contact

CATernel: The CATReloaded Kernel Project ©

List of content :

Introduction to the CATernel Project

You can download the CATernel Project from this link.

There are prerequisites to use this kernel image:-

- First you need to be running on a *nix system
- the bochs IA-32 emulator (bochs)

You build the kernel by navigating to CATerenl/ Directory and running this command in your shell *make install*, after this you'll find the kernel image in *CATernel/kern/kernel/CATernel.img*, you will find some object files due to compile process.

To run this image you need first to download and install the bochs emulator. After you've done that you only need to navigate to *CATernel/*, and run the *bochs* command. the bochs emulator will show up running the *CATernel.img*.

Boot Loader

the Bootstrapping process is pretty basic one. But more features will be added once the project has a user space. Mainly our bootloader only move to the machine protected mode and activate Gate A20.

When a computer is powered the BIOS comes in control and initializes all data. then it looks for a valid bootloader 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.

```
00000000   FA FC 31 C0  8E D8 8E C0  8E D0 BC 00  7C E4 64 A8  02 75 FA B0  D1 E6 64 E4  64 A8 02 75  ..1.........|.d..u..
0000001C   FA B0 DF E6  60 0F 01 16  4C 7C 0F 20  C0 66 83 C8  01 0F 22 C0  EA 35 7C 08  00 66 B8 10  ....`...L|. .f....".
00000038   00 8E D8 8E  D0 8E C0 8E  E0 8E E8 E8  DA 00 00 00  EB FE 66 90  E7 FF 52 7C  00 00 00 00  ..................f.
00000054   00 00 00 00  00 00 FF FF  00 00 00 9A  CF 00 FF FF  00 00 00 92  CF 00 90 90  55 BA F7 01  ....................
00000070   00 00 89 E5  EC 25 C0 00  00 00 83 F8  40 75 F5 5D  C3 55 BA F7  01 00 00 89  E5 8B 4D 0C  .....%......@u.].U..
0000008C   57 EC 25 C0  00 00 00 83  F8 40 75 F5  BA F2 01 00  00 B0 01 EE  B2 F3 88 C8  EE 89 C8 B2  W.%......@u.........
000000A8   F4 C1 E8 08  EE 89 C8 B2  F5 C1 E8 10  EE C1 E9 18  B2 F6 88 C8  83 C8 E0 EE  B0 20 B2 F7  ....................
000000C4   EE EC 25 C0  00 00 00 83  F8 40 75 F5  8B 7D 08 B9  80 00 00 00  BA F0 01 00  00 FC F2 6D  ..%......@u..}......
000000E0   5F 5D C3 55  89 E5 57 56  8B 75 10 53  8B 5D 08 C1  EE 09 89 DF  46 81 E7 FF  FF FF 00 81  _].U..WV.u.S.]......
000000FC   E3 00 FE FF  00 03 7D 0C  EB 10 56 46  53 81 C3 00  02 00 00 E8  6D FF FF FF  58 5A 39 FB  ......}...VFS.......
00000118   72 EC 8D 65  F4 5B 5E 5F  5D C3 55 89  E5 56 53 6A  00 68 00 10  00 00 68 00  00 01 00 E8  r..e.[^_].U..VSj.h..
00000134   AB FF FF FF  83 C4 0C 81  3D 00 00 01  00 7F 45 4C  46 75 41 8B  1D 1C 00 01  00 0F B7 05  ........=.....ELFuA.
00000150   2C 00 01 00  81 C3 00 00  01 00 C1 E0  05 8D 34 03  EB 14 FF 73  04 FF 73 14  FF 73 08 83  ,.............4....s
0000016C   C3 20 E8 70  FF FF FF 83  C4 0C 39 F3  72 E8 A1 18  00 01 00 25  FF FF FF 00  FF E0 EB FE  . .p......9.r......%
00000188   BA 00 8A 00  00 B8 00 8A  FF FF 66 EF  B8 00 8E FF  FF 66 EF EB  FE 00 00 00  00 00 00 00  ..........f......f..
000001A4   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ....................
000001C0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ....................
000001DC   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ....................
000001F8   00 00 00 00  00 00 55 AA                                                                  ......U.
00000214
00000230
---  boot       --0x0/0x200----------------------------------------------------------------------------------
```

When the BIOS find a bootable image it loads the first 512 byte into address 0:07C00 then jump to it. then the bootsector comes in control. it starts execution in the real mode. so what we need is to enable the protected mode and to enable the A20 gate for more addressing.Ok!, Let's take a peek at our code. *Note,* Our code is written in GNU assembly to avoid compiling and linking problems.

