
Processes, Coroutines, and Concurrency Chapter 19

When most people speak of multitasking, they usually mean the ability to run several different application programs concurrently on one machine. Given the structure of the original 80x86 chips and MS-DOS' software design, this is very difficult to achieve when running DOS. Look at how long it's taken Microsoft to get Windows to multitask as well as it does.

Given the problems large companies like Microsoft have had trying to get multitasking to work, you might think that it is a very difficult thing to manage. However, this isn't true. Microsoft has problems trying to make different applications *that are unaware of one another* work harmoniously together. Quite frankly, they have not succeeded in getting existing DOS applications to multitask well. Instead, they've been working on developers to write new programs that work well under Windows.

Multitasking is not trivial, but it is not that difficult when you write an application with multitasking specifically in mind. You can even write programs that multitask under DOS if you only take a few precautions. In this chapter, we will discuss the concept of a DOS *process*, a *coroutine*, and a general *process*.

19.1 DOS Processes

Although MS-DOS is a single tasking operating system, this does not mean there can only be one program at a time in memory. Indeed, the whole purpose of the previous chapter was to describe how to get two or more programs operating in memory at one time. However, even if we ignore TSRs for the time being, you can still load several programs into memory at one time under DOS. The only catch is, DOS only provides the ability for them to run one at a time in a very specific fashion. Unless the processes are *cooperating*, their execution profile follows a very strict pattern.

19.1.1 Child Processes in DOS

When a DOS application is running, it can load and execute some other program using the DOS EXEC function (see "MS-DOS, PC-BIOS, and File I/O" on page 699). Under normal circumstances, when an application (the parent) runs a second program (the child), the child process executes to completion and then returns to the parent. This is very much like a procedure call, except it is a little more difficult to pass parameters between the two.

MS-DOS provides several functions you can use to load and execute program code, terminate processes, and obtain the exit status for a process. The following table lists many of these operations.

Table 67: DOS Character Oriented Functions

Function # (AH)	Input Parameters	Output Parameters	Description
4Bh	al- 0 ds:dx- pointer to program name. es:bx- pointer to LOADEXEC structure.	ax- error code if carry set.	Load and execute program
4Bh	al- 1 ds:dx- pointer to program name. es:bx- pointer to LOAD structure.	ax- error code if carry set.	Load program
4Bh	al- 3 ds:dx- pointer to program name. es:bx- pointer to OVERLAY structure.	ax- error code if carry set.	Load overlay

Table 67: DOS Character Oriented Functions

Function # (AH)	Input Parameters	Output Parameters	Description
4Ch	al- process return code		Terminate execution
4Dh		al- return value ah- termination method.	Get child process return value

19.1.1.1 Load and Execute

The “load and execute” call requires two parameters. The first, in `ds:dx`, is a pointer to a zero terminated string containing the pathname of the program to execute. This must be a “.COM” or “.EXE” file and the string must contain the program name’s extension. The second parameter, in `es:bx`, is a pointer to a `LOADEXEC` data structure. This data structure takes the following form:

```
LOADEXEC      struct
EnvPtr        word          ?           ;Pointer to environment area
CmdLinePtr    dword        ?           ;Pointer to command line
FCB1          dword        ?           ;Pointer to default FCB1
FCB2          dword        ?           ;Pointer to default FCB2
LOADEXEC      ends
```

`Envptr` is the segment address of the DOS *environment* block created for the new application. If this field contains a zero, DOS creates a copy of the current process’ environment block for the child process. If the program you are running does not access the environment block, you can save several hundred bytes to a few kilobytes by pointing the environment pointer field to a string of four zeros.

The `CmdLinePtr` field contains the address of the command line to supply to the program. DOS will copy this command line to offset `80h` in the new PSP it creates for the child process. A valid command line consists of a byte containing a character count, a least one space, any character belonging to the command line, and a terminating carriage return character (`0Dh`). The first byte should contain the length of the ASCII characters in the command line, not including the carriage return. If this byte contains zero, then the second byte of the command line should be the carriage return, not a space. Example:

```
MyCmdLine     byte          12, " file1 file2",cr
```

The `FCB1` and `FCB2` fields need to point at the two default *file control blocks* for this program. FCBs became obsolete with DOS 2.0, but Microsoft has kept FCBs around for compatibility anyway. For most programs you can point both of these fields at the following string of bytes:

```
DfltFCB       byte          3, " ",0,0,0,0,0
```

The load and execute call will fail if there is insufficient memory to load the child process. When you create an “.EXE” file using MASM, it creates an executable file that grabs all available memory, by default. Therefore, there will be *no* memory available for the child process and DOS will always return an error. Therefore, you must readjust the memory allocation for the parent process before attempting to run the child process. The section “Semiresident Programs” on page 1055 describes how to do this.

There are other possible errors as well. For example, DOS might not be able to locate the program name you specify with the zero terminated string. Or, perhaps, there are too many open files and DOS doesn’t have a free buffer available for the file I/O. If an error occurs, DOS returns with the carry flag set and an appropriate error code in the `ax` register. The following example program executes the “COMMAND.COM” program, allowing a user to execute DOS commands from inside your application. When the user types “exit” at the DOS command line, DOS returns control to your program.

```
; RUNDOS.ASM - Demonstrates how to invoke a copy of the COMMAND.COM
;             DOS command line interpreter from your programs.

include      stdlib.a
```

```

        includelib stdlib.lib

dseg          segment    para public 'data'

; MS-DOS EXEC structure.

ExecStruct    word       0                ;Use parent's Environment blk.
              dword     CmdLine          ;For the cmd ln parms.
              dword     DfltFCB
              dword     DfltFCB

DfltFCB       byte      3," ",0,0,0,0,0
CmdLine       byte      0, 0dh          ;Cmd line for program.
PgmName       dword     filename        ;Points at pgm name.

filename      byte      "c:\command.com",0

dseg          ends

cseg          segment    para public 'code'
              assume    cs:cseg, ds:dseg

Main          proc

              mov        ax, dseg        ;Get ptr to vars segment
              mov        ds, ax

              MemInit          ;Start the memory mgr.

; Okay, we've built the MS-DOS execute structure and the necessary
; command line, now let's see about running the program.
; The first step is to free up all the memory that this program
; isn't using. That would be everything from zzzzzzseg on.
;
; Note: unlike some previous examples in other chapters, it is okay
; to call Standard Library routines in this program after freeing
; up memory. The difference here is that the Standard Library
; routines are loaded early in memory and we haven't free up the
; storage they are sitting in.

              mov        ah, 62h        ;Get our PSP value
              int        21h
              mov        es, bx
              mov        ax, zzzzzzseg  ;Compute size of
              sub        ax, bx         ; resident run code.
              mov        bx, ax
              mov        ah, 4ah        ;Release unused memory.
              int        21h

; Tell the user what is going on:

              print
              byte      cr,lf
              byte      "RUNDOS- Executing a copy of command.com",cr,lf
              byte      "Type 'EXIT' to return control to RUN.ASM",cr,lf
              byte      0

; Warning! No Standard Library calls after this point. We've just
; released the memory that they're sitting in. So the program load
; we're about to do will wipe out the Standard Library code.

              mov        bx, seg ExecStruct
              mov        es, bx
              mov        bx, offset ExecStruct ;Ptr to program record.
              lds        dx, PgmName
              mov        ax, 4b00h      ;Exec pgm
              int        21h

; In MS-DOS 6.0 the following code isn't required. But in various older
; versions of MS-DOS, the stack is messed up at this point. Just to be
; safe, let's reset the stack pointer to a decent place in memory.
;
; Note that this code preserves the carry flag and the value in the
; AX register so we can test for a DOS error condition when we are done

```

```

; fixing the stack.

        mov     bx, sseg
        mov     ss, ax
        mov     sp, offset EndStk
        mov     bx, seg dseg
        mov     ds, bx

; Test for a DOS error:

        jnc     GoodCommand
        print
        byte   "DOS error #",0
        puti
        print
        byte   " while attempting to run COMMAND.COM",cr,lf
        byte   0
        jmp    Quit

; Print a welcome back message.

GoodCommand:  print
              byte   "Welcome back to RUNDOS. Hope you had fun.",cr,lf
              byte   "Now returning to MS-DOS' version of COMMAND.COM."
              byte   cr,lf,lf,0

; Return control to MS-DOS

Quit:       ExitPgm
Main       endp
cseg      ends

sseg      segment    para stack 'stack'
          dw         128 dup (0)
sseg      ends

zzzzzzseg segment    para public 'zzzzzzseg'
Heap     db         200h dup (?)
zzzzzzseg ends
end       Main

```

19.1.1.2 Load Program

The load and execute function gives the parent process very little control over the child process. Unless the child communicates with the parent process via a trap or interrupt, DOS suspends the parent process until the child terminates. In many cases the parent program may want to load the application code and then execute some additional operations before the child process takes over. Semiresident programs, appearing in the previous chapter, provide a good example. The DOS "load program" function provides this capability; it will load a program from the disk and return control back to the parent process. The parent process can do whatever it feels is appropriate before passing control to the child process.

The load program call requires parameters that are very similar to the load and execute call. Indeed, the only difference is the use of the LOAD structure rather than the LOADEXEC structure, and even these structures are very similar to one another. The LOAD data structure includes two extra fields not present in the LOADEXE structure:

```

LOAD      struct
EnvPtr    word        ?           ;Pointer to environment area.
CmdLinePtr dword      ?           ;Pointer to command line.
FCB1      dword      ?           ;Pointer to default FCB1.
FCB2      dword      ?           ;Pointer to default FCB2.
SSSP      dword      ?           ;SS:SP value for child process.
CSIP      dword      ?           ;Initial program starting point.
LOAD      ends

```

The LOAD command is useful for many purposes. Of course, this function provides the primary vehicle for creating semiresident programs; however, it is also quite useful for providing extra error recovery,

redirecting application I/O, and loading several executable processes into memory for concurrent execution.

After you load a program using the DOS load command, you can obtain the PSP address for that program by issuing the DOS get PSP address call (see “MS-DOS, PC-BIOS, and File I/O” on page 699). This would allow the parent process to modify any values appearing in the child process’ PSP prior to its execution. DOS stores the termination address for a procedure in the PSP. This termination address normally appears in the double word at offset 10h in the PSP. *If you do not change this location, the program will return to the first instruction beyond the int 21h instruction for the load function.* Therefore, before actually transferring control to the user application, you should change this termination address.

19.1.1.3 Loading Overlays

Many programs contain blocks of code that are independent of one other; that is, while routines in one block of code execute, the program will not call routines in the other independent blocks of code. For example, a modern game may contain some initialization code, a “staging area” where the user chooses certain options, an “action area” where the user plays the game, and a “debriefing area” that goes over the player’s actions. When running in a 640K MS-DOS machine, all this code may not fit into available memory at the same time. To overcome this memory limitation, most large programs use *overlays*. An overlay is a portion of the program code that shares memory for its code with other code modules. The DOS load overlay function provides support for large programs that need to use overlays.

Like the load and load/execute functions, the load overlay expects a pointer to the code file’s pathname in the ds:dx register pair and the address of a data structure in the es:bx register pair. This overlay data structure has the following format:

```
overlay      struct
StartSeg    word      ?
RelocFactor word      0
overlay     ends
```

The StartSeg field contains the segment address where you want DOS to load the program. The RelocFactor field contains a relocation factor. This value should be zero unless you want the starting offset of the segment to be something other than zero.

19.1.1.4 Terminating a Process

The process termination function is nothing new to you by now, you’ve used this function over and over again already if you written any assembly language programs and run them under DOS (the Standard Library ExitPgm macro executes this command). In this section we’ll look at exactly what the terminate process function call does.

First of all, the terminate process function gives you the ability to pass a single byte *termination code* back to the parent process. Whatever value you pass in al to the terminate call becomes the return, or termination code. The parent process can test this value using the Get Child Process Return Value call (see the next section). You can also test this return value in a DOS batch file using the “if errorlevel” statement.

The terminate process command does the following:

- Flushes file buffers and closes files.
- Restores the termination address (int 22h) from offset 0Ah in the PSP (this is the return address of the process).
- Restores the address of the Break handler (int 23h) from offset 0Eh in the PSP (see “Exception Handling in DOS: The Break Handler” on page 1070)
- Restores the address of the critical error handler (int 24h) from offset 12h in the PSP (see “Exception Handling in DOS: The Critical Error Handler” on page 1071).

- Deallocates any memory held by the process.

Unless you *really* know what you're doing, you should not change the values at offsets 0Ah, 0Eh, or 12h in the PSP. By doing so you could produce an inconsistent system when your program terminates.

19.1.1.5 Obtaining the Child Process Return Code

A parent process can obtain the return code from a child process by making the DOS Get Child Process Return Code function call. This call returns the value in the al register at the point of termination plus information that tells you how the child process terminated.

This call (ah=4Dh) returns the termination code in the al register. It also returns the cause of termination in the ah register. The ah register will contain one of the following values:

Table 68: Termination Cause

Value in AH	Reason for Termination
0	Normal termination (int 21h, ah=4Ch)
1	Terminated by ctrl-C
2	Terminated by critical error
3	TSR termination (int 21h, ah=31h)

The termination code appearing in al is valid only for normal and TSR terminations.

Note that you can only call this routine *once* after a child process terminates. MS-DOS returns meaningless values in AX after the first such call. Likewise, if you use this function without running a child process, the results you obtain will be meaningless. DOS does not return if you do this.

19.1.2 Exception Handling in DOS: The Break Handler

Whenever the user presses a ctrl-C or ctrl-Break key MS-DOS may trap such a key sequence and execute an int 23h instruction¹. MS-DOS provides a default break handler routine that terminates the program. However, a well-written program generally replaces the default break handler with one of its own so it can capture ctrl-C or ctrl-break key sequences and shut the program down in an orderly fashion.

When DOS terminates a program due to a break interrupt, it flushes file buffers, closes all open files, releases memory belonging to the application, all the normal stuff it does on program termination. However, it does *not* restore any interrupt vectors (other than interrupt 23h and interrupt 24h). If your code has replaced any interrupt vectors, especially hardware interrupt vectors, then those vectors will still be pointing at your program's interrupt service routines after DOS terminates your program. This will probably crash the system when DOS loads a new program over the top of your code. Therefore, you should write a break handler so your application can shut itself down in an orderly fashion if the user presses ctrl-C or ctrl-break.

The easiest, and perhaps most universal, break handler consists of a single instruction – iret. If you point the interrupt 23h vector at an iret instruction, MS-DOS will simply ignore any ctrl-C or ctrl-break keys you press. This is very useful for turning off the break handling during critical sections of code that you do not want the user to interrupt.

1. MS-DOS always executes an int 23h instruction if it is processing a function code in the range 1-0Ch. For other DOS functions, MS-DOS only executes int 23h if the Break flag is set

On the other hand, simply turning off ctrl-C and ctrl-break handling throughout your entire program is not satisfactory either. If for some reason the user wants to abort your program, pressing ctrl-break or ctrl-C is what they will probably try to do this. If your program disallows this, the user may resort to something more drastic like ctrl-alt-delete to reset the machine. This will certainly mess up any open files and may cause other problems as well (of course, you don't have to worry about restoring any interrupt vectors!).

To patch in your own break handler is easy – just store the address of your break handler routine into the interrupt vector 23h. You don't even have to save the old value, DOS does this for you automatically (it stores the original vector at offset 0Eh in the PSP). Then, when the users presses a ctrl-C or ctrl-break key, MS-DOS transfers control to your break handler.

Perhaps the best response for a break handler is to set some flag to tell the application and break occurred, and then leave it up to the application to test this flag a reasonable points to determine if it should shut down. Of course, this does require that you test this flag at various points throughout your application, increasing the complexity of your code. Another alternative is to save the original int 23h vector and transfer control to DOS' break handler after you handle important operations yourself. You can also write a specialized break handler to return a DOS termination code that the parent process can read.

Of course, there is no reason you cannot change the interrupt 23h vector at various points throughout your program to handle changing requirements. At various points you can disable the break interrupt entirely, restore interrupt vectors at others, or prompt the user at still other points.

19.1.3 Exception Handling in DOS: The Critical Error Handler

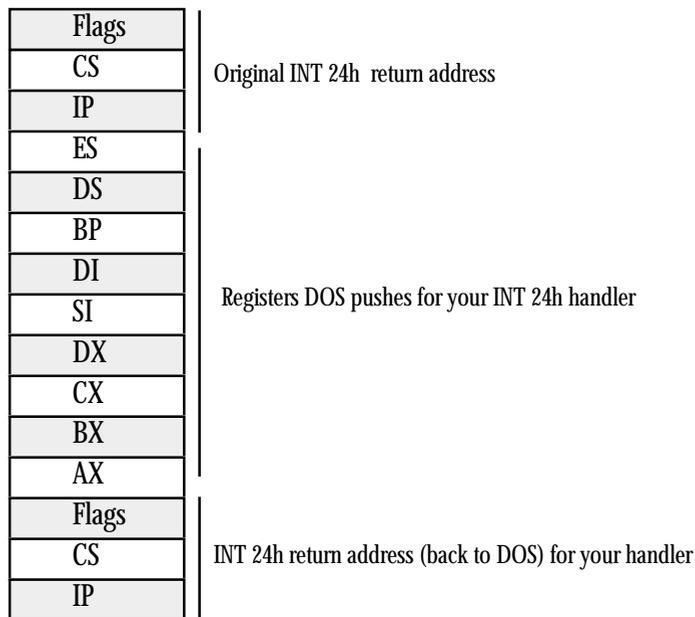
DOS invokes the critical error handler by executing an int 24h instruction whenever some sort of I/O error occurs. The default handler prints the familiar message:

```
I/O Device Specific Error Message
Abort, Retry, Ignore, Fail?
```

If the user presses an "A", this code immediately returns to DOS' COMMAND.COM program; *it doesn't even close any open files*. If the user presses an "R" to retry, MS-DOS will retry the I/O operation, though this usually results in another call to the critical error handler. The "I" option tells MS-DOS to ignore the error and return to the calling program as though nothing had happened. An "F" response instructs MS-DOS to return an error code to the calling program and let it handle the problem.

Of the above options, having the user press "A" is the most dangerous. This causes an immediate return to DOS and your code does not get the chance to clean up anything. For example, if you've patched some interrupt vectors, your program will not get the opportunity to restore them if the user selects the abort option. This may crash the system when MS-DOS loads the next program over the top of your interrupt service routine(s) in memory.

To intercept DOS critical errors, you will need to patch the interrupt 24h vector to point at your own interrupt service routine. Upon entry into your interrupt 24h service routine, the stack will contain the following data:



Stack Contents Upon Entry to a Critical Error Handler

MS-DOS passes important information in several of the registers to your critical error handler. By inspecting these values you can determine the cause of the critical error and the device on which it occurred. The high order bit of the ah register determines if the error occurred on a block structured device (typically a disk or tape) or a character device. The other bits in ah have the following meaning:

Table 69: Device Error Bits in AH

Bit(s)	Description
0	0=Read operation. 1=Write operation.
1-2	Indicates affected disk area. 00- MS-DOS area. 01- File allocation table (FAT). 10- Root directory. 11- Files area.
3	0- Fail response not allowed. 1- Fail response is okay.
4	0- Retry response not allowed. 1- Retry response is okay.
5	0- Ignore response is not allowed. 1- Ignore response is okay.
6	Undefined
7	0- Character device error. 1- Block structured device error.

In addition to the bits in `ah`, for block structured devices the `a1` register contains the drive number where the error occurred (0=A, 1=B, 2=C, etc.). The value in the `a1` register is undefined for character devices.

The lower half of the `di` register contains additional information about the block device error (the upper byte of `di` is undefined, you will need to mask out those bits before attempting to test this data).

Table 70: Block Structured Device Error Codes (in L.O. byte of DI)

Error Code	Description
0	Write protection error.
1	Unknown drive.
2	Drive not ready.
3	Invalid command.
4	Data error (CRC error).
5	Length of request structure is incorrect.
6	Seek error on device.
7	Disk is not formatted for MS-DOS.
8	Sector not found.
9	Printer out of paper.
0Ah	Write error.
0Bh	Read error.
0Ch	General failure.
0Fh	Disk was changed at inappropriate time.

Upon entry to your critical error handler, interrupts are turned off. Because this error occurs as a result of some MS-DOS call, MS-DOS is already entered and you will not be able to make any calls other than functions 1-0Ch and 59h (get extended error information).

Your critical error handler must preserve all registers except `a1`. The handler must return to DOS with an `iret` instruction and `a1` must contain one of the following codes:

Table 71: Critical Error Handler Return Codes

Code	Meaning
0	Ignore device error.
1	Retry I/O operation again.
2	Terminate process (abort).
3	Fail current system call.

The following code provides a trivial example of a critical error handler. The main program attempts to send a character to the printer. If you do not connect a printer, or turn off the printer before running this program, it will generate the critical error.

```
; Sample INT 24h critical error handler.
;
; This code demonstrates a sample critical error handler.
; It patches into INT 24h and displays an appropriate error
; message and asks the user if they want to retry, abort, ignore,
; or fail (just like DOS).
```

```

        .xlist
        include    stdlib.a
        includelib stdlib.lib
        .list

dseg          segment    para public 'data'

Value        word       0
ErrCode      word       0

dseg          ends

cseg          segment    para public 'code'
              assume     cs:cseg, ds:dseg

; A replacement critical error handler. Note that this routine
; is even worse than DOS', but it demonstrates how to write
; such a routine. Note that we cannot call any Standard Library
; I/O routines in the critical error handler because they do not
; use DOS calls 1-0Ch, which are the only allowable DOS calls at
; this point.

CritErrMsg   byte       cr,lf
              byte       "DOS Critical Error!",cr,lf
              byte       "A)bort, R)etry, I)gnore, F)ail? $"

MyInt24      proc        far
              push       dx
              push       ds
              push       ax

              push       cs
              pop        ds
Int24Lp:     lea         dx, CritErrMsg
              mov        ah, 9           ;DOS print string call.
              int        21h

              mov        ah, 1           ;DOS read character call.
              int        21h
              and        al, 5Fh        ;Convert l.c. -> u.c.

              cmp        al, 'I'        ;Ignore?
              jne        NotIgnore
              pop        ax
              mov        al, 0
              jmp        Quit24

NotIgnore:   cmp        al, 'r'        ;Retry?
              jne        NotRetry
              pop        ax
              mov        al, 1
              jmp        Quit24

NotRetry:   cmp        al, 'A'        ;Abort?
              jne        NotAbort
              pop        ax
              mov        al, 2
              jmp        Quit24

NotAbort:   cmp        al, 'F'
              jne        BadChar
              pop        ax

Quit24:     pop        ds
              pop        dx
              iret

BadChar:    mov        ah, 2
              mov        dl, 7         ;Bell character
              jmp        Int24Lp

MyInt24     endp

```

```

Main      proc
          mov     ax, dseg
          mov     ds, ax
          mov     es, ax
          meminit

          mov     ax, 0
          mov     es, ax
          mov     word ptr es:[24h*4], offset MyInt24
          mov     es:[24h*4 + 2], cs

          mov     ah, 5
          mov     dl, 'a'
          int     21h
          rcl     Value, 1
          and     Value, 1
          mov     ErrCode, ax
          printf
          byte    cr,lf,lf
          byte    "Print char returned with error status %d and "
          byte    "error code %d\n",0
          dword   Value, ErrCode

Quit:     ExitPgm                ;DOS macro to quit program.
Main     endp

cseg     ends

; Allocate a reasonable amount of space for the stack (8k).
; Note: if you use the pattern matching package you should set up a
;       somewhat larger stack.

sseg     segment    para stack 'stack'
stk      db         1024 dup ("stack ")
sseg     ends

; zzzzzzseg must be the last segment that gets loaded into memory!
; This is where the heap begins.

zzzzzzseg segment    para public 'zzzzzz'
LastBytes db         16 dup (?)
zzzzzzseg ends
end      Main

```

19.1.4 Exception Handling in DOS: Traps

In addition to the break and critical error exceptions, there are the 80x86 exceptions that can happen during the execution of your programs. Examples include the divide error exception, bounds exception, and illegal opcode exception. A well-written application will always handle all possible exceptions.

