操统实习报告 4

1000010382 颜悦

实验报告分为三个部分：
Exercises, Questions, Challenges

Exercises

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.

全局变量 base 类似于 boot_alloc 中的 nextfree，记录了空闲段内存的起始处。
这个函数的作用也类似于 boot_alloc，主要要使用 boot_map_region.

Kern/pmap.c

```
    // Your code here:
    size_t round_size = ROUNDUP(size, PGSIZE);
    if(base + round_size > MMIOLIM){
        panic("MMIOLIM exceeded!");
    }
    boot_map_region(kern_pgdir, base, round_size, pa, PTE_PCD | PTE_PWT
| PTE_W);
    unsigned old_base = (unsigned)base;
    base = base + round_size;
    return (void *) old_base;
```

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).

阅读代码，这部分内容与 boot.S 类似，区别会在 question 中讨论。
实现内容按照注释提示以及要注意（avoid adding the page at MPENTRY_PADDR to the free list）。

Kern/pmap.c

```
    size_t i;
    pages[0].pp_ref = 1;
```

```c
for( i = 1; i < npages_basemem; i++){
    for(i == PGNUM(MPENTRY_PADDR))
        pages[i].pp_ref = 1;
    else{
        pages[i].pp_ref = 0;
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
for( i = PGNUM(IOPHYSMEM); i < PGNUM(EXTPHYSMEM); i++){
    pages[i].pp_ref = 1;
}
for( i = PGNUM(EXTPHYSMEM); i < PGNUM(PADDR(boot_alloc(0))); i++){
    pages[i].pp_ref = 1;
}
for( i = PGNUM(PADDR(boot_alloc(0))); i < npages; i++){
    pages[i].pp_ref = 0;
    pages[i].pp_link = page_free_list;
    page_free_list = &pages[i];
}
```

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().

阅读了这个 exercise 之前的的指导文档，对多核的 per_CPU 信息有了一个大概的了解，具体需要实现的时候可以在查看，完成这个 exercise，按照提示即可，同时要注意 backed by physical memory 的含义。

Kern/pmap.c

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

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.)

按照注释的内容将原来的 TSS 内容修改为 per-CPU TSS 的内容。

Kern/pmap.c

```
    // LAB 4: Your code here:


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

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

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

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

做完 4 个 exercise 后，运行 JOS，结果与教程中的一致。

Exercise 5. Apply the big kernel lock as described above, by calling lock_kernel() and unlock_kernel() at the proper locations.

按照教程中的内容添加锁操作。

Kern/init.c i386_init(void)

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

Kern/init.c mp_main(void)

```
    // Now that we have finished some basic setup, call sched_yield()
    // to start running processes on this CPU.  But make sure that
    // only one CPU can enter the scheduler at a time!
    //
    // Your code here:
    lock_kernel();
```

```
```

Kern/trap.c trap()

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

```

*Kern/env.c env_run(struct Env *e)*

```
    unlock_kernel();
    env_pop_tf(&curenv->env_tf);

    //panic("env_run not yet implemented");
}
```

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

Modify kern/init.c to create three (or more!) environments that all run the program user/yield.c. You should see the environments switch back and forth between each other five times before terminating, like this:

...
Hello, I am environment 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.

按照注释写即可，要注意这里的逻辑和流程。

Kern/sche.c

```
void
sched_yield(void)
{

    // Implement simple round-robin scheduling.
```

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

    // LAB 4: Your code here.
    struct Env * curenvptr;
    //cprintf("yyCPUid: %d\n", thiscpu->cpu_id);
    if(curenv == NULL){
        //cprintf("NULL\n");
        curenvptr = envs;
    }
    else {
        //cprintf("not NULL\n");
        curenvptr = curenv + 1;
    }
    int round = 0;
    for(; round < NENV; curenvptr++, round++){
        if(curenvptr >= envs + NENV)
            curenvptr = envs;
        if(curenvptr->env_status == ENV_RUNNABLE)
            env_run(curenvptr);

    }
    if(thiscpu->cpu_env != NULL && thiscpu->cpu_env->env_status ==
ENV_RUNNING){
        //cprintf("thiscpu\n");
        env_run(thiscpu->cpu_env);
    }
/*  else{
        cprintf("No environment left!\n");
        while(1){
            monitor(NULL);
        }
    }
```

