

---

# Operating System MIT 6.828 JOS Lab4 Report

Computer Science  
ChenHao(1100012776)

2013 年 11 月 25 日

## 目录

<b>1</b>	<b>Part A: Multiprocessor Support and Cooperative Multitasking</b>	<b>3</b>
1.1	Exercise 1 . . . . .	3
1.2	Exercise 2 . . . . .	3
1.3	Question 1 . . . . .	4
1.4	Exercise 3 . . . . .	4
1.5	Exercise 4 . . . . .	4
1.6	Exercise 5 . . . . .	5
1.7	Question 2 . . . . .	5
1.8	Exercise 6 . . . . .	5
1.9	Question 3-4 . . . . .	6
1.10	Challenge 2 . . . . .	7
1.11	Question 7 . . . . .	9
1.11.1	sys_exofork . . . . .	9
1.11.2	sys_env_set_status . . . . .	10
1.11.3	sys_page_alloc . . . . .	10
1.11.4	sys_page_map . . . . .	11
1.11.5	sys_page_unmap . . . . .	11
<b>2</b>	<b>Part B: Copy-on-Write Fork</b>	<b>12</b>
2.1	Exercise 8 . . . . .	12
2.2	Exercise 9 . . . . .	13
2.3	Exercise 10 . . . . .	14
2.4	Exercise 11 . . . . .	15
2.5	Exercise 12 . . . . .	15
2.5.1	fork . . . . .	16

---

2.5.2	duppage	17
2.5.3	pgfault	17
<b>3</b>	<b>Part C: IPC</b>	<b>19</b>
3.1	Exercise 13	19
3.2	Exercise 14	20
3.3	Exercise 15	20
3.4	sys_ipc_recv	20
3.5	sys_try_send	21
3.6	ipc_recv	22
3.7	ipc_send	22
<b>4</b>	<b>Summary</b>	<b>24</b>
4.1		24
4.2	Endless Debug	24

---

## 1 Part A: Multiprocessor Support and Cooperative Multitasking

Part A 是在 JOS 上实现多核，采用 SMP 模型，即对每个 CPU 都有相同访问系统资源和 I/O 总线的能力。SMP 将 CPU 分为两类，一个是 BSP 负责对系统进行启动，以及 APs 由 BSP 在启动系统后启动。APIC 由两部分组成，一部分是 IOAPIC，其功能是负责将中断信息分派给每颗 CPU 的 LAPIC，另一部分就是 LAPIC，其决定是否接受总线上传递的中断信息，LAPIC 也可以获得 CPU 的信息。

一般来说，访问 LAPIC 是通过 MMIO 的技术来实现的，将某一段内存和 I/O 链接起来，一般来说 LAPIC 是在物理内存的 0xFE000000 处，但是在 JOS 没有这么大的内存，因此用虚拟内存的高 32M 作为该段的 MMIO。

### 1.1 Exercise 1

这是是要我们实现一个用于简单分配内存空间进行映射作为 MMIO，利用以前写的 `boot_map_region` 就可以很容易的实现。

```
void *
mmio_map_region(physaddr_t pa, size_t size)
{
    static uintptr_t base = MMIOWBASE;

    pa = ROUNDDOWN(pa, PGSIZE);
    boot_map_region(kern_pgdir, base, ROUNDUP(size, PGSIZE), pa, PTE_PCD |
        PTE_PWT | PTE_W);
    uintptr_t tmp_base = base;
    base += ROUNDUP(size, PGSIZE);
    return (void *) tmp_base;
}
```

### 1.2 Exercise 2

这一段是在说明 AP 是如何启动的，BSP 在启动 APs 前，会先收集 CPU 信息，APIC IDs, LAPIC 的 MMIO 地址等信息，然后执行 `boot_aps`，在这段中 BSP 依次让每个 AP 启动执行 `mpentry.S` 中的代码（BSP 将代码复制至内存中 `MPENTRY_ADDRESS` 的位置，并将每个的 AP 的 CS:IP 设置好），通过发送 STARTUP IPI 两次来启动 AP（在 `lapic_startup` 函数中说明了，这个启动方式是硬件支持的）之后 AP 就执行启动程序，等执行完后会给 BSP 发送 `CPU_STARTED` 的信号。表示结束，BSP 在 APs 的启动过程中一直处于空循环的状态。

因此 `MPENTRY_ADDRESS` 所在的内存地址是不能够被用于分配其它代码或者数据的，因此在 `page_init` 的需要进行修改。

```
void
page_init(void)
{
```

---

```

    page_free_list = NULL;
    size_t i;
    size_t nf_lb = IOPHYSMEM / PGSIZE;
    size_t nf_ub = PADDR(boot_alloc(0)) / PGSIZE;
    size_t mentry_page = MPENTRY_PADDR / PGSIZE;
    for (i = 0; i < npages; i++) {
        if (i != 0 && (i < nf_lb || i >= nf_ub) && i != mentry_page) {
            pages[i].pp_ref = 0;
            pages[i].pp_link = page_free_list;
            page_free_list = &pages[i];
        } else {
            pages[i].pp_ref = 1;
            pages[i].pp_link = NULL;
        }
    }
}

```

### 1.3 Question 1

kern/mpentry.S 的链接地址在 KERNBASE 以上，boot aps 之前将其代码拷贝至 MPENTRY\_PADDR，注意到 aps 实际运行的时候还是在实模式，而 MPBOOTPHYS 就是用来将高地址的位置进行转化至 MPENTRY\_PADDR 开始的低地址的，如果注释掉这个宏，则在使用绝对地址的寻址和跳转则会造成 Page Fault。

