OS Development

Luke

December 12, 2024

Abstract

This document serves as a reference for the Minimus operating system.

Contents

1 Bootloader

1.1 Headers

The disk is ordered into sectors, of size 200_{16}, starting from sector 1[3]. When the BIOS loads the OS, it copies the first sector, which would be the first 512 bytes, from the disk into memory, starting at $7C00_{16}$[1].

Memory is also ordered into segments, of size 10_{16}[4]. This means that memory addresses can overlap, for example: The address $0000{:}7C00_{16}$ is the same as $0700{:}0C00_{16}$.

You must ensure that the bytes at $1FE_{16}$ and $1FF_{16}$ contain values 55_{16} and AA_{16} respectively[5]. This is called the magic number, and is used to differentiate bootable disks from non-bootable disks.

When the BIOS hands control to the OS, the CPU will be in 16 bit (real) mode[6], which means it is using 16 bits per instruction. This is because the CPU is in 16 bit mode by default. You must ensure that your program code for the bootloader begins in 16 bit (assuming real) mode.

bootloader/main.asm

```
[org 0x7c00]                    ; memory load location
[bits 16]                   ; real mode
```

1.2 Disk read

Reading from the disk is done with Bios Interrupt 13_{16} ah=02[2].

The interrupt specification is layed out below.1:

Register	Value
AH	02
AL	Number of sectors to read
CH	Cylinder
DH	Head
CL	Sector
DL	Drive
ES:BX	Output offset

Since the DL register is initialised with the boot drive before control is handed to our program[7], as long as it is not overwritten before calling int 13_{16}, you do not have to alter it.

Since the ES register cannot be written to directly, due to no CPU instruction being availible to transfer a value to ES[8], which results in an intemediary value needing to be used.

I decided to use BX, due to it being overwritten used in the instruction after, but any unused register will do. Then, I assigned BX with $7e00_{16}$, which is 200_{16} bytes more than the start of the program[1], which is the precise number of bytes loaded by the BIOS[3].

The number of load segments (register AX)[2] have a limit of around 100 on some systems, and around 70 on QEMU. 64 is a safe number of sectors to load, and allows for 32KB of kernel instructions (which is far more than enough).

Sometimes there will be an error, stored in the carry bit[2], if this happens, retry the load from disk. An error may also be indicated by the AL register having an incorrect number of sectors read.

bootloader/main.asm

```
KERNEL_SEGS equ 64        ; KERNEL_SEGS * 512

; read kernel (https://en.wikipedia.org/wiki/INT_13H)
mov bx, 0
mov es, bx
mov bx, 0x7e00                ; offset
mov ah, 0x02                ; set read mode
mov cl, 2                 ; start from sec 2
mov al, KERNEL_SEGS         ; sectors to read
mov ch, 0                 ; cylinder
mov dh, 0                 ; head
int 0x13                 ; call
jc $$                       ; carry bit stores error, loop
cmp al, KERNEL_SEGS         ; al is sectors read
jne $$                        ; if all sectors arent read, loop
```

1.3 Usable memory

The amount of usable memory varies between systems, and some memory is reserved for hardware, VGA and VBE, console video memory, etc.

Detecting availible memory blocks is done with interrupt 15_{16}[12].3. Although you can probably get away with ignoring this, and just hoping memory above, lets say 1mb, is completely fine, I went through the extra effort to correctly get usable memory.

This area repeatedly calls int 15_{16}, and saves the result, if it is larger than the currently tested memory, in a memory location ironically not tested to see if it is reserved.

The copying and checking is done in two parts due to the fact that the numbers are larger than the 16 bit memory availible in real mode.

bootloader/main.asm

```
; get largest availible memory block
pusha                       ; push all
mov cx, 0x0                 ; clear cx for addition later
```

```
xor eax, eax
mov es, eax
mov ebx, 0x0                ; clear
mov edx, 0x534d4150         ; magic value
memreadloop:
        mov eax, 0xe820             ; magic value
        mov ecx, 0x18               ; magic value
        mov di, 0x7bd0              ; memory location for buffer
        int 0x15                ; call function
        add di, cx      ; increment di by entry size
                        ; (cx is 16 bit cl)
memreadloopvalid:
        mov eax, [0x7be0]     ; load type
        cmp eax, 1              ; check if 1 (availible memory)
        jne memreadloopend      ; go to next otherwise
memreadloopcheck:
        mov eax, [0x7bda]         ; load size of current
        mov ecx, [0x7bfa]         ; load size of biggest
        cmp eax, ecx                ; check if bigger
        jle memreadloopend         ; go to next otherwise
memreadlooprecord:
        mov eax, [0x7bd0]         ; load address of current
        mov [0x7bf0], eax         ; record biggest address
        mov eax, [0x7bd4]         ; load address of current
        mov [0x7bf4], eax         ; record biggest address
        mov eax, [0x7bd8]         ; load size of current
        mov [0x7bf8], eax         ; record biggest size
        mov eax, [0x7bdc]         ; load size of current
        mov [0x7bfc], eax         ; record biggest size
memreadloopend:
        cmp ebx, 0                 ; check if next
        jnz memreadloop                  ; repeat
```

1.4 Display

1.4.1 VGA

Enabling VGA[14] here allows me to get the font later, and also serves as compatibility for if VESA is not supported.

