UNIX AND LINIX IN INFOCOMMUNICATION
Week 9

O. Sadov

Introduction to development

And now we are ready to do some kind of development project. We will try to implement the following traditional task for novice developers — a calculator. And finally, we will get a network client-server application with a graphical user interface localized into Russian. We can find the code for this project on github:

https://github.com/itmo-infocom/calc_examples

Github supports hosting for software development and version control with Git.

Version control (also known as revision control, source control, or source code management) is a class of systems responsible for managing changes to computer programs, documents, large web sites, or other collections of information. Version control is a component of software configuration management.

VCS Release History Timeline

The first generation VCS were intended to track changes for individual files and checked-out files could only be edited locally by one user at a time. They were built on the assumption that all users would log into the same shared Unix host with their own accounts. The second generation VCS introduced networking which led to centralized repositories that contained the 'official' versions of their projects. This was good progress, since it allowed multiple users to checkout and work with the code at the same time, but they would all be committing back to the same central repository. Furthermore, network access was required to make commits. The third generation comprises the distributed VCS. In a distributed VCS, all copies of the repository are created equal — generally there is no central copy of the repository. This opens the path for commits, branches, and merges to be created locally without network access and pushed to other repositories as needed.

There are many VCS systems developed and used on UNIX-like systems: SCCS, RCS, CVS, SVN, Mercurial, Bazzar, etc., and Git is now the most popular. Git is a distributed version-control system for tracking changes in source code during software development. It was created by Linus Torvalds in 2005 for development of the Linux kernel, with other kernel developers contributing to its initial development.

Devlopment tools

The basic tool for working with Git repositories is the ‘`git`’ utility:

```
man git
```

First we have to ‘`clone`’ our repository:

```
$ git clone https://github.com/itmo-infocom/calc_examples
$ ls calc_examples/
calc calc_ui calc_ui-ru.po gdialog README.md
calc.services calc_ui.pot calc.xinetd Makefile
```

Ok — we now have our own local copy. Let’s go inside:

```
$ cd calc_examples/
```

Now let’s take a look at the tags that indicate different stages of development — tags:

```
$ git tag
Example_1
Example_2
Example_3
Example_4
Example_5
Example_6
Example_7
Example_8
Example_9
```

Let’s move on to the initial stage now:

```
$ git checkout Example_1
Note: checking out 'Example_1'.
...
$ ls
Makefile README.md
```

As we can see, most of the files have disappeared and now we only have a README and a Makefile. Let’s take a look at the README:

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

and at the Makefile:

```
$ cat Makefile
clone:
...
```

A `Makefile` is a file containing a set of directives used by a '`make`' build automation tool to generate a target/goal. Make is the oldest and most widely used dependency-tracking tool for building software projects. It is used to build large projects such as the Linux kernel with tens of millions of source code lines. Usually, using make is pretty simple: you just run make with no arguments:

```
man make
```

In this case, the utility tries to find a makefile that starts with a lowercase letter and then with a capital letter. Usually the second Makefile with an uppercase letter is used, the lowercase version is often used for some local changes while testing and customizing the Makefile of an upstream project.

The makefile format is pretty simple — It's just a lot of rules for building projects:

```
$ cat Makefile
```

In the first position we see the '`target`', after the semicolon, dependencies can be placed, which are just other targets. Lines following the description of targets and dependencies must start with a TAB character, and there are commands that must be executed to complete this target. If the dependencies are files, make checks the modification times of those files and recursively rebuilds those dependency targets.

The current Makefile just includes a few targets for working with the Git repository. The first will start by default. If we want to run another target, we must pass it as a parameter when running the '`make`' utility. 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! Ok, we have a working network service, and it's time to change our user interface to network communication with it.

```
$ git checkout Example_7
$ git diff Example_6 Example_7 | less
```

The main changes are made in the '`calc_ui`' script. At the beginning of the script, as you can see, added some default definitions: `HOST`, `PORT` for calc-server connection and `CALC` script name.

