Chapter

16

Conclusion: Debian's Future

Contents

The story of Falcot Corp ends with this last chapter; but Debian lives on, and the future will certainly bring many interesting surprises.

16.1. Upcoming Developments

Now that Debian version 10 is out, the developers are already busy working on the next version, codenamed *Bullseye*...

There is no official list of planned changes, and Debian never makes promises relating to technical goals of the coming versions. However, a few development trends can already be noted, and we can try to guess what might happen (or not).

In order to improve security and trust, an increasing number of packages will be made to build reproducibly; that is to say, it will be possible to rebuild byte-for-byte identical binary packages from the source packages, thus allowing everyone to verify that no tampering has happened during the builds. This feature might even be required by the release managers for testing migration.

In a related theme, a lot of effort will have gone into improving security by default, with more packages shipping an AppArmor profile.

Of course, all the main software suites will have had a major release. The latest version of the various desktops will bring better usability and new features. Wayland, the new display server, will likely obsolete X11 entirely.

With the widespread use of continuous integration and the growth of the archive (and of the biggest packages!), the constraints on release architectures will be harder to meet and some architectures will be dropped (like *mips*, *mipsel* and maybe *mips64el*).

16.2. Debian's Future

In addition to these internal developments, one can reasonably expect new Debian-based distributions to come to light, as many tools keep simplifying this task. New specialized subprojects will also be started, in order to widen Debian's reach to new horizons.

The Debian user community will increase, and new contributors will join the project... including, maybe, you!

There are recurring discussions about how the software ecosystem is evolving, towards applications shipped within containers, where Debian packages have no added value, or with language-specific package managers (e.g. `pip` for Python, `npm` for JavaScript, etc.), which are rendering `dpkg` and `apt` obsolete. Facing those threats, I am convinced that Debian developers will find ways to embrace those evolutions and to continue to provide value to users.

In spite of its old age and its respectable size, Debian keeps on growing in all kinds of (sometimes unexpected) directions. Contributors are teeming with ideas, and discussions on development mailing lists, even when they look like bickerings, keep increasing the momentum. Debian is sometimes compared to a black hole, of such density that any new free software project is attracted.

Beyond the apparent satisfaction of most Debian users, a deep trend is becoming more and more indisputable: people are increasingly realizing that collaborating, rather than working alone in their corner, leads to better results for everyone. Such is the rationale used by distributions merging into Debian by way of subprojects.

The Debian project is therefore not threatened by extinction...

16.3. Future of this Book

We would like this book to evolve in the spirit of free software. We therefore welcome contributions, remarks, suggestions, and criticism. Please direct them to Raphaël (hertzog@debian.org) or Roland (lolando@debian.org). For actionable feedback, feel free to open bug reports against the debian-handbook Debian package. The website will be used to gather all information relevant to its evolution, and you will find there information on how to contribute, in particular if you want to translate this book to make it available to an even larger public than today.

➠ https://debian-handbook.info/

We tried to integrate most of what our experience with Debian taught us, so that anyone can use this distribution and take the best advantage of it as soon as possible. We hope this book contributes to making Debian less confusing and more popular, and we welcome publicity around it!

We would like to conclude on a personal note. Writing (and translating) this book took a considerable amount of time out of our usual professional activity. Since we are both freelance consultants, any new source of income grants us the freedom to spend more time improving Debian; we hope this book to be successful and to contribute to this. In the meantime, feel free to retain our services!

➠ https://www.freexian.com

➠ http://www.gnurandal.com

See you soon!

Appendix

A Derivative Distributions

Contents

A.1. Census and Cooperation

The Debian project fully acknowledges the importance of derivative distributions and actively supports collaboration between all involved parties. This usually involves merging back the improvements initially developed by derivative distributions so that everyone can benefit and long-term maintenance work is reduced.

This explains why derivative distributions are invited to become involved in discussions on the debian-derivatives@lists.debian.org mailing-list, and to participate in the derivative census. This census aims at collecting information on work happening in a derivative so that official Debian maintainers can better track the state of their package in Debian variants.

➨ https://wiki.debian.org/DerivativesFrontDesk

➨ https://wiki.debian.org/Derivatives/Census

Let us now briefly describe the most interesting and popular derivative distributions.

A.2. Ubuntu

Ubuntu made quite a splash when it came on the free software scene, and for good reason: Canonical Ltd., the company that created this distribution, started by hiring thirty-odd Debian developers and publicly stating the far-reaching objective of providing a distribution for the

general public with a new release twice a year. They also committed to maintaining each version for a year and a half.

These objectives necessarily involve a reduction in scope; Ubuntu focuses on a smaller number of packages than Debian, and relies primarily on the GNOME desktop (although there are Ubuntu derivatives that come with other desktop environments). Everything is internationalized and made available in a great many languages.

So far, Ubuntu has managed to keep this release rhythm. They also publish *Long Term Support* (LTS) releases, with a 5-year maintenance promise. As of June 2019, the current LTS version is version 18.04, nicknamed Bionic Beaver. The last non-LTS version is version 19.04, nicknamed Disco Dingo. Version numbers describe the release date: 19.04, for example, was released in April 2019.

