Lab3实验报告

思考题

Thinking 3.1

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)

因为在后续取进程控制块的时候，我们使用LIST_FIRST宏，因此，逆序插入可以保证在率先取到的是按顺序获得的envs数组的进程控制块。

Thinking 3.2

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?

生成id的运算：

Step1.计算当前进程的索引：e - envs

Step2.返回或运算后的值：1<<11|idx.这样由于进程索引是唯一确定的，因此返回的进程id也是唯一确定的。

• 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?

判断e->env_id != envid：

因为上一步通过索引取envs数组中的第“id”个进程块e时，去掉了envid的前22位，而只取了后10位。因此，e->env_id != envid这一步确定进程e的id确实是传入的envid。后10位在生成的时候只与进程页的物理位置有关，`idx = e - envs`。而前面22位才是保证进程unique的关键（由调用次数决定，可以保证unique）。要保证一个进程的id号完全对应，看后十位不够，还得对比前22位也确实是一样的。如果没有这步判断会造成错误：可能输入的id并不是进程id号，而仅仅是进程的物理位置与另一个进程相同。

Thinking 3.3

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)

复制的那一部分boot_pgdir是内核的ENVS区域、PAGES区域，以及用户VPT区域，这些虚拟地址上，我们已经建立了二级页表系统。直接拿来作为模板，可以省略建立二级页表虚存系统的时间。

• 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?

UTOP 是0x7f40_0000，是用户能够操控的地址空间的最大值，之上的所有映射对于任意一个地址空间都是一样的。

ULIM 是0x8000_0000，是操作系统分配给用户地址空间的最大值。

UTOP和ULIM之间的区域是用户进程结构体、页表结构体、用户可用的页表，所以对于不同的进程，这一段，和上面2G~4G段都是一样的。而UTOP以下的内容是不一样的。UTOP到ULIM这段空间用户只读，是在内核态的时候留出来给用户进程查看其他进程信息的，用户在此处进行读取不会陷入异常。

• 在step4 中我们为什么要让pgdir[PDX(UVPT)]=env_cr3?(提示: 结合系统自映射机制)

UVPT需要自映射到他在进程的页目录中的对应的页目录地址。这样，UVPT的虚拟地址就和物理地址联系起来，能够方便地进行转换。

• 谈谈自己对进程中物理地址和虚拟地址的理解

创建进程时，操作系统会为该进程分配一个4GB大小的虚拟进程地址空间，通过页目录和页表这样的数据结构。但是实际映射的物理空间比这小。每个进程只能访问自己的虚拟地址空间中的数据，无法访问别的进程中的数据，实现了数据的保护。

Thinking 3.4

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）

User_data实际上是一个函数指针。没有这个参数不行。他确定了这个map函数具体是调用了哪个函数进行映射。

在实际应用场景中，函数指针有很广泛的应用。因为在代码中函数是写死的，而指针是灵活的，可以根据输入场景的需要动态地选择合适的函数。比如在一些语言自带的sort排序函数中，可以根据函数指针指向的函数不同，动态地选择合适的排序算法——二分或者桶排序或者快排之类。或者在神经网络、图像处理领域，在变换模块，函数指针就可以动态地从DCT变换族中选择合适的变换核矩阵，以达到最佳的频率域的变换效果。

Thinking 3.5

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?

可能会面临va存在偏移，此时要先对这部分偏移量进行映射和复制，就是从偏移量开始到下一页之间。然后是整页的复制。最后是不满一个segment的部分，要用0补全。

Thinking 3.6

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?

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?

