JOS 实验一实验记录
作者：卓达城
指导老师：邵志远
单位：华中科技大学集群网络与服务计算实验室

说明：由于在 linux 上打中文太麻烦，虽然本人英文不好，不过还是用英文写，部分地方中文注释。

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^PART1^^
首先要做的是安装开发环境。
下载 VMware，Ubuntu 并安装。
然后下载 bochs 的源代码，编译安装，编译 bochs 需要安装几个依赖包。
依赖包包括：build-essential，xorg-dev，pkg-config，gtk 2.0
编译 JOS。
编译完之后，修改 bochs 的配置文件（具体两处），使 JOS 在 bochs 中启动。
具体步骤如下：
First we need to install the development environment.
1.install vmware 7.1 in windows then create the virtual machine for linux(ubuntu)

2.install ubuntu 10.4 in the virtual machine

3.set the chinese education network update source in ubuntu 10.4
（设置教育网的 linux 更新源，不然以现在的网速更新实在令人受不了）
How to set it? google!

4.install build-essential:
using the ubuntu software center to install the "build-essential" .You can type build-essential in the software center search bar,then install the build-essential. Because of my computer is so slowly to open the software center,so i use apt-get to install it.
the command is : sudo apt-get install build-essential

5.install xorg-dev
the command is :sudo apt-get install xorg-dev

6.install pkg-config
the command is :sudo apt-get install pkg-config

7.install gtk 2.0
the command is :sudo apt-get install libgtk2.0-dev

8.
Download the bochs from our teacher's website.Then extract it to your favourite folder.

9.
Compile the bochs
Go to the folder that you place the bochs then
Type command: ./configure --prefix=/usr --enable-disasm --enable-debugger
I will install bochs in usr.if install in other folder,you may have to set the enviromnent.
Type command: make
wait.....
Type command: sudo make install

10.
Compile the JOS
Download the lab1.tar.gz from our teacher's website
Extract them.
If you want to use gmake type command:ln -s /usr/bin/make /usr/bin/gmake
If you use make then you have no need to do the step above
Go to the folder lab1
type command: make
or type : gmake

11.
Edit the bochs config file .bochsrc
This file is in the folder lab1,but it is hidden.You must checked the "Show Hidden files"(文件是隐藏的，要显示隐藏文件才能看到）

Open it and find the string:
ata0-master: type=disk, mode=flat, path="./obj/kern/bochs.img", cylinders=100, heads=10, spt=10
Repalce it with:
ata0-master: type=disk, mode=flat, path="*******/obj/kern/bochs.img", cylinders=100, heads=10, spt=10
*******is the path of the obj folder'path.
Example:
/home/zdc/labs/lab1/lab1/obj/kern/bochs.img

Find the string:
romimage: file=$BXSHARE/BIOS-bochs-latest, address=0xf0000
Replace it with:
romimage: file=$BXSHARE/BIOS-bochs-latest

12
Run bochs
Then type 2 to set the path of the .bochsrc

If success type 6
Then we run the JOS successfully.

GoodLuck!!!

Second we finish the Exercise
Exercise 1
Learn the x86 assember Language
Learn the AT&T assember Language

Exercise 2
Familiar with bochs and bochs debugger.
The most useful commands are s,c,u/n address,q,b address

Exercise 3
Understand what is the bios doing.
Bios sets the idt first,initializes the key devices and then jump to 0000:7c00
启动的过程：
CPU 加电后从地址 0xfffffff0 开始运行，这个地址指向 BIOS 的 ROM 部分。
BIOS 先设置中断表，然后检测和初始化关键设备，然后把硬盘的第一个块加载到 0x00007c00 运行（这里就是我们 JOS 的 boot)。

^^^^^^^^^^^^^^^^^^^^^^^^^PART2^^^
第二部分主要是要我们熟悉 bochs 的调试工作。
Exercise 4
Set the breakpoint in 0000:7c00 and trace it.
We can use s perform the command step and step.
Or we can use u/n address to see n instructions start from the address
The debugger command simples:
b 0x0000:0x7c00
s
u/10 0x00:0x7c00

Trace bootmain()   line number：100 in boot.S
Through read the boot.asm,We can know that function bootmain will be called in 0x0000:0x7c45(0008:00007c45).So we set the breakpoint in 0x0000:0x7c40,then type s.Or set the breakpoint in 0x0000:0x7c45 directly.

