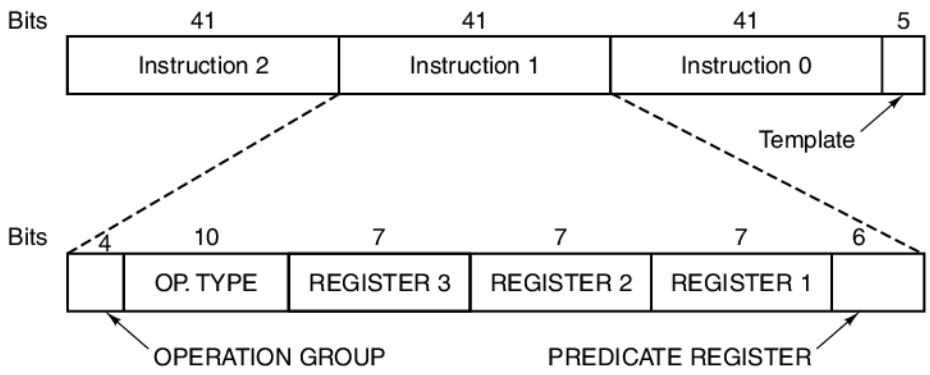

Figure 5-47. An IA-64 bundle contains three instructions.

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

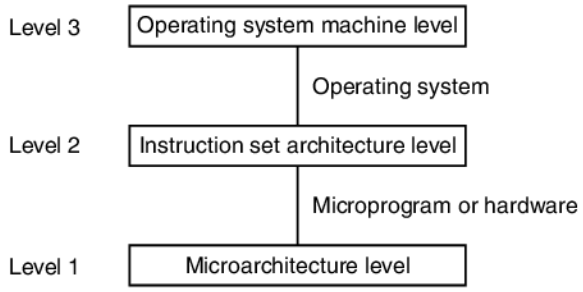

Figure 6-1. Positioning of the operating system machine level.

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.

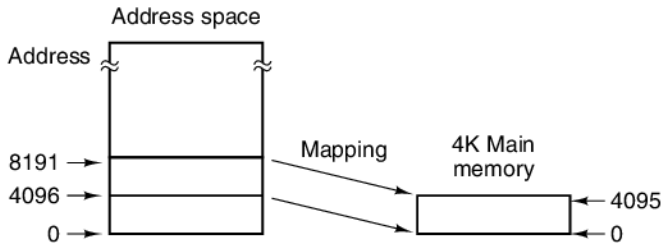

Figure 6-2. A mapping in which virtual addresses 4096 to 8191 are mapped onto main memory addresses 0 to 4095.

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

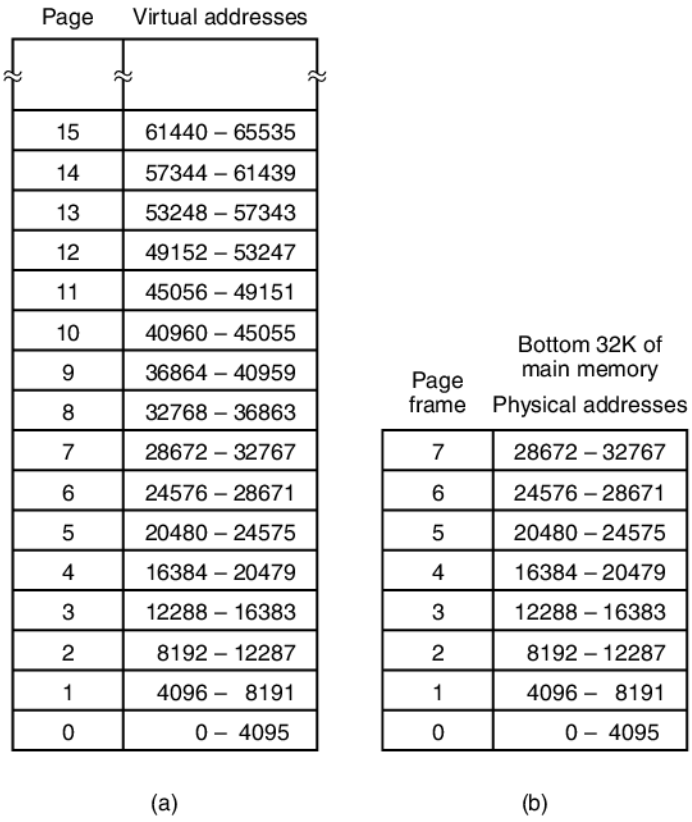

Figure 6-3. (a) The first 64 KB of virtual address space divided into 16 pages, with each page being 4K. (b) A 32-KB main memory divided up into eight page frames of 4 KB each.

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**.

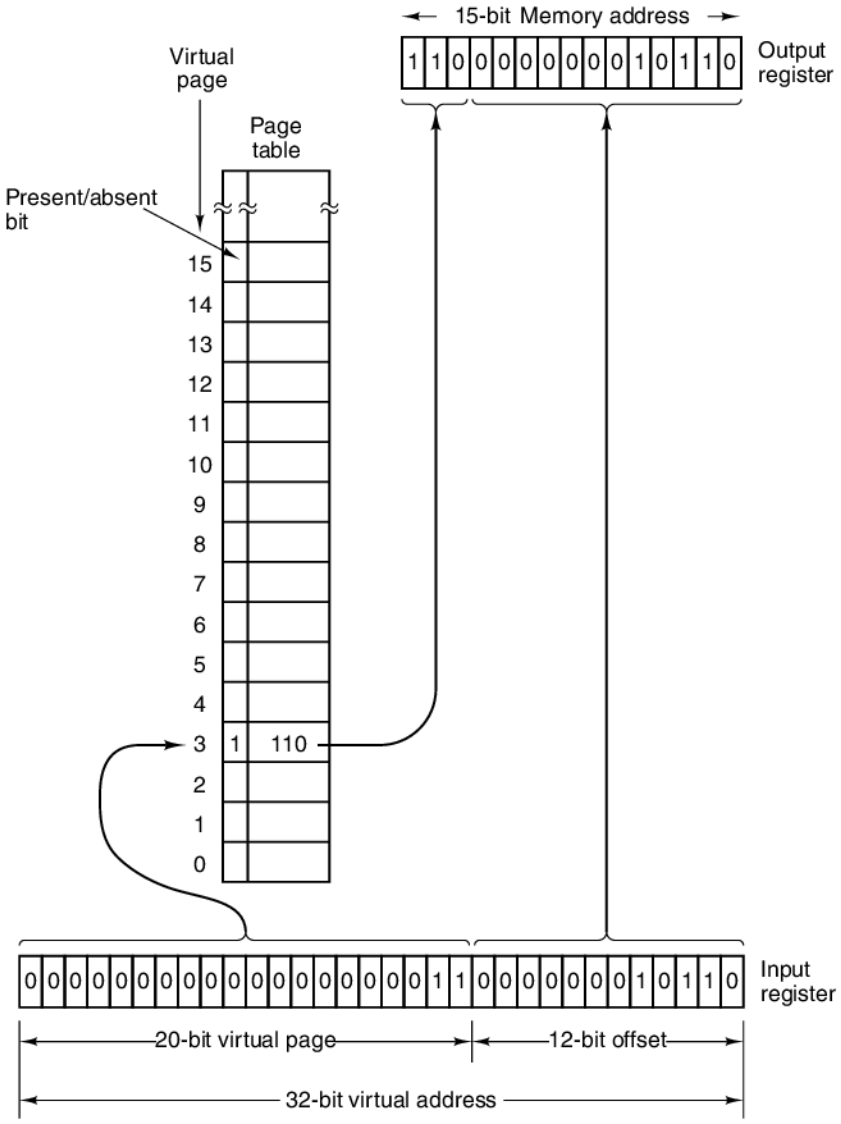

Figure 6-4. Formation of a main memory address from a virtual address.

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

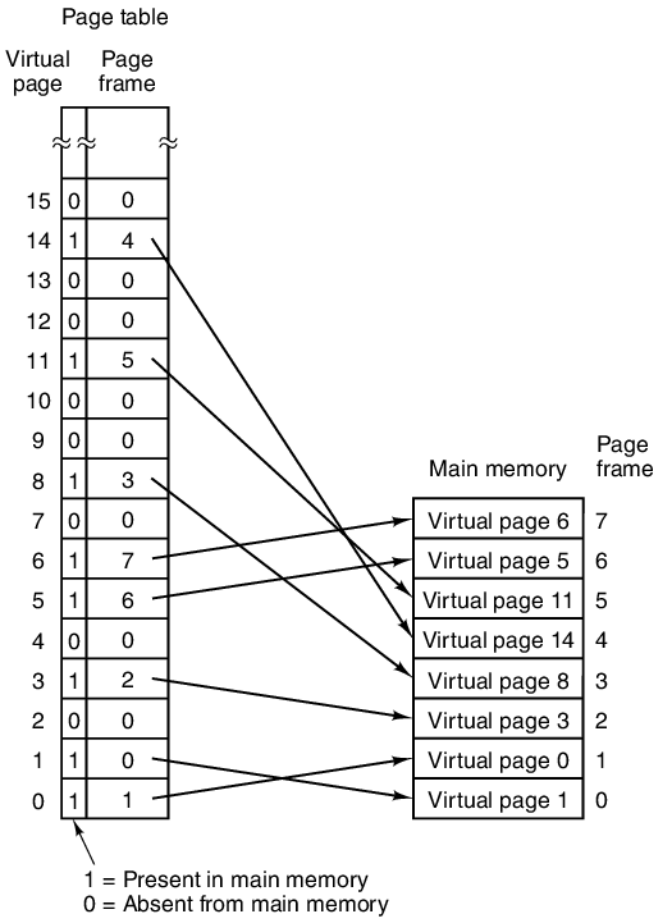

Figure 6-5. A possible mapping of the first 16 virtual pages onto a main memory with eight page frames.

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).

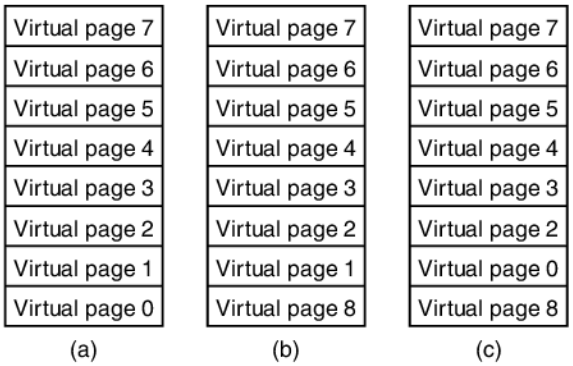

Figure 6-6. Failure of the LRU algorithm.

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

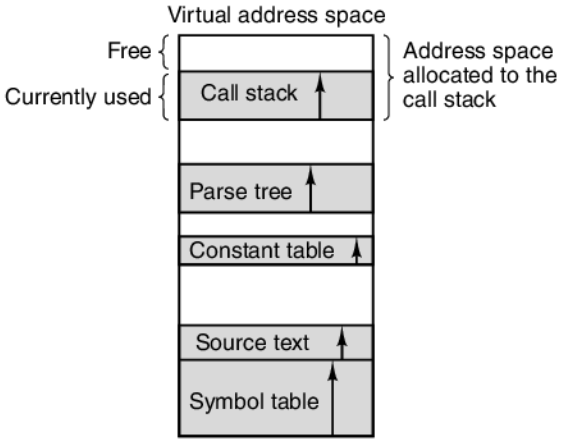

Figure 6-7. In a one-dimensional address space with growing tables, one table may bump into another.

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.

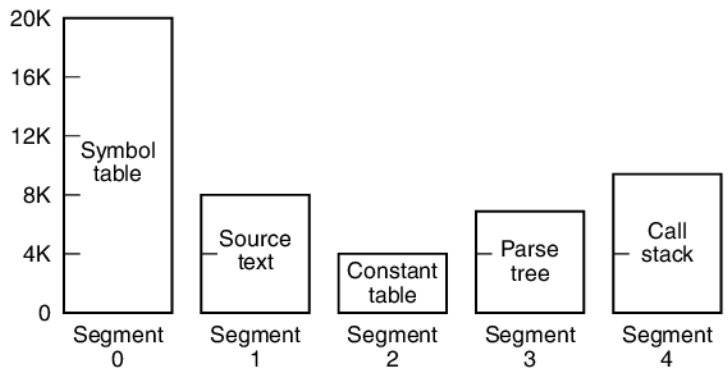

Figure 6-8. A segmented memory allows each table to grow or shrink independently of the other tables.

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**.

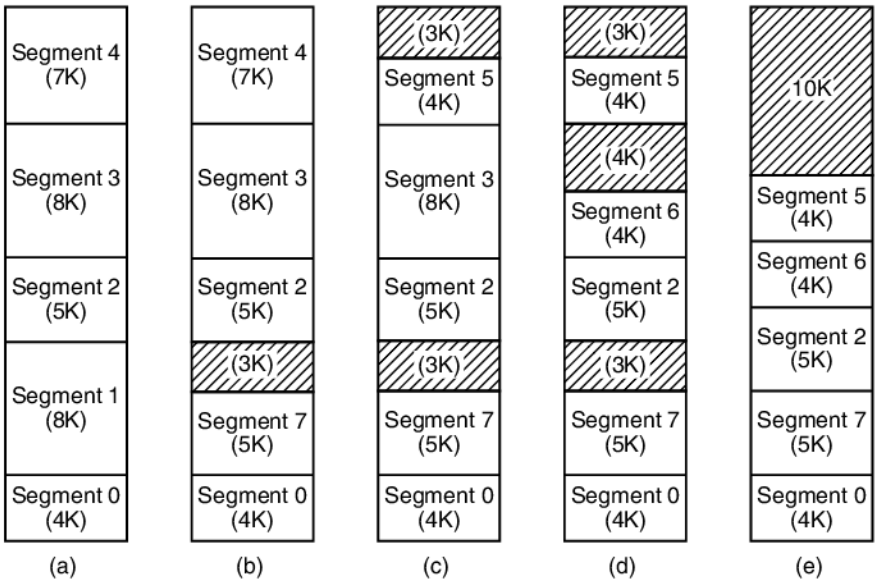

Figure 6-10. (a)–(d) Development of external fragmentation. (e) Removal of the external fragmentation by compaction.

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*.

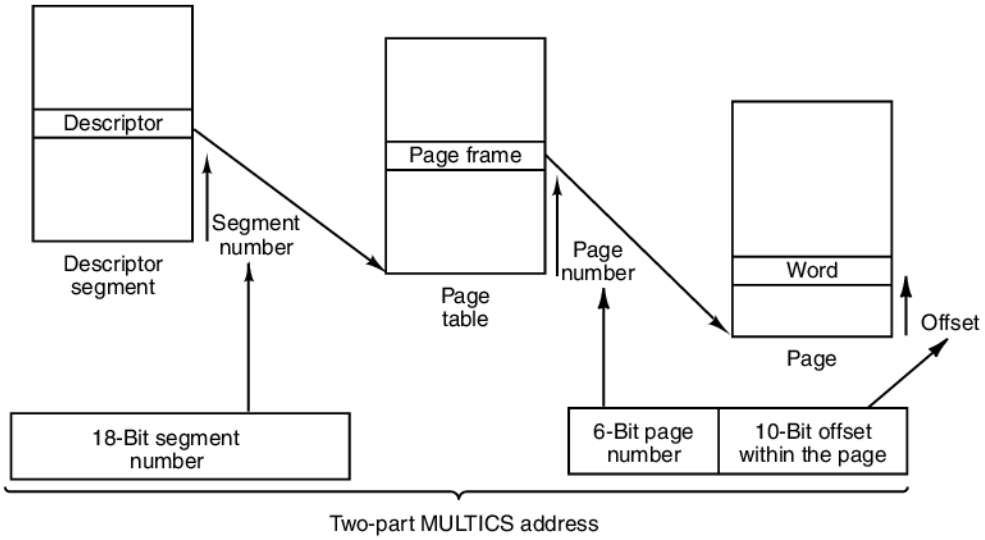

Figure 6-11. Conversion of a two-part MULTICS address into a main memory address.

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

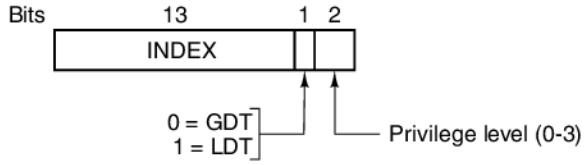

Figure 6-12. A Core i7 selector.

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.

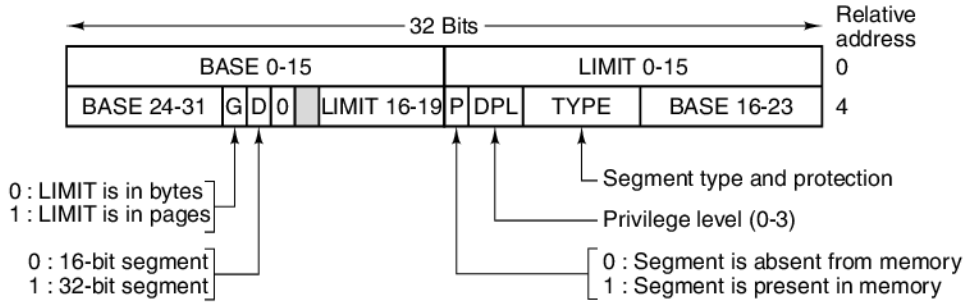

Figure 6-13. A Core i7 code segment descriptor. Data segments differ slightly.

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.

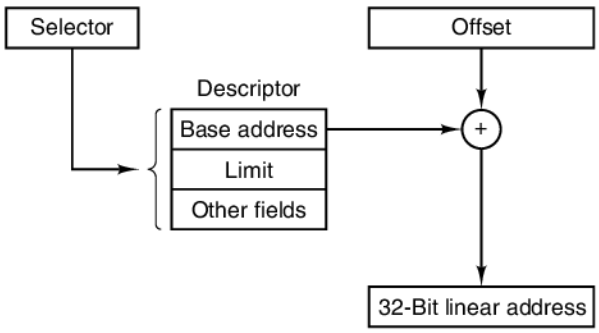

Figure 6-14. Conversion of a (selector, offset) pair to a linear address.

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.

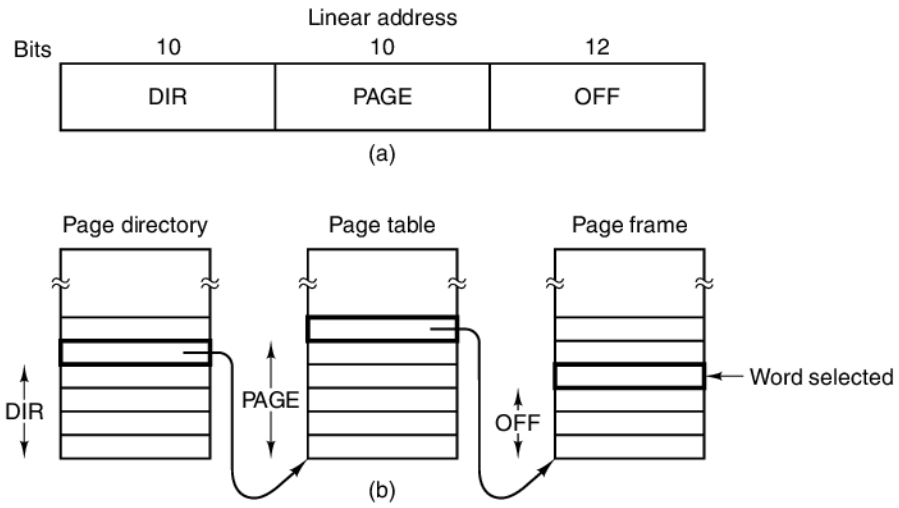

Figure 6-15. Mapping of a linear address onto a physical address.

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.

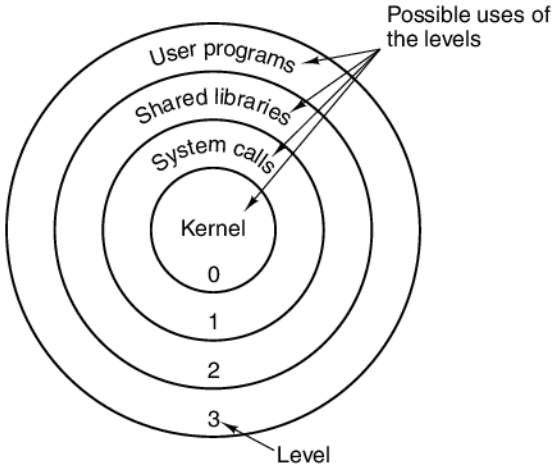

Figure 6-16. Protection on the Core i7.

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.

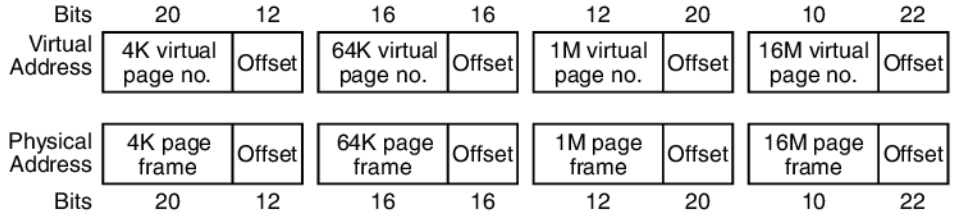

Figure 6-17. Virtual-to-physical mappings on the OMAP4430 ARM CPU.

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.

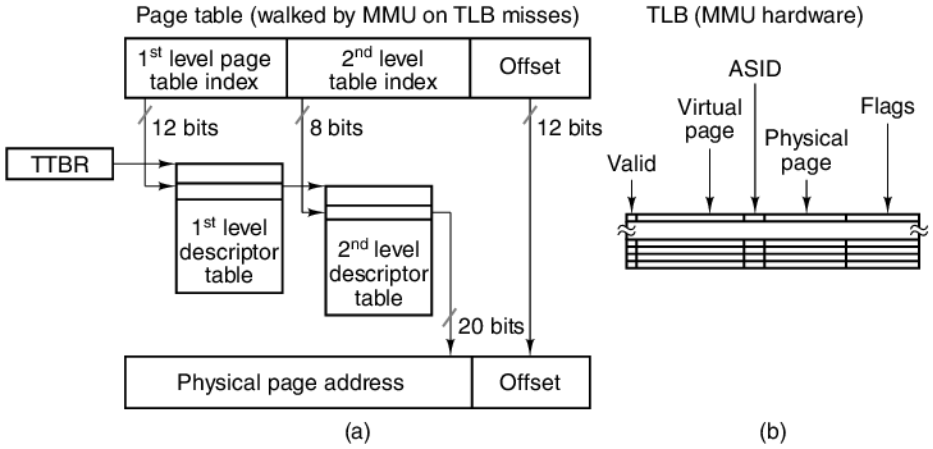

Figure 6-18. Data structures used in translating virtual addresses on the OMAP4430 ARM CPU. (a) Address translation table. (b) TLB.

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.

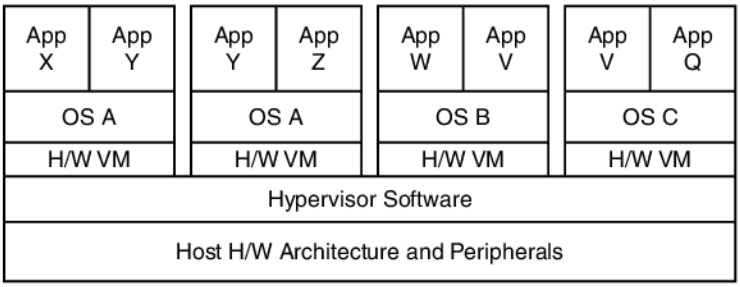

Figure 6-19. Hardware virtualization allows multiple operating systems to run simultaneously on the same host hardware. The hypervisor implements sharing of host memory and I/O devices.

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.

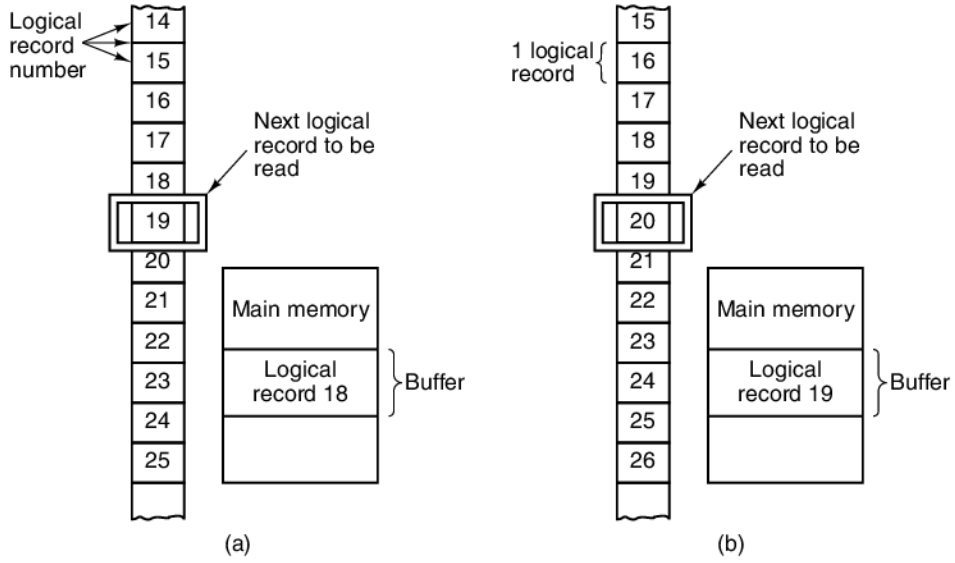

Figure 6-20. Reading a file consisting of logical records. (a) Before reading record 19. (b) After reading record 19.

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

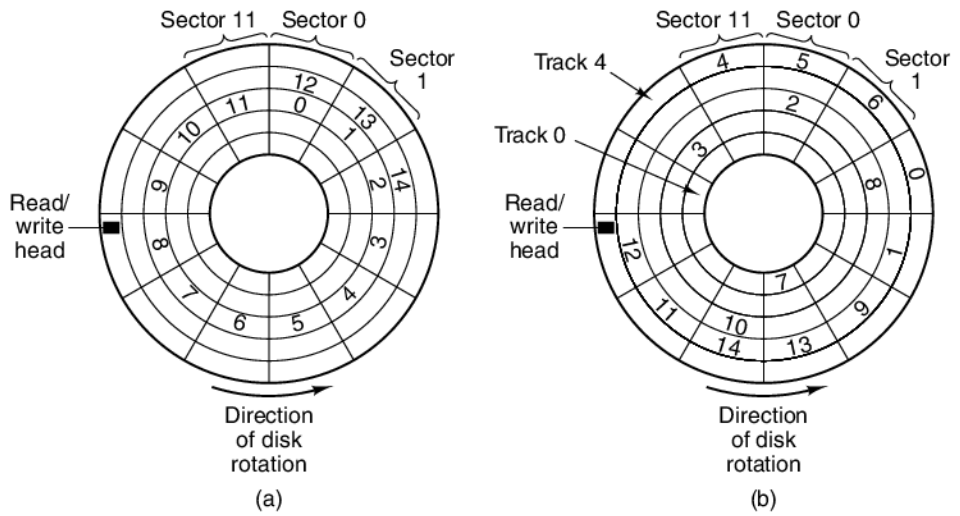

Figure 6-21. Disk allocation strategies. (a) A file in consecutive sectors. (b) A file not in consecutive sectors.

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

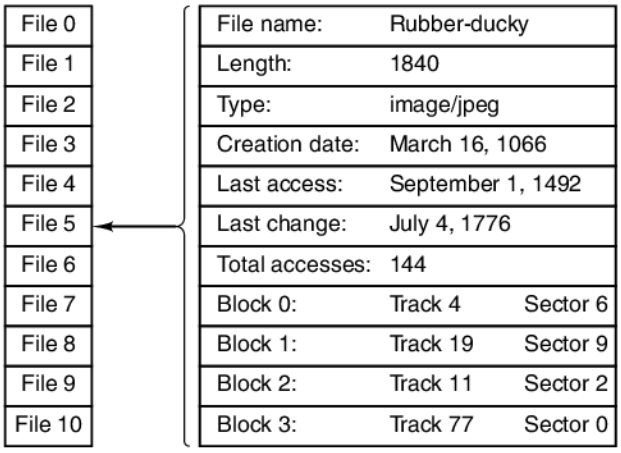

Figure 6-23. A user file directory and the contents of a typical entry in a file directory.

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.

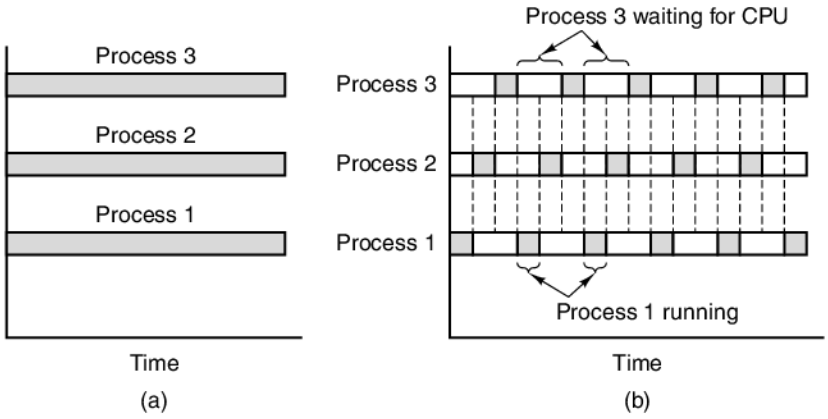

Figure 6-24. (a) True parallel processing with multiple CPUs. (b) Parallel processing simulated by switching one CPU among three processes.

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.

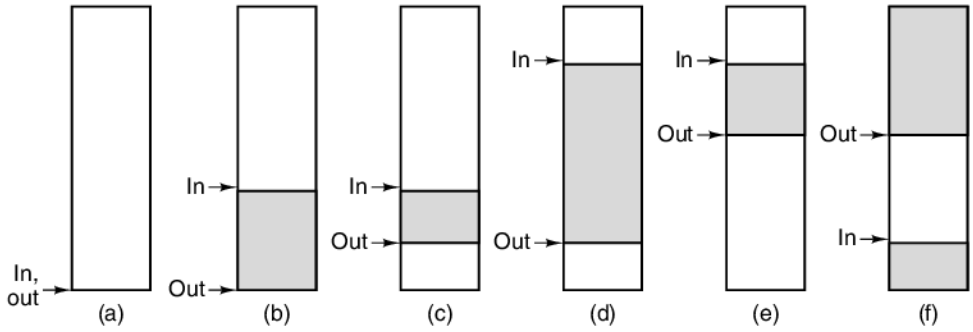

Figure 6-25. Use of a circular buffer.

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.

This solution uses Java threads to simulate parallel processes. With this solution we have a class *producer* and a class *consumer*, which are instantiated in the variables *p* and *c*, respectively. Each of these classes is derived from a base class *Thread*, which has a method *run*. The *run* method contains the code for the thread. When the *start* method of an object derived from *Thread* is invoked, a new thread is started.

Each thread is like a process, except that all threads within a single Java program run in the same address space. This feature is convenient for having them share a common buffer. If the computer has two or more CPUs, each thread can be scheduled on a different CPU, allowing parallelism. If there is only one CPU, the threads are timeshared on the same CPU. We will continue to refer to the producer and consumer as processes (since we are really interested in parallel processes), even though Java supports only parallel threads and not true parallel processes.

