UNIX AND LINIX IN INFOCOMMUNICATION
Week 9

O. Sadov

OK, let's take a step forward:

```
$ git checkout Example_2
Previous HEAD position was 8a16efc... Added simple Makefile
HEAD is now at 757a731... Added "calc" shell-script.
```

We see some changes in the README:

```
$ cat README.md
...
```

and in our repository — we can see the 'calc' script:

```
$ ls
calc Makefile README.md
```

This means — we can mark some stages of our development with tags. And then, using the "checkout" operation, we can switch between them at any time. OK — let's look at our calculator:

```
$ cat calc
#!/bin/sh

expr $*
```

Wow — looks pretty simple! We simply call the expr program with the arguments passed to our script. And as we can see, 'expr' is just an expression evaluator:

```
man expr
```

And this is the usual way of UNIX development — not reinvent the wheel, but just take parts of them and glue them with a Shell. OK — let's try to test:

```
$ ./calc 1 + 2
3
```

We understand that the expression is just the arguments of the 'expr' command and we must separate them with spaces.

```
$ ./calc 5 - 7
```

```
-2
$ ./calc 6 / 3
2
```

Looks good — let's try divide:

```
$ ./calc 6 / 3
2
$ ./calc 6 / 0
expr: division by zero
$ ./calc 7 / 3
2
```

OK, got it — '`expr`' performs integer operations and we have to use another operation for the remainder of the division:

```
$ ./calc 7 % 3
1
```

But:

```
$ ./calc 2 * 2
expr: syntax error
```

What's wrong? Let's try to debug:

```
$ sh -x ./calc 2 * 2
+ expr 2 Makefile README.md calc 2
expr: syntax error
```

Oh yeah — the asterisk are just special Shell matching characters, in this case meaning — all files in the current directory! And it is replaced by all files from the current directory. We need to fix it. Let's move on to the next tag:

```
$ git checkout Example_3
HEAD is now at d7f58c6... Shell spec. symbols escaping and input formatting for 'e
$ git diff Example_2 Example_3
...
```

As you can see, the files have changed: README and our script '`calc`'. Let's take a look at '`calc`':

```
$ cat calc
#!/bin/sh

FILE=/tmp/calc-$$

read EXPR
echo $EXPR | sed 's/\([^0-9]\)\([^0-9]\)/\1 \2/g; s/\([0-9]\+\)\([^0-9]\)/\1 \2/g;
sh $FILE
EXIT_STATUS=$?
rm -f $FILE
exit $EXIT_STATUS
```

It looks more complicated — let's analyze the changes. So the main idea behind the fixes is to read the expression from stdin, not from the script parameters, and modify it to avoid the Shell matching mechanism.

OK. On the third line, we define a `FILE` variable with a unique name to store temporary data for evaluating '`expr`'. The uniqueness is guaranteed by a special double dollar environment variable that indicates the PID of the current process.

On the fifth line, we read the expression from stdin into the `EXPR` variable. We then edit it on the fly with the '`sed`' stream editor. With the '`sed`' command, we insert spaces between numbers and signs of arithmetic operations, we do the escaping with the slash character before the asterisk and brackets. Finally, before the expression, insert the '`expr`' command and redirect the output from '`sed`' to a temporary file defined at the beginning of the script.

We then run our temporary script with '`sh`', getting the exit status from the special variable "dollar question" into "`EXIT_STATUS`" variable, deleting the temporary file, and exiting with the parameters stored in "`EXIT_STATUS`". We need such a complex construction because '`rm -rf`' will return a success status regardless of the result of evaluating the expression.

OK — let's check our fixes:

```
$ ./calc
2*2
4
```

It works! And we don't even need to insert spaces between parts of our expression. Let's check — maybe something else was broken by our corrections?

```
$ ./calc
1+2
3
$ ./calc
5-7
-2
$ ./calc
6/3
2
$ ./calc
7/3
2
$ ./calc
7%3
1
```

Looks good. What about expected errors?

```
$ ./calc
6/0
expr: division by zero
$ echo $?
2
```

Great — we got it!

