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 General Functionality

A shell receives line-by-line input from a terminal that represents user commands. Some user commands (such as `cd`) 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 spawns a new child process that will execute the program. An example would be if the user typed `ls`.

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 suspend (stop) foreground jobs, send foreground jobs into the background, and vice versa. Thus at a given point in time, a shell may be running 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 builtin commands listed in Section 6. The semantics of these commands should roughly match the semantics of the similarly named commands in bash. The ability to correctly set up the terminal to allow the user the use of `^C` (SIGINT) and `^Z` (SIGTSTP) is expected, as are informative messages about any resulting changes in the status of the children managed. Like bash, you should use consecutively numbered small integers to identify your jobs.

However, we expect most students to also implement pipes (`|`), I/O redirection (`<, >, >>`), and the ability to run programs that require exclusive access to the terminal (e.g. vim) by managing access to the controlling terminal.

3 Details

3.1 The Shell

A shell receives line-by-line input from a terminal that represents user commands. A given command line has the following structure:

- A command line contains multiple *pipelines* separated by semicolons (or ampersands).
- Each pipeline contains multiple *commands* separated by the pipe symbol `|`.
- Each command is a sequence of strings that consists of:
  - A program (which may be builtin or external).
  - The arguments to the program.
  - Any input or output redirections.

Here is one such line of input with the corresponding output:

```
$ echo "hello world!" | rev; echo "content" > file.txt; < file.txt cat
!dlrow olleh
content
```

This command line contains 3 pipelines. The first pipeline contains two commands, the second pipeline has one command with an output redirect, and the third pipeline has one command with an input redirect.

Each pipeline shown corresponds to a *job* managed by your shell. Each command in a the pipeline that is not a built-in corresponds to a *process*, and all processes in a job are run in parallel. A *job* completes when all processes managed by the job have terminated (either exited normally or through termination by a signal).

After receiving a command line, your shell will create a job for each pipeline. When creating a job, your shell will spawn a process (Section 3.2) for each command in the pipeline. These processes will be set up to:

- create or join the job’s process group (Section 3.4)
- if started as a foreground job, acquire ownership of the controlling terminal (Section 3.5)
- retrieve input from/direct output to a pipe or a file (Section 3.7)

After a job is created, your shell will wait on the job if it is a foreground job, or continue execution as normal if it is a background job. Once the job has been created and its processes started, they will run and subsequently may terminate (normally, or crash with a signal) or be stopped. We refer to such events as “state changes” which your shell needs to handle by taking appropriate actions such as informing the user, updating its job list, etc. The shell must perform this bookkeeping after receiving user input and while waiting for a foreground job (Section 3.3).

To provide each job running under your shell’s control with the illusion that it has a dedicated virtual terminal available while it is in the foreground, your shell should also save and restore the terminal state accordingly as processes stop, continue, or exit (Section 3.6).

3.2 Spawning Processes

In Unix, processes are spawned by first creating a clone of the current process via `fork()`. The clone performs any necessary setup for the new process (such as rewiring the standard I/O file descriptors to handle pipelines or redirects), then executes the new program with `exec()`.

Implementing a shell by directly using `fork()` and `exec()`, along with the multitude of APIs necessary to set up the child process correctly, is possible but difficult and prone to errors. Moreover, for a shell, the full generality of the `fork()` system call is not needed. Therefore, we will be using the `posix_spawn(3)` API, which provides an abstraction for starting a new processes with a new program. Before invoking it, you must prepare the call by setting a list of attributes and flags so that the child process will perform the necessary operations to set up or join a process group, possibly take terminal ownership, and/or to redirect inherited file descriptors as desired by the user.

Using `posix_spawn` in lieu of `fork + exec` means the control flow of your program will seem traditional and linear: `posix_spawn` will be called once, and return once, like any ordinary function (but unlike `fork()`.) 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 as if `fork` had been used directly. `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`.

3.3 Processing State 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 the operating system to inform the shell of events the shell needs to know about. We refer to such events as “changing status,” where “status” means whether the job is running[1], has been stopped, has exited, or has been terminated with a signal (for instance, crashed).

[1]We 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. This usage reflects the perspective of the shell’s user.

This also means your shell cannot make any assumptions about how or when a child process might change state; for instance, even if the user issues a `kill` built-in command to terminate a process, the process 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.

