Middle East Technical University Department of Computer Engineering

CENG 334

Introduction to Operating Systems

Spring 2016-2017

Homework 1 - Unix Shell

Due date: 31 03 2016, Friday, 23:59

1 Objective

The objective of this assignment is to learn the basics of process management, I/O and signaling. Your task is to implement a basic Shell. A shell is a command line interpreter which continually reads the standard input for the commands to be received from the user, executes corresponding programs and displays their outputs until the shell is terminated by the user. For this task, you must be able to create new processes from your main process and execute the commands, redirect their I/O and handle signals associated with them as necessary.

2 Unix Shell

A command line for a shell is the text entered by the user after the prompt is shown. It may contain exactly a single command or a sequence of commands. A command is the name defined within the environment[1] of the shell or the full path of an executable followed by optional arguments. The executable name and arguments are delimited by space character. Therefore executable names and arguments are space-free strings.
Shell has the ability to redirect standard output and input of each command. If you enter `echo 12 > test.txt`, the output of the echo operation is written to the file `test.txt` because of the redirection. Then, when you enter `increment < test.txt` it will print 13 by reading the input from the `test.txt` file.

```
> echo 12 > test.txt
> increment < test.txt
13
```

[1]Every UNIX process runs in a specific environment. An environment consists of a table of environment variables, each with an assigned value. When you log in certain login files are executed. They initialize the table holding the environment variables for the process. When this file passes the process to the shell, the table becomes accessible to the shell. For example, the PATH variable is used to determine the full path of the executable commands like ls, cat, and echo.

It is also possible to execute sequential commands on a standard shell using deliminators between each command. In this homework, we will focus on `pipe(|)` and `semicolon(;)` deliminators.
In the pipe case, standard output of each command is connected to the standard input of the command coming after it. For example the following command should print 13 where output of the command echo is transferred to the input of increment using the connection established before. Assume increment is a simple incrementation process in this case.

```
> echo 12 | increment
13
```

In the semicolon case, there is no connection between processes. The commands are executed from left to right. For example the following command should print 12 first, sleep for 2 seconds, and then it should print 13.

```
> echo 12; sleep 2; echo 13
12
13
```

Note that when sequential commands are executed using semicolon, I/O redirection and piping is possible for each command.
Background processing is the ability to execute multiple commands in the background without waiting for their completion and immediately displaying prompt after each background command. Background execution is carried out by adding an `ampersand(&)` to the end of the command.
Subshell commands are commands that are enclosed in parenthesis (i.e '(' and ')'). These commands are executed by creating a shell process as a child of the current one. The subshell process then can be used for I/O redirection, piping or background execution. For example:

```
> (echo hello; echo world;) > helloworld.txt
> cat helloworld.txt
hello
world
```

`CTRL+c` combination is used in a shell to send an INT signal to the current foreground job, as well as any descendants of that job (e.g., any child processes that it forked directly). If there is no foreground job, then the signal has no effect. The default behaviour of the INT signal is to terminate the process it is sent to.

3 Required Capabilities

3.1 Prompt

Once started and after the completion of each command or immediately after each background command, the shell should print "`> `". After printing the prompt, it is ESSENTIAL to `fflush` standard output for your program to be graded correctly.
For flushing STDOUT, the following or equivalent code must be used:

```
printf("> ");
fflush(stdout);
```

3.2 Quit

The shell must terminate with the quit command.

```
> quit
```

The only exception is when a previous command is still running in the background. In this case, your program must stop responding to new commands, wait for all the background processes to finish, print associated information, and then terminate immediately. See below background execution section for further details.

