Lab 01

Update 2016-03-06: 讲义中提供的初始化中断控制器的代码里，`outb(IO_PIC1+1, 0x1)` 最好改成 `outb(IO_PIC1+1, 0x2)`，即设置 ICW4 的第 2 个位为 1，自动返回 EOI (End Of Interruption)；或者你可以了解如何手动返回 EOI。

由于本网页是静态页面，很可能被浏览器缓存，请猛击 Ctrl-F5 强制刷新页面……

实验提交

截止时间: 2016/03/15 23:59:59 (如无特殊原因，迟交的作业将损失 50%的成绩 (即使迟了 1 秒)，请大家合理分配时间)

请大家在提交的实验报告中注明你的邮箱，方便我们及时给你一些反馈信息。

学术诚信: 如果你确实无法完成实验，你可以选择不提交，作为学术诚信的奖励，你将会获得 10%的分数；但若发现抄袭现象，抄袭双方(或团体)在本次实验中得 0 分。

提交地址：http://cslabcms.nju.edu.cn/

提交格式: 你需要将整个工程打包上传，特别地，我们会清除中间结果重新编译，若编译不通过，你将损失相应的分数（请在报告中注明你实验所使用的 gcc 的版本，以便助教处理一些 gcc 版本带来的问题）. 我们会使用脚本进行批量解压缩. 压缩包的命名只能包含你的学号。另外为了防止编码问题，压缩包中的所有文件都不要包含中文.如果你需要多次提交，请先手动删除旧的提交记录(提交网站允许下载，删除自己的提交记录)，否则若脚本解压时出现多次提交相互覆盖的现象，后果自负. 我们只接受以下格式的压缩包：

- tar.gz
- tar.bz2
- zip

若提交的压缩包因格式原因无法被脚本识别，后果自负。

请你在实验截止前务必确认你提交的内容符合要求(格式、相关内容等)，你可以下载你提交的内容进行确认。如果由于你的原因给我们造成了不必要的麻烦，视情况而定，在本次实验中你将会被扣除一定的分数，最高可达 50% 。

git 版本控制：我们建议你使用 git 管理你的项目，如果你提交的实验中包含均匀合理的 git 记录，你将会获得 25% 的分数奖励（请注意，本实验的 Makefile 是由你自己准备的，你可以选择像 PA 中一样在每一次 make 后增加新的 git 记录作为备份，但是请注意，这样生成的 git log 一般是无意义的，所以不能作为加分项）。为此，

请你确认提交的压缩包中包含一个名为 .git 的文件夹；如果你提交的实验中没有 git 记录或者 git 记录不合理，那么我们将为你准备较为严苛的答辩内容。

实验报告要求：仅接受 pdf 格式的实验报告，不超过 3 页 A4 纸，字号不能小于五号（10.5 磅，或 3.7 毫米），尽可能表现出你实验过程的心得，你攻克的难题，你踩的不同寻常的坑。你可以贴图片贴代码，如果 3 页够用的话。

分数分布：

- 实验主体：80%，完成所有实验要求；
- 实验报告：20% 。

解释：

1. 每次实验最多获得满分；
2. git 的分数奖励是在实验主体基础上计算的，也就是说如果你实验主体满分且有均匀合理的 git 记录，那么你可以不用写实验报告；
3. git 记录是否“均匀合理”由助教判定；
4. 迟交扣除整个实验分数的 50% ；
5. 作弊扣除整个实验分数的 100% ；
6. 提交格式不合理扣除整个实验分数的一定比例；
7. 实验批改将用随机分配的方式进行；
8. 保留未解释细节的最终解释权。

关于游戏

在正式开始实验之前，我们要介绍一位贯穿整个实验始终的朋友：**游戏**。

操作系统**内核**本身不产生价值，它是一个纯粹的支撑软件，是由其管理的应用程序们给予了它存在的意义，作为参照，可以了解下 Linux kernel 和 GNU 以及 Android 的关系。所以，为了让我们的内核可堪一用，需要为它提供一些应用程序，即我们的游戏，同时我们也将用这个游戏体验从无操作系统到有操作系统的过渡。

这个游戏需要你们自行编写，它的最终形态是一个用 C 编写，运行在 i386 上的程序，能够响应按键，输出图形，有一定的交互逻辑。罗马不是一天建成的，所以不要为了写出一个酷炫的游戏而耽误太多时间，我们现在只需要一个足够简单的起点，毕竟实验刚开始是一片洪荒，连框架代码都没有，如果游戏代码太复杂，很可能很难跑起来。随着我们的内核逐步完善，你们的游戏的功能将越来越丰富，也越来越模块化。下面是我们推荐的游戏起点：

```
// file: game.c
int main(void)
{
  while (1) ;
  return 0;
}
```

嗯，虽然这看起来根本不是个游戏，但是不用担心，等你们真的能把上述代码在 QEMU 里跑起来时，才是放开手脚、解放想象力的时候。

注意，不要在游戏设计上花费太多心思，只需要有图形显示，以及键盘交互即可，最好有时间概念。一些简单的例子如：贪吃蛇，打砖块，拼图，推箱子。

下面我们开始实验，为了让你对实验的流程有明确的认识，能够准确地把握自己的进度，我们将实验划分成了数个小阶段。你不需要按照阶段提交，只需要一个阶段一个阶段的完成要求即可。

