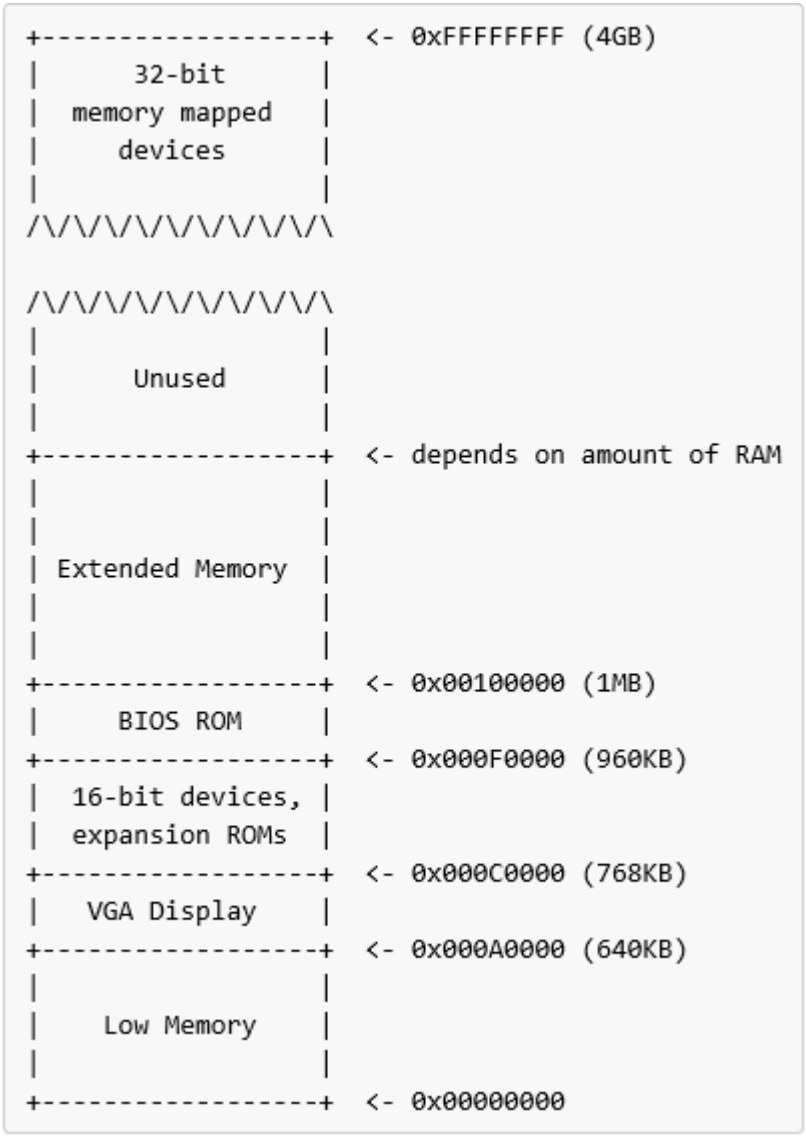
学号：5120809022
姓名：黄志强
内容：lab1 booting a pc

这个 lab 主要讲述的是计算机的启动过程，主要包括读取执行 BIOS，加载 bootloader，加载 kernel，完成之后等待 command 的输入，解析和相应操作的执行。

PART 1：BIOS



在这一部分，计算机固定加载 BIOS 在 0xf0000 至 0x100000 之间，计算机这个时候处于 real mode 状态，寻址模式是物理寻址，寻址方式 address=16*base+offset，20bit 地址模式，只能使用 1M 空间，计算机启动时，默认设置固定为 cs=0xf000 ip=0xfff0.所以计算机一启动就会将 PC 指到 0xf000*16+0xfff0=0xffff0 这个位置，由于这个是比较 high 的位置，在这里会有一条指令：ljmp $0xf000,$0xe05b，跳到 0xfe05b 这个位置，进入执行状态。整个 BIOS 主要就是检查调整一些设备，比如 VGA 等。当 BIOS 执行完成之后，cpu 扫描 disk，获取 bootloader，这个 bootloader 在 jos 中是 disk 的第一个 sector，第二个是 kernel，将第一个 sector 加载进入内存，并进行执行。

