9 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.

9.1 Physical and Virtual Addressing

The main memory of a computer system is organized as an array of *M* contiguous byte-size 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 9.1 shows an example of physical addressing in the context of a load instruction that reads the 4-byte word starting at physical address 4. 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 4-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 use a form of addressing known as *virtual addressing*, as shown in Figure 9.2.

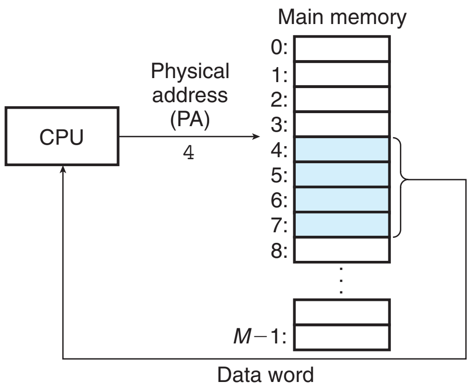


Figure 9.1
A system that uses physical addressing.

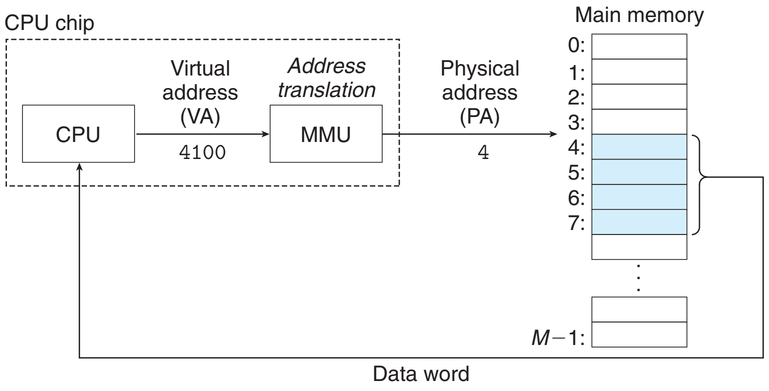


Figure 9.2
A system that uses virtual addressing.

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 main 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 lookup table stored in main memory whose contents are managed by the operating system.

9.2 Address Spaces

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

$$\{0, 1, 2, \ldots\}$$

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*:

$$\{0, 1, 2, \ldots, N-1\}$$

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:

$$\{0, 1, 2, \ldots, M-1\}$$

M is not required to be a power of 2, 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 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: $\text{K} = 2^{10}$ (kilo), $\text{M} = 2^{20}$ (mega), $\text{G} = 2^{30}$ (giga), $\text{T} = 2^{40}$ (tera), $\text{P} = 2^{50}$ (peta), or $\text{E} = 2^{60}$ (exa).

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

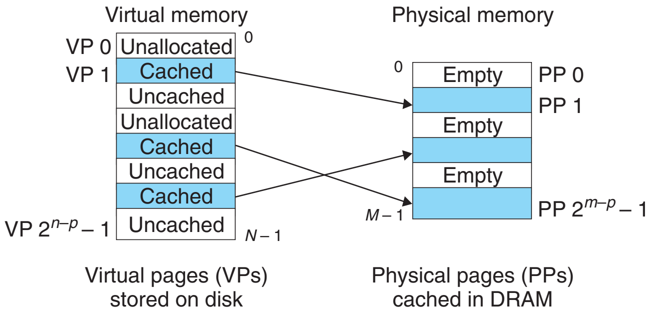


Figure 9.3
How a VM system uses main memory as a cache.

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

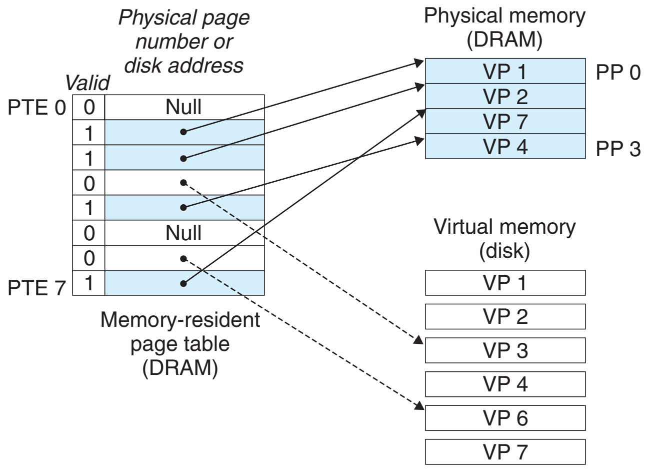


Figure 9.4
Page table.

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.

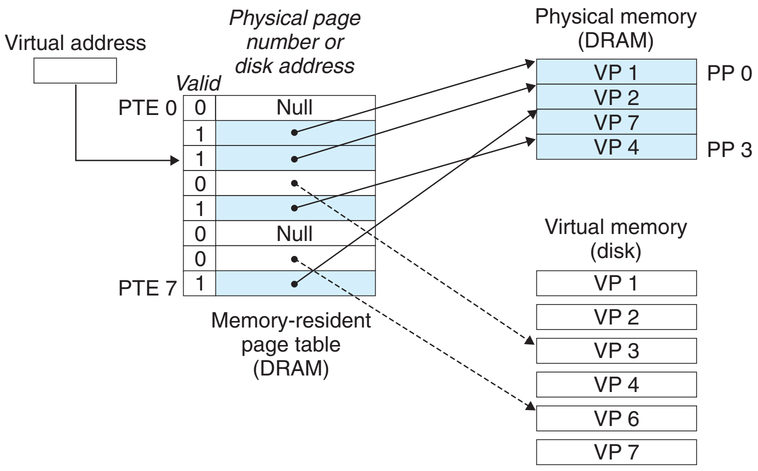


Figure 9.5
VM page hit. The reference to a word in VP 2 is a hit.

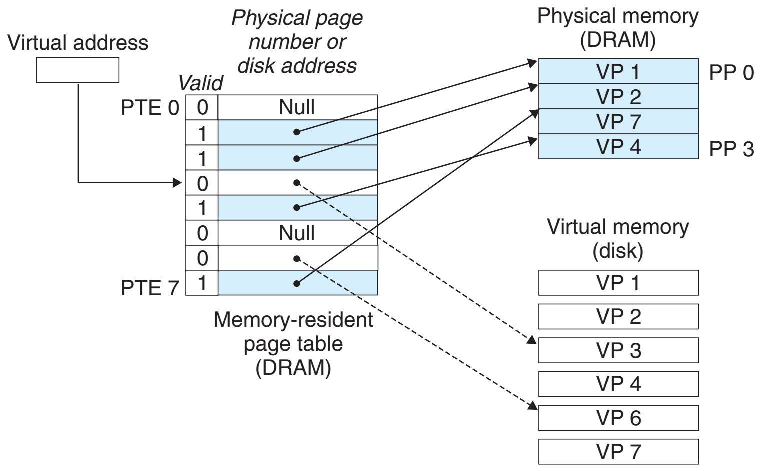


Figure 9.6
VM page fault (before). The reference to a word in VP 3 is a miss and triggers a page fault.

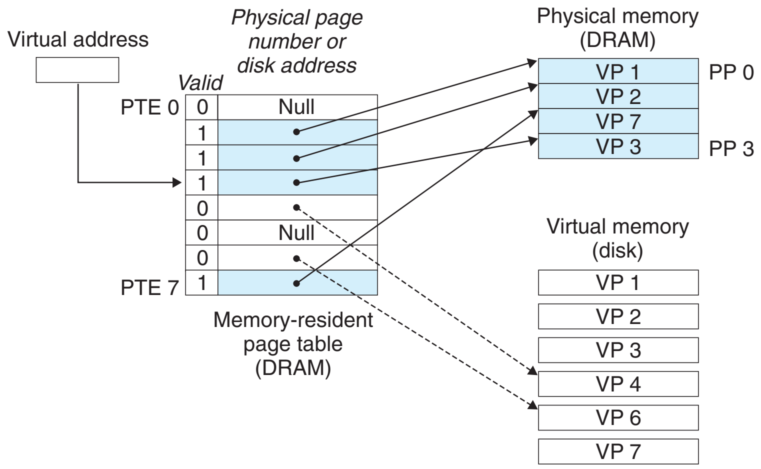


Figure 9.7
VM page fault (after). The page fault handler selects VP 4 as the victim and replaces it with a copy of VP 3 from disk. After the page fault handler restarts the faulting instruction, it will read the word from memory normally, without generating an exception.

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

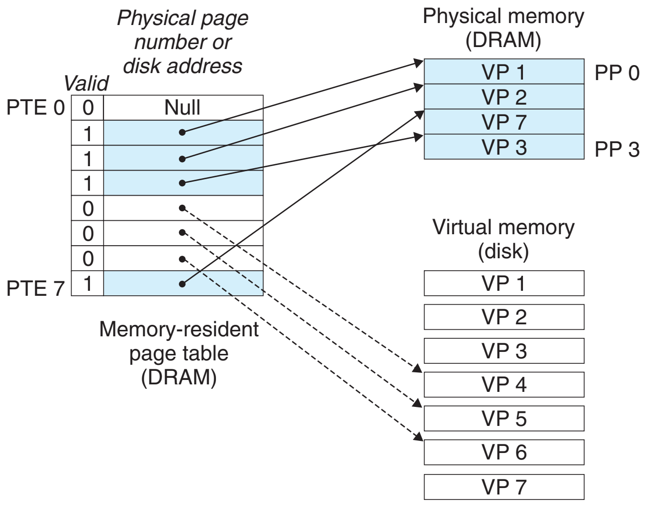


Figure 9.8
Allocating a new virtual page. The kernel allocates VP 5 on disk and points PTE 5 to this new location.

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.