In fact, we can implement a simpler solution and in the sixth line just write:

```
echo $(($EXPR)) > > $FILE
```

But we wrote a script that still works with older shells that may not support such constructs, and we also looked at how to use non-interactive editing in scripts. OK — the next example

```
$ git checkout Example_4
HEAD is now at d7f58c6... Shell spec. symbols escaping and input formatting for 'e
```

It seems strange — the commit and comment are similar to the previous example. Let's check diff:

```
$ git diff Example_3 Example_4
$
```

Oh yeah — I was wrong, I placed the tag wrong — until the real changes! This is actually a good reason to take a deeper look into our repository to fix this. First, let's move on to the next example:

```
$ git checkout Example_5
Previous HEAD position was d7f58c6... Shell spec. symbols escaping and input forma
HEAD is now at f266a24... GUI
```

Ok — seems differ. Now let's look to log:

```
$ git log
commit f266a24128b1e363eddc073682aac89dd33a86a8
...
    GUI
...
commit d6453c0c41548a55e3249ea8c3b788c71cb76f7e
...
    Text UI.
...
commit d7f58c65c3e25269977538fdde0ac13d733fbf92
...
```

We see another commit between the previously discussed commit and the GUI commit — the text interface. Let's switch to it:

```
$ git checkout d6453c0c41548a55e3249ea8c3b788c71cb76f7e
Previous HEAD position was f266a24... GUI
HEAD is now at d6453c0... Text UI.
```

And what about diff?

```
$ git diff Example_3 d6453c0c41548a55e3249ea8c3b788c71cb76f7
```

OK — changed `README`, `Makefile` and added new file:

```
$ ls
Makefile README.md calc calc_ui
```

The new 'calc_ui' script is the user interface for our simple command line calculator. Let's take a look inside:

```
$ cat calc_ui
...
```

First, we see setting environment variables for temporary files. Then we define an 'end' function in which we delete the temporary files and exit the program. The main action in the program is an infinite loop, in which we just call a 'dialog' program and then work with the results it returns. What is it 'dialog'? To understand what it is, you first need to install:

```
$ sudo yum insatll dialog
```

In Ubuntu, we have to install this program as follows:

```
$ sudo apt install dialog
```

Now we just execute 'dialog':

```
$ dialog
```

We see a lengthy help that shows us which parameters we should use to create various interface forms. For example:

```
$ dialog --yesno "To be or not to be?" 5 25
```

Make your choice and see the program exit code:

```
$ echo $?
```

You will see zero if you choose 'yes' and non-zero if 'no'. As we understand it, this is a one-shot program that shows us a uniform interface form and returns some result that we can use in our script. In our UI we use the `--inputbox` form and redirect the standard error from it to a temporary `FILE1`.

What does standard error have to do with our command? Hopefully we didn't expect any errors, just a line with our expression? Yes, but the standard

output of the dialog program is already in use for drawing the UI form. As far as we understand, to draw such pretty forms, a bunch of ESC sequences for your terminal type are sent to standard output. These sequences, generated by the ncurses library, are retrieved from a terminal type database according to the TTY environment variable.

Thus, if the dialog form ended with an error exit code, it means that we clicked the "Cancel" button and then calling the "`end`" function. If we clicked OK, we perform the following operation — we send our input to our good old '`calc`' script and redirect the output and error output to separate files. And then show a dialog form with the result if the script completed successfully, and an error form if we received an error.

So it looks good. And this is an example of the KISS design principle — we developed a simple script and just wrapped it with another simple script that implements the UI. But for the final preparation of our application for work, we need to install our '`calc`' script to the directory from the `PATH` environment variable. And for this, we added a corresponding rule to the Makefile:

```
$ cat Makefile
$ sudo make install
[sudo] password for liveuser:
install calc calc_ui /usr/local/bin
```

Now let's play with our user friendly calculator:

```
$ calc_ui
```

Great! Well. Let's go back to the fifth example:

```
$ git checkout Example_5
Previous HEAD position was d7f58c6...
Shell spec. symbols escaping and input formatting for 'expr'.
HEAD is now at f266a24... GUI
$ git diff d6453c0c41548a55e3249ea8c3b788c71cb76f7 Example_5 | less
...
```

Looks too long, but let's take a closer look — actaully added only the gdialog file:

```
$ ls
Makefile README.md calc calc_ui gdialog
```

This is not our development — it's just a GTK+ graphical analogue of the 'dialog' text program. Our changes simply add an install command for gdialog if not already installed:

```
cat Makefile
```

and change four lines in 'calc_ui':

```
cat calc_ui
```

We first check for the existence of the 'gdialog' program, and if it exists on our system, we set the DIALOG environment variable as 'gdialog', if not, we set it as 'dialog'. And then just we replaced all the places where the 'dialog' is used with the environment variable DIALOG.

