JOS-Lab-4 实验报告

熊伟伦
5120379076
azardf4yy@gmail.com

2014年11月15日-11月23日

Contents

1 前言

该报告描述了我在lab4实验的过程中遇到的问题与解决的方法，介绍了lab4的整体结构。指导中问题的解答参考上传的压缩包中的answers-lab4.txt文件

1.1 关于user/primes.c超时

经过我后来跑成绩脚本，发现原本的grade脚本设置的30秒内我跑primes只能跑到1400多，无法到1887，所以我把time out修改成了40，实际上大概需要33秒左右才能到，一个原因可能是我虚拟机的速度问题，还有部分原因是我的代码实现的效率问题。

1.2 关于ticket spinlock

使用ticket_spinlock后在之后的stressched, pingpong和primes中，由于耗时太长导致测试失败。因此我提交和自己测分的时候都把这个关掉了。助教可以看代码判断我的ticket spinlock是否正确（参考文档2.4节的exercise-4.1，代码很少），也可以在spinlock.h中#define一下跑下grade，最后几个测试会超时跑不出，但前面的测试都是正确的，由此足以证明我的ticket spinlock写的是没有问题的。

1.3 关于Challenge

Challenge我做了优先级调度那个，可以参考文档的第6章Challenge。

如果助教老师需要跑这个Challenge，需要根据第6章的说明注释替换2个文件的部分代码，分别是kern/sched.c里的部分和kern/init.c里的部分，在文档第6章有说明，代码中有注释标注。

2 Part A: Multiprocessor Support and Cooperative Multitasking

2.1 Multiprocessor Support

这里介绍了下几个概念，包括SMP，然后核又分为1个BSP和若干个APs，由硬件和BIOS决定哪个核是BSP。在SMP策略下，每个核都有一个LAPIC，这个单元负责传递终端给system，并且保存了每个核唯一的识别符。这个Lab又使用了若干个LAPIC的功能，包括获取进程使用的是哪个核，BSP发中断给APs开启其他的核，以及在Part C实现的时钟中断。

每个核通过MMIO实现LAPIC，对应于虚拟内存最顶上的32个MB。

2.2 Application Processor Bootstrap

首先BSP通过mp_init函数获取BIOS中保留的所有核的信息。

kern_init.c里的boot_aps函数一个一个boot所有的APs核，先从mpentry.S里copy启动代码到MP_ENTRYPADDR，然后调用lapic_starap函数启动核，然后进入kern_lapic.c中的该函数，此时是AP核，会调用开始copy的汇编代码，汇编代码中call mp_main，mp_main结束后会修改这个核的cpu_status，此时BSP核运行的boot_aps循环读取cpu_status获知这个AP核boot好了继续boot下一个核。

Exercise-1:这个练习看起来很明确，就是之前默认MP_ENTRY，也就是0x7000处不能分配，因为用于APs的boot的汇编代码的保存了。

根据lab1中的npages_basemem，0x7000应该在其之前，实际上是第7个Page，然后看那个汇编代码的大小应该是小于1个Page的，所以把这个page从freelist中排除应该就好了。

因此简单的修改下代码

kern/pmap.c

```
uint32_t page_mp_entry = MPENTRY_PADDR / PGSIZE;
for (i = 1; i < npages_basemem &&
     i != page_mp_entry; i++) {
    ......
}
```

如上，算出MPENTRY_PADDR的页(显然是7)，然后在循环加入free_list的判断中去除掉该页即可。此时make qemu一下通过了check_page_free_list，死在了check_kern_pgdir中，根据课程网站资料应该是没有问题的，ok。

Question-1是关于直接使用物理地址和BSP进入了Kernel之后其他的AP需要使用地址减去KERNBASE的问题，参见answers-lab4.txt文件答案。

2.3 Per-CPU State and Initialization

这里先告诉我每个CPU struct的索引方式，它的栈在memlayout.h里面定义，每个Cpu的栈大小是8个PGSIZE，按顺序在KSTACKTOP下面，每个CPU的栈中间有Gap保护间隔开。每个CPU的TSS用于保存kernel stack的状态，Taskstate定义在inc/mmu.h里。curenv对应每个CPU各一个，每个CPU的寄存器独立，

Exercise-2:练习2也是看起来不难的，按部就班的做就是，在kern/pmap.c的mem_init_mp函数中，循环调用boot_map_region就是。

kern/pmap.c

```
uint32_t i;
uint32_t per_stack_top = KSTACKTOP - KSTKSIZE;
for (i = 0; i < NCPU; i++) {
    boot_map_region(kern_pgdir, per_stack_top, KSTKSIZE,
                    PADDR(percpu_kstacks[i]), PTE_P | PTE_W );
    per_stack_top -= KSTKSIZE + KSTKGAP;
}
```

