Design of a 32-bit Multi-Tasking Operating System
(With Real Time Keyboard Interrupt Acknowledgement)

By

Anirban Sinha (University Roll:XXXXXXXX)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INSTITUTE OF ENGINEERING AND MANAGEMENT

AN HONORS THESIS
Submitted in partial fulfillment of the of the requirements of
Degree of Bachelor of Technology (B.Tech) in
Computer Science and Engineering the under the affiliation of
Kalyani University

Under the guidance of

Prof. B.K.Dutta

Project Guide and HOD
Department of Computer Science and Engineering
Institute of Engineering and Management

INSTITUTE OF ENGINEERING AND MANAGEMENT
Y-12, BLOCK – EP, SECTOR V, SALT LAKE ELECTRONICS COMPLEX
KOLKATA – 700091, WEST BENGAL, INDIA

May 2002

Acknowledgements

We are grateful to our beloved instructor, Prof Bimal Kumar Dutta who is also the head of the department, Computer Engineering, IEM for help, advice and suggestions in our endeavor.
We are also grateful to all those people who have published their work in this subject on the Internet from where we got immense help and guidance. This work would not have been successful if they had not shared their immense and valuable suggestions and experiences with all others working in this field.
Special thanks go to Christopher Giese (geezer@execpc.com) for his work on the protected mode tutorials and his guidance in the midst of our work.

Abstract

This work deals with the development of the kernel and boot loader of an operating system so that the machine can boot into a basic user environment. The user can input some specific commands for the execution by the system. Provisions are made so that more than one process can run simultaneously on different virtual consoles so that multitasking can be achieved.

System Requirements

SOFTWARE RESOURCES:

The software resources to build the kernel include DJGPP: GNU C compiler for windows, NASM: Netwide Assembler, CM32 Compiler, decompilers etc.

HARDWARE RESOURCES:

Hardware resources for running the system include a 32 bit 80386 or above Intel based Microprocessor, standard keyboard, display device, standard floppy drive controller (optional).

Contents

Chapter 1 Introduction

Developing operating system kernels is not a new task. There are several sample operating system kernels available on the Internet designed by many amateur developers. Many people have worked in this area before and many more will work in the future. However, since at the honors level, we are not supposed to implement a novel idea in a final year project, this work seemed appropriate enough. The reasons for this are many. Some of them are enumerated below.

Firstly, no one in this institute had previously taken this venture of designing an operating system from scratch. The work seemed to be challenging enough since developing an operating system required a lot of depth in understanding the x86 architecture, the operating system modules and functions along with good skills in C and assembly programming at low level. We wanted to take this challenge as this would give us the perfect opportunity to acquire strong foundation in basic areas of computer science.

Secondly, the knowledge acquired by working through this project would help us not only in testing the operating system concepts hands on but also help us in pursuing and exploring higher and more complicated areas of computer science research, basis of which lay in the areas we explore in this project.

Thirdly, this project would also give us an idea about the design and development of embedded systems which is one of the primary areas of interest in the industry.

Lastly, through this project, if successful, we would get the fun & satisfaction of creating something new and something of our very own.

Thus undertaking this project was absolutely relevant in the areas of our study. The entire work took approximately ten months from submitting the proposal to delivery of this final report. Due to shortage of time available for final year projects (less than one year) & strict deadlines, we could not ultimately implement many important features which are essential for a fully blown operating system. However, regardless of this important drawback, we found it extremely satisfying and useful working through this project and being able to design a fully working operating system kernel.

The project has been divided into two parts, namely the Booting phase and development of the Kernel which are described in succeeding chapters of this thesis.

Chapter 2 Booting Phase

This phase is further divided into two main functional sections:

1. Booting the computer and
2. Setting the environment to load the kernel into the extended memory.

2.1 Plain Booting Phase (Stage 1 of two stage booting)

In the plain booting phase we assure the following: A valid boot sector has the code 0xAA55 at an offset of 510 at the very first sector of the disk. Therefore just after POST operation BIOS simply checks the corresponding location of drive 0 for the code (depending upon the CMOS booting sequence settings specified). If a valid boot sector is found then it is loaded at location 0:07C0H in main memory. We wrote our own boot sector, assembled it into a flat binary file using NASM and loaded it into the first sector (sector 0, track 0) of the disk. We wrote our own C program to write the boot sector to the first sector of the disk. This was done using BIOS interrupt 0x13 and service number Ah=02.

The C program that writes the boot sector plain binary file to sector 0 track 0 is given below:

```
//***************START***************

#include <bios.h>
#include <stdio.h>

void main()
{
        FILE *in;
        unsigned char buffer[520];

        if((in = fopen("bootsect", "rb"))==NULL)
        {
                printf("Error loading file\n");
                exit(0);
        }

        fread(&buffer, 512, 1, in);
```

```
        while(biosdisk(3, 0, 0, 0, 1, 1, buffer));

        fclose(in);
}

//*************END***************************
```

During this phase itself, the boot sector code at first checks the processor. Since our kernel supports Intel 80386 and above processor, any lower version of the processor brings the system to a halt by the initial bootstrap code. If a valid processor is found, we land up in our second stage of our two-stage boot loader, one that performs the environment initialization process.

2.2 Environment Setting Phase (Stage 2 of two stage booting)

Things to be done before the kernel is loaded in memory are:

- Write Operating System's own Interrupt Service Routines.
- Switch to 32 bit protected mode.
- Enable A20 line of the keyboard controller chip so as to enable XMS memory.
- Reprogram the 8259 Interrupt controller to accommodate OS-defined interrupts.
- Load the kernel to the predefined location in XMS-memory from the Disk

2.2.1 Writing ISR 's:

Interrupt handlers can be written exactly in the same way as all other codes are written. For example, somewhere in our assembly code we write:

```
isr30:     pusha
           push gs
           push fs
           push es
           push ds
```

```
… … … … … … …
(actual code for the handler)
… … … … … … …
pop ds
pop es
pop fs
pop gs
popa
iret
```

Here we first save the contents of all the registers that we are going to use in our ISR and then we put the actual code for the service routine.

Now in the Interrupt Descriptor Table entry we point one of the descriptors to this ISR in the following way: ~

```
idt:        dw isr30

            dw SYS_CODE_SEL
            db 0
            db 0x8E
            dw 0
```

The descriptors are also written in the same way as we write any other assembly code, but all descriptors MUST be written in the contiguous fashion.

Some static data structures must be initialized at this time.(viz. GDT, IDT etc). They are done by loading the GDTR and IDTR with appropriate values and pointing them to the section of the code that holds these tables.

2.2.2 Switching to Protected Mode

Before switching to protected mode, a minimum set of system data structures and code modules must be loaded into memory. We have shown how that is being done. Once these tables are created, software initialization code can switch into protected mode.

Executing a MOV CR0 instruction that sets the PE flag in the CR0 register enables the protected mode. (In the same instruction, the PG flag in register CR0 can be set to enable paging.)

Execution in protected mode begins with a CPL of 0.

The 32-bit Intel Architecture processors have slightly different requirements for switching to protected mode. To insure upwards and downwards code compatibility with all 32-bit Intel Architecture processors, it is recommended that the following steps be performed:

1. Disable interrupts. A CLI instruction disables maskable hardware interrupts. NMI interrupts can be disabled with external circuitry. (Software must guarantee that no exceptions or interrupts are generated during the mode switching operation).
2. Execute the LGDT instruction to load the GDTR register with the base address of the GDT.
3. Execute a MOV CR0 instruction that sets the PE flag (and optionally the PG flag) in control register CR0.
4. Immediately following the MOV CR0 instruction, execute a far JMP or far CALL instruction. (This operation is typically a far jump or call to the next instruction in the instruction stream). The JMP or CALL instruction immediately after the MOV CR0 instruction changes the flow of execution and serializes the processor. If paging is enabled, the code for the MOV CR0 instruction and the JMP or CALL instruction must come from a page that is identity mapped (that is, the linear address before the jump is the same as the physical address after paging and protected mode is enabled). The target instruction for the JMP or CALL instruction does not need to be identity mapped.
5. If a local descriptor table is going to be used, execute the LLDT instruction to load the segment selector for the LDT in the LDTR register.
6. Execute the LTR instruction to load the task register with a segment selector to the initial protected-mode task or to a writable area of memory that can be used to store TSS information on a task switch.
7. After entering protected mode, the segment registers continue to hold the contents they had in real-address mode. The JMP or CALL instruction in step 4 resets the CS register.

Perform one of the following operations to update the contents of the remaining segment registers: -

- Reload segment registers DS, SS, ES, FS, and GS. If the ES, FS, and/or GS registers are not going to be used, load them with a null selector.
- Perform a JMP or CALL instruction to a new task, which automatically resets the values of the segment registers and branches to a new code segment.

8. Execute the LIDT instruction to load the IDTR register with the address and limit of the protected-mode IDT.
9. Execute the STI instruction to enable maskable hardware interrupts and perform the necessary hardware operation to enable NMI interrupts.

2.2.3 Enabling the Extended Memory

When the AT was introduced, it was able to access up to sixteen megabytes of memory, but in order to remain compatible with the IBM-XT, memory wraparound had to be duplicated in the AT so things would work the same The 20th address line on the bus (A20) was turned off so this "wrap-around" effect worked and software from the old XT days continued to work The A20 line is controlled by the keyboard controller unit. This is usually a derivative of the 8042 chips.

After enabling the A20 line of 8042, we can access memory beyond the 1 Megabyte boundary. This memory over and above the 1 MB limit is known as the Extended memory or the XMS memory.

