title: Jos-Lab4
categories: Jos-Lab

JOS-Lab 4: Preemptive Multitasking

Introduction

LAB4总体来讲需要我们实现多处理器架构，为用户进程实现fork操作，实现进程间通信，并最终实现抢占式多任务处理机制。本文主要讲述lab4中exercise代码的思路，question的回答请见`answer-lab4.txt`，challenge的实现请见本文档最后。

Part A

Part A需要我们拓展JOS使其能在多核系统上运行，实现一些内核态系统调用以便用户态environment能够生成更多environment，同时还要实现合作式循环调度，使得当当前environment自愿放弃CPU(或运行结束)时kernel能够从这environment切换到另一个environment。

Exercise 1

Exercise 1需要修改`kern/pmap.c`中的`page_init()`函数的实现，使其避免将`MPENTRY_PADDR`所在的页加入到free list中，从而让系统能够安全地将AP(Application Processor)的bootstrap代码拷贝到此地址并运行。只需要在`page_init()`函数的判断条件中多加一条即可：

```
if(i==0||(hole_begin<=i&&i<=hole_end)||i == PGNUM(MPENTRY_PADDR))
```

Exercise 2

Exercise2需要修改`kern/pmap.c`中的`mem_init_mp()`函数，映射每个cpu的stack。根据`inc/memlayout.h`中的示意，能发现cpu0的stack存在于地址空间`[KSTACKTOP-KSTKSIZE,KSTACKTOP]`,cpu1的stack与cpu0的stack相隔`KSTKGAP`，存在于`[KSTACKTOP-(KSTKSIZE+KSTKGAP)-KSTKSIZE,KSTACKTOP-(KSTKSIZE+KSTKGAP)]`,其余cpu以此类推。

```
static void
mem_init_mp(void)
{
    boot_map_region(kern_pgdir, IOMEMBASE, -IOMEMBASE, IOMEM_PADDR, PTE_W);

    // LAB 4: Your code here:
    int i;
    for (i = 0 ;i < NCPU ;i ++)
    {
         boot_map_region(kern_pgdir, KSTACKTOP - i * (KSTKSIZE + KSTKGAP) - KSTKSIZE, KSTKSIZE,PADDR(percpu_kstacks[i]), PTE_W | P
    }
}
```

Exercise 3

Exercise3要求我们修改`kern/trap.c`中的`trap_init_percpu()`函数，使其能够为每个cpu初始化TSS和TSS descriptor，这里，全局变量ts不能够再使用。第i个CPU的TSS是`cpus[i].cpu_ts`，TSS descripter是`gdt[(GD_TSS0 >> 3) + i]`。

```
trap_init_percpu(void)
{
    struct Taskstate * cputs = &thiscpu->cpu_ts;
    int cpuid = thiscpu->cpu_id;

    //init for sysenter
    extern void sysenter_handler();
    __asm__ __volatile__("wrmsr"
        : : "c" (0x174), "a" (GD_KT), "d" (0));
    __asm__ __volatile__("wrmsr"
        : : "c" (0x175), "a" (KSTACKTOP - cpuid * (KSTKSIZE + KSTKGAP)), "d" (0));
```

```
    __asm__ __volatile__("wrmsr"
        : : "c" (0x176), "a" ((uint32_t)(char *) sysenter_handler), "d" (0));

    // Setup a TSS so that we get the right stack
    // when we trap to the kernel.
    cputs->ts_esp0 = KSTACKTOP - cpuid * (KSTKSIZE + KSTKGAP);
    cputs->ts_ss0 = GD_KD;

    // Initialize the TSS slot of the gdt.
    gdt[(GD_TSS0 >> 3) + cpuid] = SEG16(STS_T32A, (uint32_t) (cputs),sizeof(struct Taskstate), 0);
    gdt[(GD_TSS0 >> 3) + cpuid].sd_s = 0;

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

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

Exercise 4

Exercise4要求我们利用系统提供的`lock_kernel()`以及`unlock_kernel()`函数为kernel代码的执行进行加锁，保证不能有超过一个environment同时在执行kernel态代码，按题目提示，在`i386_init()`代码中BSP唤醒其它CPU后对内核加锁，在`mp_main()`代码中当AP初始化后对内核加锁并调用`sched_yield()`函数，在`trap()`函数中执行加锁操作，在`env_run()`函数中转移到用户态前执行锁释放操作。另外，因为lab3用sysenter来进入syscall，所以这里也要加锁。

Exercise 4.1

Exercise4.1让我们在`kern/spinlock.c`中实现spinlock，spinlock的语义是在lock中存在owner_ticket和next_ticket，当一个程序获取锁时，读取next_ticket作为自己的ticket，并将next_ticket加一(读和加一为一整个原子操作)，当owner_ticket与自己的ticket相同时表示自己现在拥有锁，否则必须等待直到owner_ticket与自己的ticket相等，释放锁时只需将owner_ticket加一以将锁传递给下一ticket的拥有者。实现如下：
首先在holding中通过检查own和next是否相等来判断当前是否拿锁

