Process Management

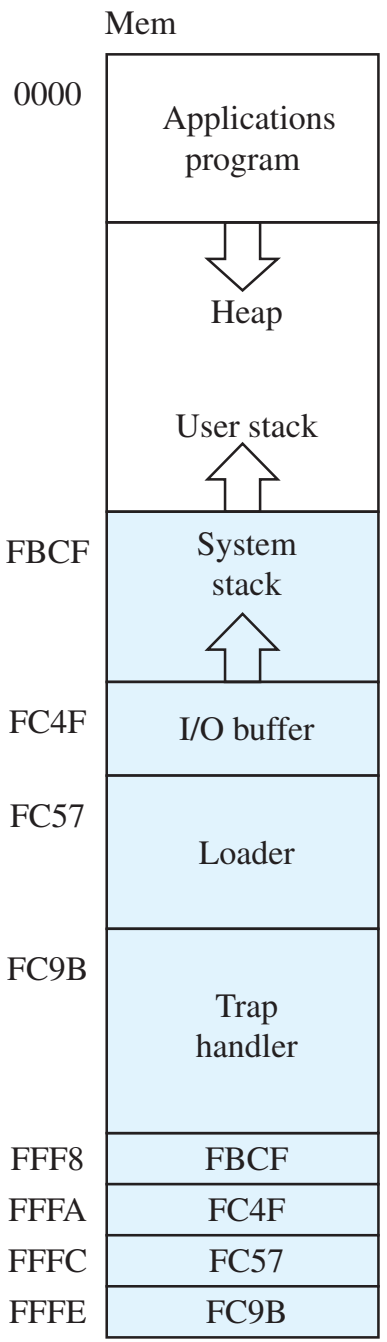
Three types of operating systems

- Single-user
  - Hand-held devices, e.g. cell phones
- Multi-user
  - Personal computers and workstations
- Real-time
  - Equipment control, e.g. car engine

The Pep/8 OS

- An operating system is a program
- One function of an operating system is to manage the jobs (application programs) that users submit
- Because the operating system is itself a program, it is stored in memory
- The location of the OS program relative to the application programs is the *memory map*

Figure 8.1



The Pep/8 loader

- The purpose is to load the application program into memory starting at address 0000
- When you invoke the Pep/8 loader:

  SP←Mem[FFFA]

  PC←Mem[FFFC]

The `.BURN` command

- If `.BURN` is in a program, the assembler assumes the program will be burned into ROM
- It generates code for instructions that follow the `.BURN`
- It does *not* generate code for instructions that precede the `.BURN`
- It computes symbol values assuming the operand of `.BURN` is the *last* address

Figure 8.2

```
                ;******* Pep/8 Operating System
                ;
                TRUE:     .EQUATE 1
                FALSE:    .EQUATE 0
                ;
                ;******* Operating system RAM
FBCF            osRAM:    .BLOCK  128         ;System stack area
FC4F            wordBuff:.BLOCK   1           ;Input/output buffer
FC50            byteBuff:.BLOCK   1           ;Least significant byte of wordBuff
FC51            wordTemp:.BLOCK   1           ;Temporary word storage
FC52            byteTemp:.BLOCK   1           ;Least significant byte of tempWord
FC53            addrMask:.BLOCK   2           ;Addressing mode mask
FC55            opAddr:   .BLOCK  2           ;Trap instruction operand address
                ;
                ;******* Operating system ROM
FC57                      .BURN   0xFFFF
```

Figure 8.3

```
                ;******* 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.
                ;
FC57   C80000 loader:  LDX     0,i         ;X := 0
FC5A   E9FC4F          STX     wordBuff,d ;Clear input buffer word
                ;
```

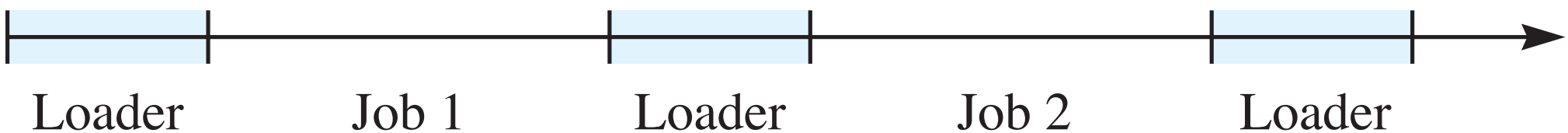
Figure 8.3
(Continued)

```
FC5D   49FC50 getChar: CHARI    byteBuff,d ;Get first hex character
FC60   C1FC4F          LDA      wordBuff,d ;Put ASCII into low byte of A
FC63   B0007A          CPA      'z',i      ;If end of file sentinel 'z'
FC66   0AFC9A          BREQ     stopLoad   ;then exit loader routine
FC69   B00039          CPA      '9',i      ;If characer <= '9', assume decimal
FC6C   06FC72          BRLE     shift      ;and right nybble is correct digit
FC6F   700009          ADDA     9,i        ;else convert nybble to correct digit
FC72   1C      shift:  ASLA                ;Shift left by four bits to send
FC73   1C              ASLA                ;the digit to the most significant
FC74   1C              ASLA                ;position in the byte
FC75   1C              ASLA
FC76   F1FC52          STBYTEA byteTemp,d ;Save the most significant nybble
FC79   49FC50          CHARI   byteBuff,d ;Get second hex character
FC7C   C1FC4F          LDA     wordBuff,d ;Put ASCII into low byte of A
FC7F   B00039          CPA     '9',i      ;If characer <= '9', assume decimal
FC82   06FC88          BRLE    combine    ;and right nybble is correct digit
FC85   700009          ADDA    9,i        ;else convert nybble to correct digit
FC88   90000F combine: ANDA    0x000F,i   ;Mask out the left nybble
FC8B   A1FC51          ORA     wordTemp,d ;Combine both hex digits in binary
FC8E   F50000          STBYTEA 0,x        ;Store in Mem[X]
FC91   780001          ADDX    1,i        ;X := X + 1
FC94   49FC50          CHARI   byteBuff,d ;Skip blank or <LF>
FC97   04FC5D          BR      getChar    ;
               ;
FC9A   00      stopLoad:STOP              ;
```

Program termination

- Pep/8 OS
  - When a program terminates with `STOP`, control returns to the user of the Pep/8 simulator
- Real-world OS
  - When a program terminates, the computer does not stop, but returns control to the operating system

Figure 8.4

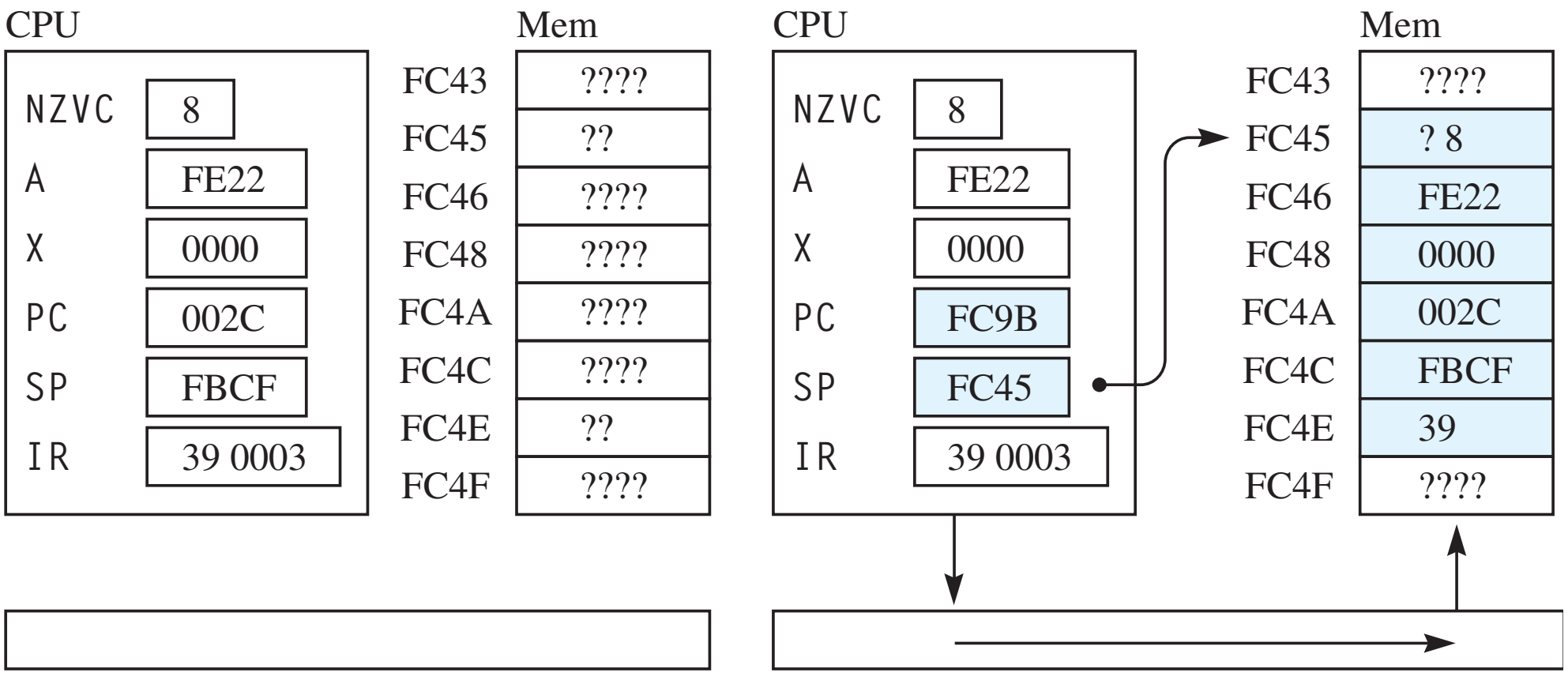


Traps

- Executed by the unimplemented nonunary instructions `DECI`, `DECO`, `STRO`, `NOP` and the unary instructions `NOP0`, `NOP1`, `NOP2`, `NOP3`
- Similar to the `CALL` instruction, but all the registers, not just SP, are stored on the system stack

The trap mechanism

$$
\begin{aligned}
\text{Temp} &\leftarrow \text{Mem}[7FFA]\ ; \\
\text{Mem}[\text{Temp} - 1] &\leftarrow \text{IR}\langle 0..7\rangle\ ; \\
\text{Mem}[\text{Temp} - 3] &\leftarrow \text{SP}\ ; \\
\text{Mem}[\text{Temp} - 5] &\leftarrow \text{PC}\ ; \\
\text{Mem}[\text{Temp} - 7] &\leftarrow \text{X}\ ; \\
\text{Mem}[\text{Temp} - 9] &\leftarrow \text{A}\ ; \\
\text{Mem}[\text{Temp} - 10]\langle 4..7\rangle &\leftarrow \text{NZVC}\ ; \\
\text{SP} &\leftarrow \text{Temp} - 10\ ; \\
\text{PC} &\leftarrow \text{Mem}[FFFE]
\end{aligned}
$$



(a) Before the trap.

