Homework 1
Assignment

Due February 3 at 11:59 PM

Starts Jan 25, 2023 8:00 AM

## *CSC 4100 Project 1*

*Memory Test*

## Skills you will learn

- Using various Unix tools to build an OS
- Using the qemu emulator and the Gnu debugger to run and debug your OS
- Compiling a boot loader and the OS
- Installing your boot loader and OS
- Writing to the screen
- Calling an assembly function from C

## Things you will need

- Virtualbox virtual machine (with all the tools) (Download and install into VirtualBox on your host machine)
- .mtoolsrc (Save to the coursework directory on the VM)
- boot1.asm (Save to you project directory on the VM)
- buddy.tar.gz (memory allocation routines and memory tester)

## Description

### Introduction

You will build a stage 1 boot loader and a stage 2 operating system (OS). This initial boot loader has already been written; You simply need to download it, compile it, and install it. It will boot from a (virtual) floppy and put the computer into protected mode. It then will load the seconds stage OS. The second stage OS will clear the screen, draw a border around the screen, and then call a function written by the professor that tests his memory allocator code (also written by the professor). Next, it will again clear the screen, and fill the entire screen with stars (the '*' characters).

### Tools that you will need

You will need to use the following tools to complete your assignment, and ***all of them are provided as part of the course virtual machine***: You will need to download and install VirtualBox. However, for the rest, you do ***not*** need to download, install, or set them up. Their descriptions below are just to familiarize you with their function.

- dosfstools. Specifically, the **mkdosfs** program will format a virtual floppy.
- The **dd** copy utility. This utility will build a virtual floppy disk (a file called a.img) for you. Also, you will use this utility to copy your stage 1 boot loader to the virtual floppy.
- The mtools suite of utilities, specifically **mcopy**. You will use this utility to copy your stage 2 OS to the virtual floppy.

- The **Qemu** x86 emulator. Your will use this emulator to run, and test your operating system.
- **GDB**, the Gnu Debugger. You will use this tool to debug your operating system.
- **DDD**, the Data Display Debugger. This program provides a graphical user interface to GDB. It has some nice features that you will not find in other debuggers, even though the interface is old.
- The **nasm** assembler. You will use this assembler to generate the stage 1 boot loader, and to generate the object file for the part of your OS written in assembly.
- **gcc**, the Gnu C compiler. You will use the cross compiler to generate object code for the main driver of your stage 2 operating system.
- **objcopy**, the object code translator. You will use objcopy to strip out all symbolic information from the stage 2 OS code.
- **Debian**, a distribution of Linux. You can use another distribution if you wish. You will, however be on your own in getting things to work.
- **VirtualBox**, a free but commercial grade emulator. VirtualBox will allow you to run Debian Linux under Windows (XP, Vista, or 7), Mac OS X, or another distribution of Linux without having to install Linux on your hard drive in a separate partition. At the end of the class, if you do not want to keep Linux, you can simple delete the virtual machine and uninstall VirtualBox.

## Creating Your OS

In order to create your operating system you need to download the boot loader at boot1.asm. This boot loader will load into memory at boot time, put the computer into 32 bit protected mode, and then load your stage 2 OS. We will talk more about the boot loader in class.

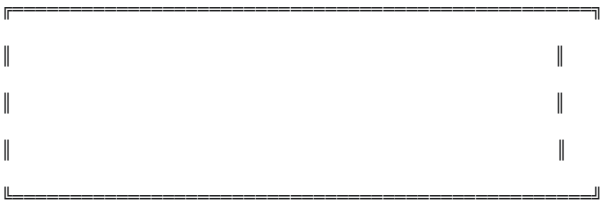
Next you will need to write a simple main driver in C. Your main will print a border around the screen and then call a function to test a memory allocator. You will used the routines from the memory allocator in future assignments. The memory allocator has already been written by the professor and, along with a function to convert a number to a string, is found here: buddy.tar.gz. So, the pseudo-code for the main is:

```
main:
    call the function to clear the screen      // written by you in C
    call the function to print the border      // written by you in C
    call run_test()                            // written by the professor
    loop INT_MAX times
    call the function to clear the screen
    loop i from 0 to 24
       print a row of 80 stars ('*') at line i, row 0
    end loop
    loop forever      // Its done, but the OS has nowhere which to return!
```

The function to print the border has the following prototype.

```
void print_border(int start_row, start_col, int width, int height);
```