```
return lock->own != lock->next && lock->cpu == thiscpu;
```

然后填补spinlock的初始化函数：

```
void
__spin_initlock(struct spinlock *lk, char *name)
{
#ifndef USE_TICKET_SPIN_LOCK
    lk->locked = 0;
#else
    //LAB 4: Your code here
    lk->own=0;
    lk->next=0;
#endif

#ifdef DEBUG_SPINLOCK
    lk->name = name;
    lk->cpu = 0;
#endif
}
```

加锁的实现比较简单，只需要获取next_ticket作为自己的ticket并等待own_ticket增长到与自己的ticket相等即可。

```
void
spin_lock(struct spinlock *lk)
{
#ifdef DEBUG_SPINLOCK
    if (holding(lk))
        panic("CPU %d cannot acquire %s: already holding", cpunum(), lk->name);
#endif
```

```
#ifndef USE_TICKET_SPIN_LOCK
    while (xchg(&lk->locked, 1) != 0)
        asm volatile ("pause");
    //cprintf("CPU %d: lock kernel\n", cpunum());
#else
    //LAB 4: Your code here
    int own = atomic_return_and_add(&lk->next, 1);
    while(atomic_return_and_add(&lk->own, 0)!= own);
    asm volatile("pause");

#endif

#ifdef DEBUG_SPINLOCK
    lk->cpu = thiscpu;
    get_caller_pcs(lk->pcs);
#endif
}
```

释放锁时只需要将own_ticket加一

```
atomic_return_and_add(&lk->next, 1);
```

Exercise 5

Exercise5要求我们实现循环调度算法，主要就是实现`kern/sched.c`中的`sched_yield()`函数，方法就是遍历当前所有envid，当找到一个状态为ENV_RUNNABLE并且不属于ENV_TYPE_IDLE的进程后就运行它，实在找不到就运行ENV_TYPE_IDLE进程。

```
void
sched_yield(void)
{
    struct Env *idle;
    int i;
    // LAB 4: Your code here.
    uint32_t env_id;
    if(curenv != NULL)
    {
        env_id = ENVX(curenv->env_id);
        i = (env_id+1)%NENV;
        while(i != env_id)
        {
            if(envs[i].env_status == ENV_RUNNABLE && envs[i].env_type != ENV_TYPE_IDLE)
            {
                env_run(&envs[i]);
            }
            i = (i+1)%NENV;
        }
        if(curenv->env_status == ENV_RUNNING)
        {
            env_run(curenv);
        }
    }

    // For debugging and testing purposes, if there are no
    // runnable environments other than the idle environments,
    // drop into the kernel monitor.
    for (i = 0; i < NENV; i++) {
        if (envs[i].env_type != ENV_TYPE_IDLE &&
            (envs[i].env_status == ENV_RUNNABLE ||
             envs[i].env_status == ENV_RUNNING))
            break;
    }
    if (i == NENV) {
        cprintf("No more runnable environments!\n");
        while (1)
            monitor(NULL);
```

```
    }

    // Run this CPU's idle environment when nothing else is runnable.
    idle = &envs[cpunum()];
    if (!(idle->env_status == ENV_RUNNABLE || idle->env_status == ENV_RUNNING))
        panic("CPU %d: No idle environment!", cpunum());
    env_run(idle);
}
```

完成函数后在syscall()中注册一下并在init.c中创建一些env来检验调度算法的正确性。

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

    ENV_CREATE(user_yield, ENV_TYPE_USER);
    ENV_CREATE(user_yield, ENV_TYPE_USER);
    ENV_CREATE(user_yield, ENV_TYPE_USER);
#endif // TEST*
```