The options for AL are defined.4.

bootloader/main.asm

```
; vga mode
mov ah, 0x0                 ; graphics mode
```

```
mov al, 0x13                ; 256 colour 200x320
int 0x10                ; set vga mode
```

1.4.2 Font

The code below gets the font from the VGA card. This is so I did not have to store my own.

The code is taken mostly from this OsDev page[13], due to the fact that I figured it was mostly copy paste anyway.

`bootloader/main.asm`

```
; get vga font
mov eax, 0x100
mov es, eax
mov ax, 0x0
mov di, ax
push ds
push es
mov ax, 0x1130                ; magic numbers
mov bh, 0x6
int 0x10                ; get vga font
;mov ds, es
push es
pop ds
pop es
mov si, bp
mov cx, 0x400
rep movsd
pop ds
```

1.4.3 VBE

The VESA bios extensions allows for greater colour depth (RGB) and higher resolutions than VGA. Getting the VBE table is done with interrupt 10_{16}[15].

Once sifted to find the right function, it can again be enabled with interrupt 10_{16}, the magic numbers for each function are in the VBE spec[15].

The resulting values (whether 24 bit or 32 bit colour is used, the location of the framebuffer, whether VBE is even supported in the first place, etc), are placed into another (not checked) area of memory, which will be used by the kernel later.

I start by checking for VBE2 support, and if it doesn't exist I jump to the end of the VBE section. Then I go through each function, checking if it has RGB colourspace, and is the correct resolution. I then update the bytes at some position (again not fully checked) so my kernel has access to whether VBE was enabled or not, and also to provide the kernel with the position of the linear framebuffer.

This is due to the fact that in VBE2, it is not required that all the VBE1 functions.5 are supported (even though they most likely are).

Enabling a VBE mode is as simple as VBE once the functions have been sorted, and is the same BIOS interrupt.6.

`bootloader/main.asm`

```
; get vesa support
mov [0x2000], DWORD "VBE2"
mov ax, 0x4f00                  ; magic number
mov di, 0
mov es, di
mov di, 0x2000                  ; offset to table
int 0x10                  ; get vbe table
cmp ax, 0x004f                  ; check if support
jne skip_vbe                ; use vga instead if not

mov di, 0                  ; offset to tmp
mov es, di
mov di, 0x2200                  ; segment to tmp
mov si, [0x2000 + 16]         ; segment of list
mov ds, si
mov si, [0x2000 + 14]         ; offset of list

vbe_loop:
; get next supported vesa function
movsw
sub di, 2                  ; (2 is added to both)
mov cx, [0x2200]         ; vesa mode
cmp cx, 0xffff                  ; check if end
je skip_vbe                ; loop

; get details
mov ax, 0x4f01                  ; magic number
int 0x10                  ; get vbe

mov ax, [0x2200]         ; check attr
and ax, 0b10000000         ; check bit 7 (linear framebuffer)
cmp ax, 0
jz vbe_loop                ; if not linear, loop

mov al, [0x2200 + 25]         ; check colourspace
mov [0x2201], BYTE 0x00         ; vbe 32 bit flag
cmp al, 24
```

```
je vbe_check_dims        ; if not rgb, loop
mov [0x2201], BYTE 0xff        ; vbe 32 bit flag
cmp al, 32
je vbe_check_dims        ; if not rgb, loop
jmp vbe_loop                ; actual loop

vbe_check_dims:
mov ax, [0x2200 + 18]        ; check width
cmp ax, 640
jne vbe_loop                ; if not desired, loop

mov ax, [0x2200 + 20]        ; check height
cmp ax, 480
jne vbe_loop                ; if not desired, loop

jmp enable_vbe                ; turn on this vbe

skip_vbe:
mov [0x2200], BYTE 0x00        ; vbe corect flag
jmp complete_vbe        ; skip

enable_vbe:
mov ax, [0x2200 + 40]        ; framebuffer
mov [0x2200 + 2], ax        ; framebuffer
mov ax, [0x2200 + 42]        ; framebuffer
mov [0x2200 + 4], ax        ; framebuffer
mov ax, 0x4f02                ; magic number
mov bx, cx                ; move mode number
or bx, 0x4000                ; set linear framebuffer
int 0x10                ; set vbe mode
cmp ax, 0x004f                ; test for error
jne skip_vbe                ; skip if error (will use vga instead)
mov [0x2200], BYTE 0xff        ; vbe correct flag

complete_vbe:
popa                        ; pop all
```

1.5 Enabling 32-bit processing

1.5.1 Segment descriptor

The Global Descriptor Table is necessary for enabling protected mode[9], and is defined below in a basic form in order to switch to protected mode.