IN PRACTICE

Ubuntu's support and maintenance promise

Canonical has adjusted multiple times the rules governing the length of the period during which a given release is maintained. Canonical, as a company, promises to provide security updates to all the software available in the `main` and `restricted` sections of the Ubuntu archive, for 5 years for LTS releases and for 9 months for non-LTS releases. Everything else (available in the `universe` and `multiverse`) is maintained on a best-effort basis by volunteers of the MOTU team (*Masters Of The Universe*). Be prepared to handle security support yourself if you rely on packages of the latter sections.

Ubuntu has reached a wide audience in the general public. Millions of users were impressed by its ease of installation, and the work that went into making the desktop simpler to use.

Ubuntu and Debian used to have a tense relationship; Debian developers who had placed great hopes in Ubuntu contributing directly to Debian were disappointed by the difference between the Canonical marketing, which implied Ubuntu were good citizens in the Free Software world, and the actual practice where they simply made public the changes they applied to Debian packages. Things have been getting better over the years, and Ubuntu has now made it general practice to forward patches to the most appropriate place (although this only applies to external software they package and not to the Ubuntu-specific software such as Mir or Unity).

➨ `https://www.ubuntu.com/`

A.3. Linux Mint

Linux Mint is a (partly) community-maintained distribution, supported by donations and advertisements. Their flagship product is based on Ubuntu, but they also provide a "Linux Mint Debian Edition" variant that evolves continuously (as it is based on Debian Testing). In both cases, the initial installation involves booting a live DVD or a live USB storage device.

The distribution aims at simplifying access to advanced technologies, and provides specific graphical user interfaces on top of the usual software. For instance, Linux Mint relies on Cinnamon instead of GNOME by default (but it also includes MATE as well as Xfce); similarly, the

package management interface, although based on APT, provides a specific interface with an evaluation of the risk from each package update.

Linux Mint includes a large amount of proprietary software to improve the experience of users who might need those. For example: Adobe Flash and multimedia codecs.

➨ https://linuxmint.com/

A.4. Knoppix

The Knoppix distribution barely needs an introduction. It was the first popular distribution to provide a *live CD*; in other words, a bootable CD-ROM that runs a turn-key Linux system with no requirement for a hard-disk — any system already installed on the machine will be left untouched. Automatic detection of available devices allows this distribution to work in most hardware configurations. The CD-ROM includes almost 2 GB of (compressed) software, and the DVD-ROM version has even more.

Combining this CD-ROM to a USB stick allows carrying your files with you, and to work on any computer without leaving a trace — remember that the distribution doesn't use the hard-disk at all. Knoppix uses LXDE (a lightweight graphical desktop) by default, but the DVD version also includes GNOME and Plasma. Many other distributions provide other combinations of desktops and software. This is, in part, made possible thanks to the *live-build* Debian package that makes it relatively easy to create a live CD.

➨ https://live-team.pages.debian.net/live-manual/

Note that Knoppix also provides an installer: you can first try the distribution as a live CD, then install it on a hard-disk to get better performance.

➨ https://www.knopper.net/knoppix/index-en.html

A.5. Aptosid and Siduction

These community-based distributions track the changes in Debian *Sid* (*Unstable*) — hence their name. The modifications are limited in scope: the goal is to provide the most recent software and to update drivers for the most recent hardware, while still allowing users to switch back to the official Debian distribution at any time. Aptosid was previously known as Sidux, and Siduction is a more recent fork of Aptosid.

➨ http://aptosid.com

➨ https://siduction.org

A.6. Grml

Grml is a live CD with many tools for system administrators, dealing with installation, deployment, and system rescue. The live CD is provided in two flavors, full and small, both available for 32-bit and 64-bit PCs. Obviously, the two flavors differ by the amount of software included and by the resulting size.

➨ https://grml.org

A.7. Tails

Tails (The Amnesic Incognito Live System) aims at providing a live system that preserves anonymity and privacy. It takes great care in not leaving any trace on the computer it runs on, and uses the Tor network to connect to the Internet in the most anonymous way possible.

➨ https://tails.boum.org

A.8. Kali Linux

Kali Linux is a Debian-based distribution specializing in penetration testing ("pentesting" for short). It provides software that helps auditing the security of an existing network or computer while it is live, and analyze it after an attack (which is known as "computer forensics").

➨ https://kali.org

A.9. Devuan

Devuan is a fork of Debian started in 2014 as a reaction to the decision made by Debian to switch to `systemd` as the default init system. A group of users attached to `sysv` and opposing drawbacks to `systemd` started Devuan with the objective of maintaining a `systemd`-less system.

➨ https://devuan.org

A.10. DoudouLinux

DoudouLinux targets young children (starting from 2 years old). To achieve this goal, it provides a heavily customized graphical interface (based on LXDE) and comes with many games and educative applications. Internet access is filtered to prevent children from visiting problematic websites. Advertisements are blocked. The goal is that parents should be free to let their children use their computer once booted into DoudouLinux. And children should love using DoudouLinux, just like they enjoy their gaming console.