The utility function *next* allows *in* and *out* to be incremented easily, without having to write code to check for the wraparound condition every time. If the parameter to *next* is 98 or lower, the next-higher integer is returned. If, however, the parameter is 99, we have hit the end of the buffer, so 0 is returned.

We need a way for either process to put itself to sleep when it cannot continue. The Java designers understood the need for this ability and included the methods *suspend* (sleep) and *resume* (wakeup) in the *Thread* class right from the first version of Java. They are used in Fig. 6-26.

Now we come to the actual code for the producer and consumer. First, the producer generates a new prime in P1. Notice the use of *m.MAX_PRIME* here. The prefix *m.* is needed to indicate that we mean the *MAX_PRIME* defined in class *m*. For the same reason, this prefix is needed for *in*, *out*, *buffer*, and *next*, as well.

Then the producer checks (in P2) to see if *in* is one behind *out*. If it is (e.g., *in* = 62 and *out* = 63), the buffer is full and the producer goes to sleep by calling *suspend* in P2. If the buffer is not full, the new prime is inserted into the buffer (P3) and *in* is incremented (P4). If the new value of *in* is 1 ahead of *out* (P5) (e.g., *in* = 17 and *out* = 16), *in* and *out* must have been equal before *in* was incremented. The producer concludes that the buffer was empty and that the consumer was, and still is, sleeping. Therefore, the producer calls *resume* to wake the consumer up (P5). Finally, the producer begins looking for the next prime.

The consumer's program is structurally similar. First, a test is made (C1) to see if the buffer is empty. If it is, there is no work for the consumer to do, so it goes to sleep. If the buffer is not empty, it removes the next number to be printed (C2) and increments *out* (*C*3). If *out* is two positions ahead of *in* at this point (C4), it must have been one position ahead of *in* before it was just incremented. Because *in* = *out* − 1 is the "buffer full" condition, the producer must have been sleeping, and thus the consumer wakes it up with *resume*. Finally, the number is printed (C5) and the cycle repeats.

Unfortunately, this design contains a fatal flaw, as illustrated in Fig. 6-27. Remember that the two processes run asynchronously and at different, possibly

varying, speeds. Consider the case where only one number is left in the buffer, in entry 21, and *in* = 22 and *out* = 21, as shown in Fig. 6-27(a). The producer is at statement P1 looking for a prime and the consumer is busy at C5 printing out the number in position 20. The consumer finishes printing the number, makes the test at C1, and takes the last number out of the buffer at C2. It then increments *out*. At this instant, both *in* and *out* have the value 22. The consumer prints the number and then goes to C1, where it fetches *in* and *out* from memory in order to compare them, as shown in Fig. 6-27(b).

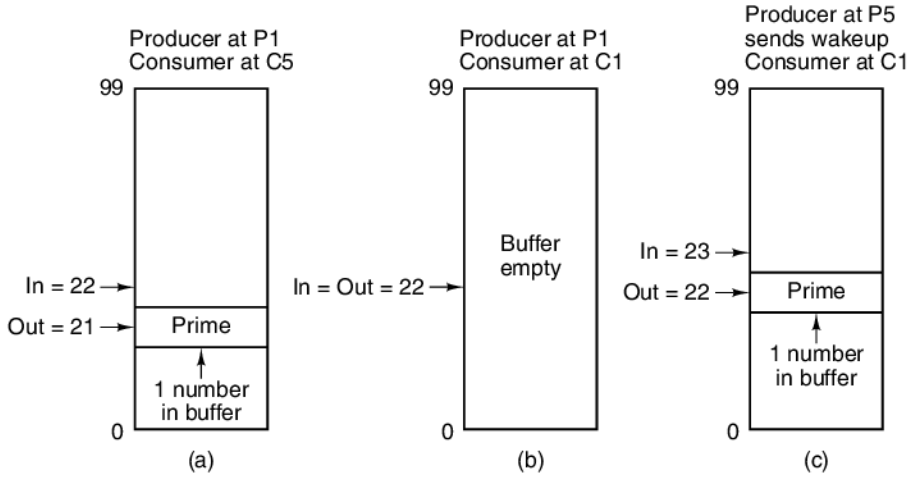

Figure 6-27. Failure of the producer-consumer communication mechanism.

At this very moment, after the consumer has fetched *in* and *out* but before it has compared them, the producer finds the next prime. It puts the prime into the buffer at P3 and increments *in* at P4. Now *in* = 23 and *out* = 22. At P5 the producer discovers that *in* = *next*(*out*). In other words, *in* is one higher than *out*, signifying that there is now one item in the buffer. The producer therefore (incorrectly) concludes that the consumer must be sleeping, so it sends a wakeup signal (i.e., calls *resume*), as shown in Fig. 6-27(c). Of course, the consumer is still awake, so the wakeup signal is lost. The producer begins looking for the next prime.

At this point in time the consumer continues. It has already fetched *in* and *out* from memory before the producer put the last number in the buffer. Because they both have the value 22, the consumer goes to sleep. Now the producer finds another prime. It checks the pointers and finds *in* = 24 and *out* = 22, therefore it assumes that there are two numbers in the buffer (true) and that the consumer is awake (false). The producer continues looping. Eventually, it fills the buffer and goes to sleep. Now both processes are sleeping and will remain so forever.

The big difficulty here is that between the moment when the consumer fetched *in* and *out* and the time it went to sleep, the producer snuck in, discovered that

in = *out* + 1, assumed that the consumer was sleeping (which it was not yet), and sent a wakeup signal that was lost because the consumer was still awake. This difficulty is known as a **race condition**, because the method's success depends on who wins the race to test *in* and *out* after *out* is incremented.

The problem of race conditions is well known. In fact it is so serious that several years after Java was introduced, Sun changed the *Thread* class and deprecated the *suspend* and *resume* calls because they led to race conditions so often. The solution offered was a language-based solution, but since we are studying operating systems here, we will discuss a different solution, one supported by many operating systems, including UNIX and Windows 7.

6.4.3 Process Synchronization Using Semaphores

The race condition can be solved in at least two ways. One solution consists of equipping each process with a "wakeup waiting bit." Whenever a wakeup is sent to a process that is still running, its wakeup waiting bit is set. Whenever the process goes to sleep when the wakeup waiting bit is set, it is immediately restarted and the wakeup waiting bit is cleared. The wakeup waiting bit stores the superfluous wakeup signal for future use.

Although this method solves the race condition when there are only two processes, it fails in the general case of n communicating processes because as many as $n - 1$ wakeups may have to be saved. Of course, each process could be equipped with $n - 1$ wakeup waiting bits to allow it to count to $n - 1$ in the unary system, but this solution is rather clumsy.

Dijkstra (1968b) proposed a more general solution to the problem of synchronizing parallel processes. Somewhere in the memory are some nonnegative integer variables called **semaphores**. Two system calls that operate on semaphores, up and down, are provided by the operating system. Up adds 1 to a semaphore and down subtracts 1 from a semaphore.

If a down operation is performed on a semaphore that is currently greater than 0, the semaphore is decremented by 1 and the process doing the down continues. If, however, the semaphore is 0, the down cannot complete; the process doing the down is put to sleep and remains asleep until the other process performs an up on that semaphore. Usually sleeping processes are strung together in a queue to keep track of them.

The up instruction checks to see if the semaphore is 0. If it is and the other process is sleeping on it, the semaphore is increased by 1. The sleeping process can then complete the down operation that suspended it, resetting the semaphore to 0 and allowing both processes to continue. An up instruction on a nonzero semaphore simply increases it by 1. In essence, a semaphore provides a counter to store wakeups for future use, so that they will not be lost. An essential property of semaphore instructions is that once a process has initiated an instruction on a semaphore, no other process may access the semaphore until the first one has either

completed its instruction or been suspended trying to perform a down on a 0. Figure 6-28 summarizes the essential properties of the up and down system calls.

Instruction	Semaphore = 0	Semaphore > 0
Up	Semaphore = semaphore + 1; if the other process was halted attempting to complete a down instruction on this semaphore, it may now complete the down and continue running	Semaphore = semaphore + 1
Down	Process halts until the other process ups this semaphore	Semaphore = semaphore – 1

Figure 6-28. The effect of a semaphore operation.

As mentioned above, Java has a language-based solution for dealing with race conditions, and we are discussing operating systems now. Thus we need a way to express semaphore usage in Java, even though it is not in the language or the standard classes. We will do this by assuming that two native methods have been written, *up* and *down*, which make the up and down system calls, respectively. By calling these with ordinary integers as parameters, we have a way to express the use of semaphores in Java programs.

Figure 6-29 shows how the race condition can be eliminated through the use of semaphores. Two semaphores are added to the *m* class, *available*, which is initially 100 (the buffer size), and *filled*, which is initially 0. The producer starts executing at P1 in Fig. 6-29 and the consumer starts executing at C1 as before. The *down* call on *filled* halts the consumer processor immediately. When the producer has found the first prime, it calls *down* with *available* as parameter, setting *available* to 99. At P5 it calls *up* with *filled* as parameter, making *filled* 1. This action releases the consumer, which is now able to complete its suspended *down* call. At this point, *filled* is 0 and both processes are running.

Let us now reexamine the race condition. At a certain point in time, *in* = 22, *out* = 21, the producer is at P1, and the consumer is at C5. The consumer finishes what it was doing and gets to C1 where it calls *down* on *filled*, which had the value 1 before the call and 0 after it. The consumer then takes the last number out of the buffer and ups *available*, making it 100. The consumer prints the number and goes to C1. Just before the consumer can call *down*, the producer finds the next prime and in quick succession executes statements P2, P3, and P4.

At this point, *filled* is 0. The producer is about to up it and the consumer is about to call *down*. If the consumer executes its instruction first, it will be suspended until the producer releases it (by calling *up*). On the other hand, if the producer goes first, the semaphore will be set to 1 and the consumer will not be suspended at all. In both cases, no wakeup is lost. This, of course, was our goal in introducing semaphores in the first place.

The essential property of the semaphore operations is that they are indivisible. Once a semaphore operation has been initiated, no other running process can use

the semaphore until the first process has either completed the operation or been suspended trying. Furthermore, with semaphores, no wakeups are lost. In contrast, the if statements of Figure 6-26 are not indivisible. Between the evaluation of the condition and the execution of the selected statement, another process can send a wakeup signal.

In effect the problem of process synchronization has been eliminated by declaring the up and down system calls made by *up* and *down* to be indivisible. In order for these operations to be indivisible, the operating system must prohibit two or more processes from using the same semaphore at the same time. At the very least, once an up or down system call has been made, no other user code will be run until the call has been completed. On single-processor systems, semaphores are sometimes implemented by disabling interrupts during semaphore operations. On multiple-processor systems, this trick does not work.

Synchronization using semaphores is a technique that works for arbitrarily many processes. Several processes may be sleeping, attempting to complete a down system call on the same semaphore. When some other process finally performs an up on that semaphore, one of the waiting processes is allowed to complete its down and continue running. The semaphore value remains 0 and the other processes continue waiting.

An analogy may make the nature of semaphores clearer. Imagine a picnic with 20 volleyball teams divided into 10 games (processes), each playing on its own court, and a large basket (the semaphore) for the volleyballs. Unfortunately, only seven volleyballs are available. At any instant, there are between zero and seven volleyballs in the basket (the semaphore has a value between 0 and 7). Putting a ball in the basket is an up because it increases the value of the semaphore; taking a ball out of the basket is a down because it decreases the value.

At the start of the picnic, each court sends a player to the basket to get a volleyball. Seven of them successfully manage to get a volleyball (complete the down); three are forced to wait for a volleyball (i.e., fail to complete the down). Their games are suspended temporarily. Eventually, one of the other games finishes and puts a ball into the basket (executes an up). This operation allows one of the three players waiting around the basket to get a ball (complete an unfinished down), allowing one game to continue. The other two games remain suspended until two more balls are put into the basket. When two more balls come back (two more ups are executed), the last two games can proceed.

6.5 EXAMPLE OPERATING SYSTEMS

In this section we will continue discussing our example systems, the Core i7 and the OMAP4430 ARM CPU. For each one we will look at an operating system used on that processor. For the Core i7 we will use Windows; for the OMAP4430 ARM CPU we will use UNIX. Since UNIX is simpler and in many ways more

```
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 int filled = 0, available = 100;             // semaphores

   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
   native void up(int s); native void down(int s);            // methods on semaphores
   public void run( ) {                                       // producer code
      long prime = 2;                                         // scratch variable

      while (prime < m.MAX_PRIME) {
         prime = next_prime(prime);                           // statement P1
         down(m.available);                                   // statement P2
         m.buffer[m.in] = prime;                              // statement P3
         m.in = m.next(m.in);                                 // statement P4
         up(m.filled);                                        // statement P5
      }
   }

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

class consumer extends Thread {                               // consumer class
   native void up(int s); native void down(int s);            // methods on semaphores
   public void run( ) {                                       // consumer code
      long emirp = 2;                                         // scratch variable

      while (emirp < m.MAX_PRIME) {
         down(m.filled);                                      // statement C1
         emirp = m.buffer[m.out];                             // statement C2
         m.out = m.next(m.out);                               // statement C3
         up(m.available);                                     // statement C4
         System.out.println(emirp);                           // statement C5
      }
   }
}
```

Figure 6-29. Parallel processing using semaphores.

elegant, we will begin with it. Also, UNIX was designed and implemented first and had a major influence on Windows 7, so this order makes more sense than the reverse.

6.5.1 Introduction

In this section we will give a brief introduction to our two example operating systems, UNIX and Windows 7, focusing on the history, structure, and system calls.

UNIX

UNIX was developed at Bell Labs in the early 1970s. The first version was written by Ken Thompson in assembler for the PDP-7 minicomputer. This was soon followed by a version for the PDP-11, written in a new language called C that was devised and implemented by Dennis Ritchie. In 1974, Ritchie and his colleague Ken Thompson published a landmark paper about UNIX (Ritchie and Thompson, 1974). For the work described in this paper they were later given the prestigious ACM Turing Award (Ritchie, 1984, Thompson, 1984). The publication of this paper stimulated many universities to ask Bell Labs for a copy of UNIX. Since Bell Labs' parent company, AT&T, was a regulated monopoly at the time and was not permitted to be in the computer business, it had no objection to licensing UNIX to universities for a modest fee.

In one of those coincidences that often shape history, the PDP-11 was the computer of choice at nearly all university computer science departments, and the operating systems that came with the PDP-11 were widely regarded as being dreadful by professors and students alike. UNIX quickly filled the void, not in the least because it was supplied with the complete source code, so people could, and did, tinker with it endlessly.

One of the many universities that acquired UNIX early on was the University of California at Berkeley. Because the complete source code was available, Berkeley was able to modify the system substantially. Foremost among the changes was a port to the VAX minicomputer and the addition of paged virtual memory, the extension of file names from 14 characters to 255 characters, and the inclusion of the TCP/IP networking protocol, which is now used on the Internet (largely due to the fact that it was in Berkeley UNIX).

While Berkeley was making all these changes, AT&T itself continued to develop UNIX, leading to System III in 1982 and then System V in 1984. By the late 1980s, two different, and quite incompatible, versions of UNIX were in widespread use: Berkeley UNIX and System V. This split in the UNIX world, together with the fact that there were no standards for binary program formats, greatly inhibited the commercial success of UNIX because it was impossible for software vendors to write and package UNIX programs with the expectation that they would run on any UNIX system (as was routinely done then with MS-DOS). After much

bickering, a standard called **POSIX** (**Portable Operating System-IX**) was created by the IEEE Standards Board. POSIX is also known by its IEEE Standards number, P1003. It later became an International Standard.

The standard is divided into many parts, each one covering a different area of UNIX. The first part, P1003.1, defines the system calls; the second part, P1003.2, defines the basic utility programs, and so on. The P1003.1 standard defines about 60 system calls that all conformant systems must support. These are the basic calls for reading and writing files, creating new processes, and so on. Nearly all UNIX systems now support the P1003.1 system calls. However many UNIX systems also support extra system calls, especially those defined by System V and/or those in Berkeley UNIX. Typically these add up to 200 system calls.

In 1987, one author of this book (Tanenbaum) released the source code for a tiny version of UNIX, called MINIX, for use at universities (Tanenbaum, 1987). One of the students who studied MINIX at his university in Helsinki and ran it on his home PC was Linus Torvalds. After becoming thoroughly familiar with MINIX, Torvalds decided to write his own clone of MINIX, which was called Linux and has become quite popular.

Many operating systems running today on ARM platforms are based on Linux. Both MINIX and Linux are POSIX conformant, and nearly everything said about UNIX in this chapter also applies to them unless stated otherwise.

A rough breakdown of the Linux system calls by category is given in Fig. 6-30. The file- and directory-management system calls are the largest and the most important categories. Linux is mostly POSIX P1003.1 compliant, although the developers did deviate from the specification in some areas. In general, however, it is not difficult to get POSIX-compliant programs to build and run on Linux.

Category	Some examples
File management	Open, read, write, close, and lock files
Directory management	Create and delete directories; move files around
Process management	Spawn, terminate, trace, and signal processes
Memory management	Share memory among processes; protect pages
Getting/setting parameters	Get user, group, process ID; set priority
Dates and times	Set file access times; use interval timer; profile execution
Networking	Establish/accept connection; send/receive message
Miscellaneous	Enable accounting; manipulate disk quotas; reboot the system

Figure 6-30. A rough breakdown of the UNIX system calls.

One area that is largely due to Berkeley UNIX rather than System V is networking. Berkeley invented the concept of a **socket**, which is the endpoint of a network connection. The four-pin wall plugs to which telephones can be connected served as the model for this concept. It is possible for a UNIX process to create a socket, attach to it, and establish a connection to a socket on a distant

machine. Over this connection it can then exchange data in both directions, typically using the TCP/IP protocol. Since networking technology has been in UNIX for decades and is very stable and mature, a substantial fraction of the servers on the Internet run UNIX.

Since there are many implementations of UNIX, it is difficult to say much about the structure of the operating system since each one is somewhat different from all the others. However, in general, Fig. 6-31 applies to most of them. At the bottom, there is a layer of device drivers that shield the file system from the bare hardware. Originally, each device driver was written as an independent entity, separate from all the others. This arrangement led to a lot of duplicated effort, since many drivers must deal with flow control, error handling, priorities, separating data from control, and so on. This observation led Dennis Ritchie to develop a framework called **streams** for writing drivers in a modular way. With a stream, it is possible to establish a two-way connection from a user process to a hardware device and to insert one or more modules along the path. The user process pushes data into the stream, which then is processed or transformed by each module until it gets to the hardware. The inverse processing occurs for incoming data.

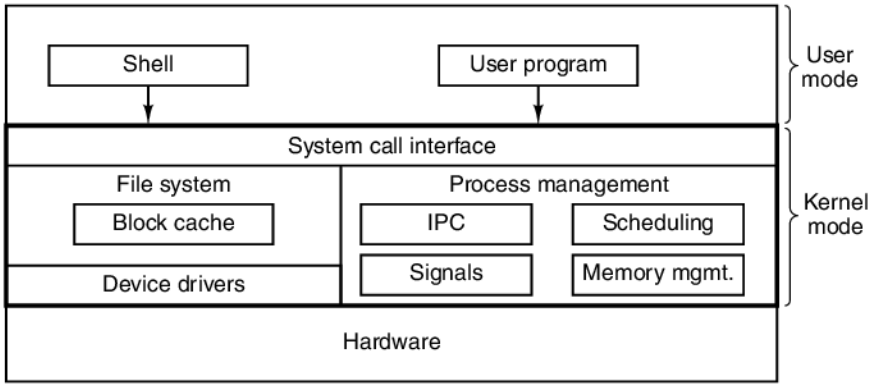

Figure 6-31. The structure of a typical UNIX system.

On top of the device drivers comes the file system. It manages file names, directories, disk block allocation, protection, and much more. Part of the file system is a **block cache**, for holding the blocks most recently read in from disk, in case they are needed again soon. A variety of file systems have been used over the years, including the Berkeley fast file system (McKusick et al., 1984), and log-structured file systems (Rosenblum and Ousterhout, 1991, and Seltzer et al., 1993).

The other part of the UNIX kernel is the process-management portion. Among its various other functions, it handles IPC (InterProcess Communication), which allows processes to communicate with one another and synchronize to avoid race conditions. A variety of mechanisms are provided. The process-management code

also handles process scheduling, which is based on priorities. Signals, which are a form of (asynchronous) software interrupt, are also managed here. Finally, memory management is done here as well. Most UNIX systems support demand-paged virtual memory, sometimes with a few extra features, such as the ability of multiple processes to share common regions of address space.

From its inception, UNIX has tried to be a small system, in order to enhance reliability and performance. The first versions of UNIX were entirely text based, using terminals that could display 24 or 25 lines of 80 ASCII characters. The user interface was handled by a user-level program called the **shell**, which offered a command-line interface. Since the shell was not part of the kernel, adding new shells to UNIX was easy, and over time a number of increasingly sophisticated ones were invented.

Later on, when graphics terminals came into existence, a windowing system for UNIX, called **X Windows**, was developed at M.I.T. Still later, a full-fledged **GUI** (**Graphical User Interface**), called **Motif**, was put on top of X Windows. These GUIs eventually developed into full-blown desktop environments with beautifully rendered window management, productivity tools, and utilities. Examples of these desktop environments include GNOME and KDE. In keeping with the UNIX philosophy of having a small kernel, nearly all the code of X Windows and its accompanying GUIs run in user mode, outside the kernel.

Windows 7

When the original IBM PC was launched in 1981, it came equipped with a 16-bit real-mode, single-user, command-line-oriented operating system called MS-DOS 1.0 This operating system consisted of 8 KB of memory-resident code. Two years later, a much more powerful 24-KB system, MS-DOS 2.0, appeared. It contained a command-line processor (shell), with a number of features borrowed from UNIX. When IBM released the 286-based PC/AT in 1984, it came equipped with MS-DOS 3.0, by now 36 KB. Over the years, MS-DOS continued to acquire new features, but it was still a command-line-oriented system.

Inspired by the success of the Apple Macintosh, Microsoft decided to give MS-DOS a graphical user interface that it called **Windows**. The first three versions of Windows, culminating in Windows 3.x, were not true operating systems but graphical user interfaces on top of MS-DOS, which was still in control of the machine. All programs ran in the same address space and a bug in any one of them could bring the whole system to a grinding halt.

The release of Windows 95 in 1995 still did not eliminate MS-DOS, although it introduced a new version, 7.0. Together, Windows 95 and MS-DOS 7.0 contained most of the features of a full-blown operating system, including virtual memory, process management, and multiprogramming. However, Windows 95 was not a full 32-bit program. It contained large chunks of old 16-bit code (as well as some 32-bit code) and still used the MS-DOS file system, with nearly all its limitations.

The only major changes to the file system were the addition of long file names in place of the 8 + 3 character file names allowed in MS-DOS and the ability to have more than 65,536 blocks on a disk.

Even with the release of Windows 98 in 1998, MS-DOS was still there (now called version 7.1) and running 16-bit code. Although a bit more functionality migrated from the MS-DOS part to the Windows part, and a disk layout suitable for larger disks was now standard, under the hood, Windows 98 was not very different from Windows 95. The main difference was the user interface, which integrated the desktop, the Internet, and to some extent, even television more closely. It was precisely this integration that attracted the attention of the U.S. Dept. of Justice, which then sued Microsoft claiming that it was an illegal monopoly. Windows 98 was followed by the short-lived Windows Millennium Edition (ME), which was a slightly improved Windows 98.

While all these developments were going on, Microsoft was also busy with a completely new 32-bit operating system being written from the ground up. This new system was called **Windows New Technology**, or **Windows NT**. It was initially hyped as the replacement for all other operating systems for Intel-based PCs (as well as the MIPS PowerPC chips), but it was somewhat slow to catch on and was later redirected to the upper end of the market, where it found a niche on large servers. The second version of NT was called Windows 2000 and became the mainstream version, also for the desktop market. The successor to Windows 2000 was Windows XP, but the changes here were relatively minor (better backward compatibility and a few more features). In 2007, the followup Windows Vista was released. Vista implemented many graphical enhancements over Windows XP, and it added many new user applications, such as a media center. Vista's adoption was slow because of its poor performance and high resource demands. A mere two years later, Windows 7 was released, which by all accounts is a tuned-up version of Windows Vista. Windows 7 runs much better on older hardware, and it requires significantly less hardware resources.

Windows 7 is sold in six different versions. Three are for home users in various countries, two are aimed at business users, and one combines all the features of all versions. These versions are nearly identical and differ primarily in focus, advanced features, and optimizations made. We will focus on the core features and not make any further distinction between these versions.

Before getting into the interface Windows 7 presents to programmers, let us take a very quick look at its internal structure, which is illustrated in Fig. 6-32. It consists of a number of modules that are structured in layers and work together to implement the operating system. Each module has some particular function and a well-defined interface to the other modules. Nearly all the modules are written in C, although part of the graphics device interface is written in C++ and tiny bits of the lowest layers are written in assembly language.

At the bottom is a thin layer called the **hardware abstraction layer**. Its job is to present the rest of the operating system with abstract hardware devices, devoid

of the warts and idiosyncracies with which real hardware is so richly endowed. Among the devices modeled are off-chip caches, timers, I/O buses, interrupt controllers, and DMA controllers. By exposing these to the rest of the operating system in idealized form, it becomes easier to port Windows 7 to other hardware platforms, since most of the modifications required are concentrated in one place.

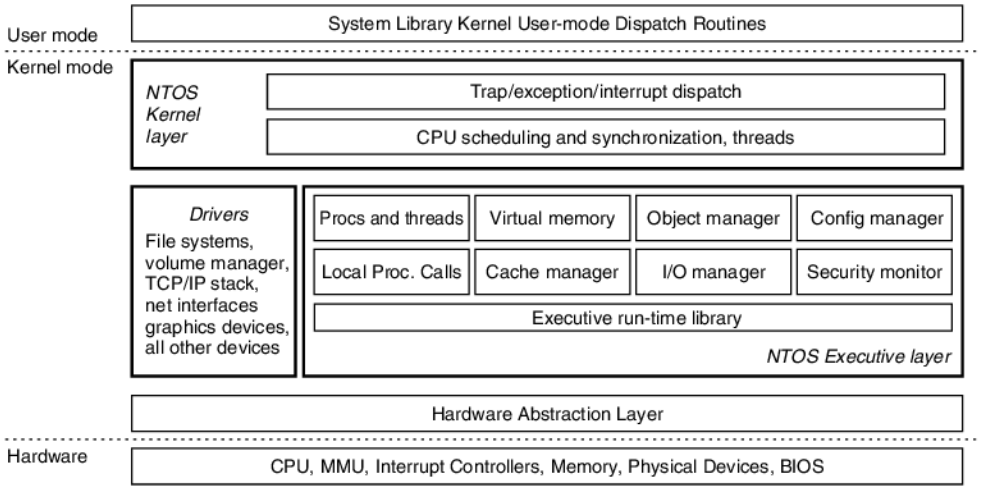

Figure 6-32. The structure of Windows 7.

Above the HAL, the code is divided into two major parts, the **NTOS executive** and the **Windows drivers**, which includes the file systems, networking, and graphics code. On top of that is the kernel layer. All of this code runs in protected kernel mode.

The executive manages the fundamental abstractions used in Windows 7, including threads, processes, virtual memory, kernel objects, and configurations. Also here are the managers for local procedure calls, the file cache, I/O, and security.

The kernel layer handles trap and exception handling, as well as scheduling and synchronization.

Outside the kernel are the user programs and the system library used to interface to the operating system. In contrast to UNIX systems, Microsoft does not encourage user programs to make direct system calls. Instead they are expected to call procedures in the library. To provide standardization across different versions of Windows (e.g., XP, Vista, and Windows 7), Microsoft defined a set of calls called the **Win32 API** (**Application Programming Interface**). These are library procedures that either make system calls to get the work done, or, in some case, do the work right in the user-space library procedure. Although many Windows 7 library calls have been added since Win32 was defined, these are the core calls and it is them we will focus on. Later, when Windows was ported to 64-bit machines,

Microsoft changed the name of Win32 to cover both the 32-bit and 64-bit versions, but for our purposes, looking at the 32-bit version is sufficient.

The Win32 API philosophy is completely different from the UNIX philosophy. In the latter, the system calls are all publicly known and form a minimal interface: removing even one of them would reduce the functionality of the operating system. The Win32 philosophy is to provide a very comprehensive interface, often with three or four ways of doing the same thing, and including many functions that clearly should not be (and are not) system calls, such as an API call to copy an entire file.

Many Win32 API calls create kernel objects of one kind or another, including files, processes, threads, pipes, etc. Every call creating a kernel object returns a result called a **handle** to the called. This handle can be subsequently used to perform operations on the object. Handles are specific to the process that created the object referred to by the handle. They cannot be passed directly to another process and used there (just as UNIX file descriptors cannot be passed to other processes and used there). However, under certain circumstances, it is possible to duplicate a handle and pass it to other processes in a protected way, allowing them controlled access to objects belonging to other processes. Every object can have a security descriptor associated with it, telling in detail who may and may not perform what kinds of operations on the object.

Windows 7 is sometimes said to be object oriented because the only way to manipulate kernel objects is by invoking methods (API functions) on their handles, which are returned when the objects are created. On the other hand, it lacks some of the most basic properties of object-oriented systems such as inheritance and polymorphism.

6.5.2 Examples of Virtual Memory

In this section we will look at virtual memory in both UNIX and Windows 7. For the most part, they are fairly similar from the programmer's point of view.

UNIX Virtual Memory

The UNIX memory model is simple. Each process has three segments: code, data, and stack, as illustrated in Fig. 6-33. In a machine with a single, linear address space, the code is generally placed near the bottom of memory, followed by the data. The stack is placed at the top of memory. The code size is fixed, but the data and stack may each grow, in opposite directions. This model is easy to implement on almost any machine and is the model used by Linux variants that run on OMAP4430 ARM CPUs.

Furthermore, if the machine has paging, the entire address space can be paged, without user programs even being aware of it. The only thing they notice is that it

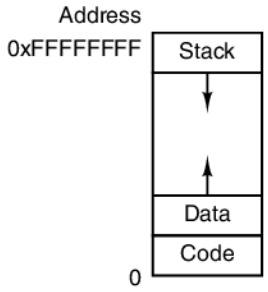

Figure 6-33. The address space of a single UNIX process.

is permitted to have programs larger than the machine's physical memory. UNIX systems that do not have paging generally swap entire processes between memory and disk to allow an arbitrarily large number of processes to be timeshared.

For Berkeley UNIX, the above description (demand-paged virtual memory) is basically the entire story. However, System V (and also Linux) include several features that allow users to manage their virtual memory in more sophisticated ways. Most important of these is the ability of a process to map a (portion of a) file onto part of its address space. For example, if a 12-KB file is mapped at virtual address 144K, a read to the word at address 144 KB reads the first word of the file. In this way file I/O can be done without making system calls. Since some files may exceed the size of the virtual address space, it is also possible to map in only a portion of a file instead of the whole file. The mapping is done by first opening the file and getting back a file descriptor, *fd*, which is used to identify the file to be mapped. Then the process makes a call

```
paddr = mmap(virtual_address, length, protection, flags, fd, file_offset)
```

which maps *length* bytes starting at *file_offset* in the file onto the virtual address space starting at *virtual_address*. Alternatively, the *flags* parameter can be set to ask the system to choose a virtual address, which it then returns as *paddr*. The mapped region must be an integral number of pages and aligned at a page boundary. The *protection* parameter can specify any combination of read, write, or execute permission. The mapping can be removed later with unmap.

