JOS 设计文档（lab1范围内）
丁卓成 5120379064

一、概述

本文档虽名为设计文档，实为本人在完成lab1过程中通过对JOS代码的阅读、分析得出的关于其设计的综述。本文的目的在于联系网络资料，深入完整地理解JOS的架构以帮助我学习操作系统启动的过程、操作系统最基本服务的实现等基础知识。以下行文将按照计算机启动的顺序依次介绍JOS的各个部分，主要内容为JOS代码分析以及对相关知识点的知识整理。

二、BIOS

BIOS运行于pre-boot阶段，不属于boot loader或OS的一部分，JOS不包含其代码，本次Lab中由QEMU模拟器负责提供，以下对Lab中提供的信息加以补充。

早期的BIOS烧录在ROM中，无法修改，功能也比较有限，而在现代PC中则通常可以进行更新，并有nonvolatile BIOS memory（一般也称为CMOS因为其通常采用CMOS RAM）与之配套。CMOS的作用是存储BIOS的设置，例如加载boot loader时各设备的优先度、BIOS密码等，BIOS启动后会加载其中的数据并据此配置硬件。

计算机启动后，BIOS首先进行的工作是加电自检（Power-on self-test，POST），该阶段负责检查各个硬件是否正常工作。此外，在自检过程中，BIOS可以提供一个用于系统配置的界面（一般通过启动时按某个键可以进入并会将配置存储于CMOS中），并会设置中断表，为操作系统提供若干底层服务（称为BIOS Interrupt call，例如INT 0x10就是显示服务，可以通过这个中断完成在屏幕上显示文字等功能）。当POST阶段结束后，BIOS会按照用户的设置依次读取boot device并试图载入其第一个扇区至0x0000:0x7C00，接下来就进入boot loader阶段。

三、Boot Loader

通常，Boot Loader由两部分构成，一部分位于设备的第一个扇区，BIOS自检完毕后率先运行，用于载入第二部分的代码，而第二部分才是功能完整的Boot Loader（例如grub2），负责引导操作系统。在本Lab中将其简化为只有一个阶段，由第一个扇区内的代码直接载入并运行内核。Boot Loader的代码由boot/boot.S和boot/main.c构成。

boot.S负责切换至32位保护模式（但不开启页表），其所做工作的具体细节上课解释过，在此不再介绍。注意经此步骤，GDT已被设置好，位于0x7c00至0x7dff之间某处，此时栈由0x7c00向下增长。

随后进入C代码，执行main.c的bootmain函数，负责读取kernel并运行。其中readsect(void *dst, uint32_t offset)函数将磁盘上第offset+1个扇区读取到dst指向的内存位置。readseg(uint32_t pa, uint32_t count, uint32_t offset)函数将kernel中offset开始的count个字节载入物理地址pa处，该函数以扇区为单位并对齐到扇区复制数据，因此可能复制了比要求更多的数据。bootmain函数首先读取一个page大小的数据，以保证ELF头及程序头表被完整载入。随后，它依照程序头表中的描述依次读取每一个segment到内存。注意其for循环内的语句：

```
readseg(ph->p_pa, ph->p_memsz, ph->p_offset);
```

这一句使用的是程序头中的p_paddr项，此项通常不使用或被操作系统赋予其他含义，通常使用的是p_vaddr项即JOS头文件中声明的p_va项，表示此段应该载入到虚拟内存的何处。由于此时页式内存管理还未开启，所以采用ELF格式中预留给物理地址的这一项进行定位。此外，一个段保存于ELF文件中的长度为p_filesz，载入到内存后的长度为p_memsz，两者不一定相等，因为.bss节在文件中不占空间，但程序载入内存后将占用内存空间。注意到此处的代码并未保证这多出的.bss节的内容置零，事实上这项工作将由kernel自身完成。

