Linux kernel

Josep Jorba Esteve

PID_00148468

Universitat Oberta de Catalunya
www.uoc.edu

Index

Introduction

The kernel of the GNU/Linux system (which is normally called Linux) [Vasb] is the heart of the system: it is responsible for booting the system, for managing the machine's resources by managing the memory, file system, input/output, processes and intercommunication of processes.

Its origin dates back to August 1991, when a Finnish student called Linus Torvalds announced on a news list that he had created his own operating system core that worked together with the GNU project software and that he was offering it to the community of developers for testing and suggesting improvements for making it more usable. This was the origin of the operating system's kernel that would later come to be known as Linux.

One of the particular features of Linux is that following the Free Software philosophy, it offers the source code of the operating system itself (of the kernel), in a way that makes it a perfect tool for teaching about operating systems.

Note

The Linux kernel dates back to 1991, when Linus Torvalds made it available to the community. It is one of the few operating systems that while extensively used, also makes its source code available.

Another main advantage, is that by having the source code, we can compile it to adapt it better to our system and we can also configure its parameters to improve the system's performance.

In this unit, we will look at how to handle this process of preparing a kernel for our system. How, starting with the source code, we can obtain a new version of the kernel adapted to our system. Similarly, we will discuss how to develop the configuration and subsequent compilation and how to test the new kernel we have obtained.

1. The kernel of the GNU/Linux system

The core or kernel is the basic part of any operating system [Tan87], where the code of the fundamental services for controlling the entire system lie. Basically, its structure can be divided into a series of management components designed to:

- Manage processes: what tasks will be run, in what order and with what priority. An important aspect is the scheduling of the CPU: how do we optimise the CPU's time to run the tasks with the best possible performance or interactivity with users?

- Intercommunication of processes and synchronisation: how do tasks communicate with each other, with what different mechanisms and how can groups of tasks be synchronised?

- Input/output management (I/O): control of peripherals and management of associated resources.

- Memory management: optimising use of the memory, paginating service, and virtual memory.

- File management: how the system controls and organises the files present in the system and access to them.

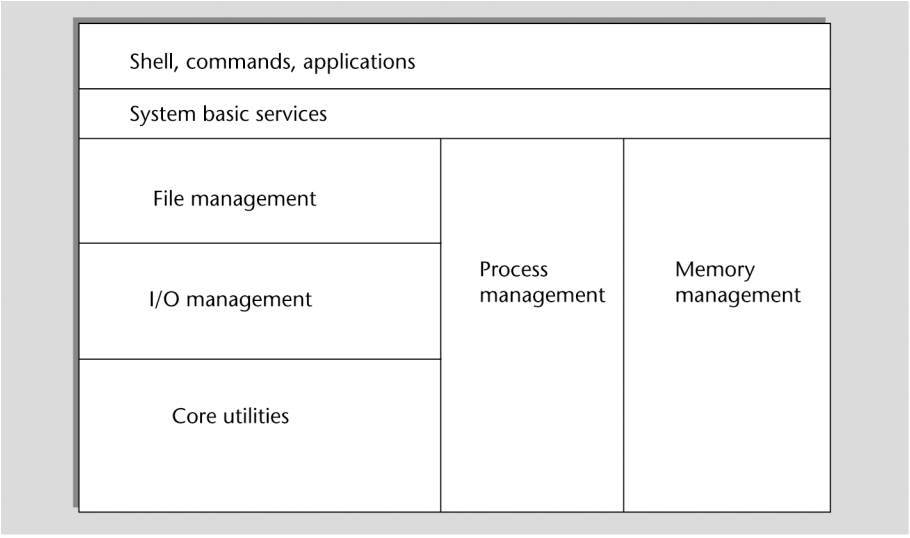

Figure 1. Basic functions of a kernel with regards to executed applications and commands

In proprietary systems, the kernel is perfectly "hidden" below the layers of the operating system's software; the end user does not have a clear perspective of what the kernel is and has no possibility of changing it or optimising it, other than through the use of esoteric editors of internal "registers" or specialised

third party programs, which are normally very expensive. Besides, the kernel is normally unique, it is the one the manufacturer provides and the manufacturer reserves the right to introduce any changes it wants whenever it wants and to handle the errors that appear in non-stipulated periods through updates offered to us in the form of error "patches" (or service packs).

One of the main problems of this approach is precisely the availability of these patches, having the error updates on time is crucial and if they are security-related, even more so, because until they are corrected we cannot guarantee the system's security for known problems. Many organisations, large companies, governments, scientific and military institutions cannot depend on the whims of a manufacturer to solve the problems with their critical applications.

The Linux kernel offers an open source solution with the ensuing permissions for modifying, correcting, generating new versions and updates very quickly by anyone anywhere with the required knowledge for doing so.

This allows critical users to control their applications and the system itself better, and offers the possibility of mounting systems with a "tailor-made" operating system adjusted to each individual's taste and in turn to have an open source operating system developed by a community of programmers who coordinate via the Internet, accessible for educational purposes because it has open source code and abundant documentation, for the final production of GNU/Linux systems adapted to individual needs or to the needs of a specific group.

Because the source code is open, improvements and solutions can be found immediately, unlike proprietary software, where we have to wait for the manufacturer's updates. Also, we can personalise the kernel as much as we wish, an essential requirement, for example, in high performance applications, applications that are critical in time or solutions with embedded systems (such as mobile devices).

Following a bit of (quick) history of the kernel [Kera] [Kerb]: it was initially developed by a Finnish student called Linus Torvalds, in 1991, with the intention of creating a similar version to Minix [Tan87] (version for PC of UNIX [Bac86]) for the Intel 386 processor. The first officially published version was Linux 1.0 in March 1994, which only included the execution for the i386 architecture and supported single-processor machines. Linux 1.2 was published in March 1995, and was the first version to cover different architectures such as Alpha, SPARC and Mips. Linux 2.0, in June 1996, added more architectures and was the first version to include multiprocessor support (SMP) [Tum]. In Linux 2.2, January 1999, SMP benefits were significantly increased, and controllers were added for a large amount of hardware. In 2.4, released in January 2001, SMP support was improved, new supported architectures were incorporated and controllers for USB, PC card devices were included (PCMCIA for laptops) part of PnP (plug and play), RAID and volumes support etc. Branch 2.6

of the kernel (December 2003), considerably improved SMP support, offered a better response of the CPU scheduling system, use of threads in the kernel, better support for 64-bit architectures, virtualisation support and improved adaptation to mobile devices.

Where the development is concerned, since the kernel was created by Linus Torvalds in 1991 (version 0.01), he has continued to maintain it, but as his work allowed it and as the kernel matured (and grew) he was helped to maintain the different stable versions of the kernel by different collaborators, while Linus continued (insofar as possible) developing and compiling contributions for the latest version of the kernel's development. The main collaborators of these versions have been [lkm]:

Note

The kernel has its origins in the MINIX system, a development by Andrew Tanenbaum, as a UNIX clone for PC.

- 2.0 David Weinehall.
- 2.2 Alan Cox (who also develops and publishes patches for most versions).
- 2.4 Marcelo Tosatti.
- 2.6 Andrew Morton / Linus Torvalds.

In order to understand a bit about the complexity of the Linux kernel, let's look at a table with a bit of a summarised history of its different versions and its size in relation to the source code. The table only shows the production versions; the (approximate) size is specified in thousands of lines (K) of source code:

Version	Publication date	Code lines (thousands)
0.01	09-1991	10
1.0	03-1994	176
1.20	03-1995	311
2.0	06-1996	649
2.2	01-1999	1800
2.4	01-2001	3378
2.6	12-2003	5930

As we can see, we have moved from about ten thousand lines to six million.

Now, development of branch 2.6.x of the kernel continues, the latest stable version, which most distributions include as the default version (although some still include 2.4.x, but 2.6.x is an option during the installation); although a certain amount of knowledge about the preceding versions is essential, because we can easily find machines with old distributions that have not been updated, which we may have to maintained or migrated to more modern versions.

Note

Today's kernel has reached a significant degree of complexity and maturity.

During the development of branch 2.6, the works on the kernel accelerated considerably, because both Linus Torvalds, and Andrew Morton (who maintain Linux 2.6) joined (in 2003) OSDL (Open Source Developement Laboratory) [OSDa], a consortium of companies dedicated to promoting the use of Open Source and GNU/Linux by companies (the consortium includes among many other companies with interests in GNU/Linux: HP, IBM, Sun, Intel, Fujitsu, Hitachi, Toshiba, Red Hat, Suse, Transmeta...). Now we are coming across an interesting situation, since the OSDL consortium sponsored the works of both the stable version of the kernel's maintainer (Andrew) and developer (Linus), working full time on the versions and on related issues. Linus remains independent, working on the kernel, while Andrew went to work for Google, where he continued his developments full time, making patches with different contributions to the kernel. After some time, OSDL became The Linux Foundation.

