操作系统实验报告

名称：操作系统 Lab4

学　　院：计算机学院　网络空间安全学院

专　　业：计算机科学与技术　信息安全

成　　员：徐晖宇 付宇 顾知晨 姜奕兵 孙铭 肖阳

课程教师：蒲凌君

时　　间：2019 年 12 月 21 日

小组成员分工

徐晖宇（1713666）：作业 12 作业 13

付　宇（1612120）：作业 4　问题 2　作业 5　问题 3

顾知晨（1711323）：作业 8　作业 9

姜奕兵（1710218）：作业 10 作业 11

孙　铭（1711377）：作业 1　问题 1　作业 2　作业 3

肖　阳（1610292）：作业 6　作业 7

make grade:

```
make[1]:正在离开目录 `/home/sunming/Desktop/lab4'
dumbfork: OK (1.5s)
Part A score: 5/5

faultread: OK (1.0s)
faultwrite: OK (0.9s)
faultdie: OK (0.9s)
faultregs: OK (1.0s)
faultalloc: OK (0.9s)
faultallocbad: OK (1.0s)
faultnostack: OK (1.8s)
faultbadhandler: OK (1.1s)
faultevilhandler: OK (1.9s)
forktree: OK (2.2s)
Part B score: 50/50

spin: OK (1.1s)
stresssched: OK (2.3s)
sendpage: OK (1.1s)
pingpong: OK (2.1s)
primes: OK (5.1s)
Part C score: 25/25

Score: 80/80
```

一、 作业 1

作业 **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_pgdir() test, which we will fix soon).

翻译：

阅读 kern/init.c 中的 boot_aps()和 mp_main()，以及在 kern/mpentry.S 中的汇编代码。确保你理解了在 AP 引导过程中的控制流转移。然后修改在 kern/pmap.c 中的、你自己的 page_init()，实现避免在 MPENTRY_PADDR 处添加页到空闲列表上，以便于我们能够在物理地址上安全地复制和运行 AP 引导程序代码。你的代码应该会通过更新后的 check_page_free_list()的测试（但可能会在更新后的 check_kern_pgdir()上测试失败，我们在后面会修复它）。

解答：

根据题目要求，首先阅读 kern/init.c 中的 boot_aps()、mp_main()以及在 kern/mpentry.S 中的汇编代码如下。

kern/init.c/boot_aps()

```
// Start the non-boot (AP) processors.
static void
boot_aps(void)
{
    extern unsigned char mpentry_start[], mpentry_end[];
    void *code;
    struct CpuInfo *c;
    // Write entry code to unused memory at MPENTRY_PADDR
    code = KADDR(MPENTRY_PADDR);
    memmove(code, mpentry_start, mpentry_end - mpentry_start);
    // Boot each AP one at a time
    for (c = cpus; c < cpus + ncpu; c++) {
        if (c == cpus + cpunum())  // We've started already.
            continue;
        // Tell mpentry.S what stack to use
        mpentry_kstack = percpu_kstacks[c - cpus] + KSTKSIZE;
        // Start the CPU at mpentry_start
        lapic_startap(c->cpu_id, PADDR(code));
```

```
        // Wait for the CPU to finish some basic setup in mp_main()
        while(c->cpu_status != CPU_STARTED)
            ;
    }
}
```

启动 APS 之前，引导程序处理器（BPS）应该会首先去收集关于多处理器系统的信息，比如总的 CPU 数、它们的 APIC ID 以及 LAPIC 单元的 MMIO 地址。在 kern/mpconfig.c 中的 mp_init()函数通过读取内存中位于 BIOS 区域里的 MP 配置表来获得这些信息。

函数 boot_aps()（kern/init.c）引导 APS 启动。APS 在实模式启动，和 bootloader 引导过程非常相像（boot/boot.S），boot_aps()复制 AP entry code(kern/mpentry.S)到实模式可寻址到的一处内存地址。但是和 bootloader 不同的是，我们对 AP 在哪执行代码有一些控制，我们 copy the entry code to 0x7000 (MPENTRY_PADDR), but any unused, page-aligned physicaladdress below 640KB would work.

之后，boot_aps()通过发送 STARTUP IPS 到 AP 相对应的 LAPIC 单元激活 AP，同时初始化该 AP 运行它 entry code (MPENTRY_PADDR in our case)的 CS:IP 地址。The entry code（in kern/mpentry.S ）与 boot/boot.S 非常相似。

一些简单的启动步骤之后，它将 A 设置为保护模式，然后调用 C setup routine mp_main()(also in kern/init.c)。boot_aps()收到 the AP 发送 CPU_STARTED flag(in cpu_status field of its struct Cpu)后再去唤醒下一个 AP。

kern/init.c/ mp_main()

```
// Setup code for APs
void
mp_main(void)
{
    // We are in high EIP now, safe to switch to kern_pgdir
    lcr3(PADDR(kern_pgdir));
    cprintf("SMP: CPU %d starting\n", cpunum());

    lapic_init();
    env_init_percpu();
    trap_init_percpu();
    xchg(&thiscpu->cpu_status, CPU_STARTED); // tell boot_aps() we're u
p

    // Now that we have finished some basic setup, call sched_yield()
```

```
    // to start running processes on this CPU.  But make sure that
    // only one CPU can enter the scheduler at a time!
    //
    // Your code here:
        lock_kernel();
        sched_yield();

    // Remove this after you finish Exercise 4
    //for (;;);
}
```

理解上述代码之后，接下来，是对 kern/pmap.c 中的 page_init()函数进行修改，根据题目要求我们明白，实际上就是标记 MPENTRY_PADDR 开始的一个物理页为已使用，只需要在 page_init()中做一个特例处理即可。唯一需要注意的就是确定这个特殊页在哪个区间内。修改代码如下。

```
    size_t i;
    for (i = 0; i < npages; i++) {
        if(i == 0)
            {   pages[i].pp_ref = 1;
                pages[i].pp_link = NULL;
            }
        else if(i == MPENTRY_PADDR/PGSIZE){
                pages[i].pp_ref = 1;
                pages[i].pp_link = NULL;
        }
        else if(i>=1 && i<npages_basemem)
        {
            pages[i].pp_ref = 0;
            pages[i].pp_link = page_free_list;
            page_free_list = &pages[i];
        }
        else if(i>=IOPHYSMEM/PGSIZE && i< EXTPHYSMEM/PGSIZE )
        {
            pages[i].pp_ref = 1;
            pages[i].pp_link = NULL;
        }
        else if( i >= EXTPHYSMEM / PGSIZE &&
                i < ( (int)(boot_alloc(0)) - KERNBASE)/PGSIZE)
        {
            pages[i].pp_ref = 1;
            pages[i].pp_link =NULL;
        }
        else
```

```
        {
            pages[i].pp_ref = 0;
            pages[i].pp_link = page_free_list;
            page_free_list = &pages[i];
        }
    }
```

现在执行 make qemu，观察运行结果如下。

```
root@sunming://home/sunming/Desktop/lab4# make qemu
+ cc kern/init.c
+ ld obj/kern/kernel
+ mk obj/kern/kernel.img
/usr/local/qemu/bin/qemu-system-i386 -hda obj/kern/kernel.img -serial mon:stdio
-gdb tcp::25000 -D qemu.log -smp 1
VNC server running on `127.0.0.1:5900'
6828 decimal is XXX octal!
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
```

可以看到，我们的代码可以通过 check_kern_pgdir()测试。作业 1 完成。

二、 问题 1

> 问题 **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.

