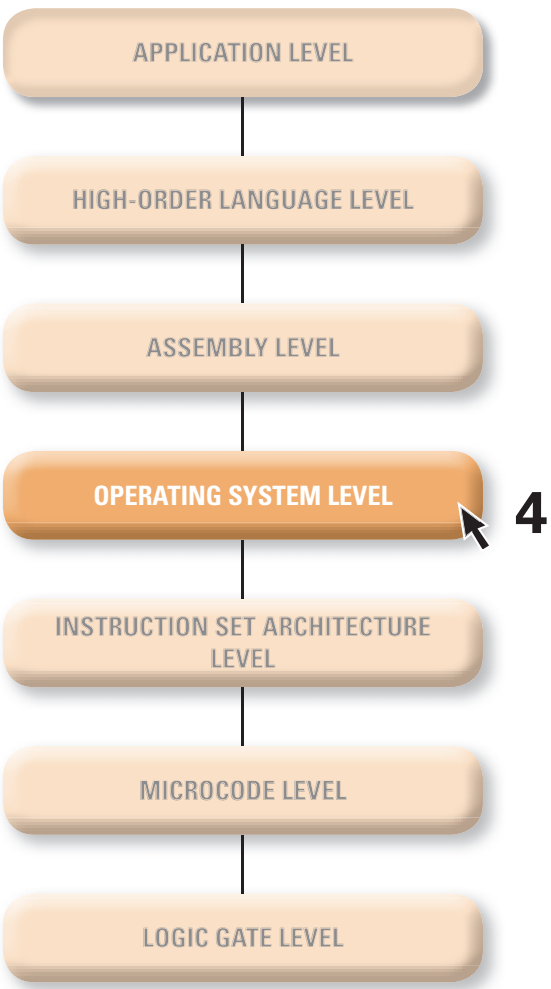
LEVEL

4

Operating System



CHAPTER

8

Process Management

TABLE OF CONTENTS

The purposes of an operating system

An operating system defines a more abstract machine that is easier to program than the machine at Level ISA3. Its purpose is to provide a convenient environment for higher-level programming and to allocate the resources of the system efficiently. The operating system level is between the assembly and machine levels. As is the case with abstraction in general, the operating system hides the details of the Level ISA3 machine from users at higher levels.

The resources of a typical computer system include CPU time, main memory, and disk memory. This chapter describes how an operating system allocates CPU time. Chapter 9 shows how it allocates main memory and disk memory.

There are three general categories of operating systems:

Three types of operating systems

- Single-user
- Multi-user
- Real-time

Mobile devices like smartphones and tablets have single-user operating systems. Such a computer is typically owned and operated by a single individual and is not shared with anyone else. Desktop and laptop computers typically have multi-user operating systems so that accounts for individual users can be set up and the computer shared. Real-time systems are used in computers that are dedicated to controlling equipment. Their inputs are from sensors and their outputs are the control signals for the equipment. For example, the computer that controls an automobile engine is a real-time system.

The Pep/9 operating system is a single-user system. It illustrates some of the techniques used to allocate CPU time. However, it does not illustrate the management of main memory or disk memory. The first two sections of this chapter include a complete listing of the Pep/9 operating system.

8.1 Loaders

An important function of an operating system is to manage the jobs that users submit to be executed. In a multi-user system, several users continually submit jobs. The operating system must decide which job to run from a list of pending jobs. After it decides which job to execute next, it must load the appropriate program into main memory and turn control of the CPU over to that program for execution.

The Pep/9 Operating System

FIGURE 8.1 shows the location of the Pep/9 operating system in main memory. The random-access memory (RAM) part of the operating system consists of the system stack, whose first byte will be allocated at FC0E, the system globals at FC0F to FC14, the input device at FC15, and the output device at FC16. The read-only memory (ROM) part of the operating system, which is shaded in the figure, consists of the loader at FC17, the trap handler at FC52, and the six machine vectors at FFF4 through FFFE. Although the Pep/9 operating system illustrates the operation of a loader, it does not illustrate the process by which the operating system must decide which job to run from a list of pending jobs.

This chapter describes the Pep/9 operating system, which is written in assembly language. Common practice is to write operating systems in a mixture of a high-order language, usually C, and the assembly language for the particular computer controlled by the operating system. Typically, more than 95% of the system is in the high-order language and less than 5% is in the assembly language. The assembly language portion is reserved for those parts of the operating system that cannot be programmed with the features available in the high-order language, or that require an extra measure of efficiency that even an optimizing compiler cannot achieve.

FIGURE 8.2 shows the global constants and variables of the Pep/9 operating system. Symbols `TRUE` and `FALSE` are declared with the `.EQUATE` command and thus generate no object code. They are used throughout the rest of the program.

Symbols `osRAM`, `wordTemp`, `byteTemp`, `addrMask`, `opAddr`, `charIn`, and `charOut` are all defined with the `.BLOCK` command. Normally, `.BLOCK` generates code, and any code generated starts at address 0000 (hex). The listing shows these `.BLOCK` commands generating no code and `osRAM` starting at FB8F instead of at 0000.

The reason for this peculiar assembler behavior is the `.BURN` command at FC17. When you include `.BURN` in a program, the assembler assumes that the program will be burned into ROM. It generates code for those instructions that follow the burn directive but not for those that precede it. The assembler also assumes that the last byte of ROM will be installed at the address given by the `.BURN` directive, leaving the top of memory for the application programs. It therefore calculates the addresses for the symbol table, such that the last byte generated will have the address specified by the burn directive.

In this listing, the burn directive indicates that the last byte should be at address FFFF. Figure 8.16, at the end of the operating system, shows that

FIGURE 8.1
A memory map of the Pep/9 memory. The shaded part is read-only memory.

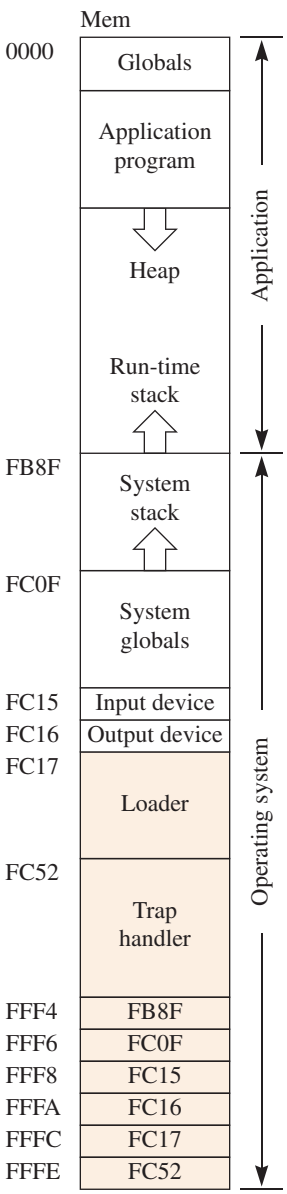


FIGURE 8.2
The global constants and variables of the Pep/9 operating system.

```
              ;******* Pep/9 Operating System, 2015/05/17
              ;
              TRUE:     .EQUATE 1
              FALSE:    .EQUATE 0
              ;
              ;******* Operating system RAM
FB8F          osRAM:    .BLOCK  128         ;System stack area
FC0F          wordTemp: .BLOCK  1           ;Temporary word storage
FC10          byteTemp: .BLOCK  1           ;Least significant byte of wordTemp
FC11          addrMask: .BLOCK  2           ;Addressing mode mask
FC13          opAddr:   .BLOCK  2           ;Trap instruction operand address
FC15          charIn:   .BLOCK  1           ;Memory-mapped input device
FC16          charOut:  .BLOCK  1           ;Memory-mapped output device
              ;
              ;******* Operating system ROM
FC17                    .BURN   0xFFFF
```

the last byte, 52 (hex), is indeed at address FFFF. Because FFFF (hex) is 65,535 (dec), the Pep/9 computer is configured with a total of 64 KiB of main memory. You can change the value in the `.BURN` directive to change where the operating system is installed and the system will still work. For example, if you change the value from 0xFFFF to 0x7FFF and select the option to assemble and install the operating system, the last byte of ROM will be at address 32 Ki minus 1 instead of at 64 Ki minus 1. The symbols and machine vectors will all be recomputed and the system will still run correctly.

The Pep/9 Loader

FIGURE 8.3 shows the Pep/9 loader. To invoke the loader, you select the load option from the simulator. This triggers the following two events:

Invoking the Pep/9 loader

$$SP \leftarrow Mem[FFF6]$$
$$PC \leftarrow Mem[FFFC]$$

Because Mem[FFF6] contains FC0F, as shown in both Figure 8.1 and Figure 8.16, the stack pointer (SP) is initialized to FC0F. Similarly, the program counter (PC) is initialized to FC17, the address of the first instruction of the loader.

FIGURE 8.3
The loader of the Pep/9 operating system.

```
            ;******* System Loader
            ;Data must be in the following format:
            ;Each hex number representing a byte must contain exactly two
            ;characters. Each character must be in 0..9, A..F, or a..f and
            ;must be followed by exactly one space. There must be no
            ;leading spaces at the beginning of a line and no trailing
            ;spaces at the end of a line. The last two characters in the
            ;file must be lowercase zz, which is used as the terminating
            ;sentinel by the loader.
            ;
FC17  C80000 loader:  LDWX    0,i         ;X <- 0
            ;
FC1A  D1FC15 getChar: LDBA    charIn,d    ;Get first hex character
FC1D  B0007A          CPBA    'z',i       ;If end of file sentinel 'z'
FC20  18FC51          BREQ    stopLoad    ;  then exit loader routine
FC23  B00039          CPBA    '9',i       ;If character <= '9', assume decimal
FC26  14FC2C          BRLE    shift       ;  and right nybble is correct digit
FC29  600009          ADDA    9,i         ;else convert nybble to correct digit
FC2C  0A      shift:  ASLA                ;Shift left by four bits to send
FC2D  0A              ASLA                ;  the digit to the most significant
FC2E  0A              ASLA                ;  position in the byte
FC2F  0A              ASLA
FC30  F1FC10          STBA    byteTemp,d  ;Save the most significant nybble
FC33  D1FC15          LDBA    charIn,d    ;Get second hex character
FC36  B00039          CPBA    '9',i       ;If character <= '9', assume decimal
FC39  14FC3F          BRLE    combine     ;  and right nybble is correct digit
FC3C  600009          ADDA    9,i         ;else convert nybble to correct digit
FC3F  80000F combine: ANDA    0x000F,i    ;Mask out the left nybble
FC42  91FC0F          ORA     wordTemp,d  ;Combine both hex digits in binary
FC45  F50000          STBA    0,x         ;Store in Mem[X]
FC48  680001          ADDX    1,i         ;X <- X + 1
FC4B  D1FC15          LDBA    charIn,d    ;Skip blank or <LF>
FC4E  12FC1A          BR      getChar     ;
            ;
FC51  00      stopLoad:STOP               ;
```

The loader begins at FC17 by clearing the index register to zero, which is the address of the first byte to load. The code from FC1A to FC42 gets the next two hex characters from the input stream into the low-order byte of the accumulator. The store byte accumulator instruction at FC45 loads the byte into memory at the address specified by the index register. The add index register instruction at FC48 increments the index register by one in preparation for loading the next byte.

The loader is in the form of a single loop that inputs a character and compares it with sentinel `z`. If the character is not the sentinel, the program checks whether it is in `'0'..'9'`. If it is not in that range, the rightmost four bits, called a *nybble* because it is half a byte, is converted to the proper value by adding 9 to it. Note that ASCII `A` is 0100 0001 (bin), so that when 9 is added to it the sum is 0100 1010. The rightmost nybble is the correct bit pattern for hexadecimal digit A. It will similarly be correct for hexadecimal digits B through F. If the character is in `'0'..'9'`, the rightmost nybble is already correct.

The definition of a nybble

The loader shifts the nybble four bits to the left and stores it temporarily in `byteTemp`. It inputs the second character of the pair, adjusts the nybble similarly, and combines both nybbles into a single byte with the `ANDA` at FC3F and `ORA` at FC42. Unfortunately, Pep/9 does not have an AND byte or an OR byte instruction. So it must use the word versions of these operations. You can see from Figure 8.2 that `byteTemp` is the least significant byte of `wordTemp`, which is why `ORA` can use `wordTemp` to access `byteTemp`. The loader terminates with the `STOP` instruction, which returns control to the simulator options.

Load modules are typically not in ASCII.

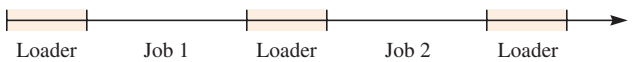
Programs to be loaded typically are not in the format of hexadecimal ASCII characters. They are already in binary, ready to be loaded. Pep/9 uses ASCII characters for the object file, so you can program directly in machine language and view the object file with a text editor.

Program Termination

The application programs presented thus far have all terminated with the `STOP` instruction. The `STOP` instruction in a real computer is rarely executed. Rather than generate a `STOP` instruction at the end of a program, a C compiler generates an instruction that returns control to the operating system. If your program ran on a personal computer, the operating system would set up the screen to wait for you to request another service. If your program ran on a remote timesharing system, the operating system would continue to process other users' jobs. In no case would the computer itself simply stop.

If there is only one CPU, it alternates between executing operating system jobs and application jobs. FIGURE 8.4 shows a time line of CPU usage when

FIGURE 8.4
A time line of CPU usage when the operating system loads and executes a sequence of jobs.



the operating system loads and executes a sequence of jobs. The shaded parts represent that part of the time spent executing the operating system.

The operating system represents the overhead necessary for doing business. When you shop at a mall, the price you pay for a widget does not reflect just the cost of production and transportation of the widget to the store. It also reflects the salesperson's salary, the electricity for the store lighting, the fringe benefits for the store manager, and so on. Similarly, 100% of a computer's resources does not go toward executing user programs. A certain fraction of the resources, in this case CPU time, must be reserved for the operating system.

8.2 Traps

When programming in assembly language at Level Asmb5, you may use these four instructions: `DECI`, `DECO`, `HEXO`, and `STRO`. Figure 4.6 shows no such instructions in the Level ISA3 machine. Instead, when the computer fetches the instructions with these opcodes, the hardware executes a trap. A trap is similar to a subroutine jump, but more elaborate. The code that executes is called a *trap routine* or *trap handler* instead of a *subroutine*. The operating system returns control to the application program by executing a return from trap instruction, `RETTR`, instead of a return from subroutine instruction, `RET`.

The trap handler implements the four instructions as if they were part of the Level ISA3 machine. Remember that one purpose of an operating system is to provide a convenient environment for higher-level programming. The abstract machine provided by the Pep/9 operating system is a more convenient machine because it contains these four additional instructions not present at Level ISA3. In addition to `DECI`, `DECO`, `HEXO`, and `STRO`, the operating system provides two unary trap instructions and one nonunary trap instruction, called *no-operations*, with mnemonics `NOP0`, `NOP1`, and `NOP`. These instructions do nothing when they execute and are provided so you can reprogram them to implement new instructions of your own choosing.