Exercise 6

在此之前，JOS执行的env都是kernel初始化时创建的，Exercise6要求我们简单地实现一些system call来生成新的用户环境，从而实现`fork()`函数。首先实现`exofork()`，它为当前进程alloc一个子进程，对于父进程返回子进程的id，对于子进程返回0。因为子进程的页表等尚未初始化，故先把子进程设为`ENV_NOT_RUNNABLE`。

```
static envid_t
sys_exofork(void)
{
    // LAB 4: Your code here.
    struct Env *child;
    int i;
    i = env_alloc(&child, curenv->env_id);
    if (i < 0)
    {
        return i;
    }
    child->env_status = ENV_NOT_RUNNABLE;
    child->env_tf = curenv->env_tf;
    child->env_tf.tf_regs.reg_eax = 0;
    return child->env_id;
    //panic("sys_exofork not implemented");
}
```

然后是`sys_env_set_status`，它改变一个进程的状态，可以用来让新alloc出来的子进程变为`ENV_RUNNABLE`，从而让子进程变为可运行并被调度运行。状态改变前还要判断一下参数id是否有效以及是否有权限更改进程状态。

```
static int
sys_env_set_status(envid_t envid, int status)
{
    // LAB 4: Your code here.
    struct Env *e;
    int i;
    i = envid2env(envid, &e, 1);
    if (i < 0)
        {
        return i;
    }
    if (status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE)
    {
            return -E_INVAL;
    }
```

```
    e->env_status = status;
    return 0;
    //panic("sys_env_set_status not implemented");
}
```

然后是`sys_page_alloc()`函数，它为envid对应的进程alloc一个权限为perm的page，在alloc之前要判断envid、va的有效性，另外，为保证原子性操作，当page_insert操作不成功时需要将已alloc的page再释放掉。

```
static int
sys_page_alloc(envid_t envid, void *va, int perm)
{
    // LAB 4: Your code here.
    int i;
    struct Env *e;
    struct Page *page1;
    if(va>=(void*)UTOP||PGOFF(va)!=0||(perm&(~PTE_SYSCALL))!=0)
    {
        return -E_INVAL;
    }
    i=envid2env(envid,&e,1);
    if(i<0)
    {
        return -E_BAD_ENV;
    }
    page1=page_alloc(ALLOC_ZERO);
    if(!page1)
    {
        return -E_NO_MEM;
    }
    i=page_insert(e->env_pgdir,page1,va,perm);
    if(i<0)
    {
        page_free(page1);
        return -E_NO_MEM;
    }
    memset(page2kva(page1),0,PGSIZE);
    return 0;
    //panic("sys_page_alloc not implemented");
}
```

然后是`sys_page_map()`函数，它将srcenvid对应的进程的地址空间中的srcva地址所在的page复制到dstenvid对应的进程的地址空间dstva，并给予其perm权限。逻辑比较简单，但这里要进行的判断比较多，包括两个envid的有效性、两个va的有效性以及最终操作是否成功。

```
static int
sys_page_map(envid_t srcenvid, void *srcva,
         envid_t dstenvid, void *dstva, int perm)
{
    // LAB 4: Your code here.
    //cprintf("page_map:perm:%d\n",perm);
    //cprintf("page_map:srcenvid:%d\n",srcenvid);
    struct Env *srcenv, *dstenv;
    struct Page *page1;
    pte_t *pte1;
    int i;
    if (((uint32_t)srcva) >= UTOP || ((uint32_t)srcva) % PGSIZE != 0)
    {
        return -E_INVAL;
    }
    if (((uint32_t)dstva) >= UTOP || ((uint32_t)dstva) % PGSIZE != 0)
    {
        return -E_INVAL;
    }
    if ((i = envid2env(srcenvid, &srcenv, 1)) < 0)
    {
        return -E_BAD_ENV;
```

```
    }
    if ((i = envid2env(dstenvid, &dstenv, 1)) < 0)
    {
        return -E_BAD_ENV;
    }

    if(!(page1 = page_lookup(srcenv->env_pgdir, srcva, &pte1)))
    {
        return -E_INVAL;
    }
    if (!(perm & PTE_P))
    {
        //cprintf("problem here1\n");
        return -E_INVAL;
    }
    if ((!((*pte1) & PTE_W)) && (perm & PTE_W))
    {
        //cprintf("problem here2\n");
        return -E_INVAL;
    }

    if ((i = page_insert(dstenv->env_pgdir, page1, dstva, perm)) < 0)
    {
        return -E_NO_MEM;
    }
    return 0;
    //panic("sys_page_map not implemented");
}
```

