FOSDEM: The circuit less travelled

Intro

Thank you very much for coming to what might be FOSDEM's most wildly speculative talk!

So – hello! My name is Liam Proven. As of this year, I've been working in IT for 30 years. For most of the first decade, in support and as a sysadmin. Since then, as a tech journalist.

However, this is only the 2nd time I've presented a talk at a tech conference. The last time was in 1996. So forgive me if I'm a little out of practice.

Big Lies

Myths and traditions

In a way, this talk is about myths, or at least traditions, which are part of the culture of computing. There are more of these than you probably suspect. I call them the Big Lies. They are nearly universal. Almost everyone accepts them, but they're not even slightly true.

But before I introduce the first of these "big lies", I think I should tell you how I stumbled across it.

How I got here

For many years, I was a freelance technology journalist. This lifestyle has challenges, such as finding stuff to write about.

For example, around 1996 to 1998, I wrote about an experimental new operating system that nobody much in the UK was covering: Linux. At the turn of the century, I covered another new, experimental technology: virtualisation. In 2010, I said containers were going to be the next big thing.

But it's always the same. After a few years, everyone is talking about this stuff.

Retrocomputing

About twelve years ago, I hit a short dry patch. For weeks, I couldn't find anything interesting to write about that someone else hadn't covered first.

I got desperate. So I wrote about a very niche news story – a new release of the Commodore Amiga operating system.

My editor was not impressed. “Nobody **cares** any more, Liam,” he said. “It’s history. It’s over. It’s dead. Let it go!”

But I didn’t have anything else, so I pressed him, and he published it.

Two days and about 250 thousand page views later, he came back to me. “Give me more like this,” he said.

So, since then, alongside all the stuff about current tech, I’ve occasionally done retrocomputing stories. But the same thing happened – it got popular and lots of journos started writing about it.

That meant I had to go digging, looking for something more obscure. Some old, forgotten computer which was important in its day but that nobody talked about much any more.

I found a bigger story than an old computer.

Don’t mention the war

I found a whole forgotten war, one that petered out in the late 1980s.

You might know the Winston Churchill quotation: “history is written by the victors”. It means that the people on the losing side of a war don’t get to tell their stories, but more than that, it often means that their culture goes away. The winners’ culture, language, music, fashion, food and drink, all end up taking over.

That happened in computing, not back when dinosaurs stalked the data centre, but around the time of the early Internet and the first personal computers with graphical user interfaces. In other words, relatively recently.

The Biggest Lie

That leads me to the first of the Big Lies. The biggest, in fact. It’s a simple one. You probably believe it. It’s this: computers today are better than they have ever been before. Not just that they have thousands of times more storage and more speed, but that everything, the whole stack – hardware, operating systems, networking, programming languages and libraries and apps – are better than ever.

The myth goes like this: early computers were simple, and they got replaced by better ones that could do more. Gradually, they evolved, getting more sophisticated and more capable, until now, we have multi-processor multi-gigabyte supercomputers in our pockets.

Which gives me an excuse to use my favourite German quote. It's from Max Planck. Please forgive my terrible pronunciation. "Das is nicht nur nicht richtig, das is nicht einmal falsch!" "That is not only not right, it is not even wrong!"

Someone asked John Glenn, America's first man in space, what it felt like just before launch. Allegedly, he replied "I felt about as good as anybody would, sitting in a capsule above a rocket that were both built by the lowest bidder."

Well, **that** is where we are today.

The story about evolution is totally wrong. What really happened is that, time after time, each generation of computers developed until it was very capable and sophisticated, and then it was totally replaced by a new generation of relatively simple, stupid ones. Then those were improved, usually completely ignoring all the lessons learned in the previous generation, until the new generation get replaced in turn, by something smaller, cheaper and far more stupid.

Evolution

The first computers, of course, were huge room-sized things that cost millions. They evolved into mainframes: very big, very expensive, but a whole company could share one and use it at the same time, using punched cards and stuff like that.

Those were replaced by minicomputers: the size of a filing cabinet, but cheap enough for a department to afford and to share, using interactive terminals.

The mainframes' intelligent peripherals and sophisticated hypervisor-based operating systems with rich role-based security? Thrown away. Forgotten.