3.3 Single Commands - 10 pts

When the shell reads a single command with or without arguments, it must execute it, block until the command is completed, and then return back to the prompt. The shell should create a child process, and convert it to the program denoted by the command. Example:

```
> ls -l
```

In the above example, your program should create a child process which turns itself to an `ls` process. Your program should then wait for the ls process to complete, and then ask for a new command by displaying the prompt.
Related UNIX system calls: `fork`, `exec` family, `wait` and/or `waitpid`

3.4 Input redirection - 10 pts

When a command whose input is redirected as "`< inputfile`" is entered, the program must redirect its child process' standard input to `inputfile` before executing the given command.
If there is no `inputfile`, you should print `inputfile not found`.
Example:

```
> cat < file.txt
Content of file.txt
> cat < nofile.txt
nofile.txt not found
```

Related UNIX system calls: `dup2`. Also see file descriptors of a process and stdin.

3.5 Output Redirection 10 pts

When a command whose output is redirected as "`> outputfile`", the program must redirect the child process' standard output to `outputfile`.
Example:

```
> cat file1 > file2
```

Related UNIX system calls: `dup2`. Also see file descriptors of a process and stdout.

3.6 Background execution - 17.5 pts

A command should be run at background when it ends with an '`&`'. The shell should not block for the completion of the command, and display prompt immediately. Then the user may continue interacting with the program and give new commands. However, your program must be aware of when the background process terminates, and report its exit status to the user together with its process id, as:

`[pid]␣retval:␣<exitcode>`

Assume that '`&`' will always be the last character in the input.
Example:

```
> sleep 5 &
> cat file.txt
Content of file.txt
> [24617] retval: 0
```

Here, the second prompt is printed immediately by the shell, whereas the pid and return value are printed after the termination of the background process.
Note that, if the shell gets a quit command while there is an alive process currently running at the background, then it must stop displaying prompts and accepting user input, and wait until all the background processes terminate. It must then print the related information and terminate immediately.
Related UNIX system calls: `sigaction`, `WEXITSTATUS`. Also see signal handling and interrupted system calls.

3.7 Sequential Commands - 30 pts

3.7.1 Pipe - 20 pts

When multiple commands are connected to each other via pipes, the output of i^{th} command is fed to the input of the $(i+1)^{th}$ command. The output of the last command is connected to standard output unless it is redirected to file. The input of the first command can also be redirected from a file other than standard input if the process requires input.
Example:

```
> find /etc | grep ssh | grep conf
```

Note that in this task, the shell should wait for all of the commands terminate before displaying the prompt.
Related UNIX system calls: `pipe`, `dup2`. Also take a look at file descriptors of a process, stdin and stdout.

3.7.2 Semicolon - 10 pts

When multiple commands are separated with a semicolon, the $(i+1)^{th}$ command is executed after the completion of the i^{th} command. For example:

```
> echo started ; sleep 1 ; echo finished.
started
(sleeps for 1 second)
finished.
```

I/O of each command can be redirected to a file as described in the sections 3.4, and 3.5. Examples:

```
> echo message > message.txt ; sleep 1 ; cat < message.txt
message
```

For convenience, assume that sequential commands will not be used simultaneously with background execution or pipes.

3.8 Subshell Commands - 20 pts

Subshell commands are defined as single, or sequential commands executed in an exact copy of your shell as the child of the original shell. Its syntax is any type of command defined above between two parenthesis ('(' and ')'). The only difference from the main shell should be in terms of continuous execution. The subshells should immediately quit after the completion of its given commands. Example:

```
> ( echo hello )
hello
> ( echo hello ; sleep 2 ; echo world )
hello
(sleeps for 2 seconds)
world
```

I/O of each subshell can be redirected to a file or piped to another process/subshell as described in the sections 3.4, 3.5 and 3.7.1.
For example:

```
> ( echo HELLO ; sleep 2 ; echo WORLD ) > out.txt
> cat out.txt
HELLO
WORLD
> ( cat | tr /A-Z/ /a-z/ ) < out.txt
hello
world
> ( echo HELLO ; sleep 2 ; echo WORLD ) | ( cat | tr /A-Z/ /a-z/ )
hello
world
```

