

北京大学操作系统实习(实验班)报告

JOS-Lab 4:

Preemptive Multitasking

黄睿哲 00948265
huangruizhe@pku.edu.cn

May 21, 2012

目录

目录

JOS-Lab 4:.....	1
Preemptive Multitasking	1
内容一：总体概述	4
I: Part A: Multiprocessor Support and Cooperative Multitasking	4
II: Part B: Copy-on-Write Fork	4
III: Part C: Preemptive Multitasking and Inter-Process communication (IPC).....	5
内容二：任务完成情况	6
I: 任务完成列表	6
II: 具体 Exercise 完成情况	6
1. (Part A) Multiprocessor Support	6
1.1 Application Processor Bootstrap	7
Exercise 1	7
Question 1	8
1.2 Per-CPU State and Initialization.....	9
Exercise 2	9
Exercise 3	9
1.3 Locking.....	11
Exercise 4	11
Question 2	12
2. (Part A) Round-Robin Scheduling	12
Exercise 5	12
Question 3	14
3. (Part A) System Calls for Environment Creation	18
Exercise 6	18
3.1 sys_exofork	19
3.2 sys_env_set_status.....	19
3.3 sys_page_alloc.....	20
3.4 sys_page_map	21
3.5 sys_page_unmap	21
4. (Part B) Copy-on-Write Fork	23
4.1 User-level page fault handling.....	23
4.1.1 Setting the Page Fault Handler	23

Exercise 7	23
4.1.2 Normal and Exception Stacks in User Environments.....	24
4.1.3 Invoking the User Page Fault Handler	25
Exercise 8.....	25
4.1.4 User-mode Page Fault Entrypoint	27
Exercise 9.....	27
Exercise 10.....	29
4.1.5 Testing	30
4.2 Implementing Copy-on-Write Fork.....	32
Exercise 11.....	32
5. (Part C) Clock Interrupts and Preemption	35
5.1 Interrupt discipline	36
Exercise 12.....	36
5.2 Handling Clock Interrupts.....	38
Exercise 13.....	38
6. (Part C) Inter-Process communication (IPC)	38
6.1 IPC in JOS	38
6.2 Sending and Receiving Messages	39
6.3 Transferring Pages	39
6.4 Implementing IPC.....	39
Exercise 14.....	39
内容三：遇到的困难以及解决方法	44
困难一：Debug，还是 Debug	44
困难二：有些函数太麻烦	45
困难三：这个 Lab 拖得太久了	45
内容四：收获和感想	46
内容五：对课程的意见和建议	47
内容六：参考文献	48

内容一：总体概述

JOS Lab4 实习的主要内容为：在 Lab3 进程和中断管理的基础上，实现系统的**多进程管理**和**进程间消息通信**的功能。Lab4 分为三个部分：

part A: 我们将实现 JOS 对多 CPU 的支持，实现 round-robin 轮转式调度，实现一些用于进程管理的系统调用（创建和销毁进程，分配和映射内存）。

part B: 我们将实现类似 Unix 的 `fork()` 函数，这样用户进程就能够在用户的特权级别上创建一个“和自己一模一样”的子进程。

part C: 我们将实现多进程之间对时钟中断、抢占式调度的支持。我们还将实现对进程间通信（IPC）的支持，这样不同的用户进程之间就能够初步实现交流和同步的功能。

I: Part A: Multiprocessor Support and Cooperative Multitasking

这一部分，我们首先将实现 JOS 对多 CPU 的支持，然后实现 round-robin 轮转式调度，最后实现一些用于进程管理的系统调用（创建和销毁进程，分配和映射内存）。

JOS 对多 CPU 的支持指的是“对称多处理器(symmetric multiprocessing, SMP)”模型：每一个 CPU 对系统资源（如内存，IO 等）具有平等的访问能力。系统首先由一个“启动引导处理器(bootstrap processor, BSP)”启动和初始化，然后 BSP 负责激活其他“应用处理器(application processors, APs, 或者 non-boot processors)”。在此之后，每个 CPU 都可以访问内核代码，因此可以通过加上“大内核锁”来处理 CPU 竞争访问资源的情形。

round-robin 轮转式调度的基本思想，是 CPU 遍历 `envs` 数组，找到下一个可以运行的进程并运行之。在这种调度策略下，用户进程之间没有优先级。在 Part A 我们将实现“自愿的、非抢占的”轮转式调度，在 Part B 我们将实现抢占式轮转调度。

既然 JOS 支持多进程，我们那么就可以通过一些系统调用，让 kernel 为用户进程提供服务，例如，使得用户进程可以创建、运行子进程等。为了实现类似 Unix 中的 `fork()` 的功能，我们在这一部分首先要实现一些必要的系统调用，包括创建和销毁进程，分配和映射内存等。应该注意的是，这些功能将在用户空间中被调用，我们需要实现它们的包装。

II: Part B: Copy-on-Write Fork

我们将实现类似 Unix 的 `fork()` 函数，这样用户进程就能够在用户的特权级别上创建一个“和自己一模一样”的子进程。

这需要“写时复制(copy on write, COW)”技术，在 `fork()` 时让儿子和父亲共用地址映射，也即在各自的地址空间中映射到相同的物理地址，这种共享直到其中的任何一方要修改该地

址中的内容。在这样的机制下，`fork()`只将父进程的“地址空间映射”复制到子进程的地址空间，而不是真正复制所映射的内容。当两个进程试图写入共享的页面时，CPU 要识别出来，并产生一个 `page fault`。

在 JOS 中，处理 `page fault` 是用户级别的处理函数。因此，用户程序需要自己设置自己的 `page fault` 处理函数的入口。既然是用户级别的处理函数，我们引入了一个“用户异常栈”的概念，这样错误处理函数是在用户异常栈上运行，而不是内核栈或用户运行栈。为此，我们还需要设计一个精巧的流程，让进程能够正确地在这三个栈之间完成切换。

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

我们将实现多进程之间对时钟中断、抢占式调度的支持。我们还将实现对进程间通信（IPC）的支持，这样不同的用户进程之间就能够初步实现交流和同步的功能。

时钟中断是 CPU 外部的中断，除了 IDT 和中断向量号有所区别外，和我们之前处理的中断没什么区别。在时钟中断的基础上我们可以实现抢占式 `round-robin` 调度，也就是说用户程序在 CPU 上运行一段时间以后，CPU 将收到时钟中断，然后转入相应的进程调度程序，调度下一个可以运行的用户进程。

进程通信的机制主要是通过系统调用实现的。用户调用库函数，库函数陷入内核，然后再在内核中完成相应的操作。这里面有一个“接收进程被阻塞”的情形，不过既然在内核中，我们很容易通过修改 `Env` 结构中相应字段，来表示进程的状态，以及完成进程通信的功能。

内容二：任务完成情况

I: 任务完成列表

Exercise		1~6				5~11				12~14			
第一周													
第二周		Y											
第三周						Y				Y			

Challenge	1	2	3	4	5	6	7	8	9	10	11
第三周	Y										

II: 具体 Exercise 完成情况

1. (Part A) Multiprocessor Support

在这个部分，我们要让 JOS 支持多 CPU 的系统。这里，我们指的是“对称多处理器 (symmetric multiprocessing, SMP)”模型：每一个 CPU 对系统资源（如内存，IO 等）具有平等的访问能力。

SMP 模型的系统启动过程分为两步，由于每一个 CPU 都有一个唯一的标识符可以识别，因此第一步，是由一个称为“启动引导处理器(bootstrap processor, BSP)”的 CPU 对系统进行初始化，这部分工作和 Lab1 中单 CPU 的情形完全一样；第二步，当操作系统启动完毕，BSP 激活其他的 CPU，称之为“应用处理器(application processors, APs, 或者 non-boot processors)”。应该来说，除了在系统启动的过程中 BSP 和 APs 有区别以外，在 Lab4 的其余部分，它们没什么区别：都是用来执行指令的！而究竟在启动时谁是 BSP，这是由硬件和 BIOS 决定的，系统启动的代码由 BSP 执行。

在 SMP 系统中，每个 CPU 有一个 local APIC(LAPIC)单元。LAPIC 单元的作用，是在系统传递中断信号，同时可以通过 LAPIC 获得该 CPU 的标识符。在 Lab4 中，我对 LAPIC 的理解是，它是每个 CPU 对系统、其他 CPU 的接口。做完这个 Lab 对 LAPIC 的理解不需太多，只是下面这几个地方会涉及 LAPIC（在 kern/lapic.c 中）：

- 读取 LAPIC 标识符(APIC ID)，了解运行当前代码的 CPU 是哪一个（cpunum()函数）；
- 系统启动后，BSP 发送一个“CPU 间中断信号”STARTUP 给每一个 AP 用于激活（lapic_startap()函数）；
- 在抢占式调度中，通过对 LAPIC 内部的计时器进行编程，提供时钟中断（apic_init()函

数)。

一个 CPU 访问它的 LAPIC 时, 采用“内存映射 IO”技术(memory-mapped I/O, MMIO)。在 MIMO 中, 一部分物理内存硬连接到 IO 设备的寄存器上, 这样一来我们可以像访问内存一样访问 IO 设备, 如使用 load/store 指令。我们在向屏幕打印信息时, 使用了物理内存起始地址为 0xA0000 的“IO 空洞”, 这就是 MIMO 技术, 它将这部分物理地址连接至 CGA display 的缓冲区。至于 LAPIC, 它是在物理内存的 0xFE000000“LAPIC 洞”处, 此处离 4GB 仅有 32MB 之遥。还记得我们在 Lab2 时将[KERNBASE, 4GB)的线性地址空间(仅 256MB, 称为 kernel address space), 映射到整个物理内存, 以提供 kernel 对整个物理内存的管理, 所谓“直接映射”。而在这里, LAPIC 的物理地址太高了, KERNBASE 以上的线性地址空间根本不足以映射到那里。所以 JOS 做了一个小小的调整, 它挪用 kernel address space 的高 32MB 来映射“LAPIC 洞”, 那么 kernel address space 就只剩下 $256-32=224$ MB 了。这些内存初始化的工作, 在 kern/pmap.c 的 mem_init_mp()函数中已经为我们写好。我在这里详述的这些内容, 对完成 Lab4 应该来说不会有什么帮助, 不过有利于再次深入理解 Lab2 中的线性地址空间布局的概念。

1.1 Application Processor Bootstrap

这个部分的任务, 是让 APs 跑起来。而在此之前, BSP 会事先从 BIOS 搜集一些系统多处理器的信息, 例如有多少个 CPU, 它们的 APIC ID, 访问 LAPIC 单元的 MIMO 地址。参见 kern/mpconfig.c 中的 mp_init()函数。

当 BSP 搜集完这些信息以后, 它通过 kern/init.c 中的 boot_aps()函数将 APs 激活并运行起来。和 Lab1 中的 boot loader(boot/boot.S)一样, APs 也是从实模式开始运行。因此, boot_aps()函数要将 APs 的入口代码(kern/mpentry.S)复制到内存中实模式可以寻址的位置, 这里是 0x7000(MPENTRY_PADDR)。

在此之后, boot_aps()函数一个接一个地将 AP 激活: 它向 AP 的 LAPIC 发送 STARTUP 信号, 以及起始的 CS:IP 地址 MPENTRY_PADDR。AP 的起始代码在 kern/mpentry.S 中, 这和我们在 boot/boot.S 中看到的代码非常相似。在完成一些初始化工作以后, AP 开启分页并进入保护模式, 接着调用 mp_main()函数(kern/init.c)。这时, BSP 的 boot_aps()函数等待来自被激活 AP 的 CPU_STARTED 信号, 然后才激活下一个 AP。

Exercise 1. Read boot_aps() and mp_main() in kern/init.c, and the assembly code in kern/mpentry.S. Make sure you understand the control flow transfer during the bootstrap of APs. Then modify your implementation of page_init() in kern/pmap.c to avoid adding the page at MPENTRY_PADDR to the free list, so that we can safely copy and run AP bootstrap code at that physical address. Your code should pass the updated check_page_free_list() test (but might fail the updated check_kern_pkdir() test, which we will fix soon).

这个 exercise 是阅读 kern/init.c 中的 boot_aps()和 mp_main()函数, 以及 kern/mpentry.S 中的汇编代码。接着在 kern/pmap.c 的 page_init()添加一些代码, 使得我们即将用来保存 APs 起始代码的物理内存区域(MPENTRY_PADDR)不被加入空闲页框链表, 以免这片内存区被错误地分配出去。

首先来看 kern/init.c 中的 boot_aps()函数, 这个函数是被 BSP 调用, 用于一个一个地激活 APs。代码执行功能的流程简要了解即可, 大概就是 BSP 依次向每个 AP 发送 STARTUP 信号并等待 CPU_STARTED。

AP 被激活后首先执行的是 kern/mpentry.S 中的汇编代码，这些代码和 kern/entry.S 中的代码几乎完全一样，同时也都是在实模式。这里有一个问题，如下：

```
# Call mp_main(). (Exercise for the reader: why the indirect call?)
movl    $mp_main, %eax
call    *%eax
```

这是和 kern/entry.S 中不一样的，在那里是直接 call i386_init，而这里需要先将 mp_main 的地址放到 %eax 寄存器中。这是为什么呢？问助教，他们也不是很清楚。因此这个问题暂时搁置吧，不过确实，如果直接 call mp_main 的话是会出错的，也不知道为什么 T.T.

