Debunking Linus's Latest

Jonathan Shapiro, Ph.D.
Systems Research Laboratory
Dept. of Computer Science
Johns Hopkins University

May 11, 2006

Well, it appears to be 1992 all over again. Andy Tanenbaum is once again advocating microkernels, and Linus Torvalds is once again saying what an ignorant fool Andy is. It probably won't surprise you much that I got a bunch of emails asking for *my* thoughts on the exchange. I should probably keep my mouth shut, but here they are. Linus, as usual, is strong on opinons and short on facts.

For those of you who are coming to this late, Andy's article *Can We Make Operating Systems Reliable and Secure?* appeared in the May 2006 **IEEE Computer**. Linus's response appeared online (in *email*). The web-based version of this note provides links to both.

May 15 Addendum: Andy has written his own rebuttal at www.cs.vu.nl/˜ast/reliable-os, which is also good reading.

1 Some Thoughts on Tanenbaum's Article

The Tanenbaum article gives a nice overview of issues in operating system reliability, and a general audience introduction to Microsoft's *Singularity*, the *L4* work on operating system rehosting (which is *not* paravirtualization), and Tanenbaum's own work on *Minix-3*. It is a general article for a general audience, and one should not confuse it for any sort of scholarly treatment of the area.

This kind of writing, by the way, is very hard for experts to do. We get caught in the details of our areas, and it is very hard to come to the surface far enough to explain this stuff to a general audience — or even to our colleagues. The space and citation limits for *IEEE Computer* add further challenges, so take what follows with caution. There is simply no way that Andy could have addressed such complex issues in a balanced way in the space that he had.

Some issues: Concerning paravirtualization, the article doesn't give credit to the right group. It doesn't mention two systems that have been demonstrating his case in research and commercial environments for decades. Finally, the article neglects two obvious facts that strongly support Andy's argument. Let me deal with each of these issues briefly, and then turn to Linus.

1.1 Paravirtualization

Paravirtualization is an important idea, both in its own right and as a partial solution for reliability. It is going to be critical in the success of the Intel and AMD hardware virtualization support.

The credit for this idea and its development properly belongs with the Xen team) at Cambridge University (see: *Xen and the Art of Virtualization*, not with the L4 team in Dresden. The idea of adapting paravirtualization to a microkernel for use *within* an operating system is, frankly, silly. Several existing microkernels, including L4.sec (though not L3 or L4), KeyKOS, EROS, Coyotos, and CapROS, have isolation and communication mechanisms that are already better suited to this task. My list intentionally does *not* include Minix-3, where IPC is unprotected. The major change between L4 and L4.sec is the migration to protected IPC.

In practice, the reason that the L4 team looked at paravirtualization was to show that doing it on a microkernel was actually *faster* than doing it on Xen. This was perfectly obvious to anybody who had actually read the research literature: the L4Linux performance was noticeably better than the Xen/Linux performance. The only question was: how well would L4Linux scale when multiple copies of Linux were run. The answer was: very well.

Xen's efforts to provide production-grade reliability and security have not yet succeeded. L4 has unquestionably demonstrated reliability, but only in situations where the applications are not hostile. L4 has *not* demonstrated practical success in security or fault isolation. This is the new push in the L4 community. It is why L4.sec (a new research project centered at Dresden) has adopted some fairly substantial architectural evolution in comparison to the L3 and L4 architectures.

1.2 Other Systems

The KeyKOS/EROS/Coyotos line of work (and now CapROS) have been *doing* what this article talks about since the late 1970's. KeyKOS was a significant commercial success. EROS was a retreat to a pure research phase. CapROS (which is based on the EROS code) is being developed commercially by Charlie Landau for medium robustness embedded applications. Coyotos (which is my own successor to EROS) will be shipping in 2007 into a number of mission critical applications around the world, and shortly thereafter into high-robustness medical applications.

