Homework 1

W4118 Spring 2021
UPDATED: 1/17/21 at 2:18pm EST
DUE: 1/25/21 @ 11:59PM

All homework submissions are to be made via Git. You must submit a detailed list of references as part your homework submission indicating clearly what sources you referenced for each homework problem. You do not need to cite the course textbooks and instructional staff. All other sources must be cited. Please edit and include this file in the top-level directory of your homework submission. Homeworks submitted without this file will not be graded.

You will be using Git via GitHub for course submissions for the class. Please make sure you sign up for a GitHub account if you do not yet have one, and follow the instructions for the W4118 GitHub organization, including filling out the Google Form listed there so that we can associate your GitHub username with your Columbia UNI. Once you have a GitHub account and login, you can create your GitHub repository for this assignment. The GitHub repository you will use can be cloned using `git clone git@github.com:W4118/s21-hmwk1-UserName.git` (Replace UserName with your own GitHub username). **Be aware that commits pushed after the deadline will not be considered.** Refer to the homework policy section on the class web site for further details.

Non-Programming Problems:

Exercise numbers refer to the course textbook, *Operating System Concepts Essentials*. Each problem is worth 5 points. Please include all your answers in a file named written.txt that is uploaded to your GitHub repository for this assignment.

1. Exercise 1.13
2. Exercise 1.19
3. Exercise 1.21
4. Exercise 2.13
5. Exercise 2.21

Programming Problems:

For all programming problems you will be required to submit source code, a Makefile, a README file documenting your files and code, and a test run of your programs. The README should explain any way in which your solution differs from what was assigned, and any assumptions you made. For this assignment, you will have a separate subdirectory for each part of the assignment, and each subdirectory should contain its own Makefile and source code. While we will provide a Makefile for you for Part 2, you should provide your own Makefile for Parts 1 and 3. In addition, you should submit your floppy image for Part 3. The README should be placed in the top level directory of your GitHub repository for this assignment. Refer to the homework submission page on the class web site for additional submission instructions.

Simple Shell and OS (75 Pts.)

Part 1: The Simple Shell

An operating system like Linux makes it easy to run programs from a shell. The shell is just another program. For example, the Bash shell is an executable named `bash` that is usually located in the /bin directory. So, /bin/bash.

Try running `/bin/bash` or just `bash` on a Linux (or BSD-based, such as Mac OS X) operating system's command line, and you'll likely discover that it will successfully run just like any other program. Type `exit` to end your shell session and return to your usual shell. (If your system doesn't have Bash, try running `sh` instead.) When you log into a computer, this is essentially what happens: Bash is executed. The only special thing about logging in is that a special entry in `/etc/passwd` determines what shell runs at log in time.

Write a simple shell in C. The requirements are as follows.

1. **Your shell executable should be named w4118_sh.** Your shell source code should be mainly in shell.c, but you are free to add additional source code files as long as your Makefile works, and compiles and generates an executable named w4118_sh in the same top level directory as the Makefile. If we cannot simply run make and then w4118_sh, you will be heavily penalized.
2. **The shell should run continuously, and display a prompt when waiting for input.** The prompt should be EXACTLY '$'. No spaces, no extra characters. Example with a command:

```
        $/bin/ls -lha /home/w4118/my_docs
```

3. **Your shell should read a line from stdin one at a time.** This line should be parsed out into a *command* and *all its arguments*. In other words, tokenize it.
    - You may assume that the only supported delimiter is the whitespace character (ASCII character number 32).
    - You do not need to handle "special" characters. Do not worry about handling quotation marks, backslashes, and tab characters. This means your shell will be unable support arguments with spaces in them. For example, your shell will not support file paths with spaces in them.
    - You may set a reasonable maximum on the number of command line arguments, but your shell should handle input lines of any length.
4. **After parsing and lexing the command, your shell should execute it.** A command can either be a reference to an executable OR a built-in shell command (see below). For now, just focus on running executables, and not

on built-in commands.

- Executing commands that are not shell built-ins is done by invoking `fork()` and then invoking `exec()`.
- You may **NOT** use the `system()` function, as it just invokes the `/bin/sh` shell to do all the work.

5. **Implement Built-in Commands, `exit` and `cd`.** `exit` simply exits your shell after performing any necessary clean up. `cd [dir]`, short for "change directory", changes the current working directory of your shell. Do not worry about implementing the command line options that the real cd command has in Bash. Just implement cd such that it takes a single command line parameter: the directory to change to.
6. **Error messages should be printed using exactly one of two string formats.** The first format is for errors where `errno` is set. The second format is for when `errno` is not set, in which case you may provide any error text message you like on a single line.

