Essa Husary | ID: 917014896
Github: EssaHusary | CSC415 Operating Systems

Assignment 3 – Simple Shell Writeup

***NOTE: My references are on the final three pages of this document where I've cited my sources as well as explained what I referenced and from which source(s).

Description:

In this assignment, we are creating a simple shell program that exists within our existing shell. So a shell within a shell (and possibly, our shell has other shells in it if we so choose). The purpose of this assignment is to acclimate us to processes - creating them, exiting from them, returning from them, waiting for them to terminate – and to acclimate us to the exec family of functions, string tokenization in C, string buffering in C, and input buffering in C.

Approach / What I Did:

After cloning the remote repository to my machine, I read the README.md file. After doing so, I thought about how to break the assignment down into phases or steps. After some thinking, I finally was able to break the assignment down into three main steps. Step one would be to tokenize user input and capture that input somehow. Step two would be to create the child process, call execvp(), and wait for it to terminate. And the final step, step three, would be to handle edge cases, including invalid user input and reaching the end of a file (or EOF).

So in the HW3 assignment page on canvas, we were provided a hint by you, Professor, where you said that we can use the function strtok() to tokenize user input. The problem is, I had no idea how to use it, and while the man pages were somewhat useful, it still wasn't very clear (see "Issues and Resolutions" below). But I eventually figured out how to do so by referencing an article titled "Finding Tokens in a String" from the GNU C Library Reference manual (cited in the Works Cited Page on the third-to-last page of this document), seeing an example of how it can be used. So when it became clear how to use strtok(), I thought, "Well, what next?". In the assignment document, it is mentioned that since we are using execvp(), which has as a second parameter a char * array, we can store pointers to the tokenized substrings in the char * array that we'll create which will act like main's char * argv[] parameter when passed to execvp(). So I declared my char * array which I called "stringParts", since it'll store the tokenized strings, and then with the help of a 'while' loop and the fgets() function, I was able to successfully tokenize any input that a user enters in. Phase one complete.

As for step/phase two, I had to create the child process, call execvp(), and then wait for the child process to terminate, printing out its pid and exit status. I thought to myself, "Okay, I watched the lecture videos and read in the textbook about how to do these things, so let me refer to these resources to remind myself of how to do them." Sure enough, I saw in the e-textbook in chapter 3, section 3.3.1, page 119, a snippet of code that the authors provided where a variable representing a process id called "pid" was declared, followed by forking and an if-else block to check whether the fork has failed or if we are in the child process or if we are in the parent process. I used this snippet (cited in my code and in the Works Cited Page) to create the child process, but I ran into an issue. I didn't know how to use the execvp() function. I mean, I know what it does, but I didn't understand HOW to use it (see "Issues and Resolutions" below). When I finally understood how to use execvp(), I called it, called wait() with its parameter being the address of the "exitStatus" variable and its return value being stored in the "pid" variable, and printed out all of the required information about the child process specified in the assignment README.md document. I did have a bit of trouble getting the correct exit status, but by referring to the man pages and the lectures, I used the correct macros to get the correct exit status (see "Issues and Resolutions" below). I have to admit, this was the trickiest part of the assignment mainly because I didn't understand fully how execvp() worked and because processes are a new concept to me, but with planning, I was able to solve the issues that I had. Phase two complete.

The final step, step three, is really just all about handling edge cases and special cases, such as opening and running a text file filled with commands (i.e., commands.txt). The way I approached this step is really just going through each edge and special case, one by one. I started with the case in which the user either specifies or does not specify a custom prompt, which turned out to be quite a simple case to handle. I thought to myself, "Well, if arg[1] has a value, then that value is the prompt, and if the user just hits enter without typing anything in, then arg[1] is just NULL and I can just print out the default prompt specified in the assignment document". And so, I wrote an if-else statement following that line of reasoning and I handled that case successfully. I then moved on to the second special case which is the case in which fgets() returns NULL when we either reach the end of a file (in our case, running commands.txt from the terminal) or when fgets encounters an error reading user input. When thinking about how to handle this, I thought, "Hmm, I'm getting a segmentation fault when running 'make run < commands.txt'. What gets stored in the buffer when running this command?" After a series of printfs, reading more about fgets, and through office hours, I realized that fgets() returns a NULL pointer when reaching EOF or when there's an error. So coming to this realization I decided to declare a variable called "returnOfFgets" of type char *, which will store the return value of fgets(). If "returnOfFgets == NULL", we'll use two functions: feof(), for the purpose of determining if the NULL value is the result of reaching EOF, and ferror(), for the purpose of determining if the NULL value is the result of an error while fetching user input, both of which having "stdin" passed in and both of which will be placed in an if-else block. If EOF, we print out a message and exit the program gracefully, otherwise, if there is an error, we report the error and exit the program with an exit(errno) function. Pretty simple.

The next special case is where a user types in “exit”. I thought, “Well, my shell doesn’t recognize that command so I must insert it after execvp()”. After attending office hours, it turned out that I approached the wrong way. All I had to do was to just simply perform a string compare using strcmp() BEFORE forking. Silly me. After that case was accounted for, I moved on to the case where an invalid command/program was entered in by the user such as the command “wow”, which obviously doesn’t exist. I thought to myself, “Well, based on what I’ve learned, if a program doesn’t exist then execvp() will fail. If it fails, then I can have a message be printed out and then exit the child process.” I didn’t know what to pass in into the exit() function, but after office hours, I learned that I can pass in “errno” and get the “file not found” exit/return code. Not too bad of an edge case to deal with.

The next edge case I needed to handle was the edge case in which an empty line was entered in by the user. For some reason, this was one of the two hardest edge cases to account for (the other difficult one will be described shortly). I just couldn't wrap my head around how to know if a user entered in anything at all (see “Issues and Resolutions” below). The solution to this turned out to be quite simple as all I had to do was handle it the way I handled whether a custom prompt was specified or not. Problem solved. And finally, the final case that needed handling was the special case in which a user enters in an input that exceeds the size of the string buffer (and fgets() as well). I had a very difficult time figuring out how to truncate the input (see “Issues and Resolutions” below). Thankfully, I figured it out in time to submit the assignment. Step three done. Assignment complete.

Issues and Resolutions:

The first issue I had was using strtok() to tokenize the strings. I had never used a function like strtok() before. To add on to my inexperience using it, the man pages weren’t quite helpful as there wasn’t a good example that could help me understand better. So to resolve my issue, I did a bit of research. I found an article called “Finding Tokens in a String” from the GNU C Manual and I finally figured it out. All I had to do was pass in the string into strtok() once with delimiters, and then in subsequent calls, pass in NULL along with delimiters.

My second issue revolved around the use of execvp(). I remember from the lectures that the exec family of functions are called in child processes to overlay their address spaces to run programs. But I didn’t quite get how to use them. Resolving this issue took A LOT of research and some more thinking. According to the man pages, we read, “The first argument, by convention, should point to the filename associated with the file being executed” (“Linux Programmer’s Manual”). And based on what we see in the lectures, the first parameter is “const char *file”. It took me a very long time to realize that the commands such as “ls”, “echo”, “w”, “cat”, “mkdir” etc. etc. are PROGRAMS! They are FILES! That was the source of my confusion! When I realized this, I passed in stringParts[0] (which again, as I mentioned in "Approach/What I Did", stringParts is my array that stores the arguments), and the program finally started working. So the resolution was the realization that the commands we run in any CLI on the planet are programs. Problem solved.

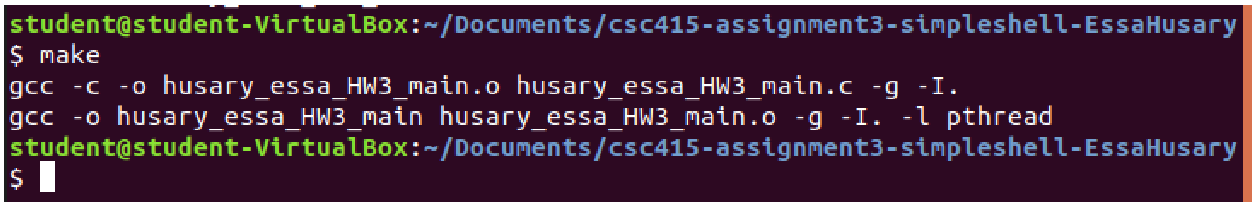
Issue number three was a rather simple issue to deal with, yet it did take some time to fix and this issue had to do with getting the exit status of the child processes. Now, getting the exit status is simple as we just call wait(&exitStatus). The tricky part was getting the correct exit status when "ls" can't find, say, "foo", to match the exit status seen on the assignment document which is a value of "2". But this issue was easily resolved when I referred back to the lectures to review the wait() function and remembered the macros that we can use. I referred to the man pages and was eventually able to get the correct exit status by using the "WIFEXITED" and "WEXITSTATUS" macros.

