Universidade Federal do Paraná

Departamento de Informática

Roberto A Hexsel

Do flying cars need wheels?

Relatório Técnico
RT_DINF 001/2015

Curitiba, PR
2015

Do flying cars need wheels?

Roberto A. Hexsel
Universidade Federal do Paraná
Departamento de Informática
roberto@inf.ufpr.br

Abstract

Until fairly recently (*circa* 2015), the design of the current desktop operating systems was premised on the use of slow magnetic disks. The introduction of non-volatile memory provides an opportunity for a complete re-design of operating systems. This text presents some of the issues an provides a few suggestions for areas that may benefit from a different perspective.

1 Introduction

If one is to design a flying car, does it have to be equipped with wheels, or three downward facing spikes would be enough? What of the wheel and tire industry? I'm thinking of Star Wars rather than Blade Runner.

If one is presented with a new device that improves on a pair of fundamental metrics by two to three orders of magnitude, at a reasonable price, would a radical new way of thinking be necessary?

If the promises come true, the "system people" are about to face the Mother of the Operating Systems Revolutions, triggered by the advent of non-volatile RAM (NVRAM). If these devices come into widespread use, the way we think about OS design and implementation must change rather dramatically.

This paper does not present any original results nor measurements. Rather it is a loose collection of ideas, most of them incomplete, that the author believes should be discussed in the community, specially amongst those involved in Education, as we are training the people who will implement the new breed of Flying Operating Systems.

Organizing the text is somewhat difficult because changes in one aspect of the system impact on others. The following sections contextualize NVRAM from an OS designer's point of view and discuss some implications of removing rotating magnetic media from the system. With secondary memory gone, the conceptual changes regarding files, processes, and security are discussed next.

Throughout, the discussion considers an Unix-like operating system, with processes, with virtual, and physical memory split into secondary and primary storage. The text was written with the intention of being processor agnostic, yet the author really likes the clean, lean and symmetric instruction set, plus the elegant application binary interfaces of MIPS processors.

2 NVRAM in context

"Tape is Dead. Disk is Tape. Flash is Disk, RAM Locality is King". Jim Gray

Flash memories have been around for some time, and commercial flash 'disk' drives are becoming affordable (in 2015). Solid state drives (SSD) are fast as there are no moving parts in them, their capacity is quite reasonable, and prices will fall, eventually. The durability problem has been solved and drives can be expected to be functional for several years, depending on usage patterns. SSDs are less power hungry than their electro-mechanical counterparts.

There are a few classes of devices that may be used to implement NVRAM, the most promising of these being phase change memory (PCM) [17][18]. PCM is faster than flash memory and has a longer endurance. PCM memory is slower than DRAM, less dense, and wears out faster, yet there are ways to compensate for these less attractive characteristics and employ PCM devices as a replacement for DRAM [15]. Surely, non-volatility is what makes NVRAM a Flying Memory.

Table 1 shows a (very) rough comparison of the characteristics of the storage technologies that are of interest to OS designers. These numbers are intended solely to provide approximations to the orders of magnitude for comparison. For simplicity, we take processor cycle time to be 0,5ns (2 GHz).

Table 1: Access time for storage technologies.

medium	disks	SSD	NVRAM	DRAM
access time	10ms	10μs	200ns	100ns
access time [cycles]	5×10^6	5×10^3	100	50

3 Do we need a scheduler?

"This is the short and the long of it". Shakespeare

A large part of what an operating system does is predicated on a storage technology that is some [12][9], to several [11], orders of magnitude slower than the processor, as shown in Table 1. If an access to secondary storage takes 10^5 to 10^6 processor cycles, it is well worth to switch the processor to another process, to hide the disk access latency. The OS enqueues the request, and the disk controller posts an interrupt to the processor when the requested block is sitting in DRAM. Having several processes performing concurrent accesses to disk is a clever trick to hide latency.