此外，由于Boot Loader必须放置于第一个扇区，它不能以ELF格式的可执行文件储存。查看boot/Makefrag可知，生成boot文件时，首先进行链接产生boot.out。链接时将start作为入口点，将text节地址设为0x7c00（这代表虚拟地址，不过此处起到物理地址的作用），这保证了内存模型的一致性，在使用标签、调用函数时会使用正确的地址。接着使用objcopy将生成的boot.out的.text节复制出来形成一个单独的文件boot。最后，使用一个perl脚本sign.pl将文件补齐至一个扇区的大小（若前一步生成的文件已经过大则产生警告）。注意到最后一步中并不要求启动引导代码小于446字节，只需小于510字节即可，引导代码之后的内容全都补零，除最后两个字节设置为0x55，0xAA。也就是说，JOS的第一个扇区并不遵循MBR的格式（同样也不遵循GPT格式，因为GPT格式的第一个扇区为了兼容性保持了MBR的格式不变），这导致整个磁盘只有一个分区，是对现实的操作系统的一种简化。

四、Kernel

进入kernel后的第一段代码负责开启（临时的）页式内存管理，由汇编写成，此后kernel就运行在虚拟地址空间的高256M，其余代码均由C写成。值得注意的是inc目录下的头文件中有部分代码在本次lab中并未用到，要到之后的lab中继续完善kernel的功能时才会用到。

（一）、开启页保护模式

首先来看汇编代码entry.S，它设置了cr0寄存器以开启页保护模式，重新设置了栈并调用i386_init函数进入C代码。我们来看它设置的页表：

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

可以看到PDT基址为entry_pgdir的物理地址，该符号定义于entrypgdir.c中，定义如下：

```
pde_t entry_pgdir[NPDENTRIES] = {
    // Map VA's [0, 4MB) to PA's [0, 4MB)
    [0]
        = ((uintptr_t)entry_pgtable - KERNBASE) + PTE_P,
    // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
    [KERNBASE>>PDXSHIFT]
        = ((uintptr_t)entry_pgtable - KERNBASE) + PTE_P + PTE_W
};
```

显然，这是一个临时的页表，其作用只有将当前运行的地址空间提升至0xF0000000开始的4M空间，根据注释，真正的页表要等lab2才会设置完毕。在inc/memlayout.h中画出了页表设置完毕后的虚拟地址空间布局，可以看到entry.S中.data段未使用的符号vpt及vpd的含义（即巧妙设置页表自引用，实现在虚拟地址空间中的virtual page table）。此时设置的栈位于.data节，在虚拟地址的0xF0100000以上，根据memlayout.h，此后还会调整栈的位置，但不在本次lab中实现。

此外，设置页表的过程，涉及eip从0x100000附近跳转到0xF0100000附近，这就要求kernel文件的链接要采取一些措施保证开头几

句语句能在0x100000附近顺利运行，而此后所有语句都在高地址运行。查看链接脚本kernel.ld可知，脚本中指定了各个segment的内容，并规定其虚拟地址从0xF0100000开始而物理地址从0x100000开始，保证了载入ELF文件时读取p_paddr，将程序加载至0x100000的物理地址，而各个符号在调用时采用的地址则是0xF0100000以上的虚拟地址。为了让开头几句汇编正确执行，程序的入口点被设置为_start，其定义为RELOC(entry)，即虚拟地址下的入口点减去0xF0000000所得到的物理地址的入口点，随后加载页表时也利用了RELOC宏将虚拟地址转换成物理地址。此外，上一节提到.bss段还未置零，因此在链接脚本中提供了edata和end两个符号，位于.bss段的开头和结尾，以便在其开始运行后再做清零工作。

（二）、输入输出的实现

进入C代码主函数i386_init后，首先用memset函数清零bss段。随后，为了进行输入输出，调用了cons_init函数，也就是说，在开机后内核还未初始化时，是没有输入输出之类的基本服务的，需要我们在内核中自己去实现，下面分析输入输出的具体实现。

输入输出分为两部分，一部分位于kern/console.c，负责进行底层的硬件操作以实现cputchar和getchar函数（即输出及输入一个字符），另一部分位于lib/printfmt.c和kern/printf.c，用于实现格式化输出（注意本lab中并未实现格式化输入，只有一个readline函数于lib/readline.c中实现）。