接着是 mp_main() 函数，这个函数是 AP 执行完汇编代码进入的第一个 C 函数，有点类似于之前对于单 CPU 的 i386_init() 函数。在这里面，AP 初始化它的内存、IDT 等等，然后通过 CPU_STARTED 通知 BSP 它已经启动好了。接着 mp_main() 开始进入所有 CPU 共享的、唯一的 kernel 代码区域，稍后我们将看到，为了保证这一区域任何时刻至多只有一个 CPU，它将会被加上一把“大内核锁”。

CPU 进入 kernel 之后做什么（此时不再需要区分 BSP 和 APs）？CPU 的苦逼本质在此体现。kernel 就好像一块唯一的“任务记录板”，上面记录着当前有哪些进程 Env 需要运行，以及他们在内存中的相关信息。任一 CPU 只要一得闲，他就进入 kernel“领取”任务，挑一个可以运行的 Env 去执行。这就是我们即将要实现的 sched_yield() 函数(exercise 4)，这个函数是从内核态切换到用户进程执行并实现调度。

别忘了在 kern/pmap.c 的 page_init() 添加一些代码，添加后如下：

```
void
page_init(void) {
    // LAB 4:
    // Change your code to mark the physical page at MPENTRY_PADDR
    // as in use
    size_t i;
    size_t i_IOPHYSMEM_begin = ROUNDDOWN(IOPHYSMEM, PGSIZE) / PGSIZE;
    size_t i_EXTPHYSMEM_end = PADDR(boot_alloc(0)) / PGSIZE;
    size_t i_MPENTRY_PADDR = MPENTRY_PADDR / PGSIZE;
    page_free_list = NULL;
    for (i = 0; i < npages; i++) {
        pages[i].pp_ref = 0;
        if(i == 0) continue;
        if(i_IOPHYSMEM_begin <= i && i < i_EXTPHYSMEM_end) continue;
        if(i == i_MPENTRY_PADDR) continue;
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```

Question 1. Compare kern/mpentry.S side by side with boot/boot.S. Bearing in mind that kern/mpentry.S is compiled and linked to run above KERNBASE just like everything else in the kernel, what is the purpose of macro MPBOOTPHYS? Why is it necessary in kern/mpentry.S but not in boot/boot.S? In other words, what could go wrong if it were omitted in kern/mpentry.S?

Hint: recall the differences between the link address and the load address that we have discussed

in Lab 1.

在 kern/mpentry.S 中我们看到了一个 MPBOOTPHYS 宏，这也是和 boot/boot.S 不同之处：

```
#define MPBOOTPHYS(s) ((s) - mpendry_start + MPENTRY_PADDR)
```

这个宏是将一个内核线性地址 *s*，转化成物理地址。为什么在这里需要这样做，而在 boot/boot.S 中并无此代码？这是因为正如任何 kernel 的代码，他们都是链接在 KERNBASE 以上的线性地址上（“链接地址”），也就是说我们从编译链接后的代码中取出的地址，都是在 KERNBASE 以上。而这在实模式下如何能够执行？还记得在 Lab1 中我们有一张“简单页表”，它进行下面的映射：

```
[KERNBASE, KERNBASE+4MB) ==> [0, 4MB)
[0, 4MB) ==> [0, 4MB)
```

这样，KERNBASE 以上的线性地址就得以映射成物理地址。现在对于 APs，我们没有采用“简单页表”的机制，而是采用“直接计算”的方式，计算方法就如 MPBOOTPHYS 宏所示。它将当前线性地址 *s* 减去起始地址 *mpentry_start*，然后加上 AP 起始代码的物理地址 MPENTRY_PADDR，这样就能够获得 *s* 对应的物理地址。

1.2 Per-CPU State and Initialization

Exercise 2. Modify `mem_init_mp()` (in `kern/pmap.c`) to map per-CPU stacks starting at `KSTACKTOP`, as shown in our revised `inc/memlayout.h`. The size of each stack is `KSTKSIZE` bytes plus `KSTKGAP` bytes of unmapped guard pages. Your code should pass the new check in `check_kern_pgdir()`.

修改 `kern/pmap.c` 中的 `mem_init_mp()` 函数，为每一个 CPU 映射栈空间。每个栈的空间开销为 `KSTKSIZE` 字节的栈大小，和 `KSTKGAP` 字节的保护页大小。映射的方式为：对于 CPU *i*，将 `percpu_kstacks[i]`（线性地址！）指向的起始物理地址，当作它的栈空间。那么 CPU *i* 的堆栈将从线性地址 `kstacktop_i = KSTACKTOP - i * (KSTKSIZE + KSTKGAP)` 向下生长。总的来说，对于 CPU *i*：

- 栈的线性地址范围：[`kstacktop_i - KSTKSIZE`, `kstacktop_i`)
 - 保护页的线性地址范围：[`kstacktop_i - (KSTKSIZE + KSTKGAP)`, `kstacktop_i - KSTKSIZE`)
- 代码如下：

```
int kstacktop_i;
int i;
for(i = 0; i < NCPU; i++)
{
    kstacktop_i = KSTACKTOP - i * (KSTKSIZE + KSTKGAP);
    boot_map_region(kern_pgdir, kstacktop_i-KSTKSIZE, KSTKSIZE,
PADDR(percpu_kstacks[i]), PTE_W | PTE_P);
}
```

Exercise 3. The code in `trap_init_percpu()` (`kern/trap.c`) initializes the TSS and TSS descriptor for the BSP. It worked in Lab 3, but is incorrect when running on other CPUs. Change the code so that it can work on all CPUs. (Note: your new code should not use the global `ts` variable any

more.)

修改 kern/trap.c 中的 trap_init_percpu() 函数，保证它不仅仅对 BSP 有效，对 APs 也有效。我们发现，在 trap_init_percpu() 函数中，它对 TSS 进行设置，这是 Lab3 的内容。我们对 TSS 的使用仅限于在发生中断/异常时，CPU 从用户运行栈切换到 TSS 指定的 kernel 栈。

既然现在有多 CPU，每个 CPU 的栈地址也不一样（见 exercise 2），因此这便是我们需要修改的地方：在设置 TSS 时要设置成当前 CPU 的 kernel 栈。明确这一点后，我们还要了解如何加载 TSS，在此不作赘述。代码如下：

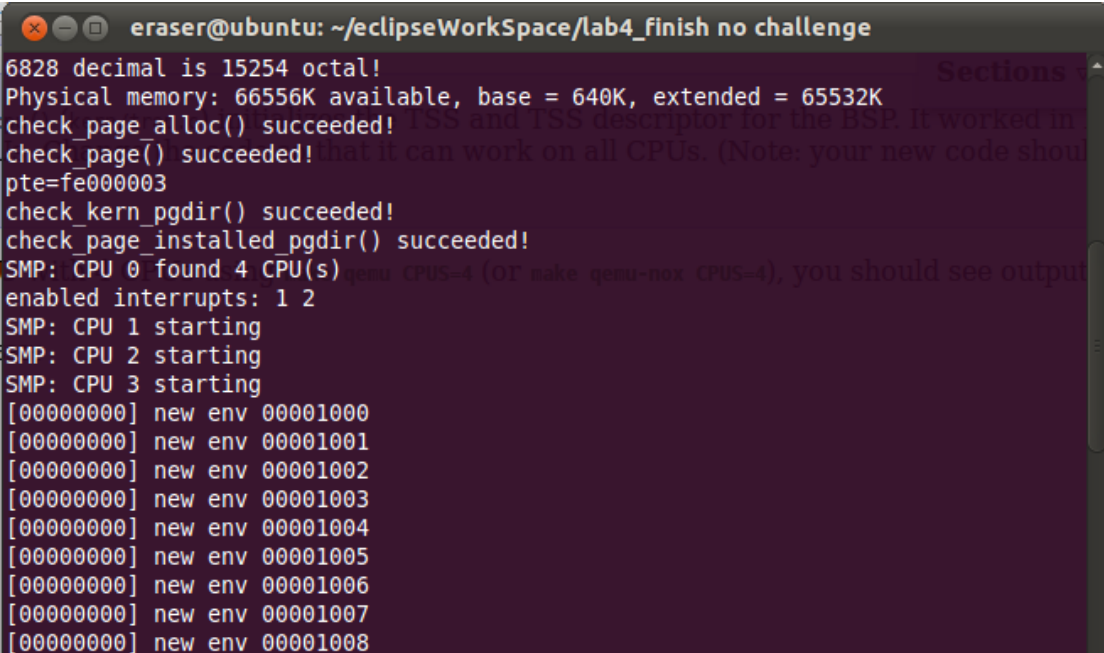
```
// Setup a TSS so that we get the right stack
// when we trap to the kernel.
int cur_cpu_i = thiscpu->cpu_id;
thiscpu->cpu_ts.ts_esp0 = KSTACKTOP - cur_cpu_i * (KSTKSIZE +
KSTKGAP);
thiscpu->cpu_ts.ts_ss0 = GD_KD;

// Initialize the TSS slot of the gdt.
gdt[(GD_TSS0 >> 3) + cur_cpu_i] = SEG16(STS_T32A, (uint32_t)
&(thiscpu->cpu_ts),
    sizeof(struct Taskstate), 0);
gdt[(GD_TSS0 >> 3) + cur_cpu_i].sd_s = 0;

// Load the TSS selector (like other segment selectors, the
// bottom three bits are special; we leave them 0)
ltr((GD_TSS0 + (cur_cpu_i << 3)) & ~0x7);

// Load the IDT
lidt(&idt_pd);
```

在完成上面的 exercise 之后，我们用 QEMU 模拟 4 个 CPU，在上面运行 JOS (make qemu CPUS=4)，输出效果如下：



```
eraser@ubuntu: ~/eclipseWorkspace/lab4_finish no challenge
6828 decimal is 15254 octal!
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
check_page() succeeded!
pte=fe000003
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 4 CPU(s)
enabled interrupts: 1 2
SMP: CPU 1 starting
SMP: CPU 2 starting
SMP: CPU 3 starting
[00000000] new env 00001000
[00000000] new env 00001001
[00000000] new env 00001002
[00000000] new env 00001003
[00000000] new env 00001004
[00000000] new env 00001005
[00000000] new env 00001006
[00000000] new env 00001007
[00000000] new env 00001008
```

1.3 Locking

我们注意到，在 `mp_main()` 的最后一行有一个自旋锁，这时 AP 就被卡在这里了。在让 AP 真正进入 kernel 前，我们要保证 kernel 的代码区域任何时刻至多只有一个 CPU 在执行——我们需要防止多个 CPU 之间出现 race condition 导致出错——因此最简单地可以上一把“大内核锁”。“大内核锁”是一个全局的锁，在任何用户进程要进入 kernel 时需要“申请锁”，在用户进程返回用户空间时需要“释放锁”。这样一来，用户空间的用户进程将可以在多个 CPU 上并行执行；然而不会有超过一个 CPU 同时在 kernel mode 下运行，否则后来的 CPU 需要等待前面的 CPU 释放锁。

`kern/spinlock.h` 中定义了大内核锁。并提供了 `lock_kernel()` 和 `unlock_kernel()` 操作。MIT 课程资料太人性化了，如此直白地告诉我们需要在下面这 4 处地方申请/释放大内核锁：

- 在 `i386_init()` 中，BSP 在激活其他 CPU 之前，申请关闭内核锁。
 - 在 `mp_main()` 中（注意到 `mp_main()` 和上面的 `i386_init()` 功能类似），初始化 AP 之后申请内核锁，然后调用 `sched_yield()` 函数开始苦逼的执行任务。
 - 在 `trap()` 中，在从 user mode 陷入 kernel 时申请内核锁，需要判断“陷入”是从 user mode 还是 kernel mode，因此可以利用 `tf_cs` 是否等于 `GD_KD`。
 - 在 `env_run()` 中，在切换回 user mode 前释放内核锁。注意释放的时机，以防死锁。
- 归纳起来，上面这 4 处地方恰好都是 user mode 和 kernel mode 之间进行切换的分界点。

Exercise 4. Apply the big kernel lock as described above, by calling `lock_kernel()` and `unlock_kernel()` at the proper locations.

按照提示操作即可。代码如下：