一样的。因为elf文件都被加载到了固定的位置，因此每个进程的运行代码的入口点是相同的。查阅了ELF文件的格式之后得知，ELF文件的开头有一些elf格式识别段和header之类，之后才是代码段。因此这种统一我认为来自于ELF文件格式的统一。

Thinking 3.7

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curenv->env_tf.cp0_epc。因为EPC寄存器存放的是异常发生时，系统正在执行的指令。

Thinking 3.8

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是保存现场的栈。在时钟中断后，将当前进程的上下文存放在TIMESTACK的栈顶指针处。

• 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)

```
.macro get_sp
       mfc0       k1, CP0_CAUSE
       andi k1, 0x107C
       xori k1, 0x1000
       bnez       k1, 1f
       nop
       li     sp, 0x82000000
       j      2f
       nop
1:
       bltz sp, 2f
       nop
       lw    sp, KERNEL_SP
       nop
2:    nop
.endm
```

这段代码在SAVE_ALL中，顾名思义是保存当前进程的上下文所有寄存器。阅读代码可知，如果CP0_CAUSE寄存器的异常code是零，即当前是中断导致的异常，那么获取的sp值就是0x8200_0000，此处这个0x8200_0000就是TIMESTACK的位置。这里也就是把sp栈指针指向了栈顶。如果异常code非零，则指向KERNEL_SP。

以及这段代码：

```
struct Trapframe *old = (struct Trapframe *)(TIMESTACK - sizeof(struct Trapframe));
```

把栈顶的一系列寄存器的值取出赋值给old。

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

通过上题的分析，我认为TIMESTACK是时钟中断后的上下文存放的栈，而KERNEL_SP是非时钟中断产生的异常时用到的栈指针。

Thinking 3.9

阅读 kclock_asm.S 文件并说出每行汇编代码的作用

```
.macro    setup_c0_status set clr
       .set  push                      // 存所有的settings
       mfc0       t0, CP0_STATUS    // 状态寄存器，cp0-12
       or    t0, \set|\clr             // 把set和clr或给t0
       xor  t0, \clr                   // 把clr异或给t0
       mtc0      t0, CP0_STATUS    // 把新的t0给CP0_STATUS
       .set  pop                      // 取出所有的settings
.endm

       .text
LEAF(set_timer)
       li t0, 0x01                 // 立即数赋值
       sb t0, 0xb5000100           // 将0x01存入这个地址。
       // 0xb5000000 是模拟器(gxemul) 映射实时钟的位置，偏移量为0x100 表示来设置实时钟中断的频率,1 则表示1 秒钟中断1次，如果写入0，表示关闭实时钟
       sw   sp, KERNEL_SP          // 栈指针指向KERNEL_SP的位置
setup_c0_status STATUS_CU0|0x1001 0   // 宏函数，传值set和clr
       jr ra
       nop

END(set_timer)
```

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。

进程切换主要依赖于函数sched_yield()和数据结构env_sched_list[2]。

首先，env_create创建进程，加入到空闲链表env_sched_list[0]中。每次时间片用完需要切换下一个进程时，从当前调度队列中拿出一项可调度的，如果没有就换一个链表寻找，拿到要切换到的进程后，将时间片计数值清零，表示当前进程已使用的时间片个数。然后用env_run()切换进程。

函数梳理

lab3-1

PART 1 分配新进程结构体

初始化进程空闲链表

```
void env_init(void) {}
```

- 这是 `envs_free_list` 初始化的函数。将**所有的**进程状态置 FREE，代表尚未被使用，并且**逆序**塞进 `envs_free_list`。
- 逆序了之后实际上在`envs_free_list`中，。`envs[i]`是顺序递增的。
- 逆序的原因：在取用的时候是使用`LIST_FIRST`宏来取的。
- `envs+i` 等价于 `&envs[i]`。
- 给envs进程数组开了NENV个元素，原因见pmap.c：`envs = (struct Env *)alloc(NENV * sizeof(struct Env), BY2PG, 1);`
- **NENV是2的10次方。所有进程控制块存放的虚拟空间是0x7f40_0000~0x7f80_0000。**
- **除此之外每个进程应该有一个自己的页目录，装的是这个进程对应的页表和物理页之类的。也有一个自己的栈，存放的是程序代码。**