1.单个字符输入输出的实现

该部分代码位于console.c，主要涉及的硬件有CGA/VGA、串口、并口以及键盘，这一部分硬件驱动是较难理解的，故列出代码详细解释，并提供一些背景知识。

1).cons_init

cons_init是初始化的过程，它调用了cga_init,kbd_init,serial_init三个函数，其中kbd_init函数为空。

serial_init函数如下：

```
static bool serial_exists;
static void serial_init(void)
{
    // Turn off the FIFO
    outb(COM1+COM_FCR, 0);

    // Set speed; requires DLAB latch
    outb(COM1+COM_LCR, COM_LCR_DLAB);
    outb(COM1+COM_DLL, (uint8_t) (115200 / 9600));
    outb(COM1+COM_DLM, 0);

    // 8 data bits, 1 stop bit, parity off; turn off DLAB latch
    outb(COM1+COM_LCR, COM_LCR_WLEN8 & ~COM_LCR_DLAB);

    // No modem controls
    outb(COM1+COM_MCR, 0);
    // Enable rcv interrupts
    outb(COM1+COM_IER, COM_IER_RDI);

    // Clear any preexisting overrun indications and interrupts
    // Serial port doesn't exist if COM_LSR returns 0xFF
    serial_exists = (inb(COM1+COM_LSR) != 0xFF);
    (void) inb(COM1+COM_IIR);
    (void) inb(COM1+COM_RX);

}
```

serial即指串口(serial port)，串口是一种已经被淘汰的借口标准，现在的电脑已基本不配备串口，但可以通过USB建立虚拟串口。串口通常用COM1,COM2等标识，COM指Communication port，是对PC上串口的通称。此段代码对COM1进行初始化，COM1可以通过端口0x3F8-0x3FF访问，该访问实际是通过读写UART的寄存器实现的，每个寄存器大小为8位，占据一个端口号（同一端口读写可对应不同寄存器）。UART(Universal Asynchronous Receiver/Transmitter)是一种用于将并行的信号转换成串行并发送或将串行的信号转换成并行并接收的集成电路元件，电脑的CPU必须通过UART才能与串口设备相互通信。通常串口数据的读取是采用中断实现的，COM1对应的中断号是IRQ4，但在本lab中中断已被屏蔽（注意boot.S开头的cli语句），因此采用了程序查询方式。此处的IRQ4是指在PIC（Programmable Interrupt Controller）中的中断号，实际中断号（即int n指令中的编号n，0-31由Intel定义不许乱用，32-255可由用户自定义）经由PIC映射后再发送给CPU，该映射是可编程的。旧式电脑的PIC采用8259系列芯片，有8个中断引脚，一般用两个PIC级联共16个IRQ，每个IRQ的作用都是约定俗成的，例如此处COM1的中断号是IRQ4。新式的电脑则采用APIC（Advanced PIC），一个芯片有24个IRQ，但同时也集成了PIC，可在两者间切换，APIC前16个IRQ有部分和8259兼容，其中就包括了串口IRQ（此外还包括这里也用到了的键盘IRQ，为IRQ1）。

来看这一段初始化代码，第一句的作用是关闭FIFO队列（FCR即FIFO Control Register），这里的FIFO队列是指UART内部的缓冲区，读写各16个字节（或较新版本为64字节）。第二步三句的作用是切换至DLAB模式（该模式用于设置传输速率），将传输速率设置为9600bps。第三步是关闭DLAB模式，并设置数据传输时的具体编码格式，具体而言就是每隔8个数据位，插入一个stop位以区别是否停止发送，关闭奇偶校验（所谓串口，数据通过串口必须逐位收发，而不像并口通过多条线路并行发送）。第四步中no modem controls意即不与连接的设备进行握手，所有操作都是无连接的【此点无网上材料支持，存疑，望老师/助教指点】。第五步见注释即可。

cga_init函数如下：

```
static unsigned addr_6845;
static uint16_t *crt_buf;
static uint16_t crt_pos;
```

```
    static void cga_init(void)
    {
        volatile uint16_t *cp;
        uint16_t was;
        unsigned pos;

        cp = (uint16_t*) (KERNBASE + CGA_BUF);
        was = *cp;
        *cp = (uint16_t) 0xA55A;
        if (*cp != 0xA55A) {
            cp = (uint16_t*) (KERNBASE + MONO_BUF);
            addr_6845 = MONO_BASE;
        } else {
            *cp = was;
            addr_6845 = CGA_BASE;
        }

        /* Extract cursor location */
        outb(addr_6845, 14);
        pos = inb(addr_6845 + 1) << 8;
        outb(addr_6845, 15);
        pos |= inb(addr_6845 + 1);

        crt_buf = (uint16_t*) cp;
        crt_pos = pos;
    }
```

