von **Festplatte**
  - **MBR** (Master Boot Record)
  - oder **GPT** (GUID-Partitionstabelle, hat *protective MBR*)
- ebenso bei **USB** (wird meist als Festplatte emuliert)

Es war einmal ein kleiner Chip …

BIOS lädt **Bootsektor** von Bootmedium

- 1. Sektor von **Floppy**, der *VBR (Volume Boot Record)*
- 1. Sektor von **Festplatte**
  - **MBR** (Master Boot Record)
  - oder **GPT** (GUID-Partitionstabelle, hat *protective MBR*)
- ebenso bei **USB** (wird meist als Festplatte emuliert)
- CD, DVD und BD (**ISO 9660**)
  - **Boot Record** (nach *El Torito*) zeigt auf Bootprogramm
  - ggf. auch **hybrid** (für USB): 1. Sektor wie Festplatte (da ersten 32K als *System Area* reserviert/ungenutzt)

Es war einmal ein kleiner Chip …

BIOS lädt **Bootsektor** von Bootmedium

- 1. Sektor von **Floppy**, der *VBR (Volume Boot Record)*
- 1. Sektor von **Festplatte**
  - **MBR** (Master Boot Record)
  - oder **GPT** (GUID-Partitionstabelle, hat *protective MBR*)
- ebenso bei **USB** (wird meist als Festplatte emuliert)
- CD, DVD und BD (**ISO 9660**)
  - **Boot Record** (nach *El Torito*) zeigt auf Bootprogramm
  - ggf. auch **hybrid** (für USB): 1. Sektor wie Festplatte (da ersten 32K als *System Area* reserviert/ungenutzt)
- via Netzwerk: **Preboot Execution Environment (PXE)**

```
$ sudo dd if=/dev/sda of=/tmp/mbr.bin bs=512 count=1 && xxd /tmp/mbr.bin
```

```
$ sudo dd if=/dev/sda of=/tmp/mbr.bin bs=512 count=1 && xxd /tmp/mbr.bin
i0000:   eb63 9010 8ed0 bc00 b0b8 0000 8ed8 8ec0                  .c..............
0010:    fbbe 007c bf00 06b9 0002 f3a4 ea21 0600                  ...|.........!..
0020:    00be be07 3804 750b 83c6 1081 fefe 0775                  ....8.u........u
0030:    f3eb 16b4 02b0 01bb 007c b280 8a74 018b                  .........|...t..
0040:    4c02 cd13 ea00 7c00 00eb fe00 0000 0000                  L.....|.........
0050:    0000 0000 0000 0000 0000 0080 0100 0000                  ................
0060:    0000 0000 fffa 9090 f6c2 8074 05f6 c270                  ...........t...p
0070:    7402 b280 ea79 7c00 0031 c08e d88e d0bc                  t....y|..1......
0080:    0020 fba0 647c 3cff 7402 88c2 52bb 1704                  . ..d|<.t...R...
0090:    f607 0374 06be 887d e817 01be 057c b441                  ...t...}.....|.A
00a0:    bbaa 55cd 135a 5272 3d81 fb55 aa75 3783                  ..U..ZRr=..U.u7.
00b0:    e101 7432 31c0 8944 0440 8844 ff89 4402                  ..t21..D.@.D..D.
00c0:    c704 1000 668b 1e5c 7c66 895c 0866 8b1e                  ....f..\|f.\.f..
00d0:    607c 6689 5c0c c744 0600 70b4 42cd 1372                  `|f.\..D..p.B..r
00e0:    05bb 0070 eb76 b408 cd13 730d 5a84 d20f                  ...p.v....s.Z...
00f0:    83d0 00be 937d e982 0066 0fb6 c688 64ff                  .....}...f....d.
0100:    4066 8944 040f b6d1 c1e2 0288 e888 f440                  @f.D...........@
0110:    8944 080f b6c2 c0e8 0266 8904 66a1 607c                  .D.......f..f.`|
0120:    6609 c075 4e66 a15c 7c66 31d2 66f7 3488                  f..uNf.\|f1.f.4.
0130:    d131 d266 f774 043b 4408 7d37 fec1 88c5                  .1.f.t.;D.}7....
0140:    30c0 c1e8 0208 c188 d05a 88c6 bb00 708e                  0........Z....p.
0150:    c331 dbb8 0102 cd13 721e 8cc3 601e b900                  .1......r...`...
0160:    018e db31 f6bf 0080 8ec6 fcf3 a51f 61ff                  ...1..........a.
0170:    265a 7cbe 8e7d eb03 be9d 7de8 3400 bea2                  &Z|..}....}.4...
0180:    7de8 2e00 cd18 ebfe 4752 5542 2000 4765                  }.......GRUB .Ge
0190:    6f6d 0048 6172 6420 4469 736b 0052 6561                  om.Hard Disk.Rea
01a0:    6400 2045 7272 6f72 0d0a 00bb 0100 b40e                  d. Error........
01b0:    cd10 ac3c 0075 f4c3 43a9 40ec 0000 8020                  ...<.u..C.@....
01c0:    2100 83fe ffff 0008 0000 dbe0 210b 00fe                  !...........!...
01d0:    ffff 82fe ffff dbe8 210b 332d de00 00fe                  ........!.3-....
01e0:    ffff 83fe ffff 0e16 000c a257 7068 0000                  ...........Wph..
01f0:    0000 0000 0000 0000 0000 0000 0000 55aa                  ..............U.
```

```
$ sudo dd if=/dev/sda of=/tmp/mbr.bin bs=512 count=1 && xxd /tmp/mbr.bin
i0000:   eb63 9010 8ed0 bc00 b0b8 0000 8ed8 8ec0                  .c..............
0010:    fbbe 007c bf00 06b9 0002 f3a4 ea21 0600                  ...|.........!..
0020:    00be be07 3804 750b 83c6 1081 fefe 0775                  ....8.u........u
0030:    f3eb 16b4 02b0 01bb 007c b280 8a74 018b                  .........|...t..
0040:    4c02 cd13 ea00 7c00 00eb fe00 0000 0000                  L.....|.........
0050:    0000 0000 0000 0000 0000 0080 0100 0000                  ................
0060:    0000 0000 fffa 9090 f6c2 8074 05f6 c270                  ...........t...p
0070:    7402 b280 ea79 7c00 0031 c08e d88e d0bc                  t....y|..1......
0080:    0020 fba0 647c 3cff 7402 88c2 52bb 1704                  . ..d|<.t...R...
0090:    f607 0374 06be 887d e817 01be 057c b441                  ...t...}.....|.A
00a0:    bbaa 55cd 135a 5272 3d81 fb55 aa75 3783                  ..U..ZRr=..U.u7.
00b0:    e101 7432 31c0 8944 0440 8844 ff89 4402                  ..t21..D.@.D..D.
00c0:    c704 1000 668b 1e5c 7c66 895c 0866 8b1e                  ....f..\|f.\.f..
00d0:    607c 6689 5c0c c744 0600 70b4 42cd 1372                  `|f.\..D..p.B..r
00e0:    05bb 0070 eb76 b408 cd13 730d 5a84 d20f                  ...p.v....s.Z...
00f0:    83d0 00be 937d e982 0066 0fb6 c688 64ff                  .....}...f....d.
0100:    4066 8944 040f b6d1 c1e2 0288 e888 f440                  @f.D...........@
0110:    8944 080f b6c2 c0e8 0266 8904 66a1 607c                  .D.......f..f.`|
0120:    6609 c075 4e66 a15c 7c66 31d2 66f7 3488                  f..uNf.\|f1.f.4.
0130:    d131 d266 f774 043b 4408 7d37 fec1 88c5                  .1.f.t.;D.}7....
0140:    30c0 c1e8 0208 c188 d05a 88c6 bb00 708e                  0........Z....p.
0150:    c331 dbb8 0102 cd13 721e 8cc3 601e b900                  .1......r...`...
0160:    018e db31 f6bf 0080 8ec6 fcf3 a51f 61ff                  ...1..........a.
0170:    265a 7cbe 8e7d eb03 be9d 7de8 3400 bea2                  &Z|..}....}.4...
0180:    7de8 2e00 cd18 ebfe 4752 5542 2000 4765                  }.......GRUB .Ge
0190:    6f6d 0048 6172 6420 4469 736b 0052 6561                  om.Hard Disk.Rea
01a0:    6400 2045 7272 6f72 0d0a 00bb 0100 b40e                  d. Error........
01b0:    cd10 ac3c 0075 f4c3 43a9 40ec 0000 8020                  ...<.u..C.@....
01c0:    2100 83fe ffff 0008 0000 dbe0 210b 00fe                  !...........!...
01d0:    ffff 82fe ffff dbe8 210b 332d de00 00fe                  ........!.3-....          Boot Signatur
01e0:    ffff 83fe ffff 0e16 000c a257 7068 0000                  ...........Wph..
01f0:    0000 0000 0000 0000 0000 0000 0000 55aa                  ..............U.
```

```
$ sudo dd if=/dev/sda of=/tmp/mbr.bin bs=512 count=1 && xxd /tmp/mbr.bin
i0000:   eb63 9010 8ed0 bc00 b0b8 0000 8ed8 8ec0                  .c..............
0010:    fbbe 007c bf00 06b9 0002 f3a4 ea21 0600                  ...|.........!..
0020:    00be be07 3804 750b 83c6 1081 fefe 0775                  ....8.u........u
0030:    f3eb 16b4 02b0 01bb 007c b280 8a74 018b                  .........|...t..
0040:    4c02 cd13 ea00 7c00 00eb fe00 0000 0000                  L.....|.........
0050:    0000 0000 0000 0000 0000 0080 0100 0000                  ................
0060:    0000 0000 fffa 9090 f6c2 8074 05f6 c270                  ...........t...p
0070:    7402 b280 ea79 7c00 0031 c08e d88e d0bc                  t....y|..1......
0080:    0020 fba0 647c 3cff 7402 88c2 52bb 1704                  . ..d|<.t...R...
0090:    f607 0374 06be 887d e817 01be 057c b441                  ...t...}.....|.A
00a0:    bbaa 55cd 135a 5272 3d81 fb55 aa75 3783                  ..U..ZRr=..U.u7.
00b0:    e101 7432 31c0 8944 0440 8844 ff89 4402                  ..t21..D.@.D..D.
00c0:    c704 1000 668b 1e5c 7c66 895c 0866 8b1e                  ....f..\|f.\.f..
00d0:    607c 6689 5c0c c744 0600 70b4 42cd 1372                  `|f.\..D..p.B..r
00e0:    05bb 0070 eb76 b408 cd13 730d 5a84 d20f                  ...p.v....s.Z...
00f0:    83d0 00be 937d e982 0066 0fb6 c688 64ff                  .....}...f....d.
0100:    4066 8944 040f b6d1 c1e2 0288 e888 f440                  @f.D...........@
0110:    8944 080f b6c2 c0e8 0266 8904 66a1 607c                  .D.......f..f.`|
0120:    6609 c075 4e66 a15c 7c66 31d2 66f7 3488                  f..uNf.\|f1.f.4.
0130:    d131 d266 f774 043b 4408 7d37 fec1 88c5                  .1.f.t.;D.}7....
0140:    30c0 c1e8 0208 c188 d05a 88c6 bb00 708e                  0........Z....p.
0150:    c331 dbb8 0102 cd13 721e 8cc3 601e b900                  .1......r...`...
0160:    018e db31 f6bf 0080 8ec6 fcf3 a51f 61ff                  ...1..........a.
0170:    265a 7cbe 8e7d eb03 be9d 7de8 3400 bea2                  &Z|..}....}.4...
0180:    7de8 2e00 cd18 ebfe 4752 5542 2000 4765                  }.......GRUB .Ge
0190:    6f6d 0048 6172 6420 4469 736b 0052 6561                  om.Hard Disk.Rea
01a0:    6400 2045 7272 6f72 0d0a 00bb 0100 b40e                  d. Error........
01b0:    cd10 ac3c 0075 f4c3 43a9 40ec 0000 8020                  ...<.u..C.@....          Partitionstabelle
01c0:    2100 83fe ffff 0008 0000 dbe0 210b 00fe                  !...........!...         (Festplatte)
01d0:    ffff 82fe ffff dbe8 210b 332d de00 00fe                  ........!.3-....         Boot Signatur
01e0:    ffff 83fe ffff 0e16 000c a257 7068 0000                  ...........Wph..
01f0:    0000 0000 0000 0000 0000 0000 0000 55aa                  ..............U.
```

```
$ sudo dd if=/dev/sda of=/tmp/mbr.bin bs=512 count=1 && xxd /tmp/mbr.bin
i0000:   eb63 9010 8ed0 bc00 b0b8 0000 8ed8 8ec0   .c..............
0010:    fbbe 007c bf00 06b9 0002 f3a4 ea21 0600   ...|.........!..
0020:    00be be07 3804 750b 83c6 1081 fefe 0775   ....8.u........u          Bootstrap
0030:    f3eb 16b4 02b0 01bb 007c b280 8a74 018b   .........|...t..          Programm
0040:    4c02 cd13 ea00 7c00 00eb fe00 0000 0000   L.....|.........
0050:    0000 0000 0000 0000 0000 0080 0100 0000   ................
0060:    0000 0000 fffa 9090 f6c2 8074 05f6 c270   ...........t...p
0070:    7402 b280 ea79 7c00 0031 c08e d88e d0bc   t....y|..1......
0080:    0020 fba0 647c 3cff 7402 88c2 52bb 1704   . ..d|<.t...R...
0090:    f607 0374 06be 887d e817 01be 057c b441   ...t...}.....|.A
00a0:    bbaa 55cd 135a 5272 3d81 fb55 aa75 3783   ..U..ZRr=..U.u7.
00b0:    e101 7432 31c0 8944 0440 8844 ff89 4402   ..t21..D.@.D..D.
00c0:    c704 1000 668b 1e5c 7c66 895c 0866 8b1e   ....f..\|f.\.f..
00d0:    607c 6689 5c0c c744 0600 70b4 42cd 1372   `|f.\..D..p.B..r
00e0:    05bb 0070 eb76 b408 cd13 730d 5a84 d20f   ...p.v....s.Z...
00f0:    83d0 00be 937d e982 0066 0fb6 c688 64ff   .....}...f....d.
0100:    4066 8944 040f b6d1 c1e2 0288 e888 f440   @f.D...........@
0110:    8944 080f b6c2 c0e8 0266 8904 66a1 607c   .D.......f..f.`|
0120:    6609 c075 4e66 a15c 7c66 31d2 66f7 3488   f..uNf.\|f1.f.4.
0130:    d131 d266 f774 043b 4408 7d37 fec1 88c5   .1.f.t.;D.}7....
0140:    30c0 c1e8 0208 c188 d05a 88c6 bb00 708e   0........Z....p.
0150:    c331 dbb8 0102 cd13 721e 8cc3 601e b900   .1......r...`...
0160:    018e db31 f6bf 0080 8ec6 fcf3 a51f 61ff   ...1..........a.
0170:    265a 7cbe 8e7d eb03 be9d 7de8 3400 bea2   &Z|..}....}.4...
0180:    7de8 2e00 cd18 ebfe 4752 5542 2000 4765   }.......GRUB .Ge
0190:    6f6d 0048 6172 6420 4469 736b 0052 6561   om.Hard Disk.Rea
01a0:    6400 2045 7272 6f72 0d0a 00bb 0100 b40e   d. Error........
01b0:    cd10 ac3c 0075 f4c3 43a9 40ec 0000 8020   ...<.u..C.@....          Partitionstabelle
01c0:    2100 83fe ffff 0008 0000 dbe0 210b 00fe   !...........!...         (Festplatte)
01d0:    ffff 82fe ffff dbe8 210b 332d de00 00fe   ........!.3-....         Boot Signatur
01e0:    ffff 83fe ffff 0e16 000c a257 7068 0000   ...........Wph..
01f0:    0000 0000 0000 0000 0000 0000 0000 55aa   ..............U.
```

