VIRTUAL MEMORY SYSTEMS AND TLB STRUCTURES

Prof. Bruce Jacob
Dept. of Electrical & Computer Engineering
University of Maryland at College Park
http://www.ece.umd.edu/~blj/

1 Virtual Memory, a Third of a Century Later

Virtual memory was designed in the late 60’s to provide automated storage allocation. It is a technique for managing the resource of physical memory that provides to the application an illusion of a very large amount of memory—typically much larger than is actually available. In a virtual memory system, only the most-often used portions of a process’s address space actually occupy physical memory; the rest of the address space is stored on disk until needed. When the mechanism was invented, computer memories were physically large (one kilobyte of memory occupied a space the size of a refrigerator), they had access times comparable to the processor’s speed (both were extremely slow), and they came with astronomical price tags. Due to space and monetary constraints, installed computer systems typically had very little memory—usually less than the size of today’s on-chip caches, and far less than the users of the systems would have liked. The virtual memory mechanism was designed to solve this problem, by using a system’s disk space as if it were memory and placing into main memory only the data used most often.

Since then we have seen constant evolution (and revolution) in the computer industry. Typical microprocessors today have more on-chip cache than the core memory found in multi-million-dollar systems of yesterday and cost orders of magnitude less. Today, memory takes up very little space: You can easily hold a gigabyte of DRAM in your hand. In recent decades, processor designers have focused on improving speed while memory-chip designers have focused on improving storage size, and, as a result, memory is now extremely slow compared to processor speeds. Due to rapidly decreasing memory prices, it is usually possible to have enough memory in one’s machine to avoid using the disk as a back-up memory space. Many of today's machines generate 64-bit addresses, some even larger; most modern machines therefore reference 16 exabytes (16 giga-gigabytes) or more of data in their address space directly. The list goes on. In fact, one of the few things that has not changed since the development of virtual memory is the basic design of the virtual memory mechanism itself, and the one problem it was invented to solve—too little memory—is no longer a factor in most systems. However, the virtual memory mechanism has proven itself valuable in other areas besides extending the memory space. Today it is used in nearly every modern operating system because of the convenience offered by its features: It simplifies memory allocation and memory protection, and it provides an intuitive programming interface to the application—the "virtual machine" interface—that simplifies program design and provides a natural path to multitasking.

2 Caching the Process Address Space

A process operates in its own little world; this is the *virtual machine* paradigm, illustrated in Figure 1. Each running process generates addresses for loads and stores as if it has the entire machine to itself—as if the computer offers an extremely large amount of memory and no other processes are executing or consuming resources. This makes the job of the programmer and compiler much easier, because no details of the hardware or memory organization are necessary to build a program.

The operating system divides the process address space into equal-sized portions for ease of management; these divisions are called *virtual pages*. A page is usually a multiple of the unit of transfer that hard disks use, and in most operating systems ranges from several kilobytes to several dozen kilobytes. A page is never fragmented; if any data in a virtual page are in physical memory then all the data in that page are, and if any of the data in a virtual page are nonexistent or being held on disk then all the data are. When the word *page* is used in a verb form, it means to allow a section of memory to be virtual—to allow it to move freely between physical memory and disk. This allows the physical memory to be used more

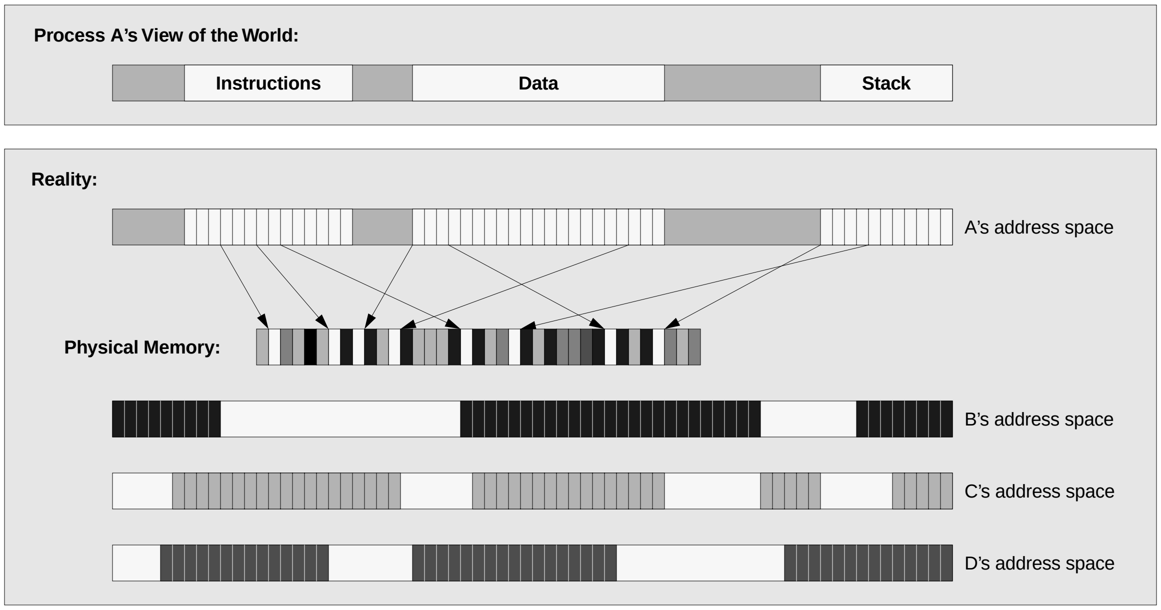

Figure 1: The virtual machine paradigm
A process operates in its own virtual environment, unaware that there are other processes executing and contending for the same limited resources. The operating system views each process address space as a collection of pages that can be cached in physical memory, or left in backing store.

efficiently: When a region of memory has not been used recently, the space can be freed up for more active pages, and pages that have been migrated to disk are brought back in as soon as they are needed again.

