GNU/Linux Kernel basis

Or how to recompile a Kernel without causing the Big Crunch

T. Castillo Girona
<toni.castillo@fa.upc.edu>
Department of Applied Physics, U.P.C.

This document will be available at https://dfa.upc.es/pub/bscw.cgi/d4651

Abstract

We will discuss how **GNU/Linux Kernel** can be compiled using a generic method, trying to avoid distribution-dependant techniques, and how the Kernel implements its device driver model. This way, we will be able to figure out what is going on behind the scenes. To conclude, this talk will focus on Kernel errors like `OOPS` and `PANIC` messages.

Contents

1 What is the Kernel?

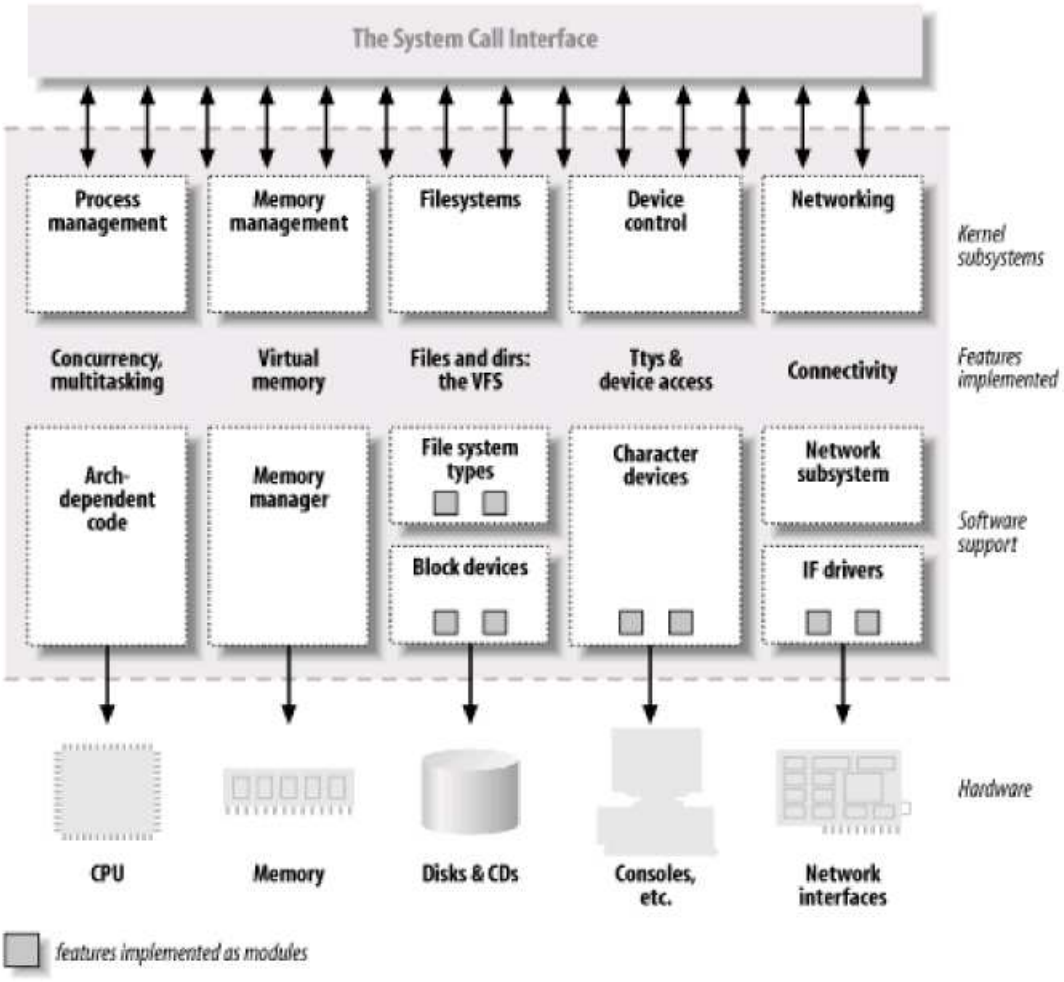

Figure 1: The GNU/Linux Micro-kernel

The GNU/Linux Kernel is a program in charge of dealing directly with the computer's hardware, assigning I/O resources to processes, scheduling these ones, among many other low-level tasks. In a simple way, it could be considered as a bridge between user-land and hardware-land.

1.1 Kernel types

Basically, we can split all existing Kernels in two main families: the **Monolithic** ones and the **Micro-kernel** ones.

1.1.1 Monolithic Kernels

All layers are stored inside the Kernel program, loaded entirely in the memory, running in Kernel Mode. Bear in mind that the unnecessary ones are loaded, too, and that can be inefficient.

