Operating Systems '13
Assignment #1

The shell

February 26, 2013

1 Objectives

1. Understand how a shell works.
2. Learn about basic UNIX concepts of combining applications using *redirection* and *pipes*, in order to solve different tasks.
3. Learn and use system calls.
4. Learn about signals – OS level events dispatched to applications.

2 Shell

For more information on shells, see Shell Computing on Wikipedia.org[1] and Unix Shell on Wikipedia.org[2]. Also, have a look at the POSIX Standard[3].

Essentially, a shell should prompt the user for input. The prompt usually consists of the current directory. After the user inputs the command line, the shell parses it, and executes each part of the command line according to the specification.

3 Assignment

You should write a simple UNIX shell. The shell should read a line of input, process it, and execute the commands. There is some basic functionality you need to implement:

[1] http://en.wikipedia.org/wiki/Shell_%28computing%29

[2] http://en.wikipedia.org/wiki/Unix_shell

[3] http://www.opengroup.org/onlinepubs/009695399/utilities/contents.html

- handling built-in commands (which are to be handled by the shell itself),
- execution of external programs,
- and handling of important shell constructs (such are pipes, redirection, conditional execution, and process control).

You can get the skeleton of the shell from the course site on Moodle, in the file "shell.tar.gz". The archive contains the basic code for parsing, processing, and printing the prompt. Look at the comments to find out where you need to change the code. When you unpack the archive (using command `tar xvzf shell.tar.gz`), you can build the shell using `make`. Your executable will be called `shell`, and you can execute it by typing `./shell` at the command prompt.

When you login on any UNIX-like system, you can type these commands build and run the code:

```
% tar xvzf shell.tar.gz
% cd task1_shell
% make
% ./shell
 Control-C
 ...
% vi shell.c # do your changes
% make
% ./shell
```

In shells, # represents a beginning of a comment, and everything after it until the end of the line should be discarded. `%` represents the command prompt.

File `shell.c` contains the code skeleton. Places where you should put your code are marked in the comments (ie. `/* add your code here */`).

4 Specification

4.1 Internal commands

Shell should support the following internal commands:

cd ARG changes the current working directory to the directory supplied as ARG. If there is no argument, you should print an error message.

status this command should print the status of the last command executed. Commands return 0 as the indication of success, and any non-zero value to indicate the failure.

exit leaves the shell.

4.2 Shell constructs

There are certain constructs your shell needs to support.

redirection - standard input is redirected using `< ARG`. This means that command receives everything written in the file `ARG`, as it would be typed on the keyboard. Standard output is redirected by using `> ARG`. You do not need to care about standard error.

pipes - two (or more) commands can be connected together[4] using the syntax `COMMAND1 | COMMAND2`. This connects the output of `COMMAND1` to the input of `COMMAND2`.

conditional execution - There are two conditional operators, `&&` (AND) and `||` (OR). For the `COMMAND1 && COMMAND2` construct, `COMMAND2` is executed only if `COMMAND1` succeeds. In the `COMMAND1 || COMMAND2` case, `COMMAND2` is executed if `COMMAND1` fails. Command is successfull if its exit status is 0, and fails for any other value.

background processes - Commands can be executed in the background, while the shell is processing the next command. Command is executed in the background using `COMMAND &`.

termination of commands - Commands can be prematurely terminated with `Ctrl-C` combination (pressing `Ctrl` and `c` keys). `Ctrl-C` must not terminate the execution of your shell.

It is not allowed to use the `system()` syscall to call any existing shell, nor to pass the obtained command to any shell via any mechanism.

5 Help and hints

It is possible to implement the functions listed in the specifications, in the order that they are listed. Implement one of them at a time, test it and when it seems that it behaves as specified, then move on to the next one.

To be able to carry out the programming assignment you must study the manual pages (`man` command under any UNIX-like operating system) for various system calls. For some of them, e.g. `exec`, there are several possibilities. You need to figure out which one suits better; sometimes there are more than one that are suitable.

Some system calls that you will definitely need are: `fork`, `exec`, `wait`, `stat`, `signal`, `pipe`, `dup2`, and `waitpid`.

Also, for handling `Ctrl-C` you should read about catching signals, specifically signal `SIGINT`.

[4] like with a pipe

Finally, do not hesitate to use any web-search to find the information you are looking for.

