Operating System Design & Implementation

Lab2: Kernel Booting

TA:陳勇旗

Objective:

In this lab you can learn

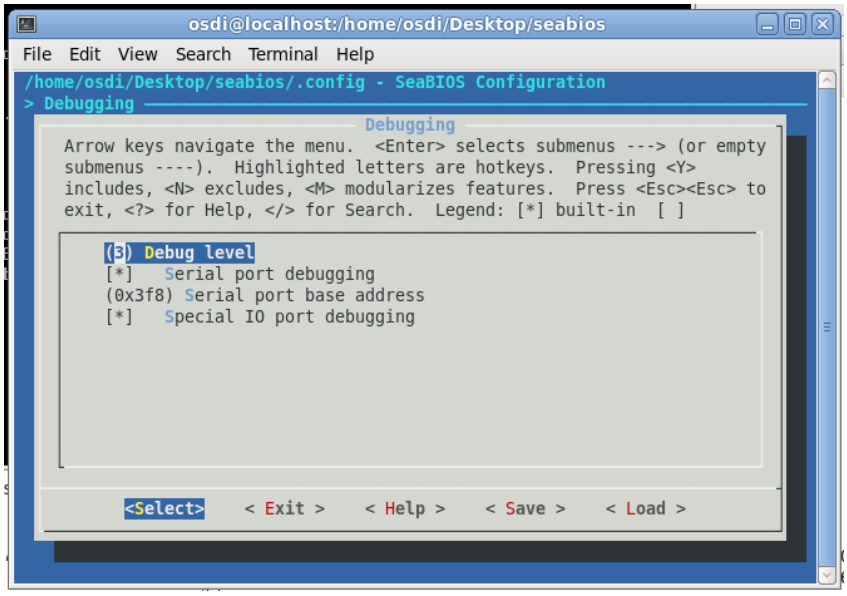
- Build the customize BIOS and show some message on system startup
- Understand the kernel booting flow and boot a “hello world” program
- Modify linux 0.11 bootsect to support multi booting

1. Lab2.1 - Build the BIOS and add some message before system start

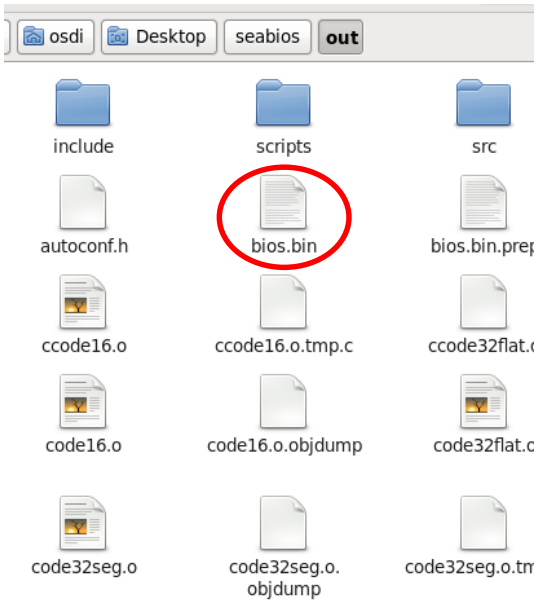
We’ll use **SeaBIOS 1.7.5.1** as our BIOS. You can download it by the following link.

http://code.coreboot.org/p/seabios/downloads/

In “make menuconfig” step you can change the debug level, it will show the BIOS debug message when system booting.



After make step you will see the bios.bin rom file in the out folder.

