25 Years - Part 1

by Tony Tebby

25 years have gone by since the launch of the QL.

More than one person has suggested that I should write a little something to celebrate the occasion. This seems to me a bit strange. I am well aware that the launch of the QL was not as disastrous as the sinking of the Titanic or the 2004 Indian Ocean Tsunami. But celebrate it? Strange!

So rather than describing the development of Domesdos (the earlier name for QDOS), I thought I would look back at all the progress that has been made in system software, and, in particular personal computer systems over the past 25 years.

To appreciate the progress made, it is necessary to wind back the clock to 1983, the year that the ZX83 was not launched. Much has already been written about this rudderless project that started out as a development of a portable version of the Spectrum (ZX82) and ended up as a quantum leap into the void, so I will not repeat, confirm or deny it here. I will just try to give the background.

Then I shall describe where Domesdos went after the ill timed launch of the QL which was just a black shadow of the planned ZX83.

Finally, I shall give my own view of the progress made in workstation operating systems since 1984.

In the beginning

The QL started as the ZX83, a development of the ZX82, The impetus to create a new operating system for it came from the decision to replace the trusty Z80A microprocessor by a cut down MC68000. This "sexy" 32 bit microprocessor obviously needed something more impressive than the old spectrum software and Sir Clive decreed that it should have a "version of Unix that works". Later on, other requirements were added.

Depending on your point of view, the resulting system was a major breakthrough and outstanding success or it was a totally disastrous deviant.

For those who were not around at the time the idea of a "version of Unix that works" must seem bizarre. Surely, all versions of Unix worked, didn't they?

Unix in 1983

Well not quite, Unix "sort-of" worked provided you did not try to use it. At the time, UNICS (and later Unix) had been around for about 13 years, during which it had become legendary for its exceptional slowness and quirkiness. Unix had three great attractions for academics – it was free and so did not require a budget – it was portable and so could be implemented on new platforms in not much more time than it would take to re-write it from scratch, keeping thousands of students off the streets – it was quirky and so using it was a real challenge. The slowness was not an attraction, but neither was it a serious problem in the academic world.

Unix however had offered a glimpse of a different idea of an operating system. It was not so much what it did, but what it might have been able to do in a world with unlimited computing power, and if it had not been Unix. The problems were that computing power was not, is not and is not likely to become unlimited, and Unix was Unix.

The ill-fated XENIX system should give an idea of the slowness of Unix. When this was launched (just after the QL) on an IBM PC XT platform (about 4 times faster than a QL with 10 times as much working memory) one journalist coined the phase "a brain dead version of Unix" – an epithet that stuck. But it was not "a brain dead version of Unix", it was a real Unix running on a desktop computer which lacked the power of a $100,000 VAX (the favourite Unix platform of the period). A 2009 personal computer is several thousand times faster, with several hundred times more memory, than a early 1980s VAX, so, although Unix is still chronically slow, this is not now as obvious.

The slowness of Unix was, however, not the only problem: it also suffered from an operating system interface that was, as the French would put it, *bordélique* and it had a well deserved reputation for chronic instability.

A Unixly chaotic operating system interface

The chaotic operating system interface was a direct result of a design choice made by the developers – minimisation of the number of operating system functions.

This minimisation of the number of operating system functions had three effects.

The initial effect was that the real world had to be twisted to fit into the Unix minimalist world. For example, the I/O system was based on the paper tape reader / punch model: when console I/O and a data storage system were added, these had to be twisted to fit the paper tape model. While this had the advantage of facilitating the implementation of scripts (which are still fundamental to the operation of a Unix system), it had very serious consequences both for the primary use of the filing system (data storage and retrieval) and the interactive use of the console. So the whole I/O system was all turned onto its head and everything was treated as a file, even if it was an interactive device. This made it even more irrational.

The second effect was that anything other than the most basic functions had to be added on. This was great, everyone could have their own flavour of extended Unix – hundreds of different, incompatible commercial and university versions and thousands of private versions. Is this really a good idea? The developers and the academic establishment thought so – natural selection of the best version was obviously better that allowing a development group to impose an arbitrary choice. I did not find this such an obviously good idea.

The third effect was that functions that could be implemented simply and efficiently by a single operating system call were pushed out into C libraries where complex and inefficient routines had to make large numbers of "primitive" OS calls for each higher level function. Later this complex and inefficient approach was elevated to a virtue. Minimalist operating systems interfaces became "a good thing".

A Unixly unstable operating system

The instability of Unix was another concern. Just before the QL came into the world, Sun Microsystems was set up to exploit the nascent interest in Unix by making workstations using the most powerful microprocessors then available (3M machines: 1 MIP, 1 Megabyte, 1 Megapixel). These were sufficiently powerful for their SunOS version of Unix to seem comatose rather than brain dead. A great innovation in SunOS was the fast boot process to recover from system crashes quickly. Paradoxically, as one reviewer pointed out at the time, this made SunOS seem even less reliable than standard Unix: an ordinary version of Unix taking five minutes to boot could only crash 12 times an hour, whereas SunOS booting in less than a minute could crash 60 times an hour. Even when Unix was not crashing, the reliability, in terms having a system that did what you wanted to do rather than what it wanted to do, was not particularly good – to quote Dave Mankins[1] who slightly misquoted Johnson: "making Unix run securely means forcing it to do unnatural acts. It's like the dancing dog at a circus, but not as funny – especially when it is your files that are being eaten by the dog", evoking the propensity of Unix for destroying your data even in the absence of deliberate attacks.

Sir Clive's "Unix that works"

Sir Clive's idea of a "version of Unix that works" was, therefore, really rather revolutionary.

I interpreted "that works" as meaning that it should be efficient, reliable and with a rational operating system interface to remove the three main Unix problems. Some people have always objected that a "version of Unix that works" should have been efficient, reliable and with a standard Unix operating system interface. However, I took the view that the standard Unix operating system interface was not only a serious problem in itself, but also a barrier to making the system efficient and reliable.

There never was even an outline specification for the new operating system for the ZX83, Sir Clive did not work that way – he had an amazing capacity for delegation, for letting his "chosen" to get on with the job and for accepting the consequences himself if it all went wrong. So what were to be the salient points of a new operating system for the ZX83?

1 Unix Hater's Handbook http://www.simson.net/ref/ugh.pdf

Single user with pre-emptive time-sharing between tasks.

Unix was (and still is) a multi-user system. Although it was possible for a single user to pretend that he was several users, this provided only limited multitasking with negligible interaction between tasks: "native" pre-emptive multitasking allowing operations on shared data structures was not to become available for many years.

At a distance, it is difficult to imagine why pre-emptive time-sharing between tasks was thought to be a good idea, but there was a group at Sinclair that was working on parallel systems, and they thought it would be useful to be able to have a hundred programs running simultaneously, sharing memory, for simulating a massively parallel processing system. So one hundred application programs running simultaneously became the target.

Maybe it was not such a bad idea. After all, this was something that you could not do using a $10,000 Apple Lisa, a $20,000 Sun workstation or a $100,000 VAX[2] and the typical Sinclair customer was an enthusiast.

From a more mundane point of view, this did allow little features such as clocks to be implemented without requiring the "main application" to continuously call a function to pass control to another task. It did allow background communications tasks to receive and transmit data without the "main application" being affected. But, in itself, it could not directly handle "switching" between applications. Why not?

Because, if you had two or more applications running simultaneously they could both be writing to the display at the same time: which would you see? If a user typed something, which application would get the keystrokes? There were two reasonable solutions to this problem: having only one application "active" at a time (single task switching) or windowing – it being much easier to do both as in the early Apple Mac.

Monospaced text I/O on bit mapped display.

This was to become, possibly, the most disappointing feature of the QL software. There were reasons for this being the obvious choice.

1. The display handling was all carried out by a device driver that could be (and was, many times) replaced, even while the machine was running. A graphical user interface (GUI), was, therefore, not a baseline requirement.
2. The conventional schemas for implementing these GUIs all placed a very heavy interface burden on applications, making it very difficult to write software for the machine. Moreover, Sinclair had contracted Psion to port an office suite designed for the monospaced text PCDOS environment.
3. Existing systems with GUIs were, to say the least, extremely limited and slow even with hardware much more powerful than the QL.
4. The Sinclair computer range had, from the start been a "instant programming machine". This is totally anti-GUI.
5. Time.

Device independent filing system.

Not the Unix "everything is a file" syndrome, but allowing, for example, data to be read from a storage device as if it were coming from a console (a line at a time) and files to be read or written (handling file length, properties and end of file) over a communications port as if you were accessing a storage device, while providing a clear separation of distinct input and output functions.

Real-time hardware management.

The QL hardware was designed using well established Sinclair principles: do not do it in hardware if you can do it in software. There was no question of trading off cost against performance. The software had to respond to a general interrupt, identify the interrupt source and transfer information to or from a device driver or application in a handful of instructions – unlike any other PC or workstation, there was no hardware buffering, FIFO or DMA.

2 Five years later, contemporary reports on the spread of the Morris Worm estimated that 1988 VAXs running BSD Unix were unable to handle more than 20 active processes.

Like most computer users and software developers at the time I had suffered from conventional operating systems I could understand the potential of Unix but I did not understand why it was so exceptionally bad. It was supposed to be simple, which should make it efficient, but its slowness was already legendary. It was supposed to be simple, which should have made it reliable, but it wasn't.

So, there I was, I had a few months to turn Unix into a working system – a task that had defeated thousands of developers who had been working on it for years – or had it?

The answer did not come in a flash, but when I finally got there, the answer was simple. The Unix that we knew had not been designed to work and the developers had not been trying to make it work. This Unix was the result of the application of academic theories of no relevance to the real world.

The Why is easier to explain than the How.

Why was Unix so bad?

That is very simple. Unix came from the academic world. Nobody's job was on the line. Whether it worked or not ceased to be an issue after the first version printed "Hello world". What was important was the application of "modern operating systems" theories. At the time, the operating systems provided by the major computer manufacturers had no theoretical basis – they were just cobbled together to meet ad hoc requirements. On the other hand, the developers of these systems did not have cosy jobs for life in research, so that if these systems did not work, their developers would soon be looking for new jobs. This was a fairly powerful incentive in the days before employment protection.

As a scientist, (I have a certificate to say that I have a degree in physics, for what it's worth) a sound theoretical basis is something that I wholeheartedly welcome, but, being a pragmatist, I feel that having a working system is far more important.

How did Unix get to be so bad?

What intrigued me in 1983, while the Unix story was still unfolding, was that the main platform for Unix was the VAX series of computers. It was all too obvious that the operating system running on these VAXes just could not be the same as the UNICS that printed out "Hello world" from a PDP7 in 1970 – The PDP7 was about as powerful as a ZX80. There was clearly something *louche* in this story that no-one was owning up to The Unix we knew could never have even given an indication of working on a PDP7. I managed to get hold of some documents about the development of Unix, written by Dennis Richie, amongst others, which shed some light on this anomaly. Although this is not the way it was put in those documents, it seems that Unix had been the victim of the upsurge in computer science that occurred over a very short period from 1962 to 1965. At that time, the leading lights in computer science were unanimous in their view of the future of computing: the near future (the 1970s) of computing was enormous time-sharing systems serving thousands of users. There was no possibility of building a single computer powerful enough to serve thousands of users – the natural consequence was that the future was in "symmetric multiprocessing" with massive arrays or networks of thousands of processors working in parallel.

The two problems to be addressed in this symmetric multiprocessing future were managing the conflicts between different processors for shared resources and distributing the users' workloads between the processors. The solutions to both problems had to be transparent and equitable (for more details, see "Modern Operating Systems", 1990 by Prof A. Tanenbaum, who still believed in these theories 25 years later).

The multiprocessing model

There was a unified model of software execution (the "multiprocessing" model) underlying the theories that were developed to meet this challenge. I am not sure that this was ever explicitly defined, I think it was just a private consensus. This model treated all hardware configurations, single processor computers, multiple processor machines with shared memory, loosely coupled networks of computer and any other imaginable configuration as being equivalent, presenting the same problems and accepting the same solutions. Above all, it elevated "symmetry" to the status of an inviolable law.

This was an approach that is very attractive: it provided solutions that were generalised and it promised a sound, universally applicable theoretical basis for operating system development.

Unfortunately, the real world was and is rather different. Problems of sharing memory that can occur on tightly coupled processor systems cannot occur on a distributed system with independent memory for each processor. Likewise, solutions that make use of shared memory cannot be used if the processors do not share memory.

I will not bore you with the almost endless list of real differences between different real hardware configurations that mean that, even where common problems can be identified, common solutions will be at best suboptimal and frequently totally unworkable on real systems. As a result, single processor multitasking systems were considered to be just a poor emulation of multiprocessor systems and so were required to emulate all the problems of these multiprocessor systems, even though these problems were not intrinsic to single processor systems.

The scenario was surreal. It was as if an eminent group of transport engineers had set out the design rules for future transport. This would mean that cars, lorries and trains could not have wheels, because they are useless on water. Ships could not have underwater propellers because roads are solid. Aircraft would have to fly at ground level, because a ship or train taking off could be quite dangerous. The generalised solution is, of course a hovercraft – a surface (land or sea) following airborne craft. In future, all transport would by by hovercraft. This metaphor is not an exaggeration, quite the opposite. A hovercraft at least has some useful applications, but I could not see any evidence that this mass of 1960s computer science theories could ever be applicable to any real system.

The surreal bit was not the theories themselves, but that, instead of being laughed into oblivion, they were actually taken seriously and were still being taught as inviolable gospel truth to computer science students. (25 years later this is still happening!)

Symmetrical multiprocessing

The unshakeable, immutable, unconditional, belief in symmetry had three main sources.

The first was a naïve idea of fairness that arose from the only form of computing envisaged: multi user time sharing systems. Symmetry should provide total fairness, but in practice it does not. Even the aim is not very sensible: surely it is better that all tasks are completed quickly if unfairly rather than slowly and equitably.

The second was a naïve idea of simplicity. It was felt that it was simpler to have all processors being equal, and, therefore, all programmed the same way. It is difficult to imagine that anyone could be so naïve as to believe that it would be simpler to have many processors each deciding which tasks were to be executed and fighting over resources rather than having a dedicated controller. But they did.

The third was a simplistic idea of reliability. The model was that of an ant colony. Individual ants can be killed but the colony carries on regardless. The big fallacy is thinking that the loss of a processor does not matter – it does: the data being processed is lost and if it is your bank account that is lost, you would think it mattered. The controller in a asymmetrical system is usually presented as a weak point. It is not. A controller failure will not lose data and it can provide recovery from individual processor failure with a simplicity and reliability that would be unimaginable in a symmetric system.

MULTICS and UNICS

The first major attempt to put these symmetric multiprocessing theories into action was in the development of the multi-user MULTICS operating system. The story of the MULTICS development project and the subsequent attempt by two of the MULTICS development team to salvage their *amour propre* by developing their own system, UNICS, on the side, is now the stuff of legends. A thoroughly revised anodyne history of the creation of Unix can be found in Wikipedia while another view can be found in "Multicians strike back"[3].

But the story I found in the "original sources" was rather different. Quite a lot of passages in these original sources were clearly untrue. There was a categorical statement that "fork" was the only

3 http://www.multicians.org/myths.html

MULTICS feature incorporated in UNICS. MULTICS was designed as a set of concentric shells round a kernel, UNICS was designed as a shell round a kernel (notice the similarity). MULTICS used a virtual machine execution model and, as "fork" works by replicating a virtual machine, UNICS also used a virtual machine model (coincidence? I do not think so) and so on. MULTICS had an execution model based on processes, UNICS had one process and later versions multiple processes (spot the difference).

I do not think that Richie et al were trying to mislead us, the striking similarities were probably more a result of UNICS being designed using the same dogmas as MULTICS, by people who had worked on MULTICS.

Despite this there were many differences (UNICS had, for example, separate "process space" and "files" unlike the MULTICS combined process and file space), but, in their various descriptions, the authors pointed out only one deliberate divergence from MULTICS: they abandoned the symmetric multiprocessing of MULTICS with its associated problems of "competition for resources" and the 1960s theories for dealing with these problems (including the synchronisation / mutual exclusion horrors). I could find no suggestion anywhere in the documents that mutual exclusion was omitted because the theories were fundamentally flawed: the authors apparently regretted leaving it out – they did so only to simplify the system so that it could be made to work.

I repeat "simplify the system so that it could be made to work". That was the key, UNICS was designed to work, it was not designed on the basis of academic theories.

Furthermore, when UNICS was extended to handle more than one process at a time, mutual exclusion was not re-introduced into the kernel. The authors noted that UNICS worked **despite** the omission of mutual exclusion, but they do not appear to have considered the possibility that UNICS worked **because of** the omission of mutual exclusion.

UNICS and Unix

Over the next few years UNICS became Unix and as it was ported to more and more powerful computers, developers started to put back into Unix those things that had been left out of UNICS – no wonder that it hardly worked any more.

By 1983, Unix had appeared on a small number of "executive workstations" (or rather executive toys) such as the Three Rivers Perq and the Sun, although, at that time, "the industry" treated Unix as a joke. Most computer manufacturers thought that their customers would prefer to have their payroll output reliably and correctly every Friday rather than sit typing commands such as `grep "\< [tT]he\> end"` all day. Or, as I heard it "How can you trust an operating system whose commands sound like body functions" (ps, sh, fc, grep, awk). The main usage of computers at the time was carrying out the same operations every day, week, month or year, reliably and predictably – in other words, boringly. Unix, on the other hand, could do almost anything – and to make it even less boring, it did do almost anything, regardless of what you might want to do.

25 Years - Out of the morass . . .

The starting point for Domesdos

Basic design criteria and philosophy

There were 5 basic design criteria for Domesdos

Compactness

Unlike most executive toy and personal computer operating systems, the self contained operating system for the QL had to be resident in a target 16k ROM.

Efficiency

It might seem obvious, but as the raw power of the QL was less than that of the first 1981 IBM PC, the operating system needed to be efficient.

Reliability

This might seem rather odd for a company like Sinclair which did not have an outstanding reputation for the reliability of its products, but there were two reasons for this, although both would disappear before the first machine was delivered. The first was that the operating system was to be delivered in ROM and it was not easily upgradeable. The second was that the machine was targeted at a more "professional" market than earlier machines because Sir Clive did not want to be in the games market - he wanted to be taken seriously.

Predictability

The dominant form of "serious" on-line computing was connection to a multi-user timesharing mainframe. This form of office working had created a new stress syndrome. A major contributing factor was the annoyance or frustration caused by highly unpredictable response which varied from sub-second to tens of seconds. The predictability of the response to user actions had become major requirement.

Accessibility

The general philosophy of a multi-user system (and this includes Unix) is of a **restricting** system whose primary aim is to restrict users access to prevent them taking control of the whole system. Domesdos, however, was to be an **enabling** system to maximise the accessibility of the system and hardware functions for both specialists and hobbyists.

Things to avoid in Domesdos

A good starting point seemed to be defining **THINGS TO AVOID** (the capitals are for fans of Terry Pratchett's Discworld - to be spoken with a hollow, death-like voice). These things to avoid were those academically popular ideas that seemed to lead straight towards complexity, poor or unpredictable performance, fragility or any combination of these.

1. Wilful ignorance
2. C programming language
3. Object oriented programming
4. Virtual memory / virtual machines
5. User based security
6. Synchronisation
7. Minimalisation

These **THINGS TO AVOID** are described in more detail in Box 1 and the means used to avoid them in Box 2. I did not realise at the time that these things to avoid would become the objects of worship by a narcissistic idolatry cult popularly known as the "Computer Scientists". I suppose that I should now call them the "Seven Cardinal Sins of System Design".

Good intentions . . .

The grand plans for a super operating system were derailed by a whole series of compromises required to fit Domesdos into the world of the QL hardware, to support the Psion office suite written for a CGA display on an MS DOS based IBM PC and to accommodate rapidly evolving in specifications and target markets.

Box 1 – Domesdos's things to Avoid

The seven cardinal sins of operating system design as seen from 1983.

1 Wilful ignorance

It should seem obvious that if you wish to build a system that works well and reliably, it is a good idea to know its performance and know its limits rather than sticking bits together following a set of arbitrary (possibly inappropriate) set of rules and hoping that it does the job. Apparently, this is not obvious.

While a certain amount of care is required to produce efficient code and there are some trade-offs between efficiency and code size, inefficiency comes mostly from not bothering to quantify the costs of operations and, therefore, wasting valuable resources through sheer laziness. Similarly, a system is likely to have a very unpredictable response if no effort is taken to evaluate worst case (or worst likely case) behaviour.

In all the documents concerning the development of Unix, I did not find a single document with calculations of the cost of any basic operating system function. The authors did not seek efficiency so Unix was inefficient by default.

In all the mass of 1960s theories on "multiprocessing" I did not find a single typical or worst case cost calculation to justify the complex mechanisms proposed for managing "competition for resources" or protecting "critical sections": these theories relied on asserting the "obvious superiority" of something that was not obviously superior.

2 C programming language

If you are going to program a version of Unix, C would be the obvious language, would it not?

I was not convinced. The various documents I had found about the original versions of Unix gave some very interesting figures on timescales. It appeared that the rewrite of Unix in C took less time than it took to write the original version in machine code, but not by much, whereas rewriting a piece of software should take much less time than writing from scratch (because you know exactly where you are going). Furthermore, the first time the C version of Unix was ported to another machine, it apparently took longer to adapt it than it had taken to write it for the first time. On the face of it C wasted time rather than saving it.

