OS Lab3 - ExprimentReport

1. 思考题

1 .为什么我们在构造空闲进程链表时必须使用特定的插入的顺序？(顺序或者逆序)

只有通过逆序插入，我们在每次从头部取出env时才会按照从小到大的顺序取出（第一次能够取出envs[0]）。

2思考env.c/mkenvid 函数和envid2env 函数：

• 请你谈谈对mkenvid 函数中生成id 的运算的理解，为什么这么做?

```
u_int mkenvid(struct Env *e)
{
    static u_long next_env_id = 0;

    /*Hint: lower bits of envid hold e's position in the envs array. */
    u_int idx = e - envs;

    /*Hint:  high bits of envid hold an increasing number. */
    return (++next_env_id << (1 + LOG2NENV)) | idx;
}
```

最后的envid的低十位是该env在总的envs组中的位置，可以直接取其低十位`ENVX()`获得该envid对应的在envs组中的index，便于访问。

而高位是关于该函数调用次数的一个递增量。

这样可以保证每次调用此函数确定的envid是一个不仅取决于env在envs数组中的位置，还取决于函数调用的次数，使得每个进程的envid独一无二。在进程被创建后杀死，再在同样的位置创建新的进程时，它们的id的高位是不同的。

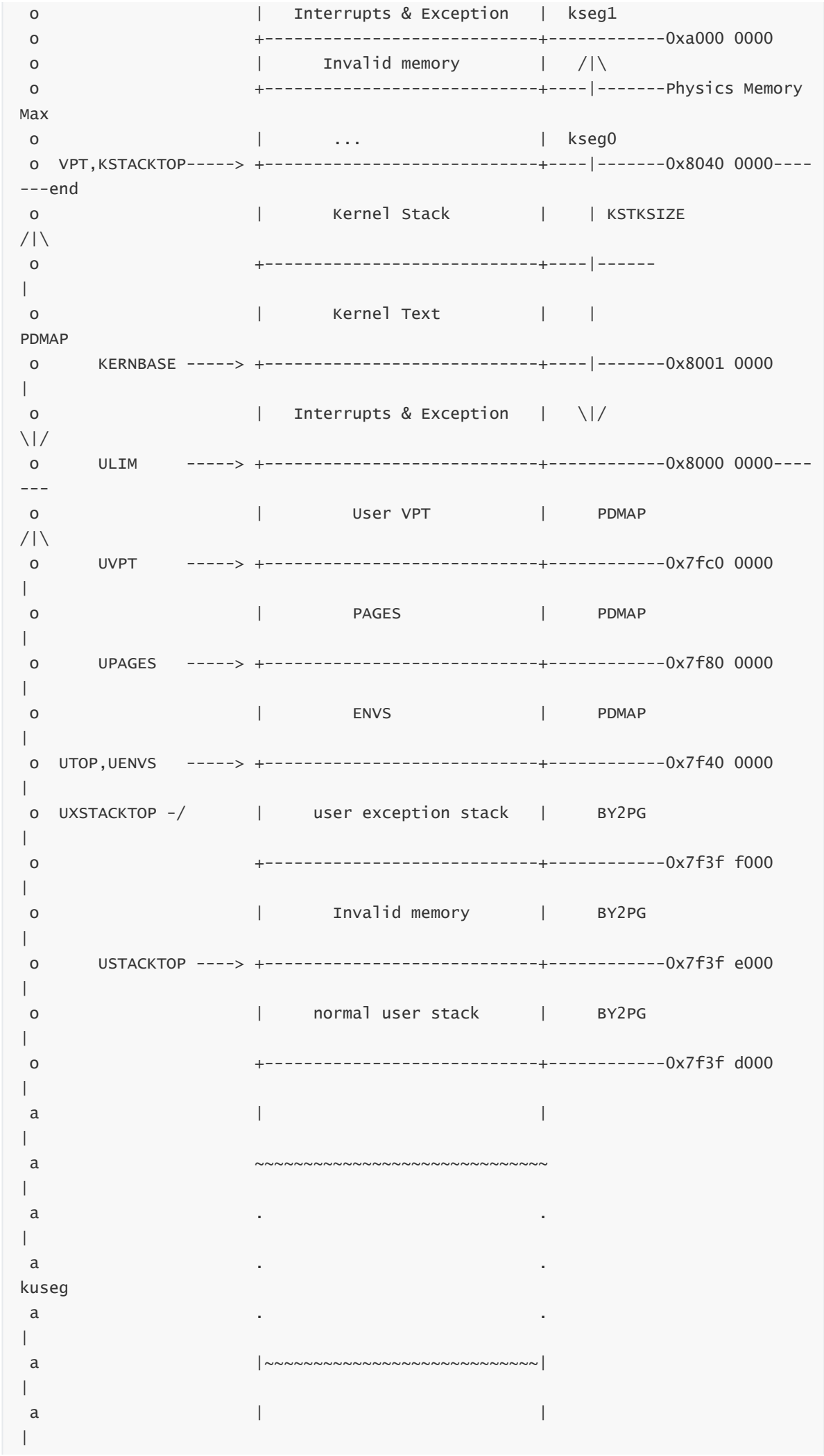
• 为什么envid2env 中需要判断e->env_id != envid 的情况？如果没有这步判断会发生什么情况?