We have to enable the A20 line to prevent memory wraparound and to use more than 1 MB of memory in protected mode. To enable the A20 line, some hardware IO is used using the Keyboard Controller chip (8042 chip) and enable it. We have to enable the A20 line to prevent memory wraparound and to use more than 1 MB of memory in protected mode. The basic algorithm is given below:-

1. *Disable Interrupts*
2. *Wait until the keyboard buffer is empty*
3. *Send command to disable the keyboard*
4. *Wait until the keyboard buffer is empty*
5. *Send command to read output port*
6. *Wait until the keyboard buffer is empty*
7. *Save byte from input port*

8. *Wait until the keyboard buffer is empty*
9. *Send command Write output port*
10. *Wait until the keyboard buffer is empty*
11. *Send saved byte OR by 2 (GATE A20 to ON)*
12. *Wait until the keyboard buffer is empty*
13. *Enable the keyboard*
14. *Enable interrupts*

And the corresponding C program is: -

```
void init_A20(void)
{
        UCHAR a;
        disable_ints();
        kyb_wait_until_done();
        kyb_send_command(0xAD);         // disable keyboard
        kyb_wait_until_done();
        kyb_send_command(0xD0);         // Read from input
        kyb_wait_until_done();
        a=kyb_get_data();
        kyb_wait_until_done();
        kyb_send_command(0xD1);         // Write to output
        kyb_wait_until_done();
        kyb_send_data(a|2);
        kyb_wait_until_done();
        kyb_send_command(0xAE);         // enable keyboard
        enable_ints();
}
```

The A20 line is now on.

2.2.4 Reprogramming 8259 - The basic idea

The 8259A is a device specifically designed for use in real time, interrupt driven microcomputer systems. It manages eight levels or requests and has built-in features for expandability to other 8259A's (up to 64 levels). It is programmed by the system's software as an I/O peripheral.

The 8259A accepts two types of command words generated by the CPU:

1. Initialization Command Words (ICWs): Before normal operation can begin, each 8259A in the system must be brought to a starting point–by a sequence of 2 to 4 bytes timed by WR pulses.

2. Operation Command Words (OCWs): These are the command words which command the 8259A to operate in various interrupt modes. These modes are:

- Fully nested mode
- Rotating priority mode
- Special mask mode
- Polled mode

The OCWs can be written into the 8259A anytime after initialization.

8259 can be reprogrammed to accommodate additional interrupts that are written when an operating system is initialized. These are the interrupts supported only by the present OS.

Each 8259 supports 8 interrupt request levels that generates call to 8 locations in memory that are spaced equally apart , spaced either 4 memory locations or 8 memory locations away from each other. The starting address, call address interval, mode of operation (single or cascaded) and level/edge triggered operating mode will have to be specified using the Initialization Code Word-1 and Initialization Code Word-2.

If there are two 8259's used in cascading modes, the master/slave configuration is specified using the Initialization Code Word-3. If required, Initialization Code Word-4 may be used to fine tune 8259 further.

Next, we will have to enable IRQ's at these interrupts using the Operational Code Word-1.

Lets say we want to enable interrupts 0x20 to 0x2F (INT 0x20-INT 0x2F) and let us suppose we will use two 8259's in cascaded mode, master will handle interrupts 0x20 to 0x27 and the slave handles interrupts 0x28 to 0x2F. Here is the example sample code:

```
mov al, 0x11
out 0x20,al
out 0xA0, al

mov al, 0x20
out 0x21,al
add al,8
out 0xA1,al
mov al,4
out 0x21,al
mov al,2
```

```
    out 0xA1,al
    mov al,1
    out 0x21,al
    out 0xA1,al
    mov al,0xFE
    out 0x21,al
    mov al, 0xFF
    out 0xA1,al
```

2.2.5 Loading the Kernel

The main task after the proper environment for the kernel has been set up is to load the kernel from disk into the main memory and begin execution of the kernel.

This is accomplished by using the BIOS interrupt 13H. This interrupt first resets the Disk controller chip and then reads a specified number of bytes from the specified sector, head, track of either the hard disk or the floppy disk after positioning the head over the specified location.

One thing is carefully considered. Before we begin, we must first know the exact size of the kernel ".bin" file and the exact physical position of the disk where it is written. Also we must fix the physical address in the main memory where the kernel will be loaded.

Another thing that is of utmost importance is that as we read the kernel binary file, we must recalculate the position of the read/write head every time a block of binary data is read. This calculation has to be done manually since the Interrupt gives the facility of only reading a fixed size of data from the disk location specified. Also we must increment the physical address of the memory location to which the block of binary data is being written, the increment being equal to the size of the block being read.

Lastly, some provisions must be made in case there is some failure in reading the data from the disk. A fixed number of attempts are made to read the data from the specified location and if it fails within this specified number of attempts, the disk is considered to be bad and appropriate message must be written to the output device.

Chapter 3 Kernel Development Phase

The development of the kernel takes place module wise. The main modules call the subordinate modules and they in turn call their dependent modules. Each of the modules once called becomes a “terminate and stay resident” program until the user issues reboot command where the memory is cleared and machine reboots. Initially, an assembly level bootstrap program performs some initial checking functions and then calls the main C program and freezes.

3.1 Kernel bootstrapping code

The main bootstrapping of the kernel is done in assembly. From within this code, other modules and functions are invoked. The assembly code is given below.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; 32-bit kernel startup code
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
%include "asm.inc"

SECTION .text
BITS 32
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; entry point
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

GLOBAL entry
entry:
; check if data segment linked, located, and loaded properly
      mov eax,[ds_magic]
      cmp eax,DS_MAGIC
      je ds_ok

; display a blinking white-on-blue 'D' and freeze
```

```
      mov word [0B8000h],9F44h
      jmp short $
ds_ok:
IMP main
      call main               ; call C code
      jmp $                   ; freeze
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; name:                 getvect
; action:         reads interrupt vector
; in:             [EBP + 12] = vector number
; out:                  vector stored at address given by [EBP + 8]
; modifies:       EAX, EDX
; minimum CPU:          '386+
; notes:          C prototype:
;                 typedef struct
;                 {     unsigned access_byte, eip; } vector_t;
;                 getvect(vector_t *v, unsigned vect_num);
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

EXP getvect
      push ebp
            mov ebp,esp
            push esi
            push ebx
                  mov esi,[ebp + 8]

; get access byte from IDT[i]
                  xor ebx,ebx
                  mov bl,[ebp + 12]
                  shl ebx,3
                  mov al,[idt + ebx + 5]
                  mov [esi + 0],eax

; get handler address from stub
                  mov eax,isr1
                  sub eax,isr0      ; assume stub size < 256 bytes
```

```
                mul byte [ebp + 12]
                mov ebx,eax
                add ebx,isr0
                mov eax,[ebx + (isr0.1 - isr0 + 1)]
                mov [esi + 4],eax
          pop ebx
          pop esi
     pop ebp
     ret
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; name:               setvect
; action:         writes interrupt vector
; in:             [EBP + 12] = vector number,
;                 vector stored at address given by [EBP + 8]
; out:                (nothing)
; modifies:       EAX, EDX
; minimum CPU:        '386+
; notes:          C prototype:
;                 typedef struct
;                 {     unsigned access_byte, eip; } vector_t;
;                 getvect(vector_t *v, unsigned vect_num);
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

EXP setvect
     push ebp
          mov ebp,esp
          push esi
          push ebx
                mov esi,[ebp + 8]

; store access byte in IDT[i]
                mov eax,[esi + 0]
                xor ebx,ebx
                mov bl,[ebp + 12]
                shl ebx,3
                mov [idt + ebx + 5],al
```

```
; store handler address in stub
                  mov eax,isr1
                  sub eax,isr0      ; assume stub size < 256 bytes
                  mul byte [ebp + 12]
                  mov ebx,eax
                  add ebx,isr0
                  mov eax,[esi + 4]
                  mov [ebx + (isr0.1 - isr0 + 1)],eax
            pop ebx
            pop esi
      pop ebp
      ret
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; interrupt/exception stubs
; *** CAUTION: these must be consecutive, and must all be the same size.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

      INTR 0            ; zero divide (fault)
      INTR 1            ; debug/single step
      INTR 2            ; non-maskable interrupt (trap)
      INTR 3            ; INT3 (trap)
      INTR 4            ; INTO (trap)
      INTR 5            ; BOUND (fault)
      INTR 6            ; invalid opcode (fault)
      INTR 7            ; coprocessor not available (fault)
      INTR_EC 8   ; double fault (abort w/ error code)
      INTR 9            ; coproc segment overrun(abort; 386/486SX only)
      INTR_EC 0Ah ; bad TSS (fault w/ error code)
      INTR_EC 0Bh ; segment not present (fault w/ error code)
      INTR_EC 0Ch ; stack fault (fault w/ error code)
      INTR_EC 0Dh ; GPF (fault w/ error code)
      INTR_EC 0Eh ; page fault
      INTR 0Fh    ; reserved
      INTR 10h    ; FP exception/coprocessor error (trap)
      INTR 11h    ; alignment check (trap; 486+ only)
```

```
      INTR 12h    ; machine check (Pentium+ only)
      INTR 13h
      INTR 14h
      INTR 15h
      INTR 16h
      INTR 17h
      INTR 18h
      INTR 19h
      INTR 1Ah
      INTR 1Bh
      INTR 1Ch
      INTR 1Dh
      INTR 1Eh
      INTR 1Fh

; isr20 through isr2F are hardware interrupts. The 8259 programmable
; interrupt controller (PIC) must be reprogrammed to make these work.
      INTR 20h    ; IRQ 0/timer interrupt
      INTR 21h    ; IRQ 1/keyboard interrupt
      INTR 22h
      INTR 23h
      INTR 24h
      INTR 25h
      INTR 26h    ; IRQ 6/floppy interrupt
      INTR 27h
      INTR 28h    ; IRQ 8/real-time clock interrupt
      INTR 29h
      INTR 2Ah
      INTR 2Bh
      INTR 2Ch
      INTR 2Dh    ; IRQ 13/math coprocessor interrupt
      INTR 2Eh    ; IRQ 14/primary ATA ("IDE") drive interrupt
      INTR 2Fh    ; IRQ 15/secondary ATA drive interrupt

; syscall software interrupt
      INTR 30h
```

```
; the other 207 vectors are undefined
%assign i 31h
%rep (0FFh - 30h)
      INTR i
%assign i (i + 1)
%endrep
```

The data structure “regs” used here is:

```
typedef struct
{
/* pushed by pusha */
      unsigned edi, esi, ebp, esp, ebx, edx, ecx, eax;
/* pushed separately */
      unsigned ds, es, fs, gs;
      unsigned which_int, err_code;
/* pushed by exception. Exception may also push err_code.
user_esp and user_ss are pushed only if a privilege change occurs. */
      unsigned eip, cs, eflags, user_esp, user_ss;
} regs_t;
```

3.2 The Main Kernel Code

The main kernel code is a C program written in ANSI C given below.

