Lab 4

一.实验内容：

在 lab3 中我们完成了对线程的创建，但当时只是单个 CPU，而且只是一个线程;在 lab4 中我们扩展 jos 使其支持多处理器，另外还需要实现协作式轮转调度(cooperative round-robin scheduling)，然后实现 JOS 的系统调用，允许用户环境来创建新的线程。

二. Application Processor Bootstrap

1.CUP 在 OS 启动时的分类：(BSP,AP)

根据实验说明的提示，我们为 JOS 添加“symmetric multiprocessing”(SMP)的支持，SMP 支持多 CPUs 有同等的机会获得系统资源(如内存，IO 总线)。虽然在 SMP 中所有的 CPUs 功能相同，但是在启动过程中分为两类：

the bootstrap processor(BSP):负责系统的初始化和启动操作系统

the application processor(APs):在 BSP 初始化并启动系统后，由 BSP 唤醒

注：哪个处理器是 BSP 由硬件和 BIOS 决定，lab4 之前，JOS 代码都运行在 BSP 上。

2.BSP 激活 APs 的过程

根据实验提示：在 SMP 系统中每个 CPU 都有一个 LAPIC（局部应用程序中断控制）单元，正是通过这个单元，BSP 将激活信号传递给每个 AP。

作业 1：

首先分析 boot_aps 和 mp_main()

其中涉及函数 mp_init(),lpaic.c 中的多个函数，下面详细分析：

(1)：在启动 APS 之前，BSP 首先应该收集有关多处理机系统信息，如 CPU 的总数目，它们的 APIC ID 和 LAPIC 单元的 MMIO(memory-mapped I/O).JOS 实验中，在 kern/mpconfig.c 的 mp_init()函数读取 BIOS 的 MP configuration table 获得这些信息。

在 kern/init.c 的 i386_init()函数中，可以知道，在 boot_aps()之前，我们首先执行 mp_init() lapic_init()。在 kern/mpentry.S 中会用到 mp_init()，所以这里线分析 mp_init().

如下是 mp_init()的局部：**该函数启动 BSP**

```
void
mp_init(void)
{
    struct mp *mp;
    struct mpconf *conf;
    struct mpproc *proc;
    uint8_t *p;
    unsigned int i;

    bootcpu = &cpus[0];
    if ((conf = mpconfig(&mp)) == 0)
        return;
    ismp = 1;
    lapic = (uint32_t *)conf->lapicaddr;

    for (p = conf->entries, i = 0; i < conf->entry; i++) {
        switch (*p) {
```

```
    case MPPROC:
        proc = (struct mpproc *)p;
        if (proc->flags & MPPROC_BOOT)
            bootcpu = &cpus[ncpu];
        if (ncpu < NCPU) {
            cpus[ncpu].cpu_id = ncpu;
            ncpu++;
        } else {
            cprintf("SMP: too many CPUs, CPU %d disabled\n",
                proc->apicid);
        }
        p += sizeof(struct mpproc);
        continue;
```

其中 struct mp(在 kern/mpconfig.c)中定义，该结构包含了配置表(configuration table)的入口地址
mpconfig():该函数搜索指定 MP 的 configuration table。
当找到 configuration table 后就就加载 mpproc(processor table entry)之后在 cpus 数组中记录该 cpu 的 id，在程序的末尾(上图为显示)标记 bootcpu 为 CPU_STARTED 状态。
其中 lapic 是在 kern/lapic.c 中定义的指向 uint32_t 的变量，将其初始化为配置表中的 lapicaddr

(2)：函数 boot_aps()(kern/init.c)引导 APS 启动，APS 在实模式启动，和 bootloader 引导过程非常像。boot_aps()复制 entry code(kern/mpentry.S)到实模式可寻址到的一处内存地址，之后 boot_aps()通过发送 STARTUP IPS 到 AP 对应的 LAPIC，然后启动该 AP，将其设置为保护模式。AP 启动后会发送 CPU_STARTED 到 BSP，BSP 就会接着启动下一个 AP

函数 boot_aps():
在分析这个函数之前，我们先看一下，在此之前之前做了哪些准备工作

```
// Start the non-boot (AP) processors.
static void
boot_aps(void)
{
    extern unsigned char mpentry_start[], mpentry_end[];
    void *code;
    struct Cpu *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)
            ;
    }
}
```

在 kern/init.c 的 i386_init()中，可以看到 lab4 在调用 boot_aps()之前，调用了 mp_init()函数，lapic_init()和 pic_init()函数

该函数的其他函数：

cpunum():函数体在 kern/lapic.c 中定义，其中 lapic 在 mp_init()中被初始化为 lapicaddr.

```
int
cpunum(void)
{
    if (lapic)
        return lapic[ID] >> 24;
    return 0;
}
```

其中的 ID 为宏定义的 8。这是因为在 kern/cpu.h 中定义了 NCPU 为 8,就是说 JOS 最多可以支持 8 个 cpus 协同工作。**cpunun()返回 BSP 的 num。**

过程：mpentry 加载到 code 中，然后对于每一个 cpu 首先检测是否被启动过（针对于 bsp,该处理器在 mp_init()就被启动了，如果没有则为其分配栈，然后调用 lapic_startap()，该函数执行 entry code。entry code 即 kern/mpentry.S.在 kern/mpentry.S 中，我们为 AP 创建页表，开启分页模式，切换到在对应 AP 的栈，然后调用 mp_main()。mpmain()初始化 lapic,加载 IDT
然后调用 trap_init_percpu()初始化 CPU 的 TSS(中断时定义内核栈)。xchg()告知 BPS 启动完成。

更改 kern/page_init()使得 MPENTRY_PADDR 不被加入到 page_free_list 中。

代码如下：

```
}else if(i < MPENTRY_PADDR/PGSIZE)
    {
        pages[i].pp_ref = 0;
            pages[i].pp_link = page_free_list;
            page_free_list = &pages[i];
    }else if(i==MPENTRY_PADDR/PGSIZE){
    pages[i].pp_ref=0;
    pages[i].pp_link=NULL;
    continue;
}else if(i<npages_basemem){
```

在原来的基础上那个减去 MPENTRY_PADDR 这个页。

分析：在 memlayout.c 中 MPENTRY_PADDR 被宏定义为 0x7000,IOPHYSMEM 为 0x0A0000
所以可以知道 MPENTRY_PADDR 在 IOPHYSMEM 之前。

结果：

```
check_page_free_list() succeeded.
check_page_alloc() succeeded!
check_page() succeeded!
```

问题 1：

mpentry.S 中宏定义 MPBOOTPHYS 有什么作用，为什么其对于 mpentry.S 是必须的，而 boot.S 就不需要该宏。

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

答：

作用：该宏定义是将 s(va)映射到相对 MPENTRY_PADDR(pa)的偏移量上，即将 s 映射为 pa。

为什么是 mpentry.S 必须的：

回忆 lab1 的加载地址和连接地址，程序将执行代码(如 kernel,mp code)加载到指定的物理地址空间

中,而链接器会将其链接到较高的地址空间中。查看 kern/init.c 可以知道，此时程序都是在虚拟地址空间中，因此如果此时我们要在实模式(.code16)下启动 APs，就需要将虚拟地址映射到相应的物理地址上，而 mpentry_start 也是虚拟地址，它表示 MP_ENTRY 的虚拟地址，因此我们只要计算其他虚拟地址相对于 mpentry_start 的偏移量，在加上 MPENTRY_PADDR,就可以得到对应的物理地址。

