Tanenbaum-Torvalds Debate: Part II

Preface

Looks like it is microkernel debate time again. Before getting into technical arguments, there are a couple of things I want to say. Many people have said or suggested that Linus and I are enemies or something like that. That is totally false. I met him once. He is a pleasant fellow and very smart. We may disagree on some technical issues, but that doesn't make us enemies. Please don't confuse disagreements about ideas with personal feuds. I have nothing against Linus and have great respect for what he has accomplished.

In the unlikely event that anyone missed it, a couple of years ago Microsoft paid a guy named Ken Brown to write a book saying Linus stole Linux from my MINIX 1 system. I refuted that accusation pretty strongly to clear Linus' good name. I may not entirely agree with the Linux design, but Linux is his baby, not my baby, and I was pretty unhappy when Brown said he plagiarized it from me.

Before getting into the details, I also want to say that my interest is not in microkernels per se. My interest is in building highly reliable (and secure) operating systems and I think microkernels are a good tool for doing that. More below on this.

Engaging mouth

There is supposedly a sign at the Air Force Academy in Colorado that reads:

> Be sure brain is in gear before engaging mouth

I don't know if it is really there, but I think it is a good idea in any event.

Over the years there have been endless postings on forums such as Slashdot about how microkernels are slow, how microkernels are hard to program, how they aren't in use commercially, and a lot of other nonsense. Virtually all of these postings have come from people who don't have a clue what a microkernel is or what one can do. I think it would raise the level of discussion if people making such postings would first try a microkernel-based operating system and then make postings like "I tried an OS based on a microkernel and I observed X, Y, and Z first hand." Has a lot more credibility.

The easiest way to try one is to go download MINIX 3 and try it. It is free and open source (BSD license) and once you have the CD-ROM image, you can burn it onto a CD-ROM, boot from it, and log in as root. However to actually do anything useful, you need to allocate a disk partition (1 GB will do) and install MINIX 3 on it. Please print and read the setup manual first. Installation takes maybe 10 minutes. Then install all the packages on the CD-ROM, as described in the setup manual. Now start X and you are now able to get some real experience. Try rebuilding the entire operating system as described in the manual. The whole build, kernel + user-mode drivers and all the user-mode servers (125 compilations in all) takes about 5-10 seconds.

Please be aware that MINIX 3 is **not** your grandfather's MINIX. MINIX 1 was written as an educational tool; it is still widely used in universities as such. Al Woodhull and I even wrote a textbook about it. MINIX 3 is that plus a *start* at building a highly reliable, self-healing, bloat-free operating system, possibly useful for projects like the $100 laptop to help kids in the third world and maybe embedded systems. MINIX 1 and MINIX 3 are related in the same way as Windows 3.1 and Windows XP are: same first name. Thus even if you used MINIX 1 when you were in college, try MINIX 3; you'll be surprised. It is a minimal but functional UNIX system with X, bash, pdksh, zsh, cc, gcc, perl, python, awk, emacs, vi, pine, ssh, ftp, the GNU tools and over 400 other programs. It is all based on a tiny microkernel and is available right now.

So **PLEASE** no more comments like "If Tanenbaum thinks microkernels are so great, why doesn't he write an OS based on one?" He did. (Actually, he, his students, and his programmers did). It is definitely not as complete or mature as Linux or BSD yet, but it clearly demonstrates that implementing a reliable, self-healing, multiserver UNIX clone in user space based on a small, easy-to-understand microkernel is doable. Don't confuse lack of maturity (we've only been at it for a bit over a year with three people) with issues relating to microkernels. We can and will add more features and port more software over time (your help is very welcome). Over 400,000 visits to the MINIX 3 site have been recorded since it went live at the end of October 2005. Try it yourself and see.

The Paper

Recently, my Ph.D. student Jorrit Herder, my colleague Herbert Bos, and I wrote a paper entitled Can We Make Operating Systems Reliable and Secure? and submitted it to IEEE Computer magazine, the flagship publication of the IEEE Computer Society. It was accepted and published in the May 2006 issue. In this paper we argue that for most computer users, reliability is more important than performance and discuss four current research projects striving to improve operating system reliability. Three of them use microkernels. IEEE put the paper on its Website. Someone then posted a link to it to Slashdot, thus rekindling an ancient discussion about microkernels vs. monolithic systems. While I cheerfully admit to coauthoring the paper, I certainly didn't expect it to reboot the 'Linux is obsolete' debate.

Linus then responded and it looks like we are about to have another little debate. OK, but please, everyone, let's stick to the technical issues.

