Back to the Past: Segmentation with Infinite and Non-Volatile Memory

Lauri P Laux Jr, Roberto A Hexsel

Departamento de Informática, Universidade Federal do Paraná

{lpljunior,roberto}@inf.ufpr.br

Abstract. *The design of the current desktop/server operating systems is premised on the use of slow magnetic disks. Two recent developments, (i) RAM capacity nearing 2^{64} bytes, and (ii) the introduction of non-volatile memory (NVRAM), provide an opportunity for a complete re-design of traditional Unix-like operating systems. We discuss some of the issues which support that proposition and offer a few suggestions for areas that may benefit from looking back at pioneering work. We then propose a* segmented *memory model for the MIPS processor.*

Introduction

Esquecer é uma necessidade.
Machado de Assis, Verba testamentária.

If one is presented with a new device that improves 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? Would work done in the 1960s help? Our answer to both questions is 'yes', and we attempt to justify this answer in what follows.

The pioneering work in what we now call "operating systems" was undertaken in the late 1960s, early 1970s, and was premised on magnetic discs. Many of the design decisions were artifacts of the slow discs, and these include demand paging, 4Kbyte pages, the buffer cache, scheduling of the processor, and file systems designed around disk blocks. If we get rid of all these, then our systems may become simpler, smaller, faster and more reliable.

The early 1990s saw the transition, in processor design, from 32 to 64 bit CPUs. By the early 2000s, primary memories were implemented with 40-44 bit addresses. A decade later, physical addresses are in the range of 50-54 bits. By the 2020s we can expect physical memories with a full 64 bit addressing range. This growth is on par with the rule of thumb which states that "the memory needed by the average program grows from 1/2 to 1 address bit per year" (1st edition of [HP12]).

We highlight some of the opportunities presented by the feasibility of implementing very large primary memories comprising of DRAM and non-volatile RAM (NVRAM) – by 'large' we mean 2^{64} bytes. We discuss the advantages of segmented memory over demand paged memory, and give a sketch of an architecture for a memory system built of DRAM and NVRAM. We conclude with the design of a translation buffer to support a segmented memory system on MIPS processors.

Non-volatile RAM

Flash memories have been around for some time, and commercial flash 'disk' drives are becoming affordable. 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) [RBB+08, SS15]. PCM is faster than flash memory and is more durable. PCM memory is slower than DRAM, less dense, and wears out faster, yet there are ways to compensate for these less attractive characteristics. With these problems solved, systems could make use of PCM devices as a replacement for DRAM [LZY+10].

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 simplicity, we take processor cycle time to be 0.5ns (2 GHz). Thus an access to NVRAM takes twice as long as an access to DRAM – a factor of two, and an access to an SSD takes about 100 times as long as an access to DRAM.

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

Do we need a scheduler?

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. Regarding the section title: *do we need a scheduler?*, the answer is *yes*, but we may take advantage of simpler mechanisms. There will always be need for a scheduler to stop processes from hogging the processor, since starvation is a nasty thing and must be avoided.

A large part of what an operating system does is predicated on a storage technology that is some, to several, orders of magnitude slower than the processor [KELS62, HP12]. If an access to secondary storage takes 10^5 to 10^6 processor cycles, it pays 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 has been copied to 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-cacheable 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 contexts, several memory references are needed on each thread switch. Even for x86 processors, with 8-16 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 drain and then fill up the pipeline with instructions. Deeper/wider pipelines may take 100s 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 [BDMF10]. The situation may not be different with servers, as many applications are IO bound: a few long running processes perform many disc references in searching or filling the huge tables common in current large-scale applications.

To date, 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 2016 have at least four processor cores, and the majority of them are idle most of the time.

If our desktop system is equipped with storage as fast as flash based SSDs, things do look very different [BCGL11, YMH12]. If an access to an SSD block costs 5,000 cycles, there is no point in performing a context switch, which itself costs, almost that 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 with a DMA transfer followed by an interrupt to signal the end of transfer. Also, the cache hierarchy may be better utilized as the interrupt service routine pollutes the cache(s); after a context switch, the entering processes finds an empty cache, and by the time it has been refilled, that process may be switched out again.

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 disk latency; and (iv) there is no need for (some of) the interrupt service routine(s) and all the attending priority and timing intricacies.

What is the use of paged virtual memory?

Paging is one large part of the machinery responsible for hiding the latency of the secondary storage [KELS62]. 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 $\langle \text{VPN} \mapsto \text{PPN} \rangle$. Each PT element also holds protection and accounting information – whether the page is writable, plus update and reference bits.