为什么 boot.s 不需要：

因为在 boot.s 时，我们还没有建立虚拟地址，只是使用段转换来使 va 映射到 pa。

三.Per-CPU State and Initialization

实验内容介绍：

在写多处理器的 OS 时,能够区分每个处理器的私有 per-CPU state 和整个系统共享的 global state 是很重要的。kern/cpu.h 定义了大部分的 per-CPU state,包括 struct Cpu(存储 per-CPU variables) 。 cpunum()返回调用它的 CPU 的 ID,这可以用来作为如 cpus array 的索引。另外,宏 thiscpu 是当前 CPU 的 cpu struct 的简写。

作业 2：

修改 kern/pmap.c 中的 mem_init_mp()，该函数的作用是为每一个 CPU 分配内核栈。

分析：因为多 CPUs 可以同时捕捉内核，我们需要为每一个处理器分配独立的 kernel stack,防止内核之间的相互干扰。根据实验提示，我们将 CPU0 的内核栈依然从 KSTACKTOP 向下增长，CPU1 的栈在 CPU0 栈底的 KSTKGAP(保护页)之后开始，CPU2....

使用 boot_map_region()将虚拟地址映射到相应的物理地址上。

代码如下：

```
    // LAB 4: Your code here:
    int num;//记录cpu num
    uintptr_t stacktop=KSTACKTOP;//记录当前标号为num的cpu的栈顶
    for(num=0;num<NCPU;num++)
    {
      boot_map_region(kern_pgdir, stacktop - KSTKSIZE, KSTKSIZE, PADDR(percpu_ksta
cks[num]), PTE_W | PTE_P);
        stacktop -= (KSTKSIZE + KSTKGAP);
    }

}
```

其中 percpu_kstacks[NUM][KSTKSIZE]是在 kern/mpconfig.c 中定义，在 kern/init.c 的 boot_aps 中初始化为各个处理器的栈。

代码中 stacktop-=KSTKSIZE+KSTKGAP 作用是为栈设置 guard page(保护页)。

作业 3.

lab3 时我们在 kern/trap.c 的 trap_init_percpu()函数中为 BSP 初始化了 TSS 和 TSS descriptor. 此时我们需要修改代码，使其能够对所有的 CPUs 都起作用。

分析：

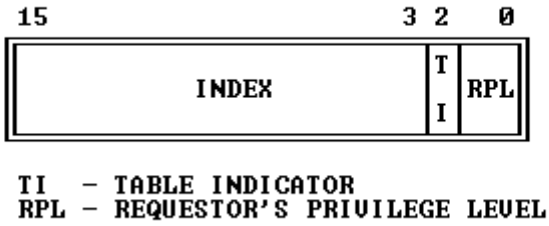
Per-CPU TSS and TSS descriptor

1.每一个 cpu 的 task state segment (TSS)被用来指定每一个 CPU 的内核栈存在的地方,The TSS for CPU i is stored in cpus[i].cpu_ts,相应的 TSS descriptor 定义在 gdt[(GD_TSS0 >> 3) + i]。 kern/trap.c 的全局变量 ts 将不再使用。

2.在 kern/trap.c 中我们看到了 JOS 为我们写了加载 CPU0 的代码，所以我们根据注释的提示修改即可。

首先使用 thiscpu 变量来指向当前的 CPU，使用 thiscpu->cpu_id 来获得 CPU 的 ID。
这里我们先介绍一下关于段选择子(segment selector),在下面网址(jos 介绍过该网址)：
https://pdos.csail.mit.edu/6.828/2010/readings/i386/s05_01.htm#fig5-6
这里详细介绍了 segment selector,在 lab3 中使用过。

Figure 5-6. Format of a Selector



TI – TABLE INDICATOR
RPL – REQUESTOR'S PRIVILEGE LEVEL

上图是 segment selector 的格式，前 13 位是 gdt(全局描述表)的索引(index)。

代码：

```
    // LAB 4: Your code here:
    //使用thiscpu获得当前cpu
    thiscpu->cpu_ts.ts_esp0=KSTACKTOP-thiscpu->cpu_id*(KSTKSIZE+KSTKGAP);
    thiscpu->cpu_ts.ts_ss0= GD_KD;
    // Setup a TSS so that we get the right stack
    // when we trap to the kernel.
    //ts.ts_esp0 = KSTACKTOP;
    //ts.ts_ss0 = GD_KD;

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

    // Load the TSS selector (like other segment selectors, the
    // bottom three bits are special; we leave them 0)
    ltr(GD_TSS0+(thiscpu->cpu_id<<3));

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

根据上面的分析，我们可以理解为什么要左移 3 位。这是因为，GD_TSS0 是在 inc/memlayout.c 中宏定义的 cpu0 的段选择子，然而选择子的前 15 才是 index，所以我们需要右移 3,即取其前 13 位。同理可以解释 ltr(GD_TSS)+(thiscpu->cpu_id<<3))中左移三位的含义。

```
SMP: CPU 0 found 4 CPU(s)
enabled interrupts: 1 2
SMP: CPU 1 starting
SMP: CPU 2 starting
SMP: CPU 3 starting
[00000000] new env 00001000
[00000000] new env 00001001
[00000000] new env 00001002
[00000000] new env 00001003
[00000000] new env 00001004
[00000000] new env 00001005
[00000000] new env 00001006
[00000000] new env 00001007
[00000000] new env 00001008
```

使用 make qemu-nox CPUS=4 的结果：

四.Locking

作业 4

在作业提示的地方加上 lock_kernel()和 unlock_kernel()

分析：在这一节中我们使用对内核加锁的方式来实现多个 cpu 轮替使用内核资源。简单来说就是当一个 cpu 占用内核资源之前需要对内核加锁，这样其他 cpu 在当前 cpu 没有解锁之前就不能使用内核资源。

在 kern/spinlock.h 中声明了 kern_lock 全局变量，并且提供了 lock_kernel()和 unlock_kernel()来对 kern_lock 进行加锁了解锁。

根据实验文档提示，我们需要在一下几个地方加锁和解锁：

(1):In i386_init(),在 BSP 唤醒其他 CPUS 之前取得锁

(2):In mp_main(),在 AP 初始化后获得锁,然后调用 sched_yield()运行在该 AP 上的环境

(3):In trap(),当在用户态时发生 trap 时获得锁,为了判断 trap 发生在用户态还是内核态,检查 tf_cs 的低位。

(4):In env_run(),切换用户模式之前释放锁

依次找到上述函数添加代码：

i386_init() (kern/init.c):

```
// Acquire the big kernel lock before waking up APs
// Your code here:
lock_kernel();
// Starting non-boot CPUs
boot_aps();
```

mp_main() (kern/init.c) :

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

trap() (kern/trap.c) :

```
    // LAB 4: Your code here.
    assert(curenv);
    lock_kernel();
```

env_run() (kern/env.c) :