Note

The Linux Foundation: www.linuxfoundation.org

We need to bear in mind that with current versions of the kernel, a high degree of development and maturity has been achieved, which means that the time between the publication of versions is longer (this is not the case with partial revisions).

Another factor to consider is the number of people that are currently working on its development. Initially, there were just a handful of people with complete knowledge of the entire kernel, whereas nowadays many people are involved in its development. Estimates are almost two thousand with different levels of contribution, although the number of developers working on the hard core is estimated at several dozen.

We should also take into consideration that most only have partial knowledge of the kernel and neither do they all work simultaneously nor is their contribution equally relevant (some just correct simple errors); it is just a few people (such as the maintainers who have full knowledge of the kernel. This means that developments can take a while to occur, contributions need to be debugged to make sure that they do not come into conflict with each other and choices need to be made between alternative features.

Regarding the numbering of the Linux kernel's versions ([lkm][DBo]), we should bear in mind the following:

a) Until kernel branch 2.6.x, the versions of the Linux kernel were governed by a division into two series: one was known as the "experimental" version (with the second number being an odd number, such as 1.3.xx, 2.1.x or 2.5.x) and the other was the "production" version (even series, such as 1.2.xx, 2.0.xx, 2.2.x, 2.4.x and more). The experimental series were versions that moved rapidly and that were used for testing new features, algorithms, device drivers etc. Because of the nature of the exper-

imental kernels, they could behave unpredictably, losing data, blocking the machine etc. Therefore, they were not suited to production environments, unless for testing a specific feature (with the associated dangers).

Production or stable kernels (even series) were kernels with a well defined set of features, a low number of known errors and with tried and tested device controllers. They were published less frequently than the experimental versions and there were a variety of versions, some better than others. GNU/Linux distributions are usually based on a specifically chosen stable kernel, not necessarily the latest published production kernel.

b) The current Linux kernel numbering (used in branch 2.6.x), continues to maintain some basic aspects: the version is indicated by numbers *X.Y.Z,* where normally *X* is the main version, which represents important changes to the kernel; *Y* is the secondary version and usually implies improvements in the kernel's performance: *Y* is even for stable kernels and odd for developments or tests; and *Z* is the build version, which indicates the revision number of *X.Y*, in terms of patches or corrections made. Distributors do not tend to include the latest version of the kernel, but rather the one they have tested most frequently and can verify is stable for the software and components it includes. On the basis of this classical numbering scheme (followed during versions 2.4.x, until the early versions of branch 2.6), modifications were made to adapt to the fact that the kernel (branch 2.6.x) is becoming more stable (fixing X.Y to 2.6), and that there are fewer and fewer revisions (thus the leap in version of the first numbers), but development remains continuous and frenetic.

Under the latest schemes, four numbers are introduced to specify in Z minor changes or the revision's different possibilities (with different added patches). The version thus defined with four numbers is the one considered to be stable. Other schemes are also used for the various test versions (normally not advisable for production environments), such as -rc suffixes (*release candidate*), -mm which refers to experimental kernels with tests for different innovative techniques, or -git which are a sort of daily snapshot of the kernel's development. These numbering schemes are constantly changing to adapt to the way of working of the kernel community and its needs to accelerate the development.

c) To obtain the latest published kernel, you need to visit the Linux kernels file (at http://www.kernel.org) or its local mirror in Spain (http://www.es.kernel.org). It will also be possible to find some patches for the original kernel, which correct errors detected after the kernel's publication.

Note

Kernel repository:
www.kernel.org

Some of the technical characteristics ([DBo][Arc]) of the Linux kernel that we should highlight are:

• Kernel of the monolithic type: basically it is a program created as a unit, but conceptually divided into several logical components.

• It has support for loading/downloading portions of the kernel, these portions are known as modules, and tend to be characteristics of the kernel or device drivers.

• Kernel threading: for internal functioning, several execution threads are used internal to the kernel, which may be associated to a user program or to an internal functionality of the kernel. In Linux, this concept was not used intensively. The revisions of branch 2.6.x offered better support and a large proportion of the kernel is run using these various execution threads.

• Multithreaded applications support: user applications support of the multithread, since many computing paradigms of the client/server type, need servers capable of attending to numerous simultaneous requests, dedicating an execution thread to each request or group of requests. Linux has its own library of threads that can be used for multithread applications, with the improvements made to the kernel, they have also allowed a better use for implementing thread libraries for developing applications.

• The kernel is of a nonpreemptive type: this means that within the kernel, system calls (in supervisory mode) cannot be interrupted while the system task is being resolved and, when the latter finishes, the execution of the previous task is resumed. Therefore, the kernel within a call cannot be interrupted to attend to another task. Normally, preemptive kernels are associated to systems that operate in real time, where the above needs to be allowed in order to handle critical events. There are some special versions of the Linux kernel for real time, that allow this by introducing some fixed points where they can be exchanged. This concept has also been especially improved in branch 2.6.x of the kernel, in some cases allowing some resumable kernel tasks to be interrupted in order to deal with others and resuming them later. This concept of a preemptive kernel can also be useful for improving interactive tasks, since if costly calls are made to the system, they can cause delays in interactive applications.

• Multiprocessor support, known as symmetrical multiprocessing (SMP). This concept tends to encompass machines that incorporate the simple case of 2 up to 64 CPUs. This issue has become particularly relevant with multicore architectures, that allow from 2 or 4 to more CPU cores in machines accessible to domestic users. Linux can use multiple processors, where each processor can handle one or more tasks. But some parts of the kernel decreased performance, since they were designed for a single CPU and forced the entire system to stop

under certain cases of blockage. SMP is one of the most studied techniques in the Linux kernel community and important improvements have been achieved in branch 2.6. Since SMP performance is a determining factor when it comes to companies adopting Linux as an operating system for servers.

• File systems: the kernel has a good file system architecture, internal work is based on an abstraction of a virtual system (VFS, *virtual file system*), which can be easily adapted to any real system. As a result, Linux is perhaps the operating system that supports the largest number of file systems, from ext2, to MSDOS, VFAT, NTFS, journaled systems, such as ext3, ReiserFS, JFS(IBM), XFS(Silicon), NTFS, ISO9660 (CD), UDF and more added in the different revisions.

Other less technical characteristics (a bit of marketing):

a) Linux is free: together with the GNU software and included in any distribution, we can have a full UNIX-like system practically for the cost of the hardware, regarding GNU/Linux distribution costs, we can have it practically free. Although it makes sense to pay a bit extra for a complete distribution, with the full set of manuals and technical support, at a lower cost than would be paid for some proprietary systems or to contribute with the purchase to the development of distributions that we prefer or that we find more practical.

b) Linux can be modified: the GPL license allows us to read and to modify the source code of the kernel (on condition that we have the required know-how).

c) Linux can run on fairly limited old hardware; for example, it is possible to create a network server on a 386 with 4 MB of RAM (there are distributions specialised for limited resources).

d) Linux is a powerful system: the main objective of Linux is efficiency, it aims to make the most of the available hardware.

e) High quality: GNU/Linux systems are very stable, have a low fault ratio and reduce the time needed for maintaining the systems.

f) The kernel is fairly small and compact: it is possible to place it, together with some basic programs, on a disk of just 1.44 MB (there are several distributions on just one diskette with basic programs).

g) Linux is compatible with a large number of operating systems, it can read the files of practically any file system and can communicate by network to offer/receive services from any of these systems. Also, with certain libraries it can also run the programs of other systems (such as MSDOS, Windows, BSD, Xenix etc.) on the x86 architecture.

h) Linux has extensive support: there is no other system that has the same speed and number of patches and updates as Linux, not even any proprietary system. For a specific problem, there is an infinite number of mail lists and forums that can help to solve any problem within just a few hours. The only problem affects recent hardware controllers, which many manufacturers are still reluctant to provide if they are not for proprietary systems. But this is gradually changing and many of the most important manufacturers in sectors such as video cards (NVIDIA, ATI) and printers (Epson, HP,) are already starting to provide the controllers for their devices.

2. Configuring or updating the kernel

As GNU/Linux users or system administrators, we need to bear in mind the possibilities the kernel offers us for adapting it to our requirements and equipment.

At installation time, GNU/Linux distributions provide a series of preconfigured and compiled binary Linux kernels and we will usually have to choose which kernel from the available set best adapts to our hardware. There are generic kernels, oriented at IDE devices, others at SCSI, others that offer a mix of device controllers [AR01] etc.

