

Chapter 10

Virtual Memory

Processes in a system share the CPU and main memory with other processes. However, sharing the main memory poses some special challenges. As demand on the CPU increases, processes slow down in some reasonably smooth way. But if too many processes need too much memory, then some of them will simply not be able to run. When a program is out of space, it is out of luck.

Memory is also vulnerable to corruption. If some process inadvertently writes to the memory used by another process, that process might fail in some bewildering fashion totally unrelated to the program logic.

In order to manage memory more efficiently and with fewer errors, modern systems provide an abstraction of main memory known as *virtual memory (VM)*. Virtual memory is an elegant interaction of hardware exceptions, hardware address translation, main memory, disk files, and kernel software that provides each process with a large, uniform, and private address space. With one clean mechanism, virtual memory provides three important capabilities. (1) It uses main memory efficiently by treating it as a cache for an address space stored on disk, keeping only the active areas in main memory, and transferring data back and forth between disk and memory as needed. (2) It simplifies memory management by providing each process with a uniform address space. (3) It protects the address space of each process from corruption by other processes.

Virtual memory is one of the great ideas in computer systems. A major reason for its success is that it works silently and automatically, without any intervention from the application programmer. Since virtual memory works so well behind the scenes, why would a programmer need to understand it? There are several reasons.

- *Virtual memory is central.* Virtual memory pervades all levels of computer systems, playing key roles in the design of hardware exceptions, assemblers, linkers, loaders, shared objects, files, and processes. Understanding virtual memory will help you better understand how systems work in general.
- *Virtual memory is powerful.* Virtual memory gives applications powerful capabilities to create and destroy chunks of memory, map chunks of memory to portions of disk files, and share memory with other processes. For example, did you know that you can read or modify the contents of a disk file by reading and writing memory locations? Or that you can load the contents of a file into memory without doing any explicit copying? Understanding virtual memory will help you harness its powerful capabilities in your applications.

- *Virtual memory is dangerous.* Applications interact with virtual memory every time they reference a variable, dereference a pointer, or make a call to a dynamic allocation package such as `malloc`. If virtual memory is used improperly, applications can suffer from perplexing and insidious memory-related bugs. For example, a program with a bad pointer can crash immediately with a “Segmentation fault” or a “Protection fault”, run silently for hours before crashing, or scariest of all, run to completion with incorrect results. Understanding virtual memory, and the allocation packages such as `malloc` that manage it, can help you avoid these errors.

This chapter looks at virtual memory from two angles. The first half of the chapter describes how virtual memory works. The second half describes how virtual memory is used and managed by applications. There is no avoiding the fact that VM is complicated, and the discussion reflects this in places. The good news is that if you work through the details, you will be able to simulate the virtual memory mechanism of a small system by hand, and the virtual memory idea will be forever demystified.

The second half builds on this understanding, showing you how to use and manage virtual memory in your programs. You will learn how to manage virtual memory via explicit memory mapping and calls to dynamic storage allocators such as the `malloc` package. You will also learn about a host of common memory-related errors in C programs and how to avoid them.

10.1 Physical and Virtual Addressing

The main memory of a computer system is organized as an array of M contiguous byte-sized cells. Each byte has a unique *physical address* (*PA*). The first byte has an address of 0, the next byte an address of 1, the next byte an address of 2, and so on. Given this simple organization, the most natural way for a CPU to access memory would be to use physical addresses. We call this approach *physical addressing*. Figure 10.1 shows an example of physical addressing in the context of a load instruction that reads the word starting at physical address 4.

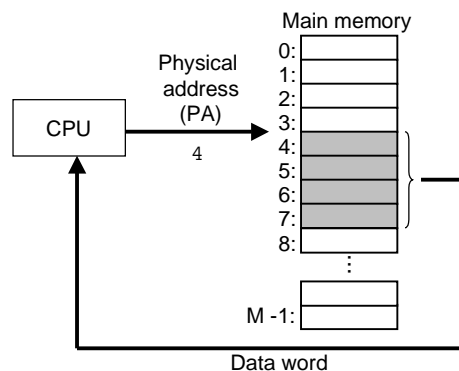


Figure 10.1: **A system that uses physical addressing.**

When the CPU executes the load instruction, it generates an effective physical address and passes it to main memory over the memory bus. The main memory fetches the four-byte word starting at physical address 4 and returns it to the CPU, which stores it in a register.

Early PCs used physical addressing, and systems such as digital signal processors, embedded microcontrollers, and Cray supercomputers continue to do so. However, modern processors designed for general-purpose computing use a form of addressing known as *virtual addressing*. (See Figure 10.2.)

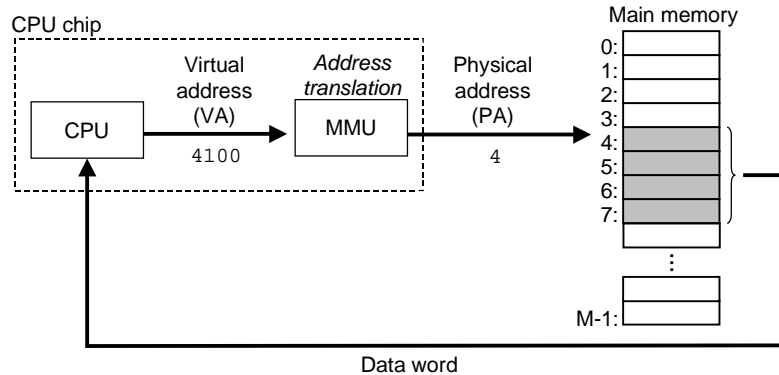


Figure 10.2: **A system that uses virtual addressing.**

With virtual addressing, the CPU accesses main memory by generating a *virtual address (VA)*, which is converted to the appropriate physical address before being sent to the memory. The task of converting a virtual address to a physical one is known as *address translation*. Like exception handling, address translation requires close cooperation between the CPU hardware and the operating system. Dedicated hardware on the CPU chip called the *memory management unit (MMU)* translates virtual addresses on the fly, using a look-up table stored in main memory whose contents are managed by the operating system.

10.2 Address Spaces

An *address space* is an ordered set of nonnegative integer addresses

$$\{0, 1, 2, \dots\}.$$

If the integers in the address space are consecutive, then we say that it is a *linear address space*. To simplify our discussion, we will always assume linear address spaces. In a system with virtual memory, the CPU generates virtual addresses from an address space of $N = 2^n$ addresses called the *virtual address space*:

$$\{0, 1, 2, \dots, N - 1\}.$$

The size of an address space is characterized by the number of bits that are needed to represent the largest address. For example, a virtual address space with $N = 2^n$ addresses is called an n -bit address space. Modern systems typically support either 32-bit or 64-bit virtual address spaces.

A system also has a *physical address space* that corresponds to the M bytes of physical memory in the system:

$$\{0, 1, 2, \dots, M - 1\}.$$

M is not required to be a power of two, but to simplify the discussion we will assume that $M = 2^m$.

The concept of an address space is important because it makes a clean distinction between data objects (bytes) and their attributes (addresses). Once we recognize this distinction, then we can generalize and allow each data object to have multiple independent addresses, each chosen from a different address space. This is the basic idea of virtual memory. Each byte of main memory has a virtual address chosen from the virtual address space, and a physical address chosen from the physical address space.

Practice Problem 10.1:

Complete the following table, filling in the missing entries and replacing each question mark with the appropriate integer. Use the following units: $K = 2^{10}$ (Kilo), $M = 2^{20}$ (Mega), $G = 2^{30}$ (Giga), $T = 2^{40}$ (Tera), $P = 2^{50}$ (Peta), or $E = 2^{60}$ (Exa).

