Malware-Analyse und Reverse Engineering

3: Laufzeit!

30.3.2017

Prof. Dr. Michael Engel

Überblick (heute)

Themen:

- Laden und Ausführen von Binärprogrammen
  - Linkerskripte
  - Dynamischer Loader ld.so
  - Statische und Dynamische Libraries
  - Speicherallokation

Übersetzung eines (C-)Programms

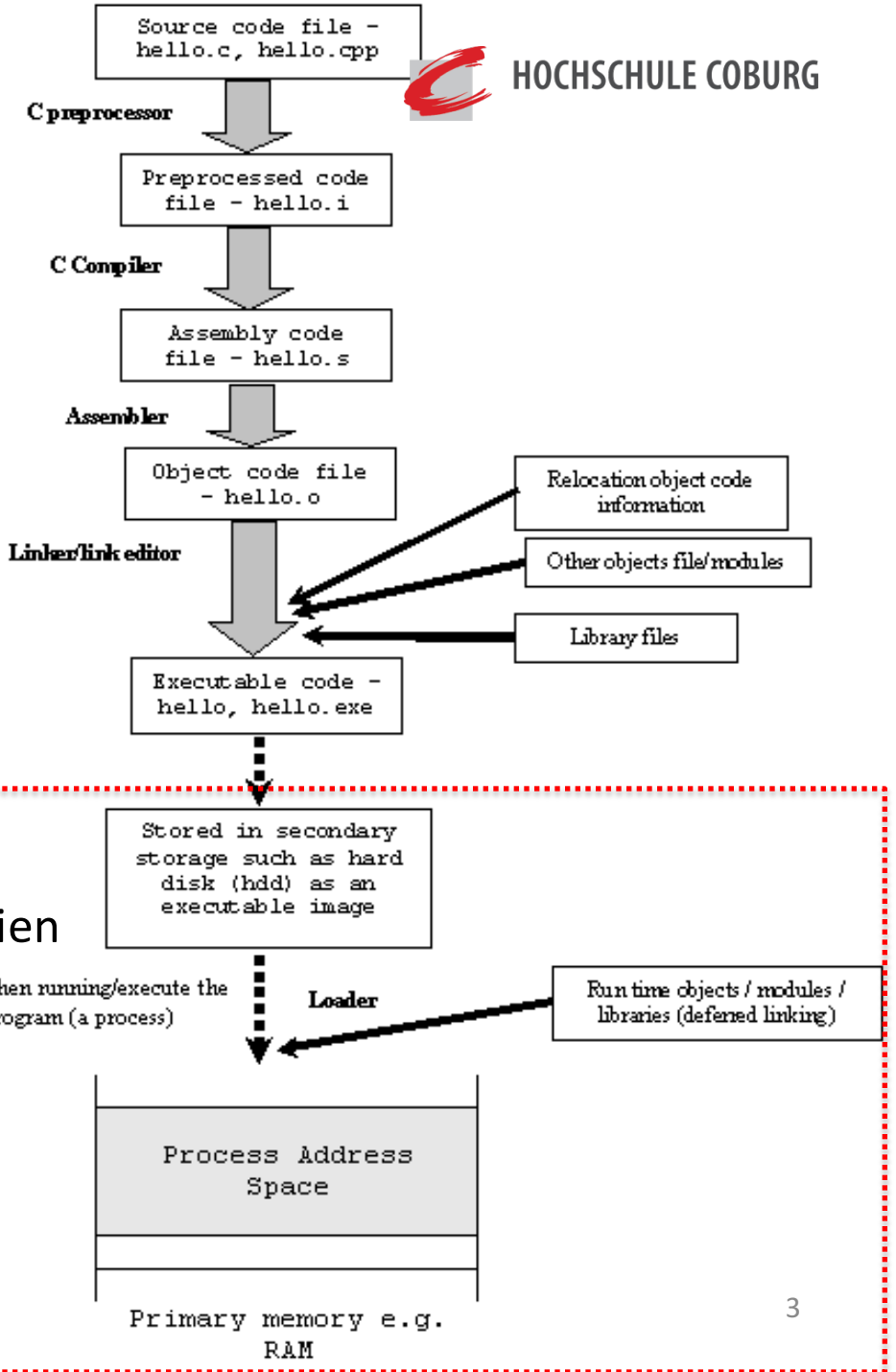
Präprozessor
- Expandiert #include und Makros

Compiler
- Erzeugt Assembler-Quellcode aus C-Sourcecode

Assembler
- Erzeugt Objektcode aus Assembler-Quellcode

Linker
- Fügt (ein oder) mehrere Objektdateien (+ evtl. Libraries) zu ausführbarer Datei (oder Library) zusammen

