INF 2201

Assignment 1, Bootup Mechanism

Magnus Dahl-Hansen

January 2022

Introduction:

This report describes how a bootblock code and a utility for creating a bootable Image for a simple operating system was implemented. The bootable image is an executable file containing a bootblock and a kernel among other things. The bootblock code is responsible for loading the operating system image from a disk/drive into RAM and transfer control to the kernel.

Design&Implementation:

The continuation of the work done in the pre-code required that a stack and the boot segment is set up for the bootblock. One requirement for the stack is that it must be placed behind the bootblock code so that it is not overwritten. Just for the sake of making things easy all the space between memory address 0x7e01 – 0x9fbff is given to the stack even though this much memory space may not be required. Since the stack grows from a high address to lower addresses in memory its segment is located at 0x9fbff and it may grow up to where the bootblock code ends. Furthermore the boot segment variable which points to a place in memory where the bootblock code is stored, is transferred into the the Data Segment register (%ds) for further use.

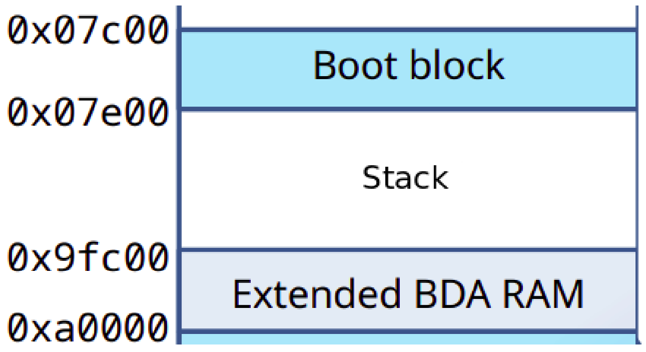

Figure 1. Illustration of the bootblock and stack location in RAM.

Next step is to copy the kernel from the bootable device into RAM. To do so an Interrupt13 (int0x13) call is necessary to interrupt the BIOS software. There are many interrupt calls that can be made but in this case the interrupt number 13 is used to provide read and write services using CHS-addressing. For the int0x13 to perform its tasks correctly some registers must be loaded with predefined values. The %ah register must contain the value 2 to tell the int0x13 call that it should "read sectors from a drive" and the %al register must contain the numbers of sectors that will be read. The number of sectors are fetched via a "kernelsize" variable and is calculated later in the create image utility. After this, the registers %ch, %cl and %dh must contain values that can locate the kernel with the help of CHS-addressing. In this case the kernel is located at Cylinder number 0, Head number 0 and Sector number 2. Furthermore the %dl register contains 0, which is the drive number, and the %es and %bx registers contain the kernel segment and the kernel offset. After the int0x13 call has completed reading the sectors, the kernel segment is moved into the %ds register which describes that the kernel should be loaded at address 0x1000 in RAM. At last the control is transferred to the kernel via a long jump to its location.

Moving over to the implementation of the create image utility. Assuming that the first provided file is a bootblock file, it is sent to a function that will parse it and extract executable code. This is done by using the provided Elf32 structures to locate the Elf-header and Program-header of a file. By locating the Elf-header the offset of where the program header can be found. Furthermore the offset of the executable code can be located via the Program-header. From here the executable code is read and put into an array of 512 bytes allocated by the "Calloc( )" function. The Calloc( ) function is used because it will automatically set allocated memory to zero, which will help padding the image file correctly. By using an array as a buffer to transport the executable code to the image file, the magic numbers and kernel size can easily be written to the code. The magic numbers "0x55" and "0xaa" are written to position 510 and 511 in the array and the kernel size is written to position 2(positions are predefined). When the modifications of the array has been made, the entire array is written to the first sector of an opened image file.

The process of parsing the the kernel file is similar to the parsing of the bootblock. Again, assuming the kernel file is given as the second argument of the create image function, the file is sent to its unique function. From here the executable code is fetched via the Program-header and the Elf-header. The code is placed inside an allocated array using "calloc( )" to make the padding correctly. The main difference in this function compared to the bootblock is that the kernel does not need to contain any magic numbers or kernel size and the size of the kernel code may be much bigger than the bootblock. The size of the kernel can be found by reading the "p_filsz" variable from the Program-header. This size is divided by 512 and then rounded up to find how many sectors the kernel occupies, which then again is used as the kernel size in the bootblock code. When the kernel executable code is being written to the image, the mode "append byte" is used as part of the "fopen( )" function. This will assure that the kernel code is written after the bootblock code in the image, rather than overwriting it. If any additional file are provided to the create image utility they will be sent through a function that will parse these files in a similar way as the kernel parsing function. The extracted executable code is then appended to the image file, behind the bootblock and kernel code.

Results&Discussion:

My code compiles and runs successfully. Using my implementation of the bootblock and the create image utility, the Bochs emulator will print the “Running a trivial test… Seems ok. Now go get some sleep” message successfully. However, compared to the design of the given bootblock code, my implementation will not print which sectors are read.

Using my own code I tried to run a multi segment kernel on the Bochs emulator but the results were not as expected. The code compiles without any problems but when it is run the “ trivial test” message is flickering. I am uncertain of why this is the case since the code has been written to the image file and the padding seems correct according to the built-in hex editor in Visual Studio Code.

Conclusion:

This assignments task was to create a bootblock code and a bootable image for a simple operating system. The assignment is solved by implementing a code that loads and transfers control to a kernel and another that extracts code from provided bootblock and kernel executable files and writes them to an operating system image. I can conclude that the implemented code runs successfully with a single bootblock and kernel file since Bochs emulator prints the final message, however I am unsure if the implementation is able to support a multi segment kernel.