Multiple processes can map onto the same file at the same time. Two options are provided for sharing. In the first one, all the pages are shared, so writes done by one process are visible to all the others. This option provides a high-bandwidth communication path between processes. The other option shares the pages as long as no process modifies them. However, as soon as any process attempts to write on a page, it gets a protection fault, which causes the operating system to give it a private copy of the page to write on. This scheme, known as **copy on write**, is used when each of multiple processes needs the illusion it is the only one mapped onto a file. In this model the sharing is an optimization, not part of the semantics.

Windows 7 Virtual Memory

In Windows 7, every user process has its own virtual address space. In the 32-bit version of Windows 7, virtual addresses are 32 bits long, so each process has 4 GB of virtual address space. The lower 2 GB are available for the process' code and data; the upper 2 GB allow (limited) access to kernel memory, except in Server versions of Windows, in which the split can be 3 GB for the user and 1 GB for the kernel. The virtual address space is demand paged, with a fixed page size (4 KB on the Core i7). The address space for the 64-bit version of Windows 7 is similar, however, the code and data space is the lower 8 terabytes of the virtual address space.

Each virtual page can be in one of three states: free, reserved, or committed. A **free page** is not currently in use and a reference to it causes a page fault. When a process is started, all of its pages are in free state until the program and initial data are mapped into its address space. Once code or data is mapped onto a page, the page is said to be **committed**. A reference to a committed page is mapped using the virtual memory hardware and succeeds if the page is in main memory. If the page is not in main memory, a page fault occurs and the operating system finds and brings in the page from disk. A virtual page can also be in **reserved** state, meaning it is not available for being mapped until the reservation is explicitly removed. Reserved pages are used when a run of consecutive pages may be needed in the future, such as for the stack. In addition to the free, reserved, and committed attributes, pages also have other attributes, such as being readable, writable, and executable. The top 64 KB and bottom 64 KB of memory are always free, to catch pointer errors (uninitialized pointers are often 0 or −1).

Each committed page has a shadow page on the disk where it is kept when it is not in main memory. Free and reserved pages do not have shadow pages, so references to them cause page faults (the system cannot bring in a page from disk if there is no page on disk). The shadow pages on the disk are arranged into one or more paging files. The operating system keeps track of which virtual page maps onto which part of which paging file. For (execute only) program text, the executable binary file contains the shadow pages; for data pages, special paging files are used.

Windows 7, like System V, allows files to be mapped directly onto regions of the virtual address spaces (i.e., runs of pages). Once a file has been mapped onto the address space, it can be read or written using ordinary memory references.

Memory-mapped files are implemented in the same way as other committed pages, only the shadow pages can be in the disk file instead of in the paging file. As a result, when a file is mapped in, the version in memory may not be identical to the disk version (due to recent writes to the virtual address space). However, when the file is unmapped or is flushed, the disk version is updated from memory.

Windows 7 explicitly allows two or more processes to map in the same file at the same time, possibly even at different virtual addresses. By reading and writing

memory words, the processes can now communicate with each other and pass data back and forth at very high bandwidth, since no copying is required. Different processes may have different access permissions. Since all the processes using a mapped file share the same pages, changes made by one of them are immediately visible to all the others, even if the disk file has not yet been updated.

The Win32 API contains a number of functions that allow a process to manage its virtual memory explicitly. The most important of these functions are listed in Fig. 6-34. All of them operate on a region consisting of either a single page or a sequence of two or more pages that are consecutive in the virtual address space.

API function	Meaning
VirtualAlloc	Reserve or commit a region
VirtualFree	Release or decommit a region
VirtualProtect	Change the read/write/execute protection on a region
VirtualQuery	Inquire about the status of a region
VirtualLock	Make a region memory resident (i.e., disable paging for it)
VirtualUnlock	Make a region pageable in the usual way
CreateFileMapping	Create a file-mapping object and (optionally) assign it a name
MapViewOfFile	Map (part of) a file into the address space
UnmapViewOfFile	Remove a mapped file from the address space
OpenFileMapping	Open a previously created file-mapping object

Figure 6-34. The principal Windows 7 API calls for managing virtual memory

The first four API functions are self-explanatory. The next two give a process the ability to hardwire some number of pages in memory so they will not be paged out and to undo this property. A real-time program might need this ability, for example. Only programs run on behalf of the system administrator may pin pages in memory. And a limit is enforced by the operating system to prevent even these processes from getting too greedy. Although not shown in Fig. 6-34, Windows 7 also has API functions to allow a process to access the virtual memory of a different process over which it has been given control (i.e., for which it has a handle).

The last four API functions listed are for managing memory-mapped files. To map a file, a file-mapping object must first be created, with CreateFileMapping. This function returns a handle to the file-mapping object and optionally enters a name for it into the file system so another process can use it. The next two functions map and unmap files, respectively. A mapped file is (part of) a disk file that can be read from or written to just by accessing the virtual address space, with no explicit I/O. The last one can be used by a process to map in a file currently also mapped in by a different process. In this way, two or more processes can share regions of their address spaces.

These API functions are the basic ones upon which the rest of the memory management system is constructed. For example, there are API functions for

allocating and freeing data structures on one or more heaps. Heaps are used for storing data structures that are dynamically created and destroyed. The heaps are not garbage collected by the operating system, so it is up to language run-time systems or user software to free blocks of virtual memory that are no longer in use. (Garbage collection is the automatic removal of unused data structures.) Heap usage in Windows 7 is similar to the use of the *malloc* function in UNIX systems, except that there can be multiple independently managed heaps.

6.5.3 Examples of OS-Level I/O

The heart of any operating system is providing services to user programs, mostly I/O services such as reading and writing files. Both UNIX and Windows 7 offer a wide variety of I/O services to user programs. For most UNIX system calls, Windows 7 has an equivalent call, but the reverse is not true, as Windows 7 has far more calls and each is far more complicated than its UNIX counterpart.

UNIX I/O

Much of the popularity of the UNIX system can be traced directly to its simplicity, which, in turn, is a direct result of the organization of the file system. An ordinary file is a linear sequence of 8-bit bytes starting at 0 and going up to a maximum of typically $2^{64} - 1$ bytes. The operating system itself imposes no record structure on files, although many user programs regard ASCII text files as sequences of lines, each line terminated by a line feed.

Associated with every open file is a pointer to the next byte to be read or written. The read and write system calls read and write data starting at the file position indicated by the pointer. Both calls advance the pointer after the operation by an amount equal to the number of bytes transferred. However, random access to files is possible by explicitly setting the file pointer to a specific value.

In addition to ordinary files, the UNIX system also supports special files, which are used to access I/O devices. Each I/O device typically has one or more special files assigned to it. By reading and writing from the associated special file, a program can read or write from the I/O device. Disks, printers, terminals, and many other devices are handled this way.

The major UNIX file system calls are listed in Fig. 6-35. The creat call (without the *e*) can be used to create a new file. It is not strictly necessary any more, because open can also create a new file now. Unlink removes a file, assuming that the file is in only one directory.

Open is used to open existing files (and create new ones). The mode flag tells how to open it (for reading, for writing, etc.). The call returns a small integer called a **file descriptor** that identifies the file in subsequent calls. When the file is no longer needed, close is called to free up the file descriptor.

The actual file I/O is done with read and write, each having a file descriptor indicating which file to use, a buffer for the data to go to or come from, and a byte

System call	Meaning
creat(name, mode)	Create a file; *mode* specifies the protection mode
unlink(name)	Delete a file (assuming that there is only 1 link to it)
open(name, mode)	Open or create a file and return a file descriptor
close(fd)	Close a file
read(fd, buffer, count)	Read *count* bytes into *buffer*
write(fd, buffer, count)	Write *count* bytes from *buffer*
lseek(fd, offset, w)	Move the file pointer as required by *offset* and *w*
stat(name, buffer)	Return information about a file
chmod(name, mode)	Change the protection mode of a file
fcntl(fd, cmd, ...)	Do various control operations such as locking (part of) a file

Figure 6-35. The principal UNIX file system calls.

count telling how much data to transmit. Lseek is used to position the file pointer, making random access to files possible.

Stat returns information about a file, including its size, time of last access, owner, and more. Chmod changes the protection mode of a file, for example, allowing or forbidding users other than the owner from reading it. Finally, fcntl does various miscellaneous operations on a file, such as locking or unlocking it.

Figure 6-36 illustrates how the major file I/O calls work. This code is minimal and does not include the necessary error checking. Before entering the loop, the program opens an existing file, *data*, and creates a new file, *newf*. Each call returns a file descriptor, *infd*, and *outfd*, respectively. The second parameters to the two calls are protection bits specifying that the files are to be read and written, respectively. Both calls return a file descriptor. If either open or creat fails, a negative file descriptor is returned, telling that the call failed.

```
/* Open the file descriptors. */
infd = open("data", 0);
outfd = creat("newf", ProtectionBits);

/* Copy loop. */
do {
        count = read(infd, buffer, bytes);
        if (count > 0) write(outfd, buffer, count);
} while (count > 0);

/* Close the files. */
close(infd);
close(outfd);
```

Figure 6-36. A program fragment for copying a file using the UNIX system calls. This fragment is in C because Java hides the low-level system calls and we are trying to expose them.

The call to read has three parameters: a file descriptor, a buffer, and a byte count. The call tries to read the desired number of bytes from the indicated file into the buffer. The number of bytes actually read is returned in *count*, which will be smaller than *bytes* if the file was too short. The write call deposits the newly read bytes on the output file. The loop continues until the input file has been completely read, at which time the loop terminates and both files are closed.

File descriptors in UNIX are small integers (usually below 20). File descriptors 0, 1, and 2 are special and correspond to **standard input**, **standard output**, and **standard error**, respectively. Normally, these refer to the keyboard, the display, and the display, respectively, but they can be redirected to files by the user. Many UNIX programs get their input from standard input and write the processed output on standard output. Such programs are often called **filters**.

Closely related to the file system is the directory system. Each user may have multiple directories, with each directory containing both files and subdirectories. UNIX systems normally are configured with a main directory, called the **root directory**, containing subdirectories *bin* (for frequently executed programs), *dev* (for the special I/O device files), *lib* (for libraries), and *usr* (for user directories), as shown in Fig. 6-37. In this example, the *usr* directory contains subdirectories for *ast* and *jim*. The *ast* directory contains two files, *data* and *foo.c*, and a subdirectory, *bin*, containing four games.

When multiple disks or disk partitions are present, they can be mounted in the naming tree so that all files on all disks appear in the same directory hierarchy, all reachable from the root directory.

Files can be named by giving their **path** from the root directory. A path contains a list of all the directories traversed from the root to the file, with directory names separated by slashes. For example, the absolute path name of *game2* is */usr/ast/bin/game2*. A path starting at the root is called an **absolute path**.

At every instant, each running program has a **working directory**. Path names may also be relative to the working directory, in which case they do not begin with a slash, to distinguish them from absolute path names. Such paths are called **relative paths**. When */usr/ast* is the working directory, *game3* can be accessed using the path *bin/game3*. A user may create a **link** to someone else's file using the link system call. In the above example, */usr/ast/bin/game3* and */usr/jim/jotto* both access the same file. To prevent cycles in the directory system, links are not permitted to directories. The calls open and creat take either absolute or relative path names as arguments.

The major directory-management system calls in UNIX are listed in Fig. 6-38. Mkdir creates a new directory and rmdir deletes an existing (empty) directory. The next three calls are used to read directory entries. The first one opens the directory, the next one reads entries from it, and the last one closes the directory. Chdir changes the working directory.

Link makes a new directory entry with the new entry pointing to an existing file. For example, the entry */usr/jim/jotto* might have been created by the call

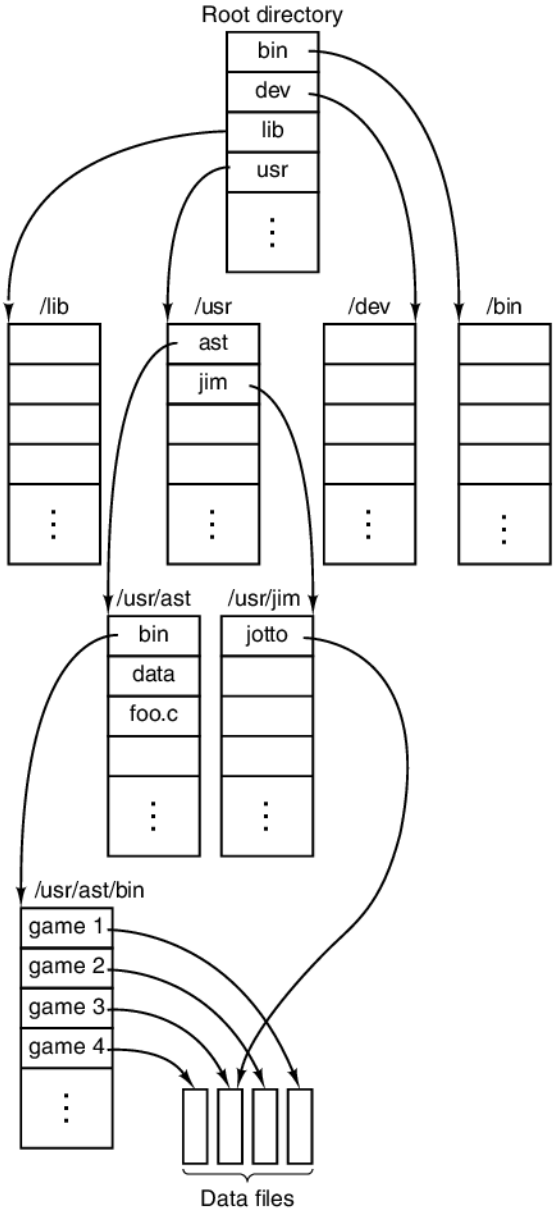

Figure 6-37. Part of a typical UNIX directory system.

```
link("/usr/ast/bin/game3", "/usr/jim/jotto")
```

or an equivalent call using relative path names, depending on the working directory of the program making the call. Unlink removes a directory entry. If the file has

System call	Meaning
mkdir(name, mode)	Create a new directory
rmdir(name)	Delete an empty directory
opendir(name)	Open a directory for reading
readdir(dirpointer)	Read the next entry in a directory
closedir(dirpointer)	Close a directory
chdir(dirname)	Change working directory to *dirname*
link(name1, name2)	Create a directory entry *name2* pointing to *name1*
unlink(name)	Remove *name* from its directory

Figure 6-38. The principal UNIX directory-management calls.

only one link, the file is deleted. If it has two or more links, it is kept. It does not matter whether a removed link is the original or a copy made later. Once a link is made, it is a first-class citizen, indistinguishable from the original. The call

```
unlink("/usr/ast/bin/game3")
```

makes *game3* accessible only via the path */usr/jim/jotto* henceforth. Link and unlink can be used in this way to "move" files from one directory to another.

Associated with every file (including directories, because they are also files) is a bit map telling who may access the file. The map contains three RWX fields, the first controlling the Read, Write, eXecute permissions for the owner, the second for others in the owner's group, and the third for everybody else. Thus RWX R-X --X means that the owner can read the file, write the file, and execute the file (obviously, it is an executable program, or execute would be off), whereas others in his group can read or execute it and strangers can only execute it. With these permissions, strangers can use the program but not steal (copy) it because they do not have read permission. The assignment of users to groups is done by the system administrator, usually called the **superuser**. The superuser also has the power to override the protection mechanism and read, write, or execute any file.

Let us now briefly examine how files and directories are implemented in UNIX. For a more complete treatment, see Vahalia (1996). Associated with each file (and each directory, because a directory is also a file) is a 64-byte block of information called an **i-node**. The i-node tells who owns the file, what the permissions are, where to find the data, and similar things. The i-nodes for the files on each disk are located either in numerical sequence at the beginning of the disk or, if the disk is split up into groups of cylinders, at the start of a cylinder group. Thus, given an i-node number, the UNIX system can locate the i-node by simply calculating its disk address.

A directory entry consists of two parts: a file name and an i-node number. When a program executes

```
open("foo.c", 0)
```

the system searches the working directory for the file name, "foo.c", in order to locate its i-node number. Having found the i-node number, it can then read in the i-node, which tells it all about the file.

When a longer path name is specified, the basic steps outlined above are repeated several times until the full path has been parsed. For example, to locate the i-node number for */usr/ast/data*, the system first searches the root directory for an entry *usr*. Having found the i-node for *usr*, it can read that file (a directory is a file in UNIX). In this file it looks for an entry *ast*, thus locating the i-node number for the file */usr/ast*. By reading */usr/ast*, the system can then find the entry for *data*, and thus the i-node number for */usr/ast/data*. Given the i-node number for the file, it can then find out everything about the file from the i-node.

The format, contents, and layout of an i-node vary somewhat from system to system (especially when networking is in use), but the following items are typically found in each i-node.

1. The file type, the 9 RWX protection bits, and a few other bits.
2. The number of links to the file (number of directory entries for it).
3. The owner's identity.
4. The owner's group.
5. The file length in bytes.
6. Thirteen disk addresses.
7. The time the file was last read.
8. The time the file was last written.
9. The time the i-node was last changed.

The file type distinguishes ordinary files, directories, and two kinds of special files, for block-structured and unstructured I/O devices, respectively. The number of links and the owner identification have already been discussed. The file length is an integer giving the highest byte that has a value. It is perfectly legal to create a file, do an lseek to position 1,000,000, and write 1 byte, which yields a file of length 1,000,001. The file would *not*, however, require storage for all the "missing" bytes.

The first 10 disk addresses point to data blocks. With a block size of 1024 bytes, files up to 10,240 bytes can be handled this way. Address 11 points to a disk block, called an **indirect block**, which contains more disk addresses. With a 1024-byte block and 32-bit disk addresses, the indirect block would contain 256 disk addresses. Files up to $10{,}240 + 256 \times 1024 = 272{,}384$ bytes are handled this way. For still larger files, address 12 points to a block containing the addresses of 256 indirect blocks, which takes care of files up to $272{,}384 + 256 \times 256 \times 1024 = 67{,}381{,}248$ bytes. If this **double indirect block** scheme is still too small, disk ad-

dress 13 is used to point to a **triple indirect block** containing the addresses of 256 double indirect blocks. Using the direct, single, double, and triple indirect addresses, up to 16,843,018 blocks can be addressed, giving a theoretical maximum file size of 17,247,250,432 bytes. If disk addresses are 64 bits instead of 32, and disk blocks are 4 KB, then files can be really, really, really big. Free disk blocks are kept on a linked list. When a new block is needed, the next block is plucked from the list. As a result, the blocks of each file are scattered around the disk.

In order to make disk I/O more efficient, when a file is opened, its i-node is copied to a table in main memory and is kept there for handy reference as long as the file remains open. In addition, a pool of recently referenced disk blocks is maintained in memory. Because most files are read sequentially, it often happens that a file reference requires the same disk block as the previous reference. To strengthen this effect, the system also tries to read the *next* block in a file, before it is referenced, in order to speed up processing. All this optimization is hidden from the user; when a user issues a read call, the program is suspended until the requested data are available in the buffer.

With this background information, we can now take a look to see how file I/O works. Open causes the system to search the directories for the specified path. If the search is successful, the i-node is read into an internal table. Reads and writes require the system to compute the block number from the current file position. The disk addresses of the first 10 blocks are always in main memory (in the i-node); higher-numbered blocks require one or more indirect blocks to be read first. Lseek just changes the current position pointer without doing any I/O.

Link and unlink are also simple to understand now. Link looks up its first argument to find the i-node number. Then it creates a directory entry for the second argument, putting the i-node number of the first file in that entry. Finally, it increases the link count in the i-node by one. Unlink removes a directory entry and decrements the link count in the i-node. If it is zero, the file is removed and all the blocks are put back on the free list.

Windows 7 I/O

Windows 7 supports several file systems, the most important of which are **NTFS** (**NT File System**) and the **FAT** (**File Allocation Table**) file system. The former is a new file system developed specifically for NT; the latter is the old MS-DOS file system, which was also used on Windows 95/98 (albeit with support for longer file names). Since the FAT file system is basically obsolete except for USB sticks and memory cards for cameras, we will study NTFS below.

File names in NTFS can be up to 255 characters long. File names are in Unicode, allowing people in countries not using the Latin alphabet (e.g., Japan, India, and Israel) to write file names in their native language. (In fact, Windows 7 uses Unicode throughout internally; versions starting with Windows 2000 have a single binary that can be used in any country and still use the local language because all

the menus, error messages, etc., are kept in country-dependent configuration files.) NTFS fully supports case-sensitive names (so *foo* is different from *FOO*). The Win32 API does not fully support case sensitivity for file names and not at all for directory names, so this advantage is lost to programs using Win32.

As with UNIX, a file is just a linear sequence of bytes, although up to a maximum of $2^{64} - 1$ bytes. File pointers also exist, as in UNIX, but are 64 rather than 32 bits wide, to handle files of the maximum length. The Win32 API function calls for file and directory manipulation are roughly similar to their UNIX counterparts, except that most have more parameters and the security model is different. Opening a file returns a handle, which is then used for reading and writing the file. However, unlike in UNIX, handles are not small integers because they are used to identify all kernel objects, of which there are potentially millions. The principal Win32 API functions for file management are listed in Fig. 6-39.

API function	UNIX	Meaning
CreateFile	open	Create a file or open an existing file; return a handle
DeleteFile	unlink	Delete an existing file entry from a directory
CloseHandle	close	Close a file
ReadFile	read	Read data from a file
WriteFile	write	Write data to a file
SetFilePointer	lseek	Set the file pointer to a specific place in the file
GetFileAttributes	stat	Return the file properties
LockFile	fcntl	Lock a region of the file to provide mutual exclusion
UnlockFile	fcntl	Unlock a previously locked region of the file

Figure 6-39. The principal Win32 API functions for file I/O. The second column gives the nearest UNIX equivalent.

Let us now examine these calls briefly. CreateFile can be used to create a new file and return a handle to it. This API function is also used to open existing files as there is no open API function. We have not listed the parameters for the Windows 7 API functions because they are so voluminous. As an example, CreateFile has seven parameters, as follows:

1. A pointer to the name of the file to create or open.
2. Flags telling whether the file can be read, written, or both.
3. Flags telling whether multiple processes can open the file at once.
4. A pointer to the security descriptor, telling who can access the file.
5. Flags telling what to do if the file exists/does not exist.
6. Flags dealing with attributes such as archiving, compression, etc.
7. The handle of a file whose attributes are to be cloned for the new file.

The next six API functions in Fig. 6-39 are fairly similar to the corresponding UNIX system calls. Note, however, that Windows 7 I/O is, in principle, asynchronous, although it is possible for a process to wait for completion. The last two functions allow a region of a file to be locked and unlocked to permit a process to get guaranteed mutual exclusion to it.

Using these API functions, it is possible to write a procedure to copy a file, analogous to the UNIX version of Figure 6-36. Such a procedure (without any error checking) is shown in Fig. 6-40. It has been designed to mimic the structure of Fig. 6-36. In practice, one would not have to program a copy file function since CopyFile is an API function (which executes something close to this program as a library procedure).

```
/* Open files for input and output. */
inhandle = CreateFile("data", GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
outhandle = CreateFile("newf", GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
        FILE_ATTRIBUTE_NORMAL, NULL);

/* Copy the file. */
do {
        s = ReadFile(inhandle, buffer, BUF_SIZE, &count, NULL);
        if (s > 0 && count > 0) WriteFile(outhandle, buffer, count, &ocnt, NULL);
} while (s > 0 && count > 0);

/* Close the files. */
CloseHandle(inhandle);
CloseHandle(outhandle);
```

Figure 6-40. A program fragment for copying a file using the Windows 7 API functions. This fragment is in C because Java hides the low-level system calls and we are trying to expose them.

Windows 7 supports a hierarchical file system, similar to the UNIX file system. The separator between component names is \ however, instead of /, a fossil inherited from MS-DOS. There is a concept of a current working directory and path names can be relative or absolute. One significant difference, however, is that UNIX allows the file systems on different disks and machines to be mounted together in a single naming tree starting at a unique root, thus hiding the disk structure from all software. Early versions of Windows (pre Windows 2000) did not have this property, so absolute file names had to begin with a drive letter indicating which logical disk was meant, as in *C:\windows\system\foo.dll*. Starting with Windows 2000 UNIX-style mounting of file systems was added.

The major directory management API functions are given in Fig. 6-41, again along with their nearest UNIX equivalents. The functions should be self-explanatory.

Windows 7 has a much more elaborate security mechanism than most UNIX systems. Although there are hundreds of API functions relating to security, the

API function	UNIX	Meaning
CreateDirectory	mkdir	Create a new directory
RemoveDirectory	rmdir	Remove an empty directory
FindFirstFile	opendir	Initialize to start reading the entries in a directory
FindNextFile	readdir	Read the next directory entry
MoveFile		Move a file from one directory to another
SetCurrentDirectory	chdir	Change the current working directory

Figure 6-41. The principal Win32 API functions for directory management. The second column gives the nearest UNIX equivalent, when one exists.

following brief description gives the general idea. When a user logs in, his or her initial process is given an **access token** by the operating system. The access token contains the user's **SID** (**Security ID**), a list of the security groups to which the user belongs, any special privileges available, the integrity level of the process, and a few other items. The point of the access token is to concentrate all the security information in one easy-to-find place. All processes created by this process inherit the same access token.

One of the parameters that can be supplied when any object is created is its **security descriptor**. The security descriptor contains a list of entries called an **ACL** (**Access Control List**). Each entry permits or prohibits some set of the operations on the object by some SID or group. For example, a file could have a security descriptor specifying that Elinor has no access to the file at all, Ken can read the file, Linda can read or write the file, and all members of the XYZ group can read the file's length but nothing else. Defaults can also be set up to deny access to anyone not explicitly listed.

When a process tries to perform some operation on an object using a handle, the security manager gets the process' access token and first checks the integrity level in the object's security descriptor against the integrity level in the token. A process cannot obtain a handle with write permission for any object with a higher integrity level. Integrity levels are primarily used to restrict what code loaded by Web browsers can do to modify the system. After the integrity-level check, the security manager goes down the list of entries in the ACL in order. As soon as it finds an entry that matches the called's SID or one of the called's groups, the access found there is taken as definitive. For this reason, it is usual to put entries denying access ahead of entries granting access in the ACL, so that a user who is specifically denied access cannot get in via a back door by being a member of a group that has legitimate access. The security descriptor also contains information used for auditing accesses to the object.

Let us now take a quick look at how files and directories are implemented in Windows 7. Each disk is statically divided up into self-contained volumes, which are the same as disk partitions in UNIX. Each volume contains bit maps, files,

directories, and other data structures for managing its information. Each volume is organized as a linear sequence of **clusters**, with the cluster size being fixed for each volume and ranging from 512 bytes to 64 KB, depending on the volume size. Clusters are referred to by their offset from the start of the volume using 64-bit numbers.

The main data structure in each volume is the **MFT** (**Master File Table**), which has an entry for each file and directory in the volume. These entries are analogous to the i-nodes in UNIX. The MFT is itself a file, and as such can be placed anywhere within the volume. This property is useful in case there are defective disk blocks at the beginning of the volume where the MFT would normally be stored. UNIX systems normally store certain key information at the start of each volume and in the (extremely unlikely) case that one of these blocks is irreparably damaged, the entire volume has to be repositioned.

The MFT is shown in Fig. 6-42. It begins with a header containing information about the volume, such as (pointers to) the root directory, the boot file, the bad-block file, the free-list administration, etc. After that comes an entry per file or directory, 1 KB except when the cluster size is 2 KB or more. Each entry contains all the metadata (administrative information) about the file or directory. Several formats are allowed, one of which is shown in Fig. 6-42.

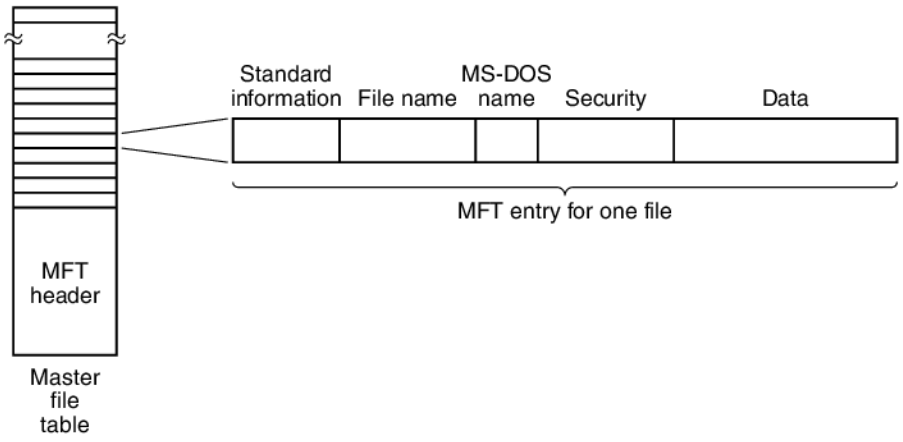

Figure 6-42. The Windows 7 master file table.

The standard information field contains information such as the time stamps needed by POSIX, the hard link count, the read-only and archive bits, etc. It is a fixed-length field and always present. The file name is of variable length, up to 255 Unicode characters. In order to make such files accessible to old 16-bit programs, files can also have a MS-DOS name, which consists of eight alphanumeric characters optionally followed by a dot and an extension of no more than three

alphanumeric characters. If the actual file name conforms to the MS-DOS 8+3 naming rule, a secondary MS-DOS name is not used.

Next comes the security information. In versions up to and including Windows NT 4.0, the security field contained the actual security descriptor. Starting with Windows 2000, all the security information was centralized in a single file, with the security field simply pointing to the relevant part of this file.

For small files, the file data itself is actually contained in the MFT entry, saving a disk access to fetch it. This idea is called an **immediate file** (Mullender and Tanenbaum, 1984). For somewhat larger files, this field contains pointers to the clusters containing the data, or more commonly, runs of consecutive clusters so a single cluster number and a length can represent an arbitrary amount of file data. If a single MFT entry is insufficiently large to hold whatever information it is supposed to hold, one or more additional entries can be chained to it.

The maximum file size is 2^{64} bytes. To get an idea of how big a 2^{64}-byte (i.e., 2^{67}-bit) file is, imagine that it were written out in binary, with each 0 or 1 occupying 1 mm of space. The 2^{67}-mm listing would be 15 light-years long, reaching far beyond the solar system, to Alpha Centauri and back.

The NTFS file system has many other interesting properties including support for multiple data streams for each file, encryption, data compression, and fault tolerance using atomic transactions. Additional information about it can be found in Russinovich and Solomon (2005).

6.5.4 Examples of Process Management

Both UNIX and Windows 7 allow a job to be split up into multiple processes that can run in (pseudo)parallel and communicate with each other, in the style of the producer-consumer example discussed earlier. In this section we will discuss how processes are managed in both systems. Both systems also support parallelism within a single process using threads, so that will also be discussed.

UNIX Process Management

At any time, a UNIX process can create a subprocess that is an exact replica of itself by executing the fork system call. The original process is called the **parent** and the new one the **child**. Right after the fork, the two processes are identical and even share the same file descriptors. Thereafter, each one goes its own way and does whatever it wants to, independent of the other one.