```
i386_init():
    // Acquire the big kernel lock before waking up APs
    // Your code here:
    lock_kernel();

mp_main():
    // Your code here:
    lock_kernel();
    sched_yield();

trap():
    // LAB 4: Your code here.
    lock_kernel();

env_run():
    unlock_kernel(); // 在env_pop_tf之前
    env_pop_tf(&curenv->env_tf);
```

完成 exercise 4 之后，记得按照提示把 `mp_main()` 的最后一行的自旋锁去掉。稍后我们实现 `sched_yield()` 调度函数之后，就可以检验我们的锁是否使用正确。这里一般不会出错，不过我还是悲剧了。我不小心在 `trap()` 中写成了 `sched_yield()`，唉，结果后来调了半天才发现。天哪，上天请赐予我细心吧。

Question 2. It seems that using the big kernel lock guarantees that only one CPU can run the kernel code at a time. Why do we still need separate kernel stacks for each CPU? Describe a scenario in which using a shared kernel stack will go wrong, even with the protection of the big kernel lock.

大内核锁保证似乎任何时刻只有一个 CPU 工作在 kernel 代码区，为什么我们仍需要对不同的 CPU 采用不同的 kernel 栈呢？能否只使用一个？

答案是，不可以。我们考虑一些边界的情况。比如说发生中断的时候，我们是在 `trap()` 函数里面才加上大内核锁。但是实际上我们什么时候就在 kernel 栈上运行？由 TSS 指定，在中断/异常发生以后，CPU 就已经自动切换到内核栈了。然后我们就会将保存现场的信息压到栈里面。这时候如果共用 kernel 栈，那么可能压栈保存现场会出现错误。也就是说这个时候，多个 CPU 对 kernel 栈的使用出现竞争，所以有可能出错。

2. (Part A) Round-Robin Scheduling

接下来我们将实现 JOS 对用户进程的轮转式调度。所谓轮转，就是 `take turns` 或者叫 `round-robin`，保证对每个人都公平。

我们首先实现“每个进程‘自觉地’让出 CPU”这个功能，也就是说，在用户程序里面，由它自己来通知操作系统，“我刚刚执行了一下哦，现在很满足了，你可以调度下一个用户进程了”。这确实弱爆了，如果让我写一个用户程序，我肯定不会这么傻地去调用这个“对自己没什么好处的”系统调用 `sys_yield()`。别担心，在 `part C` 我们将实现时钟中断的功能，届时不管一个用户进程愿不愿意下 CPU，它都得下——而不是通过这里自觉地调用一个调度的系统调用。

Exercise 5. 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 three (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 00001008.
Hello, I am environment 00001009.
Hello, I am environment 0000100a.
Back in environment 00001008, iteration 0.
Back in environment 00001009, iteration 0.
Back in environment 0000100a, iteration 0.
Back in environment 00001008, iteration 1.
Back in environment 00001009, iteration 1.
Back in environment 0000100a, iteration 1.
...
```

After the yield programs exit, when only idle environments are runnable, the scheduler should invoke the JOS kernel monitor. If any of this does not happen, then fix your code before proceeding.

首先实现 kern/sched.c 中的 sched_yield() 函数。Round-robin 轮转调度算法在 JOS 中的实现大致如下：

- 在 envs 数组中，前 NCPU 个进程 envs[0]~ envs[NCPU-1] 将被专门用于 idle 进程（见 user/idle.c），每个 idle 进程对应一个 CPU。这些进程的功能，就是简单地“浪费掉”CPU 的时间，此时 CPU 没什么其他可以处理的任务。在 idle 进程中，它永远通过一个调度的系统调用，尝试着让 CPU 调度其他进程。这明明就是一个利他主义者嘛。在 kern/init.c 中已经为我们初始化好了这 NCPU 个 idle 进程，我们直接 env_run 他们就可以了。

- kern/sched.c 中的 sched_yield() 函数的功能，是选择一个新的可以运行的进程，也就是调度。sched_yield() 函数可以比喻作一个指挥者。它按环状、顺序遍历 envs 数组，从刚刚下 CPU 的那个用户进程开始，选择接下来第一个 ENV_RUNNABLE 的用户进程，然后对它调用 env_run() 来切换到它。sched_yield() 永远不选择 idle 进程，尽管它们确实是 ENV_RUNNABLE 的。我们会通过后面的代码来处理 idle 进程。

- 如果没有其他进程 ENV_RUNNABLE，那么但当前进程仍可运行，那么就运行之。

- sched_yield() 永远不能让同一个用户进程，同时在多个不同的 CPU 上运行。怎么判断一个进程是否已经运行在一个 CPU 上了？看看它是不是 ENV_RUNNING 即可。

- 这就是我们刚才提到的系统调用：sys_yield()。通过这个系统调用，用户进程可以调用 kernel 的 sched_yield() 函数，然后自觉地让出 CPU。

- 每当 kernel 在调度时切换进程，要保证下 CPU 的进程已经好保存现场。进程的现场在哪里保存？在 Env 结构体中的那些寄存器变量中，这是在发生系统调用时，就保存好的。在 kern/trap.c 的 trap() 函数中，我们看到下面的代码：

```
// Copy trap frame (which is currently on the stack)
// into 'curenv->env_tf', so that running the environment
// will restart at the trap point.
curenv->env_tf = *tf;
```

上面的 tf，其实已经是我们所谓“保存现场”的所有信息了。现在我们将它复制（注意是值复制，而不是指针复制）到 curenv->env_tf 中，这样用户进程就可以自己记住它刚才运行到哪里，而不依赖于 kernel 栈。

kern/sched.c 的 sched_yield() 函数代码如下：

```
void
sched_yield(void) {
    struct Env *idle;
    int i;
    if(thiscpu->cpu_env != NULL) {    //小心，巨坑
        int cur_env_id = thiscpu->cpu_env->env_id;
        cur_env_id = ENVX(cur_env_id);

        for(i = (cur_env_id+1)%NENV; i != cur_env_id; i = (i+1)%NENV) {
            if(envs[i].env_type != ENV_TYPE_IDLE &&
                envs[i].env_status == ENV_RUNNABLE)
                break;
        }
    }
}
```

```

    }
    if(i != cur_env_id)
        env_run(&envs[i]);
    if(thiscpu->cpu_env->env_status == ENV_RUNNING)
        env_run(thiscpu->cpu_env);
}
// 以下代码切换到monitor或idle进程
...
}

```

注意，`sched_yield()`中需要判断 `thiscpu->cpu_env` 是不是 `NULL`，这很重要。一开始系统刚刚运行起来的时候，`thiscpu->cpu_env` 的确是 `NULL`，看下面 `kern/env.c` 中 `env_run()` 函数顶上的注释：

Note: if this is the first call to `env_run`, `curenv` is `NULL`.

那它什么时候开始不是 `NULL`？我们就来看看从开机到运行第一个 `Env` 的流程。开完机，初始化完毕后，我们看到 `i386_init()` 里面 `ENV_CREATE` 了一些用户进程，于是 `envs` 数组里就出现了一些 `ENV_RUNNABLE` 的进程。接着 `i386_init()` 或 `mp_main()` 调用 `sched_yield()` 函数，里面先判断 `thiscpu->cpu_env` 为 `NULL`，于是执行接下来的代码。因为现在还有 `ENV_RUNNABLE` 的进程，所以不会切换到 `monitor`，于是切换到 `idle` 进程。这样，`thiscpu->cpu_env` 的第一次赋值将指向 `idle` 进程，之后它就不为 `NULL` 了。

为此我又调了差不多两天，其过程艰辛不再赘述。当然，好处就是我对这整个过程理解的更加透彻了。**指针啊，判断是否为空，很重要。**

在 `kern/syscall.c` 的 `syscall()`（小心不是 `kern/trap.c` 的 `trap_dispatch()`）中分派至 `sys_yield()` 系统调用的代码：

```

case SYS_yield:
    sys_yield();
    break;

```

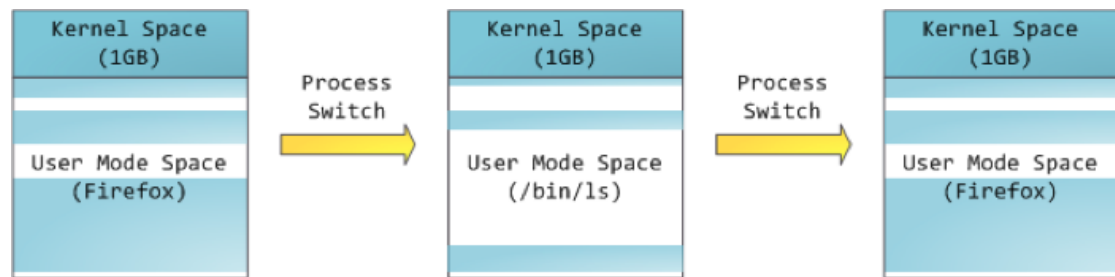
Question 3. In your implementation of `env_run()` you should have called `lcr3()`. Before and after the call to `lcr3()`, your code makes references (at least it should) to the variable `e`, the argument to `env_run`. Upon loading the `%cr3` register, the addressing context used by the MMU is instantly changed. But a virtual address (namely `e`) has meaning relative to a given address context--the address context specifies the physical address to which the virtual address maps. Why can the pointer `e` be dereferenced both before and after the addressing switch?

这个问题问得很用心。在 `env_run()` 函数中，我们切换了页目录页表；而 `e` 是一个指针，我们在 `env_run()` 切换页表前后都要使用的 `e` 的值，因此应该保证含义是不变的，即最终都得指向同一个物理地址。切换页目录页表，会不会导致 `e` 指向别的地址，造成麻烦？

答案是不会。这里需要我们深刻理解“线性地址空间布局”中 `Kernel Address Space` 的概念（见 Lab 2）。对于不同的用户进程，其 4GB 的线性地址空间中 `Kernel Address Space` 是一样的、唯一的、保持不变的，都和 `boot pgdir` 一样映射到同一片物理内存区域；只有 `User Address Space` 因进程而异。

因此这题的理由是：指针 `e` 指向的是 `Kernel Address Space` 中的 `envs` 数组的一个表项（换言之 `envs` 数组在 `kernel` 中），因此，在切换页目录/页表前后，`e` 指向的当然还是同一个地址，不会出错。

为了帮助理解“页目录/页表切换前后，Kernel Address Space 保持不变，User Address Space 因进程而异”这个重要概念，分享一张示意图：



蓝色表示被映射的线性地址，白色表示地址尚未映射。这张图就表示了在 Linux 中（JOS 也一样），进程切换时，用户进程的 4GB 线性地址空间的变化情况。其中 Kernel Space 是不变的，变的只是 User Space 所映射的内容。

Challenge. Add a less trivial scheduling policy to the kernel, such as a fixed-priority scheduler that allows each environment to be assigned a priority and ensures that higher-priority environments are always chosen in preference to lower-priority environments. If you're feeling really adventurous, try implementing a Unix-style adjustable-priority scheduler or even a lottery or stride scheduler. (Look up "lottery scheduling" and "stride scheduling" in Google.)

Write a test program or two that verifies that your scheduling algorithm is working correctly (i.e., the right environments get run in the right order). It may be easier to write these test programs once you have implemented `fork()` and IPC in parts B and C of this lab.

这个 challenge 是让我们实现 `sched_yield()` 的优先级调度算法：不是简单的轮转调度，而是对每个用户进程都赋予一个“优先级”，然后每次调度从最高优先级的进程开始。

首先，在 `inc/env.h` 中对结构体 `struct Env` 增加一个记录用户进程优先级的成员变量：

```
uint32_t env_priority;
```

同时，为了使用方便，我们在 `inc/env.h` 中定义了一些宏，表示不同的优先级：

```
#define PRIORITY_HIGH 0x10000
#define PRIORITY_DEFAULT 0x1000
#define PRIORITY_MIDDLE 0x100
#define PRIORITY_LOW 0x10
```

接下来，在 `kern/env.c` 的中的 `env_alloc()` 函数，也就是创建进程、初始化进程的 `Env` 结构体的时候，设置进程优先级为 `default`：

```
e->env_priority = PRIORITY_DEFAULT;
```

好的，至此每个进程在创建时都具有了 `default` 的优先级。那么我们怎么修改这个优先级呢？模仿 `sys_env_set_status()` 这个系统调用，我们可以让用户进程通过系统调用，在它的代码中设置自己的优先级，因此我们要实现一个 `sys_env_set_priority()` 的系统调用。

要设置新的系统调用，所涉及的文件比较多：

在 `inc/lib.h` 中添加函数声明：`int sys_env_set_priority(envid_t envid, int priority);`

在 `kern/syscall.c` 中添加函数体，代码如下（完全模仿 `sys_env_set_status()` 函数）：

```
static int
sys_env_set_priority(envid_t envid, uint32_t priority)
{
    struct Env * env;
    int r = envid2env(envid, &env, 1);
    if (r < 0) return r;
    env->env_priority = priority;
    return 0;
}
```

在 inc/syscall.h 中添加新的中断号：SYS_env_set_priority。

在 kern/syscall.c 的 syscall() 函数中添加相应 switch 分发语句。

在 lib/syscall.c 中添加供用户调用系统调用的库函数，代码如下：

```
int
sys_env_set_priority(envid_t envid, int priority)
{
    return syscall(SYS_env_set_priority, 1, envid, priority, 0, 0, 0);
}
```

最后当然要实现 kern/sched.c 中的代码了。这回，我们不是“遇到下一个可以运行的用户进程就运行”，而是遍历所有用户进程，找出优先级最高的，再运行。代码如下：

```
// Choose a user environment to run and run it.
void
sched_yield(void)
{
    // 略去前面的代码，只需要修改中间这部分
    if(thiscpu->cpu_env != NULL) {
        int cur_env_id = thiscpu->cpu_env->env_id;
        cur_env_id = ENVX(cur_env_id);

        uint32_t max_priority = 0;
        int mp_envid = -1;
        for(i = (cur_env_id+1)%NENV; i != cur_env_id; i = (i+1)%NENV) {
            if(envs[i].env_type != ENV_TYPE_IDLE &&
                envs[i].env_status == ENV_RUNNABLE &&
                envs[i].env_priority > max_priority)
            {
                max_priority = envs[i].env_priority; // 选出优先级最高的
                mp_envid = i;
            }
        }
        i = mp_envid; // 偷懒，下面就不用修改了
        if(i != cur_env_id) {
            env_run(&envs[i]);
        }
        if(thiscpu->cpu_env->env_status == ENV_RUNNING) {
```



```
        env_run(thiscpu->cpu_env);
    }
}
// 略去后面的代码
}
```

