Homework 1

W4118 Fall 2023
UPDATED: 9/14/23 at 9:36am EST
DUE: 9/20/23 @ 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.**

Before You Begin Programming

Before you begin the programming for this assignment, you must first do two things: get access to GitHub and setup a VMware virtual machine (VM) that you will use for your development work.

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. If you do not complete the form prior to the homework deadline, you will receive an automatic zero for the assignment.

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/f23-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.

You will be using a VM that you will setup for all homework assignments. Follow the instructions provided to setup a VM using the Debian Linux distribution we will use for this class. Using a VM ensures that you have a consistent development and testing environment with what we will be using to grade your assignments. For subsequent homework assignments, using a VM for testing will also ensure that a kernel you boot does not mess up your actual computer.

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. **You must provide a Makefile for each part of this assignment.** 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. In addition, please pay attention to the additional requirements listed at the bottom of this assignment.

Simple Shell and OS

Part 1: The Simple Shell

An operating system like Linux makes it easy to run programs. For example, from a shell, it is easy to write, compile, and run a simple hello world C program:

```
$vi hello.c
#include <stdio.h>
int main() { printf("hello, world\n"); }
$gcc hello.c -o hello
$./hello
hello, world
```

The operating system makes this easy by providing various functions to enable the program to perform I/O such as printing, and the shell to execute the program in response to typing the program executable name at the shell prompt. The shell itself 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. You may find `getline()` useful.
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. cd should be done by invoking `chdir()`.
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 Shell Directly Calling System Calls

The simple shell you wrote in Part 1 calls various system calls such as `fork()`, but also relies on various C library functions that in turn call other system calls. You can use `strace` to see what system calls are being called when you run simple shell. First, install strace:

```
sudo apt install strace
```

Then you can run `strace` with simple shell:

```
strace –o trace.txt w4118_sh
```

which will dump the system calls executed into the file trace.txt. For example, if you used `printf()` to output text in simple shell, you will find that it in turn calls a system call to actually perform the I/O operation because I/O is controlled by the operating system. C library functions such as `printf()` are technically not part of the C language, but made possible by relying on functionality provided by the operating system.

To gain a better understanding of how C library functions rely on operating system functionality, modify your simple shell so that it does not call any C library functions that call other system calls. Instead, your simple shell should directly call any system calls that it implicitly uses. For example, your simple shell should not call `printf()` but instead call `write()` on STDOUT. Other C library functions that you may also have to replace include `getline()`, `malloc()`, etc. You do not have to be overly concerned with efficiency, so you may find it easier to use `mmap()` instead of `sbrk()` for any dynamic memory allocation you need to do. For example, you may find it helpful to see this implementation of `malloc()`. String manipulation functions such as `strtok` and `strcmp` do not call system calls and do not need to be replaced.

Note that your implementation of the various functions only has to work specifically for your simple shell. For example, you do not need to implement all functionality supported by `printf()`, only what functionality is required to print the output that your shell generates. Similarly, your input functionality only needs to work for any ascii characters generated from a keyboard.

Your shell executable should be named w4118_sh2. Your shell source code should be mainly in shell2.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_sh2 in the same top level directory as the Makefile. If we cannot simply run make and then w4118_sh2, you will be heavily penalized. w4118_sh2 should have all the same functionality as w4118_sh, except that it does not call any C library functions that call other system calls.

Part 3: Bare-metal Hello World OS

Without an operating system, running a program on a computer will be harder. When the power button is pressed, the CPU is reset to its initial state and firmware is executed. The firmware checks the hardware resources of the computer, loads the first program on the storage device (for example, the hard drive) into the RAM and transfers control to the program.

In older systems, the BIOS (Basic Input Output System) is the firmware that checks hardware, then transfers control over to the first program. Newer systems now use a different firmware standard known as UEFI (Unified Extensible Firmware Interface), which provides small abstractions on top of the hardware to make it easier for operating system and driver developers to work with different kinds of hardware. UEFI looks for a program at an architecture-specific path on the storage device, and runs it on boot. For x86_64 machines, this first program is located at `/EFI/BOOT/BOOTX64.EFI`, whereas on aarch64 machines this program is located at `/EFI/BOOT/BOOTAA64.EFI`. Unlike Linux filesystems, in UEFI these initial filesystem paths are *case insensitive*.