Issue number four revolved around the user entering in no input, such as an empty line. It's actually very simple to deal with, yet to me, for some reason, I perceived it as one of the trickiest edge cases to deal with. As I said in "Approach/What I Did", I just couldn't wrap my head around how to tell if a user entered in anything at all. But after lots and lots of thinking, I finally came to a resolution. It hit me that just as we'd have a value of NULL if a user doesn't specify a prompt and just hits the "ENTER" key, "stringParts", my array of arguments, would have an initial value of NULL if the user doesn't type in anything. So all that was needed was a check using an 'if' statement to see whether stringParts[0] == NULL. Solved.

And, the final issue I had in this assignment was dealing with the special case in which we need to truncate input when a user types in way too much input. This issue took me three days to resolve. I was stuck figuring out how the input buffer works. I tried coming up with different ideas and methods for clearing the buffer after using fgets() but to no avail. I discovered after tons of research and trial and error a function called "getchar", which can remove data, specifically chars, from the input buffer. I thought that this would be very useful as my goal is to clear the input buffer, but the question arose: "HOW am I going to use this function?" I figured that I could place this function in a while loop such that we can clear ALL of the data from the input buffer. Then I thought, "Okay, well I can do that but under what conditions will I need such a loop?" After some thinking, I remembered that fgets() is designed such that if a user hits the "ENTER" keyboard key, a newline character gets slapped to the end of the input. And then it hit me. If the string buffer containing the input from fgets() does not contain the newline character, then that means that we have chars in the input buffer. We can iteratively call getchar(), effectively "iterating" through the input buffer, removing chars using getchar(), and when we reach either the newline character or EOF, we will break. Lo and behold, I finally came up with a solution to this issue. So my resolution was honestly just a result of perseverance after three whole days of thinking about this problem. Final issue is resolved.

Analysis: ANALYSIS NOT REQUIRED FOR THIS ASSIGNMENT

Screen shot of compilation:

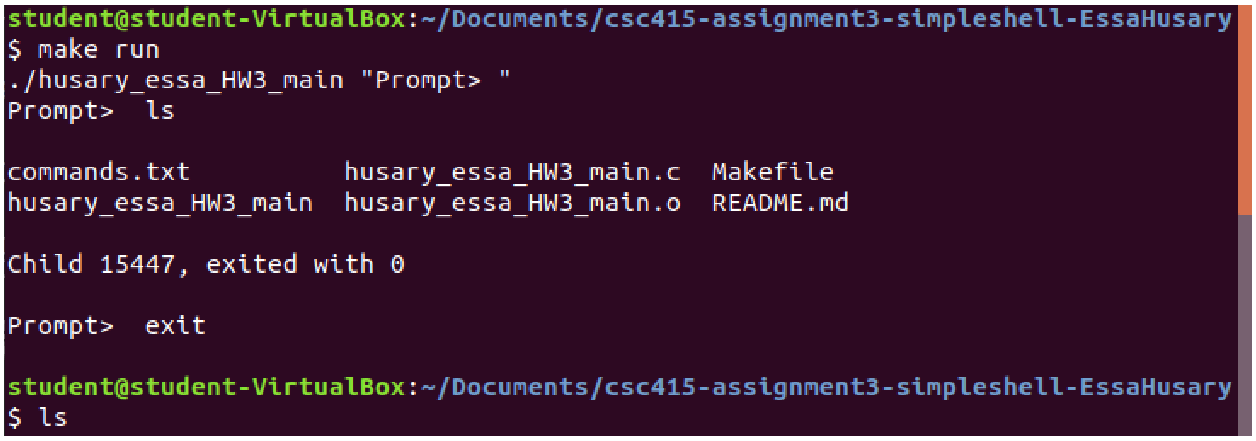


```
student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ make
gcc -c -o husary_essa_HW3_main.o husary_essa_HW3_main.c -g -I.
gcc -o husary_essa_HW3_main husary_essa_HW3_main.o -g -I. -l pthread
student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ 
```

Screen shot(s) of the execution of the program:

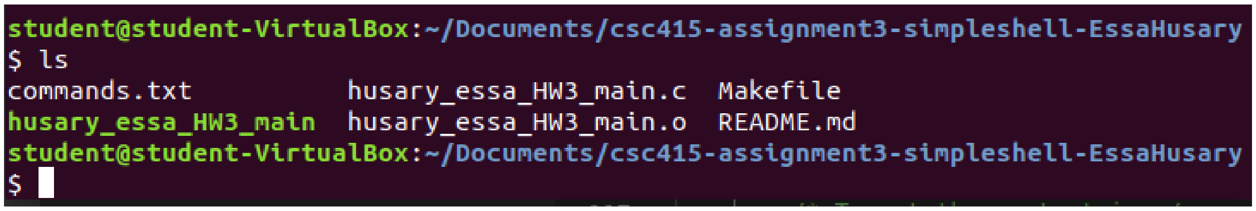
Output #1 – The “ls” command:

My shell:



```
student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ make run
./husary_essa_HW3_main "Prompt> "
Prompt>  ls

commands.txt          husary_essa_HW3_main.c  Makefile
husary_essa_HW3_main  husary_essa_HW3_main.o  README.md

Child 15447, exited with 0

Prompt>  exit

student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ ls
```

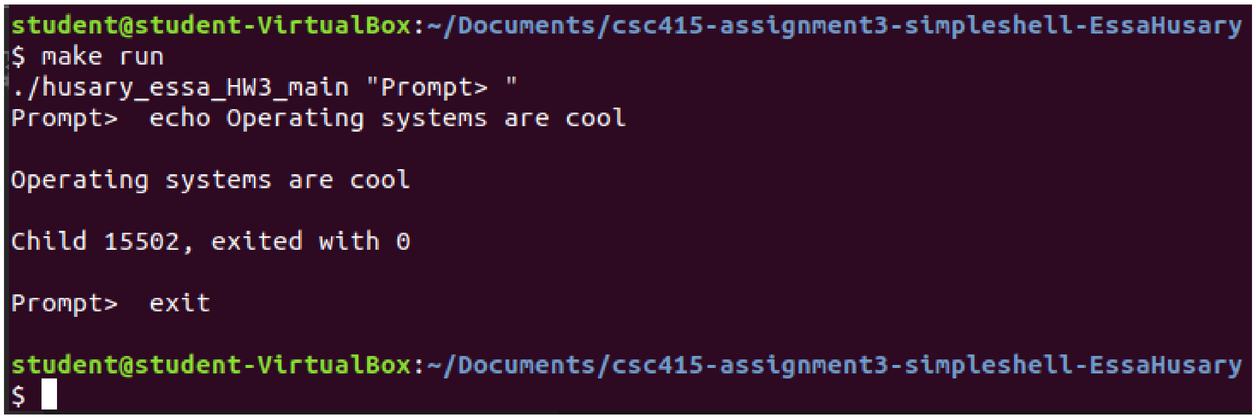
Regular CLI:



```
student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ ls
commands.txt          husary_essa_HW3_main.c  Makefile
husary_essa_HW3_main  husary_essa_HW3_main.o  README.md
student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ 
```

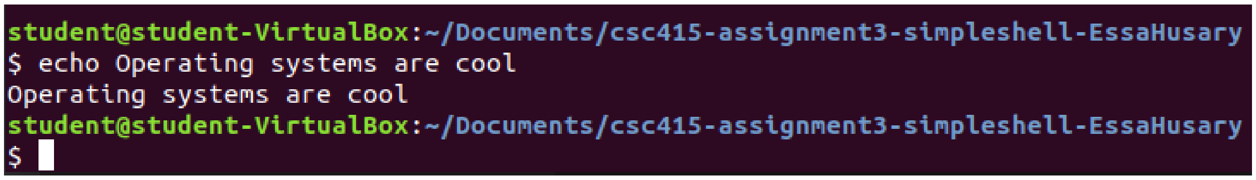
Output #2 – The "echo" command with additional args:

