LAB 4：PREEMPTIVE MULTITASKING

2015/12/3　　学号：5120809022　　姓名：黄志强

1. 关于 lab 的说明

1. challenge 写的是优先级调用，具体说明在整个文档的最后；
2. grade-lab4.sh 这个文件中的最后一个 test-primes，会出现超时的问题，把 timeout 改成 40 而不是 30 就可以通过。

2. Multiprocessor Support and Cooperative Multitasking

1. 这个部分主要做一下工作：

   1）初始化多个 cpu

   2）Lock 保证只有一个 cpu 能访问内核代码

   3）多任务的 schedule 调度

   4）与 dumbfork 有关的实现。

2. 初始化多个 cpu

   按照文章所说，先启动 BSP，再启动 APS。启动 BSP 的时候要为启动 APS 做好准备，主要的准备工作如下：

   1）更改 page_init(),把 MPENTRY_PADDR 位置设置为已用的（这个物理页用来存储 APS 的启动代码，所以不能被 alloc），代码如下：

```
(pagePhysAddr>=IOPHYSMEM&&pagePhysAddr<firstFreePhysAddr)
  ||(pagePhysAddr==MPENTRY_PADDR))
 {
  pages[i].pp_ref=1;
  pages[i].pp_link=NULL;
 }
```

   其实加入一行判断代码就可以。

   2）mem_init()加入代码 `mem_init_mp();`实现 mem_init_mp():

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

为每个 cpu 设置相应的内核 stack。

3） 为每个 cpu 创建 gdt 表里面的 TSS 描述子，代码在 trap_init_percpu()中：

```
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);

// Initialize the TSS slot of the gdt.
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;

// Load the TSS selector (like other segment selectors, the
// bottom three bits are special; we leave them 0)

ltr(GD_TSS0+(i << 3));

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

准备工作做好了之后就是启动 APS 了，这部分代码在汇编 mpentry.S 文件中。

3. Lock 保证一个 cpu 访问内核代码，按照提示走就可以了，除了这些以外，在 kern/syscall.c 文件中，自己加入的函数 my_syscall()也要 lock_kernel。这里有个 exercise 是要实现 spinlock，按照要求做，

Holding()函数中：

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

判断当前 cpu 是否拿了锁。

__spin_initlock(struct spinlock *lk, char *name)：

```
lk->own = 0;
lk->next = 0;
```

初始化 lock。

spin_lock(struct spinlock *lk)：

```
uint32_t ticket ;
ticket = atomic_return_and_add(&(lk->next), 1);
while(1)
{
    if (ticket == lk->own)
    {
        break;
```

```
        }
    }
```

判断 lock 是否轮到当前 cpu。

spin_unlock(struct spinlock *lk)：

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

把 lock 给下一个拿 ticket 的 cpu。

4. 任务调度，主要实现 sched.c 文件中的 sched_yield(void)函数：

```
    uint32_t id;
    if(curenv != NULL)
    {
        id = ENVX(curenv->env_id);
        for(i = (id+1)%NENV; i != 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);
        }
    }
```

逻辑很简单，从当前位置一直往下找，到再次轮到当前位置为止，找下一个离当前跑的 env 最近的符合条件（状态为可运行，类型不为空间类型的 env）的 env，运行 env 即可。如果不存在，判断当前位置的 env 是否为正在运行，如果正在运行则继续跑下去，不然就找不到了。

5. Dumbfork 的相关实现：

sys_exofork(void)（注意这个调用是通过 trap 的 48，调用 trap_handler 实现的，而不是 syscall，因此注意在 trap 和 trapentry.S 里面实现接口，不然会无法调用的，这里就不写实现，就几行代码）

主要就是 alloc 一个 env，设置一下状态，代码比较简单：

```
    struct Env *e;
    int r;
    r = env_alloc(&e, curenv->env_id);
    if(r < 0)
        return r;
    e->env_tf = curenv->env_tf;
    e->env_status = ENV_NOT_RUNNABLE;
    e->env_tf.tf_regs.reg_eax = 0;
    return e->env_id;
```

这个进程是不能跑的，因此设置为 not runnable。

sys_env_set_status(envid_t envid, int status)设置 env 的状态，代码：

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

sys_page_alloc(envid_t envid, void *va, int perm) 主要就是 alloc 一个 page，并且把这个 page 映射到 va，代码如下：