For my kernel, none of the features of the GDT, like paging[11], are not needed, so the GDT is very basic.2.

bootloader/main.asm

```
jmp gdt_after

; segment descriptor (reverse order)
gdt_start:
        dq 0                ; null byte start
gdt_code:
        dw 0xffff         ; segment limit
        db 0,0,0         ; segment base
        db 0b10011010         ; flags (see wiki)
        db 0b11001111         ; 4b flags (see wiki) + seg limit
        db 0                ; segment base
gdt_data:
        dw 0xffff         ; segment limit
        db 0,0,0         ; segment base
        db 0b10010010         ; flags (see wiki)
        db 0b11001111         ; 4b flags (see wiki) + seg limit
        db 0                ; segment base
gdt_end:
        dw gdt_end - gdt_start - 1         ; limit
        dd gdt_start                         ; addr 24 bit
gdt_after:
```

1.5.2 Prerequisites

To enable protected mode, you must first disable interrupts, and load the global descriptor table[16], which can be done in NASM with commands `cli` and `lgdt [GDT ADDRESS]` respectively.

bootloader/main.asm

```
cli                          ; disable interrupts
lgdt [gdt_end]                  ; gdt_end is descritor table
↪  descriptor
```

1.5.3 Protected mode

Upon the completion of all the above, you can then set bit 0 (starting from 0) to 1 in the control register CR0[16], which must be done in seperate commands, as the special register CR0 is not directly assignable[8].

You must then immediately[16] perform a long jump to your next desired instruction.

You then have to tell the assembler that you are now using 32 bits, which can be done with the `[bits 32]` command in NASM.

bootloader/main.asm

```
mov eax, cr0
or eax, 1                  ; set 1 bit in control register for protected mode
mov cr0, eax
jmp (gdt_code - gdt_start):bits32code        ; stall cpu and flush all cache (as moving to different segment) to finalize protected mode

; finally 32 bits
[bits 32]
bits32code:
```

1.6 Enabling the A20 line

Enabling the A20 line (the proper way) is by the keyboard[10]. This is incredibly tedious and repetitive, which is why I have taken this code from the OsDev page.

Two small assembly functions: a20wait; a20waitr, have been created to avoid some of the repetition. a20wait will repeatedly poll port 64_{16} until bit 1 is set (which is why `TEST` is used). a20waitr will do the same until bit 0 is set.

This is due to the fact that it is required to wait for bit 1 in port 64_{16} before writing and bit 0 in port 64_{16} before reading.

A brief outline of the functions of the A20 line are provided within this document.7.

bootloader/main.asm

```
xor al, al
call a20wait                ; wait for write
mov al, 0xad
out 0x64, al                ; send 0xad
call a20wait                ; wait for write
mov al, 0xd0
out 0x64, al                ; send 0xd0
call a20waitr                ; wait for read
in al, 0x60                ; get ack
push eax
call a20wait                ; wait for write
mov al, 0xd1
out 0x64, al                ; send 0xd1
call a20wait                ; wait for write
pop eax                        ; eax gets overwritten
or al, 0b0010                ; a20 bit
out 0x60, al                ; set a20 bit on
call a20wait                ; wait for write
```

```
mov al, 0xae
out 0x64, al                    ; send 0xae
call a20wait                    ; wait for generic
jmp skipa20                    ; go to end
a20wait:
        in al, 0x64
        test al,2
        jnz a20wait
        ret
a20waitr:
        in al, 0x64
        test al,1
        jz a20waitr
        ret

skipa20:
```

1.7 Loading the Kernel

Before loading the kernel, I initialise all the registers high parts with the segment and move the stack pointers to 7000_{16}.

The bootloader begins at $7c00_{16}$[1], and because the stack grows downwards, 7000_{16} is a perfect location for a stack. You can place yours where you like.

bootloader/main.asm

```
jmp (gdt_code - gdt_start):start_kernel

start_kernel:
        ; segment registers init
        mov ax, gdt_data - gdt_start
        mov ds, ax
        mov es, ax
        mov fs, ax
        mov gs, ax
        mov ss, ax

        ; stack pointers
        mov esp, 0x7000                 ; top of stack
        mov ebp, esp                 ; bottom of stack

        call kernel                 ; start kernel and move back to segment

; kernel
kernel:
```

```
        jmp kernel_loadseg        ; hand control to kernel
        jmp $$                        ; return -> error, loop

                        ...

; kernel load
kernel_loadseg:
call kernel_cseg
jmp $                                ; if fail restart
kernel_cseg:                        ; compiled c appeneded here
```

1.8 The boot signature

You must ensure that the bytes at $1FE_{16}$ and $1FF_{16}$ contain values 55_{16} and AA_{16} respectively[5]. This is called the boot signature, or magic number, and is used to differentiate bootable disks from non-bootable disks, and also tells the BIOS how to load your OS.

`bootloader/main.asm`

```
; padding
times 510 - ($-$$) db 0

; boot signature
db 0x55,0xaa
```

2 Kernel

The kernel is mostly written in C, with occasional exceptions like the IDT, which is written in NASM assembly.

2.1 Port Utils

Many kernel functions require reading and writing data to the serial bus, so I have made functions for code readibility.

GCC inline assembly has three major parts: the instruction - which is written in GNU assembly; the input value - (e.g. ”=a”(val) to save to val); the output value - (e.g. ”a”(val) to load from val), all seperated by colons.