Loader
- Lädt ausführbare Datei zur Ausführung in Hauptspeicher

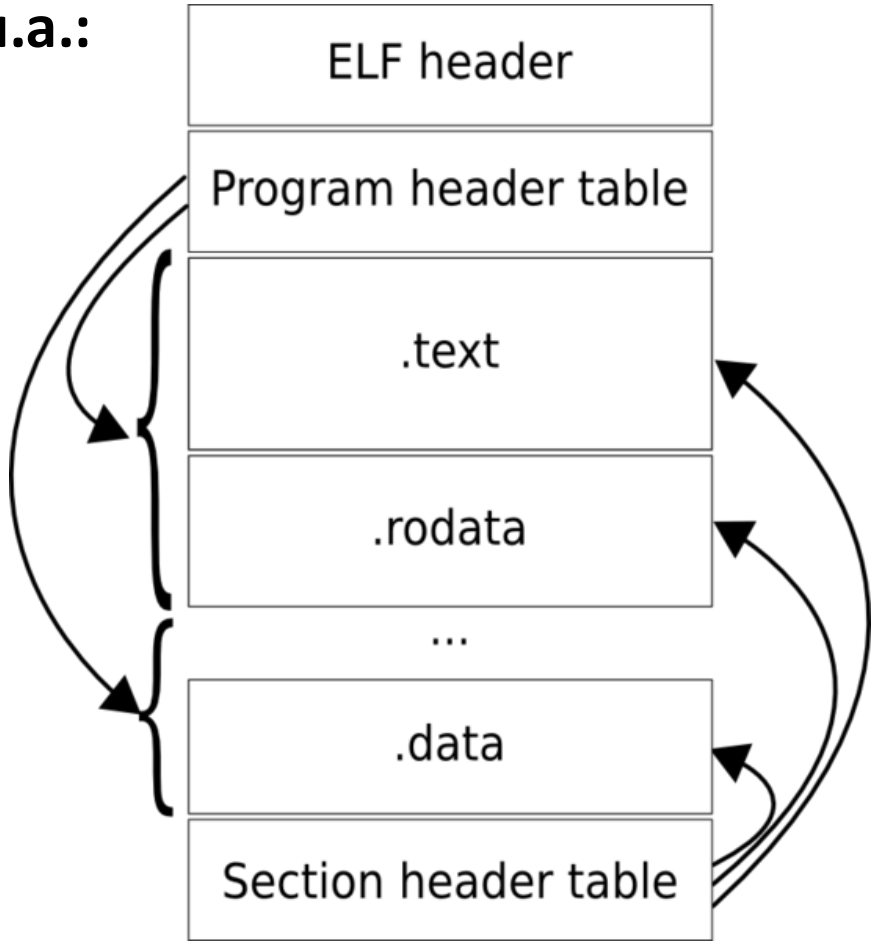


Wiederholung: ELF-Struktur

ELF-Datei besteht aus Sektionen, u.a.:

- Programmtext
- Verschiedene Datensegmente
- Symbole

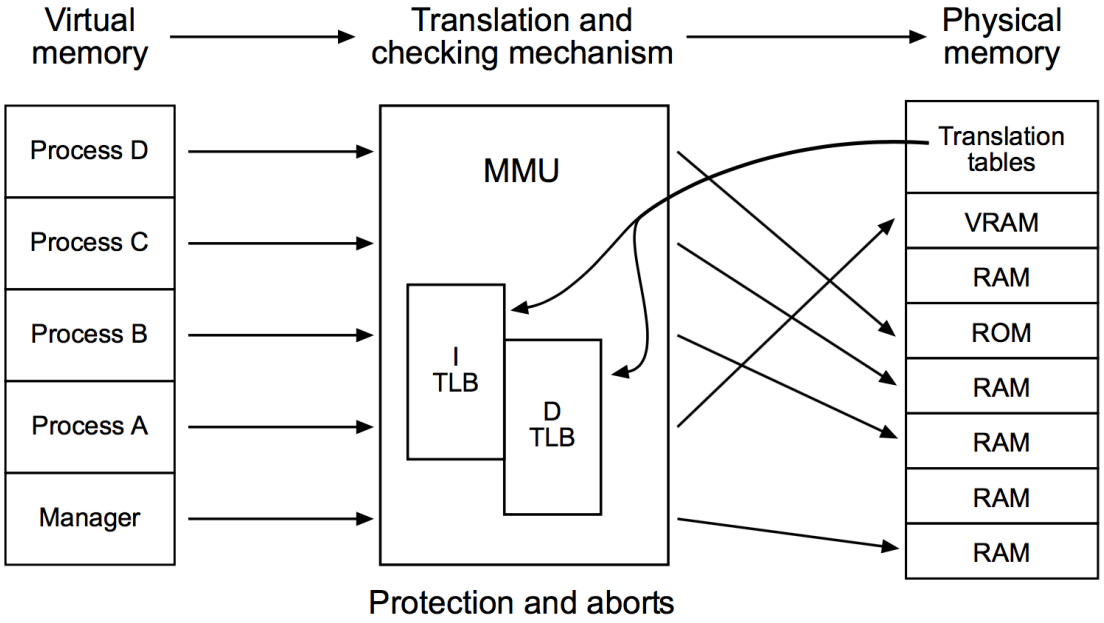
Sektion	Funktion
.text	Maschinencode (Instruktionen) und Einsprungadresse
.rodata	Vorinitialisierte Konstanten
.data	Vorinitialisierte Variable
.bss	Uninitialisierte Daten
.symtab	Adressen zu symbolischen Namen



Virtueller Speicher in Unix/Linux (1)

Linux setzt Vorhandensein einer MMU* voraus

- Übersetzt virtuelle Adressen in physikalische Adressen
  - **Illusion:** Jeder Prozess hat gesamten Adressraum für sich zur Verfügung
  - Schutz von (physikalischem) Speicher vor ungewünschtem Zugriff
  - Granularität: „Seite“ (z.B. 4096 Bytes)