1.1.2 Micro-kernel Kernels

These kernels are composed of different layers, each one in charge of some specific task. Any layer can be unloaded from memory, saving resources when they aren't necessary anymore. Obviously, any layer can use some exported symbols (discussed below) from other ones, with no need to re-implement them.

The GNU/Linux Kernel belongs to these ones.

1.2 Execution Rings

Normally, any CPU has a minimum of two main execution modes, or execution rings. The first one is called user-mode, user-space or, in Intel architectures, Ring 3. The last one is known as kernel-mode, kernel-space or simply Ring 0.

In the first one, there's no direct access to hardware resources. As long as a certain user-space process needs to gain some access to a hard disk, to the main memory, and so on, it needs to call some kernel's routine (so-called system calls) in order to jump from Ring 3 to Ring 0.

Clearly, the GNU/Linux Kernel program runs always in Ring 0.

Kernel Threads

There are some special kind of processes, well-known as Kernel Threads, which run always in Ring 0. Generally, they are created at system start up and remain their execution until the system halts.

They can be found on any GNU/Linux system by running the ps -ef command. Kernel Threads appear between brackets:

```
    ...
root 2 1 0 Mar08 ?  00:00:00 [migration/0]
root 3 1 0 Mar08 ?  00:00:00 [ksoftirqd/0]
root 11 1 0 Mar08 ?  00:00:00 [kthread]
...
```

1.3 Preemptive Kernels

The earlier GNU/Linux Kernel versions were not preemptive, that is, as long as a certain process p was running in Ring 0, there was no possibility to suspend its execution until it returned to Ring 3.

Current versions of the GNU/Linux Kernel can accomplish that (finally). Thus, as far as we are concerned, Linux is a preemptive kernel.

1.4 Reentrant Kernels

A reentrant Kernel allows more than one process running inside Ring 0 execution mode. Obviously, if there is only one processor, there could be a lot of suspended processes waiting inside Ring 0.

The GNU/Linux Kernel is reentrant, also.

1.5 Kernel versions

$$2.4^{1}.18$$
$$2.5^{2}.23^{3}$$
$$2.6.28^{4}.5^{5}$$

2 Compiling the Linux Kernel

There are a lot of methods in order to compile a new Linux Kernel's image. Some of them are distribution-dependant. We will consider the standard one, useful in all GNU/Linux's distributions.

2.1 The compiling process

The whole compilation process consists of the six steps, discussed below:

2.1.1 Getting the Kernel sources

Normally, all GNU/Linux distributions have their own Linux source code as a meta-package. For example, Debian has the package `linux-source-2.X.Y` available on its source repositories.

Thus, we can install either the GNU/Linux Kernel included in the distro repositories or the "official" one available at:

`http://www.kernel.org/pub/linux/kernel/`

2.1.2 Configuring the Kernel

In order to configure the GNU/Linux Kernel, one can choose between different "make" methods. The easiest one is by using the N-curses libraries. It's quickly, can be executed in a virtual terminal across remote secure shell connections with no latencies, and it has its own User Interface (UI). This can be done by typping:

```
# cd /usr/src/linux-2.6.XX.Y
# make menuconfig
```

2.1.3 Compiling the Kernel Image

Just after choosing the desired Kernel options, it is time to compile the new Kernel's image. To do this:

`# make [-j`*N*`]` [6] `bzImage`

The new GNU/Linux Kernel's image will be placed at platform-dependant directory **arch**, i.e: `arch/x86_64/boot/bzImage` in the particular case of EM64T architectures, and `arch/i386/boot/bzImage` in the X86 ones.

[1] Even: stable version
[2] Odd: unstable, under development
[3] Development release
[4] Stable release
[5] Stable patch
[6] *N* could be *2..n*, where n is the number of processors present on the Linux box

2.1.4 Compiling and installing the modules

To build all options marked as modules (M) in the first step, simply execute:
`# make [-jN] modules`

It's necessary to install all of them in the right place. To do this run:
`# make modules_install`

After a while, all modules will be copied to /lib/modules/2.X.YY/

2.1.5 The initrd image

Sometimes the GNU/Linux Kernel isn't capable of booting by itself because of some non-loaded drivers - such as hard disk controllers, file-systems, and so on -. So, it is in need of an initrd image, which, in turn, is in charge of doing some previous tasks in order to help its start up.

There are two main file-systems designed to create a new initrd image file: cramfs and ext2. We can choose either, it doesn't matter. In addition, we can use a g-zipped cpio file format.