In many cases, the child process juggles the file descriptors in certain ways and then executes the exec system call, which replaces its program and data with the program and data found in an executable file specified as parameter to the exec call. For example, when a user types a command *xyz* at a terminal, the command interpreter (shell) executes fork to create a child process. This child process then executes exec to run the *xyz* program.

The two processes run in parallel (with or without exec), unless the parent wishes to wait for the child to terminate before continuing. If the parent wishes to wait, it executes either the wait or waitpid system call, which causes it to be suspended until the child finishes by executing exit. After the child finishes, the parent continues.

Processes can execute fork as often as they want, giving rise to a tree of processes. In Fig. 6-43, for example, process *A* has executed fork twice, creating two children, *B* and *C*. Then *B* also executed fork twice, and *C* executed it once, giving the final tree of six processes.

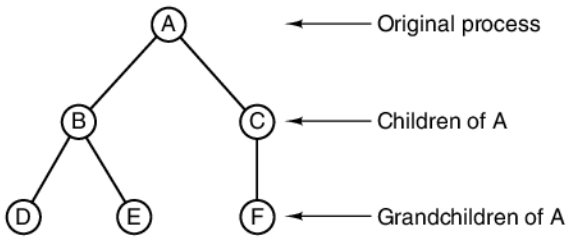

Figure 6-43. A process tree in UNIX.

Processes in UNIX can communicate with each other via a structure called a **pipe**. A pipe is a kind of buffer into which one process can write a stream of data and another can take it out. Bytes are always retrieved from a pipe in the order they were written. Random access is not possible. Pipes do not preserve message boundaries, so if one process does four 128-byte writes and the other does a 512-byte read, the reader will get all the data at once, with no indication that they were written in multiple operations.

In System V and Linux, another way for processes to communicate is by using **message queues**. A process can create a new message queue or open an existing one using msgget. Using a message queue, a process can send messages using msgsnd and receive them using msgrecv. Messages sent this way differ in several ways from data stuffed into a pipe. First, message boundaries are preserved, whereas a pipe is just a byte stream. Second, messages have priorities, so urgent ones can skip ahead of less important ones. Third, messages are typed, and a msgrecv can specify a particular type, if desired.

Another communication mechanism is the ability of two or more processes to share a region of their respective address spaces. UNIX handles this shared memory by mapping the same pages into the virtual address space of all the sharing processes. As a result, a write by one process into the shared region is immediately visible to the other processes. This mechanism provides a very high bandwidth communication path between processes. The system calls involved in shared memory go by names like shmat and shmop.

System V and Linux also provide semaphores. These work essentially as described in the producer-consumer example given in the text.

Yet another facility provided by all POSIX-conformant UNIX systems is the ability to have multiple threads of control within a single process. These threads of control, usually just called **threads**, are like lightweight processes that share a common address space and everything associated with that address space, such as file descriptors, environment variables, and outstanding timers. However, each thread has its own program counter, own registers, and own stack. When a thread blocks (i.e., has to stop temporarily until I/O completes or some other event happens), other threads in the same process are still able to run. Two threads in the same process operating as a producer and consumer are similar, but not identical, to two single-thread processes that are sharing a memory segment containing a buffer. The differences have to do with the fact that in the latter case, each process has its own file descriptors, etc., whereas in the former case all of these items are shared. We saw the use of Java threads in our producer-consumer example earlier. Often the Java runtime system uses an operating system thread for each of its threads, but it does not have to do this.

As an example of where threads might be useful, consider a World Wide Web server. Such a server might keep a cache of commonly used Web pages in main memory. If a request is for a page in the cache, the Web page is returned immediately. Otherwise, it is fetched from disk. Unfortunately, waiting for the disk takes a long time (typically 20 msec), during which the process is blocked and cannot serve new incoming requests, even those for Web pages in the cache.

The solution is to have multiple threads within the server process, all of which share the common Web page cache. When one thread blocks, other threads can handle new requests. To prevent blocking without threads, one could have multiple server processes, but this would probably entail replicating the cache, thus wasting valuable memory.

The UNIX standard for threads is called **pthreads**, and is defined by POSIX (P1003.1C). It contains calls for managing and synchronizing threads. It is not defined whether threads are managed by the kernel or entirely in user space. The most commonly used thread calls are listed in Fig. 6-44.

Let us briefly examine the thread calls shown in Fig. 6-44. The first call, pthread_create, creates a new thread. After successful completion, one more thread is running in the called's address space than before the call. A thread that has done its job and wants to terminate calls pthread_exit. A thread can wait for another thread to exit by calling pthread_join. If the thread waited for has already exited, the pthread_join finishes immediately. Otherwise it blocks.

Threads can synchronize using **mutexes**. A mutex guards some resource, such as a buffer shared by two threads. To make sure that only one thread at a time accesses the shared resource, threads are expected to lock the mutex before touching the resource and unlock it when they are done. As long as all threads obey this protocol, race conditions can be avoided. Mutexes are like binary semaphores (semaphores that can take on only the values of 0 and 1). The name "mutex" comes from the fact that mutexes are used to ensure mutual exclusion.

Thread call	Meaning
pthread_create	Create a new thread in the called's address space
pthread_exit	Terminate the calling thread
pthread_join	Wait for a thread to terminate
pthread_mutex_init	Create a new mutex
pthread_mutex_destroy	Destroy a mutex
pthread_mutex_lock	Lock a mutex
pthread_mutex_unlock	Unlock a mutex
pthread_cond_init	Create a condition variable
pthread_cond_destroy	Destroy a condition variable
pthread_cond_wait	Wait on a condition variable
pthread_cond_signal	Release one thread waiting on a condition variable

Figure 6-44. The principal POSIX thread calls.

Mutexes can be created and destroyed by the calls pthread_mutex_init and pthread_mutex_destroy, respectively. A mutex can be in one of two states: locked or unlocked. When a thread needs to set a lock on an unlocked mutex (using pthread_mutex_lock), the lock is set and the thread continues. However, when a thread tries to lock a mutex that is already locked, it blocks. When the locking thread is finished with the shared resource, it is expected to unlock the corresponding mutex by calling pthread_mutex_unlock.

Mutexes are intended for short-term locking, such as protecting a shared variable. They are not intended for long-term synchronization, such as waiting for a tape drive to become free. For long-term synchronization, **condition variables** are provided. These are created and destroyed by calls to pthread_cond_init and pthread_cond_destroy, respectively.

A condition variable is used by having one thread wait on it, and another signal it. For example, having discovered that the tape drive it needs is busy, a thread would do pthread_cond_wait on a condition variable that all the threads have agreed to associate with the tape drive. When the thread using the tape drive is finally done with it (possibly hours later), it uses pthread_cond_signal to release exactly one thread waiting on that condition variable (if any). If no thread is waiting, the signal is lost. Condition variables do not count like semaphores. A few other operations are also defined on threads, mutexes, and condition variables.

Windows 7 Process Management

Windows 7 supports multiple processes, which can communicate and synchronize. Each process contains at least one thread. Together, processes and threads (which can be scheduled by the process itself) provide a general set of tools for

managing parallelism, both on uniprocessors (single-CPU machines) and on multiprocessors (multi-CPU machines).

New processes are created using the API function CreateProcess. This function has 10 parameters, each of which has many options. This design is clearly a lot more complicated than the UNIX scheme, in which fork has no parameters, and exec has just three: pointers to the name of the file to execute, the (parsed) command-line parameter array, and the environment strings. Roughly speaking, the 10 parameters to CreateProcess are as follows:

1. A pointer to the name of the executable file.
2. The command line itself (unparsed).
3. A pointer to a security descriptor for the process.
4. A pointer to a security descriptor for the initial thread.
5. A bit telling whether the new process inherits the creator's handles.
6. Miscellaneous flags (e.g., error mode, priority, debugging, consoles).
7. A pointer to the environment strings.
8. A pointer to the name of the new process' current working directory.
9. A pointer to a structure describing the initial window on the screen.
10. A pointer to a structure that returns 18 values to the called.

Windows 7 does not enforce any kind of parent-child or other hierarchy. All processes are created equal. However, since 1 of the 18 parameters returned to the creating process is a handle to the new process (allowing considerable control over the new process), there is an implicit hierarchy in terms of who has a handle to whom. Although these handles cannot just be passed directly to other processes, there is a way for a process to make a handle suitable for another process and then give it the handle, so the implicit process hierarchy may not last long.

Each process in Windows 7 is created with a single thread, but a process can create more threads later on. Thread creation is simpler than process creation: CreateThread has only six parameters instead of 10: the security descriptor, the stack size, the starting address, a user-defined parameter, the initial state of the thread (ready or blocked), and the thread's ID. The kernel does the thread creation, so it is clearly aware of threads (i.e., they are not implemented purely in user space as is the case in some other systems).

When the kernel does scheduling, it looks only at the runnable threads and pays no any attention at all to which process each one is in. This means that the kernel is always aware of which threads are ready and which ones are blocked. Because threads are kernel objects, they have security descriptors and handles. Since a handle for a thread can be passed to another process, it is possible to have

one process control (or even create) threads in a different process. This feature is useful for debuggers, for example.

Processes can communicate in a wide variety of ways, including pipes, named pipes, sockets, remote procedure calls, and shared files. Pipes have two modes: byte and message, selected at creation time. Byte-mode pipes work the same way as in UNIX. Message-mode pipes are somewhat similar but preserve message boundaries, so that four writes of 128 bytes will be read as four 128-byte messages, and not as one 512-byte message, as would happen with byte-mode pipes. Named pipes also exist and have the same two modes as regular pipes. Named pipes can also be used over a network; regular pipes cannot.

Sockets are like pipes, except that they normally connect processes on different machines. However, they can also be used to connect processes on the same machine. In general, there is usually little advantage to using a socket connection over a pipe or named pipe for intramachine communication.

Remote procedure calls are a way for process *A* to have process *B* call a procedure in *B*'s address space on *A*'s behalf and return the result to *A*. Various restrictions on the parameters exist. For example, it makes no sense to pass a pointer to a different process. Instead, the object(s) pointed to have to be bundled up and sent to the destination process.

Finally, processes can share memory by mapping onto the same file at the same time. All writes done by one process then appear in the address spaces of the other processes. Using this mechanism, the shared buffer used in our producer-consumer example can be easily implemented.

Just as Windows 7 provides numerous interprocess communication mechanisms, it also provides numerous synchronization mechanisms, including semaphores, mutexes, critical sections, and events. All of these mechanisms work on threads, not processes, so that when a thread blocks on a semaphore, other threads in that process (if any) are not affected and can continue to run.

A semaphore is created using the CreateSemaphore API function, which can initialize it to a given value and define a maximum value as well. Semaphores are kernel objects and thus have security descriptors and handles. The handle for a semaphore can be duplicated using DuplicateHandle and passed to another process so that multiple processes can synchronize on the same semaphore. Semaphores can also be given names when they are created so that other processes can open them by name. Calls for up and down are present, although they have the speculiar names of ReleaseSemaphore (up) and WaitForSingleObject (down). It is also possible to give WaitForSingleObject a timeout, so the calling thread can be released eventually, even if the semaphore remains at 0 (although timers reintroduce races).

Mutexes are also kernel objects used for synchronization, but simpler than semaphores because they do not have counters. They are essentially locks, with API functions for locking (WaitForSingleObject) and unlocking (ReleaseMutex). Like semaphore handles, mutex handles can be duplicated and passed between processes so that threads in different processes can access the same mutex.

The third synchronization mechanism is based on **critical sections**, which are similar to mutexes, except local to the address space of the creating thread. Because critical sections are not kernel objects, they do not have handles or security descriptors and cannot be passed between processes. Locking and unlocking is done with EnterCriticalSection and LeaveCriticalSection, respectively. Because these API functions are performed entirely in user space, they are much faster than mutexes. Windows 7 also provides condition variables, lightweight reader/writer locks, lock-free operations, and other synchronization mechanisms that work only between the threads of a single process.

The last synchronization mechanism uses kernel objects called **events**. A thread can wait for an event to occur with WaitForSingleObject. A thread can release a single thread waiting on an event with SetEvent or it can release all threads waiting on an event with PulseEvent. Events come in several flavors and have a variety of options, too. Windows uses events to synchronize on the completion of asynchronous I/O and for other purposes.

Events, mutexes, and semaphores can all be named and stored in the file system, like named pipes. Two or more processes can synchronize by opening the same event, mutex, or semaphore, rather than having one of them create the object and then make duplicate handles for the others, although the latter approach is certainly an option as well.

6.6 SUMMARY

The operating system can be regarded as an interpreter for certain architectural features not found at the ISA level. Chief among these are virtual memory, virtual I/O instructions, and facilities for parallel processing.

Virtual memory is an architectural feature whose purpose is to allow programs to use more address space than the machine has physical memory, or to provide a consistent and flexible mechanism for memory protection and sharing. It can be implemented as pure paging, pure segmentation, or a combination of the two. In pure paging, the address space is broken up into equal-sized virtual pages. Some of these are mapped onto physical page frames. Others are not mapped. A reference to a mapped page is translated by the MMU into the correct physical address. A reference to an unmapped page causes a page fault. Both the Core i7 and the OMAP4430 ARM CPU have MMUs that support virtual memory and paging.

The most important I/O abstraction present at this level is the file. A file consists of a sequence of bytes or logical records that can be read and written without knowledge of how disks, tapes, and other I/O devices work. Files can be accessed sequentially, randomly by record number, or randomly by key. Directories can be used to group files together. Files can be stored in consecutive sectors or scattered around the disk. In the latter case, normal on hard disks, data structures are needed

to locate all the blocks of a file. Free disk storage can be kept track of using a list or a bit map.

Parallel processing is often supported and is implemented by simulating multiple processors by timesharing a single CPU. Uncontrolled interaction between processes can lead to race conditions. To solve this problem, synchronization primitives are introduced, of which semaphores are a simple example. Using semaphores, producer-consumer problems can be solved simply and elegantly.

Two examples of sophisticated operating systems are UNIX and Windows 7. Both support paging and memory-mapped files. They also both support hierarchical file systems, with files consisting of byte sequences. Finally, both support processes and threads and provide ways to synchronize them.

PROBLEMS

1. Why does an operating system interpret only some of the level 3 instructions, whereas a microprogram interprets all the ISA-level instructions?
2. A machine has a 32-bit byte-addressable virtual address space. The page size is 4 KB. How many pages of virtual address space exist?
3. Is it necessary to have the page size be a power of 2? Could a page of size, say, 4000 bytes be implemented in theory? If so, would it be practical?
4. A virtual memory has a page size of 1024 words, eight virtual pages, and four physical page frames. The page table is as follows:

Virtual page	Page frame
0	3
1	1
2	not in main memory
3	not in main memory
4	2
5	not in main memory
6	0
7	not in main memory

   a. Make a list of all virtual addresses that will cause page faults.
   b. What are the physical addresses for 0, 3728, 1023, 1024, 1025, 7800, and 4096?

5. A computer has 16 pages of virtual address space but only four page frames. Initially, the memory is empty. A program references the virtual pages in the order

   0, 7, 2, 7, 5, 8, 9, 2, 4

a. Which references cause a page fault with LRU?
b. Which references cause a page fault with FIFO?

6. In Sec. 6.1.4 an algorithm was presented for implementing a FIFO page replacement strategy. Devise a more efficient one. *Hint:* It is possible to update the counter in the newly loaded page, leaving all the others alone.

7. In the paged systems discussed in the text, the page fault handler was part of the ISA level and thus was not present in any OSM-level program's address space. In reality, the page fault handler also occupies pages, and might, under some circumstances (e.g., FIFO page replacement policy), itself be removed. What would happen if the page fault handler were not present when a page fault occurred? How could this be fixed?

8. Not all computers have a hardware bit that is automatically set when a page is written to. Nevertheless, it is useful to keep track of which pages have been modified, to avoid having to assume worst case and write all pages back to the disk after use. Assuming that each page has hardware bits to separately enable access for reading, writing, and execution, how can the operating system keep track of which pages are clean and which are dirty?

9. A segmented memory has paged segments. Each virtual address has a 2-bit segment number, a 2-bit page number, and an 11-bit offset within the page. The main memory contains 32 KB, divided into 2-KB pages. Each segment is either read-only, read/execute, read/write, or read/write/execute. The page tables and protection are as follows:

Segment 0		Segment 1		Segment 2	Segment 3	
Read only		Read/execute		Read/write/execute	Read/write	
Virtual page	Page frame	Virtual page	Page frame		Virtual page	Page frame
0	9	0	On disk	Page table	0	14
1	3	1	0	not in	1	1
2	On disk	2	15	main	2	6
3	12	3	8	memory	3	On disk

For each of the following accesses to virtual memory, tell what physical address is computed. If a fault occurs, tell which kind.

Access	*Segment*	*Page*	*Offset within page*
1. fetch data	0	1	1
2. fetch data	1	1	10
3. fetch data	3	3	2047
4. store data	0	1	4
5. store data	3	1	2
6. store data	3	0	14
7. branch to it	1	3	100
8. fetch data	0	2	50
9. fetch data	2	0	5
10. branch to it	3	0	60

10. Some computers allow I/O directly to user space. For example, a program could start up a disk transfer to a buffer inside a user process. Does this cause any problems if compaction is used to implement the virtual memory? Discuss.

11. Operating systems that allow memory-mapped files always require files to be mapped at page boundaries. For example, with 4-KB pages, a file can be mapped in starting at virtual address 4096, but not starting at virtual address 5000. Why?

12. When a segment register is loaded on the Core i7, the corresponding descriptor is fetched and loaded into an invisible part of the segment register. Why do you think the Intel designers decided to do this?

13. A program on the Core i7 references local segment 10 with offset 8000. The BASE field of LDT segment 10 contains 10000. Which page directory entry does the Core i7 use? What is the page number? What is the offset?

14. Discuss some possible algorithms for removing segments in an unpaged, segmented memory.

15. Compare internal fragmentation to external fragmentation. What can be done to alleviate each?

16. Supermarkets are constantly faced with a problem similar to page replacement in virtual memory systems. They have a fixed amount of shelf space to display an ever-increasing number of products. If an important new product comes along, say, 100% efficient dog food, some existing product must be dropped from the inventory to make room for it. The obvious replacement algorithms are LRU and FIFO. Which of these would you prefer?

17. In some ways, caching and paging are very similar. In both cases there are two levels of memory (the cache and main memory in the former and main memory and disk in the latter). In this chapter we looked at some of the arguments in favor of large disk pages and small disk pages. Do the same arguments hold for cache line sizes?

18. Why do many file systems require that a file be explicitly opened with an open system call before being read?

19. Compare the bit-map and hole-list methods for keeping track of free space on a disk with 800 cylinders, each one having 5 tracks of 32 sectors. How many holes would it take before the hole list would be larger than the bit map? Assume that the allocation unit is the sector and that a hole requires a 32-bit table entry.

20. A third hole allocation scheme, in addition to best fit and first fit, is worst fit, where a process is allocated space from the largest remaning hole. What advantage can be gained by using the worst fit algorithm?

21. Describe a purpose for the file open system call that was not mentioned in the text.

22. To be able to make some predictions of disk performance, it is useful to have a model of storage allocation. Suppose that the disk is viewed as a linear address space of $N \gg 1$ sectors, consisting of a run of data blocks, then a hole, then another run of data blocks, and so on. If empirical measurements show that the probability distributions for data and hole lengths are the same, with the chance of either being i sectors as 2^{-i}, what is the expected number of holes on the disk?

23. On a certain computer, a program can create as many files as it needs, and all files may grow dynamically during execution without giving the operating system any advance information about their ultimate size. Do you think that files are stored in consecutive sectors? Explain.

24. Studies of different file systems have shown that more than half the files are a few KB or smaller, with the vast majority of files less than something like 8 KB. On the other hand, the largest 10 percent of all files usually occupies about 95 percent of the entire disk space in use. From this data, what conclusion can you draw about disk block size?

25. Consider the following method by which an operating system might implement semaphore instructions. Whenever the CPU is about to do an up or down on a semaphore (an integer variable in memory), it first sets the CPU priority or mask bits in such a way as to disable all interrupts. Then it fetches the semaphore, modifies it, and branches accordingly. Finally, it enables interrupts again. Does this method work if

a. There is a single CPU that switches between processes every 100 msec?
b. Two CPUs share a common memory in which the semaphore is located?

26. The description of semaphores in Sec. 6.3.3 states: "Usually sleeping processes are strung together in a queue to keep track of them." What advantage is gained by using a queue for waiting processes as opposed to waking a random sleeping processes when an up is performed?

27. The Nevercrash Operating System Company has been receiving complaints from some of its customers about its latest release, which includes semaphore operations. They feel it is immoral for processes to block (they call it "sleeping on the job"). Since it is company policy to give the customers what they want, it has been proposed to add a third operation, peek, to supplement up and down. peek simply examines the semaphore without changing it or blocking the process. In this way, programs that feel it is immoral to block can first inspect the semaphore to see if it is safe to do a down. Will this idea work if three or more processes use the semaphore? If two processes use the semaphore?

28. Make a table showing which of the processes P1, P2, and P3 are running and which are blocked as a function of time from 0 to 1000 msec. All three processes perform up and down instructions on the same semaphore. When two processes are blocked and an up is done, the process with the lower number is restarted, that is, P1 gets preference over P2 and P3, and so on. Initially, all three are running and the semaphore is 1.

At $t = 100$ P1 does a down
At $t = 200$ P1 does a down
At $t = 300$ P2 does an up
At $t = 400$ P3 does a down
At $t = 500$ P1 does a down
At $t = 600$ P2 does an up
At $t = 700$ P2 does a down
At $t = 800$ P1 does an up
At $t = 900$ P1 does an up

29. In an airline reservation system, it is necessary to ensure that while one process is busy using a file, no other process can also use it. Otherwise, two different processes,

working for two different ticket agents, might each inadvertently sell the last seat on some flight. Devise a synchronization method using semaphores that makes sure that only one process at a time accesses each file (assuming the processes obey the rules).

30. To make it possible to implement semaphores on a computer with multiple CPUs that share a common memory, computer architects often provide a Test and Set Lock instruction. `TSL X` tests the location X. If the contents are zero, they are set to 1 in a single, indivisible memory cycle, and the next instruction is skipped. If it is nonzero, the `TSL` acts like a no-op. Using `TSL` it is possible to write procedures *lock* and *unlock* with the following properties. *lock*(x) checks to see if x is locked. If not, it locks x and returns control. If x is already locked, it just waits until it becomes unlocked, then it locks x and returns control. *unlock* releases an existing lock. If all processes lock the semaphore table before using it, only one process at a time can fiddle with the variables and pointers, thus preventing races. Write *lock* and *unlock* in assembly language. (Make any reasonable assumptions you need.)

31. Show the values of *in* and *out* for a circular buffer of length 65 words after each of the following operations. Both start at 0.

a. 22 words are put in
b. 9 words are removed
c. 40 words are put in
d. 17 words are removed
e. 12 words are put in
f. 45 words are removed
g. 8 words are put in
h. 11 words are removed

32. Suppose that a version of UNIX uses 2-KB disk blocks and stores 512 disk addresses per indirect block (single, double, and triple). What would the maximum file size be? (Assume that file pointers are 64 bits wide).

33. Suppose that the UNIX system call

```
unlink("/usr/ast/bin/game3")
```

were executed in the context of Figure 6-37. Describe carefully what changes are made in the directory system.

34. Imagine that you had to implement the UNIX system on an embedded system where main memory was in short supply. After a considerable amount of shoehorning, it still did not quite fit, so you picked a system call at random to sacrifice for the general good. You picked `pipe`, which creates the pipes used to send byte streams from one process to another. Is it still possible to implement I/O redirection somehow? What about pipelines? Discuss the problems and possible solutions.

35. The Committee for Fairness to File Descriptors is organizing a protest against the UNIX system because whenever the latter returns a file descriptor, it always returns the lowest number not currently in use. Consequently, higher-numbered file descriptors are hardly ever used. Their plan is to return the lowest number not yet used by the program rather than the lowest number currently not in use. They claim that it is trivial to implement, will not affect existing programs, and is fairer. What do you think?

36. In Windows 7 it is possible to set up an access control list in such a way that Roberta has no access at all to a file, but everyone else has full access to it. How do you think this is implemented?

37. Describe two different ways to program producer-consumer problems using shared buffers and semaphores in Windows 7. Think about how to implement the shared buffer in each case.

38. It is common to test out page replacement algorithms by simulation. For this exercise, you are to write a simulator for a page-based virtual memory for a machine with 64 1-KB pages. The simulator should maintain a single table of 64 entries, one per page, containing the physical page number corresponding to that virtual page. The simulator should read in a file containing virtual addresses in decimal, one address per line. If the corresponding page is memory, just record a page hit. If it is not in memory, call a page replacement procedure to pick a page to evict (i.e., an entry in the table to overwrite) and record a page miss. No page transport actually occurs. Generate a file consisting of random addresses and test the performance for both LRU and FIFO. Now generate an address file in which x percent of the addresses are four bytes higher than the previous one (to simulate locality). Run tests for various values of x and report on your results.

39. The program of Fig. 6-26 has a fatal race condition because two threads access shared variables in an uncontrolled way, without using semaphores or any other mutual exclusion technique. Run this program and see how long it takes to hang. If you cannot make it hang, modify it to increase the size of the window of vulnerability by putting some computing between adjusting *m.in* and *m.out* and testing them. How much computing do you have to put in before it fails, say, once an hour?

40. Write a program for UNIX or Windows 7 that takes as input the name of a directory. The program should print a list of the files in the directory, one line per file, and after the file name, print the size of the file. Print the file names in the order they occur in the directory. Unused slots in the directory should be listed as (unused).

This page intentionally left blank

7

THE ASSEMBLY LANGUAGE LEVEL

In Chapters 4, 5, and 6 we discussed three different levels present on most contemporary computers. This chapter is concerned primarily with another level that is present on all computers: the assembly language level. The assembly language level differs in a significant respect from the microarchitecture, ISA, and operating system machine levels—it is implemented by translation rather than by interpretation.

Programs that convert a user's program written in some language to another language are called **translators**. The language in which the original program is written is called the **source language** and the language to which it is converted is called the **target language**. Both the source language and the target language define levels. If a processor that can directly execute programs written in the source language is available, there is no need to translate the source program into the target language.

Translation is used when a processor (either hardware or an interpreter) is available for the target language but not for the source language. If the translation has been performed correctly, running the translated program will give precisely the same results as the execution of the source program would have given had a processor for it been available. Consequently, it is possible to implement a new level for which there is no processor by first translating programs written for that level to a target level and then executing the resulting target-level programs.

It is important to note the difference between translation, on the one hand, and interpretation, on the other hand. In translation, the original program in the source

language is not directly executed. Instead, it is converted to an equivalent program called an **object program** or **executable binary program** whose execution is carried out only after the translation has been completed. In translation, there are two distinct steps:

1. Generation of an equivalent program in the target language.
2. Execution of the newly generated program.

These two steps do not occur simultaneously. The second step does not begin until the first has been completed. In interpretation, there is only one step: executing the original source program. No equivalent program need be generated first, although sometimes the source program is converted to an intermediate form (e.g., Java byte code) for easier interpretation.

While the object program is being executed, only three levels are in evidence: the microarchitecture level, the ISA level, and the operating system machine level. Consequently, three programs—the user's object program, the operating system, and the microprogram (if any)—can be found in the computer's memory at run time. All traces of the original source program have vanished. Thus the number of levels present at execution time may differ from the number of levels present before translation. It should be noted, however, that although we define a level by the instructions and linguistic constructs available to its programmers (and not by the implementation technique), other authors sometimes make a greater distinction between levels implemented by execution-time interpreters and levels implemented by translation.

7.1 INTRODUCTION TO ASSEMBLY LANGUAGE

Translators can be roughly divided into two groups, depending on the relationship between the source language and the target language. When the source language is essentially a symbolic representation for a numerical machine language, the translator is called an **assembler** and the source language is called an **assembly language**. When the source language is a high-level language such as Java or C and the target language is either a numerical machine language or a symbolic representation for one, the translator is called a **compiler**.

7.1.1 What Is an Assembly Language?

A pure assembly language is a language in which each statement produces exactly one machine instruction. In other words, there is a one-to-one correspondence between machine instructions and statements in the assembly program. If each line in the assembly language program contains exactly one statement and

each machine word contains exactly one machine instruction, then an *n*-line assembly program will produce an *n*-instruction machine language program.

The reason that people use assembly language, as opposed to programming in machine language (in binary or hexadecimal), is that it is much easier to program in assembly language. The use of symbolic names and symbolic addresses instead of binary or hexadecimal ones makes an enormous difference. Most people can remember that the abbreviations for add, subtract, multiply, and divide are ADD, SUB, MUL, and DIV, but few can remember the corresponding numerical values the machine uses. The assembly language programmer need only remember the symbolic names because the assembler translates them to the machine instructions.

The same remarks apply to addresses. The assembly language programmer can give symbolic names to memory locations and have the assembler worry about supplying the correct numerical values. The machine language programmer must always work with the numerical values of the addresses. As a consequence, no one programs in machine language today, although people did so decades ago, before assemblers had been invented.

Assembly languages have another property, besides the one-to-one mapping of assembly language statements onto machine instructions, that distinguishes them from high-level languages. The assembly programmer has access to all the features and instructions available on the target machine. The high-level language programmer does not. For example, if the target machine has an overflow bit, an assembly language program can test it, but a Java program cannot test it. An assembly language program can execute every instruction in the instruction set of the target machine, but the high-level language program cannot. In short, everything that can be done in machine language can be done in assembly language, but many instructions, registers, and similar features are not available for the high-level language programmer to use. Languages for system programming, like C, are a cross between these types, with the syntax of a high-level language but with some of the access to the machine of an assembly language.

One final difference that is worth making explicit is that an assembly language program can run only on one family of machines, whereas a program written in a high-level language can potentially run on many machines. For many applications, this ability to move software from one machine to another is of great practical importance.

7.1.2 Why Use Assembly Language?

Assembly language programming is difficult. Make no mistake about that. It is not for wimps and weaklings. Furthermore, writing a program in assembly language takes much longer than writing the same program in a high-level language. It also takes much longer to debug and is much harder to maintain.

Under these conditions, why would anyone ever program in assembly language? There are two reasons: performance and access to the machine. First of

all, an expert assembly language programmer working very hard can sometimes produce code that is much smaller and much faster than a high-level language programmer can. For some applications, speed and size are critical. Many embedded applications, such as the code on a smart card or RFID card, device drivers, string-manipulation libraries, BIOS routines, and the inner loops of performance-critical real-time applications fall in this category.

Second, some procedures need complete access to the hardware, something usually impossible in high-level languages. For example, the low-level interrupt and trap handlers in an operating system and the device controllers in many embedded real-time systems fall into this category.

Besides these reasons for programming in assembly language, there are also two additional reasons for studying it. First, a compiler must either produce output used by an assembler or perform the assembly process itself. Thus understanding assembly language is essential to understanding how compilers work. Someone has to write the compiler (and its assembler) after all.

Second, studying assembly language exposes the real machine to view. For students of computer architecture, writing some assembly code is the only way to get a feel for what a machine is really like at the architectural level.

7.1.3 Format of an Assembly Language Statement

Although the structure of an assembly language statement closely mirrors the structure of the machine instruction that it represents, assembly languages for different machines sufficiently resemble one another to allow a discussion of assembly language in general. Figure 7-1 shows fragments of assembly language programs for the x86 which performs the computation $N = I + J$. The statements below the blank line are commands to the assembler to reserve memory for the variables I, J, and N and are not symbolic representations of machine instructions.