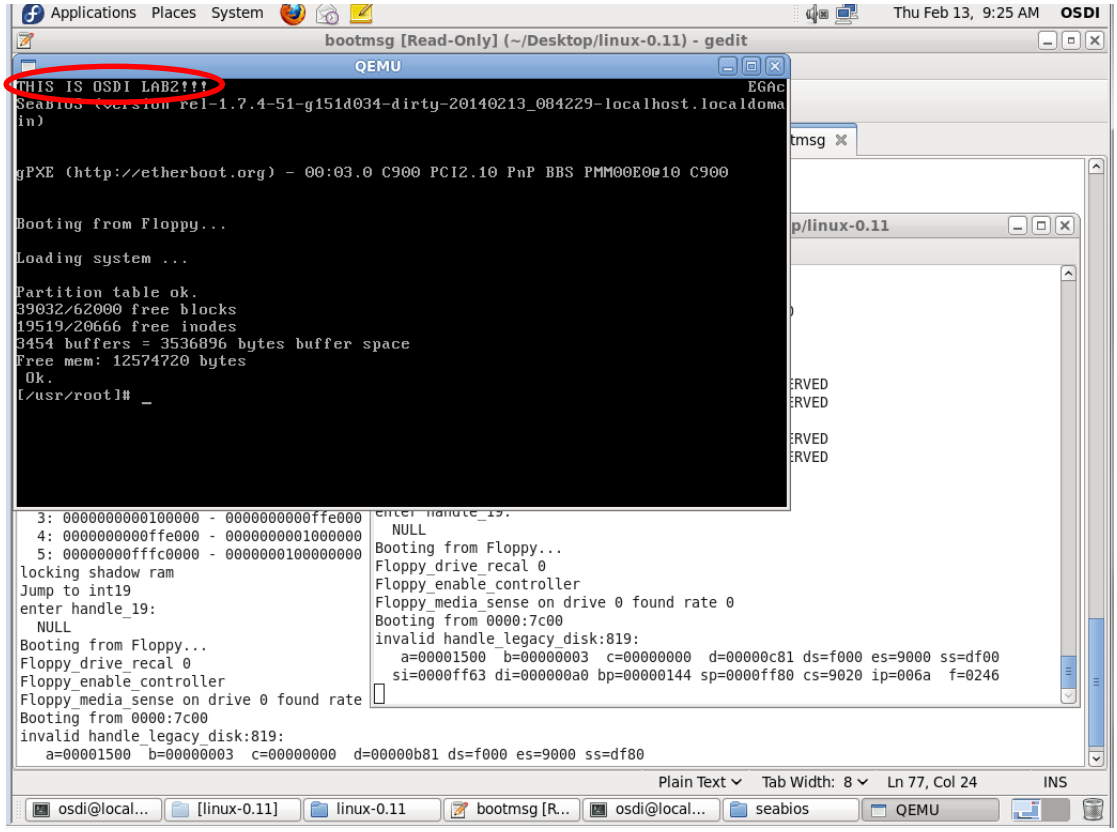


Next open the src/bootsplash.c and add the print message code in enable_vga_console function, then do make again.

```
void
enable_vga_console(void)
{
    dprintf(1, "Turning on vga text mode console\n");
    struct bregs br;

    /* Enable VGA text mode */
    memset(&br, 0, sizeof(br));
    br.ax = 0x0003;
    call16_int10(&br);

    printf("THIS IS OSDI LAB2!!!\n");
    // Write to screen.
    printf("SeaBIOS (version %s)\n", VERSION);
    display_uuid();
}
```