最后是`sys_page_unmap()`函数，它将va对应的page从envid对应的进程的地址空间中删除。

```
static int
sys_page_unmap(envid_t envid, void *va)
{
    // LAB 4: Your code here.
    struct Env *e;
    int i;

    i = envid2env(envid, &e, 1);
    if (i < 0)
    {
        return -E_BAD_ENV;
    }

    if (((uint32_t)va) >= UTOP || ((uint32_t)va) % PGSIZE != 0)
    {
        return -E_INVAL;
    }

    page_remove(e->env_pgdir, va);
    return 0;
    //panic("sys_page_unmap not implemented");
}
```

完成这些函数后将对应的syscall添加到syscall()函数即可。

Part B

Part B主要就是让我们实现写时拷贝技术，当父进程fork出一个子进程时，它们指向的地址空间仍然是共享的，只有当其中一个进程需要更改某块本来是应该单独享有的地址空间时，为了不影响另一进程，它要将这块空间上的内容拷贝到一块新的空间，将自己的映射改在这块新空间，并在新空间上进行操作。这样可以尽量避免不必要的拷贝，从而提高效率。

Exercise 7

Exercise7让我们实现`sys_env_set_pgfault_upcall`这么一个syste call，该system call是实现的功能是将某个函数与page fault相绑定，让系统能够在

出现page fault后调用该函数进行处理。实现方式就是将envid对应的env的env_pgfault_upcall置为参数中的func。

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

Exercise 8

Exercise8要我们实现上述的处理函数，并更改trap_dispatch，让系统为page fault分配处理函数。page_fault_handler函数的实现方式就是先检查处理函数的地址空间是否存在，如果不存在就应将引发错误的env摧毁掉，否则再判断env运行在用户栈还是异常栈，如果是用户栈就将当前状态压入异常栈，是异常栈就隔一段空位再压栈。

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

    fault_va = rcr2();

    if (!(tf->tf_cs & 0x3))
        panic("PGFLT trapped in kernel\n");

    // LAB 4: Your code here.

    // Destroy environment
    if(curenv->env_pgfault_upcall==NULL)
    {
        cprintf("[%08x] user fault va %08x ip %08x\n",curenv->env_id, fault_va, tf->tf_eip);
        print_trapframe(tf);
        env_destroy(curenv);
    }
    struct UTrapframe *utf;
    uint32_t trap_esp = tf->tf_esp;
    uint32_t utsize = sizeof(struct UTrapframe);

    if ((trap_esp>=UXSTACKTOP-PGSIZE) && (trap_esp<UXSTACKTOP))
    {
        utf = (struct UTrapframe*)(trap_esp-utsize-4);
    }
    else
    {
        utf = (struct UTrapframe*)(UXSTACKTOP-utsize);
    }

    user_mem_assert(curenv, (void*)utf,utsize, PTE_U | PTE_W);
    utf->utf_fault_va = fault_va;
    utf->utf_esp = tf->tf_esp;
    utf->utf_eip = tf->tf_eip;
    utf->utf_eflags = tf->tf_eflags;
    utf->utf_regs = tf->tf_regs;
    utf->utf_err = tf->tf_err;

    curenv->env_tf.tf_eip = (uint32_t)curenv->env_pgfault_upcall;
    curenv->env_tf.tf_esp = (uint32_t)utf;
```

```
    env_run(curenv);
}
```

Exercise 9

Exercise9要求我们实现`lib/pfentry.S`中的`_pgfault_upcall`，返回引发page fault的用户代码的原始点，不需要经过内核代码，同时还要切换堆栈以及eip。

```
    // LAB 4: Your code here.
    movl 0x30(%esp), %eax
    movl 0x28(%esp), %ebx
    subl $0x4, %eax
    movl %ebx, (%eax)
    movl %eax, 0x30(%esp)