Label	Opcode	Operands	Comments
FORMULA:	MOV	EAX,I	; register EAX = I
	ADD	EAX,J	; register EAX = I + J
	MOV	N,EAX	; N = I + J
I	DD	3	; reserve 4 bytes initialized to 3
J	DD	4	; reserve 4 bytes initialized to 4
N	DD	0	; reserve 4 bytes initialized to 0

Figure 7-1. Computation of $N = I + J$ on the x86.

Several assemblers exist for the Intel family (i.e., x86), each with a different syntax. In this chapter we will use the Microsoft MASM assembly language for our example. There are many assemblers for the ARM, but the syntax is comparable to the x86 assembler, so one example should siffice.

Assembly language statements have up to four parts: first, a label field, second, an operation (opcode) field, third, an operand field, and fourth, a comments field. None of these is mandatory. Labels, which are used to provide symbolic names for memory addresses, are needed on executable statements so that the statements can be branched to. Additionally, they are needed for data words to permit the data stored there to be accessible by symbolic name. If a statement is labeled, the label (usually) begins in column 1.

The example of Fig. 7-1 has four labels: *FORMULA*, *I*, *J*, and *N*. The MASM requires colons on code labels but not on data labels. There is nothing fundamental about this. Other assemblers may have other requirements. Nothing in the underlying architecture suggests one choice or the other. One advantage of the colon notation is that with it a label can appear by itself on a line, with the opcode in column 1 of the next line. This style is convenient for compilers to generate. Without the colon, there would be no way to tell a label on a line all by itself from an opcode on a line all by itself. The colon eliminates this potential ambiguity.

It is an unfortunate characteristic of some assemblers that labels are restricted to six or eight characters. In contrast, most high-level languages allow the use of arbitrarily long names. Long, well-chosen names make programs much more readable and understandable by other people.

Each machine has some registers, so they need names. The x86 registers have names like EAX, EBX, ECX, and so on.

The opcode field contains either a symbolic abbreviation for the opcode—if the statement is a symbolic representation for a machine instruction—or a command to the assembler itself. The choice of an appropriate name is just a matter of taste, and different assembly language designers often make different choices. The designers of the MASM assembler decided to use MOV for both loading a register from memory and storing a register into memory but they could have chosen MOVE or LOAD and STORE.

Assembly progams often need to reserve space for variables. The MASM assembly language designers chose DD (Define Double), since a word on the 8088 was 16 bits.

The operand field of an assembly language statement is used to specify the addresses and registers used as operands by the machine instruction. The operand field of an integer addition instruction tells what is to be added to what. The operand field of a branch instruction tells where to branch to. Operands can be registers, constants, memory locations, and so on.

The comments field provides a place for programmers to put helpful explanations of how the program works for the benefit of other programmers who may subsequently use or modify the program (or for the benefit of the original programmer a year later). An assembly language program without such documentation is nearly incomprehensible to all programmers, frequently including the author as well. The comments field is solely for human consumption; it has no effect on the assembly process or on the generated program.

7.1.4 Pseudoinstructions

In addition to specifying which machine instructions to execute, an assembly language program can also contain commands to the assembler itself, for example, asking it to allocate some storage or to eject to a new page on the listing. Commands to the assembler itself are called **pseudoinstructions** or sometimes **assembler directives**. We have already seen a typical pseudoinstruction in Fig. 7-1(a): DD. Some other pseudoinstructions are listed in Fig. 7-2. These are taken from the Microsoft MASM assembler for the x86.

Pseudoinstruction	Meaning
SEGMENT	Start a new segment (text, data, etc.) with certain attributes
ENDS	End the current segment
ALIGN	Control the alignment of the next instruction or data
EQU	Define a new symbol equal to a given expression
DB	Allocate storage for one or more (initialized) bytes
DW	Allocate storage for one or more (initialized) 16-bit (word) data items
DD	Allocate storage for one or more (initialized) 32-bit (double) data items
DQ	Allocate storage for one or more (initialized) 64-bit (quad) data items
PROC	Start a procedure
ENDP	End a procedure
MACRO	Start a macro definition
ENDM	End a macro definition
PUBLIC	Export a name defined in this module
EXTERN	Import a name from another module
INCLUDE	Fetch and include another file
IF	Start conditional assembly based on a given expression
ELSE	Start conditional assembly if the IF condition above was false
ENDIF	End conditional assembly
COMMENT	Define a new start-of-comment character
PAGE	Generate a page break in the listing
END	Terminate the assembly program

Figure 7-2. Some of the pseudoinstructions available in the MASM assembler (MASM).

The SEGMENT pseudoinstruction starts a new segment, and ENDS terminates one. It is allowed to start a text segment, with code, then start a data segment, then go back to the text segment, and so on.

ALIGN forces the next line, usually data, to an address that is a multiple of its argument. For example, if the current segment has 61 bytes of data already, then after ALIGN 4 the next address allocated will be 64.

EQU is used to give a symbolic name to an expression. For example, after the pseudoinstruction

```
BASE  EQU  1000
```

the symbol BASE can be used everywhere instead of 1000. The expression that follows the EQU can involve multiple defined symbols combined with arithmetic and other operators, as in

```
LIMIT  EQU  4 * BASE + 2000
```

Most assemblers, including MASM, require that a symbol be defined before it is used in an expression like this.

The next four pseudoinstructions, DB, DW, DD, and DQ, allocate storage for one or more variables of size 1, 2, 4, or 8 bytes, respectively. For example,

```
TABLE DB 11, 23, 49
```

allocates space for 3 bytes and initializes them to 11, 23, and 49, respectively. It also defines the symbol *TABLE* and sets it equal to the address where 11 is stored.

The PROC and ENDP pseudoinstructions define the beginning and end of assembly language procedures, respectively. Procedures in assembly language have the same function as procedures in other programming languages. Similarly, MACRO and ENDM delimit the scope of a macro definition. We will study macros later in this chapter.

The next two pseudoinstructions, PUBLIC and EXTERN, control the visibility of symbols. It is common to write programs as a collection of files. Frequently, a procedure in one file needs to call a procedure or access a data word defined in another file. To make this cross-file referencing possible, a symbol that is to be made available to other files is exported using PUBLIC. Similarly, to prevent the assembler from complaining about the use of a symbol that is not defined in the current file, the symbol can be declared as EXTERN, which tells the assembler that it will be defined in some other file. Symbols that are not declared in either of these pseudoinstructions have a scope of the local file. This default means that using, say, *FOO* in multiple files does not generate a conflict because each definition is local to its own file.

The INCLUDE pseudoinstruction causes the assembler to fetch another file and include it bodily into the current one. Such included files often contain definitions, macros, and other items needed in multiple files.

Many assemblers, support conditional assembly. For example,

```
WORDSIZE EQU 32
IF WORDSIZE GT 32
WSIZE:    DD   64
ELSE
WSIZE:    DD   32
ENDIF
```

allocates a single 32-bit word and calls its address *WSIZE*. The word is initialized to either 64 or 32, depending on the value of *WORDSIZE*, in this case, 32. Typically this construction would be used to write a program that could be assembled for either 32-bit mode or 64-bit mode. IF and ENDIF, then by changing a single definition, *WORDSIZE*, the program can automatically be set to assemble for either size. Using this approach, it is possible to maintain one source program for multiple (different) target machines, which makes software development and maintenance easier. In many cases, all the machine-dependent definitions, like *WORDSIZE*, are collected into a single file, with different versions for different machines. By including the right definitions file, the program can be easily recompiled for different machines.

The COMMENT pseudoinstruction allows the user to change the comment delimiter to something other than semicolon. PAGE is used to control the listing the assembler can produce, if requested. END marks the end of the program.

Many other pseudoinstructions exist in MASM. Other x86 assemblers have a different collection of pseudoinstructions available because they are dictated not by the underlying architecture, but by the taste of the assembler writer.

7.2 MACROS

Assembly language programmers frequently need to repeat sequences of instructions several times within a program. The most obvious way to do so is simply to write the required instructions wherever they are needed. If a sequence is long, however, or must be used a large number of times, writing it repeatedly becomes tedious.

An alternative approach is to make the sequence into a procedure and call it wherever it is needed. This strategy has the disadvantage of requiring a procedure call instruction and a return instruction to be executed every time a sequence is needed. If the sequences are short—for example, two instructions—but are used frequently, the procedure call overhead may significantly slow the program down. Macros provide an easy and efficient solution to the problem of repeatedly needing the same or nearly the same sequences of instructions.

7.2.1 Macro Definition, Call, and Expansion

A **macro definition** is a way to give a name to a piece of text. After a macro has been defined, the programmer can write the macro name instead of the piece of program. A macro is, in effect, an abbreviation for a piece of text. Figure 7-3(a) shows an assembly language program for the x86 that exchanges the contents of the variables *p* and *q* twice. These sequences could be defined as macros, as shown in Fig. 7-3(b). After its definition, every occurrence of *SWAP* causes it to be replaced by the four lines:

```
MOV EAX,P
MOV EBX,Q
MOV Q,EAX
MOV P,EBX
```

The programmer has defined *SWAP* as an abbreviation for the four statements shown above.

```
MOV   EAX,P
MOV   EBX,Q
MOV   Q,EAX
MOV   P,EBX

MOV   EAX,P
MOV   EBX,Q
MOV   Q,EAX
MOV   P,EBX
```

(a)

```
SWAP  MACRO
      MOV EAX,P
      MOV EBX,Q
      MOV Q,EAX
      MOV P,EBX
      ENDM

      SWAP

      SWAP
```

(b)

Figure 7-3. Assembly language code for interchanging P and Q twice. (a) Without a macro. (b) With a macro.

Although different assemblers have slightly different notations for defining macros, all require the same basic parts in a macro definition:

1. A macro header giving the name of the macro being defined.
2. The text that is the body of the macro.
3. A pseudoinstruction marking the end of the definition (e.g., ENDM).

When the assembler encounters a macro definition, it saves it in a macro definition table for subsequent use. From that point on, whenever the name of the macro (*SWAP* in the example of Fig. 7-3) appears as an opcode, the assembler replaces it by the macro body. The use of a macro name as an opcode is known as a **macro call** and its replacement by the macro body is called **macro expansion**.

Macro expansion occurs during the assembly process and not during execution of the program. This point is important. The program of Fig. 7-3(a) and that of Fig. 7-3(b) will produce precisely the same machine language code. Looking only at the machine language program, it is impossible to tell whether or not any macros were involved in its generation. The reason is that once macro expansion has been completed the macro definitions are discarded by the assembler. No trace of them is left in the generated program.

Macro calls should not be confused with procedure calls. The basic difference is that a macro call is an instruction to the assembler to replace the macro name

with the macro body. A procedure call is a machine instruction that is inserted into the object program and that will later be executed to call the procedure. Figure 7-4 compares macro calls with procedure calls.

Item	Macro call	Procedure call
When is the call made?	During assembly	During program execution
Is the body inserted into the object program every place the call is made?	Yes	No
Is a procedure call instruction inserted into the object program and later executed?	No	Yes
Must a return instruction be used after the call is done?	No	Yes
How many copies of the body appear in the object program?	One per macro call	One

Figure 7-4. Comparison of macro calls with procedure calls.

Conceptually, it is best to think of the assembly process as taking place in two passes. On pass one, all the macro definitions are saved and the macro calls expanded. On pass two, the resulting text is processed as though it was in the original program. In this view, the source program is read in and is then transformed into another program from which all macro definitions have been removed, and in which all macro calls have been replaced by their bodies. The resulting output, an assembly language program containing no macros at all, is then fed into the assembler.

It is important to keep in mind that a program is a string of characters including letters, digits, spaces, punctuation marks, and "carriage returns" (change to a new line). Macro expansion consists of replacing certain substrings of this string with other character strings. A macro facility is a technique for manipulating character strings, without regard to their meaning.

7.2.2 Macros with Parameters

The macro facility previously described can be used to shorten source programs in which precisely the same sequence of instructions occurs repeatedly. Frequently, however, a program contains several sequences of instructions that are almost but not quite identical, as illustrated in Fig. 7-5(a). Here the first sequence exchanges P and Q, and the second sequence exchanges R and S.

Macro assemblers handle the case of nearly identical sequences by allowing macro definitions to provide **formal parameters** and by allowing macro calls to supply **actual parameters**. When a macro is expanded, each formal parameter

```
MOV   EAX,P
MOV   EBX,Q
MOV   Q,EAX
MOV   P,EBX

MOV   EAX,R
MOV   EBX,S
MOV   S,EAX
MOV   R,EBX
```

(a)

```
CHANGE   MACRO P1, P2
         MOV EAX,P1
         MOV EBX,P2
         MOV P2,EAX
         MOV P1,EBX
         ENDM

         CHANGE P, Q

         CHANGE R, S
```

(b)

Figure 7-5. Nearly identical sequences of statements. (a) Without a macro. (b) With a macro.

appearing in the macro body is replaced by the corresponding actual parameter. The actual parameters are placed in the operand field of the macro call. Figure 7-5(b) shows the program of Fig. 7-5(a) rewritten using a macro with two parameters. The symbols *P1* and *P2* are the formal parameters. Each occurrence of *P1* within a macro body is replaced by the first actual parameter when the macro is expanded. Similarly, *P2* is replaced by the second actual parameter. In the macro call

```
CHANGE P, Q
```

P is the first actual parameter and *Q* is the second actual parameter. Thus the executable programs produced by both parts of Fig. 7-5 are identical. They contain precisely the same instructions with the same operands.

7.2.3 Advanced Features

Most macro processors have a whole raft of advanced features to make life easier for the assembly language programmer. In this section we will take a look at a few of MASM's advanced features. One problem that occurs with all assemblers that support macros is label duplication. Suppose that a macro contains a conditional branch instruction and a label that is branched to. If the macro is called two or more times, the label will be duplicated, causing an assembly error. One solution is to have the programmer supply a different label on each call as a parameter. A different solution (used by MASM) is to allow a label to be declared LOCAL, with the assembler automatically generating a different label on each expansion of the macro. Some other assemblers have a rule that numeric labels are automatically local.

MASM and most other assemblers allow macros to be defined within other macros. This feature is most useful in combination with conditional assembly.

Typically, the same macro is defined in both parts of an IF statement, like this:

```
M1   MACRO
     IF WORDSIZE GT 16
M2        MACRO
          ...
          ENDM
     ELSE
M2        MACRO
     ...
          ENDM
     ENDIF
          ENDM
```

Either way, the macro *M2* will be defined, but the definition will depend on whether the program is being assembled on a 16-bit machine or a 32-bit machine. If *M1* is not called, *M2* will not be defined at all.

Finally, macros can call other macros, including themselves. If a macro is recursive, that is, it calls itself, it must pass itself a parameter that is changed on each expansion and the macro must test the parameter and terminate the recursion when it reaches a certain value. Otherwise the assembler can be put into an infinite loop. If this happens, the assembler must be killed explicitly by the user.

7.2.4 Implementation of a Macro Facility in an Assembler

To implement a macro facility, an assembler must be able to perform two functions: save macro definitions and expand macro calls. We will examine these now.

The assembler must maintain a table of all macro names and, along with each name, a pointer to its stored definition so that it can be retrieved when needed. Some assemblers have a separate table for macro names and some have a combined opcode table in which all machine instructions, pseudoinstructions, and macro names are kept.

When a macro definition is encountered, a table entry is made giving the name of the macro, the number of formal parameters, and a pointer to another table—the macro definition table—where the macro body will be kept. A list of the formal parameters is also constructed at this time for use in processing the definition. The macro body is then read and stored in the macro definition table. Formal parameters occurring within the body are indicated by some special symbol. For example, the internal representation of the macro definition of *CHANGE* with semicolon as "carriage return" and ampersand as the formal parameter symbol might be:

```
MOV EAX,&P1; MOV EBX,&P2; MOV &P2,EAX; MOV &P1,EBX;
```

Within the macro definition table the macro body is simply a character string.

During pass one of the assembly, opcodes are looked up and macros expanded. Whenever a macro definition is encountered, it is stored in the macro table. When a macro is called, the assembler temporarily stops reading input from the input device and starts reading from the stored macro body instead. Formal parameters extracted from the stored macro body are replaced by the actual parameters provided in the call. The presence of an ampersand in front of the formal parameters makes it easy for the assembler to recognize them.

7.3 THE ASSEMBLY PROCESS

In the following sections we will briefly describe how an assembler works. Although each machine has a different assembly language, the assembly process is sufficiently similar on different machines that it is possible to describe it in general.

7.3.1 Two-Pass Assemblers

Because an assembly language program consists of a series of one-line statements, it might at first seem natural to have an assembler that read one statement, then translated it to machine language, and finally output the generated machine language onto a file, along with the corresponding piece of the listing, if any, onto another file. This process would then be repeated until the whole program had been translated. Unfortunately, this strategy does not work.

Consider the situation where the first statement is a branch to *L*. The assembler cannot assemble this statement until it knows the address of statement *L*. Statement *L* may be near the end of the program, making it impossible for the assembler to find the address without first reading almost the entire program. This difficulty is called the **forward reference problem**, because a symbol, *L*, has been used before it has been defined; that is, a reference has been made to a symbol whose definition will only occur later.

Forward references can be handled in two ways. First, the assembler may in fact read the source program twice. Each reading of the source program is called a **pass**; any translator that reads the input program twice is called a **two-pass translator**. On pass one, the definitions of symbols, including statement labels, are collected and stored in a table. By the time the second pass begins, the values of all symbols are known; thus no forward reference remains and each statement can be read, assembled, and output. Although this approach requires an extra pass over the input, it is conceptually simple.

The second approach consists of reading the assembly program once, converting it to an intermediate form, and storing this intermediate form in a table in memory. Then a second pass is made over the table instead of over the source program. If there is enough memory (or virtual memory), this approach saves I/O time. If a listing is to be produced, then the entire source statement, including all

the comments, has to be saved. If no listing is needed, then the intermediate form can be reduced to the bare essentials.

Either way, another task of pass one is to save all macro definitions and expand the calls as they are encountered. Thus defining the symbols and expanding the macros are generally combined into one pass.

7.3.2 Pass One

The principal function of pass one is to build up a table called the **symbol table**, containing the values of all symbols. A symbol is either a label or a value that is assigned a symbolic name by means of a pseudoinstruction such as

```
BUFSIZE EQU 8192
```

In assigning a value to a symbol in the label field of an instruction, the assembler must know what address that instruction will have during execution of the program. To keep track of the execution-time address of the instruction being assembled, the assembler maintains a variable during assembly, known as the **ILC** (**Instruction Location Counter**). This variable is set to 0 at the beginning of pass one and incremented by the instruction length for each instruction processed, as shown in Fig. 7-6. This example is for the x86.

Label	Opcode	Operands	Comments	Length	ILC
MARIA:	MOV	EAX, I	EAX = I	5	100
	MOV	EBX, J	EBX = J	6	105
ROBERTA:	MOV	ECX, K	ECX = K	6	111
	IMUL	EAX, EAX	EAX = I * I	2	117
	IMUL	EBX, EBX	EBX = J * J	3	119
	IMUL	ECX, ECX	ECX = K * K	3	122
MARILYN:	ADD	EAX, EBX	EAX = I * I + J * J	2	125
	ADD	EAX, ECX	EAX = I * I + J * J + K * K	2	127
STEPHANY:	JMP	DONE	branch to DONE	5	129

Figure 7-6. The instruction location counter (ILC) keeps track of the address where the instructions will be loaded in memory. In this example, the statements prior to MARIA occupy 100 bytes.

Pass one of most assemblers uses at least three internal tables: the symbol table, the pseudoinstruction table, and the opcode table. If needed, a literal table is also kept. The symbol table has one entry for each symbol, as illustrated in Fig. 7-7. Symbols are defined either by using them as labels or by explicit definition (e.g., EQU). Each symbol table entry contains the symbol itself (or a pointer to it), its numerical value, and sometimes other information. This additional information may include

1. The length of data field associated with symbol.
2. The relocation bits. (Does the symbol change value if the program is loaded at a different address than the assembler assumed?)
3. Whether or not the symbol is to be accessible outside the procedure.

Symbol	Value	Other information
MARIA	100	
ROBERTA	111	
MARILYN	125	
STEPHANY	129	

Figure 7-7. A symbol table for the program of Fig. 7-6.

The opcode table contains at least one entry for each symbolic opcode (mnemonic) in the assembly language. Figure 7-8 shows part of an opcode table. Each entry contains the symbolic opcode, two operands, the opcode's numerical value, the instruction length, and a type number that separates the opcodes into groups depending on the number and kind of operands.

Opcode	First operand	Second operand	Hex opcode	Instruction length	Instruction class
AAA	—	—	37	1	6
ADD	EAX	immed32	05	5	4
ADD	reg	reg	01	2	19
AND	EAX	immed32	25	5	4
AND	reg	reg	21	2	19

Figure 7-8. A few excerpts from the opcode table for an x86 assembler.

As an example, consider the opcode ADD. If an ADD instruction contains EAX as the first operand and a 32-bit constant (immed32) as the second one, then opcode 0x05 is used and the instruction length is 5 bytes. (Constants that can be expressed in 8 or 16 bits use different opcodes, not shown.) If ADD is used with two registers as operands, the instruction is 2 bytes, with opcode 0x01. The (arbitrary) instruction class 19 would be given to all opcode-operand combinations that follow the same rules and should be processed the same way as ADD with two register operands. The instruction class effectively designates a procedure within the assembler that is called to process all instructions of a given type.

Some assemblers allow programmers to write instructions using immediate addressing even though no corresponding target language instruction exists. Such "pseudoimmediate" instructions are handled as follows. The assembler allocates

memory for the immediate operand at the end of the program and generates an instruction that references it. For instance, the IBM 360 mainframe and its successors have no immediate instructions. Nevertheless, programmers may write

```
L 14,=F'5'
```

to load register 14 with a full word constant 5. In this manner, the programmer avoids explicitly writing a pseudoinstruction to allocate a word initialized to 5, giving it a label, and then using that label in the L instruction. Constants for which the assembler automatically reserves memory are called **literals**. In addition to saving the programmer a little writing, literals improve the readability of a program by making the value of the constant apparent in the source statement. Pass one of the assembler must build a table of all literals used in the program. All three of our example computers have immediate instructions, so their assemblers do not provide literals. Immediate instructions are quite common nowadays, but formerly they were unusual. It is likely that the widespread use of literals made it clear to machine designers that immediate addressing was a good idea. If literals are needed, a literal table is maintained during assembly, with a new entry made each time a literal is encountered. After the first pass, this table is sorted and duplicates removed.

Figure 7-9 shows a procedure that could serve as a basis for pass one of an assembler. The style of programming is noteworthy in itself. The procedure names have been chosen to give a good indication of what the procedures do. Most important, Fig. 7-9 represents an outline of pass one which, although not complete, forms a good starting point. It is short enough to be easily understood and it makes clear what the next step must be—namely, to write the procedures used in it.

Some of these procedures will be relatively short, such as *check_for_symbol*, which just returns the symbol as a character string if there is one and null if there is not. Other procedures, such as *get_length_of_type1* and *get_length_of_type2*, may be longer and may call other procedures. In general, the number of types will not be two, of course, but will depend on the language being assembled and how many types of instructions it has.

Structuring programs in this way has other advantages in addition to ease of programming. If the assembler is being written by a group of people, the various procedures can be parceled out among the programmers. All the (nasty) details of getting the input are hidden away in *read_next_line*. If they should change—for example, due to an operating system change—only one subsidiary procedure is affected, and no changes are needed to the *pass_one* procedure itself.

As it reads the program, pass one of the assembler has to parse each line to find the opcode (e.g., ADD), look up its type (basically, the pattern of operands), and compute the instruction's length. This information is also needed on the second pass, so it is possible to write it out explicitly to eliminate the need to parse the line from scratch next time. However, rewriting the input file causes more I/O to

```
public static void pass_one( ) {
   // This procedure is an outline of pass one of a simple assembler.
   boolean more_input = true;                  // flag that stops pass one
   String line, symbol, literal, opcode;       // fields of the instruction
   int location_counter, length, value, type;  // misc. variables
   final int END_STATEMENT = –2;               // signals end of input

   location_counter = 0;                       // assemble first instruction at 0
   initialize_tables( );                       // general initialization

   while (more_input) {                        // more_input set to false by END
      line = read_next_line( );                // get a line of input
      length = 0;                              // # bytes in the instruction
      type = 0;                                // which type (format) is the instruction

      if (line_is_not_comment(line)) {
         symbol = check_for_symbol(line);      // is this line labeled?
         if (symbol != null)                   // if it is, record symbol and value
            enter_new_symbol(symbol, location_counter);
         literal = check_for_literal(line);    // does line contain a literal?
         if (literal != null)                  // if it does, enter it in table
            enter_new_literal(literal);

         // Now determine the opcode type.  –1 means illegal opcode.
         opcode = extract_opcode(line);        // locate opcode mnemonic
         type = search_opcode_table(opcode); // find format, e.g. OP REG1,REG2
         if (type < 0)                         // if not an opcode, is it a pseudoinstruction?
            type = search_pseudo_table(opcode);
         switch(type) {                        // determine the length of this instruction
            case 1: length = get_length_of_type1(line);  break;
            case 2: length = get_length_of_type2(line);  break;
            // other cases here
         }
      }

      write_temp_file(type, opcode, length, line);    // useful info for pass two
      location_counter = location_counter + length;       // update loc_ctr
      if (type == END_STATEMENT) {             // are we done with input?
         more_input = false;                   // if so, perform housekeeping tasks
         rewind_temp_for_pass_two( );          // like rewinding the temp file
         sort_literal_table( );                // and sorting the literal table
         remove_redundant_literals( );         // and removing duplicates from it
      }
   }
}
```

Figure 7-9. Pass one of a simple assembler.

occur. Whether it is better to do more I/O to eliminate parsing or less I/O and more parsing depends on the relative speed of the CPU and disk, the efficiency of the file system, and other factors. In this example we will write out a temporary file containing the type, opcode, length, and actual input line. It is this line that pass two reads instead of the raw input file.

When the END pseudoinstruction is read, pass one is over. The symbol table and literal tables can be sorted at this point if needed. The sorted literal table can be checked for duplicate entries, which can be removed.

7.3.3 Pass Two

The function of pass two is to generate the object program and possibly print the assembly listing. In addition, pass two must output certain information needed by the linker for linking up procedures assembled at different times into a single executable file. Figure 7-10 shows a sketch of a procedure for pass two.

```
public static void pass_two( ) {
   // This procedure is an outline of pass two of a simple assembler.
   boolean more_input = true;                  // flag that stops pass two
   String line, opcode;                        // fields of the instruction
   int location_counter, length, type;         // misc. variables
   final int END_STATEMENT = –2;               // signals end of input
   final int MAX_CODE = 16;                    // max bytes of code per instruction
   byte code[ ] = new byte[MAX_CODE];          // holds generated code per instruction

   location_counter = 0;                       // assemble first instruction at 0

   while (more_input) {                        // more_input set to false by END
      type = read_type( );                     // get type field of next line
      opcode = read_opcode( );                 // get opcode field of next line
      length = read_length( );                 // get length field of next line
      line = read_line( );                     // get the actual line of input

      if (type != 0) {                         // type 0 is for comment lines
         switch(type) {                        // generate the output code
            case 1: eval_type1(opcode, length, line, code);  break;
            case 2: eval_type2(opcode, length, line, code);  break;
            // other cases here
         }
      }

      write_output(code);                      // write the binary code
      write_listing(code, line);               // print one line on the listing
      location_counter = location_counter + length;      // update loc_ctr
      if (type == END_STATEMENT) {             // are we done with input?
         more_input = false;                   // if so, perform housekeeping tasks
         finish_up( );                         // odds and ends
      }
   }
}
```

Figure 7-10. Pass two of a simple assembler.

The operation of pass two is more-or-less similar to that of pass one: it reads the lines one at a time and processes them one at a time. Since we have written the type, opcode, and length at the start of each line (on the temporary file), all of these

are read in to save some parsing. The main work of the code generation is done by the procedures *eval_type1*, *eval_type2*, and so on. Each one handles a particular pattern, such as an opcode and two register operands. It generates the binary code for the instruction and returns it in *code*. Then it is written out. More likely, *write_output* just buffers the accumulated binary code and writes the file to disk in large chunks to reduce disk traffic.

The original source statement and the object code generated from it (in hexadecimal) can then be printed or put into a buffer for later printing. After the ILC has been adjusted, the next statement is fetched.

Up until now it has been assumed that the source program does not contain any errors. Anyone who has ever written a program, in any language, knows how realistic that assumption is. Some of the common errors are as follows:

1. A symbol has been used but not defined.
2. A symbol has been defined more than once.
3. The name in the opcode field is not a legal opcode.
4. An opcode is not supplied with enough operands.
5. An opcode is supplied with too many operands.
6. A number contains an invalid character like 143G6.
7. Illegal register use (e.g., a branch to a register).
8. The END statement is missing.

Programmers are quite ingenious at thinking up new kinds of errors to make. Undefined symbol errors are frequently caused by typing errors, so a clever assembler could try to figure out which of the defined symbols most resembles the undefined one and use that instead. Little can be done about correcting most other errors. The best thing for the assembler to do with an errant statement is to print an error message and try to continue assembly.

7.3.4 The Symbol Table

During pass one of the assembly process, the assembler accumulates information about symbols and their values that must be stored in the symbol table for lookup during pass two. Several different ways are available for organizing the symbol table. We will briefly describe some of them below. All of them attempt to simulate an **associative memory**, which conceptually is a set of (symbol, value) pairs. Given the symbol, the associative memory must produce the value.

The simplest implementation technique is indeed to implement the symbol table as an array of pairs, the first element of which is (or points to) the symbol and the second of which is (or points to) the value. Given a symbol to look up, the

symbol table routine just searches the table linearly until it finds a match. This method is easy to program but is slow, because, on the average, half the table will have to be searched on each lookup.

Another way to organize the symbol table is to sort it on the symbols and use the **binary search** algorithm to look up a symbol. This algorithm works by comparing the middle entry in the table to the symbol. If the symbol comes before the middle entry alphabetically, the symbol must be located in the first half of the table. If the symbol comes after the middle entry, it must be in the second half of the table. If the symbol is equal to the middle entry, the search terminates.

Assuming that the middle entry is not equal to the symbol sought, we at least know which half of the table to look for it in. Binary search can now be applied to the correct half, which yields either a match, or the correct quarter of the table. Applying the algorithm recursively, a table of size n entries can be searched in about $\log_2 n$ attempts. Obviously, this way is much faster than searching linearly, but it requires maintaining the table in sorted order.

A completely different way of simulating an associative memory is a technique known as **hash coding** or **hashing**. This approach requires having a "hash" function that maps symbols onto integers in the range 0 to $k - 1$. One possible function is to multiply the ASCII codes of the characters in the symbols together, ignoring overflow, and taking the result modulo k or dividing it by a prime number. In fact, almost any function of the input that gives a uniform distribution of the hash values will do. Symbols can be stored by having a table consisting of k **buckets** numbered 0 to $k - 1$. All the (symbol, value) pairs whose symbol hashes to i are stored on a linked list pointed to by slot i in the hash table. With n symbols and k slots in the hash table, the average list will have length n/k. By choosing k approximately equal to n, symbols can be located with only about one lookup on the average. By adjusting k we can reduce table size at the expense of slower lookups. Hash coding is illustrated in Fig. 7-11.