Switching to proteced mode

```
.set Load,0x7c00        #The code start address
.set CODE_SEG,0x8       #the code Segment descriptor in the GDT
.set DATA_SEG,0x10      # the Data segment descriptor in the GDT

.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
#Here starts the A20 Gate enabling procedures
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

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
#here we are working on protected mode
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
#if failed just keep looping
spin:
        jmp spin
.p2align 2      #force a 4 byte alignment
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
gdt_end:
```

the instructions till setting the bootsector stack explains itself I think. On the *mov $start,%sp* instruction you set the start of the stack address since we are gonna start pushing and poping on our kernel loader.

enabling the A20 Gate

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 804x the IBM AT

```
D1      write output port. next byte written  to 0060
                              will be written to the 804x output port; the
                              original IBM AT and many compatibles use bit 1 of
                              the output port to control the A20 gate.
```

So in the next procedure *A20.2*, you check if the input buffer is full or not again.. then you write 0xDF to the 0x60 port which is written to the IBM AT 804x port which finally enables A20 gate.

```
DF      enable address line A20 (HP Vectra only???)
```

GDT Table

Then you load the GDT table into the GDT register via the *lgdt* instruction. But what is a GDT table? GDT table is used as a desciptor table...come on that is silly! I mean that in real mode you just address the memory with direct addressing via segments..Like this

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

in protected mode you cannot do this. You cannot directly access segments. that's where GDT table comes in handy. in the GDT table is a table where all the segments are defined.yet they are not stored as values but as descriptors. Descriptors has full informations about a segment. and it's 64 bit long. If you navigated to *CATernel/include/memvals.h* you will find an ascii graph illustrating the GDT table but I will just put that here..

```
 HOW GDT register is used
 This is the GDT register
 ---------------------------------------------
|31                16| 15                   0|
|                    |                        |
|     Base 0:15      |       Limit 0:15       |
|                    |                        |
|---------------------------------------------|
|63   56|55 52| 51  48|47      40|39        32|
| Base  |     | Limit | Access   |            |
| 24:31 |Flags| 16:19 | 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
```

This ASCII graph descripes exaclty what is a GDT descriptor. you see.. 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 most of the time equal to 0x1100. evantually 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...etc.
Let's Go back to our code. After loading the gdt address in the gdt register we set the protected mode in cr0 flag. and if you are using bochs you will see that message in the log after setting that flag..Here is a picture

```
00001385640i[BIOS ] bios_table_cur_addr: 0x000fbc04
00001511720i[VBIOS] VGABios $Id: vgabios.c,v 1.72 2011/06/27 17:58:32 vruppert Exp $^M
00001514727i[VBIOS] VBE Bios $Id: vbe.c,v 1.63 2011/04/14 16:10:09 vruppert Exp $
00001561552i[XGUI ] charmap update. Font Height is 16
00006968772i[XGUI ] charmap update. Font Height is 16
02085141664i[BIOS ] Booting from 0000:7c00
02197478920i[     ] Ctrl-C detected in signal handler.
02197478921i[     ] dbg: Quit
02197478921i[CPU0 ] CPU is in protected mode (active)
02197478921i[CPU0 ] CS.mode = 32 bit
02197478921i[CPU0 ] SS.mode = 32 bit
02197478921i[CPU0 ] | EAX=00000000  EBX=00010200  ECX=00000001  EDX=000001f7
02197478921i[CPU0 ] | ESP=00007bbc  EBP=00007bc0  ESI=00000002  EDI=00011000
02197478921i[CPU0 ] | IOPL=0 id vip vif ac vm rf nt of df if tf SF zf af PF CF
02197478921i[CPU0 ] | SEG selector     base    limit G D
02197478921i[CPU0 ] | SEG sltr(index|ti|rpl)     base    limit G D
02197478921i[CPU0 ] |  CS:0008( 0001| 0|  0) 00000000 ffffffff 1 1
02197478921i[CPU0 ] |  DS:0010( 0002| 0|  0) 00000000 ffffffff 1 1
02197478921i[CPU0 ] |  SS:0010( 0002| 0|  0) 00000000 ffffffff 1 1
02197478921i[CPU0 ] |  ES:0010( 0002| 0|  0) 00000000 ffffffff 1 1
02197478921i[CPU0 ] |  FS:0010( 0002| 0|  0) 00000000 ffffffff 1 1
02197478921i[CPU0 ] |  GS:0010( 0002| 0|  0) 00000000 ffffffff 1 1
02197478921i[CPU0 ] | EIP=00007c8e (00007c8e)
02197478921i[CPU0 ] | CR0=0x60000011 CR2=0x00000000
02197478921i[CPU0 ] | CR3=0x00000000 CR4=0x00000000
02197478921i[CMOS ] Last time is 1320447041 (Sat Nov  5 00:50:41 2011)
02197478921i[XGUI ] Exit
02197478921i[CTRL ] quit_sim called with exit code 0
```

