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. You must include a `partner.json` file in your submission (see the Submission section for details).

1 Introduction

This assignment introduces you to the principles of process management and the command language used by the standard shell in a Unix-like operating system. In this project, you will develop a simple command language interpreter that implements a subset of bash's functionality.

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

1.1 Basic Shell Functionality

As described in System Interfaces volume of POSIX.1-2024 the shell is a command language interpreter that receives input from a file, breaks the input into tokens, words, and operators, then parses the input into simple or compound commands, performs various substitutions, redirections, and executes executable files specified by the user. Lastly, the shell will optionally wait for each command to complete.

Shells may receive their input from a regular file or by reading their standard input. We refer to such inputs as shell scripts. Job control shells will also perform a set of management tasks that allows interactive use of the shell with a terminal. This project does not focus on interactive use and job control, but rather on the abilities of the shell as an interpreter of a simple command language. Specifically, your shell must be able to execute shell scripts provided as files (i.e., `./minibash scriptfile.sh`) and also accept input interactively via `readline`. The provided tests exercise the script-based mode exclusively.

A shell receives line-by-line input from a script that represents user commands. Some user commands are builtins, which are implemented by the shell itself. If the input contains the name of such a built-in command, the shell should execute this command immediately. Otherwise, the shell should 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 its input file. However, if the user appends an ampersand '&' to a command, the command is started and the shell will continue with the next command immediately. In this case, we refer to the running command as a "background job," whereas commands the shell waits for before processing the next command are called "foreground jobs."

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 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. Note that there may not be any notifications in the common case in which a foreground command finished successfully.

2 Strategy

2.1 Parsing the Input

We will be using the Tree Sitter library for our parser. Tree-sitter is a parser generator and incremental parsing library that produces C code that can be embedded in any application. We are using the bash grammar for tree sitter. A grammar for a language is a formal description of its syntax. A parser is a program that can check whether a given input string obeys this syntax. If so, it will deliver a syntax tree, which is a tree data structure that represents the recognized (or "parsed") syntax.

Since tree sitter was originally developed for tasks such as syntax highlighting, it produces a concrete syntax tree rather than an abstract syntax tree. This means that the parse tree contains nodes for all tokens that are part of the input, including special characters, semicolons, parentheses, etc. However, tree-sitter grammars may name non-terminal nodes as well. The resulting C API is very compact; the documentation is provided here.

Consider the following example input:

```
# echo two strings, reverse them and write the results to a file
echo hsab inim | rev > "outputfile.$$"
```

When using the `tree-sitter parse` command, it will output only named nodes:

```
(program [0, 0] - [2, 0]
  (comment [0, 0] - [0, 64])
  (redirected_statement [1, 0] - [1, 38]
    body: (pipeline [1, 0] - [1, 20]
      (command [1, 0] - [1, 14]
        name: (command_name [1, 0] - [1, 4]
          (word [1, 0] - [1, 4]))
        argument: (word [1, 5] - [1, 9])
        argument: (word [1, 10] - [1, 14]))
      (command [1, 17] - [1, 20]
        name: (command_name [1, 17] - [1, 20]
          (word [1, 17] - [1, 20])))))
    redirect: (file_redirect [1, 21] - [1, 38]
      destination: (string [1, 23] - [1, 38]
        (string_content [1, 24] - [1, 35])
        (simple_expansion [1, 35] - [1, 37]
          (special_variable_name [1, 36] - [1, 37])))))))
```

Named nodes here are *program*, *comment*, *redirected_statement*, *body*, *pipeline*, and so on. The

indentation level determines the tree structure of the parse tree - nodes that have identical levels of indentation are at the same level in the parse tree. You can iterate over a nodes' named children using `_named_` variants of the `ts_node_` functions.