翻译：

比较 kern/mpentry.S 和 boot/boot.S。记住，那个 kern/mpentry.S 是编译和链接后的，运行在 KERNBASE 上面的，就像内核中的其它程序一样，宏 MPBOOTPHYS 的作用是什么？为什么它需要在 kern/mpentry.S 中，而不是在 boot/boot.S 中？换句话说，如果在 kern/mpentry.S 中删掉它，会发生什么错误？提示：回顾链接地址和加载地址的区别，我们在实验 1 中讨论过它们。

解答：

注意到 kern/mpentry.S 中有这样一段话，可以说明 kern/mpentry.S 和 boot/boot.S 的区别。

```
# This code is similar to boot/boot.S except that
```

```
#    - it does not need to enable A20
#    - it uses MPBOOTPHYS to calculate absolute addresses of its
#      symbols, rather than relying on the linker to fill them
```

这段话指出，kern/mpentry.S 不需要启用 A20，且使用 MPBOOTPHYS 来计算其符号的绝对地址，而不是依赖链接器去填充绝对地址。

关于 MPBOOTPHYS 宏的作用，kern/mpentry.S 是运行在 KERNBASE 之上的，与其他的内核代码一样。也就是说，类似于 mpentry_start, mpentry_end, start32 这类地址，都位于 0xf0000000 之上，显然，实模式是无法寻址的。再仔细看 MPBOOTPHYS 的定义：

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

其意义可以表示为，从 mpentry_start 到 MPENTRY_PADDR 建立映射，将 mpentry_start + offset 地址转为 MPENTRY_PADDR + offset 地址。查看 kern/init.c，发现已经完成了这部分地址的内容拷贝：

```
// Start the non-boot (AP) processors.
static void
boot_aps(void)
{
    extern unsigned char mpentry_start[], mpentry_end[];
    void *code;
    struct CpuInfo *c;
    // Write entry code to unused memory at MPENTRY_PADDR
    code = KADDR(MPENTRY_PADDR);
    memmove(code, mpentry_start, mpentry_end - mpentry_start);
    // Boot each AP one at a time
    for (c = cpus; c < cpus + ncpu; c++) {
        if (c == cpus + cpunum())  // We've started already.
            continue;
        // Tell mpentry.S what stack to use
        mpentry_kstack = percpu_kstacks[c - cpus] + KSTKSIZE;
        // Start the CPU at mpentry_start
        lapic_startap(c->cpu_id, PADDR(code));
        // Wait for the CPU to finish some basic setup in mp_main()
        while(c->cpu_status != CPU_STARTED)
            ;
    }
}
```

因此，实模式下就可以通过 MPBOOTPHYS 宏的转换，运行这部分代码。boot.S 中不需要这个转换是因为代码的本来就被加载在实模式可以寻址的地方。

三、 作业 2

作业 **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().

翻译：

修改 mem_init_mp()（在 kern/pmap.c 中）去映射每个 CPU 的栈从 KSTACKTOP 处开始，就像在 inc/memlayout.h 中展示的那样。每个栈的大小是 KSTKSIZE 字节加上未映射的保护页 KSTKGAP 的字节。你的代码应该会通过在 check_kern_pgdir()中的新的检查。

解答：

根据题意，最初，只 map 了 BSP，这次需要 map 所有的 cpu（包括实际不存在的）。在 kern/cpu.h 中可以找到对 NCPU 以及全局变量 percpu_kstacks 的声明。

```
// Maximum number of CPUs
#define NCPU  8
… …
// Per-CPU kernel stacks
extern unsigned char percpu_kstacks[NCPU][KSTKSIZE];
```

percpu_kstacks 的定义在 kern/mpconfig.c 中可以找到：

```
// Per-CPU kernel stacks
unsigned char percpu_kstacks[NCPU][KSTKSIZE]
__attribute__ ((aligned(PGSIZE)));
```

接下来，解决该问题的思路就是修改 kern/pmap.c 中的 mem_init_mp()函数，代码如下。

```
// LAB 4: Your code here:
    int i = 0;
    uintptr_t kstacktop_i;
    for (i = 0; i < NCPU; i++)
    {
        kstacktop_i = KSTACKTOP - i * (KSTKSIZE + KSTKGAP);
        //boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, phys
addr_t pa, int perm)
        boot_map_region(kern_pgdir,
                        kstacktop_i - KSTKSIZE, //由物理内存支持
                        ROUNDUP(KSTKSIZE, PGSIZE),
                        PADDR(&percpu_kstacks[i]),
                        PTE_W);
```

```
    }
```

此代码含义是，从 KSTACKTOP 开始 map 每 CPU 堆栈，最多可 map“NCPU”个 CPU。对每个 CPU，使用 percpu_kstacks[i]所指的物理内存作为其内核堆栈。CPU 的内核堆栈从虚拟地址 kstacktop_i = KSTACKTOP - i * (KSTKSIZE + KSTKGAP)开始增长，并分成两部分，就像在 mem_init()中设置的单个堆栈一样。

至此，作业 2 完成。

四、 作业 3

作业 **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.)

翻译：

在 trap_init_percpu()（在 kern/trap.c 文件中）的代码为 BSP 初始化 TSS 和 TSS 描述符。在实验 3 中它就运行过，但是当它运行在其它的 CPU 上就会出错。修改这些代码以便它能在所有 CPU 上都正常运行。（注意：你的新代码应该还不能使用全局变量 ts）

解答：

根据题目要求，在 kern/trap.c 文件下找到了全局变量的声明，将其注释掉。

```
static struct Taskstate ts;
```

之后再根据单个 CPU 代码做改动，在 inc/memlayout.h 中可以找到 GD_TSS0 的定义如下。

```
// Global descriptor numbers
#define GD_KT     0x08     // kernel text
#define GD_KD     0x10     // kernel data
#define GD_UT     0x18     // user text
#define GD_UD     0x20     // user data
#define GD_TSS0   0x28     // Task segment selector for CPU 0
```

但是题目中以及代码中并没有其他位置说明 CPU 的任务段选择器在哪里。因此最大的难点就是找到这个值。实际上，偏移量为 cpu_id<<3。

修改 trap_init_percpu()如下。

```
// LAB 4: Your code here:
    int cpu_id = thiscpu->cpu_id;
    cprintf("cpu_id == %d\n",cpu_id );
```

```
    thiscpu->cpu_ts.ts_esp0 = KSTACKTOP - cpu_id*( KSTKSIZE  + KSTKGAP)
;
    thiscpu->cpu_ts.ts_ss0 = GD_KD;
    gdt[ (GD_TSS0 >> 3) + cpu_id] = SEG16(STS_T32A, (uint32_t) (& (this
cpu->cpu_ts) ),
                    sizeof(struct Taskstate) - 1, 0);
    gdt[(GD_TSS0 >> 3) + cpu_id].sd_s = 0;
    ltr(GD_TSS0 + 8*cpu_id);
    lidt(&idt_pd);
```

代码中，宏“thiscpu”总是指当前 cpu 的结构 CpuInfo。当前 cpu 的标识由 CPU()或该 CPU-> CPU _标识给出；使用“此 cpu->cpu_ts”作为当前 cpu 的 TSS，而不是全局“ts”变量；使用 gdt[(GD_TSS0 >> 3) + i]作为中央处理器 I 的 TSS 描述符；在内存初始化 mp()中映射了每个 CPU 的内核堆栈；此外，ltr 在 TSS 选择器中设置了一个“忙碌”标志，如果不小心在多个 CPU 上加载了相同的 TSS，将会遇到三重故障。如果设置了一个单独的中央处理器的 TSS 错误，则可能不会得到一个错误，直到尝试从该中央处理器的用户空间返回。

