CSE220 Fall 2014 - Homework 10

Due Sunday 12/07/2014 @ 11:59pm

In this assignment we will be coming full circle with C and MIPS. You will be given a special version of gcc that will compile your C code on an x86 machine into assembly for a MIPS machine.

This version of gcc is *"naked"*. What this means is that you only have available what the ANSI C language provides. There is no glibc, or other libraries that you have used in your previous assignments (stdio.h, ctype.h, stdlib.h, string.h, etc). If you want them, you will have to make these functions again yourself.

As we are compiling for MIPS/Mars, there is no system call interface for communicating between C and the OS.

Your first task will be to create the system call interface between C and Mars. To do this you will have to write functions in MIPS. You will then expose these MIPS functions to your C programs by creating header files which contain the C prototypes. Once you have created these interfaces you will be able to use these system calls in your C programs by calling the C functions.

The second task is to then recreate some of the basic c library functions (eg. printf, fgets, and some of the string library functions such as strlen).

Since the function `printf` is a variadic function, you will have to understand how gcc builds the activation record for the MIPS assembly that implements it. It is imperative that you understand this process so you can figure out how to access these variable arguments in your C code.

Finally you will use your basic functions to create a simple *"shell"* to run a few simple built-in commands and a few existing programs from disk. This interface should be able to parse commands entered by the user from the keyboard. You will need to write a combination of C code and MIPS code to finish this task.

> If you wrote your functions from HW8 correctly you can reuse some of this code. Note that any code that depends on any of the standard glibc headers will have to be changed.
>
> The program that translates the MIPSEL COFF binaries into Mars assembly cannot handle IEEE type variables and instructions. Because of this we **cannot write code that uses floating point instructions** and expect it to translate back to Mars MIPS code.

Getting Started

> ⚠ Refer to Sparky slides for basic Unix commands.

In this assignment we will be doing things a bit differently. The C to MIPS cross compiler is setup on an Ubuntu Linux machine, not sparky. So instead of using ssh to connect to Sparky, you will instead have to connect to `130.245.12.103` on port 130.

The username to log into this account is your netid.

```
$ ssh netid@130.245.12.103 -p 130
```

If your sbuid is `123456789` then your temporary default password to log in is `Sbcs123456789`.

After logging in you will be forced to change your password.

```
WARNING: Your password has expired.
You must change your password now and login again!
Changing password for cse220.
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
Connection to 130.245.12.103 closed.
```

After changing your password you will be logged out. Log in again with your new password.

If you want to change your password to something else after this initial login you can do so with the following command:

```
$ passwd
Changing password for user.
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
$
```

After handling your password, the next thing you should do is examine the contents of your home

directory.

```
$ ls
hello.c  include  ld_script  libsrc  Makefile  Makefile.common
Makefile.compile  Makefile.lib try.c
$
```

The include directory contains the file `lib.h`. This file contains the function prototypes for our system calls and all other functions we want to add to our version of libc (basic libraries).

```
$ ls include
lib.h
$
```

lib.h

```
/*
 * Function prototypes for C library functions.
 * This header file should be included in any C file
 * that needs to use any of these functions.
 */
void exit(void);
void putchar(char c);
```

The libsrc directory currently contains all the assembly stubs used to wrap the Mars system calls and make them available to C. This directory will also eventually contain your source code for your implementation of other libc functions.

```
$ ls libsrc
crt0.s  exit.s putchar.s
$
```

> ℹ Notice here how the MIPS assembly files end in `.s` instead of `.asm`. This is because the gnu assembler expects files to have the .s extension.

The `crt0.s` file just has some basic setup that is required for every program before we call the main function in your C file. You shouldn't modify this program but it wouldn't hurt to take a quick look at it. The file `exit.s` is an example of wrapping the Mars system calls and making them available to be called in your C program. `putchar.s` is another example of exposing a system call that takes an

argument.

Back in your home directory there are a few other files that you may be unfamiliar with. The `ld_script` is the linker script used by gcc to set all the sections (data, text, …) correctly when your code is assembled. The files `Makefile`, `Makefile.common`, `Makefile.compile`, and `Makefile.lib` are used to build your project with the simple command make.

