x86 Real/Protected Mode Programming Tips for MEMOS

CS552 – Operating Systems
02/27/2020

Sasan Golchin

Today’s agenda

- MEMOS-1
  - x86 real-mode programming
  - Address translation @ link-/run-time
  - Useful BIOS services
  - Example
  - Linking a flat binary program
- MEMOS-2
  - x86 protected-mode programming
  - Jumping into C from assembly
  - GRUB and the Multiboot standard
  - Example
  - Linking an ELF program
  - GRUB configuration to run our kernel!

MEMOS-1: Overview

- It's a Master Boot Record (MBR) program
  - Located at the first sector of the bootable device (Disk, USB, …)
  - Limited to `512` bytes in size ending in signature bytes `0x55` and `0xAA`
  - The signature tells the firmware that it's a valid runnable MBR, not some random data!
  - Copied into a conventional address (`0x7C00`) by the firmware (BIOS, UEFI)
  - When finished initializing the system, BIOS makes a jump to `0x7C00`
  - Your work starts here!
- Your MBR's objective
  - Detect the memory layout
  - Print it on the screen

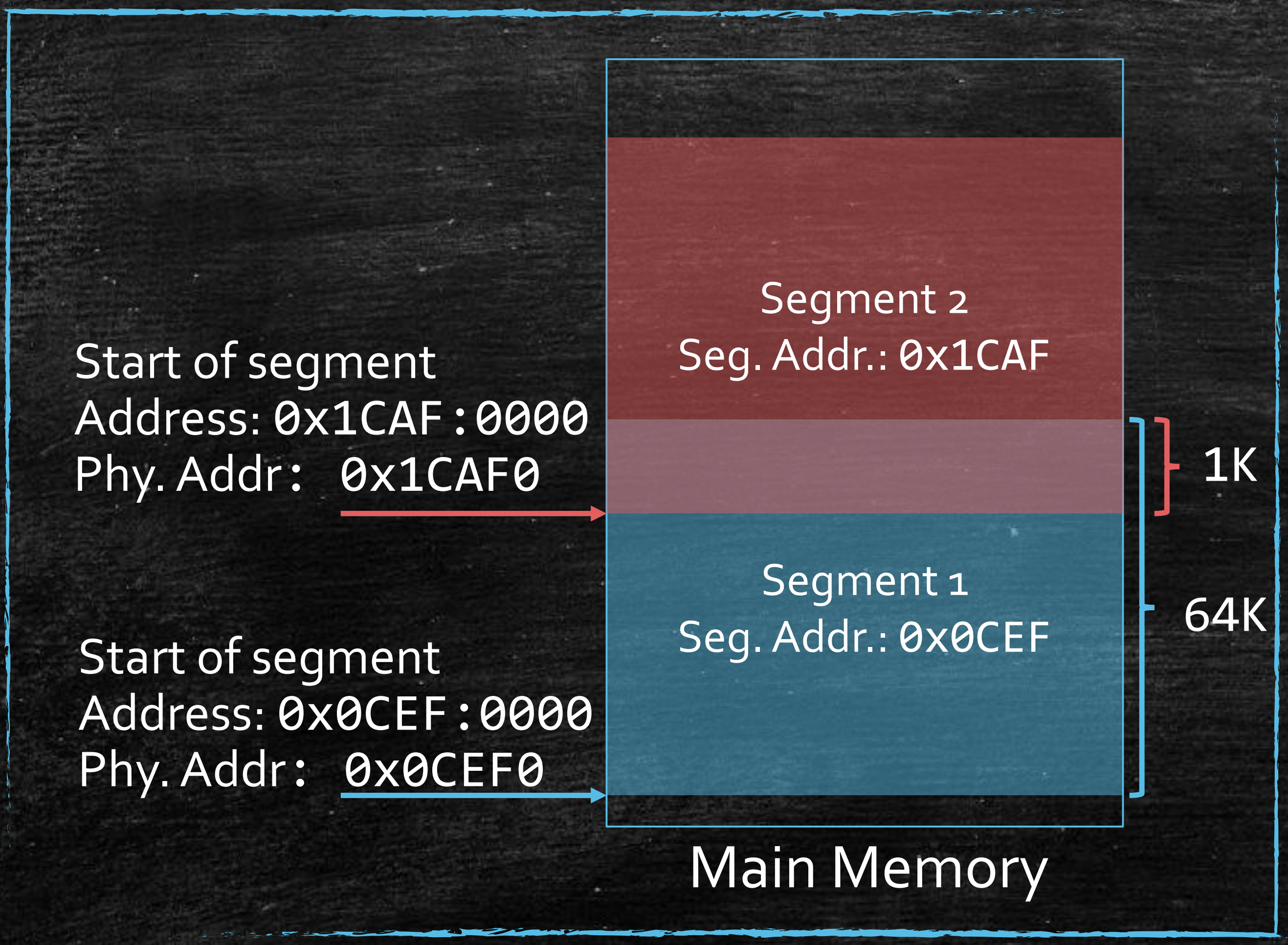
MEMOS-1: Where to begin?