```
int main(void)
{
      vector_t v;
      init_video();  // Initializes video
      kprintf("\n\nDevelopers of AVOS: \n");
      // other messeges ....
      init_keyboard(); //initializes keyboard

      init_8259s(); // initializes 8259 interrupt handler
```

```
	kprintf("Installing keyboard interrupt handler...\n");
                                        /* we don't save the old vector */
	v.eip = (unsigned)keyboard_irq;
	v.access_byte = 0x8E; /* present, ring 0, '386 interrupt gate */
	setvect(&v, 0x21);
	kprintf("Enabling hardware interrupts...\n");
	enable(); //Enable all interrupts

	kprintf("Enabling Flopy Disk Controller\n");
	fd_init(); // Enabling Floppy Controller

	kprintf("*** Press HLP to see command table ***\n");
	kprintf("*** Press Ctrl+Alt+Del to reboot ***\n\n\n");
	kprintf("$>"); // the prompt
	while(1)
		/* freeze but remain resident in memory */;
	return 0;
}
```

As can be seen from the above code, the main program module calls other functions from other program modules as and when required. Each of these functions in the module either initializes any one single device of the machine or performs certain device specific functions. For example, the program module "**video.c**" contains not only the function "init_video()" which initializes the video device, but also contains other video related specific functions. Some of these modules are described in subsequent sections.

3.3 Functions of the program module "video.c"

- void select_vc(unsigned which_vc) // selects video consoles
- static void move_csr(void) // moves the caret on the screen
- static void set_attrib(console_t *con, unsigned att) // sets attributes of the characters on the screen
- static void scroll(console_t *con) // scrolls one screen at a time

- void init_video(void) // initializes the video
- void blink(void) // how about blinking characters?
- static void putch_help(console_t *con, unsigned c) and void putch(unsigned c) // our very own printf function

The entire framebuffer is divided into a maximum of 12 virtual consoles, each operating independently. For each of the 12 VC's, independent data structure is maintained as shown below:-

```
typedef struct    /* circular queue */
{
      unsigned char *data;
      unsigned size, in_base, in_ptr/*, out_base*/, out_ptr;
} queue_t;

typedef struct
{
/* virtual console input */
      queue_t keystrokes;
/* virtual console output */
      unsigned esc, attrib, csr_x, csr_y, esc1, esc2, esc3;
      unsigned short *fb_adr;
} console_t;
```

Here, "console_t" is the data structure that maintains all variables corresponding to a single virtual console.

The function "*void select_vc(unsigned which_vc)* " selects a virtual console from among the 12 virtual consoles simply by updating the current cursor address and modifying the current base address of video display.

Thus, more than one command can be issued at the various command consoles at approximately the same time which execute paralleley, creating a multitasking effect.

3.4 Functions of the program module "kbd.c"

- static void reboot(void) // reboots machine
- static void write_kbd(unsigned adr, unsigned data) // writes to keyboard input buffer

- static unsigned convert(unsigned key)
- void keyboard_irq(void) // the keyboard irq
- int search_comnd(char *comnd) // searches for valid command word
- void init_keyboard(void) // initializes keyboard

Each of these functions are called as and when required and some of these like "keyboard_irq" remains hooked to the keyboard irq of 8259 interrupt controller and is called every time a key is pressed. The following piece of code does this:-

```
v.eip = (unsigned) keyboard_irq;
v.access_byte = 0x8E; /* present, ring 0, '386 interrupt gate */
setvect(&v, 0x21);
```

Where v is an instance of the data structure:-

```
typedef struct
{
      unsigned access_byte, eip;
} vector_t;
```

where "*access_byte*" is the access rights of the modified interrupt vector and "*eip*" is the instruction pointer pointing to the address of the first instruction of the interrupt service routine. In this case, "*eip*" points to "*keyboard_irq*", the symbolic address of the starting instruction of the isr that handles all the keyboard related functions.

Setvect is an assembly coded function given below

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; name:      setvect
; action:    writes interrupt vector
; in:        [EBP + 12] = vector number,vector stored at address [EBP + 8]
; out: (nothing)
```

```
; modifies:  EAX, EDX
; minimum CPU:      '386+
; notes:     C prototype:
;            typedef struct
;            {     unsigned access_byte, eip; } vector_t;
;            getvect(vector_t *v, unsigned vect_num);
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
EXP setvect
      push ebp
            mov ebp,esp
            push esi
            push ebx
                  mov esi,[ebp + 8]
; store access byte in IDT[i]; the main task of this subroutine
                  mov eax,[esi + 0]
                  xor ebx,ebx
                  mov bl,[ebp + 12]
                  shl ebx,3
                  mov [idt + ebx + 5],al
; store handler address in stub
                  mov eax,isr1
                  sub eax,isr0  ; assume stub size < 256 bytes
                  mul byte [ebp + 12]
                  mov ebx,eax
                  add ebx,isr0
                  mov eax,[esi + 4]
                  mov [ebx + (isr0.1 - isr0 + 1)],eax
            pop ebx
            pop esi
      pop ebp
      ret
```

The function makes the entry in the Interrupt Descriptor table for the keyboard interrupt (0x21) point to the “keyboard_irq” subroutine address. So each time the keyboard interrupt is raised, we land up in the “keyboard_irq” subroutine, which then handles the interrupt.

Here comes the real time nature of our O.S. The user presses the key in real time and instantaneously, the OS begins to service the interrupt to the user. There is no delay (saving the interrupt for later service) in our code. However, in a truly real time environment, the OS handles larger numbers of interrupts than is done by our operating system.

Chapter 4 Device Driver Modules

Some device drives are needed to access some specific hardware. We have written (with the help of Intel Manuals) only one such device driver, that of the Floppy Disk. The Floppy disk driver is written for all Intel based motherboards, having 82077A Floppy Disk Controller chip.

4.1 Description of the Floppy Disk Driver

The file "fd.c"contains all the codes for the floppy driver.
The driver consists of the following functions:

- void fd_init() // initializes floppy disk controller, the main function
- void fd_reset() // resets controler
- void fd_out(byte val) // writes a byte to controller FIFO register
- uchar fd_in() // reads a byte from controller FIFO register
- void fd_start(int drive) // Activate drive
- void fd_stop(int drive) // Deactivates Drive
- void fd_seek() // positions head to a particular cylinder, sector, track

The main function here is the "fd_init" function. It performs the following functions in sequence:

- Resets the floppy controller.
- Checks the version of the floppy controller and prints appropriate message on the screen. In case the floppy controller is of non-standard type, prints "unknown controller".

- Probes the bios CMOS settings directly to read the exact type of the floppy, 1.2 MB, 1.44 MB, 2.88MB, etc. installed. If the floppy controller is present, but the floppy is disabled from the bios CMOS settings, a probe at this stage reveals this fact.
- Finally, it demonstrates that the driver is actually properly functioning by activating and then deactivating the drive once. This process can be done each time the user tries to access the floppy through the functions in the driver.

The commands used by the driver are the standard command bytes recognized by the 82077A Floppy Disk Controller chip. The base address used for the controller is 3F0 Hex, which is the standard address used by the controller chip for all PC-AT based machines. The number of floppy drives available to the system is assumed to be two, however if more than two floppy drives present, they can be easily accommodated in this driver by minor modifications.

We used the 82077AA chmos single-chip floppy disk controller manual extensively in the coding process.

Chapter 5 The Source Codes

We used DJGPP, the GNU C compiler for DOS for building our entire source codes for this project. Before the assembly file and the C files can be patched together and compiled into a single kernel file, appropriate make files must be written so that the dependencies are well established. Makefiles are actually like dos “batch” files; they enumerate the list of commands the gnu “make.exe” should execute in sequence in order to build the project. The difference is that makefiles are much more powerful, efficient and can be used to write compact abbreviated codes for compilation.

We give below the entire contents of the make file of our project.

```
.SUFFIXES: .asm
# defines
MAKEFILE=makefile
MAKEDEP=$(MAKEFILE)
INCDIR = ../avos/inc
# chose COFF, PE/WIN32, or ELF in the next two lines
LDSCRIPT=../avos/coffkrnl.ld
NASM   =nasmw -f coff -dUNDERBARS=1 -i$(INCDIR)/
GCC    =gcc -g -Wall -W -O2 -nostdinc -fno-builtin -I$(INCDIR)
LD     =ld -g -T $(LDSCRIPT) -nostdlib
LIBC   = ../avos/lib/libc.a
OBJS   =kstart.o main.o video.o debug.o kbd.o fd.o
# targets
all: krnl.x
# implicit rules
.asm.o:
       $(NASM) -o$@ $<

.c.o:
       $(GCC) -c -o$@ $<
# dependencies
kstart.o:    kstart.asm          $(MAKEDEP)
main.o:      main.c       $(MAKEDEP)
```

```
video.o:     video.c             $(MAKEDEP)
debug.o:     debug.c             $(MAKEDEP)
kbd.o: kbd.c               $(MAKEDEP)
fd.o:        fd.c                $(MAKEDEP)

# explicit rules

$(LIBC): ../avos/lib/$(MAKEFILE)
      make -C ../avos/lib -f $(MAKEFILE)

krnl.x: $(OBJS) $(LDSCRIPT) $(LIBC) $(MAKEDEP)
      $(LD) -o$@ $(OBJS) $(LIBC)
      objdump --source $@ >krnl.lst
      nm --line-numbers $@ | sort >krnl.sym
      strip $@
```

Below is the listing of the C source codes of our kernel. They all follow Ansi C syntax.