Let's test this all out by building `hello.asm` from the file `hello.c` which was provided in your home directory. Type `make hello.asm` in your home directory to compile the file `hello.c` into a MIPS/Mars `.asm` file.

```
$ cd ~/
$ make hello.asm
make -f Makefile.lib crt0.o
make[1]: Entering directory `/home/cse220'
/usr/local/bin/mips-cc/as -G 0 -mips1 libsrc/crt0.s -o crt0.o
make[1]: Leaving directory `/home/cse220'
make -f Makefile.lib libc.a
make[1]: Entering directory `/home/cse220'
/usr/local/bin/mips-cc/as -G 0 -mips1 libsrc/exit.s -o exit.o
/usr/local/bin/mips-cc/as -G 0 -mips1 libsrc/putchar.s -o putchar.o
/usr/local/bin/mips-cc/ar -r libc.a exit.o putchar.o
/usr/local/bin/mips-cc/ar: creating libc.a
rm -f exit.o putchar.o
/usr/local/bin/mips-cc/ranlib libc.a
make[1]: Leaving directory `/home/cse220'
/usr/local/bin/mips-cc/gcc -G 0 -B/usr/local/bin/mips-cc/ -Iinclude -c -fno-
builtin -nostdinc -S hello.c
/usr/local/bin/mips-cc/as -G 0 -mips1 hello.s -o hello.o
/usr/local/bin/mips-cc/ld -T ld_script -N crt0.o hello.o -o hello.coff
libc.a
'/usr/bin/'java -jar /usr/local/bin/coff2asm/Coff2Asm.jar hello.coff
hello.asm
Loading 3 sections:
    ".text", filepos 0xd0, mempos 0x4000000, size 0xb0
    ".data", filepos 0x180, mempos 0x10010000, size 0x20
    ".bss", filepos 0x0, mempos 0x10010020, size 0x10
rm hello.coff hello.o hello.s
$
```

If this operation was successful you should now have the file `hello.asm` which contains the MIPS assembly instructions for executing the program `hello.c`. This file can now be run with Mars.

```
$ ls
```

```
crt0.o  hello.asm  hello.c  include  ld_script  libc.a  libsrc  Makefile
Makefile.common  Makefile.compile  Makefile.lib try.c
$
```

Let's test out this file with the command line version of Mars.

> ℹ There is an alias for mars in your `.bashrc` file located in your home directory. `alias mars='java -jar /usr/local/bin/mars/Mars220.jar sm db nc smc'`.

To run Mars type:

```
$ mars hello.asm
Hello, World!

$
```

This will be your workflow for this project. You must do all the compilation and testing on this server.

> ℹ You can code locally on your machine, but you must build your files on this server. You may want to look into technologies such as sshfs to ease this process.
>
> ℹ To delete all the files associated with your compilation use the command `make clean`

Part 1 - Building the basic system calls

In the libsrc directory we already have `crt0.s`, `exit.s`, and `putchar.s`. You should examine the files `exit.s` and `putchar.s` to understand what they are doing and then implement the following system calls each in their own file:

- print integer - putint.s
  - syscall 1
- open file - open.s
  - syscall 13
- close file - close.s

    - syscall 16

- read file - read.s
    - syscall 14

- print hex - puthex.s
    - syscall 34

- print binary - putbinary.s
    - syscall 35

- print unsigned - putunsigned.s
    - syscall 36

> Mars system calls reference

You will need to add each of the C prototypes of these functions to the `lib.h` file. After adding the prototypes to `lib.h` you will need to edit the file `Makefile.lib` to add commands to build these functions. Open up `Makefile.lib` and look for the line that starts with `LIBOBJS`. If you examine this line you will see that it currently includes `putchar.o` and `exit.o`. You will need to add the new system call files that you just created to this list.