最后需要编写几个不同优先级别的用户进程用于测试。为了简便，我就不 fork 了，而是简单地在 kern/init.c 中加载 4 个用户程序：

```
ENV_CREATE(user_testpriority_high, ENV_TYPE_USER);
ENV_CREATE(user_testpriority_default, ENV_TYPE_USER);
ENV_CREATE(user_testpriority_middle, ENV_TYPE_USER);
ENV_CREATE(user_testpriority_low, ENV_TYPE_USER);
```

这四个用户程序基本完全一样，就是设置优先级那里的代码不同，举例如下：

```
#include <inc/lib.h>
#include <inc/env.h>

void
umain(int argc, char **argv)
{
    sys_env_set_priority(0, PRIORITY_HIGH);
    int i;
    int n = 3;
    for (i = 0; i < n; i++) {
        cprintf("[%08x] High Priority Process is Running\n", sys_getenvid());
    }
    return;
}
```

当然，要运行新的用户程序，我们还要修改 kern/Makefrag 文件：

```
user/testpriority_high \
user/testpriority_default \
user/testpriority_middle \
user/testpriority_low
```

最后运行结果如下。观察发现，我们的调度程序的确是从优先级高到低调度用户程序的，成功！

```
eraser@ubuntu: ~/eclipseWorkSpace/lab4
[00000000] new env 00001007
[00000000] new env 00001008
[00000000] new env 00001009
[00000000] new env 0000100a
[00000000] new env 0000100b
[00001008] High Priority Process is Running
[00001008] High Priority Process is Running
[00001008] High Priority Process is Running
[00001008] exiting gracefully
[00001008] free env 00001008
[0000100a] Middle Priority Process is Running
[0000100a] Middle Priority Process is Running
[0000100a] Middle Priority Process is Running
[0000100a] exiting gracefully
[0000100a] free env 0000100a
[00001009] Default Priority Process is Running
[0000100b] Low Priority Process is Running
[0000100b] Low Priority Process is Running
[0000100b] Low Priority Process is Running
[00001009] Default Priority Process is Running
[00001009] Default Priority Process is Running
[00001009] exiting gracefully
[00001009] free env 00001009
[0000100b] exiting gracefully
[0000100b] free env 0000100b
No more runnable environments!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
eraser@ubuntu:~/eclipseWorkSpace/lab4$
```

3. (Part A) System Calls for Environment Creation

既然现在 JOS 提供了“这么强大”的多进程支持，我们要让用户进程运行得更加有意思一点。那么就可以通过一些系统调用，让 kernel 为用户进程提供服务。接下来我们将实现一些必要的系统调用，使得用户进程可以创建、运行子进程。

在 Unix 里面，这就是 `fork()` 系统调用：它将父进程的整个地址空间的内容复制一份给子进程，也即子进程就是父进程的副本。父子进程的、在 `user mode` 可以识别的唯一区别在于，在父进程中 `fork()` 的返回值为子进程的 ID，而在子进程中返回值为 0。默认情况下，每个进程的地址空间是独立的。

为了在 JOS 实现类似 `fork()` 的功能，我们首先要实现一些和“在 `user mode` 创建子进程”相关的一系列系统调用，这就是下面这个 exercise 的内容。

Exercise 6. Implement the system calls described above in `kern/syscall.c`. You will need to use various functions in `kern/pmap.c` and `kern/env.c`, 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_INVALID` in that case. Test your JOS kernel with `user/dumbfork` and make sure it works before proceeding.

下面我们将在 `kern/syscall.c` 中实现一些系统调用。其实这些系统调用，就是对 `kern/pmap.c` 和 `kern/env.c` 中的一些 kernel 级别的函数进行组织和包装，使得在用户进程中能够获得操作系统提供的相应服务。

3.1 sys_exofork

这个函数要做一些 `fork()` 的前期工作，主要是创建新的子进程的 `Env` 结构。此时这个子进程除了寄存器状态和父进程（在调用 `sys_exofork()` 时）完全一样，它的地址空间中还没有映射什么内容。在父进程中，`sys_exofork()` 返回值为子进程的 `envid_t` 或负的 `error code`；在子进程中 `sys_exofork()` 返回值为 0。应该注意的是，到此为止子进程时 `not runnable` 的，它还需要经过后续一系列的初始化，直到父进程将子进程标记为 `runnable` 时，子进程在能运行，即从 `sys_exofork()` 中返回。

`sys_exofork()` 的代码如下：

```
static envid_t
sys_exofork(void) {
    struct Env *childenv;
    int r;

    if ((r = env_alloc(&childenv, sys_getenvid())) < 0)
        return r;

    // set child not runnable
    childenv->env_status = ENV_NOT_RUNNABLE;

    // copy trapframe (registers)
    memmove((void *)&(childenv->env_tf), (const void *)&(curenv->env_tf),
sizeof(struct Trapframe));

    // make the child env's return value zero, put it in eax!
    childenv->env_tf.tf_regs.reg_eax = 0;
    return childenv->env_id;
}
```

这个函数很有意思，执行完这个函数，在 `envs` 数组中将会多出一个和当前进程几乎一模一样的用户进程来，但是它还不能够运行(`ENV_NOT_RUNNABLE`)。就好像一个人进了一个小房间，出来之后变成两个一模一样的人，这让我联想到了电影《致命魔术》（当然，在新生成的那个“人”现在只具备肉身，还没有灵魂，我们稍后要对它注入灵魂（地址空间的内容），并在返回值做一个小小的标记以示区别）。意味着新的进程一旦可以运行，它也是从 `sys_exofork()` 出来的。怎么控制返回值？我们知道 `sys_exofork()` 是个系统调用，因此显然返回值可以通过 `%eax` 寄存器来返回。

3.2 sys_env_set_status

这个系统调用用于设置指定进程的状态为 `ENV_RUNNABLE` 或 `ENV_NOT_RUNNABLE`。这个系统调用一般是在父进程将子进程的寄存器和地址空间初始化完毕后，需要将子进程标记为 `runnable` 时使用。

提示写得很清楚，大概知道 `envid2env()` 干什么的就行，代码如下：

```
static int
sys_env_set_status(envid_t envid, int status) {
    if (status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE)
```

```

        return -E_INVALID;

    struct Env *env;
    int r;

    if ((r = env2env (env, &env, 1)) < 0)
        return r;
    env->env_status = status;
    return 0;
}

```

3.3 sys_page_alloc

这个系统调用是为用户进程分配一个物理页框，并将它映射到指定的线性地址（映射都是以“页”为单位）。

从这个函数开始，我们逐渐发现麻烦之处在于，为了增强系统的鲁棒性，我们需要书写很多判断条件是否满足的分支语句。别着急，我的经验是，耐心地一项一项完成，完成一项打一个勾即可。

```

static int
sys_page_alloc(env_t env, void *va, int perm){
    if (va >= (void *)UTOP || ROUNDUP(va, PGSIZE) != va)
        return -E_INVALID;

    if ( !(perm & PTE_U) || !(perm & PTE_P)
        || ((perm & ~PTE_SYSCALL) != 0))
        return -E_INVALID;

    struct Env *env;
    int r;
    if ((r = env2env (env, &env, 1)) < 0)
        return -E_BAD_ENV;

    struct Page *pp;
    pp = page_alloc (ALLOC_ZERO);
    if (pp == NULL)
        return -E_NO_MEM;

    if ((r = page_insert (env->env_pgdir, pp, va, perm)) < 0){
        page_free(pp);
        return r;
    }

    return 0;
}

```

3.4 sys_page_map

这个系统调用是将一个地址映射(page mapping, not the contents of a page)从一个进程的地址空间复制到另一个进程的地址空间。这样，两个进程就共享同一个地址映射，即将同一个线性地址映射到相同的物理地址。

麻烦之处在于，需要一项一项地进行分支判断，返回 error code.

```
static int
sys_page_map(envid_t srcenvid, void *srcva,
             envid_t dstenvid, void *dstva, int perm){
    if (srcva >= (void *)UTOP || ROUNDUP(srcva, PGSIZE) != srcva
        || dstva >= (void *)UTOP || ROUNDUP(dstva, PGSIZE) != dstva)
        return -E_INVAL;

    if ( !(perm & PTE_U) || !(perm & PTE_P)
         || ((perm & ~PTE_SYSCALL) != 0))
        return -E_INVAL;

    struct Env *srcenv, *dstenv;
    int r;
    if ((r = envid2env (srcenvid, &srcenv, 1)) < 0)
        return -E_BAD_ENV;
    if ((r = envid2env (dstenvid, &dstenv, 1)) < 0)
        return -E_BAD_ENV;

    pte_t *pte;
    struct Page *pp;
    pp = page_lookup (srcenv->env_pgdir, srcva, &pte);
    if (pp == NULL || ((perm & PTE_W) != 0 && (*pte & PTE_W) == 0))
        return -E_INVAL;

    if ((r = page_insert (dstenv->env_pgdir, pp, dstva, perm)) < 0)
        return -E_NO_MEM;

    return 0;
}
```

3.5 sys_page_unmap

这个系统调用将给定用户进程中映射到某一给定线性地址的页面解除映射。

```
static int
sys_page_unmap(envid_t envid, void *va){
    if(va >= (void *)UTOP || ROUNDUP(va, PGSIZE) != va)
        return -E_INVAL;

    struct Env *env;
    int r;
```

```

    if ((r = env_id2env (env_id, &env, 1)) < 0)
        return -E_BAD_ENV;

    page_remove(env->env_pgdir, va);
    return 0;
}

```

别忘了，写完这几个系统调用后，记得在 `syscall()` 中进行 `dispatch`！还要小心，`dispatch` 的时候还要注意传递返回值。

```

case SYS_exofork:
    r = sys_exofork();    //小心返回值！调了三个小时！
    break;
case SYS_env_set_status:
    r = sys_env_set_status((env_id_t)a1, (int)a2);
    break;
case SYS_page_alloc:
    r = sys_page_alloc((env_id_t)a1, (void *)a2, (int)a3);
    break;
case SYS_page_map:
    r = sys_page_map((env_id_t)a1, (void *)a2, (env_id_t)a3, (void *)a4,
(int)a5);
    break;
case SYS_page_unmap:
    r = sys_page_unmap((env_id_t)a1, (void *)a2);
    break;

```

运行一下 `user/dumbfork`，结果如下：

```

[00001007] new env 00001007
[00001008] new env 00001008
[00001009] new env 00001009
0: I am the parent!
0: I am the child!
1: I am the parent!
1: I am the child!
2: I am the parent!
2: I am the child!
3: I am the parent!
3: I am the child!
4: I am the parent!
4: I am the child!
5: I am the parent!
5: I am the child!
6: I am the parent!
6: I am the child!
7: I am the parent!
7: I am the child!
8: I am the parent!
8: I am the child!
9: I am the parent!
9: I am the child!
[00001008] exiting gracefully
[00001008] free env 00001008
10: I am the child!
11: I am the child!
12: I am the child!
13: I am the child!
14: I am the child!
15: I am the child!
16: I am the child!
17: I am the child!
18: I am the child!
19: I am the child!
[00001009] exiting gracefully
[00001009] free env 00001009
No more runnable environments!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>

```

符合预期，说明写对了。part A 完成。

4. (Part B) Copy-on-Write Fork

Unix 提供 `fork()` 函数创建子进程，子进程是父进程的副本，拥有和父进程完全一样的线性地址空间。

这里有一个问题：线性地址空间相同是指“线性地址映射到相同的物理地址（因此内容相同）”，还是“线性地址映射到不同的、但保存着相同内容的物理地址”。事实上，这是两种不同的实现，开销也不一样。我们注意到，后者在物理内存中保存了两份相同的数据。

xv6 Unix 采用了第二种方式实现 `fork()` 调用，很类似我们的一个用户程序 `user/dumbfork()`。注意到将父进程物理页框中的内容，拷贝到子进程物理页框之中，在这种 `fork()` 的实现中占了很大的开销。

然而，我们注意到在调用 `fork()` 创建子进程之后，子进程一般并不使用父亲给他的内容，子进程将在其地址空间中载入新的程序和数据并运行。那么在这种情况下，儿子就浪费了父亲给他的几乎所有遗产，父亲当初何必给他这么多？

处于这个考虑，Unix 的后续版本让儿子和父亲共用地址映射，也即在各自的地址空间中映射到相同的物理地址，这种共享直到其中的任何一方要修改该地址中的内容。这种机制成为“写时复制(copy on write, COW)”。在这样的机制下，`fork()` 只将父进程的“地址空间映射”复制到子进程的地址空间，而不是真正复制所映射的内容。当两个进程试图写入共享的页面时，CPU 要识别出来，并产生一个 `page fault`。这样 Unix 的 Kernel 就能够知道任何写入，是对私有页面的写入，还是一个写时复制的页面，如果是后者要马上分配新的私有页框。因此，对新页框的分配和复制可以说是“拖”到了不得不做的时候，这种 *lazy* 的机制使得 `fork()` 的开销得到了最小化。

在接下来的 Lab 4 中，我们要实现一个类似 Unix 的写时复制的 `fork()` 库函数。之所以说是库函数，是因为它的确是在 `user mode` 运行，里面对一系列系统调用进行了包装。这样做可以让 kernel 更加轻量，也更安全，用户使用更方便。

4.1 User-level page fault handling

一个用户级别的写时复制 `fork()`，在写入共享页面时，会产生 `page fault`。我们待会儿要实现这个机制。注意到，`page fault` 有很多用处，写时复制只是这么多用处中的一种。

当一些动作执行时，可能会伴随着 `page fault`。例如，我们可以在一开始的时候只给程序分配一页内存作为栈区，随着程序对栈的使用的增加，可能会产生 `page fault`，我们才为程序分配并映射新的页面。这种“按需分配”的机制让程序觉得它的栈空间似乎想要多少就要多少。kernel 要能够知道，当程序地址空间中的不同区域产生 `page fault` 时，采取什么样的不同措施。例如，栈区的 `page fault` 将导致为栈再分配并映射一页物理页框；BSS 段发生 `page fault` 时将其分配并映射一页物理页框，并初始化为 0；有的程序甚至可以再代码段发生 `page fault`，这时就要从硬盘读入相应的可执行代码，映射，然后继续执行下去，好像刚才什么事情都没有发生过。

4.1.1 Setting the Page Fault Handler

Exercise 7. 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.