How is this done? The ultimate home for the process’s address space is *backing store*, usually a disk drive; this is where the process’s instructions and data come from and where all of its permanent changes go to. Every hardware memory structure between the CPU and the backing store is a cache for the instructions and data in the process’s address space. This includes main memory—main memory is really nothing more than a cache for a process’s virtual address space. A cache operates on the principle that a small, fast storage device can hold the most important data found on a larger, slower storage device, effectively making the slower device look fast. The large storage area in this case is the process address space, which can be many gigabytes in size. Everything in the address space initially comes from the program file stored on disk or is created on demand and defined to be zero. Figure 2 illustrates: There

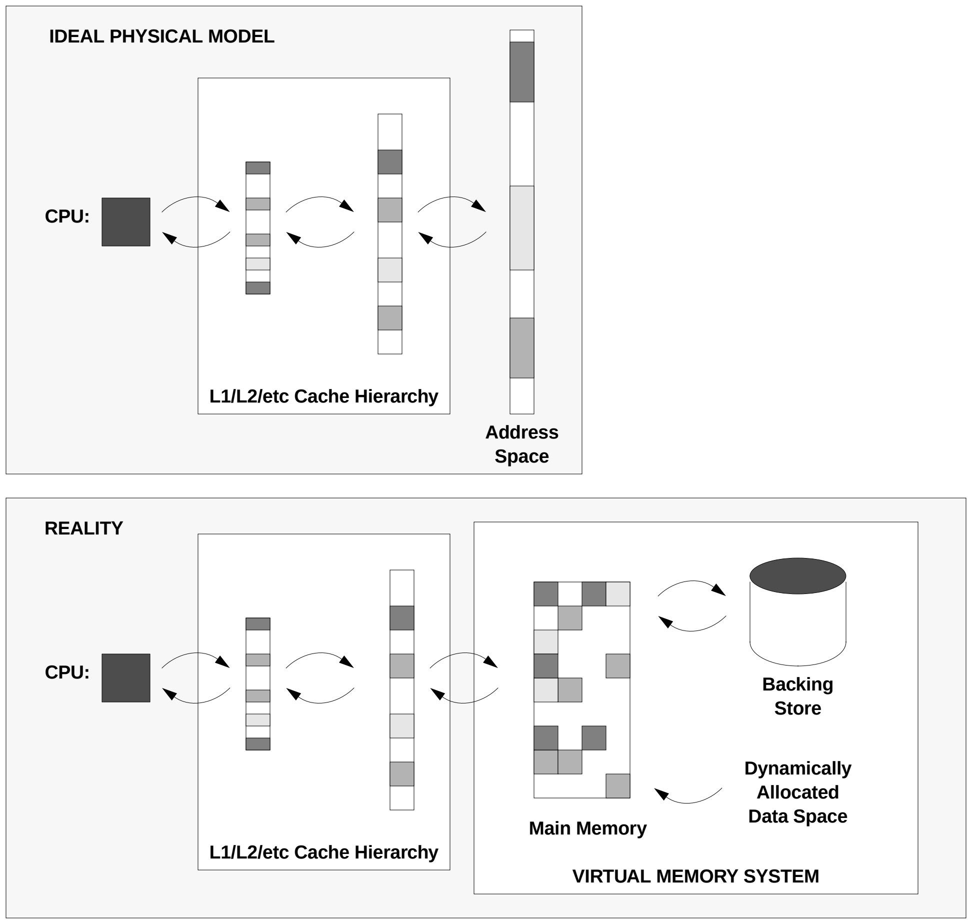

Figure 2: Caching the process address space
In the first view, a process is shown referencing locations in its address space. Note that all loads, stores, and fetches use virtual names for objects, and many of the requests can be satisfied by a cache hierarchy. The second view shows that the address space is not a linear object stored on some device, but is instead scattered across hard drives and dynamically allocated when necessary.

really is no linear array of data that houses the process address space. Its illusion is actually manufactured by the operating system through the virtual memory mechanism.

When a program first begins executing, the operating system copies a small portion of the process address space from the program file stored on disk into main memory. This typically includes the first page of instructions in the program and possibly a small amount of data that the program needs at start-up. Then, as more instructions or data are needed, the operating system brings in pages from the process's address on demand. This process, called *demand paging*, is depicted in Figure 3.

