Promising projects from the Linux kernel community

KERNEL TRICKS

Developers are constantly looking for new ways to interact with the versatile Linux kernel.

This month we study some innovative projects leading deep into kernel space.

BY JON MASTERS, JAN RÄHM, AND JOE CASAD

Most Linux users know they can rely on the kernel without ever giving it much thought. But if you do look a little deeper, you'll find some evolving technologies that extend the kernel in interesting ways. This month we cover some innovations at the edges of the Linux kernel.

Our first article describes the new userspace driver model in upcoming kernel versions. In our second story, Klaus Knopper looks at block device compression with the Cloop module, which lets Live CD developers put up to 2GBs on a 700MB CD. We'll also examine kernel-based virtualization with KVM, and we'll end with a look at the Flash Translation Layer and the LogFS flash filesystem.

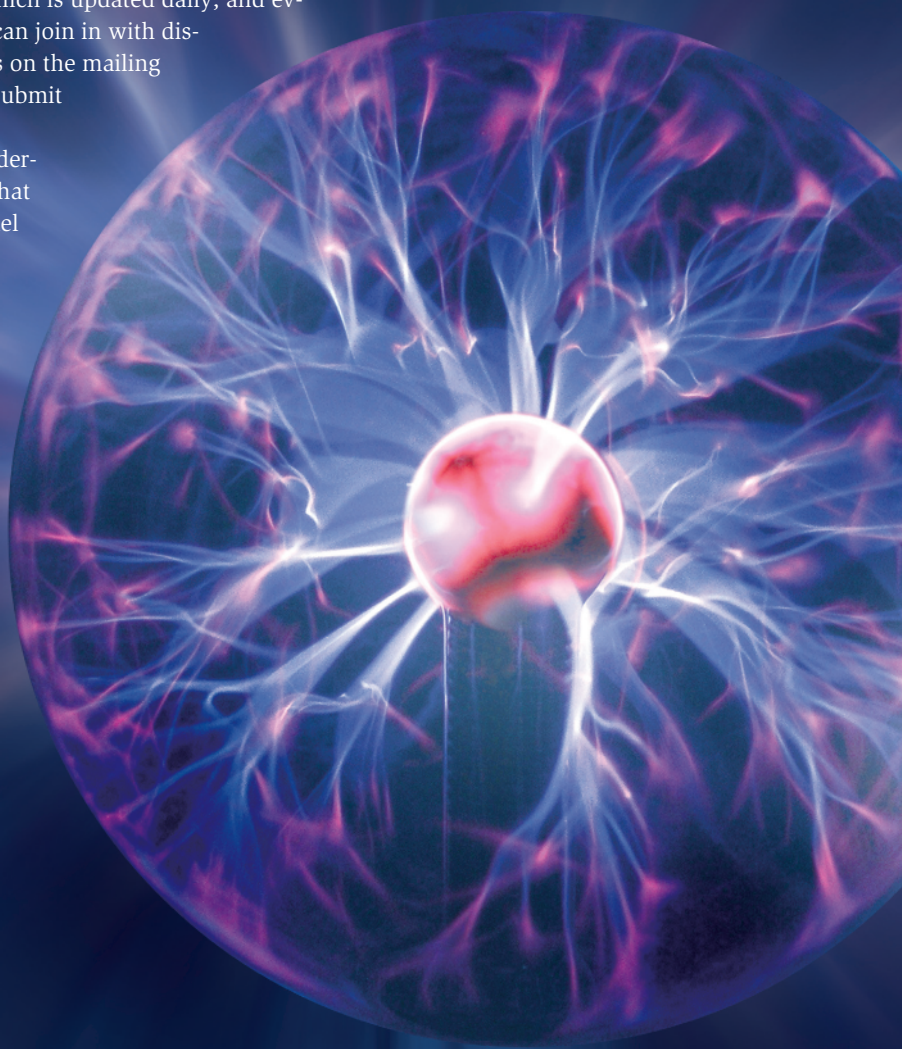
We hope you enjoy this month's Kernel Tricks cover story collection. But first, read on for a brief introduction to the brain of Linux.

The Linux Kernel

The term Linux, in the strictest sense, does not refer to the shiny collection of software components and applications that is often known as a Linux *distribution*. At the heart of that distribution is the Linux kernel itself – a complex collection of software routines that takes control whenever a machine first boots and periodically (many times each second) handles the resource requirements of applications, manages system hardware devices, and generally does a lot of the low level dirty work.

The Linux kernel is regarded as a fully functional, complex, and well-documented role model for open source software. Everyone has access to the source code, which is updated daily; and everyone can join in with discussions on the mailing lists or submit patches.

