MPX–OS Technical Manual

B. Burwell and A. Morash

Fall 2013

Contents

1 Overview of the Program **1**

2 Program Structure **3**

2.1 Processes . . . 3

2.1.1 Types . . . 3

2.1.2 Prioritization . . . 3

2.1.3 States . . . 3

2.1.4 Suspension . . . 4

2.2 System Call (`sys_call`) . . . 4

2.3 Dispatcher (`dispatch`) . . . 4

2.4 Clock Operations . . . 4

2.5 I/O Drivers . . . 5

2.5.1 COM . . . 5

2.5.2 Printer . . . 6

2.6 Scheduling . . . 7

3 Data Structures **9**

3.1 Process Control Block . . . 9

3.1.1 Ready Queue . . . 10

3.1.2 I/O Queue . . . 10

3.2 Device Control Block . . . 11

3.2.1 Ring Buffer . . . 12

4 Source Code **13**

4.1 `mpx.h` . . . 13

4.2 `clock.c` . . . 18

4.3 `com.c` . . . 22

4.4 `comhan.c` . . . 28

4.5 `dcb.c` . . . 42

4.6 `direct.c` 44
4.7 `io.c` 45
4.8 `load.c` 50
4.9 `main.c` 53
4.10 `pcb.c` 56
4.11 `printer.c` 61
4.12 `sys_req.c` 64
4.13 `sys_sppt.c` 65

Chapter 1

Overview of the Program

The MPX Operating System is designed as an exercise to examine various concepts within the theory of operating systems (rather than as a consumer-usable system). It is specially constructed to be started from within MS-DOS, at which time it will "take over" the operating system to perform many of its tasks. As MS-DOS is interrupt-driven, it uses interrupt handler routines to perform many low-level services, such as performing I/O, keeping time using the CPU clock, etc. Thus, we can replace the functionality of the interrupt handlers with code we have written by modifying the interrupt vector table.

MPX is written in C. The following files are used:

1. `mpx.h` — A C header file containing all of the functions implemented, global variables, and `#DEFINE`d macros.

2. `clock.c` — Functions for enabling, configuring, and disabling the clock interrupt, as well as the function that is referenced in the interrupt vector table that is called when a clock interrupt occurs.

3. `com.c` — Functions for enabling, configuring, and disabling the `COM` interrupts, as well as the function that is referenced in the interrupt vector table and is called when a clock interrupt occurs.

4. `comhan.c` — The code for the command handler, which is the main system process that prints a prompt and gets a user-inputted command.

5. `dcb.c` — Functions for dealing with Device Control Blocks (see page 11).

6. `direct.c` — Functions for accessing MS-DOS directory listings. This code was provided by the project.

7. `io.c` — Functions for managing the queueing of processes with regard to I/O operations.

8. `main.c` — Contains `main()`, which is called to perform setup tasks when MPX-OS first boots.

9. `pcb.c` — Functions for dealing with Process Control Blocks (see page 9), including their queueing operations.

10. `printer.c` — Functions for enabling, configuring, and disabling the printer interrupts, as well as the function that is referenced in the interrupt vector table and is called when a clock interrupt occurs.

11. `sys_req.c` — Contains the code for generating a software interrupt for when a system request is made from within MPX. This code was provided by the project.

12. `sys_sppt.c` — System support, including initializing devices, closing devices, and handling system calls.

Chapter 2

Program Structure

2.1 Processes

2.1.1 Types

Processes in MPX-OS can be one of two types: application or system processes. Application processes are any process that is loaded by a user. System processes are defined by MPX-OS for internal use; a user cannot load or run a system process. Precisely, MPX-OS defines two system processes, the first of which being the command handler. This process accepts and parses user commands to the operating system. The second system process is the idle process. This process does nothing, it only ensures that when the command handler is engaged in an I/O wait and is therefore blocked, there is something that can be running.

2.1.2 Prioritization

Each process has a priority. This allows more important processes, such as the command handler, to get more CPU time than less important ones, such as the idle process. A user can specify the priority of a process when it is loaded. Application processes can have priority p_a where $-126 \leq p_a \leq 126$, and system processes can have priority p_s where $-128 \leq p_s \leq 127$.

2.1.3 States

There are a number of states each process can be in, including `READY`, `RUNNING`, `BLOCKED`, and `ZOMBIE`. If a process is ready, it can be started immediately. It will then transition to the running state until it executes a system call. If it requests I/O, the process is blocked until the operation completes. If the user requests termination of the process while an I/O operation is in progress, the process will be placed in the zombie state. The I/O will complete, but immediately upon completion the process will be terminated. If the process requests termination, on the other hand, it is immediately terminated, the memory deallocated, and the PCB freed.

2.1.4 Suspension

A process can also be suspended, which is essentially pausing it. This is a user action; the user can put processes into and take them out of the suspended state arbitrarily. A suspended process will remain in the queue but will be skipped over when it is reached by the dispatcher.

2.2 System Call (`sys_call`)

This function is called whenever one of MPX's processes is requesting a service, such as an I/O operation or termination. It is an interrupt function (its type is `void interrupt`). The function is responsible for examining the parameters that have been placed on the stack, determining the corresponding operation, and executing it.

2.3 Dispatcher (`dispatch`)

The dispatcher is responsible for retrieving the next queued process and running it. It is also an `interrupt` function. This means that the values of all of the registers will be pushed onto the stack before the function begins, and the registers are popped back off when it is finished.

2.4 Clock Operations

The clock interrupt functions in `clock.c` are used to manage the system clock. When MPX-OS boots, it opens the clock for usage by MPX. A global `long` is used to maintain the absolute number of clock ticks, which are configured to occur 18.2 times every second.

The interrupt handler simply increments the counter variable. There is also a function for reading the clock value which takes pointers to `int`s for storing the number of hours, minutes, and seconds that make up the number of ticks.

It is of note that the clock will reset to zero when 24 hours are reached. Though partially for aesthetic reasons, it also ensures we do not overflow the capacity of a `long` when running MPX-OS for an extended period of time.

2.5 I/O Drivers

2.5.1 COM

The functionality for communicating with the `com` port is contained in `com.c`, which has the following functions defined:

`com_open`

If the COM port is already open, `-2` will be returned. This function initializes the COM port by setting the clock rate's Most Significant Byte (MSB) and Least Significant Byte (LSB) correctly in the Line Control Register. The interrupt vector table is modified such that our interrupt handler will be called when a clock interrupt occurs.

The function also needs to ensure that interrupts are enabled, now that they will be generated and handled properly. To do this, it sets bit 4 of the Interrupt Mask Register (IMR) to be zero. Bit 3 of the Modem Control Register is set to enable the MCR to receive interrupts from the UART. The Interrupt Enable Register is set to specify which serial interrupts should be enabled.

Finally, the Device Control Block (page 11) is initialized with the event flag.

`com_close`

To close the COM port, we simply restore pointer to the the MS-DOS interrupt handler in the interrupt vector table.

`com_int`

This is the function that handles COM interrupts. It saves the Base Pointer to return to later, and checks the IIR to see what the interrupt was from and calls either `com_write_int` or `com_read_int`, whichever is appropriate.

`com_read`

This function initializes a read operation from the COM port with the buffer and length pointers. Since these will be from other processes, they must be declared as `far` pointers.

`com_write`

This function initializes a write operation from the COM port, similarly to how `com_read` works.

`com_read_int`

This function is called when the COM port sends a character to the operating system. It is appended to the program buffer. If it was the last character for the buffer (determined by the `length` pointer), a `1` is returned indicating that the I/O operation has finished. Otherwise, a `0` is returned.

`com_write_int`

Similarly, this function is called when the COM port indicates it is ready to receive a character to be written. It takes the next character from the buffer and puts it into the holding register to be printed to the terminal.

2.5.2 Printer

The functionality for interfacing with the printer is stored in `printer.c`, which contains the following functions:

`prt_open`

The printer is initialized by setting up the function pointer in the interrupt vector table, enabling printer interrupts on the 8259, clearing the init bit and setting the select bit on the Printer Control Register, and initializing the Device Control Block.

`prt_write`

To write to the printer, we save the buffer and length into the DCB, enable printer interrupts on PCR, strobing the printer, writing null to the Printer Data Register, and unstrobing the printer.

When the printer is finished printing the null character, it will generate an interrupt to signal that it is ready to print the next character from the buffer.

`prt_close`

To close the printer, we disable printer interrupts using the Interrupt Mask Register and restore the MS-DOS interrupt vector.

`prt_int`

This is the function called when the printer is ready to receive a character. The base pointer is saved. If there are more characters to write, the printer is strobed, the next character from the buffer copied into the Printer Data Register, and the printer is unstrobed.

If there are no more characters to write, the printer interrupts are disabled and the event flag is set. We then call `IO_complete`, passing the identifier of the printer device and the saved base pointer.

The End of Interrupt is sent.

2.6 Scheduling

In MPX-OS, a Round-Robin dispatcher is implemented. There are several functions that handle scheduling tasks based on processes' pending I/O requests. All of this functionality is stored in `io.c`, which implements the following:

`IO_sup`

The far pointers are made to the buffer and length. Then, the proper command based on the op number and op type is called. For example, if the op number refers to COM and the op type is write, `com_write` is called with the new far pointers. If the request was for a write operation to the console, we insert the process back into the ready queue, since this is not an interrupt-driven operation.