```
*/
    // sched_halt never returns
    sched_halt();
}
```

再修改 init.c 和 syscall.c 后，运行 JOS 与题目中信息一致！

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.

这里涉及实现 5 个函数：

Kern/syscall.c

```
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;
    unsigned res = env_alloc(&e, curenv->env_id);
    if(res < 0){
        if(res == -E_NO_FREE_ENV){
            cprintf("no free env!\n");
        }
        if(res == -E_NO_MEM){
            cprintf("no free mem\n");
        }
        return res;
    }
    e->env_status = ENV_NOT_RUNNABLE;
    e->env_type = ENV_TYPE_USER;
    e->env_tf = curenv->env_tf;
    e->env_tf.tf_regs.reg_eax = 0;
    return e->env_id;
    //panic("sys_exofork not implemented");
}
```

Kern/syscall.c

```
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;
    int res = envid2env(envid, &e, 1);
    if(res < 0)return res;
    if(status == ENV_FREE){
        cprintf("ERROR: cannot find an env this way\n");
        return -E_INVAL;
    }

    if(e->env_type == ENV_TYPE_USER){
        if(status == ENV_DYING ||
            status == ENV_RUNNABLE ||
            status == ENV_RUNNING ||
            status == ENV_NOT_RUNNABLE){
            // the status is legal
            e->env_status = status;
            return 0;
        }
        else{
            cprintf("unknown status %08x\n", status);
            return -E_INVAL;
        }

    }
    return -E_INVAL;
    //panic("sys_env_set_status not implemented");
}
```

Kern/syscall.c

```
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 * target;
    struct PageInfo * p;
    int res = envid2env(envid, &target, 1);
    if(res < 0) return res;
    int perm_check = (perm ^ (PTE_AVAIL | PTE_W)) & ~(PTE_W | PTE_AVAIL
| PTE_U | PTE_P);
    if(perm_check){
        cprintf("ERROR: the permission bits are off\n");
        return -E_INVAL;
    }
    if((unsigned)va % PGSIZE != 0){
        cprintf("Va not aligned\n");
        return -E_INVAL;
    }
    if((p = page_alloc(ALLOC_ZERO))){
         //return zero
         int i = page_insert(target->env_pgdir ,p ,va , PTE_P | PTE_U |
perm);
         if(i == 0){
            return 0;
         }
         else{
            page_free(p);
            return i;
         }

    }
    else{
        cprintf("ERROR: no free memory\n");
        return -E_NO_MEM;
    }
    panic("sys_page_alloc not implemented");
}
```

Kern/syscall.c

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

    // LAB 4: Your code here.
    int res;
    struct Env *src, * dst;
    res = envid2env(srcenvid, &src, 1);
    if(res) return res;
    res = envid2env(dstenvid, &dst, 1);
    if(res) return res;

    if(((uint32_t)srcva >= UTOP || PGOFF(srcva)) ||
       ((uint32_t)dstva >= UTOP || PGOFF(dstva))){
        return -E_INVAL;
    }
    int perm_check = (perm ^ (PTE_AVAIL | PTE_W)) & ~(PTE_W | PTE_AVAIL
| PTE_U | PTE_P);
    if(perm_check){
        return -E_INVAL;
    }
    pte_t * pte;
    struct PageInfo * page = page_lookup(src->env_pgdir, srcva, &pte);
    if(!page){
        return -E_INVAL;
    }
    return page_insert(dst->env_pgdir, page, dstva, PTE_U | PTE_P | perm);
    //panic("sys_page_map not implemented");
}
```

Kern/syscall.c

```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.
    struct Env * env;
    int res;
    res = envid2env(envid, &env, 1);
    if(res < 0) return res;
    if((unsigned)va >= UTOP || PGOFF(va) ){
        return -E_INVAL;
    }
    page_remove(env->env_pgdir, va);
    return 0;
    //panic("sys_page_unmap not implemented");
}
```

Kern/syscall.c syscall()

```
        case SYS_exoforк:
            r = sys_exofork(); break;
        case SYS_env_set_status:
            r = sys_env_set_status((envid_t)a1, (int) a2); break;
        case SYS_page_alloc:
            r = sys_page_alloc((envid_t)a1, (void*)a2, (int) a3); break;
        case SYS_page_map:
            r =  sys_page_map((envid_t)a1, (void*)a2, (envid_t)a3,
(void*)a4, (int)a5); break;
        case SYS_page_unmap:
            r =  sys_page_unmap((envid_t)a1, (void*)a2); break;
```

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.