比较有趣的是原来的percpu_kstacks每个间隔8个PGSIZE，映射的虚拟地址每个间隔16个PGSIZE因为中间有没有映射的8个PGSIZE大小的GAP进行保护。

Exercise-3:在函数trap_init_percpu中，将原来的ts全部换成this->cpu_ts，某些地方加上cpu_id相关的偏移量。除此之外，由于坑爹的Lab3里添加了sysenter进行中断，在此进行5 8行的wrmsr指令功能的开启初始化，当然，可以在外部BSP循环调用APs初始化的时候就调用该指令，但以防万一，多写一次也不会出问题。

kern/trap.c

```
uint32_t i = thiscpu->cpu_id;
thiscpu->cpu_ts.ts_esp0 = KSTACKTOP-i*(KSTKSIZE+KSTKGAP);
thiscpu->cpu_ts.ts_ss0 = GD_KD;

extern void sysenter_handler();
wrmsr(0x174, GD_KT, 0);
wrmsr(0x175, thiscpu->cpu_ts.ts_esp0, 0);
wrmsr(0x176, sysenter_handler, 0);

gdt[(GD_TSS0 >> 3)+i] = SEG16(STS_T32A, (uint32_t)(&thiscpu->cpu_ts),
                    sizeof(struct Taskstate), 0);
gdt[(GD_TSS0 >> 3)+i].sd_s = 0;

ltr(GD_TSS0+(i << 3));
lidt(&idt_pd);
```

值得注意的是最后的ltr指令后3位保留所以i左移3位符合意义。随后make qemu-nox CPUS=4后输出正常，与网站资料一致。

2.4 Locking

Exercise-4:这里根据上面的提示，按部就班添加几行函数调代码就不贴了。但是还是由于坑爹的Lab3需要完成的sysenter功能，而且我还在syscall.c中使用了个包装函数my_syscall()，因此在中断进入这里的时候我开始没有lock，导致我调试调了一整天！！！整整一整天！！！都是泪，在my_syscall()也加上lock_kernel()后才能正常通过exercise-5，这里太坑了，而且还影响到我exercise-5的调度测试，这里太难调试了，我一句一句cprintf跟踪调试再和其他同学的输出结果对比才看出是这个包装函数没有加锁，都是泪。 完成了这个后make qemu-nox会卡在后面，因为我没有完成sched_yield函数。

Exercise-4.1:首先我查看了下ticket spinlock的意义和说明（参考http://www.ibm.com/developerworks/cn/linux/l-cn-spinlock/）。

理解了之后这个ticket spinlock还是很容易的，首先我在holding中使用判断own和next是否相等判断是否加了锁，如果相等此时没加锁，同时如果lock->cpu == thiscpu的话说明是该CPU加了锁，其实去掉前面一个条件同样是成立的，因为后面的条件的更新蕴含了前面的条件，但都协商也无妨。

kern/spinlock.c

```
1  static int
2  holding(struct spinlock *lock)
3  {
4  #ifndef USE_TICKET_SPIN_LOCK
5      return lock->locked && lock->cpu == thiscpu;
6  #else
7      //LAB 4: Your code here
8      //panic("ticket spinlock: not implemented yet");
9      return lock->own != lock->next && lock->cpu == thiscpu;
10
11 #endif
12 }
```

随后在spin_initlock函数里面，按照定义将own和next初始化为0，表示票号从0开始。

kern/spinlock.c

```
1  void
2  __spin_initlock(struct spinlock *lk, char *name)
3  {
4  #ifndef USE_TICKET_SPIN_LOCK
5      lk->locked = 0;
6  #else
7      //LAB 4: Your code here
8      lk->own = 0;
9      lk->next = 0;
10 #endif
11 }
```

在比较关键的spin_lock中，参考下面的第15到20行，获得锁的时候先原子性的增加next，然后得到增加前的返回值。然后无限自旋直到own的值等于得到的票号，此时认为该CPU获得该锁，符合ticket spinlock的定义。

kern/spinlock.c

```
1  void
2  spin_lock(struct spinlock *lk)
3  {
4  #ifdef DEBUG_SPINLOCK
5      if (holding(lk))
```

```
6           panic("CPU␣%d␣cannot␣acquire␣%s:␣already␣holding",
7                   cpunum(), lk->name);
8   #endif
9
10  #ifndef USE_TICKET_SPIN_LOCK
11      while (xchg(&lk->locked, 1) != 0)
12          asm volatile ("pause");
13  #else
14      //LAB 4: Your code here
15      uint32_t get_ticket = 0;
16      get_ticket = atomic_return_and_add(&(lk->next), 1);
17      while(1) {
18          if (get_ticket == lk->own)
19              break;
20      }
21
22  #endif
23
24  #ifdef DEBUG_SPINLOCK
25      lk->cpu = thiscpu;
26      get_caller_pcs(lk->pcs);
27  #endif
28  }
```