`IO_complete`

The DCB corresponding to the requested device is obtained. The stack pointer of `cop` is set as the saved value. If the process was in the blocked state (as opposed to zombie if its termination had been requested by the user during the I/O operation), its state is set back to ready. In either case, it is placed back in the ready queue.

If there are any pending I/O requests for the device that just completed, the processes requesting I/O are scheduled by calling `IO_sched`. Finally, the End of Interrupt signal is sent, and the dispatcher is called.

`IO_sched`

The operation number and type are determined and the appropriate DCB is accessed. If it has an I/O operation in progress, the process is added to the DCB queue and removed from the ready queue. Otherwise (if there is no operation in progress), the currently operating PCB for the DCB is set to the requesting process.

Chapter 3

Data Structures

3.1 Process Control Block

The operating system stores information about processes in a Process Control Block (PCB). Defined in `mpx.h`, the PCB stores the following information:

1. A pointer to the next PCB in the chain.
2. A pointer to the previous PCB in the queue.
3. A pointer to the next PCB in the queue.
4. A 9-character array to store the process name.[1]
5. A type (application or system).
6. A state (ready, running, or blocked).
7. A "suspended" flag.
8. A stack pointer for the process's stack.
9. An array of size `STACK_SIZE` to store the stack.
10. The address at which the process is loaded in memory.
11. The amount of memory allocated for the process.

[1]Note that thus, the maximum length of a process name is 8 characters, since character 9 will be `0x0`.

12. A pointer to a parameter structure, storing the operation number and operation type for the process's current IO operation (if any).

The actual code for the structure is listed here:

```
struct pcbstruct {
    struct pcbstruct * chain;
    struct pcbstruct * next;
    struct pcbstruct * prev;
    char name[9];
    short type;
    short priority;
    short state;
    short suspend;
    unsigned stack_ptr;
    unsigned stack[STACK_SIZE];
    unsigned loadaddr;
    parm *parm_add;
    int mem_size;
};
```

Each process control block is linked to the next one (except for the last one, whose next PCB is `NULL`) by a pointer for iteration purposes. Process control blocks can reside in at most one of several system-wide queues at any given time.

3.1.1 Ready Queue

The ready queue is maintained as a priority-ordered list of processes that are ready to be run. This way, the dispatcher knows which process to start next. Queue operations stored in `pcb.c` use the `next` and `prev` fields of the PCB to create a doubly-linked list and is prioritized using the PCB's `priority` field. By definition, the priority of an application process must be greater than the minimum priority and smaller than the maxiumum priority of a system process. In this way, we can assign the idle process to have the lowest possible priority and the command handler to have the highest possible priority.

3.1.2 I/O Queue

The I/O queue stores queue of processes involved in I/O operations. This queue is a normal FIFO queue, contrasted with the Ready Queue.

3.2 Device Control Block

In order to manage I/O devices, MPX-OS uses Device Control Blocks, or DCBs. The `struct` for the DCB is defined in `mpx.h`, and is accessible to all component programs.

The following information is stored:

1. The current operation.
2. A pointer to the event flag used to signal completion.
3. A pointer to the PCB currently using the device.
4. A pointer to the head of the queue of PCBs waiting to use the device.
5. A far pointer to an int to store the length of the device buffer. The far pointer allows it to point to a different section of memory, since the variable will be declared in a user program.
6. A far pointer to a char array to store the I/O buffer.
7. A count of the number of characters in the buffer.
8. A char array to act as the ring buffer, implemented as a circular queue (see below).
9. The index of the front element of the ring buffer.
10. The index of the rear element of the ring buffer.
11. The number of characters in the ring buffer.
12. The maximum size of the ring buffer.

The code for the DCB structure is listed below:

```
struct dcb_struct {
  unsigned current_op;
  unsigned * event_flag;
  pcb * current_pcb;
  pcb * pcb_head;
  far int * length;
  far char * buffer;
  int count;
  far char * c_buffer;
  char ring[INPUT_BUFFER_MAX];
  int ring_front;
  int ring_rear;
  int ring_count;
  int ring_size;
};
```

3.2.1 Ring Buffer

The ring buffer is used to implement type-ahead; that is, inputted characters are stored by the DCB when there is no current read operation and inserted into the buffer when a read request is made. There are several variables in the DCB structure used to implement the circular queue, whose logic is located in `dcb.c`

Chapter 4

Source Code

4.1 mpx.h

```
/*
 *    file:   mpx.h
 *
 *    Header file for the MPX Operating System.
 *
 *    This file contains constant, structure and function
 *    prototypes for the MPX Operating System
 */

/* MPX System request numbers. */

#define EXIT_CODE 0 /* Process requesting termination. code. */
#define CON  1      /* The console device - keyboard & monitor. */
#define PRT  2      /* The printer device - LPT1.           */
#define COM  3      /* The serial port - COM1.              */

#define DEBUG_PRINTS 0 // 1 for print out debugging statements

/* MPX System request types. */

#define READ   0  /* Read from device. */
#define WRITE  1  /* Write to device. */
#define WAIT   2  /* Semaphore P operation for device. */
#define SIGNAL 3 /* Semaphore V operation for device. */


#define MAXSIZE 20   /* Size of the directory array. */
```

```
struct dirstruct {      /* Data type for a directory entry.    */
       char dirnam[9];      /* The name of a .mpx file.     */
       int  dirsiz;         /* The size of the file (in bytes). */
};

typedef struct dirstruct dir; /* Use dir as the data typer name. */

/**
 * parm is a struct to keep track of parameters passed to sys_req
 */
struct parmstruct {
  int op_number;
  int op_type;
  char *buffer;
  int *length;
};
typedef struct parmstruct parm;

/**
 * PCB
 */

#define FREE        0
#define SYS_PROCESS 1
#define APP_PROCESS 2
#define READY       0
#define RUNNING     1
#define BLOCKED     2
#define ZOMBIE      3
#define NOT_SUSPENDED 0
#define SUSPENDED   1
#define STACK_SIZE 900

struct pcbstruct {
       struct pcbstruct * chain;
       struct pcbstruct * next;
       struct pcbstruct * prev;
       char name[9];
       short type;
       short priority;
       short state;
       short suspend;
       unsigned stack_ptr;
       unsigned stack[STACK_SIZE];
       unsigned loadaddr;
       parm *parm_add;
       int mem_size;
};
typedef struct pcbstruct pcb;

#define INPUT_BUFFER_MAX 40
#define NO_OP 0
#define READ_OP 1
#define WRITE_OP 2

#define MCR 0x3fc
#define IER 0x3f9
#define LCR 0x3fb
#define RBR 0x3f8
#define THR 0x3f8
#define BRDR_LSB 0x3f8
#define BRDR_MSB 0x3f9
#define IIR 0x3fa

#define IMR 0x21                 /* Port address of IMR   */
#define CMR 0x43                 /* Command mode register */
#define IRQ0 0x01                /* Mask for Timer        */
#define CLOCK_ENABLE (0xff - IRQ0) /* Mask to clear Timer bit */
#define CLOCK_DISABLE (0x00 + 1) /* Mask to set Timer bit */

/* Device Control Block */
struct dcb_struct {
  unsigned current_op;
  unsigned * event_flag;
  pcb * current_pcb;
  pcb * pcb_head;
  far int * length;
  far char * buffer;
  int count;
  far char * c_buffer;
  char ring[INPUT_BUFFER_MAX];
  int ring_front;
  int ring_rear;
  int ring_count;
  int ring_size;
};

typedef struct dcb_struct dcb;
```

```
/* Function prototypes. */

/* main.c */
int main(void);

/* comhan.c */
void comhan(void);                /* The MPX/OS command handler. */
int get_cmd(char args[]);
void cmd_version(void);
void cmd_date(char *[]);
void cmd_directory(void);
int cmd_stop(void);
void cmd_help(char *[]);
void cmd_prompt(char *[]);
void cmd_alias(char *[]);
void cmd_show(char *[]);
//void cmd_allocate(char *[]);
//void cmd_free(char *[]);

void cmd_load(char *[]);
void cmd_resume(char *[]);
void cmd_run(char *[]);
void cmd_suspend(char *[]);
void cmd_terminate(char *[]);
void cmd_setpri(char *[]);
//void cmd_dispatch();
void cmd_clock(char *[]);

void sys_req(int,int,char *,int *); /* MPX system request function.
    */
int  directory(dir *, int);        /* Support function to load the */
                                      /* directory array.
                                         */

/**
 * PCB.C
 */
pcb * search_pcb(pcb *, char[]);
pcb * get_pcb(pcb *);
int free_pcb(pcb *, pcb *);
int build_pcb(pcb *, char[] /*name*/, int /*type*/, int /*state*/,
       int /*suspend*/, int /*priority*/, unsigned /*_cs*/,
           unsigned /*_ip*/,
       unsigned /*_ds*/, unsigned /*psw*/);
int insert_pcb(pcb**, pcb *, int);
int remove_pcb(pcb**, pcb *);

/**
 * DCB.c
 */
void dcb_enqueue(dcb*, char);
char dcb_dequeue(dcb*);
void dcb_init(dcb*);
/**
 * Sys_sppt.c
 */
void sys_inti(void);
void sys_exit(void);
void clock_open(void);
void clock_close(void);
void stop_clock(void);
void start_clock(void);
int  set_clock(int, int, int);
void read_clock(int*, int*, int*);
void interrupt dispatch(void);
void interrupt sys_call(void);
void interrupt clock_int(void);

/**
 * Load.c
 */
int load(unsigned *,char []);

/**
 * Com.c
 */
int com_open(int *, int);
void com_close(void);
void interrupt com_int(void);
int com_read(char far *, int far *);
int com_write(char far *, int far *);
int com_read_int();
int com_write_int();

/**
 * Printer.c
 */
#define PCR 0x3be
#define PDR 0x3bc
```

```
int prt_open(int *);
int prt_write(char far *, int far *);
int prt_close(void);
void interrupt prt_int(void);


int IO_complete(int, int*);
int IO_sched(pcb *);
void IO_sup(pcb *);

/*
 *   Global variable EXTERN directives.
 *
 *      These extern declarations allow the variables to be
 *      accessed from any source code file which includes
 *      this header file. The memory space for the variables
 *      is declared in a *.c file.
 */
#define DIR_SIZE 20
extern dcb com;
extern dcb prt;
extern dcb con;
extern dir direct[]; /* Array of directory entries - see direct.c */
extern int directory(dir *direct, int dir_size);
extern pcb * pcb_list;
extern pcb * ready_queue_locked;
extern pcb * io_init_queue_locked;
extern pcb * cop; /* The currently operating process. */
extern unsigned sys_stack[];
extern unsigned sp_save; /* So that mod 4 can return to
    cmd_dispatch */
extern int prt_eflag;
extern int con_eflag;
extern int com_eflag;
```