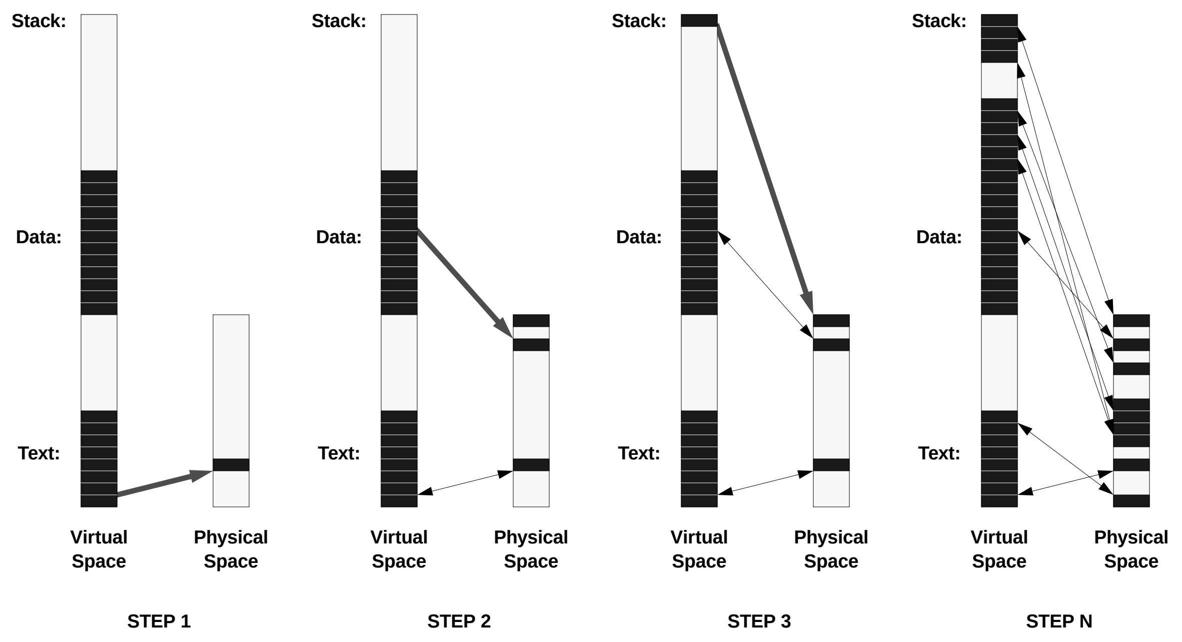

Figure 3: Demand paging at process start-up
In step 1, the operating system loads the first page of the process's instructions into physical memory, and sets the program counter to the first instruction in the program. This first instruction references a location in the process's data area, so in step 2 the operating system brings the corresponding data page into physical memory. The next instruction references a location on the process's stack, so in step 3 the operating system has allocated a stack page for the process and placed it into the process address space and main memory. Succeeding instructions reference more locations in the stack area, jump to instructions that lie outside of the initial page of instructions, and allocate extra data storage area on the heap. In step N (many steps later), these pages have been brought into main memory.

In step 1 of the figure, the operating system initializes a process address space and loads the first page of instructions into physical memory. The operating system then sets the hardware program counter to the first instruction in the program which sets the process running. Assuming that one of the first few instructions references the initialized data area, the uninitialized data area, or the (so far non-existent) stack, the operating system will have to bring in a page of data from the program file or create an uninitialized-data page or stack page and link it into the process address space. This is shown in steps 2 and 3 of the figure. When a process references an item in its address space that is not currently in physical memory, the reference causes a *page fault*, and the operating system loads the necessary pages from backing store into main memory. Clearly, the term *demand paging* refers to the fact that pages are allocated or brought into physical memory on demand. Step N of the figure shows a process that has been executing for some time, as it has several pages of data in its stack area, and several pages in its data area that were

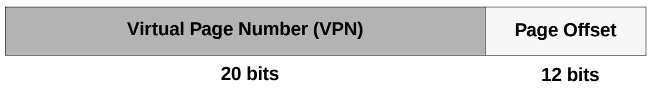

Figure 4: Virtual addresses
A Virtual address is divided into two components: the virtual page number and the page offset. The virtual page number identifies the page's location within the address space. The page offset identifies a byte's location within the page. Bit widths are shown for a 32-bit address and a 4K-byte page size.

not there when the process began executing. All of these pages were dynamically allocated by the operating system as the process needed or asked for them.

As has been pointed out before, the process is unaware of the operating system activity that moves pages in and out of main memory on its behalf. It typically does not know whether or not any given page is memory-resident or where it is located if it is memory-resident. Figure 1 at beginning of the section illustrates this by showing a process address space from two points of view. The first point of view is from the process itself; in most operating systems a process sees its address space as a contiguous span of memory locations from minimum to maximum. Somewhere in the address space is the program's instructions, or *text*; somewhere else is the program's data. Most operating systems also create a stack area, a heap area, and possibly one or more dynamically loaded libraries containing system-supplied utilities such as input/output routines or networking functions. The advantage of the virtual machine paradigm is that these can be arranged in physical memory however is most convenient, rather than having to fit things together like the pieces of a puzzle, as would be the case without address translation.

