

CATernel

SMP Monolithic Kernel
[Prototype One]

By

Saad Talaat
&
Menna Essa

Table of Contents

-Preface.....	2
1.Boot Loader.....	5-9
2.Setting up the environment	10-11
3.Drivers.....	11-22
4.Memory Management.....	23-32
5.Interrupts and System calls.....	33-37
6.Process Management.....	38-42
7.Appendix A – Problems faced.....	43-44

Preface:

CATernel project aims to develop a non-portable kernel (as a start) that uses Unix interfaces and APIs. CATernel was initiated at the start as a collaborative project in technical community called [Computer Assistance Team] ;It has been initiated (and still) by two students (Saad Talaat Saad , Menna Sherif Essa) on late 2011.Goal is to develop a limited monolithic kernel which supports IA32 architecture for educational and learning purposes.

Project is planned to support a single architecture as a start and also pass through various types of kernel models. The goal of that progressive development is to keep the kernel operating in all cases. And making it usable at any phase of development ; However, Porting the kernel would be carried out at least after we reach the monolithic model. But our main architecture is the IA32 , The Progressive model of the CATernel project can be considered as a prototype design model since in this case the Exokernel model will be a prototype to final goal.

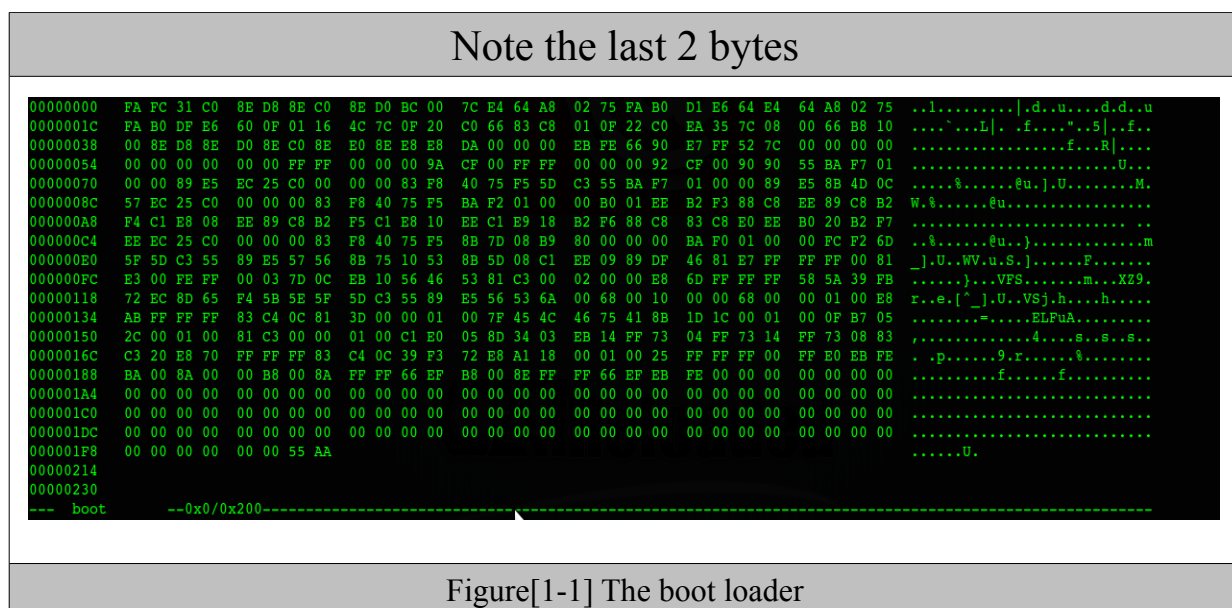
Iterative prototyping is the used software model. Every project sprint contains the components of the project plan. And since we're using the [research & code] way in developing the kernel, iterative prototyping is the convenient software model to use.

1.The Boot Loader:

1.1 Introduction:

booting is the initial set of operations that a computer system performs when electrical power is switched on ; **A boot loader** is a computer program that loads the main operating system or runtime environment for the computer after completion of self-tests.

When a computer is powered the BIOS comes in control and initializes all data. then it looks for a valid boot loader through in the order of the boot device order. a bootable sector is known by the last 2 bytes in the sector, they must be 0xAA55 (boot signature).That Image has the boot loader of our CATernel.



When the BIOS find a bootable image it loads the first 512 byte into address 0:07C00 then jump to it. then the boot sector comes in control. it starts execution in the **real mode** , Real mode is characterized by a 20 bit segmented memory address space (giving exactly 1 MiB[1] of addressable memory) and unlimited direct software access to all memory , I/O addresses and peripheral hardware. Real mode provides no support for memory protection, multitasking, or code privilege levels.

```
.global start
start:

.code16 #since we are in real mode
cli #disable interrupts
cld #clear the direction flag
xorw %ax,%ax #clear the ax register
movw %ax,%ds #clear the data segment register
movw %ax,%es #clear the extra segment register
movw %ax,%ss #clear the stack segment register
movw $start,%sp #set the stack pointer to the bootsector stack
```

Listing 1-1 : Boot sector starts at real mode

1.2 Enabling Protected Mode:

Next it switches to **protected mode** which allows system software to use features such as virtual memory, paging and safe multi tasking.

1.2.1 Enabling A20 Gate:

We start by enabling enabling the A20 Gate [2] for more addressing ; On enabling the A20 gate. first we check if input buffer is full by checking if bit 1 is set on the 0x64 port. then output 0xD1 which makes the next byte passed through the 0x60 port written to IBM AT 804x port. in the next procedure A20.2, you check if the input buffer is full or not , then write 0xDF to the 0x60 port which is written to the IBM AT 804x port which finally enables A20 gate

```
A20.1:
inb $0x64,%al
testb $0x2,%al
jnz A20.1
movb $0xd1,%al
outb %al,$0x64
A20.2:
inb $0x64,%al
testb $0x2,%al
jnz A20.2
movb $0xdf,%al
outb %al,$0x60
```

Listing 1-2 : Enabling A20 Gate

1.2.2 Setting up GDT:

Next step is to enable segmentation , For that you need to setup a GDT[3] -Global Descriptor Table- , For the reason that in Protected mode you can't refer to segments simply by doing this :

```
jmp 0000:7c00h
jmp 0002:0020h
```

You cannot directly access segments. That's why a GDT is used , In the GDT is a table where all the segments are defined . yet they are not stored as values but as descriptors. Descriptors has full information about a segment. A Descriptor is 64 bit long.

