Chapter R3
Process Scheduling

This is a revised version of portions of the Project Manual to accompany A Practical Approach to Operating Systems, by Malcolm Lane and James Mooney. Copyright 1988-2011 by Malcolm Lane and James D. Mooney, all rights reserved.

Revised: Feb. 9, 2011

R3.1 INTRODUCTION

This chapter presents a description of the third required MPX module: Process Scheduling. In this module you will construct a simple round-robin dispatcher for short-term scheduling of MPX processes. Processes will be dispatched in a round-robin fashion; each process will receive a turn in circular order until it is complete. The processes in this module are assumed to be cooperative and will not be subject to timeouts or preemption. Instead, each process will continue executing until it voluntarily gives up control by performing a system call.

This version of the dispatcher will dispatch a sequence of test processes that execute procedures that are statically loaded (*i.e.*, linked directly with your MPX project). The dispatcher will be adapted in later modules to handle dynamic loading. Optional modules may consider further scheduling variations such as true timesharing.

Your principal assignment is to design and implement two critical procedures for MPX: The *dispatcher* and the *system call handler*. The job of the dispatcher is to find the next ready process and perform a context switch to begin executing that process. The job of the system call handler is to respond to system calls by each process; these system calls typically require that the current process leave the running state and a new process be dispatched. In addition, a framework must be provided for testing and demonstrating your dispatcher and call handler. Each of these components is discussed in more detail in Section R3.3.

The construction of an effective system call handler and dispatcher will require an understanding of two important concepts: **interrupt handlers** and **context switching**. Both of these concepts rely in turn on an understanding of the role of the **stack** in the IBM-PC architecture. You should review the discussion of the stack and the 8086 registers in Chapter I3. The following section explains the construction of interrupt handlers, and the method to be used for context switching.

R3.2 KEY CONCEPTS

Interrupt Handlers

In Module R3 you will write two C functions which are structured as *interrupt handlers*. These functions are designed to be invoked in response to hardware interrupts, rather than by ordinary function or procedure calls. The system call handler will be used to handle interrupts generated explicitly by software using the 8086 *int* instruction. The same principles will be used in later modules to handle interrupts generated by external events such as I/O completion. The dispatcher is not actually invoked by an interrupt; however, it will also have the *form* of an interrupt handler, for reasons to be explained later.

There are several ways in which the structure and design of an interrupt handler must differ from that of an ordinary procedure. In general the following issues must be considered:

1. A linkage must be created between the physical interrupt and the handler, so that the handler will be invoked when the interrupt occurs.

2. Since interrupts may occur at unpredictable points in program execution, the *complete* system state must be preserved during interrupt handling. This means that after the interrupt the content of *all* machine registers must be exactly as before; moreover, all memory used by the running process, including its stack, must be undisturbed.

3. Parameters cannot be passed to an interrupt handler in the usual way.

4. Because interrupt handling is a temporary deviation from the normal execution of a running process, the handler may be severely restricted in the resources it may use, and in the subprograms it may call.

5. Interrupt handlers must allow for the appropriate handling of *other* interrupts while the handler is in progress.

6. Interrupt handlers must complete their work in the shortest possible time, leaving more complex followup tasks to other system or application software.

We will consider each of these points in more detail.

Linking to the Interrupt

The interrupt handler is linked to the physical interrupt by storing its starting address in the associated interrupt vector location. In this module your system call interrupt handler will be invoked by the *int 60h* instruction, which is associated with the 32-bit interrupt vector beginning at address 180h (i.e., 60h * 4). Ordinarily, this vector points to a routine that displays an error message, since this interrupt is not used by MS-DOS or Windows. The system support routine `sys_set_vec` will be used during initialization to place the address of *your* handler into this vector.

Saving the State

When an interrupt occurs, a 32-bit segmented address is loaded from the interrupt vector by hardware into the `CS` and `IP` registers. In addition, the interrupt flag (`IF`) in the flags "register" is cleared to temporarily disable further interrupts. To preserve the integrity of the system state, the previous content of all three of these registers must be saved. In the 8086 architecture each of these registers is automatically saved on the stack indicated by the current value of the `SS` and `SP` registers. Note that this mechanism assumes that the `SS-SP` is pointing to a valid stack with at least six bytes of space available.

