Lab Work 3 - Bootstrap Loader

In this module, we will explore the world of bootstrapping - the process of starting up a system. Since this topic is mostly platform dependent, we will focus on x86 systems.

You will need to go through x86 assembly programming (using NASM assembler - mind your syntax!), which is very similar to 8085 assembly (but with a lot more instructions to learn ⇒ more features!). You can get a text file on x86 instruction set here. We will also be using QEMU to emulate our hardware (so that you do not destroy everything you have on your PC). Both are already available on the provided live system. If you are using your own Linux installation, please make sure you have them installed for this lab.

System Startup (x86)

Booting a system (a more technical term would be bootstrapping):

- usually BIOS on non-volatile memory (ROM or NVRAM)
  - Memory location 0xFFFFFFF0 on x86 systems? (0xFFFF0 on 8086/8088)
  - on some modern systems, power-related stuffs may run before handing over to BIOS?
- handles/prepares low-level hardware/software interface
  - bios interrupt service (interrupt vector table): 10h (display), 13h (disks), 16h (keyboard)
  - stack pointer is 512 bytes after bootsector (stack size!)?
  - 512-bytes sector was standard disk sector size
- mostly bypassed by modern 'plug-n-play' operating systems
  - PnP OS writes their own service routine (overwrites IVT?)
- loads sector 0, cylinder 0 of selected boot drive
  - first sector (512 bytes) will be loaded to 0x7C00 (segment 0)
  - some bios used 0x07C0:0x0000 (same physical, different segment!)
  - execution transferred to that location
  - DL register holds boot drive number (e.g. 0x80 for first hard disk)
  - some BIOS will NOT transfer if INVALID MBR (e.g. no boot signature 0xAA55)
- the code in sector 0 (first sector) is usually the bootloader
  - for a partitioned disk, there should be an MBR (classic MBR? 'modern' MBR?)
  - for non-partitioned disk (e.g. floppy) simply the bootloader?
  - enables multiple kernel to be selected at run-time
- can be skipped for static single kernel usage? coreboot?
- loads the kernel specified in its configuration
- BIOS is now being replaced by (U)EFI - Read more @ wikipedia
  - but legacy BIOS startup is still supported

Read more @ wikipedia

Master Boot Record (MBR)

Some notes on MBR used in classic/legacy BIOS-based system.

- first sector of a partitioned storage
  - also known as disk boot sector
  - contains a partition table (info on how the disk is partitioned)
- classic MBR
  - 446 bytes executable code
  - 64 bytes partition entries (4 primary partitions)
  - 2 bytes boot signature (0x55 0xAA)
- modern MBR
  - 218 bytes executable code
  - 2 bytes (always 0x00?)
  - 4 bytes disk timestamp
  - 216 bytes executable code
  - 4 bytes disk signature
  - 2 bytes (always 0x00?)
  - 64 bytes partition entries (4 primary partitionss)
  - 2 bytes boot signature (0x55 0xAA)
- primary job of embedded code is to boot/load Volume Boot Record (VBR)
- superseeded by GUID partition table (GPT) - Read more @ wikipedia

Read more @ wikipedia

Volume Boot Record (VBR)

- first sector of a partition on a partitioned storage
  - also known as partition boot sector

Read more @ wikipedia

2023/08/29 13:04

Disk Layout

- partitioned storage usually starts after a pre-defined offset
  - this info can be found in MBR partition table or GPT
  - e.g. with MBR, first partition usually starts at 1MB offset (2048 sectors)
  - some OS use this storage space for various information
  - e.g. FreeBSD disklabel can be found within the first 64 sector
    - @offset 0x8000 : sequence id ⇒ 0x57 0x45 0x56 0x82

Bootloader Code Basics

This is a simple tutorial on how to develop a simple bootloader for x86 compatible machines, meant to be used with (more specifically, loaded from) a USB drive.

Using QEMU for Testing

It would be a nightmare to test a bootloader code using an actual USB drive booting an actual machine unless you have other machine than the one you are using for writing/building the code. The best way to do this is using a virtual machine and I recommend QEMU for this.