- putint.o
- open.o
- close.o
- read.o
- puthex.o
- putbinary.o
- putunsigned.o

> To test each of these functions you should modify `hello.c` to use these functions. Build `hello.asm`, then run `hello.asm` with Mars

Part 2 - Adding back some libc functionality

Up to this point in the class we have been unable to let you define your own header files. Luckily for you that is about to change! But before we let you go off and create your own files, we need to tell you about a few subtle issues.

If you looked at the header files we gave you they usually start with some weird preprocessor magic. You may be asking yourself "What is this for?". This preprocessor check is a way to prevent the same header file being included over and over again. If we didn't do this, the header will be included (copied into the source file) multiple times resulting in your program becoming very large. If the contents of the header file are added over and over again during the compilation process, and your project is big enough, it will slow down the compilation process. Additionally, you will have multiple definition errors for the functions, as well as possibly creating cyclic dependencies (where there are two files that both include each other).

So now that we got that cleared up let's introduce the basic process to creating this header guard.

Let's assume we have the header file `hw10.h`. To define a header guard in this file you should do the following.

- Start with adding the following `ifndef` preprocessor directive.

```
#ifndef HW10_H
/* Prototypes go here */
#endif
```

> It is general convention to check for the name of the file in all caps with underscore seperators. This is not mandatory but will help you out in the long run with odd naming conflicts if you don't follow this.

This says if the preprocessor directive `HW10_H` is not already defined then include this header file. Next we need to define the preprocessor directive `HW10_H`.

```
#ifndef HW10_H
#define HW10_H
/* Prototypes, macros, and structs go here */
#endif
```

Now when this file gets included for the first time it defines `HW10_H` so the next time this file gets included `HW10_H` is already defined so the `#ifndef` check fails and doesn't include the contents of the header file again. You should add this header guard to all header files that you create.

Below is a list of headers that you must create for this assignment but there is nothing stopping you

from creating and adding more. Make and add whatever you need to. If you add other functions or headers try to make them as similar as possible to the function prototypes in glibc.

All header files should go in the `include` folder. When including any of these headers in your `.c` files you must use the `"file.h"` format, **NOT** `<file.h>`. These headers are self defined and not part of the std libraries that GCC was configured to use (you created them).

stdlib.h

In this file define some of the common macros such as `EXIT_SUCCESS`, `EXIT_FAILURE` and `NULL`.

- Using the preprocessor define the constant `EXIT_SUCCESS` as the value `0`
- Using the preprocessor define the constant `EXIT_FAILURE` as the value `1`
- Using the preprocessor define the constant `NULL` as `((void*)0)`.

Note about NULL

In the standard library NULL is defined in multiple header files. Because of this, for each header file that you define NULL in you should perform an additional check:

```
#if !defined(NULL)
/* Define NULL here */
#endif
```

stddef.h

In this file we want to define the constant `NULL` and `size_t`.

- Using the preprocessor define the constant `NULL` as `((void*)0)`.
- Typedef `size_t` as an unsigned int.

string.h

In this file we want to define some of the common string functions. Then we want to create the corresponding `libsrc/string.c` file which implements all the string functions defined in this header file. You must add `string.o` to `Makefile.lib` to compile it.

- Create the function `strlen`.
- Create the function `strcmp`.
- Create any other string functions you think you may need in this file.
- Using the preprocessor define the constant `NULL` as `((void*)0)`.

> ℹ If you created your string functions correctly in HW8, you can use that code here. Note, you may have to change some functions if you used other functions from the standard library.

stdio.h

Next we need to start creating our input and output functions. Let's create the function `fgets` to read in a line from stdin.

> ℹ Don't forget to modify `Makefile.lib` and add your source code to `libsrc/stdio.c`

- Using the preprocessor define the constant `NULL` as `((void*)0)`.

fgets

Mars provides syscall 8 which performs the same way as `fgets`. Create the file `libsrc/readstring.s` and expose syscall 8 the same way you exposed the other system calls in Part 1. Add the `fgets` prototype to `stdio.h`, then create `libsrc/stdio.c` and implement the `fgets` function.

```
/**
```

```
 * Reads in a string from stdin up to size bytes or `\n`.
 * Which ever comes first.
 * @param size Size of the buffer.
 * @param buffer Address of buffer to store characters.
 * @return Returns buffer on success, else NULL.
 */
char *fgets(int size, char *buffer);
```

> ℹ To test `fgets`, modify hello.c to use the `fgets` function. Compile into `hello.asm`, then run `hello.asm` with Mars.