7.4 LINKING AND LOADING

Most programs consist of more than one procedure. Compilers and assemblers generally translate one procedure at a time and put the translated output on disk. Before the program can be run, all the translated procedures must be found and linked together properly. If virtual memory is not available, the linked program must be explicitly loaded into main memory as well. Programs that perform these functions are called by various names, including **linker**, **linking loader**, and **linkage editor**. The complete translation of a source program requires two steps, as shown in Fig. 7-12:

1. Compilation or assembly of the source files.
2. Linking of the object modules.

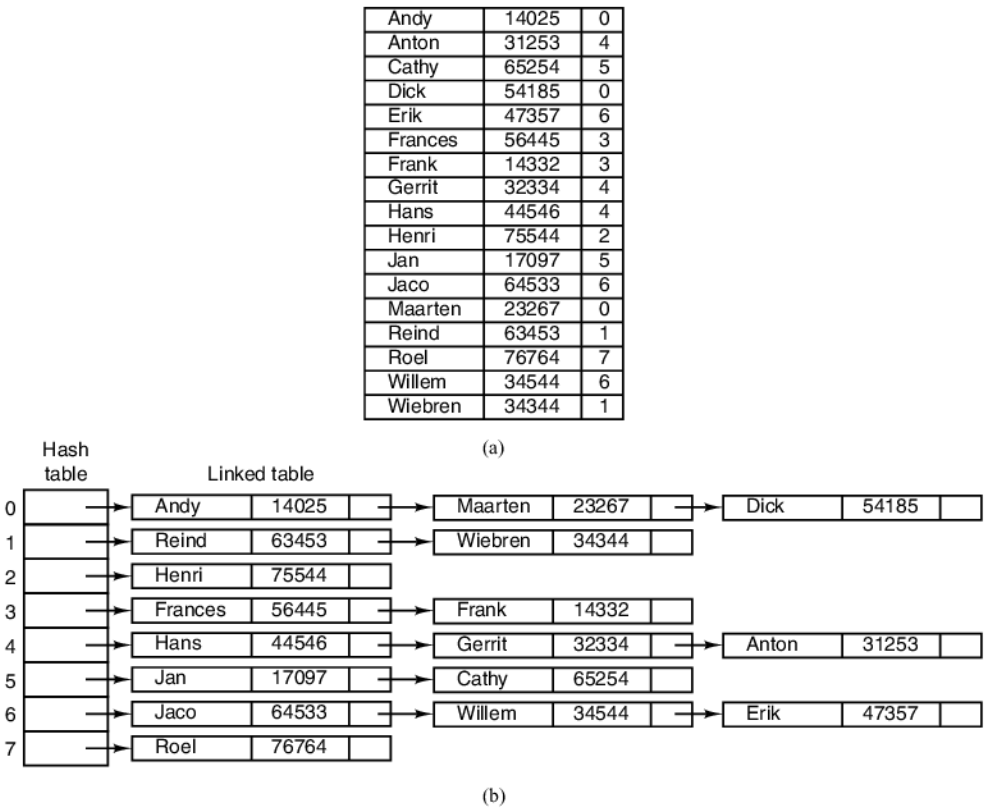

Figure 7-11. Hash coding. (a) Symbols, values, and the hash codes derived from the symbols. (b) Eight-entry hash table with linked lists of symbols and values.

The first step is performed by the compiler or assembler and the second one is performed by the linker.

The translation from source procedure to object module represents a change of level because the source language and target language have different instructions and notation. The linking process, however, does not represent a change of level, since both the linker's input and the linker's output are programs for the same virtual machine. The linker's function is to collect procedures translated separately and link them together to be run as a unit called an **executable binary program**. On Windows systems, the object modules have extension *.obj* and the executable binary programs have extension *.exe*. On UNIX, the object modules have extension *.o*; executable binary programs have no extension.

Compilers and assemblers translate each source file separately for a very good reason. If a compiler or assembler were to read a series of source procedures and immediately produce a ready-to-run machine language program, changing one

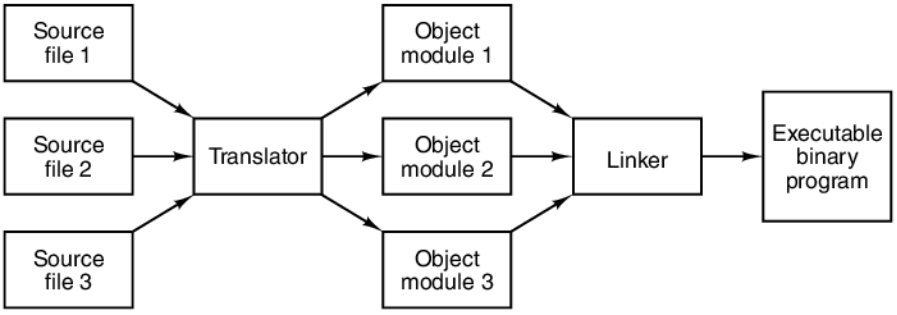

Figure 7-12. Generation of an executable binary program from a collection of independently translated source procedures requires using a linker.

statement in one source procedure would require all the procedures to be retranslated.

If the separate-object-module technique of Fig. 7-12 is used, it is only necessary to retranslate the modified procedure and not the unchanged ones, although it is necessary to relink all the object modules again. Linking is usually much faster than translating, however; thus the two-step process of translating and linking can save a great deal of time during the development of a program. This gain is especially important for programs with hundreds or thousands of modules.

7.4.1 Tasks Performed by the Linker

At the start of pass one of the assembly process, the instruction location counter is set to 0. This step is equivalent to assuming that the object module will be located at (virtual) address 0 during execution. Figure 7-13 shows four object modules for a generic machine. In this example, each module begins with a BRANCH instruction to a MOVE instruction within the module.

In order to run the program, the linker brings the object modules from the disk into main memory to form the image of the executable binary program, as shown in Fig. 7-14(a). The idea is to make an exact image of the executable program's virtual address space inside the linker and position all the object modules at their correct locations. If there is not enough (virtual) memory to form the image, a disk file can be used. Typically, a small section of memory starting at address zero is used for interrupt vectors, communication with the operating system, catching uninitialized pointers, or other purposes, so programs often start above 0. In this figure we have (arbitrarily) started programs at address 100.

The program of Fig. 7-14(a), although loaded into the image of the executable binary file, is not yet ready for execution. Consider what would happen if execution began with the instruction at the beginning of module *A*. The program would not branch to the MOVE instruction as it should, because that instruction is now at

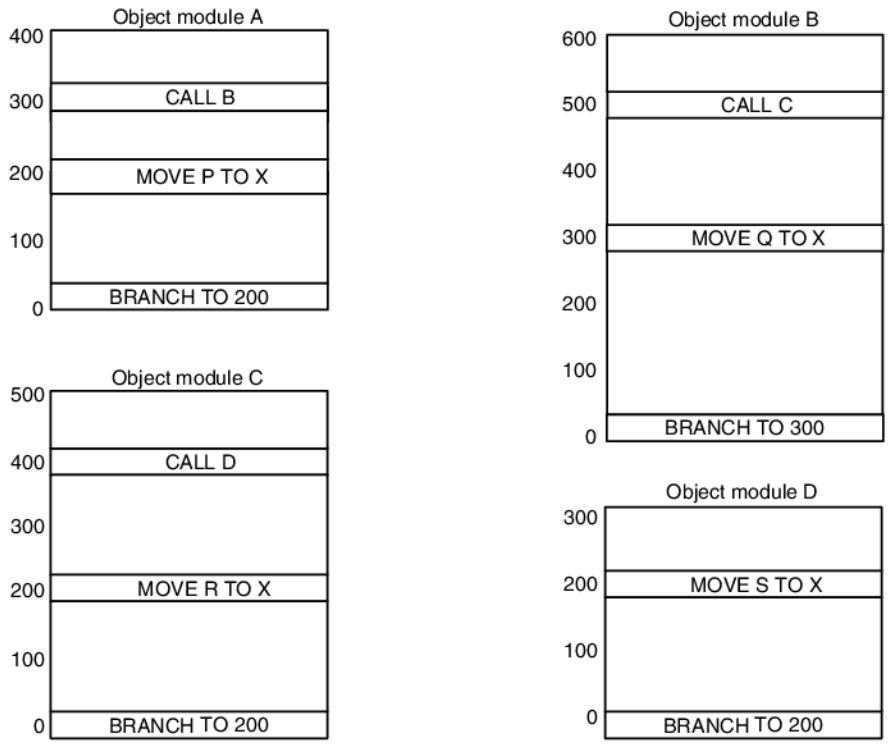

Figure 7-13. Each module has its own address space, starting at 0.

300. In fact, all memory reference instructions will fail for the same reason. Clearly something has to be done.

This problem, called the **relocation problem**, occurs because each object module in Fig. 7-13 represents a separate address space. On a machine with a segmented address space, such as the x86, theoretically each object module could have its own address space by being placed in its own segment. However, OS/2 was the only operating system for the x86 that supported this concept. All versions of Windows and UNIX support only one linear address space, so all the object modules must be merged together into a single address space.

Furthermore, the procedure call instructions in Fig. 7-14(a) will not work either. At address 400, the programmer had intended to call object module *B*, but because each procedure is translated by itself, the assembler has no way of knowing what address to insert into the CALL B instruction. The address of object module *B* is not known until linking time. This problem is called the **external reference** problem. Both of these problems can be solved in a simple way by the linker.

The linker merges the separate address spaces of the object modules into a single linear address space in the following steps:

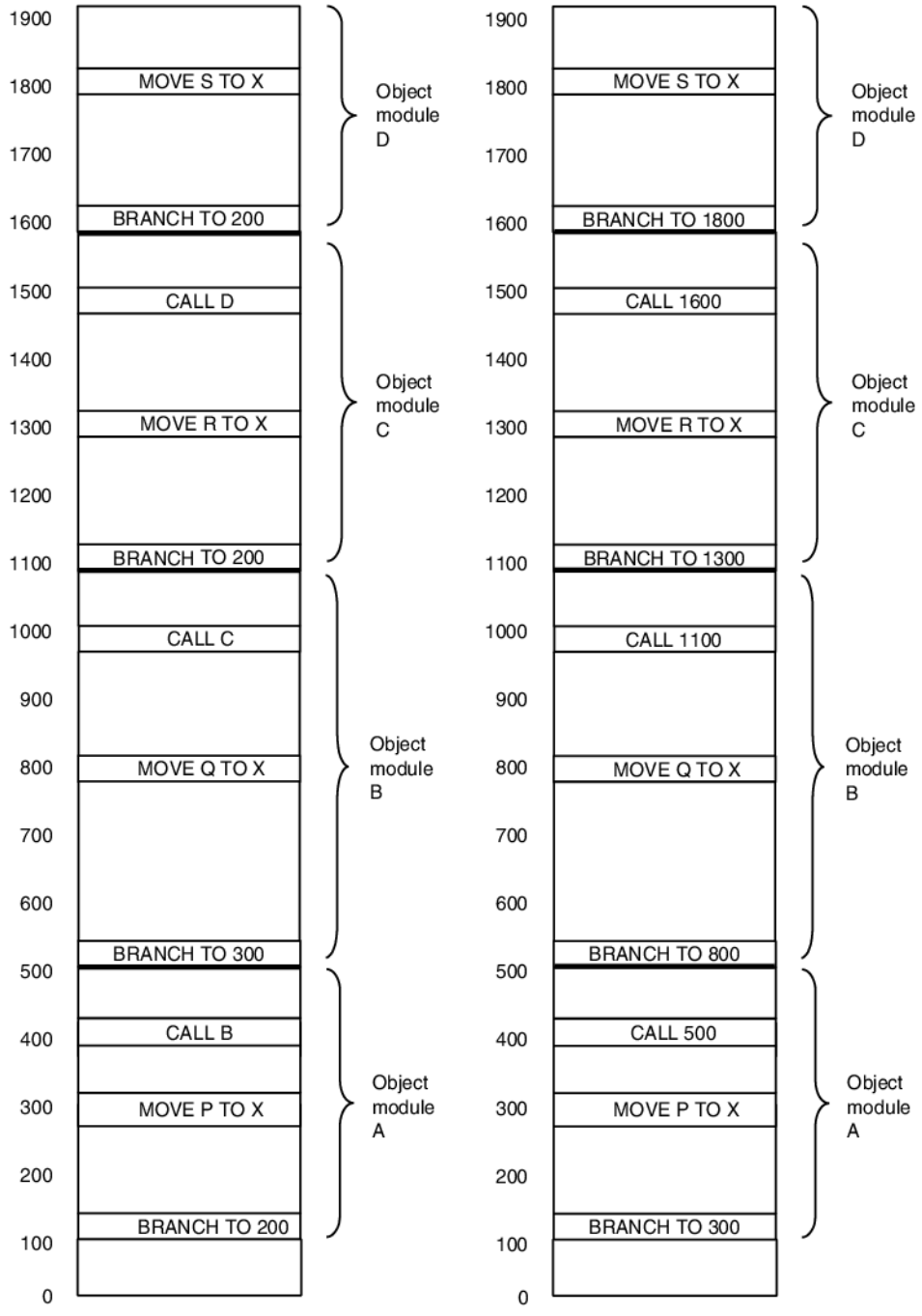

Figure 7-14. (a) The object modules of Fig. 7-13 after being positioned in the binary image but before being relocated and linked. (b) The same object modules after linking and after relocation has been performed.

1. It constructs a table of all the object modules and their lengths.
2. Based on this table, it assigns a base address to each object module.
3. It finds all the instructions that reference memory and adds to each a **relocation constant** equal to the starting address of its module.
4. It finds all the instructions that reference other procedures and inserts the address of these procedures in place.

The object module table constructed in step 1 is shown for the modules of Fig. 7-14 below. It gives the name, length, and starting address of each module.

Module	Length	Starting address
A	400	100
B	600	500
C	500	1100
D	300	1600

Figure 7-14(b) shows how the address space of Fig. 7-14(a) looks after the linker has performed these steps.

7.4.2 Structure of an Object Module

Object modules often contain six parts, as shown in Fig. 7-15. The first part contains the name of the module, certain information needed by the linker, such as the lengths of the various parts of the module, and sometimes the assembly date.

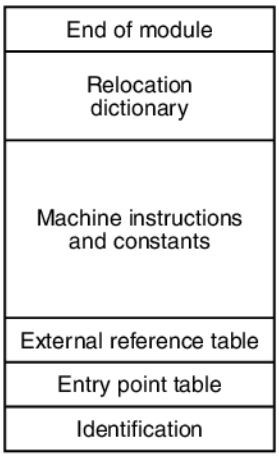

Figure 7-15. The internal structure of an object module produced by a translator. The *Identification* field comes first.

The second part of the object module is a list of the symbols defined in the module that other modules may reference, together with their values. For example, if the module consists of a procedure named *bigbug*, the entry point table will

contain the character string "bigbug" followed by the address to which it corresponds. The assembly language programmer indicates which symbols are to be declared as **entry points** by using a pseudoinstruction such as PUBLIC in Fig. 7-2.

The third part of the object module consists of a list of the symbols that are used in the module but which are defined in other modules, along with a list of which machine instructions use which symbols. The linker needs the latter list in order to be able to insert the correct addresses into the instructions that use external symbols. A procedure can call other independently translated procedures by declaring the names of the called procedures to be external. The assembly language programmer indicates which symbols are to be declared as **external symbols** by using a pseudoinstruction such as EXTERN in Fig. 7-2. On some computers entry points and external references are combined into one table.

The fourth part of the object module is the assembled code and constants. This part of the object module is the only one that will be loaded into memory to be executed. The other five parts will be used by the linker to help it do its work and then discarded before execution begins.

The fifth part of the object module is the relocation dictionary. As shown in Fig. 7-14, instructions that contain memory addresses must have a relocation constant added. Since the linker has no way of telling by inspection which of the data words in part four contain machine instructions and which contain constants, information about which addresses are to be relocated is provided in this table. The information may take the form of a bit table, with 1 bit per potentially relocatable address, or an explicit list of addresses to be relocated.

The sixth part is an end-of-module mark, perhaps a checksum to catch errors made while reading the module, and the address at which to begin execution.

Most linkers require two passes. On pass one the linker reads all the object modules and builds up a table of module names and lengths, and a global symbol table consisting of all entry points and external references. On pass two the object modules are read, relocated, and linked one module at a time.

7.4.3 Binding Time and Dynamic Relocation

In a multiprogramming system, a program can be read into main memory, run for a little while, written to disk, and then read back into main memory to be run again. In a large system, with many programs, it is difficult to ensure that a program is read back into the same locations every time.

Figure 7-16 shows what would happen if the already relocated program of Fig. 7-14(b) were reloaded at address 400 instead of address 100 where the linker put it originally. All the memory addresses are incorrect; moreover, the relocation information has long since been discarded. Even if the relocation information were still available, the cost of having to relocate all the addresses every time the program was swapped would be too high.

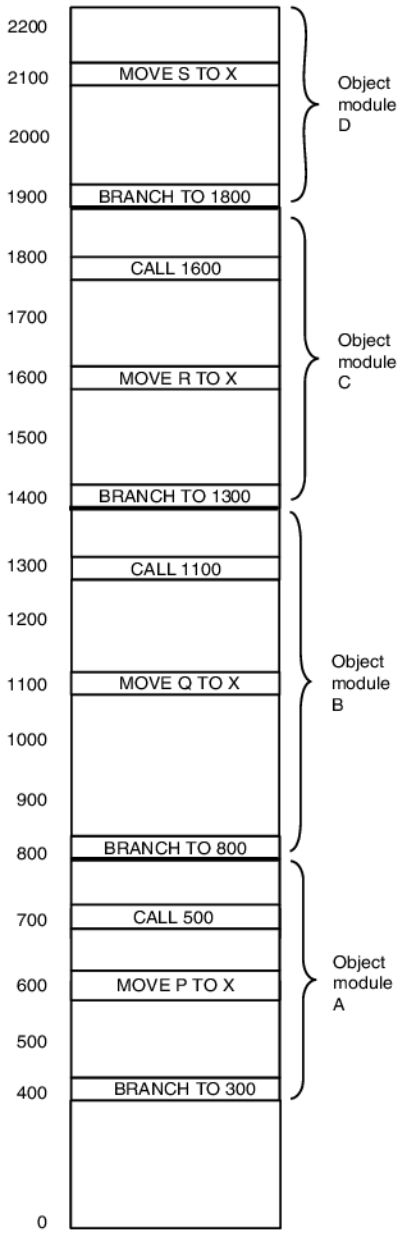

Figure 7-16. The relocated binary program of Fig. 7-14(b) moved up 300 addresses. Many instructions now refer to an incorrect memory address.

The problem of moving programs that have already been linked and relocated is intimately related to the time at which the final binding of symbolic names onto absolute physical memory addresses is completed. When a program is written it contains symbolic names for memory addresses, for example, `BR L`. The time at which the actual main memory address corresponding to *L* is determined is called the **binding time**. At least six possibilities for the binding time exist:

1. When the program is written.
2. When the program is translated.
3. When the program is linked but before it is loaded.
4. When the program is loaded.
5. When a base register used for addressing is loaded.
6. When the instruction containing the address is executed.

If an instruction containing a memory address is moved after binding, it will be incorrect (assuming that the object referred to has also been moved). If the translator produces an executable binary as output, the binding has occurred at translation time, and the program must be run at the address at which the translator expected it to be run at. The linking method described in the preceding section binds symbolic names to absolute addresses during linking, which is why moving programs after linking fails, as shown in Fig. 7-16.

Two related issues are involved here. First, there is the question of when symbolic names are bound to virtual addresses. Second, there is a question of when virtual addresses are bound to physical addresses. Only when both operations have taken place is binding complete. When the linker merges the separate address spaces of the object modules into a single linear address space, it is, in fact, creating a virtual address space. The relocation and linking serve to bind symbolic names onto specific virtual addresses. This observation is true whether or not virtual memory is being used.

Assume for the moment that the address space of Fig. 7-14(b) is paged. It is clear that the virtual addresses corresponding to the symbolic names *A*, *B*, *C*, and *D* have already been determined, even though their physical main memory addresses will depend on the contents of the page table at the time they are used. An executable binary program is really a binding of symbolic names to virtual addresses.

Any mechanism that allows the mapping of virtual addresses onto physical memory addresses to be changed easily will facilitate moving programs around in main memory, even after they have been bound to a virtual address space. One such mechanism is paging. After a program has been moved in main memory, only its page table need be changed, not the program itself.

A second mechanism is the use of a runtime relocation register. The CDC 6600 and its successors had such a register. On machines using this relocation

technique, the register always points to the physical address of the start of the current program. All memory addresses have the contents of the relocation register added to them by the hardware before being sent to the memory. The entire relocation process is transparent to the user programs. They do not even know that it is occurring. When a program is moved, the operating system must update the relocation register. This mechanism is less general than paging because the entire program must be moved as a unit (unless there are separate code and data relocation registers, as on the Intel 8088, in which case it has to be moved as two units).

A third mechanism is possible on machines that can refer to memory relative to the program counter. Many branch instructions are relative to the program counter, which helps. Whenever a program is moved in main memory only the program counter need be updated. A program, all of whose memory references are either relative to the program counter or absolute (e.g., to I/O device registers at absolute addresses) is said to be **position independent**. A position-independent procedure can be placed anywhere within the virtual address space without the need for relocation.

7.4.4 Dynamic Linking

The linking strategy discussed in Sec. 7.4.1 has the property that all procedures that a program might call are linked before the program can begin execution. On a computer with virtual memory, completing all linking before beginning execution does not take advantage of the full capabilities of the virtual memory. Many programs have procedures that are called only under unusual circumstances. For example, compilers have procedures for compiling rarely used statements, plus procedures for handling error conditions that seldom occur.

A more flexible way to link separately compiled procedures is to link each procedure at the time it is first called. This process is known as **dynamic linking**. It was pioneered by MULTICS whose implementation is in some ways still unsurpassed. In the next sections we will look at dynamic linking in several systems.

Dynamic Linking in MULTICS

In the MULTICS form of dynamic linking, associated with each program is a segment, called the **linkage segment**, which contains one block of information for each procedure that might be called. This block of information starts with a word reserved for the virtual address of the procedure and it is followed by the procedure name, which is stored as a character string.

When dynamic linking is being used, procedure calls in the source language are translated into instructions that indirectly address the first word of the corresponding linkage block, as shown in Fig. 7-17(a). The compiler fills this word with either an invalid address or a special bit pattern that forces a trap.

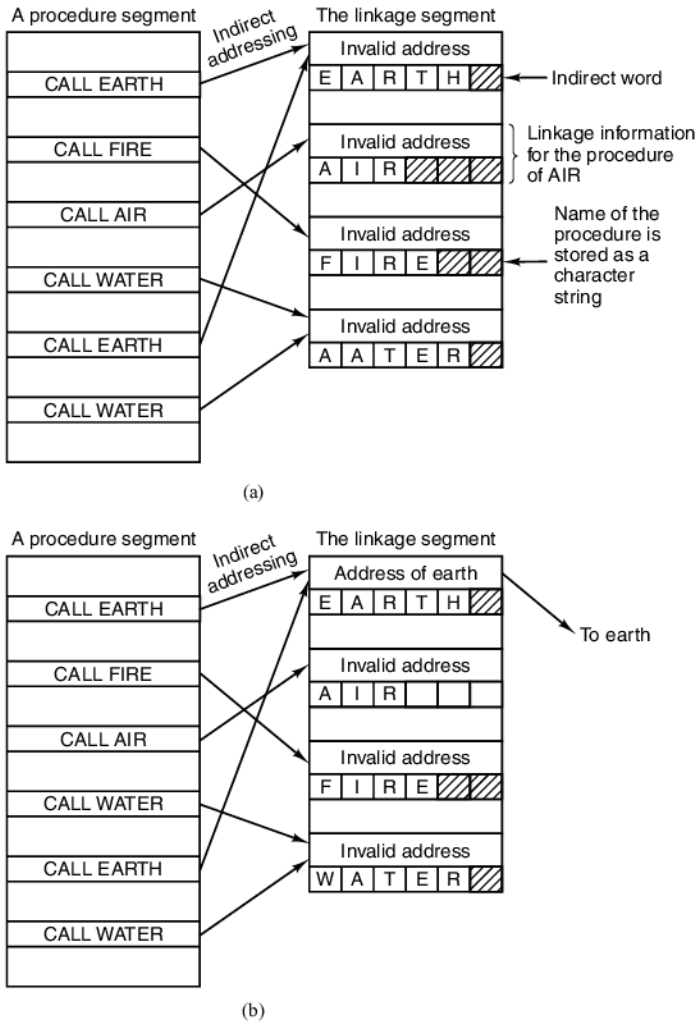

Figure 7-17. Dynamic linking. (a) Before *EARTH* is called. (b) After *EARTH* has been called and linked.

When a procedure in a different segment is called, the attempt to address the invalid word indirectly causes a trap to the dynamic linker. The linker then finds the character string in the word following the invalid address and searches the user's file directory for a compiled procedure with this name. That procedure is then assigned a virtual address, usually in its own private segment, and this virtual address overwrites the invalid address in the linkage segment, as indicated in Fig. 7-17(b). Next, the instruction causing the linkage fault is re-executed, allowing the program to continue from the place it was before the trap.

All subsequent references to that procedure will be executed without causing a linkage fault, for the indirect word now contains a valid virtual address. Consequently, the dynamic linker is invoked only the first time a procedure is called.

Dynamic Linking in Windows

All versions of the Windows operating system support dynamic linking and rely heavily on it. Dynamic linking uses a special file format called a **DLL** (**Dynamic Link Library**). DLLs can contain procedures, data, or both. They are commonly used to allow two or more processes to share library procedures or data. Many DLLs have extension *.dll*, but other extensions are also in use, including *.drv* (for driver libraries) and *.fon* (for font libraries).

The most common form of a DLL is a library consisting of a collection of procedures that can be loaded into memory and accessed by multiple processes at the same time. Figure 7-18 illustrates two processes sharing a DLL file that contains four procedures, *A*, *B*, *C*, and *D*. Program 1 uses procedure *A*; program 2 uses procedure *C*, although they could equally well have used the same procedure.

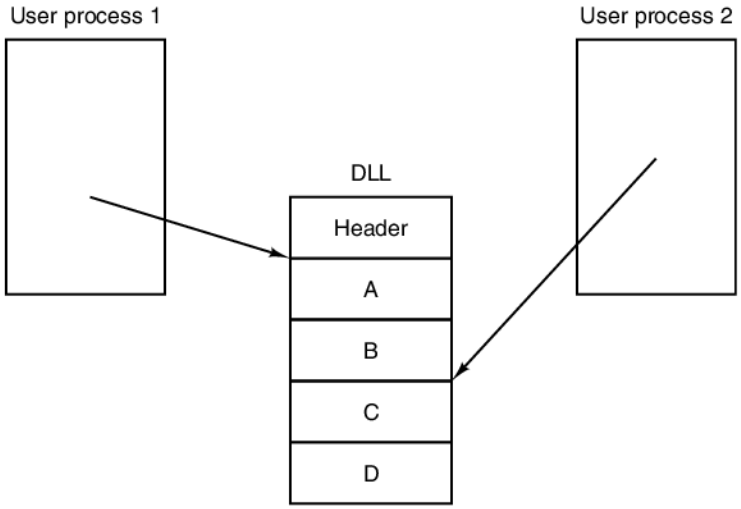

Figure 7-18. Use of a DLL file by two processes.

A DLL is constructed by the linker from a collection of input files. In fact, building a DLL file is very much like building an executable binary program, except that a special flag is given to the linker to tell it to make a DLL. DLLs are commonly constructed from collections of library procedures that are likely to be needed by multiple processes. The interface procedures to the Windows system call library and large graphics libraries are common examples of DLLs. The advantage o using DLLs is saving space in memory and on disk. If some common li-

brary were statically bound to each program using it, it would appear in many executable binaries on the disk and in memory, wasting space. With DLLs, each library appears only once on disk and once in memory.

In addition to saving space, this approach makes it easy to update library procedures, even after the programs using them have been compiled and linked. For commercial software packages, where the users rarely have the source code, using DLLs means that the software vendor can fix bugs in the libraries by just distributing new DLL files over the Internet, without requiring any changes to the main program binaries.

The main difference between a DLL and an executable binary is that a DLL cannot be started and run on its own (because it has no main program). It also has different information in its header. In addition, the DLL as a whole has several extra procedures not related to the procedures in the library. For example, there is one procedure that is automatically called whenever a new process is bound to the DLL and another one that is automatically called whenever a process is unbound from it. These procedures can allocate and deallocate memory or manage other resources needed by the DLL.

There are two ways for a program to bind to a DLL. In the first way, called **implicit linking**, the user's program is statically linked with a special file called an **import library** that is generated by a utility program that extracts certain information from the DLL. The import library provides the glue that allows the user program to access the DLL. A user program can be linked with multiple import libraries. When a program using implicit linking is loaded into memory for execution, Windows examines it to see which DLLs it uses and checks to see if all of them are already in memory. Those that are not in memory are loaded immediately (but not necessarily in their entirety, since they are paged). Some changes are then made to the data structures in the import libraries so the called procedures can be located, somewhat analogous to the changes shown in Fig. 7-17. They also have to be mapped into the program's virtual address space. At this point, the user program is ready to run and can call the procedures in the DLLs as though they had been statically bound with it.

The alternative to implicit linking is (not surprisingly) **explicit linking**. This approach does not require import libraries and does not cause the DLLs to be loaded at the same time the user program is. Instead, the user program makes an explicit call at run time to bind to a DLL, then makes additional calls to get the addresses of procedures it needs. Once these have been found, it can call the procedures. When it is all done, it makes a final call to unbind from the DLL. When the last process unbinds from a DLL, the DLL can be removed from memory.

It is important to realize that a procedure in a DLL does not have any identity of its own (as a thread or process does). It runs in the called's thread and uses the called's stack for its local variables. It can have process-specific static data (as well as shared data) and otherwise behaves the same as a statically-linked procedure. The only essential difference is how the binding to it is performed.

Dynamic Linking in UNIX

The UNIX system has a mechanism similar in essence to DLLs in Windows. It is called a **shared library**. Like a DLL file, a shared library is an archive file containing multiple procedures or data modules that are present in memory at run time and can be bound to multiple processes at the same time. The standard C library and much of the networking code are shared libraries.

UNIX supports only implicit linking, so a shared library consist of two parts: a **host library**, which is statically linked with the executable file, and a **target library**, which is called at run time. While the details differ, the concepts are essentially the same as with DLLs.

7.5 SUMMARY

Although most programs can and should be written in a high-level language, occasional situations exist in which assembly language is needed, at least in part. Programs for resource-poor computers such as smart cards and embedded processors in small consumer devices like clock radios are potential candidates. An assembly language program is a symbolic representation for some underlying machine language program. It is translated to the machine language by a program called an assembler.

When extremely fast execution is critical to the success of some application, a better approach than writing everything in assembly language is to first write the whole program in a high-level language, then measure where it is spending its time, and finally rewrite only those portions of the program that are heavily used. In practice, a small fraction of the code is usually responsible for a large fraction of the execution time.

Many assemblers have a macro facility that allows the programmer to give commonly used code sequences symbolic names for subsequent inclusion. Usually, these macros can be parameterized in a straightforward way. Macros are implemented by a kind of literal string-processing algorithm.

