

The Little Book about OS Development

By: David Wang, Jade Fenton, Yoon Young Jo, Anthony Khalil

ICSI 400: Lab Report

Professor: Carol Cusano

Introduction

For this semester, we worked on developing a simple operating system on a virtual machine. Initially, we created a simple operating system that would basically move a specified word into a register that we would be able to view and confirm that the simple operating system was working. Then later, we modified the files to allow more functions of the operating system, like displaying a character on the console and configuring a serial port. The operating system was run through a Bochs emulator that would allow us to view through a console and debug the simple operating system. Throughout the process of developing the operating system, we came to understand the concept of operating systems. The team members who worked on this project consisted of David Wang, Jade Fenton, Anthony Khalil, and Diane Jo. The members who worked directly with the operating system on the virtual machine was David Wang and Jade Fenton. One of the computers used was a MacBook Pro with a macOS High Sierra version 10.13.4, and a 2.8 GHz Intel Core i5 Processor. The other computer used was a ASUS PC with Windows 7 Home Premium, and a 2.50GHz Intel Core i5 Processor. The virtual machine used on both machines was Oracle VM VirtualBox, and that the base operating system used was Ubuntu version 16.04. During the development of the operating system, both Anthony Khalil and Diane Jo have been looking up information regarding to operating systems. The lab report was written with the contribution of all the team members in the group by dividing up the sections of the lab.

Section 2: First Steps

The objective of this section is to setup the environment and settings for the development of the OS. Also, in this section, we will be implementing a simple OS, where we would change the value of the *eax* register to a predetermined value.

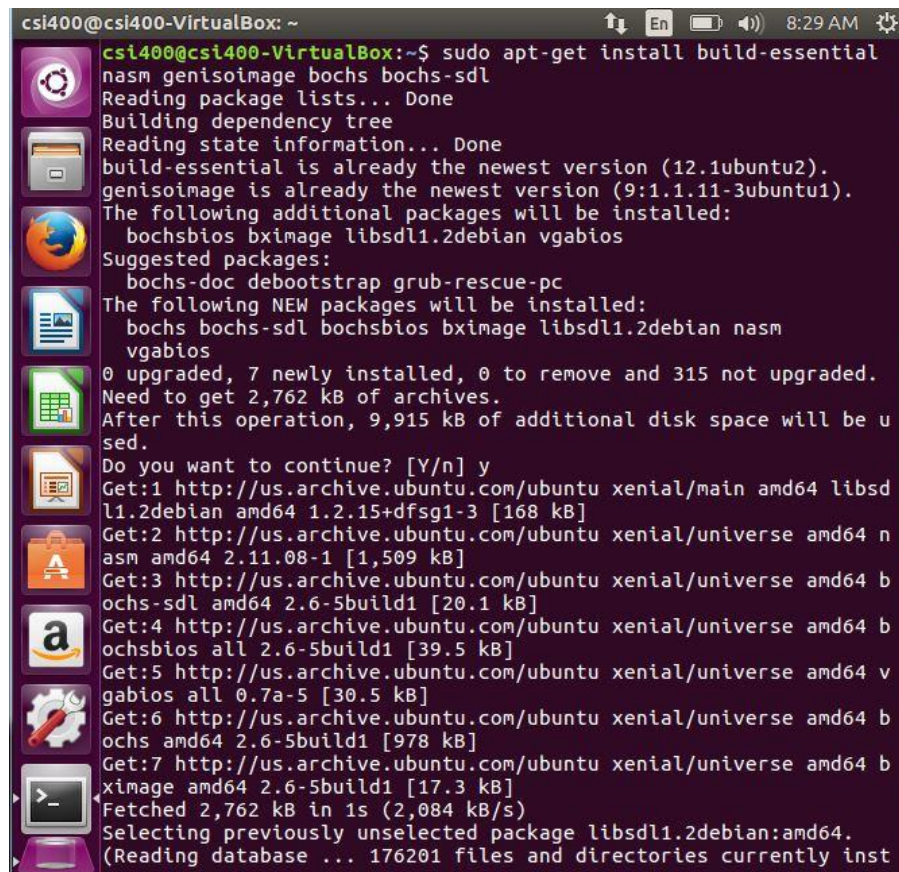
2.1 Tools

2.1.1 Quick Setup

The development of an OS was produced within a virtual machine environment with an installed Ubuntu 16.04. By using the following command:

`sudo apt-get install build-essential nasm genisoimage bochs bochs-sdl`

the required packages were downloaded for the development of the OS using *apt-get*. The outcome of the call of the command is displayed in (Figure 1).



```
csi400@csi400-VirtualBox: ~  
csi400@csi400-VirtualBox:~$ sudo apt-get install build-essential  
nasm genisoimage bochs bochs-sdl  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
build-essential is already the newest version (12.1ubuntu2).  
genisoimage is already the newest version (9:1.1.11-3ubuntu1).  
The following additional packages will be installed:  
  bochs-bios bximage libsd1.2debian vgabios  
Suggested packages:  
  bochs-doc debootstrap grub-rescue-pc  
The following NEW packages will be installed:  
  bochs bochs-sdl bochs-bios bximage libsd1.2debian nasm  
  vgabios  
0 upgraded, 7 newly installed, 0 to remove and 315 not upgraded.  
Need to get 2,762 kB of archives.  
After this operation, 9,915 kB of additional disk space will be u  
sed.  
Do you want to continue? [Y/n] y  
Get:1 http://us.archive.ubuntu.com/ubuntu xenial/main amd64 libsd  
l1.2debian amd64 1.2.15+dfsg1-3 [168 kB]  
Get:2 http://us.archive.ubuntu.com/ubuntu xenial/universe amd64 n  
asm amd64 2.11.08-1 [1,509 kB]  
Get:3 http://us.archive.ubuntu.com/ubuntu xenial/universe amd64 b  
ochs-sdl amd64 2.6-5build1 [20.1 kB]  
Get:4 http://us.archive.ubuntu.com/ubuntu xenial/universe amd64 b  
ochs-bios all 2.6-5build1 [39.5 kB]  
Get:5 http://us.archive.ubuntu.com/ubuntu xenial/universe amd64 v  
gabios all 0.7a-5 [30.5 kB]  
Get:6 http://us.archive.ubuntu.com/ubuntu xenial/universe amd64 b  
ochs amd64 2.6-5build1 [978 kB]  
Get:7 http://us.archive.ubuntu.com/ubuntu xenial/universe amd64 b  
ximage amd64 2.6-5build1 [17.3 kB]  
Fetched 2,762 kB in 1s (2,084 kB/s)  
Selecting previously unselected package libsd1.2debian:amd64.  
(Reading database ... 176201 files and directories currently inst
```

Figure 1: The figure displays the running of the apt-get command.

