INF-2201: OP fundamentals
Project 1
Lise Ekhorn Johansen
UiT id: ljo177@uit.no
GitHub user: Lise-johansen
GitHub Classroom: project-1-Lise-johansen
30.01.2023

1. Introduction

Write the bootup code for a simple operation system (OP) and create an image file. The bootloader is written in AT&T 16-bit assembly real mode and the creatimage program is written in c.
The intended result is a bootloader that resides in the boot sector, bootloader is to load the OS image from one sector to a place in PC memory. The creatimage tool creates a bootable OP image that will reside on a USB disk.

2. Design and Implementation

This project was done in two parts, the first part was the bootblock. The bootblock entails the bootloader code and transferring the kernel to the correct location in memory. And creatimage creates the IO image.

2.1 Design and implementation of bootblock

Figure 2.1 is an illustration of the design of the bootloader. The idea behind the illustration is to set up the data segment, set up the stack segment, and the stack for the bootblock. Copy kernel to 0x01000 and set up data segment for the kernel. And lastly, a long jump to the kernel, to give control back to the kernel.

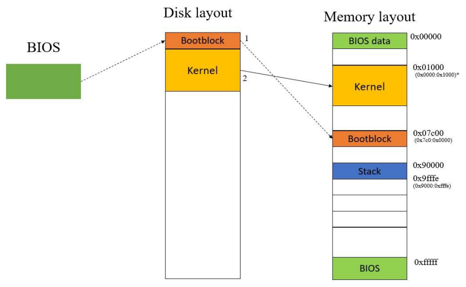


Figure 2.1: Illustration for the implantation of the boot loader. 1 is done by the BIOS code. The bootblock code does 2. The bootloader code is inside the bootblock, and the bootblock resides in a boot sector.

The first step of the implementation is to set the stack a place in memory. Important to place the stack segment and stack pointer in memory locations from 0x0700e to 0x9fc00. The next step is as followed move the stack segment to the stack segment register, and place the stack pointer in the stack pointer register. The boot segment is also moved to its data segment register. And lastly, the kernel segment is moved to “es” register, and the kernel offset is moved into the “bx” register. All segment (boot, kernel, … etc) has to be moved to a general register before being moved to a segment register. To move the kernel into memory BIOS interrupt call 13, the 20th interrupt, or just INT 13th is used. Read from the drive using function 2, and choose the number of sectors that are to be read in, cylinder number, sector number, and head number, and call INT 13 to call BIOS. Lastly, long jump gives control to the kernel.

2.2 Design and implementation of creatimage

Figure 2.2 shows the illustration of the design behind the creatimage implementation. The design is simplified in these steps:

- Create the image, this image is to become the OS image
- Open the file and read the file. In this case, there are two files. The first is the bootblock and the second is the kernel.
- Get the code and data segment. Have to go through the files and extract the segments.
- A magical signature (0x55aa), this signature represents the end of the bootblock. Are on the 511th and 512th location of a 512 boot segment.
- Padding, write in zeroes to get the correct file size.
- Kernel size is located in byte 2 in the image file. Kernel size is determined by how many segments the file has.

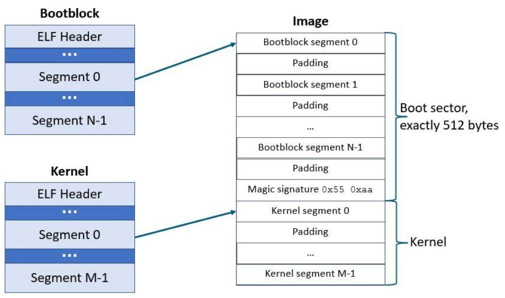
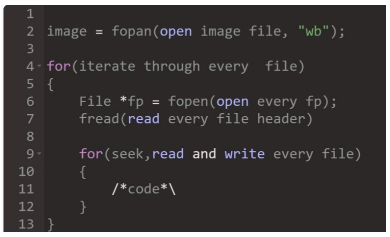


