Shell part 2

CS 453: Operating systems

Overview

In this project, we will be extending the functionality of the version 1 of the shell from an earlier project. We will add the ability to run background jobs. Additionally, we will fix bugs from the first version of the shell. During your bug fixes be careful to not introduce any new bugs.

We will be adding a job descriptor data structure to keep track of jobs running in the background. Your shell will assign job numbers to commands started in the background and store them in the job list while they execute. We will also add a new built-in command **jobs**, that allows the user to query the status of background jobs.

We will be using a shared library that contains a linked-list implementation for tracking background jobs. The library will be provided to you via backpack in the folder named p0.

Setup

Shell project part 2 setup

- WARNING! If you made any commits on your **`shell_p1`** branch for your shell part1 project, then see your instructor to properly merge those changes to your master. If the only commit on your `shell_p1` branch was the grade file, then you can leave the branch alone. If you want to merge the grade file back to the master, again see your instructor.

- Next, make sure that you are on the **master** branch in your repo and there aren't any pending changes. If you have changes, then commit them and push them.

- Then do a **`git pull --rebase`** in your backpack directory to get the new project files. You will get a new **p3** folder and a **list** folder (more on this folder in the next subsection)

- Copy your p1 project into the the p3 project folder as follows:

```
cd p1
make clean
cd  ..
cp -r p1/* p3
git add p3/*
git commit -am “p3 setup complete”
git push origin master
```

List Library setup

We have provided a doubly linked list shared library and corresponding header files in the **list** directory in your backpack. You are required to link to this shared library (**`list/lib/libmylib.so`**) from the Makefile in your **p3** directory and point to the header files in the **list/include** directory. Take a look at the provided Makefile from the p1 project to see how to modify it to link to a library.

You will need to set your `LD_LIBRARY_PATH` environment variable so that your shell can find the list library when you run the mydash shell. This can be done as follows.

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:../list/lib
```

For grading, you may assume that we will have the `LD_LIBRARY_PATH` environment variable set as shown above.

You would modify your **Makefile** from the **mydash-src** folder in the **p3** project folder so that it links to the list library and the header files provided in the **list** folder. Do not copy any files from the **list** folder into the p3 directory.

Specification

Background jobs

Your shell can start a command in the background if an **&** is the last character on the line. For each background job that is started, it prints the job id (see section on Simple Job Control for job ids) and process id of the background process and the full command after starting the command in the background. After starting a background job, the mini-shell comes back with a prompt for the next command without waiting for the background command to finish. The user should **not** be required to separate the **&** from the command by a space. For example, the commands **date &** and **date&** are both valid. Additionally, blanks after the ampersand are valid as well.

In your backpack folder you will see a folder named **list** that has a fully implemented linked list library for you to used to store all your jobs. You are required to use this linked list in your project.

History Command

Add a new built in command named **history** to your shell to print out a history of command entered. You should leverage the **readline library** to accomplish this. Use BASH as a guide for behavior.

Simple Job Control

Use a job list to keep track of background jobs. For each job, we want to keep track of the original command, the process id of the background job. The simple job control feature consists of the following items listed below:

Job List

The mini shell should keep track of commands running in the background. Whenever the user starts a job in the background, it should assign and display the job number, the associated process id and the full command (including the ampersand).

    [n] process-id command

It should also display an informative message when a background job is done. That is, every time the user presses the ENTER key, the shell should report all the background jobs that have finished since the last time the user pressed the ENTER key. The message should be of the following form:

    [n] Done process-id command

Where n is the job number and command is the actual command that was typed. You should use the **WNOHANG** option with the **waitpid** system call to determine the status of background jobs.

Jobs command

Add a new built-in command called **jobs**, that prints out all the background commands that are running or are done but whose status has not yet been reported. Here is a sample output:

```
[1] 3451 Running sleep 100 &
[2] Done  3448 sleep 5 &
[3] 3452 Running gargantuan &
```

The first job should be assigned the job number 1. Each additional job should be assigned one higher number. *If lower numbered jobs finish, then we do not reuse them unless all jobs numbered higher than that number have also finished.*

Signal Handling

Set up signal handling in your shell. The shell should not terminate on **SIGINT** (the interrupt key, usually Ctrl-c) and should not stop on the signal **SIGTSTP** (the keyboard stop signal, usually Ctrl-z). A job running in the background should not be affected by these signals.

A foreground job should respond to these signals. In case of **SIGTSTP**, the shell should display a message stating what command got suspended. For example, on typing **Ctrl-z** while the sleep command is running below:

```
mydash> sleep 100
[1]  Stopped 4114 sleep 100
mydash>
```

If there is no current foreground job, then the shell should just display a new prompt and ignore any input on the current line. Extend your **jobs** command so that it also shows stopped jobs.

Extra Credit: Recursive Invocation

How does your shell behave if invoked recursively? By recursive, we mean running **mydash** from **mydash**. Which invocation catches the signal? How do you resolve this issue? Note that your implementation should not rely on checking if the command to run is named **mydash** but be more general. Stackoverflow is wrong on recursive invocation, read the man pages as they are correct and have sample code at the bottom of most topics :).

Extra Credit: Additional job control features

Warning: this is a very challenging feature

Add the built-in commands **fg** and **bg** that allow the last process stopped (with **SIGTSTP**) to be restarted using the **SIGCONT** signal (check the man page for **signal** in section 7 (man 7 signal). The **fg** and **bg** commands apply to the last stopped process. You will probably have to use the **setpgid** and **getpgrp** system calls. You can read more about **setpgid** and **getpgrp** here: http://man7.org/linux/man-pages/man2/setpgid.2.html.

With an optional argument the **fg** and **bg** commands can apply to any stopped job or job running in background. For example, **fg 2** brings the second job to the foreground (if there is a job number 2). Note that this implies that you can use the command **fg** to bring any running background job into the foreground. Read through the libc manual pages for a great explanation on foreground and background jobs. However **be careful with libc manual pages** as they describe implementing pipelines so just copying code (without understanding) will give you incorrect shell behavior.

Here are some example scenarios. Suppose you have just stopped a foreground process and then back-grounded it, the output of your built-in command **jobs** would look something like the following (before and after the **bg** command):

```
mydash> loop
<Ctrl>-z
[4] Stopped  4001 loop
mydash> jobs
[1] Running  3939 sleep 101 &
[2] Running  3940 sleep 103 &
[3] Running  3941 sleep 105 &
[4] Stopped  4001 loop
mydash> bg
[4] 4001 loop &
mydash> jobs
[1] Running  3939 sleep 101 &
[2] Running  3940 sleep 103 &
[3] Running  3941 sleep 105 &
[4] Running  4001 loop
mydash>
```

Note that the three sleep commands were already started in the background earlier by the user so they aren't affected by the Ctrl-z. The program **loop** is just the following infinitely looping/sleeping program:

```
main(){for(;;){sleep(100);}}
```

Valgrind

You must use valgrind and resolve memory errors and leaks as much as possible. Invoke valgrind as follows:

```
valgrind --leak-check=full ./mydash
```

Note that the readline library gives many valgrind errors that are not relevant. You can suppress these errors. Submission

Files committed to git (backpack)

Required files to submit through git (backpack), if you created more helper files that is fine.

1. Makefile
2. mydash.c
3. mydash.h
4. README.md
5. and other files

Push your code to a branch for grading

Run the following commands

1. make clean
2. git add <file ...> (on each file!)
3. git commit -am “Finished project”
4. git branch shell_p2
5. git checkout shell_p2
6. git push origin shell_p2
7. git checkout master

Check to make sure you have pushed correctly

- Use the command **git branch -r** and you should see your branch listed (see the example below)

```
$ git branch -r
  origin/HEAD -> origin/master
  origin/master
  origin/shell_p2
```

Submit to Blackboard

You must submit the sha1 hash of your final commit on the correct branch. Your instructor needs this in order to troubleshoot any problems with submission that you may have.

- git rev-parse HEAD

Grading Rubric

Provided via backpack.