5.1 Module “Main.c” – The Main Kernel Code

```
/*====================================================================
MAIN KERNEL CODE
EXPORTS:
void kprintf(const char *fmt, ...);
int main(void);
====================================================================*/
#include <stdarg.h> /* va_list, va_start(), va_end() */
#include <string.h> /* NULL */
#include <x86.h> /* disable() */
#include <_printf.h> /* do_printf() */
#include "_krnl.h" /* regs_t */

/* IMPORTS from KSTART.ASM */
void getvect(vector_t *v, unsigned vect_num);
void setvect(vector_t *v, unsigned vect_num);
/* from VIDEO.C */
```

```
void blink(void);
void putch(unsigned c);
void init_video(void);

/* from KBD.C */
void keyboard_irq(void);
void init_keyboard(void);

/* from FD.C */
void fd_init(void);

/*************************************************************************
*************************************************************************/
static int kprintf_help(unsigned c, void **ptr)
{
      putch(c);
      return 0;
}
/*************************************************************************
*************************************************************************/
void kprintf(const char *fmt, ...)
{
      va_list args;

      va_start(args, fmt);
      (void)do_printf(fmt, args, kprintf_help, NULL);
      va_end(args);
}
/*************************************************************************
*************************************************************************/
void panic(const char *fmt, ...)
{
      va_list args;
      disable(); /* interrupts off */
      va_start(args, fmt);
      (void)do_printf(fmt, args, kprintf_help, NULL);
```

```
	while(1)
		/* freeze */;
}
/*************************************************************************
*************************************************************************/
void fault(regs_t *regs)
{
	static const char * const msg[] =
	{
		"divide error", "debug exception", "NMI", "INT3",
		"INTO", "BOUND exception", "invalid opcode", "no coprocessor",
		"double fault", "coprocessor segment overrun",
			"bad TSS", "segment not present",
		"stack fault", "GPF", "page fault", "??",
		"coprocessor error", "alignment check", "??", "??",
		"??", "??", "??", "??",
		"??", "??", "??", "??",
		"??", "??", "??", "??",
		"IRQ0", "IRQ1", "IRQ2", "IRQ3",
		"IRQ4", "IRQ5", "IRQ6", "IRQ7",
		"IRQ8", "IRQ9", "IRQ10", "IRQ11",
		"IRQ12", "IRQ13", "IRQ14", "IRQ15",
		"syscall"
	};
	switch(regs->which_int)
	{
/* this handler installed at compile-time
Keyboard handler is installed at run-time (see below) */
	case 0x20:  /* timer IRQ 0 */
		blink();
/* reset hardware interrupt at 8259 chip */
		outportb(0x20, 0x20);
		break;
	default:
		kprintf("Exception #%u", regs->which_int);
		if(regs->which_int <= sizeof(msg) / sizeof(msg[0]))
```

```
                kprintf(" (%s)", msg[regs->which_int]);
            panic("\nSYSTEM HALTED\nuse reset button to end");
      }
}
/*************************************************************************
*************************************************************************/
static void init_8259s(void)
{
      static const unsigned irq0_int = 0x20, irq8_int = 0x28;

/* Initialization Control Word #1 (ICW1) */
      outportb(0x20, 0x11);
      outportb(0xA0, 0x11);
/* ICW2:
route IRQs 0-7 to INTs 20h-27h */
      outportb(0x21, irq0_int);
/* route IRQs 8-15 to INTs 28h-2Fh */
      outportb(0xA1, irq8_int);
/* ICW3 */
      outportb(0x21, 0x04);
      outportb(0xA1, 0x02);
/* ICW4 */
      outportb(0x21, 0x01);
      outportb(0xA1, 0x01);
/* enable IRQ0 (timer) and IRQ1 (keyboard) */
      outportb(0x21, ~0x03);
      outportb(0xA1, ~0x00);
}
/*************************************************************************
for MinGW32
*************************************************************************/
#ifdef __WIN32__
int __main(void) { return 0; }
#endif
/*************************************************************************
*************************************************************************/
```

```
int main(void)
{
      vector_t v;

      init_video(); // Enabling video chipset

      kprintf("\n\nDevelopers of AVOS: \n");
      kprintf("Anirban Sinha, B.tech Final Year, CSE, IEM, Kolkata\n");
      kprintf("Satyajit Chakrabarti, B.tech Final Year,CSE, IEM,
      Kolkata\n");
      kprintf("Soumya Banerjee, B.tech Final Year, CSE, IEM, Kolkata\n");
      kprintf("Satyabrata   Sarkar,   B.tech   Final   Year,   CSE,   IEM,
      Kolkata\n**********************************************************
      *********\n");
      kprintf("Idea Conceived by: Satyajit Chakrabarti\n\nConsole Command
      Interpreter developed by: Anirban
      Sinha\n************************************************\n\n");

      init_keyboard(); // Enabling keyboard controller

      init_8259s(); // Enabling 8259 interrupt controller
      kprintf("Installing keyboard interrupt handler...\n");
      /* we don't save the old vector */
      v.eip = (unsigned)keyboard_irq;
      v.access_byte = 0x8E; /* present, ring 0, '386 interrupt gate */
      setvect(&v, 0x21);

      kprintf("Enabling hardware interrupts...\n");
      enable();

      kprintf("Enabling Flopy Disk Controller\n");
      fd_init(); // Enabling Floppy Controller

      kprintf("************************************************\n");
      kprintf("*** Press F1, F2, etc. to change virtual consoles (total 12
      VC's) ***\n");
```

```
      kprintf("*** Press HLP to see command table ***\n");
      kprintf("*** Press Ctrl+Alt+Del to reboot ***\n\n\n");
      kprintf("$>"); // the prompt
      while(1)
            /* freeze */;
/* this is not reached */
      return 0;
}
```

Note:

Some of the functions that are supposed to be done by the boot loader have been incorporated into the main kernel code; this makes bootloader independent of the type of kernel used and helps to configure the system according to the needs of the present kernel regardless of the bootloader used. For example: the configuring of 8259's has been done in the kernel code. This reconfiguration may also have been done in the bootloader program but that would make the bootloader dependent on the kernel since the type of interrupts used depends on the design of the kernel.

As far as possible, we have made the boot loader independent of our kernel.