DOS does not provide direct support for these exceptions, other than a possible default handler. In particular, DOS does not restore such vectors when the program terminates; this is something the application, break handler, and critical error handler must take care of. For more information on these exceptions, see “Exceptions” on page 1000.

19.1.5 Redirection of I/O for Child Processes

When a child process begins execution, it inherits all open files from the parent process (with the exception of certain files opened with networking file functions). In particular, this includes the default

files opened for the DOS *standard input*, *standard output*, *standard error*, *auxiliary*, and *printer* devices. DOS assigns the file handle values zero through four, respectively, to these devices. If a parent process closes one of these file handles and then reassigns the handle with a Force Duplicate File Handle call.

Note that the DOS EXEC call does not process the I/O redirection operators (“<”, and “>”, and “|”). If you want to redirect the standard I/O of a child process, you must do this before loading and executing the child process. To redirect one of the five standard I/O devices, you should do the following steps:

- 1) Duplicate the file handle you want to redirect (e.g., to redirect the standard output, duplicate file handle one).
- 2) Close the affected file (e.g., file handle one for standard output).
- 3) Open a file using the standard DOS Create or CreateNew calls.
- 4) Use the Force Duplicate File Handle call to copy the new file handle to file handle one.
- 5) Run the child process.
- 6) On return from the child, close the file.
- 7) Copy the file handle you duplicated in step one back to the standard output file handle using the Force Duplicate Handle function.

This technique looks like it would be perfect for redirecting printer or serial port I/O. Unfortunately, many programs bypass DOS when sending data to the printer and use the BIOS call or, worse yet, go directly to the hardware. Almost no software bothers with DOS’ serial port support – it truly is that bad. However, most programs *do* call DOS to input or output characters on the standard input, output, and error devices. The following code demonstrates how to redirect the output of a child process to a file.

```
; REDIRECT.ASM -Demonstrates how to redirect I/O for a child process.
; This particular program invokes COMMAND.COM to execute
; a DIR command, when is sent to the specified output file.

                include      stdlib.a
                includelib  stdlib.lib

dseg           segment      para public 'data'

OrigOutHandle  word         ?                ;Holds copy of STDOUT handle.
FileHandle     word         ?                ;File I/O handle.
FileName       byte         "dirctry.txt",0  ;Filename for output data.

; MS-DOS EXEC structure.

ExecStruct     word         0                ;Use parent's Environment blk.
               dword       CmdLine          ;For the cmd ln parms.
               dword       DfltFCB
               dword       DfltFCB

DfltFCB       byte         3," ",0,0,0,0,0
CmdLine       byte         7," /c DIR", 0dh ;Do a directory command.
PgmName       dword       PgmNameStr       ;Points at pgm name.
PgmNameStr    byte         "c:\command.com",0
dseg          ends

cseg           segment      para public 'code'
               assume      cs:cseg, ds:dseg

Main          proc
               mov         ax, dseg         ;Get ptr to vars segment
               mov         ds, ax
               MemInit      ;Start the memory mgr.

; Free up some memory for COMMAND.COM:

               mov         ah, 62h         ;Get our PSP value
               int         21h
```

```

        mov     es, bx
        mov     ax, zzzzzzseg    ;Compute size of
        sub     ax, bx          ; resident run code.
        mov     bx, ax
        mov     ah, 4ah         ;Release unused memory.
        int     21h

; Save original output file handle.

        mov     bx, 1           ;Std out is file handle 1.
        mov     ah, 45h         ;Duplicate the file handle.
        int     21h
        mov     OrigOutHandle, ax;Save duplicate handle.

; Open the output file:

        mov     ah, 3ch         ;Create file.
        mov     cx, 0           ;Normal attributes.
        lea     dx, FileName
        int     21h
        mov     FileHandle, ax  ;Save opened file handle.

; Force the standard output to send its output to this file.
; Do this by forcing the file's handle onto file handle #1 (stdout).

        mov     ah, 46h         ;Force dup file handle
        mov     cx, 1           ;Existing handle to change.
        mov     bx, FileHandle  ;New file handle to use.
        int     21h

; Print the first line to the file:

        print
        byte    "Redirected directory listing:", cr, lf, 0

; Okay, execute the DOS DIR command (that is, execute COMMAND.COM with
; the command line parameter "/c DIR").

        mov     bx, seg ExecStruct
        mov     es, bx
        mov     bx, offset ExecStruct ;Ptr to program record.
        lds     dx, PgmName
        mov     ax, 4b00h        ;Exec pgm
        int     21h

        mov     bx, sseg         ;Reset the stack on return.
        mov     ss, ax
        mov     sp, offset EndStk
        mov     bx, seg dseg
        mov     ds, bx

; Okay, close the output file and switch standard output back to the
; console.

        mov     ah, 3eh         ;Close output file.
        mov     bx, FileHandle
        int     21h

        mov     ah, 46h         ;Force duplicate handle
        mov     cx, 1           ;StdOut
        mov     bx, OrigOutHandle ;Restore previous handle.
        int     21h

; Return control to MS-DOS

Quit:    ExitPgm
Main     endp
cseg     ends

sseg     segment    para stack 'stack'
        dw         128 dup (0)
endstk   dw         ?
sseg     ends

```

```

zzzzzseg    segment    para public 'zzzzzseg'
Heap        db         200h dup (?)
zzzzzseg    ends
end         Main

```

19.2 Shared Memory

The only problem with running different DOS programs as part of a single application is *interprocess communication*. That is, how do all these programs talk to one other? When a typical DOS application runs, DOS loads in all code and data segments; there is no provision, other than reading data from a file or the process termination code, for one process to pass information to another. Although file I/O will work, it is cumbersome and slow. The ideal solution would be for one process to leave a copy of various variables that other processes can share. Your programs can easily do this using *shared memory*.

Most modern multitasking operating systems provide for shared memory – memory that appears in the address space of two or more processes. Furthermore, such shared memory is often *persistent*, meaning it continues to hold values after its creator process terminates. This allows other processes to start later and use the values left behind by the shared variables' creator.

Unfortunately, MS-DOS is not a modern multitasking operating system and it does not support shared memory. However, we can easily write a resident program that provides this capability missing from DOS. The following sections describe how to create two types of shared memory regions – static and dynamic.

19.2.1 Static Shared Memory

A TSR to implement *static shared memory* is trivial. It is a passive TSR that provides three functions – verify presence, remove, and return segment pointer. The transient portion simply allocates a 64K data segment and then terminates. Other processes can obtain the address of the 64K shared memory block by making the “return segment pointer” call. These processes can place all their shared data into the segment belonging to the TSR. When one process quits, the shared segment remains in memory as part of the TSR. When a second process runs and links with the shared segment, the variables from the shared segment are still intact, so the new process can access those values. When all processes are done sharing data, the user can remove the shared memory TSR with the remove function.

As mentioned above, there is almost nothing to the shared memory TSR. The following code implements it:

```

; SHARDMEM.ASM
;
; This TSR sets aside a 64K shared memory region for other processes to use.
;
; Usage:
;
;     SHARDMEM -           Loads resident portion and activates
;                        shared memory capabilities.
;
;     SHARDMEM REMOVE -   Removes shared memory TSR from memory.
;
; This TSR checks to make sure there isn't a copy already active in
; memory. When removing itself from memory, it makes sure there are
; no other interrupts chained into INT 2Fh before doing the remove.
;
;
; The following segments must appear in this order and before the
; Standard Library includes.

ResidentSeg    segment    para public 'Resident'
ResidentSeg    ends

SharedMemory    segment    para public 'Shared'

```

```

SharedMemory ends

EndResident segment para public 'EndRes'
EndResident ends

        .xlist
        .286
        include  stdlib.a
        includelib stdlib.lib
        .list

; Resident segment that holds the TSR code:

ResidentSeg segment para public 'Resident'
            assume  cs:ResidentSeg, ds:nothing

; Int 2Fh ID number for this TSR:

MyTSRID    byte    0
            byte    0                ;Padding so we can print it.

; PSP is the psp address for this program.

PSP        word    0

OldInt2F   dword   ?

; MyInt2F- Provides int 2Fh (multiplex interrupt) support for this
;          TSR. The multiplex interrupt recognizes the following
;          subfunctions (passed in AL):
;
;          00h- Verify presence. Returns 0FFh in AL and a pointer
;          to an ID string in es:di if the
;          TSR ID (in AH) matches this
;          particular TSR.
;
;          01h- Remove. Removes the TSR from memory.
;          Returns 0 in AL if successful,
;          1 in AL if failure.
;
;          10h- Return Seg Adrs. Returns the segment address of the
;          shared segment in ES.

MyInt2F    proc     far
            assume  ds:nothing

            cmp     ah, MyTSRID        ;Match our TSR identifier?
            je      YepItsOurs
            jmp     OldInt2F

; Okay, we know this is our ID, now check for a verify, remove, or
; return segment call.

YepItsOurs: cmp     al, 0                ;Verify Call
            jne     TryRmv
            mov     al, 0ffh            ;Return success.
            lesi   IDString
            ired                    ;Return back to caller.

IDString   byte    "Static Shared Memory TSR",0

TryRmv:    cmp     al, 1                ;Remove call.
            jne     TryRetSeg

; See if we can remove this TSR:

            push   es
            mov    ax, 0
            mov    es, ax
            cmp    word ptr es:[2Fh*4], offset MyInt2F
            jne    TRDone
            cmp    word ptr es:[2Fh*4 + 2], seg MyInt2F

```

```

                                je          CanRemove;Branch if we can.
TRDone:                        mov          ax, 1          ;Return failure for now.
                                pop          es
                                iredt

; Okay, they want to remove this guy *and* we can remove it from memory.
; Take care of all that here.

                                assume      ds:ResidentSeg

CanRemove:                    push        ds
                                pusha
                                cli          ;Turn off the interrupts while
                                mov          ax, 0          ; we mess with the interrupt
                                mov          es, ax        ; vectors.
                                mov          ax, cs
                                mov          ds, ax

                                mov          ax, word ptr OldInt2F
                                mov          es:[2Fh*4], ax
                                mov          ax, word ptr OldInt2F+2
                                mov          es:[2Fh*4 + 2], ax

; Okay, one last thing before we quit- Let's give the memory allocated
; to this TSR back to DOS.

                                mov          ds, PSP
                                mov          es, ds:[2Ch] ;Ptr to environment block.
                                mov          ah, 49h      ;DOS release memory call.
                                int          21h

                                mov          ax, ds      ;Release program code space.
                                mov          es, ax
                                mov          ah, 49h
                                int          21h

                                popa
                                pop          ds
                                pop          es
                                mov          ax, 0      ;Return Success.
                                iredt

; See if they want us to return the segment address of our shared segment
; here.

TryRetSeg:                    cmp          al, 10h          ;Return Segment Opcode
                                jne IllegalOp
                                mov          ax, SharedMemory
                                mov          es, ax
                                mov          ax, 0      ;Return success
                                cld
                                iredt

; They called us with an illegal subfunction value. Try to do as little
; damage as possible.

IllegalOp:                    mov          ax, 0          ;Who knows what they were thinking?
                                iredt
MyInt2F                        endp
ResidentSeg                    assume      ds:nothing
                                ends

; Here's the segment that will actually hold the shared data.

SharedMemory                    segment      para public 'Shared'
                                db          0FFFFh dup (?)
SharedMemory                    ends

cseg                            segment      para public 'code'
                                assume      cs:cseg, ds:ResidentSeg

```

```

; SeeIfPresent-      Checks to see if our TSR is already present in memory.
;                   Sets the zero flag if it is, clears the zero flag if
;                   it is not.

SeeIfPresent  proc      near
               push     es
               push     ds
               push     di
               mov      cx, 0ffh          ;Start with ID 0FFh.
IDLoop:       mov      ah, cl
               push     cx
               mov      al, 0            ;Verify presence call.
               int      2Fh
               pop      cx
               cmp      al, 0            ;Present in memory?
               je       TryNext
               strcml  byte    "Static Shared Memory TSR",0
               je       Success

TryNext:      dec      cl                ;Test USER IDs of 80h..FFh
               js      IDLoop
               cmp      cx, 0            ;Clear zero flag.
Success:      pop      di
               pop      ds
               pop      es
               ret
SeeIfPresent  endp

; FindID-           Determines the first (well, last actually) TSR ID available
;                   in the multiplex interrupt chain. Returns this value in
;                   the CL register.
;
;                   Returns the zero flag set if it locates an empty slot.
;                   Returns the zero flag clear if failure.

FindID       proc      near
               push     es
               push     ds
               push     di

IDLoop:       mov      cx, 0ffh          ;Start with ID 0FFh.
               mov      ah, cl
               push     cx
               mov      al, 0            ;Verify presence call.
               int      2Fh
               pop      cx
               cmp      al, 0            ;Present in memory?
               je       Success
               dec      cl                ;Test USER IDs of 80h..FFh
               js      IDLoop
               xor      cx, cx
               cmp      cx, 1            ;Clear zero flag
Success:      pop      di
               pop      ds
               pop      es
               ret
FindID       endp

Main         proc      meminit

               mov      ax, ResidentSeg
               mov      ds, ax

               mov      ah, 62h          ;Get this program's PSP
               int      21h              ; value.
               mov      PSP, bx

; Before we do anything else, we need to check the command line

```

```
; parameters. If there is one, and it is the word "REMOVE", then remove
; the resident copy from memory using the multiplex (2Fh) interrupt.
```

```

        argc
        cmp     cx, 1           ;Must have 0 or 1 parms.
        jb     TstPresent
        je     DoRemove
Usage:  print
        byte   "Usage:",cr,lf
        byte   " shardmem",cr,lf
        byte   "or shardmem REMOVE",cr,lf,0
        ExitPgm

; Check for the REMOVE command.

DoRemove:  mov     ax, 1
           argv
           stricmp  "REMOVE",0
           jne     Usage

           call    SeeIfPresent
           je     RemoveIt
           print
           byte   "TSR is not present in memory, cannot remove"
           byte   cr,lf,0
           ExitPgm

RemoveIt:  mov     MyTSRID, cl
           printf
           byte   "Removing TSR (ID #%d) from memory...",0
           dword  MyTSRID

           mov     ah, cl
           mov     al, 1         ;Remove cmd, ah contains ID
           int     2Fh
           cmp     al, 1         ;Succeed?
           je     RmvFailure
           print
           byte   "removed.",cr,lf,0
           ExitPgm

RmvFailure:  print
            byte   cr,lf
            byte   "Could not remove TSR from memory.",cr,lf
            byte   "Try removing other TSRs in the reverse order "
            byte   "you installed them.",cr,lf,0
            ExitPgm

; Okay, see if the TSR is already in memory. If so, abort the
; installation process.

TstPresent:  call    SeeIfPresent
            jne     GetTSRID
            print
            byte   "TSR is already present in memory.",cr,lf
            byte   "Aborting installation process",cr,lf,0
            ExitPgm

; Get an ID for our TSR and save it away.

GetTSRID:   call    FindID
            je     GetFileName
            print
            byte   "Too many resident TSRs, cannot install",cr,lf,0
            ExitPgm

; Things look cool so far, so install the interrupts
```

```

GetFileName:  mov     MyTSRID, c1
              print
              byte   "Installing interrupts...",0

; Patch into the INT 2Fh interrupt chain.

              cli                    ;Turn off interrupts!
              mov     ax, 0
              mov     es, ax
              mov     ax, es:[2Fh*4]
              mov     word ptr OldInt2F, ax
              mov     ax, es:[2Fh*4 + 2]
              mov     word ptr OldInt2F+2, ax
              mov     es:[2Fh*4], offset MyInt2F
              mov     es:[2Fh*4+2], seg ResidentSeg
              sti                    ;Okay, ints back on.

; We're hooked up, the only thing that remains is to zero out the shared
; memory segment and then terminate and stay resident.

              printf
              byte   "Installed, TSR ID %#d.",cr,lf,0
              dword  MyTSRID

              mov     ax, SharedMemory ;Zero out the shared
              mov     es, ax           ; memory segment.
              mov     cx, 32768       ;32K words = 64K bytes.
              xor     ax, ax          ;Store all zeros,
              mov     di, ax          ; starting at offset zero.
              rep    stosw

              mov     dx, EndResident ;Compute size of program.
              sub     dx, PSP
              mov     ax, 3100h       ;DOS TSR command.
              int     21h

Main
cseg        endp
           ends

sseg       segment  para stack 'stack'
stk        db      256 dup (?)
sseg       ends

zzzzzzseg segment  para public 'zzzzzz'
LastBytes db      16 dup (?)
zzzzzzseg ends
end        Main

```

This program simply carves out a chunk of memory (the 64K in the SharedMemory segment) and returns a pointer to it in es whenever a program executes the appropriate int 2Fh call (ah= TSR ID and al=10h). The only catch is how do we declare shared variables in the applications that use shared memory? Well, that's fairly easy if we play a sneaky trick on MASM, the Linker, DOS, and the 80x86.

When DOS loads your program into memory, it generally loads the segments in the same order they first appear in your source files. The UCR Standard Library, for example, takes advantage of this by insisting that you include a segment named zzzzzzseg at the end of all your assembly language source files. The UCR Standard Library memory management routines build the heap starting at zzzzzzseg, it must be the last segment (containing valid data) because the memory management routines may overwrite anything following zzzzzzseg.

For our shared memory segment, we would like to create a segment something like the following:

```

SharedMemory segment  para public 'Shared'
« define all shared variables here»
SharedMemory ends

```

Applications that share data would define all shared variables in this shared segment. There are, however, five problems. First, how do we tell the assembler/linker/DOS/80x86 that this is a *shared* segment, rather than having a separate segment for each program? Well, this problem is easy to solve; we don't bother telling MASM, the linker, or DOS anything. The way we make the different applications all share the same segment in memory is to invoke the shared memory TSR in the code above with function code 10h. This returns the address of the TSR's SharedMemory segment in the es register. In our assembly language programs we fool MASM into thinking es points at its local shared memory segment when, in fact, es points at the global segment.

The second problem is minor, but annoying nonetheless. When you create a segment, MASM, the linker, and DOS set aside storage for that segment. If you declare a large number of variables in a shared segment, this can waste memory since the program will actually use the memory space in the global shared segment. One easy way to reclaim the storage that MASM reserves for this segment is to define the shared segment *after* zzzzzzseg in your shared memory applications. By doing so, the Standard Library will absorb any memory reserved for the (dummy) shared memory segment into the heap, since all memory after zzzzzzseg belongs to the heap (when you use the standard meminit call).

The third problem is slightly more difficult to deal with. Since you will not be use the local segment, you cannot initialize any variables in the shared memory segment by placing values in the operand field of byte, word, dword, etc., directives. Doing so will only initialize the local memory in the heap, the system will not copy this data to the global shared segment. Generally, this isn't a problem because processes won't normally initialize shared memory as they load. Instead, there will probably be a single application you run first that initializes the shared memory area for the rest of the processes that using the global shared segment.

The fourth problem is that you cannot initialize any variables with the address of an object in shared memory. For example, if the variable shared_K is in the shared memory segment, you could not use a statement like the following:

```
printf
byte    "Value of shared_K is %d\n",0
dword  shared_K
```

The problem with this code is that MASM initializes the double word after the string above with the address of the shared_K variable *in the local copy of the shared data segment*. This will not print out the copy in the global shared data segment.

The last problem is anything but minor. All programs that use the global shared memory segment *must* define their variables at identical offsets within the shared segment. Given the way MASM assigns offsets to variables within a segment, if you are one byte off in the declaration of *any* of your variables, your program will be accessing its variables at different addresses than other processes sharing the global shared segment. This will scramble memory and produce a disaster. The only reasonable way to declare variables for shared memory programs is to create an include file with all the shared variable declarations for all concerned programs. Then include this single file into all the programs that share the variables. Now you can add, remove, or modify variables without having to worry about maintaining the shared variable declarations in the other files.

The following two sample programs demonstrate the use of shared memory. The first application reads a string from the user and stuffs it into shared memory. The second application reads that string from shared memory and displays it on the screen.