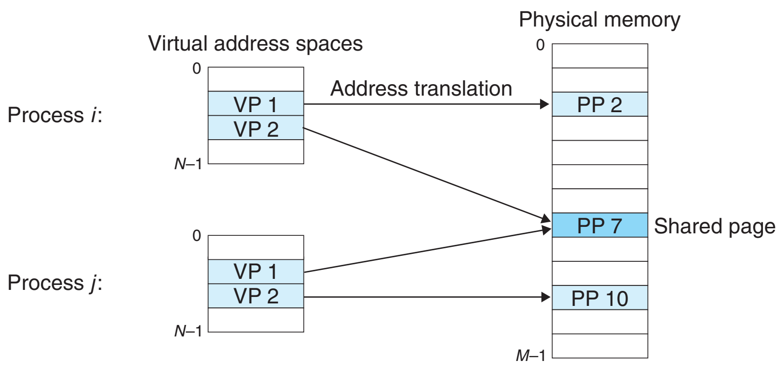


Figure 9.9
How VM provides processes with separate address spaces. The operating system maintains a separate page table for each process in the system.

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 grows downward. 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.

- *Simplifying loading.* Virtual memory also makes it easy to load executable and shared object files into memory. To load the `.text` and `.data` sections of an object file into a newly created process, the Linux loader allocates virtual pages for the code and data segments, 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 are 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*. Linux 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 9.8.

- *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.

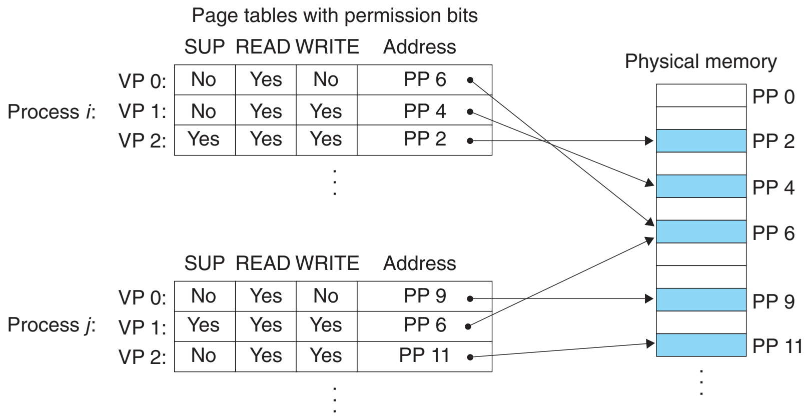
  However, in some instances it is desirable for processes to share code and data. For example, every process must call the same operating system kernel code, and every C program makes calls to routines in the standard C library such as `printf`. Rather than including separate copies of the kernel and 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, as we saw in Figure 9.9.

- *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.

9.5 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

Figure 9.10
Using VM to provide page-level memory protection.



Process *i*:

	SUP	READ	WRITE	Address
VP 0:	No	Yes	No	PP 6
VP 1:	No	Yes	Yes	PP 4
VP 2:	Yes	Yes	Yes	PP 2

Process *j*:

	SUP	READ	WRITE	Address
VP 0:	No	Yes	No	PP 9
VP 1:	Yes	Yes	Yes	PP 6
VP 2:	No	Yes	Yes	PP 11

to modify its read-only code 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 9.10 shows the general idea.

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. 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, which sends a SIGSEGV signal to the offending process. Linux shells typically report this exception as a “segmentation fault.”

9.6 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,

Symbol	Description
Basic parameters	
$N = 2^n$	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)	
VPO	Virtual page offset (bytes)
VPN	Virtual page number
TLBI	TLB index
TLBT	TLB tag
Components of a physical address (PA)	
PPO	Physical page offset (bytes)
PPN	Physical page number
CO	Byte offset within cache block
CI	Cache index
CT	Cache tag

Figure 9.11 **Summary of address translation symbols.**

that are important to hardware designers but are beyond our scope. For your reference, Figure 9.11 summarizes the symbols that we will be using throughout this section.

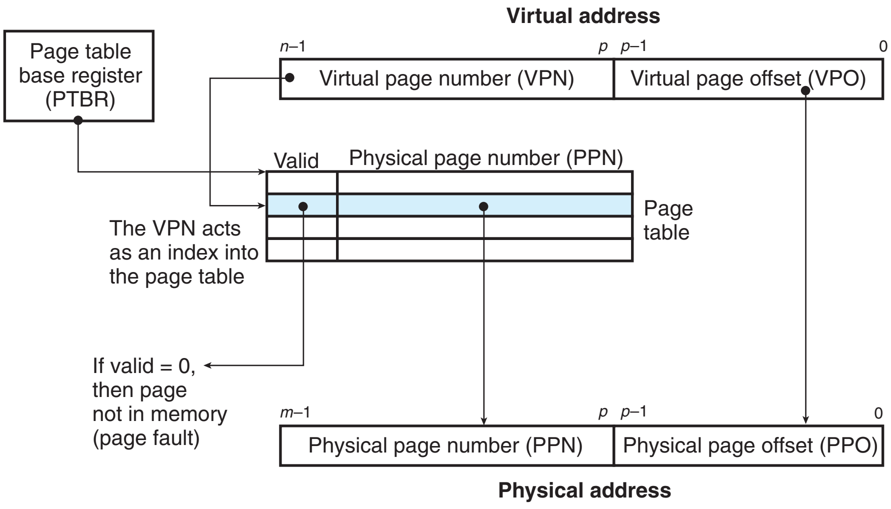
Formally, address translation is a mapping between the elements of an N-element virtual address space (VAS) and an M-element physical address space (PAS),

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

where

$$\text{MAP}(A) = \begin{cases} A' & \text{if data at virtual addr. } A \text{ are present at physical addr. } A' \text{ in PAS} \\ \emptyset & \text{if data at virtual addr. } A \text{ are not present in physical memory} \end{cases}$$

Figure 9.12 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.



Virtual address
n–1
p p–1
0
Page table base register (PTBR)
Virtual page number (VPN)
Virtual page offset (VPO)
Valid
Physical page number (PPN)
Page table
The VPN acts as an index into the page table
If valid = 0, then page not in memory (page fault)
m–1
p p–1
0
Physical page number (PPN)
Physical page offset (PPO)
Physical address

Figure 9.12 **Address translation with a page table.**

Figure 9.13(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 4. The MMU constructs the physical address and sends it to the cache/main memory.

Step 5. 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 9.13(b)).

Steps 1 to 3. The same as steps 1 to 3 in Figure 9.13(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.

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.

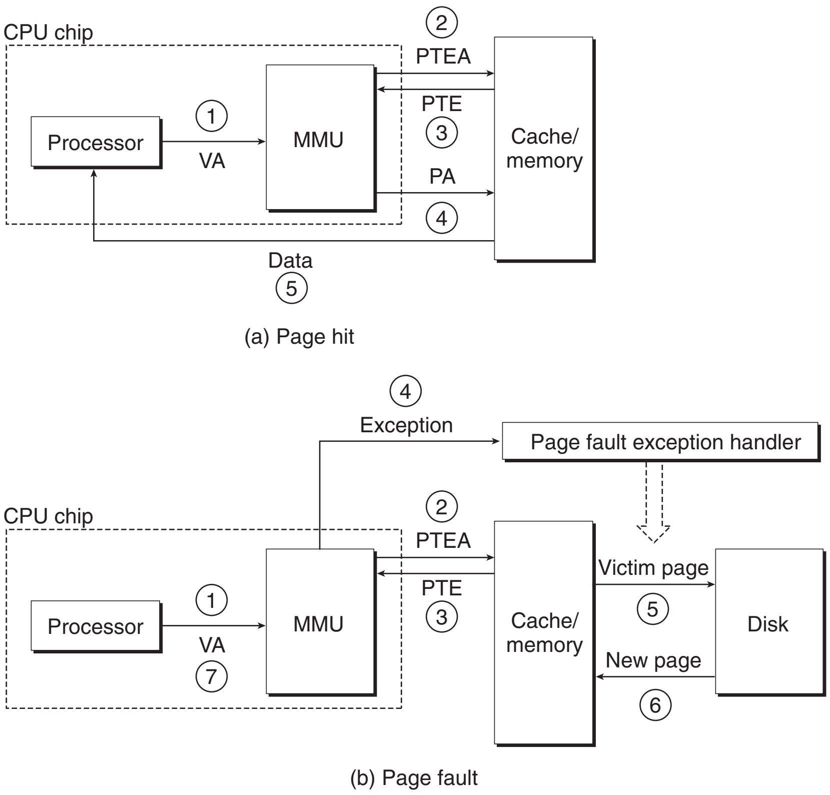


Figure 9.13 **Operational view of page hits and page faults.** VA: virtual address. PTEA: page table entry address. PTE: page table entry. PA: physical address.

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 9.13(a), the main memory returns the requested word to the processor.

Practice Problem 9.3 (solution page 917)

Given a 64-bit virtual address space and a 32-bit physical address, determine the number of bits in the VPN, VPO, PPN, and PPO for the following page sizes *P*:

	Number of			
P	VPN bits	VPO bits	PPN bits	PPO bits
1 KB				
2 KB				
4 KB				
16 KB				

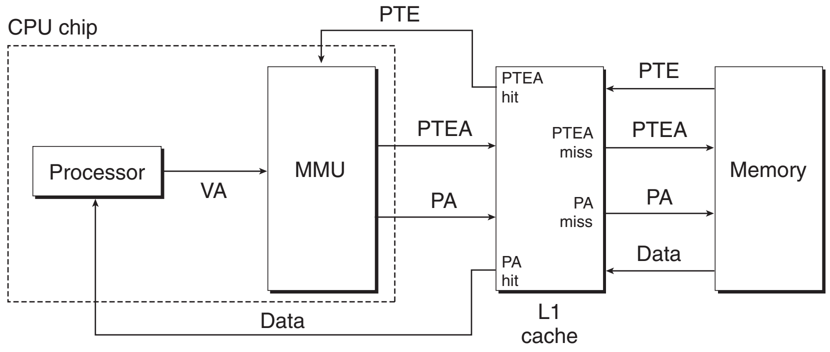


Figure 9.14 **Integrating VM with a physically addressed cache.** VA: virtual address. PTEA: page table entry address. PTE: page table entry. PA: physical address.

9.6.1 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 SRAM cache. Although a detailed discussion of the trade-offs is beyond our scope here, 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 9.14 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.