CGA（Color Graphics Adapter）是1981年IBM发布的显卡，同时也是PC上的第一块显卡。它具备16KB显存，最多支持640x200分辨率和16色，具备文字和图形模式。文字模式即只能显示文字的模式，无法操作单个像素，只能以字符为单位操作，CGA在文字模式下支持40x25或80x25个字符。同年，IBM还发布了MDA（Monochrome Display Adapter），Monochrome即黑白显示器之意，该显卡只具备4K现存和文字模式，不过分辨率更高，达到了720x350，同样支持80x25个字符，但每个字符由9x14像素构成（CGA的则为8x8）。遗憾的是，这两种历史上最早的显示标准互不兼容，它们映射到的内存地址以及IO端口都不同，因此尽管后来的产品都坚持向前兼容性，在开机后仍要面对两种可能。如上述代码说是，其中MDA的内存区域从MONO_BUF即0xB0000到0xB7FFF，端口号0x3B0-0x3BF（MONO_BASE为0x3B4），CGA的内存区域从CGA_BUF即0xB8000到0xBFFFF，端口号0x3D0-0x3DF（CGA_BASE为0x3D4）。实际上，MDA的端口号从0x3B0到0x3B7都是可用的，其中偶数号功能于0x3B4相同，奇数号功能与0x3B5相同，官方要求使用0x3B4与0x3B5，前一个端口用于选择一个MDA的寄存器出现在后一个端口的位置上，此外它还有0x3B8用于控制模式，0x3BA用于读取状态。CGA的端口号是类似的，分别对应于MDA的端口号的功能。

上述初始化代码中addr_6845即端口号，采用这个名字是因为当时MDA和CGA都是基于摩托罗拉MC 6845芯片制作，其中的控制器就是这块芯片。上述代码首先随便挑选了一个数0xA55A，检验CGA的帧缓冲区是否可写，以此来判断目前处于MDA还是CGA兼容模式，随后获取了目前光标的位置。

2).cons_getc

我们具备一个输入缓冲区cons，每次调用cons_getc时，从输入设备获取一个或多个字符（如果多个设备同时取到字符），填入缓冲区，并最终从缓冲区取出一个字符。也就是说，我们是采取查询的形式获取字符，每次调用cons_getc才从硬件获取字符，而不是通过中断的方式。具体取出一个字符的操作，通过一个函数指针传入cons_intr函数。其中串口（若存在）为serial_proc_data，如下：

```
    static int serial_proc_data(void)
    {
        if (!(inb(COM1+COM_LSR) & COM_LSR_DATA))
            return -1;
        return inb(COM1+COM_RX);
    }
```

LSR指的是Line Status Register，RX则是Read Buffer Register，因此这段代码保证了只有在串口有数据的情况下才读入一个字符。

键盘为kbd_proc_data，如下：

```
    static int kbd_proc_data(void)
    {
        int c;
        uint8_t data;
        static uint32_t shift;

        if ((inb(KBSTATP) & KBS_DIB) == 0)
            return -1;

        data = inb(KBDATAP);

        if (data == 0xE0) {
            // E0 escape character
            shift |= E0ESC;
            return 0;
        } else if (data & 0x80) {
            // Key released
```

```
            data = (shift & E0ESC ? data : data & 0x7F);
            shift &= ~(shiftcode[data] | E0ESC);
            return 0;
        } else if (shift & E0ESC) {
            // Last character was an E0 escape; or with 0x80
            data |= 0x80;
            shift &= ~E0ESC;
        }

        shift |= shiftcode[data];
        shift ^= togglecode[data];

        c = charcode[shift & (CTL | SHIFT)][data];
        if (shift & CAPSLOCK) {
            if ('a' <= c && c <= 'z')
                c += 'A' - 'a';
            else if ('A' <= c && c <= 'Z')
                c += 'a' - 'A';
        }

        // Process special keys
        // Ctrl-Alt-Del: reboot
        if (!(~shift & (CTL | ALT)) && c == KEY_DEL) {
            cprintf("Rebooting!\n");
            outb(0x92, 0x3); // courtesy of Chris Frost
        }

        return c;
    }
```