The shell can wait on status changes for processes by executing a system call (a variant of `wait()`, specifically `waitpid()`). This call can block if there were no changes in the state of child processes since the last call, or it can be configured to return immediately otherwise. The shell should then use this information to manage the status of the job the process is associated with. For example, the number of alive processes in a job decreases when the shell obtains knowledge that a process related to the job has terminated. Or, if the shell obtains knowledge that a process has been stopped (for example, due to `^Z` (SIGTSTP) sent by the user) for a job that is currently running in the foreground, the shell should return to the prompt to await further user input.

However, using `waitpid()` alone is not sufficient for a shell to properly manage the changing statuses of its spawned processes. Consider what happens when the shell after displaying the prompt waits for user input; it is likely blocking on a `read()` call to read user input, and will not return until user input is received. If processes running in the background exit during this time, the shell will be unable to react to such changes until the user completed their input. This, in turn, would lead to zombie processes that have exited but for which the shell hasn’t called `wait`.

Traditional shells use the OS’s signal delivery mechanism to be informed of child status changes. However, signal delivery, because of its asynchronous nature, introduces a number of challenges that must be overcome in a correct implementation, which is difficult to implement correctly in both C and Rust. For this reason, for this project, we provide a channel abstraction that relies on multithreading (which we’ll cover later in this class) and that takes care of calling `waitpid()` for you. See Appendix A.1 for details. However, your shell will still need to process the information obtained via `waitpid`, which is sent through the channel. This information consists of (a) the pid of the process in question, and (b) details about the status change (exit, termination via signal, or having been stopped). See Appendix A.4 for details.

3.4 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 start[2] 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

[2]For instance, the ‘make‘ utility program starts many other processes such as compilers and linkers.

the purpose of signal delivery and when waiting for processes. Specifically, the `kill(2)`, `killpg(3)`, and `waitpid(2)` calls support the naming of process groups as possible targets. In this way, if a user wants to terminate a job, it is possible for the shell to send a termination 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. The shell does this as part of the preparation it takes in setting up attributes for the `posix_spawn` call that starts a process.

First, the shell must include the `POSIX_SPAWN_SETPGROUP` in the set of flags it sets. Second, it must - via `PosixSpawnAttr::set_pgroup` specify which process group the new process should create or join. The first command in a pipeline will create a new process group, the other commands in the pipeline will join this group. This is accomplished by passing 0 as the process group to be joined for the first command, and the pid of the first process for the others. The OS assigns process group ids based on the process id of the process that started the group, which is also referred to as its leader.

Child processes inherit the process group of their parent process initially. 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 because they are then automatically included when signals are sent to their parent's group.

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

3.5 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. [3]

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()`. To allow these processes to continue, their process group must be made the foreground process group of the controlling terminal and then the process group must be sent the `SIGCONT` signal. The shell will typically take this action in response to a 'fg' command issued by the user.

Ownership of the controlling terminal can be given to a process group using the `Terminal::give_terminal_to` abstraction provided to you, which will be described in Section 4.1.

[3] Note 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.

However, `Terminal::give_terminal_to` should only be used to give terminal ownership to previously stopped processes when the user places them back into the foreground. To make sure that a foreground job has terminal ownership when it is first started, instead use the `PosixSpawnFileActions::add_tcsetpgrp_np` file action when spawning a foreground job and include the appropriate flag.

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, as described above.

3.6 Managing The Terminal's State

When a job uses the terminal, the job may change terminal settings - for instance, editors such as vim put the terminal in a special state where keys are not immediately echoed. When such jobs are stopped and later resumed, they should find the terminal in the state they put it in, as if they had their own virtual terminal to use whenever they're in the foreground.

Your shell should manage this by saving the terminal state when a job is stopped and restoring it when the user continues it. This can be done with the `Terminal::get_terminal_state` and `Terminal::give_terminal_to` methods.

Before the shell awaits user input at the prompt, it must reclaim ownership of the terminal via the `Terminal::reclaim_terminal` method. Finally, the shell should allow user programs such as `stty` to change the default terminal settings, which it should do by sampling (=reading) the terminal state such programs left when they exited. For simplicity, your shell will not have a list of programs that are allowed to change the default settings, but rather it will read back the terminal state whenever a foreground process exited normally with exit status 0 - an example of a “good enough” heuristic given that few programs will actually change the terminal state unintentionally.

A more detailed explanation is given in Appendix A.2.

3.7 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 3.5. 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 create pipes, use the `pipe2(2)` system call abstracted by the `nix::unistd::pipe2` function. The function allows you to set flags on the returned file descriptors such as `O_CLOEXEC`, which is needed to ensure the process spawned by `posix_spawn` closes any pipe file descriptors after redirecting them as necessary.