2.1.2 Programming Languages

The programming language that will be used to develop the OS is the C programming language with the GCC compiler. The code used in the development of an OS has a specific type of attribute that is only for GCC, which is the following:

`__attribute__((packed))` .

The specified attribute allows the compiler to use a struct that is defined by the code. The assembler NASM is used to assemble the assembly code used for the OS development. The scripting language used in the OS development was bash.

2.1.3 Host Operating System

The Host Operating System used is Ubuntu version 16.04, which is a UNIX like operating system.

2.1.4 Build System

The command Make will be used to compile the files referenced in the Makefile.

2.1.5 Virtual Machine

The code for the OS development is run within a virtual machine called Oracle VM VirtualBox. The use of a terminal emulator window was used, called Bochs, which is useful based on the debugging features. By using a virtual machine, one can demonstrate and simulate the OS environment, almost like how it is run through a physical computer.

2.2: Booting

The boot process of an operating system consists of multiple programs run in a specific order. The first program to run is the BIOS. The second program to run is GRUB1, then GRUB2, and finally the OS.

2.2.1 BIOS

The BIOS stands for Basic Input/Output System, where it is a boot firmware program that is run on a computer. The BIOS starts and controls once the computer is started up until the OS starts up and takes control of the computer. The BIOS initially runs a Power-On Self Test, which checks the hardware and see if all the attachments are working and present in the computer. Finally, the BIOS loads the operating system in the computer's RAM.

2.2.2 The Bootloader

The BIOS will transfer the control of the computer to the bootloader. The bootloader usually consists of multiple parts that is run before the boot of the operating system. The bootloader that will be used for the OS development is the GRUB, which stands for GNU Grand Unified Bootloader. For the OS development, by using the GRUB, one can create an ELF executable which contains the operating system. The GRUB would load the ELF executable into the appropriate memory location.

2.2.3 The Operating System

The GRUB would transfer the control of the computer to the operating system but before the

GRUB would initially confirm that the memory is where the operating system is located at before transferring the control over. Once the controls are transferred to the operating system, the operating system would have total control over the computer.

2.3: Hello Cafebabe

The objective of this section is to create a small OS that would write *0xCAFEBAFE* to the *eax* register.

2.3.1 Compiling the Operating System

The *loader.s* file contains the code displayed in (Figure 2).

```
global loader                ; the entry symbol for ELF

MAGIC_NUMBER equ 0x1BADB002  ; define the magic number constant
FLAGS        equ 0x0         ; multiboot flags
CHECKSUM      equ -MAGIC_NUMBER ; calculate the checksum
                                ; (magic number + checksum + flags should equal 0)

section .text:               ; start of the text (code) section
align 4                     ; the code must be 4 byte aligned
    dd MAGIC_NUMBER         ; write the magic number to the machine code,
    dd FLAGS                ; the flags,
    dd CHECKSUM              ; and the checksum

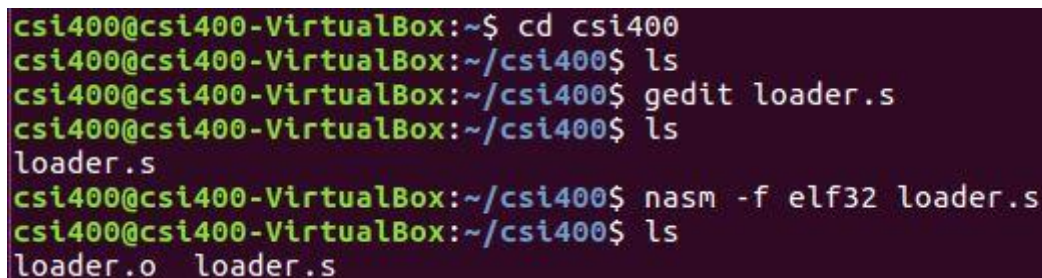
loader:                      ; the loader label (defined as entry point in linker script)
    mov eax, 0xCAFEBAFE     ; place the number 0xCAFEBAFE in the register eax
.loop:
    jmp .loop               ; loop forever
```

Figure 2: Code within the *loader.s* file

The purpose of the code stored within the *loader.s* file is to place the value *0xCAFEBAFE* within the *eax* register. To compile the *loader.s* file, the following command would be required:

nasm -f elf32 loader.s .

We came across an issue when compiling the *loader.s* file but later found out that the command used to compile the assembly file contained *elf32*, not *e1f32*. The outcome of compiling the *loader.s* file into a 32 bit ELF file is shown in (Figure 3).



```
csi400@csi400-VirtualBox:~$ cd csi400
csi400@csi400-VirtualBox:~/csi400$ ls
csi400@csi400-VirtualBox:~/csi400$ gedit loader.s
csi400@csi400-VirtualBox:~/csi400$ ls
loader.s
csi400@csi400-VirtualBox:~/csi400$ nasm -f elf32 loader.s
csi400@csi400-VirtualBox:~/csi400$ ls
loader.o  loader.s
```

Figure 3: The compiling of the *loader.s* file.

2.3.2 Linking the Kernel

The file *link.ld* contains the linker script, displayed in (Figure 4).

```
ENTRY(loader)                /* the name of the entry label */

SECTIONS {
    . = 0x00100000;           /* the code should be loaded at 1 MB */
    .text ALIGN (0x1000) :    /* align at 4 KB */
    {
        *(.text)              /* all text sections from all files */
    }

    .rodata ALIGN (0x1000) :  /* align at 4 KB */
    {
        *(.rodata*)           /* all read-only data sections from all files */
    }

    .data ALIGN (0x1000) :    /* align at 4 KB */
    {
        *(.data)              /* all data sections from all files */
    }

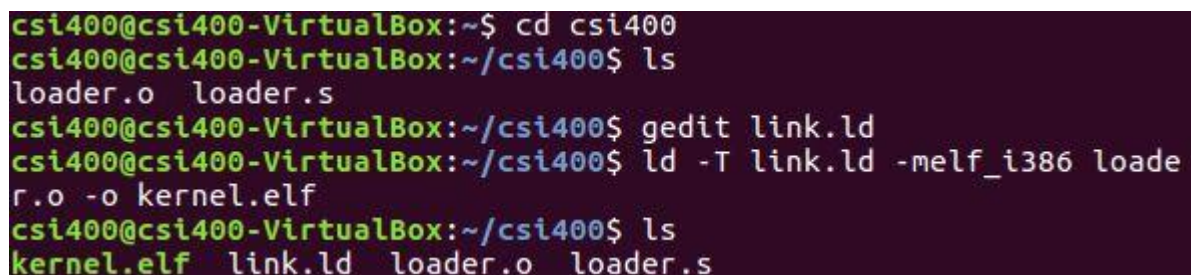
    .bss ALIGN (0x1000) :     /* align at 4 KB */
    {
        *(COMMON)             /* all COMMON sections from all files */
        *(.bss)               /* all bss sections from all files */
    }
}
```

Figure 4: The linker script within the *link.ld* file.

