

Due Date: See website for due date (Late days may be used.)

This project must be done in groups of 2 students. Self-selected groups must have registered using the grouper app (URL). Otherwise, a partner will be assigned to you.

1 Introduction

This assignment introduces you to the principles of process management and job control in a Unix-like operating system. In this project, you will develop a simple job control shell.

This is an open-ended assignment. In addition to implementing the required functionality, we encourage you to define the scope of this project yourself.

2 Base Functionality

A shell receives line-by-line input from a terminal that represents user commands. Some user commands are builtins, which are implemented by the shell itself. If the user inputs the name of such a built-in command, the shell will execute this command. Otherwise, the shell will interpret the input as containing the name of an external program to be executed, along with arguments that should be passed to it. In this case, the shell will fork a new child process and execute the program in the context of the child. Normally, the shell will wait for a command to complete before reading the next command from the user. However, if the user appends an ampersand ‘&’ to a command, the command is started and the shell will return to the prompt immediately. In this case, we refer to the running command as a “background job,” whereas commands the shell waits for before processing new input are called “foreground jobs.”

The shell provides *job control*. A user may interrupt foreground jobs, send foreground jobs into the background, and vice versa. Thus at a given point in time, a shell may run zero or more background jobs and zero or one foreground jobs. If there is a foreground job, the shell waits for it to complete before printing another prompt and reading the next command. In addition, the shell informs the user about status changes of the jobs it manages. For instance, jobs may exit, or terminate due to a signal, or be stopped for several reasons.

At a minimum, we expect that your shell has the ability to start foreground and background jobs and implements the built-in commands ‘jobs,’ ‘fg,’ ‘bg,’ ‘kill,’ ‘exit,’ and ‘stop.’ The semantics of these commands should match the semantics of the same-named commands in bash. The ability to correctly respond to ^C (SIGINT) and ^Z (SIGTSTP) is expected, as are informative messages about the status of the children managed. Like bash, you should use consecutively numbered small integers to enumerate your jobs.

For the minimum functionality, the shell need not support pipes ($|$), I/O redirection ($< > >>$), nor the ability to run programs that require exclusive access to the terminal (e.g., vim).

We expect most students to implement pipes, I/O redirection, and managing the controlling terminal to ensure that jobs that require exclusive access to the terminal obtain such access (see Section 3.3). Beyond that, *cush*’s customizability, described in Section 5, should allow for plenty of creative freedom.

3 Strategy

3.1 Handling SIGCHLD To Process Status Changes

At a given point in time, a user may have multiple jobs running, each executing arbitrary programs chosen by the user. Because the shell cannot and does not know what these programs do, it has to rely on a notification facility from the OS to be informed when these jobs encounter events the shell needs to know about. We refer to such events as “changing status,” where “status” means whether the job is running¹, has been stopped, has exited, or has been terminated with a signal (for instance, crashed).

This notification facility involves a protocol in which the OS kernel sends an asynchronous signal (SIGCHLD) to the shell, and in which the shell then follows up by executing a system call (a variant of `wait()`, specifically `waitpid()`, as shown in the provided starter code).^{2 3}

Thus, you will need to catch the SIGCHLD signal to learn about when the shell’s child processes change status. Since child processes execute concurrently with respect to the parent shell, and since the shell has no knowledge of what these processes are doing, it is impossible to predict when a child will exit (or terminate with a signal), and thus it is impossible to predict when this signal will arrive. In the worst case, a child may have already terminated by the time the parent returns from `fork()`! You also should not make any assumptions about how a child process might change state: for instance, even if the user issues a `kill` built-in command to terminate a process, the processes might not immediately terminate (or may not terminate at all), so the shell should not assume that a status change occurred *unless and until* it has first-hand information from the OS that it did.

Because of the asynchronous nature of signal delivery, you will need to block handling of the signal in those sections of your code where you access data structures that are also needed by the handler that is executed when this signal arrives. For example, consider the data structure used to maintain the current set of jobs. A new job is added after a child process has been forked; a job may need to be removed when SIGCHLD is received. To avoid a situation where the job has not yet been added when SIGCHLD arrives, or -

¹We use the word “running” here not in the sense of the simplified process state diagram, but rather in the informal sense of having been started, but not having finished, and also not currently suspended (stopped) by the user or system.

