

EE461S - Operating Systems

Project 1: yet another shell (yash)

Due Date: Tuesday, 2/15 at 11:59pm

Table of Contents

Preparation	1
Objective	1
Details	3
Features	3
Restrictions on the input	4
Restrictions on programming environment	5
Testing	5
Workload Estimation	5
Submission	5
Lab 1 FAQ	7

Preparation

Sources to refer to in doing this lab are:

1. The systems programming book (see Canvas for link)
2. Class lecture notes
3. Youtube videos in this [playlist](#)

Objective

In this lab you will be introduced to both the command line interface and the Unix programming environment. You will write a command line interpreter known as a **shell** that takes commands from standard input and executes the commands by creating processes. You will also do the whole lab **by yourself**. The future Pintos labs (Labs 2-4) will be in teams of **two**. Also, there is no starter code, so try to start the lab as early as possible.

You will need some version of Linux that has a bash shell for this lab. This means you could use a Virtual Machine with an image of Linux (e.g. Ubuntu 16.04), your Mac's terminal, Windows Subsystem for Linux (WSL), or the ECE Linux machines (LRC machines). Note that the LRC machines do *not* have readline, so many of you will probably want to use a virtual machine (like VirtualBox).

WARNING: We will test your code on an actual Linux machine, so if you do choose to develop on Mac or WSL, make sure your code works on a Linux machine/VM as well.

WARNING: A student last year found that WSL didn't work for signals (for him).

Don't worry about installing Pintos right now, but know that Pintos will require an actual Linux machine. This means WSL and Mac won't work with Pintos. Additionally, the LRC machines won't work because they don't give you sudo privileges (so you can't install anything on them). This means your easiest option is to just get a virtual machine (e.g. [virtual box](#)) and use a version of Linux that doesn't cause issues with Pintos (e.g. [Ubuntu 16.04](#)).

Details

A standard shell like bash/tcsh/csh etc. has a rich set of features that it supports. We picked a subset of these features for you to implement.

Preparing to Code: First, you should exercise all of these features in a shell like bash. Once you understand how to use them you will get a sense of how you can implement them.

Features

Here is the complete list of features you must implement:

- File redirection
 - with creation of files if they don't exist for output redirection
 - fail command if input redirection (a file) does not exist
 - `<` will replace stdin with the file that is the next token
 - `>` will replace stdout with the file that is the next token
 - `2>` will replace stderr with the file that is the next token
 - A command can have all three (or a subset) of the redirection symbols
- Piping
 - `|` separates two commands
 - The left command will have `stdout` replaced with the input to a pipe
 - The right command will have `stdin` replaced with the output from the same pipe
 - Children within the same pipeline will be in one process group for the pipeline
 - Children within the same pipeline will be started and stopped simultaneously
 - **You are only required to support (at most) one `|` per command**
- Signals (`SIGINT`, `SIGTSTP`, `SIGCHLD`)
 - `Ctrl-c` must quit current foreground process (if one exists) and not the shell and should not print the process (unlike bash)
 - `Ctrl-z` must send `SIGTSTP` to the current foreground process and should not print the process (unlike bash)
 - The shell will not be stopped on `SIGTSTP`
- Job control
 - Background jobs using `&`
 - `fg` must send `SIGCONT` to the most recent background or stopped process, print the process to stdout, and wait for completion
 - `bg` must send `SIGCONT` to the most recent stopped process, print the process to stdout in the jobs format, and not wait for completion (as if `&`)
 - `jobs` will print the job control table similar to bash. *HOWEVER there are important differences between your yash shell's output and bash's output for the jobs command! Please see the FAQ below.*
 - with a [`<jobnum>`]

- a + or - indicating the current job. Which job would be run with fg is indicated with a + and all others with a -
 - a "Stopped" or "Running" indicating the status of the process
 - and finally the original command
 - e.g:
 - [1] - Running sleep 5 &
 - [2] - Stopped sleep 5
 - [3] + Running running_command | grep > out.txt &
 - Terminated background jobs will be printed after the newline character sent on stdin with a Done in place of the Stopped or Running.
 - A job is all children within one pipeline as defined above
 - Max number of jobs running at the same time: 20
- Misc
 - Children must inherit the environment from the parent
 - Must search the PATH environment variable for every executable
 - All child processes will be dead on exit
 - The prompt **must** be printed as a "# " (pound sign with a space after it) before accepting user input.
 - **Make sure that your shell doesn't segfault (for whatever reason) when you run something like this because our grading script uses the 'printf' command with the '%b' format.** If you see the quotation marks at the beginning and end, that's fine (and expected, unless you do fancy parsing).
 - `printf(%b "\033[34m Blue \040 text \033[0m\n")`
 - `ls`