When an interrupt occurs in the IBM-PC, the *hardware* performs the following specific steps, in the order indicated:

1. The flags register, CS, and IP are pushed (in that order) onto the stack.
2. The interrupt flag (bit 9) in the flags register is cleared.
3. The content of the interrupt vector is loaded into the CS and IP, causing a transfer to the interrupt handler.

It is also necessary for the *software* to save any additional registers that the interrupt handler will use. This is most conveniently accomplished in Turbo C by declaring the handler function using the special keyword `interrupt`. For example, the system call handler that you will develop in this module will have the following prototype:

```
void interrupt sys_call (void);
```

The keyword `interrupt` is a special extension provided by Turbo C to the C language that identifies a function as an interrupt handler. When a function in Turbo C is declared as an interrupt procedure, the compiler will automatically include extra code at the beginning of the procedure to save all registers on the stack that have not been saved by the hardware. The registers saved by the Turbo C software, in order, are:

`AX, BX, CX, DX, ES, DS, SI, DI,` and `BP`

Similarly, code will be included at the end of the procedure to restore these registers when the interrupt handler completes. A final important role of the interrupt declaration is to advise the compiler to terminate the procedure using the *iret* (interrupt return) machine instruction, which restores the flags register as well as the `CS` and `IP`.

Note that `interrupt` is not a part of the ANSI C standard, and may not be found in other C compilers. If you need to write interrupt handlers using a different compiler, you may need to use a different mechanism unique to that compiler to ensure that the proper registers are saved.

Remember, also, that the *source* option for the compiler must be set to *Turbo C keywords*, not *ANSI keywords*, or extensions like `interrupt` will not be recognized.

We can illustrate the form of a Turbo C interrupt handler by presenting the detailed structure of a "null" interrupt handler, *i.e.*, an interrupt handler with *no* statements:

```
void interrupt nullintr()
```

This interrupt handler generates the executable assembler language statements shown in Figure R3-1. This code shows that the 9 extra registers are pushed onto the stack upon entry, and removed upon exit. It also shows a few extra instructions that set `DS` to the data segment used by the interrupt handler, and save `SP` in `BP` so the "call chain" will be preserved even if parameters are pushed onto the stack. Finally, the declaration designates the procedure as a *proc far*, ensuring that all calls to this procedure will use full 32-bit addresses.

```
; Assembler language statements for nullintr() */
     name nullintr
; (segment definition code deleted for this example)
;
_nullintr proc far
     push ax         ; saving the 9 registers
     push bx
     push cx
     push dx
     push es
     push ds
     push si
     push di
     push bp
     mov  bp,DGROUP  ; Set up ds register
     mov  ds,bp
     mov  bp,sp
;
@1:
     pop  bp        ; restoring the 9 registers
     pop  di
     pop  si
     pop  ds
     pop  es
     pop  dx
     pop  cx
     pop  bx
     pop  ax
     iret             ; iret will restore IP, CS,
                         ; and the flag register.
_nullintr endp
; (ending code deleted for this example)
```

Figure R3-1: A null Turbo C Interrupt Handler

The mechanisms we have described save a total of twelve 16-bit words on the stack, or 24 bytes altogether. At the time of the interrupt, the `SS-SP` must be pointing to the top of a valid stack that has room to store at least 24 bytes.

Before calling further C functions, your interrupt handler should save a copy of the `SS` and `SP` and switch to a different temporary stack. This will ensure that further function calls do not cause the original stack to overflow. Remember to load the `SS` and `SP` by two consecutive simple assignment statements, or to disable interrupts while loading (see Chapter I2). A stack size of 200 bytes is adequate for the MPX interrupt handlers. A possible sequence of code to accomplish the saving and resetting of the `SP` is as follows:

```
#define SYS_STACK_SIZE  200
byte sys_stack[SYS_STACK_SIZE];
unsigned short ss_save;
unsigned short sp_save;
unsigned short new_ss;
unsigned short new_sp;
     .
     .
     ss_save = _SS;
     sp_save = _SP;
     new_ss = FP_SEG(sys_stack);
     new_sp = FP_OFF(sys_stack);
     new_sp += SYS_STACK_SIZE;
     _SS = new_ss;
     _SP = new_sp;
```

Although interrupt procedures are designed to be invoked by interrupts, it is also possible to invoke them using ordinary C function calls. If the keyword `interrupt` has been used in the declaration, the compiler will ensure that the flags register as well as the `CS` and `IP` are pushed onto the stack in the proper order when the function is called.