The *link.ld* file makes it possible to create an executable file by calling the following command:

ld -T link.ld -melf_i386 loader.o -o kernel.elf.

The command creates a link to the executable file called *kernel.elf*. The outcome of running the linking command is shown in (Figure 5).



```
csi400@csi400-VirtualBox:~$ cd csi400
csi400@csi400-VirtualBox:~/csi400$ ls
loader.o loader.s
csi400@csi400-VirtualBox:~/csi400$ gedit link.ld
csi400@csi400-VirtualBox:~/csi400$ ld -T link.ld -melf_i386 loader.o -o kernel.elf
csi400@csi400-VirtualBox:~/csi400$ ls
kernel.elf link.ld loader.o loader.s
```

Figure 5: Display the outcome of the linking command.

2.3.3 Obtaining GRUB

The GRUB used for the development of the OS is called GRUB Legacy *stage2_eltorito* bootloader. The GRUB is created within the system by downloading the file from <ftp://alpha.gnu.org/gnu/grub/grub-0.97.tar.gz> into the virtual machine. Then the *stage2_eltorito* file is copied from the download folder into the folder containing the files,

loader.s and *link.ld*. The copying process of the *stage2_eltorito* file is shown in (Figure 6).

```
csi400@csi400-VirtualBox:~/csi400$ cp ../Downloads/stage2_eltorito .
csi400@csi400-VirtualBox:~/csi400$ ls
kernel.elf  link.ld  loader.o  loader.s  stage2_eltorito
```

Figure 6: Copying of the *stage2_eltorito* file.

After copying the *stage2_eltorito* file into the current folder, the downloaded *grub-0.97* folder would have to be moved into the same folder that the *stage2_eltorito* file was copied to. The transfer of the folder to the current folder is displayed in (Figure 7).

```
csi400@csi400-VirtualBox:~/csi400$ mv ../Downloads/grub-0.97 .
csi400@csi400-VirtualBox:~/csi400$ ls
grub-0.97  link.ld  loader.s
kernel.elf  loader.o  stage2_eltorito
```

Figure 2: Moving of the *grub-0.97* folder into current folder.

2.3.4 Building an ISO Image

For the file *kernel.elf* to be able to be loaded by the virtual machine, it would have to be stored within a media, like an ISO Image file. Before one is able to create the ISO Image file, one would have to create a series of folders and place the appropriate files in each folder. This is done with the commands displayed in (Figure 8).

```
csi400@csi400-VirtualBox:~/csi400$ mkdir -p iso/boot/grub
csi400@csi400-VirtualBox:~/csi400$ ls
grub-0.97  kernel.elf  loader.o  stage2_eltorito
iso        link.ld     loader.s
csi400@csi400-VirtualBox:~/csi400$ ls iso
boot
csi400@csi400-VirtualBox:~/csi400$ ls iso/boot
grub
csi400@csi400-VirtualBox:~/csi400$ ls iso/boot/grub
csi400@csi400-VirtualBox:~/csi400$ cp stage2_eltorito iso/boot/grub/
csi400@csi400-VirtualBox:~/csi400$ ls
grub-0.97  kernel.elf  loader.o  stage2_eltorito
iso        link.ld     loader.s
csi400@csi400-VirtualBox:~/csi400$ ls iso/boot/grub
stage2_eltorito
csi400@csi400-VirtualBox:~/csi400$ cp kernel.elf iso/boot/
csi400@csi400-VirtualBox:~/csi400$ ls iso/boot
grub  kernel.elf
```

Figure 8: Displays the outcome of the call of commands to create folders and copying of files into their appropriate folders.

Next the file *menu.lst* is created that acts as a configuration file that determine where the kernel is located at and other options that are configured manually. The configuration details of the *menu.lst* file is displayed in (Figure 9).

```
default=0
timeout=0

title os
kernel /boot/kernel.elf
```

Figure 9: Configuration settings in the `menu.lst` file for GRUB.

Once the `menu.lst` file is created, it would then have to be moved the boot folder within the `iso` folder. The outcome of the moving of the `menu.lst` file is shown in (Figure 10).

```
csi400@csi400-VirtualBox:~/csi400$ gedit menu.lst
csi400@csi400-VirtualBox:~/csi400$ ls iso
boot
csi400@csi400-VirtualBox:~/csi400$ ls iso.boot
ls: cannot access 'iso.boot': No such file or directory
csi400@csi400-VirtualBox:~/csi400$ ls iso/boot
grub  kernel.elf
csi400@csi400-VirtualBox:~/csi400$ ls iso/boot/grub
stage2_eltorito
csi400@csi400-VirtualBox:~/csi400$ mv menu.lst iso/boot/grub/
csi400@csi400-VirtualBox:~/csi400$ ls iso/boot/grub
menu.lst  stage2_eltorito
```

Figure 10: Displays the files and folders within the `iso` folder.

Later an ISO image is generated by calling the following command:

```
genisoimage -R -b boot/grub/stage2_eltorito -no-emul-boot -boot-load-size 4 -A os -input-charset utf8 -quiet -boot-info-table -o os.iso iso .
```

By calling the command, an ISO image file called `os.iso` is created that would contain the `kernel.elf` file, the GRUB bootloader called `stage2_eltorito` and the `menu.lst` file. The outcome of the command to generate an ISO image is displayed in (Figure 11).

```
csi400@csi400-VirtualBox:~/csi400$ genisoimage -R -b boot/grub/stage2_eltorito -no-emul-boot -boot-load-size 4 -A os -input-charset utf8 -quiet -boot-info-table -o os.iso iso
csi400@csi400-VirtualBox:~/csi400$ gedit bochsrc.txt
csi400@csi400-VirtualBox:~/csi400$ ls
bochsrc.txt  iso          link.ld      loader.s     stage2_eltorito
grub-0.97    kernel.elf   loader.o     os.iso
```

Figure 11: The outcome of the command to create an ISO image.

