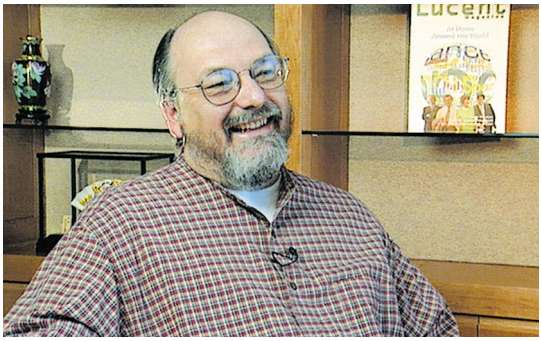
Unix and Beyond: An Interview with Ken Thompson

Computer recently visited Ken Thompson at Lucent's Bell Labs to learn about Thompson's early work on Unix and his more recent research in distributed computing.

Daniel Cooke
Texas Tech University

Joseph Urban
Arizona State University

Scott Hamilton
Computer



Ken Thompson needs no introduction: the co-creator of the Unix operating system as well as the Plan 9 and Inferno distributed operating systems; creator, along with Joseph Condon, of Belle, a world champion chess computer; 1998 US National Medal of Technology winner, along with Dennis Ritchie, for their role in developing the Unix system and C.

On the occasion of the presentation of the Computer Society's and Hitachi's inaugural Tsutomu Kanai Award for distributed computing, *Computer* visited recipient Ken Thompson at Lucent's Bell Labs. We were interested in learning about Thompson's early work on Unix and his more recent work in distributed computing. We were especially interested in learning about the creative process within Bell Labs and his sense of where computer science is heading.

CREATIVITY AND SOFTWARE DEVELOPMENT

Computer: Your nominators and endorsers for the Kanai Award consistently characterized your work as simple yet powerful. How do you discover such powerful abstractions?

Thompson: It is the way I think. I am a very bottom-up thinker. If you give me the right kind of Tinker Toys, I can imagine the building. I can sit there and see primitives and recognize their power to build structures a half mile high, if only I had just one more to make it functionally complete. I can see those kinds of things.

The converse is true, too, I think. I *can't*—from the building—imagine the Tinker Toys. When I see a top-down description of a system or language that has infinite libraries described by layers and layers, all I just see is a morass. I can't get a feel for it. I can't understand how the pieces fit; I can't understand something presented to me that's very complex. Maybe I do what I do because if I built anything more complicated, I couldn't understand it. I really must break it down into little pieces.

Computer: In your group you probably have both the bottom-up thinker and the top-down thinker. How do you interact with both?

Kanai Award

For more information on the Kanai Award, see "With Two New Awards, We Honor Unix, RISC Innovators," pp. 11-13.

0018-9162/99/$10.00 © 1999 IEEE

Thompson: I think there's room for both, but it makes for some interesting conversations, where two people think they are talking to each other but they're not. They just miss, like two ships in the night, except that they are using words, and the words mean different things to both sides. I don't know how to answer that, really. It takes both; it takes all kinds.

Occasionally—maybe once every five years—I will read a paper and I'll say, "Boy, this person just doesn't think like normal people. This person thinks at an orthogonal angle." When I see people like that, my impulse is to try to meet them, read their work, hire them. It's always good to take an orthogonal view of something. It develops ideas.

I think that computer science in its middle age has become incestuous: People are trained by people who think one way. As a result, these so-called orthogonal thinkers are becoming rarer and rarer. Of course, many of their ideas have become mainstream—like message passing, which I thought was something interesting when I first saw it. But occasionally you still see some very strange stuff.

Software development paradigms

Computer: What makes Plan 9 and the Inferno network operating system very striking is the consistent and aggressive use of a small number of abstractions. It seems clear that there's a coherent vision and team assembled here working on these projects. Could you give us further insight into how the process works?

Thompson: The aggressive use of a small number of abstractions is, I think, the direct result of a very small number of people who interact closely during the implementation. It's not a committee where everyone is trying to introduce their favorite thing. Essentially, if you have a technical argument or question, you have to sway two or three other people who are very savvy. They know what is going on, and you can't put anything over on them.

