Lab3实验报告

思考题部分

Thinking 3.1

- `e->env_pgdir[PDX(UVPT)]`中，`e->env_pgdir`代表的是页目录的起始地址，`PDX(UVPT)`代表的是页目录应该在第几个页目录项中。`e->env_pgdir[PDX(UVPT)]`作为一个整体代表着页目录自映射所在的页目录项。
- `PADDR(e->env_pgdir) | PTE_V`代表的是页目录所在的物理页框号附加上有效位，是页目录自映射的页目录项中应该填充的内容。
- `e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_V`实际完成的就是建立页目录自映射的过程。

Thinking 3.2

data来自于`load_icode`函数中传入的参数`Struct Env *e`，其含义代表二进制镜像文件的目标加载进程的进程控制块指针。只有拥有该指针，才能完成加载镜像文件并建立页表映射的任务（需要获取目标进程的页目录起始地址）。没有该指针，就不能完成镜像文件加载的过程，也就不能完成进程的创建任务。

Thinking 3.3

- 无.bss段

1. 起始地址没有页对齐
    - 如果该段长度极小，小于`BY2PG - offset`，直接复制`bin_size`，并建立映射
    - 如果长度大于等于`BY2PG - offset`，就先复制起始地址之后的不满一页的片段，再按整页复制（最后一页如果不满的话只复制一部分）并建立映射。
2. 起始地址已经页对齐，则直接进行页复制（最后一页如果不满的话只复制一部分）并建立映射。

- 有.bss段

复制.data与.text部分与无.bss相同

由于在复制.text与.data片段时，末尾可能不是页对齐的。因此需要记录上次复制的位置，避免复制时修改了.data与.txt部分的内容。

Thinking 3.4

`env_tf.cp0_epc`存储的是虚拟地址

Thinking 3.5

五个异常处理函数都在kern/genex.S中定义

Thinking 3.6

```
LEAF(enable_irq)
    li      t0, (STATUS_CU0 | STATUS_IM4 | STATUS_IEc)       //设置t0值为允许4号中断，
CPU中断位开启，CU0为1（可以在用户模式下使用一定特权指令）
    mtc0    t0, CP0_STATUS                                   //将SR寄存器的值设置为t0内容
    jr      ra                                               //返回
END(enable_irq)
```

```
timer_irq:
    sw      zero, (KSEG1 | DEV_RTC_ADDRESS | DEV_RTC_INTERRUPT_ACK)//相应模拟的外部中断
    li      a0, 0                                            //传入参数yield为0
    j       schedule                                         //跳转到调度函数
END(handle_int)
```

Thinking 3.7

操作系统实现了一个进程调度队列，并为每个进程设置了对应的时间片。通过不断触发时钟中断来执行当前进程并减少其时间片。如果时间片减至0或有yield信号或者该进程状态跳转到not_runnable状态就切换进程。如果被切换的进程仍然处于runnable状态，就重新将其放在调度队列队尾等待下一次调度。

实验难点/对实验内容的理解

env_init与env_setup_vm相关

`env_init`实际完成了两个任务：

1. 将所有进程控制块状态初始化，并添加进`env_free_list`(空闲队列)中。
2. 建立好`envs`和`pages`从虚拟地址到物理地址的映射(完成页表内容的填写)，存储在模板页表中(所有进程共同需要使用)。

`env_setup_vm`完成了两个任务：

1. 申请页作为页目录，将模板页表中对应的该部分复制进每个进程申请的页目录中。
2. 完成页目录自映射的过程。

创建进程

env_alloc（建立进程环境）

1. 申请了一个空的进程控制块
2. 利用env_setup_vm完成页目录内容的部分填充
3. 分配asid，设置parent_id
4. 完成cp0_status与$sp的设置

```
e->env_tf.cp0_status = STATUS_IM4 | STATUS_KUp | STATUS_IEp;
```

- STATUS_IM4（允许四号中断）

- STATUS_KUp与STATUS_IEp，由于SR中存在二重栈，且每次进行进程调度时都会将KUp和IEp拷贝回KUc 和 IEc，因此在此处置位，并在进程调度时利用rfe设置SR寄存器的状态

5. 将进程控制块移出空闲队列

load_icode（加载二进制镜像文件）

1. 利用elf_from来检查elf文件的合法性（魔数，大小等特征的验证）并返回对应的elf文件头ehdr
2. 遍历所有的端头，如果该段需要加载到内存中，则利用elf_load_seg加载该段。

elf_load_seg

elf_load_seg利用了load_icode_mapper申请了部分物理页复制了目标段的数据，并建立了页表的映射关系。包含.data,.text,.bss三个部分。具体复制过程整体上按页复制，并考虑多种不满整页的具体情况（具体可参考Thinking 3.3），且.bss部分默认赋值为0。

3. 设置epc为可执行文件的程序入口e_entry。

env_create

env_create实际就是利用了env_alloc与load_icode完成了进程创建的相关任务，并设置了优先级pri和状态runnable。加入到进程调度队列env_sched_list中。

进程运行与切换——env_run

1. 保存当前进程的上下文信息(寄存器信息，cp0的相关信息)到 `KSTACKTOP` 的顶层位置，如果没有就不保存。
2. 切换 curenv 为即将运行的进程，该进程的运行次数++。
3. 设置全局变量 cur_pgdir 为当前进程页目录地址，在 TLB 重填时将用到该全局变量。
4. 调用 env_pop_tf 函数，恢复现场、异常返回。

进程调度函数——schedule

```
void schedule(int yield) {
    static int count = 0;                    // remaining time slices of current env
    struct Env *e = curenv;
    count--;
    if(yield||count==0||e==NULL||e->env_status!=ENV_RUNNABLE){  //需要获取新进程的四个条件
        if(e!=NULL&&e->env_status==ENV_RUNNABLE){                //当前进程如果还是就绪需要放到
            TAILQ_INSERT_TAIL(&env_sched_list,e,env_sched_link);  调度队尾，以便再进行调度
        }
        panic_on(TAILQ_EMPTY(&env_sched_list));
        e=TAILQ_FIRST(&env_sched_list);
        TAILQ_REMOVE(&env_sched_list,e,env_sched_link);
        count = e->env_pri;                                      //获取其优先级作为运行的时间片数量
    }
```

```
    env_run(e);                                                    //调用env_run完成进程切换与页表
更改
}
```

实验体会

本次实验整体上完成了初始化所有进程控制块——申请进程控制块，建立进程环境并加载可执行文件——调度运行进程并根据时间片轮转算法切换进程的整个流程。思路上比较符合逻辑发展的顺序，因此从思路层面理解起来会更快一些。

在代码实现的层面上，与上次的Lab2相同，多文件，多函数，多宏定义，汇编语言和C语言的混合使用仍然给代码层面的理解造成了较大的困难，也给debug造成了相当的困难。所幸有了Lab2的基础，对整套操作系统的include部分更加熟悉了一些，能够帮助理解。但在完成相应的exercise时，还是比较依赖提示而不能够从思路上自然而然地填写对应部分的代码。