### 1.4 Exercise 3

这部分是对每个 CPU 进行初始化，包括虚拟内存映射的初始化，以及 trap 触发对每个 CPU 进行初始化

问题：为什么给 kern\_pgdir 即可，不用每个 cpu 的 pgdir 都赋予值？

因为每个 cpu 的 pgdir 在高位都是相同的，即内核部分除了内核栈不同外，代码和数据都是映射到相同的地址下。

```

static void
mem_init_mp(void)
{
    int cpu_id;
    for (cpu_id = 0; cpu_id < NCPU; cpu_id++) {
        boot_map_region(kern_pgdir,
                        KSTACKTOP - cpu_id * (KSTKSIZE + KSTKGAP) - KSTKSIZE,
                        KSTKSIZE,
                        PADDR(percpu_kstacks[cpu_id]),
                        PTE_W);
    }
}

```

### 1.5 Exercise 4

根据 cpu\_id 放置在对应的位置即可。

```

void
trap_init_percpu(void)
{
    // LAB 4: Your code here:
    int cpu_id = thiscpu->cpu_id;
    thiscpu->cpu_ts.ts_esp0 = KSTACKTOP - cpu_id * (KSTKSIZE + KSTKGAP);
    thiscpu->cpu_ts.ts_ss0 = GD_KD;

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

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

    lidt(&idt_pd);
}

```

```

lcch@lcch:~/OS/lab4_partA$ make qemu CPUS=4
qemu -hda obj/kern/kernel.img -serial mon:stdio -gdb tcp::26000 -D qemu.log -smp
4
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 4 CPU(s)
enabled interrupts: 1 2
SMP: CPU 1 starting
SMP: CPU 2 starting
SMP: CPU 3 starting

```

符合预期:

## 1.6 Exercise 5

根据题目要求在对应的地方放上 lock 或者 unlock 即可，连位置都有注释提示，非常简单。

## 1.7 Question 2

共享内核栈是会存在问题的，在发生中断的时候，对于现场保护的是发生在 big kernel lock 之前的，因此当共享内核栈很可能会产生参数的混乱。

## 1.8 Exercise 6

这一部分要我们实现 Round-Robin Scheduling，即类似循环链表，每次从刚执行的 env 之后选择最近的一个 RUNNABLE 的进行执行。代码如下：

```

void
sched_yield(void)
{

```

```

struct Env *idle;

// LAB 4: Your code here.
int now_env, i;
if (curenv) {                // thiscpu->cpu_env
    now_env = (ENVX(curenv->env_id) + 1) % NENV;
} else {
    now_env = 0;
}
for (i = 0; i < NENV; i++, now_env = (now_env + 1) % NENV) {
    if (envs[now_env].env_status == ENV_RUNNABLE) {
        //cprintf("I am CPU %d , I am in sched yield, I find ENV %d\n",
            thiscpu->cpu_id, now_env);
        env_run(&envs[now_env]);
    }
}
if (curenv && curenv->env_status == ENV_RUNNING) {
    env_run(curenv);
}
// sched_halt never returns
sched_halt();
}

```

```

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
[00000000] new env 00001001
[00000000] new env 00001002
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.
Back in environment 00001000, iteration 2.
Back in environment 00001001, iteration 2.
Back in environment 00001002, iteration 2.
Back in environment 00001000, iteration 3.
Back in environment 00001001, iteration 3.
Back in environment 00001002, iteration 3.
Back in environment 00001000, iteration 4.

```

符合预期:

## 1.9 Question 3-4

Question 3: 由于 Env 在 mem\_init 中分配了内存进行存储并建立了映射，而这个映射在每个 CPU 上都是一致的，因此 lcr3() 后对于 Env 的还是存在虚拟内存到物理内存的映射的。

Question 4: 保存现场是为了之后 CPU 进行继续处理这个 environment 的时候保证不会造成错误。保护现场发生在发生中断的情况下，这时候堆栈上储存了当前 environment 的信息，之后再通过 curenv->env\_tf = \*tf 来保存在 environment 中。

---

## 1.10 Challenge 2

设置一个带有优先级的调度，我实现了一个非常弱智的，每次找到可运行的优先级最高的，如果优先级相同，则优先处理刚处理完的进程号之后最近的一个进程。对于优先级的表示，我在 struct Env 中设置了一位 env\_priority 表示优先级，越大表示优先级越高。

因此调度就变为了：

```
void
sched_yield(void)
{
    struct Env *idle;

    int now_env, i;
    if (curenv) {                // thiscpu->cpu_env
        now_env = (ENVX(curenv->env_id) + 1) % NENV;
    } else {
        now_env = 0;
    }
    uint32_t max_priority = 0;
    int select_env = -1;
    for (i = 0; i < NENV; i++, now_env = (now_env + 1) % NENV) {
        if (envs[now_env].env_status == ENV_RUNNABLE && (envs[now_env].
            env_priority > max_priority || select_env == -1)) {
            //cprintf("I am CPU %d , I am in sched yield, I find ENV %d\n",
                thiscpu->cpu_id, now_env);
            select_env = now_env;
            max_priority = envs[now_env].env_priority;
        }
    }
    if (select_env >= 0 && (!curenv || curenv->env_status != ENV_RUNNING ||
        max_priority >= curenv->env_priority)) {
        env_run(&envs[select_env]);
    }

    if (curenv && curenv->env_status == ENV_RUNNING) {
        env_run(curenv);
    }

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

为了测试，我首先增加了一个更改 priority 的系统调用：