# virtual address bits (n)	# virtual addresses (N)	Largest possible virtual address
8		
	$2^? = 64K$	
		$2^{32} - 1 = ?G - 1$
	$2^? = 256T$	
64		

10.3 VM as a Tool for Caching

Conceptually, a virtual memory is organized as an array of N contiguous byte-sized cells stored on disk. Each byte has a unique virtual address that serves as an index into the array. The contents of the array on disk are cached in main memory. As with any other cache in the memory hierarchy, the data on disk (the lower level) is partitioned into blocks that serve as the transfer units between the disk and the main memory (the upper level). VM systems handle this by partitioning the virtual memory into fixed-sized blocks called *virtual pages (VPs)*. Each virtual page is $P = 2^p$ bytes in size. Similarly, physical memory is partitioned into *physical pages (PPs)*, also P bytes in size. (Physical pages are also referred to as *page frames*.)

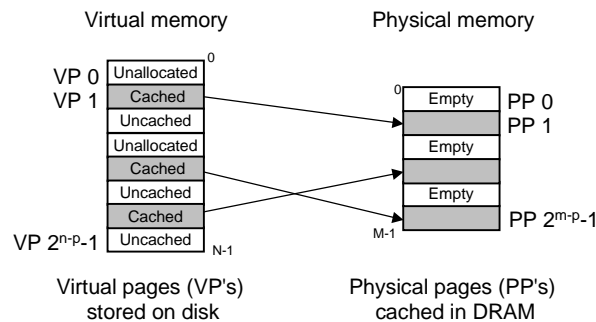


Figure 10.3: **How a VM system uses main memory as a cache.**

At any point in time, the set of virtual pages is partitioned into three disjoint subsets:

- **Unallocated:** Pages that have not yet been allocated (or created) by the VM system. Unallocated blocks do not have any data associated with them, and thus do not occupy any space on disk.

- **Cached:** Allocated pages that are currently cached in physical memory.
- **Uncached:** Allocated pages that are not cached in physical memory.

The example in Figure 10.3 shows a small virtual memory with 8 virtual pages. Virtual pages 0 and 3 have not been allocated yet, and thus do not yet exist on disk. Virtual pages 1, 4, and 6 are cached in physical memory. Pages 2, 3, 5, and 7 are allocated, but are not currently cached in main memory.

10.3.1 DRAM Cache Organization

To help us keep the different caches in the memory hierarchy straight, we will use the term *SRAM cache* to denote the L1 and L2 cache memories between the CPU and main memory, and the term *DRAM cache* to denote the VM system's cache that caches virtual pages in main memory.

The position of the DRAM cache in the memory hierarchy has a big impact on the way that it is organized. Recall that a DRAM is about 10 times slower than an SRAM and that disk is about 100,000 times slower than a DRAM. Thus, misses in DRAM caches are very expensive compared to misses in SRAM caches because DRAM cache misses are served from disk, while SRAM cache misses are usually served from DRAM-based main memory. Further, the cost of reading the first byte from a disk sector is about 100,000 times slower than reading successive bytes in the sector. The bottom line is that the organization of the DRAM cache is driven entirely by the enormous cost of misses.

Because of the large miss penalty and the expense of accessing the first byte, virtual pages tend to be large, typically four to eight KB. Due to the large miss penalty, DRAM caches are fully associative, that is, any virtual page can be placed in any physical page. The replacement policy on misses also assumes greater importance, because the penalty associated with replacing the wrong virtual page is so high. Thus, operating systems use much more sophisticated replacement algorithms for DRAM caches than the hardware does for SRAM caches. (These replacement algorithms are beyond our scope.) Finally, because of the large access time of disk, DRAM caches always use write-back instead of write-through.

10.3.2 Page Tables

As with any cache, the VM system must have some way to determine if a virtual page is cached somewhere in DRAM. If so, the system must determine which physical page it is cached in. If there is a miss, the system must determine where the virtual page is stored on disk, select a victim page in physical memory, and copy the virtual page from disk to DRAM, replacing the victim page.

These capabilities are provided by a combination of operating system software, address translation hardware in the MMU (memory management unit), and a data structure stored in physical memory known as a *page table* that maps virtual pages to physical pages. The address translation hardware reads the page table each time it converts a virtual address to a physical address. The operating system is responsible for maintaining the contents of the page table and transferring pages back and forth between disk and DRAM.

Figure 10.4 shows the basic organization of a page table. A page table is an array of *page table entries* (PTEs). Each page in the virtual address space has a PTE at a fixed offset in the page table. For our purposes, we will assume that each PTE consists of a *valid bit* and an *n-bit* address field. The valid bit

indicates whether the virtual page is currently cached in DRAM. If the valid bit is set, the address field indicates the start of the corresponding physical page in DRAM where the virtual page is cached. If the valid bit is not set, then a null address indicates that the virtual page has not yet been allocated. Otherwise, the address points to the start of the virtual page on disk.

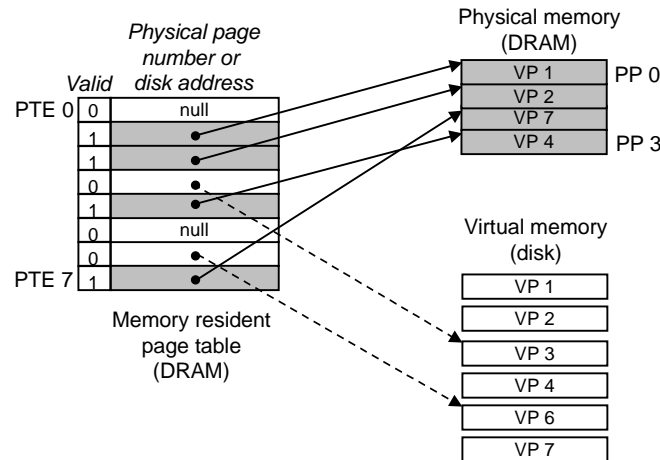


Figure 10.4: **Page table.**

The example in Figure 10.4 shows a page table for a system with 8 virtual pages and 4 physical pages. Four virtual pages (VP 1, VP 2, VP 4, and VP 7) are currently cached in DRAM. Two pages (VP 0 and VP 5) have not yet been allocated, and the rest (VP 3 and VP 6) have been allocated but are not currently cached. An important point to notice about Figure 10.4 is that because the DRAM cache is fully associative, any physical page can contain any virtual page.

Practice Problem 10.2:

Determine the number of page table entries (PTEs) that are needed for the following combinations of virtual address size (n) and page size (P):

n	$P = 2^p$	# PTEs
16	4K	
16	8K	
32	4K	
32	8K	

10.3.3 Page Hits

Consider what happens when the CPU reads a word of virtual memory contained in VP 2, which is cached in DRAM. (See Figure 10.5.) Using a technique we will describe in detail in Section 10.6, the address translation hardware uses the virtual address as an index to locate PTE 2 and read it from memory. Since the valid bit is set, the address translation hardware knows that VP 2 is cached in memory. So it uses the

physical memory address in the PTE (which points to the start of the cached page in PP 0) to construct the physical address of the word.

