

華中科技大学

课程设计报告

课程名称: 系统系统综合训练

专业班级: ACM2012

学 号: U201214997

姓 名: 姚信

指导教师: 邵志远老师

报告日期: 2015/3/28

目 录

1、 Jos Summary.....	3
2、 Jos 运行过程整体说明.....	8
3、 LAB1:Booting a PC.....	12
4、 LAB2:Memory Management.....	22
5、 LAB3:User Environments.....	36
6、 LAB4:Preemptive Multitasking.....	51
7、 LAB5:File Systems and Spawn.....	63

辅助材料：

[1] <http://grid.hust.edu.cn/zyshao/OSEngineering.htm>

邵志远老师主页 操作系统 v0.1 系列书稿

[2] <http://pdos.csail.mit.edu/6.828/2007/>

MIT 2007 年版原版习题指引

[3] 卓达城师兄系列 lab 报告

[4] 北大版张弛 Jos 实习报告

JOS Summary

它分为六个实验:

- 1.启动.
- 2.内存管理.
- 3.用户环境
- 4.可抢占多任务
- 5.文件系统
- 6.A shell

Lab 1: Booting a PC.

计算机启动->BIOS->引导启动程序->内核，也可见[4].

引导启动程序主要干两件事:

- 1.将处理器从实模式转换到 32 位保护模式. (boot/boot.S)
- 2.从硬盘读内核，将控制权转给内核. (boot/main.c)

内核获得控制之后，由于代码还在低地址运行需要开启分页机制，然后通过间接调用跳到

高地址的 C 初始化代码. (kern/entry.S, kern/entrypgdir.c)

内核初始化(kern/init.c):

- 1)console: keyboard/vga/serial. (kern/console.c)
- 2)目前直接 monitor.(kern/monitor.c) (后序实验初始化，如内存，文件系统等)

Challenges:

- 1) Enhance the console, see [3] "字符设备驱动程序"一章.

Summary: It took me much time to review not-so-familiar knowledge, i.e. assembly instructions, 80x86 registers, and gdb.

Lab 2: Memory Management.

Part 1: 物理内存管理. (kern/pmap.c)

- 1)页面的初始化.
- 2)页面的分配和回收.

Part 2: 虚拟内存. (kern/pmap.c)

- 1)虚拟地址/线性地址/物理地址三者之间的转换.
- 2)页表的管理，插入和删除页面时页目录和页表的更新.

Part 3: 内核地址空间

- 1) JOS 内存 layout. (inc/memlayout.h)
- 2) 内核的保护. (页表项, 段选择符)

Challenges:

- 1) Reducing pages for KERNBASE mapping, see [2](entrypgdir in main.c).
- 2) Malloc/free, see [2](umalloc.c) and [4](lib/malloc.c).

Summary: I'm more clear about these three addresses and memory layout/allocating/freeing, etc.

Lab 3: User Environments.

Part A: 用户环境和异常处理

JOS 的用户环境类似于 Unix 进程. (kern/env.c)

- 1) 内核需要为用户环境保存的信息. (env_id 含有个有趣的部分)
- 2) 分配用户环境空间, 创建和运行用户环境. (由于没有文件系统, JOS 嵌入 binary 到 kernel)

异常处理, 设置中断描述表 IDT. (kern/trap.c, kern/trapentry.S)

Part B: 页故障, 断点和系统调用

页故障, CR2. (kern/trap.c)

断点, 主要用来 user-mode panic. (kern/trap.c)

系统调用. (kern/syscall.c)

- 1) 参数的传递是通过 5 个 registers(edx, ecx, ebx, edi, esi), eax 保存系统调用号.

Challenges:

- 1) Clean trapentry.S up, we can script it, see [4](vectors.pl).
- 2) Support 'continue', set TF and RF of EFLAGS, then go back user.
- 3) Sysenter/sysexit, see [5].

Summary: Trap, exception, and syscall; kernel protection; kernel/user address space switching.

Lab 4: Preemptive multitasking

Part A: 合作式多任务

- 1) BSP(bootstrap processor) 激活 APs(application processors)
- 2) Per-CPU 状态和初始化. (Kernel stack, tss 和 tss 描述符, 当前 env, registers,

idle env)

Scheduling(kern/sched.c), Round-Robin.

System calls for 创建用户环境: (注意权限和地址的合法性判断)

- 1) sys_exofork, 类似 Unix fork, 创建新用户环境, child 被标记不可运行.
- 2) sys_env_set_status, 设置可运行或不可运行.
- 3) sys_page_alloc, 分配新页面.
- 4) sys_page_map, 复制一页面的映射.
- 5) sys_page_unmap, unmap 一页面.

Part B: Copy-on-Write Fork

用户级页错误处理函数:

- 1) 设置页错误处理函数.
- 2) 异常栈(Exception stack).
- 3) 调用页错误处理函数.
- 4) Entry point. ## 有点难且有趣

实现 CoW Fork: ## 难点: 理解 vpt/vpd

- 1) 设置页错误处理函数.
- 2) sys_exofork 创建新 env.
- 3) 为父/子 env 设置 cow-or-writable 页面为 cow, 除了异常栈.
- 4) 为子 env 设置页错误处理函数.
- 5) 标记子 env 可运行.

Part C: IPC

- 1) send/receive.
- 2) lib wrapper.

令我十分吃惊的是 primes.c, 它用一种新颖的思路实现 prime sieve.

这部分 Challenges 十分有趣, 花了我很多时间, 尤其是 power-series.
Challenges:

- 1) Drop big kernel lock, see [2] and [4].
- 2) Add a scheduling policy, well, todo.
- 3) SFORK, easy but heed the global ``thisenv" pointer.
- 4) Un-loop ipc_send, the only way I can occur is the block-wait.
- 5) Matrix multiplication, using sfork.
- 6) Power series, GOOD ONE, also see [6](Python implementation).
- 7) "Improving IPC by Kernel Design", so many tricks :-), todo.

Summary: Other CPUs are activated by BSP; Locking and Scheduling both are big chunks; COW fork is easy to say, but seem tricky to implement; Page Fault is also tricky; IPC...

=====

Lab 5: File System

主要工作是看代码.

文件系统在磁盘上数据结构:

- 1) 区(Sector)与块(Block)
- 2) 超级块(Superblock)
- 3) 块位图(block bitmap)
- 4) 文件元数据(File meta-data)

文件系统组成部分:

- 1) 磁盘访问, see also [4]'s "块设备驱动程序" 一节.
- 2) 块缓存的管理.
- 3) 位图的管理.
- 4) 文件操作.

该文件系统是由一个专门 env 来实现, JOS 通过 IPC 的方式让其它 env 来使用该文件系统.

Spawning processes

这段代码还挺有意思.
##

Challenges:

- 1) Crash-resilient, see [2]
- 2) Inode, see [2]
- 3) Unix-style exec, tricky as the kernel is needed to do the entire replacement.
- 4) Mmap-style, no that hard.

Summary: Delving into this simple FS, now I understand FS more clearly.

=====

Lab 6: Final Project

- 1) 共享库代码, PTE_COW.
- 2) 键盘接口, see [3].
- 3) Shell, see also [2].

Challenges:

- 1) The project, 我还不够优秀来完成这个 Project, 没有足够的想法。

Summary: This lab except the challenge requires the least code. If I have much time, I would want to see the bash source.

Needed requirements:

- 1) OS concept
- 2) 80x86 concept
- 2) Assembly(gas)/C/Inline_assembler
- 3) GDB
- 4) ELF, part.

Optional requirements:

- 1) Hardware programming, keyboard/console/disk/network etc.
- 2) Make, Makefile of course.
- 3) Shell, grade test for lab1-5.
- 4) Python, grade test for lab6-7. ## It took me 1/2 day to understand, and another 1/2 to deal with the qemu-bug.
- 5) Qemu, well I didn't dive into it.
- 6) Gcc/ld, ditto.

Tools I use:

well, it could be better to write another article.

- 1) Vim. You need a damn great editor and make full use of it. For vim, I heavily use cscope, marks, tabs, global/substitute, etc, and good plugins[8].
- 2) Tmux. A terminal multiplexer (Before I used GNU screen).

Xv6

阅读[2]给的书和源码.

Linux-0.11

阅读[3]和 linux-0.11 源码.

引用: ## JOS 主页本身含有大量好资源.

- [1] MIT-JOS, <http://pdos.csail.mit.edu/6.828/2011/overview.html>
- [2] Xv6, <http://pdos.csail.mit.edu/6.828/2011/xv6.html>
- [3] Linux0.11, <<linux 内核完全剖析>> 赵炯
- [4] Booting, <http://www.ruanyifeng.com/blog/2013/02/booting.html>
- [5] Sysenter, http://articles.manugarg.com/systemcallinlinux2_6.html
- [6] Power-series, <https://github.com/pdonis/powerseries>
- [7] Zero-copy, <http://www.ibm.com/developerworks/library/j-zerocopy/index.html>
- [8] Vim-good-plugins, <https://github.com/carlhuda/janus>

Jos 运行过程整体说明

1、BIOS：自检硬件，加载 boot loader

2、boot loader 为 boot.S 和 main.c，两个文件

开始于 0x7c00，该地址也是 Makefrag 决定的

首先是 boot.S，它主要是为代码段、数据段等一些段寄存器初始化，在将 GDT 表的首地址加载到 GDTR 当中，然后从实模式转换到保护模式。在做一个长跳转的时候，对 CS 进行了一次隐性修改，然后对一些基础段寄存器赋值，并转到 main.c 的 bootmain。如果从 bootmain 退出，该 Jos 系统会进入一个 spin 的死循环。

main.c，ELFHDR 是 kernel 开始的地方，但是这个值是随意定的，它是一个 **魔数**。该文件被放在磁盘的第一个扇区，方便启动的引导。然后第二个扇区存放内核映像文件，该 bootmain 主要就是用来加载内核到内存 0x100000 处。最后通过执行 e_entry 与 FFFFFF 相与位置(这里主要作用是将最前面的一个 F 去掉)的一个函数，将控制权交给内核。

3、内核启动

首先在编译的时候有 kernel.ld 决定了文件编译后的虚拟地址位置和入口点。入口为 entry.S 中的 _start，这里几步做的工作与 boot.S 做的很类似，只是它加载了新的段，新段的基址从-KERNBASE 开始。SEG 的作用是对段的属性进行初始化。

然后我们可以在 kernel.ld 当中找到 edata 和 end 的定义。edata 和 end 之间的内容都是 .bss 的内容。在 i386_init 一开始便对其进行了初始化。

进入 cons_init，这个事对整个屏幕、键盘、串口的初始化，包括 cga、kbd、serial。

这里对 monitor 最好要有一个了解，最好能够自己动手实现以下

然后开始检查内存。

4、内存分页

首先自检内存空间，对比在 memlayout.h 当中的内存分布，我们知道 i368_detect_memory 对基本内存和扩展内存进行了区分，确定了页的数量，并打印相关信息。

然后回到 i386_init，进行 i386_vm_init 的调用，这时 Jos 还是没有开启页式转换，物理地址和线性地址就是 pa = la + base 的关系。pgdir 是页目录首指针，然后使用 boot_alloc 分配了一页的空间给 pgdir 做页目录记录。然后传值给全局变量 boot_pgdir，并使用 boot_cr3 记录它的物理地址。后面两句紧接着的是设置 VPT 和 UVPT 的权限值，**VPT 只允许系统读写，UVPT 只允许用户读**。两者的内容实际上是一样的。

然后就开始 boot_alloc 一个 npage 个 page 结构的空间，用来存放各个页的属性，这里每次分配的空间需要尽可能 4K 对齐。

然后就是为 PCB 空闲链表分配一个区域，大小由 NENV 决定，同样要对齐。

该分配的分配完后，我们就开始对内存也进行初始化 page_init。这个初始化的工作，主要是制作出空闲页链表。首先我们先将 npage 个空闲页链接起来，然后将已经使用了的页全部从该链表上删掉，删掉的时候主要使用了一些有趣的宏定义，并将 ref 加 1 标记。

之后进入了比较正式的 check，首先使用 page_alloc 分配了 3 个页，page_alloc 在没有空间页的时候返回负值 -4，如果有空闲页的话，就直接取下空闲队列第一个页，分配出去，并将第一页从队列里移除，并返回 0。明显的，我们三次分配的页指向的虚拟地址是不等的且不为 0。page2pa 主要是得到该页在 pages 的下标，它必须小于 npage。然后又把空闲页链表首指针给 f1，并且自己清零，然后再使用 page_alloc 的时候发生缺页现象。最后使用 page_free 清理所分配的页。page_free 就是 如果 pp_ref 有值，就不管，alloc 的时候是没有值的，所以要对页清空并回收到空闲队列当中。

然后我们回来了，马上又进入了 page_check 了。首先分配三个页，没有问题，又测试

了一次缺页现象。使用 lookup，在 boot_pgdir 中找一个空页，在 lookup 当中，我们使用到了 pgdir_walk，这个函数主要是根据虚拟地址给出其所在的页表所在的目录项，先判断是否有 PTE_P(当前存在)，如果有，就直接返回 PTE_ADDR(*pd)，*pd 的内容就是页表首地址的内容，而~0xFFFF 就是高 20 位都是 1，其余都是 0，这样就是取*pd 的前 20 位，即页表的首地址(因为页也是 4K 对齐的)。如果没有，且 create 不为 1，就直接返回 NULL(本来就找不到还不让创建)。如果可以创建，就是用 page_alloc 分配 pg 一个页，又因为 pgdir_walk 来找它了，说明有引用，故对 ref 置 1，将该页初始化，然后设置一下权限，就可以和刚刚一样返回页表首地址了。

首先就是判断该页表是否不为空值，或者现在是否有效，两者一个不满足就返回空值。然后如果引用的 pte_store 为空，就将页表的首地址给他作为值。最终返回物理页地址对应的页表首址给他。当前地址 0 还没有页与其对应，故返回值也应该为空。

然后直接尝试把 alloc 的页 pp1 与虚拟地址建立起映射关系。这里首先使用 pgdir_walk 以允许创建的方式查找页表首址，此时如果返回空，说明没有页表，再请求分配页表时，会请求分页一页保存页表，然后返回值表示没有多余的物理空间，即 page_alloc 失败了(因为此时 create 值为 1)，而 page_alloc 还是则说明 page_free_list 为 NULL，在原 page_check 中，我们可以看到，page_free_list 被 init 过了，即无法分配页是正常的。紧接着又对 pp0 进行 free，free 后此时的 page_free_list 当中就有了一个空页等待分配 pp0，此时在再次建立映射时，pp1 仍无页表，然后就分配一页保存页表，此时之前被 free 掉的页刚好就可以被分配拿来做页表用。PTE_ADDR 就是取前 20 位数据，实际上就是查页表的物理首地址，该地址同样也是 pp0 的地址，故知道刚刚建立的页表时 pp0，然后使用 check_va2pa 检查 insert 是否将虚拟地址和物理地址的映射关系建立好。然后就将 pp2 页 insert 一下，因为 pp0 已经是页表了，所以和之前一样进行了检查。又因为此时 page_free_list 里什么都没有，所以 page_alloc 再次失败。[然后对 pgdir_walk 进行了检查，看它是否返回页表项地址。](#)

然后对 pp2 进行权限修改，并检查了功能是正确的：即权限变为用户级，引用次数仍未 1。然后在想把 pp0 插入时出错，原因还是没有空闲页，再将 pp1 插入 PGSIZE 处，替换了原来的 pp2，此时 pp2 消失了，在对 pp1 进行 map 的时候会解除对之前 pp2 的 map 映射，并解除引用关系。别忘记，在 ref=1 时，会被 free 掉，于是现在 page_free_list 有了空闲页 pp2。这时使用 page_remove，这里只会将 ref--，如果 ref=0，才会 free，并在最后使用函数 tlb_invalidate，将 tlb 当中相关内容清除。此时由于 0 处没有相关数据，因此得到的结果为 ~0。后面再 PGSIZE 处做了相同的事情。这时 pp1 在 page_free_list 当中，使用 page_alloc 进行一次检查即可。基本上就结束检查，剩下就是把本来的环境恢复到之前的情况。

终于结束，我们回到了 i386_vm_init，开始创建虚拟内存空间。首先是 pages 数组，这个数组在线性地址 UPAGES 对用户是只读的。可以看到在 boot_map_segment 做了映射工作。进入该函数后，首先是对 4KB 对齐，然后就是循环不断调用 pgdir_walk 创建要映射到的虚拟位置的页表的首地址，然后将物理页的内容和权限赋值给它。然后是 PCB 所需存放空间的映射，内核栈的映射，KERNBASE 以上所有空间的映射。

进入 check_boot_pgdir，主要是用 check_va2pa 检查是否物理地址和虚拟地址 map 好了。修改 cr0，并开启页式地址转换。Jos 在这里又重新开启了一个段模式，基址为 0，取消线性地址转换，重新设置一次 boot_cr3，刷新 tlb。内存基本设置完成。

4、进程管理

出来回到 i386_init，但是又进入了 env_init，这里的 PCB 空闲链表使用 env_free_list 来进行控制，与 page_free_list 的控制方法一模一样，同样地，这里也设置了一个 envs 的数组。一开始，所有的 env 都是空闲的，故设置的状态为 FREE。这里使用的一套宏定义和 memory 那边一模一样。然后回到 i386_init，调用 idt_init，这里是开启中断服务，首先加载了 gdt 表，

然后 extern 了一些异常处理程序，使用 SETGATE 为这些程序设置了中断门或异常门。这里做了提示：1 是表示设置 trap 门，0 是表示设置中断门，Gatedesc 就是门结构。在整个设置过程中，比较有作用的就是 5 个传进来的参数。Gate 是中断向量表当中的一项门，GD_KT 说明是内核代码段的部分（因为这里设置的是中断门或异常门）。dpl 是运行该中断所需要的权限，一般会使用 cpl 与其进行比较，cpl 比 dpl 大会被拒绝，其他的情况才可以陷入该中断门。设置完门之后，就对 ts 进行了一系列设置（Taskstate），然后加载了 idt 就结束了。

pic_init 是打开 8259A 中断控制器，kclock_init 打开时钟中断，实话说就是为了做分时调度，没有什么与进程管理相关的地方，pass 掉。

然后 ENV_CREATE 事实上是一个宏定义，它是用来调入已经放入 kernel 当中的用户程序。对于进程的测试方式大多是使用这种方式运行。基本上以后的程序也会使用 syscall 来与程序进行交互，所以这里直接看代码，就不跟着程序运行了。

Env.c 中主要实现的进程的控制，函数不是很多。主要根据一个进程的创建开始，之后我们通过 env_init 完成了一系列初始化的工作。

真正我们开始创建一个进程是 env_alloc，当然，为了后期工作，其实是有必要在 Env 结构中加入优先级顺序，方便完成抢占式调度。Env 的结构也不多说了，基本 tf 现场信息、自己独有的页目录。在 env_alloc 当中，我们首先看有木有空闲的 PCB 结构，没有就返回没有空闲 PCB，然后就开始为这个 PCB 设置页目录，启动函数 env_setup_vm，一如既往的使用 page_alloc 分配一页，然后将该页对应的虚拟地址给 pgdir，物理地址给 cr3。因为 mem 系列函数都是操作虚拟地址，故都是用 pgdir 的虚拟地址为参数。UTOP=UENVS=UXSTACKTOP 和 UPAGES 大小一样，只有 PTSIZE=4MB 空闲。memmove 即是 memcpy，参数差不多。然后设置一下不同位置的访问权限即可。下一个任务就是算 evid，首先要保证 id 非负，然后就是与 PCB 的编号相关。然后就是设置一些基本信息。然后就是清空进程的现场信息。然后为用户进程设置段信息和权限信息，还有用户的栈指针。然后就是开启中断功能，将页中断处理器清理掉，接受信息 flag 也清理掉，将 PCB 从空闲链表当中删掉，结束。env_create 执行后先取出一块 PCB，然后使用 load_icode 将之前提到过的用户二进制程序加载进去，这里先将一个二进制文件转换成 env_elf 到 binary。之后见检查是不是可执行文件，然后载入 ph->p_type 为 ELF_PROG_LOAD，一共执行 e_phnum 次。注意在此之前一样要更换页目录到该进程之下的页目录，然后调用 segment_alloc 对该段连续地址进行，将 va 进行 4KB 对齐。然后从该对齐后地址开始，每次分配一页，并插入到页表中，直到结束。然后先将刚刚分配的页空间清空，并将二进制文件当中的数据复制到该虚拟地址处，在 eip 中放下程序入口，设置一下用户栈，结束。

几个强大的概念需要澄清

1、实模式、保护模式与内核

实模式并不是内核模式，只有一个单一的模式，而且开启了段式地址变换，这里的地址一般是 16 位，分有 16 个段，地址变换机制为段基址左移 4 位，然后加上段内偏移，得到线性地址。此时的线性地址也就是物理地址。

改变 cr0 转换到保护模式，然后加载了自己的段表，这里当前更新了段表，空段、代码段、数据段等等。基址为 0，然后加载内核到 0x100000 处。按道理说这里还没有使用中断，我们在它后面加载内核时可以看到 Jos 还是使用的轮询的方式。

进入内核之后，它加载了新的段表，这时加载的段表和保护模式基本一致，但段基址从 KERNBASE 开始。在开启页式模式之后，它又加载了新的段表，将段基址变回了 0。

然后我们就可以开始下一个话题。

2、虚拟地址、线性地址、物理地址和内核物理地址

虚拟地址一直都是[16 位段号][32 位段内偏移]构成的，才不是想象中的 32 位虚拟地址！

线性地址才是我们想象中的 32 位地址，也是地址变换负责将线性地址转换成物理地址。在开启页式地址变换之前，线性地址和物理地址是常数距离相关的。开启之后，由于重新加载了段表，本来使得之后的物理地址本来和线性地址一样的，但是由于之后都是使用页分配技术，所以这种映射关系也不存在了，二是被开启的页式地址转换来代替。而内核物理地址由于是在 KERNBASE 下装载的，而且它的位置又没有被改变过，因此它和线性地址关于 KERNBASE 的变换规则仍然是保持的。

也就是说，**转换方式一致都是虚拟地址->线性地址->物理地址**：

实模式下：段号 $<< 4 +$ 段内位移 = 线性地址 = 物理地址

保护模式下：段选择子(段寄存器+左边 13 位*8, 段描述符 8 字节对齐)->段描述符

段基址 + 虚拟地址的段内偏移 32 位 -> 线性地址 = 物理地址

内核初期：段选择子(段寄存器+左边 13 位*8, 段描述符 8 字节对齐)->段描述符

段基址 + 虚拟地址的段内偏移 32 位 -> 线性地址

线性地址 - KERNBASE = 物理地址

内核后期：

不开启页式地址变换：

段选择子(段寄存器+左边 13 位*8, 段描述符 8 字节对齐)->段描述符

段基址 + 虚拟地址的段内偏移 32 位 -> 线性地址 = 物理地址

开启页式地址变换：

段选择子(段寄存器+左边 13 位*8, 段描述符 8 字节对齐)->段描述符

段基址 + 虚拟地址的段内偏移 32 位 -> 线性地址

线性地址前 10 位 + pgdir -> 内容为存放页表的物理首地址，因为其页表放在一页中，其关于 4KB 对齐，一共 1024 项。

拿到页表所在的物理首地址后，根据线性地址中间的 10 位放在其 11~2 位的空间，因为一个页表项为 4 字节，所以最后两位为零，如此找到页表项的起始字节处，然后从中读取内容，取前 20 位，这是线性地址所在页的物理首地址。

然后将线性地址的最后 12 位与刚刚得到的 20 位进行拼接，得到一个 32 位的地址，这时，我们才真正得到的我们要找字节的物理地址。

Finish！其实还蛮简单的。

重要函数解析

Cprintf：实质上起作用的都是 vcprintfmt，这个函数有两个重要参数，一个是 fmt、一个是 ap。

- What values does gcc on the x86 push on the stack for the call:

Printf (“the class number is %s and used to be %d \n”, “6828”, 6097)

```

andl    $-16, %esp
subl    $16, %esp
movl    $.LC0, %eax
movl    $6097, 8(%esp)
movl    $.LC1, 4(%esp)
movl    %eax, (%esp)

```

fmt 和 ap，其中 fmt 指向“the class...”，ap 指向“6828”首地址。

- Explain what va_list ap does briefly.

ap 主要是对应 fmt 的格式指向它们存放的地址，在之前，通过 va_start，va 跳过了 fmt 部分的固定参数，直接指向了那些可变参数。之后又使用 va_arg 将 ap 慢慢按照 fmt 的格式类型往后移动。

Booting a PC

Author: Xin Yao

一、导论

- PART1、熟悉 x86 汇编语言, Bochs x86 模拟器, PC 的加电启动过程
- PART2、查看其中目录下的 boot loader
- PART3、深入研究启动模块。

Part 1: PC Bootstrap

Getting Started with x86 assembly

预习部分:

PC Assembly Language Book

<http://pdos.csail.mit.edu/6.828/2007/readings/pcasm-book.pdf>

Brennan's Guide to Inline Assembly

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

Exercise 1、

读以上链接给出的书本, 因为时间比较短, 所以简略看了一遍。汇编语言在功能上都是相似的, 只要注意一些细节格式就可以了。

Simulating the x86

1、安装 Bochs

环境: Ubuntu 12.04 i386

以下是安装前的准备

```
sudo apt-get install build-essential
sudo apt-get install xorg-dev
sudo apt-get install pkg-config
sudo apt-get install libgtk2.0-dev
```

注意 Bochs 的安装时的配置, 如果你想使用图形界面进行调试的话, 有一些附加的修改是必要的。

详情请见

<http://bochs.sourceforge.net/doc/docbook/user/internal-debugger.html>

文章的最后有相关图形界面调试器的介绍, 在安装前读一下多有裨益。

一般安装:

```
./configure --prefix=/usr --enable-disasm --enable-debugger
make
```

sudo make install

基本 bebug 文件 :

```
<bochs:1> help
help - show list of debugger commands
help 'command'- show short command description
-*- Debugger control -*-
    help, q|quit|exit, set, instrument, show, trace-on, trace-off,
    record, playback, load-symbols, slist
```

-*- Execution control -*

c|cont, s|step|stepl, p|n|next, modebp

-*- Breakpoint management -*

v|vbreak, l|lbreak, pb|pbreak|b|break, sb, sba, blist,
 bpe, bpd, d|del|delete

-*- CPU and memory contents -*

x, xp, u|disas|disassemble, r|reg|registers, setpmem, crc, info, dump_cpu,
 set_cpu, ptime, print-stack, watch, ?|calc

2、解压 lab1

注意几个可能需要修改的问题

1、由于 sign.pl 的缘故

块大小的限制, 如果超过 510 字节则报错.否则,如果填满代码直至 510 字节,并最后

写入两个字节:0x55AA 到文件.至于为什么是 0x55AA?因为引导扇区最后两个字节被规定了必须这样.