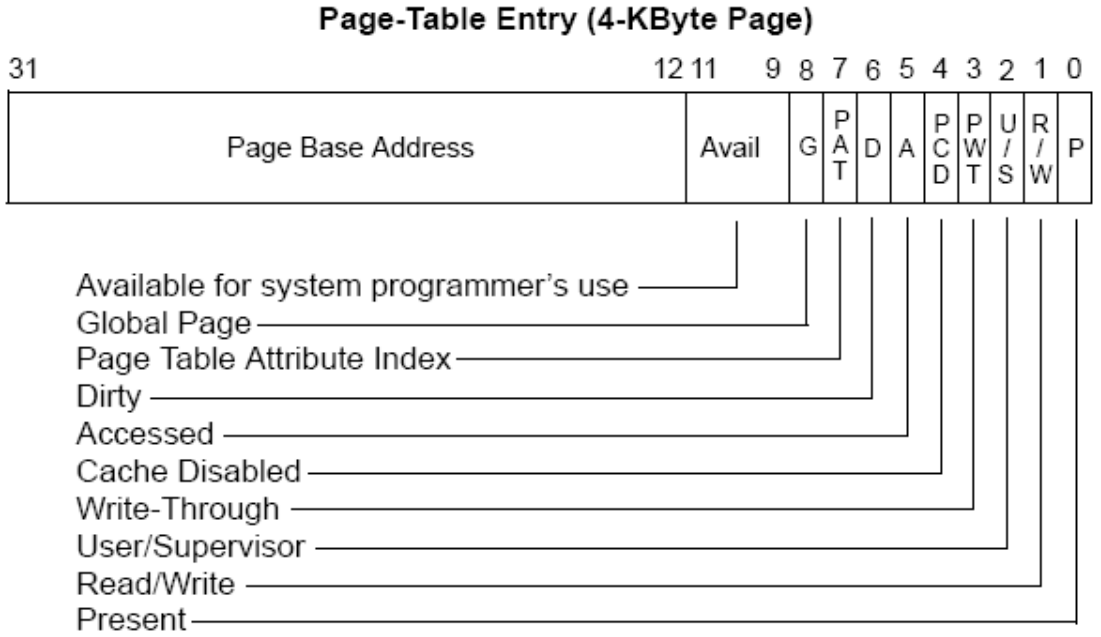


* MMU = Memory Manage-ment Unit

Virtueller Speicher in Unix/Linux (2)

Übersetzung von Adressen über Seitentabellen

- Für jede (abgebildete) *Speicherseite* (page): Adresse des zugehörigen (gleich großen) physikalischen *Seitenrahmens* (page frame)
  - **Sparse mapping:** nur im phys. Speicher vorhandene Seiten abgebildet
  - Seitentabelleneintrag (page table entry) enthält neben Adresse auch Information über *Zugriffsrechte:* Lesen/Schreiben* (und vieles mehr)

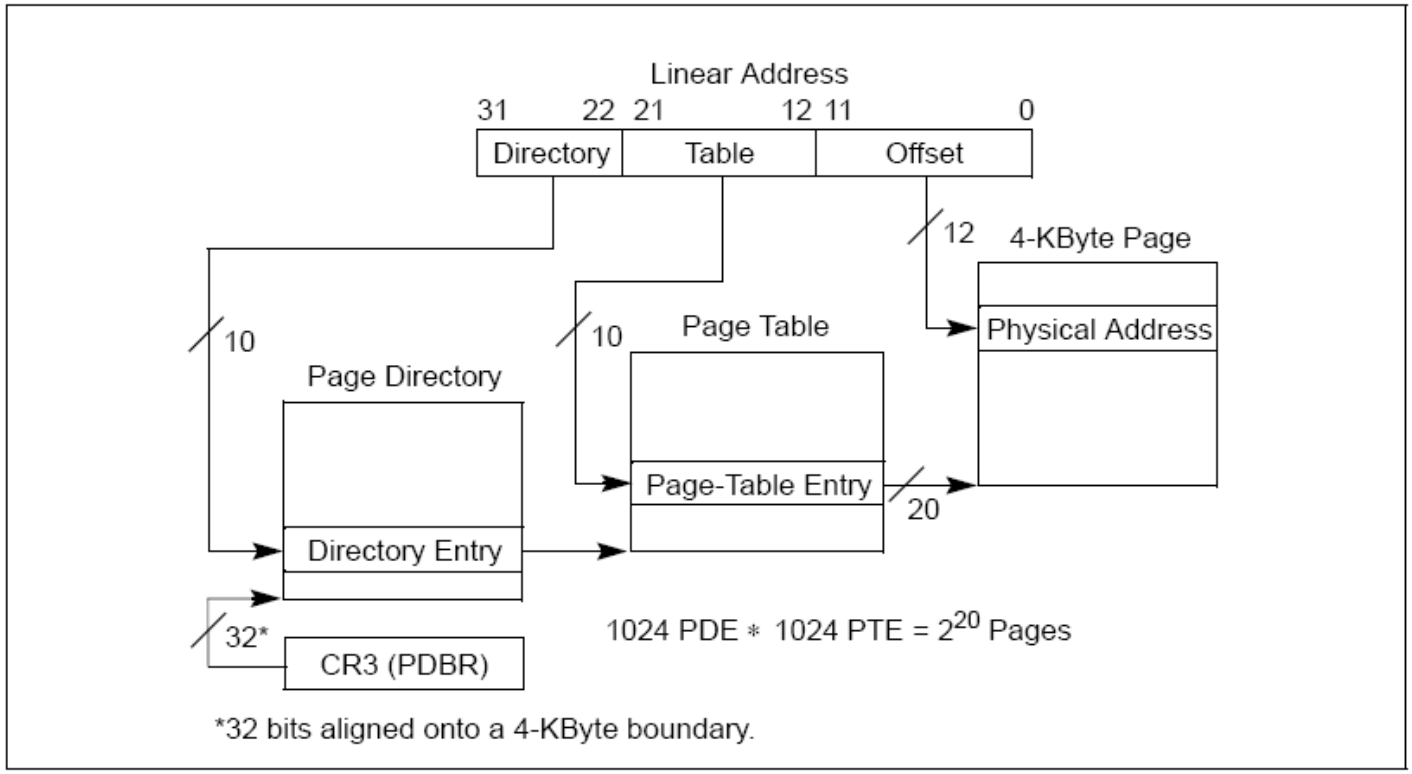


* Bei 32-Bit Intel-CPUs fehlt Bit, das *Ausführen* von Code aus der Seite verbietet!

