Linux Kernel Tester's Guide

(version 0.3)

Michal Piotrowski
michal.k.k.piotrowski@gmail.com

Coauthors:

Maciej Rutecki
maciej.rutecki@gmail.com
Unixy.pl

Rafael J. Wysocki
rjw@sisk.pl

English translation by

Rafael J. Wysocki
rjw@sisk.pl

Contributors:

Bartlomiej Zolnierkiewicz	bzolnier@gmail.com
Jarek Poplawski	jarkao2@o2.pl
Mariusz Kozlowski	m.kozlowski@tuxland.pl

Spis treści

Introduction

The testing of software plays a very important role in the process of its development, since potentially every new piece of software contains bugs. Some of the bugs are just simple mistakes, like typos or omissions, that can be spotted quite easily by the developer himself or by the reviewers of his code. However, there are bugs resulting from wrong assumptions made by the developer and usually they can only be found by running the software on a computer in which these assumptions are not satisfied. This is particularly true with respect to operating systems that often are developed under assumptions following from the observed but undocumented behavior of hardware. It is therefore important to test them at the early stage of development on as many different machines as possible in order to make sure that they will work correctly in the majority of practically relevant cases.

The purpose of the present guide is to acquaint the reader with the testing of the Linux kernel. It is divided into several short chapters devoted to the more important issues that every good tester should be familiar with. Still, we do not try to describe very thoroughly all of the problems that can be encountered during the kernel testing, since that would be boring. We also want to leave some room for the reader's own discoveries.

Why is it a good idea to test the Linux kernel?

> "Now, in my opinion, we should test it, if we want to be sure that the next versions of the kernel will correctly handle our hardware and will do what we need them to do. In other words, if we seriously want to use Linux, then it is worth spending some time checking if the next version of the kernel is not going to give us trouble, especially that recently the kernel has been changing quite a lot between consecutive stable releases (which is not even realized by the majority of its users). In fact, this is the responsibility of an Open Source user: you should test new versions and report problems. If you do not do this, then later you will not have the right to complain that something has stopped working.
>
> Of course, our testing will also benefit the other users, but we should not rather count on someone else to check whether or not the new kernel will work on *our* hardware."
>
> – Rafael J. Wysocki

Unfortunately, many Linux users tend to think that in order to test the system kernel you should be an expert programmer. Yet, this is as true as the statement that the pilots who test new airplanes should be capable of designing them. In fact, the ability to program computers is very useful in carrying out the tests, because it allows the tester to assess the

situation more accurately. It also is necessary for learning the kernel internals. Still, even if you cannot program, you can be a valuable tester.

In most cases, the testing of the Linux kernel is as simple as downloading the tarball containing its sources, unpacking it, configuring and building the kernel, installing it, booting the system and using it for some time in a usual way. Of course it can quickly get complicated as soon as we trigger a kernel failure, but this is where the interesting part of the story begins.

Certainly, you should be able to distinguish kernel failures from problems caused by user space processes. For this purpose it is quite necessary to know how the kernel is designed and how it works. There are quite a few sources of such information, like the books *Understanding the Linux Kernel* by Daniel Bovet and Marco Cesati (`http://www.oreilly.com/catalog/understandlk/`), *Linux Device Drivers* by Jonathan Corbet, Alessandro Rubini and Greg Kroah-Hartman (`http://lwn.net/Kernel/LDD3/`) or *Linux Kernel Development* by Robert Love (`http://rlove.org/kernel_book/`) (the list of all interesting kernel-related books is available from the *KernelNewbies* web page at `http://kernelnewbies.org/KernelBooks`). Very useful articles describing the design and operations of some important components of the Linux kernel can be found in the web pages of *Linux Weekly News* (`http://lwn.net/Kernel/Index/`). Still, the ultimate source of information on the kernel internals is its source code, although you need to know the C language quite well to be able to read it.

At this point you may be wondering if it is actually safe to use development versions of the kernel, as many people tend to think that this is likely to cause a data loss or a damage to your hardware. Well, this is true as well as it is true that if you use a knife, you can lose your fingers: you should just pay attention when you are doing it. Nevertheless, if your data are so important that you cannot afford to lose them in any case, you can carry out the kernel tests on a dedicated system that is not used for any other purposes. For example, it can be installed on a separate disk or partition, so that you do not have to mount any partitions from a "stable" system while a new kernel is being tested. It also is a good idea to regularly backup your data, regardless of whether the system is a test bed one or it is considered as "stable".

Część I

Rudiments

Rozdział 1

The kernel, patches, trees and compilation

1.1 The kernel

The current version of the Linux kernel can always be downloaded from *The Linux Kernel Archives* web page (`http://www.kernel.org/`) in the form of a large `tar` file compressed with either `gzip` or `bzip2`, which is called the kernel *tarball* (it is worthy of noting that the `bz2` files are smaller than the `gz` ones, but `gzip` is generally faster than `bzip2`).

After downloading the tarball you can extract the kernel's source code from it in many different ways (the virtue of UNIX-like systems). Our favorite method is to run either

```
$ tar xjvf linux-2.6.x.y.tar.bz2
```

for the tarballs compressed with `bzip2`, or

```
$ tar xzvf linux-2.6.x.y.tar.gz
```

for the ones compressed with `gzip`, in the directory that we want to contain the kernel sources. In principle, it can be an arbitrary directory, but it should be located on a partition with as much as 2 GB of free space.

Of course, you can also follow the kernel's `README` literally and extract the source code by executing one of the following instructions:

```
$ bzip2 -dc linux-2.6.x.y.tar.bz2 | tar xvf -
$ gzip -cd linux-2.6.x.y.tar.gz | tar xvf -
```

depending on what kind of a tarball you have downloaded. In any case it is not recommended to unpack the kernel tarball as `root`.

As a result of unpacking the kernel tarball we get a directory the name of which corresponds to the version of the kernel that will be obtained by compiling the downloaded source code (eg. `linux-2.6.18`). The contents of this directory are often referred to as the *kernel tree*. For this reason the word "tree" is used for naming different development branches of the kernel. For instance, instead of saying "the development branch of the kernel maintained by Andrew Morton" it is more convenient to say "the -mm tree".

1.2 Patches

It often is necessary to update the kernel source code, but you should not download the entire tarball every time, since that would be bandwidth-consuming and tedious. Usually, it is sufficient to download a patch that can be applied to an existing kernel tree from within the directory in which the tree is located. For example, if the patch in question has been compressed with `bzip2`, this can be done in one of the following ways:

```
$ bzip2 -cd /path/to/patch-2.6.x.bz2 | patch -p1
$ bzcat /path/to/patch-2.6.x.bz2 | patch -p1
```

If the patch has been compressed with `gzip`, you can apply it analogously:

```
$ gzip -cd /path/to/patch-2.6.x.gz | patch -p1
$ zcat /path/to/patch-2.6.x.gz | patch -p1
```

and for uncompressed patches you can use `cat` instead of either `zcat` or `bzcat`. Of course, these are only the simplest methods of applying patches and it is quite easy to invent some more interesting ones.

It is very useful to be able to revert patches that are no longer needed (or buggy). It can be accomplished by adding the `-R` flag to the `patch` command line:

```
$ bzcat /path/to/patch-2.6.x.bz2 | patch -p1 -R
```

It is also possible to check if the patch will apply cleanly (ie. it does not contain any pieces of code that will be rejected) with the help of the `--dry-run` command-line option of `patch`:

```
$ bzcat /path/to/patch-2.6.x.bz2 | patch -p1 --dry-run
```

Then, if there are no messages similar to

```
1 out of 1 hunk FAILED -- saving rejects to file xxx
```

the patch can be safely applied.

Most of different versions of the kernel source code are available as sets of patches (patch-sets) or even as individual patches that should be applied on top of specific kernel trees. For this reason the patches are labeled in reference with the kernel trees to which they should be applied. Namely, since stable kernel versions are labeled as `2.6.x` or `2.6.x.y`, development trees are called `2.6.x-git*`, `2.6.z-rc*` and `2.6.z-rc*-git*`, where the -rc patch with `z` equal to `x` plus one should be applied on top of the stable kernel `2.6.x`. This means, for example, that the patch called `patch-2.6.21-rc3.bz2` should be applied on top of the `linux-2.6.20` kernel tree and the patch labeled as `patch-2.6.21-rc3-git3.bz2` should be applied on top of `patch-2.6.21-rc3.bz2`.

The following example will hopefully make everything clear. Assume that we have the stable kernel source code labeled as `linux-2.6.16.25.tar.bz2` and we want to obtain the development tree `2.6.20-rc1-git1`. First, we unpack the tarball:

```
$ tar xjvf linux-2.6.16.25.tar.bz2
```

Next, we fetch the patch called `patch-2.6.16.25.bz2` from `ftp://ftp.kernel.org` and revert it:

```
$ bzip2 -cd /path/to/patch-2.6.16.25.bz2 | patch -p1 -R
```

Now, we have the sources of the `2.6.16` kernel in our work directory. This is what we wanted, since the stable and development patches only apply cleanly to the `2.6.x` kernel trees, as they can contain some code that is also included in the `2.6.x.y` versions. Accordingly, we can apply `patch-2.6.17.bz2`, `patch-2.6.18.bz2` and `patch-2.6.19.bz2` to our tree:

```
$ bzip2 -cd /path/to/patch-2.6.17.bz2 | patch -p1
$ bzip2 -cd /path/to/patch-2.6.18.bz2 | patch -p1
$ bzip2 -cd /path/to/patch-2.6.19.bz2 | patch -p1
```

to obtain the `2.6.19` kernel source code. Finally, we need to download two development patches, `patch-2.6.20-rc1.bz2` and `patch-2.6.20-rc1-git1.bz2`, and apply them in the order specified.

Isn't that easy? Well, it is, but it also is tedious and that is why some people who do not like to lose time use the tool called `ketchup` (`http://www.selenic.com/ketchup/`), described in the next section.

It should be noted that patches can only be applied if they are properly formatted. If you receive them via email or download them directly from a mailing list archives, the formatting is often changed in a way that makes them unusable. In such a case `patch` will refuse to apply the patch and you will see a message like "`patch: **** malformed patch at line`". For instance,

```
$ cat ../sched-2.patch | patch -p1 -R --dry-run
patching file kernel/sched.c
patch: **** malformed patch at line 33:         if (next == rq->idle)
```

means that the patch called `sched-2.patch` is damaged and cannot be reverted. Fortunately, if the patch has been posted to the *Linux Kernel Mailing List* (LKML), it can be downloaded from the archives at `http://marc.theaimsgroup.com/?l=linux-kernel` in the original form. For this purpose you only need to find the message that contains the patch and click on the link "Download message RAW". Still, even this will not work if the original "raw" message containing the patch is malformed.

For more information on downloading and applying patches please see the files `README` and `Documentation/applying-patches.txt` in the kernel sources.

1.3 Ketchup

The `ketchup` tool allows us to download any version of the kernel source code quickly and easily as well as to transform given kernel tree into another one. It does this quite intelligently, checking the current version and downloading only the required patches.

The users of Debian testing/unstable can simply install the `ketchup` package with the help of their favorite tool, eg.

```
# apt-get install ketchup
```

Still, for the users of Debian stable or another distribution in which the `ketchup` tool is not available the easiest way of installing it is to do:

```
$ mkdir ~/bin
```

unless the subdirectory `bin` is already present in the user's home directory, and

```
$ cd ~/bin
$ wget http://www.selenic.com/ketchup/ketchup-0.9.8.tar.bz2
$ tar xjvf ketchup-0.9.8.tar.bz2
```

It is also necessary to install the `python` package.

You should remember that `ketchup` is able to check the signatures of the downloaded patches. For this reason it is recommended to visit the web page at `http://www.kernel.org/signature.html` and save the current key in a text file (we have never managed to make `$ gpg --keyserver wwwkeys.pgp.net --recv-keys 0x517D0F0E` work on our systems). Next, you can do

```
$ gpg --import the_file_containing_the_kernel.org_key
```

and you are ready to check how the whole thing works. Namely, you can do

```
$ mkdir ~/tree
$ cd ~/tree
$ ketchup -m
```

and you should get an error message similar to the following one:

```
Traceback (most recent call last):
  File "/usr/bin/ketchup", line 695, in ?
    lprint(get_ver("Makefile"))
   File "/usr/bin/ketchup", line 160, in get_ver
     m = open(makefile)
 IOError: [Errno 2] No such file or directory: 'Makefile'
```

This only means that there is no Linux kernel tree in the current directory. You can order `ketchup` to download the latest "stable" version of the kernel by using the command "`ketchup 2.6-tip`":

```
$ ketchup 2.6-tip
None -> 2.6.16.1
Unpacking linux-2.6.15.tar.bz2
Applying patch-2.6.16.bz2
Downloading patch-2.6.16.1.bz2
--21:11:47--  http://www.kernel.org/pub/linux/kernel/v2.6/patch-2.6.16.1.bz2
           => `/home/michal/.ketchup/patch-2.6.16.1.bz2.partial'
Resolving www.kernel.org... 204.152.191.5, 204.152.191.37
```

```
Connecting to www.kernel.org|204.152.191.5|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 5,284 (5.2K) [application/x-bzip2]

100%[====================================>] 5,284          6.19K/s

21:11:49 (6.18 KB/s) - '/home/michal/.ketchup/patch-2.6.16.1.bz2.partial'
saved [5284/5284]

Downloading patch-2.6.16.1.bz2.sign
--21:11:49--  http://www.kernel.org/pub/linux/kernel/v2.6/
patch-2.6.16.1.bz2.sign
           => '/home/michal/.ketchup/patch-2.6.16.1.bz2.sign.partial'
Resolving www.kernel.org... 204.152.191.37, 204.152.191.5
Connecting to www.kernel.org|204.152.191.37|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 250 [application/pgp-signature]

100%[====================================>] 250           --.--K/s

21:11:49 (18.34 MB/s) - '/home/michal/.ketchup/
patch-2.6.16.1.bz2.sign.partial' saved [250/250]

Verifying signature...
gpg: Signature made wto 28 mar 2006 09:37:31 CEST using DSA key ID 517D0F0E
gpg: Good signature from "Linux Kernel Archives Verification Key
 <ftpadmin@kernel.org>"
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: C75D C40A 11D7 AF88 9981  ED5B C86B A06A 517D 0F0E
Applying patch-2.6.16.1.bz2
```

Now you can check if you really have the latest stable tree:

```
$ ketchup -m
2.6.16.1
```

As you can see, `ketchup` automatically downloads the necessary patches. It can also take patches from the `~/.ketchup` subdirectory of the user's home directory, so it is a good idea to put the patches that you already have in there.

Making a transition to another kernel tree is absolutely not a problem for `ketchup`. For this purpose you can simply use the command "`ketchup kernel_version`", eg.

```
$ ketchup 2.6.16-rc4-mm1
2.6.16.1 -> 2.6.16-rc4-mm1
Applying patch-2.6.16.1.bz2 -R
```

```
Applying patch-2.6.16.bz2 -R
Applying patch-2.6.16-rc4.bz2
Applying 2.6.16-rc4-mm1.bz2
```

Similarly, if you want to get the latest kernel version from the -rt branch, you can do

```
$ ketchup 2.6-rt
```

and the tool will automatically apply or revert the patches for you:

```
2.6.16-rc4-mm1 -> 2.6.16-rt12
Applying 2.6.16-rc4-mm1.bz2 -R
Applying patch-2.6.16-rc4.bz2 -R
Applying patch-2.6.16.bz2
Applying patch-2.6.16-rt12
```

It is also possible to teach `ketchup` to use patches included in the latest -mm tree (described below), which can be done in the following way:

```
$ wget -c ‘ketchup -u 2.6-mm | sed "s/.bz2/-broken-out.tar.bz2/"‘
```

As follows from the above examples, `ketchup` can recognize several development branches of the Linux kernel source code. In particular, it can handle the following trees:

- 2.6-tip – the latest "stable" kernel
- 2.6-rc – the latest "release candidate" kernel
- 2.6-git – the latest development -git kernel
- 2.6-rt – the Ingo Molnar's "real-time" tree
- 2.6-mm – the Andrew Morton's tree
- 2.6-ck – the Con Kolivas' tree (desktop)
- 2.6-cks – the Con Kolivas' tree (server)

Now, you may be wondering why we have shown you all of the commands used for applying and reverting patches. The reason is really simple: if you find and report a bug, you will probably get a fix in the form of a patch and you should be able to correctly apply it.