既然不同的 page fault 需要采取不同的处理机制，因此用户需要告诉 kernel 它想怎么处理。在 JOS 中可以这样实现，每个用户进程的 page fault 处理机制可以不一样，这是由用户进程在一开始就调用 `sys_env_set_pgfault_upcall()` 系统调用，在该用户进程 `Env` 结构的 `env_pgfault_upcall` 变量中进行注册，注册为它期待的处理函数的起始地址。代码如下：

```
static int
sys_env_set_pgfault_upcall(envid_t envid, void *func)
{
    struct Env *env;
    if (envid2env(envid, &env, 1) < 0)
        return -E_BAD_ENV;
    env->env_pgfault_upcall = func; //记录entry point
    return 0;
}
```

注意，这个机制等一下会有一些变化，有点“绕”，我们待会儿会介绍。

4.1.2 Normal and Exception Stacks in User Environments

这里要介绍一个新的栈：用户异常栈(Exception Stack)。在此之前，我们已经有两个栈：内核栈，用户运行栈。

内核栈：在 kernel 代码正常执行时，包括发生中断/异常陷入 kernel 时，kernel 运行的栈称为“内核栈(kernel stack)”，它的 `esp` 寄存器指向当前 CPU 的栈顶，栈的线性地址范围为 `[kstacktop_i - KSTACKSIZE, kstacktop_i - 1]`。中断/异常发生时，CPU 自动切换到内核栈。我们在报告的 1.2 部分已经设置过内核栈。

用户运行栈：在用户程序正常执行时，它所运行的栈称为“用户运行栈(normal user stack)”，它的 `esp` 寄存器指向 `USTACKTOP`，栈的线性地址范围是 `[USTACKTOP - PGSIZE, USTACKTOP - 1]`。

用户异常栈：是用户自己定义相应的中断处理程序后，相应处理程序运行时的栈。注意这个处理程序是 user mode 的，而不是在内核中。它的 `esp` 寄存器指向 `UXSTACKTOP`，栈的线性地址范围是 `[UXSTACKTOP - PGSIZE, UXSTACKTOP - 1]`，也是一页大小。

现在，用户空间发生了一个 page fault。理论上用户程序执行不下去了，它将陷入内核栈，然后在内核中处理 page fault，然后返回用户程序。但是，这里我们将采用一个新的机制！也就是不再在内核中处理 page fault，而是在 user mode 的用户异常栈上！

<Step 1> 用户空间发生 page fault 后，首先陷入内核（**用户运行栈→内核栈，user mode → kernel mode**），进入 `trap()` 处理中断分发，进入 `page_fault_handler()`。

<Step 2> 然后内核中的 `page_fault_handler()` 在确认是用户程序而不是内核触发了 page fault 后（如果是内核发生 page fault 就直接 panic 了），在用户异常栈上压入一个 `UTrapframe` 作为记录保存现场的信息（注意这只是内存操作，还没有切换到用户异常栈）。

<Step 3> kernel 将用户程序切换到用户异常栈（**内核栈→用户异常栈，kernel mode → user mode**），然后就让用户程序重新运行了！什么？page fault 都没有处理怎么就让用户程序运行了？这里有两点要说明：①现在用户程序是运行在用户异常栈（而不是用户运行栈）上，②此时的 `eip` 被设置为 page fault 处理程序的起始地址 `curenv->env_pgfault_upcall`。所以，接下来用户程序执行处理 page fault 的处理程序，它实际上是在用户空间中进行处理。

<Step 4> page fault 被处理完后，切换回用户运行栈（**用户异常栈→用户运行栈, user mode → user mode**），用户程序重新运行。

弄清楚 page fault 发生前后“栈的切换”、“特权级别(mode)的切换”是非常重要的。

4.1.3 Invoking the User Page Fault Handler

Exercise 8. Implement the code in `page_fault_handler` in `kern/trap.c` required to dispatch page faults to 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?)

在上面的流程中，<step 1>是 Lab 3 就已经做好的内容，接下来我们要实现<step 2>，也就是 `kern/trap.c` 中的 `page_fault_handler()` 函数。简单地说，这个函数就是对不同 page fault 的处理进行分发。

我们约定，用户程序出现 page fault 陷入内核前的状态为 trap-time state.

对于 kernel 出现的 page fault，我们已经处理，就是 panic.

对于 user 出现的 page fault 分发流程如下：

①判断 page fault handler 的地址是否已被注册好，否则用户无法处理 page fault，销毁用户进程。

②kernel 在用户异常栈上压入一个 UTrapframe，也就是保护现场的信息：

```

                                <-- UXSTACKTOP
trap-time esp
trap-time eflags
trap-time eip
trap-time eax          start of struct PushRegs
trap-time ecx
trap-time edx
trap-time ebx
trap-time esp
trap-time ebp
trap-time esi
trap-time edi          end of struct PushRegs
tf_err (error code)
fault_va              <-- %esp when handler is run
```

UTrapframe 保存在用户异常栈中，用于保护现场，为什么使用 UTrapframe 而不用现成的 Trapframe？根据张驰学长的报告，UTrapframe 是特别设计给用户自定义的、user mode 中的中断错误处理程序的（不一定是 page fault 处理程序），因此 UTrapframe 中有一个特殊的成员 `fault_va` 表示访问出错的指令涉及的地址。而 UTrapframe 中没有 `es`，`ds`，`ss` 等段寄存器的信息，这是因为“user mode 中断处理程序”和“用户程序本身”实际上属于同一个进程，因此不涉及段的切换。总的来说，Trapframe 保存了进程的现场的完整信息，而 UTrapframe 是为用户中断处理程序专门定制的，并没有在操作系统的其他地方使用到。

最后来谈谈我对“压栈”的理解。首先，“栈”在内存中，和其他内存地址并没有实质性区别，不过栈可由专门的寄存器指向：“堆栈段寄存器(SS, Stack Segment)”和“堆栈指针寄存器

[SP, Stack Pointer]”。其次，栈是一片分配好的内存空间，以特殊的顺序访问。第三，什么叫“压栈”，压栈的本质其实就是向内存写入数据的过程，只是说这个“写”是有讲究的，要按照一定的顺序，才能保证正确的入栈出栈。怎么写？把 SP 进行一定的偏移，就是可以写入的线性地址，**把它当成一个指针就好**。还要注意，栈是从高地址向低地址生长的，而我们一般的内存操作是从低地址开始操作的，两者方向相反，因此一般是减小 SP 后再从低地址向上写入数据。

③关于递归产生的 page fault. 如果用户进程已经在用户异常栈上运行，那就意味着在 user mode 的 page fault handler 再次产生 page fault. 这时，我们应该从当前 tf->esp 开始压栈而不是栈顶 UXSTACKTOP. 在压入一个新的 UTrapframe 之前，记得先压入一个空的 4 字节，这空的 4 字节后面还有大用途。

page_fault_handler()代码如下：

```
void
page_fault_handler(struct Trapframe *tf)
{
    uint32_t fault_va;
    // Read processor's CR2 register to find the faulting address
    fault_va = rcr2();

    // Handle kernel-mode page faults.
    if(tf->tf_cs == GD_KT){
        cprintf("fault va = %08x, ip = %08x\n", fault_va, tf->tf_eip);
        panic("kern/trap.c/page_fault_handler: kernel page fault!\n");
    }

    // We've already handled kernel-mode exceptions, so if we get here,
    // the page fault happened in user mode.
    // LAB 4: Your code here.
    if (curenv->env_pgfault_upcall != NULL){
        struct UTrapframe *utf;
        if (UXSTACKTOP-PGFSIZE <= tf->tf_esp && tf->tf_esp < UXSTACKTOP)
            utf = (struct UTrapframe *)
                (tf->tf_esp - sizeof(struct UTrapframe) - 4);
        else
            utf = (struct UTrapframe *)
                (UXSTACKTOP - sizeof(struct UTrapframe));
        // 为什么要先减呢？注意，栈是自顶向下生长的，而我们的内存访问是自底向上的！
        // 因此指针当然要指向一片内存区域的【低端】起始地址！

        user_mem_assert (curenv, (void *)utf, sizeof(struct UTrapframe),
PTE_U|PTE_W);

        utf->utf_eflags = tf->tf_eflags;
        utf->utf_eip = tf->tf_eip;
        utf->utf_err = tf->tf_err;
```

```

utf->utf_esp = tf->tf_esp;
utf->utf_fault_va = fault_va;
utf->utf_regs = tf->tf_regs;

curenv->env_tf.tf_eip = (uint32_t) curenv->env_pgfault_upcall;
curenv->env_tf.tf_esp = (uint32_t) utf;
env_run (curenv); // This does not return
}

// Destroy the environment that caused the fault.
printf("[%08x] user fault va %08x ip %08x\n",
       curenv->env_id, fault_va, tf->tf_eip);
print_trapframe(tf);
env_destroy(curenv);
}

```

回答 exercise 8 题干后面的那个问题，如果用户异常栈空间不足了怎么办。答案是，那就没辙了呗，系统就出错了。要不然谁来给它处理 page fault？

4.1.4 User-mode Page Fault Entrypoint

Exercise 9. 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.

现在要写 `lib/pfentry.S` 中的 `_pgfault_upcall` 过程的汇编码。这个汇编的过程是干什么用的？在哪里会被调用？在 4.1.1 中谈到 page fault handler 的 entry code 时我说过，“等一下会有点绕”，现在这里就是我说的地方。`_pgfault_upcall` 过程是实际上的 page fault handler 的 entry code，更准确地说，它是我们所期望的 page fault handler 的包装，page fault handler 将在 `_pgfault_upcall` 中被调用。那么为什么不是直接调用 page fault handler，还需要经过一个看似冗余的 `_pgfault_upcall`？

下面我来介绍三种返回：

(1) 一般的函数返回。当函数执行完后，要返回调用函数的地方。怎么返回？这是从函数调用栈（用户运行栈）上 pop 出上一个函数的信息，也即将 SP 指向上一个栈帧。这种返回不涉及堆栈切换。

(2) 一般的中断/异常返回。中断/异常发生后，用户运行栈被切换成内核栈，当处理程序执行完，它要返回发生中断/异常的用户程序的代码地址。怎么返回？这是通过 Trapframe 中保存的现场信息，将所有的寄存器的值复原，这样就返回用户程序了。

(3) 用户的中断处理函数的返回。这和一般的函数返回不一样，因为用户的中断处理程序运行在用户异常栈上，因此要涉及堆栈切换（用户异常栈→用户运行栈）。我们要进行栈切换，同时恢复 eip，这时，我们遇到了一个小小的矛盾：