在 boot/Makefrag 当中修改:

```
$ (V)$OBJCOPY -S -O binary -R ".eh_frame" $@.out $@
-R ".eh_frame" 将 eh_frame 段去掉了
```

当然你可以选择低版本的 gcc 解决这个问题

2、启动问题

System BIOS must end at 0xFFFFFFF

解决方案:

在.bochsrc 加上

display_library sdl

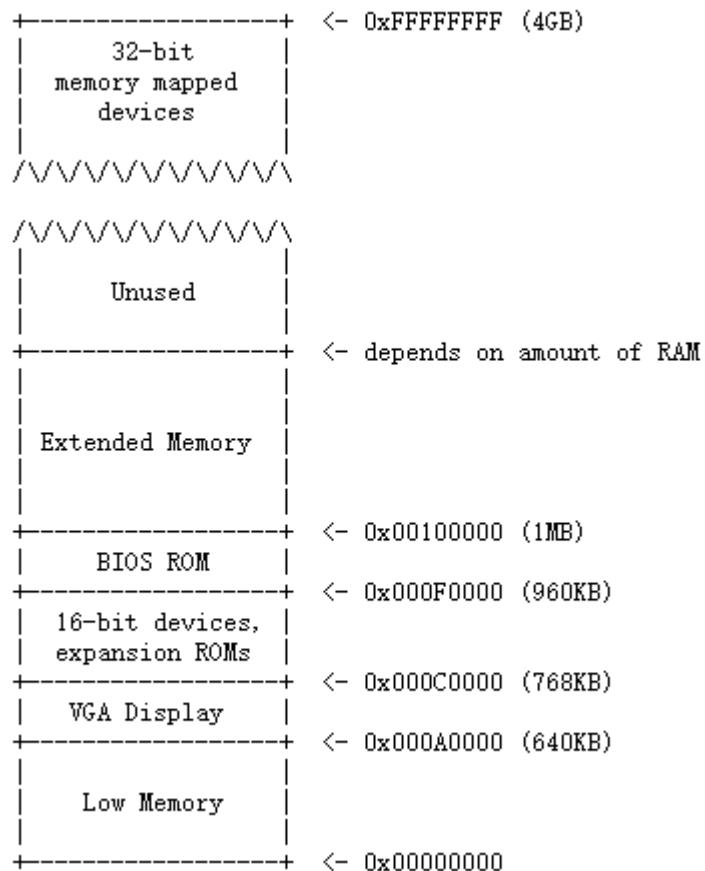
或者找到你的同学, 拷贝一份.bochsrc, 这个不是本实验的重点。

一般而言, 现在 make 一般都没有问题, 如果还是不行, 建议从头再来, 或上网查找其它方案

Exercise2、

熟悉 debug 的用法。

The PC's Physical Address Space



A PC's physical address space

根据介绍, 早期 PC 的物理地址从 0x00000000 开始, 到 0x000FFFFF 结束, 而不是 0xFFFFFFFF 处结束, 一共有 640KB 的可访问内存空间。实际上, 最早的 PC 只配备了 16kB、32kB、64kB 的空间。

从 0x000A0000 到 0x000FFFFF 的 384KB 的空间被保留作为特殊的作用: 如视频显示的缓冲。最重要被保留的地方是 BIOS, 早一些的 PC 的 BIOS 是放在 ROM 里面的, 但现在的则放在可擦写闪存里面。BIOS 执行一个基本系统, 对硬件进行了检查, 然后将操作系统从

它所在的位置进行加载，并将控制权交给操作系统。

Modern PCs therefore have a "hole" in physical memory from 0x000A0000 to 0x00100000, dividing RAM into "low" or "conventional memory" (the first 640KB) and "extended memory" (everything else). In addition, some space at the very top of the PC's 32-bit physical address space, above all physical RAM, is now commonly reserved by the BIOS for use by 32-bit PCI devices

为了兼容之前的落后的已有软件，现代 PC 的物理内存从 0x000A0000 到 0x00100000 有一个空洞，将 RAM 划分为“传统内存”（最开始的 640KB）和“扩展内存”。现在的 PC 的 32 位内存的最高位的部分空间，一般保留在 BIOS 和 PCI 硬件。

The ROM BIOS

1、PC 开始 CS = 0xF000, IP = 0xFFFF0

2、最开始执行一个跳转指令 jmp far f000:e05b

IBM 如此设计开机，BIOS 的地址范围为 0x000F0000 – 0x000FFFFF，这个设计确保了 BIOS 可以获取机器的控制权，因为开机后在 RAM 里处理器可以执行的范围里面没有其他的程序。

在处理机重置后，它进入了实模式，将 CS 设置为 0xF000, IP 为 0xFFFF0，执行从(CS:IP) 段地址开始。

地址转换：physical address = 16 * segment + offset

因此段地址转换到物理地址可得：

$$16 * 0xF000 + 0xFFFF0 = 0xFFFFF0$$

这个是 BIOS 离结束还有 16 个字节的地址，所以我们要使用 jmp 跳转到更前面的地方执行。

Exercise 3:

使用 debug 多测试几条指令，猜猜它的功能，你可以阅读相关 IO 的书籍

BIOS 跑起来的时候，设置中断描述符表，并初始化了一些硬件，如 VGA display

Part 2: The Boot Loader

如果从硬盘启动，第一个扇区为引导扇区，里面为 the boot loader 的代码，它将这 512 个字节放入内存，0x7C00 到 0x7DFF，然后使用 jmp 指令跳转到 CS:IP 0000:7C00，将控制权交给 boot loader。由于 CD 比 PC 的发展完，所以现在 BIOS 从 CD 启动都比较复杂，CD 使用 2048 个字节的扇区，这样 BIOS 可以从 CD 里加载更多的启动镜像。

boot 目录下的 boot.s 和 main.c 文件。boot loader 有两个功能：

1、处理机从实模式转移到 32 位保护模式，这个模式只允许软件访问 1M 以上的内存空间。你需要明白现在的 offset 是 32，而不是 16。（段地址到物理地址的转换）

2、boot loader 将内核读取通过直接访问 IDE disk。

查看 boot.asm 文件。

Exercise 4. Set a breakpoint at address 0x7c00, which is where the boot sector will be loaded. Continue execution until that break point. 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 u command in Bochs to disassemble sequences of instructions in the boot loader, and compare the original boot loader source code with both the GNU disassembly in obj/boot/boot.asm and the Bochs disassembly from the u command.

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.

1、At exactly what point does the processor transition from executing 16-bit code to executing 32-bit code?

```

# 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    %scr0, %eax
orl    $CR0_PE_ON, %eax
movl    %eax, %scr0

```

由注释我们就知道该段代码的功能了，处理器从 BIOS 进入 boot loader 后，在 boot/boot.S 中第 48 行到第 51 行代码，boot loader 将寄存器 cr0 的末位更改为 1，使得处理器从实模式更改到保护模式。

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

```

// call the entry point from the ELF header
// note: does not return!
((void (*)(void)) (ELFHDR->e_entry & 0xFFFF))();

```

boot loader 执行的最后一条指令为将内核 ELF 文件载入内存后，调用内核入口点，在 boot/main.c 中的第 58 行。

3、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?

boot loader 从内核 ELF 文件的文件头中可以知道该 ELF 文件被分成了多少 section 和多少 program。

objdump -h obj/kern/kernel 或者 objdump -x obj/kern/kernel 可以看到结果

Jos 就“很友好”的将编译好的 kernel 就放在 mbr 的后面，也就是第二个扇区（也可以说是第 1 个扇区，在这之前还有第 0 个扇区）的位置。

```

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

```

然后 main.c 里定义了两个函数，readseg 和 readsect，从逻辑上将，第一个函数的功能是“将相对于 kernel 基址便宜 offset 个自己处之后的 count 各字节的东西读到物理地址 pa 处”，而第二个的功能是“将相对于第二个扇区便宜 offset 字节的扇区里的内容读到 dst 的位置”。

```

architecture: i386, flags 0x000000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0xf010000c

Program Header:
  LOAD off 0x000001000 vaddr 0xf0100000 paddr 0xf0100000 align 2**12
    filesz 0x00012f88 memsz 0x00012f88 flags r-x
  LOAD off 0x00014000 vaddr 0xf0113000 paddr 0xf0113000 align 2**12
    filesz 0x00062c15 memsz 0x00063b30 flags rw-
  STACK off 0x00000000 vaddr 0x000000000 paddr 0x000000000 align 2**2
    filesz 0x00000000 memsz 0x000000000 flags rwx

Sections:
Idx Name      Size     VMA       LMA       File off  Align
 0 .text      00006135  f0100000  f0100000  00001000  2**4
              CONTENTS, ALLOC, LOAD, READONLY, CODE
 1 .rodata    00001643  f0106140  f0106140  00007140  2**5
              CONTENTS, ALLOC, LOAD, READONLY, DATA
 2 .stab      000087cd  f0107784  f0107784  00008784  2**2
              CONTENTS, ALLOC, LOAD, READONLY, DATA
 3 .stabstr   00003037  f010ff51  f010ff51  00010f51  2**0
              CONTENTS, ALLOC, LOAD, READONLY, DATA
 4 .data      00062c15  f0113000  f0113000  00014000  2**12
              CONTENTS, ALLOC, LOAD, DATA
 5 .bss       00000f10  f0175c20  f0175c20  00076c15  2**5
              ALLOC
 6 .comment   0000002a  00000000  00000000  00076c15  2**0
              CONTENTS, READONLY

```

More information:

[0x00007c2a] 0000:7c2a (unk. ctxt): mov cr0, eax, this instruction start the protect module

The boot loader's last instruction is:[0x00007d84] 0008:00007d84 (unk. ctxt): call eax

The first instruction of kernel is:[0x0010000c] 0008:0010000c (unk. ctxt): mov

word ptr ds:0x472, 0x1234

Loading the Kernel

Exercise5:

- 1、复习指针的用法
- 2、指针和地址

弄懂 boot/main.c, 你需要知道 ELF 二进制文件。注意到 obj/kern/kernel, 并使用 objdump 文件显示它。

ELF 可执行文件的文件头负责载入相关信息，它后面是几段程序节，每一段都是代码或者数据，他们会在一个特殊的内存地址被载入。

Exercise6: 内存 0x00100000 处的 8 个 word 的信息在 BISO 切换到 BIOS 中和 boot loader 切换到内核运行分别是什么？

内存 0x00100000 是内核的最终载入地址，内核由 Boot loader 负责载入。初始当 BIOS 切换到 boot loader 时，它还没有开始相应的装载工作，所以这个时候看所有的 8 个 word 全是 0。而当 boot loader 转换到内核运行时，这个时候内核已经装载完毕，所以从 0x00100000 开始就是内核 ELF 文件的文件内容了。

Link vs. Load Address

载入地址是二进制数真正被载入的地方，就好像 BIOS 被计算机硬件载入到 0xf0000，这就是 BIOS 的载入地址。BIOS 将 the boot sector 载入到 0x7c00，故这就是它的载入地址。

链接地址是存放二进制数存放的位置的内存地址。链接到一个字节意味着一个被给出链接地址将被用来去已经载入该二进制数所在的地址。如果改地址实际上没有改二进制文件，那么链接地址就不起作用了。

Exercise7:

这个练习主要目的是弄清载入地址和链接地址的关系。

1、在 boot/Makefrag 当中修改-Ttext 后面的 0x7c00 为 0x8c00，因为此时的 boot loader 实际被载入到了 0x8c00，所以在后面使用链接地址找 0x7c00 的时候，无法找到相关程序，故此时机器会重启。

2、出现问题的地方在 boot/boot.S 中第 55 行

ljmp \$PROT_MODE_CSEG, \$protcseg

很明显，只有涉及相对位置跳转的语句才有可能在链接地址和装载地址不一致的时候发生问题。

Part 3: The Kernel

内核的链接地址和加载地址有很大的差别，内核通常会被链接到很高的地址开始运行，比如 0xf0100000，这是为了保证用户进程的虚地址的地址部分能被使用，lab2，我们将会更加清楚地了解这些内容。

很多机器在 0xf0100000 没有任何其他的物理内存空间。所以我们不能将内核继续保存在这个地方。我们使用处理器的内存管理硬件将虚拟地址 0xf0100000(链接到内核运行的链接地址)映射到物理地址 0x00100000，这个地方我们曾经使用 boot loader 载入整个内核。这样，即使内核的虚拟地址既可以很高使得用户程序的使用没有障碍，也使得真正内核被存放

在物理内存中 1M 以内的空间。

我们还将整个后面的 256M 的物理内存空间(0x00000000~0x0fffffff)，映射到 0xf0000000 到 0xffffffff。你应该能够看到为什么 Jos 被限制使用仅仅 256M 的空间。

Exercise8：找到 v2p 的映射起作用的地方，GDT 的作用，哪一条指令是第一个因为映射失败而导致不能正常工作。

在 0x0010000c 处设置断点，b 0x0010000c。通过单步调试，我们可以看到

```
lgdt in [0x00100015] 0008:00100015
```

在这条指令后，新的映射方式就开始了。

boot loader 在初始化数据的时候自己定义了 GDT，切换到内核运行后，内核在载入初期马上重新定义了自己的 GDT，然后替换掉了原有的 GDT，在 kern/entry.S 第 47 行处：

```
lgdt RELOC(mygdtdesc)
```

我们可以看到内核启用了新的 GDT，RELOC 的作用就是做新的 map 功能。新的段表基址全部变成了-KERNBASE。这样就可以在每一次都能将内核链接地址当中的 0xf01xxxxx 当中的 f 变为 0，这个和 boot loader 的 boot/main.c 在载入内核 ELF 的分段时使用的与地址法在效果上是完全一致的。

同样地，main.c 当中的 readseg 也有自己的具体地址转换机制，

(boot/main.c)

让我们修改一下 printf.c 的内容，使得其可以打印 8 进制数字

printfmt.c, printf.c and console.c 文件的关联

printfmt -> vprintfmt -> putch -> cputchar -> cons-putc -> lpt-putc,cga-putc(set the char color)

Exercise 9：完成 printfmt.c 当中%o 的情形

只要细心一点，你就可以发现 case 当中的 x 是打印 10 进制数的情形，如此我们只需要模仿 16 的 case 就可以完成打印 8 进制数的情形，很简单。

1、解释 printf.c 和 console.c 的接口。

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

大意就是解释 printf.c 和 console.c 的接口功能。

console.c 主要提供一些与硬件直接进行交互的接口以便其他程序进行输入输出的调用。其中与 kern/printf.c 进行交互的为 cputchar 函数。但是其直接调用了 cons_put 函数。cputchar 函数用于将一个字符输出到显示器，在 kern/printf.c 中被 putch 函数调用。lpt_putc 直接使用 I/O 接口编程，cag_puts 为解析打印类型，并用于显示字符和它的颜色，颜色为 8-15 位，字符为 0-7 位。

Explain the following from console.c:

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

CRT_COLS 为 80，CRT_SIZE,80*25，这一行的作用是在字符打印到达屏幕的最后一行的时候，将从 crt_buf+80 开始的 24*80 内存内容复制到 crt_buf 开始之后 24*80 的地方。然后将最后一行的内容空出来，如此实际上的功能就是当屏幕满了之后，将最开始的一排去掉，然后将剩下的内容整体上移一排，再把空出来的最后一排用背景色填充。

For the following questions you might wish to consult the notes for Lecture 2. These notes cover GCC's calling convention on the x86.

Trace the execution of the following code step-by-step

```
int x = 1, y = 3, z = 4;
```

```
cprintf("x %d, y %x, z %d\n", x, y, z);
```

1、In the call to cprintf(), to what does fmt point? To what does ap point?

fmt 指向的是格式字符串，在上例中即"x %d, y %x, z %d \n"，而 ap 指向的是不定参数表的第一个参数地址，在上例中即 x。

2、List (in order of execution) each call to cons_putc, va_arg, and vcprintf. For cons_putc, list its argument as well. For va_arg, list what ap points to before and after the call. For vcprintf list the values of its two arguments.

va.arg 的作用是将 ap 每次指向的地址往后移动需要的类型个字节。

```
#define __va_size(type) \
    (((sizeof(type) + sizeof(long) - 1) / \
    sizeof(long)) * sizeof(long))

#define va_start(ap, last) \
    ((ap) = (va_list)&(last) + __va_size(last))

#define va_arg(ap, type) \
    (*(type *)((ap) += __va_size(type)), (ap) - \
    __va_size(type)))

#define va_end(ap) ((void)0)
```

3、Run the following code.

```
unsigned int i = 0x00646c72;
cprintf("H%x Wo%s", 57616, &i);
```

4、What is the output? Explain how this output is arrived out in the step-by-step manner of the previous exercise.

Hello World 因为 $57616 = \text{ellow}$ 且 $0x00646c72 = \text{rlld}$ 。

The output depends on that fact that the x86 is little-endian. If the x86 were instead big-endian what would you set i to in order to yield the same output? Would you need to change 57616 to a different value?

仅仅只是 World = Wodlr。大小端机器地址存放的方式不同

5、In the following code, what is going to be printed after 'y='? (note: the answer is not a specific value.) Why does this happen?

```
cprintf("x=%d y=%d", 3);
```

y= -267325460, 因为根据上面对参数的分析，打印的是第二个参数首地址开始 4 个字节的内容，**而这个内容实际上是不能确定的。**

6、Let's say that GCC changed its calling convention so that it pushed arguments on the stack in declaration order, so that the last argument is pushed last. How would you have to change cprintf or its interface so that it would still be possible to pass it a variable number of arguments?

在第二题当中，va_start 和 va_arg 的宏定义我们可以得到如下结论：

va_arg 每次是以地址往后增长取出下一参数变量的地址的。而这个实现方式就默认假设了编译器是以从右往左的顺序将参数入栈的。因为栈是以从高往低的方向增长的。后压栈的参数放在了内存地址的低位置，所以如果要以从左到右的顺序依次取出每个变量，那么编译器必须以相反的顺序即从右往左将参数压栈。如果编译器更改了压栈的顺序，那么为了仍然能正确取出所有的参数，那么需要修改上面代码中的 va_start 和 va_arg 两个宏，将其改成用减法得到新地址即可。

Challenge Enhance the console to allow text to be printed in different colors. The traditional way to do this is to make it interpret ANSI escape sequences embedded in the text strings printed to the console, but you may use any mechanism you like. There is plenty of information on the [6.828 reference page](#) and elsewhere on the web on programming the VGA display hardware. If you're feeling really adventurous, you could try switching the VGA hardware into a graphics mode and making the console draw text onto the graphical frame buffer.

方案一、在 cprintf → vcprintf → vprintfmt → putch → cputchar → cons_putc → cga_putc 思路，在 cga_putc 当中修改，这样可以对每个打印的字符的颜色进行定位。也为我们后来实现 ls –color 这种功能奠定了基础。

在 cga_putc(int c) 的参数 c 当中，低八位用于指定打印的字符：

15	14	13	12	11	10	9	8
B_I	B_R	B_G	B_B	F_I	F_R	F_G	F_B

15 位到 12 位用于指定字符的背景颜色，颜色有三位 RGB 颜色代码和 I 位是否显示高亮来决定。11 到 8 位用于指定字符的前景颜色。

如此在 printf 的格式化字符串中指定一个格式化参数 %C，用来指定从这个参数以外打印的所有字符串的颜色。直到碰到下一个 %C 出现，打印的数字都会以 %Co 指定的颜色打出。为了方便常用颜色的使用，定义一组三位字符为颜色指定符，用于代替数字，可以直观的使用。

具体方法就是在 vprintf 函数当中加上一条分支语句，这样可以对 C 字符进行解析，间 fmt 后连续的三位字符最为参数值解析，复制给我们定义的颜色变量 color 即可

在 cga_putc 我们在最开始就用 c = c | (color << 8) 即可将颜色值复制给将要打印的字符。

方案二，在 console.h 设一个颜色相关全局变量，然后在 console.h 当中实现对变量的设置，可以在 monitor.c 当中添加一个命令，根据这个命令设置一个函数，然后根据命令的参数调用设置函数设置这个变量。原理和上面一样。

```
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> setcolor 0x7100
set color to 0x7100
K> help
help - Display this list of commands
kerninfo - Display information about the kernel
setcolor - set the back color of the Os, Remember to add '0x'
add - add all num followed
produce - mul all num followed
backtrace - show the function in the stack
```

效果预览图

The Stack

Exercise 10. Determine where the kernel initializes its stack, and exactly where in memory its stack is located. How does the kernel reserve space for its stack? And at which "end" of this reserved area is the stack pointer initialized to point to?

在 kern/entry.S 文件当中（实际上是由编译时 kernel.ld 决定了改程序的入口地址和虚拟的载入内核的地址）：

```
_start:
    movw    $0x1234,0x472          # warm boot

    # Establish our own GDT in place of the boot loader's
    # temporary GDT.
    lgdt    RELOC(mygdtdesc)      # load descriptor table
```

内核在这里重新定义了自己的 GDT 表，进入内核后，新的 GDT 表的代码段和数据段的基址都是 0x10000000。紧接着，程序初始化段寄存器 ds、es、ss 以及 ebp、esp，esp 指向栈顶。下面的 CODE_SEL 为代码段在 mygdt 段当中的索引。

```

ljmp    $CODE_SEL,$relocated      # reload CS by jumping
relocated:

# Clear the frame_pointer register (EBP)
# so that once we get into debugging C code,
# stack backtraces will be terminated properly.
movl    $0x0,%ebp                # nuke frame pointer

# Leave a few words on the stack for the user trap frame
movl    $(bootstacktop-SIZEOF_STRUCT_TRAPFRAME),%esp

# now to C code
call    i386_init

```

而这里就是内核进行栈初始化的代码。这里将 ebp 置为 0, %esp 初始化为 bootstacktop 这里的 SIZEOF_STRUCT_TRAPFRAME 在 trap.h 当中被定义为 0x44, 在以前的代码当中是没有这一部分，这个改进是为用户的 trap frame 留一部分空间。

在此之前，在 bootstack 的声明当中，我们可以看到之前还有一段 32k 的 KSTKSIZE，属于临时堆栈区，其后的第一个字节为 bootstacktop 指向的空间。

下面在 i386_init 函数当中进行一系列的初始化，在 monitor 循环等待用户输入的指令。

```

while (1) {
    buf = readline("K> ");
    if (buf != NULL)
        if (runcmd(buf, tf) < 0)
            break;
}

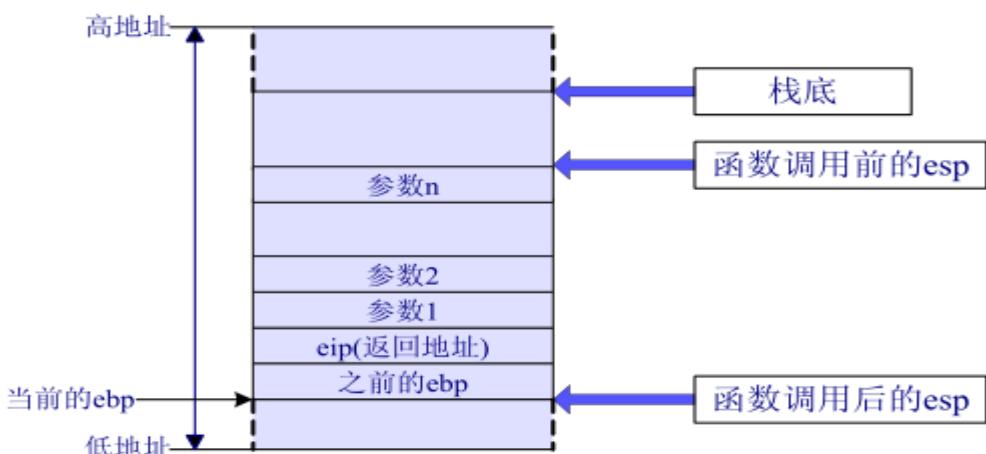
```

这里我们可以看到 readline 是用来等待用户输入的，知道一个回车就可以结束输入。当接受键盘输入的缓冲不为空的时候，程序就开始处理命令。在 runcmd 当中，就是把输入的字符串，并将内部的空格全部替换成空字符。如此便得到 argv 和 argc 两个参数。并根据 argv[0] 的值来调用 monitor 当中对应的函数。

Exercise 11. 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 in Bochs, and examine what happens each time it gets called after the kernel starts. There are two ways you can set this breakpoint: with the b command and a physical address, or with the vb command, a segment selector (use 8 for the code segment), and a virtual address. How many 32-bit words does each recursive nesting level of test_backtrace push on the stack, and what are those words?

这里会出现 esp 始终为 0 的情形，这里我们需要将 GUNmakefile 当中编译器的优化选项修改一下，因为 Mit 的这个项目是用很早的 gcc 编译的，现在的 gcc 直接编译的优化程度已经和当时的 O2 差不多了，而现在的 gcc 用 O2 编译会使得程序变得更加简单，但是由于过于简化会使得汇编代码中没有压栈的过程，esp 没有用上，所以它始终会为 0.

堆栈的原理很简单，这里就不再赘述。



eip : 存储当前执行指令的下一条指令在内存中的偏移地址

esp : 存储指向栈顶的指针。

ebp : 存储指向当前函数需要使用的参数的指针

我们需要做的就是使用 ebp 不断循环回退取值。当前 ebp 所指向的空间保存的是之前的 ebp，且在当前 ebp 上面的地址空间存放的就是我们要用的信息。还记得我们曾经将 ebp 赋值为 0 了么，是的，这个循环的终止条件就是 ebp 为 0。

具体参考见：

<pre>WHITESPACE MAXARGS struct Command variable commands function StrToInt mon_add mon_mul mon_setcolor mon_help mon_kerninfo mon_backtrace runcmd monitor read_eip</pre>	<pre>int mon_backtrace(int argc, char **argv, struct Trapframe *tf) { // Your code here. uint32_t ebp = read_ebp(), *p, eip, i; struct Eipdebuginfo bug; while (ebp > 0) { p = (uint32_t *)ebp; eip = p[1]; cprintf("ebp %x ,eip %x args", ebp, eip); for (i = 2; i < 6; i++) { cprintf(" %08x, ", p[i]); } debuginfo_eip(eip, &bug); cprintf("\n\t%s : %d : ", bug.eip_file, bug.eip_line); for (i = 0; i < bug.eip_fn_namelen; i++) cputchar(bug.eip_fn_name[i]); //the same to the name //cprintf(" %s ", bug.eip_fn_name); cprintf("+%d\n", eip - bug.eip_fn_addr); ebp = *p; } }</pre>
--	--

重要附录参考代码

<pre>void selffun(){ unsigned int i = 0x00646c72; cprintf("+++++++\n"); cprintf(" H%x Wo%\$! !\n", 57616, &i); cprintf("+++++++\n"); cprintf("welcome to the Jos system!\n"); } mon_backtrace(int argc, char **argv, struct Trapframe *tf) {</pre>	<pre>// Your code here. uint32_t ebp = read_ebp(), *p, eip, i; struct Eipdebuginfo bug; while (ebp > 0) { p = (uint32_t *)ebp; eip = p[1]; cprintf("ebp %x ,eip %x args", ebp, eip); for (i = 2; i < 6; i++) { cprintf(" %08x, ", p[i]); } debuginfo_eip(eip, &bug); cprintf("\n\t%s : %d : ", bug.eip_file, bug.eip_line); for (i = 0; i < bug.eip_fn_namelen; i++) cputchar(bug.eip_fn_name[i]); //the same to the name //cprintf(" %s ", bug.eip_fn_name); cprintf("+%d\n", eip - bug.eip_fn_addr); ebp = *p; } return 0;</pre>
---	---

Memory Management

Author: Xin Yao

The remain of lab1:

Exercise 1. Modify your stack backtrace function to display, for each EIP, the function name, source file name, and line number corresponding to that EIP. To help you we have provided debuginfo_eip, which looks up eip in the symbol table and is defined in kern/kdebug.c.

