UNIX AND LINIX IN INFOCOMMUNICATION Week 8

O. Sadov

Archiving and backups

Historically the archiver, also known simply as '`ar`', is a first Unix utility developed at 1971 in AT&T that maintains groups of files as a single archive file. Today, ar is generally used only to create and update static library files that the link editor or linker uses. An implementation of '`ar`' is included as one of the GNU Binutils.

But the most widely used archiving tools are '`tar`' and '`cpio`'.

The '`tar`' is a "tape archiver" originally developed in AT&T at end of 70s for storing data on magnetic tape. It saves many files together into a single tape or disk archive, and can restore individual files from the archive:

```
man tar
```

Basic operations with a tar archive:

- '`c`' — create archive
- '`t`' — list files in archive and
- '`x`' — extract files from archive

Useful options:

- '`v`' — verbose
- '`f`' — file or device file with archive.
- '`-`' — "Dash" means standard input or output.

  The GNU version also supports compressing/decompressing archives:

- '`-a, -auto-compress`' — use archive suffix to determine the compression program
- '`-z, -gzip, -gunzip, -ungzip`'
- '`-Z, -compress, -uncompress`'
- '`-j, -bzip2`'

- ‘-J, -xz’

```
tar cvzf arch.tar.gz some_files...
tar tvzf arch.tar.gz
tar xvzf arch.tar.gz some_file
```

For standard UNIX ‘tar’, external compression/decompression programs should be used:

```
tar cvf - some_files... | gzip -c > arch.tar.gz
gunzip -c arch.tar.gz | tar tvf -
gunzip -c arch.tar.gz | tar xvf - some_file
```

The main problem with compressed archives is if you have corruption in the middle of the archive file, you will lose all content from the tail. Just because this format is focused on storing on tape and all the metadata about files stored sequentially, not in some central directory. And if you want to improve the reliability of your data, it makes sense to compress the files separately and put them in an uncompressed archive.

Another widely used archiving tool is ‘cpio’:

```
man cpio
```

Basic operations with it:

- -o|-create
- -t|-list: Print a table of contents of the input.
- -i|-extract
- also -p|-pass-through is so-called copy-pass mode, cpio copies files from one directory tree to another, combining the copy-out and copy-in steps without actually using an archive.

Unlike ‘tar’, which works with files, ‘cpio’ works with stdin/stdout.

This is good, but such an archive may contain some special files that are not properly processed. For example, you can get hard links split across multiple files. And for real backup in UNIX-like systems, special programs have been developed that know about the internal structure of the file system, for example:

- dump/restore — ext2/3/4 filesystem backup/restore.
- xfsdump/xfsrestore — XFS filesystems backup/retore. The main arguments are: the list of files and directories for dump and '`-f dump_file`'. But we can also choose the "`dump level`", which is just an integer. A level 0, full backup, specified by -0 guarantees the entire file system is copied. A level number above 0, is so-called "incremental backup", tells '`dump`' to copy all files new or modified since the last dump of a lower level. The default level is 0.

And this makes it possible to implement various "backup strategies". For example, you can create a full backup at the end of the week and then make incremental backups every day of the week. Then at the end of the week for new full backup, you can use the oldest backup storage from the full backup pool. This way, you will have weekly full backups for a specific period and daily saved states in incremental backups for a week or two.

And then you can extract the full dump or individual files or directories from the saved dump using the restore utility:

```
man restore
```

You can also do this interactively (`-i`).

Software installation

Well. Now let's talk about installing software. An initial set of software is installed during system installation, and installed software packages are updated by system services when newer versions are published to the distribution sites. But let's take a deeper look.

Diomidis Spinellis, "Evolution of the Unix System Architecture: An Exploratory Case Study"

From the First Research Edition (November 3, 1971) in which the PDP-7 Unix was rewritten on the PDP-11 processor, UNIX documented the "User Maintained Programs" guidelines by organizing third-party code as so-called "packages" or "ports".

The two first Berkeley distributions introduced to the user community third-party software packages targeting Unix. Over the years packages proliferated and got distributed, initially through USENET newsgroups and later over the internet in the form of ports to a specific operating system distribution. The established filesystem directory hierarchy, provided a template for laying out the source code, the documentation, and the manual pages. In addition, the use of '`make`' utility provided a common way for expressing compilation and deployment rules.

And now if we are talking about free and open source, the most general way is compiling from source. Many projects simply require downloading the source, running the configure script, and running '`make install`'. This command reads an instruction from a file named "`makefile`" and installs the software into the target directory, by default '`/usr/local`'. You can change this and other settings during configuration.

Sounds good, but in most cases we may have problems uninstalling and updating installed software, because in most cases such actions may not be so simple, and the purpose of "uninstall" is not implemented in the Makefiles. And to perform a complete set of actions with the software, a special type of files called software packages and software were developed to manage them. They differ on different systems and distributions.

