UiT

THE ARCTIC UNIVERSITY OF NORWAY

Project 1

Boot-up Mechanism

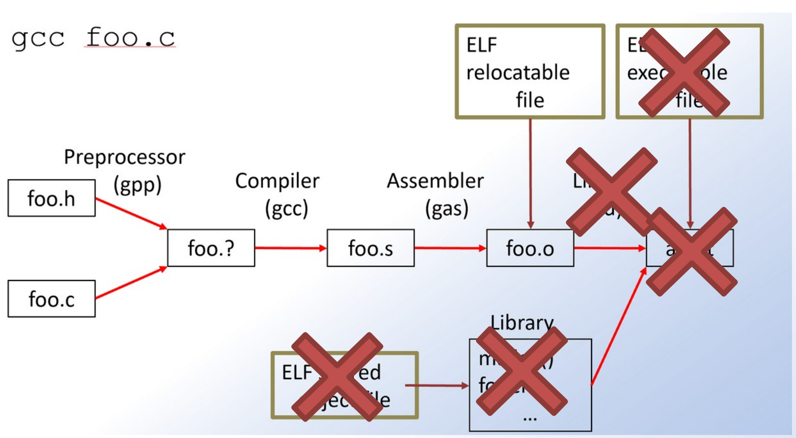
INF-2201 Operating System Fundamentals
Spring 2023

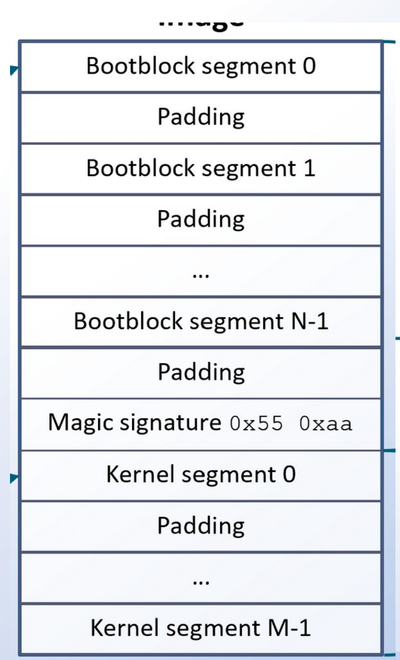
Department of Computer Science
UiT The Arctic University of Norway

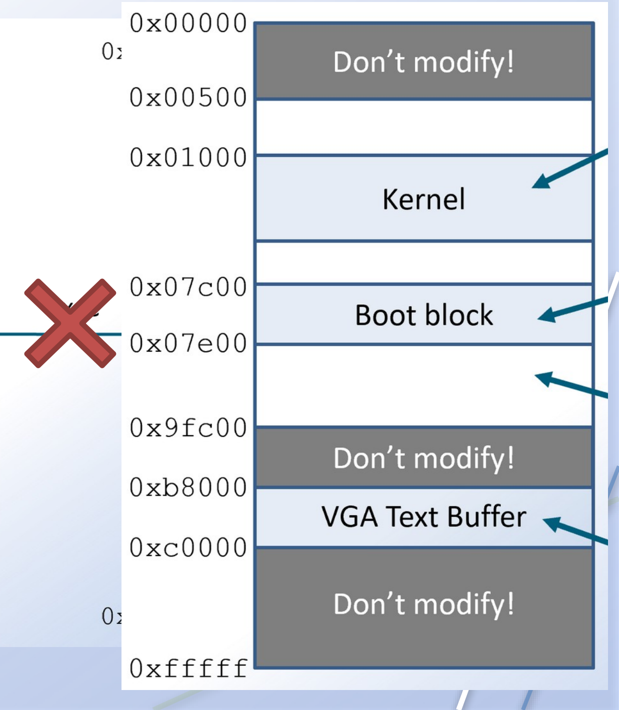
UNIVERSITY OF TROMSØ

Overview

- Create a bootable medium (USB disk) that loads a small «OS kernel».
- To achieve this, you need to:
  - Write a boot block that loads and transfers control to the kernel.
  - Write a «createimage» utility that extracts code and data from compiled binary files to produce a raw, bootable image that can be written to a USB disk.





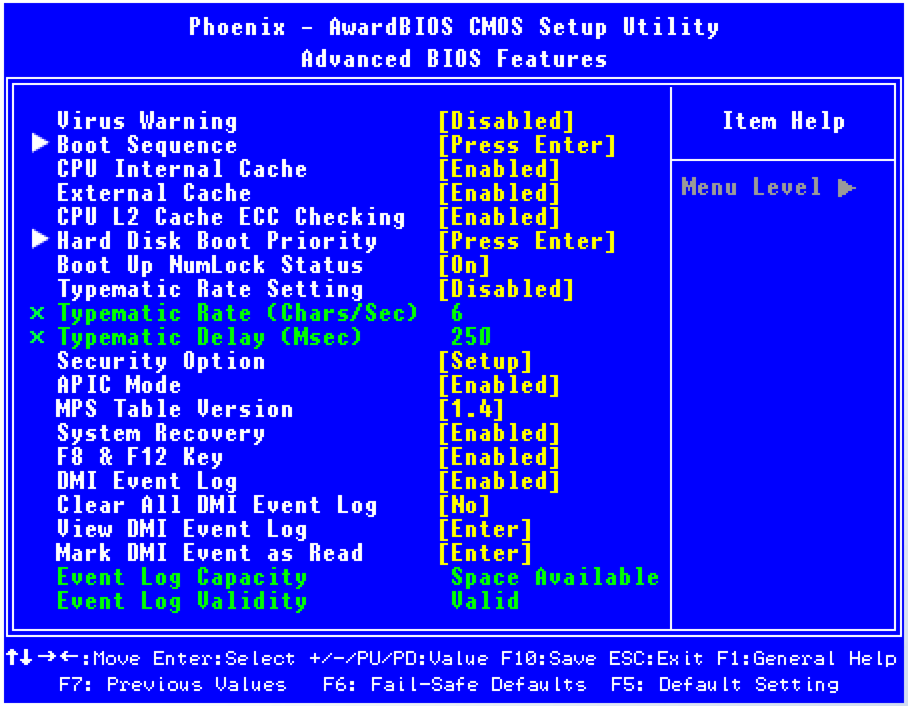


What happens when we turn on the pc?

1. PSU (power supply unit) performs self-tests and asserts the Power Good signal after all voltage outputs have stabilized.
   - The Power Good signal line is connected to the CPU's timer chip, which controls the reset line to the CPU.
   - The timer chip will constantly send a reset signal to the CPU until the Power Good signal is received.
2. CPU starts running after being reset.
   - CPU operates in 16-bit Real Mode.
   - Most registers get well-defined values.
   - The CPU will execute the instruction at the memory address of the *reset vector* (hardcoded address `0xfffffff0`).
   - The motherboard ensures that the instruction at `0xfffffff0` is a jump to address `0xf0000`, which is the start of a 64KB memory location mapped to the BIOS ROM (containing a start-up program).