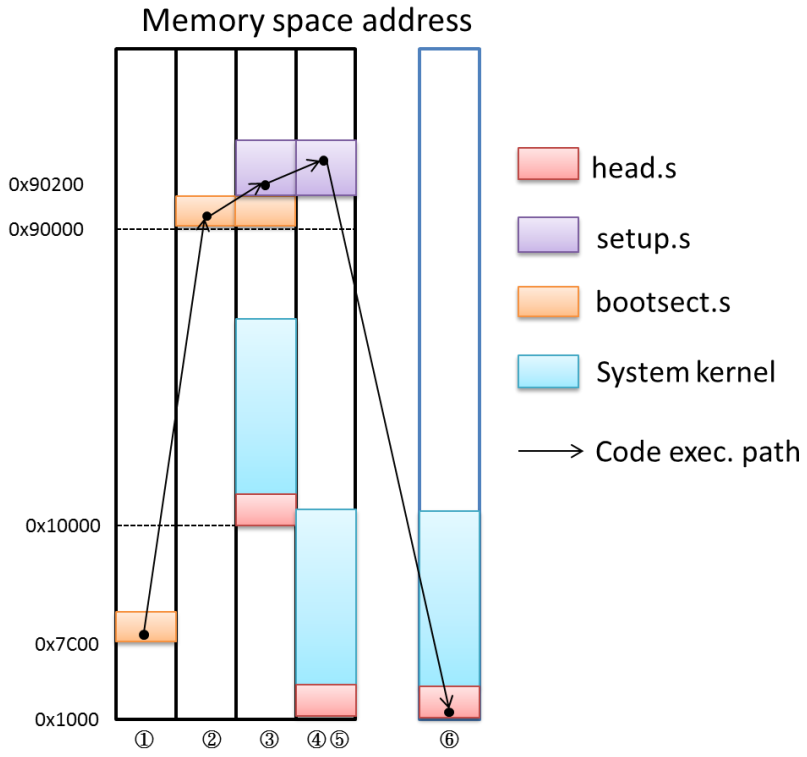
Next you can use following command to use customize BIOS rom for QEMU.

```
$qemu -m 16M -boot a -fda linux0.11/Image -hda osdi_lab1.img -bios seabios/out/bios.bin -serial stdio
```



2. Lab2.2 - Kernel Booting

In linux 0.11 is use two phase boot sequence, first is ***bootsect*** called “boot sector” use for load kernel into memory second is ***head*** it use for boot kernel and jump to main function.



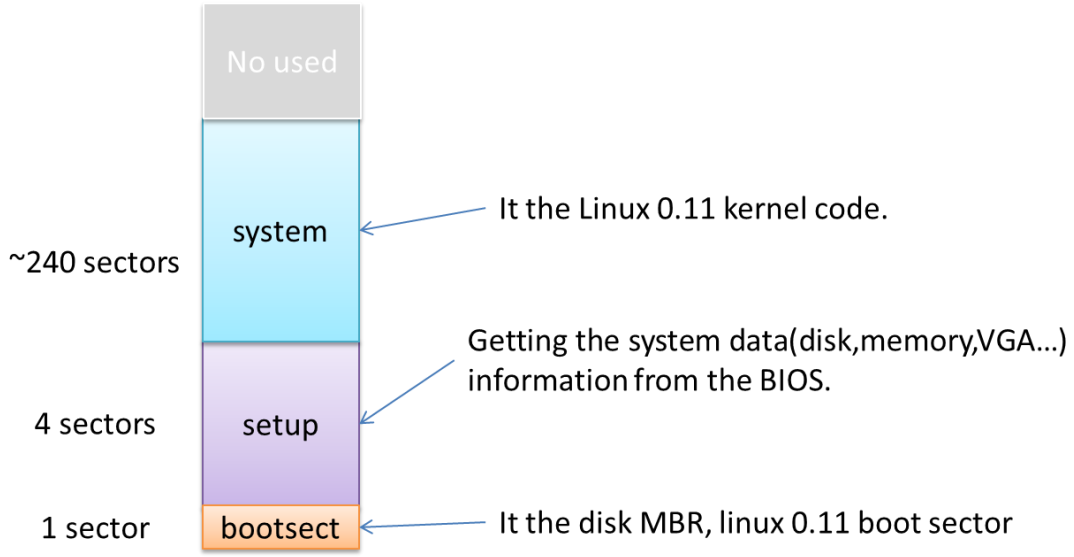
1. BIOS load the bootsect from disk MBR to 0x7c00 memory address
2. boosect copy itself to 0x90000 memory address and jump to 0x90000.
3. boosect load setup from disk to 0x90200 memory address.
4. Get some system peripheral device parameters (video, root disk, keyboard,…,etc.) and jump to 0x90200.
5. Switch system into protected mode move kernel from 0x10000(64K) to 0x01000
6. Jump to 0x1000 and execute head.s for kernel boot

2.1. Boot the hello world program

In this experiment you need modify the *boot/bootsect.s* boot loader program and make file to load a hello word program and execute it.