Virtueller Speicher in Unix/Linux (3)

Struktur der Seitentabelle

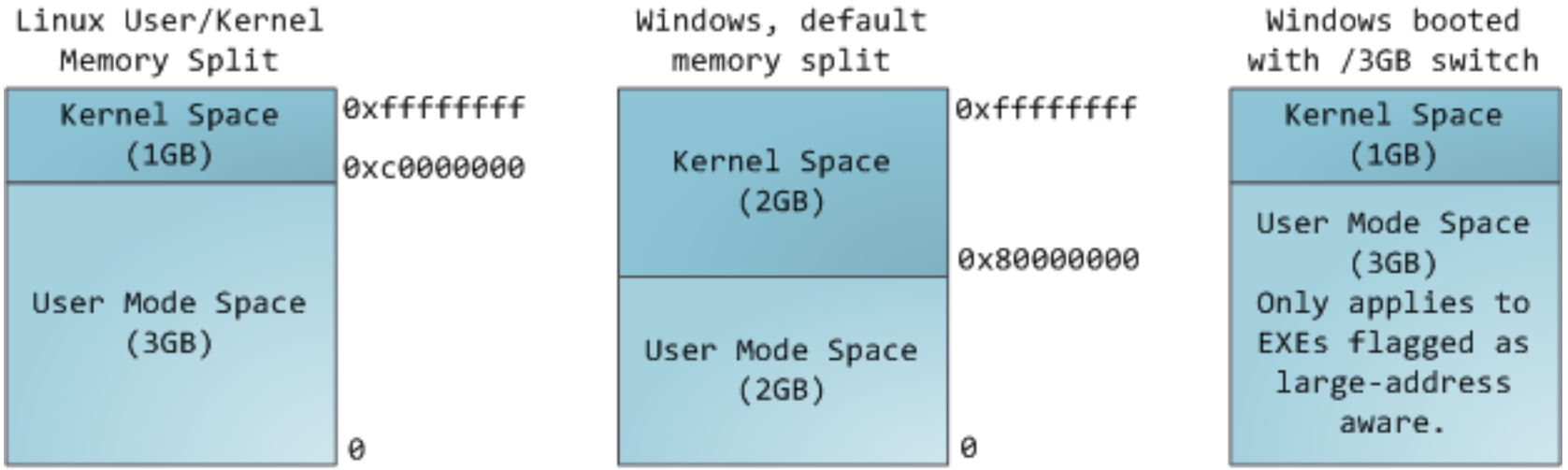
- Seitentabelle (page table) ist Datenstruktur im RAM
- Kleine Ausschnitte im *Translation Lookaside Buffer (TLB)* zwischengespeichert



Virtuelles Speicherlayout von Prozessen (1)

Aufteilung des virtuellen Adressraums (hier: 32 Bit = 4 GB)

- User Mode Space = Prozessadressraum, separat für jeden Prozess
  - Jeder Prozess besitzt eigene Seitentabelle
  - Kernel im oberem Teil (25/50%) **jedes** Prozessadressraums eingeblendet (Ausnahme: MacOS X)

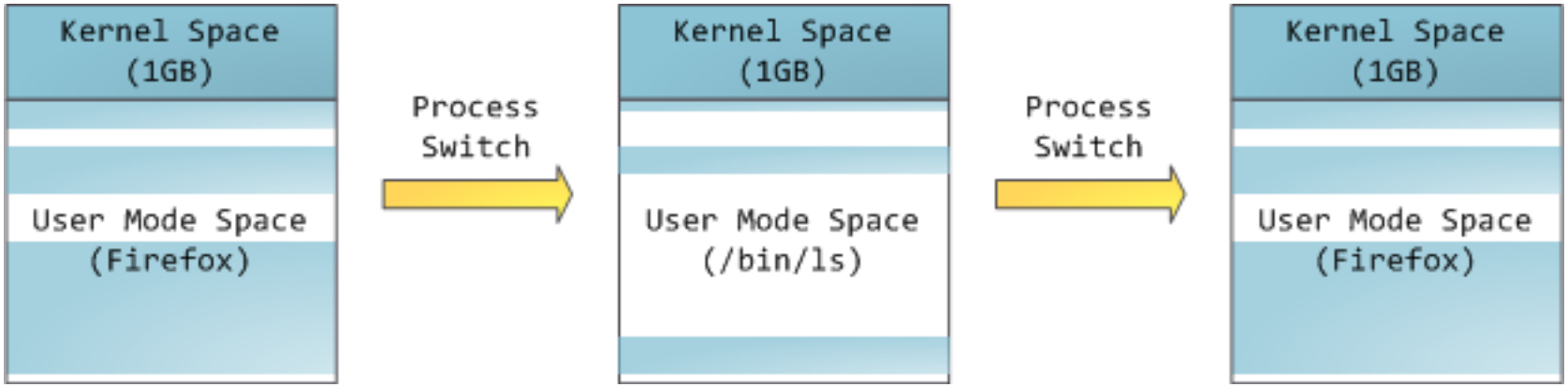


http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/

Virtuelles Speicherlayout von Prozessen (2)

Jeder Prozess besitzt eigene Seitentabelle

- Wird bei Prozessumschaltung vom Kernel umgeschaltet
- Adressraum für den Kernel (oberstes Gigabyte virtuelle Adressen) bleibt immer eingeblendet



http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/

Layout eines Prozesses im Speicher (1)