As for the process, it's hard to describe. It's chaotic, but somehow something comes out of it. There is a structure that comes out of it. I am a member of the Computing Sciences Research Center, which consists of a bunch of individuals—no teams, no leaders. It's the old Bell Labs model of research; these people just interact every day.

At different times you have nothing to do. You've stopped working for some reason—you finished a project or got tired of it—and you sit around and look for something to do. You latch on to somebody else, almost like water molecules interacting.

You get together and say, "I have an idea for a language," and somebody gets interested. Somebody else asks how we put networking in it. Well, so-and-so has a model for networking, and somebody else comes in. So you have these teams that rarely get above five or six, and usually hover around two or three. They each bring in whatever they did previously.

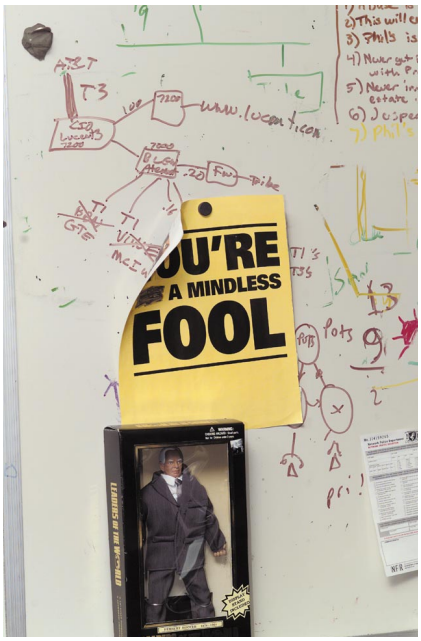
So that's the way it works. There are no projects per se in the Computing Sciences Research Center. There are projects near it of various sorts that will draw on our research as a resource. But they have to deal with our style. If people get stuck, they come to us but usually don't want to deal with the management style—which means none—that comes along with it.

Computer: You mentioned technical arguments and how you build your case. How are technical arguments resolved?

Thompson: When you know something is systemically wrong despite all the parts being correct, you say there has to be something better. You argue back and forth. You may sway or not sway, but mostly what you do is come up with an alternative. Try it. Many of the arguments end up that way.

You say, "I am right, the hell with you." And, of course the person who has been "to helled with" wants to prove his point, and so he goes off and does it. That's ultimately the way you prove a point. So that is the way most of the arguments are done—simply by trying them.

I don't think there are many people up in research who have strong ideas about things that they haven't really had experience with. They won't argue about the theory of something that's never been done. Instead, they'll say, "Let's try this." Also, there's not that much ego up there either, so if it's a failure you come back and say, "Do you have another idea? That one didn't work." I have certainly generated as many bad ideas as I have good ones.



Photos of the "Unix Room" at Bell Labs courtesy of Lucent Technologies.

Computer: What advice do you have for developers who are out there now to improve their designs so that they could be viewed as producing simple yet powerful systems?

Thompson: That is very hard; that is a very difficult question. There are very few people in my position who can really do a design and implement it. Most people are a smaller peg in a big organization where the design is done, or they do the design but can't implement it, or they don't understand the entire system. They are just part of a design team. There are very few people I could give advice to.

It's hard to give advice in a product kind of world

when what I do, I guess, is some form of computer Darwinism: Try it, and if it doesn't work throw it out and do it again. You just can't do that in a product-development environment.

Plus I am not sure there are real principles involved as opposed to serendipity: You happened to require this as a function before someone else saw the need for it. The way you happen upon what you think about is just very lucky. My advice to you is just be lucky. Go out there and buy low and sell high, and everything will be fine.

UNIX

Computer: In an earlier interview you were asked what you might do differently if you had to do Unix over again, and you said that you would add an "e" to the `creat` *system call. Seriously, in hindsight, can you give us an assessment of the problems you overcame, the elegant solutions, and the things you would have done differently.*

Thompson: I think the major good idea in Unix was its clean and simple interface: open, close, read, and write. This enabled the implementation of the shell as well as Unix's portability. In earlier systems, I/O had different entry points, but with Unix you could abstract them away: You open a file, and if the file happens to be a tape, you could write to it. Pipes allowed tools and filters that could accommodate classical monster programs like `sort`.