`kernel/libs/ioutils.c`

```c
void outb(unsigned short port, unsigned char val) {
	__asm__ volatile ("outb %b0, %w1" : : "a"(val), "Nd"(port) : "memory");
}

unsigned char inb(unsigned short port) {
	unsigned char _ret = 0;

	__asm__ volatile ("inb %w1, %b0" : "=a"(_ret) : "Nd"(port) : "memory");

	return _ret;
}

void outw(unsigned short port, unsigned short val) {
	__asm__ volatile ("outw %w0, %w1" : : "a"(val), "Nd"(port) : "memory");
}

unsigned short inw(unsigned short port) {
	unsigned short _ret = 0;

	__asm__ volatile ("inw %w1, %w0" : "=a"(_ret) : "Nd"(port) : "memory");

	return _ret;
}

void iowait() {
	outb(0x80, 0);
}
```

2.2 Memory management

2.2.1 The heap

The two most common ways to manage memory are the stack and the heap. We already have a stack, which is defined as 7000_{16}, and grows downwards, but we need a way to access more, and larger, memory, at random.

I had already wrote a heap before, but had used the concept of object orientation, due to the fact I wrote it in my (then and still now) favourite language C++, which I cannot stop glazing (I believe my opinion may change after I try Rust).

The heap is a large area of semi-organised data. The way I created my heap, it is organised into blocks, and each block points to both the previous and next block. As well as this, the size, and whether the block is occupied is stored.

When giving memory, I simply add the size of `memchunk` onto the pointer. I decided that no space for overflowing values will be required for this kernel.

The main functions, excluding the utility functions and those required by the GCC compiler, are defined below, including how I implemented my heap.

kernel/libs/vga.c

```c
#include "ioutils.h"

struct memchunk {
	struct memchunk* prev;
	struct memchunk* next;
	unsigned long size;
	char occupied;
};

struct memchunk* heap;
void initheap() {
	heap = (struct memchunk*)(*(unsigned long*)0x7bf0);
	heap->size = *(unsigned long*)0x7bf8;
	heap->occupied = 0;
}

void* malloc(unsigned int size) {
	struct memchunk* check = heap;
	do {
		if (check->size > size + 16 && !check->occupied) {
			struct memchunk* nextchunk = (struct memchunk*)(check + size + 16);

			nextchunk->size = check->size - size - 16;
```

```c
                        nextchunk->prev = check;
                        nextchunk->next = check->next;
                        nextchunk->occupied = 0;

                        check->next = nextchunk;
                        check->size = size;
                        check->occupied = 1;

                        return check + 16;
                }

                check = check->next;
        } while (check);
        return 0;
}

struct memchunk* m_memchunkstartfreesegment(struct memchunk* check) {
        if (check->prev && !check->prev->occupied)
                return m_memchunkstartfreesegment(check->prev);
        return check;
}

int free(void* ptr) {
        struct memchunk* check = ptr - 16;

        if ((check->prev && check->prev->next != check) || (check->next && check->next->prev != check)) {
                return 1;
        }

        check->occupied = 0;

        check = m_memchunkstartfreesegment(check);
        while (check->next && !check->next->occupied) {
                check->size += check->next->size;
                check->next = check->next->next;
        }
        if (check->next)
                check->next->prev = check;

        return 0;
}

void* realloc(void* ptr, unsigned int size) {
        struct memchunk* check = ptr - 16;
```

```c
    if (check->next && check->next->size >= size -
    ↪  check->size) { // 16 bit excluded as it is on both
    ↪  sides
        check->size += check->next->size;
        check->next = check->next->next;
        return ptr;
    }

    // slower solution if next memory block isnt free
    void* _ptr = malloc(size);

    for (unsigned int i = 0; i < size; i++) {
        *((char*)_ptr+i) = *((char*)ptr+i);
    }

    free(ptr);

    return _ptr;
}
```

2.3 VGA/VBE

VGA is outdated, so is only used as a backup in this OS, in case VBE2 fails to load for whatever reason, or is not supported.

2.3.1 Initialisation

VGA memory is set at $A0000_{16}$, whereas VBE uses a variable position linear framebuffer. In the bootloader, when VBE2 was enabled, the linear framebuffer was stored at 2200_{16}, so it is loaded from there.

Additionally, whether VBE was sucessfully enabled is loaded, as well as how many bits per pixel are used (on some systems it is 4).

The width and height is pre-defined for this kernel, so is not needed to be gathered from anywhere.

For compatibility, the VGA colourspace is loaded with a compressed version of RGB (for 8 bits: RRRGGGBB).

kernel/libs/vga.c

```c
#include "ioutils.h"

extern void getvgafont();

#define VGA_MEM (unsigned char*)0xa0000
#define VGA_WIDTH 320
#define VGA_HEIGHT 200
```

```c

#define VBE_WIDTH 640
#define VBE_HEIGHT 480
unsigned char vbeEnabled = 0;
unsigned char vbeAlpha = 3;
unsigned char* vbeFramebuffer = (unsigned char*)0x0;

unsigned char* font = (unsigned char*)0x1000; // defined in bootloader/main.asm

void initvga() {
	vbeEnabled = *(unsigned char*)0x2200;
	if (*(unsigned char*)0x2201)
		vbeAlpha = 4;
	else
		vbeAlpha = 3;

	if (vbeEnabled) {
		vbeFramebuffer = (unsigned char*)(*(unsigned int*)0x2202);
	}
	else {
		for (int i = 0; i < 256; i++) {
			unsigned char r = (i & 0b11100000) >> 2; // highest value is 0b00111111 as 18 bit colour space
			unsigned char g = (i & 0b00011100) << 1;
			unsigned char b = (i & 0b00000011) << 4; // used less bits for blue as eyes are less sensitive

			outb(0x3c8, i); // set DAC address
			outb(0x3c9, r); // set DAC R for i
			outb(0x3c9, g); // set DAC G for i
			outb(0x3c9, b); // set DAC B for i
		}
	}
}
```