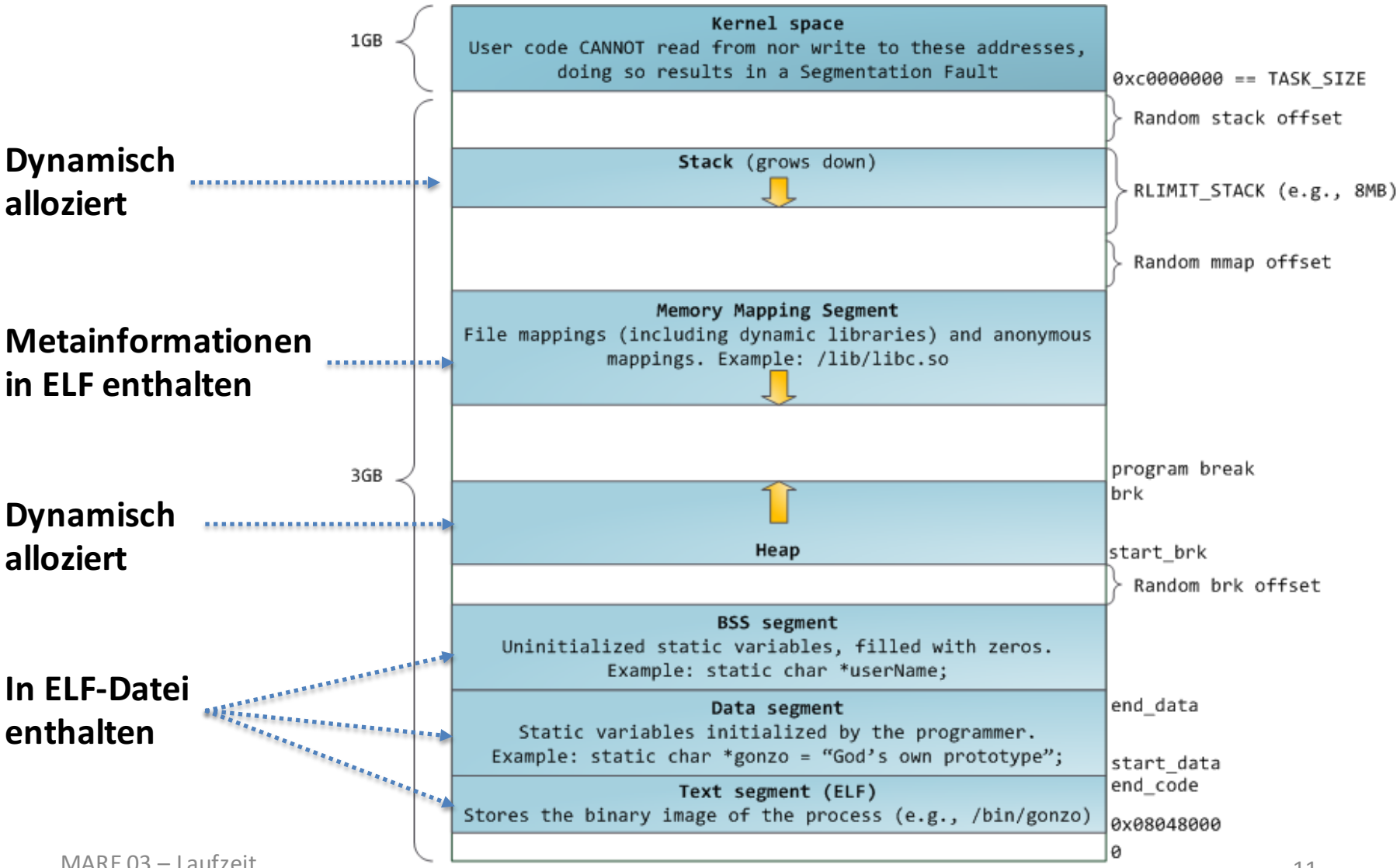
Übergang von Programm zu Prozess

- Definition:

  „Programm = Prozess in Ausführung“

- Prozess = **Laufzeitkontext** eines Programms
- Laden des Programms in RAM
  - Zuteilung *virtueller* Adressbereiche zu Programmsektionen
  - Zusätzlich Reservierung von Adressbereichen für dynamisch wachsende Bereiche
    - *Heap, Stack*
  - Adressbereich oberhalb 3GB (=0xC000 0000) für Programm nicht les-/schreib-/ausführbar (dort ist der Kernel!)

Layout eines Prozesses im Speicher (2)



Definieren des Speicherlayouts eines Programms (1)

HOCHSCHULE COBURG

Linker berücksichtigt *Linker Script*

- Definiert Adressbereiche für ELF-Sektionen
- Zusätzlich auch Einsprungadressen, Reihenfolgen, Alignment, ...

Einfaches Linker-Skript:

```
SECTIONS
{
  . = 0x10000;
  .text : { *(.text) }
  . = 0x8000000;
  .data : { *(.data) }
  .bss : { *(.bss) }
}
```

Start bei *virtueller Adresse* 0x10000:

- Textsektion

Start bei *virtueller Adresse* 0x8000 0000:

- Zuerst .data
- Direkt danach .bss

Definieren des Speicherlayouts eines Programms (2)

HOCHSCHULE COBURG

Kontrolle: Funktioniert das Linker-Skript?

- Ausgabe von Symbolen mit *nm*:

```
$ gcc -m32 -c foo.c
$ ld -m elf_i386 -T foo.ld -o foo foo.o
$ nm foo
0001002c R a
08000000 D b ✓
08000004 B c ✓
00010000 T main ✓
```

Linker-Skript *foo.ld*:

```
SECTIONS
{
   . = 0x10000;
   .text : { *(.text) }
   . = 0x8000000;
   .data : { *(.data) }
   .bss : { *(.bss) }
}
```

Beispielprogramm foo.c:

```
const int a = 42;
int b = 23;
int c;

int main(void) {
  c = a + b;
  return c;
}
```

Definieren des Speicherlayouts eines Programms (3)

HOCHSCHULE COBURG

…Sektion .rodata im Linkerskript nicht angegeben: automatisch alloziert!

```
$ gcc -m32 -c foo.c
$ ld -m elf_i386 -T foo.ld -o fox foo.o
$ nm fox
08000000 R a ✓
08000004 D b ✓
08000008 B c ✓
00010000 T main ✓
```

…geht doch!

Linker-Skript *foo.ld*:

```
SECTIONS
{
   . = 0x10000;
   .text : { *(.text) }
   . = 0x8000000;
   .rodata : { *(.rodata) }
   .data : { *(.data) }
   .bss : { *(.bss) }
}
```

Beispielprogramm foo.c:

```
const int a = 42;
int b = 23;
int c;

int main(void) {
  c = a + b;
  return c;
}
```

Linkerskripte können viel mehr – Anordnen einzelner Objektdateien, Arithmetik auf Adressen, Alignment, Aufteilen zwischen ROM/RAM-Bereichen, …

Definieren des Speicherlayouts eines Programms (4)

