**哈尔滨工业大学计算机科学与技术学院**

**2017年秋季学期《操作系统》**

**实验报告**

|  |  |  |
| --- | --- | --- |
| **姓名** | **学号** | **联系方式** |
| **韩越** | **1151230107** | **18846828245** |

目录

[一、xv6实验系统简述 2](#_Toc503330483)

[二、xv6环境配置（附过程截图） 2](#_Toc503330484)

[三、实验过程（lab1~lab6） 5](#_Toc503330485)

[Lab 1: Booting a PC 5](#_Toc503330486)

[1. 实验目的 5](#_Toc503330487)

[2. 实验内容 5](#_Toc503330488)

[3. 实验结果 22](#_Toc503330489)

[Lab 5: File system, Spawn and Shell 23](#_Toc503330490)

[1. 实验目的 23](#_Toc503330491)

[2. 实验内容 23](#_Toc503330492)

[3. 实验结果 41](#_Toc503330493)

[四、xv6实验系统分析 42](#_Toc503330494)

[（一） xv6的系统结构（结合示意图） 42](#_Toc503330495)

[（二）xv6的引导 42](#_Toc503330496)

[（三）xv6的进程与调度分析 43](#_Toc503330497)

[（四）xv6的内存管理 43](#_Toc503330498)

[（五）xv6的文件系统 43](#_Toc503330499)

[1. 磁盘结构 43](#_Toc503330500)

[2. 文件元数据 44](#_Toc503330501)

[3. 目录与普通文件 45](#_Toc503330502)

[4. 块缓存 45](#_Toc503330503)

[5. 磁盘ide驱动文件 48](#_Toc503330504)

[6. 文件操作 48](#_Toc503330505)

[7. 文件系统服务器 49](#_Toc503330506)

[8. 用户使用的fd文件描述符 50](#_Toc503330507)

[（六）xv6 的I/O 50](#_Toc503330508)

[五、总结与心得 50](#_Toc503330509)

# 一、xv6实验系统简述

Xv6是由麻省理工学院(MIT)为操作系统工程的课程（代号6.828）开发的一个教学目的的操作系统。

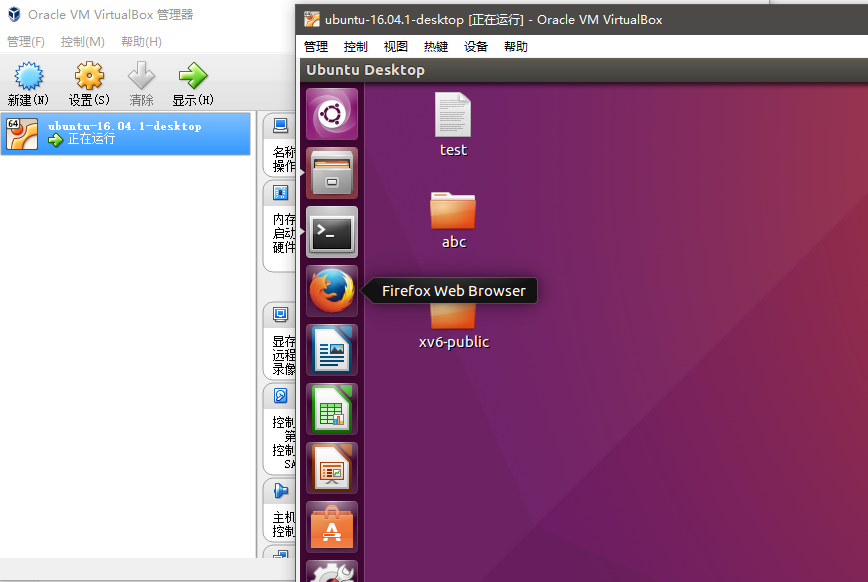
Xv6是在x86处理器上(x即指x86)用ANSI标准C重新实现的Unix第六版(Unix V6，通常直接被称为V6)。Unix V6是1975年发布的，基于DEC PDP-11小型机，当时还没有x86系列CPU，而现在PDP的机器已经很少见了，当时使用是在标准ANSI C发布之前的旧式C语言。

与Linux或BSD系统不同，Xv6很简单，可以在一个学期讲完，全部代码只有8千行多，但仍包括了Unix的重要概念和组织结构。由于是基于较早的Unix V6，Xv6的结构与现代操作系统，如Linux，Windows的差距较大。

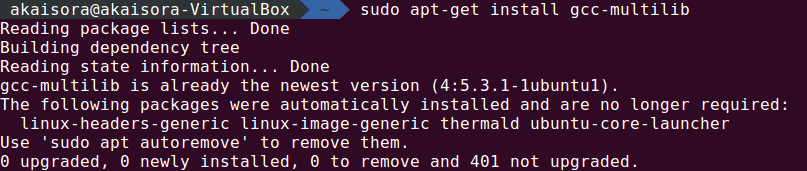
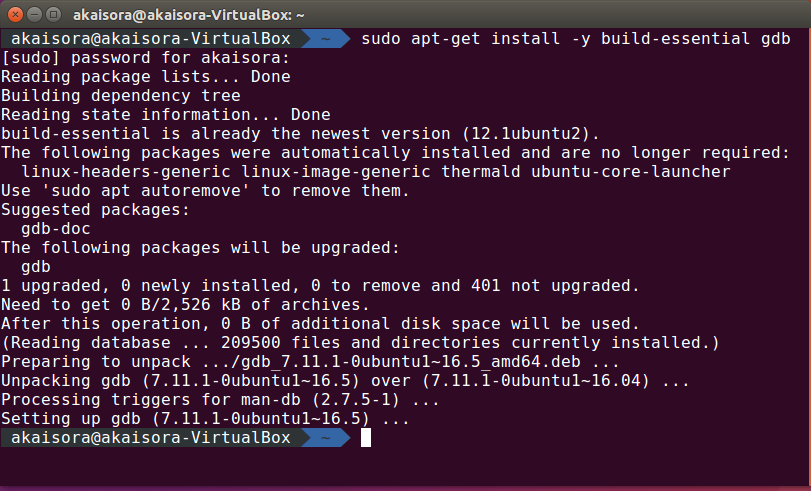
在MIT以外，很多其它大学也在操作系统课程中使用了Xv6或其变种，如耶鲁，清华等。

# 二、xv6环境配置（附过程截图）

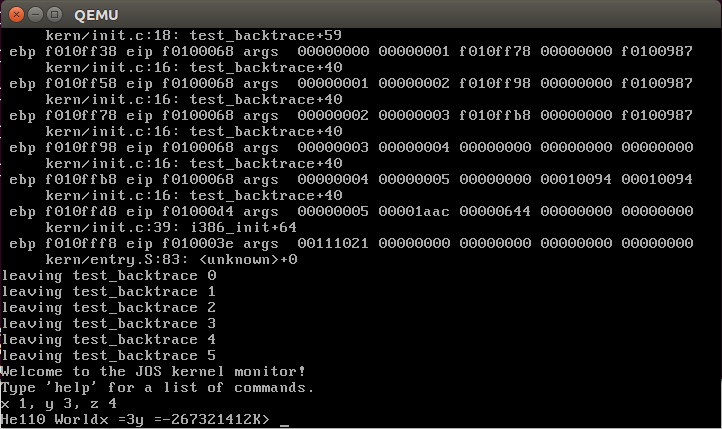
我在VirtualBox虚拟机上安装了ubuntu16.04 64bit作为实验环境。



安装工具链



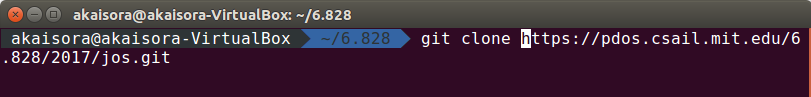
安装qemu



为了从mit的服务器上下载实验代码和维护自己的代码，还需要安装和配置git



从mit的git服务器上下载实验1所需代码，尝试运行make, make qemu。





# 三、实验过程（lab1~lab6）

## Lab 1: Booting a PC

### 实验目的

这个实验分为三个部分。第一部分着重于熟悉x86汇编语言，QEMU x86仿真器和PC的开机引导过程。第二部分检查我们的6.828内核的引导加载程序，它的文件主要在lab的boot目录中。最后，第三部分深入研究了6.828内核本身的初始模板，名为JOS，主要在在kernel目录中。

### 实验内容

#### 第一部分：PC启动引导

##### 汇编入门

Exercise1

熟悉汇编语言材料，可以参考6.828参考页

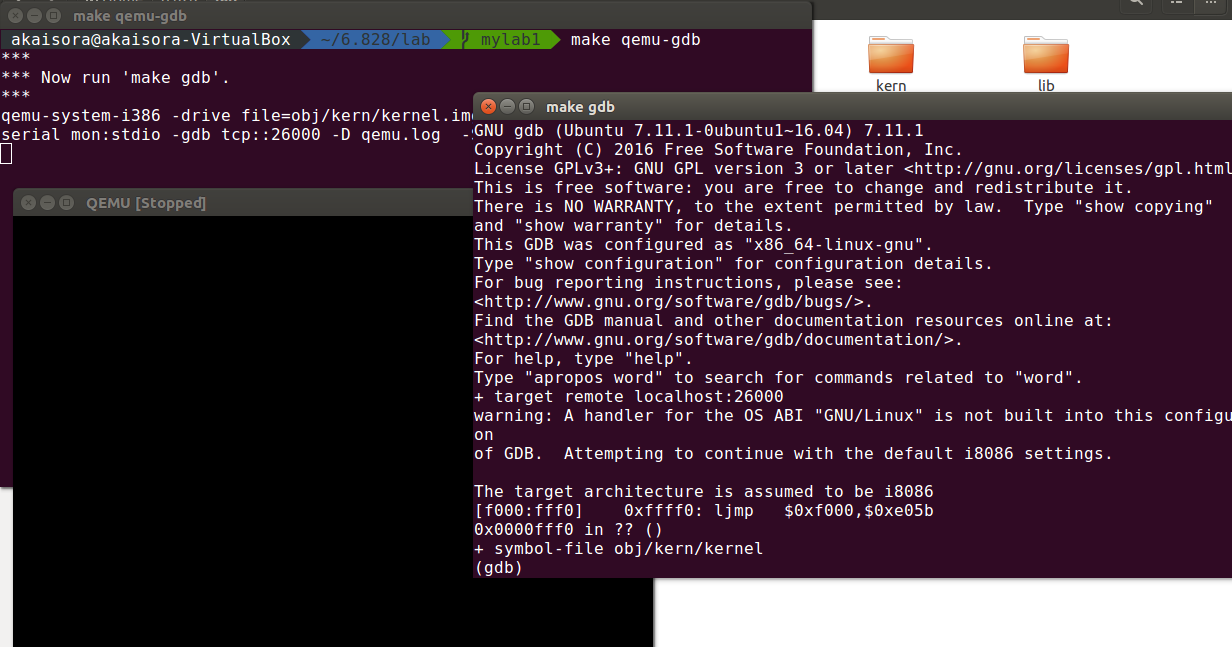
<https://pdos.csail.mit.edu/6.828/2017/reference.html>