Usually, the bootloader is located at these architecture-specific paths, and when run, 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 UEFI. The simple operating system can be as simple as a program that prints "hello, world" to the screen. However, the program that is loaded by UEFI does not, at least initially, have an easy-to-use C environment in which to execute.

1. **Implement the Hello World OS.** You will be implementing a UEFI-enabled operating system that prints **hello, world** to the console. We provide the starter C code `main.c`, which you will be modifying for your implementation. We also provide `part3_include`, which contains headers and other files necessary for your program to understand the UEFI firmware environment. A pixel font is also provided under the same folder as `font8x8_basic.h`. **The provided files under `part3_include` should NOT be modified. Add your code to `main.c` under the comment that says `WRITE YOUR CODE BELOW THIS LINE`.** The provided files are not part of the Linux kernel project, and therefore do not conform with checkpatch standards; we have structured the repository so that you can still run checkpatch on your `part3` folder without seeing style errors produced by the UEFI and font headers.

   The starter code is a UEFI entrypoint that sets up the UEFI Graphics Output Protocol abstraction (also known as GOP) and asks the UEFI firmware to set up pages of physical memory to be used as a pixel buffer. Note that this entrypoint is not a typical C program with a C main function. Additionally, your program

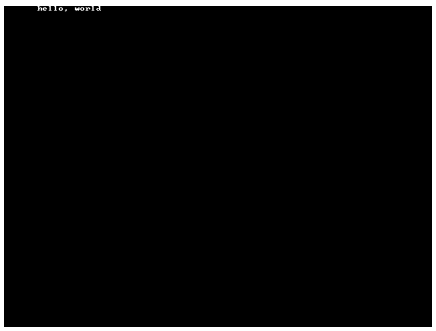
will be compiled in a specific way that makes it standalone, without the standard C library or standard include files which are not available with your simple operating system. For example, there are no functions available for dynamic memory allocation.

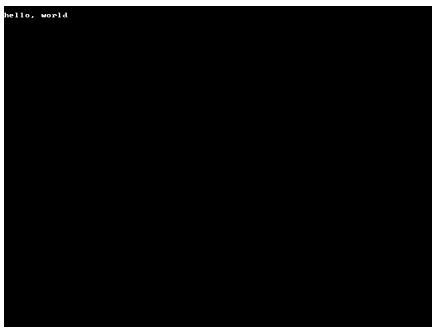
Your hello world OS implementation will use GOP to write pixels onto the screen, to form the words **hello, world**. For your convenience, pages of physical memory are provided to store pixel data for each pixel on the screen, with the starting address of this memory range stored as the variable `PIXEL_BUFFER_BASE`. Two additional variables `HEIGHT` and `WIDTH` represent the screen's pixel dimensions, which may be useful in your implementation.

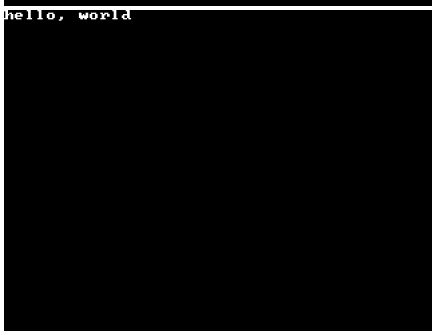
Since you will be working with pixels instead of text characters, you will need to draw characters to screen with pixels. A font is available in the provided `font8x8_basic.h`, which shows the coordinates on an 8x8 pixel grid that are "on" and "off" to properly draw a character. The fonts are taken from dhepper/font8x8; its README may be useful to understand how these fonts are encoded. When using this font, characters should be drawn with white, with a black background. **Pixels must be drawn as starting at the upper left corner of the screen. Each character must be of 8x8 size, with no extra space between pixels than what the font prescribes.**

An example of the correct text, font, and positioning (right click and open in a new tab or download for larger images):

The following are incorrect:







Note: The above examples were rendered on a 640x480 resolution display. Your VM may use a different resolution, which is ok. Do not upscale your text to your resolution - it is ok if you have more or less blank black space than the example, but the white pixels must match.

Pixels in GOP are represented by 32-bit unsigned RBG values. The struct representation of these pixels is reproduced below:

```
typedef struct {
  UINT8 Blue;
  UINT8 Green;
  UINT8 Red;
  UINT8 Reserved;
} EFI_GRAPHICS_OUTPUT_BLT_PIXEL;
```

Pixels in physical memory are not automatically drawn to screen. We have provided a call to GOP's `Blt` function, which will copy pixels to video hardware. The UEFI specification on GOP may be a useful reference to understand how this function works.