HOCHSCHULE COBURG

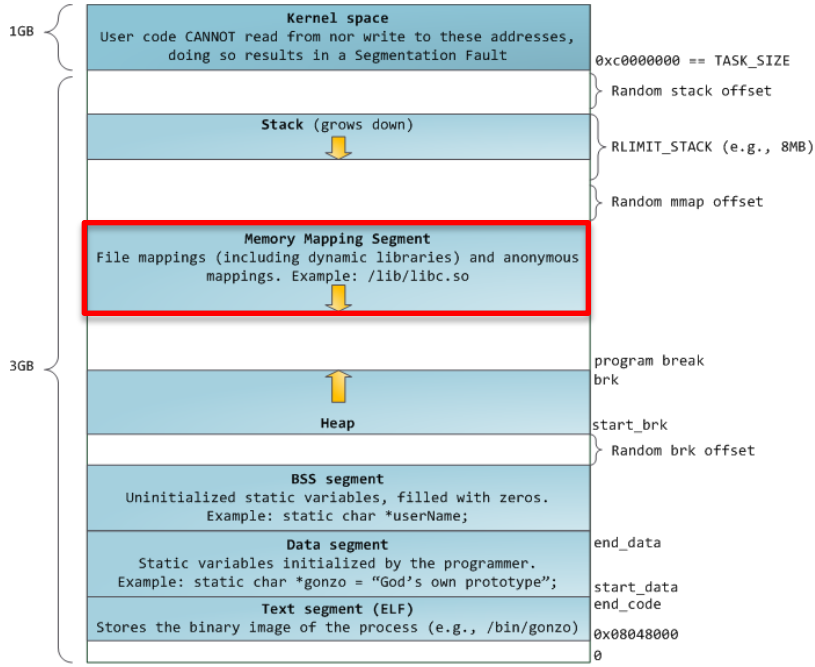
Linker-Skript *foo.ld*:

```
SECTIONS
{
  . = 0x10000;
  .text : { *(.text) }
  . = 0x8000000;
  .rodata : { *(.rodata) }
  .data : { *(.data) }
  .bss : { *(.bss) }
}
```

Kontrolle des Layouts mit readelf:

```
$ readelf -l fox

Elf file type is EXEC (Executable file)
Entry point 0x10000
There are 3 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0x00010000 0x00010000 0x00030 0x00030 R E 0x1000
  LOAD           0x002000 0x08000000 0x08000000 0x00004 0x00008 RW  0x1000
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4

 Section to Segment mapping:
  Segment Sections...
   00     .text
   01     .rodata .data .bss
   02
```

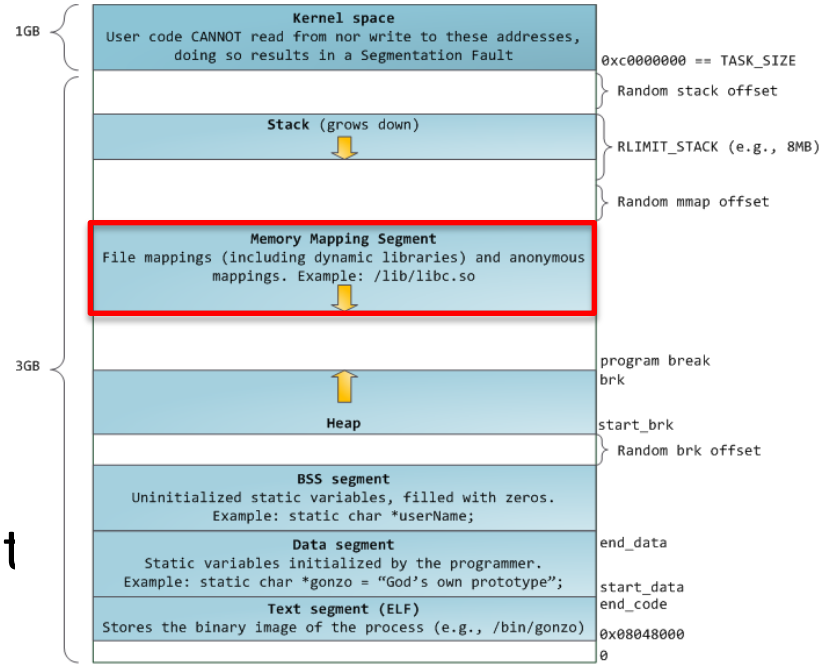
Shared Libraries (1)



Programme sind (meist) nicht standalone

- Geht auch, ist aber aufwendig...
- Libraries enthalten nützliche Funktionalität, z.B.:
  - libc – C-Standardfunktionen
  - libm – Mathematikfunktionen
  - libX11 – Funktionen für X11 Window System
- Viele Programme enthalten die selben Libraries
  - Bei statischem Linken: Kopie von Library-Code ist in jedem ausführbaren Programm enthalten → *benötigt Plattenplatz*
  - Kopie von Library-Code muss für jeden Prozess in Hauptspeicher alloziert werden → *benötigt RAM*

Shared Libraries (2)



Shared Libraries sind ELF-Dateien

- Enthalten üblicherweise kein main
- Nicht in ELF-Datei eines Programms enthaltene Funktionen/Symbole müssen in shared library/ies gesucht werden
- Namensauflösung ist aufwendig
  - Shared libraries verwenden Hashtabellen
- Mehr zu Namensauflösung nächste Woche...

Dynamischer Linker ld.so

Welche shared libraries werden von einem ELF-File referenziert?

```
$ gcc -m32 -o foo foo.o
$ ldd foo
        linux-gate.so.1 =>  (0xb7701000)
        libc.so.6 => /lib32/libc.so.6 (0xb759f000)
        /lib/ld-linux.so.2 (0xb7702000)
```

“ldd” (list dynamic dependencies) gibt Liste der shared libs aus

Weit “oben” im Prozessadressraum: “kurz vor” 0xC000 0000

Dateiname der shared library “libc”

Dateiname des zu verwendenden dynamischen Loaders

ld.so (hier: /lib/ld-linux.so.2 – für ELF)…

- sucht und lädt die shared libraries, die ein Programm benötigt
- bereitet das Programm auf die Ausführung vor
- und startet es

Besondere shared libraries

Was ist linux-gate.so.1?

```
$ gcc -m32 -o foo foo.o
$ ldd foo
        linux-gate.so.1 =>  (0xb7701000)
        libc.so.6 => /lib32/libc.so.6 (0xb759f000)
        /lib/ld-linux.so.2 (0xb7702000)
```

...nur der Vollständigkeit halber...