Part 3 - Printf

In this part you will implement a slightly modified version of `printf`. Since `printf` is a variadic function it will not be so simple to implement. You will need to understand how the activation record is built. This function also requires you to use most of the system calls which you implemented/created in Part 1 as well.

Typically when `printf` is defined in the stdlib, it makes use of system specific macros in `stdarg.h`. Since we are using the "naked" gcc compiler, we do not have these macros defined for us. Instead you will have to investigate how gcc builds the activation record for your function. Then you need to determine how to access these variables in the `printf` function without these macros. Once you do this, you can then implement then be able to implement `printf`.

```
/**
 * Evaluates the fmt string, and writes to stdout.
 * @param fmt A `\0` terminated format string.
 */
void printf(const char *fmt, ...);
```

> ℹ To declare variadic functions in C you literally put `...` to say that there is an unknown amount of arguments. If you look up code examples online you will see macros used such as `va_start`, `va_end`, and `va_arg` are used. These do not exist in the cross compiler, since we have no stdlib, and the functionality of these macros is system dependent. This is why you needed to understand how gcc builds the activation record before attempting to create this

> function.

Let's try to understand how these variable number of arguments are passed to your function in MIPS assembly. To do this we will make a function call to `printf` from your `hello.c` program, compile it into an assembly file, and then run the assembly code in Mars. This will allow you to step through the code with the Mars debugger to see how the stack is used and how the activation record is created by the "naked" gcc for the `printf` function call.

To begin, declare the prototype for `printf` in `stdio.h` and create a basic `printf` function implementation in `stdio.c`. In the body of this function declare a few local variables (initialized to unique values) and perform a few basic operations (ie add, sub) on these variables. These instructions should be simple so you can easily identify them in the MIPS code for the body of your function.

Modify `hello.c` to only call the `printf` function (with some basic unique values). Make sure to call `printf` with more than four arguments. By doing so, you are forcing the gcc compiler to "spill" the arguments past the argument registers and place them on the stack within the activation record.

Compile `hello.c` to produce `hello.asm`. Transfer `hello.asm` to your local machine to assemble and run in Mars with the debugger.

> ⚠ In Mars you **MUST** turn on a setting called delayed branching. Failure to do so will cause "crazy" and "unexplainable" things to happen when you step through your code. To turn on delayed branching in Mars, go to the Settings menu and check the "Delayed Branching" menu option. Delayed branching will cause the instructions to "seem" to execute out of order as you step through your program. This is not relevant to your current assignment and is discussed in more detail in CSE320.

Step through the assembled code line by line and monitor the `$sp`, `$fp`, `$ra` registers and the stack memory area. The goal is to understand how the activation record for the printf function is created prior to executing the body of the function.

When you first begin to step through the code, the program will begin by executing the MIPS code which was in `crt0.s`. These are the basic instructions which are used to start the main function of your C program.

As you continue to step through your code you should monitor the values which are placed on the stack and where they reside with respect to the `$fp` and your `printf` function.

Ask yourself the following questions as you step through the code:

- Where are the local variables in the `main` program defined?
- Where are the local variables for `printf` function defined?

- Which and where are the arguments for the `printf` function placed?
- Where is the `$fp` set to? When is it set there?
- How do you know when the last argument is on the stack?

It will help to draw on paper the state of the stack as you step through the code.

Using these questions you should be able to determine the structure of the activation record built by the compiler.

> Note that the compiler does place items on the stack in what may seems like an odd order. The order in which the items are placed into the activation record is not important, but rather the state of the activation record when the first instruction for the body of your `printf` function is executed.

With this knowledge ask yourself which of the items/information about the parameters do you have access to in C ? How can you refer to the other arguments with respect to this item? This is what you need to implement the `printf` function in C.