Consider how long it takes to save an execution context for a Unix process on a MIPS processor, which includes the PC, 31 general purpose registers (GPRs), and HI and LO registers. Saving state through non-cache-able memory needs 34 stores, at 50 cycles/store, adding up to 1,700 cycles. This is only half the time for a switch as the same number of registers must be restored for the incoming process. Thus far, 3,400 cycles, and not counting the 33 floating point/status registers. This is the minimum time it takes to save and restore state on a MIPS processor. As for lightweight threads, not the full set of registers may need be saved/restored, but unless there is hardware support for multiple contexts[1], many memory references are needed on each thread switch. Even for

[1]Multiple sets of registers/contexts is a feature found in MIPS32r2 and beyond.

x86 processors, with 8 integer and a plethora of vector/media dedicated registers, context switches cost anything but a trifle.

The interrupt service routine for the disk drive causes two mini-context switches, to save and then restore the processor registers which are needed to execute the handler, plus some more cycles to empty and then fill up the pipeline with instructions. Deeper and wider pipelines may take more than a couple of cycles to drain and fill up.

Under the far from innocent assumptions implied by Table 1, a fair proportion of the cycles needed to access a magnetic disk are spent on context switches. For the switch to be worthwhile, there better be several threads/processes vying for the processor. Unfortunately, current desktop users don't often have use for more than two to three concurrent threads/processes [5].

Thus far the *status quo* is this: it takes a very long time to access data on magnetic disks, the OS switches processes whenever there is a disk request in order to hide the disk latency, there are just a couple of processes to hide said latency. As an additional twist, most of the lap/desktop systems sold in 2015 have at least two processor cores, and at least one of them is idle most of the time. *Much ado about little. Not so good.*

The sequence of events in a request for a magnetic disk block includes (i) a context switch to another (often non-existing [5]) waiting process – to hide the disk latency; (ii) two mini-context switches for the disk interrupt service routine – to allow for the concurrent operation of disk driver and interrupted process(es).

If our desktop system were equipped with storage as fast as flash based SSDs, things would look very different [2][19]. If an access to an SSD block costs 5.000 cycles, there is no point in performing a context switch, which itself costs, almost the same number of cycles. Thus, whenever a process requests a block from an SSD, it is more efficient to synchronize the processor to the device by polling rather than by an interrupt. If the processor performs the copy from/to the device to/from memory, not making use of DMA, the block transfer may take less time than "the traditional way" – DMA transfer plus an interrupt to signal the end of transfer.

By using a storage medium that is *three* orders of magnitude faster, a large section of an OS can be done away with: (i) there is no need for a two-layer disk driver; (ii) there is no need for a queue of disk requests; (iii) there is no need for a context switch to hide the latency; and (iv) there is no need for the interrupt service routine and all the attending priority and timing complexities. *The car is barely touching the ground.*

Regarding the section title: *do we need a scheduler?* The answer is *yes*, but we need a simpler mechanism than the traditional one. There will be need for a scheduler to stop processes from hogging the processor since starvation is a evil thing and must be avoided. The scheduling of I/O requests changes radically when the secondary storage is so fast that there is no point in trying to hide its latency.

Space intentionally left blank.

4 Do we need paged virtual memory?

"Roads? Where we're going, we don't need roads". Dr. Emmett Brown

Paging is one large part of the mechanism responsible for hiding the latency of the secondary storage [12]. Paging participates in the automatic allocation of physical memory to processes, and is one of the protection mechanisms which is closer to the hardware.

The page table is a function that maps virtual addresses (VAs) onto physical addresses (PAs), or more specifically, the function maps virtual page numbers (VPNs) onto physical page numbers (PPNs). This function is logically implemented as the page table (PT) and each element of the PT holds one mapping ($\text{VPN} \mapsto \text{PPN}$). Each PT element also holds protection and accounting information, *viz.* the page is/is-not writable, the page has/has-not been referenced recently.

There is one PT per process and protection is enforced by managing the contents of the PTs only in kernel mode – in well designed systems, user processes do not even know there exists such a thing as a PT or even physical memory. The translation buffer (TB or TLB) is a fast, small, associative memory that sits near the processor and very quickly translates VPNs to PPNs. TBs hold translations for a small set of pages used in the near past.