开辟新的进程控制块

```
int env_alloc(struct Env** new, u_int parent_id){}
```

本函数（开辟新的进程控制块）步骤：

1. 从free_list取一个新的进程控制块：如果free_list已经为空，就返回`-E_NO_FREE_ENV`。否则从LIST_FIRST取一个进程控制块e。
2. 然后给这个进程控制块分配虚拟空间。`env_setup_vm(e)`
3. 初始化进程控制块的一些参数。
4. 然后正式将这个进程控制块从free_list里移出，并且赋值给new，传递出去。

PART 2 设置进程控制块

给进程分配虚拟空间

```
static int env_setup_vm(struct Env* e) {}
```

本函数步骤：

1. alloc一个页p，作为页目录。
2. 将页目录的前`PDX(UTOP)`项清空置零 。
3. 将内核页目录拷贝到进程页目录。 ○ 因为根据./include/mmu.h里面的布局来说，我们其实就是2G/2G模式，用户态占用2G，内核态占用2G。 ○ 对于所有的进程，他们的页目录在UTOP以上地址的内容（除了UVPT）储存内容应该是相同的 — — ○ 上方2G虚拟地址与物理地址对应（只差高位1），这部分由内核管理，对于每个进程来说都一样。所以初始化进程的时候要把上方2G虚存这部分拷贝。**因此，在用户进程开启后，访问内核地址不需要切换CR3寄存器。而是可以直接在进程中访问内核地址--》因为我们将内核页目录拷贝到了进程页目录中。** ○ 然而对于下方2G虚存，下列这段也被映射到内核中。（或许应该称为映射到进程中？但我认为其中一个进程占据了内核那么他就是临时内核）。

```
        ULIM     -----> +----------------------------+------------0x8000 0000----
---
 o                      |         User VPT           |     PDMAP
 /|\
 o      UVPT     -----> +----------------------------+------------0x7fc0 0000
 |
 o                      |         PAGES              |     PDMAP
|
 o      UPAGES   -----> +----------------------------+------------0x7f80 0000
 |
 o                      |         ENVS               |     PDMAP
|
 o  UTOP,UENVS   -----> +----------------------------+------------0x7f40 0000
 |
```

为什么要将这部分也映射给内核呢?

- 这部分是什么？ 是ENVS是envs进程数组，PAGES存的是页表结构体，以及不知道到底是什么的User VPT。
- 这部分什么用？ ENVS这块，一个4M的用户进程虚拟区，可能是用来给内核一个获得其他进程状态、信息的入口。因此，对于内核来说这部分应该是只读模式。

6. 将进程页目录的虚拟地址`pgdir`和物理地址(可以由PADDR宏，或者page2pa宏得到)都赋值给进程结构体e。
7. 将进程页目录的表示VPTx系统虚拟页目录和UVPT用户虚拟页目录的那4M空间的项都赋值为进程自己的页目录的物理地址，以及加不同的有效位。

进程id相关函数1：进程id的生成

```
u_int mkenvid(struct Env* e) {}
```

本函数步骤：

1. 计算进程索引：第 index 个进程
2. 生成id：(1 << 11) | index

> **Thinking：为什么左移11？** 我觉得是因为，后面有个函数`envid2env()`，在计算envid的索引时用的宏`ENVX`取envid的后十位。也就是说，我觉得可能envid的后十位才表示他的id，如果不左移11位的话，第十位是1，后几位是index，而我们的index不需要前面的1，所以要用左移将它除去。如果没有这个1的话，就无法左移获得一个10位（11位）的数。

进程id相关函数2：根据进程id获得对应进程控制块

```
int envid2env(u_int envid, struct Env** penv, int checkperm) {}
```

