Exercise 1, Write a Linux shell

E. Markatos

1. Description

In an operating system, a shell is the primary user interface that allows you to interact with the computer by providing a command-line or graphical way to execute commands, manage files, and control programs. It acts as an intermediary between the user and the operating system's kernel, interpreting your input into instructions the kernel can understand and execute. Common examples include Bash on Linux and macOS, and the Windows Command Prompt.

In this exercise you are going to implement a Linux shell written in the C programming language. More specifically your task is to create a command line interface shell able to create processes, establish inter-process communication through pipes, write to and read from files using redirection and, finally, control the flow of the commands using the conditional 'if' statement and 'for' loops.

2. Implementation

2.1. Prompt

Your shell will have the name 'hy345sh' and it will show a prompt in the format of

'**<username>@-<RN>-hy345sh:<directory> $**' where,

- <*username*>: is the username of the currently logged in user.
- <*RN*>: is your Registration Number (e.g. 'csd9999').
- <*directory*>: is the current working directory (e.g. '/spare/csd9999')

2.2. Simple commands

Your shell must be able to read and execute simple commands passed from the user. For each command you should create a new child process which runs the command. Each command may be passed in separate line:

```
ls
cd /tmp
echo "Test"
```

or in the same line, separated by ';':

```
ls; cd /tmp; echo "Test"
```

Regardless of the syntax both of these should yield the same result. You can run different commands such as 'ls', 'date', 'cat', 'mkdir' and more to test that your shell works as expected.

2.3. Global variables

Your shell must support global variables where the user can write to/read from. For example the following command:

```
x=world
echo hello $x
```

should declare a global variable named ‘x’ with the literate value "world". Later this variable is used to echo "hello world" to the screen.

2.4. Pipelines

Pipes denoted by the ‘|’ symbol allow us to redirect the standard output of one process to the standard input of another process. Your shell must support pipelines between two commands:

```
echo "hello world" | grep "world"
```

or multiple commands:

```
cat grades.csv | tail -n+2 | tr ',' ' ' | sort -rn -k 2
```

2.5. Redirection

Each process has three standard streams where it can write to and read from. These streams are:

1. *stdin*: standard input, where the process typically reads input from
2. *stdout*: standard output, where the process typically writes output to
3. *stderr*: standard error, where the process typically writes error output to

By default these streams write to and read from the terminal, however we can redirect them to point to specific files: Redirection is denoted by the ‘<’ symbol for stdin and the ‘>’ symbol for stdout. Your shell must support *stdin* and *stdout* redirection.

2.5.1. Stdin Redirection

Using the ‘<’ symbol, the input must be redirected from the file following the ‘<‘ symbol:

```
./myprogram < test.in
```

2.5.2. Stdout Redirection

Using the ‘>’ symbol, the output must be redirected to the file following the ‘>’ symbol:

```
./some_texty_process > /dev/null
```

Additionally your shell must also support stdout redirection that *appends to the end of file* using the ‘>>’ symbol:

```
./program >> /var/log/program.log
```

2.6. Program control flow and loops

Your shell must support control flow and loops. Specifically your shell must parse and interpret:

1. The conditional **if** statement

```
if command
then
    commands...
fi
```

2. The **for** loop

```
for variable in tokens...
do
    commands...
done
```

where tokens may be strings (e.g. "hello world" "foo" "bar"), plain tokens (e.g. one two three), variables (e.g. $x $var) or mixture of these. In every loop the respective element is assigned to the variable where it can be used inside the body.

3. Execution

Sometimes a process (either your shell or a child process) may crash the system by e.g. spawning too many processes. In order to protect you and other users from these sort of issues you may test your shell inside a virtual machine or an emulator. As such, a crash will, in the worst case, crash the virtual machine which can easily be fixed or restarted.

3.1. Virtual Disk

In this exercise you will have to setup and test your code inside a virtual machine using QEMU. QEMU is the emulator you will be using to host the VM and test your shell QEMU is installed inside every school remote system. QEMU can read a virtual disk containing any OS. In our case this is going to be Linux. There is an already existing image inside '~hy345/qemu-linux/hy345-development.img' which you may copy and use.

Since copying the virtual disk directly into your home folder will most likely exceed your quota you will have to create a directory under '/spare'. The '/spare' directory is a directory which is **not** shared between different the machines (damaskino, milo etc.), thus you will have to use the same machine for testing.

