思考题

Thinking 3.1

> 请结合 MOS 中的页目录自映射应用解释代码中 e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_V 的含义。

- **实现页目录自映射**

  `e->env_pgdir`为e所在进程的页目录基地址的虚拟地址，`PDX(UVPT)`为UVPT所处的页目录项的索引，`PADDR(e->env_pgdir)`为页目录基地址，`PTE_V`为权限位。
  因此该段代码是将UVPT虚拟地址映射到页目录本身的物理地址（基地址），并设置权限位。

Thinking 3.2

> elf_load_seg 以函数指针的形式，接受外部自定义的回调函数 map_page。请你找到与之相关的 data 这一参数在此处的来源，并思考它的作用。没有这个参数可不可以？为什么？

- `data` 来源
  由`load_icode()`函数中调用`panic_on(elf_load_seg(ph, binary + ph->p_offset, load_icode_mapper, e));`可知进程此处`data`来源为**进程控制块指针** `e`。
- 作用

  `load_icode_mapper()`函数：

  ```
  static int load_icode_mapper(void *data, u_long va, size_t offset, u_int
  perm, const void *src,size_t len) {
      struct Env *env = (struct Env *)data;
      struct Page *p;
      int r;
      page_alloc(&p);
      if (src != NULL) {
          void *dst = (void *)page2kva(p) + offset;
          memcpy(dst, src, len);
      }
      return page_insert(env->env_pgdir, env->env_asid,p, va, perm);
  }
  ```

  由上述代码和`elf_load_seg`函数定义`int elf_load_seg(Elf32_Phdr *ph, const void *bin, elf_mapper_t map_page, void *data);`可知`data`为回调函数`map_page`**提供参数**。

  因为`elf_load_seg`需要满足各种使用场景，因此需要传入的回调函数和其参数会**变化**，因此通过`void`类型传参来实现类似**泛型**的效果。

Thinking 3.3

> 结合 elf_load_seg 的参数和实现，考虑该函数需要处理哪些页面加载的情况。

- 段的虚拟地址与页边界不对齐：使用`ROUNDDOWN(va, BY2PG)`
- 文件大小小于一个页面：直接用回调函数`map_page`分配页面
- 文件大小大于一个页面：逐页进行上述分配

Thinking 3.4

> 思考上面这一段话，并根据自己在 **Lab2** 中的理解，回答：
>
> • 你认为这里的 env_tf.cp0_epc 存储的是物理地址还是虚拟地址?

- 虚拟地址

  > `env_tf.cp0_epc`储存的是进程恢复运行时 PC 应恢复到的位置，而CPU发出的地址是虚拟地址。

Thinking 3.5

> 试找出 0、1、2、3 号异常处理函数的具体实现位置。8 号异常（系统调用）涉及的 do_syscall() 函数将在 Lab4 中实现。

- 0号`handle_int`：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)
```

genex.S:

```
.macro BUILD_HANDLER exception handler
NESTED(handle_\exception, TF_SIZE + 8, zero)
    move    a0, sp
    addiu   sp, sp, -8
    jal     \handler
    addiu   sp, sp, 8
    j       ret_from_exception
END(handle_\exception)
.endm
```

```
BUILD_HANDLER tlb do_tlb_refill
BUILD_HANDLER mod do_tlb_mod
BUILD_HANDLER sys do_syscall // 未实现
```

因此2,3,4号均通过genex.S中的`BUILD_HANDLER`宏实现。

- 1号`hand_mod`：`handler`在tlbex.c中的`do_tlb_mod`实现
- 2号`hand_tlb`：`handler`在tlbex.c中的`do_tlb_refill`实现
- 3号`hand_sys`：`handler`在tlbex.c中的`do_syscall`实现（代码未实现）

Thinking 3.6

> 阅读 init.c、kclock.S、env_asm.S 和 genex.S 这几个文件，并尝试说出enable_irq 和 timer_irq 中每行汇编代码的作用。

- enable_irq:

```
LEAF(enable_irq)
        li      t0, (STATUS_CU0 | STATUS_IM4 | STATUS_IEc) // 设置状态位(CP0使能|IM4中断使能|中断使能)，并将该值写入t0寄存器
        mtc0    t0, CP0_STATUS // 将状态位设置写入CP0的STATUS寄存器
        jr      ra // 函数跳回
END(enable_irq)
```

- timer_irq:

```
timer_irq:
        sw      zero, (KSEG1 | DEV_RTC_ADDRESS | DEV_RTC_INTERRUPT_ACK) // 写该地址响应中断
        li      a0, 0 // schedule 函数参数
        j       schedule // 调用schedule 函数进行进程调度
```

Thinking 3.7

> 阅读相关代码，思考操作系统是怎么根据时钟中断切换进程的。

1. 通过异常分发程序`exc_gen_entry`保存上下文，并跳转到对应的异常处理程序`handle_int`。
2. 在异常处理程序中进一步细分终端类型`timer_irq`。
3. 调用`schedule`函数，进行进程调度。
4. `schedule`函数首先进行判断，若时间片用完或进程不就绪或尚未调用进程或`yield`参数指定必须发生改变，则根据调度算法选择一个新进程继续执行，如原进程仍就绪，则将其上下文保存并再次进入就绪队列。
5. 新进程通过调用`env_run`函数被执行。下次时钟中断发生时，重复上述步骤。

难点分析

自映射

- 页目录基地址（自映射）：`UVPT + PDX(UVPT)`
- 自映射使得页表和页目录可以看做相同的结构

其他

- 调用函数时要考虑异常返回值的处理
- 获取链表元素时要注意判断空的情况

实验体会

- 相比LAB2简单一些（感觉理解LAB2真的很重要哇），但很多常用宏和其参数需要记忆，同时还需要将LAB3中的部分内容和LAB2进行类比才能更好的完成。
- 在实验过程中逐渐意识到不能只关注当前工作文件的内容，还需要考虑项目全局的细节。