Linus' Argument

Linus' basic point is that microkernels require distributed algorithms and they are nasty. I agree that distributed algorithms are hell on wheels, although together with Maarten van Steen I wrote a book dealing with them. I have also designed, written and released two distributed systems in the past decade, Amoeba (for LANs) and Globe (For WANs). The problem with distributed algorithms is lack of a common time reference along with possible lost messages and uncertainty as to whether a remote process is dead or merely slow. None of these issues apply to microkernel-based operating systems on a single machine. So while I agree with Linus that distributed algorithms are difficult, that is not germane to the discussion at hand.

Besides, most of the user-space components are drivers, and they have very straightforward interactions with the servers. All character device drivers obey pretty much the same protocol (they read and write byte streams) and all block device drivers obey pretty much the same protocol (they read and write blocks). The number of user-space servers is fairly small: a file server, a process server, a network server, a reincarnation server, and a data store, and a few more. Each has a well-defined job to do and a well-defined interaction with the rest of the system. The data store, for example, provides a publish/subscribe service to allow a loose coupling between servers when that is useful. The number of servers is not likely to grow very much in the future. The complexity is quite manageable. This is not speculation. We have already implemented the system, after all. Go install MINIX 3 and examine the code yourself.

Linus also made the point that shared data structures are a good idea. Here we disagree. If you ever took a course on operating systems, you no doubt remember how much time in the course and space in the textbook was devoted to mutual exclusion and synchronization of cooperating processes. When two or more processes can access the same data structures, you have to be very, very careful not to hang yourself. It is exceedingly hard to get this right, even with semaphores, monitors, mutexes, and all that good stuff.

My view is that you want to avoid shared data structures as much as possible. Systems should be composed of smallish modules that completely hide their internal data structures from everyone else. They should have well-defined 'thin' interfaces that other modules can call to get work done. That's what object-oriented programming is all about--hiding information--not sharing it. I think that hiding information (a la Dave Parnas) is a good idea. It means you can change the data structures, algorithms, and design of any module at will without affecting system correctness, as long as you keep the interface unchanged. Every course on software engineering teaches this. In effect, Linus is saying the past 20 years of work on object-oriented programming is misguided. I don't buy that.

Once you have decided to have each module keep its grubby little paws off other modules' data structures, the next logical step is to put each one in a different address space to have the MMU hardware enforce this rule. When applied to an operating system, you get a microkernel and a collection of user-mode processes communicating using messages and well-defined interfaces and protocols. Makes for a much cleaner and more maintainable design. Naturally, Linus reasons from his experience with a monolithic kernel and has arguably been less involved in microkernels or distributed systems. My own experience is based on designing, implementing, and releasing multiple such operating systems myself. This gives us different perspectives about what is hard and what is not.

For a different take on the whole issue of reliable operating systems, see this piece by Jonathan Shapiro entitled Debunking Linus's Latest.

Are Microkernels for Real?

In a word: yes. There were endless comments on Slashdot Monday (May 8) of the form: "If microkernels are so good, why aren't there any?" Actually there are. Besides MINIX 3, there are:

- QNX
- Integrity
- PikeOS
- Symbian
- L4Linux

- Singularity
- K42
- Mac OS X
- HURD
- Coyotos

QNX is widely used in real commercial systems. Cisco's top-of-the-line router uses it, for example, and I can assure you, Cisco cares a **LOT** about performance.

One of the leading operating systems in the military and aerospace markets, where reliability is absolutely critical is Green Hills' Integrity, another microkernel.

PikeOS is another microkernel-based real-time system widely used in defense, aerospace, automotive, and industrial applications.

Symbian is yet another popular microkernel, primarily used in cell phones. It is not a pure microkernel, however, but something of a hybrid, with drivers in the kernel, but the file system, networking, and telephony in user space.

I could go on and on, but clearly in applications where reliability and security are mission critical, designers have often chosen microkernel-based OSes. While Linus may not be personally interested in embedded real-time systems, (where performance, reliability, and security are paramount), these markets are huge and many companies involved think that microkernels are the way to achieve these goals.

Looking at the PC world we find L4Linux, which was written by Hermann Härtig's group at the Technical University of Dresden. It runs all of Linux in user space on top of the L4 microkernel with a performance loss of only a couple of percent. But using the microkernel allowed the TUD people to build new systems such as DROPS (real time) and NIZZA (security) on top of L4 but still have access to full Linux without having to modify it for these new features. In this way, they can experiment with new facilities and still run legacy programs. Other groups are also using the L4 microkernel for operating systems research, such as Wombat, a paravirtualized Linux designed to support legacy applications in embedded systems. Another L4-based OS is TUD-OS and there are more.

Microsoft also has interest in microkernels. It understands the maintenance headache of monolithic kernels like no one else. Windows NT 3.1 was a half-hearted attempt at a microkernel system, but it wasn't done right and the performance wasn't good enough on the hardware of the early 1990s, so it gave up on the idea for a while. But recently, it tried again on modern hardware, resulting in Singularity. Now I know that a lot of people assume that if Microsoft did it, it must be stupid, but the people who drove the Singularity project, Galen Hunt and Jim Larus, are very smart cookies, and they well understand that Windows is a mess and a new approach is needed. Even the people working on Vista see they have a problem and are moving drivers into user space, precisely what I am advocating.