²Such protocols are widely used in systems programming - for instance, an operating system kernel interacts with devices in a very similar way through interrupts.

³So far, we have equated jobs and child processes in our discussion. Jobs that include multiple child processes will be discussed in Section 3.2.

worse - a situation in which SIGCHLD arrives *while* the shell is adding the job, the parent should block SIGCHLD until after it completed adding the job to the list. If the SIGCHLD signal is delivered to the shell while the shell blocks this signal, it is marked pending and will be received as soon as the shell unblocks this signal.

Use the provided helper functions in `signal_support.c` to block and unblock signals, which in turn rely on `sigprocmask(2)`. To set up signal handlers, they use the `sigaction(2)` system call with `sa_flags` set to `SA_RESTART`. The mask of blocked signals is inherited when `fork()` is called. Consequently, the child will need to unblock any signals the parent had blocked before calling `exec()`.

3.2 Process Groups

User jobs may involve multiple processes. For instance, the command line input `ls | grep filename` requires that the shell start two processes, one to execute the `ls` and the other to execute the `grep` command. Aside from this example, child processes that a user program may start⁴ should usually be part of the same job so that the user can manage them as one unit. To help manage these scenarios, Unix introduced a way to group processes that makes it simpler for the shell and for the user to address them as one unit.

Each process in Unix is part of a group. Process groups are treated as an ensemble for the purpose of signal delivery and when waiting for processes. Specifically, the `kill(2)`, `killpg(2)`, and `waitpid(2)` system calls support the naming of process groups as possible targets⁵. In this way, if a user wants to terminate or stop a job, it is possible for the shell to send a termination or stop signal to a process group that contains all processes that are part of this job. To facilitate this mechanism the shell must arrange for process groups to be created and for processes to be assigned to these groups.

Each process group has a designated leader, which is one of the processes in the group. To create a new group with itself as the leader, a process simply calls `setpgid(0, 0)`. The process group id of a process group is equal to the process id of the leader. Child processes inherit the process group of their parent process initially. They can then form their own group if desired, or their parent process can place them into a different process group via `setpgid()`. The shell must create a new process group for each job and make sure that all processes that will be created for this job become members of this group. Note that while the process group management facilities are available to all user programs, only shell programs will typically make use of them – for most other programs, the default behavior of inheriting the parent’s process group is a desirable default.

In addition to signals and `waitpid`, process groups are used to manage access to the terminal, as described next.

⁴For instance, the ‘make’ utility program starts many other processes such as compilers and linkers.

⁵Note the idiosyncracies of the API: `kill(-pid, sig)` does the same as `killpg(pid, sig)`. You can use either, but make sure to use the correct sign corresponding to the call you use.

3.3 Managing Access To The Terminal

Running multiple processes on the same terminal creates a sharing issue: if multiple processes attempt to read from the terminal, which process should receive the input? Similarly, some programs - such as `vi` - output to the terminal in a way that does not allow them to share the terminal with others.⁶

To solve this problem, Unix introduced the concept of a foreground process group. The kernel maintains such a group for each terminal. If a process in a process group that is not the foreground process group attempts to perform an operation that would require exclusive access to a terminal, it is sent a signal: `SIGTTOU` or `SIGTTIN`, depending on whether the use was for output or input. The default action taken in response to these signals is to suspend the processes in that group. If that happens, the processes’ parent (i.e., your shell) can learn about this status change by calling `waitpid(). WIFSTOPPED(status)` will be true in this case. To allow these processes to continue, their process group must be made the foreground process group of the controlling terminal via a call to `tcsetpgrp()`, and then the process group must be sent a `SIGCONT` signal. The shell will typically take this action in response to a `'fg'` command issued by the user.