A pipe must be set up by the parent shell process before a child is forked. A pipeline consisting of n commands requires $n-1$ separate pipes, which means you need to call `pipe2` $n-1$ times. You can do this either ahead of time before spawning any of the child processes, or create one pipe each time you spawn a child process (except for the last one).

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. To be precise, the k^{th} pipe's write end must be connected to the k^{th} command's standard output, and the k^{th} pipe's read end must be connected to the $(k+1)^{\text{th}}$ command's standard input stream.

Since these actions are performed with the `posix_spawn` function, these actions must be arranged via the `PosixSpawnFileActions::add_dup2` method. If the user used the `|&` instead of the `|` symbol, both standard output and standard error should be redirected to the pipe, which can be accomplished with an additional `dup2` action that would duplicate standard output to standard error.

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, which is elaborated on in Appendix A.3.

Any I/O redirections necessary can be done similarly to pipes, by redirecting the standard input or standard output file descriptors to the path provided by the user. This must be done with the `PosixSpawnFileActions::add_open` method.

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 learns 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.

4 The Codebase

The provided base code contains two *packages* in a single *workspace*. You will write all your code in the `cushion` package. The *shelltest* package creates the binary that you can use to test the functionality and correctness of your shell.

4.1 Cushion

You can build this package in debug mode by running `cargo build`, or build and run this package using `cargo run`. Note that the `cushion` binary generated by the package will be placed in `./target/x86_64-unknown-linux-gnu/{debug,release}` instead of the usual `./target/{debug,release}` due to the platform configuration we are using for `rlogin`.

The following files/modules are provided to you:

File/Module	Description
main.rs	The root of your binary crate. This contains the main loop of your shell, and is where you will implement the basic program flow.
lib.rs	The root of your library crate. This is where the `Shell` struct is provided, which contains the components used in your shell.
ast/	The command line parser and data structures. You should understand the data structures provided in `ast/mod.rs` and what each field represents. You **should not** edit this module.
terminal.rs	Utilities for managing terminal state. You should understand the methods provided in this module, and call them as necessary. You **should not** edit this file.
sigchld.rs	Utilities for handling SIGCHLD in a background thread as mentioned in Section 3.3. You should not need to use anything provided in this module. You **should not** edit this file.
args.rs	Command line argument handling. You should not need to use anything provided in this module. You **should not** edit this file.
signal.rs	Utilities for blocking and unblocking signals. You should not need to use anything provided in this module. You **should not** edit this file.

You are free to create your own modules to organize your code. For example, consider creating a module to represent jobs, and/or a module to represent a collection of jobs. How would you model the relation of jobs to their job ID? How would you model the relation between jobs and the processes associated with a job?

You are allowed to use any dependencies provided in `Cargo.toml`. You are **not** allowed to use any dependencies not listed in `Cargo.toml` without the permission of the course instructors. Please contact staff if you would like to use a dependency not listed, along with justification.

4.2 Shelltest

This package contains the test binary that you can use to ensure the correctness of your shell. You can compile and run the test binary with `cargo run --release -p shelltest`, however it is recommended to use the provided wrapper script `test_cushion.sh`. You will need to pass the script the path to your shell binary, as well as the path to the test configuration file. A sample test configuration file is provided as `test_config.toml`.

5 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/cs3984-systems-rust/project-1-cushion,

One team member should 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:teammemberwhoclonedit/project-1-cushion.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.

6 Builtins

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

For extra credit, you may implement additional builtins that you must then document the design and functionality of. Any extra credit awarded is under the purview of the course instructors. Some examples for builtins you might implement are:

- Setting and unsetting environment variables
- 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.
- Glob expansion (e.g., *.c).
- 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.

You may ask for permission to use specific crates to aid in implementing the builtins. We expect that you perform significant integration of any crates used into your shell (e.g. you are not allowed to make your builtin a simple alias for functionality a crate provides). If in doubt, ask.

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.

7 Requirements

Rubrics. This project will account for 120 points. Passing the basic tests will award 55 points. Passing the advanced tests will award 50 points. Proper documentation of your will award 10 points. Proper use of version control will award 5 points. In addition, deductions may be taken for deficiencies in coding style and lack of robustness.

These numbers are subject to change, and any changes will be announced.

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. Your code should compile with no warnings from the Rust compiler. This will be enforced in the autograder.

