news@UK

The Newsletter of UKUUG, the UK's Unix and Open Systems Users Group
Published electronically at `http://www.ukuug.org/newsletter/`

Volume 17, Number 4 **ISSN 0965-9412** **December 2008**

Contents

News from the Secretariat

The Annual General Meeting was held on 25th September. The minutes and associated documents are available online (we will send out printed copies on request):
`http://www.ukuug.org/events/agm2008/`

You will note from the minutes we have two new Council members, Niall Mansfield and Holger Kraus. Niall (Cambridge) and Holger (Leicester) are enthusiastic about UKUUG and we are hoping they can bring new ideas and assist with work on ongoing items.

Sam Smith and Alain Williams, although no longer Council members, have taken on the following non-Council roles: 'Events & Web site' and 'System Administrator' respectively.

The first meeting of the new Council will be held on 11th December here in Buntingford. This is the first time a UKUUG Council meeting has actually been held here at the UKUUG office. I trust they will all enjoy their visit to this 'historic market town'.

Paul Waring has taken the role of UKUUG Chairman and Howard Thomson is the new Treasurer.

Issues for Council at the meeting will include the organisation and programme for the Spring conference being held in London from 24th – 26th March 2009. Full event details (information booklet and booking form) will be sent to all members in early January but we hope to have a flyer with preliminary details in with this Newsletter. I am sure this event is going to prove very popular and we are currently looking for sponsors to help with general event expenses and the conference dinner.

The recent Linux event held on 7th – 9th November in Manchester was a success in terms of its programme and was much enjoyed by most of those who were present, but for some reason attendee numbers were low. I have spoken to people who did attend and they have said it was a good event. I wonder if the current financial problems or the fact the event was held in November rather than in the summer are to blame for the lack of attendees. If you usually attend the Linux event but didn't this year perhaps you could send me a quick email just to advise why this year you didn't attend.

At the time of writing this piece we are just one week from the 'Open BSD's PF' tutorial (provided by Peter N. M. Hansteen) – Wednesday 26th November. For this tutorial we have 11 delegates which is a good number for a tutorial.

UKUUG is planning more tutorials in the new year – if you have any particular topics that you think would be of interest please let us know.

The annual membership subscription invoices will be sent out in January, please look out for your invoice and as always prompt payment will be gratefully received!

I would like to take this opportunity to wish you all a very Happy Christmas and a Peaceful New Year.

The Secretariat will close from 19th December to January 5th 2009.

Please note the copy date for the next issue of the Newsletter (March 2009) is 20th February 2009. We are always looking for interesting submissions from members, so please submit any articles or other items direct to
`newsletter@ukuug.org`

Report by UKUUG Chairman

Paul Waring

Report on legal action with the British Standards Institute

Back in April, UKUUG sought a Judicial Review of BSI's decision to vote "yes" to the fast tracking of the DIS29500 (OOXML) proposed standard in ISO. The original application was rejected, but UKUUG appealed as we felt that the judge had not understood the arguments presented and wished to present our case at an oral hearing. Unfortunately, by the time this date came round it was clear that any decision by the court would not affect the fast-tracking of the standard, and permission was granted to discontinue the case. At the time of writing, we are still awaiting judgement on costs and will let members know as soon as there is an update on the progression of the case.

Linux 2008

Our annual Linux conference was held in Manchester this year. As always, the programme was full of interesting talks, ranging from porting Linux to different architectures to designing a "desktop for Dad", and even one talk which was delivered remotely via a video link. We also enjoyed two evening social events, managing to take over the entire top floor of an Indian restaurant on the famous "Curry Mile" on Friday and partaking in Chinese cuisine on Saturday for the official conference dinner. Overall it was a successful and informative event which will hopefully continue in future years.

Changes to Council

Since the last newsletter we have held our Annual General Meeting, which resulted in some changes to UKUUG's Council. Both Sam Smith and Alain Williams stepped down from Council, having served two terms of three years. Our thanks go to both Alain and Sam for the hard work which they have put into UKUUG, and we are grateful for their continued support in helping to run events, the UKUUG website and the mailing lists which we use for much of our communications. We also have two new members on Council, Niall Mansfield and Holger Kraus, who are already getting their teeth stuck into UKUUG activities.

As always, UKUUG exists to serve its members, so if you have any ideas for future events or would like to get involved with any aspect of UKUUG, please do let us know – our email address is
`council@ukuug.org`

UKUUG Linux Conference: Manchester

Roger Whittaker

UKUUG's "Linux 2008 Conference and Workshop" was held at the Manchester Conference Centre between the 7th and 9th of November. The Friday was the workshop day, when Luke Leighton presented an interesting day devoted to Python based web technologies: in particular Pyjamas and Django. The tutorial was well attended and provided a good introduction to these topics.

The conference proper began on Saturday morning. This year the conference consisted of a single track and all the talks took place in a comfortable lecture room. The programme was nicely balanced between presentations with a high degree of technical content and some more entertaining talks.

The first talk was John Pinner's "Python and System Administration" in which he showed how he had developed cross platform administration tools using Python's native capabilities, replacing some ad-hoc methods that he had previously provided to his customers.

Other highlights of the first day were Jon Dowland's description of his attempts to consolidate and organise system administrators' documentation using ikiwiki, and the talks on grid and cloud technology from Ruediger Berlich and Marcus Hardt. Arnd Bergmann of IBM and Kyle McMartin of Red Hat both spoke on different aspects of the challenges of porting Linux to multiple architectures. Jake Edge of LWN ended the day with an interesting talk on web application security.

The Conference Dinner was held in an upper room at a Chinese restaurant (even with local knowledge available, finding it in the customary Manchester rain took a while). Once there we enjoyed an excellent meal and had a lot of fun.

On the Sunday there was again a full programme of talks. Arnd Bergmann spoke about the use of PCIe as a high performance interconnect. Christoph Hellwig discussed the current state of XFS. Other highlights were Matthew Garrett's talk on power management and Steve Goodwin's highly entertaining description of his home automation system.

Sandro Groganz was unfortunately unable to be there in person to speak about “Marketing Open Source Software”, but presented his talk over a video Skype call, which worked remarkably well. The talks ended with Adam Trickett’s talk “Desktop Adapted for Dad” on how he provided a Linux desktop for his father, who had no computer experience at all.

Overall this was a good conference: the venue was suitable and the range of talks were good. It was a pity that the turnout of delegates was low this year. UKUUG Council are interested in trying to discover why this was so: please feel free to communicate your views on this.

What Pro Bono IT Experts can do for the UK Third Sector

John Davies

iT4Communities is a service which introduces IT professionals wanting to volunteer their skills to charitable organisations needing IT help. Pro bono IT expertise is highly valued by charities who often can’t afford the commercial rates of expert support.

The Scope for IT Volunteering

I’m sure a high proportion of UKUUG members already help charitable organisations with new systems, broken software and upgrading networks – to name a few examples. But there are about 1 million UK IT professionals in the UK (information from a speech by BCS President Nigel Shadbolt a couple of years ago). All of these professionals have the skills to make a real difference to the UK Third Sector It is the small to medium part of the voluntary sector who most urgently need pro bono support – they are not well equipped with IT expertise or even with sufficient knowledge to make intelligent purchases of IT expertise, equipment or software. This is where the iT4C programme helps the most.

Why do it?

So why should you think about volunteering through the iT4C programme?

- Volunteering for a charity you support feels good (whether it’s Greenpeace, Help the Aged – or Little Snoring Village Hall!)
- Its a chance to stretch your technical talents a little, improve your soft skills and get out of the usual routine
- You will see more of the world and meet some very committed people – its your choice – help in the ”nitty gritty inner city” or the local nature reserve
- You will find 100s of professionally written descriptions of what is needed on the iT4C website (more about these Project Definitions below)
- The charity you contact has already talked to iT4C about their needs and is aware of things like treating professionals professionally and the implied cost of managing an external consultant, pro bono or paid
- You can always contact iT4C for advice about specific projects or charities – or ask us to intervene if problems arise
- Open source is more and more important in the voluntary sector – so your talents are increasingly valuable (see my UKUUG Newsletter article last year)
- Company involvement in the programme is an excellent way to enhance corporate CSR for IT intensive companies – it involves staff in CSR, it builds connections and expertise and it enhances corporate image in a very IT-specific way

A brief word about definitions

At iT4C we use “charity” to mean the organisations that our volunteers and ourselves support. Many are registered charities but some are less formal voluntary groups or social enterprises. The term Third Sector is increasingly used to describe this part of UK society (the first two sectors are commercial and public). We also use *Pro Bono* to mean professional volunteering (*Pro Bono Publico* – for the public good).

So what have we achieved?

The iT4Communities (iT4C) programme was launched by the Information Technologists' Company, the 100th Livery Company of the City of London, in November 2002. Since then 5,500 IT professionals have registered with iT4C, over £3m worth of expertise has been delivered to the UK voluntary sector and almost 600 projects have been completed.

Typical projects are for small to medium charities and they range from brief advice on purchasing software or design of an HTML template to major web-based applications involving several weeks or even months of work.

iT4C uniquely (in the world as far as we know) provides end to end support for both charity and volunteer. We talk to every charity and write up their need as a Project Definition – which what you see on our website and our RSS feed.

And what do we ask of our volunteers?

You need to be an IT Professional – we ask that they have:

- One year's IT experience with relevant IT qualifications OR
- Three years' IT experience without relevant qualifications
- A professional attitude to volunteering – treat your engagements and clients exactly like paid engagements

So how do you get started?

- Look at our RSS feed
- Register with iT4C – one page registration
- Search our opportunities
- Filter our RSS (and let us know if you build tools to help you do this)
- Read our monthly newsletter
- Ask us at any stage

And if you don't have the time to volunteer then we would especially welcome your financial support. The downturn means that other sources are melting away and we rely on individual generosity to keep the service running: see
`http://www.it4communities.org.uk/it4c/home/About/support.html`

Pass this on to your richer friends if you like what we do!

And if you work for a company which supports charitable objectives them what better cause for an IT-focused company then iT4C? We are very keen to talk to companies about corporate sponsorship and corporate support for volunteering so put us in touch with CEOs and CSR people where you can.

That's all. Get in with touch me, John Davies, or the iT4C team.

Telephone: 020 7796 2144

The RSS feed of opportunities is on our home page:
`http://www.it4communites.org.uk`

References:

Open Source in the UK Voluntary Sector, John Davies
`http://www.ukuug.org/newsletter/16.1/#open__john_`

A community of IT professionals working in the voluntary sector
`http://lists.lasa.org.uk/lists/info/ukriders/`

The Growing Need for Free and Open Source Software (F/OSS) Governance

Andrew Back

In April 2008 Gartner predicted, "By 2012, more than 90% of enterprises will use open source in direct or embedded forms." To some a source of surprise others suggested that this was in fact a conservative estimate. Since that Gartner report there have been many other indicators as to just how far along the F/OSS adoption curve we are. For example Linux is seeing much increased use in consumer devices such as mobile telephones and netbooks, whilst even a staunch proponent of proprietary software that was previously diametrically opposed to F/OSS, has now assumed a position whereby it is to some extent embraced.

As enterprises continue to develop an appreciation of the benefits that F/OSS affords, the validity of the liberal licenses that underpin the development of free software continues to be tested, and subsequently upheld, in court cases where a licensee has violated the terms of a free software license. Perhaps most notably in 2008 in the Jacobsen v. Katzer/Kamind case, where the U.S. Court of Appeals upheld the validity of the Artistic License, and established that a violation of its terms would result in its termination and thus copyright infringement. This was a landmark case as it has implications for all free copyright licenses, e.g. the hugely popular GPL series. And whilst a U.S. court ruling the validity of the GPL has been previously upheld in Germany, and it is only a matter of time before a precedent is established in connection with free copyright licenses here in the UK.