The `NOP` *trap instructions*

The Trap Mechanism

Here is the register transfer language (RTL) specification for a trap instruction:

A Pep/9 trap

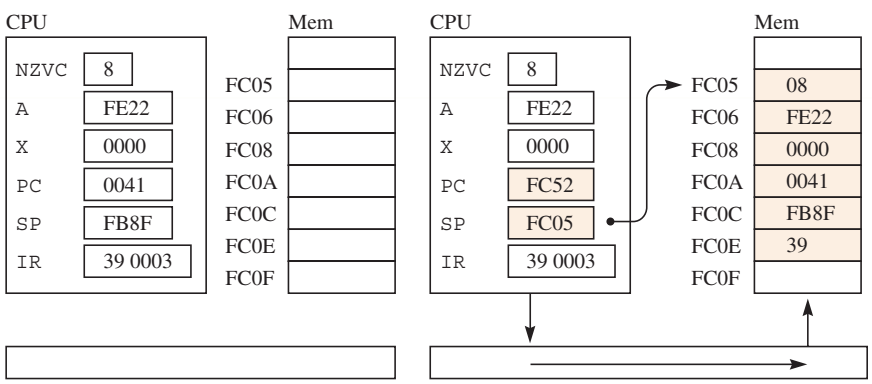
```
Temp                 ← Mem[FFF6] ;
Mem[Temp – 1]        ← IR⟨0..7⟩ ;
Mem[Temp – 3]        ← SP ;
Mem[Temp – 5]        ← PC ;
Mem[Temp – 7]        ← X ;
Mem[Temp – 9]        ← A ;
Mem[Temp – 10]⟨4..7⟩ ← NZVC ;
SP                   ← Temp – 10 ;
PC                   ← Mem[FFFE]
```

Temp represents a temporary value for notational convenience. Mem[FFF6] contains FC0F, the address of the system stack. In the first event, Temp gets FC0F. The next six events show the CPU pushing the content of all the registers onto the system stack, starting with the instruction specifier of the IR and ending with the NZVC flags. The stack pointer is then modified to point to the new top of the system stack, and the program counter gets the content of Mem[FFFE].

FIGURE 8.5 shows an example of such a trap from Figure 5.11. The program in Figure 5.11 contains the following decimal output trap:

```
003E 390003 DECO 0x0003,d ;Output the sum
```

FIGURE 8.5
A trap triggered by the execution of the DECO trap instruction 390003.



(a) Before the trap. **(b)** After the trap.

where 003E is the address of the instruction and 390003 is the object code that triggered the trap during execution.

Figure 8.5(a) shows the state of the CPU before the trap executes, and Figure 8.5(b) shows the state after the trap executes. Only the instruction specifier part of the IR is pushed onto the stack. Also note that the four NZVC bits are right justified in the byte at Mem[FC05]. The leftmost nybble of the byte is zero. SP contains FC05, the new top of the system stack, and PC contains Mem[FFFE], which is FC52, the address of the first instruction of the trap handler. Figure 8.16 shows how the operating system sets up the machine vectors at FFF6 and FFFE with the `.ADDRSS` commands.

The RETTR Instruction

The definition of a process

A program during execution is called a *process*. The trap mechanism temporarily suspends the process so the operating system can perform a service. The block of information in main memory that contains a copy of the trapped process's registers is called a *process control block* (PCB). The PCB for this example is stored in Mem[FC05] to Mem[FC0E], as shown in Figure 8.5(b).

The process control block (PCB)

After the operating system performs its service, it must eventually return control of the CPU to the suspended process so the process can complete its execution. In this example, the service performed by the Pep/9 operating system is execution of the `DECO` instruction. It returns control back to the process by executing the return from trap instruction, `RETTR`.

Here is the RTL specification for the `RETTR` instruction:

The `RETTR` instruction

$$
\begin{array}{lcl}
\text{NZVC} & \leftarrow & \text{Mem[SP]}\langle 4..7\rangle\,; \\
\text{A} & \leftarrow & \text{Mem[SP + 1]}\,; \\
\text{X} & \leftarrow & \text{Mem[SP + 3]}\,; \\
\text{PC} & \leftarrow & \text{Mem[SP + 5]}\,; \\
\text{SP} & \leftarrow & \text{Mem[SP + 7]}
\end{array}
$$

`RETTR` pops the top nine bytes off the stack into the NZVC, A, X, PC, and SP registers. This reverses the events of the trap, except that IR is not popped. The next instruction to execute will be the one specified by the new value of PC. The last register to change is SP.

If the trap handler does not modify any of the values in the PCB, `RETTR` will restore the original values in the CPU registers when the process resumes. In particular, the SP will again point to the top of the application stack, as it did at the time of the trap. On the other hand, any changes that the trap handler makes to the values in the PCB will be reflected in the CPU registers when the process resumes.

The Trap Handlers

FIGURE 8.6 shows the entry and exit points of the trap handlers. `oldIR` is the stack address of the copy of the IR register stored on the system stack from the trap mechanism. FIGURE 8.7(a) shows the stack addresses of all the registers.

FIGURE 8.6
The entry and exit points of the trap handlers in the Pep/9 operating system.

```
             ;******* Trap handler
             oldIR:    .EQUATE 9           ;Stack address of IR on trap
             ;
FC52  DB0009 trap:     LDBX    oldIR,s     ;X <- trapped IR
FC55  B80028           CPBX    0x0028,i    ;If X >= first nonunary trap opcode
FC58  1CFC67           BRGE    nonUnary    ;  trap opcode is nonunary
             ;
FC5B  880001 unary:    ANDX    0x0001,i    ;Mask out all but rightmost bit
FC5E  0B               ASLX                ;Two bytes per address
FC5F  25FC63           CALL    unaryJT,x   ;Call unary trap routine
FC62  02               RETTR               ;Return from trap
             ;
FC63  FD6B   unaryJT:  .ADDRSS opcode26    ;Address of NOP0 subroutine
FC65  FD6C             .ADDRSS opcode27    ;Address of NOP1 subroutine
             ;
FC67  0D     nonUnary:ASRX                 ;Trap opcode is nonunary
FC68  0D               ASRX                ;Discard addressing mode bits
FC69  0D               ASRX
FC6A  780005           SUBX    5,i         ;Adjust so that NOP opcode = 0
FC6D  0B               ASLX                ;Two bytes per address
FC6E  25FC72           CALL    nonUnJT,x   ;Call nonunary trap routine
FC71  02     return:   RETTR               ;Return from trap
             ;
FC72  FD6D   nonUnJT:  .ADDRSS opcode28    ;Address of NOP subroutine
FC74  FD77             .ADDRSS opcode30    ;Address of DECI subroutine
FC76  FEEB             .ADDRSS opcode38    ;Address of DECO subroutine
FC78  FF76             .ADDRSS opcode40    ;Address of HEXO subroutine
FC7A  FFC2             .ADDRSS opcode48    ;Address of STRO subroutine
```

When a trap instruction executes, the next instruction to execute is the one at FC52, the first instruction in Figure 8.6. The trap could have been triggered by any of the following instructions:

```
0010 011n,   NOPn,   Unary no-operation trap
0010 1aaa,   NOP,    Nonunary no-operation trap
0011 0aaa,   DECI,   Nonunary decimal-input trap
0011 1aaa,   DECO,   Nonunary decimal-output trap
0100 0aaa,   HEXO,   Nonunary hexadecimal-output trap
0100 1aaa,   STRO,   Nonunary string-output trap
```

The code in Figure 8.6 determines which instruction triggered the trap and calls the specific handler that implements that instruction. There are seven trap handlers, two for the unary `NOPn` instructions and five for the nonunary instructions. Remember that the fetch part of the von Neumann cycle puts the instruction specifier in the instruction register (IR). After the trap occurs, the instruction specifier of the instruction that caused the trap is available on the system stack, because it was pushed there by the trap mechanism. The code in Figure 8.6 accesses the saved instruction specifier to determine which instruction triggered the trap.

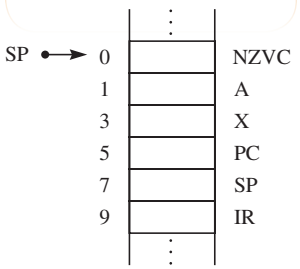
The first instruction in Figure 8.6 gets the opcode from the copy of IR that was pushed onto the system stack. The `NOP` instruction has the first nonunary opcode, 0010 1aaa. Furthermore, 0010 1000 (bin) is 28 (hex). The `CPBX` instruction at FC55 compares the trap opcode with 28 (hex). If the trap opcode is less than this value, the trap instruction is unary; otherwise, it is nonunary.

If the trap instruction is unary, it must be one of the following two instructions:

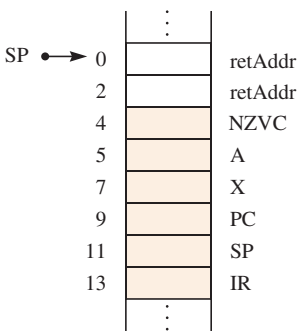
```
0010 0110,   NOP0,   rightmost bit is 0
0010 0111,   NOP1,   rightmost bit is 1
```

The `ANDX` instruction at FC5B masks out all but the rightmost bit, which is sufficient to determine which of the two instructions caused the trap. The `CALL` instruction at FC5F uses the jump table technique with indexed addressing as described in the program of Figure 6.40. That figure shows how the compiler translates a C `switch` statement using an array of addresses with the unconditional branch instruction `BR`. The code in Figure 8.6 differs slightly from that in Figure 6.40 because it uses `CALL` instead of `BR`, but the principle is the same. The jump table at FC63 is an array of addresses, each element of which is the address of the first statement to execute in the trap handler for the specific instruction that triggered the trap. Because a `CALL` executes, it pushes a return address onto the stack. The last instruction to execute in a specific trap handler is `RET`, which returns control to FC62. The

FIGURE 8.7
The stack addresses of the copies of the CPU registers.



(a) Immediately after a trap.



(b) With two return addresses on the run-time stack. The shaded region is the PCB.

The test for the `NOPn` instructions

instruction at FC62 is RETTR, which restores the registers in the CPU from the PCB and returns control to the instruction following the trap instruction.

The instructions at FC67 through FC7A do the same thing for the group of nonunary instructions. The three ASRX instructions discard the addressing mode bits, and the SUBX instruction makes an adjustment so that the content of the index register will be

The test for the nonunary trap instructions

0 if the trap IR contains 0010 1aaa, NOP
1 if the trap IR contains 0011 0aaa, DECI
2 if the trap IR contains 0011 1aaa, DECO
3 if the trap IR contains 0100 0aaa, HEXO
4 if the trap IR contains 0100 1aaa, STRO

As with the unary instructions, the CALL at FC6E branches to the trap handler for the specific instruction. After the trap handler implements the instruction, it returns control to the RETTR instruction at FC71, which in turn returns control to the statement after the one that caused the trap.

Trap Addressing Mode Assertion

Different instructions have different allowed addressing modes. For example, Figure 5.2 shows that the STWA instruction is not allowed to have immediate addressing, while the STRO instruction is only allowed to have direct, indirect, stack-relative, stack-relative deferred, and indexed addressing. Because the STWA instruction is hardwired into the CPU, the hardware detects whether an addressing error has occurred. But the trap instructions, such as STRO, are not native to the CPU. The trap handler implements them in software. The question then arises, how does a trap handler detect whether a trap instruction is attempting to use an illegal addressing mode? It does so with the addressing mode assert routine of FIGURE 8.8.

The addressing mode assert routine must access the trap IR, which is saved on the system stack. Immediately after the trap, the IR has a stack address of 9, as Figure 8.7(a) shows. However, by the time the addressing mode assert routine is called, two additional return addresses are on top of the system stack. One comes from a CALL instruction in the trap handler code of Figure 8.6, and one comes from the CALL in the specific trap handler. Figure 8.7(b) shows the PCB on the system stack after the addressing-mode assert routine is called and the two return addresses are on the stack. The stack address of the trap IR is now 13 instead of 9 because of the four bytes occupied by the two return addresses.

The routine in Figure 8.8 has the following pre- and postconditions:

Pre- and postconditions for the addressing mode assert routine

- Precondition: addrMask is a bit mask representation of the set of allowable addressing modes, and the PCB of the trap instruction is on the system stack.

FIGURE 8.8
The trap addressing mode assertion in the Pep/9 operating system.

```
             ;******* Assert valid trap addressing mode
             oldIR4:  .EQUATE 13          ;oldIR + 4 with two return addresses
FC7C  D00001 assertAd:LDBA    1,i         ;A <- 1
FC7F  DB000D          LDBX    oldIR4,s    ;X <- OldIR
FC82  880007          ANDX    0x0007,i    ;Keep only the addressing mode bits
FC85  18FC8F          BREQ    testAd      ;000 = immediate addressing
FC88  0A      loop:   ASLA                ;Shift the 1 bit left
FC89  780001          SUBX    1,i         ;Subtract from addressing mode count
FC8C  1AFC88          BRNE    loop        ;Try next addressing mode
FC8F  81FC11 testAd:  ANDA    addrMask,d  ;AND the 1 bit with legal modes
FC92  18FC96          BREQ    addrErr
FC95  01              RET                 ;Legal addressing mode, return
FC96  D0000A addrErr: LDBA    '\n',i
FC99  F1FC16          STBA    charOut,d
FC9C  C0FCA9          LDWA    trapMsg,i   ;Push address of error message
FC9F  E3FFFE          STWA    -2,s
FCA2  580002          SUBSP   2,i         ;Call print subroutine
FCA5  24FFDE          CALL    prntMsg
FCA8  00              STOP                ;Halt: Fatal runtime error
FCA9  455252 trapMsg: .ASCII  "ERROR: Invalid trap addressing mode.\x00"
      ...
```

- Postcondition: If the addressing mode of the trap instruction is in the set of allowable addressing modes, control is returned to the trap handler. Otherwise, an invalid addressing mode message is output and the program halts with a fatal run-time error.

The addressing mode assert routine is the Asmb5 version of the `assert()` statement found in some HOL6 languages. In C, the assert facility is in the `<assert.h>` library, which you can include in your programs with the `#include` compiler directive.

A trap handler uses the assert routine by first setting the value in global variable `addrMask` shown at FC11 in Figure 8.2 to indicate the allowable addressing modes for that particular instruction. Then it calls `assertAd` at FC7C in Figure 8.8. The routine assumes a common representation of a set known as the *bit-mapped representation*. In machine language, each bit can have a value of either 0 or 1. The bit-mapped representation of the set of allowable addressing modes associates each addressing mode with one bit in

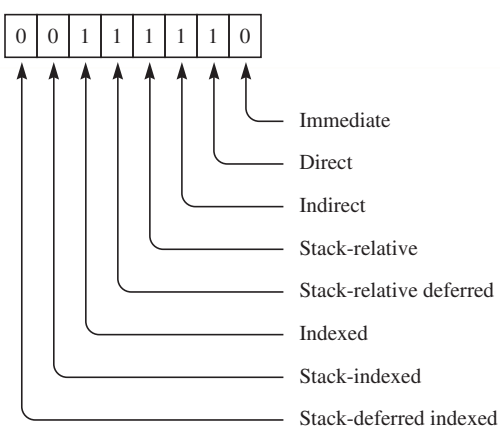
The bit-mapped representation of a set

addrMask. If the bit has the value 0, the associated addressing mode is not in the set. If the bit has the value 1, the addressing mode is in the set.

FIGURE 8.9 shows the rightmost byte in addrMask with the precondition set by the trap handler for the STRO instruction, which can use direct, indirect, stack-relative, stack-relative deferred, or indexed addressing. The bits associated with those addressing modes have a value of 1, and the others have a value of 0. Mathematically, the mask represents the set {Direct, Indirect, Stack-relative, Stack-relative deferred, Indexed}.