```
static int
sys_set_priority(envid_t envid, uint32_t new_priority)
{
    struct Env *env;
    int r = envid2env(envid, &env, 1);
    if (r < 0) return -E_BAD_ENV;

    env->env_priority = new_priority;
    return 0;
}
```

---

我在用户 `fork()` 库函数增加了一个 `static` 变量 `num`，使得其递减，并设置刚创建的子进程的 `priority` 为 `num`:

```
envid_t
fork(void)
{
    static int num = 0x100;

    // LAB 4: Your code here.
    set_pgfault_handler(pgfault);
    int childpid = sys_exofork();
    if (childpid < 0) {
        panic("fork sys_exofork error : %e\n", childpid);
    }
    int r;

    if (childpid == 0) {
        .....
    } else {
        .....

        // LAB 4 Challenge , set childenv's priority
        num--;
        if (num == 0) num = 0x100;
        r = sys_set_priority(childpid, num);
        if (r < 0) panic("fork, set priority error\n");

        // mark the child as runnable and return
        r = sys_env_set_status(childpid, ENV_RUNNABLE);
        if (r < 0) panic("fork, set child process to ENV_RUNNABLE error : %e\n",
            r);
        return childpid;
    }
}
```

这样就可以进行测试了，我把 `hello.c` 更改为了下面的代码，根据我们之前的铺垫，预期就是执行完子进程 0，才会执行子进程 1，才会执行子进程 2，才会执行子进程 3。

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

void
umain(int argc, char **argv)
{
    cprintf("hello, world\n");
    cprintf("i am environment %08x\n", thisenv->env_id);

    int i, child, j;
    for (i = 0; i != 3; i++) {
        child = fork();
        if (child == 0) {
            for (j = 0; j != 10; j++) {
                cprintf("hello world from : %d\n", i);
                sys_yield();
            }
            break;
        }
    }
}
```



```

    }
}

```

```

[00000000] new env 00001000
hello, world
i am environment 00001000
[00001000] new env 00001001
[00001000] new env 00001002
hello world from : 0
hello world from : 0;env->env_id);
hello world from : 0
hello world from : 0
hello world from : 0
hello world from : 0
hello world from : 0
hello world from : 0
hello world from : 0
hello world from : 0
hello world from : 0
[00001001] exiting gracefully
[00001001] free env 00001001
[00001000] new env 00002001
hello world from : 1
hello world from : 1
hello world from : 1
hello world from : 1
hello world from : 1
hello world from : 1
hello world from : 1
hello world from : 1
hello world from : 1
hello world from : 1
[00001002] exiting gracefully
[00001002] free env 00001002
[00001000] exiting gracefully
[00001000] free env 00001000
hello world from : 2
hello world from : 2
hello world from : 2
hello world from : 2
hello world from : 2
hello world from : 2
hello world from : 2
hello world from : 2
hello world from : 2
hello world from : 2
[00002001] exiting gracefully
[00002001] free env 00002001

```

和预期一致!

## 1.11 Question 7

### 1.11.1 sys\_exofork

这个部分是生成一个新的 environment，使得其寄存器的值与当前的 environment 的一样，对于新的 environment 返回值为 0 (%eax=0)。对于该函数返回新的 environment 的 pid 号。

```

static env_id_t
sys_exofork(void)
{
    struct Env * new_env;

```

---

```

int r = env_alloc(&new_env, curenv->env_id);
if (r < 0) return r;

new_env->env_status = ENV_NOT_RUNNABLE;
memcpy((void *)&new_env->env_tf, (void *)&curenv->env_tf, sizeof(struct
    Trapframe));

// for children environment, return 0
new_env->env_tf.tf_regs.reg_eax = 0;

return new_env->env_id;
}

```

### 1.11.2 sys\_env\_set\_status

这个函数是设置 environment 的 status。

```

static int
sys_env_set_status(env_id_t env_id, int status)
{
    if (status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE)
        return -E_INVALID;

    struct Env * env;
    int r = env_id2env(env_id, &env, 1);
    if (r < 0) return r;
    env->env_status = status;
    return 0;
}

```

### 1.11.3 sys\_page\_alloc

这个函数是为 environment 创建虚拟地址 va 的映射页，需要进行一般性的判断，例如是否有权限以及是否对齐。最后需要注意如何无法建立映射，则新分配的页需要释放掉。

```

static int
sys_page_alloc(env_id_t env_id, void *va, int perm)
{
    struct Env * env;
    int r = env_id2env(env_id, &env, 1);
    if (r < 0) return -E_BAD_ENV;

    if ((uint32_t)va >= UTOP || ROUNDUP(va, PGSIZE) != va) return -E_INVALID;

    if (!(perm & PTE_U) && (perm & PTE_P) && (perm & (~PTE_SYSCALL)) == 0)
        return -E_INVALID;

    struct PageInfo * pg = page_alloc(ALLOC_ZERO);
    if (pg == NULL) return -E_NO_MEM;
    if (page_insert(env->env_pgdir, pg, va, perm) < 0) {
        // page_insert fails, should free the page you allocated!
        page_free(pg);
    }
}

