操作系统上机实验指南第一次作业说明

单位：	南开大学机器智能研究所 Institute of Machine Intelligence. Nankai University
日期：	2016/9/27

1. 概述

本次作业共完成两部分内容，分别为：“启动计算机”、“内存管理”。在完成本次作业作业之前，请参照附件中环境配置文档，对上机环境进行配置。

作业概述：本次作业练习共 5 题，问题共 4 题，作业共 5 题。其中练习题为帮助你理解整个系统用，做自己练习，不需要提交。问题以文档的形式，写在你们最终的实验报告中。
作业是一些代码工作，需要提交源码。

2. 启动计算机

本次作业的目的是通过详细观察一个操作系统的启动过程，来了解操作系统在启动过程中的步骤和流程，并且通过这个过程，熟悉实验中要使用的 QEMU 平台和 x86 汇编语言。本章节共分为三部分，第一部分着重熟悉 x86 汇编语言、QEMU 虚拟机和 pc 的启动；第二部分检查我们所提供内核的 bootloader(在 lab1/src/lab1_1/boot 目录下)，第三部分介绍 jos 内核（在 lab1/src/lab1_1/kernel 目录）

本章节所需要的资源已经打包在 lab1/resource 下。

2.1. 计算机的引导

第一个练习包括熟悉 x86 汇编语言、了解 pc 的启动过程以及使用 GDB/QEMU 进行调试。这一小节（1.1）没有任何需要你写的作业，但是请仔细完成下述流程，并保证能够理解。

2.1.1. x86 汇编语言

想必你在之前的课程已经学过 x86 汇编语言了，如果你有些内容记不清楚了或者之前错过学习该门课程，请参考 PC Assembly Language Book，这将帮助你熟悉 x86 语言。

练习 **1**

参考书籍使用的 NASM assembler, 而我们使用的是 GNU assembler，两者语法略有不同，可参考 Brennan's Guide to Inline Assembly 查阅相关不同之处。

2.1.2. Simulating the x86

我们使用一个程序来模拟一台真正的 pc，也就是说这些代码对真机同样有效。使用模拟器的好处是可以通过设置断点的方式来完整跟 踪计算机的过程,这一点在真是的计算机上是无法实现的。

我们使用 QEMU 模拟机来进行试验，首先进入 lab1_1，然后使用 make（or gmake on bsd ）system 编译 bootloader 和 kernel。在终端看起来像下图：

```
cd lab
make
+ as kern/entry.S
+ cc kern/init.c
+ cc kern/console.c
+ cc kern/monitor.c
+ cc kern/printf.c
+ cc lib/printfmt.c
+ cc lib/readline.c
+ cc lib/string.c
+ ld obj/kern/kernel
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 414 bytes (max 510)
+ mk obj/kern/kernel.img
```

完成编译后就可以利用刚刚编译成功的 obj/kern/kernel.img 启动 QEMU 了，该镜像包括了 bootloader（obj/boot/boot）和 kernel（obj/kernel）使用命令：

```
make qemu
```