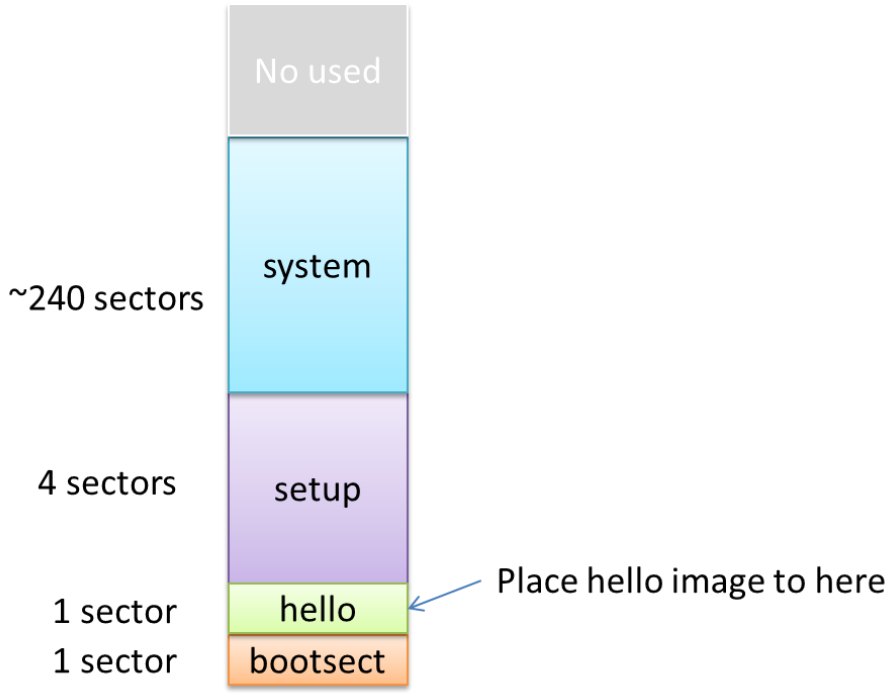
2.1.1. Image structure

By default linux 0.11 is use a 1.44MB floppy disk image to store bootsect and linux 0.11 kernel. The original image placement is shows on below diagram.



Note: Each sector has 512 bytes.

For disk space considering, this lab we will place the hello program to floppy second sector and shift back linux 0.11 setup and system one sector.



2.1.2. Modify the tools/build.sh

Add below command to write the hello binary image at floppy ***second sector*** and modify other image's sector placement

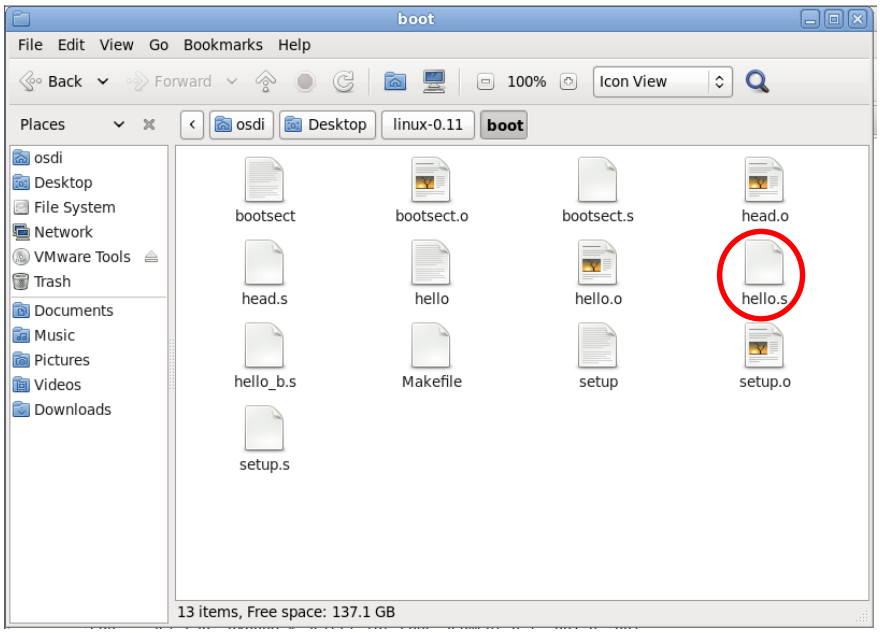
```
bootsect=$1
setup=$2
system=$3
IMAGE=$4
hello_img=$5
root_dev=$6
…
[ ! -f "$hello_img" ] && echo "there is no hello binary file there" && exit -1
dd if=$ hello_img seek=1 bs=512 count=1 of=$IMAGE 2>&1 >/dev/null
# Write setup(4 * 512bytes, four sectors) to stdout
[ ! -f "$setup" ] && echo "there is no setup binary file there" && exit -1
dd if=$setup seek=2 bs=512 count=4 of=$IMAGE 2>&1 >/dev/null


# Write system(< SYS_SIZE) to stdout
[ ! -f "$system" ] && echo "there is no system binary file there" && exit -1
```

```
system_size=`wc -c $system |cut -d" " -f1`

[ $system_size -gt $SYS_SIZE ] && echo "the system binary is too big" && exit -1

dd if=$system seek=6 bs=512 count=$((2887-1-4)) of=$IMAGE 2>&1 >/dev/null

…
```

