______ / 50 points

DUE ON BGUNIX
MONDAY MARCH 27, 2017 11:59PM
NO LATE WORK ACCEPTED

In this project, you will create and turn in your first program on BGUnix. Please see the documentation below for instructions on obtaining and configuring a BGUnix account:

http://www.bgsu.edu/arts-and-sciences/computer-science/cs-documentation.html

Once you have an account, SSH into BGUnix and join the class by issuing the command:

```
$ class -join cs4080
```

This should create a directory in your home directory named 'cs4080'. All your work this semester must occur inside this directory. Any required files will be automatically collected from this directory at the due date/time for the assignment!

Working in Pairs

For this project, you may work individually or in a pair. If you work in a pair, you have the option of breaking your pair (if things are not going well) but at that point you must continue individually.

In order for me to know what happened, if you start working in a pair you must document that at the TOP of your sssh.cpp file in a comment. State who you worked with, and provide a breakdown of percent effort (must add to 100%). If you wind up breaking your pair, note how far you made it through the project before splitting.

Both students must submit the (same) code!

When working in pairs, both students must write code! You should frequently (~30 mins) change roles between driver and navigator. I fully expect both students to be capable of explaining the code to me and I reserve the right to pull students in (individually) to have them explain their solution. Failure to explain will result in severe grade penalty.

Program Details

In this project, you will create your own shell (e.g. sh/bash/csh/ksh). We will call our shell the Super Simple SHell (sssh). It must be written in C++ (version 11).

The shell has a very limited set of features. It must accept a single line of input (you can limit the length of that line arbitrarily to 255 characters). Each line represents a job. A job consists of a command, or a set of commands piped together. Your shell must support the following features:

- **Main Loop**
  - write the prompt "sssh: " to standard out
  - read a single line of input
  - execute the command(s) from the input
    - handle invalid commands without crashing
    - handle empty string gracefully (do nothing)
  - display status of any background jobs
- **Built in Commands**
  - `exit` - this will exit your program
  - `cd dir` - changes the working directory to the specified directory
  - `wait num` - waits for the background job specified by num
- **Foreground jobs**
  - These are commands of the form `cmd arg0 arg1 arg2 ...`
  - Some examples:
    - `ls -l`
    - `xterm`
    - `cp file1 file2`
    - `rm -Rf dir`
  - Your shell should call `fork()` (or `vfork()`), then an `exec*()`, and finally `waitpid()`
  - It must respect the **`PATH`** environment variable to find the executable (i.e., if I type `ls` it actually runs `/bin/ls`)
- **Background jobs**
  - These are commands of the form `cmd arg0 arg1 arg2 ... &`
  - Some examples:
    - `ls -l &`
    - `xterm &`
  - Your shell should call `fork()` (or `vfork()`), then an `exec*()` but not `waitpid()`
  - Must maintain a list of all background jobs and update the list periodically
  - Every background job given a unique number, that never changes for the life of the job
  - Job numbers start at 1, and if no more background jobs the numbers reset back to 1
  - Main loop must show status, including list of still running background jobs and a list of any finished background jobs
  - Example, assuming there were 4 background jobs the main loop might display:

```
Running:
      [1] xterm &
      [2] sleep 80 &
      [4] cp -R * ../dir &
Finished:
      [3] sleep 5 &
```

Then if the xterm is closed, the next time the main loop displays the status it looks like:

```
Running:
      [2] sleep 80 &
      [4] cp -R * ../dir &
Finished:
      [1] xterm &
```

Notice that job 3 is not listed the second time, as it was already finished and cleared out of the list.

- **Pipes**
  - Commands of the form `job1 | job2 | job3 | ...` where each job is like a foreground job, a command followed by args
  - Takes the output of a job and feeds it as input to the next job, where jobs are separated by a vertical pipe character |
  - Some examples:
    - `cat file1 | less`
    - `ls -1 | sort`
    - `find . -type f | sort | uniq`
  - If there are *n* jobs, there should be *n-1* pipes. The first job's input is still standard input and the last job's output is still standard output.
  - Does not have to work in conjunction with background jobs

Extra Credit

You can earn up to 50 points of extra credit on this project by implementing extra features. Note that people working in pairs will receive **half** the extra credit points listed! To earn the extra credit, the feature must work without any errors. If the feature is implemented but buggy, you will receive 0 extra credit points for that feature.

- [10 pts] **Redirection to/from files**
  - Replaces standard in/out with a file stream
  - Standard in replaced via command: <file
  - Standard out replaced via command: >file (always overwrites file if it exists)
  - `sort <file1 >file2`
- [10 pts] **Background pipes**
  - combine pipes with background jobs
  - `cat file | sort | uniq &`
- [5 pts] **Tab completion**
  - Ability to hit <TAB> to complete filenames
  - If more than 1 possible completion, should show a list of possibilities
- [10 pts] **Handling Wildcards**
  - Ability to use ***** and **?** wildcards in arguments
- [5 pts] **Command History**
  - Ability to use **!!** to run the last command
  - `sudo !!` (runs last command as sudo)
- [5 pts] **Multiple foreground commands**
  - Using semicolon to separate multiple foreground jobs
  - `ls ; cd .. ; rm -Rf *`
- [5 pts] **Pipelines and file redirection**
  - Mixing file redirection and pipes
  - `sort < file | uniq > file2`
- [5 pts] **Environment variables**
  - Ability to reference environment variables in commands
  - E.g., `echo $PATH`

Documentation

Your code should include proper documentation/comments where appropriate. A portion of the grade is attributed directly to documentation (see the grading rubric provided on Canvas).

Every source file must contain a header block similar to the following:

```
// File:        name of the file
// Description: purpose of the file
// Author:      Your Name
// Course:      CS4080, Spring 2017
```

Functions should be proceeded by a comment block documenting the function's purpose.

Blocks of code should contain comments describing what that block is contributing to the overall solution.

What to Turn In

The automated turn-in collection script will save these files from your class directory:

- Makefile
- sssh.cpp
- sssh.h
- job.cpp
- job.h
- process.cpp
- process.h

The script will simply run 'make' to compile your shell. It must work with no arguments. You are free to use any library available on BGUnix, just be sure your Makefile includes it.