UNIX AND LINIX IN INFOCOMMUNICATION
Week 7

O. Sadov

Shell programming

Now it's time to start programming, Shell programming. As far as we understand, the shell works like a normal program and has several options that can be useful for debugging:

`-v`	Print shell input lines as they are read.
`-x`	Print commands and their arguments as they are executed.
`-c STRING`	Read and execute commands from STRING after processin

If your shell is Bash, you may get some help:

```
bash -c help
bash -c 'help set'
```

Let's follow the good tradition started by the classic book of Kernighan and Ritchie "The C Programming Language" and write a standard program "Hello World":

```
$ cat > hello
echo Hello word!
^D
```

We complete the input with the EOF Ctrl-D character. Let's try to run now:

```
$ sh hello
Hello word!
```

Good.

And now let's turn our script into a real executable program:

```
$ chmod +x hello
$ ./hello
Hello word!
```

Excellent! Now we will talk about the arguments. Let's look at our first positional parameter:

```
$ vi hello
echo Hello $1!
$ ./hello
Hello !
```

We called our script without parameters and got nothing. Let's add some parameter:

```
$ ./hello world
Hello world!
```

But for several parameters it does not work:

```
$ ./hello world and universe
Hello world!
```

Let's fix it:

```
$ vi hello
echo Hello $*!
$ ./hello world and universe
Hello world and universe!
```

Excellent! As we can see, there is some difference between different shells in the use of some special variable names associated with script parameters (urlhttp://sdn.ifmo.ru/education/courses/free-libre-and-open-source-software/lectures/lecture-9).

The main advantage of the latest shells over the classic Bourne Shell shells is the ability to use more than 9 parameters. Because in the Bourne Shell we only have one digit for the parameter number. In newer shells, we can use longer numbers in square brackets.

A very important thing in the UNIX world is the zero-numbered positional parameter. Let's try to look at this:

```
$ vi hello
echo \$0: $0
echo Hello $*!
$ ./hello
$0: ./hello
Hello !
```

It's just the name of the script. And if you are familiar with the C language, you probably know — the same is in argv[0]. For what needs can such a parameter be used? The most obvious answer seems to be to write a nice 'Usage' error message:

```
$ vi hello
if [ $# -lt 1 ]
then
       echo Usage: $0 who...
       exit
fi
echo Hello $*!
```

But on UNIX-like systems we can use a very interesting trick — linking files. Take a look at this super-nano-notebook. It runs on OpenWRT, a Linux distribution that you can find on your home internet router, for example. We have a fully functional set of Linux utilities, but if we take a closer look, we can see that all common UNIX utilities are just symbolic links to a single "busybox" binary.

```
# ls -l /bin
```

Busybox just looks at the name it is running under and performs the appropriate functions from the busybox library. This technique can reduce the memory consumption of the embedded system. Let's try to use this feature on our script:

```
$ vi hello
if [ $# -lt 1 ]
then
     echo Usage: $0 who...
     exit
fi
echo $0 $*!
$ ln -s hello bye
$ /hello world
./hello world!
$ ./bye bye
./bye bye!
```

Shell Controls

Basic logical operators

So we have scripts with arguments, but what about the logic? We have some operations that look like logical ones, but at first glance they look a little strange. In fact, the cornerstone of Shell's logic is this strange reserved variable:

```
$? - exit value of the last run command
```

For example, we have the following commands:

```
$ true; echo $?
0
$ false; echo $?
1
```

What does this mean? Only one thing — we have a successful program with an exit code of 0 and a failure with a non-zero exit code. It seems strange, but it is understandable — in fact, UNIX follows Leo Tolstoy's principle: "All happy families are alike; each unhappy family is unhappy in its own way." That's right — we were not interested in the details of when our program finished successfully, but the reason for the failure can be different and we should be able to separate one from the other.

And if we have successful and unsuccessful results, we can operate on those results using operations similar to the logical operators AND and OR:

- `prog1 && prog2` — means running prog2 if prog1 succeeds (with 0 exit code)
- `prog1 || prog2` — means start prog2 if prog1 failed (exited with code other than 0)

Our programming language also has the good old "`if`":
B: `if list; then list; [ elif list; then list; ] ... [ else list; ] fi`
C: `if (list) then list; [ else if (list) then list; ] ... [ else list; ] endif`

And what is this "list" in this case? These are just a few commands. The exit code of the command will determine the behavior of the 'if'.

