Chapter F

Final MPX

F.1 INTRODUCTION

This module is the final step in the MPX project. In this module you will build a complete MultiProgramming eXecutive by integrating the required and optional project modules that you have previously completed, along with other modules that your instructor may provide.

The basic elements of MPX are the user interface, process management, dispatching and interrupt handling, character device management, and **I/O management**. All of the basic capabilities except I/O management have been covered by previous required modules. In this module we will add I/O management by implementing an I/O request handler and scheduler to process I/O requests by test processes, using the MPX device drivers which you have developed or which your instructor will provide.

Another important element of the Final MPX is a new, more realistic philosophy for dispatching processes. In previous modules, the MPX main program and command handler were implemented as static, standalone programs. Dispatching of processes was carried out by command for a short time interval, after which the command handler resumed control. Only the simple test programs were executed by processes.

In the Final MPX, dispatching is continuous, and the command handler itself is a process. This means that execution of the command handler is interleaved with execution of other processes under the continuous control of the dispatcher. Each process, including the command handler, executes until it performs an MPX system request (call to sys_req) or until an interrupt occurs. System requests or interrupts may cause a new process to be scheduled by a context switch. Most system calls will be I/O requests. In the final MPX all requests for keyboard input or terminal (screen) output are intercepted and handled by MPX, even though they may be invoked by standard C functions. Interrupts are caused by I/O events.

After implementing the final MPX, you will demonstrate its capabilities by loading and activating an assortment of test processes using *Load* and *Resume* commands (all loaded processes are initially suspended). These processes will then be dispatched on a continuous basis, alternating with the command handler itself. Process priorities may be adjusted, leading to different relative execution rates. The command handler will always have the highest priority.

The test processes supplied assume that your PC is connected to a remote “terminal” (via one of the serial ports), or running the serial port simulator provided by your instructor, and that you have implemented the appropriate interrupt-driven device handlers. Adjustments should be made to the test processes if you have a different configuration.

Every process but one (the IDLE process) outputs regular messages, either to the display screen or to another device. One process besides the command handler accepts input (from a remote device). Some processes will be *CPU bound*, that is, they will compute a relatively long time between output messages. Other processes are *I/O bound* and generate messages continuously. Finally, some processes will run indefinitely until manually terminated, while others will run for a fixed number of repetitions and then terminate themselves.

Devices are not exclusively allocated in MPX, so process output will be intermixed on each device. The output pattern should reflect the relative priority and characteristic behavior of each process. This pattern should change as processes are started and stopped and as priorities are adjusted.

This chapter provides details for the structure and organization of MPX-PC. Section F.2 discusses the key concepts introduced in this module: *Continuous Dispatch* and *I/O Management.* Section F.3 presents a detailed description of the organization of MPX-PC and of the new components that must be implemented to complete the system structure. Section F.4 discusses the support software applicable to the final module. The principal addition to the support environment is a collection of test processes. In addition, a substitute terminal driver is provided to route ordinary terminal I/O from the command handler through the MPX system call handler. The purpose of this is to allow the command handler to be interrupted by other processes while terminal input and output are underway.

As usual the remaining sections of the chapter cover test procedures, documentation requirements, possible extensions, and hints and suggestions. The documentation for this module is of special importance, since it should constitute the final, comprehensive documentation package for the entire MPX project.

F.2 KEY CONCEPTS

Continuous Dispatch

Previous modules which dispatched processes have done so only when requested by a specific command, and only for a limited period of time. Most of the time MPX was executing the command handler, which was *not* considered to be a process.

In a realistic multitasking operating system, however, almost all programs (except the operating system resource managers) are viewed as processes. This includes command handlers. Execution of the command handler should be interleaved with other processes running on behalf of the same user or other users.

MPX is a single-user system, since there is only one terminal, but it is a multiple process system. In its final form the OS must permit a user to initiate many processes and keep them executing even as the command handler is running. This is the approach we take in this module; under normal execution the command handler is a process, and it competes for processor time and other resources along with any other processes that may be ready.

When dispatching is continuous, it may happen that *no* process is ready. In order to simplify the dispatcher and avoid dealing with this special case, we introduce an *idle* process which is *always* ready. This process has the lowest possible priority, but it will run when no other process is ready, thus assuring that there is always some process to run. In particular, the idle process will run while the command handler is waiting for a command, if no other processes have been loaded and activated.

