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 9.1 **(solution page 916)**

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).

Number of virtual address bits (n)	Number of virtual addresses (N)	Largest possible virtual address
4		
	$2^? = 16$ K	
		$2^{24} - 1 = ?$ M $- 1$
	$2^? = 64$ T	
54		

9.3 VM as a Tool for Caching

Conceptually, a virtual memory is organized as an array of N contiguous byte-size 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-size 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*.)

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 9.3 shows a small virtual memory with eight virtual pages. Virtual pages 0 and 3 have not been allocated yet, and thus do not yet exist

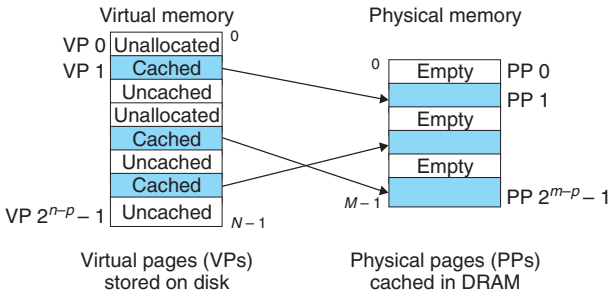

Figure 9.3
How a VM system uses main memory as a cache.

on disk. Virtual pages 1, 4, and 6 are cached in physical memory. Pages 2, 5, and 7 are allocated but are not currently cached in physical memory.

9.3.1 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, L2, and L3 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 at least 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 4 KB to 2 MB. 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 here.) Finally, because of the large access time of disk, DRAM caches always use write-back instead of write-through.

9.3.2 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

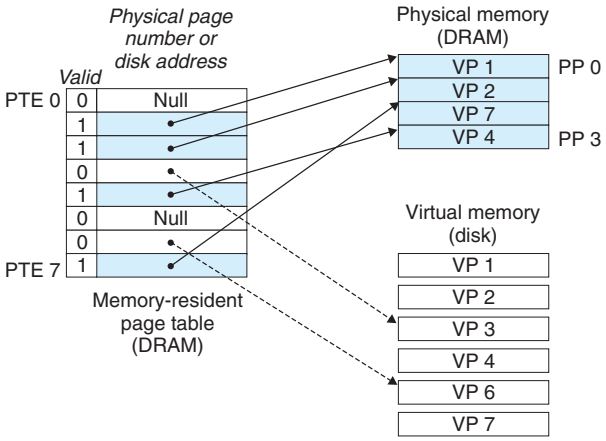

Figure 9.4
Page table.

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 9.4 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.

The example in Figure 9.4 shows a page table for a system with eight virtual pages and four 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 9.4 is that because the DRAM cache is fully associative, any physical page can contain any virtual page.

Practice Problem 9.2 (solution page 917)

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	$P = 2^p$	Number of PTEs
12	1 K	_____
16	16 K	_____
24	2 M	_____
36	1 G	_____

9.3.3 Page Hits

Consider what happens when the CPU reads a word of virtual memory contained in VP 2, which is cached in DRAM (Figure 9.5). Using a technique we will describe in detail in Section 9.6, 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 1) to construct the physical address of the word.

9.3.4 Page Faults

In virtual memory parlance, a DRAM cache miss is known as a *page fault*. Figure 9.6 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. 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.

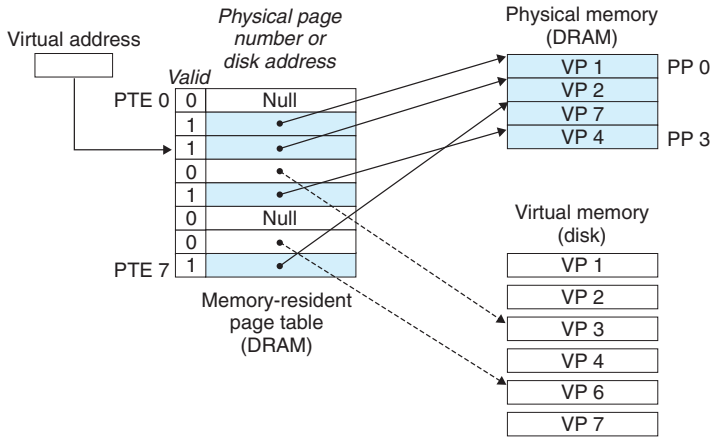

Figure 9.5
VM page hit. The reference to a word in VP 2 is a hit.