First, here is the include file containing the single shared variable declaration used by both applications:

```
; shmvars.asm
;
; This file contains the shared memory variable declarations used by
; all applications that refer to shared memory.

InputLine    byte    128 dup (?)
```



```

ExitPgm

; If the shared memory TSR is present, get the address of the shared segment
; into the ES register:

ItsThere:    mov     ah, cl           ;ID of our TSR.
             mov     al, 10h        ;Get shared segment address.
             int     2Fh

; Get the input line from the user:

             print
             byte    "Enter a string: ",0

             lea    di, InputLine    ;ES already points at proper seg.
             gets

             print
             byte    "Entered '",0
             puts
             print
             byte    "' into shared memory.",cr,lf,0

Quit:       ExitPgm                ;DOS macro to quit program.
Main        endp

cseg ends

sseg        segment    para stack 'stack'
stk         db         1024 dup ("stack ")
sseg        ends

zzzzzzseg   segment    para public 'zzzzzz'
LastBytes   db         16 dup (?)
zzzzzzseg   ends

; The shared memory segment must appear after "zzzzzzseg".
; Note that this isn't the physical storage for the data in the
; shared segment. It's really just a place holder so we can declare
; variables and generate their offsets appropriately. The UCR Standard
; Library will reuse the memory associated with this segment for the
; heap. To access data in the shared segment, this application calls
; the shared memory TSR to obtain the true segment address of the
; shared memory segment. It can then access variables in the shared
; memory segment (where ever it happens to be) off the ES register.
;
; Note that all the variable declarations go into an include file.
; All applications that refer to the shared memory segment include
; this file in the SharedMemory segment. This ensures that all
; shared segments have the exact same variable layout.

SharedMemory segment    para public 'Shared'

             include    shmvars.asm

SharedMemory ends
end          Main

```

The second application is very similar, here it is

```

; SHMAPP2.ASM
;
; This is a shared memory application that uses the static shared memory
; TSR (SHARDMEM.ASM). This program assumes the user has already run the
; SHMAPP1 program to insert a string into shared memory. This program
; simply prints that string from shared memory.
;

```

```

        .xlist
        include    stdlib.a
        includelib stdlib.lib
        .list

dseg      segment    para public 'data'
ShmID     byte      0
dseg      ends

cseg      segment    para public 'code'
          assume     cs:cseg, ds:dseg, es:SharedMemory

; SeeIfPresent Checks to see if the shared memory TSR is present in memory.
;               Sets the zero flag if it is, clears the zero flag if
;               it is not. This routine also returns the TSR ID in CL.

SeeIfPresent  proc      near
              push     es
              push     ds
              push     di
              mov      cx, 0ffh          ;Start with ID 0FFh.
IDLoop:      mov      ah, cl
              push     cx
              mov      al, 0            ;Verify presence call.
              int     2Fh
              pop      cx
              cmp      al, 0            ;Present in memory?
              je       TryNext
              strcml  byte      "Static Shared Memory TSR",0
              je       Success

TryNext:     dec      cl                ;Test USER IDs of 80h..FFh
              js      IDLoop
              cmp      cx, 0            ;Clear zero flag.
Success:     pop      di
              pop      ds
              pop      es
              ret
SeeIfPresent endp

; The main program for application #1 links with the shared memory
; TSR and then reads a string from the user (storing the string into
; shared memory) and then terminates.

Main        proc
          assume     cs:cseg, ds:dseg, es:SharedMemory
          mov      ax, dseg
          mov      ds, ax
          meminit

          print
          byte      "Shared memory application #2",cr,lf,0

; See if the shared memory TSR is around:

          call     SeeIfPresent
          je       ItsThere
          print
          byte      "Shared Memory TSR (SHARDMEM) is not loaded.",cr,lf
          byte      "This program cannot continue execution.",cr,lf,0
          ExitPgm

; If the shared memory TSR is present, get the address of the shared segment
; into the ES register:

ItsThere:   mov      ah, cl            ;ID of our TSR.
          mov      al, 10h           ;Get shared segment address.
          int     2Fh

; Print the string input in SHMAPP1:

```

```

        print
        byte    "String from SHMAPP1 is '",0

        lea    di, InputLine    ;ES already points at proper seg.
        puts

        print
        byte    "' from shared memory.",cr,lf,0

Quit:    ExitPgm                ;DOS macro to quit program.
Main    endp

cseg    ends

sseg    segment    para stack 'stack'
stk     db          1024 dup ("stack ")
sseg    ends

zzzzzzseg    segment    para public 'zzzzzz'
LastBytes   db          16 dup (?)
zzzzzzseg    ends

; The shared memory segment must appear after "zzzzzzseg".
; Note that this isn't the physical storage for the data in the
; shared segment. It's really just a place holder so we can declare
; variables and generate their offsets appropriately. The UCR Standard
; Library will reuse the memory associated with this segment for the
; heap. To access data in the shared segment, this application calls
; the shared memory TSR to obtain the true segment address of the
; shared memory segment. It can then access variables in the shared
; memory segment (where ever it happens to be) off the ES register.
;
; Note that all the variable declarations go into an include file.
; All applications that refer to the shared memory segment include
; this file in the SharedMemory segment. This ensures that all
; shared segments have the exact same variable layout.

SharedMemory    segment    para public 'Shared'

                include    shmvars.asm

SharedMemory    ends
end              Main

```

19.2.2 Dynamic Shared Memory

Although the static shared memory the previous section describes is very useful, it does suffer from a few limitations. First of all, any program that uses the global shared segment must be aware of the location of every other program that uses the shared segment. This effectively means that the use of the shared segment is limited to a single set of cooperating processes at any one given time. You cannot have two independent sets of programs using the shared memory at the same time. Another limitation with the static system is that you must know the size of all variables when you write your program, you cannot create dynamic data structures whose size varies at run time. It would be nice, for example, to have calls like `shmalloc` and `shmfree` that let you dynamically allocate and free memory in a shared region. Fortunately, it is very easy to overcome these limitations by creating a *dynamic shared memory manager*.

A reasonable shared memory manager will have four functions: `initialize`, `shmalloc`, `shmattach`, and `shmfree`. The initialization call reclaims all shared memory in use. The `shmalloc` call lets a process allocate a new block of shared memory. Only one process in a group of cooperating processes makes this call. Once `shmalloc` allocates a block of memory, the other processes use the `shmattach` call to obtain the address of the shared memory block. The following code implements a dynamic shared memory manager. The code is similar to that appearing in the Standard Library except this code allows a maximum of 64K storage on the heap.

```

; SHMALLOC.ASM
;
; This TSR sets up a dynamic shared memory system.
;
; This TSR checks to make sure there isn't a copy already active in
; memory. When removing itself from memory, it makes sure there are
; no other interrupts chained into INT 2Fh before doing the remove.
;
;
;
; The following segments must appear in this order and before the
; Standard Library includes.

ResidentSeg    segment    para public 'Resident'
ResidentSeg    ends

SharedMemory   segment    para public 'Shared'
SharedMemory   ends

EndResident    segment    para public 'EndRes'
EndResident    ends

                .xlist
                .286
                include    stdlib.a
                includelib stdlib.lib
                .list

; Resident segment that holds the TSR code:

ResidentSeg    segment    para public 'Resident'
                assume    cs:ResidentSeg, ds:nothing

NULL          equ        0

; Data structure for an allocated data region.
;
; Key-   user supplied ID to associate this region with a particular set
;        of processes.
;
; Next-  Points at the next allocated block.
; Prev-  Points at the previous allocated block.
; Size-  Size (in bytes) of allocated block, not including header structure.

Region        struct
key           word        ?
next         word        ?
prev         word        ?
blksize      word        ?
Region       ends

Startmem      equ        Region ptr [0]

AllocatedList word        0           ;Points at chain of alloc'd blocks.
FreeList      word        0           ;Points at chain of free blocks.

; Int 2Fh ID number for this TSR:

MyTSRID       byte        0
              byte        0           ;Padding so we can print it.

; PSP is the psp address for this program.

PSP           word        0

OldInt2F      dword       ?

; MyInt2F-   Provides int 2Fh (multiplex interrupt) support for this
;            TSR. The multiplex interrupt recognizes the following
;            subfunctions (passed in AL):

```

```

;
;           00h- Verify presence.       Returns 0FFh in AL and a pointer
;                                       to an ID string in es:di if the
;                                       TSR ID (in AH) matches this
;                                       particular TSR.
;
;           01h- Remove.               Removes the TSR from memory.
;                                       Returns 0 in AL if successful,
;                                       1 in AL if failure.
;
;           11h- shmalloc               CX contains the size of the block
;                                       to allocate.
;                                       DX contains the key for this block.
;                                       Returns a pointer to block in ES:DI
;                                       and size of allocated block in CX.
;                                       Returns an error code in AX. Zero
;                                       is no error, one is "key already
;                                       exists," two is "insufficient
;                                       memory for request."
;
;           12h- shmfree               DX contains the key for this block.
;                                       This call frees the specified block
;                                       from memory.
;
;           13h- shminit               Initializes the shared memory system
;                                       freeing all blocks currently in
;                                       use.
;
;           14h- shmattach             DX contains the key for a block.
;                                       Search for that block and return
;                                       its address in ES:DI. AX contains
;                                       zero if successful, three if it
;                                       cannot locate a block with the
;                                       specified key.

MyInt2F      proc      far
             assume   ds:nothing

             cmp      ah, MyTSRID;Match our TSR identifier?
             je       YepItsOurs
             jmp      OldInt2F

; Okay, we know this is our ID, now check for a verify, remove, or
; return segment call.

YepItsOurs:  cmp      al, 0                ;Verify Call
             jne     TryRmv
             mov     al, 0FFh;Return success.
             lesi   IDString
             iredt   ;Return back to caller.

IDString byte "Dynamic Shared Memory TSR",0

TryRmv:      cmp      al, 1                ;Remove call.
             jne     Tryshmalloc

; See if we can remove this TSR:

             push    es
             mov     ax, 0
             mov     es, ax
             cmp     word ptr es:[2Fh*4], offset MyInt2F
             jne     TRDone
             cmp     word ptr es:[2Fh*4 + 2], seg MyInt2F
             je      CanRemove             ;Branch if we can.
TRDone:      mov     ax, 1                ;Return failure for now.
             pop     es
             iredt

; Okay, they want to remove this guy *and* we can remove it from memory.
; Take care of all that here.

             assume  ds:ResidentSeg

```

```

CanRemove:  push    ds
            pusha
            cli                    ;Turn off the interrupts while
            mov     ax, 0           ; we mess with the interrupt
            mov     es, ax         ; vectors.
            mov     ax, cs
            mov     ds, ax

            mov     ax, word ptr OldInt2F
            mov     es:[2Fh*4], ax
            mov     ax, word ptr OldInt2F+2
            mov     es:[2Fh*4 + 2], ax

; Okay, one last thing before we quit- Let's give the memory allocated
; to this TSR back to DOS.

            mov     ds, PSP
            mov     es, ds:[2Ch]   ;Ptr to environment block.
            mov     ah, 49h       ;DOS release memory call.
            int     21h

            mov     ax, ds        ;Release program code space.
            mov     es, ax
            mov     ah, 49h
            int     21h

            popa
            pop     ds
            pop     es
            mov     ax, 0         ;Return Success.
            iret

; Stick BadKey here so that it is close to its associated branch (from below).
;
; If come here, we've discovered an allocated block with the
; specified key. Return an error code (AX=1) and the size of that
; allocated block (in CX).

BadKey:    mov     cx, [bx].Region.BlkSize
            mov     ax, 1         ;Already allocated error.
            pop     bx
            pop     ds
            iret

; See if this is a shmalloc call.
; If so, on entry -
; DX contains the key.
; CX contains the number of bytes to allocate.
;
; On exit:
;
; ES:DI points at the allocated block (if successful).
; CX contains the actual size of the allocated block (>=CX on entry).
; AX contains error code, 0 if no error.

Tryshmalloc:  cmp     al, 11h        ;shmalloc function code.
             jne Tryshmfree

; First, search through the allocated list to see if a block with the
; current key number already exists. DX contains the requested key.

            assume ds:SharedMemory
            assume bx:ptr Region
            assume di:ptr Region

            push   ds
            push   bx
            mov    bx, SharedMemory
            mov    ds, bx

```

```

        mov     bx, ResidentSeg:AllocatedList
        test   bx, bx           ;Anything on this list?
        je     SrchFreeList

SearchLoop:  cmp     dx, [bx].Key       ;Key exist already?
        je     BadKey
        mov     bx, [bx].Next   ;Get next region.
        test   bx, bx         ;NULL?, if not, try another
        jne    SearchLoop     ; entry in the list.

; If an allocated block with the specified key does not already exist,
; then try to allocate one from the free memory list.

SrchFreeList:  mov     bx, ResidentSeg:FreeList
        test   bx, bx         ;Empty free list?
        je     OutaMemory

FirstFitLp:   cmp     cx, [bx].BlkSize ;Is this block big enough?
        jbe    GotBlock
        mov     bx, [bx].Next   ;If not, on to the next one.
        test   bx, bx         ;Anything on this list?
        jne    FirstFitLp

; If we drop down here, we were unable to find a block that was large
; enough to satisfy the request. Return an appropriate error

OutaMemory:   mov     cx, 0           ;Nothing available.
        mov     ax, 2           ;Insufficient memory error.
        pop     bx
        pop     ds
        iret

; If we find a large enough block, we've got to carve the new block
; out of it and return the rest of the storage to the free list. If the
; free block is at least 32 bytes larger than the requested size, we will
; do this. If the free block is less than 32 bytes larger, we will simply
; give this free block to the requesting process. The reason for the
; 32 bytes is simple: We need eight bytes for the new block's header
; (the free block already has one) and it doesn't make sense to fragment
; blocks to sizes below 24 bytes. That would only increase processing time
; when processes free up blocks by requiring more work coalescing blocks.

GotBlock:     mov     ax, [bx].BlkSize ;Compute difference in size.
        sub     ax, cx
        cmp     ax, 32           ;At least 32 bytes left?
        jbe    GrabWholeBlk     ;If not, take this block.

; Okay, the free block is larger than the requested size by more than 32
; bytes. Carve the new block from the end of the free block (that way
; we do not have to change the free block's pointers, only the size.

        mov     di, bx
        add     di, [bx].BlkSize ;Scoot to end, minus 8
        sub     di, cx           ;Point at new block.

        sub     [bx].BlkSize, cx ;Remove alloc'd block and
        sub     [bx].BlkSize, 8  ; room for header.

        mov     [di].BlkSize, cx ;Save size of block.
        mov     [di].Key, dx     ;Save key.

; Link the new block into the list of allocated blocks.

        mov     bx, ResidentSeg:AllocatedList
        mov     [di].Next, bx
        mov     [di].Prev, NULL ;NULL previous pointer.
        test   bx, bx         ;See if it was an empty list.
        je     NoPrev
        mov     [bx].Prev, di  ;Set prev ptr for old guy.

NoPrev:      mov     ResidentSeg:AllocatedList, di
RmvDone:     add     di, 8           ;Point at actual data area.
        mov     ax, ds           ;Return ptr in es:di.
        mov     es, ax

```

```

        mov     ax, 0           ;Return success.
        pop     bx
        pop     ds
        iret

; If the current free block is larger than the request, but not by more
; that 32 bytes, just give the whole block to the user.

GrabWholeBlk: mov     di, bx
               mov     cx, [bx].BlkSize ;Return actual size.
               cmp     [bx].Prev, NULL ;First guy in list?
               je      Rmv1st
               cmp     [bx].Next, NULL ;Last guy in list?
               je      RmvLast

; Okay, this record is sandwiched between two other in the free list.
; Cut it out from among the two.

               mov     ax, [bx].Next   ;Save the ptr to the next
               mov     bx, [bx].Prev   ; item in the prev item's
               mov     [bx].Next, ax   ; next field.

               mov     ax, bx          ;Save the ptr to the prev
               mov     bx, [di].Next   ; item in the next item's
               mov     [bx].Prev, bx   ; prev field.
               jmp     RmvDone

; The block we want to remove is at the beginning of the free list.
; It could also be the only item on the free list!

Rmv1st:       mov     ax, [bx].Next
               mov     FreeList, ax    ;Remove from free list.
               jmp     RmvDone

; If the block we want to remove is at the end of the list, handle that
; down here.

RmvLast:     mov     bx, [bx].Prev
               mov     [bx].Next, NULL
               jmp     RmvDone

               assume   ds:nothing, bx:nothing, di:nothing

; This code handles the SHMFREE function.
; On entry, DX contains the key for the block to free. We need to
; search through the allocated block list and find the block with that
; key. If we do not find such a block, this code returns without doing
; anything. If we find the block, we need to add its memory to the
; free pool. However, we cannot simply insert this block on the front
; of the free list (as we did for the allocated blocks). It might
; turn out that this block we're freeing is adjacent to one or two
; other free blocks. This code has to coalesce such blocks into
; a single free block.

Tryshmfree:  cmp     al, 12h
               jne     Tryshminit

; First, search the allocated block list to see if we can find the
; block to remove. If we don't find it in the list anywhere, just return.

               assume   ds:SharedMemory
               assume   bx:ptr Region
               assume   di:ptr Region

               push    ds
               push    di
               push    bx

```

```

        mov     bx, SharedMemory
        mov     ds, bx
        mov     bx, ResidentSeg:AllocatedList

        test    bx, bx           ;Empty allocated list?
        je     FreeDone
SrchList:  cmp     dx, [bx].Key      ;Search for key in DX.
        je     FoundIt
        mov     bx, [bx].Next
        test    bx, bx           ;At end of list?
        jne    SrchList
FreeDone:  pop     bx
        pop     di               ;Nothing allocated, just
        pop     ds               ; return to caller.
        iret

; Okay, we found the block the user wants to delete. Remove it from
; the allocated list. There are three cases to consider:
; (1) it is at the front of the allocated list, (2) it is at the end of
; the allocated list, and (3) it is in the middle of the allocated list.

FoundIt:  cmp     [bx].Prev, NULL ;1st item in list?
        je     Free1st
        cmp    [bx].Next, NULL ;Last item in list?
        je     FreeLast

; Okay, we're removing an allocated item from the middle of the allocated
; list.

        mov     di, [bx].Next    ;[next].prev := [cur].prev
        mov     ax, [bx].Prev
        mov     [di].Prev, ax
        xchg    ax, di
        mov     [di].Next, ax    ;[prev].next := [cur].next
        jmp     AddFree

; Handle the case where we are removing the first item from the allocation
; list. It is possible that this is the only item on the list (i.e., it
; is the first and last item), but this code handles that case without any
; problems.

Free1st:  mov     ax, [bx].Next
        mov     ResidentSeg:AllocatedList, ax
        jmp     AddFree

; If we're removing the last guy in the chain, simply set the next field
; of the previous node in the list to NULL.

FreeLast: mov     di, [bx].Prev
        mov     [di].Next, NULL

; Okay, now we've got to put the freed block onto the free block list.
; The free block list is sorted according to address. We have to search
; for the first free block whose address is greater than the block we've
; just freed and insert the new free block before that one. If the two
; blocks are adjacent, then we've got to merge them into a single free
; block. Also, if the block before is adjacent, we must merge it as
; well. This will coalesce all free blocks on the free list so there
; are as few free blocks as possible and those blocks are as large as
; possible.

AddFree:  mov     ax, ResidentSeg:FreeList
        test    ax, ax           ;Empty list?
        jne    SrchPosn

; If the list is empty, stick this guy on as the only entry.

        mov     ResidentSeg:FreeList, bx
        mov     [bx].Next, NULL
        mov     [bx].Prev, NULL
        jmp     FreeDone

```

```

; If the free list is not empty, search for the position of this block
; in the free list:

SrchPosn:    mov     di, ax
             cmp     bx, di
             jb     FoundPosn
             mov     ax, [di].Next
             test    ax, ax           ;At end of list?
             jne    SrchPosn

; If we fall down here, the free block belongs at the end of the list.
; See if we need to merge the new block with the old one.

             mov     ax, di
             add     ax, [di].BlkSize ;Compute address of 1st byte
             add     ax, 8           ; after this block.
             cmp     ax, bx
             je     MergeLast

; Okay, just add the free block to the end of the list.

             mov     [di].Next, bx
             mov     [bx].Prev, di
             mov     [bx].Next, NULL
             jmp     FreeDone

; Merge the freed block with the block DI points at.

MergeLast:  mov     ax, [di].BlkSize
             add     ax, [bx].BlkSize
             add     ax, 8
             mov     [di].BlkSize, ax
             jmp     FreeDone

; If we found a free block before which we are supposed to insert
; the current free block, drop down here and handle it.

FoundPosn:  mov     ax, bx           ;Compute the address of the
             add     ax, [bx].BlkSize ; next block in memory.
             add     ax, 8
             cmp     ax, di           ;Equal to this block?
             jne    DontMerge

; The next free block is adjacent to the one we're freeing, so just
; merge the two.

             mov     ax, [di].BlkSize ;Merge the sizes together.
             add     ax, 8
             add     [bx].BlkSize, ax
             mov     ax, [di].Next   ;Tweak the links.
             mov     [bx].Next, ax
             mov     ax, [di].Prev
             mov     [bx].Prev, ax
             jmp     TryMergeB4

; If the blocks are not adjacent, just link them together here.

DontMerge:  mov     ax, [di].Prev
             mov     [di].Prev, bx
             mov     [bx].Prev, ax
             mov     [bx].Next, di

; Now, see if we can merge the current free block with the previous free blk.

TryMergeB4: mov     di, [bx].Prev
             mov     ax, di
             add     ax, [di].BlkSize
             add     ax, 8
             cmp     ax, bx
             je     CanMerge
             pop     bx
             pop     di           ;Nothing allocated, just
             pop     ds           ; return to caller.
             iret

```

```

; If we can merge the previous and current free blocks, do that here:
CanMerge:    mov     ax, [bx].Next
             mov     [di].Next, ax
             mov     ax, [bx].BlkSize
             add     ax, 8
             add     [di].BlkSize, ax
             pop     bx
             pop     di
             pop     ds
             ired

             assume  ds:nothing
             assume  bx:nothing
             assume  di:nothing

; Here's where we handle the shared memory initializatin (SHMINIT) function.
; All we got to do is create a single block on the free list (which is all
; available memory), empty out the allocated list, and then zero out all
; shared memory.

Tryshminit:  cmp     al, 13h
             jne     TryShmAttach

; Reset the memory allocation area to contain a single, free, block of
; memory whose size is 0FFF8h (need to reserve eight bytes for the block's
; data structure).

             push    es
             push    di
             push    cx

             mov     ax, SharedMemory ;Zero out the shared
             mov     es, ax           ; memory segment.
             mov     cx, 32768
             xor     ax, ax
             mov     di, ax
             rep     stosw

; Note: the commented out lines below are unnecessary since the code above
; has already zeroed out the entire shared memory segment.
; Note: we cannot put the first record at offset zero because offset zero
; is the special value for the NULL pointer. We'll use 4 instead.

             mov     di, 4
;             mov     es:[di].Region.Key, 0 ;Key is arbitrary.
;             mov     es:[di].Region.Next, 0 ;No other entries.
;             mov     es:[di].Region.Prev, 0 ; Ditto.
             mov     es:[di].Region.BlkSize, 0FFF8h ;Rest of segment.
             mov     ResidentSeg:FreeList, di

             pop     cx
             pop     di
             pop     es
             mov     ax, 0             ;Return no error.
             ired

; Handle the SHMATTACH function here. On entry, DX contains a key number.
; Search for an allocated block with that key number and return a pointer
; to that block (if found) in ES:DI. Return an error code (AX=3) if we
; cannot find the block.

TryShmAttach: cmp     al, 14h           ;Attach opcode.
              jne     IllegalOp
              mov     ax, SharedMemory
              mov     es, ax

FindOurs:    mov     di, ResidentSeg:AllocatedList
              cmp     dx, es:[di].Region.Key
              je      FoundOurs
              mov     di, es:[di].Region.Next

```

```

        test     di, di
        jne     FoundOurs
        mov     ax, 3           ;Can't find the key.
        ired

FoundOurs:  add     di, 8           ;Point at actual data.
        mov     ax, 0           ;No error.
        ired

; They called us with an illegal subfunction value. Try to do as little
; damage as possible.

IllegalOp:  mov     ax, 0           ;Who knows what they were thinking?
        ired
MyInt2F    endp
ResidentSeg  assume     ds:nothing
        ends

; Here's the segment that will actually hold the shared data.

SharedMemory  segment     para public 'Shared'
        db         0FFFFh dup (?)
SharedMemory  ends

cseg        segment     para public 'code'
        assume     cs:cseg, ds:ResidentSeg

; SeeIfPresent-      Checks to see if our TSR is already present in memory.
;                   Sets the zero flag if it is, clears the zero flag if
;                   it is not.

SeeIfPresent  proc      near
        push     es
        push     ds
        push     di
        mov     cx, 0ffh      ;Start with ID 0FFh.
IDLoop:      mov     ah, cl
        push     cx
        mov     al, 0         ;Verify presence call.
        int     2Fh
        pop     cx
        cmp     al, 0         ;Present in memory?
        je     TryNext
        strcmpl
        byte    "Dynamic Shared Memory TSR",0
        je     Success

TryNext:     dec     cl           ;Test USER IDs of 80h..FFh
        js     IDLoop
        cmp     cx, 0         ;Clear zero flag.
Success:     pop     di
        pop     ds
        pop     es
        ret
SeeIfPresent  endp

; FindID-           Determines the first (well, last actually) TSR ID available
;                   in the multiplex interrupt chain. Returns this value in
;                   the CL register.
;
;                   Returns the zero flag set if it locates an empty slot.
;                   Returns the zero flag clear if failure.

FindID       proc      near
        push     es