If continuous dispatching is the normal situation, a more complicated initialization procedure must be carried out to activate an initial set of processes. When a multitasking OS is first initiated, it executes an initialization program which is *not* a process. This program sets up the initial processes and calls the dispatcher for the first time. Similarly, a special procedure may be required at termination.

In the final MPX structure, the main program begins by calling the usual initialization procedures. Among other responsibilities, these procedures set up the PCBs and initialize the process management system. The main program then activates two processes: one for the command handler, and one for the *idle* routine. The command handler, like the processes of Module R3, is already present in memory; The *idle* process, like those of Module R4, must be explicitly loaded. Both processes are set to the *ready, not suspended* state, but the command handler has the highest possible priority and the idle process has the lowest.

The main program then calls the dispatcher to begin interleaved execution of these two processes. Command handler commands may now be used to explicitly load and activate other processes, which will proceed to share the resources with the original two. All processes *except* the command handler and *idle* process may be modified and terminated under user control.
The command handler terminates when a *quit* command is received. Its final action is to remove itself and *idle* as active processes. When the dispatcher detects that no processes are active, it returns to the main program (which in MPX has been allowed to remain in memory). This program performs the final cleanup and exits from MPX.

I/O Management

In previous modules you have developed a device handler for a device. This handler provides low-level procedures to open and close devices and to transfer blocks of data to or from the device. This transfer could be performed without interrupts, but to maximize concurrency we have implemented interrupt-driven device control and included interrupt handlers which are activated after each character is transferred, in order to begin transfer of the next character, or to terminate the operation if the entire block has been processed. A set of drivers such as you have implemented forms an important part of a complete I/O management system, but only a part. In your initial implementation these drivers were called directly by test programs. In a complete multitasking operating system they must be called in an orderly way in response to requests that are generated by running processes. This requires an I/O management and scheduling strategy, as described in the *Device Management* chapter of your textbook.

A key element of an I/O management system is an I/O Control Block (IOCB) for each device or channel. This data structure has a role complementary to that of the Device Control Block (DCB) which you have embedded in the driver itself, but it is *not* the same structure. The IOCB contains information allowing higher-level software to *access* the device driver.

Some of the information in the IOCB describes permanent characteristics of the device: name, channel number, etc. This information also specifies the interrupt IDs and interrupt vectors associated with the device, and the address of each procedure in the device driver. This latter information must be kept current in case the driver is loaded as needed into transient areas of memory.

Additional information in the IOCB identifies the current operation, if any, that is underway for this device. The ID of the process which requested the transfer is recorded in the IOCB. Finally, the IOCB includes an *event flag*, as introduced in Module R5, that can be used by an interrupt handler to report back to the system about the current status of the I/O transfer.

A typical I/O device can process only one transfer at a time, but additional requests may be received before the current transfer is completed. For this reason there must be a *waiting queue* for each device which may contain pending I/O requests. Each entry in this queue describes an I/O request by identifying the process making the request, the buffer location, and the type of transfer to perform. Additional information is required in the case of an addressable device such as a disk; these cases are not included in MPX-PC.

Requests for I/O transfers are initiated by processes using system calls, and passed to an *I/O scheduler*. This routine examines the IOCB to see if the device is busy; if so the request is placed in the waiting queue. This queue is normally organized in a simple first-come, first-served order.
If the device is not busy, the request may be passed to the *read* or *write* procedure in the appropriate device driver. This procedure initializes the buffer, enables the appropriate interrupt, and (if the transfer is a write) outputs the first byte to the device's output port.

In either case, the process is moved from the *ready* state to the *blocked* state (unless non-blocking I/O was requested). A context switch is performed, and another ready process is dispatched.
As the transfer of each byte is completed, an interrupt occurs, generated by the device's hardware interface. This type of interrupt-driven I/O was developed in Module R5. The interrupt handler in the device driver checks to see if more data is to be transferred. If so, it processes the next byte and returns to the interrupted process. Note that in general this is *not* the process that requested the data transfer.

