

# Linux Kernel development

Ian McDonald

University of Waikato  
ian.mcdonald@jandi.co.nz

**Abstract.** This paper focuses on the process of development for the Linux kernel. This topic is approached both from a technical viewpoint and also the interactions with the open source community. The aim of the paper is for readers not to have the same difficulties the author experienced!

*Keywords:* Operating Systems

## 1 Introduction

This paper discusses the social aspects of kernel development, followed by tips for building a kernel. The paper then focusses on releasing code and lastly, but not least important, testing and debugging.

The paper highlights some of the lessons learned by the author and the methodology used in developing within the kernel.

## 2 Building a kernel

### 2.1 Maintaining a source code tree

Keeping in synchronisation with the Linux kernel source code tree takes a non-trivial amount of time. It is necessary to keep synchronised if it is intended to have the code merged into the tree or released as an ongoing codebase.

The Linux source code is maintained in a git [1] tree. Git is a new source code management tool created by Linus Torvalds. Git stores every change as a patch addressed by a SHA1 hash. Developers can make copies of git trees and because each patch is unique it is relatively simple for developers to synchronise their code base with other developers.

For developing in the kernel it is recommended to make a copy of Linus Torvalds' source code tree by issuing a statement similar to:

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

and then using `git-pull` to keep the git tree up to date. If the maintainer for the area that is being developed keeps a separate git tree, as most do, then you will need to create a copy of it by issuing a statement similar to:

```
git-clone --reference ~/linuxsrc/linus \  
git://git.kernel.org/pub/scm/linux/kernel/git/davem/net-2.6.git \  
~/linuxsrc/davem
```

When the above syntax is used it uses the local copy where possible making synchronisation quicker and uses a fraction of the bandwidth.

It is often useful to use a patch management tool on top of git such as stgit [2] or quilt [3]. These tools allow a series of patches to be maintained as a “stack” of patches which can be applied against different or updated trees.

Often a maintainer will “rebase” their tree which means they delete their original tree, make a fresh clone of Linus’ tree and then they will reapply any outstanding patches to that tree. If a maintainer rebases their tree then it is usually not possible to update it using the `git-pull` or `stg pull` command. Presuming the use of stgit, issue the command `stg export` to export the patches (this should also be done before any pull in case of a problem), save the files that are under `patches-branch` in a temporary directory, clone the git tree again to the latest tree and then import the patches into the new tree by `stg import`.

## 2.2 Use of distcc

To speed up compilation if multiple machines are available then distcc [4] can be used which is a distributed C compiler, and then specifying to `make` how many parallel threads to run using the `-j` parameter. For kernel development there are a couple of caveats to be aware of. The same version of the C compiler `gcc` must be installed on the machines that are being used as a compile pool. It is also important to specify only one target on the command line or else the kernel will continue to rebuild from scratch each time the `make` command is issued. For example to build a kernel and prepare for it to be installed the following commands could be issued:

```
make -j6 CC=distcc all  
make modules_install INSTALL_MOD_PATH=~ /tmp
```

## 2.3 Other resources

For an introduction to Linux development there are excellent resources available such as [5] and [6]. The use of simple tools should also not be underestimated such as the use of `grep -n -r phrase` to find a symbol in the kernel. The author also maintains a Wiki page at <http://wlug.org.nz/KernelDevelopment> which contains other tips.

## 3 Releasing code

An often cited mantra in the open source community is “release early and release often” [7]. This concept is useful even prior to code release.

It can be productive to talk about what you are planning to implement. This can be useful as other people can inform you if they are working on similar code as is often the case. It also gives people an opportunity to give feedback on whether the ideas are good or need refining.

Once the ideas have been crystallised it is often useful to release a code as a request for comment (RFC) if the code is not ready to be merged. This, again, allows discussion of the ideas being implemented.

The code will probably go through many, many iterations so if the code is rejected then it is important to work on the areas highlighted rather than becoming disheartened.

