Chapter R4
Program Management

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. 23, 2011

R4.1 INTRODUCTION

This module, the fourth required module of MPX, provides a mechanism for loading actual programs dynamically from files, and dispatching processes which execute these programs. You will add a *Load-Program* command to COMHAN which creates a process and simultaneously loads a program for that process to execute. This replaces the *Create-Process* command of Module R2. One of the tasks of the *Load-Program* command is to dynamically allocate memory into which the process will be loaded. In addition, you will replace the *Delete-PCB* command with a *Terminate-Process* command that releases the program memory when the process is destroyed.

A temporary *Dispatch* command will also be implemented in this module, which will dispatch the loaded programs to verify correct execution. This command replaces the *Dispatch* command of Module R3. Each program loaded during this phase will perform one or two system calls and then terminate. During the final integration phase of MPX, you will reorganize the scheduler so that all ready processes are continually dispatched in a round-robin sequence. At that time, the *Dispatch* command will no longer be needed.

R4.2 KEY CONCEPTS

Program Loading

The processes dispatched in Module R3 executed program code contained in procedures that were directly linked into the MPX software itself. However, in a realistic operating system processes will be expected to execute code that is loaded dynamically from program files. This type of program loading is a major element of Module R4.

Loadable program files are normally produced from source code by a sequence of compiling (or assembling) and linking steps. Each resulting program may be built from multiple modules, but it is independent of other programs and of the operating system itself. Thus, the programs to be loaded in Module R4 are not linked to MPX and cannot reference MPX functions or data structures by name. The only communication between loaded programs and the operating system takes place via a fixed system call interface such as the use in MPX of *int60h*.

The information actually loaded into memory from a program file consists of machine instructions and statically initialized data. In the simplest case, the file would consist purely of a copy of this information exactly as it is to appear in memory. This is precisely the case for the simplest type of loadable file supported by MS-DOS, the `COM` file. This type of file contains a single contiguous block of data that is intended to be loaded directly into memory.

Although this approach is straightforward it presents some difficulties. A program that is ready to run contains absolute address references in branch instructions, load instructions, etc. This program will only execute properly if it is loaded into a specific, predetermined region of main memory. All `COM` files are expected to be loaded into a region of no more than 64K bytes beginning at address *0100h*, and to begin execution at location *0100h* after loading.

This convention is sometimes acceptable for a simple OS like MS-DOS, which is designed to load and execute only one program at a time. The region beginning at 0100h is designated as the program load area, and all programs are placed in it, so the `COM` assumption is valid. However, the fixed-address assumption causes difficulties for a multiprogrammed OS like MPX (or Windows!), which may need to load program files into unpredictable locations. The problem can be partially overcome by careful use of the 8086 segment registers, but the most flexible approach requires a file with relocatable program code and data. This type of file generally contains extra information describing the adjustments to be made when the load address is known, and relies on hardware and/or software translation to make the adjustments when the program is actually loaded.

A second difficulty with a "pure image" file type such as `COM` is that there is no easy way to tell a valid file from an invalid one. Indeed, MS-DOS will happily try to load and execute *any* file with a `.COM` name extension, perhaps with disastrous results. This problem can be avoided by providing a short header sequence in the file (not to be loaded), which contains (along with other information) a special code identifying the file type. Programs designed to process a certain type of file can begin by checking this code, and rejecting the file if the code is invalid.

The MS-DOS `EXE` file type was designed to address these difficulties. An `EXE` file contains a descriptive header and relocation information together with the basic data to be stored in each of the program's segments. This type of file format is more complicated to process but more flexible and reliable to use. All of the MPX files loaded by MPX in this module and future modules are actually MS-DOS `EXE` files. The MPX support software makes use of program load services built into MS-DOS and Windows to process these files.

To actually load a program file, a region of memory must be allocated. In general some type of dynamic allocation will be used, especially in a multiprogrammed OS. The necessary size for the program region (or regions) must be determined by inspecting the program file. If a segmented memory architecture such as the 8086 is used, it is important to realize that the segment register values to be used during execution of the loaded program will, in general, be different than those used by the OS itself.

Once the file is checked for validity and the memory region is allocated, the file information can be copied into the allocated region. In the case of a pure image file, this can be achieved simply by reading and storing the consecutive bytes of the file. For a relocatable file, it is necessary to perform the necessary translations for the file information as it is loaded.

-

A final important step is to determine where execution is to begin, and load the appropriate address into the `IP` in the initial context for the process. For a pure image file without header, we must assume that execution is to begin at the first location in the data block. If a header is present in the file, one field in the header can be used to specify the desired start address.