- Need to know the state of the machine when MBR execution begins
  - Kind of instructions we can execute: 16-bit Real-Mode Code
  - How to address the memory: Real-Mode Segmentation
- Execution environment
  - No underlying Operating System – We're on our own! Wait! It gets even worse!
  - No library functions e.g. printf, scanf, itoa and etc.
  - Only BIOS interrupt service routines
- MBR program organization
  - Flat binary: No specific format, a mixed bag of code and data
  - The firmware just makes a jump to `0x7C00` and expects runnable code there!
  - Usually written in Assembly! Why not C? (x86 Assembly Guide)

MEMOS-1: x86 Real-Mode

- 16-bit Instructions and Registers
  - 16-bit registers and operands by default
  - AX, BX, CX, DX, SI, DI, BP, SP
  - 32-bit registers are still available: EAX, EBX,…
- 20-bit Memory Address Space (Up to 1MB)
  - Direct access to physical memory
  - Firmware, user and devices co-exist – No memory protection!
  - Need for segmentation to access beyond 64KB
    - CS, DS, SS, ES, FS, GS
    - Don't forget to set your segment registers!
  - Memory segments can overlap.

MEMOS-1: Real-Mode Memory Segmentation



Issue:
20 Address lines (2^20 = 1MB)

16-bit registers (2^16 = 64KB)

`mov $val, (%cx)`

Solution:

16-bit segment registers

PAddr = Operand + SEG << 4

`mov $val, (%fs:%cx)`

Special purpose seg. regs.:

CS, DS, SS

More info: OSDEV

MEMOS-1: What is a flat binary

- The Executable and Linkable formats (e.g. ELF):
  - Organizes Code and Data into different sections
    - .text: machine instructions
    - .data: global tables and variables
    - .rodata: constant data (e.g. string literals, const variables,…)
    - .bss: uninitialized data
  - Each section can be loaded at a different location in the memory
  - Some data sections (like .bss) don’t exist in the file but the loader inflates them in the main memory
  - Can contain information about symbols and for debugging purposes
  - The entry point (which function is the main function) for executable binaries
- Flat binary doesn’t have any meta-data: Mixed bag of code and data
  - Order: By the order of your code and the as specified in the linker script

MEMOS-1: Address translation

- What are symbols in a program?
  - From programmer's perspective: Names of variables, functions and labels
  - From machine's perspective: Absolute or relative addresses!
  - Relative like: `jmp 1b # IP -= num. of bytes to get to 1b:`
  - Absolute like: `leaw myvariable, %ax # %ax = a number but what?`
- Compiler does not replace variables with addresses
- Linker does! How does it know where our program will end up in the main memory at run-time?
- Answer:
  - The .org directive in assembly
  - Specified in the linker script

MEMOS-1: BIOS routines come to rescue

- The firmware sets up the system devices in order to execute MBR
- It also provides a basic set of device drivers and service routines
- You can access those services by issuing software interrupts
  - Pass the parameters through registers
  - Invoke a software interrupt using the INT instruction
- Find the list of standard BIOS interrupt service routines here
- Useful BIOS interrupts:
  - INT 0x10: Video services (a very basic graphics driver)
  - INT 0x15: System services (e.g. to get system parameters)