When the interrupt handler detects that the final byte has been transferred, it must set the event flag in the IOCB. This flag is a signal to the perform an *I/O completion* sequence. This typically involves returning the requesting process to the *ready* state and setting an event flag to notify that process that the transfer is complete. The waiting queue for the device must also be checked, and if there are requests waiting, the next transfer must be initiated by a new call to the device driver. Finally, either the interrupted process may be resumed, or a new process (possibly the one which had requested the I/O) may be scheduled instead.

F.3 DETAILED DESCRIPTION

General Structure

This section presents the general organization and flow of control in the complete MPX-PC system. The elements of the system include the following:

- Main program
- Command handler
- Processing routines for each command
- Dispatcher and system call handler
- Device drivers
- I/O scheduler
- I/O completion handler
- System data structures
- Storage area for the loaded processes.

Initial operation begins with an initialization procedure which is different from that of previous MPX versions. The key responsibility of the initialization procedure is to set up two initial processes. One of these is a special IDLE process which runs only when there is no other process; the second is the command handler itself. The initialization procedure creates PCBs for each of these processes. Each is a system process; the command handler has the highest possible priority, and the idle process has the lowest. The processes are installed on the ready queue, and the dispatcher is started. Since the command handler is now the highest priority ready process, it will immediately be dispatched. The command handler will perform its own initialization and wait for a command. To receive command input an MPX system call is invoked for data transfer from the "terminal" (i.e., the keyboard). This request places the command handler in a blocked state, so other processes may be dispatched.

Using the usual commands, various test processes can then be loaded and made ready to run. When placed on the ready queue, they will automatically be dispatched when their turn comes. Most of these processes will make input and output requests, invoking the I/O management facilities which are to be implemented for this module.

A typical data transfer operation begins with a request such as READ or WRITE issued from within the normal program code of an application process. This request specifies the transfer of a block of data (typically a line of text) from a program buffer to a specified device, or vice versa. In MPX such a request is made using *int60h* which invokes the support routine `sys_req`. This in turn invokes the system call handler provided by your program.

The call handler must process READ and WRITE requests in the same manner that it has previously processed IDLE and TERMINATE operations. In principle, each request invokes a call to the appropriate routine in one of the I/O drivers. But it is not quite this

simple. There are multiple processes, and we must be sure that another process is not currently accessing the same device. Although devices are not allocated to processes on a long-term basis, we must ensure that the transfer of each block (i.e., line of characters) is completed before a new transfer begins.
To resolve competition for devices, the system call handler passes each request to the I/O scheduler. This scheduler maintains a record (in the form of a set of queues) of which processes are currently using, or waiting to use, each device. If the requested device is busy, the request is placed in the appropriate waiting queue.

If the device is not currently in use, the request can be honored immediately. In this case a call is made to the appropriate device driver procedure (e.g., `com_write`, `com_read`). This procedure sets up the transfer information in the IOCB, enables the appropriate interrupts, and begins transfer of the first byte.

In either case the process is placed in the blocked state. The dispatcher is called, and a new process is dispatched.

As each byte is transferred, an interrupt occurs which invokes the interrupt handler in the device driver. This handler determines whether there is more data to be transferred. If so, it processes the next byte and returns to the point of interruption. If not, it sets the event flag, signaling that the device has completed a transfer. This flag will be detected during the next system call, and cause activation of the I/O completion sequence.

The completion sequence has several responsibilities. It must return the process which was performing the I/O to the *ready* state, remove it from any waiting queue, and reinstall it in the ready queue. In addition, the handler must determine if there are requests pending in the waiting queue for the device. If so, it must setup the transfer and call the device driver as described above.
This cycle continues until MPX is terminated by an explicit command. The remainder of this section describes these components and operations in more detail.

Data Structures

The principal data structures to be added for this module are the IOCBs which record information about the current transfer and any other requested transfers for each device. While it is possible to maintain a combined waiting queue, we recommend a separate queue for each device.
Each active or pending I/O request must be represented by some type of descriptor. This descriptor must identify the process, the device, and the operation. In addition it should

indicate the location of the transfer buffer and the count variable. Note that, except for the process ID, this is the information passed as parameters during a system call.