和Brennan的“内联程序指南中的“The Syntax”一节 。它给出了一个很好的（很简短的）AT＆T汇编语法的描述，我们将在JOS中使用GNU汇编语言。

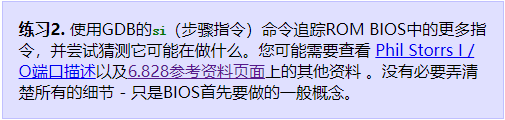
<http://www.delorie.com/djgpp/doc/brennan/brennan_att_inline_djgpp.html>

1. 模拟x86

lab1主要工作环境是两个端口，一个运行make qemu-gdb，开启qemu模拟器，另一个运行 make gdb，用于调试程序运行过程。



Exercise2



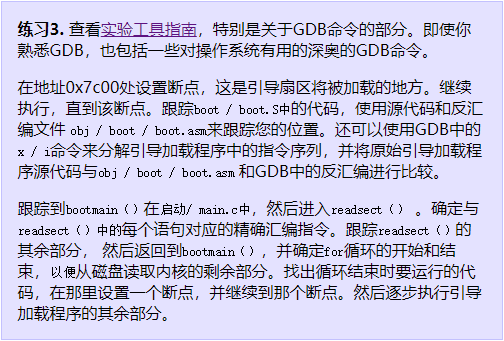
熟悉GDB中的si指令，si n指令表示执行n条指令，可以用于调试程序。

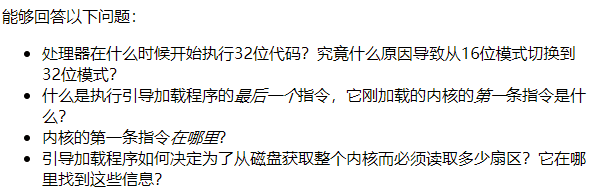
#### 第二部分：引导加载程序

##### BIOS引导加载

当BIOS找到可引导的软盘或硬盘时，它将512字节的引导扇区加载到物理地址0x7c00到0x7dff的内存中，然后使用jmp指令将CS：IP设置为0000：7c00，将控制权交给boot loader。

Exercise3





问题1：

由0x7c2d处的jump跳转到0x7c32的32位代码

0x7c2d：

# Jump to next instruction, but in 32-bit code segment.

# Switches processor into 32-bit mode.

ljmp $PROT\_MODE\_CSEG, $protcseg

0x7c32：

.code32 # Assemble for 32-bit mode

protcseg:

# Set up the protected-mode data segment registers

movw $PROT\_MODE\_DSEG, %ax # Our data segment selector

movw %ax, %ds # -> DS: Data Segment

movw %ax, %es # -> ES: Extra Segment

movw %ax, %fs # -> FS

movw %ax, %gs # -> GS

movw %ax, %ss # -> SS: Stack Segment

模式转变操作orl $CR0\_PE\_ON, %eax，movl %eax, %cr0更改cr0寄存器PE位为1

0x7c2a：

# Switch from real to protected mode, using a bootstrap GDT

# and segment translation that makes virtual addresses

# identical to their physical addresses, so that the

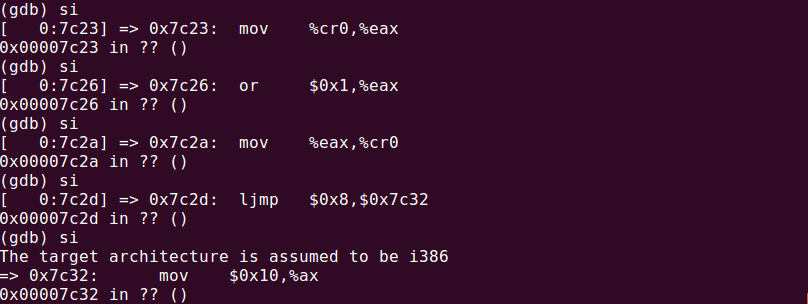
# effective memory map does not change during the switch.

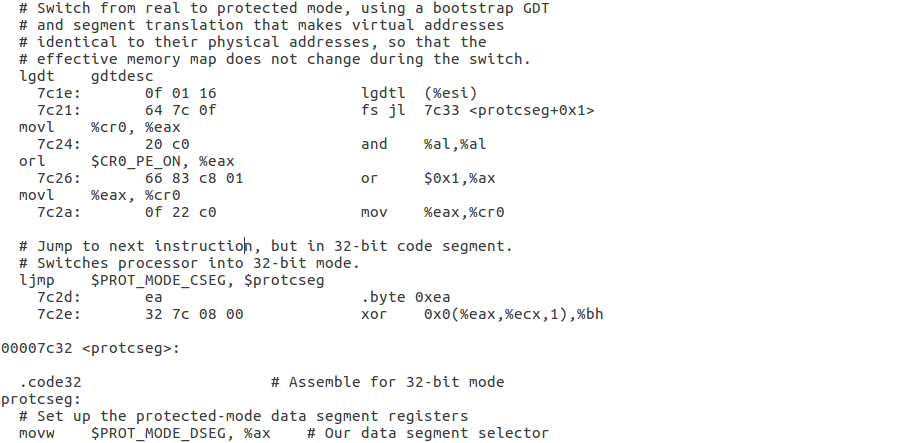
lgdt gdtdesc

movl %cr0, %eax

orl $CR0\_PE\_ON, %eax

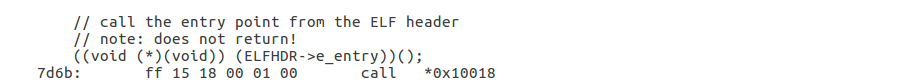
movl %eax, %cr0





问题2：

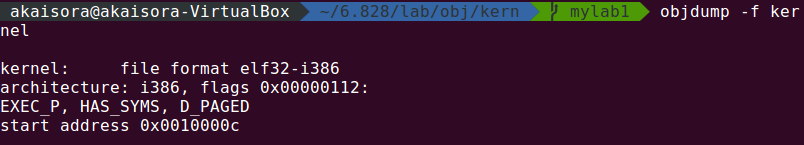
最后一条指令：



Kernel第一条指令:

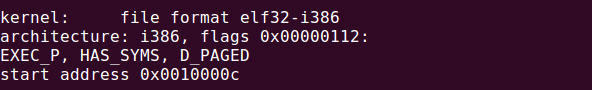
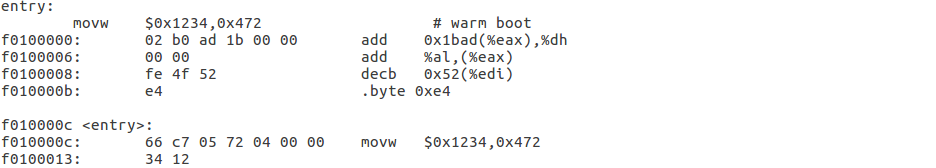


最后一条指令和进入kernel的第一条指令：



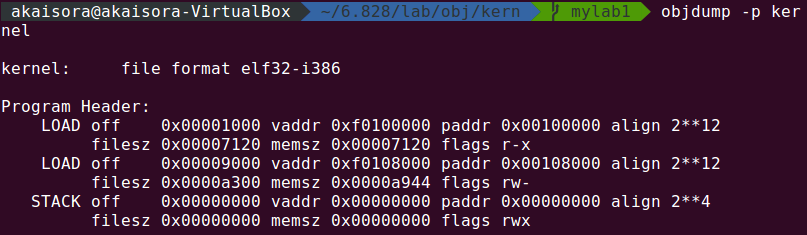
问题3：

在entry处



问题4：

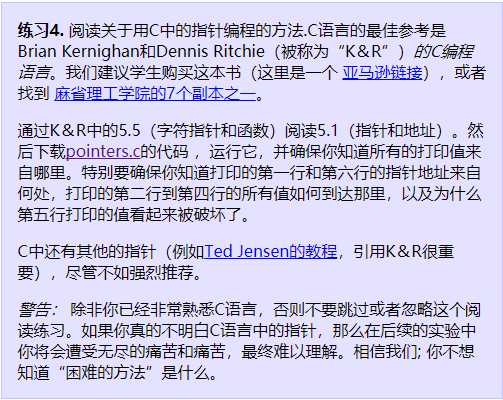
使用Program Header Table信息。



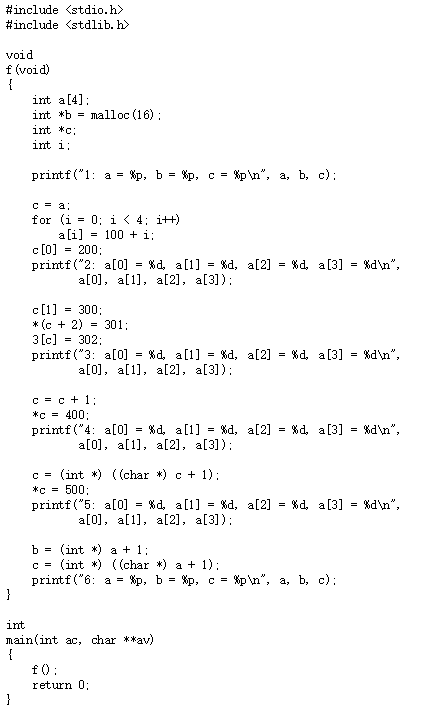
1. 加载内核

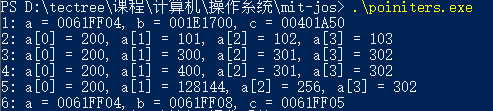
内核的加载引导主要由/boot/main.c完成。

Exercise4

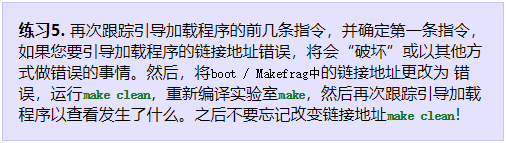


本练习主要是熟悉c语言中指针的操作，尤其指针类型变量加减时隐含的size信息。

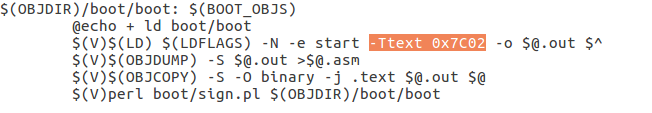




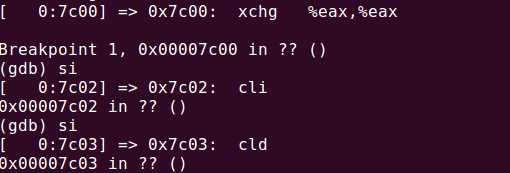
Exercise5



尝试将-Ttext值改为0x7C02，向后偏移2字节。

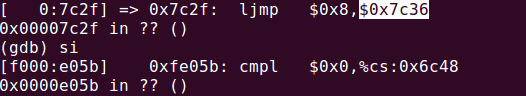


发现启动无法正常完成。

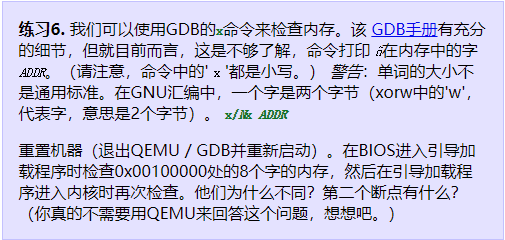


地址向后偏移了2字节。转为保护模式后跳转地址错误，本来应该是0x7c32。

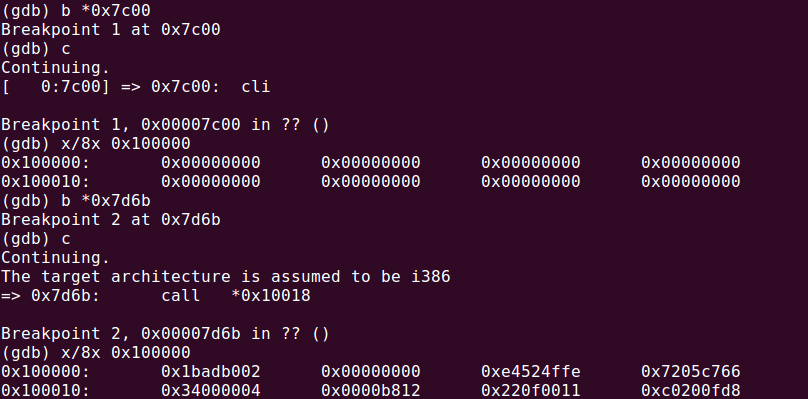
这是因为内核的加载地址固定在0x7C00，不论-Ttext值改为什么，但是-Ttext值会影响链接和地址计算，于是在跳转时会跳转到错误的地址。