```
struct Env *e;
struct Page *p;
int r;
if (va >= (void*)UTOP || (perm & 5) != 5
    || PGOFF(va) != 0 || (perm & (~PTE_SYSCALL)) != 0)
    return -E_INVAL;
r = envid2env(envid, &e, 1);
if (r < 0)
    return -E_BAD_ENV;
p = page_alloc(ALLOC_ZERO);
if (!p)
    return -E_NO_MEM;
r = page_insert(e->env_pgdir, p, va, perm);
if (r < 0){
    page_free(p);
    return -E_NO_MEM;
}
memset(page2kva(p), 0, PGSIZE);
return 0;
```

主要就是 page_alloc()和 page_insert()函数的调用。

sys_page_map(envid_t srcenvid, void *srcva, envid_t dstenvid, void *dstva, int perm)

主要就是把 srcenvid 这个 env 的 srcva 对应的物理 page 映射到 dstenv 这个 env 对应的虚拟地址 dstva。主要也是 page_lookup(0 和 page_insert()函数的调用：

```
struct Env* s_env;
struct Env* d_env;
struct Page* p;
int r;
pte_t* pte;
if (srcva >= (void*)UTOP || ROUNDUP(srcva,PGSIZE) != srcva ||
    dstva >= (void*)UTOP || ROUNDUP(dstva,PGSIZE) != dstva
    ||(perm & 5) != 5||(perm & (~PTE_SYSCALL)) != 0 )
    return -E_INVAL;
r = envid2env(srcenvid, &s_env, 1);
if (r < 0)
    return -E_BAD_ENV;
r = envid2env(dstenvid, &d_env, 1);
if (r < 0)
    return -E_BAD_ENV;
```

```
p = page_lookup(s_env->env_pgdir, srcva, &pte);
if (p == NULL)
    return -E_INVAL;
if ((perm & PTE_W) != 0 && ((*pte) & PTE_W) == 0 )
    return -E_INVAL;
r = page_insert(d_env->env_pgdir, p, dstva, perm);
if (r < 0)
    return -E_NO_MEM;
return 0;
```

sys_page_unmap(envid_t envid, void *va)主要是把 envid 这个 env 的 va 地址对应的物理 page 这个映射给 release 掉，主要就是 page_remove()这个函数的调用：

```
struct Env *e;
int r;
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;
```

当然每个函数都要在 kern/syscall.c 和 lib/syscall.c 里面设置入口点（sys_fork()除外），一边 user mide 能够调用。

到这里位置，partA 部分就结束了。

3. Copy-on-Write Fork

1. 这个部分的主要工作如下：

   1） 设置 page fault 的 handler，

   2） 调用 page fault 的 handler

   3） Fork 函数

   来讲述一下整个过程吧:首先，用户程序用 set_pgfault_handler(void (*handler)(struct UTrapframe *utf))把汇编 pfentry.S 里面的_pgfault_handler 设置成传进来的 handler，把用户的 user mode 的 page fault 设置成汇编 pfentry.S 中的_pgfault_upcall，这样 page fault 发生时就会跳到汇编 pfentry.S 进行处理。其次，处理完成之后，跳回到用户发生 page fault 的位置继续执行。最后，实现 fork 函数，设置 parent 和 child 对应的物理页为 COW，访问物理页 write 的时候发生 page fault，进入处理函数，完成整个过程。

2. 设置 page fault 的 handler。

   set_pgfault_handler(void (*handler)(struct UTrapframe *utf))给用户提供设置接口，让用户实现自己的 handler，代码：

```
set_pgfault_handler(void (*handler)(struct UTrapframe *utf))
```

```
{
    int r;
    if (_pgfault_handler == 0) {
        // First time through!
        // LAB 4: Your code here.
        uint32_t id=sys_getenvid();
        r = sys_page_alloc(id, (void*)(UXSTACKTOP-PGSIZE),
             PTE_U | PTE_P | PTE_W);
        if(r < 0)
            panic("set_pgfault_handler %e\n",r);
        sys_env_set_pgfault_upcall(id, _pgfault_upcall);
    }
    _pgfault_handler = handler;
}
```

过程与上面讲述的一样，这里的 sys_env_pgfault_upcall()进行实现，代码在 lib/syscall.c 里面，主要就是调用系统的 sys_env_set_pgfault_upcall(envid_t envid, void *func)，系统实现如下：

