实验思考题

Thinking 3.1

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？

在该判断语句的之前有一句 `e = &envs[ENVX(envid)];` 来通过 envid 来求出 env。可以发现，在这个语句中判断对应关系只使用的是 envid 低 10 位作为索引，也就是进程块的偏移。但实际上，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。

Thinking 3.2

1. **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 = 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，这以上的地址为内核地址，这是用户态程序可以访问地址的界限，更高的内存用户不可读。

   UTOP 到 ULIM 是属于用户并且只读的区域，但是内核这部分映射到了用户区，UENVS 与 envs 均映射到了 envs 对应的物理地址。开启中断后进程访问内核会产生异常来陷入内核，内核开辟这 4M 为用户进程虚拟区，用户读这 4M 空间的内容不会产生异常。

   ULIM 是用户态与内核态地址空间的分界处，UTOP 是为了将内核的一些数据暴露给用户控件而设置的地址空间分界处。 UTOP 到 ULIM 这段空间是操作系统暴露的，对于用户来说，只能读不能写。UTOP 以下是用户进程能够自由读写的空间。

2. **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。

   根据自映射计算公式：

   页表基地址 PT_{base}

   页目录基地址 $PD_{base} = PT_{base} + (PT_{base} >> 10)$

   自映射页目录项

   $PDE_{selfmap} = PT_{base} + (PT_{base} >> 10) + (PT_{base} >> 20) = PD_{base} + (PT_{base} >> 20)$

   对应到具体代码：

   `UVPT` 是页表基地址虚拟地址 PT_{base}。

   `PDX(UVPT)` 取到了虚拟地址的一级页表偏移量，即取到了 `UVPT` 的高 10 位，即 `UVPT >> 22`。

   `e->env_pgdir` 是该进程页目录的内核虚拟地址 PD_{base}。

   `e->env_pgdir[PDX(UVPT)]` 把数组改成指针为 `*(e->env_pgdir + PDX(UVPT) * 4)`，带入宏定义为 `*(e->env_pgdir + (UVPT >> 22) * 4)`，即为 `*(e->env_pgdir + (UVPT >> 20)`。

`e->env_pgdir[PDX(UVPT)]` 就是 $PD_{base} + (PT_{base} >> 20)$，按照自映射，需要在这个位置保存页目录项，所以要填上该进程页目录的物理地址，即 `e->env_cr3` 。

3. **谈谈自己对进程中物理地址和虚拟地址的理解。**

对于进程来说，访问的地址都是虚拟地址，每一个进程都有一个属于该进程的页目录，进程访问到的数据，通过这个页目录以及之前建立的页式虚拟内存管理机制，来访问到储存在实际物理内存中的数据。不同进程的不同的虚拟地址可以映射到同一个物理页框，但是相同的虚拟地址不一样一定指向同一个物理地址。

Thinking 3.3

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);
```

而其中的e则为传入 `load_icode` 中的 `struct Env *e` ，因此我们的推测得到证实。

没有进程指针，我们的加载镜像的步骤显然不能正常完成。

举例：

C 语言 `stdlib.h` 中有

```
1 void qsort(void *base, size_t num, size_t width, int(*compare)(const void*, const
  void*));
```

`width` 参数说明了程序应当如何分割 `base` 起始的内存数据，方便向比较函数传参。比较函数的参数都是 `void*` ，需要一个元素的大小，确定是什么样的指针。

Thinking 3.4

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？