1.4 Trees

We have already used the term "kernel tree" for quite a few times and we have told you that it means "the contents of the directory containing the kernel source code". However, it also has another meaning which is "a development branch of the kernel source code". Usually, such a branch is equivalent to a patchset that can be applied on top of a specific version of the kernel sources and different branches are named differently, so that they can be easily distinguished from each other. The more popular branches, also referred to as "trees", are the following:

- The -mm tree (maintained by Andrew Morton), which is built from other trees (ie. development branches of the kernel) and from many individual, often experimental, patches.
- The -rt (real-time) tree (maintained by Ingo Molnar), which contains patches that turn Linux into a real time system.
- The -ck tree (maintained by Con Kolivas), focused on the improvements of the system's performance.

Many developers and the majority of kernel subsystems maintain their own trees used in the development process. After some time specific parts of these trees are included into the main kernel tree maintained by Linus Torvalds, called the *mainline*. The -git patches discussed in the previous sections represent the snapshots of the ever-evolving Linus' tree taken at various instants of time.

1.5 The -mm tree

We have already stated that the -mm tree is a combination of some other trees and individual experimental patches. More importantly, it also is the main battlefield of the fight against bugs in the Linux kernel.

The importance of this tree stems from the fact that it combines the mainline with multiple trees used by the maintainers of kernel subsystems (subsystem trees) including patches considered as experimental and not ready for merging with the mainline. In other words, each release of the -mm tree is sort of a *future* snapshot of the mainline with some additional changes and with some bugs that (hopefully) are going to be fixed before the patches containing them will get merged. Thus by testing the current -mm kernel we check how the future mainline (ie. stable) kernels may behave and if any bugs are found at this stage, they can be fixed before hitting the mainline.

Still, some patches reach the mainline without appearing in the -mm tree, which is a consequence of the flow of changes in the kernel source code. Namely, the majority of patches merged with the mainline comes from different subsystem trees and they are included in the Linus' tree when he decides to pull the patches considered as ready for the inclusion from given subsystem tree. It is possible, and it sometimes really happens, that Linus pulls the "ready to go" patches from certain subsystem tree before they have a chance to appear in -mm (in that case the patches will appear in the -mm tree anyway, but they will come into it from the mainline rather than from the subsystem tree). For this reason it is not sufficient to test the -mm kernels alone and the development -rc or -git kernels should be tested as well.

On the other hand, there are kernel subsystems that do not have their own development trees and many patches are sent directly to Andrew Morton. These patches are first included in the -mm tree and then they go either to one of the subsystem trees or to the mainline. Thus the -mm tree is a place in which the patches coming from many different sources first meet together and are combined into an approximation of a future stable kernel that can be tested.

Since the -mm tree is a combination of many other trees, including the mainline, that always evolve, it really should be regarded as a set of patches that can be put together from

time to time. When this happens, Andrew releases the so-called -mm patches to be applied on top of specific -rc or stable kernel trees. Each of them is also available as a series of individual patches combined in order to obtain it, including some fixes introduced by Andrew himself. This makes the identification of patches that contain bugs easier, as they can be found using the `quilt` tool with the help of the bisection technique (for more information about it see Chapter 4).

In the directory containing the -mm patch and the series of individual patches it consists of (at `ftp://ftp.kernel.org/pub/linux/kernel/people/akpm/patches/2.6/2.6.X-rcY/2.6.X-rcY-mmZ/` or at `ftp://ftp.kernel.org/pub/linux/kernel/people/akpm/patches/2.6/2.6.X/2.6.X-mmZ/`, where `X`, `Y`, `Z` are numbers) there always is a directory called `hot-fixes` containing the patches that fix known bugs in this version of the -mm tree and should be applied on top of the corresponding -mm patch before testing it (the bugs that have already been identified need not be found once again).

The individual patches included in the -mm tree are sent to the *mm-commits* mailing list (`http://vger.kernel.org/vger-lists.html#mm-commits`). By following this list you can get an idea about the current contents of the -mm tree. It may also help you identify patches that introduce bugs.

From time to time Andrew sends to *mm-commits* a message with the subject like "*mm snapshot broken-out-date-time.tar.gz uploaded*" announcing that new -mm tree snapshot has been made available from the server. These snapshots are considered as less usable than the regular -mm patches and are to be used by people who want to help Andrew put the next -mm patch together.

1.6 Compilation and installation

1.6.1 Kernel compilation

First, your system should be prepared for the configuration and compilation of the kernel. You will need `gcc`, `make`, `binutils`, `util-linux` and the package containing the `ncurses` header files (`libncurses5-dev` in Debian or `ncurses-devel` in Fedora Core and OpenSUSE). If you want to use a graphical tool for configuring the kernel, you will also need the *Qt* library or *GTK+* and the corresponding development packages. The file `Documentation/Changes` in the kernel sources directory contains the list of all programs that you may need.

Now, we assume that you have already downloaded the kernel source code and patched it to obtain the required version. To compile it, change the current directory to the one that contains the kernel sources and run the following sequence of commands:

- `make menuconfig` (allows you to adjust the kernel configuration to suit you; there are many documents on the web describing the configuration of the kernel, one of them you can find at `http://www.tlug.org.za/old/guides/lkcg/lkcg_config.html`)
- `make` (starts the compilation of the kernel; if you have more than one processor core in your machine, you may want to use the `-j` option of `make`, for example "`make -j5`")
- `make modules_install` (installs the kernel modules in the `/lib/modules/` directory, creating the subdirectory named after the version of the newly compiled kernel)

- `cp arch/i386/boot/bzImage /boot/vmlinuz-<kernel_version>` (copy the kernel to the `/boot` directory; `<kernel_version>` should be the label reflecting the version of the kernel being installed)

- `cp System.map /boot/System.map-<kernel_version>` (copy the map of kernel symbols to the `/boot` directory)

Instead of "`make menuconfig`" you can use "`make config`" that will cause the kernel build system to ask you *a lot* of questions regarding its configuration, which is inefficient and tedious, but can be done without `ncurses`.

If you have already configured the kernel, you can use the configuration file `.config` from the previous compilation. For this purpose copy it to the directory containing the current kernel sources and run "`make oldconfig`" (it works like "`make config`", but does not ask so many questions). Alternatively, you can use the configuration of the distribution kernel (usually it contains *many* things you will never need and that is one of the reasons why you may want to learn how to configure the kernel):

```
$ zcat /proc/config.gz > .config
$ make oldconfig
```

1.6.2 Useful `make` options

There are some useful options you can pass to `make` during the kernel compilation:

- `make O=/directory/` (saves the result of the compilation in given directory; this is particularly useful if you do not want to litter the kernel source directory with `*.o` files, but in that case you should also use "`make O=/directory/ menuconfig`" instead of "`make menuconfig`" and "`make O=/katalog/ modules_install`" etc.)

- `make CC=<compiler>` (allows you to specify the name of the compiler to use; for example, in Debian there are several versions of `gcc`, and if you want to use `gcc-3.4` to compile the kernel it is sufficient to use "`make CC=gcc-3.4`")

- `make C=1` (before compilation the kernel sources are checked by the `sparse` tool)

- `make -j<number>` (sets the number of parallel processes that will be run during the kernel compilation)

- `make xconfig` (graphical configuration using *Qt*)

- `make gconfig` (graphical configuration using *GTK+*)

1.6.3 Kernel modules

If you want to use kernel modules, in which case some parts of the kernel will be loaded into memory only if they are needed, select the options:

```
Loadable module support  --->
[*] Enable loadable module support
[*]   Automatic kernel module loading
```

The modularization allows the kernel to be smaller and work faster.

To compile a piece of the kernel code as a module, you need to mark it with `M` during the configuration, eg.

```
<M>   Check for non-fatal errors on AMD Athlon/Duron / Intel Pentium 4
```

If the "`Automatic kernel module loading`" option has been selected, the modules should be automatically loaded by the kernel when they are needed. That is, for example, if you want to mount a CD-ROM, the kernel should automatically load the module `isofs.ko` and this modules can be removed from memory after the CD-ROM has been unmounted. You can also load and unload kernel modules manually, using the commands `modprobe`, `insmod` or `rmmod`, eg.

- `modprobe isofs` (loads the module `isofs.ko`)
- `modprobe -r isofs` (removes `isofs.ko` from memory)

All of the modules available to the currently running kernel should be located in various subdirectories of `/lib/modules/<kernel_version>/kernel` .

1.6.4 Kernel hacking options

If you build the kernel to test it, you should seriously consider setting the options located in the "`Kernel hacking`" menu. At least, you should set:

```
[*] Kernel debugging
[*]   Compile the kernel with debug info
```

The other options should also be set, if possible, because they help debug specific functionalities of the kernel, but some of them hurt performance, so you may want to leave them unset:

```
[*]   Debug shared IRQ handlers
[*]   Detect Soft Lockups
[*]   Debug slab memory allocations
[*]     Slab memory leak debugging
[*]   RT Mutex debugging, deadlock detection
[*]   Built-in scriptable tester for rt-mutexes
[*]   Lock debugging: prove locking correctness
[*]   Lock dependency engine debugging
[*]   Locking API boot-time self-tests
[*]   Highmem debugging
[*]   Debug VM
```

1.6.5 Magic SysRq

The option

```
[*] Magic SysRq key
```

is particularly useful, because it often allows you to reduce the impact of a kernel failure and to obtain some additional information about the state of the system. It turns on some keyboard shortcuts allowing you to pass specific instructions directly to the kernel:

- Alt + SysRq + b – reboot the system immediately, without unmounting filesystems and writing the contents of disk buffers to the storage (DANGEROUS)
- Alt + SysRq + c – crashdump
- Alt + SysRq + e – send the TERM signal to all processes except for INIT
- Alt + SysRq + i – send the KILL signal to all processes except for INIT
- Alt + SysRq + k – SAK (Secure Access Key); kill all processes bound to the currently active virtual console (eg. emergency shutdown of the X server when Ctrl + Alt + Backspace does not work)
- Alt + SysRq + l – send the KILL signal to all processes including INIT
- Alt + SysRq + m – print information about the state of memory
- Alt + SysRq + o – system shutdown
- Alt + SysRq + p – print the contents of CPU registers and flags
- Alt + SysRq + r – switch the keyboard to the RAW mode (eg. when Ctrl + Alt + Backspace does not work)
- Alt + SysRq + s – synchronize filesystems and write the contents of disk buffers to the storage
- Alt + SysRq + t – print the list of all tasks
- Alt + SysRq + u – remount all filesystems read-only
- Alt + SysRq + h – print help

Which key is the SysRq? That depends on the architecture of your machine:

- x86 – Print Screen
- SPARC – STOP
- serial console – Break
- PowerPC – Print Screen (or F13)

On all architectures you can trigger the action assigned to given key (say `t`) by echoing the corresponding character to the file `/proc/sysrq-trigger`, eg.

```
# echo t > /proc/sysrq-trigger
```

What is the safest way to restart the computer?

First, you should try to press the combination of Ctrl + Alt + Backspace (X Window system) or Ctrl + Alt + Del (text console). If it does not work, you can:

- press Alt + SysRq + k to kill all processes using the current console,
- press Alt + SysRq + s, Alt + SysRq + u, Alt + SysRq + b

More information related to the "`Magic SysRq key`" mechanism can be found in the file `Documentation/sysrq.txt` included in the kernel sources.

1.6.6 Installation

Some distributions allow you to install a newly compiled kernel by running

```
# make install
```

as `root` from within the directory that contains the kernel sources. Usually, however, you will need to change the boot loader configuration file manually and if you use `lilo`, you will need to run `/sbin/lilo` to make the new kernel available to it.

Moreover, if you use a modern distribution, you will need to generate the `initrd` image corresponding to the new kernel. The `initrd` images contain the kernel modules that should be loaded before the root filesystem is mounted as well as some scripts and utilities that should be executed before the kernel lets `init` run. To create the `initrd` image for your kernel and copy it to the system's `/boot` directory, you can do:

```
# mkinitrd -k vmlinuz-<kernel_version> -i initrd-<kernel_version>
```

where `<kernel_version>` is the kernel's release string. The kernel itself has to be copied to the file `/boot/vmlinuz-<kernel_version>` and the map of its symbols has to be saved in the file `/boot/System.map-<kernel_version>` before `mkinitrd` is run. The release string of the running kernel can be obtained by executing

```
$ uname -r
```

It usually corresponds to the version of the source code from which the kernel has been built.

After generating the `initrd` image for the new kernel, you have to adjust the configuration of the boot loader so that it can use this kernel. If you use GRUB, the configuration file should be `/boot/grub/menu.lst` . Open it in your favorite text editor and add the following lines:

```
title Linux <kernel_version>
       root (hdX,Y)
       kernel /boot/vmlinuz-<kernel_version> ro root=/dev/<root_partition>
#      If initrd is used
       initrd /boot/initrd-<kernel_version>
```

where `X` and `Y` are numbers used by GRUB to identify the partition that contains your system's `/boot` directory (please refer to the documentation of GRUB for more details), `<kernel_version>` is the kernel's release string and `<root_partition>` is the identification of you root partition that will be used by the kernel to mount the root filesystem and run `init`.

The users of LILO should add the following lines to its configuration file `/etc/lilo.conf`:

```
image=/boot/vmlinuz-<kernel_version>
label=linux
# If initrd is used
initrd=/boot/initrd-<kernel_version>
read-only
root=/dev/<root_partition>
```

It also is necessary to make `lilo` actually use the new kernel by running `/sbin/lilo` (please refer to the documentation of LILO for more information).

1.6.7 Automated configuration and installation

Some of the above steps or even all of them may be completed by the kernel installation script available in your distribution. Still, very often they have to be done manually. In such a case you can use the following script, which allows you to download, compile and install the latest stable kernel, after adjusting it to suit your system configuration:

```
#! /bin/sh

# Patch to the kernel source directory
SRC_PATH="/usr/src/kernel/linux-stable"
OBJ_PATH="$SRC_PATH-obj/"

cd $SRC_PATH
# Download the latest -stable
ketchup 2.6
# Save the version
VER=‘ketchup -m‘
# Generate the configuration file (based on the old one)
make O=$OBJ_PATH oldconfig
# Build the kernel
make O=$OBJ_PATH
# Install modules
sudo make O=$OBJ_PATH modules_install
# Copy the compressed kernel image into /boot
sudo cp $OBJ_PATH/arch/i386/boot/bzImage /boot/vmlinuz-$VER
# Copy System.map into /boot
sudo cp $OBJ_PATH/System.map /boot/System.map-$VER
# If you use Fedory, you have to generate the Fedora initrd
sudo /sbin/new-kernel-pkg --make-default --mkinitrd --depmod --install $VER
```

Generally, each distribution has its own preferred method of building and installing new kernels. The following table contains the links to the kernel installation instructions for several selected distributions:

Distribution	**URL**
CentOS	`http://www.howtoforge.com/kernel_compilation_centos`
Debian	`http://www.howtoforge.com/howto_linux_kernel_2.6_compile_debian`
Fedora	`http://www.howtoforge.com/kernel_compilation_fedora`
Gentoo	`http://www.gentoo.org/doc/en/kernel-upgrade.xml`
Mandriva	`http://www.howtoforge.com/kernel_compilation_mandriva`
OpenSUSE	`http://www.howtoforge.com/kernel_compilation_suse`
Ubuntu	`http://www.howtoforge.com/kernel_compilation_ubuntu`

To learn more about the configuration and compilation of the Linux kernel you can refer to the book *Linux Kernel in a Nutshell* by Greg Kroah-Hartman (`http://www.kroah.com/lkn/`).