9.6.2 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 a handful of 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 9.15, 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 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 9.15
Components of a virtual address that are used to access the TLB.

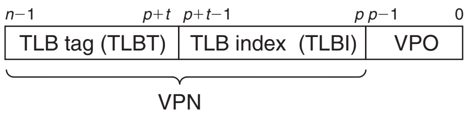


Figure 9.16
Operational view of a TLB hit and miss.

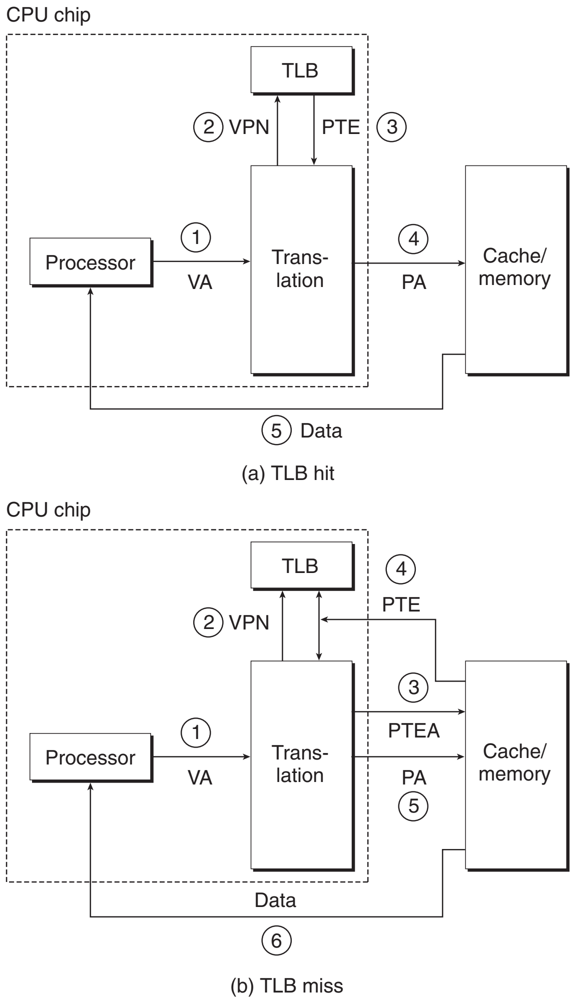


Figure 9.16(a) shows the steps involved when there is a TLB hit (the usual case). The key point here is that all of the address translation steps are performed inside the on-chip MMU and thus are fast.

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 9.16(b). The newly fetched PTE is stored in the TLB, possibly overwriting an existing entry.

9.6.3 Multi-Level Page Tables

Thus far, 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 address spaces.

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. Consider a 32-bit virtual address space partitioned into 4 KB pages, with page table entries that are 4 bytes each. Suppose also that at this point in time the virtual address space has the following form: The first 2 K pages of memory are allocated for code and data, the next 6 K pages are unallocated, the next 1,023 pages are also unallocated, and the next page is allocated for the user stack. Figure 9.17 shows how we might construct a two-level page table hierarchy for this virtual address space.

Each PTE in the level 1 table is responsible for mapping a 4 MB chunk of the virtual address space, where each chunk consists of 1,024 contiguous pages. For example, PTE 0 maps the first chunk, PTE 1 the next chunk, and so on. Given that the address space is 4 GB, 1,024 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 9.17, 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 9.17, 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 4 kilobytes, 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.

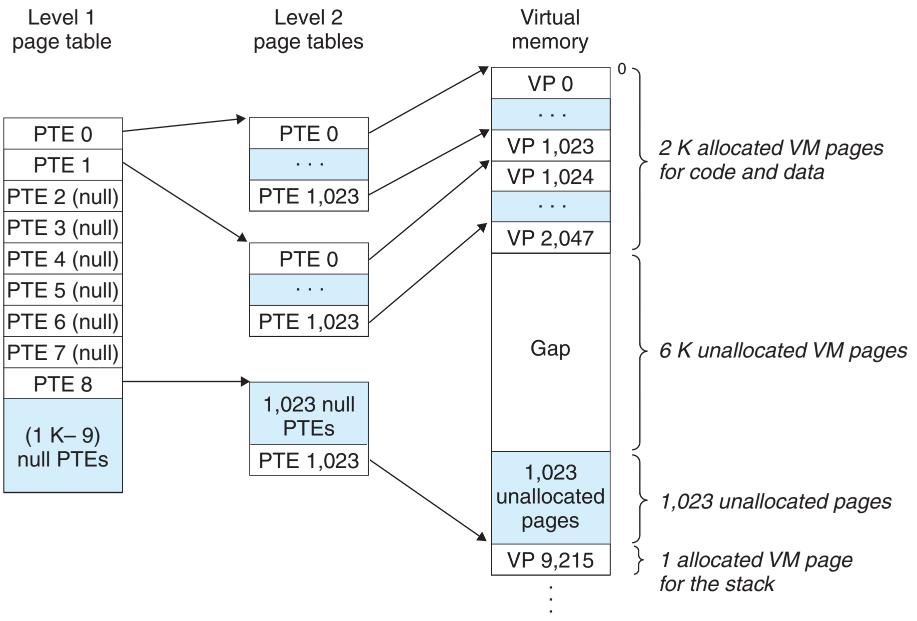


Figure 9.17 **A two-level page table hierarchy.** Notice that addresses increase from top to bottom.

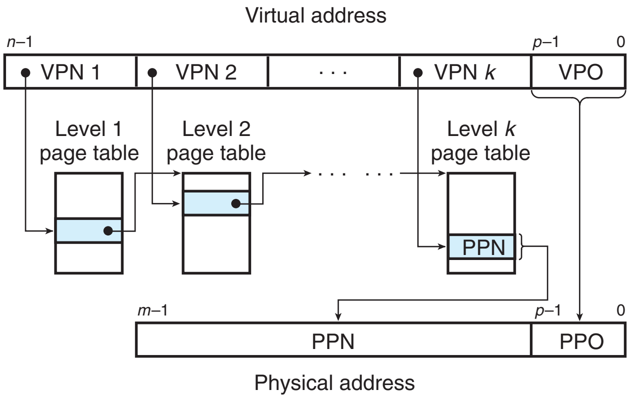


Figure 9.18 **Address translation with a k-level page table.**

Figure 9.18 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.

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.

9.6.4 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 4-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 9.19 shows the formats of the virtual and physical addresses. Since each page is $2^6 = 64$ bytes, the low-order 6 bits of the virtual and physical addresses serve as the VPO and PPO, respectively. The high-order 8 bits of the virtual address serve as the VPN. The high-order 6 bits of the physical address serve as the PPN.

Figure 9.20 shows a snapshot of our little memory system, including the TLB (Figure 9.20(a)), a portion of the page table (Figure 9.20(b)), and the L1 cache (Figure 9.20(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 as it accesses these devices.

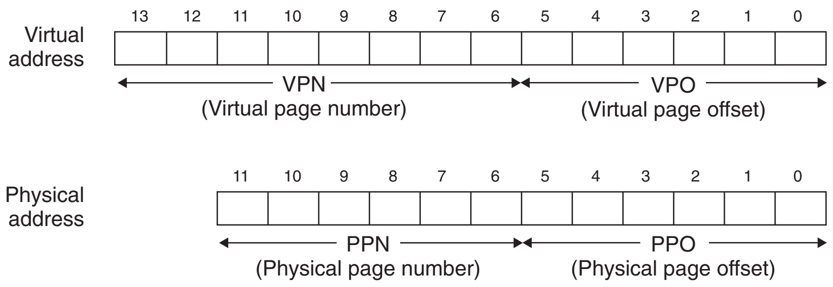
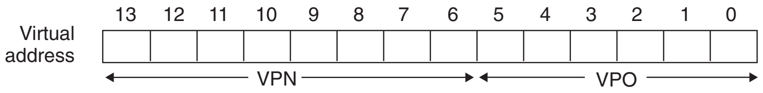


Figure 9.19 Addressing for small memory system. Assume 14-bit virtual addresses ($n = 14$), 12-bit physical addresses ($m = 12$), and 64-byte pages ($P = 64$).

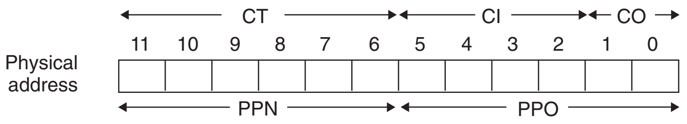


Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

(a) TLB: 4 sets, 16 entries, 4-way set associative

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

(b) Page table: Only the first 16 PTEs are shown



Idx	Tag	Valid	Blk 0	Blk 1	Blk 2	Blk 3
0	19	1	99	11	23	11
1	15	0	—	—	—	—
2	1B	1	00	02	04	08
3	36	0	—	—	—	—
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	—	—	—	—
7	16	1	11	C2	DF	03
8	24	1	3A	00	51	89
9	2D	0	—	—	—	—
A	2D	1	93	15	DA	3B
B	0B	0	—	—	—	—
C	12	0	—	—	—	—
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	—	—	—	—

(c) Cache: 16 sets, 4-byte blocks, direct mapped

Figure 9.20 TLB, page table, and cache for small memory system. All values in the TLB, page table, and cache are in hexadecimal notation.