```
      va(加载起始地址)
|_ _ _|___|___BY2PG___|___BY2PG___|___BY2PG__|____|____|___BY2PG___|___|_ _ _|
offest|                                        |                     |
      |<---         .text & .data          --->|<---    .bss     --->|
      |<---            bin_size            --->|                     |
      |<---                        sg_size                         --->|
```

`.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。

Thinking 3.5

1. **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

2. **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。**

   该值是从 `load_elf` 中的 `*entry_point = ehdr->e_entry;` 语句中进行赋值，所以该值对于每个进程是一样的。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。同时，就算一样也可以从页表映射到不同的物理地址。如果设计成不一样会给操作系统带来更多麻烦，一致则可以减少复杂度。

Thinking 3.6

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？

应当设置为 `env_tf.cp0_epc` 。因为在《计算机组成》中介绍过 EPC 寄存器就是用来存放异常中断发生时进程正在执行的指令的地址的。因此在进入异常处理时，进程上下文中的 `env_tf.pc` 应该设置为 `epc` 。

Thinking 3.7

1. **操作系统在何时将什么内容存到了 TIMESTACK 区域**

   进程切换时，进程的上下文环境会被保存到 TIMESTACK 区域。

   在 `lib/env.c/env_run()` 函数中将当前进程的上下文环境保存到了 TIMESTACK 区下面一段大小为 `sizeof(struct Trapframe)` 的区域。

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

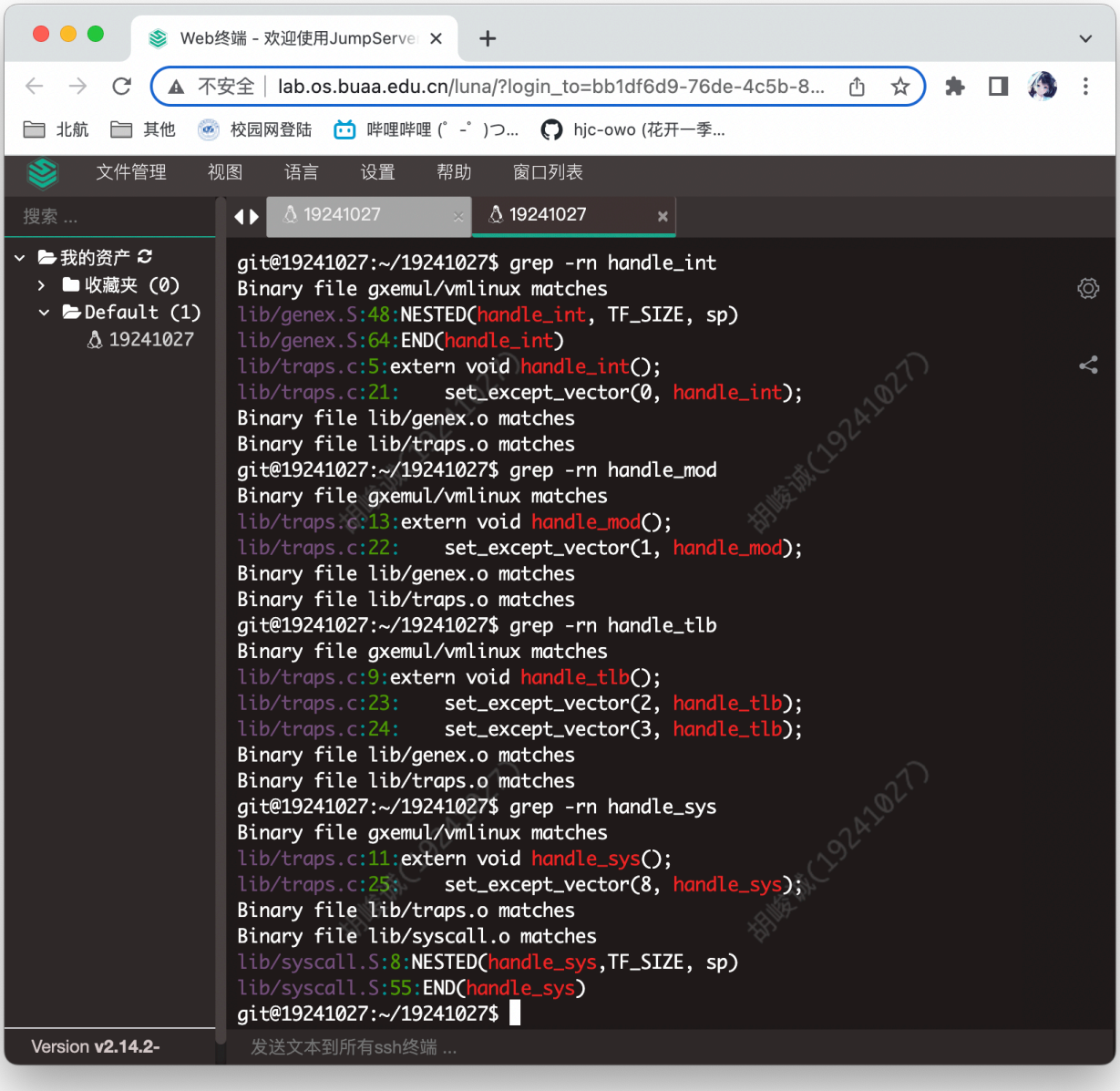
2. **TIMESTACK 和 env_asm.S 中所定义的 KERNEL_SP 的含义有何不同**

   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。

Thinking 3.8

试找出上述 5 个异常处理函数的具体实现位置。

0 号异常 的处理函数为 handle_int，genex.S
1 号异常 的处理函数为 handle_mod，genex.S
2 号异常 的处理函数为 handle_tlb，genex.S
3 号异常 的处理函数为 handle_tlb，genex.S
8 号异常 的处理函数为 handle_sys，syscall.S
如下图所示。



Thinking 3.9

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

set_timer:

