Session 2: Implementation of a shell and Memory management - Report

Tom Apol (s2701650)
Andrei Oanca (s3542505)

March 18, 2018

1 Implementing your own shell – iwish

The functionality of our shell is divided into four parts:

- Reading input
- Tokenising input (here: splitting input into whitespace-separated words)
- Parsing input (and building a datastructure that represents the structure of the input following our grammar, for easier interpreting)
- Interpreting input (and executing the commands)

Furthermore, the shell is structurally divided into two parts:

- The shell itself
- A multitude of job handlers, one for each input.

The existence of the job handlers is mainly to separate the concerns of executing the commands from the user interface. Furthermore, this allows for future extension of the shell's features by interrupting commands without interrupting the entire shell. (i.e. handling `^C` inputs)

In terms of actual behaviour, we have tried to stick as close to the in the exercise described features and bash's behaviour as possible:

- Program execution: Any program can be executed, provided a valid path is given, not just programs in the user's seach path.
- I/O redirection and piping: all redirections will be processed, which may result in the creation or truncation of specified output files that do not receive any output. Pipes will be connected last, in order to preserve the pipeline.
- Background processes: a background process will print its output and terminate whenever it desires. The user will not receive any notification of this happening. (unless that process prints to the shell's standard output, of course.)

The grammar our shell uses is a slightly tweaked version of the given LL grammar for recursive descent parsing, where newlines (\\n) can be inserted anywhere and the definition of <letter> is expanded to include the '.', ',', '/', '_' and '-' characters, as this allows for paths, I/O files and commandline arguments. Furthermore, our parser also checks whether the first <simple command element> in a <simple command> is a <word> instead of a <redirection>. In our implementation, the newlines are filtered out when reading the input, such that the parser does not have to deal with it. Lastly, we assume all tokens are separated by one or more whitespace characters and use recursive descent parsing.

The general process of handling a command goes as follows:

- The shell reads, tokenises and parses the input. If the input is deemed valid, the shell will fork a job handler to interpret and execute the command. It only waits for the job handler if the command is not supposed to be run in the background.
- The job handler interprets the command and forks/executes all parts of the pipeline. Then it will wait until all of its children have terminated, before terminating itself. If any of its children terminate with an unsuccessful exit status, the job handler will terminate prematurely.

When interpreting the input, the job handler loops through all parts of the pipeline in the following manner:

- Check for redirections and handle them one by one.
- Check for commandline arguments and add them to the argv of the to be executed program.
- Check for pipes: if there is a pipe after the command, create a new pipe, link the write end and remember the read end for the next part of the pipeline. Also link the remembered read end from the previous pipe.

To make sure no zombie processes are created, we use both `sigaction` and `prctl` functions in the following manner:

- When initialising, the shell uses `sigaction` to set the `SA NOCLDWAIT` flag on the `SIGCLD` signal, such that it does not necessarily have to call `wait` for its children to be reaped by the closest (sub)reaper program. This is particularily useful when a command is run in the background, as the shell won't wait for the relevant job handler to terminate, without it turning into a zombie process.
- When forking, we use our own `safelyFork` function which uses `prctl` to set the parent death signal of the child to `SIGINT`, such that it will terminate whenever the parent terminates. This is in case a job handler terminates prematurely or either the shell or a job handler get interrupted. (In case the parent already terminated before the child could call `prctl`, it will return -1 to indicate an unsuccessful fork.) While this call is Linux

specific, we could not come up with a nice POSIX compatible way to achieve this.

Side note: most of our functions that return a success/failure status use 0/-1 respectively, similar to POSIX library functions. Only the parser uses standard C booleans, mostly due to it not using any POSIX library functions.

2 Simulating shared memory