When developing code for the kernel it needs to be submitted to the maintainer of the subsystem. The MAINTAINERS file in the top level directory of the Linux source code tree contains the list of the maintainers. A posting to the linux-kernel list or other kernel mailing lists will usually produce a response on how the code is maintained if it is not clear.

Any code released should always be against the latest tree of the maintainer. One of the biggest reasons that code isn't accepted is that the maintainer cannot use it as it applies against an older version of Linux.

When code is submitted it must be able to be applied without problems by the maintainer. Prior to submitting test each patch with:

```
git-apply --check --whitespace=error < mypatch.diff
```

It is also required to choose a mail client to send the patch which does not alter whitespace such as spaces being substituted for tabs, breaking lines up or sending as HTML. The author uses KMail as he has found this is one of the few that will meet these criteria.

The code and patches should also fit into the existing kernel code style. The `Documentation` directory of the Linux source code contains many useful guides to assist with this.

It is recommended for a developer to submit kernel changes in multiple small patches rather than one large patch. This is that so the change can be reviewed in a series of steps which are simpler to read and find bugs in. If multiple patches are provided then part of the change can be accepted, rather than all rejected if one large patch is used.

## 4 Testing and debugging

It is important that code is well tested before it is released.

There are a number of virtualisation tools available to assist with testing software such as qemu [8], UML [9], VMWare [10] and Xen [11]. These tools can make testing easier if computers with sufficient processing power are used. Any software developed should also be tested on computers natively as well as this can uncover separate bugs.

There are debugging tools available that can be used on the kernel such as gdb. The simplest debugging tool to use for timing dependent code is the `printk`

statement in the code which acts like a `printf` statement, except the output goes to the kernel logging daemon. If it is desired to find which patch caused a bug then `git-bisect` can be used to “divide and conquer” the code base.

In the kernel there is also a mechanism for tracing code execution and capturing data called kprobes which allows hooks to be added into any function entry or exit dynamically without any changes being made to the original function. In our research we have taken an implementation of this for monitoring TCP and implemented it for DCCP.

With testing it is often necessary to transfer newly built kernels to multiple machines. This can be automated through a script such as:

```
#!/bin/bash
# syntax m machine_name directory version
H=$HOME
SRC=$H/linuxsrc/$2
VER=$3
rm $H/tmp/lib/modules/$VER/build
rm $H/tmp/lib/modules/$VER/source
rsync $SRC/System.map root@$1:/boot/System.map-$VER
rsync $SRC/arch/i386/boot/bzImage root@$1:/boot/vmlinuz-$VER
rsync -av $H/tmp/lib/modules/$VER root@$1:/lib/modules
```

## 5 Conclusion

Development in the Linux kernel is more than simply editing code and typing `make all`. It is the hope of the authour that this paper helps more people develop in the Linux kernel by taking into account other considerations and applying these.

## References

1. Web: git. <http://git.or.cz/> (Accessed 2006)
2. Web: Stacked git. <http://www.procode.org/stgit/> (Accessed 2006)
3. Web: Quilt patch management tools. <http://savannah.nongnu.org/projects/quilt/> (Accessed 2006)
4. Pool, M., et al.: Distcc: a fast, free distributed c/c++ compiler, 2002-. URL <http://distcc.samba.org>
5. Love, R.: Linux Kernel Development. Second edn. Novell Press (2005)
6. Web: Lxr. <http://lxr.linux.no/> (Accessed 2006)
7. Raymond, E.: Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. (2001)
8. QEMU, C.: Emulator. URL: <http://fabrice.bellard.free.fr/qemu>
9. Dike, J.: A user-mode port of the Linux kernel. Proceedings of the Annual Linux Showcase. Atlanta, GA, Oct (2000)
10. VMware, I.: The VMWare software package. See <http://www.vmware.com>
11. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. Proceedings of the nineteenth ACM symposium on Operating systems principles (2003) 164–177