Mini OS

Lab One
(Phase One)

Agenda

- How computer works ?
- Review AT&T assembly
- Review pointer arithematic and stack frames in C
- Tools and Material
- Material Review
- Requirements

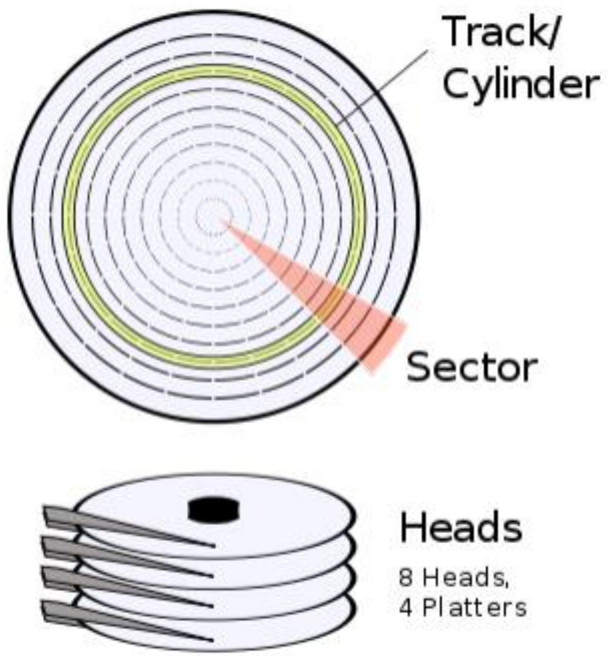
1.How computer works ?

What happens when we power on ?

- The bios starts from a hardwired place in the ROM
- It does some initializations (copy IVT & others)
- Some hardware checking is initiated
- Search for a bootable device (search for a bootloader)

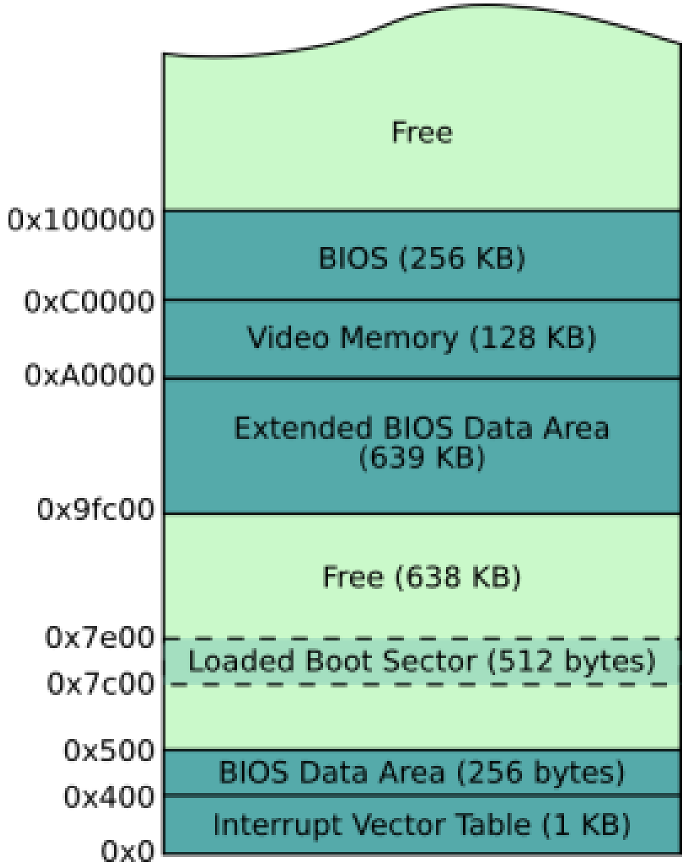
Where to search for bootable device ?!

- BIOS must read specific sectors of data (usually 512 bytes in size) from Cylinder 0, Head 0, Sector 0.*
- It checks for the magic number 0xaa55 at the last two bytes
- Copy this sector to 0x7c00
- Start running the code in realmode (16 bit mode)



Bootloader

- Bootloader runs in16 bit real mode for backward comptability
- Since bootloader is only 512 byte , it is usually used to load another stage with flexible size to load the operating system
- In real Mode , maximum memory we can access is 1 MB so it seeks to transfer to ProtectMode


Free
0x100000
BIOS (256 KB)
0xC0000
Video Memory (128 KB)
0xA0000
Extended BIOS Data Area (639 KB)
0x9fc00
Free (638 KB)
0x7e00
Loaded Boot Sector (512 bytes)
0x7c00
0x500
BIOS Data Area (256 bytes)
0x400
Interrupt Vector Table (1 KB)
0x0

Real Mode

- Less than 1 MB of RAM is available for use.
- There is no hardware-based memory protection (GDT), nor virtual memory.
- The default CPU operand length is only 16 bits.
- The memory addressing modes provided are more restrictive than other CPU modes.
- Accessing more than 64k requires the use of segment register that are difficult to work with.
- Access memory through "Segment:Offset"

