Lab3实验报告

一、实验思考题

Thinking 3.1

对于`envs[ENVX(envdi)]`处的控制块，其id唯一，但asid会因为进程运行和消亡而变化，所以`envid`和`(envs + ENV(envid))->env_id`在asid段的内容可能不同，需要判断

Thinking 3.2

结合include/mmu.h 中的地址空间布局，思考env_setup_vm 函数：

- **UTOP 和ULIM 的含义分别是什么，UTOP 和ULIM 之间的区域与UTOP以下的区域相比有什么区别?**

  UTOP：`0x7f40_0000`，用户进程读写部分的最高地址

  ULIM：`0x8000_0000`，用户进程的最高地址，kuseg顶部

  UTOP~ULIM区域包含ENVS、PAGES、User VPT，用户只读不可写；UTOP以下区域用户可读可写。

- **请结合系统自映射机制解释代码中pgdir[PDX(UVPT)]=env_cr3的含义。**

  由于页目录自映射可知，UVPT处的页目录项`pgdir[PDX(UVPT)]`为页表起始地址对应的页目录项，对应进程页目录物理地址`env_cr3`

- **谈谈自己对进程中物理地址和虚拟地址的理解。**

  每一个进程拥有4GB虚拟地址，然而约定可操作的区域均为kuseg段（且用户真正可写的部分在UTOP之下）。通过每个进程的页目录（env_cr3即可找到物理真身）映射到相应物理地址，当然这部分交给操作系统和mmu-tlb，进程自己无需操心。

Thinking 3.3

找到 user_data 这一参数的来源，思考它的作用。没有这个参数可不可以？为什么？（可以尝试说明实际的应用场景，举一个实际的库中的例子）

```
int load_icode(struct Env *e, u_char *binary, u_int size)
```

调用，将`struct Env *e`传入作为`void *user_data`

```
int load_elf(u_char *binary, int size, u_long *entry_point, void *user_data,int
(*map)(...)) {
    ...
        r = map(
                phdr->p_vaddr, phdr->p_memsz,
                binary + phdr->p_offset, phdr->p_filesz, user_data
            );
    ...
}
```

使用，将`void *user_data`传入作为`void *user_data`

```
static int load_icode_mapper(u_long va, u_int32_t sgsize,
                             u_char *bin, u_int32_t bin_size, void *user_data) {
    struct Env *env = (struct Env *)user_data;
    ...
        (env->env_pgdir)
}
```

user_data来源于所选进程控制块struct Env *e，在`load_icode_mapper`函数中提供进程页目录的kernel段虚拟地址，在分配页（增加映射）中不可或缺。

Thinking 3.4

结合load_icode_mapper 的参数以及二进制镜像的大小，考虑该函数可能会面临哪几种复制的情况？你是否都考虑到了？

va头部页不对齐；bin_size尾部页不对齐；sg_size尾部页不对齐。还有需要注意，每个部分可能连一个页都没有跨过，

（但给的代码和注释指引像是完全不考虑的样子）

Thinking 3.5

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.pc 存储的是物理地址还是虚拟地址？**

  虚拟地址

- **你觉得entry_point其值对于每个进程是否一样？该如何理解这种统一或不同？**

  load_icode()中entry_point来自调用的load_elf()中`*entry_point = ehdr->e_entry;`为ELF文件入口虚拟地址，对于每个进程自己的逻辑地址空间，其值应该相同。入口相同，简化操作

Thinking 3.6

请查阅相关资料解释，上面提到的epc是什么？为什么要将env_tf.pc设置为epc呢？

epc为发生精确异常时的受害指令pc，由CPU完成填写。在env_run()时若当前curenv不为NULL，说明之前尚有程序在跑，是通过异常被中断的（如计时器中断），则下次运行时要从epc处开始运行

Thinking 3.7

关于 TIMESTACK，请思考以下问题：

- **操作系统在何时将什么内容存到了 TIMESTACK 区域**

  TIMESTACK：`0x8200_0000`.

  操作系统的时钟中断异常环境暂存区，不如说env_destroy()、env_run()通过它来完成进程切换

- **TIMESTACK 和 env_asm.S 中所定义的 KERNEL_SP 的含义有何不同**

```
* .macro get_sp
    mfc0    k1, CP0_CAUSE
    andi    k1, 0x107C
    xori    k1, 0x1000
    bnez    k1, 1f
    nop
    li  sp, 0x82000000  # 根据CP0_CAUSE情况选择sp， 此处应为时钟中断等
    j   2f
    nop
```

```
1:
  bltz    sp, 2f
  nop
  lw  sp, KERNEL_SP    # 内核处存sp的，应该是系统调用（？没细想，好像不对
  nop

2:    nop

.endm
```

Thinking 3.8

试找出上述 5 个异常处理函数的具体实现位置

handle_int等4个异常：写在lib/genex.S文件中

handle_sys：写在lib/syscall.S文件中

Thinking 3.9

阅读 kclock_asm.S 和 genex.S 两个文件，并尝试说出 set_timer 和timer_irq 函数中每行汇编代码的作用

```
LEAF(set_timer)

    li t0, 0xc8
    sb t0, 0xb5000100    # 向0xb500_0100写入0xc8:时钟中断的频率200次/s
    sw  sp, KERNEL_SP    # 保存栈指针到KERNEL_SP
    setup_c0_status STATUS_CU0|0x1001 0
                         # 操纵Status寄存器，允许时钟中断(4号)，开中断
    jr ra
    nop

END(set_timer)
```

```
timer_irq:

    sb zero, 0xb5000110 # 向0xb500_0110写入0:估摸着在关gxemul某个功能
1:  j   sched_yield     # 进入sched_yield，开始调度
    nop
    j   ret_from_exception
    nop
```