If you tried to skip this instruction via jumping to sping before it you'll get this instead....here is another picture

```
00001385835i[PCI  ] 440FX PMC write to PAM register 59 (TLB Flush)
00001386679i[BIOS ] bios_table_cur_addr: 0x000fbc04
00001512759i[VBIOS] VGABios $Id: vgabios.c,v 1.72 2011/06/27 17:58:32 vruppert Exp $^M
00001515766i[VBIOS] VBE Bios $Id: vbe.c,v 1.63 2011/04/14 16:10:09 vruppert Exp $
00001871099i[XGUI ] charmap update. Font Height is 16
00423339759i[BIOS ] Booting from 0000:7c00
00465902583i[     ] Ctrl-C detected in signal handler.
00465902583i[     ] dbg: Quit
00465902583i[CPU0 ] CPU is in real mode (active)
00465902583i[CPU0 ] CS.mode = 16 bit
00465902583i[CPU0 ] SS.mode = 16 bit
00465902583i[CPU0 ] | EAX=000000df  EBX=00000000  ECX=00000000  EDX=00000000
00465902583i[CPU0 ] | ESP=00007c00  EBP=00000000  ESI=000e91df  EDI=0000ffac
00465902583i[CPU0 ] | IOPL=0 id vip vif ac vm rf nt of df if tf sf ZF af PF cf
00465902583i[CPU0 ] | SEG selector     base    limit G D
00465902583i[CPU0 ] | SEG sltr(index|ti|rpl)     base    limit G D
00465902583i[CPU0 ] |  CS:0000( 0004| 0|  0) 00000000 0000ffff 0 0
00465902583i[CPU0 ] |  DS:0000( 0005| 0|  0) 00000000 0000ffff 0 0
00465902583i[CPU0 ] |  SS:0000( 0005| 0|  0) 00000000 0000ffff 0 0
00465902583i[CPU0 ] |  ES:0000( 0005| 0|  0) 00000000 0000ffff 0 0
00465902583i[CPU0 ] |  FS:0000( 0005| 0|  0) 00000000 0000ffff 0 0
00465902583i[CPU0 ] |  GS:0000( 0005| 0|  0) 00000000 0000ffff 0 0
00465902583i[CPU0 ] | EIP=00007c4a (00007c4a)
00465902583i[CPU0 ] | CR0=0x60000010 CR2=0x00000000
00465902583i[CPU0 ] | CR3=0x00000000 CR4=0x00000000
00465902583i[CMOS ] Last time is 1320446863 (Sat Nov  5 00:47:43 2011)
00465902583i[XGUI ] Exit
00465902583i[CTRL ] quit_sim called with exit code 0
```

Notice the first bit in every picture in cr0.

Protected Mode

Doing a far jump using ljmp and using the code segment value as a segment and next procedure as an offset modified the CS value to the one in our GDT table. 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. **#WIN!**

Loading the Kernel Image

Our kernel image is an elf binary formatted. so what you really need before reading this is a deep knowledge of ELF. I've wrote the elf.h header which you can find in *CATernel/include/elf.h* to make it easier for us to read our kernel. But I will give you an ELF crash course here.

ELF file format

ELF stands for Executable and Linking Format. ELF is used in object files like (.o) files and shared libraries (.so) and (.kld) kernel loadable modules.
if you navigated to any of your (bin) directories and tried running the following command *readelf -h (any executable file goes here)* I've used /bin/ls as a file and that was the output..

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8049cd0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          103368 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         29
  Section header string table index: 28
```

Magic is the magic signature of ELF file that's how ELF files are identified, Let's go through our *elf.h* header and try to know what is that?