BIOS

- «That blue screen with the configuration options?»
  - Not quite…



This is the CMOS setup program – an interface for configuring the BIOS.

BIOS

- The “Basic Input-Output System” is the motherboard’s firmware.
- Stored on a non-volatile ROM chip.
  - Traditionally a CMOS chip, today typically EEPROM.
- Responsible for detecting and initializing hardware.
- Provides a service interface for interacting with hardware.

BIOS start-up program

- Power-On Self-Test (POST) is executed.
  - Initialize and test hardware!
  - If POST fails, computer is halted with error.

Power-On Self-Test (1)

1. Detect and test available RAM.
    - If the PC was “warm booted”, memory test may be skipped.
2. Detect CPU type and speed, verify cache memory and CPU registers.
3. Detect keyboard.
4. Test CMOS and verify ROM BIOS checksum.

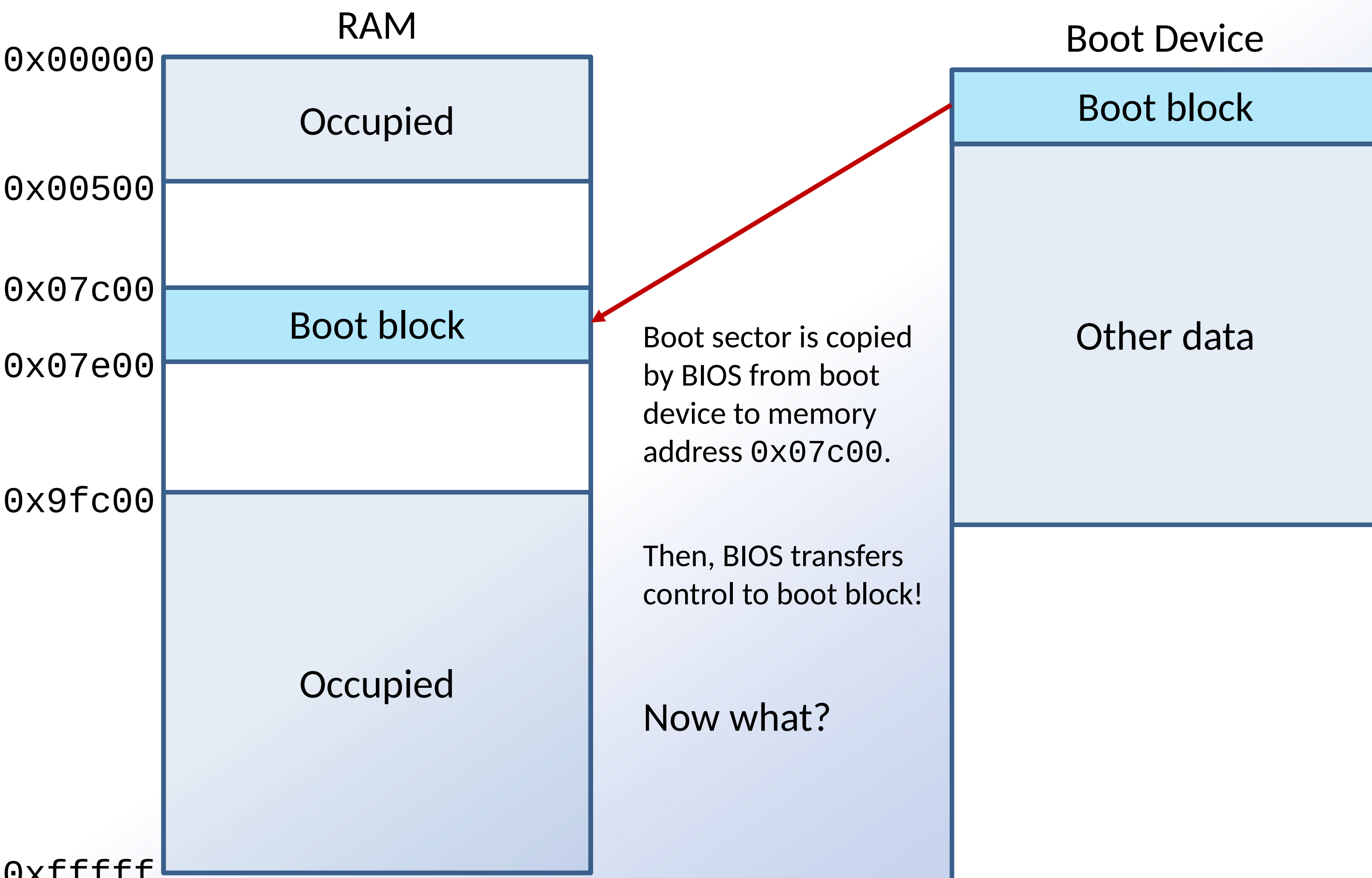
Power-On Self-Test (2)

- Detect and initialize adapters
  - Devices may provide their own ROM BIOS programs, which must be invoked by the system BIOS.
    - Identified in memory by a specific signature as two first bytes: `0x55` `0xaa`.
    - ROMs are aligned to 2KB boundaries.
  - Initialize video adapters and test video memory.
    - Execute VGA ROM BIOS that is located by scanning 2KB blocks within memory area `0xc0000`–`0xc7fff`.
  - Locate and execute general purpose ROMs of other devices, mapped into the expansion area `0xc8000`–`0xdffff`.
  - Locate and execute (if found) BIOS Extension ROM within
  - `0xe0000`–`0xeffff`.
- Initialize logical devices with label, I/O address, and interrupt request line (IRQ).
- Detect and configure PnP devices.
- Display configuration information on screen.

BIOS boot sequence

- After POST, BIOS will search for a bootable medium.
  - E.g. hard drive, USB, CD-ROM, (or floppy!).
  - CMOS stores boot sequence – which devices to test in what order.
  - BIOS looks for a valid boot sector.
    - 512 bytes of data at known location (dependent on device) that is terminated with `0x55 0xaa` as last two bytes.
    - For floppies and hard drives, the boot sector is the first sector on disk
    - (LBA 0 / CHS 0,0,1).
    - (For CDs/DVDs, the boot sector is the 18th sector – LBA 17).
  - If BIOS finds a valid boot sector, it is copied into memory and executed!
    - This means that valid instruction code must be present at the very start of the boot sector!

BIOS bootstrapping



Boot sector is copied by BIOS from boot device to memory address `0x07c00`.