Licensing is something that enterprises need to be clear on, and even when F/OSS has been sourced via a vendor assumptions should not be made and due diligence must be carried out. Any positive obligations, such as to make an offer to provide the source code to free software components, generally only come into effect with redistribution, and so for many who are simply engaged in own use it is often the case that no action is required. However, it is important to note that redistribution is not limited to the sale of software, and that the giving away of a free (as in free of charge) CD or download of software would also count as redistribution, as may the distribution of software to a partner organisation or other associated company, whilst not all free software licenses are created equal and the Affero General Public License (AGPL) regards even the use of software in service provision to a 3rd party as redistribution. Watch out SaaS providers. . . The moral of the story: check the license conditions and seek the advice of your legal counsel wherever any doubt exists!

As enterprises move up the F/OSS adoption curve and vendors increasingly integrate F/OSS into their solutions, the need for governance becomes ever more clear and licensing is just one area where care needs to be exercised. Due diligence must be carried out in a number of areas in order to ensure that enterprise adoption of F/OSS technology and principles is effective and appropriate, whilst new policy and process will be required in support of this, and education will be needed in order to bring enterprise understanding up to date.

Accelerating Adoption

The F/OSS paradigm presents enterprises with unprecedented flexibility in terms of support options: ranging from self-support through ad-hoc and multi-sourced support, to traditional contracted-out single supplier arrangements. As such the diversity of options can lead to confusion, whilst the ease at which software can be downloaded without cost and subsequently deployed, can lead to gaps in support if care is not taken. Again due diligence is required in order to ensure that, just as with proprietary software, you are either capable of supporting the software yourself else you have a contract with a 3rd party that is capable. Many enterprises will source much F/OSS via a vendor, and after all for most it would not make sense to support their own GNU/Linux distribution. However, even with GNU/Linux support in place it is very possible that developers are utilising F/OSS code from outwith the distribution, and for which there are no formal support arrangements in place. It may be that this is the case, but that your developers have a sufficiently deep knowledge of the technology and a relationship with the associated F/OSS community, which together enable them to effectively self-support. But this should not be taken for granted, and checks should be made as part of the acceptance into service process that takes place prior to a platform going live, if not earlier on in the project lifecycle.

The adoption of F/OSS will have implications for numerous functions across the Enterprise and consideration needs to be given as to how it fits into the bigger picture, for example:

- Architecture. It may be that you have an IT architectural framework; F/OSS can be readily downloaded free of charge, and in this way may bypass traditional authorisation processes and thus has the potential to lead to architectural non-conformance.
- Procurement. In cases where you buy-to-sell there may be a benefit to updating your supply contracts to take F/OSS into account, such that your suppliers disclose usage of F/OSS in their products, and provide you with the materials required in order to enable you to meet any positive obligations that come into effect at redistribution.
- Commercial. With proprietary software suppliers may have given you a large or even unlimited IPR indemnity and/or warranty. Where you source F/OSS direct these will be absent, and even in cases where you source F/OSS via a vendor don't assume that you will get the same level of protection. Commercial teams therefore need to take into account upstream IPR indemnities and warranties, and work closely with the company lawyers prior to making offer of IPR indemnities and warranties on F/OSS to customers.
- Legal. Enterprises are beginning to appreciate that in order to realise the full potential from F/OSS you need to participate and be an active community member. However, which license should you use for company contributions? What about liability? What if you have patents? Who retains copyright? These and many other questions can only be answered through working together with your legal counsel.

There will be domain experts spread across the enterprise for whom the adoption of F/OSS has implications, and who will be able to take action that will position it within the context of existing frameworks.

The role of the F/OSS governance function is to surface considerations such as those outlined, bring together relevant stakeholders and drive action that serves to accelerate enterprise adoption. It should not be simply concerned with policing use of F/OSS, and instead of creating additional fear, uncertainty or doubt it should strive to eliminate it.

Governance Communities

This article serves to only scratch the surface of F/OSS governance, however there are online resources and communities where additional information can be found and assistance sought: FOSSBazaar is a workgroup of the Linux Foundation, and describes itself as "dedicated to empowering individuals and organizations across the enterprise to understand the issues that exist with free and open source software (FOSS), the processes that can assist to properly govern the implementation and deployment of FOSS, and the tools that can be used to assist these efforts across the lifecycle of an open source project. The vision of FOSSBazaar is to serve as a gathering place to discuss, explore, share experiences and cooperatively solve issues related to FOSS governance. As such, the site serves the open source community as a forum representative of open source users and providers worldwide." Membership is open to all and the resources provided include guides on major governance topics, discussion forums and blogs, with a growing number of industry experts on hand to help answer any questions raised.
`https://fossbazaar.org`

The Free Software Foundation Europe's Freedom Task Force "coordinates a large network of lawyers called the European Legal Network" and is "particularly interested in finding experts in the EU27 countries where we don't already have coverage". Aside from being a contact point for F/OSS legal expertise, the European Legal Network is also engaged in developing resources such as industry/community legal best practice and guidelines for license compliance. Much of this is currently being drafted but will be openly published in due course.
`http://www.fsfeurope.org/projects/ftf/network.en.html`

Conclusion

F/OSS is finally reaching the stage of mass enterprise adoption, as even those who were previously detractors come to recognise it can compete with proprietary software in terms of functionality and quality, the economics stack up and the wider benefits are real. F/OSS communities and enterprises are to some extent still figuring out how to work together, whilst for the latter the unprecedented flexibility afforded can, without due care, lead to confusion and unnecessary risks being taken.

Although considerations to be made around licensing and support can at first appear daunting, the potential benefits to be realised from F/OSS by far outweigh any additional effort that may be required over use of proprietary software.

Though F/OSS governance is a relatively new discipline help is on hand, not only from dedicated governance communities but also from the wider F/OSS community. Inclusion, open access to information and a willingness to provide assistance prevails here just as in the development of F/OSS itself.

DISCLAIMER: I am not a lawyer and none of the above constitutes legal advice!

In Memoriam: Jonathan Bruce Postel (1943-1998)

Peter H Salus

Jon Postel, editor of the RFCs from 1969 till his death, and author of many important RFCs, died ten years ago of complications following surgery for heart valve replacement.

It is difficult for me, in retrospect, to discuss just how important Jon was and how valuable his contributions were.

I last had dinner with Jon in September 1995. He had been in hospital and did not look well. But he was cheery and witty.

Though there are many memoirs of Jon, my favourite is by Danny Cohen, who worked with him
`http://www.postel.org/remembrances/cohen-story.html`

He noted:

> *Jon was an authority without bureaucracy. No silly rules! Jon's authority was not derived from any management structure. It was due to his personality, his dedication, deep understanding, and demanding technical taste and style.*
>
> *Jon set the standards for both the Internet standards and for the Internet standardisation process. Jon turned the RFCs into a central piece of the standardisation process.*
>
> *One can also read that Jon was the editor of the RFCs, and may think that Jon checked only the grammar or the format of the RFCs. Nothing could be further from the truth, not that he did not check it, but in addition, being the corporate memory, Jon had indicated many times to authors that earlier work had treated the same subject, and that their work would be improved by learning about that earlier work.*

On October 16, 2008 it was a decade since Jon died. I still miss him.

Where Next?

Peter H Salus

This is my last column in the current series. For this reason, I thought I might contemplate the future, rather than the past. The past is somewhat clearer, hindsight being easier than foresight. For example, It's over 60 years since the transistor was invented. It's 25 years since rms launched the Gnu Project. Next year, the Net, UNIX and Linus Torvalds will be 40. It's 30 years since Microsoft moved from New Mexico to Washington. And both the Web and Linux will reach majority in 2009.

And on November 2nd the Morris Worm was 20.

Where are we going? What might the next decade bring? Where are today's visionaries? Not the ones who publish fiction posing as the future in *Wired* magazine. The ones who really see whole new ways of thinking about what is possible. You won't find them among the fashionista digerati – those guys think the iPhone is the future (when it is clearly the past – the future by definition doesn't exist and can't be extrapolated from one data point).

Were rms and Andy Tanenbaum and Linus the end of the line? Are Vernor Vinge and Cory Doctorow and Charlie Stross the only guys really thinking about the future?

Well, having mentioned Vernor, Stanford held its third Singularity Summit in late October. Yes, you'd have been able to listen to Ray Kurzweil, "updating his predictions" in *The Singularity is Near*, and Intel CTO Justin Rattner, who will examine the Singularity's plausibility. At the Intel Developer Forum on August 21, 2008, he spoke about why he thinks the gap between humans and machines will close by 2050. 'Rather than look back, we're going to look forward 40 years,' said Rattner. 'It's in that future where many people think that machine intelligence will surpass human intelligence.'

Kurzweil's book – now three years old – asked the familiar questions of how fast technology is improving and how those advances will affect humanity. For him, the answer is clear – humanity and technology will merge. The result will be Version 2.0 of humanity with enhanced intellect and bodies that won't "wear out." Kurzweil's "Singularity" is that point at which the merger will be complete. And "final" – a word to keep in mind. I guess he thinks that evolution just stops...

The basis of his thesis is the advance of technology, typified by GNR [Genetics, Nanotechnology and Robotics]. While these sound intimidating, one need not be highly conversant with the technologies to understand his argument. He explains them all clearly. Basing his project on the "Moore's Law" – computing power will double every eighteen months – Kurzweil shows how computer processing capacity will soon outstrip that of the human brain. Once that has been achieved, it will be a short step to enhance existing technology so that it can reform the human body. The heart, an inefficient and vulnerable pump, for example, can be replaced by a easily repairable mechanical version. But – wait a minute! – why's that a big deal? Can't we do that? Isn't cardiac surgery getting easier anyway? Yes.

Kurzweil in 2005 hadn't caught up with Vinge's 1991 *Across Realtime* – in which evolution continued, by the way. Incidentally, Vinge's "The Coming Technological Singularity" appeared toward the end of 1993 – 15 years ago...

In 1998, when Kurzweil stated we were going to become robots or fuse with robots, John Searle, a Berkeley philosopher, countered that this couldn't happen, because the robots couldn't be conscious.

And Rattner? His immediate past doesn't promote confidence in what we missed: in August, having said that he agreed with Kurzweil that the singularity would occur by 2050, he blithely moved to a pair of demos.

Jan Rabaey, a professor at the University of California-Berkeley, described his vision for the future of radio communications. Rabaey said he believes each of us will have about 1,000 radios soon, most of them vanishingly small. Radio devices, he said, will become "cognitive," so they can automatically sense where there's uncluttered radio spectrum available and which communication protocols should be used at a given moment. He also believes they'll become more collaborative – able to link together in a mesh network that collectively can transmit data faster, in greater quantity, more efficiently, and more reliably.

Then there was claymation – no, sorry, Will Vinton – claytronics from CMU. At the current stage of design, claytronics hardware operates from macroscale designs with devices that are much larger than the tiny modular robots that set the goals of this engineering research. These large devices are designed to test concepts for sub-millimeter scale modules and to explain crucial effects of the physical and electrical forces that will affect nanoscale robots.

The Phoenix Business Journal reported [21 August 2008]:

> *Robots with human intelligence, computer and technology devices that can change shape and wireless power supplies could be part of everyday life by 2050. ...*

Rattner said future advances could include technology and computers that can change physical shape via microrobots as well as devices that can recharge without power cords. He also envisions robots that perform personal tasks and have humanlike abilities, such as perception and awareness.

Real soon now.

Oh, did I mention that Intel is funding both projects? And they're the way of the future. Oh, no! Vista is. In the words of Windows senior VP Bill Veghte, Vista is trying to "deliver a world-class shopping experience that aligns with the brand promise of our online presence." I don't even know what that means.

Well, maybe I'm being nasty. Or sarcastic. But I do want to point out that we anoint vision in retrospect. So something we don't notice now might well be innovative in a few decades.