4.2 clock.c

```
/*
 * Clock support for MPX-OS
 * Ben Burwell & Averill Morash
 * CSI350 - Fall 2013
 */
```

```
#include <dos.h>
#include "mpx.h"

void interrupt (*vect08)();
unsigned long clock;      /* The clock counter    */

/*
 * Set up MXP clock
 */
void clock_open() {
       unsigned char imr;

       disable();
       /* set up interrupt vector for timer interrupts */
       vect08 = getvect(0x08);
       setvect(0x08,&clock_int);

       // set the timer mode
       outportb(CMR, 0x36);
       outportb(0x40, 0);
       outportb(0x40, 0);

       // set the count clock
       clock = 0L;

       // enable timer interrupts
       imr = inportb(IMR);     // Get current IMR
       imr = imr & CLOCK_ENABLE; // Clear timer bit
       outportb(IMR, imr);     // Set new IMR

       enable();
}

/*
 * Restore the MS DOS clock
 */
void clock_close() {
  disable();
  setvect(0x08,vect08);
  enable();
}

void interrupt clock_int() {
  disable();
  // if it's been 24 hours, reset clock to 0
  if (clock >= 1572462L) {
        clock = 0L;
  } else {
        clock++;
  }
  outportb(0x20, 0x20);
  enable();
}

void stop_clock() {
  unsigned char imr;
  disable();

  imr = inportb(IMR);      // Get the old imr
  imr = imr | CLOCK_DISABLE; // Disable timer interrupts
  outportb(IMR, imr);      // Set the new imr

  enable();
}

/*
 * Enables the timer interrupt
 */
void start_clock() {
  unsigned char imr;
  disable();

  imr = inportb(IMR);      // Get the old imr
  imr = imr & CLOCK_ENABLE; // Enable timer interrupts
  outportb(IMR, imr);      // Set the new imr

  enable();
}

/*
 * Sets the value of the clock
 */
int set_clock(int hr, int m, int s) {
  unsigned long ticks;

  // Validate input
  if (hr > 23 || hr < 0 || m > 59 || m < 0 || s > 59 || s < 0) {
        return -1;
  }
  // set clock
  ticks = 0L;
  ticks  = ((unsigned long)hr * 3600L * 91L / 5L);
  ticks += ((unsigned long)m * 60L * 91L / 5L);
  ticks += ((unsigned long)s      * 91L / 5L);

  disable();
  clock = ticks;
  enable();

  return 0;
}

/*
 * Gets the value of the clock
 */
void read_clock(int *hr, int *m, int *s) {
  unsigned long ticks;
  long total_seconds = 0L;
  int l_hr = 0;
  int l_m = 0; //local vars for hr, m

  disable();
  ticks = clock;
  enable();

  total_seconds = ticks * 10L;
  total_seconds = total_seconds / 182L;

  //count the whole hours
  while (total_seconds >= 3600L) {
       l_hr++;
       total_seconds = total_seconds - 3600L;
  }

  //count the whole minutes
  while (total_seconds >= 60L) {
       l_m++;
       total_seconds = total_seconds - 60L;
  }

  *hr = l_hr;
  *m = l_m;
  *s = total_seconds;
```

```
}
```

4.3 com.c

```
/**
 * MPX OS
 * CSI350 Operating Systems
 * Fall 2013
 * Averill Morash, Ben Burwell
 *
 * File: com.c
 */

#include <dos.h>
#include <stdio.h>
#include <string.h>
#include "mpx.h";


void interrupt (*vect0c)();

int com_opened;

// Open the com port
int com_open(int * flag, int rate) {
  unsigned char imr;
  unsigned char mcr;
  unsigned char ier;
  unsigned char lcr;

  disable();

  // Ensure com has not already been opened, if
  // it has, we must return.
  if (com_opened == NULL || com_opened == 0) {
        com_opened = 1;
  } else {
        return -2;
  }

  // Modify the interrupt vector table
  vect0c = getvect(0x0c);
```

```
setvect(0x0c, &com_int);

// set baud rate
// first we need to set bit 7 of LCR
// to be 1 so that we can set the baud
// rate divisor.
lcr = inportb(LCR);
lcr = lcr | 0x80;
outportb(LCR, lcr);

// Now we can access the divisor.
// Clock rate is 1.8432 MHz. We want
// 1200 bps/baud.
// 1843200 / (1200 * 16) = 96 = 0x0060
// So the MSB is 0x00 and the LSB is 0x60

// Ensure baud rate is valid
if (rate != 1200) {
      return -3;
}

outport(BRDR_MSB, 0x00);
outportb(BRDR_LSB, 0x60);

// Now set the LCR
lcr = inportb(LCR);

// b7, b3, b2 = 0
lcr = lcr & 0x73;

// b0, b1 = 1
lcr = lcr | 0x03;

outportb(LCR, lcr);

// set IMR to allow COM interrupts
// set b4 = 0
imr = inportb(IMR);
imr = imr & 0xef;
outportb(IMR, imr);

// set MCR to enable interrupts from
// the UART
// on MCR, b3 = 1
mcr = inportb(MCR);
```

```
  mcr = mcr | 0x08;
  outportb(MCR, mcr);

  // use IER to indicate which serial
  // interrupts are enabled
  // b0 = 1, b1 = 0
  ier = inportb(IER);
  ier = ier | 0x01;
  ier = ier & 0xfd;
  outportb(IER, ier);

  // set up the DCB
  com.event_flag = flag;
  dcb_init(&com);

  // enable interrupts
  enable();

  return 0;
}

// Com Close
void com_close() {
  disable();

  // restore MS-DOS com vector
  setvect(0x0c, vect0c);

  enable();
}

// Interrupt handler
void interrupt com_int() {
  int iir;
  int *lst_stk;

  // 0 indicates not finished, 1 indicated finshed
  int ret = 0;

  disable();

  // save base pointer
  lst_stk = _BP;

  // check IIR to see what caused the interrupt
  iir = inportb(IIR);
  // only look at bits 1, 2
  iir = iir & 0x06; // 0000-0110

  if (iir == 0x02) { // 0000-0010
       ret = com_write_int();
  } else if (iir == 0x04) { // 0000-0100
       ret = com_read_int();
  } else {
       // interrupt was not read or write
       return;
  }

  // if the operation is done, call IO_complete
  if (ret == 1) {
       IO_complete(COM, lst_stk);
  }

  // write end of interrupt to 8259
  outportb(0x20, 0x20);

  enable();
  return;

}

// Read from the com port
int com_read(char far * buffer, int far * length) {

  char dq;
  int i;

  disable();

  // if we haven't yet opened com, we can't proceed
  if (&com == NULL) {
       enable();
       return -1;
  }

  // check for invalid length
  if (*length < 1) {
       enable();
       return -3;
  }
  // check if DCB is free
  if (com.current_op != NO_OP) {
       // previous IO not yet complete
       enable();
       return -2;
  }

  if (buffer == NULL || length == NULL) {
       printf("error in com_read\n");
       enable();
       return;
  }
  // set up the DCB
  com.buffer = buffer;
  com.length = length;
  com.current_op = READ_OP;
  com.count = 0;

  // clear the buffer
  strcpy(buffer, "");

  // now we need to use the ring buffer to place typed-ahead
  // characters into the buffer

  // keep track of how many characters are copied from the
  // ring buffer.
  i = 0;

  while (com.ring_count > 0 && strlen(buffer) < *length) {
       dq = dcb_dequeue(&com);
       buffer[i] = dq;
       i++;
  }

  // place a null character at the end of the buffer
  buffer[i] = '\0';
  com.count = i;

  enable();
  return 0;
}

int com_write(char far * buffer, int far * length) {
  int ier;
  disable();
  // check that com has been initialized
  if (&com == NULL) {
        enable();
        return -1;
  }

  // check that com is not busy
  if (com.current_op != NO_OP) {
        enable();
        return -2;
  }

  // check that the length is valid
  if (*length < 1) {
        enable();
        return -3;
  }

  // initialize DCB
  com.buffer = buffer;
  com.length = length;
  com.current_op = WRITE_OP;
  com.count = 0;

  // enable THR interrupts
  ier = inportb(IER);
  // bit 1 should be 1
  ier = ier | 0x02;
  outportb(IER, ier);

  enable();
  return 0;
}

int com_read_int() {
  char rbr;

  rbr = inportb(RBR);

  if (com.current_op == READ_OP && com.count < *(com.length)) {
          if (rbr == 0x0d || com.count == (*(com.length) - 1)) {
                com.buffer[com.count] = '\0';
                com.count++;
                *com.length = com.count;
                com.current_op = NO_OP;
                *(com.event_flag) = 1;
                return 1;
           } else {
                com.buffer[com.count] = rbr;
                com.count++;
           }
  } else {
        dcb_enqueue(&com, rbr);
  }

  return 0;
}

int com_write_int() {
  int ier;

  // if there are more characters to write
  if (*(com.length) > com.count) {
        // write a character
        outportb(THR, com.buffer[com.count]);
        com.count++;
        return 0;
  } else {
        // no chars to write
        // disable THR interrupts
        ier = inportb(IER);
        ier = ier & 0xfd;
        outportb(IER, ier);

        // set event flag
        com.current_op = NO_OP;
        *(com.event_flag) = 1;
        return 1;
  }
}
```