The second point of view in the figure is from the operating system. In reality, the process address space is not a large contiguous segment in physical memory but is partially cached by physical memory. Portions of the process address space are scattered about physical memory and are likely not contiguous at all. The process is unaware of where in the system any particular portion of its address space is being held; some portions can be on disk (for example, the portions of the program that have not been used yet), some can be in main memory, and some can be in hardware caches. The operating system maintains a map for each address space so that, for every virtual page in the address space, it can tell where in memory or on

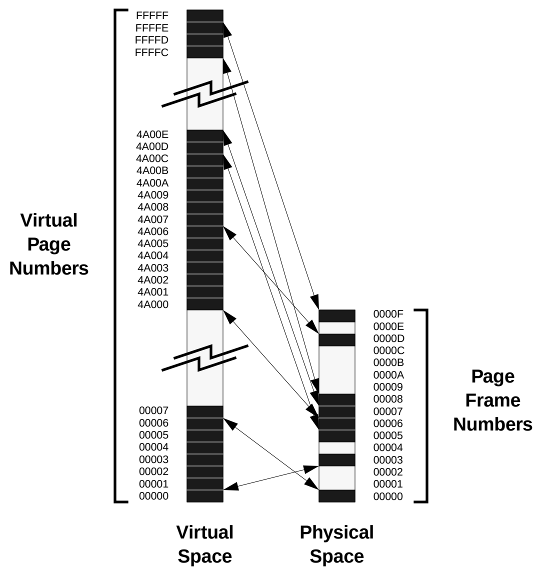

Figure 5: Page numbers (for 32-bit virtual addresses)
Every page in an address space is given a virtual page number (VPN). Every page in physical memory is given a physical page number, called a page frame number (PFN).

disk the page can be found. As the figure suggests, the virtual machine paradigm allows each process to behave as if it owns the entire machine; each process is protected from all others and does not even know that other processes exist—for example, a process cannot spoof the identity of another process, and the resource-management mechanisms implemented by the operating system to support the illusion that each process own all physical resources means that no process may dominate system resources. One of the many benefits of this organization is that it makes facilities such as multitasking very easy to implement, because process protection, resource sharing, and a clean division of process identity are provided as side-effects of the virtual machine paradigm by definition.

The mapping information that tells the location of pages in memory or on disk is organized into *page tables*, which are collections of *page table entries (PTEs)*. Virtual addresses (shown in Figure 4) are mapped at the granularity of *pages*; at its simplest, virtual memory is then a mapping of *virtual page numbers (VPNs)* to *page frame numbers (PFNs)*, shown in Figure 5. “Frame” in this context means “slot”—physical memory is divided into frames that hold pages. The page table holds one PTE for every

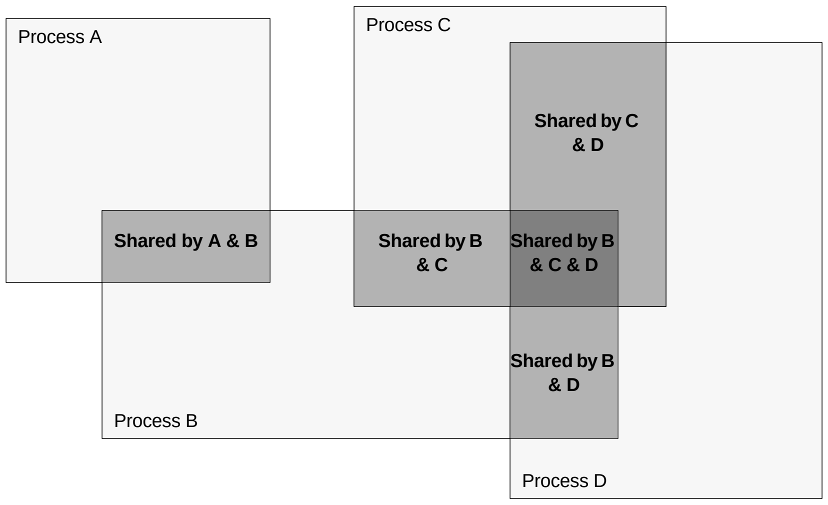

Figure 6: Shared memory

Shared memory allows processes to overlap portions of their address space while retaining protection for the non-intersecting regions; this is a simple and effective method for inter-process communication. Pictured are four process address spaces that have overlapped. The darker regions are shared by more than one process, while the lightest regions are still protected from other processes.

mapped virtual page; an individual PTE indicates whether its virtual page is in memory, on disk, or not allocated yet. The logical PTE therefore contains the VPN and either the page's location in memory (a PFN), or its location on disk (a disk block number). Depending on the organization, some of this information is redundant; actual implementations do not necessarily require both the VPN and the PFN. Later developments in virtual memory added such things as page-level protections; a modern PTE usually contains protection information as well, such as whether the page contains executable code, whether it can be modified, and if so by whom.