5.1 On some system calls

System calls are the interface between user-space applications and the Operating system. When the application needs some service from the OS, it sets parameters of the service, and calls corresponding system call (syscall). Upon return from the system call, application will have the response and status (whether syscall succeeded).

For example, if application needs to get current time, it will call syscall `time`, and pass the address of the variable which should contain time. After calling the syscall, control is passed to the OS, which will read its internal counter (time in seconds since Jan 1, 1970), and place it in the given address. Then, OS will return control to the application. After syscall finishes, application will find the time at the address it passed to the syscall.

Quick reference list of system calls in Linux can be found here [5]. This list is not complete, although it contains all necessary system calls.

UNIX-like operating systems revolve around files. For ordinary user, file is represented by a name. When an application opens a file by name, it gets a handle on that file, which is called *file descriptor*. From that point on, whenever application needs to perform an operation on a file, it passes the file descriptor to the OS as a parameter to operation. So, every open file in your program gets assigned one file descriptor. If the same file is opened from many application, it will probably have different file descriptor in each application (this means that file descriptors are local to the application).

Since UNIX considers the standard input and the standard output as files, they get their file descriptors, too. According to the POSIX standard, file descriptors are represented as integers, and STDIN (Standard Input) has file descriptor 0, while STDOUT (Standard Output) has 1. Another standard file descriptor is 2, for STDERR (Standard Error). STDERR is usually used for reporting errors from the program.

File descriptors for other files are instantiated after the `open` system call. This system call accepts the filename, and return the file descriptor which you can later use to access the file.

dup2 `dup2(int fildes, int fildes2)` system call duplicates file descriptor `fildes` and assigns it the value in `fildes2`. After calling `dup2`, you can access the file with either `fildes` or `fildes2`. This system call is useful when you want to redirect the standard input or standard output.

[5] http://www.digilife.be/quickreferences/QRC/LINUX%20System%20Call%20Quick%20Reference.pdf

Normally, standard output (and standard error) is set to the screen. For example, suppose that you want to redirect the standard output to some file (so that you can have permanent storage of output). If you have opened that file (via `open` syscall), and obtained a file descriptor 42, you would write `dup2(42, 1)`. For your convenience, header file `unistd.h` provides constants `STDIN_FILENO` and `STDOUT_FILENO`, which hold the file descriptors of standard input, and the standard output.

signal POSIX standard describes signals as an asynchronous notification system for inter-process communication. Signals enable processes to notify each others, or to receive notifications from the OS. They are a quite limited form of communication, since there is a limited number of signals which can be delivered. Upon receiving a signal, process calls a special method called the *signal handler*.

`signal` system call accepts the signal number and a pointer to the signal handler function. Then, on receiving the signal your signal handler gets called. For intercepting `Ctrl-C` you will need to override the signal handler for `SIGINT` signal.

5.2 How to test (evaluation)

Here are some simple examples you can use to test your code:

```
% sleep 5 &
% sleep 10
% ls > apa
% wc -w < apa
% ls /usr/bin | sort -r | sort | sort -r
% yes > /dev/null
^C (control-C)
% cat < apa | wc | wc > bepa
% cat bepa
% cd /tmp && ls -Fal
% cd /tmp/unknown || mkdir /tmp/unknown
```

Additionally, you will find `test/runtests.sh` script in the archive. Pass the path to you shell as an argument to this script, and it will test it. Additionally, in the file `test/testscript` you can find commands which will be passed to your shell. We will use the same script to grade your assignment.

6 Deliverables

You need to submit an archive (in `tar.gz` format) which contains source files, and the Makefile for your assignment. The deadline for this assign-

ment is March 19, 2013 23:59 CET. This is a final, hard deadline. No extensions will be given.

7 Requirements and grading

The archive needs to contain correct Makefile and source code. The Makefile should produce an executable named "shell" (provided Makefile does exactly that).

We will execute `make` in the directory containing the extracted files. Then, we will execute the tests on produced executable. If `make` fails, or we need to modify your code to get it to compile, you will get *0 points*. Only modification we will perform (without incurring any penalty on you) is the addition of necessary include files, if necessary.

It is not allowed to use any system call to execute another shell in place of yours, or to pass commands to another shell.

Further, it is not allowed to pre-record outputs of given test file, and replay it back during our testing.

Percentage of successful tests is the grade you will get on this assignment.