2.3.5 Running Bochs

Before one can use the ISO image file, `os.iso` to run the OS within the Bochs emulator, one would have to initially configure the Bochs file. The configuration is stored in a file called `bochsrc.txt`, which is displayed in (Figure 12).

```
megs:          32
display_library: sdl
romimage:      file=/usr/share/bochs/BIOS-bochs-latest
vgaromimage:   file=/usr/share/bochs/VGABIOS-lgpl-latest
ata0-master:   type=cdrom, path=os.iso, status=inserted
boot:          cdrom
log:           bochslog.txt
clock:         sync=realtime, time0=local
cpu:           count=1, ips=1000000
```

Figure 12: Configuration settings within the *bochsrc.txt* file.

Once the configuration settings are stored in the *bochsrc.txt* file, the following command would be used to run Bochs:

bochs -f bochsrc.txt -q .

The Bochs would run and to view the Bochs console window, one would have to type in *c* into the command line of Bochs. The resulting Bochs console window is displayed in (Figure 13).

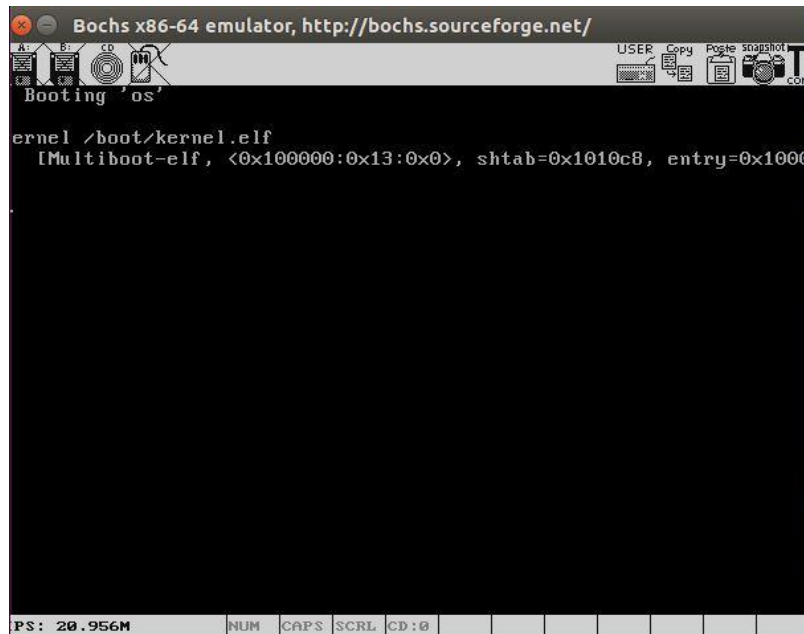


Figure 33: The Bochs console window.

After quitting Bochs, one would notice that the *eax* register would have changed to the value *0xCAFEBAE*. This change in the *eax* register can be seen by calling the command:

cat bochslog.txt .

The outcome of successfully booting the simple OS, where the *eax* register would have its' value changed to *0xCAFEBABE* is displayed in (Figure 14).

```
01473278000i[CPU0 ] | EAX=cafebabe  EBX=0002cd80  ECX=00000001  E
DX=00000000
01473278000i[CPU0 ] | ESP=00067ed0  EBP=00067ee0  ESI=0002cef0  E
DI=0002cef1
01473278000i[CPU0 ] | IOPL=0 id vip vif ac vm rf nt of df if tf s
f ZF af PF cf
01473278000i[CPU0 ] | SEG sltr(index|ti|rpl)      base      limit G
D
01473278000i[CPU0 ] |  CS:0008( 0001| 0|  0) 00000000 ffffffff 1
1
01473278000i[CPU0 ] |  DS:0010( 0002| 0|  0) 00000000 ffffffff 1
1
01473278000i[CPU0 ] |  SS:0010( 0002| 0|  0) 00000000 ffffffff 1
1
01473278000i[CPU0 ] |  ES:0010( 0002| 0|  0) 00000000 ffffffff 1
1
01473278000i[CPU0 ] |  FS:0010( 0002| 0|  0) 00000000 ffffffff 1
1
01473278000i[CPU0 ] |  GS:0010( 0002| 0|  0) 00000000 ffffffff 1
1
01473278000i[CPU0 ] | EIP=00100011 (00100011)
01473278000i[CPU0 ] | CR0=0x60000011 CR2=0x00000000
01473278000i[CPU0 ] | CR3=0x00000000 CR4=0x00000000
01473278000i[CMOS ] Last time is 1519220096 (Wed Feb 21 08:34:56
2018)
01473278000i[CTRL ] quit_sim called with exit code 1
csi400@csi400-VirtualBox:~/csi400$
```

Figure 4: The outcome of the successful booting of the simple OS.

Section 3: Getting to C

Each C and Assembly have their own advantages. For a user, C can be a more readable and friendlier language to use, and Assembly is useful for interacting with the CPU and enables maximum control over every aspect of the code. In this part, we will invert assembly into the C as the programming language for the OS.

3.1 Setting Up a Stack

Every single non-trivial C programs use a stack, which means to use C we must have a stack. Due to the difficulty of the knowing the amount of memory available by pointing to an esp, we will use uninitialized memory in the *bss* section in the ELF file of kernel. Because the

GRUB can recognize the ELF, when the OS is loading, it will allocate any reserved memory in the *bss*.

Displayed in (Figure 15), *resb* is used to declare uninitialized data.

```
KERNEL_STACK_SIZE equ 4096      ; size of stack in bytes

section .bss
align 4                          ; align at 4 bytes
kernel_stack:                    ; label points to beginning of
memory
    resb KERNEL_STACK_SIZE      ; reserve stack for the kernel
```

Figure 15: The NASM pseudo-instruction *resb* within the *loader.s* file.

The program always reads the memory location before it writes to the stack. We can use the uninitialized memory for the stack.

In line 3 displayed in (Figure 16), we set the *esp* as a stack pointer to the end of the *kernel_stack* memory.

```
loader:                          ; the loader label (defined as
entry point in linker script)
    mov esp, kernel_stack + KERNEL_STACK_SIZE ; point esp to
the start of the                      ; stack (end of memory area)
```

Figure 16: Displays the stack pointer pointing to the end of the *kernel_stack* memory.

