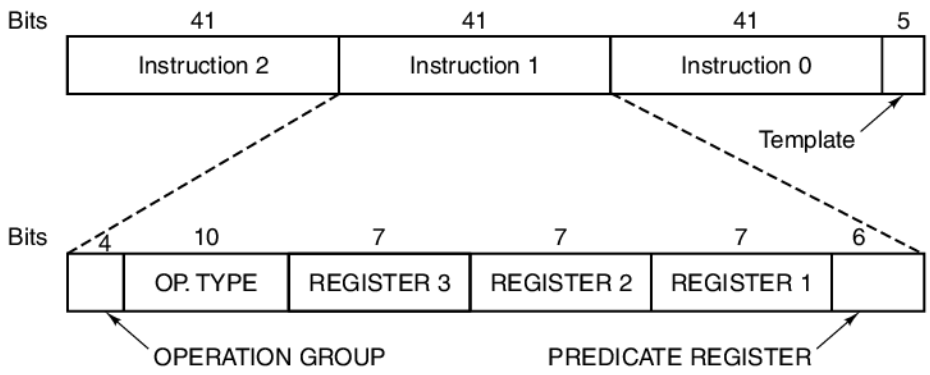

Figure 5-47. An IA-64 bundle contains three instructions.

plus another 216 to indicate an instruction-group marker after instruction 0, another 216 to indicate an instruction-group marker after instruction 1, and yet another 216 to indicate an instruction-group marker after instruction 2. With only 5 bits available, only a very limited number of these combinations are allowed. On the other hand, allowing three floating-point instructions in a bundle would not work, not even if there were a way to specify this, since the CPU cannot initiate three floating-point instructions simultaneously. The allowed combinations are the ones that are actually feasible.

5.8.5 Reducing Conditional Branches: Predication

Another important feature of the IA-64 is the new way it deals with conditional branches. If there were a way to get rid of most of them, CPUs could be made much simpler and faster. At first thought it might seem that getting rid of conditional branches would be impossible because programs are full of if statements. However, IA-64 uses a technique called **predication** that can greatly reduce their number (August et al., 1998, and Hwu, 1998). We will now briefly describe it.

In traditional architectures, all instructions are unconditional in the sense that when the CPU hits an instruction, it just carries the instruction out. There is no internal debate of the form: "To do or not to do, that is the question." In contrast, in a predicated architecture, instructions contain conditions (predicates) telling when they should be executed and when not. This paradigm shift from unconditional instructions to predicated instructions is what allows us to get rid of (many) conditional branches. Instead of having to make a choice between one sequence of unconditional instructions or another sequence of unconditional instructions, all the instructions are merged into a single sequence of predicated instructions, using different predicates for different instructions.

In order to see how predication works, let us start with the simple example of Fig. 5-48, which shows **conditional execution**, a precursor to predication. In Fig. 5-48(a) we see an if statement. In Fig. 5-48(b) we see its translation into three instructions: a comparison, a conditional branch, and a move instruction.

```
if (R1 == 0)          CMP R1,0          CMOVZ R2,R3,R1
    R2 = R3;          BNE L1
                      MOV R2,R3
                  L1:
```

(a) (b) (c)

Figure 5-48. (a) An if statement. (b) Generic assembly code for (a). (c) A conditional instruction.

In Fig. 5-48(c) we get rid of the conditional branch by using a new instruction, CMOVZ, which is a conditional move. What it does is check to see if the third register, R1, is 0. If so, it copies R3 to R2. If not, it does nothing.

Once we have an instruction that can copy data when some register is 0, it is a small step to an instruction that can copy data when some register is not 0, say CMOVN. With both of these instructions available, we are on our way to full conditional execution. Imagine an if statement with several assignments in the then part and several other assignments in the else part. The whole statement can be translated into code to set some register to 0 if the condition is false and to another value if it is true. Following the register setup, the then part assignments can be compiled into a sequence of CMOVN instructions and the else part assignments into a sequence of CMOVZ instructions.

All of these instructions, the register setup, the CMOVNs, and the CMOVZs form a single basic block with no conditional branch. The instructions can even be reordered, either by the compiler (including hoisting the assignments before the test) or during execution. The only catch is that the condition has to be known by the time the conditional instructions have to be retired (near the end of the pipeline). A simple example showing a then part and an else part is given in Fig. 5-49.

Although we have shown here only very simple conditional instructions (taken from the IA-32 ISA, actually), on the IA-64 all instructions are predicated. What this means is that the execution of every instruction can be made conditional. The extra 6-bit field referred to earlier selects one of 64 1-bit predicate registers. Thus an if statement will be compiled into code that sets one of the predicate registers to 1 if the condition is true and to 0 if it is false. Simultaneously and automatically, it sets another predicate register to the inverse value. Using predication, the machine instructions forming the then and else clauses will be merged into a single stream of instructions, the former ones using the predicate and the latter ones using its inverse. When control passes there, only one set of instructions will be executed.

```
if (R1 == 0) {          CMP R1,0          CMOVZ R2,R3,R1
    R2 = R3;            BNE L1            CMOVZ R4,R5,R1
    R4 = R5;            MOV R2,R3         CMOVN R6,R7,R1
} else {                MOV R4.R5         CMOVN R8,R9,R1
    R6 = R7;            BR L2
    R8 = R9;        L1: MOV R6,R7
}                       MOV R8,R9
                    L2:
     (a)                   (b)                  (c)
```

Figure 5-49. (a) An if statement. (b) Generic assembly code for (a). (c) Conditional execution.

Although simple, the example of Fig. 5-50 shows the basic idea of how predication can be used to eliminate branches. The CMPEQ instruction compares two registers and sets the predicate register P4 to 1 if they are equal and to 0 if they are different. It also sets a paired register, say, P5, to the inverse condition. Now the instructions for the if and then parts can be put after one another, each one conditioned on some predicate register (shown in angle brackets). Arbitrary code can be put here provided that each instruction is properly predicated.

```
if (R1 == R2)           CMP R1,R2         CMPEQ R1,R2,P4
    R3 = R4 + R5;       BNE L1            <P4> ADD R3,R4,R5
else                    MOV R3,R4         <P5> SUB R6,R4,R5
    R6 = R4 – R5        ADD R3,R5
                        BR L2
                    L1: MOV R6,R4
                        SUB R6,R5
                    L2:
     (a)                   (b)                  (c)
```

Figure 5-50. (a) An if statement. (b) Generic assembly code for (a). (c) Predicated execution.

In the IA-64, this idea is taken to the extreme, with comparison instructions for setting the predicate registers as well as arithmetic and other instructions whose execution is dependent on some predicate register. Predicated instructions can be stuffed into the pipeline in sequence, with no stalls and no problems. That is why they are so useful.

The way predication really works on the IA-64 is that every instruction is actually executed. At the very end of the pipeline, when it is time to retire an instruction, a check is made to see if the predicate is true. If so, the instruction is retired normally and its results are written back to the destination register. If the predicate is false, no writeback is done so the instruction has no effect. Predication is discussed further in Dulong (1998).

5.8.6 Speculative Loads

Another feature of the IA-64 that speeds up execution is the presence of speculative LOADs. If a LOAD is speculative and it fails, instead of causing an exception, it just stops and a bit associated with the register to be loaded is set marking the register as invalid. This is just the poison bit introduced in Chap. 4. If it turns out that the poisoned register is later used, the exception occurs at that time; otherwise, it never happens.

The way speculation is normally used is for the compiler to hoist LOADs to positions before they are needed. By starting early, they may be finished before the results are needed. At the place where the compiler needs to use the register just loaded, it inserts a CHECK instruction. If the value is there, CHECK acts like a NOP and execution continues immediately. If the value is not there yet, the next instruction must stall. If an exception occurred and the poison bit is on, the pending exception occurs at that point.

In summary, a machine implementing the IA-64 architecture gets its speed from several different sources. At the core is a state-of-the-art pipelined, load/store, three-address RISC engine. That is already a big improvement over the overly complex IA-32 architecture.

In addition, the IA-64 has a model of explicit parallelism that requires the compiler to figure out which instructions can be executed at the same time without conflicts and group them together in bundles. In this way the CPU can just blindly schedule a bundle without having to do any heavy-duty thinking. Moving work from run time to compile time is always a win.

Next, predication allows the statements in both branches of an if statement to be merged together in a single stream, eliminating the conditional branch and thus the prediction of which way it will go. Finally, speculative LOADs make it possible to fetch operands in advance, without penalty if it turns out later that they are not needed after all.

All in all, the Itanium architecture is an impressive design that appears to better serve architects and users. So, are you running an Itanium processor in your computer, are we running one in ours, is your mom running one, do you know someone that is running one? Answer: no, no, no, and (probably) no. More than a decade after its introduction, its adoption can be described politely as lackluster. But Intel is still committed to producing Itanium-based systems, although they are limited to high-end servers.

So let's bring it back to the original challenges that motivated the creation of IA-64. Itanium was designed to solve the many deficiencies in the IA-32 architecture. Given that it was not widely adopted, how did Intel address these deficiencies? As we will see in Chap. 8, the key to marching the IA-32 line forward was not in retooling the ISA, but rather in embracing parallel computing, through chip multiprocessor designs. For more information about the Itanium 2 and its microarchitecture, see McNairy and Soltis (2003) and Rusu et al. (2004).

5.9 SUMMARY

The instruction set architecture level is what most people think of as "machine language" although on CISC machines it is generally built on a lower layer of microcode. At this level the machine has a byte- or word-oriented memory consisting of some number of megabytes or gigabytes, and instructions such as MOVE, ADD, and BEQ.

Most modern computers have a memory that is organized as a sequence of bytes, with 4 or 8 bytes grouped together into words. There are normally also between 8 and 32 registers present, each one containing one word. On some machines (e.g., Core i7), references to words in memory do not have to be aligned on natural boundaries in memory, while on others (e.g., OMAP4430 ARM), they must be. But even if words do not have to be aligned, performance is better if they are.

Instructions generally have one, two, or three operands, which are addressed using immediate, direct, register, indexed, or other addressing modes. Some machines have a large number of complex addressing modes. In many cases, compilers are unable to use them in a effective way, so they are unused. Instructions are generally available for moving data, dyadic and monadic operations, including arithmetic and Boolean operations, branches, procedure calls, and loops, and sometimes for I/O. Typical instructions move a word from memory to a register (or vice versa), add, subtract, multiply, or divide two registers or a register and a memory word, or compare two items in registers or memory. It is not unusual for a computer to have well over 200 instructions in its repertoire. CISC machines often have many more.

Control flow at level 2 is achieved using a variety of primitives, including branches, procedure calls, coroutine calls, traps, and interrupts. Branches are used to terminate one instruction sequence and begin a new one at a (possibly distant) location in memory. Procedures are used as an abstraction mechanism, to allow a part of the program to be isolated as a unit and called from multiple places. Abstraction using procedures in one form or another is the basis of all modern programming. Without procedures or the equivalent, it would be impossible to write any modern software. Coroutines allow two threads of control to work simultaneously. Traps are used to signal exceptional situations, such as arithmetic overflow. Interrupts allow I/O to take place in parallel with the main computation, with the CPU getting a signal as soon as the I/O has been completed.

The Towers of Hanoi is a fun little problem with a nice recursive solution that we examined. Iterative solutions to it have been found, but they are far more complicated and less elegant than the recursive one we studied.

Last, the IA-64 architecture uses the EPIC model of computing to make it easy for programs to exploit parallelism. It uses instruction groups, predication, and speculative LOADs to gain speed. All in all, it may represent a significant advance over the Core i7, but it puts much of the burden of parallelization on the compiler. Still, doing work at compile time is always better than doing it at run time.

PROBLEMS

1. A word on a little-endian computer with 32-bit words has the numerical value of 3. If it is transmitted to a big-endian computer byte by byte and stored there, with byte 0 in byte 0, byte 1 in byte 1, and so forth, what is its numerical value on the big endian machine if read as a 32-bit integer?

2. Various computers and operating systems in the past have used separate instruction and data spaces, allowing up to 2^k program addresses and also 2^k data addresses using a k-bit address. For example, for $k = 32$, a program could access 4 GB of instructions and also 4 GB of data, for a total address space of 8 GB. Since it is impossible for a program to overwrite itself when this scheme is in use, how could the operating system load programs into memory?

3. Design an expanding opcode to allow all the following to be encoded in a 32-bit instruction:

15 instructions with two 12-bit addresses and one 4-bit register number
650 instructions with one 12-bit address and one 4-bit register number
80 instructions with no addresses or registers

4. A certain machine has 16-bit instructions and 6-bit addresses. Some instructions have one address and others have two. If there are n two-address instructions, what is the maximum number of one-address instructions?

5. Is it possible to design an expanding opcode to allow the following to be encoded in a 12-bit instruction? A register is 3 bits.

4 instructions with three registers
255 instructions with one register
16 instructions with zero registers

6. Given the memory values below and a one-address machine with an accumulator, what values do the following instructions load into the accumulator?

word 20 contains 40
word 30 contains 50
word 40 contains 60
word 50 contains 70

a. LOAD IMMEDIATE 20
b. LOAD DIRECT 20
c. LOAD INDIRECT 20
d. LOAD IMMEDIATE 30
e. LOAD DIRECT 30
f. LOAD INDIRECT 30

7. Compare 0-, 1-, 2-, and 3-address machines by writing programs to compute

$$X = (A + B \times C) / (D - E \times F)$$

for each of the four machines. The instructions available for use are as follows:

0 Address	1 Address	2 Address	3 Address
PUSH M	LOAD M	MOV (X = Y)	MOV (X = Y)
POP M	STORE M	ADD (X = X+Y)	ADD (X = Y+Z)
ADD	ADD M	SUB (X = X−Y)	SUB (X = Y−Z)
SUB	SUB M	MUL (X = X*Y)	MUL (X = Y*Z)
MUL	MUL M	DIV (X = X/Y)	DIV (X = Y/Z)
DIV	DIV M		

M is a 16-bit memory address, and *X*, *Y*, and *Z* are either 16-bit addresses or 4-bit registers. The 0-address machine uses a stack, the 1-address machine uses an accumulator, and the other two have 16 registers and instructions operating on all combinations of memory locations and registers. SUB X,Y subtracts *Y* from *X* and SUB X,Y,Z subtracts Z from *Y* and puts the result in *X*. With 8-bit opcodes and instruction lengths that are multiples of 4 bits, how many bits does each machine need to compute *X*?