(b) After the trap.

Processes

- Process
  - A program during execution
- Process Control Block
  - The block of information in main memory that contains a copy of the trapped processes' registers

The return from trap instruction

- Instruction specifier: 0101 1rrr
- Mnemonic: `RETn` (`RET0`, `RET1`, ... `RET7`)

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

The trap handlers

0010 01nn	`NOPn`	Unary no-operation
0010 1aaa	`NOP`	Nonunary no-operation
0011 0aaa	`DECI`	Nonunary decimal input
0011 1aaa	`DECO`	Nonunary decimal output
0100 0aaa	`STRO`	Nonunary string output

The test for NOPn

0010 0100 `NOP0` Right-most two bits 00

0010 0101 `NOP1` Right-most two bits 01

0010 0110 `NOP2` Right-most two bits 10

0010 0111 `NOP3` Right-most two bits 11

```
                ;******* Trap handler
                oldIR:   .EQUATE 9           ;Stack address of IR on trap
                ;
FC9B  C80000    trap:    LDX     0,i         ;Clear X for a byte compare
FC9E  DB0009             LDBYTEX oldIR,s     ;X := trapped IR
FCA1  B80028             CPX     0x0028,i    ;If X >= first nonunary trap opcode
FCA4  0EFCB7             BRGE    nonUnary    ;trap opcode is nonunary
                ;
FCA7  980003    unary:   ANDX    0x0003,i    ;Mask out all but rightmost two bits
FCAA  1D                 ASLX                ;An address is two bytes
FCAB  17FCAF             CALL    unaryJT,x   ;Call unary trap routine
FCAE  01                 RETTR               ;Return from trap
                ;
FCAF  FDB6      unaryJT: .ADDRSS opcode24    ;Address of NOP0 subroutine
FCB1  FDB7               .ADDRSS opcode25    ;Address of NOP1 subroutine
FCB3  FDB8               .ADDRSS opcode26    ;Address of NOP2 subroutine
FCB5  FDB9               .ADDRSS opcode27    ;Address of NOP3 subroutine
                ;
```

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`

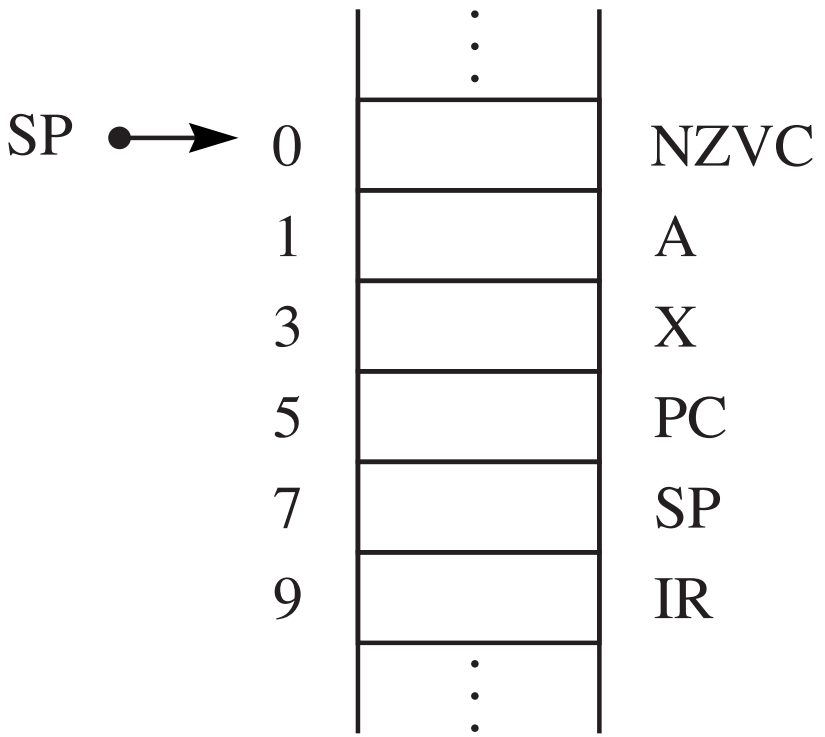
2 if the trap IR contains 0011 1aaa `DECO`

3 if the trap IR contains 0100 0aaa `STRO`

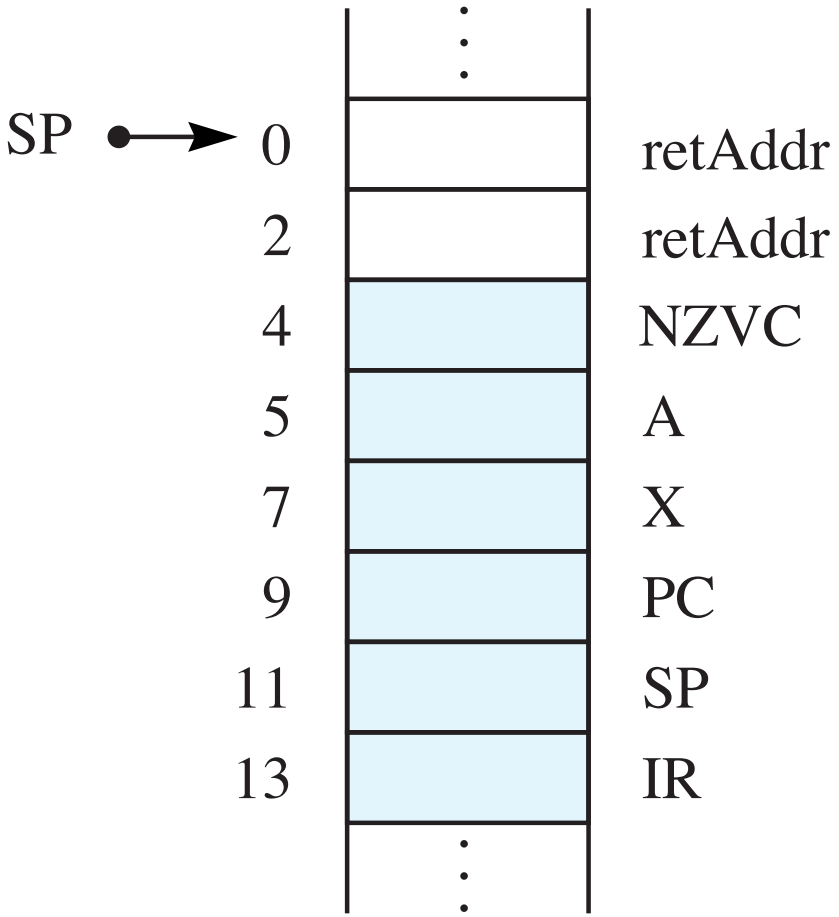
Figure 8.6
(Continued)

```
FCB7   1F      nonUnary:ASRX                ;Trap opcode is nonunary
FCB8   1F               ASRX                ;Discard addressing mode bits
FCB9   1F               ASRX
FCBA   880005           SUBX    5,i         ;Adjust so that NOP opcode = 0
FCBD   1D               ASLX                ;An address is two bytes
FCBE   17FCC2           CALL    nonUnJT,x   ;Call nonunary trap routine
FCC1   01      return:  RETTR               ;Return from trap
               ;
FCC2   FDBA    nonUnJT: .ADDRSS opcode28    ;Address of NOP subroutine
FCC4   FDC4             .ADDRSS opcode30    ;Address of DECI subroutine
FCC6   FF3B             .ADDRSS opcode38    ;Address of DECO subroutine
FCC8   FFC6             .ADDRSS opcode40    ;Address of STRO subroutine
```

Figure 8.7



(a) Immediately after a trap.