Signals that are sent as a result of user input, such as `SIGINT` or `SIGTSTP`, are also sent to a terminal’s foreground process group. Note that this sending of signals occurs automatically by the operating system, it is *not an action the shell takes*. Delivering this signal to an entire process group makes it so that when a user hits Ctrl-c to terminate a job such as `ls | grep filename` both the process running `ls` and the process running `grep` will receive the `SIGINT` signal, informing them of the user’s desire to terminate them. To ensure that such signals are delivered to the correct process group, the shell must arrange for these process groups to exist and be populated with the correct processes, and it must inform the OS kernel which process group the user intends to run in the foreground at a given point in time.

3.4 Managing The Terminal’s State

Many years ago, most Unix terminals were actual devices that had a console and a keyboard and that were connected to the main computer with some kind of serial interface such as RS-232. To control those devices, the OS device drivers would need to control a set of input and output flags collectively known as the terminal state. In modern systems, the most commonly used terminal type is a pseudo-terminal (pty) connected to an ssh network connection, yet this model still exists. You can type `stty -a` to see what those flags are, though you probably won’t care about their details.

Some processes change the state of the terminal in a certain way. For instance, `vim` puts the terminal in so-called “raw” mode where it receives keystrokes as they are typed (as opposed to “cooked” which requires the user to end a line with the enter key before it is received by a program). So does `bash` and in fact, your shell, which uses the `readline`

⁶Note that regular output via `write(2)` does not require exclusive access, unless the terminal’s `'tostop'` flag is set. The terminal will simply interleave such output.

library, does this, too, while reading user input.

This raises a management issue when the user switches between the shell’s command line and foreground process jobs. For instance, a user may start vim, then use Ctrl-z to stop it, run some other job in the foreground, then stop it, resume vim, exit vim, and resume the second job.

In this case, it is necessary to restore the terminal state whenever the vim process is resumed to what it was before vim was stopped. Interestingly, it is possible for a process to perform such restoration itself (in fact, vim does this by handling the `SIGCONT` signal).

However, if the shell performed such saving and restoration transparently, then any program that manipulates its terminal state could be run under a job control regime. Specifically, your shell should save the state of the terminal when a job process is suspended and restore it when the job is continued in the foreground by the user.⁷

When the shell returns to the prompt, it must make itself the foreground process group of the terminal. In this case, it should also restore a known good terminal state. Your shell should sample this known good terminal state when it starts. You may find the functions provided in `termstate.management.c` useful, which already handle most of the logic.

This known good state is also the state that the terminal will be in if a new job is started by the user. Therefore, programs that are agnostic with respect to the state of the terminal will continue to work. However, there has to be a way for the user to change the default terminal settings programs encounter when they are run (as well as the terminal settings that are in effect while the shell is being used by a user). The `stty` command exists for this purpose. When run, it will display and/or change existing settings to suit a user’s preferences.

The shell must respect changes made by `stty` and replace its known good terminal state with the state the terminal was put in by the `stty` command. To that end, the following convention is used: if any foreground job exits with a success (zero) exit status, the current terminal state will be sampled by the shell and becomes the new known good state (as per the user’s intent.) Your shell should do this sampling. Make sure to *not* sample the terminal state in these cases:

- A job exits that was not started as a foreground job.
- A job exits that is not a foreground job at the time of its exiting.
- A job terminates with a signal.
- A job exits but the exit status code is non-zero.

For jobs that consists of multiple processes, consider the last process in the pipeline. You will note that this heuristics is not perfect – it will in fact sample any successfully exiting

⁷This is a recommendation (not a requirement though) spelled out in the POSIX standard. Unfortunately, only the Korn shell (ksh) actually does that in practice, other widely used shells (bash, zsh, dash) do not. Under those shells, job-control unaware programs would fail.

job’s state rather than just where the user intends it – but this doesn’t pose a big problem in practice since the majority of programs doesn’t reprogram the terminal.

3.5 Pipes and I/O Redirection

A pipeline of commands is considered one job. All processes that form part of a pipeline must thus be part of the same process group, as already discussed in Section 3.3. Note that all processes that are part of a pipeline are children of the shell, e.g., if a user runs `a | b` then the process executing `b` is *not* a child process of the process executing the program `a`.

