Project 1

Out: Friday September 18
Due: Thursday October 8, 11:59pm

In this project you will create your own shell. Projects must be written in C or C++. As you will see, some aspects of the project are not specified; it will be your responsibility to take whatever steps you feel are appropriate to complete the task.

Please start early! This project requires a lot of time!

---

wsh Overview

The shell you will implement will be quite functional, but will lack most of the fancier features normally found in a shell. Like more powerful shells, your "Wimpy SHell" will use a command line interface, and will support foreground (interactive) processing, background (batch) processing, redirection, pipes, and job control. Your shell must support only a very small number of built-in commands (see below for details). Except for the built-in commands, all commands issued to your shell will be executed by creating a new process (using `fork`) and loading the appropriate executable (using `execvp` or `execlp`). Commands in a pipeline will require multiple processes!

You will need to use the following system calls. For more information, consult sections 2 and 3 of the `man` pages.

- fork
- execvp or execlp
- waitpid
- exit
- open
- close
- dup2
- pipe
- chdir
- getcwd

Required Features

5 points: makefile

Yes, you get 5 points for simply using a makefile. Name your source files whatever you like. Please name your executable **wsh**. Be sure that "make" will build your executable.

10 points: Documentation

If you have more than one source file, then you must submit a **Readme** file containing a brief explanation of the functionality of each source file. Each source file must also be well-documented. There should be a paragraph or so at the beginning of each source file describing the code; functions and data structures should also be explained. Basically, another programmer should be able to understand your code from the comments.

5 points: Main loop

The main loop of your shell is quite simple:

1. Write the prompt "wsh: " to standard output.
2. Read a command line from standard input. You may limit the number of characters that may be typed on a command line to 256.
3. Execute the command. Your shell must handle empty lines and unknown commands gracefully (i.e., without dumping core, spawning another shell, etc).
4. Show the status of all background jobs, if any (see below).
5. If the end of standard input has not yet been reached, go to (1).

15 points: Foreground jobs

A "foreground job" is a command of the form `ARG0 ARG1 ... ARGn` . Examples:

```
cp file1 file2
cp -R dir1 dir2
rm -f -R dir1 dir2 file1 file2
xterm
```

A foreground job should be executed as follows:

1. Create a new process using `fork`
2. Replace the image of the created process with the desired command using `execlp` or `execvp`.
3. Wait for the process to terminate using `waitpid`.

20 points: Background jobs

A "background job" is a command of the form `ARG0 ARG1 ... ARGn &` . Examples:

```
cp file1 file2 &
cp -R dir1 dir2 &
rm -f -R dir1 dir2 file1 file2 &
xterm &
```

A background job is just like a foreground job, except the shell **does not** wait for the process to terminate. Thus, a background job should be executed as follows:

1. Create a new process using `fork`
2. In the newly created process, replace its image with the desired command using using `execlp` or `execvp`.

Unlike foreground jobs, your shell must keep track of background job information using an appropriate data structure. This allows the shell to display the status of each background job (and is required for other features described later). Each background job should be assigned a "job number", to be used as an index when displaying the information about the job. A background job's number should remain fixed for its life. The job numbers should be incremented for each new background job, and when no background jobs exist, the job number should reset to 1.

As mentioned earlier, the main loop of your shell must display the status of all background jobs. This is done by displaying two lists: the list of background jobs still running, and the list of background jobs that have finished execution since the last time the status was displayed. The lists should contain the job number and the command line for each job. For example:

```
Running:
    [1] cp file1 file2
    [2] cp -R dir1 dir2
    [4] xterm
Finished:
    [3] rm -f -R dir1 dir2 file1 file2
```

If the xterm is then closed, the next status report might be:

```
Running:
    [1] cp file1 file2
    [2] cp -R dir1 dir2
Finished:
    [4] xterm
```

(assuming file1 is very large, as is dir1). Note that job [3] is not listed the second time. You should clear out the background job information from your jobs data structure after a job has been displayed in the "Finished" list.

15 points: Redirection of standard input and output

Both background and foreground jobs should be able to redirect standard input and output using "<" and ">", respectively. For example,

```
ls -a -l -F > file1
ls -a -l -F > file2 &
```
should send the output of ls to file1 and file2,

```
cat -n < file1
```
should read from file1 instead of standard input, and

```
cat < file1 > file2
cat > file3 < file1
```
should read from file1 and write to file2 and file3.

Your shell must recover gracefully if an input file does not exist. For example, this is "graceful":

```
wsh: cat < bogusfile
cat: Input file error
wsh:
```

but something like this is NOT graceful:

```
wsh: cat < bogusfile
Segmentation fault - core dumped
pyrite>
```

If output files exist already, you may quietly overwrite them, or "gracefully" refuse.

10 points: Built-in commands

The main difference between your shell and an industrial-strength shell is the lack of built-in commands. Your shell must handle the following commands internally, without creating a new process:

- `cd newdir`
  Change the current working directory to "newdir", then display the current working directory. "newdir" may be either an absolute or relative path. Do something graceful (i.e., don't dump core) if "newdir" is illegal.
- `wait jobnum`
  Wait for background job with specified job number to terminate. Do something graceful (i.e., don't dump core or block indefinitely) if jobnum is an illegal job number.
- `exit`
  Exit the shell.