不难看出，这一部分其实就是 lab1 的继续，但是因为我们按照 2007 完成的，所以尽量保持一致吧。

回到 kernel.ld，还记得我们说过编译时，kernel.ld 的作用就是指导了 kernel 开始的位置和 kernel 的入口。在 kernel.ld 中，我们明显地可以看到.stab 和.stabstr 的部分。

关于结构体 Stab 的定义和属性的意义，下面的注释都已经给出来了。

```
// Entries in the STABS table are formatted as follows.
struct Stab {
    uint32_t n_strx;      // index into string table of name
    uint8_t n_type;        // type of symbol
    uint8_t n_other;       // misc info (usually empty)
    uint16_t n_desc;       // description field
    uintptr_t n_value;     // value of symbol
};
```

- n_strx：该项对应的在 stabstr 节内的索引偏移
- n_type：该项描述的符号类型，下面会结合实例进行说明
- n_other：不用场位，当 n_desc 溢出是能缓冲溢出位
- n_desc：符号描述，在我们调试的角度，我们只要知道它可以表示源文件中的行号。
- n_value：和当前表项对应符号值。

我们使用 objdump -G 来查看一个 ELF 文件的 stab 节信息，如果感觉太乱，可以使用 grep 来进行限定。如 objdump -G obj/kern/kernel| grep FUN

在 FUN 的符号表当中，这些函数都是按照虚拟地址从小到大进行排序。

调试信息的赋值过程，我们通过题目的提示，可以知道在 debuginfo_eip 函数当中，如果你们开始使用了 cscope 和 taglist，找个这个函数是很简单的。它位于 kdebug.c 当中。

```
struct UserStabData {
    const struct Stab *stabs;
    const struct Stab *stab_end;
    const char *stabstr;
    const char *stabstr_end;
};
```

这个 c 文件当中仅有两个函数：stab_binsearch 和 debuginfo_eip。

先说一下 stab_binsearch 的功能，在 2007 年的版本当中它是一个 void 型函数。

但是在参数中，region_left 和 region_right 是用来传递表项的序号，但是又因为是相对寻址，所以在调用的时候 region_left 默认为 0，而 region_right 则为整个要查找范围内的表项长度-1。stabs 则是调试信息的起始地址，type 代表要搜索的代码类型，addr 则是参数 eip。

其实这个二分查找的算法思想和一般二分法一样，不过他比对的是查找类型。一旦找到后 region_left 肯定不会再为 0（也就是作为传入的参数 lfile），找到这个文件所在区域后，我们在开始找这个文件被指定的函数。同样的思路找，但是这里在找完后，需要检查一下，如果表项里地址很小的话就需要做一下地址转换，只是为了以防万一（比如查看 SLINE 的时候就会出现这种问题），注释上也写得很清楚。得到行号即使使用到 stab 的 desc 信息即可（这个前文已经叙说过了）。

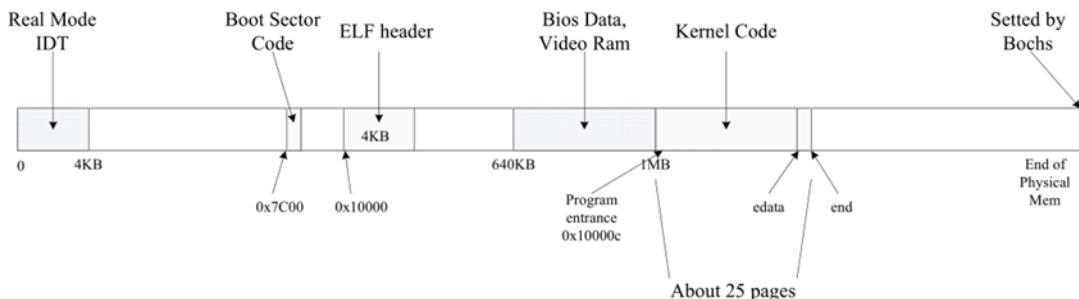
在 debuginfo_eip 当中：

```
if (addr >= ULIM) {
    stabs = __STAB_BEGIN__;
    stab_end = __STAB_END__;
    stabstr = __STABSTR_BEGIN__;
    stabstr_end = __STABSTR_END__;
} else {
```

这里就是调试信息是如何被赋值到 stab，然后在使用 stab_binsearch 先后查找文件、文件当中某个函数，最后查找函数所在的行号，并确定它的参数个数，参数默认的类型为 N_PSYM，只需比对 stab 的 n_type 是否为该类型并计数即可。

下面我们进入 lab2 的正题：

调用 i386_init()之前的 Jos 内存 layout



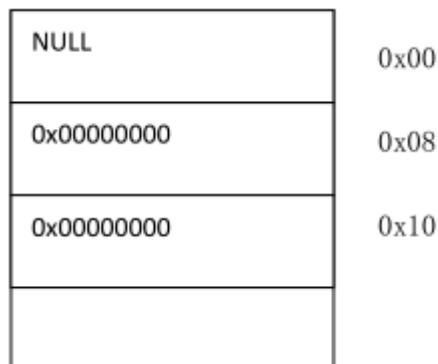
在 kernel 启动后，它将段的基址改到了 KERNBASE = 0xF0000000，通过保护模式的段式地址转换机制 $Pa = La + base$ 。将程序内 KERNBASE 开始的逻辑地址转换成物理地址。本节划分为页面管理和页表管理。页面管理：内核如何在物理内存中分配空间和建立合理的数据结构进行对物理页面的管理。页表管理：建立两级页目录(Page directory)和页表(Page tables)，在开启 x86 的分页管理后仍能够进行合理的逻辑地址到物理地址的转换。

对固定页面大小的讲解： $32 = 10+10+12$ ，12 位可以组成一个页面的业内位移，前面也是两级目录值，又因为 32 位为 4 个字节，故每一个 4KB 的页面刚好放下 2^{10} 个页号索引，故对 32 位进行 10+10+12 这样的划分。在 mmu.h 当中：

```
// A linear address 'la' has a three-part structure as follows:
//
// +-----+-----+-----+
// | Page Directory | Page Table | Offset within Page |
// |     Index      |     Index   |           |
// +-----+-----+-----+
// \--- PDX(la) --/ \--- PTX(la) --/ \--- PGOFF(la) ----/
// \----- PPN(la) -----/
//
// The PDX, PTX, PGOFF, and PPN macros decompose linear addresses as shown.
// To construct a linear address la from PDX(la), PTX(la), and PGOFF(la),
// use PGADDR(PDX(la), PTX(la), PGOFF(la)).
//
// page number field of address
#define PPN(la) (((uintptr_t)(la)) >> PTXSHIFT)
#define VPN(la) PPN(la) // used to index into vpt[]
// page directory index
#define PDX(la) (((uintptr_t)(la)) >> PDXSHIFT) & 0x3FF
#define VPD(la) PDX(la) // used to index into vpd[]
// page table index
#define PTX(la) (((uintptr_t)(la)) >> PTXSHIFT) & 0x3FF
// offset in page
#define PGOFF(la) (((uintptr_t)(la)) & 0xFFFF)
// construct linear address from indexes and offset
#define PGADDR(d, t, o) ((void*) ((d) << PDXSHIFT | (t) << PTXSHIFT | (o)))
```

Jos 的内存结构：

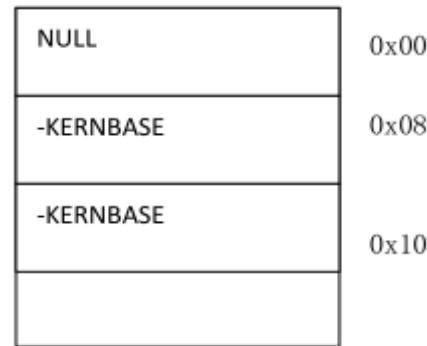
早在 boot.S 当中，在从实模式转换到保护模式后，加载了 gdt 如下：



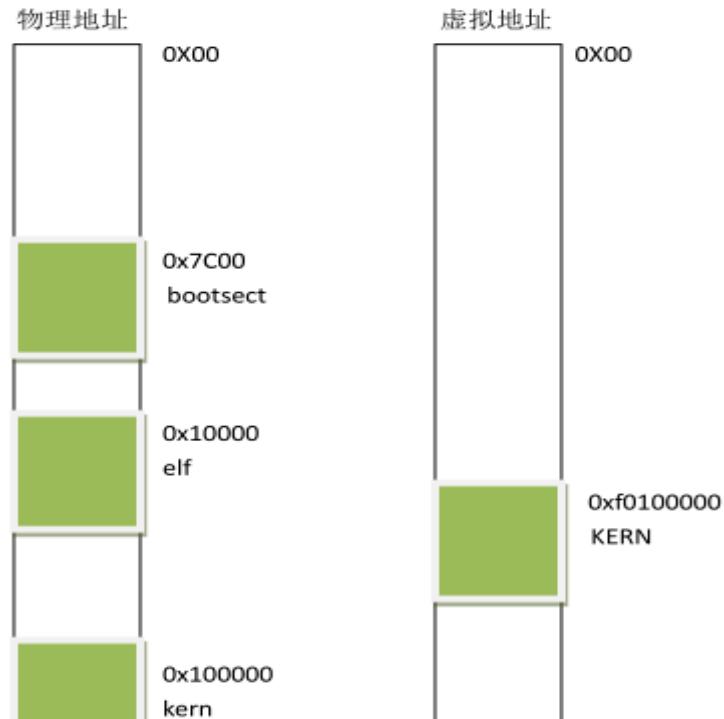
然后系统跳转到 main.c，由于段选择符为 0x08（?），所以基地址为 0x00000000。

Jos 再将 kernel 载入物理空间 0x00100000，但它的逻辑地址实际上为 0xF0100000，但是由于在 readseg 将虚拟地址和 0x00FFFFFF 相与，将最高位的 F 去掉了，故实际载入到物理地址的 0x00100000，在 bootmain.c 的最后开始执行 entry.S。

Kernel 启动后，重新加载了自己的段信息，并给出了新的基地址，为 KERNBASE，故加载到的 gdt 如下：



然后跳转到 CODE_SEL，从现在开始，段基地址变成 0xF0000000，所以所有虚拟地址 (0xF0100000++) 的地址，经过段转换之后都会变成 0x00100000++。



Part 1: Physical Page Management

Exercise 2. In the file kern/pmap.c, you must implement code for the following functions.

```
boot_alloc()
page_init()
page_alloc()
page_free()
```

You also need to add some code to i386_vm_init() in pmap.c, as indicated by comments there. For now, just add the code needed before the call to check_page_alloc().

check_page_alloc() tests your physical page allocator. You should boot JOS and see whether check_page_alloc() reports success. Fix your code so that it passes. You may find it helpful to add your own assert()s to verify that your own assumptions are correct.

我们使用一个链表结构 Page，每一个 Page 都是代表一个物理内存页面。一般的课本上说内存也可由系统决定，1KB、2KB、4KB 或者 8KB，但是，在 32 位的 i386 上，页面大小固定为 4KB。在 Extended Paging 模式下，为了寻址恒大的空间，页面的大小甚至可以为 4MB。

在分页的时候，既不能把实模式加载的数据、系统区域(BIOS、显存区域)、内核本身等分配出去，还要建立起合适的映射，达到将逻辑地址转换到物理地址的目的。

这样，页面的数目 npage = 物理内存大小>>12

1) 页面管理链表结构

```
LIST_HEAD(Page_list, Page);
typedef LIST_ENTRY(Page) Page_LIST_entry_t;

struct Page {
    Page_LIST_entry_t pp_link; /* free list link */

    // pp_ref is the count of pointers (usually in page table entries)
    // to this page, for pages allocated using page_alloc.
    // Pages allocated at boot time using pmap.c's
    // boot_alloc do not have valid reference count fields.

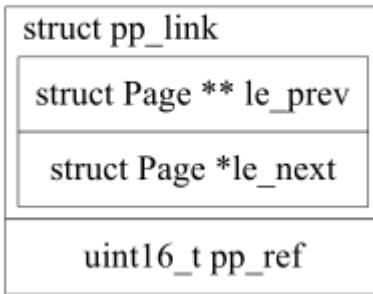
    uint16_t pp_ref;
};
```

我们直接在 queue.h 当中找到 LIST_ENTRY 的定义，综合可以写出 Page 的结构：

```
struct Page {
    struct {
        struct Page *le_next;
        struct Page **le_prev;
    } pp_link;
    uint16_t pp_ref;
};
```

用我们书上熟悉的表示方法进行表示为：

```
struct Page
```



页面管理双向链表结点结构

`le_next` 指向下一个 Page 的页面, 一个指向指针的指针 (后文说明) 以及一个 short int 型的 `pp_ref`。另外, 系统还定义了一个链表头, 代码前文有详细地说明。

```
#define LIST_HEAD(name, type)
struct name {
    struct type *lh_first; /* first element */
}
```

而在 `pamp.c` 当中定义的 `page_list` 实际上也只是包含了一个 `Page` 结构的指针 `lh_first`。而 `page_free_list` 这个定义在 `pmap.c` 定义的全局变量实际上就是指向页面管理的双向链表的头结构, 它是一个结构体, 而这个结构体只有一个 `Page` 型的指针。

在 `I386_detect_memory` 当中, 执行该段得到一些内存基本信息。

我们直接启动系统就可以看到这一段信息:

```
the lab2 check begin
Physical memory: 32768K available, base = 640K, extended = 31744K
```

我们可以看到, 传统内存为 640KB, 扩展内存为 31744KB, 整个物理内存为这两者的和, 为 32768KB,

在 `queue.h` 当中又定义了一些宏:

对链表头进行操作

```
LISH_FIRST(head) ((head)->lh_first) //取得头指针
LIST_INIT(head) do {
    LIST_FIRST((head)) == NULL;
}while(0) //将链表重置
LIST_EMPTY(head) ((head)->lh_first == NULL) //和 init 一样, 据说系统没用过
LIST_HEAD_INITIALIZER(head) {NULL} //将链表头自身赋值为空, 据说也没用
操纵节点和对该双向链表进行操作
LIST_NEXT(elm, field) ((elm)->field.le_next)
//该宏返回 elm 所在的下一个页面管理节点的地址。使用的时候, elm 应该作为一个指向页面管理节点的指针, field = pp_link
```

LIST_INSERT_HEAD:

将 `elm` 指向的页面管理节点成为整个页面管理双向链表链表的第一个元素, 在实现上, 是要求链表头结构(`page_free_list`)的 `lh_first` 指针指向该结构。当链表为空, `lh_first` 为空。以前链表不为空, 这种情况下链表头的 `lh_first` 不为空。这里的宏可以有效地处理这些情况。(详见 `queue.h`)

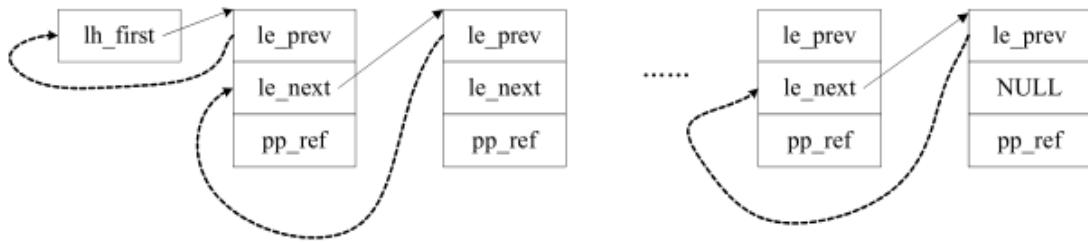
LIST_INSERT_BEFORE:

LIST_INSERT_AFTER:

这两个宏的作用就是把新的结点(`elm` 参数), 插入到结点 `liselm` 之前或者之后。对于 `Page` 结构的 `pp_link.le_next` 指针的使用, 应该比较容易理解, 需要解释的是 `pp_link.le_prev` 的使用。在这里, `le_prev` 的作用是指向前一个结点的(`(elm)->pp_link.le_next`)的地址。

下面图示页面管理的链表结构:

page_free_list



LIST_REMOVE:

该宏所做的工作是将 elm 指向的结点从页面管理链表中消除。这里要注意 pp_link.prev 的用法，它只是用来 index 前一个结点的 le_next 的地址，不能索引到整个前面的结点，这一项可以修改 le_next 的值到当前结点的 le_next 值，从而达到删除结点的作用。

LIST_FOREACH:

这个宏是一个 for 循环，它的作用是遍历页面管理链表。

关于链表的操作清楚之后，我们就可以开始做题了

浏览一下 pmap.c 文件，boot_freemem 和 boot_pgdir，前者是当前可用内存开始的地址（即内核载入以后的地址空间就是系统管理的空间从内核结束就开始了）。boot_pgdir 则是系统页目录所在空间的开始地址。两者都是虚拟地址。

阅读 i386_vm_init 代码，注意阅读注释的提示。pde_t 看上去是什么类型不熟悉，使用 cscope 追过去一看就知道他其实就是 32 位 int 型(应该是为了名称特殊化吧)。

第一个是 boot_alloc 的实现，它的功能是分配内存空间地址，并清空分配的地址段。然后将物理地址放入 boot_cr3，准备启动 x86 的页面转换机制。PGSIZE 为一个物理页的大小，为 4096B，PTSIZE 为一个页表对应实际物理内存的大小，为 $1024 \times 4KB = 4MB$ 。得到页面没有初始化，需要我们使用代码进行初始化。在清空的时候使用 memset 一定要取实际物理页对应的虚拟地址。还有就是 boot_cr3 请求一个物理地址。

```

#define ROUNDDOWN(a, n)
{
    uint32_t __a = (uint32_t) (a);
    __a -= __a % (n);
}
// Round up to the nearest multiple of n
#define ROUNDUP(a, n)
{
    uint32_t __n = (uint32_t) (n);
    __n = ROUNDDOWN((uint32_t) (a) + __n - 1, __n);
}
  
```

ROUNDDOWN 返回 ROUNDDOWN(a+n-1,n)。

而后者返回 $(a+n-1)-(a+n-1)\%n$ ，等价于 $((int)(a/n) + 1)*n$ (0 除外，但是 boot_freemem==0 在前面进行过判断了，所以这里的 boot_freemem 肯定不为 0)。

学术地说就是使分配空间的起始地址对齐，返回的地址一定不小于 boot_freemem 本身。这样使得之后分配的空间都是 4KB 对齐。根据 boot_alloc，我们还知道这里的页表管理链接表实际上只有 4KB（一个 Page，这里应该是可以动态调整的，但是既然 mit 都这么直接给了，就不去改它吧）。**关于一个全局变量 pages，它接受 boot_alloc 返回的地址。**

Pages 可以使用下标来唯一的表示一个物理页，再加上前面提到的双向链表，一共两种方式进行管理，前者直观且具有确定性，后者是修改灵活，同时还可以通过 pp_ref 知道该页面被引用或者情况。

对 pgdir 的操作其实读注释也能懂得差不多了，大意就是在 VPT 这个虚拟地址建立虚拟页表，并使得 kernel 可读写，用户什么权限都没有。UVPT 类似。

从 pages = boot_alloc(npaged * sizeof(struct Page), PGSIZE) 开始就是真正开始分配页。由前文我们知道系统一共提供了 npaged 的空间了，也确认前面的 4KB 给页表管理链接表。

分配好空间了，就进入了 page_init 了。这个函数的主要功能是把已经有过数据的内存部分标记为使用，然后将他们从空闲列表中删去。Jos 的内存编址的规定，早在全文我们就提到了，就是一个 10+10+12 的分配过程，这样可以使得每个页表所占空间也是 4KB，刚好为一个物理页，从而容易构建多级页目录。这里我们简单回顾一下宏定义的功能：

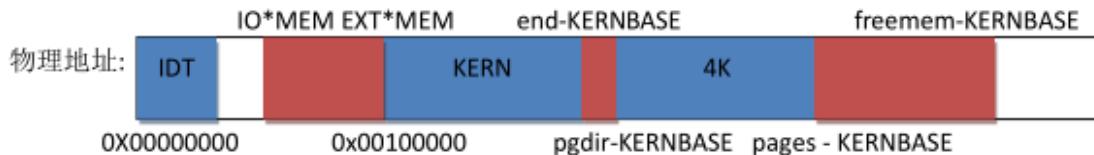
其实注释上也写的很清楚

VPD = PDX //page directory index, used to index into vpd, 目录的索引

PTX //page table index, 页表索引

PGOFF //offset in page, 页内位移

VPN = PPN // page number field of address, used to index into vpt, pages 数组中的索引；页号，我们使用物理地址取 PPN，使用返回值可以访问物理地址在 pages 中的对应页。



init 就是将空闲链表初始化后，把有颜色的部分标记后，从空闲链表中移去，使用 for 循环即可。

```
#define IOPHYSMEM    0xA0000
#define EXTPHYSMEM   0x100001
```

按照上图的地址将已被占用的空间从链表中删去，删去的方法：

pages[i/PGSIZE].pp_ref = 1;

LIST_REMOVE (&pages[i/PGSIZE], pp_link) (?) 看上去是一个全局变量。

接下来我们就进入了 page_alloc，这个函数就是用来分配一个页，然后从空闲链表删除一个页，然后返回 0，如过分配失败(失败就是 page_free_list 为空)，返回-E_NO_MEM。按照注释来做，可以不会忽视当可分配空间为 NULL 时返回-E_NO_MEM。

然后是 page_free，这里以防万一可以检查一下 pp_ref 属性，并将它插回到空闲队列。茶会队列使用 LIST_INSERT_HEAD 即可。

Part 2: Virtual Memory

Exercise3：阅读 x86 的保护模式下的内存管理，页面转换是 Jos 当中占有大比重的部分。你童颜需要了解一下段在保护模式下是如何工作的。

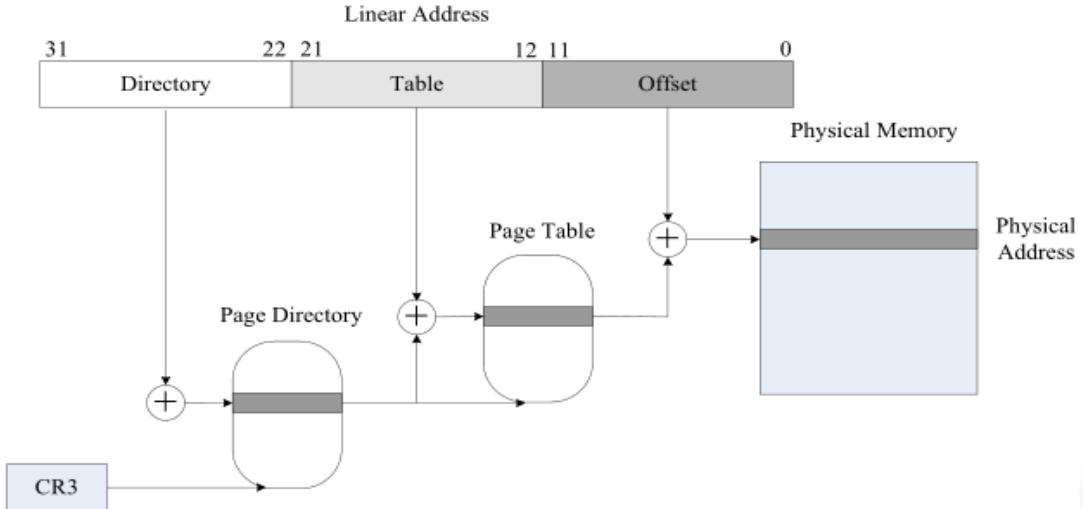
定义三种地址：

逻辑地址：程序在编译连接后，变量的名字等的符号地址，在 Jos 系统中内核的部分，改地址十一 KERNBASE(0xF0000000，可以修改)。

线性地址：经过 x86 保护模式的段式地址变换后的地址，变化过程为逻辑地址加段首地址。

物理地址：内存存储单元的编址，比如 1GB 的内存，物理编址是从 0x00000000 到 0x40000000。

x86 的段式地址变换，就是逻辑地址到线性地址的转换过程。如果未开启页式内存管理，得到的线性地址就是物理地址，开启之后则需要通过页式地址变换才可以得到物理地址。我们之前提到过将 32 划分为 10+10+12，详情分布则如图示：



无论是在之前做 `page_init`, 还是做用 `boot_alloc` 直接划分的时候, 我们知道页目录只占用了 4KB 的物理页面。而他在内核实际上备份做 1024 个单元。因为页目录索引和页表索引都只有 10 位, 只有 1024 中不同的情况。每个单元占用 4 个字节:

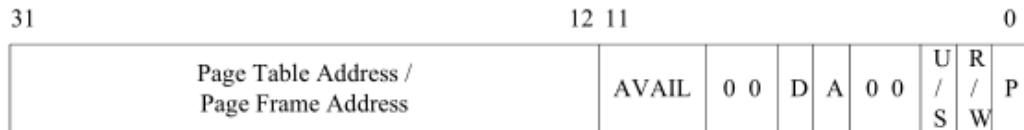


图 4-6. 页目录(表)项的格式

表项中的高 20 位地址能够定位到内存的任何一个物理页面的首地址。如果是在页目录下, 则找到的是下级页表项的物理页面首地址。如果是页表中找到就是数据页的首地址。

P 判断对应的物理页是否存在

R/W 判断对所指向的物理页面的访问权限, 1 表示可写, 0 表示只读

U/S 该位用来定义页面的访问者应该具备的权限。1 是 User, 2 是只能在 Ring0 中的程序中访问

D 是否被修改过

AVAIL 可以被系统程序使用

0 保留位, 不能使用

页目录所在的物理地址在启动 x86 时, 就已经放入 cr3(boot_cr3)。进行也是转换的时候自动从 cr3 获取页目录地址。通过线性地址前 10 位, 我们在页目录所在的 4KB 当中找到对应的页目录表项, **通过其高 20 位可以得到页表所在的物理页首地址, 从而找到页表**。然后用页表项前 20 位加上线性地址的页内偏移。

对于现在操作系统, 很多都是直接为用户进程创建页目录, 并且保护到用户进程的上下文, 每次切换的时候直接将该页目录写入 cr3, 并重启页式内存管理。这种方式占用更多的内存进行页式地址变换。Jos 中为了避免太大的开销, 整个系统只是用一个页目录项。整个系统的线性地址只有 4GB。且所有的进程之间都是公开, 故需要进行权限限制。

另一方面, 由于页目录和页表存放在物理内存的页面中, 要进行地址变换就势必先要到内存中访问页目录和页表。由于 CPU 和内存速度不匹配, 大大降低了系统的效率。在系统结构课程中, 我们学到过 TLB(快表), 就是讲页目录和页表放入 cache 当中进行访问, 从而提高访问速度。这里邵老师介绍是在内存中进行修改, 无法在 TLB 中反映, 所以必须是的 TLB 的数据失效, 一个是重载 cr3, 另一个是使用 `invlpg` 指令。在系统结构中, 应该有提到过写会和脏位(D)结合使用的方法, 就是直接修改 cache 的数据, 并标记脏位, 之后再策略替换该修改项时, 根据脏位的情况写会到内存, 来保证数据的正确性。

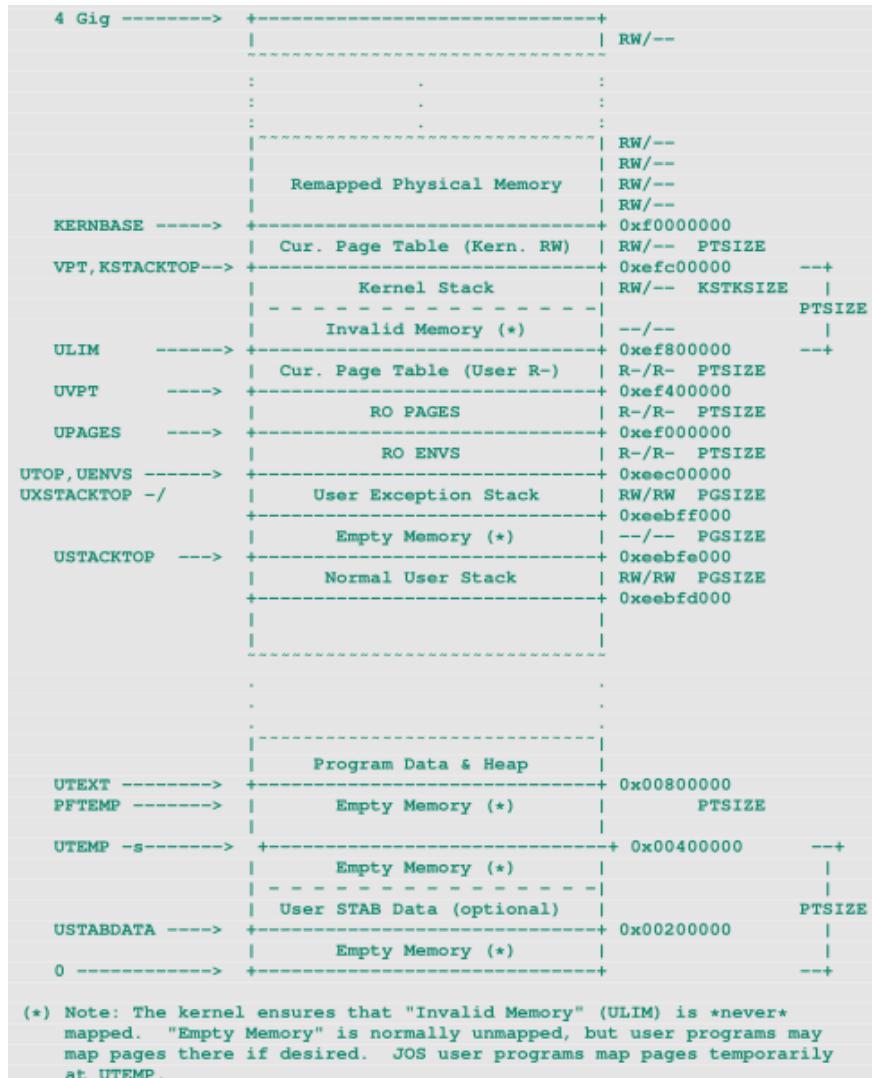
Virtual, Linear, and Physical Addresses

Question

1. Assuming that the following JOS kernel code compiles correctly and doesn't crash, what type should variable `x` have, `uintptr_t` or `physaddr_t`?