该命令将使用刚刚编译的镜像启动 QEMU,并输出信息到 terminal。在 qemu window 可能出现下图类似的文字：

```
Booting from Hard Disk...
6828 decimal is XXX octal!
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

'Booting from Hard Disk...'之后的信息都有 jos 内核输出，K>是有内核中的 monitor 输出。The kernel monitor 只有两个命令 help 和 kerninfo。

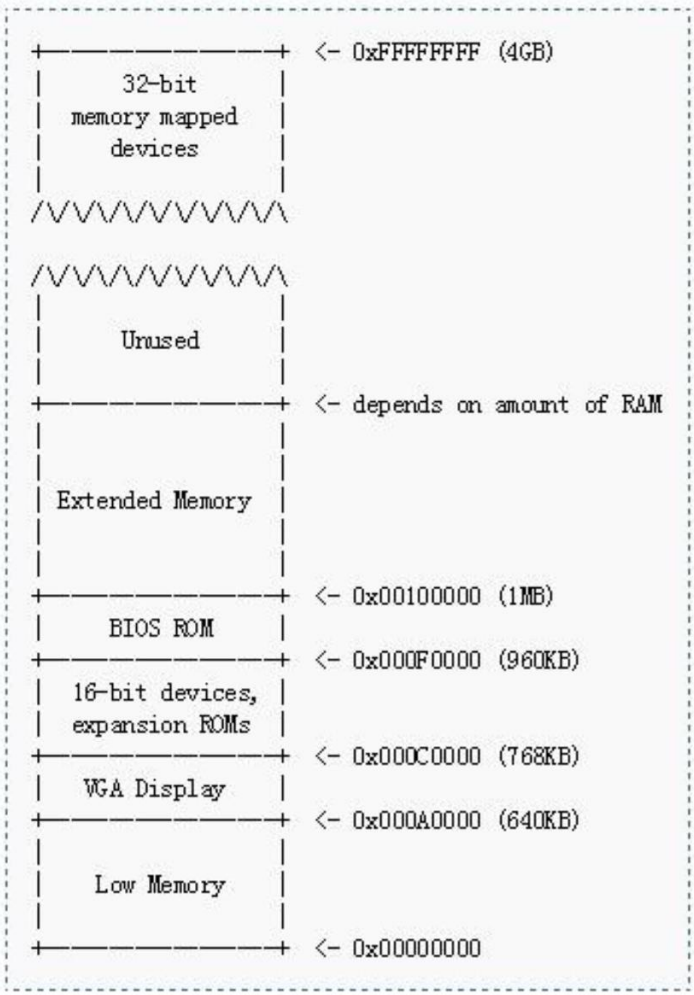
```
K> help
help - display this list of commands
kerninfo - display information about the kernel
K> kerninfo
Special kernel symbols:
  entry  f010000c (virt)  0010000c (phys)
  etext  f0101a75 (virt)  00101a75 (phys)
  edata  f0112300 (virt)  00112300 (phys)
  end    f0112960 (virt)  00112960 (phys)
Kernel executable memory footprint: 75KB
K>
```

help 命令不多解释，kerninfo 打印内核信息，因为 the kernel monitor“直接”运行"raw (virtual) hardware"因此，如果你将该镜像刻盘并运行在同一台真机上也会打印相同的内容（但是我们强烈不建议这样做！可能会导致你原来硬盘的内容都丢失!）

2.1.3. PC 的物理地址空间

下面介绍 pc 启动的详细信息。

首先介绍一台 pc 的物理内存大致分布图：



在最早的 16 位 8088 处理器上,最多只能使用 1MB 的物理内存,因此早期 PC 的物理地址空间范围为 0x00000000~0x000FFFFF 而不是 32 位地址的 0xFFFFFFFF。最底下的 640KB 内存被表示为”Low Memory”,也是早期 PC 所唯一能使用的 RAM,实际上,那时候的 PC 的物 理内存实际上根本没有那么大,一般是 16KB、32 KB 或者 64 KB。

物理地址范围 0x000A0000~0x000FFFFF 的 384KB 空间是被保留下来的,用于诸如 VGA 显示器缓冲区等硬件设备的。这一部分中最重要的就是作为基本输入输出系统(BIOS,Basic Input/Output System)的 0x000F0000~0x000FFFFF 的 64KB 区域。BIOS 负责处理系统基本的初始化任务,如激活显示器、检查内存等。结束这些初始化任务之后,BIOS 从软盘、硬盘、 光驱或者网络上载入操作系统,并将控制权转移给操作系统。

当 Intel 公司通过 80386 将可用物理地址空间扩大到 1MB 以上时,为保证对原有软硬件 的向后兼容性,计算机体系结构保留了原有对于最低 1MB 以内的地址的处理方式。现代 PC 上因此将 0x000A0000~0x00100000 成为常规内存,而将其余高地址部分成为扩展内存。同 时,

在物理地址空间的最高部分,位于所有物理 RAM 之上,有一部分也是被 BIOS 保留以用 于 32 位的 PCI 设备。

因为设计的局限性，JOS 只使用一台 pc 物理内存的第一个 256MB，所以在实验中我们都假设所有 pc 都只有一个 32bit 的物理空间。

2.1.4. BIOS

本小节使用 QEMU 的 debugging 特性了解 PC 引导的过程。打开两个 terminal，进入 lab1_1 目录下，一个输入 make qemu-gdb (or make qemu-nox-gdb)，这将启动 qemu 但是 qemu 会暂停等待 gdb 的 debug connection。另一个 terminal（与刚才的 make 在同一个目录下）下运行 “gdb”, 我们提供一个.gdbinit 文件用来启动 gdb 来进行 debug. 你会看到如下图的内容：

```
zhong@localhost:~/Desktop/lab$ gdb
GNU gdb (GDB) 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
+ target remote localhost:1234
The target architecture is assumed to be i8086
[f000:fff0] 0xffff0:    ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(gdb)
```

注意这一行：

```
[f000:fff0] 0xffff0:    ljmp   $0xf000,$0xe05b
```

这一行是 GDB 反汇编后执行的第一条指令，从中可以看出：

1） pc 从 CS = 0xf000 and IP = 0xfff0 处开始执行。

2） 第一句执行的执行是 JMP 操作，会跳到 CS = 0xf000 and IP = 0xe05b 处。

这是有 intel 设计的 8086 处理器的决定，因为 BIOS 在一台 PC 上总是处在物理内存 0x000f0000-0x000fffff，在实模式下将段地址转换为物理地址是通过如下步骤完成的: physical address = 16 * segment + offset。即：