Then we can see added support for configuration files. First, we check the user's personal configuration in the home directory — this is a hidden file starting with a "dot" `calc.conf`. If such a file exists, we load it as a Shell library file. If we do not find this file, we check the system-wide configuration '`/etc/calc.conf`' and load it if it exists. You can put your own connection variable definitions in these files.

We see a new '`help`' function for displaying a help message and some logic for processing script parameters. We now handle the '`--help`' parameter and the optional host/port positional parameters. As we can see, we can only set the host or both the host and port using the script arguments.

And finally — we check the name of the script by stripping it with the '`basename`' function from the current path to the script. If the script is called '`ncalc_ui`', we change the `CALC` variable to '`netcat`' with the host access parameters. If such parameters are not specified, we display an error message. In the line where we called the '`calc`' script, we replace it with the `CALC` variable.

That's it — we just changed the UI file without changing the main logic. Let's install it:

```
$ 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
grep -q "`cat calc.services`" /etc/services || cat calc.services >> /etc/services
install calc.xinetd /etc/xinetd.d/calc
ln -s /usr/local/bin/calc_ui /usr/local/bin/ncalc_ui
```

As we can see, there is a symbolic link from '`calc_ui`' to '`ncalc_ui`'.

OK. Let's test it:

```
$ ncalc_ui
```

It works! But maybe we're wrong and this is just our old local calculator? Let's check by stoping the '`xinetd`' service:

```
# systemctl stop xinetd
```

Our calculation ended with an error. And what about a non-networked '`calc_ui`'?

```
$ calc_ui
```

Still works. Let's run '`xinetd`' again:

```
# systemctl start xinetd
```

Networked '`ncalc_ui`' works again! Quod erat demonstrandum — the networked version of the calculator created! Now let's try to localize our program.

```
$ git checkout Example_8
```

What is it "localize"? The locale is the primary mechanism for supporting the native languages in UNIX-like systems, and we have a few strange abbreviations that describe it: I18N, L10N, and even M17N. What does this mean? This means that English does not like long words, but the terminology associated with supporting native languages is too many letters. Such as

– Multilingualization for application software consisting of 17 letters between "M" and "N" and abbreviated to M17N. M17N is performed in 2 stages:

- Internationalization (18 letters between “I” and “N” — I18N): To make a software potentially handle multiple locales.
- Localization (10 letters between “L” and “N” — L10N): To make a software handle an specific locale.

OK. Let’s take a look at our current locale settings:

```
$ locale
...
```

As we can see, there are lot of environment variables associated with this:

- LANG and LC_ALL — General language setting. These settings can be separated to:
- LC_CTYPE — Controls the behavior of character handling functions.
- LC_TIME — Specifies date and time formats, including month names, days of the week, and common full and abbreviated representations.
- LC_MONETARY — Specifies monetary formats, including the currency symbol for the locale, thousands separator, sign position, the number of fractional digits, and so forth.
- LC_NUMERIC — Specifies the decimal delimiter (or radix character), the thousands separator, and the grouping.
- LC_COLLATE — Specifies a collation order and regular expression definition for the locale.
- LC_MESSAGES — Specifies the language in which the localized messages are written, and affirmative and negative responses of the locale (yes and no strings and expressions).
- LO_LTYPE — Specifies the layout engine that provides information about language rendering. Language rendering (or text rendering) depends on the shape and direction attributes of a script.

In the C programming language, the locale name C “specifies the minimal environment for C translation” (C99 §7.11.1.1; the principle has been the same since at least the 1980s). As most operating systems are written in C, especially the Unix-inspired ones where locales are set through the `LANG` and `LC_xxx` environment variables, C ends up being the name of a “safe” locale everywhere

On POSIX platforms such as Unix, Linux and others, locale identifiers are defined by ISO standard and consists from: ‘`[language[_territory][.codeset][@modifier]]`’. For example, Australian English dialect using the UTF-8 encoding is en_AU.UTF-8.