★ 如果先切换栈（用户异常栈→用户运行栈），那么切换完了之后你去哪里找 eip？此时我们在用户运行栈上，而我们要恢复回去的 eip 保存在用户异常栈中。

★ 如果先恢复 eip，那么下一条指令就是用户程序中的某条指令，而不是堆栈切换的指

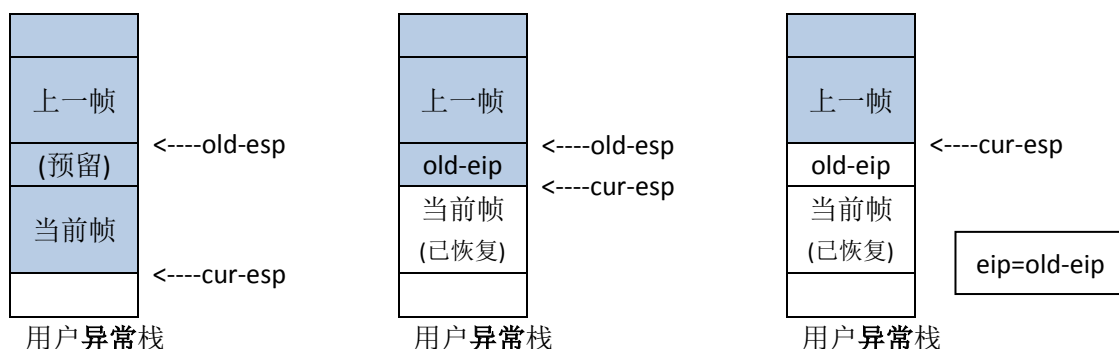
令，那么如何完成栈切换？

于是这里使用了一个**神奇的 trick**：

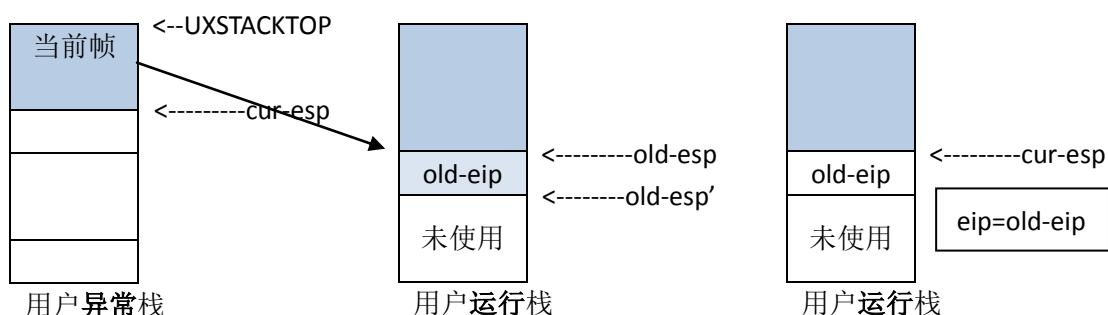
(1) 如果是递归出现的 page fault. 我们还记得在压入这一帧的 UTrapframe 时，多让 SP 减了 4：

```
if (UXSTACKTOP-PGFSIZE <= tf->tf_esp && tf->tf_esp < UXSTACKTOP)
    utf=(struct UTrapframe *) (tf->tf_esp - sizeof(struct UTrapframe) - 4);
```

这样一来，old-esp 所指向的地址的下面 4 字节，是被预留的（如左图）。因此，我们可以用这个预留的位置保存 old-eip，然后将该恢复的寄存器恢复以后，让 cur-esp（当前栈顶指针）指向保存 old-eip 的地址（如中图），这样，只要调用 ret 指令（相当于 pop %eip），就能够同时恢复栈顶地址和 eip。



(2) 如果是第一个 page fault. 这时，没有“上一帧”，当前栈帧就顶着 UXSTACKTOP，而此时的 old-esp 指向的是“用户运行栈”的栈顶，而不是用户异常栈。那么我们知道，用户运行栈的栈顶以下目前是没有被使用的，因此，可以将返回的 eip 保存在用户运行栈的栈顶下面 4 字节，不会影响用户运行栈原有的数据。待切换回用户运行栈后，同样 pop 出 eip 即可。如下图。注意，这和上面的“预留”机制类似，但是保存 eip 的位置是明显不同的。



这就是所谓“the assembly routine that will take care of calling the C page fault handler and resume execution at the original faulting instruction.”，只可意会不可言传，我都想不出中文该怎么翻译了。_pgfault_upcall 代码如下，代码的更详细的解释、示意图参看张驰学长的报告：

```
.text
.globl _pgfault_upcall
_pgfault_upcall:
// Take care of calling the C page fault handler.
pushl %esp          // function argument: pointer to UTF
movl _pgfault_handler, %eax
```

```

call *%eax
addl $4, %esp          // pop function argument

// Now the C page fault handler has returned and you must return
// to the trap time state.
// Push trap-time %eip onto the trap-time stack.

// subtract 4 from old-esp for the space of storage of old-eip
movl 0x30(%esp), %eax
subl $0x4, %eax
movl %eax, 0x30(%esp)
// put old-eip in the reserved 4 bytes space
movl 0x28(%esp), %ebx
movl %ebx, 0(%eax)

// Restore the trap-time registers. After you do this, you
// can no longer modify any general-purpose registers.
addl $0x8, %esp
popal

// Restore eflags from the stack. After you do this, you can
// no longer use arithmetic operations or anything else that
// modifies eflags.
addl $0x4, %esp
popfl

// Switch back to the adjusted trap-time stack.
pop %esp

// Return to re-execute the instruction that faulted.
ret

```

Exercise 10. Finish `set_pgfault_handler()` in `lib/pgfault.c`.

这和 exercise 7 有什么区别？实际上这是对 exercise 7 中 `sys_env_set_pgfault_upcall()` 的包装。`set_pgfault_handler()` 是一个库函数，用户传入 `handler` 起始地址，我们并不是将该地址通过 `sys_env_set_pgfault_upcall()` 系统调用注册到 `Env` 中；而是对于每个 `Env`，我们都用 exercise 9 的汇编过程 `_pgfault_upcall` 去注册，然后将 `handler` 的起始地址记录在一个全局变量 `_pgfault_handler` 中，再在 `_pgfault_upcall` 中调用 `_pgfault_handler`。这看上去有点“绕”。

下面这几个东西非常容易混淆，应注意区分清楚：

·kern/syall.c:	<code>sys_env_set_pgfault_upcall()</code>
·lib/pgfault.c:	<code>set_pgfault_handler()</code>
·lib/pfentry.S:	<code>_pgfault_upcall</code>
·lib/pgfault.c:	<code>_pgfault_handler</code>

最后贴出 `set_pgfault_handler()`代码如下:

```
void
set_pgfault_handler(void (*handler) (struct UTrapframe *utf))
{
    int r;

    if (_pgfault_handler == 0) {
        if ((r = sys_page_alloc (0, (void *) (UXSTACKTOP-PGSIZE),
PTE_U|PTE_W|PTE_P)) < 0) {
            panic ("set_pgfault_handler: %e\n", r);
        }
        sys_env_set_pgfault_upcall (0, _pgfault_upcall);
    }
    // Save handler pointer for assembly to call.
    _pgfault_handler = handler;
}
```

4.1.5 Testing

简要解释一下几个用户程序是什么意思。

`user/faultread.c`:

```
// buggy program - faults with a read from location zero
// 这个函数试图从地址0x0读入一个整数，产生缺页，但是没有设置用户中断处理函数的
// 起始地址，因此缺页后会在page_fault_handler()中被destroy.
#include <inc/lib.h>
void umain(int argc, char **argv) {
    cprintf("I read %08x from location 0!\n", *(unsigned*)0);
}
```

`user/faultdie.c`:

```
// test user-level fault handler -- just exit when we fault
// 这个函数试图访问地址0xDeadBeef并写入一个整数，产生缺页
// 然后在用户处理函数handler()中调用系统调用destroy自己.
#include <inc/lib.h>
void handler(struct UTrapframe *utf) {
    void *addr = (void*)utf->utf_fault_va;
    uint32_t err = utf->utf_err;
    cprintf("i faulted at va %x, err %x\n", addr, err & 7);
    sys_env_destroy(sys_getenvid());
}
void umain(int argc, char **argv) {
    set_pgfault_handler(handler);
    *(int*)0xDeadBeef = 0;
}
```

`user/faultalloc`:

```
// test user-level fault handler -- alloc pages to fix faults
```

```

// 这个函数试图访问这两个地址，0xDeadBeef和0xCafeBffe，并从中读取字符串
// ①对于0xDeadBeef，当它产生页面错误进入handler后，就分配一页，
// 然后调用snprintf()（Lab1，最好跟踪进去看一下）往这个地址写入一串字符
// 页面错误返回用户程序后，cprintf输出刚才写入的字符串
// ②对于0xCafeBffe，当它产生页面错误进入handler后，就分配一页，
// 然后调用snprintf()写入一串字符，【但是0xCafeBffe恰好位于页面边界】
// 因此在handler中递归产生一个页面错误，分配一页
// 最后错误都处理完后，cprintf()输出字符串
#include <inc/lib.h>
void handler(struct UTrapframe *utf){
    int r;
    void *addr = (void*)utf->utf_fault_va;
    cprintf("fault %x\n", addr);
    if((r=sys_page_alloc(0,ROUNDDOWN(addr,PGSIZE),PTE_P|PTE_U|PTE_W)) <0)
        panic("allocating at %x in page fault handler: %e", addr, r);
    snprintf((char*)addr, 100, "this string was faulted in at %x", addr);
}
void umain(int argc, char **argv){
    set_pgfault_handler(handler);
    cprintf("%s\n", (char*)0xDeadBeef);
    cprintf("%s\n", (char*)0xCafeBffe);
}

```

user/faultallocbad:

```

// test user-level fault handler -- alloc pages to fix faults
// doesn't work because we sys_cputs instead of cprintf (exercise: why?)
// 这个函数handler和上一个一样，但是访问地址0xDEADBEEF时调用的是sys_cputs()
// 而不是cprintf().sys_cputs()访问地址前会调用user_mem_assert,而此时地址
// 尚未分配内存页面，因此报错。注释掉user_mem_assert怎么样？那么pagefault将在
// kernel中产生，然后panic.
#include <inc/lib.h>
void handler(struct UTrapframe *utf){
    int r;
    void *addr = (void*)utf->utf_fault_va;
    cprintf("fault %x\n", addr);
    if((r=sys_page_alloc(0,ROUNDDOWN(addr,PGSIZE),PTE_P|PTE_U|PTE_W)) <0)
        panic("allocating at %x in page fault handler: %e", addr, r);
    snprintf((char*) addr, 100, "this string was faulted in at %x", addr);
}
void umain(int argc, char **argv){
    set_pgfault_handler(handler);
    sys_cputs((char*)0xDEADBEEF, 4);
}

```

4.2 Implementing Copy-on-Write Fork

Exercise 11. Implement `fork`, `duppage` 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.

```
1008: I am "  
1009: I am '0'  
2008: I am '00'  
2009: I am '000'  
100a: I am '1'  
3008: I am '11'  
3009: I am '10'  
200a: I am '110'  
4008: I am '100'  
100b: I am '01'  
5008: I am '011'  
4009: I am '010'  
100c: I am '001'  
100d: I am '111'  
100e: I am '101'
```

现在，在用户空间实现 `lib/fork.c` 的 `fork()` 函数，应该万事俱备了。

还记得我们有一个用户程序 `user/dumbfork.c`，这个程序对实现 `fork()` 具有很高的参考价值。`fork()` 需要实现的功能大致如下，首先创建进程的 `Env`，然后遍历父进程线性地址空间，将父进程的页面映射复制给子进程，以及对子进程进行其他必要的初始化。

`fork()` 基本的控制流如下：

1. 父进程调用 `set_pgfault_handler()` 将它的 page fault handler 设为 `pgfault()`，这个函数也是定义在 `lib/fork.c` 中。

2. 父进程调用 `sys_exofork()` 为子进程分配 `Env` 结构，并复制寄存器的值等。

3. 遍历父进程的地址空间的 `UTOP` 以下的部分，对于每张“可写(writable)”或“写时复制”页面，父进程调用 `duppage()` 函数（也是定义在 `lib/fork.c` 中）将该页面映射到子进程的地址空间中，并标记为“写时复制”。**注意**，这里还要将被映射的页面，重新以“写时复制”的方式映射回父进程的地址空间。**为什么？**这是为了防止出现“某个进程写入一个页面对另一个页面可见(the modification of memory is not private)”的情况，因此，不论是父进程还是子进程，当写入共享的 `COW` 页面时，均会产生 `page fault` 分配新页面，并复制原有页面的内容（见 `pgfault()` 函数）。那么如果父子都写入，原来那张页面怎么办，我们并没有明确地释放这张页面啊？没错，不过当我们向同一个、已经映射了页面的线性地址，再次映射一张页面时，它会将原有页面的引用数减一，如果引用数为 0 则释放原有页面。这样就不会出现我们担心的问题。

注意，尽管用户异常栈的地址 `UXSTACKTOP` 在 `UTOP` 以下，但是它当然不能映射为“写时复制”。因此，需要为它立即分配并映射一张新的物理页框。

此外, `fork()` 还需要处理那些 `PTE_P` 但是不是 `PTE_W` 或 `COW` 的映射, 复制时标记为只读即可。

4. 在父进程真正让子进程运行之前, 它还要帮助子进程设置 `page fault handler` 的起始地址。

5. 父进程将子进程设置为 `RUNNABLE`.

每当一个进程试图第一次写入一张 `COW` 页面时, 它将会产生一个 `page fault`, 处理这个 `page fault` 的流程归纳如下:

1. 用户产生 `page fault`, 按照一般的中断流程, 进入 `kern/trap.c` 的 `page_fault_handler()` 函数, 接着进入汇编码 `_pgfault_upcall`. 在 `_pgfault_upcall` 中调用 `lib/fork.c` 中的 `pgfault()` 函数, 这才是真正的处理程序。

2. `pgfault()` 检查错误如果是写入, 则对应地址的页目录项是否为 `PTE_COW`, 若不是 `panic`.