2.3.2 Setting pixel values

Setting pixel values in VGA and VBE are similar, with the exception that in VBE, you may need to have an extra alpha bit that is left alone. I handled this case as below.

In VGA, I have compressed the colourspace down to the values allowed, and while this looks low quality, it serves as a great fallback for old systems with very little overhead.

`kernel/libs/vga.c`

```c
void drawpixel(int _x, int _y, unsigned char r, unsigned char g, unsigned char b) {
	if (vbeEnabled) {
		*(vbeFramebuffer + _x * vbeAlpha + vbeAlpha - 3 + _y * VBE_WIDTH * vbeAlpha) = b;
		*(vbeFramebuffer + _x * vbeAlpha + vbeAlpha - 2 + _y * VBE_WIDTH * vbeAlpha) = g;
		*(vbeFramebuffer + _x * vbeAlpha + vbeAlpha - 1 + _y * VBE_WIDTH * vbeAlpha) = r;
	}
	else {
		_x /= 2;
		_y /= 3;
		*(VGA_MEM + _x + _y * VGA_WIDTH) = (r & 0b11100000) | ((g >> 3) & 0b00011100) | ((b >> 6) & 0b00000011);
	}
}
```

2.3.3 Drawing characters

For drawing characters, you need a font. Although the kernel currently has a filesystem, at the time I implemented the display, it did not.

I decided to append the bootloader's VGA code with code to load the VGA BIOS currently used font (as it was easier to do this with BIOS interrupts).

This code is covered in the Bootloader chapter, so I will not be covering this. The pointer has been assigned to the `font` variable.

`kernel/libs/vga.c`

```c
void drawchar(int _x, int _y, unsigned char _char) {
	unsigned char* fontchar = font + ((int)_char * 16);

	for (int y = 0; y < 16; y++) {
		for (int x = 0; x < 8; x++) {
			if (fontchar[y] & (0b10000000 >> x)) {
				drawpixel(_x * 8 + x, _y * 16 + y, 0xff, 0xff, 0xff);
			}
			else {
				drawpixel(_x * 8 + x, _y * 16 + y, 0x00, 0x00, 0x00);
```

```
                        }
                }
        }
}
```

2.4 Keyboard Input

Getting input from the keyboard, mouse, clock, and others is done through the Program Interface Controller, which is a way of sending immediate jobs to the CPU, called interrupts.

2.4.1 Interrupt Descriptor Table

The Interrupt Descriptor Table[17] is an array of values, where each index corresponds to an interrupt, and each value the memory location of that interrupt function.

kernel/libs/interrupts.c

```
global initidtasm
global interrupthandler
global idtdescriptor
global idtaddr

section .data

idtdescriptor:
        dw 256*8-1        ; size
idtaddr:
        dd 0x10000        ; address

section .text

initidtasm:
        lidt [idtdescriptor]
        ret

%macro idt 1
        extern idt%1
        global _idt%1
        _idt%1:
                pusha
                call idt%1
                popa
                iret
%endmacro
```

```

idt ...
```

kernel/libs/interrupts.c

```c
#include "ioutils.h"

void interrupthandler(unsigned char interrupt);

struct idtelement {
	unsigned short offset1; // offset 0-15
	unsigned short selector; // a code segment selector in gdt
	unsigned char base; // segment base (reserved) + ist
	unsigned char attr; // gate type, dpl, and leftmost bit must be 1
	unsigned short offset2; // offset 16-31
};

extern void* idtaddr;

// https://en.wikipedia.org/wiki/Interrupt_descriptor_table
void initidtelement(unsigned int num, unsigned int func, unsigned char trap) {
	struct idtelement* element = (struct idtelement*)(idtaddr+num*8);
	element->base = 0;
	element->selector = 8;
	element->attr = 0b10001110 | trap;
	element->offset1 = (unsigned short)(func);
	element->offset2 = (unsigned short)(func >> 16);
}
```

2.4.2 Program Interface Controller

The PIC is what sends the signals for the interrupt to the CPU, so once interrupts have been tested and are working, the PIC must be programmed.

The PIC should be set, to begin with, so that all interrupts (except the communication bus) are masked, and are unmasked as you need them. This is to save on processor time.

You must first disable interrupts, then send the ICW1 signal to both the slave and master PIC (port 20_{16} and $A0_{16}$ respectively). Then, you must set the vector offsets (ICW2), inform the PIC chips of each others existence (ICW3), and finally, enable 8086 mode (ICW4)[17].