4.4 comhan.c

```
/*
 *   file:  comhan.c
 *
 *   This file is the command handler for the
 *   MPX operating system.
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <dos.h>
#include "mpx.h"

#define BUF_SIZE 80    /* Size of the command line buffer. */

#define VERSION      0
#define DATE         1
#define DIRECTORY    2
#define STOP         3
#define HELP         4
#define PROMPT       5
#define ALIAS        6
#define SHOW         7
#define CMD_LOAD     8
#define CMD_RESUME   9
#define CMD_RUN     10
#define CMD_SUSPEND 11
#define CMD_TERMINATE 12
#define CMD_SETPRI  13
#define CLOCK    14

#define NUM_CMDS 15

int length = 1;       /* Length of the command line.   */
unsigned sp_save;     /* A stack save for mod 4 dispatch. */

char prompt[20] = "mpx>";
const char version[] = "MPX-OS 2013 ZOMBIE Edition
    (v.1010372365424) \n";
char date[] = "01/09/1991";

char *cmds[] = { "version", "date", "directory", "stop",
                              "help", "prompt", "alias", "show",
                              "load", "resume",
                              "run", "suspend", "terminate",
                                  "setpriority",
                             "clock", NULL};
char *aliases[] = {"     ", "      ", "      ", "      ",
                               "       ", "       ", "       ", "
                                        ",
                               "       ", "       ", "       ", "
                                        ",
                               "       ", "       ", "       ",
                                   NULL};

/*
 *   comhan()   This is the command handler for the MPX OS.
 *              It repeatedly prints a prompt, makes a system
 *              request to read from the console, and then
 *              carries out the command.
 *
 *   Parameters: None.
 *   Return value: None.
 */

void comhan() {
       static char *args[7]; //one more than expected number of
           arguments
       static char buffer[BUF_SIZE];
       int do_stop = 0;

       // Occasionally the first con_read request recieves an empty
           response
       // Make a dummy request to compensate
       sys_req(CON, READ, buffer, &length);

       do {
         printf("%s ",prompt);          /* Print a prompt.      */
         length = BUF_SIZE;             /* Reset length of buffer.
             */
         sys_req(CON,READ,buffer,&length); /* Request CON input  */

         // For better display, print a newline after user input
         printf("\n");

         set_args(buffer, args);

         switch (get_cmd(args[0])) {
              case VERSION:     cmd_version();      break;
              case DATE:        cmd_date(args);     break;
              case DIRECTORY:   cmd_directory();    break;
              case STOP:        do_stop = cmd_stop(); break;
              case HELP:        cmd_help(args);     break;
              case PROMPT:      cmd_prompt(args);   break;
              case ALIAS:       cmd_alias(args);    break;
              case SHOW:        cmd_show(args);     break;
              case CMD_LOAD:    cmd_load(args);     break;
              case CMD_RESUME:  cmd_resume(args);   break;
              case CMD_RUN:     cmd_run(args);      break;
              case CMD_SUSPEND: cmd_suspend(args);  break;
              case CMD_TERMINATE: cmd_terminate(args); break;
              case CMD_SETPRI:  cmd_setpri(args);   break;
              case CLOCK:       cmd_clock(args);    break;
              default:
                printf("Can't recognize. %s\n", args[0]);
                break;
         }
       } while (!do_stop);
}

int get_cmd(char cmd[]){
  /* return the number associated with a command (use STOP, HELP
      etc) */

  int i = 0; //loop control

  if (cmd == NULL) {
       return -1;
  }

  strlwr(cmd);

  while (cmds[i] != NULL) {
       if (strcmp(cmds[i], cmd)==0 || strcmp(aliases[i], cmd)==0){
         return i;
       }
       i++;
  }

  //default - means it wasn't a valid command
  return -1;
}

int set_args(char buffer[], char *args[]) {
  /* use string tok to set the contents of args from buffer
        and return the number of args (will go into argc) */
  char separators[6] = " /,:="; //Characters that separate tokens
  int i = 0; //loop control

  args[i] = strtok(buffer, separators); //Get first token
  while(args[i] != NULL){
       args[++i] = strtok(NULL, separators); //Get next tokens
  }

  return i;
}
/**
 * Print the version number.
 */
void cmd_version() {
  printf("%s", version);
}

/**
 * Print or change the date
 */
void cmd_date(char *args[]){

  int m, d, y;

  if (strcmp("", args[1])==0) {
       printf("%s \n", date);
  } else {

       m = atoi(args[1]);
       d = atoi(args[2]);
       y = atoi(args[3]);

       if (m > 0 && m < 13 && d > 0 && d < 32) {
         sprintf(date, "%d/%d/%d", m, d, y);
       } else {
         printf("Invalid date. \n");
       }
  }
}
void cmd_directory(){
  int no_proc = directory(direct, DIR_SIZE);
  int i;

  if (no_proc == 1) {
       printf("You have 1 program \n");
  } else if (no_proc == 0) {
        printf("You have no programs \n");
  } else {
        printf("You have %d programs \n", no_proc);
  }

  if (no_proc > 0) {
        printf("Size     Name \n");
        printf("========= =========================== \n");
  }

  for (i = 0; i < no_proc; i++) {
        printf("%8d %s \n", direct[i].dirsiz, direct[i].dirnam);
  }
}

/**
 * Print a goodbye message
 */
int cmd_stop(){
  char buffer[2];
  int length = 2;
  pcb *p = pcb_list;

  printf("Are you sure you want to exit? [y/n]: ");
  sys_req(CON,READ,buffer,&length);

  if (strcmp(buffer, "y") == 0 || strcmp(buffer, "Y") == 0) {
        disable();
        do {
          if(p->type != FREE) {
                freemem(p->loadaddr);
                free_pcb(pcb_list, p);
          }
          p = p->chain;
        } while(p != NULL);
        enable();
        sys_exit();
        printf("** COMHAN execution complete **\n");
        return 1;
  } else {
        return 0;
  }
}
```

```
/**
 * Print information about the COMHAN commands.
 * Will print all commands or just information specific to the
 * argument if given.
 */
void cmd_help(char *args[]) {
  int i;
  char *help[NUM_CMDS];

  help[VERSION]      = "version           Display version number";
  help[HELP]         = "help              Provide information about
      commands";
  help[DIRECTORY]    = "directory         List .mpx files";
  help[DATE]         = "date [mm/dd/yyyy] Display or set the system
      date";
  help[STOP]         = "stop              Terminate execution of
      COMHAN";
  help[PROMPT]       = "prompt string     Change the prompt for
      commands";
  help[ALIAS]        = "alias command=string Create an alias for a
      command";
  help[SHOW]         = "show              Prints PCB information";
  help[CMD_LOAD]     = "load name[=ppp]   Creates a process called
      name with priority ppp";
  help[CMD_RESUME]   = "resume name       Resumes the process called
      name";
  help[CMD_RUN]      = "run name[=ppp]    Runs a process called name
      with priority ppp";
  help[CMD_SUSPEND] = "suspend name      Suspends the process
      called name";
  help[CMD_TERMINATE] = "terminate name   Terminates the process
      called name";
  help[CMD_SETPRI]   = "setpriority name=ppp Sets the priority of
      process name";
  help[CLOCK]        = "clock [stop|start] Perform clock operations";

  if (args[1] != NULL && get_cmd(args[1]) < 0) {
       printf("Not a valid command.\n");
       return;
  }
  // print header
  printf(            "  Name               Use \n");
  printf(            "  ====================
      ================================================= \n");
  if (args[1] != NULL) {
        // print help for specific command
        printf(" %s \n", help[ get_cmd(args[1]) ] );
  } else {
        // list all help
        for (i = 0; i < NUM_CMDS; i++) {
          printf(" %s \n", help[i]);
        }
  }
}

/**
 * Change the prompt.
 */
void cmd_prompt(char *args[]){
  strcpy(prompt, args[1]);
}

void cmd_alias(char *args[]){
  //get the number of the command to alias
  int num = get_cmd(args[1]);
  strcpy(aliases[num], args[2]);
  num ++;
}

void print_pcb(pcb * p) {

        char str[4];

        if (p->type == FREE) {
                str[0] = 'f';
        } else if (p->type == SYS_PROCESS) {
                str[0] = 's';
        } else if (p->type == APP_PROCESS) {
                str[0] = 'a';
        } else {
                str[0] = '-';
        }

        if (p->state == READY) {
                str[1] = 'r';
        } else if (p->state == RUNNING) {
                str[1] = 'o';
        } else if (p->state == BLOCKED) {
                str[1] = 'b';
        } else if (p->state == ZOMBIE) {
                str[1] = 'z';
        } else {
                str[1] = '-';
        }

        if (p->suspend == SUSPENDED) {
                str[2] = 'y';
        } else if (p->suspend == NOT_SUSPENDED) {
                str[2] = 'n';
        } else {
                str[2] = '-';
        }

        str[3] = '\0';

        printf("0x%04x %8s %s 0x%04x 0x%04x 0x%04x %4d 0x%04x \n",
                  p, p->name, str, p->chain, p->prev,
                  p->next, p->priority, p->loadaddr);
}

void cmd_show(char *args[]) {
  pcb * current = pcb_list;

  if (!args[1]) {
        printf("Show: free, all, system, application, \n");
        printf("      suspended, ready, init. \n");
        return;
  }

  printf("PCB_Ad Name    TSP chain prev   next    pri l_addr \n");
  printf("------ -------- --- ------ ------ ------ ---- ------ \n");

  if (strcmp(args[1], "init") == 0 || strcmp(args[1], "ready") ==
      0) {
        disable();
        current = (strcmp(args[1], "init") == 0)?
            io_init_queue_locked : ready_queue_locked;
        enable();

        if (current == NULL) {
          return;
        }

        do {
```

```
        print_pcb(current);
        current = current->next;
      } while (current != NULL);
      return;
  }

  do {
      if (strcmp(args[1], "free") == 0 && current->type == FREE) {
            print_pcb(current);
      } else if (strcmp(args[1], "all") == 0) {
            print_pcb(current);
      } else if (strcmp(args[1], "system") == 0 &&
        current->type == SYS_PROCESS) {
            print_pcb(current);
      } else if (strcmp(args[1], "application") == 0 &&
        current->type == APP_PROCESS) {
            print_pcb(current);
      } else if (strcmp(args[1], "suspended") == 0 &&
        current->suspend == SUSPENDED) {
            print_pcb(current);
      } else if (strcmp(args[1], "ready") == 0 &&
        current->state == READY) {
            print_pcb(current);
      }

      current = current->chain;
  } while (current != NULL);
}

/**
 * Load
 */
void cmd_load(char *args[]) {
  int i, priority = 0;
  unsigned segp;
  pcb *p;
  int paragraphs;
  char* name = args[1];
  // figure out how many directory entries
  int num = directory(direct, DIR_SIZE);

  if (num < 1) {
      printf("Error: no programs available to load. \n");
      return;
  }
```

```
// search in the directory for the specified program
i = 0;
while (stricmp(direct[i].dirnam, name) != 0) {
      i++;
      if (i == num) {
        // no program found
        printf("Error: No program found with that name. \n");
        return;
      }
}

// figure out how much memory we need
paragraphs = (unsigned) ceil(direct[i].dirsiz / 16);

// try to allocate the needed memory
if (allocmem(paragraphs, &segp) != -1) {
      // insufficient memory
      printf("Error: Insufficient memory. \n");
      return;
}

// read the file into memory
if (load((unsigned *)segp, direct[i].dirnam) != 0) {
      // load error
      printf("Error: Could not load program into memory. \n");
      return;
}

// if a valid priority was specified, replace the default
if (args[2] != NULL) {
      if (atoi(args[2]) >= -126 && atoi(args[2]) <= 126) {
        priority = atoi(args[2]);
      } else {
        printf("Warning: invalid priority specified, using
            default. \n");
      }
}

// now we can put the proc into a PCB
p = get_pcb(pcb_list);

if (p == NULL) {
      printf("Error: No free PCBs. \n");
      return;
```

```
  }

  if (build_pcb(p, args[1], APP_PROCESS, READY, SUSPENDED,
        priority, segp, 0 /* ip */, 0 /* ds */, 0x200) != 1) {
        printf("Error: Unable to build PCB. \n");
        return;
  }
  disable();
  insert_pcb(&ready_queue_locked, p, 0);
  enable();
  return;
}

/**
 * Resume
 */
void cmd_resume(char *args[]) {

  pcb *p = pcb_list;

  if (args[1] != NULL) {
        if (strcmp(args[1], "*") == 0) {
          // resume all processes
          // set suspended to NOT_SUSPENDED for all application PCBs
          do {
                if (p->type == APP_PROCESS) {
                  p->suspend = NOT_SUSPENDED;
                }
                p = p->chain;
          } while (p != NULL);
        } else {
          // search for the correct PCB and set suspended to
              NOT_SUSPENDED
          p = search_pcb(pcb_list, args[1]);
          if (p != NULL) {
                p->suspend = NOT_SUSPENDED;
          } else {
                printf("No process named %s. \n", args[1]);
          }
        }
  } else {
        printf("No process specified. Please run resume proc \n");
        printf("for process proc. To resume all processes, \n");
        printf("type resume * \n");
  }
```

```
}

/**
 * Run
 */
void cmd_run(char *args[]) {
  pcb *p;
  // load the process ...
  cmd_load(args);

  // ... then set it to be not suspended
  p = search_pcb(pcb_list, args[1]);
  if (p != NULL){
        p->suspend = NOT_SUSPENDED;
  } else {
        printf("Error: process did not load correctly. \n");
  }
}

/**
 * Suspend
 */
void cmd_suspend(char *args[]) {
  pcb *p;

  // Get the pcb...
  p = search_pcb(pcb_list, args[1]);

  if (p != NULL) {
        // ... and suspend it
        p->suspend = SUSPENDED;
  } else {
        printf("No process with the specified name. \n");
  }
}

/**
 * Terminate
 */
void cmd_terminate(char *args[]) {
   pcb *p;

   // the "terminate *" case
   if (strcmp(args[1], "*") == 0) {
         p = pcb_list;
         do {
           // Set all application processes to ZOMBIE state
           // They will be terminated in dispatch()
           if (p->type == APP_PROCESS) {
                 p->state = ZOMBIE;
           }
           p = p -> chain;
         } while (p != NULL);
   } else {
         // the "terminate proc" case
         p = search_pcb(pcb_list, args[1]);

         if (p != NULL && p->type == APP_PROCESS) {
           p->state = ZOMBIE;
         } else {
           printf("No process with the specified name. \n");
         }
   }
   return;
}

/**
 * Set Priority
 */
void cmd_setpri(char *args[]) {
  pcb *p;
  int new_priority;

  // Get the pcb
  p = search_pcb(pcb_list, args[1]);
  new_priority = atoi(args[2]);

  if (p != NULL) {
        // Check the priority is valid
        if (p->type == APP_PROCESS && new_priority >= -128 &&
            new_priority <= 127) {
          disable();
          remove_pcb(&ready_queue_locked, p);  // Take pcb out of
              ready queue
          p->priority = new_priority;    // Change priority
          insert_pcb(&ready_queue_locked, p, 0); // Re-insert pcb in
              ready queue
          enable();
        } else {
          printf("Error: invalid priority. \n");
```

```
      }
  } else {
      printf("Error: invalid process name. \n");
  }
}

/*
 * stop, start, set, or show the clock based on args[1]
 */
void cmd_clock(char *args[]){
  int hr, m, s;
  int ret;
  // decide what clock operation to perform
  if (strcmp(args[1], "stop") == 0) {
      stop_clock();
      return;
  } else if (strcmp(args[1], "start") == 0) {
      start_clock();
      return;
  } else if (strcmp(args[1], "") == 0) {
      read_clock(&hr, &m, &s);
      printf("The current time is %d:%d:%d\n", hr, m, s);
      return;
  } else {
      // the remaining case is to set the clock
      // parse the input - should be "hh:mm:ss"
      ret = set_clock(atoi(args[1]), atoi(args[2]), atoi(args[3]));
      // call sys_sppt's set_clock(hr, mn, s)
      // if set_clock returns -1 print error message
      if (ret == -1) {
        printf("Error setting clock\n");
      }
  }
}
```