3. `pgfault()` 分配一张新页面, 接着将它映射到一个临时的线性地址 `PFTEMP`, 然后用 `memmove()` 将原页面的内容复制到新页面, 这样物理页中的 `data` 就有两份了。最后将 `PFTEMP` 处的临时页面映射到正确的地址, 并标记为相应读写权限, 替换掉原有的映射。

4. 从用户异常栈直接切换回用户栈, 用户程序继续运行。

最后来看代码:

`lib/fork.c` 的 `fork()`:

```
envid_t
fork(void)
{
    set_pgfault_handler(pgfault);

    envid_t envid;
    if ((envid = sys_exofork()) < 0)
        panic("lib/fork.c/fork(): %e", envid);

    if (envid == 0) {
        // We're the child.
        thisenv = &envs[ENVX(sys_getenvid())];
        return 0;
    }

    // We're the parent.
    uint32_t addr;
    for (addr = UTEXT; addr < UXSTACKTOP - PGSIZE; addr += PGSIZE) {
        if ( (vpd[PDX(addr)] & PTE_P) > 0
            && (vpt[PGNUM(addr)] & PTE_P) > 0
            && (vpt[PGNUM(addr)] & PTE_U) > 0)
            duppage (envid, PGNUM(addr));
    }
}
```

```

    // alloc a page for child's exception stack
    int r = sys_page_alloc (envid, (void *) (UXSTACKTOP - PGSIZE),
PTE_U|PTE_W|PTE_P);
    if (r < 0)
        panic ("lib/fork.c/fork(): sys_page_alloc failed: %e", r);

    // Why? Because the new env cannot set its _pgfault_upcall by itself!
    // When it go out of here, it will behave the same as its father.
    extern void _pgfault_upcall (void);
    sys_env_set_pgfault_upcall (envid, _pgfault_upcall);

    r = sys_env_set_status (envid, ENV_RUNNABLE);
    if (r < 0)
        panic ("lib/fork.c/fork(): set child env status failed : %e", r);

    return envid;
}

```

lib/fork.c 的 duppage():

```

static int
duppage (envid_t envid, unsigned pn)
{
    int r;
    void *addr = (void *) ((uint32_t)pn*PGSIZE);
    pte_t pte = vpt[PGNUM(addr)];

    if ((pte & PTE_W) > 0 || (pte & PTE_COW) > 0) {
        // map the va into the envid's address space
        r = sys_page_map(0, addr, envid, addr, PTE_U|PTE_P|PTE_COW);
        if (r < 0)
            panic ("lib/fork.c/duppage(): sys_page_map (new) failed: %e",
r);

        // map the va (again) into the envid's address space
        // this is a mechanism to guarantee consistency of the shared page,
        // i.e., if the old process want to modify the page, it has to COW
        as well!
        r = sys_page_map(0, addr, 0, addr, PTE_U|PTE_P|PTE_COW);
        if (r < 0)
            panic ("lib/fork.c/duppage(): sys_page_map (old) failed: %e",
r);
    }
    else {
        // map the va (read-only)
        r = sys_page_map(0, addr, envid, addr, PTE_U|PTE_P);
        if (r < 0)

```

```

        panic ("lib/fork.c/duppage(): sys_page_map (read-only)
failed: %e", r);
    }
    return 0;
}

```

lib/fork.c 的 pgfault():

```

static void
pgfault(struct UTrapframe *utf)
{
    void *addr = (void *) utf->utf_fault_va;
    uint32_t err = utf->utf_err;
    int r;

    if((err & FEC_WR) == 0)
        panic("lib/fork.c/pgfault(): the faulting access was not a
write!");
    if((vpd[PDX(addr)] & PTE_P)==0 || (vpt[PGNUM(addr)] & PTE_COW)==0)
        panic("lib/fork.c/pgfault(): the faulting access was not to a
copy-on-write page");

    // Allocate a new page, map it at a temporary location (PFTEMP)
    r = sys_page_alloc (0, (void *)PFTEMP, PTE_U|PTE_W|PTE_P);
    if (r < 0)
        panic("lib/fork.c/pgfault(): sys_page_alloc failed: %e", r);

    // copy the data from the old page to the new page
    addr = ROUNDDOWN (addr, PGSIZE);
    memmove (PFTEMP, addr, PGSIZE);

    // map new page it into the old page's address
    // (this will implicitly unmap old page and decrease the physical page's
"ref" by one.)
    r = sys_page_map (0, PFTEMP, 0, addr, PTE_U|PTE_W|PTE_P);
    if (r < 0)
        panic("lib/fork.c/pgfault(): sys_page_map failed: %e", r);
}

```

运行一下 user/forktree.c, 结果和期待中一样。

5. (Part C) Clock Interrupts and Preemption

在 part C, 我们将进一步完善“抢占式调度”, 抢占那些“不合作”的用户进程; 我们还要实现进程之间通信的机制。

“不合作”的用户进程, 就像 user/spin.c 创建出来的子进程一样, 它们通过死循环, 一旦获得 CPU 的控制权就在上面不下来了, 连它们的父进程和 kernel 都没有办法。为了避免

这种情况，kernel 要能够抢占运行中的用户进程，要能够将 CPU 控制权抢过来。这需要 JOS 能够支持从外部时钟硬件发来的时钟中断。

5.1 Interrupt discipline

外部中断（也就是非 CPU 的外部设备产生的中断）记作 IRQ。一共有 16 种可能的外部中断，编号为 0 至 15。但是外部中断在 IDT 中的表项并不是固定不变的。kern/picirq.c 的 pic_init() 将 0~15 号外部中断映射到 IDT 表项的 IRQ_OFFSET 至 IRQ_OFFSET+15。而在 inc/trap.h 中 IRQ_OFFSET 被定义为 32，因此外部中断的 IDT 表项为 32~47。其中，IRQ 0 是时钟中断，因此 IDT[IRQ_OFFSET+0] 中包含了时钟中断处理程序的起始地址。

在 JOS 中，我们令当控制权在 kernel 中时，外部设备的中断总是被屏蔽(disable)，在用户空间是总是被开启(enable)。这是通过设置 %eflags 寄存器的 FL_IF 位来控制的。当它置为 1 时，外部设备中断被开启。因此，我们只要注意，在进入和离开用户空间时对 %eflags 寄存器的 FL_IF 位进行相应设置即可。在 bootloader 的代码中，我们已将 FL_IF 位关闭。

Exercise 12. 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.

The processor never pushes an error code or checks the Descriptor Privilege Level (DPL) of the IDT entry when invoking a hardware interrupt handler. You might want to re-read section 9.2 of the 80386 Reference Manual, or section 5.8 of the IA-32 Intel Architecture Software Developer's Manual, Volume 3, at this time.

After doing this exercise, if you run your kernel with any test program that runs for a non-trivial length of time (e.g., spin), you should see the kernel print trap frames for hardware interrupts. While interrupts are now enabled in the processor, JOS isn't yet handling them, so you should see it misattribute each interrupt to the currently running user environment and destroy it. Eventually it should run out of environments to destroy and drop into the monitor.

首先增加一些 handler，并对 IDT 进行初始化。我们只需要了解如下信息：对于发生的外部中断，CPU 不会压入 error code，也不会检查 IDT 的 DPL。

kern/trapentry.S 增加如下代码：

```
# for IRQ handler
TRAPHANDLER_NOEC(routine_irq0, IRQ_OFFSET + 0);
TRAPHANDLER_NOEC(routine_irq1, IRQ_OFFSET + 1);
TRAPHANDLER_NOEC(routine_irq2, IRQ_OFFSET + 2);
TRAPHANDLER_NOEC(routine_irq3, IRQ_OFFSET + 3);
TRAPHANDLER_NOEC(routine_irq4, IRQ_OFFSET + 4);
TRAPHANDLER_NOEC(routine_irq5, IRQ_OFFSET + 5);
TRAPHANDLER_NOEC(routine_irq6, IRQ_OFFSET + 6);
TRAPHANDLER_NOEC(routine_irq7, IRQ_OFFSET + 7);
TRAPHANDLER_NOEC(routine_irq8, IRQ_OFFSET + 8);
TRAPHANDLER_NOEC(routine_irq9, IRQ_OFFSET + 9);
```

```
TRAPHANDLER_NOEC(routine_irq10, IRQ_OFFSET + 10);
TRAPHANDLER_NOEC(routine_irq11, IRQ_OFFSET + 11);
TRAPHANDLER_NOEC(routine_irq12, IRQ_OFFSET + 12);
TRAPHANDLER_NOEC(routine_irq13, IRQ_OFFSET + 13);
TRAPHANDLER_NOEC(routine_irq14, IRQ_OFFSET + 14);
TRAPHANDLER_NOEC(routine_irq15, IRQ_OFFSET + 15);
```

kern/trap.c 的 trap_init()函数增加如下代码:

```
extern void routine_irq0 ();
extern void routine_irq1 ();
extern void routine_irq2 ();
extern void routine_irq3 ();
extern void routine_irq4 ();
extern void routine_irq5 ();
extern void routine_irq6 ();
extern void routine_irq7 ();
extern void routine_irq8 ();
extern void routine_irq9 ();
extern void routine_irq10 ();
extern void routine_irq11 ();
extern void routine_irq12 ();
extern void routine_irq13 ();
extern void routine_irq14 ();
extern void routine_irq15 ();
```

SETGATE (idt[IRQ_OFFSET + 0], 0, GD_KT, routine_irq0, 0); //
privilege is set 0: The processor never pushes an error code or checks
the Descriptor Privilege Level (DPL) of the IDT entry when invoking a
hardware interrupt handler.

```
SETGATE (idt[IRQ_OFFSET + 1], 0, GD_KT, routine_irq1, 0);
SETGATE (idt[IRQ_OFFSET + 2], 0, GD_KT, routine_irq2, 0);
SETGATE (idt[IRQ_OFFSET + 3], 0, GD_KT, routine_irq3, 0);
SETGATE (idt[IRQ_OFFSET + 4], 0, GD_KT, routine_irq4, 0);
SETGATE (idt[IRQ_OFFSET + 5], 0, GD_KT, routine_irq5, 0);
SETGATE (idt[IRQ_OFFSET + 6], 0, GD_KT, routine_irq6, 0);
SETGATE (idt[IRQ_OFFSET + 7], 0, GD_KT, routine_irq7, 0);
SETGATE (idt[IRQ_OFFSET + 8], 0, GD_KT, routine_irq8, 0);
SETGATE (idt[IRQ_OFFSET + 9], 0, GD_KT, routine_irq9, 0);
SETGATE (idt[IRQ_OFFSET + 10], 0, GD_KT, routine_irq10, 0);
SETGATE (idt[IRQ_OFFSET + 11], 0, GD_KT, routine_irq11, 0);
SETGATE (idt[IRQ_OFFSET + 12], 0, GD_KT, routine_irq12, 0);
SETGATE (idt[IRQ_OFFSET + 13], 0, GD_KT, routine_irq13, 0);
SETGATE (idt[IRQ_OFFSET + 14], 0, GD_KT, routine_irq14, 0);
SETGATE (idt[IRQ_OFFSET + 15], 0, GD_KT, routine_irq15, 0);
```

接着修改一下 kern/env.c 中的 env_alloc(), 在进程创建时, 将它的%eflags 寄存器中的 FL_IF 位置为 1, 开启外部中断。代码如下:

```
e->env_tf.tf_eflags |= FL_IF;
```

5.2 Handling Clock Interrupts

Exercise 13. 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.

You should now be able to get the user/spin test to work: the parent environment should fork off the child, sys_yield() to it a couple times but in each case regain control of the CPU after one time slice, and finally kill the child environment and terminate gracefully.

在 kern/init.c 的 i386_init() 函数中调用了 lapic_init(), 为我们初始化时钟及其中断控制器。我们开启了外部中断, 但是还没有处理他们。

修改 kern/trap.c 中的 trap_dispatch() 函数, 使得当发生时钟中断时, 它调用 sched_yield() 函数, 调度另一个用户进程。不过根据注释, 事先还要调用一个 lapic_eoi() 用于确认中断信号。代码如下:

```
// Handle clock interrupts. Don't forget to acknowledge the
// interrupt using lapic_eoi() before calling the scheduler!
// LAB 4: Your code here.
case (IRQ_OFFSET+IRQ_TIMER):
    lapic_eoi();
    sched_yield();
    return;
```

代码测试通过。

6. (Part C) Inter-Process communication (IPC)

之前我们一直集中于操作系统如何保证进程之间的独立性, 为进程制造了一种假象, 似乎进程独占一台机器。操作系统的另一项重要服务, 是让进程之间能够通信。Unix 的“管道模型”是一个进程通信的例子。

6.1 IPC in JOS

接下来我们要实现两个系统调用 sys_ipc_recv 和 sys_ipc_try_send, 以及他们在用户库中的包装 ipc_recv 和 ipc_send, 来提供简单的进程间的通信机制。

JOS 中进程通信的消息包括两个部分: 一个 32 位的值, 以及(可选的)一个页面映射。后者能够让进程之间一次传递更多的值, 而不仅仅是一个 32 位的整数; 同时也让进程之间能够更容易共享内存地址。

6.2 Sending and Receiving Messages

接收消息：进程调用 `sys_ipc_rcv`，接着系统将阻塞当前进程，直到消息被接收到。当一个进程在等待接收消息时，任何其他进程都可以向它发送消息。

发送消息：进程给定接收进程的 `id` 和待发送的内容，调用 `sys_ipc_try_send`，如果目标进程的确在等待消息，那么就将消息传递过去并返回 0；否则返回 `-E_IPC_NOT_RECV` 表示目标进程当前并不在等待接收消息。

用户库函数 `ipc_rcv` 将 `sys_ipc_rcv` 进行封装，并从当前进程的 `Env` 结构中取出接收到的值。用户库函数 `ipc_send` 负责不断地尝试调用 `sys_ipc_try_send` 直到消息发送成功。

6.3 Transferring Pages

当一个接收进程调用 `sys_ipc_rcv` 并传入一个 `UTOP` 以下的地址变量 `dstva` 时，这就意味着它希望接收一个页面映射。如果发送者发送了一个页面，那么这个页面将被映射到接收者地址空间的 `dstva`。如果这个地址已经映射了一个页面，那么被覆盖的页面将被 `unmap`。

当一个发送进程调用 `sys_ipc_try_send` 并传入一个 `UTOP` 以下的地址变量 `srcva` 时，这就意味着发送者希望发送一个当前被映射在 `srcva` 的页面给接收者，权限指定为 `perm`。如果进程通信成功，那么发送者仍在其地址 `srcva` 保留原有的页面映射，而接收者将在其地址 `dstva` 处与发送者共享同一个映射，也就是映射到同一个物理页框！反正就这样规定了。

如果发送者或接收者的任何一方，并没有表示出希望接收一个页面映射的意愿，那么页面映射将不会被传递。在任何的 IPC 之后，`kernel` 将接收者 `Env` 结构中接收到的页面的权限位增加对 `env_ipc_perm` 的设置，如果没有接收到页面，则增加的设置位为 0。

6.4 Implementing IPC

Exercise 14. Implement `sys_ipc_rcv` and `sys_ipc_try_send` in `kern/syscall.c`. Read the comments on both before implementing them, since they have to work together. 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_rcv` and `ipc_send` functions in `lib/ipc.c`.