MEMOS-1: Sample MBR program (mbr.S)

```
.code16
.globl _start
_start:
#Initalize the data segment
movw $0x7C0,  %dx
movw %dx,     %ds
#Get the address of string
leaw msg,     %si
movw msg_len, %cx
#Print the greeting string
1:
lodsb #Loads DS:SI into AL
movb $0x0E,   %ah
int  $0x10
loop 1b
#Print “0x”
movb $’0’,    %al
movb $0x0E,   %ah
int  $0x10
movb $’x’,    %al
movb $0x0E,   %ah
int  $0x10
hlt
msg:
.string “MemOS: System Memory is:”
msg_len: .word . - msg
#Put the MBR Signature
.org 0x1FE
.byte 0x55
.byte 0xAA
```

MEMOS-1: Sample MBR program

Now we must compile and link our program:

- We want a flat binary file as BIOS does not support ELF or EXE
- The first instruction is located at `0x7C00` in the main memory

But how?

MEMOS-1: Introduction to linker scripts

- The linker collects compiled object files, resolves references to symbols to addresses and builds a binary executable/linkable file
- The linker script helps the linker to understand:
  - The desired binary format (OUTPUT_FORMAT)
    - Examples are: flat binary, 32-bit ELF, 64-bit ELF
  - The target system architecture which is later used by the loader software to verify if the binary is compatible with the architecute (OUTPUT_ARCH)
  - The entry point to the executable program (ENTRY): i.e. The first function that should be called by the loader to start the program
  - Sections of the output file (SECTIONS):
    - Where to put the sections in the main memory (base address) <- Addr. Translation
    - Where to get the sections from (which section of which object file)

MEMOS-1: Sample MBR program (link.ld)

```
OUTPUT_FORMAT("binary")
ENTRY(_start)

SECTIONS {
   . = 0x0;
   .section1 : { *.o (.*);}
}
```

MEMOS-1: Sample MBR program (Makefile)

```
as --32 memos-1.S -o memos-1.o

ld -m elf_i386 -T memos-1.ld memos-1.o -o memos-1

dd bs=1 if=memos-1 of=disk.img count=512

qemu-system-i386 -m 32 -hda disk.img
```

MEMOS-2: Overview

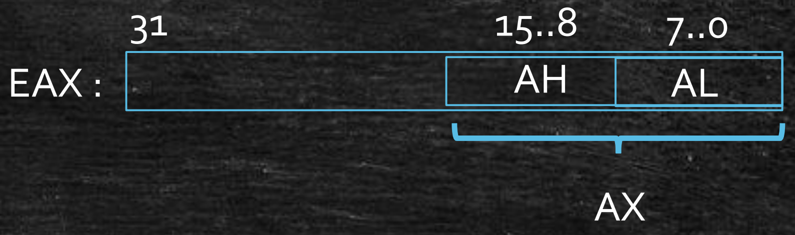
- We use GRUB's MBR instead of our MEMOS-1
- BTW, GRUB is a bootloader that:
  - Enumerates system resources (such as amount of memory available)
  - Switches the CPU to 32-bit protected mode
  - Finds your kernel executable file (ELF) from the boot media
  - This time, our kernel can be big file and is not limited to 512 bytes ☺
  - Loads it at `0x100000` (1MB) in the main memory
  - Passes the system information to your kernel
    - According to some standard format called the Multiboot Standard
  - Jumps to the memory address `0x100000`
- This time, we have to comply with GRUB, instead of BIOS!

MEMOS-2: Where to begin?

- Need to know the state of the machine when GRUB calls our code
  - 32-bit protected mode with segmentation
  - 2 flat segments of 4GB: Can run code and access data anywhere in 0x0 to 4GB
- Execution environment
  - We are in Ring-0: Most privileged level, hence, can do anything
  - No access to BIOS services – This time, we are totally alone ☹
- Program organization
  - GRUB expects an ELF binary with a multiboot header