Next, minicomputers were replaced by workstations: at first, desk-sized, later desk-side and eventually PC sized. Very powerful, very expensive, and all for just one user. (Usually someone senior and important, of course.)

The minicomputers' rich filesystems, with built-in version tracking? Their seamless clustering? Their rich groupware enabling teams to cooperate and work on shared documents? Forgotten. You got email.

And those were replaced by microcomputers. Micros. At first, they were feeble. They could hardly do anything. With these, we lost tonnes of stuff.

Hard disks? Too expensive. Multitasking? Not enough memory. Choice of programming languages? Kids don't need that. Shove BASIC in the ROM, that will do. They mostly got used for playing games, anyway.

The first ones could handle floppy disk drives and expansion cards. Then those got taken out – too expensive. You got a cassette recorder.

Those, the weakest, the most pathetic computers since the first ones in the 1940s… The 8-bit micros…

Those are the ancestors of the ones you use today.

In fact, of all the early 1980s computers, the most boring, most complicated design, the one with **no** graphics and **no** sound – **that's** the ancestor of what you use today. With one exception, which I'll get to.

They got expanded and enhanced, over and over again.

And it’s a sort of law of nature that when you try to put stuff back in that you left out, it’s never as good as if you designed it in at the beginning.

We run the much-upgraded descendants of the cheapest, simplest, stupidest computers that anyone could get to market. They were the easiest to build and to get working. Easy means cheap, and cheap sells more and makes more profit. So they are the ones that won.

There’s a proverb about choosing a product.

You can have good, fast, and cheap. Which two do you want?

We got fast and cheap. We lost out on good. I want to tell you a little about what we lost.

Why retrocomputing?

As I said, I stumbled across the traces of a war, while looking for something interesting to write about *retro-computing*.

This is a big and growing hobby now. Lots of mostly middle-aged people enjoy restoring and playing with the computers they used when they were young.

I’m going to ignore all the 8-bit stuff. They were so limited, they didn’t pass much on to later generations.

Except perhaps the BBC Micro, which influenced the ARM processor. Me, I couldn't afford one. I had Sinclair ZX Spectrums. I've still got one of them, and now it has a 4GB micro-SD drive attached, for instance, which I find hilarious.

In 1983 came the Apple Lisa, at a cool $10,000.

In 1984, the Sinclair QL, at £399, and a couple of months after that, the Apple Macintosh. It was a quarter of the price of the Lisa: $2,500. That's how fast progress was moving in those days!

In 1985, the Amiga and the Atari ST. In 1987, the Acorn Archimedes, with the new ARM processor.

They're all dead and gone now, of course. The x86 PC caught up and then passed them.

But people miss them.

Amiga fans are some of the most enthusiastic. You can sort of see why. They had the best graphics and sound, a full multitasking OS, with a GUI, running off floppy disks on a machine with 512 **kilobytes** of RAM. Half a meg!

So today, there are FOSS versions of the Amiga OS, Sinclair QDOS and Atari TOS. There's a shared-source version of Acorn RISC OS. It runs natively on the Raspberry Pi. I like it a lot, because I had an Archimedes.

And horribly, horribly limited today.

People play with them, but only a few old-timers actually use these systems today. They are all horribly compromised and limited. It's amazing what they could do on very limited hardware, but there are always costs for this.

Either they didn't multitask, or they did but it was cooperative, or it was pre-emptive but there was no memory protection. There was no networking.

They're fun to play with, but they were written in haste, by small teams, under fierce time and budget constraints. They are not remotely competitive with anything we have today.

The obvious answer:

Old computers were more fun because they were simple enough that you could understand them.

A more truthful answer:

Modern computers suck!

How dare you!

So I talked about evolution and how we got to the micro.

Now let's look at how we got from micros to today.

The first generation:

8-bits

Maxed out at 64 kB of RAM

No mass storage, loaded and saved to cassette

Most ran BASIC from ROM

The second generation, 5 years later.

16-bit

A few megabytes of RAM

Hard disks, subdirectories

The first GUIs

Colourful graphics, sound that was actually pleasant to listen to.

The third generation, the beginning of the 1990s.

Now we're getting to only PCs, Macs and Unix boxes.

32-bit chips

Partially 32-bit operating systems

Multitasking as standard, but not very good

Networking as standard, but lots of protocols

Very poor interoperability. Basically they could mostly read each other's floppy disks and that was it.