Restrictions on the input

These restrictions will help you simplify the parsing of the command line:

- Everything that can be a token (<, >, 2>, etc.) will have a space before and after it. Also, any redirections will follow the command after all its args.
- If a command has a & symbol (indicating that it will be run in the background), then the & symbol will always be the last token in the line.
 - **Valid** examples
 - `ls &`
 - `sleep 4 | sleep 6 &`
 - **Invalid** examples
 - `sleep 4 & | sleep 6 &`
 - `sleep 4 & | sleep 6`
- Each line contains either one command or two commands in one pipeline
- Lines will not exceed 2000 characters
- Each token will be no more than 30 characters
- All characters will be ASCII
- `Ctrl-d` will exit the shell

Restrictions on programming environment

- All code will be in C (ANSI, C99, GNU99, C11, etc.) (No C17)
- Code may only include headers from the operating system and the GNU C stdlib. Use of the `system` library call is not allowed. `readline()` is allowed.
- Stick to the GNU C stdlib! Code from the BSD stdlib (e.g. on macOS) might not work on Linux.
- All code will run on GNU/Linux (it will be tested on x86-64)

Testing

- See the grading rubric on Piazza for how your submission will be graded (this will only get posted on the last day of submission)

Workload Estimation

- Students often make the mistake of finishing input parsing and file redirection and then thinking the rest of the lab will be similar in difficulty ... then they spend the last day of office hours panicking at the TA
- Here's an estimate of how long you'll spend on each part of the lab
 - Input parsing (5%)
 - File redirections (5%)
 - Programming in C (incorrectly using malloc, incorrectly using pointers, segfaults, etc.) (10%)
 - Pipes (30%)
 - Signals (30%)
 - Jobs (20%)

Submission

You will submit a single file named **yash.tgz** that contains all of your files in a folder by the same name (**yash**). Make sure there is a **Makefile** that will build your shell with a single command (`make`). The executable that your makefile should create must also be called **yash**. If you have any doubts about submission please ask the TAs or Dr. Y. It is important that everybody follows these instructions so we can automate the grading process. The screenshots below show how to create the tarball and what it should look like when you untar it and type `make`.

```

~/yash >>> ls
Makefile  yash.c
~/yash >>> cd ..
~ >>> tar czvf yash.tgz yash
a yash
a yash/Makefile
a yash/yash.c
~ >>>

~ >>> tar xf yash.tgz
~ >>> cd yash
~/yash >>> ls
Makefile  yash.c
~/yash >>> make
gcc -o yash yash.c
~/yash >>> ls
Makefile  yash*  yash.c

```

Note: there are penalties for improperly formatted submissions (e.g. incorrect file/folder names, incorrect directory structure, no tar file, etc.)

Lab 1 FAQ

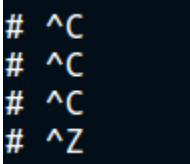
1. If given an invalid command, simply ignore it and print a new line.

```
$ ./yash
# ls
Makefile yash yash.c
# asdf
# asdf
# random_command_that_doesnt_exist
# ls
Makefile yash yash.c
#
```

- a. An "invalid command" in that case does not mean a command that outputs error to your terminal. Otherwise why would we even make you implement error redirection (2>)?
 - b. An "invalid command" is more like a program that doesn't exist on your computer so `execvp()` cannot execute it. An example of expected behavior for an invalid command is shown above. Notice how the invalid commands output nothing to stdout except a newline character to move the prompt "#" to a new line. The 'ls' command is a known command and is therefore valid.
2. Inputs will start with commands and not <, >, 2>, &, |. Here are some examples
 - a. Invalid commands
 - i. < ls
 - ii. > ls
 - iii. 2> ls
 - iv. | ls
 - v. & ls
 - b. Valid commands
 - i. ls > out.txt
 3. If there is a conflict between bash, zsh, fish, or any other shell's behavior for a command, please go with bash's output. Meaning, your yash shell should try to follow what bash does except when specified otherwise.
 - a. For example, the jobs command gives different output for each of the two shells when a situation occurs such that the jobs are numbered 1, 2, 4 and then a new job is created. Zsh gives the new process the job number 3, whereas bash gives the new process the job number 5. Since we defer to bash, your yash shell should give the new process the job number 5
 4. Which libraries are allowed? The lab document says the following: "code may only include headers from the operating system and the GNU C stdlib. Use of the system library call is not allowed."

