Lab3实验报告

Thinking

T1

思考envid2env 函数:

为什么envid2env 中需要判断e->env_id != envid 的情况？如果没有这步判断会发生什么情况?

`envid`的低10位(0-9)是对应的env结构体在envs中的索引，第11-16位是asid。而e是由`e = envs + ENVX(envid)`得到的，`ENVX(envid)`仅仅是指envid的低10位。由于一个进程控制块可以被`mkenvid`多次，但只有一个进程能对应一个`envid`。所以必须判断`e->env_id != envid`， 确保要查询的`envid`对应的是当前正在运行的进程。如果没有这一步判断，可能会出现当前进程中不存在一个进程的envid是给定的envid，但是却查询到了一个进程。

T2

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, 是用户能操作(读与写)的最大地址；ULIM = 0x8000_0000, 是用户态地址空间最大值。它们之间的区域用户只能读，是为了让用户进程能在此空间查看其他进程的信息，用户在此出读取不会陷入内核；而UTOP以下区域用户可读可写。

• 请结合系统自映射机制解释Step4中pgdir[PDX(UVPT)]=env_cr3的含义。

UVPT意为User Virtual Page Table，是用户态二级页表基地址PT_{base}， UVPT到ULIM共4MB空间。PDX(UVPT)指UVPT的高10位。由于整个4MB页表空间的高10位都相同，而且高10位是页目录项索引，故pgdir[PDX(UVPT)]对应页目录页，页目录页的1024个页目录项对应了整个4MB页表空间。故该页表项存储的物理页号为页目录所在的物理页号。或者也可以这样理解，由于自映射机制下，映射页目录的页目录项$PTE_{self\text{-}mapping} = PT_{base} | PT_{base} >> 10 | PT_{base} >> 20$， 因此，pgdir[PDX(UVPT)] = pgdir[$PT_{base} >> 22$] = $PTE_{self\text{-}mapping}$， 其中页目录项所存物理页号就是页目录页的物理页号。

• 谈谈自己对进程中物理地址和虚拟地址的理解。

进程可见的是虚拟地址，而物理地址是内存中真正的索引。虚拟地址向物理地址的转换由MMU完成。

T3

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）

在`load_icode_mapper`函数中，有`struct Env *env = (struct Env *)user_data;`说明user_data是一个PCB指针。没有这个函数当然不行，因为该函数调用了函数`page_insert(env->env_pgdir, p, va+i, PTE_R)`和`p = page_lookup(env->env_pgdir, va+i, NULL)`，其中这两个函数都需要 被创建进程 的页目录基地址，如果没有这个参数，则无法找到该进程的页目录基地址，无法将elf文件装载到指定进程的虚拟地址上。

T4

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?

四个参数分别为：

> 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)

其中，va, sgsize, bin_size分别都对应两种情况，同时需要考虑一种特殊情况：

1. va是否BY2PG对齐
2. va+bin_size是否BY2PG对齐
3. va+sgsize是否BY2PG对齐
4. va与va+bin_size是否在同一个页面
5. va+bin_size与va+sgsize是否在同一个页面

对第一种情况，设定offset表示va相对于其所在虚拟页起始地址的偏移量。对第4种情况，取BY2PG-offset和bin_size的最小值。

```
if(offset){
    ...
    size = MIN(BY2PG-offset, bin_size-i);
    bcopy(bin+i, page2kva(p)+offset, size);
    ...
}
```

对第2种情况，取BY2PG和bin_size-i的最小值，i是已经加载的偏移量。

```
while(i < bin_size) {
    ...
    size = MIN(BY2PG, bin_size - i);
    bcopy(bin+i, page2kva(p), size);
    ...
}
```

对第3种情况，类似于第二种情况的处理方式。

对第5种情况，类似于第4种情况处理方式。

T5

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，env_tf.pc等于可执行文件中的ehdr->e_entry， 是程序入口的虚拟地址。即当文件被加载到进程空间里后，入口程序在进程地址空间里的地址。

• 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?

若不同进程所执行的程序 都相同，则entry_point相同，因为该值来自于可执行文件(ELF文件)。只要是执行相同的可执行文件，那么这个值就相同。但如果是不同的可执行文件，就不一定相同

T6

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?