Note that in two different subshells pipe and semicolon can exist, however they will not be used simultaneously in a single subshell.
Subshell can also be run on background.

```
> ( echo "TIR" > test.txt ; cat < test.txt ; sleep 5 ; echo ali ) &
> TIR
(sleeps for 5 seconds)
ali
[28201] retval:0
```

For convenience, assume that subshells will not be separated with semicolon.
Related UNIX system calls: `pipe`, `dup2`, `fork`, `exec` family, `wait` and/or `waitpid`

3.9 Handling INT Signal - 2.5 pts

When you run your shell within the standard Unix shell, your shell is running in the foreground process group. If your program then creates a child process, by default, that child will also be a member of the foreground process group. Since typing `CTRL+c` sends a `SIGINT` to every process in the foreground group, it will send a `SIGINT` to your program, as well as to every background process that your program created, which obviously isn't correct.
Here is the workaround for background jobs: After the `fork`, but before the `exec` call, the child process should change its process group to a new one whose group ID is identical to its own PID. This ensures that any of the background jobs will not be in the same process group with the main program and will not receive INT signal.
Related UNIX system calls: `kill`, `sigaction`, `setpgid`

3.10 List Background Processes

The program should list the running background processes when it gets command `lbp` in the following format:

```
[process_id1]
[process_id2]
...
[process_idn]
```

Example:

```
> sleep 10 &
> sleep 15 &
> lbp
[2710]
[2734]
> [2710] retval: 0
[2734] retval: 0
```

Note that in this case, even though there may be multiple commands in a subshell, it should be listed as a single process since the commands that being executed in the subshell are not children of the main shell and therefore should appear as a single process.

4 Limits

- Maximum command line length = 512 bytes
- Maximum number of args (including path of the executable) = 20
- Maximum number of commands running parallel (sequential & background) = 40
- Maximum length of an argument or executable name = 128 bytes

5 Specifications

- The codes must be in C. No C++ codes are accepted.
- Your programs will be compiled with gcc and run on the department inek machines. No other platforms and/or gcc versions etc. will be accepted, Therefore make sure that your code works on inek machines before submitting it.
- Commands may be executed using name or full path. Both are valid.
- Commands to be used for evaluation will be syntactically true. No need for checking their validity.
- Non-existing commands will not be used for evaluation.
- In case of a background execution, the process running at the background may print something after the prompt is printed by your program. In this case the program will not print another prompt. So the user might write the next command on a blank line with no prompt.
- Beware of timing. After printing something, `fflush()` to stdout to ensure that output is not buffered.
- Please take care that the places of input and output redirections in the command can change if you are not using the parser supplied to you. `(cat > output < input` is valid as well as `cat < input > output`.)
- Your program must not leave any zombie processes(see `wait` & `waitpid`). If you leave zombie processes in a testcase, you will lose half the points for that testcase.
- The quit functionality will not be graded. However, it is essential for your homework to be graded. Please also make sure that your subshells terminate after executing its commands.
- The use of "`system()`" function (to implement the execution of the commands) is strictly forbidden in this homework. You are explicitly required to use `fork` and `exec` family. Any program using `system()` will receive 0.
- Using any piece of code that is not your own is strictly forbidden and constitutes as cheating. This includes friends, previous homeworks, or the internet. The violators will be punished according to the department regulations.
- Follow the course page on Piazza for any updates and clarifications. Please ask your questions on Piazza instead of e-mailing if the question does not contain code or solution.

6 Submission

Submission will be done via COW. Create a tar.gz file named `hw1.tar.gz` that contains all your source code files together with your Makefile. Your tar file should not contain any folders. Your code should be able to compile and your executable should run using this command sequence.

```
> tar -xf hw1.tar.gz
> make all
> make run
```

The name of your executable is not important as long as it executes with "make run" command.
If there is a mistake in any of the 3 steps mentioned above, you will lose 10 points.