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

在内核中操作数据都是以内核虚拟地址进行的，所以应该是 `uintptr_t`，实际上，因为后面一个明显是和物理地址相关的，排除就是另外一个了。



以上内容可以在 `memlayout.h` 中找到的虚拟内存布局。对其组成成分进行分析：

[USTABDATA, UTEXT]: stab 相关的数据

[UTEXT, USTACKTOP]: 用户.test 段

[USTACKTOP, UPAGES]: 用户堆栈共用区域

[UPAGES,UVPT]：在虚拟地址中这段内存就是对应的在实际物理地址里 `pages` 数组对应储存位置。（在后面的代码部分我们可以看到相应的操作语句）。这段地址在 `ULIM` 之下，也就是说操作系统开放 `pages` 数组便于让用户程序可以访问。为什么呢？比如说假如有的程序想要知道一个物理页面被引用了多少次，那么根据相应的 `pages[i].pp ref` 就可以知道了。我们看到这个区域在布局中分配了 `PTSIZE` 的大小，那么这个大小够么？因为我们知道在物理页面中 `pages` 所占用的大小为 $npage \times \text{sizeof}(\text{struct Page}) = 12B \times npage$ 。我们看看 `PTSIZE` 的空间可以装 `struct Page` 的个数为 $1024 \times 4KB / 12B = 4MB / 12B = 13M$ 。一个 `Page` 结构对应一个实际大小为 4KB 的物理页面，所以这个 `PTSIZE` 大小的虚拟地址空间能够管

理 $13M \times 4KB = 1.3GB$ 的物理内存。

[VPT,KERNBASE]: VPT 的部分被设置为用户不可见，系统可读写。其他和下面类似。

[UVPT,ULIM]: 这两个地址映射的是同一个系统页目录，即原来在程序中看到的变量 pgdir，开放给用户只读，可以使用户得到当前内存中某个虚拟地址对应的物理页面地址是多少。这段空间在虚拟地址上分配了 PTSIZE，但是实际上只使用了物理页面上的 PGSIZE 个空间（可以回头去看看 pgdir 的空间分配参数是多少）。

对 pgdir 的操作有 VPT、UVPT 两部分，对于 UVPT 来说，权限上的设置是 kernel 和 user 都是可读的。其对应到的虚拟地址在系统页目录中的表项设置为物理地址。也就是说一个用户程序向访问页目录的话，只要把对应的虚拟地址设置为 UVPT 即可。只要有页目录地址就可以查找一个任意虚拟地址所在的页面的物理地址和其对应的二级页表物理地址。如果要查询的虚拟地址是 addr， $= PDX|PTX|OFFSET$ 的话，显然只有 PDX|PTX 决定了 addr 物理页面的地址。

查找方法：

1、addr 对应的物理页面地址

构造虚拟地址 vaddr = UVPT[31:22]|PDX|PTX|00 在虚拟内存空间查询。

根据页表转换原则

- 1) vaddr 的前十位，即 UVPT[31:22]，在页目录当中进行查询，但是由于该十位等价于 PDT(UVPT)，所以得到的二级页表地址还是页目录 pgdir 本身。
- 2) 再取出 vaddr 中间 10 位，即 PDX，在 pgdir(页目录)中，得到 addr 所在目录项，通过该目录项，我们得到 addr 所在页表首地址，这个是物理地址。
- 3) 再取出最后的 12 位，PTX|00，在页表当中，每个页表为 4 个字节，故 PTX|0 相当于 4 字节一跳索引到 PTX 处，这实际上就获取了页表项。然后就可以得到 addr 所在页面的物理地址。

2、addr 对应的二级页表物理地址：

同样地，构造 vaddr = UVPT[31:22]|UVPT[31:22]|PDX|00

与之前相似地，实际上最终只找到了前面第 2 步就结束了，通过 PDX|00 和 pgdir 的物理页得到的就是二级页表的物理地址。

[KERNBASE,4GB) :

这个部分映射实际物理内存中从 0 开始的中断向量表 IDT、BIOS 程序、IO 端口以及操作系统内核等。即内核使用的虚拟地址 KERNBASE + x 进入这段地址查找到实际上的物理地址就是它自身。所以这段空间是为操作系统准备的。

原理过后，开始做题：

Exercise 4. In the file kern/pmap.c, you must implement code for the following functions.

```

pgdir_walk()
boot_map_segment()
page_lookup()
page_remove()
page_insert()

page_check(), called from i386_vm_init(), tests your page table management routines. You should make
sure it reports success before proceeding.

```

在 pmap.h 当中，我们先看一些有用的定义：

因为每一个物理页面对应一个 Page 的 struct 和一个物理页号 PPN 和唯一的物理首地址，Jos 里面提供了一些内联函数和一些宏定义来联系这些关系

- 1、一个 Page 的 PPN 就是 page2ppn(Page)，而一个 PPN 对应的 Page 则是 pages[PPN]
- 2、一个 Page 对应的物理地址为 page2pa(Page)，而一个物理地址对应的 Page 则就是 pa2page(pa)

3、page2kva(就是 KADDR)是提供了一个页面到内核虚拟地址的转换。

PADDR 的功能与 KADDR 相反。

`pgdir_walk` 函数的功能就是根据线性地址，返回线性地址对应的二级页表的页表项的虚拟地址。就是 `pgdir` 的每个页表项当中实际包含有二级页表的物理地址(10 bits)，每个二级页表当中又包含有每个物理页的物理地址(10 bits)。但是在 `Jos` 当中进行页面管理时，是通过虚拟地址来进行管理的，所以 `pgdir_walk` 返回一个之前二级页表项的虚拟地址。

先是根据 `va` 的前 10 位和页目录定位到页目录项 `pt`，然后判断该目录项的 `PTE_P` 位：

如果为 1，则说明 `va` 已经可以使用页表翻译，直接使用 `KADDR(PTE_ADDR(*pt))` 和 `PTX(va)` 返回器对应的页表项的地址即可，如果不能，就看 `create` 是否为 1，如果为 1 且 `page_alloc` 返回成功的话，就可以创建对应的页表，并返回 `va` 现在对应的页表项的地址，否则就返回 `NULL`。还有 `memset` 的一定是虚拟地址(基于 `KERNBASE` 的)。

至于 `PTE_U` 等等权限的设置可以参看前面的说明，或者是定义代码后面的注释都有详细的说明，依次检查段权限为 `DPL`、页目录对应表项访问权、页表相应表项访问权。

`boot_map_segment` 函数主要就是讲线性地址和物理地址进行绑定，本质上就是在线性地址对应的二级页表当中写入物理地址对应的物理页的地址就可以了。如果你认认真真阅读了注释的话，我相信你即便没看懂前面的内容，这个也是不会出问题的，主要的不周就是先把一共要分配的空间先 4KB 对齐，然后使用 `pgdir_walk` 对每个页进行分配，建议每次分配好一页后都可以检查一下，防止内存分配失败发生返回值为 `NULL` 的情形。

回到 `i386_vm_init`，这里需要用到 `boot_map_segment` 的使用情况：

`[UPAGES, sizeof(PAGES)] => [pages, sizeof(PAGES)]`

这是页面管理结构需要的空间

`[KSTACKTOP-KSTKSIZE, 8] => [bootstack, 8]`

`Bootstack` 为内核编译时预留的 8 个页面做堆栈

`[KERNBASE, pages in the memory] => [0, pages in the memory]`

几乎涵盖所有物理内存。

`page_lookup` 在页式地址翻译机制中查找线性地址 `va` 所对应的物理页面，如果找到，则返回该物理页面，并将对应的页表项的地址放到 `pte_store` 中；如果找不到或找到的页表项的 `PTE_P` 为不允许访问，则返回空。

`page_remove` 则是删去线性地址对应的页。注释里提示使用 `page_decref`，这个函数是将 `pp_ref` 降低，而没有真正将页面删去。同时因为页表项发生了修改，删除完成后，调用 `tlb_invalidate` 函数则更新 TLB 快表，这里我们之前也提到过，实质上是使用 `invlpg` 函数。

`page_insert` 是将页面管理的结构 `pp` 所对应的物理页面分配给线性地址 `va`，同时将对应的页表项的 `permission` 设置成 `PTE_P&perm`。这是要考虑线性地址 `va` 已经指向了另外的物理页面。需要调用 `page_remove` 将该物理页从线性地址 `va` 删除，在将 `va` 对应的页表地址复制为 `pp` 对应的物理页面。如果 `va` 本来就是参数 `pp` 对应的物理页面，则将 `va` 对应的也变相中的物理地址重新复制为 `pp` 所对应的物理页面的首地址即可。考虑已挂载页面和当前分配的物理页面是同样的情况时，因为修改权限并没有增加一个物理页面的引用数目。所以对这种情况我们不应该对应一般情况对 `pp_ref` 进行+1 操作。不过加不加好像都不影响测试结果。

Part 3: Kernel Address Space

Exercise 6. Fill in the missing code in `i386_vm_init()` after the call to `page_check()`.

Your code should now pass the `check_boot_pgdir()` check.

对于三段 `boot_map_segment` 已经在完成该函数的时候填好了。

Answer these questions:

1. What entries (rows) in the page directory have been filled in at this point? What addresses do they map and where do they point? In other words, fill out this table as much as possible:

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. After `check_boot_pgdir()`, `i386_vm_init()` maps the first four MB of virtual address space to the first four MB of physical memory, then deletes this mapping at the end of the function. Why is this mapping necessary? What would happen if it were omitted? Does this actually limit our kernel to be 4MB? What must be true if our kernel were larger than 4MB?
3. (From Lecture 4) We have placed the kernel and user environment in the same address space. Why will user programs not be able to read or write the kernel's memory? What specific mechanisms protect the kernel memory?
4. What is the maximum amount of physical memory that this operating system can support? Why?
5. How much space overhead is there for managing memory, if we actually had the maximum amount of physical memory? How is this overhead broken down?

1、每页 4KB，推算可以 va 表项的数据，然后根据虚拟地址到物理地址的转换填写对应表项即可。

2、重置页目录的第一项，并打开页表转换机制，之前是没有页式地址转换，虚拟地址经过一次段地址转换就得到物理地址了，之后线性地址 laddr 到物理地址 paddr 的页式转换才打开。这时页表的第一项 pgdir[0]被设置成 pgdir[PDX[KERNBASE]]，所以内核访问 vaddr=KERNBASE+ x 都先经过一次段式转换得到 laddr = vaddr - KERNBASE = x，然后 laddr 经过页式转换等价于访问原来虚拟地址 KERNBASE 上面的同样地址。把 pgdir[0]重设的目的就是让内核地址在打开页式转换以后仍然能找到其自己所在的位置，因为后面如内联汇编代码需要使用到内核中定义的常量 GD UD 等，内核在引用它们时都是使用的虚拟地址。之后关闭段式地址转换把原来是-KERNBASE 的段基址赋值为零即可。这个时候一个内核虚拟地址转化为的线性地址 laddr 就是 KERNBASE + x 本身，进入页式转换，查找到的就是内核其自身的物理地址。内核已经能只依靠页式转换实现正常访问其自身了，那么这时虚拟地址的低 4MB 的设置已经没用，需要还原清空，然后重置 TLB。

3、用户当然不能写内核区域，使用特殊为进行控制。

```
#define PTE_P      0x001 // Present
#define PTE_W      0x002 // Writeable
#define PTE_U      0x004 // User
#define PTE_PWT    0x008 // Write-Through
#define PTE_PCD    0x010 // Cache-Disable
#define PTE_A      0x020 // Accessed
#define PTE_D      0x040 // Dirty
#define PTE_PS     0x080 // Page Size
#define PTE_MBZ    0x180 // Bits must be zero
```

4、UPAGES 为 pages 的空间、系统分配的虚拟空间为 PTSIZE，对应能管理的空间为 1.3G 左右。

5、内存管理内存开销为 1 个页目录+1024 个页表，为 1025*4KB，为 4M+4K 大小

重要附录参考代码

```

page_alloc(struct Page **pp_store)
{
    // Fill this function in
    if(LIST_FIRST(&page_free_list) == NULL){
        return -E_NO_MEM;
    }
    else{
        *pp_store = LIST_FIRST(&page_free_list);
        //page_initpp(*pp_store);
        LIST_REMOVE(*pp_store, pp_link);
        return 0;
    }
}

```

```

page_free(struct Page *pp)
{
    // Fill this function in
    if(pp->pp_ref) return;
    else{
        page_initpp(pp);
        LIST_INSERT_HEAD(&page_free_list, pp, pp_link);
        return;
    }
}

```

```

pgdir_walk(pde_t *pgdir, const void *va, int create)
{
    // Fill this function in
    pde_t *pt = pgdir + PDX(va);
    if(*pt & PTE_P){//pt[PTE_P] == 1
        return (pte_t *)KADDR(PTE_ADDR(*pt)) + PTX(va);
    }
    struct Page *pg;
    if(create == 1 && page_alloc(&pg) == 0){
        pg->pp_ref = 1;
        memset(page2kva(pg), 0, PGSIZE); //pg turn to the va to be init
        *pt = PADDR(page2kva(pg))|PTE_U|PTE_W|PTE_P;//same with the begin
        return (pte_t *)KADDR(PTE_ADDR(*pt)) + PTX(va);
    }
    return NULL;
}

```

```

page_insert(pde_t *pgdir, struct Page *pp, void *va, int perm)
{
    // Fill this function in
    pte_t *pte;
    pte = pgdir_walk(pgdir, va, 1);
    if(pte == NULL){
        return -E_NO_MEM;
    }
    else{
        pp->pp_ref++;
        if((*pte & PTE_P)){
            //if(PTE_ADDR(*pte) == page2pa(pp)) pp->pp_ref--;
            //else
            page_remove(pgdir, va);
        }
        *pte = page2pa(pp) | PTE_P | perm;
        return 0;
    }
    return 0;
}

```

```

boot_map_segment(pde_t *pgdir, uintptr_t la, size_t size, physaddr_t pa, int perm)
{
    // Fill this function in
    uint32_t i;
    pte_t *pte;
    size = ROUNDUP(size, PGSIZE);
    for(i=0;i<size; i += PGSIZE){
        pte = pgdir_walk(pgdir,(void *)(la+i), 1);
        if(pte == NULL){
            assert(pte!=NULL);
        }
        *pte = (pa+i)| perm | PTE_P;
    }
}

page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
    // Fill this function in
    pte_t *pte;
    pte = pgdir_walk(pgdir, va, 0);
    if(pte == NULL||(*pte & PTE_P) == 0){
        return NULL;
    }
    if(pte_store != NULL){
        *pte_store = pte;
    }
    return pa2page(*pte);
}

page_remove(pde_t *pgdir, void *va)
{
    // Fill this function in
    struct Page *pg;
    pte_t *pte;
    pg = page_lookup(pgdir, va, &pte);
    if(pg ==NULL){
        return;
    }
    else{
        page_decref(pg);
    }
    if(pte != NULL){
        *pte = 0;
    }
    tlb_invalidate(pgdir, va);
}

```

User Environments

Author: Xin Yao

目标：实现进程的管理和中断功能。

大概可以划分为两个部分：进程环境和中断处理。

前者设置进程控制块、进程创建、进程终止和进程调度程序，实现如何对进程进行管理

后者设置中断描述符表 IDT，使用通用中断派发程序，实现如何管理中断。

本实验中 Jos 只创建了一个进程，进程执行特权指令时，需要处理器产生中断，从用户态切换到内核态，完成任务后中断返回到用户态。其实 Jos 对进程的管理和物理页面时一样的，都是通过双向链表来管理的。

进程的结构在 inc/env.h 中规定，每个 Environment 对应于一个进程：

```
struct Env {
    struct Trapframe env_tf; // Saved registers
    LIST_ENTRY(Env) env_link; // Free list link pointers
    envid_t env_id; // Unique environment identifier
    envid_t env_parent_id; // env_id of this env's parent
    unsigned env_status; // Status of the environment
    uint32_t env_runs; // Number of times environment has run

    // Address space
    pde_t *env_pgd; // Kernel virtual address of page dir
    physaddr_t env_cr3; // Physical address of page dir
};
```

同样地，我们可以找到类似的定义：

LIST_ENTRY(Env) env_link; //free list link pointers (和内存同样的结构)

struct Env

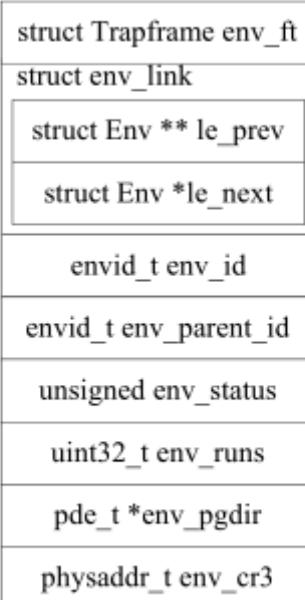


图 5-1. 进程管理双向链表结点结构

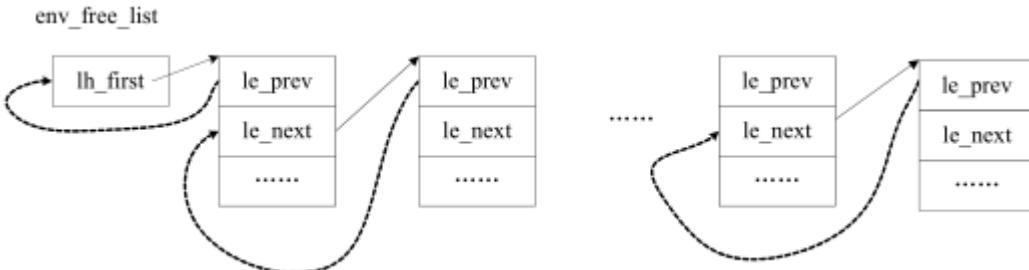
该结构存在 9 个成员：一个 CPU 现场保护信息 env_tf(保护进程切换时此时进程的寄存器的信息)、指向链表下一个结点的指针、一个指向指针的指针、两个进程描述信息(PID)、进程控制信息 env_status&env_runs，以及地址空间 env_pgd 和 env_cr3。

寄存器的结构在 int/trap.h 中规定：进程有三个状态，在 kern/env.h 中规定：

FREE、RUNNABLE、NOT_RUNNABLE。env_pgd 保存进程页表的虚拟地址，env_cr3

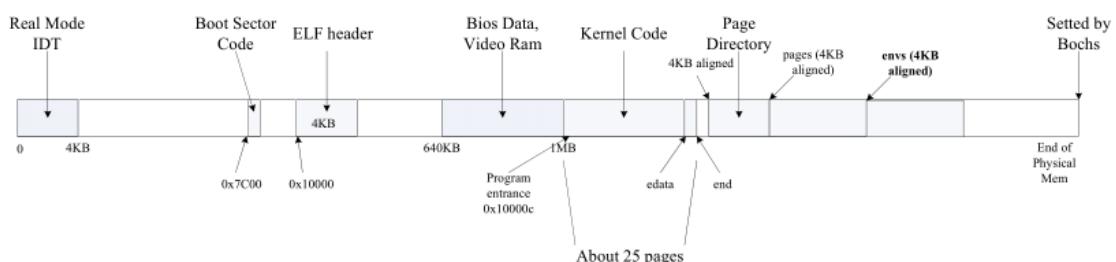
保存进程页表的物理地址。

Jos 使用一套宏定义，以类似于实验 2 对内存页式管理的方式，将 PCB 组织在一起，将他们放在内存中的固定区域，即 PAGES 页面管理结构所在之后的 4KB 空间。由此我们容易得到 Env_list 的定义，和 Page_list 的结构一样。同样的，我们可以得到其管理链表结构：



同样地，我们需要为它分配内存，使用 boot_alloc 在 i386_vm_init 当中我们需要对 env 分配管理空间。所分配的位置和上下文我们容易知道是在 pages 后面的 4KB 空间当中。

系统定义了两个全局变量 *envs 和*curenv，boot_alloc 返回值保存在 envs 当中。



envs 的用法和 pages 是一致的，对于进程管理和组织的两种方法和 pages 一样：一是通过 envs 的下标，另一种是通过双向量表。curenv 是指向当前运行的进程没在进程切换的时候会使用到。一般操作系统都可以多进程并发执行，这取决于 PCB 表的大小，Jos 当中就是 envs 数组的 size。一共有 1024 个表项，详见 inc/Env.h

```

#define LOG2NENV      10
#define NENV          (1 << LOG2NENV)
#define ENVX(envid)   ((envid) & (NENV - 1))
  
```

之后我们还需要在 pmap.c 当中使用 boot_map_segment 来进行一个物理地址和线性地址的映射。

Jos 中，每个进程都有代码段、数据段、用户栈、进程属性，由于无文件系统，系统将用户进程编译连接成原始的 ELF 二进制文件映像，所以用户进程在编译的时候就放入到内核当中了。系统在加载时读取的对象为 ELF 格式文件。

Part A: User Environments and Exception Handling

让我们先根据 2007 年 Mit 提供的材料进行一些了解：

我们之前提到过 Env 结构体，Mit 提供了更为详细的资料说明，tf 是保存当前进程使用的寄存器的值；link 指向 env_free_list，id 为内核为该结构分配的一个编号；parent_id 为创建该进程的进程 id 号；status 为进程的状态，现在只有三个；pgdir 为该进程的虚拟地址的目录；cr3 为该进程物理地址的目录。

和 unix 一样，Jos 也有线程和地址空间。线程的标志是 env_tf，地址空间的标志是 env_pgdir 和 env_cr3。

Exercise 1. Modify i386_vm_init() in kern/pmap.c to allocate and map the envs array. This array consists of exactly NENV instances of the Env structure, laid out consecutively in the kernel's virtual address space starting at address UENVS (defined in inc/memlayout.h). You should allocate and map this array the same way as you did the pages array.

这样的类似的不周，我们之前已经做过了。先使用 boot_alloc 进行物理空间的分配，然后使用 boot_map_segment 对物理地址和线性地址进行一个映射，主要就是先看线性地址有没有页表，如果有而且可访问，就直接将线性地址索引到物理地址，再讲该物理地址对应的虚拟地址返回，相当于原本的物理地址的内容不要了，重新映射到新的物理地址上去。否则就用 page_alloc 分配一个（当然由于我们使用了 boot_allow，所以实际上这里是不可能发生的）。

Exercise 2. In the file env.c, finish coding the following functions:

```
env_init():
    initialize all of the Env structures in the envs array and add them to the
    env_free_list.
env_setup_vm():
    allocate a page directory for a new environment and initialize the kernel
    portion of the new environment's address space.
segment_alloc():
    allocates and maps physical memory for an environment
load_icode():
    you will need to parse an ELF binary image, much like the boot loader
    already does, and load its contents into the user address space of a new
    environment.
env_create():
    allocate an environment with env_alloc and call load_icode load an ELF binary
    into it.
env_run():
    start a given environment running in user mode.
```

As you write these functions, you might find the new cprintf verb %e useful -- it prints a description corresponding to an error code. For example,

```
r = -E_NO_MEM;
panic("env_alloc: %e", r);
```

will panic with the message "env_alloc: out of memory".

env_init 主要用来初始化所有 NENV 个 Env 个体，将它们加入到空闲队列当中，作为可用进程控制块，可以以逆序方式插入。

env_setup_vm 主要用来为当前用户进程环境 e 分配控制结构的页目录。通过 page_alloc 分配一页，然后将当前进程的 env_cr3 字段指向该页的物理地址，将当前进程的 env_pgdir 字段指向该页的逻辑地址。这里需要将逻辑地址指向的内存清 0，因为这块内存是控制该进程所有内存页式分配的页目录。

segment_alloc 该函数主要用于给进程分配内存空间，用于存放进程运行所需资源。调用实验 2 中的 page_insert 函数就可以实现。此处分配的空间可以是根据 env_pgdir[] 页目录来控制分配，可以由虚拟地址找到分配的页面，而不是 boot_pgdir 在 kernel 的地址空间中分配。

load_icode 的功能是将给定的 2 进制文件形式存储的 elf 文件重新映射到进程空间中已经分配好的那一部分内存空间里。

主要方法：从 uint8_t * 结构的 binary 指针加上 env_elf 给定的偏移量中可以计算出 binary 中的所有段都被加载到内存中的其起始虚拟地址 va: ph->p_va，该地址是起始地址，所以在之后首先对于当前进程在该虚拟地址处分配空闲的物理页面，然后将制定大小的数据复制到从这一地址开始的内存中，然后同时必须改变当前进程的 eip 指针，将其指向这一片读入的代码的入口地址: env_elf->e_entry。从而可以根据设置好的 CS 和新的偏移量 eip 找到用户进程需要执行的代码。这里需要用到结构 Elf 和 Proghdr，顺带提一下 Secthdr 的结构。

```

struct Elf {
    uint32_t e_magic; // 标识文件是否是 ELF 文件
    uint8_t e_elf[12]; // 魔数和相关信息
    uint16_t e_type; // 文件类型
    uint16_t e_machine; // 针对体系结构
    uint32_t e_version; // 版本信息
    uint32_t e_entry; // Entry point 程序入口点
    uint32_t e_phoff; // 程序头表偏移量
    uint32_t e_shoff; // 节头表偏移量
    uint32_t e_flags; // 处理器特定标志
    uint16_t e_ehsize; // 文件头长度
    uint16_t e_phentsize; // 程序头部长度
    uint16_t e_phnum; // 程序头部个数
    uint16_t e_shentsize; // 节头部长度
    uint16_t e_shnum; // 节头部个数
    uint16_t e_shstrndx; // 节头部字符串索引
};

```

重要成员为 `e_entry`、`e_phoff`、`e_phnum`、`e_shoff`、`e_shnum`。其中 `e_elf` 为预留空间，可以看作是基本不起作用的。`e_entry` 为可执行程序的入口地址，即从内存的这个地址开始执行，其为虚拟地址，也是链接地址。`e_phoff`、`e_phnum` 可以用来找到所有程序头表项，`e_phoff` 是程序头表的第一项相对于 ELF 文件开始位置的偏移，`e_phnum` 为表项的个数，同理 `e_shoff`、`e_shnum` 可以用来找到所有的节头表项。

```

struct Proghdr {
    uint32_t p_type; // 段类型
    uint32_t p_offset; // 段位置相对于文件开始处的偏移量
    uint32_t p_va; // 段在内存中地址(虚拟地址)
    uint32_t p_pa; // 段的物理地址
    uint32_t p_filesz; // 段在文件中的长度
    uint32_t p_memsz; // 段在内存中的长度
    uint32_t p_flags; // 段标志
    uint32_t p_align; // 段在内存中的对齐标志
};

```

这个结构比较重要的成员为 `p_offset`、`p_va`、`p_filesz`、`p_memsz`。其中通过 `p_offset` 可以找到该段在磁盘中的文职，通过 `p_va` 可以知道应该把这个段放到内存的位置，而 `p_filesz` 和 `p_memsz` 是因为.bss 这种节在硬盘里没有存储空间，这些必须在内存中分配空间。

```

struct Secthdr {
    uint32_t sh_name;      // 节名称
    uint32_t sh_type;      // 节类型
    uint32_t sh_flags;     // 节标志
    uint32_t sh_addr;      // 节在内存中的虚拟地址
    uint32_t sh_offset;    // 相对于文件首部的偏移
    uint32_t sh_size;      // 节大小(字节数)
    uint32_t sh_link;      // 与其它节的关系
    uint32_t sh_info;      // 其它信息
    uint32_t sh_addralign; // 字节对齐标志
    uint32_t sh_entsize;   // 表项大小
};


```

注意几个问题：

1、对于用户程序 ELF 文件的每个程序头 ph, ph->p_memsz 和 ph->p_filesz 是两个概念，前者是该程序头应在内存中占用的空间大小，而后者是实际该程序头占用的文件大小。他们俩的区别就是 ELF 文件中 BSS 节中那些没有被初始化的静态变量，这些变量不会被分配文件储存空间，但是在实际载入后，需要在内存中给与相应的空间，并且全部初始化为 0。所以具体来讲，就是每个程序段 ph, 总共占用 p->memsz 的内存，前面 p->filesz 的空间从 binary 的对应内存复制过来，后面剩下的空间全部清 0。

2、ph->p_va 是该程序段应该被放入的虚拟空间地址，但是注意，在这个时候，虚拟地址空间是用户环境 Env 的虚拟地址空间。可是，在进入 load_icode 时，是内核态进入的，所以虚拟地址空间还是内核的空间。所以我们需要使用 lrc3 修改状态，将页表切换到用户虚拟地址空间。这样我们就可以方便的在后面使用 memset 和 memmove 等函数对一个虚拟地址进行相应的操作了。其中 e->env_cr3 的值是在前面的 env_setup_vm 设置好的。但是对于 ELF 载入完毕以后，我们就不再需要对用户空间进行操作了，需要重新载入 boot_cr3。

env_create 的任务很明确，调用 env_alloc 创建好进程的地址空间页目录，然后调用 load_icode 装入进程的地址空间即可。

env_alloc 的用法注释很清楚，这里再说明一下，组要从空闲链表当中取出一个 PCB(env)，然后初始化，接下来生成 env_id，其低十位更具 e-envs 来决定的，NENV 为 1024，也是 10 位(这里是保证低 10 位全部为 0)，如此确保每一个 id 都不会重复，接下来的几句设置好了段寄存器，特权级设置为 3，至于 gdt 表是在 i386_vm_init 当中设置完的，表项在 Segdesc gdt[] 当中。在这里，我们在 i386_init 当中看到了 user_hello，这个文件的信息在 kernel.sym 当中，之后我们再去查看。

env_run 该函数的主要功能是运行初始化完毕的进程，所以在加载新的进程 cr3 寄存器之前(这里会重定位 CS 和 eip)，必须将原先设置好的 es、ds、esp 入栈，防止之后被破坏，这里需要调用 env_pop_tf 完成，加载完毕之后重新设置这些寄存器，然后用户进程就在新的代码段中开始执行用户的程序了。在 env_pop_tf 当中，参数说明保存的是 Trapframe 的变量。在 mips 指令集当中，也有类似的思想，将该变量作为一个大栈，然后利用 pop 命令一个一个取出来，放到指定寄存器当中。

Algorithm 3: POPA - Pop All General Purpose Registers

```

begin
    EDI ← Pop();
    ESI ← Pop();
    EBP ← Pop();
    ESP ← ESP + 4 ;(* Skip next 4 bytes of stack *)
    EBX ← Pop();
    EDX ← Pop();
    ECX ← Pop();
    EAX ← Pop();
end

```

通过比对之前的 Trapframe 和 PushRegs，我们很容易知道，Trapframe 前 8 个为一个 struct PushRegs，定义的顺序和 popal 设置的完全相反，这种设计比较巧妙。

用户进程运行前的调用图：

- start (kern/entry.S)
- i386_init
 - cons_init
 - i386_detect_memory
 - i386_vm_init
 - page_init
 - env_init
 - idt_init (still incomplete at this point)
 - env_create
 - env_run
 - env_pop_tf

Handling Interrupts and Exceptions

Exercise 3. Read [Chapter 9, Exceptions and Interrupts](#) in the [80386 Programmer's Manual](#) (or Chapter 5 of the [IA-32 Developer's Manual](#)), if you haven't already.