To illustrate how the assert routine in Figure 8.8 tests for set membership, suppose the STRO instruction executes with stack-relative deferred addressing, so that its addressing-aaa field is 100. This is an allowable addressing mode. First, the LDBA statement at FC7C sets the rightmost byte of the accumulator to 0000 0001. The next two statements set the index register to 4 (dec) based on the addressing-aaa field of the trap instruction. A loop then counts down to zero starting from this value, shifting the 1 bit in the accumulator left each time through the loop. The accumulator ends up with 0001 0000, so that the 1 bit is at the bit position associated with stack-relative deferred addressing. The ANDA statement at FC8F takes the AND of the accumulator with the address mask in Figure 8.9. Because the 1's line up at the fifth bit from the right, the result is not zero, and control returns to the trap handler. If stack-relative deferred addressing were not allowed, the address mask would have a 0 at the fifth bit from the right, the result of the AND operation would be zero, and the assertion would fail.

FIGURE 8.9
The bits in addrMask associated with the allowable addressing modes of the STRO trap instruction.



Trap Operand Address Computation

The trap operand address computation is another routine called by the nonunary trap handlers. The addressing modes for the native instructions are hardwired into the CPU. But the trap instructions are implemented in software instead of hardware. So, the eight addressing modes must be simulated in software. FIGURE 8.10 shows the routine that performs the computation.

The routine in Figure 8.10 has the following pre- and postconditions:

Pre- and postconditions for the addressing mode computation routine

- Precondition: The PCB of the stack instruction is on the system stack.
- Postcondition: `opAddr` contains the address of the operand according to the addressing mode of the trap instruction.

As with the addressing-mode assert routine in Figure 8.8, the register copies on the PCB have a stack offset four bytes greater than when the trap occurs, as Figure 8.7(b) shows. The routine uses `oldIR4`, defined in the addressing-mode assert routine—as well as the similarly defined `oldX4`, `oldPC4`, and `oldSP4`—to access the copies of the saved index register, program counter, and stack pointer.

The first four statements beginning at FCCE determine the addressing mode of the trap instruction and branch to the computation for that addressing mode. The program uses the jump table technique to switch between one of eight alternatives. The code for each of the eight alternatives computes the address of the operand by inspecting the state of the CPU at the time of the trap.

The first two instructions of each computation are

The first two instructions of each computation

```
LDWX oldPC4,s
SUBX 2,i
```

Because the trap instruction is nonunary, the program counter at the time of the trap points to the byte after the two-byte operand specifier. The first instruction loads the saved program counter into the index register, and the second instruction subtracts two from it. After these two instructions execute, the index register contains the address of the operand specifier in the instruction that caused the trap.

For immediate addressing, the operand specifier is the operand. Consequently, the statements at FCEE

The computation for immediate addressing

```
STWX opAddr,d
RET
```

simply store the address of the operand specifier in `opAddr` as required.

FIGURE 8.10
The trap operand address computation in the Pep/9 operating system.

```
            ;******* Set address of trap operand
            oldX4:   .EQUATE 7           ;oldX + 4 with two return addresses
            oldPC4:  .EQUATE 9           ;oldPC + 4 with two return addresses
            oldSP4:  .EQUATE 11          ;oldSP + 4 with two return addresses
FCCE  DB000D setAddr: LDBX    oldIR4,s   ;X <- old instruction register
FCD1  880007          ANDX    0x0007,i   ;Keep only the addressing mode bits
FCD4  0B              ASLX               ;Two bytes per address
FCD5  13FCD8          BR      addrJT,x
FCD8  FCE8   addrJT:  .ADDRSS addrI      ;Immediate addressing
FCDA  FCF2            .ADDRSS addrD      ;Direct addressing
FCDC  FCFF            .ADDRSS addrN      ;Indirect addressing
FCDE  FD0F            .ADDRSS addrS      ;Stack-relative addressing
FCE0  FD1F            .ADDRSS addrSF     ;Stack-relative deferred addressing
FCE2  FD32            .ADDRSS addrX      ;Indexed addressing
FCE4  FD42            .ADDRSS addrSX     ;Stack-indexed addressing
FCE6  FD55            .ADDRSS addrSFX    ;Stack-deferred indexed addressing
            ;
FCE8  CB0009 addrI:   LDWX    oldPC4,s   ;Immediate addressing
FCEB  780002          SUBX    2,i        ;Oprnd = OprndSpec
FCEE  E9FC13          STWX    opAddr,d
FCF1  01              RET
            ;
FCF2  CB0009 addrD:   LDWX    oldPC4,s   ;Direct addressing
FCF5  780002          SUBX    2,i        ;Oprnd = Mem[OprndSpec]
FCF8  CD0000          LDWX    0,x
FCFB  E9FC13          STWX    opAddr,d
FCFE  01              RET
            ;
FCFF  CB0009 addrN:   LDWX    oldPC4,s   ;Indirect addressing
FD02  780002          SUBX    2,i        ;Oprnd = Mem[Mem[OprndSpec]]
FD05  CD0000          LDWX    0,x
FD08  CD0000          LDWX    0,x
FD0B  E9FC13          STWX    opAddr,d
FD0E  01              RET
            ;
```

```
FD0F  CB0009 addrS:   LDWX    oldPC4,s    ;Stack-relative addressing
FD12  780002          SUBX    2,i         ;Oprnd = Mem[SP + OprndSpec]
FD15  CD0000          LDWX    0,x
FD18  6B000B          ADDX    oldSP4,s
FD1B  E9FC13          STWX    opAddr,d
FD1E  01              RET
            ;
FD1F  CB0009 addrSF:  LDWX    oldPC4,s    ;Stack-relative deferred addressing
FD22  780002          SUBX    2,i         ;Oprnd = Mem[Mem[SP + OprndSpec]]
FD25  CD0000          LDWX    0,x
FD28  6B000B          ADDX    oldSP4,s
FD2B  CD0000          LDWX    0,x
FD2E  E9FC13          STWX    opAddr,d
FD31  01              RET
            ;
FD32  CB0009 addrX:   LDWX    oldPC4,s    ;Indexed addressing
FD35  780002          SUBX    2,i         ;Oprnd = Mem[OprndSpec + X]
FD38  CD0000          LDWX    0,x
FD3B  6B0007          ADDX    oldX4,s
FD3E  E9FC13          STWX    opAddr,d
FD41  01              RET
            ;
FD42  CB0009 addrSX:  LDWX    oldPC4,s    ;Stack-indexed addressing
FD45  780002          SUBX    2,i         ;Oprnd = Mem[SP + OprndSpec + X]
FD48  CD0000          LDWX    0,x
FD4B  6B0007          ADDX    oldX4,s
FD4E  6B000B          ADDX    oldSP4,s
FD51  E9FC13          STWX    opAddr,d
FD54  01              RET
            ;
FD55  CB0009 addrSFX: LDWX    oldPC4,s    ;Stack-deferred indexed addressing
FD58  780002          SUBX    2,i         ;Oprnd = Mem[Mem[SP + OprndSpec] + X]
FD5B  CD0000          LDWX    0,x
FD5E  6B000B          ADDX    oldSP4,s
FD61  CD0000          LDWX    0,x
FD64  6B0007          ADDX    oldX4,s
FD67  E9FC13          STWX    opAddr,d
FD6A  01              RET
```

For direct addressing, the operand specifier is the address of the operand. The first of the statements at FCF8

The computation for direct addressing

```
LDWX 0,x
STWX opAddr,d
RET
```

replaces the index register with the content in memory whose address is in the index register. Before the instruction executes, the index register contains the address of the operand specifier. After the instruction executes, the index register contains the operand specifier itself. Because the operand specifier is the address of the operand, that is what gets stored in `opAddr`.

For indirect addressing, the operand specifier is the address of the address of the operand. As with direct addressing, the first of the statements at FD05

The computation for indirect addressing

```
LDWX 0,x
LDWX 0,x
STWX opAddr,d
RET
```

replaces the index register with the operand specifier itself, which is the address of the address of the operand. The second instruction fetches the address of the operand, which gets stored in `opAddr`.

For stack-relative addressing, the stack pointer plus the operand specifier is the address of the operand. The first of the statements at FD15

The computation for stack-relative addressing

```
LDWX 0,x
ADDX oldSP4,s
STWX opAddr,d
RET
```

puts the operand specifier in the index register. The second instruction adds the copy of the stack pointer to it. The result is the address of the operand, which gets stored in `opAddr`.

The remaining four addressing modes use similar techniques to compute the address of the operand. Stack-relative deferred addressing is one extra level of indirection compared with stack-relative addressing, requiring one additional execution of `LDWX 0,x`. Indexed addressing is like stack-relative addressing, except the operand specifier is added to the index register instead of the stack pointer. Stack-indexed and stack-deferred indexed are variations on the same theme.

The No-Operation Trap Handlers

FIGURE 8.11 shows the code for implementation of the no-operation trap handlers. Because the no-operation instructions do not do anything, the trap handlers do no processing other than to execute `RET`, returning control to the exit points in Figure 8.6, and eventually to the statement following the trap.

The no-operation instructions are provided for you to write your own trap handlers. Some problems at the end of the chapter ask you to implement instructions that are not in the Pep/9 instruction set. The Pep/9 assembler lets you redefine the mnemonics for the trap instructions. To write a trap handler, you change the mnemonic of one of the no-operation instructions in Figure 8.11 to the mnemonic of your new instruction. Then, you edit the trap handler in the operating system by inserting your code at its entry point. For example, to redefine `NOP0`, you insert the code for your handler at FD6B. The last executable statement in your handler should be `RET`.

Figure 8.11 shows the implementation of nonunary `NOP` at FD6D. Figure 5.2 specifies that its only allowable addressing mode is immediate addressing. Therefore, the value in `addrMask` is set to 0000 0001, where the last 1 is at the bit position for immediate addressing, as Figure 8.9 shows.

FIGURE 8.11
The NOP trap handlers.

```
                ;******* Opcode 0x26
                ;The NOP0 instruction.
FD6B  01        opcode26:RET
                ;
                ;******* Opcode 0x27
                ;The NOP1 instruction.
FD6C  01        opcode27:RET
                ;
                ;******* Opcode 0x28
                ;The NOP instruction.
FD6D  C00001 opcode28:LDWA    0x0001,i    ;Assert i
FD70  E1FC11          STWA    addrMask,d
FD73  24FC7C          CALL    assertAd
FD76  01              RET
```

The DECI Trap Handler

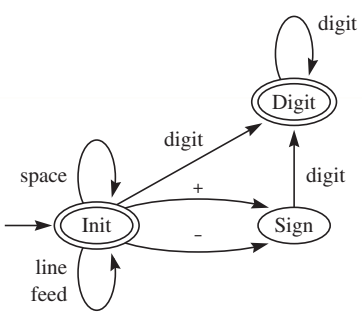
This section describes the trap handler for the DECI instruction. DECI must parse the input, converting the string of ASCII characters to the proper bits in two's complement representation. It uses the finite-state machine (FSM) of FIGURE 8.12. An outline of the logic of the FSM in the DECI trap handler appears in FIGURE 8.13. state has enumerated type with possible values init, sign, or digit.

FIGURE 8.14 is the listing of the DECI trap handler. The first four statements at FD77 call the addressing-mode assert routine and the routine to compute the trap operand address. At FD83, the handler allocates seven local variables on the stack—total, asciiCh, valAscii, isOvfl, isNeg, state, and temp. Each variable except asciiCh occupies two bytes, so SUBSP subtracts 13 from the stack pointer for the allocation. With application programs, SUBSP is the first executable statement in procedures with local variables. Here, SUBSP must execute after the first two routine calls, because those calls access quantities from the PCB. They assume only two return addresses on the stack, as Figure 8.7(b) shows.

The DECI trap handler must access the NZVC bits from the PCB. The handler is called by the CALL instruction at FC6E in Figure 8.6, which pushed a two-byte return address on the stack. When the handler accesses the value of NZVC stored on the stack at the trap, its stack address will be 15 greater than it is immediately after the trap because of the local variables and the return address. That is why oldNZVC equates to 15 instead of 0.

Beginning with the LDWA statement at FD86, the processing in the DECI interrupt handler follows the logic in Figure 8.13. The routine tests the input string for a value that is out of range. If so, it sets the V bit stored in the PCB during the trap. When RETTR returns control to the application, the

FIGURE 8.12
The finite-state machine in the DECI interrupt handler.



```
isOvfl ← FALSE
state ← init
do
   LDBA charIn,d
   STBA asciiCh,s
   switch state
   case init:
       if (asciiCh == '+') {
          isNeg ← FALSE
          state ← sign
       }
       else if (asciiCh == '-') {
          isNeg ← TRUE
          state ← sign
       }
       else if (asciiCh is a digit) {
          isNeg ← FALSE
          total ← value(asciiCh)
          state ← digit
       }
       else if (asciiCh is not <SPACE> or <LF>) {
          Exit with DECI error
       }
   case sign:
       if (asciiCh is a digit) {
          total ← value(asciiCh)
          state ← digit
       }
       else {
          Exit with DECI error
       }
   case digit:
       if (asciiCh is a digit) {
          total ← 10 * total + value(asciiCh)
          if (overflow) {
             isOvfl ← TRUE
          }
       }
           else {
                 Exit normally
           }
        end switch
while (not exit)
```

FIGURE 8.13
The program logic of the DECI trap handler.

FIGURE 8.14
The DECI trap handler.

```
             ;******* Opcode 0x30
             ;The DECI instruction.
             ;Input format: Any number of leading spaces or line feeds are
             ;allowed, followed by '+', '-' or a digit as the first character,
             ;after which digits are input until the first nondigit is
             ;encountered. The status flags N,Z and V are set appropriately
             ;by this DECI routine. The C status flag is not affected.
             ;
             oldNZVC: .EQUATE 15          ;Stack address of NZVC on interrupt
             ;
             total:   .EQUATE 11          ;Cumulative total of DECI number
             asciiCh: .EQUATE 10          ;asciiCh, one byte
             valAscii:.EQUATE 8           ;value(asciiCh)
             isOvfl:  .EQUATE 6           ;Overflow boolean
             isNeg:   .EQUATE 4           ;Negative boolean
             state:   .EQUATE 2           ;State variable
             temp:    .EQUATE 0
             ;
             init:    .EQUATE 0           ;Enumerated values for state
             sign:    .EQUATE 1
             digit:   .EQUATE 2
             ;
FD77  C000FE opcode30:LDWA    0x00FE,i    ;Assert d, n, s, sf, x, sx, sfx
FD7A  E1FC11          STWA    addrMask,d
FD7D  24FC7C          CALL    assertAd
FD80  24FCCE          CALL    setAddr     ;Set address of trap operand
FD83  58000D          SUBSP   13,i        ;Allocate storage for locals
FD86  C00000          LDWA    FALSE,i     ;isOvfl <- FALSE
FD89  E30006          STWA    isOvfl,s
FD8C  C00000          LDWA    init,i      ;state <- init
FD8F  E30002          STWA    state,s
             ;
FD92  D1FC15 do:      LDBA    charIn,d    ;Get asciiCh
FD95  F3000A          STBA    asciiCh,s
FD98  80000F          ANDA    0x000F,i    ;Set value(asciiCh)
FD9B  E30008          STWA    valAscii,s
FD9E  D3000A          LDBA    asciiCh,s   ;A<low> = asciiCh throughout the loop
FDA1  CB0002          LDWX    state,s     ;switch (state)
FDA4  0B              ASLX                ;Two bytes per address
FDA5  13FDA8          BR      stateJT,x
             ;
```

```
FDA8  FDAE   stateJT: .ADDRSS sInit
FDAA  FE08            .ADDRSS sSign
FDAC  FE23            .ADDRSS sDigit
            ;
FDAE  B0002B sInit:   CPBA    '+',i       ;if (asciiCh == '+')
FDB1  1AFDC3          BRNE    ifMinus
FDB4  C80000          LDWX    FALSE,i     ;isNeg <- FALSE
FDB7  EB0004          STWX    isNeg,s
FDBA  C80001          LDWX    sign,i      ;state <- sign
FDBD  EB0002          STWX    state,s
FDC0  12FD92          BR      do
            ;
FDC3  B0002D ifMinus: CPBA    '-',i       ;else if (asciiCh == '-')
FDC6  1AFDD8          BRNE    ifDigit
FDC9  C80001          LDWX    TRUE,i      ;isNeg <- TRUE
FDCC  EB0004          STWX    isNeg,s
FDCF  C80001          LDWX    sign,i      ;state <- sign
FDD2  EB0002          STWX    state,s
FDD5  12FD92          BR      do
            ;
FDD8  B00030 ifDigit: CPBA    '0',i       ;else if (asciiCh is a digit)
FDDB  16FDF9          BRLT    ifWhite
FDDE  B00039          CPBA    '9',i
FDE1  1EFDF9          BRGT    ifWhite
FDE4  C80000          LDWX    FALSE,i     ;isNeg <- FALSE
FDE7  EB0004          STWX    isNeg,s
FDEA  CB0008          LDWX    valAscii,s  ;total <- value(asciiCh)
FDED  EB000B          STWX    total,s
FDF0  C80002          LDWX    digit,i     ;state <- digit
FDF3  EB0002          STWX    state,s
FDF6  12FD92          BR      do
            ;
FDF9  B00020 ifWhite: CPBA    ' ',i       ;else if (asciiCh is not a space
FDFC  18FD92          BREQ    do
FDFF  B0000A          CPBA    '\n',i      ;or line feed)
FE02  1AFEBE          BRNE    deciErr     ;exit with DECI error
FE05  12FD92          BR      do
            ;                                                    (continues)
```