- a. This means that you cannot use the `system()` function provided in the `<stdlib.h>` library. This is because the function can execute any command that can run on the terminal and is allowed by your operating system. Meaning, that one function call can do your entire lab. Since we are trying to make you learn, we don't want you to use this function. Other than this rule, most libraries allowed (e.g. `<stdio.h>`, `<string.h>`, `<stdlib.h>`, etc.).
 - b. Note: `<malloc.h>` may exist on some BSD and Darwin systems, but it is deprecated and won't work on Linux. Use `<stdlib.h>`.
5. We don't expect you to implement more than we ask for. This includes implementing tab-completion or multiple pipes. If you want to add cool things to your shell to make it more realistic then please go ahead and let us know in the submission comment for Canvas. We'd love to check it out. However, just know that you could be spending your time getting ahead on the Pintos labs instead.

6. It is okay if your yash shell prints `^Z` when you press `Ctrl-Z` and `^C` for `Ctrl-C`.



```
# ^C
# ^C
# ^C
# ^Z
```

7. We won't test your shells with `top`, `apt install`, `cd`, `history`, `vim`, or `man`.
8. Make sure to remove the line feed character (`'\n'`) from the input before parsing it (if you're using `readline`, then this might not be a problem). Since you have to press the enter key to send the input, the input command will have a line feed character at the end. If you don't remove this character before sending your tokens to `execvp()`, then your yash shell won't be able to recognize the command.
 - a. For example, let's say the input is `"ls"` and then you hit enter to send the command to your shell
 - b. Your C program will say the input is `"ls\n"`. If you then send this to `execvp()`, it will try to look for a program called `"ls\n"` and then try to execute it. This won't work because there is no executable called `"ls\n"`.
 - c. Therefore, you have to remove the `'\n'` by replacing it with a NULL character (i.e. find the `'\n'` and then replace it with `'\0'`).
9. When creating files with output or error redirection, make sure to set the permission bits correctly or else you won't be able to access the file (if you don't then you'll get a "permission denied" error when you try to read from the created file)
 - a. Take a look at the [open function](#)
 - b. The first argument is the name of the file to create
 - c. The second argument is a bitwise OR of flags specifying how you want to open the file (read-only, write-only, create, etc.) (read the documentation)

d. The third argument is a list of bitwise OR of flags specifying the permissions to use when creating the file (see [documentation](#))

i. Since we don't expect you to know these, just use the following:

`S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH`

ii. This creates the file with read/write permissions for the owner of the file, read/write permissions for the group, read permissions for other users

10. Are we expected to be able to run user programs within our yash shell?

a. Yes, but it's no extra effort on your part. After all, it only involves running another program that exists on your computer (`gcc`) and then executing it like any other program (e.g. `"ls"`)

```
# gcc -o hello hello_world.c
# ./hello
hello world
```

11. Reasonable pipe test cases that involve redirection (these are just examples not actual test cases)

```
# Contents of in1.txt goes to out1.txt
# Contents of in2.txt goes to stdout
# The end of the pipe receives no input from the beginning
# of the pipe so the only input to the second cat is in2.txt
cat in1.txt > out1.txt | cat < in2.txt
```

```
# cat in1.txt > out1.txt | cat < in2.txt
in2.txt stuff
# cat out1.txt
in1.txt stuff
```

```
# contents of in1.txt goes through the pipe to
# the 2nd cat and gets output to stdout
cat in1.txt | cat
```

```
# cat in1.txt | cat
in1.txt stuff
```

12. We won't have any complex testing when it comes to the `fg`, `bg`, `jobs` commands.

Meaning, we will only test their basic functionality and will not combine those commands with redirections, pipes, etc.