Then, BIOS transfers control to boot block!

Now what?

Boot block

- Boot block is responsible for loading the kernel!
  - That is…
    - Doing necessary initialization (what initialization?).
    - Copying the sectors containing the kernel code and data from the boot device to memory (how do we know how many sectors?).
    - Transfer control to the kernel code.
- It must be written in assembly!
- It cannot use any Linux standard library functions!
- It must operate in 16-bit Real Mode (at least initially…)
  - Now, what is this «Real Mode»?

Real Mode

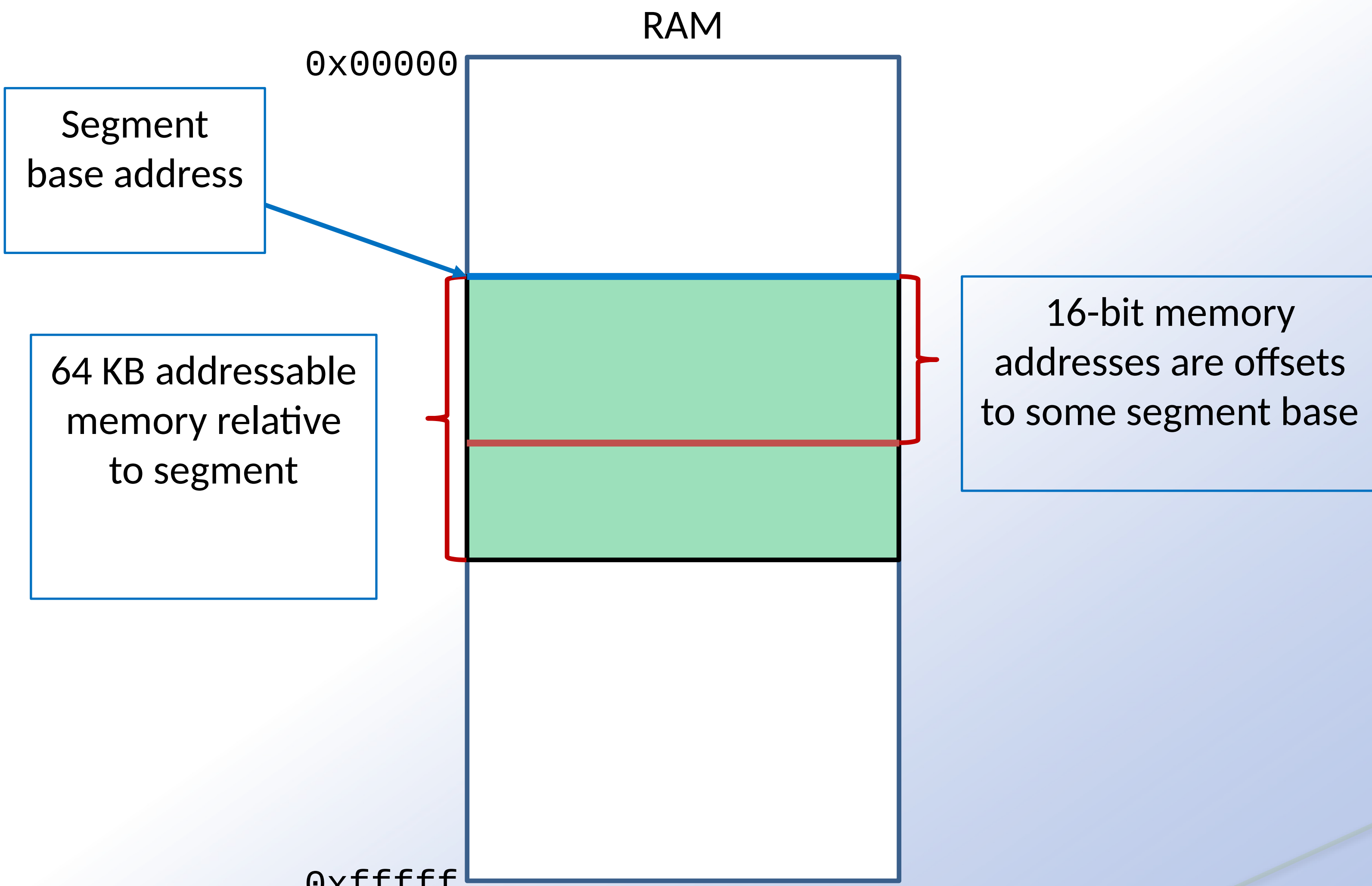
- Real mode is the 16-bit legacy mode in which all x86 family processors operate after being reset (for backwards compatibility).
- Features:
  - 1 MB memory in total! (20-bit linear address space)
  - Memory segmentation (16-bit logical addressing)
  - BIOS services are available through software interrupts ◀◀
    - Makes life easy when dealing with nasty hardware with attitude!
  - Note that even though Real Mode is a 16-bit mode, you can still use 32-bit GPRs[1]!

[1]GPRs = General Purpose Registers

Real Mode Memory Segmentation (1)

- So, we have a 20 bit linear address space, but it’s only 16-bit logically addressable!
  - Whaaat…?!?
- Segmentation – addressing is done *within* a segment.
  - Segment = a limited address subspace.
  - Logical memory addressing is relative to the linear *base address* of the segment (instead of always being relative to `0x0`).
  - A Real Mode segment is defined by a 16-bit segment selector, stored in a segment register. This selector is the 16 MSB of the 20-bit segment base.
  - Logical memory addresses are 16-bit positive offsets to the base address of a segment.
  - A segment thus spans 64KB (bytes).
    - Logical address range `0x0000` – `0xffff` as offsets relative to a segment base.

Real Mode Memory Segmentation (2)



Real Mode Memory Segmentation (3)

- Segmented memory addresses are denoted `segment:offset`.
  - E.g. `0x0000:0xabcd`, `0x0100:0x0000`, `0x0def:0x1234`
- How linear addresses are computed:

$$\underbrace{\text{Linear address}}_{\text{20-bit linear address}} = \underbrace{(\overbrace{\text{Selector}}^{\text{16-bit segment selector}} \ll 4)}_{\text{20-bit segment base}} + \overbrace{\text{Offset}}^{\text{16-bit logical address}}$$

- Offset is positive (zero-extended to 20-bit, not sign-extended).
- Linear addresses correspond to physical addresses.