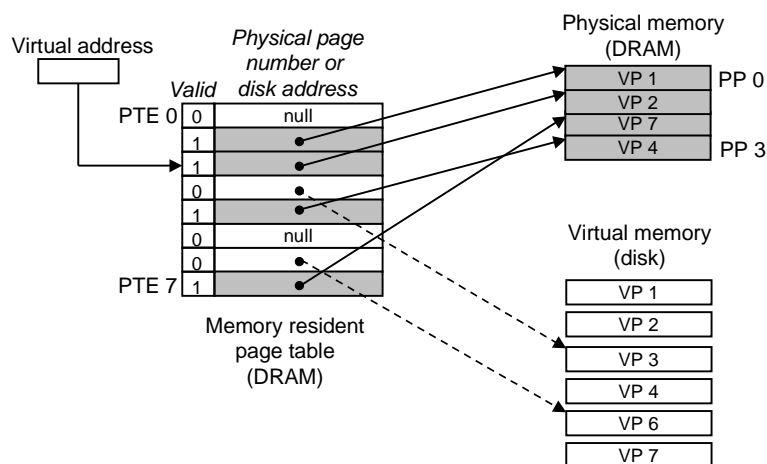


Figure 10.5: **VM page hit.** The reference to a word in VP 2 is a hit.

10.3.4 Page Faults

In virtual memory parlance, a DRAM cache miss is known as a *page fault*. Figure 10.6 shows the state of our example page table before the fault. The CPU has referenced a word in VP 3, which is not cached in DRAM. The address translation hardware reads PTE 3 from memory, infers from the valid bit that VP 3 is not cached, and triggers a page fault exception.

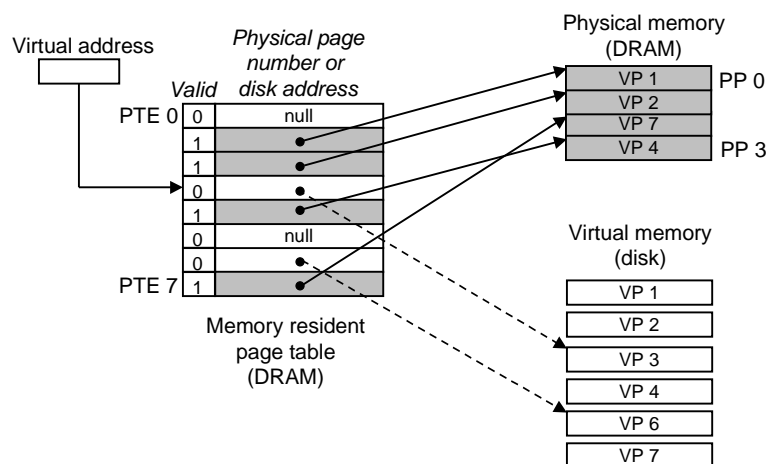


Figure 10.6: **VM page fault (before).** The reference to a word in VP 3 is a miss and triggers a page fault.

The page fault exception invokes a page fault exception handler in the kernel, which selects a victim page,

in this case VP 4 stored in PP 3. If VP 4 has been modified, then the kernel copies it back to disk. In either case, the kernel modifies the page table entry for VP 4 to reflect the fact that VP 4 is no longer cached in main memory.

Next, the kernel copies VP 3 from disk to PP 3 in memory, updates PTE 3, and then returns. When the handler returns, it restarts the faulting instruction, which resends the faulting virtual address to the address translation hardware. But now, VP 3 is cached in main memory, and the page hit is handled normally by the address translation hardware, as we saw in Figure 10.5. Figure 10.7 shows the state of our example page table after the page fault.

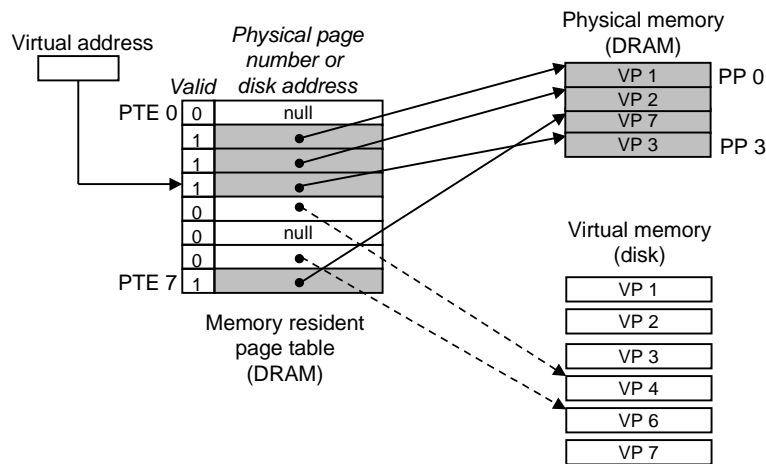


Figure 10.7: **VM page fault (after).** The page fault handler selects VP 4 as the victim and replaces it with a copy of VP 3 from disk. After the page fault handler restarts the faulting instruction, it will read the word from memory normally, without generating an exception.

Virtual memory was invented in the early 1960s, long before the widening CPU-memory gap spawned SRAM caches. As a result, virtual memory systems use a different terminology from SRAM caches, even though many of the ideas are similar. In virtual memory parlance, blocks are known as pages. The activity of transferring a page between disk and memory is known as *swapping* or *paging*. Pages are *swapped in* (*paged in*) from disk to DRAM, and *swapped out* (*paged out*) from DRAM to disk. The strategy of waiting until the last moment to swap in a page, when a miss occurs, is known as *demand paging*. Other approaches, such as trying to predict misses and swap pages in before they are actually referenced, are possible. However, all modern systems use demand paging.

10.3.5 Allocating Pages

Figure 10.8 shows the effect on our example page table when the operating system allocates a new page of virtual memory, for example, as a result of calling `malloc`. In the example, VP 5 is allocated by creating room on disk and updating PTE 5 to point to the newly created page on disk.

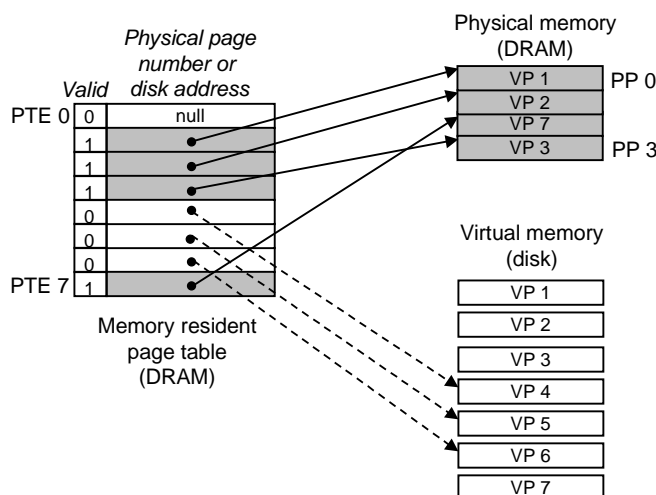


Figure 10.8: **Allocating a new virtual page.** The kernel allocates VP 5 on disk and points PTE 5 to this new location.

10.3.6 Locality to the Rescue Again

When many of us learn about the idea of virtual memory, our first impression is often that it must be terribly inefficient. Given the large miss penalties, we worry that paging will destroy program performance. In practice, virtual memory works pretty well, mainly because of our old friend *locality*.

Although the total number of distinct pages that programs reference during an entire run might exceed the total size of physical memory, the principle of locality promises that at any point in time they will tend to work on a smaller set of *active pages* known as the *working set* or *resident set*. After an initial overhead where the working set is paged into memory, subsequent references to the working set result in hits, with no additional disk traffic.

As long as our programs have good temporal locality, virtual memory systems work quite well. But of course, not all programs exhibit good temporal locality. If the working set size exceeds the size of physical memory, then the program can produce an unfortunate situation known as *thrashing*, where pages are swapped in and out continuously. Although virtual memory is usually efficient, if a program's performance slows to a crawl, the wise programmer will consider the possibility that it is thrashing.

Aside: Counting page faults.