8. Devise an addressing mechanism that allows an arbitrary set of 64 addresses, not necessarily contiguous, in a large address space to be specifiable in a 6-bit field.

9. Give a disadvantage of self-modifying code that was not mentioned in the text.

10. Convert the following formulas from infix to reverse Polish notation.

a. A + B + C + D − E
b. (A − B) × (C + D) + E
c. (A × B) + (C × D) − E
d. (A − B) × (((C − D × E) / F) / G) × H

11. Which of the following pairs of reverse Polish notation formulas are mathematically equivalent?

a. A B + C + and A B C + +
b. A B − C − and A B C − −
c. A B × C + and A B C + ×

12. Convert the following reverse Polish notation formulas to infix.

a. A B − C + D ×
b. A B / C D / +
c. A B C D E + × × /
d. A B C D E × F / + G − H / × +

13. Write three reverse Polish notation formulas that cannot be converted to infix.

14. Convert the following infix Boolean formulas to reverse Polish notation.

a. (A AND B) OR C
b. (A OR B) AND (A OR C)
c. (A AND B) OR (C AND D)

15. Convert the following infix formula to reverse Polish notation and generate IJVM code to evaluate it.

(5 × 2 + 7) − (4 / 2 + 1)

16. How many registers does the machine whose instruction formats are given in Fig. 5-24 have?

17. In Fig. 5-24, bit 23 is used to distinguish the use of format 1 from format 2. No bit is provided to distinguish the use of format 3. How does the hardware know to use it?

18. It is common in programming for a program to need to determine where a variable *X* is with respect to the interval *A* to *B*. If a three-address instruction were available with operands *A*, *B*, and *X*, how many condition code bits would have to be set by this instruction?

19. Describe one advantage and one disadvantage of program-counter-relative addressing.

20. The Core i7 has a condition code bit that keeps track of the carry out of bit 3 after an arithmetic operation. What good is it?

21. One of your friends has just come bursting into your room at 3 A.M., out of breath, to tell you about his brilliant new idea: an instruction with two opcodes. Should you send your friend off to the patent office or back to the drawing board?

22. Tests of the form

```
if (k == 0) ...
if (a > b) ...
if (k < 5) ...
```

are common in programming. Devise an instruction to perform these tests efficiently. What fields are present in your instruction?

23. For the 16-bit binary number 1001 0101 1100 0011, show the effect of:

a. A right shift of 4 bits with zero fill.
b. A right shift of 4 bits with sign extension.
c. A left shift of 4 bits.
d. A left rotate of 4 bits.
e. A right rotate of 4 bits.

24. How can you clear a memory word on a machine with no CLR instruction?

25. Compute the Boolean expression (*A* AND *B*) OR *C* for

A = 1101 0000 1010 0011
B = 1111 1111 0000 1111
C = 0000 0000 0010 0000

26. Devise a way to interchange two variables *A* and *B* without using a third variable or register. *Hint*: Think about the EXCLUSIVE OR instruction.

27. On a certain computer it is possible to move a number from one register to another, shift each of them left by different amounts, and add the results in less time than a multiplication takes. Under what condition is this instruction sequence useful for computing "constant × variable"?

28. Different machines have different instruction densities (number of bytes required to perform a certain computation). For the following Java code fragments, translate each

one into Core i7 assembly language and IJVM. Then compute how many bytes each expression requires for each machine. Assume that *i* and *j* are local variables in memory, but otherwise make the most optimistic assumptions in all cases

a. i = 3;
b. i = j;
c. i = j – 1;

29. The loop instructions discussed in the text were for handling for loops. Design an instruction that might be useful for handling common while loops instead.

30. Assume that the monks in Hanoi can move 1 disk per minute (they are in no hurry to finish the job because employment opportunities for people with this particular skill are limited in Hanoi). How long will it take them to solve the entire 64-disk problem? Express your result in years.

31. Why do I/O devices place the interrupt vector on the bus? Would it be possible to store that information in a table in memory instead?

32. A computer uses DMA to read from its disk. The disk has 64 512-byte sectors per track. The disk rotation time is 16 msec. The bus is 16 bits wide, and bus transfers take 500 nsec each. The average CPU instruction requires two bus cycles. How much is the CPU slowed down by DMA?

33. The DMA transfer described in Fig. 5-32 requires 2 bus transfers to move data between an I/O device and memory. Describe how the performance of DMA can be improved by using the bus architecture in Fig. 3-35. 3.

34. Why do interrupt service routines have priorities associated with them whereas normal procedures do not have priorities?

35. The IA-64 architecture contains an unusually large number of registers (64). Was the choice to have so many of them related to the use of predication? If so, in what way? If not, why are there so many?

36. In the text, the concept of speculative LOAD instructions is discussed. However, there is no mention of speculative STORE instructions. Why not? Are they essentially the same as speculative LOAD instructions or is there another reason not to discuss them?

37. When two local area networks are to be connected, a computer called a bridge is inserted between them, connected to both. Each packet transmitted on either network causes an interrupt on the bridge, to let the bridge see if the packet has to be forwarded. Suppose that it takes 250 μsec per packet to handle the interrupt and inspect the packet, but forwarding it, if need be, is done by the DMA hardware without burdening the CPU. If all packets are 1 KB, what is the maximum data rate on each of the networks that can be tolerated without having the bridge lose packets?

38. In Fig. 5-40, the frame pointer points to the first local variable. What information does the program need in order to return from a procedure?

39. Write an assembly language subroutine to convert a signed binary integer to ASCII.

40. Write an assembly language subroutine to convert an infix formula to reverse Polish.

41. The towers of Hanoi is not the only little recursive procedure much loved by computer scientists. Another all-time favorite is $n!$, where $n! = n(n-1)!$ subject to the limiting condition that $0! = 1$. Write a procedure in your favorite assembly language to compute $n!$.

42. If you are not convinced that recursion is at times indispensable, try programming the Towers of Hanoi without using recursion and without simulating the recursive solution by maintaining a stack in an array. Be warned, however, that you will probably not be able to find the solution.

This page intentionally left blank

6

THE OPERATING SYSTEM MACHINE LEVEL

The theme of this book is that a modern computer is built as a series of levels, each one adding functionality to the one below it. So far, we have seen the digital logic level, microarchitecture level, and instruction-set architecture level. Now it is time to move up another level, into the realm of the operating system.

An **operating system** is a program that, from the programmer's point of view, adds a variety of new instructions and features, above and beyond what the ISA level provides. Normally, the operating system is implemented largely in software, but there is no theoretical reason why it could not be put into hardware, just as microprograms normally are (when they are present). For short, we will call the level that it implements the **OSM** (**Operating System Machine**) level. It is shown in Fig. 6-1.

Although the OSM level and the ISA level are both abstract (in the sense that they are not the true hardware level), there is an important difference between them. The OSM-level instruction set is the complete set of instructions available to application programmers. It contains nearly all of the ISA level instructions, as well as the set of new instructions that the operating system adds. These new instructions are called system calls. A **system call** invokes a predefined operating system service, effectively, one of its instructions. A typical system call is reading some data from a file. We will typeset system calls in lowercase Helvetica.

The OSM level is always interpreted. When a user program executes an OSM instruction, such as reading some data from a file, the operating system carries out this instruction step by step, just as a microprogram would carry out an ADD

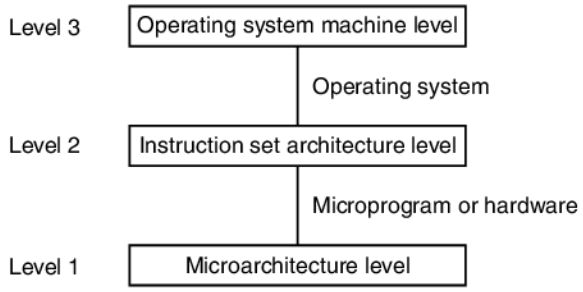

Figure 6-1. Positioning of the operating system machine level.

instruction step by step. However, when a program executes an ISA-level instruction, it is carried out directly by the underlying microarchitecture level, without any assistance from the operating system.

In this book we can provide only the briefest of introductions to the subject of operating systems. We will focus on three topics of importance. The first is virtual memory, a technique provided by many modern operating systems to make the machine appear to have more memory than it in reality has. The second is file I/O, a higher-level concept than the I/O instructions that we studied in the preceding chapter. The third topic is parallel processing—how multiple processes can execute, communicate, and synchronize. The concept of a process is an important one, and we will describe it in detail later in this chapter. For the time being, a process can be thought of as a running program together with all its state information (memory, registers, program counter, I/O status, and so on). After discussing these principles in general, we will show how they apply to the operating systems of two of our example machines, the Core i7 (running Windows 7) and the OMAP4430 ARM CPU (running Linux). Since the ATmega168 microcontroller is normally used for embedded systems, it does not have an operating system.

6.1 VIRTUAL MEMORY

In the early days of computers, memories were small and expensive. The IBM 650, the leading scientific computer of its day (late 1950s), had only 2000 words of memory. One of the first ALGOL 60 compilers was written for a computer with only 1024 words of memory. An early timesharing system ran quite well on a PDP-1 with a total memory size of only 4096 18-bit words for the operating system and user programs combined. In those days the programmer spent a lot of time trying to squeeze programs into the tiny memory. Often it was necessary to use an algorithm that ran a great deal slower than another, better algorithm simply because the better algorithm was too big—that is, a program using the better algorithm could not be squeezed into the computer's memory.

The traditional solution to this problem was the use of secondary memory, such as disk. The programmer divided the program up into a number of pieces, called **overlays**, each of which could fit in the memory. To run the program, the first overlay was brought in and it ran for a while. When it finished, it read in the next overlay and called it, and so on. The programmer was responsible for breaking the program into overlays, deciding where in the secondary memory each overlay was to be kept, arranging for the transport of overlays between main memory and secondary memory, and in general managing the whole overlay process without any help from the computer.

Although widely used for many years, this technique involved much work in connection with overlay management. In 1961 a group of researchers in Manchester, England, proposed a method for performing the overlay process automatically, without the programmer even knowing that it was happening (Fotheringham, 1961). This method, now called **virtual memory**, had the obvious advantage of freeing the programmer from a lot of annoying bookkeeping. It was first used on a number of computers during the 1960s, associated mostly with research projects in computer systems design. By the early 1970s virtual memory had become available on most computers. Now even single-chip computers, including the Core i7 and OMAP4430 ARM CPU, have highly sophisticated virtual memory systems. We will look at these later in this chapter.

6.1.1 Paging

The idea put forth by the Manchester group was to separate the concepts of address space and memory locations. Consider, as an example, a typical computer of that era, which might have had a 16-bit address field in its instructions and 4096 words of memory. A program on this computer could address 65536 words of memory. The reason is that 65536 (2^{16}) 16-bit addresses exist, each corresponding to a different memory word. Please note that the number of addressable words depends only on the number of bits in an address and is in no way related to the number of memory words actually available. The **address space** for this computer consists of the numbers 0, 1, 2, ..., 65535, because that is the set of possible addresses. The computer, however, may well have had fewer than 65535 words of memory.

Before virtual memory was invented, people would have made a distinction between the addresses below 4096 and those equal to or above 4096. Although rarely stated in so many words, these two parts were regarded as the useful address space and the useless address space, respectively (the addresses above 4095 being useless because they did not correspond to actual memory addresses). People did not make a distinction between address space and memory locations, because the hardware enforced a one-to-one correspondence between them.

The idea of separating the address space and the memory locations is as follows. At any instant of time, 4096 words of memory can be directly accessed, but

they need not correspond to memory locations 0 to 4095. We could, for example, "tell" the computer that henceforth whenever address 4096 is referenced, the memory word at address 0 is to be used. Whenever address 4097 is referenced, the memory word at address 1 is to be used; whenever address 8191 is referenced, the memory word at address 4095 is to be used, and so forth. In other words, we have defined a mapping from the address space onto the actual memory locations, as shown in Fig. 6-2.

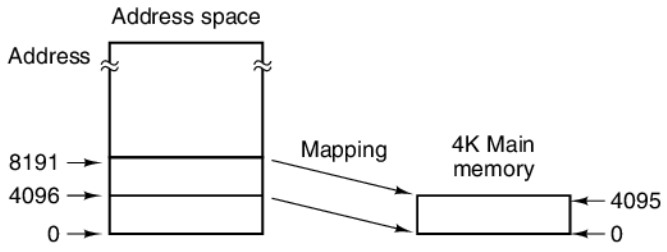

Figure 6-2. A mapping in which virtual addresses 4096 to 8191 are mapped onto main memory addresses 0 to 4095.

In terms of this picture of mapping addresses from the address space onto the actual memory locations, a 4-KB machine without virtual memory simply has a fixed mapping between the addresses 0 to 4095 and the 4096 words of memory. An interesting question is: "What happens if a program branches to an address between 8192 and 12287?" On a machine that lacks virtual memory, the program would cause an error trap that would print a suitably rude message, for example: "Nonexistent memory referenced," and terminate the program. On a machine with virtual memory, the following sequence of steps would occur:

1. The contents of main memory would be saved on disk.
2. Words 8192 to 12287 would be located on disk.
3. Words 8192 to 12287 would be loaded into main memory.
4. The address map would be changed to map addresses 8192 to 12287 onto memory locations 0 to 4095.
5. Execution would continue as though nothing unusual had happened.

This technique for automatic overlaying is called **paging** and the chunks of program read in from disk are called **pages**.

A more sophisticated way of mapping addresses from the address space onto the actual memory locations is certainly possible. To avoid confusion, we will call the addresses that the program can refer to the **virtual address space**, and the actual, hardwired (physical) memory locations the **physical address space**. A **memory map** or **page table** specifies for each virtual address what the corresponding

physical address is. We presume that there is enough room on disk to store the entire virtual address space (or at least that portion of it that is being used).

Programs are written just as though there were enough main memory for the whole virtual address space, even though that is not the case. Programs may load from, or store into, any word in the virtual address space, or branch to any instruction located anywhere within the virtual address space, without regard to the fact that there really is not enough physical memory. In fact, the programmer can write programs without even being aware that virtual memory exists. The computer just looks as if it has a big memory.