本函数步骤：

1. 如果envid是0，返回当前进程控制块。
2. 获得envid对应的进程索引：`ENVX(envid)`，然后在envs数组中找到对应的元素给e。

> **Thinking： 为什么要判断`e->env_id != envid`？** 因为上一步通过索引取envs数组中的第“id”个进程块e时，去掉了envid的前22位，而只取了后10位。因此，e->env_id != envid这一步确定进程e的id确实是传入的envid。后10位在生成的时候只与进程页的物理位置有关，`idx = e - envs`。而前面22位才是保证进程unique的关键（由调用次数决定，可以保证unique）。要保证一个进程的id号完全对应，看后十位不够，还得对比前22位也确实是一样的。如果没有这步判断会造成错误：可能输入的id并不是进程id号，而仅仅是进程的物理位置与另一个进程相同。

4. check一下当前进程`curenv`是不是有合法perm去操作这个特定进程（要么e是当前进程本身`e != curenv`，要么e是它的直接子进程`e->env_parent_id != curenv->env_id`）。

PART 3 加载二进制镜像

为进程分配栈空间 容纳程序代码

```
static void
load_icode(struct Env* e, u_char* binary, u_int size) {}
```

本函数主要步骤：

1. 申请一个物理页p，用函数`page_insert()`将物理页`p`和虚拟地址`USTACKTOP - BY2PG`联系起来，初始化一个进程的栈，表示常规用户栈normal user stack里一个4KB的一页的空间被使用。
2. 使用`load_elf()`函数将每个segment都加载到正确的地方
3. 将PC寄存器移动到代码入口地址，即`entry_point`

加载elf

将每个segment都加载到正确的地方

```
int load_elf(u_char *binary, int size, u_long *entry_point, void *user_data,
       int (*map)(u_long va, u_int32_t sgsize,
            u_char *bin, u_int32_t bin_size, void *user_data)){}
```

本函数主要内容： 本函数的主要功能是在while循环中实现的。主要有两步：

1. 加载elf文件中的内容到内存。 ○ 这一步中，先判断这个phdr是不是loadable的。如果可被加载，再加载。 ○ 然后`ptr_ph_table`递增一个entry_size的大小。 ○ phdr指向下一个元素。
2. 内存富余空间填零。 **（Q:？？？ where)**
3. user data
4. 由map函数，即`load_icode_mapper()`函数实现。`void* user_data`这个参数是一个函数指针。函数指针：可以给不同的需要加载的内容动态选择合适的mapper函数。（OSLAB中只有一个mapper函数，其实可以有多个） mapper函数是把UTEXT的部分映射到新开的page里。

PART 4 创建一个进程

进程创建

主要是这个函数：

```
void env_create_priority(u_char* binary, int size, int priority) {}
```

本函数主要创建一个进程，步骤：

1. 分配一个新的Env结构体。
2. 设置进程控制块，给进程分配虚拟空间。 以上两步在`env_alloc(&e, 0)`函数中完成。

> **Thinking: 这里为什么`env_alloc`函数的第二个参数是0?** 这是一种默认做法。

3. 将二进制代码载入到对应地址空间。 `load_icode(e, binary, size);`完成。

运行进程

```
void env_run(struct Env* e) {}
```

本函数主要负责进程的切换，步骤：

1. 保存当前进程的寄存器到 env 的 tf 。设置pc。 Trapframe： 捕获当前进程的寄存器状态，这个结构体其实就是所有的寄存器。在本实验里的寄存器状态保存的地方是TIMESTACK（时钟栈0x82000000）区域，所以说指针`*old`就指向`(TIMESTACK - sizeof(struct Trapframe));`这意思在栈顶开一个tf大小的空间。然后将上下文保存到当前进程的 `curenv->env_tf` 中。

> **Thinking: 关于`li sp, 0x82000000`** 这是在stackframe.h的一句汇编。一个get_sp的宏。我们本次做的都是时钟中断，所以说，存储上下文寄存器的栈指针sp指向的是时钟栈区TIMESTACK。

2. 恢复要启动的进程。 ○ 将当前进程curenv设置为新进程e。 ○ 用`lcontext()`汇编函数切换地址：

将`mCONTEXT`(页目录首地址)存到a0。并跳转到ra寄存器 ○ env_pop_tf：把 env 里的 tf 放到寄存器里。 ○ 把当前进程的id后5位清空。

---

lab3-2

进程调度

进程调度主要是sched_ yield函数完成的。 调度算法是时间片轮转，在我们的实验中，优先级并不是传统理解中的优先级，而是时间片长度。

- 在什么时候会调用sched_ yield函数?