➨ https://www.doudoulinux.org

A.11. Raspbian

Raspbian is a rebuild of Debian optimized for the popular (and inexpensive) Raspberry Pi family of single-board computers. The hardware for that platform is more powerful than what the Debian *armel* architecture can take advantage of, but lacks some features that would be required for *armhf*; so Raspbian is a kind of intermediary, rebuilt specifically for that hardware and including patches targeting this computer only.

➨ https://raspbian.org

A.12. PureOS

PureOS is a Debian-based distribution focused on privacy, convenience and security. It follows the GNU Free System Distribution Guidelines[1], used by the Free Software Foundation to qualify a distribution as free. The social purpose company Purism guides its development.

➨ https://pureos.net/

A.13. SteamOS

SteamOS is a gaming-oriented Debian-based distribution developed by Valve Corporation. It is used in the Steam Machine, a line of gaming computers.

➨ https://store.steampowered.com/steamos/

A.14. And Many More

The Distrowatch website references a huge number of Linux distributions, many of which are based on Debian. Browsing this site is a great way to get a sense of the diversity in the free software world.

➨ https://distrowatch.com

The search form can help track down a distribution based on its ancestry. In June 2019, selecting Debian led to 127 active distributions!

➨ https://distrowatch.com/search.php

[1] https://www.gnu.org/distros/free-system-distribution-guidelines.html

Appendix

B Short Remedial Course

Contents

B.1. Shell and Basic Commands

In the Unix world, every administrator has to use the command line sooner or later; for example, when the system fails to start properly and only provides a command-line rescue mode. Being able to handle such an interface, therefore, is a basic survival skill for these circumstances.

QUICK LOOK

Starting the command interpreter

A command-line environment can be run from the graphical desktop, by an application known as a "terminal". In GNOME, you can start it from the "Activities" overview (that you get when you move the mouse in the top-left corner of the screen) by typing the first letters of the application name. In Plasma, you will find it in the K Applications System menu.

This section only gives a quick peek at the commands. They all have many options not described here, so please refer to the abundant documentation in their respective manual pages.

B.1.1. Browsing the Directory Tree and Managing Files

Once a session is open, the `pwd` command (which stands for *print working directory*) displays the current location in the filesystem. The current directory is changed with the `cd` *`directory`* command (`cd` is for *change directory*). The parent directory is always called .. (two dots), whereas the current directory is also known as . (one dot). The `ls` command allows *listing* the contents of a directory. If no parameters are given, it operates on the current directory.

```
$ pwd
/home/rhertzog
$ cd Desktop
$ pwd
/home/rhertzog/Desktop
$ cd .
$ pwd
/home/rhertzog/Desktop
$ cd ..
$ pwd
/home/rhertzog
$ ls
Desktop    Downloads  Pictures  Templates
Documents  Music      Public    Videos
```

A new directory can be created with `mkdir` *`directory`*, and an existing (empty) directory can be removed with `rmdir` *`directory`*. The `mv` command allows *moving* and/or renaming files and directories; *removing* a file is achieved with `rm` *`file`*.

```
$ mkdir test
$ ls
Desktop    Downloads  Pictures  Templates  Videos
Documents  Music      Public    test
$ mv test new
$ ls
Desktop    Downloads  new       Public     Videos
Documents  Music      Pictures  Templates
$ rmdir new
$ ls
Desktop    Downloads  Pictures  Templates  Videos
Documents  Music      Public
```

B.1.2. Displaying and Modifying Text Files

The `cat` *`file`* command (intended to *concatenate* files to the standard output device) reads a file and displays its contents on the terminal. If the file is too big to fit on a screen, use a pager such as `less` (or `more`) to display it page by page.

The `editor` command starts a text editor (such as `vi` or `nano`) and allows creating, modifying and reading text files. The simplest files can sometimes be created directly from the command interpreter thanks to redirection: `echo` *`"text" >file`* creates a file named *file* with "*text*" as its contents. Adding a line at the end of this file is possible too, with a command such as `echo` *`"moretext" >>file`*. Note the `>>` in this example.

B.1.3. Searching for Files and within Files

The `find` *`directory criteria`* command looks for files in the hierarchy under *directory* according to several criteria. The most commonly used criterion is -name *name*: that allows looking for a file by its name.

The `grep` *`expression files`* command searches the contents of the files and extracts the lines matching the regular expression (see sidebar "Regular expression" page 283). Adding the -r option enables a recursive search on all files contained in the directory passed as a parameter. This allows looking for a file when only a part of the contents are known.

B.1.4. Managing Processes

The `ps aux` command lists the processes currently running and helps identifying them by showing their *pid* (process id). Once the *pid* of a process is known, the `kill -`*`signal pid`* command allows sending it a signal (if the process belongs to the current user). Several signals exist; most commonly used are TERM (a request to terminate gracefully) and KILL (a forced kill).

The command interpreter can also run programs in the background if the command is followed by a "&". By using the ampersand, the user resumes control of the shell immediately even though the command is still running (hidden from the user; as a background process). The `jobs` command lists the processes running in the background; running `fg %`*`job-number`* (for *foreground*) restores a job to the foreground. When a command is running in the foreground (either because it was started normally, or brought back to the foreground with `fg`), the Control+Z key combination pauses the process and resumes control of the command-line. The process can then be restarted in the background with `bg %`*`job-number`* (for *background*).