```

```

        return -E_NO_MEM;
    }
    return 0;
}

```

#### 1.11.4 sys\_page\_map

按照给定的进程，和虚拟地址，拷贝其映射至另一个进程的给定的虚拟地址上，按照注释一条一条做即可。

```

static int
sys_page_map(envid_t srcenvid, void *srcva,
             envid_t dstenvid, void *dstva, int perm)
{
    struct Env * dstenv, * srcenv;
    int r = envid2env(dstenvid, &dstenv, 1);
    if (r < 0) return -E_BAD_ENV;
    r = envid2env(srcenvid, &srcenv, 1);
    if (r < 0) return -E_BAD_ENV;

    if ((uint32_t)srcva >= UTOP || ROUNDUP(srcva, PGSIZE) != srcva) return -
        E_INVAL;
    if ((uint32_t)dstva >= UTOP || ROUNDUP(dstva, PGSIZE) != dstva) return -
        E_INVAL;

    // struct PageInfo * page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
    struct PageInfo * pg;
    pte_t * pte;
    pg = page_lookup(srcenv->env_pgdir, srcva, &pte);
    if (pg == NULL) return -E_INVAL;

    if (!(perm & PTE_U) && (perm & PTE_P) && (perm & (~PTE_SYSCALL)) == 0))
        return -E_INVAL;

    if ((perm & PTE_W) && ((*pte) & PTE_W) == 0) return -E_INVAL;

    // int page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm)
    if (page_insert(dstenv->env_pgdir, pg, dstva, perm) < 0) return -E_NO_MEM;

    return 0;
}

```

#### 1.11.5 sys\_page\_unmap

这个注释是解除页面中某一段的映射。

```

static int
sys_page_unmap(envid_t envid, void *va)
{
    struct Env * env;
    int r = envid2env(envid, &env, 1);
    if (r < 0) return -E_BAD_ENV;
}

```

---

```

    if ((uint32_t)va >= UTOP || ROUNDUP(va, PGSIZE) != va) return -E_INVAL;

    // void page_remove(pde_t *pgdir, void *va)
    page_remove(env->env_pgdir, va);

    return 0;
}

```

注意到目前写的调度都是执行完一个 environment 之后就结束，剩余最后一个作为 monitor，其余的 CPU 均 HLT 住，但是注意不能把所有 CPU 都 HLT，这样会出现中断 13。当我尝试在 init.c 中设置 environment 的数量大于 CPU 核的数量时，就会造成中断 13，因为所有的 CPU 都 HLT 了。我在这里调试了很久，花了很长时间，最后才发现竟然是没考虑清楚。

## 2 Part B: Copy-on-Write Fork

fork() 指令会根据当前进程生成一个新的与原进程一样的 (除了 pid 号不一样) 的进程，他们拥有独立的内存空间，独立的寄存器，独立的用户栈等等。因此我们对于 fork() 的执行需将内存中完全复制一份，这个代价是非常高的。我们可以采用 Copy-on-Write 的技术，使用 Copy-on-Write 的原因在于，fork() 中有的内存是多个进程是只读的，那么我们完全可以将这部分的内存不进行复制；其次并不是所有的数据都需要更改，所以我们可以共同使用同一个物理内存，当进行写操作的时候，再分为独立的内存。其次还有一个原因是许多情况下 fork() 之后会执行 exec()，因此 fork() 采用 Copy-on-Write 是一个比较很好的减少不必要操作，节省内存的一个方法。

如何得知某一块内存是被多个进程共用，如何得知这样的内存存在被写的时候需要进行复制。对于前一个问题，PTE 表项中有由一个专门的标志为 PTE\_COW，表示这个页是否使用了 Copy-on-Write。如何在写的时候进行复制，这里采用触发 Page Fault 的方法。JOS 在内核中注册了一个 Page fault 的处理程序，允许用户进程将自己的页错误函数注册到进程结构中。这样当发生页错误的时候可以在 user-level 下实现对 page level 的处理，增大了灵活性，而且一定程度上节省了 page fault 霸占内核的时间。

为了实现 user-level 的 page fault，那么就不能使用内核栈用于保存信息，因此就出现了 Exception Stack，是 user-level page fault handler 运行时用的栈，用于保存信息。

### 2.1 Exercise 8

如果用户需要使用自己的 Page Fault Handler，则事先需要向内核注册 handler 所在的位置，保存在用户进程的 env\_pagfault\_upcall 变量中，此处就是注册 user-level page fault handler。

```

static int
sys_env_set_pgfault_upcall(envid_t envid, void *func)
{
    struct Env *env;
    int r = envid2env(envid, &env, 1);
}

```

```

    if (r < 0) return r;

    env->env_pgfault_upcall = func;
    return 0;
}

```

当产生 Page Fault 的时候，如果是内核模式下产生的，则产生 panic，一定是系统出现了 bug。如果是用户模式下产生的 page fault 且存在 user-level page fault 的 handler 的时候，则需要保存现场信息至 user exception stack。并设置好当前 env 的 eip 为 user-level page fault handler 的入口地址，并将栈指向当前 user exception stack 的位置。然后返回用户态执行 user-level page fault handler。特别要注意在 user-level page fault handler 中，是在用户模式下执行的，并且所使用的栈是在 user exception stack 上。当 user-level page fault handler 执行完成后，会切换会用户进程和用户运行栈下继续执行。