This point is crucial and will be contrasted later with segmentation, where the programmer must be aware of the existence of segments. To emphasize it once more, paging gives the programmer the illusion of a large, continuous, linear main memory, the same size as the virtual address space. In reality, the main memory available may be smaller (or larger) than the virtual address space. The simulation of this large main memory by paging cannot be detected by the program (except by running timing tests). Whenever an address is referenced, the proper instruction or data word appears to be present. Because the programmer can program as though paging did not exist, the paging mechanism is said to be **transparent**.

The idea that a programmer may use some nonexistent feature without being concerned with how it works is not new to us, after all. The ISA-level instruction set often includes a MUL instruction, even though the underlying microarchitecture does not have a multiplication device in the hardware. The illusion that the machine can multiply is typically sustained by microcode. Similarly, the virtual machine provided by the operating system can provide the illusion that all the virtual addresses are backed up by real memory, even though this is not true. Only operating system writers (and students of operating systems) have to know how the illusion is supported.

6.1.2 Implementation of Paging

One essential requirement for a virtual memory is a disk on which to keep the whole program and all the data. The disk could be a rotating hard disk or a solid-state disk. Throughout the rest of this book we will refer to "disk" or "hard disk" for simplicity, but understand that this includes solid-state disks as well. It is conceptually simpler if one thinks of the copy of the program on the disk as the original one and the pieces brought into main memory every now and then as copies rather than the other way around. Naturally, it is important to keep the original up to date. When changes are made to the copy in main memory, they should also be reflected in the original (eventually).

The virtual address space is broken up into a number of equal-sized pages. Page sizes ranging from 512 to 64 KB per page are common at present, although sizes as large as 4 MB are used occasionally. The page size is always a power of 2, for example, 2^k, so that all the addresses can be represented in k bits. The physical

address space is broken up into pieces in a similar way, each piece being the same size as a page, so that each piece of main memory is capable of holding exactly one page. These pieces of main memory into which the pages go are called **page frames**. In Fig. 6-2 the main memory contains only one page frame. In practical designs it will usually contain thousands of them.

Figure 6-3(a) illustrates one possible way to divide up the first 64 KB of a virtual address space—in 4-KB pages. (Note that we are talking about 64 KB and 4K of addresses here. An address might be a byte but could equally well be a word on a computer in which consecutive words had consecutive addresses.) The virtual memory of Fig. 6-3 would be implemented by means of a page table with as many entries as there are pages in the virtual address space. For simplicity, we have shown only the first 16 entries here. When the program tries to reference a word in the first 64 KB of its virtual address space, whether to fetch instructions, fetch data, or store data, it first generates a virtual address between 0 and 65532 (assuming that word addresses must be divisible by 4). Indexing, indirect addressing, and all the usual techniques may be used to generate this address.

Figure 6-3(b) shows a physical memory consisting of eight 4-KB page frames. This memory might be limited to 32 KB because (1) that is all the machine had (a processor embedded in a washing machine or microwave oven might not need more), or (2) the rest of the memory was allocated to other programs.

Now consider how a 32-bit virtual address can be mapped onto a physical main-memory address. After all, the only thing the memory understands are main memory addresses, not virtual addresses, so that is what it must be given. Every computer with virtual memory has a device for doing the virtual-to-physical mapping. This device is called the **MMU** (**Memory Management Unit**). It may be on the CPU chip, or it may be on a separate chip that works closely with the CPU chip. Since our sample MMU maps from a 32-bit virtual address to a 15-bit physical address, it needs a 32-bit input register and a 15-bit output register.

To see how the MMU works, consider the example of Fig. 6-4. When the MMU is presented with a 32-bit virtual address, it separates the address into a 20-bit virtual page number and a 12-bit offset within the page (because the pages in our example are 4K). The virtual page number is used as an index into the page table to find the entry for the page referenced. In Fig. 6-4, the virtual page number is 3, so entry 3 of the page table is selected, as shown.

The first thing the MMU does with the page-table entry is check to see if the page referenced is currently in main memory. After all, with 2^{20} virtual pages and only eight page frames, not all virtual pages can be in memory at once. The MMU makes this check by examining the **present/absent bit** in the page-table entry. In our example, the bit is 1, meaning the page is currently in memory.

The next step is to take the page-frame value from the selected entry (6 in this case) and copy it into the upper 3 bits of the 15-bit output register. Three bits are needed because there are eight page frames in physical memory. In parallel with this operation, the low-order 12 bits of the virtual address (the page-offset field) are

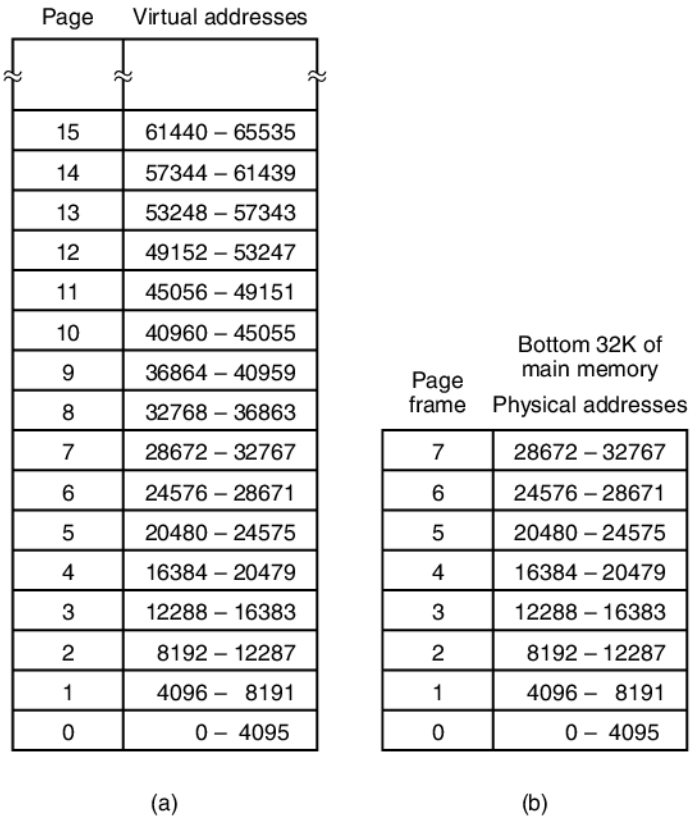

Figure 6-3. (a) The first 64 KB of virtual address space divided into 16 pages, with each page being 4K. (b) A 32-KB main memory divided up into eight page frames of 4 KB each.

copied into the low-order 12 bits of the output register, as shown. This 15-bit address is now sent to the cache or memory for lookup.

Figure 6-5 shows a possible mapping between virtual pages and physical page frames. Virtual page 0 is in page frame 1. Virtual page 1 is in page frame 0. Virtual page 2 is not in main memory. Virtual page 3 is in page frame 2. Virtual page 4 is not in main memory. Virtual page 5 is in page frame 6, and so on.

6.1.3 Demand Paging and the Working-Set Model

In the preceding discussion it was assumed that the virtual page referenced was in main memory. However, that assumption will not always be true because there is not enough room in main memory for all the virtual pages. When a reference is made to an address on a page not present in main memory, it is called a **page fault**.

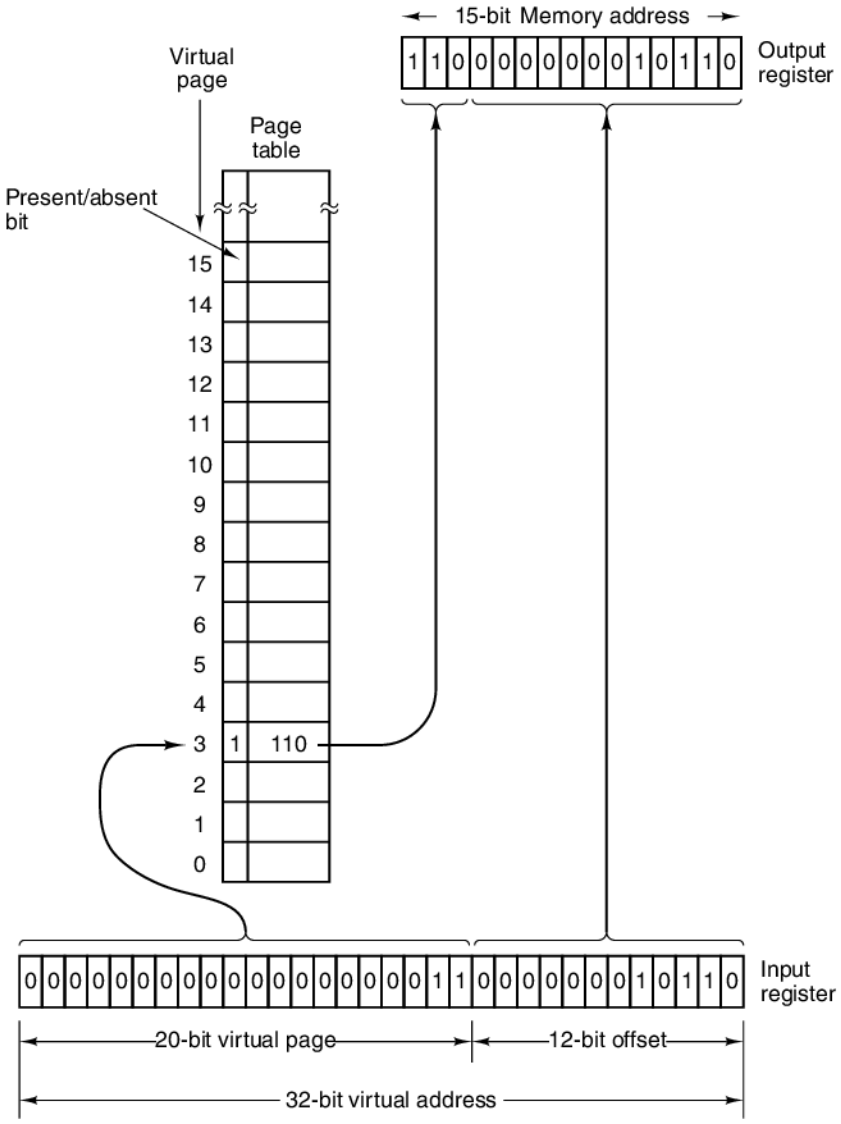

Figure 6-4. Formation of a main memory address from a virtual address.

After a page fault has occurred, the operating system must read in the required page from the disk, enter its new physical memory location in the page table, and then repeat the instruction that caused the fault.

It is possible to start a program running on a machine with virtual memory even when none of the program is in main memory. The page table merely has to be set to indicate that each and every virtual page is in the secondary memory and

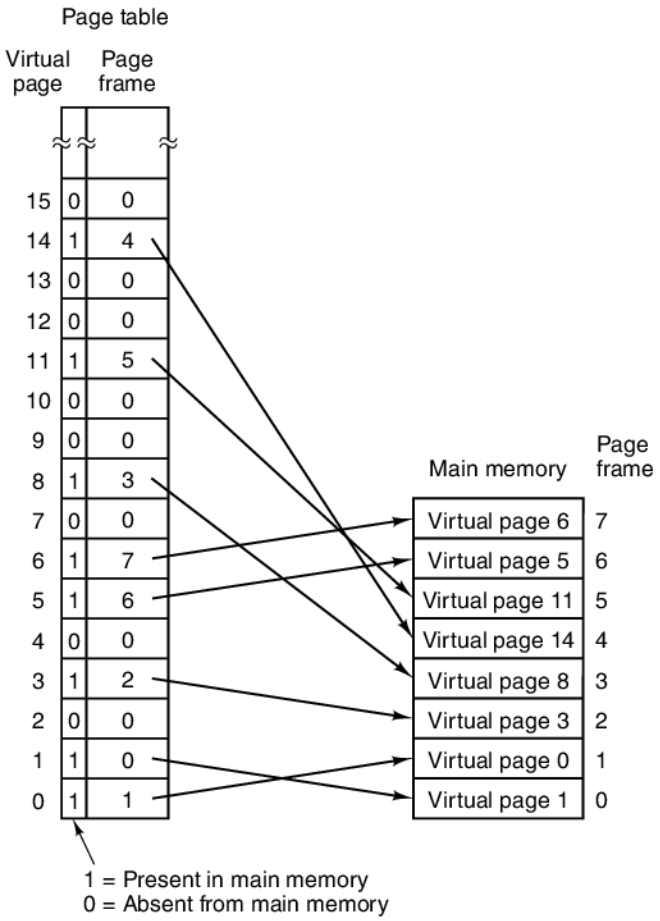

Figure 6-5. A possible mapping of the first 16 virtual pages onto a main memory with eight page frames.

not in main memory. When the CPU tries to fetch the first instruction, it immediately gets a page fault, which causes the page containing the first instruction to be loaded into memory and entered in the page table. Then the first instruction can begin. If the first instruction has two addresses, with the two addresses on different pages, both different from the instruction page, two more page faults will occur, and two more pages will be brought in before the instruction can finally execute. The next instruction may cause some more page faults, and so on.

This method of operating a virtual memory is called **demand paging**, in analogy to the well-known demand feeding algorithm for babies: when the baby cries, you feed it (as opposed to feeding it on a schedule). In demand paging, a page is brought into memory only when a request for it occurs, not in advance.

The question of whether demand paging should be used or not is relevant only when a program first starts up. Once it has been running for a while, the needed pages will already have been collected in main memory. If, however, the computer is timeshared and processes are swapped out after running 100 msec or thereabouts, each program will be restarted many times during the course of its run. Because the memory map is unique to each program, and is changed when programs are switched, for example, in a timesharing system, the question repeatedly becomes a critical one.

The alternative approach is based on the observation that most programs do not reference their address space uniformly but that references tend to cluster on a small number of pages. This concept is called the **locality principle**. A memory reference may fetch an instruction, it may fetch data, or it may store data. At any instant in time, *t*, there exists a set consisting of all the pages used by the *k* most recent memory references. Denning (1968) has called this the **working set**.

Because the working set normally varies slowly with time, it is possible to make a reasonable guess as to which pages will be needed when the program is restarted, on the basis of its working set when it was last stopped. These pages could then be loaded in advance before starting the program up (assuming they fit).

6.1.4 Page-Replacement Policy

Ideally, the set of pages that a program is actively and heavily using, called the **working set**, can be kept in memory to reduce page faults. However, programmers rarely know which pages are in the working set, so the operating system must discover this set dynamically. When a program references a page that is not in main memory, the needed page must be fetched from the disk. To make room for it, however, some other page will generally have to be sent back to the disk. Thus an algorithm that decides which page to remove is needed.