B.1.5. System Information: Memory, Disk Space, Identity

The `free` command displays information on memory; `df` (*disk free*) reports on the available disk space on each of the disks mounted in the filesystem. Its -h option (for *human readable*) converts the sizes into a more legible unit (usually mebibytes or gibibytes). In a similar fashion, the `free` command supports the -m and -g options, and displays its data either in mebibytes or in gibibytes, respectively.

```
$ free
              total        used        free      shared  buff/cache   available
Mem:       16279260     5910248      523432      871036     9845580     9128964
Swap:      16601084      240640    16360444
$ df
Filesystem                1K-blocks      Used Available Use% Mounted on
udev                        8108516         0   8108516   0% /dev
tmpfs                       1627928    161800   1466128  10% /run
/dev/mapper/vg_main-root  466644576 451332520  12919912  98% /
tmpfs                       8139628    146796   7992832   2% /dev/shm
tmpfs                          5120         4      5116   1% /run/lock
```

```
tmpfs                    8139628         0   8139628   0% /sys/fs/cgroup
/dev/sda1                 523248      1676    521572   1% /boot/efi
tmpfs                    1627924        88   1627836   1% /run/user/1000
```

The `id` command displays the identity of the user running the session, along with the list of groups they belong to. Since access to some files or devices may be limited to group members, checking available group membership may be useful.

```
$ id
uid=1000(rhertzog) gid=1000(rhertzog) groups=1000(rhertzog),24(cdrom),25(floppy),27(
       sudo),29(audio),30(dip),44(video),46(plugdev),108(netdev),109(bluetooth),115(
       scanner)
```

B.2. Organization of the Filesystem Hierarchy

B.2.1. The Root Directory

A Debian system is organized along the *Filesystem Hierarchy Standard* (FHS). This standard defines the purpose of each directory. For instance, the top-level directories are described as follows:

- `/bin/`: basic programs;
- `/boot/`: Linux kernel and other files required for its early boot process;
- `/dev/`: device files;
- `/etc/`: configuration files;
- `/home/`: user's personal files;
- `/lib/`: basic libraries;
- `/media/*`: mount points for removable devices (CD-ROM, USB keys and so on);
- `/mnt/`: temporary mount point;
- `/opt/`: extra applications provided by third parties;
- `/root/`: administrator's (root's) personal files;
- `/run/`: volatile runtime data that does not persist across reboots;
- `/sbin/`: system programs;
- `/srv/`: data used by servers hosted on this system;
- `/tmp/`: temporary files; this directory is often emptied at boot;
- `/usr/`: applications; this directory is further subdivided into `bin`, `sbin`, `lib` (according to the same logic as in the root directory). Furthermore, `/usr/share/` contains architecture-independent data. `/usr/local/` is meant to be used by the administrator for installing applications manually without overwriting files handled by the packaging system (`dpkg`).

- `/var/`: variable data handled by daemons. This includes log files, queues, spools, caches and so on.

- `/proc/` and `/sys/` are specific to the Linux kernel (and not part of the FHS). They are used by the kernel for exporting data to user space (see section B.3.4, "The User Space" page 482 and section B.5, "The User Space" page 485 for explanations about this concept).

Note that many modern distributions, Debian included, are shipping `/bin`, `/sbin` and `/lib` as symlinks to the corresponding directories below `/usr` so that all programs and libraries are available in a single tree. It makes it easier to protect the integrity of the system files, and to share those system files among multiple containers, etc.

B.2.2. The User's Home Directory

The contents of a user's home directory is not standardized, but there are still a few noteworthy conventions. One is that a user's home directory is often referred to by a tilde ("~"). That is useful to know because command interpreters automatically replace a tilde with the correct directory (usually `/home/`*`user`*`/`).

Traditionally, application configuration files are often stored directly under the user's home directory, but their names usually start with a dot (for instance, the `mutt` email client stores its configuration in `~/.muttrc`). Note that filenames that start with a dot are hidden by default; and `ls` only lists them when the -a option is used, and graphical file managers need to be told to display hidden files.

Some programs also use multiple configuration files organized in one directory (for instance, `~/.ssh/`). Some applications (such as Firefox) also use their directory to store a cache of downloaded data. This means that those directories can end up using a lot of disk space.

These configuration files stored directly in a user's home directory, often collectively referred to as *dotfiles*, have long proliferated to the point that these directories can be quite cluttered with them. Fortunately, an effort led collectively under the FreeDesktop.org umbrella has resulted in the "XDG Base Directory Specification", a convention that aims at cleaning up these files and directory. This specification states that configuration files should be stored under `~/.config`, cache files under `~/.cache`, and application data files under `~/.local` (or subdirectories thereof). This convention is slowly gaining traction, and several applications (especially graphical ones) have started following it.

Graphical desktops usually display the contents of the `~/Desktop/` directory (or whatever the appropriate translation is for systems not configured in English) on the desktop (i.e. what is visible on screen once all applications are closed or iconized).

Finally, the email system sometimes stores incoming emails into a `~/Mail/` directory.