1. 在env_destroy 函数中。这个函数的主要职责就是free一个进程并且调一个进程来运行。但目前为止还没有调用过这个函数。只是声明且定义了它。
2. 时钟中断。

时钟中断的全过程

- 什么时候会开启时钟中断?

1. 进入异常。

```
. = 0x80000080;
 .except_vec3 : {
 *(.text.exc_vec3)
 }
```

首先是进入异常处理程序的入口，一旦CPU发生异常，就自动跳转到0x8000_0080，这里放的是.text.exc_vec3代码。

2. 选择相应中断处理程序

```
NESTED(except_vec3, 0, sp)
```

> 关于.set noat 之类 .set是汇编代码的一些设置，比如at，就是开启扩展指令，前面加个no就是不开启。其他命令同理。别的函数里还有一个.set push，是把所有设置存进栈里，相应的还有.set pop

`mfc0 k1,CP0_CAUSE`

这个汇编函数在设置了之后，首先将CP0_CAUSE给了k1寄存器。 `a k0,exception_handlers`

然后将异常句柄数组（这个数组的初始化在traps.c里，用set_except_vector这个函数初始化的）的首地址给了k0。 `andi k1,0x7c`

将k1中的，即CP0的cause寄存器中的异常码区段截出来，就是异常编号。因为c的二进制是1100，也就是在异常码之后还有2个二进制的0，所以相当于将异常编号左移2位，也就是4的整倍数对齐。由于数组以字对齐，也就是异常码+2'b00可以作为异常句柄数组的索引。 `addu k0,k1`

如上所述，首地址+偏移，得到的是异常码的句柄所在的项的位置。 `lw k0,(k0)`

汇编语法不太懂，大概就是把找到的异常处理句柄赋值给k0。 `jr k0`

跳转到对应的异常处理程序。我们在实验中暂时只实现了handle_int这个句柄。

3. 保存现场

```
NESTED(handle_int, TF_SIZE, sp)
.set  noat

//1: j 1b
nop

SAVE_ALL      // 保存栈帧，把所有的寄存器给保存到栈中
CLI
.set  at
mfc0  t0, CP0_CAUSE
mfc0  t2, CP0_STATUS
and t0, t2

andi  t1, t0, STATUSF_IP4
bnez  t1, timer_irq       // 判断是否支持中断。如果支持中断，则调用timer_irq

nop
END(handle_int)
```

本段汇编函数： 分别将CP0_CAUSE和CP0_STATUS存入t0和t2两个寄存器中，然后`and`，存入t0，然后就可以获得具体中断号（ppt里看的，并不知道具体怎么操作的）。 然后判断是否支持中断。如果支持中断，则调用timer_irq。