31				16				15				0							
Base 0:15								Limit 0:15											
63		56		55		52		51		48		47		40		39		32	
Base 24:31				Flags				Limit 16:19				Access Byte				Base 16:23			

Access byte is
 byte 0 = Accessed bit set to 1 by CPU when segment is accessed. we will set it to 0
 byte 1 = read/write permissions
 byte 2 = Direction bit we will set that to 0 for growing up segments and 1 for growing down segments and conforming bit
 byte 3 = Executable bit 1 if code segment 0 if data segment
 byte 4 = always 1
 byte 5,6 = Privilege since we are a kernel we will set that to 0
 byte 7 = Present bit one for anything

Figure 1-2 : The GDT Descriptor

As in Figure 1-2 , from the first bit till the bit 15, this is the place of the first 16 bits of the segment limit address. and from bit 16 to bit 31 it's the place of the first 16 bits of the segment base address. from bit 32 to 39 more 6 bits of the base address are placed. and then we go into the access bit which is demonstrated above, and the rest of the limit address of the segment. and Flags which is usually set to 0x1100. Finally followed by the rest of the base address. Indexing these descriptors is made by adding 0x8 for every descriptor. So first descriptor's index is 0x0, 2nd descriptor index is 0x8, 3rd descriptor index is 0x10.

```
gdt_table:
.word gdt-gdt_end-1 #gdt table size....mostly 0x17
.long gdt #gdt address
gdt:
.long 0,0 #Null segment
.byte 0xff,0xff,0,0,0,0x9A,0xCF,0 #Code segment
.byte 0xff,0xff,0,0,0,0x92,0xCF,0 #Data segment
```

Listing 1-3 : Setting up GDT Table

After setting up the GDT and loading the gdt address in the gdt register we set

the protected mode in cr0 flag. Doing a far jump using `ljmp` and using the code segment value (0x8) as a segment and next procedure as an offset modified the CS value to the one in our GDT table.

```
switch_mode:
lgdt gdt_table # Load the global descriptor register
mov %cr0,%eax # Load the control register 0 into eax
orl $1,%eax # set the protected mode flag
mov %eax,%cr0 # reset the control register 0
ljmp $CODE_SEG,$protseg #make a far jump to modify the Code
segment
```

Listing 1-4: loading gdt table and switching to protected mode.

After doing this we set the value of data,stack,extra,f,g segments to the one in the GDT. Now we are fully working on protected mode and ready to load the kernel.

```
protseg:
.code32 #since we are working on protected mode
movw $DATA_SEG,%ax #move the data segment value to ax
movw %ax,%ds #set the data segment to data segment value at the
gdt table
movw %ax,%ss # same
movw %ax,%es # same
movw %ax,%fs # same
movw %ax,%gs # same
call cmain # call our kernel loader
```

Listing 1-5 : Setting Registers to the ones in the GDT

1.3 Loading the kernel Image:

Note That we're compiling the kernel as an ELF image , The *cmain* function responsible for loading the kernel is in `arch/x86/boot/main.c`. What the function is that it reads the ELF file from the Image , loads it into memory at virtual address

0x10000 then jumps to the entry point of the kernel executable which referred to through the kernel's ELF headers (`ELFHDR->entry`) .

To do so it uses two functions , *readsect* and *readseg* in order to read the executable from disk , the two functions are also found in main.c file.

2. Establishing environment.

After kernel is fetched from boot loader, environment is reset and memory thunks and segments are remapped. Several components are set before running the kernel:-

- 1- Kernel stack.
- 2- Segmentation and GDT.
- 3- Page tables and directories.

2.1 Kernel Stack

Since execution stream is now in the kernel(0x100000) and not the boot sector(0x7c00). stack must be reset in order to carry on healthy execution we must reset. Kernel stack is paged size memory thunk which we set to global to be able to refer to once we set up paging.

```
.p2align    PAGELG          # Will pad allocation to 0x1000
byte
.global     kernel_stack
kernel_stack:
    .space   KERNEL_STACK
    .global  kernel_stack_end
kernel_stack_end:
```

Listing 2.1 - Kernel stack thunk.

2.2 Segmentation and GDT

while initializing the kernel it is needed to keep in mind the Virtual memory addressing. Thus, Global descriptor table and segmentation is reset to generate new virtual addresses, new kernel segments are 0x10000000 long. Kernel virtual base address is 0xF0000000 which puts the kernel code base address to 0xF0100000. At this point it is important to notice that there's only two segments selectors set.

```
gdt:
    .long 0,0
    SEGMENT(0xffffffff,-KERNEL_ADDR, SEGACS_RW|SEGACS_X)  #
code seg
    SEGMENT(0xffffffff,-KERNEL_ADDR, SEGACS_RW)  # data seg
gdt_end:
```

Listing 2.2 - Kernel Initial Global descriptor table.

CATernel adopts a memory map that is not like any *nix system. Kernel is mapped to high memory addresses like windows. However we are not planning to be a UNIX like operating system.

2.3 Page tables and directories

Before setting up kernel the virtual page tables and page directories are set which will be needed for paging later. We shall touch this point later on on a separate chapter.

3.Drivers :

3.1: CMOS/RTC :

CMOS Complementary metal–oxide–semiconductor , including the RTC -Real time clock- is responsible for saving 50 or 114 bytes of setup information for the BIOS , it includes a battery that keeps the clock active.

CMOS is accessed through I/O ports 0x70 and 0x71 , the ports specifications are as follows : note that indexing is little endian [RTL]

```

a)
0070  w      CMOS RAM index register port (ISA, EISA)
           bit 7      = 1  NMI disabled
           = 0  NMI enabled
           bit 6-0      CMOS RAM index (64 bytes, sometimes 128 bytes)

           any write to 0070 should be followed by an action to 0071
           or the RTC will be left in an unknown state.

b)
0071  r/w     CMOS RAM data port (ISA, EISA)
           RTC registers:
           00      current second in BCD
           01      alarm second  in BCD
           02      current minute in BCD
           03      alarm minute  in BCD
           04      current hour  in BCD
           05      alarm hour    in BCD
           06      day of week  in BCD
           07      day of month in BCD
           08      month in BCD
           09      year  in BCD (00-99)
           0A      status register A
                   bit 7 = 1  update in progress
                   bit 6-4 divider that identifies the time-based
                           frequency
                   bit 3-0 rate selection output  frequency and int. rate
           0B      status register B
                   bit 7 = 0  run
                   = 1  halt
                   bit 6 = 1  enable periodic interrupt
                   bit 5 = 1  enable alarm interrupt
                   bit 4 = 1  enable update-ended interrupt
                   bit 3 = 1  enable square wave interrupt
                   bit 2 = 1  calendar is in binary format
                   = 0  calendar is in BCD format
                   bit 1 = 1  24-hour mode
                   = 0  12-hour mode
                   bit 0 = 1  enable daylight savings time. only in USA.
                           useless in Europe. Some DOS versions clear
                           this bit when you use the DAT/TIME command.
           .
           .
           .