FIGURE 8.14
The DECI trap handler. (*continued*)

```
FE08  B00030 sSign:   CPBA    '0',i       ;if asciiCh (is not a digit)
FE0B  16FEBE          BRLT    deciErr
FE0E  B00039          CPBA    '9',i
FE11  1EFEBE          BRGT    deciErr     ;exit with DECI error
FE14  CB0008          LDWX    valAscii,s  ;else total <- value(asciiCh)
FE17  EB000B          STWX    total,s
FE1A  C80002          LDWX    digit,i     ;state <- digit
FE1D  EB0002          STWX    state,s
FE20  12FD92          BR      do
                ;
FE23  B00030 sDigit:  CPBA    '0',i       ;if (asciiCh is not a digit)
FE26  16FE74          BRLT    deciNorm
FE29  B00039          CPBA    '9',i
FE2C  1EFE74          BRGT    deciNorm    ;exit normaly
FE2F  C80001          LDWX    TRUE,i      ;else X <- TRUE for later assignments
FE32  C3000B          LDWA    total,s     ;Multiply total by 10 as follows:
FE35  0A              ASLA                ;First, times 2
FE36  20FE3C          BRV     ovfl1       ;If overflow then
FE39  12FE3F          BR      L1
FE3C  EB0006 ovfl1:   STWX    isOvfl,s    ;isOvfl <- TRUE
FE3F  E30000 L1:      STWA    temp,s      ;Save 2 * total in temp
FE42  0A              ASLA                ;Now, 4 * total
FE43  20FE49          BRV     ovfl2       ;If overflow then
FE46  12FE4C          BR      L2
FE49  EB0006 ovfl2:   STWX    isOvfl,s    ;isOvfl <- TRUE
FE4C  0A      L2:     ASLA                ;Now, 8 * total
FE4D  20FE53          BRV     ovfl3       ;If overflow then
FE50  12FE56          BR      L3
FE53  EB0006 ovfl3:   STWX    isOvfl,s    ;isOvfl <- TRUE
FE56  630000 L3:      ADDA    temp,s      ;Finally, 8 * total + 2 * total
FE59  20FE5F          BRV     ovfl4       ;If overflow then
FE5C  12FE62          BR      L4
FE5F  EB0006 ovfl4:   STWX    isOvfl,s    ;isOvfl <- TRUE
FE62  630008 L4:      ADDA    valAscii,s  ;A <- 10 * total + valAscii
FE65  20FE6B          BRV     ovfl5       ;If overflow then
FE68  12FE6E          BR      L5
FE6B  EB0006 ovfl5:   STWX    isOvfl,s    ;isOvfl <- TRUE
FE6E  E3000B L5:      STWA    total,s     ;Update total
FE71  12FD92          BR      do
                ;
```

```
FE74  C30004 deciNorm:LDWA    isNeg,s     ;If isNeg then
FE77  18FE90          BREQ    setNZ
FE7A  C3000B          LDWA    total,s     ;If total != 0x8000 then
FE7D  A08000          CPWA    0x8000,i
FE80  18FE8A          BREQ    L6
FE83  08              NEGA                ;Negate total
FE84  E3000B          STWA    total,s
FE87  12FE90          BR      setNZ
FE8A  C00000 L6:      LDWA    FALSE,i     ;else -32768 is a special case
FE8D  E30006          STWA    isOvfl,s    ;isOvfl <- FALSE
            ;
FE90  DB000F setNZ:   LDBX    oldNZVC,s   ;Set NZ according to total result:
FE93  880001          ANDX    0x0001,i    ;First initialize NZV to 000
FE96  C3000B          LDWA    total,s     ;If total is negative then
FE99  1CFE9F          BRGE    checkZ
FE9C  980008          ORX     0x0008,i    ;set N to 1
FE9F  A00000 checkZ:  CPWA    0,i         ;If total is not zero then
FEA2  1AFEA8          BRNE    setV
FEA5  980004          ORX     0x0004,i    ;set Z to 1
FEA8  C30006 setV:    LDWA    isOvfl,s    ;If not isOvfl then
FEAB  18FEB1          BREQ    storeFl
FEAE  980002          ORX     0x0002,i    ;set V to 1
FEB1  FB000F storeFl: STBX    oldNZVC,s   ;Store the NZVC flags
            ;
FEB4  C3000B exitDeci:LDWA    total,s     ;Put total in memory
FEB7  E2FC13          STWA    opAddr,n
FEBA  50000D          ADDSP   13,i        ;Deallocate locals
FEBD  01              RET                 ;Return to trap handler
            ;
FEBE  D0000A deciErr: LDBA    '\n',i
FEC1  F1FC16          STBA    charOut,d
FEC4  C0FED1          LDWA    deciMsg,i   ;Push address of message onto stack
FEC7  E3FFFE          STWA    -2,s
FECA  580002          SUBSP   2,i
FECD  24FFDE          CALL    prntMsg     ;and print
FED0  00              STOP                ;Fatal error: program terminates
            ;
FED1  455252 deciMsg: .ASCII  "ERROR: Invalid DECI input\x00"
      ...
```

programmer at Level Asmb5 will be able to test for overflow after executing DECI. isOvfl is a Boolean flag that indicates the overflow condition.

FD92 is the start of the FSM loop, identified by the do symbol. ANDA at FD98 masks out all but the rightmost four bits of the input character, which leaves the binary value that corresponds to the decimal ASCII digit. For example, ASCII 5 is represented in binary as 0011 0101. The rightmost four bits are 0101, the corresponding binary value of the decimal digit. The accumulator gets the ASCII character at FD9E and keeps it throughout the loop. stateJT at FDA8 is a jump table for the switch statement in the FSM.

The code from FDAE to FE05 is the case for state having the value sInit, the start state of the FSM. The assignments are all made via the index register instead of the accumulator, because the accumulator maintains the ASCII character for comparison throughout the loop. For example, the assignment of isNeg to FALSE at FDB4 is implemented by LDWX followed by STWX, instead of LDWA followed by STWA.

The code from FE08 to FE20 is the case for state having the value sSign, and from FE23 to FE71 is the case for state having the value sDigit. Pep/9 has no instruction to multiply a value by 10 (dec). This section of code performs the multiply with several left-shift operations. Each ASLA multiplies the value by 2. Three ASLA operations multiply the value by 8, which can be added to the value multiplied by 2 to get the value multiplied by 10. After each ASLA operation and the addition, the routine checks for overflow and sets isOvfl accordingly.

The code from FE74 to FF20 is outside the loop. The algorithm exits the loop under two conditions: normally, or when it has detected an input error. If it exits normally, it checks the isNeg flag to see if the string of digits was preceded by a negative sign. If it was, the instruction at FE83 negates the number by taking the two's complement.

The number 32768 (dec), which is 8000 (hex), must be treated as a special case. If the input is –32768, the FSM will set isOvfl to true when it adds 32760 to 8 at FE62. The problem is that 32768 is out of range, even though –32768 is in range. The routine adjusts isOvfl for this special case at FE8D.

The code from FE90 to FEB1 adjusts the copies of the N, Z, and V flags that were stored at the trap. ANDX at FE93 sets NZV to 000. Note that the mask is 01 (hex), which is 0000 0001 (bin). Because C is the rightmost bit, it remains unchanged by the AND operation. LDWA at FE96 puts the parsed value into the accumulator, setting the current N, Z, and V bits in the CPU accordingly. The code sets the copies of N and Z in the PCB equal to the current values of N and Z in the CPU. It sets the copy of V in the PCB according to the value of isOvfl computed earlier in the parse.

Now that the decimal value has been input and parsed, the trap handler must store it in memory at the location specified by the operand of the DECI that caused the trap. The instructions

```
LDWA total,s
STWA opAddr,n
```

at FEB4 perform the store. LDWA loads the computed value into the accumulator. STWA stores it to opAddr with indirect addressing, for which the operand specifier is the address of the address of the operand. Recall that the address of the operand is computed earlier at FD80 and stored in opAddr. opAddr is itself, therefore, the address of the address of the operand, as required.

The code from FEBE to FED0 executes when the input string cannot be parsed legally. It prints an error message by calling prntMsg, a procedure shown in Figure 8.16 to output a null-terminated string, and terminates the application program immediately.

The DECO Trap Handler

FIGURE 8.15 is the trap handler for the DECO instruction. This routine outputs the operand of DECO in a format that is equivalent to a C printf() function call with an integer value. Because the largest value that can be stored is 32767, the routine will output, at most, five-digit characters. It precedes the value by a negative sign, the ASCII hyphen character, if necessary.

As usual, the statements at the beginning of the trap handler at FEEB assert the legal addressing modes, call the routine to compute the address of the operand, and allocate storage for the local variables. In contrast to the DECI trap handler, the statement at FEFA

```
LDWA opAddr,n
```

accesses the operand with a load instead of a store because DECO is an output statement instead of an input statement. As with the DECI handler, the operand is accessed through opAddr with indirect addressing.

The code from FEFD to FF09 tests for a negative value. If the operand is negative, the load byte and store byte instructions at FF03 output the negative sign, and the following code negates the operand. At FF0A the accumulator contains the magnitude of the operand, which is stored in remain, which stands for *remainder*.

The code from FF0D to FF34 writes the 10,000's, 1000's, 100's, and 10's place of the magnitude of the operand. To suppress any leading zeros, it initializes outYet to false, which indicates that no digit characters have yet been output.

FIGURE 8.15
The DECO trap handler.

```
                ;******* Opcode 0x38
                ;The DECO instruction.
                ;Output format: If the operand is negative, the algorithm prints
                ;a single '-' followed by the magnitude. Otherwise it prints the
                ;magnitude without a leading '+'. It suppresses leading zeros.
                ;
                remain:  .EQUATE 0          ;Remainder of value to output
                outYet:  .EQUATE 2          ;Has a character been output yet?
                place:   .EQUATE 4          ;Place value for division
                ;
FEEB  C000FF opcode38:LDWA    0x00FF,i    ;Assert i, d, n, s, sf, x, sx, sfx
FEEE  E1FC11          STWA    addrMask,d
FEF1  24FC7C          CALL    assertAd
FEF4  24FCCE          CALL    setAddr     ;Set address of trap operand
FEF7  580006          SUBSP   6,i         ;Allocate storage for locals
FEFA  C2FC13          LDWA    opAddr,n    ;A <- oprnd
FEFD  A00000          CPWA    0,i         ;If oprnd is negative then
FF00  1CFF0A          BRGE    printMag
FF03  D8002D          LDBX    '-',i       ;Print leading '-'
FF06  F9FC16          STBX    charOut,d
FF09  08              NEGA                ;Make magnitude positive
FF0A  E30000 printMag:STWA    remain,s    ;remain <- abs(oprnd)
FF0D  C00000          LDWA    FALSE,i     ;Initialize outYet <- FALSE
FF10  E30002          STWA    outYet,s
FF13  C02710          LDWA    10000,i     ;place <- 10,000
FF16  E30004          STWA    place,s
FF19  24FF44          CALL    divide      ;Write 10,000's place
FF1C  C003E8          LDWA    1000,i      ;place <- 1,000
FF1F  E30004          STWA    place,s
FF22  24FF44          CALL    divide      ;Write 1000's place
FF25  C00064          LDWA    100,i       ;place <- 100
FF28  E30004          STWA    place,s
FF2B  24FF44          CALL    divide      ;Write 100's place
FF2E  C0000A          LDWA    10,i        ;place <- 10
FF31  E30004          STWA    place,s
FF34  24FF44          CALL    divide      ;Write 10's place
FF37  C30000          LDWA    remain,s    ;Always write 1's place
```

```
FF3A  900030          ORA     0x0030,i    ;Convert decimal to ASCII
FF3D  F1FC16          STBA    charOut,d   ;  and output it
FF40  500006          ADDSP   6,i         ;Dallocate storage for locals
FF43  01              RET
            ;
            ;Subroutine to print the most significant decimal digit of the
            ;remainder. It assumes that place (place2 here) contains the
            ;decimal place value. It updates the remainder.
            ;
            remain2: .EQUATE 2          ;Stack addresses while executing a
            outYet2: .EQUATE 4          ;  subroutine are greater by two because
            place2:  .EQUATE 6          ;  the retAddr is on the stack
            ;
FF44  C30002 divide:  LDWA    remain2,s   ;A <- remainder
FF47  C80000          LDWX    0,i         ;X <- 0
FF4A  730006 divLoop: SUBA    place2,s    ;Division by repeated subtraction
FF4D  16FF59          BRLT    writeNum    ;If remainder is negative then done
FF50  680001          ADDX    1,i         ;X <- X + 1
FF53  E30002          STWA    remain2,s   ;Store the new remainder
FF56  12FF4A          BR      divLoop
            ;
FF59  A80000 writeNum:CPWX    0,i         ;If X != 0 then
FF5C  18FF68          BREQ    checkOut
FF5F  C00001          LDWA    TRUE,i      ;outYet <- TRUE
FF62  E30004          STWA    outYet2,s
FF65  12FF6F          BR      printDgt    ;and branch to print this digit
FF68  C30004 checkOut:LDWA    outYet2,s   ;else if a previous char was output
FF6B  1AFF6F          BRNE    printDgt    ;then branch to print this zero
FF6E  01              RET                 ;else return to calling routine
            ;
FF6F  980030 printDgt:ORX     0x0030,i    ;Convert decimal to ASCII
FF72  F9FC16          STBX    charOut,d   ;  and output it
FF75  01              RET                 ;return to calling routine
```

Subroutine `divide` outputs the digit character for the place value in `place` and decreases `remain` for the next call. For example, if `remain` is 24873 before the call to `divide` at FF19, then `divide` will output 2 and leave 4873 in `remain`. It will also set `outYet` to true.