20 points: Pipes

Your shell must be able to handle a command pipeline of the form `job1 | job2 | ... | jobn` , where each job is a sequence of arguments (like a foreground job). Examples:

```
cat -n file | less
ls -alF | sort -n -r +4 | head -5
```

```
cat -n file1.cc file2.cc | grep #include | sed -e s/include/exclude/ | head -
5
yes + | head -15 | cat -n | sed s/10/10x/ | tr -d \n | tr x \n | head -1 | bc
```

A pipeline should be executed as follows (not necessarily in this exact order):

- Create a new process for each job using `fork`.
- Replace the image of each process with the appropriate executable using `execlp` or `execvp`.
- Create n-1 pipes using `pipe`.
- Re-direct the output of job i to the write end of pipe i.
- Re-direct the input of job i to the read end pipe i-1.
- Keep the input of job1 as standard input.
- Keep the output of jobn as standard output.
- Close any unnecessary file descriptors.
- Wait for all jobs to terminate.

Example

```
pyrite>  wsh
wsh: ls -aF
./  ../  wimpy.cc  wsh*
wsh: xterm &
    Running:
        [1] xterm
wsh: emacs &
    Running:
        [1] xterm
        [2] emacs
wsh: echo thisqisqaqtest > file1
    Running:
        [1] xterm
        [2] emacs
wsh: cat file1
thisqisqaqtest
    Running:
        [1] xterm
    Finished:
        [2] emacs
wsh: tr q \n < file1 > file2 &
    Running:
        [1] xterm
        [3] tr q \n < file1 > file2
wsh:
    Running:
        [1] xterm
    Finished:
        [3] tr q \n < file1 > file2
wsh: cat file2
this
```

```
is
a
test
    Running:
        [1] xterm
wsh: wait 1
waiting for xterm
    Finished:
        [1] xterm
wsh: ls -aF
./  ../  file1  file2  wimpy.cc  wsh*
wsh: yes | head -10 | wc
     10      10      20
wsh: yes|head -10|wc
     10      10      20
wsh: emacs &
    Running:
        [1] emacs
wsh: wait 1
waiting for emacs
    Finished:
        [1] emacs
wsh: cd ..
Working directory /home/cs352
wsh: cd /
Working directory /
wsh: cd /home/cs352
Working directory /home/cs352
wsh: ls -F
Project1/  Project2/  Project3/
wsh: cd Project1
Working directory /home/cs352/Project1
wsh: ls -aF
./  ../  file1  file2  wimpy.cc  wsh*
wsh: rm file1 file2
wsh: exit
pyrite>
```

Things you don't need to implement

When testing your shell, you probably will discover that certain things do not work like they would in an industrial-strength shell. Here is a list of notable features that you do NOT need to implement.

- Background pipelines (e.g., `yes | head -5 &`).
- Pipelines mixed with redirection to and from files (e.g., `cat -n < file1 | head -5 > file2`).
- Tab completion of filenames.
- Wildcard characters * and ? (Try `ls *.cc` in your shell...)

- Quotes (', "). Normally, quotes will protect special characters like , |. (Try `echo "This probably > will break | your shell"`). Also, a quoted string normally counts as a single argument.
- Escape characters. Normally, special characters can be protected by "escaping" them. (Try `echo \> hi`).
- Environment variable usage. (Try `echo $PWD`).
- ~ for your home directory. (Try `ls ~`).

Submitting Your Project

You will submit your project on Blackboard. Your program must compile and run without errors on pyrite.cs.iastate.edu.

Put all your source files (including the makefile and the README file) in a folder. Then use command

```
zip -r <your ISU Net-ID> <src_folder>
```

to create a .zip file. For example, if your Net-ID is `ksmith` and `project1` is the name of the folder that contains all your source files, then you will type

```
zip -r ksmith  project1
```

to create a file named `ksmith.zip` and then submit this file.

Q & A

Q: If user types exit while background commands are running, should they be terminated immediately? Should we print a final time?
A: You should wait for the background commands to terminate, and then terminate the shell. You should print a list of background commands that have terminated before you terminate the shell.

Q: What is the desired behavior if one launches an interactive command that doesn't redirect input as a background command?
A: You can assume that a user never does this.

Q: How do we decide what numbers to assign to a background process, especially when there are gaps (like Running: [1] [3]), and then another background job need to be added.
A: As long as there exists one or more background jobs, the job numbers should be incremented for each new background job. When no background jobs exist, the job number should reset to 1. For example, suppose we have 4 background jobs running with job numbers 1, 2, 3, 4. Now jobs 2, 3, 4 terminate and then a new background job is submitted. The new job will have job number

5 and the running list contains jobs 1 and 5. When jobs 1 and 5 both terminate, the job number will reset to 1 so that a new background job will get job number 1.

Q: When I try and run emacs in the background, it doesn't work at ALL, it really fritzes out. Is it supposed to work in the background, like the spec has in the example?
A: In order to run emacs in the background, you need to setup X forwarding from the "pyrite" server to your workstation. Click here to learn how to X forward. We will not type ‘emacs &’ when testing your shell.