```

```

        push    ds
        push    di

IDLoop:  mov     cx, 0ffh      ;Start with ID 0FFh.
        mov     ah, cl
        push   cx
        mov     al, 0      ;Verify presence call.
        int    2Fh
        pop    cx
        cmp    al, 0      ;Present in memory?
        je     Success
        dec    cl          ;Test USER IDs of 80h..FFh
        js    IDLoop
        xor    cx, cx
        cmp    cx, 1      ;Clear zero flag
Success: pop    di
        pop    ds
        pop    es
        ret

FindID   endp

Main     proc
        meminit

        mov    ax, ResidentSeg
        mov    ds, ax

        mov    ah, 62h    ;Get this program's PSP
        int    21h        ; value.
        mov    PSP, bx

; Before we do anything else, we need to check the command line
; parameters. If there is one, and it is the word "REMOVE", then remove
; the resident copy from memory using the multiplex (2Fh) interrupt.

        argc
        cmp    cx, 1      ;Must have 0 or 1 parms.
        jb    TstPresent
        je    DoRemove

Usage:   print
        byte   "Usage:", cr, lf
        byte   "  shmalloc", cr, lf
        byte   "  or shmalloc REMOVE", cr, lf, 0
        ExitPgm

; Check for the REMOVE command.

DoRemove: mov    ax, 1
        argv
        stricmp    "REMOVE", 0
        jne    Usage

        call    SeeIfPresent
        je     RemoveIt
        print
        byte   "TSR is not present in memory, cannot remove"
        byte   cr, lf, 0
        ExitPgm

RemoveIt: mov    MyTSRID, cl
        printf
        byte   "Removing TSR (ID #%d) from memory...", 0
        dword MyTSRID

        mov    ah, cl
        mov    al, 1      ;Remove cmd, ah contains ID
        int    2Fh
        cmp    al, 1      ;Succeed?
        je    RmvFailure
        print

```

```

        byte      "removed.",cr,lf,0
        ExitPgm

RmvFailure:  print
             byte      cr,lf
             byte      "Could not remove TSR from memory.",cr,lf
             byte      "Try removing other TSRs in the reverse order "
             byte      "you installed them.",cr,lf,0
             ExitPgm

; Okay, see if the TSR is already in memory. If so, abort the
; installation process.

TstPresent:  call      SeeIfPresent
             jne      GetTSRID
             print
             byte      "TSR is already present in memory.",cr,lf
             byte      "Aborting installation process",cr,lf,0
             ExitPgm

; Get an ID for our TSR and save it away.

GetTSRID:    call      FindID
             je       GetFileName
             print
             byte      "Too many resident TSRs, cannot install",cr,lf,0
             ExitPgm

; Things look cool so far, so install the interrupts

GetFileName: mov      MyTSRID, c1
             print
             byte      "Installing interrupts...",0

; Patch into the INT 2Fh interrupt chain.

             cli                    ;Turn off interrupts!
             mov      ax, 0
             mov      es, ax
             mov      ax, es:[2Fh*4]
             mov      word ptr OldInt2F, ax
             mov      ax, es:[2Fh*4 + 2]
             mov      word ptr OldInt2F+2, ax
             mov      es:[2Fh*4], offset MyInt2F
             mov      es:[2Fh*4+2], seg ResidentSeg
             sti                    ;Okay, ints back on.

; We're hooked up, the only thing that remains is to initialize the shared
; memory segment and then terminate and stay resident.

             printf
             byte      "Installed, TSR ID #%.d.",cr,lf,0
             dword    MyTSRID

             mov      ah, MyTSRID    ;Initialization call.
             mov      al, 13h
             int      2Fh

             mov      dx, EndResident ;Compute size of program.
             sub      dx, PSP
             mov      ax, 3100h      ;DOS TSR command.
             int      21h

Main
cseg      endp
         ends

sseg      segment para stack 'stack'
stk       db      256 dup (?)
sseg      ends

```



```

        print
        byte      "Shared memory application #3",cr,lf,0

; See if the shared memory TSR is around:

        call     SeeIfPresent
        je      ItsThere
        print
        byte     "Shared Memory TSR (SHMALLOC) is not loaded.",cr,lf
        byte     "This program cannot continue execution.",cr,lf,0
        ExitPgm

; Get the input line from the user:

ItsThere:  mov     ShmID, c1
           print
           byte   "Enter a string: ",0

           lea   di, InputLine      ;ES already points at proper seg.
           getsm

; The string is in our heap space. Let's move it over to the shared
; memory segment.

           strlen
           inc   cx                ;Add one for zero byte.
           push  es
           push  di

           mov   dx, 1234h         ;Our "key" value.
           mov   ah, ShmID
           mov   al, 11h          ;Shmalloc call.
           int   2Fh

           mov   si, di            ;Save as dest ptr.
           mov   dx, es

           pop   di                ;Retrieve source address.
           pop   es
           strcpy

           print
           byte  "Entered '",0
           puts
           print
           byte  "' into shared memory.",cr,lf,0

Quit:     ExitPgm                ;DOS macro to quit program.
Main     endp

cseg     ends

sseg     segment   para stack 'stack'
stk      db        1024 dup ("stack ")
sseg     ends

zzzzzzseg segment   para public 'zzzzzz'
LastBytes db        16 dup (?)
zzzzzzseg ends

end      Main

```

```

; SHMAPP4.ASM
;
; This is a shared memory application that uses the dynamic shared memory
; TSR (SHMALLOC.ASM). This program assumes the user has already run the
; SHMAPP3 program to insert a string into shared memory. This program

```

```

; simply prints that string from shared memory.
;
        .xlist
        include  stdlib.a
        includelib stdlib.lib
        .list

dseg      segment    para public 'data'
ShmID     byte       0
dseg      ends

cseg      segment    para public 'code'
          assume     cs:cseg, ds:dseg, es:SharedMemory

; SeeIfPresent-Checks to see if the shared memory TSR is present in memory.
;           Sets the zero flag if it is, clears the zero flag if
;           it is not. This routine also returns the TSR ID in CL.

SeeIfPresent  proc      near
              push     es
              push     ds
              push     di
IDLoop:       mov     cx, 0ffh          ;Start with ID 0FFh.
              mov     ah, cl
              push     cx
              mov     al, 0           ;Verify presence call.
              int     2Fh
              pop      cx
              cmp     al, 0           ;Present in memory?
              je      TryNext
              strcml  "Dynamic Shared Memory TSR",0
              je      Success

TryNext:     dec     cl              ;Test USER IDs of 80h..FFh
              js     IDLoop
              cmp     cx, 0           ;Clear zero flag.
Success:     pop      di
              pop      ds
              pop      es
              ret

SeeIfPresent  endp

; The main program for application #1 links with the shared memory
; TSR and then reads a string from the user (storing the string into
; shared memory) and then terminates.

Main         proc
              assume   cs:cseg, ds:dseg, es:SharedMemory
              mov      ax, dseg
              mov      ds, ax
              meminit

              print
              byte     "Shared memory application #4",cr,lf,0

; See if the shared memory TSR is around:

              call     SeeIfPresent
              je      ItsThere
              print
              byte     "Shared Memory TSR (SHMALLOC) is not loaded.",cr,lf
              byte     "This program cannot continue execution.",cr,lf,0
              ExitPgm

; If the shared memory TSR is present, get the address of the shared segment
; into the ES register:

ItsThere:    mov      ah, cl          ;ID of our TSR.
              mov      al, 14h       ;Attach call
              mov      dx, 1234h;Our "key" value
              int     2Fh

```

```

; Print the string input in SHMAPP3:

        print
        byte    "String from SHMAPP3 is '",0

        puts

        print
        byte    "' from shared memory.",cr,lf,0

Quit:   ExitPgm                                ;DOS macro to quit program.
Main    endp

cseg ends

sseg    segment    para stack 'stack'
stk     db         1024 dup ("stack ")
sseg    ends

zzzzzzseg    segment    para public 'zzzzzz'
LastBytes   db         16 dup (?)
zzzzzzseg    ends
end        Main

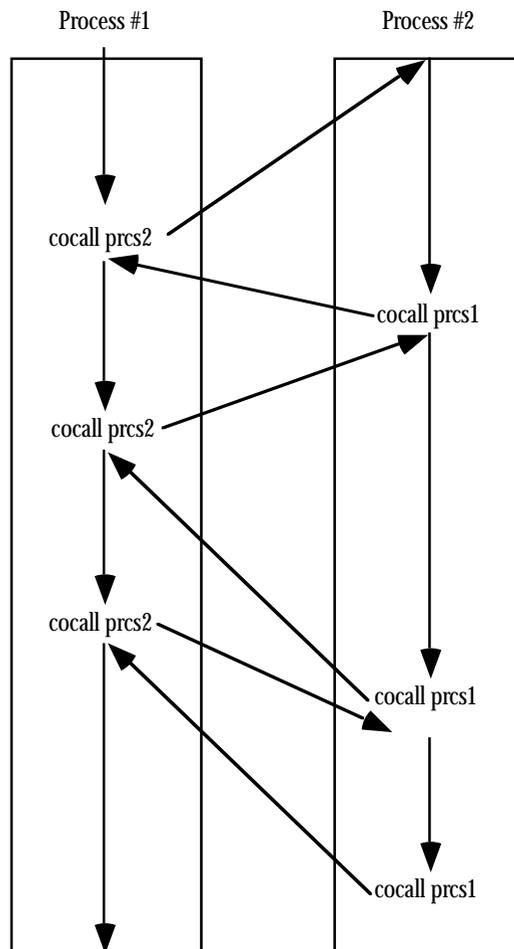
```

19.3 Coroutines

DOS processes, even when using shared memory, suffer from one primary drawback – each program executes to completion before returning control back to the parent process. While this paradigm is suitable for many applications, it certainly does not suffice for all. A common paradigm is for two programs to swap control of the CPU back and forth while executing. This mechanism, slightly different from the subroutine call and return mechanism, is a *coroutine*.

Before discussing coroutines, it is probably a good idea to provide a solid definition for the term *process*. In a nutshell, a process is a program that is executing. A program can exist on the disk; processes exist in memory and have a program stack (with return addresses, etc.) associated with them. If there are multiple processes in memory at one time, each process must have its own program stack.

A *cocall* operation transfers control between two processes. A cocall is effectively a call and a return instruction all rolled into one operation. From the point of view of the process executing the cocall, the cocall operation is equivalent to a procedure call; from the point of view of the processing being called, the cocall operation is equivalent to a return operation. When the second process cocalls the first, control resumes *not at the beginning of the first process*, but immediately after the cocall operation. If two processes execute a sequence of mutual cocalls, control will transfer between the two processes in the following fashion:



Cocall Sequence Between Two Processes

Cocalls are quite useful for games where the “players” take turns, following different strategies. The first player executes some code to make its first move, then cocalls the second player and allows it to make a move. After the second player makes its move, it cocalls the first process and gives the first player its second move, picking up immediately after its cocall. This transfer of control bounces back and forth until one player wins.

The 80x86 CPUs do not provide a cocall instruction. However, it is easy to implement cocalls with existing instructions. Even so, there is little need for you to supply your own cocall mechanism, the UCR Standard Library provides a cocall package for 8086, 80186, and 80286 processors². This package includes the `pcb` (process control block) data structure and three functions you can call: `coinit`, `cocall`, and `cocall1`.

The `pcb` structure maintains the current state of a process. The `pcb` maintains all the register values and other accounting information for a process. When a process makes a cocall, it stores the return address for the cocall in the `pcb`. Later, when some other process cocalls this process, the cocall operation simply reloads the registers, include `cs:ip`, from the `pcb` and that returns control to the next instruction after the first process' cocall. The `pcb` structure takes the following form:

```
pcb          struct
```

² The cocall package works fine with the other processors as long as you don't use the 32-bit register set. Later, we will discuss how to extend the Standard Library routines to handle the 32-bit capabilities of the 80386 and later processors.

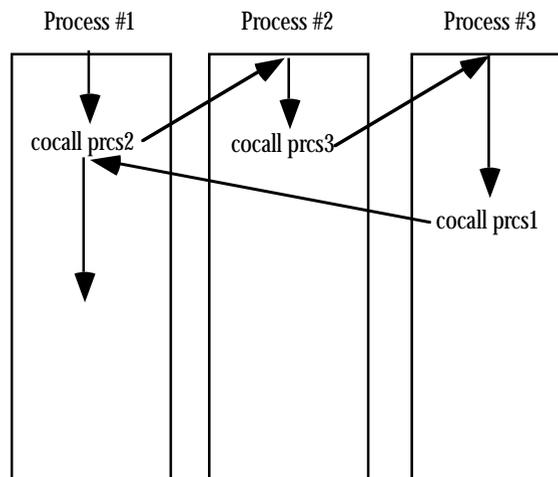
NextProc	dword	?	;Link to next PCB (for multitasking).
regsp	word	?	
regss	word	?	
regip	word	?	
regcs	word	?	
regax	word	?	
regbx	word	?	
regcx	word	?	
regdx	word	?	
regsi	word	?	
regdi	word	?	
regbp	word	?	
regds	word	?	
reges	word	?	
regflags	word	?	
PrcsID	word	?	
StartingTime	dword	?	;Used for multitasking accounting.
StartingDate	dword	?	;Used for multitasking accounting.
CPUTime	dword	?	;Used for multitasking accounting.

Four of these fields (as labelled) exist for preemptive multitasking and have no meaning for coroutines. We will discuss preemptive multitasking in the next section.

There are two important things that should be evident from this structure. First, the main reason the existing Standard Library coroutine support is limited to 16 bit register is because there is only room for the 16 bit versions of each of the registers in the pcb. If you want to support the 80386 and later 32 bit register sets, you would need to modify the pcb structure and the code that saves and restores registers in the pcb.

The second thing that should be evident is that the coroutine code preserves all registers across a cocal. This means you cannot pass information from one process to another in the registers when using a cocal. You will need to pass data between processes in global memory locations. Since coroutines generally exist in the same program, you will not even need to resort to the shared memory techniques. Any variables you declare in your data segment will be visible to all coroutines.

Note, by the way, that a program may contain more than two coroutines. If coroutine one cocal coroutine two, and coroutine two cocal coroutine three, and then coroutine three cocal coroutine one, coroutine one picks up immediately after the cocal it made to coroutine two.



Cocalls Between Three Processes

Since a cocal effectively *returns* to the target coroutine, you might wonder what happens on the *first* cocal to any process. After all, if that process has not executed any code, there is no “return address” where you can resume execution. This is an easy problem to solve, we need only initialize the return address of such a process to the address of the first instruction to execute in that process.

A similar problem exists for the stack. When a program begins execution, the main program (coroutine one) takes control and uses the stack associated with the entire program. Since each process must have its own stack, where do the other coroutines get their stacks?

The easiest way to initialize the stack and initial address for a coroutine is to do this when declaring a `pcb` for a process. Consider the following `pcb` variable declaration:

```
ProcessTwo    pcb        {0,                offset EndStack2, seg EndStack2,
                        offset StartLoc2, seg StartLoc2}
```

This definition initializes the `NextProc` field with `NULL` (the Standard Library coroutine functions do not use this field) and initialize the `ss:sp` and `cs:ip` fields with the last address of a stack area (`EndStack2`) and the first instruction of the process (`StartLoc2`). Now all you need to do is reserve a reasonable amount of stack storage for the process. You can create multiple stacks in the `SHELL.ASM` `sseg` as follows:

```
sseg          segment    para stack 'stack'

; Stack for process #2:

stk2          byte       1024 dup (?)
EndStack2     word       ?

; Stack for process #3:

stk3          byte       1024 dup (?)
EndStack3     word       ?

; The primary stack for the main program (process #1) must appear at
; the end of sseg.

stk           byte       1024 dup (?)
sseg          ends
```

There is the question of “how much space should one reserve for each stack?” This, of course, varies with the application. If you have a simple application that doesn’t use recursion or allocate any local variables on the stack, you could get by with as little as 256 bytes of stack space for a process. On the other hand, if you have recursive routines or allocate storage on the stack, you will need considerably more space. For simple programs, 1-8K stack storage should be sufficient. Keep in mind that you can allocate a maximum of 64K in the `SHELL.ASM` `sseg`. If you need additional stack space, you will need to up the other stacks in a different segment (they do not need to be in `sseg`, it’s just a convenient place for them) or you will need to allocate the stack space differently.

Note that you do not have to allocate the stack space as an array within your program. You can also allocate stack space dynamically using the Standard Library `malloc` call. The following code demonstrates how to set up an 8K dynamically allocated stack for the `pcb` variable `Process2`:

```
mov          cx, 8192
malloc
jc          InsufficientRoom
mov         Process2.ss, es
mov         Process2.sp, di
```

Setting up the coroutines the main program will call is pretty easy. However, there is the issue of setting up the `pcb` for the main program. You cannot initialize the `pcb` for the main program the same way you initialize the `pcb` for the other processes; it is already running and has valid `cs:ip` and `ss:sp` values. Were you to initialize the main program’s `pcb` the same way we did for the other processes, the system would simply restart the main program when you make a `cocall` back to it. To initialize the `pcb` for the main program, you must use the `coinit` function. The `coinit` function expects you to pass it the address of the main program’s `pcb` in the `es:di` register pair. It initializes some variables internal to the Standard Library so the first `cocall` operation will save the 80x86 machine state in the `pcb` you specify by `es:di`. After the `coinit` call, you can begin making `cocalls` to other processes in your program.

To cocall a coroutine, you use the Standard Library `cocall` function. The `cocall` function call takes two forms. Without any parameters this function transfers control to the coroutine whose `pcb` address appears in the `es:di` register pair. If the address of a `pcb` appears in the operand field of this instruction, `cocall` transfers control to the specified coroutine (don't forget, the name of the `pcb`, *not* the process, must appear in the operand field).

The best way to learn how to use coroutines is via example. The following program is an interesting piece of code that generates mazes on the PC's display. The maze generation algorithm has one major constraint - there must be no more than one correct solution to the maze (it is possible for there to be no solution). The main program creates a set of background processes called "demons" (actually, `daemon` is the correct term, but `demon` sounds more appropriate here). Each demon begins carving out a portion of the maze subject to the main constraint. Each demon gets to dig one cell from the maze and then it passes control to another demon. As it turns out, demons can "dig themselves into a corner" and die (demons live only to dig). When this happens, the demon removes itself from the list of active demons. When all demons die off, the maze is (in theory) complete. Since the demons die off fairly regularly, there must be some mechanism to create new demons. Therefore, this program randomly spawns new demons who start digging their own tunnels perpendicular to their parents. This helps ensure that there is a sufficient supply of demons to dig out the entire maze; the demons all die off only when there are no, or few, cells remaining to dig in the maze.

```

; AMAZE.ASM
;
; A maze generation/solution program.
;
; This program generates an 80x25 maze and directly draws the maze on the
; video display. It demonstrates the use of coroutines within a program.

                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list

byp            textequ    <byte ptr>

dseg          segment    para public 'data'

; Constants:
;
; Define the "ToScreen" symbol (to any value) if the maze is 80x25 and you
; want to display it on the video screen.

ToScreen      equ        0

; Maximum X and Y coordinates for the maze (matching the display).

MaxXCoord     equ        80
MaxYCoord     equ        25

; Useful X,Y constants:

WordsPerRow   =          MaxXCoord+2
BytesPerRow   =          WordsPerRow*2

StartX        equ        1                ;Starting X coordinate for maze
StartY        equ        3                ;Starting Y coordinate for maze
EndX          equ        MaxXCoord        ;Ending X coordinate for maze
EndY          equ        MaxYCoord-1      ;Ending Y coordinate for maze

EndLoc        =          ( (EndY-1)*MaxXCoord + EndX-1)*2
StartLoc      =          ( (StartY-1)*MaxXCoord + StartX-1)*2

; Special 16-bit PC character codes for the screen for symbols drawn during
; maze generation. See the chapter on the video display for details.

                ifdef     mono                ;Mono display adapter.

WallChar      equ        7dbh                ;Solid block character

```

```

NoWallChar    equ    720h                ;space
VisitChar     equ    72eh                ;Period
PathChar      equ    72ah                ;Asterisk

                                ;Color display adapter.

WallChar      equ    1dbh                ;Solid block character
NoWallChar    equ    0edbh               ;space
VisitChar     equ    0bdbh               ;Period
PathChar      equ    4e2ah               ;Asterisk

                                endif

; The following are the constants that may appear in the Maze array:

Wall          =        0
NoWall        =        1
Visited       =        2

; The following are the directions the demons can go in the maze

North         =        0
South         =        1
East          =        2
West          =        3

; Some important variables:

; The Maze array must contain an extra row and column around the
; outside edges for our algorithm to work properly.

Maze          word      (MaxYCoord+2) dup ((MaxXCoord+2) dup (Wall))

; The follow macro computes an index into the above array assuming
; a demon's X and Y coordinates are in the dl and dh registers, respectively.
; Returns index in the AX register

MazeAdrs      macro
                mov     al, dh
                mov     ah, WordsPerRow   ;Index into array is computed
                mul     ah                 ; by (Y*words/row + X)*2.
                add     al, dl
                adc     ah, 0
                shl     ax, 1             ;Convert to byte index
                endm

; The following macro computes an index into the screen array, using the
; same assumptions as above. Note that the screen matrix is 80x25 whereas
; the maze matrix is 82x27; The X/Y coordinates in DL/DH are 1..80 and
; 1..25 rather than 0..79 and 0..24 (like we need). This macro adjusts
; for that.

ScrnAdrs      macro
                mov     al, dh
                dec     al
                mov     ah, MaxXCoord
                mul     ah
                add     al, dl
                adc     ah, 0
                dec     ax
                shl     ax, 1
                endm

; PCB for the main program. The last live demon will call this guy when
; it dies.

MainPCB       pcb      {}

```

```

; List of up to 32 demons.

MaxDemons    =          32          ;Must be a power of two.
ModDemons    =          MaxDemons-1  ;Mask for MOD computation.

DemonList    pcb          MaxDemons dup ({}))

DemonIndex   byte        0          ;Index into demon list.
DemonCnt     byte        0          ;Number of demons in list.

; Random number generator seed (we'll use our random number generator
; rather than the standard library's because we want to be able to specify
; an initial seed value).

Seed         word        0

dseg         ends

; The following is the segment address of the video display, change this
; from 0B800h to 0B000h if you have a monochrome display rather than a
; color display.

ScreenSeg    segment     at 0b800h
Screen       equ         this word   ;Don't generate in date here!
ScreenSeg    ends

cseg         segment     para public 'code'
              assume     cs:cseg, ds:dseg

; Totally bogus random number generator, but we don't need a really
; great one for this program. This code uses its own random number
; generator rather than the one in the Standard Library so we can
; allow the user to use a fixed seed to produce the same maze (with
; the same seed) or different mazes (by choosing different seeds).

RandNum      proc        near
              push       cx
              mov        cl, byte ptr Seed
              and        cl, 7
              add        cl, 4
              mov        ax, Seed
              xor        ax, 55aah
              rol        ax, cl
              xor        ax, Seed
              inc        ax
              mov        Seed, ax
              pop        cx
              ret
RandNum      endp

; Init- Handles all the initialization chores for the main program.
;       In particular, it initializes the coroutine package, gets a
;       random number seed from the user, and initializes the video display.

Init         proc        near
              print      "Enter a small integer for a random number seed:",0
              getsm
              atoi
              free
              mov        Seed, ax

; Fill the interior of the maze with wall characters, fill the outside
; two rows and columns with nowall values. This will prevent the demons
; from wandering outside the maze.

; Fill the first row with Visited values.