Handling Parameters

Another important difference between interrupt handlers and ordinary procedures concerns the way that parameters are handled. In general, all parameters must be explicitly accessed from the stack. This problem is considered further in the discussion of the system call handler, below.

Resource Use

In general, it is dangerous and inappropriate for an interrupt handler to make use of resources that could also be in use by the current process. An interrupt handler should not ordinarily access files or I/O devices, nor should it allocate memory or use any memory other than its own. Except for debugging purposes, an interrupt handler should not display messages on the terminal screen.

Caution must be used by an interrupt handler, also, in calling other functions or procedures. It is important to be sure that these procedures do not in turn use inappropriate

resources or violate any of the interrupt handler disciplines. In addition, each function call adds further data to the stack, which can lead to overflow if the available stack capacity is unknown or limited. This problem can be minimized by switching to a temporary stack, as explained above.

Handling Other Interrupts

During normal operation, the IBM-PC must respond to a variety of interrupts, some at a high rate. In particular, clock interrupts must be handled at the rate of about 18 per second, or else the system clock will become inaccurate. When an interrupt occurs, the hardware clears the `IF` to disable other interrupts; this is necessary to avoid confusion during the initial saving of the system state. However, it is necessary for each interrupt handler to re-enable interrupts *as soon as possible* to avoid problems.

If an interrupt handler is *very* short, interrupts may be left disabled until the handler terminates. They will be automatically enabled when the flags register is restored by the *iret* instruction. *In MPX-PC, we will keep all interrupt handlers short enough to rely on this method*. If the handler is not short enough, the `IF` must be explicitly reset.

Special care must be taken when an interrupt handler is allowed to be interrupted. For example, the following questions may need to be considered: Is an adequate stack available for the secondary interrupts to use? Will this interrupt handler still be "short enough," even if it is interrupted? Is this handler fully **re-entrant** (*i.e.*, no global data is modified), in case a secondary interrupt requires use of the *same* handler? This is generally *not* the case if the handler calls MS-DOS functions, since many MS-DOS and BIOS functions are not re-entrant.

Keeping the Handler Short

The final concern is that an interrupt handler should complete its work and terminate in the shortest possible time. Reasons for this have been discussed in the previous subsection. In many cases an interrupt handler should do little more than set an event flag to indicate that a certain event has occurred. Other software should then examine that flag, and carry out the additional work that the event may require.

Context Switching

An important responsibility of a process dispatcher is context switching, that is, saving the context or "hardware state" of a running process in memory, and replacing it with the context of a new process to enable that process to begin (or continue) execution. In MPX-PC this context consists of the content of the 14 hardware registers available to user programs in the basic 8086 architecture. These registers must be saved in each process's PCB or stack. 12 of the registers are the same ones normally saved by the interrupt mechanism, as discussed above. These registers are `BP, DI, SI, DS, ES, DX, CX, BX, AX, IP, CS,` and flags. When a process is not running, these registers are stored (in the given order, top to bottom) in the top 24 bytes of the PCB stack.

The remaining two registers are the `SS` and `SP`. These form the 32-bit stack pointer address, which also must be saved in the PCB.

There are two ways in which register values are placed *into* the PCB:

- During process initialization, your program must store suitable initial values for the 12 registers in the stack area, and set the stack pointer to show that these values are present. In particular, the `CS` and `IP` must be initialized to the value of the starting address of the process code, and the `DS` must be initialized to the appropriate initial data area.

- Whenever a process performs a system call, the system call interrupt mechanism stores the current register values in the PCB stack. The handler must also save the stack pointer itself in the PCB.

The other side of context switching is the mechanism by which the context information in the PCB is restored to the actual hardware registers. This action effectively must cause the process to resume execution where it last stopped. Note that, since the `CS` and `IP` are among the registers restored, this restoration causes an *immediate* transfer of control to the new process. It is the responsibility of the dispatcher to perform this transfer in an orderly fashion.

As described above, a normal return by an *interrupt handler* causes the restoration of all registers except the `SS-SP`. In the usual case, the stack pointer is expected to have the same value just before this return that it had immediately after the registers were saved by the same handler. Therefore, the return causes a complete restoration of the context that existed just before the interrupt occurred.