Real Mode Memory Segmentation (4)

- Examples
  - `0x0000:0xabcd = 0x0abcd`
  - `0x0100:0x0000 = 0x01000`
  - `0x0def:0x1234 = 0x0f124`
- Test yourself
  - Can `0xabcde` be addressed relative to segment with selector `0xabd0`?
    - No, because `0xabcde` < base address `0xabd00`.
  - Can `0xea9bf` be addressed relative to segment with selector `0xda9c`?
    - Yes, `0xda9c:0xffff = 0xea9bf`.
  - Can `0x4bd93` be addressed relative to segment with selector `0x3bc9`?
    - No, because `0x4bd93` is outside of range `0x3bc90`–`0x4bc8f`, addressable relative to selector `0x3bc9`.

Real Mode Memory Segmentation (5)

- Segment registers
  - CS, DS, ES, FS, GS, SS
- Code segment
  - CS is code segment selector.
  - Used (only) by instruction pointer (IP).
  - CS:IP addresses the instruction being fetched.
  - As with IP, CS cannot be set directly, but it can be set indirectly by doing a long jump (`ljmp CS, IP`).
  - Example:

```
ljmp $0x2000, $0x0002 # CS = 0x2000, IP = 0x0002
# Next instruction is 0x2000:0x0002 = 0x20002

#<0x20002>:
jmp  $0x0009          # CS = 0x2000, IP = 0x0009
# Next instruction is 0x2000:0x0009 = 0x20009
```

Real Mode Memory Segmentation (6)

- Data segments
  - DS is data segment selector.
  - Used by default for most data accesses other than stack.
    - For example
      - `movw (%ax), %bx`
    - corresponds to
      - `BX = memory[DS << 4 + AX]`
    - That is, `(%ax)` is actually `%ds:(%ax)`.
  - It is also possible to use any of the other segment selectors – ES, FS and GS – instead of DS, by explicitly specifying the desired selector in the instruction.
    - For example
      - `movw %es:(%ax), %bx`

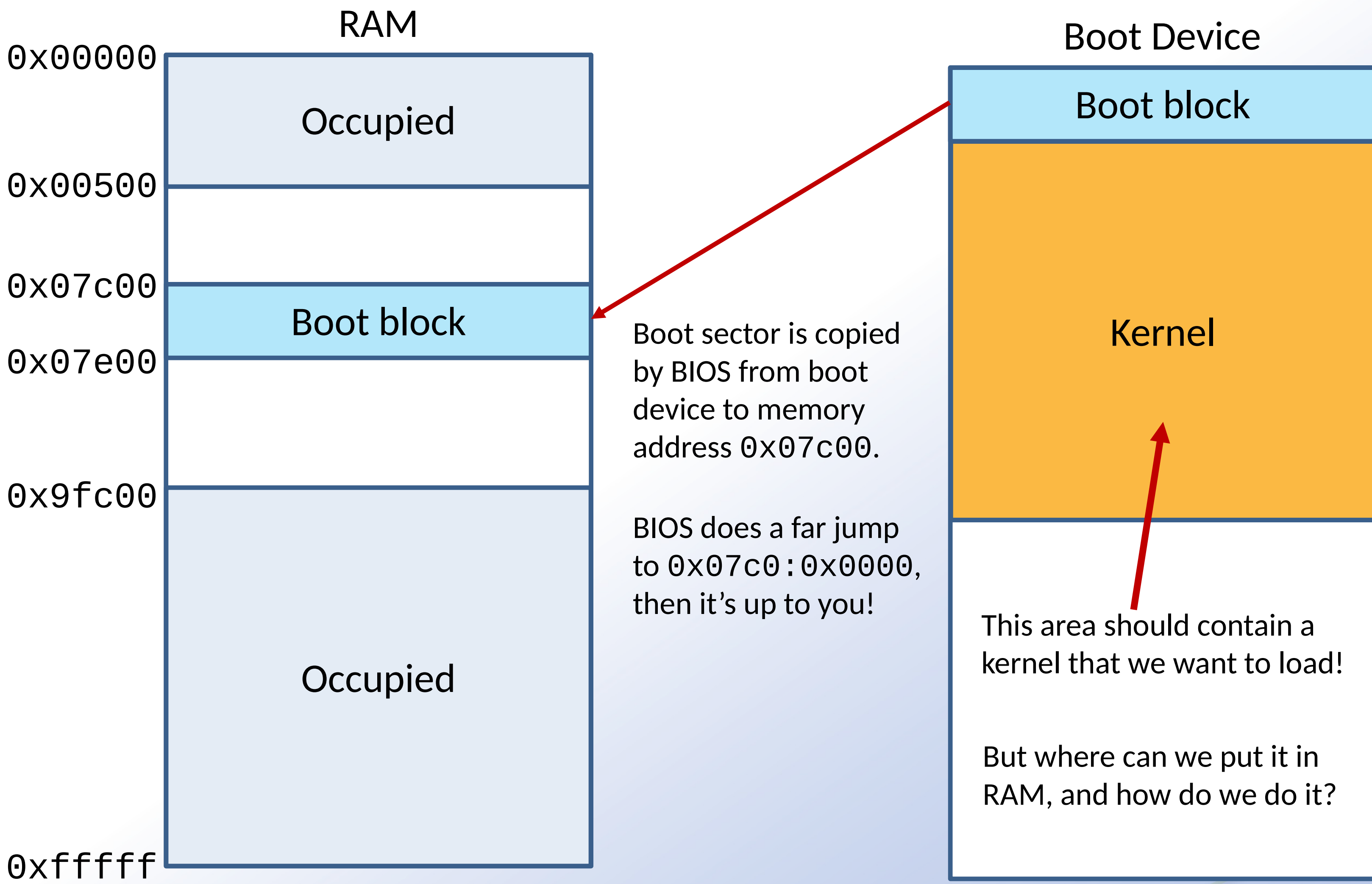
Real Mode Memory Segmentation (7)

- Stack segment
  - SS is stack segment selector.
  - Used (only) by stack pointer (SP) and base pointer (BP).
  - `SS:SP` is the linear address of the “top of the stack”.
  - `SS:BP` is the linear address of the stack frame.
  - The maximum stack size is limited by the segment size.
  - Example:
      - `pushw %ax`
    - corresponds to
      - `subl $2, %sp`
      - `movw %ax, %ss:(%sp)`

