JOS 2016 Lab 4 实习报告

陈一茹 1400012976

November 22, 2016

Contents

1 Introduction

这个 Lab 的内容较多，实现了很多操作系统的关键功能。比如对称多处理器 的支持，任务调度，以及用于新建进程、处理页错误的系统调用。还有支持写时 拷贝的 fork()，抢占式任务调度等较为高级的功能。

2 Part A: Multiprocessor Support and Cooperative Multitasking

2.1 Multiprocessor Support

这里，我们要将JOS扩展成一个支持多核处理器的系统。所有的CPU有相同的权限获得系统资源，如内存和I/O总线。每个核的地位是等价的。

在初始化CPU阶段，这些核会被分成两类: bootstrap processors（BSP）和 application processors(APS)。BSP负责在启动的初始化操作 ，以及加载操作系统，并启动APs。哪一个核承担BSP的工 作是由硬件和BIOS决定的。到lab3为止，我们可以认为JOS一直都运行在BSP这 个单核上。启动工作一旦完成，每个核就完全平等，不再进行区分了。

在SMP中，每个核都有自己的一个LAPIC单元，负责处理中断。在实习中，我们需要完成与LAPIC相关的下列工作:

- 1. 读取LAPIC的id，以获知当前代码是运行在哪一个核上的。
- 2. 从BSP向AP发送一个STARTUP中断，用于启动各个AP。
- 3. 实现有关时钟中断的工作, 一边这样在lab3中实现抢占式中断。

这里补充一下知识： PIC全称Programmable Interrupt Controller，通常是指中断控制系统。APIC全称Advanced Programmable Interrupt Controller，APIC是为了多核平台而设计的。它由两个部分组成IOAPIC和LAPIC，其中IOAPIC通常用于处理桥上的设备所产生的各种中断，LAPIC则是每个CPU都会有一个。IOAPIC通过APICBUS(现在都是通过FSB/QPI)将中断信息分派给每颗CPU的LAPIC,CPU上的LAPIC能够智能的决定是否接受系统总线上传递过来的中断信息，而且它还可以处理Local端中断的pending、nesting、masking，以及IOAPIC于Local CPU的交互处理。

每个核用于访问LAPIC的I/O设备被硬连到了内存中的一部分，称为MMIO，这和lab1的IO Hole十分相似。 首先要把MMIO的物理地址和虚拟地址联系起来， 建立映射，从而可以通过访问MMIOBASE来访问LAPIC。

Exercise 1. Implement `mmio_map_region` in `kern/pmap.c`. To see how this is used, look at the beginning of `lapic_init` in `kern/lapic.c`. You'll have to do the next exercise, too, before the tests for `mmio_map_region` will run.

```
void
lapic_init(void)
{
        if (!lapicaddr)
                return;

        // lapicaddr is the physical address of the LAPIC's 4K MMIO
        // region.  Map it in to virtual memory so we can access it.
        lapic = mmio_map_region(lapicaddr, 4096);
```

按要求，看一下mimo_map_region是怎么使用的。lapicaddr是定义在kern/mpconfig.c中的值，它表示lapic对应的IO设备硬连接的那部分内存(就是MMIO)的物理地址。因此，mmio_map_region()的作用就是把虚拟地址MMIOBASE和物理地址lapicaddr之上一定大小的空间对应起来。这个工作自然要使用boot_map_region()实现，其它相关的参数信息在注释中都已经给出了。

下面是我的代码：

```
//
// Reserve size bytes in the MMIO region and map [pa,pa+size) at this
// location.  Return the base of the reserved region.  size does *not*
// have to be multiple of PGSIZE.
//
void *
mmio_map_region(physaddr_t pa, size_t size)
{
        // Where to start the next region.  Initially, this is the
        // beginning of the MMIO region.  Because this is static, its
        // value will be preserved between calls to mmio_map_region
        // (just like nextfree in boot_alloc).
        static uintptr_t base = MMIOBASE;

        // Reserve size bytes of virtual memory starting at base and
        // map physical pages [pa,pa+size) to virtual addresses
        // [base,base+size).  Since this is device memory and not
        // regular DRAM, you'll have to tell the CPU that it isn't
        // safe to cache access to this memory.  Luckily, the page
        // tables provide bits for this purpose; simply create the
        // mapping with PTE_PCD|PTE_PWT (cache-disable and
```

```
22          // write-through) in addition to PTE_W.  (If you're
                interested
            // in more details on this, see section 10.5 of IA32
                volume
24          // 3A.)
            //
26          // Be sure to round size up to a multiple of PGSIZE and
                 to
            // handle if this reservation would overflow MMIOLIM (
                it's
28          // okay to simply panic if this happens).
            //
30          // Hint: The staff solution uses boot_map_region.
            //
32          // Your code here:
        size = ROUNDUP(size, PGSIZE);
34      if (base + size > MMIOLIM)
                    panic("mmio_map_region(): out of memory\n");
36          boot_map_region(kern_pgdir, base, size, pa, PTE_PCD |
                PTE_PWT | PTE_W);
                base += size;
38                  return (void*)(base - size);
            panic("mmio_map_region not implemented");
40  }
```

需要注意的地方就是：这里是一个设备的内存不是一个普通的DRAM，所以要告诉CPU不要cache access, 最后返回保留区域的基地址。同时不要超过MMIOLIM。

2.1.1 Application Processor Bootstrap

这是SMP的启动。

在启动AP的时候，BSP首先收集关于多处理器系统CPU个数的信息，他们的APIC ID和LAPIC的MMIO地址。 kern/mpconfig.c中的mp_init()检索这些信息。

仔细看一下 kern/init.c中的代码：

```
    void
2 i386_init(void)
  {
4         extern char edata[], end[];

6         // Before doing anything else, complete the ELF loading
               process.
          // Clear the uninitialized global data (BSS) section of
               our program.
8         // This ensures that all static/global variables start
              out zero.
          memset(edata, 0, end - edata);
10
          // Initialize the console.
12        // 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:

        // Starting non-boot CPUs
        boot_aps();
}
```

mp_init()完成收集信息的工作。完成之后，在调用lapic_init() 去初始化各个APs 的LAPIC。

叙述一下boot_aps()的流程：

- BSP首先将高地址的kern/mpentry.S的代码复制到低地址MPENTRY_PADDR(即0x7000)，这是由于AP启动时是在实模式下的，寻址范围比较小。
- boot_aps()向每一个相应的AP的LAPIC发送一个STARTUP和kern/mpentry.S的地址，然后等待AP完成启动。
- mpentry.S 中的entry code 和boot/boot.S 很相似，刚开始把AP放在实模式中， 然后简单设置之后，进入到保护模式中，并且开启了分页， 然后调用了mpmain()函数。
- 等到boot_aps()等到了AP发出一个CPU_STARTED的时候，他就开始去叫醒下一个。

Exercise 2. 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).

因为之后会将，AP的启动代码entry code to unused memory at MPENTRY_PADDR, 所以需要将这片空出来,改一下kern/pmap.c。

```
    size_t i;
    assert(page_free_list == 0);
    unsigned used_top = PADDR(boot_alloc(0));
    for (i = 0; i < npages; i++) {
        if (i == 0 || (page2pa(&pages[i]) >= IOPHYSMEM &&
            page2pa(&pages[i]) < used_top)||(page2pa(&pages[i])
             == MPENTRY_PADDR))
            continue;
        pages[i].pp_ref = 0;
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
```

只需要在for里面添一个条件就好了， 太简单啦！

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.