Generally, initrd creation can be achieved by using a script named mkinitrd or mkinitramfs. In the particular case of GNU/Linux Debian distributions, we can create our own initrd image file based on the new compiled kernel by running:
`# mkinitramfs -o /boot/initrd-2.X.Y.Z 2.X.Y.Z`

2.1.6 Installing the new kernel's image

The last step consists on installing the new Kernel's image to /boot directory, naming it accordingly and updating the Boot Loader [7] - if needed -.

```
# cp arch/platform/boot/bzImage /boot/vmlinuz-myVersion
# vim /boot/grub/menu.lst ...
# vim /etc/lilo.conf ...
```

Don't forget to add a new initrd entry if you are using one. Let us take a look at the Figure 2 - GRUB loader example - and Figure 3 - lilo loader example -.

```
 title Debian Etch 2.6.25.6 Preemptive Voluntary ticknessless
root (hd0,0)
kernel /vmlinuz-2.6.25.6-kp root=/dev/md2
initrd /initrd-2.6.25.6-kp
```

Figure 2: /boot/grub/menu.lst configuration file

2.2 Saving and restoring the Kernel's configuration

Obviously, there must be a simple way for recovering or saving the GNU/Linux Kernel's configuration selected in the first step. All GNU/Linux distros store their configuration concerning the running kernel in a file placed at `/boot/config-2.X.Y`.

[7] Bear in mind that in case of using LILO, you have to run the command `lilo` right after modifying or adding entries in `/etc/lilo.conf` configuration file.

```
 image=/boot/vmlinuz-2.6.25.6-kp
initrd=/boot/initrd-2.6.25.6-kp
root=/dev/md2
label=2.6.25.6-nokp
read-only
```

Figure 3: The same configuration but in the /etc/lilo.conf file

Nowadays, modern kernels can accomplish this by enabling `CONFIG_IKCONFIG` symbol, as well.

Thus, we can recover the entire Kernel's options by typing in a shell:

```
# zcat /proc/config.gz > /usr/src/linux-2.X.Y.Z/.config
```

Or by using the extract_config script, available inside the GNU/Linux Kernel sources:

```
# cd /usr/src/linux-2.X.Y.Z/
# scripts/extract_config /boot/vmlinuz-2.X.Y.Z > .config
```

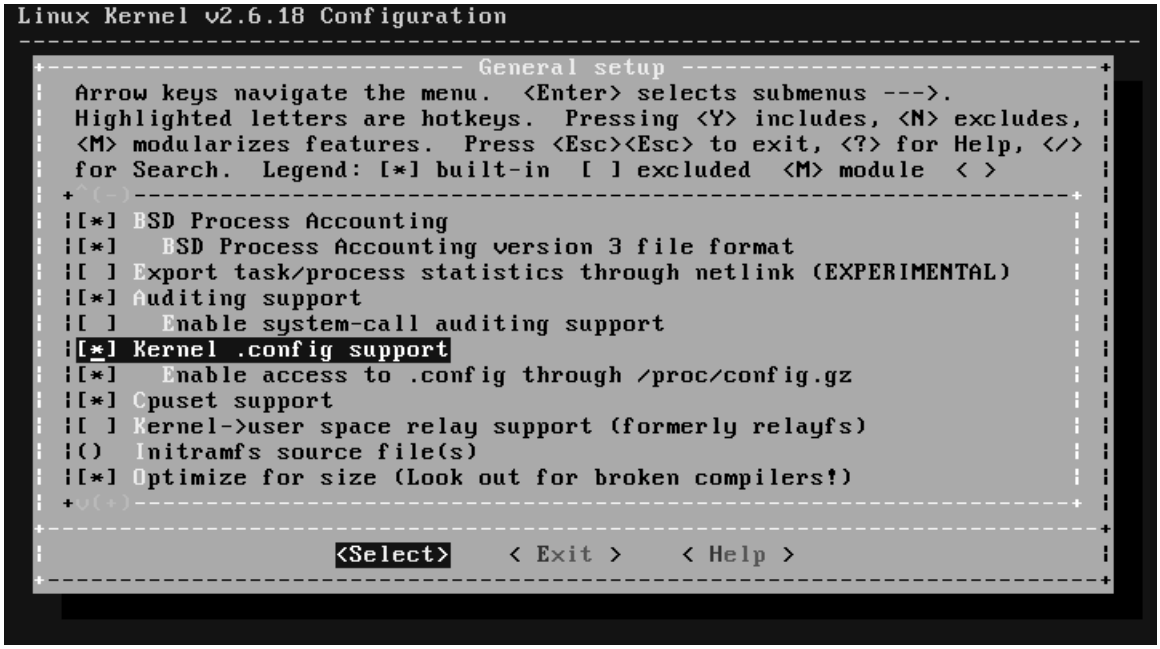

Figure 4: Enabling Kernel's config through /proc/config.gz interface

2.3 Initrd's entrails

It is feasible to modify an existing initrd image file without creating a new one from scratch. Sometimes, we need to add a new module to the initrd image file directly, or modify some module's parameters.