Test

And the most commonly used command in 'if' statements is 'test'. And the most commonly used command in statements is 'if' — 'test' or just an opening square bracket. It is often just a link to the executable 'test'. Be aware that if you use a square bracket, you must close the expression with a closing square bracket. And the space before that is important.

```
test EXPR or [ EXPR ]

Expressions:
-n STR | STR - STR is not zero
-z STR - STR is zero
! EXPR - EXPR is false
EXPR1 -a EXPR2 - AND
EXPR1 -o EXPR2 - OR
STRING1 = STRING2 - the strings are equal
STRING1 != STRING2 - the strings are not equal
INT1 -eq|ge|gt|le|lt|ne INT2 - INT1 and INT2 cmparison. Good reason to remember Fo
-f FILE - FILE exists and is a regular file
-d FILE - FILE exists and is a directory
-L FILE - FILE exists and is a symbolic link
```

and many others.

0.0.1 Loops

We also have loop statements — 'while', which execute commands while the statement command returns 0::

```
B: while list; do list; done
C: while (list) list; end
```

and 'until', executing until the statement command fails.

```
B: until list; do list; done
```

We also have a 'loop' construct, which may not seem very familiar to programmers from classical programming languages such as C:

```
B: for name in word ...; do list ; done
C: foreach name (word ...) list ; end
```

The source of the values that are set for the loop variable is simply a string of words, separated by spaces. The source of the values that are set for the loop variable is simply a string of words, separated by spaces. To emulate the classic number cycles, we have to use special commands that will generate sequences of numbers for us. Like 'seq' command:

```
$ man seq
$ for i in ‘seq 5‘; do echo $i; done
1
2
3
4
5
$ for i in $(seq 1 2 10); do echo $i; done
1
3
5
7
9
```

You can use the classic "break" and "continue" loop operators with the ability to set the loop level:

```
break [n], continue [n]
```

Case switch

A case command first expands word, and tries to match it against each pattern in turn, using the same matching rules as for pathname expansion:

```
case word in [ [(] pattern [ | pattern ] ... ) list ;; ] ... esac
```

If the ;; operator is used, no subsequent matches are attempted after the first pattern match.

```
$ vim hello
if [ $# -lt 1 ]
then
      echo Usage: $0 who...
      exit
fi

case $0 in
      *hello)
             MSG=Hello
             ;;
      *bye)
             MSG=Bye
             ;;
      *)
             MSG="I do not know what"
esac

echo $MSG $*!
$ ./hello world
Hello world!
$ ./bye bye
Bye bye!
$ ln -s hello nothing
$ ./nothing to say
I do not know what to say!
```

Function

This defines a function named name. The reserved word 'function' is optional.

```
[function] name () {list}
```

And we can '`return`' some exit code from the function.

```
return [n]
```

Arguments in the function are treated in the standard way as dollar with positional parameter number. From the point of view of an external observer, the function looks exactly the same as a regular command. And as we remember, some Shell libraring is possible:

```
$ vi lib
usage() {
         echo Usage: $1 args...
         exit
}
```

Let's check:

```
$ set
...
usage ()
{
    echo Usage: $1 args...;
    exit
}
$ usage test
Usage: test args...
exit
```

It works. And now we will use it in our script:

```
$ vi hello
source ./lib

if [ $# -lt 1 ]
then
       usage $0
fi
```

```
case $0 in
       *hello)
              MSG=Hello
              ;;
       *bye)
              MSG=Bye
              ;;
       *)
              MSG="I do not know what"
esac

echo $MSG $*!
$ ./hello
Usage: ./hello args...
```

Useful functions

And finally — a lot of useful functions embedded in Shell:

- `basename` –strip directory and suffix from filenames
- `dirname` –strip non-directory suffix from file name
- `echo` –display a line of text
- `eval` –execute expression by the shell
- `exec` –replace the shell by command
- `read` –read string from stdin to variable
- `readonly` –variables are marked readonly
- `shift` –shift parameters
- `sleep` –delay execution for a specified amount of time

And it would be helpful to be able to understand what exactly we are running when we run the command. And we have the following commands:

```
which, type - which command?
```