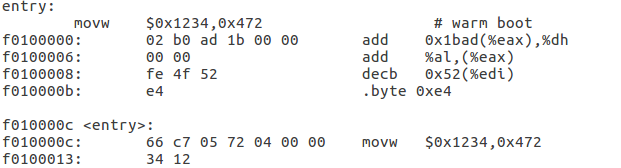


Exercise6



第一次全0，因为内核还没加载。第二次的内容与kernel入口的反编译结果相同，是kernel入口加载在0x00100000的位置。



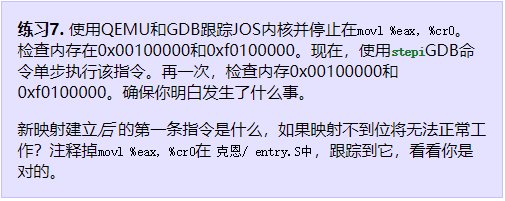


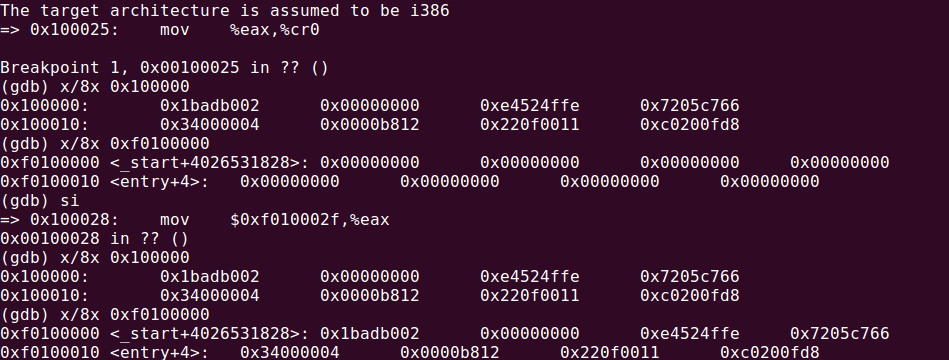
#### 第三部分：内核

1. 使用虚拟内存

操作系统内核通常希望被链接到非常高的虚拟地址（如0xf0100000），以便为用户程序留下处理器虚拟地址空间的较低部分。但是许多机器在地址0xf0100000处没有任何物理内存，所以我们不能指望能够在那里存储内核。相反，我们将使用处理器的内存管理硬件将虚拟地址0xf0100000（内核代码期望运行的链接地址）映射到物理地址0x00100000（引导加载程序将内核加载到物理内存中）。这样，尽管内核的虚拟地址足够高，为用户进程留下了足够的地址空间，但是它将会被加载到PC ROM内1MB的物理内存中，位于BIOS ROM的正上方。这种方法要求PC具有至少几兆字节的物理内存（物理地址0x00100000起作用）。事实上，在接下来的实验中，我们将映射了整个经过0x0fffffff到0xFFFFFFFF分别PC的物理地址空间的底部256MB，从物理地址0x00000000，然后虚拟地址0xf0000000。所以JOS现在只能使用第一个256MB的物理内存。

Exercise7



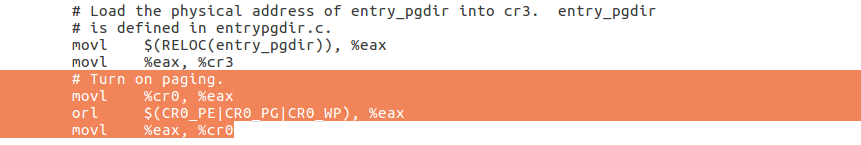


在未执行页面映射时0xf0100000处内容为空，执行后变为kernel入口的内容。

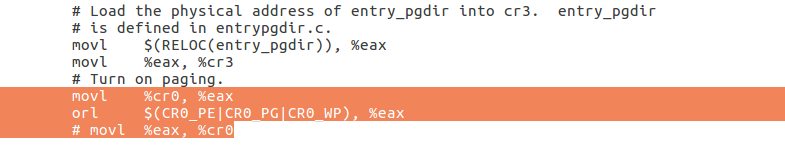
movl %cr0, %eax

orl $(CR0\_PE|CR0\_PG|CR0\_WP), %eax

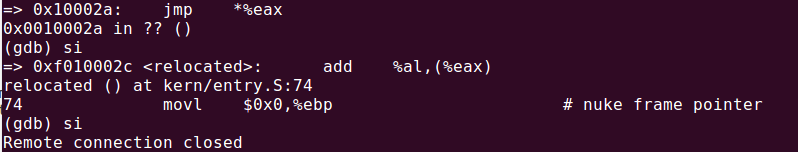
movl %eax, %cr0

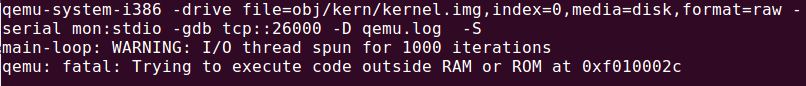


注释后：



很快就因为访问非法内存0xf010002c而出错。

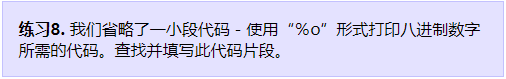


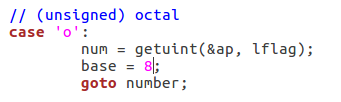


1. 格式化打印

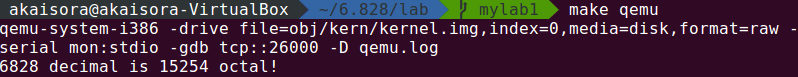
在一个操作系统内核中，我们必须自己实现所有的I / O。包括格式化打印到console。

Exercise8



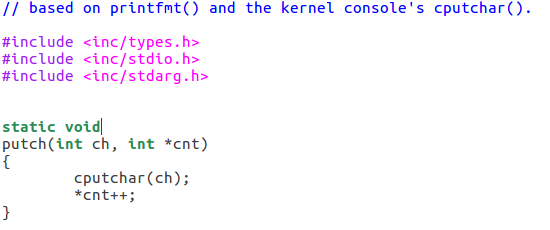


启动QEMU时观察到八进制数被正确输出。

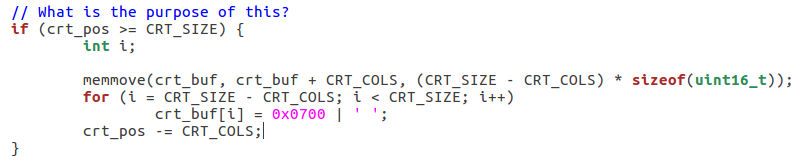


问题1：

console.c导出了cputchar()



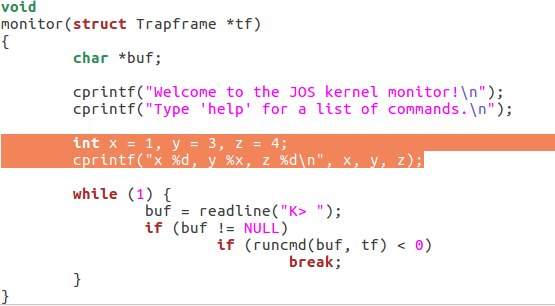
问题2：



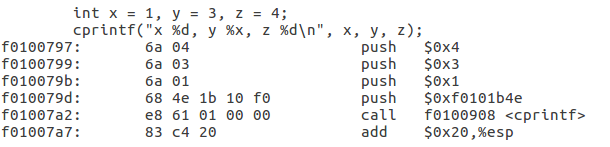
整个console是一个buffer，CRT\_SIZE是buffer的容量，crt\_pos是光标位置，当光标位置达到屏幕边界时会将所有内容向前移动列数，也就是向前移动一行，然后在最后一行用空格覆盖，最后将光标移动到行首。

问题3：

在monitor.c中加入下面两行



重新make qemu，可以在反汇编代码kernel.asm里找到两行语句的位置，由此可以追踪cprintf执行过程。

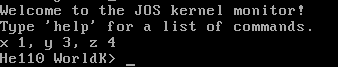




可以看到cprintf的后3个参数是从右往左压栈，ap+1依次取得每个参数，调用va\_arg前后ap分别指向当前参数和下一个参数。Fmt指向模式串参数，ap指向

问题4：





输出是He110 World。57616的16表示为e110，16进制数00646c72在小端机器内存中的表示为72 6c 64 00，转化为字符为rld[0]。在大端表示的机器中，数字在内存中的表示顺序是反过来的，所以00646c72应该反序写。而57616由cprintf输出的16进制数不会变所以57616不用改。

问题5：



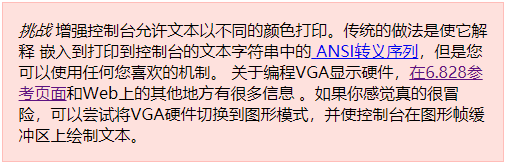


Y的值没有给参数，所以y的值由栈的下一位内容决定。

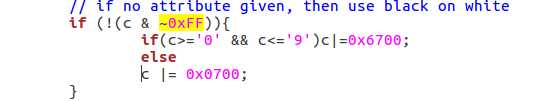
问题6：

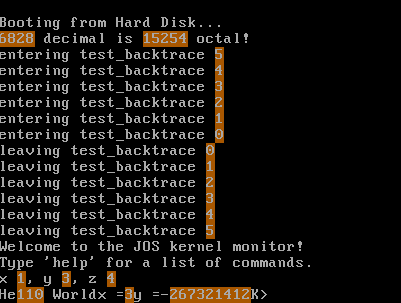
让cprintf传入参数个数cnt，这样可以esp+cnt来找到第一个参数。

Challenge



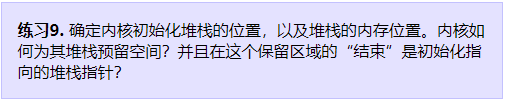
在控制台中发现控制颜色的方式是字符16位数中的高8位，将默认颜色设置改为当判断是数字时将背景色改为棕色。于是可以得到。