The mapping is a function; any virtual page can have only one location. However, the inverse map is not necessarily a function; it is possible and sometimes advantageous to have several virtual pages mapped to the same page frame (to share memory between processes or threads, or to allow different views of data with different protections, for example). Shared memory is one of the more commonly-used features of page tables. It is a mechanism whereby two address spaces that are protected from each other

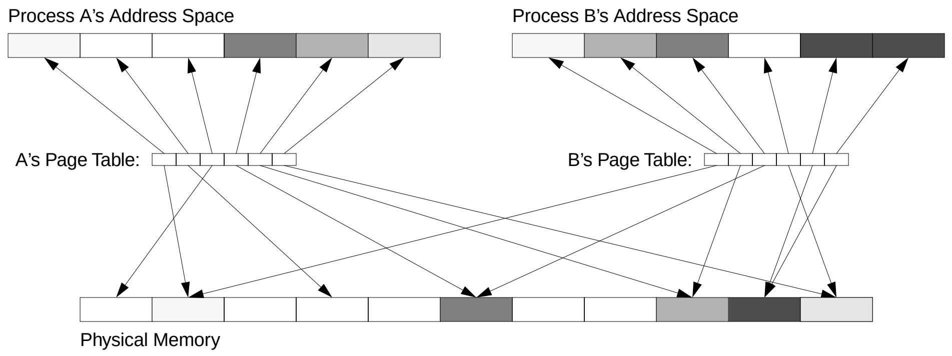

Figure 7: How page tables support shared memory
Two process address spaces are shown sharing several pages. Their page tables maintain information on where virtual pages are located in physical memory. The darkened pages are mapped to several locations; note that the darkest page is mapped at two locations in the same address space.

are allowed to intersect at points, still retaining protection over the non-intersecting regions. Several processes sharing portions of their address spaces are pictured in Figure 6. The shared memory mechanism only opens up a pre-defined portion of a process's address space; the rest of the address space is still protected, and even the shared portion is only unprotected for those processes sharing the memory. For instance, in the figure, the region of A's address space that is shared with process B is unprotected from whatever actions B might want to take, but it is safe from the actions of any other processes. Shared memory is therefore useful as a simple, secure means for inter-process communication. Shared memory also reduces requirements for physical memory; for example, in most operating systems, the text regions of processes are shared whenever multiple instances of a single program are run, or when multiple instances of a common library are used in different programs.

The mechanism works by ensuring that shared pages map to the same physical page; this is done by simply placing the same page frame number in the page tables of two processes sharing a page. A simple example is shown in Figure 7. Here, two very small address spaces are shown overlapping at several places, and one address space overlaps with itself; two of its virtual pages map to the same physical page.

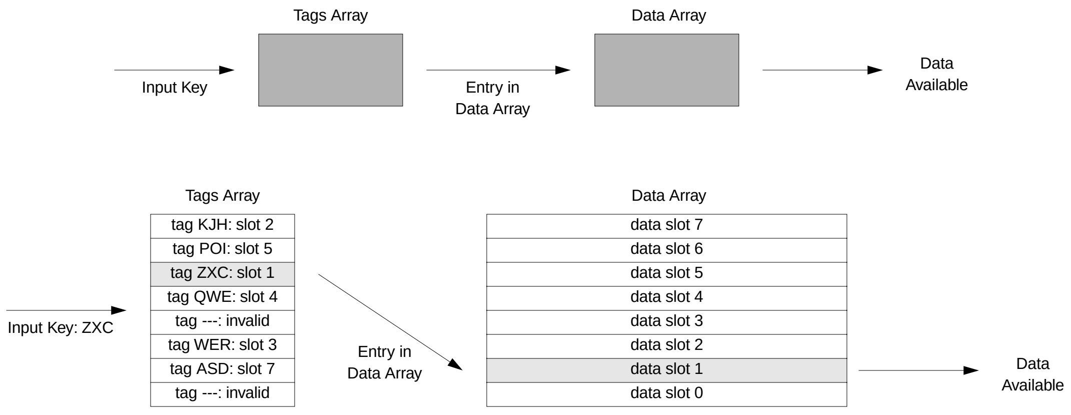

Figure 8: An idealized fully associative cache lookup
A cache is comprised of two parts: the tags array and the data array. The tags act as a database; they accept as input a key (a virtual address) and output either the location of the item in the data array, or an indication that the item is not in the data array. A fully associative cache allows an item to be located at any slot in the data array, thus the input key is compared against every key in the tags array.

This is not just a contrived example; many operating systems allow this, and it is useful for example in the implementation of user-level threads.

3 An Example Page Table Organization

So now the question is: How do page tables work? If we think of main memory as the data array of a cache, then the page table is the cache's corresponding *tags array*—it is a lookup-table that tells one what is currently stored in the data array. The traditional design of virtual memory uses a fully associative organization for main memory: Any virtual object can be placed at (more or less) any location in main memory, which reduces contention for main memory and increases performance. An idealized fully associative cache is pictured in Figure 8. A data tag is fed into the cache; the first stage compares the input tag to the tag of every piece of data in the cache. The matching tag points to the data's location in the cache. The goal of the page table organization is to support this lookup function as efficiently as possible.

