CS 2810: Shell

In this project you will build a shell using Python. This will give you practice with Linux system calls and will help you understand process job control.

The requirements

Your shell should support the following. Implement the features in this order for part 1:

1. When launched, the shell should display a prompt that includes the current working directory. It should accept a line of input and display a prompt each time.
2. The shell should understand the `exit` command, which closes the shell. It should also repond to end-of-file (when the user types ctrl-d) the same way.
3. The shell should have a built-in `cd` command that allows the user to change directory. When this is run without any arguments, it should change to the user's home directory (the environment variable HOME).
4. The user should be able to type a command such as `ls` with no arguments. Your shell will launch it, block until it completes, and then resume. If the command exits with a non-zero exit status code, you should display a message to the user.
5. The shell should parse commands with arguments, for example `ls -l`.
6. The shell should understand redirecting the output of a command to a file, e.g., `ls -l > output.txt`.
7. The shell should understand redirecting the input of a command from a file, e.g., `grep hello < input.txt`.

For part 2, implement the following features:

1. The user should be able to pipe the output of one command to the input of another command, e.g., `ls -l | grep txt`. You do NOT need to handle pipes involving more than two commands.
2. The user should be able to type `&` at the end of the line to run a command in the background. After launching the command, the shell should immediately print a prompt and start accepting additional commands. When the background process completes, the shell should print a message indicating that it exited (and display a message if the exit code was non-zero).
3. The shell should have a built-in `jobs` command that lists all current background jobs running including the command used to launch each one.

System calls in Python

Python gives access to the system calls we will need in the `os` package. Here are a few tips:

- To read a line of input, use `sys.stdin.readline()`. If it returns an empty string, that indicates end-of-file (the user has probably typed ctrl-d).
- You can write to standard out using `sys.stdout.write(s)`.
- To get the current working directory, use `os.getcwd()`.
- If you want to know the user's home directory, look up the HOME environment variable using `os.getenv('HOME')`.
- To invoke the exit system call, use `sys.exit(status)`.
- To change a directory, use `os.chdir(newdirectory)`.
- To fork, use `os.fork()`.
- To exec, use `os.execvp(cmd, args)`.
- To wait for a specific child process to exit, use `os.waitpid(pid, 0)`. To wait for any child process to exit, use `os.waitpid(0, 0)`.
- `waitpid` returns a pair of values (pid, status). pid is the pid of the child that had an event, and status is an encoded value with the details. Use `os.WIFEXITED(status)` to check if the child exited, and if so, use `os.WEXITSTATUS(status)` to get its exit status value. Use `os.WIFSIGNALED(status)` to check if the child was killed by a signal, and if so, use `os.WTERMSIG(status)` to get the number of the signal that killed it.
- To check if any child has exited (but not wait), use `os.waitpid(0, os.WNOHANG)`
- To open a file for writing, use `fd = os.open(filename, os.O_WRONLY | os.O_TRUNC | os.O_CREAT,`

`0644)` (for Python 3, use 0o644)
- To open a file for reading, use `fd = os.open(filename, os.O_RDONLY)`
- To close a file descriptor, use `os.close(fd)`
- To copy a file descriptor, use `os.dup2(oldfd, newfd)`
- To register a signal handler, define a function and then call `signal.signal(signal.SIGCHLD, yourfunction)`. The function you register must take two arguments, but you can ignore them.
- Python has many useful string methods. Try looking in the Python documentation for strings to see what is available.

Overall structure of the final shell

In the final version of the shell, there are essentially two independent loops running, which communicate using some global variables. Let's start with the global variables:

- Queue of undisplayed messages
- Dictionary of foreground jobs
- Dictionary of background jobs

These global variables are used by both the main command-line loop, and also by the grim reaper (the signal handler, detailed below). The main loop proceeds as follows:

- Loop until exit or ctrl-d
    - while there are active foreground jobs, sleep (use `time.Sleep(1)`
    - no foreground jobs left, so display the queue of messages and empty the queue
    - display a prompt, get a line of input, etc.
    - launch a new job, which may include redirects or pipes
        - if launching foreground jobs, add each one to the foreground jobs dictionary
        - if launching background jobs, add each one to the background jobs dictionary

The details of parsing commands and launching jobs are omitted here.

The second part is the signal handler. It is just a function, but after registering it using `signal.signal(signal.SIGCHLD, yourfunction)`, it will automatically be called when the operating system has a notification for you about a child process. It acts as the grim reaper: it gathers information about any deceased child processes, updates the global variables, and then returns. It does not run in an infinite loop (at least not a long-lived one), because the operating system will call it again when there is more to report.

- Loop as long as there are background or foreground jobs
    - call waitpid with the WNOHANG option to check if there is a dead child process
    - if it returns a pid of 0 (to indicate no dead child processes), break out of the loop and return from the handler function
    - if a child exited or was killed by a signal, delete it from the appropriate dictionary (since it is no longer running) and add a message to the message queue

So the grim reaper part runs asynchronously, gathering information about dead children and recording that information in global variables. The main loop checks the global variables every time it is about to display a prompt. If there are active foreground jobs, it sleeps until there are no more foreground jobs (which the grim reaper will discover and communicate to the main worker via the global variables). After that, it displays all the queued messages, then finally displays a prompt, reads a line of input, parses it, and runs a command.