Trace readseg() 反汇编 boot/main.o command objdump -S main.o -M intel > main.asm
Through tracing the instructions perform in bootmain(),we find the function readseg() is called at 0x0000:0x7d33(0008:00007d33).
Start at 7ce3:   55   push   %ebp

end at 0x7d21

readsect()
This function is called at [0x00007d0f] 0008:00007d0f,the instruction is:call 7c81 <readsect>
Start at "7c81: 55 push %ebp"
This function end at [0x00007ce2] 0008:00007ce2 (unk. ctxt): ret
After this function,the esp point to 0x7d14
Then perform serveral instructions until "7d16: 39 fb
cmp %edi,%ebx"
This instruction judge the condiction to decide whether continue to jump to 0x7d06,there are some instructions to set the parameters for readsect from 7d06 to 7d0f

At exactly what point does the processor transition from executing 16-bit code to executing 32-bit code?

```
movl %cr0, %eax
orl $CR0_PE_ON, %eax
movl %eax, %cr0
```

[0x00007c2a] 0000:7c2a (unk. ctxt): mov cr0, eax
this instruction start the protect module

What is the last instruction of the boot loader executed, and what is the first instruction of the kernel it just loaded?
The boot loader's last instruction is:[0x00007d84] 0008:00007d84 (unk. ctxt): call eax
And the first instruction of kernel is:[0x0010000c] 0008:0010000c (unk. ctxt): mov word ptr ds:0x472, 0x1234

How does the boot loader decide how many sectors it must read in order to fetch the entire kernel from disk? Where does it find this information?
The function readseg(ph->p_va, ph->p_memsz, ph->p_offset);'s second parameter tell the boot loader how many sectors shuold be load.
The SECTSIZE is 512 define in the macro.

Exercise 5
Familiar with C language

You can display a full list of the names, sizes, and link addresses of all the sections in the kernel executable by typing:
objdump -h ********/lab1/obj/kernel/kernel
not the command:
i386-jos-elf-objdump -h obj/kern/kernel

You can see the entry point by type the command:

objdump -f ********/lab1/obj/kernel/kernel
not the command:
i386-jos-elf-objdump -f obj/kern/kernel
We can see the start adress is 0xf010000c,not 0x0010000c,Why?

Exercise 6
The boot loader load the Elf to 0x10000(four zero),it initialize the structs of Elf.h.
Why are they different?
Because the boot loader move the data to 0c00100000(five zero,The structs of Elf.h include this value) using the code in main.c:

```
	ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
	eph = ph + ELFHDR->e_phnum;
	for (; ph < eph; ph++)
		readseg(ph->p_va, ph->p_memsz, ph->p_offset);
```

There is
0x00100000 <bogus+ 0>: 0x1badb002 0x00000003
0xe4524ffb 0x7205c766
0x00100010 <bogus+ 16>: 0x34000004 0x15010f12
0x0010f018 0x000010b8
at the second breakpoint.
They are executable instructions for kernel.
But why 0xf0000000 == 0x00000000
Just type help in our JOS,we will know the ture.
_start f010000c (virt) 0010000c (phys) (The load address and the link address) Why don't you tell me first!!!!!

Exercise 7
If i change the "0x7c00" to "0x8c00" in **boot/makefrag**,the bochs will run the bios programme to initialize the idt and the devices.
But when finished initialization,then jump to 0x7c00 continue to run the instructions,the bochs will restart the simulator.
Why?Because the boot loader is load at 0x8c00.There is no instructions at 0x7c00.

^^^^^^^^^^^^^^^^^^^^^^^^^^^PART3^^^

Exercise 8
Restart bochs,We should set the breakpoint in 0x0010000c(b 0x0010000c),and then type c.
Now we type s to see the JOS'kernel run step by step.
I am so luck that i see the instrction lgdt in [0x00100015] 0008:00100015.
So we can answer the problem of Exercise 8.
After [0x00100015] 0008:00100015 the new mapping takes effect.(segmentation,page?we should see the cr registors to judge it).

Open the file entry.S that in the folder kern.
We comment out(注释) the instruction "lgdt RELOC(mygdtdesc)" using #.
And then save it.
And then go to the folder lab1 type the comment:
make
Then we run the bochs and set the breakpoint at 0x00100000c
and type c,then type s,s,s,.....
We can see that
(0) [0x00100020] 0008:00100020 (unk. ctxt): jmp far 0008:f0100027 ;
ea270010f00800
<bochs:8> s
Next at t=274078546
bx_dbg_read_linear: physical memory read error (phy=0xf0100027, lin=0xf0100027)
<bochs:9> s
bx_dbg_read_linear: physical memory read error (phy=0xf0100027, lin=0xf0100027)
Why?
Becuase the new gdt isn't loaded to the gdtr.
当进入保护模式之后，段寄存器的值都变成索引值通过 gdt 转换的。
And then the 0x0008 index the segment descriptor isn't the the segment descriptor we really need.
So the virtual address to physical address change is error(it change to the old address),So the result above appear.