The 4th generation – mid 1990s

32-bit OSes:

Windows NT

Linux starts getting serious

But the grown-up OSes don't do games and media very well

Everything talks TCP/IP…

… Mostly over modems

The web begins to happen

The 5th generation – turn of the century

Everyone's running a real 32-bit OS now

USB everywhere

They do games and media too.

Built in web support

A few hard core types have multiple CPUs and multiple screens

The 6th generation – ten years on

64-bit OSes

Multiple cores are standard

Errrrr

That's it.

Look at the pace of change.

In the early years, every 5 years there was something new and radical that blew away the older generation.

Now?

Hint, I'm using a 7 year old machine to do this presentation…

So people are looking back to when computers were new and did cool new stuff, **because they don't any more.**

Progress has stalled. Now we're doing niche stuff, mostly on servers, trying to solve problems of scalability and reliability, because we haven't made any big steps in nearly 20 years.

The story so far

In the 1980s, there was tons of diversity.

Now there are only 2 OSes left.

One is FOSS re-implementation of a 1960s OS, because it was the cheap and relatively easy thing to do.

The other is a commercial successor to a 1970s OS, DEC VMS.

One leverages two things: cheap, ubiquitous PC hardware, and the accumulated cultural knowledge – and code – of nearly 50 years of Unix.

The other leverages the long dominance of the IBM PC, its vast software catalogue and its entrenched dominance of business computing.

Unix and Windows NT. They won.

Mere tricks like cross-platform binaries and network transparency weren't enough. It will need a bigger disruption than that to give something genuinely **different** a real chance.

When the going gets weird, the weird turn pro

So what might provide that disruption?

Two things. One mainly a hardware thing, one mainly a software thing.

Xerox Alto

I went looking at the origins of the GUI. That's a fairly famous machine: you've probably heard of the Xerox Alto. A $30,000 deskside workstation, it was the machine that Steve Jobs and some senior Apple people saw which inspired the Lisa and the Mac.

As Steve Jobs said, he saw 3 amazing technologies that day, but he was so dazzled by one of them that he missed the other two. He was so impressed by the GUI that he missed the object-oriented graphical programming language, Smalltalk, or the built-in ubiquitous networking – Altos all talked to a fileserver and users could email stuff to one another.

The Lisa had none of that. The Mac had less. Apple spent much of the next twenty years trying to put them back.

And that's what put me on the trail.

You look into Smalltalk, and you find people saying it's one of the two languages that changes how you think about programming.

That's interesting, right?

But even more interesting is that when I went looking for the other one, I found Apple had dabbled in it, too.

It was nearly in Apple's **other** computer.

Steve Jobs hired John Sculley from PepsiCo to run Apple, and in return, Sculley fired Jobs. What Apple came up with during Sculley's reign was the the Newton.

I have two of them. I love them. They're a vision of a different future. But I didn't actually use them, I used Psions.

The Newton that shipped, though, was a pale shadow of the Newton that Apple originally planned. There are traces of that machine out there, though, and that's what led to me uncovering the great computing war.

The Newton is a radical machine. It's designed to live in your pocket, store and track your information and habits. It had an address book, a diary, a note-taking app, astonishing handwriting recognition, and a primitive AI assistant. You could write "lunch with Alice" on the screen, and it would work out what you wrote, analyse it, work out from your diary when you normally had lunch, from your call history where you took lunch most often and which Alice you contacted most often, book a time slot in your diary and send her a message to ask her if she'd like to come.

Siri, but twenty years earlier. By the way, Apple seems to have forgotten all this. It had to buy Siri in.

NewtonOS also had no filesystem. I don't mean it wasn't visible to the user; I mean there wasn't one. It had some non-volatile memory on board, expandable via memory cards – huge PCMCIA ones the size of credit cards, half a centimetre thick – and it kept stuff in a sort of OS-integrated object database, segregated by function. The stores were called "soups" and the OS kept track of what was stored where. No filenames, no directories, nothing.

Apps were written in a language called NewtonScript, which is very distantly related to both AppleScript on modern macOS and to JavaScript.

But that was not the original plan. That was for a far more radical language, one that could be developed in an astounding graphical environment. .

The language was called Dylan, from “dynamic language”. It still exists as a FOSS compiler. Apple seems to have forgotten about it, too, because it reinvented that wheel with Swift.