As we know, an initrd image file it is based on ext2,cramfs file-systems or cpio file format. Thus, we can mount, copy, modify, add or delete files inside

the initrd image file.

Let us suppose that we have an initrd image file called /boot/initrd.img-2.6.18-test. First of all, we must check its format:

```
# file/boot/initrd.img-2.6.18-test
      /boot/initrd.img-2.6.18-test:  gzip compressed data ...
```

In this case, the initrd image file is compressed using gzip. So, obviously, we have to g-unzip it:

```
# gunzip -f -S "" -c /boot/initrd.img-2.6.18-test >
        /boot/initrd.img-2.6.18-test-gunzipped
# file /boot/initrd.img-2.6.18-test-gunzipped
       /boot/initrd.img-2.6.18-test-gunzipped:  ASCII cpio archive ...
# mkdir /tmp/initrd
# cd /tmp/initrd
# cat /boot/initrd.img-2.6.18-test-gunzipped | cpio -id
      20505 blocks
# ls -l
     bin conf etc init lib modules sbin scripts
```

Right after g-unzipping and recovering the entire directory structure from cpio file, we can add, delete or modify any existing file inside /tmp/initrd directory. All modules are stored inside /lib/modules/2.X.Y.Z directory.

The `init` script will be executed as soon as the initrd image file is decompressed at start up time by the GNU/Linux Kernel. In the particular case of ext2 initrds, instead of "init" script file we can find out another equivalent, usually named as linuxrc.

Once the initrd structure has been modified it is necessary to go backwards in the former process, that is, create the new cpio file, g-zipping it and copying it to the right place.

```
# find ./ | cpio -H newc -o > /tmp/new-initrd.cpio
       20505 blocks
# gzip -c /tmp/new-initrd.cpio > /boot/initrd.img-2.6.18-test
```

At this step, the new initrd image file has been updated, and it can be used immediately. After using it, the GNU/Linux Kernel frees all allocated memory, and so it shows through ksyslog:

```
Freeing initrd memory:  4405k freed
```

2.3.1 Initrd example: reading some args

In this subsection we are going to change the initrd behaviour: we want read a certain parameter which will be applied only to a desired kernel module, called "mymodule". Obviously, this module won't be loaded because it doesn't exist.

First of all, let us take a look at the init script, just before the shell function load_modules:

```
...
echo -ne "TCG: please, insert the module parameters for mymodule:  \>"
read marg
load_modules $marg
...
```

Clearly, we have added some code to the script in order to pass an extra parameter to the shell function load_modules, reading it using read shell-statement.

All common functions are located inside scripts/ directory. The shell function load_modules is implemented in script/functions file. Thus, we must change some aspects of it:

```
...
if [ "$m" = "mymodule" ]; then
     echo -ne "\t Personalised module $m with arg $marg \n"
     Do some tasks ...
else
     modprobe -q $m
fi
...
```

To conclude, we have to add mymodule to conf/modules:

```
# echo "mymodule" >> conf/modules
```

After re-generating once again the initrd g-zipped cpio file, the new initrd is ready to use. So, next time the system boots up, we will see what is shown in Figure 5.

```
TCG: Just a quick example using INITRD...
Begin: Loading essential drivers... ...
TCG: please, insert the module parameters for mymodule: \>  myownparm
        Personalized module mymodule with arg myownparm
```

Figure 5: initrd altered behaviour

2.4 Patching the GNU/Linux Kernel's sources

When the running Kernel must be upgraded, it is feasible to patch the current sources and recompiling them as described earlier. To do this we need to get the patch, then apply it to the current kernel source tree. For example, in order to upgrade our Kernel from 2.6.18.1 version to 2.6.18.2:

```
# wget http://www.kernel.org/pub/linux/kernel/v2.6/patch-2.6.18.2.bz2
```

Then, we have to apply the patch:

```
# cd /usr/src/linux-2.6.18.1
# bzcat /path/to/patch-2.6.18.2.bz2|patch -p1 -r /tmp/rejs.rej
...
patching file sound/core/hwdep.c
patching file sound/core/info.c
```

```
...
patching file sound/pci/emu10k1/emu10k1_main.c
```

If something goes wrong, we can take a look at /tmp/rejs.rej file in order to determine which files couldn't be patched.