Tree-sitter also provides the ability to specify field names for certain nodes; these are shown using a colon, e.g., the *redirected_statement* node contains two child nodes with names *body:* and *redirect:*. For some grammar rules, the use of these field names to find child nodes may be useful, but unfortunately the bash grammar doesn't always guarantee that field names are unique.

The numbers in square brackets in the output of the parse command refer to the input location taken up by the parsed nonterminal symbol - for instance, `[0, 0] - [2, 0]` means that it starts at (line, col) of (0, 0) and extends to line 2 and column 0 (which is the entire program). `[1, 5] - [1, 9]` refers to line 1 columns 5 to 9, which is where the word `hsab` is located.

The bash grammar distributed as part of tree-sitter is comprehensive but complex; while you may refer to it as reference, we found it easier to simply rely on the parse command to examine the syntax trees for specific features. In other words, if you wish to implement a specific bash feature, write a script that uses it, then examine the parse tree that results when this script is being parsed by tree sitter.

2.2 Expected Features

This project is open-ended - with a complete grammar, you could, time and resources permitting, reimplement all of bash (or a similar, POSIX-compliant shell). We will limit what we expect based on the tests we provide (if you implement more features, you will need to write your own tests.)

The provided tests define the required functionality. Current expectations include:

1. The ability to run simple commands and wait for them. Background execution with `&` may also be tested.
2. Builtins: at a minimum, your shell should recognize the `exit` builtin to terminate the shell. Additional builtins for job management (`kill`, `stop`, `jobs`, `fg`, `bg`) are needed only if you implement interactive job control, which is not tested.
3. The ability to run pipelines of commands. The exit status of a pipeline (`$?`) must be the exit status of the *last* command in the pipeline, matching bash's behavior.
4. Common forms of redirection (from a file, to a file), including redirecting standard error using the `|&` and `>&` syntax.
5. Handling common constructs such as single and double quotes.
6. Substitution of environment and shell variables as well as special builtin variables such as `$?`, `$$` (the shell's own PID), and `${varname}` syntax. If a child process is terminated by a signal, `$?` must be set to `128 + signal_number`, following bash convention.

7. Command substitution using `$(cmd ...)` syntax. Backtick syntax is not tested.
8. Logical operators that form lists, including `&&` and `||`. These operators have equal precedence and are evaluated left-to-right.
9. Boolean expressions via the external `test(1)` command, which is also invoked via the `[ ... ]` syntax. The bash-specific `[[ ... ]]` form is not required.
10. `if/elif/else`, `for`, and `while` loops. The `break` statement inside loops must also be supported.

2.3 Implementing Jobs, Pipelines, and I/O Redirection

Jobs may consist of multiple commands, or they may consist of only a single command. A pipeline of commands is considered one job. Each command will be run in its own process.

Your shell must create a separate process group for each job, no matter whether the job initially contains only a single process or multiple. All processes that form part of a pipeline must thus be part of the same process group. See Appendix A for details. Note: although process groups are primarily used for interactive job control (which is not tested), they are still required because the provided `wait_for_job` function and the `sigchld_handler` rely on processes being organized into groups for correct bookkeeping.

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`. See the Appendix C for more details.

Your shell must use signals and forms of the `waitpid()` system call to learn from the OS about the outcomes of the child processes it starts as part of each job, see the appendix sections B and D for details.

2.4 Creating processes with `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.

For your implementation, you should use `posix_spawn` in lieu of `fork + exec`. The `posix_spawnp(3)` function is part of the C library (glibc) and is declared in `<spawn.h>`. You do not need to build or install any additional library to use it. Your implementation will thus 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.

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

- Use the `posix_spawnp` 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_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 will need to pass the current environment as the last argument. Add an external declaration like so `extern char **environ;`.

3 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: `https://git-scm.com/docs`, but feel free to ask questions on the class Discourse forum (`https://cs3214.cs.vt.edu`) 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/minibash,

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:teammemberwhoclonedit/minibash.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/-/user_settings/ssh_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. Alternatively, you may also create and use access tokens.