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, application programmers 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. Protection faults are detected at the TB and are handled by OS code.

Most desktop/server processors in current use have datapaths which are 64 bits wide, thus capable of referencing 16 exabytes. The width of the physical address bus is edging past the 50 bits – one petabyte 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" [BCD72]. What then is the use of paging if the primary memory is large *and non-volatile*?

Harizopoulos *et alli.*, in [HAMS08], 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. In these applications, more than 90% of the instructions executed are needless work.

If, for instance, one half of the physical memory were populated with non-volatile RAM [CDC+10], 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 petabyte, on a machine with a 50 bit wide memory bus. The availability of a large non-volatile RAM would warrant a complete redesign of a large chunk of the OS.

Without slow disks, paging becomes a mechanism that performs memory allocation, and provides security through the separation of address spaces, which are mapped onto disjoint PTs, and the enforcement of access permissions for each individual page. These three functions may just as well be implemented with Multics style segmentation [BCD72].

There are proposals, *e.g.* [CNF+09, BBBD13], for changing the interface to persistent data from block-addressable to byte-addressable. This alone would bring a radical change to the way persistent information is stored and managed [Bad13]. If the adapter interface of SSDs were to allow byte/word transfers, many applications would benefit from the reduced traffic between primary and secondary memory, but greater benefits would stem from the finer granularity of units of storage that ought to be kept consistent. Consider the complex mechanisms in data base systems which are necessary in order to keep consistency in records that are stored in a full disk

block. Remove the 'block' from the picture and locking may become a thing of the past. The result will be simpler, faster, and more reliable database systems. The "block device interface" is an idea that percolates through several layers of the OS, and implementing a "character device interface" for storage may imply the removal of a great deal of complexity [BBBD13].

Segmentation is back on the agenda

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

The hardware technology available in the late 1960s and early 70s was insufficient to implement an ambitious system such as Multics [BCD72]. 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 instead of pages.

Hornyack *et.alli.* [HCG+15] present strong evidence that several large-scale applications would perform better if the memory allocation were segment-based instead of the current paging systems. These applications spend a great deal of time handling TLB misses and the performance loss ranges from a few percent of the execution cycles to 58% for certain workloads. Increasing the page size is not the solution because of the additional memory fragmentation it introduces, and large pages do not reduce significantly the number of memory mappings required.

Hornyack also shows that server-class memory-hungry applications use "virtual memory areas" which represent an item of code or data in a contiguous region of memory that spans from one to several contiguous virtual pages. Rather than using fixed size pages, hence large page tables, a segmented system would substantially reduce the amount of state needed to keep the protection and mapping information.

Do we need file systems?

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, are all artifacts of the sluggish magnetic disk. If disks become lowly I/O devices, in the same class as pen drives, the implementation of the file abstraction can also change dramatically.

Multics' designers suggested that the file system should be embedded in the virtual memory system [BCD72]. 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

cheaply 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, although most of the concepts, and even the high level implementation, were built in the Multics system [Gre93]. Our intention is to provoke and to present an alternative to the designs we grew accustomed to use and 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 just a sequence of bytes. [RT74].

Do we need processes?

Blake *et.alli.* [BDMF10] measured the number of active concurrent processes on actual desktop systems (Windows 7 and OS X) and found that two to three cores are "more than adequate for most applications". Put another way, to how many browser tabs one can look at simultaneously?

With, relatively speaking, abundant execution units – several processor cores – and infinite memory, what exactly are the processes competing for?

This is one question we have no good answers to offer. If the multiplexing of a single processor is not as important as it once was – because there are plenty of real/virtual cores available, and furthermore, the nature of interactions with secondary storage changed in the ways described in previous sections, does it make sense to think of computation as a set of processes competing for "time on the processor" and "pages of memory"? As advocated in [Lee06], do we need a new abstraction for what we name "a process"?

Do we need security?

Of course we need security. Security can be provided by the mechanisms and policies associated with the page table, and perhaps more cheaply with 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 non-volatility itself. 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.

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 in the case of a power failure? Coburn *et alli.* [CCA+11] provide a solution with a data structure for a heap implementation that forbids dangling pointers. As pointed out in [SB14], dangling pointers are a difficult problem, and no general solutions have yet been presented.

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[1] 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 [Bad13].

Data structures for segmented memory

The page table is a function with a domain that comprises the full virtual address space: $|\text{domain}| = 2^{64}/2^{12}$, assuming 4Kbyte pages. Even with a hierarchical implementation, large processes employ huge PTs.

