CS2850 Operating System Lab

Week 5: the UNIX shell

nicolo colombo

nicolo.colombo@rhul.ac.uk

Office Bedford 2-21

Outline

How the shell works.

The UNIX shell

Wild cards

Bash programming

How the shell works

A *shell* is a user-level process for interacting with your machine.

The process *starts* when you open the terminal, e.g. when you log in using `puTTY`.

The shell prints a prompt and waits for you to *type a command* in the terminal and *press* `enter`.

The shell executes the insturction and prints a new prompt when it is ready to *accept a new command*.

Command execution

When you run a command the shell creates a new child process.

The child process executes the corresponding program, e.g. `ls` is a program name. [1]

The shell waits for the child process to terminate.

After the shell has reaped the child process, it prints a prompt again and waits for the next input.

[1] The shell assumes the first word of the input is an executable file, like `./a.out`.

Example

The user moves to `Desktop/operatingSystems/`, looks at the directory content, and prints their location.

```
cd Desktop/operatingSystems/
ls
CS2850Labs  intros
pwd
/home/cim/staff/ugqm002/Desktop/operatingSystems/
ls -l CS2850Labs/
total 72
-rwx------ 1 ugqm002 staff 16832 Sep 21 17:56 a.out
drwx------ 2 ugqm002 staff  8192 Oct  4 08:45 example
drwx------ 2 ugqm002 staff  8192 Sep 23 13:01 week1
drwx------ 3 ugqm002 staff  8192 Sep 30 17:23 week2
drwx------ 2 ugqm002 staff  8192 Oct  7 18:08 week3
```

`fork` and `execve`

Child processes are created with `fork` and used to invoke a loader (`execve`) that

1. replaces the child's virtual memory with the info needed *to execute the command*, e.g. the code of the command entered by the user,
2. runs the specified program (within the child process), and
3. terminate the child process.

Example

This program works like a shell and prints `hello, world!` on the screen.

```
#include <unistd.h>
#include <wait.h>
int main() {
  char *argv[2];
  argv[0] = "a.out";
  argv[1] = NULL;
  if (fork() == 0) execv(argv[0], argv);
  wait(NULL);
}
```

`a.out` is the executable of the following code

```
#include <stdio.h>
int main() {
  printf("hello, world!\n");
}
```

The UNIX shell

The first Unix shell was developed for Unix by Steven R. Bourne in 1974.

The *Bourne shell* is now the standard shell in UNIX systems.

Each process created by a Unix shell has three open files:

1. standard input (`fd` = `STDIN_FILENO` = 0),
2. standard output (`fd` = `STDOUT_FILENO` = 1), and
3. standard error (`fd` = `STDERR_FILENO` = 2).

Extra features

IO redirection operators, < and >.

Example: the following lines *write the output* of `ls -l` on `outupt.txt` and print on the screen the first lines of `output.txt`.

```
ls > output.txt
head < output.txt
```

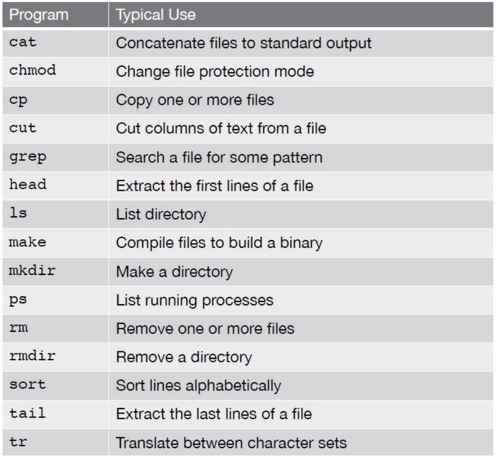
A pipe command, |, to send a command output to another command.[2]

Example: the following line prints only the files and directories modified on October 25th.

```
more output.txt | grep "Oct 25"
```

[2]Without generating extra files.

Shell commands



Program	Typical Use
`cat`	Concatenate files to standard output
`chmod`	Change file protection mode
`cp`	Copy one or more files
`cut`	Cut columns of text from a file
`grep`	Search a file for some pattern
`head`	Extract the first lines of a file
`ls`	List directory
`make`	Compile files to build a binary
`mkdir`	Make a directory
`ps`	List running processes
`rm`	Remove one or more files
`rmdir`	Remove a directory
`sort`	Sort lines alphabetically
`tail`	Extract the last lines of a file
`tr`	Translate between character sets

ls-l

The full information about the files in a directory is obtained by running

```
ls -l directoryPath
```

and includes the *location*, *type*, *size*, the owner and owner's group, their *permissions*, and the last time the file was *modified*, e.g. [3]

```
ls -l
total 96
-rwx------ 1 ugqm002 staff 16840 Oct 20 11:08 a.out
-rw------- 1 ugqm002 staff    63 Oct 20 12:11 helloWorld.c
-rw------- 1 ugqm002 staff   496 Oct 20 12:27 output.txt
drwx------ 2 ugqm002 staff  8192 Sep 10 17:18 w1
```

[3]Directories have a `'d'` as first character.

Permissions

The permissions of files are encoded in 10-character strings, specifying the permissions of the *user*, the user *group*, and the *rest of the world* to *read*, *write*, and *execute* the file (in the order).