You can monitor the number of page faults (and lots of other information) with the Unix `getrusage` function.

End Aside.

10.4 VM as a Tool for Memory Management

In the last section we saw how virtual memory provides a mechanism for using the DRAM to cache pages from a typically larger virtual address space. Interestingly, some early systems such as the DEC PDP-11/70 supported a virtual address space that was *smaller* than the physical memory. Yet virtual memory was still a

useful mechanism because it greatly simplified memory management and provided a natural way to protect memory.

To this point we have assumed a single page table that maps a single virtual address space to the physical address space. In fact, operating systems provide a separate page table, and thus a separate virtual address space, for each process. Figure 10.9 shows the basic idea. In the example, the page table for process i maps VP 1 to PP 2 and VP 2 to PP 7. Similarly, the page table for process j maps VP 1 to PP 7 and VP 2 to PP 10. Notice that multiple virtual pages can be mapped to the same shared physical page.

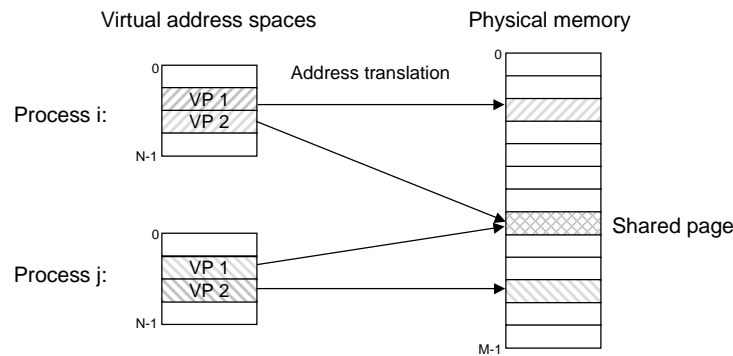


Figure 10.9: **How VM provides processes with separate address spaces.** The operating maintains a separate page table for each process in the system.

The combination of demand paging and separate virtual address spaces has a profound impact on the way that memory is used and managed in a system. In particular, VM simplifies linking and loading, the sharing of code and data, and allocating memory to applications.

10.4.1 Simplifying Linking

A separate address space allows each process to use the same basic format for its memory image, regardless of where the code and data actually reside in physical memory. For example, every Linux process uses the format shown in Figure 10.10.

The text section always starts at virtual address `0x08048000`, the stack always grows down from address `0xbfffffff`, shared library code always starts at address `0x40000000`, and the operating system code and data start always start at address `0xc0000000`. Such uniformity greatly simplifies the design and implementation of linkers, allowing them to produce fully linked executables that are independent of the ultimate location of the code and data in physical memory.

10.4.2 Simplifying Sharing

Separate address spaces provide the operating system with a consistent mechanism for managing sharing between user processes and the operating system itself. In general, each process has its own private code, data, heap, and stack areas that are not shared with any other process. In this case, the operating system creates page tables that map the corresponding virtual pages to disjoint physical pages.

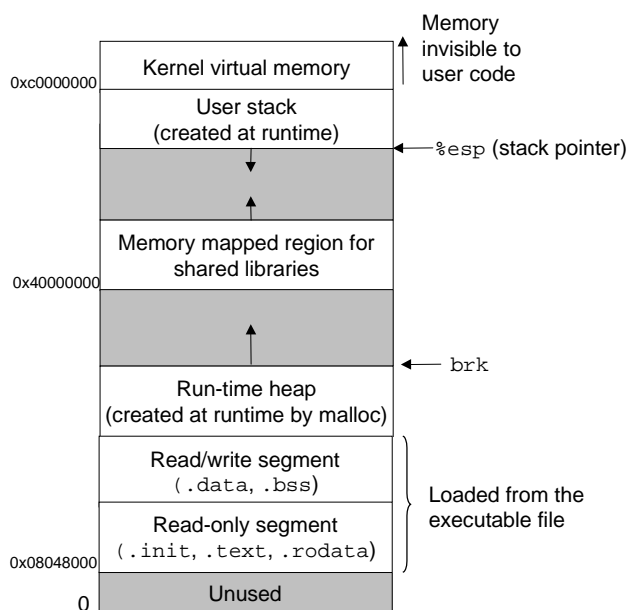


Figure 10.10: **The memory image of a Linux process.** Programs always start at virtual address `0x08048000`. The user stack always starts at virtual address `0xbfffffff`. Shared objects are always loaded in the region beginning at virtual address `0x40000000`.

However, in some instances it is desirable for processes to share code and data. For example, every process must call the same operating system kernel code, and every C program makes calls to routines in the standard C library such as `printf`. Rather than including separate copies of the kernel and standard C library in each process, the operating system can arrange for multiple processes to share a single copy of this code by mapping the appropriate virtual pages in different processes to the same physical pages.

10.4.3 Simplifying Memory Allocation

Virtual memory provides a simple mechanism for allocating additional memory to user processes. When a program running in a user process requests additional heap space (e.g., as a result of calling `malloc`), the operating system allocates an appropriate number, say k , of contiguous virtual memory pages, and maps them to k arbitrary physical pages located anywhere in physical memory. Because of the way page tables work, there is no need for the operating system to locate k contiguous pages of physical memory. The pages can be scattered randomly in physical memory.

10.4.4 Simplifying Loading

Virtual memory also makes it easy to load executable and shared object files into memory. Recall that the `.text` and `.data` sections in ELF executables are contiguous. To load these sections into a newly created process, the Linux loader allocates a contiguous chunk of virtual pages starting at address `0x08048000`, marks them as invalid (i.e., not cached), and points their page table entries to the appropriate locations in

the object file.

The interesting point is that the loader never actually copies any data from disk into memory. The data is paged in automatically and on demand by the virtual memory system the first time each page is referenced, either by the CPU when it fetches an instruction, or by an executing instruction when it references a memory location.

This notion of mapping a set of contiguous virtual pages to an arbitrary location in an arbitrary file is known as *memory mapping*. Unix provides a system call called `mmap` that allows application programs to do their own memory mapping. We will describe application-level memory mapping in more detail in Section 10.8.

10.5 VM as a Tool for Memory Protection

Any modern computer system must provide the means for the operating system to control access to the memory system. A user process should not be allowed to modify its read-only text section. Nor should it be allowed to read or modify any of the code and data structures in the kernel. It should not be allowed to read or write the private memory of other processes, and it should not be allowed to modify any virtual pages that are shared with other processes, unless all parties explicitly allow it (via calls to explicit interprocess communication system calls).

As we have seen, providing separate virtual address spaces makes it easy to isolate the private memories of different processes. But the address translation mechanism can be extended in a natural way to provide even finer access control. Since the address translation hardware reads a PTE each time the CPU generates an address, it is straightforward to control access to the contents of a virtual page by adding some additional permission bits to the PTE. Figure 10.11 shows the general idea.

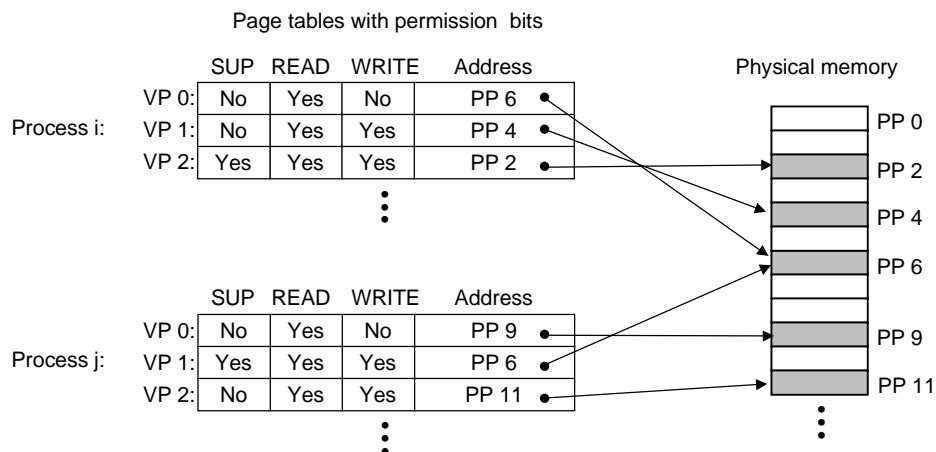


Figure 10.11: Using VM to provide page-level memory protection.