Before outputting character 0, `divide` tests `outYet` to check whether any digit characters have been output yet. If `outYet` is false, the character

is a leading zero and is not output. Otherwise, it is an embedded zero and is output. For example, if `remain` is 761 before the call at FF22, `divide` prints nothing and leaves 761 in `remain` and false in `outYet`. The code beginning at FF37 writes the 1's place regardless of the value of `outYet`. Thus, a value of zero for the original operand gets output as 0.

The code from FF44 to FF75 is the subroutine to print the most significant digit of `remain`. It determines the value to output by repeatedly subtracting `place` from `remain`, counting the number of subtractions until `remain` is less than zero. The effect is to compute the value to output as `remain / place`.

The HEXO and STRO Trap Handlers and Operating System Vectors

FIGURE 8.16 is the trap handler for the `HEXO` and `STRO` instructions. They are similar in function to the `DECO` trap handler. Because `STRO` is an output instruction, the address of the operand is first fetched with

```
LDWA opAddr,d
```

at FFCE. Then it calls the `prntMsg` subroutine at FFD7, pushing the address of the string to print on the run-time stack. In effect, a string is an array of characters, so the processing is similar to the translations of a C program where an array is passed as a parameter. The print subroutine, therefore, uses stack-deferred indexed addressing in the statements

```
LDBA msgAddr,sfx
```

to access an element of the character array.

The machine vectors are established with the `.ADDRSS` assembler directive. Compare this code with the code in Figure 8.1 and Figure 8.2. The vector at FFF4 is the address of `osRAM`, which is the top byte of operating system RAM. The hardware initializes SP to this value when the user selects the execute option from the simulator. It is the byte at the bottom of the user stack.

The vector at FFF6 is the address of `wordTemp`, which is the first system global variable. Figure 8.2 shows `wordTemp` as the next byte below the 128-byte block of storage reserved for the system stack. The hardware initializes SP to this value when the user selects the load option from the simulator. It also pushes the PCB onto the stack starting from this point when a trap instruction executes.

The next two vectors at FFF8 and FFFA are the addresses of the input device defined by the symbol `charIn` and the output device defined by the symbol `charOut`. During translation time, the application assembler includes these symbols automatically in its symbol table. During run time,

FIGURE 8.16
The trap handlers for the HEXO and STRO instructions.

```
            ;******* Opcode 0x40
            ;The HEXO instruction.
            ;Outputs one word as four hex characters from memory.
            ;
FF76  C000FF opcode40:LDWA    0x00FF,i    ;Assert i, d, n, s, sf, x, sx, sfx
FF79  E1FC11          STWA    addrMask,d
FF7C  24FC7C          CALL    assertAd
FF7F  24FCCE          CALL    setAddr     ;Set address of trap operand
FF82  C2FC13          LDWA    opAddr,n    ;A <- oprnd
FF85  E1FC0F          STWA    wordTemp,d  ;Save oprnd in wordTemp
FF88  D1FC0F          LDBA    wordTemp,d  ;Put high-order byte in low-order A
FF8B  0C              ASRA                ;Shift right four bits
FF8C  0C              ASRA
FF8D  0C              ASRA
FF8E  0C              ASRA
FF8F  24FFA9          CALL    hexOut      ;Output first hex character
FF92  D1FC0F          LDBA    wordTemp,d  ;Put high-order byte in low-order A
FF95  24FFA9          CALL    hexOut      ;Output second hex character
FF98  D1FC10          LDBA    byteTemp,d  ;Put low-order byte in low order A
FF9B  0C              ASRA                ;Shift right four bits
FF9C  0C              ASRA
FF9D  0C              ASRA
FF9E  0C              ASRA
FF9F  24FFA9          CALL    hexOut      ;Output third hex character
FFA2  D1FC10          LDBA    byteTemp,d  ;Put low-order byte in low order A
FFA5  24FFA9          CALL    hexOut      ;Output fourth hex character
FFA8  01              RET
            ;
            ;Subroutine to output in hex the least significant nybble of the
            ;accumulator.
            ;
FFA9  80000F hexOut:  ANDA    0x000F,i    ;Isolate the digit value
FFAC  B00009          CPBA    9,i         ;If it is not in 0..9 then
FFAF  14FFBB          BRLE    prepNum
FFB2  700009          SUBA    9,i         ;  convert to ASCII letter
FFB5  900040          ORA     0x0040,i    ;  and prefix ASCII code for letter
FFB8  12FFBE          BR      writeHex
FFBB  900030 prepNum: ORA     0x0030,i    ;else prefix ASCII code for number
FFBE  F1FC16 writeHex:STBA    charOut,d   ;Output nybble as hex
FFC1  01              RET
            ;
```

(*continues*)

FIGURE 8.16
The trap handlers for the HEXO and STRO instructions. (*continued*)

```
                ;******* Opcode 0x48
                ;The STRO instruction.
                ;Outputs a null-terminated string from memory.
                ;
FFC2  C0003E opcode48:LDWA    0x003E,i    ;Assert d, n, s, sf, x
FFC5  E1FC11          STWA    addrMask,d
FFC8  24FC7C          CALL    assertAd
FFCB  24FCCE          CALL    setAddr     ;Set address of trap operand
FFCE  C1FC13          LDWA    opAddr,d    ;Push address of string to print
FFD1  E3FFFE          STWA    -2,s
FFD4  580002          SUBSP   2,i
FFD7  24FFDE          CALL    prntMsg     ;and print
FFDA  500002          ADDSP   2,i
FFDD  01              RET
                ;
                ;******* Print subroutine
                ;Prints a string of ASCII bytes until it encounters a null
                ;byte (eight zero bits). Assumes one parameter, which
                ;contains the address of the message.
                ;
                msgAddr: .EQUATE 2          ;Address of message to print
                ;
FFDE  C80000 prntMsg: LDWX    0,i         ;X <- 0
FFE1  C00000          LDWA    0,i         ;A <- 0
FFE4  D70002 prntMore:LDBA    msgAddr,sfx ;Test next char
FFE7  18FFF3          BREQ    exitPrnt    ;If null then exit
FFEA  F1FC16          STBA    charOut,d   ;else print
FFED  680001          ADDX    1,i         ;X <- X + 1 for next character
FFF0  12FFE4          BR      prntMore
                ;
FFF3  01      exitPrnt:RET
                ;
                ;******* Vectors for system memory map
FFF4  FB8F            .ADDRSS osRAM       ;User stack pointer
FFF6  FC0F            .ADDRSS wordTemp    ;System stack pointer
FFF8  FC15            .ADDRSS charIn      ;Memory-mapped input device
FFFA  FC16            .ADDRSS charOut     ;Memory-mapped output device
FFFC  FC17            .ADDRSS loader      ;Loader program counter
FFFE  FC52            .ADDRSS trap        ;Trap program counter
```

the hardware simulator uses these vectors to know where the I/O devices are mapped into memory.

The vector at FFFC is the address of the loader, as Figure 8.3 shows. The hardware initializes PC to this value when the user selects the load option. The vector at FFFE is the address of the interrupt handler entry point, as Figure 8.6 shows. The hardware initializes PC to this value when a trap instruction executes.

8.3 Concurrent Processes

Remember that a process is a program during execution. Section 8.2 shows how the operating system can suspend a process during its execution to provide a service. A time line of CPU activity for a process that uses `DECI` and `DECO` would look like FIGURE 8.17. The shaded regions represent those times that the CPU executes the trap service routine. The time line is similar in form to Figure 8.4 except that this figure shows the operating system suspending a process before it terminates and then restarting it when the service is complete.

The traps described in Section 8.2 are called *software interrupts* because the executing process initiates them by the unimplemented opcodes in its listing. They are also called *synchronous interrupts*, because each time the process executes, the interrupts occur at the same time. The interrupts are synchronized with the code and are predictable.

Supervisor calls

Another way to initiate a synchronous interrupt is to execute an operating system call. A common assembly-level mnemonic for an operating system call is `SVC`, which stands for *supervisor call*. The operand specifier normally acts as a parameter for the system call and tells the system which service the program wants to request. For example, if you want to flush the contents of a buffered stream with the equivalent of `fflush(stdin)` in C and the code for `fflush()` is 27, you might execute

```
SVC 27, i
```

FIGURE 8.17
A time line of CPU usage when the operating system executes a single program containing `DECI` and `DECO` instructions.

Asynchronous Interrupts

Another type of interrupt is the *asynchronous interrupt*, which does not occur at a predictable time during execution. Two common sources of asynchronous interrupts are

Two common asynchronous interrupts

- Time outs
- I/O completions

To see how asynchronous interrupts can occur from time outs, consider a multi-user system, which allows several users to access the computer simultaneously. If the computer has only one CPU, the operating system must allocate the CPU to each user's job in turn with a technique known as *time sharing*. The operating system allocates a quantum of time called a *time slice*, typically about 100 ms (one-tenth of a second), to a job. If the job is not completed within that time (a condition known as *time out*), the operating system suspends the job temporarily and allocates another quantum of CPU time to the next job.

Time outs

To implement time sharing, the hardware must provide an alarm clock that the operating system can set to produce an interrupt after an interval of time. The reason such an interrupt is unpredictable is that it depends on how busy the system is servicing the requests of its users. If no other job is waiting for the CPU, the system may let your job run longer than the standard time slice. Then if another user suddenly requests a service, the operating system may suspend your process immediately and allocate the CPU to the requesting job. Your process would not be interrupted at the same point as if it timed out after one time slice.

Even if the computer has more than one CPU, asynchronous interrupts occur the same way. The operating system allocates a separate time slice for each CPU in the system and manages the time outs for each CPU.

I/O completions

The second common source of asynchronous interrupts is I/O completions. A basic property of I/O devices is their slow speed compared to the processing speed of the CPU. If a running process requests some input from a keyboard, in the fraction of a second that it takes the user to respond, the CPU can execute hundreds of thousands of instructions for another process. Even if the process requests input from a disk file, which is much faster than keyboard input, the CPU could still execute thousands of instructions while waiting for the information to come from the disk.

To keep from wasting CPU time, the operating system can suspend the process that makes an I/O request if it appears that the process will need to wait for the I/O to complete. It can temporarily assign the CPU to a second process with the understanding that when the I/O does complete, the first process may immediately get the CPU back. Because the second process

cannot predict when the I/O device will complete the I/O operation for the first process, it cannot know when the operating system might interrupt it to give the CPU back to the first process.

An operating system with one CPU that can switch back and forth between processes to keep the CPU busy is called a *multiprogramming system*. To implement multiprogramming, the hardware must provide connections for the I/O devices to send interrupt signals to the CPU when the devices complete their I/O operations.

Multiprogramming

Processes in the Operating System

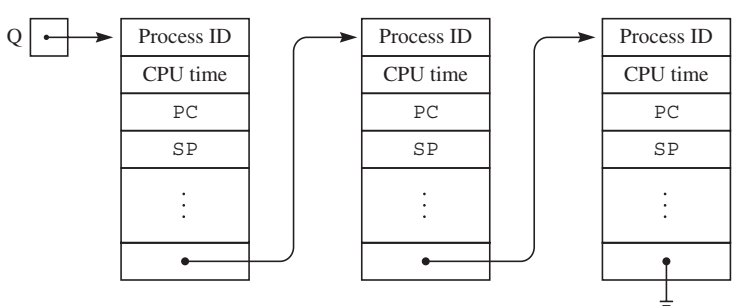
One purpose of an operating system is to allocate the resources of the system efficiently. A multiprogramming time-sharing system allocates CPU time among the jobs in the system. The objective is to keep the CPU as busy as possible executing user jobs instead of being idle waiting for I/O. The operating system tries to be fair in scheduling CPU time so that all the jobs will be completed in a reasonable time.

At any given time, the operating system must maintain many suspended processes that are waiting their turn for CPU time. It maintains all these processes by allocating a separate PCB for each one, similar to the PCB the interrupt handler maintains in the Pep/9 system. A common practice is to link the PCBs together with pointers in a linked list called a *queue*. FIGURE 8.18 shows a queue of PCBs.

Each PCB includes copies of all the CPU register values at the time of the process's most recent interrupt. The register set must include a copy of the program counter so the process can continue executing from where it was when the interrupt occurred.

The PCB contains additional information to help the operating system schedule the CPU. An example is a unique process identification number

FIGURE 8.18
A queue of process control blocks.



assigned by the system, labeled *Process ID* in Figure 8.18, that serves to reference the process. Suppose a user wants to terminate a process before it completes execution normally, and he knows the ID number is 782. He could issue a `KILL(782)` command that would cause the operating system to search through the queue of PCBs, find the PCB with ID 782, remove it from the queue, and deallocate it.

Another example of information stored in the PCB is a record of the total amount of CPU time used so far by the suspended process. If the CPU becomes available and the operating system must decide which of several suspended processes gets the CPU, it can use the recorded time to make a fair decision.

As a job progresses through the system toward completion, it passes through several states, as FIGURE 8.19 shows. The figure is in the form of a state transition diagram and is another example of an FSM. Each transition is labeled with the event that causes the change of state.

Entering the start state

When a user submits a job for processing, the operating system creates a process for it by allocating a new PCB and attaching it to a queue of processes that are waiting for CPU time. It loads the program into main memory and sets the copy of `PC` in the PCB to the address of the first instruction of the process. That puts the job in the ready state.

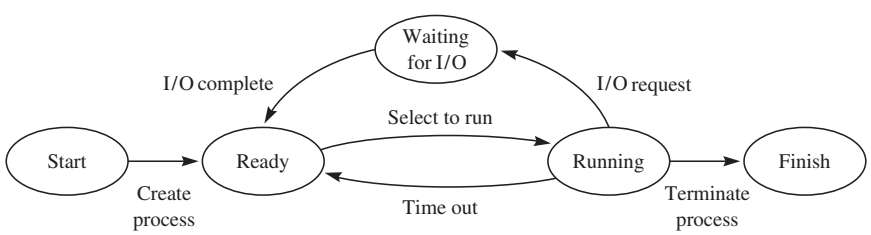
Transition from the start state

Eventually, the operating system should select the job to receive some processing time. It sets the alarm clock to generate an interrupt after a quantum of time and puts the copies of the registers from the PCB into the CPU. That puts the job in the running state.

Transitions from the running state

While in the running state, three things can happen: (1) The running process may time out if it is still executing when the alarm clock interrupts. If so, the operating system attaches the process's PCB to the ready queue, which puts it back in the ready state. (2) The process may complete its execution normally, in which case the last instruction it executes is an `SVC` to request that the operating system terminate it. (3) The process may need some input, in which case it executes an `SVC` for the request. The operating

FIGURE 8.19
The state transition diagram for a job in an operating system.



system would transfer the request to the appropriate I/O device and put the PCB in another queue of processes that are waiting for their I/O operations to complete. That puts the process in the waiting-for-I/O state.

Transition from the waiting-for-I/O state

While the process is in the waiting-for-I/O state, the I/O device should eventually interrupt the system with the requested input. When that happens, the system puts the input in a buffer in main memory, removes the process's PCB from the waiting-for-I/O queue, and puts it in the ready queue. That puts the process in the ready state, from which it will eventually receive more CPU time. Then it can access the input from the buffer.

