UNIX AND LINIX IN INFOCOMMUNICATION
Week 6

O. Sadov

X-Window concepts

The history of the UNIX graphics system goes back to the 1984 MIT Athena education project. Athena was not a research project, and the development of new models of computing was not a primary objective of the project. Indeed, quite the opposite was true. MIT wanted a high-quality computing environment for education.

In collaboration with DEC and IBM, the project developed a platform-independent graphics system to link together different systems from multiple vendors through a protocol that can both run local and remote applications. This system was the basis of the X-Window System, which began its growth in 1985 and was ported to various UNIX and not just UNIX platforms.

We now have several successors to the classic X-Window system, the most famous being the Android or Wayland graphics system on Linux desktops, but X-Windows is still relevant today. And we will discuss some non-trivial concepts related to this. To see them just look into:

```
man X
```

First of all, the X-Window System is a network oriented client-server architecture. But the relationship between the client and the server doesn't seem very clear at first glance. Generally, the X server simply accepts requests from various client programs to display graphics (application windows) and sends back user input (from keyboard, mouse, or touchscreen). And the server will run on your tablet, and the apps that run on the supercomputer are just clients.

The main principle of X-Window: "*Provide mechanism rather than policy. In particular, place user interface policy in the clients' hands.*"

And the main locator for the X server is the DISPLAY environment variable:
`DISPLAY=hostname:displaynumber.screennumber`
Hostname is optional. Its absence means localhost. The display number is a unique identifier for this X server and should always appear in the display specification. And the screen number is an optional parameter for multi-screen X server configuration.

Let's try to play with this. First, we will switch to another virtual console, for example to second one by '`Ctrl+Alt+F2`'. Actually, in Linux we have

twelve virtual consoles in terms of the number of function keys on keyboard. And we can switch between them using '`Ctrl+Alt+function key`'. On some of them, we see a login prompt and can log in here. Now let's run X server:

```
$ X
(EE)
Fatal server error:
(EE) Server is already active for display 0
       If this server is no longer running, remove /tmp/.X0-lock
       and start again.
(EE)
(EE)
Please consult the The X.Org Foundation support
        at http://wiki.x.org
 for help.
(EE)
```

This is expected — we already have a running X-server in the system, which occupies a zero display. This is not a problem — let's try to run on the next display:

```
$ X :1
X.Org X Server 1.20.4
X Protocol Version 11, Revision 0
Build Operating System: 3.10.0-957.12.2.el7.x86_64
Current Operating System: Linux localhost 3.10.0-1127.18.2.el7.x86_64 #1 SMP Thu J
Kernel command line: initrd=initrd0.img root=live:CDLABEL=NauLinux_Qnet77-LiveCD r
rhgb rd
.luks=0 rd.md=0 rd.dm=0 BOOT_IMAGE=vmlinuz0
Build Date: 07 August 2019 08:52:04AM
Build ID: xorg-x11-server 1.20.4-7.el7
Current version of pixman: 0.34.0
       Before reporting problems, check http://wiki.x.org
       to make sure that you have the latest version.
Markers: (--) probed, (**) from config file, (==) default setting,
       (++) from command line, (!!) notice, (II) informational,
       (WW) warning, (EE) error, (NI) not implemented, (??) unknown.
(==) Log file: "/var/log/Xorg.1.log", Time: Wed Aug 26 18:45:45 2020
(==) Using system config directory "/usr/share/X11/xorg.conf.d"
```

```
(II) [KMS] Kernel modesetting enabled.
resizing primary to 1024x768
primary is 0x5645745b54b0
^C(II) Server terminated successfully (0). Closing log file.
```

It looks like a black screen without any graphic elements. Something is wrong? Oh no... On the fourth virtual screen, login again and set the DISPLAY variable:

```
$ export DISPLAY=:1
```

Now let's start the good old terminal interface — the 'xterm' application:

```
$ xterm
```

OK. Let's go to the third virtual console, where we left our X server. Great — we see the terminal window! But this is strange — we can only print something while staying in the terminal window, we cannot move or resize it, moreover, we do not have a button to destroy it!

It's just because we have a developed system based on the KISS paradigm — xterm simply emulates a terminal. If we want to move or resize windows (for example, we don't need this for an information kiosk), we need a special program for this — a window manager. Let's run it on 'xterm' by starting one of the graphical user environments — GNOME:

```
$ gnome-session &
```

OK. We now have a fully functional graphical user system in which we can work with graphical applications in the usual way.

GEOMETRY=WIDTHxHEIGHT+XOFF+YOFF
Now let's turn to another important point — geometry. With this parameter, we can set the position and size of the application window:

```
$ xterm -geometry 100x30+10+10
$ xterm -geometry 150x50+100+100
```

Finally, we can choose colors for applications that support such settings. X supports the use of abstract color names as described in the configuration file /usr/share/X11/rgb.txt. In this file, we can see the red, green and blue

values for the named color definitions.
`COLOR=<color_space_name>:<value>/.../<value>`
And we can use color names like this for example as application parameters:

```
xterm -bg blue -fg red
```

X-Window fonts

Another non-trivial point is fonts. XWindow supports both bitmaps and scalable fonts. In the latter case, it's possible to use so-called font servers to remotely render scalable fonts to bitmaps, which was useful for low-level X terminals.