3.2 Calling C Code From Assembly

After setting up a stack, now we will call a C function from the assembly code. We will use *cdecl* calling convention which is used by the GCC. The *cdecl* calling convention states the arguments to a function through the stack on x86. To use a *cdecl*, we have to push the rightmost argument first. The *eax* register will hold the return value of the function.

Displayed in (Figure 17 and 18), one would be able to view an example of a function that is the sum of three values.

```
csi400@csi400-VirtualBox:~/csi400$ cat kmain.c
/* The C function */
int sum_of_three(int arg1, int arg2, int arg3)
{
    return arg1 + arg2 + arg3;
}
csi400@csi400-VirtualBox:~/csi400$ nasm -f elf32 loader.s
csi400@csi400-VirtualBox:~/csi400$
```

Figure 17: Displays the C function of *sum_of_three*.

```

; The assembly code
extern sum_of_three      ; the function sum_of_three is
defined elsewhere

push dword 3             ; arg3
push dword 2             ; arg2
push dword 1             ; arg1
call sum_of_three        ; call the function, the result
will be in eax

```

Figure 18: Displays the assembly code for the `sum_of_three` function.

3.2.1 Packing Structs

Packing structures can be more useful than unsigned integers. It is a structure which has the unsigned integers as the members. However, the compiler can add some padding between elements, causing uncertainty of whether the size of the structs are exactly 32 bits. Therefore, it is better to management GCC to avoid adding any padding.

3.3 Compiling C Code

For our OS, we are assuming there is no C standard library. So, we will be using multiple flags to GCC for compiling the C code, the following flags are:

-m32 -nostdlib -nostdinc -fno-builtin -fno-stack-protector -nostartfiles -nodefaultlibs .

Also, one would have to consider about how to handle warnings, which the following flags would account for:

-Wall -Wextra -Werror .

Now, we created a function `kmain` in a file called `kmain.c`. Right now, `kmain` does not need any arguments.

3.4 Build Tools

Here, we set up a *Makefile* so our OS is easier to compile and test-run, displayed in (Figure 19 and 20).

```

csi400@csi400-VirtualBox:~/csi400$ gedit Makefile
csi400@csi400-VirtualBox:~/csi400$

```

Figure 19: The creation of the *Makefile*.

```

OBJECTS = loader.o kmain.o
CC = gcc
CFLAGS = -m32 -nostdlib -nostdinc -fno-builtin -fno-stack-
protector \
        -nostartfiles -nodefaultlibs -Wall -Wextra -Werror -c
LDLFLAGS = -T link.ld -melf_i386
AS = nasm
ASFLAGS = -f elf

all: kernel.elf

kernel.elf: $(OBJECTS)
    ld $(LDLFLAGS) $(OBJECTS) -o kernel.elf

os.iso: kernel.elf
    cp kernel.elf iso/boot/kernel.elf
    genisoimage -R
        -b boot/grub/stage2_eltorito
        -no-emul-boot
        -boot-load-size 4
        -A os
        -input-charset utf8
        -quiet
        -boot-info-table
        -o os.iso
    iso

run: os.iso
    bochs -f bochsrc.txt -q

%.o: %.c
    $(CC) $(CFLAGS) $< -o $@

```

Figure 20: Displays the commands and flags within the Makefile.

Now we start the OS with the simple command: *make run*. Makefile compiles the kernel and boots it up in Bochs, which is displayed in (Figure 21). There was an issue with getting Bochs to open the console, but after doing some research online, we noticed that we only had to type the single character *c* in the Bochs emulator to open the console.

```

Bochs x86-64 emulator, http://bochs.sourceforge.net/
Booting 'os'
kernel /boot/kernel.elf
[Multiboot-elf, <0x100000:0x6c:0x0>, <0x101000:0x0:0x1000>, sht
entry=0x1000201]

IPS: 4.232M
NUM CAPS SCRL CD:0
[PLGIN] loaded plugin libbx_sdl.so
[ ] installing sdl module as the Bochs GUI
[ ] using log file bochslog.txt
Next at t=0
(0) [0x00000000ffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b
; ea5be00f0
<bochs:1> c

```

Figure 21: Displays the running of the console for Bochs.

Section 4: Output

When it comes to building a program, user interface involving the output is a key component to well written code. This section includes writing data to the serial port, as well as creating a driver (code that translates information between the kernel and the hardware), providing a higher abstraction than communicating directly to the hardware. Displaying text on the console includes creating a driver for the framebuffer, which will be explained going further into this section.

4.1 Interacting with the Hardware

Two different ways to interact with the hardware include memory mapped I/O and I/O ports. With memory mapped I/O, writing to a specific memory address will update the hardware with new data. With I/O ports, the assembly code instructions must be used to communicate with the hardware. The instruction *out* takes two parameters (the address of the I/O port and the data to send), while the instruction *in* takes a single parameter (the address of the I/O port) and returns data from the hardware.

4.2 The Framebuffer

The framebuffer is a hardware device that has a data structure that organizes the memory resources that allows a buffer of memory to be displayed on the screen.

4.2.1 Writing Text

Memory-mapped I/O is required for the framebuffer to be able to write text to the console. The objective is to write to a character with a unique background to be displayed in the console. To understand how it is done, first one would have to understand that the starting address of the memory-mapped I/O is located at 0x000B8000 for the framebuffer. Next to be able to write the text to the console, the character would have to be stored in *fb[0]*, while the color scheme would have to be stored in *fb[1]*. An example of the C code stored in the *kmain.c* file is displayed in (Figure 22), which would have the character A with a green foreground written to the console.

```

#define FB_GREEN      2
#define FB_DARK_GREY 8

char *fb = (char *) 0x000B8000;

/** fb_write_cell:
 * Writes a character with the given foreground and background to
 * position i
 * in the framebuffer.
 *
 * @param i The location in the framebuffer
 * @param c The character
 * @param fg The foreground color
 * @param bg The background color
 */
void fb_write_cell(unsigned int i, char c, unsigned char fg,
unsigned char bg)
{
    fb[i] = c;
    fb[i + 1] = ((fg & 0x0F) << 4) | (bg & 0x0F);
}

void kmain(){
    fb_write_cell(0, 'A', FB_GREEN, FB_DARK_GREY);
}

```

Figure 22: Displays the C code required to write the character A with a green foreground to the console.