根据*See mips run linux*，发生异常时，CPU硬件自动将异常返回地址填充到epc寄存器，异常返回地址为导致或遭受异常的指令的地址。因为当该被中断的进程再次需要执行时，会调用`env_run()`函数，该函数会调用`env_pop_tf`函数，会env_tf.pc赋值给pc寄存器。因此env_tf.pc需要存放进程恢复异常后开始执行的第一条指令的地址，保证中断结束后进程可以恢复到原来的执行状态。

T7

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

两个地方：

1. 当发生异常时，执行异常处理程序会执行`SAVE_ALL`宏，该函数调用了`get_sp`宏，如果为 时钟中断则将sp置为`TIMESTACK-TF_SIZE`，并将所有寄存器堆寄存器的值和所有CP0寄存器（除了CP0.PC）保存到TIMESTACK中。
2. 进程销毁时，将KERNEL_SP栈中内容保存到TIMESTACK中。

```
void env_destroy(struct Env *e){
    ...
    bcopy((void *)KERNEL_SP - sizeof(struct Trapframe),
          (void *)TIMESTACK - sizeof(struct Trapframe),
          sizeof(struct Trapframe));
    ...
}

```

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

由Lab4我们可以知道，发生时钟中断时，进程的上下文信息保存在TIMESTACK；发生系统调用时则保存在KERNEL_SP。

```
.macro get_sp      //macro
    mfc0    k1, CP0_CAUSE
    andi    k1, 0x107C
    xori    k1, 0x1000
    bnez    k1, 1f     //if is time_interrupt, k1 should be 0. not branch
                       //if is syscall, k1 should not be 0. branch
    nop
    li  sp, 0x82000000 // sp = TIMESTACK，这一步就是将sp设为TIMESTACK
    j   2f
    nop
1:
    bltz    sp, 2f     //sp < 0, branch
    nop
    lw  sp, KERNEL_SP
    nop
```

```
16
17  2:  nop
18  .endm
```

我们注意到，时间中断和系统调用对栈**$sp**的操作时不一样的，时钟中断是`li sp, 0x82000000`，系统调用为`lw sp, KERNEL_SP`。造成这种不一样的原因是**TIMESTACK**本身是一个宏，表示**0x82000000**；而**KERNEL_SP**是一个变量，相当于`int i = xxx`中的`i`。因此如果对要执行类似`sp = KERNEL_SP`这样指令的话，需要使用`lw`。

T8

试找出上述 5 个异常处理函数的具体实现位置。

`handle_int()`，`handle_reserved()`，`handle_mod()`，`handle_tlb()`都是在lib/genex.S中实现的。

其中`handle_reserved()`，`handle_mod()`，`handle_tlb()`的实现用到了宏`BUILD_HANDLER`。

```
1  .macro  BUILD_HANDLER exception handler clear
2      .align  5
3      NESTED(handle_\exception, TF_SIZE, sp)
4      .set    noat
5
6  nop
7
8      SAVE_ALL
9      __build_clear_\clear
10     .set    at
11     move    a0, sp
12     jal \handler
13     nop
14     j   ret_from_exception
15     nop
16     END(handle_\exception)
17 .endm
18
19 BUILD_HANDLER reserved do_reserved cli
20 BUILD_HANDLER tlb    do_refill   cli
21 BUILD_HANDLER mod    page_fault_handler cli
```

`handle_sys()`在`lib/syscall.S`实现

T9

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

1. `set_timer`

```
1  //lib.kclock_asm.S
2  .macro  setup_c0_status set clr
3  //定义一个宏setup_c0_status， 以及形参set 和 clr。set和clr使用时，前加\。
4  //该函数的作用是，将SR寄存器中set对应的位设为1，clr对应的位设为0
5      .set    push                //save all settings setted before， 比如之前的.set设
   置
6      mfc0    t0, CP0_STATUS      //t0 = CP0_STATUS，CP0_STATUS = $12
```

```
    or  t0, \set|\clr
    xor t0, \clr
    mtc0    t0, CP0_STATUS
    .set    pop                 //restore saved settings
.endm

    .text
LEAF(set_timer)

    li t0, 0xc8                     //为t0赋值0xc8
    sb t0, 0xb5000100               //将0xc8存入0xb5000100处，设置时钟发生中断信号的频率为每秒200次
    sw  sp, KERNEL_SP               //将栈指针的值存入KERNEL_SP中
    setup_c0_status STATUS_CU0|0x1001 0
    /*
    * 调用宏，将SR的第28位、第12位，第0位置1，即允许用户态使用CP0，
    * 允许4号中断（时钟中断），允许外部中断
    */
    jr ra                           //函数返回

    nop
END(set_timer)
```