First we create a 512MB[1)] USB disk image using `qemu-img`

```
qemu-img create -f raw disk.img 512M
```

Although not necessary, most USB drives nowadays are detected HDD and thus BIOS expects an MBR (I'm not going into GPT for now). To create proper partition table in MBR with a single bootable W95 FAT32 (LBA) partition I use `fdisk`

```
fdisk disk.img
```

A 'faster' way would be to simply type

```
fdisk disk.img <<EOF
n
p
1


t
c
a
1
w
EOF
```

Notice there are two newline characters (hit ENTER twice) between '1' and 't'. The `fdisk` version I'm using writes a random (at least I think it's random) 32-bit disk signature at location 0x1b8 that is optional and can be overwritten. Use

```
hexdump -C disk.img -n 512
```

to verify 0x55, 0xAA sequence at location 0x1fe and 0x1ff respectively.

This is the part which is previously NOT needed due to the fact that floppy disks do not need partition tables (non-partitioned storage). Since the bootloader code we developed is exactly 512 bytes, it is obvious that the code will overwrite the partition information in the MBR. To avoid this we only write 446 bytes instead on the full 512 bytes, which is fine since the 0x55 0xAA sequence is already there. Due to the fact that we are writing to a disk image, we need to add `conv=notrunc` option so that the image file will NOT be truncated. (This is not a problem when the target is a device file!)

```
dd if=boot.bin of=disk.img bs=1 count=446 conv=notrunc
```

Finally... to test the USB disk image

```
qemu-system-i386 usb_drive.img
```

Helper Scripts and Makefile

To make things easier, I have written some shell scripts to help with creation of disk image, copying boot code into the image and testing the boot code on the disk image (running qemu).

Project makefile (using `nasm` assembler)

makefile

```
# make for simple bootloader

MY1BOOTLD = boot.bin

AS = nasm
ASFLAGS =

.PHONY: boot

all: boot

boot: $(MY1BOOTLD)

new: clean all

clean:
    rm -rf *.bin *.lst

%.bin: %.asm
    $(AS) $(ASFLAGS) -f bin $< -o $@

%.lst: %.asm
    $(AS) $(ASFLAGS) -f bin $< -l $@
```

Disk image creator (using qemu-img)

disk_create.sh

```
#!/bin/bash

MY1USBDRV=""
MY1IMGLEN=""
PARTLABEL=""
DO_FAT_32=""

QEMU_BIN="qemu-img"
QEMU_IMG=$(which "$QEMU_BIN" 2>/dev/null)
```

```
[ ! -x "$QEMU_IMG" ] &&
    echo "Cannot find QEMU binary '$QEMU_BIN'! Aborting!" && exit 1

# parse command parameters
while [ "$1" != "" ]; do
    case $1 in
        --image|-i)
            shift
            MY1USBDRV=$1
            ;;
        --size|-s)
            shift
            MY1IMGLEN=$1
            ;;
        --fat32)
            DO_FAT_32="YES"
            ;;
        --label)
            shift
            PARTLABEL=$1
            ;;
        -*)
            echo "Unknown option '$1'"
            exit 1
            ;;
        *)
            echo "Unknown parameter '$1'!"
            exit 1
            ;;
    esac
    shift
done

# use default settings if necessary
[ "$MY1USBDRV" == "" ] && MY1USBDRV="disk.img"
[ "$MY1IMGLEN" == "" ] && MY1IMGLEN="512M"
[ "$PARTLABEL" == "" ] && PARTLABEL="MY1TESTPART"

# create image file
echo -n "Creating disk image '${MY1USBDRV}' at size '${MY1IMGLEN}'... "
${QEMU_IMG} create -f raw ${MY1USBDRV} ${MY1IMGLEN} >/dev/null
[ $? -ne 0 ] && echo "Failed! Aborting!" && exit 1
echo "done!"

# CHS params provided are bogus values... for now?
echo -n "Creating single partition disk and flag as bootable... "
fdisk -C32 -S63 -H255 ${MY1USBDRV} >/dev/null 2>&1 <<EOF
n
p
1
```

```
t
c
a
1
w
EOF
[ $? -ne 0 ] && echo "Failed! Aborting!" && exit 1
echo "done!"

# create fat32 filesystem... if requested
if [ "$DO_FAT_32" == "YES" ] ; then
    echo -n "Formatting FAT32 partition on '${MY1USBDRV}'... "
    OPTS="conv=notrunc seek=1048576"
    MY1TEMP="disk_temp.img"
    PART_SIZE=$(($(fdisk -s ${MY1USBDRV})-1024))
    dd if=/dev/zero of=${MY1TEMP} bs=1024 count=${PART_SIZE}
2>/dev/null
    mkdosfs -F 32 -n "$PARTLABEL" ${MY1TEMP} >/dev/null
    dd if=${MY1TEMP} of=${MY1USBDRV} bs=1 count=${PART_SIZE} ${OPTS}
2>/dev/null
    rm ${MY1TEMP}
    [ $? -ne 0 ] && echo "Failed! Aborting!" && exit 1
    echo "done!"
fi
```