2.1.3. Modify the Makefile

There you need write a makefile to build the hello world program.

TA already prepare a hello.s code for you, please refer the boot/Makefile see how the setup.s to build make binary image file then use it to build hello.s.



Open boot/Makefile and add a hello target

```
hello: hello.s
   @$(AS) -o hello.o hello.s -g
   @$(LD) $(LDFLAGS) -o hello hello.o
   @objcopy -R .pdr -R .comment -R.note -S -O binary hello

head.o: head.s
   @$(AS) -o head.o head.s

clean:
   @rm -f bootsect bootsect.o setup setup.o head.o hello hello.o
```

Next edit the main make file(linux0.11/Makefile).

Modify ‘Image’ target to below script and add a new target *boot/hello*

```
Image: boot/bootsect boot/setup tools/system boot/hello
   @cp -f tools/system system.tmp
   @strip system.tmp
   @objcopy -O binary -R .note -R .comment system.tmp tools/kernel
   @tools/build.sh boot/bootsect boot/setup tools/kernel Image boot/hello
$(ROOT_DEV)
```

```
	@rm system.tmp
	@rm tools/kernel -f
	@sync
boot/hello: boot/hello.s
	@make hello -C boot
```

2.1.4. Modify the boot/bootsect.s

Use “int $0x13” to load the hello image to 0x1000 memory address.

Open the boot/bootsect.s add red part code to the program and implement the blue part to load the hello image from floppy.

```
go:	mov	%cs, %ax
	mov	%ax, %ds
	mov	%ax, %es
# put stack at 0x9ff00.
	mov	%ax, %ss
	mov	$0xFF00, %sp		# arbitrary value >>512
# you can implement the load hello image code at here
load_hello:
…
##########
	.equ sel_cs0, 0x0008 # select for code segment 0 (  001:0 :00)
	ljmp $ sel_cs0,%0 #Jump to hello
#will not execute here
load_setup:
	mov	$0x0000, %dx		# drive 0, head 0
	mov	$0x0003, %cx		# setup now change to sector 3, track 0
	mov	$0x0200, %bx		# address = 512, in INITSEG
	.equ    AX, 0x0200+SETUPLEN
	mov     $AX, %ax  # service 2, nr of sectors
	int	$0x13
…
sread: .word 2+ SETUPLEN    # sectors read of current track
…
```

Note: you can refer the ***load_setup*** part to load **one sector** hello image to **0x1000** memory address.

Please note that the %es register setting, it will affect your buffer segment address.