Kern/syscall.c

```
static int
sys_env_set_pgfault_upcall(envid_t envid, void *func)
{
    // LAB 4: Your code here.
    struct Env * e;
    int res = envid2env(envid, &e, 1);
    if(res < 0)return res;
    user_mem_assert(curenv, func, PGSIZE, PTE_U | PTE_P);
    e->env_pgfault_upcall = func;
    return 0;
```

```
    //panic("sys_env_set_pgfault_upcall not implemented");
}
```

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?)

张弛的报告里这部分讲的很清楚，结合栈的布局，为什么要在再次压栈的时候留出一个字的空间，这里主要参考的他的报告。

Kern/trap.c

```
// LAB 4: Your code here.
    if(curenv->env_pgfault_upcall != NULL){
        struct UTrapframe *utf;
        if(UXSTACKTOP - PGSIZE <= tf->tf_esp && tf->tf_esp < UXSTACKTOP)
            //push a empty one 4
            utf = (struct UTrapframe *)(tf->tf_esp - sizeof(struct
UTrapframe) - 4);
        else utf = (struct UTrapframe *)(UXSTACKTOP - sizeof(struct
UTrapframe));
        user_mem_assert(curenv, (void *)utf, sizeof(struct UTrapframe),
PTE_U | PTE_W);
        utf->utf_eflags = tf->tf_eflags;
        utf->utf_eip = tf->tf_eip;
        utf->utf_err = tf->tf_err;
        utf->utf_esp = tf->tf_esp;
        utf->utf_fault_va = fault_va;
        utf->utf_regs = tf->tf_regs;
        curenv->env_tf.tf_eip = (uint32_t) curenv->env_pgfault_upcall;
        curenv->env_tf.tf_esp = (uint32_t) utf;
        env_run(curenv);
    }
    // Destroy the environment that caused the fault.
    cprintf("[%08x] user fault va %08x ip %08x\n",
        curenv->env_id, fault_va, tf->tf_eip);
    print_trapframe(tf);
    env_destroy(curenv);
```

Exercise 10. Implement the _pgfault_upcall routine in lib/pfentry.S. The interesting part is returning to the original point in the user code that caused the page fault. You'll return directly there, without going back through the kernel. The hard part is simultaneously switching stacks and re-loading the EIP.

同样参考张弛的报告，写的非常清楚，太牛了。

Lib/pfentry.S

```
_pgfault_upcall:
    // Call the C page fault handler.
    pushl %esp          // function argument: pointer to UTF
    movl _pgfault_handler, %eax
    call *%eax
    addl $4, %esp           // pop function argument

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

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


    addl $0x8, %esp
    popal

    addl $0x4, %esp
    popfl
    pop %esp
    ret
```

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

为错误处理程序申请用户错误栈空间。在 kern/syscall.c 中添加对应的系统调用分配程序。

Lib/pgfault.c

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

    if (_pgfault_handler == 0) {
        // First time through!
        // 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: %e", r);
        sys_env_set_pgfault_upcall(0, _pgfault_upcall);
    }

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

```
}
```

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'

Lib/fork.c:
Pgfault() : pagefault 的处理函数，对可写的或 COW 的页，复制旧页面的数据，并建立映射。
Duppage(): 将父进程的页面映射到子进程中（共享数据），标记为 COW，为其分配新的页。
Fork(): 流程

1. 调用 set_pgfault_handler 函数对 pgfault 处理函数进行注册。
2. 调用 sys_exofork 创建一个新进程。
3. 映射可写或者 COW 的页都为 COW 页。
4. 为子进程分配 exception stack.
5. 为子进程分配页用户级的错误处理句柄。
6. 标记子进程为 runnable.