```
16 * 0xf000 + 0xfff0   # in hex multiplication by 16 is
= 0xf0000 + 0xfff0     # easy—just append a 0.
= 0xffff0
```

因此启动后执行的第一条指令是在物理内存的（CS = 0xf000 and IP = 0xfff0）0x000FFFF0 处的“jmp e05b”，我们知道 BIOS 在内存中的上限是 0x00100000，于是在 0x000FFFF0 处执行第一条指令的话必然要向更“earlier”的地址跳转，这样才会有更多的 BIOS 指令可以执行。

为什么要以这种方式来启动呢？就是因为在刚开始的时候内存中没有任何其它的程序可以执行，于是将 CS 设置为 0xF000，将 IP 设置为 0xFFF0，物理地址为 0x000FFFF0，这样就保证了 BIOS 会在刚启动的时候得到控制权。

随着 BIOS 的继续执行,将设置 IDT(interrupt descriptor table)并初始化各种设备比如 VGA

显示输入,这也是在 QEMU 窗口所见到的" Starting SeaBIOS "的来源。 在初始化完 PCI 总线和所有重要的设备之后,BIOS 将会搜寻一个可启动设备,从中读入相应的 boot loader,最终将控制权移交。

2.2. Boot Loader

我们已经知道 BIOS 在完成它的一系列初始化后便把控制权交给 Boot Loader 程序，在我们的 JOS 实验中，我们的 Boot Loader 程序会在编译成可执行代码后被放在模拟硬盘的第一个扇区.

硬盘由于传统的原因被默认分割成了一个个大小为 512 字节的扇区，而扇区则是硬盘最小的读写单位，即每次对硬盘的读写操作只能够对一个或者多个扇区进行并且操作地址必须是 512 字节对齐的。如果说操作系统是从磁盘启动的话，则磁盘的第一个扇区就被称作“启动扇区”，因为 Boot Loader 的可执行程序就存放在这个扇区。在 JOS 实验中，当 BIOS 找到启动的磁盘后，便将 512 字节的启动扇区的内容装载到物理内存的 0x7c00 到 0x7dff 的位置，紧接着再执行一个跳转指令将 CS 设置为 0x0000，IP 设置为 0x7c00，这样便将控制权交给了

Boot Loader 程序。

在 1.12 的图 1 中可以看到 lab1_1 中的程序最终编译链接成了两个可执行文件 Boot 和 Kernel，其中 Kernel 是即将被 Boot Loader 程序装入内存的内核程序，而 Boot 便是 Boot Loader 本身的可执行程序，“boot block is 414 bytes (max 510)”这句话表示存放在第一个扇区的 Boot Loader 可执行程序的大小不能超过 510 个字节，由于磁盘的一个扇区的大小为 512 字节，这样便保证了 Boot Loader 仅仅只占据磁盘的第一个扇区。

另外我们要说的是在 PC 发展到很后来的时候才能够从 CD-ROM 启动，而 PC 架构师也重新考虑了 PC 的启动过程。然而从 CD-ROM 启动的过程略微有点复杂。CD-ROM 的一个扇区的大小不是 512 字节而是 2048 字节，并且 BIOS 也能够从 CD-ROM 装载更大的 Boot Loader 程序到内存。在本实验中由于规定是从硬盘启动，所以我们暂且不考虑从 CD-ROM 启动的问题。

下面我们就详细的讲述一下 Boot Loader 程序。在本实验中，Boot Loader 的源程序是由一个叫做的 boot.S 的 AT&T 汇编程序与一个叫做 main.c 的 C 程序组成的。这两部分分别完成两个不同的功能。

1） 其中 boot.S 主要是将处理器从实模式转换到 32 位的保护模式，这是因为只有在保护模式中我们才能访问到物理内存高于 1MB 的空间(保护模式我们之前在 1.1 节中有详细的讲解)。

2） main.c 的主要作用是将内核的可执行代码从硬盘镜像中读入到内存中，具体的方式是运用 x86 专门的 I/O 指令，在这里我们只用了解它的原理，而对 I/O 指令本身我们不用做过多深入的了解。

阅读完 boot loader 的代码之后,可以查看 obj/boot/boot.asm,这是当前 boot loader 在编译结束之后的反汇编代码。可以通过查阅该文件以学习 boot loader 的完整过程,并可以通过这个文件来跟踪整个 boot loader 的执行过程。

练习 **2**

使用 GDB 的 debug 功能完成 boot loader 的追踪过程。b 命令设置断点，如 b *0x7c00 表示在 0x7c00 处设置断点；c 命令可执行到下一断点；si 命令是单步执行。具体可参考 lab tools guide。

问题 **1**

Set a breakpoint at address 0x7c00, which is where the boot sector will be loaded. Continue execution until that breakpoint. Trace through the code in boot/boot.S, using the source code and the disassembly file obj/boot/boot.asm to keep track of where you are. Also use the x/i command in GDB to disassemble sequences of instructions in the boot loader, and compare the original boot loader source code with both the disassembly in obj/boot/boot.asm and GDB.

Trace into bootmain() in boot/main.c, and then into readsect(). Identify the exact assembly instructions that correspond to each of the statements in readsect(). Trace through the rest of readsect() and back out into bootmain(), and identify the begin and end of the for loop that reads the remaining sectors of the kernel from the disk. Find out what code will run when the loop is finished, set a breakpoint there, and continue to that breakpoint. Then step through the remainder of the boot loader.

确保你能回答以下几个问题：

1） At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?

2） What is the last instruction of the boot loader executed, and what is the first instruction of the kernel it just loaded?

3） Where is the first instruction of the kernel?

4） 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?

2.2.1. 载入内核

我们知道 Boot Loader 除了将系统从实模式转换到保护模式之外还有一个重要的任务就是将内核的可执行程序加载到内存，在 JOS 操作系统实验中，这个可执行程序实际上就是一个 ELF 文件，所以要了解内核是如何装入内存的，我们首先需要了解一下 ELF 文件。

在这里，我们仅仅之需要简单的了解一下 ELF 文件的基本组成原理，以便之后能够很好的理解内核可执行文件以及其它的一些 ELF 文件加载到内存的过程。

当开始编译和链接一个 C 程序 时,首先编译器(Compiler)将转换每一个 C 源文件(‘.c’) 为 object 文件(‘.o’)，在 object 文件中包 含着符合硬件规范的汇编指令。链接器(Linker)再将所有编译过的 object 文件合并为一个二 进制文件，如 obj/kern/kernel，该文件就是一个 ELF 文件，即(Executable and Linkable Format)。

在本次实验中，你可以简单的认为 ELF 文件是包含载入信息(loading information)的头部和多个程序块(program sections)组成，每一个程序块都包含连续的一段 data 或是 code。Loader 将会按字节的复制 这些信息到内存中,它不做任何修改,只是执行。

ELF 文件的前部是一个固定长度的文件头(ELF header)，之后跟随多个可变长度的程序头

(Program header)，这些 header 的定义在 inc/elf.h 中，与本次实验有关的部分如下:

.text: The program's executable instructions.

.rodata: Read-only data, such as ASCII string constants produced by the C compiler. (We will not bother setting up the hardware to prohibit writing, however.)

.data: The data section holds the program's initialized data, such as global variables declared with initializers like int x = 5;.

可以通过如下命令还获得所有部分的名称、尺寸和链接地址

% objdump -h obj/kern/kernel

在 ELF 中还有一个是十分重要的,叫做 e_entry。这个地方保存着 程序的 entry point 的链接地址(the link address of the entry point in the program,link address 见 1.2.2)你可 以通过如下命令还获得 entry point：

% objdump -f obj/kern/kernel

2.2.2. 连接地址和载入地址

使用% objdump -h obj/kern/kernel 命令查看 elf 文件时，注意"VMA" (or link address) and the "LMA" (or load address) of the .text section。链接地址实际上是程序自己假设在内存中的存放的位置，即编译器编译时认定程序将会存放在以链接地址开始的连续内存空间；载入地址实际指的是程序真正存放的地址。两者一般情况是一样的，如

%objdump -h obj/boot/boot.out

可以看到 BIOS 加载 boot 的 link address 为 0x7c00，它的 load address 也是 0x7c00，这才保证了程序能够正常加载，如果修改 boot 的 link address(boot/Makefrag 第 28 行)重新编译 (make clean 后重新 make)，那么 link address 与 load address 不一致，boot 将无法启动。现在你可以理解 boot/main.c 的 ELF loader.它将每一个 section 读入到内存的 load address, 然后跳转到了 kernel 的 entry point。