Another option during the installation is the kernel version. Distributions normally use an installation that they consider sufficiently tested and stable so that it does not cause any problems for its users. For example, nowadays many distributions come with versions 2.6.x of the kernel by default, since it is considered the most stable version (at the time the distribution was released). In certain cases, as an alternative, more modern versions may be offered during the installation, with improved support for more modern (latest generation) devices that perhaps had not been so extensively tested at the time when the distribution was published.

Distributors tend to modify the kernel to improve their distribution's behaviour or to correct errors detected in the kernel during tests. Another fairly common technique with commercial distributions is to disable problematic features that can cause errors for users or that require a specific machine configuration or when a specific feature is not considered sufficiently stable to be included enabled by default.

Note

The possibility of updating and adapting the kernel offers a good adjustment to any system through tuning and optimisation.

This leads us to consider that no matter how well a distributor does the job of adapting the kernel to its distribution, we can always encounter a number of problems:

- The kernel is not updated to the latest available stable version; some modern devices are not supported.
- The standard kernel does not support the devices we have because they have not been enabled.
- The controllers a manufacturer offers us require a new version of the kernel or modifications.
- The opposite, the kernel is too modern, and we have old hardware that is no longer supported by the modern kernels.
- The kernel, as it stands, does not obtain the best performance from our devices.

- Some of the applications that we want to use require the support of a new kernel or one of its features.
- We want to be on the leading edge, we risk installing the latest versions of the Linux kernel.
- We like to investigate or to test the new advances in the kernel or would like to touch or modify the kernel.
- We want to program a driver for an unsupported device.
- ...

For these and other reasons we may not be happy with the kernel we have; in which case we have two possibilities: updating the distribution's binary kernel or tailoring it using the source.

Let's look at a few issues related to the different options and what they entail:

1) Updating the distribution's kernel: the distributor normally also publishes kernel updates as they are released. When the Linux community creates a new version of the kernel, every distributor joins it to its distribution and conducts the relevant tests. Following the test period, potential errors are identified, corrected and the relevant update of the kernel is made in relation to the one offered on the distribution's CDs. Users can download the new revision of the distribution from the website, or update it via some other automatic package system through a package repository. Normally, the system's version is verified, the new kernel is downloaded and the required changes are made so that the following time the system functions with the new kernel, maintaining the old version in case there are any problems.

> This type of update simplifies the process for us a lot, but may not solve our problems, since our hardware may not yet be supported or the feature of the kernel to be tested is still not in the version that we have of the distribution; we need to remember that there is no reason for distributor to use the latest available version (for example in kernel.org) but rather the one it considers stable for its distribution.

If our hardware is not enabled by default in the new version either, we will find ourselves in the same situation. Or simply, if we want the latest version, this process is no use.

2) Tailoring the kernel (this process is described in detail in the following sections). In this case, we will go to the sources of the kernel and "manually" adjust the hardware or required characteristics. We will pass through a process of configuring and compiling the source code of the kernel so as to create a binary kernel that we will install on the system and thus have it available the following time the system is booted.

Here we may also encounter two more options, either by default we will obtain the "official" version of the kernel (kernel.org), or we can go to the sources provided by the distribution itself. We need to bear in mind that distributions like Debian and Fedora do a lot of work on adapting the kernel and correcting kernel errors that affect their distribution, which means that in some cases we may have additional corrections to the kernel's original code. Once again, the sources offered by the distribution do not necessarily have to correspond to the latest published version.

This system allows us maximum reliability and control, but at a high administration cost; since we will need to have extensive knowledge of the devices and characteristics that we are selecting (what they mean and what implications they may have), in addition to the consequences that the decisions we make may imply.

3. Configuration and compilation process

Configuring the kernel [Vasb] is a costly process and requires extensive knowledge on the part of the person doing it, it is also one of the critical tasks on which the system's stability depends, given the nature of the kernel, which is the system's central component.

Note

The process of obtaining a new personalised kernel involves obtaining the sources, adapting the configuration, and compiling and installing the obtained kernel on the system.

Any error in the procedure can cause instability or the loss of the system. Therefore, it is advisable to make a backup of user data, configurations we have tailored, or, if we have the required devices, to make a complete system backup. It is also advisable to have a start up diskette (or Live CD distribution with tools) to help us in the event of any problem, or a rescue disk which most distributions allow us to create from the distribution's CDs (or by directly providing a rescue CD for the distribution).

Without meaning to exaggerate, if the steps are followed correctly, we know what we are doing and take the necessary precautions, errors almost never occur.

Let's look at the process required to install and configure a Linux kernel. In the following sections, we look at:

1) The case of old 2.4.x versions.

2) Some considerations regarding migrating to 2.6.x

3) Specific details regarding versions 2.6.x.

4) A particular case with the Debian distribution, which has its own more flexible compilation system (*debian way*).

Versions 2.4.x are practically no longer offered by current distributions, but we should consider that on more than one occasion we may find ourselves obliged to migrate a specific system to new versions or to maintain it on the old ones, due to incompatibilities or the existence of old unsupported hardware.

The general concepts of the compilation and configuration process will be explained in the first section (2.4.x), since most of them are generic, and we will subsequently see the differences with regard to the new versions.

3.1. Kernel compilation versions 2.4.x

The instructions are specifically for the Intel x86 architecture, by root user (although part of the process can be done as a normal user):

1) Obtaining the kernel: for example, we can visit www.kernel.org (or its FTP server) and download the version we would like to test. There are mirrors for different countries. In most GNU/Linux distributions, such as Fedora/Red Hat or Debian, the kernel's source code is also offered as a package (normally with some modifications included), if we are dealing with the version of the kernel that we need, it may be preferable to use these (through the kernel-source packages or similar). If we want the latest kernels, perhaps they are not available in the distribution and we will have to go to kernel.org.

2) Unpack the kernel: the sources of the kernel were usually placed and unpacked from the directory `/usr/src`, although we advise using a separate directory so as not to mix with source files that the distribution may carry. For example, if the sources come in a compressed file of the bzip2 type:

```
bzip2 -dc linux-2.4.0.tar.bz2 | tar xvf -
```

If the sources come in a gz file, we will replace bzip2 with gzip. When we decompress the sources, we will have generated a directory linux-version_kernel that we will enter in order to configure the kernel.

Before taking the steps prior to compilation, we should make sure that we have the right tools, especially the gcc compiler, make and other complementary gnu utilities for the process. For example, the *modutils*, the different utilities for using and handling the dynamic kernel modules. Likewise, for the different configuration options we should take into account a number of pre-requirements in the form of libraries associated to the configuration interface used (for example ncurses for the menuconfig interface).

In general, we advise checking the kernel documentation (whether via the package or in the root directory of the sources) to know what pre-requirements and versions of the kernel source will be needed for the process. We advise studying the README files in this root directory of the kernel source, and *Documentation/Changes* or the documentation index of the kernel in *Documentation/00-INDEX*.

If we have made previous compilations in the same directory, we need to make sure that the directory we use is clear of previous compilations; we can clear it using *make mrproper* (from the "root" directory).

For the process of configuring the kernel [Vasb], we have several alternative methods, which offer us different interfaces for adjusting the various parameters of the kernel (which tend to be stored in a configuration file, normally *.config* in the "root" directory of the sources). The different alternatives are:

• **make config:** from the command line we are asked for each option, and we are asked for confirmation (y/n) – yes or no, the option, or we are asked for the required values. Or the long configuration, where we are asked for many answers, and depending on each version, we will likewise have to answer almost a hundred questions (or more depending on the version).

• **make oldconfig:** it is useful if we want to reuse an already used configuration (normally stored in a *.config* file, in the root directory of the sources), we need to take into account that it is only valid if we are compiling the same version of the kernel, since different kernel versions can have variable options.

• **make menuconfig:** configuration based on text menus, fairly convenient; we can enable or disable what we want and it is faster than *make config*.

• **make xconfig:** the most convenient, based on graphic dialogues in X Window. We need to have tcl/tk libraries installed, since this configuration is programmed in this language. The configuration is based on tables of dialogues and buttons/checkboxes, can be done fairly quickly and has help with comments on most options. But it has a defect, which is that some options may not appear (it depends on whether the configuration program is updated and sometimes it is not). In this last case, *make config* (or *menuconfig*) is the only one we can be sure will offer all the options we can choose; for the other types of configuration it depends on whether the programs have been adapted to the new options in time for the kernel being released. Although in general they try to do it at the same time.

