LAB03实验报告

THINKING3.1

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？

如果没有判断`e->env_id!=envid`的话，可能实际上这一进程并没有分配id，但是因为未分配时低10位是0而被误判为envs中第一个进程。`envid`还有高位部分，高位不同代表这个进程块被调用过不止一次，仍然不是一个进程。但由于一个进程块同时只能对标一个正在执行的进程，所以若高位不同，代表所查询的`envid`所对应的进程一定不存在，因此返回`-E_BAD_ENV`。若没有这步判断，则会在查询一个不存在的进程id时却能够得到对应的进程，导致程序错误。

THINKING3.2

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`含义是用户可以使用的空间中的最高地址,`ULIM`含义是用户空间的最高地址（再往上就是内核空间了）,它们之间的区域应该是用户没有权限修改的.`UTOP = 0x7f400000`，其含义为用户所能操纵的地址空间的最大值；`ULIM = 0x80000000`，其含义为操作系统分配给用户地址空间的最大值。这一段空间被定义为一个只读片段，属于“内核态”，主要功能在于让用户进程去查看其他进程的信息，用户在此处进行读取不会陷入异常。

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

`UVPT`的含义为User Virtual Page Table，因此这一段需要映射到他的进程在`pgdir`中的页目录地址。所以我们在将这一段空间的虚拟地址转化为物理地址时可以很快找到对应的页目录，`env_cr3`储存的是进程页目录的物理地址，通过这样的赋值完成了页目录的自映射。

• 谈谈自己对进程中物理地址和虚拟地址的理解。

进程只能操作虚拟地址，而实现虚拟地址和物理地址之间的映射由操作系统完成，但是操作系统好像还不能直接修改物理地址中的内容，要通过CPU别人换进换出。

THINKING3.3

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）

在函数`load_icode_mapper`中，被传入的`user_data`被用于这样一个语句中：

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

因此这个所谓的`user_data`实际上在函数中的真正含义就是这个被操作的进程指针。那么我们来一步步追溯这个变量最开始是以什么形式被传入的。

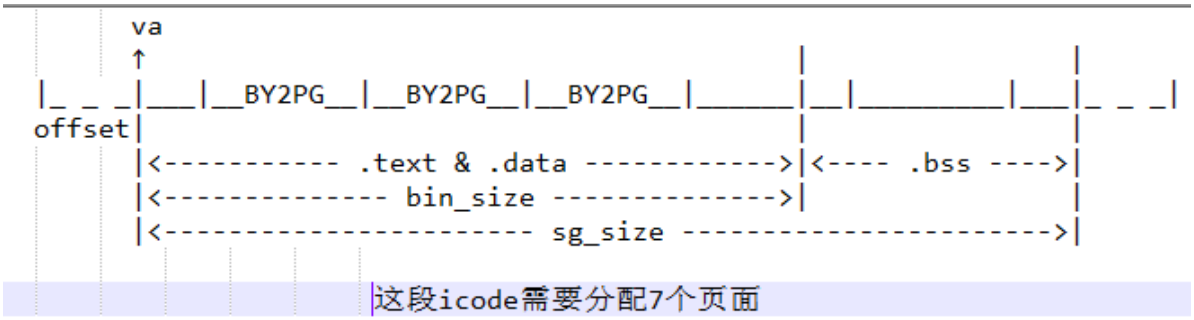
回到`load_elf`函数中，我们可以看到`user_data`从函数本身被传入到调用`load_icode_mapper`中没有改变，那再回到调用`load_elf`的`load_icode`中，我们发现在调用`load_elf`时的语句为：

```
1 r = load_elf(binary, size, &entry_point, e, load_icode_mapper);
```

需要这个参数传递的信息才可以完成加载二进制镜像的作用，如果没有的话得不到页目录的信息。`user_data`这个参数允许我们更好的定制`load_elf`的行为，没有这个参数会影响系统的灵活性。我们在load时，可能会使用多种不同的`mapper`，这些`mapper`可能会需要不同的额外数据来辅助进行映射，`void *`类型的`user_data`是一个最好的传递额外数据的方式，因为向`void *`型指针强制转换可以自动完成，同时`void *`可读性也更好。