To implement the pipes itself, use the `pipe(2)` system call, or alternatively the `pipe2(2)` GNU extension. The latter allows you to set flags on the returned file descriptors such as `O_CLOEXEC`. A pipe must be set up by the parent shell process before a child is forked. Forking a child will inherit the file descriptors that are part of the pipe. The child must then redirect its standard file descriptors to the pipe’s input or output end as needed using the `dup2(2)` system call. If the user used the `&` instead of the `|` symbol, both standard output and standard error should be redirected to the pipe.

Although the parent shell process creates pipes for each pair of communicating children before they are forked, it will not itself write to the pipes or read from the pipes it creates. Therefore, you must make sure that the parent shell process closes the file descriptors referring to the pipe’s ends *after* each child was forked. This is necessary for two reasons: first, in order to avoid leaking file descriptors. Second, to ensure the proper behavior of programs such as `/bin/cat` if the user asks the shell to execute them. To see why, we must first discuss what happens to file descriptors on `fork()`, `close()`, and `exit()`.

Each file descriptor represents a reference to an underlying kernel object. `fork()` makes a shallow copy of these descriptors. After `fork()`, both the child and the parent process have access to any object the parent process may have created (i.e., open files or other kernel objects). Closing a file descriptor in the (parent) shell process affects only the current process’s access to the underlying object. Hence when the parent shell closes the file descriptor referring to the pipe it created, the child processes will still be able to access the pipe’s ends, allowing it to communicate with the other commands in the pipeline.

The actual object (such as a pipe or file) is destroyed only when the last process that has at least one open file descriptor referring to the object closes the last file descriptor referring to it. If you failed to close the pipe’s file descriptors in the parent process (your shell), you compromise the correct functioning of programs that rely on taking action when their standard input stream signals the end of file condition. For instance, the `/bin/cat` program will exit if its standard input stream reaches EOF, which in the case of a pipe happens if and only if all descriptors pointing to the pipe’s output end are closed. So if `cat`’s standard input stream is connected to a pipe for which the shell still has an open file descriptor, `cat` will never “see” EOF for its standard input stream and appear stuck.

Lastly, note that when a process terminates for whatever reason, via `exit()` or via a signal, all file descriptors it had open are closed by the kernel as if the process had called

`close()` before terminating. This means that you do not need to worry about making sure that file descriptors you open for the shell's child processes are closed after these child processes exit. However, since the shell is a long running program that does not exit between user commands, the shell must close *its own* copies of these file descriptors to avoid above-mentioned leakage. If it did not, it would eventually run out of file descriptors because the OS imposes a per-process limit on their number.

Although the processes that are part of pipeline typically interact with each other through the pipe that connects their standard streams, they are still independent processes. This means they can exit, or terminate abnormally, independently and separately. When your shell calls `waitpid()` to learn about these processes' status changes, it will learn about each one *separately*. You will need to map the information you learn about one process to the job to which it belongs, using a suitable data structure you define in your shell implementation.

Here is a brief table summarizing facts about the status changes and the corresponding macros you can apply to the `status (out)` parameter⁸ returned by `waitpid`:

Event	How to check for it	Additional info	Process stopped?	Process dead?
User stops fg process with Ctrl-Z	WIFSTOPPED	WSTOPSIG equals SIGTSTP	yes	no
User stops process with <code>stop</code> (cush) or <code>kill -STOP</code> (bash)	WIFSTOPPED	WSTOPSIG equals SIGSTOP	yes	no
non-foreground process wants terminal access	WIFSTOPPED	WSTOPSIG equals SIGTTOU or SIGTTIN	yes	no
process exits via <code>exit()</code>	WIFEXITED	WEXITSTATUS has return code	no	yes
user terminates process with Ctrl-C	WIFSIGNALED	WTERMSIG equals SIGINT	no	yes
user terminates process with <code>kill</code>	WIFSIGNALED	WTERMSIG equals SIGTERM	no	yes
user terminates process with <code>kill -9</code>	WIFSIGNALED	WTERMSIG equals SIGKILL	no	yes
process has been terminated (general case)	WIFSIGNALED	WTERMSIG equals signal number	no	yes