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, you must 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.

3.1 Code Base

To build the provided code, read the README.md file. You need to build the tree-sitter and tommyds libraries first, then build your shell in the `src/` directory:

```
(cd tommyds; make)        # build the hash table library
(cd tree-sitter; make)    # build the tree-sitter core library
cd src && make            # build minibash
```

The tree-sitter bash grammar (`tree-sitter-bash/`) is compiled inline by the `src/Makefile`.

Read the provided code first.

3.2 Provided Code Overview

The following files are provided in the `src/` directory. You should read and understand them before writing code.

`minibash.c` The main shell source file. Contains the read/eval loop, job data structures (`struct job`, `job_list`, `jid2job[]`), the `sigchld_handler`, `wait_for_job`, and the stub `handle_child_status` and `run_program` functions that you must implement.

`signal_support.h / .c` Wrappers for POSIX signal operations: `signal_block()`, `signal_unblock()`, `signal_is_blocked()`, and `signal_set_handler()`. These simplify signal mask manipulation.

`hashtable.h` A convenience wrapper around the `tommyds` hash table library. Provides `hash_put(ht, key, value)`, `hash_get(ht, key)`, and `hash_del(ht, key)` for managing shell variables as string→string mappings. The shell's variable table (`shell_vars`) is already initialized in `main()`.

`ts_helpers.h` Helper functions for working with tree-sitter parse tree nodes, including:

- `ts_extract_node_text(input, node)` – extracts the source text corresponding to a parse tree node (returns a newly allocated string that you must `free`).
- `ts_extract_node_length(node)` – returns the byte length of a node's text without extracting it.
- `ts_peek_at_node_text(input, node)` – returns a pointer into the original input (not zero-terminated; use cautiously).
- `ts_print_node_info(node, label)` – prints debugging information about a node.

`ts_symbols.h` An enumeration of all tree-sitter symbol identifiers for the bash grammar. Use these constants to check the type of a parse tree node, for example: `ts_node_symbol(no == sym_pipeline`.

`utils.h / .c` Utility functions: `utils_error()` prints an error message with `errno` information; `utils_fatal_error()` does the same and exits; `utils_set_cloexec(fd)` sets the close-on-exec flag; `utils_string_concat(s1, s2)` concatenates and frees both inputs.

`list.h / .c` An intrusive doubly-linked list implementation (from Pintos). Used for the job list. The key pattern is: embed a `struct list_elem` in your structure and use `list_entry()` to convert back. See the detailed comments in `list.h`.

In addition, tree-sitter field IDs (`bodyId`, `nameId`, `conditionId`, etc.) are initialized in `main()` and can be used with `ts_node_child_by_field_id()` to access named children of certain grammar rules.

4 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.

We expect that your shell does not have memory leaks, which we will use valgrind to test.

Some 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.

4.1 Test Structure

Tests are located in the `tests/` directory. Each test is a shell script (e.g., `050-pipeline-1.sh`) paired with an expected output file (`.out` for exact match, or `.reg` for regex-based matching). The test driver runs your shell with the script as input and compares `stdout` against the expected output.

You must first build the test helper programs:

```
(cd tests; make)
```

4.2 Running Tests

Run the test driver from the `src/` directory (after building `minibash`):

```
minibash_driver.py                # run all tests (3-phase)
minibash_driver.py -b             # run only basic tests
minibash_driver.py -a             # run only advanced tests
minibash_driver.py --no-valgrind  # skip valgrind phase
minibash_driver.py --test 050-pipeline-1  # run a single test
minibash_driver.py --list-tests   # list all available tests
minibash_driver.py -v             # verbose output (shows diffs)
```

On rlogin, `minibash_driver.py` is available in `~cs3214/bin`. Use `../tests/minibash_driver.` if you wish to run your local copy.