Lib/fork.c

```
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.

    // 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.
    //   No need to explicitly delete the old page's mapping.

    // LAB 4: Your code here.
    if((err & FEC_WR) == 0 || (uvpd[PDX(addr)] & PTE_P) == 0 ||
(uvpt[PGNUM(addr)] & PTE_COW) == 0)
        panic ("pgfault: not a write or attempting to access a non-COW
page");
    if((r = sys_page_alloc (0, (void *)PFTEMP, PTE_U|PTE_P|PTE_W)) < 0)
        panic("pgfault: page allocation failed : %e", r);
    addr = ROUNDDOWN (addr, PGSIZE);
    memmove (PFTEMP, addr, PGSIZE);
    if ((r = sys_page_map (0, PFTEMP, 0, addr, PTE_U|PTE_P|PTE_W)) < 0)
        panic ("pgfault: page mapping failed : %e", r);
    //panic("pgfault not implemented");
}
```

Lib/fork.c

```
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 = uvpt[PGNUM(addr)];
    if((pte & PTE_W) > 0 || (pte & PTE_COW) > 0) {
        if((r = sys_page_map (0, addr, envid, addr, PTE_U|PTE_P|PTE_COW))
< 0)
            panic ("duppage: page re-mapping failed at 1 : %e", r);
        if((r = sys_page_map (0, addr, 0, addr, PTE_U|PTE_P|PTE_COW)) <
0)
```

```
            panic ("duppage: page re-mapping failed at 2 : %e", r);
    }
    else{
        if ((r = sys_page_map (0, addr, envid, addr, PTE_U|PTE_P)) < 0)
            panic ("duppage: page re-mapping failed at 3 : %e", r);
    }
    return 0;
    //panic("duppage not implemented");
    //return 0;
}
```

Lib/fork.c

```
envid_t
fork(void)
{
    // LAB 4: Your code here.
    set_pgfault_handler (pgfault);
    envid_t envid;
    uint32_t addr;
    int r;
    envid = sys_exofork();
    if (envid < 0)
        panic("sys_exofork: %e", envid);
    if (envid == 0) {
        thisenv = &envs[ENVX(sys_getenvid())];
        return 0;
    }
    for (addr = UTEXT; addr < UXSTACKTOP - PGSIZE; addr += PGSIZE) {
        if ((uvpd[PDX(addr)] & PTE_P) > 0 && (uvpt[PGNUM(addr)] & PTE_P) >
0 && (uvpt[PGNUM(addr)] & PTE_U) > 0)
        duppage (envid, PGNUM(addr));
    }
    if ((r = sys_page_alloc (envid, (void *)(UXSTACKTOP - PGSIZE),
PTE_U|PTE_W|PTE_P)) < 0)
        panic ("fork: page allocation failed : %e", r);
    extern void _pgfault_upcall (void);
    sys_env_set_pgfault_upcall (envid, _pgfault_upcall);
    // Start the child environment running
    if ((r = sys_env_set_status(envid, ENV_RUNNABLE)) < 0)
        panic("fork: set child env status failed : %e", r);
    return envid;
    //panic("fork not implemented");
```

```
}
```

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 or checks the Descriptor Privilege Level (DPL) of the IDT entry when invoking a hardware interrupt handler. You might want to re-read section 9.2 of the 80386 Reference Manual, or section 5.8 of the IA-32 Intel Architecture Software Developer's Manual, Volume 3, at this time.

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

trapentr.S 模仿之前注册中断调用的部分。

Kern/trapentry.S

```
    #for IRQ HANDLER
    TRAPHANDLER_NOEC(routine_irq0, IRQ_OFFSET + 0);
    TRAPHANDLER_NOEC(routine_irq1, IRQ_OFFSET + 1);
    TRAPHANDLER_NOEC(routine_irq2, IRQ_OFFSET + 2);
    TRAPHANDLER_NOEC(routine_irq3, IRQ_OFFSET + 3);
    TRAPHANDLER_NOEC(routine_irq4, IRQ_OFFSET + 4);
    TRAPHANDLER_NOEC(routine_irq5, IRQ_OFFSET + 5);
    TRAPHANDLER_NOEC(routine_irq6, IRQ_OFFSET + 6);
    TRAPHANDLER_NOEC(routine_irq7, IRQ_OFFSET + 7);
    TRAPHANDLER_NOEC(routine_irq8, IRQ_OFFSET + 8);
    TRAPHANDLER_NOEC(routine_irq9, IRQ_OFFSET + 9);
    TRAPHANDLER_NOEC(routine_irq10, IRQ_OFFSET + 10);
    TRAPHANDLER_NOEC(routine_irq11, IRQ_OFFSET + 11);
    TRAPHANDLER_NOEC(routine_irq12, IRQ_OFFSET + 12);
    TRAPHANDLER_NOEC(routine_irq13, IRQ_OFFSET + 13);
    TRAPHANDLER_NOEC(routine_irq14, IRQ_OFFSET + 14);
    TRAPHANDLER_NOEC(routine_irq15, IRQ_OFFSET + 15);
```