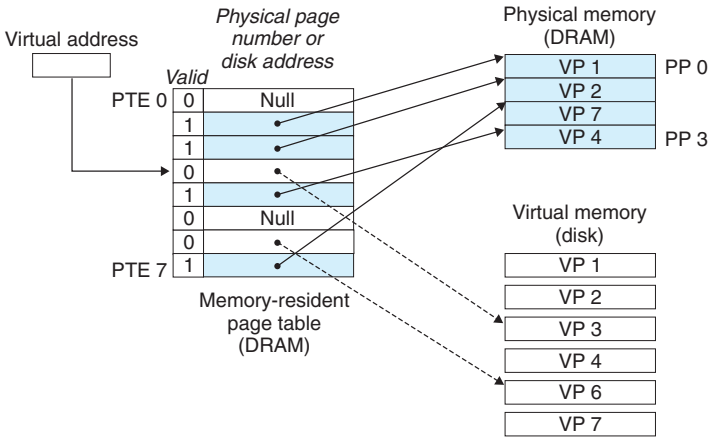

Figure 9.6
VM page fault (before). The reference to a word in VP 3 is a miss and triggers a page fault.

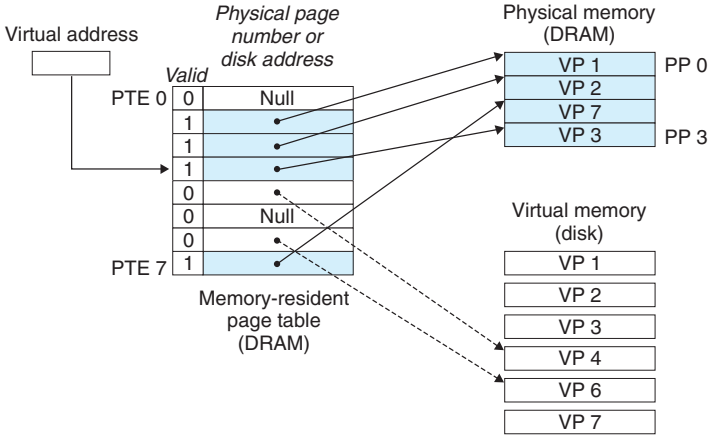

Figure 9.7
VM page fault (after). The page fault handler selects VP 4 as the victim and replaces it with a copy of VP 3 from disk. After the page fault handler restarts the faulting instruction, it will read the word from memory normally, without generating an exception.

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. Figure 9.7 shows the state of our example page table after the page fault.

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

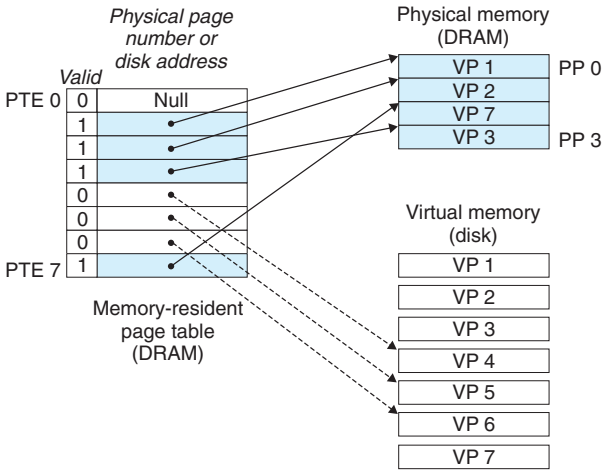

Figure 9.8
Allocating a new virtual page. The kernel allocates VP 5 on disk and points PTE 5 to this new location.

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.

9.3.5 Allocating Pages

Figure 9.8 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.

9.3.6 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 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 Linux `getrusage` function.

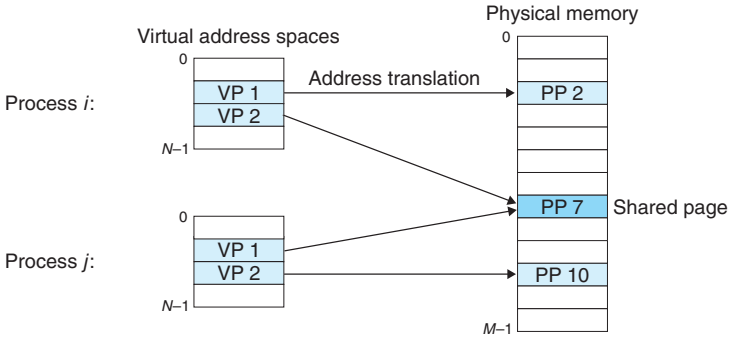

Figure 9.9
How VM provides processes with separate address spaces. The operating system maintains a separate page table for each process in the system.

9.4 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 available physical memory. Yet virtual memory was still a useful mechanism because it greatly simplified memory management and provided a natural way to protect memory.

Thus far, 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 9.9 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.

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.

- *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, as we saw in Figure 8.13, every process on a given Linux system has a similar memory format. For 64-bit address spaces, the code segment *always* starts at virtual address `0x400000`. The data segment follows the code segment after a suitable alignment gap. The stack occupies the highest portion of the user process address space and