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

    // Return to re-execute the instruction that faulted.
    // LAB 4: Your code here.
    ret
```

Exercise 10

Exercise10要求我们完成`lib/pgfault.c`中的`set_pgfault_handler()`函数，为用户提供自定义的page fault处理。实现方式比较简单，先检查handler函数是否已被设置过，如果没有就先为handler函数分配一块空间，然后将handler函数设置成自己想要的处理函数。

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

    if (_pgfault_handler == 0) {
        if((r = sys_page_alloc(0,(void*)(UXSTACKTOP-PGSIZE),PTE_U|PTE_P|PTE_W)) < 0)
        {
            panic("set_pgfault_handler: %e\n", r);
        }
        sys_env_set_pgfault_upcall(0, _pgfault_upcall);
    }

    _pgfault_handler = handler;
}
```

Exercise 11

Exercise11首先要求我们真正意义上实现一个写时拷贝的fork函数，fork函数所做的事情如下：首先用之前实现的`set_pgfault_handler()`函数来将`pgfault()`注册为page fault的处理函数，然后调用`sys_exofork()`函数来生成子进程，然后父进程遍历UTOP以下的地址空间，将自己的每个可写的或者写时拷贝的page映射给子进程并标记为写时拷贝，然后为子进程设置page fault的入口点，最后将子进程标记为可运行的。

```
envid_t
```

```
fork(void)
{
    // LAB 4: Your code here.
    extern void _pgfault_upcall (void);
    int i;
    int pagei;
    set_pgfault_handler(pgfault);
    envid_t childid;
    childid = sys_exofork();
    if (childid < 0)
    {
        panic("lib:fork.c:fork():fork does not success:%e",childid);
    }
    else if (childid == 0)
    {
        thisenv = &envs[ENVX(sys_getenvid())];
        return 0;
    }
    for(pagei = UTEXT/PGSIZE; pagei < UTOP/PGSIZE; pagei++)
    {
        if (pagei != (UXSTACKTOP-PGSIZE) / PGSIZE)
        {
            if (((vpd[pagei/NPTENTRIES] & PTE_P) != 0) && ((vpt[pagei] & PTE_P) != 0) && ((vpt[pagei] & PTE_U) != 0))
            {
                duppage(childid, pagei);
            }
        }
    }
    i = sys_page_alloc(childid,(void *)(UXSTACKTOP-PGSIZE),PTE_U|PTE_W|PTE_P);
    if(i < 0)
    {
        panic("lib:fork.c:fork(): exception stack error %e\n",i);
    }
    i = sys_env_set_pgfault_upcall(childid, (void *)_pgfault_upcall);
    if(i < 0)
    {
        panic("lib:fork.c:fork(): pgfault_upcall error %e\n",i);
    }
    i = sys_env_set_status(childid,ENV_RUNNABLE);
    if(i < 0)
    {
        panic("lib:fork.c:fork(): status error %e\n",i);
    }
    return childid;
    //panic("fork not implemented");
}
```

接下来是`duppage`函数，它负责将父进程的page映射给子进程，同时做写时拷贝的标记。具体就是将第pn个page映射给envid代表的env，如果该page是可写的或者是写时拷贝的，就将映射关系标记为写时拷贝，以便之后触发page fault进行相应处理。

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

    // LAB 4: Your code here.
    void* addr = (void*)(pn*PGSIZE);
    if ((uint32_t)addr >= UTOP)
    {
        cprintf("lib/fork.c:duppage: duplicate page above UTOP!");
        return -1;
    }
    if ((vpt[PGNUM(addr)] & PTE_P) ==0)
    {
        cprintf("lib/fork.c:duppage: page table not present!");
        return -1;
```