看一下这里的宏：

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

答： 这里要调这个宏， kern/mpentry.S是 加载在低地址、但是运行在高地址上的，所以在运行的过程中需要通过这样 的地址转换来找到需要执行的代码。这个宏就是一次由高地址向低地址的地址变换。相比之下，boot.S中的代码执行和加载都是在低地址，所以就不需要这样的地址转换。

2.1.2 Per-CPU State and Initialization

在支持多核的系统的实现时，区分开每一个核的状态是非常重要的。在JOS中，用于描述一个核的状态的结构体是CpuInfo，定义在kern/cpu/h中。

```
// Per-CPU state
struct CpuInfo {
        uint8_t cpu_id;                 // Local APIC ID; index
             into cpus[] below
        volatile unsigned cpu_status;   // The status of the
            CPU
```

```
5         struct Env *cpu_env;            // The currently-
              running environment.
          struct Taskstate cpu_ts;        // Used by x86 to find
              stack for interrupt
7 };
```

JOS还有一个函数cpunum(), 他总是会返回调用这个函数的核的编号，JOS中还有一个宏thiscpu,他返回一个指向当前代码的核的CPUinfo结构的CPuinfo类型指针。通过这两种方式我们可以 很快的访问当前核。

还有需要注意的就是一些CPU状态：

- 内核栈 ： 由于多个核上的程序可能同时发生陷入，所以我们需要为每一个核都准备一个内核栈以存放陷入的时候用户的进程状态。数组percpu_kstacks[NCPU][KSTKSIZE]保存着各个核的内存栈空间。
- TSS 和TSS描述符，TSS描述符用来记录每个活跃的kernel栈的地址。全局变量ts不在有用了。
- 当前运行进程的指针:相当于我们在lab3中的curenv。这里，用cpus[cpunum()].cpu env就 可以访问到。
- 寄存器信息:每个核都有专属于自己的寄存器。

Exercise 3. Modify `mem_init_mp()` (in `kern/pmap.c`) to map per-CPU stacks starting at `KSTACKTOP`, as shown in `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()`.

这部分要求我们把每个核的栈数组对应到虚拟地址空间当中。这部分参考memlayout.h, 我们可以看到， 每个核的内核栈占据了KSTKSIZE+KSTKGAP的大小。

```
1 // Modify mappings in kern_pgdir to support SMP
  //   - Map the per-CPU stacks in the region [KSTACKTOP-PTSIZE,
      KSTACKTOP)
3 //
      static void
5 mem_init_mp(void)
  {
7     // Map per-CPU stacks starting at KSTACKTOP, for up to '
          NCPU' CPUs.
      //
9     // For CPU i, use the physical memory that 'percpu_kstacks[
          i]' refers
      // to as its kernel stack. CPU i's kernel stack grows down
          from virtual
11    // address kstacktop_i = KSTACKTOP - i * (KSTKSIZE +
          KSTKGAP), and is
```

```
        // divided into two pieces, just like the single stack you
            set up in
13      // mem_init:
        //      * [kstacktop_i - KSTKSIZE, kstacktop_i)
15      //           -- backed by physical memory
        //      * [kstacktop_i - (KSTKSIZE + KSTKGAP), kstacktop_i -
             KSTKSIZE)
17      //           -- not backed; so if the kernel overflows its
            stack,
        //              it will fault rather than overwrite another
            CPU's stack.
19      //              Known as a "guard page".
        //      Permissions: kernel RW, user NONE
21      //
        // LAB 4: Your code here:
23      int c;
        for (c = 0; c < NCPU; ++c) {
25          boot_map_region(kern_pgdir, KSTACKTOP - c * (KSTKSIZE +
                KSTKGAP) - KSTKSIZE,
                    ROUNDUP(KSTKSIZE, PGSIZE), PADDR(percpu_kstacks
                        [c]), PTE_W);
27      }
    }
```

Exercise 4. 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.)

这里需要初始化一下TSS和TSS描述符， 然后发现在单CPU的时候我们进行过设置，只需要稍作修改就可以了。一下代码：

```
    // Initialize and load the per-CPU TSS and IDT
2       void
    trap_init_percpu(void)
4   {
        // The example code here sets up the Task State Segment (
            TSS) and
6       // the TSS descriptor for CPU 0. But it is incorrect if we
            are
        // running on other CPUs because each CPU has its own
            kernel stack.
8       // Fix the code so that it works for all CPUs.
        //
10      // Hints:
        //   - The macro "thiscpu" always refers to the current CPU
            's
12      //     struct CpuInfo;
        //   - The ID of the current CPU is given by cpunum() or
14      //     thiscpu->cpu_id;
```

```
        //   - Use "thiscpu->cpu_ts" as the TSS for the current CPU
            ,
16      //     rather than the global "ts" variable;
        //   - Use gdt[(GD_TSS0 >> 3) + i] for CPU i's TSS
            descriptor;
18      //   - You mapped the per-CPU kernel stacks in mem_init_mp
            ()
        //
20      // ltr sets a 'busy' flag in the TSS selector, so if you
        // accidentally load the same TSS on more than one CPU, you
            'll
22      // get a triple fault.  If you set up an individual CPU's
            TSS
        // wrong, you may not get a fault until you try to return
            from
24      // user space on that CPU.
        //
26      // LAB 4: Your code here:
           int i = thiscpu->cpu_id;
28
        thiscpu->cpu_ts.ts_esp0 = KSTACKTOP - i * (KSTKSIZE +
            KSTKGAP);
30      thiscpu->cpu_ts.ts_ss0 = GD_KD;

32   // Initialize the TSS slot of the gdt.
        gdt[(GD_TSS0 >> 3) + i] = SEG16(STS_T32A, (uint32_t) (&
            thiscpu->cpu_ts), sizeof(struct Taskstate), 0);
34


36      gdt[GD_TSS0 >> 3+i].sd_s = 0;

38      // Load the TSS selector (like other segment selectors, the
        // bottom three bits are special; we leave them 0)
40      ltr(GD_TSS0+(i<<3));

42      // Load the IDT
        lidt(&idt_pd);
44   }
```

关于这个TSS0再说两点： 刚开始的时候在kern/env.c设置了全局描述符表，

```
      // Per-CPU TSS descriptors (starting from GD_TSS0) are
          initialized
 2        // in trap_init_percpu()
          [GD_TSS0 >> 3] = SEG_NULL
```

然后在这里对这块区域进行初始化。全局描述符中的gdt[(GD_TSS0 ¿¿ 3) + i] 代表的是一个TSS描述符，然后 GD_TSS0+(i¡¡3)是地址， load进tr寄存器里。

make qemu-nox CPUS=4

检查正确性：是正确的！！！

2.1.3 Locking

这部分说的是big kernel lock。在多CPU的情况下，我们要解决CPU之间的竞争问题，即保证同一时间内最多只可以有一个CPU来访问内核态程序。我们用的是big kernel lock, 在一个CPU访问内核态之前，先会试图加锁。如果此时内核态没有CPU访问，则当前的CPU就会锁住内核，开始访问内核。如果恰好有另外一个CPU在访问内核，那么当前的CPU就会阻塞在加锁的位置。当然，一个CPU结束对内核的访问之后就会对内核解锁。

kern/spinlock.h中有锁，JOS为我们提供了两个函数：lock_kernel()和unlock_kernel()。分别用于加锁和解锁。

- In i386_init(), acquire the lock before the BSP wakes up the other CPUs.
- In mp_main(), acquire the lock after initializing the AP, and then call sched_yield() to start running environments on this AP.
- In trap(), acquire the lock when trapped from user mode. To determine whether a trap happened in user mode or in kernel mode, check the low bits of the tf_cs.
- In env_run(), release the lock right before switching to user mode. Do not do that too early or too late, otherwise you will experience races or deadlocks.