In this example, we have added three permission bits to each PTE. The SUP bit indicates whether processes must be running in kernel (supervisor) mode to access the page. Processes running in kernel mode can access any page, but processes running in user mode are only allowed to access pages for which SUP is 0.

Basic parameters	
Symbol	Description
$N = 2^n$	Number of addresses in virtual address space
$M = 2^m$	Number of addresses in physical address space
$P = 2^p$	Page size (bytes)

Components of a virtual address (VA)	
Symbol	Description
VPO	Virtual page offset (bytes)
VPN	Virtual page number
TLBI	TLB index
TLBT	TLB tag

Components of a physical address (PA)	
Symbol	Description
PPO	Physical page offset (bytes)
PPN	Physical page number
CO	Byte offset within cache block
CI	Cache index
CT	Cache tag

Figure 10.12: **Summary of address translation symbols.**

The READ and WRITE bits control read and write access to the page. For example, if process i is running in user mode, then it has permission to read VP 0 and to read or write VP 1. However, it is not allowed to access VP 2.

If an instruction violates these permissions, then the CPU triggers a general protection fault that transfers control to an exception handler in the kernel. Unix shells typically report this exception as a “segmentation fault.”

10.6 Address Translation

This section covers the basics of address translation. Our aim is to give you an appreciation of the hardware’s role in supporting virtual memory, with enough detail so that you can work through some concrete examples by hand. However, keep in mind that we are omitting a number of details, especially related to timing, that are important to hardware designers, but are beyond our scope. For your reference, Figure 10.12 summarizes the symbols that we will use throughout this section.

Formally, address translation is a mapping between the elements of an N -element virtual address space (VAS) and an M -element physical address space (PAS),

$$\text{MAP: VAS} \rightarrow \text{PAS} \cup \emptyset$$

where

$$\text{MAP}(A) = A' \text{ if data at virtual addr } A \text{ is present at physical addr } A' \text{ in PAS.}$$

$= \emptyset$ if data at virtual addr A is not present in physical memory.

Figure 10.13 shows how the MMU uses the page table to perform this mapping. A control register in the CPU, the *page table base register (PTBR)* points to the current page table. The n -bit virtual address has two components: a p -bit *virtual page offset (VPO)* and an $(n - p)$ -bit *virtual page number (VPN)*. The MMU uses the VPN to select the appropriate PTE. For example, VPN 0 selects PTE 0, VPN 1 selects PTE 1, and so on. The corresponding physical address is the concatenation of the *physical page number (PPN)* from the page table entry and the VPO from the virtual address. Notice that since the physical and virtual pages are both P bytes, the *physical page offset (PPO)* is identical to the VPO.

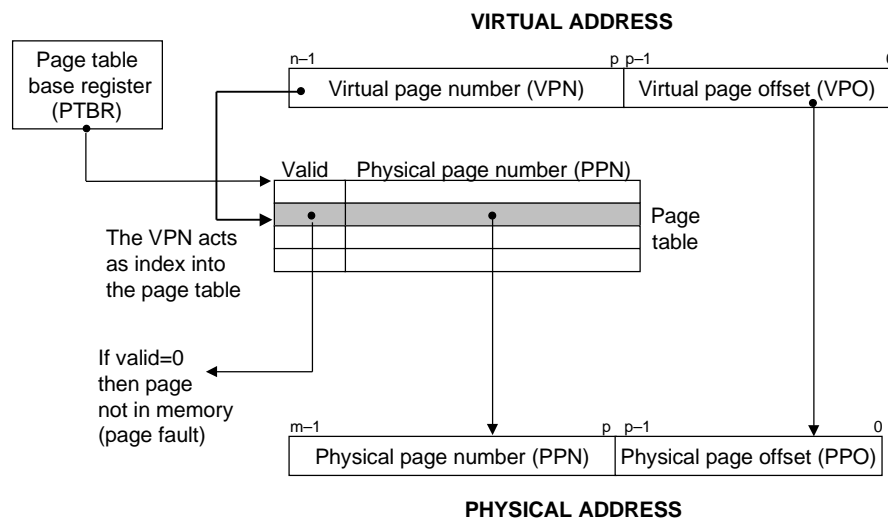


Figure 10.13: **Address translation with a page table.**

Figure 10.14(a) shows the steps that the CPU hardware performs when there is a page hit.

- *Step 1:* The processor generates a virtual address and sends it to the MMU.
- *Step 2:* The MMU generates the PTE address and requests it from the cache/main memory.
- *Step 3:* The cache/main memory returns the PTE to the MMU.
- *Step 3:* The MMU constructs the physical address and sends it to cache/main memory.
- *Step 4:* The cache/main memory returns the requested data word to the processor.

Unlike a page hit, which is handled entirely by hardware, handling a page fault requires cooperation between hardware and the operating system kernel (Figure 10.14(b)).

- *Steps 1 to 3:* The same as Steps 1 to 3 in Figure 10.14(a).
- *Step 4:* The valid bit in the PTE is zero, so the MMU triggers an exception, which transfers control in the CPU to a page fault exception handler in the operating system kernel.

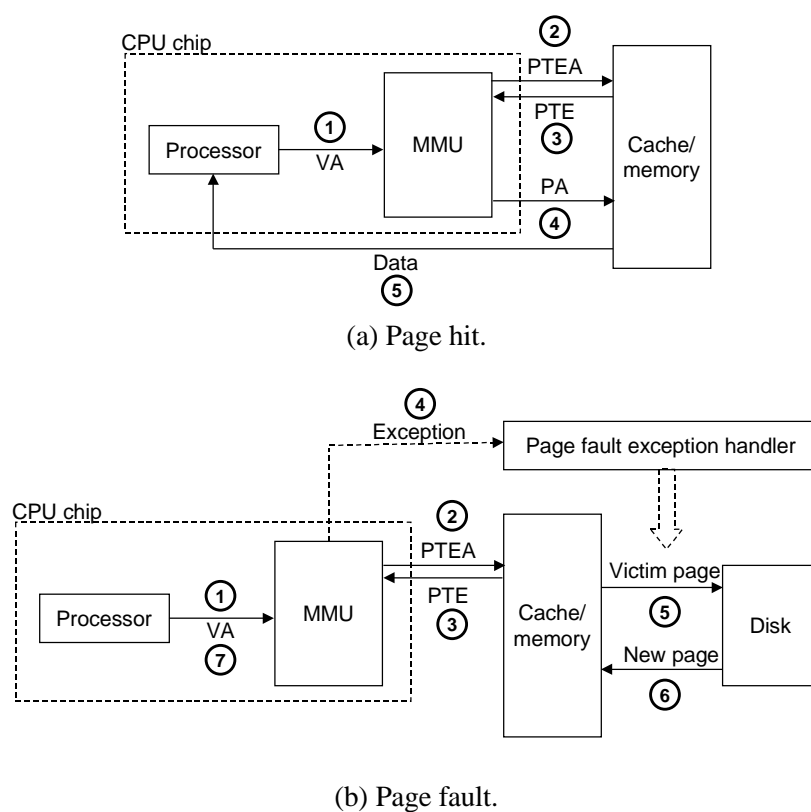


Figure 10.14: **Operational view of page hits and page faults.** VA: virtual address. PTEA: page table entry address. PTE: page table entry. PA: physical address.