Setting aside my own work, there is also the example of the AS/400. The OS/400 operating system isn't a microkernel system, but the OS/400 infrastructure *does* support component-structured applications (which is the *real* goal of microkernel designs), and the people who have built software on it swear by both the robustness and the maintainability of the resulting systems. Unfortunately, I'm not aware of any good comparative measurements supporting this assertion. If you know of any, please *tell* me!

1.3 The Facts

Ultimately, there are two compelling reasons to consider microkernels in high-robustness or high-security environments:

- There are *several* examples of microkernel-based systems that have succeeded in these applications *because* of the system structuring that microkernel-based designs demand.
- There are *zero* examples of high-robustness or high-security monolithic systems.

With that said, let me move on to Linus.

2 Linus's Latest Response

Linus makes some statements that are (mostly) true, but he draws the wrong conclusions.

> *... It's ludicrous how microkernel proponents claim that their system is "simpler" than a traditional kernel. It's not. It's much much more complicated, exactly because of the barriers that it has raised between data structures.*
>
> *The fundamental result of [address] space separation is that you can't share data structures. That means that you can't share locking, it means that you must copy any shared data, and that in turn means that* ***you have a much harder time handling coherency.***

The last sentence is obviously wrong: when you do not share data structures, there is no coherency problem by definition. Technically, it *is* possible to share memory in microkernel-based applications, but the statement is true in the sense that this practice is philosophically discouraged.

I don't think that experienced microkernel advocates have ever argued that a microkernel system is simpler overall. Certainly, no such argument has appeared in the literature. The components are easier to test and engineer, but Linus

makes a good point when he says *The fact that each individual piece is simple and secure does not make the aggregate ... simple* (he adds: ***or*** *secure*, which is wrong). I don't think that any of us would claim that large systems are simple, but this complexity is an intrinsic attribute of large systems. It has nothing to do with software construction.

What modern microkernel advocates claim is that properly component-structured systems are engineerable, which is an entirely different issue. There are many supporting examples for this assertion in hardware, in software, in mechanics, in construction, in transportation, and so forth. There are *no* supporting examples suggesting that unstructured systems are engineerable. In fact, the suggestion flies in the face of the entire history of engineering experience going back thousands of years. The triumph of 21st century software, if there is one, will be learning how to structure software in a way that lets us apply what we have learned about the *systems* engineering (primarily in the fields of aeronautics and telephony) during the 20th century.

Linus argues that certain kinds of systemic performance engineering are difficult to accomplish in component-structured systems. At the level of drivers this is true, and this has been an active topic of research in the microkernel community in recent years. At the level of applications, it is completely false. The success of things like GNOME and KDE rely utterly on the use of IDL-defined interfaces and separate component construction. Yes, these components share an address space when they are run, but this is an artifact of implementation. The important point here is that these applications scale *because* they are component structured.

Ultimately, Linus is missing the point. The alternative to structured systems is *unstructured* systems. The type of sharing that Linus advocates is the central source of reliability, engineering, and maintenance problems in software systems today. The goal is not to do sharing efficiently. The goal is to structure a system in such a way that sharing is minimized and carefully controlled. Shared-memory concurrency is *extremely* hard to manage. Consider that *thousands* of bugs have been found in the Linux kernel in this area alone. In fact, it is well known that this approach cannot be engineered for robustness, and shared memory concurrency is routinely excluded from robust system designs for this reason.

Yes, there are areas where shared memory interfaces are required for performance reasons. These are much fewer than Linus suppposes, but they are indeed hard to manage (see: *Vulnerabilities in Synchronous IPC Designs*). The reasons have to do with resource accountability, not with system structure.

When you look at the evidence in the field, Linus's statement "the whole argument that microkernels are somehow 'more secure' or 'more stable' is also total crap" is simply wrong. In fact, *every* example of stable or secure systems in the field today is microkernel-based. There are no demonstrated examples of highly secure or highly robust unstructured (monolithic) systems in the history of computing.

The essence of Linus's argument may be restated as "Microkernel-based systems make it very hard to successfully use a design approach that is known to be impossible to engineer robustly."

I agree completely.