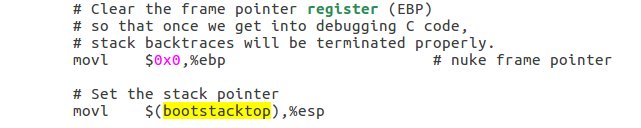


1. 堆栈

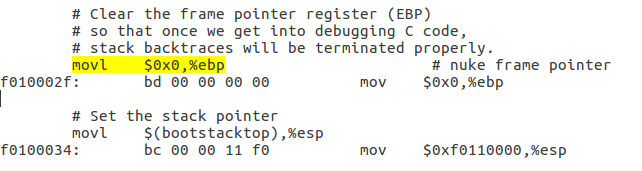
Exercise9

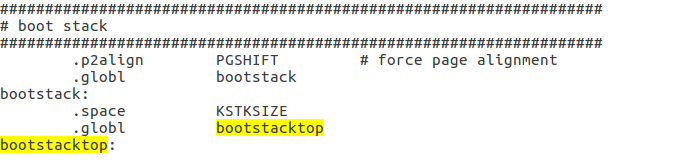


Entry.S中下面两条指令初始化ebp和esp，指定栈底和栈顶位置。在Kernel.asm中看到栈顶的位置为0xf0110000。栈顶下方的内容预留为栈空间，大小为KSTKSIZE，KSTKSIZE=(8\*PGSIZE)，PGSIZE定义在mmu.h中，为8kb，也就是说栈大小为32kb。

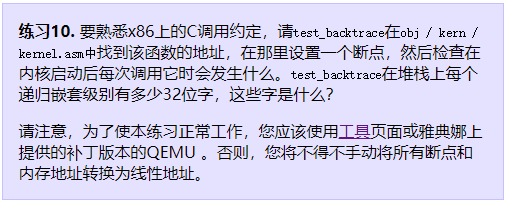


Kernel.asm





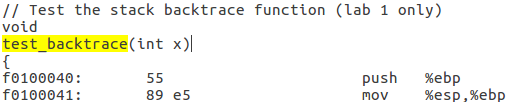
Exercise10



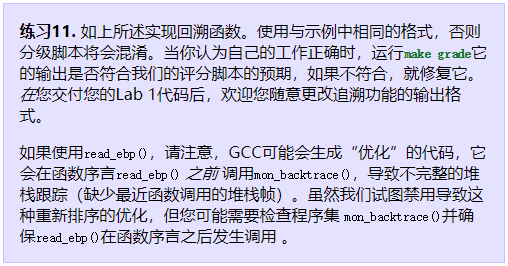
进入test\_backtrace时会将旧栈底压入，使用esp作为新栈底（就是建立新栈帧）。退出时反过来恢复栈。

栈帧的结构为：

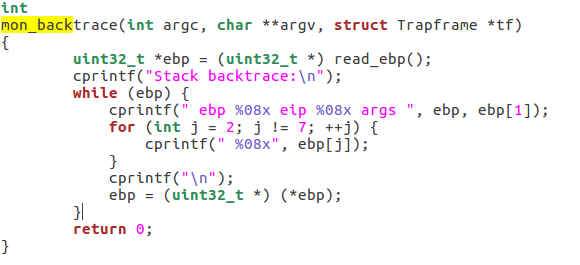
|  |
| --- |
| saved ebp |
| saved ebx |
| args5 |
| args4 |
| args3 |
| args2 |
| args1 |
| ret eip |



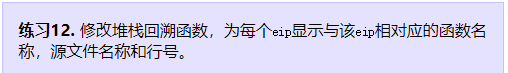
Exercise11



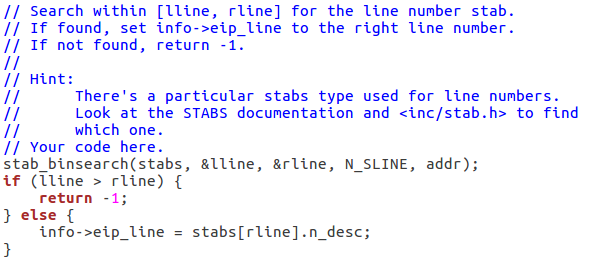
使用mon\_backtrace来追踪栈中多层调用的结构，在mon\_backtrace中添加如下代码，总最里面的ebp开始一次返回上一个ebp追踪到每一个函数的信息。判断到调用头的方法是看ebp是不是0，因为在entry.S中初始化时将ebp初始化为0。

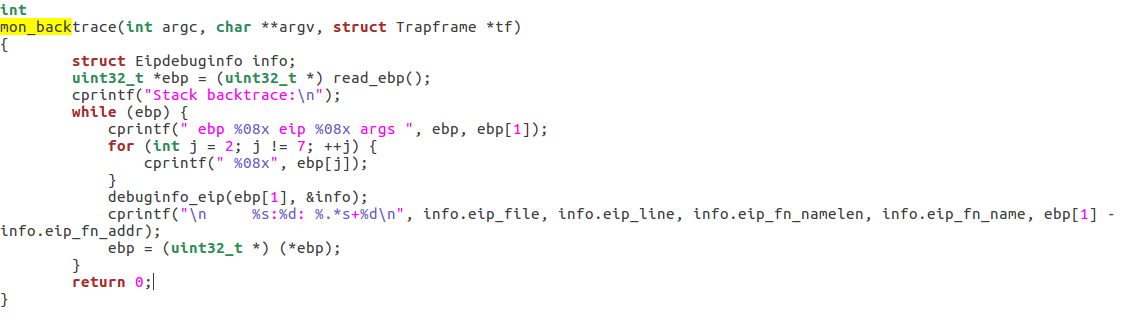


Exercise12：



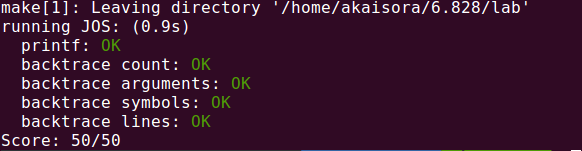
加入每层的函数名和地址信息。需要在kdebug.c中加入以下代码来找到函数信息，其中使用stab\_binsearch以查找地址的行号。，并添加mon\_backtrace中的输出。





### 实验结果

使用make grade测试代码编写是否正确



## Lab 5: File system, Spawn and Shell

### 实验目的

在本实验中，我们将实现spawn一个加载和运行磁盘可执行文件的库调用。然后充实我们的内核和库操作系统，在控制台上运行一个shell。这些功能需要一个文件系统，本实验将介绍一个简单的读/写文件系统。

### 实验内容

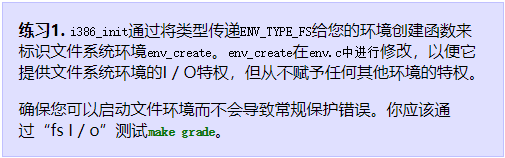
#### 第一部分：文件系统实现

实验指导书首先简单介绍了一般文件系统的结构，包括扇区、块、超级块、块位图、文件元数据和目录的概念。在jos中不会实现一个真实的文件系统的全部功能。我们实现的操作系统不会包含权限的限制，默认是单用户操作系统。组织文件时不使用inode，而是直接使用元数据。为了存储更大的文件将使用一层间接块。

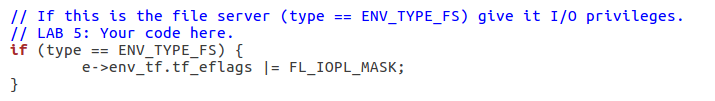
1. 磁盘访问

当前jos操作系统中的文件系统环境需要能够访问磁盘，但我们还没有在内核中实现任何磁盘访问功能。除了采用传统的“单块”操作系统策略，即将IDE磁盘驱动程序与必要的系统调用一起添加到内核中以允许文件系统访问它之外，我们还将IDE磁盘驱动程序作为用户级文件的一部分系统环境。我们仍然需要稍微修改内核，以便进行设置，以便文件系统环境具有实现本身磁盘访问所需的权限。”

Exercise1

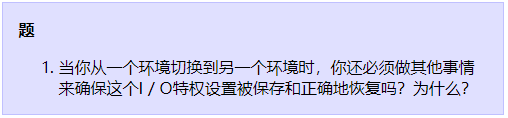


首先要打开文件系统访问I/O的权限，在创建进程时判断如果是文件系统的进程，那么给与I/O访问权限。

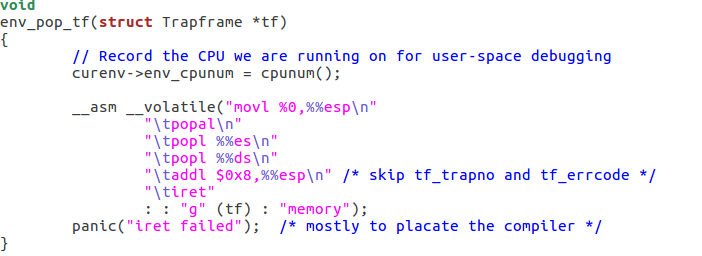




问题1：



回答：不用，切换进程时调用的pop\_tf函数中的iret指令恢复了包括eflags的寄存器。



1. 块缓存

Jos中我们通过缓存技术来实现访问磁盘块。该机制可以支持最大3GB的磁盘。

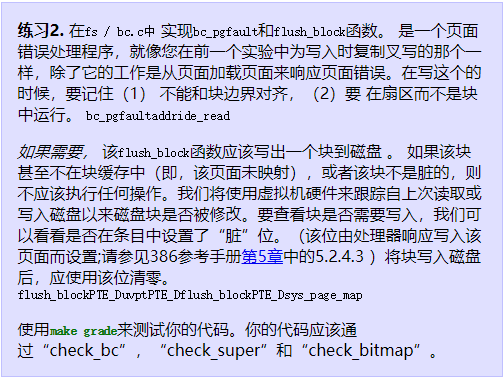
我们将文件系统服务进程的虚拟地址空间(0x10000000 (DISKMAP)到0xD0000000 (DISKMAP+DISKMAX))对应到磁盘的地址空间(3GB)。

由于现代磁盘大于3GB，在32位机器上的真正的文件系统实现会很麻烦。这种缓冲区高速缓存管理方法在64位地址空间的机器上仍然是合理的。

映射方法：我们假装整个磁盘都已经被缓存到内存中，当我们想访问虚拟空间中的一个页时，由于虚拟空间还没有被映射，会发生页错误。在页错误处理程序中则会实际进行磁盘块到虚拟地址的映射并将该块磁盘内容缓存到内存中。