Choosing a page to remove at random is probably not a good idea. If the page containing the faulting instruction should happen to be the one picked, another page fault will occur as soon as an attempt is made to fetch the next instruction. Most operating systems try to predict which of the pages in memory is the least useful in the sense that its absence would have the smallest adverse effect on the running program. One way of doing so is to make a prediction when the next reference to each page will occur and remove the page whose predicted next reference lies furthest in the future. In other words, rather than evict a page that will be needed shortly, try to select one that will not be needed for a long time.

One popular algorithm evicts the page least recently used because the a priori probability of its not being in the current working set is high. It is called the **LRU** (**Least Recently Used**) algorithm. Although it usually performs well, there are pathological situations, such as the one described below, where it fails miserably.

Imagine a program that is executing a large loop that extends over nine virtual pages on a machine with room for only eight pages in physical memory. After the

program gets to page 7, the main memory will be as shown in Fig. 6-6(a). An attempt is eventually made to fetch an instruction from virtual page 8, which causes a page fault. A decision has to be made about which page to evict. The LRU algorithm will choose virtual page 0, because it has been used least recently. Virtual page 0 is removed and virtual page 8 is brought in to replace it, giving the situation in Fig. 6-6(b).

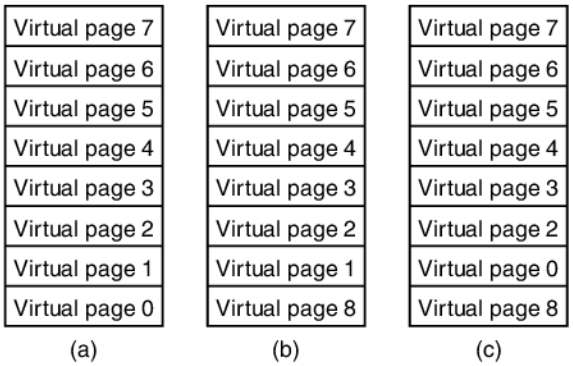

Figure 6-6. Failure of the LRU algorithm.

After executing the instructions on virtual page 8, the program branches back to the top of the loop, to virtual page 0. This step causes another page fault. Virtual page 0, which was just thrown out, has to be brought back in. The LRU algorithm chooses page 1 to be thrown out, producing the situation in Fig. 6-6(c). The program continues on page 0 for a little while. Then it tries to fetch an instruction from virtual page 1, causing a page fault. Page 1 has to be brought back in again and page 2 will be thrown out.

It should be apparent by now that here the LRU algorithm is consistently making the worst choice every time (other algorithms also fail under similar conditions). If, however, the available main memory exceeds the size of the working set, LRU tends to minimize the number of page faults.

Another page-replacement algorithm is **FIFO** (**First-In First-Out**). FIFO removes the least recently loaded page, independent of when this page was last referenced. Associated with each page frame is a counter. Initially, all the counters are set to 0. After each page fault has been handled, the counter for each page presently in memory is increased by one, and the counter for the page just brought in is set to 0. When it becomes necessary to choose a page to remove, the page whose counter is highest is chosen. Since its counter is the highest, it has witnessed the largest number of page faults. This means that it was loaded prior to the loading of any of the other pages in memory and therefore (hopefully) has a large a priori chance of no longer being needed.

If the working set is larger than the number of available page frames, no algorithm that is not an oracle will give good results, and page faults will be frequent.

A program that generates page faults frequently and continuously is said to be **thrashing**. Needless to say, thrashing is an undesirable characteristic to have in your system. If a program uses a large amount of virtual address space but has a small, slowly changing working set that fits in available main memory, it will give little trouble. This observation is true, even if, over its lifetime, the program uses hundreds of times as many words of virtual memory as the machine has words of main memory.

If a page about to be evicted has not been modified since it was read in (a likely occurrence if the page contains program rather than data), it is not necessary to write it back onto disk, because an accurate copy already exists there. If it has been modified since it was read in, the copy on the disk is no longer accurate, and the page must be rewritten.

If there is a way to tell whether a page has not changed since it was read in (page is clean) or whether it, in fact, has been stored into (page is dirty), all the rewriting of clean pages can be avoided, thus saving a lot of time. Many computers have 1 bit per page, in the MMU, which is set to 0 when a page is loaded and set to 1 by the microprogram or hardware whenever it is stored into (i.e., is made dirty). By examining this bit, the operating system can find out if the page is clean or dirty, and hence whether it need be rewritten or not.

6.1.5 Page Size and Fragmentation

If the user's program and data accidentally happen to fill an integral number of pages exactly, there will be no wasted space when they are in memory. Otherwise, there will be some unused space on the last page. For example, if the program and data need 26,000 bytes on a machine with 4096 bytes per page, the first six pages will be full, totaling $6 \times 4096 = 24{,}576$ bytes, and the last page will contain $26{,}000 - 24576 = 1424$ bytes. Since there is room for 4096 bytes per page, 2672 bytes will be wasted. Whenever the seventh page is present in memory, those bytes will occupy main memory but will serve no useful function. The problem of these wasted bytes is called **internal fragmentation** (because the wasted space is internal to some page).

If the page size is n bytes, the average amount of space wasted in the last page of a program by internal fragmentation will be $n/2$ bytes—a situation that suggests using a small page size to minimize waste. On the other hand, a small page size means many pages, as well as a large page table. If the page table is maintained in hardware, a large page table means that more registers are needed to store it, which increases the cost of the computer. In addition, more time will be required to load and save these registers whenever a program is started or stopped.

Furthermore, small pages make inefficient use of disk bandwidth. Given that one is going to wait 10 msec or so before the transfer can begin (seek + rotational delay), large transfers are more efficient than small ones. With a 100-MB/sec transfer rate, transferring 8 KB adds only 70 μsec compared to transferring 1 KB.

However, small pages also have the advantage that if the working set consists of a large number of small, separated regions in the virtual address space, there may be less thrashing with a small page size than with a big one. For example, consider a 10,000 × 10,000 matrix, A, stored with $A[1,1]$, $A[2,1]$, $A[3,1]$, and so on, in consecutive 8-byte words. This column-ordered storage means that the elements of row 1, $A[1,1]$, $A[1,2]$, $A[1,3]$, and so on, will begin 80,000 bytes apart. A program performing an extensive calculation on all the elements of this row would use 10,000 regions, each separated from the next one by 79,992 bytes. If the page size were 8 KB, a total storage of 80 MB would be needed to hold all the pages being used.

On the other hand, a page size of 1 KB would require only 10 MB of RAM to hold all the pages. If the available memory were 32 MB, with an 8-KB page size, the program would thrash, but with a 1-KB page size it would not. All things considered, the trend is toward larger page sizes. In practice, 4 KB is the minimum these days.

6.1.6 Segmentation

The virtual memory discussed above is one-dimensional because the virtual addresses go from 0 to some maximum address, one address after another. For many problems, having two or more separate virtual address spaces may be much better than having only one. For example, a compiler might have many tables that are built up as compilation proceeds, including

1. The symbol table, containing the names and attributes of variables.
2. The source text being saved for the printed listing.
3. A table containing all the integer and floating-point constants used.
4. The parse tree, containing the syntactic analysis of the program.
5. The stack used for procedure calls within the compiler.

Each of the first four tables grows continuously as compilation proceeds. The last one grows and shrinks in unpredictable ways during compilation. In a one-dimensional memory, these five tables would have to be allocated as contiguous chunks of virtual address space, as in Fig. 6-7.

Consider what happens if a program has an exceptionally large number of variables. The chunk of address space allocated for the symbol table may fill up, even if there is lots of room in the other tables. The compiler could, of course, simply issue a message saying that the compilation cannot continue due to too many variables, but doing so does not seem very sporting when unused space is left in the other tables.

Another possibility is to have the compiler play Robin Hood, taking space from the tables with much room and giving it to the tables with little room. This

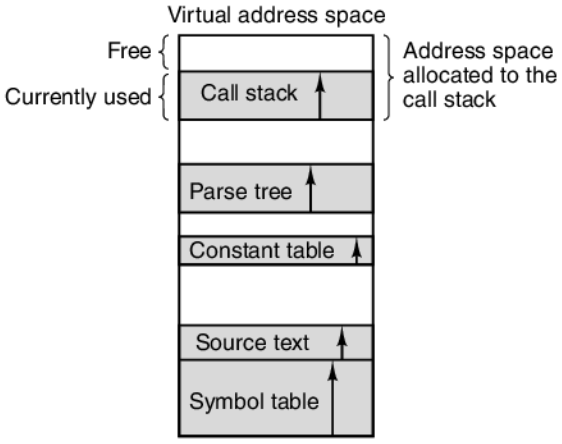

Figure 6-7. In a one-dimensional address space with growing tables, one table may bump into another.

shuffling can be done, but it is analogous to managing one's own overlays—a nuisance at best and a great deal of tedious, unrewarding work at worst.

What is really needed is a way of freeing the programmer from having to manage the expanding and contracting tables, in the same way that virtual memory eliminates the worry of organizing the program into overlays.

A straightforward solution is to provide many completely independent address spaces, called **segments**. Each segment consists of a linear sequence of addresses, from 0 to some maximum. The length of each segment may be anything from 0 to the maximum allowed. Different segments may, and usually do, have different lengths. Moreover, segment lengths may change during execution. The length of a stack segment may be increased whenever something is pushed onto the stack and decreased whenever something is popped off the stack.

Because each segment constitutes a separate address space, different segments can grow or shrink independently, without affecting each other. If a stack in a certain segment needs more address space to grow, it can have it, because there is nothing else in its address space to bump into. Of course, a segment can fill up completely but segments are usually very large, so this occurrence is rare. To specify an address in this segmented or two-dimensional memory, the program must supply a two-part address: a segment number, and an address within the segment. Figure 6-8 illustrates a segmented memory being used for the compiler tables discussed earlier.

We emphasize that a segment is a *logical* entity, which the programmer is aware of and uses as a single logical entity. A segment might contain a procedure, or an array, or a stack, or a collection of scalar variables, but usually it does not contain a mixture of different types.

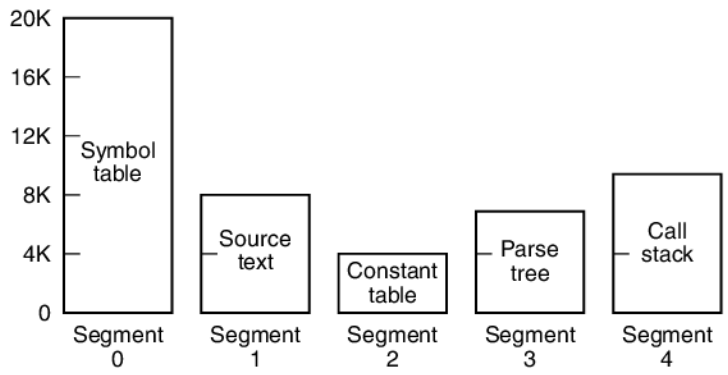

Figure 6-8. A segmented memory allows each table to grow or shrink independently of the other tables.

A segmented memory has other advantages besides simplifying the handling of data structures that are growing or shrinking. If each procedure occupies a separate segment, with address 0 as its starting address, the linking up of procedures compiled separately is greatly simplified. After all the procedures that constitute a program have been compiled and linked up, a procedure call to the procedure in segment n will use the two-part address (n, 0) to address word 0 (the entry point).

If the procedure in segment n is subsequently modified and recompiled, no other procedures need be changed (because no starting addresses have been modified), even if the new version is larger than the old one. With a one-dimensional memory, the procedures are normally packed tightly next to each other, with no address space between them. Consequently, changing one procedure's size can affect the starting address of other, unrelated, procedures. This, in turn, requires modifying all procedures that call any of the moved procedures, in order to incorporate their new starting addresses. If a program contains hundreds of procedures, this process can be costly.

Segmentation also facilitates sharing procedures or data among several programs. If a computer has several programs running in parallel (either true or simulated parallel processing), all of which use certain library procedures, it is wasteful of main memory to provide each one with its own private copy. If we make each procedure a separate segment, they can be shared easily, thus eliminating the need for more than one physical copy of any shared procedure to be in main memory. As a result, memory is saved.

Because each segment forms a logical entity of which the programmer is aware, such as a procedure, or an array, or a stack, different segments can have different kinds of protection. A procedure segment could be specified as execute only, prohibiting attempts to read from it or store into it. A floating-point array could be specified as read/write but not execute, and attempts to branch to it would be caught. Such protection is frequently helpful in catching programming errors.

Try to understand why protection makes sense in a segmented memory but not in a one-dimensional (i.e., linear) paged memory. In a segmented memory the user is aware of what is in each segment. Normally, a segment would not contain both a procedure and a stack, for example, but one or the other. Since each segment contains only one type of object, the segment can have the protection appropriate for that particular type. Paging and segmentation are compared in Fig. 6-9.

Consideration	Paging	Segmentation
Need the programmer be aware of it?	No	Yes
How many linear addresses spaces are there?	1	Many
Can virtual address space exceed memory size?	Yes	Yes
Can variable-sized tables be handled easily?	No	Yes
Why was the technique invented?	To simulate large memories	To provide multiple address spaces

Figure 6-9. Comparison of paging and segmentation.

The contents of a page are, in a sense, accidental. The programmer is unaware that paging is even occurring. Although putting a few bits in each entry of the page table to specify the access allowed would be possible, to utilize this feature the programmer would have to keep track of where in his address space the page boundaries were. The trouble with this idea is that this is precisely the sort of administration that paging was invented to eliminate. Because the user of a segmented memory has the illusion that all segments are in main memory all the time, they can be addressed without having to be concerned with the administration of overlaying them.

6.1.7 Implementation of Segmentation

Segmentation can be implemented in one of two ways: swapping and paging. In the former scheme, some set of segments is in memory at a given instant. If a reference is made to a segment not currently in memory, that segment is brought in. If there is no room for it, one or more segments must be written to disk first (unless a clean copy already exists there, in which case the memory copy can just be abandoned). In a certain sense, segment swapping is not unlike demand paging: segments come and segments go as needed.