Most assemblers are two pass. Pass one is devoted to building up a symbol table for labels, literals, and explicitly declared identifiers. The symbols can either be kept unsorted and then searched linearly, first sorted and then searched using binary search, or hashed. If symbols do not need to be deleted during pass one, hashing is usually the best method. Pass two does the code generation. Some pseudoinstructions are carried out on pass one and some on pass two.

Independently-assembled programs can be linked together to form an executable binary program that can be run. This work is done by the linker. Its primary tasks are relocation and binding of names. Dynamic linking is a technique in which certain procedures are not linked until they are actually called. Windows DLLs and UNIX shared libraries use dynamic linking.

PROBLEMS

1. For a certain program, 2% of the code accounts for 50% of the execution time. Compare the following three strategies with respect to programming time and execution time. Assume that it would take 100 man-months to write it in C, and that assembly code is 10 times slower to write and four times more efficient.

a. Entire program in C.
b. Entire program in assembler.
c. First all in C, then the key 2% rewritten in assembler.

2. Do the considerations that hold for two-pass assemblers also hold for compilers?

a. Assume that the compilers produce object modules, not assembly code.
b. Assume that the compilers produce symbolic assembly language.

3. Most assemblers for the x86 have the destination address as the first operand and the source address as the second operand. What problems would have to be solved to do it the other way?

4. Can the following program be assembled in two passes? EQU is a pseudoinstruction that equates the label to the expression in the operand field.

```
P EQU Q
Q EQU R
R EQU S
S EQU 4
```

5. The Dirtcheap Software Company is planning to produce an assembler for a computer with a 48-bit word. To keep costs down, the project manager, Dr. Scrooge, has decided to limit the length of allowed symbols so that each symbol can be stored in a single word. Scrooge has declared that symbols may consist only of letters, except the letter Q, which is forbidden (to demonstrate their concern for efficiency to the customers). What is the maximum length of a symbol? Describe your encoding scheme.

6. What is the difference between an instruction and a pseudoinstruction?

7. What is the difference between the instruction location counter and the program counter, if any? After all, both keep track of the next instruction in a program.

8. Show the symbol table after the following x86 statements have been encountered. The first statement is assigned to address 1000.

```
EVEREST:    POP BX          (1 BYTE)
K2:         PUSH BP         (1 BYTE)
WHITNEY:    MOV BP,SP       (2 BYTES)
MCKINLEY:   PUSH X          (3 BYTES)
FUJI:       PUSH SI         (1 BYTE)
KIBO:       SUB SI,300      (3 BYTES)
```

9. Can you envision circumstances in which an assembly language permits a label to be the same as an opcode (e.g., *MOV* as a label)? Discuss.

10. Show the steps needed to look up Ann Arbor using binary search on the following list: Ann Arbor, Berkeley, Cambridge, Eugene, Madison, New Haven, Palo Alto, Pasadena, Santa Cruz, Stony Brook, Westwood, and Yellow Springs. When computing the middle element of a list with an even number of elements, use the element just after the middle index.

11. Is it possible to use binary search on a table whose size is prime?

12. Compute the hash code for each of the following symbols by adding up the letters (A = 1, B = 2, etc.) and taking the result modulo the hash table size. The hash table has 19 slots, numbered 0 to 18.

els, jan, jelle, maaike

Does each symbol generate a unique hash value? If not, how can the collision problem be dealt with?

13. The hash coding method described in the text links all the entries having the same hash code together on a linked list. An alternative method is to have only a single *n*-slot table, with each table slot having room for one key and its value (or pointers to them). If the hashing algorithm generates a slot that is already full, a second hashing algorithm is used to try again. If that one is also full, another is used, and so on, until an empty is found. If the fraction of the slots that are full is *R*, how many probes will be needed, on the average, to enter a new symbol?

14. As technology progresses, it may one day be possible to put thousands of identical CPUs on a chip, each CPU having a few words of local memory. If all CPUs can read and write three shared registers, how can an associative memory be implemented?

15. The x86 has a segmented architecture, with multiple independent segments. An assembler for this machine might well have a pseudoinstruction `SEG N` that would direct the assembler to place subsequent code and data in segment *N*. Does this scheme have any influence on the ILC?

16. Programs often link to multiple DLLs. Would it not be more efficient just to put all the procedures in one big DLL and then link to it?

17. Can a DLL be mapped into two process' virtual address spaces at different virtual addresses? If so, what problems arise? Can they be solved? If not, what can be done to eliminate them?

18. One way to do (static) linking is as follows. Before scanning the library, the linker builds a list of procedures needed, that is, names defined as `EXTERN` in the modules being linked. Then the linker goes through the library linearly, extracting every procedure that is in the list of names needed. Does this scheme work? If not, why not and how can it be remedied?

19. Can a register be used as the actual parameter in a macro call? How about a constant? Why or why not?

20. You are to implement a macro assembler. For esthetic reasons, your boss has decided that macro definitions need not precede their calls. What implications does this decision have on the implementation?

21. Think of a way to put a macro assembler into an infinite loop.

22. A linker reads five modules, whose lengths are 200, 800, 600, 500, and 700 words, respectively. If they are loaded in that order, what are the relocation constants?

23. Write a symbol table package consisting of two routines: *enter(symbol, value)* and *lookup(symbol, value)*. The former enters new symbols in the table and the latter looks them up. Use some form of hash coding.

24. Repeat the previous problem, only this time instead of using a hash table, after the last symbol is entered, sort the table and use a binary-lookup algorithm to find symbols.

25. Write a simple assembler for the Mic-1 computer of Chap. 4. In addition to handling the machine instructions, provide a facility for assigning constants to symbols at assembly time, and a way to assemble a constant into a machine word. These should be pseudoinstructions, of course.

26. Add a simple macro facility to the assembler of the preceding problem.

8

PARALLEL COMPUTER ARCHITECTURES

Although computers keep getting faster, the demands placed on them are increasing at least as fast. Astronomers want to simulate the entire history of the universe, from the big bang until the show is over. Pharmaceutical engineers would love to be able to design medicines for specific diseases on their computer instead of having to sacrifice legions of rats. Aircraft designers could come up with more fuel-efficient products if computers could do all the work, without the need for constructing physical wind-tunnel prototypes. In short, for many users, however much computing power is available, especially in science, engineering, and industry, it is never enough.

Although clock rates are continually rising, circuit speed cannot be increased indefinitely. The speed of light is already a major problem for designers of high-end computers, and the prospects of getting electrons and photons to move faster are dim. Heat-dissipation issues are turning supercomputers into state-of-the-art air conditioners. Finally, as transistor sizes continue to shrink, at some point each transistor will have so few atoms in it that quantum-mechanical effects (e.g., the Heisenberg uncertainty principle) may become a major problem.

Therefore, in order to be able to handle larger and larger problems, computer architects are turning increasingly to parallel computers. While it may not be possible to build a computer with one CPU and a cycle time of 0.001 nsec, it may well be possible to build one with 1000 CPUs each with a cycle time of 1 nsec. Although the latter design uses slower CPUs, its total computing capacity is theoretically the same. Herein lies the hope.

Parallelism can be introduced at various levels. At the lowest level, it can be added to the CPU chip, through pipelining and superscalar designs with multiple functional units. It can also be added by having very long instruction words with implicit parallelism. Special features can be added to a CPU to allow it to handle multiple threads of control at once. Finally, multiple CPUs can be put together on the same chip. Together, these features can pick up perhaps a factor of 10 in performance over purely sequential designs.

At the next level, extra CPU boards with additional processing capacity can be added to a system. Usually, these plug-in CPUs have specialized functions, such as network packet processing, multimedia processing, or cryptography. For specialized applications, they can also gain a factor of perhaps 5 to 10.

However, to win a factor of a hundred or a thousand or a million, it is necessary to replicate entire CPUs and to make them all work together efficiently. This idea leads to large multiprocessors and multicomputers (cluster computers). Needless to say, hooking up thousands of processors into a big system leads to its own problems that need to be solved.

Finally, it is now possible to lash together entire organizations over the Internet to form very loosely coupled compute grids. These systems are only starting to emerge, but have interesting potential for the future.

When two CPUs or processing elements are close together, have a high bandwidth and low delay between them, and are computationally intimate, they are said to be **tightly coupled**. Conversely, when they are far apart, have a low bandwidth and high delay and are computationally remote, they are said to be **loosely coupled**. In this chapter we will look at the design principles for these various forms of parallelism and study a variety of examples. We will start with the most tightly coupled systems, those that use on-chip parallelism, and gradually move to more and more loosely coupled systems, ending with a few words on grid computing. This spectrum is crudely illustrated in Fig. 8-1.

The whole issue of parallelism, from one end of the spectrum to the other, is a hot topic of research. Accordingly, many references are given in this chapter, primarily to recent papers on the subject. Many conferences and journals publish papers on the subject as well and the literature is growing rapidly.

8.1 ON-CHIP PARALELLISM

One way to increase the throughput of a chip is to have it do more things at the same time. In other words, exploit parallelism. In this section, we will look at some of the ways of achieving speed-up through parallelism at the chip level, including instruction-level parallelism, multithreading, and putting more than one CPU on the chip. These techniques are quite different, but each helps in its own way. In all cases the idea is to get more activity going at the same time.

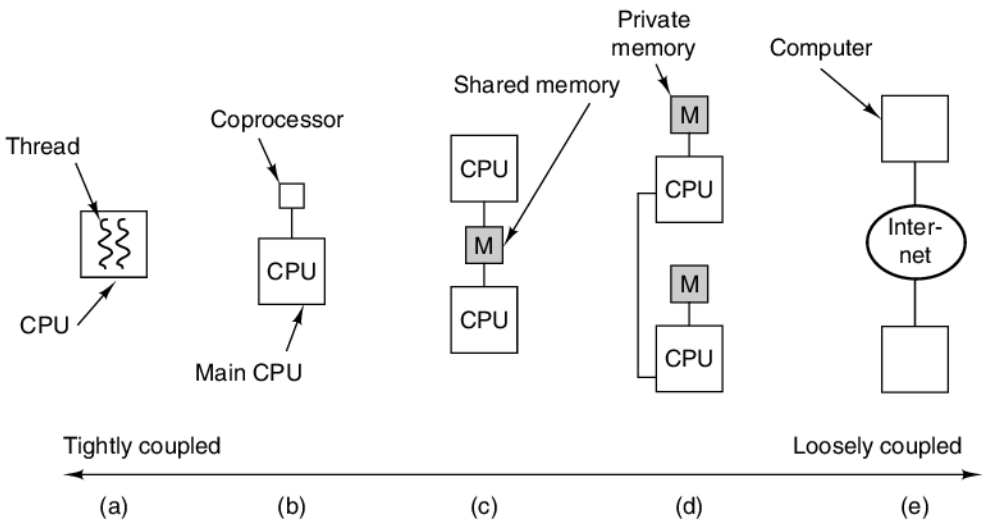

Figure 8-1. (a) On-chip parallelism. (b) A coprocessor. (c) A multiprocessor. (d) A multicomputer. (e) A grid.

8.1.1 Instruction-Level Parallelism

At the lowest level, one way to achieve parallelism is to issue multiple instructions per clock cycle. Multiple-issue CPUs come in two varieties: superscalar processors and VLIW processors. We have touched on both earlier in the book, but it may be useful to review this material here.

We have seen superscalar CPUs before (e.g., Fig. 2-5). In the most common configuration, at a certain point in the pipeline an instruction is ready to be executed. Superscalar CPUs are capable of issuing multiple instructions to the execution units in a single clock cycle. The number of instructions actually issued depends on both the processor design and current circumstances. The hardware determines the maximum number that can be issued, usually two to six instructions. However, if an instruction needs a functional unit that is not available or a result that has not yet been computed, the instruction will not be issued.

The other form of instruction-level parallelism is found in **VLIW** (**Very Long Instruction Word**) processors. In the original form, VLIW machines indeed had long words containing instructions that used multiple functional units. Consider, for example, the pipeline of Fig. 8-2(a), where the machine has five functional units and can perform two integer operations, one floating-point operation, one load, and one store simultaneously. A VLIW instruction for this machine would contain five opcodes and five pairs of operands, one opcode and operand pair per functional unit. With 6 bits per opcode, 5 bits per register, and 32 bits per memory address, instructions could easily be 134 bits—quite long indeed.

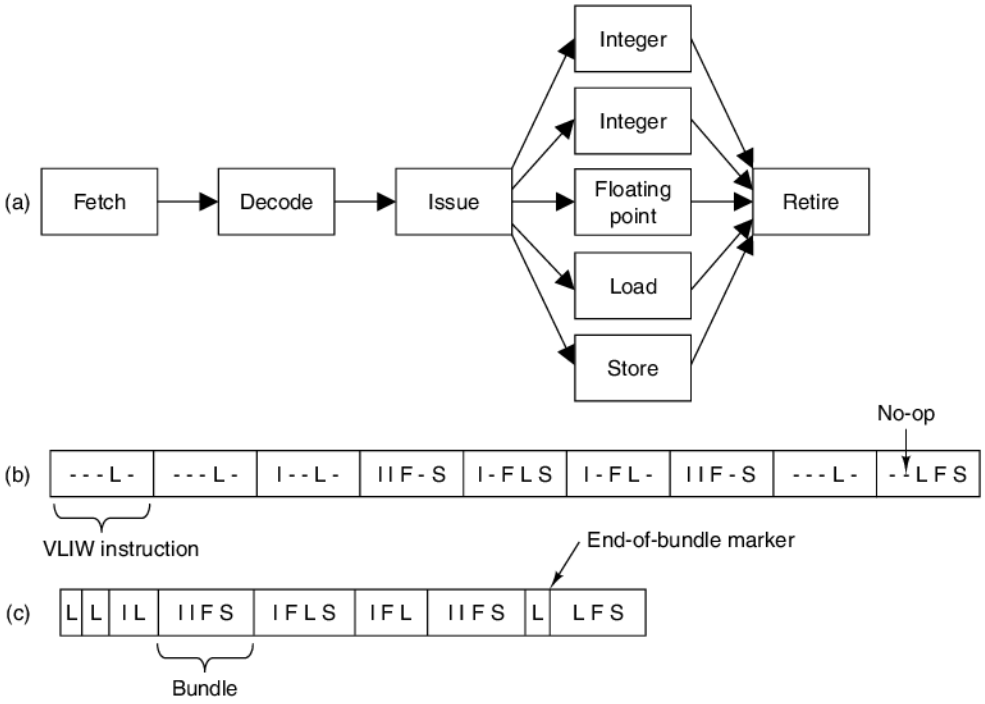

Figure 8-2. (a) A CPU pipeline. (b) A sequence of VLIW instructions. (c) An instruction stream with bundles marked.

However, this design proved too rigid because not every instruction was able to utilize every functional unit, leading to many useless NO-OPs used as filler, as illustrated in Fig. 8-2(b). Consequently, modern VLIW machines have a way of marking a bundle of instructions as belonging together, for example with an "end-of-bundle" bit, as shown in Fig. 8-2(c). The processor can then fetch the entire bundle and issue it all at once. It is up to the compiler to prepare bundles of compatible instructions.

In effect, VLIW shifts the burden of determining which instructions can be issued together from run time to compile time. Not only does this choice make the hardware simpler and faster, but since an optimizing compiler can run for a long time if need be, better bundles can be assembled than what the hardware could do at run time. Of course, such a radical change in CPU architecture will be difficult to introduce, as demonstrated by the slow acceptance of the Itanium except for niche applications.

It is worth noting in passing that instruction-level parallelism is not the only form of low-level parallelism. Another is memory-level parallelism, in which multiple memory operations are in flight at the same time (Chou et al., 2004).

The TriMedia VLIW CPU

We studied one example of a VLIW CPU, the Itanium-2, in Chap. 5. Let us now look at a very different VLIW processor, the **TriMedia**, designed by Philips, the Dutch electronics company that also invented the audio CD and CD-ROM. The TriMedia is intended for use as an embedded processor in image-, audio-, and video-intensive applications such as CD, DVD, and MP3 players, CD and DVD recorders, interactive TV sets, digital cameras, camcorders, and so on. Given these application areas, it is not surprising that it differs considerably from the Itanium-2, which is a general-purpose CPU intended for high-end servers.

The TriMedia is a true VLIW processor with every instruction holding as many as five **operations**. Under completely optimal conditions, every clock cycle one instruction is started and the five operations are issued. The clock runs at 266 MHz or 300 MHz, but since five operations per cycle can be issued, the effective clock speed is as much as five times higher. In the discussion below, we will focus on the TM3260 implementation of the TriMedia; other versions differ in minor ways from it.

A typical instruction is illustrated in Fig. 8-3. The instructions vary from standard 8-, 16-, and 32-bit integer instructions through IEEE 754 floating-point instructions to parallel multimedia instructions. As a consequence of the five issues per cycle and the parallel multimedia instructions, the TriMedia is fast enough to decode streaming DV from a camcorder at full size and full frame rate in software.

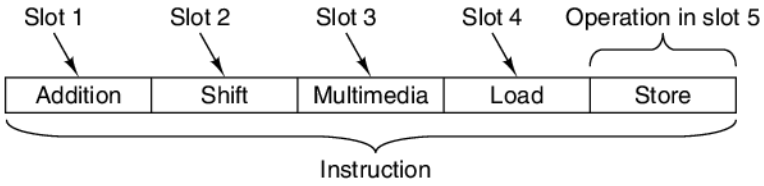

Figure 8-3. A typical TriMedia instruction, showing five possible operations.

The TriMedia has a byte-oriented memory, with the I/O registers mapped into the memory space. Half words (16 bits) and full words (32 bits) must be aligned on their natural boundaries. It can run either as little endian or big endian, depending on a PSW bit that the operating system can set. This bit affects only the way load operations and store operations transfer between memory and registers. The CPU contains a split 8-way set-associative cache, with a 64-byte line size for both the instruction cache and the data cache. The instruction cache is 64 KB; the data cache is 16 KB.

There are 128 general-purpose 32-bit registers. Register `R0` is hardwired to 0. Register `R1` is hardwired to 1. Attempting to change either one gives the CPU a heart attack. The remaining 126 registers are all functionally equivalent and can be used for any purpose. In addition, four special-purpose, 32-bit registers also exist.

These are the program counter, program status word, and two registers that relate to interrupts. Finally, a 64-bit register counts the number of CPU cycles since the CPU was last reset. At 300 MHz, it takes nearly 2000 years for the counter to wrap around.

The Trimedia TM3260 has 11 different functional units for doing arithmetic, logical, and control flow operations (as well as one for cache control that we will not discuss here). They are listed in Fig. 8-4. The first two columns name the unit and give a brief description of what it does. The third column tells how many hardware copies of the unit exist. The fourth column gives the latency, that is, how many clock cycles it takes to complete. In this context, it is worth nothing that all the functional units except the FP square-root/divide unit are pipelined. The latency given in the table tells how long before the result of an operation is available, but a new operation can be initiated every cycle. Thus, for example, each of three consecutive instructions can hold two load operations, resulting in six loads in various stages of execution at the same time.

Finally, the last five columns show which instruction slots can be used for which functional unit. For example, floating-point compare operations must appear only in the third slot of an instruction

Unit	Description	#	Lat.	1	2	3	4	5
Constant	Immediate operations	5	1	x	x	x	x	x
Integer ALU	32-Bit arithmetic, Boolean ops	5	1	x	x	x	x	x
Shifter	Multibit shifts	2	1	x	x	x	x	x
Load/Store	Memory operations	2	3				x	x
Int/FP MUL	32-Bit integer and FP multiplies	2	3		x	x		
FP ALU	FP arithmetic	2	3	x			x	
FP compare	FP compares	1	1			x		
FP sqrt/div	FP division and square root	1	17		x			
Branch	Control flow	3	3		x	x	x	
DSP ALU	Dual 16-bit, quad 8-bit multimedia arithmetic	2	3	x		x		x
DSP MUL	Dual 16-bit, quad 8-bit multimedia multiplies	2	3		x	x		

Figure 8-4. The TM3260 functional units, their quantity, latency, and which instruction slots they can use.

The constant unit is used for immediate operations, such as loading a number stored in the operation itself into a register. The integer ALU does addition, subtraction, the usual Boolean operations, and pack/unpack operations. The shifter can shift a register in either direction a specified number of bits.

The load/store unit fetches memory words into registers and writes them back. The TriMedia is basically an augmented RISC CPU, so normal operations operate on registers and the load/store unit is used to access memory. Transfers can be 8, 16, or 32 bits. Arithmetic and logical instructions do not access memory.

The multiply unit handles both integer and floating-point multiplications. The next three units handle floating-point additions/subtractions, compares, and square roots and divisions, respectively.

Branch operations are executed by the branch unit. There is a fixed 3-cycle delay after a branch, so the three instructions (up to 15 operations) following a branch are always executed, even for unconditional branches.

Finally, we come to the two multimedia units, which handle the special multimedia operations. The DSP in the name of the functional unit refers to **Digital Signal Processor**. which the multimedia operations effectively replace. We will describe the multimedia operations briefly below. One noteworthy feature is that they all use **saturated arithmetic** instead of two's complement arithmetic used by the integer operations. When an operation produces a result that cannot be expressed due to overflow, instead of generating an exception or giving a garbage result, the closest valid number is used. For example, with 8-bit unsigned numbers, adding 130 and 130 gives 255.

Because not every operation can appear in every slot, often an instruction does not contain all five potential operations. When a slot is not used, it is compacted to minimize the amount of space wasted. Operations that are present occupy 26, 34, or 42 bits. Depending on the number of operations actually present, TriMedia instructions vary from 2 to 28 bytes, including some fixed overhead.

The TriMedia does not make run-time checks to see whether the operations in an instruction are compatible. If they are not, it just executes them anyway and gets the wrong answer. Leaving the check out was a deliberate decision to save time and transistors. The Core i7 does do run-time checking to make sure all the superscalar operations are compatible, but at a huge cost in complexity, time, and transistors. The TriMedia avoids this expense by putting the burden of scheduling on the compiler, which has all the time in the world to carefully optimize the placement of operations in instruction words. On the other hand, if an operation needs a functional unit that is not available, the instruction will stall until it becomes available.

As in the Itanium-2, TriMedia operations are predicated. Each operation (with two minor exceptions) specifies a register that is tested before the operation is executed. If the low-order bit of the register is set, the operation is executed; otherwise, it is skipped. Each of the (up to) five operations is individually predicated. An example of a predicated operation is

```
IF R2 IADD R4, R5 –> R8
```

which tests R2 and, if the low-order bit is 1, adds R4 to R5 and stores the result in R8. An operation can be made unconditional by using R1 (which is always 1) as the predicate register. Using R0 (which is always 0) makes it a no-op.

The TriMedia multimedia operations can be grouped into the 15 groups listed in Fig. 8-5. Many of the operations involve clipping, which specifies an operand

and a range and forces the operand into the range, using the lowest or highest values for operands that fall outside it. Clipping can be done on 8-, 16-, or 32-bit operands. For example, when clipping is performed with a range of 0 to 255 on 40 and 340, the clipped results are 40 and 255, respectively. The clip group performs clip operations.

Group	Description
Clip	Clip 4 bytes or 2 half words
DSP absolute value	Clipped, signed, absolute value
DSP add	Clipped signed addition
DSP subtract	Clipped signed subtraction
DSP multiply	Clipped signed multiplication
Min, max	Get minimum or maximum of four byte pairs
Compare	Bytewise compare of two registers
Shift	Shift a pair of 16-bit operands
Sum of products	Signed sum of 8- or 16-bit products
Merge, pack, swap	Byte and half word manipulation
Byte quad averages	Unsigned byte-wise quad averaging
Byte averages	Unsigned byte-wise average of four elements
Byte multiplies	Unsigned 8-bit multiply
Motion estimation	Unsigned sum of absolute values of signed 8-bit diffs
Miscellaneous	Other arithmetic operations

Figure 8-5. The major groups of TriMedia custom operations.

The next four groups in Fig. 8-5 perform the indicated operation on operands of various sizes, clipping the results into a specific range. The min, max group examines two registers and for each byte finds the smallest or largest value. Similarly, the compare group regards two registers as four pairs of bytes and compares each pair.

Multimedia operations are rarely performed on 32-bit integers because most images are composed of RGB pixels with 8-bit values for each of the red, green, and blue colors. When an image is being processed (e.g., compressed), it is normally represented by three components, one for each color (RGB space) or a logically equivalent form (YUV space, discussed later in this chapter). Either way, a lot of processing is done on rectangular arrays containing 8-bit unsigned integers.

The TriMedia has a large number of operations specifically designed for processing arrays of 8-bit unsigned integers efficiently. As a simple example, consider the upper left-hand corner of an array of 8-bit values stored in (big-endian) memory as illustrated in Fig. 8-6(a). The 4×4 block shown in the corner contains 16 8-bit values labeled *A* through *P*. Suppose, for example, that the image needs to be transposed, to produce Fig. 8-6(b). How can this task be achieved?

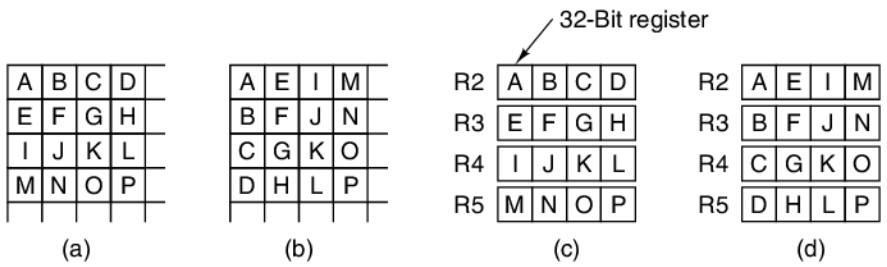

Figure 8-6. (a) An array of 8-bit elements. (b) The transposed array. (c) The original array fetched into four registers. (d) The transposed array in four registers.

One way to do the transposition is to use 12 operations that each load a byte into a different register, followed by 12 more operations that each store a byte in its correct location. (*Note*: the four bytes on the diagonal do not move in the transposition.) The problem with this approach is that it requires 24 (long and slow) operations that reference memory.

An alternative approach is to start with four operations that each load a word into four different registers, R2 through R5, as shown in Fig. 8-6(c). Then the four output words are assembled by masking and shifting operations to achieve the desired output, as shown in Fig. 8-6(d). Finally, these words are stored in memory. Although this way of doing it reduces the number of memory references from 24 to 8, the masking and shifting is expensive due to the many operations required to extract and insert each byte in the correct position.

The TriMedia provides a better solution than either of these. It begins by fetching the four words into registers. However, instead of building the output using masking and shifting, special operations that extract and insert bytes within registers are used to build the output. The result is that with eight memory references and eight of these special multimedia operations, the transposition can be accomplished. The code first contains an operation with two load operations in slots 4 and 5, respectively, to load words into R2 and R3, followed by another such operation to load R4 and R5.

The instructions holding these operations can use slots 1, 2, and 3 for other purposes. When all the words have been loaded, the eight special multimedia operations can be packed into two instructions to build the output, followed by two operations to store them. Only six instructions are needed, and 14 of the 30 slots in these instructions are available for other operations. In effect, the entire job can be done with the effective equivalent of about three or so instructions. Other multimedia operations are also efficient. Between these powerful operations and the five issue slots per instruction, the TriMedia is highly efficient at doing the kinds of calculations needed in multimedia processing.

8.1.2 On-Chip Multithreading

All modern, pipelined CPUs have an inherent problem: when a memory reference misses the level 1 and level 2 caches, there is a long wait until the requested word (and its associated cache line) are loaded into the cache, so the pipeline stalls. One approach to dealing with this situation, called **on-chip multithreading**, allows the CPU to manage multiple threads of control at the same time in an attempt to mask these stalls. In short, if thread 1 is blocked, the CPU still has a chance of running thread 2 in order to keep the hardware fully occupied.

Although the basic idea is fairly simple, multiple variants exist, which we will now examine. The first approach, called **fine-grained multithreading**, is illustrated in Fig. 8-7 for a CPU with the ability to issue one instruction per clock cycle. In Fig. 8-7(a)–(c), we see three threads, *A*, *B*, and *C*, for 12 machine cycles. During the first cycle, thread *A* executes instruction *A1*. This instruction completes in one cycle, so in the second cycle instruction *A2* is started. Unfortunately, this instruction misses on the level 1 cache so two cycles are wasted while the word needed is fetched from the level 2 cache. The thread continues in cycle 5. Similarly, threads *B* and *C* also stall occasionally as well, as illustrated in the figure. In this model if an instruction stalls, subsequent instructions cannot be issued. Of course, with a more sophisticated scoreboard, sometimes new instructions can still be issued, but we will ignore that possibility in this discussion.

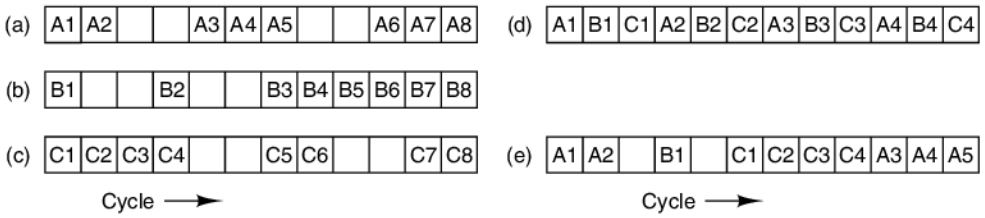

Figure 8-7. (a)–(c) Three threads. The empty boxes indicate that the thread has stalled waiting for memory. (d) Fine-grained multithreading. (e) Coarse-grained multithreading.

Fine-grained multithreading masks the stalls by running the threads round robin, with a different thread in consecutive cycles, as shown in Fig. 8-7(d). By the time the fourth cycle comes up, the memory operation initiated in *A1* has completed, so instruction *A2* can be run, even if it needs the result of *A1*. In this case the maximum stall is two cycles, so with three threads the stalled operation always completes in time. If a memory stall took four cycles, we would need four threads to insure continuous operation, and so on.

Since different threads have nothing to do with one another, each one needs its own set of registers. When an instruction is issued, a pointer to its register set has to be included along with the instruction so that if a register is referenced, the

hardware will know which register set to use. Consequently, the maximum number of threads that can be run at once is fixed at the time the chip is designed.

Memory operations are not the only reason for stalling. Sometimes an instruction needs a result computed by a previous instruction that is not yet complete. Sometimes an instruction cannot start because it follows a conditional branch whose direction is not yet known. In general, if the pipeline has k stages but there are at least k threads to run round robin, there will never be more than one instruction per thread in the pipeline at any moment, so no conflicts can occur. In this situation, the CPU can run at full speed, never stalling.

Of course, there may not be as many threads available as there are pipeline stages, so some designers prefer a different approach, known as **coarse-grained multithreading**, illustrated in Fig. 8-7(e). Here thread *A* starts and continues to issue instructions until it stalls, wasting one cycle. At that point a switch occurs and *B1* is executed. Since the first instruction of thread *B* stalls, another thread switch happens and *C1* is executed in cycle 6. Since a cycle is lost whenever an instruction stalls, coarse-grained multithreading is potentially less efficient than fine-grained multithreading, but it has the big advantage that many fewer threads are needed to keep the CPU busy. In situations with an insufficient number of active threads, to be sure of finding a runnable one, coarse-grained multithreading works better.