```
    lcr3(PADDR(curenv->env_pgdir));
    unlock_kernel();
    env_pop_tf(&(curenv->env_tf));
```

问题 2：

这 spinlock 情况下，同一时间只有一个 cpu 占用内核空间，为什么还需要为每一个 cpu 分配一个内核栈。

分析：在回答这个问题之前，我们需要知道内核栈的作用。
在 lab2 的“控制权转移：异常和中断”中介绍 TSS 时讲到了内核栈的作用。当 x86 处理器响应一个中断或陷阱时，会引起从用户态到内核态权限的改变，也 会切换到内核的栈。 TTS 指定了新栈的 the segment selector and address 。处理器将 SS, ESP, EFLAGS, CS, EIP, and an optional error code 压入新栈，然后从 IDT 加载 CS 和 EIP(中断处理程序), 设置 SS 与 ESP 为新栈地址。
从上述分析可以知道，内核栈是在中断发生时，保存中断时程序的状态信息和中断处理程序的地址。结合作业 4，我们知道在中断发生时，lock_kernel()是发生在 trap()中，也就是在状态数据压栈完成之后。如果此时其他 cpu 抢占内核空间，执行其他进程，就会导致前 cpu 内核栈中的中断状态信息的丢失。致使后续的中断处理和中断恢复无法正常进行。

五.Round-Robin Scheduling

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:
分析：该部分要求我们修改 JOS 内核，使得它不总是运行 idle environment，而是与运行在我们之前创建的环境中，这里所使用的是轮转调度的工作方式。
1.程序刚启动的时候，cpu 总是运行在 idle environment 上(user/idle.c)，这个程序的目的就是使 cpu 空转。
2.函数 sched_yield()(kern/seched.c):负责选择一个新的环境来运行，它以循环遍历的方式从上那个一个 running 态的进程之后搜索 envs[]数组（如果上一个 running 进程为 envs[]最后一项，或者没有 running 进程，则从 envs[]首部开始遍历），选择第一个状态为 ENV_RUNNABLE 的进程使其进入 running。而且值得注意的是，sched_yield()可以根据进程的 ENV_TYPE_IDLE 判断该进程是否为 idle 进程，只有没有其他进程运行时才选择 idle 进程。
3.sched_yield()不会在两个处理器上同时处理一个进程，它可以根据当前进程是否为 running 来判断该进程是否已在其他 cpu 上运行。
4.下面我们来仔细查看一下 sched_yield()函数，根据该函数的注释提示，我们从 envs[]数组的首部开始遍历，寻找 runnale 进程。如果当前进程依然是 running 态，就继续 env_run()该进程，如果为 runnable 也 running。

```
// LAB 4: Your code here.
idle=(curenv==NULL||curenv-envs==NENV-1)?&envs[0]:curenv+1;
for(i=0;i<NENV;i++){
   if(idle==curenv){
    if(idle->env_status==ENV_RUNNING)
        env_run(idle);
 }else if(idle->env_status==ENV_RUNNABLE)
        env_run(idle);
  idle = (idle - envs == NENV - 1) ? &envs[0] : idle + 1;
}
```

5.因为 sched_yield()为系统调用(48 号中断)，在 trap_dispache()(kern/trap.c)中我们调用了 syscall()(kern/syscall.c)函数，该函数根据 trapno 来判断是哪一种调用;我们进入 syscall.c 函数添加 sched_yield 即可。

```
case SYS_yield:
sys_yield();
break;
```

6.在 kern/init.c 中创建线个线程，都运行 user/yield.c.

```
// 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);
```

运行结果：

```
Hello, I am environment 00001000.
Hello, I am environment 00001001.
Hello, I am environment 00001002.
Hello, I am environment 00001003.
Hello, I am environment 00001004.
Hello, I am environment 00001005.
Hello, I am environment 00001006.
Hello, I am environment 00001007.
Hello, I am environment 00001008.
Back in environment 00001008, iteration 0.
Back in environment 00001008, iteration 1.
Back in environment 00001008, iteration 2.
Back in environment 00001008, iteration 3.
Back in environment 00001008, iteration 4.
```

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?

分析：我们先来看一下 ern_run()函数，该函数将指定进程置为 running，然后加载该进程的 env_pgdir 页目录进 lcr3 页表寄存器。因为进程都属于内核空间，即它们的地址都在 KERNBASE 之上，而在前面的实验中我们知道 jos 将 0 ~ 256M 物理空间映射到了高 256M 空间中，所以该部分地址对应物理地址对于每一个进程来说都是一样的。

2.6 System Calls For Environment Creation

一.实验介绍：

虽然我们在作业五中可以运行和切换多个用户环境态，但是受限于只能运行那性由内核创建的环境。（即在 kern/init.c 中使用 ENV_CREATE 创建的）现在需要实现必要的系统调用以便使用户环境可以创建和启动其他新的用户环境。

作业 6.

Unix 提供了 fork()函数作为创建进程的原型，Unix 的 fork()函数 copy 整个 calling process(其父亲进程)的地址空间来创建新的进程，唯一可观察到的不同是 process IDs 和 parent process IDs.在父进程中，fork()函数返回子进程的 process ID,而在孩子进程中 fork()返回 0.

下面我们需要完成一下的函数：

1.sys_exfork():

该系统调用创建一个几乎空白的新环境:没有东西映射到它地址空间的用户部分,也不能运行。在 sys_exofork 调用时,这个新环境和它父环境有一样的 registers.

In the parent, sys_exofork 返回新环境的 envid_t (or a negative error code if the environment

allocation failed).
In the child, it will return 0(因为子环境被标记为不可用,sys_exofork 不会返回,除非它的父环境把它标记为可用)

```
// LAB 4: Your code here.
//根据注释我们创建一个寄存器和当前进程(父进程)一样的进程
//使用env_alloc()创建进程
struct Env* newenv;
int ret;
if((ret=env_alloc(&newenv,sys_getenvid()))<0)//失败时返回-E_NO_FREE_ENV
    return ret;
//设置为not runnable
newenv->env_status=ENV_NOT_RUNNABLE;
//copy trapframe
newenv->env_tf=curenv->env_tf;
//makethechildenv'sreturnvaluezero
newenv->env_tf.tf_regs.reg_eax=0;//系统调用返回值在eax中
return newenv->env_id;
//panic("sys_exofork not implemented");
```

分析：根据注释的提示，我们需要使用 env_alloc()来创建一个新的进程，env_alloc()函数接受两个参数，第一个参数为新进程地址的引用，第二个是父进程 ID（当前进程）。
然后将新进程的状态置为 ENV_NOT_RUNNABLE，tf 栈设为父进程的寄存器状态。

2.sys_env_set_status:
static int sys_env_set_status(envid_t envid, int status)
设置某个环境的状态为 ENV_RUNNABLE or ENV_NOT_RUNNABLE,这个系统调用一般被用来标记一个新环境已经准备好可用了(当它的地址空间和寄存器都已经初始化好)

```
// LAB 4: Your code here.
//当进程状态既不是ENV_RUNNABLE或者ENV_NOT_RUNNABLE时返回-E_INVAL
if(status!=ENV_RUNNABLE&&status!=ENV_NOT_RUNNABLE)
    return -E_INVAL;
struct Env* e;
int ret=envid2env(envid,&e,1);//成功则返回0
//不成功返回-E_BAD_ENV
if(ret)
  return ret;
e->env_status=status;
return 0;
//panic("sys_env_set_status not implemented");
```

分析：根据注释的提示，该函数只对状态为 ENV_RUNNABLE 和 ENV_NOT_RAUNNBALE 所以对于其他的状态直接返回-E_INVAL;
如果当前要设置状态的进程不存在或者当前调用进程对该子进程没有权限，那么就返回-E_BAD_ENV;
其中检查进程是否存在以及父进程对其是否由权限都在 kern/env.c 的 envid2env()中完成。

3.sys_page_map:
复制一个环境的 page mapping(not the contents of a page!)到另一个环境,使得新老环境都指向物理内存的同一页。