阶段 1 Boot Loader 和串口输出

阶段 1 的目标是编写一个 boot loader，将上述那个简单的根本不是游戏装载进内存并执行，然后，在游戏代码中完成串口的相关工作，使得你的游戏代码能在控制台里输出字符串。

你首先需要像实验准备讲义里那样，将 boot loader 程序制作成 512 字节的合法的引导扇区，然后在 boot loader 中完成系统的部分初始化工作，并将游戏程序装载到内存，最后，你要在游戏中完成另外一些初始化工作，实现相关代码，使得游戏能够输出字符串。

下面首先介绍你要在 boot loader 中需要做哪些系统初始化工作。

系统启动

计算机加电后, 一些寄存器会被设置初值, 计算机将运行在实模式(real-address mode)下, 其中 CS:IP 指向 BIOS 的第一条指令, 即首先取得控制权的是 BIOS。 BIOS 将检查各部分硬件是否正常工作, 然后按照 CMOS RAM 中设置的启动设备查找顺序, 来寻找可启动设备。可启动设备的特征是：第一个扇区的末尾是一个约定了的魔数 0x55 和 0xaa， BIOS 会据此判定一个设备是否可启动。因此 BIOS 会将各个设备的第一个扇区加载到内存的 0x7c00 处，然后常看它们的第 512 字节是否魔数。BIOS 确定第一个扇区是可启动的之后，便会执行 0x7c00 处的地址，执行你的程序了。

历史遗留：A20 地址线 http://blog.csdn.net/ruyanhai/article/details/7181842

历史遗留：实模式 https://en.wikipedia.org/wiki/Real_mode

段描述符和全局描述符表： 如果你对这两个概念比较陌生，请你复习 ics 课本中的相关内容，或者阅读 PA3 的讲义。

内联汇编： https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html

现在我们知道了系统启动时便会执行 0x7c00 处的代码，那么能不能在这里就开始运行由 C 代码编译而来的程序呢？答案是否定的，因为我们处于实模式下并且 A20 地址线未打开。

首先，在实模式下，地址线的宽度是 16 位，配合段寄存器只能访问 1M 的内容；并且我们只能执行 16 位的机器指令，而编译器编译 C 代码一般得到的应该是 32 位的机器码（如果你的机器是 64 位的，那么默认生成 64 位代码，注意给 `gcc` 加上 `-m32` 编译选项）。

如果我们设置了 CR0 的 `PE` 位从而进入了保护模式，可以执行 32 位代码了，但是如果 A20 地址线没有打开，则可以访问的内存只能是奇数 1M 段，即 1M, 3M, 5M... 也就是 00000-FFFFF, 200000-2FFFFF, 300000-3FFFFF...

为了将游戏运行起来，首先需要打破这些枷锁。那么我们是先开启保护模式还是先打开 A20 地址线呢？在开启保护模式前，可以方便地使用 BIOS 中断来开启 A20 地址线，所以我们建议先打开 A20 地址线。

打开 A20 地址线

这是 jos 中开启 A20 地址线的实现：

```
seta20.1:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.1

  movb    $0xd1,%al               # 0xd1 -> port 0x64
  outb    %al,$0x64

seta20.2:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.2

  movb    $0xdf,%al               # 0xdf -> port 0x60
  outb    %al,$0x60
```

将这段代码插入到你的 boot loader 中你认为合适的位置。

进入保护模式

要想进入保护模式我们首先得有一张全局描述符表，这张表得由你自己手动填写。虽然在 PA 中有相关的内容，但我们还是在这里再熟悉一遍。

定义段描述符

在这个实验中，我们建议大家使用 "flat mode"，在 i386 手册中有关于 "falt mode" 的介绍，简单的讲就是让 CPU 在保护模式下时使用的虚拟地址空间是 0~4G 的段，这样线性地址和物理地址在物理内存范围内都是一一对应的（开启分页之前）。而实现扁平模式非常简单：使 CS 指向基地址为 0，长度限制为 4G 的段描述符；使 DS, SS, ES, FS 和 GS 指向基地址为 0，长度限制为 4G 的段描述符。 但是二者不是完全相同的，因为前者是可执行但不可写的，而后者是可读可写的，所以你要先理解段描述符中各个字段的意义，为 2 种段描述符设置不同的属性。 要在 C 代码中定义段描述符，你可以使用位域结构体。而在汇编代码中你可以使用以下的宏来定义段描述符（来自 jos）：

```
#define SEG(type,base,lim)                                      \
    .word (((lim) >> 12) & 0xffff), ((base) & 0xffff);          \
    .byte (((base) >> 16) & 0xff), (0x90 | (type)),             \
        (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)
```