You must address any errors returned by system calls and library functions. You may not silently ignore any errors returned, and you may only `unwrap` or `expect` any results obtained if failing calls constitute a bug in your program, or if any error received is unrecoverable.

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, cd, history, \^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>
```

A grade penalty will be given for a missing or incomplete design document.

Good Luck!

A Appendix

A.1 Signal Handling Approach

If processes running in the background exit during while the shell is waiting at the prompt, the shell is notified via the asynchronous signal SIGCHLD. The shell must act on this signal and call `waitpid()` in a timely manner because waiting on a terminated child is necessary for the operating system to release the resources associated with the child. Otherwise, the shell accumulates zombie children.

Due to the asynchronous nature of signal delivery, programs register functions known as *signal handlers* that will be called when the specified signal is delivered by the operating system. Execution of the program is suspended when the signal handler is called, and execution continues when the signal handler returns. This means that if data structures are accessed in a signal handler, care must be taken to avoid receiving the signal (thereby executing the signal handler) during parts of the program that modify the same data structures [4].

In Rust, apart from complications surrounding signal handlers, accessing shared data structures requires managing the ownership and borrowing rules required by the compiler. Therefore, to maintain the notion of a single flow of control, for this project your shell will have a different design, which we provide:

1. The shell will spawn a background thread whose job is to call `waitpid()` when SIGCHLD is received. The status of each child process that have changed is then sent over a *channel*, a concurrent data structure used for message passing. [5]
2. The main thread (your shell) will use a blocking read on the channel to wait for a change in status (ie. when your shell is waiting on a foreground job). This corresponds to `SigchldHandler::get_status` in the provided code.
3. The main thread (your shell) will use a non-blocking read on the channel to check if any child processes have changed state while it was blocked waiting for user input. This corresponds to `SigchldHandler::try_get_status` in the provided code.

As noted, the base code provides facilities that handle the asynchronous signal delivery and reaping of child processes for you. Therefore, you need not concern yourself with the concept of multithreaded execution. However, you will have to handle the statuses of child processes and modify any job data structures accordingly.

A.2 Background on Terminal State Management

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

[4]For more information, see `signal-safety(7)`. Note that signal handlers are hard to write correct, and cause many security complications in real world code.

[5]Note that in the background thread, the signal handler registered for SIGCHLD does not access the channel. Rather, a flag is set that is then read by the background thread outside of the signal handler. This is transparently handled by the `signal-hook` crate.

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 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.[6] This can be done with the `Terminal::get_terminal_state` and `Terminal::give_terminal_to` methods.

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. This can be done with the `Terminal::sample` and `Terminal::reclaim_terminal` methods.

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, if the shell always restored the same good terminal state it sampled when the shell itself was started, then the `stty` command would not work - while it could change the terminal state, any such changes would be undone once the shell learns that the stty command has exited and returns to the prompt. To avoid that, shells sample the terminal state when a job has exited and replace their known good state with the sampled state. Your shell should do the same, but only if

- the job exited (didn't terminate with a signal)
- the job was in the foreground at the time it exited
- its exit status as reported by the wait system call was 0

[6]This 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.

A.3 File Descriptor Management

To understand why the parent shell process must close the file descriptors referring to the pipes it creates for its child processes, 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.

A.4 Wait Status

Here is a brief table summarizing facts about the status changes and the corresponding variants of the `WaitStatus` enum returned by `waitpid`[7]:

Event	Variant	Additional info	Process stopped?	Process dead?
User stops fg process with Ctrl-Z	`WaitStatus::Stopped`	`Signal` is SIGTSTP	yes	no
User stops process with kill -STOP	`WaitStatus::Stopped`	`Signal` is SIGSTOP	yes	no
Non-foreground process wants terminal access	`WaitStatus::Stopped`	`Signal` is SIGTTOU or SIGTTIN	yes	no
Process exits via `exit()`	`WaitStatus::Exited`		no	yes
User terminates process with Ctrl-C	`WaitStatus::Signaled`	`Signal` is SIGINT	no	yes
User terminates process with kill	`WaitStatus::Signaled`	`Signal` is SIGTERM	no	yes
User terminates process with kill -9	`WaitStatus::Signaled`	`Signal` is SIGKILL	no	yes
Process has been terminated (general case)	`WaitStatus::Signaled`		no	yes

Additional information on signals can be found in the GNU C library manual, available at `http://www.gnu.org/s/libc/manual/html_node/index.html`. Read, in particular, the sections on Signal Handling and Job Control.

[7] A 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 where 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.