```
    // LAB 4: Your code here.
    struct Env* srcenv;
    struct Env* dstenv;
    if(!envid2env(srcenvid,&srcenv,1))
        return -E_BAD_ENV;
    if(!envid2env(dstenvid,&dstenv,1))
        return -E_BAD_ENV;

    //  -E_INVAL if srcva >= UTOP or srcva is not page-aligned,
    //       or dstva >= UTOP or dstva is not page-aligned.
    if(srcva>=(void*)UTOP||ROUNDDOWN(srcva,PGSIZE)!=srcva)
        return -E_INVAL;
    if(dstva>=(void*)UTOP||ROUNDDOWN(dstva,PGSIZE)!=dstva)
        return -E_INVAL;

    //-E_INVAL is srcva is not mapped in srcenvid's address space.
    pte_t* pte;
    struct Page* pg;
    pg=page_lookup(srcenv->env_pgdir,srcva,&pte);
    if(!pg) return -E_INVAL;

    //-E_INVAL if perm is inappropriate (see sys_page_alloc)
    int perm_flag=PTE_U|PTE_P;
    if((perm&perm_flag)!=perm_flag)
        return -E_INVAL;

    //-E_INVAL if (perm & PTE_W), but srcva is read-only in srcenvid's addre
ss space.
    if(((*pte&PTE_W) == 0) && (perm&PTE_W)) return -E_INVAL;

    //-E_NO_MEM if there's no memory to allocate any necessary page tables.
    int ret = page_insert(dstenv->env_pgdir, pg, dstva, perm);
    return ret;
```

分析：Map the page of memory at 'srcva' in srcenvid's address space at 'dstva' in dstenvid's address space with permission 'perm'. Perm has the same restrictions as in sys_page_alloc, except that it also must not grant write access to a read-only page.
根据注释的提示，该函数将源进程的地址空间映射到目的进程的地址空间，我们知道，在刚开始创建新进程的时候，子进程跟父进程暂时拥有相同的寄存器和地址空间。
根据注释一步步写出代码：
（1）当源进程和目的进程不存在时，返回-E_BAD_ENV。
（2）如果 srcva 或者 dstva 超过了 UTOP 返回-E_INVAL
（3）如果 srcva 或者 dstva 没有 4K 对齐时返回-E_INVAL
（4）如果 srcva 不再 srcenv 的地址空间返回-E_INVAL
（5）如果调用进程不具备足够权限(PTE_P,PTE_U)返回-E_INVAL
（6）如果 dstva 此时没有页表项且此时没有足够的空间分配给 dstva 则返回-E_NO_MEM

4.sys_page_unmap:
Unmap a page mapped at a given virtual address in a given environment.
以上所有的系统调用都接受 environment IDs,JOS 内核支持简便写法 0 表示"the current

environment”,简便写法的实现在 envid2env() in kern/env.c.

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

    // LAB 4: Your code here.
//  -E_INVAL if va >= UTOP, or va is not page-aligned.
    if (va>=(void*)UTOP || ROUNDDOWN(va,PGSIZE)!=va)
        return -E_INVAL;
    struct Env *e;
//  -E_BAD_ENV if environment envid doesn't currently exist,
//      or the caller doesn't have permission to change envid.
    int ret = envid2env(envid, &e, 1);
    if (ret) return ret;
    page_remove(e->env_pgdir, va);
    return 0;

    panic("sys_page_unmap not implemented");
}
```

分析：根据注释提示，写出该函数。
（1）如果 va 大于 UTOP，或者 va 没有 4K 对齐则返回-E_INVAL
（2）如果 envid 不存在或者当前进程没有足够的权限解除映射，则返回-E_BAD_ENV.

六.写时复制（copy-on-write）

Part A:如前所述,Unix 提供系统调用 Fork()其主要的进程创建方式。系统调用 fork()通过对所调用进程(父)的地址空间进行复制的方式来创建新的进程(子)
。
UNIX xv6 的 fork()把父进程的整个数据段(data segment)复制到为子进程新分配的内存区域。这与 dumbfork()的实现是一致的。将父进程的地址复制到子进程是整个 fork()调用过程中代价最大的一部分。
通常情况下调用 fork()之后,子进程就会调 exec()函数以一个新的程序来覆盖当前子进程的空间。大多数情况下,shell 程序就是这么做的。这样,花费在拷贝父进程地址空间的时间被浪费了,因为子进程在调用 exec()之前只使用很少的内存。
基于这个原因,后续版本的 Unix 利用虚拟内存硬件,允许父进程和子进程共享同一块能分别映射到各自地址空间的内存区,直到其中的一个进程需要修改这片内存。这就是写时复制(copy-on-write,也翻译为写前拷贝)。基于这个目的,内核在执行 fork()时,将只拷贝父进程地址空间的映射到子进程,而不是映射页的内容,并且将当前这部分共享页的属性设置为只读的(read-only)。也就是说,两个进程共享了它们的内存区域。当这两个进程中的一个想要修改一个共享的内存页时,程序发生一个缺页错误(page fault)。此时,Unix 内核意识到这个页是一个虚拟的或者说是 copy-on-write 的,所以就会为这个程序创建一个新的私有的可写的内存页。这样,页的内容直到在被修改时才进行复制。这个优化使得 fork()之后在子进程调用 exec()变得更快:子进程在调用 exec()之前也许只需要拷贝一个内存页(堆栈的所在页)。
在接下来的试验中,我们将实现一个合适的类似于 Unix 的 fork()函数,作为用户的库函数,添加 copy-on-write 优化。在用户空间实现 fork()函数和 copy-on-write 支持的好处是使得内核依旧十分简单因而不容易出错。这也支持用户程序定义自己的 fork()。如果程序需要 fork()的不同实现(比如类似于每次都进行拷贝的 dumbfork(),或者父子进程一直共享内存),可以自行实现。

根据提示知道：上面的实验中我们在子进程的时候，需要使用 sys_page_map()函数将父进程的地址空间复制到子进程的地址空间，这一过程会耗费大量的时间，所以为了优化，就采取写时复制的方法。写时复制即是在创建子进程中，并不把父进程所有的页面都映射到子进程中地址空间，而是把只读内存映射到子进程的地址空间。这样在子进程需要写内存的时候，就会出发缺页中断，这时候再为子进程创建新的地址空间。

Part B.用户级缺页处理

1.大致分为两种情况：
（1）一个发 生在栈区域的缺页错误将引发分配和映射新的物理页;一
（1）个发生在 BSS 段区域的缺页错误将 引发分配新的物理页,将之填充为零并把它映射到正确的地址上。

2.设置缺页处理函数
为了让用户自行处理缺页错误,进程需要在 JOS 内核中注册一个缺页处理函数的入口点
(page fault handler entry point)。用户进程可以使用新的 sys_env_set_pgfault_upcall 系统调用来设置这个入口点。在 Env 结构中,已经添加了一个成员 evc_pgfault_upcall 来记录这个入口信息。