本实验中，我们将根据 intel 对中断、异常的定义进行这次实验。异常和中断都是一种保护转移机制，能够将处理器从用户态变为核态，使得用户程序有机会试用内核函数或者其他属性来进行运作。在 Intel 的说明当中，中断总是异步发生的，比如 I/O 操作。异常总是同步发生的，比如除零错或者访问到违例地址。

为了保证保护转换机制是有效的，中断和异常的发生进入到内核的途径是被确定下来的，一个进程的执行不能对内核和其他进程产生干扰，当用户进程需要执行特权指令或者内核功能时，需要一个系统调用，使处理器从用户态切换到内核态，处理完毕返回用户态。中断和异常可以完成这些功能，他们是保护控制转移由两个特殊机制实现：IDT(中断描述符)和 TSS(任务状态段)

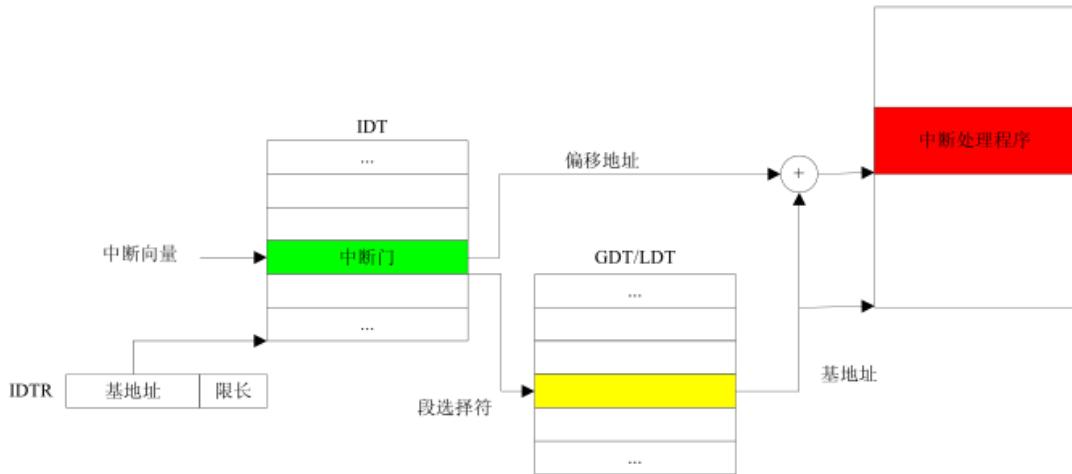
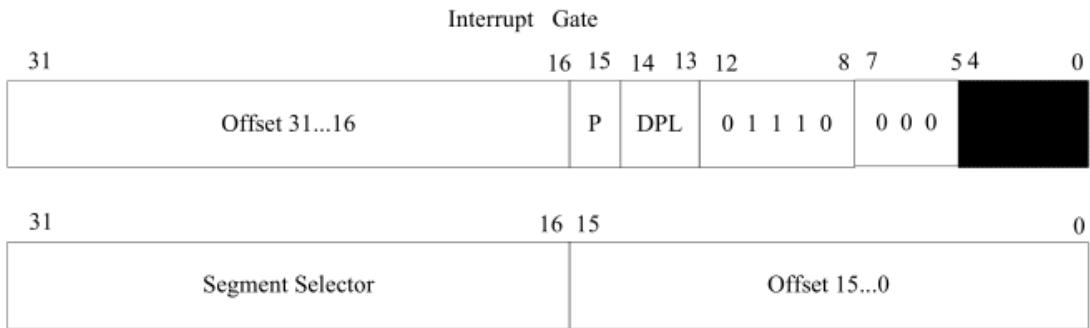


图 5-5. 中断处理过程

x86 允许 256 个不同的中断或者异常入口，每一个中断和异常都有一个唯一的整数值表示中断向量。中断向量被 CPU 用来作为 IDT 的索引访问对应门描述符，而中断门用来指向目标代码，即 Segment Selector+Offset，其中中断门为：



该结构 struct Gatedesc 在 inc/mmu.h 中定义。由结构可知，中断或异常也有特权级别，由中断门描述符中的 DPL 约束，门描述符中的段选择子中断 CPL 说明中断处理程序运行的特权级别。

TSS 在现代的操作系统里面是一个特殊的数据结构，一个任务的所有状态信息存储在其中。处理器处理进程切换时需要 TSS 来保存旧的处理器状态，一边在中断返回时恢复以前的过程。当 x86 处理器中断或者异常时，切换到内核区域的一个堆栈也是由 TSS 指出来的。和 Gatedesc 在同一个文件里面，名称为 Taskstate。

I/O Map Base Address		T
	LDT Segment Selector	
	GS	
	FS	
	DS	
	SS	
	CS	
	ES	
	EDI	
	ESI	
	EBP	
	ESP	
	EBX	
	EDX	
	ECX	
	EAX	
	EFLAGS	
	EIP	
	CR3	
	SS2	
	ESP2	
	SS1	
	ESP1	
	SS0	
	ESP0	
	Previous Task Link	
Reserved		

中断映射布局，在 x86 体系有中断向量 0-31 为内部处理器异常，31 以上的中断仅被用

于软件中断或者异步硬件中断。在 trap.h 当中，我们可以看到他们的定义。由中断向量的类型可知，Jos 系统的系统调用 T_SYSCALL 为 int\$0x30。

Mit 举了几个例子进行说明：

1) 除零中断

堆栈切换为重点。在内核模式的异常处理当中，需要将异常参数压倒当前堆栈；而在用户模式下的异常/中断处理中，需要切换到 TSS 中的 SS0, ESP0 所指向的堆栈。

除零中断中，我们要切换 TSS 中 SS0(GD_KD), ESP0(KSTACKTOP) 所指向的堆栈，然后在内核堆栈中压入必要信息，结构如图：

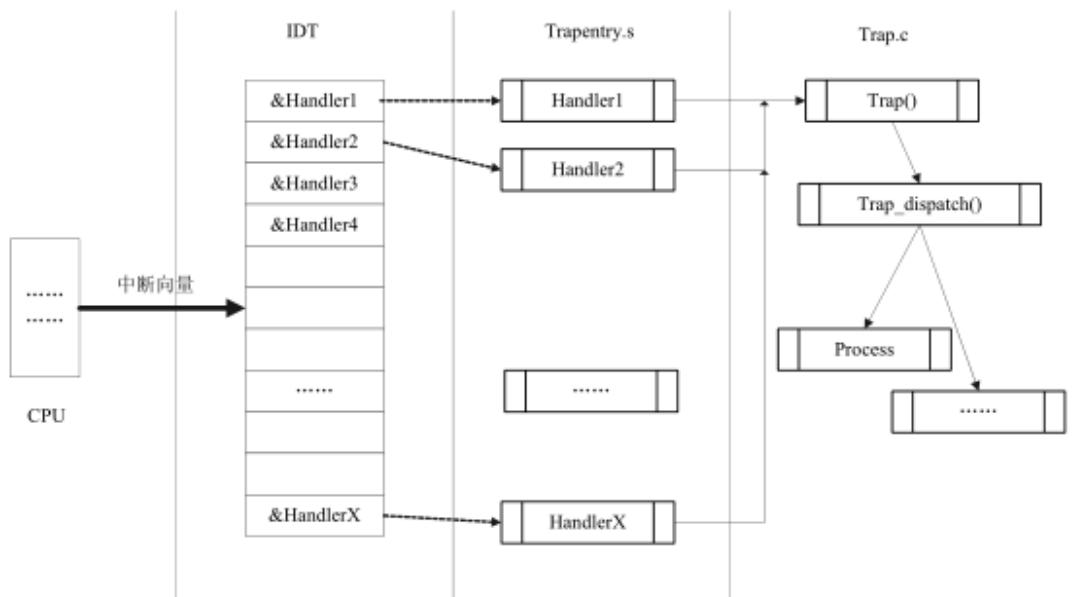
+-----+ KSTACKTOP	
0x00000 old SS " - 4	
old ESP " - 8	
old EFLAGS " - 12	
0x00000 old CS " - 16	
old EIP " - 20 <---- ESP	
+-----+	

然后设置 CS: EIP 使其指向 IDT 中 0 号中断对应中断门中保存的中断处理函数地址，其过程如 5.4.1.1 中的图 5 所示，在除零中断处理函数处理完该中断后返回用户进程。

对于一些 x86 异常，如缺页异常(page fault)，处理上述 5 个字段外，有时会加上一个 error code，此时就会在内核堆栈多压入一个字 error code：

+-----+ KSTACKTOP	
0x00000 old SS " - 4	
old ESP " - 8	
old EFLAGS " - 12	
0x00000 old CS " - 16	
old EIP " - 20	
error code " - 24 <---- ESP	
+-----+	

Jos 系统中断过程的控制流



中断的初始化：在 trapentry.S 当中，我们需要判断出每一种异常或者中断是否需要压入 error code，然后完成标号_alltraps 的部分。

在 IDT Descriptors 中的 IDT 有三种类型，Task gates、Interrupt gates、Trap gates，在 x86 当中还有一个 Call gates。通用寄存器 EFLAGS 保存的是 CPU 的执行状态和控制信息，如下图，我们需要关注其中的两个寄存器 IF 和 TF 即可：

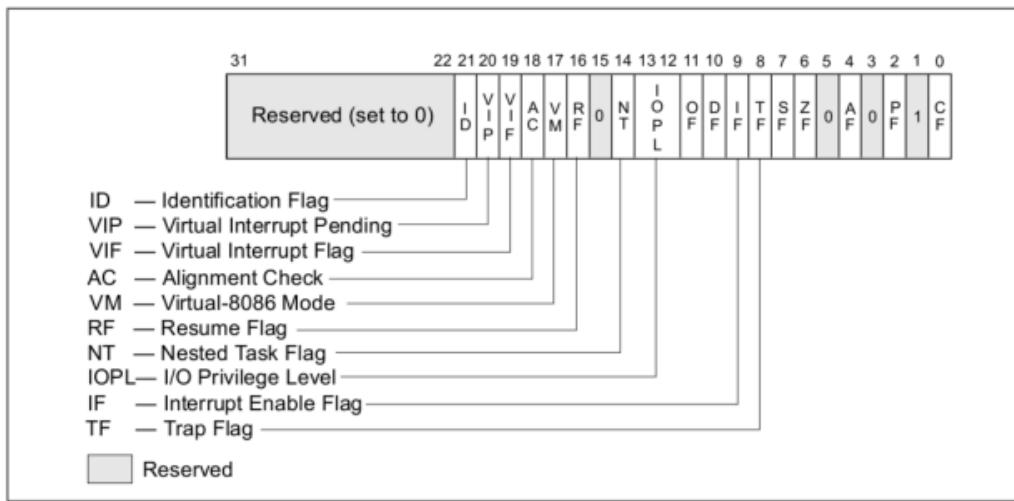


Figure 2-4. System Flags in the EFLAGS Register

TF(Trap Flag)

跟踪标志，置 1 开启单步执行调试模式，置 0 则关闭。在单步执行模式下，处理器在每条指令后产生一个调试异常，这样在每条执行后都可以查看执行程序的状态。

IF(Interrupt enable)

中断许可标志，控制处理器对可屏蔽硬件中断请求的响应。置 1 开启可屏蔽硬件中断响应，置 0 则关闭可屏蔽硬件中断响应。IF 标志不影响异常和不可屏蔽中断 NMI 的产生。

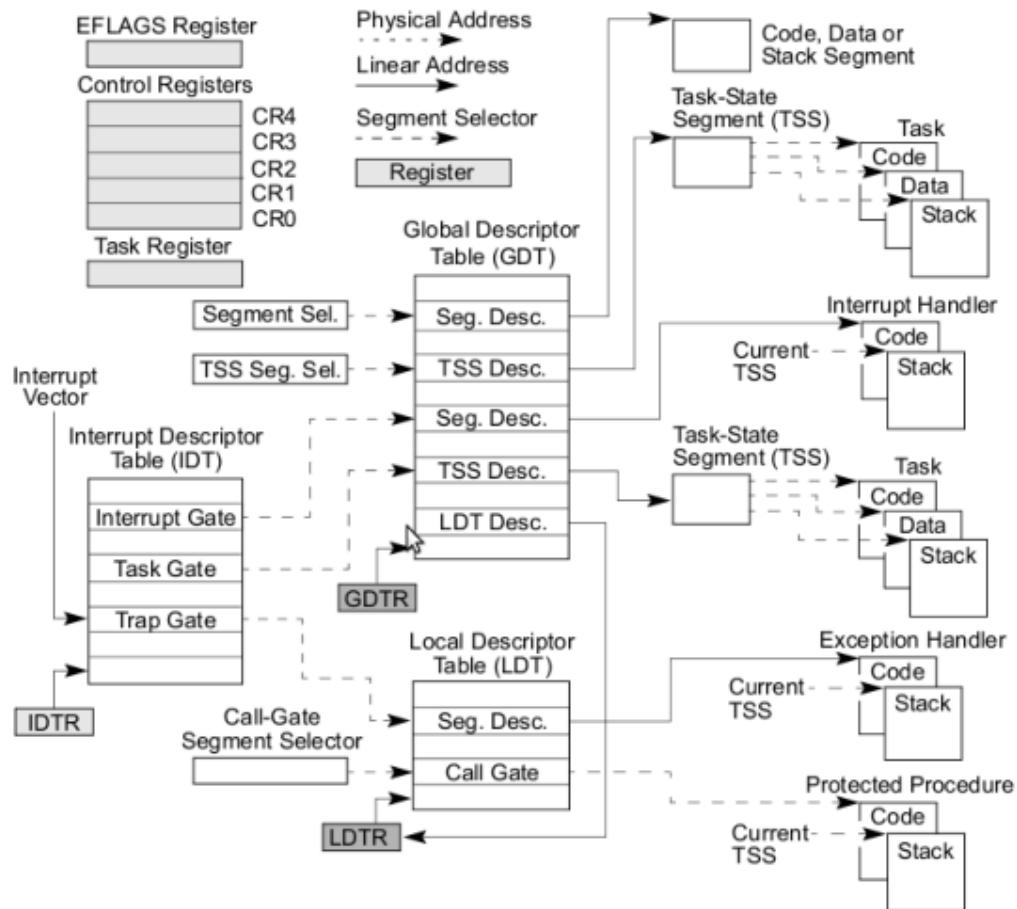
门是用来实现性一段代码跳转到另一条代码(不同的代码段，不同的特权级)时的保护机制问题。中断们和异常门和其他两个门的主要区别就是，前两者是专门处理处理器中断或异常，后面两种一般处理用户软件进程间切换。

中断门和异常门的描述符基本上没有区别，而实际上他们的执行工作也很类似，但是中断门会修改 IF，会对中断响应进行屏蔽，即不再响应接下来的中断。如果在中断当中使用异常门，即不会对下一个中断昌盛屏蔽，则如果中断到来过于频繁，导致前一个中断未处理完就被打断，会引起数据崩溃。所以必须进行屏蔽。还有一个是断点异常的处理，这个中断必须暴露给用户程序进行调用，但是又不同于中断，所以使用 trap gate 来处理。

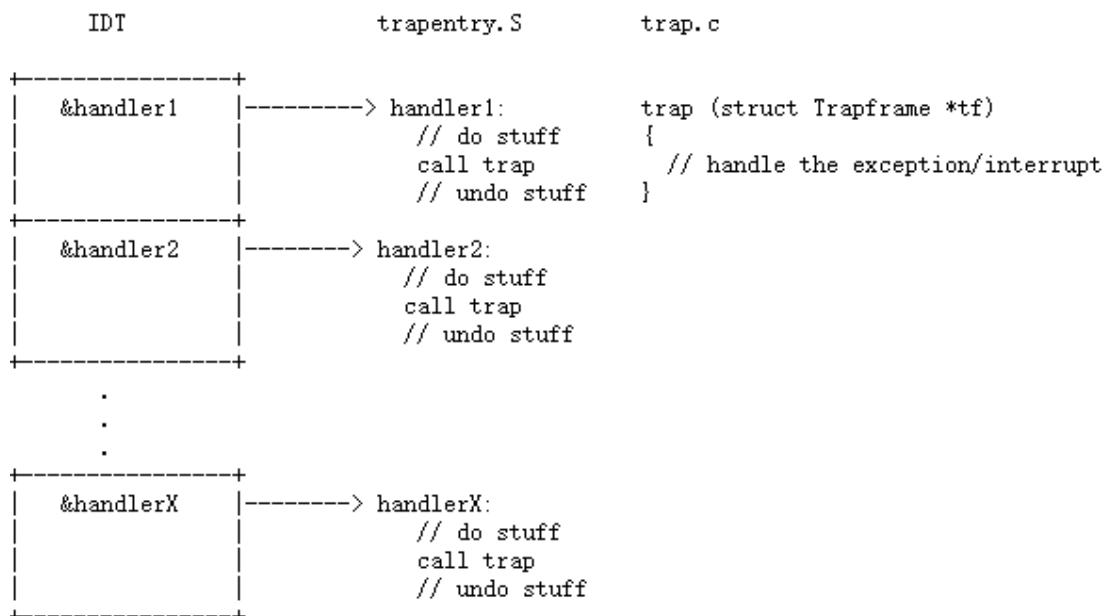
Call gate 和 Task gate 的区别，task 是一个具体的可执行单位，可以运行、挂起、重启等，在这个单位上，可以将其保存状态到 TSS 当中，我们可以通过一个 CALL 或者 JMP 指令来调用一个 task 程序。首先两者都可以用来切换一个 task，但是 task gate 的寻址需要经过一个 TSS 找到 code selector，虽然比 call gate 麻烦，但是可以：

- 1、切换时原来 task 的上下文环境被自动保存 TSS
- 2、如果使用 task gate 来处理中断例程，可以使程序和其他例程分开，使其

有独立空间。



Setting Up the IDT



Exercise 4. Edit trapentry.S and trap.c and implement the features described above. The macros TRAPHANDLER and TRAPHANDLER_NOEC in trapentry.S should help you, as well as the T_* defines in inc/trap.h. You will need to add an entry point in trapentry.S (using those macros) for each trap defined in inc/trap.h. You will also need to modify idt_init() to initialize the idt to point to each of these entry points defined in trapentry.S; the SETGATE macro will be helpful here.

Hint: your code should perform the following steps:

1. push values to make the stack look like a struct Trapframe
2. load GD_KD into %ds and %es
3. pushl %esp to pass a pointer to the Trapframe as an argument to trap()
4. call trap
5. pop the values pushed in steps 1-3
6. iret

Consider using the pushal and popal instructions; they fit nicely with the layout of the struct Trapframe.

Test your trap handling code using some of the test programs in the user directory that cause exceptions before making any system calls, such as user/divzero. You should be able to get make grade to succeed on the divzero, softint, and badsegment tests at this point.

IDT 的数据结构如下

```
/* Interrupt descriptor table. (Must be built at run time because
 * shifted function addresses can't be represented in relocation records.)
 */
struct Gatedesc idt[256] = { { 0 } };
struct Pseudodesc idt_pd = {
    sizeof(idt) - 1, (uint32_t) idt
};
```

其中 idt_pd 是系统寄存器 IDTR 的对应结构，门描述符数据结构 struct Gatedesc
Exercise 分为两步

1、在 trapentry.S 中定义好每个中断对应的中断处理程序

2、在 trap.c 的 idt_init 中将那些第一步定义好的中断处理程序安装进 IDT

每个 interrupt handler 都必须要做的事情在内核栈中设置好一个 Trapframe 布局，然后将这个结构传给 trap 进行一步不离，最后在 trap_dispatch 中进行具体中断处理程序的分发。

trapentry.S 中有两个宏定义，一个是 TRAPHANDLER，另一个是 TRAPHANDLER_NOEC，前者的功能主要是接受一个函数名和对应处理的中断向量编号，然后定义出一个相应的以该函数命名的中断处理程序，这样的中断向量程序的执行流程就是向栈里压入相关错误代码和中断号，然后跳转到_alltraps 来执行共有部分。对于错误代码，如果是系统运行产生的中断，根据不同的类型，在切换后，处理器会向栈中放入一个错误代码。但是 Divide Zero 等一些例外不会放入，当用户使用 int 手动条用中断时，处理器是不会放入错误代码的。此时，我们的中断程序就要自己补齐这个空间。TRAPHANDLER_NOEC 宏就是帮我们完成这个事情。

在 trapentry.S 当中完成这些事项，格式如下：

程序补充错误代码：

TRAPHANDLER_NOEC(异常名字<这个随便，不过在 trap 中必须对应>, T_ 对应异常号)

包括 divide_zero、debug、nmi、breakpoint、overflow、bounds、invalid_op、device_not_available、double_fault、float_point_error、syscall_call。

自动填入错误代码：

TRAPHANDLER(异常名字<同上>, T_ 对应异常号)

包括 invalid_TSS、segment_not_present、stack_segment、general_protection、page_fault、alignment_check、machine_check、SIMD_float_point_error。

<是否需要错误代码在 Intel 手册 0-31 号系统预留中断类型可以看到>

然后在_alltraps 中设置好剩下所有的结构:

```
_alltraps:
    pushw $0x0
    pushw %ds
    pushw $0x0
    pushw %es
    pushal
    movl $GD_KD, %eax
    movw %ax, %ds
    movw %ax, %es
    pushl %esp
    call trap
```

在 idt_init 当中先 extern 所有的函数, 这里的函数名和刚才你所定义的名字是一一对应的, 然后在后面紧接着使用 SETGATE 将 idt[T_中断向量号<之前说是异常号不准确>], 和你的函数建立对应关系即可。

格式: SETGATE(idt[T_编号], 0, GD_KT, 函数名, 0);

在这里我们先回顾一下 SETGATE 是如何设置一个门的

```
// Set up a normal interrupt/trap gate descriptor.
// - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
// - sel: Code segment selector for interrupt/trap handler
// - off: Offset in code segment for interrupt/trap handler
// - dpl: Descriptor Privilege Level -
//       the privilege level required for software to invoke
//       this interrupt/trap gate explicitly using an int instruction.
#define SETGATE(gate, istrap, sel, off, dpl) \
{ \
    (gate).gd_off_15_0 = (uint32_t) (off) & 0xffff; \
    (gate).gd_ss = (sel); \
    (gate).gd_args = 0; \
    (gate).gd_rsv1 = 0; \
    (gate).gd_type = (istrap) ? STS_TG32 : STS_IG32; \
    (gate).gd_s = 0; \
    (gate).gd_dpl = (dpl); \
    (gate).gd_p = 1; \
    (gate).gd_off_31_16 = (uint32_t) (off) >> 16; \
}
```

英文注释很清楚, idt 是一个 Gatedesc 的数组, 对比之前该类型的解释即可。

第三个参数为内核代码段 GD_KT, 最后一个参数为用户特权级的设置, (为什么设置?)。

下面我们来整理一下全部思路: 根据代码, 如果一个除零中断被捕获, 就会在 trapentry.S 中使用 TRAPHANDLER_NOEC、或者 TRAPHANDLER, 然后在_alltraps 中进行 Trapframe 的设置。设置完后, 就跳转到 trap.c 的 trap 中, 最终在 trap_dispatch 中打印存储器信息。

现在 JOS 的中断处理程序在真正的处理之前要将中断号放入内核栈, 以组织成 Trapframe 的结构, 但是如果所有中断都跳到同一个处理程序, 那么就无法区分是哪个中断调用进来的, 也就无法正确设置它们的中断号了。

在中断向量里设置的 14 号 Page fault 的调用权限是 0, 即只能内核抛出, 所以直接在 softint 中用 int 指令调用肯定产生的 General Protection Fault 权限错误, 类似于 user_hello。如果在我们 SETGATE 的时候讲最后一个参数改为 3, 是可以显示 Page Fault 的。但是 Page fault 中断是需要压入错误代码, 而用户用 int 指令调用中断是不会压入错误代码的。可是在 kern/trapentry.S 中为 Page fault 指定的中断处理程序默认认为系统为我们放入了错误码, 所以不会补齐。故在使用 trap_dispatch 打印寄存器的信息时会发生错位。此时在访问 ss 寄存器的时候会访问到 KSTACKTOP 之上的空间。

从 memlayout.h 当中, 我们可以知道之上的空间为 VPT, 这是系统页目录。故以 VPT 的

虚拟地址进行方寸，VPT 的 PDX 会找到系统目录，PTX 为 0，就是页目录的第 0 个页表的物理地址，又因为 offset 也是 0，故最终访问的是第 0 个页表的第 0 个页表项。Jos 在载入 softint 的时候会分配物理页到 elf 文件中，在 user/user.ld 中关于文件中 stab 节的链接地址为 0x2000000，这就是我们 ss 寄存器访问到的地址。我们在 user.ld 上删去这一段，重新编译运行 Jos，就可以看到，在第一个 Page fault 吃力程序中，在打印 ss 的时候因为没有找到这一段，于是又会发生第二个 Page fault。

Part B: Page Faults, Breakpoints Exceptions, and System Calls

Exercise 5. Modify trap_dispatch() to dispatch page fault exceptions to page_fault_handler(). You should now be able to get make grade to succeed on the faultread, faultreadkernel, faultwrite, and faultwritekernel tests. If any of them don't work, figure out why and fix them.