```

Listing 3.1 CMOS Ports , a) I/O Port 0x70 , b) I/O Port 0x71 [1]

CMOS values are accessed one byte at a time so we refer to each byte as a CMOS Register, The first 14 CMOS registers access and control the Real-Time Clock. In port 0x70 -CMOS RAM index register port- as it's name suggests saves the CMOS RAM index which is a Nonvolatile BIOS memory refers to a small memory on PC motherboards that is used to store BIOS settings , Also the last bit in the register indicates whether the NMI -non maskable interrupts-[2] are enabled (0) or disabled (1).

So port 0x70 is used to select the CMOS Register to read , so if you want to read register 0A which holds Status A register you simply do this :
`outb(0xA, 0x70) ;`

For making things readable , we define all the indexes and constants in `cmos.h` headers

```
/* CMOS Registers */
#define CMOS_INDEXPORT    0x70
#define CMOS_DATAPORT     0x71
/* RTC Registers */
#define RTC_SECONDS       0x0
#define RTC_ALRMSECOND    0x1
#define RTC_MINUTES       0x2
.
.
.
```

Listing 3.2 cmos.h

From here we can introduce the first CMOS function ,`cmos_get_reg`; which will take the register the needs to be read as an input , offset the index register and read it. to explain the procedure in more details , if you see the code in listing [3-3] you'll see that we first read Status A register if bit 7 is 1 , that is the register = "10000000" = 0x80 , means that the CMOS is being updated and you can't use it now so the function busy waits and keeps reading the register until it's free . Then it offsets the index register to the desired register to read held in the parameter "value" , finally reads it from the Data port(0x71) and return it.

```
uint32_t cmos_get_reg(uint8_t value){
    uint32_t val;
    uint8_t update;
    //check status
    while(update == 0x80){
        outb(RTC_STATUS_A, CMOS_INDEXPORT);
        update = inb(CMOS_DATAPORT);
    }
    cli();
    //get the value
    outb(value, CMOS_INDEXPORT);
    val = inb(CMOS_DATAPORT);
    return val;
}
```

Listing 3.3 cmos.c , reading cmos registers

Next function is *cmos_get_time* which is similar to *cmos_get_reg* which deals only with time -register 0:9- , The function is to be deprecated.

Finally , *cmos_set_power_stat* is responsible for supporting status B register which includes the power options and status It follows the same sequence as the previous functions but sets the register to the values mentioned in status B described in listing [3-4] .

```
/*those 3 sets the respective bit to zero so we mask with AND*/
if(stat==STAT_RUN || stat==STAT_CAL_BCD ||
stat==STAT_CAL_HR12)
    New_Stat &= stat;
else
    New_Stat |= stat;
```

Listing 3.4 setting status B power options.

The rest of the registers are not yet used hence not yet supported.

[1]: The full detailed prots description can be found at

<http://bochs.sourceforge.net/techspec/PORTS.LST>

[2]: Refer to interrupts chapter for more details.

3.2 Video

Almost all new kernels and operating systems access and use video mode using the VESA/VBE interface. to interface with the attached video card. however in CATernel we choosed to support legacy and old devices before supporting newer versions, therefore we choosed to use CGA(Color Graphic Adapter) to support video and console. There's few differences between CGA and EGA and VGA.

3.2.1 CGA(Color Graphic Adapter)

As mentioned earlier CGA is an old graphic adapter which we choosed to support first in CATernel. It supports two modes, Text mode and Graphic mode. for now we are only using the Text mode.

3.2.1.1 Text Mode

CGA has two text modes with a fixed character size.

40x25 Mode : Each character is 8x8 dots size and has up to 16 colors with resolution of 320x200.

80x25 Mode : It has the same character size and same color count, but with 640x200 resolution.

on CATernel we will support the CGA in text mode(80x25).the memory storage is two bytes of video RAM used for each character. 1st byte is the character code and the 2nd is the attribute. a screen might be 2000 byte or 4000 byte (40*25*2) , (80*25*2). and CGA's video RAM is 16Kb. and of course all what we can output is ASCII.

bit 0 = Blue foreground bit 1 = Green foreground bit 2 = Red foreground bit 3 = Bright foreground bit 4 = Blue background bit 5 = Green background bit 6 = Red background bit 7 = Bright background (blink characters)

Listing 3.5 Character color attributes
--

To control screen cursor and lines two registers are used, Index and Data registers at address 0x3D4 and 0x3D5 respectively. If you took a look at video.c code you will find setters and getters for position. so position desired to read is supplied to index register which is 0xF for the first byte in the position and 0xE for the second byte.

and since we are working on a 80*25 then we wont need more than 4 bytes. for getting the value i read from the data register after specifying the index I want to read and I *inb* the value coming from the data port. for example:

suppose the cursor position is at 0x5a0, so what you will first get is the first byte of the position which is 0xa0. and as you might have noticed we do no operations on that. but on the second position you get 0x05. the operations is for mixing the first byte and second byte so they would make 0x5a0.

also you may notice that CGA_BUFF_OFFSET. which is the offset of the CGA video RAM in memory.

cga_putc:

what i do here is that i put the character i want to type on the screen to the CGA video RAM. and since we are working on 80*25 resolution bytes after the offset 0xB87D0 wont be written to screen yet they will be written to video RAM. this issue can be handled using memory trick like...move all binaries from

0xB8080 to 0xB87D0 80 byte backward which is the row size in 80*25 resolution. and then move the cursor position 80 place backward.

cga_putstr:

this function passes a pointer to an array of characters which are passed in a loop character at a time till we reach the null terminator character.

CGA Ports:

```
0x3D4      -      index register
0x3D5      -      data register
0x3D6      -      same as 0x3D4
0x3D7      -      same as 0x3D5

0x3D8      -      CGA mode control register
|-> bit 5 - blink register
|-> bit 4 - 640*200 High-Resolution register
|-> bit 3 - video register (if cleared the screen wont output)
|-> bit 2 - Monochrome
|-> bit 1 - text mode
|-> bit 0 - Resolution

0x3D9      -      Palette Register / Color control register
|-> bit5 - chooses color set
|-> bit4 - if set the characters show in intense
|-> bit3 - intense border in 40*25 and intense background in
          300*200 and intense foreground in 640*200
|-> bit2 - red borders in 40*25, red background in 300*200,
          red foreground in 640*200
|-> bit1 - green border in 40*25, red background in 300*200,
          red foreground in 640*200
|-> bit0 - blute border in 40*25, red background in 300*200,
          red foreground in 640*200

0x3DA      -      Status register
|-> bit3 - if set then in vertical retrace.
|-> bit2 - light pin switch off
|-> bit1 - positive edge from light pen has set trigger
```

```
|-> bit0 - 0 do not use memory.  
  
0x3DB    -    clear light pen trigger  
0x3DC    -    set    light pen trigger
```

Listing 3.6 CGA Ports

Console dependency:

The early console depends on the video driver and keyboard. On the console there's several wrapper functions for the video driver like *console_putc* , *putchr* and *console_clear*.