Rozdział 2

Testing

Generally, there are many ways in which you can test the Linux kernel, but we will concentrate on the following four approaches:

1. Using a test version of the kernel for normal work.
2. Running special test suites, like LTP, on the new kernel.
3. Doing unusual things with the new kernel installed.
4. Measuring the system performance with the new kernel installed.

Of course, all of them can be used within one combined test procedure, so they can be regarded as different phases of the testing process.

2.1 Phase One

The first phase of kernel testing is simple: we try to boot the kernel and use it for normal work.

- Before starting the system in a fully functional configuration it is recommended to boot the kernel with the `init=/bin/bash` command line argument, which makes it start only one `bash` process. From there you can check if the filesystems are mounted and unmounted properly and you can test some more complex kernel functions, like the suspend to disk or to RAM, in the minimal configuration. In that case the only kernel modules loaded are the ones present in the `initrd` image mentioned in Subsection 1.6.6. Generally, you should refer to the documentation of your boot loader for more information about manual passing command line arguments to the kernel (in our opinion it is easier if GRUB is used).
- Next, it is advisable to start the system in the runlevel 2 (usually, by passing the number 2 to the kernel as the last command line argument), in which case network servers and the X server are not started (your system may be configured to use another runlevel for this purpose, although this is very unusual, so you should look into `/etc/inittab` for confidence). In this configuration you can check if the network interfaces work and you can try to run the X server manually to make sure that it does not crash.

- Finally, you can boot the system into the runlevel 5 (ie. fully functional) or 3 (ie. fully functional without X), depending on your needs.

Now, you are ready to use the system in a normal way for some time. Still, if you want to test the kernel quickly, you can carry out some typical operations, like downloading some files, reading email, browsing some web pages, ripping some audio tracks (from a legally bought audio CD, we presume), burning a CD or DVD etc., in a row to check if any of them fail in a way that would indicate a kernel problem.

2.2 Phase Two (AutoTest)

In the next phase of testing we use special programs designed for checking if specific kernel subsystems work correctly. We also carry out regression and performance tests of the kernel. The latter are particularly important for kernel developers (and for us), since they allow us to identify changes that hurt performance. For example, if the performance of one of our filesystems is 10% worse after we have upgraded the `2.6.x-rc1` kernel to the `2.6.x-rc2` one, it is definitely a good idea to find the patch that causes this to happen.

For automated kernel testing we recommend you to use the *AutoTest* suite (`http://test.kernel.org/autotest/`) consisting of many test applications and profiling tools combined with a fairly simple user interface.

To install *AutoTest* you can go into the `/usr/local` directory (as `root`) and run

```
# svn checkout svn://test.kernel.org/autotest/trunk autotest
```

Although it normally is not recommended to run such commands as `root`, this particular one should be safe, unless you cannot trust your DNS server, because it only downloads some files and saves them in `/usr/local` . Besides, you will need to run *AutoTest* as `root`, since some of its tests require superuser privileges to complete. For this reason you should not use *AutoTest* on a production system: in extreme cases the data stored in the system the privileged tests are run on can be damaged or even destroyed, and we believe that you would not like this to happen to your production data.

By design, *AutoTest* is noninteractive, so once started, it will not require your attention (of course, if something goes *really* wrong, you will have to recover the system, but this is a different kettle of fish). To start it you can go to `/usr/local/autotest/client` (we assume that *AutoTest* has been installed in `/usr/local`) and execute (as `root`)

```
# bin/autotest tests/test_name/control
```

where `test_name` is the name of the directory in `/usr/local/autotest/client/tests` that contains the test you want to run. The control file `tests/test_name/control` contains instructions for *AutoTest*. In the simplest cases only one such instruction is needed, namely

```
job.run_test('test_name')
```

where `test_name` is the name of the directory that contains the control file. The contents of more sophisticated control files can look like this:

```
job.run_test('pktgen', 'eth0', 50000, 0, tag='clone_skb_off')
job.run_test('pktgen', 'eth0', 50000, 1, tag='clone_skb_on')
```

where the strings after the test name represent arguments that should be passed to the test application. You can modify these arguments, but first you should read the documentation of the test application as well as the script `tests/test_name/test_name.py` (eg. `tests/pktgen/pktgen.py`) used by *AutoTest* to actually run the test (as you have probably noticed, the *AutoTest* scripts are written in Python). The results of the execution of the script `tests/test_name/test_name.py` are saved in the directory `results/default/test_name/`, where the `status` file contains the information indicating whether or not the test has been completed successfully. To cancel the test, press *Ctrl+C* while it is being executed.

If you want to run several tests in a row, it is best to prepare a single file containing multiple instructions for *AutoTest*. The instructions in this file should be similar to the ones contained in the above-mentioned control files. For example, the file `samples/all_tests` contains instructions for running all of the available tests and its first five lines are the following:

```
job.run_test('aiostress')
job.run_test('bonnie')
job.run_test('dbench')
job.run_test('fio')
job.run_test('fsx')
```

To run all of the tests requested by the instructions in this file, you can use the command

```
bin/autotest samples/all_tests
```

but you should remember that it will take *a lot* of time to complete. Analogously, to run a custom selection of tests, put the instructions for *AutoTest* into one file and provide its name as a command line argument to `autotest`.

To run several tests in parallel, you will need to prepare a special control file containing instructions like these:

```
def kernbench():
        job.run_test('kernbench', 2, 5)

def dbench():
        job.run_test('dbench')

job.parallel([kernbench], [dbench])
```

While the tests are being executed, you can stop them by pressing *Ctrl+C* at any time.

For people who do not like the command line and configuration files, ATCC (*AutoTest Control Center*) has been created. If you run it, for example by using the command `ui/menu`, you will be provided with a simple menu-driven user interface allowing you to select tests and profiling tools, view the results of their execution and, to a limited extent, configure them.

If you are bored with the selection of tools available in the *AutoTest* package, you can visit the web page `http://ltp.sourceforge.net/tooltable.php` containing a comprehensive list of tools that can be used for testing the Linux kernel.

2.3 Phase Three

Has your new kernel passed the first two phases of testing? Now, you can start to experiment. That is, to do stupid things that nobody sane will do during the normal work, so no one knows that they can crash the kernel. What exactly should be done? Well, if there had been a "standard" procedure, it would have certainly been included in some test suite.

The third phase can be started, for example, from unplugging and replugging USB devices. While in theory the replugging of a USB device should not change anything, at least from the user's point of view, doing it *many* times in a row may cause the kernel to crash if there is a bug in the USB subsystem (this may only cause the problem to appear provided that no one has ever tried this on a similarly configured system). Note, however, that this is also stressful to your hardware, so such experiments should better be carried out on add-on cards rather than on the USB ports attached directly to your computer's mainboard.

Next, you can write a script that will read the contents of files from the `/proc` directory in a loop or some such. In short, in the third phase you should do things that are never done by normal users (or that are done *very* rarely: why would anyone mount and unmount certain filesystem in an infinite loop? :)).

2.4 Measuring performance

As we have already mentioned, it is good to check the effects of the changes made to the kernel on the performance of the entire system (by the way, this may be an excellent task for beginner testers, who do not want to deal with development kernels yet, although they eagerly want to help develop the kernel). Still, to do this efficiently, you need to know how to do it and where to begin.

To start with, it is recommended to choose one subsystem that you will test regularly, since in that case your reports will be more valuable to the kernel developers. Namely, from time to time messages like "Hello, I've noticed that the performance of my network adapter decreased substantially after I had upgraded from `2.6.8` to `2.6.20`. Can anyone help me?" appear on the LKML. Of course, in such cases usually no one has a slightest idea of what could happen, because the kernel `2.6.20` was released two and a half years (and gazillion random patches) after `2.6.8`. Now, in turn, if you report that the performance of your network adapter has dropped 50% between the kernels `2.6.x-rc3` and `2.6.x-rc4`, it will be relatively easy to find out why. For this reason it is important to carry out the measurements of performance regularly.

Another thing that you should pay attention to is how your tests actually work. Ideally, you should learn as much as possible about the benchmark that you want to use, so that you know how to obtain reliable results from it. For example, in some Internet or press publications you can find the opinion that running

```
$ time make
```

in the kernel source directory is a good test of performance, as it allows you to measure how much time it takes to build the kernel on given system. While it is true that you can use this kind of tests to get some general idea of how "fast" (or how "slow") the system

is, they generally should not be regarded as *measurements* of performance, since they are not sufficiently precise. In particular, the kernel compilation time depends not only on the "speed" of the CPU and memory, but also on the time needed to load the necessary data into memory from the hard disk, which in turn may depend on where exactly these data are physically located. In fact, the kernel compilation is quite I/O-intensive and the time needed to complete it may depend on some more or less random factors. Moreover, if you run it twice in a row, the first run usually takes more time to complete than the second one, since the kernel caches the necessary data in memory during the first run and afterwards they can simply be read from there. Thus in order to obtain reproducible results, it is necessary to suppress the impact of the I/O, for example by forcing the kernel to load the data into memory before running the test. Generally, if you want to carry out the "`time make`" kind of benchmarks, it is best to use the `kernbench` script (`http://ck.kolivas.org/kernbench/`), a newer version of which is included in the *AutoTest* suite (see Section 2.2). Several good benchmarks are also available from `http://ltp.sourceforge.net/tooltable.php` (some of them are included in *AutoTest* too). Still, if you are interested in testing the kernel rather than in testing hardware, you should carefully read the documentation of the chosen benchmark, because it usually contains some information that you may need.

The next important thing that you should always remember about is the stability (ie. invariableness) of the environment in which the measurements are carried out. In particular, if you test the kernel, you should not change anything but the kernel in your system, since otherwise you would test two (or more) things at a time and it would not be easy to identify the influence of each of them on the results. For instance, if the measurement is based on building the kernel, it should always be run against the same kernel tree with *exactly* the same configuration file, using the same compiler and the other necessary tools (of course, once you have upgraded at least one of these tools, the results that you will obtain from this moment on should not be compared with the results obtained before the upgrade).

Generally, you should always do your best to compare apples to apples. For example, if you want to test the performance of three different file systems, you should not install them on three different partitions of the same disk, since the time needed to read (or write) data from (or to) the disk generally depends on where exactly the operation takes place. Instead, you should create one partition on which you will install each of the tested filesystems. Moreover, in such a case it is better to restart the system between consecutive measurements in order to suppress the effect of the caching of data.

Concluding, we can say that, as far as the measurements of performance are concerned, it is important to

- carry out the tests regularly
- know the "details" allowing one to obtain reliable results
- ensure the stability of the test environment
- compare things that are directly comparable

If all of these conditions are met, the resulting data will be very valuable source of information on the performance of given kernel subsystem.

2.5 *Hello world!*, or what exactly are we looking for?

What is it we are looking for? Well, below you can find some examples of messages that are related to kernel problems. Usually, such messages appear on the system console, but sometimes (eg. when the problem is not sufficiently serious to make the kernel crash immediately) you can also see them in the logs.

```
=============================================
[ INFO: possible recursive locking detected ]
---------------------------------------------
idle/1 is trying to acquire lock:
 (lock_ptr){....}, at: [<c021cbd2>] acpi_os_acquire_lock+0x8/0xa

but task is already holding lock:
 (lock_ptr){....}, at: [<c021cbd2>] acpi_os_acquire_lock+0x8/0xa

other info that might help us debug this:
1 lock held by idle/1:
 #0:  (lock_ptr){....}, at: [<c021cbd2>] acpi_os_acquire_lock+0x8/0xa

stack backtrace:
 [<c0103e89>] show_trace+0xd/0x10
 [<c0104483>] dump_stack+0x19/0x1b
 [<c01395fa>] __lock_acquire+0x7d9/0xa50
 [<c0139a98>] lock_acquire+0x71/0x91
 [<c02f0beb>] _spin_lock_irqsave+0x2c/0x3c
 [<c021cbd2>] acpi_os_acquire_lock+0x8/0xa
 [<c0222d95>] acpi_ev_gpe_detect+0x4d/0x10e
 [<c02215c3>] acpi_ev_sci_xrupt_handler+0x15/0x1d
 [<c021c8b1>] acpi_irq+0xe/0x18
 [<c014d36e>] request_irq+0xbe/0x10c
 [<c021cf33>] acpi_os_install_interrupt_handler+0x59/0x87
 [<c02215e7>] acpi_ev_install_sci_handler+0x1c/0x21
 [<c0220d41>] acpi_ev_install_xrupt_handlers+0x9/0x50
 [<c0231772>] acpi_enable_subsystem+0x7d/0x9a
 [<c0416656>] acpi_init+0x3f/0x170
 [<c01003ae>] _stext+0x116/0x26c
 [<c0101005>] kernel_thread_helper+0x5/0xb
```

The above message indicates that the kernel's runtime locking correctness validator (often referred to as "lockdep") has detected a possible locking error. The errors detected by lockdep need not be critical, so if you have enabled lockdep in the kernel configuration, which is recommended for testing (see Subsection 1.6.4), from time to time you can see them in the system logs or in the output of `dmesg`. They are always worth reporting, although sometimes lockdep may think that there is a problem even if the locking is used in a correct way. Still, in such a case your report will tell the kernel developers that they should teach lockdep not to trigger in this particular place any more.

```
BUG: sleeping function called from invalid context at /usr/src/linux-mm/sound/core/info.c:117
in_atomic():1, irqs_disabled():0
 <c1003ef9> show_trace+0xd/0xf
 <c100440c> dump_stack+0x17/0x19
 <c10178ce> __might_sleep+0x93/0x9d
 <f988eeb5> snd_iprintf+0x1b/0x84 [snd]
 <f988d808> snd_card_module_info_read+0x34/0x4e [snd]
 <f988f197> snd_info_entry_open+0x20f/0x2cc [snd]
 <c1067a17> __dentry_open+0x133/0x260
 <c1067bb7> nameidata_to_filp+0x1c/0x2e
 <c1067bf7> do_filp_open+0x2e/0x35
 <c1068bf2> do_sys_open+0x54/0xd7
 <c1068ca1> sys_open+0x16/0x18
```

```
 <c11dab67> sysenter_past_esp+0x54/0x75
BUG: using smp_processor_id() in preemptible [00000001] code: init/1
caller is __handle_mm_fault+0x2b/0x20d
 [<c0103ba8>] show_trace+0xd/0xf
 [<c0103c7a>] dump_stack+0x17/0x19
 [<c0203bcc>] debug_smp_processor_id+0x8c/0xa0
 [<c0160e60>] __handle_mm_fault+0x2b/0x20d
 [<c0116f7b>] do_page_fault+0x226/0x61f
 [<c0103959>] error_code+0x39/0x40
 [<c019d4c1>] padzero+0x19/0x28
 [<c019e716>] load_elf_binary+0x836/0xc02
 [<c017db53>] search_binary_handler+0x123/0x35a
 [<c019d3b9>] load_script+0x221/0x230
 [<c017db53>] search_binary_handler+0x123/0x35a
 [<c017deee>] do_execve+0x164/0x215
 [<c0101e7a>] sys_execve+0x3b/0x7e
 [<c02fabc3>] syscall_call+0x7/0xb
```

The above message means that one of the kernel's functions has been called from a wrong place. In this particular case the execution of the function `snd_iprintf()` might be suspended until certain condition is satisfied (in such cases the kernel developers say that the function might sleep), so it should not be called from the portions of code that have to be executed atomically, such as interrupt handlers (ie. from *atomic context*). However, apparently `snd_iprintf()` has been called from atomic context and the kernel reports this as a potential problem. Such problems need not cause the kernel to crash and the related messages, similar to the above one, can appear in the system logs or in the output of `dmesg`, but they are *serious* and should always be reported.