- *Step 5:* The fault handler identifies a victim page in physical memory, and if that page has been modified, pages it out to disk.
- *Step 6:* The fault handler pages in the new page and updates the PTE in memory.
- *Step 7:* The fault handler returns to the original process, causing the faulting instruction to be restarted. The CPU resends the offending virtual address to the MMU. Because the virtual page is now cached in physical memory, there is a hit, and after the MMU performs the steps in Figure 10.14(b), the main memory returns the requested word to the processor

Practice Problem 10.3:

Given a 32-bit virtual address space and a 24-bit physical address, determine the number of bits in the VPN, VPO, PPN, and PPO for the following page sizes P :

P	# VPN bits	# VPO bits	# PPN bits	# PPO bits
1 KB				
2 KB				
4 KB				
8 KB				

10.6.1 Integrating Caches and VM

In any system that uses both virtual memory and SRAM caches, there is the issue of whether to use virtual or physical addresses to access the cache. Although a detailed discussion of the trade-offs is beyond our scope, most systems opt for physical addressing. With physical addressing it is straightforward for multiple processes to have blocks in the cache at the same time and to share blocks from the same virtual pages. Further, the cache does not have to deal with protection issues because access rights are checked as part of the address translation process.

Figure 10.15 shows how a physically addressed cache might be integrated with virtual memory. The main idea is that the address translation occurs before the cache lookup. Notice that page table entries can be cached, just like any other data words.

10.6.2 Speeding up Address Translation with a TLB

As we have seen, every time the CPU generates a virtual address, the MMU must refer to a PTE in order to translate the virtual address into a physical address. In the worst case, this requires an additional fetch from memory, at a cost of tens to hundreds of cycles. If the PTE happens to be cached in L1, then the cost goes down to one or two cycles. However, many systems try to eliminate even this cost by including a small cache of PTEs in the MMU called a *translation lookaside buffer (TLB)*.

A TLB is a small, virtually addressed cache where each line holds a block consisting of a single PTE. A TLB usually has a high degree of associativity. As shown in Figure 10.16, the index and tag fields that are used for set selection and line matching are extracted from the virtual page number in the virtual address. If

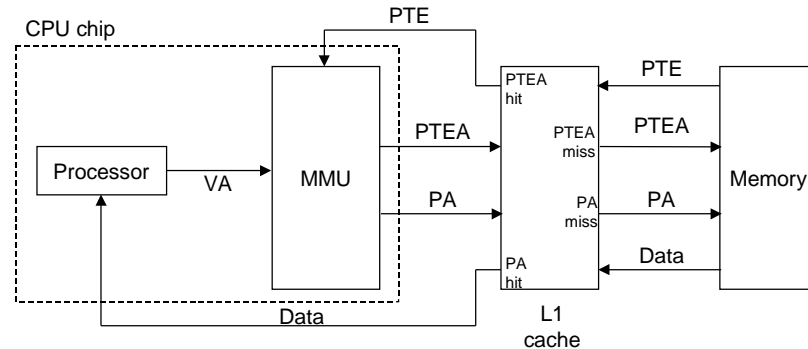


Figure 10.15: **Integrating VM with a physically addressed cache.** VA: virtual address. PTEA: page table entry address. PTE: page table entry. PA: physical address.

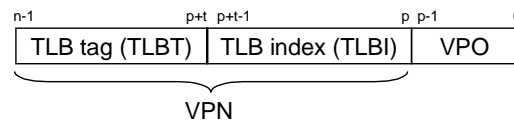


Figure 10.16: **Components of a virtual address that are used to access the TLB.**

the TLB has $T = 2^t$ sets, then the *TLB index (TLBI)* consists of the t least significant bits of the VPN, and the *TLB tag (TLBT)* consists of the remaining bits in the VPN.

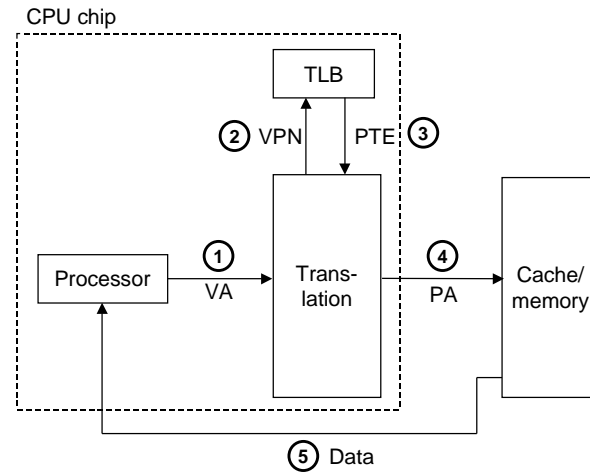
Figure 10.17(a) shows the steps involved when there is a TLB hit (the usual case). The key point here is that all of the address translation steps are performed inside the on-chip MMU, and thus are fast.

- *Step 1:* The CPU generates a virtual address.
- *Steps 2 and 3:* The MMU fetches the appropriate PTE from the TLB.
- *Step 4:* The MMU translates the virtual address to a physical address and sends it to the cache/main memory.
- *Step 5:* The cache/main memory returns the requested data word to the CPU.

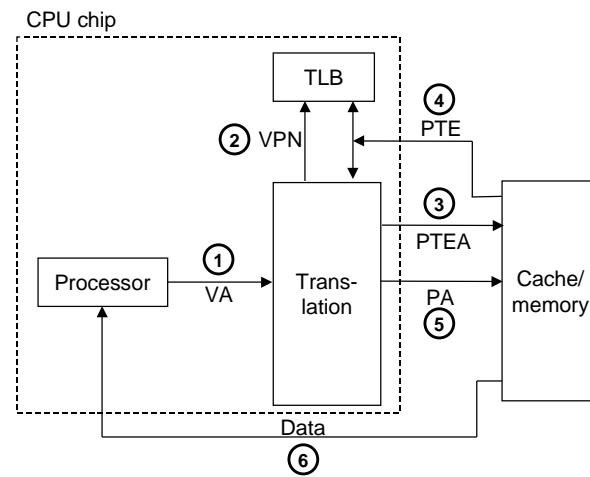
When there is a TLB miss, then the MMU must fetch the PTE from the L1 cache, as shown in Figure 10.17(b). The newly fetched PTE is stored in the TLB, possibly overwriting an existing entry.

10.6.3 Multi Level Page Tables

To this point we have assumed that the system uses a single page table to do address translation. But if we had a 32-bit address space, 4-KB pages, and a 4-byte PTE, then we would need a 4-MB page table resident in memory at all times, even if the application referenced only a small chunk of the virtual address space. The problem is compounded for systems with 64-bit addresses spaces.



(a) TLB hit.



(b) TLB miss.

Figure 10.17: **Operational view of a TLB hit and miss.**

The common approach for compacting the page table is to use a hierarchy of page tables instead. The idea is easiest to understand with a concrete example. Suppose the 32-bit virtual address space is partitioned into four-KB pages, and that page table entries are four bytes each. Suppose also that at this point in time the virtual address space has the following form: The first 2K pages of memory are allocated for code and data, the next 6K pages are unallocated, the next 1023 pages are also unallocated, and the next page is allocated for the user stack. Figure 10.18 shows how we might construct a two-level page table hierarchy for this virtual address space.

