**Lab3实验报告**

1. 思考题

3.1

因为在进程创建时，要保证第一个被选中的进程是envs[0]，所以在从头插入空闲链表时，要倒序插入。

3.2

·使用一个静态变量next\_env\_id记录进程编号，每次加一。使用进程在envs中的位置确定进程。将next\_env\_id移位到高位，将进程位置idx保存在低位。

·如果e->env\_id != envid，表明进程已经不存在了，所以需要进行判断。

3.3

·因为所有进程的内核区域的映射都是相同的，所以直接复制就好。

·ULIM之上的部分是kseg0，不需要经过MMU地址转换。UTOP到ULIM是3个PDMAP大小，存放了pages数组、envs数组和用户页表空间，本质上用户进程也是不能操作的。

·env\_cr3是用户进程页目录的物理地址，这个赋值是让用户进程页目录的基地址等于用户进程页目录的物理地址，完成了用户页目录的初始化。

·进程中，仍然需要虚拟地址到物理地址的转换。进程在没有陷入内核态时，只能操作用户空间的页表，即低2G的虚地址，且不包含UTOP到ULIM。在进程切换时，需要更新映射关系。

3.4

user\_data是为了传入进程的PCB，从而可以更新页表映射。没有这个参数就无法找到进程了，从而没有办法更新进程的页表映射，地址转换会无法完成。

3.5

会出现4种不同情况。

·首先，开头可能不是对齐的，这就需要先复制不对齐的部分。但是有可能内容很短，没到一页就结束了，需要将文件长度与剩余长度进行比较，找到较小的值，按照这个值进行复制。

·然后，在末尾可能也是不对齐的。需要将文件剩余长度与页面大小进行比较，找到较小的值，按照这个值进行复制。

3.6

·应该是虚拟空间。

·应该是一样的。因为每个进程都有一个自己独有的虚拟空间，所以虽然entry\_point的值是一样的，但是所对应的物理地址却是不同的。

3.7

env\_tf.pc应该设置为old的cp0\_epc的值。因为在进程切换时，需要将下一条指令的地址存在EPC寄存器中，然后在切换回来时恢复PC寄存器的值，从而继续执行。

3.8

·TIMESTACK是保存切换进程的上下文的一个栈。

·而KERNEL\_SP是内核栈，虽然也是保存了上下文，但完成的是系统调用相关的功能。

·在trap.h文件中，用一些列的define定义了trapfrmae的布局。从0号寄存器到31号寄存器，还包括CP0\_ STATUS，HI，LO，CP0\_ BADVADDR，CP0\_ CAUSE，EPC，PC等特殊寄存器，可以说是把几乎所有的寄存器都预留了位置。

·在stackframe.h文件中。get\_sp函数在CP0\_CAUSE的第12位为1，2-6位为0时，会将sp设置为0x82000000，即TIMESTACK栈顶。而在SAVE\_ALL函数中，就先调用了get\_sp函数，然后对当前的寄存器进行保存。可以说，真正负责写TIMESTACK的是SAVE\_ALL函数。但是，值得注意的是，唯独缺少了对PC寄存器的保存。这也意味着需要自己在代码中处理PC寄存器的值，所以需要设置curenv->env\_tf.pc = old->cp0\_epc;

·在env\_ams.S文件中，env\_pop\_tf函数恢复栈中的值。它通过传入的Trapframe的地址，从而定位TIMESTACK。然后依次恢复CP0\_STATUS，HI，LO，EPC和0-31号寄存器的值。最后跳转到栈中保存的PC。从而完成从函数切换中的返回。

3.9

我在调试时发现，进程调度顺序并非想象中的112112轮流切换，而是112211交替切换。经过仔细思考后，我发现切换时按照这个算法操作会导致每切换一轮后，就绪队列就颠倒一次，所以造成了这样的结果。

如果只有两个进程，那么就不需要就绪队列了，切换时直接选择另一个进程就好，这样直接保证按照112112调度。

1. 实验难点分析

**难点1： 理解env.h**

宏定义：

NENV： 进程个数

ENVX(envid)： 进程在 envs 中的序号

GET\_ENV\_ASID(envid)： 进程 TLB 的索引

PCB定义：

struct Trapframe env\_tf; // 保存寄存器(上下文环境)

LIST\_ENTRY(Env) env\_link; // 链表遍历变量

u\_int env\_id; // 进程标识符

u\_int env\_parent\_id; // 父进程的 env\_id

u\_int env\_status; // 进程状态(3种取值)

Pde \*env\_pgdir; // 进程页目录的内核虚地址

u\_int env\_cr3; // 进程页目录的物理地址

LIST\_ENTRY(Env) env\_sched\_link; // 用来构造就绪状态进程链表

u\_int env\_pri; // 优先级

**难点2： 填写load\_icode\_mapper**

/\* ELF 加载器提取每个段, 之后加载器调用这个函数将每个段映射到正确的虚地址

bin\_size 是 bin 在文件中的大小, sgsize 是段在内存中的大小

先前条件:

bin 不为空, va 可能不是 4KB 对齐的

结束状态:

成功返回 0, 否则返回负值 \*/

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; // userdata为env的 PCB

struct Page \*p = NULL;

u\_long offset = va - ROUNDDOWN(va, BY2PG); // va不足一个页面大小的部分

int r;

u\_long i;

void\* src;

void\* dst;

size\_t len;

/\*Step 1: 将 bin 中所有的内容加载到内存 \*/

for (i = 0; i < bin\_size; i += BY2PG) {

// 分配一个页面并增加引用计数

r = page\_alloc(&p);

if (r != 0)

return r;

p->pp\_ref++;

// 考虑不同的复制情况

if (i == 0) { // [bin, BY2PG-offset] --> page\_0[offset, BY2PG]

src = bin;

dst = (u\_char \*)(page2kva(p) + offset);

len = (size\_t)MIN(BY2PG - offset, bin\_size);

}

else { // [bin-offset, page\_end] --> page\_i[0, BY2PG]

src = bin + i - offset;

dst = (u\_char \*)page2kva(p);

len = (size\_t)MIN(BY2PG, bin\_size - i);

}

bcopy(src, dst, len);

// 更新页映射

r = page\_insert(env->env\_pgdir, p, va + i, PTE\_V | PTE\_R);

if (r != 0)

return r;

}

/\*Step 2: 如果 bin\_size < sgsize, 分配页面达到 sgsize, i 是现在 bin\_size 的值 \*/

while (i < sgsize) {

r = page\_alloc(&p);

if (r != 0)

return r;

p->pp\_ref++;

r = page\_insert(env->env\_pgdir, p, va + i, PTE\_V | PTE\_R);

if (r != 0)

return r;

i += BY2PG;

}

return 0;

}

**难点3： 理解**TIMESTACK

见上面的Thinking 3.8

**难点4： 理解中断处理过程**

在genex.S文件中：

/\* 时钟中断处理函数 \*/

.set noreorder

.align 5

NESTED(handle\_int, TF\_SIZE, sp)

.set noat

SAVE\_ALL

CLI

.set at

mfc0 t0, CP0\_CAUSE

mfc0 t2, CP0\_STATUS

and t0, t2

/\* 判断是不是对应的 4 号中断位引发的中断 \*/

andi t1, t0, STATUSF\_IP4

/\* 如果是, 执行 timer\_irq \*/

bnez t1, timer\_irq

nop

END(handle\_int)

/\* 中断服务函数 \*/

.extern delay

timer\_irq:

j sched\_yield /\* 跳转到调度函数sched\_yield \*/

nop

j ret\_from\_exception

nop

所以一个时钟中断处理过程如下：

·一旦实时钟中断产生, 就会触发 MIPS 中断, 从而 MIPS 将 PC 指向0x80000080, 从而跳转到 .text.exc\_vec3 代码段执行

·对于实时钟引起的中断, 通过 text.exc\_vec3 代码段的分发, 最终会调用 handle\_int 函数来处理实时钟中断

·handle\_int 判断 CP0\_CAUSE 寄存器是不是对应的 4 号中断位引发的中断, 如果是, 则执行中断服务函数 timer\_irq

·函数timer\_irq跳转到调度函数sched\_yield

·调度函数sched\_yield调度进程，从而完成对本次中断的处理。

**难点5：填写sched.c**

在env.c中搜索发现并没有语句将进程加入到就绪链表env\_sched\_list中，所以在函数env\_create\_priority中加入语句：LIST\_INSERT\_HEAD(&env\_sched\_list[0], e, env\_sched\_link); 将进程加入，然后才可以调度。

在每次中断到来时，使用times记录次数，判断是否切换进程，从而进行下一步的操作。如果不需要切换，直接调用env\_run执行。如果需要切换，先对链表进行调整再调用env\_run执行。

需要特别注意的一点是，要判断当前有无进程在执行，如果没有进程在执行，需要直接弹出第一个进程，调用env\_run执行。

1. 体会与感想

在做lab3时，我遇到了不小的困难。首先，在填写load\_icode\_mapper时，一开始不太理解这些参数的作用，后来反复斟酌才搞明白。其次，在填写sched.c时，忽略了一些问题，反复尝试才得以解决。另外，本次实验代码量较大，而且许多是汇编代码，还需要和C代码结合理解，有些不习惯，所以难度较大。不过经过这次实验，我的汇编理解能力也得到了提高。希望在下一个实验中迎接新的挑战。

1. 残留难点

·在stackframe.h中定义了RESTORE\_SOME和RESTORE\_ALL\_AND\_RET函数，那么env\_pop\_tf函数其实可以调用这些函数来完成，而不必写许多代码。

·本次实验只是完成了进程的创建和调度，那么如何创建子进程？如何清除一个进程？期待下一个实验的答案。

·进程切换是如何跟TLB更新联系起来的，mCONTEXT变量是做什么的呢？