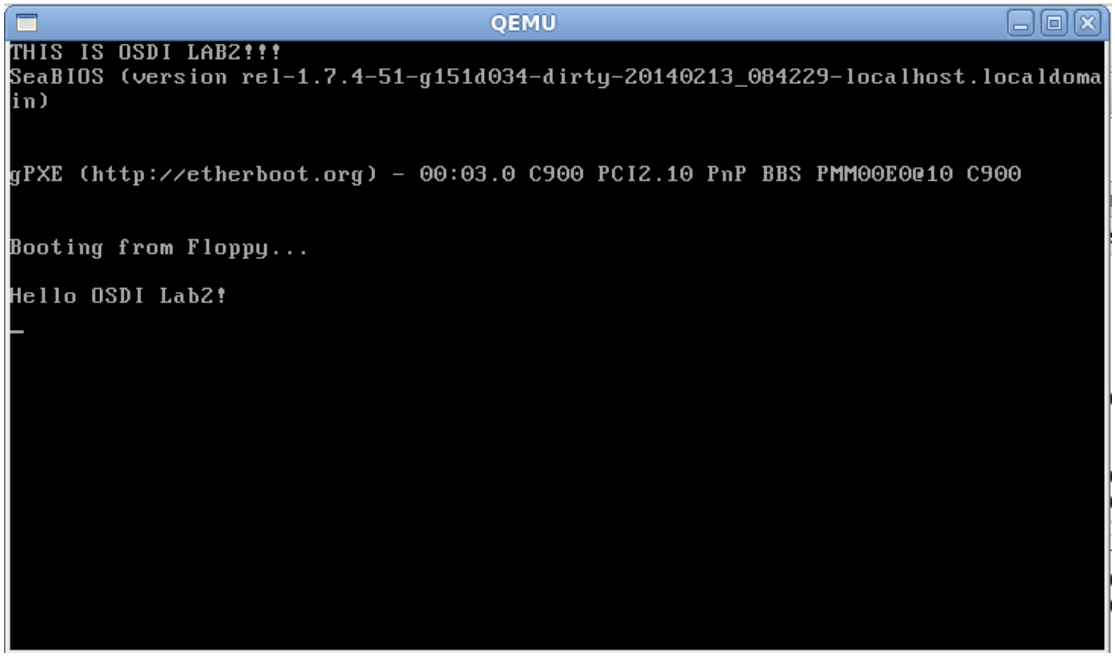
Reference: INT 13h, load disk sector to memory

http://www.ctyme.com/intr/rb-0607.htm

Floppy disk geometry:

http://media.uri.edu/cs/csc485/Disks/floppy_disk_geometry_TOC.pdf

2.1.5. Run the disk image on QEMU



3. Lab2.3 – Multi booting support

In this experiment you need implement simple keyboard character reader when user

Press ‘1’ to boot linux 0.11

Press ‘2’ to boot hello world

Hint: use the BIOS “int $0x16” service to read keystroke, when press ‘1’ jump to ***load_setup***, press ‘2’ jump to ***load_hello***, otherwise read keystroke again.

http://www.ctyme.com/intr/rb-1754.htm

4. DEMO and SUMMIT

We’ll demo the add BIOS message and multi booting parts. After demo, please commit you the lab2 patch file with filename <your id>_lab2.patch to SVN server.

Note: Before diff, please ***“make clean”*** your project first.

5. Appendix: Assembly quick review

5.1. Syntax

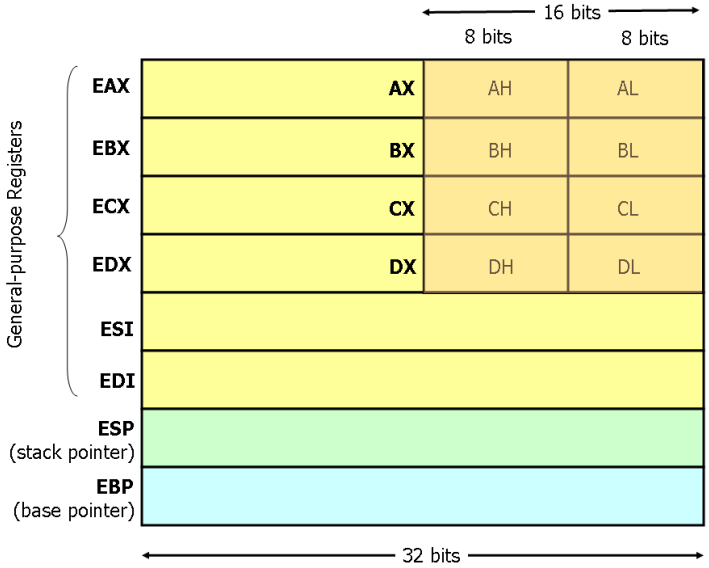
In linux assembly use AT&T syntax, below table shows the main different between Intel syntax and AT&T syntax.