However, the implementation of segmentation differs from paging in a very essential way: pages are of fixed size and segments are not. Figure 6-10(a) shows an example of physical memory initially containing five segments. Now consider what happens if segment 1 is evicted and segment 7, which is smaller, is put in its place. We arrive at the memory configuration of Fig. 6-10(b). Between segment 7 and segment 2 is an unused area—that is, a hole. Then segment 4 is replaced by segment 5, as shown in Fig. 6-10(c), and segment 3 is replaced by segment 6, as in

Fig. 6-10(d). After the system has been running for a while, memory will be divided up into a number of chunks, some containing segments and some containing holes. This phenomenon is called **external fragmentation** (because space is wasted external to the segments, in the holes between them). Sometimes external fragmentation is called **checkerboarding**.

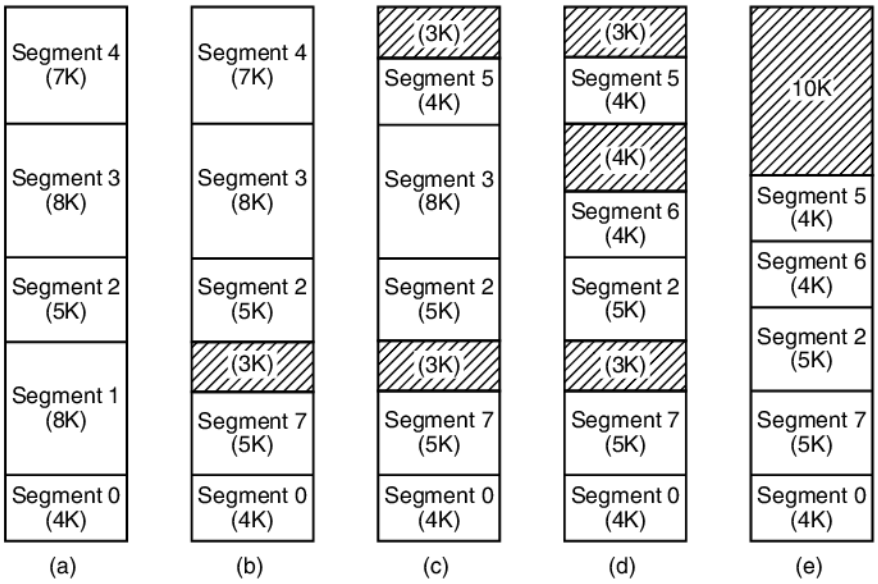

Figure 6-10. (a)–(d) Development of external fragmentation. (e) Removal of the external fragmentation by compaction.

Consider what would happen if the program referenced segment 3 at the time memory was suffering from external fragmentation, as in Fig. 6-10(d). The total space in the holes is 10K, more than enough for segment 3, but because the space is distributed in small, useless pieces, segment 3 cannot simply be loaded. Instead, another segment must be removed first.

One way to avoid external fragmentation is as follows: every time a hole appears, move the segments following the hole closer to memory location 0, thereby eliminating that hole but leaving a big hole at the end. Alternatively, one could wait until the external fragmentation became quite serious (e.g., more than a certain percentage of the total memory wasted in holes) before performing the compaction (squeezing out the holes). Figure 6-10(e) shows how the memory of Fig. 6-10(d) would look after compaction. The intention of compacting memory is to collect all the small useless holes into one big hole, into which one or more segments can be placed. Compacting has the obvious drawback that some time is wasted doing the compacting. Compacting after every hole is created is usually too time consuming.

If the time required for compacting memory is unacceptably large, an algorithm is needed to determine which hole to use for a particular segment. Hole management requires maintaining a list of the addresses and sizes of all holes. One popular algorithm, called **best fit**, chooses the smallest hole into which the needed segment will fit. The idea is to match holes and segments so as to avoid breaking off a piece of a big hole, which may be needed later for a big segment.

Another popular algorithm, called **first fit**, circularly scans the hole list and chooses the first hole big enough for the segment to fit into. Doing so obviously takes less time than checking the entire list to find the best fit. Surprisingly, first fit is also a better algorithm in terms of overall performance than best fit, because the latter tends to generate a great many small, totally useless holes (Knuth, 1997).

First fit and best fit tend to decrease the average hole size. Whenever a segment is placed in a hole bigger than itself, which happens almost every time (exact fits are rare), the hole is divided into two parts. One part is occupied by the segment and the other part is the new hole. The new hole is always smaller than the old hole. Unless there is a compensating process re-creating big holes out of small ones, both first fit and best fit will eventually fill memory with small useless holes.

One such compensating process is the following one. Whenever a segment is removed from memory and one or both of its nearest neighbors are holes rather than segments, the adjacent holes can be coalesced into one big hole. If segment 5 were removed from Fig. 6-10(d), the two surrounding holes and the 4 KB used by the segment would be merged into a single 11-KB hole.

At the beginning of this section, we stated that there are two ways to implement segmentation: swapping and paging. The discussion so far has centered on swapping. In this scheme, whole segments are shuttled back and forth between memory and disk on demand. The other way to implement segmentation is by dividing each segment up into fixed-size pages and demand paging them. In this scheme, some of the pages of a segment may be in memory and some may be on disk. To page a segment, a separate page table is needed for each segment. Since a segment is just a linear address space, all the techniques we have seen so far for paging apply to each segment. The only new feature here is that each segment gets its own page table.

An early operating system that combined segmentation with paging was **MULTICS** (**MULTiplexed Information and Computing Service**), initially a joint effort of M.I.T., Bell Labs, and General Electric (Corbató and Vyssotsky, 1965; and Organick, 1972). MULTICS addresses had two parts: a segment number and an address within the segment. There was a descriptor segment for each process, which contained a descriptor for each segment. When a virtual address was presented to the hardware, the segment number was used as an index into the descriptor segment to locate the descriptor for the segment being accessed, as shown in Fig. 6-11. The descriptor pointed to the page table, allowing each segment to be paged in the usual way. To speed up performance, the most recently used segment/page combinations were held in a 16-entry hardware **associative memory**

that allowed them to be looked up quickly. Although MULTICS is gone now, it had a very long run, from 1965 to Oct. 30, 2000, when the last MULTICS system was shut down. Few other operating systems have lasted 35 years. Futhermore, its spirit lives on because the virtual memory of every Intel CPU since the 386 has been closely modeled on it. History and other aspects of MULTICS are described at *www.multicians.org*.

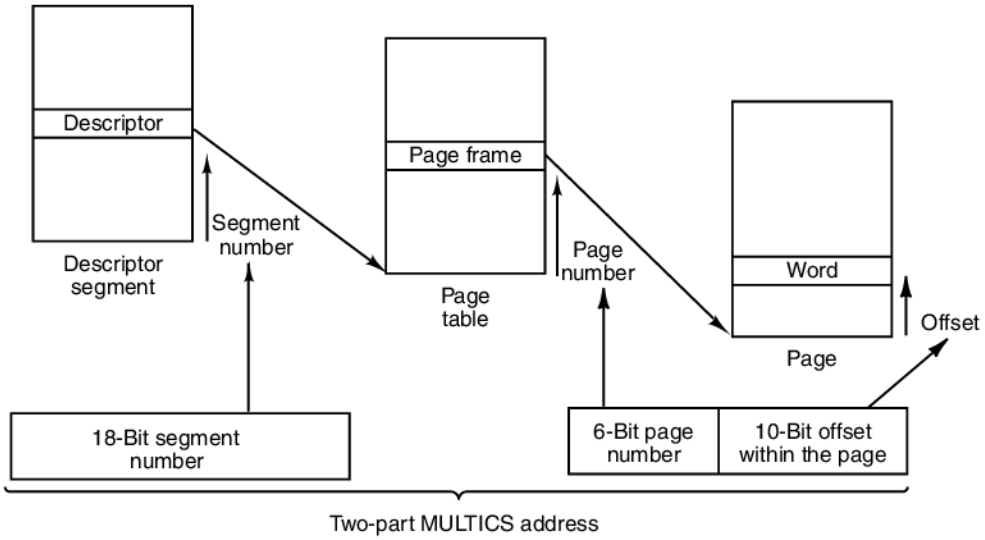

Figure 6-11. Conversion of a two-part MULTICS address into a main memory address.

6.1.8 Virtual Memory on the Core i7

The Core i7 has a sophisticated virtual memory system that supports demand paging, pure segmentation, and segmentation with paging. The heart of the Core i7 virtual memory consists of two tables: the **LDT** (**Local Descriptor Table**) and the **GDT** (**Global Descriptor Table**). Each program has its own LDT, but a single GDT is shared by all the programs on the computer. The LDT describes segments local to each program, including its code, data, stack, and so on, whereas the GDT describes system segments, including the operating system itself.

As we described in Chap. 5, to access a segment, a Core i7 program first loads a selector for that segment into one of the segment registers. During execution, CS holds the selector for the code segment, DS holds the selector for the data segment, and so on. Each selector is a 16-bit number, as shown in Fig. 6-12.

One of the selector bits tells whether the segment is local or global (i.e., whether it is held in the LDT or GDT). Thirteen other bits specify the LDT or GDT entry number, so these tables are each restricted to holding 8 KB (2^{13}) descriptors

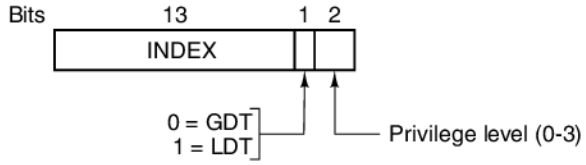

Figure 6-12. A Core i7 selector.

for segments. The other 2 bits relate to protection and will be described later. Descriptor 0 is invalid and causes a trap if used. It may be safely loaded into a segment register to indicate that the segment register is not currently available, but it causes a trap if used.

At the time a selector is loaded into a segment register, the corresponding descriptor is fetched from the LDT or GDT and stored in internal MMU registers, so it can be accessed quickly. A descriptor consists of 8 bytes, including the segment's base address, size, and other information, as depicted in Fig. 6-13.

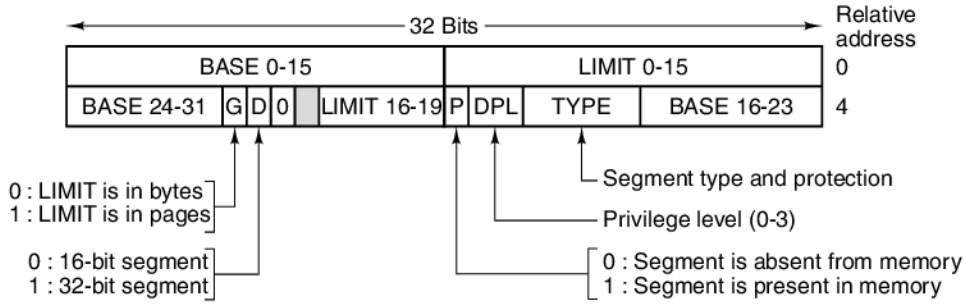

Figure 6-13. A Core i7 code segment descriptor. Data segments differ slightly.

The format of the selector has been cleverly chosen to make locating the descriptor easy. First either the LDT or GDT is selected, based on selector bit 2. Then the selector is copied to an MMU scratch register, and the 3 low-order bits are set to 0, effectively multiplying the 13-bit selector number by eight. Finally, the address of either the LDT or GDT table (kept in internal MMU registers) is added to it, to give a direct pointer to the descriptor. For example, selector 72 refers to entry 9 in the GDT, which is located at address GDT + 72.

Let us trace the steps by which a (selector, offset) pair is converted to a physical address. As soon as the hardware knows which segment register is being used, it can find the complete descriptor corresponding to that selector in its internal registers. If the segment does not exist (selector 0) or is currently not in memory (P is 0), a trap occurs. The former case is a programming error; the latter case requires the operating system to go get it.

It then checks to see if the offset is beyond the end of the segment, in which case a trap also occurs. Logically, there should simply be a 32-bit field in the descriptor giving the size of the segment, but only 20 bits are available, so a different scheme is used. If the G (Granularity) field is 0, the LIMIT field is the exact segment size, up to 1 MB. If it is 1, the LIMIT field gives the segment size in pages instead of bytes. The Core i7 page size is never smaller than 4 KB, so 20 bits is enough for segments up to 2^{32} bytes.

Assuming that the segment is in memory and the offset is in range, the Core i7 then adds the 32-bit BASE field in the descriptor to the offset to form what is called a **linear address**, as shown in Fig. 6-14. The BASE field is broken up into three pieces and spread all over the descriptor for backward compatibility with the 80286, in which the BASE is only 24 bits. In effect, the BASE field allows each segment to start at an arbitrary place within the 32-bit linear address space.

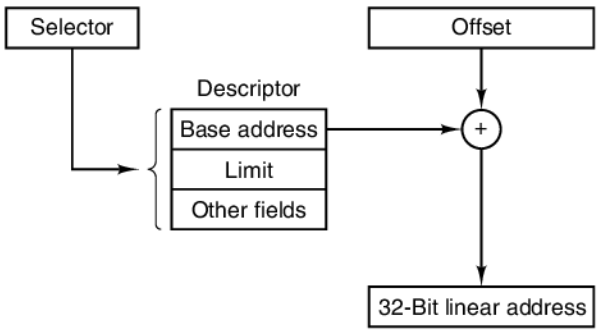

Figure 6-14. Conversion of a (selector, offset) pair to a linear address.

If paging is disabled (by a bit in a global control register), the linear address is interpreted as the physical address and sent to the memory for the read or write. Thus with paging disabled, we have a pure segmentation scheme, with each segment's base address given in its descriptor. Segments are permitted to overlap, incidentally, probably because it would be too much trouble and take too much time to verify that they were all disjoint.

On the other hand, if paging is enabled, the linear address is interpreted as a virtual address and mapped onto the physical address using page tables, pretty much as in our examples. The only complication is that with a 32-bit virtual address and a 4-KB page, a segment might contain 1 million pages, so a two-level mapping is used to reduce the page-table size for small segments.

Each running program has a **page directory** consisting of 1024 32-bit entries. It is located at an address pointed to by a global register. Each entry in this directory points to a page table also containing 1024 32-bit entries. The page-table entries point to page frames. The scheme is shown in Fig. 6-15.