5.2 Module “Video.c” – Video Handling Module

```
/*====================================================================
TEXT VIDEO ROUTINES
EXPORTS:
void blink(void);
void select_vc(unsigned which_con);
void putch(unsigned char c);
void init_video(void);
====================================================================*/
#include <string.h> /* memcpy(), memsetw() */
#include <ctype.h> /* isdigit() */
#include <x86.h> /* outportb(), inportb() */
#include "_krnl.h" /* MAX_VC, console_t */
#include <conio.h> /* KEY_nn */

/* IMPORTS from MAIN.C */
void kprintf(const char *fmt, ...);

#define     VGA_MISC_READ     0x3CC
console_t _vc[MAX_VC];
static      unsigned short _num_vcs;
static      console_t *_curr_vc;
static unsigned short *_vga_fb_adr;
static unsigned _crtc_io_adr, _vc_width, _vc_height;
/*********************************************************************
*********************************************************************/
void blink(void)
{
      (*(unsigned char *)_vga_fb_adr)++;
}
/*********************************************************************
*********************************************************************/
static void scroll(console_t *con)
{
```

```
	unsigned short *fb_adr;
	unsigned blank, temp;
	blank = 0x20 | ((unsigned)con->attrib << 8);
	fb_adr = con->fb_adr;
/* scroll up */
	if(con->csr_y >= _vc_height)
	{
		temp = con->csr_y - _vc_height + 1;
		memcpy(fb_adr, fb_adr + temp * _vc_width,
			(_vc_height - temp) * _vc_width * 2);
/* blank the bottom line of the screen */
		memsetw(fb_adr + (_vc_height - temp) * _vc_width,
			blank, _vc_width);
		con->csr_y = _vc_height - 1;
	}
}
/*****************************************************************
*****************************************************************/

static void set_attrib(console_t *con, unsigned att)
{
	static const unsigned ansi_to_vga[] =
	{
		0, 4, 2, 6, 1, 5, 3, 7
	};
	unsigned new_att;

	new_att = con->attrib;
	if(att == 0)
		new_att &= ~0x08;	/* bold off */
	else if(att == 1)
		new_att |= 0x08;	/* bold on */
	else if(att >= 30 && att <= 37)
	{
		att = ansi_to_vga[att - 30];
		new_att = (new_att & ~0x07) | att;/* fg color */
```

```
	}
	else if(att >= 40 && att <= 47)
	{
		att = ansi_to_vga[att - 40] << 4;
		new_att = (new_att & ~0x70) | att;/* bg color */
	}
	con->attrib = new_att;
}
/*************************************************************************
*************************************************************************/

static void move_csr(void)
{
	unsigned short temp;

	temp = (_curr_vc->csr_y * _vc_width + _curr_vc->csr_x) +
		(_curr_vc->fb_adr - _vga_fb_adr);
	outportb(_crtc_io_adr + 0, 14);
	outportb(_crtc_io_adr + 1, temp >> 8);
	outportb(_crtc_io_adr + 0, 15);
	outportb(_crtc_io_adr + 1, temp);
}
/*************************************************************************
*************************************************************************/
void select_vc(unsigned which_vc)
{
	unsigned i;

	if(which_vc >= _num_vcs)
		return;
	_curr_vc = _vc + which_vc;
	i = _curr_vc->fb_adr - _vga_fb_adr;
	outportb(_crtc_io_adr + 0, 12);
	outportb(_crtc_io_adr + 1, i >> 8);
	outportb(_crtc_io_adr + 0, 13);
	outportb(_crtc_io_adr + 1, i);
```

```
/* oops, forgot this: */
      move_csr();
}
/*************************************************************************
*************************************************************************/
static void putch_help(console_t *con, unsigned c)
{
      unsigned short *fb_adr;
      unsigned att;

      att = (unsigned)con->attrib << 8;
      fb_adr = con->fb_adr;
/* state machine to handle the escape sequences ESC */
      if(con->esc == 1)
      {
            if(c == '[')
            {
                  con->esc++;
                  con->esc1 = 0;
                  return;
            }
            /* else fall-through: zero esc and print c */
      }
/* ESC[ */
      else if(con->esc == 2)
      {
            if(isdigit(c))
            {
                  con->esc1 = con->esc1 * 10 + c - '0';
                  return;
            }
            else if(c == ';')
            {
                  con->esc++;
                  con->esc2 = 0;
                  return;
```

```
            }
/* ESC[2J -- clear screen */
            else if(c == 'J')
            {
                  if(con->esc1 == 2)
                  {
                        memsetw(fb_adr, ' ' | att,
                              _vc_height * _vc_width);
                        con->csr_x = con->csr_y = 0;
                  }
            }
/* ESC[num1m -- set attribute num1 */
            else if(c == 'm')
                  set_attrib(con, con->esc1);
            con->esc = 0;     /* anything else with one numeric arg */
            return;
      }
/* ESC[num1; */
      else if(con->esc == 3)
      {
            if(isdigit(c))
            {
                  con->esc2 = con->esc2 * 10 + c - '0';
                  return;
            }
            else if(c == ';')
            {
                  con->esc++; /* ESC[num1;num2; */
                  con->esc3 = 0;
                  return;
            }
/* ESC[num1;num2H -- move cursor to num1,num2 */
            else if(c == 'H')
            {
                  if(con->esc2 < _vc_width)
                        con->csr_x = con->esc2;
```

```
                  if(con->esc1 < _vc_height)
                        con->csr_y = con->esc1;
            }
/* ESC[num1;num2m -- set attributes num1,num2 */
            else if(c == 'm')
            {
                  set_attrib(con, con->esc1);
                  set_attrib(con, con->esc2);
            }
            con->esc = 0;
            return;
      }
/* ESC[num1;num2;num3 */
      else if(con->esc == 4)
      {
            if(isdigit(c))
            {
                  con->esc3 = con->esc3 * 10 + c - '0';
                  return;
            }
/* ESC[num1;num2;num3m -- set attributes num1,num2,num3 */
            else if(c == 'm')
            {
                  set_attrib(con, con->esc1);
                  set_attrib(con, con->esc2);
                  set_attrib(con, con->esc3);
            }
            con->esc = 0;
            return;
      }
      con->esc = 0;

/* escape character */
      if(c == 0x1B)
      {
            con->esc = 1;
```

```
            return;
      }
/* backspace */
      if((c == 0x08) || (c== KEY_LFT))
      {
            if(con->csr_x != 2)
            {
                  con->csr_x--;
                  unsigned short *where;
                  unsigned blk;
                  blk=0x20;
                  where = fb_adr + (con->csr_y * _vc_width + con->csr_x);
                  *where = (blk | att);
            }

      }
/* tab */
      else if(c == 0x09)
            con->csr_x = (con->csr_x + 8) & ~(8 - 1);
/* carriage return */
      else if(c == '\r')      /* 0x0D */
            con->csr_x = 0;
/* line feed */
//    else if(c == '\n')      /* 0x0A */
//          con->csr_y++;
/* CR/LF */
      else if(c == '\n')      /* ### - 0x0A again */
      {
            con->csr_x = 0;
            con->csr_y++;
      }
/* printable ASCII */
      else if(c >= ' ')
      {
            unsigned short *where;
```

```
            where = fb_adr + (con->csr_y * _vc_width + con->csr_x);
            *where = (c | att);
            con->csr_x++;
      }
      if(con->csr_x >= _vc_width)
      {
            con->csr_x = 0;
            con->csr_y++;
      }
      scroll(con);
/* move cursor only if the VC we're writing is the current VC */
      if(_curr_vc == con)
            move_csr();
}

/*************************************************************************
*************************************************************************/
void putch(unsigned c)
{
/* all kernel messages to VC #0 */
//    putch_help(_curr_vc, c);
/* all kernel messages to current VC */
      putch_help(_curr_vc, c);
}
/*************************************************************************
*************************************************************************/
void init_video(void)
{
      unsigned i;

/* check for monochrome or color VGA emulation */
      if((inportb(VGA_MISC_READ) & 0x01) != 0)
      {
            _vga_fb_adr = (unsigned short *)0xB8000L;
            _crtc_io_adr = 0x3D4;
      }
```

```
      else
      {
            _vga_fb_adr = (unsigned short *)0xB0000L;
            _crtc_io_adr = 0x3B4;
      }

/* read current screen size from BIOS data segment(addresses 400-4FF) */
      _vc_width = *(unsigned short *)0x44A;
      _vc_height = *(unsigned char *)0x484 + 1;

/* figure out how many VCs we can have with 32K of display memory.
Use INTEGER division to round down. */
      _num_vcs = 32768L / (_vc_width * _vc_height * 2);
      if(_num_vcs > MAX_VC)
            _num_vcs = MAX_VC;

/* init VCs, with a different foreground color for each */
      for(i = 0; i < _num_vcs; i++)
      {
            _curr_vc = _vc + i;
            _curr_vc->attrib = i + 7;
            _curr_vc->fb_adr = _vga_fb_adr +
                  _vc_width * _vc_height * i;
/* ESC[2J clears the screen */
            kprintf("\x1B[2J    Advanced  OS  ver:  1.0.0  \n  You  are  at
VirtualConsole #%u (of 1-%u)\n",
                  i+1, _num_vcs);
            kprintf("Inititializing   video:   %s   emulation,   %u   x   %u,
framebuffer at "
            "0x%lX\n", (_crtc_io_adr == 0x3D4) ? "color" : "mono",
            _vc_width, _vc_height, _vga_fb_adr);
            if (i!=0)
                  kprintf("$>"); // the prompt
      }
      select_vc(0);
```

```
}
```