Zu **linux-gate.so.1** existiert keine Datei!

- Virtuelle shared library, vom Kernel in den Prozessadressraum eingeblendet
- Stellt portable Mechanismen für Systemaufrufe zur Verfügung

Details unter http://www.trilithium.com/johan/2005/08/linux-gate/

Speicherlayout eines Prozesses zur Laufzeit (1)

HOCHSCHULE COBURG

Virtuelles Dateisystem /proc

- Enthält Informationen des Kernels über System- und Prozesszustände, z.B.:
  - Speicherbelegung: /proc/meminfo
  - Systemlaufzeit: /proc/uptime
  - Version des Kernels: /proc/version
- Information über laufende Prozesse in /proc/[pid]/, z.B.
  - /proc/[pid]/cmdline: Kommandozeile beim Aufruf
  - /proc/[pid]/sched: Scheduling-Statistik
  - /proc/[pid]/stack: Aktueller Stackinhalt (Funktionshierarchie)
  - /proc/[pid]/maps: **Belegung des virtuellen Speichers**

Speicherlayout eines Prozesses zur Laufzeit (2)

HOCHSCHULE COBURG

/proc/[pid]/maps

- Beispiel: Speicherlayout für Prozess „foo“ herausfinden

```
$ ps ax | grep foo
10323
```

Wird dynamisch bei Start von Prozess vergeben

Welche Prozess-ID hat foo hier? **10323**

```
$ cat /proc/10323/maps
08048000-08049000  r-xp  00000000  90:5a  10520727  /home/me/f/foo
08049000-0804a000  rw-p  00000000  90:5a  10520727  /home/me/f/foo
b75f8000-b75f9000  rw-p  00000000  00:00  0
b75f9000-b7745000  r-xp  00000000  90:5a  8529693   /emul/ia32-linux/lib/libc-2.7.so
b7745000-b7746000  r--p  0014c000  90:5a  8529693   /emul/ia32-linux/lib/libc-2.7.so
b7746000-b7748000  rw-p  0014d000  90:5a  8529693   /emul/ia32-linux/lib/libc-2.7.so
b7748000-b774b000  rw-p  00000000  00:00  0
b7758000-b775b000  rw-p  00000000  00:00  0
b775b000-b775c000  r-xp  00000000  00:00  0         [vdso]
b775c000-b7779000  r-xp  00000000  90:5a  8529726   /emul/ia32-linux/lib/ld-2.7.so
b7779000-b777b000  rw-p  0001c000  90:5a  8529726   /emul/ia32-linux/lib/ld-2.7.so
bf9ee000-bfa03000  rw-p  00000000  00:00  0         [stack]
```

