一，实验思考题

> Thinking 3.1 为什么我们在构造空闲进程链表时使用了逆序插入的方式？ ■

答：首先我们在构建时使用的函数是 INSERT_HEAD 这个函数每次会向链表头部插入一个节点，而在取出的时候我们使用的是 LIST_FIRST。那么既然要取出 envs[0]那么我们在插入的时候就需要保证将 envs[0]作为链表头部，也就是将 envs[0]最后一个插入。故而在循环中这样写（如下图）

```
for (i = NENV-1; i >= 0; i--) {
    LIST_INSERT_HEAD(&env_free_list, &envs[i], env_link);
    envs[i].env_status = ENV_FREE;
}
```

> Thinking 3.2 思考 env_ setup_ vm 函数:
>
> - 第三点注释中的问题: 为什么我们要执行`pgdir[i] = boot_pgdir[i]`这个赋值操作?换种说法,我们为什么要使用`boot_pgdir`作为一部分模板?(提示:mips 虚拟空间布局)
> - 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?
> - (选做) 我们为什么要让`pgdir[PDX(UVPT)]=env_cr3`?(提示: 结合系统自映射机制)
>
> ■

答：

1. 首先我们此次试验中 mips R3000 采取的是 2G/2G 模式，程序切换到内核态不需要更换地址空间，因此每一个进程都有潜在转换成内核进程的机会，因此每个进程都有可能会调用内核函数。故而每个进程页目录中都需要保存一份内核页目录。
2. UTOP-ULIM 之间共 12MB 空间，属于用户可读不可写的部分，保存了 Pages 数组，Envs 以及页目录，各 4MB。ULIM 指的是用户可以使用的最大的地址范围，ULIM 之上是内核，用户不可读不可写。UTOP 我的理解是用户可以写的最高地址，UTOP 之下是用户可读可写的虚拟地址空间。
3. env_vr3 中存储的是页目录的物理地址，故而需要将页目录中对应页目录虚拟地址的页表项的值改为这个物理地址。也即第 PDX（UVPT）项

> Thinking 3.3 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) ■

答：不可以，这个参数很有用。我们在创建进程的时候，通过三个函数包装，将二进制可执行文件读取进了内存，这个过程中给这个进程分配的 env 就是这个 user_data，他作为一个参数将这个分配好的 env 一路传递下去。在 Page_insert 建立进程页目录到物理页面的映射的时候，env 就记载了进程页目录的虚拟地址。

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

```
for (i = 0; i < bin_size; i += BY2PG) {
    if (page_alloc(&p) == 0 && page_insert(env->env_pgdir, p, va, PTE_V) == 0)
        return -1;
    bzero((void *)page2kva(p), BY2PG);
    bcopy((void *)bin, (void *)(page2kva(p) + offset), BY2PG);
    bin += BY2PG;
    va += BY2PG;
```

Thinking 3.4 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？
- 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？
- 从布局图中找到你认为最有可能是`entry_point`的值。

答：

1. 针对虚拟地址空间
2. 虚拟地址在数值上看是一样的，因为入口虚拟地址都是在刨去 elf 开头固定长度信息之后的地址。但是每个虚拟地址映射到的物理地址是不一样的，因为不同进程回加载到内存中的不同位置。
3. UTEXT

Thinking 3.5 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？

答：进程切换使用的方法是触发中断，因此应该设置 env_tf.pc 为中断产生的时候的 pc 值也即 cp0_epc 以方便中断处理结束后继续执行。

Thinking 3.6 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
- 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)
- 思考 TIMESTACK 和第 18 行的 KERNEL_SP 的含义有何不同

答：

1. 保存了当前正在运行的进程的现场。
2. 答：

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

如图所示，在 env_run 方法中我们使用 TIMESTACK 来获得当前进程运行的环境，并将这个 Trapframe 存储进当前即将被启动的进程的 Trapframe 中作为上下文环境。