按照提示加锁 就可以了！
两句话加锁就可以了
kern/init.c:

```
1   // Acquire the big kernel lock before waking up APs
      // Your code here:
3     spin_initlock(&kernel_lock);
      lock_kernel();
```

刚开始 要初始化一次。
mp_main()，trap()很简单，不展示了。
env_run(), 这里注意要在用户态返回内核态之前去除Kernel lock, 这位置应该在env_pop_tf()之前。
这个没有提示代码位置，我把他加载env_pop_tf()的最前面。

```
        void
2 env_pop_tf(struct Trapframe *tf)
  {
4         // Record the CPU we are running on for user-space
              debugging
          curenv->env_cpunum = cpunum();
6         unlock_kernel();
```

所以这样就完成了。

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.

假设有这样的情形， 如果所有的CPU共享一个内核栈，当CPU0上发生一个中断的时候，中断信息在trapentry.S中被压入内核栈中， 然后 调用trap函数，给kernel 加锁。 这时候CPU1也发生一个中断，相关信息被压栈，然后CPU1试图陷入。 但是此时，有CPU0正在访问内核态，所以big kernel lock被加上，CPU1阻塞。如果恰巧这个时候CPU0的中断处理完毕了，需要回复场景，那么这个时候 从栈中弹出的信息就是CPU1刚刚压入的信息， 是错误了。

2.2 Round-Robin Scheduling

多进程调度： 这里要实现的是轮转调度。 JOS内核中负责实现这个功能的是kern/sched.c中的 sched_yield()。 每次调度都是CPU主动让步完成的。 每次当一个用户态进程主动下CPU时， 他会调用sched_yield() 函数实现进程让步， 由sched_yield() 来决定下一个上CPU的进程是谁。 需要提出的是， sched_yield()并不是由用户直接调用的， 他其实是调用了一个sys_yield() 的系统调用，然后由这个系统调用去调用sched_yield() 。

Exercise 6. Implement round-robin scheduling in `sched_yield()` as described above. Don't forget to modify `syscall()` to dispatch `sys_yield()`.

Make sure to invoke `sched_yield()` in `mp_main`.

Modify `kern/init.c` to create three (or more!) environments that all run the program `user/yield.c`.

Run `make qemu`. You should see the environments switch back and forth between each other five times before terminating, like below.

Test also with several CPUS: `make qemu CPUS=2`.

```
...
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.
...
```

After the `yield` programs exit, there will be no runnable environment in the system, the scheduler should invoke the JOS kernel monitor. If any of this does not happen, then fix your code before proceeding.

If you use `CPUS=1` at this point, all environments should successfully run. Setting CPUS larger than 1 at this time may result in a general protection fault, kernel page fault, or other unexpected interrupt once there are no more runnable environments due to unhandled timer interrupts (which we will fix below!).

```
    // Choose a user environment to run and run it.
2     void
  sched_yield(void)
4 {
      struct Env *idle;
6
      // Implement simple round-robin scheduling.
8     //
      // Search through 'envs' for an ENV_RUNNABLE environment in
10    // circular fashion starting just after the env this CPU
          was
      // last running.  Switch to the first such environment
          found.
12    //
      // If no envs are runnable, but the environment previously
14    // running on this CPU is still ENV_RUNNING, it's okay to
      // choose that environment.
16    //
      // Never choose an environment that's currently running on
18    // another CPU (env_status == ENV_RUNNING). If there are
      // no runnable environments, simply drop through to the
          code
20    // below to halt the cpu.
      // LAB 4: Your code here.
22    int i, j;

24    if (thiscpu->cpu_env) i = thiscpu->cpu_env - envs;
      else i = NENV - 1;
26
      for (j = (i + 1) % NENV; j != i; j = (j + 1) % NENV) {
28        if (envs[j].env_status == ENV_RUNNABLE)
              env_run(&envs[j]);
30    }
      if (thiscpu->cpu_env && thiscpu->cpu_env->env_status ==
          ENV_RUNNING)
32            env_run(thiscpu->cpu_env);

34    // sched_halt never returns
      sched_halt();
36 }
```

i = thiscpu-¿cpu_env - envs就是取出当前的进程是总进程列表中的第几个，然后依次轮转调度。

最后因为是系统调用触发的，所以添上系统调用，sys_yield()。

然后在i386_Init()添加三个进程，run yield.c。

```
  ENV_CREATE(user_yield, ENV_TYPE_USER);
2 ENV_CREATE(user_yield, ENV_TYPE_USER);
  ENV_CREATE(user_yield, ENV_TYPE_USER);
```

刚开始是有点问题，env_set_vm中不能一个一个映射，因为现在已经不

是lab3 了，不止一个内核栈。 这里需要再改一下。

而其前面的trap_dispatch()函数在merge 的时候也出现了问题， 也需要将break 改成return。

有点坑 = = 。 。 。 。 。

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?
4. Whenever the kernel switches from one environment to another, it must ensure the old environment's registers are saved so they can be restored properly later. Why? Where does this happen?

因为这些变量都是内核空间的变量，(即 KERNBASE 以上部分)，并且所有 进程的内核空间映射都是一样的。所以无论什么进程在运行，指向的空间都是一样的。

肯定要保存之前进程的上下文，以便之后恢复。因为系统要让进程调度对每个 进程透明，每个进程都认为自己是机器上运行的唯一进程。

在陷入内核态时，硬件已经保存了部分进程上下文在内核栈中，之后中断处理 程序的_alltraps 部分又保存了另一部分，它们共同组成了一个 Trapframe，存 在内核栈顶端，在 trap()中的 tf 参数就指向这个 Trapframe。位于 trap()中间 部分的:

```
1           // Copy trap frame (which is currently on the stack)
          // into 'curenv->env_tf', so that running the
              environment
3         // will restart at the trap point.
          curenv->env_tf = *tf;
5         // The trapframe on the stack should be ignored from
              here on.
          tf = &curenv->env_tf;
```

将这个 Trapframe 存在 env 结构体中，在 env_pop_tf()时再将这些值恢复到寄 存器中。

2.3 System Calls for Environment Creation

我们可以从多个用户进程之间切换，但是我目前还是只能在内核态生成进程，现在我要实现fork()，来支持从用户进程生成一个新的进程。

在父进程中，返回子进程的ID号，子进程中返回0。

给出几个fork 要求实现：

- sys_exofork:创建一个空的进程，即它的地址空间里没有内容，但是相关的寄存器信息和父进程一致。
- sys_env_set_status:设置一个进程的状态。
- sys_page_alloc:为一个进程分配内存页面。
- sys_page_map:从一个进程中复制一个页对应到另一个进程中。注意，这里只复制对应关系，而不复制实际存储的内容。
- sys_page_unmap:解除一个页的对应。

Exercise 7. 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.