2. `timer_irq`

```
timer_irq:

    sb zero, 0xb5000110
1:  j   sched_yield    //进程的调度
    nop
    j   ret_from_exception //如果之前没有写sched_yield函数，则会执行该函数，从异常返回；但当完成sched_yield函数后，不会执行该函数
    nop
```

T10

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。

当外部时钟发出周期性的中断信号时，pc跳转到异常处理程序`handle_int`，进入中断服务程序`time_irq`，执行`sched_yield`，进行进程的调度。

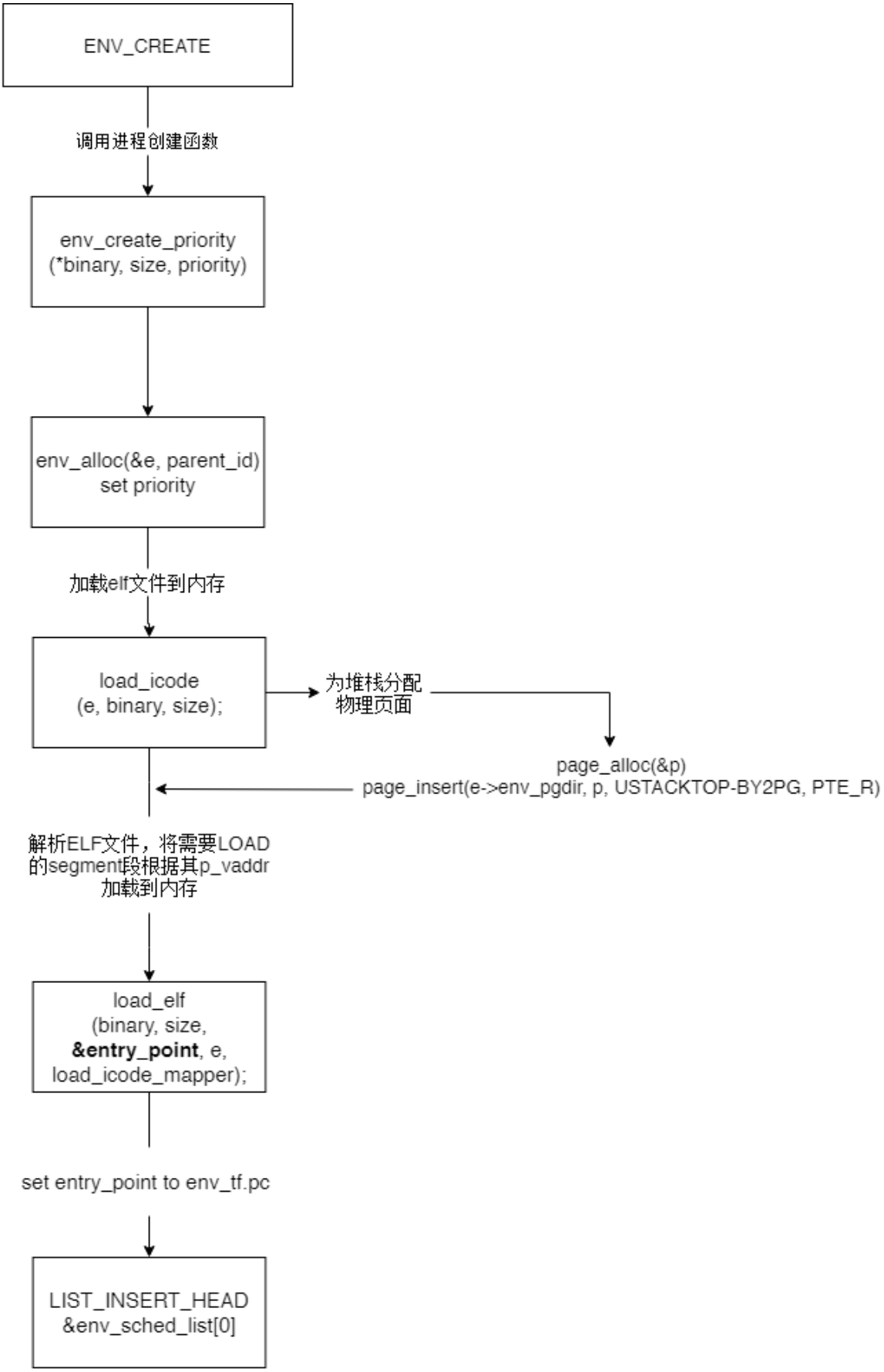
对于进程的调度，设置了两个调度链表，`active`和`expired`，`active`作为当前调度链表，当操作系统需要调度新的进程时，在当前调度链表中寻找可运行进程；`expired`链表存放时间片用完被替换下来的进程，当`active`链表为空时，则互换`active`和`expired`链表。