The next message is a so-called `Oops`, which means that it represents a problem causing the kernel to stop working. In other words, it means that something *really* bad has happened and the kernel cannot continue running, since your hardware might be damaged or your data might be corrupted otherwise. Such messages are often accompanied by so-called *kernel panics* (the origin of the term "kernel panic" as well as its possible meanings are explained in the *OSWeekly.com* article by Puru Govind which is available at `http://www.osweekly.com/index.php?option=com_content&task=view&id=2241&Itemid=449`).

```
BUG: unable to handle kernel paging request at virtual address 6b6b6c07
 printing eip:
c0138722
*pde = 00000000
Oops: 0002 [#1]
4K_STACKS PREEMPT SMP
last sysfs file: /devices/pci0000:00/0000:00:1d.7/uevent
Modules linked in: snd_timer snd soundcore snd_page_alloc intel_agp agpgart
ide_cd cdrom ipv6 w83627hf hwmon_vid hwmon i2c_isa i2c_i801 skge af_packet
ip_conntrack_netbios_ns ipt_REJECT xt_state ip_conntrack nfnetlink xt_tcpudp
iptable_filter ip_tables x_tables cpufreq_userspace p4_clockmod speedstep_lib
binfmt_misc thermal processor fan container rtc unix
CPU:    0
EIP:    0060:[<c0138722>]    Not tainted VLI
EFLAGS: 00010046   (2.6.18-rc2-mm1 #78)
EIP is at __lock_acquire+0x362/0xaea
eax: 00000000   ebx: 6b6b6b6b   ecx: c0360358   edx: 00000000
esi: 00000000   edi: 00000000   ebp: f544ddf4   esp: f544ddc0
ds: 007b   es: 007b   ss: 0068
Process udevd (pid: 1353, ti=f544d000 task=f6fce8f0 task.ti=f544d000)
Stack: 00000000 00000000 00000000 c7749ea4 f6fce8f0 c0138e74 000001e8 00000000
        00000000 f6653fa4 00000246 00000000 00000000 f544de1c c0139214 00000000
        00000002 00000000 c014fe3a c7749ea4 c7749e90 f6fce8f0 f5b19b04 f544de34
Call Trace:
[<c0139214>] lock_acquire+0x71/0x91
[<c02f2bfb>] _spin_lock+0x23/0x32
```

```
[<c014fe3a>] __delayacct_blkio_ticks+0x16/0x67
[<c01a4f76>] do_task_stat+0x3df/0x6c1
[<c01a5265>] proc_tgid_stat+0xd/0xf
[<c01a29dd>] proc_info_read+0x50/0xb3
[<c0171cbb>] vfs_read+0xcb/0x177
[<c017217c>] sys_read+0x3b/0x71
[<c0103119>] sysenter_past_esp+0x56/0x8d
DWARF2 unwinder stuck at sysenter_past_esp+0x56/0x8d
Leftover inexact backtrace:
[<c0104318>] show_stack_log_lvl+0x8c/0x97
[<c010447f>] show_registers+0x15c/0x1ed
[<c01046c2>] die+0x1b2/0x2b7
[<c0116f5f>] do_page_fault+0x410/0x4f0
[<c0103d1d>] error_code+0x39/0x40
[<c0139214>] lock_acquire+0x71/0x91
[<c02f2bfb>] _spin_lock+0x23/0x32
[<c014fe3a>] __delayacct_blkio_ticks+0x16/0x67
[<c01a4f76>] do_task_stat+0x3df/0x6c1
[<c01a5265>] proc_tgid_stat+0xd/0xf
[<c01a29dd>] proc_info_read+0x50/0xb3
[<c0171cbb>] vfs_read+0xcb/0x177
[<c017217c>] sys_read+0x3b/0x71
[<c0103119>] sysenter_past_esp+0x56/0x8d
Code: 68 4b 75 2f c0 68 d5 04 00 00 68 b9 75 31 c0 68 e3 06 31 c0 e8 ce 7e fe ff
e8 87 c2 fc ff 83 c4 10 eb 08 85 db 0f 84 6b 07 00 00 <f0> ff 83 9c 00 00 00 8b
55 dc 8b 92 5c 05 00 00 89 55 e4 83 fa
EIP: [<c0138722>] __lock_acquire+0x362/0xaea SS:ESP 0068:f544ddc0
```

The following message represents an error resulting from a situation that, according to the kernel developers, cannot happen:

```
KERNEL: assertion ((int)tp->lost_out >= 0) failed at net/ipv4/tcp_input.c (2148)
KERNEL: assertion ((int)tp->lost_out >= 0) failed at net/ipv4/tcp_input.c (2148)
KERNEL: assertion ((int)tp->sacked_out >= 0) failed at net/ipv4/tcp_input.c (2147)
KERNEL: assertion ((int)tp->sacked_out >= 0) failed at net/ipv4/tcp_input.c (2147)
```

```
BUG: warning at /usr/src/linux-mm/kernel/cpu.c:56/unlock_cpu_hotplug()
 [<c0103e41>] dump_trace+0x70/0x176
 [<c0103fc1>] show_trace_log_lvl+0x12/0x22
 [<c0103fde>] show_trace+0xd/0xf
 [<c01040b0>] dump_stack+0x17/0x19
 [<c0140e19>] unlock_cpu_hotplug+0x46/0x7c
 [<fd9560b0>] cpufreq_set+0x81/0x8b [cpufreq_userspace]
 [<fd956109>] store_speed+0x35/0x40 [cpufreq_userspace]
 [<c02ac9f2>] store+0x38/0x49
 [<c01aec16>] flush_write_buffer+0x23/0x2b
 [<c01aec69>] sysfs_write_file+0x4b/0x6c
 [<c01770af>] vfs_write+0xcb/0x173
 [<c0177203>] sys_write+0x3b/0x71
 [<c010312d>] sysenter_past_esp+0x56/0x8d
 [<b7fbe410>] 0xb7fbe410
 [<c0103fc1>] show_trace_log_lvl+0x12/0x22
 [<c0103fde>] show_trace+0xd/0xf
 [<c01040b0>] dump_stack+0x17/0x19
 [<c0140e19>] unlock_cpu_hotplug+0x46/0x7c
 [<fd9560b0>] cpufreq_set+0x81/0x8b [cpufreq_userspace]
 [<fd956109>] store_speed+0x35/0x40 [cpufreq_userspace]
 [<c02ac9f2>] store+0x38/0x49
 [<c01aec16>] flush_write_buffer+0x23/0x2b
 [<c01aec69>] sysfs_write_file+0x4b/0x6c
 [<c01770af>] vfs_write+0xcb/0x173
 [<c0177203>] sys_write+0x3b/0x71
 [<c010312d>] sysenter_past_esp+0x56/0x8d
```

Apart from the messages that can appear on the console, in the output of `dmesg` or in the system logs, some problems can be reported in a less direct way. For example, if the kernel

memory leak detector is used (so far, it has not been included in stable kernels), there is the file `/sys/kernel/debug/memleak`, in which possible kernel memory leaks are registered, eg.

```
orphan pointer 0xf5a6fd60 (size 39):
c0173822: <__kmalloc>
c01df500: <context_struct_to_string>
c01df679: <security_sid_to_context>
c01d7eee: <selinux_socket_getpeersec_dgram>
f884f019: <unix_get_peersec_dgram>
f8850698: <unix_dgram_sendmsg>
c02a88c2: <sock_sendmsg>
c02a9c7a: <sys_sendto>
```

This information, supplemented with the kernel configuration file (see Section 1.6), may allow the kernel developers to fix the bug that you have managed to find.

The examples shown above are related to problems that occur when the kernel is running, called *run-time* errors. Obviously, to get a run-time error you need to build, install and boot the kernel. Surprisingly, however, it is possible that you will not be able to build the kernel due to a compilation error. This is not a frequent problem and it usually indicates that the author of certain piece of kernel code was not careful enough. Still, this happens to many developers, including us, and if you find a compilation error, report it immediately (you can even try to fix it if you are good at programming in C). To find examples of what happens after someone finds a compilation problem in the kernel, you can look at one of the discussions taking place on the LKML after Andrew Morton announces a new -mm kernel (for more information about the -mm tree see Section 1.5).

It should be stressed that some kernel bugs are not immediately visible. Some of them show up only in specific situations and may manifest themselves, for example, in hanging random processes or dropping random data into files that are written to. For instance, there is a whole category of kernel problems that appear only when the system is suspended to RAM or hibernated (ie. suspended to disk), either during the suspend, or while the kernel is resuming normal operations. All in all, you will never know what surprises the kernel has got for you, so you should better be prepared.

Generally, kernel run-time errors can be divided into three categories:

- easily reproducible – such that we know exactly what to do to provoke them to happen
- fairly reproducible – such that occur quite regularly and we know more or less in what situations
- difficult to reproduce – such that occur in (seemingly) random circumstances and we have no idea how to make them occur

The easily reproducible bugs are the easiest to fix, since in these cases it is quite easy, albeit often quite time-consuming, to find a patch that has introduced the problem. For this reason, the easily reproducible bugs are usually fixed relatively quickly. In turn, the bugs that are difficult to reproduce usually take a lot of time to get fixed, since in these cases the source of the problem cannot be easily identified. If you encounter such a bug, you will probably need help of the developers knowing the relevant kernel subsystem and you will have to be *very* patient.

2.6 Binary drivers and distribution kernels

Quite often you can hear that so-called "binary" drivers are "evil" and you should not use them. Well, this is generally true, apart from the fact that sometimes you have no choice (eg. new AMD/ATI graphics adapters are not supported by any Open Source driver known to us). In our opinion there is at least one practical argument for not using binary drivers. Namely, if you find a bug in the kernel that occurs while you are using a binary driver, the kernel developers may be unable to help you, because they have no access to the driver's source code.

When you are using a binary driver, the kernel is "tainted", which means that the source of possible problems may be unrelated to the kernel code (see `https://secure-support.novell.com/KanisaPlatform/Publishing/250/3582750_f.SAL_Public.html` for more details). You can check whether or not the kernel was tainted when the problem occurred by looking at the corresponding error message. If can you see something similar to the following line:

```
EIP:    0060:[<c046c7c3>]    Tainted: P   VLI
```

(the word `Tainted` is crucial here), the kernel was tainted and most probably the kernel developers will not be able to help you. In that case you should try to reproduce the problem without the binary driver loaded. Moreover, if the problem does not occur without it, you should send a bug report to the creators of the binary driver and ask them to fix it.

In the file `Documentation/oops-tracing.txt`, included in the kernel sources, there is a list of reasons why the kernel can be considered as tainted. As follows from this document, the presence of a binary module is not the only possible reason of tainting the kernel, but in practice it turns out to be the most frequent one. Generally, you should avoid reporting problems in tainted kernels to the LKML (or to the kernel developers in general) and the problems related to binary drivers should be reported to their providers.

Another case in which you should not report kernel problems to the LKML, or directly to the kernel developers, is when you are using a distribution kernel. The main reasons of this are the following:

- Distribution kernels often contain modifications that are not included in the kernels available from `ftp://ftp.kernel.org` and have not been accepted by the maintainers of relevant kernel subsystems
- Some of these modification are *very* experimental and they tend to introduce bugs that are not present in the "official" kernels
- Distribution kernels are meant to be supported by their distributors rather by the kernel developers, so the problems in these kernels should be reported to the distributors *in the first place* (usually, the distributor will contact the kernel developers anyway if that is necessary)

Of course, if the problem can be reproduced using the "original" kernel on which the distribution one is based, it can be reported to the kernel developers. Still, it usually is better to let the distributor know of the problem anyway and in our opinion it does not make sense to report the same problem twice, does it?

Rozdział 3

Collecting kernel messages

There are several methods of collecting kernel messages, some of them more efficient than the others, and each of them has some drawbacks and advantages.

3.1 Syslog, console and `dmesg`

The most popular method is to use `klogd`, the daemon that writes the kernel messages to a log file, usually included in the `syslog` package. Depending on your distribution's configuration of `klogd`, the kernel messages can be written to `/var/log/kern.log`, `/var/log/messages` or `/var/log/kernel`, or some more complicated approach may be used (eg. on OpenSUSE systems boot messages are saved in the file `/var/log/boot.msg` and the messages received by `klogd` after the system has reached the required runlevel are written to `/var/log/messages`). The advantage of this method is that practically every distribution enables `klogd` by default and you do not really have to configure it. Unfortunately, it usually is started quite late during the boot process and stopped early during the system shutdown, which often makes it impossible to use `klogd` to collect messages appearing early during the system start or very late when it is going down. Moreover, if there is a run-time error that causes the kernel to crash (ie. to decide that it cannot continue running), the corresponding message cannot be written into a file and `klogd` may not be able to process it. For this reason, even if you use `klogd`, which is recommended, you will need an additional means of collecting critical messages from the kernel.

In principle, you can use the system console for this purpose. Generally speaking, the console is where the most important kernel messages appear. By default it is the current virtual terminal, but in fact it can be many places. The kernel's command line parameter `console=` can be used to direct kernel messages to specific virtual terminal or to specified serial port (see the file `Documentation/kernel-parameters.txt` in the kernel sources for more details). When `klogd` starts, it may redirect kernel messages to some other destinations, usually specified in `/etc/syslog.conf` (for more information about this file refer to the documentation of `syslog` provided in your distribution).

The "traditional" method of collecting console messages is to read them from the system console and write them down, either on a piece of paper, or using a text editor on a second computer, provided that you have one. Of course, this is time-consuming and nobody likes

to do this, because it usually is very difficult to rewrite the message sufficiently precisely. Sometimes the most important part of the message "escapes" from the screen and you cannot really help it, except for increasing the console resolution, either by booting the kernel with the `vga=1` command line parameter (80x50 console), or by using a frame buffer console (see the file `Documentation/fb/fbcon.txt` in the kernel sources). Unfortunately, "interesting" errors often make the kernel print huge messages and even the increased console resolution need not be sufficient in such cases.

Obviously, instead of rewriting messages from the console manually, you can simply photograph them and make such a "screen dump" available from a web page (if you change the resolution of the picture to 1024x768, for example, and save it in a grey scale, it will be more "download-friendly"). You should not, however, send the picture directly to the kernel developers, unless someone asks you to do this. In particular, the LKML has the message size limit of 100 KB and it does not make sense to send e-mail attachments larger than 100 KB to it.

Another problem with the system console is that it may not be always visible. For example, if you use the X Window System, you will not see the console, unless you run the `xconsole` program (or its equivalent). Moreover, even if you run it, the kernel may crash in a spectacular way and the related messages need not make it to `xconsole`. In that case you also may not be able to switch from X to the text console to read the error messages. For this and the above reasons, it often is necessary to send kernel messages out of the system that they are generated on and some methods of doing it are described below.

Sometimes you may need to read the kernel messages even if it does not crash and you can use the `dmesg` utility for this purpose. By default `dmesg` reads all messages from the kernel's ring buffer and sends them to the standard output, so you can redirect the output of `dmesg` to a file or to your text viewer of choice. The number of messages printed by `dmesg` depends on the size of the kernel's ring buffer which is configurable at compilation time

```
Kernel hacking  --->
Kernel debugging  --->
Kernel log buffer size (16 => 64KB, 17 => 128KB)  --->
```

It also depends on the size of the buffer used by `dmesg` that can be adjusted with the help of the `-s` option (see the `dmesg` man page for more details). If you report a bug to the kernel developers, they may ask you to send the output of `dmesg`, so you should be familiar with it.

3.2 Serial console

The method of collecting kernel messages with the help of the serial console has two important drawbacks. First, it requires a serial port to be present in the computer running the tested kernel ("test bed computer") that will generate the messages of interest, so it cannot be used, for example, with the majority of contemporary notebooks. Second, it requires you to use another computer for receiving the messages. On the other hand, the serial console can be used for collecting messages generated in the early stage of the kernel's life time.

To use the serial console you need to connect one of the test bed computer's serial ports (often referred to as a COM ports) to a serial port installed in the other computer (this

serial port may be emulated, for example with the help of a USB-to-serial converter) with a so-called null modem cable.