Additional information can be found in the GNU C library manual, available at <http://>

⁸A common mistake some students make is to confuse the exit status and the job status. The exit status is a single integer value that a child process can pass to the `exit(2)` system call and which the parent can retrieve via `waitpid()`, whereas the job status is an internal shell variable/struct field that records the shell's knowledge about the job control status of a job, e.g., whether it's running or stopped. `waitpid` will also use status to report when processes were stopped (or terminated) by a signal, so your shell must use the process status information obtained via `waitpid` to update the job's job control status as necessary.

www.gnu.org/s/libc/manual/html_node/index.html. Read, in particular, the sections on Signal Handling and Job Control.

3.6 Use of `posix_spawn`

In a 2019 paper published at the HotOS workshop, Baumann et al [1] criticized the use and teaching of the Unix style of creating a new process by first creating a clone via `fork()`, then customizing the new process’s environment through actions the clone performs on itself before executing a new program. A key weakness of this approach is that it is incompatible with multithreaded programs. They propose the use of an existing alternative API instead, i.e., `posix_spawn(3)`. This call combines `fork()` and `exec()` into one, and it also can be customized so that the child process will perform the necessary operations to set up or join a process group and to redirect inherited file descriptors as desired.

However, `posix_spawn` as defined by POSIX lacks one important feature, which is to provide the child process with ownership of its terminal. This action cannot be performed in the parent since doing so would create a race condition: the child may reach a point where it assumes it had terminal ownership before the parent assigns ownership to it. For this project, you have access to a version of `posix_spawn` that includes a non-portable extension `posix_spawnattr_t` `tcsetpgrp_np(posix_spawnattr_t *attr, int fd)` that allows you to provide a file descriptor referring to the terminal for which the child process should acquire ownership.

For your implementation, you are encouraged to use `posix_spawn` in lieu of `fork + exec`. If you choose to do so, your implementation will avoid the potential sources of bugs that the use of `fork()` introduces, such as inadvertently attempting to update parent data structures in the child process, and in general will exhibit to easier-to-understand control flow and memory access semantics. Control flow will be traditional and linear: `posix_spawn` will be called once, and return once, like any ordinary function. It will spawn a new program in a new process as a side effect. This child process will never directly access data structures inherited from the parent, though it relies on inheriting open file descriptors like in the `fork` case. `posix_spawn` also does not change the fact that the created process will immediately run concurrently with the parent process when it returns. In other words, you may think of it as a combination of `fork` and `exec`, not of `fork`, `exec`, and `wait`.

However, it is difficult to use `posix_spawn` successfully if you do not understand how `fork` and `exec` interact with file descriptors and process groups, so the explanation in the preceding sections still applies and must be thoroughly understood. Everything related to job management applies equally as it is independent of the method used to start the child processes.

When using `posix_spawn`, you must observe all of the following hints

- Use the `posix_spawnnp` variant to be able to find programs in the user’s path.
- Use `posix_spawn_file_actions_adddup2` to wire up pipe file descriptors and handle the redirection of standard error.

- Use `posix_spawn_file_actions_addopen` to wire up I/O redirection from/to files.
- Use `posix_spawnattr_setpgroup` along with the `POSIX_SPAWN_SETPGROUP` flag to establish or join a new process group.
- Use `posix_spawnattr_tcssetpgrp_np` along with the `POSIX_SPAWN_TCSETPGROUP` flag to give the child’s process group terminal ownership.
- Use `posix_spawnattr_setflags` to set the desired flags. You may include `POSIX_SPAWN_USEVFORK` to make use of the specialized (and slightly faster) `vfork()` system call. Note that you may call this function *only once* since later calls will replace the flags set in earlier ones. Thus, you need to bitwise combine all necessary flags into one value before calling it with this value.
- You do not need to perform a `setpgid()` call in the parent since the race condition necessitating this call no longer exists: the call to spawn won’t return until after the child has been placed into its process group.
- You will need to pass the current environment as the last argument. Add an external declaration like so `extern char **environ;`.
- Lastly, note that the resulting code won’t necessarily be shorter (my version is 94 vs. 67 lines for the fork/exec variant), but very likely less confusing.
- The Makefile will set the correct include path and library flags to link with the required version of `posix_spawn`, which overrides the version in the installed GNU C library. You will need to build the library first, use

```
(cd posix_spawn; make)
```

to that end.