In *Amazing Stories* in 1929, Buck Rogers introduced his new jetpack. 32 years later, Wendell Moore of Bell Aerosystems, demonstrated his "small rocket lift device." He suffered a shattered knee. In 1965 we all saw it in *Thunderball.* We've heard little since, except that the US Army has a jetpack that will operate for "up to 20 minutes." What about Leonard Nimoy's universal translator on "Star Trek"? I can't even get sensible French to English, German to English or Dutch to English out of `dictionary.com` (which, incidentally translates streetcar into German as "Streetcar," rather than "Strassenbahn"). One more: robots. We have robopets and the Roomba, but what happened to that humaniform object with (dare I say it?) a positronic brain?

Let's face it, we do a really lousy job at prediction.

Dr Susan Calvin made her first appearance in "Robbie," but she wasn't in the version that appeared in *Super Science Stories* in September 1940. Asimov "adjusted" his history and inserted her in a revised version. But that wasn't all he revised: he moved the action from 1982 to 1998. As he died in 1992, I guess he didn't concern himself with shoving things further.

Nonetheless, having destroyed my original premise, I will forge ahead.

While every day must be some sort of anniversary, years ending in 5 and 0 seem to echo with us. And this year and next host a large number of birthdays. Darwin's *Origin of Species*, Marx's *Critique of Political Economy*, and Wagner's *Tristan* will all celebrate their sesquicentennials. 1963 saw the beginning of Project MAC at MIT and with it came time-sharing systems. Federico Corbato, thinking of Maxwell, coined "daemon" for a background process. 45 years ago. Ten years later, in 1973, Ritchie and Thompson gave the first UNIX paper. And a decade later, in September 1983, when he was 30, Richard Stallman announced the GNU Project. A mere quarter century ago.

By then, Linus Torvalds, born in Helsinki in 1969, was 14 years old and was playing with a Commodore VIC-20. Only a few years later, he spent his savings on a Sinclair QL, running at 7.5MHz with 128kB of RAM (the VIC-20 had 5k).

In 1991, Linus announced the availability of a part of his OS by ftp and it soon became popular, with several Linux companies established over the next few years, SuSE, Yggdrasil, and then 15 years ago, Mark Ewing established Red Hat.

Next year, 2009, both the Net and Unix will be 40. And so will Linus.

But there are some routes that won't be followed, that are (now) dead ends. For example, after six Nobel Prizes, the invention of the transistor, the laser and countless contributions to computer science and technology, it is the end of the road for Bell Labs' fundamental physics research lab. (The UNIX lab was "merged" over two years ago.)

Alcatel-Lucent, the parent company of Bell Labs, is pulling out of basic science, material physics and semiconductor research and will instead be focusing on more immediately marketable areas such as networking, high-speed electronics, wireless, nanotechnology and software.

If you had told me in 1958, when I studied FORTRAN, that I would see languages like C or C++ or Tcl or Java or Perl or Ruby, I'd have been incredulous. And if you'd tried then to tell me that 20 years later I'd be stopping folks in the hallway at the University of Toronto because I'd received email from a friend at

Columbia University in New York, I'd have thought you were nuts. And what about the Web? And on the hardware side, the PDP-8 I used at Toronto in 1969 had 4k and paper tape. My wife has a memory stick with 4G. 50 years since FORTRAN was any kind of big deal. 40 years since 4k was considered adequate memory. And Kurzweil and Rattner are pretending to look over 40 years ahead.

Well, I can fantasize too.

There's more thinking and experimentation going on in programming languages now than there has been in ages. Of course, the LISP contingent claims everything is old news because whatever it is, they invented it before the continents rose from the ocean. But that's not quite true.

And there's a lot going on in machine architecture now that simply yelling louder doesn't make the CPU go any faster. and that has huge implications for operating systems, not to mention machine organization (and that's at a level above architecture).

Remember that the very job of computing is innovation, and in the innovation business, better-faster-cheaper is just as valid as "new science," although we will likely be getting a healthy dose of each.

If the memristor really turns into what it looks like it could, the whole model of what a computer looks like will go out the window. How to exploit a machine with 100–200 gigabytes of stable, persistent fast "main memory," with processor state being persistent as well, very likely, is an interesting question. The basic ideas in operating systems will need to be revisited. But it was only on April 30, 2008, that a team at HP Labs led by the scientist R. Stanley Williams announced the discovery of a switching memristor. Based on a thin film of titanium dioxide, it has been presented as an approximately ideal device.

But, again, the MIT crowd believes everything worth knowing was done in Multics. I'm not trying to deny its influence, but that's my point. It's only been 40 years – you'd think that an original idea might emerge at some point. At least it should!

But how about something really novel? – Imagine what computers can do besides run Microsoft Orifice. The very definition of "computing" as known by 90% of the cognisant human race is so deep in a rut that they don't realise that looking sideways and seeing something other than dirt is an option. I still cling to that quaint notion of "a computer for your person" – something that aids and abets your activities, possibly even augmenting your analogue brain with some digital cleverness, as opposed to the "personal computer" where the human's job is to supply the mental mammary glands that the PC locks lips upon and won't let go for love nor money, resulting in a deafening sucking noise.

Everything will keep on getting smaller. Nanoisation will be limited by physics, but we're not there yet, as the memristor shows. In medicine, this means that *Fantastic Voyage*-style medical instruments will be able to operate internally. And wires will get fewer. 802.11 and its alphabet soup will get better and better and we'll have data transfer rates beyond the 100 megabits that .11n will propose (probably at the end of next year).

Fifteen years ago, Vint Cerf, one of the true fathers of the internet, went around wearing a button that read: **IP on everything**. We might be getting there. After all, it's only a bit over a decade since RFC 1886 and eight years since I predicted that IPv6 was "imminent."

We're not going to stop the growth of the internet nor (more important) the multiplicity of small devices that require net access. So we'd best get going on that addressing, or we really will run out. The cell phone has proliferated like kudzu and applications for it (or for Android) are multiplying.

In transportation, I really hope that something supersedes the personal internal-combustion automobile. But I have no idea what it might be. I have all sorts of global warming fears that make me think we might not even be around to worry about things a half-century from now. And I don't mean a *Waterworld* sort of dystopia. I mean gone the way of the dodo and the aurochs. Many insects will survive; few mammals will.

And, outside of sf novels, humans won't be off terraforming distant planets. Astronomers have found about 300 objects orbiting distant stars. None of those stars is like our sun; none of those worlds is at an "appropriate" distance from its luminary. They seem to be Jupiter-like. And I really don't think we'll have soap-bubble cities on the moon or Mars or Venus. We've got this planet to destroy; we won't be off destroying others. Klaatu's warning may be why the ET visits just haven't happened.

While I'm on the movies, it might be that *The Six Million Dollar Man* movie of 1973 and the programs of 1974 to 1978 and *The Bionic Woman* sequel (1975) weren't that far off. Medicine has made great strides and will doubtless make many more. Effectively, this means that I see the singularity in much the same way that Vinge does: we will be able to ameliorate all sorts of failings through prosthetics and nano-surgery. Look up the "Open Prosthetics Project" and see the potential of open source design – free for anyone to use.

In a roundabout fashion, that brings me to what most of you work with: free and open software. And that means we share a pet hate in a Seattle suburb. Well, despite the apparent disaster of Vista, Microsoft won't disappear soon. But it will level off and then shrink. A brontosaurus couldn't change direction with alacrity. Microsoft can't manoeuver as IBM did a decade or so ago. All Microsoft has is an ungainly clunker of an OS and a lot of junk attached to it.

They can't just go to a truly new OS. They can't unlink their browser – who'd purchase Internet Destroyer willingly? There's nothing they can turn to. But death will come slowly.

According to Webmonkey, Microsoft is already planning to remove e-mail, photo editing, movie making and other secondary software offerings from the upcoming Windows 7. Windows Photo Gallery, Windows Mail, and Windows Movie Maker were all part of Vista, but, in an effort to cut down on development time of future OS updates, Microsoft plans to give them the heave-ho.

Interesting. In light of Microsoft's troubles in the EU, I wonder whether Windows 7 will accommodate Firefox or OpenOffice or . . . I'm not joking. Mary Jo Foley, a Microsoft tracker whose views I value, noted that the peek she had at Windows 7 didn't look or feel that different from Vista. If this is true, rather than Microsoft merely dragging a false trail, it will be a disaster for Redmond. And I can't say I'll feel bad about it.

But these aren't much in the way of predictions.

Maybe that's it. I'm not Ray Kurzweil. maddog says my "integrity is beyond bounds." Modestly, I'll say that I try to be honest.

So . . . free and open software will wax and florish. Microsoft will continue on a long-term slide as ever more countries adopt Linux and open applications.

Hardware will get smaller and smaller with ever-increasing memory.

Medical applications will be in the forefront with communications and entertainment following close behind. And IPv6 will finally achieve full acceptance.

I could claim lots more – I won't be here in 40 or 50 years to have to explain why I got it wrong. But I won't. And I wish there were a trans-Atlantic tunnel – it's only 36 years since Harrison's novel..

Concurrent Patterns: Parallels in System and Language Design

Jason Dusek

Concurrency and parallelism are usually discussed together, and they are both important in reorganising a program for execution on multiple cores or multiple computers. Parallelism asks, "How can we break a program up into independently executing parts?" It is very much in the domain of compiler writers and library providers, as the necessary program transformations are tedious at best. Concurrency asks, "How can independent programs share a value once in a while?" It bedevils program designers at every level – providers of parallelism toolkits, operating systems designers, application programmers, system administrators providing distributed network services, and even folks just trying to sell a few books.

Sharing

For programs to share a value, they may both possess a copy or they may both possess a handle.

If they both have a handle, that is cheapest and allows them to communicate most efficiently. For example, with file handles, two applications opening the same file are using less memory than two applications with one sending a copy to the other. They also communicate more quickly. However, handles can result in confusion and danger – for example, memmapped files. If two programs memmap the same file, and the "receiver" is not careful to close it and reopen it at the right time, one can very likely crash the receiver by writing into the memmapped file. The less extreme example of two database clients overwriting parts of one another's changes, resulting in a mutually confusing (but readable) record, is a more common pitfall of these handle-based (we'll call them "shared-memory") approaches to concurrency. Down this path lay transactions and locking. There are numerous forms of locking, the most dreadful being spin-lock, deadlock, and live-lock.

When we pass around copies, each process can pretty much go on its merry way from the point of copy transfer. As mentioned earlier, this can result in a great waste of memory. These "message-passing" strategies are inevitable, though, in network services – an LDAP server cannot share memory with its clients; it has to just send them the data. If we structure all our applications this way, without regard to locality, then it is at least consistent, if not efficient. Garbage collection is very important in message-passing architectures, because you create a lot of garbage by copying things all over the place. However, a program is safe from interruption from other programs as long as it does not get (or request) any messages. The problem of data synchronisation can be resolved through frequent polling or publish-subscribe (possible in LDAP, if you are brave) – but at least there is some kind of warning, some kind of formal indicator, that a resource has changed.

At this point you may wonder how the two database clients are sharing memory while the LDAP server and its clients are passing messages? It is really a matter of where I choose to point the camera. Between the servers and clients there is message passing – clients make a request and servers send back a message full of data. Any changes in the server's data do not affect the clients at all as long as they don't ask for updates. Between the two database clients, there is shared state – each application writes to the database and reads from the database to communicate with the other processes. The difference shows up in where we put the locks – a database locks its tables, not its socket.

We Don't Need Concurrency In Programming Languages…

It stands to reason (although I cannot prove it) that all models of concurrency are explored in systems first and in languages later. Network services and operating systems offer a kind of parallelism – operating system process, multiple clients, servers running on distinct machines – and thus invite all the problems we usually associate with threading and message passing.