之后的spin_unlock只需要加下面一条指令即可，原子性的给own增加1，这样拿到了下一个票的CPU在自旋的时候会判断到own等于自己的ticket从而执行接下来的命令。

kern/spinlock.c

```
1   //LAB 4: Your code here
2   atomic_return_and_add(&(lk->own), 1);
```

PS:在之后的stressched和primes的测试中，由于太慢测试无法通过，因此我注释掉了#define跑整个grade。我觉得主要原因是相比于普通的spinlock，ticket spinlock由于有固定的顺序导致锁的切换实际上是更慢的，而这两个测试创建子线程又频繁的sched_yield因此就更慢了，票太多，每次还会执行小一段时间，因此超时。

2.5 Round-Robin Scheduling

Exercise-5:这里终于要实现sched了，不实现这个前面都没办法测，当然我是写完了整个Part A再开始写Part A的文档的。

这里稍微有个问题是对于之前init.c里面BSP创建的IDLE进程，到底要不要在没有其他RUNNABLE进程的时候让CPU们都进入自己的ID作为index的IDLE ENV中无限sched_yield()，因为这样做才能通过ticket_spinlock_test（因为进入monitor会获得lock其他在test函数中的CPU永远无法获得lock），但是这样又永远不能进入monitor，和题目描述中的：

“After the yield programs exit, when only idle environments are runnable,

the scheduler should invoke the JOS kernel monitor. If any of this does not happen, then fix your code before proceeding.”

冲突，因此我按照题目exercise-5的描述最后会有一个CPU进入monitor，没有按照exercise-4.1中保证每个CPU测试通过spinlock_test。但是这不代表我的ticket spinlock有问题，我的这个ticket spinlock的实现是完全正确的。(参考该文档1.2节和2.4节)

kern/sched.c

```
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);
        for(i = (env_id+1)%NENV; 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 (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);
}
```

代码看似很长，但后面一部分是已经给好的代码，我把大部分注释删除了。

按照上面贴的代码，在第11行的for循环，会每次增加env_id并且模NENV，因此是个无限循环所有的ENV，当有一个ENV_RUNNABLE并且不是ENV_TYPE_IDLE的ENV（参考上面的解释），会选择执行它，如果循环回自身了，判断原来的ENV是否是ENV_RUNNING，是的话就env_run(curenv)。后面的代码是自带的，不多加描述。

这个练习远没有上面写的这么简单，在这里我又调试了很久，因为Lab3坑爹的int30和sysenter（参考exercise-4），都需要加锁，，一开始我都不知道我的shced_yield是否写对了，因为我其他地方处理中断时候调用的lock，unlock，sched_yield都写错了！感觉挖坑太深，自掘坟墓，积累的坑和bug越来越多，lab一个比一个难写，坑像滚雪球一样，扛不住了。

2.6 System Calls for Environment Creation

Exercise-6:接下来要实现JOS里的fork()操作，和Linux的fork()定义没有区别，fork()返回0给子进程，返回子进程的pid给父进程。

首先是sys_exofork()，主要就是父进程先给子进程创建一个Env结构，并且标记为不可执行，因为这个时候还没初始化完子进程。并且子进程的返回值是0，父进程的返回值是子进程的env_id，这和定义一样，没什么好说的。

kern/syscall.c

```
1  static envid_t
2  sys_exofork(void)
3  {
4      // LAB 4: Your code here.
5      //panic("sys_exofork not implemented");
6      struct Env *child_env;
7      int r;
8      r = env_alloc(&child_env, curenv->env_id);
9      if(r < 0)
10         return r;
11     child_env->env_tf = curenv->env_tf;
12     child_env->env_status = ENV_NOT_RUNNABLE;
13     (child_env->env_tf).tf_regs.reg_eax = 0;
14     return child_env->env_id;
15 }
```

第二个实现的是sys_env_set_status()，主要是设置子进程的状态是否可执行，就像函数名一样，没什么特别要说明的。

kern/syscall.c

```
1  static int
2  sys_env_set_status(envid_t envid, int status)
3  {
4      // LAB 4: Your code here.
5      int r;
```

```
    struct Env *e;
    if ((r = envid2env(envid, &e, 1)) < 0)
        return r;
    if (status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE)
        return -E_INVAL;
    e->env_status = status;
    return 0;
}
```