B.3. Inner Workings of a Computer: the Different Layers Involved

A computer is often considered as something rather abstract, and the externally visible interface is much simpler than its internal complexity. Such complexity comes in part from the number of pieces involved. However, these pieces can be viewed in layers, where a layer only interacts with those immediately above or below.

An end-user can get by without knowing these details... as long as everything works. When confronting a problem such as, "The internet doesn't work!", the first thing to do is to identify in which layer the problem originates. Is the network card (hardware) working? Is it recognized by the computer? Does the Linux kernel see it? Are the network parameters properly configured? All these questions isolate an appropriate layer and focus on a potential source of the problem.

B.3.1. The Deepest Layer: the Hardware

Let us start with a basic reminder that a computer is, first and foremost, a set of hardware elements. There is generally a main board (known as the *motherboard*), with one (or more) processor(s), some RAM, device controllers, and extension slots for option boards (for other device controllers). Most noteworthy among these controllers are IDE (Parallel ATA), SCSI and Serial ATA, for connecting to storage devices such as hard disks. Other controllers include USB, which is able to host a great variety of devices (ranging from webcams to thermometers, from keyboards to home automation systems) and IEEE 1394 (Firewire). These controllers often allow connecting several devices so the complete subsystem handled by a controller is therefore usually known as a "bus". Option boards include graphics cards (into which monitor screens will be plugged), sound cards, network interface cards, and so on. Some main boards are pre-built with these features, and don't need option boards.

IN PRACTICE

Checking that the hardware works

Checking that a piece of hardware works can be tricky. On the other hand, proving that it doesn't work is sometimes quite simple.

A hard disk drive is made of spinning platters and moving magnetic heads. When a hard disk is powered up, the platter motor makes a characteristic whir. It also dissipates energy as heat. Consequently, a hard disk drive that stays cold and silent when powered up is broken.

Network cards often include LEDs displaying the state of the link. If a cable is plugged in and leads to a working network hub or switch, at least one LED will be on. If no LED lights up, either the card itself, the network device, or the cable between them, is faulty. The next step is therefore testing each component individually.

Some option boards — especially 3D video cards — include cooling devices, such as heat sinks and/or fans. If the fan does not spin even though the card is powered up, a plausible explanation is the card overheated. This also applies to the main processor(s) located on the main board.

B.3.2. The Starter: the BIOS or UEFI

Hardware, on its own, is unable to perform useful tasks without a corresponding piece of software driving it. Controlling and interacting with the hardware is the purpose of the operating system and applications. These, in turn, require functional hardware to run.

This symbiosis between hardware and software does not happen on its own. When the computer is first powered up, some initial setup is required. This role is assumed by the BIOS or UEFI, a piece of software embedded into the main board that runs automatically upon power-up. Its primary task is searching for software it can hand over control to. Usually, in the BIOS case, this involves looking for the first hard disk with a boot sector (also known as the *master boot record* or MBR), loading that boot sector, and running it. From then on, the BIOS is usually not involved (until the next boot). In the case of UEFI, the process involves scanning disks to find a dedicated EFI partition containing further EFI applications to execute.

TOOL

Setup, the BIOS/UEFI configuration tool

The BIOS/UEFI also contains a piece of software called Setup, designed to allow configuring aspects of the computer. In particular, it allows choosing which boot device is preferred (for instance, you can select an USB key or a CD-ROM drive instead of the default harddisk), setting the system clock, and so on. Starting Setup usually involves pressing a key very soon after the computer is powered on. This key is often Del or Esc, sometimes F2 or F10. Most of the time, the choice is flashed on screen while booting.

The boot sector (or the EFI partition), in turn, contains another piece of software, called the bootloader, whose purpose is to find and run an operating system. Since this bootloader is not embedded in the main board but loaded from disk, it can be smarter than the BIOS, which explains why the BIOS does not load the operating system by itself. For instance, the bootloader (often GRUB on Linux systems) can list the available operating systems and ask the user to choose one. Usually, a time-out and default choice is provided. Sometimes the user can also choose to add parameters to pass to the kernel, and so on. Eventually, a kernel is found, loaded into memory, and executed.

NOTE

UEFI, a modern replacement to the BIOS

Most new computers will boot in UEFI mode by default, but usually they also support BIOS booting alongside for backwards compatibility with operating systems that are not ready to exploit UEFI.

This new system gets rid of some of the limitations of BIOS booting: with the usage of a dedicated partition, the bootloaders no longer need special tricks to fit in a tiny *master boot record* and then discover the kernel to boot. Even better, with a suitably built Linux kernel, UEFI can directly boot the kernel without any intermediary bootloader. UEFI is also the basic foundation used to deliver *Secure Boot*, a technology ensuring that you run only software validated by your operating system vendor.

The BIOS/UEFI is also in charge of detecting and initializing a number of devices. Obviously, this includes the IDE/SATA devices (usually hard disk(s) and CD/DVD-ROM drives), but also PCI

devices. Detected devices are often listed on screen during the boot process. If this list goes by too fast, use the Pause key to freeze it for long enough to read. Installed PCI devices that don't appear are a bad omen. At worst, the device is faulty. At best, it is merely incompatible with the current version of the BIOS or main board. PCI specifications evolve, and old main boards are not guaranteed to handle newer PCI devices.