如上题所述，两个进程可能具有相同的idx，但其env_id是不同的。如果没有这步判断可能导致将一个已经被杀死的进程当做现在在该进程块上注册的进程而导致出错。

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

• 我们在初始化新进程的地址空间时为什么不把整个地址空间的pgdir 都清零，而是复制内核的boot_pgdir作为一部分模板？(提示:mips 虚拟空间布局)

```
/*
 o     4G ----------->  +------------------------------+------------0x100000000
 o                      |       ...                    |  kseg3
 o                      +------------------------------+------------0xe000 0000
 o                      |       ...                    |  kseg2
 o                      +------------------------------+------------0xc000 0000
```


Interrupts & Exception
kseg1
0xa000 0000
Invalid memory
Physics Memory
Max
...
kseg0
VPT,KSTACKTOP
0x8040 0000
end
Kernel Stack
KSTKSIZE
Kernel Text
PDMAP
KERNBASE
0x8001 0000
Interrupts & Exception
ULIM
0x8000 0000
User VPT
PDMAP
UVPT
0x7fc0 0000
PAGES
PDMAP
UPAGES
0x7f80 0000
ENVS
PDMAP
UTOP,UENVS
0x7f40 0000
UXSTACKTOP
user exception stack
BY2PG
0x7f3f f000
Invalid memory
BY2PG
USTACKTOP
0x7f3f e000
normal user stack
BY2PG
0x7f3f d000
kuseg

```
 o       UTEXT    -----> +----------------------------+
|
 o                       |                            |      2 * PDMAP
\|/
 a     0 ------------>  +----------------------------+ ------------------------
---
 o
*/
```

内存布局图如上：ULIM是给用户分配的最大虚拟地址（0-2G）。其中：UTOP及其以下的部分是每个进程独有的，而UTOP-ULIM**虽属于用户地址空间，但进程在用户态仅具有读取的权利，实际由内核态的操作系统分配**（读取各个进程的状态，读取自己的页表完成地址的转换）再往上即内核态地址空间。

在该内存布局下， UTOP及其上的部分在为进程创建其内存空间时完全相同。因此，在创建页表时，有一半（实际略大于一半）的部分是相同的，因此可以将boot_pgdir作为模板。

• UTOP 和ULIM 的含义分别是什么，在UTOP 到ULIM 的区域与其他用户区相比有什么最大的区别?

ULIM是给用户分配的最大虚拟地址（0-2G）。其中：UTOP及其以下的部分是每个进程独有的，而UTOP-ULIM**虽属于用户地址空间，但进程在用户态仅具有读取的权利（查看自己各个进程的状态，读取自己的页表完成地址的转换），实际由内核态的操作系统分配并编辑**，可以将这一部分当做内核态暴露给用户态获取信息的部分。用户态下对此空间进行写操作也算作越界访问的异常(PTE_R位为0)。进程只能读写UTOP以下的空间。当然，不同于内核态，对这一区域的读取不被视作异常。

• 在env_setup_vm 函数的最后，我们为什么要让pgdir[PDX(UVPT)]=env_cr3?(提示: 结合系统自映射机制)

根据自映射的机制，pgdir中的第PDX(UVPT)项中存储的内容即应当为pgdir的物理的地址。设置之后即可通过pgdir找到其页目录的物理地址，方便后续建立二级页表的映射关系。