代码实现后，make qemu 编译。

```
/usr/local/qemu/bin/qemu-system-i386 -hda obj/kern/kernel.img -serial mon:stdio
-gdb tcp::25000 -D qemu.log -smp 1
VNC server running on `127.0.0.1:5900'
6828 decimal is XXX octal!
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
cpu_id == 0
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2
[00000000] new env 00001000
[00000000] new env 00001001
[00000000] new env 00001002
Hello, I am environment 00001000.
Hello, I am environment 00001001.
Hello, I am environment 00001002.
Back in environment 00001000, iteration 0.
Back in environment 00001001, iteration 0.
Back in environment 00001002, iteration 0.
Back in environment 00001000, iteration 1.
Back in environment 00001001, iteration 1.
Back in environment 00001002, iteration 1.
Back in environment 00001000, iteration 2.
Back in environment 00001001, iteration 2.
Back in environment 00001002, iteration 2.
Back in environment 00001000, iteration 3.
Back in environment 00001001, iteration 3.
Back in environment 00001002, iteration 3.
Back in environment 00001001, iteration 4.
```

即可看到正确的运行结果。至此，作业三完成。

五、 作业 4

- `i386_init()` 中，在 BSP 唤醒其他 CPU 之前获得内核锁
- `mp_main()` 中，在初始化完 AP 后获得内核锁，接着调用 `sched_yield()` 来开始在这个 AP 上运行进程。
- `trap()` 中，从用户模式陷入(trap into)内核模式之前获得锁。你可以通过检查 `tf_cs` 的低位判断这一 trap 发生在用户模式还是内核模式（译注：Lab 3 中曾经使用过这一检查）
- env_run() 中，恰好在 **回到用户进程之前** 释放内核锁。不要太早或太晚做这件事，否则可能会出现竞争或死锁。

作业 **4**

Apply the big kernel lock as described above, by calling lock_kernel() andunlock_kernel() at the proper locations.

```
void
i386_init(void)
{
        extern char edata[], end[];
        // Before doing anything else, complete the ELF loading process.
        // Clear the uninitialized global data (BSS) section of our program.
        // This ensures that all static/global variables start out zero.
        memset(edata, 0, end - edata);
        // Initialize the console.
        // Can't call cprintf until after we do this!
        cons_init();
        cprintf("6828 decimal is %o octal!\n", 6828);
        // Lab 2 memory management initialization functions
        mem_init();
        // Lab 3 user environment initialization functions
        env_init();
        trap_init();
        // Lab 4 multiprocessor initialization functions
        mp_init();
        lapic_init();
        // Lab 4 multitasking initialization functions
        pic_init();
        // Acquire the big kernel lock before waking up APs
        // Your code here:
        //在 BSP 唤醒其他 CPU 之前获得内核锁
        lock_kernel();
        // Starting non-boot CPUs
        boot_aps();
```

（1） boot_aps()函数的调用如上注释所述：启动未唤醒的 CPUs。根据题目要求，在 BSP 唤醒其他 CPU 之前获得内核锁，即在 boot_aps()函数调用前加入 lock_kernel()函数。

```
void
mp_main(void)
{
        // We are in high EIP now, safe to switch to kern_pgdir
        lcr3(PADDR(kern_pgdir));
        cprintf("SMP: CPU %d starting\n", cpunum());

        lapic_init();
        env_init_percpu();
        trap_init_percpu();
        xchg(&thiscpu->cpu_status, CPU_STARTED); // tell boot_aps() we're up

        // Now that we have finished some basic setup, call sched_yield()
        // to start running processes on this CPU.  But make sure that
        // only one CPU can enter the scheduler at a time!
        //
        // Your code here:
        //在初始化完 AP 后获得内核锁，接着调用 sched_yield() 来开始在这个 AP 上运行进程。
        lock_kernel();
        sched_yield();

        // Remove this after you finish Exercise 4
        //for (;;);
}
```

（2） 在 mp_main()中初始化完应用处理器（AP）后获得内核锁，因此将 lock_kernel()附在该代码块末尾，按照题目要求接着调用 sched_yield()函数。

```
if ((tf->tf_cs & 3) == 3) {
        // Trapped from user mode.
        // Acquire the big kernel lock before doing any
        // serious kernel work.
        // LAB 4: Your code here.如果是从用户态进入到内核态的话，需要获得锁
        lock_kernel();
        assert(curenv);
```

（3） 在 trap.c 的 trap(struct Trapframe *tf)函数中找到检查 tf_cs 的低位判断的条件语句，语句块中填写 lock_kernel()，保证在用户模式陷入内核模式之前获得锁 。

```
        // Step 2: Use env_pop_tf() to restore the environment's
        //         registers and drop into user mode in the
        //         environment.
```

```
        //在env_pop_tf执行结束之后，就回到用户态了，所以一定要在此之前释放
        unlock_kernel();
        env_pop_tf(& (curenv->env_tf) );
        panic("env_run not yet implemented");
```

（4） 在 env.c 的 env_run(struct Env *e)函数中找到调用 env_pop_tf()函数的位置（step 2 中提到调用该函数后回到用户态），而我们要恰好在回到用户进程之前释放内核锁。因此，将 unlock_kernel()调用放在 env_pop_tf()调用的前面。

六、 问题 2

问题 **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.

因为不同的内核栈上可能保存有不同的信息，在一个 CPU 从内核退出来之后，有可能在内核栈中留下了一些将来还会用的数据，所以一定要有单独的栈来保存这些信息。例如，在某进程即将陷入内核态的时候（尚未获得锁），其实在 trap() 函数之前已经在 trapentry.S 中对内核栈进行了操作，压入了寄存器信息。如果共用一个内核栈，那显然会导致信息错误。

七、 作业 5

作业 **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.

```
	// Search through 'envs' for an ENV_RUNNABLE environment in
	// circular fashion starting just after the env this CPU was
	// last running.  Switch to the first such environment found.
	//
	// If no envs are runnable, but the environment previously
	// running on this CPU is still ENV_RUNNING, it's okay to
	// choose that environment.
	//
	// Never choose an environment that's currently running on
	// another CPU (env_status == ENV_RUNNING). If there are
	// no runnable environments, simply drop through to the code
	// below to halt the cpu.
```

```
	// LAB 4: Your code here.
	idle = curenv;
	size_t idx;
	if(idle!=NULL){
		idx=ENVX(idle->env_id);
	}else{
		idx=-1;
	}
	//在最后一次运行该CPU的env之后，以循环方式在“ envs”中搜索ENV_RUNNABLE环境。 切换到找到的第一个这样的环境。
	for (size_t i=0; i<NENV; i++) {
		idx = (idx+1 == NENV) ? 0:idx+1;
		if (envs[idx].env_status == ENV_RUNNABLE) {
			env_run(&envs[idx]);
			return;
		}
	}
	//如果没有可运行的环境，但是以前在此CPU上运行的环境仍为ENV_RUNNING，则可以选择该环境。
	if (idle && idle->env_status == ENV_RUNNING) {
		env_run(idle);
		return;
	}
	// sched_halt never returns
	sched_halt();
```

```
extern struct Env *envs;		// All environments
#define curenv (thiscpu->cpu_env)	// Current environment
extern struct Segdesc gdt[];
```

curenv 在 env.h 中被宏定义为 thiscpu->cpu_env，即当前 cpu 下的 cpu 运行环境。根据注释提示，先让 struct Env*类型的 idle 指向 curenv，若当前 cpu 下的不存在运行环境，则 idx=-1，若存在，则用 ENVX(idle->env_id)根据 id 取出该环境对应的索引号赋值给 idx。遍历所有进程环境，环境数量为 NENV，如果 idx=-1 或 NENV，则从索引为 0 开始遍历是否有 ENV_RUNNABLE 的环境，利用 env_run()切换到第一个这样的环境；若没有，则看 curenv 下环境状态是否为 ENV_RUNNING，是的话则 env_run()该环境，否则的话中止 cpu。

```
#if defined(TEST)
	// Don't touch -- used by grading script!
	ENV_CREATE(TEST, ENV_TYPE_USER);
#else
	// Touch all you want.
	ENV_CREATE(user_yield, ENV_TYPE_USER);
	ENV_CREATE(user_yield, ENV_TYPE_USER);
	ENV_CREATE(user_yield, ENV_TYPE_USER);

#endif
```