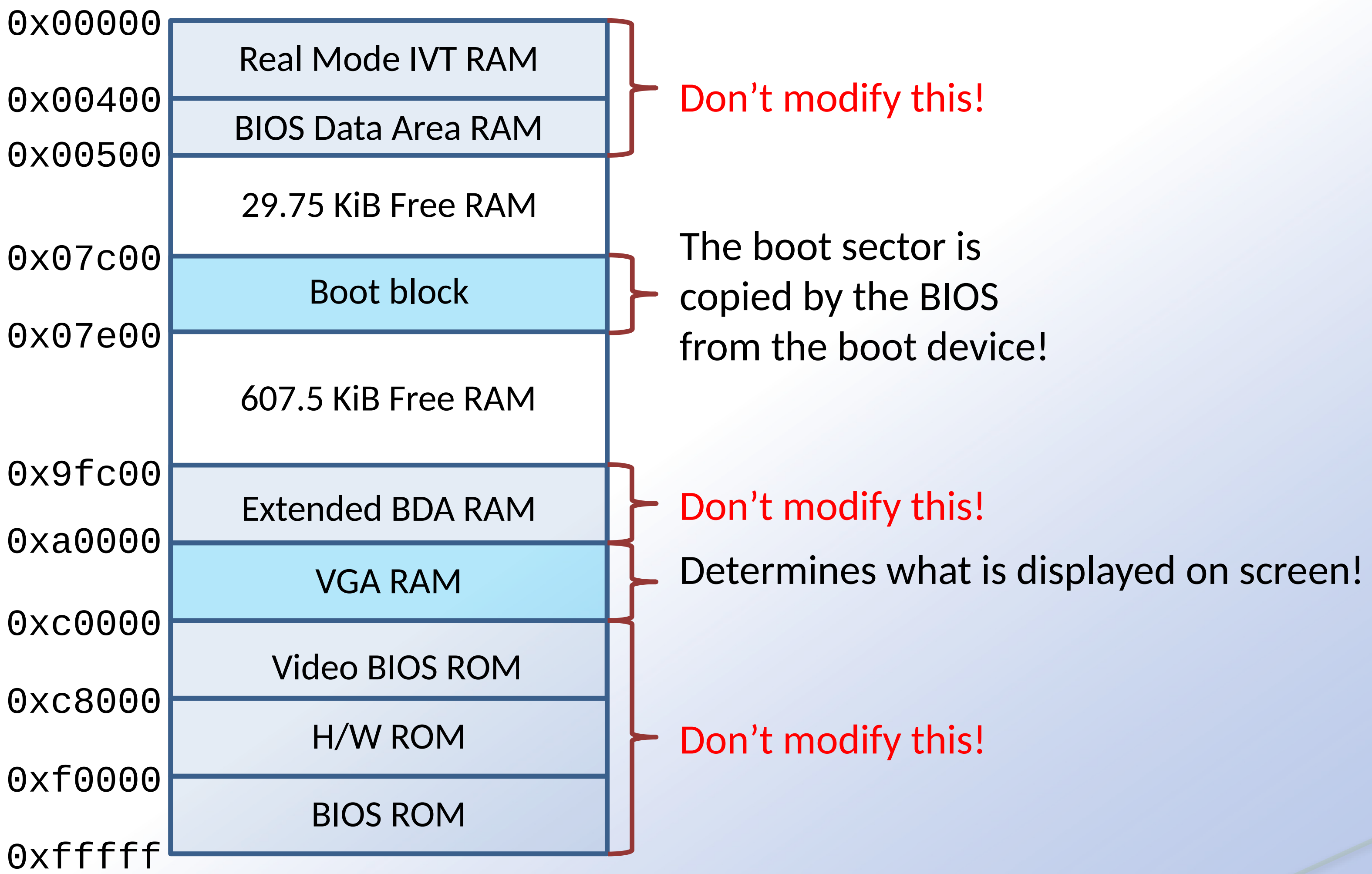
Real Mode Memory Segmentation (8)

- Assigning values to segment selectors
  - …cannot be done directly. The value must be passed through a register!
    - For example
      - `movw $0xabcd, %es`
    - is invalid, whereas
      - `movw $0xabcd, %bx`
      - `movw %bx, %es`
    - is allowed.
  - Remember that `CS` can only be set implicitly by a far jump, far call (or similar).

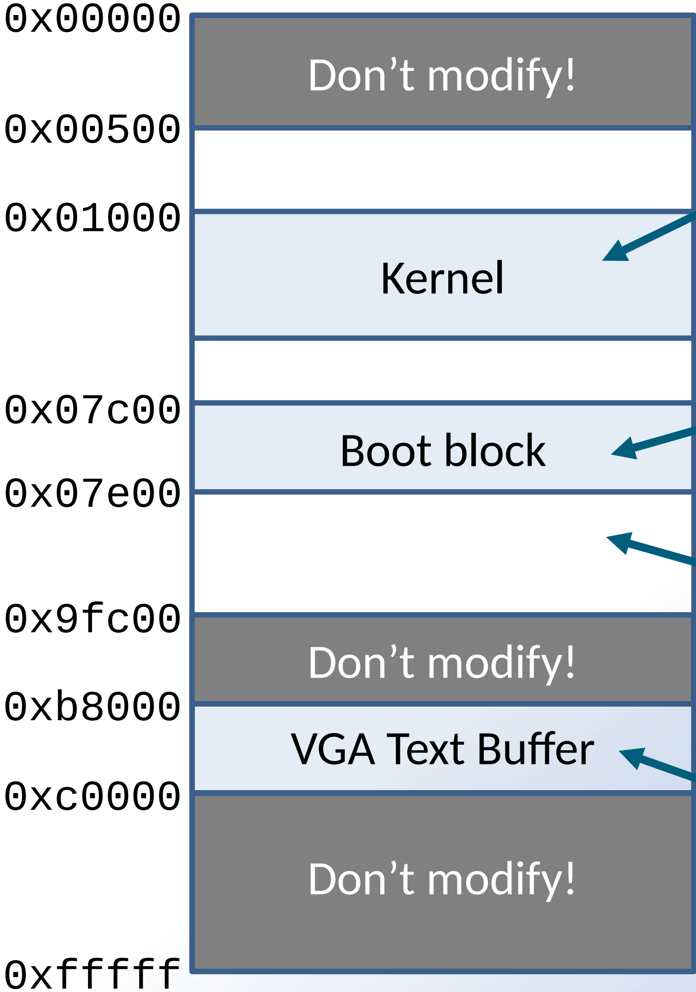
BIOS bootstrapping revisited



Real mode memory map (not to scale)



Memory map simplified (not to scale)



The kernel code and data shall be copied from the boot device to memory address `0x01000`!

The boot sector contains the code (written by you) for copying the kernel into memory and transferring control to it!

We also need a stack, and it should be placed somewhere around here…

We can write to the VGA text buffer to change the characters displayed on the screen and their color attributes.