关于键盘的知识更广为人知，许多人都了解PS/2接口的键盘与USB接口的键盘的区别，但实际上，为了保持与旧软件的兼容性，主板一般将USB键盘和鼠标模拟成PS/2设备，因此我们只需关注PS/2设备的操作即可。与串口类似，PS/2设备也有一个对应的控制器集成于主板，相对更复杂一些，早期由独立的8042芯片负责，故也称之为8042控制器。它占据了两个端口0x60和0x64，其中0x60用于读写数据，0x64用于读取状态和发送命令。代码中KBSTATP即0x64，第一句的意图是检查是否buffer中有字符，如果没有就返回-1。PS/2控制器也有它的中断号，为IRQ1和IRQ12，有两个中断号的原因是PS/2控制器出键盘外还要管理鼠标，通常键盘为第一接口，使用IRQ1，鼠标为第二接口，使用IRQ12，使用终端时就无需像第一句一样再检查状态寄存器，直接获取字符即可。

关于键盘的输入，还有一点值得注意，那就是我们从键盘获得的数据，并不是用户输入了A，就从0x60端口读出了字符A，我们读取出来的称为Scan Code的编码。这种编码，在用户按下按键是产生一个输入，称为通码(Make Code)，在用户释放按键时又产生一个输入，称为断码(Break Code)，此外并不是一个操作对应一个字节，有时按下一个键会导致产生两字节甚至三字节Scan Code。Scan Code的标准有三套，分别称为Scan Code Set 1到3，按发布时间命名。Set 1是IBM的XT Keyboard使用的Scan Code，当时没有专门配套的控制器，而是采用一个多用途PPI(Intel 8048,Programmable Peripheral Interface)提供该功能，现在这种控制方式已经彻底淘汰甚至不提供兼容，但其Scan Code保留了下来。Set 2是IBM的AT Keyboard配套的Scan Code，当时已经配套有相应的PS/2控制器了（即Intel 8042，尽管只支持单一设备即一块键盘），因此此后的产品不断保持向前兼容性，至今这套Scan Code仍是使用最广泛的。当用户按下某个键时，键盘的硬件产生的Scan Code现在一般都是Set 2的（但可以通过out 0x64发命令修改其设定），但是为了兼容旧的Set 1，PS/2控制器会自动将其转换成Set 1（但可以通过out 0x64发命令禁止自动转换），此时从0x60端口读出的是Set 1的Scan Code（最初在8042上的实现是查表，这张表是用于将Set 2转为Set 1，因此如果硬件设置成产生Set 3同时允许转换，可能会导致结果有微妙的差别，在此不细究此问题）。Set 1的特点是通码为c，则断码为c+0x80，通码为e0 c，则断码为e0 c+0x80，我们看上述代码，正是符合了Set 1的特点。此外上述初始化代码还说明了一个有趣的现象，即平时按CAPSLOCK大小写会改变，按住Shift再按其他键按键功能会改变，这些功能并不是由键盘的硬件完成的，而是由操作系统负责实现，在内核中维护相应的状态，体现了职责单一化的思想。

3).cons_putc

该函数通过串口（如果有）发送字符，并显示在显示屏上，此外还会通过并口将字符传给打印机（如果有）打印出来。

串口通过函数serial_putc发送字符，如下：

```
static void serial_putc(int c)
{
    int i;

    for (i = 0;
         !(inb(COM1 + COM_LSR) & COM_LSR_TXRDY) && i < 12800;
         i++)
        delay();

    outb(COM1 + COM_TX, c);
}
```

其中delay函数等待的时间在微妙级，通过一个for循环设置了最长等待时间大约在毫秒级。

并口通过lpt_putc发送字符，如下：

```
static void lpt_putc(int c)
{
    int i;

    for (i = 0; !(inb(0x378+1) & 0x80) && i < 12800; i++)
```

```
            delay();
        outb(0x378+0, c);
        outb(0x378+2, 0x08|0x04|0x01);
        outb(0x378+2, 0x08);
    }
```