TLB. The TLB is virtually addressed using the bits of the VPN. Since the TLB has four sets, the 2 low-order bits of the VPN serve as the set index (TLBI). The remaining 6 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 16 of these. For convenience, we have labeled each PTE with the VPN that indexes it; but keep in mind 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 1-byte words rather than 4-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					

To begin, the MMU extracts the VPN (`0x0F`) from the virtual address and checks with the TLB to see if it 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 9.4 (solution page 917)

Show how the example memory system in Section 9.6.4 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

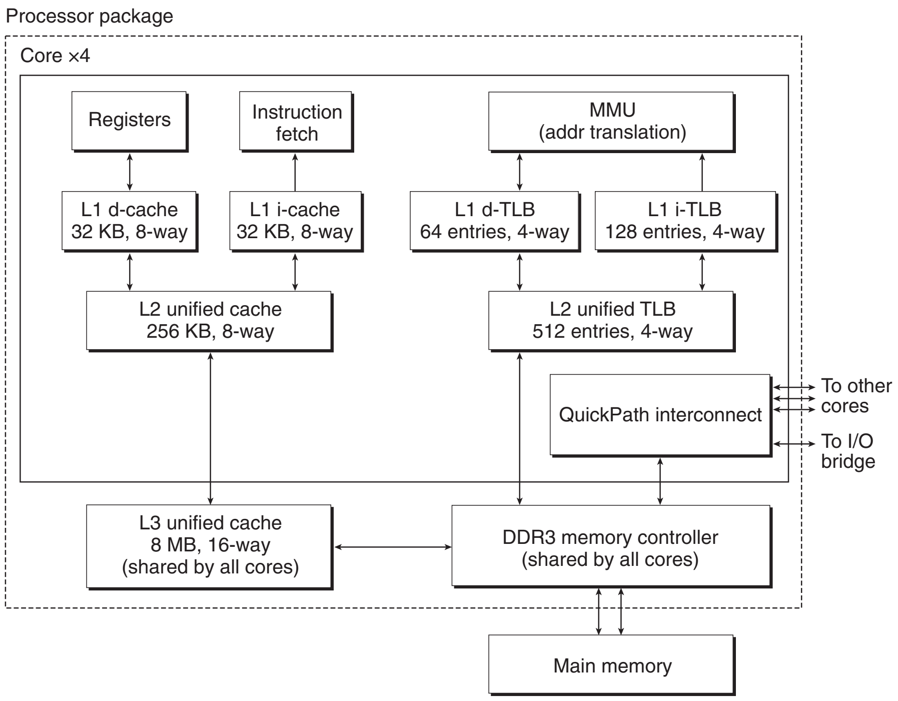
D. Physical memory reference

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

9.7 Case Study: The Intel Core i7/Linux Memory System

We conclude our discussion of virtual memory mechanisms with a case study of a real system: an Intel Core i7 running Linux. Although the underlying Haswell microarchitecture allows for full 64-bit virtual and physical address spaces, the current Core i7 implementations (and those for the foreseeable future) support a 48-bit (256 TB) virtual address space and a 52-bit (4 PB) physical address space, along with a compatibility mode that supports 32-bit (4 GB) virtual and physical address spaces.

Figure 9.21 gives the highlights of the Core i7 memory system. The *processor package* (chip) includes four cores, a large L3 cache shared by all of the cores, and



Processor package
Core ×4
Registers
Instruction fetch
MMU (addr translation)
L1 d-cache 32 KB, 8-way
L1 i-cache 32 KB, 8-way
L1 d-TLB 64 entries, 4-way
L1 i-TLB 128 entries, 4-way
L2 unified cache 256 KB, 8-way
L2 unified TLB 512 entries, 4-way
QuickPath interconnect
To other cores
To I/O bridge
L3 unified cache 8 MB, 16-way (shared by all cores)
DDR3 memory controller (shared by all cores)
Main memory

Figure 9.21 The Core i7 memory system.

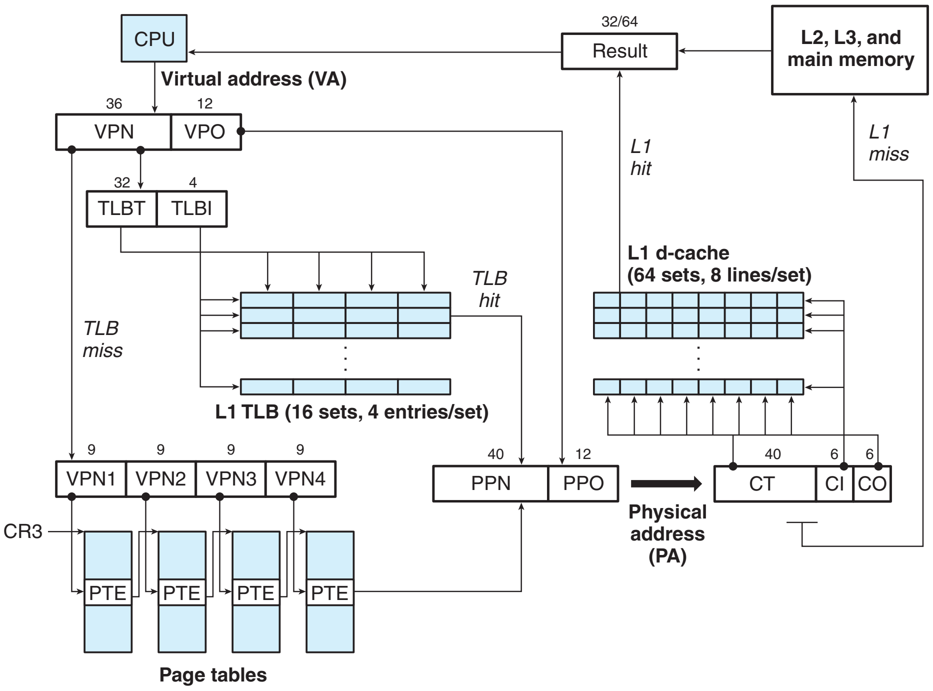
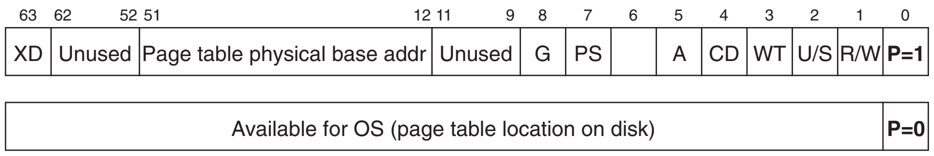


Figure 9.22 Summary of Core i7 address translation. For simplicity, the i-caches, i-TLB, and L2 unified TLB are not shown.

a DDR3 memory controller. Each core contains a hierarchy of TLBs, a hierarchy of data and instruction caches, and a set of fast point-to-point links, based on the QuickPath technology, for communicating directly with the other cores and the external I/O bridge. The TLBs are virtually addressed, and 4-way set associative. The L1, L2, and L3 caches are physically addressed, with a block size of 64 bytes. L1 and L2 are 8-way set associative, and L3 is 16-way set associative. The page size can be configured at start-up time as either 4 KB or 4 MB. Linux uses 4 KB pages.

9.7.1 Core i7 Address Translation

Figure 9.22 summarizes the entire Core i7 address translation process, from the time the CPU generates a virtual address until a data word arrives from memory. The Core i7 uses a four-level page table hierarchy. Each process has its own private page table hierarchy. When a Linux process is running, the page tables associated with allocated pages are all memory-resident, although the Core i7 architecture allows these page tables to be swapped in and out. The *CR3* control register contains the physical address of the beginning of the level 1 (L1) page table. The value of CR3 is part of each process context, and is restored during each context switch.


63 62 52 51 12 11 9 8 7 6 5 4 3 2 1 0
XD
Unused
Page table physical base addr
Unused
G
PS
A
CD
WT
U/S
R/W
P=1
Available for OS (page table location on disk)
P=0

Field	Description
P	Child page table present in physical memory (1) or not (0).
R/W	Read-only or read-write access permission for all reachable pages.
U/S	User or supervisor (kernel) mode access permission for all reachable pages.
WT	Write-through or write-back cache policy for the child page table.
CD	Caching disabled or enabled for the child page table.
A	Reference bit (set by MMU on reads and writes, cleared by software).
PS	Page size either 4 KB or 4 MB (defined for level 1 PTEs only).
Base addr	40 most significant bits of physical base address of child page table.
XD	Disable or enable instruction fetches from all pages reachable from this PTE.

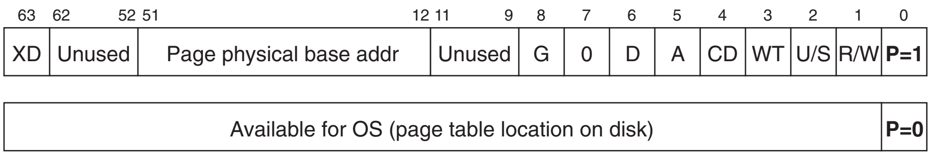
Figure 9.23 Format of level 1, level 2, and level 3 page table entries. Each entry references a 4 KB child page table.

Figure 9.23 shows the format of an entry in a level 1, level 2, or level 3 page table. When $P = 1$ (which is always the case with Linux), the address field contains a 40-bit physical page number (PPN) that points to the beginning of the appropriate page table. Notice that this imposes a 4 KB alignment requirement on page tables.

Figure 9.24 shows the format of an entry in a level 4 page table. When $P = 1$, the address field contains a 40-bit PPN that points to the base of some page in physical memory. Again, this imposes a 4 KB alignment requirement on physical pages.

The PTE has three permission bits that control access to the page. The R/W bit determines whether the contents of a page are read/write or read-only. The U/S bit, which determines whether the page can be accessed in user mode, protects code and data in the operating system kernel from user programs. The XD (execute disable) bit, which was introduced in 64-bit systems, can be used to disable instruction fetches from individual memory pages. This is an important new feature that allows the operating system kernel to reduce the risk of buffer overflow attacks by restricting execution to the read-only code segment.

As the MMU translates each virtual address, it also updates two other bits that can be used by the kernel's page fault handler. The MMU sets the A bit, which is known as a *reference bit*, each time a page is accessed. The kernel can use the reference bit to implement its page replacement algorithm. The MMU sets the D bit, or *dirty bit*, each time the page is written to. A page that has been modified is sometimes called a *dirty page*. The dirty bit tells the kernel whether or not it must



Field	Description
P	Child page present in physical memory (1) or not (0).
R/W	Read-only or read/write access permission for child page.
U/S	User or supervisor mode (kernel mode) access permission for child page.
WT	Write-through or write-back cache policy for the child page.
CD	Cache disabled or enabled.
A	Reference bit (set by MMU on reads and writes, cleared by software).
D	Dirty bit (set by MMU on writes, cleared by software).
G	Global page (don't evict from TLB on task switch).
Base addr	40 most significant bits of physical base address of child page.
XD	Disable or enable instruction fetches from the child page.

