Lab3实验报告

> 学号：21371220
>
> 姓名：杨硕

一、思考题

Thinking 3.1

根据自映射原理：

设页表基地址为PT_{base}

页目录基地址为$PD_{base} = PT_{base} + (PT_{base} >> 10)$

自映射页目录项 $PDE_{selfmap} = PD_{base} + (PT_{base} >> 20)$

而`e->env_pgdir[PDX(UVPT)]`即为`*(e->env_pgdir + PDX(UVPT)*4)`，根据宏定义，即为`*(e->env_pgdir + (UVPT>>22)*4)`=`*(e->env_pgdir + (UVPT>>20))`

`UVPT`就是页表基地址的虚拟地址，也就是说，`e->env_pgdir[PDX(UVPT)]`的地址就是$PDE_{selfmap} = PD_{base} + (PT_{base} >> 20)$，根据自映射关系，需要在这个地方存放自映射页目录项的物理地址，也就是`PADDR(e->env_pgdir) | PTE_V`。这里，使用`PADDR`宏获得物理地址，`PTE_V`为有效位

Thinking 3.2

`elf_load_seg`函数定义为：

```
int elf_load_seg(Elf32_Phdr *ph, const void *bin, elf_mapper_t map_page, void
*data);
```

`load_icode`和`load_icode_mapper`函数定义为：

```
static void load_icode(struct Env *e, const void *binary, size_t size);
static int load_icode_mapper(void *data, u_long va, size_t offset, u_int perm,const
void *src, size_t len);
```

在`env.c`中，我们找到`load_icode`函数对`elf_load_seg`函数的调用为

```
elf_load_seg(ph, binary + ph->p_offset, load_icode_mapper, e)
```

根据前面的参数定义，data这一参数的来源，实际就是`e`这个进程指针，也就是传入`load_icode`函数的参数`struct Env *e`

没有这个参数，就无法完成加载镜像

首先，`elf_load_seg`函数实际是在内部调用了回调函数`map_page`，在这里，也就是`load_icode_mapper`函数。而`load_icode_mapper`函数用来完成单页的加载，加载的实现需要其函数参数的`data`，也就是`elf_load_seg`函数参数的`data`，也就是传入`load_icode`函数的参数`struct Env *e`，所以这个参数不能去掉

Thinking 3.3

- 首尾均不对齐
- 首对齐尾不对齐
- 首不对齐尾对齐
- 首尾均对齐。

Thinking 3.4

这里的`env_tf.cp0_epc`存储的是虚拟地址

Thinking 3.5

5 个异常处理函数的具体实现位置在`genex.S`中

```
NESTED(handle_int, TF_SIZE, zero)
    mfc0    t0, CP0_CAUSE
    mfc0    t2, CP0_STATUS
    and     t0, t2
    andi    t1, t0, STATUS_IM4
    bnez    t1, timer_irq
    // TODO: handle other irqs
timer_irq:
    sw      zero, (KSEG1 | DEV_RTC_ADDRESS | DEV_RTC_INTERRUPT_ACK)
    li      a0, 0
    j       schedule
END(handle_int)

BUILD_HANDLER tlb do_tlb_refill

#if !defined(LAB) || LAB >= 4
BUILD_HANDLER mod do_tlb_mod
BUILD_HANDLER sys do_syscall
#endif

BUILD_HANDLER reserved do_reserved
```

Thinking 3.6

```
LEAF(enable_irq)
    li      t0, (STATUS_CU0 | STATUS_IM4 | STATUS_IEc)
    mtc0    t0, CP0_STATUS
    jr      ra
END(enable_irq)
```

前两行把STATUS_CU0 | STATUS_IM4 | STATUS_IEc的值写入t0，再把t0的值给CP0_STATUS，这两行的作用是开启四号中断；第三行，函数返回

```
timer_irq:
    sw      zero, (KSEG1 | DEV_RTC_ADDRESS | DEV_RTC_INTERRUPT_ACK)
    li      a0, 0
    j       schedule
END(handle_int)
```

第一行，响应时钟中断

第二行，把a0设为0，作为参数传给schedule函数

第三行，跳转到schedule函数

Thinking 3.7

设置了一个进程就绪队列（调度链表），并且给每一个进程添加了一个时间片，这个时间片起到计时的作用。一旦时间片用完，则代表该进程需要执行时钟中断操作，将该进程移动到就绪队列队尾，并复原其时间片。再从就绪队列队首取出一个新的进程进行调度，如此循环，实现进程切换

二、实验难点

exercise 3.12 schedule函数

本题需要完成实现进程切换的schedule函数，难点在于，需要充分考虑应进行进程切换的各种情况

• 尚未调度过任何进程

• 当前进程已经用完了时间片

• 当前进程不再就绪（如被阻塞或退出）

• yield 参数指定必须发生切换

因此，第一步，我们需要进行判断

```
if (yield != 0 || count == 0 || e == NULL || e->env_status !=
ENV_RUNNABLE)
```

当上述四种情况的其中一种发生时，就需要进行进程切换，当然根据不同的情况，我们还需要进一步判断

- 如果当前进程仍是就绪的，但需要发生进程切换，我们需要把当前进程插入到就绪队列队尾

  ```
  if (e != NULL) {
      if (e->env_status == ENV_RUNNABLE){
          TAILQ_INSERT_TAIL(&env_sched_list, e, env_sched_link);
      }
  }
  ```

- 如果是需要从就绪队列取出一个新的进程，这时我们应该先判断就绪队列是否为空，即没有就绪的进程可用，这时应该报错

```
if (TAILQ_EMPTY(&env_sched_list)) {
    panic("if that list is empty");
}
```

不为空，就从队首取出一个进程进行调度，注意取出进程后，还要记得将其从就绪队列移除，并将该进程的时间片数量设为其优先级

```
e = TAILQ_FIRST(&env_sched_list);
TAILQ_REMOVE(&env_sched_list, e, env_sched_link);
count = e->env_pri;
```

当不需要进程切换时，就正常执行当前进程的调度，并让时间片数量减一

综上，schedule函数代码如下

```
void schedule(int yield) {
    static int count = 0; // remaining time slices of current env
    struct Env *e = curenv;
    if (yield != 0 || count == 0 || e == NULL || e->env_status !=
ENV_RUNNABLE) {
        if (e != NULL) {
            if (e->env_status == ENV_RUNNABLE){
                TAILQ_INSERT_TAIL(&env_sched_list, e, env_sched_link);
            }
        }
        if (TAILQ_EMPTY(&env_sched_list)) {
            panic("if that list is empty");
        }
        e = TAILQ_FIRST(&env_sched_list);
        TAILQ_REMOVE(&env_sched_list, e, env_sched_link);
        count = e->env_pri;
    }
    count--;
    env_run(e);
}
```

三、实验心得

相比前面几次实验，lab3的实现难度笔者感觉是降低了，只需要仔细阅读指导书以及代码前的Hint，理解每个要实现的函数的功能及其实现机制，便可以很顺利的完成任务

当然，对于进程调度的学习，仅仅完成课下实验是不够的，还是需要对每一步的操作充分理解，与理论课的学习相结合，才能很好的掌握这一部分的知识

实验中，遇到过不少"能写出代码，但不完全理解是什么意思"的情况，或是对Hint提示的操作流程并没有很清楚理清思路，在上网查阅相关知识，以及询问同学后，有了更清楚的认识，这样不仅实现任务更轻松，并且也获得了实际的知识，而不是稀里糊涂的单单完成实验而已