作业 7：
该作业需要实现系统调用 sys_env_set_pgfault_upcall()函数。
分析：根据上面的分析可以知道，因为我们在创建子进程的时候并没有给子进程创建足够的页面，而只是将父进程的只读内存映射到了用户进程空间。所以在子进程写内存的时候会发生缺页中断，这时就需要 sys_env_set_pgfault_upcall()函数来进行分页操作。
代码：

```
static int
sys_env_set_pgfault_upcall(envid_t envid, void *func)
{
    // LAB 4: Your code here.
    //首先找到envid对应的env
    struct Env* env;
    if(envid2env(envid,&env,1)<0)
        return -E_BAD_ENV;
    env->env_pgfault_upcall=func;
    return 0;
    //panic("sys_env_set_pgfault_upcall not implemented");
}
```

根据注释提示，首先检查 envid 是否存在，以及该当前调用进程是否由足够的权限。
像 sys_page_map()函数一样，使用 envid2env 来实现。
然后在出现缺页中断时一个分支函数来处理缺页中断，我们只需要将这个分支函数的地址赋值到 ENV 结构的 env_pgfault_upcall 即可。这个缺页处理函数在后面我们会看到，在子进程创建刚开始将暂时使用父进程的缺页处理函数。

作业 8：
要求我们修改 page_fault_handler()函数来处理用户模式下的缺页中断
1.用户进程的普通栈和异常栈
普通栈：在正常的执行中,JOS 中的用户进程运行在普通(normal)的用户堆栈上:它的 ESP 指针初始时指向 USTACKTOP,并且堆栈的数据保存在 USTACKTOP-PGSIZE 与 USTACKTOP-1 之间的 页上。然而,当在用户模式下出现缺页错误时,内核就会在一个被称为用户异常栈(user exception stack)的堆栈上运行指定的用户级缺页处理函数。

异常栈：JOS 的用户异常栈只有一个页的大小,它的头部被定义在虚拟地址的 UXSTACKTOP,所以这个堆栈的可用范围是 UXSTACKTOP-PGSIZE 到 UXSTACKTOP-1。当在这个堆栈上运行时, 用户级的缺页处理函数就可以使用 JOS 的其他系统调用来映射新的页或者调整相关映射来解决导致这个缺页错误的问题。之后用户级的处理程序通过汇编指令返回到在原先堆栈上出错代码的位置

2.调用用户的缺页处理函数
作业 8 要求我们修改 page_fault_handler()函数来处理用户模式下的缺页中断。
分析：在作业 7 中完成了 sys_env_set_pagefault_upcall 函数，该函数在用户进程发生缺页中断的时候由 kernel 产生的一个分支函数。

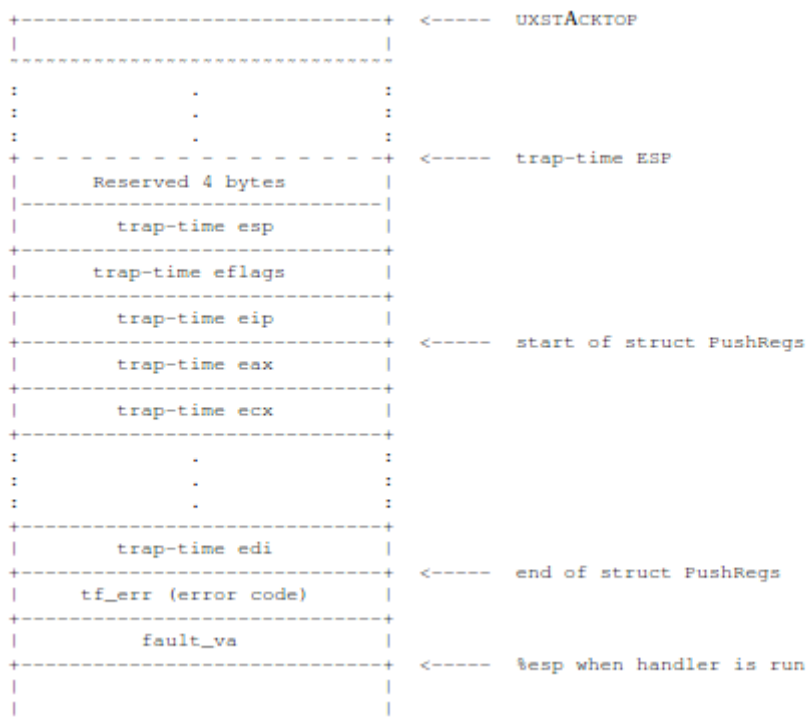
回忆一下中断处理的流程，当进程发生中断时，会在 IDT 中查找相应的处理函数，然后跳到 _alltraps 段将当前寄存器状态压入栈，之后调到 trap()（kern/trap.c）函数中，然后进入 trap_dispatch()函数根据 trapno 调用不同的处理函数。
此时我们的任务便是完成其中的一个中断处理函数 page_fault_handler().

根据注释提示：
（1）当进程的 env_pgfault_upcall 为空的时候（即没有相应的处理函数）销毁该进程
（2）因为栈顶的一个字的空间用来存储处理函数的返回值，所以如果有递归则在每次调用递归函数的时候需要压入一个空字来占位。
（3）然后依次将 tf 的各个变量压入进栈。以便后续的恢复进程。
（4）最后将当前进程的下一条指令改为 env_pgfault_upcall 来调用该处理函数，然后运行该线程。

```
    // LAB 4: Your code here.
    uintptr_t esp_ptr;
    if (curenv->env_pgfault_upcall == NULL)
        goto error;

    if (tf->tf_esp >= UXSTACKTOP-PGSIZE && tf->tf_esp < UXSTACKTOP)
        esp_ptr = tf->tf_esp - sizeof(uint32_t);
    else
        esp_ptr = UXSTACKTOP;

    user_mem_assert(curenv, (const void *)(esp_ptr - sizeof(struct UTrapfram
e)), sizeof(struct UTrapframe), PTE_W);

    esp_ptr -= sizeof(uintptr_t);
    memmove((void *)esp_ptr, &tf->tf_esp, sizeof(uintptr_t));
    esp_ptr -= sizeof(uint32_t);
    memmove((void *)esp_ptr, &tf->tf_eflags, sizeof(uint32_t));
    esp_ptr -= sizeof(uintptr_t);
    memmove((void *)esp_ptr, &tf->tf_eip, sizeof(uintptr_t));
    esp_ptr -= sizeof(struct PushRegs);
    memmove((void *)esp_ptr, &tf->tf_regs, sizeof(struct PushRegs));
    esp_ptr -= sizeof(uint32_t);
    memmove((void *)esp_ptr, &tf->tf_err, sizeof(uint32_t));
    esp_ptr -= sizeof(uint32_t);
    memmove((void *)esp_ptr, &fault_va, sizeof(uint32_t));

    tf->tf_esp = esp_ptr;
    tf->tf_eip = (uintptr_t)curenv->env_pgfault_upcall;

    env_run(curenv);
    // Destroy the environment that caused the fault.
error:
    cprintf("[%08x] user fault va %08x ip %08x\n",
        curenv->env_id, fault_va, tf->tf_eip);
```

