Operating System MIT 6.828 JOS Lab4 Report

Computer Science
ChenHao(1100012776)

2013 年 10 月 26 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？

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);
}
```

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。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 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

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() 的执行需要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，而且一定程度下节省了 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_pagfalut_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 的之前只需要把 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;
```

```
}
```