在 init.c 中创建三个环境。

```
	int ret = 0;
	switch(syscallno){
		case SYS_cputs:			sys_cputs( (const char *)a1, (size_t) a2);
						break;
		case SYS_cgetc:			ret = sys_cgetc();
						break;
		case SYS_getenvid:		ret =sys_getenvid();
						break;
		case SYS_env_destroy:		ret= sys_env_destroy(a1);
						break;
		case SYS_yield:			sys_yield();
						break;
```

在 syscall.c 中添加 case SYS_yield。

make qemu 运行结果：

```
Hello, I am environment 00001000.
Hello, I am environment 00001001.
Hello, I am environment 00001002.
Back in environment 00001000, iteration 0.
Back in environment 00001001, iteration 0.
Back in environment 00001002, iteration 0.
Back in environment 00001000, iteration 1.
Back in environment 00001001, iteration 1.
Back in environment 00001002, iteration 1.
Back in environment 00001000, iteration 2.
Back in environment 00001001, iteration 2.
Back in environment 00001002, iteration 2.
Back in environment 00001000, iteration 3.
Back in environment 00001001, iteration 3.
Back in environment 00001002, iteration 3.
Back in environment 00001000, iteration 4.
```

八、 问题 3

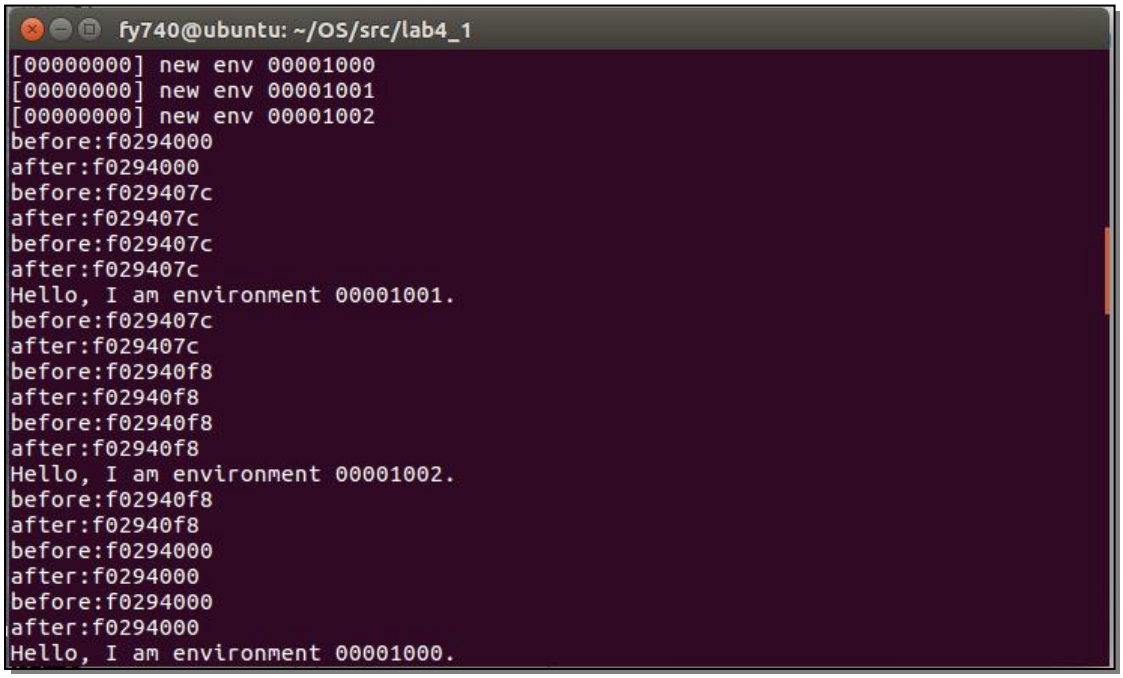
问题 **3**

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?

为了了解 e 的地址是否变化，我们在调用 lcr3()前后打印 e 的地址信息，如下为增添的代码：

```
curenv->env_runs++;
cprintf("before:%08x\n", e);
lcr3( PADDR(curenv->env_pgdir) );
cprintf("after:%08x\n", e);
//在env_pop_tf执行结束之后，就回到用户态了，所以一定要在此之前释放
```

打印结果如下：



可以看出 e 的地址在装载 %cr3 寄存器之后依然没有改变。我们知道 KERNBASE 地址为 f0000000，并且在这以上都是系统区，是映射至同一片物理内存区域的，故切换页表的前后 e 是不会受到影响的。

九、 作业 6

> 作业 **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_INVAL in that case. Test your JOS kernel with user/dumbfork and make sure it works before proceeding.

sys_exofork：该系统调用创建一个几乎完全空白的新进程：它的用户地址空间没有内存映射，也不可以运行。这个新的进程拥有和创建它的父进程（调用这一方法的进程）一样的寄存器状态。在父进程中，sys_exofork 会返回刚刚创建的新进程的 envid_t（或者一个负的错误代码，如果进程分配失败）。在子进程中，它应当返回 0。

```
static envid_t
sys_exofork(void)
{
    // Create the new environment with env_alloc(), from kern/env.c.
    // It should be left as env_alloc created it, except that
    // status is set to ENV_NOT_RUNNABLE, and the register set is copied
    // from the current environment -- but tweaked so sys_exofork
    // will appear to return 0.

    // LAB 4: Your code here.
    struct Env * child_env;//子进程的指针
    int error_code;//错误代码
    if ((error_code = env_alloc(&child_env, curenv->env_id)) < 0)//调用env_alloc函数创建一个子进程
        return error_code;//创建失败，返回错误代码
    child_env->env_tf = curenv->env_tf;//复制中断保护现场
    child_env->env_status = ENV_NOT_RUNNABLE;//设置子进程的运行状态
    child_env->env_tf.tf_regs.reg_eax = 0;//修改子进程中断保护现场的eax寄存器，让子进程返回0
    return child_env->env_id;
}
```

sys_env_set_status：将一个进程的状态设置为 ENV_RUNNABLE 或 ENV_NOT_RUNNABLE。这个系统调用通常用来在新创建的进程的地址空间和寄存器状态已经初始化完毕后将它标记为就绪状态。