The result of the code provided in *kmain.c* file is displayed in (Figure 23).

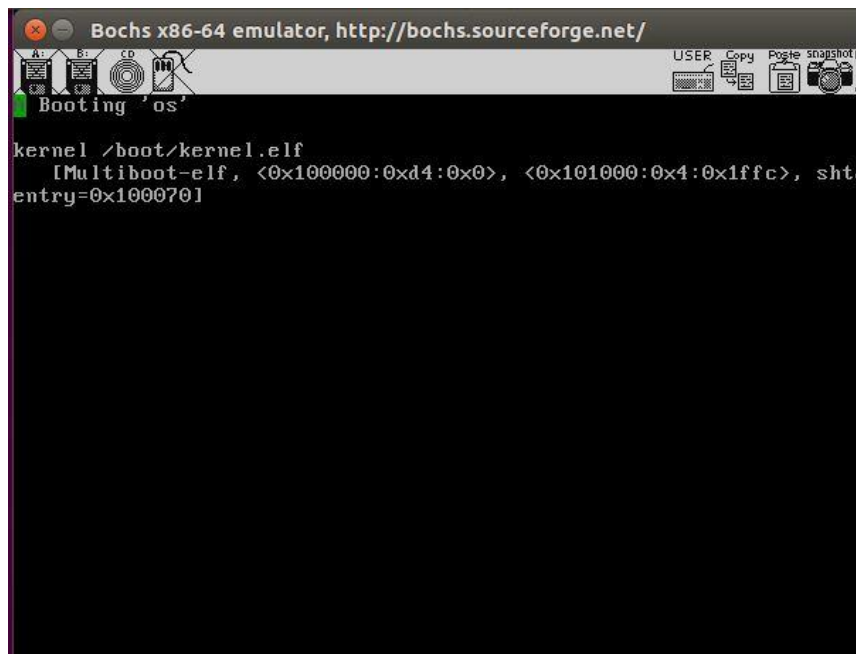


Figure 23: Displays the character A with a green foreground on the console.

4.2.2 Moving the Cursor

To be able to move the cursor, the framebuffer is required to use two different I/O ports. One port for accepting the data, and another port for describing the data being received. The data would be contained in port 0x3D5, while the description of the data is contained in port 0x3D4. An assembly code instruction is created to be able to send bytes to the I/O port, which is stored in the *io.s* file. The *out* assembly code instruction is wrapped in a function in assembly code that is displayed in (Figure 24).

```
global outb                                ; make the label outb visible
outside this file

; outb - send a byte to an I/O port
; stack: [esp + 8] the data byte
;        [esp + 4] the I/O port
;        [esp    ] return address

outb:
    mov al, [esp + 8]                      ; move the data to be sent into
the al register                             ;
    mov dx, [esp + 4]                      ; move the address of the I/O
port into the dx register                  ;
    out dx, al                             ; send the data to the I/O port
    ret                                   ; return to the calling function
```

Figure 24: Displays the *out* assembly code instruction.

A header file *io.h* is created, so that it would allow C to conveniently access the *out* assembly code instruction. The structure of the header file for the *out* function is displayed in (Figure 25).

```
csi400@csi400-VirtualBox:~/csi400$ cat io.h
#ifndef INCLUDE_IO_H
#define INCLUDE_IO_H

/** outb:
 * Sends the given data to the given I/O port. Defined in io.s
 *
 * @param port The I/O port to send the data to
 * @param data The data to send to the I/O port
 */
void outb(unsigned short port, unsigned char data);

#endif /* INCLUDE_IO_H */
```

Figure 25: Displays the header file *io.h* for the *out* function.

Next a C function is created that wraps the moving of the cursor, which is displayed in (Figure 26).

```
csi400@csi400-VirtualBox:~/csi400$ cat kmain.c
#include "io.h"

/* The I/O ports */
#define FB_COMMAND_PORT      0x3D4
#define FB_DATA_PORT        0x3D5

/* The I/O port commands */
#define FB_HIGH_BYTE_COMMAND 14
#define FB_LOW_BYTE_COMMAND  15

/** fb_move_cursor:
 * Moves the cursor of the framebuffer to the given position
 *
 * @param pos The new position of the cursor
 */
void fb_move_cursor(unsigned short pos)
{
    outb(FB_COMMAND_PORT, FB_HIGH_BYTE_COMMAND);
    outb(FB_DATA_PORT, ((pos >> 8) & 0x00FF));
    outb(FB_COMMAND_PORT, FB_LOW_BYTE_COMMAND);
    outb(FB_DATA_PORT, pos & 0x00FF);
}
```

Figure 26: Displays the C function called `fb_move_cursor` and the I/O port commands.

The `fb_move_cursor` function would then be called by the `kmain` function, displayed in (Figure 27).

```
void kmain(){
    fb_write_cell(0, 'A', FB_GREEN, FB_DARK_GREY);
    fb_move_cursor(100);
}
```

Figure 27: Displays the `kmain` function calling the `fb_move_cursor` function.

This would then cause a shift of the cursor to be displayed on the console, which the result is displayed in (Figure 28).

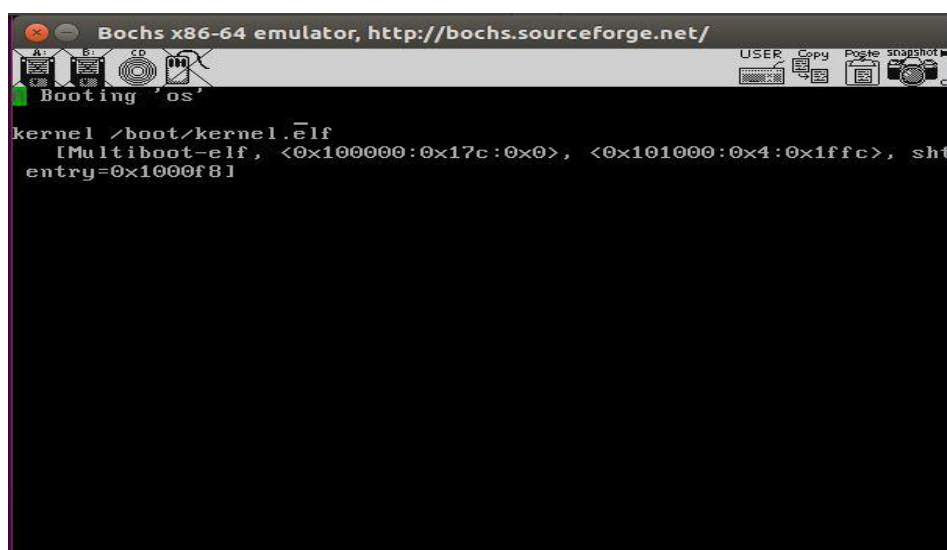


Figure 28: Displays the shifting of the cursor.