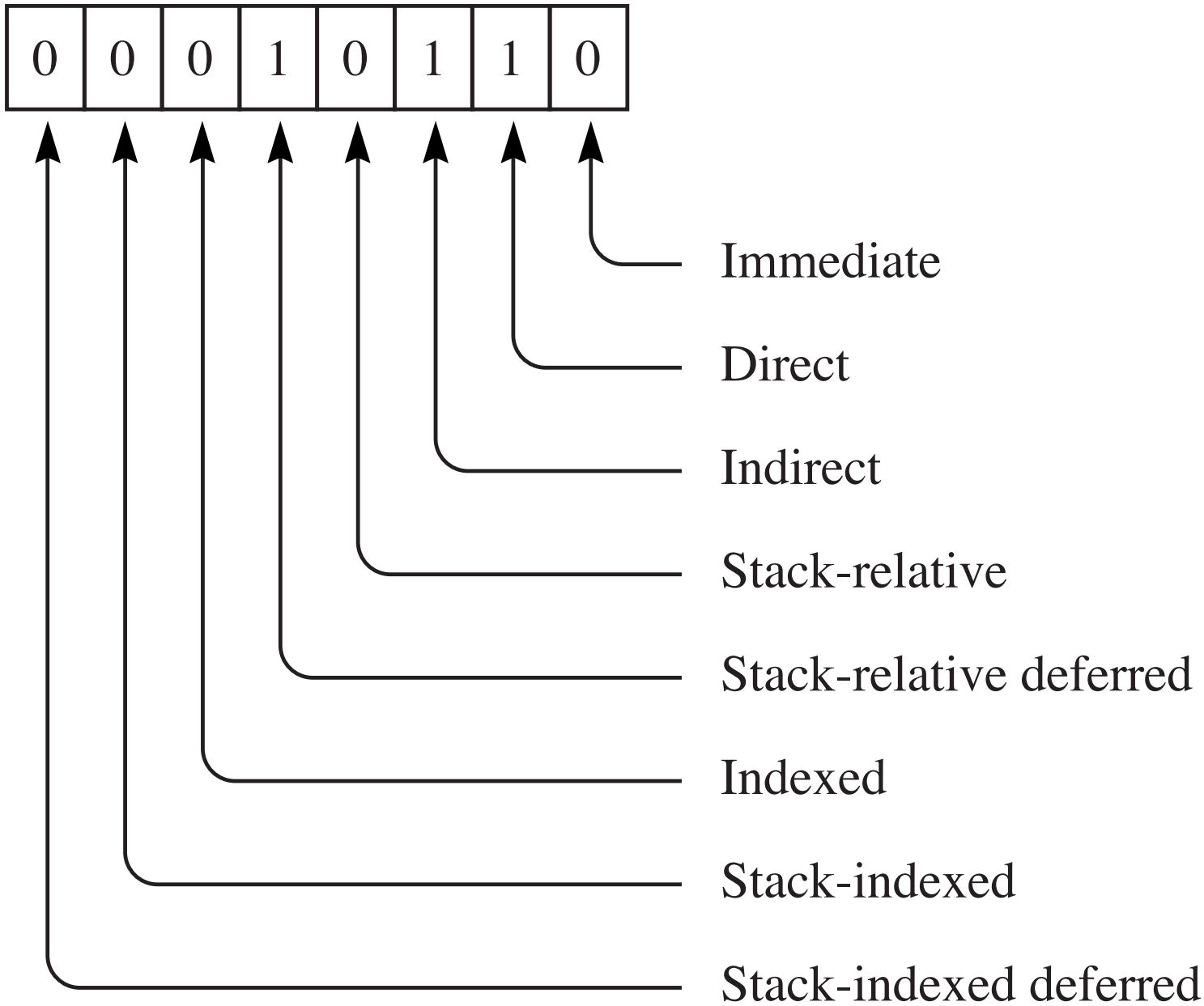


(b) With two return addresses on the run-time stack. The shaded region is the PCB.

Trap addressing mode assertion

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

Figure 8.9



Trap addressing mode assertion

- 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

Figure 8.8

```
                ;******* Assert valid trap addressing mode
                oldIR4:  .EQUATE 13          ;oldIR + 4 with two return addresses
FCCA  C00001 assertAd:LDA     1,i           ;A := 1
FCCD  DB000D           LDBYTEX oldIR4,s     ;X := OldIR
FCD0  980007           ANDX    0x0007,i     ;Keep only the addressing mode bits
FCD3  0AFCDD           BREQ    testAd       ;000 = immediate addressing
FCD6  1C      loop:    ASLA                 ;Shift the 1 bit left
FCD7  880001           SUBX    1,i          ;Subtract from addressing mode count
FCDA  0CFCD6           BRNE    loop         ;Try next addressing mode
FCDD  91FC53  testAd:  ANDA    addrMask,d   ;AND the 1 bit with legal modes
FCE0  0AFCE4           BREQ    addrErr
FCE3  58               RETO                 ;Legal addressing mode, return
FCE4  50000A addrErr:  CHARO   '\n',i
FCE7  C0FCF4           LDA     trapMsg,i    ;Push address of error message
FCEA  E3FFFE           STA     -2,s
FCED  680002           SUBSP   2,i          ;Call print subroutine
FCF0  16FFE2           CALL    prntMsg
FCF3  00               STOP                 ;Halt: Fatal runtime error
FCF4  455252 trapMsg:  .ASCII  "ERROR: Invalid trap addressing mode.\x00"
      ...
```

Trap operand address computation

- 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

Figure 8.10

```
                ;******* 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
FD19  DB000D    setAddr: LDBYTEX oldIR4,s    ;X := old instruction register
FD1C  980007             ANDX    0x0007,i    ;Keep only the addressing mode bits
FD1F  1D                 ASLX                ;An address is two bytes
FD20  05FD23             BR      addrJT,x
FD23  FD33      addrJT:  .ADDRSS addrI       ;Immediate addressing
FD25  FD3D               .ADDRSS addrD       ;Direct addressing
FD27  FD4A               .ADDRSS addrN       ;Indirect addressing
FD29  FD5A               .ADDRSS addrS       ;Stack relative addressing
FD2B  FD6A               .ADDRSS addrSF      ;Stack relative deferred addressing
FD2D  FD7D               .ADDRSS addrX       ;Indexed addressing
FD2F  FD8D               .ADDRSS addrSX      ;Stack indexed addressing
FD31  FDA0               .ADDRSS addrSXF     ;Stack indexed deferred addressing
                ;
FD33  CB0009    addrI:   LDX     oldPC4,s    ;Immediate addressing
FD36  880002             SUBX    2,i         ;Oprnd = OprndsSpec
FD39  E9FC55             STX     opAddr,d
FD3C  58                 RET0
                ;
```

Figure 8.10
(Continued)

```
FD3D   CB0009 addrD:   LDX     oldPC4,s   ;Direct addressing
FD40   880002          SUBX    2,i        ;Oprnd = Mem[OprndSpec]
FD43   CD0000          LDX     0,x
FD46   E9FC55          STX     opAddr,d
FD49   58              RETO
                ;
FD4A   CB0009 addrN:   LDX     oldPC4,s   ;Indirect addressing
FD4D   880002          SUBX    2,i        ;Oprnd = Mem[Mem[OprndSpec]]
FD50   CD0000          LDX     0,x
FD53   CD0000          LDX     0,x
FD56   E9FC55          STX     opAddr,d
FD59   58              RETO
                ;
FD5A   CB0009 addrS:   LDX     oldPC4,s   ;Stack relative addressing
FD5D   880002          SUBX    2,i        ;Oprnd = Mem[SP + OprndSpec]
FD60   CD0000          LDX     0,x
FD63   7B000B          ADDX    oldSP4,s
FD66   E9FC55          STX     opAddr,d
FD69   58              RETO
                ;
FD6A   CB0009 addrSF:  LDX     oldPC4,s   ;Stack relative deferred addressing
FD6D   880002          SUBX    2,i        ;Oprnd = Mem[Mem[SP + OprndSpec]]
FD70   CD0000          LDX     0,x
FD73   7B000B          ADDX    oldSP4,s
FD76   CD0000          LDX     0,x
FD79   E9FC55          STX     opAddr,d
FD7C   58              RETO
```

Figure 8.10
(Continued)

```
FD7D   CB0009 addrX:    LDX     oldPC4,s    ;Indexed addressing
FD80   880002           SUBX    2,i         ;Oprnd = Mem[OprndSpec + X]
FD83   CD0000           LDX     0,x
FD86   7B0007           ADDX    oldX4,s
FD89   E9FC55           STX     opAddr,d
FD8C   58               RET0
                ;
FD8D   CB0009 addrSX:   LDX     oldPC4,s    ;Stack indexed addressing
FD90   880002           SUBX    2,i         ;Oprnd = Mem[SP + OprndSpec + X]
FD93   CD0000           LDX     0,x
FD96   7B0007           ADDX    oldX4,s
FD99   7B000B           ADDX    oldSP4,s
FD9C   E9FC55           STX     opAddr,d
FD9F   58               RET0
                ;
FDA0   CB0009 addrSXF:  LDX     oldPC4,s    ;Stack indexed deferred addressing
FDA3   880002           SUBX    2,i         ;Oprnd = Mem[Mem[SP + OprndSpec] + X]
FDA6   CD0000           LDX     0,x
FDA9   7B000B           ADDX    oldSP4,s
FDAC   CD0000           LDX     0,x
FDAF   7B0007           ADDX    oldX4,s
FDB2   E9FC55           STX     opAddr,d
FDB5   58               RET0
```

The no-operation trap handlers

- Do nothing when executed
- Provided for systems programmer to write her own trap handler

Figure 8.11

```
                ;******* Opcode 0x24
                ;The NOP0 instruction.
FDB6   58       opcode24:RET0
                ;
                ;******* Opcode 0x25
                ;The NOP1 instruction.
FDB7   58       opcode25:RET0
                ;
                ;******* Opcode 0x26
                ;The NOP2 instruction.
FDB8   58       opcode26:RET0
                ;
                ;******* Opcode 0x27
                ;The NOP3 instruction.
FDB9   58       opcode27:RET0
                ;
                ;******* Opcode 0x28
                ;The NOP instruction.
FDBA   C00001   opcode28:LDA     0x0001,i    ;Assert i
FDBD   E1FC53            STA     addrMask,d
FDC0   16FCCA            CALL    assertAd
FDC3   58                RET0
```

The DECI trap handler

- Parses the input, converting the string of ASCII characters to the proper bits in two's complement representation
- Based on a finite state machine