练习 5

这里我们需要完成 pfentry.S 汇编函数。根据提示先来看一下从 page_fault_handler()返回后的栈情况。



（1）在 pfentry.S 中刚开始 jos 就将%esp 的值减了四，将函数参数弹出，所以此时%esp 指向 fault_va 的顶部。

（2）根据提示，我们需要恢复出现缺页中断之前的栈的状态，以便在 pfentry.S 执行完成之后继续执行原来的程序。即得到上图中 trap time eip（下一条指令的地址），将其放入 trap time 的栈顶函数返回位置(预留的四个字节)。

这里我们线观察一下 UTrapframe 的结构：

```
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));
```

```
struct PushRegs {
    /* registers as pushed by pusha */
    uint32_t reg_edi;
    uint32_t reg_esi;
    uint32_t reg_ebp;
    uint32_t reg_oesp;      /* Useless */
    uint32_t reg_ebx;
    uint32_t reg_edx;
    uint32_t reg_ecx;
    uint32_t reg_eax;
} __attribute__((packed));
```

可以从上面的数据结构得到 trap-time eip 在栈中的位置。此时%esp 偏移 48 个字节的地方就是 trap time esp 在栈中的位置。因为在递归调用缺页处理函数的时候，我们预留了 4 个字节空间用于接收返回值。所以我们需要将该 esp 减 4 则得到 trap time 的栈顶即 trap-time 的 ESP。

```
movl 0x30(%esp),%eax
subl $0x4,%eax
movl %eax,0x30(%esp)
```

此时 trap time 的 esp 指向栈顶，%eax 项 time trap 的 ESP-4 的地方，就是预留 4 个字节的栈顶。

（3）trap-time 的 eip 值为下一条指令的地址,就是我们想要的。根据上面的栈结构可以知道 trap time 的 esp 是%esp 的 0x28 偏移处的位置，即（2）中的%eax。我们需要做的就是将 trap-time 的 eip 放入预留的 4 个字节的地方，即函数的返回值，这样在处理函数结束返回后，原程序的栈顶就是下一条指令的地址。

```
movl 0x28(%esp),%ebx
movl %ebx,(%eax)
```

（4）根据提示此时我们需要恢复源程序的通用寄存器

```
addl $0x8,%esp
popal
// Restore eflags from the stack.  After you do this, you can
// no longer use arithmetic operations or anything else that
// modifies eflags.
// LAB 4: Your code here.
addl $0x4,%esp
popfl
// Switch back to the adjusted trap-time stack.
// LAB 4: Your code here.
pop %esp
```

将%esp 加 8,使%esp 指向 PushRegs 的顶部，即跳过 fault_va 和 tf_err。
然后使用 popal 用栈中的 PushRegs 结构中的变量来恢复通用寄存器。
然后跳过 trap time 的 eip（此时它已经没有用了），将 utf_eflags 弹出

作业 9：
完成 set_pgfault_handler()（lib/pgfault.c）
分析：这个函数用来完成每个进程 page_fault 处理程序在 kernel 中的注册。
该函数的主要目的是为用户进程申请用户错误栈空间。
其中_pagefault_upcall 为当前 c 语言缺页处理函数。
代码：

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

    if (_pgfault_handler == 0) {
        // First time through!
        //首先使用sys_page_allo()来分配空间
        // LAB 4: Your code here.
        if((r=sys_page_alloc(0,(void*)(UXSTACKTOP-PGSIZE),PTE_U|PTE_P|PTE_W
)<0)
            panic("set_pgfault_handler failed:%e",r);
        //其中_pagefault_upcall为当前c的缺页处理函数
        sys_env_set_pgfault_upcall(0,_pgfault_upcall);
        //panic("set_pgfault_handler not implemented");
    }

    // Save handler pointer for assembly to call.
    _pgfault_handler = handler;
```

如果_pgfault_handler 不等于 0,说明该进程已经在内核有了缺页中断处理函数，不需要做处理。如果为 0 则调用 sys_env_set_pgfault_upcall()函数为其定义缺页处理函数。

作业 10：

完成 lib/fork.c 中的函数：pgfault()，duppage()和 fork()

分析：

pgfault():

该函数是用户自定义缺页处理函数。前面说过父进程通过 sys_env_set_pgfault_upcall()来完成向 kernel 注册缺页处理函数，然后父进程通过 sys_exofork()创建一个子进程。对于父进中每一个标记为可写的或者写时复制的虚拟地址位于 UTOP 以下的页, 父进程将子进程地址空间中的对应页映射为写时复制属性,并且重新将自身地址空 间中的该页映射为写时复制属性。父进程将两个进程的 PTE 都设置为不可写,并且 在”avail”区域中设置了 PET_COW(写时复制)属性,用以区别写时复制页和只读 页。

父进程为子进程设置与其相同的用户缺页处理函数入口

当用户模式下发生缺页中断的时候，进程会调用该处理函数完成相关的处理。

函数 pgfault()检查这个错误是写操作(FEC_WR)并且缺页的 PTE 是含有 PTE_COW 属性的。如果不是,调用 panic 给出警告。

```
// LAB 4: Your code here.
if((err&FEC_WR)==0)
    panic("not write fault");
if((vpd[PDX(addr)]&PTE_P)==0)
    panic("page directory entry is not present!");
if((vpt[PGNUM(addr)]&PTE_COW)==0)
    panic("not copy on write");
```

根据注释提示，我们首先检查该错误是否为写错误，如果不是则 panic()

然后检查页表入口是否为 PTE_COW(再次之前需要检查在页目录中是否有该页表项)

接着根据注释，需要为当前进程分配一个页，然后将原来页面的数据拷贝到新的页面上，最后将其映射到当前进程的地址空间。

```
// LAB 4: Your code here.
//为当前进程分配一个页
if((r=sys_page_alloc(0,(void*)PFTEMP,PTE_U|PTE_P|PTE_W))<0)
    panic("page alloc failed");
//copy data from old page to new page
addr=ROUNDDOWN(addr,PGSIZE);
memmove(PFTEMP,addr,PGSIZE);
//move the new page to the old page's address
if((r=sys_page_map(0,PFTEMP,0,addr,PTE_P|PTE_U|PTE_W))<0)
    panic("sys_page_map failed");
//panic("pgfault not implemented");
```

函数 pgfault()在临时空间分配一个页,然后将整个缺页的内容拷贝到这个页上。然后,缺页处理函数将这个页映射到适当的地方,并赋予读/写权限。

duppage():

将指定的虚拟页映射到指定进程地址空间的相同地址上。

首先检查该页是否为可写或者写时复制，如果是则将其映射到 envid 地址空间的相同地址上，同时将此时映射到当前进程地址空间，并且赋予写时复制的属性。

如果该页面是只读页面，则将其映射到当前地址空间相同地址上。

```
//将虚拟页映射到指定进程的相同虚拟地址上
static int
duppage(envid_t envid, unsigned pn)
{
    int r;

    // LAB 4: Your code here.
    void* addr=(void*)((uint32_t)pn*PGSIZE);
    pte_t pte=vpt[PGNUM(addr)];
    if((pte&PTE_W)||(pte&PTE_COW))
      {
        if((r=sys_page_map(0,addr,envid,addr,PTE_U|PTE_P|PTE_COW))<0)
          panic("duppage:page map failed");
        if((r=sys_page_map(0,addr,0,addr,PTE_U|PTE_P|PTE_COW))<0)
          panic("duppage: page re-map failed");