首先实现： sys_exofork()

```
// Allocate a new environment.
// Returns envid of new environment, or < 0 on error.  Errors are:
//      -E_NO_FREE_ENV if no free environment is available.
//      -E_NO_MEM on memory exhaustion.
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 *e = NULL;
    int r;
    if ((r = env_alloc(&e, curenv->env_id())) < 0) return r;
    memcpy(&e->env_tf, &curenv->env_tf, sizeof(e->env_tf));
    e->env_tf.tf_regs.reg_eax = 0;
    e->env_status = ENV_NOT_RUNNABLE;
    return e->env_id;
}
```

这里的eax设置成了0，因为之后新生成的进程会从这里去取sys_exofork()的返回值，所以要返回0。

实现sys_env_set_status()：

```
// Set envid's env_status to status, which must be ENV_RUNNABLE
// or ENV_NOT_RUNNABLE.
//
// Returns 0 on success, < 0 on error.  Errors are:
//      -E_BAD_ENV if environment envid doesn't currently exist,
//              or the caller doesn't have permission to change envid.
//      -E_INVAL if status is not a valid status for an environment.
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 *e;
    if (envid2env(envid, &e, 1) == -E_BAD_ENV) return -E_BAD_ENV;
    if (status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE)
        return -E_INVAL;
    e->env_status = status;
    return 0;

}
```

sys_page_alloc(): Allocate a page of memory and map it at 'va' with permission 'perm' in the address space of 'envid'.

```
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 *e;
    struct PageInfo *p;
    perm &= PTE_SYSCALL;
    if (envid2env(envid, &e, 1) == -E_BAD_ENV) return -E_BAD_ENV;
```

```
16      if ((uintptr_t)va >= UTOP || (uintptr_t)va % PGSIZE != 0)
            return -E_INVAL;
        if ((perm & (PTE_U | PTE_P)) != (PTE_U | PTE_P)) return -
            E_INVAL;
18      if (!(p = page_alloc(ALLOC_ZERO))) return -E_NO_MEM;
        if (page_insert(e->env_pgdir, p, va, perm) == -E_NO_MEM) {
20          page_free(p);
            return -E_NO_MEM;
22      }
        return 0;
24  }
```

按照注释一行一行写咯 这里，要注意的也就一个地方吧，就是page_insert不成功的话，就要返回一个错误，并且，要执行page_free()。

sys_page_map()。要实现的是Map the page of memory at 'srcva' in srcenvid's address space at 'dstva' in dstenvid's address space with permission 'perm'.

```
         static int
 2   sys_page_map(envid_t srcenvid, void *srcva,
            envid_t dstenvid, void *dstva, int perm)
 4   {
        // Hint: This function is a wrapper around page_lookup()
            and
 6      //   page_insert() from kern/pmap.c.
        //   Again, most of the new code you write should be to
            check the
 8      //   parameters for correctness.
        //   Use the third argument to page_lookup() to
10      //   check the current permissions on the page.

12      // LAB 4: Your code here.
        struct Env *esrc, *edst;
14      struct PageInfo *p;
        pte_t *pte;
16      perm &= PTE_SYSCALL;
        if (envid2env(srcenvid, &esrc, 1) == -E_BAD_ENV) return -
            E_BAD_ENV;
18      if (envid2env(dstenvid, &edst, 1) == -E_BAD_ENV) return -
            E_BAD_ENV;
        if ((uintptr_t)srcva >= UTOP || (uintptr_t)srcva % PGSIZE
            != 0) return -E_INVAL;
20      if ((uintptr_t)dstva >= UTOP || (uintptr_t)dstva % PGSIZE
            != 0) return -E_INVAL;
        if (!(p = page_lookup(esrc->env_pgdir, srcva, &pte)))
            return -E_INVAL;
22      if ((perm & (PTE_U | PTE_P)) != (PTE_U | PTE_P)) return -
            E_INVAL;
        if (perm & PTE_W & ~*pte) return -E_INVAL;
24      if (page_insert(edst->env_pgdir, p, dstva, perm) == -
            E_NO_MEM){
            page_free(p);
26          return -E_NO_MEM;
        }
```

```
28          return 0;
    }
```

要注意的是， 不能把写权限赋给一个只读的。

最后是sys_page_unmap()。easy! Unmap the page of memory at 'va' in the address space of 'envid'. If no page is mapped, the function silently succeeds.

```
1       static int
  sys_page_unmap(envid_t envid, void *va)
3 {
      // Hint: This function is a wrapper around page_remove().
5
      // LAB 4: Your code here.
7     struct Env *e;
      if (envid2env(envid, &e, 1) == -E_BAD_ENV) return -
          E_BAD_ENV;
9     if ((uintptr_t)va >= UTOP || (uintptr_t)va % PGSIZE != 0)
          return -E_INVAL;
      page_remove(e->env_pgdir, va);
11    return 0;
  }
```

3 Part B: Copy-on-Write Fork

fork()的时候提到，fork()创建的子进程的地址空间中的内容 和父进程是一样的。这也就意味着我们在创建进程时需要将父进程的地址空间 中的内容复制到子进程当中。不过，这样做的效率太低了。在JOS(和新版本的Unix)中，取而代之的，采用的方法是在创建进程时使用一种叫做copy on write的技术。

- 1. 在创建进程时，并不将父进程的全部内容都复制到子进程中，而是只将 父进程的页目录复制到子进程的页目录当中。换言之，此时的父、子进 程的虚拟地址空间指向相同的页面。
- 2. 对于父进程中所有可写的页面，将子进程中对应的页面设置为一种特 殊的访问权限:COW(即Copy On Write)。当然，如果父进程中的某一个页面本身就是COW的，那么这个页面在子进程中的映射也应当 是COW的。
- 3. 当两个进程中的某一个试图去修改这些页面时，由于页面的访问权限 是COW，会产生一次发生在用户态中的Page fault。
- 4. Page fault发生之后，内核启动做相应的处理，为子进程分配一页并将父 进程的信息真正的复制过去，并将这一页的权限改为可写。

所以，在实现fork()之前，我们还需要再做一件准备工作，那就是对于用户 态中缺页中断的处理。

3.1 User-level page fault handling

这里需要补充一下用户态错误处理栈，User Exception Stack（UXSTACK），这个栈定义在用户栈的正上方。

当一个用户态进程正在运行的时候，当前栈是USTACK。缺页发生后，首先会发生一次中断，此时会从用户态进程陷入内核态进程，此时栈就会被切换到KSTACK。当内核态中的相关处理程序发现这个缺页是来自用户进程而不是内核进程的时候，系统会在(UXSTACK)中分配一个UTrapframe的大小的空间，在其中存放出错进程的信息，然后把栈切换到UXSTACK，运行用户错误处理程序。最终，当错误解决了之后，利用存放在UTrapframe中的信息，系统再将栈切换到USTACK继续之前发生的用户态进程。