Chain Loading

Problem: Max. 446 Bytes für den Startup-Code

Chain Loading *(Eiertanz)*

Problem: Max. 446 Bytes für den Startup-Code
Lösung: Stufenweise (*Stages*) hochsteigen (*Boot Strapping*)

Chain Loading *(Eiertanz)*

Problem: Max. 446 Bytes für den Startup-Code
Lösung: Stufenweise (*Stages*) hochsteigen (*Boot Strapping*)

- **BIOS** lädt **Bootstrap Programm** aus **MBR**

Chain Loading *(Eiertanz)*

Problem: Max. 446 Bytes für den Startup-Code
Lösung: Stufenweise (*Stages*) hochsteigen (*Boot Strapping*)

- **BIOS** lädt **Bootstrap Programm** aus **MBR**
- **Bootstrap Programm** lädt Bootloader **Stage 1.5** aus weiteren Sektoren der Zielpartition (Lücke zwischen MBR und erster Partition)

Chain Loading *(Eiertanz)*

Problem: Max. 446 Bytes für den Startup-Code
Lösung: Stufenweise (*Stages*) hochsteigen (*Boot Strapping*)

- **BIOS** lädt **Bootstrap Programm** aus **MBR**
- **Bootstrap Programm** lädt Bootloader **Stage 1.5** aus weiteren Sektoren der Zielpartition (Lücke zwischen MBR und erster Partition)
- **Stage 1.5** bringt Dateiystemtreiber mit und lädt damit die nächste Bootloader **Stage 2**

Chain Loading *(Eiertanz)*

Problem: Max. 446 Bytes für den Startup-Code
Lösung: Stufenweise (*Stages*) hochsteigen (*Boot Strapping*)

- **BIOS** lädt **Bootstrap Programm** aus **MBR**
- **Bootstrap Programm** lädt Bootloader **Stage 1.5** aus weiteren Sektoren der Zielpartition (Lücke zwischen MBR und erster Partition)
- **Stage 1.5** bringt Dateiystemtreiber mit und lädt damit die nächste Bootloader **Stage 2**
- **Stage 2** ist endlich unser eigentlicher Bootloader

Chain Loading *(Eiertanz)*

Problem: Max. 446 Bytes für den Startup-Code
Lösung: Stufenweise (*Stages*) hochsteigen (*Boot Strapping*)

- **BIOS** lädt **Bootstrap Programm** aus **MBR**
- **Bootstrap Programm** lädt Bootloader **Stage 1.5** aus weiteren Sektoren der Zielpartition (Lücke zwischen MBR und erster Partition)
- **Stage 1.5** bringt Dateiystemtreiber mit und lädt damit die nächste Bootloader **Stage 2**
- **Stage 2** ist endlich unser eigentlicher Bootloader

(verkürztes Beispiel **GRUB Legacy**)

Chain Loading *(Eiertanz)*

Problem: Max. 446 Bytes für den Startup-Code

Lösung: Stufenweise (*Stages*) hochsteigen (*Boot Strapping*)

- **BIOS** lädt **Bootstrap Programm** aus **MBR**
- **Bootstrap Programm** lädt Bootloader **Stage 1.5** aus weiteren Sektoren der Zielpartition (Lücke zwischen MBR und erster Partition)
- **Stage 1.5** bringt Dateiystemtreiber mit und lädt damit die nächste Bootloader **Stage 2**
- **Stage 2** ist endlich unser eigentlicher Bootloader

(verkürztes Beispiel **GRUB Legacy**)

Chain Loading *(Eiertanz)*

Problem: Max. 446 Bytes für den Startup-Code

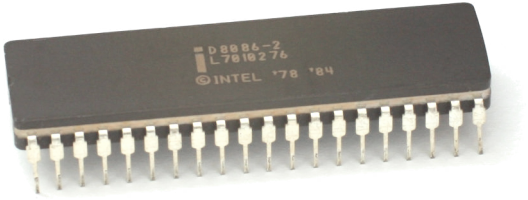
Lösung: Stufenweise (*Stages*) hochsteigen (*Boot Strapping*)

- **BIOS** lädt **Bootstrap Programm** aus **MBR**
- **Bootstrap Programm** lädt Bootloader **Stage 1.5** aus weiteren Sektoren der Zielpartition (Lücke zwischen MBR und erster Partition)
- **Stage 1.5** bringt Dateiystemtreiber mit und lädt damit die nächste Bootloader **Stage 2**
- **Stage 2** ist endlich unser eigentlicher Bootloader

(verkürztes Beispiel **GRUB Legacy**)

→ **Aber wie funktioniert das jetzt?**

In einem Land vor unserer Zeit ...



Quelle: Wikipedia

... begann das *reale* Leben.

Der 8086 – Real Mode

- 16 bit-CPU
- Speicher: max. 1 MB durch Segmentierung (20 bit Adressbus)
- BIOS Funktionen via Interrupts aufrufbar
- 14 Register,
  darunter Register für Segmentselektoren (ds = Datensegment)

„Moderne“ Prozessoren …

Quelle: Wikipedia

Quelle: Wikipedia

„Moderne“ Prozessoren …

Quelle: Wikipedia

Quelle: Wikipedia

- … laufen im `Protected Mode`,

„Moderne“ Prozessoren …

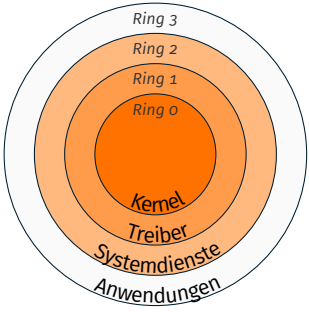
Quelle: Wikipedia

Quelle: Wikipedia

- … laufen im `Protected Mode`,
- starten aber **immer** im `Real Mode` (Abwärtskompatibilität)

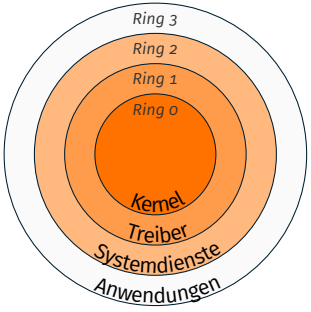
Protected Mode

Name aufgrund der vier Schutzringe (*Privilege Level*)



Protected Mode

Name aufgrund der vier Schutzringe
(*Privilege Level*)



80286 16 bit, 16 MB Speicher adressierbar
- Segementierung mit 24 bit Adressbus

80386 32 bit, 4 GB Speicher adressierbar
- Segementierung mit 32 bit Adressbus
- zusätzlich auch Paging (ab 80386DX)

Wechsel in 32 bit Protected-Mode

Ausgehend vom *Real Mode*

- Laden von weiterem StartUp-Code
- Interrupts (inklusive NMI) ausschalten
- A20 Gate aktivieren
- *Global Descriptor Table* laden
- *Protection Enable*-bit (PE) im Kontrollregister 0 (`cr0`) setzen

Wechsel in 32 bit Protected-Mode

Ausgehend vom *Real Mode*

- **Laden von weiterem StartUp-Code**
- Interrupts (inklusive NMI) ausschalten
- A20 Gate aktivieren
- *Global Descriptor Table* laden
- *Protection Enable*-bit (PE) im Kontrollregister 0 (`cr0`) setzen