```
/* Start of Magic Definitions */
#define ELF_MAGIC 0x464C457F
#define MAGIC_LEN 16

#define M_CLASS_OFF 4   //File Class offset
#define M_CLASSNONE 0   //Invalid Class
#define M_CLASS32 1     //32-bit Objects
#define M_CLASS64 2     //64-bit Objects
#define M_CLASSNUM 3

#define M_DATA_OFF 5    //Data encoding byte offset
#define M_DATANONE 0    //Invalid Data encoding
#define M_DATA2LE       1       // 2's complement Little endian
#define M_DATA2BE 2     //2's complement Big endian
#define M_DATANUM 3

#define M_VERSION 6     //File version offset

#define M_OSABI 7               //OS/ABI offset
#define M_OSABI_SYSV 0  //Unix System V
#define M_OSABI_HPUX 1  //HP-UX

#define M_ABIVERSION 8  //ABI version offset
#define M_ELF_PADDING 9 //Padding bytes offset
```

As you can see the ELF_MAGIC is the same 4 bytes as above but in little endian. the fifth byte is 0x01 which means that this file is in 32-bit format. sixth bit is also 0x01 which means that data are in 2's compelement format Little endian. the seventh bit is 0 because this is a unix system V.and finally ABI version is 0 at the eights offset. now try to hexdump the first 32 byte of the executable file. I got this myself

```
saad@MachineOnLinux:~/CPrograms/CATernel$ hd -n 32 /bin/ls
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 03 00 01 00 00 00  d0 9c 04 08 34 00 00 00  |............4...|
```

Well..Let's take a look at our *elf.h*

```
/* File types */
#define T_TYPE_NONE 0
#define T_TYPE_REL 1
#define T_TYPE_EXEC 2
#define T_TYPE_DYN 3
#define T_TYPE_CORE 4
#define T_TYPE_LOPROC 0xff00
#define T_TYPE_HIPROC 0xffff

/*Machine types "since i will only work in i386 i will def one value"*/
#define M_MACHINE_I386  3       //intel Machine
```

the two bytes at offset 0x10 which are 02 00 means that the type of this file is executable. the next two bytes are 03 00 which has the machine type. which means intel i386. and i only supported that in my *elf.h*. next two bytes has the version number which is 01 00..yet another snippet from our elf.h

```
/* Version types */
#define V_VERSION_NONE 0
```

```
#define V_VERSION_CURRENT 1
#define V_VERSION_NUM 2
```

skip the next two bytes, the four bytes at offset 0x18 which are d0 9c 04 08 are the entry point address for this code which is 0x08049cd0.
the next byte indicates the offset of start of program headers which is 0x34 = 52 in decimal.. ok let's hexdump more of that file!

```
saad@MachineOnLinux:~/CPrograms/CATernel$ hd -n 128 /bin/ls
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 03 00 01 00 00 00  d0 9c 04 08 34 00 00 00  |............4...|
00000020  c8 93 01 00 00 00 00 00  34 00 20 00 09 00 28 00  |........4. ...(.|
00000030  1d 00 1c 00 06 00 00 00  34 00 00 00 34 80 04 08  |........4...4...|
00000040  34 80 04 08 20 01 00 00  20 01 00 00 05 00 00 00  |4... ... .......|
00000050  04 00 00 00 03 00 00 00  54 01 00 00 54 81 04 08  |........T...T...|
00000060  54 81 04 08 13 00 00 00  13 00 00 00 04 00 00 00  |T...............|
00000070  01 00 00 00 01 00 00 00  00 00 00 00 00 80 04 08  |................|
```

the four bytes at offset 0x20 indicate the offset of section headers c8 93 01 00 which is 0x193c8 and 103368 in decimal. then the four flags bytes which are all zeroes. then the two bytes at offset 0x28 indicates the size of the elf header which is 0x34 and 52 bytes in decimal. and the next two bytes indicate the size of program headers which is 0x20 = 32 bytes in decimal. next two bytes are the number of program headers which are 9 headers. and then size of section headers which is 0x28 = 40 header. and then the number of section headers which are 0x1d or 29. and finally the string table index of the section header which is 28.

Now to sections

You can easily know the sections in an ELF file using the following command

```
saad@MachineOnLinux:~/CPrograms/CATernel$ readelf -S kern/kernel/kernel
There are 11 section headers, starting at offset 0xa390:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        f0100000 001000 000c20 00  AX  0   0  4
  [ 2] .rodata           PROGBITS        f0100c20 001c20 000398 00   A  0   0  4
  [ 3] .stab             PROGBITS        f0100fb8 001fb8 000001 0c  WA  4   0  1
  [ 4] .stabstr          STRTAB          f0100fb9 001fb9 000001 00  WA  0   0  1
  [ 5] .data             PROGBITS        f0101000 002000 008320 00  WA  0   0 4096
  [ 6] .bss              NOBITS          f0109320 00a320 000616 00  WA  0   0  4
  [ 7] .comment          PROGBITS        00000000 00a320 000023 01  MS  0   0  1
  [ 8] .shstrtab         STRTAB          00000000 00a343 00004c 00      0   0  1
  [ 9] .symtab           SYMTAB          00000000 00a548 0004a0 10     10  36  4
  [10] .strtab           STRTAB          00000000 00a9e8 000282 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

that's what I got! you see sections like NULL section..I dont care about that.. But you see the *.text* section Address 0xf0100000 which is the address where this section should be put into. and the Off which is the offset and size has the size of the section. and Flg indicates the section attributes which is AX(Section is readable and executable). Actually .text section mostly is the name of the section to be exectued into the Code segment. *.data* segment is the segment that has pre-defined variables..anyway every section has it's usage but now we only care about *.text* section.