Figure 8.12

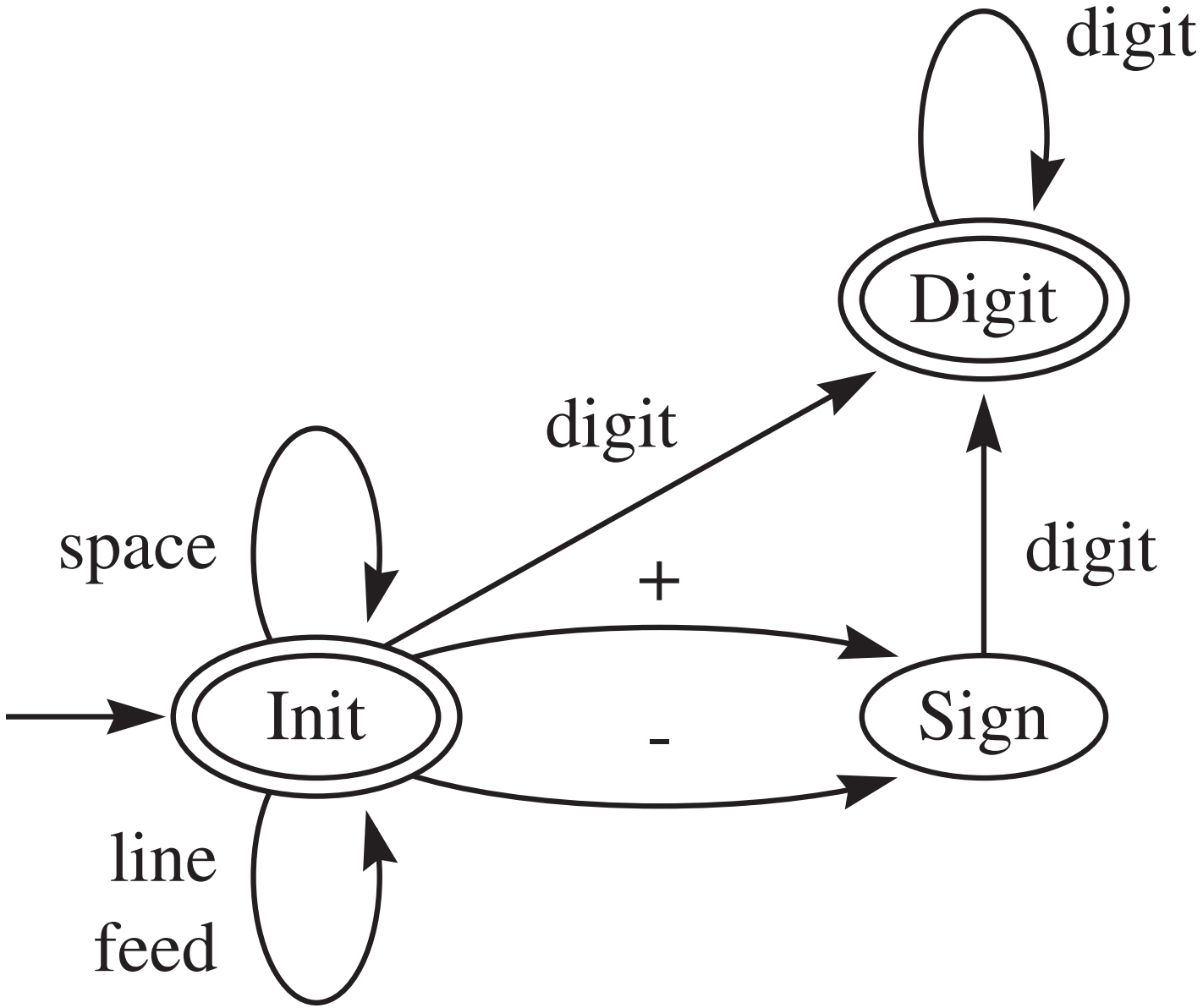


Figure 8.13

```
isOvfl := false
state := init
do
   CHARI asciiCh
   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
      }
```

Figure 8.13
(Continued)

```
   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 8.14

```
;******* 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 14          ;Stack address of NZVC on interrupt
;
total:   .EQUATE 10          ;Cumulative total of DECI number
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
;
```

Figure 8.14
(Continued)

```
FDC4   C000FE opcode30:LDA     0x00FE,i   ;Assert d, n, s, sf, x, sx, sxf
FDC7   E1FC53          STA     addrMask,d
FDCA   16FCCA          CALL    assertAd
FDCD   16FD19          CALL    setAddr    ;Set address of trap operand
FDD0   68000C          SUBSP   12,i       ;Allocate storage for locals
FDD3   C00000          LDA     FALSE,i    ;isOvfl := FALSE
FDD6   E30006          STA     isOvfl,s
FDD9   C00000          LDA     init,i     ;state := init
FDDC   E30002          STA     state,s
FDDF   C00000          LDA     0,i        ;wordBuff := 0 for input
FDE2   E1FC4F          STA     wordBuff,d
                ;
FDE5   49FC50 do:      CHARI   byteBuff,d ;Get asciiCh
FDE8   C1FC4F          LDA     wordBuff,d ;Set value(asciiCH)
FDEB   90000F          ANDA    0x000F,i
FDEE   E30008          STA     valAscii,s
FDF1   C1FC4F          LDA     wordBuff,d ;A = asciiCh throughout the loop
FDF4   CB0002          LDX     state,s    ;switch (state)
FDF7   1D              ASLX               ;An address is two bytes
FDF8   05FDFB          BR      stateJT,x
                ;
FDFB   FE01    stateJT: .ADDRSS sInit
FDFD   FE5B             .ADDRSS sSign
FDFF   FE76             .ADDRSS sDigit
                ;
```

Figure 8.14
(Continued)

```
FE01  B0002B sInit:   CPA     '+',i       ;if (asciiCh == '+')
FE04  0CFE16          BRNE    ifMinus
FE07  C80000          LDX     FALSE,i     ;isNeg := FALSE
FE0A  EB0004          STX     isNeg,s
FE0D  C80001          LDX     sign,i      ;state := sign
FE10  EB0002          STX     state,s
FE13  04FDE5          BR      do
                ;
FE16  B0002D ifMinus: CPA     '-',i       ;else if (asciiCh == '-')
FE19  0CFE2B          BRNE    ifDigit
FE1C  C80001          LDX     TRUE,i      ;isNeg := TRUE
FE1F  EB0004          STX     isNeg,s
FE22  C80001          LDX     sign,i      ;state := sign
FE25  EB0002          STX     state,s
FE28  04FDE5          BR      do
                ;
FE2B  B00030 ifDigit: CPA     '0',i       ;else if (asciiCh is a digit)
FE2E  08FE4C          BRLT    ifWhite
FE31  B00039          CPA     '9',i
FE34  10FE4C          BRGT    ifWhite
FE37  C80000          LDX     FALSE,i     ;isNeg := FALSE
FE3A  EB0004          STX     isNeg,s
FE3D  CB0008          LDX     valAscii,s  ;total := Value(asciiCh)
FE40  EB000A          STX     total,s
FE43  C80002          LDX     digit,i     ;state := digit
FE46  EB0002          STX     state,s
FE49  04FDE5          BR      do
```

Figure 8.14
(Continued)

```
FE4C  B00020 ifWhite: CPA     ' ',i       ;else if (asciiCh is not a space
FE4F  0AFDE5          BREQ    do
FE52  B0000A          CPA     '\n',i      ;or line feed)
FE55  0CFF11          BRNE    deciErr     ;exit with DECI error
FE58  04FDE5          BR      do
                ;
FE5B  B00030 sSign:   CPA     '0',i       ;if asciiCh (is not a digit)
FE5E  08FF11          BRLT    deciErr
FE61  B00039          CPA     '9',i
FE64  10FF11          BRGT    deciErr     ;exit with DECI error
FE67  CB0008          LDX     valAscii,s  ;else total := Value(asciiCh)
FE6A  EB000A          STX     total,s
FE6D  C80002          LDX     digit,i     ;state := digit
FE70  EB0002          STX     state,s
FE73  04FDE5          BR      do
                ;
```

Figure 8.14
(Continued)

```
FE76  B00030 sDigit:  CPA     '0',i       ;if (asciiCh is not a digit)
FE79  08FEC7          BRLT    deciNorm
FE7C  B00039          CPA     '9',i
FE7F  10FEC7          BRGT    deciNorm    ;exit normaly
FE82  C80001          LDX     TRUE,i      ;else X := TRUE for later assignments
FE85  C3000A          LDA     total,s     ;Multiply total by 10 as follows:
FE88  1C              ASLA                ;First, times 2
FE89  12FE8F          BRV     ovfl1       ;If overflow then
FE8C  04FE92          BR      L1
FE8F  EB0006 ovfl1:   STX     isOvfl,s    ;isOvfl := TRUE
FE92  E30000 L1:      STA     temp,s      ;Save 2 * total in temp
FE95  1C              ASLA                ;Now, 4 * total
FE96  12FE9C          BRV     ovfl2       ;If overflow then
FE99  04FE9F          BR      L2
```

Figure 8.14
(Continued)

```
FE9C  EB0006 ovfl2:  STX     isOvfl,s    ;isOvfl := TRUE
FE9F  1C     L2:     ASLA                ;Now, 8 * total
FEA0  12FEA6         BRV     ovfl3       ;If overflow then
FEA3  04FEA9         BR      L3
FEA6  EB0006 ovfl3:  STX     isOvfl,s    ;isOvfl := TRUE
FEA9  730000 L3:     ADDA    temp,s      ;Finally, 8 * total + 2 * total
FEAC  12FEB2         BRV     ovfl4       ;If overflow then
FEAF  04FEB5         BR      L4
FEB2  EB0006 ovfl4:  STX     isOvfl,s    ;isOvfl := TRUE
FEB5  730008 L4:     ADDA    valAscii,s  ;A := 10 * total + valAscii
FEB8  12FEBE         BRV     ovfl5       ;If overflow then
FEBB  04FEC1         BR      L5
FEBE  EB0006 ovfl5:  STX     isOvfl,s    ;isOvfl := TRUE
FEC1  E3000A L5:     STA     total,s     ;Update total
FEC4  04FDE5         BR      do
                     ;
```

Figure 8.14
(Continued)