   ```
   "error: %s\n", strerror(errno)

   OR

   "error: %s\n", "your error message"
   ```

   So for example, you would likely use: `fprintf(stderr, "error: %s\n", strerror(errno));`
7. **Check the return values of all functions utilizing system resources.** Do not blithely assume all requests for memory will succeed and all writes to a file will occur correctly. Your code should handle errors properly. Many failed function calls should not be fatal to a program.

   Typically, a system call will return -1 in the case of an error (malloc will return NULL). If a function call sets the errno variable (see the function's man page to find out if it does), you should use the first error message as described above. As far as system calls are concerned, you will want to use one of the mechanisms described in Reporting Errors or Error Reporting.
8. **A testing script skeleton is provided in a GitHub repository to help you with testing your program.** You should make sure your program works correct with this script. For grading purposes, we will conduct much more extensive testing than what is provided with the testing skeleton, so you should make sure to write additional test cases yourself to test your code.

Part 2: Simple OS and Simple Program

Without an operating system, running a program on a computer will be harder. The modern operating system is generally loaded by a bootloader. The bootloader sets up the basic running environment for the operating system, loads the operating system into the memory and finally transfers control to the operating system.

`GRUB` (GRand Uefi Bootloader) is the most commonly used bootloader for operating systems using the Linux kernel. It also supports booting other operating system that follow its boot protocol. GRUB loads operating system into the RAM at offset 1MB and starts executing the operating system at that position.

Write a simple operating system that can be loaded by GRUB and can in turn run a simple program to display output on the screen. The requirements are as follows.

1. **Build and install the vanilla Simple OS.**. We provide the starter assembly code `entry.S`. The starter code sets up the header for the operating system so that the GRUB can recoginize and load it. To build the Simple OS, you can simply use the provide Makefile and do

   ```
   make && sudo make install
   ```

   and the Simple OS will be installed to the boot parition of the VM. You can then reboot the VM and select the `os` option from GRUB. Currently, the Simple OS does nothing and will crash. To prevent it from crashing, you should add an infinite loop to the code so that it will loop forever and show a blank screen:

   ```
   loop:
        jmp loop
   ```

   For this part, a Makefile and linker script `link.ld` have already been provided for you; you do not need to modify them.

2. **Call C functions from assembly**. Low-level assembly language is not easy to use and read. So you should modify `entry.S` to switch your Simple OS from assembly to C. A stack is needed to switch from assembly code to C. The starter code has reserved a chunk of memory (4KB) for the stack. To swtich the stack, you need to add a line of code to your `entry.S`:

   ```
   mov $stack, %esp
   ```

   Then you can call the `main` function defined in `main.c`:

   ```
   call main
   ```

   Note that this main function is not a standard C main function for general C programs so it does not have to return an integer. The name of the function does not even have to be `main`. Since you are writing an operating system, when the main function returns, it returns to where it was called. You should ensure that when the function returns, it does not let the machine run into a state that is out of your control. Note that your C code is compiled with various options indicating that it is standalone and does not rely on the standard C library or standard include files which are not available with your simple OS.

3. **Write a simple VGA console driver**. Modify `main.c` by writing C code to output to the VGA console so you can print any characters supported by ascii at any location of the screen. The function should also be able to control the color of the character. The framebuffer of the VGA console is mapped to the RAM starting from `0xb8000`. In text mode, you can print out characters by writing ascii codes starting at that position. Each character takes two bytes - one byte for the ascii code and the other for the color. Therefore, to print "hello, world", you should write 'h' to `0xb8000`, 'e' to `0xb8002`, etc. Note that GRUB switches the CPU to 32-bit mode and leaves the CPU in real mode so you are able to access the RAM locations for the framebuffer directly in C.
4. **Control the color of the character**. The memory for each character on the VGA console is two bytes. You can modify the second byte to change the color. The color byte can be used to control both the foreground and background color of the character, with the higher 4 bits for the background and the lower 4 bits for the foreground:

```
Bit:     | 7 6 5 4 | 3 2 1 0 |
Content: | BG      | FG      |
```

and the color is defined as:

Color	Value	Color	Value	Color	Value	Color	Value
Black	0	Red	4	Dark grey	8	Light red	12
Blue	1	Magenta	5	Light blue	9	Light magenta	13
Green	2	Brown	6	Light green	10	Light brown	14
Cyan	3	Light grey	7	Light cyan	11	White	15

5. **Print out "hello, world" with white background color and black foreground color at the center of the console.** The default VGA console can print 80*25 characters. You should print "hello, world" with the specified color and position it so it is aligned to the center of the console vertically and horizontally, *all lower case with the comma after the first word and a space after the comma before the second word*.

Part 3: Bare-metal Hello World OS

Modern computers have several bootstrap stages. We have already seen how a bootloader loads an operating system. However, this is not the very first step of the bootstrap. When the power button is pressed, the CPU is reset to its initially state and firmware, the BIOS, is executed. The BIOS checks the hardware resources of the computer, loads the first progrom on the stroage device, for example the hard drive, into the RAM and transfers control to the program.