另外，mmu.h(来自 jos) 中定义了段描述符的各个字段的宏以及在 C 代码中可以使用的结构体，可以方便你编程。

开启保护模式

为了进入保护模式，我们在启动过程中需要做三件事：

1. 正如前文所述，全局描述符表是由段描述符构成的数组。我们需要告诉 cpu 我们定义的描述符的地址，通过 lgdt 指令实现；
2. 在此之前我们是没有全局描述符表的，现在要告诉 cpu 已经准备好了全局描述符，应该修改 cr0，让 cr0 中关于保护模式的字段有效；
3. 开启 cr0 中的 PE 字段之后，应该使用 ljmp 指令（同时修改 cs:eip）进入保护模式， ljmp 的 cs 即你的代码段描述符对应的段选择子，而 eip 则是一个 32 位代码的起始地址。

如何从汇编代码跳转到 C 函数？你首先要明白函数名不过是一个符号，它的值即它的第一条指令的首地址。以 main 函数为例，你首先在汇编代码中使用伪指令 `.globl main` 让编译器编译时看到这个外部符号，然后直接将这个符号名作为操作数使用 call 或者 ljmp 即可跳转。如果你使用 call 指令跳转到 C 代码，那么建议如下组织切换保护模式和跳转执行的代码：

```
  ...
  ljmp $<your-cs-value>, $next
.code32
next:
  ...
  call main
```

现在我们能使用较大的内存了，进入保护模式了，能跳转执行 C 程序了，我们可以运行我们的游戏了！话说，我们的游戏在哪呢？

加载游戏

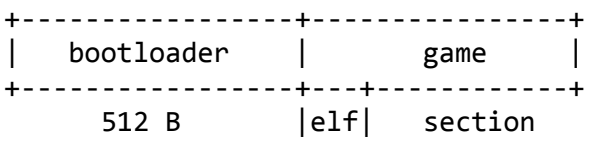
BIOS 最初只加载 512 字节到内存中，为了使我们的游戏能够运行，我们必须将它从磁盘加载到内存中来，所以，我们首先需要解决两个问题：

1. 明确两个程序（boot loader 和 game）在磁盘中的布局，这样我们才能找到它；
2. 读取磁盘的内容。

我们那个简单的游戏虽然寥寥几行代码，编译出来也不会超过 5 条指令，不过加上 ELF 信息和符号表，超过 512 字节还是绰绰有余的。

磁盘文件说明

在实现了 boot loader 和 game 后，分别对两个部分进行编译链接。然后将二进制文件拼接在一起成为一个完整的磁盘文件。你可以简单粗暴的使用 cat 工具完成这一工作，而我们提供的 Makefile 中则使用到了 dd 工具。你可以参照 Makefile 中的内容对此进行理解，也可以自行通过 man 手册进行查询。在拼接完成后，我们得到的磁盘文件最终的结构如图所示：



需要注意的是我们拼接的 game 部分是包含 elf 头的，而 boot loader 则是纯粹的机器指令序列。Boot loader 通过 elf 头的信息从磁盘文件中读取相应的代码到内存中。完成读磁盘的代码时候请不要游戏程序文件的内容是从第二个扇区开始的。

解决磁盘 IO

对磁盘的 I/O 是通过 in、 out 等指令实现的， in 和 out 是向上文提到的独立编址的地址空间输入或输出数据的指令。 对独立编址的空间进行 I/O 时需要使用到内联汇编，我们提供了将常见的汇编代码封装成 C 语言的函数的头文件 x86.h （来自 jos）。为了理解某个具体的指令的意义，你可能需要阅读 i386 手册。 另外，为了对磁盘进行 I/O 你还需要了解与磁盘相关的 I/O 端口。为了屏蔽具体的硬件细节，以下是来自 jos 的读取磁盘数据的实现：

```
void
waitdisk(void)
{
    // wait for disk reaady
    while ((inb(0x1F7) & 0xC0) != 0x40)
        // do nothing;
}
```

```
void
readsect(void *dst, uint32_t offset)
{
    // wait for disk to be ready
    waitdisk();

    outb(0x1F2, 1);     // count = 1
    outb(0x1F3, offset);    //address = offset | 0xe0000000
    outb(0x1F4, offset >> 8);
    outb(0x1F5, offset >> 16);
    outb(0x1F6, (offset >> 24) | 0xE0);
    outb(0x1F7, 0x20);  // cmd 0x20 - read sectors

    // wait for disk to be ready
    waitdisk();

    // read a sector
    insl(0x1F0, dst, SECTSIZE/4);
}
```

解析 ELF

利用 readsect 函数，你可以从内存中载入足够大的数据块，使其完整地包含了 elf header 。下一步就是对 elf header 进行解析，首先要熟悉 elf header 的结构，可以参考以下结构：

```
/* ELF32 二进制文件头 */
struct ELFHeader {
    unsigned int   magic;
    unsigned char  elf[12];
    unsigned short type;
    unsigned short machine;
    unsigned int   version;
    unsigned int   entry;
    unsigned int   phoff;
    unsigned int   shoff;
    unsigned int   flags;
    unsigned short ehsize;
    unsigned short phentsize;
    unsigned short phnum;
    unsigned short shentsize;
    unsigned short shnum;
    unsigned short shstrndx;
};
```