C represented a specific computer instruction set which was not appropriate and, even worse, it was tied to the Unix environment and concepts and, therefore, likely to induce typical Unix errors.

Finally it was so ill-conceived that, while it was possible to do really stupid things writing in machine code, writing in C you could do really stupid things without even knowing it.

3 Object oriented programming

Object oriented programming is based on "encapsulation" a fancy term for hiding all the dirty little tricks you do not wish others to know about inside a hard shell. Furthermore, rather than being explicit about the operations that are carried out, and how they are done, every operation is implicit, abstract or both: programmers are not supposed to know what goes in inside an object. The result is that nobody knows how a system created using object oriented programming works because nobody is supposed to know. It is all deep magic (when it works) or wilful ignorance (when it does not) – a **BAD IDEA**.

To cap it all, all operations using objects are stunningly inefficient. You need a byte of data from an object? It should take one machine instruction. With object oriented programming it takes at minimum tens of instructions and can be several hundred: just to make the system obscure.

4 Virtual memory and virtual machines

These two concepts are entirely independent but, as both require a dynamic address translation unit (a unit that converts the "virtual addresses" seen by an applications program into "real memory addresses"), they are often associated.

The supposed advantage of **virtual memory** was that it allowed the system to degrade more gently when there was a shortage of memory allowing systems to use less memory. This theoretical view is the result of a dramatic oversimplification of memory allocation processes. Experience pointed to the opposite conclusion. For example, in the late 70s when changing from IBM MVT (real memory system) to MVS (virtual memory system) the main memory had to be doubled in order to handle the same workload. This experience was repeated many times on many different systems.

The **virtual machine** model is a fundamental part of the 1960s dogma for multi-user systems. The idea is that each user "sees" a virtual computer that is completely isolated from the virtual computers seen by all other users, thus providing a naïvely simplistic security mechanism. This was, of course, totally irrelevant to personal computer usage where there is only one user and it had already be demonstrated to provide a fundamental security breach rather than a security mechanism. The other main drawback to the virtual machine model is that most operating system functions are concerned with transferring information to, from or between tasks - while the virtual machine model not only made systems more vulnerable, it made inter task communication both more complex and more costly.

5 User based security

In 1983, the dominant form of "serious" computing was time-sharing a multi-user central computer system. The same scenario formed the basis of the 1960s multiprocessing theories. For this type of system, security was limited to preventing individual users hijacking, using or corrupting other users' data. Oddly enough, UNICS, which was designed as a single user system, had user based security concepts from the start. For a personal computer (single user workstation) the multi-user problem does not exist, so there is no need for user based security mechanisms. At best they are merely obstructive and annoying while giving a false sense of security.

Unix had two separate user based security mechanisms: the "process" model of program execution and the file system owner/group/all and root/notroot concepts.

Processes are closely related to virtual machines. As Unix type virtual machines are more of a security risk than a security mechanism, the Unix process model merely makes a single user workstation more vulnerable.

The Unix owner/group/all concept of file system security was almost unworkable on multi-user systems as it assumed a strict hierarchy implying that each user belongs to only one group, and that only one group could be allowed access to a file. For a workstation it was totally ineffective: where a machine has accessible file store media (even if you need a crowbar to access it) or can be re-booted to a different operating system on a external drive, removable medium or over a network; the **only** effective mechanism against data theft is file encryption (using per-file keys NOT per-user keys) and there is no protection against data loss or destruction except for mirroring the data on remote storage.

6 Synchronisation

In the 1960s a whole edifice of theories was built up on the basis of using synchronisation as a means of providing mutual exclusion to resolve access conflicts between "processes". In fact this was misleading. The theories were not concerned with resolving the conflicts themselves, but concerned with resolving the problems arise when mutual exclusion is used to try to deal with these conflicts.

This created a self-sustaining spiral. The basic mutual exclusion theories simply made the underlying problems worse, which led to the development of synchronisation theories which exacerbated the problems of mutual exclusion which led to more theories...

In conventional systems, synchronisation mechanisms had also been adopted for signalling between tasks, for example indicating that data was available for processing. This too had proved to be the source of many fundamental system design problems.

Avoiding synchronisation and all its associated nasties was, therefore, a primary design aim.

7 Minimalisation

The concept of minimalisation is associated with the "less is more" and "worse is better" system design philosophies that developed in the 1970s and eighties to justify increasing idleness and incompetence.

The principle is that by minimalising the operating system functions, the complexity is pushed into the application programs, making it simpler and easier to design the operating system.

In practice the effect of this approach is rather different. As an operating system should not just be considered to be a set of core functions but the whole of the support for the applications programs, minimising the core functions has the effect of increasing the complexity and size of the "higher level" functions providing applications support.

It becomes even worse when the minimalisation is compromised. A real minimalist approach to reading data is to treat input from any device of any type as a stream and have just one "non-blocking" operating system call to read a either one byte or a given number of bytes from the stream. As the call returns immediately whether the read is complete or not, then this call can be used for both checking for input and reading from a file. To read any data that was not instantly available, the program would have to cycle in a tight loop retrying the call, which in most cases would be unnecessarily complex and inefficient.

The first typical minimalist compromise was to provide two calls: a (non-blocking) call to test whether there is data available and a (blocking) call to read a fixed number of bytes from the stream.

This did not solve the problems. It did not allow for keyboard input where the user may type characters and then edit them before hitting ENTER. This meant that the minimalist approach was then further compromised by introducing switches changing the behaviour of the read bytes function depending on the device and how the application wished to interact with it, increasing the system complexity significantly.

The end result is that compromised minimalisation not only makes applications programs and application support software inevitably more complex than providing an appropriate range of core functions, but the minimalised core functions themselves are very likely to be more complex than more complete set of regular core functions.

Box 2 – Avoiding the things to avoid 1

1 Avoiding wilful ignorance

All the Domesdos system data structures were completely defined (with provision for expansion) before any part of the system was coded.

In design, the execution time of all critical sections of code was calculated for both typical and extreme scenarios. For example, the scheduler design was fixed when it was able to schedule 100 application programs, active or waiting for I/O, with a worst case overhead of 50% of the processor time. The performance of the system was known before it was coded.

All timing critical services interfacing directly to the hardware had known worst case timings.

2 Avoiding C programming language

Conventionally, operating systems had usually been written in assembler (a family of programming languages based directly on the machine's instructions: one line of assembler translates into a single instruction). Unix was a notable exception.

Domesdos was not, however, written in machine code or assembler. It was written in pseudo code (the fancy name for any representation of a program using rules that are made up as you go along) which was then "hand compiled" even though hands had nothing to do with it.

This ensured that the implementation was not constrained by the limitation and peculiarities of C or any other programming language.

3 Avoiding object oriented programming

Any one with a knowledge of the principles of object oriented programming looking at the structure of Domesdos might think that the attempt to avoid object oriented programming had failed completely: every item in memory, including jobs, could be considered to be an instance of an object complete with constructor, destructor, and a variety of methods and properties.

A channel to a file, for example, could be considered to be an instance of a "file channel" object which added file specific methods and properties (position, flush, etc.) to the methods and properties (read, write, etc.) inherited from the "I/O channel" object which itself inherited basic methods and properties (create, destroy, ownership) from the "memory" object.

The Domesdos approach was, however, very different. Domesdos used "data design", a slightly earlier concept which, because of its simplicity, found little favour with academics. Most programming languages are algorithmic or procedural and not particularly concerned with data. Object oriented programming is the apogee of the procedural approach as the data is completely inaccessible.

Data design was a programming approach that took, as its basis, the primordial value of data and the relative insignificance of procedures and algorithms. This is not an ideal approach for calculating the value of PI or drawing fractals but, then, as now, most computing outside research laboratories was concerned with data handling rather than intensive calculation.

The principle of data design was that data structures should be designed to be well defined for all possible states. For example, rather than writing an algorithm or procedure for suspending a job, the executing and suspended states of the "job control data structures" are first defined and then the code for suspending a job "writes itself".

There are similarities between the Domesdos use of data design and some of the aims of object oriented programming. A "file channel block" had all the data structure of a basic "I/O channel block", so that all code that could operate on an "I/O channel block" could also operate on a "file channel block". Likewise, an "I/O channel block" had all the data structure for a "memory block", so that all code that could operate on an "memory block" could also operate on an "I/O channel block" and a "file channel block".

There are also major differences. Because the data blocks in Domesdos were defined explicitly, the Domesdos file system device driver (privileged code) could not only access the file channel block defining a "channel" from the application to a file, but also the associated filing system block (shared between all files open in a particular filing system), the associated physical device block (shared between all filing systems on a disk) the associated disk interface block (for all disks on a particular bus) and the operating system block which held all information common to applications, device drivers and hardware.

This simplicity led to ridiculous accusations that the system was unsafe: an error in the privileged device driver would not necessarily be contained. This is absolute nonsense based on the academic view that reliability is a matter of keeping the system going regardless of how much damage is being done to the data. An error in the filing system will destroy data: the system will be broken whether or not other system structures are damaged. Intrusive containment measures simply increase the complexity and, therefore, the increase the probability of there being errors while reducing the probability that those errors will be detected.

The data design principles used in Domesdos, therefore, provided the useful features of object oriented programming in an open, clear, explicit, efficient, natural way instead of the closed, obscure, implicit, inefficient, object oriented way.

4 Avoiding virtual memory and virtual machines

Avoiding virtual memory was not difficult as the hardware did had neither dynamic address translation nor fast backup storage. The memory management strategies used did not, however, preclude the use of virtual memory.

It would have been possible to implement a virtual machine memory model, by shuffling the contents of memory on every task switch, but as this would merely have added to the inefficiencies inherent in the virtual machine model, a real address memory model was implemented.

5 Avoiding user based security

The Domesdos application task model was based on the classic "job" concept. I have seen Domesdos jobs described as processes, but they certainly are not. If you were trying to be contentious, a Domesdos job could be described as combining all the advantages of Unix processes and Unix threads while avoiding the drawbacks of either. But I will not describe them that way as Unix enthusiasts are not noted for their sense of humour.

Although the *hardware* did not support any form of protection against accidental or deliberate corruption by one task of the data belonging to another, Domesdos did have a rights system in the form of "ownership" and "usership". This rights system could have been enforced if appropriate hardware had been available. The "ownership" and "usership" of data and program memory blocks made the system self-cleaning provided that tasks did not abuse the rights system.

A Domesdos job has its own code base and data space. It can spawn independent jobs having no access to the spawning job's code base and data space (like processes). It can spawn dependent jobs which, by virtue of the separation of "ownership" and "usership" rights, may have their own code base and data space (like processes but unlike threads) and may access their owner's code base or data space (unlike processes but like threads).

As user rights to files on a workstation are completely unenforceable, files were not flagged with user rights. Per file encryption, which would have been the only effective data protection mechanism, was considered too complex.

6 Avoiding synchronisation

The earliest versions of Unix did not use synchronisation mechanisms for dealing with access conflicts; they relied on operating system calls being atomic unless voluntarily suspended. For application programs, where a response time of some tens of milliseconds is adequate, this is an simple, efficient and safe approach and it was used to a certain extent in Domesdos.

It is, however, unsuitable for dealing with contention for access to shared data structures between interrupt servers and other software. Rather than using invasive mechanisms such as disabling interrupts to protect "critical sections" or, even worse, using symmetric synchronisation mechanisms, Domesdos implemented a range of asynchronous, asymmetric access mechanisms. For example, if an interrupt server needs to release a scheduled task, it can do it at any time, even while the scheduler is in the process of rescheduling, without any lost events or any precautions being required in the scheduler code or the interrupt code to prevent access conflicts. These mechanisms do not have any "critical sections" and so, in Domesdos terminology, they were called "intrinsically safe". These mechanisms were developed specifically for Domesdos but some of the ideas were partly based on the concepts for asynchronously updating distributed databases that were being developed by the systems group at the CADCentre, my previous employer.

Synchronisation mechanisms were also avoided when flagging completion of asynchronous processes such as transmitting or receiving data on an I/O port. Events (intrinsically safe) were used instead.

7 Avoiding minimalisation

Rather than seeking to minimise the operating systems interfaces, Domesdos sought to regularise the interface by providing a broad, coherent set of basic functions. Using a simplistic analogy, the broader the base, the more stable the edifice built on it.

The best example was the I/O sub-system. For reading data, separate calls were provided for testing, reading a single byte, reading a "line", reading a string of bytes and unbuffered direct reads. There was no arbitrary "blocking / non- blocking" behaviour on individual calls: all calls had a timeout parameter from 0 to 10 minutes (or wait forever) whether or not a timeout had any sense for a particular call.

Because of this regularity, and because the operating system itself handled the timeout, buffering and event signalling, writing a comprehensive IO device driver for Domesdos was much easier than writing a primitive IO device driver for Unix or even MSDOS (or at least it did seem that way to me – some accurate documentation would have been a help for others!).

Desperately Seeking QL

Although I made good progress in completing my QL collection last year I'm still looking for the following:

- Video footage on the Sinclair/QL in TV programs (on VHS tape or as files).
- Photographs of Sinclair/QL appearance in IT-shows such as Personal Computer World show (PCW), London, Which Computer? show, Birmingham, Earl's Court Computer Fair, London or ZX Microfair, London (1984 to 1988).
- More software for GST's 68K/OS beside the two bundled Microdrive cartridges.
- More QL software/articles written by Linus Torvalds.
- Sinclair QL FB ROM (physical EPROMs or binary file).
- Sinclair QL PM ROM (physical EPROMs or binary file).
- Any Sinclair QL Professional Computer, serial number (S/N) pre D04-001371.
- Any Sinclair QL Professional Computer, S/N post D16-122418, a D17 would be very nice.
- Any Sinclair QL Professional Computer US-Edition, S/N post S13-005854.
- Any Sinclair QL Professional Computer German-Edition, S/N post SG18-010800.
- Other Sinclair QL regional editions (Spanish, French, Italian, Danish, Turkish, Greek, Portuguese, Norwegian, Swedish, Finnish and Arabic).
- Any names or contact to people who worked at a subcontractor of Sinclair Research like Thorn (EMI) Datatech.
- QUEST CP/M 68K for the QL.

Just drop me a line (mailto:**urs_koenig@bluewin.ch**). Thank you very much.

I hope you enjoyed this journey back in time. QL forever!

25 Years - Part 2

by Tony Tebby

Welcome to the next part of our series. More has been prepared, and Tony writes "Just the FUTURE to do". We ended last issue with "25 Years - out of the Morrass ..." and continue:

25 Years - . . . into oblivion

Domesdos hits the fan

The peer review

How well was the system received in the computer science world? Not very well. The main criticisms were that it did not incorporate all the idiocies that I had deliberately avoided. In other words, it was criticised for how it worked rather than whether it worked well.

Other criticisms centred on the OS interface. "It stinks" was one verbal reaction - the silent reactions were probably worse. For me, the oddest thing about the criticisms was the irrationality - the thing that stank the worst seemed to be the I/O calls. These were designed to meet a wide range of requirements including communications and interactive use where an application might have to react if data was not received within a given time. This meant that all I/O calls had a timeout. This combination of scheduling (for the timeout) and data transfer seemed to be fundamentally offensive to any true believers in operating system purity. Apparently the "correct" way of dealing with this was to make the application spin in a tight loop checking the elapsed time, checking for data, transferring the data and processing it a byte at a time. Is this really better, in any respect, than making a single operating system call? A lot of people thought so.

The third group of criticisms centred on the security or, rather, lack of security in Domesdos. The main concern was that as the hardware had no protection mechanisms, the system did not try to provide a false sense of security by implementing arbitrary, highly obstructive protection mechanisms that would, in any case, have been totally ineffective on the QL platform.

Users' reactions

I have no doubt at all that there were many users who were disappointed in the system and there really was no adequate documentation on the operation of the system, no guidelines and interface information. On the other hand, "ordinary" computer enthusiasts did not suffer from the preconceptions of systems "experts" and so a fair number succeeded in making the system do all sorts of extraordinary things. The lack of formal guidelines for programming showed up some unforeseen characteristics: on the one hand, the ability of the system to put up with extreme abuse while continuing to function and, on the other hand, the ability of a certain type of person to take great pleasure in exploiting all the undefined holes in the system interface that will always occur if you adopt the "garbage in, garbage out" approach used by Domesdos.

My own assessment

Relief that it worked. Although I had believed the theoretical basis to be sound, it was still a relief that I had not completely screwed up.

Anger at myself for having compromised the integrity of the system to incorporate SuperBASIC rather than changing SuperBASIC to be "Domesdos friendly".

Pleasure when, on a simple system test with 100 application programs, the QL outperformed a VAX running VMS.

Frustration that I could not redesign it from scratch and "do it right" the second time around on a better platform without the compromises for CGA compatibility and the QL hardware.

Meeting the design criteria

Compactness

The first version of Domesdos met its target for compactness. The core occupied less than 5 kbytes even though the range of core operating system functions was very much more comprehensive than Unix. The complete set of device and filing system drivers occupied less than 10 kbytes.

The compactness of the core functions was largely due to the use of pseudo code for programming, the "real world" approach to task management, the use of events in place of synchronisation and the adoption of regular, coherent interfaces.

The compactness of the device and filing system drivers was largely due to the simplicity of the interfaces provided by the data design concept and the use of intrinsically safe intertask communications between the interrupt servers and the operating system functions.

The compactness was not achieved by "crafting" the code. Subsequent development indicated that some sections of the code could have been reduced by more than 20% by more careful coding.

Efficiency

The first version of Domesdos was not as efficient as it could have been, but all the targets were met. Basic operating system functions were more than an order of magnitude faster than Unix SV. One hundred applications running simultaneously did not bring the system to its knees. The only real bottleneck in performance was the display handling, but this was not a basic design fault, it was simply an unsatisfactory trade-off between speed and size of code in the first cut that was never to be revised.

The efficiency came from an awareness of the costs of various options that were selected and the many options rejected, the absence of any intrusive synchronisation mechanisms, the use of a real memory address space model and the extensive use of shared data structures that this memory model and the intrinsically safe access mechanisms made possible.

Once again, the efficiency was not obtained by crafting the code. Some critical sections could have coded up to 30% faster.

Reliability

The reliability of the system has three different aspects. There may be problems with the fundamental system design, there may be problems recovering from or handling external errors (hardware faults, exception conditions, OS call parameter errors, etc.) and there are simple coding errors.

These three sources of errors were all dealt with by design.

The probability of simple coding errors was reduced by using a data design approach translated into machine code via state diagrams and pseudo code. The error handling was made simpler by having a single, rather primitive, mechanism for reporting errors from procedures which, together with the regularity of the interfaces, reduced the probability of errors in the error handling. The two major differences between Domesdos and conventional operating systems were, however, the total elimination of synchronisation and mutual exclusion mechanisms and the regularisation of the operating systems interface which, between them, eliminated the root causes of most of the known design problems in operating systems at the time.

In a fit of hubris over this "design for reliability" approach, the new operating system was called Domesdos (a home (domestic) DOS, even though it was designed for business use and it was a ROM operating system and not a disk operating system) after the slogan for a brand of bleach "Domestos" which "kills 99% of all known germs (bugs)".

How well did this approach succeed in eliminating bugs at source? In reality better than the system's reputation might indicate. When the first QLs were shipped, Sinclair set out to create the impression of bug-ridden software to mask the chronic hardware problems. To back this up, machines were deliberately delivered with pre-test firmware (c.f. "QL firmware bugs myth"4). Furthermore, Sinclair's failure to organise any form of operating system documentation led to a whole bug-hunting industry, with journalists fighting it out to produce the most extravagant lists of "bugs" without actually knowing what the system was meant to do. Mark Knight gathered all these together, weeded them and created the "definitive" list of 77 bugs and quirks, most of which were only in pre-release versions or the SuperBASIC interpreter and its associated utilities, procedures and graphics. These bugs are analysed in "QL ROM bugs - an annotated list"5 which shows that there were 10 bugs and serious quirks in the first release version (V1.03 JM) of the operating system and device drivers, excluding SuperBASIC and the graphics (see Box 3).

It is quite possible that there were other errors and the system could have been better, but by industry standards, for an operating system developed from scratch within a six month deadline using new, radically different, paradigms, it was probably better than industry average.

Predictability

From the a user's point of view the ratio of the median response (typical response) to the worst case and the incidence rate of long delays are two reasonable measures of the predictability.

In some respects, users would have found the system occasionally slow and unpredictable. With a mean Microdrive access time of 3.5 seconds, fetching data from files was bound to be slow. By comparison with simple buffering, the file imaging (borrowed from CST's 68KOS) and pre-fetch strategies significantly reduced the median access times but could do nothing to improve the worst case delays, thus the predictability measured as the ratio of median to worst case delay was made worse. These strategies did, however, significantly reduce the incidence rate of long delays .

On the other hand, the character drawing speed was certainly too predictable! The screen driver was designed for compactness first, speed was only a secondary consideration. The most compact design was to draw all characters using the same code, regardless of the complexity of the operation. As common cases could have been be drawn much quicker than the general case, using separate code for these would have increased the apparent performance at the cost of predictability. It should have been done.

Accessibility

Domesdos was very accessible.