It is possible to define a special I/O descriptor record type to hold this information. Alternately, space for an I/O descriptor may be allocated within each PCB. Note that, in MPX, a process may have only one I/O request at a time.

The IOCB must make it possible to locate the current active request for a given device, and to select a new one from the waiting queue when necessary. The IOCB must also provide the event flag to be used to control I/O for each specific device.

System Initialization and Termination

In previous modules you may have set up your command handler as the main procedure for MPX, or you may have provided a separate main procedure which invoked the command handler after suitable initialization steps. At this point we recommend that you use a separate main program. The details of our discussion here will be based on this strategy.

The initial steps of the main procedure must perform all required initialization for the various components of your MPX package. In particular this must include a call to `sys_init` (using the parameter `MODULE_F` plus the code for any optional modules included). A call to `sys_set_vec` is also required to notify the system of the identity of your system call handler.
It is also necessary at this point to call the open procedures for the terminal (console) and the other devices you will be using. The terminal driver is supplied with the MPX support software; the other drivers were developed in Module R5 or will be supplied by your instructor. These procedures include (at least) `trm_open,` and `com_open`. No I/O operations, including terminal output via `printf,` should be performed until the corresponding drivers are opened. At the same time, the IOCBs and their associated waiting queues should be initialized.

The next steps to be performed are those previously carried out by the *Dispatch* command in Module R3. These steps include initialization of the PCB collection, setup of two PCBs, and installation of both PCBs on the ready queue. One PCB is assigned to the command handler. This PCB is initialized in the same manner as the PCBs for the "directly linked" processes of Module R3. Its initial execution address (and the initial IP value on the stack) must be set to the entry point of the command handler procedure. This process is designated a System process, given the highest possible priority, and placed in the *ready, not suspended* state. Be sure to use the correct segment values for the initial

context of each of these processes, as in Module R3 and R4.
The PCB assigned to the command handler may require a *larger stack* than the other PCBs. This is because the command handler is a full C program which may perform complex nested operations including file transfers. A recommended minimum stack size *for this PCB only* is 4K bytes.

The second PCB is assigned to the IDLE process, which must now be loaded. This process is found in the executable file `IDLE.MPX`. It should be loaded into a suitable allocated memory area, using the strategy developed for the *Load Program* command of Module R4. This process is also designated a System process, given the lowest possible priority, and placed in the *ready, not suspended* state.

The final action of the initialization sequence is to invoke the dispatcher. This is done as in the Dispatch command of Module R3 or R4. The dispatcher will proceed to dispatch processes, starting with the command handler. It will return only when the ready queue becomes *completely* empty.

Termination of MPX is invoked in response to the *Terminate MPX* command, as in previous modules. When the command processing loop is exited, the command handler should terminate *all* processes present, including its own process and the IDLE process. The dispatcher will detect an empty queue and return to its original caller, the main program.

The main program should then perform any necessary final cleanup actions, including the closing of all device drivers. This closing is necessary to restore the original interrupt vectors. When this is completed, the program may exit.

An outline of the MPX main procedure, as described above, is as follows:

```
sys_init(MODULE_F + ...)
sys_set_vec(sys_call)
open device drivers
initialize DCBs and waiting queues
initialize PCBs and ready queue
install command handler as a process
load IDLE and setup a process
other initialization as needed
call dispatcher
close device drivers
other cleanup as needed
sys_exit()
```

Command Handling

The command handler should be organized as in previous modules, except that it no longer performs overall system initialization, and it does perform termination as described above. The outline for this version of the command handler is:

```
display opening message
while not done
        display prompt
        read command
        analyze command
        execute command
end while
display closing message
terminate all processes
```

The commands to be used for Module F consist of all of the permanent commands introduced in previous required or optional modules. No new commands are required.

Dispatcher

The dispatch procedure for the Final MPX continues to operate as in previous modules. No change should be required.

System Call Handler

The system call handler for the Final MPX should have the same structure as in previous modules but expanded functionality. It must now handle READ and WRITE operations in addition to IDLE and TERMINATE. These operations will be passed to the system call handler by `sys_req`.