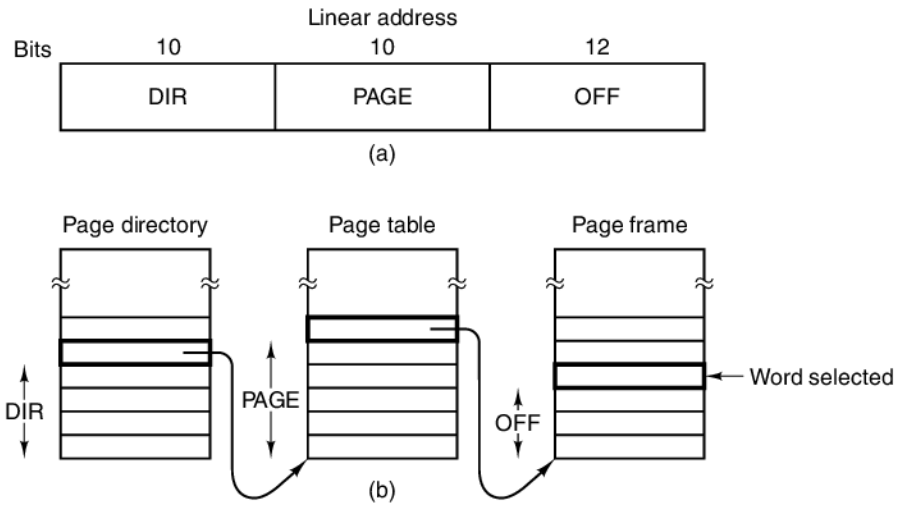

Figure 6-15. Mapping of a linear address onto a physical address.

In Fig. 6-15(a) we see a linear address broken up into three fields: DIR, PAGE, and OFF. The DIR field is first used as an index into the page directory to locate a pointer to the proper page table. Then the PAGE field is used as an index into the page table to find the physical address of the page frame. Finally, OFF is added to the address of the page frame to get the physical address of the byte or word addressed.

The page table entries are 32 bits each, of which 20 contain a page-frame number. The remaining bits contain access and dirty bits, set by the hardware for the benefit of the operating system, protection bits, and other utility bits.

Each page table has entries for 1024 4-KB page frames, so a single page table handles 4 megabytes of memory. A segment shorter than 4M will have a page directory with a single entry, a pointer to its one and only page table. In this way, the overhead for short segments is only two pages, instead of the million pages that would be needed in a one-level page table.

To avoid making repeated references to memory, the Core i7 MMU has special hardware support to look up the most recently used DIR – PAGE combinations quickly and map them onto the physical address of the corresponding page frame. Only when the current combination has not been used recently are the steps shown in Fig. 6-15 actually carried out.

A little thought will reveal the fact that when paging is used, there is really no point in having the BASE field in the descriptor be nonzero. All that BASE does is cause a small offset to use an entry in the middle of the page directory, instead of at the beginning. In truth, the real reason Intel included BASE at all is to allow pure

(nonpaged) segmentation, and for backward compatibility with the old 80286, which did not have paging.

It is also worth mentioning that if a particular application does not need segmentation, but is content with a single, paged, 32-bit address space, that is easy to obtain. All the segment registers can then be set up with the same selector, whose descriptor has BASE = 0 and LIMIT set to the maximum. The instruction offset will then be the linear address, with only a single address space used—in effect, traditional paging.

We are now finished with our treatment of virtual memory on the Core i7. We have looked only at a small (but commonly used) part of the Core i7 virtual memory system; the motivated reader can delve into the Core i7's documentation to also learn about the 64-bit virtual address extensions and support for virtualized physical address spaces. However before leaving the topic, it is worth saying a few words about protection, since this subject is intimately related to the virtual memory. The Core i7 supports four protection levels, with level 0 being the most privileged and level 3 the least. These are shown in Fig. 6-16. At each instant, a running program is at a certain level, indicated by a 2-bit field in its **PSW** (**Program Status Word**), a hardware register that holds the condition codes and various other status bits. Furthermore, each segment in the system also belongs to a certain level.

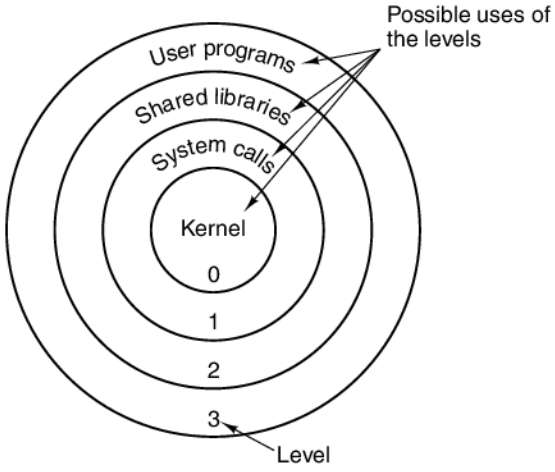

Figure 6-16. Protection on the Core i7.

As long as a program restricts itself to using segments at its own level, everything works fine. Attempts to access data at a higher level are permitted. Attempts to access data at a lower level are illegal and cause traps. Attempts to call procedures at a different level (higher or lower) are allowed, but in a carefully controlled way. In order to make an interlevel call, the CALL instruction must contain a

selector instead of an address. This selector designates a descriptor called a **call gate**, which gives the address of the procedure to be called. Thus it is not possible to branch into the middle of an arbitrary code segment at a different level. Only official entry points may be used.

A possible use for this mechanism is suggested in Fig. 6-16. At level 0, we find the kernel of the operating system, which handles I/O, memory management, and other critical matters. At level 1, the system call handler is present. User programs may call procedures here to have system calls carried out, but only a specific and protected list of procedures may be called. Level 2 contains library procedures, possibly shared among many running programs. User programs may call these procedures but may not modify them. Finally, user programs run at level 3, which has the least protection. Like the Core i7's memory-management scheme, the protection system is closely based on MULTICS.

Traps and interrupts use a mechanism similar to the call gates. They, too, reference descriptors, rather than absolute addresses, and these descriptors point to specific procedures to be executed. The TYPE field in Figure 6-13 distinguishes between code segments, data segments, and the various kinds of gates.

6.1.9 Virtual Memory on the OMAP4430 ARM CPU

The OMAP4430 ARM CPU is a 32-bit machine and supports a paged virtual memory based on 32-bit virtual addresses that are translated to a 32-bit physical address space. As such, an ARM CPU can support up to 2^{32} bytes (4 GB) of physical memory. Four page sizes are supported: 4 KB, 64 KB, 1 MB, and 16 MB. The mappings implied by these four page sizes are illustrated in Fig. 6-17.

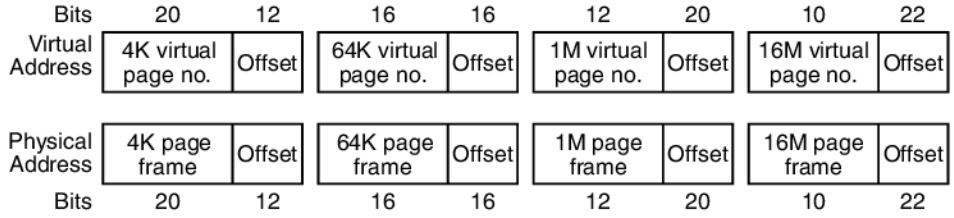

Figure 6-17. Virtual-to-physical mappings on the OMAP4430 ARM CPU.

The OMAP4430 ARM CPU uses a page-table structure similar to that of the Core i7. The page-table mapping for a 4-KB virtual address page is shown in Fig. 6-18(a). The first-level descriptor table is indexed with the most significant 12 bits of the virtual address. The first-level descriptor-table entry indicates the physical address of the second-level descriptor table. This address, combined with the next 8 bits of the virtual address, produce the page-descriptor address. The page descriptor contains the address of the physical page frame plus permission information regarding page accesses.

The OMAP4430 ARM CPU virtual memory mapping accommodates four page sizes. Page sizes of 1 MB and 16 MB are mapped with a page descriptor located in the first-level descriptor table. There is no need for second-level tables in this case, as all of the entries would point to the same large physical page. The 64-KB pages descriptors are located in the second-level descriptor table. Because each entry of the second-level descriptor table maps 4 KB of virtual address page to a 4-KB physical address page, 64-KB pages require 16 identical descriptors in the second-level descriptor table. Now why would any sane OS programmer declare a page as 64 KB in size when the same space would be required to map the page to more flexible 4-KB pages? Because, as we will see shortly, 64-KB pages require fewer TLB entries, which are a critical resource to good performance.

Nothing slows a program down more than a constricting memory bottleneck. If you were keeping score in Fig. 6-18, you probably noticed that for every program memory access two additional memory accesses are required for address translation. This 200% overhead in memory accesses for virtual address translation would bring any program to a crawl. To avoid this bottleneck, the OMAP4430 ARM CPU incorporates a hardware table called a **TLB** (**Translation Lookaside Buffer**) that quickly maps virtual page numbers onto physical-page-frame numbers. For the 4-KB page size, there are 2^{20} virtual page numbers, which is over 1 million. Clearly, not all of them can be mapped.

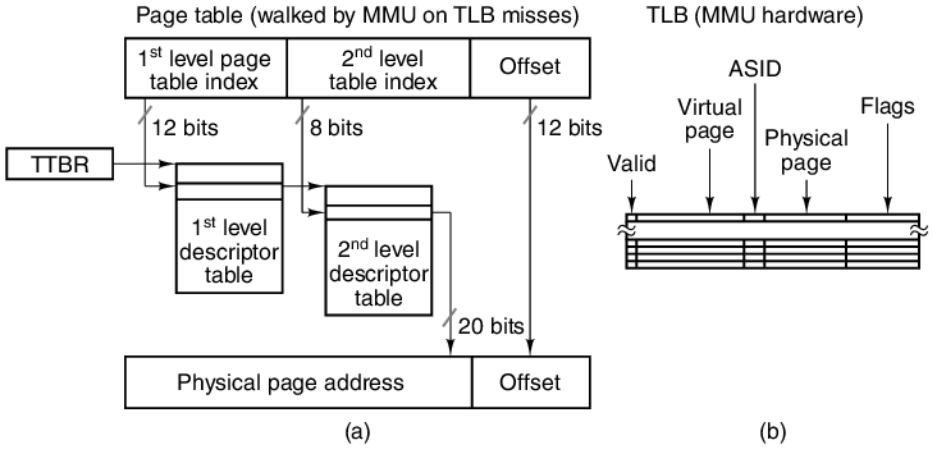

Figure 6-18. Data structures used in translating virtual addresses on the OMAP4430 ARM CPU. (a) Address translation table. (b) TLB.

Instead, the TLB holds only the most recently used virtual page numbers. Instruction and data pages are kept track of separately, with the TLB holding the 128 most recently used virtual page numbers in each category. Each TLB entry holds a virtual page number and the corresponding physical page-frame number. When a process number, called an **address space identifier** (ASID), and a virtual address

within that address space is presented to the MMU, it uses special circuitry to compare the virtual page number contained in it to all the TLB entries at once. If a match is found, the page frame number in that TLB entry is combined with the offset taken from the virtual address to form a 32-bit physical address and produce some flags, such as protection bits. The TLB is illustrated in Fig. 6-18(b).

However, if no match is found, a **TLB miss** occurs, which initiates a hardware "walk" of the page tables. When the new physical-page descriptor entry is located in the page table, it is checked to see if the page is in memory and, if so, its corresponding address translation is loaded into the TLB. If the page is not in memory, a standard page-fault action is started. Since the TLB has only a few entries, it is quite likely to displace an existing entry in the TLB. Future accesses to the displaced page will have to once again walk the page tables to get an address mapping. If too many pages are being touched too quickly, the TLB will thrash, and most memory accesses will require a 200% overhead for address translation.

It is interesting to compare the Core i7 and OMAP4430 ARM CPU virtual memory systems. The Core i7 supports pure segmentation, pure paging, and paged segments. The OMAP4430 ARM CPU has only paging. Both the Core i7 and the OMAP4430 use hardware to walk the page table to reload the TLB in the event of a TLB miss. Other architectures, such as SPARC and MIPS, just give control to the operating system on a TLB miss. These architectures define special privileged instructions to manipulate the TLB, such that the operating system can perform the page-table walks and TLB loads necessary for address translation.

6.1.10 Virtual Memory and Caching

Although at first glance, (demand-paged) virtual memory and caching may look unrelated, they are conceptually very similar. With virtual memory, the entire program is kept on disk and broken up into fixed-size pages. Some subset of these pages are in main memory. If the program mostly uses the pages in memory, there will be few page faults and the program will run fast. With caching, the entire program is kept in main memory and broken up into fixed-size cache blocks. Some subset of these blocks are in the cache. If the program mostly uses the blocks in the cache, there will be few cache misses and the program will run fast. Conceptually, the two are identical, only operating at different levels in the hierarchy.

Of course, virtual memory and caching also have some differences. For one, cache misses are handled by the hardware, whereas page faults are handled by the operating system. Also, cache blocks are typically much smaller than pages (e.g., 64 bytes vs. 8 KB). In addition, the mapping between virtual pages and page frames is different, with page tables organized by indexing on the high-order bits of the virtual address, whereas caches index on the low-order bits of the memory address. Nevertheless, it is important to realize that these are implementation differences. The underlying concept is very similar.

6.2 HARDWARE VIRTUALIZATION

Traditionally, hardware architectures have been designed with the expectation that they will run one operating system at a time. The proliferation of shared computing resources, such as cloud computing servers, benefit from having the ability to run multiple operating systems at the same time. For example, Internet hosting services typically provide a complete system to paying clients, upon which can be built web services. It would be prohibitively expensive to install a new computer in the server room each time a new customer enrolls. Instead, hosting services typically use **virtualization** to support the execution of multiple complete systems, including the operating system, on one server. Only when the existing servers become too overloaded does the hosting service have to install a new physical server in the server pool.

While software-only approaches to virtualization do exist, they typically slow down the virtual system, and they require specific operating system modifications or utilize complex code analyzers to rewrite programs on the fly. These drawbacks have led architects to enhance the OSM level of the architecture to support efficient virtualization directly in hardware.

Hardware virtualization, as illustrated in Fig. 6-19, is a combination of hardware and software support that enables the simultaneous execution of multiple operating systems on a single physical computer. To the user, each **virtual machine** running on the host computer appears to be a complete standalone computing system. The **hypervisor** is a software component, much like an operating system kernel, that creates and manages instances of virtual machines. The hardware provides the software-visible events that are necessary for the hypervisor to implement sharing policies for the CPU, storage, and I/O devices.