4.3 Test Categories and Scoring

Tests are divided into categories based on their numeric prefix:

- **Basic tests** (`001–099`): Simple commands, arguments, quoting, variables, command substitution, pipelines, redirection, and logical operators. These account for **70%** of the functional test points.
- **Advanced tests** (`100+`): Control flow (`if/elif/else`, `for`, `while`), `break`, and complex combinations. These account for **30%** of the functional test points.

The test driver runs in three phases by default:

1. Basic tests (functional correctness)
2. Advanced tests (functional correctness)
3. All tests with valgrind (memory safety, 15 additional points)

5 Grading

Rubrics. This project will account for 100 points, or roughly 1/4 of the achievable project points.

Points are distributed as follows:

- **70 points**: basic tests (tests `001–099`), distributed evenly among tests.
- **30 points**: advanced tests (tests `100+`), distributed evenly among tests.
- **15 points**: memory safety via valgrind, awarded proportionally based on the fraction of tests that pass without memory leaks.

10 points are awarded for correct use of version control. 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. We do not require a design document for this project.

You must submit a .tar.gz file whose root contains the following structure:

```
README.txt
partner.json
src/
  Makefile
  minibash.c
  ... (all other source files)
```

The `partner.json` file must be a JSON array containing the CS login PIDs of both group members. For example, if Alice (`alice01`) and Bob (`bob02`) worked together:

```
["alice01", "bob02"]
```

The `src/` directory must appear as a subdirectory in your tar file and must contain a `Makefile`. 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). You do not need to include the `tommyds/` or `tree-sitter/` directories.

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.

Appendix

The following sections provide background information on process management concepts that are relevant to implementing your shell. Not all of these concepts are tested directly, but understanding them will help you write correct code and debug problems.

A 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[1] 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 targets[2]. 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. 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.

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.

B Processing Job/Process Status Changes

At a given point in time, a shell script 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

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

[2] Note the idiosynchracies 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.

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 running[3], 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).[4]

If such notifications arrive asynchronously, they could arrive at inopportune points in time where the shell is not prepared to handle them. For this reason, we ask the OS to delay their delivery until the shell is ready to handle them. This is done by delaying the delivery of this signal. Note that the shell may also directly ask the OS to block it until any child has changed state - it does that whenever it waits for job, see the provided code.

C Implementing Pipes

A pipe must be set up by the parent shell process before a child is forked, which happens inside the `posix_spawn` system call. 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. When a new process is created, a shallow copy of these descriptors is made. Afterwards, 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.

[3] 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.

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

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.

D Process Status Overview

Here is a brief table summarizing facts about the status changes and the corresponding macros you can apply to the `status` (out) parameter[5] returned by `waitpid`:

[5] 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.

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 `stop` (bash) or `kill -STOP` (bash)	WIFSTOPPED	WSTOPSIG equals SIGSTOP	yes	no
non-foreground process wants terminal access	WIFSTOPPED	WSTOPSIG equals SIGTTOU or SIGTTIN	yes	no
process exits via `exit()`	WIFEXITED	WEXITSTATUS has return code	no	yes
user terminates process with Ctrl-C	WIFSIGNALED	WTERMSIG equals SIGINT	no	yes
user terminates process with kill	WIFSIGNALED	WTERMSIG equals SIGTERM	no	yes
user terminates process with kill -9	WIFSIGNALED	WTERMSIG equals SIGKILL	no	yes
process has been terminated (general case)	WIFSIGNALED	WTERMSIG equals signal number	no	yes

Since we do not focus on interactive job control in this project, we will not test the cases involving `SIGTSTP`, `SIGINT`, `SIGTTIN`, or `SIGTTOU`. Your shell must, however, correctly handle the `WIFEXITED` and `WIFSIGNALED` cases in `handle_child_status`, since the tests rely on correct exit status reporting via `$?`.

Additional information can be found in the GNU C library manual, available at `https://sourceware.org/glibc/manual/latest/html_mono/libc.html`.