The end point

The release of the QL was also the effective endpoint of development of that operating system concept. The general hostility of the computer science world to the concept, coupled with the fact that the system was closely tied to a flawed hardware platform and Sinclair's decision to make the firmware a scapegoat for the early QL production problems would have turned Domesdos into a little footnote in the history of operating systems if the Sinclair marketing department had not buried it.

4 http://www.t-t-web.com/OS/QL_firmware_bugs_myth.pdf

5 http://www.t-t-web.com/OS/QL_ROM_bugs_list.pdf

Domesdos was now called QDOS by the marketing department. Possibly they did not know that QDOS already existed and was alive, if not very well, and that Bill Gates had been to Sinclair Research in a attempt to license it to Sinclair for the QL. Possibly they thought that the operating system on the QL had been licensed from Microsoft! As the ultimate insult, they took its name away, and hung the albatross-like "QUICK and DIRTY OS" epithet round its neck.

Box 3 - First release bug list

The numbers are those in Mark Knight's original "definitive" bug list

1. A trivial coding error (bug 57) checks only one plug in card (Plug 'n Play in 1984). Simple workaround.
2. Serious oversight (bug 8) remapping the colours on changing from eight colour to four colour mode. The driver did not check whether the "ink" and "paper" mapped onto the same colour in the 4 colour mode.
3. Simple coding error (bug 15) panning a window less than 8 pixels wide. Why would you do this?
4. Trivial coding error (bug 43) filling a full screen width block. It actually did nothing at all, I do not know why.
5. One bit coding error (bug 37) closing a serial port. A little endian IPC connected to a big endian MC68000.
6. Serious oversight (bug 33) pre-fetching Microdrive sectors when you had run out of memory. This rather reduced the buffering efficiency.
7. Serious coding error (bug 19) in the Microdrive device driver opening MDV8. There was no MDV8!
8. Fatal oversight (bug 71) sending a null file over the network. One end just waited for nothing for ever.
9. Serious coding error (bug 50) in the Trap #4 / Trap #3 patch to cater for a "moving task" (SuperBASIC).
10. Design error (bug 49) handling job release events. This did not allow for nested release events making it possible to break the system using apparently legitimate OS calls.

25 Years - Development of the Domesdos concepts

SMS2

When it had been well established that Domesdos did work reliably and significantly more efficiently than conventional multitasking systems, I set about writing version 2. The interface was cleaned up to remove "QL nasties", which meant that there was no SuperBASIC, graphics or Microdrive support and the OS calls that had been patched in to support direct access to the QL hardware were not included. This SMS2 (Small Microcomputer System V2) was hardly any larger than the original Domesdos core and, typically, important operating system calls were 30% to 100% faster as register handling was optimised for the more complex calls, rather than the simplest, and the code was crafted more carefully.

There was no permanent user interface program but, whenever there were no jobs in the system, a default application (job 0) was started. The only job 0 application that was written was a simple command line interpreter that could read from a file or the console. This was, effectively, going back to the original Domesdos concept.

This system was implemented on the Atari ST monochrome system and feasibility trials were carried out on a small number of embedded systems. It was never made available commercially on "standard" platforms and no project using it ever made it to market.

Windowing software

Windowing is not normally a fundamental part of an operating system, it may be a function of the display driver or it may managed by an independent task, in either case the emergent windowing systems 25 years ago all relied on the applications programs doing most of the work, although some of this work may have been hidden in "API" libraries. This approach of requiring application programs to re-draw, at any time, regardless of what they were doing at that time, any parts of their windows that became uncovered, was so costly and fragile that it seemed that the real way forward was hardware windowing. This should have been able to provide a much better price/performance ratio than a 100% software approach.

The original Domesdos display driver was based on writing to windows rather than the whole display. This was a prerequisite for a extension to a windowing system although it did not actually provide windowing facilities. The upgrade path was laid out but no more than that.

The QJump Pointer Interface QPTR was designed as a stopgap measure for the QL. It provided true windowing without clipping, it was "optimised" for a platform three times slower than the recently released Apple Mac, it had to be compatible with existing QL software and it was based on the assumption that windowing hardware would soon become available. As a result, the overlapping window pro-

blem was dealt with by freezing partially covered windows: a less than satisfactory approach, but, possibly, the only practical short term solution.
At the "presentation level" (the Window Manager), however, there were a number of innovations which were intended to deal with some of the less desirable features of what was to become the "standard" windowing interface as presented on the Apple Mac, the Commodore Amiga and the Atari ST. Some of these innovations (right click menu, button bar, etc.) were adopted in improved form in later mainstream systems (context menu, task bar / dock, etc.). Others, such as the Hotkeys, were not.

SMSQ

SMSQ was SMS2 to which SuperBASIC compatibility was retrofitted. It was intended to provide a QL compatible operating system for various QL emulators. In general it was much faster than QDOS (executing in equivalent speed memory). The expanded console driver could draw strings of characters at speeds within a few percent of the QDOS add-on record holder and it could draw single characters faster. It was purely retrospective and did not further the Domesdos style OS principles.

SMSQ-E

SMSQ-E was SMSQ bundled with the standard QJUMP Extended Environment. There was no development effort available to provide a mechanism for writing to buried windows which had become practical with the availability of "QL compatible" platforms 5-10 times faster than the original QL. Various enthusiasts and third party developers produced their own solutions by continuously updating the display buffer from the off-screen window buffers. This was a bit of a patch that was later adopted by Mac OS X and Windows Vista.

Minerva

Minerva was a re-engineered QDOS / SuperBASIC ROM destined only for retrofitting to QLs. it provided improved performance with a reasonably high level of QL compatibility. Although it provided interpreted multitasking BASIC it did not represent any form of advance in operating system principles.

Windowing hardware

A hardware windowing display circuit was designed using 1980s technology. The principle was very simple: for each frame there was a table of "pixel runs": the start address in the display memory, the number of pixels in the run and the attributes (colour depth, colour map, and display/porch/blanking/sync). While this made displaying a line slightly more complex than a simple rectangular memory map for the whole screen, it greatly simplified the generation of the blanking and synchronisation which were just special pixel runs and entirely defined in the pixel run table. The principal additional cost component was a deep FIFO to smooth out the pixel rate at the window edges.
This system allowed for arbitrary shaped windows, which could be moved simply by changing the pixel run table. At the time, large display memories were expensive so the design allowed different windows to use different palettes or different colour depths to economise on memory usage.
It never made it to market.

Stella

Stella was designed as a "lets get it right the second time round" Domesdos. After a few years, I felt that I understood most of the problems with the original Domesdos. Stella was based on an updated, more rigorous version of the Domesdos design principles applied to a wider range of platforms.

- Elimination of the UNIX legacy
- Extended data design concept
- Wider OS interface base
- Higher efficiency
- Less memory constrained
- Generalised entity management
- External event management for guaranteed real time response
- Rationalised and more rigorous ownership and usership concepts
- Mix and match modularisation of core operating system functions

Stella on test

At the request of someone at Sun Microsystems, Stella was benchmarked against Unix SVR4 (Solaris 2) on a Sun3x equivalent platform (Sun's project to replace BSD Unix was not giving the desired improvement in performance). The benchmark conditions were not very well defined but, on the first tests, Stella outperformed SVR4 by around 2 orders of magnitude on simple OS calls and filing system operations. My contact in Sun told me that he could not possibly pass these figures onto his project manager as he would lose all credibility!

Stella on the market

Stella was implemented as the embedded operating system for a number of projects, but none ever made it to market.

25 Years - 1984 to 1986 - GUIs and RISC

The first mass market GUI machines

Apple Macintosh

The Apple Mac was launched soon after the QL, although it had been in development for nearly 5 years. With a price tag three times that of the QL and operating system development costs about 50 times that of Domesdos, the machine was far more polished than the QL. It has since become a landmark, for many of the wrong reasons. Many histories of the development of personal computers placed the Apple Macintosh as the computer that started the windowing revolution. In reality it had very little lasting effect.

A recent history of the Mac states "it had very little memory . . . low processor speed and limited graphics ability". The "little memory" was surprising - it had no more memory than the vastly cheaper QL and less than typical for a new PC but was aimed above the PC market. Given the lack of expansion capability, this was a clear marketing error. The "low processor speed" is more interesting. The processor was comparable to a PC of the period, but the system was slowed down by the windowing software which was based on the principle of pushing all the real work up into the application. It is not that the software was particularly inefficient, it was more that the windowing principles were totally inappropriate for the single tasking Mac.

The operating system was reasonably compact, having a basic I/O and filing system, primitive memory management, scaled pixel fonts and desktop in 64k RAM.

It fell down very badly on accessibility. Early software development was limited to a few privileged partners who had received advance specifications and cross development systems. Ordinary developers had to wait two years before the MPW native development platform became available and they were able to sample the dubious delights of programming in an environment where the operating system imposed very serious constraints but provided very little assistance.

This lack of system accessibility meant that, despite its popularity with journalists, the persistently over-the-top reviews and an extravagant marketing campaign, the Mac did not manage to stop the much more basic MSDOS PCs becoming dominant. Software development was just too difficult.

Over the next few years, Apple's marketing strategy for the Mac was pursued through advertising and vigorous, totally baseless, legal action presenting Apple as the underdog suffering under the market dominance of IBM and from theft of its intellectual property. At the time the campaign started the IBM PC was a long way behind the Apple II in terms of installed base and third party software support - IBM was the underdog - and the disputed intellectual property did not even belong to Apple.

Although both parts of Apple's strategy were based on misrepresentation, this has left a lasting impression that the Mac was an innovative machine that was pushed out of the market by IBM monopolistic marking practices. In reality, its software was 99% derivative and it was unable to compete with Apple's own Apple II series.

The Mac had one feature borrowed from the Lisa that did get copied. This was the menu bar at the top of the screen. The menus for each program or window were not part of the program's windows, but "belonged" to the screen as a whole. This bizarre arrangement also appeared on two computers that set out to succeed where Apple had failed: the Atari ST and the Commodore Amiga.

The Atari / Commodore twins

The Commodore Amiga (which was actually an Atari games console) and the Atari ST (which was designed by Commodore or by engineers that had left Commodore, depending on who you believe) both had user interfaces based on the Mac. Put on the market a year after the Mac, both had more powerful hardware than the Mac and both were milestones in the development of personal computer system software.

The similarities between them and the differences between them and the Mac were striking.

- Their MacAlike GUIs were patched in at a late development stage, whereas the Mac was designed that way.
- They both opted for the efficiency and simplicity of mono-spaced fonts, whereas the Mac was designed to use proportional, scalable, fonts.
- They were both designed for expansion whereas the Mac was a take-it-or-leave-it closed system.
- Their systems were jumbled heaps of recycled bits of archaic systems and reverse engineered clone software, whereas the Mac system was designed for the job.

It is this last aspect that made them the true precursors of the systems that have dominated systems development over the last 25 years.

There were, however, significant differences between the operating systems.

The Atari ST, with its rather simpler system, easily outperformed the Mac although its inbuilt mono-spaced text made the screen (desktop) look look rather clunky. A large part of TOS (The Operating System) was recycled from CP/M.

The Amiga screen (workbench) also looked rather clunky. However, apart from the demos which bypassed the operating system, it was slow (particularly the filing system), even by comparison with the Mac, despite the Amiga's more powerful hardware. Moreover, the although the operating system was arguably less capable than the Mac (the Domesdos experience had shown that scalable fonts "cost" far more than multitasking), the operating system was grossly oversized, eating up about 4 times as much memory as the Mac. Part of the reason for both the low performance and the enormous size of the operating system was that it was bodged together from three separate and largely incompatible units: the recycled Exec and AmigaDOS and the reverse engineered MacAlike Workbench.

What was amazing was that the press, as a whole, accepted the proposition that the Amiga was slow because it had a "powerful" operating system. This gave a whole new, hitherto unsuspected meaning to the word "powerful".

The Amiga operating system was "powerful" in the sense that, by comparison with other personal computer operating systems of the period, it consumed vast resources and, therefore, required a powerful hardware platform for it to work at all.

It is difficult to see "powerful" in this sense being applied to any other engineering domain. Can you imagine automobile journalists raving over the "power" of a car just because it guzzled fuel in unprecedented quantities despite an acceleration that would make model T Ford blush and a top speed that guaranteed that you could never never pick up a fine?

24 years on, "powerful" in the sense "memory hungry and inefficient" is now accepted usage. Only today, I was helping out someone who was lamenting that the latest software versions (and you must have the latest!) were too "powerful" for his 2 year old machine.

The Amiga, therefore, was the computer that established

- "scrapyard" (recycled, re-used) software technology,
- "bloatware" and
- the new meaning of "powerful".

1984 - X Window System

The X Window System is now closely associated with Unix but the original version "W" was written for the V operating system. The development of X at MIT, which started a year after Domesdos, has had a long lasting impact in two respects: not only is X the dominant display management software in the Unix world, but the egocentric minimalist design philosophy adopted for X has also been recycled many times in computer science. To be fair to Bob Scheifler and Jim Gettys, the developers of the X Window System, X was never designed for the purposes for which it is now misused and they did not originate the egocentric minimalist design philosophy: they just gave it an air of respectability by formalising it as part of their 7 golden rules.

Box 4 - X Window System's 7 golden rules

1 **Do not add new functionality unless an implementor cannot complete a real application without it.**

If this golden rule were to be believed and followed rigidly by systems developers, it would be a disaster for the world. This is the minimalist philosophy in its most rigid form. As "application implementors" can, in principle, do anything that an operating system implementor can do, this implies that the system should do nothing.

With this rule, there is no question of whether implementing useful but non-essential facilities could be better for the overall system (operating system + applications). All functionality is dumped on the applications developers regardless of the overall cost in reliability, performance and development effort.

This rule, although being the first, most important rule, is so stupid that it was not was not followed by the developers of the X Window System. This provided a number of non-essential drawing primitives, in particular glyphs (characters). Drawing glyphs is clearly not "required" as applications can, of course, do it themselves.

2 **It is as important to decide what a system is not as to decide what it is. Do not serve all the world's needs; rather, make the system extensible so that additional needs can be met in an upwardly compatible fashion.**

There are two different completely different concepts in this golden rule. The first is a platitude that restates a fundamental engineering rule "design to purpose". This implies that the purpose is known - for operating systems that means reading the future. The second would seem to be good advice, but is usually taken to mean **"do not do today what you can leave until someone else is forced to do it".**

3 **The only thing worse than generalizing from one example is generalizing from no examples at all.**

This was prophetic. Although originally the X Window System avoided generalisation by being designed for a single case, it must be the most significant example of "generalising from one example". Designed for a system architecture that was already archaic (X Terminals connected to a centralised time sharing system) it has been generalised for architectures for which it is fundamentally unsuitable (such as stand-alone workstations).

But there is much worse than the X Window System. The 1960s operating system theories that have been the cause of so many system design errors and problems were based on a, rather ill defined, abstract system architecture that corresponded to no real computer. They are the prime example of "generalizing from no examples at all" so maybe this golden rule is right.

But are there other things worse than generalising from one example? Unix generalized the "file" concept to apply to I/O devices, with horrific results. MSDOS had no generalized I/O at all. If you believe that the MSDOS approach was worse than Unix (I am not taking sides) then you will have to take it that not generalizing at all is also worse than generalizing from one example.

4 **If a problem is not completely understood, it is probably best to provide no solution at all.**

If you cannot completely understand the problems that you have to deal with, first try changing the problem, if that does not help, change jobs to something less mentally exacting (a trader in the derivatives market, for example). Copping out by providing no solution at all is, at best, lazy and irresponsible, at worst, criminally negligent.

5 **If you can get 90 percent of the desired effect for 10 percent of the work, use the simpler solution.**

Even if 90% were good enough (would you buy a car where only 90% worked?), it is highly improbable that you could make a system 90% functional for only 10% of the effort. This is no more than a justification of **"if you can get 10 percent of the desired effect without thinking about it, that's OK, take a break".**

6 **Isolate complexity as much as possible.**

The X Window System does some fairly complex things, but they are certainly not isolated into easily maintainable modules, they are built into the amorphous mass. Isolate, in this context should, therefore, be take to mean bury out of sight.

7 **Provide mechanism rather than policy. In particular, place user interface policy in the clients' hands.**

This is abdication of responsibility. Policy is far more difficult than mechanism. It is very wise to separate policy from mechanism. It is very wise to provide flexible policies. It is very wise to allow to policy to be adapted to changing circumstances. A system with every application imposing its own policies is, however, an end user's nightmare.

X Window System's 7 golden rules

A windowing system has the same requirements for compactness, efficiency, reliability, predictability and accessibility as any other system software, even if "accessibility" for developers (coherence and richness of the functionality) and accessibility for users (ease of use) are measured in different ways. Bob Scheifler and Jim Gettys' seven golden rules, however, do not address any user requirements (neither end users nor developers): they are concerned merely with producing something that gives an appearance of "working" while requiring a minimum of effort.

A fuller analysis of these rules is to found in Box 4, but I have tried to translate these into plain English.

1. Push all functionality possible up to the applications regardless of the effect on overall system reliability, complexity, efficiency or cost.
2. Leave anything that you do not feel like dealing with for someone else to deal with later.
3. Design for specific cases only.
4. Do not do anything that that might require thought.
5. Quick hacks are always justified.
6. Bury your hacks so that no-one will ever find them.
7. Do not bother about what the system will do, just how it will do it.

X Window System's implementation

The early X Window System implementations crawled out slowly until X11 was released in 1988. The system was nowhere near as bad as it would have been if the rules had been followed as it included a large number of useful but non-essential features (breaking the first three rules). It had, however, three major failings: the speed, the complexity required in the applications and the bizarre architecture.

The speed

There seems to be very little concrete information about the speed, apart from endless nagging. In 1992[6], 8 years after the development started, The professor of electrical engineering at Berkeley replied to criticism "with the 50 MIPs computers we are seeing now, performance is not a problem". This reply implies that the performance was a problem up to then and there is much anecdotal evidence to to suggest that the performance remained a problem for most of the next decade. In 2004, FBUI[7] was patched into Linux 2.6.9 to replace X "which can be an impossible burden".

The complexity of the applications

The display refresh policy required each application to redraw its windows when they were uncovered by another window being moved or removed. Although an attempt was made to justify this window redraw policy in a seminal paper published in ACM Transactions, the "justification" is very thin: In reality the policy used by earlier windowing systems was simply adopted without any real consideration of the possibilities for "local display refresh" where the windowing system itself deals with changes in the window arrangement. The basis was that applications were better able to redraw the display than the windowing system itself.

Those who have never tried to program for a mainstream windowing environment may not understand just how far removed from reality this is. This "better" refresh policy requires applications to be able to respond to redraw events within the human perception time (0.1 sec), whatever they might be busy doing at the time, and maintain their own data structures *always* in a state such that the window can be redrawn. Failure maintain the state of the data structures can result, at worst, in crashes and, at best, in the famous "bits of window" left over syndrome. Failure to respond can produce the trails of windows immortalised in the title sequence of the "IT Crowd", or, on more recent versions, swathes of blank screen.

The summary elimination of local display refresh was short termism in extreme. By the time that there were machines fast enough to handle the X Window System, memory sizes had increased far faster than processing power and removed the only real objection to using local display refresh. This made the redraw policy used by the X Window System obsolete and ripe for sending to the museum of horrors.

The oddest feature of this was that X11 did have off-screen window buffers (for flicker-free window updates). It, therefore, had all the code required to write to off-screen window buffers and to refresh the display from these buffers. It just did not do it!

The bizarre architecture

The bizarre architecture of X was more the result of trying to patch the system into computers with inflexible, totally inadequate operating systems than of deliberately bad design. Moreover, as it was

6 I cannot find the reference. It might have been 1993. I cannot find the name of the professor.

7 http://home.comcast.net/~fbui/

designed for multi-vendor environments, it had to be designed to the lowest common denominator: you cannot get lower than Unix.

The X Window System has been described in glowing terms as being *modular*, having a *layered architecture* and providing *hardware abstraction*. These are all considered generally desirable in operating system software. X is modular in the sense that the amorphous mass of X is not integrated in any way with the operating system. X is layered in the sense that X provides a lump with the low level functions, protocol handling and the graphics device driver while the applications have to provide all the "upper level" functions. X does not provide any form of hardware abstraction (See Box 5).

The most bizarre feature of X was that, as X could not be implemented "properly" as a system component on the lowest common denominator operating system, it was implemented as a application program accessing the display hardware directly.

This anomaly has persisted right through to the present: "X is unlike any other subsystem of Linux in that the hardware-accelerated video drivers it uses are located within the X server, which is outside the kernel ... normally Linux drivers and vital subsystems such as keyboard, USB, filesystem, serial I/O, et cetera are all located inside the kernel"[8].

25 years later, there are signs that the X windows architectural legacy is finally being eliminated, even if its egocentric minimalist design philosophy design lives on.

The major lesson to learn from X is that, although it was built using a design philosophy that set out to minimise the functionality and implementation effort in order to make it compact, efficient and quick to implement, X turned out oversized, painfully slow and very long in gestation.

Box 5 - Hardware abstraction in X and Unix

Hardware abstraction is one of the conventional requirements of an operating system. Its ensures that applications programs do not themselves need to support all possible peripherals that can be used for a particular function. X, however, works only with a single type of hardware (a frame buffer type of display controller). The lack of hardware abstraction was always a very serious problem with X, in particular its failure to support printers. Various fixes were attempted (such as combining X with display postscript) but the trauma caused by the arrival of Microsoft Windows, which did have hardware abstraction allowing pages to be output to a printer using the same application code as used for writing to a window, provoked the creation of a parallel system "XPrint". This was released in X11R6.3 in December 1996, a decade after the first release of X11 and later removed from X.Org Server in 2008 because "X is not an API". More to the point, X had never had an applications program interface (API) for any operating system, because Unix did not support hardware abstraction in any meaningful way.