Have a look. It’s pretty cool. It’s very readable, it’s very high level, and before commercial realities got to them – time and money, mainly – Apple planned to write an OS in it and the apps for that OS.

How come? The same language for the OS and the apps?

Yes, because Dylan is a sort of wrapper around one of the oldest programming languages that’s still in active use: Lisp.

And both Smalltalk and Lisp is very much still around. For both, there both are commercial and FOSS versions. They run on .NET CLR and the JVM. There’s a Smalltalk that runs in your browser on the Javascript engine.

But these are only the traces left behind after the war.

Because once, both were not just languages that ran on commodity OSes.

There **were** OSes in their own right.

I started digging into that, and that’s when the ground crumbled away and I found I was over a cave with a whole ruined city, like some very impressive CGI special effects.

Once, Lisp ran on the bare metal, on purpose-built computers which ran operating systems written in Lisp, ones with GUIs that could connect to the Internet.

And if you find the people who used Lisp Machines… wow. They **loved** them, with a passion that makes Amiga owners look like, well, amateur hobbyists.

Some of the references are easy to find. There's a wonderful book called the "Unix Hater's Handbook", which I highly recommend. It's easy to read and it's really funny. It's a digest of a long-running Internet community from the time when universities were getting rid of Lisp Machines and replacing them with cheaper, faster Unix workstations – Suns and things like that.

Lisp Machine users were not impressed. For them, Unix was a huge step backwards. The code was all compiled, whereas Lisp OSes ran a sort of giant interpreter. On a Unix machine, if you didn't like the way a program did something, you had to get the source code, edit it, save it, compile it, replace your existing binary with the new one, restart the program – and probably find that you'd made a mistake and it didn't work, and try again.

On the Lisp Machines, your code wasn't trapped inside frozen blocks. You could just edit the live running code and the changes would take effect immediately. You could inspect or even change the values of variables, as the code ran.

You didn't even boot them up often. At the end of the day, it wrote the values of all its objects and variables to disk – called saving a "world" -- and stopped. When you turned it back on, it reread these values and resumed exactly where it was.

And most of this applies to Smalltalk machines too.

This is what Steve Jobs missed. He was distracted by the shiny. He brought the world the GUI, but he got his team to re-implement it on top of fairly conventional OSes, originally in a mixture of assembly and Pascal.

He left the amazing rich development environment, where it was objects all the way down, behind.

And we never got it back.

We got networking back, sure, but not this.

Now, they're just niche languages, but both Smalltalk and Lisp were once whole other universes. Full-stack systems.

The difference, though, is where the biggest losses in the war came.

Smalltalk machines ran on relatively normal processors.

Smalltalk is all about objects, and you can't really handle objects at hardware level.

Well, you can – a very expensive hifi manufacturer called Linn tried with a machine called the Rekursiv, but it flopped badly. So did Intel's attempt to do a chip that implemented high-level stuff in hardware – not the Itanium, no, long before that, the iAPX 432.

But Lisp Machines did run on dedicated chips, and this is where stuff gets real.

You know what I mean by “stuff,” right? The stuff that hits the fan.

As an indication of how significant the biggest Lisp Machine maker was, the first ever dot-com domain on the Internet was symbolics.com.

As the hipsters say, you’ve probably never heard of Symbolics.

The company’s dead, but the OS, OpenGenera, is still out there and you can run it on an emulator on Linux. It’s the end result of several decades of totally separate evolution from the whole Mac/Windows/Unix world, so it’s kind of arcane, but it’s out there.

There are a lot of accounts of the power and the productivity possible in Lisp and on Lisp Machines.

One of the more persuasive is from a man called Kalman Reti, who is the last working Symbolics engineer. So loved are these machines that people are still working on their thirty-year-old hardware, and Reti maintains them. He’s made some Youtube videos demonstrating OpenGenera on Linux.

He talks about the process of implementing the single-chip Lisp Machine processors.

"We took about 10 to 12 man years to do the Ivory chip, and the only comparable chip that was contemporaneous to that was the MicroVAX chip over at DEC. I knew some people that worked on that and their estimates were that it was 70 to 80 man-years to do the microVAX. That in a nutshell was the reason that the Lisp Machine was great."

Now that is significant.