由此可见，用户态进程发生的中断不是由内核来处理，而是交由用户态自己处理，内核扮演的是一个调度者的角色。

所以，JOS用户态缺页终端的处理流程是：首先，当发生缺页中断时，又一次陷入，调用链是kern/trap.c: trap()、trap_dispatcher()、page_fault_handler()。最后在page_fault_handler()中判断出这是一次用户态缺页，因此在UXSTACK中申请一个UTrapframe，设置好其相应内容，然后转向用户态缺页处理的入口:lib/pfentry.S中的 pgfault upcall。在_pgfault_upcall处，执行用户自定义的中断处理程序_pgfault_handler(这是一个函数指针，指向用户程序中的某一个缺页处理函数)。执行结束后，还是在 _pgfault _upcall处，恢复之前的用户进程状态，继续执行。

下边看看UTrapframe。UTrapframe是放置在UXSTACK中用于存放错误处理信息的结构，它定义在inc/trap.h:

```
struct UTrapframe {
        /* information about the fault */
        uint32_t utf_fault_va;  /* va for T_PGFLT, 0 otherwise */
        uint32_t utf_err;
        /* trap-time return state */
        struct PushRegs utf_regs;
        uintptr_t utf_eip;
        uint32_t utf_eflags;
        /* the trap-time stack to return to */
        uintptr_t utf_esp;
} __attribute__((packed));
```

这里和Trapframe 很相似。有一点小的区别：

- UTrapframe比Trapframe多了一个fault_va域，这是由于UTrapframe是用来处理用户态错误的，所以需要这样一个域来记录用户态进程中的出错地址。
- UTrapframe相比于Trapframe少了es、ds、cs等寄存器信息，这是因为运行用户态错误处理程序的进程实际上与发生错误的进程是同一个进程，所以这里并没有涉及段的切换，也就自然不需要保存这些寄存器信息。

3.1.1 Setting the Page Fault Handler

Exercise 8. 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.

```
// Set the page fault upcall for 'envid' by modifying the corresponding struct
// Env's 'env_pgfault_upcall' field.  When 'envid' causes a page fault, the
// kernel will push a fault record onto the exception stack, then branch to
// 'func'.
//
// Returns 0 on success, < 0 on error.  Errors are:
//      -E_BAD_ENV if environment envid doesn't currently exist,
//              or the caller doesn't have permission to change envid.
static int
sys_env_set_pgfault_upcall(envid_t envid, void *func)
{
    // LAB 4: Your code here.
    struct Env *e;
    if (envid2env(envid, &e, 1) == -E_BAD_ENV) return -E_BAD_ENV;
    e->env_pgfault_upcall = func;
    return 0;
}
```

这里主要就将Env结构中的'env_pgfault_upcall' field设置成func，就是将错误处理设置成了函数func， 然后在做一些检查错误的事情。

3.1.2 Normal and Exception Stacks in User Environments

加深一下印象，内核起到了一个代表用户进程进行自动栈转换的功能，user exception stack只有一页大小，from UXSTACKTOP-PGSIZE through UXSTACKTOP-1 inclusive。当在这个异常栈上 运行的时候，用户态的page fault handler可以用JOS正常的系统调用去映射或者调整新的页。 最后这个错误处理程序返回通过汇编语言，返回到原来栈上。

3.1.3 Invoking the User Page Fault Handler

Exercise 9. 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?)

正如之前说的，当我们捕捉到一个用户态的缺页中断时，建立好UTrapframe，然后将控制权交给具体处理中断的函数。流程大致是这样的：

- 需要判断一下当前的中断是否是发生在用户态中断处理程序中。当一个用户进程发生缺页中断时，直接进中断处理程序。这时候中断处理程序中可能再次发生缺页中断，这也是缺页的递归。如果发生的中断是来自中断处理程序的，那么我们申请UTrapframe的时候需要在它的上面添加4字节的预留空间。判断也很简单，只要判断当前栈指针是否在UXSTACK之中就行了。
- 注释里面要求我们user_mem_assert()检查一下所在地址的空间权限。
- 设置UTrapframe的相关内容
- 把EIP设置成中断处理函数的地址， ESP设置成utf,即将控制权转交给中断处理函数。

```
1         struct UTrapframe *utf;
      uint32_t esp = tf->tf_esp;
3
      if (curenv->env_pgfault_upcall) {
5         if (esp < UXSTACKTOP - PGSIZE || esp >= UXSTACKTOP) tf
              ->tf_esp = UXSTACKTOP+4;
          utf = (struct UTrapframe*)(tf->tf_esp - 4 - sizeof(
              struct UTrapframe));
7         user_mem_assert(curenv, (const void*)utf, 1, PTE_W|
              PTE_U);
          lcr3(PADDR(curenv->env_pgdir));
9         utf->utf_fault_va = fault_va;
          utf->utf_err = tf->tf_err;
11        utf->utf_regs = tf->tf_regs;
          utf->utf_eip = tf->tf_eip;
13        utf->utf_eflags = tf->tf_eflags;
          utf->utf_esp = esp;
15        tf->tf_esp = (uint32_t)utf;
          tf->tf_eip = (uint32_t)curenv->env_pgfault_upcall;
17        env_run(curenv);
      }
```

3.1.4 User-mode Page Fault Entrypoint

这段代码需要理解一下UXSTACK中的UTrapframe的信息，然后完成缺页处理程序之后进程恢复工作。本质就是把UXSTACK栈中的UTrapframe中

存放的信息弹到相应的寄存器中。 这里就可以发现原来递归调用留下的那个padding 是非常有用的了！

```
      // LAB 4: Your code here.
2     subl $4, 48(%esp)
      movl 48(%esp), %eax
4     movl 40(%esp), %edx
      movl %edx, (%eax)
```

48(%esp)这个就是utf_esp的值，减去4， 然后保存utf_eip， 这样之后ret（相当于pop ip）的时候， 就可以找到正确的程序执行地点了。

上面那种是当从用户态发生page fault 的时候，如果是中断处理程序发生page fault,仍然这么操作，最后执行的就是，将eip 填到了事先留的padding处。

```
1     // Restore the trap-time registers.  After you do this, you
      // can no longer modify any general-purpose registers.
3     // LAB 4: Your code here.

5     addl $8, %esp
      popal
```

跳过八字节utf_fault_va; uint32_t utf_err; 然后将通用寄存器全部pop出来。

```
     // Restore eflags from the stack.  After you do this, you
        can
2     // no longer use arithmetic operations or anything else
         that
      // modifies eflags.
4     // LAB 4: Your code here.
      addl $4, %esp
6     popfl
```

跳过四字节，将EFLAGS弹出，注意这里的EFLAGS主要提供程序的状态和相应的控制。

```
    // Switch back to the adjusted trap-time stack.
2     // LAB 4: Your code here.
      popl %esp
4
      // Return to re-execute the instruction that faulted.
6     // LAB 4: Your code here.

8     ret
```

切换成原来的栈，这时候栈指针指向刚才保存过的trap time eip，然后ret，pop 出eip , 程序就能够正确执行了。

Exercise 11. Finish `set_pgfault_handler()` in `lib/pgfault.c`.

