CSCI 347
Assignment 1

100pts

Modern programs are very complicated systems. To manage this complexity, the source code is divided into separate source files. Each file provides a single part of the overall behavior. Building the program, however, involves compiling each source file and linking the resulting object files into a single executable.

Rather than type hundreds or even thousands of commands to build a complex system, developers use a tool to automate the build process. One such tool is called `make`. `Make` actually performs a lot of logic. For now, let's just consider the scripting nature of `make`.

Through most of this course, you will develop a minimal `make` program that will automate build processes. Let's call it micro-make or `umake`. I have provided a starting point for this project.

In GitLab is a group associated with this class. That group owns a single `git` project called `umake`. Crate a fork of this project in your own account. I will provide instructions of this part of GitLab.

Right now, the program will open and read a file called `uMakefile`. Any line that it finds in that file starting with a tab character is interpreted as a command. It can only execute commands without arguments (e.g., `ls`, `pwd`).

Modify the program so that it can execute commands that take arguments:

- Write a function called `arg_parse` with the following signature

  ```
  char** arg_parse(char* line)
  ```

  This function returns a new array of pointers that point to characters in `line`. You may only call `malloc` once and you must not over-allocate space. That is, `malloc` only the space necessary for the array. You do not need to copy characters out of `line`. However, you will be replacing some of the characters in `line` with the null character (`'\0'`). Note that arguments are separated by white space, but not necessarily a single space. For example, the following is only three arguments

  ```
  One    two four
  ```

  In this case, `arg_parse` should return an array of four pointers: the first pointer points at the `'O'` in one, the second points at the `'t'` of two, the third at the `'f'` of four, and the last pointer is null. Notice that the leading spaces are ignored. *Do not use `strtok(3)` for this assignment.*

- Modify the `processline` to use `arg_parse` to run commands with arguments. The parent should call `arg_parse` (before the fork()) and do not forget to free the pointer returned by `arg_parse` in the parent shell process. Read the `man` page on `exec(3)` and choose the right form of the system call. Hint: the right call is neither `execlp` nor `execve`.
- Make sure you have no warnings when compiled. I will use the command `gcc -Wall` on Ubuntu as provided by the department in the labs. (Remember, you have remote access to the linux-01 to linux-12 machines. Domain is cs.wwu.edu. Some kind of ssh client is required to remote acc ess these machines.)
- Submit your changes to `git`. That is

  ```
  $ git add umake.c
  $ git commit -m 'An informative commit message'
  $ git push origin master
  ```

  Okay, the "origin master" bit is only required the first time you push to GitLab.

You must start with my code. Do not make unnecesary modifications.

Test File

Here is a good starting point for testing. Name this file `uMakefile`.

```
all : umake

umake.o: umake.c
     gcc -c umake.o

umake: umake.o
     gcc -o umake-new umake.o
     mv -i umake-new umake
```

Please notice that the indented lines lead with a tab character. Make sure that your editor inserts tabs and not spaces. Atom, for example, is really bad at this.

Outcomes

- Thorough understanding of development in the UNIX environment