When different people tell you that they can achieve such a huge differential in productivity – one tenth of the people taking one tenth of the time to do the same job – you have to pay attention.

Weaknesses of LispMs

But I am not here to tell you that Lisp Machines were some ultimate super machine.

An environment that is mostly semi-interpreted code, running in a single shared memory space, is not very stable

And when it crashes, if you don't have a snapshot to go back to, they were slow to cold boot.

But why did people love them?

Smalltalk machines had a lot in common with Lisp Machines

1 language (mostly) all the way down

Data centric, not file centric

Everything's accessible – live, dynamic environment, not static compiled blobs

Standard operating procedure is to "save" the "world" – write all the variables' state to disk – so when you turn it back on, it picks up exactly where you were.

Differences

Smalltalk is objects everywhere. Lisp is lists everywhere.

This means in Lisp…

Code is data

Code can manipulate itself. Lisp macros are vastly more powerful than in any other language.

Generality: the same language could be used for an OS, for apps, and for scripting. There is no other language that can claim this.

And the thing is, even 30 years ago, they could implement list processing and management in hardware, and it was relatively fast and efficient.

Late-generation Lisp machines ran Fortran and C as well, and benchmarks showed that garbage-collected dynamic memory management ran faster than C's deterministic manual memory management.

So what?

I'm not holding either Smalltalk workstations or Lisp machines up as examples of great OS design.

The **real** point here is that there are real benefits to an OS where as much as possible of the whole stack is in a single language, which may be running as some kind of just-in-time compiled bytecode, but where objects and data are visible down the whole stack.

This shows up another of the big lies that everyone just takes as read. This one has layers like a cabbage.

For speed, you need a language that's close to the metal

This has costs but they're worth it

Be careful, be very very careful

But as the stack matures, there are more layers.

the higher layers are further from the metal

they can do other stuff

so you can have higher-level languages

but you need to recode in a lower level one for speed

HLLs good for prototyping, good for non-speed critical code

But pros do it in C++ or something for speed

So what if C++ is huge? You don't need all of it!

So what if it's hard to read? It was hard to write!

That there is a necessary contrast between readable and fast

This is one of the big assumptions behind both the Unix and Windows schools of OS design. That different languages are best for different jobs, and so you need a mix. That means that the layers are sealed off from one another, because of different models: of variable storage, of memory management, etc.

And two families of now-extinct computers show that this just wasn't so.

Pick the right language and you can do it all in one.

Smalltalk isn't that language, because it doesn't map well onto hardware. Lisp does. But you don't feel it if it is just sitting on top of whole sealed-off layer cake of compiled code in other languages.

Lisp, it must be said, is really hard to read.

But Dylan shows that it doesn't have to be.

If you lose the list-based notation, yes, there is a price in efficiency and power, but the result is readable by mere mortals.

Dylan is not the only try. There are quite a few – PLOT, CGOL, sweet expressions, and more. By all means go explore. I'll post links.

There is real potential here. But it has a high cost. We pretty much would have to throw everything away and start again.

I think we can do better. Incorporate all we've learned from decades of Unix and its kin, **and** from Smalltalk **and** from Lisp and Dylan and PLOT and so on. Make it powerful and yet readable and accessible.

Oddly, there is one example out there… but it is perhaps tainted.

It is called Urbit. It is comprises a virtual machine, a low-level list-oriented language called Nock, and a high-level functional language called Hoon. It exists for the purpose of secure communications and cryptocurrency trading across an untrusted medium – the internet – on untrusted machines: Unix and Windows.

It's the work of a smart but very odd man called Curtis Yarvin, who blogs under the name "Mencius Moldbug". He is a thought leader of the neo-conservative movement and has among others the stated goal of bringing down the US government and replacing it with a monarchy. He has also speculated in his blogging about the relative intelligence of different human races – and you can probably guess how that went.

Perhaps the most remarkable thing about it other than that is exists and works is that Yarvin claims to have deduced its principles from scratch with no prior knowledge of Lisp Machines.

I am not sure I would entirely trust it, and I am entirely sure that I would not trust Mr Yarvin. However, as a piece of programming and nothing more, I admire it hugely.

And it shows that what I am discussing can be done.

There have been more limited prior efforts: Movitz is a Lisp OS for x86-32, for instance. A newer Lisp-based hardware-independent OS is CrysaLisp, by Chris Hinsley, original author of Taos.