In your `libsrc/stdio.c` file implement printf to support the following format specifiers.

- `%c` - For a single character
- `%s` - For null terminated strings
- `%d` - For integers
- `%o` - For octal numbers
  - There is no system call in Mars that prints octal numbers. You should create a new file `putoct.s` and write the code for printing numbers in octal format (You can use syscall 1 or 11 to print the value to stdout).
- `%x` - For hexadecimal numbers
- `%b` - For binary numbers
- `%u` - For unsigned numbers

You should also implement the width value which controls the minimum amount of spaces for strings (%5s), and also the `-` specifier to left justify strings (%-5s). Note, the width and `-` specifiers used with any other format specifier should be considered an error.

If the `fmt` string contains any errors stop printing where the error occurred and return.

Implement the pattern `%%` to escape the `%` sign so you can print out `%` with your `printf` function.

```
printf("Hello a %% sign\n") -> 'Hello a % sign'
```

> ℹ To test `printf`, modify hello.c (or create your own testing file) to use the `printf` function. Build the assembly file, then run it with Mars.

Part 4 - Creating a simple program loader

You are almost ready to do something interesting. You need to implement a system call that can execute code loaded into the data section of the program.

Creating a more advanced system call

First create the system call `load` in the file `libsrc/load.s`. You need to add its prototype to `lib.h`, and add `load.o` to `Makefile.lib`. This system call takes the address of memory that you want to execute in the register `$a0`. It should then jump and link to the address (look up the MIPS instruction `jalr`). At this point you put total control of your system in the hands of this loaded code. This loaded code will run, performing some task and then return control back to your program (using `jr $ra`). You will find the exit code of this program in `$v0`, your system call should return this value in `$v0`.

Next create `include/unistd.h` which will contain the prototype for the function `exec`.

```
/**
 * Attempts to execute a binary on the filesystem.
 * @param path Path to the executable.
 * @return Returns the return code of the executed program or -1 for
failure.
 */
int exec(const char *path);
```

You should then create the C function `exec` in `libsrc/unistd.c`.

This function should attempt to open the file as a read only file with mode 0 using the `open` system call that you defined in Part 1. If the file was successfully opened it will then use the `read` system call to read in the contents of the file into a globally defined buffer. If it fails to open, return -1. After reading the file into the buffer, you should then close the file, using the `close` system call from part 1. Finally if the previous actions were all successful you should then use the `load` system call that you just created to run the code in the buffer.

You should create a global program buffer with a size of 2048 bytes. Declare the following preprocessor constant in `unistd.h`.

```
#define EXEC_BUFFER_SIZE 2048
```

> ℹ To test `exec`, modify hello.c (or create your own testing file) to use the `exec` function.
>
> ⚠ If you run the program on your own machine with the debugger, you will need to additionally check the "Self Modifying Code" option in the Mars Settings menu.
>
> ℹ We provide two program binaries to execute in the directory `/usr/local/bin/shellbin`
>
> - The program `fun` prints a message to the screen
> - The program `rand` prints a random number to the screen

Part 5 - Putting it all together

Now that you have created the basic functionality to do something interesting, you should now create a new file `hw10.c` in the root directory of your project which will utilize all the things we made in this assignment.

> ℹ The root directory of your project is the same directory where `hello.c` is located.

You will create a very simple *"shell"* that can perform a few tasks. The basic operation of this shell is that it will go into a while loop which should accept input from a user. The user types something and your program evaluates that input. The maximum size of the global input buffer should be 2048 bytes. Put the preprocessor define in your `hw10.c` file (or `hw10.h` if you created one).

```
#define INPUT_BUFFER 2048
```

First you should implement a `help` command as a "built-in" utility of your shell. This will inform the user of your built-in utilities.

```
> help
List of utilities
help      Displays this help menu
exit      Terminates execution of the shell
echo      Simple utility that writes to stdout
>
```

After the `help` command runs, it should set the `$v0` register to zero.

Next you should implement the `exit` command which forces the *"shell"* to quit.

```
> exit
Exiting Program
```

> After typing exit, your should print out `Exiting Program`. Then Mars should stop running.