To access a page in physical memory, it is necessary to look up the appropriate PTE to find where the page resides. This lookup can be simplified if PTEs are organized contiguously, so that a page number can be used as an offset to find the appropriate PTE. This leads to two primary types of page table organization: the *forward-mapped* or *hierarchical page table*, indexed by the virtual page number; and the *inverse-mapped* or *inverted page table*, indexed by the physical page number (page frame number). Each design has its strengths and weaknesses. The hierarchical table supports a simple lookup algorithm and simple sharing mechanisms but can require a significant fraction of physical memory. The inverted table supports efficient hardware table-walking mechanisms and requires less physical memory than a hierarchical table but inhibits sharing by not allowing the mappings for multiple virtual pages to exist in the table simultaneously if those pages map to the same page frame. Detailed descriptions of these can be found elsewhere [Jacob & Mudge 1998a].

Rather than describe all possible page table organizations, we will look in some detail at a concrete example: the virtual memory implementation of one of the oldest and simplest virtual memory systems, 4.3BSD Unix [Leffler et al. 1989]. The intent is to show how mapping information is used by the operating system and how the physical memory layout is organized. Version 4.3 of Berkeley Unix provides support for shared text regions, address space protection, and page-level protection. There is a separate page table for every process, and the page tables cannot be paged to disk. As we will see, address spaces are organized to minimize memory requirements.

BSD defines segments to be contiguous regions of virtual space. A process address space is composed of five primary segments: the *text* segment, holding the executable code; the *initialized data* segment, containing those data that are initialized to specific non-zero values at process start-up; the *bss* segment, containing data initialized as zero at process start-up; the *heap* segment, containing uninitialized data and the process’s heap; and the *stack*. Beyond the stack is a region holding the kernel’s stack (used when executing system calls on behalf of this process, for example) and the *user struct*, a kernel data structure holding a large quantity of process-specific information. Figure 9 illustrates the layout of these segments in a process’s address space: The initialized data segment begins immediately after the text

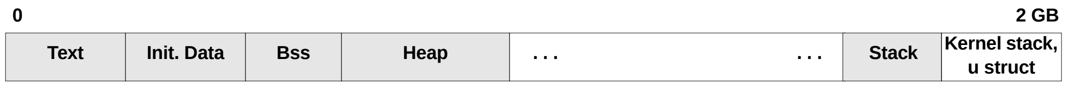

Figure 9: The 4.3BSD per-process virtual address space

segment, the bss segment begins immediately after the initialized data segment, and the heap segment begins immediately after the bss segment. This is possible because the text, initialized data, and bss regions by definition cannot change size during the execution of a process. The heap segment can grow larger, as can the stack. Therefore, these two begin at opposite ends of the address space and grow towards each other. Beyond the 2GB point, the address space belongs to the kernel; a user reference causes an exception.

There are a number of reasons why this design makes sense. When the OS was designed, memory was at a premium. The choice was made to wire down the page tables. Given this, it makes most sense to restrict an address space to be composed of a minimal number of contiguous regions; this would ensure a compact page table (contiguous pages implies densely-packed PTEs). The process model includes a single thread of execution per address space; 4.3BSD did not have multiple threads within an address space, nor did it use dynamically loaded libraries. Therefore, there was no need to support sparsely populated address spaces.

Figure 10 depicts the layout of process address spaces and the associated process page tables. The page tables are kept in the kernel’s virtual address space and are relocatable even if wired down. As shown in the figure, each user-process page table mirrors the process’s address space; the PTEs that map the text, data, bss, and heap segments are at the bottom end of a contiguous range of PTEs (which are held in the kernel’s virtual pages), and the PTEs that map the user’s stack are near the top of the range of PTEs. A user page table is therefore as compact as it can be, with no more than a page of wasted space; the empty space between the ranges of PTEs allows for expansion of the heap and stack segments.

When a process needs to expand its address space beyond the confines of its user page table, the operating system adds an additional page to the page table and shifts all following process page tables up