You also need to configure the tested kernel to support the serial console:

```
Device Drivers  --->
Character devices  --->
Serial drivers  --->
<*> 8250/16550 and compatible serial support
[*]   Console on 8250/16550 and compatible serial port
```

and you need to tell the kernel to actually *use* it, for example by appending the following parameters to the kernel's command line:

```
console=ttyS0,115200n8 console=tty0
```

(the `console=tty0` means that we want the kernel to use the virtual terminal 0 as a console *apart from* the serial one). Additionally, if you use GRUB as the boot loader, you can teach it to receive commands via the serial link from the other machine, for example by adding the following lines to its configuration file (usually `/boot/grub/menu.lst`):

```
serial --unit=0 --speed=115200 --word=8 --parity=no --stop=1
terminal --timeout=5 serial console
```

Still, this is not necessary if you have a "normal" keyboard and monitor attached to the test bed machine (refer to the documentation of GRUB for more information about the options used above).

Usually, it also is a good idea to configure the test bed system to use the serial console as one of its terminals. In Fedora you can do it according to the instructions below, but the other distributions may require you to modify the system configuration in a different way.

First, edit the file `/etc/sysconfig/init` and assign the value `serial` to the variable `BOOTUP` . Next, in the file `/etc/sysconfig/kudzu` set `SAFE=yes` and in `/etc/securetty` add `ttyS0` . Now, add the line

```
S1:23:respawn:/sbin/mgetty -L ttyS0 115200 vt100
```

to `/etc/inittab` (this requires you to have the `mgetty` package installed). It is recommended to apply these changes to all systems on which you intend to use the serial terminal (you can always comment out the settings that enable it when it is no longer necessary).

Of course, the machine that will receive the messages over the serial link has to be configured too, but this usually is quite straightforward. First, run

```
# minicom -o -C log.txt
```

on it (you need to do this as `root`). Most probably you will get the error message

```
Device /dev/modem/ acces failed: (...)
```

which means that `minicom` could not find the special device file `/dev/modem` . To overcome this problem you can, for example, create a symbolic link from `ttyS0` to it

```
# ln -s /dev/ttyS0 /dev/modem
```

and then `minicom` should run. It still is necessary to set up `minicom` itself, but this can be done with the help of its configuration menu. By default `minicom` is configured to use modem lines, so you have to change this setting. You also need to set the serial link's transmission parameters to reflect the settings that you have done on the test bed computer (note that some serial ports do not work with the highest baud rates and in such cases you will need to decrease the baud rate and this has to be done on both ends of the serial link, because they both *must* use exactly the same parameters of transmission).

For more information about using the serial console with the Linux kernel refer to the file `Documentation/serial-console.txt` included in the kernel sources.

3.3 Network console

If there are no serial ports in the computer on which new kernels are going to be tested, you will need to use some other means of collecting kernel messages, such as the network console.

Of course, for the network console to work an additional computer is necessary, on which the messages will appear. Moreover, this computer should be connected to the same Ethernet LAN as your test bed machine (unfortunately, as of today Ethernet is the only type of network that the network console can be used with), which may be regarded as a drawback. Another drawback of the network console is that it will not work before the networking has been initialized on the test bed system, so it cannot be used for transmitting some early kernel messages. In turn, an undoubted advantage of it is that the distance between the test bed computer and the machine used for collecting images may be relatively large (at least in comparison with the serial console).

To make the tested kernel support the network console, you need configure it appropriately

```
Device Drivers  --->
Network device support  --->
<*>   Network console logging support (EXPERIMENTAL)
```

(for more information about the configuration of the kernel see Section 1.6). Next, to tell it that you want it to send the messages over the network, you can append the `netconsole=` parameter to its command line, eg.

```
netconsole=4444@192.168.100.1/eth0,6666@192.168.100.2/00:14:38:C3:3F:C4
```

where the first three settings are related to the test bed system, namely

- `4444` is the number of the UDP port,
- `192.168.100.1` is the IP address of the network interface (obviously, it has to correspond to an Ethernet card),
- `eth0` is the name of the the network interface,

that will be used for sending kernel messages, and the remaining three ones are related to the other system, ie.

- `6666` is the number of the UDP port,
- `192.168.100.2` is the IP address of the network interface,
- `00:14:38:C3:3F:C4` is the MAC (ie. hardware Ethernet) address of the network adapter,

that will be used for receiving them. Remember to make sure that the last two settings reflect the actual configuration of the machine supposed to receive the messages over the network.

The computer that will be used for collecting messages from the tested kernel need not be configured in any special way, except that it should be running a program that will receive the messages and save them or print them on the screen. For this purpose you can use the `netcat` utility (`http://netcat.sourceforge.net/`), for example in the following way:

```
$ netcat -u -l -p 6666
```

where the port number (`6666` in this example) has to be the one that you have provided to the tested kernel.

It is worthy of noting that the network console driver can be built as a module and loaded at run time, but we are not going to cover this case. More information about that can be found in the file `Documentation/networking/netconsole.txt` included in the kernel sources.

Rozdział 4

Git, quilt and binary searching

Generally, Linux kernel source trees are maintained using two basic tools, `git` and `quilt`. While it is true that you can also use some `git` "frontends", like *Cogito*, *Stacked GIT*, *Patchy Git* (`pg`), `(h)gct`, which simplify user interactions with this tool, we will focus on `git` itself and on `quilt`, since each of them is strictly related to a specific way of maintaining a source tree.

4.1 Git

Git is a versioning control system written by Linus Torvalds specifically for the maintenance of the main Linux kernel source tree (ie. the mainline). The current maintainer of `git` itself is Junio C. Hamano and the latest version is `git-1.5.1.3`.

Of course, to use `git` you need to install it in your system, but all of the contemporary Linux distributions provide `git` packages that can be installed in usual ways. The most important of these packages is usually called `git-core`, as it contains the most basic `git` components. Alternatively, you can download the source code of `git` from `http://www.kernel.org/pub/software/scm/git/` and build it yourself.

Like some other versioning control systems (eg. *CVS*), `git` assumes that there is a central "master" directory tree used as a reference for all changes made to the maintained source code. This directory tree contains the source code itself along with the history of all changes made to it since certain point in time. To be registered in the "master" tree, the changes must be accepted by its maintainer (eg. by Linus, in the case of the Linux kernel "master" tree) and after they have been registered ("committed" in the `git`-related terminology) all of the users of this tree can "see" the source code with these changes applied. Still, they can also see how the source code looked like before specific modifications and they can follow its history (ie. view all modifications made to it after given point that may be specified in many different ways). For this reason each modification of the source code is registered in the "master" tree along with additional information including, among other things, the name and email address of its author and a short description of the purpose and scope of the modification (known as the "changelog").

If you create a copy of the Linux kernel "master" tree, all of the information contained in it will be available to you locally and can be used for many different purposes. In particular,

you can use them to identify the modifications that have introduced bugs.

To make a copy of this tree, or to "clone" it in the `git` language, you can run

```
$ git-clone \
git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git \
linux-git
```

This will download the entire Linus' `git` tree to your machine and the local copy of it will be located in the subdirectory `linux-git` of the current directory (ie. the one that was current when the above command was being executed). You should remember, however, that the Linus' tree is huge, so the download may take quite a lot of time, depending on your Internet connection's bandwidth and the current load of the `kernel.org`'s `git` server. If you are not going to use the `git` capabilities described below, it might be less cumbersome to download stand-alone patches and apply them as described in Section 1.2.

Once you have downloaded the Linus' `git` tree, it generally is a good idea to tell `git` that it is the the tree that you want to start with. To do this, change directory to `linux-git` and run

```
$ git-checkout -f
```

Now, you are able to do some really nice things with the help of `git` . For example, you can synchronize your local copy of the Linus' tree with the original one by downloading *only* the changes that were committed by Linus *after* you had run `git-clone` . For this purpose, run

```
$ git-pull \
git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git
$ git-checkout
```

Of course, this means that you do not need to run `git-clone` every time the Linus' tree changes. Moreover, you should not do this. In fact, it is only necessary do download the Linus' tree *once* and then use `git-pull` to update your local copy. Naturally, if you make your own changes to the local copy of the tree, `git-pull` may fail, but then you can use '`git-checkout -f`' to revert all of the conflicting changes. Still, if you want to save your modifications, it is not that easy any more, but we will not discuss this case any further (for more information see, for example, the tutorial at `http://www.kernel.org/pub/software/scm/git/docs/tutorial.html`).

Apart from the `git` commands presented above, you will probably find the following ones useful:

- `git-whatchanged <file>` – shows all changes affecting given source file (the patch relative to the `git` tree's root directory should be provided)
- `git-bisect *` – used for binary searching for buggy patches (see below)
- `git-revert *` – reverts given change ("commit" in the `git` terminology)
- `gitk` – graphically visualizes the tree
- `git-show` – prints given commit

- `git-log` – shows the list of changes made to the tree

In our opinion particularly useful is the `git-log` command that allows you to track the history of the source code. By running

```
$ git-log
```

you can see all changes made since the Linus' tree was created. Well, albeit impressive, this is not particularly useful, because there are *very* many of them, but you can easily narrow the scope of its output. For example, the command

```
$ git-log v2.6.19..+
```

will print the changes made since the version `2.6.19` of the kernel (this will not work for `v2.6.21..`, but we do not know why), and the following one:

```
$ git-log --since="7 days ago"
```

will give you all changes made since – you have guessed it – 7 days ago. Isn't it nice?

While playing with `git-log` you can see that each change, or commit, in the log starts from the word `commit` and a long hexadecimal number. These numbers are unique commit identifiers that can be used for referring to specific commits in many `git` commands. For example, if you want to see the source code modifications associated with commit `b5bf28cde894b3bb3bd25c13a7647020562f9ea0` in the form of a patch, run

```
$ git-show b5bf28cde894b3bb3bd25c13a7647020562f9ea0
```

It sometimes is unnecessary to use the entire commit identifier here, since several initial characters may be sufficient to identify the object. Thus to get the same result as from the above command, it may be sufficient to run

```
$ git-show b5bf28
```

Of course, if you want to save the resulting patch in a file, it is only necessary to do

```
$ git-show b5bf28 > b5bf28.patch
```

where `b5bf28.patch` can be replaced with an arbitrary file name.

When you run such `git` commands as `git-clone` or `git-pull`, some data are being transferred over the Internet. In such cases you should always use the `git://` protocol designed specifically for transferring `git` objects. Still, sometimes you may not be able to do this (eg. if you are behind a firewall that you cannot configure) and then you can use the "ordinary" `http://` protocol, but it is not generally recommended.

It is worthy of noting that the Linus' tree is not the only `git` tree downloadable from `kernel.org` . The other ones correspond to various development branches of the kernel introduced in Section 1.3. At `http://git.kernel.org` there is the list of all `git` trees available from `kernel.org` . Another list of kernel `git` tree is available from `http://git.infradead.org` .

To learn more about `git`, you can read the documentation distributed along with it or visit, for instance, the web page at `http://www.kernel.org/pub/software/scm/git/docs`

4.2 Quilt

Quilt is a tool to manage series of patches. First of all, it makes the creation of patches easy and allows one to preserve the right ordering of patches within the series. It also automates the application and reverting of patches as well as the updating of patches in accordance with manual modifications of the source code. Usually, it can be installed from a package provided by your distribution, but you can also install it from the source code available at `http://savannah.nongnu.org/projects/quilt/` . In principle, it can be used for managing modifications of an arbitrary set of text files located in a single directory tree, but we will focus on using it with respect to the Linux kernel source code.

Some Linux kernel developers use `quilt` as their primary patch management tool and some of the patchsets discussed in Section 1.4 are distributed as `quilt`-friendly series of patches. The most important of them is the `-mm` tree maintained by Andrew Morton (see Section 1.5).

A `quilt`-friendly series of patches consists of the patches themselves and the file `series`, in which the patch names are saved in the right order (they must be the same as the names of the files that contain the patches). To apply the patches, you only need to create a subdirectory called `patches` in the root of the kernel source tree and place the entire patch series, including the `series` file, in it. Then, you can use the '`quilt push -a`' command to apply them all in one shot.

Suppose, for example, that you want to apply the series of patches available at `http://www.sisk.pl/kernel/hibernation_and_suspend/2.6.22-rc1/patches/` and consisting of the following files:

```
01-freezer-close-theoretical-race-between-refrigerator-and-thaw_tasks.patch
02-freezer-fix-vfork-problem.patch
03-freezer-take-kernel_execve-into-consideration.patch
04-fix-kthread_create-vs-freezer-theoretical-race.patch
05-fix-pf_nofreeze-and-freezeable-race-2.patch
06-move-frozen_process-to-kernel-power-processc.patch
07-power-management-use-mutexes-instead-of-semaphores.patch
08-swsusp-fix-sysfs-interface.patch
09-acpi-fix-suspend-resume-ordering.patch
10-swsusp-remove-platform-callbacks-from-resume-code.patch
11-swsusp-reduce-code-duplication-between-user_c-and-disk_c.patch
series
```

on top of the `2.6.22-rc1` kernel source. For this purpose, go to the directory that contains the kernel sources, create the directory `patches` and copy the files that the series consists of into it. Next, run

```
$ quilt push -a
```

and all patches in the series will be applied. Alternatively, you can run

```
$ quilt push \
11-swsusp-reduce-code-duplication-between-user_c-and-disk_c.patch
```

and `quilt` will apply all patches in the series up to and including the one given as the argument to '`quilt push`'. The ordering of patches is based on the contents of the `series` file. That is, the patches the names of which are at the beginning of the `series` file are applied first.

Generally, you should not modify the `series` file manually, but sometimes it is convenient to do that. Suppose, for instance, that you do not want the patch

```
09-acpi-fix-suspend-resume-ordering.patch
```

from the above series to be applied. In that case, you can place '`#`' before the name of the patch in the `series` file:

```
01-freezer-close-theoretical-race-between-refrigerator-and-thaw_tasks.patch
02-freezer-fix-vfork-problem.patch
03-freezer-take-kernel_execve-into-consideration.patch
04-fix-kthread_create-vs-freezer-theoretical-race.patch
05-fix-pf_nofreeze-and-freezeable-race-2.patch
06-move-frozen_process-to-kernel-power-processc.patch
07-power-management-use-mutexes-instead-of-semaphores.patch
08-swsusp-fix-sysfs-interface.patch
#09-acpi-fix-suspend-resume-ordering.patch
10-swsusp-remove-platform-callbacks-from-resume-code.patch
11-swsusp-reduce-code-duplication-between-user_c-and-disk_c.patch
```

and that will make `quilt` regard this line as a comment and skip the patch. You can also change the ordering of patches by changing the ordering of names in the `series` file manually. Remember, however, *not to change* the ordering of names of the patches that have already been applied, because that will confuse `quilt`.

The applied patches are treated by `quilt` as though they were on a stack. The first applied patch is on the bottom of the stack, while the last applied patch is on the top of it. That's why '`quilt push`' is used to apply patches. Specifically, '`quilt push`' applies the next patch in the series that has not been applied yet and places it on the top of the stack. Analogously, '`quilt pop`' reverts the most recently applied patch (ie. the one on the top of the stack) and removes it from the stack. The command '`quilt push -a`' applies all patches in the series that have not been applied yet and places them on the stack in the right order, while '`quilt pop -a`' reverts all of the applied patches and removes them from the stack, one by one.

You can make `quilt` print the name of the most recently applied patch (ie. the one on the top of the stack) by running '`quilt top`', while '`quilt next`' will show you the name of the next patch to be applied. Similarly, '`quilt previous`' prints the name of the patch that has been applied right before the last one, '`quilt applied`' prints the names of all the currently applied patches (in the order in which they have been applied, so the name of the "top" patch is printed last), and '`quilt series`' prints the names of all patches in the series.