这里 trap_init 我沿用了之前 challenge 的写法，要注意的地方是 trapentry.S 中的写法，决定了 vector 角标在在 for 循环中的初始值。

Kern/trap.c

```
void
trap_init(void)
{
```

```
    extern struct Segdesc gdt[];

    // LAB 3: Your code here.
    extern int vectors[];
    int idx;
    for(idx = 0; idx < 19; idx ++){
        int dpl = 0;
        if(idx == T_BRKPT || idx == T_OFLOW || idx == T_BOUND)dpl = 3;
        SETGATE(idt[idx], 0, GD_KT, vectors[idx], dpl);
    }

    extern int routine_syscall;
    SETGATE(idt[T_SYSCALL], 0, GD_KT, &routine_syscall, 3);

    for(idx = 0; idx < 16; idx ++){
        SETGATE(idt[IRQ_OFFSET + idx], 0, GD_KT, vectors[21 + idx], 0);
    }


    trap_init_percpu();
}
```

别忘了要开启外部中断的标记位。

Kern/env.c

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

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.

在 trap_dispatch 中增加时间中断的处理：

Kern/trap.c

```
    // 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 + IRQ_TIMER){
        //cprintf("NO: %d\n",tf->tf_trapno);
        lapic_eoi();
        sched_yield();
    }
```

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. You might find it interesting to read user/primes.c to see all the forking and IPC going on behind the scenes.

Material 中 IPC 的机制讲的比较清楚，按照提示写即可。

Kern/syscall.c

```
static int
sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned
perm)
{
    // LAB 4: Your code here.
    //curenv->env_ipc_send_to = envid;

    // LAB 4: Your code here.
    struct Env * env;
    struct PageInfo * page;
    pte_t * pte;

    if(envid2env(envid, &env, 0) < 0)
        return -E_BAD_ENV;

    if(env->env_ipc_recving == 0){
        //cprintf("E_IPC_NOT_RECV\n");
        return -E_IPC_NOT_RECV;
    }

    if(srcva && (uintptr_t) srcva < UTOP){
        if((uintptr_t) srcva % PGSIZE)
            return -E_INVAL;

        if(!(perm & PTE_P) || !(perm & PTE_U))
            return -E_INVAL;

        if((perm & 0xfff) & ~(PTE_AVAIL | PTE_P | PTE_W | PTE_U))
            return -E_INVAL;
    }
```

```
    if(srcva && env->env_ipc_dstva && ((uintptr_t) srcva < UTOP)){
        if((page = page_lookup(curenv->env_pgdir, srcva, &pte)) == NULL)
            return -E_INVAL;

        if((perm & PTE_W) && !(*pte & PTE_W))
            panic("Are you sure you want to mapping a read-only page to a
status that can be written?");

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

        env->env_ipc_perm = perm;
    }
    else {
        env->env_ipc_perm = 0;
    }
    env->env_tf.tf_regs.reg_eax = 0;
    env->env_ipc_recving  = 0;
    env->env_ipc_from     = sys_getenvid();
    env->env_ipc_value    = value;
    env->env_status       = ENV_RUNNABLE;

   // KDEBUG("\e[0;31m%08x unblocked\e[0;00m\n", env->env_id);

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

static int
sys_ipc_recv(void *dstva)
{
    // LAB 4: Your code here.
    if((uintptr_t) dstva < UTOP && ((uintptr_t) dstva % PGSIZE)) {
        return -E_INVAL;
    }

    curenv->env_ipc_value   = 0;
    curenv->env_ipc_from    = 0;
    curenv->env_ipc_perm    = 0;
    curenv->env_ipc_recving = 1;
    curenv->env_ipc_dstva   = dstva;
    curenv->env_status      = ENV_NOT_RUNNABLE;
```

```
    sched_yield ();
   // KDEBUG("\e[0;31m%08x blocked\e[0;00m\n", curenv->env_id);

    return 0;

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

Lib/ipc.c