Exercise 6. Modify trap_dispatch() to make breakpoint exceptions invoke the kernel monitor. You should now be able to get make grade to succeed on the breakpoint test.

Exercise 7. Add a handler in the kernel for interrupt vector T_SYSCALL. You will have to edit kern/trapentry.S and kern/trap.c's idt_init(). You also need to change trap_dispatch() to handle the system call interrupt by calling syscall() (defined in kern/syscall.c) with the appropriate arguments, and then arranging for the return value to be passed back to the user process in %eax. Finally, you need to implement syscall() in kern/syscall.c. Make sure syscall() returns -E_INVAL if the system call number is invalid. You should read and understand lib/syscall.c (especially the inline assembly routine) in order to confirm your understanding of the system call interface. You may also find it helpful to read inc/syscall.h.

Run the user/hello program under your kernel. It should print "hello, world" on the console and then cause a page fault in user mode. If this does not happen, it probably means your system call handler isn't quite right.

在 trap_dispatch 当中，需要检查 Trapframe 结构中的中断向量号，并将中断分发到响应的中断处理过程。如遇到缺页中断就分发到 page_fault_hander 处理；遇到断点异常就分发到 monitor 进行处理。

在 Jos 当中，我们使用 int\$0x30 指令引起处理器中断，完成系统调用。通过系统调用可以使得用户进程可以要求内核完成一些功能，当内核执行完响应的代码返回用户进程继续执行。Jos 已将 T_SYSCALL 定义为 0x30，我们需要将其加入到中断描述符中。

我们要设置 idt[0x30] 的 idl 时必须为 3，这样才会允许系统进行调用。在 int n 的软终端调用时会检查 CPL 和 DPL 是否满足对应的级别关系，只有当前的 CPL 数值上小于等于要调用的 DPL 时才可以调用，否则会产生 General Protection 中断。

kern/syscall.c 中的 syscall 是 Jos 系统调用实现的函数，参数一共有 6 个，第一个是调用号，后面 5 个是参数。因此，当我们在 trap_dispatch 函数中发现要系统调用时，需要将当前进程控制块记录的 eax、edx、ecx、ebx、esi、edi 寄存器的值压入函数堆栈，然后将返回值刷新当前进程控制块中的记录 eax 即可。

User-mode startup

Exercise 8. Add the required code to the user library, then boot your kernel. You should see user/hello print "hello, world" and then print "i am environment 000000800". user/hello then attempts to "exit" by calling sys_env_destroy() (see lib/libmain.c and lib/exit.c). Since the kernel currently only supports one user environment, it should report that it has destroyed the only environment and then drop into the kernel monitor.

操作系统最终要运行用户进程，在 Jos 中，用户进程是 lib/entry.S 开始运行的，然后转入到函数 libmain，libmain 中对于 env 的设置之前，entry.S 将 env 指向了 UENV，现在需要更改为指向当前进程控制块的地址，对于全局数组 envs[]，我们只需要根据当前进程编号在 envs 里面进行搜索就可以找到当前的进程控制块，ENVX 的作用是保留进程号的低 10 位，

之前我们介绍过进程号产生的方法，低 10 位应该正好是当前进程与 envs[0]的距离。

我在这里遇到一个问题，比较坑，那就是没有“i am en..”这句话，出现问题所在是写代码的一时疏忽，以前都会好好加上的地方，这里没有加上，在我们做根据中断向量号分类的时候，我们会在 trap_dispatch 的每一种情况处理完后就 return 的。

Page faults and memory protection

这里会引入内存保护，可以确保用户进程中断 bugs 不会破坏其他进程或者内核。当用户进程试图访问一个无效的或者没有权限的地址时，处理器就会中断进程陷入到内核，如果错误可以修复，内核就可以修复并让进程继续执行，否则用户进程就不能继续执行。而在系统调用当中导致了内存保护。许多系统调用接口运行把指针传给 kernel，而这些指针指向用户 buffer，为防止一些恶意程序破坏内核，需要内核对用户传递进来的指针进行权限检查。主要由 user_mem_check 和 user_mem_assert 实现，主要检查访问权限和是否越界。

Exercise 9. Change kern/trap.c to panic if a page fault happens in kernel mode.

Hint: to determine whether a fault happened in user mode or in kernel mode, check the low bits of the tf_cs.

Read user_mem_assert in kern/pmap.c and implement user_mem_check in that same file.

Change kern/syscall.c to sanity check arguments to system calls.

Change kern/init.c to run user/buggyhello instead of user/hello. Compile your kernel and boot it. The environment should be destroyed, and the kernel should *not* panic. You should see:

```
[00001000] user_mem_check assertion failure for va 00000001
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
```

Exercise 10.

Change kern/init.c to run user/evilhello. Compile your kernel and boot it. The environment should be destroyed, and the kernel should not panic. You should see:

```
[00000000] new env 00001000
[00001000] user_mem_check assertion failure for va f0100020
[00001000] free env 00001000
```

Exercise 9:

首先在 page_fault_hander 当中对 tf->tf_cs&3 进行判断当前状态是否为内核态，因为之前我们已经处理过内核态([找找在哪里？](#))，所以只需处理用户态的缺页即可，根据注释我们知道那是实验四的内容，现在暂时不用管。

user_mem_assert 之前提到过是用来检查访问权限和是否越界。但是我们定位到函数内部，可以发现其主要还是通过对 user_mem_check 进行的调用实现效果。检查方法主要是从低地址到高地址，如果越界，或者用 pgdir_walk 取出来的页表项权限不够或者为空时返回缺页信息即可。

之后需要在 sys_cputs、kdebug 当中完成 user_mem_check 的引用即可（根据注释来）。

Exercise 10:

修改一下吧，直接在 init.c 当中修改 hello 为 evilhello，然后你重新编译后就可以找到错误信息了。

重要附录参考代码

```
env_init(void)
{
    // LAB 3: Your code here.
    int i;
    LIST_INIT(&env_free_list);
    for(i = NENV-1;i>=0;i--){
        envs[i].env_id = 0; // set id = 0
        envs[i].env_status = ENV_FREE; // envs as free
        LIST_INSERT_HEAD(&env_free_list, &envs[i], env_link); //set into the free_list
    }
}
```

load_icode 关键代码(注意现场切换)

```

struct Elf *env_elf = (struct Elf*)binary;//turn the binary to Elf
struct Proghdr *ph, *eph;
if(env_elf->e_magic != ELF_MAGIC){
    panic("load_icode: this is not a elf file\n");
}
ph = (struct Proghdr*)((uint8_t *)env_elf + env_elf->e_phoff);// 
eph = ph + env_elf->e_phnum;

lcr3(e->env_cr3);
for(;ph<eph;ph++){
    if(ph->p_type == ELF_PROG_LOAD){// must elf_prog_load
        segment_alloc(e, (void *)ph->p_va, ph->p_memsz); //get the vm for e
        memset((void *)ph->p_va, 0, ph->p_memsz); //reset it clearly
        memmove((void *)ph->p_va, binary+ph->p_offset, ph->p_filesz); //copy to it
    }
}
lcr3(boot_cr3);

e->env_tf.tf_eip = env_elf->e_entry;

```

trap_dispatch 部分代码(后期不需要 panic 代码断掉 syscall)

```

trap_dispatch(struct Trapframe *tf)
{
    // Handle processor exceptions.
    // LAB 3: Your code here.
    //cprintf("the trapno is 0x%x\n", tf->tf_trapno);
    if(tf->tf_trapno == T_PGFLT){ //page fault
        //cprintf("%x \n", tf->tf_err);
        page_fault_handler(tf);
        return ;
    }

    if(tf->tf_trapno == T_BRKPT){ //break point fault
        monitor(tf);
        return ;
    }

    if(tf->tf_trapno == IRQ_OFFSET + IRQ_TIMER){
        sched_yield();
        return ;
    }

    if(tf->tf_trapno == T_SYSCALL){ //syscall
        int r = syscall(
            tf->tf_r0, tf->tf_r1, tf->tf_r2, tf->tf_r3,
            tf->tf_r4, tf->tf_r5, tf->tf_r6, tf->tf_r7,
            tf->tf_r8, tf->tf_r9, tf->tf_r10, tf->tf_r11,
            tf->tf_r12, tf->tf_r13, tf->tf_r14, tf->tf_r15,
            tf->tf_r16, tf->tf_r17, tf->tf_r18, tf->tf_r19,
            tf->tf_r20, tf->tf_r21, tf->tf_r22, tf->tf_r23,
            tf->tf_r24, tf->tf_r25, tf->tf_r26, tf->tf_r27,
            tf->tf_r28, tf->tf_r29, tf->tf_r30, tf->tf_r31
        );
        if(r < 0)
            panic("trap_dispatch: error in syscall\n");
        tf->tf_r0 = r;
    }
}

```

Preemptive Multitasking

Author: Xin Yao

本次实验基于实验 3，在实验 3 中，我们通过管理 PCB 空闲链表，并加载 elf 文件。在中断异常处理当中，使用了 tarp 当中的一些函数，每一个中断处理之前都会有各自的 trapframe 结构，其中包含了中断处理的信息，我们通过解析这些函数，并通过 SETGATE 设置了一些中断处理的权限，从而对系统进行了保护，最后我们检查了内存是否溢出、越界。

本实验在 Jos 当中实现多进程管理和进程间消息通信的功能。利用分时技术，操作系统上同时可以运行多个程序。分时技术将 CPU 的运行时间划分为一个个规定长度的时间片，让每一个进程在一个时间片上运行。进程的切换通过跳读程序完成的。当一个进程执行时，CPU 所有的寄存器中的值、进程的状态以及堆栈中的内容被称为该进程的上下文。当内核需要切换到另一个进程时，它需要保存当前进程的所有状态，即保存当前进程的上下文，以便在再次执行该进程时，能够恢复到切换时的状态执行下去。当内核切换进程时，它需要保存当前进程的所有状态，即保存当前进程的上下文，以便于再次执行进程时，能够恢复到切换时的状态继续执行。

Os 中有抢占式和非抢占式两种调度方式，本实验采用抢占式，即如果有重要或紧迫的进程到达，则先运行进程将被迫放弃处理器，处理器将立刻被分配给新到达的进程，其需要时钟中断处理程序实现。

代码主要集中在 kern 和 lib 当中，第一部分通过循环轮转算法实现多用户进程，第二部分通过类似于 Unix 进程创建的 fork 函数来创建新的进程以及实现用户态下的缺页错误处理，第三部分通过时钟实现用户进程间的消息通信。

在 i386_init 当中，我们可以看到又添加了两个初始化的函数，通过函数的注释内容，我们知道 pic_init 是初始化 8259A 中断控制器，为后面的时钟做准备，kclock_init 初始化 8253 可编程定时器，用来产生时钟中断(系统定义每秒 100 次，即 8253 被设置为每隔 10ms 就发出一个时钟中断信号)。

在 Jos 内核启动后，系统穿件了一个用户 idle 进程，其为一个周期性运行或总是等待某个时间的后台进程，是系统中的后台服务进程，他是一个生存期较长的进程，通常独立于控制中断并且周期性的执行某种任务或者等待处理某些发生的时间，也称作守护进程。守护进程在系统引导装入时启动，在系统关闭时终止。Linux 系统有很多守护进程，大多数服务是通过守护进程实现的。在 Jos 系统中，idle 进程周期性的调用调度函数，调度执行就绪的进程。在 Jos 当中，内核函数为 i386_init，调用调度函数 sched_yield，然后开始执行第一个用户进程并周期性地执行就绪的用户进程。

Part A: User-level Environment Creation and Cooperative Multitasking

Exercise 1. Implement round-robin scheduling in sched_yield() as described above. Don't forget to modify syscall() to dispatch sys_yield().

Modify kern/init.c to create two (or more!) environments that all run the program user/yield.c. You should see the environments switch back and forth between each other five times before terminating, like this:

```
...
Hello, I am environment 00001001.
Hello, I am environment 00001002.
Back in environment 00001001, iteration 0.
Back in environment 00001002, iteration 0.
Back in environment 00001001, iteration 1.
Back in environment 00001002, iteration 1.
...

```

After the yield programs exit, the idle environment should run and invoke the JOS kernel debugger. If all this does not happen, then fix your code before proceeding.

最开始的用户进程是 envs[0]，从现在开始将是一个特殊的初始态 idle 进程，它会一直运行 user/idle.c 的程序。它总是尝试将 CPU 让给其他的进程。阅读 idle.c 的代码。

函数 sched_yield 在 sched.c 当中，它负责选择一个新的进程在 CPU 上运行。它不断地在 envs 当中进行循环搜索，从前一个这个在运行的进程开始搜索，找到第一个处于就绪态

的进程，然后使用 env_run 使 CPU 切换到那个进程。该函数永远不会自主选择第一个进程，除非没有任何一个就绪的进程。

现在我们有一个新的系统调用 sys_yield，该系统调用可以调用内核函数 sched_yield，因此就会将 CPU 让给一个不同的进程。

调度算法：Jos 的循环轮转算法将 envs 数组当作循环数组进行搜索，并跳过 idle 进程，只有在没有进程可以运行时，才运行 idle 进程。调度的实际既可以是进程自愿放弃 CPU 或某种时间阻塞时进行进程切换。后一种涉及到了抢占式调度，比如当前进程在 I/O 是并没有用到处理器，此时内核需要强制暂停当前进程，执行一个紧急的就绪进程。

完成 sched_yield 后，就在 kern/syscall.c 添加相关的分发机制，然后在 kern/init.c 中系统启动之初创建 user_idle，然后再创建 user_yield 这个用户程序的功能就是 5 次 sys_yield 的系统调用。

syscall 的修改按照下面的要求进行：

Exercise 2. Implement the system calls described above in kern/syscall.c. You will need to use various functions in kern/pmap.c and kern/env.cc, particularly envid2env(). For now, whenever you call envid2env(), pass 1 in the checkperm parameter. Be sure you check for any invalid system call arguments, returning -E_INVAL in that case. Test your JOS kernel with user/dumbfork and make sure it works before proceeding.

传统上，Unix 提供 fork 系统调用创建进程，涉及到一些特权指令，如进程的状态设置和内存分配。在这些过程中很容易产生一些恶意代码对系统进行破坏。在创建进程时，我们需要用户进程陷入内核态执行。

涉及到的函数有：

sys_exofork、sys_env_set_status、sys_page_alloc、sys_page_map、sys_page_unmap。

envid2env 将一个 envid 转换成指向 env 的一个指针，在 fork 中调用本函数一定要把 bool 设置为 1，一次来说明设置的进程要么是父进程，要么是当前进程的子进程。

sys_exofork 是 Jos 实现进程创建的一个最重要的函数，创建一个新的进程，其寄存器状态和当前父进程寄存器状态一样，并标记为 ENV_NOT_RUNNABLE，在父进程中，它返回创建子进程的 envid_t，而在子进程中返回 0。当进程调用 sys_exofork 创建子进程时，所有的寄存器的值都被复制到子进程里，包括 eip 的值，这样当子进程继续执行的时候它的内容与父进程完全相同，它也会从父进程执行系统调用的下一条指令处开始执行，而此处的汇编指令应该是将 eax 的值复制给一个变量待后续作为返回值返回，这样如果在系统调用 sys_exofork 中将新创建的子进程的 eax 寄存器置为 0，就将使得子进程继续执行时认为其系统调用的返回值为 0。

sys_env_set_status，一旦进程的地址空间和寄存器状态都初始化，用来标示一个新的执行状态信息，设置 envid_t 进程状态为 status(ENV_RUNNABLE|ENV_NOT_RUNNABLE)。

sys_page_alloc，分配一页物理内存，并将它以 perm 属性映射到 envid 进程 va 所对应的地址空间。分配时需要将页初始化为 0，以防止脏数据产生异常。系统在释放内存时不会清除内存信息，一些内存还保留有以前的信息。

sys_page_map，将 srcenvid 进程地址空间的线性地址 srcva 的页映射到 dstnvid 进程地址空间中的 dstva 地址处，并设置也属性为 perm。此处的操作并非数据拷贝而是页表操作，即两进程共享一个页的地址，这样可以防止内存空间的浪费。

sys_page_unmap，取消 envid 地址空间中线性地址 va 所对应的页映射，删除线性地址 va 所对应的物理页面。

在 syscall 当中不玩了以上的引用之后，就可以在前面加入 user_yield 进行测试，当然你可以在 env_run 里面打印所有 e 的相关信息，这样会让你明白你的做法是正确的。

顺便写一下 challenge，这里可以设置一个优先级，在同等优先级的情况下再使用循环轮转算法。

Exercise 2. Implement the system calls described above in kern/syscall.c. You will need to use various functions in kern/pmap.c and kern/env.cc, particularly envid2env(). For now, whenever you call envid2env(), pass 1 in the checkperm parameter. Be sure you check for any invalid system call arguments, returning -E_INVAL in that case. Test your JOS kernel with user/dumbfork and make sure it works before proceeding.

```
// Only flags on PTE_USER can be used in system access.
#define PTE_USER    (PTE_AVAIL | PTE_P | PTE_W | PTE_U)
```

多使用 cscope，它有力地帮你找到了 PTE 的一系列常量值，如上面的一个截图显示了用户的权限为可读写。

envid2env 函数是根据 env 的 id 来寻找 env 结构的，找不到的情况返回一个负值。

在 sys_exofork 当中，我们可以大体把中断和异常弄成了四类，fault、trap、interrupt、abort。中断和异常进行特权级的切换，而 call 则不行，call tss 切换的时候只能是同级之间或者向更低级别切换。

call 一个 function 时，先将 cs、eip 入栈，eip 是执行 call 的下一条指令，所以 function 执行完后(即 ret 之后会淡出之前所有入栈的东西)，程序在 call 的下一条指令执行。中断不仅入栈 cs、eip，它还会入栈 ss、esp、eflag、cs、eip、errno、interrupt num。这里 interrupt 和 call 情况一样，都是程序员调用。

对于 fault 跟 call 不一样的地方就是 eip 的处理，eip 是指向引发异常的地方的指令，所以在处理完异常之后会返回到这条指令，cpu 将重新执行这条指令。fault 都不是有程序员调用的，是硬件发出的。处理完后可以直接运行下一条指令，trap 执行时 eflag 的 IF 位不会置位，interrupt 是会置位的。

对于 abort，这个不允许程序或者任务继续执行，用来直接报告错误。函数返回时，其返回值总是放在 eax 当中，这就是我们函数只能返回一个值的原因。envid = sys_exofork() 实际上在汇编后会被分成多条指令，函数调用返回后，才会将 eax 的值复制到 envid 当中，所以在创建一个子进程后，应该将它的 eax 复制为 0，这样在调度之后就会在恢复现场的时候将该子线程的 eax 返回，这就是为什么 fork 一个子线程后，该子线程的返回值为 0 的原因。

关于内存的分配按照注释写起来是毫不费劲的。

该内容的测试文件为 dumbfork。

User-level page fault handling

还有在用户态 fork 时会涉及到页表出错的处理，即当创建的进程引用一个不存在页面中的内存地址时，就会触发 CPU 产生页错误异常中断，并把引起中断的线性地址放到 CR2 控制寄存器中。因此处理该中断的过程就可以知道发生页异常的确切地址，从而可以把进程要求的页面从二级存储空间加载到物理内存中。

在处理用户的页错误时，由于进程运行在用户级别，即 Ring3，而在内存操作需要在特权级别，因此，我们在处理用户级别的页错误是需要用到系统调用陷入到特权级别。由于进程添加了页错误处理，所以在 Env 中添加了页错误处理项，为 env_pgfault_upcall。该项用于记录页错误处理函数的入口。

为完成用户级页错误处理，我们需要完成以下几件事，让内核知道进程错误信息并处理完后直接返回用户态进程：

1、用户注册 user level 的 page fault 处理函数到 env 环境当中共内核调用。

2、内核处理部分为 user level 的 page fault 处理函数设置 exception stack 使得 user level 处理函数处理完后直接返回用户态出错代码继续执行，并使内核返回到 user level 的 page fault 处理函数继续执行。

3、user level 的 page fault 处理函数具体处理 page fault。

为处理用户态错误，需要在 JOS 内核注册一个页错误处理函数，用新增的数据项记录该信息，实现在 syscall 当中的 sys_env_set_pgfault_upcall 函数。

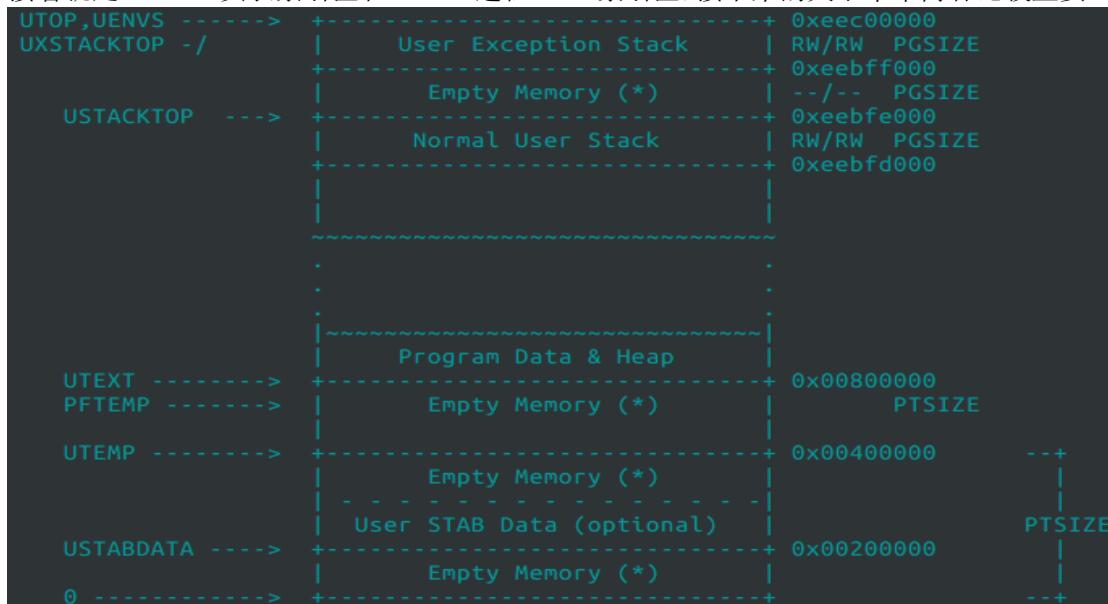
用户进程运行在用户栈，其 esp 指针指向 USTACKTOP，其堆栈数据保存在 USTACKTOP-PGSIZE 和 USTACKTOP-1 之间。当发生页错误时，内核将制定异常处理程序运行在指定的页错误处理堆栈上，即 user exception stack。其有效的地址是 UXSTACKTOP-PGSIZE 和 UXSTACKTOP-1 之间，在此堆栈行运行时，用户级页处理函数可以使用 JOS 的系统调用解决引起页错误的问题，然后直接返回到用户态出错代码继续执行。

JOS 在这个实验里增设一个与 Trapframe 对应的结构体，命名为 UTrapframe，该结构具体定义在 inc/trap.h 当中，这个结构是为了保存 Trapframe 的相关信息。这个结构保存了用户态

出错代码的寄存器信息，当页错误程序处理完后直接返回用户态出错代码继续执行。

Exercise 4. Implement the `sys_env_set_pgfault_upcall` system call. Be sure to enable permission checking when looking up the environment ID of the target environment, since this is a "dangerous" system call.

这个按照实验写具体步骤即可。实在不会的可以仿照之前写过的，将对应的所有情况写清楚，以免出错。然后，我们再来看一下 memlayout.h 的布局：KERNBASE 以上都是待映射的物理内存空间，KERNBASE 到 VPT，KSTACKTOP 为 page table，是内核可读写区，在往下 KSTKSIZE 的区间为内核栈的部分，ULIM ~ UVPT 的 PTSIZE 为用户页表可读区，紧接着就是 PAGES 页表索引区和 ENVS 进程 PCB 索引区，接下来的关于本章内容比较重要：



这里显示了用户异常栈是即可读写的，占用了一个 PGSIZE 的大小。用户栈是正常用户使用的堆栈，用户数据的堆的分配方向与栈相反。后面有几个内存空洞，其中有一份存放 STAB 数据段。

内核态系统栈运行内核相关程序，这里我们关注中断触发后，CPU 将会自动将栈切换到内核栈上来。这个是在 trap.c 中的 idt_init 当中实现的。如此，在 trapentry.S 当中进入_alltraps 时，所处的栈就已经是内核栈。_alltraps 是不断的用 push 指令进行压栈，为了形成一个 Trapframe 的内存结构，好作为参数并调用 trap 函数。

用户运行栈是在用户进程初始时设置的，在 env.c 的 env_alloc 中可以看到其设置的过程。这个时候起虚拟地址对应的地方只有一页物理地址，在载入 ELF 文件后时分配了 USTACKTOP 下一页到 USTACKTOP 之间的空间给该进程，即上图的正常用户堆栈区部分。如果使用了更多的空间，就会引发缺页中断，从而转到内核栈上去运行中断处理程序。用户运行栈是一边使用一边分配大小，系统内核栈和用户态错误栈都是固定大小的。

用户态错误栈是用户自己定义响应的中断处理程序后，响应处理程序运行时的栈。当用户进程调用前面的 `sys_env_set_pgfault_upcall` 向系统注册缺页中断处理程序后，当用户程序引发缺页中断：

- 首先系统进入内核态，栈从用户运行栈切换到内核栈，进入 trap 处理中断分发，然后进入 page_fault_handler。
 - 当确认用户程序不是内核引发缺页中断后，内核缺页直接 panic，为其在用户线程栈里分配了一个 UTrapframe 的大小。
 - 把栈切换到用户错误栈，运行对应的用户中断处理程序。
 - 最后返回用户程序，栈恢复到用户运行栈。只有注册了自己的缺页中断服务程序的用户进程才会分配用户错误栈。

物理页面。只有这样在进入内核的 trap 中在作物栈中防止相应信息才不会引发缺页中断，所

以这个栈是自己创建的。

Exercise 5. Implement the code in kern/trap.c required to dispatch page faults the user-mode handler. Be sure to take appropriate precautions when writing into the exception stack. (What happens if the user environment runs out of space on the exception stack?)

这里需要我们完成对用户态发生的页错误的处理。

有两种可能会造成缺页中断：

1、用户程序正常运行中访问到一个错误地址，触发页错误中断

2、用户定义的页错误处理程序在运行时访问到错误地址，同样也会触发页错误中断。

因为这里 1 进入中断处理后，会发生 2 的缺页中断，而 2 又可能再次引发该类型的缺页中断，如此形成了处理的递归。

page_fault_handler 函数要做的事，就是将触发页错误的进程信息保存到 UTrapframe，并将它压入到用户错误栈，然后进入新的页错误处理程序开始运行。

正常用户进程执行出现页错误时：用户运行栈->内核态系统栈(被 Os 捕捉)->用户错误栈。错误处理程序出现页错误时：用户错误栈->内核态系统栈->用户错误栈。这里错误处理程序实质上还是用户进程。

在 page_fault_handler 中应该要做以下几步：

1、判断发生页错误的袁锦城是否已经运行在用户错误栈上，如果是，则中断程序进行的递归，此时要在错误栈中压入一个空字和一个 UTrapframe，否则就是正常用户进程发生页错误，则在错误栈的栈顶放入一个 UTrapframe。

2、检查新需要的错误栈空间是否已经按照用户可写的权限正确映射。因为该块空间早在运行之前中断错误处理程序申请的。因为错误栈大小被固定，所以有可能溢出，需要检查。

3、将发生错误的进程的运行信息保存在 UTrapframe 中。

4、开始切换到用户定义的中断错误处理程序开始运行，因为该程序也是在用户进程中运行，故我们只需修改一下栈顶的位置和指令的入口就可以 env_run 了。

Exercise 6. Implement the _pgfault_upcall routine in lib/pfentry.S. The interesting part is returning to the original point in the user code that caused the page fault. You'll return directly there, without going back through the kernel. The hard part is simultaneously switching stacks and re-loading the EIP.