Before trying to recompile the new patched kernel mainstream, we have to clean up any object files running:

```
# make mrproper
```

3 The Linux Device Driver model

Looking back at Figure 1, the GNU/Linux Kernel is composed of many different layers, each of them in charge of some concrete tasks. Some of these layers can be compiled as modules, and some others not. Usually, when compiling the GNU/Linux Kernel we can choose when a layer will be a module or not. Bear in mind any non- module layer will be linked and inserted physically inside the Kernel's image file. As a result, the new Kernel will occupy more memory. Clearly, the image's resultant file will be larger in size than another one with more layers compiled as modules.

Talking about The GNU/Linux Device Driver Model introduces the concept of Linux Kernel Module (LKM).

3.1 About Modules

It's just an ELF object file, not linked, stored on the hard drive. In modern kernels, these files are recognised because of its extension, ko. In the older 2.4x Kernel's branch, they had *.o extension. These files can be drivers or not. In other words, not all modules are drivers. For example, there are some LKMs which allow users to mount a file-system, like vfat.ko or ext3.ko. As a matter of fact, they aren't drivers at all.

In all GNU/Linux kernels, the compiled modules are saved in /lib/modules/ 2.X.Y.Z directory.

In order to compile some layer as a module, we have to select M in the make menuconfig screen, as shown in Figure 6.

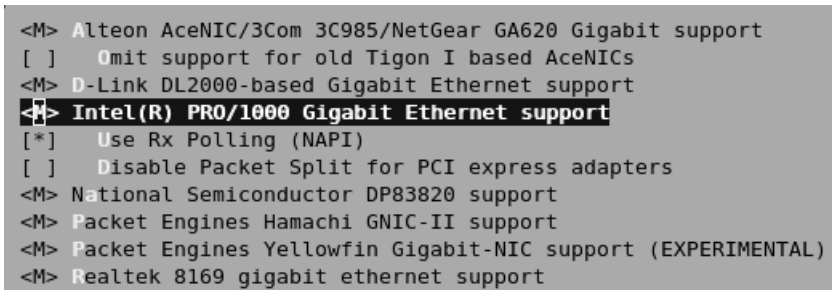

Figure 6: Compiling some layers as LKMs

3.1.1 Dealing with modules

In order to deal with the LKMs, the GNU/Linux Kernel offers a group of tools, well-known as the modutils. With these sort of commands, we can load, unload or just list which modules are loaded. In fact, all loaded modules are linked inside the GNU/Linux kernel program. Bear in mind that this link procedure will take place as soon as a new module is "loaded" using the modutils, and unlinked as soon as it is unloaded.

3.1.2 Loading a module

There are two methods for loading modules:

`# insmod /path/to/module.ko` arg_1`=value` [8] `...` arg_N`=value`$_N$

or

`# modprobe module` arg_1`=value ...` arg_N`=value`$_N$

There are int, short, long, char*, int[], short[],long[] and char** parameters. For example, in order to pass three parameters to a module called my_module.ko, two of them an int value and the last one an array of integers, we must run something like:

```
# modprobe my_module io=0x220 irq=5 dma=1,5
```

Generally, in any GNU/Linux distribution there's a file in order to configure what modules will be loaded automatically with their own parameters. In addition, the GNU/Linux Kernel tries to auto-load some modules as soon as they are needed, no matter what modules are written down in the module's configuration file. This option must be explicitly configured in the make menuconfig screen, as shown in Figure 7.

Figure 7: Enabling kernel modules auto-loading option

Before loading the desired module, all modules needed to load it perfectly must be linked in the runtime Kernel. In case of using insmod, it could be difficult to realize what modules would be loaded previously to avoid unknown symbol messages - see below -. Thus, we need to use modprobe. This program tries to load all dependencies looking at /lib/modules/2.X.Y.Z/modules.dep file, loading them just before loading the given module.

In particular, the module "msdos.ko", in charge of allowing access to msdos file-systems, needs the symbols exported by the module fat.ko which, in turn, has no dependencies:

```
/lib/modules/2.6.18-kmodest/kernel/fs/msdos/msdos.ko:
     /lib/modules/2.6.18-kmodest/kernel/fs/fat/fat.ko
...
/lib/modules/2.6.18-kmodest/kernel/fs/fat/fat.ko:
```

[8] To determine the allowed module's parameters, run `# modinfo module`.

Modules.dep is generated and updated after any call to make modules_install, as discussed earlier. The command needed to re-generate it by hand is:

```
# depmod -ae
```

3.1.3 Unloading a module

To unload a module we have to use rmmmod command:

```
# rmmod module
```

Sometimes, it could be not possible to unload a module using rmmod because of its reference counter. Clearly, some modules have dependencies, and these ones cannot be unloaded as long as others dependants modules are loaded and in use. The lsmod command reports us if a certain module is being used:

```
Module Size Used by
cdrom 32544 1 ide_cd
```

The reference counter of a certain LKM is maintained internally by the Kernel, but, sometimes, an LKM developer can prefer to write some code for controlling it directly, thanks to try_module_get() and module_put() C functions.