Just a minute, you might say, there is a lot of Unix system documentation that states clearly that Unix provides hardware abstraction. **Rubbish!** Unix started off life pretending that all peripherals were paper tape reader / punches and then changed this to all peripherals being files, which is even worse.

At the time X was being designed, communication with a Unix system was almost exclusively by "glass teletype" terminals: "during the late 1970s and early 1980s, there were dozens of manufacturers of terminals including DEC, Wyse, Televideo, Hewlett Packard, IBM, Lear-Siegler and Heath, many of which had incompatible command sequences" (Wikipedia).

In order to use these terminals for anything other than Teletype (scrolling text) emulation, each type of terminal had its own "protocol" (command sequences). So before, you start editing a file with emacs or vi, for example, you needed to type

```
set term=vt100          for a DEC VT100
set term=wyse60         for a Wyse 60
```

Quite a lot of emacs manuals say that these commands "tell Unix the type of terminal being used". **Rubbish!** Unix does not know about terminals. These commands set an "environment variable" to be accessed by an application, so that the *application itself* can generate the right protocols for the terminal. This requires every application to have its own protocol handling for every terminal *supported by the application*.

The whole purpose of hardware abstraction is to provide a well defined common interface and have the right drivers installed in the operating system so that applications only need to implement a common interface and do not have to have their own device handlers for each type of peripheral. This requires an operating systems interface (or API) that directly supports the whole range of functions that a particular class of peripherals can provide - totally contrary to the Unix minimalist approach.

8 http://home.comcast.net/~fbui/

1986 - The rise and fall of RISC

Computer hardware design had its own equivalent of the software minimalisation theories but rather more successful.

In 1986, when the first commercial RISC (Reduced Instruction Set Computer) processors were shipped, the concept was already fairly old. From the mid 1970s it had been observed that the compilers of the time were often unable to take advantage of features intended to facilitate machine coding and that complex addressing inherently takes many cycles.

The argument was that such functions would better be performed by sequences of simpler instructions that could be execute faster (in a single cycle). RISC became synonymous with the use of uniform, fixed length instructions with arithmetic only in registers and dedicated load-store instructions to access memory.

The first production processors based on RISC principles were the Sun Microsystems SPARC processor, based on the Berkeley RISC project, and the rather different ARM from Acorn. These were followed by the Intel i860/i960, the IBM/Motorola Power processors, and MIPS, DEC Alpha, etc. These RISC processors all had dedicated workstation architectures built around them but looking back from 2009, it is clear that the technology has not been successful at displacing complex instruction set computer architectures. Of the large number of RISC architectures produced, only the SPARC and ARM are still holding their own.

What was wrong with the concept and why were the SPARC and ARM particularly successful?

The problem with the concept was that it was a one-size-fits-all theory, so while there is a general explanation of its failure (Box 6), the two successes had radically different explanations (Box 7).

In the case of the SPARC, Sun Microsystem's long standing opposition to the "Axis of Wintel" as a marketing ploy has weakened of late and their commitment to the SPARC is looking less and less firm. The June 2009 Top500 list of supercomputers gives two Sun clusters in the top 10: both are x86 architecture. There is every sign that the SPARC is on its way out, leaving the ARM as the only survivor.

Box 6 - The RISC of failure

The argument at the time that compilers were unable to take advantage of the instruction sets of the mainstream processors is a polite way of saying "the C language and the standard C compiler were designed for PDP7 computers". The "back ends" of these compilers converted primitive PDP7 type instructions into the nearest equivalent groups of instructions on the target computers. They therefore generated excessive and often redundant code for CISC (Complex Instruction Set) computers.

This still tends to be true, but more recent C compilers for complex instruction set computers include "optimisation" to identify groups of instructions that can be reduced to fewer instructions. A recent C compiler for a CISC processor will normally generate fewer instructions than a compiler for a RISC processor, thus giving an advantage to CISC architectures in terms of code size and, therefore, instruction memory bandwidth required.

The argument that an instruction that includes complex addressing will take more than one cycle to execute is still valid but it is illogical as a justification.

On a RISC processor, it will still take more than one cycle, because it requires separate instructions for the addressing. Technology advances since the 1970s, however, tilts the balance firmly in favour of CISC as, with a pipelined architecture the address calculations can be performed by a small dedicated shift / multiply / add / fetch addressing unit at an earlier stage of the pipeline so that addressing cycles overlap instruction execution cycles.

The real reason for the failure of RISC, however, was already apparent by the time that the first RISC machines appeared. The QL processor was memory bandwidth limited, not instruction execution time limited. This was generally true of 1984 workstation technology. The potentially higher instruction execution speed of a RISC processor would give no advantage. The fixed width, rather wide instructions and the use of several RISC instructions to do the job of one CISC instruction increased the memory bandwidth required to execute most operations: a great disadvantage.

In the short term, the smaller core of a RISC processor made it possible to add instruction caches with burst accesses to the main memory to compensate for the increased bandwidth required. It was not long, however, before technology advances made instruction caches practical for CISC processors as well. Because the CISC core took more space, there was less space for the instruction cache, but, because CISC code was more compact, less cache space was needed.

As processor speed improvements consistently outstripped memory speed improvements over the next decades, any possible advantages of RISC architectures for general purpose workstations just became more and more hypothetical.

Box 7 - The RISC of success

The SPARC

The SPARC processor owes its existence and its survival to just one company: Sun Microsystems.

In the early days, SPARC processors really did yield performance benefits in the C / Unix environment where they were used. These processors could be regarded as C / Unix optimised in two ways.

The primitive SPARC instructions were well suited to code generation by early C compilers

The processor has a dedicated register stack to mitigate the inefficiency of the C function call conventions (the C function call conventions require that parameter values cannot be modified - this is efficient and simple where the stack is used for passing parameters, as required for the PDP7, but very inefficient where values are passed in registers in more modern processors as it requires the register values to be saved and restored).

The SPARC only has a small hardware window onto the register stack, everything above the window has to be pushed out to, and retrieved from memory. The larger the window, the more efficient the execution, but, unfortunately the more costly the task switching as the whole window has to be saved. This was not a problem for earlier Unix versions as Unix was a multi-user system with very expensive, very slow and infrequent task switching. This became a problem when native threads were introduced in an attempt to pass Unix off as a multitasking system.

The ARM

The ARM architecture seems likely to have a much better long term future. The ARM instruction set really only qualifies as RISC from the point of view of being designed for single cycle instruction execution. With features such as generalised conditional execution of each instruction, one ARM instruction can sometimes replace two CISC instructions. It would be more appropriate to consider it not as a Reduced but as a Regular Instruction Set Computer. The ARM architecture escaped the fate of real RISC architectures for two other reasons.

The first was the licensing strategy which propelled it into the domain of embedded and integrated processors. For embedded applications with much less RAM than a typical workstation, low power, fast, static RAM gives far higher bandwidth than power hungry, slower, dynamic RAM. When executing from on chip RAM and ROM the external memory bus bandwidth is not a performance limiting factor, making the bandwidth inefficiency RISC instruction set less important, so the compact, low power fast ARM core provides a very competitive solution, particularly in custom and semi custom chip solutions.

The second reason that the ARM architecture has come to dominate portable embedded and integrated systems is that the last pretence of RISC was abandoned with the introduction of the Thumb instruction set that packs the most used 32 bit instructions into 16 bits, thus reducing the bus bandwidth problems as well as bringing the code size closer into line with CISC processors.

Strategy

by Stephen Poole

It would be very incomplete not to include a game of strategy in this present series of classic computers games for the QL. Strategy games generally consist of a territory which has to be secured against an adversary, with both tactics and chance deciding the outcomes of the conflict.

So I decided to try to design such a game as simply as possible, using two players so as not to have to program the QL using Artificial Intelligence, which would greatly lengthen the code.

The territory is simply a grid, whose size is determined by the difficulty chosen. Each player (green or red), in turn chooses a position to try to gain (by hitting an across 'x' then down 'y' figure), and the QL then decides which player has the most force for that spot, and draws the winning player's letter colour and force figure on the grid. (To find out how to augment your force factor, study the listing!).

If you choose a location that has already been contested, you can still win it if you obtain a higher force than that which your opponent has already in place. If both adversaries get the same force, the QL beeps and the game moves on to the next player.

The force of each player for each try is shown at the bottom of the screen. Maximum force is equal to the difficulty level per grid. Once maximum force has been reached, a position is won irrevocably. When one player has won the biggest share of territory, the game is over!

As mentioned previously, there is a winning strategy, but I shall give you no clues!

25 Years - Part 3

by Tony Tebby

Welcome to the next part of our series. This time, we look at the time before Windows appears ... and this is where we continue in the next volume ... enjoy!

25 Years - 1988 to 1990 - Sci-Fi, Worms and Unix

Worms in the system

In 1988 viruses had been circulating freely attached to programs on PC diskettes for several years. For even longer, the academic and science fiction worlds had been toying with ideas for "free living" agents or worms that travelled around networks of computers carrying out maintenance tasks.

1988 - The Morris Worm

The Morris worm turned sci-fi into reality. This worm brought down many of the VAXs and Sun 3s running BSD Unix connected to the Arpanet. Different people have interpreted the incident in different ways, but for me the incident brought the degenerate state of the IT "establishment" to my attention. I am not referring to Robert Tappan Morris, the author of the worm, but the computer centre managers and system administrators of sites with Unix machines connected to the Arpanet.

The worm was analysed in a report[9] from Purdue University:

> Although UNIX has long been known to have some security weaknesses (*[citations going back to 1979]*), the scope of the break-ins came as a great surprise to almost everyone ... The most noticeable effect was that systems became more and more loaded with running processes as they became repeatedly infected. As time went on, some of these machines became so loaded that they were unable to continue any processing *[about 20 processes]*.

Why should this have been a great surprise to anyone? The risks had been known for about a decade. Even earlier, in 1975, Dennis Ritchie had written "The first fact to face is that UNIX was not developed with security, in any realistic sense, in mind"[10]. The machines affected were mainly in academic and research establishments and in the US Department of Defence and they were very lucky as the worm did not carry a malicious payload. Morris created and released the worm, but it was the computer centre managers and system administrators that created the conditions that allowed it to spread. It appears that, although Morris was convicted, no computer centre manager or system administrator lost his job or was convicted of criminal negligence for connecting Unix systems directly or indirectly to a public network. This failure to take action against those who were really responsible for the damage caused by the attack was to have serious consequences.

1990 - Desert Storm

Two years after the Morris Worm, the US launched Operation Desert Storm and Operation Desert Shield. The ground had been well prepared: in the intervening period, a group of hackers had placed backdoors in a number of military, DoE and DoD servers, using known Unix security flaws, giving them almost complete freedom to access the US military networks. This time the "worms" had payloads, not to destroy the systems, but to download everything about US military plans and capabilities. It appears that the operation was run for profit from the Netherlands, but it was not profitable as they could not find any buyers for the information. Had all the potential buyers been there first?

When I first heard about this story, I had my doubts, surely those responsible for US military networks could not be so stupid or negligent that they continued to use Unix machines? But then I came across the US Navy Computer Incident Response Guidebook[11]

9 http://homes.cerias.purdue.edu/~spaf/tech-reps/823.pdf
10 http://www.tom-yam.or.jp/2238/ref/secur.pdf
11 http://all.net/books/ir/nswc/P5239-19.html

> Many of the cases of unauthorized access to U.S. military systems during Operation Desert Storm and Operation Desert Shield were the manifestation of espionage activity against the U.S. Government.

Apparently, the "espionage" attacks were not the majority of "unauthorized accesses" and that does not include the intrusions that were not detected - it is just mind boggling.

Everyday worms and other attacks

Worm attacks continued, carrying payloads such as trojan horses, key loggers and denial of service co-ordinators. The situation took a turn for the worse when the world's largest software supplier abandoned its own system software in favour of a remodelled Unix system. With the arrival of Windows NT, Microsoft's systems became vulnerable to the same type of attacks as mainstream Unix, culminating with the "Code Red" worm in July 2001 which cost computer users an estimated $3B.

The Morris and Code Red worms exploited fundamental design weakness in the Unix / C execution model to break programs executing on the target computer, but other attacks exploited human fallibility.

In 2001 / 2002, Gary McKinnon "scanned a large number of computers in the .mil network, was able to access the computers and obtained administrative privileges ... McKinnon would then use the hacked computer to find additional military and NASA victims" (US Department of Justice). With administrative privileges, he could have done enormous damage, but he was only "looking for evidence of UFOs". How did he get in? Just by using remote login with blank or obvious passwords!

In 2003, H. Orman wrote, in The Morris Worm: A Fifteen-Year Perspective[12],

> Today, the Morris worm is remembered as the first of many such attacks, as what might have been a wake-up call to system administrators and security researchers, and as the first certain signal to those who still held utopian beliefs about the Internet, that it was not to be a friendly place.

"Might have been a wake up call", but it was not. System administrators just dozed on with "business as usual".

> As the years passed, knowledge about subverting Unix access permissions abounded and spread. The number of loopholes, and their varieties, had begun to look unmanageable to many system administrators and computer-security experts. Two camps developed, one hoping to fix all the problems, and another advocating keeping one step away from the Internet.

Why only two camps? Surely the obvious solution was to ditch Unix. A number of attempts were made to develop "replacements", but, in general these were Unix rebranded, Unix restructured or Unix rewritten. Even though there were numerous private attempts to create "better" systems, there were no significant attempts to deploy systems that were radically more efficient, radically cleaner, radically more predictable or radically more secure.

1990 - Plan 9

Plan 9 (the title of a sci-fi film) was the Bell Labs Unix team's attempt at getting it right the second time around. What is most striking was the difference between quick and dirty design of the original Unix and the X Window System, and the "we are going to get it right" philosophy of the Plan 9 development. The story in Box 8 is as it is told by the developers[13]. Remember, when you read this text, it was written by the creators and developers of Unix: there is, therefore, a certain bias! I did not write Unix, therefore there is a certain bias in my comments.

Apart from the "Unix centric" approach, this discussion starts off very reasonably, explaining why the central time sharing concept of the 1960s had lost favour and why the personal computer systems that were replacing it were, themselves, far from satisfactory <u>for large organisations</u>. Then it slips away and they end up aiming to build straightforward, archaic 1960s time sharing systems with central machines and terminals! Or rather they aimed to emulate this archaic systems architecture using cheap or not so cheap microcomputers.

12 IEEE Security & Privacy September/October 2003
13 http://plan9.bell-labs.com/sys/doc/9.html

As the discussion goes on it becomes clear how little functional difference there was between Unix and Plan 9: the same kernel concept, the user based access rights, the same hierarchical file system *and the same vulnerabilities.*

Scattered throughout the text are the keywords "complete" and "consistent". Compared to the earlier Unix and X Window System philosophies of not caring about either, this is a welcome return to rigorous design. The authors noted, with some surprise, that taking a general, consistent design approach actually saved time when it came to extending the capability. Why should they have been surprised? Because by the time they came to extend Plan 9, the belief that rigorous, complete and consistent design can save you time had come to be considered as a sign of incurable mental illness or senility (see Worse is Better).

Some quotes (circa 1991) from Ken Thompson and Rob Pike of Unix and Plan 9 Fame.

Object oriented programming

- Object-oriented design is the Roman numerals of computing
- We have persistent objects: they're called files

Structured programming

- If you want to go somewhere, goto is the best way to get there

Unix

- Not only is Unix dead, it's starting to smell really bad.

X

- The X server has to be the biggest program I've ever seen that doesn't do anything for you.

With attitude like that, they cannot be all bad.

Box 8 - Plan 9 rationale and implementation

Plan 9 began in the late 1980's as an attempt to have it both ways: to build a system that was centrally administered and cost-effective using cheap modern microcomputers as its computing elements. The idea was to build a time-sharing system out of workstations, but in a novel way. Different computers would handle different tasks: small, cheap machines in people's offices would serve as terminals providing access to large, central, shared resources such as computing servers and file servers. For the central machines, the coming wave of shared-memory multiprocessors seemed obvious candidates *[see "shared memory multiprocessing" below]*. The early catch phrase was to build a Unix out of a lot of little systems, not a system out of a lot of little Unixes.

By the mid 1980's, the trend in computing was away from large centralized time-shared computers towards networks of smaller, personal machines, typically Unix 'workstations' *[How isolated from reality can you get? Unix had negligible penetration outside the academic world]*. People had grown weary of overloaded, bureaucratic timesharing machines and were eager to move to small, self-maintained systems, even if that meant a net loss in computing power *[Most people, as they were not using Unix, saw a massive net gain in computing power]*. As microcomputers became faster, even that loss was recovered, and this style of computing remains popular today.

In the rush to personal workstations, though, some of their weaknesses were overlooked. First, the operating system they run, Unix, is itself an old timesharing system and has had trouble adapting to ideas born after it. Graphics and networking were added to Unix well into its lifetime and remain poorly integrated and difficult to administer *[the comment is also true of MSDOS, the dominant workstation OS at the time]*. More important, the early focus on having private machines made it difficult for networks of machines to serve as seamlessly as the old monolithic timesharing systems. Timesharing centralized the management and amortization of costs and resources; personal computing fractured, democratized, and ultimately amplified administrative problems *[this fails to draw the very important distinction between centralised data and centralised processing - and their relative merits in different types of organisation - another example of one-size-fits-all]*.

...

The problems with Unix were too deep to fix, but some of its ideas could be brought along. The best was its use of the file system to coordinate naming of and access to resources, even those, such as devices, not traditionally treated as files. *[Ouch!]* ... our laboratory has a history of building experimental peripheral boards. To make it easy to write device drivers, we want a system that is available in source form *[what system, other than Unix, required you to have the OS source in order to write device drivers?]* ... [Instead of] normal Unix-style processes and light-weight kernel threads, Plan 9 provides a single class of process but allows fine control of the sharing of a process's resources such as memory and file descriptors *[Just like Stella and not far from Domesdos c.f. Box 2]*. A single class of process is a feasible approach in Plan 9 because the kernel has an efficient system call interface and cheap process creation and scheduling *[but still orders of magnitude slower than Stella]*.

JOCHEN MERZ SOFTWARE

Kaiser-Wilhelm-Str. 302
47169 Duisburg, Germany

Fax +49 203 502012
EMail: SMSQ@J-M-S.com

QPC2 Version 3 + SMSQ/E Software QL-Emulator for PC's		**EUR 59,90**
QPC2 Version 3 - Upgrade from QPC2 Version 2		**EUR 19,90**
QPC2 Version 3 - Upgrade from QPC2 Version 1		**EUR 39,90**
SMSQ/E ATARI or (Super)GoldCard or QXL		**EUR 39,90**
QPC Print - printer emulation driver for QPC		**EUR 39,90**
Agenda Agenda program for WMAN and Prowess	[V1.09]	**EUR 14,90**
Suqcess Database front-end for WMAN	[V2.05]	**EUR 19,90**
QD2003 Pointer-Environment-Editor	[VB.01]	**EUR 39,90**
QD2003 Upgrade from QD98	[VB.01]	**EUR 9,90**
QD2003 Upgrade from previous versions	[VB.01]	**EUR 19,90**
QMAKE Pointer-driven MAKE for GST/Quanta Assembler	[V4.31]	**EUR 14,90**
BASIC Linker	[V1.21]	**EUR 14,90**
WINED Floppy/Harddisk Sector- & File-Editor	[V1.26]	**EUR 14,90**
FiFi II File-Finder - Extremely useful!	[V4.31]	**EUR 14,90**
FiFi II Upgrade from Fifi V1, 2 or 3	[V4.31]	**EUR 9,90**
EPROM Manager	[V3.02]	**EUR 14,90**
QSpread2003 Spreadsheet Program	[V4.04]	**EUR 39,90**
QSpread2003 Upgrade from QSpread2001	[V4.04]	**EUR 9,90**
QSpread2003 Upgrade from V1	[V4.04]	**EUR 29,90**
QPAC I Utility programs	[V1.11]	**EUR 19,90**
QPAC II Files, Jobs & other Things	[V1.45]	**EUR 29,90**
QTYP II Spell checker	[V2.17]	**EUR 24,90**
QPTR Pointer Toolkit	[V0.30]	**EUR 39,90**
DISA Interactive Disassembler	[V3.04]	**EUR 29,90**
typeset-ESC/P2 text87 driver for all ESC/P2 printers (incl. Stylus)		**EUR 24,90**
CueShell	[V2.14]	**EUR 29,90**
CueShell for QPC	[V2.14]	**EUR 20,00**
SER Mouse software mouse driver for serial mice		**EUR 10,00**
EasyPTR Version 4	[V4]	**EUR 59,90**
EasyPTR Version 4 - Upgrade from earlier versions	[V4]	**EUR 39,90**
text87plus4patch - now for QPC, QXL, Q40, Q60, Aurora		**EUR 10,90**
QDT - QL Desktop program		**EUR 59,90**
QMENU Version 8 - New! with new, printed Manual	[V8.02]	**EUR 24,90**
QMENU Version 8 - Update from earlier Versions, also with printed.manual		**EUR 17,90**
QMENU Version 8 - New/Update for QL Today subscribers, with prtd manual **ONLY**		**EUR 14,90**

Please add EUR 4,50 for postage EUROPE, or EUR 6,50 for postage REST OF WORLD

We accept VISA, MasterCard & Diners Club online and offline! Amex only by mail or fax, not email!
New payment methods for our customers: Money transfer to "local" account in many countries!