3.3 PS/2 Keyboard

Unlike video driver, CATernel's keyboard driver is similar to every kernel's keyboard support. However, since Keyboard is a serial device It uses almost a unified interface like any other PS/2 device. CATernel keyboard driver supports only one function which is a keyboard interrupt handler, Of course at this point a keyboard interrupt does not occur since CATernel busy wait on any I/O device.

Like any other device Keyboard commands are passed through I/O ports. In CATernel we barely make use of the keyboard controller, the only two operations are made through the status and data port. First operation is to check the keyboard data register, second one is two read that character from the data port.

```
0064 r    KB controller read status (ISA, EISA)
        bit 7 = 1 parity error on transmission from keyboard
        bit 6 = 1 receive timeout
        bit 5 = 1 transmit timeout
        bit 4 = 0 keyboard inhibit
        bit 3 = 1 data in input register is command
              0 data in input register is data
        bit 2    system flag status.
        bit 1 = 1 input buffer full
        bit 0 = 1 output buffer full
```

Listing 3.7 Keyboard Status port

There's three different scan code sets, CATernel uses the first scan code set. a scan code determine what key is pressed and three keyboard maps are provided, the first is a character map on normal case, second is a character map of keyboard on shift case and third is a character map of keyboard once a toggle button is on.

A keyboard interrupt handler reads the scan code and starts determining what key was pressed and then it is returned to interrupt issuer.

Console dependency

Console uses keyboard controller as an input device, a wrapper function called *console_getc* issues a keyboard interrupt and an index that char to a screen position (but char is not printed).

3.4 Intel 8259 PIC (Programmable Interrupt Controller)

As mentioned before on CATernel we tend to support legacy and old devices first, therefore we choosed to support Intel 8259 PIC before APIC(Advanced PIC) and IOAPIC are supported. Interrupts are the only way to manage execution over x86 machines since Intel is an interrupt driven ISA. on old machines when only real mode was used PIC was the controller for interrupts and interrupts were handled by what is called vector store in an Interrupt vector table. An interrupt vector table is similar to the modern interrupt decriptor table except that IVT has the vectors already stored on the BIOS. PIC consists of two chip (Master/Slave) each has 8 interrupts which makes the total interrupts 16. an PIC interrupt is called IRQ(interrupt request) since an interrupt can be blocked when a higher priority interrupt handler is currently executing.

```

INT# 00 > F000:FF53 (0x000fff53) DIVIDE ERROR ; dummy iret
INT# 01 > F000:FF53 (0x000fff53) SINGLE STEP ; dummy iret
INT# 02 > F000:FF53 (0x000fff53) NON-MASKABLE INTERRUPT ; dummy iret
INT# 03 > F000:FF53 (0x000fff53) BREAKPOINT ; dummy iret
INT# 04 > F000:FF53 (0x000fff53) INT0 DETECTED OVERFLOW ; dummy iret
INT# 05 > F000:FF53 (0x000fff53) BOUND RANGE EXCEED ; dummy iret
INT# 06 > F000:FF53 (0x000fff53) INVALID OPCODE ; dummy iret
INT# 07 > F000:FF53 (0x000fff53) PROCESSOR EXTENSION NOT AVAILABLE ; dummy iret
INT# 08 > F000:FEA5 (0x000ffea5) IRQ0 - SYSTEM TIMER
INT# 09 > F000:E987 (0x000fe987) IRQ1 - KEYBOARD DATA READY
INT# 0a > F000:FF53 (0x000fff53) IRQ2 - LPT2 ; dummy iret
INT# 0b > F000:FF53 (0x000fff53) IRQ3 - COM2 ; dummy iret
INT# 0c > F000:FF53 (0x000fff53) IRQ4 - COM1 ; dummy iret
INT# 0d > F000:FF53 (0x000fff53) IRQ5 - FIXED DISK ; dummy iret
INT# 0e > F000:EF57 (0x000fef57) IRQ6 - DISKETTE CONTROLLER
INT# 0f > F000:FF53 (0x000fff53) IRQ7 - PARALLEL PRINTER ; dummy iret
INT# 10 > C000:014A (0x000c014a) VIDEO
INT# 11 > F000:F84D (0x000ff84d) GET EQUIPMENT LIST
INT# 12 > F000:F841 (0x000ff841) GET MEMORY SIZE
INT# 13 > F000:E3FE (0x000fe3fe) DISK
INT# 14 > F000:E739 (0x000fe739) SERIAL
INT# 15 > F000:F859 (0x000ff859) SYSTEM
INT# 16 > F000:E82E (0x000fe82e) KEYBOARD
INT# 17 > F000:efd2 (0x000fed2) PRINTER
INT# 18 > F000:B023 (0x000fb023) CASSETTE BASIC

```

Figure 3.1 IVT listed by Bochs

3.4.1 Master and Slave PIC

To handle all the 16 we must be able to index the right interrupt to the right PIC and when an EOI is issued we should be able to determine which PIC should handle the EOI. Master PIC and slave PIC have their own ports, Master has 0x20/0x21 and Slave has 0xA0/0xA1. the functionality is similar except for the interrupt types they handle.

```
1- Intel i8253 PIT
2- Keyboard
3- Video Interrupt
4- Serial port 2
5- Serial port 1
6- Fixed Disk
7- Floppy disk
8- Parallel printer
9- Real time clock (RTC)
10- Cascade Redirect
13- Mouse interrupt
14- Coprocessor exception
15- Primary Hard disk
16- Secondary Hard disk
```

Listing 3.8 PIC Interrupt requests (IRQs)

PIC I/O is a little different than former devices we dealt with in CATernel. PICs adopt a terminology called ICW(Initialization command word) and OCW(Operation command word). Intel 8259 manual defines ICW that It is used before any normal operation. as for OCW it can be executed at any point after initialization.

```
0020 w    PIC initialization command word ICW1
          bit 7-5 = 0    only used in 80/85 mode
          bit 4 = 1     ICW1 is being issued
          bit 3 = 0     edge triggered mode
                  = 1   level triggered mode
```

bit 2	= 0	successive interrupt vectors use 8 bytes
	= 1	successive interrupt vectors use 4 bytes
bit 1	= 0	cascade mode
	= 1	single mode, no ICW3 needed
bit 0	= 0	no ICW4 needed
	= 1	ICW4 needed

Listing 3.9 Port 0x20 flags for ICW1

3.4.2 PIC in protected mode

In x86 protected mode interrupts are only handled by the IDT and the IVT is omitted. such a case will put us in a problem whenever an IRQ is issued since it will conflict with Intel default 0~32 exceptions, therefore we won't be able to distinguish an exception from an IRQ. Luckily, Offsetting the IRQ indexes is a functionality can be performed through PIC ICWs. This is done through ICW2 in particular.