```
FEC7  C30004 deciNorm:LDA     isNeg,s    ;If isNeg then
FECA  0AFEE3           BREQ    setNZ
FECD  C3000A           LDA     total,s    ;If total != 0x8000 then
FED0  B08000           CPA     0x8000,i
FED3  0AFEDD           BREQ    L6
FED6  1A               NEGA               ;Negate total
FED7  E3000A           STA     total,s
FEDA  04FEE3           BR      setNZ
FEDD  C00000 L6:       LDA     FALSE,i    ;else -32768 is a special case
FEE0  E30006           STA     isOvfl,s   ;isOvfl := FALSE
              ;
```

Figure 8.14
(Continued)

```
FEE3   DB000E setNZ:    LDBYTEX oldNZVC,s   ;Set NZ according to total result:
FEE6   980001           ANDX    0x0001,i    ;First initialize NZV to 000
FEE9   C3000A           LDA     total,s     ;If total is negative then
FEEC   0EFEF2           BRGE    checkZ
FEEF   A80008           ORX     0x0008,i    ;set N to 1
FEF2   B00000 checkZ:   CPA     0,i         ;If total is not zero then
FEF5   0CFEFB           BRNE    setV
FEF8   A80004           ORX     0x0004,i    ;set Z to 1
FEFB   C30006 setV:     LDA     isOvfl,s    ;If not isOvfl then
FEFE   0AFF04           BREQ    storeFl
FF01   A80002           ORX     0x0002,i    ;set V to 1
FF04   FB000E storeFl:  STBYTEX oldNZVC,s   ;Store the NZVC flags
               ;
FF07   C3000A exitDeci:LDA      total,s     ;Put total in memory
FF0A   E2FC55           STA     opAddr,n
FF0D   60000C           ADDSP   12,i        ;Deallocate locals
FF10   58               RET0                ;Return to trap handler
               ;
FF11   50000A deciErr:  CHARO   '\n',i
FF14   C0FF21           LDA     deciMsg,i   ;Push address of message onto stack
FF17   E3FFFE           STA     -2,s
FF1A   680002           SUBSP   2,i
FF1D   16FFE2           CALL    prntMsg     ;and print
FF20   00               STOP                ;Fatal error: program terminates
               ;
FF21   455252 deciMsg:  .ASCII  "ERROR: Invalid DECI input\x00"
       ...
```

The DECO trap handler

- Outputs the operand of DECO in a format that is equivalent to the C++ `cout <<` operation on an integer value
- Outputs at most five characters, preceded by the hyphen character – if necessary

Figure 8.15

```
                ;******* 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
                chOut:    .EQUATE 2           ;Has a character been output yet?
                place:    .EQUATE 4           ;Place value for division
                ;
FF3B  C000FF  opcode38:LDA      0x00FF,i    ;Assert i, d, n, s, sf, x, sx, sxf
FF3E  E1FC53            STA     addrMask,d
FF41  16FCCA            CALL    assertAd
FF44  16FD19            CALL    setAddr     ;Set address of trap operand
FF47  680006            SUBSP   6,i         ;Allocate storage for locals
FF4A  C2FC55            LDA     opAddr,n    ;A := oprnd
FF4D  B00000            CPA     0,i         ;If oprnd is negative then
FF50  0EFF57            BRGE    printMag
FF53  50002D            CHARO   '-',i       ;Print leading '-' and
FF56  1A                NEGA                ;make magnitude positive
```

Figure 8.15
(Continued)

```
FF57  E30000 printMag:STA     remain,s   ;remain := abs(oprnd)
FF5A  C00000          LDA     FALSE,i    ;Initialize chOut := FALSE
FF5D  E30002          STA     chOut,s
FF60  C02710          LDA     10000,i    ;place := 10,000
FF63  E30004          STA     place,s
FF66  16FF91          CALL    divide     ;Write 10,000's place
FF69  C003E8          LDA     1000,i     ;place := 1,000
FF6C  E30004          STA     place,s
FF6F  16FF91          CALL    divide     ;Write 1000's place
FF72  C00064          LDA     100,i      ;place := 100
FF75  E30004          STA     place,s
FF78  16FF91          CALL    divide     ;Write 100's place
FF7B  C0000A          LDA     10,i       ;place := 10
FF7E  E30004          STA     place,s
FF81  16FF91          CALL    divide     ;Write 10's place
FF84  C30000          LDA     remain,s   ;Always write 1's place
FF87  A00030          ORA     0x0030,i   ;Convert decimal to ASCII
FF8A  F1FC50          STBYTEA byteBuff,d
FF8D  51FC50          CHARO   byteBuff,d
FF90  5E              RET6
```

Figure 8.15
(Continued)

```
;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
chOut2:  .EQUATE 4          ;subroutine are greater by two becaus
place2:  .EQUATE 6          ;the retAddr is on the stack
;
```

Figure 8.15
(Continued)

```
FF91  C30002 divide:  LDA     remain2,s  ;A := remainder
FF94  C80000          LDX     0,i        ;X := 0
FF97  830006 divLoop: SUBA    place2,s   ;Division by repeated subtraction
FF9A  08FFA6          BRLT    writeNum   ;If remainder is negative then done
FF9D  780001          ADDX    1,i        ;X := X + 1
FFA0  E30002          STA     remain2,s  ;Store the new remainder
FFA3  04FF97          BR      divLoop
                ;
FFA6  B80000 writeNum:CPX     0,i        ;If X != 0 then
FFA9  0AFFB5          BREQ    checkOut
FFAC  C00001          LDA     TRUE,i     ;chOut := TRUE
FFAF  E30004          STA     chOut2,s
FFB2  04FFBC          BR      printDgt   ;and branch to print this digit
FFB5  C30004 checkOut:LDA     chOut2,s   ;else if a previous char was output
FFB8  0CFFBC          BRNE    printDgt   ;then branch to print this zero
FFBB  58              RET0               ;else return to calling routine
                ;
FFBC  A80030 printDgt:ORX     0x0030,i   ;Convert decimal to ASCII
FFBF  E9FC4F          STX     wordBuff,d ;for output
FFC2  51FC50          CHARO   byteBuff,d
FFC5  58              RET0               ;return to calling routine
```

The STRO instruction

- Outputs a null-terminated string from memory

The OS vectors

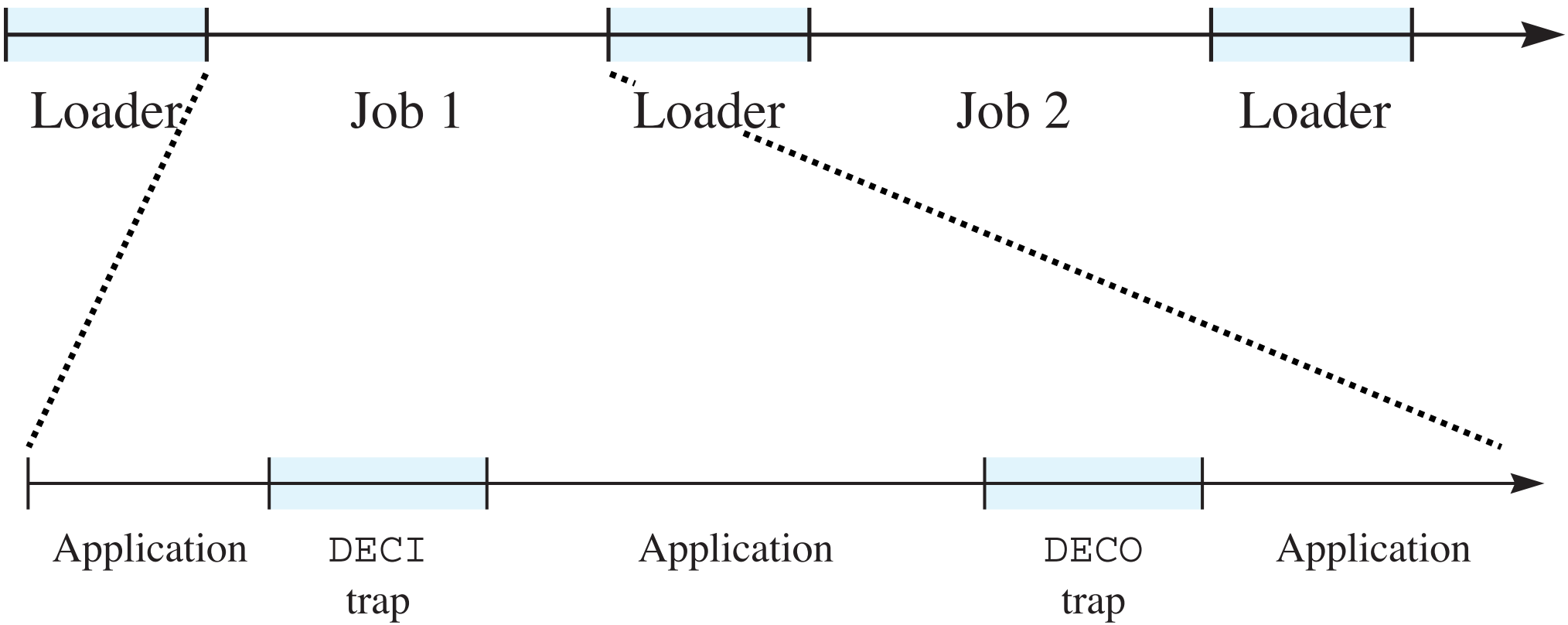
- Established with .ADDRSS command

Figure 8.16