- a. **Not allowed:** `fg &`
- b. **Not allowed:** `fg | echo`
- c. **Allowed:** `fg`
- d. **Allowed:** `bg`
- e. **Allowed:** `jobs`

13. What do I print out to stdout for `fg/bg`?

a. Example 1

```
# sleep 4
^Z# fg
sleep 4
```

- i. Note how the command was printed back out to stdout after executing the `fg` command (same as bash)

b. Example 2

```
# sleep 4
^Z# bg
[1]+ sleep 4 &
```

- i. Note how the command was printed back out to stdout in the jobs format with an `&` at the end after executing `bg` (same as bash)

14. What do I print out to stdout for a command that was run in the background using `&`?

Example:

```
# sleep 4 &
# jobs
[1]+  Running                  sleep 4 &
```

Note how nothing was printed back out to stdout after running the `sleep` command in the background (this is **different** from bash)

15. What should the output of the `jobs` command look like?

- a. For the most part, it should look the way bash does it, not zsh
- b. Major differences between our jobs output and Bash's jobs output. The intent (I think) of the differences was to make it easier to write your yash shell
- i. Our yash shell has a minus sign (-) next to **EVERY** job entry, whereas Bash only has a single minus sign next to the previous stopped command.
- ii. Our yash shell will put a plus sign (+) next to the most recent command **regardless of whether it was a stopped or backgrounded** command. In contrast, Bash only puts a plus/minus sign next to stopped commands
- c. Don't worry about spacing for the jobs command output
- d. Don't worry about printing the output of a command before/after the "Done" output (either of the two outputs below is fine)

```
# sleep 2 &
# ls
[1]+  Done                    sleep 2 &
i. Makefile yash yash.c
```

```
# sleep 2 &
# ls
Makefile yash yash.c
#
[1]+  Done                  sleep 2 &
```

ii.

e. Examples of correct yash input and output for the jobs command

i. Example 1 - basic jobs output

```
# sleep 5
^Z# sleep 10
^Z# sleep 15
^Z# sleep 20
^Z# jobs
[1]-  Stopped                  sleep 5
[2]-  Stopped                  sleep 10
[3]-  Stopped                  sleep 15
[4]+  Stopped                  sleep 20
```

1. Note where the ^Z (Ctrl-z) is being entered

ii. Example 2 - jobs output w/ background processes that haven't finished yet

```
# sleep 10
^Z# sleep 15
^Z# sleep 5 &
# sleep 20
^Z# jobs
[1]-  Stopped                  sleep 10
[2]-  Stopped                  sleep 15
[3]-  Running                  sleep 5 &
[4]+  Stopped                  sleep 20
```

1. Note that 'sleep 5 &' had **not** completed in the background yet, and then I ran 'jobs'

iii. Example 3 - jobs output w/ background processes that have finished

1. The input is the same as Example 2

2. This time, let's say 'sleep 5 &' **has completed** in the background and then you run 'jobs'

```

# sleep 10
^Z# sleep 15
^Z# sleep 5 &
# sleep 20
^Z# jobs
[3]-  Done                sleep 5 &
[1]-  Stopped             sleep 10
[2]-  Stopped             sleep 15
[4]+  Stopped             sleep 20
# jobs
[1]-  Stopped             sleep 10
[2]-  Stopped             sleep 15
[4]+  Stopped             sleep 20

```

- iv. Example 4 - jobs output numbering (follow bash not zsh)
 1. Note in the example above, after the sleep 5 command finished and you entered jobs the 2nd time, the job entry is gone from the table
 2. Note also that the job number goes $1 \rightarrow 2 \rightarrow 4$ instead of $1 \rightarrow 2 \rightarrow 3$
- v. Example 5 - jobs output numbering
 1. Let's say you then created and stopped a new job
 - o Input
sleep 25 → Ctrl-z
 - o Output


```

[1]-  Stopped    sleep 10
[2]-  Stopped    sleep 15
[4]-  Stopped    sleep 20
[5]+  Stopped    sleep 25
              
```
 2. Again notice that the number 3 was skipped
 - o This is because bash and your yash shell assign (1 + highest job number) to the new process.
 - o See Example 5 for an example showing this
- vi. Example 5 - jobs output numbering continuation of Example 4 and use of fg
 1. Let's say you're now continuing from Example 4 and run these commands
 2. Input
 - o fg → wait (if you have to) for job 5 to complete
 - i. Remember that fg always brings the most recent job (AKA the job at the bottom of your table) (AKA the job with the current highest job number) to the foreground
 - ii. Now that it is in the foreground, your stopped command can finish or be killed with Ctrl-c