What happens, however, if an interrupt handler should "return" using a *different* stack pointer than its initial one? The answer is, if the stack pointer points to a different but valid saved context, it will restore this different context, and "return" to a different place. In particular, if the return is preceded by loading the `SS-SP` with the saved stack pointer from a new PCB, the act of returning will effectively restore the context of that process. The new process will now be running, even if that was *not* the point from which the handler was invoked.

This is precisely the technique that your dispatcher will use, and it shows why the dispatcher must be structured as an interrupt handler. In general, the dispatcher will be "called" like an ordinary procedure, but it will modify its stack pointer to perform an automatic context switch when it returns. With one exception, the dispatcher will *never* return to a point from which it was called.

Further details of the dispatcher mechanism are explained below.

R3.3 DETAILED DESCRIPTION

General Discussion

The central goal of this module is to implement a simple round-robin scheduler/dispatcher using the PCB structure from Module R2 to represent each process. The operation of the scheduler is somewhat complicated, and it is important to understand this operation in detail. We recommend that you reread the description of the PC architecture in Chapter I3, with special attention to the use of the registers and stack. You should also be sure you understand the concepts discussed in the previous section. The scheduling process will be overviewed in this section, and we will then describe the required components in more detail.

During initialization of your dispatcher, you must create five test processes. Each process should be established as an application process with default priority. Suggested names for these processes are `test1` through `test5`. All processes should be set to the *ready, non-suspended* state.

Unlike the processes of Module R2, these processes will actually be dispatched; therefore they must have a program to execute. In this module each process will execute a specific procedure (C function) that is linked to your project. These test procedures are available on the MPX support web page in the C source file `PROCS-R3.C`. This file should be added to your Turbo C project. The procedure names are `test1-R3` through `test5-R3.` During initialization of your PCBs you must set the *execute address* to the starting address of the appropriate procedure. The load address and memory size are not significant for this module, and they may be given null values.

To provide a starting context for a process, the stack must be initialized to contain initial values for all of the PC registers which are part of the process context. The initial value for the IP itself must be set to the execute address. Most of the other register values are not significant.

Each test procedure executes a loop which displays a message on the screen a specific number of times. After each message the procedure gives up control by calling `sys_req` using the op code `IDLE`. When the loop has completed the proper number of times a final message is displayed, and `sys_req` is called once more using the `EXIT` op code. This should cause the process to terminate. Note that, under correct operation, the test procedures will never "return" in the usual fashion. If a process is dispatched again after calling EXIT, an error message will be displayed.

The outline of each test procedure is as follows:

```
repeat N times
display progress message
sys_req(IDLE, NO_DEV, NULL, NULL);
end loop
display completion message
sys_req(EXIT, NO_DEV, NULL, NULL);
display error message
```

The principal procedures you are to implement are a **dispatcher** and a **system call handler.** Let's assume these procedures are named `dispatch` and `sys_call`. Both procedures should be defined as functions returning `void` with no arguments. Your initialization procedure creates five processes and installs them on the ready queue. The action begins when you call `dispatch.` The first job for the dispatcher is to locate the process at the head of the ready queue. Since this process is about to transit from the *ready* state to the *running* state, its PCB is removed from the queue. The dispatcher then prepares for a context switch by saving the current `SP` (stack pointer), and setting the `SP` to the current "top" of the process's PCB stack. When `dispatch` then "returns," as explained above, it will actually transfer to the starting point of the new process instead of to its caller.

At this point the first test process is executing, using its own stack instead of the original MPX stack. All is normal until the process calls `sys_req`. Starting with Module R3, `sys_req` will respond to certain op codes (including `IDLE` and `EXIT`) by invoking a *system call interrupt*. using the 8086 *int 60h* instruction. Ordinarily, the interrupt handler associated with *int 60h* would display an error message, since this interrupt is not used by MS-DOS. However, your Module R3 initialization code must change this vector (using the support routine `sys_set_vec`) to point to your system call handler (`sys_call`). Once this is done, each *int 60h* will in effect become a call to `sys_call.`

It is now the responsibility of `sys_call` to process the system call as determined by the parameters supplied. Remember that `sys_call` was declared without arguments. However, `sys_req` has placed the required parameters on the stack. `sys_call` must get these parameters directly from the stack, interpret them properly, and perform the required action.