开启 envid2env 的检查选项，检查发起系统调用的进程是否有权限给 envid 的进程设置状态（同时检查进程是否存在）。

```
static int
sys_env_set_status(envid_t envid, int status)
{
    // Hint: Use the 'envid2env' function from kern/env.c to translate an
    // envid to a struct Env.
    // You should set envid2env's third argument to 1, which will
    // check whether the current environment has permission to set
    // envid's status.

    // LAB 4: Your code here.
    struct Env *env_store;//进程指针
    int errorcode;//错误代码
    if ((errorcode = envid2env(envid, &env_store, 1)) < 0)//检查当前进程是否有权限给进程设置运行状态，并判断进程是否存在
        return errorcode;
    if (status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE)//只能设置两种状态
        return -E_INVAL;
    env_store->env_status = status;//修改进程的运行状态
    return 0;
}
```

sys_page_alloc：分配一个物理内存页面，并将它映射在给定进程虚拟地址空间的给定虚拟地址上。

```
static int
sys_page_alloc(envid_t envid, void *va, int perm)
{
    // Hint: This function is a wrapper around page_alloc() and
    //   page_insert() from kern/pmap.c.
    //   Most of the new code you write should be to check the
    //   parameters for correctness.
    //   If page_insert() fails, remember to free the page you
    //   allocated!

    // LAB 4: Your code here.
    struct Env *env_store;//进程指针
    struct PageInfo * ppg;//页指针
    int flag = PTE_P | PTE_U;//权限标志位
    if (envid2env(envid, &env_store, 1) < 0)//检查当前进程是否有权限给进程设置运行状态，并判断进程是否存在
        return -E_BAD_ENV;
    if (((uintptr_t)va >= UTOP) || PGOFF(va) || ((perm & flag) != flag))//判断虚拟地址空间是否合法，权限是否符合条件
        return -E_INVAL;
    if (perm & ~(PTE_SYSCALL))//检查是否有不符合条件的权限
        return -E_INVAL;
    if ((ppg = page_alloc(ALLOC_ZERO)) == NULL)//调用page_alloc函数分配一个物理页
        return -E_NO_MEM;
    if (page_insert(env_store->env_pgdir, ppg, va, perm) < 0) {//将分配的物理页插入到进程的页表中
        page_free(ppg);//插入失败则释放物理页
        return -E_NO_MEM;
    }
    return 0;
}
```

sys_page_map：从一个进程的地址空间拷贝一个页的映射（**不是** 页的内容）到另

一个进程的地址空间，新进程和旧进程的映射应当指向同一个物理内存区域，使两个进程得以共享内存。

```
static int
sys_page_map(envid_t srcenvid, void *srcva,
         envid_t dstenvid, void *dstva, int perm)
{
    // Hint: This function is a wrapper around page_lookup() and
    //   page_insert() from kern/pmap.c.
    //   Again, most of the new code you write should be to check the
    //   parameters for correctness.
    //   Use the third argument to page_lookup() to
    //   check the current permissions on the page.

    // LAB 4: Your code here.
    struct Env *env_src, *env_dst;//源进程，目的进程
    struct PageInfo * ppg;//共享页
    pte_t *ppte;
    int errorcode;//错误代码
    int flag = PTE_P | PTE_U;//权限标志位
    if ((envid2env(srcenvid, &env_src, 1) < 0) || (envid2env(dstenvid, &env_dst, 1) < 0))//检查当前进程是否有权限，并判断进程是否存在
        return -E_BAD_ENV;
    if (((uintptr_t)srcva >= UTOP) || PGOFF(srcva) || ((uintptr_t)dstva >= UTOP) || PGOFF(dstva))//检查虚拟地址的范围，是否页对齐
        return -E_INVAL;


    if ((ppg = page_lookup(env_src->env_pgdir, srcva, &ppte)) == NULL)//调用page_lookup函数找到源进程虚拟地址对应的物理页
        return -E_INVAL;
    if ((perm & ~(PTE_SYSCALL)) || ((perm & flag) != flag))//权限要求同上
        return -E_INVAL;
    if ((perm & PTE_W) && !(*ppte & PTE_W))//如果权限要求可写，那么物理页的权限也必须是可写的
        return -E_INVAL;
    if (page_insert(env_dst->env_pgdir, ppg, dstva, perm) < 0) //将该物理页插入目的进程的页表中
        return -E_NO_MEM;
    return 0;
}
```

sys_page_unmap：取消给定进程在给定虚拟地址的页映射。

```
static int
sys_page_unmap(envid_t envid, void *va)
{
    // Hint: This function is a wrapper around page_remove().

    // LAB 4: Your code here.
    struct Env *env_store;//进程指针
    struct PageInfo *pp;//页指针
    if (envid2env(envid, &env_store, 1) < 0)//检查当前进程是否有权限，并判断进程是否存在
        return -E_BAD_ENV;
    if (((uintptr_t)va >= UTOP) || PGOFF(va))//检查虚拟地址的范围，是否页对齐
        return -E_INVAL;
    page_remove(env_store->env_pgdir, va);//调用page_remove函数删除映射关系
    return 0;
}
```

十、 作业 7

> 作业 7
>
> 实现系统调用 sys_env_set_pgfault_upcall。注意在寻找目标进程的 ID 时进行权限检查，因为这个系统调用是比较“危险”的。

为了处理自己的缺页中断，用户环境需要在 JOS 内核中注册缺页中断处理程序的入口。用户环境通过 sys_env_set_pgfault_upcall 系统调用注册它的缺页中断入口。我们在 Env 结构体中增加了一个新成员 env_pgfault_upcall 来记录这一信息。

```
static int
sys_env_set_pgfault_upcall(envid_t envid, void *func)
{
    // LAB 4: Your code here.
    struct Env *env_store;//进程指针
    if (envid2env(envid, &env_store, 1) < 0)//检查当前进程是否有权限，并判断进程是否存在
        return -E_BAD_ENV;
    env_store->env_pgfault_upcall = func;//将进程的缺页处理函数设置为对应的func
    return 0;
}
```

十一、 作业 8

作业 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?)

此处需要帮助用户进程切换栈，压入保存的数据，并启动用户态处理函数。如果出现异常栈溢出则直接结束程序。注意检查用户页错误处理函数是否注册，用户异常栈是否分配，是否溢出，是否页错误嵌套。并注意给新的 UTrapframe 和嵌套返回值保留空间。

判断如果是内核态的缺页，则在此函数中不作处理进入内核态处理，如果这个缺页错误的地址不在异常堆中，那么我们直接在该堆中开辟一块空间， 存储该缺页信息发生时的寄存器信息，如果这个缺页错误的地址在栈中，那么我们开辟一块新的空间存储若有权限，则复制寄存器信息，并保存 fault_va,这是导致缺页错误的虚拟地址，复制记录异常缺页过程的结构体内部信息，之后回到当前用户态进程。