Multiprocessing

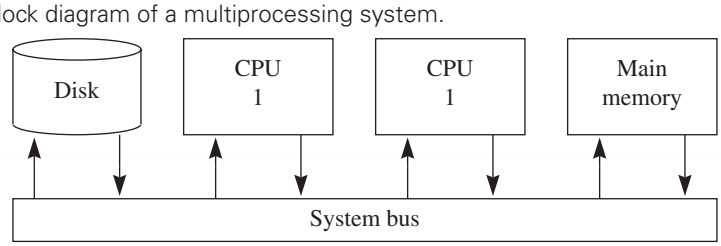
As far as the user is concerned, a job simply executes from start to finish. The interruptions are invisible in the same way that the `DECI` interrupt is invisible to the assembly language programmer at Level Asmb5. The details at the operating system level are invisible to the users at a higher level of abstraction.

The only perceptible difference to the user is that it will take longer for the program to execute if many jobs are in the system. One way to speed the progress is to attach more than one CPU to the system. Each core in a multicore chip is a separate CPU. Such a configuration is called a *multiprocessing system*. FIGURE 8.20 shows a multiprocessing system with two processors.

In multiprogramming, the processes appear to be executing concurrently because the CPU switches between them so rapidly. In multiprocessing, the operating system can schedule more than one process to execute concurrently because there is more than one processor.

It would be nice if increasing the number of processors in the system increased the performance proportionally. Unfortunately, that is not usually the case. When you add more processors to the system, you place a greater demand on the communication links of the system. For example, if you attach the processors to a common bus, as in Figure 8.20, the bus may limit the performance of the system. If both CPUs request a read from the input device at the same time, one CPU will have to wait. The more processors you add, the more frequently those conflicts occur.

FIGURE 8.20
Block diagram of a multiprocessing system.



The communication overhead inherent in a multiprocessor system typically yields a performance curve as in FIGURE 8.21. The dashed line shows the theoretical maximum benefit of adding processors. On the dashed line, for example, if you double the number of processors, you double the performance. In practice, the performance does not increase that much.

A Concurrent Processing Program

The processes considered thus far have all been independent of each other. Each process belongs to a different user, and there is no interaction between the processes. Under those circumstances, the result of a computation does not depend on when an interrupt occurs. The only effect the interrupt has is to increase the amount of time it takes to execute the process.

Cooperating processes

In practice, processes managed by an operating system frequently need to cooperate with one another to perform their tasks. The program in FIGURE 8.22 describes a situation in which two processes must cooperate to avoid producing incorrect results.

Suppose the operating system must manage an airline's database, with records accessed concurrently by several users. Each flight has a record in the database that contains, among other things, the number of reservations that have been made for that flight. Travel agencies from throughout the city access the system on behalf of prospective passengers. Requests for information from the database are somewhat random because it is impossible to predict when a given agent will need to access the system.

FIGURE 8.21
The increase in performance by adding processors in a multiprocessing system.

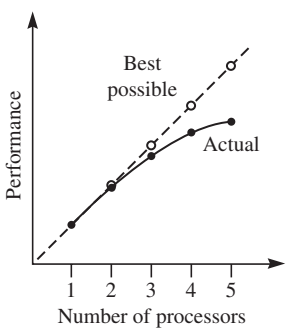


FIGURE 8.22
Concurrent processes at two levels of abstraction.

C Level

Process P1
```
...
numRes++
...
```

Process P2
```
...
numRes++
...
```

Assembly Level

Process P1
```
...
LDWA numRes,d
ADDA 1,i
STWA numRes,d
...
```

Process P2
```
...
LDWA numRes,d
ADDA 1,i
STWA numRes,d
...
```

One day, two different agents have customers who want to make reservations at exactly the same time for the same future flight. The operating system creates a process for each job called *P1* and *P2*. Figure 8.22 shows a code fragment from each process. `numRes` stands for *number of reservations*. It is an integer variable whose value is in main memory while P1 and P2 progress through the system.

Suppose `numRes` has the value 47 before either agent makes a reservation for her customer. After both transactions, `numRes` should have the value 49. At the C level, each process wants to increment `numRes` by 1 with the assignment statement `numRes++`. If assignment statements are *atomic*—that is, indivisible—then the code fragment at the C level will produce correct results regardless of which process executes its assignment statement first. If P1 executes first, it will make `numRes` 48, and P2 will make `numRes` 49. If P2 executes first, it will make `numRes` 48, and P1 will make `numRes` 49. In either case, `numRes` gets the correct value of 49.

Nonatomic statements

The problem is that assignment statements at the C level are not atomic. They are compiled to `LDWA`, `ADDA`, and `STWA`, and are executed in a system in which an interrupt may occur between any assembly language statements. FIGURE 8.23 is a trace of an execution sequence that shows what can go wrong. A(P1) is the content of P1's accumulator, either in the CPU when P1 is running or in the PCB when P1 is suspended. A(P2) is P2's accumulator.

In this sequence, P1 executes `LDWA`, which puts 47 in its accumulator, then `ADDA`, which increments the accumulator to 48. Then the operating system interrupts P1 and gives P2 some processor time. P2 executes all three of its statements, changing `numRes` in memory to 48. When P1 eventually resumes, it gives 48 to `numRes` as well. The net result is that `numRes` has the value 48 instead of 49, even though each process executed all its statements.

Statement Executed	A(P1)	A(P2)	numRes
	?	?	47
(P1) `LDWA numRes,d`	47	?	47
(P1) `ADDA 1,i`	48	?	47
(P2) `LDWA numRes,d`	48	47	47
(P2) `ADDA 1,i`	48	48	47
(P2) `STWA numRes,d`	48	48	48
(P1) `STWA numRes,d`	48	48	48

FIGURE 8.23
A trace of one possible execution sequence of Figure 8.22.

The logical equivalence of multiprogramming and multiprocessing

This problem can occur whether the processes execute in a multiprocessing system with true concurrency or in a multiprogramming system where the concurrency is only apparent. In a multiprocessing system, it would be possible for P1 and P2 to execute their `ADDA` instructions at exactly the same time. But if they tried to execute their `STWA` instructions at exactly the same time, the hardware would force one process to wait while the other wrote the value to memory. From a logical point of view, the problems that can occur are the same whether the concurrency is real or not.

Critical Sections

The basic problem arises because P1 and P2 share the part of main memory that contains the value of `numRes`. Whenever concurrent processes share a variable, there is always the possibility that the results depend on the timing of the interrupts. To solve the problem, we need a way to ensure that when one process accesses the shared variable, the other process is prevented from accessing it until the first process has completed its access.

Critical sections

Sections of code in two processes that are mutually exclusive are called *critical sections*. For concurrent programs to execute correctly, the software must guarantee that if one process is executing a statement in a critical section, the other process cannot be executing a statement in its critical section. To solve the problem of Figure 8.22, we need a way of putting the assignment statements in critical sections so that the interleaved execution will not occur at the assembly level.

Entry and exit sections

A critical section requires two additional pieces of code called the *entry section* and the *exit section*. The entry section for P1 is written just before its critical section. Its function is to test whether P2 is executing in its critical section and, if so, to delay somehow the execution of P1's critical section until P2 is finished with its critical section. The exit section for P1 is written just after its critical section. Its function is to alert P2 that P1 is no longer in a critical section so P2 may enter its critical section.

The code fragment at the C level for each process of Figure 8.22 must be modified as follows:

```
remainder section
entry section
numRes++ //the critical section
exit section
remainder section
```

The remainder sections are all those parts of the code that can execute concurrently with the other process with no ill effects. The critical sections are those parts of the code that must be mutually exclusive.

FIGURE 8.24
The general form of critical section programs.

```
Process P1                    Process P2
do                            do
    entry section                 entry section
    critical section              critical section
    exit section                  exit section
    remainder section             remainder section
while (! done1);              while (! done2);
```

The following programs show attempts to implement the entry and exit sections that guard the process's access to its critical section. Each program assumes that P1 and P2 have the general form of FIGURE 8.24.

`done1` and `done2` are local Boolean variables (not shared) that are modified somewhere in the remainder sections.

A First Attempt at Mutual Exclusion

The program in FIGURE 8.25, our first attempt at designing the entry and exit sections, uses `turn`, a shared integer variable. The entry section consists of a `do` loop that tests `turn`, and the exit section consists of an assignment statement that modifies `turn`. Although the listing does not show it, assume that `turn` is initialized either to 1 or 2 before the processes enter the `do` loops.

FIGURE 8.25
An attempt at programming mutual exclusion.

```
Process P1                        Process P2
do                                do
   while (turn != 1)                 while (turn != 2)
      ; //nothing                       ; //nothing
   critical section                  critical section
   turn = 2;                         turn = 1;
   remainder section                 remainder section
while (!done1);                   while (!done2);
```

The body of the do loop in the entry section is an empty C statement that generates no code at the assembly level. The code for the entry section of P1 translates to

The nonatomic nature of the do *statement*

```
Loop: LDWA turn,d
      CPWA 1,i
      BRNE Loop
```

Suppose turn is initialized to 1 and both processes try to enter their critical sections at the same time. No matter how you interleave the executions of the assembly statements in the entry section, P2 will continually loop until P1 enters its critical section. When P1 finishes its critical section, its exit section will set turn to 2, after which P2 will be able to enter its critical section.

This algorithm guarantees that the critical sections are mutually exclusive. P2 can be in its critical section only if turn is 2, during which time P1 cannot be in its critical section, and vice versa. When P2 leaves its critical section, it sets turn to 1, which acts as a signal to P1 that it may enter its critical section.

Although the algorithm guarantees mutual exclusion, it has the undesirable property of requiring the processes to strictly alternate their do loops. The processes communicate through the shared variable turn, which keeps track of whose turn it is to execute a critical section. The user may want P1 to execute its do loop several times without P2 executing its loop at all. That could never happen with these entry and exit sections.

A Second Attempt at Mutual Exclusions

To allow a process to execute its do loops unrestrained by the execution of the other process (except for the mutual exclusion requirement), the program in FIGURE 8.26 uses two shared Boolean variables, enter1 and enter2. Assume that enter1 and enter2 are both initialized to false.

If P2 is in its remainder section, enter2 must be false. Then P1 can execute its do loop as often as it likes. It simply sets enter1 to true, tests enter2 once in the while loop, executes its critical section, sets enter1 to false, and executes its remainder section. It can repeat the sequence as long as it likes. Similarly, P2 can loop repeatedly if P1 is in its remainder section.

This implementation guarantees mutual exclusion. When P1 sets enter1 to true, it is signaling P2 that it is trying to enter a critical section. If P2 has just a little earlier fetched enter1 with

```
LDWA enter1,d
```

in its while test, P2 will not immediately know of P1's intentions. P2 may be executing its critical section already. However, if P2 is in its critical section, enter2 must be true, and P1's while loop will keep it from entering its

FIGURE 8.26
Another attempt at programming mutual exclusion.

```
Process P1                      Process P2
do                              do
    enter1 = TRUE;                  enter2 = TRUE;
    while (enter2)                  while (enter1)
        ; //nothing                     ; //nothing
    critical section                critical section
    enter1 = FALSE;                 enter2 = FALSE;
    remainder section               remainder section
while (!done1);                 while (!done2);
```

critical section at the same time. When P2 finally exits, it sets `enter2` to false, which allows P1 into its critical section.

The problems that confront the designer of cooperating processes can be quite subtle and unexpected. This algorithm is a case in point. Although it guarantees mutual exclusion and does not constrain the `do` loop execution as in the previous program, it nevertheless has a serious bug.

FIGURE 8.27 shows a trace where P1 sets `enter1` to true and then experiences an interrupt. P2 sets `enter2` to true and then begins executing its `while` loop. The `while` loop will continue executing until P2 times out and P1 resumes, because `enter1` is true. But P1 will also loop indefinitely because `enter2` is true.

P1 and P2 are in a state in which each one wants to enter a critical section. P1 cannot enter until P2 enters, executes its critical section, and sets

FIGURE 8.27
A trace of the program in Figure 8.26 that produces deadlock.

Statement Executed	enter1	enter2
	false	false
(P1) `enter1 = TRUE;`	true	false
(P2) `enter2 = TRUE;`	true	true
(P2) `while (enter1);`	true	true
(P1) `while (enter2);`	true	true

Definition of deadlock

`enter2` to false. But P2 cannot enter until P1 enters, executes its critical section, and sets `enter1` to false. Each process is waiting for an event that will never occur, a condition called *deadlock*. Deadlocks, like endless loops, are conditions to avoid.

Peterson's Algorithm for Mutual Exclusion

We need a solution that guarantees mutual exclusion, allows the outer `do` loops of each process to execute without restraint, and avoids deadlock. FIGURE 8.28, an implementation of Peterson's algorithm, combines features from Figures 8.25 and 8.26 to achieve all these objectives. The basic idea is that `enter1` and `enter2` provide the mutual exclusion as in Figure 8.26, and `turn` allows one of the processes to enter its critical section even if both processes try to enter at the same time. `enter1` and `enter2` initially are false, and `turn` initially can be 1 or 2.

Proof that Peterson's algorithm guarantees mutual exclusion

To see that mutual exclusion is guaranteed, consider the situation if P1 and P2 were both executing their critical sections simultaneously. `enter1` and `enter2` would both be true. In P1, the `while` test would imply that `turn` has the value 1 because `enter2` is true. But in P2, the `while` test would imply that `turn` has the value 2 because `enter1` is true. This contradiction implies that P1 and P2 cannot execute their critical sections simultaneously.

But what if P1 and P2 try to enter their critical sections at about the same time? Is there some interleaving of the executions in the entry section that will permit them to both execute their critical sections simultaneously? No there is not, even though the `while` test with the AND operation is not atomic at the assembly level. There are two ways that P1 can get past the

FIGURE 8.28
Peterson's algorithm for mutual exclusion.

Process P1

```
do
   enter1 = TRUE;
   turn = 2;
   while (enter2 && (turn == 2))
      ; //nothing
   critical section
   enter1 = FALSE;
   remainder section
while (!done1);
```

Process P2

```
do
   enter2 = TRUE;
   turn = 1;
   while (enter1 && (turn == 1))
      ; //nothing
   critical section
   enter2 = FALSE;
   remainder section
while (!done2);
```

while test into its critical section: if enter2 is false, or if turn is 1. If either of these conditions holds, P1 can enter regardless of the other condition.

Suppose that P1 gets past the while test because when it gets the value of enter2 with

```
LDWA enter2,d
```

enter2 has the value false. That can happen only when P2 is in its remainder section. Even if P1 is interrupted after it loads the value of enter2, and P2 then sets enter2 to true and turn to 1, P2 will not be able to enter its critical section because P1 has set enter1 to true and turn is now 1.

Suppose that P1 gets past the while test because when it gets the value of turn with

```
LDWA turn,d
```

turn has the value 1. Because the previous instruction in P1 set turn to 2, that can happen only if P1 was interrupted between its previous instruction and the while test, and P2 set turn to 1. But then, P2 again will be prevented from getting past its while loop into its critical section, because P1 has set enter1 to true and turn now has the value 1.

Proof that Peterson's algorithm avoids deadlock

To see that deadlock cannot occur, assume that both processes are deadlocked, both executing their while loops concurrently (in a multiprocessing system) or during alternate time slices (in a multiprogramming system). The while test in P1 implies that turn must have the value 2, but the test in P2 implies that turn must have the value 1. This contradiction shows that both processes cannot be looping together.

Suppose both processes try to set turn at the same time with

```
STWA turn,d
```

In a multiprogramming system, P1's assignment to turn will occur either before or after P2's assignment because they must execute in different time slices. In a multiprocessing system, if both processes try to store a value to turn in main memory at exactly the same time, the hardware will force one of the processes to wait while the other executes its STWA. In either system, the process that stores to turn first will enter its critical section and deadlock will not occur.