Boot image installer (using dd)

disk_init.sh

```
#!/bin/bash

MY1USBDEF="disk.img"
MY1USBDRV=""
MY1BOOTLD=""

# parse command parameters
while [ "$1" != "" ]; do
    case $1 in
        --image|-i)
            shift
            MY1USBDRV=$1
            ;;
        -*)
            echo "Unknown option '$1'"
            exit 1
            ;;
        *)
            [ "$MY1BOOTLD" != "" ] &&
                echo "Unknown parameter ($1)!" && exit 1
```

```
            MY1BOOTLD=$1
            ;;
    esac
    shift
done

[ "$MY1USBDRV" = "" ] && MY1USBDRV="$MY1USBDEF"
[ ! -f "$MY1USBDRV" ] &&
    echo "Cannot find disk image '$MY1USBDRV'! Aborting!" && exit 1

[ "$MY1BOOTLD" = "" ] && MY1BOOTLD="boot.bin"
[ ! -f "$MY1BOOTLD" ] &&
    echo "Cannot find boot image '$MY1BOOTLD'! Aborting!" && exit 1

DD_OPT_MBR1="bs=1 count=446 conv=notrunc"
DD_OPT_MBR2="bs=1 count=2 conv=notrunc skip=510 seek=510"

echo -n "Copying boot code '$MY1BOOTLD' to disk image '$MY1USBDRV'... "
dd if=$MY1BOOTLD of=$MY1USBDRV $DD_OPT_MBR1 2>/dev/null
[ $? -ne 0 ] && echo "Failed! Aborting!" && exit 1
dd if=$MY1BOOTLD of=$MY1USBDRV $DD_OPT_MBR2 2>/dev/null
[ $? -ne 0 ] && echo "Failed! Aborting!" && exit 1
echo "done!"
```

Help script to run qemu

disk_test.sh

```
#!/bin/bash

MY1USBDRV="$1"
[ "$MY1USBDRV" == "" ] && MY1USBDRV="disk.img"

QEMU_BIN="qemu-system-i386"
QEMU_SYS=$(which "$QEMU_BIN" 2>/dev/null)
[ ! -x "$QEMU_SYS" ] &&
    echo "Cannot find QEMU binary '$QEMU_BIN'! Aborting!" && exit 1

[ ! -f "$MY1USBDRV" ] &&
    echo "Cannot find disk image '$MY1USBDRV'! Aborting!" && exit 1

${QEMU_SYS} ${MY1USBDRV}
```

The makefile is for you to assemble (using NASM) your bootloader code. If your code is `thing.asm`, run

```
make thing.bin
```

To create a disk image (instead of running the commands above), you can simply run

```
sh disk_create.sh
```

to create a 512MB image file named `disk.img`.