Most desktop/server processors in current use are 64 bits wide, thus capable of referencing 16 exabytes (or the terrible name 16 exbibytes). The width of the physical address bus is slowly moving towards 60 bits, and an exabyte is a large memory indeed, by today's standards for primary memory.

Paging was invented to hide the latency of disc accesses *and* to amortize the cost of transferring a block of storage between primary and secondary storage. In essence, virtual memory was meant to give the programmer the impression of working with "infinite memory". The size of the address spaces referenced by ordinary programs used to grow at approximately 1/2 to 1 address bit per year, according to the first edition of [11] – programs roughly double in size every year.

What would be the use of paging if the primary memory were large *and non-volatile*? There are proposals for making the interface to persistent data byte addressable rather than block addressable [8][4]. This alone would bring a radical change in the way persistent information is stored and managed [1].

First, the easy part. If the adapter interface of SSDs were to allow byte/word transfers, many applications would benefit from the reduced traffic between memory and disk, but greater benefits would stem from the finer granularity of units of storage that ought to be kept consistent. Consider the complex machinery in data base systems which is necessary in order to keep consistency in records that are stored in a full disk block. Remove the 'block' from the system and locking may become a thing of the past. The result will be simpler, faster, and more reliable systems. The "block device interface" is an idea that percolates through several layers of the OS, and implementing a "character interface" for storage may imply the removal of a great deal of complexity [4].

Now for the revolutionary part. If, for instance, one half of the physical memory were populated with non-volatile RAM [6], then what would change in the OS? Let's take it a bit further, and assume that the capacity of DRAM+NVRAM is somewhere near the exabyte, on a machine with a 60 bit wide memory bus. Take pause. Ponder...

Harizopoulos *et alli.*, in [10], report that over 90% of the instructions – or processor cycles – spent on processing queries on an OLTP 'operating system' can be optimized away if the machinery added to cope with slow disks were removed. More than 90% is needless work.

The availability of a large non-volatile RAM would warrant a complete redesign of a large chunk of the OS. *Look! No wheels.*

5 Do we need a buffer cache?

"Defer no time, delays have dangerous ends". Shakespeare

With plentiful NVRAM, the magnetic disk becomes a third class citizen, not unlike a pen-drive, with a driver that is loaded only on demand, when the user must save or restore a copy of long term data, such as a movie or last year's income revenue forms. All the interesting action is now on the memory bus.

Some fast, volatile, RAM will always be needed to support the stack and some non-persistent data structures. NVRAM's access is 2–3× times slower than DRAM's. The time needed for NVRAM block updates can be effectively hidden [15] and there is no need for refresh cycles. Plus, NVRAM is naturally byte addressable.

Recall the discussion of fast SSDs in Section 3. If the secondary memory is so fast that polling is more efficient than rescheduling plus an interrupt, what is needed when the persistent primary memory is so fast that not even polling is needed, but just twice as many cycles as when waiting for a DRAM reference?

Under last paragraph's assumption there is no need for the buffer cache as a staging post between the slow disk and the fast RAM, since the (very large) NVRAM holds the needed files. Of course, some memory allocation mechanism is necessary, but it will probably not be on the critical path, as does the buffer cache, sitting between DRAM and disk.

The buffer cache interacts closely with paging in order to minimize the memory–disk traffic. With no disk, and no buffer cache, paging becomes a mechanism that performs memory allocation, and provides security through the separation of page tables and the bits of status of each individual page. These two functions may very well be implemented with Multics style segmentation [3].

Segmentation is closer to the way we think and program than paging is. Each program is split into a code segment, a data/heap segment and a stack segment, and these last two may grow to accommodate the program's dynamics. The OS maintains a small segment table for each process. A translation buffer keeps, near the processor, the base and limit 'registers', plus access rights information.

The hardware technology available in the late 1960s and early 70s was insufficient to implement an ambitious system such as Multics. The landscape looks very different now, and the hardware to efficiently support segmentation can be implemented without too much effort – essentially an adaptation of the paging TB, mapping variable size segments rather than pages.