```
struct Env *e;
int r;
r = envid2env(envid, &e, 1);
if(r < 0)
    return -E_BAD_ENV;
e->env_pgfault_upcall = func;
return 0;
```

在 lib/syscall.c 和 kern/syscall.c 里面实现相应的接口，这里就不再说明。

3. 调用 page fault 的 handler。

   在 trap.c 的 page_fault_handler(struct Trapframe *tf)函数中进行 page fault 的处理。

   代码：

```
if (curenv->env_pgfault_upcall == NULL) {
    // 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);
}
struct UTrapframe *utf;
uint32_t trap_esp = tf->tf_esp;
uint32_t size = sizeof(struct UTrapframe);
if ((trap_esp>=UXSTACKTOP-PGSIZE) && (trap_esp<UXSTACKTOP))
    utf = (struct UTrapframe*)(trap_esp-size-4);
else
    utf = (struct UTrapframe*)(UXSTACKTOP-size);
user_mem_assert(curenv, (void*)utf,size, PTE_U | PTE_W);
utf->utf_esp = tf->tf_esp;
utf->utf_eflags = tf->tf_eflags;
utf->utf_eip = tf->tf_eip;
utf->utf_regs = tf->tf_regs;
utf->utf_err = tf->tf_err;
utf->utf_fault_va = fault_va;
curenv->env_tf.tf_eip = (uint32_t)curenv->env_pgfault_upcall;
```

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

主要逻辑就是先判断是否存在设置好的 page fault handler，不存在，就 destroy，不然就在 UXSTACKTOP 建立一个 stack，使得 stack 结构如下所示：

```
                    <-- UXSTACKTOP
trap-time esp
trap-time eflags
trap-time eip
trap-time eax       start of struct PushRegs
trap-time ecx
trap-time edx
trap-time ebx
trap-time esp
trap-time ebp
trap-time esi
trap-time edi       end of struct PushRegs
tf_err (error code)
fault_va            <-- %esp when handler is run
```

至于判断 trap_esp，是为了指导当前是否是第一次进入 handler，如果不是，说明之前已经进入 handler，然后处理 handler 过程中又发生了 page fault，这个时候就要先 push 一个 4byte（并没有说明为什么要这样，我猜大概是为了模仿 call 结构，call 会先 push 4byte 的 ret addr，这里模仿一下），在 push 上面的栈结构。

因为 *curenv->env_pgfault_upcall* 必然是 pfentry.S 中的_pgfault_upcall，因此就会进入这个函数进行处理，这个函数的过程如下：

```
    pushl %esp          // function argument: pointer to UTF
    movl _pgfault_handler, %eax
    call *%eax
    addl $4, %esp           // pop function argument
    movl 0x30(%esp), %eax   // get old esp
    movl 0x28(%esp), %ebx   // get old eip
    subl $0x4, %eax
    movl %ebx, (%eax)   // move old eip to reserved space

    movl %eax, 0x30(%esp)   //push oldesp-4 back
    // 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
```

_pgfault_handler 就是用户设置的 handler（在 set_pgfault_handler 函数中设置），处理完成以后，

根据上面的 stack 的图：

```
    movl 0x30(%esp), %eax    // get old esp
    movl 0x28(%esp), %ebx    // get old eip
```

拿的就是原来的 esp 和 eip，这里的 esp 就是 user mode 执行到 page fault 点的 stack 的 esp 值，eip 就是 user mode 执行到 page fault 点的 eip 的值，因此，

```
    subl $0x4, %eax
    movl %ebx, (%eax)    // move old eip to reserved space
```

把 user mode 的 stack 向下扩展 4byte，把 eip 存储在那个位置，

```
    movl %eax, 0x30(%esp)    //push oldesp-4 back
```

把向下扩展之后的 esp 的值存储在上述图的存储 esp 的位置，相当于更改了 user mode 的 stack esp 的值，

```
    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
```

简单的向上增加 esp，使得 esp 的位置执行上述图存储 esp 值得那个位置，然后

```
    popl %esp

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

popl %esp 就把 esp 设置为 user mode stack esp 向下扩展之后的 esp 的位置，ret 就把当前扩展的 esp 中存储的值（即 eip）的值赋值给 eip，现在 eip 就是 user mode 执行到 page fault 点的 eip 的值，因此控制就交还给了 user，继续向下执行。

(very beautiful !!!)