Care must be taken to ensure that the loaded program will terminate properly after execution, allowing the OS to regain control. One approach is to insist that every program terminates with an explicit system call; this is the approach adopted by MPX. Alternately, the loaded program could be viewed as a procedure (and expected to terminate with a *return* instruction). In this case an additional return address must be placed on the initial stack, providing a suitable final place for the program to return to.

In principle, loadable program files for MPX can be produced by Turbo C or another high-level compiler. However, all Turbo C programs are linked to a large runtime library, and so have a minimum size of over 10K bytes. In general these programs may also require a stack size of at least 4K bytes. The possibility of supporting the loading of Turbo C programs as processes, perhaps with some restrictions, is suggested as an optional extension (see Section R4.7). For the basic implementation, a suitable set of test processes has been provided written in assembly language, and assembled using the Borland Turbo Assembler.

R4.3 DETAILED DESCRIPTION

General Discussion

This Module introduces mechanisms to control the loading and unloading of programs to be executed by processes. A process should be created when a program is loaded, and terminated when a program is unloaded. You must allocate memory dynamically for program loading using the system support routines.

The dispatching mechanism developed in Module R3 will be used here to dispatch and execute the loaded programs.

The project requires implementation of three commands. These are described in detail below.

Program Management Commands

In this module you will add commands to your command handler to load and unload programs, and also to create and terminate, respectively, processes which execute these programs. In some operating systems, such as UNIX, process creation and program loading are separate operations. In MPX these operations will always be done together. The commands for creation/loading and deletion/unloading will become permanent parts of your MPX system. You will also need to implement a temporary command to dispatch the processes on the ready queue.

This section describes the required command operations to be added. As usual, each description includes *suggestions* for command and argument definitions. You may modify these suggestions to suit your own command format. However, all of the specified functionality should be included in your system.

-

Load Program

This command is an expansion of the *Create PCB* command of Module R2. As before, it should allocate and setup a new PCB. In addition, it must allocate program memory using the support procedure `sys_alloc_mem`, and load a program into that memory using the support procedure `sys_load_program`. The support procedure `sys_check_program` may be used to ensure that a given program file is present, and to determine the memory size it requires. Much of the work is performed by the support procedures using the relocating loader built into MS-DOS. The outline for the load command algorithm is simply:

```
Check arguments
Check program
Create and setup PCB
Allocate program memory
Load program
```

The arguments to be specified include *process name*, *program file name*, and *priority*. The process class for all loaded processes is *application*. As an alternative to specifying two distinct names, you may wish to derive the process name from the program name. For example, specifying the name `TEST1` as an argument may signify that the program is to be loaded from file `TEST1.MPX` (which should be present in the appropriate directory), and also that the process created to execute this program should be named `TEST1`.

The new process should initially be placed in the *suspended-ready* state. A *Resume Process* command will be required to ready each new process for actual execution.

The memory size, load address, and execute address in the PCB must now be set to suitable values. The size is returned by `sys_check_program`; the load address is determined by `sys_alloc_mem`; and the execute address is computed from the load address and the offset also returned by `sys_check_program`.

Be certain that the initial context (register contents) for the program is properly initialized. In particular the `CS` and `IP` must be set to values derived from the execute address, which is a far pointer, using the `FP_SEG` and `FP_OFF` functions. The `DS` and `ES` should be initialized to their current values in MPX; these will be reset as needed by the code generated by Turbo C. The `SS` and `SP` are recorded in the PCB in the form of the stack pointer; these values will be set by the dispatcher when the program executes.

An appropriate error message should be displayed if the arguments are not valid; if no PCB can be allocated; if the program file is not found or is invalid; if the program has already been loaded; if sufficient program memory cannot be allocated, or if the process name duplicates an existing one.

Terminate Process

This command should terminate a process by deallocating its PCB and releasing its allocated program memory. This is an extension of the *Delete PCB* command of Module R2. The

-

PCB should be deallocated as in *Delete PCB*. In addition, the program memory must be released using the `sys_free_mem` support procedure.

The only argument for this command is the process name. An error message should be displayed if the specified process does not exist.

Dispatch

This is a *temporary* command which should dispatch the processes in the ready queue in round-robin order. Its operation is very similar to *Dispatch* of Module R3, except that the PCBs are no longer set up by this command. Instead, they should be set up as each program is loaded. This command replaces any previous *Dispatch* command.