B.3.3. The Kernel

Both the BIOS/UEFI and the bootloader only run for a few seconds each; now we are getting to the first piece of software that runs for a longer time, the operating system kernel. This kernel assumes the role of a conductor in an orchestra, and ensures coordination between hardware and software. This role involves several tasks including: driving hardware, managing processes, users and permissions, the filesystem, and so on. The kernel provides a common base to all other programs on the system.

B.3.4. The User Space

Although everything that happens outside of the kernel can be lumped together under "user space", we can still separate it into software layers. However, their interactions are more complex than before, and the classifications may not be as simple. An application commonly uses libraries, which in turn involve the kernel, but the communications can also involve other programs, or even many libraries calling each other.

B.4. Some Tasks Handled by the Kernel

B.4.1. Driving the Hardware

The kernel is, first and foremost, tasked with controlling the hardware parts, detecting them, switching them on when the computer is powered on, and so on. It also makes them available to higher-level software with a simplified programming interface, so applications can take advantage of devices without having to worry about details such as which extension slot the option board is plugged into. The programming interface also provides an abstraction layer; this allows video-conferencing software, for example, to use a webcam independently of its make and model. The software can just use the *Video for Linux* (V4L) interface, and the kernel translates the function calls of this interface into the actual hardware commands needed by the specific webcam in use.

The kernel exports many details about detected hardware through the `/proc/` and `/sys/` virtual filesystems. Several tools summarize those details. Among them, `lspci` (in the *pciutils* package) lists PCI devices, `lsusb` (in the *usbutils* package) lists USB devices, and `lspcmcia` (in the *pcmciautils* package) lists PCMCIA cards. These tools are very useful for identifying the exact model of a device. This identification also allows more precise searches on the web, which in turn, lead to more relevant documents.

Example B.1 *Example of information provided by `lspci` and `lsusb`*

```
$ lspci
[...]
00:02.1 Display controller: Intel Corporation Mobile 915GM/GMS/910GML Express
       Graphics Controller (rev 03)
00:1c.0 PCI bridge: Intel Corporation 82801FB/FBM/FR/FW/FRW (ICH6 Family) PCI Express
        Port 1 (rev 03)
00:1d.0 USB Controller: Intel Corporation 82801FB/FBM/FR/FW/FRW (ICH6 Family) USB
       UHCI #1 (rev 03)
[...]
01:00.0 Ethernet controller: Broadcom Corporation NetXtreme BCM5751 Gigabit Ethernet
       PCI Express (rev 01)
02:03.0 Network controller: Intel Corporation PRO/Wireless 2200BG Network Connection
       (rev 05)
$ lsusb
Bus 005 Device 004: ID 413c:a005 Dell Computer Corp.
Bus 005 Device 008: ID 413c:9001 Dell Computer Corp.
Bus 005 Device 007: ID 045e:00dd Microsoft Corp.
Bus 005 Device 006: ID 046d:c03d Logitech, Inc.
[...]
Bus 002 Device 004: ID 413c:8103 Dell Computer Corp. Wireless 350 Bluetooth
```

These programs have a -v option, that lists much more detailed (but usually not necessary) information. Finally, the `lsdev` command (in the *procinfo* package) lists communication resources used by devices.

Applications often access devices by way of special files created within `/dev/` (see sidebar "Device access permissions" page 176). These are special files that represent disk drives (for instance, `/dev/hda` and `/dev/sdc`), partitions (`/dev/hda1` or `/dev/sdc3`), mice (`/dev/input/mouse0`), keyboards (`/dev/input/event0`), soundcards (`/dev/snd/*`), serial ports (`/dev/ttyS*`), and so on.

B.4.2. Filesystems

Filesystems are one of the most prominent aspects of the kernel. Unix systems merge all the file stores into a single hierarchy, which allows users (and applications) to access data simply by knowing its location within that hierarchy.

The starting point of this hierarchical tree is called the root, `/`. This directory can contain named subdirectories. For instance, the home subdirectory of `/` is called `/home/`. This subdirectory can, in turn, contain other subdirectories, and so on. Each directory can also contain files, where the actual data will be stored. Thus, the `/home/rmas/Desktop/hello.txt` name refers to a file named hello.txt stored in the Desktop subdirectory of the rmas subdirectory of the home

directory present in the root. The kernel translates between this naming system and the actual, physical storage on a disk.

Unlike other systems, there is only one such hierarchy, and it can integrate data from several disks. One of these disks is used as the root, and the others are "mounted" on directories in the hierarchy (the Unix command is called `mount`); these other disks are then available under these "mount points". This allows storing users' home directories (traditionally stored within `/home/`) on a second hard disk, which will contain the `rhertzog` and `rmas` directories. Once the disk is mounted on `/home/`, these directories become accessible at their usual locations, and paths such as `/home/rmas/Desktop/hello.txt` keep working.