```
    }
    if ((vpd[PDX(addr)] & PTE_P) == 0)
    {
        cprintf("[lib/fork.c duppage]: page directory not present!");
        return -1;
    }
    if (vpt[PGNUM(addr)] & (PTE_W | PTE_COW))
    {
        r = sys_page_map(0, addr, envid, addr, PTE_U | PTE_P | PTE_COW);
        if (r < 0)
        {
            cprintf("lib/fork.c:duppage: sys_page_map failed1!\n");
            return -1;
        }
        r = sys_page_map(0, addr, 0, addr, PTE_U | PTE_P | PTE_COW);
        if (r < 0)
        {
            cprintf("lib/fork.c:duppage: sys_page_map failed2!\n");
            return -1;
        }
    }
    else
    {
        r = sys_page_map(0, addr, envid, addr, PTE_U | PTE_P);
        if (r < 0)
        {
            cprintf("lib/fork.c:duppage:sys_page_map failed3!\n", r);
            return -1;
        }
    }
    //panic("duppage not implemented");
    return 0;
}
```

然后是实现`pgfault`函数，它将引发page fault的地址`utf->utf_fault_va`所在的page拷贝到一个新的page，然后将新page与va形成映射。`fork`函数将其作为参数传给`set_pgfault_handler`，以便之后的page fault用其来处理。

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


    // LAB 4: Your code here.
    if ((err & FEC_WR) == 0||(vpd[PDX(addr)] & PTE_P) == 0||(vpt[PGNUM(addr)] & PTE_COW) == 0)
    {
        panic("invalid parameter!\n");
    }

    // LAB 4: Your code here.
    r = sys_page_alloc(0, (void*)PFTEMP, PTE_U | PTE_W | PTE_P);
    if (r < 0)
    {
        panic("sys_page_alloc failed: %e", r);
    }
    void* va = (void*)ROUNDDOWN(addr, PGSIZE);
    memmove((void*)PFTEMP, va, PGSIZE);
    r = sys_page_map(0, (void*)PFTEMP, 0, va, PTE_U | PTE_W | PTE_P);
    if (r < 0)
    {
        panic("sys_page_map failed: %e", r);
    }
    //panic("pgfault not implemented");
}
```

Part C

Part C要求我们实现时钟中断，从而让系统能够进行抢断式系统调度，同时还要我们实现进程间的通信。

Exercise 12

Exercise12要我们修改`kern/trapentry.S`和`kern/trap.c`，在IDT中加入中断的处理信息，并为IRQ0到15提供处理函数。仿照之前lab3的写法进行修改`kern/trapentry.S`：

```
    TRAPHANDLER_NOEC(irq0, IRQ_OFFSET+0 );
    TRAPHANDLER_NOEC(irq1, IRQ_OFFSET+1 );
    TRAPHANDLER_NOEC(irq2, IRQ_OFFSET+2 );
    TRAPHANDLER_NOEC(irq3, IRQ_OFFSET+3 );
    TRAPHANDLER_NOEC(irq4, IRQ_OFFSET+4 );
    TRAPHANDLER_NOEC(irq5, IRQ_OFFSET+5 );
    TRAPHANDLER_NOEC(irq6, IRQ_OFFSET+6 );
    TRAPHANDLER_NOEC(irq7, IRQ_OFFSET+7 );
    TRAPHANDLER_NOEC(irq8, IRQ_OFFSET+8 );
    TRAPHANDLER_NOEC(irq9, IRQ_OFFSET+9 );
    TRAPHANDLER_NOEC(irq10, IRQ_OFFSET+10 );
    TRAPHANDLER_NOEC(irq11, IRQ_OFFSET+11 );
    TRAPHANDLER_NOEC(irq12, IRQ_OFFSET+12 );
    TRAPHANDLER_NOEC(irq13, IRQ_OFFSET+13 );
    TRAPHANDLER_NOEC(irq14, IRQ_OFFSET+14 );
    TRAPHANDLER_NOEC(irq15, IRQ_OFFSET+15 );
```

`kern/trap.c`

```
    SETGATE(idt[IRQ_OFFSET+0], 0, GD_KT, irq0, 0);
    SETGATE(idt[IRQ_OFFSET+1], 0, GD_KT, irq1, 0);
    SETGATE(idt[IRQ_OFFSET+2], 0, GD_KT, irq2, 0);
    SETGATE(idt[IRQ_OFFSET+3], 0, GD_KT, irq3, 0);
    SETGATE(idt[IRQ_OFFSET+4], 0, GD_KT, irq4, 0);
    SETGATE(idt[IRQ_OFFSET+5], 0, GD_KT, irq5, 0);
    SETGATE(idt[IRQ_OFFSET+6], 0, GD_KT, irq6, 0);
    SETGATE(idt[IRQ_OFFSET+7], 0, GD_KT, irq7, 0);
    SETGATE(idt[IRQ_OFFSET+8], 0, GD_KT, irq8, 0);
    SETGATE(idt[IRQ_OFFSET+9], 0, GD_KT, irq9, 0);
    SETGATE(idt[IRQ_OFFSET+10], 0, GD_KT, irq10, 0);
    SETGATE(idt[IRQ_OFFSET+11], 0, GD_KT, irq11, 0);
    SETGATE(idt[IRQ_OFFSET+12], 0, GD_KT, irq12, 0);
    SETGATE(idt[IRQ_OFFSET+13], 0, GD_KT, irq13, 0);
    SETGATE(idt[IRQ_OFFSET+14], 0, GD_KT, irq14, 0);
    SETGATE(idt[IRQ_OFFSET+15], 0, GD_KT, irq15, 0);
```