4.5 dcb.c

```
/**
 * MPX OS
 * Operating Systems - Fall 2013
 * Averill Morash, Ben Burwell
 */
```

```
#include "mpx.h"

/**
 * Returns the character at the front of the ring queue
 * associated with the dcb, or '\0' if the queue is empty.
 */
char dcb_dequeue(dcb *d) {
  char c;

  if (d->ring_count == 0) {
        return '\0';
  }

  c = d->ring[d->ring_front];

  if (d->ring_front == d->ring_size - 1) {
        d->ring_front = 0;
  } else {
        d->ring_front++;
  }

  d->ring_count--;

  return c;
}

/**
 * Adds the given character to the end of the ring queue
 * in the given dcb if there is room
 */
void dcb_enqueue(dcb *d, char c) {
  if (d->ring_count >= d->ring_size) {
        return;
  }

  if (d->ring_rear == d->ring_size - 1) {
        d->ring_rear = 0;
  } else {
        d->ring_rear++;
  }

  d->ring[d->ring_rear] = c;

  d->ring_count++;
  return;
}

/**
 * Set up a dcb's circular queue, and set the current_op to NO_OP
 */
void dcb_init(dcb *d) {
  d->ring_front = 0;
  d->ring_rear = -1;
  d->ring_size = INPUT_BUFFER_MAX;
  d->ring_count = 0;
  d->current_op = NO_OP;
  return;
}
```