My shell:



```
student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ make run
./husary_essa_HW3_main "Prompt> "
Prompt>  echo Operating systems are cool

Operating systems are cool

Child 15502, exited with 0

Prompt>  exit

student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ 
```

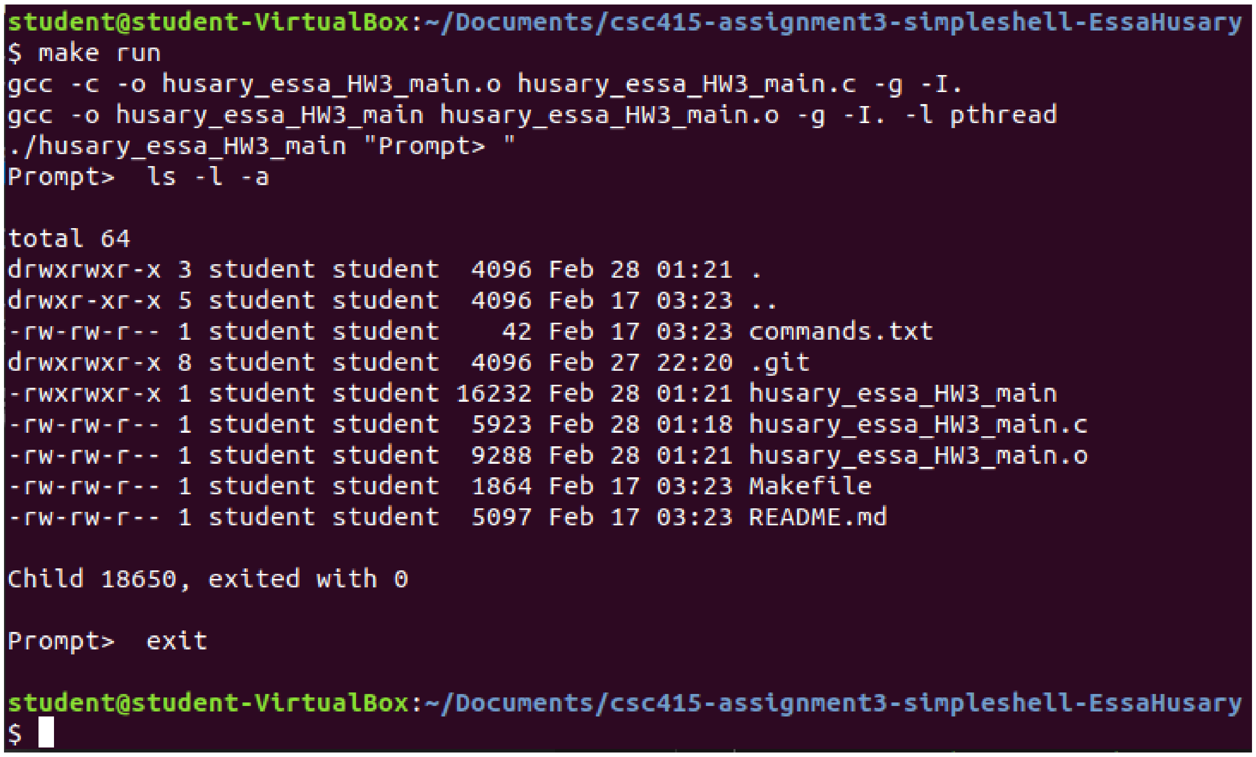
Regular CLI:



```
student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ echo Operating systems are cool
Operating systems are cool
student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ 
```

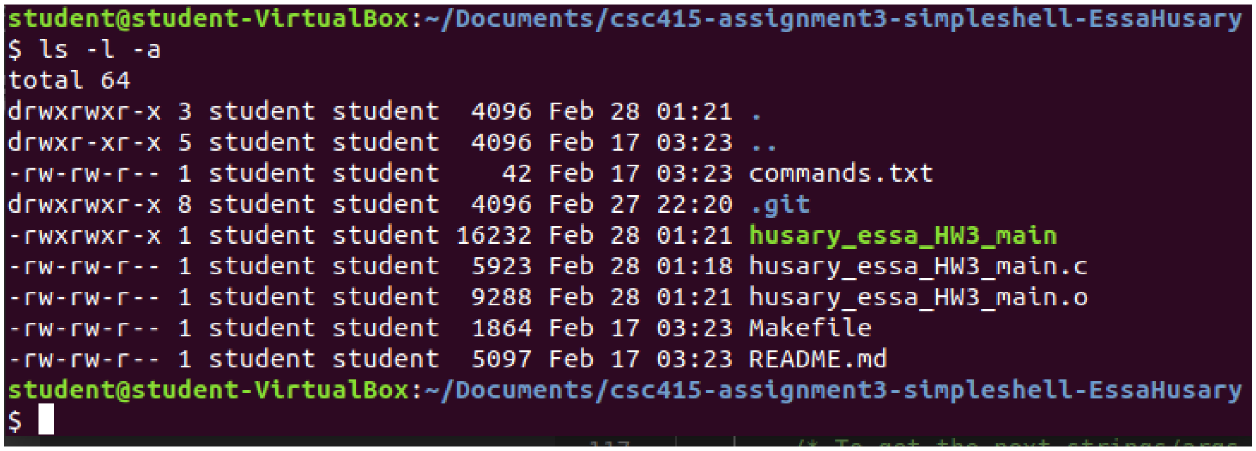
Output #3 – The "ls" command with the "-l -a" args:

My shell:



```
student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ make run
gcc -c -o husary_essa_HW3_main.o husary_essa_HW3_main.c -g -I.
gcc -o husary_essa_HW3_main husary_essa_HW3_main.o -g -I. -l pthread
./husary_essa_HW3_main "Prompt> "
Prompt>  ls -l -a

total 64
drwxrwxr-x 3 student student  4096 Feb 28 01:21 .
drwxr-xr-x 5 student student  4096 Feb 17 03:23 ..
-rw-rw-r-- 1 student student    42 Feb 17 03:23 commands.txt
drwxrwxr-x 8 student student  4096 Feb 27 22:20 .git
-rwxrwxr-x 1 student student 16232 Feb 28 01:21 husary_essa_HW3_main
-rw-rw-r-- 1 student student  5923 Feb 28 01:18 husary_essa_HW3_main.c
-rw-rw-r-- 1 student student  9288 Feb 28 01:21 husary_essa_HW3_main.o
-rw-rw-r-- 1 student student  1864 Feb 17 03:23 Makefile
-rw-rw-r-- 1 student student  5097 Feb 17 03:23 README.md

Child 18650, exited with 0

Prompt>  exit

student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ 
```