Figure 9.24 **Format of level 4 page table entries.** Each entry references a 4 KB child page.

write back a victim page before it copies in a replacement page. The kernel can call a special kernel-mode instruction to clear the reference or dirty bits.

Figure 9.25 shows how the Core i7 MMU uses the four levels of page tables to translate a virtual address to a physical address. The 36-bit VPN is partitioned into four 9-bit chunks, each of which is used as an offset into a page table. The CR3 register contains the physical address of the L1 page table. VPN 1 provides an offset to an L1 PTE, which contains the base address of the L2 page table. VPN 2 provides an offset to an L2 PTE, and so on.

9.7.2 Linux Virtual Memory System

A virtual memory system requires close cooperation between the hardware and the kernel. Details vary from version to version, and a complete description is beyond our scope. Nonetheless, our aim in this section is to describe enough of the Linux virtual memory system to give you a sense of how a real operating system organizes virtual memory and how it handles page faults.

Linux maintains a separate virtual address space for each process of the form shown in Figure 9.26. We have seen this picture a number of times already, with its familiar code, data, heap, shared library, and stack segments. Now that we understand address translation, we can fill in some more details about the kernel virtual memory that lies above the user stack.

The kernel virtual memory contains the code and data structures in the kernel. Some regions of the kernel virtual memory are mapped to physical pages that

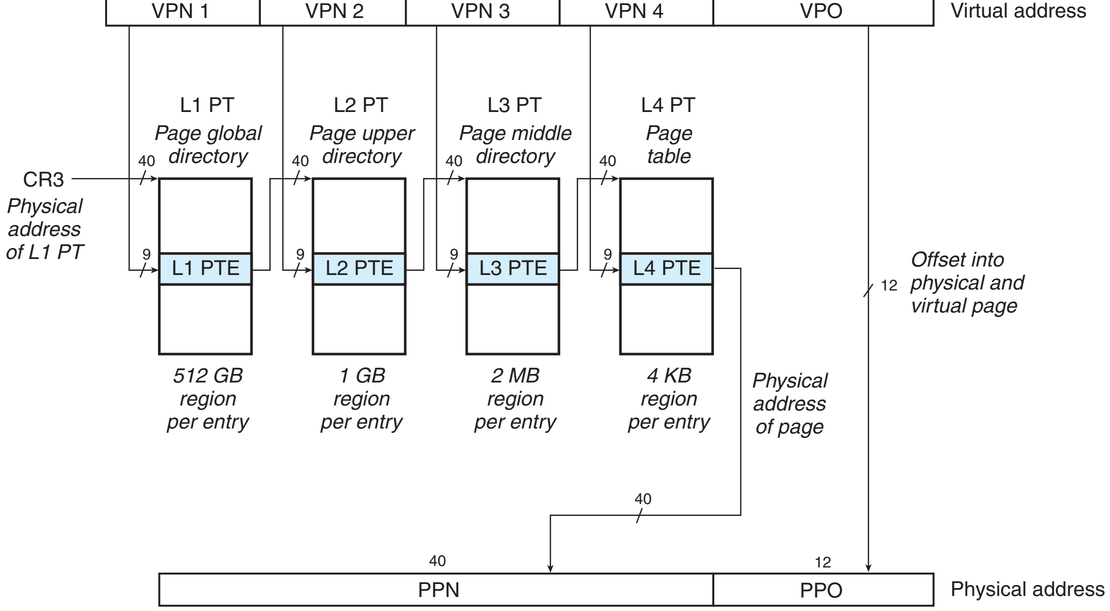
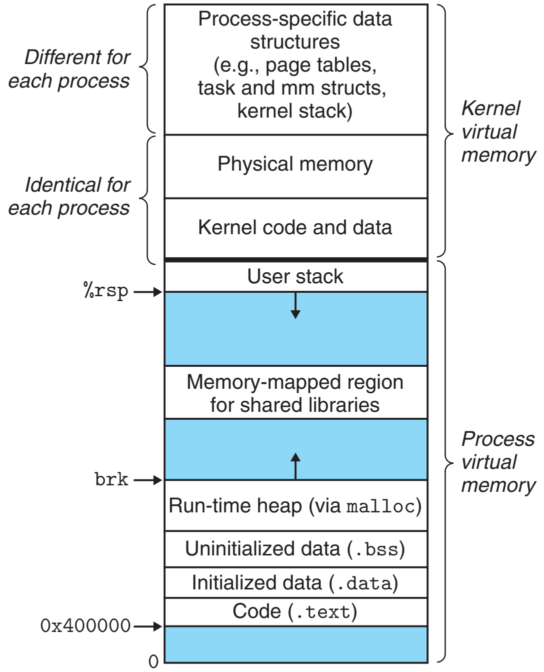


Figure 9.25 **Core i7 page table translation.** PT: page table; PTE: page table entry; VPN: virtual page number; VPO: virtual page offset; PPN: physical page number; PPO: physical page offset. The Linux names for the four levels of page tables are also shown.

Figure 9.26
The virtual memory of a Linux process.


Different for each process
Identical for each process
Process-specific data structures (e.g., page tables, task and mm structs, kernel stack)
Physical memory
Kernel code and data
Kernel virtual memory
User stack
%rsp
Memory-mapped region for shared libraries
brk
Run-time heap (via malloc)
Uninitialized data (.bss)
Initialized data (.data)
Code (.text)
0x400000
0
Process virtual memory

Aside Optimizing address translation

In our discussion of address translation, we have described a sequential two-step process where the MMU (1) translates the virtual address to a physical address and then (2) passes the physical address to the L1 cache. However, real hardware implementations use a neat trick that allows these steps to be partially overlapped, thus speeding up accesses to the L1 cache. For example, a virtual address on a Core i7 with 4 KB pages has 12 bits of VPO, and these bits are identical to the 12 bits of PPO in the corresponding physical address. Since the 8-way set associative physically addressed L1 caches have 64 sets and 64-byte cache blocks, each physical address has 6 ($\log_2 64$) cache offset bits and 6 ($\log_2 64$) index bits. These 12 bits fit exactly in the 12-bit VPO of a virtual address, which is no accident! When the CPU needs a virtual address translated, it sends the VPN to the MMU and the VPO to the L1 cache. While the MMU is requesting a page table entry from the TLB, the L1 cache is busy using the VPO bits to find the appropriate set and read out the eight tags and corresponding data words in that set. When the MMU gets the PPN back from the TLB, the cache is ready to try to match the PPN to one of these eight tags.

are shared by all processes. For example, each process shares the kernel's code and global data structures. Interestingly, Linux also maps a set of contiguous virtual pages (equal in size to the total amount of DRAM in the system) to the corresponding set of contiguous physical pages. This provides the kernel with a convenient way to access any specific location in physical memory—for example, when it needs to access page tables or to perform memory-mapped I/O operations on devices that are mapped to particular physical memory locations.

Other regions of kernel virtual memory contain data that differ for each process. Examples include page tables, the stack that the kernel uses when it is executing code in the context of the process, and various data structures that keep track of the current organization of the virtual address space.

Linux Virtual Memory Areas

Linux organizes the virtual memory as a collection of *areas* (also called *segments*). An area is a contiguous chunk of existing (allocated) virtual memory whose pages are related in some way. For example, the code segment, data segment, heap, shared library segment, and user stack are all distinct areas. Each existing virtual page is contained in some area, and any virtual page that is not part of some area does not exist and cannot be referenced by the process. The notion of an area is important because it allows the virtual address space to have gaps. The kernel does not keep track of virtual pages that do not exist, and such pages do not consume any additional resources in memory, on disk, or in the kernel itself.

Figure 9.27 highlights the kernel data structures that keep track of the virtual memory areas in a process. The kernel maintains a distinct task structure (`task_struct` in the source code) for each process in the system. The elements of the task structure either contain or point to all of the information that the kernel needs to

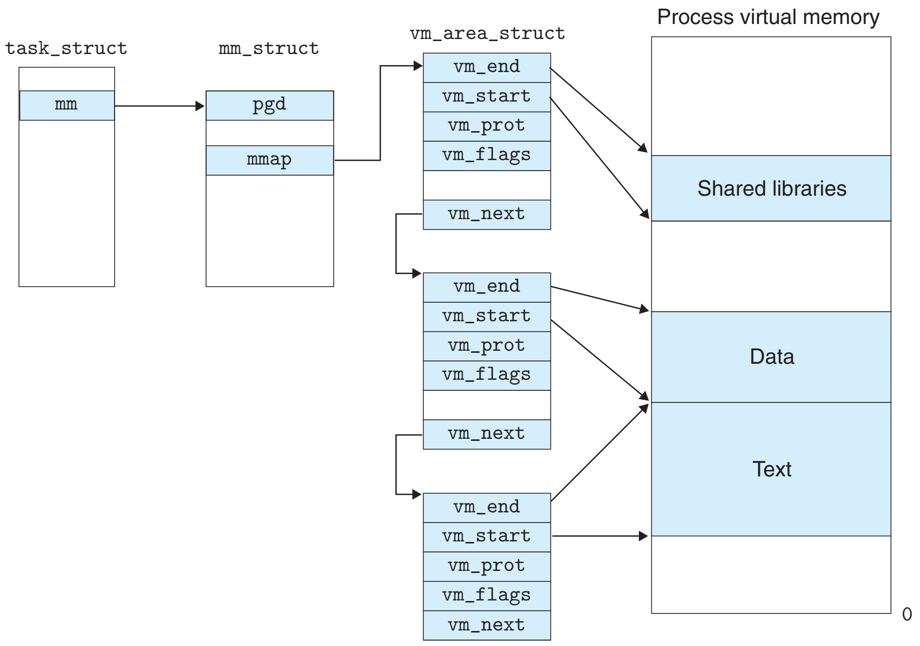


Figure 9.27 **How Linux organizes virtual memory.**

run the process (e.g., the PID, pointer to the user stack, name of the executable object file, and program counter).