Now to program headers!

Section header table is not loaded into memory because kernel will not be able to use this table. using these sections is done via program headers.
simply running the next command gives you the program headers you have in your binary.

```
saad@MachineOnLinux:~/CPrograms/CATernel$ readelf -W -l kern/kernel/kernel

Elf file type is EXEC (Executable file)
Entry point 0xf0100014
There are 2 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0xf0100000 0xf0100000 0x09320 0x09936 RWE 0x1000
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x4

 Section to Segment mapping:
  Segment Sections...
   00     .text .rodata .stab .stabstr .data .bss
   01
```

here we have only two program headers.. LOAD and GNU_STACK.
we are now just interested in the LOAD program header.
Enough ELFing! It's not yet chrismas!! *trollface*

Loading Kernel

this is the code that loads the kernel from Hard disk (Read x86.h before you read this)

```
#include
#include

#define SECTOR 512
#define ELFHDR ((struct elfhdr *) 0x10000)
void readsect(void*,uint32_t);
void readseg(uint32_t,uint32_t,uint32_t);
void
cmain(void)
{
        struct proghdr *p,*p2;  // program headers;

        readseg((uint32_t) ELFHDR,SECTOR*8,0); // Load kernel from disk to memory
        if( ELFHDR->magic != ELF_MAGIC ) //Check if the kernel is ELF file format, if it doesn't match get the hell out
                goto getout;


        p=(struct proghdr *) ( (uint8_t *) ELFHDR+ ELFHDR->phroff); // Load program segments
        p2= p + ELFHDR->phrnum;

        for (; p < p2 ; p++)
                //LOAD THEM INTO MEMORY
                readseg(p->vaddr,p->memsz,p->offset);
        __asm __volatile("jmp   %%eax"::"a" ( (uint32_t *)((ELFHDR->entry)&0xffffff)));
        while(1);
getout:
        while(1);
}
void
waitdisk(void){
         while ((inb(0x1F7) & 0xC0) != 0x40);
}
void
readseg(uint32_t va,uint32_t count,uint32_t offset)
{

        uint32_t end_va;
        va &= 0xFFFFFF;
        end_va = va + count;
        va &= ~(SECTOR -1);
        offset = (offset/SECTOR)+1;
        while(va < end_va) {
                readsect((uint8_t *)va,offset);
                va += SECTOR;
                offset++;
        }
}

void
readsect(void *dst,uint32_t offset)
{
        waitdisk();
        outb(1,0x1F2);          // sector count
        outb(offset,0x1F3);     // sector number
        outb(offset >> 8 ,0x1F4); //Cylinder Low
        outb(offset >> 16,0x1F5); //Cylinder High
        outb( (offset >> 24) | (0xE0),0x1F6);
        outb(0x20,0x1F7);       //Read sectors with a retry
        waitdisk();

        insl(dst,SECTOR/4,0x1F0); // Load binaries from disk to dst address (ELFHDR)
}
```

first we read 8 sectors of the kernel, But how does this work?
readseg function is easy to guess, but the readsect function needs more explaination. first we wait till disk is available, then you pass to 0x1F2 one! which means you want to read one sector. then you pass the offset of the sector you want to read. then you pass the cylinder Low and high numbers. the Outb for 0x1F6 passes the head value. and on out operation for 0x1f7 you pass how you want to read the value and 0x20 means you want to read it with a retry. and finally you read them from the data register by repeating the insl instruction (check x86.h).
after loading the kernel we verify if it is an ELF image, if it is not we just do and infinite loop. then we load the program headers into memory and Dang we jump to the entry point and start executing our kernel.