3.4.2.1 Initializing PICs

PIC initialization is done once the kernel have reached protected mode main-flow execution. It enables IRQs to be handled using IDT after offsetting them so CATernel would be able to use PIC in protected mode. this is done on four steps

First Step, ICW1 is passed a value with Flags ICW4 needed and ICW1 issued flags. Second Step, ICW2 is passed a value with the base offset desired to IRQs, and Since we do this step for both PICs the slave PIC base offset is passed as master PIC offset plus 8. Third Step, ICW3 is passed a value that holds one shifted by the interrupt pin of the slave PIC. and slave PIC ICW3 takes slave PIC index. Fourth Step, ICW4 takes a value with 8088/8086 mode flag set. However other flags are also active (0/1).

3.4.3 Masked Interrupts

Masked interrupts is another terminology an i8256 adopts, since it supports enabling and disabling interrupts through setting masks holds flags of desired interrupts and undesired. an interrupt could be disabled by setting its corresponding flag. Interrupt masks are held in a PIC register called IMR(Interrupt Mask Register). each PIC has its own IMR since each PIC has its own type of interrupts.

```
0021 r/w PIC master interrupt mask register
OCW1:
    bit 7 = 0  enable parallel printer interrupt
    bit 6 = 0  enable diskette interrupt
    bit 5 = 0  enable fixed disk interrupt
    bit 4 = 0  enable serial port 1 interrupt
    bit 3 = 0  enable serial port 2 interrupt
    bit 2 = 0  enable video interrupt
    bit 1 = 0  enable keyboard, mouse, RTC interrupt
    bit 0 = 0  enable timer interrupt
```

Listing 3.10 IMR in Master PIC

Such a functionality gives the ability to disable the whole PIC by setting all flags on both master and slave PICs.

3.4.4 EOI End Of Interrupt

EOI must be used by an IRQ handler since it notifies the PIC that issued an interrupt that the interrupt handler has finished its execution, so the PIC should insert its blocked interrupt (if exists) to processor.

```
0020 w OCW2:
    bit 7-5 = 000 rotate in auto EOI mode (clear)
              = 001 nonspecific EOI
              = 010 no operation
```


	= 011	specific EOI
	= 100	rotate in auto EOI mode (set)
	= 101	rotate on nonspecific EOI command
	= 110	set priority command
	= 111	rotate on specific EOI command
bit 4	= 0	reserved
bit 3	= 0	reserved
bit 2-0		interrupt request which the command applies

Listing 3.11 EOI using OCW2

3.5 Intel 8254 PIT(Programmable Interrupt Timer)

Due to the problem with RTC periodic interrupts [Appendix A] we had to support another device for time slicing execution. The next modern device is PIT but still PIT is a legacy device. however, PIT was easier to program than RTC.

PIT has three channels. First channel is for counter divisor, second channel is for RAM refresh counter and third one is for issuing a beep on cassette or speaker on an interval. For the kernel clock we shall only use the first one which is the divisor of the frequency of the PIT to issue an interrupt on a subsequent interval. in CATernel we set this to issue 20 interrupt per second and handle those interrupts by scheduler.

4. Memory Management

On kernel level, Memory management and allocation is a very crucial and critical part that composes an efficient performance and protection. Dealing with memory on kernel level has to be very careful and clever since kernel doesn't have the user space luxuries like memory allocation errors.

Unlike Linux, BSD and Spartan kernels, CATernel doesn't yet contain any *Zone terminology* although there's different thunks of memory CATernel doesn't make use of the whole memory. Only two thunks of memory are used, Base and extended memory. and allocation on kernel level is done by manually allocating a memory unit which here is Paging.

4.1 Paging

A page is the smallest unit of memory in CATernel. Although a page is considerably huge comparing to processor smallest unit of memory which is a byte the Intel MMU deals with pages as the basic unit of virtual memory. IA32 has different types of paging methods called "Paging Modes", First is 32-bit paging which addresses 32-bit physical addresses to 32-bit virtual addresses, Second is PAE Paging which is used to translate 52-bit Physical addresses to 32-bit Virtual addresses and the third mode is called IA32e Paging which is used to translate same size of former physical addresses to 48-bit virtual addresses.

Since we're applying a minimal implementation for paging in CATernel we are only making use of 32-bit paging which has two types of data structures Page tables and page directories and two modes each has a different page size.

4.1.1 32-bit Paging structures

Two types of Data structures exist to index a page in 32-bit mode. It can be considered a two dimensional page array with the higher level is the page directory. Page directory contains page tables addresses and several flags, each member of page directory is called PDE(Page Directory Entry). Page table contains the addresses of physical pages and several flags, each member of page table is called PTE(Page Table Entry).

a 32-bit KByte Paging PDE would be like this.
[0,11] Page Table Permissions and Ignored bits.

```

[12,31] Page Table address.

a 32-bit KByte Paging PTE would be like this.
[0,11] Page permissions
[12,31] Page physical address

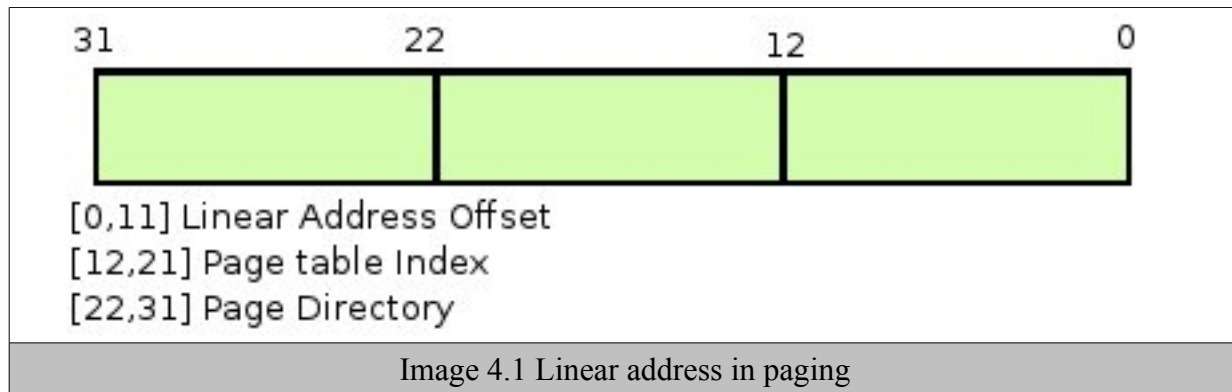
the permissions of the PDE is almost the same of the
permissions of the PTE. PDE permissions:
[0] First bit must be always 1 which marks page as
(Present), if it's not set the entry is ignored.
[1] R/W permissions, If bit is cleared no write
operations are allowed
[2] U/S, it indicates whether the page/page table
belongs to user or supervisor (ring 0,3). if it's
cleared, it means it belongs to supervisor which
forbids access from CPL =3
[3] WT, Write through flag it indicates the memory
type used to access this page either
write back caching or write through caching.
[4] CD, it indicates if this page is cachable or not,
if set it's not cachable.
[5] A, Accessed flag refers to whether a software
accessed this page or not.
[6] D, Dirty flag is set if a software did a write
operation to this page.
[7] PS, Page size flags if set it means we're using
4MByte paging, if not it's 4KB paging
[8] G, if set it means that the directory translation
is Global. we shall refer to it later.