此时就可以恢复文件系统进程进行正常的文件访问。

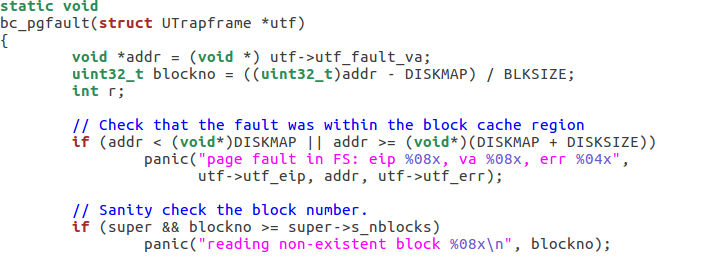
Exercise2

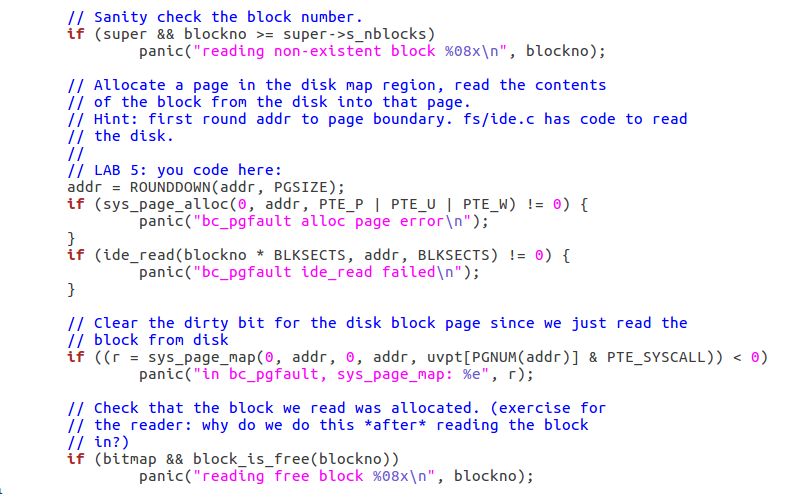


Bc\_pgfault函数负责处理页面错误，处理的同时进行页面映射并从磁盘中缓存对应的块到内存中。

处理的步骤：

1. 根据地址计算出对应的blockno（块编号）
2. 检查地址范围是否超出(0x10000000 (DISKMAP)到0xD0000000 (DISKMAP+DISKMAX))的映射边界
3. 检查块编号是否超出块数
4. 将地址对齐到页起始位置
5. 页面映射
6. 调用ide\_read函数读取缓存块对应的数个磁盘页到内存中
7. 更新脏位

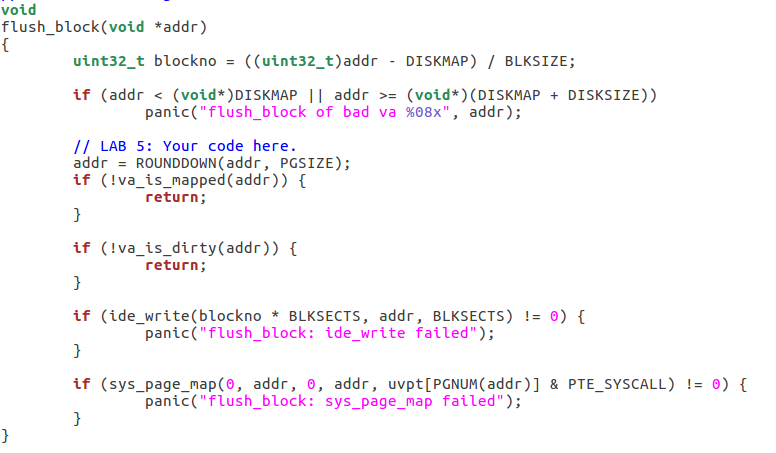




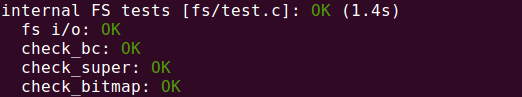
Flush\_block函数负责将缓存中的块写回到磁盘中。

处理步骤：

1. 根据地址计算出对应的blockno（块编号）
2. 检查地址范围是否超出(0x10000000 (DISKMAP)到0xD0000000 (DISKMAP+DISKMAX))的映射边界
3. 检查如果对应块没有映射就不写回
4. 检查如果对应块不是脏块（没有发生改动）就不写回
5. 最后将缓存写回到磁盘，检查是否成功并更新脏位



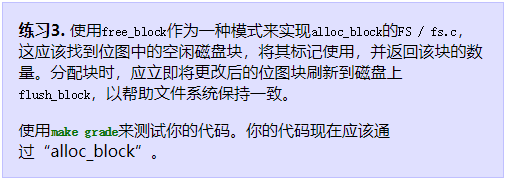
通过make grade检查是否成功



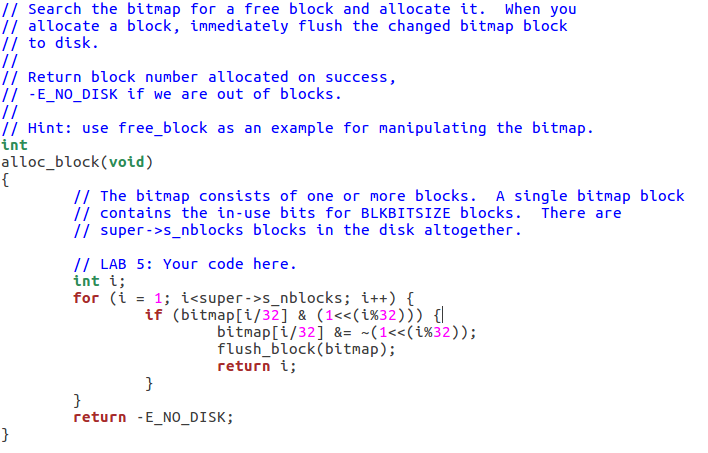
1. 块分配表

Block bitmap记录磁盘上的一块是否写有内容。在fs\_init设置bitmap指针之后，我们可以将其bitmap视为一个打包的位数组，每个磁盘块对应一个位。我们可以使用block\_is\_free函数检查给定块在位图中是否被标记空的。

Exercise3



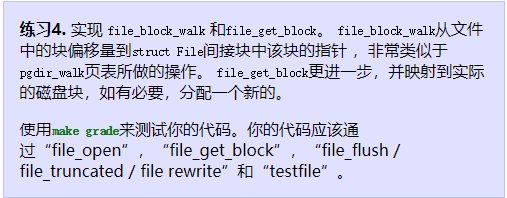
依次遍历所有bitmap位，当发现bitmap的第i位为1（表示能用）时将此位赋0并返回块号i。



1. 文件操作

在fs / fs.c中提供了各种函数来实现管理File的基本结构，扫描和管理目录文件的条目，从根目录文件系统来解析绝对路径名。

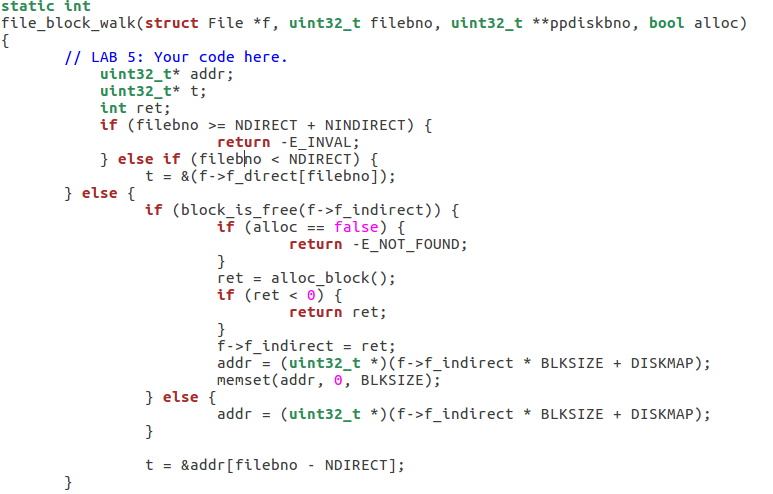
Exercise4



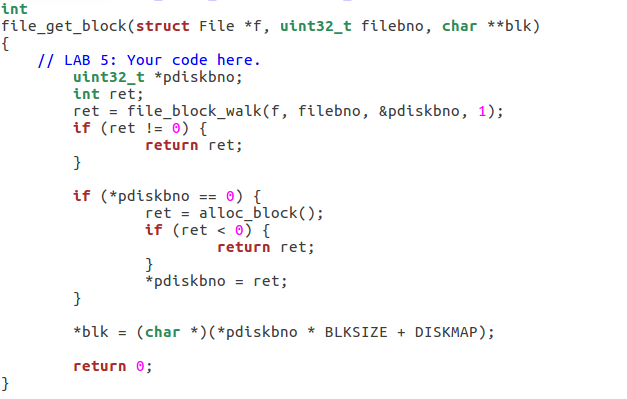
此步需要实现实现 file\_block\_walk 和file\_get\_block函数。

Fs.c文件中编写了与文件操作有关的一系列操作，例如浏览目录，解析路径等。

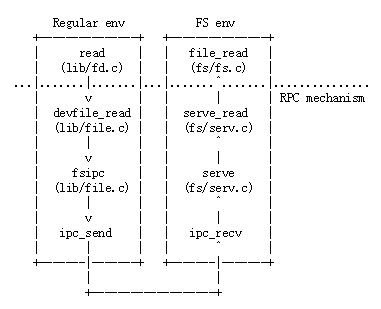
file\_block\_walk函数寻找一个文件结构f中的第fileno个块指向的磁盘块编号放入ppdiskbno。类似于遍历目录结构。



file\_get\_block函数先调用file\_walk\_block函数找到文件中的目标块，然后将其转换为地址空间中的地址赋值给blk。



1. 文件系统接口



实现了文件系统内的函数后，我们需要向外界提供更方便的统一接口。

虚线下面的所有内容都是从常规环境到文件系统环境的读取请求的机制。read（提供的接口）在任何文件描述符上工作，并简单地调度到适当的设备读取功能，在这种情况下 devfile\_read（我们可以有更多的设备类型，如管道）。 devfile\_read 实现read专门针对磁盘上的文件。这个和lib / file.c中的其他devfile\_\*函数 实现了FS操作的客户端，并且都以大致相同的方式工作，将请求结构中的参数捆绑在一起，调用 发送IPC请求，解包并返回结果。该fsipc 函数只是处理向服务器发送请求和接收答复的常见细节。

文件系统服务器代码可以在fs / serv.c中找到。它在serve函数中循环通过IPC接收请求，将该请求分派给相应的处理函数，并通过IPC发回结果。在读取的例子中， serve将调度到serve\_read，将处理特定的IPC细节读取请求，如解开请求结构，最后调用 file\_read实际执行文件读取。

JOS的IPC机制让一个环境发送一个单一的32位数字，并可选择共享一个页面。为了将请求从客户端发送到服务器，我们使用32位数字作为请求类型（文件系统服务器的RPC编号，就像syscalls的编号一样）并将参数存储union Fsipc在页面上的请求中 通过IPC共享。在客户端，我们总是共享页面fsipcbuf; 在服务器端，我们在fsreq （0x0ffff000）处映射传入的请求页面。

服务器也通过IPC发回响应。我们使用32位数字作为函数的返回码。对于大多数RPC，这是他们所有的返回值。 FSREQ\_READ并FSREQ\_STAT返回数据，他们只是写入客户端发送请求的页面。在响应IPC中不需要发送此页面，因为客户端首先将其与文件系统服务器共享。此外，在答复中，FSREQ\_OPEN与客户分享一个新的“Fd页”。我们很快会返回到文件描述符页面。