• 谈谈自己对进程中物理地址和虚拟地址的理解

同lab2，我们在建立页表机制、创建env的内存环境时绝大部分用到的都是虚拟地址。只有在向页表中填写页表项时，才需要向其中填入具体的物理地址（在进程初始化时的转化仅需要pa = (va - ULIM)）。对于进程而言，操作系统为其屏蔽了物理地址，其只能通过读写虚拟地址改变计算机的状态。

4. 思考user_data 这个参数的作用。没有这个参数可不可以？为什么？ （如果你能说明哪些应用场景中可能会应用这种设计就更好了。可以举一个实际的库中的例子)

该函数的第一行就将该用户数据指针转化为了进程控制块指针,而这个进程控制块指针是后续得到该进程的页目录的虚拟地址，从而将ELF加载到合适位置的基础。同时，这个进程指针是连接三个重要的加载镜像的函数（load_icode_mapper、load_icode、load_elf）的纽带。因此不可以没有这个参数。

```
struct Env *env = (struct Env *)user_data;
```

暂时还不能理解为什么要把它转换为void*。

5 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？ （提示：1、页面大小是多少；2、回顾lab1中的ELF文件解析，什么时候需要自动填充.bss段）

1.bin的首地址是否BY2PG对齐。

2.bin的剩余的未被复制的空间占据的空间是否小于一个页面空间。

对于这些情况，实现的解决办法是：

对于第一次进入循环：如果offset不为0（bin起始地址没有页面对齐），则需要在页面首地址空出相应的offset位后进行复制，复制的大小取决于需要换页（BY2PG-offset（因为前offset已经为空））或者到达bin的结尾。

对于后续循环：(bin+i-offset)是未被复制的部分的起始地址（已经BY2PG对齐），目标地址即新分配的首地址，复制的大小即为页面大小或到达bin的尾部，函数的实现如下所示：

```
    for (i = 0; i < bin_size; i += BY2PG) {
        /* Hint: You should alloc a new page. */
        r = page_alloc(&p);
        if (r<0) {
            return -E_NO_MEM;
        }
        p->pp_reff+;
        if (i==0){
            bcopy((void*)bin, (void*)page2kva(p)+offset, MIN(BY2PG-offset,
binsize))
        }
        else {
            bcopy((void*)bin+offset-i, (void*)page2kva(p), MIN(BY2PG, bin_size-
i));
        }
        r = page_insert(env->env_pgdir, p, va+i, PTE_V|PTE_R);
        if (r<0){
            return -E_NO_MEM;
        }

    }
```

.bss段即sgsize-binsize的部分，这部分空间需要自动补0。

6 思考上面这一段话，并根据自己在lab2 中的理解，回答：

• 我们这里出现的" 指令位置" 的概念，你认为该概念是针对虚拟空间，还是物理内存所定义的呢?

应当是针对虚拟空间。

• 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?

entry_point对于每个进程应该是一样的。`load_elf`中的`*entry_point = ehdr->e_entry;`语句对entry_point进行赋值。因为entry_point是对于ELF格式的文件而言的，ELF文件严格规定了所有程序的统一入口地址能够使得其作为操作系统加载程序的依据时更加规范严谨。

7. 思考一下，要保存的进程上下文中的env_tf.pc的值应该设置为多少？为什么要这样设置?

应该设置为当前进程在陷入内核态后cp0中epc的值，在MIPS的CP0寄存器中，epc存储当前被打断执行的指令的pc或是下一条pc或者由软件维护的应该从内核态返回后执行的第一条指令的pc到epc中，符合进程切换的条件。下一次运行这一进程时也应该从epc处开始执行。

8 思考TIMESTACK 的含义，并找出相关语句与证明来回答以下关于TIMESTACK 的问题：

• 请给出一个你认为合适的TIMESTACK 的定义

TIMESTACK是发生时钟中断后保存进程上下文环境的栈区。

• 请为你的定义在实验中找出合适的代码段作为证据(请对代码段进行分析)

在`env_destroy`函数中：

```
bcopy((void *)KERNEL_SP - sizeof(struct Trapframe),
      (void *)TIMESTACK - sizeof(struct Trapframe),
      sizeof(struct Trapframe));
```