In addition, the GNU/Linux Kernel can be compiled without the MODULE_UNLOAD symbol: in this case, unloading a module is not allowed.

3.2 Device drivers

In short, there are three main device drivers types:

1. Character device drivers.
2. Block device drivers.
3. Network interface cards.

In fact, all of them but NICs have their associated special device file under /dev directory, controlled by UDEV subsystem in modern GNU/Linux Kernel implementations. In the particular case of NICs, there's no /dev entry because of their own implementation is so far away from the main idea of streaming, like character or block devices are.

3.2.1 Character devices

All char device drivers are accessed with no buffers, and they allow to read or write any number of bytes per read/write access. Generally, this device drivers implement hooks for the calls open(), close(), lseek(), read(), write(), fcntl(), ioctl() ...

Inside the /dev directory, these devices are recognised by the character "c":

```
crw------- 1 root root 4, 1 Mar 13 08:54 /dev/tty1
crwxrwxrwx 1 root root 1, 3 Jan 26 2006 /dev/null
crw-rw-rw- 1 root root 1, 5 Jan 26 2006 /dev/zero
```

3.2.2 Block devices

They are associated to hard drives, hard disk controllers, etcetera. This particular case of device drives work always using intermediate buffers, and the reads or writes are always done by byte-blocks. Internally, there are a lot of

differences but, as far as the user is concerned, either char or block devices seem quite similar.

They can be recognised inside /dev directory by the character "b":

```
brw-rw---- 1 root disk 8, 0 Feb 17 08:11 /dev/sda
brw-rw---- 1 root disk 8, 1 Jan 31 2005 /dev/sda1
```

3.2.3 Major and minor numbers

The GNU/Linux Kernel organises all device drivers through major and minor numbers. As a matter of fact, this is the way for identifying a particular device driver from a Kernel's point of view. The file name inside /dev directory can be changed, obviously.

The major number means the class or group for this particular device driver, such as a Hard disk. Thanks to it, the GNU/Linux Kernel, as soon as an open() system call is called by a running process, knows which driver will be able to dispatch it.

For example, the SATA hard disk sda belongs to family with major number of 8:

```
brw-rw---- 1 root disk 8, 0 Mar 13 08:43 /dev/sda
```

The minor number is directly related to the major one. Allows to the GNU/Linux Kernel to fine-tune which particular device will respond to the call for that particular device driver family. For example, in the case of a hard disk, it could be the first disk-partition, such a /dev/sda1:

```
brw-rw---- 1 root disk 8, 0 Mar 13 08:43 /dev/sda1
brw-rw---- 1 root floppy 8, 33 Mar 13 08:43 /dev/sdc1
```

4 Exported Symbols

The GNU/Linux Kernel belongs to Microkernel family. So, some layers can be stacked on top of other ones, using some functions previously implemented in lower layers. Thanks to it, an LKM developer - or just a Kernel developer - can re-use some well-known tested code or functions inside the Kernel. Clearly, this matter presents a real layer-dependancy issue. As soon as some Kernel layer needs some aids implemented (offered) by another one, there must exist a total agreement between these functionalities in order to avoid erroneous behaviours, like Kernel OOPS, unknown symbol messages, and so on (see below).

4.1 What is a symbol?

A symbol can be a function, a variable, a data structure, etc. For example, do_lookup function - analysed in section Kernel OOPS-, is a symbol. This symbol is an exported one, too. That is, any external layer from where it is defined, can use it.

In order to check if this symbol is really exported, we can read /proc/kallsyms file. This file stores all Kernel's exported symbol table, placed in memory between __start_ksymtab and _stop_ksymtab. The field's format is like nm's output.

```
# cat /proc/kallsyms |grep do_lookup
c0164a7f t do_lookup
```

```
 ...
c0102b7f t check_userspace
c0102b94 T resume_userspace
c0102bac T sysenter_entry
c0102bb3 t sysenter_past_esp
c0102c2c T system_call
c0102c55 t no_singlestep
c0102c6c t syscall_call
c0102c77 t syscall_exit
c0102c86 t restore_all
c0102c9e t restore_nocheck
c0102c9e t restore_nocheck_notrace
...
```

Figure 8: The Kernel's exported symbol table cut short

Yes, it exists! We can find it at kernel's address 0xc0164a7f, and the "t" field denotes it is a local symbol stored in the text segment.