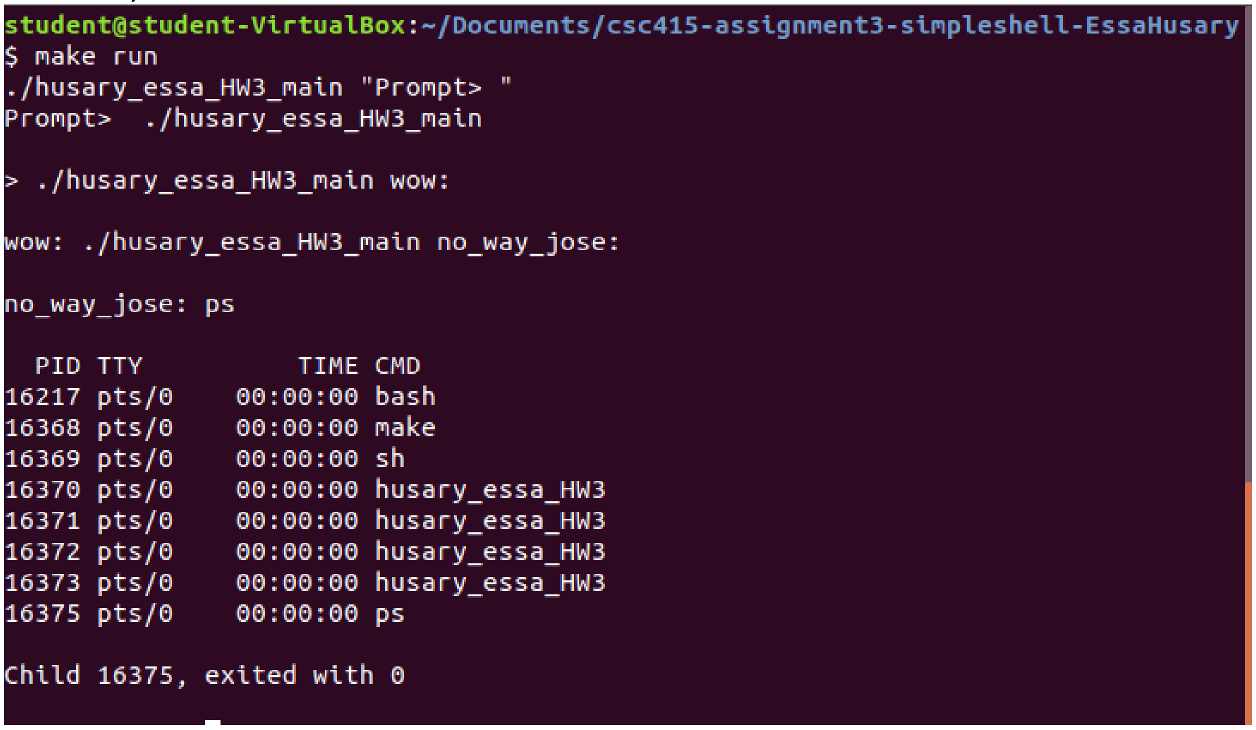
Regular CLI:



```
student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ ls -l -a
total 64
drwxrwxr-x 3 student student  4096 Feb 28 01:21 .
drwxr-xr-x 5 student student  4096 Feb 17 03:23 ..
-rw-rw-r-- 1 student student    42 Feb 17 03:23 commands.txt
drwxrwxr-x 8 student student  4096 Feb 27 22:20 .git
-rwxrwxr-x 1 student student 16232 Feb 28 01:21 husary_essa_HW3_main
-rw-rw-r-- 1 student student  5923 Feb 28 01:18 husary_essa_HW3_main.c
-rw-rw-r-- 1 student student  9288 Feb 28 01:21 husary_essa_HW3_main.o
-rw-rw-r-- 1 student student  1864 Feb 17 03:23 Makefile
-rw-rw-r-- 1 student student  5097 Feb 17 03:23 README.md
student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ 
```

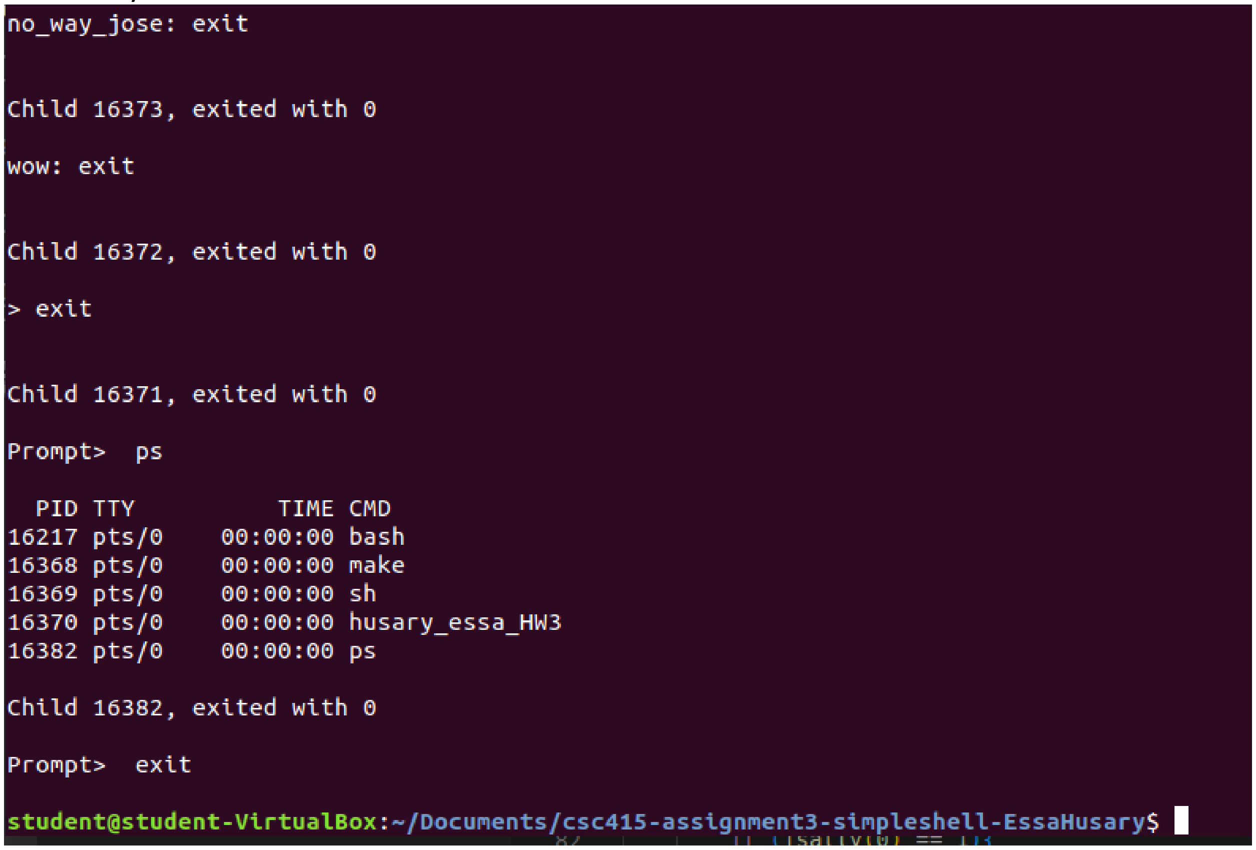
Output 4 – The “ps” command after creating a shell within a shell within a shell within a shell:

My shell:



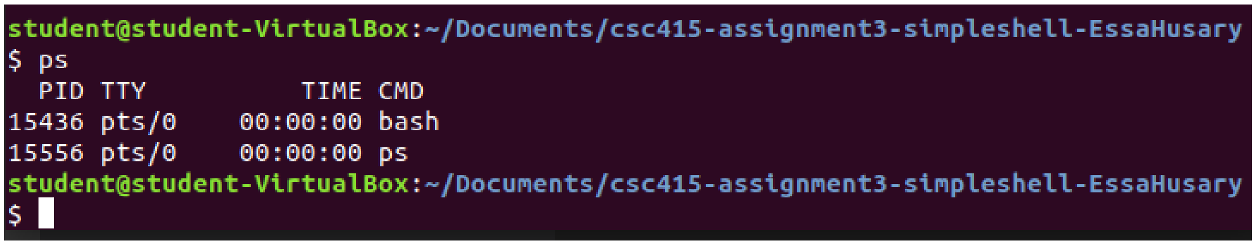
```
student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ make run
./husary_essa_HW3_main "Prompt> "
Prompt>  ./husary_essa_HW3_main

> ./husary_essa_HW3_main wow:

wow: ./husary_essa_HW3_main no_way_jose:

no_way_jose: ps

  PID TTY          TIME CMD
16217 pts/0    00:00:00 bash
16368 pts/0    00:00:00 make
16369 pts/0    00:00:00 sh
16370 pts/0    00:00:00 husary_essa_HW3
16371 pts/0    00:00:00 husary_essa_HW3
16372 pts/0    00:00:00 husary_essa_HW3
16373 pts/0    00:00:00 husary_essa_HW3
16375 pts/0    00:00:00 ps

Child 16375, exited with 0
```

My shell after a series of “exit” commands:



```
no_way_jose: exit


Child 16373, exited with 0

wow: exit


Child 16372, exited with 0

> exit


Child 16371, exited with 0

Prompt>  ps

  PID TTY          TIME CMD
16217 pts/0    00:00:00 bash
16368 pts/0    00:00:00 make
16369 pts/0    00:00:00 sh
16370 pts/0    00:00:00 husary_essa_HW3
16382 pts/0    00:00:00 ps

Child 16382, exited with 0

Prompt>  exit

student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary$
```