4.2.3 The Driver

An interface is preferred to allow interaction with the framebuffer for the rest of the code in the OS, and this should be provided by the driver. This can be done with a *write* function that would be able to write the contents in a buffer of a certain length to the screen, while shifting the cursor.

4.3: The Serial Ports

The serial port allows for hardware devices to be able to transmit and communicate with each other through an interface. It is available on most motherboards, though it is exposed to the user in the form of a DE-9 connector. In this section, we will be using a serial port as an output, not an input. Also, serial ports are controlled through the use of I/O ports.

4.3.1 Configuring the Serial Port

The configuration data is the first to be sent to the serial port. This is an important step in allowing for two devices to be able to communicate with each other. Both devices have to agree on a couple of things, like the rate of sending data, whether parity bits are required, and the representation of a unit of data.

4.3.2 Configuring the Line

The configuration of how the data is sent over the line is labelled as configuring the line. The *line command port* is an I/O port that is used configuration by the serial port. Initially the rate of sending data needs to be determined and set. The internal clock for a serial port tends to run at 115200Hz. To set the rate of sending data, one would have to send a divisor to the serial port. A function to configure the rate of sending data for the serial port is displayed in (Figure 29).

```

#include "io.h"

/* All the I/O ports are calculated relative to the data port. This is because
 * all serial ports (COM1, COM2, COM3, COM4) have their ports in the same
 * order, but they start at different values.
 */

#define SERIAL_COM1_BASE      0x3F8      /* COM1 base port */

#define SERIAL_DATA_PORT(base)    (base)
#define SERIAL_FIFO_COMMAND_PORT(base) (base + 2)
#define SERIAL_LINE_COMMAND_PORT(base) (base + 3)
#define SERIAL_MODEM_COMMAND_PORT(base) (base + 4)
#define SERIAL_LINE_STATUS_PORT(base) (base + 5)

/* The I/O port commands */

/* SERIAL_LINE_ENABLE_DLAB:
 * Tells the serial port to expect first the highest 8 bits on the data port,
 * then the lowest 8 bits will follow
 */
#define SERIAL_LINE_ENABLE_DLAB      0x80

/** serial_configure_baud_rate:
 * Sets the speed of the data being sent. The default speed of a serial
 * port is 115200 bits/s. The argument is a divisor of that number, hence
 * the resulting speed becomes (115200 / divisor) bits/s.
 *
 * @param com      The COM port to configure
 * @param divisor  The divisor
 */
void serial_configure_baud_rate(unsigned short com, unsigned short divisor)
{
    outb(SERIAL_LINE_COMMAND_PORT(com),
        SERIAL_LINE_ENABLE_DLAB);
    outb(SERIAL_DATA_PORT(com),
        (divisor >> 8) & 0x00FF);
    outb(SERIAL_DATA_PORT(com),
        divisor & 0x00FF);
}

```

Figure 29: Displays the function stored in kmain.c file, where it would set the rate of sending data over the serial port.

The way data must be sent will have to be configured. The line command port is able to configure the way that data is being sent over the line by sending a byte. A function to configure the way that data is being sent over the line, is displayed in (Figure 30).

```

/** serial_configure_line:
 * Configures the line of the given serial port. The port is set to have a
 * data length of 8 bits, no parity bits, one stop bit and break control
 * disabled.
 *
 * @param com  The serial port to configure
 */
void serial_configure_line(unsigned short com)
{
    /* Bit:      | 7 | 6 | 5 4 3 | 2 | 1 0 |
     * Content:   | d | b | prty  | s | dl  |
     * Value:     | 0 | 0 | 0 0 0 | 0 | 1 1 | = 0x03
     */
    outb(SERIAL_LINE_COMMAND_PORT(com), 0x03);
}

```

Figure 30: Displays the function stored in kmain.c file, where it would set the way that data is being sent.

4.3.3 Configuring the Buffers

For data to be sent over the serial ports, the data would have to be stored in buffers. This happens for both when receiving and sending data over the serial ports. If the rate of sending data to the serial port is greater than the rate of transmission over the wire, then the data would be placed on in a buffer. Also, there is another situation where if the rate of sending data is too fast and the buffer fills up, then some data will be lost.

4.3.4 Configuring the Modem

Through the use of Ready To Transmit and Data Terminal Ready pins, the modem control register is to maintain the hardware flow control. By setting the Ready To Transmit and Data Terminal Ready pins to 1, it acts as an indicator to let us know that the data is ready to be sent. Interrupts won't be needed because no received data is being handled.

4.3.5 Writing Data to the Serial Port

The data I/O port is used for writing data to the serial port, but is required to initially check if the transmit FIFO queue is empty or not. To be able to read the contents of the data in the I/O port, we would need the *in* assembly code instruction. The *in* assembly code instruction is wrapped, to allow the code to be used by C. This is shown in (Figure 31 and 32).

```
glob /** inb:
*   Read a byte from an I/O port.
; in*
; st*
;    *   @param port The address of the I/O port
inb:*   @return    The read byte
port */
unsigned char inb(unsigned short port);
```

Figure 32: Displays the header file io.h for the in function.

Figure 31: Displays the in assembly code instruction in the file io.s.

Next a function is created to check if the transmit FIFO is empty, which is displayed in (Figure 33).

```
/** serial_is_transmit_fifo_empty:
*   Checks whether the transmit FIFO queue is empty or not for the given COM
*   port.
*
*   @param com The COM port
*   @return 0 if the transmit FIFO queue is not empty
*           1 if the transmit FIFO queue is empty
*/
int serial_is_transmit_fifo_empty(unsigned int com)
{
    /* 0x20 = 0010 0000 */
    return inb(SERIAL_LINE_STATUS_PORT(com)) & 0x20;
}
```

Figure 33: Displays the serial_is_transmit_fifo_empty function that is stored in the kmain.c file.

By writing to the serial port, the data would be written to the data I/O port. But for that to happen, it would have to be spinning, which is true as long as the transmit FIFO queue is not empty.