How large is the PT of one large process? A process running the TPC-H benchmark may allocate the full 100Gbytes available on a NUMA machine [Alm16]. Considering an efficient, two-level PT implementation in which a second-level node (one 4Kbytes page) maps 1Mbytes, 1024 nodes map 1Gbytes, and 100K nodes map the physical memory needed. The page table itself uses up ≈ 100.000 pages.

Each process needs a segment table (ST), which is potentially much smaller than a page table, as the ST domain is the number of segments. An upper limit could be, somewhat arbitrarily, set to 1024 for (large) processes which link with many libraries, or which share several data segments with other process(es). For most small-ish processes, 32–64 segments would be plenty, so as to map 3 segments for code, heap and stack, 3 more for stdin, stdout, stderr, 2 for each library the process links in to, and one for each file kept open.

If a segment descriptor holds 64 bytes of addressing, protection and accounting information, a large process would need 64Kbytes for its ST whereas a small process would need only 4Kbytes.

For paging systems, the mappings of free/used pages can be implemented with bit maps as all pages are the same size. Systems with super-pages – more than one page size – can still use simple data structures to track the usage of page frames.

Keeping track of the free memory on a segmented system is more involved as the segments have variable size, and segments may grow during the lifetime of a process. In our proposed implementation (see Sec. 6), for reasons of speed, segment size is limited to be a power of 2. This can be exploited while implementing the data structures to keep track of the free and used memory, such as using the buddy algorithm to find, or create, space to accommodate a new segment.

Segmentation and fragmentation

One serious disadvantage of a segmented memory is fragmentation: as the processes come and go, the memory available for allocation to new processes is fragmented in smaller chunks. At some point, the small chunks have to be re-allocated in order for new segments to fit in the available space. Even if we speak of "infinite memory", if a system runs for long enough, its memory will become fragmented.

Some form of compaction must be employed to reclaim the scattered fragments. We propose a hardware mechanism, akin to DMA, to move the segments

[1] One of the authors contends that *all* data in NV memory ought to be encrypted. The other is not so paranoid.

onto more compacted allocations, by stealing free bus cycles. In order for this to be viable, the compiler must generate position independent code, and a given segment must be inactive while it is being moved. This DMA-like segment moving machine might also encrypt data as it is moved onto new locations.

Program updates in infinite, non-volatile memory

A problem related to compaction is the version update of running programs. We propose to separate processes into (at least) two classes: *ephemeral*, which are short-lived programs such as `ls`, `cat` and `gzip`; and *perennial*, such as Apache, data base managers or browsers.

In all cases, the segment descriptor for the program code must hold a reference count; if there are no instances executing, the old version is simply replaced by the new. The update of ephemeral programs may, occasionally entail the co-existence of two or three versions in the system – those which are running/suspended and the newly installed. When the reference count reaches zero, the old version is removed.

Perennial programs must include some functionality by which the OS informs of their eminent termination. When thus signaled, the process saves its state to NVRAM, informs the OS that it is in an 'upgrade-able' state and waits for the kill signal. The new version recovers the state from NVRAM and carries on executing. This behavior is similar to that of applications for mobile devices, except the amount of state to be saved might be very large.

Segmented memory for the MIPS processors

A conceptual model for a virtual memory system is shown in Figure 1. The program counter (PC) generates a reference stream for instructions, whereby the virtual address v_i is used to reference instruction i, and v_i is translated to the physical address p_i where i is actually held. The address generation unit (AGU) generates virtual address v_d to reference the datum d, which is held in RAM at address p_d.

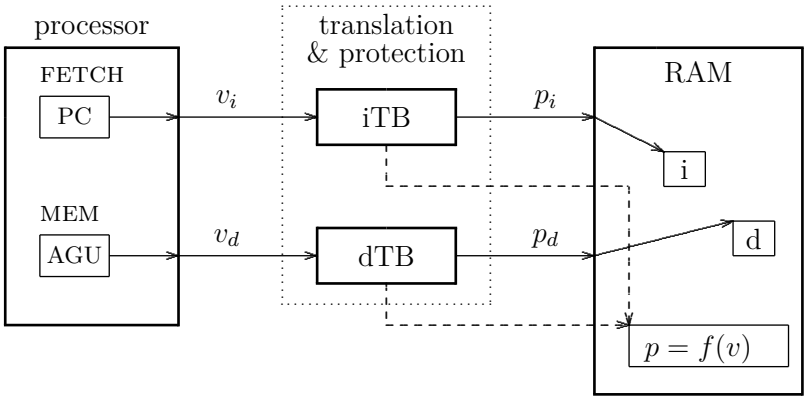

Figure 1. Conceptual model for virtual memory and protection.