Wechsel in 32 bit Protected-Mode

Ausgehend vom *Real Mode*

- Laden von weiterem StartUp-Code
- **Interrupts (inklusive NMI) ausschalten**
- A20 Gate aktivieren
- *Global Descriptor Table* laden
- *Protection Enable*-bit (PE) im Kontrollregister 0 (`cr0`) setzen

Wechsel in 32 bit Protected-Mode

Ausgehend vom *Real Mode*

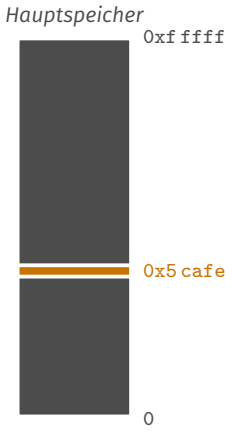
- Laden von weiterem StartUp-Code
- Interrupts (inklusive NMI) ausschalten
- **A20 Gate aktivieren**
- *Global Descriptor Table* laden
- *Protection Enable*-bit (PE) im Kontrollregister 0 (`cr0`) setzen

Der 8086 – Real Mode Segmentierung

- 1 MB Hauptspeicher
- 20 bit Adressbus
- aber 16 bit CPU (maximal `0xffff`)
- Zugriff via ds:eax
- Physikalische Adresse:
  Selektor * 0x10 + *Offset* = Zieldaten

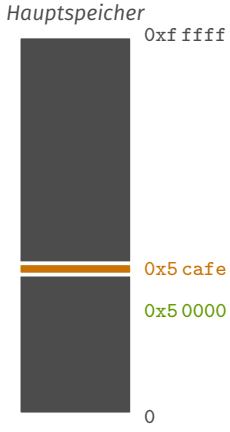
Der 8086 – Real Mode Segmentierung

Zugriff auf Zieldaten in `0x5cafe`



Zugriff auf Zieldaten in `0x5cafe`

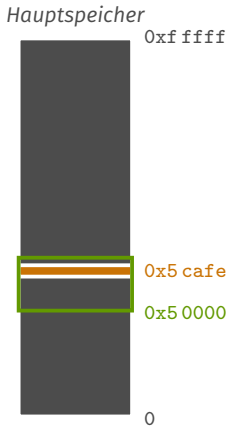
1. Setze `ds` auf `0x5000`



Der 8086 – Real Mode Segmentierung

Zugriff auf Zieldaten in `0x5cafe`

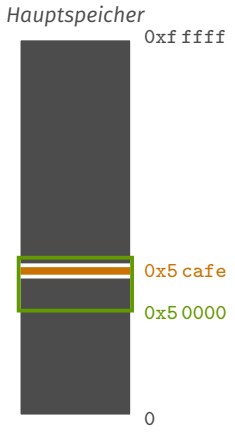
1. Setze `ds` auf `0x5000`
   (Bereich `0x50000` – `0x5ffff`)



Der 8086 – Real Mode Segmentierung

Zugriff auf Zieldaten in `0x5cafe`

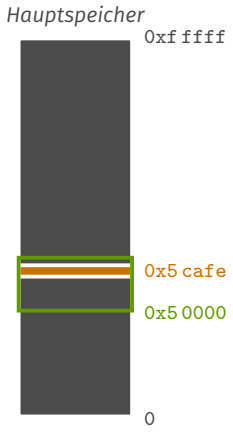
1. Setze `ds` auf `0x5000`
   (Bereich `0x50000` – `0x5ffff`)
2. Zugriff über `ds:0xcafe`



Der 8086 – Real Mode Segmentierung

Zugriff auf Zieldaten in `0x5cafe`

1. Setze `ds` auf `0x5000`
   (Bereich `0x50000` – `0x5ffff`)
2. Zugriff über `ds:0xcafe`
   (`ds` × `0x10` + `0xcafe` = `0x5cafe`)

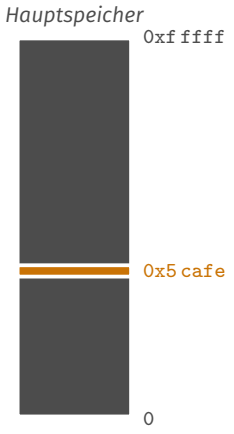


Der 8086 – Real Mode Segmentierung

Zugriff auf Zieldaten in `0x5cafe`

1. Setze `ds` auf `0x5000`
   (Bereich `0x50000` – `0x5ffff`)
2. Zugriff über `ds:0xcafe`
   (`ds` × `0x10` + `0xcafe` = `0x5cafe`)

Alternativ:



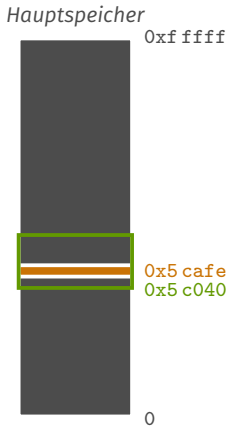
Der 8086 – Real Mode Segmentierung

Zugriff auf Zieldaten in `0x5cafe`

1. Setze `ds` auf `0x5000`
   (Bereich `0x50000` – `0x5ffff`)
2. Zugriff über `ds:0xcafe`
   (`ds` × `0x10` + `0xcafe` = `0x5cafe`)

Alternativ:

1. Setze `ds` auf `0x5c04`
   (Bereich `0x5c040` – `0x6c039`)



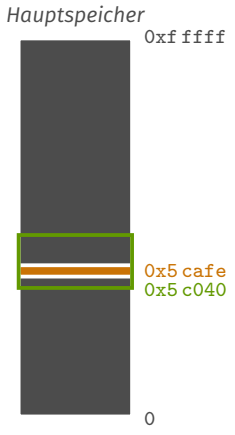
Der 8086 – Real Mode Segmentierung

Zugriff auf Zieldaten in `0x5cafe`

1. Setze `ds` auf `0x5000`
   (Bereich `0x50000` – `0x5ffff`)
2. Zugriff über `ds:0xcafe`
   (`ds` × `0x10` + `0xcafe` = `0x5cafe`)

Alternativ:

1. Setze `ds` auf `0x5c04`
   (Bereich `0x5c040` – `0x6c039`)
2. Zugriff über `ds:0xabe`



Besonderheit:

Der 8086 – Real Mode Segmentierung

Besonderheit:

Der 8086 – Real Mode Segmentierung

Besonderheit: Überlauf ist valid.

A20 Gate

80286 Adressbus

Problem: Zugriff auf `ds:0xcafe` mit `ds` = `0xf600` bei 24 bit Adressbus

A20 Gate

Problem: Zugriff auf `ds:0xcafe` mit `ds` = `0xf600` bei 24 bit Adressbus ergibt `0x1c0fe` (statt nur `0xc0fe` nach Überlauf bei 20 bit)

A20 Gate

Problem: Zugriff auf `ds:0xcafe` mit `ds` = `0xf600` bei 24 bit Adressbus ergibt `0x1c0fe` (statt nur `0xc0fe` nach Überlauf bei 20 bit)

- *MS-DOS 1.25* verlässt sich auf das Überlaufverhalten

A20 Gate

80286 Adressbus

Problem: Zugriff auf `ds:0xcafe` mit `ds` = `0xf600` bei 24 bit Adressbus ergibt `0x1c0fe` (statt nur `0xc0fe` nach Überlauf bei 20 bit)

- *MS-DOS 1.25* verlässt sich auf das Überlaufverhalten

  Abwärtskompatibilität des 80286 sollte jedoch unbedingt gewährleistet sein!

A20 Gate

80286 Adressbus

Problem: Zugriff auf `ds:0xcafe` mit `ds` = `0xf600` bei 24 bit Adressbus ergibt `0x1c0fe` (statt nur `0xc0fe` nach Überlauf bei 20 bit)

- *MS-DOS 1.25* verlässt sich auf das Überlaufverhalten

  Abwärtskompatibilität des 80286 sollte jedoch unbedingt gewährleistet sein!

→ Logikgatter zur Steuerung der A20 Leitung

A20 Gate

80286 Adressbus 

Problem: Zugriff auf `ds:0xcafe` mit `ds` = `0xf600` bei 24 bit Adressbus ergibt `0x1c0fe` (statt nur `0xc0fe` nach Überlauf bei 20 bit)

- *MS-DOS 1.25* verlässt sich auf das Überlaufverhalten

  Abwärtskompatibilität des 80286 sollte jedoch unbedingt gewährleistet sein!

→ Logikgatter zur Steuerung der A20 Leitung

  - Zieht die 20. Adressleitung auf 0

A20 Gate

80286 Adressbus 

Problem: Zugriff auf `ds:0xcafe` mit `ds` = `0xf600` bei 24 bit Adressbus ergibt `0x1c0fe` (statt nur `0xc0fe` nach Überlauf bei 20 bit)

- *MS-DOS 1.25* verlässt sich auf das Überlaufverhalten

  Abwärtskompatibilität des 80286 sollte jedoch unbedingt gewährleistet sein!

→ Logikgatter zur Steuerung der A20 Leitung
  - Zieht die 20. Adressleitung auf 0
  - A20 Gate bei Start geschlossen (Leitung deaktiviert)

A20 Gate

80286 Adressbus 

Problem: Zugriff auf `ds:0xcafe` mit `ds` = `0xf600` bei 24 bit Adressbus ergibt `0x1c0fe` (statt nur `0xc0fe` nach Überlauf bei 20 bit)

- *MS-DOS 1.25* verlässt sich auf das Überlaufverhalten

  Abwärtskompatibilität des 80286 sollte jedoch unbedingt gewährleistet sein!

→ Logikgatter zur Steuerung der A20 Leitung
  - Zieht die 20. Adressleitung auf 0
  - A20 Gate bei Start geschlossen (Leitung deaktiviert)
  - Steuerbar über 8042 Tastaturcontroller

A20 Gate

80286 Adressbus 

Problem: Zugriff auf `ds:0xcafe` mit `ds` = `0xf600` bei 24 bit Adressbus ergibt `0x1c0fe` (statt nur `0xc0fe` nach Überlauf bei 20 bit)

- *MS-DOS 1.25* verlässt sich auf das Überlaufverhalten

  Abwärtskompatibilität des 80286 sollte jedoch unbedingt gewährleistet sein!

→ Logikgatter zur Steuerung der A20 Leitung

- Zieht die 20. Adressleitung auf 0
- A20 Gate bei Start geschlossen (Leitung deaktiviert)
- Steuerbar über 8042 Tastaturcontroller
- Langsam, komplizierter Zugriff, (anfangs) Zustand nicht lesbar

A20 Gate

80286 Adressbus

Problem: Zugriff auf `ds:0xcafe` mit `ds` = `0xf600` bei 24 bit Adressbus ergibt `0x1c0fe` (statt nur `0xc0fe` nach Überlauf bei 20 bit)

- *MS-DOS 1.25* verlässt sich auf das Überlaufverhalten

  Abwärtskompatibilität des 80286 sollte jedoch unbedingt gewährleistet sein!