## 2.2 Exercise 9

此处就为 kernel 在接受到 page\_fault 异常的时候，进行的处理。如果是在用户模式且存在注册的 page fault handler，则需要存栈。如果是 page fault 嵌套，则需要在异常栈开始的地方多压一个空的 32-bit word。为什么要这样做，后面的 exercise 会进行解释。

```

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

    // Read processor's CR2 register to find the faulting address
    fault_va = rcr2();

    // Handle kernel-mode page faults.

    // LAB 3: Your code here.
    if (tf->tf_cs == GD_KT)
        panic("page_fault_handler : page fault in kernel\n");

    if (curenv->env_pgfault_upcall != NULL) {
        // exist env's page fault upcall

        struct UTrapframe * ut;
        if (tf->tf_esp >= UXSTACKTOP - PGSIZE && tf->tf_esp <= UXSTACKTOP - 1) {
            // already in user exception stack, should first push an empty
            // 32-bit word
            ut = (struct UTrapframe *)((void *)tf->tf_esp - sizeof(struct
                UTrapframe) - 4);
            user_mem_assert(curenv, (void *)ut, sizeof(struct UTrapframe) + 4,
                PTE_U | PTE_W);
        } else {
            // it's the first time in user exception stack
            ut = (struct UTrapframe *) (UXSTACKTOP - sizeof(struct UTrapframe));
            user_mem_assert(curenv, (void *)ut, sizeof(struct UTrapframe),
                PTE_U | PTE_W);
        }
    }
}

```

```

    }

    ut->utf_esp = tf->tf_esp;
    ut->utf_eflags = tf->tf_eflags;
    ut->utf_eip = tf->tf_eip;
    ut->utf_regs = tf->tf_regs;
    ut->utf_err = tf->tf_err;
    ut->utf_fault_va = fault_va;

    curenv->env_tf.tf_eip = (uint32_t)curenv->env_pgfault_upcall;
    curenv->env_tf.tf_esp = (uint32_t)ut;
    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);
}

```

## 2.3 Exercise 10

`_pgfault_upcall` 实际上实现到真正 handler 的跳转，待 handler 处理完后进行环境的恢复。对于栈的恢复是比较简单的，因为 `old-esp` 和 `old-ebp` 已经存在了栈中，而对于 `eip` 的恢复就需要使用 `ret` 的机制，即弹出栈顶元素作为 `eip`，跳转到 `eip` 的位置。因此我们在 `ret` 之前只需要把 `eip` 存放在栈顶的位置即可，应该放在哪里？还记得如果产生了 `page fault` 的嵌套，则在栈中会多存放一个空的 4 字节，这就是用于返回 `eip` 的存放。对于第一层的 `page fault`，则可以把 `eip` 放在原来的用户运行栈的下一个 4 字节位置，并将 `%esp` 指向 `eip` 的位置，使用 `ret` 就可以实现跳转了。

对于 4 字节是否是必须的？为什么不能在最后再将返回的 `%eip` 弹入栈中？这是因为你是无法不污染寄存器的情况下实现这样的操作的，即为了在最后将 `%eip` 弹入栈中，你必须需要寄存器来存储。而这样对于保存其值的寄存器便无法不被污染。

```

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

// set trap-time %eip
movl 0x28(%esp), %ebx
movl %ebx, (%eax)

// Restore the trap-time registers. After you do this, you
// can no longer modify any general-purpose registers.
addl $0x08, %esp // ignore err_code and fault_va
popal           // restore registers

// Restore eflags from the stack. After you do this, you can
// no longer use arithmetic operations or anything else that

```

---

```

// modifies eflags.
addl $0x04, %esp // ignore eip
popfl           // modify eflags

// Switch back to the adjusted trap-time stack.
popl %esp

// Return to re-execute the instruction that faulted.
ret

```

## 2.4 Exercise 11

这里实际上是提供给用户的库函数，需要做的有申请分配 exception stack 的空间，以及系统调用设置 user-level page fault 函数。注意给系统的实际上是 `_pgfault_upcall`，这是 user-level page fault 的入口程序，在实际的 handler 存放在 `_pgfault_handler` 中，这样从内核态到用户态 page fault 的转换，实际上是到了 `_pgfault_upcall`，再在其中实现到真正 handler: `_pgfault_handler` 的跳转。

```

void
set_pgfault_handler(void (*handler)(struct UTrapframe *utf))
{
    int r;
    if (_pgfault_handler == 0) {
        // First time through!
        // LAB 4: Your code here.
        //int sys_page_alloc(envid_t envid, void *va, int perm)
        r = sys_page_alloc(0, (void*)(UXSTACKTOP - PGSIZE), PTE_U | PTE_W |
            PTE_P);
        if (r < 0) {
            panic("sys_page_alloc error : %e\n", r);
        }
        // how to know envid, put 0, envid2env will help us to get curenv in
        // syscall
        r = sys_env_set_pgfault_upcall(0, _pgfault_upcall);
        if (r < 0) {
            panic("sys_env_set_pgfault_upcall error : %e\n", r);
        }
    }
    // Save handler pointer for assembly to call.
    _pgfault_handler = handler;
}

```

## 2.5 Exercise 12

这个部分就是要真正实现一个带 Copy-on-Write 的提供给用户的 fork 函数了。为了实现 Copy-on-Write 需要先注册一个 `pgfault_handler`，这样当某一个进程对某一块共享区域进行写入的时候，就绪要进行内存的复制使得这块内存不共享。然后进行 `sys_exofork()`，由于此时子进程还处于 NOT\_RUNNABLE 的状态，父进程需要对子进程构建内存的映射（这里只需要复制 0 ~ UTOP 中存在的并且是用户的内存条目，对于内核态的数据和代码都是共享的，已经存

