Problem 1 -- The argument vector of a shell script

You are logged in to a Unix system and running /bin/bash as your shell. You create a shell script file:

```
$ cat >/tmp/script.sh
#!/bin/sh -vx
ls -l $*
$ # We pressed Control-D here so the file /tmp/script.sh contains the two lines above
$ chmod +x /tmp/script.sh
$ /tmp/script.sh file1.c file2.c
```

When we invoked the shell script with the last line, tell us:

1) What binary executable file gets exec'd (after fork, of course) by the bash shell?

2) What is argc and argv for that binary?

3) Where is the ls command found? (give us the full pathname)

4) What is the argc and argv that the ls program sees?

Problem 2 -- Simple Shell Program

In this assignment, you will write an extremely simplistic UNIX shell which is capable of launching one program at a time, with arguments, waiting for and reporting the exit status and resource usage statistics.

Your shell shall accept lines of input from standard input until EOF. [But see below about being invoked as a shell script interpreter!] Don't worry about issuing a prompt, command-line editing, etc. Each line of input is a command to be executed. For each line:

1) if the line begins with a # (this is known as an octothorpe, or a pound sign, not a "hash tag"!), ignore it.

2) split up the line into tokens (see below) and separate the I/O redirection tokens from the command and its arguments.

3) fork to create a new process in which to run the command

4) establish the requested I/O redirection (if any) in the child

5) exec the command

6) `wait` for the command to complete

7) report the exit status, real time elapsed, user and system time (This information can be obtained by using the `wait3` system call, with `times` system call, or with `getrusage` system call. Be careful of issues of cummulative vs per-child statistics.)

8) If any errors are encountered at any of these steps, report the error, but **do not quit**! Just go on to the next line of input.

Syntax and Tokenization

You may assume that each command is formatted as follows:

```
command {argument {argument...} } {redirection_operation {redirection_operation...}}
```

This is an extreme simplification of shell syntax, but this is after all a course in operating systems, not compilers. The above optional arguments and operators are whitespace-delimited, i.e. they are separated by one or more tabs or spaces. This will simplify parsing and you can use `strtok` to break up the input line into its components. The real shell accepts `command argument>output` as well, but you don't have to parse that if you don't want to. **A line that begins with the # character is a comment and must be ignored.**

Built-in commands

In any shell, the `cd` command must be a built-in. This is because the current working directory is an attribute of any given process. It is changed using the `chdir` system call. But if `cd` were an actual program, typing `cd whatever` into the shell would have no effect, since it would only change it for the child (cd). Therefore, if you see a `cd` command, implement that as a built-in command, and `chdir`. You can also optionally implement the `pwd` built-in command. Also implement a built-in `exit` command which takes an optional single argument. When this

command is received, your shell will exit immediately, with a return value as specified. See "Exit Status" if no value is specified. NOTE: You do not need to implement I/O redirection for the built-in commands.

I/O redirection

Support the following redirection operations (note that pipes are not required since we haven't talked about them yet):

```
<filename          Open filename and redirect stdin
>filename          Open/Create/Truncate filename and redirect stdout
2>filename         Open/Create/Truncate filename and redirect stderr
>>filename         Open/Create/Append filename and redirect stdout
2>>filename        Open/Create/Append filename and redirect stderr
```

Note that a given command launch can have 0, 1, 2 or 3 possible redirection operators. A failure to establish any of the requested I/O redirections should result in an error message and the command should not be launched. You may consider it an error or undefined behavior if a given file descriptor is redirected more than once in a command launch. For best results, the I/O redirection should be done in the child process. This means you could have a child process that fails and exits because of an I/O redirection error, prior to the actual child program being exec'd. That's OK.

Clean File Descriptor Environment

Your shell should fork and exec the command with a standard, clean file descriptor environment. Only file descriptors 0, 1 and 2 should be open, possibly redirected as above. There should be no "dangling" file descriptors.

Example

Note that in the example below, there is "debugging" output. This is permissible, however, all errors and/or debugging must go to standard error, NEVER standard out. Notice that `ls.out` is not polluted with these messages. The informational messages such as the amount of time consumed also go to stderr.

```
$ mysh
cd /some/directory
Changed directory to /some/directory
ls -l >ls.out
Executing command ls with arguments "-l"
Command returned with return code 0,
consuming 0.005 real seconds, 0.002 user, 0.001 system
end of file read, exiting shell with exit code 0
$ cat ls.out
mysh.c
mysh
$
```

Shell Scripts

Your shell must also support being invoked as a script interpreter:

```
#!/absolute/path/to/your/shell
#This is an example of a shell script that your shell must execute correctly
#notice that lines starting with a # sign are ignored as comments!
#let's say this here file is called testme.sh.  you created it with say
#vi testme.sh ; chmod +x testme.sh
#you invoked it with
#./testme.sh
cat >cat.out
#at this point, type some lines at the keyboard, then create an EOF (Ctrl-D)
#your shell invoked the system cat command with output redirected to cat.out
cat cat.out
#you better see the lines that you just typed!
exit 123
#after your shell script exits, type echo $? from the UNIX system shell
#the value should be 123.  Since your shell just exited, the following
#bogus command should never be seen
################################################################################
#!/absolute/path/to/your/shell
#here is another example, say it is called test2.sh
#you invoked it with
#./test2.sh <input.txt
cat >cat2.out
#since you invoked the shell script (via the system shell such as bash)
#with stdin redirected, your shell runs cat which gets stdin from input.txt
exit
#the above exit had no specified return value, so your shell exited with 0
#because the last child spawned, cat, would have returned 0
#again, test this with echo $?
```

In order for this to work, your shell must, if provided with with an argument, open that file and execute each line as a command (ignoring comments).

Exit status

The exit status of your shell itself should be the exit code of the **last child spawned** if your shell exits because it sees end-of-file or the built-in `exit` command with no specified exit value. If the exit is because of an `exit` built-in command with an explicit value, that sets the exit code (see also description under "Built-in commands")

Submission

Your submission must include: source code, "screenshot" demonstrating your shell working in an interactive case (commands from keyboard) and also demonstrating that the cd built-in command works, screenshot showing your shell working as a script interpreter, including source code of shell script used to test this feature. You should also demonstrate I/O redirection.