6 Do we need files?

"Memory is the mother of all wisdom". Aeschylus

File systems are also build on the premise that secondary storage is implemented with slow and unreliable magnetic disks. I-nodes, complex indexing structures, journaling, *et alli.* are all artifacts of the sluggish magnetic disk. If, as suggested earlier, disks become lowly I/O devices, the implementation of the file abstraction can also change dramatically.

If our system has 'infinite' non-volatile memory, and is segmented, why not turn files into segments that remain in memory for a very long time? When a process opens a file, the system would add a new segment descriptor to the process' segment table. To close a file, the corresponding segment goes into the limbo of temporarily unused files. If a file is just another segment, that can be inexpensively added or removed from a program's address space, several premises that underpin file system design cease to hold.

Obviously, this discussion is glossing over many complex implementation details. The intention is to provoke and to present an alternative to the designs we grew accustomed to use and to think of as a 'file' and a "file system". The one thing that ought not to be lost is the clean abstraction for a file as being just a sequence of bytes.

7 Do we need processes?

"We will perform in measure, time and place". Shakespeare

This is one question I have no answers to offer. If the multiplexing of the processor is not as important as it once was, because the nature of the interactions with secondary storage changed in the ways suggested, does it make sense to think of computation as a set of processes competing for time on the processor? Do we need a new abstraction [16] for a 'process'?

8 Do we need security?

Surely! Flying cars come with seat belts, airbags, and parachutes.

Of course we need security. Security can be provided by the devices associated with the page table, and perhaps at less expense with the devices provided by segmentation, because less bits of state are needed to define the protection level of one segment than for a set of pages.

Complications do arise with non-volatile memory, and they are caused by the very non-volatility. If all the computation state is permanently kept on NVRAM, the cost of putting the computer to sleep/hibernate, and then waking it up is very small. Entire processes do not need to be copied to secondary storage; only the (smallish) fractions kept in DRAM need to be safely stored onto NVRAM – this operation is a copy from fast DRAM to not-so-fast NVRAM, and state is later copied back onto DRAM.

Now, what happens if there was a pointer in volatile memory pointing to a chunk of memory in non-volatile memory? Can that pointer be correctly recovered? Coburn *et alli.* [7] provide a solution with a data structure for a heap implementation that forbids dangling pointers.

For mobile devices, it would seem reasonable for all the user data to be stored in non-volatile memory. Only some small amount of DRAM would be needed to hold the stack and heap for the execution of applications. For security reasons, the user data must be encrypted before being stored; otherwise, if one were to lose his or her mobile device, all the personal records could be retrieved by accessing the device's NVRAM [1].

9 In conclusion

"The beginning is the most important part of the work". Plato

For over half a century the design of operating systems was predicated on fast primary memory (RAM) and slow secondary memory (magnetic disks). With the arrival of non-volatile RAM, this premise no longer holds and large sections of the OS can be eliminated or much simplified.

An arbitrarily ordered and non-exhaustive list includes the following changes. First, fast solid state drives eliminate the need for interrupts since polling is more efficient than a context switch plus interrupt service routine. Second, if the data interface of SSDs is re-designed so the transfer unit is one byte or one word, then the block interfaces that percolate through several layers of the OS may also be simplified away. Third, as there is no need to hide the long access latencies of disks by time multiplexing the processor, the scheduler can be re-designed. Fourth, as the secondary memory is replaced by non-volatile RAM (NVRAM) primary memory, the function of paging reduces to a mechanism for memory allocation and another for providing security, and both of these can just as well be provided by Multics-style segmentation. Fifth, file systems are conceived to match the organization magnetic disks and without these, files may just be segments that stay in memory for a long time. Sixth, the process abstraction may need to be re-thought as time multiplexing the processor loses importance. Seventh, non-volatility introduces its own artifacts such as non-volatile dangling pointers and the very persistence of sensitive data.

Acknowledgments

Some of the ideas presented here arose in discussions with PhD candidates Tiago Kepe and Ivan L Picoli and my colleagues Eduardo C de Almeida. Daniel Weingartner, Luis C E de Bona, Renato Carmo. The students enrolled in CI312 also provided invaluable input. Lauri P Laux Jr measured the memory spaces of several large applications and his work reinforces the idea that the time is ripe for segmentation [13, 14]. My first contact with the idea of "infinite memory" was in fruitful dialogue with Rodolfo Azevedo. The outrageous proposals in the text are all my own.