观察服务端主循环轮询时处理文件操作的过程：

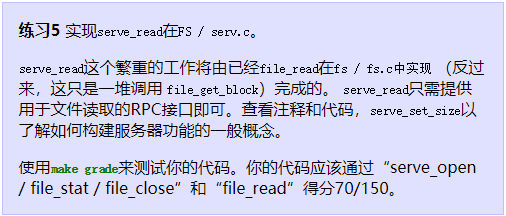
1、从IPC接受1个请求类型req以及数据页fsreq

2、然后根据req来执行相应的服务端处理函数

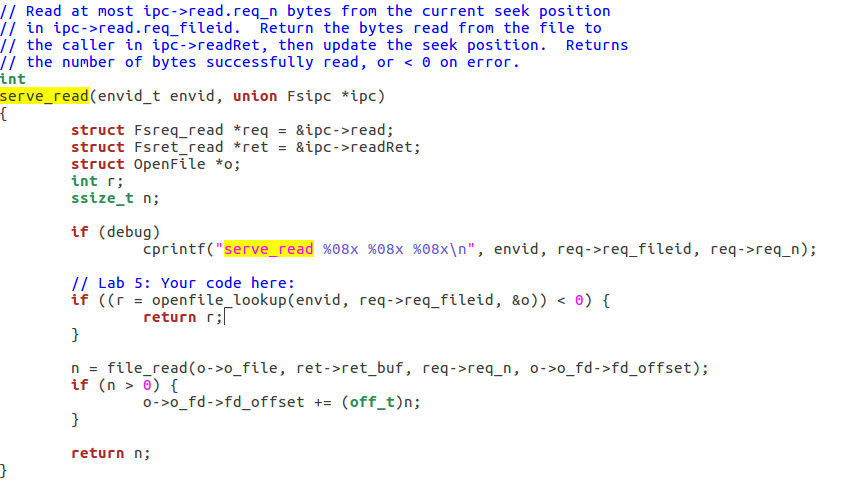
3、将相应服务端函数的执行结果(如果产生了数据也则有pg)通过IPC发送回调用进程

4、将映射好的物理页fsreq取消映射

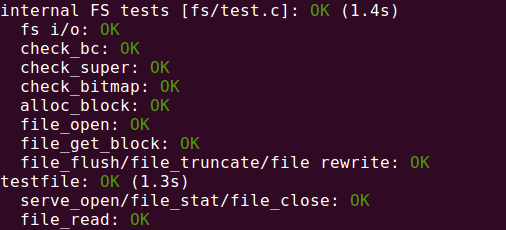
Exercise5



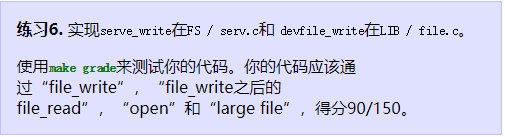
Serve\_read函数用于读文件请求。从ipc结构体中获取需要读取的目标信息，调用file\_read函数读取文件放置在ret\_buf中。



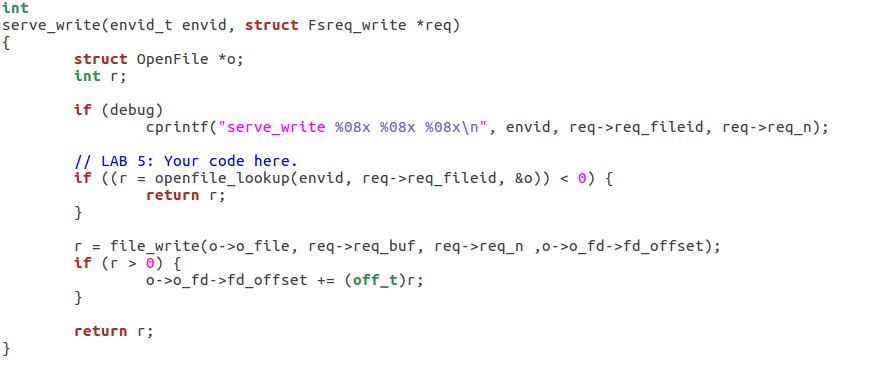
Make grade测试：



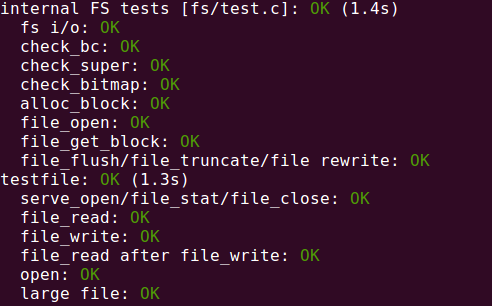
Exercise6



serve\_write同样是获取请求信息，将req\_buf中的文件内容写入到磁盘块。



Devfile\_write函数主要功能是将写文件需求通过fsipc打包成请求以便于发送给文件服务系统。

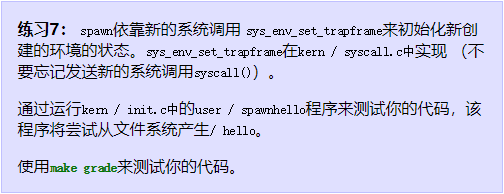


#### 第二部分：Spawning过程

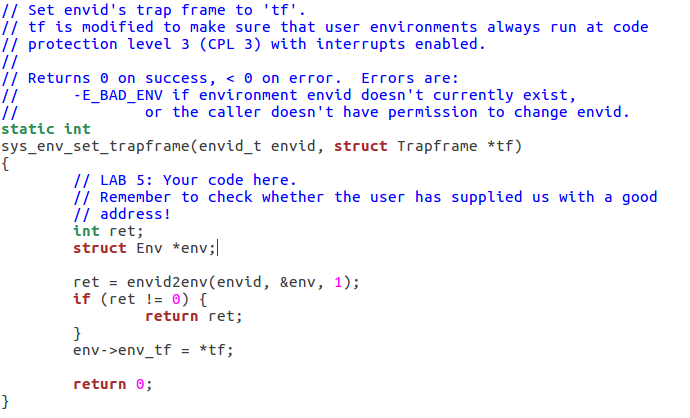
1. 环境继承

Spawn函数可以从父进程中调用，装载一个新的程序执行，类似于UNIX的fork后直接exec装载新的程序。

Exercise7



sys\_env\_set\_trapframe将父进程的环境（寄存器）复制为子进程的环境。



加入系统调用



使用make run-spawnhello可以看到父进程的信息和子进程的信息都输出了，输出中间的fs文件系统检查内容是由于多进程交替执行输出到控制台的结果。



1. 在fork和spawn之间共享库状态

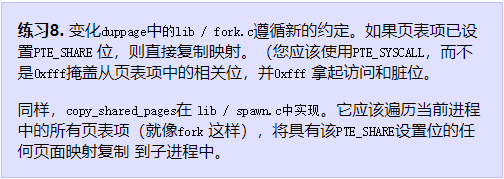
UNIX文件描述符包括pipe，console I/O。 在JOS中，这些设备类型都有1个与与它关联的struct Dev，里面有实现read/write等文件操作的函数指针。在lib/fd.c中实现了传统UNIX的文件描述符接口。

　　在lib/fd.c中也包括每个客户进程的文件描述符表布局，开始于FSTABLE。这块空间为每个描述符保留了1个页的地址空间。 在任何时候，只有当文件描述符在使用中才在文件描述符表中映射页。

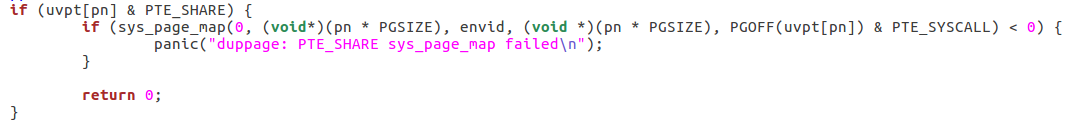
　　我们想要共享文件描述符状态在调用fork和spawn创建新进程。当下，fork函数使用COW会将状态复制1份而不是共享。在spawn中，状态则不会被拷贝而是完全舍弃。

　　所以我们将改变fork来共享状态。在inc/lib.h中新定义了PTE\_SHARE位来标识页共享。当页表入口中设置了该位，则在fork和spawn时应该从父进程中拷贝PTE映射到子进程。

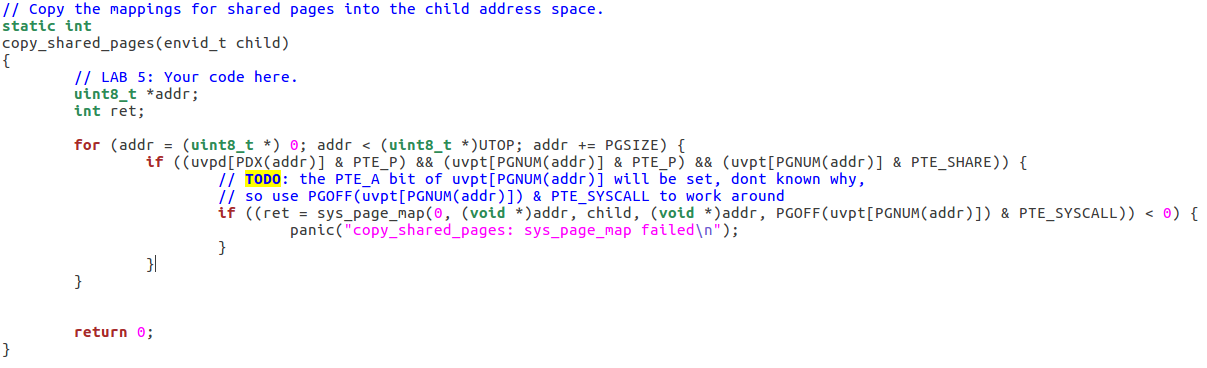
Exercise8



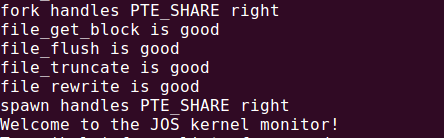
在duppage中添加对PTE\_SHARE位进行判断，若有PTE\_SHARE位则直接复制映射。

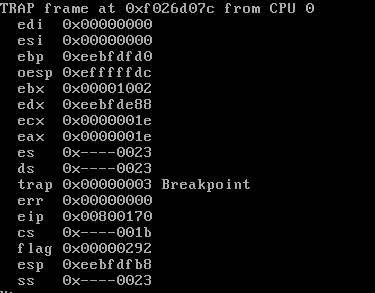


在copy\_shared\_pages中扫描进程中所有页，发现PTE\_SHARE标记时，调用sys\_page\_map进行映射复制。

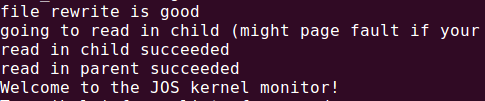


运行make run-testpteshare，可以看到"fork handles PTE\_SHARE right" 和 "spawn handles PTE\_SHARE right"





运行make run-testfdsharing，可以看到"read in child succeeded" 和 "read in parent succeeded"

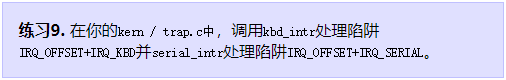


#### 第三部分：Shell

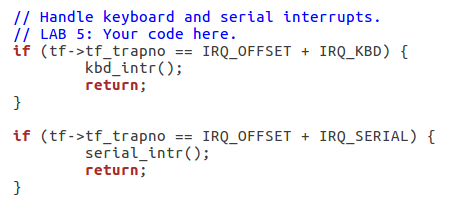
1. 键盘接口

为了使shell工作，我们需要一种方法来输入它。QEMU一直在显示我们写到CGA显示器和串行端口的输出，但到目前为止，我们只在内核监视器上进行输入。在QEMU中，输入在图形窗口中的输入显示为从键盘到JOS的输入，而输入到控制台的输入在串行端口上显示为字符。 kern / console.c已经包含了自实验1以来已被内核监视器使用的键盘和串口驱动程序，但是现在我们需要将这些驱动程序附加到系统的其他部分。