In Module R3, the action required by an `IDLE` call is to change the process from the *running* to the *ready* state. This is accomplished mainly by reinserting the process on the ready queue. The action required by an `EXIT` call is to terminate this process; in this module termination is equivalent to the *Delete PCB* operation of Module R2.

In either case, a new process must be dispatched. The final act of sys_call is to call dispatch. The dispatcher will then start the next process. As the action continues, dispatch will start or continue each process in circular fashion until all processes are terminated. When the ready queue is finally empty, dispatch will restore the original saved SS and SP, and return to its original caller. Notice that dispatch never returns to sys_call.

Overall Structure and Initialization

The overall structure of this module is based on the implementation of a special temporary *Dispatch* command within your COMHAN command handler. Although this command will not be used in future modules, the `sys_call` and `dispatch` *procedures* which you will write for this module will continue to play a critical role in later versions of the project. It is important that the support routines `sys_init` and `sys_exit` be properly invoked, since they will have important responsibilities beginning with this module. You may rely on your main procedure to invoke these routines. However, the module parameter for `sys_init` must be `MODULE_R3`.

With either approach, the required initialization must be performed prior to the first call of the dispatch procedure. This will be carried out when your *Dispatch* command is first issued. The steps required for the initialization are as follows:

1. Create five PCBs, and initialize them as described in the previous section. These PCBs are linked to the five test procedures.

2. Insert all five PCBs on the ready queue.

3. Link the *int 60h* interrupt to your system call handler using the support procedure `sys_set_vec` (described below). Note that `sys_exit` will restore this vector to its original value.

4. Call your dispatcher to begin execution.

The creation of an initial context in the PCB stack requires a little more discussion. A convenient way to access the register storage area on the stack is to define a C structure which models this area, and create a pointer to the stack which behaves as a pointer to your defined structure type. This can be accomplished with the following statements:

```
typedef struct context {
unsigned int BP, DI, SI, DS, ES;
unsigned int DX, CX, BX, AX;
unsigned int IP, CS, FLAGS;
} context;
context *context_p;
```

The appropriate initial value for the saved stack pointer can be computed using the C `sizeof` operator. This example assumes that the stack for each PCB is allocated separately, and that the PCB contains the stack base address, the stack size, and the current stack pointer:

```
pcb[n].stack_p = pcb[n].stack_base + pcb[n].stack_size
     - sizeof(context);
```

Finally, the context pointer may be set to the correct base value using type casting:

```
context_p = (context*) pcb[n].stack_p;
```

The register copies in the context area can now be accessed as structure elements using `context_p->AX`, `context_p->IP`, etc.

The registers whose initial values are significant include the various segment registers, the `IP`, and the flags. The initial values for `DS` and `ES` should be set to their actual values during MPX execution. The `CS` and `IP` must be initialized to the start address of the appropriate test procedure, and the flags should be initialized to a value in which the `IF` is 1, the `DF` and `TF` are zero, and the status flags are immaterial. Recall that the `IF` is bit 9 of the flags register. The required initialization can be accomplished with the following statements:

```
context_p->DS = _DS;
context_p->ES = _ES;
context_p->CS = FP_SEG(&testn_R3);
context_p->IP = FP_OFF(&testn_R3);
context_p->FLAGS = 0x200;
```

The System Call Handler

The system call handler is responsible for interpreting system call parameters and invoking the dispatcher as necessary. As described above, this handler will be invoked indirectly by `sys_req` using the *int 60h* interrupt. During this invocation, the following sequence of events must take place:

1. The `sys_req` procedure prepares for a system call interrupt by explicitly placing the required parameters on the current stack. In our case the system call parameters are identical to the parameters for `sys_req` itself.
2. `sys_req` generates interrupt 60 using the *int 60h* machine instruction.
3. The interrupt hardware pushes the flag register, `CS`, and `IP` onto the stack, in that order. It then clears the `IF` and loads the `CS` and `IP` with the address contained in the interrupt vector, thus transferring control to `sys_call`.

Because it is invoked by an interrupt, `sys_call` must be structured as an interrupt handler, as explained above. The TURBO C `interrupt` keyword should be used to cause the remaining registers to be saved on the stack.