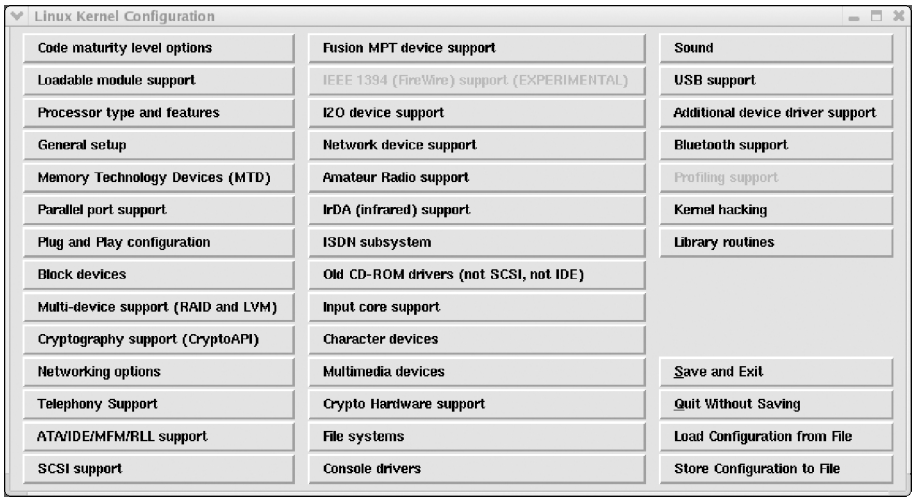

Figure 2. Configuration of the kernel (make xconfig) from graphic interface in X Window

Once the configuration process has been done, we need to save the file (*.config),* since the configuration requires a considerable amount of time. Also, it may be useful to have the configuration done if the plan is to do it on several similar or identical machines.

Another important issue concerning configuration options is that in many cases we will be asked if we want a specific characteristic integrated into the kernel or as a module (in the section on modules we will provide more details on them). This is a fairly important decision, since in certain cases our choice will influence the performance of the kernel (and therefore of the entire system).

The Linux kernel has become very large, due both to its complexity and to the device controllers (drivers) [AR01] that it includes. If we integrated everything, we could create a very large kernel file that would occupy a lot of memory and, therefore, slow down some functioning aspects. The modules of the kernel [Hen] are a method that makes it possible to divide part of the kernel into smaller sections, which will be loaded dynamically upon demand or when they are necessary for either explicit load or use of a feature.

The normal choice is to integrate what is considered fundamental for functioning or critical for performance within the kernel and to leave parts or controllers that will be used sporadically as modules for future extensions of the equipment.

• A clear case are the device controllers: if we are updating the machine, it may be that when it comes to creating the kernel we are not sure what hardware it will have: for example, what network card; but we do know that it will be connected to a network, so, the network support will be integrated into the kernel, but for the card controllers we can select a few (or all) of them and install them as modules. Then, when we have the card we can load the required module or

if we need to change one card for another later, we will just have to change the module to be loaded. If just one controller were integrated into the kernel and we changed the card, we would be forced to reconfigure and recompile the kernel with the new card's controller.

• Another case that arises (although it is not very common) is when we have two devices that are incompatible with each other, or when one or the other is functioning (for example, this tends to happen with a parallel cable printer and hardware connected to the parallel port). Therefore, in this case, we need to put the controllers as modules and load or download the one we need.

• Another example is the case of file systems. Normally we would hope that our system would have access to some of them, like ext2 or ext3 (belonging to Linux), VFAT (belonging to Windows 95/98/ME), and we will enable them in configuring the kernel. If at some moment we have to read another unexpected type, for example data stored on a disk or partition of the Windows NT/XP NTFS system, we would not be able to: the kernel would not know how to or would not have support to do so. If we have foreseen that at some point (but not usually) we may need to access these systems, we could leave the other file systems as modules.

3) Compiling the kernel

We will start the compilation using *make*, first we will have to generate the possible dependencies between the code and then the type of image of the kernel that we want (in this case, a compressed image, which tends to be the normal case):

```
make dep
make bzImage
```

When this process is completed, we will have the integrated part of the kernel; we are missing the parts that we have set as modules:

```
make modules
```

At this point we have done the configuring and compiling of the kernel. This part could be done by a normal user or by the root user, but now we will definitely need the root user, because we will move onto the installation part.

4) Installation

We'll start by installing the modules:

```
make modules_install
```

And the installation of the new kernel (from the directory /usr/src/linux-version or the one we have used as temporary):

```
cp arch/i386/boot/bzImage /boot/vmlinuz-2.4.0
cp System.map /boot/System.map-2.4.0
```

the file bzImage is the newly compiled kernel, which is placed in the */boot* directory. Normally, we will find the old kernel in the same */boot* directory with the name vmlinuz or vmlinuz-previous-version as a symbolic link to the old kernel. Once we have our kernel, it is better to keep the old one, in case any faults occur or the new one functions badly, so that we can recover the old one. The file System.map contains the symbols available for the kernel and is necessary for the processing of starting it up; it is also placed in the same directory.

On this point, we also need to consider that when the kernel starts up it may need to create initrd type files, which serve as a compound image of some basic drivers and is used when loading the system, if the system needs those drivers before booting certain components. In some cases, it is vital because in order to boot the rest of the system, certain drivers need to be loaded in a first phase; for example specific disk controllers such as RAID or volume controllers, which would be necessary so that in a second phase, the disk can be accessed for booting the rest of the system.

The kernel can be generated with or without an initrd image, depending on the needs of the hardware or system in question. In some cases, the distribution imposes the need to use an initrd image, in other cases it will depend on our hardware. It is also often used to control the size of the kernel, so that its basics can be loaded through the initrd image and later the rest in a second phase in the form of modules. In the case of requiring the initrd image, it would be created using the mkinitrd utility (see man, or chapter workshop), within the */boot* directory.

5) The following step is to tell the system what kernel it needs to boot with, although this depends on the Linux booting system:

- From booting with lilo [Zan][Skoa], whether in the MBR (*master boot record*) or from an own partition, we need to add the following lines to the configuration file (in: /etc/lilo.conf):

```
image = /boot/vmlinuz-2.4.0
 label = 2.4.0
```

where *image* is the kernel to be booted, and *label* is the name that the option will appear with during booting. We cam add these lines or modify the ones of the old kernel. We recommend adding them and leaving the old kernel, in case any problems occur, so that the old one can be recovered. In the file */etc/lilo.conf* we may have one or more start up configurations, for either Linux or other systems (such as Windows).

Every start up is identified by its line image and the label that appears in the boot menu. There is a line *default = label* that indicates the label that is booted by default. We can also add root = /dev/... to the preceding lines to indicate the disk partition where the main file system is located (the '/'), remembering that the disks have devices such as /dev/hda (1st disk ide) /dev/hdb (2 disk ide) or /dev/sdx for SCSI (or emulated) disks, and the partition would be indicated as *root = /dev/hda2* if the '/' of our Linux were on the second partition of the first ide disk. Using "*append* =" we can also add parameters to the kernel start up [Gor]. If the system uses initrd, we will also have to indicate which is the file (which will also be located in /boot/initrd-versionkernel), with the option "initrd=". After changing the lilo configuration, we need to write it for it to boot:

```
/sbin/lilo -v
```

We reboot and start up with the new kernel.

If we have problems, we can recover the old kernel, by selecting the option of the old kernel, and then, using the retouch lilo.conf, we can return to the old configuration or study the problem and reconfigure and recompile the kernel.

• Boot with grub [Kan01][Pro]. In this case, handling is simple, we need to add a new configuration consisting of the new kernel and adding it as another option to the grub file. Next, reboot in a similar way as with lilo, but remembering that in grub it is sufficient to edit the file (typically /boot/grub/menu.lst) and to reboot. It is also better to leave the old configuration in order to recover from potential errors.

3.2. Migration to kernel 2.6.x

In the case of having to update versions of old distributions, or changing the kernel generation using the source code, we will have to take some aspects into account, due to the novelties introduced into kernel branch 2.6.x.

Here is a list of some of the specific points to consider:

- Some of the kernel modules have changed their name, and some may have disappeared, we need to check the situation of the dynamic modules that are loaded (for example, examine /etc/modules and/or /etc/modules.conf) and edit them to reflect the changes.

- New options have been added to the initial configuration of the kernel: like make gconfig, a configuration based on gtk (Gnome). In this case, as a prerequisite, we will need to look out for Gnome libraries. The option make xconfig has now been implemented with the qt libraries (KDE).

- The minimum required versions of various utilities needed for the compilation process are increased (consult Documentation/Changes in the kernel sources). Especially, the minimum gcc compiler version.

- The default package for the module utilities has changed, becoming module-init-tools (instead of modutils used in 2.4.x). This package is a prerequisite for compiling kernels 2.6.x, since the modules loader is based on this new version.

- The devfs system becomes obsolete in favour of *udev*, the system that controls the *hotplug* start up (connection) of devices (and their initial recognition, in fact simulating a hotplug start up when the system boots), dynamically creating inputs in the directory /dev, only for devices that are actually present.