---

在于子进程的页表中，无需拷贝)，以及创建分配一块 exception stack 的内存，并设置子进程的 pgfault\_handler。当这些都完成了，说明子进程已经可以开始运行了，就可以标记子进程的状态为 RUNNABLE 了。

特别注意当子进程进行执行的时候，需要更新子进程的 thisenv 号。

### 2.5.1 fork

```
envid_t
fork(void)
{
    set_pgfault_handler(pgfault);
    int childpid = sys_exofork();
    if (childpid < 0) {
        panic("fork sys_exofork error : %e\n", childpid);
    }
    int r;

    if (childpid == 0) {
        // child process
        // Remember to fix "thisenv" in the child process. ???
        thisenv = &envs[ENVX(sys_getenvid())];
        // cprintf("fork child ok\n");
        return 0;
    } else {
        // map page to new environment
        // kernel page is already in new environment
        uint32_t i;
        for (i = 0; i != UTOP; i += PGSIZE)
            if ((uvpd[PDX(i)] & PTE_P) && (uvpt[i / PGSIZE] & PTE_P) && (uvpt[i / PGSIZE] & PTE_U)) {
                duppage(childpid, i / PGSIZE);
            }

        // allocate exception stack
        r = sys_page_alloc(childpid, (void *) (UXSTACKTOP - PGSIZE), PTE_U | PTE_W | PTE_P);
        if (r < 0) panic("fork, sys_page_alloc user exception stack error : %e\n", r);

        // set user environment user page fault handler
        extern void _pgfault_upcall(void);
        r = sys_env_set_pgfault_upcall(childpid, _pgfault_upcall);
        if (r < 0) panic("fork, set pgfault upcall fail : %e\n", r);

        // mark the child as runnable and return
        r = sys_env_set_status(childpid, ENV_RUNNABLE);
        if (r < 0) panic("fork, set child process to ENV_RUNNABLE error : %e\n", r);

        // cprintf("fork father ok!");
        return childpid;
    }
}
```



---

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

### 2.5.2 duppage

这部分就是实现页表的复制，需要区分读和写即可。如果是读，则需要更改子进程的同时自己页表的标志位也需要更改。

```
static int
duppage(envid_t envid, unsigned pn)
{
    // do not dup exception stack
    if (pn * PGSIZE == UXSTACKTOP - PGSIZE) return 0;

    int r;
    void * addr = (void *) (pn * PGSIZE);
    if ((uvpt[pn] & PTE_W) || (uvpt[pn] & PTE_COW)) {
        // cow
        r = sys_page_map(0, addr, envid, addr, PTE_COW | PTE_P | PTE_U);
        if (r < 0) panic("duppage sys_page_map error : %e\n", r);

        r = sys_page_map(0, addr, 0, addr, PTE_COW | PTE_P | PTE_U);
        if (r < 0) panic("duppage sys_page_map error : %e\n", r);
    } else {
        // read only
        r = sys_page_map(0, addr, envid, addr, PTE_P | PTE_U);
        if (r < 0) panic("duppage sys_page_map error : %e\n", r);
    }

    return 0;
}
```

### 2.5.3 pgfault

这部分就是真正处理 Cope-On-Write 的地方，就是将共享的内存进行复制即可。这里我出现了一个低级错误，对于 utf->utf\_fault\_va 我想当然地以为是对齐 PGSIZE，实际上并不是，这里花了我挺长时间进行调试的。

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

    if ((err & FEC_WR) == 0)
        panic("pgfault, the fault is not a write\n");

    uint32_t uaddr = (uint32_t) addr;
    if ((uvpd[PDX(addr)] & PTE_P) == 0 || (uvpt[uaddr / PGSIZE] & PTE_COW) == 0)
    {
```

---

```
        panic("pgfault, not a copy-on-write page\n");
    }

    // static int sys_page_alloc(envid_t envid, void *va, int perm)
    r = sys_page_alloc(0, (void *)PFTEMP, PTE_W | PTE_U | PTE_P);
    if (r < 0) panic("pgfault, sys_page_alloc error : %e\n", r);

    // Oh my god, I forget this at the first, it waste me a lot of time to debug
    !!!
    addr = ROUNDDOWN(addr, PGSIZE);

    memcpy(PFTEMP, addr, PGSIZE);

    r = sys_page_map(0, PFTEMP, 0, addr, PTE_W | PTE_U | PTE_P);
    if (r < 0) panic("pgfault, sys_page_map error : %e\n", r);

    return;
}
```

---

```

SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2
[00000000] new env 00001000
1000: I am ''
[00001000] new env 00001001
[00001000] new env 00001002
1001: I am '0'
[00001001] new env 00001003
[00001001] new env 00001004
1003: I am '00'
[00001003] new env 00001005
[00001003] new env 00001006
1005: I am '000'
[00001005] exiting gracefully
[00001005] free env 00001005
[00001000] exiting gracefully
[00001000] free env 00001000
[00001001] exiting gracefully
[00001001] free env 00001001
1002: I am '1'
[00001002] new env 00002001
[00001003] exiting gracefully
[00001003] free env 00001003
[00001002] new env 00002003
1004: I am '01'
[00001004] new env 00002000
1006: I am '001'
[00001006] exiting gracefully
[00001006] free env 00001006
2000: I am '010'
[00002000] exiting gracefully
[00002000] free env 00002000
2001: I am '10'
[00002001] new env 00003000
[00002001] new env 00002006
[00001002] exiting gracefully
[00001002] free env 00001002
3000: I am '100'
[00003000] exiting gracefully
[00003000] free env 00003000
[00002001] exiting gracefully
[00002001] free env 00002001
2003: I am '11'
[00002003] new env 00003001
[00001004] new env 00004000
[00001004] exiting gracefully

```

运行 lib/fork.c, 效果和给定的一致。这样 PartB 就结束了。

## 3 Part C: IPC

### 3.1 Exercise 13

这部分是设置外部中断，注意要修改 eflags 寄存器的 FL\_IF 位，来接受外部中断。这部分和 Lab3 几乎一样。