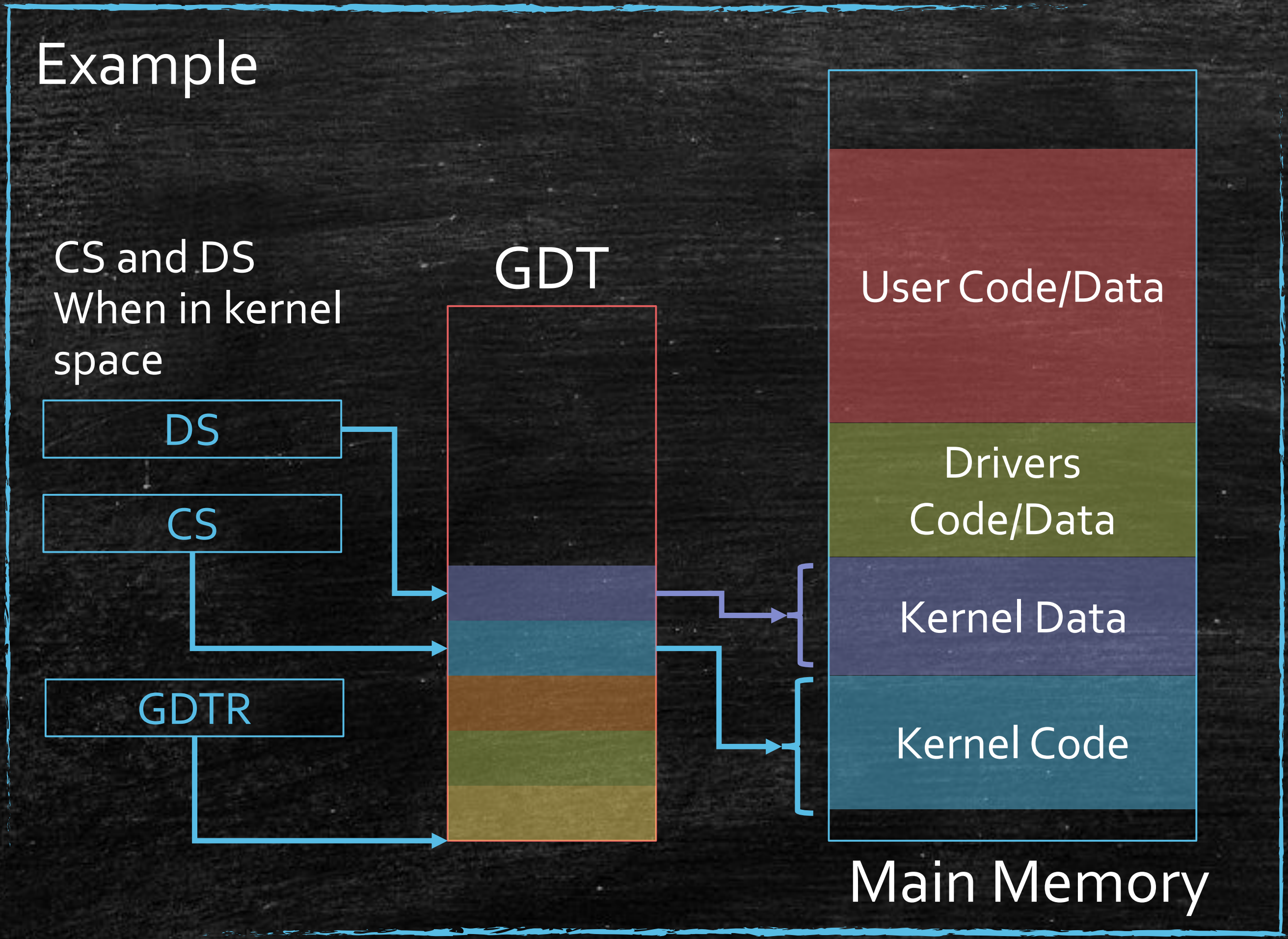
MEMOS-2: x86 Protected Mode

- 32-bit instructions and registers
  - Still can access smaller parts of a register:



- Can address up to 4GB of memory
  - Through segmentation provided by GDT and LDT
    - Here, segment registers (CS, DS, SS and etc) point to entries in GDT/LDT
  - Through virtual memory (Paging)
    - Not needed now
  - Provides memory protection by restricting types of instruction you can run in a segment of certain privilege
    - 4 Different levels: Ring-0 (most privileged) to Ring-3 (least privileged)

MEMOS-2: Memory Segmentation in PM



GDT/LDT are stored in RAM

Up to 8192 segments

Segments can overlap

Can cover up to 4GB

Each GDT entry specifies

- Base address
- Size
- Growing direction
- Access rights:
  - Ring/Privilege
  - Executable
  - Read Access
  - Write Access

MEMOS-2: What does GRUB do for US?