```
bootmain(void)
{
        struct Proghdr *ph, *eph;

        // read 1st page off disk
        readseg((uint32_t) ELFHDR, SECTSIZE*8, 0);

        // is this a valid ELF?
        if (ELFHDR->e_magic != ELF_MAGIC)
                goto bad;

        // load each program segment (ignores ph flags)
        ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
        eph = ph + ELFHDR->e_phnum;
        for (; ph < eph; ph++)
                // p_pa is the load address of this segment (as well
                // as the physical address)
                readseg(ph->p_pa, ph->p_memsz, ph->p_offset);

        // call the entry point from the ELF header
        // note: does not return!
        ((void (*)(void)) (ELFHDR->e_entry))();

bad:
        outw(0x8A00, 0x8A00);
        outw(0x8A00, 0x8E00);
        while (1)
                /* do nothing */;
}
```

2.3. The Kernel

2.3.1. 使用 virtual memory

操作系统内核经常被链接和执行在一个较高的虚拟地址,比如 0xf0100000,已将内存的较低部分留给用户程序。但是由于实际的物理内存可能不存在这个地址,我们需要通过处理器的内存管理设备将虚拟地址的 0xf0100000 到物理地址的 0x00100000,前者是链接地址, 即内核假设的运行地址,后者则是内核实际的载入地址。这样一来,内核的虚拟地址可以足 够高以留给足够的空间给用户程序,同时也可将内核载入到 1MB 内存的顶端,即在 BIOS 的上部。 事实上,我们将映射整个底部的 256MB 物理地址空间,即 0x0000000~0x0fffffff,到虚拟地址空间的 0xf0000000~0xffffffff。这也是为什么 JOS 内核被限制在只能使用 256MB 物理 内存

这部分后续作业会有详细说明。

2.3.2. 控制台的格式化输出

现在,我们接触一个常用的函数 printf(),这虽然是 C 语言的函数,但是在操作系统内 核, 所有的 I/O 处理都必须由我们生成。

阅读 kern/printf.c、lib/printfmt.c 和 kern/console.c,弄明白它们之间的关系。

问题 **2**

1)Explain the interface between printf.c and console.c. Specifically, what function does console.c export? How is this function used by printf.c?

2)Explain the following from console.c:

```
1      if (crt_pos >= CRT_SIZE) {
2              int i;
3              memcpy(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS) * sizeof(uint16_t));
4              for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
5                      crt_buf[i] = 0x0700 | ' ';
6              crt_pos -= CRT_COLS;
7      }
```

作业 **1**

We have omitted a small fragment of code - the code necessary to print octal numbers using patterns of the form "%o". Find and fill in this code fragment.

Warning：以下内容可能会帮助到你宏定义

(1)用 va_list 可以定义一个 va_list 型的变量，这个变量是指向参数的指针。

2) 用 va_start 宏可以初始化一个 va_list 变量，这个宏有两个参数，第一个是 va_list 变量本身，第二个是可变的参数的前一个参数，是一个固定的参数。

3) 用 va_arg 宏可以返回可变的参数，这个宏也有两个参数，第一个是 va_list 变量，即指向参数的指针，第二个是我们要返回的参数的类型。 4) 最后我们还可以用 va_end 宏结束可变参数的获取

lpt-putchar 和 outb 函数跟设备操作有关，不必细化了解。

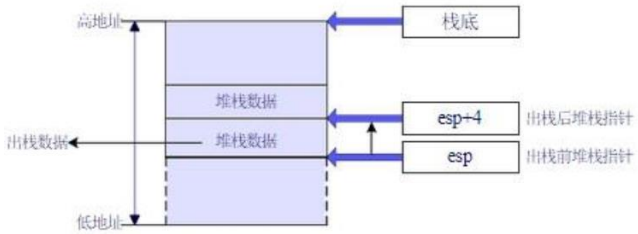
格式变量：

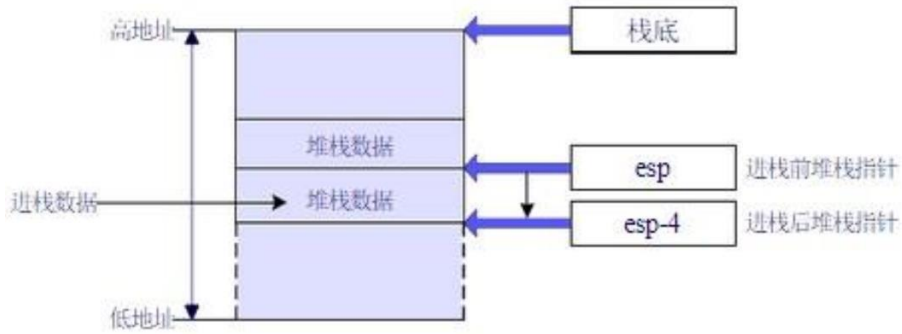
padc、width、precision、lflag、altflag。padc 代表的是填充字符，在初始化的时候 padc 变量会被初始化为空格符，而当程序在显示字符串的’%’字符后读到’ -’或者’0’的字符时便会将’ -’或者’0’赋值给 padc。width 代表的是打印的一个字符串或者一个数字在屏幕上所占的宽度，而 precision 则特指一个字符串在屏幕上应显示的长度，当 precision 大于字符串本身长度时相当于 precision 就等于字符串长度。于是在显示字符串的时候 precision 小于 width 的部分则由之前所说的填充字符 padc 来补充，如果 width 小于 precision 则字符串的宽度就等于 precision，而 precision 得默认值-1 代表显示长度为字符串本来的长度。当打印字符串的时候，padc=’-’代表着字符串需要左对齐，右边补空格，padc=’’代表字符串右对齐，而左边由空格补齐，padc=’0’代表字符串右对齐，左边由 0 补齐。在我们这个实验中当输出数字时会一律的右对齐，左边补 padc，数字显示长度为数字本身的长度。lfag 变量则是专门在输出数字的时候起作用，在我们这个实验中为了简单起见实际上是不支持输出浮点数的，于是 vprintfmt 函数只能够支持输出整形数，，输出整形数时，当 lflag=0 时，表示将参数当做 int 型的来输出，当 lflag=1 时，表示当做 long 型的来输出，而当 lflag=2 时

表示当做 long long 型的来输出。最后，altflag 变量表示当 altflag=1 时函数若输出乱码则用’?’ 代替。

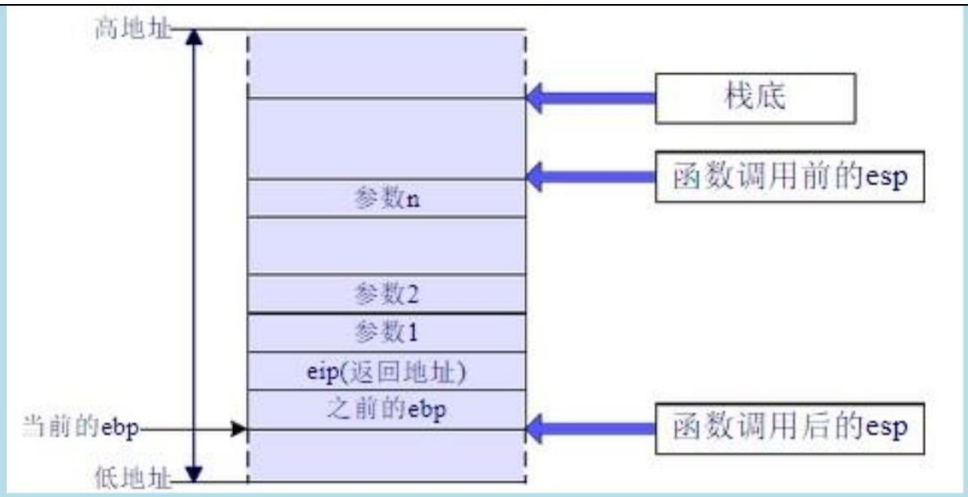
2.3.3. 堆栈

这一部分，我们将了解 x86 体系计算机的堆栈(stack)的设置,并编写一个函数 backtrace, 用以输出保存在堆栈中的 IP(Instruction Pointer)信息。在我们常规的理解下，栈顶应该是在内存的高地址处，而栈底是在低地址处，然而实际上在内存中却恰恰相反。栈底是固定的，栈顶是变化的，出栈和入栈如图：





首先了解这一过程相关的寄存器的概念 eip 存储当前执行指令的下一条指令在内存中的移地址，esp 存储指向栈顶的指针，而 ebp 则是存储指向当前函数需要使用的参数的指针。在程序中，如果需要调用一个函数，首先会将函数需要的参数进栈，然后将 eip 中的一字进栈，也就是下一条指令在内存中的位置，这样在函数调用结束后便可以通过堆栈中的 eip 值返回调用函数的程序。而在一进入调用函数的时候，第一件事便是将 ebp 进栈，然后将当前的 esp 的值赋给 ebp，这样此时 ebp 便指向了堆栈中存储 ebp、eip 和函数参数的地方，所以 ebp 通常都是指向当前函数所需要的参数，相当于每个函数都有自己的一个 ebp，所以当一个函数在内部调用另一个函数的时候，被调用函数执行时的 ebp 的值指向调用它的函数的 ebp 值存放的位置。



练习 **3**

To become familiar with the C calling conventions on the x86, find the address of the test_backtrace function in obj/kern/kernel.asm, set a breakpoint there, and examine what happens each time it gets called after the kernel starts. How many 32 -bit words d oes each recursive nesting level of test_backtrace push on the stack, and what are those words?

查看 test_backtrace 的 c 代码(kern/init.c 中),完成其中 mon_backtrace(),mon_backtrace 的原型已经在 kern/Monitor .c 中 。