如果当前进程的时间片已经用完，则将其插入到`expired`链表尾端，并在`active`链表中从头找状态为`ENV_RUNNABLE`的进程，若找到则执行该进程；如果当前进程时间片没有用完，则当前进程时间片数量减1，继续执行。

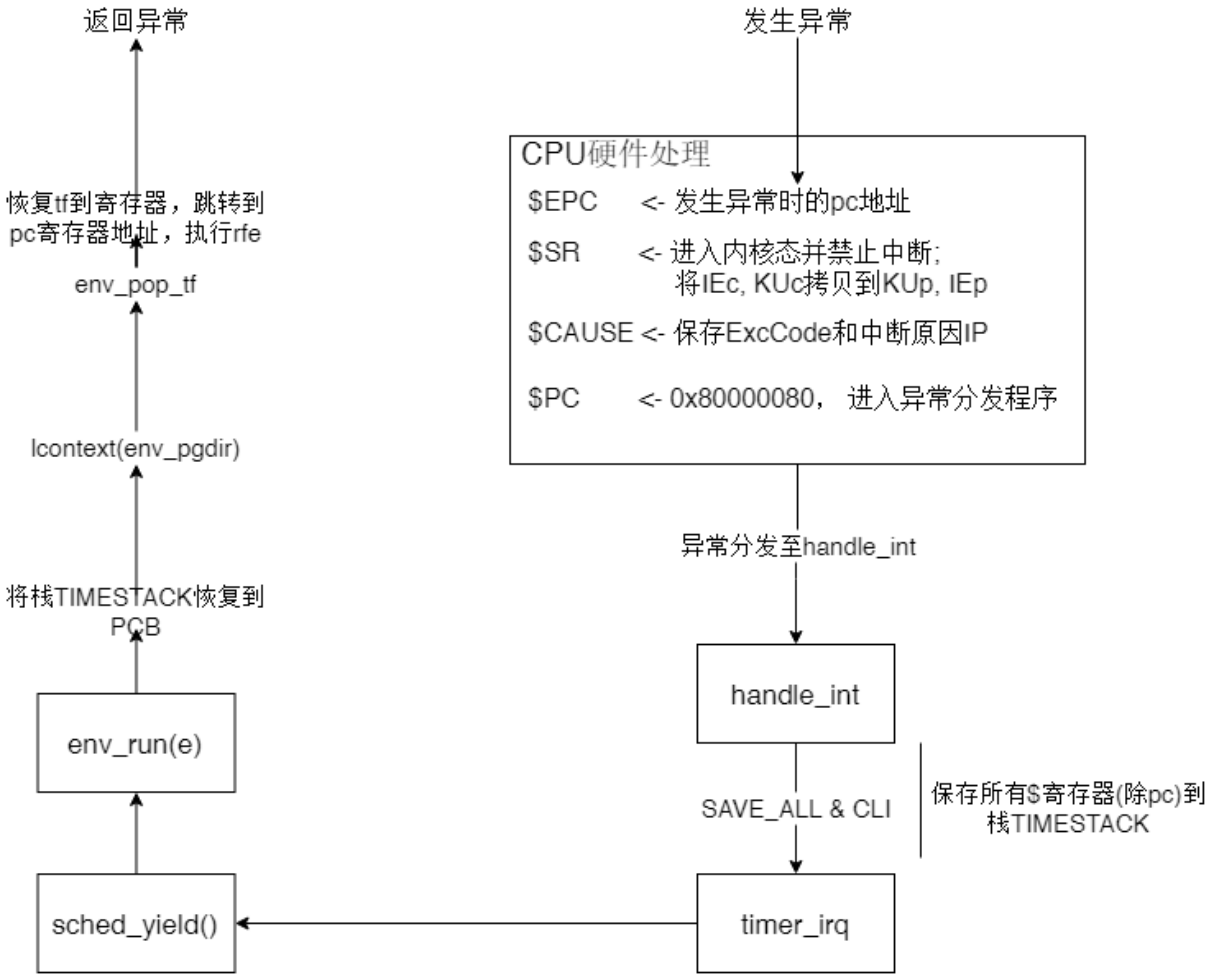
实验难点

该实验的难点主要集中在进程运行和异常处理的流程是否搞清楚了。

- 创建进程


ENV_CREATE
调用进程创建函数
env_create_priority
(*binary, size, priority)
env_alloc(&e, parent_id)
set priority
加载elf文件到内存
load_icode
(e, binary, size);
为堆栈分配
物理页面
page_alloc(&p)
page_insert(e->env_pgdir, p, USTACKTOP-BY2PG, PTE_R)
解析ELF文件，将需要LOAD
的segment段根据其p_vaddr
加载到内存
load_elf
(binary, size,
&entry_point, e,
load_icode_mapper);
set entry_point to env_tf.pc
LIST_INSERT_HEAD
&env_sched_list[0]

- 异常处理（针对时钟中断）



实验疑点

以下是本人做Lab3时产生的疑点，都已解决。

1. 内核页表的作用?

   页表本身作用有很多。

   - 内核启动过程中，为了实现分页机制，需要内核页表。

2. entry_point是干什么的?

   `entry_point`是ehdr->e_entry， 此字段指明程序入口的虚拟地址。即当文件被加载到进程空间里后，入口程序在进程地址空间里的地址。对于可执行程序文件来说，当 ELF 文件完成加载之 后，程序将从这里开始执行

3. `load_icode_mapper`函数中`va`如果不按页对齐，而那个页面已经存放别的内容了，有影响吗?

   各个进程是相互独立的，互相不可见的。创建进程时，对于进程低2G空间都是空的，不会有别的东西的。

4. tlb重填的时候，怎么知道在哪里找页表?