- Deutschland: Jochen Merz, Account 493 50 431, Postbank Essen, BLZ 360 100 43
- Österreich: Jochen Merz, Account 85055317, PSK Wien, BLZ 60000
- Switzerland: Jochen Merz, Account 60-690080-4, PostFinance, Clearing-Nr. 09000
- The Netherlands: Jochen Merz, Gironummer 3258439, Postbank NL Amsterdam
- and from all other countries in EUR with IBAN and BIC to account
  Jochen Merz, Deutsche Postbank AG, IBAN: DE21 3601 0043 0611 1004 37 / BIC: PBNKDEFF 360
- UK customers can pay in £ (convert EUR prices above to £ by multiplying with 0.90) to
  Jochen Merz, Account 83795395, Citibank UK, Sort code 30-00-45
  or send cheques in £ - no fee for UK sterling cheques!
- US customers can pay in US$ (convert EUR prices above to US$
  by multiplying with 1.30) - no fee for US cheques in US$!

Cheques payable to Jochen Merz only!

Price list valid until 30th of July 2010

25 Years - 1991 to 1994 - Better or Worse, Windows, Linux

25 Years - 1991 - The Rise of "Worse is Better"

"The rise of worse is better" is a chapter of Richard Gabriel's lament for Lisp[14] published in 1991. This chapter, which circulated as a article in its own right, was clearly intended to be provocative and he adopts the devil's advocate method of argument. It has often been quoted as the definitive explanation for the dominance of very, very, bad system software from the mid 1980s onwards.

He attempted to explain why, although Unix and C are clearly "worse" than "his" AI operating system and Lisp compiler, the Unix+C pair was apparently more successful. He compared the "New Jersey" (Bell Labs) "Worse is Better" philosophy with the MIT "Right Thing". These two philosophies are detailed in Box 9.

The New Jersey approach is described as providing a quick and dirty, incomplete solution which is then evolved by armies of developers working independently: ultimately the best solution is that which attracts the most developers. MIT "Right Thing" approach is described as leading to the interminable development of slow, overly complex, oversized, inefficient programs.

There are some serious flaws in the arguments. The MIT / New Jersey distinction is nonsense. The X Window System was quite deliberately designed breaking every one of his "Right Thing" rules - even more so than Unix - and X was created at MIT - whereas Plan 9 was designed using a strict "Right Thing" philosophy - and Plan 9 was from New Jersey.

The comparison between C and Lisp is no better. Two years before the article was written, C had become mature enough to be standardised as ANSI X3.159-1989, "only" 20 years after its creation, whereas Lisp, created 10 years before C, was still evolving and would not be standardised until three years later as ANSI X3.226-1994. Lisp is, therefore, a better example of the "New Jersey" approach than C.

Box 9 - MIT versus New Jersey by Richard Gabriel

The essence of the [MIT/Stanford] style can be captured by the phrase **the right thing** ... it is important to get all of the following characteristics right:

- Simplicity – the design must be simple, both in implementation and interface. It is more important for the interface to be simple than the implementation.
- Correctness – the design must be correct in all observable aspects. Incorrectness is simply not allowed.
- Consistency – the design must not be inconsistent. A design is allowed to be slightly less simple and less complete to avoid inconsistency. Consistency is as important as correctness.
- Completeness – the design must cover as many important situations as is practical. All reasonably expected cases must be covered. Simplicity is not allowed to overly reduce completeness.

I believe most people would agree that these are good characteristics. I will call the use of this philosophy of design the MIT approach

The [Unix] worse-is-better philosophy is only slightly different:

- Simplicity – the design must be simple, both in implementation and interface. It is more important for the implementation to be simple than the interface.
- Correctness – the design must be correct in all observable aspects. It is slightly better to be simple than correct.
- Consistency – the design must not be overly inconsistent. Consistency can be sacrificed for simplicity in some cases, but it is better to drop those parts of the design that deal with less common circumstances than to introduce either implementational complexity or inconsistency.
- Completeness – the design must cover as many important situations as is practical. All reasonably expected cases should be covered. Completeness can be sacrificed in favor of any other quality. In fact, completeness must sacrificed whenever implementation simplicity is jeopardized. Consistency can be sacrificed to achieve completeness if simplicity is retained; especially worthless is consistency of interface.

14 http://www.dreamsongs.com/NewFiles/LispGoodNewsBadNews.pdf

His description of the two basic *right thing* development scenarios is even more peculiar.

> The big complex system scenario goes like this: First, the Right Thing needs to be designed. Then its implementation needs to be designed. Finally it is implemented. Because it is the Right Thing, it has nearly 100% of desired functionality, and implementation simplicity was never a concern so it takes a long time to implement. It is large and complex. It requires complex tools to use properly. The last 20% takes 80% of the effort, and so the right thing takes a long time to get out, and it only runs satisfactorily on the most sophisticated hardware.
>
> The diamond-like jewel scenario goes like this:
>
> The right thing takes forever to design, but it is quite small at every point along the way. To implement it to run fast is either impossible or beyond the capabilities of most implementors."

In a rebuttal[15] Richard Gabriel did point out the he was not supporting "Worse is Better" and he came close to apologising for calling Unix a virus, but his premise remained intact: trying to design software is totally futile - whatever you do it will turn out oversize and underperforming and the more complete, coherent and correct you try to make it, the longer it will take to release.

He does not appear to consider the classic "Right Way" scenario.

> First, the Right Thing needs to be designed. Then its implementation needs to be designed. Finally it is implemented. Because the scope is known from the start it has 100% of the design functionality. Because no short cuts are taken on implementation, no reworking is required, saving time. It is compact, clearly structured and efficient. The regularity of the interface makes it easy to use. Because most of the time is spent building solid foundations, the last 80% is a downhill run so it finished quickly and the system needs only the minimum of resources to run.

It is clear from the Plan 9 article cited above, that the authors of Plan 9 believed that they had implemented The Right Thing in the Right Way. Having been involved in dozens of development projects (for quite a few, called in at the last minute panic stage), I have to admit that projects using a Right Thing design philosophy may not always follow the Right Way scenario, but they will never result in a big complex system. The most likely problem is that doubts set in half way through the project when there is still (quite rightly) no visible sign of life. At this point there is a risk the the project veers off into the quick hack ("Worse is Better") approach, the software starts ballooning, the errors start multiplying and the completion date starts receding. What is worse, this is justified as "pragmatism" to get the system out "even if it is not perfect".

Richard Gabriel also illustrates these two development philosophies on a specific example: the "PC loser-ing" problem. The MIT approach to solving the problem was to add complex and fragile code, after the problem had been identified, to give the "right" results. The Unix "Worse is Better" approach just gets it wrong but flags the error so that applications can take the necessary corrective action (or else crash).

The MIT solution was clearly not the "Right Thing" because if the system had been designed according to the Right Thing philosophy (the correctness rule), the problem would not have occurred: Domesdos was designed to be "correct" so "PC loser-ing" was impossible. This took some thought in design but, because there was no "tricky" code or extra error handling, it was faster to implement, it was more compact, it was faster in execution and the applications did not need to allow for an error that could not occur: in this case, it was the "Right Thing" done the "Right Way".

Finally, he concludes that the Worse is Better approach wins over the Right Thing because the Right Thing delays the release and winner is the first to be released, no matter how incomplete, incoherent or difficult to use it may be. Even on this he is clearly wrong, Lisp was widely used a decade before anyone outside Bell Labs had ever heard of C, but Lisp lost out to C.

He seems to have missed out on the point: that, in the beginning, Unix and C were freeware. Even if they were worth less than you paid for them, most organisations' accounting methods did not, and still do not, consider time wasted as an expense. In academic establishments, time wasted is "research". Lisp and Lisp machines cost money - they never stood a chance.

15 http://dreamsongs.com/Files/worse-is-worse.pdf

Behind the rhetoric, there is, however, something far more sinister: in computer science courses, around the world, students were being taught that writing correct, complete, coherent software was either impossible or not worth doing and that users are unimportant.

A year before Richard Gabriel's rant, Professor A Tannenbaum published his "Modern Operating Systems" in which he wrote on the evolution of operating systems "this evolution is very similar to the evolution from assembly language programming, where the machine came first, to programming in high level languages, where the programmer came first – programmers are no longer able to write compact, efficient software[16]".

This is an extraordinary "insider view" of computer science. In the "bad old days" when programmers were able to write compact, efficient *and reliable* software, they did not write this compact, efficient *and reliable* software for the machine itself, but wrote compact, efficient *and reliable* software *for the users of the machine*.

1991 - The Rise and Fall of Virtual Memory

Although Unix workstations had suffered from virtual memory for many years, it was only introduced to mass market personal computers in Mac System 7 in 1991 and Windows 3.1 in 1992

There are three popular views of virtual memory and at least one controversial view.

1. Virtual memory allows you to execute programs that need more than the physical memory of the computer.
2. Virtual memory increases hardware costs and reduces performance.
3. Virtual memory allows the performance to degrade gracefully as demand for memory outstrips supply.
4. Virtual memory increases the demand for real memory.

The idea of virtual memory being virtually unlimited is a simplistic view believed by too many software developers, but it is vaguely true, provided you do not care at all about usability.

That virtual memory is costly is inescapably true. Virtual memory requires additional hardware over and above that required for memory access protection and all accesses to data and programs are slowed down - even under the most favourable conditions.

The concept of degrading gracefully depends on your definition of "graceful". 25 years ago this was a idealistic view of virtual memory, but is now completely unrealistic.

That virtual memory increases the demand for real memory is totally contrary to the theory, but observably true. There must, therefore, be something wrong with the theory.

The failings are discussed in Box 10.

The end of virtual memory in sight

In 2009, there seems to be a certain amount of dispute amongst "tweekies" as to whether it is a good idea turning off virtual memory on systems such as Windows.

Some tweekies recommend it, others trot out conventional wisdoms such as "you gain no performance improvement by turning off the pagefile", clearly without trying it. My own experience on a rather memory limited portable is that overall performance is significantly better as I no longer suffer from slow motion windows. Most software that I use runs perfectly. The only significant problem is that, from time to time, Firefox gives up with out of memory, usually when it is just sitting around doing nothing. This is not serious, as it can be restarted, re-opening all the tabs while occupying a few hundred megabytes less memory.

With diskless computing becoming more common, we can possibly look forward to a future without virtual memory. In the shorter term, double your RAM every one or two years to keep ahead of virtual memory guzzlers.

16 This is my own translation of the French translation of a book written in English by a Dutchman.

Box 10 - The failings of virtual memory

Disgraceful degradation

Virtual memory does allow programs executing to take more memory space than is really available. The principle is that not all the memory space taken is actually used, having the unused or little used space "swapped out" to disk releases valuable "real memory" for use. The memory is divided into pages. If a page is in memory, it can be accessed almost as fast as if real memory were used. If there is a "page fault" because the processor tries to access a page that has been swapped out, there is a long delay. The delay does not depend on the memory overload, but the page fault rate does.

25 years ago, a typical average disk access time was about 40ms with memory cycle times of about 400ns: a ratio of about 100,000 times. A one in ten thousand page fault rate would have caused a computer to run 10 times more slowly than the "benchmark" speed: this could be considered graceful degradation.

In 2009, an optimistic typical average disk access time is about 6ms with Ll cache access times of 1ns: a ratio of about 6,000,000 times. A one in ten thousand page fault rate will cause a modern desktop computer to run 600 times more slowly than the "benchmark" (L1 cache) speed: this cannot be considered to be graceful degradation.

What is the memory overload that will cause a one in ten thousand page fault? There is no simple answer as it depends on the details of the memory usage of all the programs executing. Experience with a fairly large variety of PCs from Windows 95 onwards (business and private usage) has indicated that users tend to think that their machine is suffering from viruses (slow, erratic execution) when the memory overload reaches 20%-30%.

For at least a decade, the standard cure for PC performance problems has not been to increase the processor speed, but to increase the size of the RAM to avoid using virtual memory.

The costs of virtual memory

Virtual memory requires a dynamic address translation unit (more commonly and incorrectly known these days as a memory management unit, MMU, although it does not do any memory management). This translates the "virtual addresses" seen by the processor into the real memory addresses and raises a page fault error if the virtual address is not in real memory.

25 years ago, MMUs were more expensive than processors. In 2009, they are built into the processor bus interface and, possibly, account for less than 20% of the processor cost.

The address translation in the MMU is not free, it takes time to calculate the real memory address (if there is one) for each and every access to data or program. 25 years ago, this typically added 20% to the memory access time under the most favourable conditions. In 2009, it not only adds overheads to external memory accesses but complicates caching as well.

Virtual memory increases the demand for real memory

"We all" know that this is true, but why?

The glib answer is in Parkinson's law as applied to computer memory: "programs expand to fill the available memory". The result is that the virtual memory space fills up, overloading the real memory. Is this inevitable? Why should this happen? Why should this be serious?

On introducing virtual memory, there is an immediate effect and a more pernicious longer term effect.

In the the short term, memory overload occurs with applications written BVM (before virtual memory) for two main reasons.

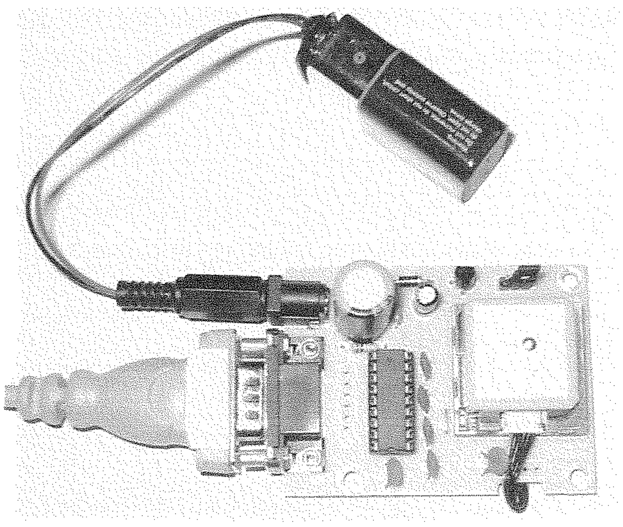
1. Well behaved software, such as the older versions of Microsoft Word, will check whether there is enough room before taking extra temporary working memory. Without virtual memory, fall back methods are used if there is not enough real memory. With virtual memory, extra memory always seems to be available so it will be taken and the availability of virtual memory will automatically increase the demand for real memory.
2. With virtual memory, users can start applications for which there is not enough room in real memory without stopping other applications. This results in instant memory overload which can make it almost impossible to remove any of the applications as the machine has become so slow. (In one extreme case I had to deal with a PC that that had been "corrupted" by a minor software upgrade: it took more than 25 minutes to start up - the user thought it was dead, but it was just a simple memory overload).

In the longer term, memory overload occurs with applications written AVM (after virtual memory) because many developers have adopted the attitude that memory is unlimited and they just do not care.

Improving QL Emulator I/O, Oscilloscope

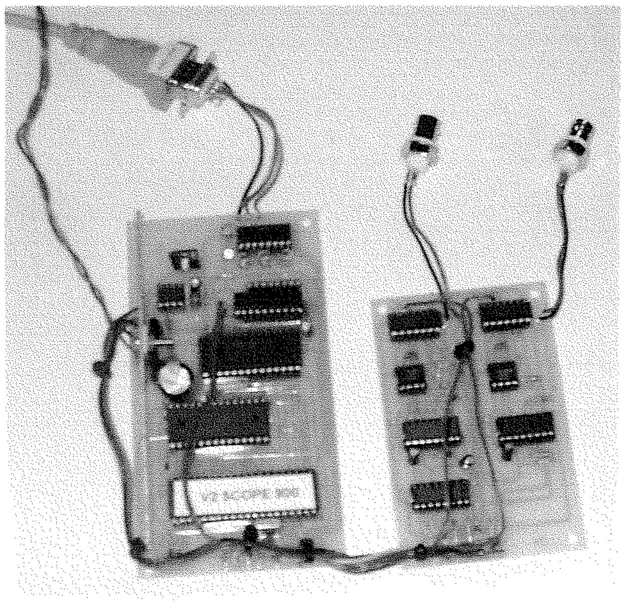
by Ian Burkinshaw

[Editor:] First, two apologies to Ian: one for having held back his second article for two issues - but there's a good reason for this: we do not have very many hardware articles, so we thought it would be a good idea to spread it a bit.
When I started with the layout of this article, I noticed that I used the wrong picture to go with Ians GPS article in Issue 1 of Volume 14.
The correct picture, which should have been printed together with the article of issue 1, is shown to the right.



And now enjoy the next of Ians very interesting hardware articles...

In my first article on improving the QL emulator I/O published in volume 12, issue 3 of QL Today I showed how you can get parallel inputs and outputs via the RS232 port on your PC. As promised in that article here is the second part dealing with the oscilloscope project. This, in the event took me longer than I expected, sorting out routines to carry out the FFT (Fast Fourier Transform). More on this later.
The hardware for this project was originally published in the August and September 2007 issues of Everyday Practical Electronics magazine. The project has two PCB's one handles the analogue processing and contains the analogue to digital converters. There are two channels on this card so you have a two channel oscilloscope as a result. It is not impossible to change the entire project to deal with more channels. However you would need to have some PIC programming experience to achieve this. 8 channels should be possible. The basic hardware, with additional input cards can support this. The second card contains the memory and a PIC processor to take the streams of data produced by the analogue to digital converters in to a RS232 stream. There is a lot more data here to be handled, compared to the parallel I/O project. Because of the higher data rates involved, 57,600 baud is required. A PC fitted with a RS232 port or if your PC does not have one then you can use a USB to RS232 converter. However please note, most QL original hardware will not run at these speeds. The analogue to digital converters are able to handle signals up to 40KHz. Not very high by today's standards but for simple audio type projects it is good enough. The signal processing is 8 bit so is not CD quality. Also it is more a storage oscilloscope. I will explain. There is also a mode you can put the hardware into which will make the oscilloscope a 8 bit digital analyser, this aspect I will not be covering in this article, but is not difficult to achieve. As I have said before we are supposed to be tinkerers. It is not my aim to give you a total solution but give you ideas.



The right picture for this article

25 Years - Part 4

by Tony Tebby

1992 - Microsoft Windows

The first three versions of Windows marked a gradual drift from the 'add on windowing' of the Atari ST, the Amiga and X, towards integrated windowing as found on the Apple Mac.

Windows 1, 2, 3

Version 1 was started before Domesdos, although it was not released until 1985. It had very little success. Version 2 was a minor upgrade, with hardly any more success. Version 3, grafted on MSDOS, was somewhat better, but it was not until Microsoft released Version 3.1 in 1992, with TrueType fonts, that Microsoft had a competitive GUI with scalable, proportionally spaced fonts, virtual memory and cooperative multitasking, slightly after Apple released system 7 for the Mac with a similar specification.

Microsoft has often been accused of being the king of bloatware, but Windows 3.1 worked quite happily on a machine with 2 MByte of RAM - only 3 times larger than a fully expanded QL.

Windows network

Microsoft SMB (Server Message Block) is a peer-to-peer networking system that was built into Windows for Workgroups in 1992 to provide shared access to files and printers. Some hardy QL types might remember the TK2 network for the QL: although the raw performance was rather limited (about 100 times slower than the Ethernet of the period) by the network hardware which comprised a one bit software driven I/O port without even an interrupt facility, it did actually work, serving files over the network, sharing printers, even providing clunky instant messaging. The QL network was derided, not because it was slow (and because of the extreme hardware limitations it was sometimes slower than Novell on a PC with Ethernet) but because it was peer-to-peer which was considered a poor relation of the established Unix dedicated server architecture.

Microsoft, bless their hearts, had realised that the dedicated server approach was a pretty stupid solution for ordinary office requirements and provided a peer-to-peer system that worked fairly well and still does – moreover, like the TK2 network, it does not preclude the use of dedicated servers, so you get the best of all worlds.

SMB is the most important Microsoft technology to be built into Linux.

Windows NT 3

Windows NT must be considered to be the sort of brainstorm error that can only be made by companies that have entirely lost contact with reality.

When it was released in 1993, Bill Gates said that NT stood for New Technology, but, although it did incorporate some peripheral technology from the early 1970s (Internet Protocol, Alto-derived GUI), for the rest it was just another system based on recycled 1960s theories. Other explanations given for NT is that Windows NT started off life as WNT (one letter up from VMS Dave Cutler's system for the VAX) or N-Ten – the development system used.

Is Windows NT really different from Unix or is it just another flavour of Unix? The parentage is in no doubt. Both NT and Unix are descended from Multics: Unix via UNICS, NT via OS2 on one side and via VAX/VMS and Mica on the other. Mica was designed to be Unix (BSD and Posix) compatible, NT was designed to 'resemble Unix'.

The defining differences between Multics and UNICS were the elimination of multiprocessor support, the radically simplified multi user security model and the hierarchical filing system. As Unix developed, the multiprocessor support was restored and the 'process / thread dichotomy' (to use the Plan 9's developers' polite term) was added. NT has the hierarchical filing system and process / thread dichotomy of Unix with a slightly improved multi user security model which is, however, still closer to Unix than to MULTICS.