Create a directory under '/spare' (use your RN as a name in order to not conflict with others)

```
MYSPARE=/spare/csd9999
mkdir ${MYSPARE}
chmod 700 ${MYSPARE}
cp ~hy345/qemu-linux/hy345-development.img ${MYSPARE}
```

3.2. Using QEMU

When you copy the virtual disk you may run QEMU as follows

```
qemu-system-i386 -hda ${MYSPARE}/hy345-development.img -display curses
```

where the '-hda' option takes a path of the virtual disk we are using and the '-display curses' option indicates that the VM should use the terminal's display and not open a separate window. After running this command you will be prompted to enter a username and a password, both of these are "hy345"

3.3. File transfer

In order to transfer files from and to the guest OS which is running inside QEMU you may use scp with the virtual ip address '10.0.2.2'.

Transfering a file named "file.txt" from the host OS to the guest OS can be done using the following command inside the guest OS (note the '.' at the end, which is destination of the copy):

```
scp csd9999@10.0.2.2:~/file.txt .
```

the opposite can be done be as such:

```
scp file.txt csd9999@10.0.2.2:~/
```

3.4. Saving files

If the system inside QEMU crashes you may risk losing data stored inside it. In order to prevent up you're given the 'save' command which is suggested to run before testing your shell. This lowers the risk of data loss in the case of a crash. For best measure it is highly suggested to keep the original files in the host.

3.5. Exiting QEMU

In order to leave QEMU and guarantee a normal exit you may run:

```
turnoff
```

3.6. Remote access

3.6.1. SSH

In order to connect remotely you may use the ssh command available in both Windlows, WSL and Linux. More details can be found in the CSD website.

3.6.2. Windows

If you're on Windows you can alternatively use WinSCP for file transfer and putty for remote access.

4. Examples

To make sure that your program works correctly you may use simple as well as more complex commands that contain both pipelines and redirections and check against the output of the native Linux shell (which is bash in the most systems). Here is a list of commands that you may test your program with.

```
# Stdout redirection
ls -al>y1
# Commands separated by ';'
cat y1; rm y1
# Variable
age=20; echo I am $age years old
# Multiple pipes
ls|sort|uniq|wc
# Stdout redirection (both overwriting and appending), input
# redirection, long pipelines
echo hello > yy; echo world >> yy; cat < yy | sort | uniq | wc > y1; cat y1
# Multiple variables
func="5+5="; res=10;echo $func$res
# Redirect variable into a file
my_var="This is my var"; echo $my_var | wc > xx; cat<xx
x="5|5;10"; echo $x>file; cat<file
# 'if' statement in a single line
x=5; if [ $x == 5 ]; then echo "hello world"; fi
# 'if' statement broken down into multiple lines
x=5
if [ $x == 5 ] # the spaces inside the brackets are needed
then
    echo $x
fi
# 'for' loop in a single line
for i in 1 2 3; do echo $i; done
```

```
# 'for' loop broken down into multiple lines
for c in "h" "e" "l" "l" "o"
do
    echo $i
done
```

Notes:

- These are **just examples** and your code will may tested against other combinations of commands as well.
- Some symbols such as ';', '|', '>' *may* have spaces in between, in these cases the spaces are ignored.
- The **if** checks the output of a command in order to determine which clause it should run. The '[' symbol in the above cases actually refers to test(1) which is symlinked to '['.

5. Submission instructions

Submissions for the exercise will be done over at CS345 E-Learn page. You will submit a **ZIP** file containing:

- All your relevant source files.
- A 'Makefile' which builds your project by running:

```
make all
```

and cleans it up by running:

```
make clean
```

- A 'README' file in which you explain your work and any notes that you deem important to include.

Do **NOT** include the '.hda' image in your ZIP file

6. Useful Sources

Process management:

- fork(2)
- exec(3)

Environment management:

- setenv(3)
- getenv(3)

Interprocess communication:

- pipe(2)
- dup2(2)

Input reading:

- readline(3)

Parsing:

- lex(1p)
- yacc(1p)

7. Assignment Instructions and Regulations

- This exercise is strictly individual. The submitted files will be checked by a plagiarism detection system. Any instances of plagiarism will result in automatic disqualification.
- At the beginning of your code, include your full name and student ID in comments.
- Keep your code tidy and add comments explaining its functionality where necessary.
- The use of code generated by AI is strictly prohibited.