具体代码实现：

```
void
page_fault_handler(struct Trapframe *tf)
{
    uint32_t fault_va;

    // Read processor's CR2 register to find the faulting address
    fault_va = rcr2();
    //返回一个错误的地址
    // Handle kernel-mode page faults.

    // LAB 3: Your code here.
    if(tf->tf_cs == GD_KT)
    //判断如果是内核态的缺页，则在此函数中不作处理进入内核态
        panic("page fault happens in the kern mode");

    // We've already handled kernel-mode exceptions, so if we get here,
    // the page fault happened in user mode.

    // Call the environment's page fault upcall, if one exists.  Set up
 a
    // page fault stack frame on the user exception stack (below
```

```
    // UXSTACKTOP), then branch to curenv->env_pgfault_upcall.
    //
    // The page fault upcall might cause another page fault, in which c
ase
    // we branch to the page fault upcall recursively, pushing another
    // page fault stack frame on top of the user exception stack.
    //
    // The trap handler needs one word of scratch space at the top of t
he
    // trap-time stack in order to return.  In the non-recursive case,
we
    // don't have to worry about this because the top of the regular us
er
    // stack is free.  In the recursive case, this means we have to lea
ve
    // an extra word between the current top of the exception stack and
    // the new stack frame because the exception stack _is_ the trap-ti
me
    // stack.
    //
    // If there's no page fault upcall, the environment didn't allocate
 a
    // page for its exception stack or can't write to it, or the except
ion
    // stack overflows, then destroy the environment that caused the fa
ult.
    // Note that the grade script assumes you will first check for the
page
    // fault upcall and print the "user fault va" message below if ther
e is
    // none.  The remaining three checks can be combined into a single
test.
    //
    // Hints:
    //   user_mem_assert() and env_run() are useful here.
    //   To change what the user environment runs, modify 'curenv->env_
tf'
    //   (the 'tf' variable points at 'curenv->env_tf').

    // LAB 4: Your code here.
    if(!curenv->env_pgfault_upcall){
        // Destroy the environment that caused the fault.
       //判断目前是否有此缺页处理函数，此时摧毁该用户进程。
        cprintf("[%08x] user fault va %08x ip %08x\n",
```

```
            curenv->env_id, fault_va, tf->tf_eip);
        print_trapframe(tf);
        env_destroy(curenv);
    }
    /************debug code*
    cprintf("the curenv->eid =  %d\n",curenv->env_id );

    *******debug code*******/

    unsigned int newEsp=0;
    struct UTrapframe UT;
       //the Exception has not been built
    if( tf->tf_esp < UXSTACKTOP-PGSIZE || tf->tf_esp >= UXSTACKTOP) {
         // 如果这个缺页错误的地址不在异常堆中，那么我们直接在该堆中开辟一块空
         //间，  存储该缺页信息发生时的寄存器信息
        newEsp = UXSTACKTOP - sizeof(struct UTrapframe);
    }
    else
        //note: it is not like the requirement!!! there is two block
        newEsp = tf->tf_esp - sizeof(struct UTrapframe) -8;
       // 如果这个缺页错误的地址在栈中，那么我们开辟一块新的空间存储
       //因为不能够在发生错误的堆栈上进行存储操作
    user_mem_assert(curenv, (void*)newEsp, 0, PTE_U|PTE_W|PTE_P);
    //查询权限，看一看，该用户这一段有无权限。
    //若有权限，则复制寄存器信息，并保存 fault_va,这是导致缺页错误的虚拟地址。
    //
    UT.utf_err = tf->tf_err;
    UT.utf_regs = tf->tf_regs;
    UT.utf_eflags = tf->tf_eflags;
    UT.utf_eip = tf->tf_eip;
    UT.utf_esp = tf->tf_esp;
    UT.utf_fault_va = fault_va;

    user_mem_assert(curenv,(void*)newEsp, sizeof(struct UTrapframe),PTE
_U|PTE_P|PTE_W );
    //同上查看在这一结构体整体有无权限进行读写修改
    memcpy((void*)newEsp, (&UT) ,sizeof(struct UTrapframe));
    //复制记录异常缺页过程的结构体内部信息
    tf->tf_esp = newEsp;//将该新开的地址赋给 page-fault.
    tf->tf_eip = (uintptr_t)curenv->env_pgfault_upcall;
    env_run(curenv);
    //回到当前用户态进程
}
```

十二、 作业 9

用户态，通过系统调用注册函数。并在第一次时给异常栈申请空间。没有足够内存在栈中为其开辟空间时,直接 panic，若有条件，则完成注册，若申请请求本身出现缺页问题，则直接 panic.

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

    if (_pgfault_handler == 0) {
        // First time through!
        // LAB 4: Your code here.
        void* addr = (void*) (UXSTACKTOP-PGSIZE);
        r=sys_page_alloc(thisenv->env_id, addr, PTE_W|PTE_U|PTE_P);
       //分配地址空间，给予其读写访问权限
       //P 允许访问，R/ W 位为读写
        if( r < 0)
            panic("No memory for the UxStack, the mistake is %d\n",r);
         //此处为没有足够内存在栈中为其开辟空间时,直接 panic
        //panic("set_pgfault_handler not implemented");
    }

    // Save handler pointer for assembly to call.
    //
    _pgfault_handler = handler;
    //若有条件，则完成注册，
    if(( r= sys_env_set_pgfault_upcall(sys_getenvid(), _pgfault_upcall))
<0)
        panic("sys_env_set_pgfault_upcall is not right %d\n", r);
   //若申请请求本身出现缺页问题，则直接 panic

}
```

十三、 作业 10

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.

pgfault()：pgfault_upcall 中调用的页错误处理函数。在调用之前，父子进程的页错误地址都引用同一页物理内存，该函数作用就是分配一个物理页面使得两者独立。首先分配一个页面，映射到交换区 PFTEMP 这个虚拟地址，然后通过 memmove() 函数将 addr 所在页面拷贝至 PFTEMP，此时有两个物理页保存了同样的内容。再将 addr 也映射到 PFTEMP 对应的物理页，最后解除了 PFTEMP 的映射，此时就只有 addr 指向新分配的物理页了，如此就完成了错误处理。

```
static void

pgfault(struct UTrapframe *utf)

{

   void *addr = (void *)utf->utf_fault_va;

   uint32_t err = utf->utf_err;

   int r;

   // 要求 write 权限，否则报错

   if ((err & FEC_WR) == 0)

   {

      cprintf("The va is 0x%x¥n", (int)addr);

      cprintf("The Eip is 0x%x¥n", utf->utf_eip);

      panic("The err is not right of the pgfault¥n ");

   }

   pte_t PTE = uvpt[PGNUM(addr)];

   // 要求是 copy-on-write 页面，否则报错

   if ((PTE & PTE_COW) == 0)

      panic("The pgfault perm is not right¥n");
```

```
    // 分配临时页面
    if (sys_page_alloc(sys_getenvid(), (void *)PFTEMP, PTE_U | PTE_W | PTE_P) < 0)
        panic("pgfault sys_page_alloc is not right¥n");
    // 赋值数据
    addr = ROUNDDOWN(addr, PGSIZE);
    memcpy((void *)PFTEMP, addr, PGSIZE);
    // addr 所在页面内存复制到临时虚拟页对应的物理页上
    if ((r = sys_page_map(sys_getenvid(), (void *)PFTEMP, sys_getenvid(), addr, PTE_U | PTE_W |
PTE_P)) < 0)
        panic("The sys_page_map is not right, the errno is %d¥n", r);
    // 释放临时虚拟页
    if ((r = sys_page_unmap(sys_getenvid(), (void *)PFTEMP)) < 0)
        panic("The sys_page_unmap is not right, the errno is %d¥n", r);
    return;
}
```

fork() ：设置 page fault handler，即 page fault upcall 调用的函数；duppage 的范围不同，fork() 不需要复制内核区域的映射；为子进程设置 page fault upcall，因为 sys_exofork() 并不会复制父进程的 e->env_pgfault_upcall 给子进程。fork() 函数的流程题目中已给出，按照流程代码实现如下：

```
envid_t
fork(void)
{
    // LAB 4: Your code here.
    extern void *_pgfault_upcall();
```

```
// 缺页处理函数设置
set_pgfault_handler(pgfault);
int childEid = sys_exofork();
if (childEid < 0)
    panic("sys_exofork() error, the errno is  %d¥n", childEid);
if (childEid == 0)
{
    thisenv = &envs[ENVX(sys_getenvid())];
    return childEid;
}
int r = sys_env_set_pgfault_upcall(childEid, _pgfault_upcall);
if (r < 0)
    panic("sys_env_set_pgfault_upcall error , the errno is %d¥n", r);
int pn = 0;
for (pn = 0; pn * PGSIZE < UTOP; pn++)
{
    if (((uvpd[PDX(pn * PGSIZE)] & PTE_P) != 0) &&
        ((uvpt[PGNUM(pn * PGSIZE)] & PTE_U) != 0) &&
        ((uvpt[PGNUM(pn * PGSIZE)] & PTE_P) != 0))
    {
        // 为进程构造堆栈
        if (pn * PGSIZE == UXSTACKTOP - PGSIZE)
            sys_page_alloc(childEid, (void *)(pn * PGSIZE), PTE_U | PTE_W | PTE_P);
        else
            r = duppage(childEid, pn);
        if (r < 0)
            panic("fork() is wrong, the errno is %d¥n", r);
    }
}
```