You are then able to set the mask such that only the chips can communicate, and enable the required PIC interrupts as you go along in your kernel.

kernel/libs/interrupts.c

```c
#define PIC1 0x20 // master pic
#define PIC2 0xa0 // slave pic

#define PIC1_OFFSET 0x20
#define PIC2_OFFSET (PIC1_OFFSET+8)

void sendeoi(unsigned char reg) {
        if (reg >= 8) {
                outb(PIC2, 0x20);
        }
        outb(PIC1, 0x20);
}

// https://wiki.osdev.org/8259_PIC#Programming_the_PIC_chips
void initpic() {
        // disable interrupts
        __asm__ volatile ("cli");

        // each wait allows PIC to process

        // set initialisation command for cascade (master and
        ↪  slave) (icw1)
        outb(PIC1, 0x11);
        iowait();
        outb(PIC2, 0x11);
        iowait();

        // set vector offsets (icw2)
        outb(PIC1+1, PIC1_OFFSET);
        iowait();
        outb(PIC2+1, PIC2_OFFSET);
        iowait();

        // inform master of slave pic (icw3)
        outb(PIC1+1, 0b0100);
        iowait();
        // inform slave of slave pic (icw3)
        outb(PIC2+1, 0b0010);
        iowait();

        // enable 8086 mode (icw4)
        outb(PIC1+1, 0b0001);
        iowait();
        outb(PIC2+1, 0b0001);
```

```
	iowait();

	// masks
	outb(PIC1+1, 0xff & ~(1 << 2));
	outb(PIC2+1, 0xff);

	// enable interrupts
	__asm__ volatile ("sti");
}

void enablepic(unsigned char irq) {
	unsigned short port = PIC1+1;

	if (irq >= 8) {
		port = PIC2+1;
		irq -= 8;
	}

	unsigned char val = inb(port) & ~(1 << irq); // remove mask bit (set 0)
	outb(port, val);
}

void disablepic(unsigned char irq) {
	unsigned short port = PIC1+1;

	if (irq >= 8) {
		port = PIC2+1;
		irq -= 8;
	}

	unsigned char val = inb(port) | (1 << irq); // add mask bit (set 1)
	outb(port, val);
}

// https://pdos.csail.mit.edu/6.828/2014/readings/hardware/8259A.pdf
// 4. INTERRUPT REQUEST REGISTER (IRR) AND IN-SERVICE REGISTER (ISR)
unsigned short irqreg(int cmd) {
	outb(PIC1, cmd);
	outb(PIC2, cmd);
	return (inb(PIC2) << 8) | inb(PIC1);
}

```

```c
#define idtbody(x, y) \
extern unsigned int _idt##y; \
void idt##y() { \
	interrupthandler(x); \
	sendeoi(x); \
} \

#define idthead(x, y) \
initidtelement(y, (unsigned int)&_idt##y, 0); \
enablepic(x); \
```

A helper function for the PIC is also defined, allowing easy binding of multiple functions to an interrupt.

`kernel/libs/pic.c`

```c
void (*picFunc[16][16])();

int addPicFunc(int pic, void (*func)()) {
	for (int i = 0; i < 16; i++) {
		if (!picFunc[pic][i]) {
			picFunc[pic][i] = func;
			return 0;
		}
	}
	return -1;
}

// interrupts.h
void interrupthandler(unsigned char interrupt) {
	for (int i = 0; i < 16; i++)
		if (picFunc[interrupt][i])
			picFunc[interrupt][i]();
}
```

2.5 File system

2.5.1 Reading from Disk

To read from the disk, you can switch back to real mode, or you can create a device driver.

I decided to create a simple ATA driver, as it would work with most types of disks (due to most being compatible with ATA).

There are many guides on ATA controllers, so I will not go into the details within this document. You could even use somebody else's, but I believe that the educational benefit of creating a device driver

The main principle of reading from the disk with ATA is that it is read in segments, and each segment is read at a certain time. You can figure out that time by either polling, or interrupts. I decided on polling as it is faster for now, but I may change the design later on.

`kernel/libs/disk.c`

```c
#include "ioutils.h"
#include "memory.h"

void diskWait() {
        for (int i = 0; i < 15; i++) {
                inb(0x1f7);
        }
        unsigned char status = inb(0x1f7);
        while ((status & 0b10000000 && !(status & 0b1000)))
                status = inb(0x1f7);
}

void* diskReadSector(unsigned int lba, unsigned char sectors) {
        unsigned short mallocSec = sectors;
        if (mallocSec == 0)
                mallocSec = 256;
        unsigned char* buffer = malloc(mallocSec * 512);

        outb(0x1f6, 0xe0 | ((lba >> 24) & 0x0f)); // drive and upper 4 bits of lba
        outb(0x1f1, 0); // ignored but necessary on some systems
        outb(0x1f2, sectors);
        outb(0x1f3, lba); // lower 8 bits
        outb(0x1f4, lba >> 8); // mid lower 8 bits
        outb(0x1f5, lba >> 16); // mid upper 8 bits
        outb(0x1f7, 0x20); // read flag

        for (int sector = 0; sector < mallocSec; sector++) {
                diskWait();

                for (int i = 0; i < 256; i++) {
                        *(unsigned short*)(buffer + sector * 512 + i * 2) = inw(0x1f0);
                }
        }

        return buffer;
}

