

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 protection in several different ways, as we show throughout the chapter. Here, we outline one possible implementation.

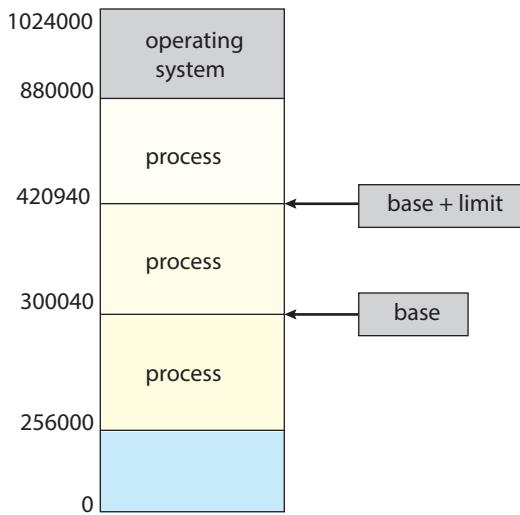


Figure 9.1 A base and a limit register define a logical address space.

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

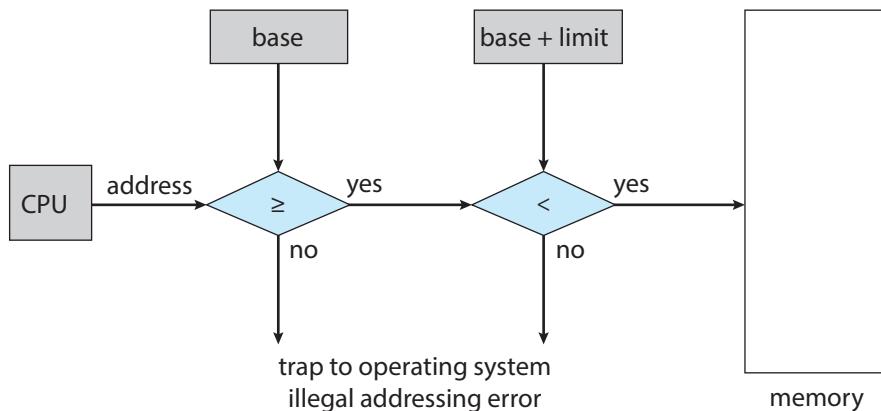


Figure 9.2 Hardware address protection with base and limit registers.

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.