In class we will discuss a microshell. Source code may be found on my web site, http://facultyweb.cs.wwu.edu/~phil/classes/m23/347/microshell.c. I want you to extend this microshell to do simple argument processing in the shell. Specifically, I want you to do the following:

- Read the introduction to UNIX found on the class web page. Even if you already know UNIX. Especially notice the information on the program a2ps(1). I want what you turn in to me to be at least as nice as what a2ps produces. Your a2ps output may be two logical pages per physical page. There is also a ps2pdf(1) program that turns postscript (.ps) files into .pdf files. So you can use a2ps to generate .pdf files.

- Read the man pages on execve(2) and related functions (exec(3)). (Yes, you will need to use a different exec... function than what is currently used in the microshell.)

- Read the man page on malloc and if needed, consult a C programming text on how to use malloc(3) and the C sizeof function. Also, read about free(3).

- (10 points) Create a new private project on gitlab.cs.wwu.edu with the name of “csci347_m23” (without the quotes and it must be all lower case and exactly as specified). Give user “phil” and user “grader4phil” access to your gitlab project. Please give both users “reporter” access. This is important. (On the web site, go to Settings/Members to add access to your project.)

- Clone your new project. This clone can be either to your home machine or a lab machine. After you clone your project, add a README.md that has your name and quarter in it. Next, add a directory named “ush” to your project. All ush files must be in this directory. Copy the microshell.c to your project’s ush directory and name it ush.c. Commit and push both files.

- (60 points) Write a function “arg_parse” that must have the prototype: char ** arg_parse (char *line, int *argcptr). The “line” parameter points to the line (character array) that contains the command to be processes. The “argcptr” parameter points to an integer variable where
this function must put the number of arguments found in the “line”. The
return value must be a pointer to a malloced area that points into the line
parameter and matches what is needed by execve for the “argv” parameter.
The malloced area should be the exact size needed, not too small and not
too big. You should call malloc(3) only once. Make sure you check for
malloc(3) errors. (Note: You should use argcptr only just before a return.
You should have a local “argc” variable that keeps track of the number of
arguments and then use argcptr to save argc to the correct place just before
arg_parse() returns.)

Do your processing by breaking up the line on spaces. Do not do more
complex processing at this time. You should not copy any characters out
of the line parameter. You only put zero characters in the line parameter
and set pointers to characters in the line parameter in your malloced area.
Notice that two or more consecutive spaces is equivalent to a single space.
For example, a call to arg_parse with the parameter line

mycommand    arg1    nextarg    arg3    not-arg-5-but-4

should return a pointer to an “array” of 6 elements. The first element (index
0) should point to the character ’m’ in mycommand, the second element
should point to the ’a’ of arg1, and so forth. The last element should be a
NULL pointer. The characters “mycommand” are not copied to any other
storage. The space just after “mycommand” is turned into a zero character
so “mycommand” is a zero terminated string. Also, leading and trailing
spaces should be ignored! (Before the command name and after the last
argument.) Note: Do not use strtok(3) from the string library.

• (10 points) Modify the microshell function processline() to use arg_parse
to run commands with arguments. Call arg_parse before the fork() call
and if there are no parameters on the line, do not fork(), but just return
from processline() without doing more. Do not forget to free the pointer
returned by arg_parse() in the parent shell process. You will also need to
modify the child code in processline() to call a different exec function. Do
not use execve. You must not have any “int *” variables in processline().

• (10 points) At this point, you should have a shell that can run commands
with command line arguments. Create a script to show you have a working
shell at this point. See below in the “Turn-in” section and save the output
in a file called “script-nq”. Commit your work at this point and create a
branch called “a1-nq”. Make sure you keep working on the master branch
for the remainder of the assignment. Note: No physical turn-in is required at this point. The branch time of the a1-nq is the time you turned in. If you have commits to the a1-nq branch, the latest commit on the branch is considered the turn-in time. The following commands should create your “a1-nq” branch and push it to gitlab.

```
  git branch a1-nq
  git push --set-upstream origin a1-nq
```

All work to this point must be finished no later than June 24, allowing one day to complete the rest of the assignment. It might be best to see if you can get this done by June 23.

- (40 points) Now, add double quote processing to arg_parse(). Implementing arg_parse() with quote processing first will cost you points. If a quote is found, look for a matching quote and everything between the two quotes is considered to be consecutive characters in the current argument. For example:

```
  prog "this is a single arg"
```

has 2 arguments and

```
  prog this" "is" "a" "single" "arg
```

also has exactly 2 arguments. And again,

```
  prog thi"s is a s"ingle a"rg
```

has exactly 2 arguments and is identical in result to the above 2. Also, you can make a program name with spaces like:

```
  "prog name with spaces"
```

Note: a quote may appear in the middle of an argument as you can see by the second example above. Also, the actual quote character is removed from the final version of the argument. If you find an odd number of quotes in the line, print an error message to “standard error” and stop processing the current line. (Do NOT call exit() to stop ush.)
• Make sure you have no warnings when compiled with “gcc -Wall -Wextra” 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 access these machines and you now have to connect to the WWU campus VPN. The CS department has a page for helping you connect.
https://gitlab.cs.wwu.edu/cs-support/public/-/wikis/home/survival_guide/day_to_day

• (20 points for remaining work) When you are completed with your work for this assignment, create a branch named “a1” in your git repo. This branch will be used to check out and grade your assignment 1. You should not have any commits on your “a1” branch. All of your work should be on branch “master”. (Note: If you accidentally get commits on your “a1” branch, you will want to merge the “a1” branch back into “master” to get changes on “a1” back on “master” where you are to work. This work-flow is different than many companies and open source projects want when using git. A very typical work-flow is for a person to do their work on a branch and get it working. Then they merge a working branch back to “master”. In that way, nobody is working on “master” and committing directly to master. All work is merged from a branch.)

After you create the branch, you will need to push that branch to gitlab. You do this using the command:

git push --set-upstream origin a1

• Turn in a pdf copy of your code and a “typescript” of a compile and run of your shell as well as the first run before adding quote processing. Use script(1) to make the “typescript”. (Read the man page.) Remember to use a2ps and ps2pdf or something better. I also want a cover page that includes your name, “CS 347 Quarter year” and the assignment number. You need to do this for ALL your assignments this quarter. The turn-in will be on canvas. Commits to your assignment branch after the turn-in deadline will make your assignment late, even if you submitted to canvas on time.

Note: These are the only modifications you are to do to this shell. You must start with my code. Any other kinds of modifications will cause you to lose points. (e.g. Changing the prompt.)

Note 2: I will be sending e-mail from time to time to the class. I will be using your student e-mail address. Make sure that you either read your student e-mail often or have it forwarded to your preferred e-mail address.
Note 3: You should have an account on the CS machines. You need to try to login before June 22. If you can’t login, send me email and I’ll make sure you can login. Commit times on your gitlab repository and submission to canvas will be used to determine if your assignment is late. ANY commit after the due date and time on your assignment branch will make your assignment late. (Commit on master after the due date and time will be considered as part of the next assignment.)

Note 4: When you e-mail me about problems you are having, please commit your current code, even if it is not working, and push it. I will be able to check-out your code and help you. And again, do not create the “a1” until you are completed with the assignment and ready to turn it in.

Note 5: Given the rapid nature of this class, please make sure you contact me when you are having problems. Don’t spend more than ONE HOUR with no progress before trying to contact me. Zoom “office” sessions can happen at any time of the day (or night) if I am available as well as in person ones right after class.