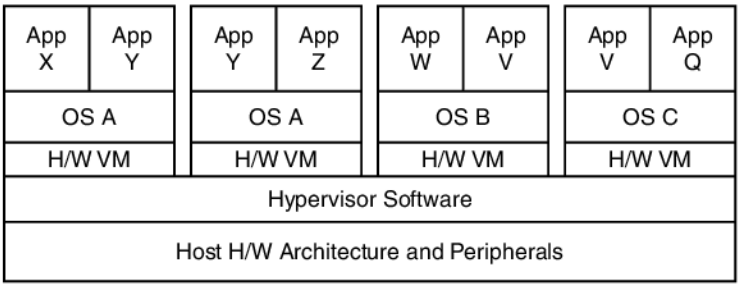

Figure 6-19. Hardware virtualization allows multiple operating systems to run simultaneously on the same host hardware. The hypervisor implements sharing of host memory and I/O devices.

The existence of multiple virtual machines on one host computer, each possibly running a different operating system, provides many benefits. In server systems,

virtualization gives system administrators the ability to place multiple virtual machines on the same physical server and to move running virtual machines between servers to better distribute the total load. Virtual machines also give server administrators fine-grained control over I/O device access. For example, the bandwidth of a virtualized network port could be partitioned based on users' service levels. For individual users, virtualization offers the ability to run multiple operating systems simultaneously.

To implement virtualization in hardware, all instructions in the architecture must only access the resources of the current virtual machine. For most instructions, this is a trivial requirement. For example, arithmetic instruction need only access the register file, which can be virtualized by copying a virtual machine's registers into the host processor register file at virtual machine context switches.

Virtualizing memory access instructions (e.g., loads and stores) is slightly more challenging, as these instructions must only access physical memory allocated to the currently executing virtual machine. Typically, a processor supporting hardware virtualization will provide an additional page-mapping facility that maps virtual machine physical memory pages to host machine physical memory pages. Finally, I/O instructions (including memory-mapped I/O) must not directly access physical I/O devices, since many virtualization policies partition access to I/O devices. This fine-grained I/O control is typically implemented with interrupts to the hypervisor any time a virtual machine attempts to access an I/O device. The hypervisor can then implement the I/O resource access policy of its own choosing. Typically, some set of I/O devices is supported and the operating systems running in the virtual machines, called **guest operating systems** are expected to use these supported devices.

6.2.1 Hardware Virtualization on the Core I7

Hardware virtualization on the Core i7 is supported by the virtual machine extensions (VMX), a combination of instruction, memory, and interrupt extensions that allow the efficient management of virtual machines. With VMX, memory virtualization is implemented with the **EPT** (**Extended Page Table**) system that is enabled with hardware virtualization. The EPT translates virtual machine physical page addresses to host physical addresses. The EPT implements this mapping with an additional multilevel page table structure that is traversed during a virtual machine TLB miss. The hypervisor maintains this table, and in doing so it can implement any physical memory sharing policy desired.

Virtualization of I/O operations, for both memory-mapped I/O and I/O instructions, is implemented through extended interrupt support defined in the **VMCS** (**Virtual-Machine Control Structure**) A hypervisor interrupt is invoked anytime a virtual machine accesses an I/O device. Once the interrupt is received by the hypervisor, it can implement the I/O operation in software using the policies necessary to allow sharing of the I/O device among virtual machines.

6.3 OSM-LEVEL I/O INSTRUCTIONS

The ISA-level instruction set is completely different from the microarchitecture instruction set. Both the operations that can be performed and the formats for the instructions are quite different at the two levels. The occasional existence of a few instructions that are the same at both levels is essentially accidental.

In contrast, the OSM-level instruction set contains most of the ISA-level instructions, with a few new, but important, instructions added and a few potentially damaging instructions removed. Input/output is one of the areas where the two levels differ considerably. The reason is simple: a user who could execute the real ISA-level I/O instructions could read confidential data stored anywhere in the system, write in other users' directories, and, in general, make a big nuisance of himself or herself and even threaten the security of the system itself. Second, normal, sane programmers do not want to do I/O at the ISA level themselves because doing so is extremely tedious and complex. It is done by setting fields and bits in a number of device registers, waiting until the operation is completed, and then checking to see what happened. As an example of the latter, disks typically have device-register bits to detect the following errors, among many others:

1. Disk arm failed to seek properly.
2. Nonexistent memory specified as buffer.
3. Disk I/O started before previous one finished.
4. Read timing error.
5. Nonexistent disk addressed.
6. Nonexistent cylinder addressed.
7. Nonexistent sector addressed.
8. Checksum error on read.
9. Write-check error after write operation.

When one of these errors occurs, the corresponding bit in a device register is set. Few users want to be bothered keeping track of all these error bits and a great deal of additional status information.

6.3.1 Files

One way of organizing the virtual I/O is to use an abstraction called a **file**. In its simplest form, a file consists of a sequence of bytes written to an I/O device. If the I/O device is a storage device, such as a disk, the file can be read back later; if the device is not a storage device (e.g., a printer), it cannot be read back, of course.

A disk can hold many files, each with some particular kind of data, for example, a picture, a spreadsheet, or the text of a book chapter. Different files have different lengths and other properties. The abstraction of a file allows virtual I/O to be organized in a simple way.

To the operating system, a file is normally just a sequence of bytes, as we have described above. Any further structure is up to the application programs. File I/O is done by system calls for opening, reading, writing, and closing files. Before a file can be read, it must be opened. The process of opening a file allows the operating system to locate the file on disk and bring into memory information necessary to access it.

Once a file has been opened, it can be read. The read system call must have the following parameters, at a minimum:

1. An indication of which open file is to be read.
2. A pointer to a buffer in memory in which to put the data.
3. The number of bytes to be read.

The read call puts the requested data in the buffer. Usually, it returns the count of the number of bytes actually read, which may be smaller than the number requested (you cannot read 2000 bytes from a 1000-byte file).

Associated with each open file is a pointer telling which byte will be read next. After a read it is advanced by the number of bytes read, so consecutive reads read consecutive blocks of data from the file. Usually, there is a way to set this pointer to a specific value, so programs can randomly access any part of the file. When a program is done reading a file, it can close it, to inform the operating system that it will not be using the file any more, thus allowing the operating system to free up the table space being used to hold information about the file.

Mainframe computers are still around (especially for running very large e-commerce Websites) and some of them still run traditional operating systems (although many run Linux). The traditional mainframe operating systems have a different model of what a file is, and it is worth taking a brief look at this model, just to show that the UNIX way is not the only way to do things. In these traditional systems, a file is a sequence of **logical records**, each with a well-defined structure. For example, a logical record might be a data structure consisting of five items: two character strings, "Name," and "Supervisor"; two integers, "Department" and "Office"; and a Boolean, "SexIsFemale." Some operating systems make a distinction between files in which all the records in a file have the same structure and files which contain a mixture of different record types.

The basic virtual input instruction reads the next record from the specified file and puts it into main memory beginning at a specified address, as illustrated in Fig. 6-20. To perform this operation, the virtual instruction must be told which file to read and where in memory to put the record. Often there are options to read a specific record, specified either by its position in the file or by its key.

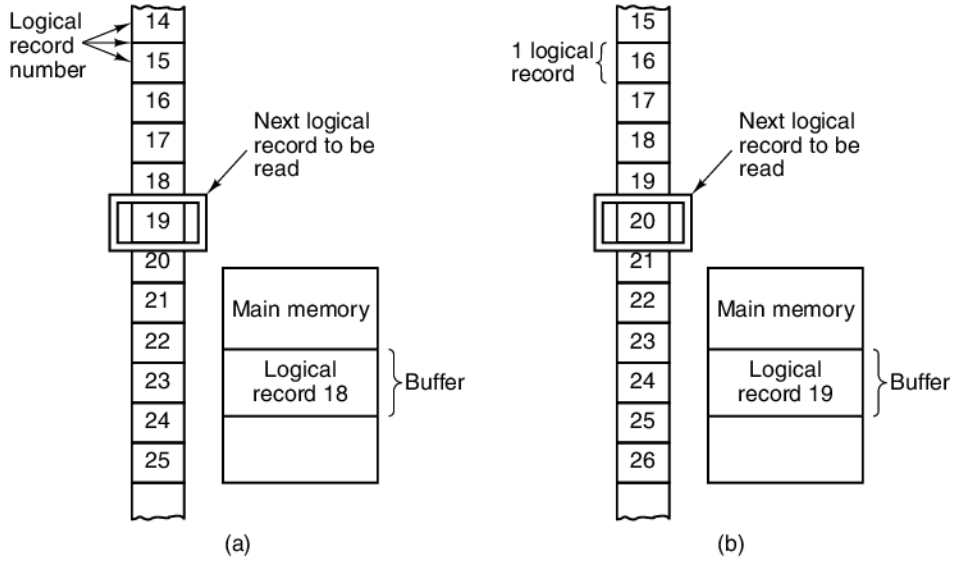

Figure 6-20. Reading a file consisting of logical records. (a) Before reading record 19. (b) After reading record 19.

The basic virtual output instruction writes a logical record from memory onto a file. Consecutive sequential write instructions produce consecutive logical records on the file.

6.3.2 Implementation of OSM-Level I/O Instructions

To understand how virtual I/O instructions are implemented, we need to examine how files are organized and stored. A basic issue that must be dealt with by all file systems is allocation of storage. The allocation unit (sometimes called a block) can be a single disk sector, but often it consists of a block of consecutive sectors.

Another fundamental property of a file-system implementation is whether a file is stored in consecutive allocation units or not. Figure 6-21 depicts a simple disk with one surface consisting of five tracks of 12 sectors each. Figure 6-21(a) shows an allocation scheme in which the sector is the basic unit of space allocation and in which a file consists of consecutive sectors. Consecutive allocation of file blocks is commonly used on CD-ROMs. Figure 6-21(b) shows an allocation scheme in which the sector is the basic allocation unit but in which a file need not occupy consecutive sectors. This scheme is the norm on hard disks (and, of course, solid state disks).

There is a key distinction between the application programmer's and the operating system's view of a file. The programmer sees the file as a linear sequence of

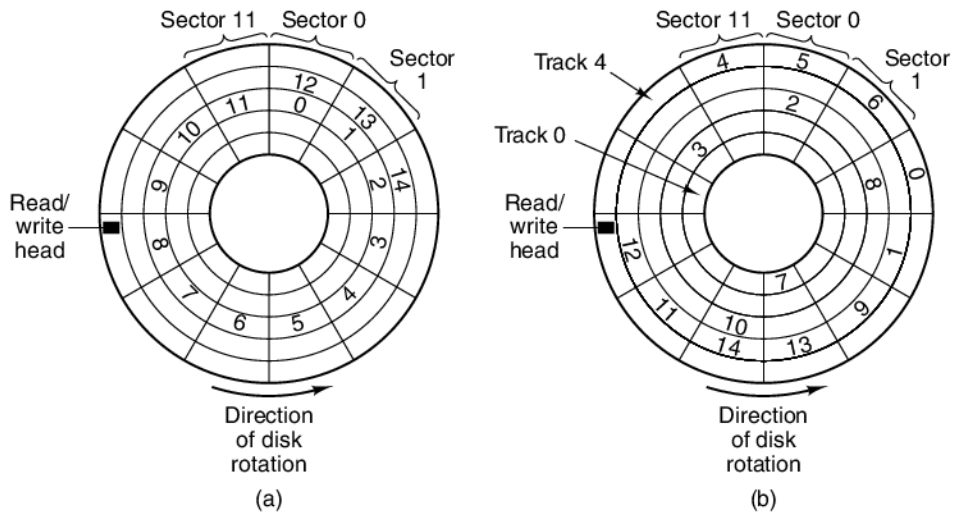

Figure 6-21. Disk allocation strategies. (a) A file in consecutive sectors. (b) A file not in consecutive sectors.

bytes or logical records. The operating system sees the file as an ordered, although not necessarily consecutive, collection of allocation units on disk.

In order for the operating system to deliver byte or logical record *n* of some file on request, it must have some method for locating the data. If the file is allocated consecutively, the operating system need only know the location of the start of the file in order to calculate the position of the byte or logical record needed.

If the file is not allocated consecutively, it is not possible to calculate the position of an arbitrary byte or logical record in the file from the position of the start of the file alone. Rather, a table called a **file index**, giving the allocation units and their actual disk addresses, is needed. The file index can be organized either as a list of disk block addresses (used by UNIX), a list of runs of consecutive blocks (used by Windows 7), or as a list of logical records, giving the disk address and offset for each one. Sometimes each logical record has a **key** and programs can refer to a record by its key, rather than by its logical record number. In this case, the latter organization is required, with each entry containing not only the location of the record on disk, but also its key. This organization is common on mainframes.

An alternative method of locating the allocation units of a file is to organize the file as a linked list. Each allocation unit contains the address of its successor. One way to implement this scheme efficiently is to keep the table with all the successor addresses in main memory. For example, for a disk with 64K allocation units, the operating system could have a table in memory with 64K entries, each one giving the index of its successor. For example, if a file occupied allocation

units 4, 52, and 19, entry 4 in the table would contain a 52, entry 52 would contain a 19, and entry 19 would contain a special code (e.g., 0 or −1) to indicate end of file. The file systems used by MS-DOS and Windows 95 and Windows 98 worked this way. Newer versions of Windows (2000, XP, Vista, and 7) still support this file system but also have their own native file systems that work more like UNIX.

Up until now we have discussed both consecutively and nonconsecutively allocated files but have not specified why both kinds are used. Consecutively allocated files have simpler block administration, but when the maximum file size is not known in advance, it is rarely possible to use this technique. If a file is started at sector *j* and allowed to grow into consecutive sectors, it may bump into another file at sector *k* and have no room to expand. If the file is not allocated consecutively, this situation presents no problem, because succeeding blocks can be put anywhere on the disk. If a disk contains a number of growing files, none of whose final sizes is known, storing each of them as a consecutive file will be nearly impossible. Moving an existing file is sometimes possible but always expensive in terms of time and system resources.

On the other hand, if the maximum size of all files is known in advance, as it is when a CD-ROM is burned, the recording program can preallocate a run of sectors exactly equal in length to each file. Thus if files with lengths of 1200, 700, 2000, and 900 sectors are to be put on a CD-ROM, they can be simply begun at sectors 0, 1200, 1900, and 3900, respectively (ignoring the table of contents here). Finding any part of any file is simple once the file's first sector is known.