这个函数的作用就是注册一下用户态缺页处理程序

```
//
// Set the page fault handler function.
// If there isn't one yet, _pgfault_handler will be 0.
// The first time we register a handler, we need to
// allocate an exception stack (one page of memory with its top
// at UXSTACKTOP), and tell the kernel to call the assembly-language
// _pgfault_upcall routine when a page fault occurs.
//
void
set_pgfault_handler(void (*handler)(struct UTrapframe *utf))
{
int r;

if (_pgfault_handler == 0) {
// First time through!
// LAB 4: Your code here.
sys_page_alloc(0, (void*)(UXSTACKTOP - PGSIZE), PTE_U | PTE_W | PTE_P);
sys_env_set_pgfault_upcall(0, (void*)_pgfault_upcall);
}
// Save handler pointer for assembly to call.
_pgfault_handler = handler;
}
```

如果 是第一次设置，我们要去分配exception stack 栈，并且告诉内核去调用汇编代码。然后设置_pgfault_handler = handler。这样之后就能够完成page fault的处理了。

3.1.5 Testing

不赘述。

3.2 Implementing Copy-on-Write Fork

有了硬件的支持，我们可以去实现COW了。 dumpfork()复制每个页，但是fork()只复制页的映射，当有一个进程需要向里面写的时候，才需要复制整个页的内容。

Exercise 12. 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.

```
1000: I am ''
1001: I am '0'
2000: I am '00'
2001: I am '000'
1002: I am '1'
3000: I am '11'
3001: I am '10'
4000: I am '100'
1003: I am '01'
5000: I am '010'
4001: I am '011'
2002: I am '110'
1004: I am '001'
1005: I am '111'
1006: I am '101'
```

```
//
// Custom page fault handler - if faulting page is copy-on-write,
// map in our own private writable copy.
//
static void
pgfault(struct UTrapframe *utf)
{
    void *addr = (void *) utf->utf_fault_va;
    uint32_t err = utf->utf_err;
    int r;

    // Check that the faulting access was (1) a write, and (2) to a
    // copy-on-write page.  If not, panic.
    // Hint:
    //   Use the read-only page table mappings at uvpt
    //   (see <inc/memlayout.h>).

    // LAB 4: Your code here.
    if (!(utf->utf_err & FEC_WR)) panic("pgfault(): FEC_WR\n");
    if (!uvpd[((uintptr_t)addr >> 22)]) panic("pgfault(): page not mapped");
    if (!(pte = uvpt[(uintptr_t)addr >> 12])) panic("pgfault(): page not mapped");
    if (!(pte & PTE_COW)) panic("pgfault: PTE_COW\n");


    // Allocate a new page, map it at a temporary location (PFTEMP),
    // copy the data from the old page to the new page, then move the new
```

```
    // page to the old page's address.
    // Hint:
    //   You should make three system calls.

    // LAB 4: Your code here.
    sys_page_alloc(0, (void*)PFTEMP, PTE_P | PTE_U | PTE_W);
    memcpy((void*)PFTEMP, (void*)ROUNDDOWN((uintptr_t)addr,
        PGSIZE), PGSIZE);
    sys_page_map(0, (void*)PFTEMP, 0, (void*)ROUNDDOWN((
        uintptr_t)addr, PGSIZE), PTE_P | PTE_U | PTE_W);
    sys_page_unmap(0, (void*)PFTEMP);
}
```

这个函数首先判断是否是非法写操作引起的，接着判断是当前一页的页表是否在内存中， 我用了两个判断来判断 先判断页目录项，在判断页。 最后判断当前页面是否是COW的。如果上述判断都成立，那么确认当前中断是一次COW缺页中断。

处理一下这样的中断，所有页面复制到一份新的页面中去，再将新页面对应到出错进程的虚拟地址空间中去，新页面的权限设为可写。注：JOS中为我们提供了一个数 组uvpt[]，以页面的编号为下标就可以查到页表。同样JOS也为我们提供了另一 个数组uvpd[]以便我们能够查找页目录项

```
    static int
duppage(envid_t envid, unsigned pn)
{
    int r;

    // LAB 4: Your code here.
    int perm = uvpt[pn] & 0xfff;

    if (perm & (PTE_W | PTE_COW)) {
        if ((r = sys_page_map(0, (void*)(pn * PGSIZE), envid, (
            void*)(pn * PGSIZE), (perm & ~PTE_W) | PTE_COW)))
            return r;
        if ((r = sys_page_map(0, (void*)(pn * PGSIZE), 0, (void
            *)(pn * PGSIZE), (perm & ~PTE_W) | PTE_COW)))
            return r;
    }
    else {
        if ((r = sys_page_map(0, (void*)(pn * PGSIZE), envid, (
            void*)(pn * PGSIZE), perm)))
            return r;
    }
    return 0;
}
```

这里实现的就是如果是写或者是写时拷贝，就将父进程的相应页map到子进程的相应地址，同时 还需要map一下自己的设置为写时拷贝，如果本页面只可读，那么就map过来只能读的页。

这里还有一个注意点，就是注释里的问题，我还需要将父进程的重新映

射一遍，全部映射成为写时拷贝，因为如果父进程是写的话，子进程映射成为写时拷贝，父进程还可以往原来的地方写，但是子进程还会在原来的页面上读，这样显然就会产生错误。所以嘛，父进程的所有能够写的页都要改成写时拷贝。

```
//
// User-level fork with copy-on-write.
// Set up our page fault handler appropriately.
// Create a child.
// Copy our address space and page fault handler setup to the child.
// Then mark the child as runnable and return.
//
// Returns: child's envid to the parent, 0 to the child, < 0 on error.
// It is also OK to panic on error.
//
// Hint:
//   Use uvpd, uvpt, and duppage.
//   Remember to fix "thisenv" in the child process.
//   Neither user exception stack should ever be marked copy-on-write,
//   so you must allocate a new page for the child's user exception stack.
//
envid_t
fork(void)
{
    // LAB 4: Your code here.
    extern void _pgfault_upcall(void);
    uintptr_t p = 0;
    envid_t envid;
    set_pgfault_handler(pgfault);

    envid = sys_exofork();
    if (envid < 0) return envid;
    if (envid == 0) {
        thisenv = &envs[ENVX(sys_getenvid())];
        return 0;
    }

    while (p < UTOP) {
        if (!uvpd[p >> 22]) {
            p += PGSIZE << 10;
            continue;
        }
        if (p != UXSTACKTOP - PGSIZE && uvpt[p >> 12]) duppage(envid, p >> 12);
        p += PGSIZE;
    }
    if (sys_page_alloc(envid, (void*)(UXSTACKTOP - PGSIZE), PTE_P | PTE_U | PTE_W) < 0) panic("fork(): 111111\n");
    if (sys_env_set_pgfault_upcall(envid, _pgfault_upcall) < 0)
```

```
            panic("fork(): 22222222\n");
        if (sys_env_set_status(envid, ENV_RUNNABLE) < 0) panic("
            fork(): 3333333\n");
44      return envid;
    }
```

这个函数先创建一个进程，然后子进程检查加入队列就返回。如果是父进程，就把出了异常栈之外的内容全部拷贝到子进程，然后初始化一下异常栈，最后设置一下缺页处理函数，将进程设置成RUNNABLE。