第三个实现的是sys_page_alloc()，函数功能跟名字一样，给子进程分配物理页。由于函数上面的注释很详细了，一步一步按照注释做，就不多做说明了。

kern/syscall.c

```
static int
sys_page_alloc(envid_t envid, void *va, int perm)
{
    // LAB 4: Your code here.
    int r;
    struct Env *e;
    struct Page *new_page;
    if (va >= (void*)UTOP ||
        (perm & PTE_SYSCALL) != PTE_SYSCALL ||
        PGOFF(va)!= 0 || (perm & (~PTE_SYSCALL))!= 0)
        return -E_INVAL;
    r = envid2env(envid, &e, 1);
    if (r < 0)
        return -E_BAD_ENV;
    new_page = page_alloc(ALLOC_ZERO);
    if (!new_page)
        return -E_NO_MEM;
    r = page_insert(e->env_pgdir, new_page, va, perm);
    if (r < 0) {
        page_free(new_page);
        return -E_NO_MEM;
    }
    memset(page2kva(new_page), 0, PGSIZE);
    return 0;
}
```

第四个实现sys_page_map，实现共享地址映射的过程，和上面一个一样错误判断分支比较多比较烦，但是注释很详细。

kern/syscall.c

```
static int
sys_page_map(envid_t srcenvid, void *srcva,
         envid_t dstenvid, void *dstva, int perm)
{
    // LAB 4: Your code here.
    int r;
    pte_t *pte;
    struct Env *env_s;
    struct Env *env_d;
```

```
    struct Page *p;
    if (srcva >= (void *)UTOP || ROUNDUP(srcva,PGSIZE) != srcva ||
        dstva >= (void *)UTOP || ROUNDUP(dstva,PGSIZE) != dstva )
        return -E_INVAL;
    if ((perm & PTE_SYSCALL) != PTE_SYSCALL)
        return -E_INVAL;
    if ((perm & (~PTE_SYSCALL)) != 0)
        return -E_INVAL;
    r = envid2env(srcenvid, &env_s, 1);
    if (r < 0)
        return -E_BAD_ENV;
    r = envid2env(dstenvid, &env_d, 1);
    if (r < 0)
        return -E_BAD_ENV;
    p = page_lookup(env_s->env_pgdir, srcva, &pte);
    if (!p)
        return -E_INVAL;
    else if ((perm & PTE_W) != 0 && ((*pte) & PTE_W) == 0 )
        return -E_INVAL;
    r = page_insert(env_d->env_pgdir, p, dstva, perm);
    if (r < 0)
        return -E_NO_MEM;
    return 0;
}
```

最后一个要实现的是sys_page_unmap，故名思议和上面一个相反把映射解除掉，这个比较简短，错误判断少，而且还提示了调用page_remove()。

kern/syscall.c

```
static int
sys_page_unmap(envid_t envid, void *va)
{
    // LAB 4: Your code here.
    int r;
    struct Env *e;
    r = envid2env(envid, &e, 1);
    if (r < 0)
        return -E_BAD_ENV;
    if (va >= (void*)UTOP || ROUNDUP(va, PGSIZE) != va)
        return -E_INVAL;
    page_remove(e->env_pgdir, va);
    return 0;
}
```

这几个函数被后面大部分测试调用到，因此写错了很明显可以cprintf调试发现，会出现page_fault或者直接panic。调试起来不是很难，但是工作量还是比较大的，毕竟这段代码不少。

由于这是syscall，还需要在syscall里面的路由中添加对应的函数，这里返回值参考syscall.h的定义。这里参数错误会有明显的提示所以不会调试起来很坑。

kern/syscall.c

```
case SYS_exofork:
    return sys_exofork();
case SYS_env_set_status:
    return sys_env_set_status((envid_t) a1, (int) a2);
case SYS_page_alloc:
    return sys_page_alloc((envid_t) a1, (void *) a2, (int) a3);
case SYS_page_map:
    return sys_page_map((envid_t)*((uint32_t*)a1),
                        (void*)*((uint32_t*)a1+1),
                        (envid_t)*((uint32_t*)a1+2),
                        (void*)*((uint32_t*)a1+3),
                        (int)*((uint32_t*)a1+4));
case SYS_page_unmap:
    return sys_page_unmap((envid_t) a1, (void *) a2);
```

这个时候make grade是可以通过part A拿到5分的，其中艰辛不言而喻，我就这5分做了整整一个周末+3天，泪流满面。

3 Part B: Copy-on-Write Fork

这里要实现的是一个Copy-on-Write Fork，也就是fork的时候不是真写，只有在用的时候再写，这个好像ICS中提到过有点印象。

这样做的感觉是可以提高效率，就像作业拖到最后几天不得不做的时候效率就会很高，前面的时间可以尽情的玩耍，对应于计算机就是调用其他进程。

这样的操作需要维护和判断的东西比较多，也比前面的fork要复杂的多的多，感觉这个Lab跟前面3个不是一个难度级别的。

3.1 User-level page fault handling

言归正传，因为很多子进程都不用父进程继承给它的资源，因此可以在fork子进程的时候可以只传递父进程资源的映射地址而不是值，使用子进程的时候再复制这些资源的值，这样如果子进程不使用这些资源的话就不会复制，提高效率。