Repositories and packages

BSD UNIX packaging has been extended to the FreeBSD 'ports' machinery, which provides a mechanism for compiling and installing third-party packages and their dependencies. The main package Linux utilities are '`rpm`' (RPM Package Manager) for RH-like systems and '`dpkg`' for Debian-based distributions:

```
man rpm
man dpkg
```

A package is a file that you can install, remove, and update. But when we have many packages with a complex system of dependencies between them, it can be too difficult to handle the full set of dependent packages during package manipulation.

To solve this problem, so-called repositories have been developed, which collect many packages, resolve dependencies between them and put information about this in metadata files.Such repositories are hosted on the servers of the respective projects, and we can access them via the Internet.

The main tools for working with repositories are:

- '`yum`' — Yellowdog Updater Modified for RPM repositories, now replaced with '`dnf`' package manager.
- '`apt`' — package manager for Debian-based repositories. As we can see, using these tools, we can perform the same actions as with '`rpm`' and '`dpkg`': install, remove and update packages with dependencies. We can also get information about packages and package groups in the repositories, get a list of packages, search for packages by name or by files included in the package.

Also may be useful '`yumdownloader`' — program for downloading RPMs from Yum repositories.

You can find descriptions of the repositories in:

- `/etc/yum.repos.d/` —
- `/etc/apt/sources.list`

You can see the Table of equivalent commands for package management on both Ubuntu/Debian and Red Hat/Fedora systems

Under the hood — Devices and drivers

CPAN, PyPi, NPM, static binaries, docker containers

Booting and services starting/stopping

All right. But how does our system get started? In fact, when you turn on the computer, a special piece of code is launched, built into the hardware – firmware. There are many such firmware: legacy PC BIOS, UEFI, Uboot, OpenBoot, Coreboot, etc. The firmware reads the main bootloader from disk: BIOS from MBR, UEFI from EFI system partition, and so on.

Then the secondary bootloader started. This loader can be more or less complex and customizable. The most commonly used Linux bootloaders currently are lightweight boot system `SYSLINUX` for FAT file system and `ISOLINUX` for ISO images, which is mainly used to boot installation or live images, and the general purpose Grub bootloader.

Usually Grub is installed during system installation, including configuring the boot of other operating systems installed on your computer. But in some cases MS Windows can reinstall the bootloader without asking you, in which case you may lose your boot settings. To restore it, you need to boot from the installation media in repair mode and run the '`grub2-install`' program for your system storage. For example:

```
grub2-install /dev/sda
```

After the kernel is loaded, a special process called '`init`' is started. In original UNIX, as well as BSD '`init`', just run the script '`/etc/rc`', which completely determines the further behavior of the system.

A different '`init`' machinery is implemented for SYSV systems. On such systems, you can invoke the '`init`' program at any time by setting the run-level as a parameter. Runlevels defined in the '`init`' configuration is located in the '`/etc/inittab`' file. Each line in the `inittab` file consists of four colon-delimited fields and describes:

- what processes to start, monitor, and restart if they terminate
- what actions to take when the system enters a new runlevel
- the default runlevel

Inittab's fields:

```
id:runlevels:action:process
```

id (identification code) — consists of a sequence of one to four characters that identifies its function.

- runlevels — lists the run levels to which this entry applies.
- action — specific codes in this field tell init how to treat the process. Possible values include: initdefault, sysinit, boot, bootwait, wait, and respawn.

- process — defines the command or script to execute.

The line '`initdefault`' defines the default runlevel. Different systems define different init level hierarchies, but some of them have the same meaning:

- Runlevel 0 is halt.
- Runlevel 6 is reboot.
- Runlevel 1 is single-user.
- 2–5 are most often some multiuser runlevels.

Most often, the executable scripts in '`inittab`' are just some '`rc`' scripts that go through the appropriate `/etc/rc<runlevel>.d/` directories and run the stop scripts first, starting with a big `K` (kill) with a '`stop`' parameter, and then starting the scripts that start with large `S` with a '`start`' parameter:

```
$ ls -l /etc/rc.d/rc5.d/
```

And, as we can see, this script is simply symbolic links to scripts from '`/etc/init.d/`', moreover, the Kill scripts and the Start scripts can be linked to the same file. If we look at them, we will see — there are just scripts that do something according to the start or stop parameter:

```
$ less /etc/init.d/network
```

And if you want to implement our own service script, you just have to support the '`start`' and '`stop`' parameters. To configure your own policy for stopping and starting services at any level, you simply have to link the scripts which you need from '`/etc/init.d/`' to the appropriate runlevel directories. The order in which scripts are run is determined by the numbers at the beginning of the filenames.

Some commands that can help you with this work:

- '`service`' – run a System V init script
- '`setup`' and '`chkconfig`' — updates and queries runlevel information for system services

The most commonly used Linux distributions now use ‘systemd’ instead of such systems. The main advantage of this system is faster parallel launch of services at system startup, as well as unification of services and work with devices. These utilities and scripts are still present for compatibility, but now the main tool is ‘systemctl’:

```
man systemctl
```

We can list system services, start, stop and get their status, enable and disable them to automatically start at boot time.