可见我们的实验中以TIMESTACK - sizeof(struct Trapframe)为基址保存进程的状态。

在发生时钟中断时， 我们会调用lib/genex.S里面的handle_int函数，在handle_int中，会调用include/stackframe.h中的SAVE_ALL、中定义的宏get_sp，该宏函数在判断cp0_cause寄存器的值后，如果为中断，则设置栈指针为0x82000000，否则设置为KERNEL_SP

```
154 .macro get_sp
155     mfc0    k1, CP0_CAUSE
156     andi    k1, 0x107C
157     xori    k1, 0x1000
158     bnez    k1, 1f      //是否为中断异常，是则不跳转
159     nop
160     li  sp, 0x82000000 //TIMESTACK
161     j   2f
162     nop
163 1:
164     bltz    sp, 2f
165     nop
166     lw  sp, KERNEL_SP
167     nop
168
169 2:  nop
```

• 思考TIMESTACK 和第18 行的KERNEL_SP 的含义有何不同

TIMESTACK是在发生时钟中断后CPU内部状态的存储栈。

KERNEL_SP是在发生系统调用等exception或trap时CPU内部状态的存储栈。

9. 阅读 kclock_asm.S 文件并说出每行汇编代码的作用

```
#include <asm/regdef.h>
#include <asm/cp0regdef.h>
#include <asm/asm.h>
#include <kclock.h>
.macro  setup_c0_status set clr #宏函数，设定CP0的初始状态
        .set    push
        mfc0    t0, CP0_STATUS
        or      t0, \set|\clr
        xor     t0, \clr
        mtc0    t0, CP0_STATUS
        .set    pop
.endm

        .text
LEAF(set_timer)

        li t0, 0x01
        sb t0, 0xb5000100  #0xb5000000是时钟设备的基地址，在偏移量为0x100（此处写值将改变时钟的频率）处写入1表示时钟频率为1s.
        sw      sp, KERNEL_SP #将栈指针保存在内核栈空间保证正常产生时钟中断
setup_c0_status STATUS_CU0|0x1001 0 #调用宏函数设置CP0的状态
        jr ra #函数结束返回

        nop
END(set_timer)
```

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

本次实验设置了两个就绪队列。存储一个count值记录当前正在运行的进程剩余的时间片数量，每经过一次时钟中断自减一（表明该进程消耗了一个单位的运行时间），当count自减到0时，意味着当前运行的进程已经用光了自己的时间片，需要进行进程切换：