4. 调用timer_irq 函数里只有一句跳转到sched_yield和返回（ret_from_exception）

sched_yield 基于时间片的进程调度(本次实验难点)

```
void sched_yield(void)
{
    // 记录当前进程已经使用的时间片数目
    static int count = 0;
    // t 记录进程链表序号，0或1
    static int t = 0;
    // 当前进程已使用时间片+1
    count++;
```

```
    /*
     * 切换进程的条件
     * 1. 当前进程时NULL，这种情况只发生在运行第一个进程的时候。
     * 此时还没env_run()，而这个函数负责将curenv设置为e。
     * 2. 当前进程的时间片已经用完了
    */
    if(curenv == NULL || count >= curenv->env_pri) {
        // 如果不是第一次运行进程，则要将当前进程添加到另一个待调度队列中以便下一次调度。
        // 为什么是insert_tail呢，我觉得和链表每个进程能被公平调度有关。
        // 取用的时候是list_first，放回的时候就塞到队尾。
        if(curenv != NULL) {
            LIST_INSERT_HEAD(&env_sched_list[1 - t], curenv, env_sched_link);
        }
        // 若两个链表都没找到合适进程，就返回。这个flag用于标志已经遍历过的链表。
        int flag = 0;
        while(1) {
            struct Env *e = LIST_FIRST(&env_sched_list[t]);
            // 若当前进程列表没有可以调度的进程，就换一个链表。
            // 同时flag+1，表示已经遍历了其中一个链表。
            if(e == NULL) {
                if(flag == 0) flag = 1;
                else return; // 若两个链表都没有找到，就直接return返回。
                t = 1 - t; // 换一个链表找，这里用t^=1也可，并且位运算速度比加减法快。
                continue;
            }
            // 若找到一个可以执行的进程
            if(e->env_status == ENV_RUNNABLE){
                // 从待调度队列中移出
                LIST_REMOVE(e,env_sched_link);
                // 初始化已使用的时间片个数
                count = 0;
                // 运行找到的这个进程e，当成当前进程
                env_run(e);
                break;
            }
        }
    } else {
    // 如果剩余时间片不为0，将剩余时间片-1，并且env_run接着执行当前进程。
        env_run(curenv);
    }
}
```

在网上有好多种写法，但主要意思都是一样的。 进程调度的主要思路是：进入这个函数 -》当前进程已用时间片+1 -》若当前进程用完时间片-》找一个新的进程并运行-》若当前进程没有用完时间片-》继续运行当前进程。 详细来说见注释。 此外，

- 双队列减少进程间的不公平。
- 两个队列都存储RUNNABLE的进程，减少遍历时间。
- 需要在priority设置完之后再把env插入到第一个队列中。而不是设置完status之后插入，因为此时priority默认为0，也就是时间片还是0就被加入待调用的进程链表中。（对ppt抱有异议）

体会与感想

lab3主要是进程的调度。我觉得难度和lab2平分秋色，都很难。 lab3-1和lab3-2的代码量差距太大。lab3-1的内容有点多，各个函数也不好写。而lab3-2只有一个sched_yield函数不好写，别的理解起来也不困难。

指导书反馈

建议各个部分的工作量差别不要太大。可以适当减少lab3-1的部分，增加lab3-2。因为lab3-2的量也比较少，所以在理解深度上可能不如Lab3-1。 这个lab的指导书我觉得写得还可，在需要我们重点理解的部分写的还行。但在其他同样需要我们理解的部分，可能不是理解的重点，但是就没写。这会导致我们理解错误而且能够自圆其说，后来可能在某个时候突然发现是错的，这是最可怕的。 比如env_alloc的时候第二个参数parent_id为什么一直是0？本来我觉得给的代码不会出错，所以想了个可能的原因为当前进程的id是0，所以由当前进程产生的其他进程的parent_id也应该是0。而且在envid2env这个函数的时候如果envid是0就返回当前进程，印证了我的观点，然而这是错的。parent_id在刚alloc的时候设为0只是一个默认的做法。 所以，我也不知道这些东西是写在指导书里更好还是让我们自己理解更好。我认为指导书还可以多写一些辅助和引导性的东西。就目前来看，代码量和指导书的文字描述比例不太好。

残留问题

我不知道为什么进程调度那边的算法要这样写，我想知道实际上真实的进程时间片是如何划分的。进程的任务完成了之后如何退出什么的。 以及代码中有的那些free，destroy函数，都在什么时候会被调用？我觉得我现在对于进程调度的前因后果还没有一个完整的认识。