About 10 years ago IBM began developing a new high-performance operating system from scratch for its very large customers. An explicit design goal was to move system functionality from the kernel to servers and application programs, similar to a microkernel. This system, called K42, has now been deployed at the DoE and elsewhere.

Mac OS X is sort of microkernelish. Inside, it consists of Berkeley UNIX riding on top of a modified version of the Mach microkernel. Since all of it runs in kernel mode (to get that little extra bit of performance) it is not a true microkernel, but Carnegie Mellon University had Berkeley UNIX running on Mach in user space years ago, so it probably could be done again, albeit with a small amount of performance loss, as with L4Linux. Work is underway to port the Apple BSD code (Darwin) to L4 to make it a true microkernel system.

Although high on promise and low on execution, GNU HURD is also based on a microkernel. Actually two of them. The first version was based on Mach; the second on L4. A third version will probably be based on yet another microkernel, Coyotos. HURD was designed by Richard Stallman, the author of emacs, gcc, and much other widely used software as well as creator of the GPL and winner of the prestigious MacArthur 'Genius' Award.

Another microkernel system in development is Coyotos, the successor to EROS. Here the focus is more on security than reliability, but the two issues are related in that bloated kernels can lead to problems on both fronts.

And I haven't even discussed hypervisors, such as Xen and Trango, which are quite different from microkernels in many ways, but do share the characteristic of having only a tiny bit of code running in kernel mode, which I believe is the key to building a reliable and secure system.

While MINIX 3, QNX, Integrity, PikeOS, Symbian, L4Linux, Singularity, K42, HURD, Coyotos, and others are an eclectic bunch, clearly I am not alone in seeing something in microkernels. If you are wondering why microkernels aren't even more widely used, well, there is a lot of inertia in the system. Why haven't Linux or Mac OS X replaced Windows? Well, there is a lot of inertia in the system. Most cars in Brazil can run on home-grown ethanol so Brazil uses relatively little oil for vehicle fuel. Why doesn't the U.S. do that and reduce its dependence on the volatile Middle East? Well, there is a lot of inertia in the system. Getting people to change, even to superior practices, is very hard.

What Am I Trying to Prove?

Actually, MINIX 3 and my research generally is **NOT** about microkernels. It is about building highly reliable, self-healing, operating systems. I will consider the job finished when no manufacturer anywhere makes a PC with a reset button. TVs don't have reset buttons. Stereos don't have reset buttons. Cars don't have reset buttons. They are full of software but don't need them. Computers need reset buttons because their software crashes a lot. I know that computer software is different from car software, but users just want them both to work and don't want lectures why they should expect cars to work and computers not to work. I want to build an operating system whose mean time to failure is much longer than the lifetime of the computer so the average user never experiences a crash. MINIX 3 has many specific reliability features. While we are by no means done (e.g. virtual memory is on the agenda for later this year), I think improving reliability is the greatest challenge facing operating systems designers at this moment. The average user does not care about even more features or squeezing the last drop of performance out of the hardware, but cares a lot about having the computer work flawlessly 100% of the time and never crashing. Ask your grandma.

So what do microkernels have to do with this goal? They make it possible to build self-healing systems. That is what I care about and my research is about. Moving most of the OS to a bunch of user processes, one for each driver plus various servers, does not reduce the number of bugs in the code but it greatly reduces the ability of each bug to do serious damage and it also reduces the size of the trusted computing base. In our design, when most drivers fail, the reincarnation server can restart a fresh copy, and optionally save the core image of the dead driver for debugging, log the event, send email to an administrator or developer, etc. The system continues to run, and at the very least can be shut down gracefully without loss of work or data. Some components, such as the reincarnation server itself, the file server, and the process server are critical, and losing them crashes the system, but there is absolutely no reason to allow a faulty audio, printer, or scanner driver to wipe out the system. They should just be restarted and work should continue. This is our goal: systems that can detect and repair their own faults. You can do this easily in a microkernel system. It is much harder in a monolithic kernel, although researchers at the University of Washington have done some good work with Nooks and a group at the University of Karlsruhe has also done interesting work with virtual machine technology.

Linus on Linux

Late update. Linus has come to realize that Linux has become awfully bloated. See here for his comments.

Homework

Before pontificating about what microkernels can and cannot do, go get and try MINIX 3 and be an informed pontificator. It increases your credibility. To learn more about the MINIX 3 design, see the paper in the May IEEE Computer, this paper on modular programming that just appeared in USENIX *;login*, or this technical report.

If you have made it this far, thank you for your time.

Andy Tanenbaum, 12 May 2006