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

这一部分实现多进程间进程通信。

4.1 Clock Interrupts and Preemption

现在我实现的JOS中的进程都是非抢占式的，如果有一个进程，就像spinlock这样，一直霸占CPU，那么其他进程和内核的程序都将无法重新获得CPU的使用权，所以我必须引进一个抢占的概念。最简单的实现方式就是使用时钟中断，当中断发生的时候，内核强行接管并且进行调度，这其中就将剥夺当前进行的程序的使用权，发生抢占。

4.1.1 Interrupt discipline

外部中断（又叫设备中断）有16种，0-15，所以这儿有16种IRQ号，0-15，这个从IRQ到IDT entry 的映射不是固定的。而是从IRQ_OFFSET through IRQ_OFFSET+15。这个是设定在inc/trap.h里面。

JOS里面，这一切和XV6相比做了很多的简化，在内核中，外部设备的中断总是禁止的，用户空间内外部设备的中断时允许的。外部设备的中断是由FL_IF flag bit of the %eflags register (see inc/mmu.h)控制的，当这一位被设置了，那么我们可以开启外部中断，JOS 为了简化，只允许在进入和离开用户模式时做保存和重置 %eflags的时候去修改这一位。

我们必须要确保FL_IF是被设置的在用户状态的时候，这样我们才能实现当外部中断到来的时候让处理器调用我们的中断处理代码。

注意，外部中断到来时，CPU是不压入错误码的！

Exercise 13. 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 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.

IRQ_OFFSET查找出来这个值是32，所以中断的函数就是vector32到vector47。所以我到kern/trapentry.S去添加一下就好了。

```
        TRAPHANDLER(vector32, 32)
    TRAPHANDLER(vector33, 33)
    TRAPHANDLER(vector34, 34)
    TRAPHANDLER(vector35, 35)
    TRAPHANDLER(vector36, 36)
    TRAPHANDLER(vector37, 37)
    TRAPHANDLER(vector38, 38)
    TRAPHANDLER(vector39, 39)
    TRAPHANDLER(vector40, 40)
    TRAPHANDLER(vector41, 41)
    TRAPHANDLER(vector42, 42)
    TRAPHANDLER(vector43, 43)
    TRAPHANDLER(vector44, 44)
    TRAPHANDLER(vector45, 45)
    TRAPHANDLER(vector46, 46)
    TRAPHANDLER(vector47, 47)
```

这里是继上个lab的challenge之后，我把vectors变成了一个全局变量，且是在data中，这样每一个vectors的内容都是一个指向中断处理函数的指针。这其中所有的IRQ中断都是不压入错误码的，我在上面的text中已经定义了，判定的相关内容。不需要做相关的修改了。然后我去kern/trap.c中，设置相应的中断描述符。

```
    for (i = 0; i <= 0x30; ++i) {
        switch (i) {
            case T_BRKPT:
            case T_SYSCALL:
            case IRQ_OFFSET + IRQ_TIMER:
                SETGATE(idt[i], 0, GD_KT, vectors[i], 3);
                break;
            default:
                SETGATE(idt[i], 0, GD_KT, vectors[i], 0);
```

```
10          }
        }
```

这其中， 我添加了一行，就是

```
    case IRQ_OFFSET + IRQ_TIMER:
```

因为这个也是用户态可触发的所以把他加在这里。
然后去kern/env.c中，确保用户进程总是开着中断。

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

EASY!

4.1.2 Handling Clock Interrupts

这里要求我引进周期性的时钟中断，强制使得控制权返回给内核。

Exercise 14. 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.

```
1             // Handle clock interrupts. Don't forget to
                  acknowledge the
      // interrupt using lapic_eoi() before calling the scheduler
          !
3     // LAB 4: Your code here.
      if (tf->tf_trapno == IRQ_OFFSET + IRQ_TIMER) {
5         lapic_eoi();
          sched_yield();
7         return;
      }
```

这里就是判断如果是时钟中断，就先执行 lapic_eoi()确认中断，然后进程调度。

4.2 Inter-Process communication (IPC)

这里我要实现一个简单的IPC进程通信。

4.2.1 IPC in JOS

实现两个系统调用sys_ipc_recv and sys_ipc_try_send. 实现two library wrappers ipc_recv and ipc_send.

用户进程可以用来用IPC相互发送的信息包含两块： a single 32-bit value, and optionally a single page mapping。

4.2.2 Sending and Receiving Messages

一个进程会调用一个系统调用sys_ipc_recv来获取信息， 操作系统会给这个进程降级，并且不会在调用这个进程，直到这个信息被收到了。当一个进程正在调用这个函数的时候，任何其他的进程可以给这个进程发送消息，因为发送消息并不会是其他进程出错。

sys_ipc_try_send加进程ID可以给某个特定的进程发送消息，如果该进程也调用了sys_ipc_recv并且目前没有返回值，那么发送和接受成功，返回0，否则返回 -E_IPC_NOT_RECV去暗示目标进程现在不希望接受信息。

A library function ipc_send就是将系统调用包装了一下，这样可以防止反复调用这个send的系统调用。

需要提出的是，JOS中，JOS中一个消息是一个32位整数。

当一个进程想要发送消息时，为保险起见，它会先执行sys_ipc_try_send()检查是否可以发送消息;如果检查通过，那么它会调用sys_ipc_send()传递信息。 相应的，有sys_ipc_try_recv()和sys_ipc_recv()。

4.2.3 Transferring Pages

页传输， 当一个进程调用sys_ipc_recv with a valid dstva parameter (below UTOP)， 表明，他需要一个页的映射。 如果发送的进程发送一个页，那么这个页会映射到接受者地址空间中的虚拟地址dstva处，如果接受者这个地址空间已经有页了，那么这个页会被解除映射。

如果这是恰好一个进程调用sys_ipc_try_send和一个源地址srcva，这意味着发送方想要发送一个当前map在srcva地址上的页，并且权限是perm。这样，发送方就会和接收方共享这个页。

IPC完成之后，接受进程设置新的值：env_ipc_perm，在env中，如果这个值为0，那么没有页能被接受。

4.2.4 Implementing IPC

Exercise 15. 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. `user/primes` will generate for each prime number a new environment until JOS runs out of environments. You might find it interesting to read `user/primes.c` to see all the forking and IPC going on behind the scenes.

sys_ipc_recv:

```
// Block until a value is ready.  Record that you want to receive
// using the env_ipc_recving and env_ipc_dstva fields of struct Env,
// mark yourself not runnable, and then give up the CPU.
//
// If 'dstva' is < UTOP, then you are willing to receive a page of data.
// 'dstva' is the virtual address at which the sent page should be mapped.
//
// This function only returns on error, but the system call will eventually
// return 0 on success.
// Return < 0 on error.  Errors are:
//      -E_INVAL if dstva < UTOP but dstva is not page-aligned.
static int
sys_ipc_recv(void *dstva)
{
    // LAB 4: Your code here.
    if ((uintptr_t)dstva < UTOP && (uintptr_t)dstva % PGSIZE)
        return -E_INVAL;
    struct Env *e = curenv;
    e->env_ipc_recving = 1;
    e->env_ipc_dstva = dstva;
    e->env_ipc_perm = 0;
    e->env_status = ENV_NOT_RUNNABLE;
    return 0;
}
```