Although we have depicted coarse-grained multithreading as doing thread switches on a stall, that is not the only option. Another possibility is to switch immediately on any instruction that might cause a stall, such as a load, store, or branch, before even finding out whether it does cause a stall. This allows a switch to occur earlier (as soon as the instruction is decoded), and may make it possible to avoid dead cycles. In effect, it is saying: "Run until there might be a problem, then switch just in case." Doing so makes coarse-grained multithreading somewhat more like fine-grained multithreading with its frequent switches.

No matter which kind of multithreading is used, some way is needed to keep track of which operation belongs to which thread. With fine-grained multithreading, the only serious possibility is to attach a thread identifier to each operation, so that as it moves through the pipeline, its identity is clear. With coarse-grained multithreading, another possibility exists: when switching threads, let the pipeline clear and only then start the next thread. In that way, only one thread at a time is in the pipeline and its identity is never in doubt. Of course, letting the pipeline run dry on a thread switch makes sense only if thread switches take place at intervals very much longer than the time it takes to empty the pipeline.

So far we have assumed that the CPU can issue only one instruction per cycle. As we have seen, however, modern CPUs can issue multiple instructions per cycle. In Fig. 8-8 we assume the CPU can issue two instructions per clock cycle, but we maintain the rule that when an instruction stalls, subsequent instructions cannot be issued. In Fig. 8-8(a) we see how fine-grained multithreading works with a dual-issue superscalar CPU. For thread *A*, the first two instructions can be issued in the

first cycle, but for thread *B* we immediately hit a problem in the next cycle, so only one instruction can be issued, and so on.

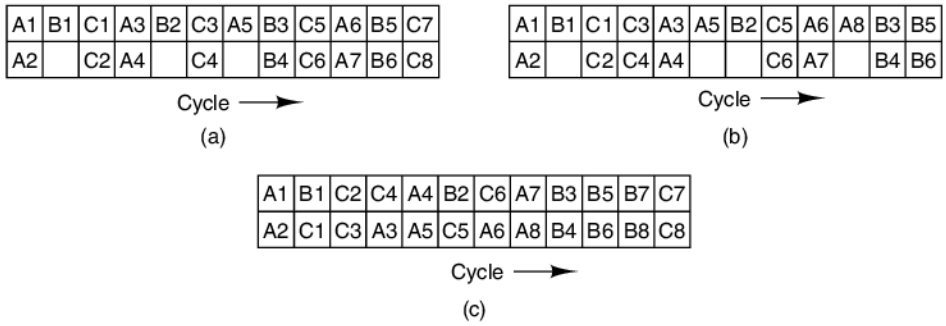

Figure 8-8. Multithreading with a dual-issue superscalar CPU. (a) Fine-grained multithreading. (b) Coarse-grained multithreading. (c) Simultaneous multithreading.

In Fig. 8-8(b), we see how coarse-grained multithreading works with a dual-issue CPU, but now with a static scheduler that does not introduce a dead cycle after an instruction that stalls. Basically, the threads are run in turn, with the CPU issuing two instructions per thread until it hits one that stalls, at which point it switches to the next thread at the start of the next cycle.

With superscalar CPUs, a third possible way of doing multithreading is available, called **simultaneous multithreading** and illustrated in Fig. 8-8(c). This approach can be seen as a refinement to coarse-grained multithreading, in which a single thread is allowed to issue two instructions per cycle as long as it can, but when it stalls, instructions are immediately taken from the next thread in sequence to keep the CPU fully occupied. Simultaneous multithreading can also help keep all the functional units busy. When an instruction cannot be started because a functional unit it needs is occupied, an instruction from a different thread can be chosen instead. In this figure, we are assuming that *B8* stalls in cycle 11, so *C7* is started in cycle 12.

For more information about multithreading, see Gebhart et al. (2011) and Wing-kei et al. (2011).

Hyperthreading on the Core i7

Having looked at multithreading in the abstract, let us now consider a practical example: the Core i7. In the early 2000s, processors such as the Pentium 4 were not delivering the performance boosts that Intel needed to keep up sales. After the Pentium 4 was already in production, the architects at Intel looked for various ways to speed it up without changing the programmers' interface, something that would never have been accepted. Five ways quickly popped up:

1. Increasing the clock speed.
2. Putting two CPUs on a chip.
3. Adding functional units.
4. Making the pipeline longer.
5. Using multithreading.

An obvious way to improve performance is to increase the clock speed without changing anything else. Doing this is relatively straightforward and well understood, so each new chip that comes out is generally slightly faster than its predecessor. Unfortunately, a faster clock also has two main drawbacks that limit how great an increase can be tolerated. First, a faster clock uses more energy, which is a huge problem for notebook computers and other battery-powered devices. Second, the extra energy input means the chip gets hotter and there is more heat to dissipate.

Putting two CPUs on a chip is relatively straightforward, but it comes close to doubling the chip area if each one has its own caches and thus reduces the number of chips per wafer by a factor of two, which essentially doubles the unit manufacturing cost. If the two chips share a common cache as big as the original one, the chip area is not doubled, but cache size per CPU is halved, cutting into performance. Also, while high-end server applications can often fully utilize multiple CPUs, not all desktop applications have enough inherent parallelism to warrant two full CPUs.

Adding additional functional units is also fairly easy, but it is important to get the balance right. Having 10 ALUs does little good if the chip is incapable of feeding instructions into the pipeline fast enough to keep them all busy.

A longer pipeline with more stages, each doing a smaller piece of work in a shorter time period increases performance but also increases the negative effects of branch mispredictions, cache misses, interrupts, and other factors that interrupt normal pipeline flow. Furthermore, to take full advantage of a longer pipeline, the clock speed has to be increased, which means more energy is consumed and more heat is produced.

Finally, multithreading can be added. Its value is in having a second thread utilize hardware that would otherwise have lain fallow. After some experimentation, it became clear that a 5% increase in chip area for multithreading support gave a 25% performance gain in many applications, making this a good choice. Intel's first multithreaded CPU was the Xeon in 2002, but multithreading was later added to the Pentium 4, starting with the 3.06-GHz version and continuing with faster versions of the Pentium processor, including the Core i7. Intel calls the implementation of multithreading used in its processors **hyperthreading**.

The basic idea is to allow two threads (or possibly processes, since the CPU cannot tell what is a thread and what is a process) to run at once. To the operating

system, a hyperthreaded Core i7 chip looks like a dual processor in which both CPUs happen to share a common cache and main memory. The operating system schedules the threads independently. If two applications are running at the same time, the operating system can run each one at the same time. For example, if a mail daemon is sending or receiving email in the background while a user is interacting with some program in the foreground, the daemon and the user program can be run in parallel, as though there were two CPUs available.

Application software that has been designed to run as multiple threads can use both virtual CPUs. For example, video editing programs usually allow users to specify certain filters to apply to each frame in some range. These filters can modify the brightness, contrast, color balance, or other properties of each frame. The program can then assign one CPU to process the even-numbered frames and the other to process the odd-numbered frames. The two can then run in parallel.

Since the two threads share all the hardware resources, a strategy is needed to manage the sharing. Intel identified four useful strategies for resource sharing in conjunction with hyperthreading: resource duplication, partitioned resources, threshold sharing, and full sharing. We will now touch on each of these in turn.

To start with, some resources are duplicated just for threading. For example, since each thread has its own flow of control, a second program counter had to be added. The table that maps the architectural registers (`EAX`, `EBX`, etc.) onto the physical registers also had to be duplicated, as did the interrupt controller, since the threads can be independently interrupted.

Next we have **partitioned resource sharing**, in which the hardware resources are rigidly divided between the threads. For example, if the CPU has a queue between two functional pipeline stages, half the slots could be dedicated to thread 1 and the other half to thread 2. Partitioning is easy to accomplish, has no overhead, and keeps the threads out of each other's hair. If all the resources are partitioned, we effectively have two separate CPUs. On the down side, it can easily happen that at some point one thread is not using some of its resources that the other one wants but is forbidden from accessing. As a consequence, resources that could have been used productively lie idle.

The opposite of partitioned sharing is **full resource sharing**. When this scheme is used, either thread can acquire any resources it needs, first come, first served. However, imagine a fast thread consisting primarily of additions and subtractions and a slow thread consisting primarily of multiplications and divisions. If instructions are fetched from memory faster than multiplications and divisions can be carried out, the backlog of instructions fetched for the slow thread and queued but not yet fed into the pipeline will grow in time.

Eventually, this backlog will occupy the entire instruction queue, bringing the fast thread to a halt for lack of space in the instruction queue. Full resource sharing solves the problem of a resource lying idle while another thread wants it, but creates a new problem of one thread potentially hogging so many resources that it slows the other one down or stops it altogether.

An intermediate scheme is **threshold sharing**, in which a thread can acquire resources dynamically (no fixed partitioning) but only up to some maximum. For resources that are replicated, this approach allows flexibility without the danger that one thread will starve due to its inability to acquire any of the resource. If, for example, no thread can acquire more than 3/4 of the instruction queue, no matter what the slow thread does, the fast thread will be able to run. The Core i7 hyperthreading uses different sharing strategies for different resources in an attempt to address the various problems alluded to above. Duplication is used for resources that each thread requires all the time, such as the program counter, register map, and interrupt controller. Duplicating these resources increases the chip area by only 5%, a modest price to pay for multithreading. Resources available in such abundance that there is no danger of one thread capturing them all, such as cache lines, are fully shared in a dynamic way. On the other hand, resources that control the operation of the pipeline, such as the various queues within the pipeline, are partitioned, giving each thread half of the slots. The main pipeline of the Sandy Bridge microarchitecture used in the Core i7 is illustrated in Fig. 8-9, with the white and gray boxes indicating how the resources are allocated between the white and gray threads.

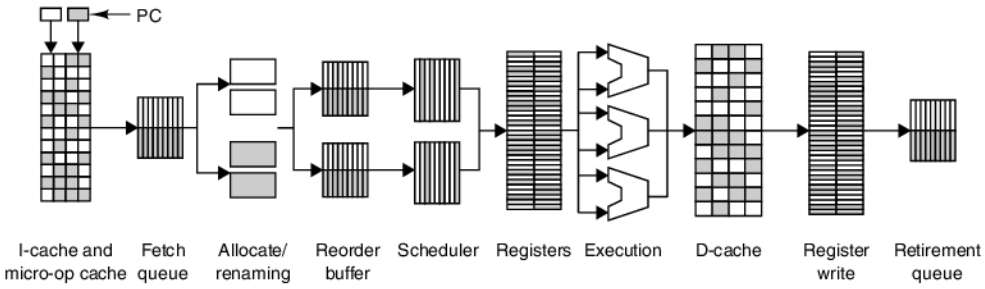

Figure 8-9. Resource sharing between threads in the Core i7 microarchitecture.

In this figure we can see that all the queues are partitioned, with half the slots in each queue reserved for each thread. In this one, neither thread can choke off the other. The register allocator and renamer is also partitioned. The scheduler is dynamically shared, but with a threshold, to prevent either thread from claiming all of the slots. The remaining pipeline stages are fully shared.

All is not sweetness and light with multithreading, however. There is also a downside. While partitioning is cheap, dynamic sharing of any resource, especially with a limit on how much a thread can take, requires bookkeeping at run time to monitor usage. In addition, situations can arise in which programs work much worse with multithreading than without it. For example, imagine two threads that each need 3/4 of the cache to function well. Run separately, each one works fine and has few (expensive) cache misses. Run together, each one has numerous cache misses and the net result may be far worse than without multithreading.

More information about multithreading and its implementation inside Intel processors is given in Gerber and Binstock (2004) and Gepner et al. (2011).

8.1.3 Single-Chip Multiprocessors

While multithreading provides significant performance gains at modest cost, for some applications a much larger performance gain is needed. To get this gain, multiprocessor chips are being developed. Two areas where these chips, which contain two or more CPUs, are of interest are high-end servers and in consumer electronics. We will now briefly touch on each of them.

Homogeneous Multiprocessors on a Chip

With advances in VLSI technology, it is now possible to put two or more powerful CPUs on a single chip. Since these CPUs often share the same level 2 cache and main memory, they qualify as a multiprocessor, as discussed in Chap. 2. A typical application area is a large Web server farm consisting of many servers. By putting two CPUs in the same box, sharing not only memory but also disks and network interfaces, the performance of the server can often be doubled without doubling the cost (because even at twice the price, the CPU chip is only a fraction of the total system cost).

For small-scale single-chip multiprocessors, two designs are prevalent. In the first one, shown in Fig. 8-10(a), there is really only one chip, but it has a second pipeline, potentially doubling the instruction execution rate. In the second one, shown in Fig. 8-10(b), there are separate cores on the chip, each containing a full CPU. A **core** is a large circuit, such as a CPU, I/O controller, or cache, that can be placed on a chip in a modular way, usually next to other cores.

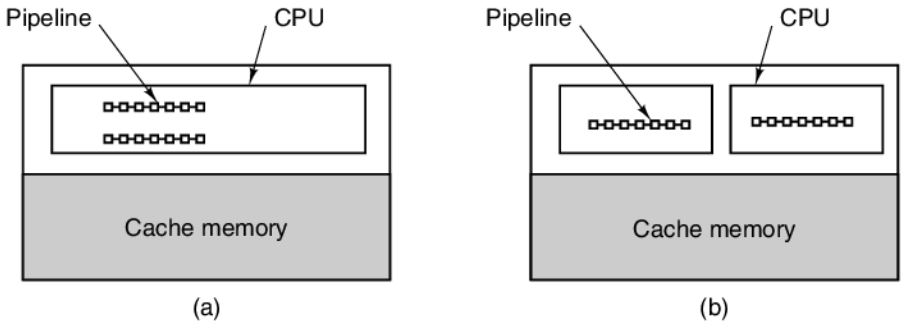

Figure 8-10. Single-chip multiprocessors. (a) A dual-pipeline chip. (b) A chip with two cores.

The former design allows resources, such as functional units, to be shared between the processors, thus allowing one CPU to use resources the other does not

need. On the other hand, this approach requires redesigning the chip and it does not scale well much above two CPUs. In contrast, just putting two or more CPU cores on the same chip is relatively easy to do.

We will discuss multiprocessors later in the chapter. While that discussion is somewhat focused on multiprocessors built from single-CPU chips, much of it applies to multi-CPU chips as well.

The Core i7 Single-Chip Multiprocessor

The Core i7 CPU is a single-chip multiprocessor that is manufactured with four or more cores on a single silicon die. The high-level organization of a Core i7 processor is illustrated in Fig. 8-11.

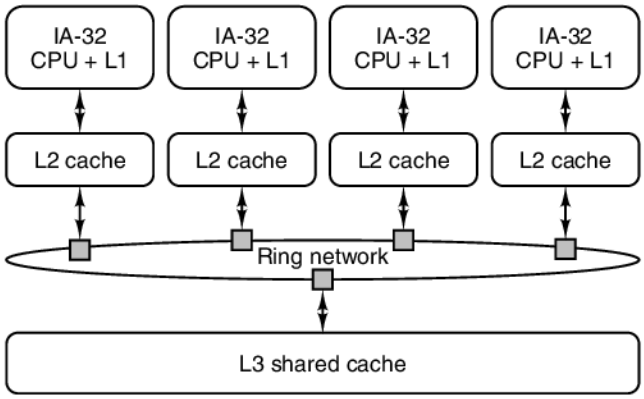

Figure 8-11. Single-chip multiprocessor architecture of the Core i7.

Each processor in the Core i7 has its own private L1 instruction and data caches, plus its own private L2 unified cache. The processors are connected to the private caches with dedicated point-to-point connections. The next level of the memory hierarchy is the shared and unified L3 data cache.

The L2 caches connect to the L3 shared cache using a **ring network**. When a communication request enters the ring network, it is forwarded to the next node on the network, where it is checked to see if it has reached its destination node. This process continues from node to node on the ring until the destination node is found or the request arrives at its source again (in which case the destination does not exist). The advantage of a ring network is it is a cheap way to get high bandwidth, at the cost of increased latency as requests hop from node to node. The Core i7 ring network serves two primary purposes. First, it provides a way to move memory and I/O requests between the caches and processors. Second, it implements the checks necessary to ensure that each processor is always seeing a coherent view of memory. We will learn more about these coherence checks later in this chapter.

Heterogeneous Multiprocessors on a Chip

A completely different application area calling for single-chip multiprocessors is embedded systems, especially in audiovisual consumer electronics, such as television sets, DVD players, camcorders, game consoles, cell phones, and so on. These systems have demanding performance requirements and tight constraints. Although these devices look different, more and more of them are simply small computers, with one or more CPUs, memories, I/O controllers, and some custom I/O devices. A cell phone, for example, is merely a PC with a CPU, memory, dwarf keyboard, microphone, loudspeaker, and a wireless network connection in a small package.

Consider, as an example, a portable DVD player. The computer inside has to handle the following functions:

1. Control of a cheap, unreliable servomechanism for head tracking.
2. Analog-to-digital conversion.
3. Error correction.
4. Decryption and digital rights management.
5. MPEG-2 video decompression.
6. Audio decompression.
7. Encoding the output for NTSC, PAL, or SECAM television sets.

This work must be done subject to stringent real-time, quality-of-service, energy, heat-dissipation, size, weight, and price constraints.

CD, DVD, and Blu-ray disks contain a long spiral containing the information, as illustrated in Fig. 2-25 (for a CD). In this section we will discuss DVDs since they are still more common than Blu-ray disks, but Blu-ray disks are very similar to DVDs except they use MPEG-4 instead of MPEG-2 for encoding. With all optical media, the read head must accurately track the spiral as the disk rotates. The price is kept low by using a relatively simple mechanical design and tight control over the head position in software. The signal coming off the head is an analog signal, which must be converted to digital form before being processed. After it has been digitized, heavy error correction is required because DVDs are pressed and contain many errors, which must corrected in software. The video is compressed using the MPEG-2 international standard, which requires complex (Fourier-transform-like) computations for decompression. Audio is compressed using a psycho-acoustic model, which also requires sophisticated calculations for decompression. Finally, audio and video have to be rendered in a suitable form for output to NTSC, PAL, or SECAM television sets, depending on the country to which the DVD player is shipped. It should come as no surprise that doing all this work in real time in software on a cheap general-purpose CPU is not possible.

What is needed is a heterogeneous multiprocessor containing multiple cores, each specialized for one particular task. An example DVD player is given in Fig. 8-12.

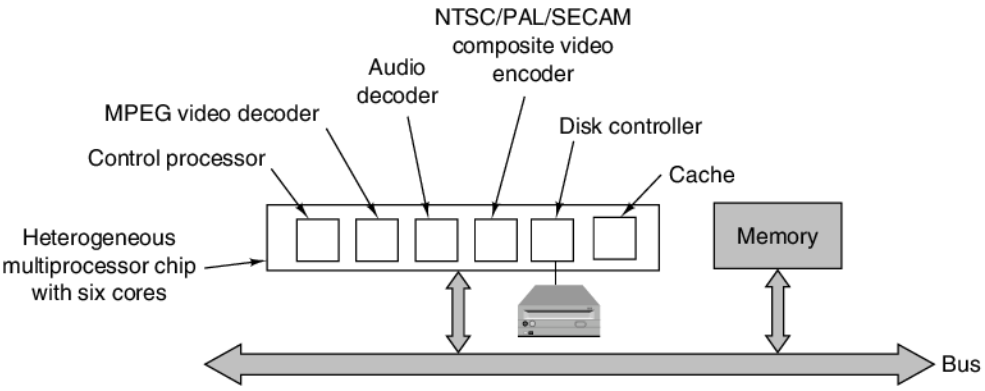

Figure 8-12. The logical structure of a simple DVD player contains a heterogeneous multiprocessor containing multiple cores for different functions.

The functions of the cores in Fig. 8-12 are all different, with each being carefully designed to be extremely good at what it does for the lowest possible price. For example, DVD video is compressed using a scheme known as **MPEG-2** (after the **Motion Picture Experts Group** that invented it). It consists of dividing each frame up into blocks of pixels and doing a complex transformation on each one. A frame can consist entirely of transformed blocks or it can specify that a certain block is the same as one found in the previous frame but located at an offset of $(\Delta x, \Delta y)$ from its current position except with a couple of pixels changed. Doing this calculation in software is extremely slow, but it is possible to build an MPEG-2 decoding engine that can do it in hardware quite rapidly. Similarly, audio decoding and reencoding the composite audio-video signal to conform to one of the world's television standards can be done better by dedicated hardware processors. These observations quickly lead to heterogeneous multiprocessor chips containing multiple cores specifically designed for audio-visual applications. However, because the control processor is a general-purpose programmable CPU, the multiprocessor chip can also be used in other, similar applications, such as a DVD recorder.

Another device requiring a heterogeneous multiprocessor is the engine inside an advanced cell phone. Current phones sometimes have still cameras, video cameras, game machines, web browsers, email readers, and digital satellite radio receivers, using either cell-phone technology (CDMA or GSM, depending on the country) or wireless Internet (IEEE 802.11, also called WiFi) built in; future ones may include all of these. As devices take on more and more functionality, with watches becoming GPS-based maps and eyeglasses becoming radios, the need for heterogeneous multiprocessors will only increase.

Fairly soon, large chips will have tens of billions of transistors. Such chips are far too large to design one gate and one wire at a time. The human effort required would render the chips obsolete by the time they were finished. The only feasible approach is to use cores (essentially libraries) containing fairly large subassemblies and to place and interconnect them on the chip as needed. Designers then have to determine which CPU core to use for the control processor and which special-purpose processors to throw in to help it. Putting more of the burden on software running on the control processor makes the system slower but yields a smaller (and cheaper) chip. Having multiple special-purpose processors for audio and video processing takes up chip area, increasing the cost, but produces higher performance at a lower clock rate, which means lower power consumption and less heat dissipation. Thus chip designers increasingly contend with these macroscopic trade-offs rather than worrying about where to place each transistor.

Audiovisual applications are very data intensive. Huge amounts of data have to be processed quickly, so typically 50% to 75% of the chip area is devoted to memory in one form or another, and the amount is rising. The design issues here are numerous. How many levels of cache should be used? Should the cache(s) be split or unified? How big should each cache be? How fast should each be? Should some actual memory go on the chip, too? Should it be SRAM or SDRAM? The answers to each of these questions have major implications for the performance, energy consumption, and heat dissipation of the chip.

Besides design of the processors and memory system, another issue of considerable consequence is the communication system—how do all the cores communicate with each other? For small systems, a single bus will usually do the trick, but for larger ones it rapidly becomes a bottleneck. Often the problem can be solved by going to multiple buses or possibly a ring from core to core. In the latter case, arbitration is handled by passing a small packet called a **token** around the ring. To transmit, a core must first capture the token. When it is done, it puts the token back on the ring so it can continue circulating. This protocol prevents collisions on the ring.

As an example of an on-chip interconnect, look at the IBM **CoreConnect**, illustrated in Fig. 8-13. It is an architecture for connecting cores on a single-chip heterogeneous multiprocessor, especially complete system-on-a-chip designs. In a sense, CoreConnect is to one-chip multiprocessors what the PCI bus was to the Pentium—the glue that holds all the parts together. (With modern Core i7 systems, PCIe is the glue, but it is a point-to-point network without a shared bus like PCI.) However, unlike the PCI bus, CoreConnect was designed without any requirements to be backward compatible with legacy equipment or protocols and without the constraints of board-level buses, such as limits on the number of pins the edge connector can have.

CoreConnect consists of three buses. The **processor bus** is a high-speed, synchronous, pipelined bus with 32, 64, or 128 data lines clocked at 66, 133, or 183 MHz. The maximum throughput is thus 23.4 Gbps (vs. 4.2 Gbps for the PCI bus).

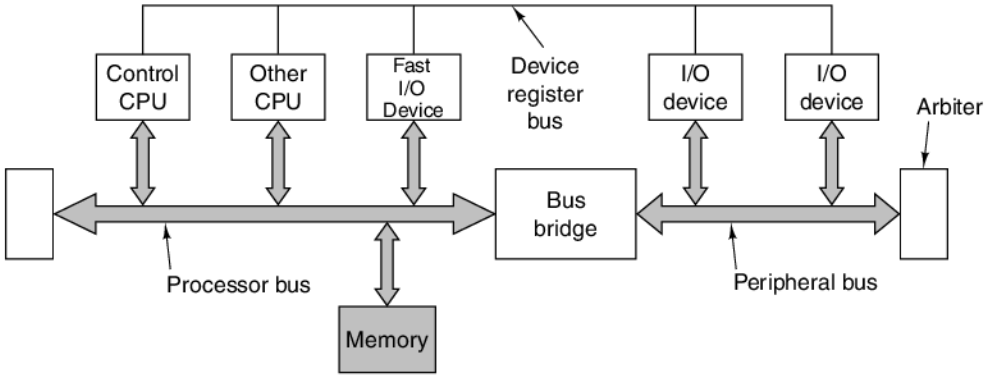

Figure 8-13. An example of the IBM CoreConnect architecture.

The pipelining features allow cores to request the bus while a transfer is going on, and allow different cores to use different lines at the same time, similar to the PCI bus. The processor bus is optimized for short block transfers. It is intended to connect fast cores, such as CPUs, MPEG-2 decoders, high-speed networks, and similar items.

Stretching the processor bus over the entire chip would reduce its performance, so a second bus is present for low-speed I/O devices, such as UARTs, timers, USB controllers, serial I/O devices, and so forth. This **peripheral bus** has been designed to simplify interfacing 8-, 16-, and 32-bit peripherals to it using no more than a few hundred gates. It, too, is a synchronous bus, with a maximum throughput of 300 Mbps. The two buses are connected by a bridge, not unlike the bridges that were used to connect the PCI and ISA buses in PCs until the ISA bus was phased out a number of years ago.

The third bus is the **device register bus**, a very low-speed, asynchronous, handshaking bus used to allow the processors to access the device registers of all the peripherals in order to control the corresponding devices. It is intended for infrequent transfers of only a few bytes at a time.

By providing a standard on-chip bus, interface, and framework, IBM hopes to create a miniature version of the PCI world, in which many manufacturers produce processors and controllers that plug together easily. One difference, however, is that in the PCI world the manufacturers produce and sell actual boards that PC vendors and end users buy. In the CoreConnect world, third parties design cores but do not manufacture them. Instead, they license them as intellectual property to consumer electronics and other companies, which then design custom heterogeneous multiprocessor chips based on their own and licensed third-party cores. Since manufacturing such large and complex chips requires a massive investment

in fabrication facilities, in most cases the consumer electronics company just does the design, subcontracting the chip manufacturing out to a semiconductor vendor. Cores exist for numerous CPUs (ARM, MIPS, PowerPC, etc.) as well as for MPEG decoders, digital signal processors, and all the standard I/O controllers.

The IBM CoreConnect is not the only popular on-chip bus on the market. The **AMBA** (**Advanced Microcontroller Bus Architecture**) is also widely used to connect ARM CPUs to other CPUs and I/O devices (Flynn, 1997). Other, somewhat less popular on-chip buses are the **VCI** (**Virtual Component Interconnect**) and **OCP-IP** (**Open Core Protocol-International Partnership**), which are also competing for market share (Bhakthavatchalu et al., 2010). On-chip buses are only the start; people are now putting complete networks on a chip (Ahmadinia and Shahrabi, 2011).

With chip manufacturers having increasing difficulty in raising clock frequencies due to heat-dissipation problems, single-chip multiprocessors are a very hot topic. More information can be found in Gupta et al. (2010), Herrero et al. (2010), and Mishra et al. (2011).

8.2 COPROCESSORS

Having examined some ways of achieving on-chip parallelism, let us now move up a step and look at how the computer can be speeded up by adding a second, specialized processor. These **coprocessors** come in many varieties, from small to large. On the IBM 360 mainframes and all of their successors, independent I/O channels exist for doing input/output. Similarly, the CDC 6600 had 10 independent processors for doing I/O. Graphics and floating-point arithmetic are other areas where coprocessors have been used. Even a DMA chip can be seen as a coprocessor. In some cases, the CPU gives the coprocessor an instruction or set of instructions and tells it to execute them; in other cases, the coprocessor is more independent and runs pretty much on its own.

Physically, coprocessors can range from a separate cabinet (the 360 I/O channels) to a plug-in board (network processors) to an area on the main chip (floating-point). In all cases, what distinguishes them is the fact that some other processor is the main processor and the coprocessors are there to help it. We will now examine three areas where speed-ups are possible: network processing, multimedia, and cryptography.

8.2.1 Network Processors

Most computers nowadays are connected to a network or to the Internet. As a result of technological progress in network hardware, networks are now so fast that it has become increasingly difficult to process all the incoming and outgoing data in software. As a consequence, special network processors have been developed to

handle the traffic, and many high-end computers now have one of these processors. In this section we will first give a brief introduction to networking and then discuss how network processors work.

Introduction to Networking

Computer networks come in two general types: **local-area networks**, or **LAN**s, which connect multiple computers within a building or campus, and **wide-area networks** or **WAN**s, which connect computers spread over a large geographic area. The most popular LAN is called **Ethernet**. The original Ethernet consisted of a fat cable into which a wire coming from each computer was forcibly inserted using what was euphemistically referred to a **vampire tap**. Modern Ethernets have the computers attached to a central switch, as illustrated in the right-hand portion of Fig. 8-14. The original Ethernet crawled along at 3 Mbps, but the first commercial version was 10 Mbps. It was eventually replaced by fast Ethernet at 100 Mbps and then by gigabit Ethernet at 1 Gbps. A 10-gigabit Ethernet is already on the market and a 40-gigabit Ethernet is in the pipeline.

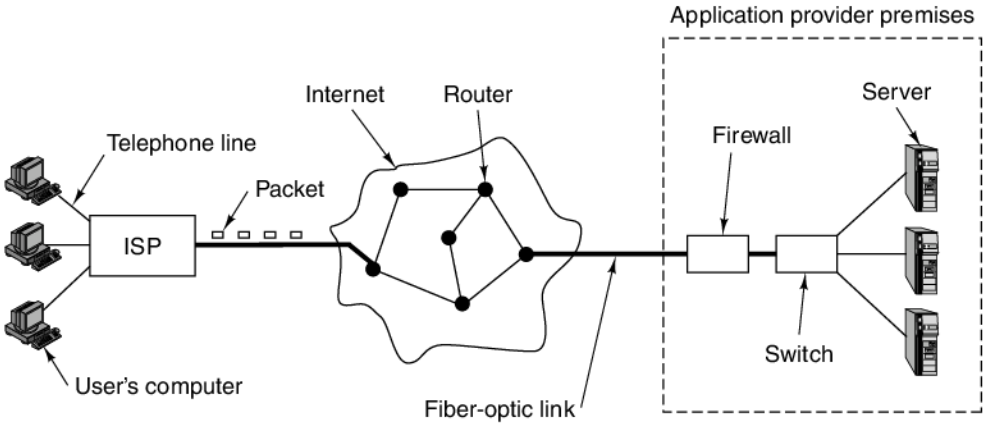

Figure 8-14. How users are connected to servers on the Internet.

WANs are organized differently. They consist of specialized computers called **routers** connected by wires or optical fibers, as shown in the middle of Fig. 8-14. Chunks of data called **packets**, typically 64 to about 1500 bytes, are moved from the source machine through one or more routers until they reach their destination. At each hop, a packet is stored in the router's memory and then forwarded to the next router along the path as soon as the needed transmission line is available. This technique is called **store-and-forward packet switching**.

Although many people think of the Internet as a single WAN, technically it is a collection of many WANs connected together. However, for our purposes, that distinction is not important. Figure 8-14 gives a bird's-eye view of the Internet from the perspective of a home user. The user's computer is typically connected to a