Once you have finished drawing pixels to the screen, you need to prevent your program from exiting. In particular, the provided `main.c` does nothing further after `efi_main` exits, and control is transferred back to the UEFI firmware. As a result, UEFI will clear the screen and try to boot from elsewhere in the system, or open the UEFI diagnostics screen. You should ensure that your hello world OS does not let the machine do this and keep the pixels you have drawn. A loop of some kind here would be useful.

A good resource for understanding the provided UEFI setup code, as well as other graphics manipulation functions, is the UEFI specification document. The detailed reader may note that UEFI provides other convenience functions to print 16-bit character strings to the console, using the Simple Text Output Protocol. It is optional, but you may find these functions to be useful while debugging; however, **you may not use these functions in your final submission.** Only the functions under GOP should be used.

2. **Compile and create a disk image that holds the Hello World OS.** 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 it contains your UEFI program in the proper file path.

   You will need to install additional tools to build your UEFI OS:

   ```
   sudo apt install clang lld mtools
   ```

   x86 VM: If you are using an x86 VM, the commands you need to execute are:

   ```
   # compilation
   clang -target x86_64-unknown-windows -ffreestanding -fshort-wchar -mno-red-zone -c main.c
   clang -target x86_64-unknown-windows -nostdlib -Wl,-entry:efi_main -Wl,-subsystem:efi_application -fuse-ld=lld-link -o BOOTX64.EFI

   # disk image creation
   dd if=/dev/zero of=disk.img bs=1k count=1440
   mformat -i disk.img -f 1440 ::
   mmd -i disk.img ::/EFI
   mmd -i disk.img ::/EFI/BOOT

   # copy program into the disk
   mcopy -i disk.img BOOTX64.EFI ::/EFI/BOOT
   ```

   Arm VM: If you are using an Arm VM, the commands you need to execute are:

   ```
   # compilation
   clang -target aarch64-unknown-windows -ffreestanding -fshort-wchar -mno-red-zone -c main.c
   clang -target aarch64-unknown-windows -nostdlib -Wl,-entry:efi_main -Wl,-subsystem:efi_application -fuse-ld=lld-link -o BOOTAA64.EF

   # disk image creation
   dd if=/dev/zero of=disk.img bs=1k count=1440
   mformat -i disk.img -f 1440 ::
   mmd -i disk.img ::/EFI
   mmd -i disk.img ::/EFI/BOOT

   # copy program into the disk
   mcopy -i disk.img BOOTAA64.EFI ::/EFI/BOOT
   ```

   Note that the instructions are nearly identical for x86_64 and aarch64 (Arm), except for the target architecture and the file name for the UEFI program. Compiling against a Windows target is required, since UEFI uses the Windows Portable Executable file format.

   A Makefile has been provided to help make this compilation and image creation process easier. The Makefile will attempt to detect your machine's architecture, which you can override by running `make ARCH=aarch64` or `make ARCH=x86_64`.

3. **Use your disk image to boot the VM.** Copy the disk image from the previous step onto your host. Create a new VM in VMware, using the created disk image as the CD drive.

   For x86_64 VMs: configure your new VM by going to "Settings", then under "Options" and "Advanced" specify UEFI as the Firmware type. Note that this step is required for x86 VMs, not Arm VMs, because VMware uses BIOS by default for x86 VMs but UEFI for Arm VMs.

   You should be able to boot your VM with the Hello World OS you just built. Note that if you see a UEFI shell, or a diagnostics menu appears, then your hello world OS program has exited or was not found by UEFI.

4. **Submission.** As part of your submission, you should ensure that a Makefile is included under the part3 directory to compile your code and create a bootable disk image (see above for disk creation instructions, or refer to the provided Makefile). The bootable disk image must be named disk.img with the EFI program in the correct filesystem location. By default, we will run `make ARCH=x86_64` and expect that a disk.img is created for an x86_64 host.

   For students on Arm Mac computers (e.g. with M1 or M2 CPU): if you want your part3 submission to be built/tested for Arm, you must create and submit a file called .armpls in the part3 directory of your repo; feel free to use the following one-liner:

   ```
   cd "$(git rev-parse --show-toplevel)/part3" && touch .armpls && git add .armpls && git commit -m "Arm pls"
   ```

   We will then run `make ARCH=aarch64` and boot your program on an aarch64 host.

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.