Loading the kernel

- We know where the kernel begins on boot device!
  - The boot block is the first sector, and it occupies exactly one sector.
  - Kernel starts at the second sector (LBA 1 / CHS 0,0,2).
- The BIOS will help us transfer sectors from a hard drive to RAM.
  - You need to read up on BIOS software interrupts, INT 0x13 and CHS addressing (use Google, Wikipedia, osdev.org, etc.)
  - Make sure to save initial value of `DL` register, as it contains boot drive number when control is given to bootblock, which is later needed as argument to BIOS functions.
- How do we know how large the kernel is?
  - We write the size (number of sectors) to a pre-determined location in the boot block (this will be done by the *createimage* utility).

bootblock.S

```
.equ …                         # Corresponds to defines in C
.text                          # Code segment
.globl _start                  # The entry point must be global
.code16                        # 16-bit code (Real Mode)
.org 0x0                       # Code starts at address 0x0

_start:                        # Entry point, logical address 0x0
    jmp beyondReservedSpace # <-- How many bytes is this instruction?

kernelSize:
    .word 0

beyondReserve
    # Rest of
```

16 bits = sizeof(short int). The kernel size is written to this location by createimage. In the assembly code, you can dereference the data at this location as `(kernelSize)`.

Bootblock – summary

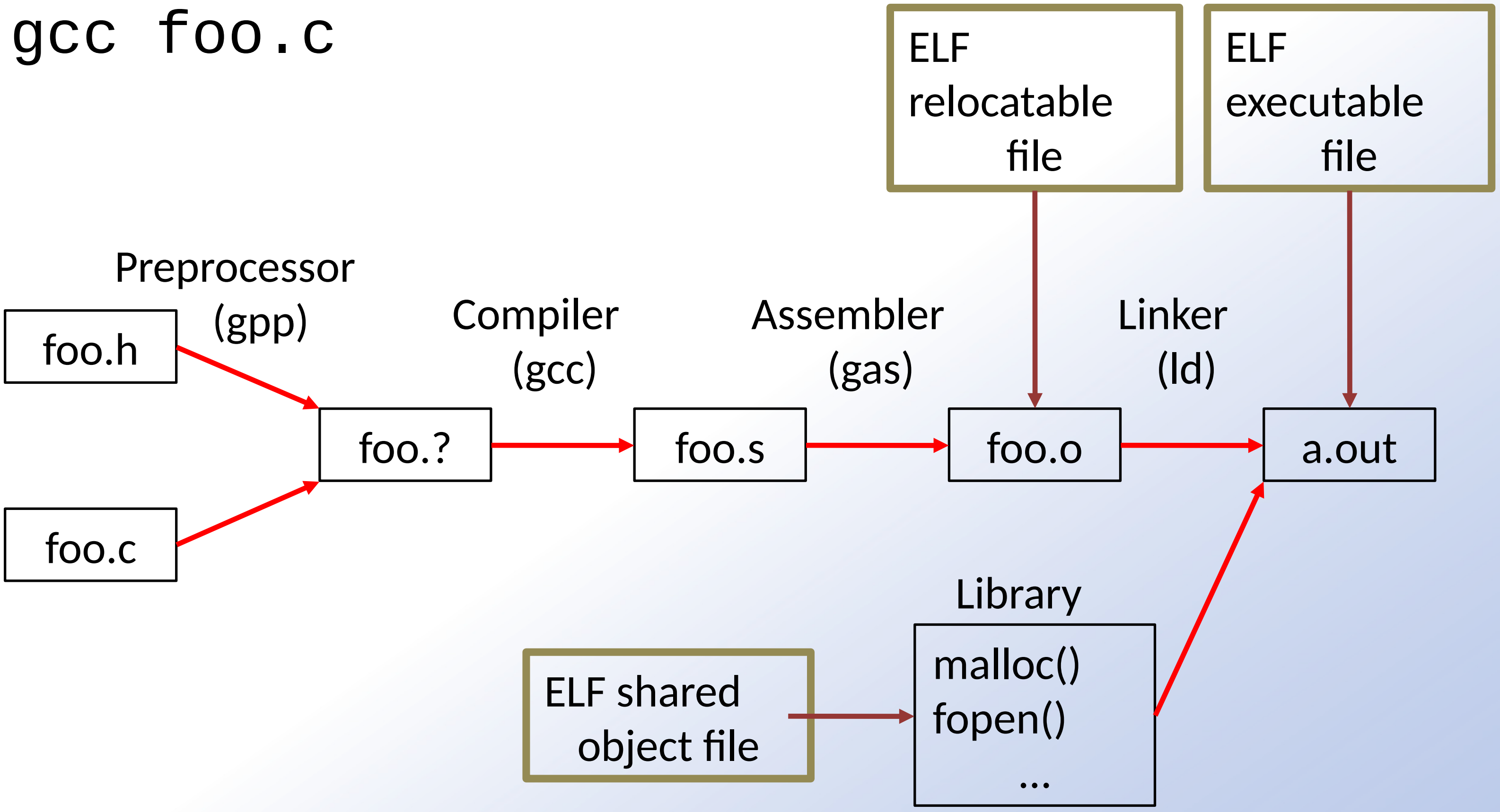
- ~100 lines of assembly code (and comments).
- GNU assembly: AT&T syntax.
- No standard library available.
- 16-bit Real Mode.
- You need to:
  - Setup data segment for boot block.
    - (Code segment is already set to `0x07c0` by BIOS).
  - Setup stack segment and stack.
  - Copy kernel to `0x01000`.
  - Setup data segment for kernel.
  - Long jump to kernel.
    - `ljmp $KERNEL_SEGMENT, $KERNEL_OFFSET`
  - That’s it! ⏪
  - Make sure to comment your assembly code extensively!

How to create the bootable image (1)