To understand what the kernel is really doing, take a look at Figure 1. The kernel appears in the center of the diagram, surrounded by a sample of the operations it assists in performing. On the left, hardware interrupts from devices such as disks, network cards, sound devices, graphics cards, and so on arrive at unpredictable times (data has perhaps been requested, but there is no way to know exactly when the device will actually deliver that data). In a similar way, the system timer (on most sys-



COVER STORY

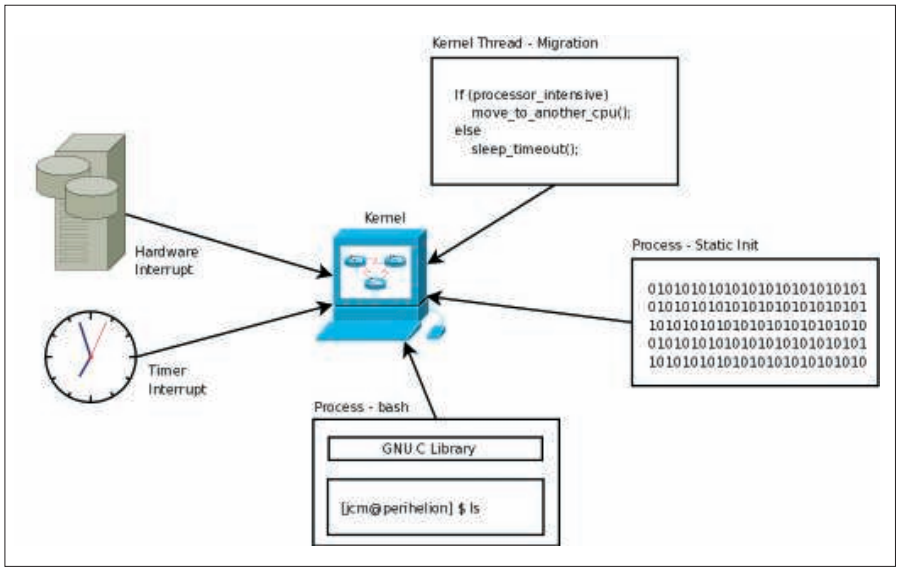


Figure 1: The Linux kernel manages processes, hardware interrupts, and housekeeping tasks.

tems) fires regularly to allow Linux to perform periodic housekeeping operations many times per second.

A kernel would not be particularly useful if it were not able to also service the user in the provision of support for running programs (called tasks, within the world of the Linux kernel). On the right side of the diagram are three tasks. Each represents a different kind of application. The first, at the bottom, is a regular user program – in this case, a Bash shell running on a user's desktop. Most programs don't communicate with the kernel directly. Instead, they use the GNU C Library functions, which in turn call standard kernel interfaces in order to provide required services.

Sometimes, an application does not use the C library but talks to the kernel more directly instead. This is the case with certain low-level, statically linked tools – for example, busybox (commonly used as a recovery tool, as well as in embedded gadgets such as the Nokia N810 Internet Tablet). In the diagram, you can see one of these special tools providing the services of *init*. Finally, at times the kernel runs special kernel code as if it were almost a regular program (but with privileges).

The key point of Figure 1 is that a kernel is far from magical. Its behaviors and processes can be explained. Many people believe that a kernel is somehow "running" all the time and constantly overseeing system operations. Although it is true that Linux does perform a variety of system monitoring functions, the kernel itself should be thought of more as a collection of software routines in a privileged library. These specially privileged routines are always executed in response to specific events, and at that time, whatever else was running is temporarily saved as the system switches into kernel code. Most of the time, kernel code runs either as a result of timers and hardware interrupts, or as the result of a request from an application. It also runs at startup and in various error conditions.

A Little History

The Linux operating system dates back to Linus Torvalds' first experiments in 1991. At that time, Linux was merely a toy operating system alternative. It ran only on Intel 386-compatible processors; however, it was very hackable, and it wasn't long before Linux was available for a wide variety of different machines – first came the more generic PC support, but thereafter, developers began to port Linux to other architectures.

After almost a decade of development, Linux 2.4 was released in the late 1990s. Linux 2.4 was significant because it was the first kernel release widely used by the masses – not just computer enthusiasts, but also those trying Linux for the first time on their desktop computers, servers, and even embedded gadgets. Version 2.4 was also used in the new generation of "Enterprise" distributions from major Linux vendors.