References

[1] Anirudh Badam. How persistent memory will change software systems. *IEEE Computer*, 46(8):45–51, Ago 2013.

[2] K Bailey, L Ceze, S D Gribble, and H M Levy. Operating system implications of fast, cheap, non-volatile memory. In *Proc USENIX Conf on Hot Topics in Operating Systems*, pages 2–2, 2011.

[3] A Bensoussan, C T Clingen, and R C Daley. The Multics virtual memory: Concepts and design. *Comm of the ACM*, 15(5):308–318, Mai 1972.

[4] M Bjørling, P Bonnet, L Bouganim, and N Dayan. The necessary death of the block device interface. In *Proc 6th Biennial Conf on Innovative Data Systems Research (CIDR13)*, 2013.

[5] G Blake, R G Dreslinski, T Mudge, and K Flautner. Evolution of thread-level parallelism in desktop applications. In *ISCA'10: 37th Intl Symp on Computer Arch*, pages 302–313, Jun 2010.

[6] A M Caulfield, A De, J Coburn, T I Mollow, R K Gupta, and S Swanson. Moneta: a high-performance storage array architecture for next-generation, non-volatile memories. In *Proc 43rd IEEE/ACM Int Symp on Microarchitecture (MICRO'10)*, pages 385–395, 2010.

[7] J Coburn, A M Caulfield, A Akel, L M Grupp, R K Gupta, R Jhala, and S Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. *SIGPLAN Not.*, 46(3):105–118, Mar 2011.

[8] J Condit, E B Nightingale, C Frost, E Ipek, B Lee, D Burger, and D Coetzee. Better I/O through byte-addressable, persistent memory. In *Proc ACM 22nd Symp Operating Systems Principles (SIGOPS)*, pages 133–146, 2009.

[9] Edsger W Dijkstra. The structure of the THE-multiprogramming system. *Comm of the ACM*, 11(5):341–346, Mai 1968.

[10] S Harizopoulos, D J Abadi, S Madden, and M Stonebraker. OLTP through the looking glass, and what we found there. In *Proc ACM Int Conf on Management of Data (SIGMOD'08)*, pages 981–992, 2008.

[11] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 5th edition, 2012.

[12] T Kilburn, D B G Edwards, M J Lanigan, and F H Sumner. One-level storage system. In *IRE Trans on Electronic Computers*, EC-11, pages 223–235, 1962.

[13] Lauri P Laux Jr and R A Hexsel. Back to the past: Segmentation with infinite and non-volatile memory. In *WSCAD-SSC'16: XVII Workshop em Sistemas Computacionais de Alto Desempenho*, pages 278–289, Out 2016.

[14] Lauri P Laux Jr and R A Hexsel. Back to the past: When segmentation is more efficient than paging. In *wperformance'18: XVII Workshop em Desempenho de Sistemas Computacionais e de Comunicação*, pages 278–289, Ago 2018.

[15] B C Lee, P Zhou, J Yang, Y Zhang, B Zhao, E Ipek, O Mutlu, and D Burger. Phase-change technology and the future of main memory. *IEEE Micro*, 30(1):143–143, Jan 2010.

[16] Edward A Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, Mai 2006.

[17] S Raoux., G W Burr, M J Breitwisch, C T Rettner, Y-C Chen, R M Shelby, M Salinga, D Krebs, S-H Chen, H-L Lung, and C H Lam. Phase-change random access memory: a scalable technology. *IBM J. Res. Dev.*, 52(4):465–479, Jul 2008.

[18] Kosuke Suzuki and Steven Swanson. The non-volatile memory technology database (nvmdb). Technical Report CS2015-1011, Dept of Computer Science & Engineering, Univ of California, San Diego, Mai 2015.

[19] J Yang, D B Minturn, and F Hady. When poll is better than interrupt. In *Proc 10th USENIX Conf on File and Storage Technologies*, pages 1–7, 2012.