Although it was declared with no parameters, `sys_call` must gain access to the parameters supplied by `sys_req`. These parameters were placed on the stack prior to the interrupt, and they may now be found on the stack *under* all of the saved registers. The specific parameters to be obtained are:

```
int op_code; /* operation code */
int device_id; /* associated device, if any */
char *buffer; /* address of data buffer, if any */
int *count; /* ptr to (max) no. of bytes to transfer */
```

Only the first of these parameters is significant in Module R3. Others will be used in later modules.

When your system call handler gains control, the stack has the organization shown in Figure R3-2. The offset addresses are given *relative to the address of the top element,* which is the current value of `SP`.

```
Offset Element
0024 (previous stack contents)
0020 buffer length pointer parameter
001C buffer address parameter
001A device_id parameter
0018 op_code parameter
0016 flag
0014 CS
0012 IP
0010 AX
000E BX
000C CX
000A DX
0008 ES
0006 DS
0004 SI
0002 DI
0000 BP
```

Figure R3-2: Stack organization at System Call

There are several ways to access these parameters from your system call handler in Turbo C without resorting to assembly language. One possible solution would be to declare your handler as though all of the registers were actually parameters, along with the desired arguments. However, an important goal of this project module is to learn to manage the stack more explicitly. The method we recommend for parameter access is similar to the method described above for context initialization. It requires the definition of a C structure to represent the parameter stack frame:

```
typedef struct params {int op_code;int device_id;
     byte *buf_addr;int *count_addr;} params;
```

The stack structure must be a named type. A pointer to this structure is also required:

```
params *param_p;
```

Finally, the following assignment will set the pointer to the address of the actual stack frame to be accessed:

```
param_p = (params*)(MK_FP(_SS,_SP) + sizeof(context));
```

It is now possible to refer to the `op_code` value as

```
param_p->op_code.
```

The `op_code` may be used by `sys_call` to identify the system call type and take the necessary action. In this module, the only expected op codes are `IDLE` and `EXIT`. As explained above, the appropriate response to `IDLE` is to prepare the process for a new turn by returning it to the ready queue. The appropriate response to `EXIT` is to terminate the process. Both of these tasks can be accomplished using routines from Module R2.

Since the system call handler is responding to an explicit interrupt request generated by software, it is possible and appropriate to return a result value. The result value for `sys_call` should be a zero value to indicate success, or a negative error code to indicate that a problem occurred. One possible problem would be detection of an invalid op code. We will follow MS-DOS conventions by returning this result in the `AX` register. However, since the process context has been saved, the result must actually be placed in the *copy* of `AX` in the PCB, using a statement such as

```
context_p->AX = result.
```

This result will be retrieved by `sys_req`, and used as its own return value, when the context is restored.

With its work completed, `sys_call` now must trigger the dispatching of the next ready process (if any) by calling `dispatch`. This call should be the last statement in `sys_call`, since in fact `dispatch` will never return.

The Dispatcher

The role of the dispatcher is to identify the next ready process, if any, and dispatch it by loading its context from its PCB. The dispatcher is actually a very short and simple routine, but also a very critical one. In this discussion, let's assume your dispatcher is named `dispatch.` Assuming that a PCB is found in the ready queue, the dispatcher must perform the following steps:

1. Remove the first PCB from the ready queue.

2. Set a pointer to this PCB identifying it as the running process (we suggest the name `cop` for "current operating process").

3. Copy the stack pointer from this PCB to the actual `SP` to prepare for context restoration.

One special consideration is the fact that the dispatcher rarely returns in the normal way. In fact, it *never* returns to the system call handler, since a system call always results in the calling process being rescheduled or terminated. The only other call to `dispatch` is the initial call from your test program or *dispatch* command. `Dispatch` *will* eventually return to the point of this call, but only after the entire sequence of process dispatching has been completed. To accomplish this trick, dispatch must save the `SS` and `SP` the very first time it is called, and restore this saved `SS` and `SP` when the ready queue is finally empty.

To save the initial `SP`, we must make use of global variables `ss_save` and `sp_save` which must be initialized to `NULL`. `Dispatch` will copy the actual `SS` and `SP` into these global variables if and only if `sp_save` is found to have a null value. When dispatch is ready for its final return, it must copy `ss_save` back to the `SS` and `sp_save` back to the `SP`.

