Operating System MIT 6.828 JOS Lab3 Report

Computer Science
ChenHao(1100012776)

2013 年 10 月 5 日

目录

1 Part A: User Environments and Exception Handling

lsof -i:xxxx (xxxx 是被占用的端口, 得到占用端口的进程的 PID)

1.1 Exercise 1

分配物理内存和创建虚拟内存映射给 envc，类似 Lab2 即可。

```
    envs = (struct Env *) boot_alloc(NENV * sizeof(struct Env));

// ... ...

    boot_map_region(kern_pgdir,
                    UENVS,
                    ROUNDUP(NENV * sizeof(struct Env), PGSIZE),
                    PADDR(envs),
                    PTE_U);
```

1.2 Exercise 2

pmap 只对内核进行了内存管理，而对于每个进程，都用有一个独立的内存空间，并且每个进程看起来都拥有整个内存空间，因此我们需要对进程也进行虚拟内存的管理，以及管理如何创建进程和进程的切换的问题。

1.2.1 env_init

env_init 类似 page_init，用来初始化 NENV 个进程管理结构，并且用单向链表来组织空闲的 Env。其中要求 env_free_list 初始指向 &envs[0]。似乎这个的原因是在 init.c 中其会执行 envs[0]。

```
void
env_init(void)
{
    // Set up envs array
    // LAB 3: Your code here.
    uint32_t i;
    env_free_list = envs;
    for (i = 0; i < NENV; i++) {
        envs[i].env_id = 0;
        envs[i].env_status = ENV_FREE;
        if (i + 1 != NENV)
            envs[i].env_link = envs + (i + 1);
        else
            envs[i].env_link = NULL;
    }

    // Per-CPU part of the initialization
    env_init_percpu();
}
```

1.2.2 env_setup_vm

env_setup_vm 分配进程独立的 Page Directory，即创建该进程的页目录。对于高于 UTOP 的虚拟地址 PDE 应与 Kernel 的页目录，对于低于 UTOP 的位置需要清 0，这部分就是真正用户进程使用的页目录条目。

为什么进程的页目录高于 UTOP 的虚拟地址的映射和 Kernel 的页目录一致？

我觉得原因在于在内核管理进程的时候，在需用对进程使用的内存进行访问或者使用的时候就需要改用进程的 Page Directory，但是同时还需要使用内核的代码或数据，因此保持一直可以保证这一点，不会造成错误和不必要的麻烦。而由于高于 UTOP 的虚拟地址的权限都是 kernel 权限的，因此在用户态的情况可以防止用户进行访问和修改，而且对于 UTOP 以上的内存对于用户进程是不允许访问的，这部分对于用户进程来说是不会使用的。

```
static int
env_setup_vm(struct Env *e)
{
    int i;
    struct PageInfo *p = NULL;

    // Allocate a page for the page directory
    if (!(p = page_alloc(ALLOC_ZERO)))
        return -E_NO_MEM;

    p->pp_ref++;
    e->env_pgdir = (pde_t *)page2kva(p);
    memcpy(e->env_pgdir, kern_pgdir, PGSIZE);
    memset(e->env_pgdir, 0, PDX(UTOP) * sizeof(pde_t));

    // UVPT maps the env's own page table read-only.
    // Permissions: kernel R, user R
    e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_P | PTE_U;

    return 0;
}
```

1.2.3 region_alloc

region_alloc 用于为进程分配物理内存，因此应该使用对应进程的页目录和页表。

```
static void
region_alloc(struct Env *e, void *va, size_t len)
{
    uint32_t addr = (uint32_t)ROUNDDOWN(va, PGSIZE);
    uint32_t end  = (uint32_t)ROUNDUP(va + len, PGSIZE);
    struct PageInfo *pg;
    // cprintf("region_alloc: %u %u\n", addr, end);
    for ( ; addr != end; addr += PGSIZE) {
        pg = page_alloc(1);
        if (pg == NULL) {
            panic("region_alloc : can't alloc page\n");
        } else {
```

```
            if (page_insert(e->env_pgdir, pg, (void *)addr, PTE_U | PTE_W) != 0)
                 {
                panic("region_alloc : page_insert fail\n");
            }
        }
    }
    return;
}
```

1.2.4 load_icode

load_icode 将目标文件放入内存中，存放的虚拟内存的位置由目标文件指定。这个函数有两个需要注意的地方，第一个是首先使用 region_alloc 分配对应虚拟地址的内存，而这个映射仅在该进程的页表中存在，在内核中是不存在的，因此在 memcpy 和 memset 的时候需要使用的该进程的页目录，而不应该使用内核的页目录。这个地方非常阴险，我一开始就掉进了这个陷阱中。

第二个需要注意的地方就是需要将 elf->e_entry 即目标文件的入口放入进程环境的 eip 中。

```
static void
load_icode(struct Env *e, uint8_t *binary, size_t size)
{
    struct Elf * elf = (struct Elf *)binary;
    if (elf->e_magic != ELF_MAGIC) {
        panic("error elf magic number\n");
    }
    struct Proghdr *ph, *eph;
    ph = (struct Proghdr *) ((uint8_t *) elf + elf->e_phoff);
    eph = ph + elf->e_phnum;

    lcr3(PADDR(e->env_pgdir));
    for (; ph < eph; ph++) {
        if (ph->p_type == ELF_PROG_LOAD) {
            region_alloc(e, (void *)ph->p_va, ph->p_memsz);
            memcpy((void *)ph->p_va, binary + ph->p_offset, ph->p_filesz);
            memset((void *)(ph->p_va) + ph->p_filesz, 0, ph->p_memsz - ph->
                p_filesz);
        }
    }
    e->env_tf.tf_eip = elf->e_entry;

    lcr3(PADDR(kern_pgdir));
    region_alloc(e, (void *)(USTACKTOP - PGSIZE), PGSIZE);

    return;
}
```

1.2.5 env_create

这个函数需要做就是将代码导入内存中，需要分两布：第一创建进程的地址空间的页目录以及设置环境变量，第二是将目标文件的代码导入内存中。

```
void
env_create(uint8_t *binary, size_t size, enum EnvType type)
{
    struct Env * e;
    int r = env_alloc(&e, 0);
    if (r < 0) {
        panic("env_create: %e\n", r);
    }
    load_icode(e, binary, size);
    e->env_type = type;
    return;
}
```

1.2.6 env_run

只需要进行切换一下即可。遗留问题如果 curenv 的状态为别的状态怎么办？之后回来再来看好了。

```
void
env_run(struct Env *e)
{
    if (curenv != NULL) {
        // context switch
        if (curenv->env_status == ENV_RUNNING) {
            curenv->env_status = ENV_RUNNABLE;
        }
        // how about other env_status ? e.g. like ENV_DYING ?
    }
    curenv = e;
    curenv->env_status = ENV_RUNNING;
    curenv->env_runs++;

    lcr3(PADDR(curenv->env_pgdir));

    env_pop_tf(&curenv->env_tf);
    panic("env_run not yet implemented");
}
```

gdb 得到结果顺利到达 int $0x30 处。

1.3 Exercise 3