The dispatching procedure should use the *sys_call* and *dispatch* procedures developed in Module R3. If these procedures were correctly designed, there should be *no changes* required for this Module.

R4.4 SUPPORT SOFTWARE

Your software for Module R-4 will make use of the support procedures described for Modules R-1 and R-3, plus two additional routines. `Sys_init` and `sys_exit` will of course still be used, invoked as usual by your main program, with `MODULE_R4` as the appropriate `sys_init` parameter. `Sys_alloc_mem` and `sys_free_mem` may be used as before for PCB or queue element allocation; these routines *must* be used for program memory allocation. `Sys_set_vec` will be used essentially as in Module R3; however, it should now be called once only, immediately after `sys_init`, rather than within the *Dispatch* command.

As usual, the remaining support procedures described in Module R1 will continue to be used by the commands that were implemented for that module.

The new support procedures are `sys_check_program` and `sys_load_program`. Their descriptions are given in this section. As before, all support procedures are found in the supplied C library file `MPX_SUPT.C`, and their defining prototypes, along with other necessary definitions, are in the file `MPX_SUPT.H`.

sys_check_program

The `sys_check_program` procedure attempts to access a specified program file to determine if it exists and if it is valid. The file specified must have the MS-DOS `EXE` format. If the file is valid, the procedure returns two important items of information: the amount of memory which must be allocated to load the file, and the location (relative to the start of the load area) at which execution should begin.

The prototype for `sys_check_program` is:

```
int sys_check_program (char dir_name[],
char prog_name[], int *prog_len_p,
int *start_offset_p);
```

-

The meaning of the parameters is as follows:

`dir_name:` specifies the pathname for the directory to be searched for the specified file. Only one directory will be searched. The pathname may be absolute or relative, and may contain a drive specifier; if it is null, the current directory will be searched.

`prog_name:` specifies the file name for the program to be checked. This name should be given with no extension. The file is assumed to have the extension `.MPX` (although its type is `EXE`).

`prog_len_p:` address of a variable of type `int` which will receive the program length in bytes. This specifies the exact number of bytes which must be allocated for loading this program (including code, data and stack segments). The MS-DOS loader requires that the allocated region be paragraph-aligned.

`start_offset_p:` address of a variable of type `int` to receive the offset in bytes from the start of the load area at which execution should begin. This is assumed to be the first location in the code segment.

The returned value will be zero if no problem occurred; otherwise it will be an error code. Possible error codes are:

`ERR_SUP_NAMLNG`	pathname is too long
`ERR_SUP_FILNFD`	file not found
`ERR_SUP_FILINV`	file is not valid

The symbols for these codes are defined in `MPX_SUPT.H`.

sys_load_program

The `sys_load_program` procedure is called to request the actual loading of a program from a valid (`EXE`) program file. The procedure makes use of the built-in relocating loader of MS-DOS. The file is expected to have already been checked and measured using `sys_check_program`. However, `sys_load_program` calls `sys_check_program` again to verify the file size.

The prototype for `sys_load_program` is:

```
int sys_load_program (void *load_addr, int max_size,
        char dir_name[],char prog_name[]);
```

The meaning of the parameters is as follows:

load_addr: specifies the starting address of the memory region into which the program should be loaded. This is normally the address returned by `sys_alloc_mem`.

-

This address must be aligned on a paragraph boundary; all blocks allocated by `sys_alloc_mem` meet this criterion.

max_size: specifies the size of the available memory region. An error will be signaled if the program exceeds this size. In this case no loading will occur.

dir_name: specifies the pathname for the directory to be searched for the specified file. Only one directory will be searched. The pathname may be absolute or relative, and it may optionally include a drive specifier; if it is null, the current directory will be searched.

prog_name: Specifies the file name for the program to be loaded. This name should be given with no extension. The file is assumed to have the extension `.MPX` (although its type is `EXE`).

The returned value will be zero if no problem occurred; otherwise it will be an error code. Possible error codes are:

ERR_SUP_NAMLNG	pathname is too long
ERR_SUP_FILNFD	file not found
ERR_SUP_FILINV	file is not valid
ERR_SUP_PROGSZ	program too big for region
ERR_SUP_LDADDR	invalid load address (*e.g.*, not aligned)
ERR_SUP_LDFAIL	load failed

The symbols for these codes are defined in MPX_SUPT.H.

Test Processes

Five executable program files are supplied with the support software for testing the commands of Module R4. These files have the names `PROC1.MPX` through `PROC5.MPX`. They are explained in detail in the next section.