THINKING3.4

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？



由上图可得：

`.text & .data`：

- 第一段，需要切除前半部分的`offset`的一段。
- 中间的普通段。
- 最后一段，即前半部分属于`.test & .data`，后半部分属于`.bss`。
- 需要考虑的特殊情况有：
  - 第一段的前半段已经装载过内容，因此不能在这一段进行`alloc`与`insert`操作，从而保留前半段内容。
  - `offset` = 0，此时从最开始的所有端可以当做正常页处理。
  - `.test & .data`与`.bss`被某一个页分割恰好切开，不存在共同占用一个`page`的情况。
  - `.test & .data`这一段的长度极小，即第一个`page`就为最后一个`page`，因此需要同时对两侧的页面分割进行判定与相应操作。

`.bss`：

- 第一段，需要同前半段的`.test & .data`段协同考虑。
- 中间的普通段。
- 最后一段，即前半部分属于`.bss`，后半段在需要复制的内容之外。
- 需要考虑的特殊情况有：
  - 第一段的前半段已经在`.text & .data`段被装载过相关内容，为保证那一段内容不被破坏，在处理`.bss`的这一段时，不能使用`alloc`以及`insert`来进行新的页面插入。**注：在操作正确的情况下，只要两段不是恰好的页面分割，那么一定会出现这种情况！！**
  - `.test & .data`与`.bss`被某一个页分割恰好切开，不存在共同占用一个`page`的情况。
  - `.bss`这一段的长度极小，即第一个`page`就为最后一个`page`。
  - 最后一页被恰好在`page`的交界分开。

THINKING3.5

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?

存储的为虚拟空间地址。

• 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?

`entry_point`对于每个进程来说是相同的，都是从elf文件中读取的，这种值来源于他们都是从ELF文件中的同一个部分进行取值的，elf文件结构的统一决定了该值的统一，不过他们储存的物理地址是不同的。

THINKING3.6

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?

是`env_tf.cp0_epc`，如果要进行进程切换，一定是因中断发生后的处理过程中，进入env_run时如果当前curenv不是null，则当前进程进入中断时的寄存器状态必定在TIMESTACK处存放，（中断处理时会先调用`.\include\stackframe.h`中saveall，而saveall依赖的sp指针值在时钟中断（目前唯一的中断）时正是TIMESTACK）。由于是通过中断进入的，EPC指向的值就是受害指令，如果我们以后要恢复这个进程的运行，当然是从受害指令开始重新执行，因此应设为`env_tf.cp0_epc`。

THINKING3.7

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`是内存中的一块栈空间，用于存储进程的状态。

`TIMESTACK`在env.c中再两处地方被调用：

`env_destroy`：

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

`env_run`：

```
old = (struct Trapframe *)(TIMESTACK - sizeof(struct Trapframe));
                bcopy((void *)old, (void *)&(curenv->env_tf), sizeof(struct
Trapframe));
```

我们不难发现，在我们的操作系统代码中对这个变量的调用均在`bcopy`中进行，并且利用的都是在`TIMESTACK`以下的一段长度为`struct Trapframe`的空间。在`env_destroy`中，将存于`KERNEL_SP`的进程状态复制到`TIMESTACK`处，而在`env_run`中，则是从这一段空间中取出进程状态并转移给当前进程。

在mmu.h中，我们得到`TIMESTACK`的具体值为`0x82000000`，对这段地址的调用进行查找，我们在汇编代码stackframe.S中又找到了一处调用该段地址的地方，即：

```
.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
```

这段代码的功能为获取栈指针的值，若检测到是中断异常则将栈指针置于`TIMESTACK`处，这样在发生中断时我们就能将当前进程的状态存入`TIMESTACK`处，从而进行保存。

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

将栈指针设在`TIMESTACK`还是`KERNEL_SP`与`CP0_CAUSE`有关，在发生中断时将进程的状态保存到`TIMESTACK`中，在发生系统调用时，将进程的状态保存到`KERNEL_SP`中。