Even Microsoft's senior management is unsure whether NT aims to be better than Unix or a better Unix: 'The day I come in front of a Gartner audience and say I have a better Unix than Linux, that'll be a good day.' (Microsoft's CEO Steve Ballmer, October 20, 2005).

So is NT is a bastardised Unix, an evolved Unix; or a failed Unix copy? It was intended to be POSIX compatible (i.e. more Unix than Unix) and Microsoft's detailed list of differences between NT and mainstream Unix is only two pages long (which is a bit shorter than the 3000 pages describing one minor Linux revision).

How was it received? Very badly! The workstation versions were targeted as business use: they were just a bad joke. The server versions did not make much of inroads into the Unix dominated market.

1994 - Early Linux development

Although the first full version dates from 1994, Linux version 0.01 was released in September 1991; it had 10,239 lines of code.

Linux 0

It is widely reported that Linux grew out of Minix, but scratching around, I can find little evidence for this although there was some borrowing. Minix was created by Prof. A Tanenbaum, a fanatical believer in semaphores, microkernels and "burying interrupt handling as deeply as possible in the operating system".

Linux was much more like the original UNICS: Linus Torvalds defended the monolithic kernel vigorously against attacks by Prof. Tanenbaum, he had much to say about the mental health of anyone who suggested improving Linux by using semaphores and he wrote, right at the start, "I also happen to LIKE interrupts, so interrupts are handled without trying to hide the reason behind them" (comp.os.minix).

Setting aside the minor fact that in the ensuing (very public) argument Prof. Tanenbaum was provably wrong on almost every count, Linus Torvalds' approach was vindicated when Linux started displacing "advanced" versions of Unix, with their semaphores, microkernels and the rest of the 1960s theoretical junk, simply on the basis that Linux worked better – as should have been expected by anyone who bothered to do a proper theoretical analysis (or by anyone who really understood computer systems).

Linux 1

Version 1, including the X Window System, was the first complete Linux version, which appeared in 1994, 2 years later than announced.

Although it was free, the take-up was minimal outside the academic world. It did not usually work when installed, even if the user managed to install it. The shortage of device drivers meant that Linux could only be used with very basic PC configurations. The situation improved gradually over the next decade.

1995 to 2000 – Wirth's law and its proof

1995 - Wirth's Law

"Software is getting slower more rapidly than hardware becomes faster"[17]

This is the core of Niklaus Wirth's Plea for Lean Software. Unfortunately it really comes out as a lament for his Oberon operating system and programming language. In the same way as Richard Gabriel could only explain the general preference for Unix and C over his operating system and language by resorting to the "Worse is Better" proposition, Niklaus Wirth could not understand why the world seemed to prefer bloated software to his neat, compact operating system and language.

The "law" was based on a comment in Martin Reiser's preface to the Oberon System Manual, "The hope is that the progress in hardware will cure all software ills. However, a critical observer may observe that software manages to outgrow hardware in size and sluggishness". As this was written at the latest in 1991, it can only refer to mainstream Unix, not, as many people have claimed, Windows NT.

17 A Plea for Lean Software, Computer, vol. 28, no. 2, pp. 64-68, Feb. 1995

Unfortunately, in the IT world, only cranks and those living in the past do not know that it does not matter how inefficiently software is written because you can always get a more powerful computer. He is reputed to have said with self deprecating humour "Whereas Europeans generally pronounce my name the right way ('Nicklouse Veert'), Americans invariably mangle it into 'Nickel's Worth'. This is to say that Europeans call me by name, but Americans call me by value".

If, by this comment he meant to imply that his ideas on software neatness were not appreciated the other side of the Atlantic, the USA / Europe software divide has no more value than Richard Gabriel 's MIT / Bell labs divide. Europeans are as much responsible for bloat as our American cousins. On the other hand, it might have been a joke about parameter passing.

1995 - Microsoft Windows Grows

With the failure of NT 3 to achieve any significant sales, part of Microsoft pressed on with making products to sell.

Windows 95

Windows 95, released in 1995, was radically different from the 1,2,3 line, but was compromised to maintain compatibility with the earlier versions. Although the only "headline" architectural difference between Windows 95 and Windows 3.1 was the "pre-emptive multitasking", the system was much better integrated and the windowing system was WIN32: Windows 3 that had been re-written for NT 3 and then carried back to Microsoft's home operating system.

It also marked the start of Microsoft bloatware: although it was not in the same league as Unix bloatware (requiring about a fifth of the RAM and a quarter of the processor speed of Unix / X for an equivalent workload), it did gobble up about 4 times as much RAM as Windows 3.1 (for a very small increase in functionality).

The taskbar (which had appeared on the QL as the rather primitive "Button bar" (1986), re-cycled as the Acorn Arthur "Icon bar" (1987), the NextStep "Dock" (1989) and later patented by Apple in 2008) made its first appearance on a mainstream computer system. Unlike most other manifestations, it was, in true Microsoft "hedging-their-bets" style, not a replacement for the desktop icon mess, but an adjunct. It might not have been very original, but it was quite well done.

Windows NT 4

The best thing about Windows 95 was that it was not Unix. The Unix clan within Microsoft, however did not give up and they had support right from the top. In 1996, Microsoft launched Windows NT 4. This inherited the Windows 95 user interface and it had Microsoft's Internet Information Server as well as the SMB peer-to-peer network. For servers, the rapidly increasing computer RAM sizes and processor speeds meant that the stunning inefficiency of NT was becoming less of a problem. IIS proved to be competitive with mainstream Unix internet protocol servers (and as vulnerable) and in combination with SMB for local file serving, NT 4, with its far higher level of integration and coherence than any Unix system, started to be adopted for "enterprise servers". It was significantly helped by the inability of the Unix world to get to grips with the mass changeover, in the 1970s, to displays and keyboards with both upper and lower case letters.

As a workstation operating system, NT 4 was a disaster. Its customers, both business and personal, preferred Windows 95 and Microsoft was forced to continue supporting it and released Windows 98 as an upgrade.

Windows 98

Windows 98 and 98SE provided very minor upgrades (USB, Internet connection sharing) and fixed some problems at quite a high cost: the recommended RAM size for 98SE was three times greater than for Windows 95 – Wirth's law in action – welcome to bloatware.

For a while Microsoft seemed to be the unlikely saviour of the world as the last bastion against the encroaching Unix hordes. Unfortunately, the Unix rot was too pervasive within Microsoft.

Windows NT 5

Microsoft upgraded NT to NT 5 and tried branding it as Windows 2000, but very few people were taken in. All that NT offered them over Windows 98 was lower performance, an apparently insatiable demand for memory and seriously difficult system maintenance – who would want that?

Forced into either continuing to support Windows 98 or upgrading it, Microsoft decided to make a sort of hybrid "Millennium edition". Disaster again.

1996 - Linux 2, a radical shift

Linux 2, released in 1996 marked a major shift towards the academic world and the mainstream Unix server market with support for shared memory symmetric multiprocessing: first with a Big Kernel Lock and then with finer grained locking. It also marked a major shift in policy. In the early days, Linus Torvalds had vigorously opposed not only locking, but also making separate versions of Linux for different applications. Linux 2 could be compiled with and without symmetric multiprocessing support.

2001 to 2005 – Steps back and forwards

2001 - Microsoft stabs its own customers in the back

Microsoft's customers had made it very clear what they thought about Microsoft's Unix-like NT operating system, but Microsoft did not seem to consider that the NT development policy could possibly have been misguided.

Microsoft's recovery plan for the Millenium edition mess was to rush out a minor upgrade to NT5 (NT 5.1) in 2001, branding it as Windows XP with a massive publicity campaign, this time targeting their core market, small users. The new version of NT had both "home" and "professional" editions. Realising that the customer resistance to Unix levels of performance would have to be overcome, nothing was left to chance. On the one side, Microsoft advertised the "power" (i.e. inefficiency and insatiable hunger for memory: remember the "powerful" Amiga operating system) of XP – the recommended minimum configuration was 8 times more RAM (128k vs 16k) and two generations of processor (Pentium II vs i486) by comparison with Windows 98. XP was, therefore, destined exclusively for a new generation of "Designed for Microsoft Windows XP" labelled PCs: there was no question of running it on a one-year-old machine. On the other side, Microsoft withdrew Millennium edition and announced the imminent withdrawal of support for Windows 98. As a result, XP was widely, if reluctantly, adopted for new machines. After all, what alternative did users have? Linux?

2001 - Linux 2 grows

Over the years, Linus Torvalds seems to have lost his grip on Linux.

Kernel version 0.01 had 10,239 lines of C (against Domesdos's 5,000 lines of assembler), kernel version 2.6.30 had 11,637,137 lines of C. Did it really have a thousand times the functionality of version 0.01?

Not only is the code size of version 2 ballooning for little visible improvement, with the recent 2.6.26-rc1 kernel, the AIM benchmark ran 40% slower than with the previous release[18]. The problem was tracked down to just one semaphore (how on earth did that get there?).

The problem was caused by removing 7000 lines of "unlamented" code from the general semaphore. Semaphores are usually treated academically as having zero cost because the cost cannot be quantified, but to streamline this semaphore, someone had removed about four times as much code as the whole of Domesdos and all its device drivers! Unfortunately the 7000 lines of "unlamented" code proved to have been necessary.

Linux is notorious for its hundreds of changes between its frequent releases, but these are mostly minor. I could well be wrong, but I can find only two changes that mark significant shifts in capability.

In 2001 Linux 2.4 introduced support for USB.

This was three years after Windows and Mac OS, which shows one of the limitations of the Linux development method. For the first time, Linux became a contender for PCs and workstations.

18 http://www.linuxworld.com/news/2008/052008-kernel.html

By 2004, various patches had appear to produce a real time version of Linux.

These were supposed to be included in the Kernel in later versions of Linux 2.6, but at the time of writing in 2009, the "real time" aspect of Linux performance seems to depend more on brute processor force and vain hope than on any real design to meet time constraints.

2002 - Mac OS X

In 2002 Apple withdrew the long running original classic Mac OS series in favour of the Unix based Mac OS X that had been previewed since 1999. There was now no non-Unix system in the mainstream personal computer and workstation market.

To compensate for this major step backwards, Mac OS X introduced a number of small but significant, steps forward.

For the first time on any mainstream system, the windowing was based on off-screen buffers (c.f. Pointer Interface for QDOS, 1986). The display was kept up to date, not by the applications, but by a background task, the Compositor, copying from the off-screen buffers to the display frame buffer. This was based on the same techniques as the various patches to the QJump extended environment (PEX, PNICE, PIE) in use from the early 1990s. As Apple was supposed to have control over its own hardware platforms, it is astounding that this was not handled directly by new display hardware with the separate window buffers being displayed directly. In fact, Apple no longer had control over its hardware platforms for workstations: it was now in the business of packaging and branding standard PC hardware – how the proud are fallen. A very clever solution was found by treating the window buffers as large 3D textures which could be painted by games-oriented graphic processors in the display controllers. "Clever" is one of the worst insults in systems development and maintenance.

The compositor also introduced drop shadows on windows (familiar to old QLers) had a patented "dock" (QPac2 1986) and allowed for translucent windows (novel and pretty, but still looking for a useful application nearly 10 years later).

2005 and a bit - Windows NT 6

Windows Vista (NT 6) was announced in July 2005, but was not released until more than a year later. However, chronologically, it belongs here. Packed full of features to make it faster and easier to use than earlier versions, it was, as a result, enormously larger, slower, less coherent and more difficult to use.

Vista take-up was effectively limited to those who did not know how to avoid it - three years after it was released, it had only reached a penetration of 25-30% of the Windows base. If Vista had been no worse than XP,normal replacement, new sales and piracy should have pushed it to well over 70% after three years.

The most visible feature of Vista was the windowing. Best considered as a copy of Mac OS X Quartz, the new Desktop Window Manager (DWM), based on off-screen buffers, showed "none of the redraw artefacts, latency, or tearing effects that you may encounter in earlier versions". "With the Desktop Window Manager, applications do not draw directly to the video memory; instead, they draw their contents to off-screen buffers in system memory that are then composited together by DWM to render the final screen, a number of times per second".

Microsoft developer Greg Schechter[19] explained the significance; "when a window moves across the screen in XP and before, the portions of background windows that are newly visible only get painted when the background application wakes up and starts painting ... For non-responsive background applications, or even responsive ones that happen to be paged out, this can yield a very poor user experience."

In 2005, this had been known for decades, but the real significance is that although the "user experience" is very poor even if all of the applications' display data structures are "100% correct, 100% of the time", there is enormous cost and difficulty in ensuring that this is true – another major contribution to software unreliability and bloatware – so why was this technique ever used?

Vista also introduced another "new feature" just after it appeared in mainstream Unix, Linux and Mac OS X: Address Space Layout Randomisation (ASLR).

19 http://blogs.msdn.com/greg_schechter/archive/2006/03/05/544314.aspx

Box 11 - The Unix buffer overrun vulnerability

Why Unix in particular?

Because the Unix environment (which includes C compiled executables) brings together three 'design choices', each of which should probably have been avoided for various other reasons, that create a unique vulnerability.

1. Fixed address, virtual machine model of execution.
2. Intrinsically unbounded string data structure.
3. Variable length data storage on the procedure return address stack.

Together, these make it possible for an outside agent (another computer on an internal or external network, for example) to send the target computer data which will overflow a string buffer on the stack and overwrite the procedure return address. The data overwriting the return address defines the point in memory where the execution continues when the current procedure returns. This could be code within the program or a dynamic library or code 'injected' into a string buffer on the stack or elsewhere in memory.

1 Fixed address, virtual machine model of execution

This model was the basis of the multi-user MULTICS system. The intention was to create a virtual machine for each user or 'process', so that, each time a program executed, on any computer, the environment (including all addresses) would be identical. Each process, therefore, had its own fixed address space that duplicated the address space of every other process.

This was not the execution model used by most conventional multitasking operating systems of the time, where all programs shared the same address space: every time a program was executed, it could, potentially, be loaded at a different address in memory. This meant every program had to be either 'position independent' (this is explicit in the MC68000 instruction set) to execute at any address or 'relocatable' to execute at the address where it was loaded. The shared address space execution model has the advantages of being simpler for the operating system, potentially cheaper in hardware terms and providing a much more efficient operating system interface. These are very good reasons for not choosing a fixed address execution model, even if you are not worried about vulnerability.

If the address at which a program executes is fixed, then it is possible to have predictable results if a subroutine return address is overwritten with any one of a very large number of values. On the other hand, if the position at which a program executes is not fixed, then it is 'practically' impossible to overwrite return addresses on the stack and have any sort of predictable result.

2 Intrinsically unbounded string data structure

Although string variables are not intrinsically defined by the C language, string constants are. As a result, the whole of the C environment assumes that string variables use the same structure as string constants. This structure defines neither the length of a string nor the length of the buffer in which a string is stored: the end of the string is marked by a sentinel and the end of the buffer is not marked at all.

This C structure is less efficient than having defined length strings for any of the range of string operations that are commonly used other than copying or concatenating short strings. Furthermore copying a short C string is only significantly more efficient for certain processor architectures and only if no check is made for buffer overrun.

To make it worse, variable length strings are not identified as such in the language, they are merely a convention so there can be no systematic checking. The C choice has been justified on the grounds of efficiency: an intrinsically inefficient string representation was chosen and so bounds checking could not be considered as this inefficient representation makes bounds checking very costly.

3 Variable length data storage on the procedure return address stack.

C is not the only language where local variables are stored on the return address stack, this is a natural consequence using a recursive programming language on a stack based processor. One of the 1960s programming language dogmas was that recursion was 'elegant' and, therefore, should be considered to be the normal paradigm for subroutine calls rather than an exceptional or even aberrant method of handing nested data definitions.

For C and other languages of the period, this was not too much of a problem for fixed size data, but storing variable length data on the stack is, however, a different matter if you have no idea how large the data is going to be in advance.

C adds a twist to this: lengths of C strings are not just unknown in advance, but unknown even when they are being processed. The usual C design philosophy was to allocate a much larger buffer than usually necessary (very wasteful) in the knowledge that buffer overrun would be very unlikely unless you deliberately set out to do it. Who cares about people who deliberately set out out create buffer overruns?

2005 - Address space layout randomisation

The arrival of address space layout randomisation in mid 2005 (Mac OS X.5, June 2005; Linux 2.6.12, June 2005; Windows Vista, announced July 2005, but already longer in development than the others) is notable for the vast quantity of exaggerated claims, pseudo-mathematical nonsense and misleading information that has been generated about a dirty little patch for a problem that should never have existed.

The intrinsic vulnerability of the combination of the Unix virtual machine execution model and the C function call mechanism to buffer overrun attacks (see Box 11) had been known since the 1970s. The first major attack (the Morris Worm on BSD Unix) took place 3 years before the version 0 of Linux was developed, 5 years before Windows NT was released and 11 years before Mac OS X was previewed.

Why is this vulnerability so important, why would anybody produce a new operating system with this known vulnerability and why did it take so long to patch it?

The importance of the vulnerability is twofold. Firstly, buffer overrun exploits can potentially infect Unix type computers at any point where they read data from the outside world, in particular via Internet protocols. New exploits are close to undetectable until after the computer has been infected and the damage has been done. Viruses, on the other hand, because they can only spread by the transfer of infected executable code from one machine to another, can be prevented by simple prophylaxis. Secondly, because the vulnerability exists at the level of normal Internet data transfers, rather than being limited to the installation of infected software, it creates the possibility of a computer being "taken over" and controlled "invisibly" by a remote system or user whether or not internal or external firewalls and anti-virus systems are used.

I do not think that Linux, Windows NT and Mac OS X were deliberately designed to be vulnerable. It seems more likely that their designers were suffering from the same tunnel vision as the designers of Plan 9, who, in the early 1990s, were able to write that the majority of personal computer workstations were running Unix when in the real world the usage of Unix was so low that it did not even figure in the charts. There was a whole generation of computer scientists who, although they may had the occasional brush with "real world" operating systems, had learnt computing using Unix (the original open source operating system), had learnt Unix fundamentals as "universal truths" and simply did not know that the vulnerabilities they had learnt about were specific to Unix.

It took a long time before any patches were produced to deal with the vulnerability because it was not the result of a simple oversight: the problem was fundamental to the design of the Unix / C environment. It was so fundamental that, for the major weakness (the fixed load address for executable code), the patch for Windows was only effective for dynamic libraries and special executable programs marked as relocatable and not for existing programs, the patch for Mac OS X was ineffective for any executable program, while the patch for Linux ...

In the next issue, Tony explains the 2005ish situation and more recent developments.

Zoostorm Freedom XL 10-270

by Dilwyn Jones

For some time now, I've been using an eeePC netbook, running Windows XP, as a kind of portable QL. My QL system these days consists of QPC2 running on any suitable Windows platform, so as a portable QL system, it was fine even if the 7 inch screen and tiny keyboard and slow 900MHz processor were a bit of a drag. That said, QPC2 ran fast enough for me, and if I really wanted to I could plug in a full size keyboard and external monitor.

Recently, with the decreasing costs of netbook systems, I decided to upgrade from the eeePC when I saw an Argos special offer on the Zoostorm Freedom XL 10-270 netbook, at just £199 (part number 508-3053 - I don't know when the special offer ends, though)

At the time of writing, I've been using this ultra-cheap 10 inch screen netbook for about 3 weeks and am really getting to like it! It's some way from being the most advanced Windows netbook or small laptop you can get, but at this price I'm not complaining as it runs QPC2 (and from what little I've run QemuLator on it, that seems to work fine too).

The spec is quite reasonable. It has good *has now been shown to successfully work with all of the following equipment as a complete replacement for the good old floppy disk drive:*

- Any computer/piece of equipment that uses PC formatted floppy disks (3.5", 5.25" or even 8" drives)
- Atari ST/STF/Falcon
- Amstrad CPC6128
- Commodore Amiga (currently write only)
- Dragon 32 / 64 (VDK or JVC disk format, which should also therefore work with the Tandy CoCo)
- Emax and Emax II Sampler
- Ensoniq Mirage Sampler
- Korg DSS-1 Synthesizer
- MSX2
- Oberheim DPX1 Sampler
- Oric Computer (with MicroDisk)
- PC
- PC88
- SAM Coupe *.MGT and *.SAD formats
- Sinclair QL raw disk images
- Sinclair ZX Spectrum +3 or Sinclair ZX Spectrum with PlusD disk Interface
- Super Wildcard DS-SWC3201
- Thomson TO8D
- TI99/4A
- x68000

Boots On?

Although QL Today is posted to all non-German readers from Austria on the same day, some readers have to wait longer than others to receive the magazine. In fact it is delivered in Canada earlier than any other land than Austria itself, 3 days after posting. Next is the UK at 9 days and then next door Switzerland at 13 days. France takes 14 days and Belgium and Norway 15 days.

Dilwyn Jones was merciless:

"Well, there we have it. The ultimate cost-saver for Jochen. He would be able to walk all over Europe to drop off copies knowing that in most cases he'd get there quicker than the postal services could manage. You're WALKING - a few good pair of shoes is all you need."

Clearly Dilwyn (Dylwin? Dillwyn?) has not forgiven QL Today for spelling his name in three different ways three issues ago.

New QL Forum

Peter Scott sent a last-second news item:

www.qlforum.co.uk

It looks great!