R4.5 TESTING AND DEMONSTRATION

The test procedure for Module R4 requires the loading, unloading, and dispatching of a set of test processes (program files). The files to be used are named `PROC1.MPX` through `PROC5.MPX`, and are supplied with your support software. These are the only program files which should be used for Module R4, and they are intended for use only with this module. Each process requests termination (*i.e.*, unloading) at the end of its execution.

The following is a suggested testing procedure:

1. Use the *Load Program* command to load one process (for example, `PROC1`).

2. Use the *Show* commands from Module R2 to verify that a PCB exists for this process and that its contents are correct.

-

3. Load and Terminate several programs, and verify that the PCB's are correct. Demonstrate operations from Module R2 such as *Suspend*, *Resume*, and *Set Priority*.

4. Load and resume a single process, and invoke the *Dispatch* command. The process should be dispatched exactly twice. Each process makes two system calls using *int 60h*. The first is an *idle* function, and the second is an *exit* request. If correctly dispatched, the process will display two lines such as:

   ```
   Proc1 has been dispatched. Proc1 is requesting termination.
   ```

   If the termination request is not recognized, an error message will be displayed such as

   ```
   Proc1 dispatched after termination!
   ```

   In this case the process may repeat execution indefinitely.

5. Load all five processes, resume and dispatch. Each process should display two messages in interleaved order. At the end of execution all processes should terminate. Use *Show all* to verify that no processes are present.

6. Reload processes in various orders and with various priorities. Resume and dispatch. Verify that they are dispatched in priority order, and for processes of equal priority, in order of loading.

7. Reload all processes and use *Suspend* and *Resume* prior to dispatch to show that only *ready, not suspended* processes are dispatched.

R4.6 DOCUMENTATION

Your two manuals should continue to be maintained for this module. The operation of the *Load-Program* and *Terminate-Process* commands should be described in your *User's Manual*, and the internal organization of your loading procedures should be documented in the *Programmer's Manual*. The *Dispatch* command is still temporary and need not be included in the manuals.

R4.7 OPTIONAL FEATURES

The project as described relies on system support routines to access the program files. This simplifies life for the implementor, but shields you from many of the details.

For a greater understanding of the structure of program files you may wish to substitute your own analysis of the file structure for the `sys_check_program` routine. This structure is documented in several web sites such as

http://www.delorie.com/djgpp/doc/exe/

-

Note that Windows has used several more advanced versions of this format. MPX works only with the basic `EXE` used by MS-DOS.

A command could be added to provide a complete display of the characteristics of a given program file.

You may also wish to consider the feasibility of writing your own loader, and of accepting other loadable file types such as `.COM` files.

A valuable extension would be to provide support for more general programs, including those generated directly by Turbo C, as suggested at the end of Section R4.2. These programs must invoke MPX system services either using `sys_req` (in which case they must be linked with `MPX_SUPT.C`) or directly via *int60h* (using the Turbo C `intdos` function). Unless they are very short they must invoke MPX services periodically.

A related extension would be to provide a return mechanism on the stack for processes which terminate without calling the *exit* function. This return must produce the equivalent effect as calling `sys_req` with the *exit* operation code. In this case both assembler programs and Turbo C functions would be freed from the requirement of calling *exit*.

Your instructor will advise you if credit will be offered for any of these extensions.

R4.8 HINTS AND SUGGESTIONS

The most likely causes of difficulty with this module are incorrect memory allocation, or stack manipulation errors left over from Module R3. The Turbo C debugger may be used effectively to monitor operation unless a serious error has occurred.

Verify the *Load Program* command carefully before attempting to dispatch the loaded programs. Use the debugger to stop the program just before calling `sys_load_program`. Check that the load address pointer is correct and properly aligned. Clear the content of the allocated load area (or the first and last locations) and check to ensure that they are properly filled.

If the program "hangs" or fails to terminate properly, double-check the stack contents at critical points. In most cases `_SS` and `_SP` may be inspected with the debugger, or `printf` statements may be used to display their contents, if a reasonable stack is present. A useful check is to verify that the stack pointer is identical when it should be before and after a call that seems to be causing difficulties.

A final area of difficulty may come from incorrect data types or type conversion. In particular, `_SP` and other 16-bit register pseudovariables have the type *unsigned int*. It is sometimes necessary to convert between `_SP` and pointer types. If this is done carelessly, or if signed or long values are used, the `SP` may receive a completely unexpected value.

-