我们需要一块连续的内存空间来容纳 ELF 文件头，通过磁盘读取函数将游戏的 elf header 读入到选定的内存位置，通过 phoff 可以找到 ProgramHeader ，通过 phnum 可知道程序段的数目，通过 entry 可以得到程序的入口地址。

program header 的结构如下：

```
/* ELF32 Program header */
struct ProgramHeader {
    unsigned int type;
    unsigned int off;
    unsigned int vaddr;
    unsigned int paddr;
    unsigned int filesz;
    unsigned int memsz;
    unsigned int flags;
    unsigned int align;
};
```

通过磁盘读写函数将各程序段从磁盘加载到 paddr 处，并将 filesz, memsz 之间的物理区间清零。这个工作你已经在 PA2 中做过了，如果对此部分内容有所遗忘，你可以查看 PA2 讲义中与 loader 相关的部分和你自己实现的 loader 的代码。根据解析 elf header 的结果，你可以从磁盘中将整个游戏加载到内存中。

跳转到游戏

现在整个游戏的代码全部都在内存中，你终于可以运行它了。且慢，要让 C 语言的程序可以执行，需要提供合适的运行环境，最重要的就是栈空间，所以你需要给 esp 一个确定的值，并且保证它不会与代码段和数据段重叠。

之后，你就可以用从 ELF 头中获得的 entry 跳转到游戏的入口函数了。

如果你不修改链接脚本，不对 entry 符号做特殊处理，那么你很可能拿到的是一个虚拟地址的 entry，这个时候显然是不能使用的，所以我们建议你暂时先将游戏直接链接到物理地址空间，如何做可以参考后面我们提供的链接脚本和 Makefile 。

关于实现 bootloader 更多细节

1. 在此阶段中完成的汇编代码，建议你在 boot/boot.S 中完成，包括：关中断，开启 A20 地址线，开启保护模式，初始化运行环境；

2. 在此阶段中完成的 c 代码，建议你在 boot/main.c 中完成，包括：解决磁盘 I/O 的代码，解析 elf header 和加载余下部分游戏的代码，以及跳转到游戏。

3. 第一条指令你应该关中断，因为此时你还没有填写自己的 IDT ，如果开中断会因为 triple fault 而不断重启；

4. 定义段描述符的汇编代码应该放在主体代码的后面（这是为什么？）

串口输出

串口用于控制台的输出，对应的端口地址为 0x3f8 ~ 0x3ff。串口的初始化和判断空闲的代码如下：

```
#define PORT 0x3f8   /* COM1 */


void init_serial() {
   outb(PORT + 1, 0x00);
   outb(PORT + 3, 0x80);
   outb(PORT + 0, 0x03);
   outb(PORT + 1, 0x00);
   outb(PORT + 3, 0x03);
   outb(PORT + 2, 0xC7);
   outb(PORT + 4, 0x0B);
}

int is_serial_idle() {
   return inb(PORT + 5) & 0x20;
}
```

你不必掌握每个端口 IO 的具体含义，只需要知道 init_serial 完成了初始化工作，而 serial_idle 则用于判断对应的端口是否处于空闲状态。当串口处于空闲状态时，则可以通过 out 指令将一个字符输出到该端口完成一次输出。

https://en.wikipedia.org/wiki/COM_(hardware_interface) 请根据这个说明，参考已提供的代码自行完成串口输出函数。

你**可能**需要自己实现 `inb`, `outb` 等需要直接使用 x86 指令的函数，也就需要书写内联汇编代码，OSDev: Inline Assembly Examples 有一个份内联汇编的常用样例(cheat sheet!)，GCC-Inline-Assembly-HOWTO 对于理解内联汇编语法很有帮助。

实现 printk

从功能上来讲，printk 与 printf 并没有任何区别，它们的作用都是格式化输出。唯一的区别是：printk 工作在内核空间，printf 工作在用户空间。目前并没有操作系统内核的概念，但这不影响你使用 printk。

printk 可以接收不固定数目的参数(但至少要有一个)，gcc 会把这些参数从右到左压入堆栈。具体形式请自行查找资料，至于如何使用第二个以后的参数，相信聪明的你会想到办法的。

建议先将 printk 接收的格式化字符串转化为字符串常量，你需要实现%d, %x, %s, %c 四种格式转换说明符,，然后调用你自己封装好的输出函数将实际的字符串输出（具体输出到哪里请自行决定，你可以输出到串口，也可以输出到屏幕）

