The Linux Edge

Linus Torvalds

Linux today has millions of users, thousands of developers, and a growing market. It is used in embedded systems; it is used to control robotic devices; it has flown on the space shuttle. I'd like to say that I knew this would happen, that it's all part of the plan for world domination. But honestly this has all taken me a bit by surprise—I was much more aware of the transition from one Linux user to one hundred Linux users than the transition from one hundred to one million users.

Linux has succeeded not because the original goal was to make it widely portable and widely available, but because it was based on good design principles and a good development model. This strong foundation made portability and availability easier to achieve.

Originally Linux was targeted at only one architecture: the Intel 80386 CPU. Today Linux runs on everything from PalmPilots to Alpha workstations; it is the most widely ported operating system available for PCs. If you write a program to run on Linux, then, for a wide range of machines, that program can be "write once, run anywhere." It's interesting to look at the decisions that went into the design of Linux, and how the Linux development effort evolved, to see how Linux managed to become something that was not at all part of the original vision.

Linux today has achieved many of the design goals that people originally assumed only a microkernel architecture could achieve. When I began to write the Linux kernel, the conventional wisdom was that you had to use a microkernel-style architecture. However, I am a pragmatic person, and at the time I felt that microkernels (a) were experimental, (b) were obviously more complex, and (c) executed notably slower. Speed matters a lot in a real-world operating system, and I found that many of the tricks researchers were developing to speed microkernel processing could just as easily be applied to traditional kernels to accelerate their execution.

By constructing a general kernel model drawn from elements common to all typical architectures, the Linux kernel gets many of the portability benefits that otherwise require an abstraction layer, without paying the performance penalty paid by microkernels.

By allowing for kernel modules, hardware-specific code can often be confined to a module, keeping the core kernel highly portable. Device drivers are a good example of effective use of kernel modules to keep hardware specifics in the modules.

This is a good middle ground between putting all the hardware specifics in the core kernel, which makes for a fast but unportable kernel, and putting all the hardware specifics in user space, which results in a system that is either slow, unstable, or possibly both.

But Linux's approach to portability has been good for the development community surrounding Linux as well. The decisions that motivate portability also enable a large group to work simultaneously on parts of Linux without the kernel getting beyond my control. The architecture generalizations on which Linux is based give me a frame of reference to check kernel changes against, and provide enough abstraction that I don't have to keep completely separate forks of the code for separate architectures. So even though a large number of people work on Linux, the core kernel remains something that I can keep track of. And the kernel modules provide an obvious way for various programmers to work independently on parts of the system that should be independent.

I'm sure we made the right decision with Linux to do as little as possible in the kernel space. At this point I don't envision major updates to the kernel. A successful software project should mature at some point, after which the pace of change within the project typically slows down. There aren't a lot of major new innovations in store for the kernel. It's more a question of supporting a wider range of systems than anything else: taking advantage of Linux's portability to bring it to new systems.

Some particular application areas will continue to drive kernel development. Web serving has always been an interesting problem, because it's the one real application that is really kernel intensive. In a way, Web serving has been dangerous for me, because I get so much feedback from the community using Linux as a Web serving platform that I could easily end up optimizing only for Web serving. I have to keep in mind that Web serving is an important application but not everything.

I want Linux to be on the cutting edge, and even a bit past the edge, because what's past the edge today is what's on your desktop tomorrow. In the near future, areas for development are clustering, Symmetric Multi-Processing (SMP), and embedded systems. But at this point, the most exciting developments for Linux will happen in user space, not kernel space. The changes in the kernel will seem small compared to what's happening further out in the system. From this perspective, where the Linux kernel will be isn't as interesting a question as what features will be in Red Hat 17.5 or where Wine (the Windows emulator) is going to be in a few years.

In 15 years, I expect somebody else to come along and say, hey, I can do everything that Linux can do but I can be lean and mean about it because my system won't have 20 years of baggage holding it back. They'll say Linux was designed for the 80386 and the new CPUs are doing the really interesting things differently. Let's drop this old Linux stuff. This is essentially what I did when creating Linux. And in the future, they'll be able to look at our code, use our interfaces, and provide binary compatibility, and if all that happens I'll be happy. C

The most exciting developments for Linux will happen in user space, not kernel space. The changes in the kernel will seem small compared to what's happening further out in the system.

LINUS TORVALDS (torvalds@transmeta.com) works at Transmeta Corporation.