Operating Systems (0368-2162)

Mini Shell Assignment

Individual work policy

The work you submit in this course is required to be the result of your individual effort only. You may discuss concepts and ideas with others, but **you must program individually**. **You should never observe another student's code**, from this or previous semesters.

Students violating this policy will receive a **250 grade in the course** ("did not complete course requirements").

1 Introduction

The goal of this assignment is to gain experience with process management, pipes, signals, and the relevant system calls. To do this, you will implement a simple shell program. You will implement the function that receives a shell command and performs it. We provide a skeleton shell program (`shell.c`), which reads lines from the user, parses them into commands, and invokes your function. You also have to implement any initialization/finalization code that you require (more details below).

1.1 The shell skeleton

The skeleton we provide executes an infinite loop. Each iteration does the following:

1. Reads a string containing a shell command from standard input. If Ctrl-D is pressed, the skeleton exits.
2. Parses the command string into an array of *words*. A *word* is a non-empty sequence of non-whitespace characters, where whitespace means space, tab (`\t`), or newline (`\n`). The end of the array is designated with a `NULL` entry.
3. Invokes your function, passing it the parsed command line array. The skeleton detects and ignores empty lines (i.e., won't invoke your function on an empty command).

1.2 Shell functionality

The shell will support the following operations, which are described in more detail in Section 2.2:

1. **Executing commands.** The user enters a command, i.e., a program and its arguments, such as `sleep 10`. The shell executes the command and waits until it completes before accepting another command.
2. **Executing commands in the background.** The user enters a command followed by `&`, for example: `sleep 10 &`. The shell executes the command but does not wait for its completion before accepting another command.

3. **Single piping.** The user enters two commands separated by a pipe symbol (`|`), for example: `cat foo.txt | grep bar`. The shell executes both commands concurrently, piping the standard output of the first command to the standard input of the second command. The shell waits until both commands complete before accepting another command.

4. **Input redirecting.** The user enters one command and input file name separated by the redirection symbol (`<`), for example: `cat < file.txt`. The shell executes the command so that its standard input is redirected from the input file (instead of the default, which is to the user's terminal). The shell waits for the command to complete before accepting another command.

5. **Output redirecting.** The user enters one command and output file name separated by the redirection symbol (`>`), for example: `cat foo > file.txt`. The shell executes the command so that its standard output is redirected to the output file (instead of the default, which is to the user's terminal). If the specified output file does not exist, it is created. If it exists, it is overwritten. The shell waits for the command to complete before accepting another command. By default, stdout and stderr are printed to your terminal. But we can redirect that output to a file using the (`>`) operator. The `> file.txt` does two things: A) It creates a file named "file" if it does not exist, and B) it replaces the content of "file" with new contents.

The shell doesn't need to support built-in commands such as `cd` and `exit`. It only support execution of program binaries as described above.

ASSUMPTIONS. You can assume the following:

1. If a command line contains the `|` symbol, then (1) it appears only once and (2) it is correctly placed, i.e., at least one word appears before and after it, and it is separated by whitespace from them.

2. If a command line contains the `&` symbol, then it appears last on the command line and is separated by whitespace. In other words, it is the last word of the command line.

3. If a command line contains the `<` symbol, then it appears one before last on the command line and is separated by whitespace. In other words, the redirection operator and file name are the last two words on the command line.

4. If a command line contains the `>` symbol, then it appears one before last on the command line and is separated by whitespace. In other words, the redirection operator and file name are the last two words on the command line.

5. A command line will contain at most one of the `|`, `&`, `<` and `>` symbols. In other words, pipes, background execution, input redirection and output redirection are not combined.

6. Command lines will not contain quotation marks or apostrophes. (In normal shells, these are used to support arguments that contain whitespace. Your shell doesn't need to support this feature.)

2 Assignment description

Implement the following functions in a file named `myshell.c`: `prepare()`, `process_arglist()`, and `finalize()`. The following details the specifications of these functions.

2.1 `int prepare(void)`

The skeleton calls this function before the first invocation of `process_arglist()`. This function returns 0 on success; any other return value indicates an error.

You can use this function for any initialization and setup that you think are necessary for your `process_arglist()`. If you don't need any initialization, just have this function return immediately; but you must provide it for the skeleton to compile.

2.2 `int process_arglist(int count, char **arglist)`

Input This function receives an array `arglist` with `count` non-NULL words. This array contains the parsed command line. The last entry in the array, `arglist[count]`, is NULL. (So overall the array size is `count+1`.)

Behavior

1. Commands specified in the `arglist` should be executed as a child process using `fork()` and `execvp()` (**not** `execv()`). Notice that if the `arglist` array contains a shell symbol (`&`, `|` `<` or `>`), the `arglist` should not be passed to `execvp()` as-is (more details below).

2. Executing commands in the background:

   - If the last non-NULL word in `arglist` is `"&"` (a single ampersand), run the child process in the background. The parent should not wait for the child process to finish, but instead continue executing commands.
   - Do not pass the `&` argument to `execvp()`.
   - Assume background processes don't read input (`stdin`).
   - Do not use fork twice. Only use a single call to fork.