```
// Test the stack backtrace function (lab 1 only)
void
test_backtrace(int x)
{
        cprintf("entering test_backtrace %d\n", x);
        if (x > 0)
                test_backtrace(x-1);
        else
                mon_backtrace(0, 0, 0);
        cprintf("leaving test_backtrace %d\n", x);
}
```

作业 **2**

You can do mon_backtrace()entirely in C ，You'll also have to hook this new function into the kernel monitor's command list so that it can be invoked interactively by the user.The backtrace function should display a listing of function call frames in the following format:

Waring:

read_ebp(较为底层的函数 ，返回值为当前的 ebp 寄存器的值)

display format

```
Stack backtrace:
  ebp f0109e58  eip f0100a62  args 00000001 f0109e80 f0109e98 f0100ed2 00000031
  ebp f0109ed8  eip f01000d6  args 00000000 00000000 f0100058 f0109f28 00000061
  ...
```

3. 内存管理

在本实验中，你将会编写你的操作系统内存管理的代码。内存管理有两个组成部分。第一部分是内核的物理内存分配器，它使得内核可以分配和释放内存。分配器将以 4096 字节为单元进行操作，这被称为页。你们的任务就是维护记录哪一个物理页被释放、哪一个页被分配的数据结构，以及有多少内存共享每个被分配的页。你页可以自己写分配和释放内存的方法。

第二部分是虚拟内存，在此我们将初步建立起内存管理单元(MMU, Memory Management Unit),其作用是将软件所使用的虚拟内存地址映射为硬件内存中的实际物理地址。你需要根据要求修改 JOS 的内核来设置虚拟内存。

此部分的源码，需要在完成第一部分“计算机引导”之后的代码目录中执行 checkout 和 merge 的操作，获取内存管理部分的代码，如下命令（注意：在执行以下命令之前，一定要备份好“计算机引导”部分的源码，作为该部分提交的作业！）：