```
int32_t
ipc_recv(envid_t *from_env_store, void *pg, int *perm_store)
{
    // LAB 4: Your code here.

    int32_t val;
    envid_t sender;
    int perm;
    if(!pg)
    pg = (void *) UTOP;
    //IPC_DEBUG("making blocking call to ipc_recv\n");
    if((val = sys_ipc_recv(pg)) < 0){
          //IPC_DEBUG("ipc_recv returned %e... dealing\n", val);
        sender = 0;
        perm = 0;
    }
    else{
          //IPC_DEBUG("ipc_recv returned %d!\n", val);
        sender = thisenv->env_ipc_from;
        perm = thisenv->env_ipc_perm;
        val = thisenv->env_ipc_value;
        //if(perm)
          // IPC_DEBUG("ipc_recv did map a page at %08x\n", pg);
    }
    if(from_env_store)
        *from_env_store = sender;
    if(perm_store)
        *perm_store = perm;
    return val;

}

// Send 'val' (and 'pg' with 'perm', if 'pg' is nonnull) to 'toenv'.
// This function keeps trying until it succeeds.
```

```
// It should panic() on any error other than -E_IPC_NOT_RECV.
//
// Hint:
//   Use sys_yield() to be CPU-friendly.
//   If 'pg' is null, pass sys_ipc_recv a value that it will understand
//   as meaning "no page".  (Zero is not the right value.)
void
ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
{
    // LAB 4: Your code here.
    int err;
    if(!pg && perm) {
      perm = 0;
    }
    else if (pg && !perm) {
      pg = 0;
    }
    while(1){
        err = sys_ipc_try_send(to_env, val, pg, perm);
        if(err == -E_IPC_NOT_RECV){
           sys_yield();
        }
        else if(!err){
          return ;
        }
        else{
            panic("ipc_send\n");
        }
    }
}
```

QEUSTIONS

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.

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.

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?

Answer:

1.

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

这句命令是用来计算 s 的物理地址的，这里使用这种方式，而在 boot/boot.S 中不使用的原因是：
在 boot.S 中，还没有进入保护模式，启动分页机制，我们可以指定任意要访问的地址，但在 mpentry.S 中，已经启动了分页机制，不能直接访问相应物理地址，但可以通过虚拟线性地址映射的方式。

2. The big kernel lock 不能保护所有内核程序的执行，在加锁之前，已经有一些 trapping 工作运行。当内核在这些位置执行时，每个进程还需要一个栈。如果多个进程都在这些没有保护的区域运行，就会相互影响之间的栈空间内容。

另一个原因，每个 CPU 都有一个 per-CPU 结构，这个结构是存在内核空间的，如果多个 CPU 共享一个内核空间，那么当一个 CPU 运行加锁之后，其它 CPU 无法访问相应的 per-CPU 结构。

3. 在切换用户，更改 cr3 前后，KERNBASE 之上的空间映射是没有变的，即所有进程访问内核空间变量的物理地址是一致的。全局变量 envs 是在内核空间中定义的，所以 e 的地址在 KERNBASE 之上，所以在切换页表前后都可以正常访问。

4. 压栈这些寄存器是存储一个进程执行的必要信息，这些信息在恢复后可以确保一个进程的继续执行，这些行为是由内核控制的。
当从用户控件切换到内核空间的时（trap, exception, interrupt），寄存器的状态在 environment 的 Trapframe 中，压栈工作是由 kern/trapentry.S 中的_alltraps 完成的，压栈的内容在内核空间的 envs 数组中，当内核切换回进程时（kern/env.c 中的 env_run()），

弹栈工作由 kern/env.c 中的 env_pop_tf()完成。

CHALLENGES

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

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

算法实现：在相同优先级下使用 round robin.

需要修改的文件包括：

Inc/env.h (添加 env 的定义内容， 定义一些优先级的宏)

Inc/syscall.h (添加 syscall number)

Inc/lib.h(注册函数)

Lib/syscall.c (在 lib 中添加设置权限的接口)

Kern/syscall.c(在 syscall 中注册，以及添加相关处理函数)

Kern/sched.c(修改调度算法)

Kern/sched.c