News

25 Years - Part 5

by Tony Tebby

2005ish – The beginning of the end for two paradigms

Objects lose out to defined data structures

2005 marks the year that Microsoft's Office Open XML was proposed as a standard document format. This was published in December 2006 as standard ECMA-376. What has that to do with objects? Nothing! That is the point. A fundamental part of the object concept is that the structure of the data is hidden and cannot be accessed directly. A collection of algorithms (properties and methods) is provided to set, retrieve and manipulate data without applications needing to know about the structure of the data.

One of the long standing computer science paradigms is that objects, with their hidden data structures, would displace "raw" data structures as the data carrier for data processing, storage and transfer.

Objects embedded in documents

The major advantage of representing embedded elements of documents (charts, formulae, etc.) as objects is that the data structures for these elements do not need to be pre-defined as they are manipulated only by the methods and properties exposed by the object.

Embedding objects in documents was the primary objective of Microsoft's OLE (Object Linking and Embedding). This was a nasty little patch that, according to the blurb, enabled you to embed, for example, a pie chart in a Word document. The principle was that the embedded object could be manipulated by its associated code which could, effectively, be executed within the application container. The practice did not live up to the principle. Documents with embedded objects proved to be neither very portable nor easily maintainable.

Microsoft's Office Open XML introduced the idea of nested defined data structures. Thus a DOCX text document file (a defined data structure) can include an XLSX file (another defined data structure) with a chart eliminating the need for OLE in most cases. Unfortunately, given Microsoft's attachment to OLE, this does not seem to happen by default.

The timing of the announcements for Office Open XML appear to have been forced by an attempt by OASIS (a consortium of losers in the office workstation market) to hijack the process by getting a similar but incompatible standard (ODF) based on Sun Microsystems's proprietary office document format published by ISO/IEC just seven days before Office Open XML was approved by ECMA International.

Office Open XML, like ODF, has an archaic expression of the data (XML) and a document concept based directly on the nearly 30 year old separation of word processor, spreadsheet and slide show rather than a more easily processed binary format with separation of the content (in a common format) from the presentation (in a coherent set of formats). Office Open XML does, however, mark a major step back towards sanity: in most cases OLE can be replaced by embedded data structures.

Objects for distributing information and processing

The advantage of using objects for distributed processing is that, since the data inside the object cannot be accessed directly by an applications program, it does not make any real difference whether the data is on the same machine or on the other side of the world: the operations can all be done passing messages. There is one little difference: using an object within a program is stunningly inefficient, whereas using an object on another machine is mind-bogglingly inefficient.

Distributed Objects Everywhere (DOE) was Sun Microsystems's attempt to allow data to to processed remotely by using object methods called from one machine to process object data on another. It was based on the Common Object Requesting Broker Architecture (CORBA) which allowed clients to retrieve objects (data and the code to process it) over networks. Five years in development, it was released as NEO in 1995 and withdrawn in 1996. It is not at all clear whether CORBA is dead as well.

Sun replaced DOE by Java applets for clients and servlets and Javabeans for servers. More than 10 years later, PHP (not really object) seems to have displaced object oriented server side Java for all but a hard core of true believers. As far as transferring data to net clients, Java seems to have been reduced to an animation niche. The bulk of data distribution over the net is not in objects but in defined data structures: starting with html, through standard image formats, jpg, gif and png, to swf, pdf, avi, etc.

Objects in programming

The intention of object oriented programming was to reduce development costs and improve quality by simplifying the process of creating software. improve the maintainability and re-usability of software, going one stage beyond conventional modularity. From a naïve point of view, object oriented programming has two advantages: applications can manipulate different types of object using common methods, without knowing how the methods work on each particular object, and the internal data structures can be modified or extended without any impact at all on application code.

As a consequence, since the operation of the methods depend on both the data structures (which are not "exposed" and liable to change between versions) and classes of object (which are not necessarily known in advance) methods and their side effects can only be defined in the very vaguest of terms. Some defenders of object oriented programming claim that methods and their side effects can be well defined but, in general, this can only be true if the methods apply only to a known class and if the data structures are fixed. However, if the class is known in advance and the internal data structures are fixed, the only differences between using well-defined methods and well defined, old-fashioned procedures with well defined data structures and functions are the syntax, the efficiency and the flexibility.

Object oriented programming is extremely inefficient, reducing the performance of software while increasing its size, greatly contributing to the bloatware phenomenon. According to computer science dogma this is unimportant as users can always get a more powerful computer: the inefficiency is "justified" as the approach reduces development costs and improves reliability.

Does it? Over the past 25 years object oriented programming has taken over from other software development methods while software development costs have ballooned and quality has declined.

There is a small minority view (see Box 12) that puts the peculiar flexibility of object oriented programming as one of the causes of this decline, while the majority view seems to be that things would have been even worse if object oriented programming had not been adopted. This majority view that object oriented programming has saved the world has the advantage that, since it can neither be proven nor disproven, it can be taken on faith: anyone who believes otherwise is clearly demented.

Box 12 - The flexible animal object

Programming using defined data structures and object oriented programming provide radically different notions of flexibility. With defined data structures, an application programmer can easily extend the range of functions and procedures beyond the "standard" library functions for that data structure, but he also has the flexibility to destroy the integrity of the data. With object oriented programming, an applications programmer is strictly limited to the standard methods as only programmers with inside knowledge can extend the functionality. Object oriented programming provides, however, the flexibility to use a given method to manipulate radically different objects without needing to know how the method works. Is this a good idea?

Searching the net for any sign of a rational justification for object oriented programming I came across the "animal" object which was used in a computer science course as an example of how the flexibility of object oriented programming simplifies software development, reducing costs and improving quality.

The animal object has a method goFaster. For some animals goFaster makes them run, for others it makes them fly. There is no "fly" method because not all animals can fly, and there is no "run" method because not all animals can run. goFaster is, therefore, an abstraction for either. A programmer does not need to know how an animal responds to goFaster, it just works for all animals. Magic!

Imagine, for a moment, that the animal object is used in a simulation with birds. When a bird pecking at seed on the ground reaches a stream, we can get it across the stream with the goFaster method. This will work fine until the system is used with flightless birds, when, except in a few remarkable cases, the flightless bird will drown.

You might say that the problem was caused by the programmer who did not know that there are flightless birds. That is missing the point entirely: in object oriented programming, programmers are not supposed to know what goes on inside an object, so they can have no idea of the limitations on the applicability of any particular method.

You might say that the problem was caused by the programmer using the wrong method. That is missing the point entirely: the number of methods "exposed" by an object is limited and programmers have to use the nearest match unless write their own methods, which would negate the whole purpose of using object oriented programming as they would have to know about the object's internals, which they are not supposed to.

Why would anyone think that forcing ignorance on software developers is a good idea? Beats me.

Multiprocessing hits a brick wall

Although symmetric multiprocessing was fundamental to the 1960s systems theories that have dominated systems design from the time MULTICS was designed and although symmetric multiprocessing versions of Unix and Windows had been around for more than a decade, by 2005 symmetric multiprocessing was still a very restricted sector. In May 2005 AMD introduced its first multicore workstation processors, the Opteron and the Athlon 64 X2, and Intel introduced the Pentium D. Over the next year, experience of multiprocessing increased rapidly.

From the 1960s the IT establishment had viewed symmetric multiprocessing as the natural choice for performance The driving force behind Windows NT was to produce a shared memory symmetric multiprocessing competitor to UNIX. In the guide to Windows 2000 (NT5), Microsoft stated its position, which corresponded roughly with the establishment view.

> If you wait long enough, perhaps your performance problems will just go away with the next generation of computer chips! Another proven technique is multiprocessing, building computers with two, four, or more microprocessors, all capable of executing the same workload in parallel. Instead of waiting another 18 months for processor speed to double again, you might be able to take advantage of multiprocessing technology to double or quadruple your performance today.

The widespread adoption of multi-core processors in desktop PCs has changed the perception from simple scalability towards doubtful benefit. Very few would now be bold enough to describe shared memory symmetric multiprocessing as a "proven technique ... capable of executing the same workload in parallel ... to double or quadruple your performance (two or four microprocessors)". This is a goal (the unachievable "Holy Grail" of computer science) that has kept tens (or is it hundreds?) of thousands of computer scientists off the streets for nearly half a century.

Multiprocessing problems

Microsoft's "Multiprocessor Considerations for Kernel-Mode Drivers"[20] (October 2004) states by way of introduction "future technologies mean that all new machines will eventually support more than one processor". The document concerns shared memory symmetric multiprocessing as implemented in Windows NT. There is no attempt to justify shared memory symmetric multiprocessing rather than asymmetric architectures and there is no suggestion that single processor architectures could continue to provide a cost effective solution for a large part or even the majority of systems. In the 1960s, multiprocessing was seen as a means to an end, for the last 25 years it has been an end in itself.

In shared memory multiprocessing systems, the problems of memory contention are both far more performance critical and far more delicate than for other architectures. Whereas Microsoft was bullish about their proven multiprocessing technology in the blurb for Windows NT5, 4 years later in in this document they became less categoric about the reliability of their system.

> However, in a few situations, you must prevent or control reordering. The volatile keyword in C and the Windows synchronization mechanisms can also enforce program order of execution in nearly all situations.

So, according to Microsoft, the memory contention mechanisms used by Windows device drivers for multiprocessor systems will work in "nearly all situations". What happens in other situations? This is hardly "proven technology" of the year 2000.

Multi core performance benchmarking

The performance benefits of multiprocessing depend very strongly on the type of applications being run. There is a class of "embarrassingly parallel" problems that comprise a very large number of similar, independent calculations (such as calculating fractals, image generation, finite element analysis, playing chess, etc) or similar, independent operations (such as indexing, data mining, spiders, etc). Although they are all easily implemented as parallel processes, they are not necessarily suitable for shared memory multiprocessing: if the algorithms are more data intensive than calculation intensive, then memory bandwidth or disk bandwidth will be the limiting factor and using computer arrays or farms will be a far better approach than using multi-core or multiprocessor systems. Furthermore, these problems are rarely true workstation applications in that they yield results over timescales from minutes to days.

In the enthusiastic rush by the press to print articles extolling the virtues of multi-core processors in personal workstations, there was little objectivity. A typical example is the comprehensive AnandTech report[21] summarising the performance on a range of standard benchmarks (See Box 13). Not only did the results show that dual core processors nearly always had a worse price / performance ratio than similar single core processors, they also showed that, on a number of important tests, the performance of dual core processors could be improved simply by disabling one of the cores.

Three types of usage gave distinct results.

For ordinary "one thing at a time" usage, where a single operation dominated even though system maintenance tasks would be working away in the background, the second core was either useless or degraded the performance. This type of usage had two groups of applications: some potentially embarrassingly parallel applications which would become parallel by 2009, and "office" applications where a single core system is likely to be best choice for the foreseeable future as the most important performance criterion is not the "throughput" measured in these tests but the response time – the time to carry out a single action in response to a single event (keystroke, mouse click, etc.) – and the predictability.

20 http://download.microsoft.com/downloa d/e/b/a/eba1050f-a31d-436b-9281-92cdfeae4b45/MP_issues.doc
21 http://www.anandtech.com/printarticle.aspx?i=2410

Box 13 - Multi-core processor benchmarks

A good example of independent benchmarking in 2005 of the new dual core processors was published by AnandTech. The set of benchmark results included two AMD processors with identical technology and clock speed (Athlon 64 4000+, single core and Athlon 64 x2 4800+, dual core) but twice as much high speed cache for the dual core processor. The benchmarks were carried out for three different types of workload.

1 One thing at a time workstation usage

- For 8 straight office applications (Business Winstone, PC Worldbench and WinRAR) the dual core was 0.88 to 1.00 times as fast as the single core processor, i.e the dual core processor would have performed better if one of the cores had been disabled.
- For the single-threaded computer generated image test (with two different sets of data), the dual core was 1.01 and 1.03 times as fast as the single core processor.
- For 6 graphics intensive games, the dual core was 1.01 to 1.04 times as fast as the single core processor.

For all of these, an equivalent cost single core processor would clearly be a better choice for performance.

2 Embarrassing parallel multi-threaded applications

- For 8 straight media encoding tests, there were three groups of results. Three of the tests (image and sound processing) gave no significant difference between dual and single core despite the fact that two of the three programs were explicitly multi-threaded (more recent tests have shown some "improvement" in these applications). For three of the tests (video encoding) the dual core was 1.16 to 1.38 times as fast as the single core processor. For the other two tests (DivX and WMV9 HD video encoding) the dual core was 1.73 and 1.91 times as fast as the single core processor.
- For the single 3D image generation test (repeated five times with different data) the dual core was an average of 1.82 times faster than the single core processor.

The median speed advantage of the dual core processor was 1.24. However, the test conditions were biased against the single core processor by using the same number of threads for the single core processor as was used for the dual core processor introducing unnecessary overheads on the single processor system.

3 Multitasking user

The third group of tests simulated users carrying out background tasks while working with a principle application in the foreground. Aggregate speed is a very poor measure of perceived performance.

- For 3 office applications where the user was performing several tasks simultaneously (Sysmark 2004) the aggregate speed for the dual core was 0.95 to 1.27 times the single core processor speed.
- For 5 media creation activities with the user generating or encoding images (1) or videos (4) while carrying out other operations the aggregate speed for the dual core was 1.40 to 1.55 times the single core processor speed.

There were a number of tests showing the difference in background task speeds, but only in one case were both speeds given. This showed that the Windows XP scheduler was fairly effective in that the execution speed of the foreground task was very similar for single and dual core, but that, with a single core processor, Windows NT sacrificed the speed of the background task(s) to maintain the foreground task speed.

The results were very different for the embarrassingly parallel applications. Multi-threaded image generation and video processing should give the most favourable results for multi-core processors. But, even if as many as 5% of workstations are used for these applications an average of 10% of the time, they would currently represent less than 1% of workstation usage. Despite this they figured in 16 out of the 36 straight benchmark results, which indicates the level of bias in this and other reports. Even so, an equivalent cost single core processor with correctly configured applications would have given as good or better performance on 4 or 5 out of the 9 tests dedicated to embarrassingly parallel applications. The other tests gave genuinely better results for the dual core processor but they fell very far short of the "expected" doubling of performance. The principal reason is that the standard "desktop" workstation configuration is very ill-suited to these types of applications: using two separate computers, each with a single core processor and half the main memory could have provided a much higher performance, at little extra cost, than the dual core processor tested.

The third type of usage was the "multitasking user". These tests rarely measure anything meaningful. Where a user is working with an application while, for example, fetching e-mails in the background, the speed of the application in the foreground is important, but the speed of the background task is not. Measuring aggregate speed (or even worse, just the speed of the background tasks as in most of these benchmarks) gives results that are very favourable to multiprocessor architectures, but rarely applicable to the real world. The only test that measured the foreground task speed showed no advantage for the dual core processor.

All these benchmarks taken together indicate that, although there are certain cases where the dual core processor performed better, the dual core processor would certainly underperform a single core processor with similar technology, similar total cache and similar cost (and, therefore, higher clock speed and more "cache per processor") under typical workstation conditions. "Power users" might find that when they are running a number of tasks simultaneously the dual core might give a higher performance, and this higher performance might offset the lower performance of the dual core processor on a more mundane workload – but it is not certain and much will depend on the effectiveness of the operating system's scheduling algorithm and the prioritisation of the foreground and background tasks.

Multicore processors 4 years on

In 2009 AnandTech published another benchmark report[22] featuring multi-core processors. As the report set out to compare quad core processors, ordinary workstation use was excluded. Even so, the performance improvement in embarrassingly parallel applications was only about a factor 2 in the four years since the previous benchmark cited above, very much below the previous rate of a factor of 2 every 18 months for applications in general. Furthermore, even under the very favourable benchmark conditions, the report pointed out that the migration from dual core processors to quad core processors only increased the performance by about 30%: the largest contribution to performance improvement was the new cache architecture in both Intel and AMD processors.

Hardware for parallel processing in workstations

One of the notable features of benchmark reports from 2005 onwards was the absence of critical comments pointing out that, even for embarrassingly parallel problems where the work can easily be divided into a number of independent tasks, the new generation clearly failed to approach the "proven" 2 or 4 times increase in processor power using 2 or 4 processors.

Fundamentally, multi-core processing is a loser technology for workstations (see Box 14), as was the earlier RISC architecture. One of the common features of the applications where the benchmarks gave multi-core processors a significant advantage was that they carried out intensive processing on relatively small datasets.

Intensive processing on relatively small datasets is ideal for "computer farms" where an "intelligent" controller "farms" out the work to not very intelligent, but very fast, calculating units, ideally with "calculator" instructions (fixed point arithmetic, table interpolation, etc.) and graphical data handling (pixel masking, anti-aliasing, etc. as in GPUs). A calculating unit could be packaged as a modest quantity of fast RAM tightly coupled to a processor that occupied much less chip space than one core of an equivalent technology x86 processor while delivering several times the calculating power. Moreover, for the type of applications concerned, the processing speed would be almost proportional to the number of calculating units.

The Transputer, the first single chip computer specifically designed for farms and similar architectures, was released 25 years ago in 1984. It was an unconditional failure. Intensive calculation is the only application for which parallel processing is of clear interest, but the technological limitations of the time, coupled with the firm belief in the dogma that symmetric multiprocessing was the only true way for all computing, meant that the Transputer was not well targeted for intensive calculation and too expensive for anything else. Asymmetric (one controller for many calculators) computer farms have since become fairly commonplace for a variety of seriously intensive calculations. Why not in a workstation?

The simple answer is that intensive calculation is a tiny minority interest (I am in that tiny minority) and apparently cannot justify the development costs. But if it is only a tiny minority interest, why does this type of computing dominate current workstation benchmarks and why were multi-core processors

22 http://www.anandtech.com/printarticle.aspx?i=3492

Box 14 - Multi-core, a loser technology

While it should be fairly obvious that on dominantly single task workloads, multi-core processors will not provide a better performance than an equivalent cost single core processor, why can disabling cores on a multi-core processor improve the performance and why is the speed increase for ideally parallel processing very much less than the number of cores? The answer to both questions lies in the main memory bandwidth and caching.

1 Caching

At the limit, a computer's performance will be limited by the main memory access time: the speed at which the processor can move data in and out of memory. This is masked, to a certain extent, by using processor caches to hold data to improve the speed of repeated accesses to the same data items and hide the write back time from the processor.

The largest contribution to benchmark performance increases since 2005 has been improved caching with the introduction of three level caching. The 2009 AnandTech article cited gave claimed, measured and estimated access times for the three levels of caches in Intel and AMD processors.

The fast (multi-ported, prefetched) L1 caches nearest the execution unit had access times (latencies) of 3-4 cycles, the L2 caches had access times of 11-15 cycles and the large shared L3 caches had access times of 40-50 cycles. External RAM accesses took 200-250 cycles. The timings are fairly balanced with about a four times increase in access time at each level. An increase in the overall cache miss rate of 1% could increase the average data access times by about 50%.

With a multi-core processor the largest cache is usually shared between the cores. When all cores are executing, the largest cache will see continuous accesses by all the cores, with each of the tasks executing concurrently "stealing" cache continuously from the other tasks.

For a dominantly single task workload on a single core processor, this cache stealing still occurs but is limited in effect as the processor only switches tasks at well spaced intervals. With a multi-core processor, the continuous cache stealing by the background tasks can significantly increase the number of cache misses by the dominant task, reducing its performance by more than the small advantage gained by executing the background tasks on other cores. This can be seen in the 2005 benchmarks.

For an ideally parallel workload on a multi-core processor, the cache stealing can be far more serious as it not only directly increases the cache miss rate, it also increases the risk that the processor falls near or into "cache thrashing" where the miss rate increases dramatically and the performance drops. While cache thrashing can occur with just one core, the more cores that are accessing a shared cache, the more likely it becomes. Under strain, therefore, the performance of a multi core processor may degrade more quickly than an equivalent single core processor. Because this is an "occasional" phenomenon, it will not show up very clearly in benchmarks that test only average speeds, but it should rule out the use of multi-core processors in response critical systems.

2 Main memory bandwidth

25 years ago, memory bandwidth was the brick wall limiting processor performance. The introduction of caches has cushioned this performance barrier but not removed it. There will always be instructions that cannot be executed entirely from cache.

But, if there are two cores, one core can be executing from cache while the other core is waiting for the external memory. Can this provide a doubling of performance under an ideal parallel workload? The simple answer is no. Not only is it very unlikely that the cache misses (which are randomish) will interleave nicely, cache stealing will increase the cache miss rate and so the performance of each core will be severely degraded. For each additional core, there will be less memory bandwidth available for any of the cores, reducing their performance and there will be more cache stealing, further reducing their performance. With each additional core, the performance gain is less and the performance loss greater.

Even for ideal parallel workloads, unless you can guarantee that nearly all the data required for the execution of all tasks on all cores can be held in private caches or in a multi-ported cache shared between the cores, there is a limit to the number of cores that can be used before the overall performance actually drops. For ordinary workstation benchmark tests in 2005 (Box 13), this limit was one.

The dream of massively multi-core processors (64, 128 etc.) is just a nightmare.

developed for workstations when their only advantage over single core processors is for this tiny minority interest?

The emergence of multi-core processors in workstations has nothing to do with performance, it is just the pursuit of a 40 year old dogma.

More than 20 years ago, the designers of Plan 9 (and Unix) based their whole concept on the eagerly anticipated "coming wave of shared-memory multiprocessors".