```

// in trapentry.S add :
TRAPHANDLER_NOEC(vec32, IRQ_OFFSET + IRQ_TIMER)
TRAPHANDLER_NOEC(vec33, IRQ_OFFSET + IRQ_KBD)

```

---

```

    TRAPHANDLER_NOEC(vec36, IRQ_OFFSET + IRQ_SERIAL)
    TRAPHANDLER_NOEC(vec39, IRQ_OFFSET + IRQ_SPURIOUS)
    TRAPHANDLER_NOEC(vec46, IRQ_OFFSET + IRQ_IDE)
    TRAPHANDLER_NOEC(vec51, IRQ_OFFSET + IRQ_ERROR)

// in trap_init add :
    SETGATE(idt[IRQ_OFFSET + IRQ_TIMER], 0, GD_KT, vec32, 0);
    SETGATE(idt[IRQ_OFFSET + IRQ_KBD], 0, GD_KT, vec33, 0);
    SETGATE(idt[IRQ_OFFSET + IRQ_SERIAL], 0, GD_KT, vec36, 0);
    SETGATE(idt[IRQ_OFFSET + IRQ_SPURIOUS], 0, GD_KT, vec39, 0);
    SETGATE(idt[IRQ_OFFSET + IRQ_IDE], 0, GD_KT, vec46, 0);
    SETGATE(idt[IRQ_OFFSET + IRQ_ERROR], 0, GD_KT, vec51, 0);

// in env_alloc add :
    e->env_tf.tf_eflags |= FL_IF;

```

### 3.2 Exercise 14

这部分是处理时钟中断，实现 Time-Sharing。lab 已经帮我实现好细节了，我们只需要在 trap.c 中遇到时钟中断调用相应函数即可：

```

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

```

### 3.3 Exercise 15

这部分我们提供两个系统调用来实现进程间的通信，程序注释说的非常详细，一步一步做即可。

### 3.4 sys\_ipc\_recv

```

static int
sys_ipc_recv(void *dstva)
{
    // LAB 4: Your code here.
    if (((uint32_t)dstva < UTOP) && ROUNDUP(dstva, PGSIZE) != dstva) return -
        E_INVAL;
    curenv->env_ipc_recving = true;           // Env is blocked receiving
    curenv->env_ipc_dstva = dstva;             // VA at which to map received page
    curenv->env_ipc_from = 0;                  // set from to 0
    curenv->env_status = ENV_NOT_RUNNABLE;     // mark it not runnable
    // cprintf("I am receiving!!!\n");
    sched_yield();                             // give up the CPU

    return 0;
}

```

---

### 3.5 sys\_try\_send

中间我很不小心填错发消息的 `envid`，导致陷入了无尽的调试之中，花了将近 2 个小时才找到错误，太伤心了。

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

    // 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.
    // So set the checkperm flag to 0
    struct Env * env;
    int r = envid2env(envid, &env, 0);
    // -E_BAD_ENV if environment envid doesn't currently exist.
    if (r < 0) return -E_BAD_ENV;

    // -E_IPC_NOT_RECV if envid is not currently blocked in sys_ipc_recv,
    // or another environment managed to send first.
    if (env->env_ipc_recving == false || env->env_ipc_from != 0) {
        return -E_IPC_NOT_RECV;
    }

    // -E_INVALID if srcva < UTOP but srcva is not page-aligned.
    if ((uint32_t)srcva < UTOP && ROUNDUP(srcva, PGSIZE) != srcva)
        return -E_INVALID;

    // -E_INVALID if srcva < UTOP and perm is inappropriate
    if ((uint32_t)srcva < UTOP && (!(perm & PTE_U) && (perm & PTE_P) && (perm &
        (~PTE_SYSCALL)) == 0)))
        return -E_INVALID;

    // -E_INVALID if srcva < UTOP but srcva is not mapped in the caller's address
    // space
    if ((uint32_t)srcva < UTOP) {
        pte_t * pte;
        struct PageInfo * pg = page_lookup(curenv->env_pgdir, srcva, &pte);
        if (pg == NULL) return -E_INVALID;

        // -E_INVALID if (perm & PTE_W), but srcva is read-only in the
        // current environment's address space.
        if ((perm & PTE_W) && (*pte & PTE_W) == 0)
            return -E_INVALID;

        // -E_NO_MEM if there's not enough memory to map srcva in envid's
        // address space.
        if (env->env_ipc_dstva != NULL) {
            r = page_insert(env->env_pgdir, pg, env->env_ipc_dstva, perm);
            if (r < 0) return -E_NO_MEM;
            env->env_ipc_perm = perm;
        } else env->env_ipc_perm = 0;
    }

    env->env_ipc_recving = false;
}
```

---

```

    // ... I mistake write env->env_ipc_from = env_id in the first
    // ... Debug a lot of time...
    env->env_ipc_from = curenv->env_id;
    env->env_ipc_value = value;
    env->env_tf.tf_regs.reg_eax = 0;
    env->env_status = ENV_RUNNABLE;

    return 0;
}