By running the functions that configures the serial port, there is no output that would be able to be viewed by the user on the console.

4.3.6 Configuring Bochs

To be able to save the output from the first serial port, the *bochsrc.txt* configuration file had to be modified. The modification to the bochsrc.txt file is as follows:

com1: enabled=1, mode=file, dev=com1.out .

By modifying the *bochsrc.txt* file, the output of the serial port one will be stored in the *com1.out* file.

4.3.7 The Driver

An interface would be preferred that would allow interaction with the serial port, and this should be provided by the driver. This can be done with a *write* function that would be able to write the contents in a buffer of a certain length to the serial port.

Section 5: Segmentation

Segmentation in x86 means that one accesses the memory through segments. These segments are smaller parts of an address space specified by a base address and a limit. To address a byte in segmented memory, it is best to use a 48-bit *logical address*. The 48-bit logical address is separated into two parts. The first 16 bits are used to specify the segment selector while the rest of the 32-bits are used to specify the offset that we would want. The linear address is derived from the addition of the offset and the base address of the segment. Then the linear address would have to be compared to see if it had past the segment's limit. This process is shown in (Figure 34).

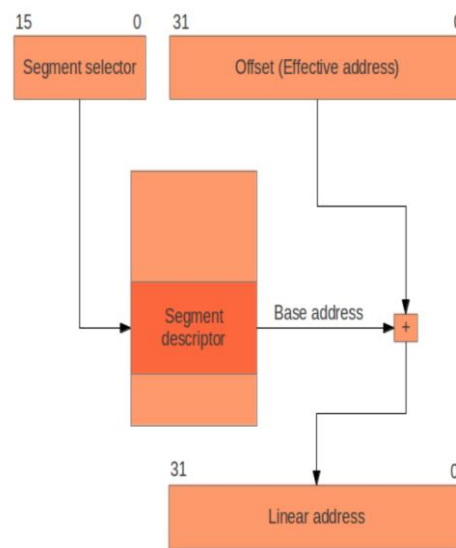


Figure 34: Displays the process of segmentation.

A segment descriptor table is required to describe each segment for the process of segmentation. In x86, the *Global Descriptor Table* (GDT) and *Local Descriptor Tables* (LDT) represent the two types of descriptor tables. User-space processes are what manages and sets up *the Local Descriptor Tables* and that would result in all processes having their own LDT.

5.1 Accessing Memory

When accessing memory, there is no reason to specify which segment to use. This processor

```

func:
    mov eax, [ss:esp+4]
    mov ebx, [ds:eax]
    add ebx, 8
    mov [ds:eax], ebx
    ret
  
```

Figure 35: Displays an example of segment registers.

contains 6 different 16-bit segment registers: *cs*, *ss*, *ds*, *es*, *gs* and *fs*. The *cs* register stands for code segment. It is used to specify the segment to use when fetching instructions. The *ss* register is used whenever you need to access the stack. The *ds* register is used for other data accesses. The OS can use the registers *es*, *gs* and *fs* in whatever way it wants. An example of using the segment registers is displayed in (Figure 35).

5.2 The Global Descriptor Table (GDT)

The Global Descriptor Table is a data structure used by Intel x86-family processors starting with the 80286 in order to define the characteristics of the various memory areas used during program execution, including the base address, the size, and access privileges like executability and writability. These memory areas are called segments in Intel terminology. It is also an array of 8-byte segment descriptors that holds a null value for the first descriptor. For this system to work, two segments are needed: one segment for executing code to put in cs and one segment for reading and writing data to put in the other segment registers. “*privilege levels* required to use the segment. x86 allows for four privilege levels (PL), 0 to 3, where PL0 is the most privileged. In most operating systems (eg. Linux and Windows), only PL0 and PL3 are used.” it is good to remember that segments can overlap. The use of these levels and the segments needed are shown in (Figure 36).

Index	Offset	Name	Address range	Type	DPL
0	0x00	null descriptor			
1	0x08	kernel code segment	0x00000000 – 0xFFFFFFFF	RX	PL0
2	0x10	kernel data segment	0x00000000 – 0xFFFFFFFF	RW	PL0

Figure 36: Displays a table of the segments.

5.3 Loading the GDT

Loading the GDT into the processor can be done with a simple code instruction. Lgdt takes the address of the struct which tells you the beginning and the size of the GDT. Encoding this information can be done using a packed struct. The structure of the gdt in C is shown in (Figure 37).

```
struct gdt{
    unsigned int address;
    unsigned short size;
}__attribute__((packed));
```

Figure 37: Displays structure of the gdt.

If the eax register is the address to such a struct, then the GDT can be loaded with the assembly code:

lgdt [eax] .

After the GDT has been loaded the segment registers needs to be loaded with their corresponding segment selectors. Displayed in (Figure 38), it represents the content of a segment selector.

Bit:	15				3	2	1 0	
Content:	offset (index)					ti	rpl	

Figure 38: Displays the contents of a segment selector.

Loading the segment selector registers is as easy as copying the correct offsets to the registers. An example of copying the correct offsets to the registers is shown in (Figure 39).

```
mov ds, 0x10
mov ss, 0x10
mov es, 0x10
```

Figure 39: Displays the copying of the offsets to the registers.

By using the “far jump”, we would be able to load the *cs*, which is shown in (Figure 40).

```
; code here uses the previous cs
jmp 0x08:flush_cs ; specify cs when jumping to flush_cs

flush_cs:
; now we've changed cs to 0x08
```

Figure 40: Displays the far jump to load *cs*.

Conclusion:

From the beginning of the project to the current state of the project, we started with a simple operating system that would store a predetermined value into a register, but then we developed the operating system even further. We developed to the point that the operating system can write to the console, as well as configure a serial port. At the current state of the project, we left off at segmentation, where we learned that we can view memory by using segments. This can be related to the importance of how processes are able to function like they do when trying to complete tasks for the operating system. One can say that during the development of the operating system, we learned many things. But the main thing we learned about this project is how the operating system can get very complicated because everything has a part or function that makes the operating system run properly. But this is all possible through memory manipulation by the use of a combination of C code and assembly code.

Works Cited

- <http://www.cs.yale.edu/homes/aspnes/pinewiki/UsingBochs.html>
- <https://kb.iu.edu/d/ahtz>
- https://www.cs.tau.ac.il/telux/lin-club_files/linux-boot/slide0002.htm
- <http://math.hws.edu/graphicsbook/c7/s4.html>