→ Logikgatter zur Steuerung der A20 Leitung

- Zieht die 20. Adressleitung auf 0
- A20 Gate bei Start geschlossen (Leitung deaktiviert)
- Steuerbar über 8042 Tastaturcontroller
- Langsam, komplizierter Zugriff, (anfangs) Zustand nicht lesbar

Siehe auch: Xbox Hack (17 Mistakes Microsoft Made in the Xbox Security System)

Wechsel in 32 bit Protected-Mode

Ausgehend vom *Real Mode*

- Laden von weiterem StartUp-Code
- Interrupts (inklusive NMI) ausschalten
- **A20 Gate aktivieren**
- *Global Descriptor Table* laden
- *Protection Enable*-bit (PE) im Kontrollregister 0 (`cr0`) setzen

Wechsel in 32 bit Protected-Mode

Ausgehend vom *Real Mode*

- Laden von weiterem StartUp-Code
- Interrupts (inklusive NMI) ausschalten
- A20 Gate aktivieren
- ***Global Descriptor Table* laden**
- *Protection Enable*-bit (PE) im Kontrollregister 0 (`cr0`) setzen

Segmentierung im Protected Mode

- Einführung von Deskriptortabellen

  GDT Global Descriptor Table

  LDT Local Descriptor Table

  IDT Interrupt Descriptor Table

- jeder der Deskriptoren (max. 8 192 pro Tabelle) besteht aus
  - Basisadresse (24 bit bzw. 32 bit)
  - Länge (16 bit bzw. 20 bit)
  - Parameter wie Typ, Berechtigung, Aktiv, ...

Segmentierung im Protected Mode

- Einführung von Deskriptortabellen

  GDT Global Descriptor Table

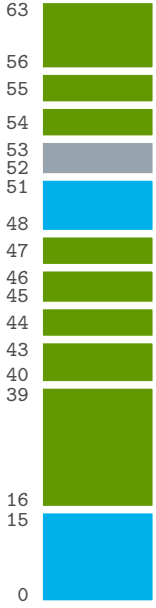
  LDT Local Descriptor Table

  IDT Interrupt Descriptor Table

- jeder der Deskriptoren (max. 8 192 pro Tabelle) besteht aus
  - Basisadresse (24 bit bzw. 32 bit)
  - Länge (16 bit bzw. 20 bit)
  - Parameter wie Typ, Berechtigung, Aktiv, ...

⚠ **Segmentselektoren zeigen nun auf Einträge in GDT/LDT**

Eintrag in 32 bit Global Deskriptor Table



Base Address (high): *Oberer* Teil der Startadresse

Granularity: Byte `(0)` oder 4 KB-Block `(1)`

Operation Size: 16 bit `(0)` oder 32 bit `(1)`

Reserviert

Segment Limit (high): *Oberer* Teil der Segmentgröße

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

Descriptor Privilege Level: Erlaubter Ring

Deskriptortyp: System `(0)` oder Code/Daten `(1)`

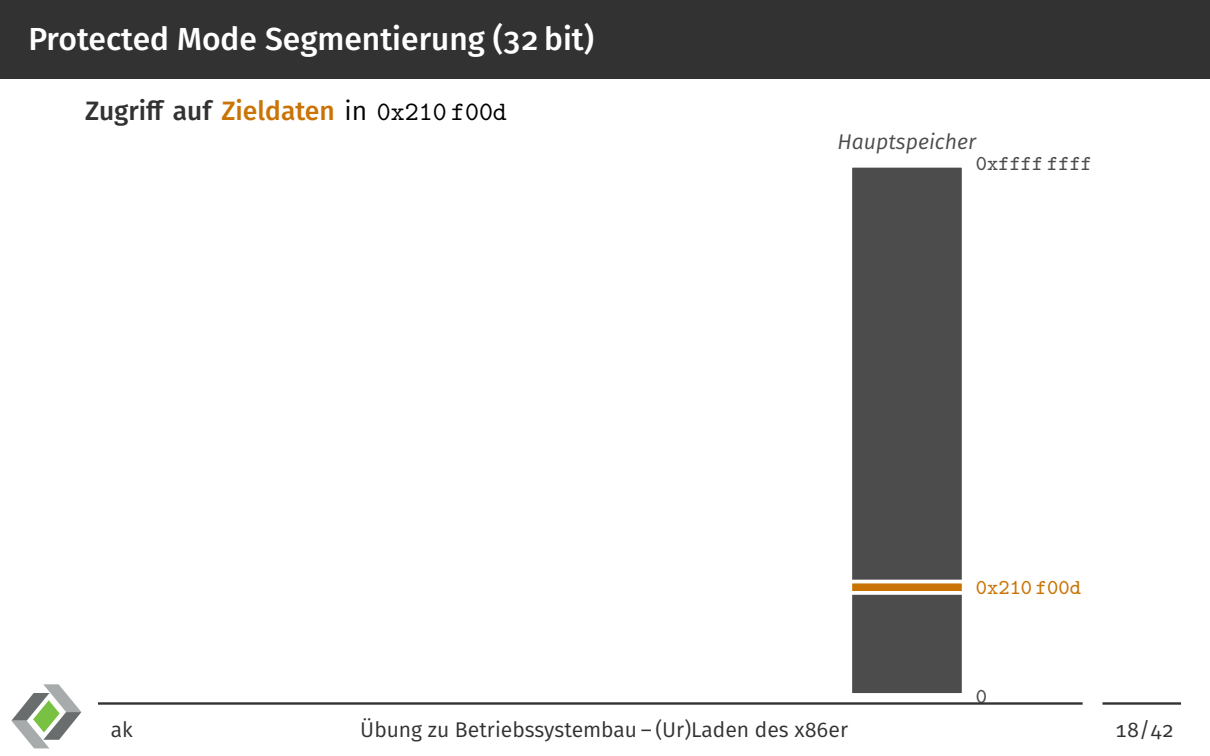
Type: Legt Berechtigungen fest (r/w/x)

Base Address (low): *Unterer* Teil der Startadresse

Segment Limit (low): *Unterer* Teil der Segmentgröße

Protected Mode Segmentierung (32 bit)

Zugriff auf Zieldaten in `0x210f00d`



Protected Mode Segmentierung (32 bit)

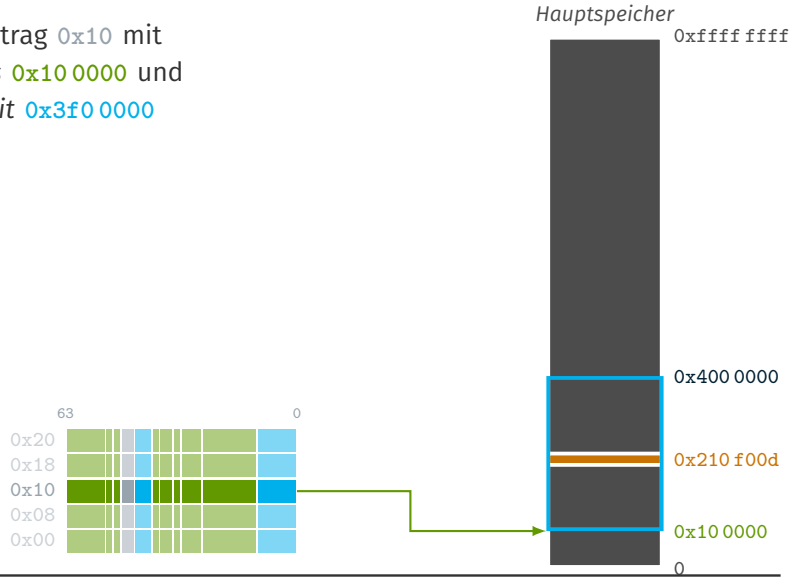
Zugriff auf Zieldaten in `0x210f00d`

- über **GDT**

Protected Mode Segmentierung (32 bit)

Zugriff auf Zieldaten in `0x210 f00d`

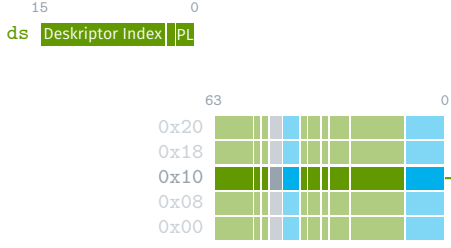
- über **GDT** Eintrag `0x10` mit
  Base Address `0x10 0000` und
  Segment Limit `0x3f0 0000`

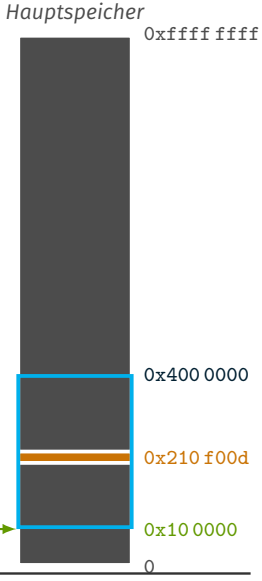


Protected Mode Segmentierung (32 bit)

Zugriff auf Zieldaten in `0x210f00d`

- über **GDT** Eintrag `0x10` mit
  Base Address `0x10 0000` und
  Segment Limit `0x3f0 0000`
- mit Segmentregister `ds`



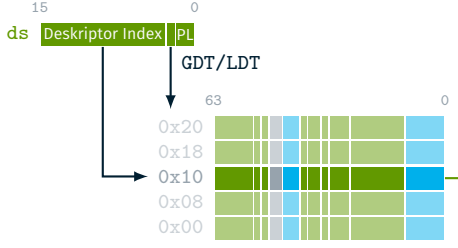


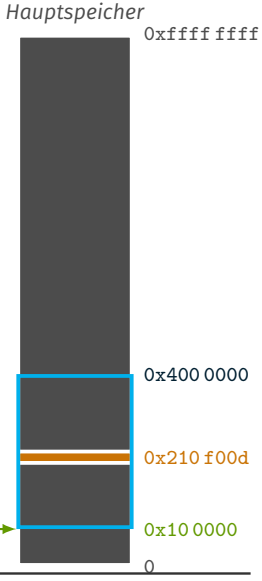
Protected Mode Segmentierung (32 bit)

Zugriff auf Zieldaten in `0x210f00d`

- über **GDT** Eintrag `0x10` mit
  Base Address `0x10 0000` und
  Segment Limit `0x3f0 0000`
- mit Segmentregister `ds` mit

```
mov ax, 0x10
mov ds, ax
```



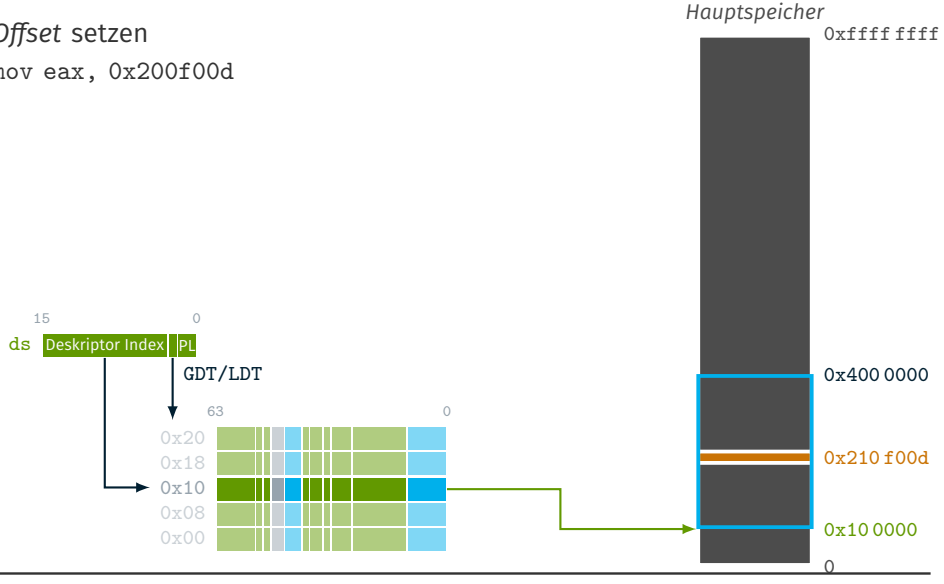


Protected Mode Segmentierung (32 bit)