To insert your bootloader code `thing.bin` into the disk image, you may run

```
sh disk_init.sh thing.bin
```

Finally, to test the image, run (duh!)

```
sh disk_test.sh
```

Simple Bootloader Code

A simple code that just hangs around...

boot.asm

```
bits 16
org 0x7c00

; hang around
hang:
    jmp hang

; filler
    times 510-($-$$) db 0

; the boot signature
    db 0x55,0xAA
```

BIOS Interrupts

The BIOS (now called legacy BIOS in [U]EFI systems) system is actually very handy to have early on the system. It allows bootloader code to access all available hardware on the system with ease. This is done via the BIOS interrupt. Listed below are the commonly used interrupt routines:

- Video:
  - INT 0x10, AH = 1 – set up the cursor
  - INT 0x10, AH = 3 – cursor position
  - INT 0x10, AH = 0xE – display char
  - INT 0x10, AH = 0xF – get video page and mode
  - INT 0x10, AH = 0x11 – set 8×8 font
  - INT 0x10, AH = 0x12 – detect EGA/VGA

    - INT 0x10, AH = 0x13 – display string
    - INT 0x10, AH = 0x1200 – Alternate print screen
    - INT 0x10, AH = 0x1201 – turn off cursor emulation
    - INT 0x10, AX = 0x4F00 – video memory size
    - INT 0x10, AX = 0x4F01 – VESA get mode information call
    - INT 0x10, AX = 0x4F02 – select VESA video modes
    - INT 0x10, AX = 0x4F0A – VESA 2.0 protected mode interface
- Disk:
    - INT 0x13, AH = 0 – reset floppy/hard disk
    - INT 0x13, AH = 2 – read floppy/hard disk in CHS mode
    - INT 0x13, AH = 3 – write floppy/hard disk in CHS mode
    - INT 0x13, AH = 0x41 – test existence of INT 13 extensions
    - INT 0x13, AH = 0x42 – read hard disk in LBA mode
    - INT 0x13, AH = 0x43 – write hard disk in LBA mode
- Memory:
    - INT 0x12 – get low memory size
    - INT 0x15, EAX = 0xE820 – get complete memory map
    - INT 0x15, AX = 0xE801 – get contiguous memory size
    - INT 0x15, AX = 0xE881 – get contiguous memory size
    - INT 0x15, AH = 0x88 – get contiguous memory size
    - INT 0x15, AH = 0xC0 – Detect MCA bus
    - INT 0x15, AX = 0x0530 – Detect APM BIOS
    - INT 0x15, AH = 0x5300 – APM detect
    - INT 0x15, AX = 0x5303 – APM connect using 32 bit
    - INT 0x15, AX = 0x5304 – APM disconnect
- Keyboard:
    - INT 0x16, AH = 0 – read keyboard scancode (blocking)
    - INT 0x16, AH = 1 – read keyboard scancode (non-blocking)
    - INT 0x16, AH = 3 – keyboard repeat rate
- Other:
    - INT 0x11 – Hardware detection

More Bootloader Codes

Print Single Char

This code prints a single letter on the screen...

boot1.asm

```
bits 16
org 0x7c00

; main code
init:
    mov al, 0x41 ; 'A'?
    mov ah, 0x0e ; function: display char (tty?)
    int 0x10 ; video display interrupt
```

```
; hang around
hang:
    jmp hang

; filler
    times 510-($-$$) db 0

; the boot signature
    db 0x55,0xAA
```

Print String

The code above uses BIOS interrupt to send a character to display. A more useful code would be to display a string instead of a single character.

boot2.asm

```
bits 16
org 0x7c00

; main code
init:
    mov bp,mesg
loop:
    mov al,[bp]
    or al,al
    jz hang
    mov ah,0x0e ; function: display char (tty?)
    int 0x10 ; video display interrupt
    inc bp
    jmp loop

; hang around
hang:
    jmp hang

; message
mesg:
    db 13,10,"Hello UniMAP!",13,10,0

; filler
    times 510-($-$$) db 0

; the boot signature
    db 0x55,0xAA
```

An even more efficient way to move an array of bytes is to use the `lodsb` instruction, which in turn

requires the use of `si` register instead of `bp`.

boot3.asm

```
bits 16
org 0x7c00

; main code
init:
    mov si,mesg
loop:
    lodsb
    or al,al
    jz hang
    mov ah,0x0e ; function: display char (tty?)
    int 0x10 ; video display interrupt
    jmp loop

; hang around
hang:
    jmp hang

; message
mesg:
    db 13,10,"Hello UniMAP! ... again!",13,10,0

; filler
    times 510-($-$$) db 0

; the boot signature
    db 0x55,0xAA
```

Get Char

What about getting an input? A keystroke from the keyboard...