Each time it is invoked, before calling the specific service routines, the system call handler should perform two additional functions:

1. Call the procedure `trm_getc`, as described in Section F.4. This procedure transfers pending keyboard characters from the MS-DOS buffer to the MPX buffer.
2. Determine if any event flags are set, and perform the required IO completion sequences as described below.

Each I/O system request should be passed in turn by the system call handler to the I/O Scheduler. This procedure will check the state of the device and determine whether to initiate I/O or to place the request in a waiting queue. The request parameters (which are still on the stack), and the identity of the calling process, must be provided to the I/O Scheduler. The Scheduler should check the validity of the parameters, and return an error code if the request is invalid.

I/O Scheduler

The I/O Scheduler processes input and output requests. Its first task is to examine the system call parameters and ensure that the request is valid. In particular, the operation must be READ or WRITE; the device must be a recognized one, and the operation must be legal for the specified device. If these conditions are not met, an error code should be returned.

The next task is to check the status of the requested device by accessing its IOCB. If there is no current process using the device, then the request can be processed immediately. In this case the requesting process is made the active one by installing a pointer to its PCB in the IOCB. The buffer address and length must also be placed in the IOCB. The appropriate driver procedure is then called. (In the first edition of the text, these steps are performed by a separate component called the *I/O Supervisor*.)

If the device was busy, the request is installed on the waiting queue. The information in each queue element must include the PCB pointer, device ID, and operation code.

In either case, the requesting process is switched to the *blocked* state, and removed (if it was still present) from the ready queue. Note that in the case of output to the terminal using `trm_write`, `trm_clear`, or `trm_gotoxy`, no imterrupts are used, and the output will be completed *immediately*. Nevertheless, processes using these operations should cycle through the blocked state to give other processes a turn to operate.

Finally, the I/O Scheduler returns to the system call handler, which in turn will invoke the dispatcher to dispatch the next process.

I/O Processing

I/O processing for the duration of the block transfer is managed by the routines of the device driver, primarily the interrupt handler. These drivers should be unchanged from previous modules. Note that while the transfer is continuing under interrupt control, processes other than the requesting process will be executing.

When the interrupt handler detects that the entire block has been transferred, it sets the event flag to request the I/O completion procedure.

I/O Completion

Each time the system call handler is invoked, one of its responsibilities is to examine the data structures that represent active I/O transfers, to determine if any of their event flags are set. For each flag that is set, the appropriate completion sequence must be performed.

First, the active process must be switched from the *blocked* state to the *ready* state. This may require both setting of state variables and adjustment of the appropriate queues.

Second, the active data structure must be cleared, signaling that no request is currently active for this device.

The routine then searches the waiting queue for another process waiting to use the now-available device. If such a process is found, the I/O scheduler is called to start the I/O. This sequence is repeated for all devices with a just-completed I/O request.

F.4 SUPPORT SOFTWARE

Test Processes

A collection of test processes is provided for MPX in the form of executable program files. Each file is provided in the support package with a name extension of `.MPX;` these are actually `.EXE` files similar to those loaded in Module R4. These programs have been written in assembly language (using Borland's Turbo Assembler) to avoid the sizable run-time framework which is attached to all programs generated using Turbo C (this would amount to as much as 10K bytes per process). The assembly language files are included for study with the support software.

Twelve test processes are supplied. If you have access to an assembler you may prepare additional test processes following a similar model. The supplied processes are:

- IDLE. The process which is used to ensure that the ready queue is never empty, and that the system always has some process to execute. IDLE does nothing but wait in a simple loop until an interrupt occurs.
- CPUTERM. A process which repeatedly displays a message line on the display screen. This process is CPU bound; it waits for a while in a loop between messages, consuming processor time. It runs continuously until terminated.
- CPUCOM. Similar to CPUTERM, but this process outputs its messages to the communication port. We assume that a standard terminal, or a separate PC or workstation running terminal emulation software, is connected to this port.
- IOTERM. Similar to CPUTERM, but this process includes no delay loop. It attempts to continuously output messages, making a new request as soon as the last one has been completed.
- IOCOM. Similar to IOTERM, but this process outputs its messages to the communications port.
- IOTRM25. Similar to IOTERM; however, this process displays its normal message exactly 25 times, then requests termination. A special message is displayed before the termination request. If the process is dispatched after its termination request, an error message is displayed, and the process restarts.
- IOCOM25. Similar to IOTRM25, but this process outputs all of its messages to the serial port.