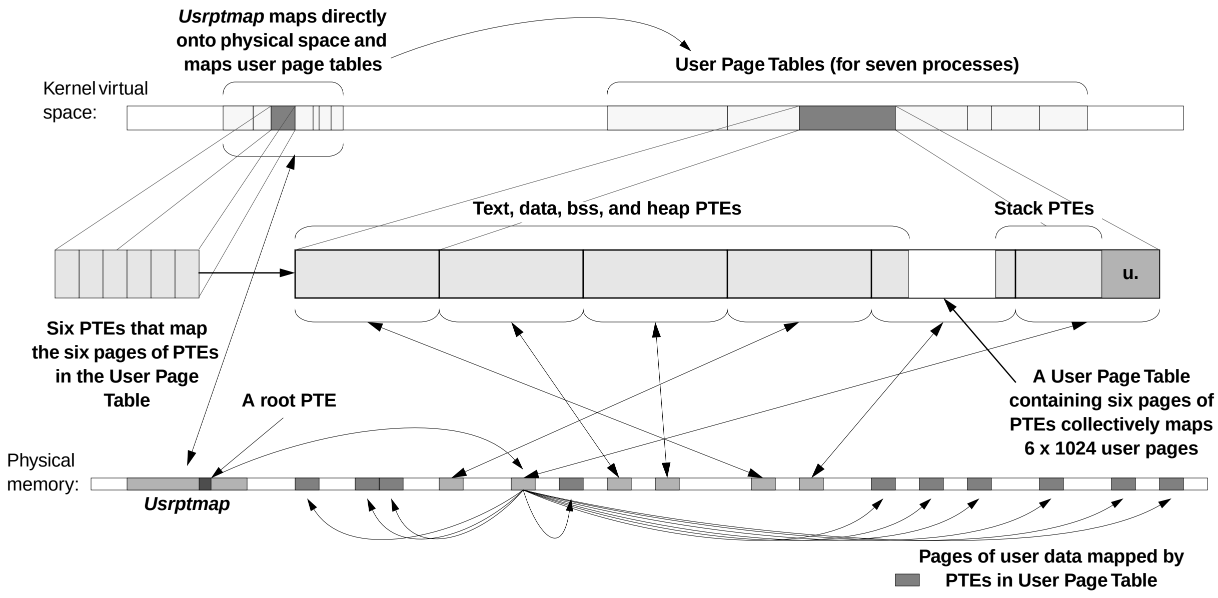
Usrptmap maps directly onto physical space and maps user page tables
User Page Tables (for seven processes)
Kernel virtual space:
Text, data, bss, and heap PTEs
Stack PTEs
u.
Six PTEs that map the six pages of PTEs in the User Page Table
A root PTE
A User Page Table containing six pages of PTEs collectively maps 6 x 1024 user pages
Physical memory:
Usrptmap
Pages of user data mapped by PTEs in User Page Table

Figure 10: User-process page tables in 4.3BSD Unix

by one virtual page. This is the advantage of placing the user page tables in virtual space; the displaced data need not be recopied. The disadvantage is that there needs to be another level of mapping to determine where in the physical memory the pages that comprise a process's user page table are located. The *Usrptmap* is a structure that mirrors the entire set of user page tables, and for every page in a process's user page table, there is one PTE in the Usrptmap.

When a user reference requires a lookup in the page table, the operating system first determines which process caused the fault; this identifies the appropriate page table within the region of user page tables. The operating system then determines whether the access was to the user's stack or one of the text, bss, or data segments. If the access is to the user's stack, the operating system indexes backward from the top of the appropriate user page table to find the PTE; if the access is to the text, data, bss, or heap segment the operating system indexes forward from the bottom of the user page table.

The *usrptmap* begins at a known location in physical memory; therefore, any process address space can be mapped. The appropriate root PTE within the *usrptmap* can always be found, given a process ID, and each root PTE points to a page of PTEs in physical memory, each of which then points to a page in the user address space.

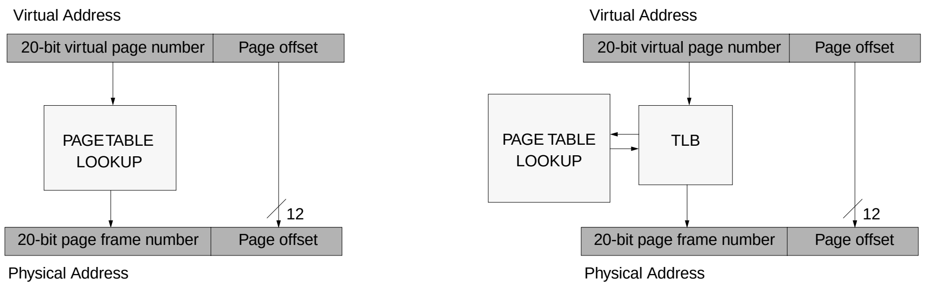

Figure 11: Address translation with and without a TLB
Address translation without a TLB is shown on the left; translation with a TLB is shown on the right. The only difference is that the TLB caches the most recently used entries in the page table, and the page table is only referenced when a lookup misses the TLB.

4 Translation Lookaside Buffers: Caching the Page Table

There is an obvious question of performance to consider: If every memory access by a user program requires a lookup to the page table, how does anything ever get done? The answer is a familiar one: we cache things. Rather than perform a page-table lookup on every memory reference (which returns a PTE that gives us mapping information), we cache the most frequently used PTEs in hardware. The hardware structure is called a *translation lookaside buffer* (TLB), and because it holds mapping information, the hardware can perform the address translations of those PTEs that are currently cached in the TLB without having to access the page table. Figure 11 illustrates. If the appropriate PTEs are stored in hardware, a memory reference completes at the speed of hardware, rather than being limited by the speed of looking up PTEs in the page table.

Most architectures provide a TLB to support memory management; the TLB is a special-purpose cache that holds only virtual-physical mappings. When a process attempts to load from or store to a virtual address, the hardware searches the TLB for the virtual address's mapping. If the mapping exists in the TLB, the hardware can translate the reference to a physical address without the aid of the page table. If the mapping does not exist in the TLB (an event called a *TLB miss*), the process cannot continue until the correct mapping information is loaded into the TLB.

Translation lookaside buffers are fairly large; they usually have on the order of 100 entries, making them several times larger than a register file. They are typically fully associative, and they are often accessed every clock cycle. In that clock cycle they must translate both the I-stream and the D-stream. Thus, they are often split into two halves, each devoted to translating either instruction or data references. They can constrain the chip's clock cycle as they tend to be fairly slow, and they are also power-hungry (both are a function of the TLB's high degree of associativity).