```

```c
void diskWriteSector(unsigned int lba, unsigned char sectors,
↪  unsigned char* buffer) {
        unsigned short mallocSec = sectors;
        if (mallocSec == 0)
                mallocSec = 256;

        outb(0x1f6, 0xe0 | ((lba >> 24) & 0x0f)); // drive and
        ↪  upper 4 bits of lba
        outb(0x1f1, 0); // ignored but necessary on some systems
        outb(0x1f2, sectors);
        outb(0x1f3, lba); // lower 8 bits
        outb(0x1f4, lba >> 8); // mid lower 8 bits
        outb(0x1f5, lba >> 16); // mid upper 8 bits
        outb(0x1f7, 0x30); // write flag

        for (int sector = 0; sector < mallocSec; sector++) {
                diskWait();

                for (int i = 0; i < 256; i++) {
                        outw(0x1f0, *(unsigned int*)(buffer +
                        ↪  sector * 512 + i * 2));
                }
        }
}
```

2.5.2 Creating a pagefile

A pagefile is just a giant list of filenames and where that filename points to on the disk. It is similar to the implementation of the heap, except that all the list begins at the start, and is in one place. This saves us from having to check many different sectors on our disk for small pieces of fragmented information.

For this kernel, since the amount of files will be relatively low, I have decided to load all of the files into memory. This, although inefficient on memory, will have hardly any impact. For example, a thousand files (with reasonable filename sizes of 10 characters) will take up only 19kB, which is far less memory than is needed.

It is for this reason that I have decided to load all of the filepage into memory, as the cost of loading them in and out of memory is (probably) more than the kilobytes of memory you will save.

On user based operating systems, this cost isnt negligable, as they may have thousands of files, but I can easily change this code later on, so I have decided to do it this way.

kernel/libs/file.c

```c
#include "disk.h"
```

```c
#include "strutils.h"
#include "memory.h"

#define PAGEADDRDISK 0x5000

struct filePage {
        unsigned long address;
        unsigned long size;
        char name;
};
#define filePageSize 20

struct filePage* pageaddr;

struct filePage* fileDescriptor(char* filename) {
        struct filePage* ptr = pageaddr;

        while (ptr->address) {
                if (!strcmp(&(ptr->name), filename)) {
                        return ptr;
                }
                *((unsigned char*)&ptr) += strlen(&(ptr->name)) + filePageSize;
        }

        return (struct filePage*)0;
}

void* fileRead(char* filename) {
        struct filePage* descriptor = fileDescriptor(filename);

        if (descriptor)
                return diskRead(descriptor->address, descriptor->size);
        else
                return (void*)0;
}

void fileDelete(char* filename) {
        struct filePage* descriptor = fileDescriptor(filename);
        if (!descriptor)
                return;
        unsigned int len = strlen(&(descriptor->name)) + filePageSize;
        unsigned char* ptr = (unsigned char*)descriptor;

```

```c
        unsigned int conseqZero = 0;
        for (unsigned long i = 0; conseqZero < len; i++) {
                ptr[i] = ptr[i + len];
                if (ptr[i] == 0)
                        conseqZero++;
                else
                        conseqZero = 0;
        }
}

void fileWrite(char* filename, unsigned char* buffer, unsigned long size) {
        fileDelete(filename);

        struct filePage* descriptor = pageaddr;
        unsigned long descaddr = 0;

        while (descriptor->address) {
                descaddr = descriptor->address + descriptor->size;
                *((unsigned char*)&descriptor) += strlen(&(descriptor->name)) + filePageSize;
        }

        descriptor->address = descaddr;
        descriptor->size = size;
        memcpy(&(descriptor->name), filename, strlen(filename));

        diskWriteSector(PAGEADDRDISK / 512, 0, (void*)pageaddr);

        diskWrite(descriptor->address, size, buffer);
}

char** fileList() {
        int sizeTrack = sizeof(void*);
        char** wordList = malloc(sizeTrack);
        wordList[0] = (char*)0;

        struct filePage* ptr = pageaddr;

        while (ptr->address) {
                sizeTrack += sizeof(void*);
                wordList = realloc(wordList, sizeTrack);
                wordList[sizeTrack / sizeof(void*) - 2] = &(ptr->name);
                wordList[sizeTrack / sizeof(void*) - 1] = 0;
```

```c
                *((unsigned char*)&ptr) += strlen(&(ptr->name)) + filePageSize;
        }

        return wordList;
}

void initfs() {
        pageaddr = (struct filePage*)diskReadSector(PAGEADDRDISK / 512, 0);
        pageaddr->address = PAGEADDRDISK + 256 * 512;
        pageaddr->size = 0;
        pageaddr->name = 0;

        diskWriteSector(PAGEADDRDISK / 512, 1, (void*)pageaddr);
}
```

3 Appendix

.1 INT 13_{16} AH 02_{16}

Register	Value
AL	Number of sectors to read
CH	Cylinder
DH	Head
CL	Sector
DL	Drive
ES:BX	Output offset