You can also provide `quilt` with the number of patches to be applied or reverted. Namely, if you want it to apply the next two patches in the series, run

```
$ quilt push 2
```

Similarly, to make it revert the last two most recently applied patches, use

```
$ quilt pop 2
```

(these commands are very useful in binary searching for "bad" patches, discussed in the next section). For example, having applied the first two patches from the series introduced above, you may want to apply the next three patches:

```
$ quilt push 3
Applying patch 03-freezer-take-kernel_execve-into-consideration.patch
patching file kernel/power/process.c

Applying patch 04-fix-kthread_create-vs-freezer-theoretical-race.patch
patching file kernel/kthread.c

Applying patch 05-fix-pf_nofreeze-and-freezeable-race-2.patch
patching file include/linux/freezer.h

Now at patch 05-fix-pf_nofreeze-and-freezeable-race-2.patch
```

Now, `quilt` tells you that `05-fix-pf_nofreeze-and-freezeable-race-2.patch` is on the top of the stack:

```
$ quilt top
05-fix-pf_nofreeze-and-freezeable-race-2.patch
$ quilt next
06-move-frozen_process-to-kernel-power-processc.patch
$ quilt previous
04-fix-kthread_create-vs-freezer-theoretical-race.patch
```

Next, suppose that you want to apply two patches more:

```
$ quilt push 2
Applying patch 06-move-frozen_process-to-kernel-power-processc.patch
patching file include/linux/freezer.h
patching file kernel/power/process.c

Applying patch 07-power-management-use-mutexes-instead-of-semaphores.patch
patching file drivers/base/power/main.c
patching file drivers/base/power/power.h
patching file drivers/base/power/resume.c
patching file drivers/base/power/runtime.c
patching file drivers/base/power/suspend.c

Now at patch 07-power-management-use-mutexes-instead-of-semaphores.patch
```

Reverting of the three most recently applied patches is also simple:

```
$ quilt pop 3
Removing patch 07-power-management-use-mutexes-instead-of-semaphores.patch
Restoring drivers/base/power/resume.c
Restoring drivers/base/power/main.c
Restoring drivers/base/power/runtime.c
Restoring drivers/base/power/power.h
Restoring drivers/base/power/suspend.c

Removing patch 06-move-frozen_process-to-kernel-power-processc.patch
Restoring kernel/power/process.c
Restoring include/linux/freezer.h

Removing patch 05-fix-pf_nofreeze-and-freezeable-race-2.patch
Restoring include/linux/freezer.h

Now at patch 04-fix-kthread_create-vs-freezer-theoretical-race.patch
```

Sometimes you may want `quilt` to revert patches util specific patch is on the top of the stack. To do this, use '`quilt pop`' with an argument being the name of the patch that you want to be on the top of the stack after the operation. Suppose, for instance, that you have applied the first ten patches from our example series, but now you want `04-fix-kthread_create-vs-freezer-theoretical-race.patch` to be on the top of the stack (ie. to become the last recently applied one). You can make this happen in the following way:

```
$ quilt pop 04-fix-kthread_create-vs-freezer-theoretical-race.patch
Removing patch 10-swsusp-remove-platform-callbacks-from-resume-code.patch
Restoring kernel/power/user.c

Removing patch 09-acpi-fix-suspend-resume-ordering.patch
Restoring kernel/power/main.c

Removing patch 08-swsusp-fix-sysfs-interface.patch
Restoring kernel/power/disk.c
Restoring kernel/power/main.c

Removing patch 07-power-management-use-mutexes-instead-of-semaphores.patch
Restoring drivers/base/power/resume.c
Restoring drivers/base/power/main.c
Restoring drivers/base/power/runtime.c
Restoring drivers/base/power/power.h
Restoring drivers/base/power/suspend.c

Removing patch 06-move-frozen_process-to-kernel-power-processc.patch
Restoring kernel/power/process.c
Restoring include/linux/freezer.h
```

```
Removing patch 05-fix-pf_nofreeze-and-freezeable-race-2.patch
Restoring include/linux/freezer.h

Now at patch 04-fix-kthread_create-vs-freezer-theoretical-race.patch
```

Analogously, you can make `quilt` apply patches until specific one is on the top of the stack:

```
$ quilt push 09-acpi-fix-suspend-resume-ordering.patch
Applying patch 05-fix-pf_nofreeze-and-freezeable-race-2.patch
patching file include/linux/freezer.h

Applying patch 06-move-frozen_process-to-kernel-power-processc.patch
patching file include/linux/freezer.h
patching file kernel/power/process.c

Applying patch 07-power-management-use-mutexes-instead-of-semaphores.patch
patching file drivers/base/power/main.c
patching file drivers/base/power/power.h
patching file drivers/base/power/resume.c
patching file drivers/base/power/runtime.c
patching file drivers/base/power/suspend.c

Applying patch 08-swsusp-fix-sysfs-interface.patch
patching file kernel/power/disk.c
patching file kernel/power/main.c

Applying patch 09-acpi-fix-suspend-resume-ordering.patch
patching file kernel/power/main.c

Now at patch 09-acpi-fix-suspend-resume-ordering.patch
```

As you can see in the above examples, for each applied or reverted patch `quilt` prints the names of the files modified in the process (in fact, these names are printed by the `patch` program introduced in Sec. 1.2, used by `quilt`). You can also make it print the names of the files modified by the most recently applied patch (ie. the one on the top of the stack) by using '`quilt files`':

```
$ quilt top
09-acpi-fix-suspend-resume-ordering.patch
$ quilt files
kernel/power/main.c
```

Additionally, it is possible to add some "external" patches to a `quilt` series. The recommended way of doing this is to use the '`quilt import`' command with the additional argument being the name of the file containing the patch that you want to add to the series.

If this command is used, the file with the patch is automatically copied to the `patches` directory and the `series` file is updated by adding the name of this file right after the most recently applied one. Thus the patch becomes the next one to be applied by `quilt`, although it is not applied automatically.

Our description of `quilt` is by no means a complete one. It may only allow you to get a general idea of how `quilt` works and how flexible it is. Generally speaking, it allows one to do much more than we have shown above. In particular, it automates the creation and updating of patches in a very convenient way, but this part of its functionality is beyond the scope of our discussion. For more information about `quilt` refer to the documentation distributed with it, available at `http://download.savannah.gnu.org/releases/quilt/` .

4.3 General idea of binary searching

Imagine that you have found a kernel bug, but you have no idea which of the thousand or more patches included in the tree that you are testing might have caused it to appear. In principle, you could revert the patches one by one and test the kernel after reverting each of them, but that would take *way* too much time. The solution is to use *binary searching*, also known as *bisection*.

To explain what it is we first assume that the kernel does not work for you any more after you have applied a series of n patches (n may be of the order of 1000). We also assume that it takes Δt minutes on the average to compile and run the kernel on your system after reverting one patch (Δt may be of the order of 10). Under these assumptions, if you tried to revert patches one by one and test the kernel each time, you could spend about $\Delta t \cdot n$ minutes worst-case before finding the offending patch. Of course, you might be lucky and the bug might have been introduced by the last patch in the series, but generally this is not very probable. More precisely, if you know nothing about the patches in question, you should assume that each of them is equally likely to have introduced the bug and therefore the probability of the bug being introduced by a specific patch is equal to $1/n$ (hence, the worst case is not very probable either, but that does not improve the situation very much).

Naturally, you can revert k patches out of n and then test the kernel to see whether or not one of these k patches has introduced the bug. Still, the question arises what number k should be equal to. To answer it, let's estimate the maximum expected time needed to find the patch that has introduced the bug under the assumption that k patches are reverted in the first step. For this purpose, let $A(k)$ denote the event that one of the k patches reverted in the first step has introduced the bug and let $B(k)$ denote the event that the bug has been introduced by one of the remaining $n - k$ patches. If $A(k)$ occurs, the time needed to find the buggy patch will not be greater than $T_A(k) = \Delta t \cdot k$ (we need to search a series of k patches and it takes Δt minutes to check one patch on the average) and the probability of occurrence of $A(k)$ is $p_A(k) = k/n$. In turn, if $B(k)$ occurs, the time needed to find the buggy patch will not be greater than $T_B(k) = \Delta t \cdot (n-k)$ and the probability of occurrence of $B(k)$ is $p_B(k) = (n - k)/n$. Hence, since $A(k)$ and $B(k)$ are mutually exclusive and one of them must occur, the maximum average time needed to find the buggy patch is given by

$$T(k) = T_A(k)p_A(k) + T_B(k)p_B(k) = \frac{\Delta t}{n}\left[k^2 + (n-k)^2\right] .$$

Now, it takes a little algebra to show that $T(k)$ attains minimum for $k = n/2$, so it is most reasonable to revert $n/2$ patches in the first step.

Further, if the kernel still doesn't work after reverting $n/2$ patches from the series, we get the problem that is formally equivalent to the initial one with the number of patches to search reduced by half. Then, we can repeat the above reasoning for $n' = n/2$ and it will turn out that the most reasonable thing we can do is to revert $n'/2$ patches and test the kernel.

On the other hand, if the kernel works after we have reverted $n/2$ patches from the series, we need to reapply some of the reverted patches and test it once again. In that case, however, we still have $n' = n/2$ patches to check and a reasoning analogous to the above one leads to the conclusion that the most reasonable number of patches to reapply before testing the kernel once again is $n'/2$.

Thus we can establish a procedure, referred to as bisection or binary searching, allowing us to reduce the number of patches to check by half in every step. Namely, if the initial number of patches to check is n, we revert $n/2$ patches and test the kernel. Next, depending on the result of the previous test, we either revert or reapply $n/4$ patches and test the kernel once again. Subsequently, depending on the result of the previous test, we either revert or reapply $n/8$ patches and test the kernel once again. By repeating this approximately $\log_2 n$ times we can reduce the number of suspicious patches to one and thus find the patch that has introduced the bug. It is not very difficult to observe that the average time needed to find the buggy patch this way is proportional to $\log_2 n$.

4.4 Binary searching with the help of `quilt`

Suppose that we have a `quilt` series of patches with the following contents of the `series` file:

```
01-kmemleak-base.patch
02-kmemleak-doc.patch
03-kmemleak-hooks.patch
04-kmemleak-modules.patch
05-kmemleak-i386.patch
06-kmemleak-arm.patch
07-kmemleak-false-positives.patch
08-kmemleak-keep-init.patch
09-kmemleak-test.patch
10-kmemleak-maintainers.patch
#11-new-locking.patch
#12-new-locking-fix.patch
13-vt-memleak-fix.patch
14-new-locking-fix.patch
15-fix-for-possible-leak-in-delayacctc.patch
```

and we cannot build the kernel after applying all of the above patches. To find the patch that causes the problem to appear, we can carry out a binary search.

Of course, having only 13 patches to check, we can use the 'quilt patches <file_name>' command to identify patches modifying the file which fails to compile, so that we can revert them and test the kernel without them, but if the tested patchset is huge, it usually is not a good idea to revert random patches from the middle of it.

Suppose that we have applied all of the patches from the series:

```
$ quilt push -a
Applying patch patches/01-kmemleak-base.patch
patching file include/linux/kernel.h
[..]
patching file kernel/delayacct.c
Now at patch patches/15-fix-for-possible-leak-in-delayacctc.patch
```

and it turns out that something is wrong, because we cannot compile the kernel any more. First, we obtain the number of applied patches:

```
$ quilt applied | wc -l
13
```

and we revert one half of them:

```
$ quilt pop 6
Removing patch patches/15-fix-for-possible-leak-in-delayacctc.patch
Restoring include/linux/delayacct.h
[..]
Restoring lib/Kconfig.debug
Now at patch patches/07-kmemleak-false-positives.patch
```

Then, the number of patches to revert or apply in the next step is 3 (we must remember this number, so it is best to write it down somewhere). We test the kernel and find that the problem is still appearing, so we revert 3 patches:

```
$ quilt pop 3
[...]
Now at patch patches/04-kmemleak-modules.patch
```

and note that the number of patches to check in the next step is 1.

Next, we test the kernel again and find that the problem has disappeared. We thus apply one patch:

```
$ quilt push
[...]
Now at patch patches/05-kmemleak-i386.patch
```

and note that the number of patches to check in the next step will depend on the result of the subsequent kernel test. Namely, if we test the kernel and the problem reappears, we will know that `05-kmemleak-i386.patch` is the source of it, since none of the "earlier" patches has caused it to be present. Otherwise, we will need to apply one patch more

```
$ quilt push
[...]
Now at patch patches/06-kmemleak-arm.patch
```

and if the problem reappears, we will know that this patch is "guilty". Otherwise, the next patch, which is `07-kmemleak-false-positives.patch`, will have to be the offending one, since the problem is not present with all of the patches "below" it applied.

Some more useful hints about using `quilt` for binary searching can be found in the Andrew Morton's article *How to perform bisection searches on -mm trees* (`http://www.zip.com.au/~akpm/linux/patches/stuff/bisecting-mm-trees.txt`). In particular, you can learn from it that if the `-mm` tree contains a series of patches similar to the following one:

```
patch-blah.patch
patch-blah-blah.patch
patch-blah-blah-fix1.patch
patch-blah-blah-fix2.patch
patch-blah-blah-fix3.patch
```

you should treat them as a single patch. That is, you should always revert them all or apply them all at once.

A practical example of binary searching with the help of `quilt` is shown in the movie available at `http://www.youtube.com/watch?v=LS_hTnBDIYk` .

4.5 Binary searching with the help of `git-bisect`

Since binary searching for buggy patches is a very important debugging technique, as far as the Linux kernel is concerned, and `git` is the primary versioning control system used by the kernel developers, the `git` package contains a tool that automates binary searching, called `git-bisect`. It is powerful and easy to use, so it can be recommended to inexperienced kernel testers.

To explain how to use `git-bisect`, we suppose that there is a problem in the `2.6.18-rc5` kernel and we do not know which commit has introduced it. We know, however, that it is not present in the `2.6.18-rc4` kernel.

We start the bisection by running

```
$ git-bisect start
```

and mark the current kernel version as the first known bad one:

```
$ git-bisect bad
```

Next, we use `gitk` to get the commit identifier associated with the `2.6.18-rc4` version of the kernel, which is `9f737633e6ee54fc174282d49b2559bd2208391d`, and mark this version as the last known good one:

```
$ git-bisect good 9f737633e6ee54fc174282d49b2559bd2208391d
```

Now, git-bisect will select a commit, more or less equally distant from the two corresponding to the first known bad and the last known good kernel versions, that we should test:

```
Bisecting: 202 revisions left to test after this
[c5ab964debe92d0ec7af330f350a3433c1b5b61e] spectrum_cs: Fix firmware
uploading errors
```

(using 'git-bisect visualize' we can see the current status of the bisection in gitk). Then, we need to compile, install and test the kernel.

Suppose that we have done it and the problem is still appearing, so we mark the current kernel version (ie. the one corresponding to the commit previously selected by git-bisect) as the first known bad one and git-bisect selects the next commit to test:

```
$ git-bisect bad
Bisecting: 101 revisions left to test after this
[1d7ea7324ae7a59f8e17e4ba76a2707c1e6f24d2] fuse: fix error case in
fuse_readpages
```

We compile the kernel, install and test it. Suppose that this time the problem is not present, so we mark the current kernel version, corresponding to the last commit selected by git-bisect, as the last known good one and let git-bisect select another commit:

```
git-bisect good
Bisecting: 55 revisions left to test after this
[a4657141091c2f975fa35ac1ad28fffdd756091e]
Merge gregkh@master.kernel.org:/pub/scm/linux/kernel/git/davem/net-2.6
```

Next, we compile, install and test the kernel, and so on, until we get a message similar to the following one:

```
$ git-bisect good
1d7ea7324ae7a59f8e17e4ba76a2707c1e6f24d2 is first bad commit
commit 1d7ea7324ae7a59f8e17e4ba76a2707c1e6f24d2
Author: Jan Kowalski <a@b>
Date:   Sun Aug 13 23:24:27 2006 -0700

    [PATCH] fuse: fix error case in fuse_readpages

    Don't let fuse_readpages leave the @pages list not empty when exiting
    on error.
[...]
```

which contains the identifier of the commit that, most probably, has introduced the bug.

Still, we need to make sure that the bug has *really* been introduced by this particular commit. For this purpose we return to the initial kernel version:

```
$ git-bisect reset
```

and try to revert the commit that we have identified as the source of the problem with the help of `git-bisect`:

```
$ git-revert 1d7ea7324ae7a59f8e17e4ba76a2707c1e6f24d2
```

If we are lucky, the commit will be reverted cleanly and we will be able to test the kernel without it to make sure that it is buggy. Otherwise, there are some commits that depend on this one and in fact we should revert them all for the final testing.

