**Things you will need**

* (Download and install into VirtualBox on your host machine)
* [.mtoolsrc](https://elearn.tntech.edu/d2l/common/dialogs/quickLink/quickLink.d2l?ou=8947521&type=coursefile&fileId=.mtoolsrc) (Save to the coursework directory on the VM)
* [boot1.asm](https://elearn.tntech.edu/d2l/common/dialogs/quickLink/quickLink.d2l?ou=8947521&type=coursefile&fileId=boot1.asm) (Save to you project directory on the VM)

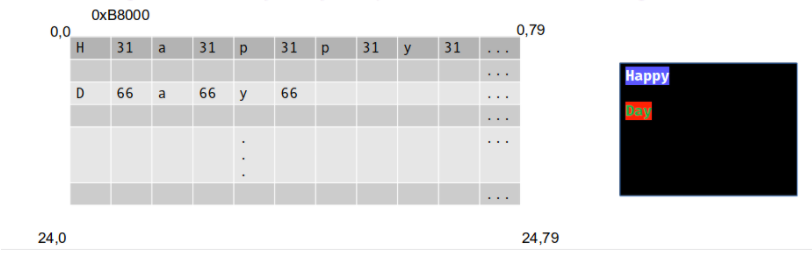
**Description  
  
  
Introduction**

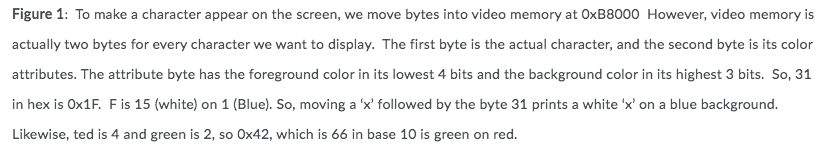
You will build a stage 1 boot loader and a stage 2 operating system (OS). This initial boot loader has already been written; You simply need to download it, compile it, and install it. It will boot from a (virtual) floppy and put the computer into protected mode. It then will load the seconds stage OS. The second stage OS will clear the screen, draw a border around the screen, and the print a message in the middle of the screen.  The message will be "OS 0000".

**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.

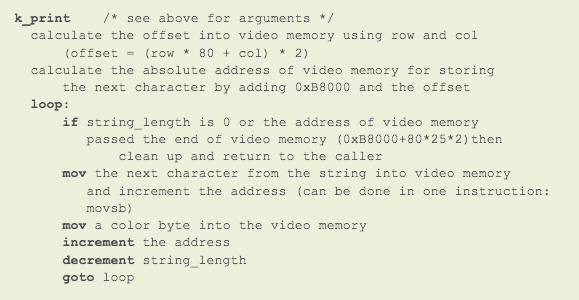
* **Bximage**. This utility will build a virtual floppy disk (a file called a.img) for you.
* dosfstools.  Specifically, the **mkdosfs** program will format a virtual floppy.
* The **dd**copy utility. You will use this utility to copy your stage 1 boot loader to the virtual floppy.
* The mtools suite 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.
* 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.  You need VirtualBox only if you are not booting Debian Linux natively.  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 is 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](https://elearn.tntech.edu/d2l/common/dialogs/quickLink/quickLink.d2l?ou=8947521&type=coursefile&fileId=boot1.asm).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 followed by a message in the middle of the screen.  To get its work done, the main  will call two functions written in assembly, called k\_print() and print\_border(), to get its work done. The functions have the following prototypes:  
 **void k\_clearscr();  
void k\_print(char \*string, int string\_length, int row, int col);  
void print\_border(int start\_row, int start\_col, int end\_row, int end\_col);**  
The k\_print() 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\_print() to get their work done. Note that you will need to understand how parameters are passed from C to an assembly language function.   
  
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.