Within the *NIX filesystem, as in other shared-memory systems, we need to protect processes from mutual confusion – we need transactions, locks, and semaphores. The *NIX filesystem models all three. Through file locking, multiple POSIX programs protect one another from contradictory updates. A *shared lock* allows multiple shared locks to coexist with it, whereas an *exclusive lock* can only be taken when there are no other locks. When the locks are advisory – the default for *NIX – the lock is merely a baton, which processes can only possess under certain circumstances. These advisory locks are really semaphores, tokens of resource ownership without enforcement [1, 2]. In contrast, mandatory locks are true locks – processes block if they don't have the right lock. No reads can proceed without some kind of lock, and writes cannot proceed without a write lock. Often, an operating system will provide special signals and subscriptions to allow a process to know that a file is again open for reading, that another process wants to write a file, and other things. We can see how message passing and shared memory complement one another [3].

A transaction is a collection of operations on different resources that is performed in an all-or-nothing manner. To perform a transactions in a *NIX filesystem we must:

- Lock the greatest parent of all the files in the transaction and every subordinate file. (We'll be waiting a long time to get all those locks in an active system.)
- Recursively copy the greatest parent into a new directory.
- Perform our changes.
- Use `mv` to atomically overwrite the old greatest parent with the now updated copy.

Our rather overly generous lock scope has prevented anything from changing in the meantime.

Why must we make such a gratuitous copy? Because the filesystem does not offer a means to atomically `mv` several files. What if we got halfway through the `mv`s and the user sent `SIGKILL`? Then we'd have left the system in an inconsistent state, with no means of recovery. So we have to find a way to mv all the changes at once, and that means we have to do the copy. If we had local version control we could be more flexible, simply reverting changes if there were a failure, while keeping everything exclusively locked so no other processes could read the corrupt data until the "rollback" is complete. (Can we be sure reverts never fail?) Although rollbacks are not required for transactions, we see how they can protect us from over-locking.

As mentioned earlier, *NIX provides "signals", a form of message passing. Sockets, a more general and flexible mechanism, allow processes to communicate with one another by establishing a connection and sending messages back and forth, which corresponds to the *channels* of Hoare's CSP [4, 5]. A one-way channel is a *pipe*, both in Hoare's terminology and in *NIX. Unfortunately, *NIX does not allow us to open multiple pipes to a process, so this is a point where *NIX and Hoare diverge. A named pipe – a FIFO – is like a mailbox, a well-known place to leave a package for another process. Unfortunately, FIFOs allow IO interleaving and thus cannot really be used as mailboxes. But I think you get the idea. So what are signals? Signals are mailboxes – the kernel knows where to put messages for every process. We look up its PID to send it a message. Channels, pipes, and mailboxes can fake one another:

- To get a mailbox from channels, the receiver should simply aggregate all messages from every sender regardless of the channel they came in on. To get channels from mailboxes, we must always send a "return address" as part of the message. The receiver, when reading the messages, uses the return address to know which connection it is handling and where to return results.
- To get a pipe from a channel, only send messages in one direction. To get a channel from pipes, we need two pipes, one in either direction. The sender and the receiver must have the sense to keep the pipes together.
- To get pipes from mailboxes, we can *multiplex* the mailbox. Once again, we use the "return address", but this time, we only use the return address to tell us where the message came from, not how to send it back.

Insofar as they work, *NIX concurrency features support a strategy of composing concurrent systems from operating system processes. This is nice in so many ways – the file system handles garbage collection and caching, the scheduler distributes processes over the hardware, and programs in multiple languages can share resources and concurrency model. However, *NIX file locking is brittle, and the only out-of-the-box message-passing model is channels, by way of sockets. *NIX turns out not to be the best platform for an application that is looking for operating-system-level concurrency – but a search for an alternative leads us far afield.

Bell Labs' Plan 9, an evolution of UNIX with some of the original UNIX team on board, offers pipes in the sense of Hoare's CSP. Through resource multiplexing, the same name in the filesystem refers to different resources for different processes, converting a FIFO to a pipe to avert the IO interleaving that bedevils *NIX FIFOs [6]. We could probably emulate this system on any other *NIX, using bind mounts, union mounts, and pipe polling, but it would not be pretty.

At just about the time of Plan 9's emergence, Tandem's NonStop platform was in decline. NonStop SQL ran on top of the Guardian cluster operating system. In Guardian, every resource – every file, even – was a "process" that could receive messages [7]. Pervasive message passing is the door to easy clustering. NonStop SQL was able to run transactions across the cluster, which is a nontrivial task; and Guardian was able to fail over a process from one machine to another if the need arose [8].

There are numerous "cluster operating systems", but what they address is more a matter of resource usage than concurrent design primitives [9-11]. Plan 9 and Guardian are special because they make message-passing tools available to the application programmer and provide an environment where those message-passing tools are widely used.

Networks services model a few concurrency patterns not mentioned above.

Multi-view concurrency, which we might call transactional copy-on-write, is used in some SQL databases and is a long-tested approach to ACIDity. To read, read. To write, copy everything you wish to read or write, prepare the changeset, and the database will apply it if (only if) nothing that was read has been written since, and nothing that was written has been read or written since [12-14].

IRC is an example of a publish-subscribe message-passing service. We subscribe to the channel and receive change sets to the channel – messages – as they arrive. There is no way to pull "the" channel, though – and so we sometimes miss messages, to the amusement of all [15]. In contrast, OpenLDAP offers publish-subscribe messaging as an optimisation on the shared memory model. A client subscribes to an LDAP subtree and, as changes are made, they are forwarded [16]. However, there is one true tree – the tree on the LDAP server – and we can synchronise with it to ensure our copy is correct [17].

...But We Like It

Even when operating-system-level concurrency works, there's some mnemonic load in handling it. Network-service-level concurrency is in some sense simpler, but it also handles less – process management is delegated to the operating system. Concurrent programming languages specify an entire concurrent system: resources and a model for their use, as well as processes and a means of creating them.

Language-level concurrency has tended toward message passing. Early versions of Smalltalk were distinctively message-passing, and Carl Hewitt, the founder of the "Actors model", was inspired by Smalltalk [18]. More recent message-passing languages include Stackless Python, used to implement the massively multiplayer game EVE [19]; Limbo, a project by the team that worked on Plan 9 [20]; and Erlang [20]. The former two are channel languages, whereas Erlang is a mailbox language. MPI, a message-passing library and runtime for C, Java, C++, O'Caml, and Fortran, brings messagepassing to languages that do not have it natively [21, 22]. Shared memory is an unusual paradigm at the language level.

Erlang, a Message-Passing Language

Erlang, superficially similar to Prolog, makes RPC a language primitive. Processes (function calls) can send messages, listen for messages, and access their present PID. They perform message sending and non-concurrent things (e.g., math and string handling and ASN.1 parsing) until they hit a receive statement, which is rather like a case statement, only with no argument. The argument is implicit – the next message in the mailbox. After a message is handled, the executing function may call itself or another function recursively, or it may do nothing, which ends the process.

Erlang processes run within instances of the Erlang virtual machine, called nodes. Nodes are clustered to form applications that run on several machines at once. Although Erlang is often called a functional programming language, this is missing the point. Erlang offers easy IPC, a notion of process hierarchy and identity, rapid process spawning, and excellent libraries for process control and monitoring. Erlang is what the shell could have been.

Processes register with a cross-node name server so that they can offer named services to other processes. Processes are "linked" to other processes in the sense that a parent is made aware of the linkee's exit status. The combination of global name registry and linking allows us to implement reliable services in the face of "fail fast" behaviour. Processes are arranged with a monitor – an error handler – linked to a worker. The worker runs a function that registers itself for the global name and then calls the main function that handles requests and recursively calls itself over and over. The monitor hangs out. If the server process dies, the monitor receives a signal to that effect and recalls the function.

The special thing about Erlang is not that we can write programs this way, but that we write *every* program this way. Programs are collections of communicating services in Erlang, not a main thread of execution with subroutines (at least, not on the surface.) The innumerable executing threads are easy to parallelise, because they share no state with one another and thus can be interrupted and resumed at any time. When we see Erlang trumpeted as the multi-core solution, that is why.

Erlang's bias toward concurrent design is perhaps too great. Although the standard libraries are rich in protocol handlers and design patterns for concurrent applications, the language is weak at string handling and arithmetic.

Haskell's Approach to Shared Memory

Programming languages that offer safe access to shared memory are rare. Haskell, a purely functional language, offers "Software Transactional Memory", which is much like multi-view concurrency in databases.

Haskell's type system allows it to make very strong guarantees about shared memory operations. A pure function is a function that yields the same answer for the same arguments [e.g., `sin(x)`] whereas an impure function – hereafter a procedure – returns different things depending on the context [e.g., `gettime()`].

How do we perform input and output and get the time of day in a purely functional language? It turns out that there is a "pure view" of these impure operations. The principle is not hard to understand – values from impure functions are wrapped in a special container – so `gettime` does not return an `Int`; it returns an `IO Int` , an `Int` within the `IO` container.

A pure function, even when it shares state with another pure function, is not allowed to mutate it. It is immaterial in which order we evaluate the pure functions, so long as we evaluate calls that a function depends on before evaluating the function itself. This no-mutation property makes parallelisation easy, and the Haskell compiler takes advantage of that.

So we have all these functions, and they are likely to execute at any time. They do not have process identifiers, and there is no global registry of names. How do these functions communicate? One means, an early one, was to in fact brand every concurrent function with `IO` and force it to operate on references. This brings us all the problems of shared memory and none of the solutions; it also forces the compiler to be as paranoid about code accessing references as it is about code accessing the filesystem or network. The Glasgow Haskell Compiler project later introduced a new container, `STM`, which is specifically for operations that work on a large global store of values [24]. A procedure that executes within `STM` creates a log of its reads and writes. When the procedure ends, the log is checked to ensure that none of the values have changed since the procedure read them. As long as the reads are okay, the writes in the log are committed. If they are not okay, the procedure retries until it works (including forever).

`STM` offers true fine-grained transactions on a runtime system that can run millions of threads, but there are no provisions for clustering or process hierarchy. This is in some sense inevitable; shared memory systems don't dovetail nicely with the network's natural separation of state across servers.

Concurrency and You

Splitting a program into concurrently running parts can simplify design and always parallelises the program. Whether this results in a net performance benefit depends on the overhead of communication.

To take advantage of concurrent design and implicit parallelism, one must adopt a new way of thinking about program structure, data structures, and efficiency. In-place update is efficient and natural in sequential computing, but in concurrent systems it is fraught with peril and obstructs parallelism. Bolting concurrency onto a sequential language – an imperative language – leads to inconsistency at best, and so it is understandable that much work in concurrency has taken place under the declarative tent.

Concurrent Perl would find many users if it existed, and there have been numerous attempts to bring concurrent programming to C and C++. Concurrency-friendly languages are unusual languages, bringing more or less of the functional paradigm with them. Will a new language gain a foothold, or will we find a way to bring message passing or transactional memory to C?

Perhaps, like object-oriented design, concurrent programming will find wide use only after it has been integrated with C. Toolkits for parallelism in C, C++, and Java are certainly catching up, although they are used mostly by game programmers and authors of scientific visualisation software. Languages used mostly in Web programming and system administration have not received that kind of attention, and consequently we see the growth of Erlang in the area of high-availability network services. Niche programming languages will always have their niche, and it's likely that mainstream programming languages will always have the mainstream.