```

```

        cld
        mov     cx, WordsPerRow
        lesi   Maze
        mov     ax, Visited
    rep     stosw

; Fill the last row with NoWall values.

        mov     cx, WordsPerRow
        lea     di, Maze+(MaxYCoord+1)*BytesPerRow
    rep     stosw

; Write a NoWall value to the starting position:

        mov     Maze+(StartY*WordsPerRow+StartX)*2, NoWall

; Write NoWall values along the two vertical edges of the maze.

EdgesLoop:    lesi     Maze
              mov     cx, MaxYCoord+1
              mov     es:[di], ax           ;Plug the left edge.
              mov     es:[di+BytesPerRow-2], ax   ;Plug the right edge.
              add     di, BytesPerRow
              loop    EdgesLoop

              ifdef    ToScreen

; Okay, fill the screen with WallChar values:

              lesi     Screen
              mov     ax, WallChar
              mov     cx, 2000
    rep     stosw

; Write appropriate characters to the starting and ending locations:

              mov     word ptr es:Screen+EndLoc, PathChar
              mov     word ptr es:Screen+StartLoc, NoWallChar

              endif    ;ToScreen

; Zero out the DemonList:

              mov     cx, (size pcb)*MaxDemons
              lea     di, DemonList
              mov     ax, dseg
              mov     es, ax
              xor     ax, ax
    rep     stosb

Init         ret
            endp

; CanStart- This function checks around the current position
; to see if the maze generator can start digging a new tunnel
; in a direction perpendicular to the current tunnel. You can
; only start a new tunnel if there are wall characters for at
; least two positions in the desired direction:
;
;
;           ##
;           *##
;           ##
;
; If "*" is current position and "#" represent wall characters
; and the current direction is north or south, then it is okay
; for the maze generator to start a new path in the east dir-
; ection. Assuming "." represents a tunnel, you cannot start
; a new tunnel in the east direction if any of the following
; patterns occur:

```

```

;
;      .#  #.      ##      ##      ##      ##
;      *## *##      *.#     *#.     *##     *##
;      ##  ##      ##      ##      .#     #.
;
; CanStart returns true (carry set) if we can start a new tunnel off the
; path being dug by the current demon.
;
; On entry,   dl is demon's X-Coordinate
;            dh is demon's Y-Coordinate
;            cl is demon's direction

CanStart     proc      near
             push     ax
             push     bx

             MazeAdrs          ;Compute index to demon(x,y) in maze.
             mov      bx, ax

; CL contains the current direction, 0=north, 1=south, 2=east, 3=west.
; Note that we can test bit #1 for north/south (0) or east/west (1).

             test     cl, 10b      ;See if north/south or east/west
             jz      NorthSouth

; If the demon is going in an east or west direction, we can start a new
; tunnel if there are six wall blocks just above or below the current demon.
; Note: We are checking if all values in these six blocks are Wall values.
; This code depends on the fact that Wall characters are zero and the sum
; of these six blocks will be zero if a move is possible.

             mov     al, byt Maze[bx+BytesPerRow*2] ;Maze[x, y+2]
             add     al, byt Maze[bx+BytesPerRow*2+2] ;Maze[x+1,y+2]
             add     al, byt Maze[bx+BytesPerRow*2-2] ;Maze[x-1,y+2]
             je      ReturnTrue

             mov     al, byt Maze[bx-BytesPerRow*2] ;Maze[x, y-2]
             add     al, byt Maze[bx-BytesPerRow*2+2] ;Maze[x+1,y-2]
             add     al, byt Maze[bx-BytesPerRow*2-2] ;Maze[x-1,y-2]
             je      ReturnTrue

ReturnFalse:  clc                      ;Clear carry = false.
             pop     bx
             pop     ax
             ret

; If the demon is going in a north or south direction, we can start a
; new tunnel if there are six wall blocks just to the left or right
; of the current demon.

NorthSouth:  mov     al, byt Maze[bx+4];Maze[x+2,y]
             add     al, byt Maze[bx+BytesPerRow+4];Maze[x+2,y+1]
             add     al, byt Maze[bx-BytesPerRow+4];Maze[x+2,y-1]
             je      ReturnTrue

             mov     al, byt Maze[bx-4];Maze[x-2,y]
             add     al, byt Maze[bx+BytesPerRow-4];Maze[x-2,y+1]
             add     al, byt Maze[bx-BytesPerRow-4];Maze[x-2,y-1]
             jne     ReturnFalse

ReturnTrue:  stc                      ;Set carry = true.
             pop     bx
             pop     ax
             ret

CanStart     endp

; CanMove-   Tests to see if the current demon (dir=cl, x=dl, y=dh) can
;            move in the specified direction. Movement is possible if
;            the demon will not come within one square of another tunnel.
;            This function returns true (carry set) if a move is possible.
;            On entry, CH contains the direction this code should test.

```

```

CanMove      proc
              push      ax
              push      bx

              MazeAdrs          ;Put @Maze[x,y] into ax.
              mov         bx, ax

              cmp         ch, South
              jb         IsNorth
              je         IsSouth
              cmp        ch, East
              je         IsEast

; If the demon is moving west, check the blocks in the rectangle formed
; by Maze[x-2,y-1] to Maze[x-1,y+1] to make sure they are all wall values.

              mov         al, byp Maze[bx-BytesPerRow-4];Maze[x-2, y-1]
              add         al, byp Maze[bx-BytesPerRow-2];Maze[x-1, y-1]
              add         al, byp Maze[bx-4];Maze[x-2, y]
              add         al, byp Maze[bx-2];Maze[x-1, y]
              add         al, byp Maze[bx+BytesPerRow-4];Maze[x-2, y+1]
              add         al, byp Maze[bx+BytesPerRow-2];Maze[x-1, y+1]
              je         ReturnTrue
ReturnFalse:  clc
              pop         bx
              pop         ax
              ret

; If the demon is going east, check the blocks in the rectangle formed
; by Maze[x+1,y-1] to Maze[x+2,y+1] to make sure they are all wall values.

IsEast:      mov         al, byp Maze[bx-BytesPerRow+4];Maze[x+2, y-1]
              add         al, byp Maze[bx-BytesPerRow+2];Maze[x+1, y-1]
              add         al, byp Maze[bx+4];Maze[x+2, y]
              add         al, byp Maze[bx+2];Maze[x+1, y]
              add         al, byp Maze[bx+BytesPerRow+4];Maze[x+2, y+1]
              add         al, byp Maze[bx+BytesPerRow+2];Maze[x+1, y+1]
              jne        ReturnFalse
ReturnTrue:  stc
              pop         bx
              pop         ax
              ret

; If the demon is going north, check the blocks in the rectangle formed
; by Maze[x-1,y-2] to Maze[x+1,y-1] to make sure they are all wall values.

IsNorth:     mov         al, byp Maze[bx-BytesPerRow-2];Maze[x-1, y-1]
              add         al, byp Maze[bx-BytesPerRow*2-2];Maze[x-1, y-2]
              add         al, byp Maze[bx-BytesPerRow];Maze[x, y-1]
              add         al, byp Maze[bx-BytesPerRow*2];Maze[x, y-2]
              add         al, byp Maze[bx-BytesPerRow+2];Maze[x+1, y-1]
              add         al, byp Maze[bx-BytesPerRow*2+2];Maze[x+1, y-2]
              jne        ReturnFalse
              stc
              pop         bx
              pop         ax
              ret

; If the demon is going south, check the blocks in the rectangle formed
; by Maze[x-1,y+2] to Maze[x+1,y+1] to make sure they are all wall values.

IsSouth:     mov         al, byp Maze[bx+BytesPerRow-2];Maze[x-1, y+1]
              add         al, byp Maze[bx+BytesPerRow*2-2];Maze[x-1, y+2]
              add         al, byp Maze[bx+BytesPerRow];Maze[x, y+1]
              add         al, byp Maze[bx+BytesPerRow*2];Maze[x, y+2]
              add         al, byp Maze[bx+BytesPerRow+2];Maze[x+1, y+1]
              add         al, byp Maze[bx+BytesPerRow*2+2];Maze[x+1, y+2]
              jne        ReturnFalse
              stc

```

```

                pop     bx
                pop     ax
                ret

CanMove        endp

; SetDir- Changes the current direction. The maze digging algorithm has
; decided to change the direction of the tunnel begin dug by one
; of the demons. This code checks to see if we CAN change the direction,
; and picks a new direction if possible.
;
; If the demon is going north or south, a direction change causes the demon
; to go east or west. Likewise, if the demon is going east or west, a
; direction change forces it to go north or south. If the demon cannot
; change directions (because it cannot move in the new direction for one
; reason or another), SetDir returns without doing anything. If a direction
; change is possible, then SetDir selects a new direction. If there is only
; one possible new direction, the demon is sent off in that direction.
; If the demon could move off in one of two different directions, SetDir
; "flips a coin" to choose one of the two new directions.
;
; This function returns the new direction in al.

SetDir        proc     near

                test     cl, 10b           ;See if north/south
                je      IsNS              ; or east/west direction.

; We're going east or west. If we can move EITHER north or south from
; this point, randomly choose one of the directions. If we can only
; move one way or the other, choose that direction. If we can't go either
; way, return without changing the direction.

                mov     ch, North         ;See if we can move north
                call    CanMove
                jnc     NotNorth
                mov     ch, South        ;See if we can move south
                call    CanMove
                jnc     DoNorth
                call    RandNum          ;Get a random direction
                and     ax, 1            ;Make it north or south.
                ret

DoNorth:      mov     ax, North
                ret

NotNorth:     mov     ch, South
                call    CanMove
                jnc     TryReverse

DoSouth:      mov     ax, South
                ret

; If the demon is moving north or south, choose a new direction of east
; or west, if possible.

IsNS:        mov     ch, East           ;See if we can move East
                call    CanMove
                jnc     NotEast
                mov     ch, West        ;See if we can move West
                call    CanMove
                jnc     DoEast
                call    RandNum          ;Get a random direction
                and     ax, 1b          ;Make it East or West
                or     al, 10b
                ret

DoEast:      mov     ax, East
                ret

```

```

DoWest:      mov     ax, West
             ret

NotEast:     mov     ch, West
             call    CanMove
             jc      DoWest

; Gee, we can't switch to a perpendicular direction, see if we can
; turn around.

TryReverse:  mov     ch, cl
             xor     ch, 1
             call    CanMove
             jc      ReverseDir

; If we can't turn around (likely), then keep going in the same direction.

             mov     ah, 0
             mov     al, cl           ;Stay in same direction.
             ret

; Otherwise reverse direction down here.

ReverseDir:  mov     ah, 0
             mov     al, cl
             xor     al, 1
             ret

SetDir      endp

; Stuck-      This function checks to see if a demon is stuck and cannot
;             move in any direction. It returns true if the demon is
;             stuck and needs to be killed.

Stuck       proc     near
             mov     ch, North
             call    CanMove
             jc      NotStuck
             mov     ch, South
             call    CanMove
             jc      NotStuck
             mov     ch, East
             call    CanMove
             jc      NotStuck
             mov     ch, West
             call    CanMove
NotStuck:    ret
Stuck       endp

; NextDemon- Searches through the demon list to find the next available
;             active demon. Return a pointer to this guy in es:di.

NextDemon   proc     near
             push    ax

NDLoop:     inc     DemonIndex           ;Move on to next demon,
             and     DemonIndex, ModDemons ; MOD MaxDemons.
             mov     al, size pcb       ;Compute index into
             mul     DemonIndex         ; DemonList.
             mov     di, ax             ;See if the demon at this
             add     di, offset DemonList ; offset is active.
             cmp     byp [di].pcb.NextProc, 0
             je      NDLLoop

             mov     ax, ds
             mov     es, ax
             pop     ax
             ret
NextDemon   endp

```

```

; Dig-          This is the demon process.
;              It moves the demon one position (if possible) in its current
;              direction. After moving one position forward, there is
;              a 25% chance that this guy will change its direction; there
;              is a 25% chance this demon will spawn a child process to
;              dig off in a perpendicular direction.

Dig            proc        near

; See if the current demon is stuck. If the demon is stuck, then we've
; go to remove it from the demon list. If it is not stuck, then have it
; continue digging. If it is stuck and this is the last active demon,
; then return control to the main program.

                call       Stuck
                jc         NotStuck

; Okay, kill the current demon.
; Note: this will never kill the last demon because we have the timer
; process running. The timer process is the one that always stops
; the program.

                dec        DemonCnt

; Since the count is not zero, there must be more demons in the demon
; list. Free the stack space associated with the current demon and
; then search out the next active demon and have at it.

MoreDemons:    mov         al, size pcb
                mul        DemonIndex
                mov        bx, ax

; Free the stack space associated with this process. Note this code is
; naughty. It assumes the stack is allocated with the Standard Library
; malloc routine that always produces a base address of 8.

                mov        es, DemonList[bx].regss
                mov        di, 8                                ;Cheating!
                free

; Mark the demon entry for this guy as unused.

                mov        byp DemonList[bx].NextProc, 0      ;Mark as unused.

; Okay, locate the next active demon in the list.

FndNxtDmn:     call        NextDemon
                cocal     ;Never returns

; If the demon is not stuck, then continue digging away.

NotStuck:      mov         ch, cl
                call       CanMove
                jnc        DontMove

; If we can move, then adjust the demon's coordinates appropriately:

                cmp        cl, South
                jb         MoveNorth
                je         MoveSouth
                cmp        cl, East
                jne        MoveWest

; Moving East:

                inc        dl
                jmp        MoveDone

MoveWest:      dec         dl

```

```

                                jmp      MoveDone
MoveNorth:  dec      dh
                                jmp      MoveDone

MoveSouth: inc      dh

; Okay, store a NoWall value at this entry in the maze and output a NoWall
; character to the screen (if writing data to the screen).

MoveDone:  MazeAdrs
           mov      bx, ax
           mov      Maze[bx], NoWall

           ifdef   ToScreen
           ScrnAdrs
           mov      bx, ax
           push    es
           mov      ax, ScreenSeg
           mov      es, ax
           mov      word ptr es:[bx], NoWallChar
           pop     es
           endif

; Before leaving, see if this demon shouldn't change direction.

DontMove:  call     RandNum
           and     al, 11b           ;25% chance result is zero.
           jne    NoChangeDir
           call    SetDir
           mov     cl, al

NoChangeDir:

; Also, see if this demon should spawn a child process

           call    RandNum
           and     al, 11b           ;Give it a 25% chance.
           jne    NoSpawn

; Okay, see if it's possible to spawn a new process at this point:

           call    CanStart
           jnc    NoSpawn

; See if we've already got MaxDemons active:

           cmp     DemonCnt, MaxDemons
           jae    NoSpawn

           inc     DemonCnt           ;Add another demon.

; Okay, create a new demon and add him to the list.

           push   dx                   ;Save cur demon info.
           push   cx

; Locate a free slot for this demon

FindSlot:  lea     si, DemonList- size pcb
           add     si, size pcb
           cmp     byt [si].pcb.NextProc, 0
           jne    FindSlot

; Allocate some stack space for the new demon.

           mov     cx, 256             ;256 byte stack.
           malloc

; Set up the stack pointer for this guy:

```

```

        add     di, 248           ;Point stack at end.
        mov     [si].pcb.regss, es
        mov     [si].pcb.regsp, di

; Set up the execution address for this guy:

        mov     [si].pcb.regcs, cs
        mov     [si].pcb.regip, offset Dig

; Initial coordinates and direction for this guy:

        mov     [si].pcb.regdx, dx

; Select a direction for this guy.

        pop     cx               ;Retrieve direction.
        push    cx

        call    SetDir
        mov     ah, 0
        mov     [si].pcb.regcx, ax

; Set up other misc junk:

        mov     [si].pcb.regds, seg dseg
        sti
        pushf
        pop     [si].pcb.regflags
        mov     byp [si].pcb.NextProc, 1       ;Mark active.

; Restore current process' parameters

        pop     cx               ;Restore current demon.
        pop     dx

NoSpawn:

; Okay, with all of the above done, it's time to pass control on to a new
; digger. The following cocall passes control to the next digger in the
; DemonList.

GetNextDmn:  call    NextDemon

; Okay, we've got a pointer to the next demon in the list (might be the
; same demon if there's only one), pass control to that demon.

        cocall
        jmp     Dig
Dig         endp

; TimerDemon- This demon introduces a delay between
;             each cycle in the demon list. This slows down the
;             maze generation so you can see the maze being built
;             (which makes the program more interesting to watch).

TimerDemon  proc     near
            push    es
            push    ax

            mov     ax, 40h           ;BIOS variable area
            mov     es, ax
            mov     ax, es:[6Ch]     ;BIOS timer location
Wait4Change:  cmp     ax, es:[6Ch]       ;BIOS changes this every
            je     Wait4Change       ; 1/18th second.

            cmp     DemonCnt, 1
            je     QuitProgram
            pop     es
            pop     ax
            call    NextDemon
            cocall
            jmp     TimerDemon

```

```

QuitProgram:  cocall    MainPCB          ;Quit the program
TimerDemon   endp

; What good is a maze generator program if it cannot solve the mazes it
; creates? SolveMaze finds the solution (if any) for this maze. It marks
; the solution path and the paths it tried, but failed on.
;
; function solvemaze(x,y:integer):boolean

sm_X          textequ    <[bp+6]>
sm_Y          textequ    <[bp+4]>

SolveMaze     proc      near
              push      bp
              mov       bp, sp

; See if we've just solved the maze:

              cmp       byte ptr sm_X, EndX
              jne       NotSolved
              cmp       byte ptr sm_Y, EndY
              jne       NotSolved
              mov       ax, 1          ;Return true.
              pop       bp
              ret       4

; See if moving to this spot was an illegal move. There will be
; a NoWall value at this cell in the maze if the move is legal.

NotSolved:    mov       dl, sm_X
              mov       dh, sm_Y
              MazeAdrs
              mov       bx, ax
              cmp       Maze[bx], NoWall
              je        MoveOK
              mov       ax, 0          ;Return failure
              pop       bp
              ret       4

; Well, it is possible to move to this point, so place an appropriate
; value on the screen and keep searching for the solution.

MoveOK:       mov       Maze[bx], Visited

              ifdef    ToScreen
              push     es              ;Write a "VisitChar"
              ScrnAdrs          ; character to the
              mov      bx, ax          ; screen at this X,Y
              mov      ax, ScreenSeg   ; position.
              mov      es, ax
              mov      word ptr es:[bx], VisitChar
              pop      es
              endif

; Recursively call SolveMaze until we get a solution. Just call SolveMaze
; for the four possible directions (up, down, left, right) we could go.
; Since we've left "Visited" values in the Maze, we will not accidentally
; search back through the path we've already travelled. Furthermore, if
; we cannot go in one of the four directions, SolveMaze will catch this
; immediately upon entry (see the code at the start of this routine).

              mov      ax, sm_X          ;Try the path at location
              dec      ax              ; (X-1, Y)
              push     ax
              push     sm_Y
              call     SolveMaze
              test     ax, ax          ;Solution?
              jne     Solved

              push     sm_X          ;Try the path at location

```

```

mov     ax, sm_Y           ; (X, Y-1)
dec     ax
push   ax
call   SolveMaze
test   ax, ax             ;Solution?
jne    Solved

mov     ax, sm_X           ;Try the path at location
inc     ax                 ; (X+1, Y)
push   ax
push   sm_Y
call   SolveMaze
test   ax, ax             ;Solution?
jne    Solved

push   sm_X               ;Try the path at location
mov     ax, sm_Y           ; (X, Y+1)
inc     ax
push   ax
call   SolveMaze
test   ax, ax             ;Solution?
jne    Solved
pop    bp
ret     4

Solved:
ifdef  ToScreen           ;Draw return path.
push   es
mov     dl, sm_X
mov     dh, sm_Y
ScrnAdrs
mov     bx, ax
mov     ax, ScreenSeg
mov     es, ax
mov     word ptr es:[bx], PathChar
pop     es
mov     ax, 1              ;Return true
endif

pop    bp
ret     4
SolveMaze endp

```

```

; Here's the main program that drives the whole thing:

```

```

Main    proc
mov     ax, dseg
mov     ds, ax
mov     es, ax
meminit

call    Init              ;Initialize maze stuff.
lesi    MainPCB           ;Initialize coroutine
coinit  ; package.

; Create the first demon.
; Set up the stack pointer for this guy:

mov     cx, 256
malloc
add     di, 248
mov     DemonList.regsp, di
mov     DemonList.regss, es

; Set up the execution address for this guy:

mov     DemonList.regcs, cs
mov     DemonList.regip, offset Dig

; Initial coordinates and direction for this guy:

```

```

        mov     cx, East           ;Start off going east.
        mov     dh, StartY
        mov     dl, StartX
        mov     DemonList.regcx, cx
        mov     DemonList.regdx, dx

; Set up other misc junk:

        mov     DemonList.regds, seg dseg
        sti
        pushf
        pop     DemonList.regflags
        mov     byp DemonList.NextProc, 1       ;Demon is "active".
        inc     DemonCnt
        mov     DemonIndex, 0

; Set up the Timer demon:

        mov     DemonList.regsp+(size pcb), offset EndTimerStk
        mov     DemonList.regss+(size pcb), ss

; Set up the execution address for this guy:

        mov     DemonList.regcs+(size pcb), cs
        mov     DemonList.regip+(size pcb), offset TimerDemon

; Set up other misc junk:

        mov     DemonList.regds+(size pcb), seg dseg
        sti
        pushf
        pop     DemonList.regflags+(size pcb)
        mov     byp DemonList.NextProc+(size pcb), 1
        inc     DemonCnt

; Start the ball rolling.

        mov     ax, ds
        mov     es, ax
        lea     di, DemonList
        ccall

; Wait for the user to press a key before solving the maze:

        getc

        mov     ax, StartX
        push    ax
        mov     ax, StartY
        push    ax
        call    SolveMaze

; Wait for another keystroke before quitting:

        getc

        mov     ax, 3             ;Clear screen and reset video mode.
        int     10h

Quit:   ExitPgm                   ;DOS macro to quit program.
Main   endp

cseg    ends

sseg    segment    para stack 'stack'

; Stack for the timer demon we create (we'll allocate the other
; stacks dynamically).

TimerStk    byte    256 dup (?)
EndTimerStk word    ?