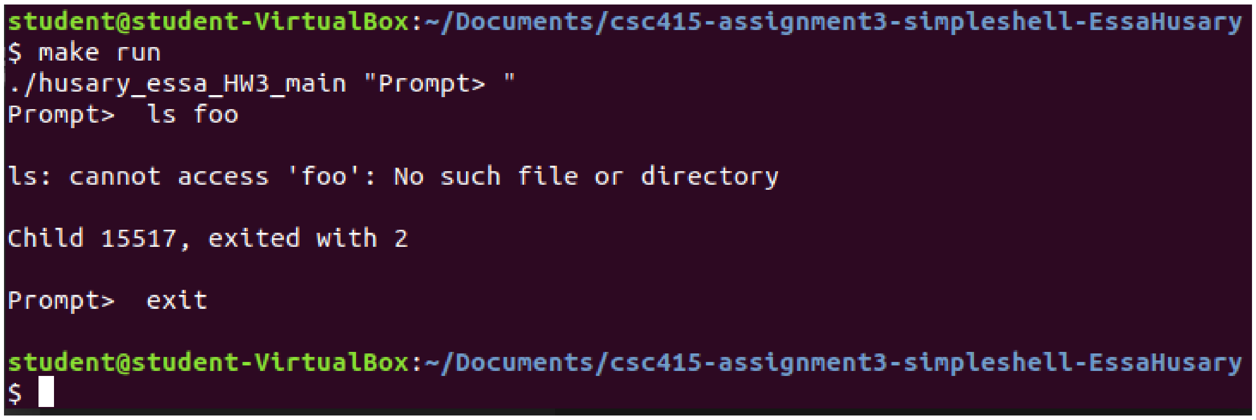
Regular CLI (if I enter the "exit" command, the terminal window will close):



```
student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ ps
  PID TTY          TIME CMD
15436 pts/0    00:00:00 bash
15556 pts/0    00:00:00 ps
student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ 
```

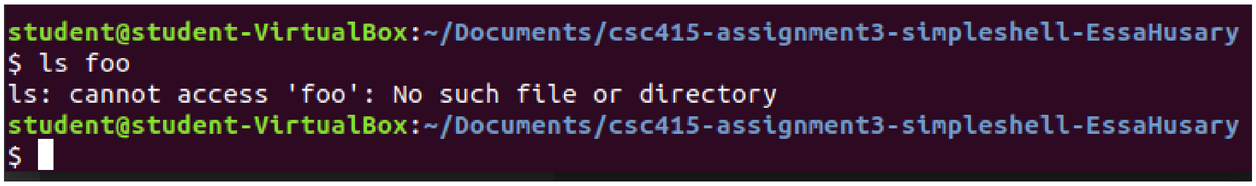
Output #5 – When a directory can't be found:

My shell:



```
student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ make run
./husary_essa_HW3_main "Prompt> "
Prompt>  ls foo

ls: cannot access 'foo': No such file or directory

Child 15517, exited with 2

Prompt>  exit

student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ 
```

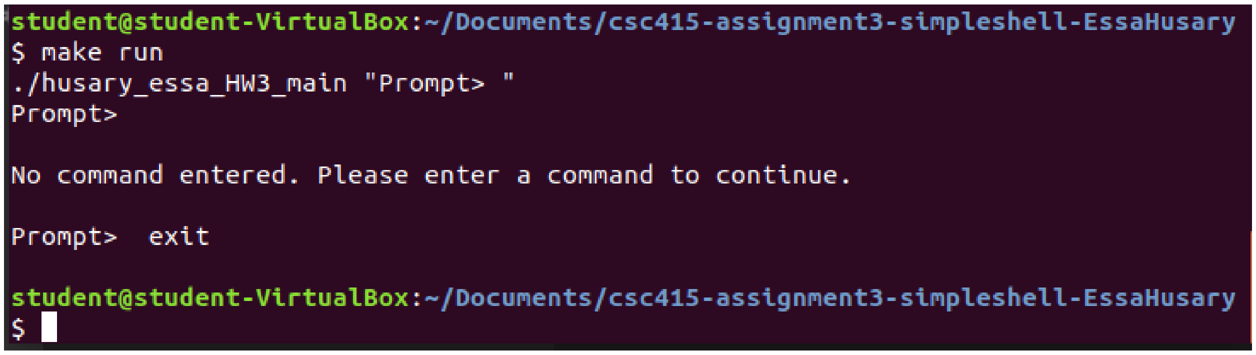
Regular CLI:



```
student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ ls foo
ls: cannot access 'foo': No such file or directory
student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ 
```

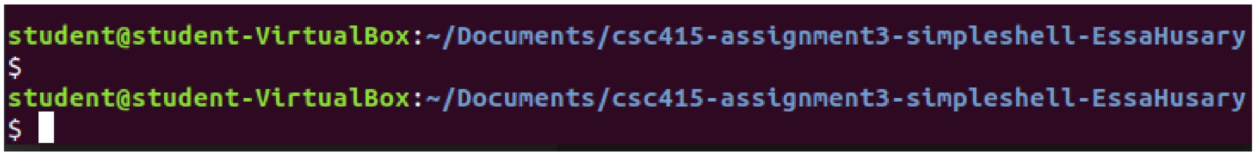
Output #6 – When an empty line is entered (doesn't do anything except request new input):

My shell:



```
student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ make run
./husary_essa_HW3_main "Prompt> "
Prompt>

No command entered. Please enter a command to continue.

Prompt>  exit

student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ 
```

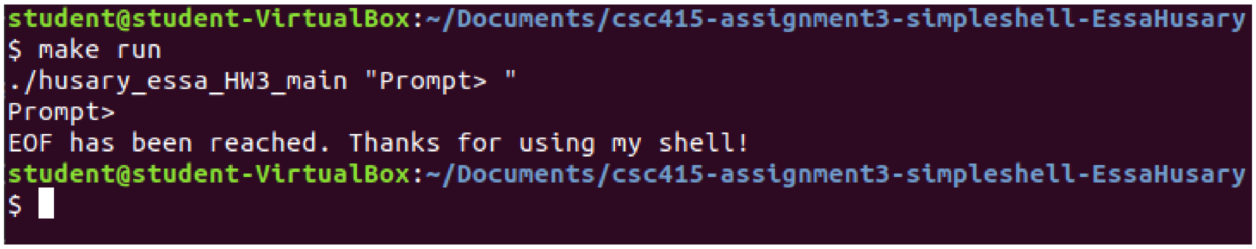
Regular CLI:



```
student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$
student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ 
```

Output #7 – When EOF is reached (CTRL + D):

My shell:



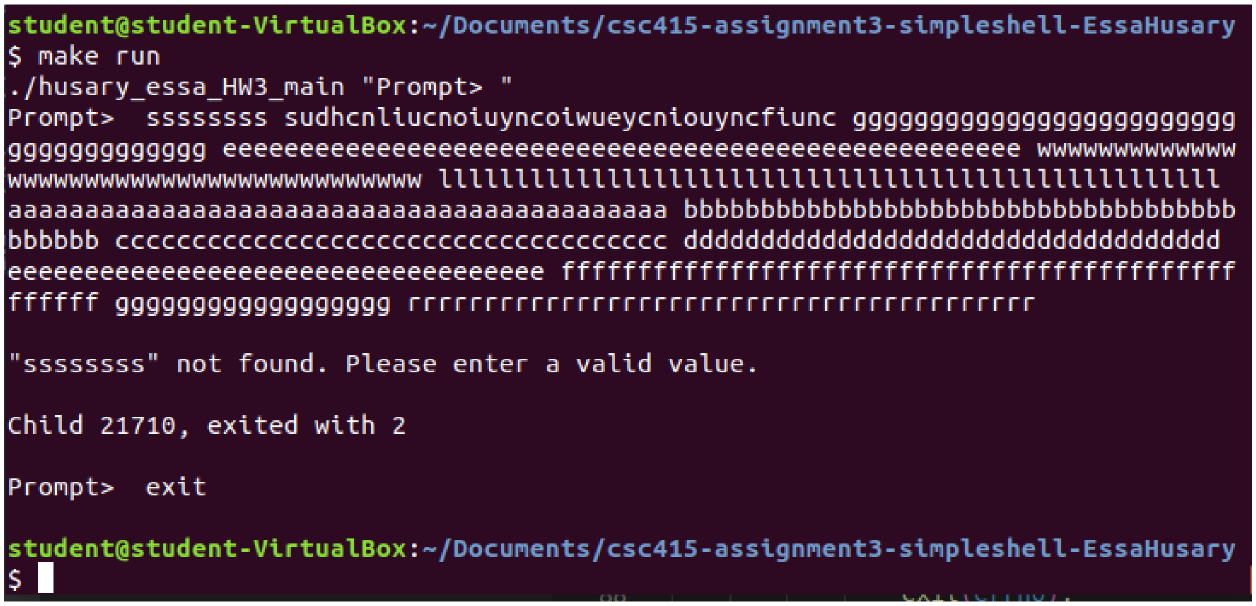
```
student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ make run
./husary_essa_HW3_main "Prompt> "
Prompt>
EOF has been reached. Thanks for using my shell!
student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ 
```