解决一下注释中的问题： (Hint: does the sys_ipc_recv function ever actually return?) 我觉得这是返回的，因为这个函数是在内核中调用的，内核将某个进程设置成ENV_NOT_RUNNABLE, 这个进程被阻塞了，但是这个函数会在内核中迅速的返回。

sys_ipc_try_send：

```
1     // Try to send 'value' to the target env 'envid'.
  // If srcva < UTOP, then also send page currently mapped at '
      srcva',
3 // so that receiver gets a duplicate mapping of the same page.
  //
5 // The send fails with a return value of -E_IPC_NOT_RECV if the
  // target is not blocked, waiting for an IPC.
7 //
  // The send also can fail for the other reasons listed below.
9 //
  // Otherwise, the send succeeds, and the target's ipc fields
      are
11 // updated as follows:
  //    env_ipc_recving is set to 0 to block future sends;
13 //    env_ipc_from is set to the sending envid;
  //    env_ipc_value is set to the 'value' parameter;
15 //    env_ipc_perm is set to 'perm' if a page was transferred,
      0 otherwise.
  // The target environment is marked runnable again, returning 0
17 // from the paused sys_ipc_recv system call.  (Hint: does the
  // sys_ipc_recv function ever actually return?)
19 //
  // If the sender wants to send a page but the receiver isn't
      asking for one,
21 // then no page mapping is transferred, but no error occurs.
  // The ipc only happens when no errors occur.
23 // Returns 0 on success, < 0 on error.
  // Errors are:
25 //      -E_BAD_ENV if environment envid doesn't currently exist
      .
  //              (No need to check permissions.)
27 //      -E_IPC_NOT_RECV if envid is not currently blocked in
      sys_ipc_recv,
  //              or another environment managed to send first.
29 //      -E_INVAL if srcva < UTOP but srcva is not page-aligned.
  //      -E_INVAL if srcva < UTOP and perm is inappropriate
31 //              (see sys_page_alloc).
  //      -E_INVAL if srcva < UTOP but srcva is not mapped in the
       caller's
33 //              address space.
  //      -E_INVAL if (perm & PTE_W), but srcva is read-only in
      the
35 //              current environment's address space.
  //      -E_NO_MEM if there's not enough memory to map srcva in
      envid's
37 //              address space.
      static int
39 sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva,
      unsigned perm)
  {
41     // LAB 4: Your code here.
      struct Env *e;
43     struct PageInfo *p;
```

```
    pte_t *pte;
    perm &= PTE_SYSCALL;
    if (envid2env(envid, &e, 0) == -E_BAD_ENV) return -
        E_BAD_ENV;
    if (!e->env_ipc_recving) return -E_IPC_NOT_RECV;
    if ((uintptr_t)srcva < UTOP) {
        if ((uintptr_t)srcva % PGSIZE) return -E_INVAL;
        if ((perm & (PTE_P | PTE_U)) != (PTE_P | PTE_U)) return
            -E_INVAL;
        if (!(p = page_lookup(curenv->env_pgdir, srcva, &pte)))
            return -E_INVAL;
        if (perm & PTE_W & ~*pte) return -E_INVAL;
    }
    e->env_ipc_recving = 0;
    e->env_ipc_from = curenv->env_id;
    e->env_ipc_value = value;
    if ((uintptr_t)e->env_ipc_dstva < UTOP && (uintptr_t)srcva
        < UTOP) {
        if (page_insert(e->env_pgdir, p, e->env_ipc_dstva, perm
            ) == -E_NO_MEM) {
            return -E_NO_MEM;
        }
        e->env_ipc_perm = perm;
    }
    e->env_status = ENV_RUNNABLE;
    e->env_tf.tf_regs.reg_eax = 0;
    return 0;
}
```

按照注释写就可以了。

ipc_recv

```
    // Receive a value via IPC and return it.
// If 'pg' is nonnull, then any page sent by the sender will be
    mapped at
//      that address.
// If 'from_env_store' is nonnull, then store the IPC sender's
    envid in
//      *from_env_store.
// If 'perm_store' is nonnull, then store the IPC sender's page
    permission
//      in *perm_store (this is nonzero iff a page was
    successfully
//      transferred to 'pg').
// If the system call fails, then store 0 in *fromenv and *perm
    (if
//      they're nonnull) and return the error.
// Otherwise, return the value sent by the sender
//
// Hint:
//   Use 'thisenv' to discover the value and who sent it.
//   If 'pg' is null, pass sys_ipc_recv a value that it will
    understand
```

```
16 //   as meaning "no page".  (Zero is not the right value, since
       that's
   //   a perfectly valid place to map a page.)
18     int32_t
   ipc_recv(envid_t *from_env_store, void *pg, int *perm_store)
20 {
       // LAB 4: Your code here.
22     int r;
       if (!pg) pg = (void*)UTOP;
24     if ((r = sys_ipc_recv(pg)) < 0) {
           if (from_env_store) *from_env_store = 0;
26         if (perm_store) *perm_store = 0;
           return r;
28     }
       if (from_env_store) *from_env_store = thisenv->env_ipc_from
           ;
30     if (perm_store) *perm_store = thisenv->env_ipc_perm;
       return thisenv->env_ipc_value;
32 }
```

如果系统调用成功返回，就会把相应的至全部赋成0，如果不成功，就会进入下面的相应的赋值。 ipc_send

```
       // Send 'val' (and 'pg' with 'perm', if 'pg' is nonnull) to
            'toenv'.
 2 // This function keeps trying until it succeeds.
   // It should panic() on any error other than -E_IPC_NOT_RECV.
 4 //
   // Hint:
 6 //   Use sys_yield() to be CPU-friendly.
   //   If 'pg' is null, pass sys_ipc_try_send a value that it
       will understand
 8 //   as meaning "no page".  (Zero is not the right value.)
       void
10 ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
   {
12     // LAB 4: Your code here.
       int r;
14     while ((r = sys_ipc_try_send(to_env, val, (pg ? pg : (void
           *)UTOP), perm))) {
           if (r != -E_IPC_NOT_RECV) panic("ipc_send()\n");
16         sys_yield();
       }
18 }
```

这个就是不停的try send, 如果是其他错误就panic，如果只是-E_IPC_NOT_RECV，就继续循环尝试发送。

This completes the Lab!!!!! 我要推选它为耗时最长的lab!

```
Part A score: 5/5

faultread: OK (1.3s)
faultwrite: OK (1.3s)
faultdie: OK (1.3s)
faultregs: OK (1.4s)
faultalloc: OK (1.5s)
faultallocbad: OK (1.4s)
faultnostack: OK (1.4s)
faultbadhandler: OK (1.4s)
faultevilhandler: OK (1.3s)
forktree: OK (1.4s)
Part B score: 50/50

spin: OK (2.3s)
    (Old jos.out.spin failure log removed)
stresssched: OK (1.7s)
    (Old jos.out.stresssched failure log removed)
sendpage: OK (2.1s)
    (Old jos.out.sendpage failure log removed)
pingpong: OK (1.4s)
    (Old jos.out.pingpong failure log removed)
primes: OK (4.5s)
    (Old jos.out.primes failure log removed)
Part C score: 25/25

Score: 80/80
```