This stuff is not impossible.

Why would we do that? The cost, the manpower, the time it will need – all are significant.

Ah, well, the reason why is the next and last bit.

The next big shift

Things like the Newton tried to show another way to do computing.

Personal devices that live in your pocket, with no keyboard, that learn your habits and help you.

But they were a step too far and the tech didn't work.

We have them now… or something like them.

Naturally, when Steve Jobs came back to Apple, one of the first things he did was kill "that scribble thing" – Sculley's baby, the Newton. He didn't invent it, so he didn't like it.

But later, to win a bet with Bill Gates, he came up with a tablet people would actually like to use.

It was too big, though, so first, he shrank it to the size of an iPod and combined it with a smartphone.

That sold great, so then the tablet-sized one followed.

But under the hood, the iPhone and iPad are Unix machines. They're traditional computers, with a friendly face, and yes, a software keyboard.

But take a step back and look at the design.

They are diskless computers. They have some RAM and some Flash and that's it.

The Flash emulates a disk drive, but there's no SATA connector here. These are computers that can only handle onboard Flash, nothing else.

But there's another transition coming.

There are faster forms of non-volatile memory coming. Several of them.

HP is working on memristors, a new kind of electronic component that keeps its contents when it's turned off.

Intel is already selling 3D X-Point, which is more like flash but hundreds of times faster.

You can buy Flash DIMMs for servers today: putting the non-volatile memory right on the CPU's memory bus.

The writing is on the wall.

Soon, the first new kind of computer since the early micros will appear.

It will have only one kind of memory. A processor, plus some RAM, but RAM that keeps its contents when it’s turned off.

No booting. You turn it on, it continues exactly where it left off. No shutdown or suspend mode, either.

You never install an OS. It boots off a network connection at the factory, and that’s it, it’s in RAM, running. Forever, updates and freezes allowing.

If it gets corrupted, well, reboot over the Internet, just like a Mac does today.

No need for disks, emulated or not. It’s all just RAM. There’s nothing to load from or to, nothing to save to. It’s all one flat space.

Sure, you could partition off a bit, make a pretend filesystem in it. And some companies will, because you can’t have Windows or Unix without a filesystem. The idea of files, of types, of permissions, of executables and data files, are absolutely inherent in all such operating systems.

So that’s how they will be implemented, at first. Pseudo disks: real filesystems, on fake disk drives. Remember RAM disks? They’ll come back.

But this is a very inefficient way to use such an amazing resource.

This is a new way to build a computer, for the first time since the first disk drives were invented in the 1950s.

So this is a huge opportunity, and a huge threat to the established order of operating systems.

Note, no mainframe manufacturers were terribly successful moving into minicomputers, nor into workstations. OK, some – well, one -- minicomputer maker did OK in workstations, but mainly by making single-user minis.

None of them thrived in micros.

Except IBM, and you may note that IBM no longer makes PCs or workstations – just mainframes and Unix servers now. Everything else got sold off or discontinued.

But there's one product. One I've seen twice and used once in a 30 year career.

AS/400. IBM's minicomputer, launched after the mini era was over.

It's dead now, but its OS lives on as an alternative OS for POWER servers. It used to be OS/400 but now it's called IBM i.

It's a very unusual OS. It doesn't really have a filesystem, as such. All of its storage is treated as one flat space. The OS manages which blocks are on disk and which are in RAM, but it's invisible to the programmer.

This is called Single Level Store and IBM i is the last surviving system to use this architecture.

But like hypervisors, like containers, like multitasking and networking and all the other things the micro has had to re-invent from scratch… it's coming back.

This is a massive transition, and it will really shake things up.

It's a huge chance for the IT industry. A chance to get rid of 30+ years of cruft and clutter.

And there are some historical examples to learn from.

Because this model is a very good fit for operating systems like those of Lisp Machines and Smalltalk boxes. OSes which juggled objects in memory, rather than the contents of files on disk.

And it's a big enough transition that it might mean a shift away from x86 and towards more power-efficient, simpler architectures. And if we shift architectures, well, then maybe it's time to re-examine the idea of CPUs that support a higher level of abstraction than the bytes and words that C demands. CPUs that understand lists as the basic unit of storage.

It's a big shift. But the rewards could be more than big enough…