Zugriff auf Zieldaten in `0x210f00d`

- *Offset* setzen

  `mov eax, 0x200f00d`

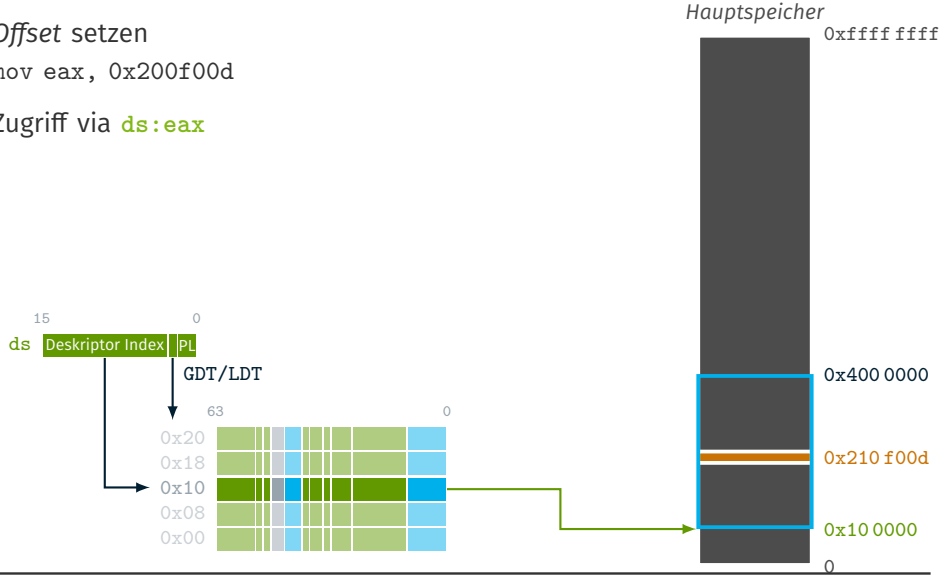


Zugriff auf Zieldaten in `0x210f00d`

- *Offset* setzen

  `mov eax, 0x200f00d`

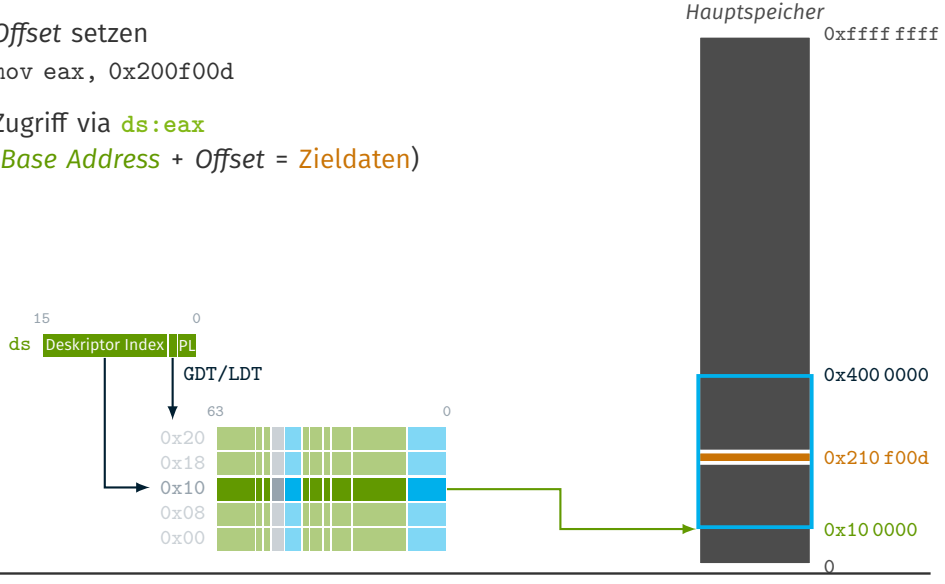
- Zugriff via `ds:eax`



Protected Mode Segmentierung (32 bit)

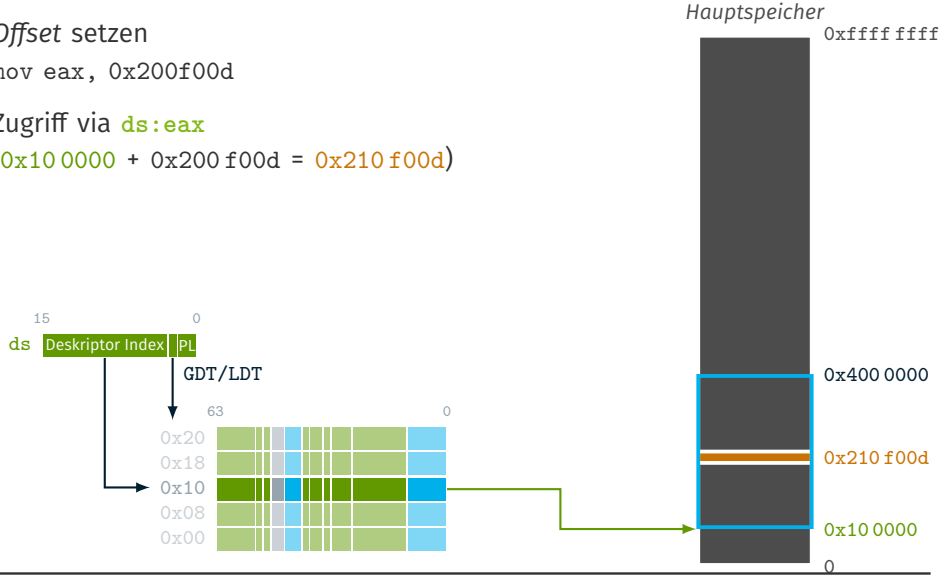
Zugriff auf Zieldaten in `0x210f00d`

- *Offset* setzen
  `mov eax, 0x200f00d`
- Zugriff via `ds:eax`
  (*Base Address* + *Offset* = Zieldaten)



Zugriff auf Zieldaten in `0x210f00d`

- *Offset* setzen
  `mov eax, 0x200f00d`
- Zugriff via `ds:eax`
  (`0x10 0000` + `0x200f00d` = `0x210f00d`)



Wird Segmentierung heutzutage noch genutzt?

Ausgangslage: Paging allein ist eigentlich ausreichend.

Wird Segmentierung heutzutage noch genutzt?

Ausgangslage: Paging allein ist eigentlich ausreichend.

Problem: Deaktivieren der Segmentierung nicht möglich!

Wird Segmentierung heutzutage noch genutzt?

Ausgangslage: Paging allein ist eigentlich ausreichend.

Problem: Deaktivieren der Segmentierung nicht möglich!

Lösung: Neutralisieren, indem in der *GDT* alles eingeblendet wird

1. Nulleintrag (*vorgeschrieben*)

Wird Segmentierung heutzutage noch genutzt?

Ausgangslage: Paging allein ist eigentlich ausreichend.

Problem: Deaktivieren der Segmentierung nicht möglich!

Lösung: Neutralisieren, indem in der *GDT* alles eingeblendet wird

1. Nulleintrag (*vorgeschrieben*)
2. Kernel Code Segment (von 0 – 4 GB, Ring 0)
3. Kernel Data Segment (von 0 – 4 GB, Ring 0)

Wird Segmentierung heutzutage noch genutzt?

Ausgangslage: Paging allein ist eigentlich ausreichend.

Problem: Deaktivieren der Segmentierung nicht möglich!

Lösung: Neutralisieren, indem in der *GDT* alles eingeblendet wird

1. Nulleintrag (*vorgeschrieben*)
2. Kernel Code Segment (von 0 – 4 GB, Ring 0)
3. Kernel Data Segment (von 0 – 4 GB, Ring 0)
4. User Code Segment (von 0 – 4 GB, Ring 3)
5. User Data Segment (von 0 – 4 GB, Ring 3)

Wechsel in 32 bit Protected-Mode

Ausgehend vom *Real Mode*

- Laden von weiterem StartUp-Code
- Interrupts (inklusive NMI) ausschalten
- A20 Gate aktivieren
- ***Global Descriptor Table* laden**
- *Protection Enable*-bit (PE) im Kontrollregister 0 (`cr0`) setzen

Wechsel in 32 bit Protected-Mode

Ausgehend vom *Real Mode*

- Laden von weiterem StartUp-Code
- Interrupts (inklusive NMI) ausschalten
- A20 Gate aktivieren
- *Global Descriptor Table* laden
- ***Protection Enable*-bit (PE) im Kontrollregister 0 (`cr0`) setzen**

Quelle: KnowYourMeme

Zwischenstand

Resultat: Wir wollen keinen eigenen Bootloader schreiben!

Zwischenstand

Resultat: Wir wollen keinen eigenen Bootloader schreiben!

Kein Problem, gibt genug:

- GNU **GRUB** (Grand Unified Bootloader)
  - GRUB Legacy
  - GRUB 2

Zwischenstand

Resultat: Wir wollen keinen eigenen Bootloader schreiben!

Kein Problem, gibt genug:

- GNU **GRUB** (Grand Unified Bootloader)
  - GRUB Legacy
  - GRUB 2
- **SYSLINUX** (PXELINUX)
- **QEMU**-intern (via `-kernel` Parameter)

Zwischenstand

Resultat: Wir wollen keinen eigenen Bootloader schreiben!

Kein Problem, gibt genug:

- GNU **GRUB** (Grand Unified Bootloader)
  - GRUB Legacy
  - GRUB 2
- **SYSLINUX** (PXELINUX)
- **QEMU**-intern (via `-kernel` Parameter)

Aber wie verwenden?

Zwischenstand

Resultat: Wir wollen keinen eigenen Bootloader schreiben!

Kein Problem, gibt genug:

- GNU **GRUB** (Grand Unified Bootloader)
  - GRUB Legacy
  - GRUB 2
- **SYSLINUX** (PXELINUX)
- **QEMU**-intern (via `-kernel` Parameter)

Aber wie verwenden?