```
zhong@localhost:~/Desktop/lab1_1$ git pull
    Already up-to-date.
    zhong@localhost:~/Desktop/lab1_1$ git checkout -b lab2 origin/lab2
    Branch lab2 set up to track remote branch refs/remotes/origin/lab2.
    Switched to a new branch "lab2"
    zhong@localhost:~/Desktop/lab1_1$ git merge lab1
    Merge made by recursive.
```

完成以上操作后，则获取到了内存管理部分的源码，请将文件夹的名字从原来的 lab1_1 改为 lab1_2，作为“内存管理”部分的源码。

3.1. 物理页管理

除了要设置处理器的硬件来将虚拟地址正确的转换为物理地址,操作系统也必须要记录每一部分的物理 RAM 是空闲还是被使用的。JOS 将会使用页粒度(page granularity)来管理物理内存,这样就可以使用 MMU 来映射和保护每一份被分配的内存。

你现在将会写这个物理页的分配器。它使用一个由 PageInfo object 结构组成的链表记录了哪些页是空闲的，每一个 PageInfo object 代表一个物理页。在写剩下的虚拟内存的实现之

前，你需要写这个物理页分配器，因为你的页表管理代码将会需要分配存储页表的物理内存。

作业 **3** 在文件 kern/pmap.c 中，你需要实现以下函数的代码（如下，按序给出）：

boot_alloc() mem_init() (only up to the call to check_page_free_list(1)) page_init() page_alloc() page_free()

check_page_free_list() 和 check_page_alloc()将测试你的物理页分配器。你需要引导 JOS 然后查看 check_page_alloc()的成功报告。加入你自己的 assert()来验证你的假设是否正确将会有所帮助。

3.2. 虚拟内存

在做其他事之前，请熟悉 x86 保护模式下内存管理架构：即分段模式（segmentation）和页转换（page translation）。

练习 **4**

阅读 Intel 80386 Reference Manual 的第 5 、6 章，阅读关于页转换和基于页的保护（5.2 和 6.4）。我们推荐你们略过关于分段模式的章节，因为 JOS 使用的虚拟内存和保护，段转换和基于段的保护不会在 x86 上被禁用，所以你们对他可以有一个基础的了解。

3.2.1. 虚拟地址、线性地址和物理地址

在 x86 术语中，一个虚拟地址包括一个段选择器和一个段中的 offset，一个线性地址是你在段转换之后、页转换之前获得的。一个物理地址是你在段、页转换后最终获得的、最终出现在硬件总线中。

虚拟地址的的 offset 是一个 C 指针。在 boot/boot.S 中，我们安装一个 Descriptor Table (GDT)，它通过将所有段的基地址设为 0，并限制在 0xffffffff 以内，可以高效的禁用段转换。

由此，选择器没有作用，线性地址经常等于虚拟地址的 offset。

在本实验中，我们扩充这个为：映射物理内存的前 256MB，从虚拟地址 0xf0000000 开始，同时映射到虚拟内存的其他区域。

练习 **5**

尽管 GDB 仅能通过虚拟地址访问 QEMU 的内存，但它在建立虚拟地址的时候检查物理地址方面很有效果。学习 QEMU 的 monitor command，特别是 xp 命令，它能够使你检查物理内存。在 Terminal 中按 Ctrl-a c 打开 QEMU monitor。

在 qemu 中使用 xp 命令，在 GDB 中使用 x 命令，来检查内存相应的物理地址、虚拟地址，确保你能看见相同的数据。

我们给出的补丁版本的 QEMU 提供了一个 Info pg 命令：它可以将当前页表简洁但详细的显示出来，包括所有映射范围、权限、flag。

从 CPU 执行的代码来看，一旦我们在保护模式下，没有办法直接使用一个线性的或者物理地址。所有的内存引用被解释为虚拟地址，并且被 MMU 转换，它意味着所有 C 中指针都是虚拟地址。

JOS 内核操作的地址经常是不可见的或者是作为整数值，并未解引用，例如在物理内存分配器中。为了帮助解释这部分的代码，JOS 源区别了两种情况：类型 uintptr_t 代表不可见的虚拟地址，uintptr_t 代表物理地址，两个类型都是相同的 32 位整数（uint32_t），因此编译器不会阻止你将一个类型指定为另一个类型！由于他们是整形并非指针，编译器将会报错如果你试图解引用他们。

JOS 内核可以通过将 uintptr_t 投影到一个指针类型上来对它解引用。相较而言，内核不能灵敏的解引用一个物理地址，由于 MMU 转换了所有的内存引用。如果你可以将一个 physaddr_t 投影到一个指针上并解引用它，你可能会可以加载和存储到结果地址（硬件会组织因为它是一个虚拟地址），但你将不会获得你想要的内存位置。

总结：

C type	Address type
T*	Virtual
uintptr_t	Virtual
physaddr_t	Physical

问题 **3** 假设以下内核代码是正确的，那么变量 x 将会是什么类型，uintptr_t 或者 physaddr_t?