Probably the glaring error in Unix was that it underevaluated the concept of remoteness. The open-close-read-write interface should have been encapsulated together as something for remoteness; something that brought a group of interfaces together as a single thing—a remote file system as opposed to a local file system.

Unix lacked that concept; there was just one group of open-close-read-write interfaces. It was a glaring omission and was the reason that some of the awful things came into Unix like `ptrace` and some of the system calls. Every time I looked at later versions of Unix there were 15 new system calls, which tells you something's wrong. I just didn't see it at the time. This was fixed in a fairly nice way in Plan 9.

Computer: Going back a little bit further, what were the good and not so good aspects of Multics that were the major drivers in the Unix design rationale?

Thompson: The one thing I stole was the hierarchical file system because it was a really good idea—the difference being that Multics was a virtual memory system, and these "files" weren't files but naming conventions for segments. After you walk one of these hierarchical name spaces, which were tacked onto the side and weren't really part of the system, you touch it and it would be part of your address space and then you use machine instructions to store the data in that segment. I just plain lifted this.

By the same token, Multics was a virtual memory system with page faults, and it didn't differentiate between data and programs. You'd jump to a segment as it was faulted in, whether it was faulted in as data or instructions. There were no files to read or write—nothing you could remote—which I thought was a bad idea. This huge virtual memory space was the unifying concept behind Multics—and it had to be tried in an era when everyone was looking for the grand unification theory of programming—but I thought it was a big mistake.

I wanted to separate data from programs, because data and instructions are very different. When you're reading a file, you're almost always certain that the data will be read sequentially, and you're not surprised when you fault a and read $a + 1$. Moreover, it's much harder to excise instructions from caches than to excise data. So I added the `exec` system call that says "invoke this thing as a program," whereas in Multics you would fault in an instruction and jump to it.

Development history

Computer: What about the development history of Unix?

Thompson: The early versions were essentially me experimenting with some Multics concepts on a PDP-7 after that project disbanded, which is about as small a team as you can imagine. I then picked up a couple of users, Doug McIlroy and Dennis Ritchie, who were interested in languages. Their criticism, which was very expert and very harsh, led to a couple of rewrites in PDP-7 assembly.

At one point, I took BCPL from Martin Richards at MIT and converted it into what I thought was a fairly straight translation, but it turned out to be a different language so I called it B, and then Dennis took it and added types and called it C.

We bought a PDP-11—one of the very first—and I rewrote Unix in PDP-11 assembly and got it running. That was exported to several internal Bell telephone applications, to gather trouble reports and monitor various things like rerouted cables. Those applications, independent of what we were doing, started political pressure to get support for the operating system; they demanded service. So Bell Labs started the Unix Support Group, whose purpose was to serve as the interface to us, to take our modifications and interface them with the applications in the field, which demanded a more stable environment. They didn't like surprises.

This grew over time into the commercial version from AT&T and the more autonomous version from USL.

Independently, we went on and tried to rewrite Unix in this higher level language that was evolving simultaneously. It's hard to say who was pushing whom—whether Unix was pushing C or C was pushing Unix. These rewrites failed twice in the space of six months, I believe, because of problems with the language. There would be a major change in the language and we'd rewrite Unix.

The third rewrite—I took the OS proper, the kernel, and Dennis took the block I/O, the disk—was successful; it turned into version 5 in the labs and version 6 that got out to universities. Then there was a version 7 that was mostly a repartitioning of the system in preparation for Steve Johnson and Dennis Ritchie making the first port to an Interdata 832. Unknown to us, there was a similar port going on in Australia.

Around version 6, ARPA [Advanced Research Projects Agency] adopted it as the standard operating system for the Arpanet community. Berkeley was contracted to maintain and distribute the system. Their major contributions were to adapt the University of Illinois TCP/IP stack and to add virtual memory to Bell Lab's port to the VAX.

There's a nice history of Unix written by Dennis that's available on his home page [*ed.*—"The Evolution of the Unix Time-Sharing System," http://cm.belllabs.com/cm/cs/who/dmr/hist.html].

Computer: What accounted for the success of Unix, ultimately?