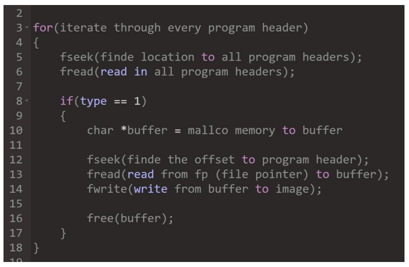
Figure 2.2: After the code is compiled Bootblock and kernel are separated by the pre-code. Have to iterate through the bootblock and kernel and extract the needed segments and information (offset, file size, …, etc), and write it to the image file.

The easiest way to explain the implementation of the creatimage code is through pseudo-code. The implementation starts out by creating an image file by using the function fopen. In a for loop, every file is opened the first elf header is read, as picture 2.3 shows. In a nested loop, every program header is iterated through, and by using fseek the offset for the program headers is found. Now the process of finding and extracting the code and data segments starts. Picture 2.4 describes this process.



```

image = fopan(open image file, "wb");

for(iterate through every  file)
{
    File *fp = fopen(open every fp);
    fread(read every file header)

    for(seek,read and write every file)
    {
        /*code*\
    }
}
```

Picture 2.3: the pseudo-code shows the process of making an image file and opening and reading a number of files.



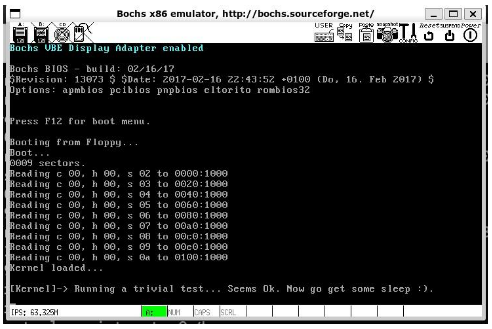
```

for(iterate through every program header)
{
    fseek(finde location to all program headers);
    fread(read in all program headers);

    if(type == 1)
    {
        char *buffer = mallco memory to buffer

        fseek(finde the offset to program header);
        fread(read from fp (file pointer) to buffer);
        fwrite(write from buffer to image);

        free(buffer);
    }
}
```

Picture 2.4: This pseudo-code shows how they read and extract the data and code segment from every file.

Implementation of the magical signature is done by making an array allocate enough place for its two elements, 0x55 and 0xaa. Use function fseek to seek to the end of a segment (one segment has a size of 510 +2 bytes). And then call fwrite to write the array (0x55 and 0xaa) inside of the image. The padding function is almost identical to the magical signature function, but inside of writing an array to image, a short int is written to image. The last function is kernel size. Where kernel size shall reside in image is on byte two. So the first step is to fseek to byte two, then use fwrite to write the kernel size to image.

3. Experiments & Result

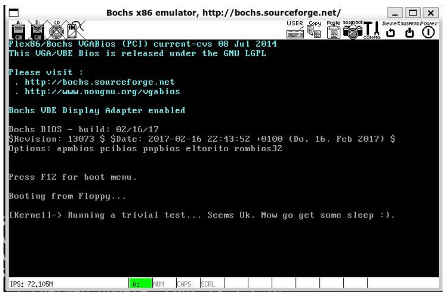
To test if the implementation is working, Boch was used to see if the code works on a simulation of a machine without an OS. Figure 3.1 shows the expected result from the simulation. The final test is to try to boot the code from a real PC without an OS, and if the message “Running a trivial test … Seems OK. Now get some sleep :)” is displayed on the PC the code 100% works.



```
Bochs x86 emulator, http://bochs.sourceforge.net/

Bochs VBE Display Adapter enabled

Bochs BIOS - build: 02/16/17
$Revision: 13073 $ $Date: 2017-02-16 22:43:52 +0100 (Do, 16. Feb 2017) $
Options: apmbios pcibios pnpbios eltorito rombios32


Press F12 for boot menu.

Booting from Floppy...
Boot...
0009 sectors.
Reading c 00, h 00, s 02 to 0000:1000
Reading c 00, h 00, s 03 to 0020:1000
Reading c 00, h 00, s 04 to 0040:1000
Reading c 00, h 00, s 05 to 0060:1000
Reading c 00, h 00, s 06 to 0080:1000
Reading c 00, h 00, s 07 to 00a0:1000
Reading c 00, h 00, s 08 to 00c0:1000
Reading c 00, h 00, s 09 to 00e0:1000
Reading c 00, h 00, s 0a to 0100:1000
Kernel loaded...

[Kernel]-> Running a trivial test... Seems Ok. Now go get some sleep :).

IPS: 63.325M      A:  NUM  CAPS  SCRL
```