4. 实现 fork

```
    extern void _pgfault_upcall (void);
    int r;
    int pno;
    set_pgfault_handler(pgfault);
    envid_t id;
    id = sys_exofork();
    if (id < 0) {
        panic("fork error:%e",id);
    }
```

```
    else if (id == 0) {
        thisenv = &envs[ENVX(sys_getenvid())];
        return 0;
    }
    for (pno = UTEXT/PGSIZE; pno < UTOP/PGSIZE; pno++) {
        if (pno == (UXSTACKTOP-PGSIZE) / PGSIZE)
            continue;
        if (((vpd[pno/NPTENTRIES] & PTE_P) != 0) &&
             ((vpt[pno] & PTE_P) != 0) &&
            ((vpt[pno] & PTE_U) != 0)) {
            duppage(id, pno);
        }
    }
    r = sys_page_alloc(id,(void *)(UXSTACKTOP-PGSIZE),PTE_U|PTE_W|PTE_P);
    if(r < 0)
        panic("[lib/fork.c fork]: exception stack error %e\n",r);
    r = sys_env_set_pgfault_upcall(id, (void *)_pgfault_upcall);
    if(r < 0)
        panic("[lib/fork.c fork]: pgfault_upcall error %e\n",r);
    r = sys_env_set_status(id,ENV_RUNNABLE);
    if(r < 0)
        panic("[lib/fork.c fork]: status error %e\n",r);
    return id;
```

主要逻辑就是首先设置好 handler，其次将 UTEXT-UTOP 之间的物理页同时映射给 parent 和 child，最后，设置 child 的状态。注意 1）exception stack 不能共享，必须分开；2）设置 handler 的时候，set_pgfault_handler(pgfault)只会设置当前 curenv 的 handler，即 parent，因此后面 child 的 handler 要 parent、设置（child 自己也可以设置），代码就按照逻辑写就可以了，主要注意的是 pgfault 这个 handler（一个函数），和 duppage()函数。

Duppage():

```
    void* addr = (void*)(pn*PGSIZE);
    if ((uint32_t)addr >= UTOP)
        panic("duppage: duplicate page above UTOP!");
    pde_t pde;
    pde = vpd[PDX(addr)];
    if ((pde & PTE_P) == 0)
        panic("[lib/fork.c duppage]: page directory not present!");
    pte_t pte;
    pte = vpt[PGNUM(addr)];
    if ((pte & PTE_P) == 0)
        panic("[lib/fork.c duppage]: page table not present!");
    int r;
    uint32_t id=sys_getenvid();
    if (pte & (PTE_W | PTE_COW))
    {
        r = sys_page_map(id, addr, envid, addr,
            PTE_U | PTE_P | PTE_COW);
        if (r < 0)
            panic("[lib/fork.c duppage]: map page copy on write %e", r);
        r = sys_page_map(id, addr, id, addr,
            PTE_U | PTE_P | PTE_COW);
        if (r < 0)
```

```
            panic("[lib/fork.c duppage]: map page copye on write %e", r);
    }
    else
    {
        r = sys_page_map(id, addr, envid, addr, PTE_U | PTE_P);
        if (r < 0)
            panic("[lib/fork.c duppage]:map page in read only %e", r);
    }
    return 0;
```