```

Listing 4.1 Paging structures entries

4.1.2 32-bit Paging Modes

First is 4Kbyte page 32-bit Paging, It uses two data structures to index a page (Page Directory - Page Table), Second type is 4Mbyte page 32-bit paging, which uses only one data structure to index a page (Page Directory). In CATernel the first mode is used since it will enable smaller pages hence, smaller basic memory units. A smaller memory unit has some advantages and disadvantages. having a 4Kbyte page as a virtual memory smallest unit will provide less fragmentation. on the other hand, a bigger memory size will be used to store paging data structures.



A linear address in 4Kbyte 32-bit paging mode contains three fields, Linear address offset, Page table entry index and page directory entry index. in 4KByte Paging, and since we refer to 4Kbyte sized page which is 2^{13} it means we can offset with FFF into the page. from 0xFFFFF000 to 0xFFFFFFFF for example. the other [12,21] bits indicate the index of the page in the page table.

4.1.3 Initializing paging

Allocating/Clearing Page Directory

we use the boot time allocation scheme to allocate 4096 bytes of memory right after the kernel LOAD segment and clear it. and to provide access to page table for both user and supervisor to access the page directory by making it recursively reference itself when a virtual page table address is used. in our case, VIRTPTGT, USERVIRTPTGT. which lie in 0xEFC00000 , 0xEF400000 repectively. So for those page numbers/linear address to refer to page directory itself we map it to itself by this line.

```
pdr[PGDIRX(VIRTPTGT)] = KA2PA(VIRTPTGT) | PAGE_PRESENT | PAGE_WRITABLE;
```

and the index of this entry is 3BF. it looks in bochs like this.

```

<bochs:5> x/10x 0xf010befc
[bochs]:
0xf010befc <bogus+ 0>: 0x0010b003 0x03ffd027 0x03ffc007 0x03ffb007
0xf010bf0c <bogus+ 16>: 0x03ffa007 0x03ff9007 0x03ff8007 0x03ff7007
0xf010bf1c <bogus+ 32>: 0x03ff6007 0x03ff5007
<bochs:6>

```

Image 4.2 recursive page directory indexing

Pages data structure

What first comes in your mind if you need to detect whether there's a free page or not is to scan the page directory and table and detect free pages and search whether the page you want to map lies between those free pages or not. this would create a MASSIVE overhead. But a better way to do this is to create a linked lists, of Pages structures or whatever it might be called, it's not actually page structures but it's a (struct Page list) this is a simple backward linkedlist entry with a pointer to previous element and a value field, in our case this field is called ref which indicates how many pointers or procs refer to that page of course if it's allocated. if this ref field is 0 it makes this page free to use. and this is how it looks in memory.

```

<bochs:19> x/16x 0xf010d000
[bochs]:
0xf010d000 <bogus+ 0>: 0x00000000 0xf010d00c 0x00000001 0x00000000
0xf010d010 <bogus+ 16>: 0xf010d018 0x00000000 0xf010d00c 0xf010d024
0xf010d020 <bogus+ 32>: 0x00000000 0xf010d018 0xf010d030 0x00000000
0xf010d030 <bogus+ 48>: 0xf010d024 0xf010d03c 0x00000000 0xf010d030
<bochs:20>

```

Image 4.3 pages list

Initializing structures.

after setting up the environment, we start to initialize page directories and tables. and since we need to be still operating after paging activation we need to put entries for both Kernel code and stack so after paging is active the same addresses would be translated to same physical position.

in steps,

we map the whole memory into pages and start filling out the free pages list. this can be done by a loop. but there's a memory we need to mark used that has the ACPI system calls and Memory mapped I/O [Section 6.1] plus the kernel code/stack segments are also in use, so those we need to mark used as well. after filling the free pages list. memory mapping procedure should be supported to map physical segments to virtual addresses in runtime. This is the `map_segment_page` function. to provide such a function we need other functionalities, Like the ability to find and create a page table at a specific position, and insert or remove a page. and allocate a page. a simple page allocation and freeing functions is to simply remove and add a page member to the free pages global list.

to insert and remove pages you need to be able to locate and create new Page dir Entries that are dependent on the virtual address. for this `x86_pgdir_find` function is defined, it takes the virtual address which's PDE is desired to be found or created. the function first checks if this entry already exist, if yes a PDE is returned. (it is referred in the code as PTE since it references the table, however it's called PDE in intel manuals) if not it is created if desired.

to remove or insert a page directory entry to a page directory other two functions were defined, removal function uses a lookup function to determine the existence of PDE, then it executes a detach function that determine whether there's processes that are still using this page or not, if not page is freed. then the pte is set to NULL or 0 and TLB is updated. to insert a page, the function first checks if there's a PDE/PTE referring to this page or the creatability of PDE/PTE referring to this page, if the page already is referred. if yes it's freed and reallocated. but we won't be using these functions ATM.

At this point mapping a segment of memory to virtual memory is trivial, for each page of the segment you insert a PDE and and a PTE that refer to this VA

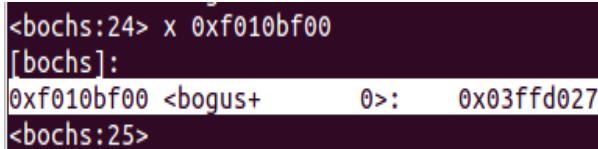
and an opposing PA, this makes the PTE random.