Thompson: I mostly view it as serendipitous. It was a massive change in the way people used computers, from mainframes to minis; we crossed a monetary threshold where computers became cheaper. People used them in smaller groups, and it was the beginning of the demise of the monster comp center, where the bureaucracy hidden behind the guise of a multimillion-dollar machine would dictate the way computing ran. People rejected the idea of accepting the OS from the manufacturer and these machines would never talk to anything but the manufacturer's machine.

I view the fact that we were caught up in that—where we were glommed onto as the only solution to maintaining open computing—as the main driving force for the revolution in the way computers were used at the time.

There were other, smaller things. Unix was a very small, understandable OS, so people could change it at their will. It would run itself—you could type "go" and in a few minutes it would recompile itself. You had total control over the whole system. So it was very beneficial to a lot of people, especially at universities, because it was very hard to teach computing from an IBM end-user point of view. Unix was small, and you could go through it line by line and understand exactly how it worked. That was the origin of the so-called Unix culture.

Computer: In a sense, Linux is following in this tradition. Any thoughts on this phenomenon?

Thompson: I view Linux as something that's not Microsoft—a backlash against Microsoft, no more and no less. I don't think it will be very successful in the long run. I've looked at the source, and there are pieces that are good and pieces that are not. A whole bunch of random people have contributed to this source, and the quality varies drastically.

My experience and some of my friends' experience is that Linux is quite unreliable. Microsoft is *really* unreliable but Linux is *worse*. In a non-PC environment, it just won't hold up. If you're using it on a single box, that's one thing. But if you want to use Linux in firewalls, gateways, embedded systems, and so on, it has a long way to go.

DISTRIBUTED COMPUTING: NETWORK OPERATING SYSTEMS AND LANGUAGES

Computer: How does your work on Plan 9 and Inferno derive from your earlier work on Unix? What are some of the new ideas arising out of this work that could and should apply to distributed operating systems in general?

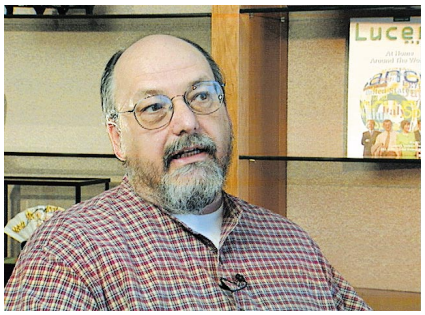
Thompson: Saying these ideas haven't been applied before is tough because, if you look closely, everything is reinvented, nothing's new. There are good ideas and bad ideas in Unix. You can't escape your history. What you think today is not much different from what you thought yesterday. And, by induction, it is not that different from what you thought twenty years ago.

In Plan 9 and Inferno, the key ideas are the protocol for communicating between components and the simplification and extension of particular concepts. In Plan 9, the key abstraction is the file system—anything you can read and write and select by names in a hierarchy—and the protocol exports that abstraction to remote channels to enable distribution. Inferno works similarly, but it has a layer of language interaction above it through the Limbo language interface—which is like Java, but cleaner, I think.

Limbo

Computer: How would you characterize Limbo as a language?

Thompson: First, I have to say that the language itself is almost exclusively the work of Sean Dorward, and in my talking about it I don't want to imply I had much to do with it.



I think it's a good language. In a pragmatic sense, it's a simplification of the larger languages like C++ and Java. The inheritance rules are much simpler, it's easier to use, and the restrictions there for simplicity don't seem to impair its functionality.

In C++ and Java I experience a certain amount of angst when you ask how to do this and they say, "Well, you do it like this or you could do it like that." There are obviously too many features if you can do something that many ways—and they are more or less equivalent. I think there are smaller concepts that fit better in Inferno.

Computer: We know that Plan 9 was done in C. It would almost seem that the group needed Limbo to develop Inferno. Do we need new types of languages to build distributed systems?

Thompson: The language, I think, doesn't matter per se. The language's actual size and features are almost separate issues from the distribution of the language. It shouldn't be too large or too small; it should be some nice language that you can live with. The idea, though, is that it is dynamically loadable so that you can replace little modules. And through some other mechanisms like encryption you can validate those modules, and when they are loaded you have some confidence that it's the module you wanted and that someone hasn't spoofed you.