可以看到采用同样的技术以保证由一个最大等待时间。LPT(Logical Parallel Port)即并口端口，并口是早期为了连接打印机时提高传输速率而采取并行方式传输数据的接口（即通过8根数据线同时传输数据，串口只能1位1位发送），因此基本上成了专用于打印机的端口。现在的PC通常没有并口，稍早些的则通常具备两个LPT端口，LPT1为0x378到0x37F，使用IRQ7，LPT2为0x278到0x27F，使用IRQ5。早期存在位于0x3BC到0x3BF的端口，是同最早的MDA捆绑在一起的（即MDA自带并口支持打印功能），BIOS通常仍会检查该端口是否连接了打印机，但实际该端口从DOS时代就不再使用。并口的编程较简单，如上所示一共三个寄存器，+0位置是data register，+1位置是status register，+2位置是control register。

显示器通过cga_putc函数显示字符，其工作除了显示一个普通字符（如果到达最后一行后还需换行，还要做一个scroll操作，将屏幕向上卷屏以腾出空间输出下一行，此即lab中提到的一个问题，问注释提问的那一段代码的作用），还负责把特殊字符转化为普通的显示（例如\n\b等）。

2.格式化输出的实现

首先必须理解C语言对变长参数的实现。在C语言中，变长参数通过一个指针va_list获取，在一个变长参数的函数中，先声明一个va_list变量，再利用va_start对其初始化，通过va_arg获取参数，最后使用完毕后用va_end销毁该变量（置为NULL）。注意va_start第一个参数为va_list类型，第二个参数为变长参数的前一个参数，也就是说C标准规定不允许int foo(...);这样的函数声明，此外变长参数的具体个数由多少也是编译时刻无法确定的，需要在调用时通过某种方式指出，例如printf通过格式化字符串，execl通过最后一个参数设为空字符串等等。

了解了以上内容，就能明白变长参数列表中的参数到底如何取出，printf.c和printfmt.c中函数的组织也就一目了然：最外层的函数XXX为变长参数，它调用vXXX函数负责实现业务逻辑，vXXX函数自身做少量必要的工作并将核心的逻辑交由vprintfmt函数实现。vprintfmt函数需要接受一个函数指针来实现输出功能，这是因为printf与snprintf分别输出到屏幕和字符串，因此需做一个抽象，使用函数指针，该函数指针还附带计数功能以实现返回值为输出字符数量的功能。通过对vprintfmt中switch语句块的修改，即可实现本次Lab的Exercise 8-10（实现时printf的功能参照cplusplus.com上的文档所述）。

（三）、Monitor命令行交互

现阶段本Lab的kernel，在进行几个简单测试后，就进入了kernel monitor，也就是一个命令行模式，由kern/monitor.c实现。该功能的实现很简单，就是无限循环readline再runcmd，不支持复杂的语法，仅有几条命令在一个数组中查找并调用（通过函数指针）。其中backtrace和time命令需要自行补充完整代码，并需实现一个overflow，这一部分也与上一部分的格式化输出实现相关的练习相似，主要是查阅一些资料并看懂其代码的功能，本身代码量不大。代码中较为困难的部分，即backtrace中实现调试信息的获取，已经基本实现，是通过二分查找STABS格式的debug信息节，我们只需填写几行代码即可。实际上，看这次lab的makefile可知为使调试信息为STABS格式，gcc的选项被设置为-gstabs而不是通常的-g，通常的-g选项经我测试会编译成Dwarf格式的调试信息，调试信息分布于形如.debug.XXX的若干个节中，其储存的信息较STABS格式更为丰富。另外，overflow题目中有一个陷阱，即不能直接覆盖start_overflow函数的返回地址，这会导致do_overflow无法正确返回。正确的做法是进一步将do_overflow对应的返回地址覆盖，这样使得do_overflow能顺利返回overflow_me（但也有破坏其栈的风险，这可以通过返回地址及保存的旧ebp向低位移动4字节规避，ebp的值也要相应减四，不过本次lab恰好不会有问题因此我没有做此设计）。至此，所有这一部分要求完成的Exercise都已介绍完毕，更多系统内核相关的问题，就留待下次lab时再分析。