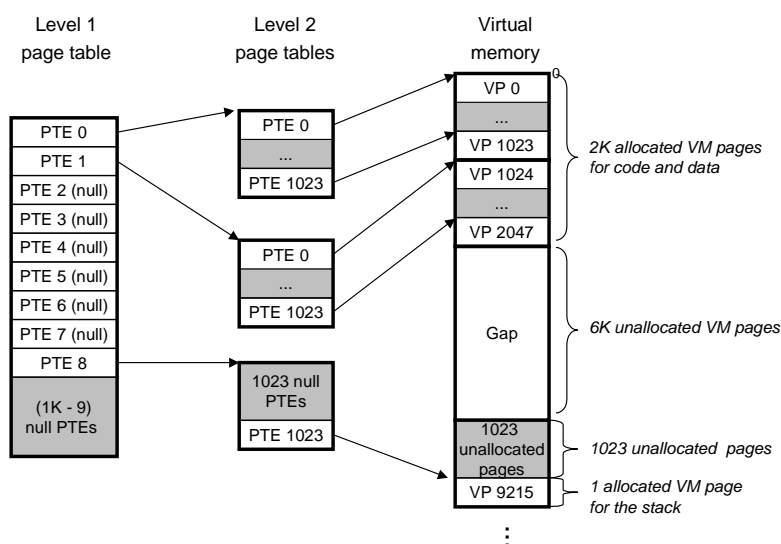


Figure 10.18: **A two-level page table hierarchy.** Notice that addresses increase from top to bottom.

Each PTE in the level-1 table is responsible for mapping a four-MB chunk of the virtual address space, where each chunk consists of 1024 contiguous pages. For example, PTE 0 maps the first chunk, PTE 1 the next chunk, and so on. Given that the address space is four GB, 1024 PTEs are sufficient to cover the entire space.

If every page in chunk i is unallocated, then level-1 PTE i is null. For example, in Figure 10.18, chunks 2–7 are unallocated. However, if at least one page in chunk i is allocated, then level-1 PTE i points to the base of a level-2 page table. For example, in Figure 10.18, all or portions of chunks 0, 1, and 8 are allocated, so their level-1 PTEs point to level-2 page tables.

Each PTE in a level-2 page table is responsible for mapping a 4-KB page of virtual memory, just as before when we looked at single-level page tables. Notice that with 4-byte PTEs, each level-1 and level-2 page table is 4K bytes, which conveniently is the same size as a page.

This scheme reduces memory requirements in two ways. First, if a PTE in the level-1 table is null, then the corresponding level-2 page table does not even have to exist. This represents a significant potential savings, since most of the 4-GB virtual address space for a typical program is unallocated. Second, only the level-1 table needs to be in main memory at all times. The level-2 page tables can be created and paged in and out by the VM system as they are needed, which reduces pressure on main memory. Only the most heavily used level-2 page tables need to be cached in main memory.

Figure 10.19 summarizes address translation with a k -level page table hierarchy. The virtual address is partitioned into k VPNs and a VPO. Each VPN i , $1 \leq i \leq k$, is an index into a page table at level i . Each PTE in a level- j table, $1 \leq j \leq k - 1$, points to the base of some page table at level $j + 1$. Each PTE in a level- k table contains either the PPN of some physical page or the address of a disk block. To construct the physical address, the MMU must access k PTEs before it can determine the PPN. As with a single-level hierarchy, the PPO is identical to the VPO.

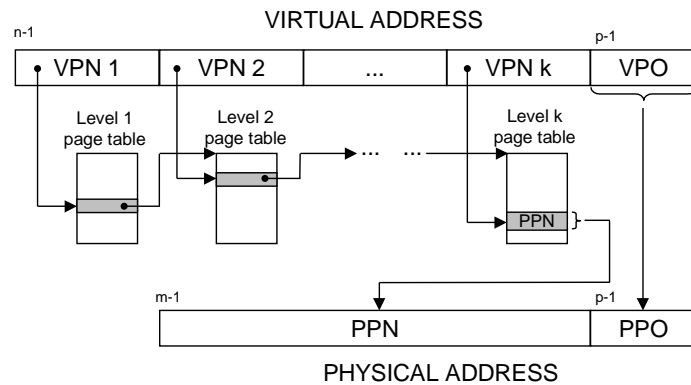


Figure 10.19: Address translation with a k -level page table.

Accessing k PTEs may seem expensive and impractical at first glance. However, the TLB comes to the rescue here by caching PTEs from the page tables at the different levels. In practice, address translation with multi level page tables is not significantly slower than with single-level page tables.

10.6.4 Putting it Together: End-to-end Address Translation

In this section we put it all together with a concrete example of end-to-end address translation on a small system with a TLB and L1 d-cache. To keep things manageable, we make the following assumptions:

- The memory is byte addressable.
- Memory accesses are to **1-byte words** (not 4-byte words).
- Virtual addresses are 14 bits wide ($n = 14$).
- Physical addresses are 12 bits wide ($m = 12$).
- The page size is 64 bytes ($P = 64$).
- The TLB is four-way set associative with 16 total entries.
- The L1 d-cache is physically addressed and direct mapped, with a 4-byte line size and 16 total sets.

Figure 10.20 shows the formats of the virtual and physical addresses. Since each page is $2^6 = 64$ bytes, the low-order six bits of the virtual and physical addresses serve as the VPO and PPO respectively. The

high-order eight bits of the virtual address serve as the VPN. The high-order six bits of the physical address serve as the PPN.

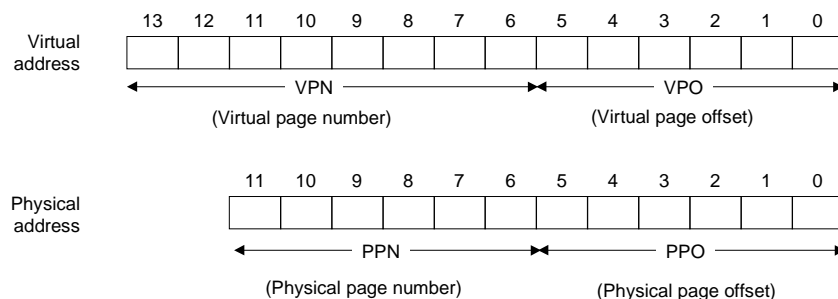


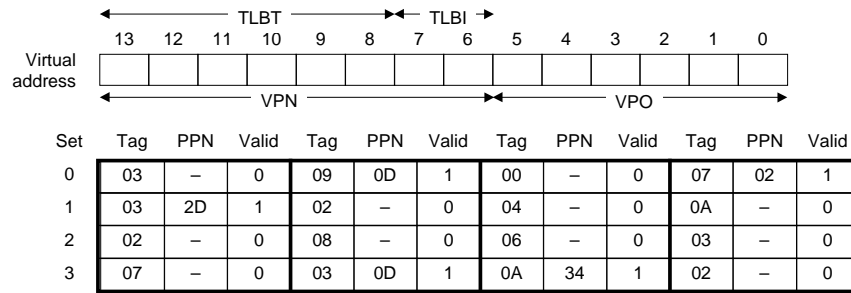
Figure 10.20: **Addressing for small memory system.** Assume 14-bit virtual addresses ($n = 14$), 12-bit physical addresses ($m = 12$), and 64-byte pages ($P = 64$).

Figure 10.21 shows a snapshot of our little memory system, including the TLB (a), a portion of the page table (b), and the L1 cache (c). Above the figures of the TLB and cache, we have also shown how the bits of the virtual and physical addresses are partitioned by the hardware it accesses these devices.