PART 2：Boot Loader

加载 bootloader 时，cs=0，ip=7c00，pc 将指向 0x7c00 这个位置，因此加载 bootloader 的位置也设置成 0x7c00，保证跳转指令的正确性。Pc 在 0x7c00 这个位置开始执行，当执行到：

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

将 PE 设置为 1，系统将从 real mode 进入保护模式，这个时候的寻址方式就变成了 address=base+offset，offset 将在指令中给出，而 base 就通过 cs 在 GDT 表中寻找段的 describer，这样 cs 依旧是 16bit，与 real mode 一致，至于 GDT 表，在执行 PE=1 之前就有指令：lgdt　　gdtdesc，加载 GDT 表，让我们来看看 GDT 表格的加载：

```
gdt:
  SEG_NULL                    # null seg
  SEG(STA_X|STA_R, 0x0, 0xffffffff)    # code seg
  SEG(STA_W, 0x0, 0xffffffff)                # data seg

gdtdesc:
  .word    0x17                                    # sizeof(gdt) - 1
  .long    gdt                                     # address gdt
```

GDT 加载了 3 个段，第一个默认格式，第二个 code 段基地址为 0x0，范围为 4G，可执行和可读权限，第三个 data 段，段基地址为 0x0，范围为 4G，可写权限。也正是这个 GDT 的加载，将变成 32bit 的寻址位，空间大小为 4G，不限于 1M 空间，当然，这个时候依旧是物理寻址，ljmp　　$PROT_MODE_CSEG, $protcseg，由于第一个 PROT_MODE_CSEG,对应的基址为 0，所以只是跳到$protcseg，这个$protcseg 只是简单的重新设置一些寄存器，毕竟模式发生了改变，然后调用了 call bootmain 函数。

Bootmain 主要是加载 kernel，kernel 是以 elf 格式进行存储的，所以按照 elf 每个段来加载 kernel 就比较合理，而不是以 page 来加载，kernel 位于第二个 sector 之后，每个段的加载是通过 sector 来加载的，加载完成之后就是调用 entry 入口，这个是 elf 文件注定的格式，位于 elfheader 中，这是一个好的技巧，至于加载的位置在 0x100000，就在 BIOS 上方，然后跳到相应位置执行指令，由于 link address 和 load address 不一致，所以在页表打开之前，用 RELOC（）来确定相应 link address 的 load address，通过这种方式将 cr3 赋值：

```
  movl   $(RELOC(entry_pgdir)), %eax
    movl      %eax, %cr3
```

完成之后，就打开页表，开启虚拟内存模式，从这里开始，内存寻址就是 32bit 的虚拟内存模式，在这里不得不提的一点就是：由于 kernel 的 link address 就是虚拟的，从 kernel.asm 就可以看出来，那么 entry 入口就是虚拟的，为了找到物理地址，我们采用了 RELOC（）函数，就是 addr-kernelbase，开启页表之后，为了保证虚拟地址映射的一致性，映射方式也就是简单的讲 VA-kernelbase，这样真个 kernel 0xf0100000 就映射到了 0x100000，大小是 4M，这是一个不得不惊叹的技巧。开启页表之后，就要进入内核运行，在运行之前，就要有 stack，jos 的方法是：

```
  movl   $0x0,%ebp                  # nuke frame pointer
    # Set the stack pointer
```

```
movl    $(bootstacktop),%esp
# now to C code
call  i386_init
```