There are certain features you must have—some form of object orientation, for example. You could replace Limbo with Java—I wouldn't want to—and not change Inferno's basic principles other than the way it meets system requirements. Sean decided the whole system had to have a garbage-collected language at a much higher level in that it's not separate interacting processes maintaining their own addresses, with some being garbage-collected and some not.

The language and the system are all garbage-collected together. Whatever protection mechanisms you have for the language apply all the way down through the system. For example, if you open a file, you don't have to close it. If you stop using it, just return from the function and it will be garbage-collected and the file will be closed. So the system and the language are part of the same ball of wax.

In addition, the language implementation—and again I don't want to take any credit—doesn't have big mark-and-sweep type garbage collection. It has reference counting: If you open something and then return, it's gone by reference count. Thus, you don't have high and low watermarks because 99 percent of the garbage goes away as soon as it is dereferenced. If you store a null in a pointer, it chases the pointer and all that stuff goes.

If you make cycles, there is a background distributed mark-and-sweep algorithm that just colors the next step a little bit at a time. It doesn't have to be very aggressive because there is not much garbage around in most applications: People really don't leave dangling loop structures that require this kind of algorithm. So you can devote just one percent of your time in the background looking for garbage without these monster mark-and-sweep things.

So, again, it's pragmatic. It's not the theoretical top-of-the-line garbage collection paper. It's just a way of doing it that seems to be very, very effective.

CURRENT WORK

Computer: What are you working on now?

Thompson: A few of us in research were tapped by a newly formed development organization within Lucent to work on a product called the PathStar Access Server. It's essentially a central office switch and router for IP phone and data services. It's strictly IP-based. You pick up the phone, dial it, and make conference calls.

I think packet switching will replace circuit switching in the phone system and will invert the hierarchy. Whereas data is currently carried in the leftover space of a circuit-switched network, eventually the backbone will be a packet-switched network with the phone implemented under it. You don't have to go out on a limb to say this—probably 90 percent of the people believe that now. But this project is "put up or shut up." We are actually inverting the phone system to run across a pretty classical packet-switched router.

In this kind of application what you need to pay attention to is maintenance and configuration, which is where Inferno comes in. All of the configuration code is Inferno and Limbo. You have to pay attention to quality of ser-

vice so that you can raise the loading above minimal and still get real-time voice, in this particular case.

There were some fun parts: The actual call processing, which is typically done by a huge finite state machine, was fun to do. We did it by making a finite-state-machine-generation language. The object of the language is a finite state machine, but the source is not. The actual phone conversation or feature is a group of interacting finite state machines, almost like processes. And, of course, they have to be distributed because you make calls to other phones.

Computer: So this language generates the finite state machines. Did you create the language to allow for experimentation to come up with different finite state machines?

Thompson: Well, at first we thought it was simple: You just write a finite state machine for this phone system. And at first it *was* simple. You just say, "Well if you're here do this, and if you're there do that, and just manually lay out these finite state machines." And that works just beautifully for the very first implementation, which is just picking up a phone, dialing a number, calling another phone, picking that phone up, conversing, and hanging up. You can just picture those states laying out.

But when you get to some of the simple features—three-way calls, for example—what happens when caller ID or call waiting comes in on a three-way call? The classical phone just says busy because it can't handle more than three phones.

So you build a model, which was initially a finite-state-machine model, and then you slowly add the features you need until the model breaks. It breaks pretty quickly, so you build a second one until it breaks, and so on. You just do it by exhaustion. So that's how the FSM-generation language came about; it wasn't "let's sit down and do everything at once." I think that's probably the way computer languages were built.

Interestingly, this work was extended further by Gerard Holzmann, someone in our area who has been into state verification—running exhaustive studies finding error states in the finite state system. He was just delighted with this little FSM-generation language because now he could build his models, and he inverted it. He took it the way it is, which is to build finite state machines, but he also took it to build drivers. So he has my model, which runs the phones on the inside, but then he needed telephones to drive his model. So he can now build another finite state machine to model the telephone and do not only the synthesis but the analysis.

Jukebox music collection

Computer: You're also collecting music?

Thompson: It's kind of a personal/research hobby/project. Let me explain it from an external point of view. Basically, I'm just collecting music. I'm getting lists from various sources—top 10s, top 50s—and I try to collect the music.