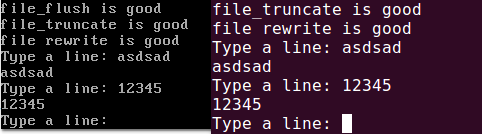
Exercise9



在trap.c中添加对于IRQ\_OFFSET+IRQ\_KBD和IRQ\_OFFSET+IRQ\_SERIAL的判断，分别调用kbd\_intr和serial\_intr进行处理。



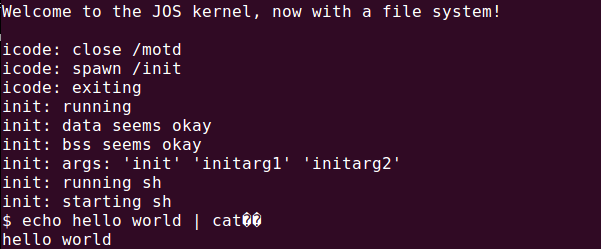
运行make run-testkbd测试：



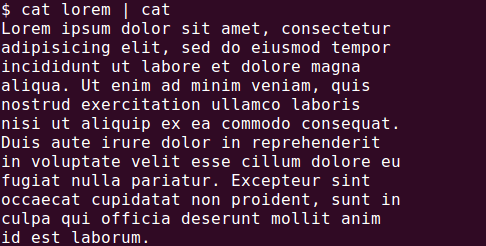
1. Shell命令

首先运行make run-icode测试

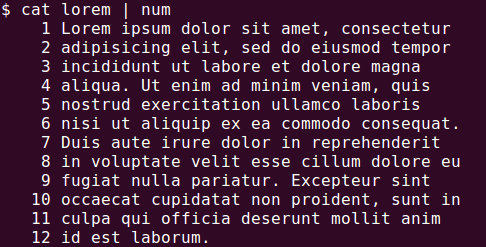
echo hello world | cat 将hello world文本传送给cat程序，cat程序用来显示文本



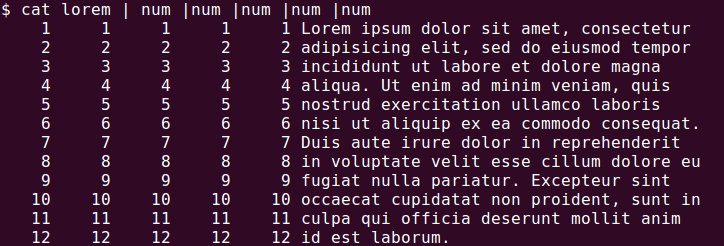
cat lorem |cat 显示文本再传送给cat



cat lorem |num 给文本加上行号



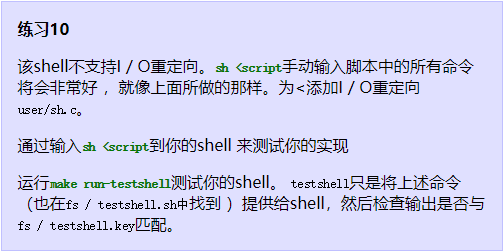
cat lorem |num |num |num |num |num将加上行号的文本作为新文本再添加行号，多次迭代



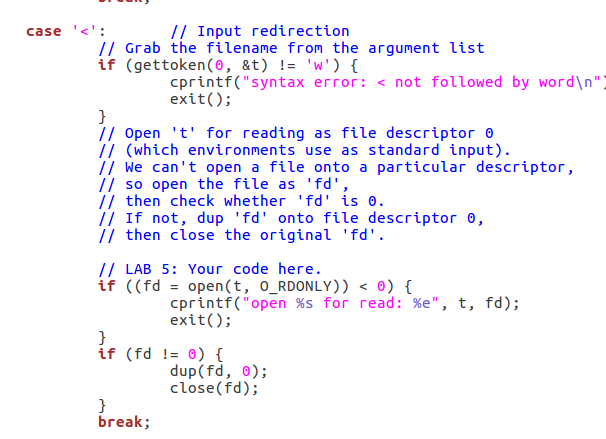
lsfd



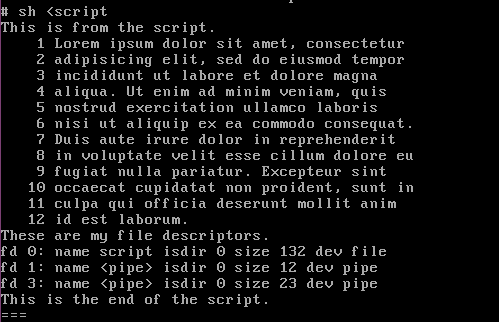
Exercise10



要求我们实现shell的重定向特性，使用open()打开文件来获得命令输入。



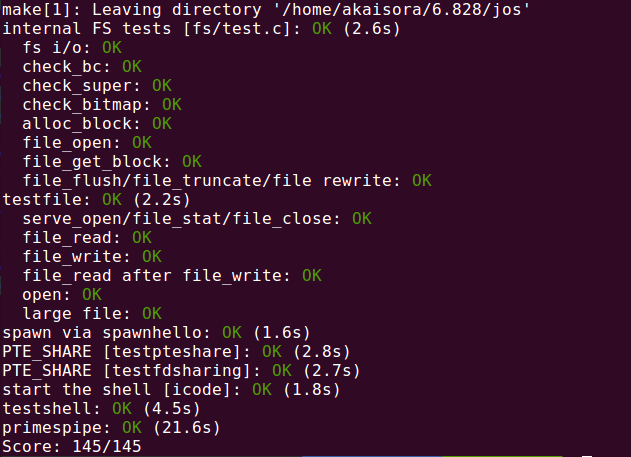
运行make run-testshell来测试shell功能,成功重定向



### 实验结果

运行make grade进行测试。

Lab5：



# 四、xv6实验系统分析

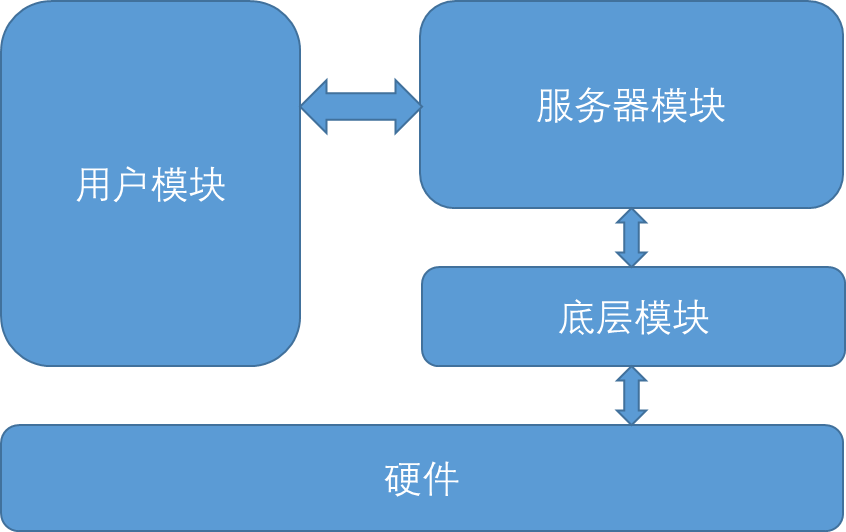
## xv6的系统结构（结合示意图）

## （二）xv6的引导

## （三）xv6的进程与调度分析

## （四）xv6的内存管理

## （五）xv6的文件系统



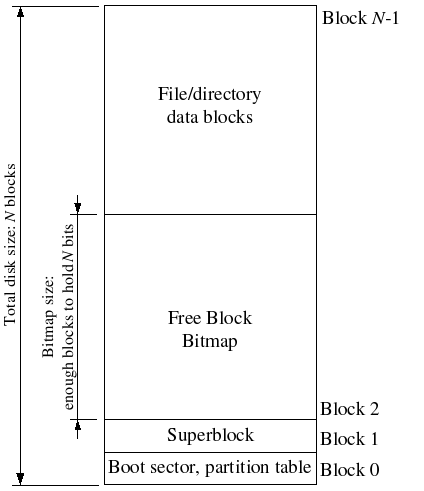
文件系统架构总体上分为3大模块，分别是用户模块，服务器模块，底层模块。

服务器模块负责打包底层模块文件读写函数，为外界提供统一的文件读写接口。

底层模块负责详细的页面划分，加载，缓存以及控制磁盘驱动。

用户模块与文件服务系统进程通信来进行文件读写活动。

1. 磁盘结构

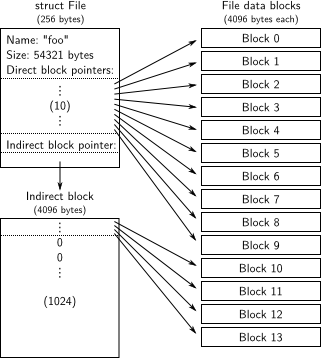
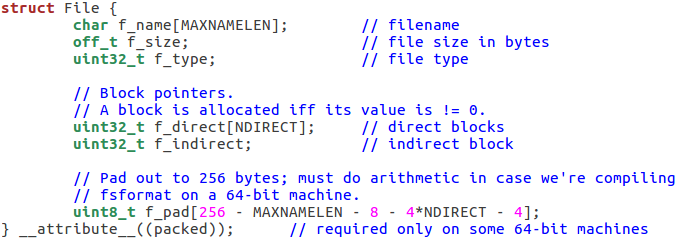


磁盘扇区大小512字节，而jos在此之上组织的块大小为4kB。与Linux不同的是此文件不使用inode，而是直接存储文件元数据，所以也不支持硬链接。

磁盘上文件结构是

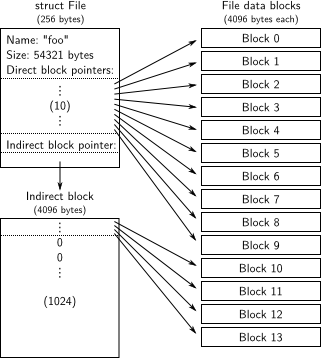
* Block0 引导扇区，分区表
* Block1 超级块
* Block2~k Bitmap块，存储磁盘块占用信息
* Blockk+1~N-1 数据块

1. 文件元数据



文件元数据使用file结构体存储，包含文件名，文件大小，10个直接块，一个间接块。文件的前10个block号会存在直接块里，其他的block会存在间接块指向的block里，可以额外存储4096/4=1024个文件块。

