Tekstmanipulatie, week 15

1. More about Unix: processes ('ps', 'kill', &), compression ('tar'), 'find', timing ('at'), variables ('set')

Most of the commands that we have learnt are actually executable (most often, binary) files. You can locate those files in the directory system by using the command "which" (for instance, "which cp" will return something like: /bin/cp; "which grep" may return: /usr/bin/grep). If you don't give the full path (absolute or relative path) of a program to be executed (let it be a standard Unix command or an executable file that you have created yourself), then the shell uses the variable "PATH", and checks the directories specified therein. You can read the value of this variable, either by typing echo $PATH, or by typing set | grep PATH.

It is a key idea of Unix to run more processes in the same time. Each user will run his or her processes. For each user, at least a shell is running. Furthermore, even one user can run more processes in the same time, and this is indeed the case almost always. The system also runs processes by its own.

A process can launch a *child process*. This is the case for instance when you run a command or a program in shell: then shell launches a child process. If the command line ends with the & symbol, then you get back the prompt immediately. Otherwise, shell waits for its child process to terminate.

The key idea for having multiple processes in the same time is *time sharring*. The CPU time is divided into small time slots, and each process is assigned some time slots. Because these time slots are very small, the human user has the feeling that the CPU deals exclusively with him or her.

The command "time" is usefull when you want to check how much time the CPU really needs to execute a command line. If you measured the time on your own watch, you would also measured the time needed to execute all kinds of other processes running in the same time.

The command 'at' can be used, if you don't want to run a process immediately, but at some moment in the future (for instance, in the night, when the computer is less used by other processes).

What to do if you want to kill a process (e.g.: your program has run into an infinite loop; an application is frosen down; ...)?

- Use the command 'ps' to list at all running processes. It is useful to type: ps -e | more .
- Check the process ID of the process to be killed. You can kill only your own processes. Be sure that you have the correct ID, otherwise you will kill some other processes. Then type: `kill -9 process_id [process_id ...]` .

More shell scripts: conditions, cycles

2. Conditions

There are two way of branching your program in a shell script.

The first one is '`case`'. Its syntax is:

```
case <selector> in
  <value1> ) <commands1> ; ;
  <value2> ) <commands2> ; ;
  <value3> ) <commands3> ; ;
...
  <valueN> ) <commandsN> ; ;
esac
```

Notice the) paranthesis after the values, as well as the double ; ; semicolons after the commands.

The way this works is the following: you take the value of <selector> (e.g. an argument or a positional parameter / argument of your script) and compares it with <value1>. (The values can include the same wildcards as seen with `ls`, etc.) If there is a match then <commands1> are executed, and continue with the line after '`esac`' (='case' reversed). Otherwise we try to match the selector with <value2>: if there is match then <commands2> are executed, and continue with the line after 'esac', otherwise we go forward to <value3>, etc. If there is no match even with the last line, then we go forward. Notice that only the first match is executed, and not all of them! An example:

```
birot@hagen:~> cat branch
case $1 in
  al?a) echo 'al?a';;
  ????) echo '????';;
  [ab]*) echo '[ab]*';;
  *) echo else;;
esac

birot@hagen:~> branch alfa
al?a
birot@hagen:~> branch beta
????
birot@hagen:~> branch apple
[ab]*
birot@hagen:~> branch everything else
else
```

Notice the way that we have constructed the last branching value: it is an "anything else" branch.

The other way to insert conditional branchings into your shell script is by using the if command. Its syntax is:

> if <commands1> ; then <commands2> ; else <commands3> ; fi

How does this work? How comes that we have commands even after 'if', where one would expect conditions? The answer is that the execution of each command line has three 'results' (see also week 3): the output (in the -- redirected -- standard output, that is usually the real set of "results"), the "error" (not necessarily real errors, but another type of output text, used usually for messages related to the way the program runs), as well as the **return value**. For most commands a return value of 0 means that the program ran without any problems, and other values

are characteristical to different sorts of errors, or other special cases (e.g. the return value of grep is 0 if the pattern was found in the given file, 1 if the pattern was not found, a case that is not a real error, and the return value is 2 if there were syntax errors in the pattern or if the input file was inaccessible, etc.).

If there were a pipeline or a series of commands (devided by ; for instance) then the return value of the last command is the return value to be taken into consideration. It is also important to remember that these are real commands that are really running: they can change your file system and write on the screen (to avoid it, you can redirect the input to a trash-file or to /dev/null, see the notes on the buttom of this page).

Let's come back to 'if'. Now we'll get a little bit confused. If the return value is 0 (i.*e.* the command line didn't encounter any trouble) then <commands2> are executed. Otherwise <commands3> are executed. In other words: 0 is considered "true", and other values are considered "false" in the condition, unlike in any other cases (e.g. the way expr deals with logical values). But to get back to normal logic, the return value of expr is 0 if its output (the value of the expression to be evaluated) is non-0, and the return value is 1 if the output is 0. (A return value of 2 means some syntactic error has occured.)

Therefore you can use the following construction:

```
if expr $1 = apple > /dev/null; then echo Goed;
else echo Niet goed; fi
```

If the first argument of your script is 'apple' then you will get the text 'Goed' on your screen, otherwise the text 'Niet goed' will appear.

Notice the syntax: all commands are terminated by a semicolon (;), or alternatively by an end-of-line. So the commands 'then', 'else' and 'fi' either should be preceeded by a semicolon (;) or should appear in a new line.