Each process is provided in both assembly language form (e.g. `IDLE.ASM`) and loadable form (`IDLE.MPX`). The loadable processes should be copied to your preferred process directory, so they can be listed using the *Directory* command and loaded using the *Load* command.

Terminal Driver

The second component of the support software is a terminal driver. This driver supplies a set of routines to invoke terminal (screen) output and terminal (keyboard) input, analogous to the routines in the serial port driver. The prototypes for the standard driver routines are:

```
int trm_open (int far *eflag_p);
int trm_close(void);
int trm_read(char far *buf_p, int far *length_p);
int trm_write(char far *buf_p, int far *length_p);
```

The purpose of these routines and the meaning of the parameters is similar to that of the serial driver routines and parameters, as explained in Module R5.

One additional terminal driver function is provided, called `trm_getc`. It has the following prototype:

```
        void trm_getc(void);
```

Keyboard input is acquired via interrupts and stored in a ring buffer. The `trm_getc` function echoes stored characters and transfers them to the requestor's buffer. This function should be called by the I/O Scheduler just before it calls the dispatcher, if terminal input is active. It is not harmful to call this routine when it is not needed, but of course it is inefficient.

F.5 TESTING AND DEMONSTRATION

Testing of this project is performed by loading and running as many of the supplied processes as possible, but you should build up to this capability a little at a time. First ensure that the command handler still works properly now that it has been converted to a process. Repeat the testing of various commands from Modules R1 and R2.

Next try to load and resume one process that performs serial output, such as CPUCOM. If this process works properly, then (if you have incorporated the necessary driver) try a process which accesses the serial port. See if you can run these processes both separately and together.
Loading multiple processes for the same device is usually where it all falls apart. This is because until multiple processes request the *same* device, no process ends up in the waiting queue. Don't be surprised if your MPX-PC crashes when you load multiple processes that request an I/O operation to the same device.

If your project still survives, the final step is to try to activate as many processes as possible at the same time, and to test a full range of commands while these processes are running.

F.6 DOCUMENTATION

You must finalize your documentation in this module. Your *User's Manual* must contain documentation on *all* commands and error messages. The *Programmer's Manual* must include descriptions of all of *your* procedures and data structures. If you have been adding to this manual in each module (as required), you will have only a limited number of additions.

Both of your manuals must have a title page and a table of contents and their pages must be numbered. An index in each is desirable. Refer back to section I1.6 for a review of the information required in each manual.

F.7 OPTIONAL FEATURES

Since the primary goal of this module is to integrate the required and optional elements which you have developed in previous modules, no optional extensions will be specified here.

F.8 HINTS AND SUGGESTIONS

The software for this module is more complex than any other, and a great deal of concurrent activity must be managed. However, a thoughtful approach will keep you from being overwhelmed. The actual amount of new software needed for this module is small. Do not make changes to previous components unless absolutely necessary. If this rule is followed, you can be sure that any difficulties are localized to a small number of routines.

Secondly, follow a careful, incremental testing strategy, as described in F.5. First, ensure that the command handler and idle process work properly. At this point you are not using any of your own device drivers, and there can be no processes on a waiting queue. Thus the debugging process is simplified.

With some care, the Turbo C++ debugger can still be used, and breakpoints can be set. However, note that the debugger automatically restores the normal keyboard interrupt vector. It is not generally possible to resume MPX execution *after* a breakpoint unless the MPX interrupt vector is explicitly replaced. The Turbo Debugger formerly sold separately, and the debugger included with Borland C++, have more comprehensive features to enable debugging of interrupt handlers and I/O routines. Note, however, that the separate debugger is no longer sold, and existing copies will *not* work with Turbo C++ versions later than 1.0.

When the basic processes are operating, choose new processes to load carefully, to exercise one new feature at a time.

Consider very carefully the situations in which interrupts may occur, and the possibly conflicting actions which could be taken in response to those interrupts. If an interrupt occurs during queue manipulation or memory allocation, and the interrupt handler attempts to invoke the same procedures, problems will occur. It may be necessary to disable or defer interrupts during certain critical operations.