我们为大家提供了测试代码：

```
printk("Printk test begin...\n");
printk("the answer should be:\n");
printk("###########################################################\n");
```

```
printk("Hello, welcome to OSlab! I'm the body of the game. ");
printk("Bootblock loads me to the memory position of 0x100000, and Make
file also tells me that I'm at the location of 0x100000. ");
printk("~!@#$^&*()_+`1234567890-=..... ");
printk("Now I will test your printk: ");
printk("1 + 1 = 2, 123 * 456 = 56088\n0, -1, -2147483648, -1412505855,
-32768, 102030\n0, ffffffff, 80000000, abcdef01, ffff8000, 18e8e\n");
printk("##############################################################\n");
printk("your answer:\n");
printk("==============================================================\n");
printk("%s %s%scome %co%s", "Hello,", "", "wel", 't', " ");
printk("%c%c%c%c%c! ", 'O', 'S', 'l', 'a', 'b');
printk("I'm the %s of %s. %s 0x%x, %s 0x%x. ", "body", "the game", "Boo
tblock loads me to the memory position of",
    0x100000, "and Makefile also tells me that I'm at the location of",
 0x100000);
printk("~!@#$^&*()_+`1234567890-=..... ");
printk("Now I will test your printk: ");
printk("%d + %d = %d, %d * %d = %d\n", 1, 1, 1 + 1, 123, 456, 123 * 45
6);
printk("%d, %d, %d, %d, %d, %d\n", 0, 0xffffffff, 0x80000000, 0xabcedf0
1, -32768, 102030);
printk("%x, %x, %x, %x, %x, %x\n", 0, 0xffffffff, 0x80000000, 0xabcedf0
1, -32768, 102030);
printk("==============================================================\n");
printk("Test end!!! Good luck!!!\n");
```

实现了 printk 后, 你可以使用它帮助你调试代码，比如在游戏中输出变量的值.

阶段 2 游戏：中断与显示

硬件中断

游戏需要两种最基本的中断: 时间中断和键盘中断。为了接受和处理这两种中断，除了填写 IDT 以外，还需要对相关硬件进行初始化设置。下面简要地介绍如何初始化相关的硬件。

中断控制器

Intel 80386 需要可编程中断控制器(Programmable Interrupt Controller, PIC)的支持，才能响应多种外部设备的中断。qemu 使用 8259 中断控制器，在 qemu 控制台中，输入如下命令可以进行验证：

```
(qemu) help info pic
info pic  -- show i8259 (PIC) state
```

在 qemu 的控制台中使用 `info pic` 查看 PIC 的状态：

```
(qemu) info pic
pic0: irr=15 imr=b8 isr=00 hprio=0 irq_base=08 rr_sel=0 elcr=00 fnm=0
pic1: irr=40 imr=8e isr=00 hprio=0 irq_base=70 rr_sel=0 elcr=0c fnm=0
```

其中，需要关心的是 imr 和 irq_base。 imr 即 Interrupt Mask Register ，是中断位的掩码，为 1 时对应引脚的中断无效（一个 8259 有 8 个中断引脚）。 irq_base 是偏移量，中断引脚的编号加上 irq_base，才是用来查询中断描述符表(IDT)的下标。每个中断引脚对应的中断事件可以在 OSDev Interrupts 条目上查到，这里列出目前需要的两个中断：

IRQ	Description
0	Programmable Interrupt Timer Interrupt
1	Keyboard Interrupt

这里需要注意的是，以时间中断为例， IRQ(0H) + irq_base(08H) = 08H，这个编号在 IDT 中属于 Intel 保留的中断，对应 Double Fault。所以在开启中断前需要对 PIC 进行初始化，主要工作如下：

1. 设置需要的中断屏蔽
2. 设置中断号偏移量等属性

考虑到 8259 的各种端口的使用不是实验的重点，所以提供一份流行的初始化 8259 控制器的代码，请自行封装成函数并在合适的地方调用它：

```
#define IO_PIC1 0x20
#define IO_PIC2 0xA0
#define IRQ_OFFSET 0x20
#define IRQ_SLAVE 2

// modify interrupt masks
outb(IO_PIC1 + 1, 0xFF);
outb(IO_PIC2 + 1, 0xFF);

// Set up master (8259A-1)

// ICW1:  0001g0hi
//    g:  0 = edge triggering, 1 = level triggering
//    h:  0 = cascaded PICs, 1 = master only
//    i:  0 = no ICW4, 1 = ICW4 required
outb(IO_PIC1, 0x11);

// ICW2:  Vector offset
outb(IO_PIC1+1, IRQ_OFFSET);

// ICW3:  bit mask of IR lines connected to slave PICs (master PIC),
```

```
//        3-bit No of IR line at which slave connects to master(slave P
IC).
outb(IO_PIC1+1, 1 << IRQ_SLAVE);

// ICW4:  000nbmap
//    n:  1 = special fully nested mode
//    b:  1 = buffered mode
//    m:  0 = slave PIC, 1 = master PIC
//    (ignored when b is 0, as the master/slave role
//    can be hardwired).
//    a:  1 = Automatic EOI mode
//    p:  0 = MCS-80/85 mode, 1 = intel x86 mode
outb(IO_PIC1+1, 0x1);

// Set up slave (8259A-2)
outb(IO_PIC2, 0x11);                 // ICW1
outb(IO_PIC2 + 1, IRQ_OFFSET + 8);// ICW2
outb(IO_PIC2 + 1, IRQ_SLAVE);     // ICW3
// NB Automatic EOI mode doesn't tend to work on the slave.
// Linux source code says it's "to be investigated".
outb(IO_PIC2 + 1, 0x01);          // ICW4

// OCW3:  0ef01prs
//   ef:  0x = NOP, 10 = clear specific mask, 11 = set specific mask
//    p:  0 = no polling, 1 = polling mode
//   rs:  0x = NOP, 10 = read IRR, 11 = read ISR
outb(IO_PIC1, 0x68);             /* clear specific mask */
outb(IO_PIC1, 0x0a);             /* read IRR by default */

outb(IO_PIC2, 0x68);             /* OCW3 */
outb(IO_PIC2, 0x0a);             /* OCW3 */
```