Although the binary searching in the above example is pretty straightforward, generally it can be more complicated. For example, we may be unable to test the commit selected by `git-bisect`, because some more commits must be applied so that we can build the kernel. In that case we need to tell `git-bisect` where to continue and '`git-reset --hard`' can be used for this purpose.

Suppose, for instance, that you have:

```
$ git-bisect start
$ git-bisect bad 5ecd3100e695228ac5e0ce0e325e252c0f11806f
$ git-bisect good f285e3d329ce68cc355fadf4ab2c8f34d7f264cb
Bisecting: 41 revisions left to test after this
[c1a13ff57ab1ce52a0aae9984594dbfcfbaf68c0] Merge branch
'upstream-linus' of
master.kernel.org:/pub/scm/linux/kernel/git/jgarzik/netdev-2.6
```

but you cannot compile the kernel version corresponding to the selected commit. Then, you can use '`git-reset --hard`' to manually select the next commit for `git-bisect`:

```
$ git-reset --hard c4d36a822e7c51cd6ffcf9133854d5e32489d269
HEAD is now at c4d36a8... Pull osi-now into release branch
```

Now, after running '`git-bisect visualize`' you will see that the commits `5ecd3100e695...` and `f285e3d329ce...` are still marked as "bad" and "good", respectively, but the commit `c4d36a822e7c51cd6ffcf9133854d5e32489d269` that you have selected is marked as "bisect" instead of `c1a13ff57ab1ce52a0aae9984594dbfcfbaf68c0`.

It is also possible that a new version of the kernel appears on the *kernel.org* server while you are carrying out a bisection search. In that case you may suspect that the bug under investigation has been fixed in the new kernel version, but nevertheless you may want to continue the bisection if it turns out not to be the case. Then, you can run

```
$ cp .git/BISECT_LOG ../bisect.replay
$ git-bisect reset
```

and use `git-pull`, as usual, to get the new kernel version (see Section 4.1). After testing it, if the bug is still present, you can do:

```
$ git-bisect replay ../bisect.replay
```

to return to the point at which you have "suspended" the binary search.

To summarize, the most often used commands related to binary searching with the help of `git-bisect` are:

- `git-bisect start` – starts a new binary search
- `git-bisect reset` – goes back to the initial kernel version and finishes the bisection
- `git-bisect good <commit>` – marks the commit given by `<commit>`, or the current commit, as corresponding to the last known good kernel version
- `git-bisect bad <commit>` – marks the commit given by `<commit>`, or the current commit, as corresponding to the first known bad kernel version
- `git-bisect visualize` – uses `gitk` to show changes between the last known good and the first known bad kernel versions

Additionally,

- `git-bisect replay` – may be used to return to the point at which the bisection has been "suspended" in order to test a new version of the kernel
- `git-reset --hard` – can be used to select the next commit for `git-bisect` manually

For more information about `git-bisect` see its manual page ('`man git-bisect`') that contains more detailed description of it. You can also refer to the `git` documentation available on the Web (eg. at `http://www.kernel.org/pub/software/scm/git/docs`).

A practical example of binary searching with the help of `git-bisect` is shown in the movie available at `http://www.youtube.com/watch?v=R7_LY-ceFbE`

4.6 Caveats

Although in general binary searching allows one to find patches that introduce reproducible bugs relatively quickly, you should be aware that it may fail, as well as any other debugging technique.

First of all, it tends to single out patches that *expose* problems, but they need not be the same as the patches that actually *introduce* them. Usually, the patch that introduces bugs breaks the kernel, but sometimes the breakage results from some other changes that would work just fine if the bugs were not present. You should always remember about it, especially when you report the problem to the kernel developers (ie. *never* assume blindly that the patch identified as "bad" by a bisection must be buggy).

Second, in some quite rare situations, the patch that introduces the problem observed by you simply cannot be identified. For example, there may be two or more patches that introduce problems with similar symptoms and in that case it is difficult to say what you are *really* observing. Apart from this, the buggy patch may belong to a series of several patches depending on each other and you can only compile the kernel with either all of them, or none of them applied. Moreover, two or more such series of patches may be mixed within the tree in a fashion that makes them impossible to untangle. If that happens, you will only be able to identify a range of commits including the buggy patch and that need not be very useful.

It is also possible to make a mistake in binary searching, even if you are using `git-bisect`. Yes, *it is*. For instance, if you run '`git-bisect good`' instead of '`git-bisect bad`', or vice

versa, by mistake at one point, the entire procedure will lead to nowhere. Another common mistake is to run '`make oldconfig`' (see Section 1.6) in every step of bisection in such a way that the kernel configuration is slightly different each time. This may cause the options that actually trigger the bug to be turned on and off in the process, in which case the binary search will not lead to any consistent result. To avoid this, it is recommended to preserve the kernel configuration file known to trigger the bug and copy it to the build directory (usually it is the same as the kernel source directory – see Section 1.6 for details) before each compilation of the kernel (you may be asked to set or unset some options anyway and in that case you will need to remember the appropriate settings and apply them every time in the same way).

Finally, if you know something about the series of patches containing the buggy one, you can make the binary search for it end (or *converge*, as it is usually said) faster. Namely, knowing what changes are made by different patches in the series, you can manually skip the patches that are surely not related to the observed problem. For example, if you have found a bug related to a file system, you should be able to omit the patches that are only related to networking, since most likely they have nothing to do with the bug. This way you can save several kernel compilations, which may be equivalent to a fair amount of time.

Rozdział 5

Reporting bugs

If you find a bug in the Linux kernel, you should notify the kernel developers of it as soon as reasonably possible. There are at least two important reasons to do this. First, the configuration of your system may be unique and you may be the only person seeing this particular problem. In that case, if you do not report it, then most likely the next versions of the kernel will not work properly on your computer. Second, the problem observed by you may be related to some other issues being investigated by the kernel developers and by reporting it you may provide them with a valuable data point.

You should not be afraid of reporting known bugs. At worst, if you report one, someone will tell you that it is known. However, if the problem is related to hardware, your report may contain the additional information needed to identify the source of it. For this reason, it generally is also a good idea to confirm problems reported by other people, if you are observing them too.

In general, the kernel developers' preferred way of reporting bugs is email, because it allows them to react quickly to the problems that are easy to fix. Still, later on, if the reported problem turns out to be difficult, you may be asked to open an entry in the Linux kernel bug tracking system at `http://bugzilla.kernel.org`. Usually, this requires you to do some more work than just sending an email message with a bug report, but it often is necessary to collect all information related to the reported bug in one place, so that it is easily accessible at any time.

Email messages containing bug reports should generally be sent to the *Linux Kernel Mailing List* (LKML) or to the mailing list dedicated to the affected subsystem. You may send bug reports to two or three mailing lists simultaneously, but if you send them to more than three lists at a time, people will likely get angry with you.

It also is a good idea to notify the maintainer of the affected subsystem and the maintainer of the tree in which the bug is present (eg. Andrew Morton, if the bug appears in the `-mm` tree) by adding their email addresses to the CC list of the bug report message. The email addresses of maintainers of the majority of kernel subsystems can be found in the `MAINTAINERS` file in the root of the kernel source tree.

If you know which patch has caused the problem to appear, you should also add the email address of its author to the CC list of your bug report (this address is usually present in the '`From:`' field of the patch header). Additionally, it is recommended to notify all of the people involved in the process of merging the patch (you can find their addresses in the

'`Signed-off-by:`' and '`Acked-by:`' fields of the patch header). This way you can increase the probability that someone "in the know" will notice the report and respond to it quickly. Apart from this, you should make it clear that your message contains a bug report, for example by adding the word "BUG" in front of the message's subject line.

Nevertheless, sometimes bug reports are not responded to even if they contain all of the right email addresses etc. If that happens to your bug report, you should first check if it has not been intercepted by a spam filter. This is easy if you have sent the report to a mailing list, since in that case it only is necessary to look into the list's archives to see if the message is there. If it turns out that the report has reached the list and no one is responding to it, the developers might have overlooked it or they are too busy to take care of it right now. In that case you should wait for some time (usually, a couple of days) and send it once again (if you resend the report, you may add the word "resend" to the message's subject to indicate that this is not the first time). If that does not help and there still is no response, it is best to open a new entry in the bug tracking system at `http://bugzilla.kernel.org` .

As far as the contents of bug reports are concerned, you should generally avoid putting irrelevant information into them. Of course, that may be difficult, because you may not know which information is relevant in given particular case. Still, you can always safely assume that if more information is needed, the kernel developers will ask you to provide it.

First of all you need to say what the bug is, which kernel version it is present in and how to reproduce it. You also need to describe the symptoms and include all of the corresponding kernel messages, if you can collect them (see Chapter 3). In particular, the following things should be present in a good bug report:

- description of the reported problem,
- version of the kernel in which the problem appears,
- the last known version of the kernel in which the problem does not appear (if applicable),
- architecture of the system on which the problem is observed (that may include some more detailed hardware information if the problem seems to be hardware-related),
- steps to reproduce the problem,
- kernel messages related to the problem (if available).

Additionally, if you know which patch has introduced the problem, the name of it or the identifier of its commit should also be included in the report. You can refer to the file `REPORTING-BUGS` in the root of the kernel source tree for more information about the reporting of bugs.

After reporting a bug you may be provided with a patch to test. In that case you ought to test it and report back to the developer who have sent it to you whether or not it fixes the bug. If it does not fix the bug, you will likely receive more patches to test. Otherwise, it is polite to thank the developer for his work.

The preparation of bug reports may be automated with the help of `ORT`. After running it:

```
$ ./ort.sh oops.txt
```

(`oops.txt` is a file with some scary kernel messages) you will need to specify the report type:

- short – basic information only
- custom – user-defined report contents
- template – report from a template

Next, you need to type in the required information and choose the options that you want. The report is then generated according to the template included in the `REPORTING-BUGS` file.

The newest version of `ORT` is available at `http://www.stardust.webpages.pl/ltg/files/tools/ort/` . If you use it, be careful not to generate reports containing too much irrelevant information, which unfortunately is quite easy to do with the help of this tool.

Rozdział 6

Testing of hardware

Before you carry out any serious kernel tests, you should make sure that your hardware is functioning correctly. In principle it should suffice to thoroughly test all of the hardware components once. Later, if you suspect that one of them may be faulty, you can check it separately once again. For example, hard disks are generally prone to failures, so you may want to test your hard disk from time to time.

For scanning the hard disk surface for unusable areas you can use the standard tool `badblocks`. After running it:

```
# /sbin/badblocks -v /dev/<your disk device>
```

you should learn relatively quickly if the disk is as good as you would want it to be.

Additional information on the hard disk state may be obtained from its S.M.A.R.T., by running

```
# smartctl --test=long /dev/<your disk device>
```

and then

```
# smartctl -a /dev/<your disk device>
```

to see the result of the test.

For memory testing you can use the program `Memtest86+` (it is recommended to download the ISO image from the project's web page at `http://www.memtest.org/` and run the program out of a CD). For this purpose you can also use the older program `Memtest86` (`http://www.memtest86.com/`) or any other memory-testing utility known to work well.

If you overclock the CPU, the memory, or the graphics adapter, you ought to resign from doing that while the kernel is being tested, since the overclocking of hardware may theoretically introduce some distortions and cause some random errors to appear.

It is worthy of checking if the voltages used to power the components of your computer are correct. You can do this with the help of the `lm_sensors` program. Alternatively, sometimes you can use a utility provided by the motherboard vendor for this purpose (unfortunately, these utilities are usually Windows-only).

Additionally, you should remember about various errors related to hardware that may appear even if the hardware is not broken. For example, cosmic rays and strong electromagnetic fields may sometimes cause hardware to fail and there are no 100% effective safeguards

against them. For this reason, various technologies, such as the ECC (*Error Correction Codes*), are developed in order to detect and eventually correct the errors caused by unexpected physical interactions of this kind. For instance, ECC memory modules store additional bits of information, often referred to as the parity bits, used to check if the regular data bits stored in the memory are correct and to restore the right values of that bits if need be. Of course, such memory modules are more expensive than the non-ECC modules of similar characteristics, but it generally is a good idea to use them, especially in mission critical systems.

While testing the kernel you can also encounter the so-called MCEs (*Machine Check Exceptions*) thrown by the CPU in some problematic situations. Some processors generate them whenever a parity error is detected in an ECC memory module and they can also be generated if, for example, there is a problem with the CPU's internal cache or when the CPU is overheating. In such cases the kernel may start to consider itself as "tainted" (see Section 2.6) and if it crashes in that state, the corresponding error message will contain the letter 'M' in the instruction pointer status line, eg.

```
EIP:   0060:[<c046c7c3>]    Tainted: PM     VLI
```

There is one more potential source of hardware-related problems that you should be aware of, which is your computer's BIOS (*Basic Input-Output System*). In the vast majority of temporary computers the BIOS, sometimes referred to as the platform firmware, is responsible for configuring the hardware before the operating system kernel is loaded. It also provides the operating system kernel with the essential information on the hardware configuration and capabilities. Thus, if the BIOS is buggy, the Linux kernel will not be able to manage the hardware in the right way and the entire system will not work correctly.

To check if your computer's BIOS is compatible with the Linux kernel you can use the *Linux-ready Firmware Developer Kit* `http://www.linuxfirmwarekit.org/` . Of course, if it turns out that the BIOS is not Linux-compatible, you will not be able to do much about that, except for updating the BIOS and notifying the mainboard vendor of the problem, but you will know that some issues are likely to appear. This, in turn, may help you assess whether the unexpected behavior of the kernel that you observe is a result of a software bug or it stems from the BIOS incompatibility.

Dodatek A

Dodatek A

A.1 Wysyłanie łatek

Chcesz wysłać łatkę poprawiającą znaleziony błąd?
Przeczytaj `Documentation/SubmittingPatches`, oraz zwróć szczególną uwagę na SECTION 3 - REFERENCES. Znajdują się tam odnośniki do dokumentów, z którymi należy się zapoznać przed wysłaniem łatki. Pamiętaj o wyłączeniu zawijania tekstu w kliencie poczty. Niektóre programy pocztowe np. Thunderbird lubią zmieniać białe znaki na inne białe znaki - zainstalowanie rozszerzenia do szyfrowania listów z reguły rozwiązuje takie problemy (jeśli chcesz używać Thunderbirda do wysyłania łatek, to polecam lekturę `http://mbligh.org/linuxdocs/Email/Clients/Thunderbird`).

Jeżeli chcesz wysłać całą serię łatek, to lepszym rozwiązaniem jest wykorzystanie skryptu `http://www.aepfle.de/scripts/42_send_patch_mail.sh` . Jeżeli nie chcesz się męczyć z rozgryzaniem wszystkiego, to zamieszczam krótką instrukcję:

- zainstaluj `sendmail` i `mutt`
- na początku skryptu wpisz „`sudo /etc/init.d/sendmail start`" a na końcu „`sudo /etc/init.d/sendmail stop`" - ograniczy to czas otwarcia potencjalnej dziury w systemie (sendmail ma bardzo długą historię luk).
- do pliku `.muttrc` wpisz „`set realname='Imię Nazwisko'`"
  i „`set from=jakiś_adres@email`"
- w 99 linijce skryptu zmień adres -bcc na swój
- zmień nazwę swojego hosta na nazwę bramy, wtedy nie będziesz otrzymywał listów zwrotnych

```
----- The following addresses had permanent fatal errors -----
<ktoś@gdzieś.com>
(reason: 553 5.1.8 <ktoś@gdzieś.com>... Domain of sender address
ktoś2@gdzieś2.com does not exist) [..]
```

Teraz trzeba utworzyć plik, który zostanie wysłany jako [PATCH 0/x], jego format jest następujący:

```
Subject: Jakiś temat
To: adres_jakiejś@listy.com
CC: adres_jakiegoś@odbiorcy1.com, adres_jakiegoś@odbiorcy2.com

Opis łatek, diffstat etc.
```

Również na początku każdej łatki dodajemy podobny nagłówek. Następnie tworzymy plik z serią łatek (taki sam jak do `quilt`) i już możemy wysłać wszystkie łatki:
`$ 42_send_patch_mail.sh -d plik_z_wiadomością.txt -s plik_z_serią_łatek`

Ciekawym skryptem umilającym wysyłanie łatek jest również `http://www.speakeasy.org/~pj99/sgi/sendpatchset` - jego niewątpliwą zaletą jest to, że nie wymaga instalacji programów sendmail i mutt. Wystarczy tylko stworzyć plik kontrolny:

```
SMTP: adres_serwera.smtp.z.którego@korzystamy.com
From: Imię i Nazwisko <nasz_adres@email>
To: Adresat1 <adres1@gdzieś.com>
Cc: Adresat2 <adres2@gdzieś.com>
Cc: Adresat3 <adres3@gdzieś.com>
Subject: [PATCH 1/n] Opis pierwszej łatki
File: ścieżka-do-łatki
Subject: [PATCH n/n] Opis ostatniej łatki
File: ścieżka-do-ostatniej-łatki
```

Następnie wysyłamy łatki:
`$ sendpatchset control`

A.2 System testowy

Nasz system testowy powinien być odseparowany od systemu, na którym normalnie pracujemy - nie powinniśmy w nim montować żadnych partycji z systemu stabilnego. Dzięki temu, w razie znalezienia jakiegoś poważnego błędu w systemie plików, lub innym podsystemie jądra, który wiąże się z utratą danych, nasz system stabilny nie powinien na tym ucierpieć. Jeżeli nowe jądro zniszczy nasz system testowy, to znaleźliśmy poważny błąd o którym powinniśmy powiadomić deweloperów systemu.

