Problem 3 -- 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 and redirections, waiting for and reporting the exit status and resource usage statistics.

The structure of your shell program is a loop, with the following steps:

1) Read a **line** of input from stdin (or, if your shell is being called as an interpreter, from the supplied script file). You are not restricted to using just the read system call to do this. I would recommend fgets, getline, or similar standard library function. You are not required to print out a prompt like the "real shell" would.

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

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

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

5C) Within the child process, establish any I/O redirection that was specified in the command. Note that a redirection could result in an error, e.g. command <noent.txt where noent.txt is (fittingly) a nonexistent file. If there is an error with any redirection, give a good error message, then cause the child process to exit with a status of 1. This is consistent with how the "real" shell behaves.

6C) Within the child process, exec the command. Note that, just like the real shell, a fully qualified pathname is not required for the command. E.g. if you type just ls that must work. If the exec system call fails, report the error message and exit the child with an exit status of 127. (Again, this matches how the real shell behaves)

5P) Within the parent process, call one of the variants of the wait system call and wait for the child to complete

6P) Report the following information about the command that was just completed: exit status or signal which killed the process (see unit #4 for information about the wait status when the child died by signal), real time elapsed (this is the time between the fork and the wait completing), user CPU time and system CPU time consumed by the child. All time values must be at least millisecond precision. This information can be obtained by using the wait3 system call, with the times system call, with the getrusage system call, or possibly in other ways. Be careful of issues of cummulative vs per-child statistics. Be careful of issues with subtracting time represented in sec+msec or sec+nsec formats.

7P) remember the exit status, which will be needed by the exit built-in command (see below). The exit status is either the WEXITSTATUS part (the most significant 8 bits) of the 16-bit exit status word if the command ended voluntarily, or the least significant 8 bits if the command was killed by a signal.

8P) 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. Quit only when you see EOF at step (1).

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. You can further assume that the redirection operators (if any) will only appear after the command and arguments (if any) as illustrated above. **Note:** A line that begins with the # character is a comment and must be ignored.

Built-in commands

Implement the following built-in commands. Built-in commands do not result in the forking and exec'ing, but are handled directly within your shell. You do not need to implement I/O redirection for built-in commands.

`cd {dir}`: Change the current working directory of the shell with the `chdir` system call. If the directory is not specified, use the value of the environment variable `HOME`.

`pwd`: Display the current working directory. The `getcwd` library function or similar can be used for this. Some interesting cases can arise if you cd to a directory via a symlink. You aren't required to deal with this...just report whatever `getcwd` returns.

`exit {status}`: Exit your shell immediately, using the specified integer status value as the return value from your shell. If no status is given, use the **exit status of the last command** which you spawned (or attempted to spawn). When your shell ends because it has reached EOF on its input, it is as if there were an explicit `exit` command, with no specified exit status, i.e. use the exit status of the last spawned command.

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 you aren't required to support a space between the $<$ or $>$ character and the filename. 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.

AVOID CUT AND PASTE CODING! Yes, there are 5 possible I/O redirection types. This doesn't mean there should be 5 giant blocks of code, cut and pasted from each other, to handle this! Likewise, do not write one blob of 100 lines for an interactive shell and another for a script interpreter.

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 in the exec'd child, possibly redirected as above. There should be no "dangling" file descriptors. This means "good housekeeping" when you perform I/O redirection, and don't forget about the script input file if applicable. You can assume that *your* shell was invoked the by "real" shell with such a clean environment.

Example

Note that in the example below, the informational messages such as the amount of time consumed and the exit status of the command are going to stderr, not stdout. That is the proper place for them, as well as for any "debugging" output that you leave in your code. Note that `ls.out` in the example is not "polluted" by these messages.

```
$ ./mysh              #I am invoking my shell from the real shell
#These lines below are commands that I type into my shell
cd /some/directory
pwd
/some/directory
ls -l >ls.out
#These informational lines are printed by my shell to stderr
Child process 11628 exited normally
Real: 0.002s User: 0.001s  Sys: 0.001s
#Now I am examing the output file that the ls command created
cat ls.out
ls.out
mysh.c
mysh
#More informational lines from the cat command execution
Child process 11629 exited normally
Real: 0.010s User: 0.001s  Sys: 0.001s
#Here is an example of an ls command that will exit with a non-zero value
ls asfljsakfjasklf
ls: cannot access asfljsakfjasklf: No such file or directory
Child process 11630 exited with return value 2
Real: 0.001s User: 0.000s  Sys: 0.000s
#This is an example of my invoking a program that dies with a SIGSEGV
#Note the exit code is 139 (128 is core dump + the signal number 11)
 ./dumpcore
Child process 11631 exited with signal 11 (Segmentation fault)
Real: 0.001s User: 0.000s  Sys: 0.000s
#At this point, I press Control-D on the keyboard to exit my shell
#The informational message below appears on stderr, not stdout
end of file read, exiting shell with exit code 139
#And this is my checking the exit code that my shell passed back to the real shell
$ echo $?
139
```

Shell Scripts

Your shell must also support being invoked as a script interpreter: In order for this to work, your shell must, if provided with with a single argument, open that file and execute each line as a command (ignoring comments). (Open the file directly and read from it. Do not redirect standard input to be the script file, because you will then lose the real standard input for any spawned commands.) I will provide a few sample scripts that your simple shell should be able to handle, along with instructions on how to test.

Exit status

Re-read the description of the `exit` built-in command if you are unsure what exit status your shell should return.

Submission

Your submission must include: source code, "screenshot" or text session grab showing your shell running. In particular, show your runs against the supplied shell script test cases.