OK. Install the new version:

```
$ sudo make install
[sudo] password for liveuser:
install calc calc_ui /usr/local/bin
which gdialog >/dev/null 2>&1 || install gdialog /usr/local/bin
```

and start our script again:

```
$ calc_ui
```

It works. But remember, the only thing used to communicate between the XWindow server and the client is the DISPLAY environment variable:

```
$ echo $DISPLAY
:0
```

Let's unset it and try to run our UI again:

```
$ unset DISPLAY
$ calc_ui
```

Tadaam – we've got a text interface again! Simply because the 'gdialog' script automagically switches to the text 'dialog' if we are working in text mode. Just set DISPLAY and we get the GUI again:

```
$ export DISPLAY=:0
$ calc_ui
```

Let's add some networking to our design:

```
$ git checkout Example_6
Previous HEAD position was f266a24... GUI
HEAD is now at d3e5228... Network server
$ git diff Example_5 Example_6
...
$ ls
Makefile README.md calc calc.services calc.xinetd calc_ui gdialog
```

As you can see, just two new files are `calc.services` and `calc.xinetd`. Because we will go the easy way — we will create a service for the `xinetd` superserver. As we recall, for this we just need a program that reads standard input and writes to standard output. And we also have such a program — this is our '`calc`' script!

To start our own network service, we just need to create the correct configuration for the `xinetd` server and we have to extend our '`install`' target in the Mnetcatakefile:

```
$ cat Makefile
...
```

We check the '`/etc/services`' configuration file for a '`calc`' service port definition and if it doesn't exist, add it:

```
$ cat calc.services
calc 1234/tcp
```

As we can see, we are configuring our service on TCP port 1234.

And finally we install calc xinetd config file a `/etc/xinetd.d/calc`:

```
$ sudo yum install xinetd
```

or in Ubuntu:

```
$ sudo apt install xinetd
```

and then:

```
$ sudo make install
install calc calc_ui /usr/local/bin
which gdialog >/dev/null 2>&1 || install gdialog /usr/local/bin
grep -q "‘cat calc.services‘" /etc/services || cat calc.services >> /etc/services
install calc.xinetd /etc/xinetd.d/calc
```

Let's restart '`xinetd`' after installing of our service:

```
$ sudo service xinetd restart
```

We can use '`telnet`' or the lighter '`netcat`' to test our server — this is the '`nc`' package on RH-like distributions:

```
$ sudo yum install nc
$ nc localhost 1234
Ncat: Connection refused.
```

or '`netcat`' on Ubuntu:

```
$ sudo apt install netcat
$ nc localhost 1234
$
```

Does not work... Let's understand the problem — look in the system log. As we recall, we can find it in '`/var/log/syslog`' on Debian-based Ubuntu and '`/var/log/messages`' on RH-like systems. And on any '`systemd`' based system we can use '`journalctl`' for this:

```
$ sudo journalctl
...
... xinetd[4449]: Server /data/home/sadov/works/courses/calc is not executable
... xinetd[4449]: Error parsing attribute server - DISABLING SERVICE [file=/et
```

Well. We found the root of the problem: this is another mistake of mine — I did not change my experimental "`calc`" script path to our standard "`/usr/local/bin/calc`". Let's fix it:

```
$ sudo vim /etc/xinetd.d/calc
...
```

```
        server = /usr/local/bin/calc
...
```

And restart 'xinetd' service:

```
$ sudo service restart
```

Redirecting to /bin/systemctl restart xinetd.service

```
$ nc localhost 1234
2+3
5
^C
```

Great - we're in the network now!