- Switches to Protected Mode with a 4-GB flat memory segmentation
  - Our kernel can execute, read and write code and data anywhere in the first 4-GB
  - We are in Ring-0 and can run all the privileged instructions
- Finds our kernel in the disk and loads it at 0x100000 in the mem.
- Looks at the Multiboot header, gathers the information required to run the kernel according to the header
- Runs the kernel and passes the boot information according to the Multiboot Standard

MEMOS-2: The MultiBoot standard

- Kernel must define a header early on in its binary file in order to:
  - Specify what kind of information the bootloader must pass to the kernel
  - To verify if the binary file is a valid Multiboot-compliant kernel
- Defines the desired machine state before calling the kernel
- Defines the boot information format
  - It's a data structure that GRUB fills its fields that describe the system
    - Can you find the memory map there?
  - It's address in the memory is written in %EBX before jumping to the kernel
    - So the kernel can read EBX and access the information

MEMOS-2: C, sweet C!

- To call a C function:
  - Setup the stack pointer: %esp
  - Follow the C calling convention for argument passing
  - Jump to the function using the CALL instruction
- Make sure your C code does not depend on any external or GCC built-in library
  - Compiler flags: -fno-builtin -nostdinc

MEMOS-2: Video RAM

- No BIOS INT 0x10 to deal with the graphics ☹, but:
- You can ask GRUB (in the multiboot header) to set a specific video mode
- The basic 80x25 text-based VGA buffer is mapped to `0xB8000` in the main memory
- Changing each word of that buffer directly affects what gets displayed
- Maybe get a pointer and start poking at it?
- More info [here]

MEMOS-2: Example PM program (stub.S)

```
.text

.globl _start
_start:
   jmp     real_start
   # Multiboot header – Must be in 1st page of memory for GRUB
   .align 4
   .long   0x1BADB002 # Multiboot magic number
   .long   0x00000003 # Align modules to 4KB, req. mem size
   # See 'info multiboot' for further info
   .long   0xE4524FFB # Checksum

real_start:
   #TODO: Setup a proper stack for C
   #TODO: Prepare the boot information to pass to kmain
   call    kmain
   hlt
```

MEMOS-2: Example PM program (kentry.c)

```c
#include “memos.h”
static unsigned short *videoram = (unsigned short *)0xB8000; //Base address of the VGA frame buffer
static int attrib = 0x0F; //black background, white foreground
static int csr_x = 0, csr_y = 0;
#define COLS 80

void putc(unsigned char c){
  if(c == 0x09){ // Tab (move to next multiple of 8)
    csr_x = (csr_x + 8) & ~(8 - 1);
  }else if(c == '\r'){ // CR
    csr_x = 0;
  }else if(c == '\n'){ // LF (unix-like)
    csr_x = 0; csr_y++;
  }else if(c >= ' '){ // Printable characters
    *(videoram + (csr_y * COLS + csr_x)) = c | (attrib << 8); // Put the character w/ attributes
    csr_x++;
  }
  if(csr_x >= COLS){ csr_x = 0; csr_y++;} // wrap around!
}
void puts(char *text){
  for (int i = 0; i < strlen((const char*)text); i++) // You know how to implement strlen ;)
    putc(text[i]);
}

void kmain(boot_info_t* binfo){
  puts("MemOS: Welcome *** Total Free Memory: ");
}
```

MEMOS-2 Example PM program (link.ld)

```
OUTPUT_FORMAT("elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(_start)

SECTIONS {
  . = 0x100000;
  .ksection : {
    *(.*);
    . = ALIGN(0x1000);
  }
}
```

MEMOS-2: Example PM program (Makefile)

```
as --32 stub.S -o stub.o

gcc -m32 -fno-stack-protector -fno-builtin -nostdinc
-c kentry.c -o kentry.o

ld -m elf_i386 -T link.ld stub.o kentry.o -o
memos2.elf
```

MEMOS-2: Configuring GRUB

- Install grub on your disk image
- Copy your ELF binary image to the disk image
- Configure grub (through menu.lst) to load your ELF binary as a kernel

```
title MEMOS-2
root (hd0,0)
kernel /path/to/memos2.elf
```