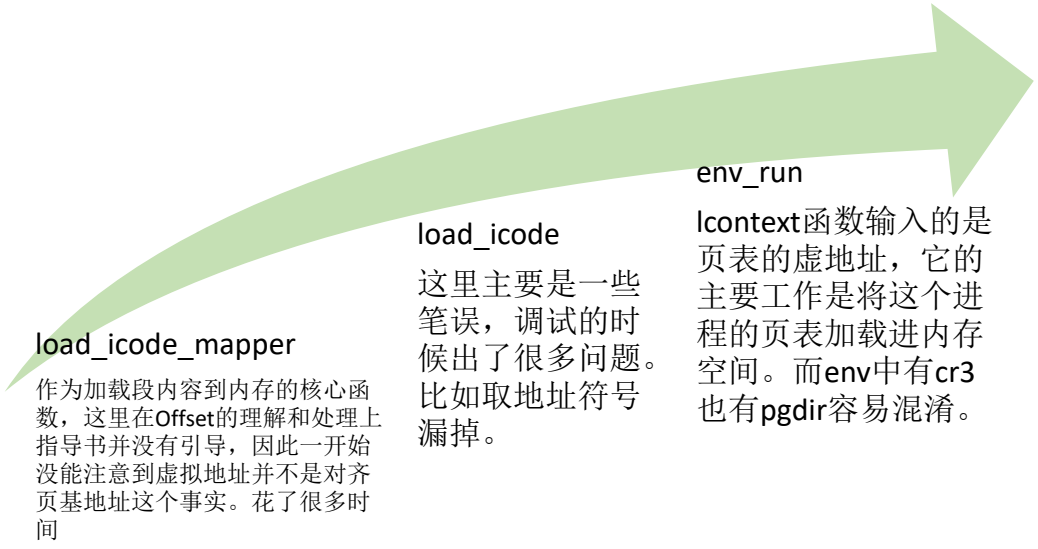
3. KERNEL_SP 指的是当前正在运行的进程的内核栈栈顶，而 TIMESTACK 我猜测是当前进程用户栈

> Thinking 3.7 思考一下你的调度程序，这种调度方式由于某种不可避免的缺陷而造成对进程的不公平。
>
> - 这种不公平是如何产生的？
> - 如果实验确定只运行两个进程，你如何改进可以降低这种不公平？
>
> ■

答：

1. 如果长任务前面不停的有短任务被分配进来，那么经过循环，每次只给一个时间片运行的话后来的短任务会被优先完成，而前面的长任务很久才能循环到一次。
2. 适当延长长任务的时间片，比如按照两个任务的预期完成时间分配不同比例的时间片，就像上机实验中的 1：2 一样分配。

二，实验难点图示



三，体会与感想

此次 lab 大家都说会体现出前面 lab 的坑，但是我还比较幸运 lab2 基本正确了。但是由于 lab3 中所有地址都是 u_long 的格式，导致出现两个问题，首先 mapper 函数里我一开始为了映射对不齐的 bin 使用了减法来计算，从而导致段内容比较短的时候会出现负数，因此循环的时候变成了一个很大的整数，将内存页面全部分配完了。其次这个格式刚好也符合地址的长度，因此后面写代码中如果漏掉了一个&取地址符号，线下的 IDE 静态检查也不会有问题，而在 os 中作为函数参数传入访问某个地址的时候就会出

TOO LOW 问题。

总体来说，我觉得指导书给出的参考资料和学习讨论气氛不够好，大家很有可能卡在了同样地 bug 但是也没有人一起讨论，群里大家只能泛泛一说，你的 lab2 有问题吧，这样反而误导了很多同学。比如说我最后一个 bug 就是 lab3 的笔误导致的，但是群里同学都说是 lab2 的两个 walk 有问题，我还是花费了很多不必要的时间去检查。

四，指导书反馈

希望在 mapper 的 offset 部分多一点指引

进程调度算法的运行给一点详细解释

五，残留难点

其实到现在，我依然不是很理解进程调度的具体过程，比如那个函数会调用 env_destroy 或者 free，而且 yeild 函数我也不知道是在哪里，怎么样被调用的。Poptf 的具体执行，希望指导书能给出扩展阅读。有一个整体的观感。