Operating Systems Lab 4: Shell

For the last lab of *Operating Systems*, you can choose between two assignments. Either, you can work on implementing background processes and signals in your existing shell, or you can work on implementing your own version of the `exec` system call you've become familiar with in the previous labs. Both options are similarly challenging, and can be tricky to get fully working - so make sure to start early and have plenty of time for debugging! Both assignments offer the opportunity to get bonus points. Note that switching between the assignments midway through is highly discouraged, in case you decide to switch after already making a submission to Themis, please send an email to the helpdesk, so we can make sure to grade the correct submission.

This document describes the shell assignment, for the `exec` assignment refer to the other document on Brightspace.

Shell - Background Processes and Signals

In the last iteration of the shell assignment, you will implement background processes and signal handling. You will implement support for the `&` operator and for handling signals sent to your shell. Additionally, you will implement the commands `jobs` and `kill`, that will list background processes and send a specific signal to a specific background process, respectively.

Background process behaviour

As in a traditional shell, a command (or pipeline) is treated as a 'background command' by using the `&` operator at the end of the command, such as `sleep 10 & echo a`, which will sleep in the background and thus print the character `a` immediately.

Starting background processes should not be very complicated as you effectively already do so if you implemented the pipeline properly. The main difference is that you should not actively wait for the commands in the pipeline to finish before giving back control to the user, and you should not connect `stdin` to the entry of the pipeline. Of course, the output of a background pipeline should still be connected to the terminal `stdout` or a file when output redirection is specified. Consequently, `echo "a" & echo "b" & echo "c" &` should print all these characters, but crucially, the order is undefined.

When the user types `exit` while background processes are still running, your shell should not exit, but instead inform the user about this so they can stop the background processes first: `Error: there are still background processes running!`. For example, `sleep 1 & exit` should print an error and not exit the shell, while `sleep 1 && exit` should exit your shell after one second. Note that reaching `EOF` (pressing Ctrl+D) should always gracefully (i.e. without memory leaks) terminate your shell, possibly leaving behind orphan processes.

You can assume the exit code of a background process to be `0` (since the 'creation of a background process' succeeded). We will not implement a way to track the 'real' exit code of background processes. The exit code of `exit` when there are still processes running is obviously `2`.

Signals

The most versatile way to keep track of background processes is to implement *signal handlers*. In general, signals are a form of inter-process communication that are mainly used to inform a process of lifecycle events: for example, `SIGTERM` instructs a process to shut down, `SIGKILL` immediately kills a process, etc. You can run `man "signal(7)"` in your terminal to explore the various signals that Linux offers, or `man "sigaction(2)"` for more information about handling signals.

Each signal has a default action; typically either the signal will be ignored by default (e.g. for `SIGCHLD`) or the signal will cause the receiving process to be terminated (e.g. for `SIGTERM`, `SIGINT`, `SIGKILL`). It is possible to register a signal handler to implement a custom behaviour in response to a signal; for example you can implement a signal handler to free resources when receiving a `SIGTERM`. You can register such a signal handler by calling `sigaction()` and passing a pointer to a 'callback function' you want to be called when the signal is received. Note that it is not possible to register handlers for `SIGKILL`.

Signal Handling

For this assignment, you should be able to handle at least the following signals: `SIGCHLD` and `SIGINT`. The `SIGCHLD` signal is sent to a parent process when one of its child processes stopped or terminated. You can use this signal to keep track of when background processes exit. Your callback function will receive an instance of the `siginfo_t` struct which you can inspect to find the process ID of the terminated child. With this information, you can keep your background process list up to date.

Note that installing a signal handler does not mean you are no longer required to use `wait` (or `waitpid`) to 'reap' child processes! This call should then be made in the callback function. Alternatively, you can use the `SA_NOCLDWAIT` option when setting up the signal handler (`sigaction()`) to automatically reap processes after delivering the signal, but then it might harder to trace events when a signal message gets lost.

Additionally, you should install a signal handler for `SIGINT` (triggered by Ctrl+C in your terminal). When the shell itself has 'focus' (i.e. no foreground command is running on which the shell is actively waiting), this signal should be equivalent to calling `exit` (and might produce an error when background processes are still running). On the other hand, when a command *is* running in the foreground (and possibly `stdin` is connected to a child process), the signal should be ignored in the shell so that it can be passed on to the child (which presumably then quits).

Take care of managing your signal handlers across `fork`s and `exec`s: signal handlers will be copied to child processes upon `fork` (but will not be carried on to the child program itself across `exec`). Child processes should and will always be started with the default signal handlers.

Using Signals

Be aware that processes that have terminated in response to an unhandled signal (e.g. `SIGINT`, `SIGKILL`, `SIGTERM`) should get a special exit code in your shell to indicate this condition. Use `WIFSIGNALED` and `WTERMSIG` to recover the signal (after you have `wait`ed for your child to finish) and assign exit code `128 + sig` to such a process. We add 128 to indicate the presence of a signal.

Lastly, we should offer some more control to the user by implementing two more built-in commands: `kill {idx} {sig=SIGTERM}` and `jobs`. The first should send the signal `sig` (default `SIGTERM = 15`) to the background process with the given `idx`. This command should not print any output upon success, but print the following in these error conditions:

- `Error: command requires an index!` when no `idx` is provided.
- `Error: invalid index provided!` when an invalid `idx` is provided (not an `int`).
- `Error: this index is not a background process!` when the `idx` provided is unknown.