Next we will implement the `echo` utility. The command `echo` will simply write back to stdout the characters provided after the command.

```
> echo Hello, World!
Hello, World!
>
```

The command `echo` will also evaluate the special string `$?` as the return code of the previously ran command. You can find this value in the `$v0` register.

```
> help
List of utilities
help      Displays this help menu
exit      Terminates execution of the shell
echo      Simple utility that writes to stdout
> echo $?
0
>
```

Finally if the user types an input which is not one of the predetermined commands, your program should attempt to execute the command using your `exec` function that you created in part 4. If the file exists on disk, it should then be executed by your `exec` function. If it does not exist you should prompt the user letting them know that the command was not found. You should check to make sure that the path entered by a user is no longer than 256 characters. If it is too long print out "Path is too long" and return to the prompt.

```
> dkjdskl
dkjdskl: command not found
> echo $?
-1
> /home/cse220/shellbin/rand
1416423908
> thisisatestofareallylongpathname/seriouslyareallylongpathname/thatshouldbe
morethan256charatersinlength/itsstillnotlongenough/ihavetokeeptypingtomakei
treallytoolong/moreandmoreandmoreandmoreandmoreandmoreandmoreandmore
andmoreandmoreandmoreandmore
Path is too long
>
```

> ℹ You should specify the full path to the file that you are looking to execute on disk.
>
> ℹ We provide two program binaries to execute in the directory `/usr/local/bin/shellbin`
>
> - The program `fun` prints a message to the screen
> - The program `rand` prints a random number to the screen

Congratulations! You have completed HW10!

Below are a number of extra credit options. At the end of the document is the handin instructions (they are different, so make sure to read them!!).

Extra Credit 45 points

There are two parts to the extra credit. If you wish to do the options in the Extra Credit II then you **must complete** Extra Credit I. If you have done Extra Credit I, you can do either or both tasks in part II.

As with all previous extra credit assignments, for each part that you attempt, it must be a full implementation worth all or nothing.

Extra Credit I - setjmp, longjmp (10 pts)

We will try to implement a very basic form of cross procedure transfer control functions. These are the first steps to how something like signal handling, or a try/catch block might be implemented. You will need to create a struct called `jmp_buf` in the header file `include/setjmp.h` which contains the following information:

- The contents of the return address register.
- The contents of the stack pointer.
- The contents of the frame pointer.
- The contents of the callee-save registers ($s0 - $s7)

You will then create the function `int setjmp(struct jmp_buf *env)` which initializes the data in the jmp_buf struct. When you first set the `jmp_buf` using `setjmp`, it records the current state of the program, and then the function should return the value zero. You will also need to create the function `void longjmp(struct jmp_buf *env, int val)`. This will set the value of `$v0` to val, and then restore the stack pointer, frame pointer, return address, and callee-saved registers back to what is stored in the handler. In this way you have restored the state of the program back to the state prior to the call to `setjmp`.

Let's take a look at how these functions would work together.

```
// try.c
#include "setjmp.h"
#include "stdio.h"

void foo();
void bar();
struct jmp_buf env;

int main(int argc, char *argv[]) {
        int val;
        if((val = setjmp(&env)) == 0) {
            /* "Try block" */
```

```
            printf("In the 'try' block\n");
            foo();
        } else {
            /* "Catch" block */
            printf("In the 'catch' block, val=%d\n", val);
        }
        printf("Program terminating\n");
    return EXIT_SUCCESS;
}

void foo() {
    printf("In function foo\n");
    bar();
    printf("Oops, after call to bar (shouldn't happen)\n");
}

void bar() {
    printf("In function bar\n");
    longjmp(&env, 42);
    printf("Oops, after longjmp (shouldn't happen)\n");
}
```

If you run the above program the output should be:

```
In the 'try' block
In function foo
In function bar
In the 'catch' block, val=42
Program terminating
```

You should add the `setjmp` and `longjmp` prototypes to the header file `include/setjmp.h` and implement the functions in `libsrc/setjmp.c`.