Let's try to run:

```
$ type usage
usage is a function
usage ()
{
```

```
    echo Usage: $1 args...;
    exit
}
$ which usage
/usr/bin/which: no usage in (/usr/local/bin:/usr/local/sbin:/usr/bin:/usr/sbin:/bi
```

Why are the results so different? Just because the first is a built-in function and the other is a real command:

```
$ type type
type is a shell builtin
$ type which
which is aliased to ‘alias | /usr/bin/which --tty-only --read-alias --show-dot --s
```

For an external binary, the results are the similar:

```
$ which sh
/usr/bin/sh
$ type sh
sh is /usr/bin/sh
```

UNIX/Linux administration

By installing the system on your computer, you become more or less an administrator and you need to have some basic administration skills. The most important tasks:

- Users and groups management;
- Working with repositories and packages;
- Devices and drivers handling;
- File systems configuring;
- Archiving and backups;
- Network administration.

Typically, system administration in different UNIX-like systems is the most different part of the system, although the general approaches to administration are more or less the same everywhere. On some systems, you have tools that can help you perform some of the *adminisconsolehelpertrative* tasks. For example:

- `gnome-control-center` in systems with GNOME UI
- RHEL: simple text config — `setup`, GUI-configs — `system-config-*`
- commercial systems provide their own more or less administrator-friendly tools

As we understand it, we need superuser rights to perform such tasks. Some systems may require stricter restrictions where system administration tasks can be decoupled from those of a security officer using mandatory access control (MAC) systems, such as those developed by the National Security Agency (NSA) SELinux subsystem in the Linux kernel.

Let's take a look at the RH '`setup`' tool:

```
$ setup
You are attempting to run "setup" which requires administrative
privileges, but more information is needed in order to do so.
Authenticating as "root"
Password:
```

We have to enter the root password and after that we can do some settings:

- Authentication configuration
- Keyboard configuration
- System services

But when we run '`system-config-date`', the system asks for the user's password. This is because these programs use different machinery for increasing privileges:

```
$ ls -l /usr/bin/setup
lrwxrwxrwx. 1 root root 13 Nov 9 2019 /usr/bin/setup
                                              -> consolehelper
```

The setup program is just a symbolic link to 'consolehelper', a tool that allows console users to easily run system programs. And the pkexec runner is used to execute 'system-config-date':

```
$ cat /usr/bin/system-config-date
#!/bin/sh

exec /usr/bin/pkexec \
          /usr/share/system-config-date/system-config-date.py
```

A more general way is to just switch to the 'root' superuser, and the first way to do this is with the su command:

```
man su
```

su — run a command with substitute user and group ID, by default — to 'root' superuser. For such a switch, we need to say the password of this user. When called without arguments 'su' defaults to running an interactive shell as 'root'. A very important option is just a 'dash', it's mean — starts the shell as login shell with an environment similar to a real login.

After switching to superuser "root" your prompt will change from a dollar sign to a hash sign:

```
$ id
...
$ su -
Password:
# id
...
# logname
...
```

On BSD systems, for security reasons, only users in the 'wheel' group (group 0) can use 'su' as 'root', even with the 'root' password. In many UNIXes and Linux the Plugin Authentication Module (PAM) is now being used to fine tune the privilege change. The settings for this subsystem are located in the /etc/pam.d/ directory.

```
$ ls /etc/pam.d/
```

And one of the applications whose config files we can find in this directory is the '`sudo`' command. The default PAM security policy allows users configured appropriately in '`/etc/sudoers`' to run commands with 'root privileges. And you don't need to know the password of 'root' user to do this.

Also, by default only one command is executed with '`sudo`', instead of '`su`' where we have to use the '`-c`' option to run one command. This reduces the chances of an unexpected error for an inexperienced user. And this is, for example, the default policy for Ubuntu systems. When Ubuntu is installed, a standard root account is created, but no password is assigned to it. You cannot log in as root until you assign a password for the root account. Only '`sudo`' may be used with such default settings.

To allow a regular user to run '`sudo`' this way on RH based systems such as Fedora, RHEL, CentOS, our NauLinux, you must add this user to the '`wheel`' group (as in BSD). And the easiest way to get a 'root' shell session like in 'su' with 'sudo' in Ubuntu is to just run it '`sudo -i`' (interactive).