4.2 A peek inside an LKM object file

All modules have their own symbols, of course. These symbols can be exported or not, depending on the design or the concept in with this module is involved. Suppose we have a module, say kmodest.ko, with a lot of functions, variables, data structures and so on. After compiling it and obtaining the module's object file, we can get a list of its symbols by running nm:

```
# nm kmodest.ko
00000004 B bytes_to_read
00000000 T cleanup_module
U copy_from_user
U copy_to_user
00000000 D current_cmd
U do_fsync
00000557 T do_insert_fd
000003b2 T do_remap_fd
0000043c T do_restore_task
0000028c T fd_deinstall_by_task
000002f6 T fd_install_by_task
00000020 B fds_info
0000003d T fill_file_information
...
```

As shown in the previous symbol list, we can realize that some symbols are undefined (U). These symbols must be exported in order to use this module. Thus, the copy_from_user, copy_to_user and do_fsync symbols will be called from this module. If these ones will not available at Kernel's exported symbol table, this module cannot be loaded through insmod or modprobe at all.

4.3 Unknown symbols

The undefined symbols can become as unknown ones. For example, in this particular case:

```
# insmod ./kmodest.ko
     insmod:  error inserting './kmodest.ko':  -1 Unknown symbol in module
# dmesg|tail -3
     kmodest:  Unknown symbol do_fsync
     kmodest:  Unknown symbol sys_kill
     kmodest:  Unknown symbol get_files_struct
```

When there are unknown symbols reported by insmod, we can fix them probably loading a previous module using modprobe. Sometimes, either, we need to patch the Kernel sources in order to export these symbols explicitly. For example, in order to solve the previous issue, we need the current Kernel sources. With cscope [9] we can find out where this non-exported functions are and export them by adding the macro EXPORT_SYMBOL.

Finally, we have to recompile the kernel and, sometimes, reboot the system in order to load the new one. We can take a look at /proc/kallsyms always, in order to test if a certain symbol has been exported. Bear in mind that some symbols will be exported as soon as a certain chain of modules will be linked into the runtime Kernel via modprobe.

```
...
asmlinkagelong sys_kill(int pid, int sig)
          ...
          ...
}
EXPORT_SYMBOL(sys_kill);
```

Figure 9: Exporting sys_kill symbol by hand

5 Dealing with Kernel OOPS and Kernel PANIC messages

When a Kernel OOPS message appears, it is possible to continue using the system. But when a critical error happens, the Kernel PANIC message appears, hanging up the system completely. Depending on its configuration, this Kernel Panic message can be followed, after some seconds, in a system reboot.

[9] A tool designed for browsing inside C source code files, such as GNU/Linux Kernel.
It is feasible to run, inside the GNU/Linux Kernel source directory, this command:
`# make cscope`
After a while, we can browse the sources in a comfortable way by running:
`# cscope -k`

```
(...)
Pid:  3378, comm:  find Not tainted (2.6.24-etchnhalf.1-686 #1)
EIP: 0060:[<c01875d5>] EFLAGS: 00000286 CPU: 1
EIP is at __d_lookup+0xbe/0xd9
EAX: dbc649e4 EBX: dbc649d4 ECX: 00000011 EDX: c17e2100
ESI: efbcdf04 EDI: dbc649e4 EBP: d9b347c8 ESP: efbcddc4
( ...  )
       [<c017ec3a>] do_lookup+0x24/0x14e
       [<c0180869>] __link_path_walk+0x73f/0xb46
       [<c0180cb4>] link_path_walk+0x44/0xb3
       [<c0180f9c>] do_path_lookup+0x162/0x1c4
       [<c017ff66>] getname+0x59/0xad
       [<c018176d>] __user_walk_fd+0x2f/0x40
       [<c017b6ae>] vfs_lstat_fd+0x16/0x3d
       [<c017b742>] sys_lstat64+0xf/0x23
       [<c0103ede>] syscall_call+0x7/0xb
       [<c02b0000>] skb_icv_walk+0x21e/0x262
(...)
```

Figure 10: Kernel OOPS in __d_lookup() function

5.1 Kernel OOPS

This kind of error messages appear as soon as there's a NULL de-reference pointer. Generally, all information reported by the GNU/Linux Kernel can be analysed in order to make out where the problem causing this OOPS is. All data is written in the log files through syslog, and this message is printed out thanks to printk() Kernel's function - so similar to printf in user-land development -.

Normally, it is feasible to continue using the system. But, sometimes, the OOPS message points to a hardware error, like corrupted memory DIMMs. Obviously, the offending process is killed immediately.