The MBR (Master Boot Record) is first sector (512B) of the persistent storage device which ends with magic number `0x55` and `0xaa` at the last two bytes. It holds the initial bootstrap code and all of the necessary information to boot the machine into the operating system. The BIOS reads the first sector of the hard drive and if the sector is an MBR, the BIOS loads it into the RAM at `0x7c00` and starts executing from that position.

Usually, the bootloader is located at the MBR sector and loads the operating system, which can implement more complex functionality. But it is not necessary. A simple operating system can also be loaded directly by the BIOS.

1. **Build the Hello World OS.** Modify the file `bare_hello.S` by writing GNU assembly code to print "hello, world" at the beginning of the VGA console. Unlike the previous Part 2 of the assignment, you should not rely on GRUB. Without GRUB, the machine is in 16-bit real mode, so your code should run in 16-bit mode. For simplicity, all of your code should be in x86 assembly and fit in the MBR itself. You will also not able to directly access the framebuffer because the starting address for the framebuffer is beyond the addressing limit (ie 0xb8000 > 0xffff). Instead, you will need to segmentation, which requires you to load the segment base of the starting address (ie 0xb800) into a data segment; the move instruction `mov` should be all that you need. Then you will use assembly instead of C to write the characters to the VGA console; the move byte instruction `movb` will be useful. Note that you will need a while loop at the end of your code to avoid the machine going off into undefined behavior; use the jump instruction `jmp`.
2. **Create a floppy disk image that holds the Hello World OS and boot it on your VM.** To start the computer from power on without requiring a bootloader, you need to create a new storage device to hold your Hello World OS such that its first sector is an MBR. For this purpose, you can create a floppy disk image such that the last two bytes of the first sector contain the required magic number `0x55`. To do this, you will build the object file `bare_hello.o`, copy the text section of the object file into a file (ie `floppy.flp`) that will be your floppy image, then edit the binary floppy image to add the magic number to the 511th and 512th bytes of the image.

   ```
   gcc -c -m16 bare_hello.S -o bare_hello.o
   objcopy -O binary -j .text bare_hello.o floppy.flp
   dd if=/dev/zero bs=512 count=1 >> floppy.flp
   ```

   will create the floppy image `floppy.flp`, padded with zero bytes at the end beyond the 512th byte. Then, use `vim` to open floppy.flp and edit it in hexdump mode by using the `vim` command

   ```
   :%!xxd
   ```

   to transform the binary into hex dump using `xxd`. Counting the first byte as offset 0, you should change the two bytes located at offset 510B and 511B to `0x55` and 0xaa, respectively, and use

   ```
   :%!xxd -r > floppy.flp
   :q!
   ```

   to save your changes. Copy the floppy image from your VM to your host computer.
3. **Use your floppy image to boot the VM.** Shutdown your VM and open settings of the VM. Add a new floppy drive then select the floppy image you just copied from your VM. Then change the startup device to the floppy

drive. You should be able to boot your VM with the Hello World OS you just build.

Additional Requirements

1. *All code (including test programs) must be written in C.*
2. *Make at least ten commits with Git. The point is to make incremental changes and use an iterative development cycle.*
3. *Follow the following coding style rules:*
   - *Tab size: 8 spaces.*
   - *Do not have more that 3 levels of indentations (unless the function is extremely simple).*
   - *Do not have lines that goes after the 80th column (with rare exceptions).*
   - *Do not comment your code to say obvious things. Use /* ... */ and not // ...*
   - *Follow the Linux kernel coding style. Use checkpatch, a script which checks coding style rules.*
4. *Use a makefile to control the compilation of your code. The makefile should have at least a default target that builds all assigned programs.*
5. *When using `gcc` to compile your code, use the `-Wall` switch to ensure that all warnings are displayed.* ***Do not*** *be satisfied with code that merely compiles; it should compile with no warnings. You will lose points if your code produces warnings when compiled.*
6. *Check the return values of all functions utilizing system resources for all parts of the programming assignment.*
7. *Your code should not have memory leaks and should handle errors gracefully.*
8. *Per the Class Collaboration/Copying Policy, please include in your submission a separate references.txt file with a list of references to materials that you used to complete your assignment, including URLs to websites and names of other students you asked for help.*

Tips

1. *For this assignment, your primary reference will be Programming in C. You might also find the Glibc Manual useful.*
2. *Many questions about functions and system behaviour can be found in the system manual pages; type in `man` function to get more information about function. If function is a system call, `man 2` function can ensure that you receive the correct man page, rather than one for a system utility of the same name.*
3. *If you are having trouble with basic Unix/Linux behavior, you might want to check out the resources section of the class webpage.*
4. *A lot of your problems have happened to other people. If you have a strange error message, you might want to try searching for the message on Google.*