4.6 direct.c

```
/*
 *   file: direct.c
 *
 *   This file contains the function which reads the names and
 *   sizes of the MPX/OS process files from the disk, and stores
 *   them in the directory entry array.
 */

#include <dos.h>      /* Borland header file */
#include <fcntl.h>    /* Borland header file */
#include <string.h>   /* Borland header file */
#include <dir.h>      /* Borland header file */
#include "mpx.h"

dir direct[MAXSIZE];  /* The array of directory entries. */

/*
 *
 *  directory -  This procedure performs a sequential read of the
    MPX
 *               directory, obtaining all .MPX files up to the size
    of the
 *               directory. These are the file that can be loaded by
 *               COMHAN. Note that the file specific details are
    saved in
 *               the dir structure, direct.
 *
 *   Parameters: direct  - the array of directory entries.
 *               dir_size - the capacity of the direct array.
 *
 *   Return value: The number of files entries made to the
 *                direct array.
 */


int directory(dir *direct, int dir_size)
{
  int   num_procs;     /* Number of .mpx files found.            */
  char  filename[15];  /* Name of a file with .mpx extension.    */
  int   done;          /* Flags when no more .mpx files can be
      found. */
  struct ffblk ffblk;

  num_procs = 0;  /* number of .MPX file entries placed in
      directory */

  done = findfirst ("*.MPX",&ffblk,0);
  while (!done && num_procs < dir_size) {
      strcpy (filename,ffblk.ff_name);
      strcpy(direct->dirnam,filename);
      direct->dirnam[strcspn(filename,".")] = '\0';
      direct->dirsiz = ffblk.ff_fsize;
      ++num_procs;
      direct++;
      done = findnext(&ffblk);
  }

  return(num_procs);
}
```

4.7 io.c

```
/*
 * IO functionality
 * for MPX-OS
 * Ben Burwell & Averill Morash
 * CSI350 - Fall 2013
 */

#include <dos.h>
#include <stdio.h>
#include "mpx.h"

#define DS_OFFSET 6

void IO_sup(pcb *p) {
  char far *buffer; // far pointer to the io buffer
  int far *length; // far pointer to an int representing length of
      the io buffer
  unsigned *ds_add; // data segment address
  int con_w = 0;   // flag for if the request was a con_write
      request
  int rc;

  disable();

  if (DEBUG_PRINTS) { printf("in io_sup \n");}

  //make the far pointers (see page 107)
  ds_add = p->stack_ptr + DS_OFFSET;
  buffer = MK_FP(*ds_add, p->parm_add->buffer);
  length = MK_FP(*ds_add, p->parm_add->length);

  //call com_read or equivalent
  switch (p->parm_add->op_number){
        case COM:
          // IO_sched should have validated input
          // Assume that op_type is either read or write
          if (p->parm_add->op_type == READ) {
                if (DEBUG_PRINTS){printf("com read req \n");}
                com_eflag = 0;
                rc = com_read(buffer, length);
          } else {
                if (DEBUG_PRINTS) {printf("com write req \n");}
                com_eflag = 0;
                rc = com_write(buffer, length);
          }
          break;
        case CON:
          if (p->parm_add->op_type == READ) {
               if (DEBUG_PRINTS){printf("con read req\n");}
               con_eflag = 0;
               con.current_op = READ_OP;
               rc = con_read(buffer, length);
          } else {
               if (DEBUG_PRINTS){printf("con_write req\n");}
               con_w = 1;
               rc = con_write(buffer, length);
          }
          break;
        case PRT:
          if (DEBUG_PRINTS){printf("prt write req\n");}
          prt_eflag = 0;
          rc = prt_write(buffer, length);
          break;
  }

  if (rc < 0) {
        printf("Somthing went wrong in io_sup\n");
  }

  // put the pcb back in the ready queue, if it was a con_write
  if (con_w == 1) {
        insert_pcb(&ready_queue_locked, p,0);
        p->state = READY;
  } else {
        p->state = BLOCKED;
  }
  enable();
  return;
}

// Handle completion of an IO request
int IO_complete(int device, int *stk_ptr) {

  pcb *current;
  dcb *d;

  disable();
  // get the dcb for the requested device
  switch (device){
        case CON:
          d = &con;
          // the con dcb isn't managed by con drive
          // so we need to reset it here, but the
```

```
        // other dcbs are handled in those files
        // before calling io_complete
        d->current_op = NO_OP;
        break;
      case COM:
        d = &com;
        break;
      case PRT:
        d = &prt;
        break;
}

if (DEBUG_PRINTS){printf("in io complete\n");}

// save interrupted process's stack pointer
cop->stack_ptr = stk_ptr;

// insert running process into ready queue
insert_pcb(&ready_queue_locked, cop, 0);

// set running process to ready state
cop->state = READY;

// get address of PCB whose IO just completed
current = d->current_pcb;

// insert into ready queue
insert_pcb(&ready_queue_locked, current, 0);

// set its state to ready, unless it became
// a zombie while it was waiting for io to complete
if (current->state != ZOMBIE) {
      current->state = READY;
}

if (io_init_queue_locked != NULL) {
      // if there are pending IO requests schedule them
      current = io_init_queue_locked;
      while (current != NULL && current->parm_add->op_number !=
          device) {
        current = current->next;
      }

      if (current != NULL) {
        // take the pcb out of the io_init queue
```

```
         remove_pcb(&io_init_queue_locked, current);

         // if it wasn't terminated while waiting for io...
         if (current->state != ZOMBIE) {
               // ... schedule the io
               IO_sched(current);
         } else {
               // send the zombie back to be ... dispatched ...
                   muahaha
               insert_pcb(&ready_queue_locked, current, 0);
         }
       }
  }

  outportb(0x20,0x20); // EOI signal
  enable();
  dispatch();

  return 0;
}

// Service an IO request
// Returns:
//    0 for success
//   -1 for invalid operation
//   -2 for invalid device
int IO_sched(pcb *p) {

  dcb *d;
  int op_num;
  int op_type;

  disable();
  if (DEBUG_PRINTS) {printf("in io sched\n");}

  // determine which device is requested
  op_num = p->parm_add->op_number;
  op_type = p->parm_add->op_type;

  switch (op_num) {
        case CON:
         if (op_type != READ
               && op_type != WRITE) {
                 return -1;
         }
```

```
          d = &con;
          break;
        case COM:
          if (op_type != READ
                && op_type != WRITE) {
                  return -1;
          }
          d = &com;
          break;
        case PRT:
          if (op_type != WRITE) {
                return -1;
          }
          d = &prt;
          break;
        default:
          return -2;
  }

  // If the device is busy
  if (d->current_op != NO_OP) {
        disable();
        // put pcb in IO queue
        insert_pcb(&io_init_queue_locked, p, 1);
        p->state = BLOCKED;
        enable();
  } else {
        // If the device wasn't busy, start the io
        d->current_pcb = p;
        IO_sup(p);
  }
  enable();
  return 0;
}
```