- In Debian as of certain versions of branch 2.6.x, for the binary images of the kernels, headers and source code, the name of the packages changes from kernel-images/source/headers to linux-image/source/headers.

- In some cases, new technology devices (like SATA) may have moved from /dev/hdX to /dev/sdX. In these cases, we will have to edit the configurations of /etc/fstab and the bootloader (lilo or grub) in order to reflect the changes.

- There may be some problems with specific input/output devices. The change in name of kernel modules has affected, among others, mouse devices, which likewise can affect the running of X-Window, until the required models are verified and the correct modules are loaded (for example *psmouse*). At the same time, the kernel integrates the Alsa sound drivers. If we have the old OSS, we will have to eliminate them from the loading of modules, since Alsa already takes care of emulating these.

- Regarding the architectures that the kernel supports, we need to bear in mind that kernel 2.6.x, in its different revisions, has been increasing the supported architectures which will allow us to have the binary images of the kernel in the distributions (or the options for compiling the kernel) best suited to supporting our processors. Specifically, we can find archi-

tectures such as i386 (for Intel and AMD): supporting the compatibility of Intel in 32 bits for the entire family of processors (some distributions use the 486 as the general architecture), some distributions integrate differentiated versions for i686 (Intel from pentium pro thereafter), for k7 (AMD Athlon thereafter), and those specific to 64 bits, for AMD 64 bits, and Intel with em64t extensions of 64 bits such as Xeon, and multicores. At the same time, there is also the IA64 architecture for 64bit Intel Itanium models. In most cases, the architectures have SMP capabilities activated in the kernel image (unless the distribution supports versions with and without SMP, created independently, in this case, the suffix -smp is usually added to the image that supports it).

- In Debian, to generate inirtrd images, as of certain versions of the kernel (>=2.6.12) the *mkinitrd* tools are considered obsolete, and are replaced with new utilities such as *initramfs* tools or *yaird*. Both allow the initrd image to be built, but the former is the recommended one (by Debian).

3.3. Compilation of the kernel versions 2.6.x

In versions 2.6.x, bearing in mind the abovementioned considerations, the compilation takes place in a similar way to the one described above:

Having downloaded the kernel 2.6.x (with x the number or pair of numbers of the kernel revision) to the directory that will be used for the compilation and checking the required versions of the basic utilities, we can proceed to the step of compiling and cleaning up previous compilations:

```
# make clean mrproper
```

configuration of parameters (remember that if we have a previous .config, we will not be able to start the configuration from zero). We do the configuration through the selected *make* option (depending on the interface we use):

```
# make menuconfig
```

construction of the kernel's binary image

```
# make dep
# make bzImage
```

construction of the modules (those specified as such):

```
# make modules
```

installation of the created modules (/lib/modules/version)

```
# make modules_install
```

copying of the image to its final position (assuming i386 as the architecture):

```
# cp arch/i386/boot/bzimage /boot/vmlinuz-2.6.x.img
```

and finally, creating the initrd image that we consider necessary, with the necessary utilities according to the version (see subsequent comment). And adjustment of the lilo or grub bootloader depending on which one we use.

The final steps (vmlinuz, system.map and initrd) of moving files to /boot can normally also be done with the process:

```
# make install
```

but we need to take into account that it does the entire process and will update the bootloaders, removing or altering old configurations; at the same time, it may alter the default links in the */boot* directory. We need to bear this in mind when it comes to thinking of past configurations that we wish to save.

Regarding the creation of the initrd, in Fedora/Red Hat it will be created automatically with the *install* option. In Debian we should either use the techniques of the following section or create it expressly using mkinitrd (versions <=2.6.12) or, subsequently, with *mkinitramfs*, or a utility known as *update-initramfs*, specifying the version of the kernel (it is assumed that it is called vmlinuz-version within the /boot directory):

```
# update-initramfs -c -k 'version'
```

3.4. Compilation of the kernel in Debian (Debian way)

In Debian, in addition to the examined methods, we need to add the configuration using the method known as Debian Way. A method that allows us to build the kernel in a fast and flexible manner.

For the process, we will need several utilities (install the packages or similar): kernel-package, ncurses-dev, fakeroot, wget, bzip2.

We can see the method from two perspectives, rebuilding a kernel equivalent to the one provided by the distribution or tailoring it and then using the method for building an equivalent personalised kernel.

In the first case, we initially obtain the version of the kernel sources provided by the distribution (meaning x the revision of the kernel 2.6):

```
# apt-get install linux-source-2.6.x
$ tar -xvjf /usr/src/linux-source-2.6.x.tar.bz2
```

where we obtain the sources and decompress them (the package leaves the file in */usr/src*).

Note

We can see the Debian way process in a detailed manner in: http://kernel-handbook.alioth.debian.org/

Installing the basic tools:

```
# apt-get install build-essential fakeroot
```

Checking source dependencies

```
# apt-get build-dep linux-source-2.6.x
```

And construction of the binary, according to the pre-established package configuration (similar to that included in the official image packages of the kernel in Debian):

```
$ cd linux-source-2.6.x
$ fakeroot debian/rules binary
```

There are some extra procedures for creating the kernels based on different patch levels provided by the distribution and possibilities of generating different final configurations (view the reference note to complement these aspects).

In the second, more common case, when we would like a personalised kernel, we will have to follow a similar process through a typical tailoring step (for example, using make menuconfig); the steps would be:

obtaining and preparing the directory (here we obtain the distribution's packages, but it is equivalent to obtaining the sources from *kernel.org*):

```
# apt-get install linux-source-2.6.x
$ tar xjf /usr/src/linux-source-2.6.x.tar.bz2
$ cd linux-source-2.6.x
```

next, we configure the parameters, as always, we can base ourselves on .config files that we have used previously, to start from a known configuration (for tailoring we can also use any of the other methods, xconfig, gconfig...):

```
$ make menuconfig
```

final construction of the kernel depending on initrd or not, without initrd available (we need to take care with the version we use; as of a certain version of the kernel, the use of the initrd image can be mandatory):

```
$ make-kpkg clean
$ fakeroot make-kpkg --revision=custom.1.0 kernel_image
```

or if we have initrd available (already built)

```
$ make-kpkg clean
$ fakeroot make-kpkg --initrd --revision=custom.1.0
kernel_image
```

The process will end with adding the associated package to the kernel image, which we will finally be able to install:

```
# dpkg -i ../linux-image-2.6.x_custom.1.0_i386.deb
```

In this section, we will also add another peculiarity to be taken into consideration in Debian, which is the existence of utilities for adding dynamic kernel modules provided by third parties. In particular, the *module-assistant* utility helps to automate this process on the basis of the module sources.

We need to have the headers of the kernel installed (package linux-headers-version) or the sources we use for compiling the kernel. As of here, the module-assistant can be used interactively, allowing us to select from an extensive list of previously registered modules in the application, and it can be responsible for downloading the module, compiling it and installing it in the existing kernel.

Also from the command line, we can simply specify (m-a is equivalent to module-assistant):

```
# m-a prepare
# m-a auto-install module_name
```

which prepares the system for possible dependencies, downloads the module sources, compiles them and, if there are no problems, installs them for the current kernel. We can see the name of the module on the interactive list of the module assistant.

4. Patching the kernel

In some cases the application of patches to the kernel [lkm] is also common.

> A patch file in relation to the Linux kernel is an ASCII text file that contains the differences between the original source code and the new code, with additional information on file names and code lines. The patch program (see *man patch*) serves to apply it to the tree of the kernel source code (normally in */usr/src*).

The patches are usually necessary when special hardware requires some modification of the kernel or some bugs (errors) have been detected subsequent to a wide distribution of a kernel version or else a new specific feature is to be added. In order to correct the problem (or add the new feature), it is usual to distribute a patch instead of an entire new kernel. When there are already several of these patches, they are added to various improvements of the preceding kernel to form a new version of the kernel. In all events, if we have problematic hardware or the error affects the functioning or stability of the system and we cannot wait for the next version of the kernel; we will have to apply the patch.

The patch is usually distributed in a compressed file of the type bz2 (bunzip2, although you can also find it in gzip with the extension .gz), as in the case of for example:

```
patchxxxx-2.6.21-pversion.bz2
```

where xxxx is usually any message regarding the type or purpose of the patch 2.6.21 would be the version of the kernel to which the patch is to be applied, and pversion would refer to the version of the patch, of which there can also be several. We need to bear in mind that we are speaking of applying patches to the sources of the kernel (normally installed, as we have already seen, in /usr/src/linuxor a similar directory).