References

[1] `fcntl` man page: fcntl - manipulate file descriptor:
`http://linux.die.net/man/2/fcntl`

[2] Semaphores:
`http://www.ecst.csuchico.edu/~beej/guide/ipc/semaphores.html`

[3] Andy Walker, "Mandatory File Locking for the Linux Operating System", April 15, 1996:
`http://www.cs.albany.edu/~sdc/Linux/linux/Documentation/mandatory.txt`

[4] C.A.R. Hoare, "Communicating Sequential Processes: 7.3.2 Multiple Buffered Channels", June 21, 2004:
`http://www.usingcsp.com/`

[5] Sean Dorward, Rob Pike, David Leo Presotto, Dennis M. Ritchie, Howard Trickey, Phil Winterbottom, "The Inferno Operating System":
`http://doc.cat-v.org/inferno/4th_edition/inferno_OS`

[6] Rob Pike, "Rio: Design of a Concurrent Window System", February 4, 2000:
`http://herpolhode.com/rob/lec5.pdf`

[7] The Tandem Database Group, "NonStop SQL, a Distributed, HighPerformance, High-Availability Implementation of SQL", April 1987:
`http://www.hpl.hp.com/techreports/tandem/TR-87.4.html`

[8] Wikipedia, Tandem Computers:
`http://en.wikipedia.org/wiki/Tandem_Computers#History`

[9]: C. Morin, R. Lottiaux, G. Vallee, P. Gallard, D. Margery, J.-Y. Berthou, I. Scherson, "Kerrighed and Data Parallelism: Cluster Computing on Single System Image Operating Systems", September 2004:
`http://www.ics.uci.edu/~schark/cluster04.ps`

[10] Wikipedia, QNX:
`http://en.wikipedia.org/wiki/QNX`

[11]: Nancy P. Kronenberg, Henry M. Levy, William D. Strecker, "VAXclusters: A Closely-Coupled Distributed System", May 1986:
`http://lazowska.cs.washington.edu/p130-kronenberg.pdf`

[12] Wikipedia, MultiView concurrency control:
`http://en.wikipedia.org/wiki/Multiversion_concurrency_control`

[13] David Patrick Reed, "Naming and Synchronization in a Decentralized Computer System", September 1978:
`http://publications.csail.mit.edu/lcs/specpub.php?id=773`

[14] Philip A. Bernstein and Nathan Goodman, "Concurrency Control in Distributed Database Systems", June 1981:
`http://www.sai.msu.su/~megera/postgres/gist/papers/concurrency/p185-bernstein.pdf`

[15] RFC 1459: IRC Concepts:
`http://www.irchelp.org/irchelp/rfc/ chapter3.html`

[16] LDAP for Rocket Scientists: Replication refreshAndPersist (Provider Push):
`http://www.zytrax.com/books/ldap/ch7/#ol-syncrepl-rap`

[17] LDAP for Rocket Scientists: Replication refreshOnly (Consumer Pull):
`http://www.zytrax.com/books/ldap/ch7/#ol-syncrepl-ro`

[18] Alan Kay, "The Early History of Smalltalk: 1972-76 – The First Real Smalltalk":
`http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html`

[19] About Stackless:
`http://www.stackless.com/`

[20] Limbo:
`http://www.vitanuova.com/inferno/limbo.html`

[21] Concurrent programming:
`http://www.erlang.org/doc/getting_started/conc_prog.html`

[22] Wikipedia, MPI:
`http://en.wikipedia.org/wiki/Message_Passing _Interface`

[23] MPI Basics:
`http://www-unix.mcs.anl.gov/dbpp/text/node96.html`

[24] Simon Peyton Jones, “Beautiful Concurrency”, December 22, 2006:
`http://research.microsoft.com/~simonpj/Papers/stm/beautiful.pdf`

*Originally published in **;login: The USENIX Magazine**, vol. 33, no. 5 (Berkeley, CA: USENIX Association, 2008). Reprinted by kind permission.*

Driving the Evolution of Software Languages to a Concurrent Future

Andrew Brownsword

Concurrent: existing, happening, or done at the same time.

Concurrency has become a hot topic in the past few years because the concurrency available in hardware is increasing. Processor designers have reached a point where making sequential tasks take less time has become impractical or impossible – the physical limits encountered demand engineering tradeoffs. But writing software that takes advantage of concurrency is hard, and making that software perform as well on different CPU architectures is all but impossible. In this article, we will explore the reasons why this is currently true with specific examples and consider how this evolution represents a changing paradigm that renders the traditional imperative programming model fragile and inefficient.

For the past 20 plus years, hardware designers have been using concurrency in the form of pipelining, superscalar issue, and multitransaction memory buses to improve the apparent performance of what appears to be sequential hardware. Since about the late 1990s most microprocessors have also included SIMD instructions, which are typically capable of 4 FLOPs (Floating Point OPerations) per instruction. More recently, some processors support multiple threads per core (from two in Intel’s Hyperthreading, up to eight in Sun’s Niagara architecture), and now processors with two or four cores are becoming the norm. Michael McCool’s April 2008 ;login: article provides an overview of these techniques.

Computational efficiency

Most high-performance modern hardware these days is theoretically capable of about 10–20 FLOPS for each floating point value moved through the processor. In terms of latency, approximately 400–4800 FLOPS could theoretically be done in the time it takes to fetch a value directly from memory. These are theoretical peak computation rates, based on the four-way SIMD capabilities that most processors have now. Theoretical rates are not reached in practice, so what can we really expect to achieve and what do we see in practice?

How efficiently a processor executes is a function of the details of the processor’s operation, its memory subsystem, and the software that is running. Typically all operations are scalar because that is how most languages are defined. ALGOL set the pace, and most imperative languages since then have been embellishing on the basic model. They embody the ideas of the sequential Von Neumann architecture and are notations for describing a sequence of scalar operations. These operations are usually dependent on the output of operations that came immediately before, or nearly so.

Also, typical code sequences are filled with decision points and loops, which appear as branch instructions that disrupt the efficient flow of instructions. Branch frequency and data dependencies in typical code are a frequently measured metric by hardware and compiler developers. In the mid 1990s, IBM found during the development of the PowerPC 604 that branches occurred, on average, once every 6 instructions. This makes it largely unproductive to have hardware dispatch more than about 3 or 4 instructions per clock cycle. To this day most hardware is aimed at 2- to 4-way dispatch, which is unsurprising, since software hasn’t changed substantially. More recent tests on an inorder processor showed that most of the software for gaming platforms averaged about 10% of the potential instruction dispatch rate. And very little of that software was using the considerable SIMD capabilities of the hardware, leaving the realisation at less than 3% of the processor's theoretical computational capability. Instead of the theoretical potential ˜25 GFLOP, less than 1 GFLOP was realised. Out-of-order-execution processors will do somewhat better than this, but usually only by a factor of 2 or 3. In recent years both the number of cores in one system and, on some cores, the number of threads executing on each core have increased beyond unity. I will ignore the distinction between hardware threads sharing a core and multiple cores for the rest of this article and will refer to simply "threads".

Memory and Threads

Multiple threads must share the system in which they exist. The most important shared resource is main memory (RAM). In most systems there is a single large pool of RAM. The system may also have a GPU processor, and some architectures give the GPU its own pool of RAM. Some systems partition the CPU memory by attaching parts of it directly to each processor; this is called NUMA (NonUniform Memory Access). How each processor accesses data at a given memory address, and how long it takes to do it, depends on where that memory is physically located. The sharing of memory is complicated by the fact that processors use on-chip caches to speed up memory access. When the cache isn't shared by all threads, the potential exists for a given memory location's actual current state to be sitting in one thread's cache when another one needs it. Most hardware implements "cache coherency", which is a mechanism that tracks where each memory location's actual state is and retrieves it from that location when requested. Having multiple threads reading the same location is dealt with by having multiple copies of the state in the various caches. Writing to the same location, however, is problematic because a single current state must be kept up to date and it is usually placed in the cache of the thread that most recently modified it. If more than one thread is continuously updating the same location at the same time considerable inter-cache traffic may result.

From a developer's perspective the problem with multiple threads, at least in an ALGOLlike language, is that the program must be explicitly written to take advantage of them. Given a program written for a single processor machine, all but one thread will sit idle unless the operating system has something for additional threads to do. Typically the OS has enough to keep a second thread at least partially busy, and some OS services are now internally moving computation to other threads (graphics, movie playing, animation, etc.) if those services are in use. If the hardware has more than two threads, however, then they are going to be doing little to improve your software's performance. As a result, if you are using 3% of one thread's potential and you have a four-core machine, for example, you are now only using <1% of the system potential. In theory you could speed up your software by a factor of over 100 or be doing 100 times as much computation on the same hardware.

Programming for Concurrency

So how do you take advantage of this ability of hardware to operate concurrently? Broadly speaking, there are two kinds of available concurrency: instruction-level and thread-level.

Instruction-level concurrency has largely become the domain of the compiler, but that doesn't mean there is nothing you can do about it. The choice of language you use and how you use it have an enormous impact. As already mentioned, most current languages embody a fundamentally scalar and sequential programming model. Some compile to an intermediate form that also embodies a simple sequential machine, and the need for efficient JIT severely limits how the compiler can optimise. Fully natively compiled languages, or those with more powerful intermediate forms, may have compilers that can perform aggressive optimisations (such as auto-vectorisation) in specific circumstances, but these techniques tend to be fragile and provide limited results. In C and C++, the language provides fairly low-level-detail control over the emitted instructions, and the compilers often support hardware-specific language extensions to access SIMD instructions and other unique hardware features. With careful, labour-intensive techniques, highly specialised platform-specific code can be written to dramatically improve performance. This kind of code optimisation requires substantial expertise, detailed knowledge of the hardware platform, and a great deal of time, and it incurs substantial maintenance costs. On modern hardware it commonly delivers 4–20x performance improvements, raising the 3% utilisation into the 10–60% range. Notice that this is as good as or better than the improvement you might expect by going from a single to a many-threaded processor. Not all problems can be optimised in this fashion, but more are amenable to it than most developers seem to think. This kind of data-parallel solution also tends to be more amenable to being divided across multiple threads, making it easier to achieve thread-level concurrency. Unfortunately, the aforementioned problems make the process of experimentation and exploration to find efficient new data-parallel solutions very difficult and expensive.

Achieving a high level of instruction-level concurrency boils down to writing code that fits into a set of constraints. The precise definition of these constraints depends on the hardware, but there are some general principles that apply. Choose data structures and access patterns that are uniform and predictable, organise data so that access to it is spatially and temporally dense, choose data structure alignments and sizes, apply precisely the same sequence of operations to many data elements, minimise data dependencies across parallel elements, define interfaces where each interaction specifies a large amount of work to be performed, and so forth.

Unfortunately, most programming languages do nothing to aid the programmer in these pursuits, if they allow these things to be under explicit programmer control at all. Even worse, code written in these languages contains too much detail (i.e., the semantic information available to the compiler is at a very primitive level) and this limits what the compilers are capable of, and what they are permitted to do by the language rules. There are a few languages and alternative notations available (Michael McCool's own RapidMind being one proprietary example; others are StreamIt, Intel's Ct, etc.), but they are far from pervasive and thus far there is no widely available standard that integrates tightly enough with the standard environments (such as C/C++) to be adopted. Tight integration and, where possible, familiar syntax is essential for practical efficiency reasons, for programmer comfort, and for gradual adoption by an industry heavily invested in existing languages. Exactly what such a language should look like is open for debate, but my opinion is that burdening an existing grammar of already excessive complexity is not the correct solution.

Thread-level Concurrency

Thread-level concurrency tends to receive more attention, largely because it is easier to see and understand. It is accomplished in most languages by calling a threading interface provided by the OS, as part of a support library, or as a language feature. These vary in their details and features. The detailed semantics and capabilities of how threads work between platforms and interfaces vary significantly. How they get scheduled to run, whether priorities can be set, how the threads interact based on these priorities, whether a given software thread will stay on a given hardware thread, how long it will run without being interrupted, and so on can show up latent bugs and result in noteworthy performance differences that are hard to diagnose.

Threads running concurrently will inevitably want to interact and will therefore need to access some piece of shared state. This generally isn't a problem until one or more of the threads wants to modify the shared state. Then we run into a problem called the "lack of atomicity". Most operations are actually composed of several hardware operations. Incrementing an integer variable, for example, reads the value from memory into a register, performs the add operation, then stores the value back to memory. Even with hardware that has an instruction that appears to add a value directly to memory, it in reality implements this internally as an read–modify–write sequence. When each of these operations is a pipelined instruction, which is itself broken down into many stages, it should be clear that there are a lot of steps involved in doing even this most trivial of examples. When these steps are happening in a machine where there are additional threads sharing the memory location being modified, the possibility exists for conflicting updates. Exactly what happens to the value depends on the operation, its hardware implemententation, and the precise timing of each incident of conflicting change. This means that even this trivial example can do something different every time it is executed, and even if timing were somehow stable then executing on different (even nominally compatible) hardware may impact the outcome.