Each virtual address reference is translated to its physical address reference by means of the translation table ($f(\,)$), which is held in RAM. The translation buffers iTB and dTB hold, in fast associative memory, a subset of the most recently referenced address translations, caching the most recently used pairs (v, p) in $f(\,)$.

The translation buffers iTB and dTB perform two important tasks: (i) to quickly, and often, provide a virtual-physical address translation, and (ii) to implement protection checks, by catching protection violations, and references to unmapped locations. If a protection violation occurs, the offending process is killed. If a reference to an unmapped location is benign – it can be fixed – then the function $f()$ is updated accordingly; if it is malicious – an illegal reference – then the process is killed.

Notice that the two preceding paragraphs describe highly desirable features of a memory system but do not mention demand paging. These features, protection and mapping, can be implemented just as well by means of segmentation. The iTB and dTB, plus the appropriate sections of the OS, can provide the same functionality as their demand paging counterparts by implementing variable sized segments.

Programs usually comprise three segments, text, data and stack. When one considers dynamically linked libraries, two additional segments (text and data) are needed for each library linked into the process. The segment translation buffers should be implemented to hold, at least, 16 segment descriptors, thus holding mappings for the process and a handful of libraries and/or open files. This should be easy to implement as 32-64 element TLBs are fairly common [HP12].

The MIPS architecture provides minimum hardware for implementing a virtual memory system. There is no hardware support for walking the page table, thus the OS designer is free to pick and choose the most efficient data structures. In theory, segmentation could be implemented on top of the existing TLB. We chose to design a new structure, with a clean programming interface, which is also more efficient than adding a software layer on top of an already complex piece of the OS.

Our design for a segment translation buffer (SB) attempts to keep the programming interface similar to that described in [MIP05]. Each element of the SB holds a 10 bit virtual segment number as tag (VSN), a 64 bit base address (Sbase) and a 64 bit segment limit (Slimit). The ASID register holds the address space identifier (ASID) of the process that owns the segment, segment size mask, plus status and permission bits (write-protected, referenced, modified, shared). The ASID may also be considered when checking the validity of a reference. Figure 2 shows a schematic diagram of the segment translation buffer.

A virtual address is split into its segment number (SN) and a displacement within that segment. The segment number is used to associatively search the SB; in case of a hit, the protection bits are checked, and the base+displacement is checked against the physical segment limit. If the checks succeed, then the physical address is sent to the memory; if they fail, the appropriate exception is raised. If the segment mapping is not present in the SB, an SB miss exception is raised, and the mapping is filled from the segment table.

We now go back to the discussion on segment table size, in Section 5.4. Large perennial processes need a 1024 element ST, hence the virtual address comprises a 10 bit VSN and a 54 bit displacement. Small ephemeral processes need a 64 element ST, with a 6 bit VSN and a 58 bit displacement. The different VSN sizes can be easily dealt with by masking off some of the virtual address bits while associatively