```

### 3.6 ipc\_recv

```

int32_t
ipc_recv(envid_t *from_env_store, void *pg, int *perm_store)
{
    // LAB 4: Your code here.
    int r;
    if (pg != NULL) {
        r = sys_ipc_recv(pg);
    } else {
        r = sys_ipc_recv((void *)UTOP);
    }

    if (r == 0) {
        if (from_env_store != NULL) *from_env_store = thisenv->env_ipc_from;
        if (perm_store != NULL) *perm_store = thisenv->env_ipc_perm;
        // cprintf("Receive %d\n", thisenv->env_ipc_value);
        return thisenv->env_ipc_value;
    } else {
        // fails;
        if (from_env_store != NULL) *from_env_store = 0;
        if (perm_store != NULL) *perm_store = 0;
        return r;
    }
}

```

### 3.7 ipc\_send

```

void
ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
{
    int r;
    if (pg == NULL) pg = (void *)UTOP;

    while ((r = sys_ipc_try_send(to_env, val, pg, perm)) != 0) {
        if (r == -E_IPC_NOT_RECV) {
            // cprintf("Try Again and Again...\n");
            sys_yield();
        } else {
            panic("ipc_send error %e\n", r);
        }
    }
}

```

```

    }
}
return;
}

```

```

SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2
[00000000] new env 00001000
[00001000] new env 00001001
send 0 from 1000 to 1001
1001 got 0 from 1000
1000 got 1 from 1001
1001 got 2 from 1000
1000 got 3 from 1001
1001 got 4 from 1000
1000 got 5 from 1001
1001 got 6 from 1000
1000 got 7 from 1001
1001 got 8 from 1000
1000 got 9 from 1001
[00001000] exiting gracefully
[00001000] free env 00001000
1001 got 10 from 1000
[00001001] exiting gracefully
[00001001] free env 00001001
No runnable environments in the system!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> _

```

pingpong 执行情况

```

CPU 0: 8009 [000013f0] new env 000013f1
CPU 0: 8011 [000013f1] new env 000013f2
CPU 0: 8017 [000013f2] new env 000013f3
CPU 0: 8039 [000013f3] new env 000013f4
CPU 0: 8053 [000013f4] new env 000013f5
CPU 0: 8059 [000013f5] new env 000013f6
CPU 0: 8069 [000013f6] new env 000013f7
CPU 0: 8081 [000013f7] new env 000013f8
CPU 0: 8087 [000013f8] new env 000013f9
CPU 0: 8089 [000013f9] new env 000013fa
CPU 0: 8093 [000013fa] new env 000013fb
CPU 0: 8101 [000013fb] new env 000013fc
CPU 0: 8111 [000013fc] new env 000013fd
CPU 0: 8117 [000013fd] new env 000013fe
CPU 0: 8123 [000013fe] new env 000013ff
CPU 0: 8147 [000013ff] user panic in <unknown> at lib/fork.c:123: fork sys_exofork
rk error : out of environments

Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf036af80 from CPU 0
edi 0xffffffffb
esi 0xeebdf54
ebp 0xeebdf40

```

primes 执行情况:

---

```
make[1]:正在离开目录 `/home/lcch/OS/jos/lab4'
dumbfork: OK (1.8s)
Part A score: 5/5

faultread: OK (0.8s)
faultwrite: OK (1.0s)
faultdie: OK (1.0s)
faultregs: OK (1.9s)
faultalloc: OK (1.2s)
faultallocbad: OK (1.7s)
faultnostack: OK (1.2s)
faultbadhandler: OK (1.9s)
faultevilhandler: OK (1.2s)
forktree: OK (1.8s)
Part B score: 50/50

spin: OK (2.0s)
stresssched: OK (1.5s)
pingpong: OK (1.5s)
primes: OK (4.7s)
Part C score: 20/20

Score: 75/75
lcch@lcch:~/OS/jos/lab4$ _
```

make grade 纪念:

## 4 Summary

### 4.1 Endless Debug

通过这个 Lab 我深刻理解了操作系统和我平时一般写的程序的区别，在写操作系统，是站在系统的角度，要综合考虑系统的宏观和微观，首先要有一个宏观的框架，在这个框架下搭建小的细节。而且操作系统之间存在用户模式和内核模式，在这两块切换处理非常绕，以及操作系统又要和软件打交道又要和硬件打交道，这些都是非常考验对操作系统宏观的理解的。虽然 JOS 是个小型的操作系统，但是写起来非常费劲，有时候写一个程序还需要考虑和别的程序之间的调用情况，还要防止内核的错误以及用户的不正确行为。在 JOS 编写中，遇到了 bug 真的是非常头疼，特别是当浏览程序都发先没有什么问题的时候真是着实让人抓狂。

在这个 Lab 中我花了非常非常多的时间在 Debug 中，其中很大一部分原因是自己的粗心造成的，还有就是对问题的不够理解。主要犯的 bug 有在 Copy-on-Write 的地方，我以为 `utf->utf_fault_va` 是对齐 PGSIZE，实际上不是，我调试了整整一个白天。以及在 ipc 的地方也因为代码错误导致了陷入 debug 中无法自拔。太伤心了。我在 lab4 中的花的大部分时间都在不断的 debug 上面。

通过这个 Lab 我发现虚拟机真的是个好东西，我调试采用了 `printf` 的调试方案，崩一次换一个地方 `printf`。如果是真机上面的写的操作系统，我觉得我的硬件早就被我玩坏了。