- 🟥 `Error: invalid signal provided!` when an invalid signal is provided (not an `int`).

Instead of using `pid` to track child processes, we use an index: each background process gets a unique incrementing index, starting at `1`. Your `kill` command should then translate the index to the correct `pid`. This way, the output of a particular script is deterministic for Themis, as `pid`s are not. These indices should not be reused.

The output of `jobs` should either be 🟥 `No background processes!` when none are active, or a list as follows, with the PIDs decreasing:

```
> jobs
Process running with index 3
Process running with index 2
Process running with index 1
```

As with the previous two iterations, submissions should be made to Themis. Your code will automatically be tested, and special attention will be given to ensuring no orphan processes are left behind and that background processes function properly.

Hints

- **Note that a call will return after a signal has been received,** even though the end of file (end of `stdin`) has not been reached. You can prevent this behaviour by setting the flag `SA_RESTART` in your `struct sigaction` handler. This behaviour might complicate working with flex and/or bison.
- Note that after the error condition of `exit` (when background processes are still running), you should hand back control to the user!
- The operators `&`, `&&`, `||`, and `;` all only act on the immediate previous command and have equal 'importance'. This means that a command like `sleep 1 && echo "a" & echo "b"` should first sleep, and then run both `echo`s at the same time with the first one in the background. In contrast, bash treats the `&&` chain with higher importance and will echo

of the bash process itself. If you run this command in 'normal' shell directly, make sure to escape `\$\$` to prevent passing the `pid` of the calling shell.

- You should **not** use `setpgid()` or similar methods, as this might break Themis when you do not terminate child processes correctly! Such an issue will then cause processes to be orphaned, which will keep Themis waiting indefinitely.
- Make sure to disable input and output buffering using `setbuf(stdin, NULL);` and `setbuf( stdout, NULL);`, to prevent out-of-order prints in the Themis output (and consequently failing testcases).
- If you use flex, make sure to set the option `%option always-interactive`.
- Themis will only check `stdout`, so make sure to print any errors there and **NOT** on `stderr`!
- Themis will check for memory leaks using `valgrind` in all testcases! Memory leaks will be indicated by a 'Runtime error' with exit code . Don't worry about memory leaks between a `fork` and `exec`: `valgrind` won't report those as the entire process will be replaced by the `exec` call. Also exiting your child process after a failed `exec` won't be reported to Themis.

Grammar

The full grammar describing the syntax to accept in your shell is defined below. In general, your intuition and experience with shells will be sufficient to know what to allow/forbid in input and note that we will not test 'exotic' inputs in Themis - the objective of this lab is not to build a perfect parser, but a usable shell.

In the grammar, an 'executable' can be the path to a file, or can refer to a file in the user's `$PATH`, while 'builtin' refers to a built-in command. The 'options' part represents any (possibly empty or very large!) set of parameters/strings that should be put into `argv` of the program or command.

From the grammar, it should be clear that a built-in command cannot join any of the I/O redirection and piping that we have implemented in the previous lab. This might simplify your implementation. Additionally, note that any character appearing in the grammar is a 'reserved character' that can only otherwise occur in strings (`""`). Strings are only parsed using double quotes, single quotes can be treated as normal characters.

```
⟨command⟩        ::= ⟨executable⟩ ⟨options⟩

⟨pipeline⟩       ::= ⟨command⟩ | ⟨pipeline⟩
                   | ⟨command⟩

⟨redirections⟩   ::= < ⟨filename⟩ > ⟨filename⟩
                   | > ⟨filename⟩ < ⟨filename⟩
                   | > ⟨filename⟩
                   | < ⟨filename⟩
                   | ⟨empty⟩

⟨chain⟩          ::= ⟨pipeline⟩ ⟨redirections⟩
                   | ⟨builtin⟩ ⟨options⟩

⟨inputline⟩      ::= ⟨chain⟩ & ⟨inputline⟩
                   | ⟨chain⟩ && ⟨inputline⟩
                   | ⟨chain⟩ || ⟨inputline⟩
                   | ⟨chain⟩ ; ⟨inputline⟩
                   | ⟨chain⟩
                   | ⟨empty⟩
```

Bonuses

Once again, there is the possibility to implement some bonuses to obtain up to 2 bonus points (yielding a maximal grade of 12). For this last iteration, these are some of our suggestions:

- **For 4 bonus points** (thus a maximal grade of 14), also implement the userland `exec` assignment.
- Implement a built-in command to limit the CPU time of a child process, or limit other resources. Refer to `man "getrlimit(2)"` and the `SIGXCPU` signal.
- Allow for more fine-grained command ordering, for example by implementing brackets that will order the execution. For example, `(sleep 1 && echo "a") & echo "b"` should then run the `sleep`-`echo` chain in the background as a whole, while printing `b` immediately.
- Implement 'sub-shells' that allow capturing the output of a command as a string, as in the bash command `echo $(pwd)`.
- Handle other signals and implement appropriate actions.
- Keep track of exit codes of background processes and report them to the user.

Note that despite some of these options being the same as in previous labs, you cannot get bonus points for a bonus more than once. Of course, you can get as creative as you want and do more, we will award points proportional to the features you implemented with a cap of 2 points. If you decide to implement some of these bonuses, make sure to make them togglable using a compilation flag (`#if EXT_PROMPT`, compile with `-DEXT_PROMPT`) and submit to Themis in the separate 'bonuses' entry. Make sure to include a ≡ `README` documenting your work - we can only award points for bonuses discussed there.