Alle unterstützen (mehr oder weniger) den **Multiboot Standard**

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 **32 bit ELF** oder **a.out** vorliegen

Überblick zur Multiboot Specification

- offener PC **Bootloader Standard**, ab 1995 entwickelt
- Betriebssystem muss als **32 bit 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
  - setzt optional auch Grafikmodus

Überblick zur Multiboot Specification

- offener PC **Bootloader Standard**, ab 1995 entwickelt
- Betriebssystem muss als **32 bit 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
  - 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 **32 bit 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
  - setzt optional auch Grafikmodus
- übergibt dem BS **„vitale“ Informationen** über das System
- lädt ggf. auch die **initiale Ramdisk** in den Speicher

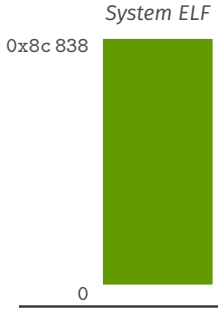
Überblick zur Multiboot Specification

- offener PC **Bootloader Standard**, ab 1995 entwickelt
- Betriebssystem muss als **32 bit 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
  - setzt optional auch Grafikmodus
- übergibt dem BS **„vitale“ Informationen** über das System
- lädt ggf. auch die **initiale Ramdisk** in den Speicher
- Referenzimplementierung ist **GRUB**

Aufbau einer MULTIBOOT-kompatiblen Binärdatei

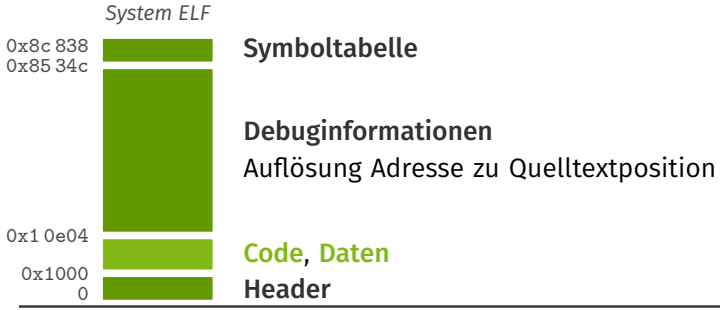
Beispiel:

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



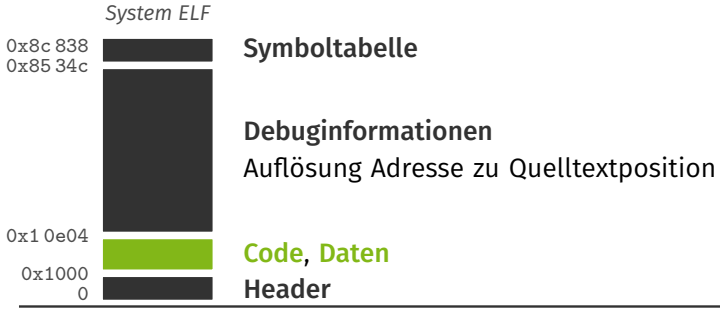
Beispiel:

`make` generiert eine 563K große `.build/system` ELF.
Analyse mittels `readelf` offenbart folgende Struktur



Beispiel:

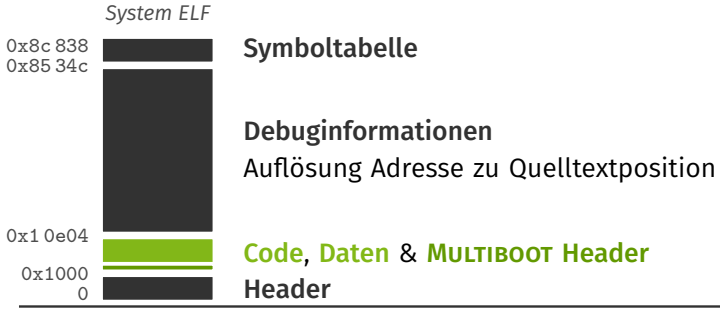
`make` generiert eine 563K große `.build/system` ELF.
Analyse mittels `readelf` offenbart folgende Struktur, für uns ist jedoch nur die (durch das Linkerskript) zusammengefasste Code- (`.text`) und Datensektion (`.[ro]data`) interessant



Aufbau einer MULTIBOOT-kompatiblen Binärdatei

Beispiel:

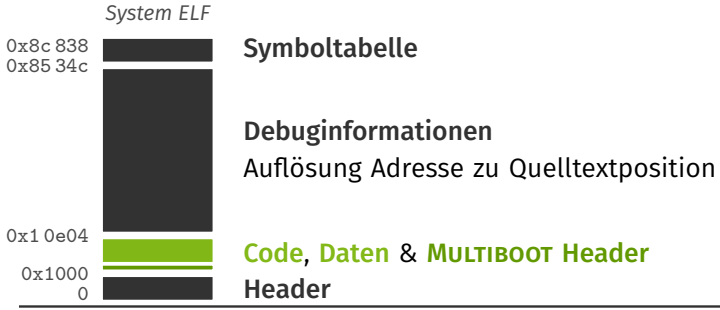
`make` generiert eine 563K große `.build/system` ELF.
Analyse mittels `readelf` offenbart folgende Struktur, für uns ist jedoch nur die (durch das Linkerskript) zusammengefasste Code- (`.text`) und Datensektion (`.[ro]data`) interessant, in welcher auch der **MULTIBOOT Header** liegt



Aufbau einer MULTIBOOT-kompatiblen Binärdatei

MULTIBOOT Header

- Erkennung durch Wert `0x1bad b002` (und Prüfsumme)
- muss in den ersten 8 192 Bytes (der ELF) liegen
- bei uns in `boot/multiboot/header.asm` definiert
- beinhaltet Konfiguration (via Flags)



Multiboot Header in StuBS

☞ boot/multiboot/header.asm

```
1  [SECTION .multiboot_header]
2  ; Konstanten definiert in boot/multiboot/config.inc
3  MULTIBOOT_MAGIC      equ 0x1badb002  ; Magischer Header
4
5  MULTIBOOT_PAGE_ALIGN equ 1<<0        ; initrd an 4K ausrichten
6  MULTIBOOT_MEM_INFO   equ 1<<1        ; Speicher Informationen
7  MULTIBOOT_FLAGS equ MULTIBOOT_PAGE_ALIGN | MULTIBOOT_MEM_INFO
8
9  MULTIBOOT_CHKSUM equ -(MULTIBOOT_MAGIC + MULTIBOOT_FLAGS)
10
11 align 4                              ; Ausrichten an 4K
12 multiboot_header:
13     dd MULTIBOOT_MAGIC
14     dd MULTIBOOT_FLAGS
15     dd MULTIBOOT_CHKSUM
```

Linkerskript in StuBS

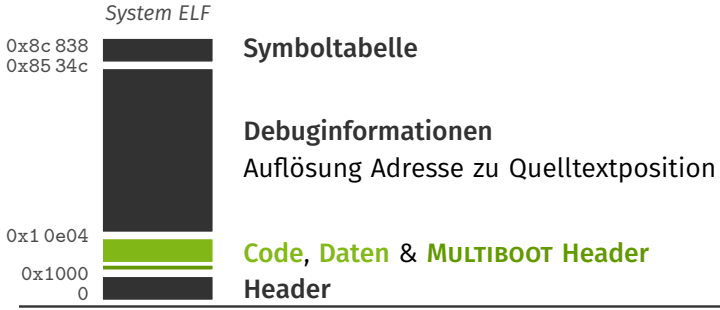
☞ compiler/sections.ld

```
1  ENTRY(startup_bsp)          /* Einsprungsfunktion */
2
3  SECTIONS {
4    . = 16M;                  /* Startadresse des Systems */
5
6    .boot : {
7      *(.multiboot_header)    /* Multiboot am Anfang */
8    }
9
10   .text : {
11     *(".text")              /* Programmcode */
12     *(".text$")
13     *(".init")
14     *(".fini")
15     *(".gnu.linkonce.*")
16   }
17 }
```

Aufbau einer MULTIBOOT-kompatiblen Binärdatei

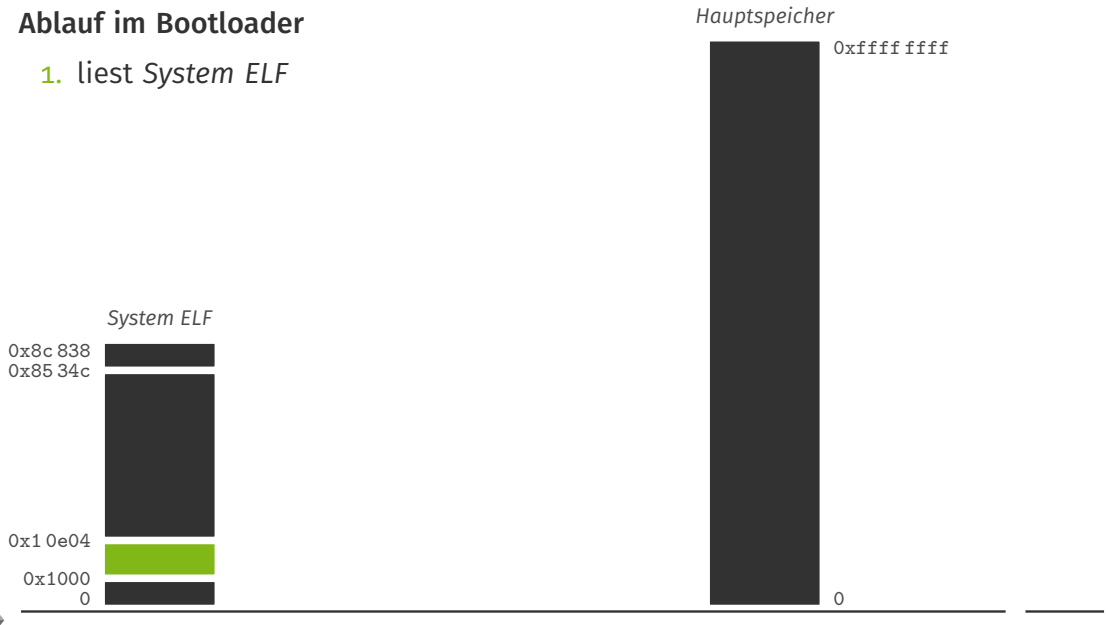
MULTIBOOT Header

- Erkennung durch Wert `0x1bad b002` (und Prüfsumme)
- muss in den ersten 8 192 Bytes (der ELF) liegen
- bei uns in `boot/multiboot/header.asm` definiert
- beinhaltet Konfiguration (via Flags)



Ablauf im Bootloader

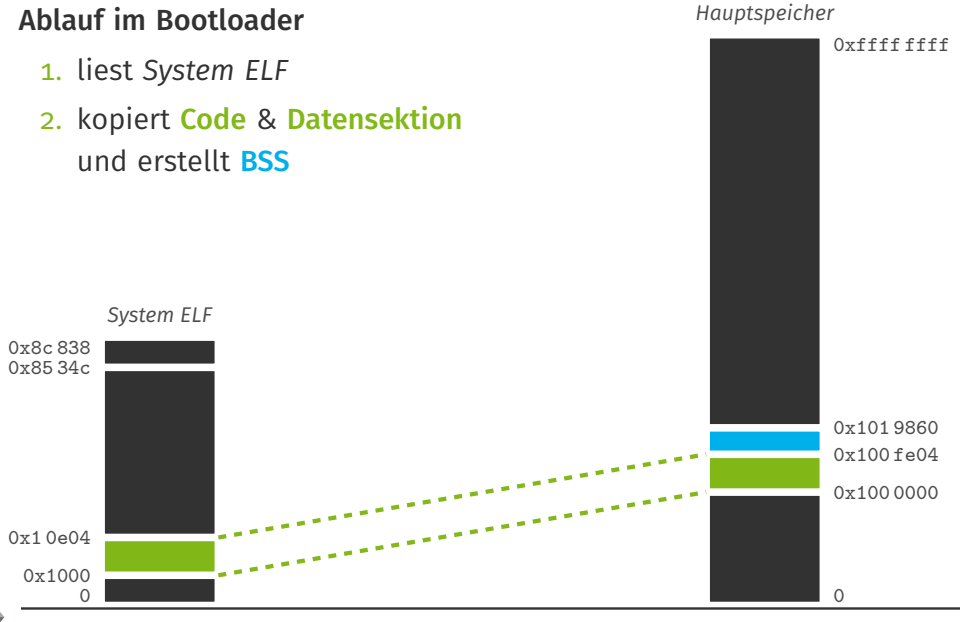
1. liest *System ELF*



Laden einer MULTIBOOT-kompatiblen Binärdatei

Ablauf im Bootloader

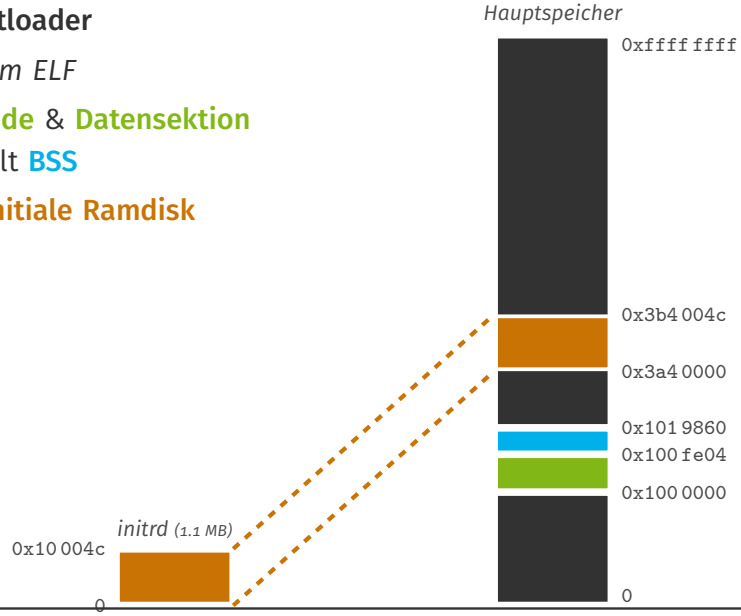
1. liest *System ELF*
2. kopiert **Code** & **Datensektion** und erstellt **BSS**