Once Linux 2.4 was safely out of the way, work began in earnest on a 2.5 development series, leading up to a major 2.6 kernel release a few years ago. Version 2.6 was a complete revolution for Linux. It had a rewritten scheduler (that can scale to many thousands of CPUs by now), a full device management subsystem with greatly improved and re-written dynamic device support, and literally countless other improvements that came as a result of the growing acceptance of Linux as a viable commercial offering. By this point, features weren't coming in from enthusiasts at nearly the same rate that they were being offered by well-known Linux hackers employed by Linux vendors and companies like IBM.

Linux Is Not Just Linus

Although it should be obvious by now, Linux is not the day-to-day product of one man's efforts. Linus Torvalds is widely recognized as the inventor and head of the Linux kernel project, but Linus serves more as a highly skilled project manager at this point (Figure 2). From a qualitative point of view, Linus

Figure 2: Linus Torvalds at the Kernel Summit 2007: the founding father of the Linux kernel still makes a major contribution to its development.

Anzeige
Nokia

Figure 3: Andrew Morton is the number two in kernel development. His -mm branch provides the impetus for new features in the Linux kernel.

may not be the main code developer; however, he still coordinates releases and work on the kernel, contributes to mailing lists, and takes an active part in discussions on new functions.

Below Linus Torvalds in the kernel hierarchy are a number of *maintainers* supervising work on different parts of the kernel. A maintainer decides on the development roadmap, typically in collaboration with other developers. Andrew Morton (Figure 3) is often considered the second in command of the kernel development team. Morton maintains the *-mm* kernel branch, as well as various subprojects, such as the development of the netdev driver and the Ext3 and Ext4 filesystems.

The *MAINTAINERS* document in the main branch of the kernel source package lists the various maintainers associated with Linux kernel development.

Which Linux Kernel?

Many variants of the Linux kernel are currently in use. For many reasons – including timing of individual product releases – few distributions ship exactly the same release of the kernel, and each may apply different patches that modify the kernel in some way. For example, last-minute updates might need to be made to various sound drivers just before release, and these may not yet be in the official Linux kernel from *kernel.org*. In some cases, Linux vendors apply their own optimizations as a means of adding value. Whatever the reason for the differences, you should know that it is unlikely that your Linux system is running the current "official" Linux kernel posted at *kernel.org*.

When you read the document, you may notice some gaps and that the column contains an *s* for *orphaned* status. These orphaned projects are projects that nobody is currently working on. Of course, the developer community is always happy for somebody to step in and assume responsibility for an orphaned development sector, but if you are new to kernel development, you should probably consider sticking to bug fixes and patches before attempting the maintainer role.

Linux Day-to-Day Development

Linux kernel development takes place on a minute-by-minute basis, around the clock, every day of the year. Central co-ordination happens using the Linux Kernel Mailing List (LKML), an archive of which you can find at *lkml.org*. Although general Linux kernel development discussion happens on the LKML, literally dozens of other popular emailing lists cover each of the many features within Linux.

The central repository of Linux kernel source is held at *kernel.org*, to which only a limited number of people have direct write access, but from which everyone can freely download the Linux source itself. Most developers, however, don't download source archives from *kernel.org*. They instead "pull" from Linus' git repository. Git is the Linux kernel source management utility written by Linus Torvalds.

Tree Structure

The kernel tree structure consists of four main trees above the branches or subtrees: *main 2.6.x*, *2.6.x.y-stable*, *2.6.x-git*, and *2.6.x-mm*. Linus Torvalds maintains the *main 2.6.x* tree. When a new kernel version is imminent, Torvalds opens a two-week window in which developers can send him their diffs.

In nearly every case, changes destined for the main tree have been tested in the *-mm* kernel over a period of days or weeks. Andrew Morton's *2.6.x-mm* tree differs from Torvald's Vanilla kernel in that it contains untested changes by Morton himself or resulting from patches reviewed by Morton. Morton integrates changes to all subsystems and patches from the mailing list with his version. The *-mm* branch is regarded as a playground for new developments and features. Once a patch has proved its reliability, it is very likely to make it into Torvalds' kernel tree.

Under normal circumstances, Linus publishes the first release candidate at the end of the two-week window. From then, major changes are typically ruled out, although Linus Torvalds might make an exception in case of critical

Constructive Contributions

Before you send bug reports or patches to the kernel community, it makes sense to keep to a couple of rules: Patches must be available as diffs. If you have clearly identified and removed the bug, the maintainer of the subsystem affected by the patch is the right person to contact. An undocumented patch is more or less useless. Add a description of the bug, the patch itself, and the effect of the patch to make it easier for the maintainer to investigate your proposal and, at the same time, approve your chances of it being accepted.

Claims such as "This patch makes 2000 lines of code superfluous …" or "I have tested this patch on five different architectures…" will grab the maintainer's attention. But mails saying things like: "I've been doing it this way for 20 years, so pay attention …" are likely to get on the maintainer's nerves.