5.3 Module “kbd.c” – Keyboard Handling Module

```
/*=========================================================================
KEYBOARD ROUTINES

EXPORTS:
void keyboard_irq(void);
void init_keyboard(void);
=========================================================================*/

#include <conio.h> /* KEY_nnn */
#include <x86.h> /* outportb(), inportb() */
#include "_krnl.h" /* MAX_VC */

/* IMPORTS
from VIDEO.C */
extern console_t _vc[];
void select_vc(unsigned which_vc);
void putch(unsigned c);
int search_comnd(char*);
// searches inputted command string for valid command word
/* from MAIN.C */
void kprintf(const char *fmt, ...);

/* "raw" set 1 scancodes from PC keyboard. Keyboard info here:*/
#define     RAW1_LEFT_CTRL          0x1D
#define     RAW1_RIGHT_CTRL         0x1D  /* same as left */
#define     RAW1_LEFT_SHIFT         0x2A
#define     RAW1_RIGHT_SHIFT  0x36
#define     RAW1_LEFT_ALT           0x38
#define     RAW1_RIGHT_ALT          0x38  /* same as left */
#define     RAW1_CAPS_LOCK          0x3A
#define     RAW1_F1                 0x3B
```

```
#define     RAW1_F2                 0x3C
#define     RAW1_F3                 0x3D
#define     RAW1_F4                 0x3E
#define     RAW1_F5                 0x3F
#define     RAW1_F6                 0x40
#define     RAW1_F7                 0x41
#define     RAW1_F8                 0x42
#define     RAW1_F9                 0x43
#define     RAW1_F10          0x44
#define     RAW1_NUM_LOCK           0x45
#define     RAW1_SCROLL_LOCK  0x46
#define     RAW1_F11          0x57
#define     RAW1_F12          0x58

#define     KBD_BUF_SIZE            64
/*********************************************************************
*********************************************************************/
void delay()
{
      long cnt1;
      long cnt2;

    cnt1=999999999;
      cnt2=59;
      while(cnt1!=0)
      {
            while(cnt2!=0)
            {
                  cnt2--;
            }

            cnt1--;
      }

      return;
}
```

```
/*************************************************************************
*************************************************************************/
static void reboot(void) // reboots machine
{
      unsigned temp;

      disable();
/* flush the keyboard controller */
      do
      {
            temp = inportb(0x64);
            if((temp & 0x01) != 0)
            {
                  (void)inportb(0x60);
                  continue;
            }
      } while((temp & 0x02) != 0);
/* pulse the CPU reset line */
      outportb(0x64, 0xFE);
/* ...and if that didn't work, just halt */
      while(1)
            /* nothing */;
}
/*************************************************************************
*************************************************************************/
static void write_kbd(unsigned adr, unsigned data)
{
      unsigned long timeout;
      unsigned stat;

      for(timeout = 500000L; timeout != 0; timeout--)
      {
            stat = inportb(0x64);
/* loop until 8042 input buffer empty */
            if((stat & 0x02) == 0)
```

```
                  break;
      }
      if(timeout != 0)
            outportb(adr, data);
}

/*********************************************************************
*********************************************************************/
static unsigned convert(unsigned key)
{
      static const unsigned char set1_map[] =
      {
/* 00 */0,  0x1B, '1',  '2',  '3',  '4',  '5',  '6',
/* 08 */'7',      '8',  '9',  '0',  '-',  '=',  '\b', '\t',
/* 10 */'q',      'w',  'e',  'r',  't',  'y',  'u',  'i',
/* 1Dh is left Ctrl */
/* 18 */'o',      'p',  '[',  ']',  '\n', 0,    'a',  's',
/* 20 */'d',      'f',  'g',  'h',  'j',  'k',  'l',  ';',
/* 2Ah is left Shift */
/* 28 */'\'',     '`',  0,    '\\', 'z',  'x',  'c',  'v',
/* 36h is right Shift */
/* 30 */'b',      'n',  'm',  ',',  '.',  '/',  0,    0,
/* 38h is left Alt, 3Ah is Caps Lock */
/* 38 */0,  ' ',  0,    KEY_F1,     KEY_F2,     KEY_F3,     KEY_F4,
      KEY_F5,
/* 45h is Num Lock, 46h is Scroll Lock */
/* 40 */KEY_F6,   KEY_F7,     KEY_F8,     KEY_F9,     KEY_F10,0,  0,
      KEY_HOME,
/* 48 */KEY_UP,   KEY_PGUP,'-',     KEY_LFT,'5',      KEY_RT,     '+',
      KEY_END,
/* 50 */KEY_DN,   KEY_PGDN,KEY_INS,KEY_DEL,0,   0,    0,    KEY_F11,
/* 58 */KEY_F12
      };
      static unsigned short kbd_status, saw_break_code;
/**/
      unsigned short temp;
```

```
/* check for break key (i.e. a key is released) */
      if(key >= 0x80)
      {
            saw_break_code = 1;
            key &= 0x7F;
      }
/* the only break codes we're interested in are Shift, Ctrl, Alt */
      if(saw_break_code)
      {
            if(key == RAW1_LEFT_ALT || key == RAW1_RIGHT_ALT)
                  kbd_status &= ~KBD_META_ALT;
            else if(key == RAW1_LEFT_CTRL || key == RAW1_RIGHT_CTRL)
                  kbd_status &= ~KBD_META_CTRL;
            else if(key == RAW1_LEFT_SHIFT || key == RAW1_RIGHT_SHIFT)
                  kbd_status &= ~KBD_META_SHIFT;
            saw_break_code = 0;
            return 0;
      }
/* it's a make key: check the "meta" keys, as above */
      if(key == RAW1_LEFT_ALT || key == RAW1_RIGHT_ALT)
      {
            kbd_status |= KBD_META_ALT;
            return 0;
      }
      if(key == RAW1_LEFT_CTRL || key == RAW1_RIGHT_CTRL)
      {
            kbd_status |= KBD_META_CTRL;
            return 0;
      }
      if(key == RAW1_LEFT_SHIFT || key == RAW1_RIGHT_SHIFT)
      {
            kbd_status |= KBD_META_SHIFT;
            return 0;
      }
/* Scroll Lock, Num Lock, and Caps Lock set the LEDs. These keys
```

```
have on-off (toggle or XOR) action, instead of momentary action */
      if(key == RAW1_SCROLL_LOCK)
      {
            kbd_status ^= KBD_META_SCRL;
            goto LEDS;
      }
      if(key == RAW1_NUM_LOCK)
      {
            kbd_status ^= KBD_META_NUM;
            goto LEDS;
      }
      if(key == RAW1_CAPS_LOCK)
      {
            kbd_status ^= KBD_META_CAPS;
LEDS:       write_kbd(0x60, 0xED);  /* "set LEDs" command */
            temp = 0;
            if(kbd_status & KBD_META_SCRL)
                  temp |= 1;
            if(kbd_status & KBD_META_NUM)
                  temp |= 2;
            if(kbd_status & KBD_META_CAPS)
                  temp |= 4;
            write_kbd(0x60, temp);  /* bottom 3 bits set LEDs */
            return 0;
      }
/* ignore invalid scan codes */
      if(key >= sizeof(set1_map) / sizeof(set1_map[0]))
            return 0;
/* convert raw scancode in key to unshifted ASCII in temp */
      temp = set1_map[key];
/* defective keyboard? non-US keyboard? more than 104 keys? */
      if(temp == 0)
            return temp;
/* handle the three-finger salute */
      if((kbd_status & KBD_META_CTRL) && (kbd_status & KBD_META_ALT) &&
(temp == KEY_DEL))
```

```
      {
                        kprintf("\x1B[2J""\n\n\n\n\n\n\n\n\n\n\n\n\n");
                        kprintf("
                                        ***  Good bye  *** \n");
                        kprintf("
                                    ***  Wait while Rebooting!  ***");


                delay();

                reboot();
      }

      return temp;
}
/*********************************************************************
*********************************************************************/
void keyboard_irq(void) // interpretes keyboard key inputs
{
      unsigned key, i;
      static unsigned vc;
      static char cmd_buff[5];
      static unsigned buff_ptr;
      static unsigned bad_comnd;
      int  comnd;
      bad_comnd=0;

/* get scancode from port 0x60 */
      key = inportb(0x60);

/* if it's F1, F2 etc. switch to the appropriate virtual console */
      switch(key)
      {
      case RAW1_F1:
            i = 0;
            goto SWITCH_VC;
```

```
      case RAW1_F2:
            i = 1;
            goto SWITCH_VC;
      case RAW1_F3:
            i = 2;
            goto SWITCH_VC;
      case RAW1_F4:
            i = 3;
            goto SWITCH_VC;
      case RAW1_F5:
            i = 4;
            goto SWITCH_VC;
      case RAW1_F6:
            i = 5;
            goto SWITCH_VC;
      case RAW1_F7:
            i = 6;
            goto SWITCH_VC;
      case RAW1_F8:
            i = 7;
            goto SWITCH_VC;
      case RAW1_F9:
            i = 8;
            goto SWITCH_VC;
      case RAW1_F10:
            i = 9;
            goto SWITCH_VC;
      case RAW1_F11:
            i = 10;
            goto SWITCH_VC;
      case RAW1_F12:
            i = 11;
SWITCH_VC:
            select_vc(i);
            vc=i;
            break;
```

```
      default:
            i = convert(key);
            if((i==KEY_HOME) || (i==0x09) || (i==KEY_INS) || (i==KEY_LWIN)
||  (i==KEY_RWIN)  ||  (i==KEY_MENU)  ||  (i==KEY_PRNT)  ||  (i==KEY_PAUSE)||
(i==KEY_DEL) || (i==KEY_UP) || (i==KEY_PGUP) || (i==KEY_RT) || (i==KEY_END)
|| (i==KEY_DN) || (i==KEY_PGDN))
            {
                  break;
            }

            if(((i==0x08) || (i==KEY_LFT)) && (buff_ptr!=0) )
                                                      // for back space
            {
                  if(buff_ptr<=4)
                        cmd_buff[buff_ptr-1]='\0';
                  buff_ptr--;
                  putch(i);
                  break;
            }

            if((i != 0) && (i!= '\n') && (i!= 0x08))
            {
                  if(buff_ptr<=4)
                  {
                        cmd_buff[buff_ptr]=i;

                  }
                  else if(buff_ptr > 4)
                        bad_comnd=1;

                  buff_ptr++;
                  putch(i);
                  break;
            }
            if(i=='\n' && (buff_ptr>0))
            {
```

```
            if(bad_comnd)
            {
                  kprintf("\nInvalid Command\n");
                  bad_comnd=0;
            }

            comnd=search_comnd(cmd_buff);

            switch(comnd)
                  {

                        case 1:
                                          kprintf("\x1B[2J   Advanced
                                          OS ver: 1.0.0 \nYou are at
                                          Virtual           Console:
                                          %u\n",vc+1);

                                          break;
                        case 2:
                                          kprintf("\nAdvanced      OS
                                          version 1.0.0\n");
                                          break;
                        case 3:


kprintf("\x1B[2J""\n\n\n\n\n\n\n\n\n\n\n\n");
kprintf("
                              ***  Good bye  *** \n");
kprintf("
                            ***  Wait while Rebooting!  ***");

                        delay();
                                    reboot();
                                    break;
                        case 4:
                        kprintf("\n\nAVOS Help:\n----------\n\n");
```

```
                        kprintf("F1       Console #1\n");
                        kprintf("F2       Console #2\n");
                        kprintf("F3       Console #3\n");
                        kprintf("F4       Console #4\n");
                        kprintf("F5       Console #5\n");
                        kprintf("F6       Console #6\n");
                        kprintf("F7       Console #7\n");
                        kprintf("F8       Console #8\n");
                        kprintf("F9       Console #9\n");
                        kprintf("F10            Console #10\n");
                        kprintf("F11            Console #11\n");
                        kprintf("F12            Console #12\n\n");
                        kprintf("cls      Clears Current Console
                        Screen\n");
                        kprintf("ver      Prints AVOS Version\n");
                        kprintf("reb      Reboots Computer\n\n");
                        break;
                        case 5:
                        case -1:
                              kprintf("\nInvalid Command\n");
                  }

            putch(i);
            kprintf("$>"); // prompt
            buff_ptr=0;
            cmd_buff[0]='\0';
            cmd_buff[1]='\0';
            cmd_buff[2]='\0';
            cmd_buff[3]='\0';
            cmd_buff[4]='\0';
            break;
      }
      else if((i=='\n') && (buff_ptr==0)) // single enter key
      {
            putch(i);
            kprintf("$>");
```

```
                break;
            }

      }
/* reset hardware interrupt at 8259 chip */
      outportb(0x20, 0x20);
}

/*************************************************************************
*************************************************************************/
int search_comnd(char *comnd) //searches inputted command string for ..
                                        // .. valid command word
{
      if(*(comnd+3)!='\0') // max length of commands is 3 characters
            return -1;
      switch(*(comnd+0))
      {
            case 'c':
            case 'C':
                              switch(*(comnd+1))
                              {
                                    case 'l':
                                    case 'L':
                                                switch (*(comnd+2))
                                                      {
                                                            case 's':
                                                            case 'S':

      return 1;

      break;
                                                            default:
      return -1;
                                                      }
                                    break;
                                    default: return -1;
```

```
                        }
                        break;

        case 'v':
        case 'V':
                        switch(*(comnd+1))
                        {
                              case 'e':
                              case 'E':
                                          switch (*(comnd+2))
                                                {
                                                      case 'r':
                                                      case 'R':

return 2;

break;
                                                      default:
return -1;
                                                }
                              break;
                              default: return -1;
                        }
                        break;

        case 'r':
        case 'R':
                        switch(*(comnd+1))
                        {
                              case 'e':
                              case 'E':
                                    switch (*(comnd+2))
                                                {
                                                      case 'b':
                                                      case 'B':
```

```
                                                return 3;

        break;
                                                      default:
        return -1;
                                                }
                                    break;
                                    default: return -1;
                              }
                              break;
              case 'h':
              case 'H':
                              switch(*(comnd+1))
                              {
                                    case 'l':
                                    case 'L':
                                                switch (*(comnd+2))
                                                {
                                                      case 'p':
                                                      case 'P':
                                                            return 4;

        break;
                                                      default:
        return -1;
                                                }
                                    break;
                                    default: return -1;
                              }
                              break;
              case 'd':
              case 'D':
                              switch(*(comnd+1))
                              {
                                    case 't':
```

```
                                    case 'T':
                                                return 5;
                                                break;
                                    default:    return -1;
                              }

                              break;

            default: return -1;
      }
}



/*********************************************************************
*********************************************************************/
void init_keyboard(void)
{
      static unsigned char buffers[KBD_BUF_SIZE * MAX_VC];
      int i;
      for(i = 0; i < MAX_VC; i++)
      {
            _vc[i].keystrokes.data = buffers + KBD_BUF_SIZE * i;
            _vc[i].keystrokes.size = KBD_BUF_SIZE;
      }
      kprintf("Initializing keyboard: %u buffers, %u bytes each\n",
                                                MAX_VC, KBD_BUF_SIZE);
}
```