因为没有创建真正的值，子进程调用资源时候会产生page fault，在这时候再分配物理页并且拷贝资源，就是写时拷贝。

4 Setting the Page Fault Handler

Exercise-7:这个函数的功能是注册一个handle函数，在发生了pgfault的时候调用该函数处理，是实现整个过程的初始函数注册阶段。

kern/syscall.c

```
static int
sys_env_set_pgfault_upcall(envid_t envid, void *func)
{
    // LAB 4: Your code here.
    struct Env *e;
    int r;
    r = envid2env(envid, &e, 1);
    if(r < 0)
        return -E_BAD_ENV;
    e->env_pgfault_upcall = func;
    return 0;
}
```

代码如上所示，比较简短，就是先获取对应的envid的struct Env，然后将handle函数赋给它的env_pgfault_upcall指针。当然这里还需要处理envid2env失败的情况。

4.1 Normal and Exception Stacks in User Environments

这里说明了一个用户异常栈，在UXSTACKTOP下的一个PGSIZE，也解答了我从Lab1就开始的疑问。也就是用户自己定义的中断handle函数使用的栈空间，属于user mode。

这个问题恰好在我上周面试ipads实验室的时候夏老师问过，早点做就好了，当时回答的不是很清楚。首先发生page fault的时候会进入内核，内核会路由到用户设定的handle函数；然后进入用户的handle函数，handle函数会再UXSTACK中压Trapframe保存异常之前的状态；此时handle函数开始执行，也是在用户异常栈里执行；handle函数执行完毕之后，切换回用户的运行栈，此时重新回到user mode。

4.2 Invoking the User Page Fault Handler

Exercise-8:接下来需要真正实现这个handle函数，这个函数需要完成上面描述的如下功能：

1.先判断handle函数的处理空间是否存在，不存在就destroy env。 2.压入当前状态信息Trapeframe入异常栈。3.迭代得产生page fault，可能会重复调用pafault_handler进行处理。

这个函数的处理比较复杂的地方是要判断用户异常栈是否存在已保存的数据，需要往栈下走，这里需要判断一下，先上代码。

kern/syscall.c

```
void
page_fault_handler(struct Trapframe *tf)
{
    uint32_t fault_va;
    fault_va = rcr2();
    // LAB 3: Your code here.
```

```
7     if (!(tf->tf_cs & 0x3)) {
8         panic("kernel page fault");
9     }
10
11    // LAB 4: Your code here.
12    void* upcall = curenv->env_pgfault_upcall;
13    if (curenv->env_pgfault_upcall == NULL) {
14        // Destroy the environment that caused the fault.
15        cprintf("[%08x] user fault va %08x ip %08x\n",
16        curenv->env_id, fault_va, tf->tf_eip);
17        print_trapframe(tf);
18        env_destroy(curenv);
19    }
20    struct UTrapframe *uptf;
21    uint32_t trap_esp = tf->tf_esp;
22    uint32_t utsize = sizeof(struct UTrapframe);
23    if ((trap_esp>=UXSTACKTOP-PGSIZE) && (trap_esp<UXSTACKTOP))
24        uptf = (struct UTrapframe*)(trap_esp-utsize-4);
25    else
26        uptf = (struct UTrapframe*)(UXSTACKTOP-utsize);
27    user_mem_assert(curenv, (void*)uptf,utsize, PTE_U | PTE_W);
28    uptf->utf_esp = tf->tf_esp;
29    uptf->utf_eflags = tf->tf_eflags;
30    uptf->utf_eip = tf->tf_eip;
31    uptf->utf_regs = tf->tf_regs;
32    uptf->utf_err = tf->tf_err;
33    uptf->utf_fault_va = fault_va;
34    curenv->env_tf.tf_eip = (uint32_t)curenv->env_pgfault_upcall;
35    curenv->env_tf.tf_esp = (uint32_t)uptf;
36    env_run(curenv);
37 }
```