Kernel Fetching

After Loading our kernel into memory we have to setup our memory layout. check our this code

```
#include

#define RELOC(x) ((x) - KERNEL_ADDR)
.set CODE_SEG,0x8
.set DATA_SEG,0x10
```

```
#define FLAGS ((1<<0) | (1<<1))
#define CHECKSUM (-( 0x1BADB002 + FLAGS))

.text
jmp _start
# The Multiboot header
.align 4
.long 0x1BADB002
.long FLAGS
.long CHECKSUM

.global         _start
_start:
        movw    $0x1234,0x472                   # soft-reboot
        lgdt    RELOC(gdt_table)                # Load the GDT Register
        movl    $DATA_SEG, %eax                 # Load the Data Segment
        movw    %ax,%ds                         # Copy Data Segment
        movw    %ax,%es
        movw    %ax,%ss
        ljmp    $CODE_SEG,$get_to_work          # Do a far jump to go to protected mode

get_to_work:
        xor     %ebp,%ebp                       # Clear the frame pointer
        movl    $(kernel_stack_end),%esp                # Load the stack into the stack pointer
        call    work_it_out

spin:
        jmp     spin

# Virtual Page Table
############################################################################
.data
        .global virtpgt
        .set    virtpgt, VIRTPGT
        .global virtpgd
        .set    virtpgd, (VIRTPGT + (VIRTPGT>>10))


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

#Global Descriptor table
#############################################################################
#YOU REALLY NEED TO READ THE MEMVALS HEADER BEFORE TRYING TO UNDERSTAND THIS
        .p2align        2               # pad alloc by 4
gdt_table:
        .word gdt-gdt_end-1
        .long RELOC(gdt)
gdt:
        .long 0,0
        SEGMENT(0xffffffff,-KERNEL_ADDR, SEGACS_RW|SEGACS_X)    # code seg
        SEGMENT(0xffffffff,-KERNEL_ADDR, SEGACS_RW)     # data seg
gdt_end:
```

Of course you need to look into the included header. anyway let's skip the code now and look into the layout. first we set the address of *virtpgt* this is the address of the start of the data segment which we will use as a virtual page table. then the *virtpgd* which we will use as the virtual page directory.

then we need to set the stack up. we put it after the virtual page directory, yet we will pad the allocation for a 2^0x1000 alignment. that will give the stack enough space to work!

finally we setup our gdt table.

if you tried objdump(ing) the kernel you will find that the stack is big enough!..well that's what I got!

```
f0101000 <kernel_stack> :
        ...

f0109000 <kernel_stack_end>:
f0109000:       e7 ff                   out    %eax,$0xff
f0109002:       06                      push   %es
f0109003:       90                      nop
f0109004:       10 00                   adc    %al,(%eax)

f0109006 <gdt>:
        ...
f010900e:       ff                      (bad)
f010900f:       ff 00                   incl   (%eax)
f0109011:       00 00                   add    %al,(%eax)
f0109013:       9a cf 10 ff ff 00 00    lcall  $0x0,$0xffff10cf
```

```
f010901a:       00 92 cf 10 00 00       add    %dl,0x10cf(%edx)

f010901e <gdt_end>:
```

You see we've got enought space for stack.Now let's move back to the code..!
First we make a soft-reboot then we load the GDT table at it's address+f0000000h !
Umm, remember in our main.c boot sector we jumped at ELFHDR->entry & 0xffffff, we masked the entry so actually now we are working on address 0x100000
so we just load the gdt table from the virtual address. then we reload the data segment and stack segment..etc with the data segment descriptor index. and we make the old far jump again to reload the code segment register, and initiate the frame pointer and stack pointer and get to work! and our kernel starts!
Anyway, you might have been asking all the way down till here, what are those FLAGS and CHECKSUM and that 0x1BADB002!
try reading this article about MBRs and how to fetch a kernel (Bare Bones).

Kernel Execution

Once we jump to our kernel real binary, we first initalize some devices, then we clear up the screen. and re-initalize! and start our prompt!
This actually wont get in more details since you need to know first how did we support these devices!

Supporting Devices

CMOS and RTC

We've supported the CMOS/RTC...Actually most of it! in *cmos.c* and *cmos.h*, for much of future usage.
The implementation of the code is a bit easy to understand a pretty much dynamic! Let's take a look at the code!

```
#include <types.h>
#include <x86.h>
#include <cmos.h>


/*
 *  Status B Power options Refer to Ports.lt for details
 */
uint8_t cmos_set_power_stat(uint8_t stat)
{
uint8_t update , New_Stat;
//check that the register is not in update mode
 while(update == 0x80){
        outb(RTC_STATUS_A,CMOS_INDEXPORT);
        update = inb(CMOS_DATAPORT);
        }
        cli();
        outb(RTC_STATUS_B,CMOS_INDEXPORT);
        New_Stat=inb(CMOS_DATAPORT); //read initial state
 //those 3 sets the respective bit to there so we mask with AND
 if(stat==STAT_RUN || stat==STAT_CAL_BCD || stat==STAT_CAL_HR12)
          New_Stat &= stat;
 else
          New_Stat |= stat;
 outb(RTC_STATUS_B,CMOS_INDEXPORT);
 outb(New_Stat,CMOS_DATAPORT);//write the new status to the port.

 return New_Stat;
 //for debugging the function will be void later.

}


//Get RTC Values
uint8_t
cmos_get_time(uint8_t value) //value holds whether it's day,month,seconds,etc..
{
        uint8_t time;
        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);
        time = inb(CMOS_DATAPORT);
        return time;
}
```