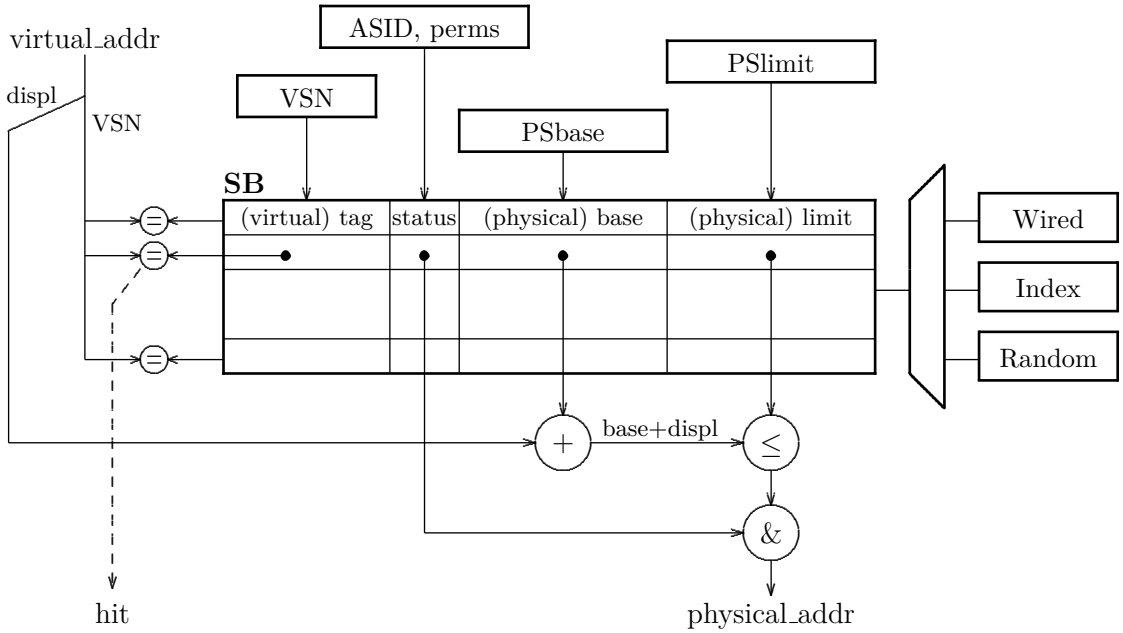

Figure 2. Segment translation buffer.

searching for a given VSN.

The programmer interface registers are similar to those in the MIPS paging TLB. To assemble an element to be inserted into the SB, the programmer must first write the appropriate values to the VSN, PSbase, PSlimit and ASID registers. The element of the SB to be written may be explicitly chosen by writing to the Index register, or may be randomly picked. The Wired register stops elements with index less than its contents from participating in the random replacements. The Random register is a free running counter that counts from Wired to the SB capacity.

There are two adders in the critical path to memory: the first adds the displacement to the physical base of the segment, and the second compares that address to the segment limit. The PSlimit register could hold a mask, in order to obviate the second sum: rather than a 64 bit adder, the limit could be and-ed to a mask, an operation that can be made faster than an addition. The downside of the mask is that it limits the segments to be powers of two, but this may not be a problem if we are thinking of infinite memory. Furthermore, managing fragmentation and the list of free memory may be easier if the memory is allocated in chunks which are powers of two.

In conclusion

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 evolve to a much simplified implementation.

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 changes into a mechanism for memory allocation plus a separate mechanism for enforcing security, and both of these can be provided by segmentation. Fifth, file systems are conceived to match the organization magnetic disks, and without these, files may be thought of as segments that stay in memory for a long time. Sixth, the process abstraction can be re-thought as time-multiplexing the processor loses the importance it once had. Seventh, non-volatility introduces its own artifacts such as non-volatile dangling pointers and the very persistence of sensitive data and these must be dealt with.

We propose segmentation as a more efficient means to provide virtual-to-physical memory mapping and security. We present the design for a segmentation buffer (similar to a TLB) for the MIPS processors. With this design completed, we will implement a segmented operating system and then compare its efficiency to that of a paging system.

Acknowledgments Some of the ideas presented here arose in discussions with PhD candidates Tiago Kepe and Ivan L Picoli and our colleagues Eduardo C de Almeida. Daniel Weingartner, Luis C E de Bona, Renato Carmo, and the students enrolled in ci312 (2015-1) also provided invaluable input. Our first contact with the idea of "infinite memory" was in fruitful dialogue with Rodolfo Azevedo.

References

[Alm16] Eduardo C Almeida. Memory footprint of large TCP-H benchmarks. Personal communication, UFPR, Sep 2016.

[Bad13] Anirudh Badam. How persistent memory will change software systems. *IEEE Computer*, 46(8):45–51, Aug 2013.

[BBBD13] 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.

[BCD72] A Bensoussan, C T Clingen, and R C Daley. The Multics virtual memory: Concepts and design. *Comm of the ACM*, 15(5):308–318, May 1972.

[BCGL11] 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.

[BDMF10] 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.

[CCA+11] 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.

[CDC+10] 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.

[CNF+09] 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.

[Gre93] Paul Green. Multics virtual memory – tutorial and reflections. Essay, Multics Project, 1993. `ftp://ftp.stratus.com/vos/multics/pg/mvm.html`.

[HAMS08] 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.

[HCG+15] P Hornyack, L Ceze, S Gribble, D Ports, and H M Levy. A study of virtual memory usage and implications for large memory. In *Proc Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, 2015.

[HP12] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 5th edition, 2012.

[KELS62] 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.

[Lee06] Edward A Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, May 2006.

[LZY+10] 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.

[MIP05] MIPS. MIPS32 architecture for programmers, volume III: The MIPS32 privileged resource architecture. Rev. 2.50, MIPS Technologies, 2005.

[RBB+08] 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.

[RT74] Dennis M Ritchie and Ken Thompson. The UNIX time-sharing system. *Comm of the ACM*, 17(7):365–375, Jul 1974.

[SB14] Karin Strauss and Doug Burger. What the future holds for solid-state memory. *IEEE Computer*, 47(1):24–31, Jan 2014.

[SS15] 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, May 2015.

[YMH12] 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.