Protected Mode

- No Interrupts
- Allow paging
- Allow protection
- Accessing Memory more than 1MB
- Allow Virtual Memory
- Can work in 32bit mode (and 64 in modern pcs)
- Access memory through "Descriptor:Offset"

GDT, LDT , IDT

- GDT , Global Descriptor table
- LDT , Local Descriptor table
- IDT , Interrupt Descriptor table

we will talk about them later in next lab (phase isA)

Switch from RM to PM

- To make the switch we need to
  - enable A20 line
  - Disable interrupts
  - load GDT , IDT
  - set protected bit to enable
  - far Jmp to 32 instruction using Descriptor:offset
  - Adjust memory mapping (segment registers)
  - Enable interrupts

2. Review At&T assembly

AT&T vs Intel assembly

	AT&T	Intel
Register name	prefixed by "%" i.e. %eax , %ecx,%ebp	written directly eax
src/dest order	source come first, i.e. movl src,dest	destination come first mov dest,src
immediate values	prefix with $,i.e. movl $0xd00dh, %eax	written directly ,i.e mov eax, 0d00dh
operator size specification	each command have to be followed by (l,w,b) to specify the widths of the operands (l --> long , w --> word , b --> byte) movl %eax, %ebx movw %ax, %bx movb %ah, %bh	Same command for all sizes and the operand adjust the size if needed mov ebx , eax mov bx , ax mov bh ,ah

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

3. Review pointer arithematic and stack frames in C

Remember

- Pointer is an address of a variable
- int arr[] = {1 , 2, 3 , 4 } is a constant pointer to arr
- assume address of arr[0] = 0x100
- int * ptr is a integer pointer
  - ptr = &arr[0]   // ptr = 0x100 , *ptr = 1
  - ptr++   // ptr = &arr[1] = 0x104 (size of integer = 4)
  - *ptr = arr[3]   // ptr = 0x104 , *ptr=4 , arr[1] = 4
  - int x =*(ptr-1)   // x = 1 = arr[0]
  - x= (int)ptr+1   // x = 0x103

- http://pdosnew.csail.mit.edu/6.828/2014/readings/pointers.pdf
- http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/BitOp/pointer.html

4. Tools and Material

Getting Tools (assume Ubuntu)

To install some package use the following command

```
sudo apt-get install package_name
```

- If you are using Ubuntu , make sure you have the following
  - gcc
  - gdb
  - objdump
  - git
- Qemu (virtual machine)
  - git clone https://github.com/geofft/qemu.git -b 6.828-1.7.0
  - ./configure --disable-kvm [--prefix=PFX] [--target-list="i386-softmmu x86_64-softmmu"]
  - make && make install
- To test your tool chain use the following command

  ```
  objdump -i | grep 'elf32-i386'
  ```

  the result shouldn't be empty

error

- For tools usage visit http://pdos.csail.mit.edu/6.828/2014/labguide.html
- if you have any problem in compiler tools please refer to

  http://pdos.csail.mit.edu/6.828/2014/tools.html

Material

- Original Document (if you didn't attend the section it is recommended to use this one)

  http://pdosnew.csail.mit.edu/6.828/2014/labs/lab1/
- Our optimized document version here
- To get first lab material use the following command

```
git clone http://pdos.csail.mit.edu/6.828/2014-jos.git lab
```

you can always refer to the website reference page it contains valuable resources

5. Material Review

Material structure

- boot/
  - contains the source files to make bootloader
- kern/
  - contains the source files to make kernel
- inc/
  - contain headers , type definitions
- lib/
  - contains our made C library because there is no std
- obj/
  - contains output of make
- user/
  - contains the source files to make user programs
- conf/
  - contains environment configuration , use it if your make file can't locate QEMU

JOS obj/

- When building JOS, the makefile also produces some additional output files that may prove useful while debugging:

  - obj/boot/boot.asm, obj/kern/kernel.asm, obj/user/hello.asm, etc.

  Assembly code listings for the bootloader, kernel, and user programs.

  - obj/kern/kernel.sym, obj/user/hello.sym, etc.

  Symbol tables for the kernel and user programs.

  - obj/boot/boot.out, obj/kern/kernel, obj/user/hello, etc

  Linked ELF images of the kernel and user programs. These contain symbol information that can be used by GDB.

Let's take a look on Make file

- GNUmakefile
  - parent Makefile
- Makefrag
  - bootloader Makefile
- sign.pl
  - it adds the signature to the bootloader in order to be defined
- Good reference for Kbuild Makefiles

http://lwn.net/Articles/21835/

6. Requirements

Required to do :)

- Pick your partner
- Install tools
- Download materials
- Read the material well and answer the questions at each exercise
- Implement the missing code
  - Follow Naming Convention found in CODING file in the material
- For Questions use google first :D , then ask on our emails using the subject [OS][MiniOS]question
- For delivery , attach a Pdf document with questions answers and your whole folder directory as zipped file to our emails with the subject [OS][MiniOS]delivery#1
- **Hint:** kern/Monitor.c contains the interface of our kernel

Questions

- Why we don't Do it from scratch ?
- Why we Do it in first place ?