Once we have the patch, we must apply it, we will find the process to follow in any readme file that accompanies the patch, but generally the process follows the steps (once the previous requirements are checked) of decompressing the patch in the source files directory and applying it over the sources of the kernel, for example:

```
cd /usr/src/linux (or /usr/src/linux-2.6.21 or any other ver-
sion).
```

```
bunzip2 patch-xxxxx-2.6.21-version.bz2
patch -p1 < patch-xxxxx-2.6.21-version
```

and afterwards we will have to recompile the kernel in order to generate it again.

The patches can be obtained from different places. Normally, we can find them in the kernel storage site (www.kernel.org) or else in www.linuxhq.com, which has a complete record of them. Some Linux communities (or individual users) also offer corrections, but it is better to search the standard sites in order to ensure that the patches are trustworthy and to avoid possible security problems with "pirate" patches. Another way is the hardware manufacturer, which may offer certain modifications of the kernel (or controllers) so that its devices work better (one known example is Linux NVIDIA and the device drivers for its graphic cards).

Finally, we should point out that many of the GNU/Linux distributions (Fedora/Red Hat, Mandriva...), already offer the kernels patched by themselves and systems for updating them (some even automatically, as in the case of Fedora/Red Hat and Debian). Normally, in production systems it is more advisable to keep up with the manufacturer's updates, although it does not necessarily offer the latest published kernel, but rather the one that it finds most stable for its distribution, at the expense of missing the latest generation features or technological innovations included in the kernel.

Note

For systems that we want to update, for testing reasons or because we need the latest features, we can always go to www.kernel.org and obtain the latest published kernel.

5. Kernel modules

The kernel is capable of loading dynamic portions of code (modules) on demand [Hen], in order to complement its functionality (this possibility is available from kernel version 1.2 and higher). For example, the modules can add support for a file system or for specific hardware devices. When the functionality provided by the module is not necessary, the module can be downloaded, freeing up memory.

On demand, the kernel usually identifies a characteristic not present in the kernel at that moment it makes contact with a thread of the kernel known as kmod (in kernel versions 2.0.x the daemon was called *kerneld*), this executes a command, modprobe, to try and load the associated module from or of a chain with the name of the module or else from an generic identifier; this information is found in the file /etc/modules.conf in the form of an alias between the name and the identifier.

Next, we search in `/lib/modules/version_kernel/modules.dep`

to find out whether there are dependencies with other modules. Finally, with the insmod command the module is loaded from /lib/modules/ version_kernel/ (the standard directory for modules), the version_kernel is the current version of the kernel using the uname -r command in order to set it. Therefore, the modules in binary form are related to a specific version of the kernel, and are usually located in /lib/modules/version-kernel.

If we need to compile them, we will need to have the sources and/or headers of the version of the core for which it is designed.

Note

The modules offer the system a large degree of flexibility, allowing it to adapt to dynamic situations.

There are some utilities that allow us to work with modules (they usually appear in a software package called *modutils*, which was replaced by the module -init-tools for managing modules of the 2.6.x branch):

- **lsmod**: we can see the loaded modules in the kernel (the information is obtained from the pseudofile /proc/modules). It lists the names and dependencies with others (in [ ]), the size of the module in bytes, and the module use counter; this allows it to be downloaded if the count is zero.

Example

Some modules in a Debian distribution:

Module	Size	Used by	Tainted: P
agpgart	37.344	3	(autoclean)
apm	10.024	1	(autoclean)
parport_pc	23.304	1	(autoclean)
lp	6.816	0	(autoclean)
parport	25.992	1	[parport_pc lp]
snd	30.884	0	
af_packet	13.448	1	(autoclean)
NVIDIA	1.539.872	10	
es1371	27.116	1	
soundcore	3.972	4	[snd es1371]
ac97_codec	10.9640	0	[es1371]
gameport	1.676	0	[es1371]
3c59x	26.960	1	

• **modprobe**: tries the loading of a module and its dependencies.

• **insmod**: loads a specific module.

• **depmod**: analyses dependencies between modules and creates a file of dependencies.

• **rmmod**: removes a module from the kernel.

• Other commands can be used for debugging or analysing modules, like *modinfo*, which lists some information associated to the module or *ksyms*, which (only in versions 2.4.x) allows examination of the symbols exported by the modules (also in /proc/ksyms).

In order to load the module the name of the module is usually specified, either by the kernel itself or manually by the user using *insmod* and specific parameters optionally. For example, in the case of devices, it is usual to specify the addresses of the I/O ports or IRQ or DMA resources. For example:

```
insmod soundx io = 0x320 irq = 5
```

6. Future of the kernel and alternatives

At certain moments, advances in the Linux kernel were released at very short intervals, but now with a fairly stable situation regarding the kernels of the 2.6.x series, more and more time elapses between kernel versions, which in some ways is very positive. It allows time for correcting errors, seeing what ideas did not work well, and trying new ideas, which, if they work, are included.

In this section, we'll discuss some of the ideas of the latest kernels and some of those planned for the near future in the development of the kernel.

Note

The kernel continues to evolve, incorporating the latest in hardware support and improved features.

The previous series, series 2.4.x [DBo], included in most current distributions, contributions were made in:

- Fulfilling IEEE POSIX standards, this means that many existing UNIX programs can be recompiled and executed in Linux.
- Improved devices support: PnP, USB, Parallel Port, SCSI...
- Support for new file systems, like UDF (CD-ROM rewritable like a disc). Other journaled systems, like Reiser from IBM or the ext3, these allow having a log (*journal*) of the file system modifications and thus they are able to recover from errors or incorrect handling of files.
- Memory support up to 4 GB, in its day some problems arose (with the 1.2x kernels) which would not support more memory than 128 MB (at that time it was a lot of memory).
- The /proc interface was improved. This is a pseudo-filesystem (the directory /proc) that does not really exist on the disc, but that is simply a way of accessing the data of the kernel and of the hardware in an organised manner.
- Sound support in the kernel: Alsa controllers, which were configured separately beforehand, were partially added,.
- Preliminary support for RAID software and the dynamic volumes manager LVM1 was included.

In the current series, kernel branch 2.6.x [Pra] has made important advances in relation to the previous one (with the different.x revisions of the 2.6 branch):

- Improved SMP features, important for the multi-core processors widely used in business and scientific environments.

- Improvements in the CPU scheduler.

- Improvements in the multithread support for user applications. New models of threads NGPT (IBM) and NPTL (Red Hat) are incorporated (over time NPTL was finally consolidated).

- Support for USB 2.0.

- Alsa sound controllers incorporated in the kernel.

- New architectures for 64-bit CPUs, supporting AMD x86_64 (also known as amd64) and PowerPC 64 and IA64 (Intel Itanium architecture).

- Support for journaled file systems: JFS, JFS2 (IBM), and XFS (Silicon Graphics).

- Improved I/O features, and new models of unified controllers.

- Improvements in implementing TCP/IP, and the NFSv4 system (sharing of the file system with other systems via the network).

- Significant improvements for a preemptive kernel: allowing the kernel to manage internally various tasks that can interrupt each other, essential for the efficient implementation of real time systems.

- System suspension and restoration after rebooting (by kernel).

- UML, User Mode Linux, a sort of virtual Linux machine on Linux that allows us to see a Linux (in user mode) running on a virtual machine. This is ideal for debugging now that a version of Linux can be developed and tested on another system, which is useful for the development of the kernel itself and for analysing its security.

- Virtualisation techniques included in the kernel: the distributions have gradually been incorporating different virtualisation techniques, which require extensions to the kernel; we should emphasise, for example, kernels modified for Xen, or Virtual Server (Vserver).

- New version of the volumes support LVM2.

- New pseudo file system */sys*, designed to include the system information and devices that will be migrating from the */proc* system, leaving the latter

with information regarding the processes and their development during execution.

- FUSE module for implementing file systems on user space (above all the NTFS case).

In the future, improvement of the following aspects is planned:

- Increasing the virtualisation technology in the kernel, for supporting different operating system configurations and different virtualisation technologies, in addition to better hardware support for virtualisation included in the processors that arise with new architectures.

- The SMP support (multi-processor machines) of 64-bit CPUs (Intel's Itanium, and AMD's Opteron), the support of multi-core CPUs.

- Improved file systems for clustering and distributed systems.

- Improvement for kernels optimised for mobile devices (PDA, teléfonos...).

- Improved fulfilment of the POSIX standard etc.

Web site

POSIX specifications
www.UNIX-systems.org/

- Improved CPU scheduling; although in the initial series of the 2.6.x branch many advances were made in this aspect, there is still low performance in some situations, in particular in the use of interactive desktop applications, different alternatives are being studied to improve this and other aspects.