同lab3一样，因为这里使用了本文件没有定义的处理函数，因此要先用extern标记一下。

```
extern void irq0();
extern void irq1();
extern void irq2();
extern void irq3();
extern void irq4();
extern void irq5();
extern void irq6();
extern void irq7();
extern void irq8();
extern void irq9();
extern void irq10();
extern void irq11();
extern void irq12();
extern void irq13();
extern void irq14();
extern void irq15();
```

然后根据OS网站提示还需要更改`kern/env.c`中的`env_alloc()`函数，让系统允许中断发生。

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

但是文档中没说`env_run()`函数中在`unclok_kernel()`之前也需要设置一下FL_IF位，这也是我调了半天仍不通过后问过同学才知道的，真的气。。。

Exercise 13

Exercise13需要我们修改`trap_dispatch()`函数，让系统在遇到在时钟中断后能够调用调度函数进行进程切换。

```
static void
trap_dispatch(struct Trapframe *tf)
{
    // Handle spurious interrupts
    // The hardware sometimes raises these because of noise on the
    // IRQ line or other reasons. We don't care.
    if (tf->tf_trapno == IRQ_OFFSET + IRQ_SPURIOUS)
    {
        cprintf("Spurious interrupt on irq 7\n");
        print_trapframe(tf);
        return;
    }

    // Handle clock interrupts. Don't forget to acknowledge the
    // interrupt using lapic_eoi() before calling the scheduler!
    // LAB 4: Your code here.
    if(tf->tf_trapno == IRQ_OFFSET+0 )
    {
        lapic_eoi();
        sched_yield();
    }
    // Handle processor exceptions.
    // LAB 3: Your code here.
    struct PushRegs *regs;
    int ret;
    switch(tf->tf_trapno){
        case T_DEBUG:
        case T_BRKPT:
            monitor(tf);
            break;
        case T_PGFLT:
            page_fault_handler(tf);
            break;
        case IRQ_OFFSET+IRQ_TIMER:
            lapic_eoi();
            sched_yield();
            return;
        default:
            break;
    }
    // Unexpected trap: The user process or the kernel has a bug.
    print_trapframe(tf);
    if (tf->tf_cs == GD_KT)
    {
        panic("unhandled trap in kernel");
    }
    else
    {
        env_destroy(curenv);
        return;
    }
}
```

Exercise 14

Exercise14要求我们实现进程间的通信。首先是完成`kern/syscall.c`中的`sys_ipc_recv`和`sys_ipc_try_send`函数。

`sys_ipc_recv`函数通过设置`env_ipc_recving`和`env_ipc_dstva`来表示自己需要接收信息。然后将当前进程的状态改为NOT_RUNNABLE，从而将cpu让给其他进程来运行。根据文档提示还需要注意当dstva小于UTOP并且没有页对齐时就需要返回错误。

```
static int
sys_ipc_recv(void *dstva)
{
    // LAB 4: Your code here.
    if (PGOFF(dstva) != 0 && dstva < (void*)UTOP)
    {
        return -E_INVAL;
    }
    curenv->env_ipc_recving = 1;
    curenv->env_ipc_dstva = dstva;
    curenv->env_status = ENV_NOT_RUNNABLE;
    curenv->env_ipc_from = 0;

    sched_yield();
    //panic("sys_ipc_recv not implemented");
    return 0;
}
```