There are many filesystem formats, corresponding to many ways of physically storing data on disks. The most widely known are *ext3* and *ext4*, but others exist. For instance, *vfat* is the system that was historically used by DOS and Windows operating systems, which allows using hard disks under Debian as well as under Windows. In any case, a filesystem must be prepared on a disk before it can be mounted and this operation is known as "formatting". Commands such as `mkfs.ext3` (where `mkfs` stands for *MaKe FileSystem*) handle formatting. These commands require, as a parameter, a device file representing the partition to be formatted (for instance, `/dev/sda1`). This operation is destructive and should only be run once, except if one deliberately wishes to wipe a filesystem and start afresh.

There are also network filesystems, such as NFS, where data is not stored on a local disk. Instead, data is transmitted through the network to a server that stores and retrieves them on demand. The filesystem abstraction shields users from having to care: files remain accessible in their usual hierarchical way.

B.4.3. Shared Functions

Since a number of the same functions are used by all software, it makes sense to centralize them in the kernel. For instance, shared filesystem handling allows any application to simply open a file by name, without needing to worry where the file is stored physically. The file can be stored in several different slices on a hard disk, or split across several hard disks, or even stored on a remote file server. Shared communication functions are used by applications to exchange data independently of the way the data is transported. For instance, transport could be over any combination of local or wireless networks, or over a telephone landline.

B.4.4. Managing Processes

A process is a running instance of a program. This requires memory to store both the program itself and its operating data. The kernel is in charge of creating and tracking them. When a program runs, the kernel first sets aside some memory, then loads the executable code from the filesystem into it, and then starts the code running. It keeps information about this process, the most visible of which is an identification number known as *pid* (*process identifier*).

Unix-like kernels (including Linux), like most other modern operating systems, are capable of "multi-tasking". In other words, they allow running many processes "at the same time". There is actually only one running process at any one time, but the kernel cuts time into small slices and runs each process in turn. Since these time slices are very short (in the millisecond range), they create the illusion of processes running in parallel, although they are actually only active during some time intervals and idle the rest of the time. The kernel's job is to adjust its scheduling mechanisms to keep that illusion, while maximizing the global system performance. If the time slices are too long, the application may not appear as responsive as desired. Too short, and the system loses time switching tasks too frequently. These decisions can be tweaked with process priorities. High-priority processes will run for longer and with more frequent time slices than low-priority processes.

NOTE

Multi-processor systems (and variants)

The limitation described above of only one process being able to run at a time, doesn't always apply. The actual restriction is that there can only be one running process *per processor core* at a time. Multi-processor, multi-core or "hyper-threaded" systems allow several processes to run in parallel. The same time-slicing system is still used, though, so as to handle cases where there are more active processes than available processor cores. This is far from unusual: a basic system, even a mostly idle one, almost always has tens of running processes.

Of course, the kernel allows running several independent instances of the same program. But each can only access its own time slices and memory. Their data thus remain independent.

B.4.5. Rights Management

Unix-like systems are also multi-user. They provide a rights management system that supports separate users and groups; it also allows control over actions based on permissions. The kernel manages data for each process, allowing it to control permissions. Most of the time, a process is identified by the user who started it. That process is only permitted to take those actions available to its owner. For instance, trying to open a file requires the kernel to check the process identity against access permissions (for more details on this particular example, see section 9.3, "Managing Rights" page 213).

B.5. The User Space

"User space" refers to the runtime environment of normal (as opposed to kernel) processes. This does not necessarily mean these processes are actually started by users because a standard system normally has several "daemon" (or background) processes running before the user even opens a session. Daemon processes are also considered user-space processes.

B.5.1. Process

When the kernel gets past its initialization phase, it starts the very first process, `init`. Process #1 alone is very rarely useful by itself, and Unix-like systems run with many additional processes.

First of all, a process can clone itself (this is known as a *fork*). The kernel allocates a new (but identical) process memory space, and another process to use it. At this time, the only difference between these two processes is their *pid*. The new process is usually called a child process, and the original process whose *pid* doesn't change, is called the parent process.

Sometimes, the child process continues to lead its own life independently from its parent, with its own data copied from the parent process. In many cases, though, this child process executes another program. With a few exceptions, its memory is simply replaced by that of the new program, and execution of this new program begins. This is the mechanism used by the init process (with process number 1) to start additional services and execute the whole startup sequence. At some point, one process among `init`'s offspring starts a graphical interface for users to log in to (the actual sequence of events is described in more details in section 9.1, "System Boot" page 198).

When a process finishes the task for which it was started, it terminates. The kernel then recovers the memory assigned to this process, and stops giving it slices of running time. The parent process is told about its child process being terminated, which allows a process to wait for the completion of a task it delegated to a child process. This behavior is plainly visible in command-line interpreters (known as *shells*). When a command is typed into a shell, the prompt only comes back when the execution of the command is over. Most shells allow for running the command in the background, it is a simple matter of adding an **&** to the end of the command. The prompt is displayed again right away, which can lead to problems if the command needs to display data of its own.

B.5.2. Daemons

A "daemon" is a process started automatically by the boot sequence. It keeps running (in the background) to perform maintenance tasks or provide services to other processes. This "background task" is actually arbitrary, and does not match anything particular from the system's point of view. They are simply processes, quite similar to other processes, which run in turn when their time slice comes. The distinction is only in the human language: a process that runs with no interaction with a user (in particular, without any graphical interface) is said to be running "in the background" or "as a daemon".