The entire construction terminates by the command ' fi', that is the reverse of 'if'. It shows the end of the structure, and once either the 'then' branch or the 'else' branch has been executed (or nothing has been done if the else-branch was missing and the condition was false), we go to the command appearing after ' `fi` '.

If the 'else'-branch starts with another 'if' then 'else if' can be abriviated as 'elif'.

A last example:

```
birot@hagen:~> cat example-if
#!/bin/bash
if echo $*; echo These are your arguments; expr $1 +
$2 = 2; then echo You are lucky; fi
echo Bye...

birot@hagen:~> example-if 1 1
1 1
These are your arguments
1
You are lucky
Bye...

birot@hagen:~> example-if 2 3
2 3
These are your arguments
0
Bye...

birot@hagen:~> example-if 3 -1
3 -1
These are your arguments
1
You are lucky
Bye...

birot@hagen:~> example-if 3 -1 2 tree house and so
on...
3 -1 2 tree house and so on...
These are your arguments
1
You are lucky
```

```
Bye...
```

Notice that:

1. The else-branch is not compulsory, so if you have only one branch, try to formulate your condition in such a way that you would use just the then-branch.
2. The command line between 'if' and 'then' is always executed,...
3. ...including the outputting of the 'expr' command (0 stands for false, 1 for true). If you want to avoid this, use `> /dev/null` (everything is a file... even the nothing is a file...).
4. $* is replaced by all my arguments.

3. Cycles (loops)

There are three types of cycle in Unix, and all of them checks the condition before the core of the cycle. Here they are:

`for` <variable> `in` <list> ; `do` <command command ...> ; `done`

The variable takes all the values in the list, and the commands are executed for each value. Example:

```
birot@hagen:~> for a in hello 'how are you'
fine thanks ; do echo $a; done
hello
how are you
fine
thanks
```

In a shell script the ; symbol can be replaced by just putting the next command in a new line.

The second type of cycle is:

`while` <command command...> ; `do` <command command

...> ; done

The third one is:

until <command command...> ; do <command command ...> ; done

The way they work is the following:

1. They check the return value of the first command line (the return value of the last command there), as seen in the case of "if".
2. If it is true for the while-cycle, or if it is false for the until-cycle, then the second command line is executed (van 'do' tot 'done'), and then we get back to the first commands (the conditions)
3. if the condition was false for the while-cycle, or true for the until-cycle, then the second command line is not executed any more, we continue with the command after 'done'.

The condition is called "cycle condition" for the while-cycle (the core of the cycle gets executed while / as long as the cycle condition is true). The condition is called "exit condition" for the until-cycle, because we leave the cycle when it is true (the core of the cycle gets executed as long as the condition is false).

An example:

```
birot@hagen:~> a=1; while expr $a \< 10 >
/dev/null; do echo $a ; a=`expr $a + 1`; done
1
2
3
4
5
6
7
8
```

```
9
```

A useful tip: if you use `expr` (or something else), but you don't want to get the result on the screen because you just need the return value (e.g. it is a command in an if-condition or in the condition of a cycle in a shell script), you should re-direct its input. You can do that either into a file of yours (e.g. `> trash`), or to the file /`dev/null`. This latter reminds you the "terminal-files" (/`dev/tty1`,...), but is just a "black hole": whatever you put into it, would never appear.

E.g.:

```
while expr $fibo \< 10000 > /dev/null ; do ...
; done
```

Summary of the syntax:

It is always very important to pay attention to syntax, whatever programming language or operating system you are working with. Pay attention to the exact spelling of the commands, the order of the arguments, to compulsory semicolons or other symbols, to what is compulsory and what is optional, etc.

In the following `<commands>` means a list of one or more commands (or pipelines) separated either by a semicolon (`;`) or by a new-line character. Although it is possible to write the commands in one line, I propose to use the multi-line syntax, in order to make your script readable. (Otherwise it will become "write only"...)

1.

```
case <selector> in
     <value1> ) <commands> ;;           // Notice the
     unusual closing bracket and the double semi-column!
     <value2> ) <commands> ;;
     ...
     <valueN> ) <commands> ;;
esac
```

2.
```
    if <commands> ; then <commands> ; [ else <commands> ;]
    fi
```
3.
```
    if <commands>
          then <commands>
          [ else <commands>]
```
// The [..] brackets stand for optionality
```
    fi
```

4.
```
    for <variables> in <list> ; do <commands> ; done
```
5.
```
    for <variables> in <list>
          do <commands>
          done
```
6.
```
    while <commands> ; do <commands> ; done
```
7.
```
    while <commands>
          do <commands>
          done
```
8.
```
    until <commands> ; do <commands> ; done
```
9.
```
    until <commands>
          do <commands>
          done
```

Useful examples:

How to read the argument list of the schell script, one-by-one? Use something like: `for VARIABLE in $*` . In this case, the variable will be assigned all the different arguments of the shell script, one-by-one, and the kernel of the script will be run for each argument.

As an example, suppose that you have hundred files, and you want to rename all of

them. In particular, you want to add the "extension" `.exec` to all file names. One solution is to use the command `rename` . However, if you do not trust it, you can write a short script performing this task:

```
> ls
apple pear zebra lion
> cat my_rename
    #! /bin/bash
   for FILE_NAME in $*
       do
        mv $FILE_NAME $FILE_NAME.exec
      done
> chmod 744 my_rename
> ./my_rename *
> ls
apple.exec pear.exec zebra.exec lion.exec
```

Bíró Tamás:
e-mail
English web site
Magyar honlap

Last modified: Thu Jul 3 11:39:17 MEDST 2003