`sys_ipc_try_send`函数负责发送信息，如果目标进程不是处于等待信息状态，则发送失败，返回`-E_IPC_NOT_RECV`，信息发送成功后需要将目标进程的状态进行修改，将`env_ipc_recving`修改为0表示目标进程的这次等待已结束，`env_ipc_from`更改位发送信息的进程的envid，`env_ipc_value`改为发送的信息，如果发送的信息时一个page的话，`env_ipc_perm`负责设置page的权限。按照文档提示，还需要处理一些其他的意外情况。

```
static int
sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)
{
    // LAB 4: Your code here.
    //panic("sys_ipc_try_send not implemented");
    int r;
    pte_t* pte1;
    struct Env* dstenv;
    struct Page* page1;
    if ((r = envid2env(envid, &dstenv, 0)) < 0)
    {
        return -E_BAD_ENV;
    }
    if (!dstenv->env_ipc_recving || dstenv->env_ipc_from != 0)
    {
        return -E_IPC_NOT_RECV;
    }
    if (srcva < (void*)UTOP)
    {
        if(ROUNDUP(srcva, PGSIZE) != srcva||(perm & ~PTE_SYSCALL) != 0||(perm & 5) != 5)
        {
            return -E_INVAL;
        }
        page1 = page_lookup(curenv->env_pgdir, srcva, &pte1);
        if (page1 == NULL || ((perm & PTE_W) > 0 && !(*pte1 & PTE_W) > 0))
        {
             return -E_INVAL;
        }
        if(page_insert(dstenv->env_pgdir, page1, dstenv->env_ipc_dstva, perm)<0)
        {
            return -E_NO_MEM;
        }
    }
    dstenv->env_ipc_recving = 0;
    dstenv->env_ipc_from = curenv->env_id;
    dstenv->env_ipc_value = value;
    dstenv->env_ipc_perm = perm;
    dstenv->env_tf.tf_regs.reg_eax = 0;
    dstenv->env_status = ENV_RUNNABLE;
    return 0;
```

```
}
```

然后是修改`lib/ipc.c`中的`ipc_recv`和`ipc_send`函数。
`ipc_recv`函数接收IPC发来的信息并返回，这里有几个参数，如果pg不为NULL，则要将IPC发来的page映射到pg地址，如果from_env_store不为空，则要将发送信息的进程的envid赋给*from_env_store，如果perm_store不为空，则要将发送者发来的page的perm赋给*perm_store，如果系统调用失败，则应将*fromenv和*perm均设置为0.

```
int32_t
ipc_recv(envid_t *from_env_store, void *pg, int *perm_store)
{
    // LAB 4: Your code here.
    //panic("ipc_recv not implemented");
    if (pg==NULL)
    {
        pg = (void*)UTOP;
    }
    int r = sys_ipc_recv(pg);
    if (r < 0)
    {
        if(perm_store != NULL)
        {
            *perm_store = 0;
        }
        if(from_env_store != NULL)
        {
            *from_env_store = 0;
        }
        return r;
    }
    if(from_env_store != NULL)
    {
        *from_env_store = thisenv->env_ipc_from;
    }
    if(perm_store != NULL)
    {
        *perm_store = thisenv->env_ipc_perm;
    }
    return thisenv->env_ipc_value;
}
```

`ipc_send`函数比较简单，它将val发送到to_env代表的env，同时，如果pg不为空的话，则还要发送一个page以及page的perm。它会一直循环调用`sys_ipc_try_send`函数直到发送成功，这里为了cpu的效率问题当发送不成功时调用一下`sys_yield()`让其他进程先运行。

```
void
ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
{
    // LAB 4: Your code here.
    //panic("ipc_send not implemented");
    if(!pg)
    {
        pg = (void*)UTOP;
    }
    int r;
    while((r = sys_ipc_try_send(to_env,val,pg,perm)) != 0)
    {
        if(r != -E_IPC_NOT_RECV )
        {
            panic ("lib/ipc.c:ipc_send: sys_ipc_try_send failed : %e", r);
        }
        sys_yield();
    }
}
```

最后，在syscall中注册一下，就可以运行了。

```
        case SYS_ipc_recv:
            return sys_ipc_recv((void*)a1);
        case SYS_ipc_try_send:
            return sys_ipc_try_send((envid_t)a1, a2, (void*)a3, (int)a4);
```

至此，lab4全部完成。

Challenge

本次challenge我实现的是优先级调度算法，具体策略就是在env.h中添加了priority属性，按优先级从低到高分别是0、1、2、3.在调度算法中会在当前RUNNABLE而且非IDLE的env中选择优先级最高的一个env进行运行。测试方法是取消`kern/sched.c`中的`//#define TEST_PRIORITY`宏定义的注释，然后在`kern/init.c`中创建几个不同优先级的env(将原来的TEST注释掉)，然后`make qemu CPUS =4`测试。