Now we restore the entry.S.
And make the JOS again.

To see the source of JOS'kernel,we need to install some software like sourceInsight,but in linux we use vim+cscope instead of it.
1Now we need to build the index in ctags like SourceInsight.(The teacher says that sourceinsight simulate the ctags,but now i use ctags to simulate the sourceInsight,ha ha!he he! :-))
Go the the root catalogue of lab1,and then type the command:ctags -R
Then the index is established.But we will use it later.

Exercise 9
The relationship among in printfmt.c,printf.c and console.c is:
printfmt -> vprintfmt -> putch -> cputchar -> cons-putc -> lpt-putc,cga-putc(set the char color)
(printfmt.c.....) (printf.c)(console.c.....)

The fill in the function "vprintfmt" is:
```
case 'o':
	// Replace this with your code.
	//putch('X', putdat);
```

```
        //putch('X', putdat);
        //putch('X', putdat);
        num = getuint(&ap, lflag);
        base =8;
        goto number;
        //break;
```

The specification of console.c
question 1:
lpt_putc() using in I/O prarallel port pragraming
cag_putc() set the char and char'colour that display on the screen. 15-8 bits are the attribute of the char,8-0 bits are the char's ascii code.
cons_putc() print the char on the screen use the funtion above.
cputchar() call the function cons_putc()

cprintf-> vcprintf -> vprintfmt -> putch -> cputchar -> cons-putc ->
lpt-putc,cga-putc(set the char color)
(printf.c.....) (printfmt.c.....) (printf.c) (console.c.....)

question 2:
In order to answer the question 2,I comment out this function.

```
void
cga_putc(int c)
{
        // if no attribute given, then use black on white
        if (!(c & ~0xFF))
                c |= 0x0700;
//whether are 31-8 bits zero?If they are set 8,9,10 bit 1,If not continue.

//whether are low 8 bits '\b','\n','\r','\t'?If they are,preform corresponding operation.
        switch (c & 0xff)
{                                                       //if not,display the character.
        case '\b':
                if (crt_pos > 0) {
                        crt_pos--;
                        crt_buf[crt_pos] = (c & ~0xff) | ' ';
                }
                break;
        case '\n':
                crt_pos += CRT_COLS;
                /* fallthru */
        case '\r':
                crt_pos -= (crt_pos % CRT_COLS);
                break;
```

```
		case '\t':
			cons_putc(' ');
			cons_putc(' ');
			cons_putc(' ');
			cons_putc(' ');
			cons_putc(' ');
			break;
		default:
			crt_buf[crt_pos++] = c;		/* write the character */
			break;
		}

		// What is the purpose of this?
		if (crt_pos >= CRT_SIZE) {
			int i;
//if the crt_pos(The charaters display on the Screen >= 25*80
			memmove(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE -
CRT_COLS) * sizeof(uint16_t)); //we move all the characters that display on the
Screen up one row
			for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
//And then set the last row ' '.
				crt_buf[i] = 0x0700 | ' ';
			crt_pos -= CRT_COLS;
		}

		/* move that little blinky thing */
		outb(addr_6845, 14);
		outb(addr_6845 + 1, crt_pos >> 8);
		outb(addr_6845, 15);
		outb(addr_6845 + 1, crt_pos);
}
```

question 3
First we add the code below to init.c

```
cprintf("6828 decimal is %o octal!\n", 6828); //in function i386_init(void).this line is
the source code.you can find it in the function i386_init(void).
//The code below is add by zhuodacheng
//data 2010-9-20
int x=1,y=3,z=4;
cprintf("x %d, y %x, z %d\n", x, y, z);
```

compile the kernel again!

We disassem the printf.o and the init.o
We use the command objdump -S filename -M intel > filename.asm
The function's address that i386_init(void) call.
You should set the breakpoint on the address,then you can debug it quickly.
0x0010015d : memset is called
0x00100162 : cons_init is called
0x00100176 : cprintf is called
0x0010017b : this address is the start point of the code we add.
0x0010019a : cprintf is called
0x001009bb : the first instruction of cprintf

We can see the instructions that push the parameters in to the stack before call the function cprintf.
First it will push the parameters to the stack from right to left.
And then call the function cprintf.
Fmt is point to the address of the first parameter in the stack.
ap is set in the function using va_start(ap, fmt)
ap is point to the second parameter in the stack.
ap = (sizeof(fmt) + 3) / 4 * 4 + fmt's address
And then the third parameter'address is:
ap = ap + sizeof(the type of the second parameter)
.....