_pgfault_upcall 是所有用户页错误处理程序的入口，有这里调用用户自定义的处理程序，并在处理完成后，从错误栈中保存的 UTrapframe 中恢复相应信息，然后跳回到发生错误之前的指令，恢复原来的进程运行。数据存储在栈中，又要同时做到恢复 EIP 和 ESP。

```
+-----+ <---- UXSTACKTOP
|       |
|       |
|       :
|       .   :
|       :
|       :
+-----+ <---- trap-time ESP
|   Reserved 4 bytes   |
|-----|
|   trap-time esp      |
+-----+
|   trap-time eflags   |
+-----+
|   trap-time eip      |
+-----+ <---- start of struct PushRegs
|   trap-time eax      |
+-----+
|   trap-time ecx      |
+-----+
|       .   :
|       .   :
|       .   :
+-----+
|   trap-time edi      |
+-----+ <---- end of struct PushRegs
|   tf_err (error code) |
+-----+
|   fault_va           |
+-----+ <---- %esp when handler is run
```

我们在 handler 当中对用户错误栈进行堆栈的结果。在 trap_time esp 上的空间，我们特意留下了一个 4 字节空间(减去了一个常量 4)，这是中断程序递归的情形，没有该空间的情形就是用户进程直接出错。先将栈中的 trap-time esp 减去 4，如果是中断程序递归的情形，esp 减去 4 以后就是空出 4 个字节的首址，否则 esp 还是原来用户运行栈的栈顶：

```
+-----+ <---- trap-time ESP
| Reserved 4 bytes |
|-----| <---- %eax, namely trap-time ESP - 4
| trap-time esp - 4 |
+-----+
| trap-time eflags |
+-----+
| trap-time eip |
+-----+
```

然后我们将原出错程序的 EIP 放入空出的四个字节当中，以便于我们后期使用：

```
+-----+ <---- trap-time ESP
| trap-time eip |
|-----| <---- %eax, namely trap-time ESP - 4
| trap-time esp - 4 |
+-----+
| trap-time eflags |
+-----+
| trap-time eip |
+-----+
```

恢复所有的通用寄存器，这个指令过后所有的通用寄存器都不用能在使用。

紧接着恢复 EFLAGS 标志寄存器，得到布局：

```
+-----+ <---- trap-time ESP
| trap-time eip |
|-----|
| trap-time esp - 4 |
+-----+ <---- %esp
| trap-time eflags |
+-----+
| trap-time eip |
+-----+
```

这时再使用 ret 指令，它是 pop %eip 的作用。得到即为正确的结果。

在我们对 tf_esp 赋值的时候，使用的 utf 的地址。cpu 切换到用户态，我们知道它影响 ret 指令的操作，同级返回 ret 指令只 pop eip，不同特权级会 pop ip 和 cs(call 也一样)，因为这里的 cs 是不变的，我们知道错误处理程序还是用户进程。



Exercise 7. Finish set_pgfault_handler() in lib/pgfault.c.

这个需要调用 sys_env_set_pgfault_upcall 函数完成。

我们结合以上内容，得到整个过程：



图 6-2. 页错误处理流程

Part B: Copy-on-Write Fork

这一节的内容主要为用户态进程创建子进程。Fork 函数使用了一种叫做 COW 的技术实现。在 fork 的函数调用当中需要复制父进程的地址空间到子进程中。这里的复制只是进程的基本资源的复制，如 task_struct 结构、系统空间堆栈、页面表等等。传统的 fork 系统调用直接把所有资源复制到新创建的进程。这种实现效率比较低下，因为拷贝的数据也许不会共享，如果新线程打算执行一个新的映像，复制的消耗就属于无用功了。所以 Jos 当中我们使用 copy-on-write 页实现。

该技术可以推迟甚至免去拷贝数据的技术，即内核此时并不复制整个进程地址空间，而是让父进程和子进程共享同一份拷贝。只有在需要写入的时候，数据才会被复制，从而使得各个进程拥有各自的拷贝。也就是说资源的复制只有在需要写入数据时才会进行，之前都是以制度的方式保存。一些页在不会被写入的情形是不会被复制的。Fork 的实际开销就是复制父进程的页表以及给予子进程创建唯一的进程描述符。(全部拷贝的方法见 dumpfork)

Exercise 8. Implement fork and pgfault in lib/fork.c.

Test your code with the forktree program. It should produce the following messages, with interspersed 'new env', 'free env', and 'exiting gracefully' messages. The messages may not appear in this order, and the environment IDs may be different.

```

1001: I am ''
1802: I am '0'
2801: I am '00'
3802: I am '000'
2003: I am '1'
5001: I am '11'
4802: I am '10'
6801: I am '100'
5803: I am '110'
3004: I am '01'
8001: I am '011'
7803: I am '010'
4005: I am '001'
6006: I am '111'
7007: I am '101'

```

pgfault 主要是用来做页错误处理。看注释我们知道它的基本步骤，给定一个虚拟地址，构造新的虚拟地址找到其对应的页目录表项和页表表项的方法在第二章说明了。即使用 UVPT[31:22]找到的始终为整个目录的首地址。在 lib/entry.S 当中，我们可以看到 Jos 里

用到了类似的方法：

.set vpd, (UVPT+(UVPT>>12)*4)

这里 vpd 实际上就是 UVPT[31:22]||UVPT[31:22]||0000000000|00。

在 memlayout.h 当中，我们知道假设待查询的虚拟地址为 va，则

对应页目录表项为： vpd[VPD(va)]， VPD 等价于 PDX 宏

对应页表表项为： vpt[VPN(va)]， VPN 等价于 PPN 宏

奇怪的是 vpd、vpt 定义的是一个没有 size 的数组，其实这样就相当于是首地址+offset*size。

pgfault 是一个具体处理 page_fault 的函数，它根据情况分配新野或者进行其他工作，当出现页报错时，主要对标记为可写或者 COW 的页面分配新的页面，复制旧页的数据到新页并映射到旧页的地址处。

duppage 将父进程的页表空间映射到子进程中，即共享数据，并都标记为 COW，为了以后任意进程写数据时产生页错误，为其分配新的一页。

```
#define UTEMP ((void*) PTSIZE)
// Used for temporary page mappings for the user page-fault
handler
```

为了写 fork，先让我们大体了解一下 dumpfork 的过程。duppage 这个函数使用了三个系统调用，第一个是为子环境分配一个页，并映射到 dstenv 的 addr 上，然后将该页映射到 UTEMP 下的一个 page 中，由于现在是在父环境中运行，所以需要这么做，然后把父进程的 addr 对应页的内容复制到 UTEMP 中，然后删除父环境中 UTEMP 的映射，这里并没有删除页，因为它还映射到子环境中，就是说 pp_ref > 0。

fork 则是创建新进程的入口，使用 env_alloc 创建一个新的进程，然后扫描父进程的整个地址空间将其映射到子进程相关的页表中，对于父进程，返回子进程的金盛好，对于子进程，返回 0。其主要流程如下：

- 1、首先调用 set_pgfault_handler 函数对 pgfault 处理函数进行注册。
- 2、调用 sys_exofork 创建一个新的进程。
- 3、映射可写或者 COW 的页都为 COW 的页。
- 4、为子进程分配 exception stack。
- 5、为子进程设置用户级的也错误处理句柄。
- 6、标记子进程为 runnable

首先调用的是 set_pgfault_handler，而不是系统调用 sys_env_set_pgfault_upcall，因为前者会检查用户程序的错误栈是否存在，如果没有则分配相应空间。我们无法得知父进程在调用 fork 前是否已经为错误栈分配了空间，所以不能直接调用后者

然后拷贝从 0 到 UXSTACKTOP 之间的所有用户页面。因为我们在创建用户 env 时，已经在 env_setup_vm 中将 UTOP（即 UXSTACKTOP）之上的所有页面都映射到和内核一样，但是不要映射错误栈的空间，因为我们马上要给它创建新物理页。

为子进程设置页错误处理程序。因为使用 env_alloc 创建的 env 的处理程序指针都为空，但是这时我们已经明确的为其错误栈分配了物理页面，所以可以直接使用系统调用指定错误处理的入口了，_pgfault_upcall 为所有用户页错误处理程序的总入口。

做到这里的事后，我为了方便一下，将调用 duppage 进行了一下修改：

```
//pte_t pte;
//pde_t pde;
//scan from UTEXT to UXSTACKTOP - PGSIZE
for(addr = UTEXT; addr < UXSTACKTOP - PGSIZE; addr += PGSIZE){
    /*pte = vpt[VPN(addr)];
    pde = vpd[VPD(addr)];
    if((pde & PTE_P) > 0 && (pte & PTE_P) > 0 && (pte & PTE_U) > 0){
        duppage(newenv, VPN(addr));
    }*/
}
```

结果发生了意想之外的错误，找了好久都没找到，还以为是之前的写错了，后来修改成：

```
for (addr = UTEXT; addr < UXSTACKTOP - PGSIZE; addr += PGSIZE) {
    if ((vpd[VPD(addr)] & PTE_P) > 0 && (vpt[VPN(addr)] & PTE_P) > 0 && (vpt[VPN(addr)] & PTE_U) > 0)
        duppage (envid, VPN(addr));
}
```

这样就过了 forktree 的案例。

对于 sfork 的思想，实际上和 fork 差不多，只不过需要分更多的情况进行讨论，主要修改的就是对于页面的映射问题，即如果在父进程中是栈页面，则映射为 COW，否则不为 COW。如此，我们只需要在 duppage 再加上一个标记是否为 COW 属性即可。

Part C: Preemptive Multitasking and Inter-Process communication (IPC)

主要内容是实现时钟的中断调度，可剥夺调度。

可剥夺调度：抢占调度方式，进程调度程序可根据某种原则停止正在执行的进程，将分配给当前进程的处理机收回，重新分配给另一个处于就绪状态的进程。在 Jos 中我们抢占原则是时间片原则，当时间片用完或者进程被阻塞，系统就停止该进程的执行而重新进行调度。此处需要用到外部中断，即时间中断。

Exercise 9. Modify kern/trapentry.S and kern/trap.c to initialize the appropriate entries in the IDT and provide handlers for IRQs 0 through 15. Then modify the code in env_alloc() in kern/env.c to ensure that user environments are always run with interrupts enabled.

picirq.c 将 IRQs0-15 映射到 IDT 表中 IRQ_OFFSET 到 IRQ_OFFSET+15。在 Jos 中，内核态一直是禁止外部中断的，外部中断时 eflags 标志寄存器的 FL_IF 位控制的，在 Jos 中我们只在进入和离开用户模式时保存和恢复 eflags 寄存器。并确保 FL_IF 标志在用户进程运行时置位。

还有在 env_alloc 的时候，需要将 eflags 和 FL_IF 相与来开中断。

Handling Clock Interrupts

Exercise 10. Modify the kernel's trap_dispatch() function so that it calls sched_yield() to find and run a different environment whenever a clock interrupt takes place.

我们只需加上这一段到 trap.c 和 trapentry.S 当中就可以测试 spin，和以前中断处理是一样的，在 trap_dispatch 里面需要根据中断号选择处理方式，进行进程调度即可。

Inter-Process communication (IPC)

Exercise 11. Implement sys_ipc_recv and sys_ipc_can_send in kern/syscall.c. When you call envid2env in these routines, you should set the checkperm flag to 0, meaning that any environment is allowed to send IPC messages to any other environment, and the kernel does no special permission checking other than verifying that the target envid is valid.

Then implement the ipc_recv and ipc_send functions in lib/ipc.c.

在前面，Jos 考虑操作系统的隔离性，每个进程都独占机器，互不干扰。但如果不行通信，很容易发生死锁现象。所以操作系统必须提高进程间通信功能，从而使进程可以协同完成一个任务。消息方式或为一个单字，或为一个页的映射，后者可以高效传送大量数据。

传统上有很多进程间通信方式，比如信号量，管道等，我们在 Jos 当中试用简单的消息传递来实现进程间通信。

再看 Env 结构，会发现 lab4 的注释下面多了 5 个变量：

env_ipc_recving :

当进程使用 sys_ipc_recv 等待信息时，会将这个成员置为 1，然后阻塞等待；当一个进程像它发消息解除阻塞后，发送进程将此成员修改为 0。

env_ipc_dstvva :

如果进程要接受消息，并且是传送页，则该地址 \leq UTOP

env_ipc_value :

若等待消息的进程接受到了消息，发送方将接受方此成员置为消息值

env_ipc_from :
发送方负责设置该成员为自己的 envid 号
env_ipc_perm :
如果进程要接受消息，并且传送页，那么发送方发送页以后将传送的页权限传给这个成员。

实现 IPC 我们首先需要完成 kern/syscall.c 中的 sys_ipc_recv 和 sys_ipc_try_send 系统调用。主要功能如下：

sys_ipc_recv: 阻塞调用进程直到接收到一个消息，然后设置 env 中的相应项，调整调度函数。这里需要设置返回值 eax 为 0，即永不执行 return 语句，直接调用调度函数即可。

sys_ipc_try_send: 发生一个消息到 envid 进程，当 srcva 为 0 时，传送一个单字，否则传送一页，即将当前进程的 srcva 地址以外的页映射到接收进程同一地址处。注意在以上函数上调用 envid2env 时设置 checkperm 为 0，这里的 checkperm 是 envid2env 函数的第三个参数，这样任何进程都可以发送消息给任何其他进程，内核只需检查目标进程是否存在。一个进程调用 sys_ipc_recv 接收一个消息，这个系统调用将会阻塞当前进程，除非它接收到一个消息。单个进程等待接收一个消息时，任何进程都可以发消息给它。一个进程也可以调用 sys_ipc_try_send 发送一个消息给指定的进程，如果指定的进程正在阻塞等待消息，则发送消息并返回 0。

完成系统调用的 IPC 后，需要在 lib 当中修改完成 ipc.c 文件中的 ipc_recv 和 ipc_send 函数。前者主要是调用 sys_ipc_recv 函数接收消息，并返回接收的 value 的值，后者则调用 sys_ipc_try_send 函数发送 val 到进程 to_env，直到发送成功。

两个强调的地方：

1、在系统调用那里，我们也许使用了 panic 来防止系统调用返回负数的情况，在这里，我们要去掉这句话，因为事实上，再一个进程在 send 的时候，直到让指定进程收到信息为止，事实上是一直都在系统调用 send 的，也就是说直到对方接受为止，这些系统调用返回值都是负数(当然指定是-E_NOT_RECV，其他的不行)。

2、再就是挑战题，这个挑战题属于开放题，大家可以想各种策略去优化，比如可以通过通信成功的次数来决定当前接受那个向它发送消息但成功最少的一个。

重要附录参考代码

sched_yield 部分代码(抢占式，带有优先级)

```

for(able = run+1; round<NENV; able++,round++){
    if(able >= envs+NENV){
        able = envs+1;
    }
    if(able->env_status == ENV_RUNNABLE){
        if(level > able->env_priority){
            run = able;//decide to run
            level = able->env_priority;
        }
    }
}
if(level<21){//there must be some one can be run
    env_run(run);
}
// Run the special idle environment when nothing else is runnable.
if (envs[0].env_status == ENV_RUNNABLE)
    env_run(&envs[0]);
else {
    cprintf("Destroyed all environments - nothing more to do!\n");
    while (1)
        monitor(NULL);
}

```

```

fork(void)
{
    // LAB 4: Your code here.
    //set_pgfault_handler
    set_pgfault_handler(pgfault);
    //cprintf("finish set pgfault handler\n");
    //new a child
    envid_t newenv;
    uint32_t addr;
    int r;
    newenv = sys_exofork(); //new envid
    //cprintf("finish create a env %d\n", newenv);
    if(newenv < 0) panic("envid < 0 ,failed in exofork\n");
    if(newenv == 0){
        //ENVX get the id's low 10 bit
        env = &envs[ENVX(sys_getenvid())];
        return 0;
    }
    //map all the page (W or COW) to COW
    //duppage(newenv, 1);
    //pte_t pte;
    //pde_t pde;
    //scan from UTEXT to UXSTACKTOP - PGSIZE
    for(addr = UTEXT; addr < UXSTACKTOP - PGSIZE; addr += PGSIZE){
        *pte = vpt[VPN(addr)];
        pde = vpd[VPD(addr)];
        if((pde & PTE_P) > 0 && (pte & PTE_P) > 0 && (pte & PTE_U) > 0){
            duppage(newenv, VPN(addr));
        }/*
        if((vpd[VPD(addr)] & PTE_P) > 0 && (vpt[VPN(addr)] & PTE_P) > 0 && (vpt[VPN(addr)] & PTE_U) > 0)
            duppage (newenv, VPN(addr));
        */
    }
    //cprintf("finish map PGSIZE\n");
    //user exception stack
    if((r = sys_page_alloc(newenv, (void *)(UXSTACKTOP - PGSIZE), PTE_P|PTE_U|PTE_W)) < 0){
        panic("fork alloc page failed\n");
    }
    //
    extern void _pgfault_upcall(void);
    sys_env_set_pgfault_upcall(newenv, _pgfault_upcall);
    //panic("fork not implemented");
    //set it runnable
    if((r = sys_env_set_status(newenv, ENV_RUNNABLE)) < 0){
        panic("set runnable failed\n");
    }
    return newenv;
}

```

```

sys_ipc_recv(void *dstva)
{
    // LAB 4: Your code here.
    if((dstva < (void *)UTOP) && ROUNDDOWN(dstva, PGSIZE) != dstva){ //receive a page of data
        panic("this page is not page-aligned\n");
        return -E_INVAL;
        //panic("this page is not page-aligned\n");
    }
    curenv->env_ipc_dstva = dstva;
    curenv->env_ipc_recving = 1; //wait for recv
    curenv->env_ipc_from = 0;
    curenv->env_status = ENV_NOT_RUNNABLE;
    curenv->env_tf.tf_regs.reg_eax = 0;
    //why from is 0;
    sched_yield();
    //panic("sys_ipc_recv not implemented");
    return 0;
}

```

```

sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)
{
    // LAB 4: Your code here.
    struct Env *dstenv;
    int r;
    if((r = envid2env(envid, &dstenv, 0)) < 0) return -E_BAD_ENV;
    if(!dstenv->env_ipc_recving || dstenv->env_ipc_from != 0) return -E_IPC_NOT_RECV;
    if((srcva < (void *)UTOP) && ROUNDOWN(srcva, PGSIZE) != srcva){//receive a page of data
        //cprintf("this page is not page-aligned\n");
        return -E_INVAL;
        //panic("this page is not page-aligned\n");
    }
    //cprintf("this page is page-aligned\n");
    if(srcva < (void *)UTOP){
        if((perm & PTE_U) == 0 && (perm & PTE_P) == 0) return -E_INVAL;
        if((perm & ~PTE_USER) > 0) return -E_INVAL;
    }
    //cprintf("perm normol\n");
    pte_t *pte;
    struct Page *pg;
    //not mapped in current env
    if(srcva < (void *)UTOP){
        if((pg = page_lookup(curenv->env_pgdir, srcva, &pte)) == NULL) return -E_INVAL;
        if((*pte & PTE_W) == 0 && (perm & PTE_W) > 0) return -E_INVAL;
    }
    //if env_ipc_dstva != 0 , send a page
    if(srcva >= (void *)UTOP) dstenv->env_ipc_perm = 0;
    if(srcva < (void *)UTOP && dstenv->env_ipc_dstva <(void *)UTOP){//!= 0
        if(page_insert(dstenv->env_pgdir, pg, dstenv->env_ipc_dstva, perm) < 0) return -E_NO_MEM;
        dstenv->env_ipc_perm = perm;
    }
    dstenv->env_ipc_recving = 0;
    dstenv->env_ipc_from = curenv->env_id;
    dstenv->env_ipc_value = value;
    dstenv->env_status = ENV_RUNNABLE;
    //dstenv->env_tf.tf_regs.reg_eax = 0;
    return 0;
}

```

File Systems and Spawn

Author: Xin Yao

这里提到微内核和单内核的概念，我们的 Jos 采用微内核的方式。

微内核中，大部分内核作为单独的进程在特权下运行，他们通过消息传递进行通讯。在典型情况下，每个概念模块都有一个进程。因此，假如在设计当中有一个系统调用模块，那么就必然有一个的进程来接受系统调用，并与该模块进行通讯以完成所需任务。当系统调用模块给文件系统模块发消息时，消息直接通过内核转发。这宗法师有利于模块间隔离。最根本还是要保证微内核尽量小，这样只需把微内核本身进行移植后就可以将整个 Os 移植到新平台。其他模块都不直接依赖硬件，而是依赖于微内核。微内核的长处就是不影响系统其他部分就可以更加高效地实现代替现有的文件系统模块的工作会更加容易。我们甚至能够在系统运行时将研发的新模块进行直接替换。另一个长处就是长期不需要的模块不会被加载到内存当中，因此微内核就是更有效的利用内存。

单内核是一个很大的进程，它内部又可以被划分为若干模块。其模块间通讯是通过直接调用其他模块的函数实现的，而不是消息传递。

本文件系统不支持权限、链接、时间等，属于最基础的一种，Unix 系统将磁盘划分成两个域：inode 和 data 域。inode 用来保存文件的状态属性，以及所指向数据块的指针。data 则包含了 data 块，这里存放文件的内容和目录的元信息(包含的文件名以及指向文件 i 节点的指针)。如果文件系统中的多个目录都指向文件的 inode 结点，则称此文件为硬连接的，在 Jos 系统中，由于不支持硬链接，用不到硬链接，有兴趣的话可以自己尝试尝试。这里只要把文件元数据存放在所属的目录里即可。

1、扇区块，512 字节，是磁盘的物理属性

2、块，文件系统分配和使用磁盘空间的单位，是 Sector 的整数倍。

3、超级块：一个特定的物理位置，一般是第一块和最后一块。包含描述文件系统的元数据：block 大小、磁盘大小、根目录位置、文件系统挂载时间、上次进行磁盘检查时间等。大多数真正文件系统维护多个超级块，通过复制分散才防止超级块损坏带来的问题。本实验 inc/fs.h 的 Super 结构定义了磁盘布局，Jos 里的一个超级块为 block1，block0 在 boot sector 和磁盘分区表里。

4、块位图，文件系统必须管理存储块一个时刻仅用于一种目的。使用块位图来管理空闲磁盘块，容易存储，并节省了磁盘空间。这个空闲块位图需要极大的空间，为每一个磁盘块设置一个位。Jos 中，block2 为块位图，涵盖所有磁盘块。

5、文件元数据，文件运输局有 inc/fs.h 的结构 File 描述：包括文件名、大小、类型和指向文件所包含的磁盘块的指针。有些变量再内存里才有意义，所在在读 file 到内存中时，都要将这些域清空。

File 结构可以表示文件或目录，两者的区别在于 type 域。文件系统不会解析代表文件的 file 的数据块内容，但是会解析代表目录的 file 结构数据块内容来获得其所包含的文件和子目录的信息。它还保存了文件头 10 个块的全部内容，大于 10 个块的文件需要间接寻址。

直接磁盘块：在 File 的块数组存储了一个文件的前 10 个块的块号，可以直接访问。不超过 $10 * 4096 = 40960$ 共 40KB 的小文件，其文件的所有块号都可以直接挡在 File 结构当中。

间接磁盘块：对于大于 40KB 的文件，需要另外分配一个磁盘块，叫做间接磁盘块，可以保存 $4096 / 4 = 1024$ 个磁盘块号。在 Jos 系统档子那个，为了使等级目录更简单，采取不适用间接磁盘块的前 10 个块号，所以该文件最大支持 4M 的文件大小，一般很多文件系统都是用了多级间接磁盘块。

文件系统磁盘块图形表示和文件元数据表示如下：

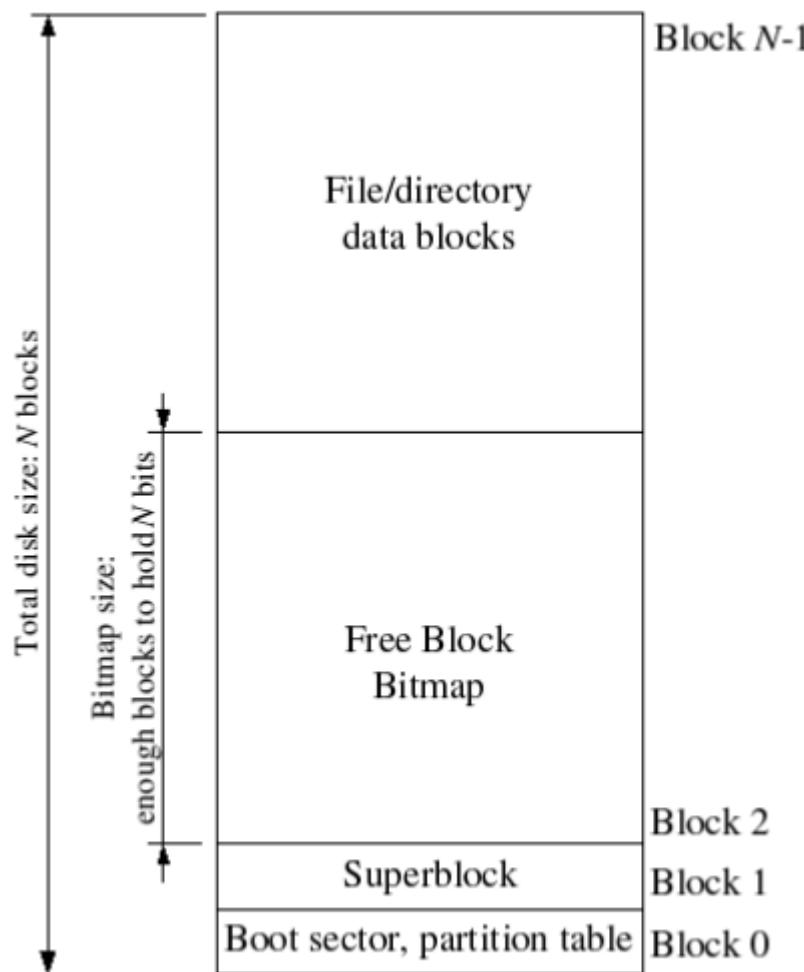
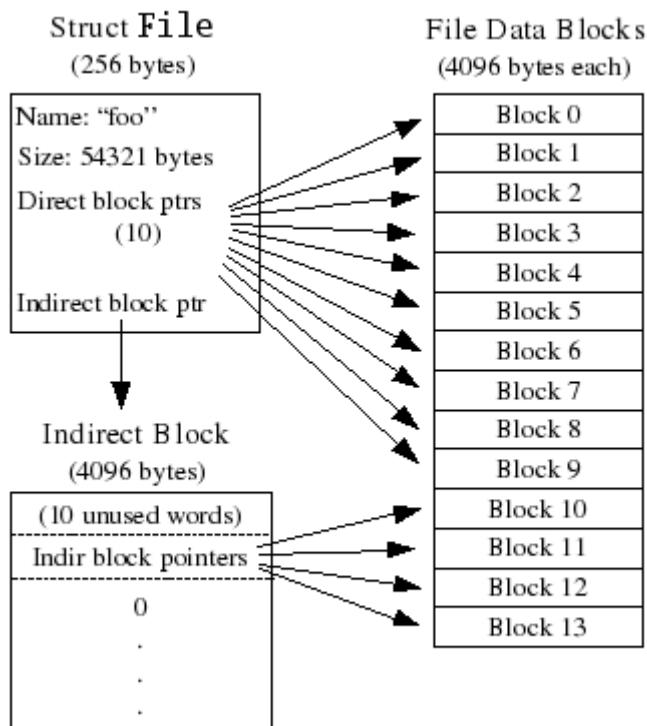


图 7-1. 磁盘块



Part A: The File System Server

本次主要集中查看 fs 目录和几个相关文件。先介绍一些构成文件系统的数据结构：

File 结构

```
struct File {
    char f_name[MAXNAMELEN];      // filename
    off_t f_size;                // file size in bytes
    uint32_t f_type;              // file type

    // Block pointers.
    // A block is allocated iff its value is != 0.
    uint32_t f_direct[NDIRECT];   // direct blocks
    uint32_t f_indirect;          // indirect block

    // Points to the directory in which this file lives.
    // Meaningful only in memory; the value on disk can be garbage.
    // dir_lookup() sets the value when required.
    struct File *f_dir;

    // Pad out to 256 bytes; must do arithmetic in case we're compiling
    // fsformat on a 64-bit machine.
    uint8_t f_pad[256 - MAXNAMELEN - 8 - 4*NDIRECT - 4 - sizeof(struct File*)];
} __attribute__((packed)); // required only on some 64-bit machines
```

其图样表示已经在前面提到过了，文件包括 7 个成员：文件名、大小、类型、指向文件所包含磁盘块的指针。

Super 结构

```
struct Super {
    uint32_t s_magic;            // Magic number: FS_MAGIC
    uint32_t s_nblocks;          // Total number of blocks on disk
    struct File s_root;          // Root directory node
};
```

三个成员：文件系统的魔数，一个包含磁盘中 block 的总个数以及目录登陆点。

Dev 结构(文件描述句柄)

```
struct Dev {
    int dev_id;
    char *dev_name;
    ssize_t (*dev_read)(struct Fd *fd, void *buf, size_t len, off_t offset);
    ssize_t (*dev_write)(struct Fd *fd, const void *buf, size_t len, off_t offset);
    int (*dev_close)(struct Fd *fd);
    int (*dev_stat)(struct Fd *fd, struct Stat *stat);
    int (*dev_seek)(struct Fd *fd, off_t pos);
    int (*dev_trunc)(struct Fd *fd, off_t length);
};
```

该结构对应一个块设备，并为该设备提高常用读、写、查看状态信息等操作，其指针函数对应 lib/file.c 文件汇总的函数。

```
struct Stat {
    char st_name[MAXNAMELEN];
    off_t st_size;
    int st_isdir;
    struct Dev *st_dev;
};
```

```
struct Fd {
    int fd_dev_id;
    off_t fd_offset;
    int fd_oemode;
    union {
        // File server files
        struct FdFile fd_file;
    };
};
```

Stat 保存文件的状态信息，文件名，文件大小，文件类型以及文件所属设备
Fd 为文件句柄，对应一个文件，保存文件的操作模式，比如只读、可写。

```

struct FdFile {
    int id;
    struct File file;
};

// Return the 'struct Fd*' for file descriptor index i
#define INDEX2FD(i) ((struct Fd*) (FDTABLE + (i)*PGSIZE))
// Return the file data pointer for file descriptor index i
#define INDEX2DATA(i)  ((char*) (FILEBASE + (i)*PTSIZE))

```

INDEX2FD 返回索引节点 i 所对应的文件句柄。

INDEX2DATA 返回文件描述符索引节点 i 所对应的数据。

FILEBASE 是文件数据区域的底部，FDTABLE 是文件描述符区域的底部。

fd2data 是求出文件描述符句柄所指向的数据。

fd2num 是将文件描述符句柄 fd 转换成文件描述符索引节点。

fd_close 释放一个文件描述符句柄，通过关闭 fd 指向的相关文件并取消 fd 所映射的页。

dev_lookup 再已有设备信息表查找与 devid 匹配的设备。

Disk Access

Exercise 1. Modify your kernel's environment initialization function, env_alloc in env.c, so that it gives environment 1 I/O privilege, but never gives that privilege to any other environment.