4 Use of Git

You will use **Git** for managing your source code. Git is a distributed version control system in which every working directory contains a full repository, and thus the system can be used independently of a (centralized) repository server. Developers can commit changes to their local repository. However, in order to share their code with others, they must then push those commits to a remote repository. Your remote repository will be hosted on `git.cs.vt.edu`, which provides a facility to share this repository among group members. For further information on git in general you may browse the official Git documentation: <http://git-scm.com/documentation>, but feel free to ask questions on the forum as well! The use of git (or any distributed source code control system) may be new to some students, but it is a prerequisite skill for most programming related internships or jobs.

You will use a departmental instance of Gitlab for this class. You can access the instance with your SLO credentials at <https://git.cs.vt.edu/>.

The provided base code for the project is available on Gitlab at <https://git.cs.vt.edu/cs3214-staff/cs3214-cush>,

One team member must fork this repository by viewing this page and clicking the fork link. This will create a new repository for you with a copy of the contents. From there you must view your repository settings, and *set the visibility level to private*. On the settings page you may also invite your other team member to the project so that they can view and contribute.

Group members may then make a local copy of the repository by issuing a `git clone <repository>` command. The repository reference can be found on the project page such as `git@git.cs.vt.edu:teammemberwhoclonesit/cs3214-cush.git`. To clone over SSH (which you may need to do on rlogin), you will have to add an SSH public key to your profile by visiting <https://git.cs.vt.edu/profile/keys>. This key is separate from the key you added to your `/.ssh/authorized_keys` file. Although you could use the same key pair you use to log into rlogin, we recommend using a separate key pair. This way you can avoid storing the private key you use to access rlogin on rlogin itself.

If updates or bug fixes to this code are required, they will be announced on the forum. You will be required to use version control for this project. When working in a team, both team member should have a roughly equal number of committed lines of code to show their respective contributions.

Please note. To facilitate the automated grading of your git usage, please follow the following rules:

- Do not rename the repo when you fork it.
- Do not create a git group; fork the repo under the namespace of one of the two group members.
- Make sure that, once you have finished, your final product will be on the master branch.
- Make sure that the git commit log on this branch shows the contributions of both team partners under their CS pid.
- You may use branches during development, but if you do, make sure to merge those branches. Don't squash your commits when you do so.
- You must use `git.cs.vt.edu` and not any external git server.

4.1 Code Base

To build the provided code, run `make` in the `src` directory. (Don’t forget to build the `posix_spawn` library first.)

The code contains a command line parser that implements the following grammar:

```
cmd_line : cmd_list

cmd_list :
    | pipeline
    | cmd_list ';'
    | cmd_list '&'
    | cmd_list ';' pipeline
    | cmd_list '&' pipeline

pipeline : command
    | pipeline '|' command
    | pipeline '|'&' command

command : WORD
    | input
    | output
    | command WORD
    | command input
    | command output

input : '<' WORD

output : '>' WORD
    | '>>' WORD
    | '>&' WORD
```

Look at the provided `cush.c` main function to see how to invoke the parser. If a command line is semantically correct, the parser code will create a `ast_command_line` data structure, which refers to a list of `ast_pipeline` structures. Each `ast_pipeline` is used to create a job. It may consist of one or more individual commands that form a pipeline. Each command is represented as a `ast_command` structure. Study the definitions of these structures.

By default, the provided code will read a line, parse it, and dump the parsed command line to `stdout`.

The files `signal_support.c` and `termstate_management.c` contain a number of utility functions for dealing with signals and managing the terminal state, which do most of the heavy lifting for you. We *strongly* recommend you use these functions rather than directly calling the functions described in the textbook.

5 Builtins

The basic builtins our tests expect include `kill`, `fg`, `bg`, `jobs`, `stop`, `exit`.