5.4 Library Routines

5.4.1 Hardware Interface Functions

1. Disable Interrupts

```
/*************************************************************
*************************************************************/
unsigned disable(void)
{
   unsigned ret_val;

   __asm__ __volatile__("pushfl\n"
         "popl %0\n"
         "cli"
         : "=a"(ret_val)
         :);
   return ret_val;
}
```

2. Enable Interrupts

```
/*************************************************************
*************************************************************/
void enable(void)
{
   __asm__ __volatile__("sti"
         :
         :
         );
}
```

3. Input from Port

```
/************************************************************
************************************************************/
unsigned inportb(unsigned short port)
{
   unsigned char ret_val;

   __asm__ __volatile__("inb %1,%0"
         : "=a"(ret_val)
         : "d"(port));
   return ret_val;
}
```

4. Output to Port

```
/************************************************************
************************************************************/
void outportb(unsigned port, unsigned val)
{
   __asm__ __volatile__("outb %b0,%w1"
         :
         : "a"(val), "d"(port));
}
```

5.4.2 String Manipulation Functions

1. Copy contents of memory from one location to other

```
   /*********************************************************
   *********************************************************/
   void *memcpy(void *dst_ptr, const void *src_ptr, size_t count)
   {
         void *ret_val = dst_ptr;
         const char *src = (const char *)src_ptr;
```

```
        char *dst = (char *)dst_ptr;

    /* copy up */
        for(; count != 0; count--)
            *dst++ = *src++;
        return ret_val;
    }
```

2. Find Length of String

```
    /*********************************************************
    *********************************************************/
    size_t strlen(const char *str)
    {
        size_t ret_val;

        for(ret_val = 0; *str != '\0'; str++)
            ret_val++;
        return ret_val;
    }
```

3. Fill “count” sized block of memory with a specific value “val”

```
    /*********************************************************
    *********************************************************
    void *memsetw(void *dst, int val, size_t count)
    {
        unsigned short *temp = (unsigned short *)dst;
        for( ; count != 0; count--)
            *temp++ = val;
        return dst;
    }
```

5.4.3 Segment Jump Functions

1. Intra Segment Jump

```
int setjmp(jmp_buf buf)
{
      __asm__ __volatile__("movl %%edi,(%0)\n"
            "movl %%esi,4(%0)\n"
            "movl %%ebp,8(%0)\n"
            "movl %%ebx,16(%0)\n"
            "movl %%edx,20(%0)\n"
            "movl %%ecx,24(%0)\n"
/* ESP value after we RET from this function */
            "leal 8(%%ebp),%%eax\n"
            "movl %%eax,32(%0)\n"
/* EIP value after we RET from this function */
            "movl 4(%%ebp),%%eax\n"
            "movl %%eax,36(%0)\n" : : "b"(buf));
      return 0;
}
```

2. Inter Segment Jump (Long Jump)

```
void longjmp(jmp_buf buf, int ret_val)
{
      unsigned *esp2;
/* make sure return value is not 0 */
      if(ret_val == 0)
            ret_val++;
/* store it in buf->eax */
      buf->eax = ret_val;
/* get ESP for new stack */
      esp2 = (unsigned *)buf->esp2;
/* push return EIP on new stack */
      esp2--;
      *esp2 = buf->eip;
/* update stored ESP */
```

```
        buf->esp2 = (unsigned)esp2;
    /* make buf the stack */
        __asm__ __volatile__("movl %0,%%esp\n"
    /* pop GP registers */
                "popa\n"
    /* pop ESP for new stack */
                "pop %%esp\n"
    /* return */
                "ret\n"
                :
                : "m"(buf));
    }
```

5.4.4 The “Printf” Function’

```
/*************************************************************
name:           do_printf
action:         minimal subfunction for ?printf, calls function
                'fn' with arg 'ptr' for each character to be output
returns:        total number of characters output
notes:
does not handle long long (64-bit) values, far pointers, floats,
precision part of field width, leading sign, or leading blanks.
*************************************************************/
/* flags used in processing format string */
#define         PR_LJ 0x01  /* left justify */
#define         PR_CA 0x02  /* use A-F instead of a-f for hex */
#define         PR_SG 0x04  /*  signed  numeric  conversion  (%d  vs.
%u) */
#define         PR_32 0x08  /* long (32-bit) numeric conversion */
#define         PR_16 0x10  /* short (16-bit) numeric conversion */
#define         PR_WS 0x20  /* PR_SG set and num was < 0 */
#define         PR_LZ 0x40  /* pad left with '0' instead of ' ' */
#define         PR_FP 0x80  /* pointers are far */

/* largest number handled is 2^32-1, lowest radix handled is 8.
```

```
2^32-1 in base 8 has 11 digits (add 5 for trailing NUL and for
slop) */
#define        PR_BUFLEN   16

typedef int (*fnptr_t)(unsigned c, void **helper);

int do_printf(const char *fmt,va_list args, fnptr_t fn, void *ptr)
{
   unsigned state, flags, radix, actual_wd, count, given_wd;
   unsigned char *where, buf[PR_BUFLEN];
   long num;
   state = flags = count = given_wd = 0;
/* begin scanning format specifier list */
   for(; *fmt; fmt++)
   {
         switch(state)
         {
/* STATE 0: AWAITING % */
         case 0:
               if(*fmt != '%')   /* not %... */
               {
                     fn(*fmt, &ptr);   /* ...just echo it */
                     count++;
                     break;
               }
/* found %, get next char and advance state to check if next char
is a flag */
               state++;
               fmt++;
               /* FALL THROUGH */
/* STATE 1: AWAITING FLAGS (%-0) */
         case 1:
               if(*fmt == '%')   /* %% */
               {
                     fn(*fmt, &ptr);
                     count++;
```

```
                        state = flags = given_wd = 0;
                        break;
                }
                if(*fmt == '-')
                {
                        if(flags & PR_LJ)/* %-- is illegal */
                                state = flags = given_wd = 0;
                        else
                                flags |= PR_LJ;
                        break;
                }
/* not a flag char: advance state to check if it's field width */
                state++;
/* check now for '%0...' */
                if(*fmt == '0')
                {
                        flags |= PR_LZ;
                        fmt++;
                }
                /* FALL THROUGH */
/* STATE 2: AWAITING (NUMERIC) FIELD WIDTH */
        case 2:
                if(*fmt >= '0' && *fmt <= '9')
                {
                        given_wd = 10 * given_wd +
                                (*fmt - '0');
                        break;
                }
/* not field width: advance state to check if it's a modifier */
                state++;
                /* FALL THROUGH */
/* STATE 3: AWAITING MODIFIER CHARS (FNlh) */
        case 3:
                if(*fmt == 'F')
                {
                        flags |= PR_FP;
```

```
                        break;
                }
                if(*fmt == 'N')
                        break;
                if(*fmt == 'l')
                {
                        flags |= PR_32;
                        break;
                }
                if(*fmt == 'h')
                {
                        flags |= PR_16;
                        break;
                }
/* not modifier: advance state to check if it's a conversion char
*/
                state++;
                /* FALL THROUGH */
/* STATE 4: AWAITING CONVERSION CHARS (Xxpndiuocs) */
        case 4:
                where = buf + PR_BUFLEN - 1;
                *where = '\0';
                switch(*fmt)
                {
                case 'X':
                        flags |= PR_CA;
                        /* FALL THROUGH */
/* xxx - far pointers (%Fp, %Fn) not yet supported */
                case 'x':
                case 'p':
                case 'n':
                        radix = 16;
                        goto DO_NUM;
                case 'd':
                case 'i':
                        flags |= PR_SG;
```

```
                        /* FALL THROUGH */
                case 'u':
                        radix = 10;
                        goto DO_NUM;
                case 'o':
                        radix = 8;
/* load the value to be printed. l=long=32 bits: */
DO_NUM:                         if(flags & PR_32)
                                num = va_arg(args, unsigned long);
/* h=short=16 bits (signed or unsigned) */
                        else if(flags & PR_16)
                        {
                                if(flags & PR_SG)
                                        num = va_arg(args, short);
                                else
                                        num =va_arg(args,unsigned short);
                        }
/* no h nor l: sizeof(int) bits (signed or unsigned) */
                        else
                        {
                                if(flags & PR_SG)
                                        num = va_arg(args, int);
                                else
                                        num = va_arg(args, unsigned int);
                        }
/* take care of sign */
                        if(flags & PR_SG)
                        {
                                if(num < 0)
                                {
                                        flags |= PR_WS;
                                        num = -num;
                                }
                        }
/* convert binary to octal/decimal/hex ASCII
OK, I found my mistake. The math here is _always_ unsigned */
```

```
                        do
                        {
                              unsigned long temp;
                              temp = (unsigned long)num % radix;
                              where--;
                              if(temp < 10)
                                    *where = temp + '0';
                              else if(flags & PR_CA)
                                    *where = temp - 10 + 'A';
                              else
                                    *where = temp - 10 + 'a';
                              num = (unsigned long)num / radix;
                        }
                        while(num != 0);
                        goto EMIT;
                  case 'c':
/* disallow pad-left-with-zeroes for %c */
                        flags &= ~PR_LZ;
                        where--;
                     *where = (unsigned char)va_arg(args, unsigned char);
                        actual_wd = 1;
                        goto EMIT2;
                  case 's':
/* disallow pad-left-with-zeroes for %s */
                        flags &= ~PR_LZ;
                        where = va_arg(args, unsigned char *);
EMIT:                   actual_wd = strlen(where);
                        if(flags & PR_WS)
                              actual_wd++;
/* if we pad left with ZEROES, do the sign now */
                   if((flags & (PR_WS | PR_LZ)) ==(PR_WS | PR_LZ))
                        {
                              fn('-', &ptr);
                              count++;
                        }
/* pad on left with spaces or zeroes (for right justify) */
```

```
EMIT2:              if((flags & PR_LJ) == 0)
                    {
                          while(given_wd > actual_wd)
                          {
                                fn(flags & PR_LZ ? '0' :
                                      ' ', &ptr);
                                count++;
                                given_wd--;
                          }
                    }
/* if we pad left with SPACES, do the sign now */
                    if((flags & (PR_WS | PR_LZ)) == PR_WS)
                    {
                          fn('-', &ptr);
                          count++;
                    }
/* emit string/char/converted number */
                    while(*where != '\0')
                    {
                          fn(*where++, &ptr);
                          count++;
                    }
/* pad on right with spaces (for left justify) */
                    if(given_wd < actual_wd)
                          given_wd = 0;
                    else given_wd -= actual_wd;
                    for(; given_wd; given_wd--)
                    {
                          fn(' ', &ptr);
                          count++;
                    }
                    break;
               default:
                    break;
               }
          default:
```

```
                state = flags = given_wd = 0;
                break;
            }
        }
        return count;
    }
```