One of the entries in the task structure points to an `mm_struct` that characterizes the current state of the virtual memory. The two fields of interest to us are `pgd`, which points to the base of the level 1 table (the page global directory), and `mmap`, which points to a list of `vm_area_structs` (area structs), each of which characterizes an area of the current virtual address space. When the kernel runs this process, it stores `pgd` in the CR3 control register.

For our purposes, the area struct for a particular area contains the following fields:

`fvm_start`. Points to the beginning of the area.

`vm_end`. Points to the end of the area.

`vm_prot`. Describes the read/write permissions for all of the pages contained in the area.

`vm_flags`. Describes (among other things) whether the pages in the area are shared with other processes or private to this process.

`vm_next`. Points to the next area struct in the list.

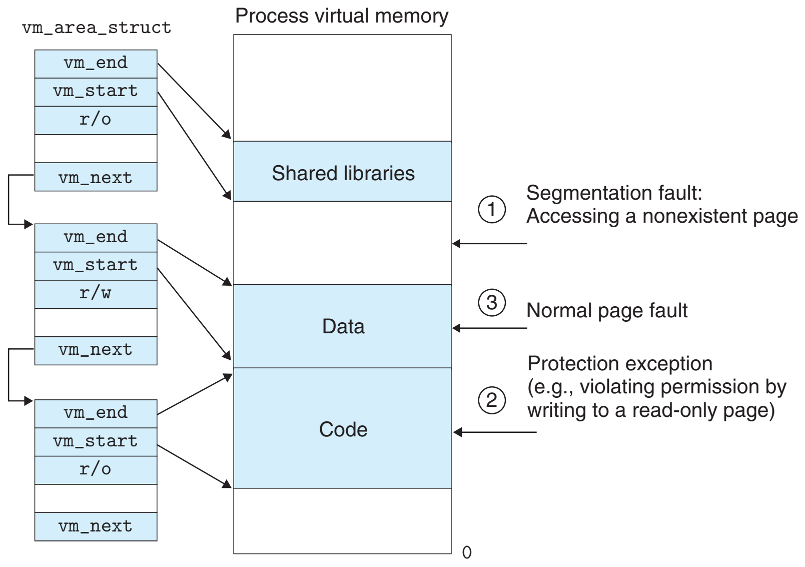


Figure 9.28
Linux page fault handling.

Linux Page Fault Exception Handling

Suppose the MMU triggers a page fault while trying to translate some virtual address *A*. The exception results in a transfer of control to the kernel's page fault handler, which then performs the following steps:

1. Is virtual address *A* legal? In other words, does *A* lie within an area defined by some area struct? To answer this question, the fault handler searches the list of area structs, comparing *A* with the `vm_start` and `vm_end` in each area struct. If the instruction is not legal, then the fault handler triggers a segmentation fault, which terminates the process. This situation is labeled "1" in Figure 9.28.

   Because a process can create an arbitrary number of new virtual memory areas (using the `mmap` function described in the next section), a sequential search of the list of area structs might be very costly. So in practice, Linux superimposes a tree on the list, using some fields that we have not shown, and performs the search on this tree.
2. Is the attempted memory access legal? In other words, does the process have permission to read, write, or execute the pages in this area? For example, was the page fault the result of a store instruction trying to write to a read-only page in the code segment? Is the page fault the result of a process running in user mode that is attempting to read a word from kernel virtual memory? If the attempted access is not legal, then the fault handler triggers a protection exception, which terminates the process. This situation is labeled "2" in Figure 9.28.
3. At this point, the kernel knows that the page fault resulted from a legal operation on a legal virtual address. It handles the fault by selecting a victim page, swapping out the victim page if it is dirty, swapping in the new page,

and updating the page table. When the page fault handler returns, the CPU restarts the faulting instruction, which sends *A* to the MMU again. This time, the MMU translates *A* normally, without generating a page fault.

9.8 Memory Mapping

Linux initializes the contents of a virtual memory area by associating it with an *object* on disk, a process known as *memory mapping*. Areas can be mapped to one of two types of objects:

1. *Regular file in the Linux file system:* An area can be mapped to a contiguous section of a regular disk file, such as an executable object file. The file section is divided into page-size pieces, with each piece containing the initial contents of a virtual page. Because of demand paging, none of these virtual pages is actually swapped into physical memory until the CPU first *touches* the page (i.e., issues a virtual address that falls within that page's region of the address space). If the area is larger than the file section, then the area is padded with zeros.
2. *Anonymous file:* An area can also be mapped to an anonymous file, created by the kernel, that contains all binary zeros. The first time the CPU touches a virtual page in such an area, the kernel finds an appropriate victim page in physical memory, swaps out the victim page if it is dirty, overwrites the victim page with binary zeros, and updates the page table to mark the page as resident. Notice that no data are actually transferred between disk and memory. For this reason, pages in areas that are mapped to anonymous files are sometimes called *demand-zero pages*.

In either case, once a virtual page is initialized, it is swapped back and forth between a special *swap file* maintained by the kernel. The swap file is also known as the *swap space* or the *swap area*. An important point to realize is that at any point in time, the swap space bounds the total amount of virtual pages that can be allocated by the currently running processes.

9.8.1 Shared Objects Revisited

The idea of memory mapping resulted from a clever insight that if the virtual memory system could be integrated into the conventional file system, then it could provide a simple and efficient way to load programs and data into memory.

As we have seen, the process abstraction promises to provide each process with its own private virtual address space that is protected from errant writes or reads by other processes. However, many processes have identical read-only code areas. For example, each process that runs the Linux shell program `bash` has the same code area. Further, many programs need to access identical copies of read-only run-time library code. For example, every C program requires functions from the standard C library such as `printf`. It would be extremely wasteful for each process to keep duplicate copies of these commonly used codes in physical memory. Fortunately, memory mapping provides us with a clean mechanism for controlling how objects are shared by multiple processes.

An object can be mapped into an area of virtual memory as either a *shared object* or a *private object*. If a process maps a shared object into an area of its virtual address space, then any writes that the process makes to that area are visible to any other processes that have also mapped the shared object into their virtual memory. Further, the changes are also reflected in the original object on disk.

Changes made to an area mapped to a private object, on the other hand, are not visible to other processes, and any writes that the process makes to the area are *not* reflected back to the object on disk. A virtual memory area into which a shared object is mapped is often called a *shared area*. Similarly for a *private area*.

Suppose that process 1 maps a shared object into an area of its virtual memory, as shown in Figure 9.29(a). Now suppose that process 2 maps the same shared ob-

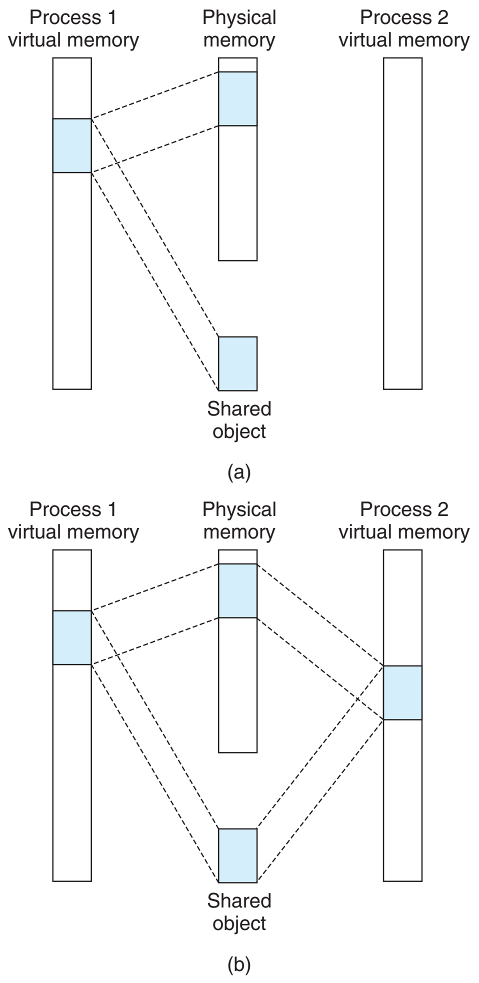


Figure 9.29
A shared object. (a) After process 1 maps the shared object. (b) After process 2 maps the same shared object. (Note that the physical pages are not necessarily contiguous.)

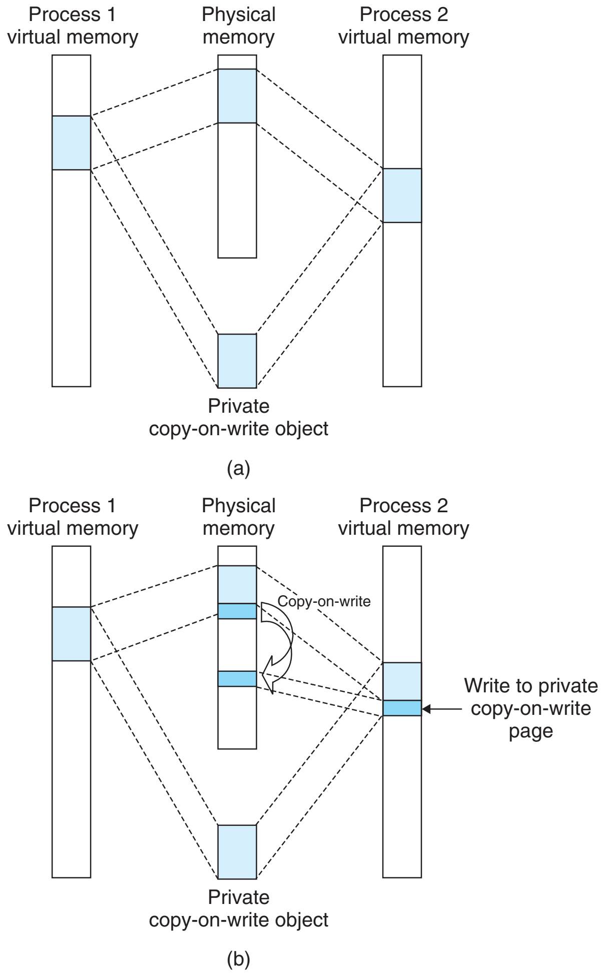


Figure 9.30

A private copy-on-write object. (a) After both processes have mapped the private copy-on-write object. (b) After process 2 writes to a page in the private area.

ject into its address space (not necessarily at the same virtual address as process 1), as shown in Figure 9.29(b).