Classic XWindow fonts are handled by utilities: '`xfontsel`', '`xfd`' and '`xlsfonts`'.

```
xfontsel
```

In the font specification, we see the manufacturer's name, type, variety, size, resolution, encoding, and so on:
`-adobe-courier-medium-?-normal-10-100-75-75-m-60-iso8859-*`

Font names tend to be fairly long as they contain all of the information needed to uniquely identify individual fonts. However, the X server supports wildcarding of font names, so the full specification.

This is good, but not good enough for the modern WYSIWYG world. The standard XWindow paradigm knows nothing about presentation on paper, only on screen. All documents are executed by applications creating PostScript language output for high quality printing.

And with the widespread distribution of office suites, this paradigm turns out to be insufficient. For the modern WYSIWYG graphical interfaces, a new font engine has been developed — FontConfig, which works with PostScript and TrueType fonts and is processed by utilities: `fc-cache`, `fc-list`, `fc-cat` and `fc-match`.

*

X-Window options

Classic XWindow applications are built using the XToolkit library and generally support a standard set of options:

- `-display` — name of the X server to use
- `-geometry` — initial size and location of the window
- `-title` — window title
- `-bg`, `-background`, `-fg color`, `-foreground` — window background/-foreground color
- `-fn`, `-font` — font to use for displaying text
- `-name` — name under which resources for the application should be found
- `-xrm` — resource name and value to override any defaults

X-Window resources

Finally, the so-called resources described in the manual pages for such applications can be used to customize the default settings for XWindow applications. Resources must be located in the '`.Xdefaults`' or '`.Xresources`' file in the $HOME directory and can be processed by the '`xrdb`' utility on the fly.

The description looks like this:
`obj.subobj[.subobj].attr:  value`
And in XWindow, the object-oriented paradigm was implemented even before it was implemented in well-known programming languages. Program components are named in a hierarchical fashion, with each node in the hierarchy identified by a class and an instance name. At the top level is the class and instance name of the application itself. By convention, the class name of the application is the same as the program name, but with the first letter capitalized:

- `Obj` — class name
- `obj` — instance name

Some examples:

```
XTerm*Font: 6x10
emacs*Background: rgb:5b/76/86
```

GNOME user interface uses its own resource management — Gconf (`gconf-editor`, `gconftool-2`).

Xserver

OK. Let's look to some classical XWindow applications.

The first one as we know is a X server:

```
man X
```

Most important options:

- `:displaynumber` — default is 0
- `-fp fontPath` — search path for fonts
- `-s minutes` — screen-saver timeout time in minutes

And some options that can help organize a local XWindow network with low-cost X terminals and application servers:

- `-query hostname` — enables XDMCP and sends Query packets to the specified hostname on which this or that display manager is running;
- `-broadcast` — enables XDMCP and broadcasts BroadcastQuery packets to the network. In this way, simple load balancing between application servers can be organized.
- `-indirect hostname` — enables XDMCP and send IndirectQuery packets to the specified hostname. In this case, you will see a list of available application servers that you can select.

Xserver settings

After starting the X-server, it is possible to change some parameters on the fly by '`xset`' command:

```
man xset - user preference utility for X
```

Options:
`-display display` — set display
`q` — current settings
`[+|-]fp[+|-|=] path,...` — set the font path for X-server, including font-server
`fp default` — font path to be reset to the server's default.
`fp rehash` — reset the font path to its current value (server reread the font databases in the current font path)
`p` — pixel color values
`s` — screen saver parameters

X-Window utiltities

As we discussed earlier, the main principle of the X Window System is "Provide a mechanism, not a policy." And the look and feel in X Window can be anything — it is simply determined by the set of widgets on which a particular application is built. It is not a paradox, but the appearance of the original XWindow applications may seem a little odd to modern users, as they are based on an ancient set of widgets from the Athena project. It looks "ugly" at now days, but they were often used in the period of X history that he describes as the "GUI wars", as a safe alternative to the competing Motif and Open Look toolkits.

Let's look at the well-known for us '`xterm`' application:

```
xterm
```

mverb

As we can see, these are very simple 2D graphics with very unusual scrollbar behavior, which often discourages new users. The general abstraction of a mouse pointer in an XWindow is a three-button device. If you have a mouse with fewer buttons, the middle button is emulated, for example, by

simultaneously pressing the left and right buttons. So here: pressing the left button on the scroll bar scrolls forward, the right button backward, and the middle button scrolls to the selected position.

Yet another classic XWindow utilities:

`xkill` — kill a client by its X resource

`xdpyinfo` — display information utility for X

`xwininfo` — window information utility for X

`xlsclients` — list client applications running on a display

`showrgb` — display an rgb color-name database

`appres` — list X application resource database

`xrdb` — X server resource database utility

`editres` — a dynamic resource editor for X Toolkit applications

`xsetroot` — root window parameter setting utility for X

`xev` — print contents of X events

`xmodmap` — utility for modifying keymaps and pointer button mappings in X

`setxkbmap` — set the keyboard using the X Keyboard Extension

`xrefresh` — refresh all or part of an X screen

and others.

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
```