The following is a complete outline for the dispatcher:

```
if sp_save is null,
ss_save = _SS
sp_save = _SP
remove the PCB at the head of the ready queue
if a PCB was found
set cop to this PCB
set _SS and _SP to the PCB's stack pointer
else
set cop to NULL
set _SS to ss_save
set _SP to sp_save
end if
"return from interrupt"
```

R3.4 SUPPORT SOFTWARE

Your software for Module R3 will make use of a subset of the support procedures described for Module R1, plus one additional routine. `sys_init` and `sys_exit` will of course still be used, invoked as usual by your main program, with `MODULE_R3` as the appropriate `sys_init` parameter. The new support procedure is `sys_set_vec`. Its description is given below. No other support procedures are needed directly by the software for this module.

As before, all support procedures are found in the C source file `MPX_SUPT.C`, and their defining prototypes, along with other necessary definitions, are in the file `MPX_SUPT.H`.

sys_set_vec

The `sys_set_vec` function is called to set the appropriate interrupt vector to point to your system call handler. This establishes the necessary linkage so that the interrupt instruction that will be generated by `sys_req`, beginning with Module R3, for certain op codes, will in fact cause a transfer to your interrupt handler.

The prototype for `sys_set_vec` is:

```
int sys_set_vec (void (*handler)());
```

The single parameter `handler` is a pointer to your system call handler. If your handler has the name `sys_call`, then it should have the prototype:

```
void sys_call (void);
```

Your call to `sys_set_vec`, which should occur during initialization, will have the form

```
sys_set_vec(sys_call);
```

The returned value will be zero if no problem occurred; otherwise it will be an error code. The error can occur if the specified handler address is invalid. The returned error code in this case is:

ERR_SUP_INVHAN	invalid handler name

The symbol for this code is defined in `MPX_SUPT.H`.

R3.5 TESTING AND DEMONSTRATION

The test procedure for Module R3 is simply to run the scheduling sequence, either by starting your standalone program or by issuing the dispatch command to COMHAN. This should cause the five test processes `test1` through `test5` to be dispatched in sequence. Each process will display a one-line message on the terminal. These messages have the form

`test`*n* `dispatched; loop count = 1`

The processes should continue to be dispatched in order until each one terminates. Each time a process is dispatched it will display a similar message, with the loop count incremented by 1. The five test processes are set to terminate after 1, 2, 3, 4, and 5 iterations, respectively. *Your* software should display a message when a termination request is detected. If a process is dispatched after it requests termination, it will display the error message

`test`*n* `dispatched after it exited!!!`

When all processes have terminated, a final message should be displayed by your software. The *Dispatch* command should then return to COMHAN. It is important to demonstrate that the system will return cleanly to normal operation after all processes have terminated. It should be possible to repeat the *Dispatch* command as often as desired.

R3.6 DOCUMENTATION

You should briefly document the procedure for demonstrating your dispatcher. Because the dispatch sequence for Module R3 is a temporary command, you should not add its description to your *User's Manual*.

The *Programmer's Manual* should be updated to include a careful description of your `sys_call` and `dispatch` procedures, and any related support procedures and data structures which you defined.

R3.7 OPTIONAL FEATURES

Since the system call handler and dispatcher are critical elements of MPX, we do not recommend any extensions or variations for this module.

R3.8 HINTS AND SUGGESTIONS

The key to success in this module is to *carefully* manipulate the stack. The Turbo C debugger can be used to verify register and stack contents; however, the debugger itself can sometimes become confused when the `SS` and `SP` change (see Chapter I5).

Nevertheless, judicious use of breakpoints and watchpoints are probably your best strategy. Do not use `printf` statements in your program while the `SS-SP` is set to a PCB or other

limited stack; these statements can overload the stack and cause failures that would not occur when the statements are removed.

Remember to always load `_SS` and `_SP` using two consecutive simple assignment statements, and/or to disable interrupts while they are being loaded.

The two most critical places to verify the stack and registers are at the beginning of `sys_call` and the end of `dispatch.` a valuable check which *can* be done with printf statements is to verify that `_SP` has the same value before and after the `dispatch` procedure is called by the *Dispatch* command routine.

-