      }else{
        sys_page_map(0,addr,envid,addr,PTE_U|PTE_P);
        }
    //panic("duppage not implemented");
    return 0;
}
```

fork():

该函数用来创建子进程，而且是写时复制。

分析：首先我们来看一下创建子进程的流程：

(1)父进程设置用户模式下缺页处理函数

(2)使用系统调用 sys_exofork()完成子进程的创建，该函数会创建一个和父进程寄存器状态一样的进程,而且此时子进程的状态为 ENV_NOT_RUNNABLE

(3)然后将用户空间(UTOP 以下)映射到子进程的地址空间中。

(4)父进程为子进程设置与其相同的用户缺页处理函数

(5)父进程运行子进程

```
    // LAB 4: Your code here.
    //首先为当前进程设置缺页处理函数
    set_pgfault_handler(pgfault);

    envid_t envid;
    uint32_t addr;
    int r;
    //调用sys_exofork()创建子进程
    envid=sys_exofork();
    if(envid<0)
      panic("sys_exofork failed");
    //return 0 when env is child
    if(envid==0){
      env=&envs[ENVX(sys_getenvid())];
      return 0;
    }

    //父进程将用户空间的页面映射到子进程
    for (addr = 0; addr < USTACKTOP; addr += PGSIZE)
        if ((vpd[PDX(addr)] & PTE_P) && (vpt[PGNUM(addr)] & PTE_P)
            && (vpt[PGNUM(addr)] & PTE_U)) {
            duppage(envid, PGNUM(addr));
        }
    //为子进程设置错误栈
    if (sys_page_alloc(envid, (void *)(UXSTACKTOP-PGSIZE), PTE_U|PTE_W|PTE_P
) < 0)
          panic("sys_page_alloc() failed");

    //设置env_pgfault_upcall
    extern void _pgfault_upcall();
        sys_env_set_pgfault_upcall(envid, _pgfault_upcall);

        if (sys_env_set_status(envid, ENV_RUNNABLE) < 0)
          panic("sys_env_set_status");

    return envid;
```

七.抢占式多任务和进程间通信

1.时钟中断和抢占

此时的 jos 内核还支持外部的时钟中断，因此认可一个无限循环的程序都会导致进程一直占用 CPU 不放弃，以致于系统失去响应。因此为了保证内核可以从一个正在运行的进程中夺取 cpu 控制权，就必须修改 jos 内核支持持外部的时钟中断

2.中断原理

(1)外部中断(比如硬件中断)被称作 **IRQ（中断请求）**。一共有 16 种 IRQ,编号从 0 到 15。从 IRQ 号

(2)到 IDT 的映射关系不是固定的。文件 picirq.c 中的 pic_init()函数将 IRQ 0-15 映射到 IDT 的

(3)IRQ_OFFSET 到 IRQ_OFFSET+15。

在文件 kern/picirq.h 中,IRQ_OFFSET 被定义为 10 进制的 32。这样,IDT 的 32 到 47 就对应着 IRQ 的 0 到 15。

(4)在内核时外部设备中断是关闭的,在 用户模式下时才响应。外部设备中断被%eflags 寄存器的 FL_IF 标志位所控制(inc/mmu.h)。当 这位为 1 时,外部中断是开启的。这一位可以通过多种方式进行修改,但根据我们的简化实 现,我们将只在进出用户模式时通过保存和回写%eflags 寄存器进行修改。

(5)在进程中我们需要保证 FL_IF 被置位,这样在这个进程运行时出现中断,就可以到达处理器并被相应的中断处理代码所处理。否则,中断就会被屏蔽或者忽略直到外部中断被开启。在处理器重启动之后,中断是默认屏蔽的,并且到目前为止我们还没有开启它。

作业 11：

修改 kern/trapentry.S 和 kern/trap.c，在 IDT 中提供外部中断的入口。

分析：根据作业 11 的提示，在发生硬件中断的时候进程并不会压入 error code。所以使用 TRAPHANDLER_NOEC()

```
//IRQ handler
//IRQ_OFFSET=32
TRAPHANDLER_NOEC(trap_irq0,32)
TRAPHANDLER_NOEC(trap_irq1,33)
TRAPHANDLER_NOEC(trap_irq2,34)
TRAPHANDLER_NOEC(trap_irq3,35)
TRAPHANDLER_NOEC(trap_irq4,36)
TRAPHANDLER_NOEC(trap_irq5,37)
TRAPHANDLER_NOEC(trap_irq6,38)
TRAPHANDLER_NOEC(trap_irq7,39)
TRAPHANDLER_NOEC(trap_irq8,40)
TRAPHANDLER_NOEC(trap_irq9,41)
TRAPHANDLER_NOEC(trap_irq10,42)
TRAPHANDLER_NOEC(trap_irq11,43)
TRAPHANDLER_NOEC(trap_irq12,44)
TRAPHANDLER_NOEC(trap_irq13,45)
TRAPHANDLER_NOEC(trap_irq14,46)
TRAPHANDLER_NOEC(trap_irq15,47)
```

```
extern void trap_irq0();
extern void trap_irq1();
extern void trap_irq2();
extern void trap_irq3();
extern void trap_irq4();
extern void trap_irq5();
extern void trap_irq6();
extern void trap_irq7();
extern void trap_irq8();
extern void trap_irq9();
extern void trap_irq10();
extern void trap_irq11();
extern void trap_irq12();
extern void trap_irq13();
extern void trap_irq14();
extern void trap_irq15();
```

```
SETGATE(idt[32],0,GD_KT,trap_irq0,0);
SETGATE(idt[33],0,GD_KT,trap_irq1,0);
SETGATE(idt[34],0,GD_KT,trap_irq2,0);
SETGATE(idt[35],0,GD_KT,trap_irq3,0);
SETGATE(idt[36],0,GD_KT,trap_irq4,0);
SETGATE(idt[37],0,GD_KT,trap_irq5,0);
SETGATE(idt[38],0,GD_KT,trap_irq6,0);
SETGATE(idt[39],0,GD_KT,trap_irq7,0);
SETGATE(idt[40],0,GD_KT,trap_irq8,0);
SETGATE(idt[41],0,GD_KT,trap_irq9,0);
SETGATE(idt[42],0,GD_KT,trap_irq10,0);
SETGATE(idt[43],0,GD_KT,trap_irq11,0);
SETGATE(idt[44],0,GD_KT,trap_irq12,0);
SETGATE(idt[45],0,GD_KT,trap_irq13,0);
SETGATE(idt[46],0,GD_KT,trap_irq14,0);
SETGATE(idt[47],0,GD_KT,trap_irq15,0);
```

3.处理时钟中断

在 kern/picirq.c 中的函数 pic_init()和 kern/kclock_init()用来设置时钟和中断控制器来生成中断。

作业 12：

修改 trap_dispatch()（kern/trap.c）,使得在每次时钟中断发生的时候就调用 sched_yield().

我们知道在 trap_dispatch()中使用 trapno 来判定是哪种中断，在 inc/trap.h 中定义了各种中断号。其中时钟中断为 IRQ_OFFSET+IRQ_TIMER

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

4.进程间通信
在 jos 中，实现了两个系统调用 sys_ipc_recv 和 sys_ipc_try_send，和两个库函数(用户进程调用的函数)ipc_recv（其会调用 sys_ipc_recv）和 ipc_send(其会调用 sys_ipc_try_send).

（1）进程间通信的流程：当一个进程需要接收消息时，它就会调用 ipc_recv 等待接收消息，在调用 ipc_recv 之后该进程就会被挂起并在接收到消息之前停止运行。而当一个进程需要发送数据时，它会调用 ipc_send,该函数指定目标进程的 ID 和将要发送的消息，如果此时目标位于 ipc_recv 状态就会接收数据，否则 ipc_send 就会不断的调用 sys_ipc_try_send 发送消息。

（2）页的传递
在（1）中我们介绍消息的发送进程，所发送的消息包括两个部分，一个 32 位的值以及可选的一个页 的映射关系，这样我们就可以实现更多数据的传输。下面介绍一下页传递的流程：
首先我们先查看一下 sys_ipc_recv 和 sys_ipc_try_send 的函数原型：

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