```
                ;******* Opcode 0x40
                ;The STRO instruction.
                ;Outputs a null-terminated string from memory.
                ;
FFC6  C00016 opcode40:LDA     0x0016,i   ;Assert d, n, sf
FFC9  E1FC53          STA     addrMask,d
FFCC  16FCCA          CALL    assertAd
FFCF  16FD19          CALL    setAddr    ;Set address of trap operand
FFD2  C1FC55          LDA     opAddr,d   ;Push address of string to print
FFD5  E3FFFE          STA     -2,s
FFD8  680002          SUBSP   2,i
FFDB  16FFE2          CALL    prntMsg    ;and print
FFDE  600002          ADDSP   2,i
FFE1  58              RET0
```

Figure 8.16
(Continued)

```
                ;
                ;******* 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
                ;
FFE2  C80000    prntMsg: LDX     0,i        ;X := 0
FFE5  C00000             LDA     0,i        ;A := 0
FFE8  D70002    prntMore:LDBYTEA msgAddr,sxf;Test next char
FFEB  0AFFF7             BREQ    exitPrnt   ;If null then exit
FFEE  570002             CHARO   msgAddr,sxf;else print
FFF1  780001             ADDX    1,i        ;X := X + 1 for next character
FFF4  04FFE8             BR      prntMore
                ;
FFF7  58        exitPrnt:RET0
                ;
                ;******* Vectors for System Memory Format
FFF8  FBCF               .ADDRSS osRAM      ;User stack pointer
FFFA  FC4F               .ADDRSS wordBuff   ;System stack pointer
FFFC  FC57               .ADDRSS loader     ;Loader program counter
FFFE  FC9B               .ADDRSS trap       ;Trap program counter
```



Asynchronous interrupts

- Time outs
- I/O completions

Multiprogramming

- An operating system that can switch back and forth between processes to keep the CPU busy is called a *multiprogramming system*
- It maintains a queue of process control blocks (PCBs)

Figure 8.18

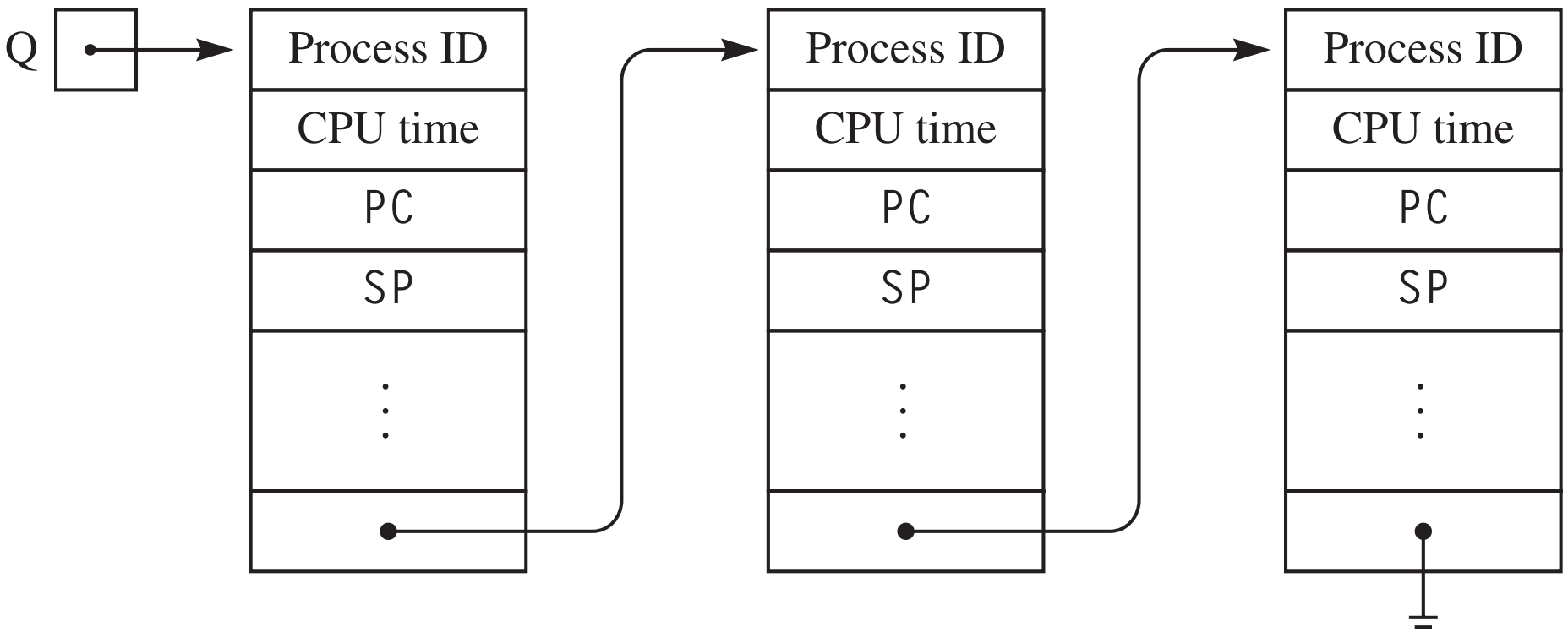
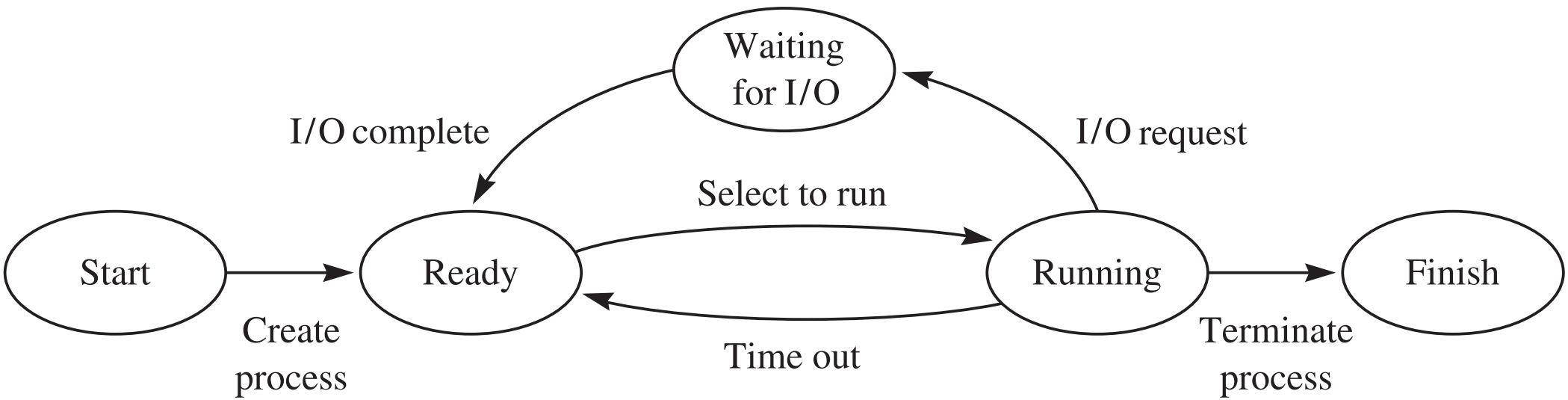


Figure 8.19



Multiprocessing

- A computer system with more than one physical CPU
- Also maintains a queue of PCBs, but more than one process can be running at the same time

Figure 8.20

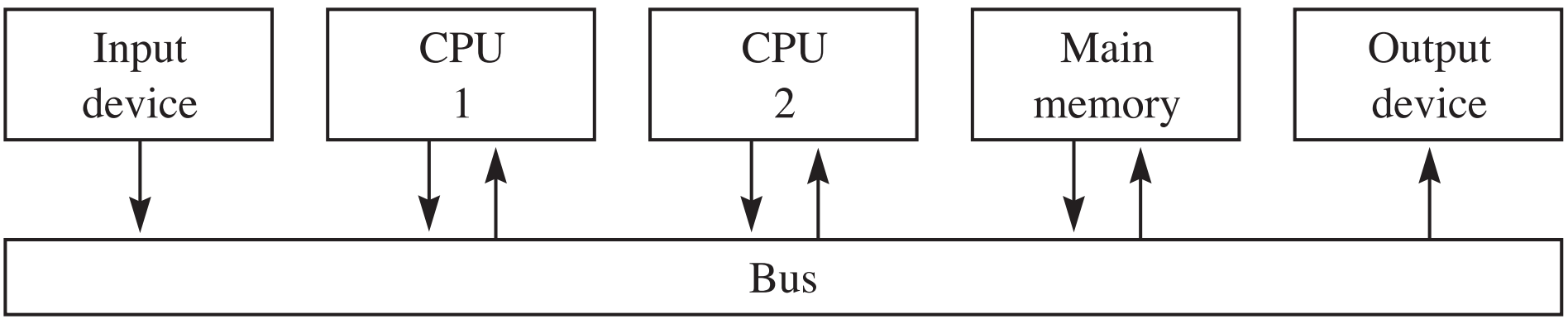
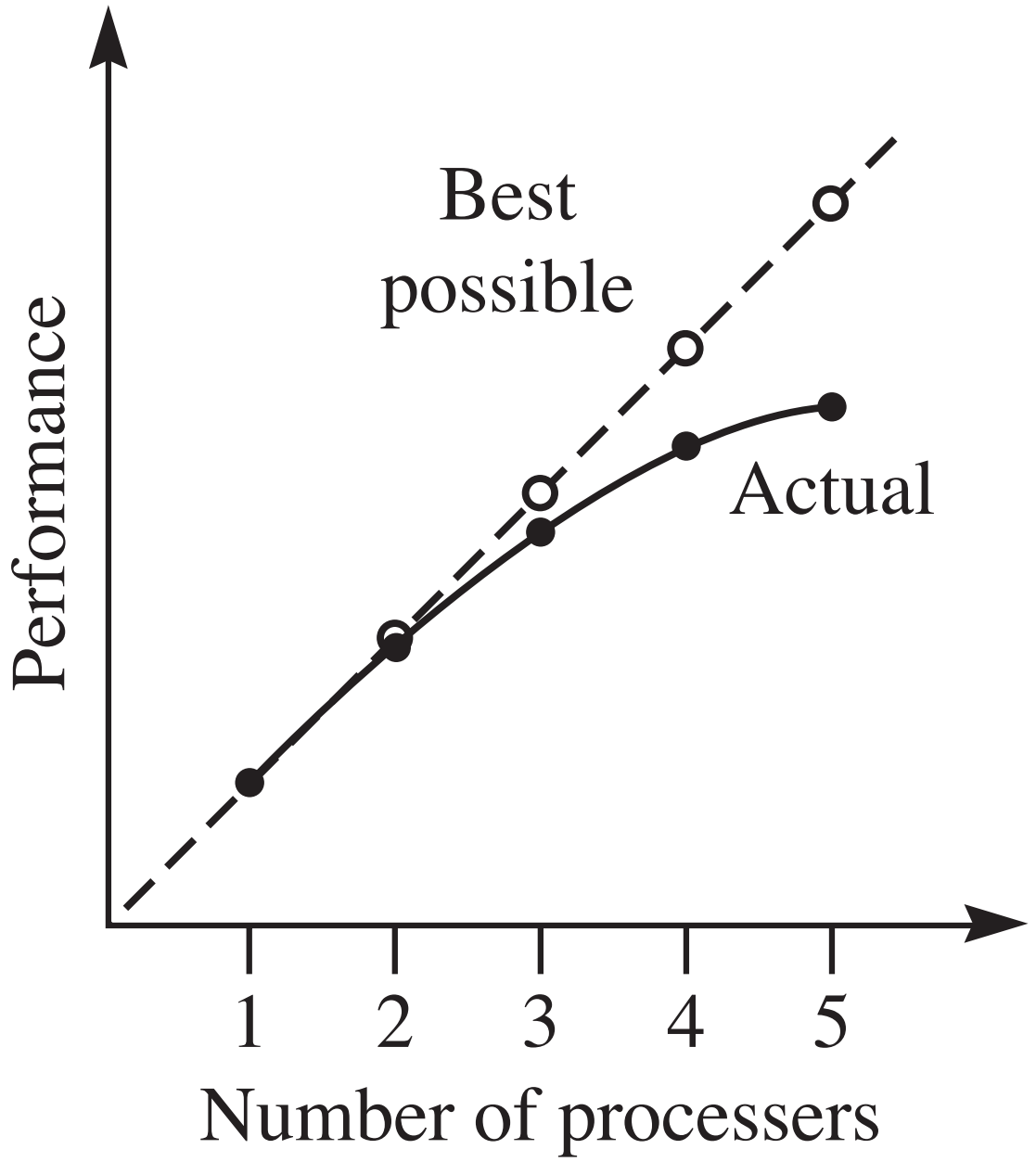


Figure 8.21



C++ Level

Process P1