In general, if the necessary translation information is on-chip in the TLB, the system can translate a virtual address to a physical address without requiring an access to the page table. In the event that the translation information is not found in the TLB, one must search the page table for the translation and insert it into the TLB before processing can continue. This activity can be performed by the operating system or by the hardware directly; a system is said to have a *software-managed TLB* if the OS is responsible, or a *hardware-managed TLB* if the hardware is responsible. The classic hardware-managed design, as seen in the DEC VAX, GE 645, PowerPC, and Intel x86 architectures [Clark & Emer 1985, Organick 1972, IBM & Motorola 1993, Intel 1993], provides a hardware state machine to perform this activity; in the event of a TLB miss, the state machine would walk the page table, locate the translation information, insert it into the TLB, and restart the computation. Software-managed designs are seen in the Compaq Alpha, the SGI MIPS processors, and the Sun SPARC architecture [Digital 1994, Kane & Heinrich 1992, Weaver & Germand 1994].

The performance difference between the two is due to the page table lookup and the method of operation. In a hardware-managed TLB a hardware state machine walks the page table; there is no interaction with the instruction cache. By contrast, the software-managed design uses the general interrupt mechanism to invoke a software TLB miss-handler—a primitive in the operating system usually 10-100 instructions long. If this miss-handler is not in the instruction cache at the time of the TLB miss exception, the time to handle the miss can be much longer than in the hardware-walked scheme. In addition, the use of the general-purpose interrupt mechanism adds a number of cycles to the cost by draining the pipeline and flushing a possibly large number of instructions from the reorder buffer; this can add up to something on the order of 100 cycles. This is an overhead that the hardware-managed TLB does not incur; when hardware walks the page table, the pipeline is not flushed, and in some designs (notably the Pentium Pro [Upton 1997]), the pipeline keeps processing in parallel with the TLB-miss handler those instructions that are not dependent on the one that caused the TLB miss. The benefit of the software-managed TLB design is that it allows the operating system to choose any organization for the page table, while the hardware-managed scheme defines an organization for the operating system. If TLB misses are infrequent, the flexibility afforded by the software-managed scheme can outweigh the potentially higher per-miss cost of the design. For the interested reader, a survey of hardware mechanisms is provided in [Jacob & Mudge 1998b], and a performance comparison of different hardware/operating-system combinations is provided in [Jacob & Mudge 1998c].

Lastly, to put modern implementations in perspective, note that TLBs are not a necessary component for virtual memory, though they are used in every contemporary general-purpose processor. Virtually addressed caches would suffice because they are indexed by the virtual address directly, requiring address translation only on the (hopefully) infrequent cache miss. Such a scheme is detailed and evaluated in [Jacob & Mudge 2001].

5 Acknowledgments

Bruce Jacob is supported in part by the National Science Foundation under NSF CAREER Award CCR-9983618 and NSF grant EIA-0000439, by the Department of Defense under the MURI award AFOSR-F496200110374, and by Compaq and IBM.

6 References

D. W. Clark and J. S. Emer. "Performance of the VAX-11/780 translation buffer: Simulation and measurement." *ACM Transactions on Computer Systems*, 3(1). 1985.

Digital. *DECchip 21064 and DECchip 21064A Alpha AXP Microprocessors Hardware Reference Manual*. Digital Equipment Corporation, Maynard MA. 1994.

IBM and Motorola. *PowerPC 601 RISC Microprocessor User's Manual*. IBM Microelectronics and Motorola. 1993.

Intel. *Pentium Processor User's Manual*. Intel Corporation, Mt. Prospect IL. 1993.

Bruce Jacob and Trevor Mudge. "Virtual memory: Issues of implementation." *IEEE Computer* vol. 31, no. 6, pp. 33–43. June 1998a. <http://www.ece.umd.edu/~blj/papers/computer31-6.pdf>

Bruce Jacob and Trevor Mudge. "Virtual memory in contemporary microprocessors." *IEEE Micro* vol. 18, no. 4, pp. 60–75. July/August 1998b. <http://www.ece.umd.edu/~blj/papers/micro18-4.pdf>

Bruce Jacob and Trevor Mudge. "A look at several memory management units, TLB-refill mechanisms, and page table organizations." In *Proc. Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*, pp. 295–306. San Jose CA, October 1998c.

Bruce Jacob and Trevor Mudge. "Uniprocessor virtual memory without TLBs." *IEEE Transactions on Computers* vol. 50, no. 5. May 2001. <http://www.ece.umd.edu/~blj/papers/ieeetc50-5.pdf>

G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice-Hall, Englewood Cliffs NJ. 1992.

Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley Publishing Company, 1989.

E. I. Organick. *The Multics System: An Examination of its Structure*. The MIT Press, Cambridge MA. 1972.

M. Upton. *Personal communication*. 1997.

D. L. Weaver and T. Germand, editors. *The SPARC Architecture Manual version 9*. PTR Prentice Hall, Englewood Cliffs NJ. 1994.