Laden einer MULTIBOOT-kompatiblen Binärdatei

Ablauf im Bootloader

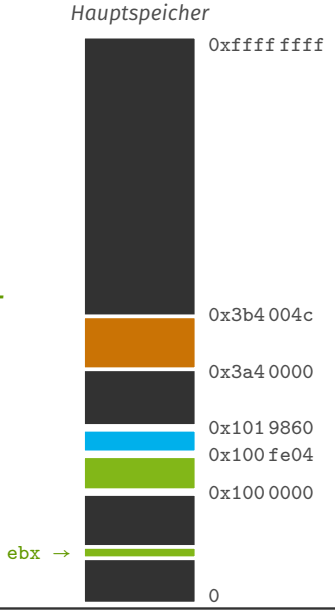
1. liest *System ELF*
2. kopiert **Code** & **Datensektion** und erstellt **BSS**
3. lädt ggf. **initiale Ramdisk**



Laden einer MULTIBOOT-kompatiblen Binärdatei

Ablauf im Bootloader

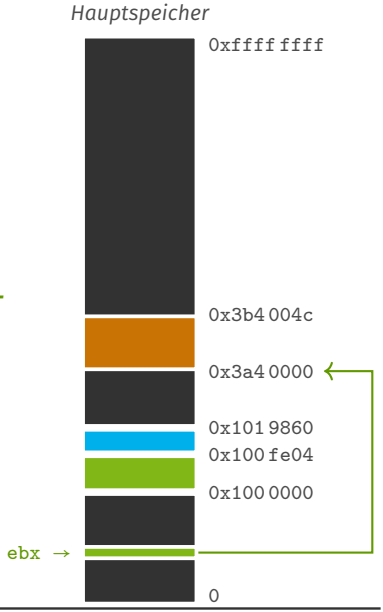
1. liest *System ELF*
2. kopiert **Code** & **Datensektion** und erstellt **BSS**
3. lädt ggf. **initiale Ramdisk**
4. setzt `eax` auf `0x2badb002` sowie `ebx` als Zeiger auf Struktur mit **MULTIBOOT Information**



Laden einer MULTIBOOT-kompatiblen Binärdatei

Ablauf im Bootloader

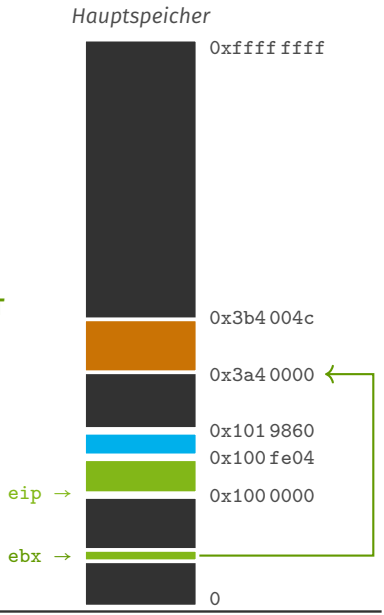
1. liest *System ELF*
2. kopiert **Code** & **Datensektion** und erstellt **BSS**
3. lädt ggf. **initiale Ramdisk**
4. setzt `eax` auf `0x2badb002` sowie `ebx` als Zeiger auf Struktur mit **MULTIBOOT Information**



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)



Initialisierung

Was fehlt?

Initialisierung

Was fehlt?

Initialisierung

Was fehlt?

- Wechsel in 64 bit-Modus
- Initialisierung der Umgebung
- *optional* Mehrkernunterstützung aktivieren

Initialisierung

Was fehlt?

- **Wechsel in 64 bit-Modus**
- Initialisierung der Umgebung
- *optional* Mehrkernunterstützung aktivieren

Long Mode

Quelle: Wikipedia

Long Mode

Namen: x64, x86_64, AMD64, Intel 64, IA-32e, EM64T

Long Mode

Namen: x64, x86_64, AMD64, Intel 64, IA-32e, EM64T *(aber nicht IA-64!)*

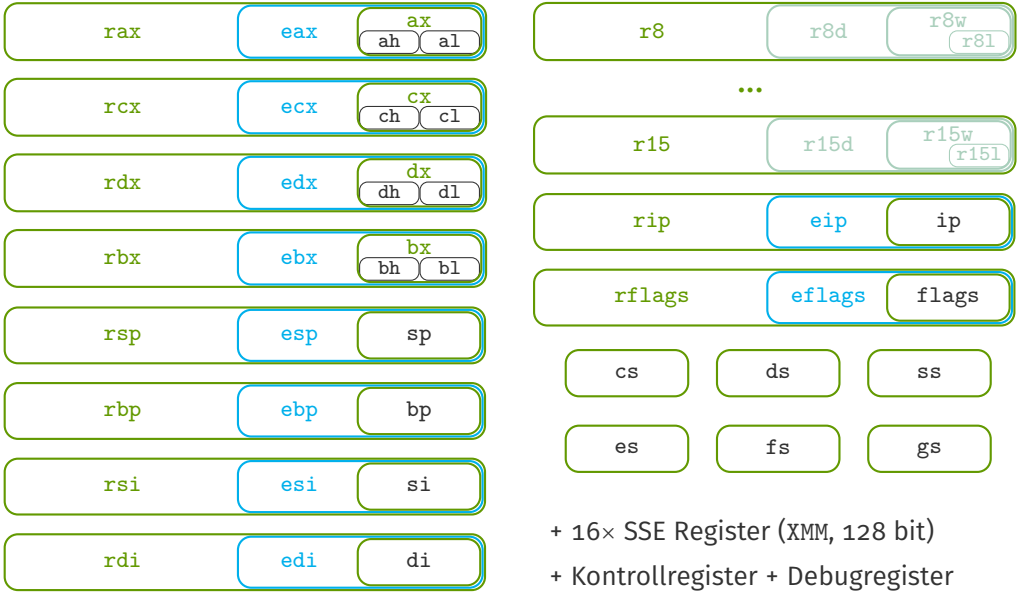
- 64 bit
- **abwärtskompatibel**
- Speicher: Adressbus 48-bit (256 TB)
- Verdopplung der Registeranzahl
- *Streaming SIMD Extensions* (SSE)

Long Mode

Namen: x64, x86_64, AMD64, Intel 64, IA-32e, EM64T *(aber nicht IA-64!)*

- 64 bit
- **abwärtskompatibel**
- Speicher: Adressbus 48-bit (256 TB)
- Verdopplung der Registeranzahl
- *Streaming SIMD Extensions* (SSE)
- **keine Speichersegmentierung** im Long Mode
- vierstufiges Paging

Long Mode Register



+ 16× SSE Register (`XMM`, 128 bit)
+ Kontrollregister + Debugregister

Wechsel in Long Mode

☞ boot/longmode.asm

(Kann doch nicht so kompliziert sein?)

Wechsel in Long Mode *(Eiertanz, Teil 2)*

☞ `boot/longmode.asm`

1. Ist `cpuid` vorhanden?
   (21. Bit im `flags` Register versuchen umzudrehen – über Stack.
   Falls das geht gibt es den `cpuid`-Befehl)

Wechsel in Long Mode *(Eiertanz, Teil 2)*

☞ `boot/longmode.asm`

1. Ist `cpuid` vorhanden?
2. Kann `cpuid` erweiterte Funktionen?
   (`cpuid` mit Parameter `0x8000 0000` gibt die Nummer der höchsten unterstützten Funktion zurück)

Wechsel in Long Mode *(Eiertanz, Teil 2)*

☞ `boot/longmode.asm`

1. Ist `cpuid` vorhanden?
2. Kann `cpuid` erweiterte Funktionen?
3. Unterstützt die CPU den *Long Mode*?
   (`cpuid` mit Parameter `0x8000 0001` – falls vorhanden – gibt die erweiterten Prozessor-Informationen zurück. Falls dann in `edx` das 29. Bit gesetzt ist, ist *Long Mode* vorhanden)

Wechsel in Long Mode *(Eiertanz, Teil 2)*

☞ `boot/longmode.asm`

1. Ist `cpuid` vorhanden?
2. Kann `cpuid` erweiterte Funktionen?
3. Unterstützt die CPU den *Long Mode*?
4. Tabellen für Paging aufsetzen
   (bei uns Identitätsabbildung der ersten 4 GB unter Verwendung von 2 MB *Huge Pages*, damit brauchen wir nur 6 Tabellen)

Wechsel in Long Mode *(Eiertanz, Teil 2)*

☞ `boot/longmode.asm`

1. Ist `cpuid` vorhanden?
2. Kann `cpuid` erweiterte Funktionen?
3. Unterstützt die CPU den *Long Mode*?
4. Tabellen für Paging aufsetzen
5. Paging vorbereiten
   (Adresse der obersten Tabelle (Level 4) in Kontrollregister 3 (`cr3`) schreiben und Physikalische Adresserweiterung (PAE) durch das Setzen des 5. Bit in Kontrollregister 4 (`cr4`) aktivieren)

Wechsel in Long Mode *(Eiertanz, Teil 2)*

☞ `boot/longmode.asm`

1. Ist `cpuid` vorhanden?
2. Kann `cpuid` erweiterte Funktionen?
3. Unterstützt die CPU den *Long Mode*?
4. Tabellen für Paging aufsetzen
5. Paging vorbereiten
6. *Long Mode* aktivieren
   (über das *Model-Specific Register* (MSR) Nummer `0xC000 0080` – *Extended Feature Enable Register* (EFER) – das 8. Bit setzen, via `wrmsr`)

Wechsel in Long Mode *(Eiertanz, Teil 2)*

☞ `boot/longmode.asm`

1. Ist `cpuid` vorhanden?
2. Kann `cpuid` erweiterte Funktionen?
3. Unterstützt die CPU den *Long Mode*?
4. Tabellen für Paging aufsetzen
5. Paging vorbereiten
6. *Long Mode* aktivieren
7. Paging aktivieren
   (das 31. Bit in Kontrollregister 0 (`cr0`) setzen)

Wechsel in Long Mode *(Eiertanz, Teil 2)*

☞ `boot/longmode.asm`

1. Ist `cpuid` vorhanden?
2. Kann `cpuid` erweiterte Funktionen?
3. Unterstützt die CPU den *Long Mode*?
4. Tabellen für Paging aufsetzen
5. Paging vorbereiten
6. *Long Mode* aktivieren
7. Paging aktivieren
8. vorübergehende GDT laden
   (Neben dem 0 Eintrag gibt es nur noch einen weiteren Eintrag für das globale Codesegment)

Wechsel in Long Mode *(Eiertanz, Teil 2)*

☞ `boot/longmode.asm`

1. Ist `cpuid` vorhanden?
2. Kann `cpuid` erweiterte Funktionen?
3. Unterstützt die CPU den *Long Mode*?
4. Tabellen für Paging aufsetzen
5. Paging vorbereiten
6. *Long Mode* aktivieren
7. Paging aktivieren
8. vorübergehende GDT laden
9. *Weiter* Sprung in den 64 bit Code

   (via `jmp 0x8:long_mode_start`, dort Segmentregister auf `0` setzen und Hochsprachenfunktion `kernel_init()` aufrufen)

Long Mode mit Multiboot

Ausgangslage: Wir haben nun unseren Kernel als 64 bit ELF (`.build/system64`)

Long Mode mit Multiboot

Ausgangslage: Wir haben nun unseren Kernel als 64 bit ELF (`.build/system64`)

Problem: MULTIBOOT akzeptiert nur 32 bit ELF

Long Mode mit Multiboot

Ausgangslage: Wir haben nun unseren Kernel als 64 bit ELF (`.build/system64`)

Problem: MULTIBOOT akzeptiert nur 32 bit ELF

Lösung: ELF mittels `objcopy` umschreiben (nur Metadaten, Programmcode bleibt gleich)

Ausgangslage: Wir haben nun unseren Kernel als 64 bit ELF (`.build/system64`)

Problem: MULTIBOOT akzeptiert nur 32 bit ELF

Lösung: ELF mittels `objcopy` umschreiben (nur Metadaten, Programmcode bleibt gleich)

via Makefile Target:

```
1 .build/system: .build/system64
2     objcopy -I elf64-x86-64 -O elf32-i386 $< $
```

Initialisierung

Was fehlt?