```
...
NumRes++
...
```

Process P2

```
...
NumRes++
...
```

Assembly Level

Process P1

```
...
LDA   numRes,d
ADDA  1,i
STA   numRes,d
...
```

Process P2

```
...
LDA   numRes,d
ADDA  1,i
STA   numRes,d
...
```

Figure 8.23

Statement Executed		A (P1)	A (P2)	numRes
		?	?	47
(P1) LDA	numRes,d	47	?	47
(P1) ADDA	1,i	48	?	47
(P2) LDA	numRes,d	48	47	47
(P2) ADDA	1,i	48	48	47
(P2) STA	numRes,d	48	48	48
(P1) STA	numRes,d	48	48	48

Critical sections

- Critical sections are code sections in two processes that are mutually exclusive
- An entry section comes *before* a critical section to prevent illegal entry
- An exit section comes *after* a critical section to allow another process to enter its critical section
- A remainder section is not critical

Figure 8.24

Process P1

```
do
        entry section
        critical section
        exit section
        remainder section
while (!done1);
```

Process P2

```
do
        entry section
        critical section
        exit section
        remainder section
while (!done2)
```

A first attempt at mutual exclusion

- `turn` is a shared integer variable
- `turn` can be initialized to 1 or 2 before the processes begin executing

Figure 8.25

Process P1

```
do
    while (turn != 1) {
        ; //nothing
    critical section
    turn = 2;
    remainder section
while (!done1);
```

Process P2

```
do
    while (turn != 1) {
        ; //nothing
    critical section
    turn = 2;
    remainder section
while (!done2)
```

Behavior of first attempt at mutual exclusion

- Critical sections are mutually exclusive regardless of assembly language interleaving
- Processes must strictly alternate the bodies of their do loops

A second attempt at mutual exclusion

- `enter1` and `enter2` are two shared boolean variables
- `enter1` and `enter2` are both initialized to false before

Figure 8.26

Process P1

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

Process P2

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

Behavior of second attempt at mutual exclusion

- Critical sections are mutually exclusive regardless of assembly language interleaving
- Deadlock is possible

Figure 8.27

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

Deadlock

- Each process is waiting for an event that will never occur
- Deadlocks are conditions to avoid

Peterson's algorithm

- Use `enter1` and `enter2` to provide mutual exclusion
- Use `turn` to avoid deadlock

Figure 8.28

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)
```

Spin locks

- A spin lock is a loop whose only purpose is to stall the process before entering its critical section until it is (asynchronously) interrupted, allowing the other process to finish executing its critical section
- Spin locks waste CPU time

Semaphores

- A shared integer `s` with a queue of PCBs `sQueue`
- Semaphores enable the programmer to implement critical sections without spin locks
- Three atomic operations on a semaphore:

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

```
init(s)
s = 1;
sQueue = an empty list of PCBs
```

```
wait(s)
s--;
if (s < 0)
    Suspend this process, add to sQueue
```