```

```

; Main program's stack:

stk          byte    512 dup (?)
sseg        ends

zzzzzzseg   segment  para public 'zzzzzz'
LastBytes   db       16 dup (?)
zzzzzzseg   ends
end         Main

```

The existing Standard Library coroutine package is not suitable for programs that use the 80386 and later 32 bit register sets. As mentioned earlier, the problem lies in the fact that the Standard Library only preserves the 16-bit registers when switching between processes. However, it is a relatively trivial extension to modify the Standard Library so that it saves 32 bit registers. To do so, just change the definition of the pcb (to make room for the 32 bit registers) and the s1_cocall routine:

```

                .386
                option    segment:usel6

dseg           segment  para public 'data'

wp            equ       <word ptr>

; 32-bit PCB. Note we only keep the L.O. 16 bits of SP since we are
; operating in real mode.

pcb32         struc
regsp         word     ?
regss         word     ?
regip         word     ?
regcs         word     ?

regeax        dword   ?
regebx        dword   ?
regecx        dword   ?
regedx        dword   ?
regesi        dword   ?
regedi        dword   ?
regebp        dword   ?

regds         word     ?
reges         word     ?
regflags      dword   ?
pcb32         ends

DefaultPCB    pcb32    <>
DefaultCortn  pcb32    <>

CurCoroutine  dword    DefaultCortn ;Points at the currently executing
; coroutine.

dseg          ends

cseg          segment  para public 'slcode'

;=====
;
; 32-Bit Coroutine support.
;
; COINIT32- ES:DI contains the address of the current (default) process' PCB.

CoInit32      proc     far
                assume  ds:dseg
                push    ax

```

```

        push    ds
        mov     ax, dseg
        mov     ds, ax
        mov     wp dseg:CurCoroutine, di
        mov     wp dseg:CurCoroutine+2, es
        pop     ds
        pop     ax
CoInit32  endp

```

; COCALL32- transfers control to a coroutine. ES:DI contains the address
; of the PCB. This routine transfers control to that coroutine and then
; returns a pointer to the caller's PCB in ES:DI.

```

cocall32  proc     far
          assume   ds:dseg
          pushfd
          push     ds
          push     es           ;Save these for later
          push     edi
          push     eax
          mov     ax, dseg
          mov     ds, ax
          cli           ;Critical region ahead.

```

; Save the current process' state:

```

        les     di, dseg:CurCoroutine
        pop     es:[di].pcb32.regeax
        mov     es:[di].pcb32.regebx, ebx
        mov     es:[di].pcb32.regecx, ecx
        mov     es:[di].pcb32.regedx, edx
        mov     es:[di].pcb32.regesi, esi
        pop     es:[di].pcb32.regedi
        mov     es:[di].pcb32.regebp, ebp

        pop     es:[di].pcb32.reges
        pop     es:[di].pcb32.regds
        pop     es:[di].pcb32.regflags
        pop     es:[di].pcb32.regip
        pop     es:[di].pcb32.regcs
        mov     es:[di].pcb32.regsp, sp
        mov     es:[di].pcb32.regss, ss

        mov     bx, es           ;Save so we can return in
        mov     ecx, edi        ; ES:DI later.
        mov     edx, es:[di].pcb32.regedi
        mov     es, es:[di].pcb32.reges
        mov     di, dx          ;Point es:di at new PCB

        mov     wp dseg:CurCoroutine, di
        mov     wp dseg:CurCoroutine+2, es

        mov     es:[di].pcb32.regedi, ecx ;The ES:DI return values.
        mov     es:[di].pcb32.reges, bx

```

; Okay, switch to the new process:

```

        mov     ss, es:[di].pcb32.regss
        mov     sp, es:[di].pcb32.regsp
        mov     eax, es:[di].pcb32.regeax
        mov     ebx, es:[di].pcb32.regebx
        mov     ecx, es:[di].pcb32.regecx
        mov     edx, es:[di].pcb32.regedx
        mov     esi, es:[di].pcb32.regesi
        mov     ebp, es:[di].pcb32.regebp
        mov     ds, es:[di].pcb32.regds

        push    es:[di].pcb32.regflags
        push    es:[di].pcb32.regcs
        push    es:[di].pcb32.regip
        push    es:[di].pcb32.regedi

```

```

                                mov     es, es:[di].pcb32.reges
                                pop     edi
                                iret
cocall32    endp

```

```

; CoCall321 works just like cocall above, except the address of the pcb
; follows the call in the code stream rather than being passed in ES:DI.
; Note: this code does *not* return the caller's PCB address in ES:DI.
;

```

```

cocall321    proc     far
              assume  ds:dseg
              push    ebp
              mov     bp, sp
              pushfd
              push    ds
              push    es
              push    edi
              push    eax
              mov     ax, dseg
              mov     ds, ax
              cli
                                ;Critical region ahead.

```

```

; Save the current process' state:

```

```

              les     di, dseg:CurCoroutine
              pop     es:[di].pcb32.regeax
              mov     es:[di].pcb32.regebx, ebx
              mov     es:[di].pcb32.regecx, ecx
              mov     es:[di].pcb32.regedx, edx
              mov     es:[di].pcb32.regesi, esi
              pop     es:[di].pcb32.regedi
              pop     es:[di].pcb32.reges
              pop     es:[di].pcb32.regds
              pop     es:[di].pcb32.regflags
              pop     es:[di].pcb32.regebp
              pop     es:[di].pcb32.regip
              pop     es:[di].pcb32.regcs
              mov     es:[di].pcb32.regsp, sp
              mov     es:[di].pcb32.regss, ss

              mov     dx, es:[di].pcb32.regip ;Get return address (ptr to
              mov     cx, es:[di].pcb32.regcs ; PCB address.
              add     es:[di].pcb32.regip, 4 ;Skip ptr on return.
              mov     es, cx                ;Get the ptr to the new pcb
              mov     di, dx                ; address, then fetch the
              les     di, es:[di]          ; pcb val.
              mov     wp dseg:CurCoroutine, di
              mov     wp dseg:CurCoroutine+2, es

```

```

; Okay, switch to the new process:

```

```

              mov     ss, es:[di].pcb32.regss
              mov     sp, es:[di].pcb32.regsp
              mov     eax, es:[di].pcb32.regeax
              mov     ebx, es:[di].pcb32.regebx
              mov     ecx, es:[di].pcb32.regecx
              mov     edx, es:[di].pcb32.regedx
              mov     esi, es:[di].pcb32.regesi
              mov     ebp, es:[di].pcb32.regebp
              mov     ds, es:[di].pcb32.regds

              push    es:[di].pcb32.regflags
              push    es:[di].pcb32.regcs
              push    es:[di].pcb32.regip
              push    es:[di].pcb32.regedi
              mov     es, es:[di].pcb32.reges
              pop     edi
              iret

```

```

cocall321    endp
cseg        ends

```

19.4 Multitasking

Coroutines provide a reasonable mechanism for switching between processes that must take turns. For example, the maze generation program in the previous section would generate poor mazes if the daemon processes didn't take turns removing one cell at a time from the maze. However, the coroutine paradigm isn't always suitable; not all processes need to take turns. For example, suppose you are writing an action game where the user plays against the computer. In addition, the computer player operates independently of the user in real time. This could be, for example, a space war game or a flight simulator game (where you are dog fighting other pilots). Ideally, we would like to have *two* computers. One to handle the user interaction and one for the computer player. Both systems would communicate their moves to one another during the game. If the (human) player simply sits and watches the screen, the computer player would win since it is active and the human player is not. Of course, it would considerably limit the marketability of your game were it to require two computers to play. However, you can use *multitasking* to simulate two separate computer systems on a single CPU.

The basic idea behind multitasking is that one process runs for a period of time (the *time quantum* or *time slice*) and then a timer interrupts the process. The timer ISR saves the state of the process and then switches control to another process. That process runs for its time slice and then the timer interrupt switches to another process. In this manner, each process gets some amount of computer time. Note that multitasking is very easy to implement if you have a coroutine package. All you need to do is write a timer ISR that coccalls the various processes, one per timer interrupt. A timer interrupt that switches between processes is a *dispatcher*.

One decision you will need to make when designing a dispatcher is a policy for the process selection algorithm. A simple policy is to place all processes in a queue and then rotate among them. This is known as the *round-robin policy*. Since this is the policy the UCR Standard Library process package uses, we will adopt it as well. However, there are other process selection criteria, generally involving the priority of a process, available as well. See a good text on operating systems for details.

The choice of the time quantum can have a big impact on performance. Generally, you would like the time quantum to be small. The time sharing (switching between processes based on the clock) will be much smoother if you use small time quanta. For example, suppose you choose five second time quanta and you were running four processes concurrently. Each process would get five seconds; it would run very fast during those five seconds. However, at the end of its time slice it would have to wait for the other three process' turns, 15 seconds, before it ran again. The users of such programs would get very frustrated with them, users like programs whose performance is relatively consistent from one moment to the next.

If we make the time slice one millisecond, instead of five seconds, each process would run for one millisecond and then switch to the next processes. This means that each processes gets one millisecond out of five. This is too small a time quantum for the user to notice the pause between processes.

Since smaller time quanta seem to be better, you might wonder "why not make them as small as possible?" For example, the PC supports a one millisecond timer interrupt. Why not use that to switch between processes? The problem is that there is a fair amount of overhead required to switch from one processes to another. The smaller you make the time quantum, the larger will be the overhead of using time slicing. Therefore, you want to pick a time quantum that is a good balance between smooth process switching and too much overhead. As it turns out, the $1/18$ th second clock is probably fine for most multitasking requirements.

19.4.1 Lightweight and HeavyWeight Processes

There are two major types of processes in the world of multitasking: *lightweight processes*, also known as *threads*, and *heavyweight processes*. These two types of processes differ mainly in the details of memory management. A heavyweight process swaps memory management tables and moves lots of data

around. Threads only swap the stack and CPU registers. Threads have much less overhead cost than heavyweight processes.

We will not consider heavyweight processes in this text. Heavyweight processes appear in protected mode operating systems like UNIX, Linux, OS/2, or Windows NT. Since there is rarely any memory management (at the hardware level) going on under DOS, the issue of changing memory management tables around is moot. Switching from one heavyweight application to another generally corresponds to switching from one application to another.

Using lightweight processes (threads) is perfectly reasonable under DOS. Threads (short for “execution thread” or “thread of execution”) correspond to two or more concurrent execution paths within the same program. For example, we could think of each of the demons in the maze generation program as being a separate thread of execution.

Although threads have different stacks and machine states, they share code and data memory. There is no need to use a “shared memory TSR” to provide global shared memory (see “Shared Memory” on page 1078). Instead, maintaining *local* variables is the difficult task. You must either allocate local variables on the process’ stack (which is separate for each process) or you’ve got to make sure that no other process uses the variables you declare in the data segment specifically for one thread.

We could easily write our own threads package, but we don’t have to; the UCR Standard Library provides this capability in the *processes package*. To see how to incorporate threads into your programs, keep reading...

19.4.2 The UCR Standard Library Processes Package

The UCR Standard Library provides six routines to let you manage threads. These routines include `prcsinit`, `prcsquit`, `fork`, `die`, `kill`, and `yield`. These functions let you initialize and shut down the threads system, start new processes, terminate processes, and voluntarily pass the CPU off to another process.

The `prcsinit` and `prcsquit` functions let you initialize and shutdown the system. The `prcsinit` call prepares the threads package. You must call this routine before executing any of the other five process routines. The `prcsquit` function shuts down the threads system in preparation for program termination. `Pracsinit` patches into the timer interrupt (interrupt 8). `Pracsquit` restores the interrupt 8 vector. It is very important that you call `prcsquit` before your program returns to DOS. Failure to do so will leave the int 8 vector pointing off into memory which may cause the system to crash when DOS loads the next program. Your program must patch the break and critical error exception vectors to ensure that you call `prcsquit` in the event of abnormal program termination. Failure to do so may crash the system if the user terminates the program with `ctrl-break` or an abort on an I/O error. `Pracsinit` and `prcsquit` do not require any parameters, nor do they return any values.

The `fork` call spawns a new process. On entry, `es:di` must point at a `pcb` for the new process. The `regss` and `regsp` fields of the `pcb` must contain the address of the *top* of the stack area for this new process. The `fork` call fills in the other fields of the `pcb` (including `cs:ip`)/

For each call you make to `fork`, the `fork` routine returns *twice*, once for each thread of execution. The parent process *typically* returns first, but this is not certain; the child process is usually the second return from the `fork` call. To differentiate the two calls, `fork` returns two process identifiers (PIDs) in the `ax` and `bx` registers. For the parent process, `fork` returns with `ax` containing zero and `bx` containing the PID of the child process. For the child process, `fork` returns with `ax` containing the child’s PID and `bx` containing zero. Note that both threads return and continuing executing the same code after the call to `fork`. If you want the child and parent processes to take separate paths, you would execute code like the following:

```

    lesi      NewPCB          ;Assume regss/regsp are initialized.
    fork
    test     ax, ax          ;Parent PID is zero at this point.
    je      ParentProcess    ;Go elsewhere if parent process.

; Child process continues execution here

```

The parent process should save the child's PID. You can use the PID to terminate a process at some later time.

It is important to repeat that you must initialize the `regss` and `regsp` fields in the `pcb` before calling `fork`. You must allocate storage for a stack (dynamically or statically) and point `ss:sp` at the last word of this stack area. Once you call `fork`, the process package uses whatever value that happens to be in the `regss` and `regsp` fields. If you have not initialized these values, they will probably contain zero and when the process starts it will wipe out the data at address `0:FFFE`. This may crash the system at one point or another.

The `die` call kills the current process. If there are multiple processes running, this call transfers control to some other processes waiting to run. If the current process is the only process on the system's *run queue*, then this call will crash the system.

The `kill` call lets one process terminate another. Typically, a parent process will use this call to terminate a child process. To kill a process, simply load the `ax` register with the PID of the process you want to terminate and then call `kill`. If a process supplies its own PID to the `kill` function, the process terminates itself (that is, this is equivalent to a `die` call). If there is only one process in the run queue and that process kills itself, the system will crash.

The last multitasking management routine in the process package is the `yield` call. `Yield` voluntarily gives up the CPU. This is a direct call to the dispatcher, that will switch to another task in the run queue. Control returns after the `yield` call when the next time slice is given to this process. If the current process is the only one in the queue, `yield` immediately returns. You would normally use the `yield` call to free up the CPU between long I/O operations (like waiting for a keypress). This would allow other tasks to get maximum use of the CPU while your process is just spinning in a loop waiting for some I/O operation to complete.

The Standard Library multitasking routines only work with the 16 bit register set of the 80x86 family. Like the coroutine package, you will need to modify the `pcb` and the dispatcher code if you want to support the 32 bit register set of the 80386 and later processors. This task is relatively simple and the code is quite similar to that appearing in the section on coroutines; so there is no need to present the solution here.

19.4.3 Problems with Multitasking

When threads share code and data certain problems can develop. First of all, reentrancy becomes a problem. You cannot call a non-reentrant routine (like DOS) from two separate threads if there is ever the possibility that the non-reentrant code could be interrupted and control transferred to a second thread that reenters the same routine. Reentrancy is not the only problem, however. It is quite possible to design two routines that access shared variables and those routines misbehave depending on where the interrupts occur in the code sequence. We will explore these problems in the section on synchronization (see "Synchronization" on page 1129), just be aware, for now, that these problems exist.

Note that simply turning off the interrupts (with `cli`) may not solve the reentrancy problem. Consider the following code:

```

    cli                ;Prevent reentrancy.
    mov     ah, 3Eh    ;DOS close call.
    mov     bx, Handle
    int     21h
    sti                ;Turn interrupts back on.

```

This code will not prevent DOS from being reentered because DOS (and BIOS) turn the interrupts back on! There is a solution to this problem, but it's not by using `ccli` and `sti`.

19.4.4 A Sample Program with Threads

The following program provides a simple demonstration of the Standard Library processes package. This short program creates two threads – the main program and a timer process. On each timer tick the background (timer) process kicks in and increments a memory variable. It then yields the CPU back to the main program. On the next timer tick control returns to the background process and this cycle repeats. The main program reads a string from the user while the background process is counting off timer ticks. When the user finishes the line by pressing the enter key, the main program kills the background process and then prints the amount of time necessary to enter the line of text.

Of course, this isn't the most efficient way to time how long it takes someone to enter a line of text, but it does provide an example of the multitasking features of the Standard Library. This short program segment demonstrates all the process routines except `die`. Note that it also demonstrates the fact that you must supply `int 23h` and `int 24h` handlers when using the process package.

```

; MULTI.ASM
; Simple program to demonstrate the use of multitasking.

                .xlist
                include  stdlib.a
                includelib stdlib.lib
                .list

dseg            segment    para public 'data'

ChildPID        word      0                ;Child's PID so we can kill it.
BackGndCnt      word      0                ;Counts off clock ticks in backgnd.

; PCB for our background process. Note we initialize ss:sp here.

BkgndPCB        pcb       {0,offset EndStk2, seg EndStk2}

; Data buffer to hold an input string.

InputLine       byte     128 dup (0)

dseg            ends

cseg            segment    para public 'code'
                assume    cs:cseg, ds:dseg

; A replacement critical error handler. This routine calls prcsquit
; if the user decides to abort the program.

CritErrMsg      byte     cr,lf
                byte     "DOS Critical Error!",cr,lf
                byte     "A)bort, R)etry, I)gnore, F)ail? $"

MyInt24         proc      far
                push     dx
                push     ds
                push     ax

                push     cs
                pop      ds
Int24Lp:        lea     dx, CritErrMsg
                mov     ah, 9                ;DOS print string call.
                int     21h

                mov     ah, 1                ;DOS read character call.
                int     21h

```

```

                                and     al, 5Fh           ;Convert l.c. -> u.c.
                                cmp     al, 'I'           ;Ignore?
                                jne     NotIgnore
                                pop     ax
                                mov     al, 0
                                jmp     Quit24
NotIgnore:                       cmp     al, 'r'           ;Retry?
                                jne     NotRetry
                                pop     ax
                                mov     al, 1
                                jmp     Quit24
NotRetry:                        cmp     al, 'A'           ;Abort?
                                jne     NotAbort
                                prcsquit                ;If quitting, fix INT 8.
                                pop     ax
                                mov     al, 2
                                jmp     Quit24
NotAbort:                        cmp     al, 'F'
                                jne     BadChar
                                pop     ax
Quit24:                          pop     al, 3
                                pop     ds
                                pop     dx
                                ired
BadChar:                         mov     ah, 2
                                mov     dl, 7             ;Bell character
                                jmp     Int24Lp
MyInt24                          endp

; We will simply disable INT 23h (the break exception).
MyInt23                          proc     far
                                ired
MyInt23                          endp

; Okay, this is a pretty weak background process, but it does demonstrate
; how to use the Standard Library calls.
Background                        proc
                                sti
                                mov     ax, dseg
                                mov     ds, ax
                                inc     BackGndCnt       ;Bump call Counter by one.
                                yield                    ;Give CPU back to foregnd.
                                jmp     Background
Background                        endp

Main                              proc
                                mov     ax, dseg
                                mov     ds, ax
                                mov     es, ax
                                meminit

; Initialize the INT 23h and INT 24h exception handler vectors.
                                mov     ax, 0
                                mov     es, ax
                                mov     word ptr es:[24h*4], offset MyInt24
                                mov     es:[24h*4 + 2], cs
                                mov     word ptr es:[23h*4], offset MyInt23
                                mov     es:[23h*4 + 2], cs

                                prcsinit                ;Start multitasking system.

```

```

        lesi      BkgndPCB          ;Fire up a new process
        fork
        test     ax, ax           ;Parent's return?
        je      ParentPrCs
        jmp     BackGround       ;Go do backgroun stuff.
ParentPrCs:  mov     ChildPID, bx   ;Save child process ID.

        print
        byte    "I am timing you while you enter a string. So type"
        byte    cr,lf
        byte    "quickly: ",0

        lesi    InputLine
        gets

        mov     ax, ChildPID      ;Stop the child from running.
        kill

        printf  "While entering '%s' you took %d clock ticks"
        byte    cr,lf,0
        dword   InputLine, BackGndCnt

        prcsquit

Quit:       ExitPgm              ;DOS macro to quit program.
Main       endp

cseg       ends

sseg       segment   para stack 'stack'

; Here is the stack for the background process we start

stk2       byte    256 dup (?)
EndStk2    word    ?

;Here's the stack for the main program/foreground process.

stk        byte    1024 dup (?)
sseg       ends

zzzzzzseg  segment   para public 'zzzzzz'
LastBytes  db      16 dup (?)
zzzzzzseg  ends
end        Main

```

19.5 Synchronization

Many problems occur in cooperative concurrently executing processes due to *synchronization* (or the lack thereof). For example, one process can *produce* data that other processes *consume*. However, it might take much longer for the producer to create than data than it takes for the consumer to use it. Some mechanism must be in place to ensure that the consumer does not attempt to use the data before the producer creates it. Likewise, we need to ensure that the consumer uses the data created by the producer before the producer creates more data.

The *producer-consumer problem* is one of several very famous *synchronization* problems from operating systems theory. In the producer-consumer problem there are one or more processes that produce data and write this data to a shared buffer. Likewise, there are one or more consumers that read data from this buffer. There are two synchronization issues we must deal with – the first is to ensure that the producers do not produce more data than the buffer can hold (conversely, we must prevent the consumers from removing data from an empty buffer); the second is to ensure the integrity of the buffer data structure by allowing access to only one process at a time.

Consider what can happen in a simple producer-consumer problem. Suppose the producer and consumer processes share a single data buffer structure organized as follows:

```
buffer      struct
Count      word      0
InPtr      word      0
OutPtr     word      0
Data       byte      MaxBufSize dup (?)
buffer     ends
```

The Count field specifies the number of data bytes currently in the buffer. InPtr points at the next available location to place data in the buffer. OutPtr is the address of the next byte to remove from the buffer. Data is the actual buffer array. Adding and removing data is very easy. The following code segments *almost* handle this job:

```
; Producer- This procedure adds the value in al to the buffer.
;           Assume that the buffer variable MyBuffer is in the data segment.

Producer   proc      near
           pushf
           sti                    ;Must have interrupts on!
           push     bx

; The following loop waits until there is room in the buffer to insert
; another byte.

WaitForRoom:  cmp      MyBuffer.Count, MaxBufSize
             jae      WaitForRoom

; Okay, insert the byte into the buffer.

             mov      bx, MyBuffer.InPtr
             mov      MyBuffer.Data[bx], al
             inc      MyBuffer.Count    ;We just added a byte to the buffer.
             inc      MyBuffer.InPtr    ;Move on to next item in buffer.

; If we are at the physical end of the buffer, wrap around to the beginning.

             cmp      MyBuffer.InPtr, MaxBufSize
             jb      NoWrap
             mov      MyBuffer.InPtr, 0

NoWrap:     pop      bx
           popf
           ret

Producer   endp

; Consumer- This procedure waits for data (if necessary) and returns the
;           next available byte from the buffer.

Consumer   proc      near
           pushf
           sti                    ;Must have interrupts on!
           push     bx

WaitForData:  cmp      Count, 0        ;Is the buffer empty?
             je      WaitForData      ;If so, wait for data to arrive.

; Okay, fetch an input character

             mov      bx, MyBuffer.OutPtr
             mov      al, MyBuffer.Data[bx]
             dec      MyBuffer.Count
             inc      MyBuffer.OutPtr
             cmp      MyBuffer.OutPtr, MaxBufSize
             jb      NoWrap
             mov      MyBuffer.OutPtr, 0

NoWrap:     pop      bx
           popf
           ret

Consumer   endp
```

The only problem with this code is that it won't always work if there are multiple producer or consumer processes. In fact, it is easy to come up with a version of this code that won't work for a single set of producer and consumer processes (although the code above will work fine, in that special case). The problem is that these procedures access global variables and, therefore, are not reentrant. In particular, the problem lies with the way these two procedures manipulate the buffer control variables. Consider, for a moment, the following statements from the Consumer procedure:

```

                dec         MyBuffer.Count

    « Suppose an interrupt occurs here »

                inc         MyBuffer.OutPtr
                cmp         MyBuffer.OutPtr, MaxBufSize
                jb         NoWrap
                mov         MyBuffer.OutPtr, 0
NoWrap:

```

If an interrupt occurs at the specified point above and control transfers to another consumer process that reenters this code, the second consumer would malfunction. The problem is that the first consumer has fetched data from the buffer but has yet to update the output pointer. The second consumer comes along and removes *the same byte as the first consumer*. The second consumer then properly updates the output pointer to point at the next available location in the circular buffer. When control eventually returns to the first consumer process, it finishes the operation by incrementing the output pointer. *This causes the system to skip over the next byte which no process has read*. The end result is that two consumer processes fetch the same byte and then skip a byte in the buffer.