In addition to an unequivocal description, submitters should note the following: Large patches that solve complex problems are easier for the maintainer to investigate if you split them into several smaller subpatches. Maintainers are unlikely to have much time to investigate work by a newbie near the end of a merge window.

Because kernel developers always check patches before accepting them, you should keep your patches as simple and transparent as possible. Unfinished patches accompanied with a promise to fix or complete later are typically ignored or dropped into a black hole. For more information, read the documents *applying-patches.txt*, *SubmittingPatches*, *SubmittingDrivers* ,and *SubmitChecklist* in the kernel source *Documentation* directory.

drivers or security patches. If everything works as planned, a second release candidate appears after another week. The differences between the two are typically no more than cosmetic code modifications. After the sixth release candidate, the process is normally completed, and the next stable version of the Linux kernel is ready.

The *2.6.x.y-stable* tree, the kernel versions with four-digit release numbers, see only minor changes, with the focus on security fixes. This branch is considered particularly reliable. If no version has a four-digit number, the highest 2.6.x version is typically the most stable.

The *2.6.x-git* tree is a daily snapshot of Torvald's kernel tree on the Git version control system initiated by Linus himself. This tree is far more experimental than a release candidate. The snapshots are taken automatically without developer interaction.

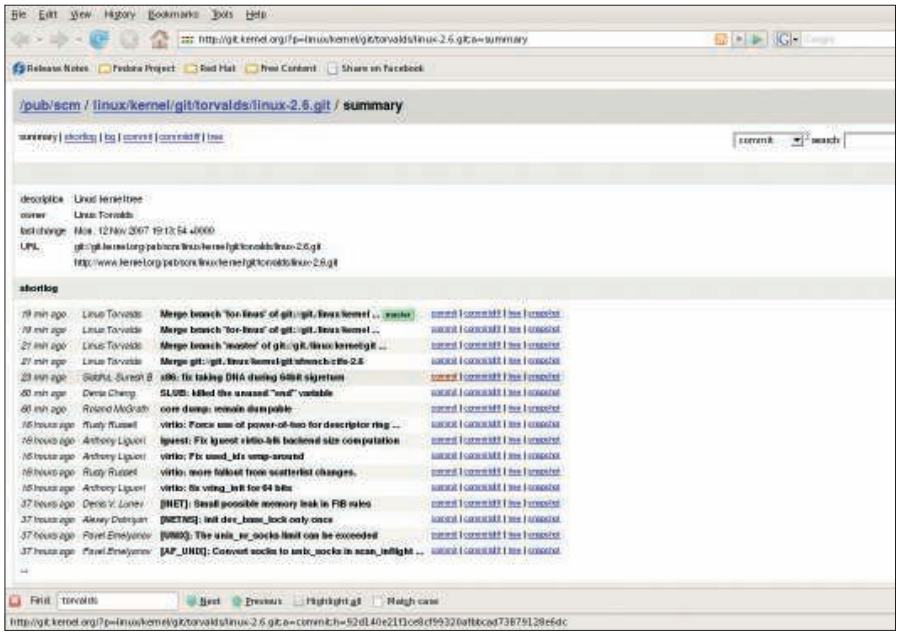


Figure 4: Linux source code is maintained through the git code management tool, which was designed and developed by Linus himself.

Pulling from the git Tree

If you want to experiment with living like a kernel developer, you can obtain a list of all of the public git repositories at *git.kernel.org*. Browse Linus' git tree at: *http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=summary*.

To obtain a copy of this development tree for your own uses, you first need to make sure you have the git tools installed. Linux distributions typically

include the git tools, which you can install using:

```
$ yum install git ➥
# Fedora and OpenSUSE systems
$ apt-get install git ➥
# Debian and Ubuntu systems
```

You can also install these tools from source, which is available directly from *git.kernel.org*.

Once you have installed git, you can use it to obtain the latest release of Linus' code tree:

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

This command will place a copy of Linus' kernel in a local directory called *linus-2.6*. To update this copy to the latest version, at any time, simply go to that directory and type:

```
$ git pull
```

You can find a complete set of git commands in the git reference documentation. Don't forget to experiment with the gitk utility if you would like a graphical source browser. Once you have copied the code to the *linus-2.6* directory, you can build the kernel as usual (see the box titled "Building the Kernel"), or you can just study the code to learn more about the art of kernel programming.