Semaphores

Although the program in Figure 8.28 solves the critical section problem while avoiding deadlock, it does have an undesirable inefficiency. The mechanism that prevents a process from entering its critical section is a while loop. The loop's only purpose is to stall the process until it is interrupted, allowing

Spin locks

time for the other process to finish executing its critical section. Such a loop is called a *spin lock* because the process is locked out of its critical section by spinning around the loop.

Spin locks are a waste of CPU time, especially if the process is executing in a multiprogramming system and has just been allocated a new time slice. It would be more efficient if the CPU were allocated to another process that could use the time to perform useful work. *Semaphores* are shared variables that most operating systems provide for concurrent programming. They enable the programmer to implement critical sections without spin locks.

A semaphore is an integer variable whose value can be modified only by an operating system call. The three operations on semaphore `s` are

The three operations on a semaphore

- `init(s)`
- `wait(s)`
- `signal(s)`

where `init()`, `wait()`, and `signal()` are procedures provided by the operating system. At the assembly level, the procedures would be invoked by an `SVC` with the appropriate operand specifier. A semaphore is another example of an abstract data type (ADT) with operations whose meanings are known to the programmer but whose implementations are hidden at a lower level of abstraction. (`wait(s)` and `signal(s)` are frequently written `p(s)` and `v(s)`, respectively.)

Each semaphore, `s`, has associated with it a queue of process control blocks, called `sQueue`, that represents suspended processes. The meanings of the operations are

```
init(s)
s = 1;
sQueue = an empty list of process control blocks

wait(s)
s--
if (s < 0)
        Suspend this process by adding it to sQueue

signal(s)
s++
if (s ≤ 0)
        Transfer a process from sQueue to the ready queue
```

An important characteristic of each operation is that the operating system guarantees them to be atomic. For example, it is impossible for two processes

to execute `signal(s)` simultaneously with `s` incremented only by 1 as `numRes` is in Figure 8.22. The assembly-level statements for the assignments will never be interleaved.

FIGURE 8.29 is the state transition diagram for a job in an operating system that provides semaphores. A process in the waiting-for-`s` state is suspended, its PCB in `sQueue`, in the same way that a process in the ready state is suspended, its PCB in the ready queue. Such a process is blocked from running, because it must make a transition to the ready state before it can run.

If a running process executes `wait(s)` when `s` is greater than 0, then `wait(s)` simply decrements `s` by 1 and the process continues executing. A running process makes a transition to the waiting-for-`s` state by executing `wait(s)` when `s` is less than or equal to 0. If a running process executes `signal(s)` when `s` is greater than or equal to 0, it simply increments `s` by 1 and continues executing. A running process that executes `signal(s)` when `s` is less than 0 causes some other process that is waiting for `s` to be selected by the operating system and placed in the ready state. The process that executed `signal(s)` continues to run.

The meaning of a negative semaphore value

From the definitions of `wait()` and `signal()`, it follows that a negative value of `s` means that one or more processes are blocked in `sQueue`. Furthermore, the magnitude of `s` is the number of processes blocked. For example, if the value of `s` is –3, then three processes are blocked in `sQueue`.

If more than one process is blocked when `signal(s)` executes, the operating system tries to be fair in selecting the process to transfer to the ready state. A common strategy is to use *first-in, first-out* (FIFO) scheduling so the process that was blocked for the longest period of time gets sent to the ready state. FIFO is the characteristic that distinguishes a queue from a stack, which is a *last-in, first-out* (LIFO) list.

Figure 8.29 shows only one semaphore wait state. In a system that provides semaphores, the programmer can declare as many different

FIGURE 8.29
The state transition diagram for a job in an operating system that provides a semaphore.

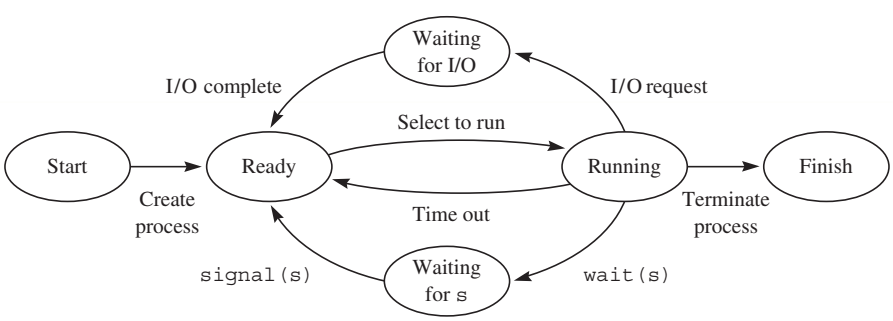


FIGURE 8.30
Critical sections with semaphore.

```
Process P1                          Process P2
do                                  do
    wait(mutEx);                        wait(mutEx);
    critical section                    critical section
    signal(mutEx);                      signal(mutEx);
    remainder section                   remainder section
while (!done1);                     while (!done2);
```

semaphores as she likes. The operating system will maintain a queue of blocked processes for each one.

Critical Sections with Semaphores

Critical sections are trivial to program if the operating system provides semaphores. The program in FIGURE 8.30 assumes that `mutEx` is a semaphore initialized to 1 with `init(mutEx)`.

The first process to execute `wait(mutEx)` will change `mutEx` from 1 to 0 and enter its critical section. If the other process executes `wait(mutEx)` in the meantime, it will change `mutEx` from 0 to –1, and the operating system will immediately block it. When the first process eventually leaves its critical section, it will execute `signal(mutEx)`, which will put the other process in the ready state.

Because the operating system guarantees that `wait()` and `signal()` are atomic, the programmer need not worry about interleaving within the entry and exit sections. Also, time is not wasted on spin locks because the system immediately puts the second process on the wait queue for `mutEx`. Of course, hiding the details does not eliminate them. The operating system designer must use the features of the hardware, along with algorithmic reasoning such as that employed in the previous programs, to provide the semaphores. Semaphores satisfy both goals of the operating system—to provide a convenient environment for higher-level programming and to allocate the resources of the system efficiently.

8.4 Deadlocks

The program in Figure 8.26 shows how concurrent processing can produce a deadlock between two processes that share a variable in main memory. The deadlock phenomenon can occur when processes share other resources as

well. Resources that an operating system must manage include printers and disk files. Sharing any of these resources among concurrent processes can lead to deadlock.

As an example of a deadlock with these resources, suppose a computer system has a hard drive with two files on it, and process P1 requests file 1 for data input. The operating system opens file 1 for P1, which holds it until it does not need the file any longer. P2 may then request input from file 2, which the operating system opens and allocates to that process.

Now suppose P1 needs to write to file 2 that P2 is accessing. It requests access, but the operating system cannot grant the request because the file is already open for P2. The operating system blocks P1 until the file becomes available. If P2 similarly requests to write to file 1, the operating system will block it as well until P1 releases the file.

In this case, the processes are in a state of deadlock. P1 cannot proceed until P2 relinquishes file 2, and P2 cannot proceed until P1 relinquishes file 1. Both are waiting for an event that will never happen, suspended by the operating system.

Resource Allocation Graphs

To manage its resources effectively, the operating system needs a way to detect possible deadlocks. It does so with a structure called a *resource allocation graph*. A resource allocation graph is a visual depiction of the processes and resources in the system that shows which resources are allocated to which processes and which processes are blocked by a request on which resources.

FIGURE 8.31 shows the resource allocation graph for the state in which P1 and P2 are deadlocked over disk file 1 and disk file 2 in the preceding scenario. Processes and resources are nodes in the graph, with processes in circles and resources as solid dots inside boxes. There are two types of edges, allocation edges and request edges.

An *allocation edge* (al) from a resource to a process means the resource is allocated to the process. In the figure, the edge labeled al from disk file 1 to P1 means the operating system has allocated file 1 to process P1. A *request edge* (req) from a process to a resource means the process is blocked, waiting for the resource. The edge labeled req from P2 to disk file 1 means P2 is blocked waiting for disk file 1 to be allocated to it.

The deadlock is evident from the fact that the edges describe a closed path from P1 to R2 to P2 to R1 and back to P1. Such a closed path in a graph is called a *cycle*. A cycle in a resource allocation graph means that a process is blocked on a resource because it is allocated to another process that is blocked on another resource, and so on, with the last resource allocated to the first process. If the cycle cannot be broken, there is a deadlock.

Sometimes the resources in a class are indistinguishable from each other. A process may request one resource from the class and not be

FIGURE 8.31
A resource allocation graph with a deadlock cycle.

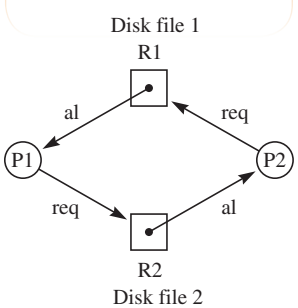
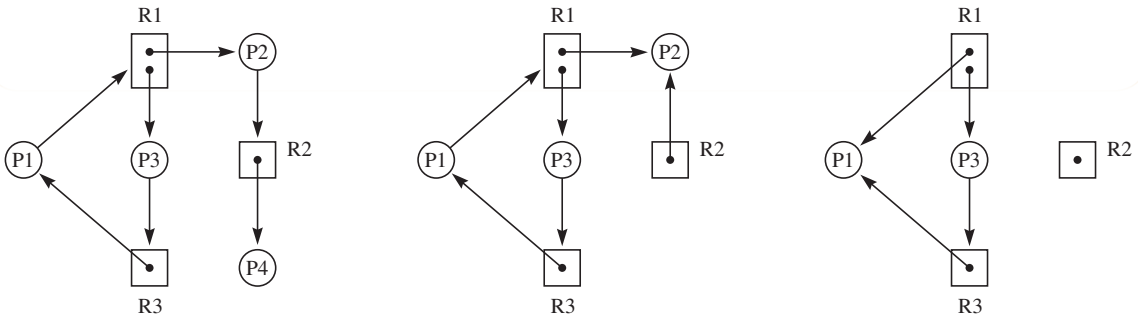


FIGURE 8.32
A resource allocation graph with a cycle but with no deadlock.



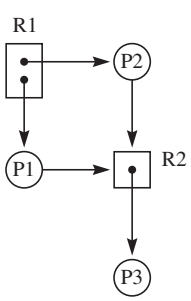
(a) Initial state. **(b)** P4 completes. **(c)** P2 completes.

concerned with which particular resource it gets because they are all equivalent. An example is a group of identical hard drives or identical printers. If a process needs a printer and it does not care which one, the operating system can allocate any printer that is free.

A resource allocation graph represents a class of *n* identical resources with *n* solid dots inside the rectangular box. When the operating system allocates a resource from the class, the allocation edge starts from one of the dots that represents the individual resource. A request edge, however, points to the box because the requesting process does not care which resource within the class it gets.

FIGURE 8.32(a) shows a situation in which both resources in class R1 are allocated. P1 has a request outstanding for one of those resources and does not care which resource it gets.

FIGURE 8.33
A resource allocation graph with no cycle and, therefore, no deadlock.



Even though this graph has cycle (P1, R1, P3, R3, P1), it does not represent a deadlock because the cycle can be broken. Any process in a resource allocation graph that has no request edge from it is not blocked. P4 is not blocked in this figure, and you can assume that it will eventually complete, releasing R2. The operating system will grant P2's request for resource R2. The system then changes the request edge from P2 to R2, to an allocate edge from R2 to P2, as Figure 8.32(b) shows.

Now P2 can use R2 and R1 and run to completion, eventually releasing those resources. When P2 releases an R1 resource, the operating system can allocate the resource to P1, yielding Figure 8.32(c). P1 can run to completion because it has all the resources it needs, after which P3 can complete also. All the processes can complete, so there is no deadlock.

You must consider the direction of the edges when looking for a cycle. You may be tempted to consider (R1, P1, R2, P2, R1) a cycle in FIGURE 8.33,

but it is not. There is no edge from R2 to P2, or from P2 to R1. A cycle is a necessary but not sufficient condition for a deadlock. In this figure, there is no cycle and, therefore, no deadlock.

Deadlock Policy

An operating system may employ one of three general policies for dealing with deadlock:

Three deadlock policies

- Prevent
- Detect and recover
- Ignore

Several different policies may be found in a given operating system. The system may use one policy for one set of resources and another policy for another set.

The prevention policy

The prevention policy employs techniques to ensure that deadlock will not occur. One technique is to require that a job request and be granted once, at the beginning of execution, all the resources it will need to run to completion. The deadlock of Figure 8.31 could not have occurred if P1 had been granted R1 and R2 simultaneously. A deadlock cycle can be set up only if a process is granted a resource and later requests another resource to complete the cycle.

The detect and recover policy

The detect and recover policy allows deadlocks to occur. With it, the operating system periodically executes a program to detect a deadlock cycle in the system. If the operating system discovers a deadlock, it takes away one of the resources held by a process in the cycle. Because the process has partially executed, the operating system usually must terminate the process, unless the state of the resource can be reconstructed later when it is granted to the process.

All these policies have a cost. The prevention policy puts a constraint on the user, particularly if the resources required depend on the input and cannot be known in advance. The detect and recover policy requires CPU time for the detection and recovery algorithms—time that could be spent on user jobs.

The ignore policy

The third policy is to ignore deadlocks. This policy is effective if the costs of the other policies are considered too great, the probability of deadlock is small, or the occurrence of deadlock is inconsequential. For example, a time-shared system may be shut down periodically for routine maintenance, at which time all jobs, including any that are deadlocked, will be removed.

Chapter Summary

The goals of an operating system are to provide a convenient environment for higher-level programming and to allocate the resources of the system efficiently. One important function of an operating system is to manage the jobs that users submit to be executed. A loader is the part of the operating system that places a job into main memory for execution. After the job finishes executing, it turns control of the CPU back to the operating system, which can then load another application program.

A trap handler performs processing for a job, hiding the lower-level details from the programmer. When a trap occurs, the running job's process control block (PCB) is stored while the operating system services the interrupt. The PCB consists of the state of the process: copies of the program counter, the status bits, and the contents of all the CPU registers. To resume the job, the operating system places the PCB back into the CPU. An asynchronous interrupt functions in the same fashion as a procedure call, except that an interrupt is initiated by the operating system, not by the application programmer's code.

A process is a program during execution. In a multiprogramming system, one CPU switches between several processes. In a multiprocessing system, there is more than one CPU. Both multiprogramming and multiprocessing systems maintain concurrently executing processes. To execute cooperating processes concurrently, the operating system must be able to guarantee mutual exclusion of critical sections and avoid deadlock. Peterson's algorithm fills both these requirements. A semaphore is an integer variable provided by the operating system. Its operations include `wait()` and `signal()`, each of which is atomic, or indivisible. Semaphores can also be used to satisfy the mutual exclusion and deadlock requirements.

The deadlock phenomenon can also occur when processes share resources managed by the operating system. A resource allocation graph consists of nodes representing resources and processes, and edges between the nodes representing resource allocations and requests. If the resource allocation graph contains a cycle that cannot be broken, a deadlock has occurred.

