UNIX AND LINIX IN INFOCOMMUNICATION
Week 9

O. Sadov

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...