上述代码头两行 out 指令将所有中断都屏蔽了，你可以直接在这里将 `0xFF` 修改成你需要的屏蔽模式（至少取消键盘和时钟中断的屏蔽），也可以在别的地方下面这样迭代地修改：

```
uint8_t mask = inb(IO_PICX + 1);
outb(IO_PICX + 1, mask ^ BIT_FOR_THIS_IRQ);
```

时钟中断

时钟(Programmable Interval Timer, PIT)也需要进行一些初始化工作，主要目的是设置频率。详细信息可以参考 http://wiki.osdev.org/Programmable_Interval_Timer，这里做简单的介绍。

PIT 有三个 channel 产生脉冲，其中 channel 0 与 IRQ 0 直接相连，所以我们需要的是 channel 0 的脉冲。 PIT 的端口表如下：

I/O port	Usage
0x40	Channel 0 data port (read/write)
0x41	Channel 1 data port (read/write)
0x42	Channel 2 data port (read/write)
0x43	Mode/Command register (write only, a read is ignored)

下面用 C 代码演示基本的初始化步骤：

```
#define PORT_CH_0 0x40
#define PORT_CMD 0x43
#define PIT_FREQUENCE 1193182
#define HZ 100

union CmdByte {
  struct {
    uint8_t present_mode : 1;
    uint8_t operate_mode : 3;
    uint8_t access_mode  : 2;
    uint8_t channel      : 2;
  };
  uint8_t val;
};

union CmdByte mode = {
  .present_mode = 0,  // 16-bit binary
  .operate_mode = 2,  // rate generator, for more accuracy
  .access_mode  = 3,  // low byte / high byte, see below
  .channel      = 0,  // use channel 0
};

int counter = PIT_FREQUENCE / HZ
outb(PORT_CMD, mode.val);
outb(PORT_CH_0, counter & 0xFF);         // access low byte
outb(PORT_CH_0, (counter >> 8) & 0xFF);  // access high byte
```

与 8295 一样，请自行封装该代码，在内核中你认为合适的地方调用，时钟的频率 HZ 也可以视具体情况调整。

键盘中断

键盘中断没有什么需要初始化的，只需要发生键盘中断后，在对应的中断处理程序里，用 `inb(0x60)`获取键盘吗进行相应的操作即可。键盘中断具体采取什么行为与你的系统架构和游戏设计有关。可以参考 PA 的打字小游戏和 sdlpal 。

图像输出

由于实验要求在 VGA 图形模式下制作一个小游戏，所以首先要决定使用何种图形模式，并做相应的设置。

常规情况下，我们可以使用 320 x 200 x 256 色的标准 VGA 显式模式。PA 的打字小游戏和 sdlpal 也是在这种模式下显式图像的，所以你们应该也很清楚这种模式存在调色版这种东西，下图是默认情况下调色板的 256 种颜色：

img

img

标准 VGA 下的图形输出

设置图形模式

为了使用标准 VGA 模式，需要将显示方式设置为图形模式。在开启保护模式以后，要想开启图形模式需要进行复杂的 in/out 操作，这显然是你不想看到的。 考虑到我们的调试信息通过串口输出，在启动过程中，屏幕上没有输出信息的需求，所以建议大家在**进入保护模式之前**就开启图形模式(在 boot.S 中实现)，开启图形模式的代码只有 2 行：

```
movw  $0x13, %ax
int   $0x10
```

上述代码是把 0x13 作为参数调用 BIOS 中断，因此这种模式又有一个有趣的名字叫做“Mode_13h” (https://en.wikipedia.org/wiki/Mode_13h) ##### 显存 设置好图形模式之后，我们就可以输出了，怎么输出呢？ 这是低端内存空间内存的映射情况，可以看到 0xa0000~0xb0000 段的空间是可以用于 VGA 显示的：

img

img

Mode 13h 中使用的颜色深度 (https://en.wikipedia.org/wiki/Color_depth) 是 8-bit color，也就是用一个 byte 表示一个像素点的颜色，分辨率是 320*200。 因此整个屏幕的颜色值可以存放在一个 320*200 的 unsigned char 类型数组 color_buffer 中。要想修改屏幕上坐标为(x,y)的点 的颜色值，你可以用 color_buffer[x*320+y]来访问它。当你填好所有的颜色的值之后，将这个 buffer 拷贝到显存的起始处即可。