To find out the log messages about boot startup and system operation, we can look at the system log files:

- /var/log/messages — RH-like Linuxes
- /var/log/syslog — Debian, Ubuntu

In modern Linux based on ‘systemd’ we have ‘journalctl’ to work with the ‘systemd’ journal system.

Network configuration

Finally, let’s discuss the administrative tasks associated with the network. In most cases, after installing the system, you have a more or less configured network in accordance with the DHCP settings of your local network. The most you need is to set a password for your WiFi.

But in some cases it would be helpful to have some utilities to view and manage network settings. Unlike other devices, network interfaces do not have their own representations in the device files in the /dev directory. You can work with them only with the help of special utilities. Traditional set of utilities for network configuration:

- ifconfig — configure a network interface
- route – show/manipulate the IP routing table

With no arguments, ifconfig just shows us the current state of enabled network interfaces. By pointing to a network interface, we can “up” and “down” them, we can also manually set the IP address and netmask:

```
man ifconfig
```

You can use the 'route' utility to work with the routing table. To view the route table we may run command:

```
route -n
```

The option '-n' show numerical addresses instead of trying to determine symbolic host names. This can be useful if you are having problems accessing the DNS server.

```
man route
```

With this command we can add and remove routes to hosts and networks. We can set gateways to them.

In modern Linux distributions, these 'net-tools' are outdated and are not installed by default. They are migrating to the more versatile 'ip' utility, which also supports more advanced networking options than 'net-tools':

```
man ip
```

You can use 'ip link' to perform the same tasks as 'ifconfig' and 'ip route' to replace 'route':

```
$ ip link help 2>&1 | less
$ ip route help 2>&1 | less
```

The next important task when setting up your network is setting up DNS resolving. This configuration is placed in '/etc/resolv.conf':

```
$ cat /etc/resolv.conf
```

Here we can configure up to 3 nameservers.

```
man resolv.conf
```

Also, using the 'search' configuration option, we can configure the domains in which short names will be searched.

Network access

And finally, a few words about regulating network access to your computer. As we understand, network attacks are currently the most dangerous. And the most famous tool for restricting access is a firewall.

At its deep level, the firewall in Linux is controlled by the '`iptables`' utility and moving to '`nftables`'. But at a higher level, different systems manage it in different distributions:

- Canonical's Uncomplicated Firewall ('`ufw`') to configure the iptables on every Ubuntu and Debian system I've used in recent years. The firewall isn't enabled by default in Ubuntu nor installed by default in Debian. As its name suggests, it's fairly uncomplicated to set up and maintain. It has an easy to remember syntax that's more friendly to human users than the underlying iptables rules.

- Fedora and Red Hat Enterprise Linux enables FirewallD by default. Its '`firewall-cmd`' front-end has almost the same feature set for basic firewall management as '`ufw`', and adds network zone management to the mix. Zone management allows you to set up presets with rules for different network conditions/locations. For example "Home" and "Office" where all communications with local machines are allowed, and "Public Wi-Fi" where no communication with the same subnet would be allowed. Rules can be applied automatically per network interface, or used through NetworkManager and the GNOME network GUI '`system-config-firewall`'.

Both front-ends come with pre-defined rules for common server services and protocols. These rules include a keyword/name and associated industry standards and default ports for running services publicly. They each come in their own formats that aren't interoperable with each other, of course. `ufw` uses service-named files containing one line with port and protocol, and FirewallD uses six lines of XML to create the same profile.

The best policy is to simply close all services from the Internet that you do not need on your computer, and only open those that you need for remote access. For example — SSH.

And then just use the lighter tweak tool — host access control files:

```
/etc/hosts.deny:
         ALL: ALL
/etc/hosts.allow:
         ALL: LOCAL @some_netgroup
         ALL: .foo.bar EXCEPT hacker.foo.bar
```

This is the configuration for the so-called tcp wrapper, which was originally developed as a '`tcpd`' service for the '`inetd`' superserver, and now its functionality is included in the '`libwrap`' library that is used by several network services such as NFS.

Again, the easiest way is to disable everything in the configuration file '`/etc/hosts.deny`' and allow only a fixed list of hosts to access your computer, for example, via SSH in '`/etc/hosts.allow`'. Using this mechanism is easier than using a firewall because all changes are made immediately after saving the configuration file — you don't need to reconfigure and reload any services.

Network services

And if we talk about older classical systems, then tcp_wrapper '`tcpd`' is configured through the '`inetd`' service. As we'll see, this is a so-called super-server designed to make the life of network developers easier. To create the server side of a network application, you just need to develop an application that reads stdin and writes to stdout, and write the service configuration to the '`inetd`' configuration file. All the work of listening on network sockets and maintaining the connection makes a super-server for you.

Classic services such as FTP, POP3, and telnet were designed to work with '`inetd`'. It is also possible to configure an HTTP server for access via '`inetd`'. Newer systems have replaced '`inetd`' with '`xinetd`', which provides better protection against DOS attacks, and replaced with '`systemd`' in newest systems.

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.