Since each object has a unique filename, the kernel can quickly determine that process 1 has already mapped this object and can point the page table entries in process 2 to the appropriate physical pages. The key point is that only a single copy of the shared object needs to be stored in physical memory, even though the object is mapped into multiple shared areas. For convenience, we have shown the physical pages as being contiguous, but of course this is not true in general.

Private objects are mapped into virtual memory using a clever technique known as *copy-on-write*. A private object begins life in exactly the same way as a shared object, with only one copy of the private object stored in physical memory. For example, Figure 9.30(a) shows a case where two processes have mapped a private object into different areas of their virtual memories but share the same physical copy of the object. For each process that maps the private object, the page table entries for the corresponding private area are flagged as read-only, and the area struct is flagged as *private copy-on-write*. So long as neither process attempts to write to its respective private area, they continue to share a single copy of the object in physical memory. However, as soon as a process attempts to write to some page in the private area, the write triggers a protection fault.

When the fault handler notices that the protection exception was caused by the process trying to write to a page in a private copy-on-write area, it creates a new copy of the page in physical memory, updates the page table entry to point to the new copy, and then restores write permissions to the page, as shown in Figure 9.30(b). When the fault handler returns, the CPU re-executes the write, which now proceeds normally on the newly created page.

By deferring the copying of the pages in private objects until the last possible moment, copy-on-write makes the most efficient use of scarce physical memory.

9.8.2 The `fork` Function Revisited

Now that we understand virtual memory and memory mapping, we can get a clear idea of how the `fork` function creates a new process with its own independent virtual address space.

When the `fork` function is called by the *current process*, the kernel creates various data structures for the *new process* and assigns it a unique PID. To create the virtual memory for the new process, it creates exact copies of the current process's `mm_struct`, area structs, and page tables. It flags each page in both processes as read-only, and flags each area struct in both processes as private copy-on-write.

When the `fork` returns in the new process, the new process now has an exact copy of the virtual memory as it existed when the fork was called. When either of the processes performs any subsequent writes, the copy-on-write mechanism creates new pages, thus preserving the abstraction of a private address space for each process.

9.8.3 The `execve` Function Revisited

Virtual memory and memory mapping also play key roles in the process of loading programs into memory. Now that we understand these concepts, we can understand how the `execve` function really loads and executes programs. Suppose that the program running in the current process makes the following call:

```
execve("a.out", NULL, NULL);
```

As you learned in Chapter 8, the `execve` function loads and runs the program contained in the executable object file `a.out` within the current process, effectively replacing the current program with the `a.out` program. Loading and running `a.out` requires the following steps:

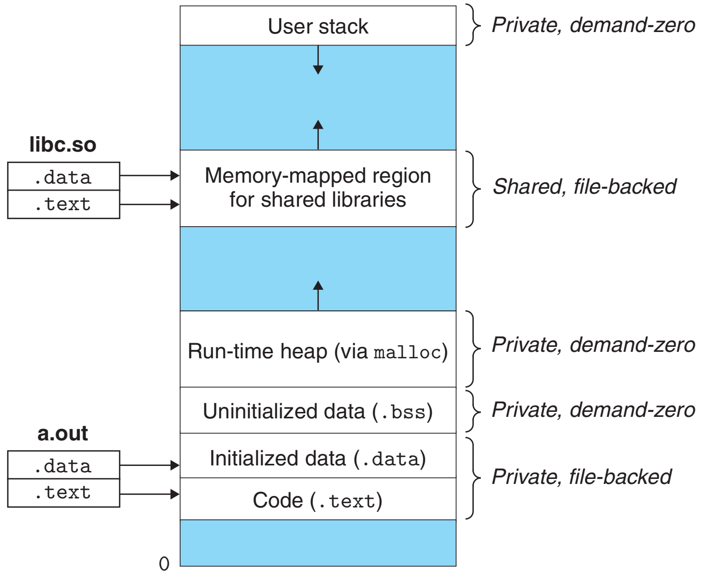


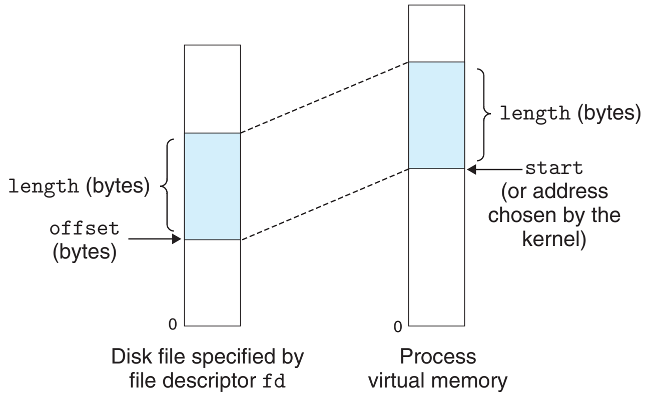
Figure 9.31
How the loader maps the areas of the user address space.

1. *Delete existing user areas.* Delete the existing area structs in the user portion of the current process's virtual address.
2. *Map private areas.* Create new area structs for the code, data, bss, and stack areas of the new program. All of these new areas are private copy-on-write. The code and data areas are mapped to the `.text` and `.data` sections of the `a.out` file. The bss area is demand-zero, mapped to an anonymous file whose size is contained in `a.out`. The stack and heap area are also demand-zero, initially of zero length. Figure 9.31 summarizes the different mappings of the private areas.
3. *Map shared areas.* If the `a.out` program was linked with shared objects, such as the standard C library `libc.so`, then these objects are dynamically linked into the program, and then mapped into the shared region of the user's virtual address space.
4. *Set the program counter (PC).* The last thing that `execve` does is to set the program counter in the current process's context to point to the entry point in the code area.

The next time this process is scheduled, it will begin execution from the entry point. Linux will swap in code and data pages as needed.

9.8.4 User-Level Memory Mapping with the `mmap` Function

Linux processes can use the `mmap` function to create new areas of virtual memory and to map objects into these areas.



length (bytes)
offset (bytes)
0
Disk file specified by file descriptor fd
length (bytes)
start (or address chosen by the kernel)
0
Process virtual memory

Figure 9.32 **Visual interpretation of `mmap` arguments.**

```
#include <unistd.h>
#include <sys/mman.h>

void  *mmap(void *start, size_t length, int prot, int flags,
            int fd, off_t offset);
```
Returns: pointer to mapped area if OK, MAP_FAILED (−1) on error

The `mmap` function asks the kernel to create a new virtual memory area, preferably one that starts at address `start`, and to map a contiguous chunk of the object specified by file descriptor `fd` to the new area. The contiguous object chunk has a size of `length` bytes and starts at an offset of `offset` bytes from the beginning of the file. The `start` address is merely a hint, and is usually specified as NULL. For our purposes, we will always assume a NULL start address. Figure 9.32 depicts the meaning of these arguments.

The `prot` argument contains bits that describe the access permissions of the newly mapped virtual memory area (i.e., the `vm_prot` bits in the corresponding area struct).

PROT_EXEC. Pages in the area consist of instructions that may be executed by the CPU.

PROT_READ. Pages in the area may be read.

PROT_WRITE. Pages in the area may be written.

PROT_NONE. Pages in the area cannot be accessed.

The `flags` argument consists of bits that describe the type of the mapped object. If the MAP_ANON flag bit is set, then the backing store is an anonymous object and the corresponding virtual pages are demand-zero. MAP_PRIVATE indicates a private copy-on-write object, and MAP_SHARED indicates a shared object. For example,

```
bufp = Mmap(NULL, size, PROT_READ, MAP_PRIVATE|MAP_ANON, 0, 0);
```

asks the kernel to create a new read-only, private, demand-zero area of virtual memory containing `size` bytes. If the call is successful, then `bufp` contains the address of the new area.

The `munmap` function deletes regions of virtual memory:

```
#include <unistd.h>
#include <sys/mman.h>

int munmap(void *start, size_t length);
```

Returns: 0 if OK, −1 on error

The `munmap` function deletes the area starting at virtual address `start` and consisting of the next `length` bytes. Subsequent references to the deleted region result in segmentation faults.

Practice Problem 9.5 (solution page 918)

Write a C program `mmapcopy.c` that uses `mmap` to copy an arbitrary-size disk file to `stdout`. The name of the input file should be passed as a command-line argument.

9.9 Dynamic Memory Allocation

While it is certainly possible to use the low-level `mmap` and `munmap` functions to create and delete areas of virtual memory, C programmers typically find it more convenient and more portable to use a *dynamic memory allocator* when they need to acquire additional virtual memory at run time.

A dynamic memory allocator maintains an area of a process's virtual memory known as the *heap* (Figure 9.33). Details vary from system to system, but without loss of generality, we will assume that the heap is an area of demand-zero memory that begins immediately after the uninitialized data area and grows upward (toward higher addresses). For each process, the kernel maintains a variable `brk` (pronounced "break") that points to the top of the heap.

An allocator maintains the heap as a collection of various-size *blocks*. Each block is a contiguous chunk of virtual memory that is either *allocated* or *free*. An allocated block has been explicitly reserved for use by the application. A free block is available to be allocated. A free block remains free until it is explicitly allocated by the application. An allocated block remains allocated until it is freed, either explicitly by the application or implicitly by the memory allocator itself.

Allocators come in two basic styles. Both styles require the application to explicitly allocate blocks. They differ about which entity is responsible for freeing allocated blocks.