```
    // 设置子进程正常可运行
    if (sys_env_set_status(childEid, ENV_RUNNABLE) < 0)
        panic("sys_env_set_status");
    return childEid;
}
```

duppage()：复制父、子进程的页面映射。因为 sys_page_map() 页面的权限有要求，所以要修正一下权限，即将所有的 writable 或 copy-on-write 权限页面设置为 copy-on-write。

```
static int
duppage(envid_t envid, unsigned pn)
{
    int r;
    pte_t PTE = uvpt[PGNUM(pn * PGSIZE)];
    // 修改权限位
    int perm = PTE_U | PTE_P;
    if ((PTE & PTE_W) || (PTE & PTE_COW))
        perm |= PTE_COW;
    // envid 进程
    if ((r = sys_page_map(sys_getenvid(), (void *)(pn * PGSIZE), envid, (void *)(pn
 * PGSIZE), perm)) < 0)
        return r;
    // 当前进程
    if ((r = sys_page_map(sys_getenvid(), (void *)(pn * PGSIZE), sys_getenvid(), (v
oid *)(pn * PGSIZE), perm)) < 0)
```

```
    return r;
  return 0;
}
```

十四、 作业 11

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.

可以找到关于中断 IRQ 的一些宏定义：(inc\trap.h)

```
    // Hardware IRQ numbers. We receive these as (IRQ_OFFSET+IRQ_WHATEVER)
#define IRQ_TIMER       0
#define IRQ_KBD         1
#define IRQ_SERIAL      4
#define IRQ_SPURIOUS    7
#define IRQ_IDE         14
#define IRQ_ERROR       19
```

首先声明处理函数，之后使用 SETEGATE 设置表项。

```
  //LAB4
  void handlerIRQ0();
  void handlerIRQ1();
  void handlerIRQ4();
  void handlerIRQ7();
  void handlerIRQ14();
```

```
    void handlerIRQ19();



  SETGATE(idt[IRQ_OFFSET+IRQ_TIMER],  0, GD_KT, handlerIRQ0, 0);
  SETGATE(idt[IRQ_OFFSET+IRQ_KBD],    0, GD_KT, handlerIRQ1, 0);
  SETGATE(idt[IRQ_OFFSET+IRQ_SERIAL], 0, GD_KT, handlerIRQ4, 0);
  SETGATE(idt[IRQ_OFFSET+IRQ_SPURIOUS], 0, GD_KT, handlerIRQ7, 0);
  SETGATE(idt[IRQ_OFFSET+IRQ_IDE],    0, GD_KT, handlerIRQ14, 0);
  SETGATE(idt[IRQ_OFFSET+IRQ_ERROR],  0, GD_KT, handlerIRQ19, 0);
```

当调用用户中断处理函数时，处理器不会将 error code 压栈，也不会检查 IDT 入口的描述符特权等级，所以在 trapentry.S 中使用 TRAPHANDLER_NOEC（），当某个中断发生时，根据偏移量将对应的中断号压栈，然后开始执行相应的中断处理函数（call trap）：

```
TRAPHANDLER_NOEC(handlerIRQ0, IRQ_OFFSET+IRQ_TIMER)
TRAPHANDLER_NOEC(handlerIRQ1, IRQ_OFFSET+IRQ_KBD)
TRAPHANDLER_NOEC(handlerIRQ4, IRQ_OFFSET+IRQ_SERIAL)
TRAPHANDLER_NOEC(handlerIRQ7, IRQ_OFFSET+IRQ_SPURIOUS)
TRAPHANDLER_NOEC(handlerIRQ14, IRQ_OFFSET+IRQ_IDE)
TRAPHANDLER_NOEC(handlerIRQ19, IRQ_OFFSET+IRQ_ERROR)
```

确保用户进程总是在中断被打开的情况下运行，保证 FL_IF 被置位，如果这个进程运行时出现中断，中断就可以到达处理器并被相应的中断处理代码所处理。所以在 kern/env.c 的 env_alloc()中加入：

```
  // LAB 4: Your code here.
    e->env_tf.tf_eflags |= FL_IF;
```

十五、 作业 12

4.1.2. 处理时钟中断

在程序 user/spin 中，当子进程第一次运行后，它就不停地在循环，内核再也不能得到控制权。现在，我们需要让硬件定期的触发时钟中断，以强制将控制权转移到内核，这样就可以切换到其他的进程来运行。

函数 pic_init()和 kclock_init()（在 init.c 中的 i386_init）用来设置时钟和中断控制器来生成中断。现在，你需要编写代码来处理这些中断。

作业 **12**

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.

根据题意需要修改 trap_dispatch()函数，使之可以支持时钟中断，它调用 sched_yield()函数去处理始终中断；

因此，修改代码如下：

```
        break;
case (IRQ_TIMER + IRQ_OFFSET):
        lapic_eoi();
        sched_yield();
```

当 trapno=IRQ_TIMER+IRQ_OFFSET 时，根据提示需要先使用 lapic_eoi()确认中断，然后使用 sched_yield()处理时钟中断；

十六、 作业 13

在 JOS 的实验指导中，对进程间通信有如下描述：

我们需要实现一些额外的 JOS 内核的系统调用来提供一个简单的 IPC 机制。首先，将实现两个系统调用 sys_ipc_recv 和 sys_ipc_can_send。之后 ，再实现两个库函数 ipc_recv 和 ipc_send。

进程使用 JOS 的 IPC 机制所发送的消息包括两部分：一个 32 位的值以及可选的一个页的映射关系。在消息中允许进程传递页映射可以提供一个有效的方式来交换更多的数据，并使得进程之间可以很方便的设置共享内存。

要接收一个消息，进程必须调用 sys_ipc_recv。这个系统调用挂起当前进程并在收到消息之前停止运行。当一个进程在等待接受消息时，任何其他的进程都可以发送消息给它，而不是只有父子进程可以。也就是说，在前面实现的权限检查在 IPC 时将不会应用到 IPC 中，

因为 IPC 函数被认为是安全的：一个进程不可能通过发送消息就使得其他进程出现故障（除非目标进程本身有问题）。

为了发送一个消息，进程需要调用 sys_ipc_can_send，设置要接受的进程的 ID 和要发送的消息。如果目标进程正在接收数据（调用 sys_ipc_recv 并且还没有收到数据），则发送进程的函数就发送数据并返回 0。否则返回-E_IPC_NOT_RECV 以表明目标进程当前并没有准备接受数据。

用户空间的库函数 ipc_recv 将会调用 sys_ipc_recv 然后在当前进程的 Env 结构中获得接收的数据。

与此相类似，另一个库函数 ipc_send 将会反复的调用 sys_ipc_can_send 直到发送成功。

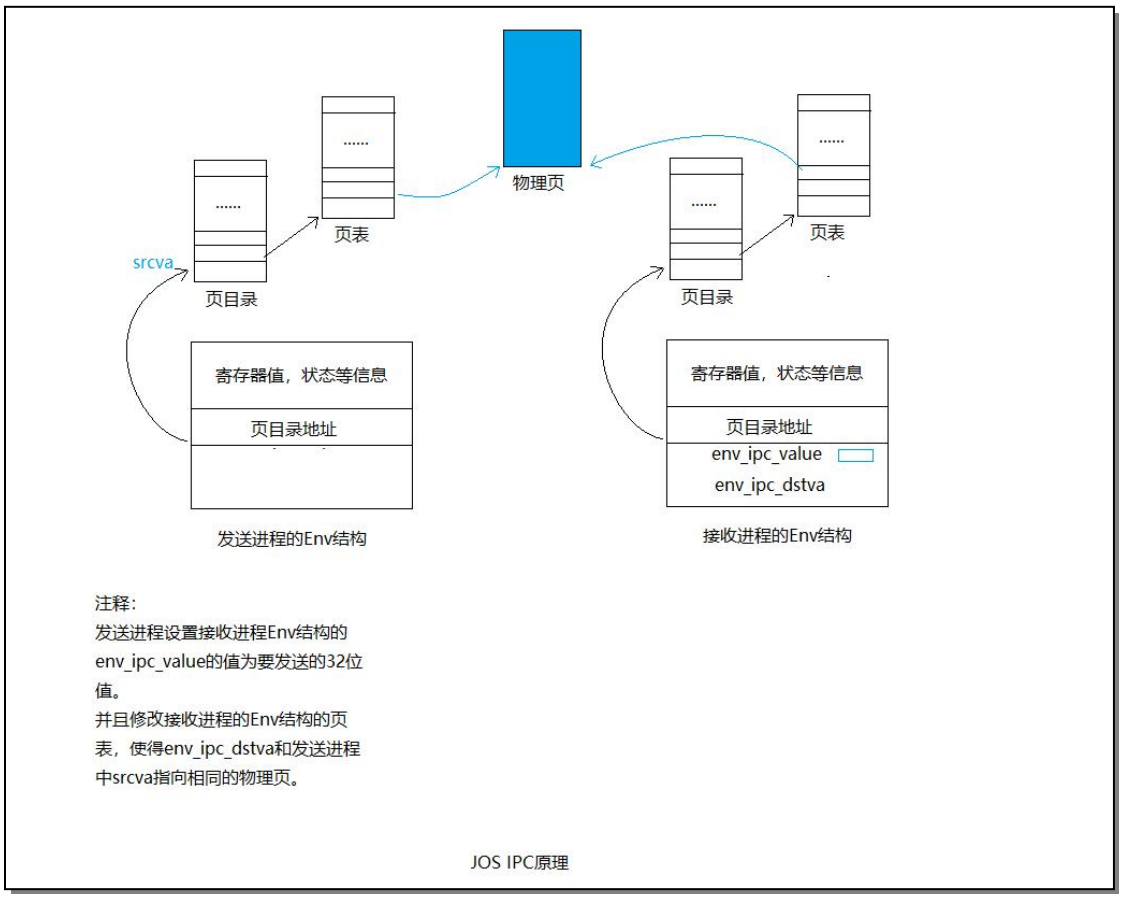
作业 **13**

Implement sys_ipc_recv 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_recv 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.

根据题意我们需要完成 IPC：实现 sys_ipc_recv()和 sys_ipc_try_send()。包装函数 ipc_recv()和 ipc_send()；本质还是进入内核修改 Env 结构的的页映射关系，原理理解如下：



为了实现进程间通信，我们需要查看 Enc 结构的变化：