boot4.asm

```
bits 16
org 0x7c00

; main code
init:
    mov si,mesg
    call puts
    ; read keyboard, once we get one print something else
    mov ah,0 ; function: read scancode (blocking)
    int 0x16 ; keyboard interrupt
```

```
    ; AH = BIOS scan code ; AL = ASCII character
    ; for non-blocking version (ah=1)
    ; ZF set if no keystroke available ; ZF clear if keystroke
available
    mov si,more
    call puts

; hang around
hang:
    jmp hang

; message
mesg:
    db 13,10,"Hello UniMAP! ... again!",13,10,0
more:
    db 13,10,"Got you!",13,10,0

; sub-routine to print string pointed by si
puts:
    lodsb
    or al,al
    jz .done
    mov ah,0x0e ; function: display char (tty?)
    int 0x10 ; video display interrupt
    jmp puts
.done:
    ret

; filler
    times 510-($-$$) db 0

; the boot signature
    db 0x55,0xAA
```

Notice that the code to print a string has been re-organized as a sub-routine (portion of code that can be reuse, called from other parts of the program).

Get String

What if we need to read a string? (**Hint**: need to detect a space or the <ENTER> key)

boot5.asm

```
bits 16
org 0x7c00

; main code
init:
```

```
    mov si,mesg
    call puts
    mov di,save
    call gets
    mov si,more
    call puts
    mov si,save
    call puts
    mov si,next
    call puts

; hang around
hang:
    jmp hang

; message
mesg:
    db 13,10,"Enter your name: ",0
more:
    db 13,10,"Hello, ",0
next:
    db "!",13,10,0
save:
    times 20 db 0

; sub-routine to print string pointed by si
puts:
    lodsb
    or al,al
    jz .done
    mov ah,0x0e ; function: display char (tty?)
    int 0x10 ; video display interrupt
    jmp puts
.done:
    ret

; sub-routine to get a string and save to mem pointer by di
gets:
    mov ah,0 ; function: read scancode (blocking)
    int 0x16 ; keyboard interrupt
    cmp al,13
    jz .done
    stosb
    mov ah,0x0e ; function: display char (tty?)
    int 0x10 ; video display interrupt
    jmp gets
.done:
    xor al,al
    stosb
    ret
```

```
; filler
    times 510-($-$$) db 0

; the boot signature
    db 0x55,0xAA
```

Note that both gets and puts subroutines utilizes `di` and `si` registers (and therefore, need proper assignments befire using them) respectively.

Some useful subroutines

Figure out what these does and see if you need them 😀

```
str2ui:
    xor dx,dx
    mov bx,10
.more:
    lodsb
    or al,al
    jz .done
    sub al,0x30
    xor cx,cx
    mov cl,al
    mov ax,dx
    mul bx ; if bl, only al is multiplied
    or dx,dx
    jnz .done ; abort if 16-bit value overflows
    mov dx,ax
    add dx,cx
    jmp .more
.done:
    mov ax,dx
    ret
```

```
ui2str:
    mov bp,di
    mov si,di
    mov cx,ax
    mov bl,10
.more:
    xor ax,ax
    mov al,ch
    div bl
    mov dh,al ; save quotient
    mov ch,ah ; save remainder
    mov ax,cx
    div bl
```

```
    mov dl,al ; save quotient
    mov al,ah ; send out remainder
    add al,0x30
    mov [bp],al
    inc bp
    mov cx,dx
    or cx,cx
    jnz .more
    mov [bp],cl ; null
.revs:
    dec bp
    cmp bp,si
    jc .done
    jz .done
    mov ah,[bp]
    lodsb
    mov [bp],al
    mov al,ah
    stosb
    jmp .revs
.done:
    ret
```

Things to Tinker

Thing1 Write a boot code that display the number in ax (16-bits).

Thing2 Write a boot code that gets 2 single digit number, adds them up and display the result back on screen.

Thing3 (CHALLENGE!) Write a boot code for a simple calculator - can you fit them into 510 byte frame?

Thing4 (CHALLENGE!) Write a boot code that lists the files in the root path of a disk partition.

1)

This is big enough to create FAT32 partition, but still not TOO big to be annoying 😛