Regular CLI:

Regular CLI will just close if I hit CTRL + D

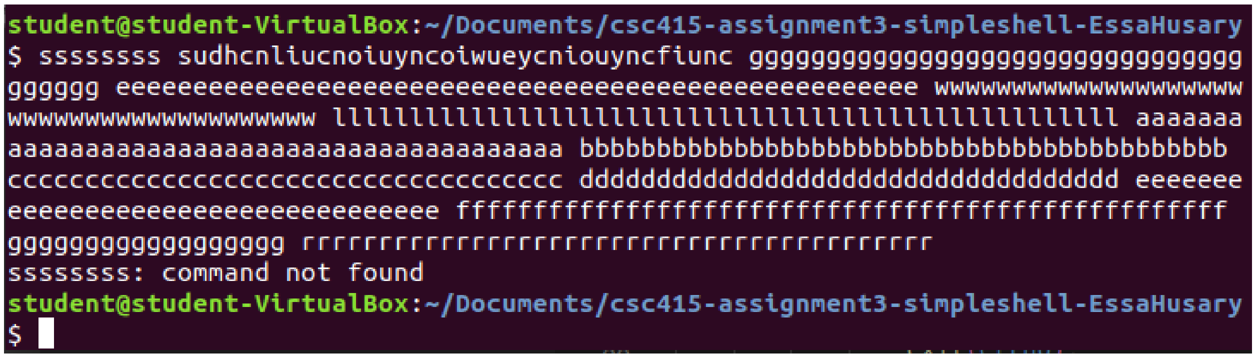
Output #8 – Both when an invalid program name is entered and when a user input is too long (will be truncated):

My shell:



```
student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ make run
./husary_essa_HW3_main "Prompt> "
Prompt>  ssssssss sudhcnliucnoiuyncoiwueycniouyncfiunc ggggggggggggggggggggggggg
ggggggggggggg eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee wwwwwwwwwwwww
wwwwwwwwwwwwwwwwwwwwwwwwwww lllllllllllllllllllllllllllllllllllllllllllllllllll
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbb cccccccccccccccccccccccccccccccccccc ddddddddddddddddddddddddddddddddddd
eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee ffffffffffffffffffffffffffffffffffffffffffff
ffffff gggggggggggggggggg rrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr

"ssssssss" not found. Please enter a valid value.

Child 21710, exited with 2

Prompt>  exit

student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ 
```

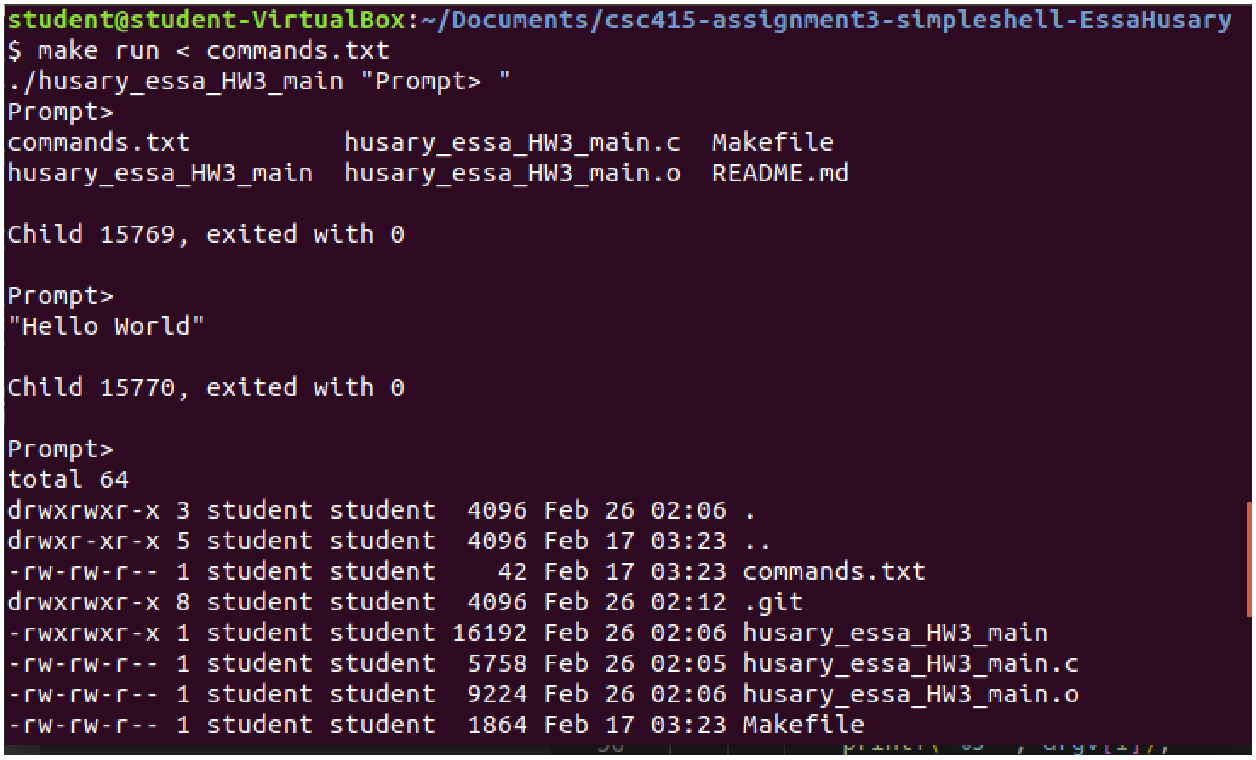
Regular CLI:



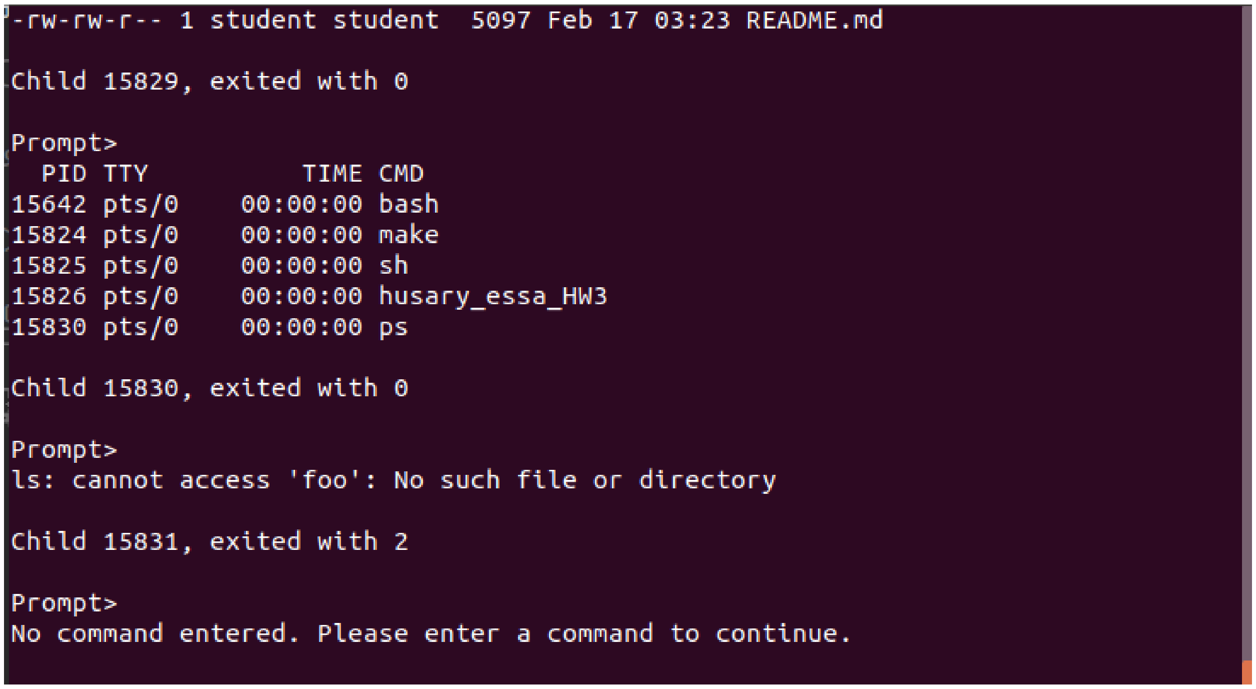
```
student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ ssssssss sudhcnliucnoiuyncoiwueycniouyncfiunc gggggggggggggggggggggggggggggggg
gggggg eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee wwwwwwwwwwwwwwwwwwww
wwwwwwwwwwwwwwwwwwww lllllllllllllllllllllllllllllllllllllllllllllllllll aaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
cccccccccccccccccccccccccccccccccccc ddddddddddddddddddddddddddddddddddd eeeeeee
eeeeeeeeeeeeeeeeeeeeeeeeeeee ffffffffffffffffffffffffffffffffffffffffffffffffff
gggggggggggggggggg rrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr
ssssssss: command not found
student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ 
```