这里的注释比较详细，我在pdf里删除掉了。首先在第13行判断是否设置handle，如果没有就直接destroy env，简单粗暴。

如果设置了handle则继续后面的处理，在第23行需要判断esp在栈的位置，从而写在异常栈的不同地方，之后复制原来的信息，保存状态信息，然后继续env_run。

如果异常栈满了，我看好像这个JOS Lab中没有任何地方让我们处理这种情况，应该会直接崩盘。可以限制一下用户异常handle的最大迭代数解决这个问题。

4.3 User-mode Page Fault Entrypoint

Exercise-9:汇编写起来比较棘手，我承认借助了一些帮助才能写这个汇编代码。

lib/pfentry.c

```
1 // LAB 4: Your code here.
2 movl 0x30(%esp), %eax    // get old esp
3 movl 0x28(%esp), %ebx    // get old eip
4 subl $0x4, %eax          // sub esp so when pop we get
5                          // didn't change the esp behavior
6 movl %ebx, (%eax)        // move old eip to reserved space
```

```
7
8  movl %eax, 0x30(%esp)   //push oldesp-4 back
9  // Restore the trap-time registers.  After you do this, you
10 // can no longer modify any general-purpose registers.
11 // LAB 4: Your code here.
12 addl $0x8, %esp
13 popal
14
15 // Restore eflags from the stack.  After you do this, you can
16 // no longer use arithmetic operations or anything else that
17 // modifies eflags.
18 // LAB 4: Your code here.
19 addl $0x4, %esp
20 popfl
21 // Switch back to the adjusted trap-time stack.
22 // LAB 4: Your code here.
23 popl %esp
24
25 // Return to re-execute the instruction that faulted.
26 // LAB 4: Your code here.
27 ret
```

这里涉及到了用户异常栈的递归调用，如果是第一次调用就直接接着UXSTACKTOP，因为old-esp指向的是用户正常的运行栈而不是异常栈。但如果是递归调用的话old-esp指向的应该是上一次的异常栈。具体每行代码上面的代码中有注释。

Exercise-10:这里再包装一下exercise-7中实现的函数，用于真正的调用。这里先判断handler函数有没有注册，如果没有就分配一块空间给它。就在异常栈下面的一个PGSIZE。

lib/pgfault.c

```
1  void
2  set_pgfault_handler(void (*handler)(struct UTrapframe *utf))
3  {
4      int r;
5      if (_pgfault_handler == 0) {
6          // First time through!
7          // LAB 4: Your code here.
8          r = sys_page_alloc(0, (void*)(UXSTACKTOP-PGSIZE),
9                             PTE_U | PTE_P | PTE_W);
10         if(r < 0)
11             panic("set_pgfault_handler %e\n",r);
12         sys_env_set_pgfault_upcall(0, _pgfault_upcall);
13     }
14     _pgfault_handler = handler;
15 }
```

接下来执行test，有的是故意page fault的test比如deadbeef，这个测试我在sys_page_alloc在被sysenter调用的时候才发现栈组织有问题我调试了好久。在这些test里加了些cprintf用于调试，老实exercise-9我确实写不出来，因为当时我稍微

理解有偏差然后调试根本调不动，这个Lab难就难在调试太难了，又有中断，又是多核同时cprintf，很难分辨究竟是哪里的问题。

4.4 Implementing Copy-on-Write Fork

Exercise-11:接下来完成fork，这个fork和前面part A完成的fork的流程有一部分是相似的。

fork函数首先在下面代码所示的第8行调用set_pgfault_handler函数将page fault的handle设置成自己定义的函数，随后在第10行调用sys_exofork创建子进程。11到17行进行错误判断处理。

接下来的循环是在父进程中进行的，遍历UTOP一下的空间，调用duppage函数映射到子进程并标记为写时复制。需要注意的是19行判断UXSTACK所在的PGSIZE空间是不能被映射的。

随后父进程还需要给子进行分配一个PGSIZE的空间给用户异常栈作为起始使用，并且每个接下来调用的syscall都进行下错误处理增强鲁棒性，也便与调试。

lib/fork.c

```
1  envid_t
2  fork(void)
3  {
4      // LAB 4: Your code here.
5      extern void _pgfault_upcall (void);
6      int r;
7      int pno;
8      set_pgfault_handler(pgfault);
9      envid_t childid;
10     childid = sys_exofork();
11     if (childid < 0) {
12         panic("fork␣error:%e",childid);
13     }
14     else if (childid == 0) {
15         thisenv = &envs[ENVX(sys_getenvid())];
16         return 0;
17     }
18     for (pno = UTEXT/PGSIZE; pno < UTOP/PGSIZE; pno++) {
19         if (pno == (UXSTACKTOP-PGSIZE) / PGSIZE)
20             continue;
21         if (((vpd[pno/NPTENTRIES] & PTE_P) != 0) &&
22             ((vpt[pno] & PTE_P) != 0) &&
23             ((vpt[pno] & PTE_U) != 0)) {
24             duppage(childid, pno);
25         }
26     }
27     r = sys_page_alloc(childid,
28             (void *)(UXSTACKTOP-PGSIZE),
29             PTE_U|PTE_W|PTE_P);
30     if(r < 0)
```

```
31          panic("[lib/fork.c fork]: exception stack error %e\n",r);
32      r = sys_env_set_pgfault_upcall(childid,
33                      (void *)_pgfault_upcall);
34      if(r < 0)
35          panic("[lib/fork.c fork]: pgfault_upcall error %e\n",r);
36      r = sys_env_set_status(childid,ENV_RUNNABLE);
37      if(r < 0)
38          panic("[lib/fork.c fork]: status error %e\n",r);
39      return childid;
40  }
```

duppage函数完成子进程到父进程的PAGE的映射和标记为写时复制的flag，其余的都是错误处理。

而pagafault函数是给fork函数进行handler绑定的，这个函数的原理和作用开始分析过了。这两个函数理解起来不是很困难，因此我就没有贴代码，可以在lib/fork.c里进行查看。