This function simply draws a border using ASCII characters around the edge of the screen. The professor will leave the details of writing this function to you (you should be able to figure it out - its easy). The constraints are that the corners of the border are these following characters from VGA code page 437 (the code page to which the VGA text mode defaults): top left is \xC9, top right is \xBB, bottom left is \xC8, and the bottom right is \xBC. The top and bottom edges are \xCD , and the right and left sides are \xBA. Note that the screen has 25 rows and 80 columns, so the top edge is at row 0, the bottom edge is at row 24, the left edge is at column 0, and the right edge is at column 79. The results of print the screen should result in a box that looks like the following:



In order to clear the screen and print to the screen (which is needed not only to print the border, but also by the memory allocator test), you will need to write two functions. One function, called k_printstr(), will be in assembly. The other function called K_clearscr() can be written in C. The functions have the following prototypes:

```
// Note that k_clearscr() can also clear a portion of the screen
void k_clearscr(int start_row, int start_col, int width, int height);
void k_printstr(char *string, int row, int col);
```

The k_printstr() function must be written in assembly to get credit for this assignment. You can write the k_clearscr() and print_border() functions in C/C++, but they ***must*** call k_printstr() to get their work done. Note that you will need to understand how parameters are passed from C to an assembly language function. The professor will talk about how to do so in class, and the tutorial slides on assembly have a description and examples. The k_printstr function will simply print characters to the screen, one at a time, advancing the column, until it reaches a zero byte (the NULL terminator for strings).
How can you print a character on the screen? It is actually easier than you might think. When your computer boots, it maps memory from the video card into what is called the ISA hole, which is memory at the top of the first megabyte of RAM. In particular, the video cards memory is mapped to the address 0xB8000, as shown in the Figure 1.

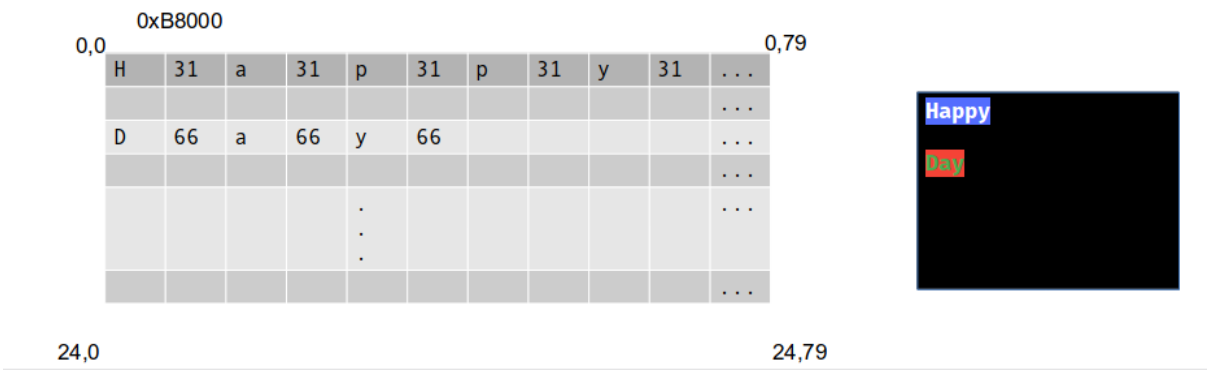



Figure 1: To make a character appear on the screen, we move bytes into video memory at 0xB8000 However, video memory is actually two bytes we want to display. The first byte is the actual character, and the second byte is its color attributes. The attribute byte has the foreground color in it the background color in its highest 3 bits. So, 31 in hex is 0x1F. F is 15 (white) on 1 (Blue). So, moving a 'x' followed by the byte 31 prints a white background. Likewise, ted is 4 and green is 2, so 0x42, which is 66 in base 10 is green on red.

The default text mode of a VGA video card has 80 columns and 25 rows. In other words, you can write 80 characters on a single line, and you can write 25 of these lines on a single screen. So, you might think that the amount of memory mapped to address 0xB8000 is 80 characters (bytes) time 25. However, it is exactly twice that! Each character also must be succeeded by a color byte. For example, if you want to place an 'A' at the top left hand corner of the screen, and make it white on a blue background, then you would move the 'A' into 0xB8000, and a 31 (which represents a blue background and white text) at 0xB8001. Likewise, if you wanted an X to follow the A so that it is immediately to its right, you would move an 'X' into 0xB8002 and a 31 into 0xB8003. So, the algorithm for k_printstr() would be the following:

```
k_printstr    /* see above for arguments */
  calculate the offset into video memory using row and col
     (offset = (row * 80 + col) * 2)
  calculate the absolute address of video memory for storing
     the next character by adding 0xB8000 and the offset
  loop:
     if string address poins to 0 byte then
           go to loop_end
     if the address where to write the next character has
```

```
        passed the end of video memory (0xB8000+80*25*2) then
             go to loop_end
      mov the next character from the string into video memory
          and increment the string address and the address where to
        write the next character (can be done in one instruction: movsb)
      mov a color byte into the video memory
      increment the address where to write the next character
      goto loop
    loop_end:
      clean up and return to the caller
```

k_clearscr() is really easy. Write a function in C that calls k_printstr() twenty five times passing it a string of the required number of spaces. That's it!