More than 20 years ago, Windows NT was designed to support shared memory symmetric multiprocessing which was, at the time, a 20 year old, past-sell-by-date concept based on a 1960s misreading of the future of computer hardware in the 1970s. As a result, it was astoundingly slow, complex and oversize: the size did not matter because RAM prices were dropping and the speed did not matter because you could always use "***two or four microprocessors .. to double or quadruple your performance***"!

Software for multiprocessing

One remarkable feature of the arrival of multi-core processors for ordinary workstations that seems to have escaped comment is that existing software actually worked on these new platforms. The reason is very simple: designing all software specifically for symmetric multiprocessing has been a central computer science dogma for very long time. This could be viewed in two ways.

The conventional view is that this vindicates the 1960s dogmas of symmetry and transparency for parallel applications and the amazing foresight of the academics that developed the theories underlying these dogmas and the even more amazing foresight of the industry in developing suitable software well in advance of the arrival of mass-market, shared memory multiprocessing systems.

The minority view is that this is result of an astounding collective madness that, for 40 years, has compromised the performance and quality of software that should have been written for the single processor systems that were actually in use rather than for a hypothetical computer architecture which was to become close to a reality some time in the distant future.

The contention between processors in a symmetric multiprocessing system creates problems that need to be handled in software. The conventional methods, principally synchronisation, used to deal with the problems of shared memory symmetric multiprocessing reduce systems' performance and increase their complexity. Dealing with the increased complexity further reduces the performance while reducing the quality and increasing both the size and development costs. Is this really a sane approach for single processor or asymmetric systems where symmetric multiprocessing problems cannot occur?

On the other hand, can a system such as Domesdos or Stella, designed specifically for a single processor, work on a multi-core or multiprocessor shared memory computer? Yes it can. But can it work more efficiently than a system specifically designed for shared memory symmetric multiprocessing? Yes it can. It is a question of scalability.

An ideally scalable system would have constant overheads per processor (core), per active task and per task. Domesdos and Stella were not ideally scalable. In particular, the basic operating system overheads had a tiny scalable component and a potentially larger (N-1) component where N is the number of symmetric processors or cores sharing the same main memory. This second component is zero for a single processor and jumps as soon as there is more than one processor, leading to the accusation that these types of systems cannot be used for shared memory symmetric multiprocessors.

The reality is different. The (N-1) component for Stella is so much lower than the constant overhead of conventional shared memory symmetric multiprocessing systems (two to three orders of magnitude) that, for a modest number of processors or cores (less than 100?), Stella would maintain its advantage. Furthermore, it is finally being accepted that conventional shared memory symmetric multiprocessing systems based on synchronisation are themselves far from ideally scalable as the cost of waits and context switches that are forced by the synchronisation mechanisms increases rapidly with the number of tasks that are executing concurrently.

The question is not whether Stella would outperform BSD, Linux or NT on a symmetric 16 core processor system but whether prolonging the life of an archaic systems architecture that is totally irrelevant to current and foreseeable future requirements would be morally justifiable.

In the next issue, Tony looks at 2009 and will be "Gazing into the future"

25 Years - Part 6

by Tony Tebby

Gazing into the future

The past 25 years have brought a more or less continuous decline in software performance (efficiency, reliability, predictability) coupled with ballooning software sizes and development costs. What does the future have in store?

The view from 1984

Perspectives on fifth generation computing[23], by Brian R Gaines, is a retrospective view of the major developments in computing. It was published the year that the QL was launched. The paper includes Withington's 1974 futurology, slightly modified to fit the circumstances.

Withington's analysis describes the first three generations of computers and predicts the fourth and fifth.

1. 1953–1958 Gee whiz – computers can do anything, technological curiosities: thermionic valves and mercury delay lines.
2. 1958–1966 Paper pushers, business machines: transistors and magnetic core memory.
3. 1966–1974 Communicators: LSI and teletypes, operating systems and communications protocols.
4. 1974–1982 "Information custodians": central databases and satellite timesharing systems
5. 1982–xxxx "Action aids": distributed intelligent systems

References to the fourth generation had appeared from the mid 1960s onwards: Withington's bold stoke was to predict that it was about to appear in 1974, the same year as he published. It did not.

The fourth generation did appear later, in 1990, in the form of Plan 9 (see Vol.14, I.4, Page 25) and, possibly, other systems, but, by then, it had already been overtaken by the un-planned, un-generational, PC.

Seven years after Withington's article, in 1981, with the fourth generation still nowhere in sight, a Japanese group announced their intention of skipping the disappearing fourth generation and going straight for Withington's fifth generation: this was the Fifth Generation Project which shook up the computing world.

Gaines' bold stroke in 1984, was to announce that the fifth generation was already in place, which it was not, except as vague, impractical concepts in research laboratories isolated from reality. His vision of the future was more of the same.

Gaines' concepts are firmly anchored in a 1960s view of computing architectures: time-sharing clusters or clouds, usually with some form of centralised administration, serving terminals. In his 1984 paper, very little weight is given to personal computers and workstations and none at all to embedded systems both of which were emergent. His "distributed intelligent systems" did not consider the possibility of distributing the intelligence to independent units that might have only limited communications, as has been the dominant form of computing since the 1980s, but clung to the 1960s idea of interdependent units working closely together.

Time sharing sharing systems serving terminals had some advantages over totally independent systems. Within the "original" computing environment, university campuses, any student or any member of staff could access "their" data by logging in to any terminal – they did not all need their own workstation. The terminals had negligible intelligence so, quite naturally they had to share the central processing power as well. Since the early 1980s, however, the main requirement for sharing IT resources has become sharing information, not processing power.

Is applying the campus time-sharing architecture to all computing systems just another case of "generalising from one example" (see X Window System, Vol.14, I.3, Page 36) where an architecture designed for one application (university campuses) was applied to other application regardless of the individual requirements?

23 http://pages.cpsc.ucalgary.ca/~gaines/reports/MFIT/OSIT84/OSIT84.pdf

Not at all. By 1984 it should have been obvious that, even for a campus environment, having terminals time-sharing a cluster or cloud of processing units was not a sensible approach where the distributed processing power in the terminals would far outweighed the processing power of the central cluster. The original campus time-sharing architecture was obsolescent even in its original environment.

In 1984, the new requirement was independent processing and, to a lesser extent, the central management of shared and private data, but the entire edifice of the 1960s theories was based on the now irrelevant concept of "competing processes" on shared computers. From 1984, large system architectures should have moved towards independent processing, on independent machines, accessing shared data. Furthermore, the new requirement for shared data was for "transactional" (asynchronous) access to long term data storage, giving rise to a totally different set of problems to those addressed by the seriously flawed 1960s "synchronisation" theories.

The critical difference between the new approach required and the old approach is that the 1960s theories were algorithm based and conflated data with processing. This conflation reached its apogee in object oriented programming where data and processing of that data are made indivisible. For an independent processing / shared data approach, the data, and the management of that data, must be rigorously separated from the processing of that data. This already tends to happen by default, as it is often the only practical approach. It should be done by design.

The view from Berkeley (California) 2008

Parallel Computing: A View From Berkeley[24] sets out the Berkeley view of the future of mainstream computing. The title is slightly ambiguous. Does the title imply that the paper is only concerned with the future of parallel computing or does it imply that parallel computing is the future?

In a world where the arrival of the massively parallel fifth generation of computing had been announced 24 years before but was still not even vaguely on the horizon, this paper takes the position that the only way forward is parallel processing.

The importance of this paper is not in the content, but the in extent to which, like Withington's and Gaines' papers, it has been adopted, quoted and recycled within the computer science community.

The paper sets out its stall in two ways, one approach considers "conventional wisdoms" and how these need to change and the other considers the state of the art and the future of computer hardware.

Berkeley's conventional wisdoms

The paper sets out a number of "old conventional wisdoms" on system design with corresponding "new conventional wisdoms". These terms are very strange: for "old conventional wisdoms" a better term would be "parody of reality" and the "new conventional wisdoms" would be better described as "old dogma".

The first few "conventional wisdoms" merely repeat the 1960s dogma that processors cannot be improved significantly. Then there is a group of "conventional wisdoms" (CWs) that represent a total denial of reality.

9. Old CW: Don't bother parallelizing your application, as you can wait a little while for a faster sequential computer
9. New CW: A faster sequential computer won't arrive for a long time
11. Old CW: Increasing clock frequency is the primary method for improving processor performance
11. New CW: Increasing parallelism is the primary method for improving processor performance
12. Old CW: Less than linear scaling for a multiprocessing application is a failure
12. New CW: Any speedup via parallelism is a success

These are surreal.

24 http://www.cs.nyu.edu/srg/talks/BerkeleyView.pdf

The old Berkeley conventional wisdom, for the past 40 years, was 'don't bother about making your application efficient, you can always parallel the processors for more speed', whereas, in reality, rapidly increasing processor speed has been the practical means of compensating for rapidly declining software performance. In these conventional wisdoms, they are claiming that parallel processing, the fundamental basis of the fourth generation which was due to arrive about 1974, is THE NEW IDEA for 2008.

Berkeley's start of the art

It would be too easy to dismiss the View From Berkeley as merely an attempt to put a 21st century gloss on a 1960s tract in favour of parallel processing, but there is more to it than that.

The paper starts off with a graph showing that, up to 2002, processor ***benchmark*** performance doubled every 18 months but that the rate is now much lower. This corresponds roughly to the introduction of multi core processors, first for RISC processors, then for CISC processors. This shows that the generalisation of these multi core processors did not maintain the upward trend in performance.

This is followed by a glimpse of reality. "Parallel processing is nothing new. In the past, research in parallel processing was driven by new breakthroughs which opened design space, with uniprocessors always prevailing in the end." This seems to accept that forty years of systems development concentrated almost entirely on symmetric multiprocessing had totally failed to displace single processor systems as the dominant form of computing.

So Berkeley's state of the art in 2008 was that processor performance was running into a brick wall and parallel processing had hit a brick wall years before.

Berkeley's hardware future

The whole of Berkeley's hardware future is based on their 10th "conventional wisdom". Their conclusion from this is that the future is in massively multi core processors with "thousands of cores on chip".

10. Old CW: Uniprocessor performance doubles every 18 months

10. New CW: Power Wall + Memory Wall + ILP Wall = Brick Wall. Uniprocessor performance now only doubles roughly every five years

Once again, the "Old Conventional Wisdom" is not – it is just an observation that was true for strictly defined and not very useful conditions between about 1984 and 2002. The "New Conventional Wisdom" is just an idiocy plus a deliberately misleading statement.

It is true that the graph of processor benchmark performance shows that the performance used to double every 18 months but now the improvement rate is much slower. The misleading statement, however, dissimulates two important factors. The first is that the gap between real performance and benchmark performance has widened over this period and the second is that the tail end of the graph represents the introduction of multi core processors, not the stalling of single processor performance.

The idiocy shows how even the most stupid propositions gain an air of veracity if they are repeated often enough. The well known, frequently cited, and totally wrong "Power Wall + Memory Wall + ILP Wall = Brick Wall" is raised as an argument in favour of multi core processors.

The "Power Wall" is a ridiculous simplification of the problems of reducing the cycle time of the processor core. Power is a factor, but in the 1970s, IBM was already delivering water-cooled processors, so this is not a barrier, it is just a one of a whole series of physical limitations on the processor cycle time. These limitations have, however, not stopped processor core execution speeds increasing far faster than overall processor execution speeds as processors have become more and more limited by the "Memory Wall".

The "ILP Wall" (instruction level parallelism) simply does not exist. The gains that can be obtained using ILP are extremely limited but these gains are proportional to the processor clock speed. If a given ILP strategy can double the effective speed of a 1 GHz core to 2 GHz (two instructions per cycle), then the same ILP strategy will double the effective speed of a 2 GHz core to 4 GHz. At the processor core level, ILP is purely scalable but not very useful: at best ILP is just a quick patch to get around the problems of primitive instruction sets with excess serialisation. The reason that ILP does not yield its theoretical scalability in real processors is that an increase in instruction execution speed does not translate directly into an increase in overall processor speed because, since the early 1980s, processors have been running into the 'Memory Wall'.

The 'Memory Wall' is a real problem. Elsewhere in the paper the author states 'The gap between memory access speed and processor speed increases by 50% per year'. Over this period, main memory access times have increased by about a factor of 5 and memory bandwidth has increased even more with wider buses and larger burst accesses, but this is still much less than the 10,000 times increase in processor core speeds.

The most important thing about the 'Memory Wall' is that it imposes the same ultimate performance limit ***regardless of the number or speed of the processor cores.*** It does not add to the core performance limits, as would be suggested by the ridiculous 'Brick Wall' formula, it overrides them.

The 'Memory Wall' is the multi core killer.

The paper has a big title 'Multicore Not the Solution' that acknowledges the failure of multi core technology (which had ran into the memory wall even before it even became widely available) but proposes adopting it in an extreme form: '1000s of cores per chip', totally ignoring the memory access problems. It is difficult to see any logic in this.

Berkeley's software future

This paper does mark two distinct 'advances' on the 1960s approach to parallel processing.

By the end of the 1960s, the computer science establishment believed that it had the whole solution to parallel processing problems, largely based on the twin concepts of synchronisation and symmetric multiprocessing. Later, Berkeley was one of the prime movers in putting synchronisation back into Unix for symmetric multiprocessing. This paper is, maybe, a mea culpa, as it advocates finding alternatives to synchronisation.

> Locking (synchronisation) is notoriously difficult to program for many reasons (deadlock, noncomposability). Locking is also wasteful in that it busy-waits or uses interrupts/context switches.

So, in plainspeak, synchronisation increases software development cost, while making systems less reliable and less efficient. It has just taken Berkeley 40 years to realise this fundamental truth.

> Switching from sequential programming to moderately parallel programming brings with it great difficulty without much improvement in power-performance. Thus this paper's position is that we desperately need a new solution for parallel hardware and software.

Finally accepting, after 40 years of development of the 1960s multiprocessing theories for parallel execution, that a new solution is desperately needed might be considered some form of advance, even if the proposal is to wind the clock back to 1962 and start again in the same direction. Since it would appear that there is no intention of challenging the false premisses on which the whole multiprocessing edifice was built, it would seem likely that the same mistakes will be repeated all over again.

The view from cloud-cuckoo land

Since the early 1960s one of the unshakeable dogmas underlying systems architecture development has been that single processor systems were doomed as they could never meet increasing performance expectations and the only way forward was to distribute the processing across thousands of processors using a well organised symmetric multiprocessing model. In practice, processing has been been distributed but, instead of all processing being carried out on organised systems with thousands of processors working in parallel, as envisaged in the 1960s, the bulk of processing is now carried out by a totally anarchic mass of independent PCs, embedded computers and hand-helds.

From a computer science point of view, this is wrong. From a user perspective, the current anarchy is certainly better than the fourth generation / fifth generation horror that it has successfully kept at bay. The challenge is not to bring the anarchy under control, it is to make it work better by working with it rather than against it.

Improving software quality (efficiency, predictability, reliability) on the individual computers would be a good start.

Wirth's law on bloatware is not theoretically justified, it is merely an observation which seems to be as true today as when it was formulated. There is no justification at all for the ballooning costs, size, and inefficiency that has thrown away the performance that has been gained by the extraordinary development of computer hardware over the past 25 years.

The computer science world has taken perverse pride in promoting software inefficiency as an implicit aim: anyone who challenges this is treated with utter disdain. For 25 years "you can always get a more powerful computer" has been the "conventional wisdom": efficiency is just a dirty word used by grubby little hackers.

Recovering the processing power gains that have been achieved since 1984 is, however, not necessarily the most serious long term problem to be addressed. On the one hand, workstation and server hardware architectures have been compromised by trying to fit in with 1960s symmetric multiprocessing, on the other hand, system architectures have been compromised by a mindset on realising the fourth generation or even the fifth generation.

In 25 years, this 1960s mindset appears to have become only more rigid so that, to achieve any significant deployment in the short term, any new development will have to be within the existing archaic hardware and infrastructure architectures.

The cheapest route to long term gains in system performance is not increasing clock speeds or paralleling processors, it is improving the software quality. This cannot be done retrospectively: the quality must be built in by designing for reliability, predictability and efficiency. Improved software quality brings with it reduced hardware demands, lower power consumption, reduced size, longer battery life as well as more predictable performance: are these really as unimportant as conventional computer science would have us believe?

The argument against a revolution in systems software is that compatibility must be maintained at all cost. This is nonsense, although there have been outstanding failures to revolutionise a market (the Apple Mac into the PC market in 1984 and Linux in the PC market ten years later), the problem was that the "new" systems were promoted as being different (which they were, superficially) rather than being significantly better for the target applications (which they were not). On the other hand, the PCs of 25 years ago were completely incompatible with mainframe computers, but they wiped them out. Palm Pilots were incompatible with everything else but they were a great success. The pocket computers that wiped out the Palm Pilot were incompatible with anything that went before.

It is possible to make systems that are significantly better, but this cannot be done without making them significantly different, built in a significantly different way. We know it can be done. We can be sure that most of those setting out to do it will just recycle all the old junk theories and end up with just another Unix. Who will be bold enough to break the cycle?

In the 1970s and early 1980s, the development time for an operating system used to be about 1 year. The development of an application program (a word processor, for example) used to be about 1 year. The development of a programming method used to be about 1 year.

It is true that we ask more of applications now than 25 years ago. But it is also true that we should be starting from a more advanced position with more advanced tools so that the core development should be much faster. The additional functions that we have come to expect do not have much effect on the core structure, so they can be developed in parallel. It should, therefore, be possible with a modest outlay (about 0.01% of the estimated true development cost of Linux) to develop, within one year, a complete system with all the applications required of a data server, workstation or personal computer and with a responsiveness and reliability that does not make you wish for your 1980s system.

There are three problems with this scenario.

The first is that the development would not be starting from a significantly more advanced position than 25 years ago because there has been almost no significant advance – just lots of hot air and recycling. It would be possible to leverage some peripheral technologies such as 16 bit (or more) character sets, glyph definitions and image and video file formats. However, unlike 25 years ago, the system would be constrained by not be able to set its own standards but having to cope with "industry standards" that have gone completely off the rails, making development significantly more difficult.

The second is that there do not seem to be any tools that are significantly more advanced than those available 25 years ago: if they do exist, it would probably take longer to track them down and evaluate them than to create suitable tools from scratch.

The next problem is building a market outside the established workstation market. There is no point in attacking established systems head-on in their own market, so systems must be targeted at new applications and markets that will be made possible by a "quantum leap" in price/performance, something that has been done many times over the past 25 years in many domains, and can be done again, even in the computer world. Already, there is visible discontent with the feature overloading that even cell phones now suffer from and there is a growing demand for a more lightweight approach ("less is more") in the developed world. More radically, the dramatic reduction in hardware costs and power consumption and dramatic improvement in systems accessibility that would come from adopting a more rational software technology opens up vast possibilities on the other side of the digital divide.

Finally there is the problem of finding the funding, not so much for the development, but for the marketing. There, the problem is that there are very few people of the stature of Sir Clive Sinclair. He looked at Unix and saw no justification for its appalling performance and fragility and he could not see why an equivalent system could not be implemented on a hobby computer. He bet his own money and was proved to be right. In the computer science world, challenging orthodoxy and being proved wrong is a forgiveable aberration, challenging orthodoxy and being proved right is totally unforgivable.

A Triangular Exercise

by George Gwilt

I recently wrote a PE program allowing a user to pick three points, A, B and C which were displayed on the screen with lines joining them. Then the angle ABC was calculated and shown. This seems simple enough. This is what I did.

Finding the Points

The first step was to find the positions of A, B and C. This was done by allowing the user to move the pointer to the position he wanted to choose and then clicking the mouse button. This was achieved by reading the pointer with the function BRPTR

The actual instruction was:

```
    ptr = BRPTR(10,oldk,wwd)
```

The number 10 causes the pointer's position to be returned either if the pointer has moved from the original position, oldk, or if the mouse key was pressed. The item "wwd" is the address of the window working definition.

The function BRPTR is part of TurboPTR but you can use the trap #3 routine IOP.RPTR to do the same thing in assembler if you want to do it that way.

Printing the Letters

When the user has chosen a position, the appropriate letter is printed at that position. Since the position of the pointer is given in absolute co-ordinates these have to be corrected to the relevant co-ordinates before printing can occur. The adjustment is made by subtracting the absolute position of the top left of the main window from the absolute pointer position and then, further, subtracting the origin of the sub-window in which the points appear. The information is found easily enough from the window working definition.

Once the position of the point is found, in pixel co-ordinates, the appropriate letter, A, B or C can be printed there. There is one problem here though. If the point is chosen too near the edge of the sub-window an out of range error will occur when I try and position the cursor. Thus, since I decided to print the letters so that their midpoint coincided with the point in question, I restricted the search for points to an area smaller than the sub-window in which the search is taking place by a margin of three pixels horizontally and five vertically at each edge.

I did this by means of the procedure SET_PTR which sets the pointer to a particular position. This procedure is based on IOP.SPTR

Drawing the Lines

An interesting problem arose when I tried to print lines from A to B and from B to C. I assume it is obvious that these should be drawn by using the graphics procedure LINE. However, although the keyword CURSOR and the underlying SD_GCUR allow the pixel positioning of the cursor relative to a graphics point, there seems no easy means of finding the graphics coordinate of a given pixel position.