单个像素点的颜色

那么颜色的值怎么取呢？这是 mode 13h 默认的调色板：

img

img

左上角的黑色对应的数值是 0，它右边的蓝色对应的数值是 1，以此类推。 如果你想玩得更嗨，使用自己的图片和自己的调色板，那么 2012 级的 oslab -1 中有相关介绍

真彩色支持

如果你们的游戏只有一些简单的几何体和字符，那么上面所说的标准 VGA 模式和默认的调色板就够用了，但是如果需要使用图片素材，则可能需要修改调色板，在 32 位保护模式下，只能通过直接的端口 IO 进行替换，而且如果要同时呈现两张图片的话，则比较困难。

qemu 2.0.0 支持 vbe 3.0 扩展，扩展提供了更多高分辨率的显式模式和 24 位真彩色的支持。关于如何设置扩展的显示模式，可以参考 VESA Functions。由于 `int 10H` 等 BIOS 中断只能在 16 位模式下使用，所以设置模式也只能在 boot loader 的早期进行。然而从 vbe 2.0 往后，官方不再规定统一的模式编号（不再分配新的模式号，旧的模式号不需要兼容），一张显卡支持哪些模式，需要获得 Controller Info 进而获得 Mode Info 数组进行查询，在 512B 的 boot loader 中实现这些功能是比较困难的，或者你可以设计两段 boot loader，延长位于 16 位实模式下的时间。

一个取巧的策略是，虽然旧的模式官方不要求兼容，但是实际上也没谁会没事找事去不兼容，所以可以从这里找到一些通用的模式编号，试出可用的模式。比如，在 qemu 2.0.0 上，模式 800 x 600 x 24-bit (对应编号 0115H) 就确实是可用的，显示图片的效果如下：

img

img

一个额外需要注意的问题就是显存的位置。做过 PA 的各位都知道标准 VGA 模式的显存地址从 0xA0000 开始，但是 vbe 模式则没有统一规定，虽然 0xA0000 映射了一部分显存，但是 0xA0000 一般不会是真正的显存起始地址。所以还是需要根据上面提供的链接去获取 Mode Info。从 Memory Map 中查看在 1MB 内存中哪些部分可以用来存放数据，然后内核和 boot loader 互相约定一个相同的物理地址当做 Mode Info 的地址，然后在内核初始化时用 Mode Info 的 physbase 域确定显存的**物理地址**。

补充：参考工程配置

这里提供一份参考的项目结构，以及 Makefile、Boot Sector 制作脚本和链接脚本。

参考项目结构：

```
.
├── boot
├── kernel
│   └── kernel.ld
```

```
├── lib
├── Makefile
└── mbr.rb
```