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

Figure 8.29

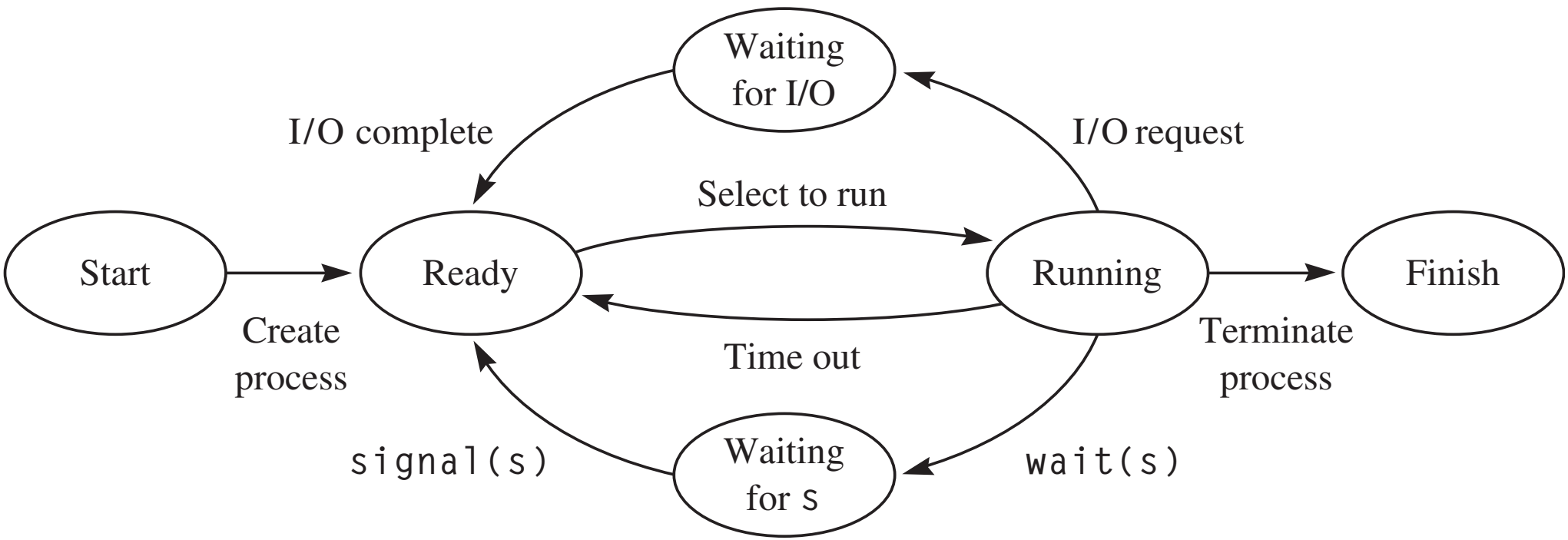


Figure 8.30

Process P1

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

Process P2

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

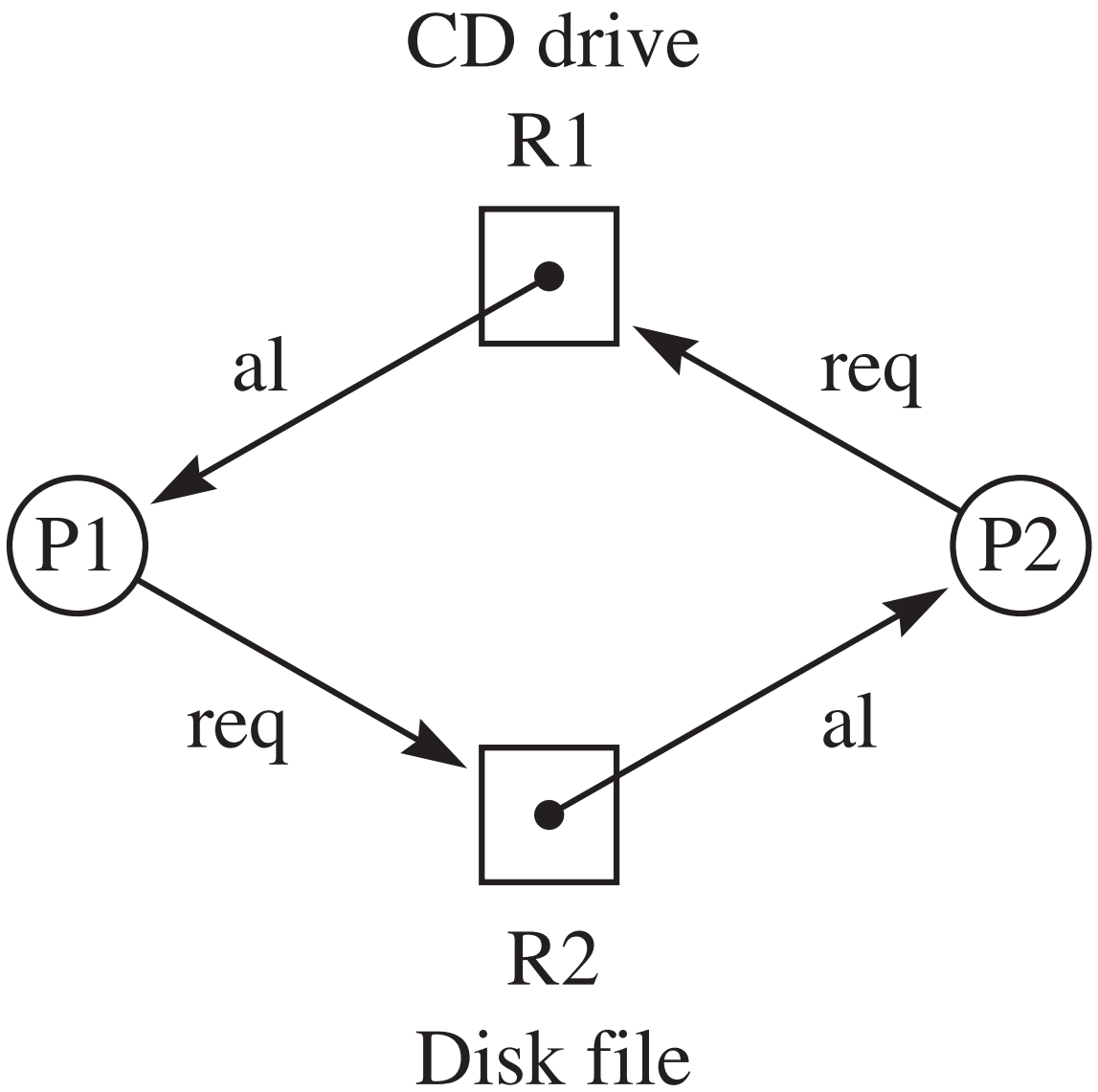
Negative semaphore values

- If `s` is negative, then one or more processes is blocked in `sQueue`
- The magnitude of `s` is the number of processes blocked

Resource allocation graphs

- A circle represents a process
- A solid dot inside a rectangle represents a resource
- An allocation edge from a resource to a process means the resource is allocated to the process
- A request edge from a process to a resource means the process is blocked waiting for the resource

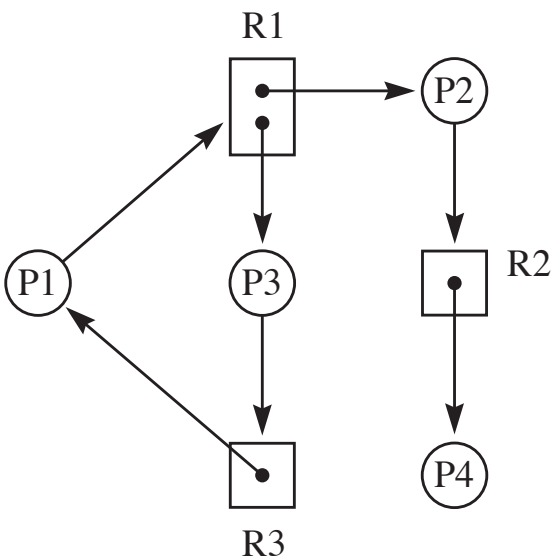
Figure 8.31



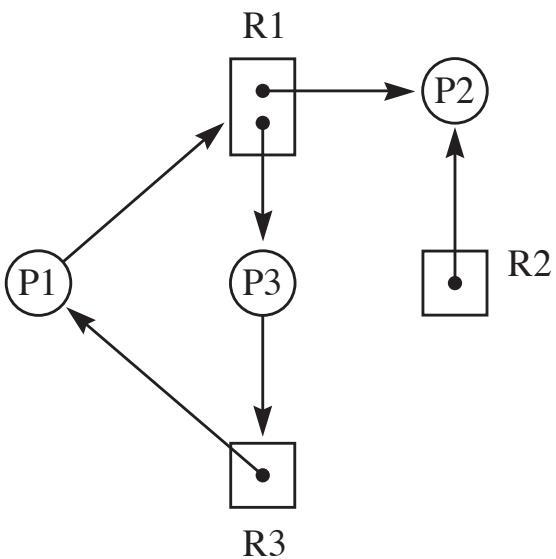
Detecting deadlock

- If a cycle in a resource allocation graph cannot be broken, there is deadlock
- A cycle is a necessary but not sufficient condition for deadlock

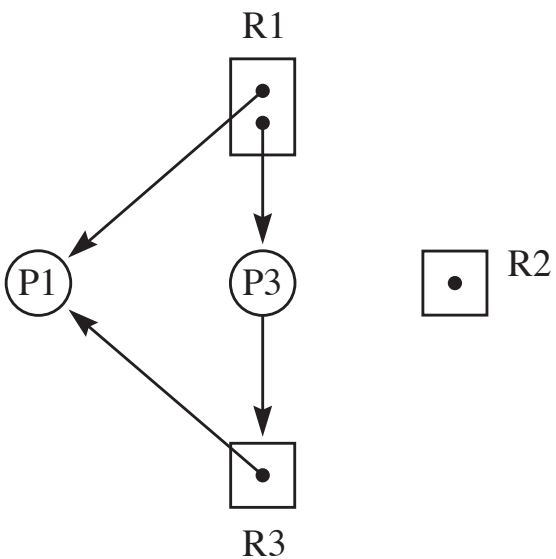
Figure 8.32



(a) Initial state.

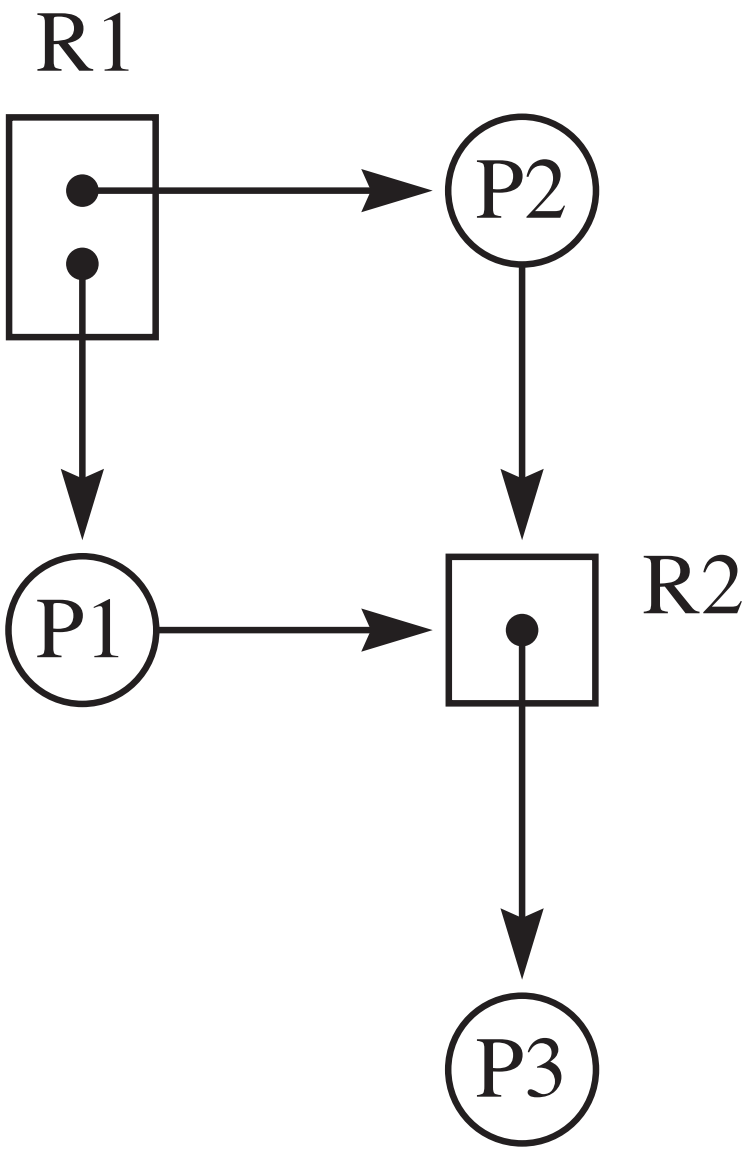


(b) P4 completes.



(c) P2 completes.

Figure 8.33



Deadlock policies

- Prevent
- Detect and recover
- Ignore