Also, although it is separate from the Linux systems, the FSF (Free Software Foundation) and its GNU project continue working on the project to finish a complete operating system. It is important to remember that the main objective of the GNU project was to obtain a free software UNIX clone and the GNU utilities are just the necessary software for the system. In 1991, when Linux managed to combine its kernel with some GNU utilities, the first step was taken towards the culmination in today's GNU/Linux systems. But the GNU project continues working on its idea to finish the complete system. Right now, they already have a core that can run its GNU utilities. This core is known as Hurd; and a system built with it known as GNU/Hurd. There are already some test distributions, specifically, a Debian GNU/Hurd.

Web site

The GNU project:
http://www.gnu.org/gnu/thegnuproject.html

Hurd was designed as a core for the GNU system around 1990 when its development started, since most of the GNU software had already been developed at the time, and the only thing that was missing was the kernel. It was in 1991 when Linus combined GNU with his Linux kernel that the history of GNU/

Reference

GNU and Linux, by RichardStallman: http://www.gnu.org/gnu/linux-and-gnu.html

Linux systems began. But Hurd continues to develop. The development ideas for Hurd are more complex, since Linux could be considered a conservative design, based on already known and implemented ideas.

Specifically, Hurd was conceived as a collection of servers implemented on a Mach microkernel [Vah96], which is a kernel design of the microkernel type (unlike Linux, which is of the monolithic type) developed by the University of Carnegie Mellon and subsequently by that of Utah. The basic idea was to model the functionalities of the UNIX kernel as servers that would be implemented on a basic Mach kernel. The development of Hurd was delayed while the design of the Mach was being finished and this was finally published as free software, which would allow its use for developing Hurd. At this point, we should mention the importance of Mach, since many operating systems are now based on ideas extracted from it; the most outstanding example is Apple's MacOS X.

The development of Hurd was further delayed due to its internal complexity, because it had several servers with different tasks of the multithread type (execution of multiple threads), and debugging was extremely difficult. But nowadays, the first production versions of GNU/Hurd are already available, as well as test versions of a GNU/Hurd distribution.

It could be that in the not too distant future GNU/Linux systems will coexist with GNU/Hurd, or even that the Linux kernel will be replaced with the Hurd kernel, if some lawsuits against Linux prosper (read the case of SCO against IBM), since it would represent a solution for avoiding later problems. In all events, both systems have a promising future ahead of them. Time will tell how the balance will tip.

7. Tutorial: configuring de kernel to the requirements of the user

In this section we will have a look at a small interactive workshop for the process of updating and configuring the kernel in the two distributions used: Debian and Fedora.

The first essential thing, before starting, is to know the current version of the kernel we have with uname -r, in order to determine which is the the next version that we want to update to or personalise. And the other is to have the means to boot our system in case of errors: the set of installation CDs, the floppy disc (or CD) for recovery (currently the distribution's first CD is normally used) or some Live CD distribution that allows us to access the machine's file system, in order to redo any configurations that may have caused problems. It is also essential to back up our data or important configurations.

We will look at the following possibilities:

1) Updating the distribution's kernel. Automatic case of Debian.

2) Automatic update in Fedora.

3) Adapting a generic kernel (Debian or Fedora). In this last case, the steps are basically the same as those presented in the section on configuration, but we will make a few more comments:

7.1. Configuring the kernel in Debian

In the case of the Debian distribution, the installation can also be done automatically, using the APT packages system. It can be done either from the command line or with graphic APT managers (synaptic, gnome-apt...).

We are going to carry out the installation using the command line with apt-get, assuming that the access to the apt sources (above all to the Debian originals) is properly configured in the /etc/apt/sources.list file. Let's look at the steps:

1) To update the list of packages.

```
# apt-get update
```

2) To list the packages associated with images of the kernel:

```
# apt-cache search linux-image
```

3) To select a version suitable for our architecture (generic, 386/486/686 for Intel, k6 or k7 for amd or in particular for 64Bits versions amd64, intel and amd or ia64, for Intel Itanium). The version is accompanied by kernel version, Debian revision of the kernel and architecture. For example: 2.6.21-4-k7, kernel for AMD Athlon, Debian revision 4 of the kernel 2.6.21.

4) Check for the selected version that the extra accessory modules are available (with the same version number) With apt-cache we will search for whether there are other dynamic modules that could be interesting for our hardware, depending on the version of the kernel to be installed. Remember that, as we saw in the Debian way, there is also the module-assistant utility, which allows us to automate this process after compiling the kernel. If the necessary modules are not supported, this could prevent us from updating the kernel if we consider that the functioning of the problematic hardware is vital for the system.

5) Search, if we also want to have the source code of the kernel, the Linux-source-version (only 2.6.21, that is, the principal numbers) and the corresponding kernel headers, in case we later want to make a personalised kernel: in this case, the corresponding generic kernel patched by Debian.

6) Install what we have decided: if we want to compile from the sources or simply to have the code:

```
# apt-get install linux-image-version
# apt-get install xxxx-modules-version (if some modules are
necessary)
```

and

```
# apt-get install linux-source-version-generic
# apt-get install linux-headers-version
```

7) Install the new kernel, for example in the lilo bootloader (check the boot utility used, some recent Debian versions use grubas boot loader), this is done automatically. If we are asked if the initrd is active, we will have to verify the lilo file (/etc/lilo.conf) and, in the lilo configuration of the new image, include the new line:

initrd = /initrd.img-version (or /boot/initrd.img-version)

once this is configured, we would have to have a a lilo of the mode (fragment), supposing that initrd.img and vmlinuz are links to the position of the files of the new kernel:

```
default = Linux

image = /vmlinuz
   label = Linux
   initrd = /initrd.img
#  restricted
#  alias = 1
image = /vmlinuz.old
   label = LinuxOLD
   initrd = /initrd.img.old
#  restricted
#  alias = 2
```

We have the first image by default, the other is the former kernel. Thus, from the lilo menu we can ask for one or the other or, simply by changing the default, we can recover the former. Whenever we make any changes in /etc/lilo.conf we should not forget to rewrite in the corresponding sector with the command /sbin/lilo or /sbin/lilo -v.

7.2. Configuring the kernel in Fedora/Red Hat

Updating the kernel in the Fedora/Red Hat distribution is totally automatic by means of its package management service or else by means of the graphic programs that the distribution includes for updating; for example, in business versions of Red Hat there is one called up2date. Normally, we will find it in the task bar or in the Fedora/Red Hat system tools menu (check the available utilities in tools/Administration menus, the currently available graphic tools are highly distribution version dependent).

This updating program basically checks the packages of the current distribution against a Fedora/Red Hat database and offers the possibility of downloading the updated packages, including those of the kernel. This Red Hat service for businesses works via a service account and Red Hat offers it for payment. With this type of utilities the kernel is updated automatically.

For example, in figure 10, we can see that once running, a new available version of the kernel has been detected, which we can select for downloading:

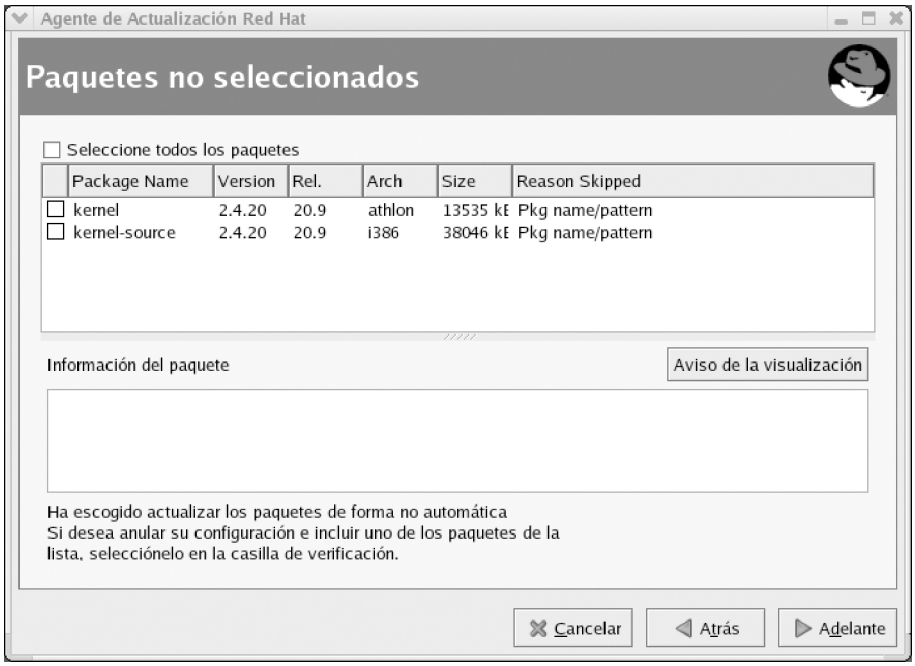

Figure 3. The Red Hat updating service (Red Hat Network up2date) shows the available kernel update and its sources.