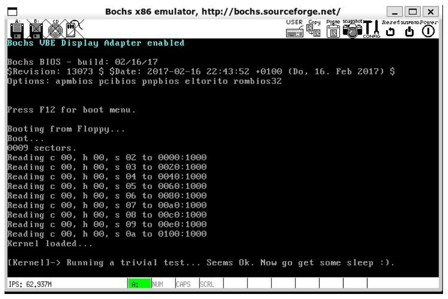
Picture 3.1: By compiling bootblock.given and creatimage.given the correct result is given and shown in boch. The message “Running a trivial test… Seems OK. Now get some sleep :)” is a sign that everything should be working.

Picture 3.2 shows the result when compiling the creatimage.given and bootblock. And the result matches the result in picture 3.1. The result from picture 3.3, when compiling bootblock.given and creatimage also matches the expected result in picture 3.1.



```
Bochs x86 emulator, http://bochs.sourceforge.net/

Plex86/Bochs VGABios (PCI) current-cvs 08 Jul 2014
This VGA/VBE Bios is released under the GNU LGPL

Please visit :
 . http://bochs.sourceforge.net
 . http://www.nongnu.org/vgabios

Bochs VBE Display Adapter enabled

Bochs BIOS - build: 02/16/17
$Revision: 13073 $ $Date: 2017-02-16 22:43:52 +0100 (Do, 16. Feb 2017) $
Options: apmbios pcibios pnpbios eltorito rombios32


Press F12 for boot menu.

Booting from Floppy...

[Kernel]-> Running a trivial test... Seems Ok. Now go get some sleep :).

IPS: 72.105M      A:  NUM  CAPS  SCRL
```

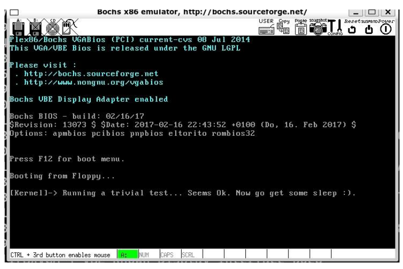
Picture 3.2: When compiling bootblock with the creatimage.given matches the result from picture, since the message “Running a trivial test… Seems OK. Now get some sleep :)” shows.



```
Bochs x86 emulator, http://bochs.sourceforge.net/

Bochs VBE Display Adapter enabled

Bochs BIOS - build: 02/16/17
$Revision: 13073 $ $Date: 2017-02-16 22:43:52 +0100 (Do, 16. Feb 2017) $
Options: apmbios pcibios pnpbios eltorito rombios32


Press F12 for boot menu.

Booting from Floppy...
Boot...
0009 sectors.
Reading c 00, h 00, s 02 to 0000:1000
Reading c 00, h 00, s 03 to 0020:1000
Reading c 00, h 00, s 04 to 0040:1000
Reading c 00, h 00, s 05 to 0060:1000
Reading c 00, h 00, s 06 to 0080:1000
Reading c 00, h 00, s 07 to 00a0:1000
Reading c 00, h 00, s 08 to 00c0:1000
Reading c 00, h 00, s 09 to 00e0:1000
Reading c 00, h 00, s 0a to 0100:1000
Kernel loaded...

[Kernel]-> Running a trivial test... Seems Ok. Now go get some sleep :).

IPS: 62.937M      A:  NUM  CAPS  SCRL
```

