# Operating System MIT 6.828 JOS Lab4 Report

Computer Science
ChenHao(1100012776)

2013 年 10 月 21 日

## 目录

# 1 Part A: Multiprocessor Support and Cooperative Multitasking

## 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的函数，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 = MMIOBASE;

    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 mpentry_page = MPENTRY_PADDR / PGSIZE;
    for (i = 0; i < npages; i++) {
        if (i != 0 && (i < nf_lb || i >= nf_ub) && i != mpentry_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，而 AP 在是在实模式下无法使用页寻址，因此我们需要手动地计算出其所在的物理地址。如果注释了这句，则会产生缺页异常。

## 1.4 Exercise 3

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

```
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);
}
```

## 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();
}
```

## 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，

## 1.10 Question 7

### 1.10.1 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 envid_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;
}
```

### 1.10.2 sys_env_set_status

这个是设置 environment 的 status，按照注释做即可。

```
static int
sys_env_set_status(envid_t envid, int status)
{
    if (status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE)
        return -E_INVAL;

    struct Env * env;
    int r = envid2env(envid, &env, 1);
    if (r < 0) return r;
    env->env_status = status;
    return 0;
}
```

### 1.10.3 sys_page_alloc

这个函数是为 environment 创建虚拟地址 va 的映射页，按照注释一条一条做即可。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(envid_t envid, void *va, int perm)
{
    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;

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

    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;
}
```

### 1.10.4 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;
}
```

### 1.10.5 sys_page_unmap

按照注释一条一条做即可。

```
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 核的数量的时候，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