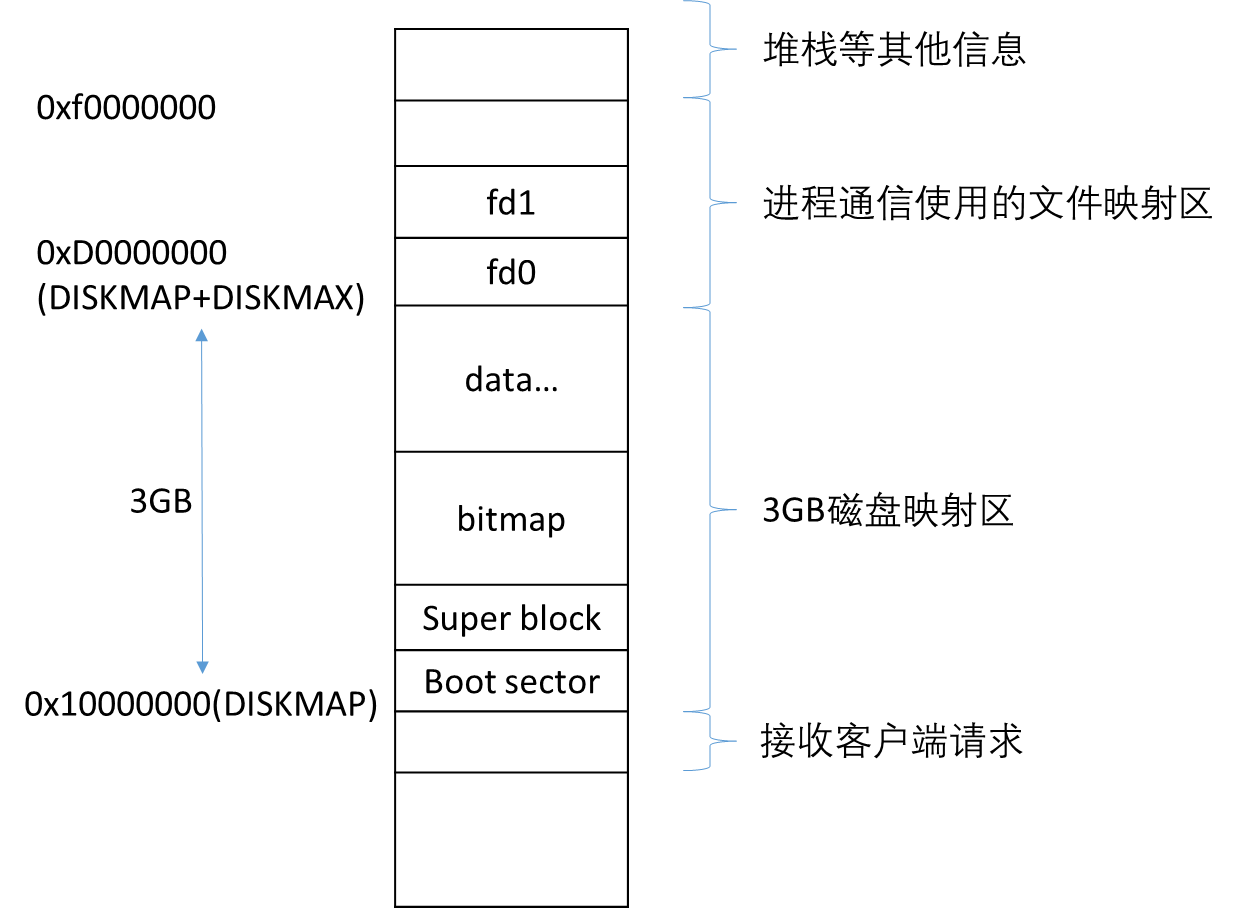
此文件系统不支持双重间接块和三重间接块，所以文件不能大于1034个文件块。



1. 目录与普通文件

文件系统以相同的方式管理目录文件与普通文件，以f\_type区分，当文件块描述目录时，关联数据块内容为目录下的其他文件File数据块。

1. 块缓存



Jos中我们通过缓存技术来实现访问磁盘块。该机制可以支持最大3GB的磁盘。

我们将文件系统服务进程的虚拟地址空间(0x10000000 (DISKMAP)到0xD0000000 (DISKMAP+DISKMAX))对应到磁盘的地址空间(3GB)。

由于现代磁盘大于3GB，在32位机器上的真正的文件系统实现会很麻烦。这种缓冲区高速缓存管理方法在64位地址空间的机器上仍然是合理的。

映射方法：我们假装整个磁盘都已经被缓存到内存中，当我们想访问虚拟空间中的一个页时，由于虚拟空间还没有被映射，会发生页错误。在页错误处理程序中则会实际进行磁盘块到虚拟地址的映射并将该块磁盘内容缓存到内存中。

此时就可以恢复文件系统进程进行正常的文件访问。

Bc\_pgfault函数负责处理页面错误，处理的同时进行页面映射并从磁盘中缓存对应的块到内存中。

处理的步骤：

1. 根据地址计算出对应的blockno（块编号）

2. 检查地址范围是否超出(0x10000000 (DISKMAP)到0xD0000000 (DISKMAP+DISKMAX))的映射边界

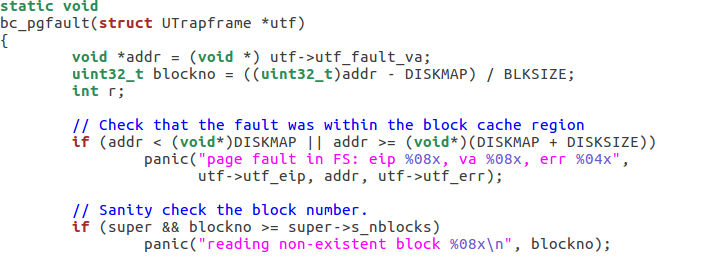
3. 检查块编号是否超出块数

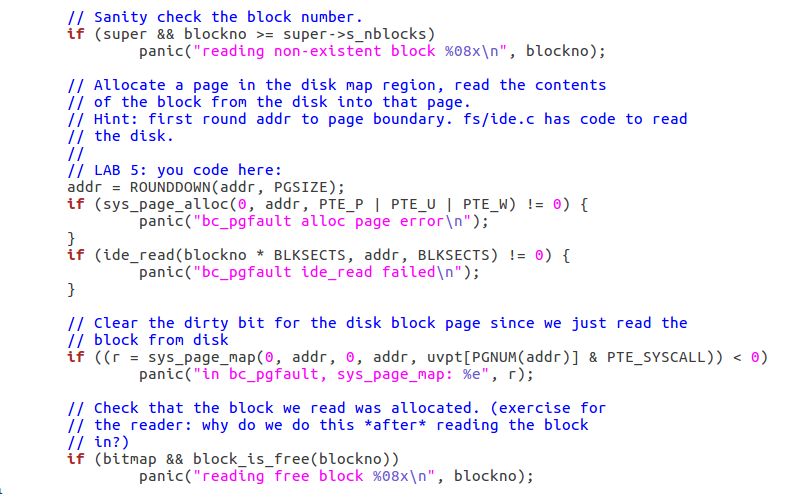
4. 将地址对齐到页起始位置

5. 页面映射

6. 调用ide\_read函数读取缓存块对应的数个磁盘页到内存中

7. 更新脏位





Flush\_block函数负责将缓存中的块写回到磁盘中。

处理步骤：

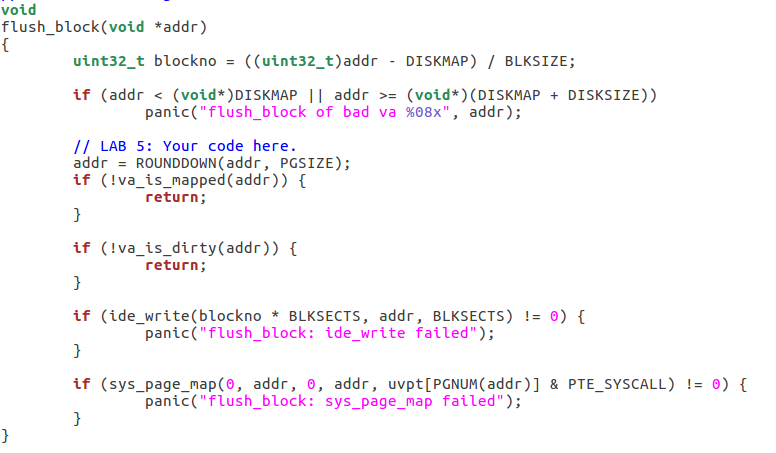
1. 根据地址计算出对应的blockno（块编号）

2. 检查地址范围是否超出(0x10000000 (DISKMAP)到0xD0000000 (DISKMAP+DISKMAX))的映射边界

3. 检查如果对应块没有映射就不写回

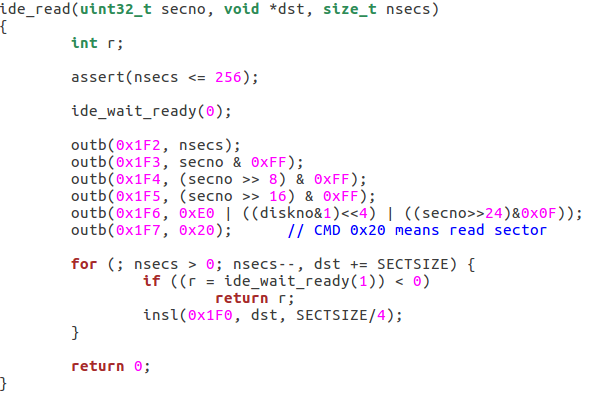
4. 检查如果对应块不是脏块（没有发生改动）就不写回

5. 最后将缓存写回到磁盘，检查是否成功并更新脏位



1. 磁盘ide驱动文件

Fs/ide.c中实现了ide\_read 和 ide\_write方法



1. 文件操作

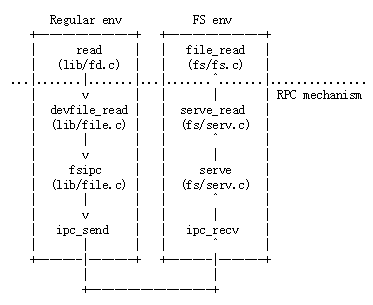
fs / fs.c中提供了各种函数 来实现您将需要解释和管理File结构，扫描和管理目录文件的条目，从根目录文件系统来解析绝对路径名。有一下几个主要函数：

* Walk\_path 完成目录解析
* File\_block\_walk 找到一个文件的某一块
* file\_get\_block 获取文件某一块
* file\_read 读取某一个文件
* file\_write 写入某一个文件

1. 文件系统服务器

与启动引导时把文件读写作为系统调用不同，而是作为一个用户进程于是我们在创建文件读写进程时需要把I/O读写权限打开

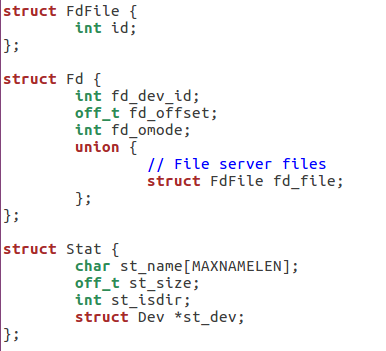
文件服务器进程采用轮询的方式接收用户进程发送的文件读写请求，向用户进程返回信息



虚线下面的所有内容都是从常规环境到文件系统环境的读取请求的机制。从头开始，read（我们提供的）在任何文件描述符上工作，并简单地调度到适当的设备读取功能，在这种情况下 devfile\_read（我们可以有更多的设备类型，如管道）。 devfile\_read 实现read专门针对磁盘上的文件。这个和lib / file.c中的其他devfile\_\*函数 实现了FS操作的客户端，并且都以大致相同的方式工作，将请求结构中的参数捆绑在一起，调用 发送IPC请求，解包并返回结果。该fsipc 函数只是处理向服务器发送请求和接收答复的常见细节。

文件系统服务器代码可以在fs / serv.c中找到。它在serve函数中循环，无休止地通过IPC接收请求，将该请求分派给相应的处理函数，并通过IPC发回结果。在读取的例子中， serve将调度到serve\_read，将处理特定的IPC细节读取请求，如解开请求结构，最后调用 file\_read实际执行文件读取。

1. 用户使用的fd文件描述符



## （六）xv6 的I/O

# 五、总结与心得

（做了这些实验之后有哪些收获；觉得MIT-XV6系统有哪些地方值得改进；对操作系统实验有哪些建议）