Right now, my list has around 35,000 songs, of which I've collected around 20,000. I compress the songs with a Bell-Labs-invented algorithm called PAC [Perceptual Audio Coding] and store them on a jukebox storage system. I started this before MP3 was heard of on the network. PAC is vastly superior to MP3.

My collection is not generally available because of the legal aspects. I went to legal and told them I was collecting a lot of music, but I don't think they realized what I meant by "a lot." Anyway, they said that in the case of research there's something similar to fair use and that they'd back me, but wouldn't go to jail for me. So I can't release it generally. But it's pretty impressive. It's split-screen like a Web browser; you can walk down lists, years, or weeks.

Computer. It's a personal hobby.

Thompson: It's hard to differentiate since, if you haven't noticed, almost everything I've done is personal interest. Almost everything I've done has been supported and I'm allowed to do it, but it's always been on the edge of what's acceptable for computer science at the time. Even Unix was right on the edge of what was acceptable at Bell Labs at the time. That's almost been my history.

COMPUTER SCIENCE AND THE FUTURE

Computer: You've been there through Multics, Unix, Inferno, and so on. Any thoughts about where computer science is going or should be going?

Thompson: Well, I had to give advice to my son, and my advice to him—to the next generation—was to get into biology.

Computer science is coming into its middle age. It's turning into a commodity. People don't know about Carnot cycles for refrigerators, yet they buy refrigerators. It's happening in computing, too. Who knows about compilers? They buy computers to play games and balance their checkbooks. So my advice to my child was—I am unfortunately talking to *Computer* magazine—to go into biology, not classic biology but gene therapy and things like that.

I think that computing is a finite field and it's reaching its apex and we will be on a wane after this. I am

sorry to say that, but that's the way I feel. You look at any aspect of computer science—what's being taught today, PhD theses, publications, any metric you can think of and compare it to history—and you realize that aspects of computer science are becoming more specialized.

Computer: Which aspects?

Thompson: Operating systems, in particular, have to carry so much baggage. Today, if you're going to do something that will have any impact, you have to compete with Microsoft, and to do that you have to carry the weight of all the browsers, Word, Office, and everything else. Even if you write a better operating system, nobody who actually uses computers today knows what an operating system interface is; their interface is the browser or Office.

You can have the best and most beautiful interface in the world and the most extensible operating system that ports to anything, and then you have to port on top of it a thousand staff-years' worth of applications that you can't obtain the source for. You have two choices: Go to Microsoft and ask for the source to Office to port to your operating system and they'll laugh at you; or get a user's manual and reengineer the code and they'll sue you anyway. Basically, it'll never happen because the entry fee is too high.

Anything new will have to come along with the type of revolution that came along with Unix. Nothing was going to topple IBM until something came along that made them irrelevant. I'm sure they have the mainframe market locked up, but that's just irrelevant. And the same thing with Microsoft: Until something comes along that makes them irrelevant, the entry fee is too difficult and they won't be displaced.

Computer: So you're not precluding the possibility of a paradigm shift.

Thompson: Absolutely not. Anybody who says there's no more innovation in the world is doomed to be among the last 400 people who have stated this since the birth of Christ.

Computer: You're still having fun?

Thompson: Yes, there are still a lot of fun programs to write.

WHAT I DID ON MY WINTER VACATION

Computer: We can't let you go without asking why on earth you traveled to Russia to fly a Mig-29?

Thompson: How often does the Soviet Union collapse? It would be just a shame if you couldn't do something you have always wanted to do as a result. They are selling rides in what was once the top fighter. A mere two years earlier you would only get hints about its existence in Jane's books. Now you can get in, use the laser sights, and go straight up at 600 miles per hour. Who wouldn't do that? When things like that come along, I'll take them. They're fun. ❖

Acknowledgments

We thank Patrick Regan of Lucent Technologies for arranging this interview and Brigitta Hanggi for providing the photos. Photos of the "Unix Room" in the Computing Sciences Research Center are courtesy of Lucent Technologies.

Daniel Cooke is chair of the Computer Science Department at Texas Tech University.

Joseph Urban is a professor in the Computer Science and Engineering Department at Arizona State University.

Scott Hamilton is senior acquisitions editor for Computer.

Contact the authors at {d.cooke, j.urban, s.hamilton}@computer.org.