%esp 被设置成为 top，%ebp 被设置成 0，又是一个技巧，这个技巧的使用在后面再说，最后进入 init 函数。

PART 3

进入内核，调用 init 函数，我们的 lab 就开始了，我们要写的函数就是在这里被测试调用的，让我们来看看这个代码：

```
cons_init();

cprintf("6828 decimal is %o octal!%n\n%n", 6828, &chnum1, &chnum2);
cprintf("pading space in the right to number 22: %-8d.\n", 22);
cprintf("chnum1: %d chnum2: %d\n", chnum1, chnum2);
cprintf("%n", NULL);
memset(ntest, 0xd, sizeof(ntest) - 1);
cprintf("%s%n", ntest, &chnum1);
cprintf("chnum1: %d\n", chnum1);
cprintf("show me the sign: %+d, %+d\n", 1024, -1024);
test_backtrace(5);

// Drop into the kernel monitor.
while (1)
    monitor(NULL);
```

中间部分就是我们要测试的代码，主要是 cprintf 和 test_backtrace。

Cprintf 的实现主要就是两步，第一步确定格式；第二步输出到控制台。第一步就是我们要完成的，这个就具体看代码，在这里不解释，第二步就是一个个 byte 输出到端口，这样显示。从这里我们来看，jos 用到的就是底层接口，没有任何系统调用或者库调用，也就是说，jos 是一个完整的 kernel，它不需要我们调用外部函数，要用到的全部都在内部实现。

Backtrace 的实现主要依赖 stack 的排布，关键性的几点：1，%ebp 存储的是原来%ebp 的值；2，参数位置在 8（%ebp），12（%ebp），。。。。。而且依次为第一个参数，第二个参数；3，4（%ebp）位置是 eip 的值；同时他的视线依赖于 elf 文件格式，即我们可执行文件中的 symtab 那一个段，我们可以根据这个 table 来获取 symbol 的相关信息，主要实现如下：

```
while(bp!=0)
    {
        debuginfo_eip(ip,&info);
        cprintf("eip %x ebp %x args %08x %08x %08x %08x %08x",ip,bp,args[0],args[1],
        args[2],args[3],args[4]);
        cprintf("%s:%d: %.*s+%d\n",info.eip_file,info.eip_line,info.eip_fn_namelen,
        info.eip_fn_name,ip-info.eip_fn_addr);
        bp=*(uint32_t*)bp;
        if(bp!=0)
```

```
        {
            ip=*((uint32_t*)bp+1);
            for(i=0;i<5;i++)
            {
                args[i]=*((uint32_t*)bp+i+2);
            }
        }
    }
```

在这里我们可以看到为什么进入 kernel 之前，要把 ebp 设置为 0，就是为了这个 while 循环，可见多么巧妙，另一个巧妙之处就是 ebp 位置存的就是旧的 ebp，所以不断向上可以获取轨迹。

Command 的实现主要就是进入 monitor（），循环，判断输入是否为已经存在的指令，如果是，就执行相应的代码：

```
1.for (i = 0; i < NCOMMANDS; i++) {
        if (strcmp(argv[0], commands[i].name) == 0)
            return commands[i].func(argc, argv, tf);
    }
2.struct Command {
    const char *name;
    const char *desc;
    // return -1 to force monitor to exit
    int (*func)(int argc, char** argv, struct Trapframe* tf);
};
static struct Command commands[] = {
    { "help", "Display this list of commands", mon_help },
    { "kerninfo", "Display information about the kernel", mon_kerninfo },
    { "backtrace","stack backtrace",mon_backtrace },
    { "time help","Display this list of commands",time_help },
    { "time kerninfo","Display information about the kernel",time_kerninfo },
    { "time backtrace","stack backtrace",time_backtrace }
};
```

非常巧妙地代码，这样根据输入字符串就可以判断是否是 command，以及该执行什么样的函数，然后实现函数就可以了，具体实现看代码，还有一个函数比较巧妙：

```
static __inline__ unsigned long long rdtsc(void)
{
    unsigned long long x;
    __asm__ volatile (".byte 0x0f,0x31":"=A"(x));
    return x;
}
```

这是获取时间的，由于不能使用库函数，系统调用，所以底层实现，这是嵌入式汇编，比较经典。

总之，大致 lab1 就说到这里，如果还有不足的，请见谅。