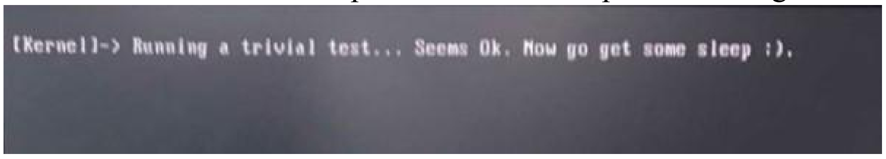
Picture 3.3: Compiled creatimage and bootblock.given and boch shows the expected result.

The next step is the compile bootblock and creatimage, picture 3.4 shows the result. The result is the message “Running a trivial test … Seems OK. Now get some sleep :)”.



```
Bochs x86 emulator, http://bochs.sourceforge.net/

Plex86/Bochs VGABios (PCI) current-cvs 08 Jul 2014
This VGA/VBE Bios is released under the GNU LGPL

Please visit :
 . http://bochs.sourceforge.net
 . http://www.nongnu.org/vgabios

Bochs VBE Display Adapter enabled

Bochs BIOS - build: 02/16/17
$Revision: 13073 $ $Date: 2017-02-16 22:43:52 +0100 (Do, 16. Feb 2017) $
Options: apmbios pcibios pnpbios eltorito rombios32


Press F12 for boot menu.

Booting from Floppy...

[Kernel]-> Running a trivial test... Seems Ok. Now go get some sleep :).

CTRL + 3rd button enables mouse      A:  NUM  CAPS  SCRL
```

Last test to see if the code works is to copy the OS image to a USB stick and try to boot a PC without an OS. The result shown in picture 3.5 is the expected message.



```
[Kernel]-> Running a trivial test... Seems Ok. Now go get some sleep :).
```

Picture 3.5: Last test

4. Discussion

4.1 Experiment result

Based on the results of the experiment, a working boot-up code was made. One non-important difference was that the given bootblock as shown in picture 3.2 has nine print statements. This statement is only cosmetic and posts non-real functional differences. It is believed that this statement comes from some well-placed print and jump statements in the bootblock.

4.2 Drawbacks of the implementation

This implementation of creatimage is dynamic in all places except one, the kernel size. Kernel size is hard coded into 9 sectors. But still, as per the project requirement, the code can handle over 18 sections. Tested up to 24 sections and the code still ran as required. So will say this is a non-issue when it comes to fulfilling the requirements of funning 18 sections. The last drawback is in the padding. For the padding to be perfect, it has to use the kernel size to get the number of bytes to pad. One way to get the kernel size is as followed:

1. Find the kernel size: kernel size mod 512 = rest
2. Padd the rest: padding = fwrite(image, rest, 1, fp);

As it is now padding happens to a wished size that is manually written in the padding function. The wished size is set to the file size. One known downside is that the bootloader never reeds kernel size from the image. So to change the number of sectors that are being read, the reader has to change the number of sections in bootblock and creatimage.

5. Conclusion

The intended result is a bootloader placed inside the boot sector, where the bootloader loads the OS image to a USB stick was achieved. And all experiments that were done on the code support that statement. The only deviation from the "perfect result" is that to change the number of sectors the kernel reads, you have to manually write it in bootblock and creatimage.

Bibliography

[1] (Wikipedia), “INT 13H”, Wikipedia, Last updated: 23.01.2023, Read:14.01.2023
URL: INT 13H - Wikipedia

[2] (wiki.osdev.org), “ELF”, Wiki.osdev.org, Last updated: 04.12.2022, Read: 19.01.2023
URL: ELF - OSDev Wiki