```
void
sched_yield(void)
{

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

    // LAB 4: Your code here.
    struct Env * curenvptr;
```

```c
    //cprintf("yyCPUid: %d\n", thiscpu->cpu_id);
    if(curenv == NULL){

        curenvptr = envs;
    }
    else {
          curenvptr = curenv + 1;
    }
    int round = 0;
    struct Env * runenv = curenvptr;
    int run_priority = ENV_PRIO_LOW - 1;
    bool find = false;
    for(; round < NENV * 10 ; curenvptr++, round++){
        if(curenvptr >= envs + NENV)
            curenvptr = envs;
        if(curenvptr->env_status == ENV_RUNNABLE &&
curenvptr->env_priority > run_priority){

            find = true;
            runenv = curenvptr;
            run_priority = curenvptr->env_priority;
        }


    }
    if(thiscpu->cpu_env != NULL && thiscpu->cpu_env->env_status ==
ENV_RUNNING &&
     thiscpu->cpu_env->env_priority > run_priority){
        env_run(thiscpu->cpu_env);
    }
    if(find /*&& run_priority >= curenv->env_priority*/){
        env_run(runenv);
    }
    if(thiscpu->cpu_env != NULL && thiscpu->cpu_env->env_status ==
ENV_RUNNING){
        env_run(thiscpu->cpu_env);
    }
    sched_halt();
}
```

Kern/env.c(在 env 创建时指定优先级为 default)

测试程序是在 dumbfork 的基础上修改的:

User/dumbfork.c

```
// Ping-pong a counter between two processes.
// Only need to start one of these -- splits into two, crudely.

#include <inc/string.h>
#include <inc/lib.h>

envid_t dumbfork(void);

void
umain(int argc, char **argv)
{
    envid_t who;
    int i;

    // fork a child process
    who = dumbfork();

    // print a message and yield to the other a few times
    for (i = 0; i < (who ? 10 : 20); i++) {
        sys_yield();
        cprintf("%d: I am the %s! priority is: %d\n", i, who ? "parent" :
"child", (&envs[ENVX(sys_getenvid())])->env_priority);
        sys_yield();
    }
}

void
duppage(envid_t dstenv, void *addr)
{
    int r;

    // This is NOT what you should do in your fork.
    if ((r = sys_page_alloc(dstenv, addr, PTE_P|PTE_U|PTE_W)) < 0)
        panic("sys_page_alloc: %e", r);
    if ((r = sys_page_map(dstenv, addr, 0, UTEMP, PTE_P|PTE_U|PTE_W)) <
0)
        panic("sys_page_map: %e", r);
    memmove(UTEMP, addr, PGSIZE);
    if ((r = sys_page_unmap(0, UTEMP)) < 0)
        panic("sys_page_unmap: %e", r);
}

envid_t
dumbfork(void)
```

```
{
    envid_t envid;
    uint8_t *addr;
    int r;
    extern unsigned char end[];

    // Allocate a new child environment.
    // The kernel will initialize it with a copy of our register state,
    // so that the child will appear to have called sys_exofork() too -
    // except that in the child, this "fake" call to sys_exofork()
    // will return 0 instead of the envid of the child.
    envid = sys_exofork();
    if (envid < 0)
        panic("sys_exofork: %e", envid);
    if (envid == 0) {
        // We're the child.
        // The copied value of the global variable 'thisenv'
        // is no longer valid (it refers to the parent!).
        // Fix it and return 0.
        thisenv = &envs[ENVX(sys_getenvid())];
        return 0;
    }

    // We're the parent.
    // Eagerly copy our entire address space into the child.
    // This is NOT what you should do in your fork implementation.
    sys_env_set_priority(sys_getenvid(), ENV_PRIO_HIGH);
    for (addr = (uint8_t*) UTEXT; addr < end; addr += PGSIZE)
        duppage(envid, addr);