   进程开始执行时，函数`env_run()`会执行`lcontext()`函数，更新mCONTEXT为当前进程页表基地址。mCONTEXT 中存储了当前进程一级页表基地址位于 kseg0 的虚拟地址，tlb重填就是根据mCONTEXT重填。

5. **NESTED**函数的定义：

```
//include/asm/asm.h
#define NESTED(symbol, framesize, rpc)                  \
                .globl  symbol;                         \
                .align  2;                              \
                .type   symbol,@function;               \
                .ent    symbol,0;                       \
symbol:         .frame  sp, framesize, rpc
              //.frame framereg, framesize, returnreg

```

其中最关键的是**.frame**，该directive有三个操作数：

- **framereg**: 寄存器用来获取本地栈，通常为**$sp**
- **returnreg**：保存函数返回地址的寄存器，通常为**$0**，表明返回地址存在栈中。如果为叶子函数（不调用其他函数）则为**$31**
- **framesize**：为函数分配的栈帧大小，满足**$sp + framesize = previous $sp**

因此我们可以看到，非叶函数**NESTED(symbol, framesize, rpc)**，后两个参数其实是一个标识，便于程序员快速确定这个函数的栈帧大小。至于为什么我们代码中rpc是sp，还没有搞懂

体会与感想

本次实验没有那么难，虽然也难，总耗时无法估计，持续时间太长了。本人对本届指导书的评价还是蛮好的，读下来细细钻研还是能看懂的，也加添了学习操作系统的兴趣和热情。