文件系统服务器需要能访问磁盘，传统的微内核模式的操作系统往往在系统调用中增加一个 IDE 磁盘驱动器来允许文件系统去访问磁盘，而在 Jos 当中，我们将 IDE 磁盘驱动器作为文件系统进程用户的一部分。为了简化，系统采用轮询方式，而不是磁盘中断：查询 EFLAGS 寄存器中的 IOPL 位，此处可以从代码 fs/serv.c 中了解。

由于我们需要访问所有的 IDE 磁盘寄存器信息都是放在 x86 处理器的 I/O 空间，而不是映射到内存空间。如果要访问磁盘，我们需要将 I/O 特权级赋予文件系统进程。X86 处理器使用标志寄存器 EFLAGS 中的 IOPL 位来决定保护模式的代码是否拥有访问 I/O 权限。要使文件系统进程可以访问磁盘，只需要将其标志寄存器 EFLAGS 的 IOPL 置位即可。

我们的任务就是修改内核进程初始化函数：env.c 中的函数 env_alloc，使其仅给文件系统进程 I/O 权限。

这里在进程切换的时候，env_pop_tf 恢复了所有寄存器的状态，先是通过 popal 恢复了所有的通用寄存器，然后在 iret 中恢复了 eip、cs、eflags 寄存器。

对于一些文件进行说明：

ide.c: 提供 IDE 磁盘的驱动，比如对特定扇区的读写以及切换操作磁盘

bc.c: 提供吸盘的块缓存机制，当用户请求读写一块磁盘区域时，将其加载到文件系统进程的虚拟地址里，这样就可以用比较小的内存操作很大一块磁盘。(该文件在 2007 年版本中暂时还没有涉及，这一部分是我在看 2011 年的部分时添加的内容)

fs.c: 文件系统核心功能实现。

The Block Cache

Exercise 2. Implement the read_block and write_block functions in fs/fs.c. The read_block function should test to see if the requested block is already in memory, and if not, allocate a page and read in the block using ide_read. Keep in mind that there are multiple disk sectors per block/page, and that read_block needs to return the virtual address at which the requested block was mapped.

The write_block function may assume that the indicated block is already in memory, and simply writes it out to disk. We will use the VM hardware to keep

一个进程可以拥有 4G 虚拟空间，但文件系统当前可以处理最多为 3G。在 Jos 当中，我们将这 3G 空间固定在文件系统进程的地址空间，从 0x10000000(DISKMAP) 到 0xD0000000(DISKMAP+DISKMAX) 作为缓冲区，当磁盘块读入内存时，用来映射相关的页。

比如磁盘块 0 无论如何都映射到地址 0x10000000 处，磁盘块 1 映射到地址 0x10001000 处。

如下图：

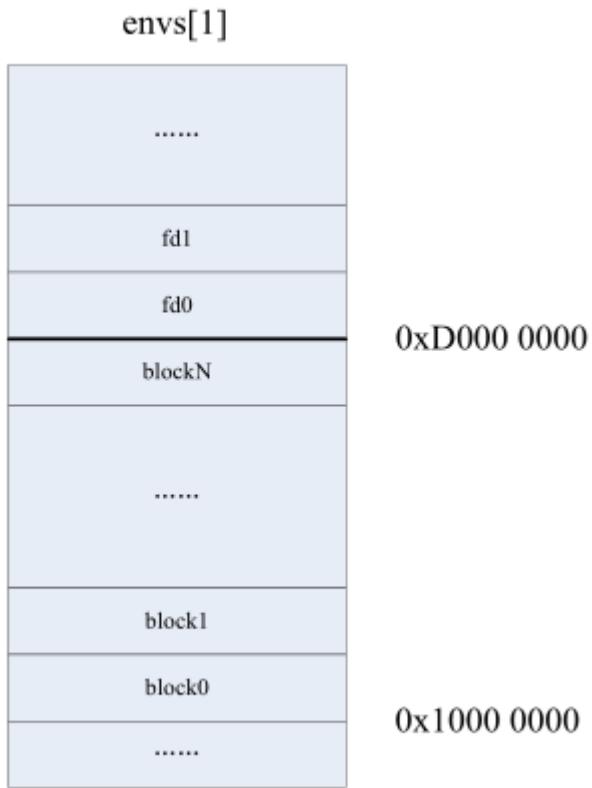


图 7-3. 块缓存

由于我们的文件系统进程的虚拟空间地址独立于系统总其他进程的虚拟地址空间，而其唯一要做的事情就是提供文件的访问，因此以这种方法保留文件系统大部分的地址空间是很有效的。

`diskaddr` 返回磁盘块对应的虚拟地址。

`block_is_mapped` 检查磁盘块 `blockno` 是否已经映射到内存。

`map_block` 检查磁盘块是否已经映射，否则分配一页保存磁盘块数据。

`ide_read` 读取磁盘块 `secno` 中 `nsecs` 个扇区数据到地址 `dst` 处。

`ide_write` 将 `src` 地址处的 `nsecs` 个扇区的数据写到磁盘块 `secno` 中。

`read_block` 将磁盘块 `blockno` 数据读入到内存中。首先检查 `blockno` 是否存在，如果没有，则需要分配一页并使用 `ide_read` 将其读入到 `blockno` 所对应的虚拟地址处。

`write_block` 将磁盘块中的当前内容写到磁盘中。判断磁盘块中数据是否需要些，我们只需要参看其页表中 `PTE_D` 位即可，当块中数据改变时，此为处理器直接设置。写回后，需要将 `PTE_D` 位清 0。

在 2011 年的版本中，`fs_init` 还做了一件事，就是使用了后来添加了 `bc.c` 中的 `bc_init` 函数，主要是安装页错误处理程序，并设置了文件系统超级块。

在 2007 年版中，我们的 `fs_init` 直接进行了设置超级块指向和空闲块位图。`read_super` 当中我们实现了通过读取 1 号磁盘块来初始化超级块指针。

现在重点说明一下 `fs/fsformat.c` 的功能，该文件的功能和它的名字一样，格式化磁盘，磁盘不格式化是不能使用的，通过这个文件将一个物理磁盘进行格式化，才能被系统识别。通过阅读 `main` 函数，我们知道使用 `opendisk` 创建了一个磁盘文件，并初始化超级块，并创建一个根目录，并将超级块的文件名定义为'/'，这说明了超级块的文件名事实上就是我们的根目录的名称。但是在 2011 年的版本当中，就不是这样，它会使用另外一个被称作 `root` 的结构在作为根目录名使用。

The Block Bitmap

之前提到过 Jos 文件系统第二个功能就是 `read_bitmap` 来检查磁盘的位示图。Jos 总是将这个块位图都读取到内存当中。

Exercise 3. Implement `read_bitmap`. It should check that all of the "reserved" blocks in the file system - block 0, block 1 (the superblock), and all the blocks holding the block bitmap itself, are marked in-use. Use the provided `block_is_free` routine for this purpose. You may simply panic if the file system is invalid.

Exercise 4. Use `block_is_free` as a model to implement `alloc_block_num`, which scans the block bitmap for a free block, marks that block in-use, and returns the block number. When you allocate a block, you should *immediately* flush the changed bitmap block to disk with `write_block`, to help file system consistency.

全局变量为 `bitmap`, `block_is_free` 检查磁盘块 `blockno` 的位图是否空闲, `read_bitmap` 检查文件系统保留的所有磁盘块的块位图, 将其独到内存, 标记为可用, 并是全局变量 `bitmap` 指向块位图的首地址。`alloc_block_num` 用 `block_is_free` 来茶轴一个空闲的位示图, 分配后执行 `write_block`, 将位示图块的状态进行刷新, 以保持文件系统的一致性。

File Operations

Exercise 5. Fill in the remaining functions in `fs/fs.c` that implement "top-level" file operations: `file_open`, `file_get_block`, `file_truncate_blocks`, and `file_flush`.

一些文件操作, 属于面向用户对象使用的, 其表示进程已经打开文件, 可以直接对文件进行处理, 而不是超级块、索引节点。这些基本的文件操作包括解析、管理 File 结构, 分配或寻找指定文件的指定块, 扫描并管理目录文件的表项, 从根目录便利文件系统, 得到文件的绝对路径等。

`walk_path`: 从根目录开始查找, 解析路径 `path`, 保存期路径指向的文件。

`file_map_block`: 将文件的第 `filebno` 磁盘块映射到文件系统中的 `diskbno` 块。

`file_open`: 利用 `walk_path` 打开路径 `path` 中的文件, 获得文件句柄。

`file_get_block`: 利用 `file_map_block` 和 `read_block` 从文件中获得 `filebno` 磁盘块的数据, 是 `blk` 指向其首地址。

`file_truncate_blocks`: 将文件大小截断为新的大小 `newsize`, 并释放点文件中没有使用的磁盘块, 包括间接磁盘块。

`file_flush`: 将文件刷新到磁盘上, 支队文件中标记为 `dirty` 的数据操作。

Client/Server File System Access

Jos 当中的基本操作 `write` 和 `read` 是 Jos 中的文件系统通过内核中的 IPC 将文件系统的页映射到各户进程, 然后客户进程直接读写。因此我们就要使用文件系统进程允许用户进程使用文件进程。C/S 结构的文件系统访问, 使得文件系统可以被其他进程访问。每一个客户进程都使用 `ipc_send` 发送消息给服务器, 即提出访问文件系统的请求, 然后使用 `ipc_recv` 等待响应, 完成文件操作。

在 `inc/fs.h` 当中, 有很多数据结构:

`Fsreq_open` 是客户进程想文件系统发送打开文件请求, 以 `req_omode` 模式打开文件

`Fsreq_map` 是客户进程想文件系统服务器发送映射文件请求, 大小为 `req_offset`

`Fsreq_set_size` 是客户进程想文件系统服务器发送设置文件大小请求, 大小为 `req_size`

`Fsreq_close` 是客户进程向文件系统服务器发送关闭文件请求

`Fsreq_dirty` 是客户进程想文件系统服务器发送数据修改请求, 偏移为 `req_offset`

`Fsreq_remove` 是客户进程向文件系统服务器发送删除文件请求, 文件路径为 `req_path`

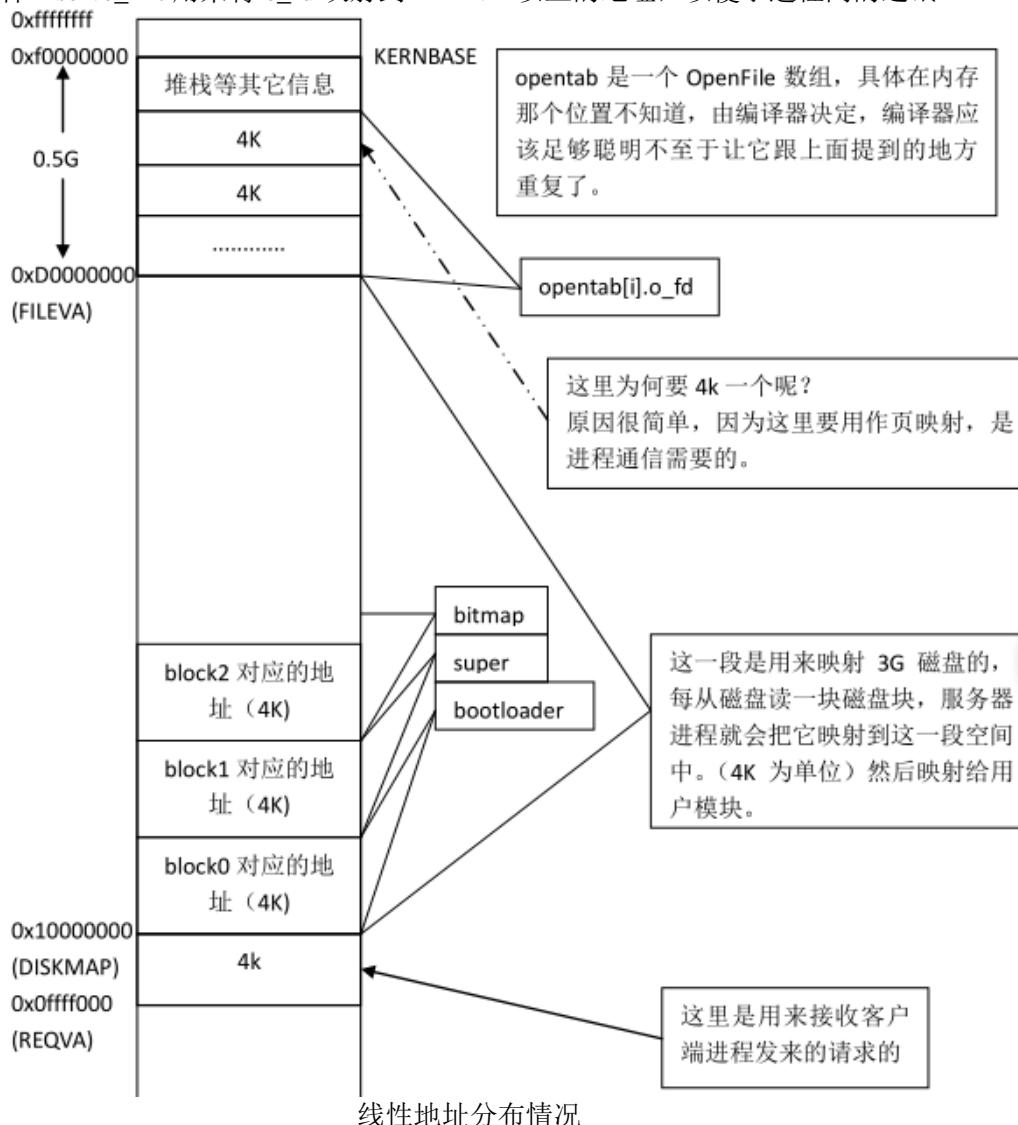
对于 client stubs, 系统已经在 `lib/fsipc.c` 中实现了, 使用系统中 IPC 的协议, 当客户进程需要访问问及爱你系统时, 使用 client stubs 执行相应的请求。我们需要了解 client stubs 请求文件系统服务。

`fsipc` 都是用过此函数向服务器发送一个请求, 并等待应答。次函数使用 `ipc_send` 向服务器发送请求, `ipc_recv` 接受服务器的消息, 其中 `type` 为请求类型, 如 `FSREQ_OPEN`、

FSREQ_CLOSE 等, fsreq 为包含请求数据的页, dstva 是接受文件系统数据的虚拟地址, perm 为接收页表的页属性。与对应数据结构类似的, 我们有 fsipc 的一系列函数: map、set_size、close、dirty、remove、sync。对于 server stubs, 我们在 serv.c 中实现, 这些 stubs 从客户进程接受 IPC 请求, 解析这些命令。

```
struct OpenFile {
    uint32_t o_fileid; // file id
    struct File *o_file; // mapped descriptor for open file
    int o_mode; // open mode
    struct Fd *o_fd; // Fd page
};
```

`o_file` 用来操作底层文件系统, `o_fd` 是用来传到客户进程的, 客户进程根据 `o_fd` 的信息操作文件。serve_init 用来将 `o_fd` 映射到 FILEVA 以上的地址, 以便于进程间的通讯

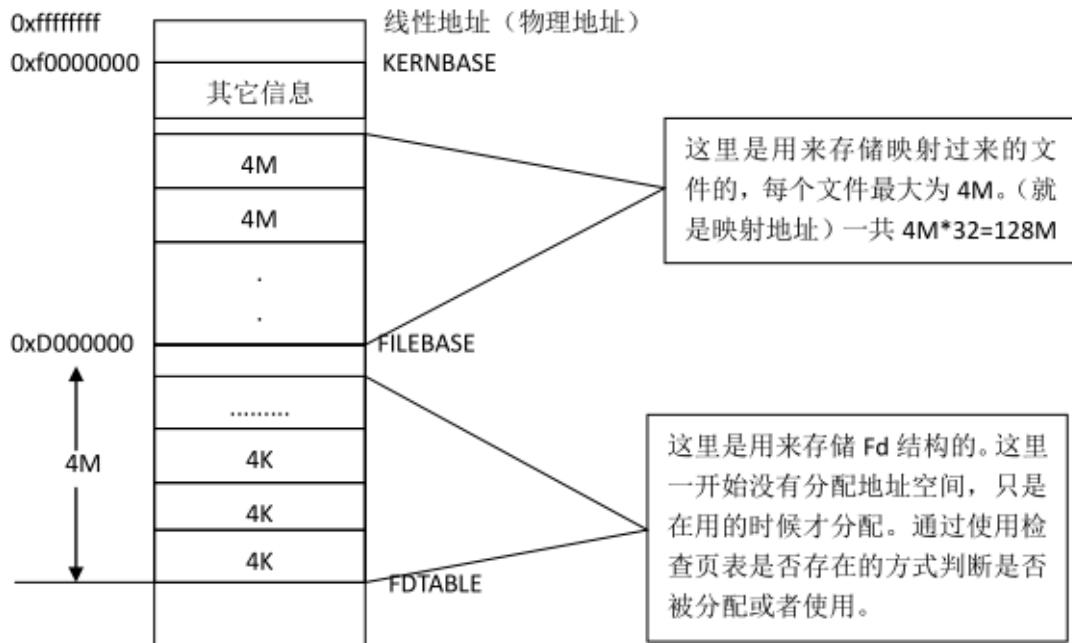


最关键的结构为 Fd, 用户进程通过 Fd 保存的信息实现相关的文件操作。具体的通信函数我们在前面已经提到了。发送函数格式为 fsipc(FSREQ_OPEN, req, fd, &perm)第一个参数是操作类型, 第二个是传递的参数, 第三个是 fd(即 openfile 的 o_fd), 第四个是权限。

四个模块合作流程为: 用户需要文件操作->用户进程向服务器进程发出请求->服务器进程操作底层文件模块->服务器进程向用户进程发出已经完成操作的信息, 并把相关信息传回->用户进程获得相关文件。

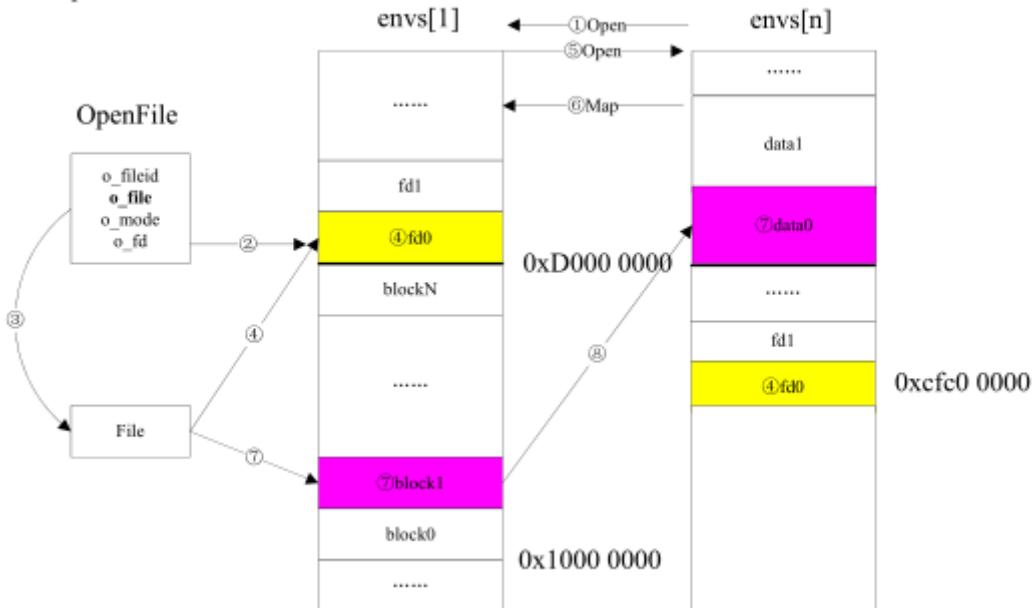
底层使用 File 结构, 而用户多使用 fd 结构, 服务器将两者用 OpenFile 联系起来。

用户模块如下表示：



客户进程打开文件的示意图

以open操作为例



客户进程用过 FSIPC 发送打开文件请求，文件系统服务器寻找一个可用的文件句柄，并将文件映射到文件句柄，然后将句柄映射到客户进程，客户进程将数据 data0 映射到文件系统的磁盘块，发送 map 命令，文件系统中将文件句柄对应的磁盘块映射到客户端进程中，完成文件打开操作。

Exercise 6. The server stubs are located in the file server itself, implemented in `fs/serv.c`. These stubs accept IPC requests from clients, decode and validate the arguments, and serve those requests using the file access functions in `fs/fs.c`. We have provided a skeleton for this server stub code, but you will need to fill it out. Use the client stubs in `lib/fsipc.c` to help you figure out the exact protocol between the client and the server.

在 `serv.c` 当中的 `umain` 函数，我们可以看到整个流动过程，也很容易看到在 `serve()` 里有一个死循环，总在等待处理外来信息。

重要附录参考代码

read_bitmap 主要用来设置位图法管理空闲磁盘

```

bitmap_blkno = super->s_nblocks / BLKBITSIZE;
if(super->s_nblocks % BLKBITSIZE) bitmap_blkno++;
for(i=0;i<bitmap_blkno;i++){
    if((r = read_block(2+i, &blk)) < 0){
        panic("cannot read bitmap block: %e", r);}
}
bitmap = (uint32_t *)diskaddr(2);
//panic("read_bitmap not implemented");

// Make sure the reserved and root blocks are marked in-use.
assert(!block_is_free(0));
assert(!block_is_free(1));
assert(bitmap);

// Make sure that the bitmap blocks are marked in-use.
// LAB 5: Your code here.
for(i=0;i<bitmap_blkno;i++){
    assert(!block_is_free(2+i));
}

```

```

alloc_block_num(void)
{
    // LAB 5: Your code here.
    //panic("alloc_block_num not implemented");
    int i;
    //if((r = alloc_block()) < 0)
    for(i=3; i<super->s_nblocks; i++){
        if((bitmap[i/32] && (1<<(i%32)))){//find the place
            bitmap[i/32] &= ~(1<<(i%32));
            write_block(2 + i/BLKBITSIZE);
            return i;
        }
    }
    return -E_NO_DISK;
}

```

```

write_block(uint32_t blockno)
{
    char *addr;
    int r;
    if (!block_is_mapped(blockno))
        panic("write unmapped block %08x", blockno);

    // Write the disk block and clear PTE_D.
    // LAB 5: Your code here.
    addr = diskaddr(blockno);
    if(!va_is_dirty(addr)) return;//not dirty
    if((r = ide_write(blockno * BLKSECTS, addr, BLKSECTS)) < 0){
        panic("ide_write failed\n");
    }
    //remove the PTE_D
    if((r = sys_page_map(0, addr, 0, addr, PTE_USER)) < 0){
        panic("sys_page_map failed\n");
    }
}

```

```
read_block(uint32_t blockno, char **blk)
{
    int r;
    char *addr;

    if (super && blockno >= super->s_nblocks)
        panic("reading non-existent block %08x\n", blockno);

    if (bitmap && block_is_free(blockno))
        panic("reading free block %08x\n", blockno);

    // LAB 5: Your code here.
    addr = diskaddr(blockno);
    if(blk == NULL){ *blk = addr;}//set blk to the address
    if(va_is_mapped(addr)) return 0;//has been mapped
    if((r = map_block(blockno)) < 0) return r;//map_block failed
    if((r = ide_read(blockno * BLKSECTS, addr, BLKSECTS)) < 0) return r;//ide_read fail
    //panic("read_block not implemented");
    return 0;
}
```

Open a file

```
struct Fd *fd;
int r;
if((r = fd_alloc(&fd)) < 0) return r;//alloc a fd
if((r = fsipc_open(path, mode, fd)) < 0) return r;//fsipc_open
if((r = fmap(fd, 0, fd->fd_file.file.f_size)) < 0){
    fd_close(fd, 1); return r;}//close the file if map failed
return fd2num(fd);
```

Close a file

```
int r;
// fd -> unmap
if((r = funmap(fd, fd->fd_file.file.f_size, 0, 1)) < 0) return r;
//close the file
if((r = fsipc_close(fd->fd_file.id)) < 0) return r;
return 0;
```

```
serve_map(envid_t envid, struct Fsreq_map *rq)
{
    int r;
    char *blk;
    struct OpenFile *o;
    int perm;

    _if (debug)
        cprintf("serve_map %08x %08x %08x\n", envid, rq->req_fileid, rq->req_offset);

    // Map the requested block in the client's address space
    // by using ipc_send.
    // Map read-only unless the file's open mode (o->o_mode) allows writes
    // (see the O_ flags in inc/lib.h).

    // LAB 5: Your code here.
    if((r = openfile_lookup(envid, rq->req_fileid, &o)) < 0){
        ipc_send(envid, r, 0, 0); return;
    if((r = file_get_block(o->o_file, ROUNDUP(rq->req_offset, BLKSIZE)/BLKSIZE, &blk)) < 0){
        ipc_send(envid, r, 0, 0); return;
    if((o->o_mode & O_ACCMODE) == O_RDONLY) perm = PTE_P|PTE_U;//read
    else perm = PTE_P|PTE_U|PTE_W;
    //check for the perm and send it
    ipc_send(envid, 0, blk, perm);
    return ;
    //panic("serve_map not implemented");
}}
```