除了那些判断代码之外，主要逻辑就是在 page table 寻找传进来 parent page 的 pte，如果 pte 是读的，把这个 page 映射给 child，如果是 COW 或者 W 的，就把 page 映射给 child 和 parent，同时设成 COW（为什么要设置 parent?）因为 parent 改写物理页也是会引起物理页的更改，也要调用 handler，因此 parent 也要设置。

pgfault(struct UTrapframe *utf)：

```
if ((err & FEC_WR) == 0)
    panic("[lib/fork.c pgfault]: not a write fault!");
if ((vpd[PDX(addr)] & PTE_P) == 0)
    panic("[lib/fork.c pgfault]: page directory not exists!");
if ((vpt[PGNUM(addr)] & PTE_COW) == 0)
    panic("[lib/fork.c pgfault]: not a copy-on-write fault!");
uint32_t id=sys_getenvid();
r = sys_page_alloc(id, (void*)PFTEMP, PTE_U | PTE_W | PTE_P);
if (r < 0)
panic("[lib/fork.c pgfault]: alloc temporary location failed %e", r);
void* va = (void*)ROUNDDOWN(addr, PGSIZE);
memmove((void*)PFTEMP, va, PGSIZE);
r = sys_page_map(id, (void*)PFTEMP, id, va, PTE_U | PTE_W | PTE_P);
if (r < 0)
    panic("[lib/fork.c pgfault]: %e", r);
```

处理判断之外，主要逻辑就是新建一个 page（sys_page_alloc），把当前 page 里面的内容移到新建的 page 里面（memmove），更改 page table（sys_page_map）就可以了。

到此，partB 就完成了。

4. Preemptive Multitasking and Inter-Process communication (IPC)

1. 这部分工作主要为：

   1） 时钟中断和 IRQ；

   2） IPC

2. 时钟中断和 IRQ

   IRQ 在 trap.c 和 trapentry.S 更改，代码比较简单，按照以前写的就可以了，对于时钟中断处理函数，只有 2 行代码：

```
if(tf->tf_trapno == IRQ_OFFSET+0 ){ //clock interrupts
    lapic_eoi();
```

```
        sched_yield();
    }
```

然后为了保证时钟中断正常处理，在 env_alloc()函数中：

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

开启中断，进入 kernel 的 trap handler 关闭中断：

```
    asm volatile("cld" ::: "cc");
```

这样正好符合逻辑。

3. IPC

整个逻辑是用户调用 lib/ipc 的借口，ipc 调用 lib/syscall.c 的接口，lib/syscall.c 调用 kern/syscall.c 进行处理，这里主要介绍 lib/ipc，和 lib/syscall.c，至于一些信号和接口比较简单，这里就不再说明。

IPC 文件：

```
ipc_recv(envid_t *from_env_store, void *pg, int *perm_store)
{
    // LAB 4: Your code here.
    //panic("ipc_recv not implemented");
    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;
}
```

主要逻辑就是调用系统的 sys_ipc_recv(pg)，实现：

```
sys_ipc_recv(void *dstva)
{
    // LAB 4: Your code here.
    //panic("sys_ipc_recv not implemented");
    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;
    curenv->env_ipc_perm = 0;
    sched_yield();
    return 0;
}
```

主要逻辑就是首先把运行状态设置成不能运行，其次把 env_ipc_recving 设置成 1，表示这个 env 进入接受状态，最后调用 sche_yield()挂起这个 env，等待它的状态变成可运行状态的时候再运行（即已经收到了）。

```
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;
    r = sys_ipc_try_send(to_env,val,pg,perm);
    while(r != 0)
    {
        if(r != -E_IPC_NOT_RECV )
        panic ("[lib/ipc.c ipc_send]: sys try send failed : %e", r);
        r = sys_ipc_try_send(to_env,val,pg,perm);
    }
    sys_yield();
}
```

逻辑就是不断的调用系统的 send 函数，直到接受或者 panic，然后调用 sys_yield()(为什么？)因为挂起的 receive 的那个 env 在系统 send 函数里面已经改成了 RUNNABLE 状态，调用 sys_yield()可以再次调度以便选择那个 receive 的 env 继续运行，当然这不是必须的。函数代码：

```
    if(!pg)
        pg = (void*)UTOP;
    int r;
    r = sys_ipc_try_send(to_env,val,pg,perm);
    while(r != 0)
    {
        if(r != -E_IPC_NOT_RECV )
        panic ("[lib/ipc.c ipc_send]: sys try send failed : %e", r);
        r = sys_ipc_try_send(to_env,val,pg,perm);
    }
    sys_yield();
```