Finally, you will write the main(). Your main() will simply call k_clearscr() to clear the entire screen followed by print_border(), followed by a call the a function named run_test(). You do not write the run_test() function. It is provided in the code that you downloaded (in buddy.tar.gz). However, the run_test() functions makes calls to your k_printstr(). So, you must follow the prototype above exactly, including capitalization. Next, as show in the pseucode in the main displayed earlier in this assignment description, you will write a loop that spins for approximately INT_MAX (found in limits.h) iterations, and add another loop that fills the screen with stars.

Your C code should go into a file called kernel.c, and your assembly code should go into a file called kernel.asm. After you have written the C and assembly language parts of your stage 2 OS, you will need to build your OS, install it, and run it.

## Building Your OS

You will create a makefile to build your OS. The following is the output of my makefile when I build my OS (with comments):

```
# First, create the stage 2 OS
gcc -g -m32 -fno-stack-protector -c -o kernel_c.o kernel.c
nasm -g -f elf -F dwarf -o kernel_asm.o kernel.asm
gcc -g -m32 -fno-stack-protector -c -o convert_c.o convert.c
gcc -g -m32 -fno-stack-protector -c -o buddy_c.o buddy.c
ld -g -melf_i386 -Ttext 0x10000 -e main -o kernel.exe kernel_c.o kernel_asm.o convert_c.o buddy_c.o

 # Now compile the boot loader. Notice that we fix up the entry point to
# the stage 2 OS.    Note that the actual command in the Makefile should
# be the following because the ENTRY may change between compilations:
# nasm -DENTRY=0x$(shell nm kernel.exe | grep main | cut -d " " -f 1) boot1.asm
# Note: remove the word "shell" from the command above to execute it on
# the command line instead of a Makefile.

nasm -DENTRY=0x00010000 boot1.asm


# Strip out all symbolic (ELF) information from the stage 2 OS
objcopy -j .text* -j .data* -j .rodata* -S -O binary kernel.exe boot2
```

Your Makefile should compile your OS similarly. Make sure you write a "good" makefile. In other words, you should set up your dependencies such that no unecessary compiling occurs. **If you do not write a correct Makefile and do not implement the dependencies to reflect the dependency graph below, then you will get a zero on this assignment!** A working Makefile should, if you change your assembly language file, compile only the assembly file but not your C file. Figure 2 shows the dependency tree that your Makefile should implement.