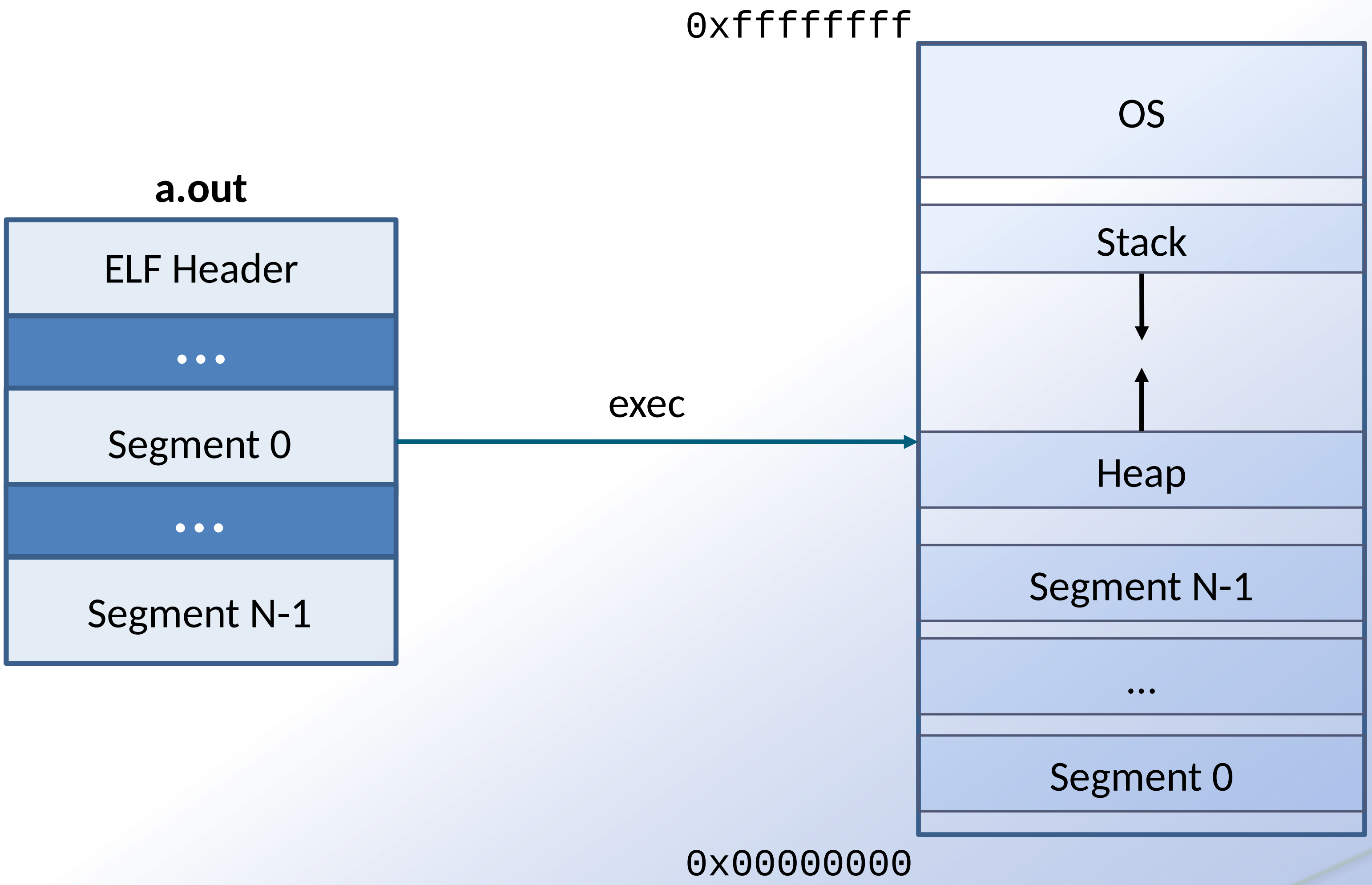
- First, we compile the boot block and the kernel separately, using the GNU compiler toolchain.

What happens when we compile a C program using gcc?

```
$ gcc foo.c
```



Loading an ELF executable (Linux)



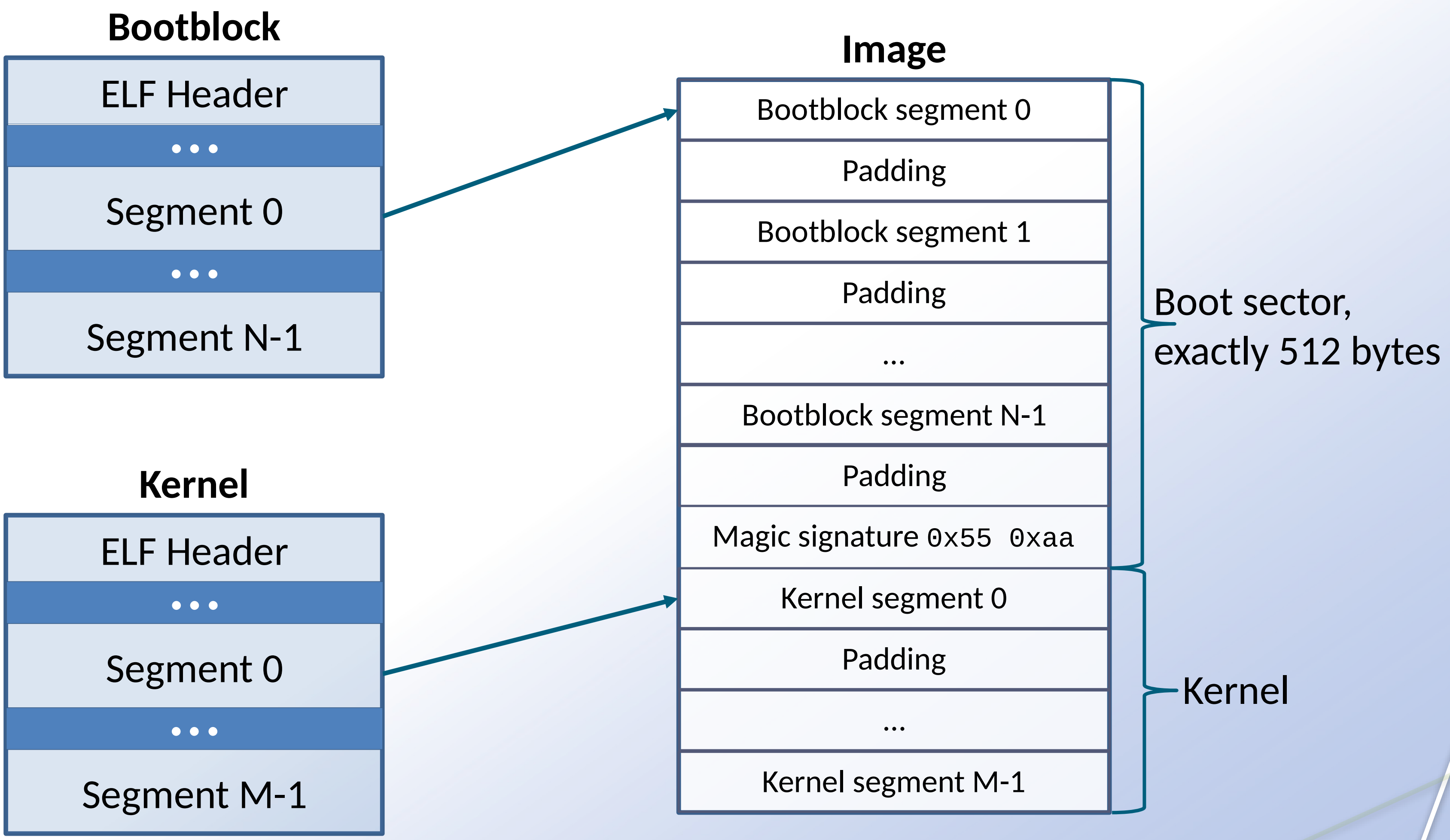
ELF format

- ELF = Executable and Linkable Format.
- Object file format on Linux.
- Contains
  - Raw instruction code and data *segments* (not to be confused with Real Mode memory segments).
  - Address of where in RAM each segment should be loaded.
  - Other stuff like import/export symbol tables, metadata, etc.
- You have to read up on parts of the ELF specification!
  - Available in git repo.