```
LEAF(set_timer)
  # 首先先将 0x01 写入地址 0xb5000100 中
  # 其中基地址0xb5000000为gxemul用于映射实时钟的地址，偏移量0x100代表时钟的频率。
  li t0, 0xc8
  sb t0, 0xb5000100

  # 将栈指针设置为KERNEL_SP中能够正确产生时钟中断
  sw  sp, KERNEL_SP

```

```
  # 再调用宏函数 setup_c0_status 来设置 CP0_STATUS 的值
  setup_c0_status STATUS_CU0|0x1001 0

  # 通过 jr ra 来返回
  jr ra
  nop
END(set_timer)
```

timer_irq:

```
# 在 handle_ int 中判断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
timer_irq:
  # 首先写 0xb5000110 地址响应时钟中断
  sb zero, 0xb5000110
  # 在 timer_ irq 里直接跳转到 sched_yield 中执行
  # 而 sched_yield 函数会调用引起进程切换的函数来完成进程的切换
1:  j sched_yield
  nop
  j ret_from_exception
  nop
```

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。

在我们的操作系统中，设置了一个进程就绪队列，并且给每一个进程添加了一个时间片，这个时间片起到计时的作用，一旦时间片的时间走完，则代表该进程需要执行时钟中断操作，则再将这个进程移动到就绪队列的尾端，并复原其时间片，再让就绪队列最首端的进程执行相应的时间片段，按照这种规律实现循环往复，从而做到根据时钟周期切换进程。

实验难点

加载二进制文件镜像部分

这里主要是要考虑清楚各种情况，而考虑各种情况主要是保证两件事情：拷贝时候不越界、分配足够的空间。

- env.c 中的 `load_icode` 函数的工作为
  - 分配内存
  - 将二进制代码装入分配好的内存中，kernal_elfloader.c 中的 `load_elf` 完成，而 `load_elf` 的工作又被分为
    - 解析 `ELF` 结构
    - 将 `ELF` 的内容复制到内存中，env.c 中的 `load_icode_mapper` 完成。

尤其是 `load_icode_mapper` 对处理出的 elf 文件的各个 segment 进行二进制镜像加载时，要考虑按 BY2PG 对齐后的 offset 对 bcopy 函数中开始与结束位置的影响，也要考虑每一次拷贝的大小是不是一页的大小，需要非常清晰的思路，很容易在某一个细节因为考虑疏忽而出错。

调度机制 填写 sched_yield 函数

主要是弄清楚进程的调度是怎么操作的。

1. 设置两个队列，其中一个为目前的进程调度队列 `q0` ，另一个为一个空队列 `q1` 。
2. 首先判断当前队列指针指向的队首进程的 env_status
    - 如果为 `ENV_FREE` ，则要将该进程从队列中移除
    - 如果为 `ENV_NOT_RUNNABLE` ，则直接将其插入另一个队列的尾部
    - 如果为 `ENV_RUNNABLE` ，则判断这个进程的时间片是否用完，若用完则复原其时间片并将其插入到另一个队列尾部
    - 当一个队列为空时，将指针转移到另一个队列队首

其他

好几份汇编代码需要我们仔细阅读，包括env_asm.S, genex.S, kclock_asm.S 等，需要将汇编代码的功能与 C 语言代码的功能结合起来思考，才能正确理解中断发生以及进程调度的时候对于硬件以及软件需要做出的处理。

体会与感想

Lab3 带领我们整体上带我们过了一遍进程以及调度、中断以及异常相关的知识，在 lab2 完成对内存的管理后，lab3 带我们理解了进程运行时在操作系统以及硬件内部正在发生的事情。同样地，lab3 也训练了我们阅读大量的，包含各种宏定义、汇编、C 代码的工程代码的能力，做 exercise 的时候还是没办法准确定位 bug，全靠一行后面加一个 printf 实现精准定位（x。主要是你软院上学期计组没讲中断异常，导致花费的时间++。