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\_print() would be the following:



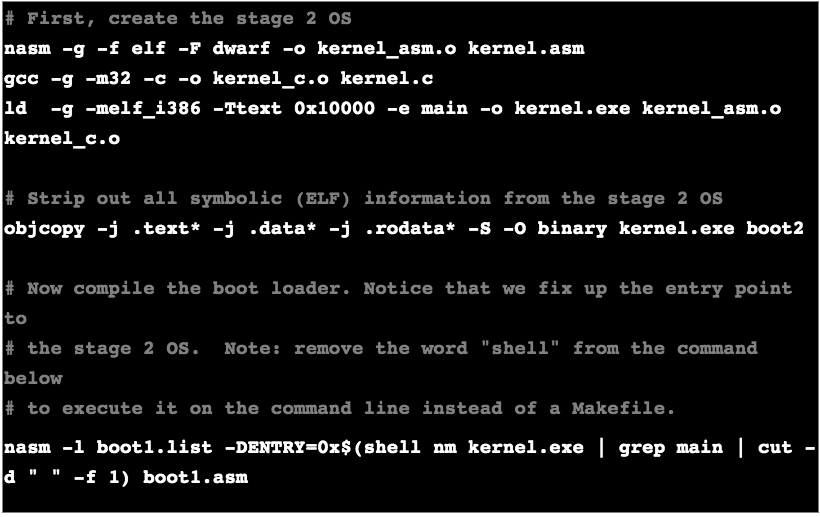
k\_clearscr() is really easy.  You just right a space plus a color byte into video memory to clear a particular character on the screen.  Do so for the whole screen, and you have cleared the screen (in other words, start at 0xB8000, write a ' ' byte followed by a 31, then increment the address by two and do it again.  In fact, do so 80\*25 times, and you have cleared the screen.)  Note that an easy way to implement this function is to call k\_print() twenty five times passing it a string of spaces 80 characters long.  
  
Programming the print\_border() function is not so difficult.   Call the k\_print() function to help your create your borders. You need to write strings of the border characters to the top, bottom, left and right of the screen.  Also, you need to connect the corners.  Use dashes for the top and bottom, the pipe symbol (vertical bar) for the sides, and the plus symbol for the corners.  Consider the following example box:

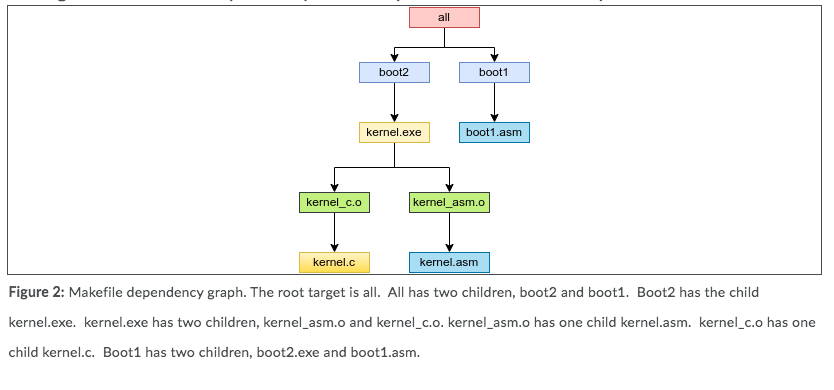


Finally, you will write the main().  Your main() will simply call k\_clearscr() followed by print\_border(), followed by a call the k\_print() to print the message to the middle of the screen

You 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):

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.



*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).

**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 bximage utility found in /usr/local/bochs/bximage as follows:  
  
**bximage -mode=create -fd=1.44M  -q a.img**

After you create the image, you will need to format it using the mformat command. Download this configuration file at [mtoolsrc.conf](https://elearn.tntech.edu/d2l/common/dialogs/quickLink/quickLink.d2l?ou=8947521&type=coursefile&fileId=mtoolsrc.conf" \t "_self) and save it as .mtoolsrc in your home directory. Then execute the following command to format the virtual floppy:  
  
**/sbin/mkdosfs a.img**  
  
The following command will install the stage 1 boot loader:  
  
**dd if=boot1 of=a.img bs=1 count=512 conv=notrunc**  
  
Finally, copy the stage 2 OS to the virtual floppy with 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 -boot a -fda a.img**

**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 -boot a -fda a.img**  
  
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:  
  
**gdb kernel.exe**   
  
Next, At the gdb prompt, enter the following:  
  
**target remote localhost:1234**  
  
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.