THINKING3.8

试找出上述 5 个异常处理函数的具体实现位置。

0号在中断处实现，1号在存储异常处实现，2、3号在TLB异常处实现，8号在syscall中实现。

THINKING3.9

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

```
.text
LEAF(set_timer)
li t0, 0x01
sb t0, 0xb5000100
sw sp, KERNEL_SP
setup_c0_status STATUS_CU0|0x1001 0
jr ra
nop
END(set_timer)
```

时钟一秒产生一次中断，0x50000100为时钟控制地址，接下来设置异常处理栈的值，设置CP0的status寄存器，最后返回函数。

```
timer_irq:

    sb zero, 0xb5000110
1:  j   sched_yield
    nop
    /*li t1, 0xff
    lw    t0, delay
    addu  t0, 1
    sw  t0, delay
    beq t0,t1,1f
    nop*/
```

```
    j   ret_from_exception
    nop

LEAF(do_reserved)
END(do_reserved)
```

把时钟置为零，跳转到sched_yield函数然后跳转到ret_from_exception函数，结束。

THINGKING3.10

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。

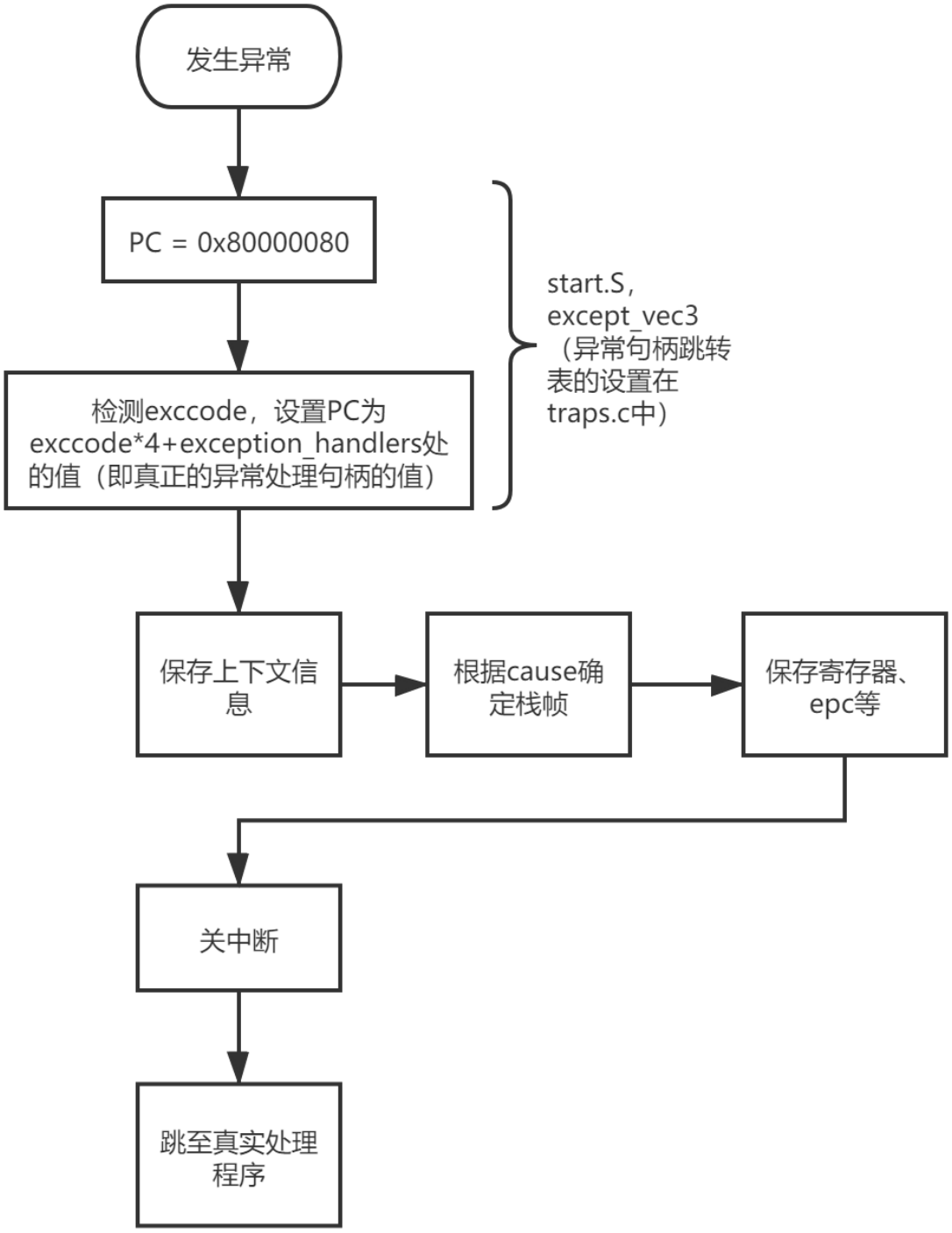
时钟中断发生时，系统在保存上下文之后跳转到sched_yield函数，进行进程的调度。