Many thread interfaces provide some basic "atomic operations". These functions use capabilities provided by the processor to ensure atomicity for this kind of simple read–modify–write operation. Usually the hardware provides a base mechanism to use instead of the atomic operation itself. This is usually either a compare-and-set operation or a reserve-and-conditional-store operation. I'm not going to describe what these two things are here, but the salient point is that using them correctly and ensuring atomicity is significantly more expensive in terms of complexity, instructions, performance, and programmer knowledge than just a simple non-atomic operation. They also don't guarantee atomicity across a series of operations.

An important point to note about primitive hardware synchronisation operations is that they cause the hardware to synchronise. This seemingly trivial point has unfortunate consequences. It usually means that at least part of the hardware must stop and wait for some set of operations to complete, and often this means requests on the memory bus. This interferes with the bandwidth across the bus, which is one of the system's key performance bottlenecks. The more aggressive the processor is about concurrent and out-of-order operations, the more there is to lose by forcing it to synchronise.

The problem with the lack of atomicity goes far deeper than the trivial increment example just cited. Consider this simple piece of C code:

```
int done; // assume this is an
          // initialised shared variable
if (done == 0)
{
  // do ``stuff'' here that you want
  // to happen once
  done = 1;
}
```

If you have two threads and they try to execute this code at the same time, then both may perform the test and do the "stuff" before done is set to 1. You may think that by putting the `done=1` before doing the "stuff" you can fix the problem, but all you will do is make it happen less often. The instant that done is read for the comparison, another thread may come along and read the same value so that it can make the same test succeed. Subtle changes in the hardware or OS can dramatically impact how often this code fails in practice. One solution to this example is to use an atomic operation that your threading interface provides, but this doesn't get you very far because it will provide only a limited set of atomic operations and you can't restrict your programming to just using those!

Synchronisation

The standard solution to this is to use synchronisation primitives. Each threading interface provides some set of "synchronisation primitives". These are usually fairly similar among interfaces, but the semantics can differ in subtle but important ways. Each primitive also brings with it different performance implications. Using some lightweight synchronisation, even if blocking or waiting doesn't happen, might consume only a few tens or hundreds of cycles, whereas others might consume thousands (possibly many, many thousands). And this is the cost if there is no contention between threads!

Most of these primitives are in the form of a lock/unlock pair of operations. In some cases it is called enter/leave instead of lock/unlock because the pair is defining a "critical section" of code that you enter with the lock (or enter) operation and that you exit with the unlock (or leave) operation. Only a single thread can be inside such a critical section at a time. Other threads attempting to enter are forced to wait until the one in the critical section leaves it (i.e., they are "mutually excluded", which is shortened to "mutex"). This ensures that the code in the critical section is executed atomically. Unfortunately, this also forces this part of the program to be effectively single-threaded, and spending too much time in critical sections reduces your performance to that of a single hardware thread (or less since executing synchronisation primitives has a cost).

```
int done; // assume this is an
          // initialised shared variable
enter_critical_section();
if (done == 0)
{
  // do stuff here that you want
  // to happen once
  done = 1;
}
leave_critical_section();
```

From this you might think that synchronisation is only needed when a value is going to be modified. Unfortunately this isn't so.

```
int *p; // assume this is an initialised
```

```
        // shared variable
if (p != NULL)
{
  if (*p == 0)
  { // do something wonderful
  }
}
```

Here we are testing p to be sure that it is valid before using it. Unfortunately, some other code might come along and modify it (to NULL, for example) before it is dereferenced, causing unexpected behaviour, an outright crash, or an exception.

One idea to attempt a fix might be this:

```
int *p; // assume this is an initialised
        // shared variable
int *mycopy = p;
if (mycopy != NULL)
{
  if (*mycopy == 0)
  {
    // do something wonderful
  }
}
```

However, this can be a disaster as well. The memory referenced by p is part of the shared state, so simply making a copy of the pointer doesn't solve the problem. For example, another thread could deallocate the memory referenced by p or otherwise re-purpose it. In a garbage-collected language the object will not have been deallocated, but if p now references a different object than mycopy you may be relying on (or changing) stale data. In other situations this might be a valid and efficient strategy.

There are likely to be multiple places in the code which modify a particular piece of shared state, so we need to be able to lock them all to make them mutually exclusive. To allow this, synchronisation primitives are almost always objects – mutex objects, critical section objects, etc. The lock/unlock operations become methods, and all the normal issues of creation and life-time management come into play. A synchronisation object is a piece of shared state that is used to arbitrate between threads.

```
mutex m; // assume this is initialised
         // shared state

int *p; // assume this is an
        // initialized shared variable

m.lock();
if (p != NULL)
{
  if (*p == 0)
  {
     // do something wonderful
  }
}
m.unlock();
```

One obvious problem with this is that it becomes easy to forget the unlock operation, or to retain the lock for long periods of time. This is particularly an issue if the lock and unlock operations are enacted indirectly through a higher level piece of code via an interface, or if expensive function calls to other subsystems are made while the lock is held. To mitigate these problems some threading interfaces provide lexically scoped synchronisation. For example, in C#:

```
object  m;
int *p; // assume this is an initialised
        // shared variable
lock (m)
{
  if (p != NULL)
  {
    if (*p == 0)
    {
       // do something wonderful
    }
  }
}
```

Synchronisation primitives are shared resources, and in C++ it is appropriate to apply the resource acquisition through a construction paradigm, as in, for example:

```
class MutexLock
{
public:
  MutexLock(Mutex m) : mMutex(m)
    { mMutex.Lock(); }
  ~MutexLock()
    { mMutex.Unlock(); }
private:
  Mutex mMutex;
};
```

As a project grows in size and more code needs to operate concurrently, a program will come to have multiple synchronisation objects. A new problem arises in code with multiple synchronisation primitives: deadlock. Imagine two threads (#1 and #2), which are using two mutexes (A and B). Now consider what happens in this scenario: Thread #1 acquires mutex A, thread #2 acquires mutex B, thread #1 attempts to acquire mutex B (and blocks until #2 releases it), and finally thread #2 attempts to acquire mutex A (and blocks until #1 releases it). These two threads are now stopped, waiting for the other to release its resource, which of course they cannot do because they are stopped.

Deadlock is relatively simple to be aware of and to avoid in a simple piece of software. It becomes considerably more complex to avoid in larger pieces of software. One solution is to use a single mutex instead of two different ones. Some early threaded operating systems adopted this approach to protect their internal resources. The problem with this approach was, as previously described, that the program can rapidly degenerate to being effectively single threaded. The more hardware threads you have, the more of a loss this is.

An alternative to sharing mutexes widely is to avoid shared state and to use no other systems from within critical sections. In large projects this can become difficult or impossible. Powerful and important design patterns such as delegates and iterators can lead to unexpected situations where deadlock is possible.

The term "thread safety" is often used to describe objects (or systems) that are designed to be used from multiple threads simultaneously. It is often not clear what is meant by an object being thread-safe. One approach is to simply make each of the methods in the object lock a mutex while it executes to ensure that the object's state remains consistent during the call. That does not ensure that the object's state remains consistent between calls. For example, in C++ with STL:

```
vector<int> a; // assume this is shared
int len = a.size();  // assume this is a
                     // synchronised op
int last_value = a[len-1];
    // ... and so is this one
```

This code can fail even though all the operations on a are individually thread-safe. If another thread removes an element from a after the count is retrieved, then this thread will index past the end. If another thread adds an element to a then it won't be the last value that is retrieved.

A naive solution to this problem is to provide a lock/unlock operation as part of the object's interface. The user of the object holds the lock while working with it. Unfortunately, working on objects in isolation is rare – most interesting code uses multiple objects to accomplish something interesting. If more than one of those objects uses this approach you may now find yourself in a potential deadlock situation. One example of this arises when iterating across containers. What happens if another thread changes the container while it is being iterated? The .NET framework's containers, for example, throw an exception if this happens. The code doing the iteration must ensure that this cannot happen. The obvious solution is to lock a mutex for the duration of the iteration, but heavily used or large containers or slow per-element operations can make this prohibitively expensive. Furthermore, if the operation involves calls to other code (such as methods on the objects from the container) then deadlock can become an issue. A common example is moving objects from one container to another where both containers are using this locking strategy.

Amdahl's Law

It is useful to understand Amdahl's Law as it applies to concurrency. It actually applies to the principle of optimisation in general, but here I'll just point out its implication with respect to synchronisation. The time to execute a program is equal to the sum of the time required by the sequential and parallel parts of the program: $T = S + P$. Optimisations to the sequential part can reduce S, and optimisations to the parallel part can reduce P. The obvious optimisation to the parallel part is to increase the number of processors. Twice as many processors might reduce P by 2, an infinite number might reduce it to virtually nil. If the division of work between S and P cannot be changed, however, then the maximum speedup possible by parallel optimisations is 1/S (i.e., P has gone to zero). If the sequential part of the program is 50% of the execution time, adding an infinite number of processors will only double its performance! It should therefore be clear that minimising the amount of time spent in sequential parts of the program (i.e., critical sections or holding mutex locks) is essential to performance scaling.

One important point about the preceding paragraph is whether the split between S and P can be changed. Changing this split is a powerful approach, akin to improving your algorithm instead of your implementation. It can be accomplished in three basic ways. The obvious, although difficult, one is to replace some of your sequential work with parallel equivalents. The second is simply to not do some of the sequential work, which is not always an option but sometimes worth considering. And the third is to do more parallel work. The parallel work is what will scale with increasingly concurrent hardware, so do more of it.

Conclusions

The point of this discussion of threaded programming is not to enumerate all the potential pitfalls of threaded programming. Nor do I claim that there aren't solutions to each of these individual problems, even if they require careful design and implementation and rigorous testing. What I am trying to convey is that there is a minefield of nasty, subtle, intractable problems and that the programming model embodied by the common programming languages does nothing to help the programmer deal with it. These languages were conceived for programming computers that no longer exist.

So what, with respect to concurrency, might a language and compiler take care of that makes life easier for the programmer? Here's a sampling (none of which I'm going to explain), just to give you an idea: structure and field alignment, accounting for cache line size and associativity, accounting for cache read/write policies, using cache prefetch instructions, accounting for automatic hardware prefetching, dealing with speculative hardware execution (particularly stores), leveraging DMA hardware, using a context switch versus spin-lock synchronisation, dealing with variables co-inhabiting cache lines, loading variables into different register sets based on capabilities versus cost of inter-set moves versus usage, dealing with different register sizes, handling capability and precision differences among data types, organising vectors of structures based on algorithmic usage and memory system in-flight transaction capabilities, vector access patterns based on algorithms and hardware load/store/permute capabilities, compare and branch versus bitwise math, branching versus selection, dealing with register load/store alignment restrictions, choosing SoA versus AoS in-memory and in-register organisations, hoisting computations from conditionals to fill pipeline stalls, software pipelining of loops, organising and fusing loops based on register set size, selecting thread affinities, latching shared values for atomicity, object synchronisation, trade-off between instruction and thread-level parallelism, and leveraging hardware messaging and synchronisation capabilities. Any one of these things can result in doubling the performance of your algorithm and, although they may not combine linearly, you typically have to get them all right to get close to maximum performance. And many of them will break your code if you get them wrong. Some of them will simply make your code non-portable and fragile.