VOCABULARY

Daemon, demon, a derogatory term?

Although *daemon* term shares its Greek etymology with *demon*, the former does not imply diabolical evil, instead, it should be understood as a kind of helper spirit. This distinction is subtle enough in English; it is even worse in other languages where the same word is used for both meanings.

Several such daemons are described in detail in chapter 9, "Unix Services" page 198.

An isolated process, whether a daemon or an interactive application, is rarely useful on its own, which is why there are several methods allowing separate processes to communicate together, either to exchange data or to control one another. The generic term referring to this is *inter-process communication*, or IPC for short.

The simplest IPC system is to use files. The process that wishes to send data writes it into a file (with a name known in advance), while the recipient only has to open the file and read its contents.

In the case where you do not wish to store data on disk, you can use a *pipe*, which is simply an object with two ends; bytes written in one end are readable at the other. If the ends are controlled by separate processes, this leads to a simple and convenient inter-process communication channel. Pipes can be classified into two categories: named pipes, and anonymous pipes. A named pipe is represented by an entry on the filesystem (although the transmitted data is not stored there), so both processes can open it independently if the location of the named pipe is known beforehand. In cases where the communicating processes are related (for instance, a parent and its child process), the parent process can also create an anonymous pipe before forking, and the child inherits it. Both processes will then be able to exchange data through the pipe without needing the filesystem.

IN PRACTICE

A concrete example

Let's describe in some detail what happens when a complex command (a *pipeline*) is run from a shell. We assume we have a `bash` process (the standard user shell on Debian), with *pid* 4374; into this shell, we type the command: `ls | sort`.

The shell first interprets the command typed in. In our case, it understands there are two programs (`ls` and `sort`), with a data stream flowing from one to the other (denoted by the `|` character, known as *pipe*). `bash` first creates an unnamed pipe (which initially exists only within the `bash` process itself).

Then the shell clones itself; this leads to a new `bash` process, with *pid* #4521 (*pids* are abstract numbers, and generally have no particular meaning). Process #4521 inherits the pipe, which means it is able to write in its "input" side; `bash` redirects its standard output stream to this pipe's input. Then it executes (and replaces itself with) the `ls` program, which lists the contents of the current directory. Since `ls` writes on its standard output, and this output has previously been redirected, the results are effectively sent into the pipe.

A similar operation happens for the second command: `bash` clones itself again, leading to a new `bash` process with pid #4522. Since it is also a child process of #4374, it also inherits the pipe; `bash` then connects its standard input to the pipe output, then executes (and replaces itself with) the `sort` command, which sorts its input and displays the results.

All the pieces of the puzzle are now set up: `ls` reads the current directory and writes the list of files into the pipe; `sort` reads this list, sorts it alphabetically, and displays the results. Processes numbers #4521 and #4522 then terminate, and #4374 (which was waiting for them during the operation), resumes control and displays the prompt to allow the user to type in a new command.

Not all inter-process communications are used to move data around, though. In many situations, the only information that needs to be transmitted are control messages such as "pause execution" or "resume execution". Unix (and Linux) provides a mechanism known as *signals*, through which a process can simply send a specific signal (chosen from a predefined list of signals) to another process. The only requirement is to know the *pid* of the target.

For more complex communications, there are also mechanisms allowing a process to open access, or share, part of its allocated memory to other processes. Memory now shared between them can be used to move data between the processes.

Finally, network connections can also help processes communicate; these processes can even be running on different computers, possibly thousands of kilometers apart.

It is quite standard for a typical Unix-like system to make use of all these mechanisms to various degrees.

B.5.4. Libraries

Function libraries play a crucial role in a Unix-like operating system. They are not proper programs, since they cannot be executed on their own, but collections of code fragments that can be used by standard programs. Among the common libraries, you can find:

- the standard C library (*glibc*), which contains basic functions such as ones to open files or network connections, and others facilitating interactions with the kernel;
- graphical toolkits, such as Gtk+ and Qt, allowing many programs to reuse the graphical objects they provide;
- the *libpng* library, that allows loading, interpreting and saving images in the PNG format.

Thanks to those libraries, applications can reuse existing code. Application development is simplified since many applications can reuse the same functions. With libraries often developed by different persons, the global development of the system is closer to Unix's historical philosophy.

CULTURE

The Unix Way: one thing at a time

One of the fundamental concepts that underlies the Unix family of operating systems is that each tool should only do one thing, and do it well; applications can then reuse these tools to build more advanced logic on top. This philosophy can be seen in many incarnations. Shell scripts may be the best example: they assemble complex sequences of very simple tools (such as `grep`, `wc`, `sort`, `uniq` and so on). Another implementation of this philosophy can be seen in code libraries: the *libpng* library allows reading and writing PNG images, with different options and in different ways, but it does only that; no question of including functions that display or edit images.

Moreover, these libraries are often referred to as "shared libraries", since the kernel is able to only load them into memory once, even if several processes use the same library at the same time. This allows saving memory, when compared with the opposite (hypothetical) situation where the code for a library would be loaded as many times as there are processes using it.