4.1.4 Triggering paging

to activate paging we need to first map the pages array so we can still access it via the same virtual address, also we need to map the kernel stack and the kernel code. a kernel code mapping for instance looks like this:-

```
map_segment_page(pgdir, KERNEL_ADDR, 0x10000000, 0, PAGE_PRESENT | PAGE_WRITABLE);
```

this maps virtual address 0xF0000000 to 0xFFFFFFFF to the pages from 0 to 0xFFFFFFFF. let's do this manually we're not treating the 0xF0000000 anymore as segment base, but as a page number. which's PDE Index = 3C0 which means the entry's offset from pgdir base is $3C0 * 4 = F00$, let's check that in the bochs debugger.



```
<bochs:24> x 0xf010bf00
[bochs]:
0xf010bf00 <bogus+ 0>: 0x03ffd027
<bochs:25>
```

Image 4.4 PDE in memory

It is noticable that the PDE is marked Accessed since it's already executing. now let's read the Page table, the 0x03ffdXXX refer to the physical address of page table, and since it's in the 0 ~ 0x10000000 kernel address space we'll just add a FFFFFFFF to it.

```
0x051f0000 <bogus> 0x051f0000: physical address not available for linear 0x051f0000
<bochs:27> x/10x 0xf3ffd000
[bochs]:
0xf3ffd000 <bogus+ 0>: 0x00000003 0x00001003 0x00002003 0x00003003
0xf3ffd010 <bogus+ 16>: 0x00004003 0x00005003 0x00006003 0x00007003
0xf3ffd020 <bogus+ 32>: 0x00008003 0x00009003
<bochs:28> █
```

as you see pages are sequentially ordered in page table as PTEs.

eventually, we load page directory address to cr3 and trigger paging on CR0. after we set the first PDE as the Kernel codes PE. since after paging the 1st PDE is loaded. then we reset segment table to full 4GB memory since we're able to convert 4GB of linear addresses to physical addresses. and a far jump is done to the same code it preserve the execution of the code after removing the page directory[0]. and for the CS to get updated.

4.1.3 Allocation

CATernel uses a weak memory allocation schemes, First is used on boot time which allocates memory heaps after kernel code section. Second is used by paging manager which allocates one page at a time. However, that's a subject to be looked in later, Zoning and slab allocators might be used.

4.2 Segmentation

In CATernel we use segmentation effectively on pure segmentation. and we only use the Global descriptor table. We use segmentation in re-mapping the kernel physical address into a virtual address, and maintain the 32-bit addresses. A global descriptor table is used for the OS level/ring 0 to locate its Code/Data/TSS segments. other tasks use the Local descriptor Table [LDT] which differs from a task to another. Like *Nix and WinXP we only use Paging to have protection and addresses virtualization. once paging is activated, we set GDT selectors base address to 0 with MAX memory limit. since we don't need longer addresses like protected mode. But, It's intended to have full power of Intel memory management like in OS/2.

5. Interrupts and system calls

5.1 Interrupts:

According to Intel Software developer manual Vol 3 [Chp.6 Interrupts and Exception Handling] an Interrupt can be defined as follows:-

Interrupts occur in random times during execution, and they are invoked by hardware. Hardware uses interrupts to handle events that are external to the processor such as request to service a device. software can generate an interrupt also by instruction `INT <interrupt_number>` also according to varying sources interrupt numbers break into the following.

-IRQ [Interrupt ReQuest]:

Interrupt requests are from 0 to 16

-Interrupts:

Interrupts are from 0~31, 8~16, 70h~78h

-PIC/Keyboard Ports

Programmable Interrupt controller and keyboard ports.

an Exception can be defined as follows:-

Exception occurs when a processor detects an error during trying to execute an instruction like division by 0. the intel processor detects many error conditions including protection violation as page faults.

0	Programmable Interrupt Timer Interrupt
1	Keyboard Interrupt
2	Cascade (used internally by the two PICs. never raised)
3	COM2 (if enabled)
4	COM1 (if enabled)
5	LPT2 (if enabled)
6	Floppy Disk

7	LPT1 / Unreliable "spurious" interrupt (usually)
8	CMOS real-time clock (if enabled)
9	Free for peripherals / legacy SCSI / NIC
10	Free for peripherals / SCSI / NIC
11	Free for peripherals / SCSI / NIC
12	PS2 Mouse
13	FPU / Coprocessor / Inter-processor
14	Primary ATA Hard Disk
15	Secondary ATA Hard Disk
Listing [5-1] Standard ISA IRQs	

Int	Description
0-31	Protected Mode Exceptions (Reserved by Intel)
8-15	Default mapping of IRQ0-7 by the BIOS at bootstrap
70h-78h	Default mapping of IRQ8-15 by the BIOS at bootstrap
Listing [5-2] Default PC Interrupt Vector Assignment	

5.1.1 Dummy interrupts

For the sake of prototyping, There was a dummy exception and interrupts handler created under a generic name for both as Interrupt. However this still stands till now, but few modifications was mode which will be mentioned later on the document.

an interrupt occurrence causes an interrupt handler to be executed from the protected mode Interrupt Descriptor Table. the indexed function that matches interrupt number will be executed. To do that we must first initialize the IDT by filling the first 64 interrupt by a semi interrupt handler and filling the rest 196 interrupt with a dummy iret. Once an interrupt occur the execution is altered to kernel space and starts executing the interrupt handler. Before an interrupt handler executes by default there's a stack frame storing the previous cpu state (the interrupt issuer) to be able to return to it, however this is not enough for us. In CATernel we define a soft cpu state structure holding all general purpose and segment registers and address space of the current environment.

Since there's no interrupts (not exceptions) that are yet handled through CATernel except for RTC periodic interrupts and system calls, any other interrupt once issued from user it goes through the interrupt mapping function and returns to user if no handler exists. yet there's illegal interrupts to be issued by user like the RTC which is used in scheduling.

5.1.2 Back end

-The gatedesc structure :

Our initial gate descriptor structure is more like the one in HelenOS, although i find it to be almost useless, since We won't be really using the args, reserved. And it's time consuming executing an assignment statement for every member of type,dpl and present bit. such a structure is implemented in both Linux/Minix as a one type_dpl_present field. We shall save this for later.

-CPU state frames :

to provide an informative and effective switching between caller and interrupt vector, cpu state frame holds info about variants of caller environment. such a frame in Minix for instance holds(vector, error code, eip, cs, eflags, esp, ss) in Linux all registers exist which is the same as HelenOS, although order is different.

5.1.3 Page faults

To this point we have a minimal user space, and poor scheduling mechanism and no signaling since we busy wait on resources. This simplifies the page fault handling for us in CATernel. Page fault handler checks if a page fault came from kernel mode, If so kernel is panicked. If it came from user space by user trying to access kernel space, the exception issuer proc is killed. If a page fault is done by user by jumping to wrong address (by instruction fetch) issuer proc is killed. if the issuer proc did issue a page fault by exceeding the stack, the proc stack is increased if it didn't reach max size for a user space stack.

5.2 System calls

A system call is a request for service that a program makes of the kernel. The service is generally something that only the kernel has the privilege to do, such as doing I/O. users do not need to concern themselves with system calls as they will be all done virtually in a library .

for a system call to happen a number of steps are followed:

|user code|->|Intermediate Library|->|Kernel code|

First , The program calls the function in the user library , this function is responsible for indexing and passing the arguments of the kernel function , then it issues and SYSCALL(0x30) interrupt

Before going into code there's an important table to mention , that is the *sys_call_table*

The table contains function pointers to the kernel level system calls handlers.

```
fnptr_t sys_call_table[] = { sys_exec ,
                             sys_fork ,
                             sys_printf
                             ....
                             };
```

Listing[5-3] System call table.