Output #9, the final output – Opening and running the commands.txt file:

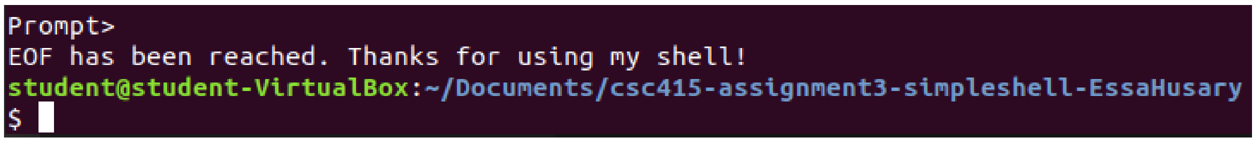
My shell:



```
student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ make run < commands.txt
./husary_essa_HW3_main "Prompt> "
Prompt>
commands.txt          husary_essa_HW3_main.c  Makefile
husary_essa_HW3_main  husary_essa_HW3_main.o  README.md

Child 15769, exited with 0

Prompt>
"Hello World"

Child 15770, exited with 0

Prompt>
total 64
drwxrwxr-x 3 student student  4096 Feb 26 02:06 .
drwxr-xr-x 5 student student  4096 Feb 17 03:23 ..
-rw-rw-r-- 1 student student    42 Feb 17 03:23 commands.txt
drwxrwxr-x 8 student student  4096 Feb 26 02:12 .git
-rwxrwxr-x 1 student student 16192 Feb 26 02:06 husary_essa_HW3_main
-rw-rw-r-- 1 student student  5758 Feb 26 02:05 husary_essa_HW3_main.c
-rw-rw-r-- 1 student student  9224 Feb 26 02:06 husary_essa_HW3_main.o
-rw-rw-r-- 1 student student  1864 Feb 17 03:23 Makefile
```

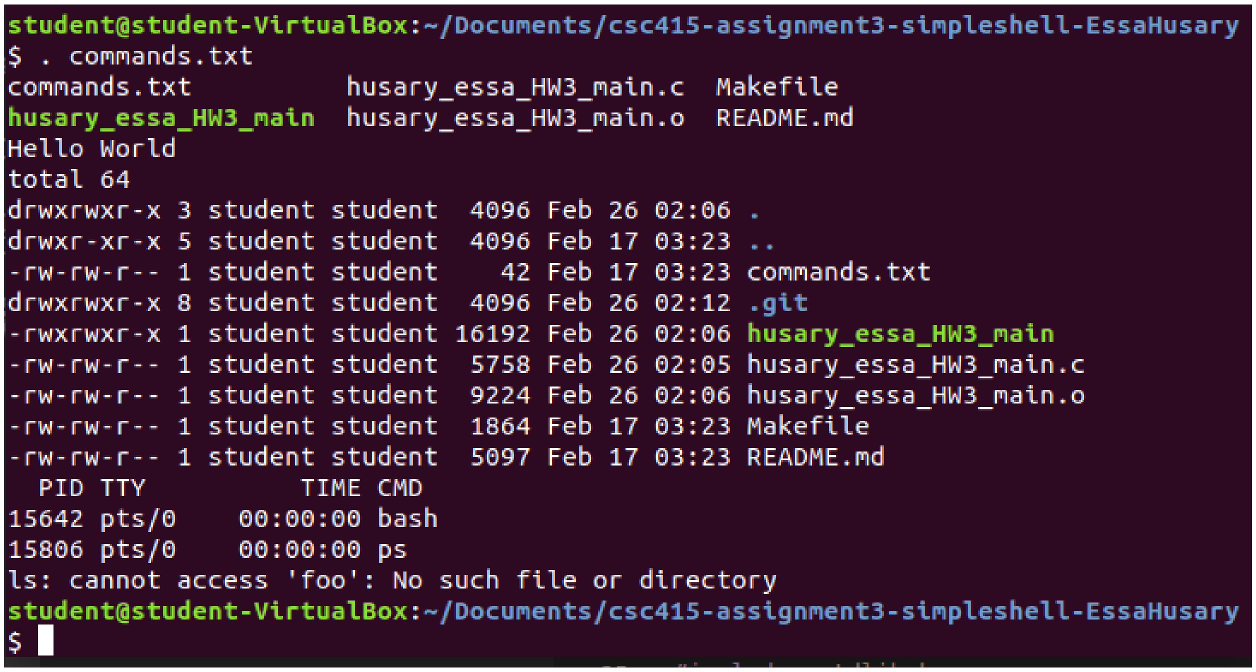


```
-rw-rw-r-- 1 student student  5097 Feb 17 03:23 README.md

Child 15829, exited with 0

Prompt>
  PID TTY          TIME CMD
15642 pts/0    00:00:00 bash
15824 pts/0    00:00:00 make
15825 pts/0    00:00:00 sh
15826 pts/0    00:00:00 husary_essa_HW3
15830 pts/0    00:00:00 ps

Child 15830, exited with 0

Prompt>
ls: cannot access 'foo': No such file or directory

Child 15831, exited with 2

Prompt>
No command entered. Please enter a command to continue.
```



```
Prompt>
EOF has been reached. Thanks for using my shell!
student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ 
```

Regular CLI:



```
student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ . commands.txt
commands.txt          husary_essa_HW3_main.c  Makefile
husary_essa_HW3_main  husary_essa_HW3_main.o  README.md
Hello World
total 64
drwxrwxr-x 3 student student  4096 Feb 26 02:06 .
drwxr-xr-x 5 student student  4096 Feb 17 03:23 ..
-rw-rw-r-- 1 student student    42 Feb 17 03:23 commands.txt
drwxrwxr-x 8 student student  4096 Feb 26 02:12 .git
-rwxrwxr-x 1 student student 16192 Feb 26 02:06 husary_essa_HW3_main
-rw-rw-r-- 1 student student  5758 Feb 26 02:05 husary_essa_HW3_main.c
-rw-rw-r-- 1 student student  9224 Feb 26 02:06 husary_essa_HW3_main.o
-rw-rw-r-- 1 student student  1864 Feb 17 03:23 Makefile
-rw-rw-r-- 1 student student  5097 Feb 17 03:23 README.md
  PID TTY          TIME CMD
15642 pts/0    00:00:00 bash
15806 pts/0    00:00:00 ps
ls: cannot access 'foo': No such file or directory
student@student-VirtualBox:~/Documents/csc415-assignment3-simpleshell-EssaHusary
$ 
```

Works Cited Page:

I learned from and used a snippet of code from page 119 of our e-textbook, *Operating System Concepts*, to create the child process via forking, to check whether we are in the child process or in the parent process (or if process creation has failed), and I used it to learn how to wait for the child process to terminate. I included a screenshot of the example I used from them on the next page of this document. I also used the article titled, "Finding Tokens in a String", from the GNU C Library Manual to learn how to use strtok(), and took inspiration from their examples to write my code to tokenize user input. I included a screenshot of an example I used from them in the final page of this document. And, I used the Linux Programmer's Manual by typing in, for example, "man strtok" into the Ubuntu terminal to learn how to use certain functions such as strtok, fgets, feof, ferror, clearerr, strchr, getchar, strcmp, fork, execvp, wait, and the wait macros.