此时make grade一下part B的测试可以全部通过。

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

Part-C主要实现时钟中断，抢占式调度和进程间的通信。

有一些进程占用的时间很长，甚至是恶意进程永远不会sched_yield()或者退出，会大量占用CPU的时间，因此需要kernel有权限能够强行sched进程，这个时候就需要时钟中断，每隔一段时间进行一次中断强行切换进程。

5.1 Interrupt discipline

Exercise-12:外部中断缩写成IRQ，一共16个对应编号就是IRQ0到IRQ15，对应在IDT中是32到47，时钟中断就是IRQ0也就是IDT的32。

还记得在Lab1的boot的时候首先要做的就是屏蔽外部中断，现在需要开启外部中断，需要eflags寄存器的FL_LF位开启就开启了外部中断。

和Lab3的中断一样，添加IRQ首先需要在kern/trapentry.S中注册对应的中断和handle函数名。然后在kern/trap.c的trap_init()函数中也同样模仿Lab3的做法先extern函数，然后调用SETGATE绑定。这里代码都比较单调我就不贴了。

5.2 Handling Clock Interrupts

Exercise-13:按照课程网站的资料，首先需要在i386init里调用lapic_init和pic_init函数，已经给我们写好了。

此时还没有路由发生了IRQ0也就是time中断的处理，因此在kern/trap.c的trap_dispatch()函数中路由这个case。

kern/trap.c

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

这样简单的处理了下之后，make grade能通过课程网站上说的测试，与其行为相符合。

5.3 Inter-Process communication (IPC)

接下来是这个Lab最后的一部分了，需要实现进程间的通信也就是IPC。

主要需要实现2个syscall，sys_ipc_recv和sys_ipc_try_send和他们的包装函数ipc_recv和ipc_sned。根据课程网站的资料，JOS里的IPC可以使用一个32位的value或者一个PGSIZE。

进行syscall调用sys_ipc_recv，该进程停止在那里等待消息，消息到了才继续执行，否则就卡在那，此时任何进程都可以调用sys_ipc_send对等待消息的进程发送消息。同时调用send的进程也会不断卡在那里知道发送成功。

5.4 Implementing IPC

Exercise-14:首先不要忘记了在syscall里添加路由。

kern/syscall.c

```
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);
```

sys_ipc_recv函数很简短明了，得到dstva后先判断时候合法，之后复制发送的信息到curenv中，然后调度。

kern/syscall.c

```
static int
sys_ipc_recv(void *dstva)
{
    // LAB 4: Your code here.
    if (ROUNDDOWN (dstva, PGSIZE) != dstva
        && dstva < (void*)UTOP)
        return -E_INVAL;
```

```
    curenv->env_status = ENV_NOT_RUNNABLE;
    curenv->env_ipc_dstva = dstva;
    curenv->env_ipc_from = 0;
    curenv->env_ipc_recving = 1;
    sched_yield();
    return 0;
}
```

这里的sys_ipc_try_send的流程和上面的解释分析一样，函数前面的注释十分详细，让人感动。比较复杂的是错误处理。

kern/syscall.c

```
static int
sys_ipc_try_send(envid_t envid, uint32_t value,
                 void *srcva, unsigned perm)
{
    // LAB 4: Your code here.
    int r;
    pte_t* pte;
    struct Env* dstenv;
    struct Page* p;
    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)
            return -E_INVAL;
        if ((perm & ~PTE_SYSCALL) != 0)
            return -E_INVAL;
        if ((perm & 5) != 5)
            return -E_INVAL;
        dstenv->env_ipc_perm = 0;
        p = page_lookup(curenv->env_pgdir, srcva, &pte);
        if (p == NULL || ((perm & PTE_W) > 0 &&
                          !(*pte & PTE_W) > 0))
            return -E_INVAL;
        if(page_insert(dstenv->env_pgdir, p,
                    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;
}
```

前面说到了，还有对应的包装函数，在lib/ipc.c中。分别是ipc_recv和ipc_send。完全按照前面的注释的内容添加即可，难度不大，再次注释感人。

lib/ipc.c

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

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 try send failed : %e", r);
    }
    sys_yield();
}
```

至此make grade可以跑75分，如果用ticket spinlock由于效率问题只能跑60分，最后的3个测试时间较多会跪掉。

6 Challenge

因为前面一部分调试的时间太久自我感觉理解的比较深入一些，又觉得第一个加小锁太麻烦而且调试反人类，而优先级调度感觉实现起来容易点，因此实现了优先级调度的Challenge。而且windows的任务管理器中同样可以设置进程的优先级。

很多地方env_priority可以参考env_status设置，因为都是用一个变量保存状态，传参的方式完全可以一样。

这里步骤比较多，我首先在lib/env.h里面的struct增加了一个

lib/env.h