This problem is easily solved by recognizing the fact that the code that manipulates the buffer data is a critical region. By restricting execution in the critical region to one process at a time, we can solve this problem. In the simple example above, we can easily prevent reentrancy by turning the interrupts off while in the critical region. For the consumer procedure, the code would look like this:

```

; Consumer-   This procedure waits for data (if necessary) and returns the
;            next available byte from the buffer.

Consumer     proc         near
                pushf                    ;Must have interrupts on!
                sti
                push         bx
WaitForData:  cmp         Count, 0         ;Is the buffer empty?
                je         WaitForData    ;If so, wait for data to arrive.

; The following is a critical region, so turn the interrupts off.

                cli

; Okay, fetch an input character

                mov         bx, MyBuffer.OutPtr
                mov         al, MyBuffer.Data[bx]
                dec         MyBuffer.Count
                inc         MyBuffer.OutPtr
                cmp         MyBuffer.OutPtr, MaxBufSize
                jb         NoWrap
                mov         MyBuffer.OutPtr, 0
NoWrap:
                pop         bx
                popf                    ;Restore interrupt flag.
                ret
Consumer     endp

```

Note that we cannot turn the interrupts off during the execution of the whole procedure. Interrupts must be on while this procedure is waiting for data, otherwise the producer process will never be able to put data in the buffer for the consumer.

Simply turning the interrupts off does not always work. Some critical regions may take a considerable amount of time (seconds, minutes, or even hours) and you cannot leave the interrupts off for that amount

of time³. Another problem is that the critical region may call a procedure that turns the interrupts back on and you have no control over this. A good example is a procedure that calls MS-DOS. Since MS-DOS is not reentrant, MS-DOS is, by definition, a critical section; we can only allow one process at a time inside MS-DOS. However, MS-DOS reenables the interrupts, so we cannot simply turn off the interrupts before calling an MS-DOS function and expect this to prevent reentrancy.

Turning off the interrupts doesn't even work for the consumer/producer procedures given earlier. Note that interrupts *must* be on while the consumer is waiting for data to arrive in the buffer (conversely, the producers must have interrupts on while waiting for room in the buffer). It is quite possible for the code to detect the presence of data and just before the execution of the `c1i` instruction, an interrupt transfers control to a second consumer process. While it is not possible for both processes to update the buffer variables concurrently, it is possible for the second consumer process to remove the *only* data value from the input buffer and then switch back to the first consumer that removes a phantom value from the buffer (and causes the Count variable to go negative).

One poorly thought out solution is to use a flag to control access to a critical region. A process, before entering the critical region, tests the flag to see if any other process is currently in the critical region; if not, the process sets the flag to "in use" and then enters the critical region. Upon leaving the critical region, the process sets the flag to "not in use." If a process wants to enter a critical region and the flag's value is "in use", the process must wait until the process currently in the critical section finishes and writes the "not in use" value to the flag.

The only problem with this solution is that it is nothing more than a special case of the producer/consumer problem. The instructions that update the in-use flag form their own critical section that you must protect. As a general solution, the in-use flag idea fails.

19.5.1 Atomic Operations, Test & Set, and Busy-Waiting

The problem with the in-use flag idea is that it takes several instructions to test and set the flag. A typical piece of code that tests such a flag would read its value and determine if the critical section is in use. If not, it would then write the "in-use" value to the flag to let other processes know that it is in the critical section. The problem is that an interrupt could occur after the code tests the flag but before it sets the flag to "in use." Then some other process can come along, test the flag and find that it is not in use, and enter the critical region. The system could interrupt that second process while it is still in the critical region and transfer control back to the first. Since the first process has already determined that the critical region is not in use, it sets the flag to "in use" and enters the critical region. Now we have two processes in the critical region and the system is in violation of the *mutual exclusion requirement* (only one process in a critical region at a time).

The problem with this approach is that testing and setting the in-use flag is not an uninterruptable (*atomic*) operation. If it were, then there would be no problem. Of course, it is easy to make a sequence of instructions non-interruptible by putting a `c1i` instruction before them. Therefore, we can test and set a flag in an atomic operation as follows (assume in-use is zero, not in-use is one):

```

TestLoop:    pushf
             cli                    ;Turn ints off while testing and
             cmp                     Flag, 0          ; setting flag.
             je                      IsInUse         ;Already in use?
             mov                     Flag, 0          ;If not, make it so.
IsInUse:    sti                      ;Allow ints (if in-use already).
             je                      TestLoop        ;Wait until not in use.
             popf

; When we get down here, the flag was "not in-use" and we've just set it
; to "in-us." We now have exclusive access to the critical section.

```

3. In general, you should not leave the interrupts off for more than about 30 milliseconds when using the 1/18th second clock for multitasking. A general rule of thumb is that interrupts should not be off for much more than about 50% of the time quantum.

Another solution is to use a so-called “test and set” instruction – one that both tests a specific condition and sets the flag to a desired value. In our case, we need an instruction that both tests a flag to see if it is not in-use and sets it to in-use at the same time (if the flag was already in-use, it will remain in use afterward). Although the 80x86 does not support a specific test and set instruction, it does provide several others that can achieve the same effect. These instructions include `xchg`, `shl`, `shr`, `sar`, `rcl`, `rcr`, `rol`, `ror`, `btc/btr/bts` (available only on the 80386 and later processors), and `cmpxchg` (available only on the 80486 and later processors). In a limited sense, you can also use the addition and subtraction instructions (`add`, `sub`, `adc`, `sbb`, `inc`, and `dec`) as well.

The exchange instruction provides the most generic form for the test and set operation. If you have a flag (0=in use, 1=not in use) you can test and set this flag without messing with the interrupts using the following code:

```
InUseLoop:    mov     al, 0                ;0=In Use
              xchg   al, Flag
              cmp    al, 0
              je     InUseLoop
```

The `xchg` instruction atomically swaps the value in `al` with the value in the flag variable. Although the `xchg` instruction doesn’t actually test the value, it does place the original flag value in a location (`al`) that is safe from modification by another process. If the flag originally contained zero (in-use), this exchange sequence swaps a zero for the existing zero and the loop repeats. If the flag originally contained a one (not in-use) then this code swaps a zero (in-use) for the one and falls out of the in use loop.

The shift and rotate instructions also act as test and set instructions, assuming you use the proper values for the in-use flag. With in-use equal to zero and not in-use equal to one, the following code demonstrates how to use the `shr` instruction for the test and set operation:

```
InUseLoop:    shr     Flag, 1          ;In-use bit to carry, 0->Flag.
              jnc    InUseLoop        ;Repeat if already in use.
```

This code shifts the in-use bit (bit number zero) into the carry flag and clears the in-use flag. At the same time, it zeros the Flag variable, assuming Flag always contains zero or one. The code for the atomic test and set sequences using the other shift and rotates is very similar and appears in the exercises.

Starting with the 80386, Intel provided a set of instructions explicitly intended for test and set operations: `btc` (bit test and complement), `bts` (bit test and set), and `btr` (bit test and reset). These instructions copy a specific bit from the destination operand into the carry flag and then complement, set, or reset (clear) that bit. The following code demonstrates how to use the `btr` instruction to manipulate our in-use flag:

```
InUseLoop:    btr     Flag, 0          ;In-use flag is in bit zero.
              jnc    InUseLoop
```

The `btr` instruction is a little more flexible than the `shr` instruction because you don’t have to guarantee that all the other bits in the Flag variable are zero; it tests and clears bit zero without affect any other bits in the Flag variable.

The 80486 (and later) `cmpxchg` instruction provides a very generic synchronization primitive. A “compare and swap” instruction turns out to be the only atomic instruction you need to implement almost *any* synchronization primitive. However, its generic structure means that it is a little too complex for simple test and set operations. You will get an opportunity to design a test and set sequence using `cmpxchg` in the exercises. For more details on `cmpxchg`, see “The `CMPXCHG`, and `CMPXCHG8B` Instructions” on page 263.

Returning to the producer/consumer problem, we can easily solve the critical region problem that exists in these routines using the test and set instruction sequence presented above. The following code does this for the Producer procedure, you would modify the Consumer procedure in a similar fashion.

```
; Producer-    This procedure adds the value in al to the buffer.
;              Assume that the buffer variable MyBuffer is in the data segment.

Producer      proc      near
```

```

        pushf
        sti                                ;Must have interrupts on!

; Okay, we are about to enter a critical region (this whole procedure),
; so test the in-use flag to see if this critical region is already in use.

InUseLoop:  shr     Flag, 1
            jnc     InUseLoop

            push   bx

; The following loop waits until there is room in the buffer to insert
; another byte.

WaitForRoom:  cmp     MyBuffer.Count, MaxBufSize
            jae     WaitForRoom

; Okay, insert the byte into the buffer.

            mov     bx, MyBuffer.InPtr
            mov     MyBuffer.Data[bx], al
            inc     MyBuffer.Count    ;We just added a byte to the buffer.
            inc     MyBuffer.InPtr    ;Move on to next item in buffer.

; If we are at the physical end of the buffer, wrap around to the beginning.

            cmp     MyBuffer.InPtr, MaxBufSize
            jnb     NoWrap
            mov     MyBuffer.InPtr, 0

NoWrap:
            mov     Flag, 1           ;Set flag to not in use.
            pop     bx
            popf
            ret

Producer     endp

```

One minor problem with the test and set approach to protecting a critical region is that it uses a *busy-waiting* loop. While the critical region is not available, the process spins in a loop waiting for its turn at the critical region. If the process that is currently in the critical region remains there for a considerable length of time (say, seconds, minutes, or hours), the process(es) waiting to enter the critical region continue to waste CPU time waiting for the flag. This, in turn, wastes CPU time that could be put to better use getting the process in the critical region through it so another process can enter.

Another problem that might exist is that it is possible for one process to enter the critical region, locking other processes out, leave the critical region, do some processing, and then reenter the critical region all during the same time slice. If it turns out that the process is always in the critical region when the timer interrupt occurs, none of the other processes waiting to enter the critical region will ever do so. This is a problem known as *starvation* – processes waiting to enter the critical region never do so because some other process always beats them into it.

One solution to these two problems is to use a synchronization object known as a semaphore. Semaphores provide an efficient and general purpose mechanism for protecting critical regions. To find out about semaphores, keep reading...

19.5.2 Semaphores

A semaphore is an object with two basic methods: wait and signal (or release). To use a semaphore, you create a semaphore variable (an instance) for a particular critical region or other *resource* you want to protect. When a process wants to use a given resource, it *waits* on the semaphore. If no other process is currently using the resource, then the wait call sets the semaphore to in-use and immediately returns to the process. At that time, the process has exclusive access to the resource. If some other process is already using the resource (e.g., is in the critical region), then the semaphore *blocks* the current process by moving it off the run queue and onto the semaphore queue. When the process that currently holds the

resource *releases* it, the release operation removes the first waiting process from the semaphore queue and places it back in the run queue. At the next available time slice, that new process returns from its wait call and can enter its critical region.

Semaphores solve the two important problems with the busy-waiting loop described in the previous section. First, when a process waits and the semaphore blocks the process, that process is no longer on the run queue, so it consumes no more CPU time until the point that a release operation places it back onto the run queue. So unlike busy-waiting, the semaphore mechanism does not waste (as much) CPU time on processes that are waiting for some resource.

Semaphores can also solve the starvation problem. The wait operation, when blocking a process, can place it at the end of a FIFO semaphore queue. The release operation can fetch a new process from the front of the FIFO queue to place back on to the run queue. This policy ensures that each process entering the semaphore queue gets equal priority access to the resource⁴.

Implementing semaphores is an easy task. A semaphore generally consists of an integer variable and a queue. The system initializes the integer variable with the number of processes that may share the resource at one time (this value is usually one for critical regions and other resources requiring exclusive access). The wait operation decrements this variable. If the result is greater than or equal to zero, the wait function simply returns to the caller; if the result is less than zero, the wait function saves the machine state, moves the process' pcb from the run queue to the semaphore's queue, and then switches the CPU to a different process (i.e., a `yield` call).

The release function is almost the converse. It increments the integer value. If the result is not one, the release function moves a pcb from the front of the semaphore queue to the run queue. If the integer value becomes one, there are no more processes on the semaphore queue, so the release function simply returns to the caller. Note that the release function does *not* activate the process it removes from the semaphore process queue. It simply places that process in the run queue. Control always returns to the process that made the release call (unless, of course, a timer interrupt occurs while executing the release function).

Of course, any time you manipulate the system's run queue you are in a critical region. Therefore, we seem to have a minor problem here – the whole purpose of a semaphore is to protect a critical region, yet the semaphore itself has a critical region we need to protect. This seems to involve circular reasoning. However, this problem is easily solved. Remember, the main reason we do not turn off interrupts to protect a critical region is because that critical region may take a long time to execute or it may call other routines that turn the interrupts back on. The critical section in a semaphore is very short and does not call any other routines. Therefore, briefly turning off the interrupts while in the semaphore's critical region is perfectly reasonable.

If you are not allowed to turn off interrupts, you can always use a test and set instruction in a loop to protect a critical region. Although this introduces a busy-waiting loop, it turns out that you will never wait more than two time slices before exiting the busy-waiting loop, so you do not waste much CPU time waiting to enter the semaphore's critical region.

Although semaphores solve the two major problems with the busy waiting loop, it is very easy to get into trouble when using semaphores. For example, if a process waits on a semaphore and the semaphore grants exclusive access to the associated resource, then that process never releases the semaphore, any processes waiting on that semaphore will be suspended indefinitely. Likewise, any process that waits on the same semaphore twice without a release in-between will suspend itself, and any other processes that wait on that semaphore, indefinitely. Any process that does not release a resource it no longer needs violates the concept of a semaphore and is a logic error in the program. There are also some problems that may develop if a process waits on multiple semaphores before releasing any. We will return to that problem in the section on *deadlocks* (see “Deadlock” on page 1146).

4. This FIFO policy is but one example of a release policy. You could have some other policy based on a priority scheme. However, the FIFO policy does not promote starvation.

Although we could write our own semaphore package (and there is good reason to), the Standard Library process package provides its own wait and release calls along with a definition for a semaphore variable. The next section describes those calls.

19.5.3 The UCR Standard Library Semaphore Support

The UCR Standard Library process package provides two functions to manipulate semaphore variables: `WaitSemaph` and `RlsSemaph`. These functions wait and signal a semaphore, respectively. These routines mesh with the process management facilities, making it easy to implement synchronization using semaphores in your programs.

The process package provides the following definition for a semaphore data type:

```
semaphore      struct
SemaCnt        word          1
smaphrLst      dword         ?
endsmaphrLst   dword         ?
semaphore      ends
```

The `SemaCnt` field determines how many more processes can share a resource (if positive), or how many processes are currently waiting for the resource (if negative). By default, this field is initialized to the value one. This allows one process at a time to use the resource protected by the semaphore. Each time a process waits on a semaphore, it decrements this field. If the decremented result is positive or zero, the wait operation immediately returns. If the decremented result is negative, then the wait operation moves the current process' `pcb` from the run queue to the semaphore queue defined by the `smaphrLst` and `endsmaphrLst` fields in the structure above.

Most of the time you will use the default value of one for the `SemaCnt` field. There are some occasions, though, when you might want to allow more than one process access to some resource. For example, suppose you've developed a multiplayer game that communicates between different machines using the serial communications port or a network adapter card. You might have an area in the game which has room for only two players at a time. For example, players could be racing to a particular "transporter" room in an alien space ship, but there is room for only two players in the transporter room at a time. By initializing the semaphore variable to two, rather than one, the wait operation would allow two players to continue at one time rather than just one. When the third player attempts to enter the transporter room, the `WaitSemaph` function would block the player from entering the room until one of the other players left (perhaps by "transporting out" of the room).

To use the `WaitSemaph` or `RlsSemaph` function is very easy; just load the `es:di` register pair with the address of desired semaphore variable and issue the appropriate function call. `RlsSemaph` always returns immediately (assuming a timer interrupt doesn't occur while in `RlsSemaph`), the `WaitSemaph` call returns when the semaphore will allow access to the resource it protects. Examples of these two calls appear in the next section.

Like the Standard Library coroutine and process packages, the semaphore package only preserves the 16 bit register set of the 80x86 CPU. If you want to use the 32 bit register set of the 80386 and later processors, you will need to modify the source code for the `WaitSemaph` and `RlsSemaph` functions. The code you need to change is almost identical to the code in the coroutine and process packages, so this is nearly a trivial change. Do keep in mind, though, that you will need to change this code if you use any 32 bit facilities of the 80386 and later processors.

19.5.4 Using Semaphores to Protect Critical Regions

You can use semaphores to provide mutually exclusive access to any resource. For example, if several processes want to use the printer, you can create a semaphore that allows access to the printer by only one process at a time (a good example of a process that will be in the "critical region" for several minutes

at a time). However the most common task for a semaphore is to protect a critical region from reentry. Three common examples of code you need to protect from reentry include DOS calls, BIOS calls, and various Standard Library calls. Semaphores are ideal for controlling access to these functions.

To protect DOS from reentry by several different processes, you need only create a DOSsmaph variable and issue appropriate WaitSemaph and RlsSemaph calls around the call to DOS. The following sample code demonstrates how to do this.

```

; MULTIDOS.ASM
;
; This program demonstrates how to use semaphores to protect DOS calls.

        .xlist
        include    stdlib.a
        includelib stdlib.lib
        .list

dseg          segment    para public 'data'

DOSsmaph      semaphore  {}

; Macros to wait and release the DOS semaphore:

DOSWait       macro
              push      es
              push      di
              lesi      DOSsmaph
              WaitSemaph
              pop       di
              pop       es
              endm

DOSRls        macro
              push      es
              push      di
              lesi      DOSsmaph
              RlsSemaph
              pop       di
              pop       es
              endm

; PCB for our background process:

BkgndPCB      pcb        {0,offset EndStk2, seg EndStk2}

; Data the foreground and background processes print:

StrPtrs1      dword     str1_a, str1_b, str1_c, str1_d, str1_e, str1_f
              dword     str1_g, str1_h, str1_i, str1_j, str1_k, str1_l
              dword     0

str1_a        byte     "Foreground: string 'a'",cr,lf,0
str1_b        byte     "Foreground: string 'b'",cr,lf,0
str1_c        byte     "Foreground: string 'c'",cr,lf,0
str1_d        byte     "Foreground: string 'd'",cr,lf,0
str1_e        byte     "Foreground: string 'e'",cr,lf,0
str1_f        byte     "Foreground: string 'f'",cr,lf,0
str1_g        byte     "Foreground: string 'g'",cr,lf,0
str1_h        byte     "Foreground: string 'h'",cr,lf,0
str1_i        byte     "Foreground: string 'i'",cr,lf,0
str1_j        byte     "Foreground: string 'j'",cr,lf,0
str1_k        byte     "Foreground: string 'k'",cr,lf,0
str1_l        byte     "Foreground: string 'l'",cr,lf,0

StrPtrs2      dword     str2_a, str2_b, str2_c, str2_d, str2_e, str2_f
              dword     str2_g, str2_h, str2_i
              dword     0

str2_a        byte     "Background: string 'a'",cr,lf,0
str2_b        byte     "Background: string 'b'",cr,lf,0

```

```

str2_c      byte    "Background: string 'c'",cr,lf,0
str2_d      byte    "Background: string 'd'",cr,lf,0
str2_e      byte    "Background: string 'e'",cr,lf,0
str2_f      byte    "Background: string 'f'",cr,lf,0
str2_g      byte    "Background: string 'g'",cr,lf,0
str2_h      byte    "Background: string 'h'",cr,lf,0
str2_i      byte    "Background: string 'i'",cr,lf,0

dseg        ends

cseg        segment para public 'code'
            assume  cs:cseg, ds:dseg

; A replacement critical error handler. This routine calls prcsquit
; if the user decides to abort the program.

CritErrMsg  byte    cr,lf
            byte    "DOS Critical Error!",cr,lf
            byte    "A)bort, R)etry, I)gnore, F)ail? $"

MyInt24     proc    far
            push    dx
            push    ds
            push    ax

            push    cs
            pop     ds
Int24Lp:    lea     dx, CritErrMsg
            mov     ah, 9           ;DOS print string call.
            int     21h

            mov     ah, 1           ;DOS read character call.
            int     21h
            and     al, 5Fh        ;Convert l.c. -> u.c.

            cmp     al, 'I'        ;Ignore?
            jne     NotIgnore
            pop     ax
            mov     al, 0
            jmp     Quit24

NotIgnore:  cmp     al, 'r'        ;Retry?
            jne     NotRetry
            pop     ax
            mov     al, 1
            jmp     Quit24

NotRetry:  cmp     al, 'A'        ;Abort?
            jne     NotAbort
            prcsquit              ;If quitting, fix INT 8.
            pop     ax
            mov     al, 2
            jmp     Quit24

NotAbort:  cmp     al, 'F'
            jne     BadChar
            pop     ax
            mov     al, 3

Quit24:    pop     ds
            pop     dx
            iret

BadChar:   mov     ah, 2
            mov     dl, 7           ;Bell character
            jmp     Int24Lp

MyInt24     endp

; We will simply disable INT 23h (the break exception).

MyInt23     proc    far
            iret
MyInt23     endp

```

```

; This background process calls DOS to print several strings to the
; screen. In the meantime, the foreground process is also printing
; strings to the screen. To prevent reentry, or at least a jumble of
; characters on the screen, this code uses semaphores to protect the
; DOS calls. Therefore, each process will print one complete line
; then release the semaphore. If the other process is waiting it will
; print its line.

```

```

BackGround    proc
              mov     ax, dseg
              mov     ds, ax
              lea    bx, StrPtrs2    ;Array of str ptrs.
PrintLoop:    cmp     word ptr [bx+2], 0 ;At end of pointers?
              je     BkGndDone
              les     di, [bx]       ;Get string to print.
              DOSWait
              puts    ;Calls DOS to print string.
              DOSRls
              add     bx, 4           ;Point at next str ptr.
              jmp    PrintLoop

BkGndDone:    die
BackGround    endp

```

```

Main          proc
              mov     ax, dseg
              mov     ds, ax
              mov     es, ax
              meminit

```

```

; Initialize the INT 23h and INT 24h exception handler vectors.

```

```

              mov     ax, 0
              mov     es, ax
              mov     word ptr es:[24h*4], offset MyInt24
              mov     es:[24h*4 + 2], cs
              mov     word ptr es:[23h*4], offset MyInt23
              mov     es:[23h*4 + 2], cs

              prcsinit                ;Start multitasking system.

              lesi    BkgndPCB        ;Fire up a new process
              fork
              test   ax, ax            ;Parent's return?
              je     ParentPrs
              jmp    BackGround       ;Go do background stuff.