5.1.1 A Kernel OOPS case-study

Let's take a look at the Kernel OOPS screen in Figure 10, showing up a real issue occurred to a Linux Box running Debian Etch "and-a-half", X86 architecture.

Despite the fact there are a lot of "strange" messages in here, the most important ones are, basically, the EIP register, the offending process and the call-trace. So, first of all, we have to write down this basic information gathered from the Kernel OOPS message:
Offending process: find command, PID 3378.
Current instruction executed: __d_lookup(), at offset 0xbe.
Call-trace: look at the Figure.

Well, it makes sense, doesn't it? The "find" command, with PID 3378, fires a Kernel OOPS message while running in Ring 0 the __do_lookup() function, just inside 0xbe offset. Checking out the call-trace, it seems so related to the "find" command executed: all Kernel functions executed here have been designed for walking inside a file-system through the VFS (Virtual File-System layer, shown

in Figure 1).

Generally, a find command does not fires a Kernel OOPS message. It's quite strange, so probably the find command isn't really involved in this awful matter at all. It seems to us more related to some problem inside the GNU/Linux Kernel, maybe.

Thus, we need to analyse the _do_lookup function. We need the current Kernel sources in order to do that, of course! Using cscope , we find out this function declared in include/linux/dcache.h header file. Here is its signature:

```
extern struct dentry * __d_lookup(struct dentry *, struct qstr *);
```

A dentry structure stores all about a file or directory inside the file system walked by the VFS. This function simply looks for the dentry's parent transfered as a parameter and returns a pointer in case of finding it. The GNU/Linux Kernel always works with directories or files using dentry objects, directly mapped to the memory, in order to improve time-consumption. These kind of dentry objects are well-known as dcache typedefs.

In short, we know that find command, while searching for a some parent directory, fired a Kernel OOPS message. How was it possible? Clearly, because of some physical error memory. If all dcache/dentry objects are keep in memory for improving system troughtput, and the Kernel OOPS was saying : Not tainted, then there was an invalid memory access. Why? It could be a BUG. Why not? But, as a matter of fact, the system was running perfectly for a long time, so ... the answer should be related to physical DIMM memory errors. The action taken in order to determine the real memory error cause was running a trivial memory tester (memtest86+). After a bit, the first memory errors appeared. What a relief ;-) !!

5.2 Kernel PANIC

A Kernel PANIC message means there's nothing the system can do in order to recover the minimal stability and is in need of an urgent reboot. Sometimes it is in need of a complete shutdown of any attached hardware, too - such as an external Ultra-SCSI disk controller -.

When this message appears along the runtime life of the GNU/Linux Kernel program, there's some hardware related error or an undocumented BUG. When it occurs at boot time, it can be related to a not linked driver - such as ahci.ko SATA controller, or a not known file-system due to some missing module.-

In order to reboot the system after n seconds just after appearing the Kernel PANIC message, it is necessary to configure this timeout in /etc/sysctl.conf:

```
kernel.panic=10
```

It is feasible to change this behaviour at runtime typing:

```
# echo n[10] > /proc/sys/kernel/panic
```

Otherwise, the system will remain powered on and hanged up forever.

```
VFS: Cannot open root device "hda1" or unknown-block(0,0)
Please append a correct "root=" boot option
Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(0,0)
```

Figure 11: Not loaded Hard disk controller

[10] A 0 value means no reboot

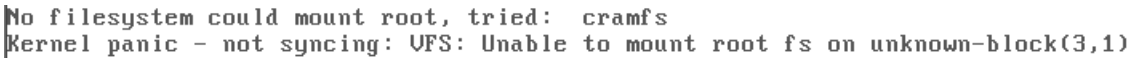

Figure 12: Not file-system support for mounting the root partition

5.3 Overriding the init program

Sometimes we need to restore root's password we have forgotten. To do this, we must pass a parameter to the GNU/Linux Kernel through the Boot Loader: init=/bin/bash. As soon as we can access to the bash prompt, we must remount the slash partition in order to allow write access: mount / -o remount,rw. Finally, we can use the passwd command as usual.

Glossary

cpio — A file-format similar to tar, all files packed in one., 6

EM64T architecture — 64 bit architectures, Intel Core processors., 5

GNU/Linux distro(s) — A recopilation of tools, software and a GNU/Linux Kernel in one package, 5

Kernel Image — The entire GNU/Linux Kernel program, g-zipped. It will be uncompressed at runtime, in memory., 5

layer — A sub-system in charge of doing some tasks. For example, the network layer., 3

N-Curses libraries — Libraries designed for building applications running inside Terminals with more or less Graphical User Interface capabilities., 5

X86 architectures (a.k.a i386) — 32 bit architectures, AMD or Intel processors., 5