4.8 load.c

```
/***************************************************************************
*
*       Module Name:   load
*
*       Purpose: To provide support for loading files.
```

```
*
*      Functions in Module:
*             load

*****************************************************************/

#include <dos.h>      /* Borland header file */
#include <fcntl.h>    /* Borland header file */
#include <dir.h>      /* Borland header file */

#include "mpx.h"

#define intnum 33     /* int86 number for load only */
#define success 0     /* return code for successful process load */
#define error  -1     /* return code for failure on process load */


/*******************************************************************
 *
 *     Function: load
 *     Abstract: Dynamically loads a file from disk
 *     Algorithm:
 *               This procedure dynamically loads a process with a
 *               file extension of .MPX. The memory address provided
 *               is the segment address at which to load the
    process.
 *               The process with the name in the string pname is
 *               loaded into memory at the segment address found in
 *               load_addr. If load is successful, load returns a 0
 *               return code, otherwise it returns a -1 return code
 *
 *******************************************************************/

int load(unsigned *load_addr,char pname[])
{
   int i,flags,carry;
   char fname[30];

   struct PRMBLK
     {
        int segaddr;
        int reloctn;
     };

   struct PRMBLK prmbk;
```

```
    union REGS *inregs;   /* input registers pointer */
    union REGS *outregs;  /* output registers pointer */
    struct SREGS *segregs; /* segment registers pointer */

    union REGS inr;      /* input register structure */
    union REGS outr;     /* output register structure */
    struct SREGS segr;   /* segment register structure */

    disable();

    /* SEGMENT REGISTER VALUE CALL */

    segregs = &segr;
    segread(segregs);  /* returns current segment register values */

    strcpy(fname, pname);
    strcat(fname, ".MPX");

    inregs = &inr;
    outregs = &outr;
    inregs->h.ah = 75;
    inregs->x.dx = &fname[0];
    prmbk.segaddr = load_addr;
    prmbk.reloctn = load_addr;
    segr.es = segr.ds;       /* assigning the ES to the data
        segment */
    inregs->x.bx = &prmbk;
    inregs->h.al = 3;        /* load only */

    flags = int86x(intnum,inregs,outregs,segregs);
    carry = flags; /* Could be omitted */
    carry = outregs->x.cflag;
    if(carry)
    {
       printf("\nerror code:%3d",outregs->x.ax);
       enable();
       return(error);
    }

    /* load successful */
    enable();
    return(success);
  }

/************************************************************************/
```

4.9 main.c

```
/*  file: main.c
 *
 *  This file contains the function main
 *  for the MPX Operating System - this is where
 *  the program's execution begins
 */

#include <stdio.h>
#include <dos.h>
#include "mpx.h"

pcb * pcb_list;
pcb * ready_queue_locked;
pcb * io_init_queue_locked;
pcb * cop;

pcb pcb0;
pcb pcb1;
pcb pcb2;
pcb pcb3;
pcb pcb4;
pcb pcb5;
pcb pcb6;
pcb pcb7;
pcb pcb8;
pcb pcb9;
pcb pcb10;
pcb pcb11;

dcb com;
dcb prt;
dcb con;
int com_eflag;
int con_eflag;
int prt_eflag;

unsigned sys_stack[STACK_SIZE];

int main(void) {
```

```
pcb * pcb_addr;
dcb * d = &con;//test
char *args[] = {"load", "idle"};

printf("Booting MPX... \n");

sys_init();

if (DEBUG_PRINTS) printf("did init \n");

d->event_flag = &con_eflag;

if (DEBUG_PRINTS) printf("making chain \n");

// create the chain
pcb0.chain = &pcb1;
pcb1.chain = &pcb2;
pcb2.chain = &pcb3;
pcb3.chain = &pcb4;
pcb4.chain = &pcb5;
pcb5.chain = &pcb6;
pcb6.chain = &pcb7;
pcb7.chain = &pcb8;
pcb8.chain = &pcb9;
pcb9.chain = &pcb10;
pcb10.chain = &pcb11;
pcb11.chain = NULL;

if (DEBUG_PRINTS) printf("building pcbs \n");

build_pcb(&pcb0, "      ", FREE, -1, -1, 0, NULL, NULL,
    NULL, NULL);
build_pcb(&pcb1, "      ", FREE, -1, -1, 0, NULL, NULL,
    NULL, NULL);
build_pcb(&pcb2, "      ", FREE, -1, -1, 0, NULL, NULL,
    NULL, NULL);
build_pcb(&pcb3, "      ", FREE, -1, -1, 0, NULL, NULL,
    NULL, NULL);
build_pcb(&pcb4, "      ", FREE, -1, -1, 0, NULL, NULL,
    NULL, NULL);
build_pcb(&pcb5, "      ", FREE, -1, -1, 0, NULL, NULL,
    NULL, NULL);
build_pcb(&pcb6, "      ", FREE, -1, -1, 0, NULL, NULL,
    NULL, NULL);
```

```
build_pcb(&pcb7, "       ", FREE, -1, -1, 0, NULL, NULL,
    NULL, NULL);
build_pcb(&pcb8, "       ", FREE, -1, -1, 0, NULL, NULL,
    NULL, NULL);
build_pcb(&pcb9, "       ", FREE, -1, -1, 0, NULL, NULL,
    NULL, NULL);
build_pcb(&pcb10, "      ", FREE, -1, -1, 0, NULL, NULL,
    NULL, NULL);
build_pcb(&pcb11, "      ", FREE, -1, -1, 0, NULL, NULL,
    NULL, NULL);

pcb_list = &pcb0;

// initialize the currently running process
cop = NULL;

// set up command handler as a PCB
pcb_addr = get_pcb(pcb_list);
build_pcb(pcb_addr, "ComHan", SYS_PROCESS, READY,
    NOT_SUSPENDED, 127, _CS,
  (unsigned)comhan, _DS, 0x200);

disable();
insert_pcb(&ready_queue_locked, pcb_addr, 0);
enable();

if (DEBUG_PRINTS) printf("comhan in queue \n");


// set up idle process
cmd_load(args);

if (DEBUG_PRINTS) printf("loaded \n");

// update priority for idle process
// we can't do this when initially calling because of
// APP_PROCESS type priority validation
pcb_addr = search_pcb(pcb_list, "idle");
pcb_addr->priority = -128;
pcb_addr->suspend = NOT_SUSPENDED;
pcb_addr->type = SYS_PROCESS;

if (DEBUG_PRINTS) printf("pri set \n");

// ALREADY IN QUEUE THANKS TO CMD_LOAD
```

```
	//insert_pcb(&ready_queue, pcb_addr, 0);

	if (DEBUG_PRINTS) printf("idle in queue \n");

	dispatch();

	sys_exit();

	return 0;
}
```

4.10 pcb.c

```
#include "mpx.h"
#include <string.h>
#include <stdio.h>
#include <dos.h>

#define STK_PSW   (STACK_SIZE - 1)
#define STK_CS    (STACK_SIZE - 2)
#define STK_IP    (STACK_SIZE - 3)
#define STK_DS    (STACK_SIZE - 9)
#define INIT_STACK (STACK_SIZE - 12)

/**
 * Returns the address of the PCB with the specified name
 */
pcb * search_pcb(pcb * list, char name[]) {
  pcb * current = list;

  // step through the chain
  do {
	if (stricmp(current->name, name) == 0) {
	  return current;
	}
	current = current->chain;
  } while (current != NULL);

  // nothing found
  return NULL;
}
```

```
/**
 * Returns the address of the first free PCB.
 * If all PCBs are in use, it returns NULL.
 */
pcb * get_pcb(pcb * list) {
  pcb * current = list;

  // step through the list
  do {
        if (current->type == FREE) {
          return current;
        }
        current = current->chain;
  } while (current != NULL);

  // no free PCBs
  return NULL;
}

/**
 * Sets the specified PCB to be free.
 */
int free_pcb(pcb * list, pcb * addr) {
  pcb * current = list;

  do {
        if (current == addr) {

          // check if PCB is already free
          if (current->type == FREE) {
                // it is, so we don't free it
                return -2;
          } else {
                // we need to free the PCB
                build_pcb(current, "     ", 0, -1, -1, 0, NULL,
                    NULL, NULL, NULL);
                return 1;
          }
        }
        current = current->chain;
  } while (current != NULL);

  // invalid PCB address
  return -1;
}
```

```
/**
 * Sets the properties of the given PCB.
 */
int build_pcb(pcb * addr, char name[], int type, int state,
                        int suspend, int priority, unsigned cs,
                            unsigned ip,
                        unsigned ds, unsigned psw) {
  // check that address is valid
  if (addr == 0 || addr == NULL) {
       return -1;
  }

  // check that type is FREE, SYS_PROCESS, APP_PROCESS
  if (type != FREE && type != SYS_PROCESS && type != APP_PROCESS) {
       return -2;
  }

  // check that state is READY, RUNNING, BLOCKED, or -1 for free
  if (state != READY && state != RUNNING &&
         state != BLOCKED && state != -1) {
       return -3;
  }

  // check that suspend is SUSPENDED, NOT_SUSPENDED or -1 for free
  if (suspend != SUSPENDED && suspend != NOT_SUSPENDED && suspend
      != -1) {
       return -4;
  }

  strcpy(addr->name, name);
  addr->type = type;
  addr->state = state;
  addr->suspend = suspend;
  addr->next = NULL;
  addr->prev = NULL;
  addr->priority = priority;

  // set up the stack
  addr->stack[STK_PSW] = psw;
  addr->stack[STK_CS] = cs;
  addr->stack[STK_IP] = ip;
  addr->stack[STK_DS] = ds;

  addr->stack_ptr =(unsigned) &(addr->stack[INIT_STACK]);
  addr->loadaddr = cs;
  return 1;
}

/**
 * Puts the specified PCB into the specified queue, using either
 * FIFO or priority queue (method==0: priority, method==1: FIFO)
 */
int insert_pcb(pcb **queue, pcb * addr, int method) {
  pcb * current = *queue;
  pcb * one_after;


  // if there's nothing in the queue yet, make the PCB the start
  if (current == NULL) {
        *queue = addr;
        return 1;
  }

  if (method == 1) {
        // insert at end of queue

        // skip to the end of the queue
        while(current->next != NULL) {
          current = current->next;
        }

        // add the PCB
        current->next = addr;
        addr->next = NULL;
        addr->prev = current;
        return 1;
  } else if (method == 0) {
        // insert in priority order

        // if we need to skip forward at all...
        if (current->priority >= addr->priority) {

          // ...do it
          while (current->next != NULL &&
                        (current->next)->priority >= addr->priority) {
                          current = current->next;
          }

          // and set the pointers
```

```
         one_after = current->next;
         current->next = addr;
         addr->next = one_after;
         addr->prev = current;
         if (one_after != NULL) {
              one_after->prev = addr;
         }
       } else {

         // otherwise, put it at the head
         current->prev = addr;
         addr->next = current;
         *queue = addr;
       }
       return 1;
  } else {
       // There was a problem, return error code
       return -1;
  }
}

/**
 * Takes the PCB out of the specified queue
 */
int remove_pcb(pcb **queue, pcb * addr) {
  pcb * current = *queue;

  if (addr == NULL || *queue == NULL) {
       printf("bad times ahead \n");
       return;
  }

  // are we removing the head?
  if (addr == *queue) {
       *queue = addr->next;
       if (addr->next != NULL) {
         (addr->next)->prev = NULL;
         addr->next = NULL;
       }
       return 0;
  }

  // not removing the head:
  do {
       if (current->next == addr) {
         current->next = addr->next;
         if (current->next != NULL) {
              (current->next)->prev = current;
         }
         addr->next = NULL;
         addr->prev = NULL;
         return 0;
       }
       current = current->next;
 } while (current != NULL);
 return -1;
}
```