    // Also copy the stack we are currently running on.
    duppage(envid, ROUNDDOWN(&addr, PGSIZE));

    // Start the child environment running
    if ((r = sys_env_set_status(envid, ENV_RUNNABLE)) < 0)
        panic("sys_env_set_status: %e", r);

    return envid;
}
```

运行结果为：

```
yy@ubuntu:~/JOS_lab4_challenge$ make run-dumbfork
make[1]: 正在进入目录 `/home/yy/JOS_lab4_challenge'
```

```
+ cc kern/init.c
+ ld obj/kern/kernel
+ mk obj/kern/kernel.img
make[1]:正在离开目录 `/home/yy/JOS_lab4_challenge'
qemu -hda obj/kern/kernel.img -serial mon:stdio -gdb tcp::26000 -D qemu.log -smp 1
6828 decimal is 15254 octal!
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2
[00000000] new env 00001000
[00001000] new env 00001001
0: I am the parent! priority is: 15
1: I am the parent! priority is: 15
2: I am the parent! priority is: 15
3: I am the parent! priority is: 15
4: I am the parent! priority is: 15
5: I am the parent! priority is: 15
6: I am the parent! priority is: 15
7: I am the parent! priority is: 15
8: I am the parent! priority is: 15
9: I am the parent! priority is: 15
[00001000] exiting gracefully
[00001000] free env 00001000
0: I am the child! priority is: 10
1: I am the child! priority is: 10
2: I am the child! priority is: 10
3: I am the child! priority is: 10
4: I am the child! priority is: 10
5: I am the child! priority is: 10
6: I am the child! priority is: 10
7: I am the child! priority is: 10
8: I am the child! priority is: 10
9: I am the child! priority is: 10
10: I am the child! priority is: 10
11: I am the child! priority is: 10
12: I am the child! priority is: 10
13: I am the child! priority is: 10
14: I am the child! priority is: 10
15: I am the child! priority is: 10
16: I am the child! priority is: 10
```

```
17: I am the child! priority is: 10
18: I am the child! priority is: 10
19: I am the child! priority is: 10
[00001001] exiting gracefully
[00001001] free env 00001001
No runnable environments in the system!
Welcome to the JOS kernel monitor!
```

符合按优先级调度!

Challenge! The JOS kernel currently does not allow applications to use the x86 processor's x87 floating-point unit (FPU), MMX instructions, or Streaming SIMD Extensions (SSE). Extend the Env structure to provide a save area for the processor's floating point state, and extend the context switching code to save and restore this state properly when switching from one environment to another. The FXSAVE and FXRSTOR instructions may be useful, but note that these are not in the old i386 user's manual because they were introduced in more recent processors. Write a user-level test program that does something cool with floating-point.

要实现这个 challenge，

需要在

Inc/trap.h 中增加表示浮点的变量和对齐（有必要的话）。

(char tf_cfloat[512], uint_32t tf_padding0[3])

修改 kern/trapentry.S 中的_alltrap 函数，压栈 fpu 内容。

```
_alltraps:

    pushw   $0x0
    pushw   %ds
    pushw   $0x0
    pushw   %es
    pushal

    /*save FPU*/

    subl $524, %esp;
    fxsave (%esp);

    movl    $GD_KD, %eax
    movw    %ax, %ds
    movw    %ax, %es
    pushl   %esp

    call  trap
```

修改 kern/env.c 中的 env_pop_tf()函数，弹栈 fpu 内容。

```
void
env_pop_tf(struct Trapframe *tf)
```

```
{
	curenv->env_cpunum = cpunum();
	__asm __volatile("movl %0,%%esp\n"
		"\tfxrstor (%%esp)\n" /*restore fpu*/
		"\taddl $524, %%esp\n"
		"\tpopal\n"
		"\tpopl %%es\n"
		"\tpopl %%ds\n"
		"\taddl $0x8, %%esp\n"
		"\tiret"
		: : "g" (tf) : "memory");
	panic("iret failed");	/* mostly to placate the compiler */
}
```

测试程序可以在 dumbfork 中增加一些浮点操作。

前三个部分比较容易实现，但第四个部分，我没有找到合适的测试方法，貌似有一些问题…