Intel	AT&T	AT&T syntax Note
mov ax, 0x10	mov $0x10, %ax	• Rource value place on left hand side and destination register or memory address place on right hand side. • Register prefix with "%" • Value prefix with "$"
[basepointer + indexpointer*indexscale + immed32]	immed32(basepointer,indexpointer,indexscale)	Memory reference

Reference: http://nano-chicken.blogspot.tw/2010/12/inline-assembly.html

http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html

5.2. Registers



5.2.1. General purpose registers

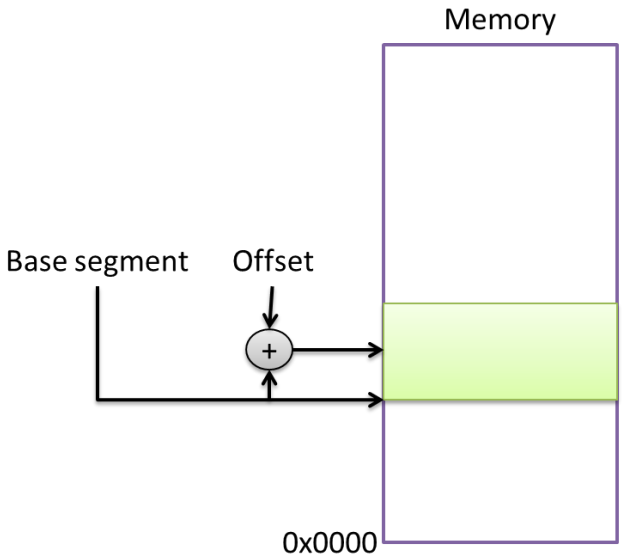
It can be used for moving data between register and memory or as a offset register for segmentation.

5.2.2. Segment registers

In x86 system use some segment register to control current code, data or stack segment placement.

Register	
cs	Code segment register, use “ip” (instruction pointer) as the offset register
ds	Data segment register, Use “bp”(base pointer) as the offset register
ss	Stack segment register, Use “sp” (stack pointer) as the offset register
es	Extra segment, could use a general purpose register as the offset register
fs, gs,	Induce from i386 microprocessor, use for support large segment.

Next lab we will explain the x86 memory model in protected mode, and how to access these register for task context switch support.



Reference:

http://en.wikipedia.org/wiki/X86_memory_segmentation

http://en.wikipedia.org/wiki/Intel_Memory_Model

5.3. Useful instructions

Instruction (Intel syntax)	Note:

mov <reg>,<reg>
mov <reg>,<mem>
mov <mem>,<reg>
mov <reg>,<const>
mov <mem>,<const>
	Moving data between register, memory or direct assign a constant value.

push <reg32> push <mem> push <con32>	Push data to stack, after this instruction the ESP/SP register will decrease by 4
pop <reg32> pop <mem>	Pop data to register or memory address, after this instruction the ESP/SP register will increase by 4

cmp <reg>,<reg>
cmp <reg>,<mem>
cmp <mem>,<reg>
cmp <reg>,<con>
	Compare two data, equivalent to the sub instruction

je <label> (jump when equal) jne <label> (jump when not equal) jz <label> (jump when last result was zero) jg <label> (jump when greater than) jge <label> (jump when greater than or equal to) jl <label> (jump when less than) jle <label> (jump when less than or equal to)	Conditional jump, usually use after cmp instruction
jmp <lable>	Jump PC to current code segment “lable” offset.
ljmp ,<offset>	Jump PC to :<offset> address, after this instruction will update the cs register to

Reference: http://www.cs.virginia.edu/~evans/cs216/guides/x86.html