3. Piping:

   - If `arglist` contains the word `"|"` (a single pipe symbol), run two child processes, with the output (stdout) of the first process (executing the command that appears before the pipe) piped to the input (stdin) of the second process (executing the command that appears after the pipe).
   - To pipe the child processes input and output, use the `pipe()` and `dup2()` system calls.
   - Use the same array for all `execvp()` calls by referencing items in `arglist`. There's no need to allocate a new array and duplicate parts of the original array.

4. Input redirection:

   - If arglist contains the word `"<"` (a single redirection symbol), open the specified file (that appears after the redirection symbol) and then run the child process, with the input (stdin) redirected from the input file.
   - To redirect the child process' input, use the `dup2()` system call.

5. Output redirection:

   - If arglist contains the word `">"` (a single redirection symbol), open the specified file (that appears after the redirection symbol) and then run the child process, with the output (stdout) redirected to the output file.

- To redirect the child process' output, use the `dup2()` system call.

6. Handling of `SIGINT`:

   - **Background on SIGINT** (this bullet contains things you should know, not things you need to implement in the assignment): When the user presses Ctrl-C, a `SIGINT` signal is sent (by the OS) to the shell and all its child processes. The `SIGINT` signal can also be sent to a specific process using the `kill()` system call. (There's also a `kill` program that invokes this system call, e.g., `kill -INT <pid>`.)
   - After `prepare()` finishes, the parent (shell) should not terminate upon `SIGINT`.
   - Foreground child processes (regular commands or parts of a pipe) should terminate upon `SIGINT`.
   - Background child processes should not terminate upon `SIGINT`.
   - **NOTE:** The program `execvp()`ed in a child process might change `SIGINT` handling. This is something the shell has no control over. Therefore, the above two bullets apply only to (1) the behavior before `execvp()` and (2) if the `execvp()`ed program doesn't change `SIGINT` handling. Most programs don't change the `SIGINT` handling they inherent, so these bullets apply to basically any program you are likely to test with (`sleep`, `ls`, `cat`, etc.).
   - **IMPORTANT:** To use some signal-related system calls, C11 code must have a certain macro defined. Be sure to use the compilation command line provided in Section 4 or your code might not compile.
   - You may only use the `signal` system call to set the signal disposition to `SIG_IGN` or to `SIG_DFL`. For any other use, particularly setting a signal handler, you must use the `sigaction` system call.

7. Assume that the results of the provided parser are correct.

8. You should prevent zombies and remove them as fast as possible.

Output

1. The `process_arglist()` function should not return until every foreground child process it created exits.

2. In the original (shell/parent) process, `process_arglist()` should return 1 if no error occurs. (This makes sure the shell continues processing user commands.) If `process_arglist()` encounters an error, it should print an error message and return 0. (See below for what constitutes an error.)

Error handling

1. If an error occurs in the shell parent process, there's no need to notify any running child processes.

2. If an error occurs in a signal handler in the shell parent process, there's no need to notify `process_arglist()`. Just print a proper error message and terminate the shell process with `exit(1)`.

3. If `wait`/`waitpid` in the shell parent process return an error for one of the following reasons, it is not considered an actual error that requires exiting the shell:

   - `ECHILD`

- `EINTR`. (You can also avoid an `EINTR` "error" in the first place. Hint: read about the `SA_RESTART` option in `sigaction`.)

Read the `wait` documentation to understand what these reasons mean.

4. If an error occurs in a child process (before it calls `execvp()`), print a proper error message and terminate only the child process using `exit(1)`. Nothing should change for the parent or other child processes.

5. User commands might be invalid (e.g., a non-existing program or redirecting to a file without write permission). Such cases should be treated as an error in the child process (i.e., they must not terminate the shell).

6. Error messages should be printed to `stderr`. (This applies to both parent and child processes.)

7. Error message do not have to be worded exactly as in a real shell. Anything returned by `strerror()` is ok.

8. There's no requirement to exit "cleanly" on error. Processes may terminate without freeing memory.

2.3 `int finalize(void)`

The skeleton calls this function before exiting. This function returns 0 on success; any other return value indicates an error.

You can use this function for any cleanups related to `process_arglist()` that you think are necessary. If you don't need any cleanups, just have this function return immediately; but you must provide it for the skeleton to compile. Note that cleaning up the `arglist` array is **not** your responsibility. It is taken care of by the skeleton code.

3 General guidelines

1. Learn about and use the following system calls: `dup2()`, `execvp()` (not to be confused with `execv()`!), `getpid()`, `open()`, `pipe()`, `sigaction()`, `SIGCHLD`, `wait()`, `waitpid()`.

2. In general, your shell will behave like the standard `bash` shell. But there is an important exception to be aware of: standard shells use a special OS mechanism that prevents `SIGINT` from being sent to background processes when the user presses Ctrl-C (read about `setpgid()` and this link if you're curious). Your shell won't use this mechanism, so the behavior of background processes receiving a SIGINT won't be the same as in bash.

3. For debugging, the `strace` program can be useful. It provides a trace of all system calls executed by a process, including (optionally) by child processes that the traced process creates.

4 Submission instructions

1. Submit just your `myshell.c` file. (Do not assume you can change the `shell.c` file!) Document your code with explanations for every non-trivial part of your code. Help the grader understand your solution and the flow of your code.

2. **The program must compile cleanly on the course udocker image** (no errors or warnings) when the following command is run in a directory containing the source code files:

```
gcc -O3 -D_POSIX_C_SOURCE=200809 -Wall -std=c11 shell.c myshell.c
```