In addition, you should implement at least 2 builtin commands or a functionality extension, a simple one and a more complex one. Ideas for simple builtins include:

- A custom prompt (e.g. outputting hostname and current directory)
- Setting and unsetting environment variables
- `cd` to change the current directory (should support changing to the home directory when invoked with just `cd`)
- Other simple commands

Ideas for “more complex” builtins include

- A user-customizable prompt (e.g. like `bash`’s `PS1`) that provides a means for the user to set the prompt. Implement a substantial subset of ‘`PS1`’s prompt escape sequences, see [here](#).
- Command-line history (perhaps using’s GNU History library) (this should include the features commonly provided by GNU history, such as event designators. If GNU history is properly integrated, these will come for free.
- Glob expansion (e.g., `*.c`). You may use GNU’s `glob` library, see `glob(3)`.
- Support for aliases (definition and expansion)
- Shell variables
- Timing commands: “time” or a builtin version time-outs.
- A directory stack maintained via `pushd`, `popd`, etc.
- Backquote substitution
- Smart command-line completion, i.e., help with mistyped commands
- Embedding applications: scripting languages, web servers, etc.

Generally, we expect for more complex builtins to add significant value for the user.

A side-note on Unix philosophy - in general, Unix implements functionality using many small programs and utilities. As such, built-in commands are often only those that must be implemented within the shell, such as `cd`. In addition, essential commands such as ‘kill’ are often built-in to make sure an operator can execute those commands even if no new processes can be forked. Your builtins should generally stay with this philosophy and implement only functionality that is not already available using Unix commands or that would be better implemented using separate programs. If in doubt, ask.

6 Testing

We will provide a test driver to test your project, and tests for the basic and advanced functionality. The tests are part of the repository, which may be updated once before the deadline.

The basic and advanced tests are also in the Gitlab repository that you forked to start the project. If updates to the tests come out you will have to pull from the remote repository to update your local copy.

Note: you are required to add tests for the builtin commands you add, using the example.

7 Grading

Rubrics. This project will account for 140 points. 50 points will be assigned for passing the base tests. 50 points for advanced tests, and up to 20 additional points can be earned through builtins. Builtins requires tests to be considered for credit.

10 points are awarded for correct use of version control, and 10 points for documentation. In addition, deductions may be taken for deficiencies in coding style and lack of robustness.

Coding Style. Your coding style should match the style of the provided code. You should follow proper coding conventions with respect to documentation, naming, and scoping.

You must check the return values of all system calls and library functions, with the sole exception of `malloc(3)` or `calloc(3)`. (Production code would need to check for those as well; this is a simplification for this project.) This requirement includes calls such as `kill(2)` and `close(2)`.

You may not use unsafe string functions such as `strcpy()` or `strcat()`, see the website for a complete list.

Submission. You must submit a design document, `README.txt`, as an UTF-8 encoded Unicode document using the following format to describe your implementation:

```
Student Information
```

```
-----
```

```
<Student 1 Information>
```

```
<Student 2 Information>
```

```
How to execute the shell
```

```
-----
```

```
<describe how to execute from the command line>
```

Important Notes

<Any important notes about your system>

Description of Base Functionality

<describe your IMPLEMENTATION of the following commands:
jobs, fg, bg, kill, stop, ^C, ^Z >

Description of Extended Functionality

<describe your IMPLEMENTATION of the following functionality:
I/O, Pipes, Exclusive Access >

List of Additional Builtins Implemented

(Written by Your Team)
 <builtin name>
 <description>

The TA will assign credit only for the functionality for which test cases and documentation exist.

You must submit a .tar.gz file of your 'src' directory, which contains a Makefile. 'src' directory must appear as a subdirectory in your tar file. You need to run 'make clean' on your directory before you create your tarball. Make sure to also delete all temporary folders and files (i.e. clean your submission to pertinent files).

Please use the submit.py script or web page and submit your tar file under 'p1'. Only one group member needs to submit. See the website for further submission instructions.

Good Luck!

References

- [1] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. A fork() in the road. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 14–22, 2019.