Use the `user/pingpong` and `user/primes` functions to test your IPC mechanism. You might find it interesting to read `user/primes.c` to see all the forking and IPC going on behind the scenes.

`sys_ipc_rcv` 代码如下：

```
static int
sys_ipc_rcv(void *dstva)
{
    if (dstva < (void *)UTOP && ROUNDDOWN (dstva, PGSIZE) != dstva)
        return -E_INVAL;

    curenv->env_ipc_recving = 1;
```

```

    curenv->env_ipc_dstva = dstva;
    curenv->env_ipc_from = 0; // to mark that no sender has send any value
    to us.
    curenv->env_status = ENV_NOT_RUNNABLE;

    sched_yield();
    return 0;
}

```

这里面要注意，将 `curenv->env_ipc_from` 设置为 0，表示当前还没有接收到任何进程发来的消息。这样是防止两个进程同时向同一个进程发送消息，后来的发送者将前面的发送者发送的消息覆盖掉。

`sys_ipc_try_send` 的代码如下：

```

static int
sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned
perm)
{
    struct Env *dstenv;
    int r;

    // get target env (may not exist)
    if ((r = envid2env (envid, &dstenv, 0)) < 0)
        return -E_BAD_ENV;

    // if envid is not currently blocked in sys_ipc_recv, or another
    environment managed to send first.
    if (!dstenv->env_ipc_recving || dstenv->env_ipc_from != 0)
        return -E_IPC_NOT_RECV;

    // if srcva < UTOP but srcva is not page-aligned
    if (srcva < (void *)UTOP && ROUNDDOWN(srcva, PGSIZE) != srcva)
        return -E_INVAL;

    // if srcva < UTOP and perm is inappropriate (see sys_page_alloc)
    if (    srcva < (void *)UTOP
        && (!(perm & PTE_U) || !(perm & PTE_P) || (perm & ~PTE_SYSCALL) != 0))
        return -E_INVAL;

    pte_t *pte;
    struct Page *pp;

    // if srcva < UTOP but srcva is not mapped in the caller's address
    space.
    if (    srcva < (void *)UTOP
        && (pp = page_lookup (curenv->env_pgdir, srcva, &pte)) == NULL)

```



```

        return -E_INVALID;

    // if (perm & PTE_W), but srcva is read-only in the current
    // environment's address space.
    if (    srcva < (void *)UTOP
        && (perm & PTE_W) > 0
        && (*pte & PTE_W) == 0)
        return -E_INVALID;

    // send a page
    // if there's not enough memory to map srcva in env_id's address space.
    if (    srcva < (void *)UTOP
        && dstenv->env_ipc_dstva != 0) {
        r = page_insert (dstenv->env_pgdir, pp, dstenv->env_ipc_dstva,
perm);
        if (r < 0)
            return -E_NO_MEM;

        dstenv->env_ipc_perm = perm;
    }

    dstenv->env_ipc_from = curenv->env_id;
    dstenv->env_ipc_value = value;
    dstenv->env_status = ENV_RUNNABLE;
    dstenv->env_ipc_recving = 0;
    dstenv->env_tf.tf_regs.reg_eax = 0;    // sys_ipc_recv will return 0

    return 0;
}

```

这又是一个非常繁琐的函数。该注意的都在注释里面写清楚了。

还有别忘了 lib/ipc.c 中的 ipc_recv()和 ipc_send()用户库函数。它们是对上面两个系统调用的包装，当然也添加了一些功能。按照代码中的注释提示操作即可，代码如下：

```

int32_t
ipc_recv(env_id_t *from_env_store, void *pg, int *perm_store)
{
    int r;

    if (pg != NULL)
        r = sys_ipc_recv(pg);
    else
        r = sys_ipc_recv((void *)UTOP); // UTOP表示不希望接收页面映射

    struct Env *curenv = (struct Env *) envs + ENVX (sys_getenv_id ());
}

```

```

    if (from_env_store != NULL)
        *from_env_store = (r < 0 ? 0 : curenv->env_ipc_from);
    if (perm_store != NULL)
        *perm_store = (r < 0 ? 0 : curenv->env_ipc_perm);

    if (r < 0)
        return r;
    else
        return curenv->env_ipc_value;
}

void
ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
{
    int r;
    void *srcva = (void *)UTOP;
    if (pg != NULL)
        srcva = pg;

    while ((r = sys_ipc_try_send(to_env, val, srcva, perm)) != 0) {
        if (r == -E_IPC_NOT_RECV)
            sys_yield();
        else
            panic("lib/ipc.c/ipc_send(): sys_ipc_try_send error: %e", r);
    }
    return;
}

```

我们运行一下 user/pingpong.c, 效果如下:

```

QEMU
[00000000] new env 00001005
[00000000] new env 00001006
[00000000] new env 00001007
[00000000] new env 00001008
[00001008] new env 00001009
send 0 from 1008 to 1009
1009 got 0 from 1008
1008 got 1 from 1009
1009 got 2 from 1008
1008 got 3 from 1009
1009 got 4 from 1008
1008 got 5 from 1009
1009 got 6 from 1008
1008 got 7 from 1009
1009 got 8 from 1008
1008 got 9 from 1009
[00001008] exiting gracefully
[00001008] free env 00001008
1009 got 10 from 1008
[00001009] exiting gracefully
[00001009] free env 00001009
No more runnable environments!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>

```

我们运行一下 user/primes.c, 效果如下:

```
eraser@ubuntu: ~/eclipseWorkspace/lab4_finish no challenge
CPU 0: 7937 [000013f3] new env 000013f4
CPU 0: 7949 [000013f4] new env 000013f5
CPU 0: 7951 [000013f5] new env 000013f6
CPU 0: 7963 [000013f6] new env 000013f7
CPU 0: 7993 [000013f7] new env 000013f8
CPU 0: 8009 [000013f8] new env 000013f9
CPU 0: 8011 [000013f9] new env 000013fa
CPU 0: 8017 [000013fa] new env 000013fb
CPU 0: 8039 [000013fb] new env 000013fc
CPU 0: 8053 [000013fc] new env 000013fd
CPU 0: 8059 [000013fd] new env 000013fe
CPU 0: 8069 [000013fe] new env 000013ff
CPU 0: 8081 [000013ff] user panic in <unknown> at lib/fork.c:132: lib/fork.c/fork(): out of environments
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf02aef84 from CPU 0
edi 0xffffffff
esi 0xeebdf34
ebp 0xeebdf20
oesp 0xefbfffdc
ebx 0x008018a3
edx 0xeebdfdc8
```

至此, lab 4 代码写完了, make grade 留念:

```
eraser@ubuntu: ~/eclipseWorkspace/lab4_finish no challenge
dumbfork: OK (1.9s)
Part A score: 5/5

faultread: OK (1.4s)
faultwrite: OK (1.4s)
faultdie: OK (1.4s)
faultregs: OK (1.4s)
faultalloc: OK (1.4s)
faultallocbad: OK (1.4s)
faultnostack: OK (1.4s)
faultbadhandler: OK (1.4s)
faultevilhandler: OK (1.4s)
forktree: OK (1.7s)
Part B score: 50/50

spin: OK (1.5s)
stresssched: OK (2.7s)
pingpong: OK (1.5s)
primes: OK (31.1s)
Part C score: 20/20

Score: 75/75
eraser@ubuntu:~/eclipseWorkspace/lab4_finish no challenge$
```

内容三：遇到的困难以及解决方法

困难一：Debug，还是 Debug

这次实习不如 Lab 3 来得抽象，我遇到的最大的困难就是 Debug。由于这次实习非常具有综合性，涉及 Lab1 ~ Lab3 的代码，因此出现一个 bug 的调试代价是比较大的。不过值得注意的是，虽然涉及前面的代码，但在我的 bug 最终调试出来之后，发现其中的绝大多数是 Lab 4 新写的代码。所以说，对自己之前认真做的东西，还是要有自信啊。

粗心和着急是 bug 的最大来源，如果想要调试时间少，一定要事前想清楚，写的时候专心一点，抄都不要抄错。

在 debug 中我使用得最多的方法就是 printf。就是在程序中打印一些变量的信息，看看程序有没有执行到那里，输出的内容和自己期待的一不一样，一步一步地缩小范围找到症结。

例如，我在测试 user/faultalloc 这个程序时，调试了整整一天。这个程序真的涉及了 Lab1 ~ Lab3 的内容，调试起来真不容易。调不出来，别忘了请教同学。我请教了刘驰和刘智猷大牛，现在 user/faultalloc 的 handler 没有返回，下面是刘智猷大牛的调试过程：

- 1、看看如果不返回，该地址中有没有内容，在 handler 中 cprintf 看看
--->是的，没问题，cpprintf 输出正确结果。那么内存是没有问题的。
- 2、知道哪里出错：user pagefault 在“返回”的时候出了问题。
- 3、是在哪里返回？那个 trick 那里，即 lib/pfentry.S 中的 _pgfault_upcall。
我们先来明确 handler 返回的过程：返回就是恢复 tf 或 utf 到寄存器的过程。在这里是将 utf 的内容恢复到寄存器，一 ret 就从之前的断点重新执行啦。
- 4、在 handler 输出看看 utf->eip,esp，留着等一下对照。
- 5、在 obj/user/faultalloc 中搜索：“_pgfault_upcall:”
- 6、gdb：将断点设置在 call handler 返回之后。
- 7、(gdb) x/16x \$esp，看看栈中的内容对不对
--->需要对着 UTrapFrame 对比，一个 32 位 int 就对应一个 0x00000000
- 8、一步一步 si，一直到 ret 前面
- 9、发现 ret 前 eip 不对!!! 怎么回事？
- 10、额，前面有个 ebx 的地方，写成 ebp 了，真 sb!!!!

我们从上面发现，整个调试的过程是充满严密的逻辑思考和推断的。

此外，我还遇到了这样的 bug：指针使用前忘记判断是否为 NULL，将 UXSTACKTOP 误写成了 USTACKTOP，等等。

困难二：有些函数太麻烦

在这个 lab 中，我们遇到了这样的函数：虽然仅仅是对一些功能的包装，但是函数的描述和实现流程需要进行各种各样的判断和异常处理，让人感觉好“晕”。

对于这种情况，我总结出一个办法，就是把要做的事情一项一项列出来，按照先后顺序做完一项打个勾，这样就可以有条不紊地完成。

困难三：这个 Lab 拖得太久了

恩，又迟交了...T_T。

内容四：收获和感想

知识上，主要是对“进程调度”有了新的认识（特别是强大的 `env_run()` 函数），对各种“栈”有了深刻的体会（以及那 `trick`），对中断流程、系统调用进行了复习和巩固，以及最重要的“写时复制”技术和 `fork()` 创建子进程。

调试能力得到进一步提高。特别是看了刘智猷大牛怎么调代码。

剩下还有很多体会，很受益，不过我就不想多写那些套话了。

最后，我要感谢刘驰同学，他耐心地指导了我很多不明白的地方，经常和我讨论。

我要感谢刘智猷同学，多亏他拨冗和我一起调试程序，百发百中切中要害，大牛就是大牛。

我要感谢素未谋面的张驰学长，和我一直受益匪浅的他的实习报告。

内容五：对课程的意见和建议

（无）

内容六：参考文献

- [1] MIT 6.828 课程资料, MIT, 2011;
- [2] 邵志远, JOS 实验讲义第六章, 华中科技大学, 2010;
- [4] 张驰, 操作系统 JOS 第四次实习报告, 北京大学, 2011;
- [5] <http://www.docin.com/p-243196196.html>