Umm,Let's first talk about the 2nd function which is *cmos_get_time*. you see this function takes a byte as arugment and returns one byte, the byte it takes is the type of time you want to read! the argument must be one of these constants. which are defined in *cmos.h* header.

```
#define RTC_SECONDS     0x0
```

```
#define RTC_ALRMSECOND  0x1
#define RTC_MINUTES     0x2
#define RTC_ALRMMINUTE  0x3
#define RTC_HOUR        0x4
#define RTC_ALRMHOUR    0x5
#define RTC_DAY_WEEK    0x6
#define RTC_DAY_MONTH   0x7
#define RTC_MONTH       0x8
#define RTC_YEAR        0x9
```

the time value will return as a hexdecimal decimal like format (BCD). So if today is 17 and you're reading the RTC_DAY_MONTH. you will get 0x17 as a return value instead of 0x11.
inside the function we first check if the rtc is updating using the *0A* status register which means that the rtc is updating if the last bit was set(10000000 = 0x80), if it is updating just loop until it's not updating. then we clear interrupts flag and output what we want to read (one of the values above!) and we take output from the data port, if you are asking what are those index and data ports. then you really need to look up devices and I/O. But, I will make it a bit easy for you! these are the CMOS/RTC specs.

```
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)
```

this is the CMOS index register and from that we give an order (we pass the index we want to read/write) this must be followed by an operation on the CMOS data register.

```
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)
```

that is the data port and (some) of the indeces you can use!
Also the status register 0B is supported
as for CMOS status B , If you refrence CMOS ports you'll find this
Here's the the values in the register

```
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.
```

It's implemented in the function cmos_set_power_stat , what happens is that it reads the register value from the RTC into New_Stat , then in order to set the bits to the mode you want , simple OR for 1's and AND for 0's finally the resulting values will be put in the register again
The masking valuse are added in the cmos.h header

```
#define STAT_HALT       0x80
#define STAT_PER_INTR   0x40
#define STAT_ALRM_INTR  0x20
#define STAT_UPDT_INTR  0x10
#define STAT_SQRWV_INTR 0x08
#define STAT_CAL_BIN    0x04
#define STAT_CAL_BCD    0xFB
#define STAT_CAL_HR24   0x02
#define STAT_CAL_HR12   0xFD
#define STAT_DAY_LGHT   0x01
```

Color Graphic Adapter

on CATernel we will support the CGA in text mode.
we have two text modes (80x25) and (40x25), both has 8x8 pixel characters. result resolution is either 320x200 or 640x200 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.

```
Here is the attribute byte specifications:-
```

```
        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)
```

I wont talk about graphics mode since we wont need it :>
In our *video.c* code there is a function that is called *cga_set_attr* here we set an attribute of the text that will be written on screen to whatever color we need. You can find colors attributes as constants in the *video.h* header..Here is a snippet

```
#define COLOR_DARK_GRAY 0x0700
#define COLOR_BLUE      0x0100
#define COLOR_GREEN     0x0200
#define COLOR_RED       0x0400
#define COLOR_GRAY      0x0800
#define COLOR_WHITE     0x0f00
#define BACKGROUND_GRAY 0x7000
#define BACKGROUND_BLUE 0x1000
#define BACKGROUND_GREEN        0x2000
#define BACKGROUND_RED  0x4000
#define BACKGROUND_BLINK        0x8000
#define BACKGROUND_WHITE        0xF000
```

you pass one or more (ORRED values) as argument to the *cga_set_attr* function and we get our text colored on the screen.
CGA I/O 0x3D0~0x3DF Registers

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

0x3D8   -       CGA mode control register
|->     bit 5 - blink register (if this bit is set the bits with attribute bit 7 will start blinking)
|->     bit 4 - 640*200 High-Resolution register ( if set CGA works in 2 color 640 wide mode, if not CGA works in 320 wide mode
|->     bit 3 - video register (if cleared the screen wont output)
|->     bit 2 - Monochrome (if set to one you get a 2 colors output black/white, if cleared you get compsite color mode)
|->     bit 1 - text mode (if set the video ram is treated bitmap graphics NOT TEXT, if cleared you work on text mode)
|->     bit 0 - Resolution (if set the video output is 80*25*2 if cleared the output is 40*25*2)