How to create the bootable image (2)

- First, we compile the boot block and the kernel separately, using the GNU compiler tool chain.
- Then we must parse the produced ELF files to extract code/data segments, and write these to an image file.
- Finally, the contents of the image file are copied as raw data to the USB disk.

How to create the bootable image (3)



How to locate and copy ELF segments

- Read the ELF header to find size and file offset of Program Header Table.
- Each entry in the Program Header Table contains information about one segment.
  - File offset
  - Size
  - Memory address
- Make sure the segments are padded correctly when written to image file.

Createimage

- Syntax:
  - `createimage [--extended] <bootblock> <file1> [<file2> ...]`
- Notice that even though we only will use the createimage utility with boot block + a single file (kernel) in the first assignment, the implementation must be generic and support an arbitrary number (>= 1) of files in addition to boot block.
- How should the files be positioned/padded?
  - Hint: how are data loaded from disk to RAM?

createimage.c – summary

- ~200 lines of C code.
- Createimage is a Linux utility, so you can use any standard libraries you want.
- Use `Elf32_Ehdr` and `Elf32_Phdr` data structures provided by `elf.h`.
  - Use `fseek()` and `fread()` to read the data.
  - Example:
    - `Elf32_Ehdr hdr;`
    - `FILE *fp = fopen(filename, "rb");`
    - `int ret = fread(&hdr, sizeof(Elf32_Ehdr), 1, fp);`
- Write code and data segments to image file.
- Write magic signature `0x55 0xaa` to end of first sector of image file.
- Write kernel size to the memory area in the image file corresponding to `(kernelSize)` in the boot block.

Design review

- For each assignment you are required to give a short presentation of how you are solving it!
  - Need not be formal – just you and TA!
- Come prepared!
  - We recommend you bring notes on paper or screen

- You must have figured out all details before the design review
- You must show us convincingly that you can solve the assignment
- Pass / No Pass grading

Some questions for the design review

- What if boot block needs to be larger than 512 bytes?
- What if kernel is larger than 54 sectors (and hence needs the memory where the boot block resides)?
- How does CHS addressing work?
  - How do we know the number of tracks, sectors and heads?
- How do we navigate the ELF file to locate the segments?
- How do we write segments to image file with regards to position and padding?
- How do we write several files to image file with regards to position and padding?
- How do we know how large the kernel code/data is?
- Where in the image file should we write the kernel size?
- When and why must we set the DS segment?
- What should the kernel DS be set to?
- What should the stack segment and GPRs be set to?

Bochs

- You can use the bochs emulator for development.
- TA’s will help set it up for you.

Code – Formalities

- Code must run properly after being compiled on the provided test computers
  - Doesn't matter if it compiles on your machine, if we can't get it to work on the provided test computers
- Don't look at other people's code
- Write structured "clean" code
  - Unstructured code will count negatively
- Don't look at other people's code
- Comment your code properly
  - Uncommented code will very likely not be taken into consideration when evaluating your assignment
  - Show that you know what you're doing
- Don't look at other people's code

Report

- Code comments should cover most (implementation) details.
- Max 4 pages.

Hand-in – Formalities

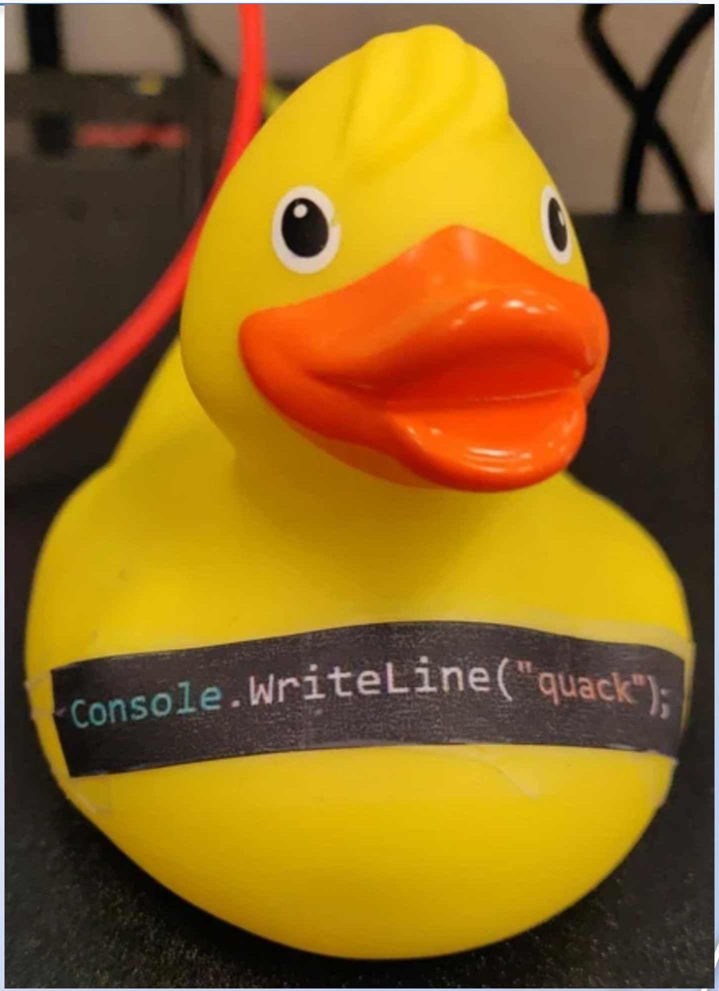
- Hand-in will be done using GitHub
- You will receive a link which creates a private repository containing pre code
- Your latest pushed commit at deadline will be graded
  - So remember to push, push, push!
- No extensions will be granted, or late submissions accepted
  - Unless reason for late submission is documented, e.g. on medical grounds!

Feedback on your assignment

- Some numbers:
- 64 students
- 4 TA's
- 1-2 persons grading all assignments (x3)

➔You should rely on your TA for feedback

- How do I know my solution is correct?
- Can you add some unit tests/ assertions to you code?
- Does it run the provided test cases (processes)?
- Can you change some parameters to stress-test your code?
- What would you like someone else to check?
  - Ask TAs



Word of advice

- Start in time (ASAP)!
- Read the assignment text and precept carefully!
- Read and understand the pre-code before writing your own code!
- Read the assignment text and precept multiple times!