Assignment 1: 42sh

Date: April 2nd, 2017
Deadline: April 12th, 2017 23:59 CEST

Objectives

You must implement a Unix shell.

Requirements

Your shell must not be based on (or exploit) another shell to achieve the objective. For example, `system(3)` is forbidden. You must submit your work as a tarball [1]. Next to the source code, your archive must contain a text file file named “`AUTHORS`” containing your name and Student ID. **It is strongly advised you use version control.**

Getting started

1. Unpack the provided source code archive; then run `make`
2. Try out the generated `42sh` and familiarize yourself with its interface. Try entering the following commands at its prompt:

   ```
   echo hello
   sleep 1; echo world
   ls | more
   ```

   To understand the default output, read the file `ast.h`.
3. Familiarize yourself with using `valgrind` to check a program.
4. Read the Wikipedia page on “man pages” [2]. Read it carefully.
5. Read the manual pages for exit(3), chdir(2), fork(2), execvp(3), waitpid(2) (and the other `wait` functions), pipe(2), dup2(2), sigaction(2). You will need (some of) these function to implement the shell.
6. Try and modify the file `shell.c` and see the effects for yourself. This is probably where you should start implementing your shell.

> ***Note***
>
> Your understanding of the Unix manual and its division in sections will be checked during the exam. Really pay attention to this.

1 http://lmgtfy.com/?q=how+to+make+a+tarball
2 http://en.wikipedia.org/wiki/Man_page

Grading

Your grade starts from 0, and the following tests determine your grade (in no particular order):

- +0,5pt if you have submitted an archive in the right format with an `AUTHORS` file.
- +0,5pt if your source code builds without errors and you have modified `shell.c` in any way.
- +1pt if your shell runs simple commands properly.
- +0,5pt if your shell recognizes and properly handles the `exit` built-in command.
- +0,5pt if your shell recognizes and properly handles the `cd` built-in command.
- +1pt if your shell runs sequences of 3 commands or more properly.
- +1pt if your shell runs pipes of 2 simple commands properly.
- -1pt if `valgrind` reports memory errors while running your shell.
- -1pt if `clang -W -Wall` reports warnings when compiling your code.

The following extra features will be tested to obtain higher grades, but only if you have obtained a minimum of 5 points on the list above already:

- +1pt if your shell runs pipes of more than 2 parts consisting of sequences or pipes of simple commands.
- +1pt if your shell supports redirections.
- +0,5pt if your shell supports detached commands.
- +0,5pt if your shell supports executing commands in a sub-shell.
- +0,5pt if your shell supports the `set` and `unset` built-ins for managing environment variables.
- +2pt if your shell supports simple job control: Ctrl+Z to suspend a command group, `bg` to continue a job in the background, and `fg` to recall a job to the foreground.
- +1pt if you parse the PS1 correct and support at least the hostname, username and current path. For more information see PROMPTING in the `sh` manual page
- -1pt if your shell fails to print an error message on the standard output when an API function fails.
- -1pt if your shell stops when the user enters Ctrl+C on the terminal, or if regular commands are not interrupted when the user enters Ctrl+C.
- -1pt if any of your source files contains functions of more than 30 lines of code.
- -1pt if your source files are not neatly indented.

The grade will be maximized at 10, so you do not need to implement all features in this second list to get a top grade.

Note

Your shell will be evaluated largely automatically. This means features only get a positive grade if they work perfectly, and there will be no half grade for "effort".

Example commands

```
## simple commands:
ls
sleep 5   # must not show the prompt too early
```

```
## simple commands, with built-ins:
mkdir t
cd t
/bin/pwd  # must show the new path
exit 42   # terminate with code
```

```
## sequences:
echo hello; echo world # must print in this order
exit 0; echo fail  # must not print "fail"
```

```
## pipes:
ls | grep t
ls | more    # must not show prompt too early
ls | sleep 5 # must not print anything, then wait
sleep 5 | ls # must show listing then wait
ls /usr/lib | grep net | cut -d. -f1 | sort -u
```

```
## redirects:
>dl1 ls /bin; <dl1 wc -l
>dl2 ls /usr/bin; >>dl1 cat dl2 # append
<dl2 wc -l; <dl1 wc -l # show the sum
>dl3 2>&1 find /var/. # errors redirected
```

```
## detached commands:
sleep 5 &  # print prompt early
{ sleep 1; echo hello }& echo world; sleep 3 # invert output
```

```
## sub-shell:
( exit 0 ) # top shell does *not* terminate
cd /tmp; /bin/pwd; ( cd /bin ); /bin/pwd # "/tmp" twice
```

```
## environment variables
set hello=world; env | grep hello # prints "hello=world"
(set top=down); env | grep top # does not print "top=down"

# custom PATH handling
mkdir /tmp/hai; touch /tmp/hai/waa; chmod +x /tmp/hai/waa
set PATH=/tmp/hai; waa # OK
unset PATH; waa # execvp() reports failure
```

Syntax of built-ins

Built-in: `cd <path>`

Change the current directory to become the directory specify in the argument. Your shell does not need to support the syntax “`cd`” without arguments like Bash does.

Built-in: `exit <code>`

Terminate the current shell process using the specified numeric code. Your shell does not need to support the syntax “`exit`” without arguments like Bash does.

Built-in (advanced): `set <var>=<value>`

Set the specified environment variable. Your shell does not need to support the syntax “`set`” without arguments like Bash does.

Built-in (advanced): `unset <var>` **(optional)**

Unset the specified environment variable.

Error handling

Your shell might encounter two types of error:

- when an API function called by the shell fails, for example `execvp(2)` fails to find an executable program. For these errors, your shell must print a useful error message on its standard error (otherwise you can lose 1pt on your grade above 5). You may/should use the helper function `perror(3)` for this purpose.
- when a command launched by the shell exits with a non-zero status code, or a built-in command encounters an error. For these errors, your shell *may* print a useful indicative message, but this will not be tested.

In any case, your program should not “leak” resources like leaving file descriptors open or forget to wait on child processes.

Some tips about the shell

1. It is not necessary that your shell implements advanced features using '*', '?', or '~'.
2. If you do not know how to start, it is best to first start with simple commands, i.e., the node type 'COMMAND'.

```c
if (node->type == NODE_COMMAND)
{
  char *program = node->command.program;
  char **argv = node->command.argv;
  // here comes a good combination of fork and exec
  ...
}
```

3. a shell usually supports redirections on all places of a simple command: 'ls > foo' and '>foo ls' are equivalent. The environment of your shell only supports 'ls > foo'.
4. Within a 'pipe' construction, all parts must be forked, even if they only contain built-in commands. This keeps the implementation easier.

```sh
exit 42 # closes the shell
exit 42 | sleep 1  # exit in sub-shell, main shell remains
```

```
cd /tmp # changes the directory
cd /tmp | sleep 1  # change directory in sub-shell
                   # main shell does not
```

Some tips about the environment

- if `CLANG` is not available, use `make CC=gcc`
- in case of `OSX`: `make LIBS=-lreadline`
- in case of `OSX`: `valgrind --dsymutil=yes`
- your own private git repository at the UvA: https://gitlab-fnwi.uva.nl/