Das Programm selbst

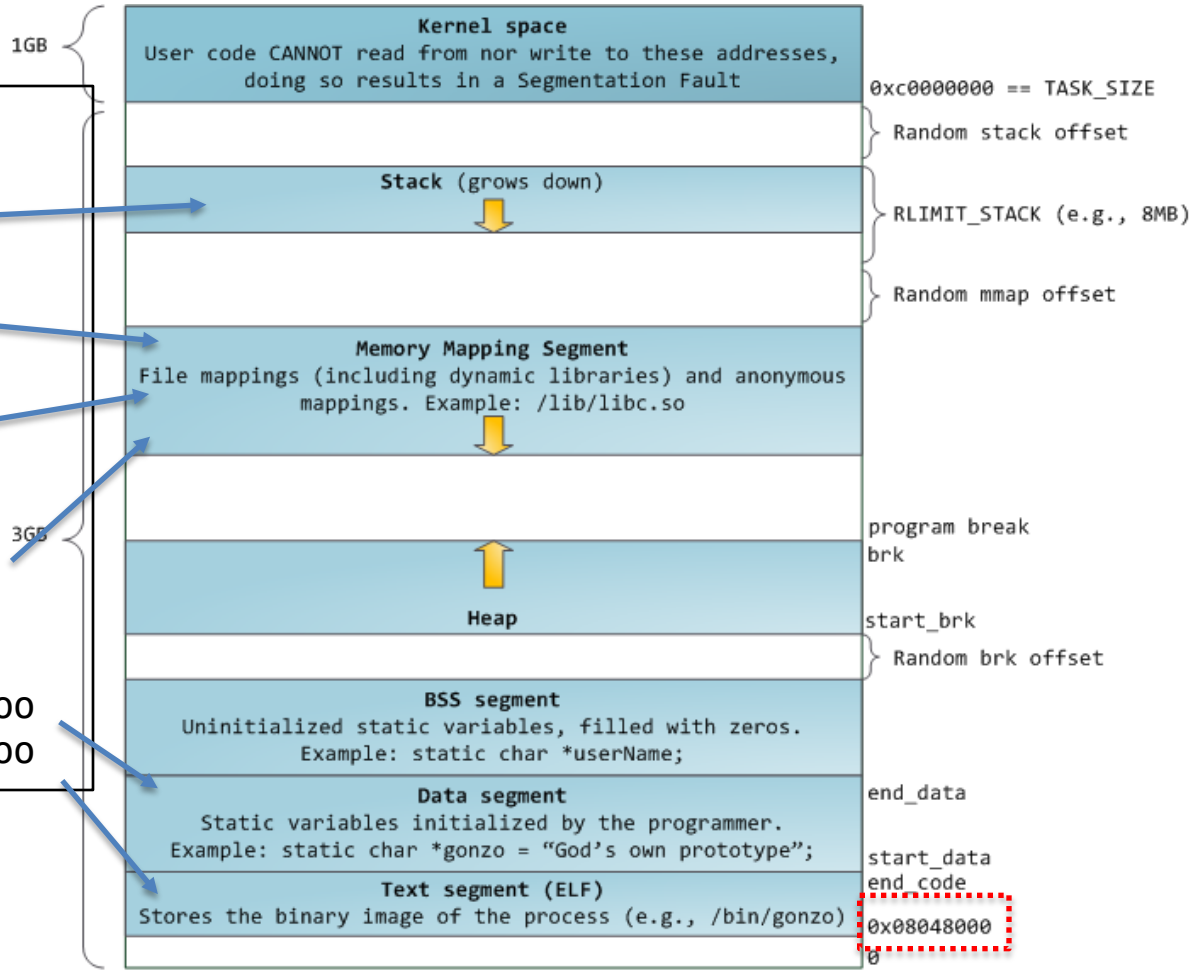
libc

linux-gate.so

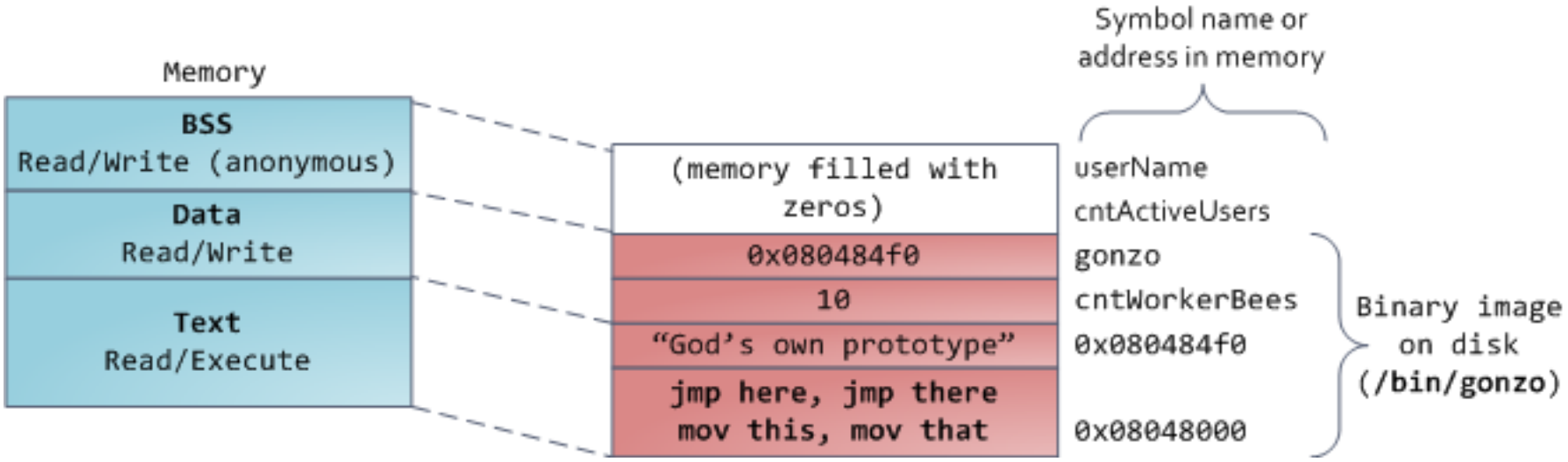
Dynamischer Loader

Speicherlayout eines Prozesses zur Laufzeit (3)

```
...umsortiert und gekürzt...

bf9ee000-bfa03000  [stack]

b7779000-b777b000  ld-2.7.so
b775c000-b7779000  ld-2.7.so

b775b000-b775c000  [vdso]

b7746000-b7748000  libc-2.7.so
b7745000-b7746000  libc-2.7.so
b75f9000-b7745000  libc-2.7.so

08049000-0804a000  /home/me/f/foo
08048000-08049000  /home/me/f/foo
```



Laden:
von ELF-Datei zu Speicherinhalt



http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/

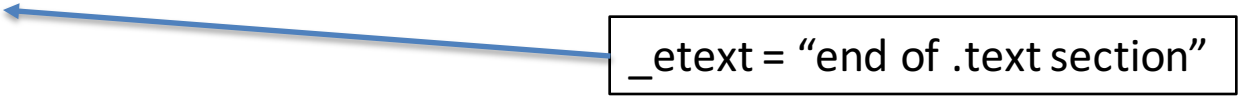
Speicherlayout von C aus ermitteln (1)

HOCHSCHULE COBURG

Von crt0 vordefinierte globale Variable (via Linkerskript):

etext.c

```
#include <stdio.h>

extern char __executable_start;
extern char __etext;

int main(void)
{
  printf("0x%lx\n", (unsigned long)&__executable_start);
  printf("0x%lx\n", (unsigned long)&__etext);
  return 0;
}
```



Ausgabe:

```
$ gcc -m32 -o etext etext.c && ./etext
0x8048000
0x80484a8
```

Speicherlayout von C aus ermitteln (2)

HOCHSCHULE COBURG

Welche Variablen stehen zur Verfügung?

```
$ gcc -m32 -o etext -Wl,--verbose etext.c && ./etext
```

`-Wl,xxx` übergibt Parameter "xxx" an den Linker!

```
... 250 Zeilen Ausgaben (Demo) ...
```

Fazit

Linken und Laden

- Virtueller Speicher und Speicherlayout von Prozessen
  - Definition über Linkerskripte
- Shared Libraries und dynamischer Lader
- Speicherlayout
  - ...und zugehörige Linux-Tools

Literatur

ELF

- TIS Committee, *„Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification“*, Version 1.2
  Online unter https://www.uclibc.org/docs/elf.pdf

Linker

- John R. Levine, *„Linkers and Loaders“*, MKP 1999, ISBN 1-55860-496-0
  Online (beta) unter https://www.iecc.com/linker/
- **Working with libraries and the linker**
  http://www.bottomupcs.com/libraries_and_the_linker.xhtml

ld.so und Shared Libraries

- Ulrich Drepper, *„How to write shared libraries“*
  https://software.intel.com/sites/default/files/m/a/1/e/dsohowto.pdf
- Optimizing linker load times – https://lwn.net/Articles/192624/

ABI-Konventionen

- https://sites.google.com/site/x32abi/