The first thing the call sets is the index for the correct function pointer in that table , indexes are defined in `include/sys.h`

```
asm("movl %0,%%eax" :: "a"(S_PRINTF));    ;set call index
asm("movl %0,%%ebx":: "a"(1));              ;for the function
asm("movl %0,%%ecx" : "=g"(str));          ;function argument
asm("int %0" : : "a"(0x30));                ;SYSCALL interrupt
```

Listing[5-4] printf.c prototype

after the function issues the interrupt ,and the interrupt handler finds that it's a

SYSCALL interrupt , it'll call map_syscall function and pass the current cpu_stat structre to it , the function will index the sys_call_table and call the function with the arguments in ecx register and return the return value (error code) of the call.

```
s_errno= (sys_call_table [cpu_state->eax])((char *)cpu_state->ecx);  
    if(s_errno < 0 ) {return -s_errno;} //error code  
    else {return 0;}
```

Listing[5-5] mapping system calls

6. Process Management:

6.1 Loading Process:

Currently we're only support ELF formats , *elf_load_to_proc* reads the binary from disk and populates the proc structure with the process information and sets the entry point.

```
typedef struct proc {
    gpr_regs_t    gpr_regs;
    seg_regs_t    seg_regs;
    reg_t         eip;
    uint32_t      cs;
    reg_t         eflags;
    reg_t         esp;
    uint32_t      ss;
    uint32_t      proc_id;
    uint32_t      proc_status;
    pde_t         *page_directory;
    uint32_t      cr3;
    uint32_t      preempted;
    uint32_t      dequeued;
    LIST_ENTRY(proc) link;
    LIFO_ENTRY(proc) q_link;
} proc_t;
```

Listing [6-1] Process structure

The function will remind you of kernel fetching, based on the binary offset it will seek it and read the disk's block into an elfhdr structure , which will then iterate through the headers to copy the entire file into memory , update the page table and the CR3 register as an initialization to the user environment.

6.2 Scheduling

Scheduling

Scheduling is the process of organizing the context switching between processes in order to achieve multi-tasking

In prototype1 we're using a simple LIFO Time sharing Round robin (Last In First Out) scheduling algorithm

For a quick recap , LIFO refers to the way items stored in a data structure are processed. The last data to be added to the structure will be the first data to be removed. LIFO mechanisms include data structures such as stackss. A LIFO structure can be illustrated with the example of a crowded elevator. When the elevator reaches its destination, the last people to get on are typically the first to get off , the same thing applies to processes , the last process added to the queue is the first process to be taken out of the queue so that another one would take it's place.

another important concet in scheduling is context switching , a "context" is a virtual address space, the executable contained in it, its data etc.

A "context switch" occurs for a variety of reasons - because a kernel function has been called, the application has been preempted, or because it had yielded its time slice.

A context switch involves storing the old state and retrieving the new state. The actual information stored and retrieved may include EIP, the general registers, the segment registers, CR3 (and the paging structures), FPU/MMX registers, SSE registers and other things. Because a context switch can involve changing a large amount data it can be the one most costly operation in an operating system.

since Resource management is not yet implemented we currently have 2 queues , a ready queue and a running queue after a quantum of time passes

the schedule function checks the running and ready queues , last process in the running queue will be popped and replaced with the last process in the ready queue , which is to be scheduled then a context switch to this process occurs with the *switch_address_space* function.

`schedule(void)`

```
{
    uint32_t idx= 0;
    proc_t *proc, *pproc;
    if(!LIFO_EMPTY(&running_procs))
    {
        pproc = LIFO_POP(&running_procs, q_link);
        printk("[*] Proc running: %d\n",pproc->proc_id);
    }
    else
        printk("[*] No running procs\n");

    if(!FIFO_EMPTY(&ready_procs))
    {
        proc = FIFO_POP(&ready_procs);
        printk("[*] Ready proc: %d\n",proc->proc_id);
    }
    else
    {
        printk("[*] No ready procs found\n");
        proc = pproc;
    }

    if(!LIFO_EMPTY(&running_procs))
        FIFO_PUSH(&ready_procs, pproc);

    LIFO_PUSH(&running_procs, proc ,q_link);

    printk("[*] Scheduling to process: %d\n", proc->proc_id);

    switch_address_space(proc);
}
```

Listing[6-2] proc.c

switch_address_space fools the CPU in order to switch to another address space ,

it needs to reset the stack top to point the the new process structure , set the cpu state to the process' value the issues an iret.

```
void
switch_address_space(proc_t *proc_to_run) {
    proc_to_run->seg_regs.fs = 0x23;
    proc_to_run->seg_regs.es = 0x23;
    proc_to_run->seg_regs.gs = 0x23;
    proc_to_run->seg_regs.ds = 0x23;
    write_cr3(proc_to_run->cr3);
    asm volatile("movl %0,%%esp":: "g" (proc_to_run) : "memory");
    asm volatile("popal");
    asm volatile("popl %gs\npopl %fs\npopl %es");
    asm volatile("popl %ds");
    asm volatile("iret\n5:\n");

    while(1);
}
```

Listing[6-3] Switching to process address space.

Appendix A – Problems faced :

On User environment initialization:-

=====

Mainly we initialize a user environment at this point to test how effective interrupts are when issued from a user space. the way we did a user space is to simply put another elf image on the disk shifted by several sectors from kernel and load it into memory and jump to it just like we did on boot sector. a far jump with user code segment. But we faced several problems.

1-We forgot to put a writable permission on the page to be able to load data from disk to memory. But such a behavior is not acceptable. since a code segment shouldn't be writable.

2- We forgot to provide a Ring 3 (SEG|3) Or to a user segment

3- We did depend on a statically compiled/linked elf as user environmet.

Goals are:

1- to be able to load code into memory without writable permission, of course this can be done by disabling writing to page after loading the code, but this should be the role of LDT later.

2- to be able to switch to user mode. which can be done only by "Fooling the x86 cpu"

3- Provide a data structure with processes operating.

On scheduling

=====

to schedule we need a kernel clock to issue interrupts. scheduling can be based on two criteria (time slicing, cycle slicing). however we chose to support time slicing scheduling first by using RTC(which is very naive but we chose to support legacy first). and we faced a problem that the RTC interrupt doesn't occur on protected mode. unlike the Intel 8253 PIT interrupt which does happen on protected mode. However, Some RTC timer code may not work on some real machines. The observed problem is a timer tick happened about once every second. I'm not sure why this is, and am trying to find a solution. This Makes RTC is not a serious solution for real life cases. However, on Bochs emulator an RTC interrupt occur from the slave PIC, But sadly Slave PIC doesn't support auto EOI like the master PIC therefore a handler has to issue an EOI once interrupt is handled which we did. But there was never a second interrupt from Slave PIC.