```
// LAB4 challenge
uint32_t env_priority;
```

然后在inc/env.h里添加了优先级的常量

lib/env.h

```
// use for priority sched challenge
#define PRIORITY_SUPER  0x3
#define PRIORITY_HIGH   0x2
#define PRIORITY_MIDDLE 0x1
#define PRIORITY_LOW    0x0
```

然后在kern/env.c里的env_alooc()函数在初始化env的时候添加了env_priority = PRIORITY_MIDDLE。

在inc/lib.h里添加了函数的声明，在kern/syscall.c仿照sys_env_set_status写了个函数，只是把status换成了priority。

在inc/syscall.h中添加了中断号码SYS_env_set_priority。

在kern/syscall.c里面的syscall()中添加了路由。

在lib/syscall.c中添加系统调用，同样是模仿status的系统调用。

最后在kern/sched.c中进行了修改能够按优先级调度，添加了两个变量，分别保存当前搜索到的最大优先级和对应的env_id，在扫完一遍之后选择优先级最大的env执行，假设最大的一样比如有2个SUPER级别的，按照我的流程会有限执行前面一个，因为我的判定是大于而不是大于等于，所以不会覆盖。

另外我还在user文件夹下创建了对应的测试文件，并在kern/init.c中加入了这4个测试文件，在kern/Makefreg同样要添加这4个测试文件，开始的时候由于后面的反斜号忘记删除了导致莫名其妙的编译不通过，调试了一段时间才发现Orz。感觉这个Challenge难点不是调度而是要修改这么多文件注册测试和函数，最后我还在kern/init.c里BSP启动的时候创建了4个对应的不同优先级的ENV。

助教老师如果想要跑我写的测试的话，需要把kern/init.c里BSP创建的4个ENV的注释去掉，把kern/sched.c对应的challenge的部分替换掉default部分，我都用注释引了起来，直接将default加上注释将challenge部分去掉注释，改完这2个部分再make qemu就能跑出下面的结果。

Listing:

```
[00001009] Super Priority Process is Running
[00001009] Super Priority Process is Running
[00001009] Super Priority Process is Running
[00001009] exiting gracefully
```

```
[00001009] free env 00001008
[00001008] High Priority Process is Running
[00001008] High Priority Process is Running
[00001008] High Priority Process is Running
[00001008] exiting gracefully
[00001008] free env 00001009
[0000100b] Middle Priority Process is Running
[0000100b] Middle Priority Process is Running
[0000100b] Low Priority Process is Running
[0000100b] exiting gracefully
[0000100b] free env 0000100a
[0000100a] Low Priority Process is Running
[0000100a] Low Priority Process is Running
[0000100a] Low Priority Process is Running
[0000100a] exiting gracefully
[0000100a] free env 0000100b
```

不管跑多少次都是上面的结果，我再吧sched替换成原来的，无视priority的话，就是下面的结果，会出现乱序，不过很大几率还是根据我的EnV创建的顺序来的。

Listing:

```
[00001009] High Priority Process is Running
[00001009] High Priority Process is Running
[00001009] High Priority Process is Running
[00001009] exiting gracefully
[00001009] free env 00001009
[00001008] Super Priority Process is Running
[00001008] Super Priority Process is Running
[00001008] Super Priority Process is Running
[00001008] exiting gracefully
[00001008] free env 00001008
[0000100a] Middle Priority Process is Running
[0000100a] Middle Priority Process is Running
[0000100a] Middle Priority Process is Running
[0000100a] exiting gracefully
[0000100a] free env 0000100a
[0000100b] Low Priority Process is Running
[0000100b] Low Priority Process is Running
[0000100b] Low Priority Process is Running
[0000100b] exiting gracefully
[0000100b] free env 0000100b
```

7 总结

这个lab把我lab3的漏洞充分暴露无遗，尤其是partA和partB的sys_page_alloc的调用，还是sysenter的锅，我大部分时间花在了lab的前一半部分，反而后面的部分注释比较详细，在中断和调度写对的情况下按部就班的写难度比前面低。前面调试多核和中断太难了，原来CSE的lab调试一个多线程的状态机就已经够难了，这下配上中断，很多调试的cprintf是乱序的只能脑补，而且中断的各种调用不是很好直接调试。

总的来说还是由于我对Lab3的中断有些地方理解的不深入导致这个lab的part A花了大量的时间调试。希望后面的Lab5和Lab6能和前面4个Lab的关系小一些。