系统 send 代码：

```
    struct Env* dstenv;
    struct Page* p;
    int r;
    pte_t* pte;
    r = envid2env(envid, &dstenv, 0);
    if (r < 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;
        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_perm=perm;
    }
    else
        dstenv->env_ipc_perm = 0;
    dstenv->env_ipc_recving = 0;
    dstenv->env_ipc_from = curenv->env_id;
    dstenv->env_ipc_value = value;
    dstenv->env_tf.tf_regs.reg_eax = 0;
    dstenv->env_status = ENV_RUNNABLE;
    return 0;
```

主要逻辑就是插入 page 到 page table，然后把 env_ipc_recving 设置为 0，表示不接收，把 dstenv->env_status = ENV_RUNNABLE;，保证 receive 的 env 能够继续运行。

到此，partC 结束了。

5. Challenge

我做的 challenge 是优先级调度，代码比较简单，主要是更改 kern 的 MakeFrag 文件，并且在 user 目录下写自己的测试文件，同时在 env 中加入变量 env_priority 表示优先级，代码分为 3 部分.

1. sche.c 文件：

```
/*uint32_t max_priority = 0;
int32_t max_id = -1;
uint32_t id;
if(curenv != NULL)
{
    id = ENVX(curenv->env_id);
    for(i = 0; i <NENV;i++)
    {
        if((envs[i].env_status == ENV_RUNNABLE||
            envs[i].env_status == ENV_RUNNING) &&
             envs[i].env_type != ENV_TYPE_IDLE &&
            envs[i].env_priority >= max_priority)
        {
            max_priority = envs[i].env_priority;
            max_id = i;
        }
    }
    if(id==max_id)
    {
        env_run(curenv);
    }
```

```
        else
        {
            env_run(&envs[max_id]);
        }
    }
```

逻辑比较简单，就是轮循一遍，找出最大优先级的调度，注意，如果正在运行的优先级最大，那么再次调度的时候还会是它（我是这样设计的）。

2. 然后再 init.c 建立 3 个优先级的 env：

```
/*ENV_CREATE(user_priority_low, ENV_TYPE_USER);
 ENV_CREATE(user_priority_middle, ENV_TYPE_USER);
 ENV_CREATE(user_priority_high, ENV_TYPE_USER);
 envs[8].env_priority=PRIORITY_LOW;
 envs[9].env_priority=PRIORITY_MIDDLE;
 envs[10].env_priority=PRIORITY_HIGH;*/
```

我按照 low middle high 的优先级建立 env（这样更有说服力，因为调度总是先调度前面的，我后面的先调度足够说明是优先级的原因，排除调度的原因），然后运行结果总是：

```
[00000000] new env 00001006
[00000000] new env 00001007
[00000000] new env 00001008
[00000000] new env 00001009
[00000000] new env 0000100a
[    100a] Priority High is Running!
[    100a] Priority High is Running!
[    100a] Priority High is Running!
[0000100a] exiting gracefully
[0000100a] free env 0000100a
[    1009] Priority Middle is Running!
[    1009] Priority Middle is Running!
[    1009] Priority Middle is Running!
[00001009] exiting gracefully
[00001009] free env 00001009
[    1008] Priority Low is Running!
[    1008] Priority Low is Running!
[    1008] Priority Low is Running!
[00001008] exiting gracefully
[00001008] free env 00001008
No more runnable environments!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
```

足够说明我代码的正确性。

3. 我写了关于设置优先级（用户设置的）接口，但是我在测试程序里面并没有调用，原因如下：

   1）init.c 他会立即进入 sche.c 文件，这个时候并没有运行测试代码，那么优先级接口也就没有调用，因此这个时候选择哪个 env 是与优先级无关的（因为还没有设置），所以拿不到正确的结果。

2） 就算在测试文件里面加入优先级设置代码，然后调用 sys_yield()，这个时候一个 env 设置完成并不代表另外 2 个也设置完成了，如果另外两个还没有设置，也得不到正确的结果。

3） 由于时钟中断也会调用 sche.c，使得结果更加不确定。（虽然可以注释，但还有上面 2 个原因）。

4） 因此，我就在 init.c 里面直接设置优先级，然后 make qemu 测试，这样保证拿到的一定是正确的结果。

整个 lab4 到此结束。