Citations:

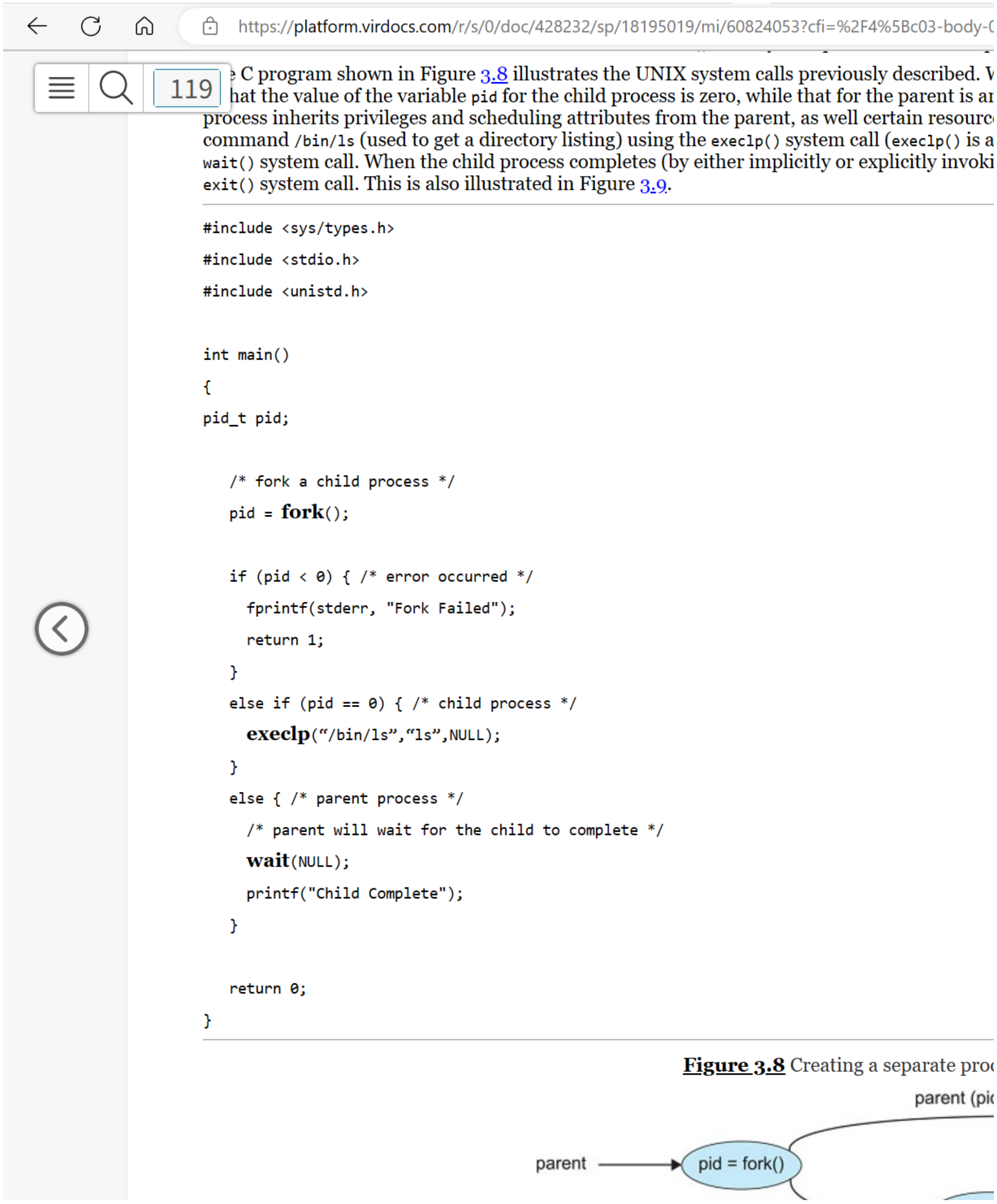
"Finding Tokens in a String." *gnu.org*, GNU's Not Linux. Web. 20 Feb. 2023.
Finding Tokens in a String (The GNU C Library)

"Linux Programmer's Manual." 15 Sept. 2017. (Man pages)

Silberschatz, Avi, et al. *Operating System Concepts*. 10th ed., e-book ed., John Wiley & Sons, Inc., 2018.

(SCREENSHOTS OF CODE SNIPPETS ARE ON THE NEXT COUPLE OF PAGES)

This is the code snippet I referenced from the textbook:

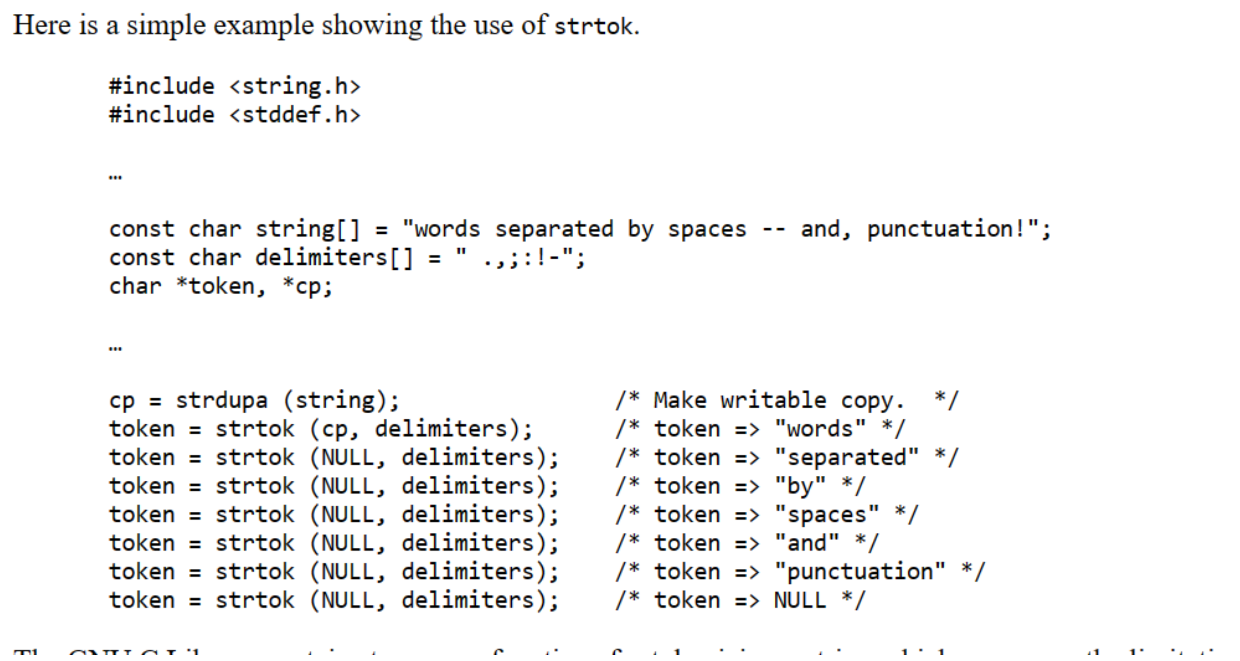

https://platform.virdocs.com/r/s/0/doc/428232/sp/18195019/mi/60824053?cfi=%2F4%5Bc03-body-(
119

e C program shown in Figure 3.8 illustrates the UNIX system calls previously described. V
hat the value of the variable pid for the child process is zero, while that for the parent is ar
process inherits privileges and scheduling attributes from the parent, as well certain resourc
command /bin/ls (used to get a directory listing) using the execlp() system call (execlp() is a
wait() system call. When the child process completes (by either implicitly or explicitly invoki
exit() system call. This is also illustrated in Figure 3.9.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

   /* fork a child process */
   pid = fork();

   if (pid < 0) { /* error occurred */
     fprintf(stderr, "Fork Failed");
     return 1;
   }
   else if (pid == 0) { /* child process */
     execlp("/bin/ls","ls",NULL);
   }
   else { /* parent process */
     /* parent will wait for the child to complete */
     wait(NULL);
     printf("Child Complete");
   }

   return 0;
}
```

Figure 3.8 Creating a separate pro

This is the code snippet I referenced from the GNU C Manual:



Here is a simple example showing the use of `strtok`.

```
#include <string.h>
#include <stddef.h>

…

const char string[] = "words separated by spaces -- and, punctuation!";
const char delimiters[] = " .,;:!-";
char *token, *cp;

…

cp = strdupa (string);                /* Make writable copy.  */
token = strtok (cp, delimiters);      /* token => "words" */
token = strtok (NULL, delimiters);    /* token => "separated" */
token = strtok (NULL, delimiters);    /* token => "by" */
token = strtok (NULL, delimiters);    /* token => "spaces" */
token = strtok (NULL, delimiters);    /* token => "and" */
token = strtok (NULL, delimiters);    /* token => "punctuation" */
token = strtok (NULL, delimiters);    /* token => NULL */
```