```
static int
sys_ipc_recv(void *dstva)
```

当一个接收进程调用 sys_ipc_recv 并传递给其一个非 0 的 dstva 参数时，就表明该进程需要得到一个页的映射;同理当发送进程调用 sys_ipc_try_send 时并传递给其一个非 0 的 srcva 时表明该进程要发送一个页的映射。
如果接收进程和发送同时表明需要传递页的映射，则完成页的传递实现共享页;否则传递失败

作业 13：
完成 sys_ipc_recv()，sys_ipc_try_send()，ipc_recv 和 ipc_send

sys_ipc_try_send():
分析：根据注释我们可以知道每一种情况对应的处理方法
首先我们先看一下 Env 结构相对于 lab3 的变化

```
// 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
```

可以看到 Env 结构新加了记录 ipc 状态的各个变量，详细作用参加注释
因为每个进程都拥有相同的 kernel 空间，所以 kernel 地址空间不需要考虑
（1）如果目标进程不存在，则返回-E_BAD_ENV
（2）如果目标进程不处于消息接收状态，则返回-E_IPC_NOT_RECV
（3）如果 srcva 在用户空间但没有对齐则返回-E_INVAL
（4）如果 srcva 在用户空间而且 perm 不正确，即 sys_ipc_try_send 的参数 perm 与 srcva 所在页的 perm 不匹配，则返回-E_INVAL
（5）如果 srcva 在用户空间，但在不再发送进程的地址空间则返回-E_INVAL
（6）sys_ipc_try_send 的 perm 参数包含 PTE_W,但是当前进程空间的该页为只读，则返回-E_INVAL
（7）如果目标进程没有足够的空间来映射传递的页，则返回-E_NO_MEM
当检查完成上述 7 条后，就开始发送消息了，将目标进程的 ipc 各个变量设为要发送的消息值。

并将接收进程结束阻塞态。

```
struct Env* dstenv;
int r;
if(envid2env(envid,&dstenv,0)<0)
    return -E_BAD_ENV;
//envid is not currently blocked in sys_ipc_recv
//or another environment managed to send first.
if(dstenv->env_ipc_recving||dstenv->env_ipc_form)
    return -E_IPC_NOT_RECV;
//if srcva < UTOP but srcva is not page-aligned.
if(srcva<(void*)UTOP&&ROUNDDOWN(srcva,PGSIZE)!=srcva)
    return -E_INVA;
//if srcva < UTOP and...
if(srcva<(void*)UTOP){
  pte_t* pte;
  struct Page*pg=page_lookup(curenv->env_pgdir, srcva, &pte);
//srcva is not mapped in the caller's address space.
  if(!pg) return -E_INVAL;
//perm is inappropriate
  if((perm&*pte)!=perm) return -E_INVAL;
//if (perm & PTE_W), but srcva is read-only in the
//current environment's address space.
  if((perm|PTE_W)&&(*pte|PTE_W)==0)
    return -E_INVAL;
//-E_NO_MEM if there's not enough memory to map srcva in envid's
//address space
  if(env->env_ipc_dstva < (void*)UTOP){
    r = page_insert(e->env_pgdir, pg, e->env_ipc_dstva, perm
        if (r) return r;
        env->env_ipc_perm = perm;
  }
}

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

sys_ipc_recv():

分析：进程通过调用该函数，进入接收消息的状态

根据注释提示，如果 dstva 在用户空间但是没有 4K 对齐就返回-E_INVAL

然后将 env_ipc_recving 置为 1 表示正在接收消息，此时接收进程进入阻塞态，通过调用 sched_yield()函数进行轮转调度。

```
static int
sys_ipc_recv(void *dstva)
{
    // LAB 4: Your code here.
    if(dstva<(void*)UTOP){
      if(dstva!=ROUNDDOWN(dstva,PGSIZE));
        return -E_INVAL;
    }
    curenv->env_ipc_recving = 1;
        curenv->env_status = ENV_NOT_RUNNABLE;
        curenv->env_ipc_dstva = dstva;
        sys_yield();
        return 0;
    //panic("sys_ipc_recv not implemented");
    return 0;
}
```

ipc_recv():

当一个用户进程调用 ipc_recv 时，该函数会调用 sys_ipc_recv 然后该函数进入阻塞态，等待发送消息的进程将其激活，然后将参数*form_env_store 置为发送进程的 ID，perm_store 置为映射页的权限。最后返回 env_ipc_value

```
// LAB 4: Your code here.
int r;
void *dstva;
if(pg)
  dstva=pg;
else
  dstva=(void*)UTOP;
int r=sys_ipc_recv(dstva);
if(r<0)return r;
if(from_env_store)
  *from_env_store=thisenv->env_ipc_form;
if(perm_store)
  *perm_store=thisenv->env_ipc_perm;
//panic("ipc_recv not implemented");
return thisenv->env_ipc_value;
```

ipc_send():

进程调用该函数进行发送消息，该函数会调用 sys_ipc_try_send,如果没有发送成功则继续发送直至发送成功位置，但是如果是因为目标进程不在 recving 而导致的失败，则跳出循环，通信失败。

```
// LAB 4: Your code here.
if (!pg) pg = (void*)-1;
int ret;
    while ((ret = sys_ipc_try_send(to_env, val, pg, perm))) {
    if (ret == 0) break;
    if (ret != -E_IPC_NOT_RECV) panic("not E_IPC_NOT_RECV, %e", ret);
    sys_yield();
    }
panic("ipc_send not implemented");
```

至此实验全部完成。

make grade 的结果如下：

```
dumbfork: OK (1.8s)
Part A score: 5/5

faultread: OK (1.0s)
faultwrite: OK (1.0s)
faultdie: OK (0.9s)
faultregs: OK (1.1s)
faultalloc: OK (1.0s)
faultallocbad: OK (1.8s)
faultnostack: OK (2.2s)
faultbadhandler: OK (2.0s)
faultevilhandler: OK (2.0s)
forktree: OK (2.2s)
Part B score: 50/50

spin: OK (2.1s)
stresssched: OK (2.7s)
pingpong: OK (2.1s)
primes: OK (19.6s)
Part C score: 20/20

Score: 75/75
```