As we can see, our application already has l10n.

```
$ LANG=C calc_ui
$ LANG=en_US.UTF-8 calc_ui
```

As we can see, our application looks the same in base C and American English locales. In Russian we see the Russian title and buttons:

```
$ LANG=ru_RU.UTF-8 calc_ui
```

In Continental Simplified and Traditional Taiwanese Chinese, we can see different hieroglyphs sets on the title:

```
$ LANG=zh_CN.UTF-8 calc_ui
$ LANG=zh_TW.UTF-8 calc_ui
```

And for the Arabic language, the title is written from right to left:

```
$ LANG=ar_SY.UTF-8 calc_ui
```

Looks good, but this is just a basic localization of the ‘`zenity`’ utility used by ‘`gdialog`’. We need to translate our text strings in the program. And for that we can use the standard ‘`gettext`’ machinery that was first implemented by Sun Microsystems in 1993. To work with this, we need a ‘`gettext`’ utilities set:

```
# yum install gettext
# apt install gettext
```

We have a utility ‘`gettext`’ that translates the text messages passed to it as arguments according to the settings of the locale’s environment variables:

```
$ man gettext
```

As we can see, for this we just need to add a call to the ‘gettext’ utility in all places where we used a text string as a parameter to ‘gettext’, and insert the result of execution into command lines by quoting the command with “back apostrophes”:

```
$ git diff Example_7 Example_8
```

And now we can fetch the gettext strings from the source using the xgettext utility:

```
$ make calc_ui.pot
xgettext -o calc_ui.pot -L Shell calc_ui
```

We got the portable object template file ‘calc_ui.pot’ that we will use as a basis for translation:

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

For example to Russian language:

```
$ msginit --locale=ru --input=calc_ui.pot
...
$ cat ru.po
```

```
$ git checkout Example_9
error: The following untracked working tree files would be overwritten by checkout
       calc_ui.pot
Please move or remove them before you can switch branches.
```

Ok. Let’s delete this file — in the next example we already have it:

```
$ rm calc_ui.pot
$ git checkout Example_9
Previous HEAD position was 156bba2... L10N enabling for calc_ui
HEAD is now at b24e4a1... Fixed linking ncalc_ui
$ git diff Example_8 Example_9 | less
```

As you can see, we just made a translation into Russian of the corresponding PO file and added installation steps to the Makefile. Let's look at the PO file with Russian translation:

```
$ cat calc_ui-ru.po
```

First we have the metadata and then the pairs of translation strings: '`msgid`' with the source and '`msgstr`' with the translation. This is not a good translation strategy because, for example, some messages in one language may have different meanings in other languages in a different context. And in larger software projects like Firefox or Open/LibreOffice, other localization engines have been used in which each line from the user interface has its own unique identifier and translation for each one. And to simplify and unify the translation process, they use such complex tools as editors with support for the translation memory mechanism.

But gettext is widely used in the UNIX-like world, and that's enough for our purposes. Let's compile and install our translation now:

```
$ sudo make install
[sudo] password for liveuser:
msgfmt -o calc_ui-ru.mo calc_ui-ru.po
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
ln -sf /usr/local/bin/calc_ui /usr/local/bin/ncalc_ui
install calc_ui-ru.mo /usr/share/locale/ru/LC_MESSAGES/calc_ui.mo
```

As we can see, we compile our translation file '`calc_ui-ru.po`' into '`calc_ui-ru.mo`' and install the resulting binary localization file into the `LC_MESSAGES` locale directory. Let's check the result:

```
$ LANG=ru_RU.UTF-8 calc_ui
```

Great! We have a calculator in Russian now! Let's switch to English again:

```
$ LANG=en_US.UTF-8 calc_ui
```

In English again. For other languages, we still have the same interface because we didn't translate the messages for them:

```
$ LANG=zh_CN.UTF-8 calc_ui
```

And maybe it's a good time to translate our application into your favorite language…