The function called order
cprintf-> vcprintf -> vprintfmt -> (the function that get the char for printing -> va_arg),putch -> cputchar -> cons-putc -> lpt-putc,cga-putc(set the char color)
(printf.c.....) (printfmt.c.....)
(stdarg.h) (printf.c) (console.c.....)

The function cons-putc's parameter is an int
The 15-8 bits save the attribute of the char,the attribute inlude back colour and so on.
The 7-0 bits save the char that we will display
Others are not using.

The function va_arg's parameters are the cprintf's Nth(except first) parameter's address and type
This function return the Nth parameter's address and set the ap point to the (N+1)th parameter.
Before call this function the ap point to the Nth parameter of the cprintf.
After call it the ap point to the (N+1)th parameter.

The function vcprintf's parameters are:

Fmt is point to the address of the first parameter in the stack.
ap is point to the second parameter in the stack.
In the code that i add to the JOS,these two parameters' value are:
fmt = 0xf0101744
ap = 0xf0101744 + (17 + 3) / 4 *4

question 4
Now we add the code:

unsigned int i = 0x00646c72;
cprintf("H%x Wo%s", 57616, &i);
cprintf("\n");
below the code that we add above.
Recompile the kern.
Restart the bochs.
We can see Hellow World display on the screen.

Why?Because 57616 = ellow and 0x00646c72 = rld
What will heppen in the big-endian? World = Wodlr. Why? Because the order of the data store in the memory is different between the little-endian and the big-endian. In the little-endian,the data's order in the memory is from high to low like the register.But in big-endian,it is reverse.

question 5
Now we add the code:
cprintf("x=%d y=%d", 3);
cprintf("\n");
below the code that we add above.
Recompile the kern.
Restart the bochs.
And then we can see
x=3 y=-267325460
Display in the screen.
Why y=-267325460,because when the ap = the second parameter's address + 4(int's size),the data in it is unknow.So it print y=-267325460.

Challenge 2
Change the colour:
Add the code that below in first of the function cga_putc in console.c
c |= 0x7100;
recompile the JOS

Exercise 10
The kernel set it's stack in the entry.S,the instructions are:

```
movw    %ax,%ss
movl    $(bootstacktop),%esp
```

The kernel reserve the stack space throgh the instructions below:

```
        .p2align  PGSHIFT                   # force page alignment
        .globl          bootstack
bootstack:
        .space          KSTKSIZE
        .globl          bootstacktop
bootstacktop:
```

Though trace in the entry.S,we can find that the ss register is sat by 0x0010 (segment selector)
And the esp is sat by 0xf010f000 (the offset),so the top of stack's virtual address is 0x0010:0xf010f000,the physical address is gdt[0x0010]+offset.gdt[0x0010]is the 2th gd in the gdt.(0th gd NULL,1th gd code segment,2th gd data segment)
Now we type the command:
info gdt
in bochs,then we can see the gdt[2](equal to gdt[0x0010] above) is 0x10000000.
so the physical address of the esp(stack top pointer) is 0x10000000 + 0xf010f000 = 0x0010f000.
The linear address of the kernel is 0xf0100000,so the offset in the kernel is 0xf0100000++,and the data segment address is 0x10000000.Then the kernel is loaded to the physical address at 0x100000,When the linear address and the offset add,the address will equal to the physical address.
So amazing!!!

The stack pointer is pointing to the bootstacktop.bootstacktop is the end of the reserve space.(which end?)

Exercise 11
The address of the test_backtrace function is 0x001001f4
Before call the test_backtrace function,it will push one 32-bits parameter in to the stack.
And the call instruction will push next instruction's cs and eip into the stack.
But cs is 16 bits
Jump into the function.
And then push the ebp into stack
set ebp = esp
Then run the instruction of the function.
So before jump into the function test_backtrace,2 32 bits words will push into the stack
Before run the instruction of the function test_backtrace,3 32 bits words will push into the stack.

In this experiment
parameter : 5
next instruction's ip : 0x001001f9
ebp : 0xf010efc8

Exercise 12
Type the code that below into the function mon_backtrace in monitor.c:

```
unsigned int ebp;
ebp = read_ebp();
int a=0;
for(;ebp > 0;)
{
        cprintf("ebp = %x eip = %x arguments: %x %x %x\n",ebp,*((unsigned int
*)ebp+1),*((unsigned int *)ebp+2),*((unsigned int *)ebp+3),*((unsigned int
*)ebp+4));
        ebp = *( unsigned int *)ebp;
}
```

And then compile the JOS
Restart the bochs
And then we can see the result!