```
struct Env {
        struct Trapframe env_tf;        // Saved registers
        struct Env *env_link;           // Next free Env
        envid_t env_id;                 // Unique environment identifier
        envid_t env_parent_id;          // env_id of this env's parent
        enum EnvType env_type;          // Indicates special system environments
        unsigned env_status;            // Status of the environment
        uint32_t env_runs;              // Number of times environment has run
        int env_cpunum;                 // The CPU that the env is running on

        // Address space
        pde_t *env_pgdir;               // Kernel virtual address of page dir

        // Exception handling
        void *env_pgfault_upcall;       // Page fault upcall entry point

        // Lab 4 IPC
        bool env_ipc_recving;           // Env is blocked receiving
        void *env_ipc_dstva;            // VA at which to map received page
        uint32_t env_ipc_value;         // Data value sent to us
        envid_t env_ipc_from;           // envid of the sender
        int env_ipc_perm;               // Perm of page mapping received
};
```

bool env_ipc_recving:当调用 ipc_recv 时，将该值置为 1，然后阻塞等待，当另外一个进程发送消息时，再解除阻塞态，将其置为 0；

env_ipc_dstva:用来设置接收的页面

env_ipc_value:发送的 32 位值

env_ipc_from:发送的进程 id

下面进入代码实现：

```
static int
sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)
{

        // LAB 4: Your code here.

        //if(envid == 0x1004)
        //      cprintf("when the envid =0x1004, the value is %d\n", value);
        struct Env *env=0;
        int r =0;
        pte_t * pte =0;
        if((r = envid2env(envid, &env, 0)) < 0)
                return -E_BAD_ENV;


        if(env->env_ipc_recving == 0)
                return -E_IPC_NOT_RECV;


        if((int)srcva < UTOP){

                if ( (int)srcva < UTOP &&  ((int)srcva % PGSIZE != 0) )
                        return -E_INVAL;

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

                if(  (perm & PTE_P) ==0 )
                        return  -E_INVAL;

                struct PageInfo *page  = page_lookup(curenv->env_pgdir, srcva, &pte);
                if( (perm & PTE_W) && ( (*pte & PTE_W) == 0) )
                        return  -E_INVAL;

                if((int)env->env_ipc_dstva >= UTOP)
                        return 0;
                r = page_insert(env->env_pgdir, page, env->env_ipc_dstva ,perm);
                if(r < 0)
                        return -E_NO_MEM;


        }

        env->env_ipc_value = value;
        env->env_ipc_from = curenv->env_id;
        env->env_ipc_perm = perm;
        env->env_ipc_recving = 0;
        env->env_status = ENV_RUNNABLE;
        env->env_tf.tf_regs.reg_eax = 0;

        return 0;
        //panic("sys_ipc_try_send not implemented");



}
```

```
static int
sys_ipc_recv(void *dstva)
{
        // LAB 4: Your code here.
        //panic("sys_ipc_recv not implemented");

        if((int)dstva >= UTOP)
                curenv->env_ipc_dstva = (void*)UTOP;
        else{
                if((int)dstva % PGSIZE != 0)
                        return -E_INVAL;
                else curenv->env_ipc_dstva = dstva;
        }

        curenv->env_status = ENV_NOT_RUNNABLE;
        curenv->env_ipc_recving = 1;
        sys_yield();

        return 0;



}
```

操作系统 Lab4

```
int32_t
ipc_recv(envid_t *from_env_store, void *pg, int *perm_store)
{
        // LAB 4: Your code here.
        int r =0;
        int a;
        if(pg == 0)
                r= sys_ipc_recv( (void *)UTOP);
        else
                r = sys_ipc_recv(pg);
        if(r == 0){
                if( from_env_store != 0 )
                        *from_env_store = thisenv->env_ipc_from;

                if(perm_store != 0 )
                        *perm_store = thisenv->env_ipc_perm;
        }
        else{
                panic("The ipc_recv is not right, and the errno is %d\n",r);
                if(from_env_store != 0 )
                        *from_env_store = 0;

                if(perm_store != 0 )
                        *perm_store = 0;
                return r;
        }
        if(thisenv->env_ipc_value == 0)
                cprintf("the value is 0, the envid is %x\n", thisenv->env_id);
        return thisenv->env_ipc_value;
        //panic("ipc_recv not implemented");
        //return 0;




}
```

```
void
ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
{
        // LAB 4: Your code here.
        //panic("ipc_send not implemented");

        int r =0;
        while(1){
                if(pg == 0)
                        r=sys_ipc_try_send(to_env,  val, (void*) UTOP,  perm);
                else
                        r = sys_ipc_try_send(to_env,  val, pg,  perm);

                if(r <0 && r != -E_IPC_NOT_RECV){
                        cprintf("the envid is %x\n", sys_getenvid());
                        panic("ipc_send is error, and the errno is %d\n", r);
                }
                else if(r == -E_IPC_NOT_RECV)
                        sys_yield();
                else break;
        }



}
```