Exercises

Section 8.1

1. What are the two purposes of an operating system?

2. The loader in Figure 8.3 executes with the following input:

```
12 00 05 00 00 31 00 03 39 00 03 D0 00 0A F1 FC
16 49 00 15 00 54 68 61 74 27 73 20 61 6C 6C 2E
0A 00 zz
```

Assume that the loop from FC1A to FC4E is executing for the 30th time. State the values in the following registers as four hexadecimal digits:

*(a) A⟨8..15⟩ after LDBA at F61A
*(b) A⟨8..15⟩before ASLA at FC2C
*(c) A⟨8..15⟩ after ASLA at FC2F
(d) A⟨8..15⟩ after LDBA at FC33
(e) A⟨8..15⟩ after ANDA at FC3F
(f) A⟨8..15⟩ after ORA at FC42
(g) X⟨8..15⟩ after ADDX at FC48

3. Do Exercise 2 for the 32nd execution of the loop.

Section 8.2

4. The program of FIGURE 8.34 executes, generating an interrupt for DECI. For Figure 8.6, the entry to and exit from the trap handler, state the values in the following registers as four hexadecimal digits:

*(a) X⟨8..15⟩ after LDBX at FC52
*(b) X after ASRX at FC69
(c) X after SUBX at FC6A
(d) PC after CALL at FC6E
(e) PC after RETTR at FC71

*5. Do Exercise 4 for the DECO instruction.

FIGURE 8.34
The program for Exercise 4.

```
0000  120005          BR      main        ;Branch around data
0003  0000    num:    .BLOCK  2           ;Global variable
0005  310003  main:   DECI    num,d       ;Input decimal value
0008  390003          DECO    num,d       ;Output decimal value
000B  D0000A          LDBA    '\n',i
000E  F1FC16          STBA    charOut,d   ;Output message
0011  490015          STRO    msg,d
0014  00              STOP
0015  546861  msg:    .ASCII  "That's all.\x00"
      742773
      20616C
      6C2E00
0021                  .END
```

*6. Do Exercise 4 for the `STRO` instruction.

7. The program in Exercise 4 runs with an input of `37`. For Figure 8.14, the `DECI` trap handler, state the values in registers a–h as four hexadecimal digits and answer the question in (i):

   *(a) A after `ANDA` at FD98 the first time it executes
   *(b) A after `ANDA` at FD98 the second time it executes
   *(c) X after `LDWX` at FDA1 the first time it executes
   (d) X after `LDWX` at FDA1 the second time it executes
   (e) PC after `BR` at FDA5 the first time it executes
   (f) PC after `BR` at FDA5 the second time it executes
   (g) A after `LDWA` at FE96
   (h) X before `STBX` at FEB1, assuming that the carry bit is zero before the trap
   (i) What statement executes just before `LDWA` at FE74?

*8. Do Exercise 7 with an input of `-295`.

9. The program in Exercise 4 runs with an input of `37`. For Figure 8.15, the `DECO` trap handler, state the values in the following registers as four hexadecimal digits:

   *(a) A after `LDWA` at FEFA
   *(b) A before `STWA` at FF0A

   In the following parts, assume that subroutine `divide` is called from `CALL` at FF2B:

   *(c) A after `LDWA` at FF44
   (d) X before `CPWX` at FF59
   (e) A after `LDWA` at FF68

   In the following parts, assume that subroutine `divide` is called from `CALL` at FF34:

   (f) A after `LDWA` at FF44
   (g) X before `CPWX` at FF59
   (h) X after `ORX` at FF6F

10. Do Exercise 9 with an input of `-2068`.

11. The program in Exercise 4 runs and executes the `STRO` instruction. For Figure 8.16, the `STRO` trap handler, state the values in the registers in hexadecimal:

   (a) A after `LDWA` at FFCE
   (b) A⟨8..15⟩ after `LDBA` at FFE4 the first time it executes
   (c) X after `ADDX` at FFED the first time it executes
   (d) A⟨8..15⟩ after `LDBA` at FFE4 the fifth time it executes
   (e) X after `ADDX` at FFED the fifth time it executes

12. The `DECI` instruction with direct addressing at 0005 in Figure 5.11 executes, generating a trap. The trap handler calls the `setAddr` routine in Figure 8.10. State the values in the index register as four hexadecimal digits:

(a) after `LDWX` at FCF2 **(b)** after `SUBX` at FCF5
(c) after `LDWX` at FCF8

13. The `DECO` instruction with indirect addressing at 004B in Figure 6.41 executes, generating a trap. The trap handler calls the `setAddr` routine in Figure 8.10. State the values in the index register as four hexadecimal digits:

(a) after `LDWX` at FCFF **(b)** after `SUBX` at FD02
(c) after `LDWX` at FD05 **(d)** after `LDWX` at FD08

14. The `DECI` instruction with stack-relative addressing at 0009 in Figure 6.4 executes, generating a trap. The trap handler calls the `setAddr` routine in Figure 8.10. State the values in the index register as four hexadecimal digits:

(a) after `LDWX` at FD0F **(b)** after `SUBX` at FD12
(c) after `LDWX` at FD15 **(d)** after `ADDX` at FD18

15. The `DECI` instruction with stack-indexed addressing at 0013 in Figure 6.36 executes for the second time, generating a trap. The trap handler calls the `setAddr` routine in Figure 8.10. State the values in the index register as four hexadecimal digits:

(a) after `LDWX` at FD42 **(b)** after `SUBX` at FD45
(c) after `LDWX` at FD48 **(d)** after `ADDX` at FD4B
(e) after `ADDX` at FD4E

16. The `DECI` instruction with stack-deferred indexed addressing at 0016 in Figure 6.38 executes for the second time, generating a trap. The trap handler calls the `setAddr` routine in Figure 8.10. State the values in the index register as four hexadecimal digits:

(a) after `LDWX` at FD55 **(b)** after `SUBX` at FD58
(c) after `LDWX` at FD5B **(d)** after `ADDX` at FD5E
(e) after `LDWX` at FD61 **(f)** after `ADDX` at FD64

Section 8.3

17. A short notation for the interleaved execution sequence in Figure 8.23 is 112221, which represents statements executed by P1, P1, P2, P2, P2, P1 in Figure 8.22. **(a)** How many different execution sequences are possible? List each possible sequence in the short notation. For each sequence, state whether `numRes` has the correct value. **(b)** What percentage of

the total number of possible sequences produces an incorrect value? **(c)** Would you expect that percentage to be approximately the probability of an incorrect value when the program runs? Explain.

18. The following attempt to implement critical sections is similar to the program in Figure 8.26 except for the order of the statements in the entry section:

```
Process P1                          Process P2
do                                  do
   while (enter2)                      while (enter1)
      ; //nothing                         ; //nothing
   enter1 = TRUE                       enter2 = TRUE
   critical section                    critical section
   enter1 = FALSE;                     enter2 = FALSE;
   remainder section                   remainder section
while (!done1);                     while (!done2);
```

*(a) Does the algorithm guarantee mutual exclusion? If not, show an execution sequence that lets both processes run in their critical sections simultaneously. **(b)** Does the algorithm prevent deadlock? If not, show an execution sequence that deadlocks P1 and P2.

19. Show from the definitions of `wait()` and `signal()` that the magnitude of `s` is the number of processes blocked.

20. Let I represent an execution of `init(s)`, W of `wait(s)`, and S of `signal(s)`. Then, for example, IWWS represents the sequence of calls `init(s)`, `wait(s)`, `wait(s)`, and `signal(s)` by some processes in an operating system. For each of the following sequences of calls, state the value of `s` and the number of processes blocked after the last call in the sequence:

*(a) IW **(b)** IS **(c)** ISSSW
(d) IWWWS **(e)** ISWWWW

21. Suppose three concurrent processes execute the following code:

```
Process P1                  Process P2                  Process P3
do                          do                          do
   wait(mutEx);                wait(mutEx);                wait(mutEx);
   critical section            critical section            critical section
   signal(mutEx);              signal(mutEx);              signal(mutEx);
   remainder section           remainder section           remainder section
while (!done1);             while (!done2);             while (!done3);
```

Explain how the code guarantees mutual exclusion of all three critical sections.

22. Suppose `s` and `t` are two semaphores initialized with `init(s)` and `init(t)`. Consider the following code fragment of two concurrent processes:

Process P1	Process P2
`wait(s);`	`wait(t);`
`wait(t);`	`wait(s);`
critical section	*critical section*
`signal(s);`	`signal(t);`
`signal(t);`	`signal(s);`
remainder section	*remainder section*

* **(a)** Does the algorithm guarantee mutual exclusion? If not, show an execution sequence that lets both processes run in their critical sections simultaneously. **(b)** Does the algorithm prevent deadlock? If not, show an execution sequence that deadlocks P1 and P2.

23. Consider the code fragment of two concurrent processes:

Process P1	Process P2
Statement 1	*Statement 4*
Statement 2	*Statement 5*
Statement 3	*Statement 6*

Modify the code fragment to guarantee that Statement 5 occurs before Statement 2. Use a semaphore.

24. Each of the following code fragments contains a bug in the entry or exit section. For each fragment, state whether mutual exclusion still holds. If it doesn't, show an execution sequence that violates it. State whether deadlock can occur. If it can, show an execution sequence that produces it.

* **(a)**

Process P1	Process P2
`do`	`do`
`wait(mutEx);`	`signal(mutEx);`
critical section	*critical section*
`signal(mutEx);`	`wait(mutEx);`
remainder section	*remainder section*
`while (!done1);`	`while (!done2);`

(b)

Process P1

```
do
    signal(mutEx);
    critical section
    wait(mutEx);
    remainder section
while (!done1);
```

Process P2

```
do
    signal(mutEx);
    critical section
    wait(mutEx);
    remainder section
while (!done2);
```

(c)

Process P1

```
do
    wait(mutEx);
    critical section
    signal(mutEx);
    remainder section
while (!done1);
```

Process P2

```
do
    wait(mutEx);
    critical section
    wait(mutEx);
    remainder section
while (!done2);
```

(d)

Process P1

```
do
    wait(mutEx);
    critical section
    signal(mutEx);
    remainder section
while (!done1);
```

Process P2

```
do
    wait(mutEx);
    critical section
    remainder section
while (!done2);
```

(e)

Process P1

```
do
    wait(mutEx);
    critical section
    signal(mutEx);
    remainder section
while (!done1);
```

Process P2

```
do
    critical section
    signal(mutEx);
    remainder section
while (!done2);
```

Section 8.4

25. An operating system has processes P1, P2, P3, and P4 and resources R1 (one resource), R2 (one resource), R3 (two resources), and R4 (three resources). The notation (1, 1), (2, 2), (1, 2) means that P1 requests R1, then P2 requests R2, then P1 requests R2. Note that the first two

requests produce allocation edges on the resource allocation graph, but the third request produces a request edge on the graph because R2 is already allocated to P2.

Draw the resource allocation graph after each sequence of requests. State whether the graph contains a cycle. If it does, state whether it is a deadlock cycle.

*(a) (1, 1), (2, 2), (1, 2), (2, 1)
*(b) (1, 4), (2, 4), (3, 4), (4, 4)
(c) (1, 1), (2, 1), (3, 1), (4, 1)
(d) (3, 3), (4, 3), (2, 2), (3, 2), (2, 3)
(e) (1, 2), (1, 3), (1, 4), (2, 2), (2, 3), (2, 4)
(f) (2, 1), (1, 2), (2, 3), (3, 3), (2, 2), (1, 3)
(g) (2, 1), (1, 2), (2, 3), (3, 3), (2, 2), (1, 3), (3, 1)
(h) (1, 4), (2, 3), (3, 3), (2, 1), (3, 4), (1, 3), (4, 4), (3, 1), (2, 4)
(i) (1, 4), (2, 3), (3, 3), (2, 1), (3, 4), (1, 3), (4, 4), (3, 1), (2, 4), (4, 3)

Problems

Section 8.2

26. Implement a new unary instruction in place of `NOP0` called `ASL2` that does two left shifts on the accumulator. `NZC` should should correlate with the new value in the accumulator from the second shift. `V` should be set if either the first or the second shift produced an overflow. Use the test program provided in the Pep/9 app to test the features of the new instruction.

27. Implement a new nonunary instruction in place of `NOP` called `ASLMANY` whose operand is the number of times the accumulator is shifted left. Allow only direct addressing. `NZC` should correlate with the new value in the accumulator from the last shift. `V` should be set if any of the shifts produced an overflow. Use the test program provided in the Pep/9 app to test the features of the new instruction.

28. Implement a new nonunary instruction in place of `NOP` called `MULA` that multiplies the operand by the accumulator and puts the result in the accumulator. Allow only direct addressing. Use the interative shift-and-add algorithm of Problem 6.24. `NZC` should should correlate with the new value in the accumulator from the last addition. `V` should be

set if any of the left shifts or additions produced an overflow. Use the test program provided in the Pep/9 app to test the features of the new instruction.

29. Direct addressing is immediate addressing deferred. Indirect addressing is direct addressing deferred. You can carry this concept one level further with double indirect addressing, which is indirect addressing deferred. Implement a new instruction in place of `NOP0` with mnemonic `STWADI`, which stands for *store word accumulator double indirect*. It should store the accumulator using double indirect addressing. None of the status flags are affected. `NOP0` is a unary instruction as far as the assembler and CPU are concerned, but your program must implement it as a nonunary instruction. You will need to increment the saved `PC` to skip over the operand specifier. Use the test program provided in the Pep/9 app to test the features of the new instruction.

30. Implement a new nonunary instruction in place of `NOP` called `BOOLO`, which means *Boolean output*. It should output `false` if the operand is zero and `true` otherwise. Allow immediate, direct, and stack-relative addressing. None of the status flags are affected. Use the test program provided in the Pep/9 app to test the features of the new instruction.

31. Implement a new unary instruction in place of `NOP0` called `STKADD`. It should replace the two topmost items on the stack with their sum. Set `NZVC` according to the results of the addition. Use the test program provided in the Pep/9 app to test the features of the new instruction.

32. Implement a new nonunary instruction in place of `NOP` called `XORA`, which computes the bitwise exclusive OR operation with the operand and the accumulator, placing the result in the accumulator. Allow only direct addressing. Status bits NZ should be set according to the results of the operation, and VC should remain unchanged. Use the test program provided in the Pep/9 app to test the features of the new instruction.

33. This problem is to implement new nonunary instructions to process floating point numbers. Assume that floating point numbers are stored with all the special values of IEEE 754 but with a two-byte cell having one sign bit, six exponent bits, nine significand bits, and a hidden bit. The exponent uses excess 31 notation except for denormalized numbers, which use excess 30.

(a) Implement a new unary instruction in place of `DECO` called `BINFO`, which stands for binary floating point output. Permit the same addressing modes as with `DECO`. The value 3540 (hex), which represents the normalized

number 1.101×2^{-5}, should be output as `1.101000000b011010`, where the letter `b` stands for two raised to a power and the bit sequence following the `b` is the excess 30 representation of –5. The value 0050 (hex), which represents the denormalized number 0.00101×2^{-30}, should be output as `0.001010000b-30`, where the power will always be –30 for denormalized numbers. Output a NaN value as `NaN`, positive infinity as `inf`, and negative infinity as `-inf`.

(b) Implement a new unary instruction in place of `DECI` called `BINFI`, which stands for *binary floating point input*. Permit the same addressing modes as with `DECI`. Assume that the input will be a normalized binary number. The input `1.101000000b011010`, which represents the normalized number 1.101×2^{-5}, should be stored as 3540 (hex).

(c) Implement a new unary instruction in place of `NOP` called `ADDFA`, which stands for *add floating point accumulator*. Permit the same addressing modes as with `ADDA`. For normalized and denormalized numbers, you may assume that the exponent fields of the two numbers to add are identical, but the exponent field of the sum may not be the same as the initial exponent fields. Your implementation will need to insert the hidden bit before performing the addition and remove it when storing the result. Take into account the possibility that one or both of the operands may be a NaN or infinity.

(d) Work part (c) assuming the exponent fields of the normalized or denormalized numbers may not be identical.