There are alternative categories of languages, and some of these offer programming models with strong advantages in concurrent programming. Functional languages (e.g., Haskell, Lisp, APL, ML, Erlang) offer some powerful advantages, but most bring with them various disadvantages and none are mainstream. Array programming languages (e.g., APL, MATLAB) are powerfully expressive of data parallel concepts but have limitations and haven't reached their full potential in terms of optimisation – and they typically eschew the object-oriented paradigms that are now deeply (and rightfully) entrenched in modern software development. Less traditional programming models and languages exist that embody powerful concurrent concepts such as message-passing objects networks (e.g., Microsoft's Robotics Studio), but they are far from standard, are not portable, and typically cannot be integrated into existing environments. Highly specialised languages and tools such as StreamIt, RapidMind, and Ct also suffer from the same issues. A few very successful specialised languages exist, particularly in the domain of graphics: HLSL, GLSL, and Cg have given graphics software and hardware developers tremendous power and flexibility. They have integrated well into existing software systems, and being part of the C language family makes them readily accessible to the C/C++/Java/C# communities. The wide adoption of these shading languages gives some indication and hope for what is possible. All of these alternatives point in directions in which we can take our programming languages and models in the future.

Suggested reading

Herb Sutter's talk "Machine Architecture: Things Your Programming Language Never Told You" at NWCPP on September 19, 2007 (video and slides):
`http://video.google.com/videoplay?docid=-4714369049736584770`
`http://www.nwcpp.org/Meetings/2007/09.html`

Michael D. McCool, "Achieving High Performance by Targeting Multiple Hardware Mechanisms for Parallelism", **;login:**, April 2008. Transactional memory:
`http://research.microsoft.com/~simonpj/papers/stm /index.htm`

Language taxonomy:
`http://channel9.msdn.com/Showpost.aspx?postid=326762`

Nested data parallelism:
`http://research.microsoft.com/~simonpj/papers/ndp/NdpSlides.pdf`

Erlang:
`http://www.sics.se/~joe/talks/ll2_2002.pdf`

P. Grogono and B. Shearing, "A Modular Programming Language Based on Message Passing":
`http://users.encs.concordia.ca/~grogono/Erasmus/E01.pdf`

This is an adjusted version of a paper originally published in ***;login: The USENIX Magazine,*** *vol. 33, no. 3 (Berkeley, CA: USENIX Association, 2008). Reprinted by kind permission.*

Apache2 Pocket Reference
Andrew Ford
O'Reilly Media
ISBN: 978-0-596-51888-2
208pp.
£ 8.99
Published: 24th October 2008

reviewed by Andy Thomas

Somewhat thicker (and heavier!) than most other titles in the O'Reilly Pocket Reference series, this book certainly has come a long way from the single Desktop Reference card by the same author that used to accompany the second edition of Ben and Peter Laurie's Apache: The Definitive Guide as an insert inside the front cover.

Organised into eleven chapters, the book is logically laid out and covers the main tasks an Apache server administrator would be faced with. Rather simply filling the book with an alphabetically-ordered and tedious list of configuration directives, the author has chosen to group these into distinct topics and devotes a chapter to each – URL mapping, access control, filtering and so on and it is easy to find all the directives related to a given task. Each chapter is then broken down into sub-sections each containing a concise description of that sub-topic with additional information where necessary. This makes the pocket book very readable and interesting even – I found myself reading it on my daily commute, especially the section on filters (where I found some new features that crept into Apache 2.2 that I was completely unaware of).

Topics covered in chapter 1 include basic configuration and starting and stopping the apache server while following chapters describe the server environment, access controls, URL mapping, the SSL/TLS subsystem and logging. Less common aspects of Apache operation such as document meta-information, content handlers, caching, proxying and filtering are given equal weight as the popular topics - nothing is skimped or glossed over. One thing this book does not cover is installation; this is probably well outside the remit of this title and even full-sized books generally avoid this area as the documentation accompanying the Apache source covers this adequately while installing a pre-built Apache package will usually be just the same as installing any other package for that operating system version. The assumption is that the reader has access to an already-installed and usable apache server.

Rounding off the book is a comprehensive index that includes every Apache 2.x configuration directive followed by some useful appendices containing some additional HTTP reference material; I was surprised to find a detailed description of the C language `strftime()` format in one of the appendices – not the sort of information that would concern many apache administrators but it does add interest to what could have been a very sparse pocket booklet and gives it a feeling of completeness.

Summing up, this pocket reference lives up to the usual high standard we have come to expect from O'Reilly and this Apache old-timer was not at all disappointed. For those well-versed in earlier versions of Apache such as 1.3, there is enough material here even to make the jump to Apache 2.x – this pocket book together with an earlier edition of the full-size Apache reference are in most cases all you would need to cover all versions of Apache. It does exactly what it says on the tin (i.e. the rear cover) and whilst it is neither a getting-started tutorial nor a definitive reference tome, experienced Apache administrators will find this a very useful addition to their bookshelf (or more likely, their backpack). And at the amazingly low asking price, it's a no-brainer!

Version Control with Subversion

C Pilato, Ben Collins-Sussman and Brian Fitzpatrick

O'Reilly Media
ISBN: 978-0-596-51033-6
430pp.
£ 24.99
Published: 2nd October 2008

reviewed by Paul Waring

I use Subversion on a daily basis for several projects, though I've always felt that I'm not getting the most out of it and there are plenty of things which I'm still not sure how to do, so a book claiming to cover every aspect of using Subversion is probably just what I need.

The book starts off with a useful introduction to the fundamental concepts of version control and the principles behind how Subversion handles revisions. In the second chapter I was pleased to see an introduction to the internal help provided by the Subversion command line client. Internal help is something which is often overlooked in technical books, but I find myself using it all the time. If you prefer a dead tree version, there's a copy of all the help in a later chapter as well.

Other chapters give comprehensive coverage of all the main version control tasks, such as creating branches, tagging releases and merging branches back into the main trunk. System administrators get two chapters on creating and maintaining repositories – a useful addition given that often the users and the administrators are one and the same in many small projects. Should you want to embed Subversion in other pieces of software or learn how to use the various APIs, there's a chapter dedicated to this too.

For those of you who are more familiar with Subversion's predecessor, CVS, there is an appendix which covers most of the major differences, both in syntax and design. By the time I started using version control Subversion was the de facto choice for new projects, so I've never spent much time with CVS, but the appendix looks sufficiently comprehensive to cover most of the common areas.

The only things which are missing from this book is an explanation of why you might want to use Subversion as opposed to any other version control system (particularly Git), although one could presume that if you have picked up a book on this topic you're already decided that Subversion is the tool for you. The lack of any mention of clients other than the standard command line option was also somewhat disappointing, especially given that there are plenty of useful clients and plugins out there (e.g. Subclipse for the popular Eclipse IDE) which could easily have been introduced briefly in an appendix.

Overall, this book is a worthwhile purchase if you use Subversion on a regular basis, and particularly so if you work on projects with several branches. This text also covers Subversion 1.5, so it's about as up to date as you can get, although anyone using Debian Etch will still be on 1.4 by default. Finally, the text has been released under a free licence, so you really can try before you buy – and if you're not happy with any aspect of it, you can always rewrite a few chapters and release your own version.

Sustainable Energy – Without the Hot Air

David J C MacKay
UIT
ISBN: 978-0-9544529-3-3
384pp.
£ 19.99
Published: 1st December 2008

reviewed by Paul Waring

This book might seem a little off the beaten track, compared with the usual technical books on review in the newsletter. Sustainable energy is a hot topic at the moment though, especially for the IT industry where huge data centres suck in enormous amounts of power just to keep machines cool, bringing in both environmental and financial problems, and you can't turn on the news today without hearing about climate change.

The author begins by lambasting most of the conventional reports on sustainable energy, with quotes along the lines of "if everyone does a little, we'll achieve only a little". One gets the feeling that this text isn't your standard run of the mill book on climate change, a view which is confirmed the further on you read.

The first few chapters of the book present a broad overview of energy use and requirements, particularly focusing on Britain's share of world energy consumption. Colourful graphs and figures make the information easy to absorb, and the technical detail is reserved for the notes at the end of each chapter in most cases. Further into the book, topics such as the energy required to produce most of our consumables are tackled, with some particularly interesting figures, such as the amount of fossil fuels required to produce a personal computer (250kg – excluding the transportation costs). Transport geeks will enjoy the chapters dedicated to this area of energy use, with hybrid, electric and hydrogen vehicles all given a degree of coverage. All areas of renewable energy are covered too, from solar to wind, followed by a fairly balanced discussion of nuclear's role.

There are lots of references throughout the text backing up the author's points – this is certainly a well researched book. Technical detail abounds, including plenty of photographs, so if you are interested in all things mechanical there's plenty of material to get your teeth into. Part III in particular is entirely devoted to the mathematics and physics of how various sources of sustainable energy work, though I must admit to getting lost at this point as my mechanics knowledge is rusty to say the least.

If you want a refreshing outlook on sustainable energy and can handle plenty of figures and calculations, this book makes interesting reading. You do need to know a little bit about the subject of energy generation in order to get the full benefit, and you have to overlook the author's impression that he is always right, but if you can cope with that it's definitely worthwhile taking a look at this text if you're interested in how energy can be generated in a sustainable manner.

Desktop GIS: Mapping the Planet with Open Source Tools

Gary Sherman

Pragmatic Bookshelf
ISBN: 978-1-934356-06-7
368pp.
£ 21.99
Published: 28th October 2008

reviewed by Gavin Inglis

Maps fascinate people, and their digital versions are becoming ubiquitous. Whether through a portable GPS unit, mobile phone, web browser or vehicle navigation system, we now expect new maps wherever and whenever we need them. And why not? With Google Earth, we have the ability to fly from space to a photographic rendering of anywhere in the world, and just possibly, see our car parked there on the street.

A richer, but less accessible world is that of the desktop geographic information system (GIS). This book leads beginners through the GIS landscape and the open source tools available for its exploration.

From the first page we are introduced to Harrison, a keen hillwalker who wants to make something of his geotagged bird sightings. Harrison provides a route through GIS concepts, as he investigates his theory that each sighting is within a certain distance of a lake. This is an excellent approach which makes it clear why you would choose a desktop application rather than a web site – and why you might not.

Thus begins an engaging book which feels rather odd. This is down to its breadth: one chapter describing the point-and-click operation of a GIS application will be followed by a dirty discussion of spatial databases, then a slice of theory about projections and coordinate systems.

An inexperienced mapper may feel adrift early on, with only a vague understanding of what constitutes spatial data. However there is fun to be had in mapping the world and its cities using the downloaded data which accompanies the book. Chapter six does briefly fill in the missing pieces and adds some worthwhile thoughts on organising and managing geodata. The section on how to digitise your own maps is practical and useful.

There is a clear emphasis throughout that different levels of user have different mapping needs (amusingly, their initials spell CIA) and it is also established that rarely will one application meet all your needs. Throughout there is a clear Linux focus, this being a book specifically on open source; but neither MacOS X nor Windows users are excluded from the bulk of the examples.

The first GIS used is uDig. This seems clunky but the text soon moves on to the more polished Quantum GIS. Only later does it scale to the mighty heights of GRASS, with a lengthy appendix devoted to the system. There is some general discussion of how to keep software up to date and obtain support through mailing lists; this is reasonable but generic stuff.

As with most books of this type, there is an American bias to the geodata. Blame freely available government data in the USA versus the Ordnance Survey's licensing terms; however, it's surprising to find no mention of the gutsy OpenStreetMap project in a book about open source and geodata.

Desktop GIS is an enjoyable stroll for the newcomer to GIS, with many colour illustrations throughout.

iPhone Forensics

Jonathan Zdziarski
O'Reilly Media
ISBN: 978-0-596-15358-8
138pp.
£ 24.99
Published: 17th September 2008

reviewed by Graham Lee

The iPhone has been a runaway success, propelling Apple to the position of being the #2 mobile phone company in only a couple of years. Part of that success has been due to its popularity with enterprises, and it's likely to be the corporate I.T. administrator or security officer who gets the most out of this short book.

The technical discussion of recovering data from a device – including bypassing any passcode protection – is accurate and well-presented for both Mac and Windows users, but ultimately is only as engaging as a recipe to follow can be. It also would be unlikely to present any surprises to a computer security or forensics professional; especially one who already has an understanding of UNIX systems and their operation (the iPhone being, after all, a small UNIX computer).