5.5 Driver Routines

5.5.1 Floppy Disk Driver

```
/*
 * fd.c - floppy disk handling
 */

/*============================================================
EXPORTS:
void fd_init(void)
============================================================*/

// IMPORTS from MAIN.C:
void kprintf(const char *fmt, ...);

#include <x86.h> /* for outportb(), inportb() */

typedef unsigned char byte;
typedef unsigned char uchar;
typedef unsigned short ushort;
typedef unsigned long ulong;
typedef unsigned int uint;

typedef unsigned int size_t;
#define _SIZE_T

#ifndef NULL
#define NULL ( (void*)0 )
#endif
#endif
static uchar version;
static uchar cmos_type;
static uchar type;

static uchar motor_mask = 0;
```

```
static uint cur_op = 0;
static uint cur_drv;
static uint cur_head;
static uint cur_trk;

#define FD_SEEK   0x02

#define     NFD      (2) // Number of drives supported

#define     FD_BASEIO   0x3F0 // Base Address of FDC

// Commands
#define     F_MASTER       0x80
#define     F_DIR          0x40
#define     F_CMDBUSY   0x10
#define     FD_DATA     0x05 // Data FIFO Register
#define     FD_STATUS   0x04 // Main Status Register

/*
 * fdc commands
 */
#define     FDC_VERSION    0x10
#define     FDC_INTS       0x08  /* SENSEI */
#define     FDC_RESET      0x04  /* SENSE */
#define     FDC_SEEK       0x0F

#define     FDC_VER_765A   0x80
#define     FDC_VER_765B   0x90

#define     FDTYPE_MASK 0x0F

#define     FDTYPE_NONE 0x00
#define     FDTYPE_360  0x01
#define     FDTYPE_1200 0x02
#define     FDTYPE_720  0x03
#define     FDTYPE_1440 0x04
```

```
#define     FDTYPE_2880 0x05
#define     ENABLE_INTS 0x0C
#define     FD_MOTOR       0x02
#define     MOTOR_0     0x10
#define     MOTOR_1     0x20
#define     RTCSEL      0x70
#define     RTCDATA     0x71
#define     NVRAM_FD       0x10

uchar fd_in() // reads a byte from FIFO register
{
   uint c;
   int r;

   c = 1000000;

   do {
      r = (inportb(FD_BASEIO+FD_STATUS)
         & (F_MASTER | F_DIR | F_CMDBUSY));
      if( r == (F_MASTER | F_DIR | F_CMDBUSY ))
         break;
      c--;
   } while(c);

   if( c == 0 )
      return 0;

   return inportb(FD_BASEIO+FD_DATA);
}

void fd_out(byte val) // writes a byte from FIFO register
{
   uint c;
   int r;
   c = 1000000;
   do {
```

```
      r = (inportb(FD_BASEIO+FD_STATUS) & (F_MASTER | F_DIR));
      if( r == F_MASTER)
         break;
      c--;
   } while(c);

   if( c == 0 ) {
      kprintf("fd_out: timeout\n");
      return;
   }
   outportb(FD_BASEIO+FD_DATA, val);
}

void fd_reset() // Reset Drive
{
   int c;
   disable();

   outportb(FD_BASEIO+FD_MOTOR, 0);
   outportb(FD_BASEIO+FD_MOTOR, ENABLE_INTS);

   enable();

   for(c = 0; c < 10000; c++);
}

void fd_start(int drive) // activate motor of floppy drive
{
   motor_mask |= drive;
   if( drive == 0x00 ) {
      motor_mask |= MOTOR_0;
   } else {
      motor_mask |= MOTOR_1;
   }
   outportb(FD_BASEIO+FD_MOTOR, motor_mask);
}
```

```
void fd_stop(int drive) // Stop motor
{
   if( drive == 0x00 ) {
      motor_mask &= ~MOTOR_0;
   } else {
      motor_mask &= ~MOTOR_1;
   }
   outportb(FD_BASEIO+FD_MOTOR, motor_mask);
}
void fd_seek() // seek to a physical location in the drive
{
   cur_op = FD_SEEK;
   cur_head = 0;
   cur_trk = 0;
   cur_drv = 0;

   fd_out(FDC_SEEK);
   fd_out((cur_head << 2) | cur_drv);
   fd_out(cur_trk);
}
void fd_init() // Initializes Floppy disk
{
   int c;
   uint l;
   kprintf("\nInitializing FDC: \n");
   fd_reset();
   fd_out(FDC_VERSION);
   version = fd_in();
   kprintf("Version %d found: ", version);

   if( version == FDC_VER_765A )
      kprintf("stardard FDC found\n");
   else
   if( version == FDC_VER_765B )
      kprintf("enhanced FDC found\n");
   else
```

```
   if( version == 0 )
      kprintf("timeout\n");
   else
      kprintf("unknown controller\n");

   for(c = 0; c < NFD; ++c) {
      outportb(RTCSEL, NVRAM_FD);
      cmos_type = inportb(RTCDATA);

      type = (cmos_type >> (4 * (NFD - 1 - c ))) & FDTYPE_MASK;

      switch(type) {
         case FDTYPE_1200:
            kprintf("\tfd_init   ->   %d   :   1.2MB   floppy   drive
found\n",c);
            break;
         case FDTYPE_1440:
            kprintf("\tfd_init   ->   %d   :   1.44MB   floppy   drive
found\n",c);
            break;
         default:
            kprintf("\tfd_init -> %d : no drive\n",c);
            break;
      }
   }
   motor_mask |= (FDC_INTS | FDC_RESET);
   fd_start(0);
   for(l = 0; l < 1000000; l++);
   fd_stop(0);
}
```

Chapter 6 Future Work

The operating system developed is a very preliminary one with a simple command interpreter that interprets some basic commands. Lots of enhancements can be done based on the existing work to make it a fully grown operating system.

1. Firstly, we have not implemented any file system interface and basic file handling operations. So, the user cannot browse through the existing files on the disk. With appropriate modifications and with the help of the file system drivers, the kernel can be made to identify the file system and do basic file handling operations.
2. No CPU scheduling policy has been implemented; neither proper memory, IO & disk management modules are present. These are definitely areas which needs a lot of work.
3. Proper and well defined API's must be provided so that user level applications for the system can be designed.
4. We can write device drivers for other devices such as mouse, CD ROM and even printer, which give greater functionality to the operating system and greater user friendliness.
5. Certain sound functions through the PC speaker can also be incorporated so as to alert the user at certain critical command executions and errors.
6. Greater number of system processes can be incorporated, like processes that inform the user of any anomaly in the system, processes that do the garbage collection, processes that handle some other devices like the printer etc and these processes can be synchronized using semaphores or message based protocols.

Bibliography

[1] Protected mode and operating systems:

http://www.execpc.com/~geezer/os/

[2] OS Development For Dummies (OSD)

http://www.execpc.com/~geezer/osd/index.htm

[3] Write Your Own Operating System

http://www.mega-tokyo.com/os/os-faq.html

[4] Ralf Brown's Files

http://www.cs.cmu.edu/afs/cs.cmu.edu/user/ralf/pub/WWW/files.html,

http://www.ctyme.com/rbrown.htm

[5] OSD PC bootstrap process

http://www.execpc.com/~geezer/osd/boot/index.htm#grub

[6] My 'FAQ' for amateur OS Designers

http://www.overwhelmed.org/shawn/faq.html

[7] OSD Text-mode console output for VGA

http://www.execpc.com/~geezer/osd/cons/index.htm#vmm

[8] Interrupt Vector Table

http://www.geocities.com/electronic_ed/tae/appd/ivt.htm

[9] Inline asm with GCC

http://www.delorie.com/djgpp/doc/brennan/brennan_att_inline_djgpp.html

[11] DJGPP can be downloaded freely from the following website:

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

[12] Operating Systems Resource Center

http://www.nondot.org/sabre/os

[13] OSRC Disk and Disc Drives

http://www.nondot.org/sabre/os/articles/DiskandDiscDrives/

[14] MMURTL V1 by Richard A. Burgess

http://www.sensorypublishing.com/mmurtl.html

[15] Intel Programmers Reference Manuals

http://www.microsym.com/386intel.pdf

or the three original Intel manuals in PDF:-

http://kos.enix.org/pub/intelvol1.pdf.gz (Manual Vol: 1)

http://kos.enix.org/pub/intelvol2.pdf.gz (Manual Vol: 2)

http://kos.enix.org/pub/intelvol3.pdf.gz (Manual Vol: 3)

[16] Intel Floppy Disk Controller, 8272A Manual

http://www.nondot.org/sabre/os/files/Disk/82077AA_FloppyControllerDatasheet.pdf

http://www.nondot.org/sabre/os/files/Disk/FLOPPY.TXT

[17] Intel 8259 Programmable Interrupt Controller Manual

http://satya.virtualave.net/8259.html

or the original Intel Manual in PDF:

ftp://download.intel.com/support/controllers/peripheral/231468.pdf