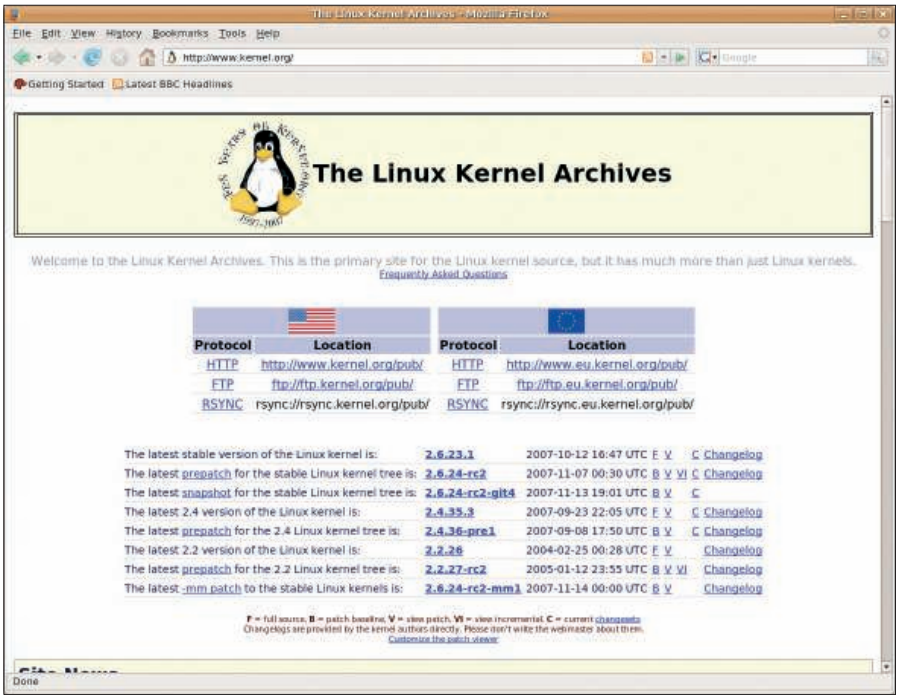


Figure 5: The Linux Kernel Archive at kernel.org is the epicenter for Linux kernel development.

Building the Kernel

The steps for building the Linux kernel may vary from one distribution to another (your distribution will supply an example of how to do this for modern 2.6 Linux kernels). This brief summary is intended for illustration purposes. Consult your vendor documentation.

In the case of a Fedora-like system, you'd first want to install the standard Fedora kernel config file from your */boot* directory:

```
$ cp /boot/config- ➥
`uname -r` .config
```

```
$ make oldconfig
```

Linux supports a variety of *make* targets, including both *oldconfig*, *menuconfig*, and many others. The *oldconfig* command will pull in existing selections from the *.config* file that was supplied by your Linux vendor, though there will likely be some differences in configuration options (you will be prompted about these, and offered some help in deciding appropriate options), so don't get too flustered when these come up. Once you have imported an existing configuration, you can also run the *menuconfig* target for a visual menu of available configuration options:

```
$ make menuconfig
```

After saving the kernel, you can build it using:

```
$ make
```

and install the updated kernel modules using:

```
$ make modules_install
```

Finally, you will need to rebuild any initial ramdisk (initrd/initramfs) that you are using. Your distribution will have more detail, but in the case of a Fedora system, try a command similar to:

```
$ mkinird /boot/initrd=➥
initrd-2.6.x.y 2.6.x.y
```

where *x.y* reflects whatever version of the 2.6 Linux kernel you just built. You will then need to update your bootloader (such as grub) by editing */boot/grub/menu.lst* to reference your new kernel before you can try test booting it. Most major Linux vendor distributions include tools to help with this. Fedora provides the *new-kernel-pkg* command, which automates the last two steps (rebuilding the initrd and installing the modified grub bootloader entries).

Further Reading

If you are getting started with Linux kernel development, try joining the Linux Kernel Newbies project *http://www.kernelnewbies.org/*. Here, you can meet with like-minded people who help each other learn about the Linux kernel. You might also want to try the Kernel Janitors Project – an ongoing effort at making various trivial cleanups to the Linux kernel source tree (a great way to get started). You'll even find a kernelnewbies Facebook group.

Although there is no substitute for reading the Linux Kernel Mailing List (LKML) if you want to keep on top of individual patches as they are posted, most people don't have nearly enough time in their day for such things. Especially those people who are not paid to follow the development so closely. This is why online resources such as Linux Weekly News (LWN) were created. For a small contribution of less than US$ 10 per month, you can read the latest (and highly detailed) summaries of kernel development from Jonathan Corbet (author of *Linux Device Drivers* and a long-time Linux kernel hacker). Even if you don't feel like subscribing, you can always read postings from previous weeks for free online. ■