- **TLB:** The TLB is virtually addressed using the bits of the VPN. Since the TLB has four sets, the two low-order bits of the VPN serve as the set index (TLBI). The remaining six high-order bits serve as the tag (TLBT) that distinguishes the different VPNs that might map to the same TLB set.
- **Page table.** The page table is a single-level design with a total of $2^8 = 256$ page table entries (PTEs). However, we are only interested in the first sixteen of these. For convenience, we have labelled each PTE with the VPN that indexes it; but keep in mind though that these VPNs are not part of the page table and not stored in memory. Also, notice that the PPN of each invalid PTE is denoted with a dash to reinforce the idea that whatever bit values might happen to be stored there are not meaningful.
- **Cache.** The direct-mapped cache is addressed by the fields in the physical address. Since each block is 4 bytes, the low-order 2 bits of the physical address serve as the block offset (CO). Since there are 16 sets, the next 4 bits serve as the set index (CI). The remaining 6 bits serve as the tag (CT).

Given this initial setup, let's see what happens when the CPU executes a load instruction that reads the byte at address $0x03d4$. (Recall that our hypothetical CPU reads one-byte words rather than four-byte words.) To begin this kind of manual simulation, we find it helpful to write down the bits in the virtual address, identify the various fields we will need, and determine their hex values. The hardware performs a similar task when it decodes the address.

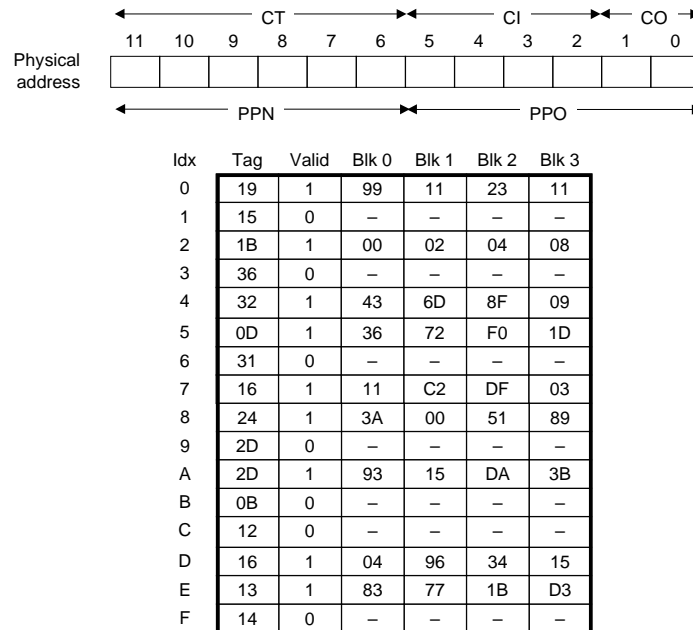
	TLBT						TLBI							
	0x03						0x03							
bit position	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VA = 0x03d4	0	0	0	0	1	1	1	1	0	1	0	1	0	0
	VPN								VPO					
	0x0f								0x14					



(a) TLB: Four sets, sixteen entries, four-way set associative.

VPN	PPN	Valid	VPN	PPN	Valid
00	28	1	08	13	1
01	—	0	09	17	1
02	33	1	0A	09	1
03	02	1	0B	—	0
04	—	0	0C	—	0
05	16	1	0D	2D	1
06	—	0	0E	11	1
07	—	0	0F	0D	1

(b) Page table: Only the first sixteen PTEs are shown.



(c) Cache: 16 sets, four-byte blocks, direct mapped.

Figure 10.21: **TLB, page table, and cache for small memory system.** All values in the TLB, page table, and cache are in hexadecimal notation.

To begin, the MMU extracts the VPN (0x0F) from the virtual address and checks with the TLB to see if has cached a copy of PTE 0x0F from some previous memory reference. The TLB extracts the TLB index (0x03) and the TLB tag (0x3) from the VPN, hits on a valid match in the second entry of Set 0x3, and returns the cached PPN (0x0D) to the MMU.

If the TLB had missed, then the MMU would need to fetch the PTE from main memory. However, in this case we got lucky and had a TLB hit. The MMU now has everything it needs to form the physical address. It does this by concatenating the PPN (0x0D) from the PTE with the VPO (0x14) from the virtual address, which forms the physical address (0x354).

Next, the MMU sends the physical address to the cache, which extracts the cache offset CO (0x0), the cache set index CI (0x5), and the cache tag CT (0x0D) from the physical address.

	CT						CI				CO	
	0x0d						0x05				0x0	
bit position	11	10	9	8	7	6	5	4	3	2	1	0
PA = 0x354	0	0	1	1	0	1	0	1	0	1	0	0
	PPN						PPO					
	0x0d						0x14					

Since the tag in Set 0x5 matches CT, the cache detects a hit, reads out the data byte (0x36) at offset CO, and returns it to the MMU, which then passes it back to the CPU.

Other paths through the translation process are also possible. For example, if the TLB misses, then the MMU must fetch the PPN from a PTE in the page table. If the resulting PTE is invalid, then there is a page fault and the kernel must page in the appropriate page and rerun the load instruction. Another possibility is that the PTE is valid, but the necessary memory block misses in the cache.

Practice Problem 10.4:

Show how the example memory system in Section 10.6.4 translates a virtual address into a physical address and accesses the cache. For the given virtual address, indicate the TLB entry accessed, physical address, and cache byte value returned. Indicate whether the TLB misses, whether a page fault occurs, and whether a cache miss occurs. If there is a cache miss, enter “—” for “Cache byte returned”. If there is a page fault, enter “—” for “PPN” and leave parts C and D blank.

Virtual address: 0x03d7

A. Virtual address format

13	12	11	10	9	8	7	6	5	4	3	2	1	0

B. Address translation

Parameter	Value
VPN	
TLB index	
TLB tag	
TLB hit? (Y/N)	
Page fault? (Y/N)	
PPN	

C. Physical address format

11	10	9	8	7	6	5	4	3	2	1	0

D. Physical memory reference

Parameter	Value
Byte offset	
Cache index	
Cache tag	
Cache hit? (Y/N)	
Cache byte returned	

10.7 Case Study: The Pentium/Linux Memory System

We conclude our discussion of caches and virtual memory with a case study of a real system: a Pentium-class system running Linux. Figure 10.22 gives the highlights of the Pentium memory system. The Pentium has a 32-bit (4 GB) address space. The *processor package* includes the CPU chip, a unified L2 cache, and a cache bus (backside bus) that connects them. The CPU chip proper contains four different caches: an instruction TLB, data TLB, L1 i-cache, and L1 d-cache. The TLBs are virtually addressed. The L1 and L2 caches are physically addressed. All caches in the Pentium (including the TLBs) are four-way set associative.

The TLBs cache 32-bit page table entries. The instruction TLB caches PTEs for the virtual addresses generated by the instruction fetch unit. The data TLB caches PTEs for the virtual addresses of data. The instruction TLB has 32 entries. The data TLB has 64 entries. The page size can be configured at start-up time as either 4 KB or 4 MB. Linux running on a Pentium uses 4-KB pages.

The L1 and L2 caches have 32-byte blocks. Each L1 caches is 16 KB in size and has 128 sets, each of which contains four lines. The L2 cache size can vary from a minimum of 128 KB to a maximum of 2 MB. A typical size is 512 KB.

10.7.1 Pentium Address Translation

This section discusses the address translation process on the Pentium. For your reference, Figure 10.23 summarizes the entire process, from the time the CPU generates a virtual address until a data word arrives from memory.