0x3D9   -       Palette Register / Color control register
|->     bit5 - chooses color set (if set color set is red/green/(brown/yellow), if cleared color set is magenta/cyan/white
|->     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 backround 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, if this bit is set then RAM can be accessed without causing snow
|->     bit2 - the current status of the light pen switch cleared if on set if off
|->     bit1 - light pen trigger, the trigger is cleared by writing any value to port 0x3DB port
|->     bit0 - if this bit is set then the CPU can access the video ram and cause snow

0x3DB   -       clear light pen trigger
0x3DC   -       set   light pen trigger
```

if you took a look at video.c code you will find setters and getters for position. these work on the Index port 0x3d4 and data port 0x3d5. so I supply the index of the position i want to read 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. you might find the shifting and anding little confusing. so to clear that up i will give and 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. and we actually wont need that!
on the *cga_init* function:
we set the address of the character buffer of the video RAM, and get the current cursor position so we don't work on garbage!

PS/2 Auxiliary Keyboard

Illustrating how keyboard I/O works will take much much time! and space..you only need to know how it works since the keyboard is already supported.
if you navigated to our *CATernel/kernel/kbc.c* you will find the keyboard driver that we are using. First we define some of the special keys.

```
#define ESCODE  (1<<6)
#define CTL     (1<<1)
#define SHIFT   (1<<0)
#define ALT     (1<<2)
#define CAPSLOCK (1<<3)
#define NUMLOCK  (1<<4)
#define SCROLLLOCK (1<<5)
```

Then we define some of the scancodes we'll need you can see all the scancodes here.
then we map our normal and shifted and ctrled scancodes with their scancode values. try this link
to read from the keyboard we use our *kbc_data* function, we first check the status port if there is data in the output buffer or not.
This is the PS/2 Keyboard status register specs

```
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: 0=power up or reset  1=selftest OK
                 bit 1 = 1 input buffer full (input 60/64 has data for 8042)
                 bit 0 = 1 output buffer full (output 60 has data for system)
```

and these are some constants from our *kbc.h* header.

```
#define KBC_DATAIN      0x01    /** New Data in buffer **/
#define KBC_FULLBUF     0x02    /** Buffer is full **/
#define KBC_REBOOT      0x04    /** soft reboot **/
#define KBC_COMMAND     0x08    /** data in output register is a command **/
#define KBC_SECLOCK     0x10    /** Security lock engaged **/
#define KBC_TTIMEOUT    0x20    /** transmission timeout error **/
#define KBC_RTIMEOUT    0x40    /** recieve timeout error **/
#define KBC_PARITY      0x80    /** Parity error **/
```

in our *kbc_data* function we check if the bit 0 if set or not .if it is set we continue, and read the data buffer.
then we check if the code we read is and Escaped scancode or if the key is released. and if it was escaped scancode. specify the character and the CAPS.
But how do we actually read in the console? - we read via a buffer and written character counter and read character counter

```
void
console_interrupt(int (*intr)(void)){
        int c;
        while ((c = (*intr)()) != -1){
                if(c==0)
                        continue;
                cons.buf[cons.wpos++] = c;
                if(cons.wpos == MAXBUFSIZE)
                        cons.wpos=0;
        }

}
```

this function reads recived characters from the paramter function to the buffer..as long as the paramter function returns something!
In our *kbc.c* the interrupt function is

```
void
kbc_interrupt(void)
{
        console_interrupt(kbc_data);
}
```

which executes the previous function giving *kbc_data* as paramter function. so when you want to read a character you just do this interrupt and read the buffer.

```
int
console_getc(void){
        int c;
        kbc_interrupt();
        if (cons.rpos != cons.wpos){
                c = cons.buf[cons.rpos++];
                if(cons.rpos == MAXBUFSIZE)
                        cons.rpos=0;
                return c;
        }
        return 0;
}
```

this executes the interrupt and if a character was read the function returns it.
but of course you need to wait for the user to input the character.

```
int
getchar(void){
        int ch;
        while((ch=console_getc()) == 0);
        return ch;

}
```

There!! that's how you get a character from a user!

Supporting freeBSD and Linux techniques

Standard I/O

the Standard Output (printk) is found in *printf.c* try reading the code..I will document it later :).

That's our Compiled Kernel