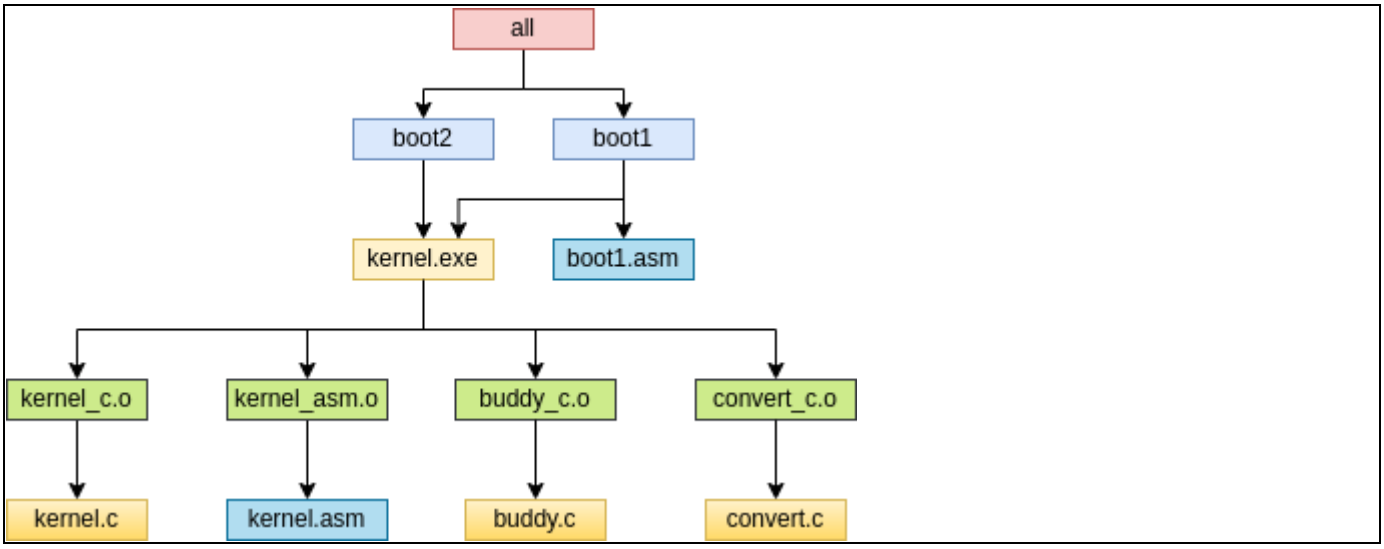



Figure 2: Makefile dependency graph. The root target is all. All has two children, boot2 and boot1. Boot2 has the child kernel.exe. kernel.exe has four children, kernel_asm.o, kernel_c.o, buddy_c.o, and convert_c.o. kernel_asm.o has one child kernel.asm. kernel_c.o has one child kernel.c. Buddy_c.o has child buddy.c and convert_c.o has child convert.c. Boot1 has two children, kernel.exe and boot1.asm.

*Include a target called "clean" in your makefile.* You should be able to type "make clean" to wipe out all of your object files, your boot loader binary, and your OS binary and their intermediate files (in other words, only the source should be left). Also, include a target for run and debug (see below) in your makefile.

## Installing your OS

Now that everything is built, you will need to create a virtual floppy, install the stage 1 boot loader, and install the stage 2 OS. To create the virtual floppy, run the dd utility as follows:

```
dd if=/dev/zero of=a.img bs=512 count=2880
```

After you create the image, you will need to format it using the mkdosfs command. Execute the following command to format the virtual floppy:

```
/sbin/mkdosfs a.img
```

The following command will install the stage 1 boot loader (copy it to sector zero of the virtual floppy):

```
dd if=boot1 of=a.img bs=1 count=512 conv=notrunc
```

Finally, copy the stage 2 OS to the virtual floppy using mcopy. Download this configuration file at mtoolsrc.conf and save it as .mtoolsrc in your home directory (notice the period as the first character of the file name), then run the following command:

```
mcopy -o boot2 a:BOOT2
```

## Running Your OS

Now you can run your OS and see if it works. At the Linux prompt, issue the following command:

```
qemu-system-i386 -drive format=raw,file=a.img,if=floppy
```

## Debugging Your OS

To debug your OS, you will need to start Qemu in debug mode by adding both the -S and -s options as follows.

```
qemu-system-i386 -S -s -drive format=raw,file=a.img,if=floppy
```

Qemu will launch and then immediately halts at the very first instruction, which happens to be in the BIOS. At this point, Qemu is waiting for you to attach to it with the GDB debugger. So, open another Linux terminal and enter the following command:

```
ddd kernel.exe
```

Next, At the gdb prompt, enter the following:

```
target remote localhost:1234
```

Make your debugging life super easy by using the following as a rule in your makefile. This rule, when you type "make debug" will automatically start qemu, start the debugger, connect to qemu, and set a breakpoint on main():

```
debug:
    qemu-system-i386 -S -s -drive format=raw,file=a.img,if=floppy &
    ddd --debugger 'gdb -ex "target remote localhost:1234" -ex "break main" -
ex "continue"' kernel.exe
```

Now, at the gdb prompt you can entry a number of debugging commands that allow you to look at registers, memory, single step through the code (in assembler), continue booting, etc. You should reference the gdb manual found at http://sourceware.org/gdb/current/onlinedocs/gdb/. Also, your VM has a graphical debugger called DDD that you may find easier to use. Run it by executing "DDD kernel.exe". Then type the command given above at the gdb prompt at the bottom of the graphical interface. DDD will attach to qemu, and you can single step, step over, and use other debugging functions by simply pressing buttons, and you can hover over variables to see their values in balloon pop-ups.

Your makefile should include three targets that will make managing your project easier to install and run, one called "install", one called "debug", and one called "run". I should be able to type "make install" to copy your boot loader and OS to the virtual floppy. I should be able to type "make debug" to start Qemu in debugging mode so that I can open a terminal and attach a debugger to it. Finally, if I want to watch your OS without pause (without it going into debugging mode), I should be able to type "make run" to run Qemu, *without* the -S and -s options, and watch your OS boot up and print its message.

A video of how the finished product should look, including how the Makefile should behave can be found at the following link:

Homework 1 Demo

## Turn In

You will submit your entire project's directory. Name your project directory as yourTnTechId_p1. For example, if your TnTech email is sclaus42, then your project directory name should be sclause42_p1. "make clean" your directory first. Then, you can create your tarball with the following command:

```
tar -czf yourTnTechId_p1.tar.gz yourTnTechId_p1/
```

(Make sure to substitute your TnTech id for yourTechId as you did for your directory name.)