1.选择一个非空队列，第一个元素（进程）出队，设置count值为该进程的时间片数量（在本实验中即为该进程的优先级）。

2.将之前已经跑外的进程进入到另一个队列的队尾。

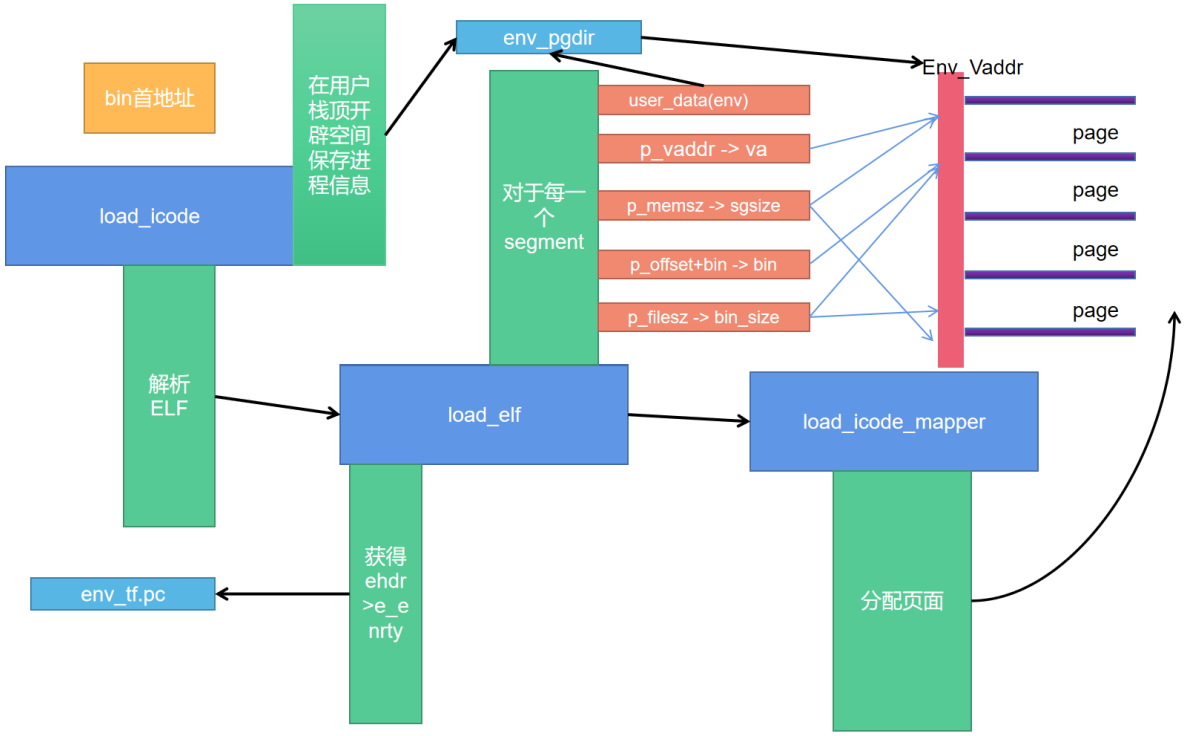
3.执行当前进程。

当然，同时需要注意，当所选择的进程为阻塞状态或已经结束，则重复上述过程直到找到一个就绪进程。且已经结束的进程不需要加入就绪队列了。

2.实验难点图示

1. 装载程序镜像到内存：

三个函数的调用关系，及其向PCB及内存中写入相关信息的过程。



3.实验总结

lab3总体难度和lab2差不多（都挺难的），在课下部分，我们重点实现了创建PCB、以及其编号、状态、页表、进程上下文等参数的设置、加载镜像到内存空间、设定时钟中断和实现简单的调度算法来根据时间片调度进程。在课上测试中，第一次测试重点考察了由fork关系确定的进程树的一系列操作，第二次则重点考察了对时间片调度算法的熟悉程度并基于此简单的扩展功能。（其实个人感觉中间难度较大的进程上下文切换和相关汇编代码、加载二进制镜像等内容缺少了考察~ ）虽然期间debug的过程十分痛苦，但在真正理解了相关原理时还是对自己能够理解MOS的进程相关的知识感到比较高兴的。

4.指导书反馈

指导书：Note 4.2.2 用户态和内核态的概念大家已经了解了，内核态即计算机系统的特权态，用户态就是非特权态。mips 汇编中使用一些特权指令如mtc0、mfc0、syscall等都会陷入特权态（内核态）。mtc0 mfc0应该不会陷入内核态吧。

指导书中给出的异常分发代码中的12行`nop`给大写了。。。

指导书中关于设置进程控制块的部分中CP0的状态被写作了CPU的状态。

关于load_icon_mapper的一点吐槽：

在提示中已经给好了for循环，个人感觉如果硬要使用这个循环，则实现难度会增大，可读性也会下降：

```
    for (i = 0; i < bin_size; i += BY2PG) {
        /* Hint: You should alloc a new page. */
        r = page_alloc(&p);
        if (r<0) {
            return -E_NO_MEM;
        }
        p->pp_reff+;
        if (i==0){
```

```
            bcopy((void*)bin, (void*)page2kva(p)+offset, MIN(BY2PG-offset,
binsize))
        }
        else {
            bcopy((void*)bin+offset-i, (void*)page2kva(p), MIN(BY2PG, bin_size-
i));
        }
        r = page_insert(env->env_pgdir, p, va+i, PTE_V|PTE_R);
        if (r<0){
            return -E_NO_MEM;
        }
    }
```

这样的代码尤其在bcopy的复制范围上稍不留意就会出错。不如根据实际情况分类讨论简单一些。因此最好的选择是：多给出点注释而不要给好循环的代码。

5.残留难点

1.在env_setup_vm中，强掉了设置页目录中每个页表项的权限位：

```
/* UVPT map the env's own page table, with read-only permission. */
e->env_pgdir[PDX(UVPT)] = e->env_cr3 | PTE_V;
return 0;
```

那为什么在之前将boot_pgdir作为模板复制时，和全部清0时不需要设置权限位呢?

2.env_destroy函数中，在其中有一段将TIMESTACK中的内容复制到KERNELSTACK中。对此自己有些不太理解这样做的必要性。目前猜测可能后来env_destroy要作为MOS的系统调用的一部分，而此时需要将时钟中断栈中的内容拷贝到系统调用的异常栈中。因此先在此处存异看看能否在后面的lab中得到合理的解释。

3.在内存布局中，有两个PageTable：VPT和UVPT，在我们的实验中是否所有的进程块在内核态共享同样的VPT，而在用户态下单独使用自己的UVPT？（在env_vm_init的提示中，只要求我们对UVPT页表进行了设置，因此在这里认为VPT是所有进程块共享的）。