- **Wechsel in 64 bit-Modus**
- Initialisierung der Umgebung
- *optional* Mehrkernunterstützung aktivieren

Initialisierung

Was fehlt?

- Wechsel in 64 bit-Modus
- **Initialisierung der Umgebung**
- *optional* Mehrkernunterstützung aktivieren

Initialisierung der [Hochsprachen-]Umgebung

☞ boot/startup.cc


```
1 extern "C" [[noreturn]] void kernel_init()
```

Initialisierung der [Hochsprachen-]Umgebung

☞ boot/startup.cc


```
1 extern "C" [[noreturn]] void kernel_init()
```

1. Interruptdeskriptortabelle installieren
   ➔ Übung zu Aufgabe 2

Initialisierung der [Hochsprachen-]Umgebung

☞ boot/startup.cc


```
1 extern "C" [[noreturn]] void kernel_init()
```

1. Interruptdeskriptortabelle installieren
2. Initialisierungsroutinen der C Laufzeitumgebung ausführen
   (Aufruf der durch den Übersetzer erstellten Funktion `_init()` via `CSU::initializer()` in ☞ compiler/libc.cc.
   Prologe der *C Runtime Objects* liegen in ☞ compiler/crti.asm)

Initialisierung der [Hochsprachen-]Umgebung

☞ boot/startup.cc


```
1 extern "C" [[noreturn]] void kernel_init()
```

1. Interruptdeskriptortabelle installieren
2. Initialisierungsroutinen der C Laufzeitumgebung ausführen

   Initialisierung globaler Objekte
   (Übersetzer legt Array `__init_array_start[]` mit Funktionspointer an)

Initialisierung der [Hochsprachen-]Umgebung

☞ boot/startup.cc


```
1 extern "C" [[noreturn]] void kernel_init()
```

1. Interruptdeskriptortabelle installieren
2. Initialisierungsroutinen der C Laufzeitumgebung ausführen

   Initialisierung globaler Objekte
3. *Advanced Configuration and Power Interface* (ACPI) abfragen

   (Informationen zu APIC und den vorhandenen Kernen bekommen ☞ machine/apic.cc)

Initialisierung der [Hochsprachen-]Umgebung

☞ boot/startup.cc


```
1 extern "C" [[noreturn]] void kernel_init()
```

1. Interruptdeskriptortabelle installieren
2. Initialisierungsroutinen der C Laufzeitumgebung ausführen
   Initialisierung globaler Objekte
3. *Advanced Configuration and Power Interface* (ACPI) abfragen
4. *Local APIC* konfigurieren
   (u.a. Logische ID setzen, alle Interrupts zulassen)

Initialisierung der [Hochsprachen-]Umgebung

☞ boot/startup.cc


```
1 extern "C" [[noreturn]] void kernel_init()
```

1. Interruptdeskriptortabelle installieren
2. Initialisierungsroutinen der C Laufzeitumgebung ausführen
   Initialisierung globaler Objekte
3. *Advanced Configuration and Power Interface* (ACPI) abfragen
4. *Local APIC* konfigurieren
5. `main()` aufrufen
   (unser Betriebssystem ist in der `main()` implementiert und sollte daher nicht zurückkehren, deswegen brauchen die Destruktoren eigentlich nicht mehr aufgerufen werden)

Initialisierung

Was fehlt?

- Wechsel in 64 bit-Modus
- **Initialisierung der Umgebung**
- *optional* Mehrkernunterstützung aktivieren

Initialisierung

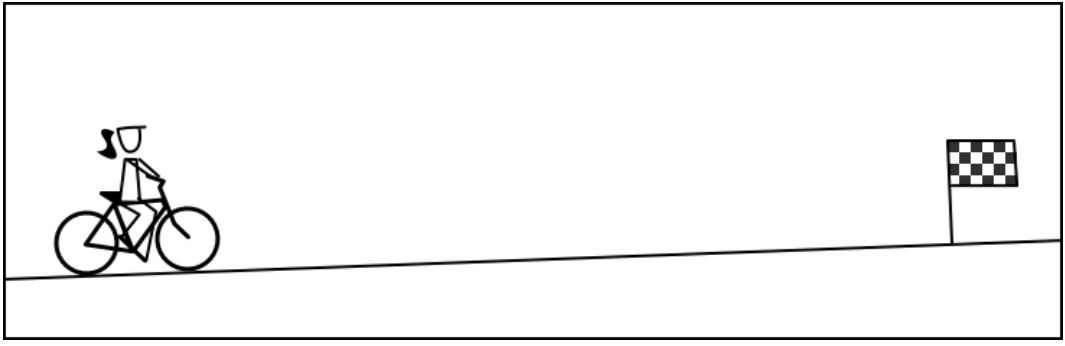
Was fehlt?

- Wechsel in 64 bit-Modus
- Initialisierung der Umgebung
- ***optional* Mehrkernunterstützung aktivieren**

Mehrkernunterstützung: Start der CPUs

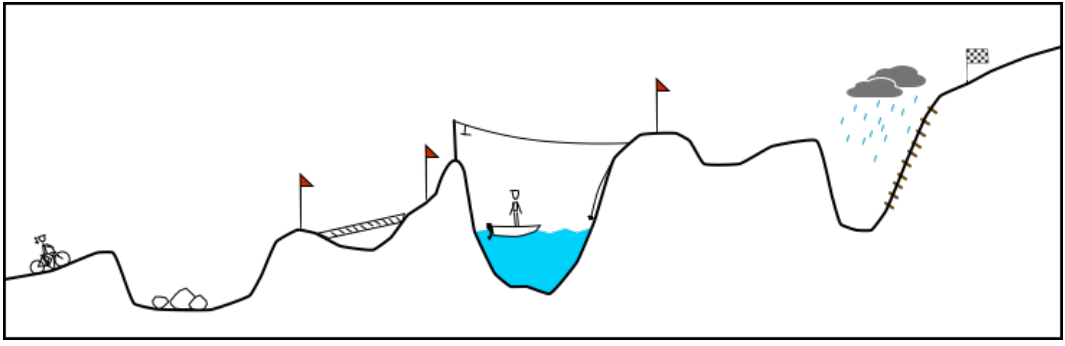
Wie kompliziert kann das schon sein?

Erwartung



Quelle: DogHouseDiaries

Realität



Quelle: DogHouseDiaries

Vorbereitung auf Bootstrap-Prozessor

☞ boot/startup_ap.cc

Bootstrap-Prozessor (BSP, CPU 0)

1. Aufruf von `ApplicationProcessor::boot()`

Bootstrap-Prozessor (BSP, CPU 0)

1. Aufruf von `ApplicationProcessor::boot()`
2. Startroutine und GDT an eine Adresse unterhalb 1 MB relozieren
   (*Application Processors* (AP) starten im *Real Mode*, deshalb platzieren wir die Routine an 0x4 0000. Theoretisch kann dies auch per Linkerskript im ELF geschehen, aber die Multibootimplementierung von Qemu ignoriert dies)

Bootstrap-Prozessor (BSP, CPU 0)

1. Aufruf von `ApplicationProcessor::boot()`
2. Startroutine und GDT an eine Adresse unterhalb 1 MB relozieren
3. Spezielle Interprozessorunterbrechung (*Startup IPI*) an alle Prozessorkerne schicken

   (Der Interruptvektor verweist auf die Einsprungsroutine: `0x40` – Adresse um 12 Bits nach rechts geschoben ergibt Vektor)

Applikationsprozessor (AP)

1. Start im *Real Mode*
   (in der Einsprungsroutine an `0x4 0000`)

Applikationsprozessor (AP)

1. Start im *Real Mode*
2. Wechsel in den *Protected Mode*
   (Interrupts deaktivieren, vorübergehende 16 bit GDT laden, 1. Bit in `cr0` setzen – A20 Gate bereits durch Bootstrapprozessor aktiviert/geöffnet)

Start der Applikationsprozessoren

☞ `boot/startup_ap.asm`

Applikationsprozessor (AP)

1. Start im *Real Mode*
2. Wechsel in den *Protected Mode*
3. Segmente initialisieren
   (vorübergehende 32 bit GDT laden, Segmentregister initialisieren)

Start der Applikationsprozessoren

☞ `boot/startup_ap.asm`

Applikationsprozessor (AP)

1. Start im *Real Mode*
2. Wechsel in den *Protected Mode*
3. Segmente initialisieren
4. Eigenen Stapel suchen
   (damit keine zwei gleichzeitig startenden APs auf den gleichen Stack zugreifen, wird atomar Speicher aus einem vorhandenen Stackarray reserviert)

Start der Applikationsprozessoren

☞ `boot/startup_ap.asm`

Applikationsprozessor (AP)

1. Start im *Real Mode*
2. Wechsel in den *Protected Mode*
3. Segmente initialisieren
4. Eigenen Stapel suchen
5. Wechsel in den *Long Mode*
   (`cpuid`, Paging, GDT, ..., Hochsprachenfunktion)

Start der Applikationsprozessoren

☞ `boot/startup_ap.asm`

Applikationsprozessor (AP)

1. Start im *Real Mode*
2. Wechsel in den *Protected Mode*
3. Segmente initialisieren
4. Eigenen Stapel suchen
5. Wechsel in den *Long Mode*
6. *Local APIC* konfigurieren
   (der LAPIC muss für jede CPU konfiguriert werden!)

Start der Applikationsprozessoren

☞ `boot/startup_ap.asm`

Applikationsprozessor (AP)

1. Start im *Real Mode*
2. Wechsel in den *Protected Mode*
3. Segmente initialisieren
4. Eigenen Stapel suchen
5. Wechsel in den *Long Mode*
6. *Local APIC* konfigurieren
7. `main_ap()` aufrufen
   (sollte auch nicht zurückkehren)

The End

ROM → Medium → Urlader → Initialisierung → `main[_ap]()`

The End

(Beginn der eigentlichen Betriebssystementwicklung)

Na, immer noch Lust auf Betriebssystementwicklung? ☺