4.11 printer.c

```
/**
 *  Operating Systems
 *  Mod 6
 *  Averill and Ben
 *  Writing awesome printer stuff.
 */

#include <dos.h>
#include <stdio.h>
#include "mpx.h";

void interrupt (*vect0f)();

int prt_opened;

int prt_open(int * prt_flag) {
  unsigned char imr;

  disable();

  if (prt_opened == NULL || prt_opened == 0) {
        prt_opened = 1;
  } else {
        return -2;
  }

  // set up vector interrupt
  vect0f = getvect(0x0f);
  setvect(0x0f, &prt_int);

  // enable printer interrupts on 8259
  imr = inportb(IMR);
  imr = imr & 0x7f; //0111-1111
  outportb(IMR, imr);

  // clear init bit (b2), set select bit (b3)
  // 0000 1010
  outportb(PCR, 0x0a);

  // initialize a printer control block
  prt.current_op = NO_OP;
  prt.event_flag = prt_flag;
  prt.count = 0;

  enable();
  return 0;
}

int prt_write(char far *buffer, int far *length) {

  disable();
  // check that prt isn't busy
  if (prt.current_op != NO_OP) {
       return -2;
  }

  // check that length is valid
  if (*length < 1) {
       return -3;
  }

  //save buffer and length in prt dcb
  prt.buffer = buffer;
  prt.length = length;
  prt.current_op = WRITE_OP;
  prt.count = 0;

  // enable printer interrupts on PCR
  // 0001 1010
  outportb(PCR, 0x1a);

  // strobe printer
  // 0001 1011
  outportb(PCR, 0x1d);

  // print a null
  // NOTE: something is not right here because when i print
      something
  // other than a null it still doesn't print
  outportb(PDR, 0x00);

  // unstrobe printer
  // 0001 1010
  outportb(PCR, 0x1c);

  enable();
  return 0;
}

int prt_close() {
  unsigned char imr;

  disable();

  // reset printer
  // set init bit (b2) to 1 all else clear
  // 0000 0100
  outportb(PCR, 0x04);

  //disable printer interrupts
  imr = inportb(IMR);
  imr = imr | 0x80;
  outportb(IMR, imr);

  // restore ms-dos's printer interrupt vector
  setvect(0x0f, vect0f);

  enable();
  return 0;
}

void interrupt prt_int() {
  unsigned char pcr;
  int *lst_stk;

  disable();
  lst_stk = _BP;
  // if no more char to write
  if (prt.count == *(prt.length)) {
        // disable printer interrupts at PCR
        // clear b4 on PCR
        pcr = inportb(PCR);
        pcr = pcr & 0xef; // 1110-1111
        outportb(PCR, pcr);

        // set event flag
        prt.current_op = NO_OP;
        *(prt.event_flag) = 1;

        // call io-complete
        IO_complete(PRT, lst_stk);

        // send end of interrupt signal
        outportb(0x20, 0x20);
  } else {

        // strobe printer
        // 0001 1101
        outportb(PCR, 0x1d);

        // write char to PDR
        outportb(PDR, prt.buffer[prt.count]);
        prt.count++;

        // unstrobe printer
        // 0001 1100
        outportb(PCR, 0x1c);
  }

  // send EOI
  outportb(0x20, 0x20);
  enable();
}
```

4.12 sys_req.c

```
/***********************************************************************
*
```

```
*       Procedure Name: sys_req
*
*
*       Purpose: System request for round robin dispatcher test.
*
*       Sample Call: sys_req(op_num,op_type,&buffer,&len)
*                   where
*                         op_num specifies the operation requested
*                         op_type specifies read or write
*                         buffer is the data buffer
*                         length is the number of characters to read
    or write
*
*       Algorithm: Request to sys_call is made via an int 60.
    Parameters
*                  passed to sys_req remain on the stack for use by
    sys_call.
*
*************************************************************************/


#include <dos.h>

extern void interrupt sys_call();


void sys_req(int op_number,int op_type,char *buff_add,int *length)
{
  geninterrupt(0x60); /* If all is set up right, this should */
}                     /* invoke sys_call() to handle the request. */
```

4.13 sys_sppt.c

```
/************************************************************************
*
*       Name:   sys_sppt.c
*
*       Purpose: Support Routines for Modules 3 and 4 of MPX-PC
*
*       Sample Call:
*                    sys_init()
*                    sys_exit()
```

```
*
*      Procedures In Module:
*                    sys_init - Sets up interrupt vector 60 for
    MPX-PC
*                    sys_exit - Resets interrupt vector 60 and exits
    to DOS
*
*
*****************************************************************************/

#include <stdio.h>
#include <dos.h>
#include "mpx.h"

void interrupt (*vect60)(); /* Storage for DOS int 60h */
                                                    /* interrupt
                                                       vector.    */

void sys_init()
{
       /* set up interrupt vector for MPX sys_calls */
       vect60 = getvect(0x60);
       setvect(0x60,&sys_call);

       // set up the clock
       clock_open();

       // set up com
       com_open(&com_eflag, 1200);

       // set up printer
       prt_open(&prt_eflag);

       // set up con
       con_open(&con_eflag);
}


void sys_exit()
{
  disable();
  // restore the clock
  clock_close();

  // close IO devices
  com_close();
  con_close();
  prt_close();

  /* restore interrupt vector 60 and exit */
  setvect(0x60,vect60);

  enable();
  exit();
}

// dispatch the process at the head of the ready queue
void interrupt dispatch()
{
 disable();
 if (DEBUG_PRINTS) {printf("in dispatch \n ");}

 cop = ready_queue_locked;

 // kill zombie processes
 while (cop->state == ZOMBIE) {
   remove_pcb(&ready_queue_locked, cop);
   freemem(cop->loadaddr);
   free_pcb(pcb_list, cop);
   cop = ready_queue_locked;
 }

 // skip over suspended processes
 while (cop != NULL && cop->suspend == SUSPENDED) {
   cop = cop -> next;
 }

 if (DEBUG_PRINTS){
   if (cop == NULL) {
         printf("!!cop null?!!!\n");
   } else {
         printf("cop - %s\n", cop->name);
   }
 }

 remove_pcb(&ready_queue_locked, cop);
 cop->state = RUNNING;

 enable();
 _SP = cop -> stack_ptr;
```

```
}


void interrupt sys_call()
{
   static parm *parm_add;

   if (DEBUG_PRINTS) {printf("in sys_call \n ");}

        /* Save stack pointer for current process */
        cop->stack_ptr = _SP;

        /* Get address of sys_call parameters */
        parm_add = _SP + 0x1c; // parameter offset = 0x1C

        /* Note that you should save this parameter address
           somewhere in the pcb as suggested below*/
        cop->parm_add = parm_add;

        _SP = &sys_stack[STACK_SIZE-1];

        // find out if the process wants to die... and kill it
        if (parm_add->op_number == EXIT_CODE) {
          freemem(cop->loadaddr);
          free_pcb(pcb_list, cop);
          // NOTE: the process just made this sys_req so it cannot
              have any io
        } else {
          IO_sched(cop);
        }

        dispatch();
}
```

Index

Device Control Block, 11

PCB, 9
Process Control Block, 9