Thinking 3.10

阅读相关代码，思考操作系统是怎么根据时钟周期切换进程的。

gxemul按设定，定期产生时钟中断，进程被迫异步发生异常，进入sched_yield函数，认为消耗一个时间片。当设定时间片用完/进程运行完/进程被设为ENV_NOT_RUNNABLE（多在被设置状态后直接进入此处）时进行进程调度，切换新进程开始运行。

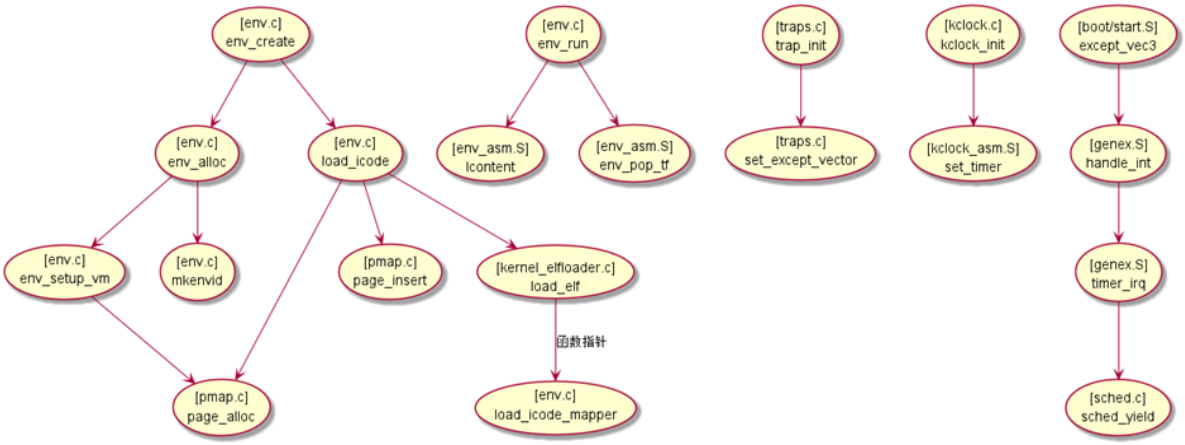
二、实验难点

1 进程块初始化、分配、运行概念厘清

就因为这点，在lab3-1-exam添加版本号时，在env_alloc时就分配了asid，导致运行全错，虚空debug还莫名其妙，最终挂掉。殊不知asid是针对运行时的进程而言的，这里有被`mkenvid()`函数、`e->env_id`误导

2 复杂的函数调用关系梳理

例如**加载二进制文件**（好像现在感觉那里.bss处理解错了，要重写以下）那里。起始整个MOS都如此，现在我还有好些调用流程关系没搞清。只有完全厘清所有运行顺序和逻辑，才能少写bug/debug游刃有余。像env.c等复杂文件可以写文档梳理所有函数。然后调用关系导图不可少，这里嫖了一张，尤其是那种汇编和C交叉乱用的流程



3 内存管理贯穿始终

个人认为操作系统最大boss，需要彻底想清

三、体会与感想

做lab3上半时还惦记着对lab2很多概念模棱两可的理解，写得很混乱，由其实关于进程空间和加载二进制镜像处相关内容，到看lab4指导书时突然感觉想通了内存管理核心概念，再回看一切就变得自然多了。

> 以下为做lab3课下过程中顺序产生的感想

1.env_init(void)中说是/*Step 1: Initial env_free_list. */给人误解只需要初始化free_list即可

3.“ 为了进一步简化你的理解难度，我们已经为你定义好了这个“自定义函数”的框架。load_elf() 函数会从ELF 文件文件中解析出每个segment 的四个信息：va(该段需要被加载到的虚地址)、sgsize(该段在内存中的大小)、bin(该段在ELF 文件中的起始位置)、bin_size(该段在文件中的大小)，并将这些信息传给我们的“自定义函数” ”这几段

看了代码注释才理解，所谓的自定义函数是指前面写的load_icode_mapper()。**理解太难了！**

看代码注释要比看指导书容易理解的多

4.`load_icode_mapper`函数的填写有点离谱。指导书强调了va与页各种不对齐的情况，给的代码和注释完全没说这一点，直接一个for循环一个while，要不是login256我都不敢加什么东西

5 记录.text.exc_vec3 段需要被链接器放到特定的位置，在 R3000 中这一段是要求放到地址 0x80000080 处，这个地址处存放的是异常处理程序的入口地址；相应的.S文件语法和之前计组写的汇编有些不同，还有很多Derivative不会用

6 tools/scse0_3.lds中那句Exercise3.13的注释位置插得很有误导性，虽然明显不应该从其下面插句子，但那么放着很别扭

7 include/stackframe.h不用.S，没有代码高亮，很怪，不理解