```
mystery_t x;
char* value = return_a_pointer();
*value = 10; x =
(mystery_t) value;
```

JOS 内核有时会需要读取或者修改内存，其中它仅知道物理地址。例如，增加一个页表映射可能会需要分配物理内存来存储一个页目录，并初始化这个内存。然而，内核也像其他软件一样，不能忽视虚拟内存转换，由此不能直接加载和存储到物理内存。JOS 将所有物理内存从物理地址 0 在虚拟地址 0xf0000000 的重新映射的一个原因是为了帮助内核读写内存，其中它仅知道物理地址。为了转换一个物理地址到内核可以直接读写的虚拟地址，内核为了寻找在重新映射区域内相应的虚拟地址，必须增加 0xf0000000 到物理地址，做这个增加你会用到 KADDR（pa）。

3.2.2. 引用计数

在未来的时严重你会经常同时的在多个虚拟地址使用相同的物理页映射（或者在多环境的地址空间）。你将会在 PageInfo 结构中的 pp_ref 保存相应物理页的引用计数，当这个计数到了 0 时，这个页面就可以被释放了。实际上，这个计数应该等于在所有页表中这个物理页在 UTOP 之下出现的次数（在 UTOP 之上的是在内核引导的时候建立的，永远页不能被释放，所以对于这些不需要引用计数）。

当使用 page_alloc 时要注意，它返回的页常常又一个为 0 的计数引用，所以 pp_ref 应该在你对返回的页完成了一些事情之后立即递增（例如将它插入到一个页表中）。有时它被其他函数处理（如 page_insert），有时函数 page_alloc 必须直接做这件事。

3.2.3. 页表管理

现在你需要写一系列的页表管理方法：增加和删除线性到物理（linear-to physical）的映射，当需要时创建页表页。

作业 **4** 在文件 kern/pmap.c 文件中，你必须实现以下函数的代码。

pgdir_walk() boot_map_region() page_lookup() page_remove() page_insert() mem_init()调用的 check_page()，用于测试你的页表管理方法。

3.3. 内核地址空间

JOS 将处理器的 32 位线性地址空间分成了两部分。用户环境（进程）将会管理 lower part 的内容，内核维护 upper part 的管理。分隔线被 inc/memlayout.h 中的 ULIM 符号定义，保留内核的大约 256MB 的虚拟地址空间。

在本次实验中，参考 inc/memlayout.h 中的 JOS 内存布局图将会有所帮助。

3.3.1. 权限和故障隔离

由于内核和用户内存在每个环境的地址空间中都存在，我们在 x86 页表中需要使用权限位来使得用户代码仅能访问地址空间的用户部分。否则，用户代码中的 bug 可能会覆盖内核数据，引起崩溃；或者用户代码就可以偷取其他环境的私有数据。

用户环境将对 ULIM 之上的任何内存都没有权限，而内核可以读写这部分内存。对地址范围[UTOP，ULIM），内核和用户环境都有相同的权限：都只可以读不可以写。在 UTOP 之下的地址空间供用户环境使用，用户环境将会设置权限访问这部分内存

3.3.2. 初始化内核部分的线性地址空间

现在你将在 UTOP 之上建立地址空间：内核部分的地址空间。inc/memlayout.h 显示了你会用到的布局。你将会使用你写的函数建立适当的线性到物理的映射。

作业 **5** 在调用 check_page()之后，填写 mem_init()丢失的代码。

你的代码需要通过 check_kern_pgdir()和 check_installed_pgdir 的检验。

问题 **4**

1)在这一点上页目录中的哪些行已经被填写了？他们映射了什么地址，指向了哪？换言之，尽量填写以下表：

Entry	Base Virtual Address	Points to (logically):
1023	?	Page table for top 4MB of phys memory
1022	?	?
.	?	?
.	?	?
.	?	?
2	0x00800000	?
1	0x00400000	?
0	0x00000000	[see next question]

2)我们已经将内核和用户环境放在了相同的地址空间。为什么用户程序不能读或者写内核内存？什么样的具体机制保护内核内存？

3)这个操作系统最大能支持多大的物理内存？为什么？

4)管理内存有多大的空间开销，如果我们拥有最大的物理内存？这个空间开销如何减小？

本次实验完成。确保你通过了所有的 make grade 测试，别忘了写出问题的答案。

4. 作业要求

4.1. 代码部分

✧ 修改后的源码

4.2. 文档部分

需要在提交的作业中提供本次作业的文档，包括但不限于以下内容：

- ✧ 小组的组号，组中成员的姓名和学号；
- ✧ 小组中每一位成员在本次作业中的分工以及贡献比例；
- ✧ 本次作业的设计思路；
- ✧ 作业中的问题部分。

4.3. 提交方式

- ✧ 当面提交，提交地点：计控学院楼 427　宋佳慧　曹先 蒋建飞
- ✧ 提交要求：演示修改的源码，证明完成作业内容，讲解思路。

4.4. 提交时间

本次作业于 2016年 9 月 27 日发布， 请于 2016年 10 月 29 日当天提交，当天为周六，请班长将小组分为上下午进行作业的提交。过期提交者，将有可能在本次作业的评分上产生不利影响。

另外，有什么问题可以发送邮件至邮箱：nkshi_os@163.com ，邮箱将会定期查看并回复一些基本问题。