- *Explicit allocators* require the application to explicitly free any allocated blocks. For example, the C standard library provides an explicit allocator called the `malloc` package. C programs allocate a block by calling the `malloc`

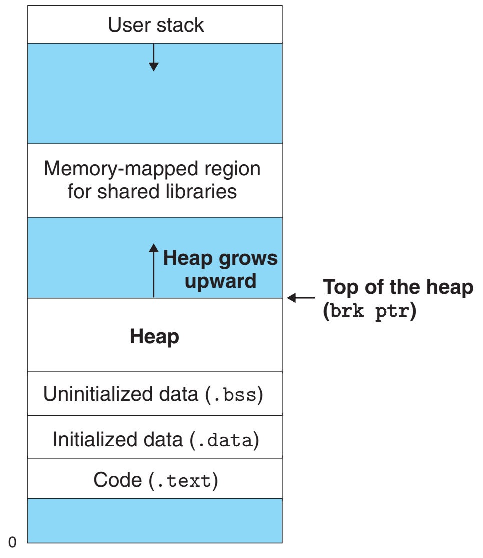

User stack
Memory-mapped region for shared libraries
Heap grows upward
Top of the heap (brk ptr)
Heap
Uninitialized data (.bss)
Initialized data (.data)
Code (.text)
0

Figure 9.33 The heap.

function, and free a block by calling the `free` function. The `new` and `delete` calls in C++ are comparable.

- *Implicit allocators*, on the other hand, require the allocator to detect when an allocated block is no longer being used by the program and then free the block. Implicit allocators are also known as *garbage collectors*, and the process of automatically freeing unused allocated blocks is known as *garbage collection*. For example, higher-level languages such as Lisp, ML, and Java rely on garbage collection to free allocated blocks.

The remainder of this section discusses the design and implementation of explicit allocators. We will discuss implicit allocators in Section 9.10. For concreteness, our discussion focuses on allocators that manage heap memory. However, you should be aware that memory allocation is a general idea that arises in a variety of contexts. For example, applications that do intensive manipulation of graphs will often use the standard allocator to acquire a large block of virtual memory and then use an application-specific allocator to manage the memory within that block as the nodes of the graph are created and destroyed.

9.9.1 The `malloc` and `free` Functions

The C standard library provides an explicit allocator known as the `malloc` package. Programs allocate blocks from the heap by calling the `malloc` function.

```
#include <stdlib.h>

void *malloc(size_t size);
```

Returns: pointer to allocated block if OK, NULL on error

Aside How big is a word?

Recall from our discussion of machine code in Chapter 3 that Intel refers to 4-byte objects as *double words*. However, throughout this section, we will assume that *words* are 4-byte objects and that *double words* are 8-byte objects, which is consistent with conventional terminology.

The `malloc` function returns a pointer to a block of memory of at least `size` bytes that is suitably aligned for any kind of data object that might be contained in the block. In practice, the alignment depends on whether the code is compiled to run in 32-bit mode (`gcc -m32`) or 64-bit mode (the default). In 32-bit mode, `malloc` returns a block whose address is always a multiple of 8. In 64-bit mode, the address is always a multiple of 16.

If `malloc` encounters a problem (e.g., the program requests a block of memory that is larger than the available virtual memory), then it returns NULL and sets `errno`. `Malloc` does not initialize the memory it returns. Applications that want initialized dynamic memory can use `calloc`, a thin wrapper around the `malloc` function that initializes the allocated memory to zero. Applications that want to change the size of a previously allocated block can use the `realloc` function.

Dynamic memory allocators such as `malloc` can allocate or deallocate heap memory explicitly by using the `mmap` and `munmap` functions, or they can use the `sbrk` function:

```
#include <unistd.h>

void *sbrk(intptr_t incr);
```

Returns: old `brk` pointer on success, −1 on error

The `sbrk` function grows or shrinks the heap by adding `incr` to the kernel's `brk` pointer. If successful, it returns the old value of `brk`, otherwise it returns −1 and sets `errno` to ENOMEM. If `incr` is zero, then `sbrk` returns the current value of `brk`. Calling `sbrk` with a negative `incr` is legal but tricky because the return value (the old value of `brk`) points to `abs(incr)` bytes past the new top of the heap.

Programs free allocated heap blocks by calling the `free` function.

```
#include <stdlib.h>

void free(void *ptr);
```

Returns: nothing

The `ptr` argument must point to the beginning of an allocated block that was obtained from `malloc`, `calloc`, or `realloc`. If not, then the behavior of `free` is undefined. Even worse, since it returns nothing, `free` gives no indication to the application that something is wrong. As we shall see in Section 9.11, this can produce some baffling run-time errors.

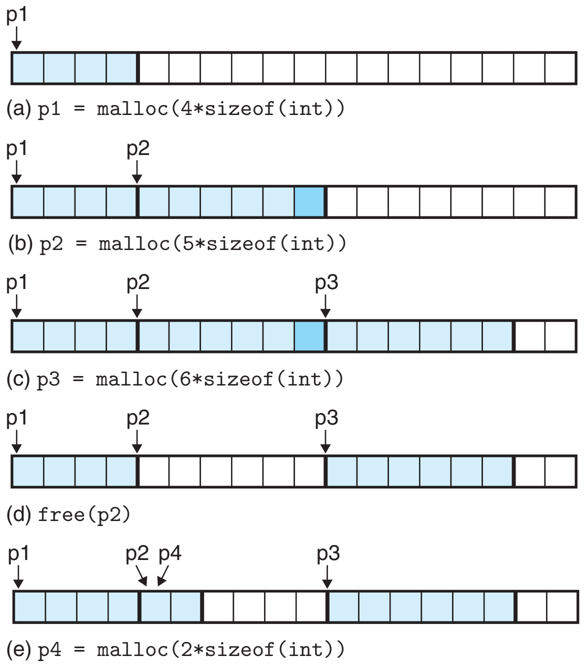

p1
(a) p1 = malloc(4*sizeof(int))
p1 p2
(b) p2 = malloc(5*sizeof(int))
p1 p2 p3
(c) p3 = malloc(6*sizeof(int))
p1 p2 p3
(d) free(p2)
p1 p2 p4 p3
(e) p4 = malloc(2*sizeof(int))

Figure 9.34
Allocating and freeing blocks with `malloc` and `free`. Each square corresponds to a word. Each heavy rectangle corresponds to a block. Allocated blocks are shaded. Padded regions of allocated blocks are shaded with a darker blue. Free blocks are unshaded. Heap addresses increase from left to right.

Figure 9.34 shows how an implementation of `malloc` and `free` might manage a (very) small heap of 16 words for a C program. Each box represents a 4-byte word. The heavy-lined rectangles correspond to allocated blocks (shaded) and free blocks (unshaded). Initially, the heap consists of a single 16-word double-word-aligned free block.[1]

Figure 9.34(a). The program asks for a four-word block. `Malloc` responds by carving out a four-word block from the front of the free block and returning a pointer to the first word of the block.

Figure 9.34(b). The program requests a five-word block. `Malloc` responds by allocating a six-word block from the front of the free block. In this example, `malloc` pads the block with an extra word in order to keep the free block aligned on a double-word boundary.

Figure 9.34(c). The program requests a six-word block and `malloc` responds by carving out a six-word block from the free block.

Figure 9.34(d). The program frees the six-word block that was allocated in Figure 9.34(b). Notice that after the call to `free` returns, the pointer `p2`

1. Throughout this section, we will assume that the allocator returns blocks aligned to 8-byte double-word boundaries.

still points to the freed block. It is the responsibility of the application not to use p2 again until it is reinitialized by a new call to malloc.

Figure 9.34(e). The program requests a two-word block. In this case, malloc allocates a portion of the block that was freed in the previous step and returns a pointer to this new block.

9.9.2 Why Dynamic Memory Allocation?

The most important reason that programs use dynamic memory allocation is that often they do not know the sizes of certain data structures until the program actually runs. For example, suppose we are asked to write a C program that reads a list of *n* ASCII integers, one integer per line, from stdin into a C array. The input consists of the integer *n*, followed by the *n* integers to be read and stored into the array. The simplest approach is to define the array statically with some hard-coded maximum array size:

```
#include "csapp.h"
#define MAXN 15213

int array[MAXN];

int main()
{
    int i, n;

    scanf("%d", &n);
    if (n > MAXN)
        app_error("Input file too big");
    for (i = 0; i < n; i++)
        scanf("%d", &array[i]);
    exit(0);
}
```

Allocating arrays with hard-coded sizes like this is often a bad idea. The value of MAXN is arbitrary and has no relation to the actual amount of available virtual memory on the machine. Further, if the user of this program wanted to read a file that was larger than MAXN, the only recourse would be to recompile the program with a larger value of MAXN. While not a problem for this simple example, the presence of hard-coded array bounds can become a maintenance nightmare for large software products with millions of lines of code and numerous users.

A better approach is to allocate the array dynamically, at run time, after the value of *n* becomes known. With this approach, the maximum size of the array is limited only by the amount of available virtual memory.

```
#include "csapp.h"

int main()
{
    int *array, i, n;

    scanf("%d", &n);
    array = (int *)Malloc(n * sizeof(int));
    for (i = 0; i < n; i++)
        scanf("%d", &array[i]);
    free(array);
    exit(0);
}
```

Dynamic memory allocation is a useful and important programming technique. However, in order to use allocators correctly and efficiently, programmers need to have an understanding of how they work. We will discuss some of the gruesome errors that can result from the improper use of allocators in Section 9.11.

9.9.3 Allocator Requirements and Goals

Explicit allocators must operate within some rather stringent constraints:

Handling arbitrary request sequences. An application can make an arbitrary sequence of allocate and free requests, subject to the constraint that each free request must correspond to a currently allocated block obtained from a previous allocate request. Thus, the allocator cannot make any assumptions about the ordering of allocate and free requests. For example, the allocator cannot assume that all allocate requests are accompanied by a matching free request, or that matching allocate and free requests are nested.

Making immediate responses to requests. The allocator must respond immediately to allocate requests. Thus, the allocator is not allowed to reorder or buffer requests in order to improve performance.

Using only the heap. In order for the allocator to be scalable, any nonscalar data structures used by the allocator must be stored in the heap itself.

Aligning blocks (alignment requirement). The allocator must align blocks in such a way that they can hold any type of data object.

Not modifying allocated blocks. Allocators can only manipulate or change free blocks. In particular, they are not allowed to modify or move blocks once they are allocated. Thus, techniques such as compaction of allocated blocks are not permitted.