In order to allocate space on the disk for a file, the operating system must keep track of which blocks are available and which are already in use storing other files. For a CD-ROM, the calculation is done once and for all in advance, but for a hard disk, files come and go all the time. One method consists of maintaining a list of all the holes, a hole being any number of contiguous allocation units. This list is called the **free list**. Figure 6-22(a) illustrates the free list for the disk of Fig. 6-21(b) with one sector per allocation unit.

An alternative method is to maintain a bit map, with 1 bit per allocation unit, as shown in Fig. 6-22(b). A 1 bit indicates that the allocation unit is already occupied and a 0 bit indicates that it is available.

The first method has the advantage of making it easy to find a hole of a particular length. However, it has the disadvantage of being variable sized. As files are created and destroyed, the length of the list will fluctuate, an undesirable characteristic. The bit table has the advantage of being constant in size. In addition, changing the status of an allocation unit from available to occupied is just a matter of changing 1 bit. However, finding a block of a given size is difficult. Both methods require that when any file on the disk is allocated or returned, the allocation list or table be updated.

Before leaving the subject of file-system implementation, it is worth commenting about the size of the allocation unit. Several factors play a key role here. First, seek time and rotational delay dominate disk accesses. Having invested 5–10 msec

Track	Sector	Number of sectors in hole
0	0	5
0	6	6
1	0	10
1	11	1
2	1	1
2	3	3
2	7	5
3	0	3
3	9	3
4	3	8

(a)

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

(b)

Figure 6-22. Two ways of keeping track of available sectors. (a) A free list. (b) A bit map.

to get to the start of an allocation unit, it is far better to read 8 KB (about 80 μsec) than 1 KB (about 10 μsec), since reading 8 KB as eight 1-KB units may require eight seeks. Transfer efficiency argues for large units. Of course, as solid-state disks become cheaper and more common, this argument ceases to hold, since these devices have no seek time at all.

Also arguing for large allocation units is the fact that having small allocation units means having many of them. Having, many allocation units, in turn, means large file indices or large linked-list tables in memory. As a historical note, MS-DOS started out with the allocation unit being one sector (512 bytes) and 16-bit numbers being used to identify sectors. When disks grew beyond 65,336 sectors, the only way to use all the space on the disk and still use 16-bit numbers to identify the allocation units was to use bigger and bigger allocation units. The first release of Windows 95 had the same problem, but a subsequent release used 32-bit numbers. Windows 98 supported both sizes.

However, arguing in favor of small allocation units is the fact that few files occupy exactly an integral number of allocation units. Therefore, some space will be wasted in the last allocation unit of nearly every file. If the file is much larger than the allocation unit, the average space wasted will be half an allocation unit. The larger the allocation unit, the larger the amount of wasted space. If the average file is much smaller than the allocation unit, most of the disk space will be wasted.

For example, on an MS-DOS or Windows 95 release 1 disk partition of 2 GB, the allocation units were 32 KB, so a 100-character file wasted 32,668 bytes of disk space. Storage efficiency argues for small allocation units. Due to the ever-decreasing price of large disks, efficiency in time (i.e., faster performance) tends to be the most important factor nowadays, so allocation units tend to be increasing over time and the wasted disk space simply accepted.

6.3.3 Directory Management Instructions

In the early days of computing, people kept their programs and data on punched cards in file cabinets in their offices. As the programs and data grew in size and number, this situation became less and less desirable. It eventually led to the idea of using the computer's secondary memory (e.g., disk) as a storage place for programs and data as an alternative to file cabinets. Information that is directly accessible to the computer without the need for human intervention is said to be **online**, as contrasted with **offline** information, which requires human intervention (e.g., inserting a tape, CD-ROM, USB stick, or SD card) before the computer can access it.

Online information is stored in files, making it accessible to programs via the file I/O instructions discussed above. However, additional instructions are needed to keep track of the information stored online, collect it into convenient units, and protect it from unauthorized use.

The usual way for an operating system to organize online files is to group them into **directories**. Figure 6-23 shows an example directory organization. System calls are provided for at least the following functions:

1. Create a file and enter it in a directory.
2. Delete a file from a directory.
3. Rename a file.
4. Change the protection status of a file.

All modern operating systems allow users to maintain more than one file directory. Each directory is typically itself a file and, as such, may be listed in another directory, thus giving rise to a tree of directories. Multiple directories are particularly useful for programmers working on several projects. They can then group all the files related to one project together in one directory. While working on that project, they will not be distracted by unrelated files. Directories are also a convenient way for people to share files with members of their group.

6.4 OSM-LEVEL INSTRUCTIONS FOR PARALLEL PROCESSING

Some computations can be most conveniently programmed for two or more cooperating processes running in parallel (i.e., simultaneously, on different processors) rather than for a single process. Other computations can be divided into pieces, which can then be carried out in parallel to decrease the elapsed time required for the total computation. In order for several processes to work together in

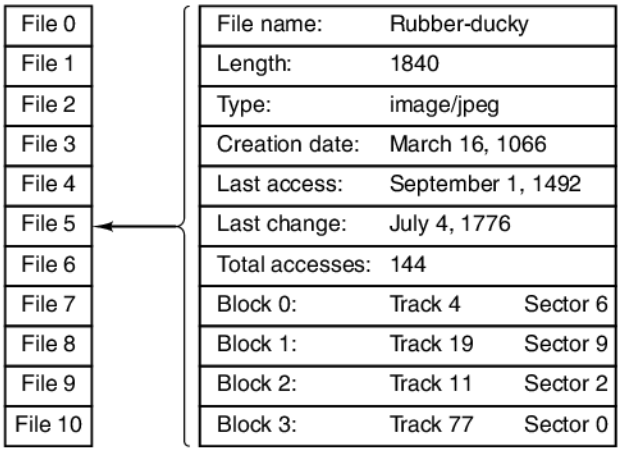

Figure 6-23. A user file directory and the contents of a typical entry in a file directory.

parallel, certain virtual instructions are needed. These instructions will be discussed in the following sections.

The laws of physics provide yet another reason for the current interest in parallel processing. According to Einstein's special theory of relativity, it is impossible to transmit electrical signals faster than the speed of light, which is nearly 1 ft/nsec in vacuum, less in copper wire or optical fiber. This limit has important implications for computer organization. For example, if a CPU needs data from the main memory 1 ft away, it will take at least 1 nsec for the request to arrive at the memory and another nanosecond for the reply to get back to the CPU. Consequently, subnanosecond computers will need to be extremely tiny. An alternative approach to speeding up computers is to build machines with many CPUs. A computer with a thousand 1-nsec CPUs may (in theory) have the same computing power as one CPU with a cycle time of 0.001 nsec, but the former may be much easier and cheaper to construct. Parallel computing is discussed in detail in Chap. 8.

On a computer with more than one CPU, each of several cooperating processes can be assigned to its own CPU, to allow the processes to progress simultaneously. If only one processor is available, the effect of parallel processing can be simulated by having the processor run each process in turn for a short time. In other words, the processor can be shared among several processes.

Figure 6-24 shows the difference between true parallel processing, with more than one physical processor, and simulated parallel processing, with only one physical processor. Even when parallel processing is simulated, it is useful to regard each process as having its own dedicated virtual processor. The same communication problems that arise when there is true parallel processing arise also in the simulated case. In both cases, debugging the problems is very difficult.

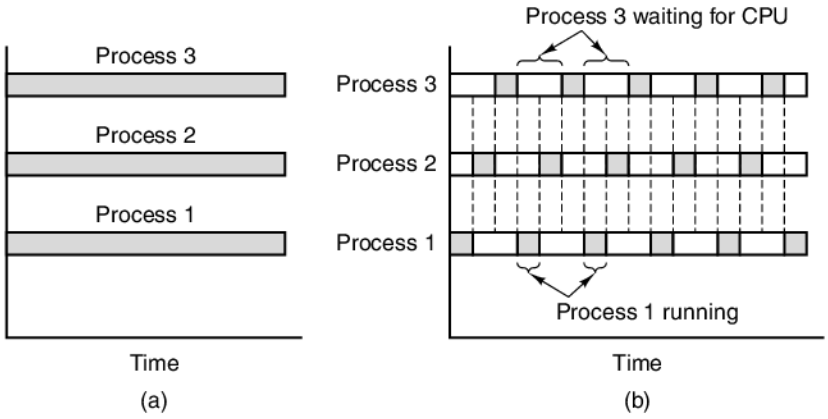

Figure 6-24. (a) True parallel processing with multiple CPUs. (b) Parallel processing simulated by switching one CPU among three processes.

6.4.1 Process Creation

When a program is to be executed, it must run as part of some process. This process, like all others, is characterized by a state and an address space through which the program and data can be accessed. The state includes at the very least the program counter, a program status word, a stack pointer, and the general registers.

Most modern operating systems allow processes to be created and terminated dynamically. To take full advantage of this feature to achieve parallel processing, a system call to create a new process is needed. This system call may just make a clone of the called, or it may allow the creating process to specify the initial state of the new process, including its program, data, and starting address.

In some cases, the creating (parent) process maintains partial or even complete control over the created (child) process. To this end, virtual instructions exist for a parent to stop, restart, examine, and terminate its children. In other cases, a parent has less control over its children: once a process has been created, there is no way for the parent to forcibly stop, restart, examine, or terminate it. The two processes then run independently of one another.

6.4.2 Race Conditions

In many cases, parallel processes need to communicate and synchronize in order to get their work done. In this section, process synchronization will be examined and some of the difficulties explained by means of a detailed example. A solution to these difficulties will be given in the following section.

Consider a situation consisting of two independent processes, process 1 and process 2, which communicate via a shared buffer in main memory. For simplicity we will call process 1 the **producer** and process 2 the **consumer**. The producer computes prime numbers and puts them into the buffer one at a time. The consumer removes them from the buffer one at a time and prints them.

These two processes run in parallel at different rates. If the producer discovers that the buffer is full, it goes to sleep; that is, it temporarily suspends its operation awaiting a signal from the consumer. Later, when the consumer has removed a number from the buffer, it sends a signal to the producer to wake it up—that is, restart it. Similarly, if the consumer discovers that the buffer is empty, it goes to sleep. When the producer has put a number into the empty buffer, it wakes up the sleeping consumer.

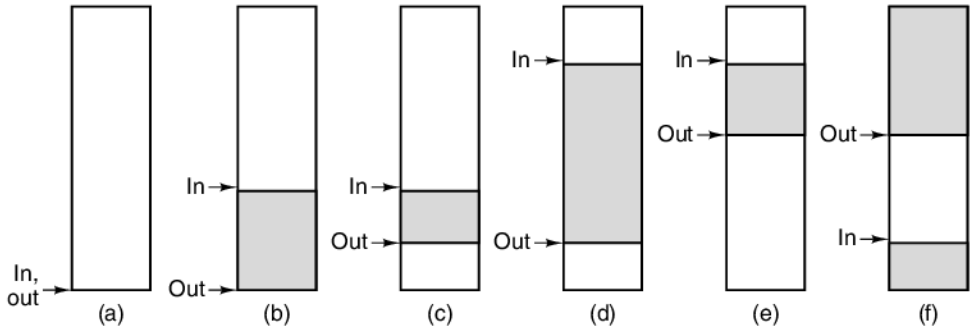

Figure 6-25. Use of a circular buffer.

In this example we will use a circular buffer for interprocess communication. The pointers *in* and *out* will be used as follows: *in* points to the next free word (where the producer will put the next prime) and *out* points to the next number to be removed by the consumer. When *in* = *out*, the buffer is empty, as shown in Fig. 6-25(a). After the producer has generated some primes, the situation is as shown in Fig. 6-25(b). Fig. 6-25(c) illustrates the buffer after the consumer has removed some of these primes for printing. Figure 6-25(d)–(f) depict the effect of continued buffer activity. The top of the buffer is logically contiguous with the bottom; that is, the buffer wraps around. When there has been a sudden burst of input and *in* has wrapped around and is only one word behind *out* (e.g., *in* = 52, and *out* = 53), the buffer is full. The last word is not used; if it were, there would be no way to tell whether *in* = *out* meant a full buffer or an empty one.

Figure 6-26 shows a simple way to implement the producer-consumer problem in Java. This solution uses three classes, *m*, *producer*, and *consumer*. The *m* (main) class contains some constant definitions, the buffer pointers *in* and *out*, and the buffer itself, which in this example holds 100 primes, going from *buffer*[0] to *buffer*[99]. The *producer* and *consumer* classes do the actual work.

```
public class m {
   final public static int BUF_SIZE = 100;                    // buffer runs from 0 to 99
   final public static long MAX_PRIME = 100000000000L;        // stop here
   public static int in = 0, out = 0;                         // pointers to the data
   public static long buffer[ ] = new long[BUF_SIZE];         // primes stored here
   public static producer p;                                  // name of the producer
   public static consumer c;                                  // name of the consumer

   public static void main(String args[ ]) {                  // main class
      p = new producer( );                                    // create the producer
      c = new consumer( );                                    // create the consumer
      p.start( );                                             // start the producer
      c.start( );                                             // start the consumer
   }

   // This is a utility function for circularly incrementing in and out
   public static int next(int k) {if (k < BUF_SIZE – 1) return(k+1); else return(0);}
}

class producer extends Thread {                               // producer class
   public void run( ) {                                       // producer code
      long prime = 2;                                         // scratch variable

      while (prime < m.MAX_PRIME) {
         prime = next_prime(prime);                           // statement P1
         if (m.next(m.in) == m.out) suspend( );               // statement P2
         m.buffer[m.in] = prime;                              // statement P3
         m.in = m.next(m.in);                                 // statement P4
         if (m.next(m.out) == m.in) m.c.resume( );            // statement P5
      }
   }

   private long next_prime(long prime){ ... }                 // function that computes next prime
}

class consumer extends Thread {                               // consumer class
   public void run( ) {                                       // consumer code
      long emirp = 2;                                         // scratch variable

      while (emirp < m.MAX_PRIME) {
         if (m.in == m.out) suspend( );                       // statement C1
         emirp = m.buffer[m.out];                             // statement C2
         m.out = m.next(m.out);                               // statement C3
         if (m.out == m.next(m.next(m.in))) m.p.resume( );    // statement C4
         System.out.println(emirp);                           // statement C5
      }
   }
}
```

Figure 6-26. Parallel processing with a fatal race condition.