sched_yield函数首先判断当前进程时间片是否用完，若未用完继续执行当前进程，否则根据调度算法选择一个新进程继续执行，原进程上下文被保存并再次进入就绪队列。

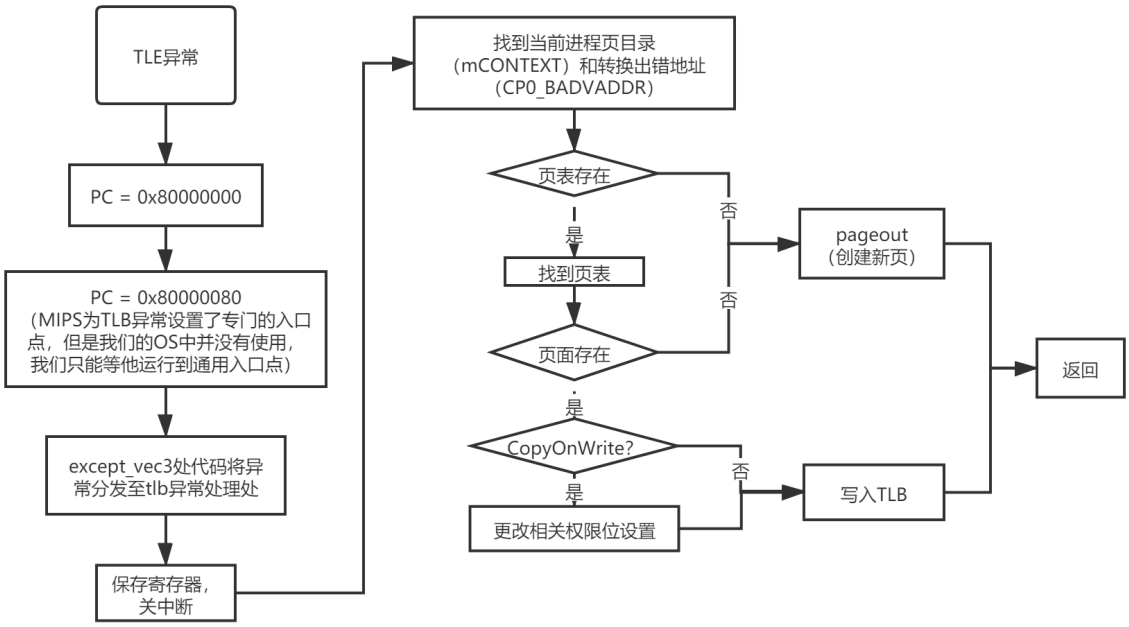
最终，新的进程通过调用env_run函数被执行。

实验难点

中断：


发生异常
PC = 0x80000080
检测exccode，设置PC为exccode*4+exception_handlers处的值（即真正的异常处理句柄的值）
start.S，except_vec3（异常句柄跳转表的设置在traps.c中）
保存上下文信息
根据cause确定栈帧
保存寄存器、epc等
关中断
跳至真实处理程序

TLB流程：



难点

我的调度算法写了好多好多好多好多好多好多好多好多好多好多好多好多好多好多遍，因为我是直接照着hint和指导书写的，下次不希望直接照着hint写可以不写hint，大可不必打哑谜，实在是乐了。

残留疑问

因为21系好像没怎么学习CP0这一块的知识，汇编这一块我看的好费劲，只能大概揣摩其中到底在干什么，对于细节确实是没能理解。