The 10 characters below show that the user can read, write, and execute, the user group can read and execute, and the rest of the world can only execute. e.g.

```
-rwxr-x--x
```

chmod

You can set or change the permission bits with

```
chmod [who] operator [permissions] file_name
```

where `who` $\in \{\mathtt{u}, \mathtt{g}, \mathtt{o}, \mathtt{a}\}$, `operator` $\in \{+, -, =\}$, and `permission` $\in \{\mathtt{r}, \mathtt{w}, \mathtt{x}\}$.

Example: If you want to make a file executable to yourself you need

```
chmod u+x file_name
```

Check Section 1.4 of Linux and Shell Programming for using `chmod` in absolute mode, where the operation string with an octal number, e.g. 0100 means *“the owner can execute”*.

Shell variables

The shell uses environment variables to set a working environment when you log in.

You can inspect *all* environment variables by running `env`, e.g.

```
env
SHELL=/bin/bash
LANGUAGE=en_GB:en
PWD=/home/cim/staff/ugqm002/Desktop/operatingSystems
    /2021/labs/week5
...
```

You can print a list of selected variables, `VARNAME1`, `VARNAME2`, . . . , by running

```
echo $VARNAME1 $VARNAME2 ...
```

Setting variable values

You can *set* the value of `VARNAME` to a specific `value` using

```
VARNAME=value
```

This changes the variable only for the current process[4]

You can clear a variable with

```
unset VARNAME
```

[4]Use `export VARNAME` to affects its subprocesses (but not its parent process).

Example

Try the following commands to understand how this works

```
ONE=1; export ONE; TWO=2
./changeVariables.sh
1

1
3
echo $ONE
1
echo $TWO
2
```

`changeVariables.sh` is a file containing the following lines

```
#!/bin/bash
echo $ONE; echo $TWO
ONE=1; TWO=3; export TWO
echo $ONE; echo $TWO
```

Wild cards

The shell has a set of pattern-matching meta-characters to match strings based on patterns:

- `*` matches any string including a null string, e.g. `ls *.c`,
- `?` matches any one character, e.g. `ls ??Script.sh`,
- `[...]` matches any characters enclosed in the square brackets, e.g `ls *[S]*`, and
- `[!...]` matches any characters other than the characters following the exclamation mark, e.g. `ls *.[!s]*`

The resulting general expressions can be used for running shell commands.[5]

[5]The shell expands all wild cards before executing the commands.

Example

The following bash lines combine shell commands and regular expressions

```
ls -l *.* > output.txt
grep '\.c' output.txt
```

`ls -l *.* > output.txt` writes the info of all items with name matching `item_name.ext_name` on `output.txt`

`grep '\.c' output.txt` picks the lines corresponding to all `.c` files.

See more about `grep` (global regular expression print) in Chapter 8 of Linux and Shell Programming.

Shell scripts

You can compose a list of commands for the shell to execute.

A shell script is a file holding *a program built with shell commands*.

Scripts can use some of the *usual constructs* to make simple decisions and loops, i.e. the shell is a programming language.

Example

```
#!/bin/bash
i=start; echo $i > output.txt
for i in 1 2 3 4 5 6 7 8
do
  echo $i "hello world" >> output.txt
done
more output.txt
```

The script above prints the following lines.

```
./myScript.sh
start
1 hello world
2 hello world
...
7 hello world
8 hello world
```

Script variables

Local variables can be declared, modified, and printed on the terminal, e.g.

```
VARNAME=value
$VARNAME=newValue
echo "VARNAME=" $VARNAME
```

Special variables can be used to process the program input, e.g.

$#: number of parameters passed to the script,

$0: script name,

$1: first parameter,

$2: second parameter

...

$?: exit value of last command.

Tests

Let `VAR` be a *string* and `N` an *integer*, e.g. `VAR=one` and `N=1`.

You can make tests through

```
if [ $VAR = test_string ]
if [ $N -eq test_value ]
```

Note: all *empty spaces* are strictly required.

You can modify numerical variables using `expr`, e.g.

```
N=`expr $N + $N + 1`
```

Loops

You can use `for`-loops and `while`-loops as in many other languages, e.g.

```
#!/bin/bash
VAR=0
echo "VAR=" $VAR
for i in 1 2 3; do
        VAR=`expr $VAR + $i`
done
echo "VAR=" $VAR
while [ $VAR -ge 0 ]; do
        VAR=`expr $VAR - 1`
done
echo "VAR=" $VAR
```

Example

Run this script with 2 inputs, `string` $\in$ {`plus`, `minus`} and `value` $\in \mathbb{N}$

```
#!/bin/bash
VAR=$1; N=$2
echo "VAR=" $VAR ", N=" $N
if [ $VAR  =  plus ]; then
  N=`expr $N + 1`; VAR=good
else
  if [ $N -ge 0 ]; then
    N=`expr -1`
  fi
  N=`expr $N - 1`; VAR=...
fi
echo "VAR=" $VAR ", N=" $N
if [ $N -lt 0 ]; then
  echo "wait..."
  while [ $N -le 0 ]; do
    N=`expr $N + 1`; echo "N=" $N
  done
  VAR=good
fi
echo "VAR=" $VAR ", N=" $N
```