Your implementations of `setjmp` and `longjmp` should permit any number of `struct jmp_buf` to be active at any given time (each one with its own `struct jmp_buf` initialized by a different call to `setjmp`). In addition, it should be possible to call `setjmp` from any function (not just from `main` as shown above). Similarly, it should be possible to call `longjmp` from anywhere but only under the following conditions:

- The `jmp_buf` structure passed to `longjmp` must have previously been initialized by a call to `setjmp`.
- The function activation that made the previous call to `setjmp` must not have returned between the call to `setjmp` and the subsequent call to `longjmp`.
- No previous call to `longjmp` on the same handler structure has occurred since the previous call

to catch.

Modify `try.c` which is in your home directory to test other cases including multiple `jmp_buf` structures in the same program. We will test with our own file during grading.

Extra Credit II - Total 35 Points

Below are two more extra credit tasks that you can tackle. You can do either or both, just remember that to receive points each must be a full working implementation (no partial credit is given), and you **MUST** have also completed Extra Credit I.

fprintf - 15 Points

For our printf calls we were able to use the putchar, and putint, etc. to write to standard out. To write to files other than stdout, we need to re-implement the system calls so they can write to arbitrary files. Write a modified version of `fprintf` as follows.

New system calls

1. write file - write.s
    - syscall 15

unistd.h

Add the following constants for the file descriptors used for stdin, stdout, and stderr.

- Using the preprocessor, define the constant `STDIN_FILENO` as the value `0`
- Using the preprocessor, define the constant `STDOUT_FILENO` as the value `1`
- Using the preprocessor, define the constant `STDERR_FILENO` as the value `2`

```
/**
 * Evaluates the fmt string, and writes to the provided file descriptor.
 * @param fd Open file descriptor to a file.
 * @param fmt A `\0` terminated format string.
 * @return Returns the number of character written to the file, and a
negative
 *  number on error.
 */
int fprintf(int fd, const char *fmt, ...);
```

You should implement all the functionality of `printf` that you did for the normal assignment. You should define the prototype for `fputc` in `stdio.h` and implement the function `fputc` in `stdio.c`. Use this function to assist in writing `fprintf`.

```
/**
 * Writes a single character to a file.
 * @param c Character to write to file.
 * @param fd Open file descriptor to a file.
 * @return Returns the character written, or EOF on error.
 */
int fputc(int fd, int c);
```

You need to implement all the functionality for each of the formats for converting the numbers to strings before printing out using `fputc`.

Dynamic Memory - 20 Points

While we have added back some very basic functionality to our system, we currently have no way to allocate any memory dynamically.

- Implement the system call sbrk in `sbrk.s` (syscall 9) and add the prototype to `lib.h`.
- Implement the functions free and malloc.
    - These functions will be tough to implement and any of the following implementations or variations are acceptable as long as it works correctly (meaning allocate space requested reasonably and be able to free the space).
        - Doug Lea Implementation
        - Malloc Lecture 1
        - Malloc Lecture 2

> ℹ There is an implementation of `malloc` in the K&R C book. This can be a good starting point if you are completely confused where to start but since they give you all the code this is not an acceptable solution to this extra credit assignment.

To assist in debugging your program, if your library is compiled with `-DCSE220` your program should print out some debug messages to help us follow the steps of your algorithm. You should include enough output to convince us that your `malloc` and `free` implementations are actually working. You must include things such as the amount of total space allocated, how much space is actually in use, the address of each memory location, etc.

Hand-in instructions

You should create a `README` file in your home directory which contains you full name, sbuid, netid, and recitation number. In this `README` file you should state any interesting things that could assist in grading your program correctly such as anything that is not working 100%, any of the extra credit parts that you did, and maybe anything else that you felt was novel and we should take a look at.

There is no submission procedure. Your home directory on `130.245.12.103` is the submission. Since this project contains multiple files, when the deadline has passed your home directory will be changed to read only access so you can no longer modify the files stored on the server. When the TA logs into the server he should simply be able to run the command `make hw10.asm` and your project should build with no issues.