In Fedora we can either use the equivalent graphic tools or simply use yum directly, if we know that new kernels are available:

```
# yum install kernel kernel-source
```

Once downloaded, we proceed to install it, normally also as an automatic process, whether with grub or lilo as boot managers. In the case of grub, it is usually automatic and leaves a pair of options on the menu, one for the newest version and the other for the old one. For example, in this grub configuration (the file is in /boot/grub/grub.conf or else /boot/grub/menu.lst), we have two different kernels, with their respective version numbers.

```
#file grub.conf
default = 1
timeout = 10
splashimage = (hd0,1)/boot/grub/splash.xpm.gz

title Linux (2.6.20-2945)
root (hd0,1)
kernel /boot/vmlinuz-2.6.20-2945 ro root = LABEL = /
initrd /boot/initrd-2.6.20-18.9.img

title LinuxOLD (2.6.20-2933)
root (hd0,1)
kernel /boot/vmlinuz-2.4.20-2933 ro root = LABEL = /
initrd /boot/initrd-2.4.20-2933.img
```

Each configuration includes a title that appears during start up. The root or partition of the disc from where it boots, the directory where the file corresponding to the kernel is found and the corresponding initrd file.

In the case of having lilo (by default grub is used) in the Fedora/Red Hat as manager, the system will also update it (file /etc/lilo.conf), but then we will have to rewrite the boot manually with the command /sbin/lilo.

It is also important to mention that with the previous installation we had the possibility of downloading the sources of the kernel; these, once installed, are in/usr/src/linux-version and can be compiled and configured following the usual procedure as if it was a generic kernel. We should mention that the Red Hat company carries out a lot of work on the patches and corrections for the kernel (used after Fedora) and that its kernels are modifications to the generic standard with a fair number of additions, which means that it could be better to use Red Hat's own sources, unless we want a newer or more experimental kernel than the one supplied.

7.3. Configuring a generic kernel

Let's look at the general case of installing a kernel starting from its sources. Let's suppose that we have some sources already installed in /usr/src (or the corresponding prefix). Normally, we would have a Linux directory or linux-version or simply the version number. This will be the tree of the sources of the kernel.

These sources can come from the distribution itself (or we may have downloaded them during a previous udpate), first it will be interesting to check whether they are the latest available, as we have already done before with Fedora or Debian. Or if we want to have the latest and generic versions, we can go to kernel.org and download the latest available version (better the stable one than the experimental ones), unless we are interested in the kernel's development. We download the file and in /usr/src (or another selected directory, even better) decompress the kernel sources. We can also search to see if there are patches for the kernel and apply them (as we have seen in section 4.4).

Next, we will comment on the steps that will have to be carried out: we will do it briefly, as many of them have been mentioned before when working on the configuration and tailoring.

See also

It would be advisable to reread section 3.4.3.

1) Cleaning the directory of previous tests (where applicable):

```
make clean mrproper
```

2) Configuring the kernel with, for example: *make menuconfig (or xconfig, gconfig or oldconfig*). We saw this in section 4.3.

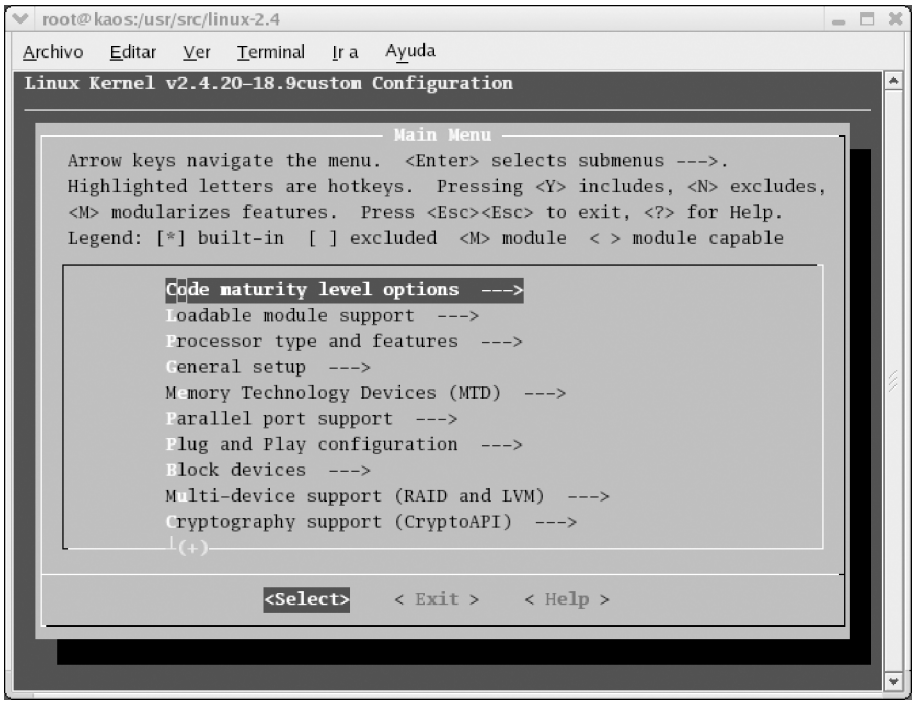

Figure 4. Configuring the kernel using text menus

4) Dependencies and cleaning of previous compilations:

```
make dep
```

5) Compiling and creating an image of the kernel: make bzImage. zImage would also be possible if the image was smaller, but *bzImage* is more normal, as it optimises the loading process and compression of larger kernels. On some ancient hardware it may not work and *zImage* may be necessary. The process can last from a few minutes to an hour on modern hardware and hours on older hardware. When it finishes, the image is found in: /usr/src/directory-sources/arch/i386/boot.

6) Now we can compile the modules with *make modules*. Until now we have not changed anything in our system. Now we have to proceed to the installation.

7) In the case of the modules, if we try an older version of the kernel (branch 2.2 or the first ones of 2.4), we will have to be careful, since some used to overwrite the old ones (in the last 2.4.x or 2.6.x it is no longer like this).

But we will also need to be careful if we are compiling a version that is the same (exact numbering) as the one we have (the modules are overwritten), it is better to back up the modules:

```
cd /lib/modules
tar -cvzf old_modules.tgz versionkernel-old/
```

This way we have a version in .tgz that we can recover later if there is any problem And, finally, we can install the modules with:

```
make modules install
```

8) Now we can move on to installing the kernel, for example with:

```
# cd /usr/src/directory-sources/arch/i386/boot
# cp bzImage /boot/vmlinuz-versionkernel
# cp System.map /boot/System.map-versionkernel
# ln -s /boot/vmlinuz-versionkernel /boot/vmlinuz
# ln -s /boot/System.map-versionkernel /boot/System.map
```

This way we store the symbols file of the kernel (System.map) and the image of the kernel.

9) Now all we have to do is put the required configuration in the configuration file of the boot manager, whether lilo (/etc/lilo.conf) or grub (/boot/grub/grub.conf) depending on the configurations we already saw with Fedora or Debian. And rememeber, in the case of lilo, that we will need to update the configuration again with /sbin/lilo or /sbin/lilo -v.

10) Restart the machine and observe the results (if all has gone well).

Activities

1) Determine the current version of the Linux kernel incorporated into our distribution. Check the available updates automatically, whether in Debian (*apt*) or in Fedora/Red Hat (via yum).

2) Carry out an automatic update of our distribution. Check possible dependencies with other modules used (whether pcmcia or others) and with the bootloader (lilo or grub) used. A backup of important system data (account users and modified configuration files) is recommended if we do not have another sytem that is available for tests.

3) For our branch of the kernel, to determine the latest available version (consult http://www.kernel.org) and carry out a manual installation following the steps described in the unit. The final installation can be left optional, or else make an entry in the bootloader for testing the new kernel.

4) In the case of the Debian distribution, in addition to the manual steps, we saw how there is a special way (recommended) of installing the kernel from its sources using the kernel-package.

Bibliography

Other sources of reference and information

[Kerb] Site that provides a repository of the different versions of the Linux kernel and its patches.

[Kera] [lkm] Web sites that refer to a part of the Linux kernel community. It offers various documentary resources and mailing lists of the kernel's evolution, its stability and the new features that develop.

[DBo] Book about the Linux 2.4 kernel, which details the different components, their implementation and design. There is a first edition about the 2.2 kernel and a new update to the 2.6 kernel.

[Pra] An article that describes some of the main innovations of the new 2.6 series of the Linux kernel.

[Ker] [Mur] Documentation projects of the kernel, incomplete but with useful material.

[Bac86] [Vah96] [Tan87] Some texts about the concepts, design and implementation of the kernels of different UNIX versions.

[Skoa][Zan01][Kan][Pro] For further information on lilo and grub loaders.