Kolejną rzeczą, o której warto jest pamiętać, to to, że możemy pracować na innym systemie z poziomu systemu stabilnego. Wystarczy zamontować partycję gdzieś w `/mnt`, a następnie za pomocą `chroot` przejść do drugiego systemu. Teraz możemy przygotować nowe jądro, skompilować i zainstalować go (najlepiej za pomocą jakiegoś skryptu, który za nas wszystko zrobi), nie odrywając się od normalnych zajęć. Gdy znajdziemy chwilę czasu, możemy zrestartować system i przejść do systemu testowego.

Przy wyborze dystrybucji służącej nam za system testowy, powinniśmy się kierować tylko i wyłącznie naszą wygodą - powinna nam ułatwić pracę, a nie przysparzać dodatkowej. Gdy

testujemy nowe wersje jądra, to chcielibyśmy też testować jego nową funkcjonalność, dlatego fajnie jest, gdy dystrybucja oferuje nowe wersje narzędzi i bibliotek. Dobrym wyborem jest testowa wersja Debiana - posiada aktualne wersje narzędzi, oraz starsze wersje `gcc` (niestety nie wszyscy deweloperzy sprawdzają czy ich kod działa po skompilowaniu inną wersją `gcc` niż ta dostarczana razem z ich ulubioną dystrybucją). Fedora Core posiada bardzo dobre wsparcie dla SELinuksa i wszystkich nowości, które oferują najnowsze wersje jądra.

A.3 KLive

KLive jest narzędziem, dzięki któremu deweloperzy Linuksa mogą się dowiedzieć jak długo dane drzewo było testowane. Dlaczego warto go używać? Linus Torvalds tak zaanonsował wydanie `2.6.15-rc5`

> „There's a rc5 out there now, largely because I'm going to be out of email contact for the next week, and while I wish people were religiously testing all the nightly snapshots, the fact is, you guys don't."

Deweloperzy nie są jasnowidzami i nie wiedzą ile osób i jak długo testowało konkretne wydanie - czasami zdarza się sytuacja, że wszystko działa jak należy i nikt nie zgłasza żadnych błędów - wtedy opiekun drzewa nie ma żadnej pewności, czy ktokolwiek testował dane wydanie. Jeżeli chcesz zacząć używać KLive, to na początku proponuje odwiedzić stronę projektu `http://klive.cpushare.com/` - można na niej znaleźć informacje o sposobie działania i instalacji programu (ta ostatnia jest bardzo prosta, wystarczy zainstalować wymagane pakiety, a następnie pobrać i uruchomić skrypt (`sh klive.sh --install`).

A.4 Jak zostać deweloperem Linuksa?

Czasami na LKML pojawia się to pytanie (ja na nie nie będę odpowiadał z prostego powodu nie jestem deweloperem), dlatego Greg KroahHartman napisał krótki dokument na ten temat. Jeżeli chcesz poznać odpowiedź na to pytanie, to po prostu przeczytaj `Documentation/HOWTO` :).

Dodatek B

Dodatek B

B.1 Jak pomóc w dalszym rozwoju podręcznika?

Wersja „źródłowa" podręcznika znajduje się pod adresem:
`http://www.stardust.webpages.pl/ltg/files/handbook-current.tar.bz2`
(przed przystąpieniem do modyfikacji warto sprawdzić, czy posiadamy najnowszą).

Po wprowadzeniu modyfikacji wystarczy zrobić

```
$ diff -uprN wersja-oryginalna/handbook.tex \\
 wersja-zmodyfikowana/handbook.tex > łatka.patch
```

i wysłać ją na adres `michal.k.k.piotrowski@gmail.com` (przy większych zmianach proszę też dodać CC na listę dyskusyjną LTG).

B.2 Gdzie uzyskać pomoc w testowaniu?

Jeśli potrzebujesz pomocy przy rozwiązaniu jakiegoś problemu związanego z jądrem Linux, masz wątpliwości czy znalazłeś(aś) błąd, to śmiało zadawaj pytania!

Strona pomocy w wiki `http://www.stardust.webpages.pl/ltg/wiki/index.php/Pomoc`

Strona listy dyskusyjnej `http://groups.google.com/group/linux-testers-group-pl`

B.3 Trochę o Linux Testers Group

Czyli odpowiedzi na wcześniej niezadane pytania.

P: Czym się zajmuje LTG?
O: Testowaniem jądra Linux.

P: Jak można dołączyć do LTG?
O: Wystarczy zacząć testować...

P: Gdzie jest jakaś strona domowa?
O: Pod adresem `http://www.stardust.webpages.pl/ltg/`

P: Jest jakaś lista dyskusyjna?
O: Jasne - pod adresem `http://groups.google.com/group/linux-testers-group-pl`

B.4 Przyczyny propagacji błędów do stabilnych wersji Linuksa

Poniższy artykuł został napisany dla Dragonia Magazine

Nie będzie to odkrywczym stwierdzeniem jak napiszę, że w Linuksie jest dużo błędów. Jednak skąd się one konkretnie biorą i jak można zapobiegać ich propagacji do stabilnych wersji systemu? Na to pytanie postaram się odpowiedzieć w poniższym artykule.

Na początku musimy sobie odpowiedzieć na proste pytanie - co to jest błąd w programie? Błąd w programie jest niezamierzoną (zdarzają się przypadki, że zamierzoną) pomyłką programisty, której skutkiem jest nieprawidłowe działanie programu. To nieprawidłowe działanie może się objawiać na kilka różnych sposobów:

- zawieszanie się programu
- podawanie niepoprawnych wyników obliczeń
- możliwość przejęcia kontroli nad programem przez włamywacza
- zafałszowywanie/uszkadzanie danych wejściowych i/lub wyjściowych
- niewydajne/nieefektywne algorytmy
- wycieki pamięci, sytuacje wyścigowe etc.

Błąd może wynikać z pomyłki podczas pisania kodu źródłowego programu, jak i na etapie planowania jego modelu. Zdarzają się również sytuacje, w których błąd powstaje za sprawą niepoprawnego działania narzędzi służących do kompilacji programu.

Błędy, które powstają na etapie pisania kodu źródłowego są często dużo łatwiejsze do wyeliminowania niż te, które powstają na etapie planowania modelu programu. Niewłaściwe decyzje podjęte w tej fazie mogą powodować występowanie bardzo trudnych do wyeliminowania błędów.

Jakie są główne przyczyny powstawania błędów w Linuksie? Zasadniczą przyczyną jest to, że jest to projekt duży i skomplikowany, dlatego większość ludzi pracujących nad systemem skupia się na swoich ulubionych podsystemach, które są im dobrze znane. Problem pojawia się wtedy, gdy ktoś próbuje zmodyfikować kod, którego dobrze nie zna i do końca nie rozumie. Dlatego też za sporą część błędów są odpowiedzialni ludzie, którzy Linuksem zajmują się dorywczo.

Kolejną przyczyną jest coś co się nazywa Nowy Model Rozwoju `http://www.stardust.webpages.pl/ltg/wiki/index.php/Proces_rozwoju_j%C4%85dra_Linux`. Dawniej Linux był rozwijany na zasadach podobnych do tych, na jakich jest rozwijane normalne oprogramowanie:

- wydanie wersji stabilnej i usuwanie błędów, które się do niej przedostały (np. wersje 2.2.x)

- cykl rozwojowy w którym jest dodawana nowa funkcjonalność, następnie następowała próba stabilizacji (np. wersje 2.3.x)
- kolejne wydania stabilne (np. wersje 2.4.x)

Jednak od wersji 2.6.8 model rozwoju uległ zmianie i teraz wygląda tak (na przykładzie wersji 2.6.20 i 2.6.21):

- wydanie wersji stabilnej 2.6.20, następnie są publikowane poprawki oznaczane jako 2.6.20.x
- praca nad nową wersją, dodawanie nowej funkcjonalności - 2.6.20-gitX (przeważnie około dwóch tygodni)
- zakończenie prac nad wersją rozwojową, rozpoczęcie cyklu stabilizacyjnego - 2.6.21-rcX
- wydanie stabilnej wersji 2.6.21

Taki przyspieszony model rozwoju ma swoje wady i zalety. Do wad można zaliczyć to, że postępuje on naprawdę bardzo szybko, więc dużo błędów może przejść z wersji rozwojowej do stabilnej. Do zalet powinniśmy zaliczyć to, że nie trzeba czekać na nową funkcjonalność przez długi okres czasu (dawne wersje niestabilne były rozwijane bardzo długo, a pierwsze wersje stabilne przeważnie trudno było za takie uznać).

Jaka jest przyczyna wprowadzania w Linuksie błędów związanych z bezpieczeństwem systemu? W doskonałej książce „*Secure Programming for Linux and Unix HOWTO*" `http://www.dwheeler.com/secure-programs/` David A. Wheeler wymienił kilka powodów `http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO.html#WHY-WRITE-INSECURE` dla których w programach powstają błędy, które mogą zostać wykorzystane przez włamywaczy. Ze swojej strony dorzucę tylko jeden powód, który nie znalazł się na tej liście - mała ilość odpowiednich programów (lub ich mała skuteczność) ułatwiających wykrywanie potencjalnych luk.

Deweloperzy Linuksa przywiązują bardzo duże znaczenie do jakości kodu. Przed zgłoszeniem do przeglądu nowego kodu, deweloper powinien upewnić się, że spełnia on wszystkie wymogi wymienione w `Documentation/SubmitChecklist`. Następnie kod jest czytany przez innych deweloperów, którzy zgłaszają swoje uwagi - najczęściej odnośnie:

- stylu kodowania - styl kodowania w Linuksie powinien być jednolity - jest to bardzo ważna sprawa, ponieważ ułatwia czytanie i rozumienie kodu (można się z nim zapoznać czytając `Documentation/CodingStyle`)
- zastosowanych algorytmów - każdemu zależy na tym, aby były one jak najwydajniejsze
- poprawności i przejrzystości struktur danych - nie ma gorszej rzeczy, niż nieprzemyślane struktury danych

W zasadzie trudno jest określić, jaka konkretnie część błędów (lub potencjalnych błędów) jest eliminowana na etapie przeglądania kodu.

Kolejnym krokiem jest testowanie kodu w drzewie danego podsystemu lub -mm, gdzie jest

on sprawdzany pod względem współgrania z innymi częściami jądra systemu. Jest to bardzo ważny etap, ponieważ w nim powinny zostać wyłapane wszystkie poważniejsze błędy. Następnie kod jest włączany do rozwojowej wersji drzewa Linusa i tam następuje ostateczna próba jego stabilizacji.

Włączenie kodu do Linuksa nie jest takie proste - musi on spełniać narzucone standardy. Najczęściej następuje to za pośrednictwem tak zwanej „sieci zaufania" - kod jest włączany do drzewa danego podsystemu i dopiero z niego trafia do mainline. Przyczyną takiego stanu rzeczy jest to, że Linus Torvalds nie ma czasu czytać każdej wprowadzanej poprawki i wychodzi z założenia, że opiekunowie podsystemów wiedzą lepiej jak powinny one wyglądać oraz, że włączone do nich łatki zostały już odpowiednio przetestowane.

Proces dbania o jakość kodu w Linuksie jest bardzo rygorystyczny - dużo bardziej niż w innych projektach. Dlaczego więc do wersji stabilnych przedostaje się tak dużo błędów? Ostatnia poprawka „stable" `2.6.20.2` składa się z ponad stu łatek poprawiających konkretne problemy.

Odpowiedź jest bardzo prosta - niewystarczająca ilość osób poświęca czas na testowanie i poprawianie błędów w nowej funkcjonalności.

Niektórzy uważają, że remedium na te problemy powinien być powrót do starego modelu rozwoju, jednak ma on bardzo poważne wady:

- na nową funkcjonalność trzeba czekać bardzo długo (ostatni duży cykl rozwojowy - 2.5.x - trwał ponad dwa lata!)
- deweloperzy ponoszą dodatkowe koszty czasowe związane z przenoszeniem nowej funkcjonalności do stabilnych jąder dystrybucyjnych - ten czas mogliby poświęcić na jej tworzenie, poprawianie błędów etc.
- długi czas potrzebny na stabilizacje po długim cyklu rozwojowym

Jak widać nowy model rozwoju ma poważne zalety, jednak bez wystarczającej ilości ludzi testujących wydania rozwojowe nie będzie się sprawdzał. Sprawa jest bardzo skomplikowana, ponieważ deweloperzy nie mogą poprawiać błędów o których istnieniu nie wiedzą. Większość problemów wychodzi dopiero, gdy inni ludzie zaczynają używać danego kodu. Sytuacja może się pogorszyć do tego stopnia, że nowe stabilne wersje Linuksa będą działały dobrze tylko na maszynach o zbliżonej konfiguracji do tych, na jakich były testowane ich wersje rozwojowe.

Dobrym rozwiązaniem obok zwiększenia liczby osób testujących kod, może być zmniejszenie szybkości wprowadzania zmian. Jednak czy naprawdę tego chcemy? Z jednej strony zahamowanie rozwoju Linuksa może mieć bardzo pozytywny wpływ na jego stabilność, z drugiej strony trzeba się liczyć z tym, że wszystkie potrzebne nowe funkcje będą wchodzić do niego z coraz większym opóźnieniem.

Wszystkich zainteresowanych zmianą tego stanu rzeczy zapraszam do współpracy oraz dyskusji na naszej liście `http://groups.google.com/group/linux-testers-group-pl`.

B.5 Licencja

Uznanie autorstwa 2.5 Polska
Wolno:

- kopiować, rozpowszechniać, odtwarzać i wykonywać utwór
- tworzyć utwory zależne
- użytkować utwór w sposób komercyjny

Na następujących warunkach:

- Uznanie autorstwa. Utwór należy oznaczyć w sposób określony przez Twórcę lub Licencjodawcę
- W celu ponownego użycia utworu lub rozpowszechniania utworu należy wyjaśnić innym warunki licencji, na której udostępnia się utwór.
- Każdy z tych warunków może zostać uchylony, jeśli uzyska się zezwolenie właściciela praw autorskich.

Powyższe postanowienia w żaden sposób nie naruszają uprawnień wynikających z dozwolonego użytku ani żadnych innych praw.

`http://creativecommons.org/licenses/by/2.5/pl/legalcode`

Oczywiście, jeżeli licencja CC uznanie autorstwa Ci nie odpowiada, to możesz użyć dowolnej licencji uznanej za licencje Open Source `http://www.opensource.org/licenses/`, jeśli tylko zachowasz informacje o autorach, oraz ludziach, którzy przyczynili się do powstania tego podręcznika.