The real interest in this book comes from the discussions of exactly what information is there to be taken; for instance the iPhone keeps a series of screenshots of the applications run by the user. For people using or administering iPhones this will provide an enlightening and perhaps worrying exposn the perils of losing their beloved device (beyond of course the financial cost), although the book is too light in this respect to supply interesting context and explanations of these issues. A UNIX-savvy reader interested in those aspects of security would be better serviced by researching the general computer security field first, perhaps from O'Reilly's own "Practical UNIX Security" book. phone.

The Art of Capacity Planning

John Allspaw
O'Reilly Media, Inc.
ISBN: 978-0-596-51857-8
152pp.
£ 27.99
Published: 15th September 2008

reviewed by Raza Rizvi

I had some qualms about reviewing (at short notice) a book on capacity planning. They have always tended to be full of queueing theory and complicated graphs, hardly the stuff for quick absorption.

But this book really does break that mould and provide practical, proven information for administrators of LAMP (LINUX, Apache, MySQL, PHP|PERL) deployments. Yes there are a lot of graphs but they all appear as actual examples from the working systems the author has been and is currently responsible for, rather than being simulations or predictions based on theoretical data. The book is exceptionally well illustrated which further adds to the lifting of the topic and greatly improves the readability of the subject matter.

After setting out why one needs to measure the current capacity of systems and then the high-level of how one might go about measuring the metrics (including a very apt example showing the impact of rationalising a server farm when upgrading the hardware) the author spends a 40 page chapter on the actual topics that need consideration, taking in turn traffic, logs, databases, I/O, storage, and traffic. These are all described in the context of the author's day job (running services for Flickr, the photo storage web site). Somewhat surprisingly there isn't a specific section on CPU issues though it is touched on in each of the sub-sections. The author sensibly includes material on front-end cache deployment but it was also very interesting to read briefly on the specific issues caused by deployment of an API something any respecting Web 2.0 site needs to build a partner community.

At this point we know what we must measure and why, and have some idea of how to do the mechanics of it, but how do we actually use all this information? Chapter four is where the Excel graphing techniques get used to predict the future, and where we realise that capacity planning is really a full-time activity if one is to avoid knee-jerk reactions to resource shortages.

The book rounds off with a few pages on automated deployments, which seem somewhat out of place, and then a useful summary appendix on quick strategies of how to deal with a sudden increase in demand. The final appendix has, of course, a list of software applications for the monitoring, measurement, and deployment of systems. The whole goal of the book is to help you delay expenditure so that you deploy equipment at the right time for your system. It does a very good job of meeting this goal though as it does contain information on build and deployment matters, though my personal preference would have been to have further information on how one would scale such systems (though there is at least one other O'Reilly publication covering that area). I would also have liked to have seen more information on capacity planning of the network infrastructure (though load balancing is mentioned in passing) rather than the sole focus on single systems and server farms (you are by the way expected to know how to aggregate your information to get a holistic view).

I do sound a bit critical but if you are responsible over the next two years for making sure your web based users can get to your hosted services within a decent response time, you would be foolish to ignore the information this slim but useful books provides.

Subject to Change

Peter Merholz, Brandon Shauer, David Verba and Todd Wilkins

O'Reilly Media
ISBN: 978-0-596-51683-3
186pp.
£ 15.50
Published: 15th April 2008

reviewed by Roger Whittaker

Attentive readers of this Newsletter will know that I like to review some of the books we receive that are marginally "off-topic", but still relevant to the concerns of our community. In the case of O'Reilly books, the fact that Tim O'Reilly thinks that the subject matter is interesting and relevant is a fairly good recommendation for me in itself.

This book is written by the people behind Adaptive Path, which is a US company that specialises in helping others to think about how to design products and services intelligently:
`http://www.adaptivepath.com/`

Adaptive Path's "big idea" is to understand the "user experience" that customers take from a product. One of the examples the book uses is the first Kodak camera that used roll film and how the "user experience" of photography was transformed by that innovation. There are various other examples in the book both of failed and successful products together with an analysis of what it was about the user experience that contributed to success or failure.

So the book is about good and bad design, but not design purely in the sense that an artist might understand it. One of the failing examples that is cited is Apple's G4 Cube computer, which could have won (and probably did) multiple awards for its looks, but did not work in the market. Another interesting example is Boeing's proposed "Sonic Cruiser" which constituted a radical re-think of how aeroplane design could be done. One of the effects of the plane's radical shape was that passengers had to sit in a large open cabin with few visible windows. The reactions that Boeing observed from ordinary people just sitting in mockups of the cabin on the ground was one of the reasons why the project was cancelled.

There are software-related examples in the book. Some of these are discussions of user interfaces in desktop applications and web sites, but equally interesting are the references to the entire user experience with the success of Flikr cited as one example. In the case of Flikr, as with other examples in the book, the authors identify a strategy that goes beyond simply having a well designed user interface. Previous web photo sites had simply adopted the “album” metaphor as the basic way of collecting photos together. Flikr’s use of tagging, photo sets, groups and maps broadened the usefulness of a photo site dramatically, and provided an experience that users could understand and enjoy.

The authors seem to have thought long and hard about the combination of design and user experience. Given that they make their living by consulting on these matters, that’s not surprising. This book is an interesting and thought-provoking introduction to their ideas.

Pragmatic Thinking and Learning: Refactor Your Wetware

Andy Hunt
Pragmatic Bookshelf
ISBN: 978-1-934356-05-0
279pp.
£ 21.99
Published: 28th October 2008

reviewed by Roger Whittaker

This is a self-help book, written specifically (or at least mainly) for programmers and geeks, by one of the authors of the original “Pragmatic Programmer” book and of the Ruby ”pickaxe” book.

On looking at the book for the first time, I was slightly put off by the subtitle: *Refactor Your Wetware*, and was expecting too many brain/computer comparisons and metaphors. There are quite a few of these, but only when they are constructive and useful.

Some years ago, I was given a book called “Drawing on the Right Side of the Brain” by Betty Edwards. This book’s author shows you that despite what you might think, you can draw. She does this by giving you exercises that force your brain out of its rational linear “**L**-mode” into what she calls $\mathcal{R}$-mode: most particularly by copying pictures that are upside down. I was amazed to find that it worked even for me.

Andy Hunt takes up these concepts of **L**-mode and $\mathcal{R}$-mode and makes them one of the core ideas of his book. **L**-mode is linear, rational, linguistic, left-brain thinking. $\mathcal{R}$-mode is “rich mode”, non-verbal, intuitive, perceptual thinking. While one might expect that programmers would have little need of the latter type of thinking, the author believes that the ability to switch between **L**-mode and $\mathcal{R}$-mode provides you with powerful problem solving capabilities that you will not have if you spend all your time in **L**-mode.

There are many other suggestions and insights in this book. The author is deeply scathing about what he calls “sheep-dip” training methods, insisting that real learning only takes place when the student has a definite (but limited) aim and the necessary motivation. He uses the “Dreyfus model” of skill acquisition to show that programming is not very different from other fields (nurses and pilots are mentioned as comparisons). There is the same progression through *Novice*, *Advanced Beginner*, *Competent*, and *Proficient* to *Expert*, and in each profession the relative numbers of practitioners in each category follows a similar profile, with the majority in the second. He believes that organisations that fail to understand that individuals at different stages need to be handled and managed in different ways can be expected to lose their best employees, and are unlikely to be able to make best use of any of their staff.

Also covered are note taking, mind maps, personality types, discovering how your primary learning type and productive and unproductive ways of “wasting time”.

If this all sounds slightly wishy-washy – the kind of book you wouldn’t bother to open: think again. It is written with programming as the context and with people with our types of personalities in mind. Most people will find something here that is useful to them.

Programming Collective Intelligence

Toby Segaran
O'Reilly Media
ISBN: 978-0-596-52932-1
360pp.
£ 24.99
Published: 16th August 2007

reviewed by Roger Whittaker

The subtitle of this book is *Building Smart Web 2.0 Applications*, but don't let that put you off. This is not about AJAX or the user end of things: it is about the algorithms that make clever stuff on the Web possible at all.

A wide variety of material is covered here, and there are some examples of using some of the web service APIs provided by sites such as Kayak (flight search), Hot or Not (photos and dating) and eBay (auctions). But this type of material does not constitute the bulk of the book, which is about how to actually do the clever stuff yourself.

For instance, there is code for collaborative filtering (the kind of thing that gives a customer recommendations based on his own and others' previous actions). There are examples of grouping and clustering (for example using analysis of content to find similar and dissimilar blogs). There is code for spam filtering, showing how it works in principle, and not shying away from explaining Bayes' Theorem. There are examples of optimisation techniques, again with the underlying mathematics explained. The book ends with a chapter on genetic programming.

Some of the clever stuff is very clever indeed, or at least seemed so to me. The type of problems addressed are at least close to real problems that people want and need to solve.

Fully to understand and apply all the concepts in this book would require some hard work, but even a superficial reading gives one a good idea of the types of methods that can be used.

All the code examples in the book are written in Python, and hence are accessible and readable. Although some of the subject matter is complex, the book is written in very clear language and is very approachable.

Contributors

Andrew Back holds the position of Open Source Strategist at Osmosoft, BTs open source innovation arm, and leads BTs engagement with the FOSSBazaar workgroup of the Linux Foundation.

Andrew Brownsword is Chief Architect at Electronic Arts BlackBox in Vancouver, Canada. He has been with the company since 1990 and has a BSc in Computing Science from the University of British Columbia.

Jason Dusek is a systems architect with a self-funded startup in the Bay Area. His technical interests include distributed computing, functional programming, and serialization.

Gavin Inglis works in Technical Infrastructure for the EDINA National Data Centre. His interests include punk music, Egyptian mythology and complicated diagrams.

Graham Lee is the Senior Macintosh software engineer at Sophos, a computer security company based in Abingdon. He is currently occupied writing anti-virus software, and deleting everything from his mobile phone.

Jane Morrison is Company Secretary and Administrator for UKUUG, and manages the UKUUG office at the Manor House in Buntingford. She has been involved with UKUUG administration since 1987. In addition to UKUUG, Jane is Company Secretary for a trade association (Fibreoptic Industry Association) that she also runs from the Manor House office.

Raza Rizvi has just finished adding a load more network capacity to the world's largest Internet Exchange Point.

Peter H Salus has been (inter alia) the Executive Director of the USENIX Association and Vice President of the Free Software Foundation. He is the author of *A Quarter Century of Unix* (1994) and other books.

Andy Thomas is a UNIX/Linux systems and network administrator working for both Dijit New Media and Imperial College London, as well as freelance through two consultancies. Having started with Linux when it first appeared in the early 1990's, he now enjoys working daily with Solaris, Tru64 and the various flavours of BSD UNIX as well. Married with a teenage son, he lives in north London with a cat and a lot of computers for company.

Paul Waring is chairman of UKUUG and currently trying to finish writing an MPhil thesis. He is also responsible for organising the UKUUG Spring conference in 2009.

David Wagner is a professor in the computer science division at the University of California at Berkeley. He studies computer security, cryptography, and electronic voting.

Contacts

Paul Waring
UKUUG Chairman
Manchester

John M Collins
Council member
Welwyn Garden City

Phil Hands
Council member
London

Holger Kraus
Council member
Leicester

Niall Mansfield
Council member
Cambridge

John Pinner
Council member
Sutton Coldfield

Howard Thomson
Treasurer; Council member
Ashford, Middlesex

Jane Morrison
UKUUG Secretariat
PO Box 37
Buntingford
Herts
SG9 9UQ
Tel: 01763 273475
Fax: 01763 273255
`office@ukuug.org`

Sunil Das
UKUUG Liaison Officer
Suffolk

Roger Whittaker
Newsletter Editor
London

Alain Williams
UKUUG System Administrator
Watford

Sam Smith
Events and Website
Manchester