3. Output

- [1]- Stopped sleep 10
- [2]- Stopped sleep 15
- [4]+ Stopped sleep 20

4. Input

- fg → wait (if you have to) for job 4 to complete

5. Output

- [1]- Stopped sleep 10
- [2]+ Stopped sleep 15

6. Input

- sleep 60 → Ctrl-z

7. Output

- [1]- Stopped sleep 10
- [2]- Stopped sleep 15
- [3]+ Stopped sleep 60

8. Notice that our job number is back at 3 again

- This is because bash and your yash shell assign (1 + highest job number) to the new process
- i.e., our job numbers are now (1, 2, 3) and **NOT** (1, 2, 6)

16. You need to handle multiple file redirections of **different types** per line. However, you don't need to handle multiple file redirections of the **same type** per line. Each end of the pipeline will only have at most one of each file redirection type. See examples below.

- Allowed:** `echo "hello world" > temp.txt`
- Allowed:** `cat < temp.txt`
- Allowed:** `cat < temp.txt > output.txt`
- Allowed:** `cat < temp.txt > output.txt 2> error.txt`
- Allowed:** `cat < in1.txt > out1.txt | cat < in2.txt > out2.txt`
- Not Allowed:** `cat temp.txt > output1.txt > output2.txt`
- Not Allowed:** `cat < output1.txt < output2.txt`

17. Do we need to parse for quotation marks? TLDR: no just parse for whitespace

- Input:** `echo "hello world"`
- Expected output:** `"hello world"`
 - This output is easier to implement because it doesn't worry about quotation marks. This output can be achieved by simply parsing the input command for whitespace and then sending each token to the echo command.
 - Token 0: `"hello`
 - Token 1: `world"`
- Bash Output:** `hello world`

- i. This is the output given by your bash shell and we don't expect you to do your parsing this way. Please output "hello world" with the quotation marks

18. Why did I get an error message output to stdout when I used error redirection?

- a. **Input:** `wc -l < fakefile.txt 2> error.txt`
- a. **Output:** `bash: fakefile.txt: No such file or directory`
 - i. You get an error message output to stdout and "error.txt" is empty because the error you're having is a bash error, not an error with the "wc" command. Bash is looking for the file "fakefile.txt" and since it can't find it, it notifies you through stdout. The stderr redirection only occurs within the child process that runs "wc." However, "wc" doesn't get a chance to run before bash realizes "fakefile.txt" doesn't exist.
- b. **Input:** `wc -l fakefile.txt 2> error.txt`
- c. **Output of error.txt:** `wc: fakefile.txt: No such file or directory`
 - i. This time, we send "fakefile.txt" as the direct input to the "wc" command instead of using input redirection (notice there is no "<" after the "wc -l" command). As a result, we will not get any output to stdout this time. Instead, the error message will be redirected to "error.txt" as expected.

19. Why do commands like "cd" and "history" not work in my yash shell? TL;DR: we don't expect you to implement "cd" or "history."

- a. The only commands you have to write/implement yourself are the `jobs`, `fg`, and `bg` commands. Every other command will be executed through existing programs on your computer (i.e. they come with your Linux operating system). For example, if you type `ls` on your yash shell or bash shell, what really happens is your shell checks your `$PATH` environment variable to see if it can find an existing file/executable called "ls". If your shell can find an "ls" executable, it then calls that program with the arguments provided to the shell. Note: in Linux, executables are denoted with no file ending.
- b. However, "cd" and "history" are programs built into the bash shell itself. Therefore, when you try to run those commands in your yash shell, it won't be able to find programs to execute those commands.
- c. To see which programs exist on your computer, run the "which" command.
 - i. `which ls`
 - 1. This outputs `/bin/ls`
 - 2. Pretty cool that "ls" is really just an executable that exists on your computer
 - ii. `which cd`
 - 1. This outputs nothing. Try `man builtin` in bash to see a list of builtin commands.