```

```

; The parent process will print a bunch of strings at the same time
; the background process is doing this. We'll use the DOS semaphore
; to protect the call to DOS that PUTS makes.

```

```

ParentPrs:    DOSWait                ;Force the other process
              mov     cx, 0           ; to wind up waiting in
DlyLp0:       loop   DlyLp0           ; the semaphore queue by
DlyLp1:       loop   DlyLp1           ; delay for at least one
DlyLp2:       loop   DlyLp2           ; clock tick.
              DOSRls

              lea    bx, StrPtrs1    ;Array of str ptrs.
PrintLoop:    cmp     word ptr [bx+2],0 ;At end of pointers?
              je     ForeGndDone
              les     di, [bx]       ;Get string to print.
              DOSWait
              puts    ;Calls DOS to print string.
              DOSRls
              add     bx, 4           ;Point at next str ptr.
              jmp    PrintLoop

ForeGndDone:  prcsquit

```

```

Quit:      ExitPgm      ;DOS macro to quit program.
Main      endp

cseg      ends

sseg      segment      para stack 'stack'

; Here is the stack for the background process we start

stk2      byte      1024 dup (?)
EndStk2   word      ?

;Here's the stack for the main program/foreground process.

stk       byte      1024 dup (?)
sseg      ends

zzzzzzseg segment      para public 'zzzzzz'
LastBytes db      16 dup (?)
zzzzzzseg ends
end      Main

```

This program doesn't directly call DOS, but it calls the Standard Library `puts` routine that does. In general, you could use a single semaphore to protect all BIOS, DOS, and Standard Library calls. However, this is not particularly efficient. For example, the Standard Library pattern matching routines make no DOS calls; therefore, waiting on the DOS semaphore to do a pattern match while some other process is making a DOS call unnecessarily delays the pattern match. There is nothing wrong with having one process do a pattern match while another is making a DOS call. Unfortunately, some Standard Library routines *do* make DOS calls (`puts` is a good example), so you must use the DOS semaphore around such calls.

In theory, we could use separate semaphores to protect DOS, different BIOS calls, and different Standard Library calls. However, keeping track of all those semaphores within a program is a big task. Furthermore, ensuring that a call to DOS does not also invoke an unprotected BIOS routine is a difficult task. So most programmers use a single semaphore to protect all Standard Library, DOS, and BIOS calls.

19.5.5 Using Semaphores for Barrier Synchronization

Although the primary use of a semaphores is to provide exclusive access to some resource, there are other synchronization uses for semaphores as well. In this section we'll look at the use of the Standard Library's semaphores objects to create a *barrier*.

A barrier is a point in a program where a process stops and waits for other processes to synchronize (reach their respective barriers). In many respects, a barrier is the dual to a semaphore. A semaphore prevents more than n processes from gaining access to some resource. A barrier does not grant access until at least n processes are requesting access.

Given the different nature of these two synchronization methods, you might think that it would be difficult to use the `WaitSemaph` and `RelSemaph` routines to implement barriers. However, it turns out to be quite simple. Suppose we were to initialize the semaphore's `SemaCnt` field to zero rather than one. When the first process waits on this semaphore, the system will immediately block that process. Likewise, each additional process that waits on this semaphore will block and wait on the semaphore queue. This would normally be a disaster since there is no active process that will signal the semaphore so it will activate the blocked processes. However, if we modify the wait call so that it checks the `SemaCnt` field before actually doing the wait, the n^{th} process can skip the wait call and reactivate the other processes. Consider the following macro:

```

barrier      macro      Wait4Cnt
              local     AllHere, AllDone
              cmp       es:[di].semaphore.SemaCnt, -(Wait4Cnt-1)
              jle      AllHere
              WaitSemaph
              cmp       es:[di].semaphore.SemaCnt, 0
              je       AllDone
AllHere:     RlsSemaph
AllDone:
              endm

```

This macro expects a single parameter that should be the number of processes (including the current process) that need to be at a barrier before any of the processes can proceed. The SemaCnt field is a negative number whose absolute value determines how many processes are currently waiting on the semaphore. If a barrier requires four processes, no process can proceed until the fourth process hits the barrier; at that time the SemaCnt field will contain minus three. The macro above computes what the value of SemaCnt should be if all processes are at the barrier. If SemaCnt matches this value, it signals the semaphore that begins a chain of operations with each blocked process releasing the next. When SemaCnt hits zero, the last blocked process does not release the semaphore since there are no other processes waiting on the queue.

It is very important to remember to initialize the SemaCnt field to zero before using semaphores for barrier synchronization in this manner. If you do not initialize SemaCnt to zero, the WaitSemaph call will probably not block any of the processes.

The following sample program provides a simple example of barrier synchronization using the Standard Library's semaphore package:

```

; BARRIER.ASM
;
; This sample program demonstrates how to use the Standard Library's
; semaphore objects to synchronize several processes at a barrier.
; This program is similar to the MULTIDOS.ASM program insofar as the
; background processes all print a set of strings. However, rather than
; using an inelegant delay loop to synchronize the foreground and background
; processes, this code uses barrier synchronization to achieve this.

                .xlist
                include  stdlib.a
                includelib stdlib.lib
                .list

dseg            segment    para public 'data'

BarrierSemaph  semaphore  {0}                ;Must init SemaCnt to zero.
DOSsmaph       semaphore  {}

; Macros to wait and release the DOS semaphore:

DOSWait        macro
              push      es
              push      di
              lesi      DOSsmaph
              WaitSemaph
              pop       di
              pop       es
              endm

DOSRls         macro
              push      es
              push      di
              lesi      DOSsmaph
              RlsSemaph
              pop       di
              pop       es
              endm

; Macro to synchronize on a barrier:

```

```

Barrier      macro      Wait4Cnt
              local     AllHere, AllDone
              cmp       es:[di].semaphore.SemaCnt, -(Wait4Cnt-1)
              jle      AllHere
              WaitSemaph
              cmp       es:[di].semaphore.SemaCnt, 0
              jge      AllDone
AllHere:
AllDone:
              RlsSemaph
              endm

; PCBs for our background processes:

BkgndPCB2    pcb        {0,offset EndStk2, seg EndStk2}
BkgndPCB3    pcb        {0,offset EndStk3, seg EndStk3}

; Data the foreground and background processes print:

StrPtrs1     dword     str1_a, str1_b, str1_c, str1_d, str1_e, str1_f
              dword     str1_g, str1_h, str1_i, str1_j, str1_k, str1_l
              dword     0

str1_a       byte     "Foreground: string 'a'",cr,lf,0
str1_b       byte     "Foreground: string 'b'",cr,lf,0
str1_c       byte     "Foreground: string 'c'",cr,lf,0
str1_d       byte     "Foreground: string 'd'",cr,lf,0
str1_e       byte     "Foreground: string 'e'",cr,lf,0
str1_f       byte     "Foreground: string 'f'",cr,lf,0
str1_g       byte     "Foreground: string 'g'",cr,lf,0
str1_h       byte     "Foreground: string 'h'",cr,lf,0
str1_i       byte     "Foreground: string 'i'",cr,lf,0
str1_j       byte     "Foreground: string 'j'",cr,lf,0
str1_k       byte     "Foreground: string 'k'",cr,lf,0
str1_l       byte     "Foreground: string 'l'",cr,lf,0

StrPtrs2     dword     str2_a, str2_b, str2_c, str2_d, str2_e, str2_f
              dword     str2_g, str2_h, str2_i
              dword     0

str2_a       byte     "Background 1: string 'a'",cr,lf,0
str2_b       byte     "Background 1: string 'b'",cr,lf,0
str2_c       byte     "Background 1: string 'c'",cr,lf,0
str2_d       byte     "Background 1: string 'd'",cr,lf,0
str2_e       byte     "Background 1: string 'e'",cr,lf,0
str2_f       byte     "Background 1: string 'f'",cr,lf,0
str2_g       byte     "Background 1: string 'g'",cr,lf,0
str2_h       byte     "Background 1: string 'h'",cr,lf,0
str2_i       byte     "Background 1: string 'i'",cr,lf,0

StrPtrs3     dword     str3_a, str3_b, str3_c, str3_d, str3_e, str3_f
              dword     str3_g, str3_h, str3_i
              dword     0

str3_a       byte     "Background 2: string 'j'",cr,lf,0
str3_b       byte     "Background 2: string 'k'",cr,lf,0
str3_c       byte     "Background 2: string 'l'",cr,lf,0
str3_d       byte     "Background 2: string 'm'",cr,lf,0
str3_e       byte     "Background 2: string 'n'",cr,lf,0
str3_f       byte     "Background 2: string 'o'",cr,lf,0
str3_g       byte     "Background 2: string 'p'",cr,lf,0
str3_h       byte     "Background 2: string 'q'",cr,lf,0
str3_i       byte     "Background 2: string 'r'",cr,lf,0

dseg        ends

cseg        segment    para public 'code'
              assume   cs:cseg, ds:dseg

```

```

; A replacement critical error handler. This routine calls prcsquit
; if the user decides to abort the program.

```

```

CritErrMsg  byte    cr,lf
            byte    "DOS Critical Error!",cr,lf
            byte    "A)abort, R)etry, I)gnore, F)ail? $"

MyInt24     proc    far
            push   dx
            push   ds
            push   ax

            push   cs
            pop    ds
Int24Lp:    lea    dx, CritErrMsg
            mov    ah, 9           ;DOS print string call.
            int    21h

            mov    ah, 1           ;DOS read character call.
            int    21h
            and    al, 5Fh        ;Convert l.c. -> u.c.

            cmp    al, 'I'        ;Ignore?
            jne    NotIgnore
            pop    ax
            mov    al, 0
            jmp    Quit24

NotIgnore:  cmp    al, 'r'        ;Retry?
            jne    NotRetry
            pop    ax
            mov    al, 1
            jmp    Quit24

NotRetry:  cmp    al, 'A'        ;Abort?
            jne    NotAbort
            prcsquit             ;If quitting, fix INT 8.
            pop    ax
            mov    al, 2
            jmp    Quit24

NotAbort:  cmp    al, 'F'
            jne    BadChar
            pop    ax
            mov    al, 3

Quit24:    pop    ds
            pop    dx
            iret

BadChar:   mov    ah, 2
            mov    dl, 7           ;Bell character
            jmp    Int24Lp

MyInt24     endp

```

; We will simply disable INT 23h (the break exception).

```

MyInt23     proc    far
            iret
MyInt23     endp

```

; This background processes call DOS to print several strings to the
; screen. In the meantime, the foreground process is also printing
; strings to the screen. To prevent reentry, or at least a jumble of
; characters on the screen, this code uses semaphores to protect the
; DOS calls. Therefore, each process will print one complete line
; then release the semaphore. If the other process is waiting it will
; print its line.

```

BackGround1 proc
            mov    ax, dseg
            mov    ds, ax

```

```

; Wait for everyone else to get ready:

        lesi     BarrierSemaph
        barrier  3

; Okay, start printing the strings:

PrintLoop:  lea     bx, StrPtrs2      ;Array of str ptrs.
            cmp     word ptr [bx+2],0 ;At end of pointers?
            je      BkGndDone
            les     di, [bx]        ;Get string to print.
            DOSWait
            puts                    ;Calls DOS to print string.
            DOSRls
            add     bx, 4            ;Point at next str ptr.
            jmp     PrintLoop

BkGndDone: die
BackGround1 endp

BackGround2 proc
            mov     ax, dseg
            mov     ds, ax

            lesi     BarrierSemaph
            barrier  3

PrintLoop:  lea     bx, StrPtrs3      ;Array of str ptrs.
            cmp     word ptr [bx+2],0 ;At end of pointers?
            je      BkGndDone
            les     di, [bx]        ;Get string to print.
            DOSWait
            puts                    ;Calls DOS to print string.
            DOSRls
            add     bx, 4            ;Point at next str ptr.
            jmp     PrintLoop

BkGndDone: die
BackGround2 endp

Main       proc
            mov     ax, dseg
            mov     ds, ax
            mov     es, ax
            meminit

; Initialize the INT 23h and INT 24h exception handler vectors.

            mov     ax, 0
            mov     es, ax
            mov     word ptr es:[24h*4], offset MyInt24
            mov     es:[24h*4 + 2], cs
            mov     word ptr es:[23h*4], offset MyInt23
            mov     es:[23h*4 + 2], cs

            prcsinit                    ;Start multitasking system.

; Start the first background process:

            lesi     BkgndPCB2        ;Fire up a new process
            fork
            test     ax, ax            ;Parent's return?
            je      StartBG2
            jmp     BackGround1       ;Go do backgroun stuff.

; Start the second background process:

StartBG2:  lesi     BkgndPCB3        ;Fire up a new process
            fork

```

```

        test     ax, ax           ;Parent's return?
        je      ParentPrCs
        jmp     BackGround2      ;Go do background stuff.

; The parent process will print a bunch of strings at the same time
; the background process is doing this. We'll use the DOS semaphore
; to protect the call to DOS that PUTS makes.

ParentPrCs:  lesi     BarrierSemaph
             barrier  3

PrintLoop:   lea     bx, StrPtrs1 ;Array of str ptrs.
             cmp     word ptr [bx+2],0 ;At end of pointers?
             je      ForeGndDone
             les     di, [bx]      ;Get string to print.
             DOSWait
             puts                    ;Calls DOS to print string.
             DOSRls
             add     bx, 4          ;Point at next str ptr.
             jmp     PrintLoop

ForeGndDone: prcsquit

Quit:       ExitPgm              ;DOS macro to quit program.
Main       endp

cseg       ends

sseg       segment   para stack 'stack'

; Here are the stacks for the background processes we start

stk2       byte     1024 dup (?)
EndStk2    word     ?

stk3       byte     1024 dup (?)
EndStk3    word     ?

;Here's the stack for the main program/foreground process.

stk        byte     1024 dup (?)
sseg       ends

zzzzzzseg  segment   para public 'zzzzzz'
LastBytes  db       16 dup (?)
zzzzzzseg  ends
end        Main

```

Sample Output:

```

Background 1: string 'a'
Background 1: string 'b'
Background 1: string 'c'
Background 1: string 'd'
Background 1: string 'e'
Background 1: string 'f'
Foreground: string 'a'
Background 1: string 'g'
Background 2: string 'j'
Foreground: string 'b'
Background 1: string 'h'
Background 2: string 'k'
Foreground: string 'c'
Background 1: string 'i'
Background 2: string 'l'
Foreground: string 'd'
Background 2: string 'm'
Foreground: string 'e'
Background 2: string 'n'
Foreground: string 'f'
Background 2: string 'o'
Foreground: string 'g'

```

```

Background 2: string 'p'
Foreground: string 'h'
Background 2: string 'q'
Foreground: string 'i'
Background 2: string 'r'
Foreground: string 'j'
Foreground: string 'k'
Foreground: string 'l'

```

Note how background process number one ran for one clock period before the other processes waited on the DOS semaphore. After this initial burst, the processes all took turns calling DOS.

19.6 Deadlock

Although semaphores can solve any synchronization problems, don't get the impression that semaphores don't introduce problems of their own. As you've already seen, the improper use of semaphores can result in the indefinite suspension of processes waiting on the semaphore queue. However, even if you correctly wait and signal individual semaphores, it is quite possible for correct operations on *combinations* of semaphores to produce this same effect. Indefinite suspension of a process because of semaphore problems is a serious issue. This degenerate situation is known as *deadlock* or *deadly embrace*.

Deadlock occurs when one process holds one resource and is waiting for another while a second process is holding that other resource and waiting for the first. To see how deadlock can occur, consider the following code:

```

; Process one:

        lesi        Semaph1
        WaitSemaph

        « Assume interrupt occurs here »

        lesi        Semaph2
        WaitSemaph
        .
        .
        .

; Process two:

        lesi        Semaph2
        WaitSemaph
        lesi        Semaph1
        WaitSemaph
        .
        .
        .

```

Process one grabs the semaphore associated with Semaph1. Then a timer interrupt comes along which causes a *context switch* to process two. Process two grabs the semaphore associated with Semaph2 and then tries to get Semaph1. However, process one is already holding Semaph1, so process two blocks and waits for process one to release this semaphore. This returns control (eventually) to process one. Process one then tries to grab Semaph2. Unfortunately, process two is already holding Semaph2, so process one blocks waiting for Semaph2. Now both processes are blocked waiting for the other. Since neither process can run, neither process can release the semaphore the other needs. Both processes are deadlocked.

One easy way to prevent deadlock from occurring is to never allow a process to hold more than one semaphore at a time. Unfortunately, this is not a practical solution; many processes may need to have exclusive access to several resources at one time. However, we can devise another solution by observing the pattern that resulted in deadlock in the previous example. Deadlock came about because the two processes grabbed different semaphores and then tried to grab the semaphore that the other was holding. In

other words, they grabbed the two semaphores in a different order (process one grabbed Semaph1 first and Semaph2 second, process two grabbed Semaph2 first and Semaph1 second). It turns out that two process will never deadlock if they wait on common semaphores *in the same order*. We could modify the previous example to eliminate the possibility of deadlock thusly:

```

; Process one:

    lesi    Semaph1
    WaitSemaph
    lesi    Semaph2
    WaitSemaph
    .
    .
    .

; Process two:

    lesi    Semaph1
    WaitSemaph
    lesi    Semaph2
    WaitSemaph
    .
    .
    .

```

Now it doesn't matter where the interrupt occurs above, deadlock cannot occur. If the interrupt occurs between the two WaitSemaph calls in process one (as before), when process two attempts to wait on Semaph1, it will block and process one will continue with Semaph2 available.

An easy way to keep out of trouble with deadlock is to number *all* your semaphore variables and make sure that all processes acquire (wait on) semaphores from the smallest numbered semaphore to the highest. This ensures that all processes acquire the semaphores in the same order, and that ensures that deadlock cannot occur.

Note that this policy of acquiring semaphores only applies to semaphores that a process holds concurrently. If a process needs semaphore six for a while, and then it needs semaphore two after it has released semaphore six, there is no problem acquiring semaphore two after releasing semaphore six. However, if at any point the process needs to hold *both* semaphores, it must acquire semaphore two first.

Processes may release the semaphores in any order. The order that a process releases semaphores does not affect whether deadlock can occur. Of course, processes should always release a semaphore as soon as the process is done with the resource guarded by that semaphore; there may be other processes waiting on that semaphore.

While the above scheme works and is easy to implement, it is by no means the only way to handle deadlock, nor is it always the most efficient. However, it is simple to implement and it always works. For more information on deadlocks, see a good operating systems text.

19.7 Summary

Despite the fact that DOS is not reentrant and doesn't directly support multitasking, that doesn't mean your applications can't multitask; it's just difficult to get different applications to run independently of one another under DOS.

Although DOS doesn't switch among different programs in memory, DOS certainly allows you to load multiple programs into memory at one time. The only catch is that only one such program actually executes. DOS provides several calls to load and execute ".EXE" and ".COM" files from the disk. These processes effectively behave like subroutine calls, with control returning to the program invoking such a program only after that "child" program terminates. For more details, see

- "DOS Processes" on page 1065
- "Child Processes in DOS" on page 1065

- “Load and Execute” on page 1066
- “Load Program” on page 1068
- “Loading Overlays” on page 1069
- “Terminating a Process” on page 1069
- “Obtaining the Child Process Return Code” on page 1070

Certain errors can occur during the execution of a DOS process that transfer control to exception handlers. Besides the 80x86 exceptions, DOS’ *break handler* and *critical error handler* are the primary examples. Any program that patches the interrupt vectors should provide its own exception handlers for these conditions so it can restore interrupts on a ctrl-C or I/O error exception. Furthermore, well-written program always provide replacement exception handlers for these two conditions that provide better support than the default DOS handlers. For more information on DOS exceptions, see

- “Exception Handling in DOS: The Break Handler” on page 1070
- “Exception Handling in DOS: The Critical Error Handler” on page 1071
- “Exception Handling in DOS: Traps” on page 1075

When a parent process invokes a child process with the LOAD or LOADEXEC calls, the child process inherits all open files from the parent process. In particular, the child process inherits the *standard input*, *standard output*, *standard error*, *auxiliary I/O*, and *printer* devices. The parent process can easily redirect I/O to/from these devices before passing control to a child process. This, in effect, *redirects* the I/O during the execution of the child process. For more details, see

- “Redirection of I/O for Child Processes” on page 1075

When two DOS programs want to communicate with each other, they typically read and write data to a file. However, creating, opening, reading, and writing files is a lot of work, especially just to share a few variable values. A better alternative is to use *shared memory*. Unfortunately, DOS does not provide support to allow two programs to share a common block of memory. However, it is very easy to write a TSR that manages shared memory for various programs. For details and the complete code to two shared memory managers, see:

- “Shared Memory” on page 1078
- “Static Shared Memory” on page 1078
- “Dynamic Shared Memory” on page 1088

A coroutine call is the basic mechanism for switching control between two processes. A “cocal” operation is the equivalent of a subroutine call and return all rolled into one operation. A cocal transfers control to some other process. When some other process returns control to a coroutine (via cocal), control resumes with the first instruction after the cocal code. The UCR Standard Library provides complete coroutine support so you can easily put coroutines into your assembly language programs. For all the details on coroutines, plus a neat maze generator program that uses coroutines, see

- “Coroutines” on page 1103

Although you can use coroutines to simulate multitasking (“cooperative multitasking”), the major problem with coroutines is that each application must decide when to switch to another process via a cocal. Although this eliminates certain reentrancy and synchronization problems, deciding when and where to make such calls increases the work necessary to write multitasking applications. A better approach is to use *preemptive multitasking* where the timer interrupt performs the context switches. Reentrancy and synchronization problems develop in such a system, but with care those problems are easily overcome. For the details on true preemptive multitasking, and to see how the UCR Standard Library supports multitasking, see

- “Multitasking” on page 1124
- “Lightweight and HeavyWeight Processes” on page 1124
- “The UCR Standard Library Processes Package” on page 1125
- “Problems with Multitasking” on page 1126
- “A Sample Program with Threads” on page 1127

Preemptive multitasking opens up a Pandora's box. Although multitasking makes certain programs easier to implement, the problems of process synchronization and reentrancy rears its ugly head in a multitasking system. Many processes require some sort of synchronized access to global variables. Further, most processes will need to call DOS, BIOS, or some other routine (e.g., the Standard Library) that is not reentrant. Somehow we need to control access to such code so that multiple processes do not adversely affect one another. Synchronization is achievable using several different techniques. In some simple cases we can simply turn off the interrupts, eliminating the reentrancy problems. In other cases we can use test and set or semaphores to protect a *critical region*. For more details on these synchronization operations, see

- “Synchronization” on page 1129
- “Atomic Operations, Test & Set, and Busy-Waiting” on page 1132
- “Semaphores” on page 1134
- “The UCR Standard Library Semaphore Support” on page 1136
- “Using Semaphores to Protect Critical Regions” on page 1136
- “Using Semaphores for Barrier Synchronization” on page 1140

The use of synchronization objects, like semaphores, can introduce new problems into a system. *Deadlock* is a perfect example. Deadlock occurs when one process is holding some resource and wants another and a second process is hold the desired resource and wants the resource held by the first process⁵. You can easily avoid deadlock by controlling the order that the various processes acquire groups of semaphores. For all the details, see

- “Deadlock” on page 1146

5. Or any chain of processes where everyone in the chain is holding something that another process in the chain wants.