.2 Segment Descriptor

Bits	Segment	Value
0-15	*(1)*	Segment limit
16-39	*(1)*	Segment base
40-43	*(flag)*	Type
44	*(flag)*	S
45-46	*(flag)*	DPL
47	*(flag)*	P
48-51	*(2)*	Segment limit
52	*(flag)*	A
53	*(null)*	0
54	*(flag)*	DB
55	*(flag)*	G
56-63	*(2)*	Segment base

.3 INT 15_{16} EAX $E820_{16}$

Register	Value
EAX	$E820_{16}$
EBX	0
ECX	24
EDX	$534D4150_{16}$
ES:DI	Output buffer

.4 VGA Options

Value	**Type**	**Resolution**	**Colourspace**
00_{16}	Text	40x25	1
01_{16}	Text	40x25	16
02_{16}	Text	80x25	1
03_{16}	Text	80x25	16
04_{16}	CGA	320x200	4
05_{16}	CGA	320x200	1
06_{16}	CGA	640x200	2
07_{16}	MDA	80x25	1
$0D_{16}$	EGA	320x200	16
$0E_{16}$	EGA	640x200	16
$0F_{16}$	EGA	640x350	1
10_{16}	EGA	640x350	16
11_{16}	VGA	640x480	1
12_{16}	VGA	640x480	16
13_{16}	VGA	320x200	256

.5 VBE1 Functions

Value	**Type**	**Resolution**	**Colourspace**
100_{16}	Graphics	640x400	256
101_{16}	Graphics	640x480	256
103_{16}	Graphics	800x600	256
104_{16}	Graphics	1024x768	16
105_{16}	Graphics	1024x768	256
106_{16}	Graphics	1280x1024	16
107_{16}	Graphics	1280x1024	256
$10D_{16}$	Graphics	320x200	32K (1:5:5:5)
$10E_{16}$	Graphics	320x200	64K (5:6:5)
$10F_{16}$	Graphics	320x200	16.8M (8:8:8)
110_{16}	Graphics	640x480	32K (1:5:5:5)
111_{16}	Graphics	640x480	64K (5:6:5)
112_{16}	Graphics	640x480	16.8M (8:8:8)
113_{16}	Graphics	800x600	32K (1:5:5:5)
114_{16}	Graphics	800x600	64K (5:6:5)
115_{16}	Graphics	800x600	16.8M (8:8:8)
116_{16}	Graphics	1024x768	32K (1:5:5:5)
117_{16}	Graphics	1024x768	64K (5:6:5)
118_{16}	Graphics	1024x768	16.8M (8:8:8)
119_{16}	Graphics	1280x1024	32K (1:5:5:5)
$11A_{16}$	Graphics	1280x1024	64K (5:6:5)
$11B_{16}$	Graphics	1280x1024	16.8M (8:8:8)

.6 INT 10_{16} AX $4F02_{16}$

Register	Value
AX	$4F02_{16}$
BX	Mode
BX	$4XXX_{16}$ for linear framebuffer

.7 Enabling A20 Line through the Keyboard Controller

Port	Value
64_{16}	AD_{16}
64_{16}	$D0_{16}$
60_{16}	*read*
64_{16}	$D1_{16}$
60_{16}	*(read)* $_{OR}$ *2*
64_{16}	AE_{16}

References

[1] Compaq Computer Corporation, Phoenix Technologies Ltd, Intel Corporation (1996) *BIOS Boot Specification* pg 29 ch 6.5.1 *Booting from BAIDs*

[2] Ralf Brown (2000) *Ralf Browns Interrupt List* interrup b *int 13*

[3] IDEMA (2013) *The Advent of Advanced Format*

[4] Intel Corporation (2016) *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture* pg 1.6 ch 1.3.4 *Segmented Addressing*

[5] Compaq Computer Corporation, Phoenix Technologies Ltd, Intel Corporation (1996) *BIOS Boot Specification* pg 12 ch 3.3 *Devices with PnP Expansion Headers*

[6] Intel Corporation (2024) *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1* pg 2.1 ch 2 *System Architecture Overview*

[7] Compaq Computer Corporation, Phoenix Technologies Ltd, Intel Corporation (1996) *BIOS Boot Specification* pg 43 ch D.1 *Use DL for Drive Number*

[8] Intel Corporation (2022) *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B, 2C, 2D): Instruction Set Reference, A-Z* pg 4.35 ch 3.3 *Model-Specific Registers*

[9] Wikipedia (2024) *Global Descriptor Table* ch Description

[10] Robert Collins (2001) *A20/Reset Anomalies* ch A20/Reset Anomalies

[11] Wikipedia (2024) *Global Descriptor Table* ch Modern Usage

[12] OsDev (2024) *Detecting Memory (x86)* ch 2.1

[13] OsDev (2024) *VGA Fonts* ch 2.4

[14] Wikipedia (2024) *Mode 13h* ch 2.4

[15] Video Electronics Standards Association (1998) *VESA BIOS EXTENSION (VBE) Core Functions Standard* pg 17

[16] Intel Corporation (2016) *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1* pg 9.17 ch 9.9.2 *Switching to Protected Mode*

[17] Intel Corporation (2024) *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B, 2C, 2D): Instruction Set Reference, A-Z* pg 3.610 ch 3.3 *Load Global/Interrupt Descriptor Table Register*