参考 Makefile:

```
BOOT   := boot.bin
KERNEL := kernel.bin
IMAGE  := disk.bin

CC      := gcc
LD      := ld
OBJCOPY := objcopy
DD      := dd
QEMU    := qemu-system-i386
GDB     := gdb

CFLAGS := -Wall -Werror -Wfatal-errors #开启所有警告，视警告为错误，第一个
错误结束编译
CFLAGS += -MD #生成依赖文件
CFLAGS += -std=gnu11 -m32 -c #编译标准，目标架构，只编译
CFLAGS += -I . #头文件搜索目录
CFLAGS += -O0 #不开优化，方便调试
CFLAGS += -fno-builtin #禁止内置函数
CFLAGS += -ggdb3 #GDB 调试信息

QEMU_OPTIONS := -serial stdio #以标准输入输为串口(COM1)
QEMU_OPTIONS += -d int #输出中断信息
QEMU_OPTIONS += -monitor telnet:127.0.0.1:1111,server,nowait #telnet mo
nitor

QEMU_DEBUG_OPTIONS := -S #启动不执行
QEMU_DEBUG_OPTIONS += -s #GDB 调试服务器: 127.0.0.1:1234

GDB_OPTIONS := -ex "target remote 127.0.0.1:1234"
GDB_OPTIONS += -ex "symbol $(KERNEL)"

OBJ_DIR        := obj
LIB_DIR        := lib
BOOT_DIR       := boot
KERNEL_DIR     := kernel
OBJ_LIB_DIR    := $(OBJ_DIR)/$(LIB_DIR)
OBJ_BOOT_DIR   := $(OBJ_DIR)/$(BOOT_DIR)
OBJ_KERNEL_DIR := $(OBJ_DIR)/$(KERNEL_DIR)

LD_SCRIPT := $(shell find $(KERNEL_DIR) -name "*.ld")

LIB_C := $(wildcard $(LIB_DIR)/*.c)
```

```
LIB_O := $(LIB_C:%.c=$(OBJ_DIR)/%.o)

BOOT_S := $(wildcard $(BOOT_DIR)/*.S)
BOOT_C := $(wildcard $(BOOT_DIR)/*.c)
BOOT_O := $(BOOT_S:%.S=$(OBJ_DIR)/%.o)
BOOT_O += $(BOOT_C:%.c=$(OBJ_DIR)/%.o)

KERNEL_C := $(shell find $(KERNEL_DIR) -name "*.c")
KERNEL_S := $(wildcard $(KERNEL_DIR)/*.S)
KERNEL_O := $(KERNEL_C:%.c=$(OBJ_DIR)/%.o)
KERNEL_O += $(KERNEL_S:%.S=$(OBJ_DIR)/%.o)

$(IMAGE): $(BOOT) $(KERNEL)
    @$(DD) if=/dev/zero of=$(IMAGE) count=10000         > /dev/null #
准备磁盘文件
    @$(DD) if=$(BOOT) of=$(IMAGE) conv=notrunc          > /dev/null #
填充 boot loader
    @$(DD) if=$(KERNEL) of=$(IMAGE) seek=1 conv=notrunc > /dev/null #
填充 kernel, 跨过 mbr

$(BOOT): $(BOOT_O)
    $(LD) -e start -Ttext=0x7C00 -m elf_i386 -nostdlib -o $@.out $^
    $(OBJCOPY) --strip-all --only-section=.text --output-target=binary
$@.out $@
    @rm $@.out
    ruby mbr.rb $@

$(OBJ_BOOT_DIR)/%.o: $(BOOT_DIR)/%.S
    @mkdir -p $(OBJ_BOOT_DIR)
    $(CC) $(CFLAGS) -Os $< -o $@

$(OBJ_BOOT_DIR)/%.o: $(BOOT_DIR)/%.c
    @mkdir -p $(OBJ_BOOT_DIR)
    $(CC) $(CFLAGS) -Os $< -o $@

$(KERNEL): $(LD_SCRIPT)
$(KERNEL): $(KERNEL_O) $(LIB_O)
    $(LD) -m elf_i386 -T $(LD_SCRIPT) -nostdlib -o $@ $^ $(shell $(CC)
$(CFLAGS) -print-libgcc-file-name)

$(OBJ_LIB_DIR)/%.o : $(LIB_DIR)/%.c
    @mkdir -p $(OBJ_LIB_DIR)
    $(CC) $(CFLAGS) $< -o $@

$(OBJ_KERNEL_DIR)/%.o: $(KERNEL_DIR)/%.[cS]
    mkdir -p $(OBJ_DIR)/$(dir $<)
    $(CC) $(CFLAGS) $< -o $@

DEPS := $(shell find -name "*.d")
```

```
-include $(DEPS)

.PHONY: qemu debug gdb clean

qemu: $(IMAGE)
    $(QEMU) $(QEMU_OPTIONS) $(IMAGE)

# Faster, but not suitable for debugging
qemu-kvm: $(IMAGE)
    $(QEMU) $(QEMU_OPTIONS) --enable-kvm $(IMAGE)

debug: $(IMAGE)
    $(QEMU) $(QEMU_DEBUG_OPTIONS) $(QEMU_OPTIONS) $(IMAGE)

gdb:
    $(GDB) $(GDB_OPTIONS)

clean:
    @rm -rf $(OBJ_DIR) 2> /dev/null
    @rm -rf $(BOOT)    2> /dev/null
    @rm -rf $(KERNEL)  2> /dev/null
    @rm -rf $(IMAGE)   2> /dev/null
```

参考内核链接脚本 kernel.ld：

```
OUTPUT_FORMAT(
    "elf32-i386",  /* Default       */
    "elf32-i386",  /* Big endian    */
    "elf32-i386"   /* Little endian */
)
OUTPUT_ARCH(i386)
ENTRY(main)  /* 入口函数，你可以不定义成 main，只要是代码中的符号即可 */

SECTIONS
{
    /**
     * 可执行程序的地址空间从 1MB (2^20 B) 往上开始
     */
    . = 0x100000;

    PROVIDE(start = .);

    .text : AT(0x100000) {  /* AT 指定物理地址 */
        *(.text .text.*)
    }

    PROVIDE(etext = .);  /* 定义符号 etext, 其值为代码段之后的地址 */

    .rodata : {
```

```
        *(.rodata .rodata.*)
    }

    . = ALIGN(0x1000);  /* 将数据段按页对齐 */

    .data : {
        *(.data)
    }

    .bss : {
        *(.bss)
    }

    PROVIDE(end = .);  /* 定义符号 end, 其值为程序/数据段之后的地址 */
}
```

Boot Sector 制作脚本(Ruby 版本，你可以用任何语言来制作)：

```
#!/usr/bin/env ruby

obj = open(ARGV[0], "ab")

if obj.size <= 510
    fill = 510 - obj.size
    fill.times { obj.write("\x00") }
    obj.write("\x55")
    obj.write("\xaa")
else
    puts "#{ARGV[0]}'s size is too large"
end
```

这份 Makefile 中包含了一些调试功能，比如：

1. GDB 调试：在一个终端输入 `make debug` 启动 qemu，在另一个终端输入 `make gdb` 启动 gdb 进行调试
2. 启动 qemu 后，在另一个终端输入 `telnet 127.0.0.1 1111` 登陆 qemu 的控制台，主要使用 `info registers` 查看 CPU 的完整状态
3. qemu 的 `-serial stdio` 将串口输出到标准输出
4. qemu 的 `-d int` 将打印所有的中断事件，在开中断前可以方便地检查造成 triple fault 的原因