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)

- 注释中说：Insert in reverse order, so that the first call to env_alloc() returns envs[0]. 我们使用的是 LIST_INSERT_HEAD 宏，即每次都插入到链表的头部，我们从 freelist 中取出一项时，使用 LIST_FIRST 宏，那么既然要让取出的是 env[0]，那我们必须要让 env[0]最后插入，也就是倒序插入的方式构造链表。除此原因以外，我们在后面的时间片轮转算法中，是从 envs 数组的 envs[0]开始的，因此我们需要逆序插入，这样才能保证最先来到的进程最先被分配，最先被轮转。

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？

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？

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)

• 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 和 ULIM 差距是 3*4Mb 的大小。这 12Mb 的虚拟地址实际上存储的是记录页面使用情况的 4M 大小的 PAGES 数组，4M 进程控制块 ENVS 数组和用户页表域的那 4M 虚拟空间。由于这些变量应该是全局的，它们也应属于“内核态”。因此，我们在赋值的时候，是从第 PDX(UTOP)项开始把 boot_pgdir 赋给 envs_pgdir 的，而不是 ULIM。而 ULIM 以上的地址只是可以直接去掉高位进行物理-虚拟地址转换。

• 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)

- env_cr3 存储的是用户进程页目录的物理地址， e->env_pgdir[PDX(UVPT)] = e->env_cr3 | PTE_V | PTE_R;的赋值就是让用户进程页目录中的第 PDX(UVPT)项存储的物理地址等于用户进程页目录的物理地址，这样就完成了自映射。当我们给出一个属于 UVPT 域(UVPT ~ UVPT+4M)的虚拟地址时，进行虚拟到物理地址转换时，就会找到用户进程页目录的物理地址。

• 谈谈自己对进程中物理地址和虚拟地址的理解

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）

- 这个参数很关键。在我写 load_icode_mapper、load_icode、load_elf 这三个函数时，由于这三个函数是同一个进程的操作，那我们需要将记录该进程信息的数据结构一脉相承地作为参数传递下去，user_data 这个参数位可以为我们很好的服务。比如，我们需要在 load_icode_mapper 中申请物理页面，把物理页面使用 page_insert 函数映射到该进程的页目录、页表中，那么我们当然就需要一个记录该进程的数据结构的指针，这样我们才能顺利找到该进程的页目录。否则，没有这个参数的穿线，这三个函数怎么算是为一个进程服务的呢？或者，一些关于 I\O 的进程，也需要这个参数传递与用户交互的数据。

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？

- 这个函数是拷贝进程镜像的最后一关。通过调用 load_elf 找到了代码的各个区段，然后在 load_elf 中又调用 mapper 将各个区段（segment）映射好。最后，在该函数内，申请一页，添加映射关系（page_insert），将该进程的栈空间放在该物理页上，虚拟地址设置为 USTACKTOP-BY2PG 的位置上，保证栈顶位于 USTACKTOP 处。最后，由于我们没有在 env_alloc 函数内为进程设置 PC，因此我们在该函数内还需要给进程设置 PC 值。这里，需要感谢学长将本年的操作系统实验难度降低，

去除了给 PC 赋值的部分。使 PC 的赋值还有个各个 segment 的定位都放到了 elf_loader 函数内实现。

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？

- 是一样的。由于每个进程都有属于自己的地址空间，我们在切换进程的时候，切换了 cr3 寄存器，也就是对于每个进程有不同的页目录、页表，那么就有不同的虚拟、物理地址映射关系。因此，虽然 entry_point 的值对于每个进程都一样，但是其对应的物理内存是不一样的，自然也对应了不同的指令。虚拟地址相同，具体内容不同。

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?

- 需要把 env_tf.pc 的地址设置为 cp0_epc 的位置。进程切换是由于各种中断触发的，所以，这个进程在再次被恢复执行的时候， 应该执行导致（或遭受）异常的那条指令，相当于异常处理程序处理结束，返回到原来的程序中， 所需要执行的那条指令。而异常返回地址记录在 EPC 中。也就是说，应该执行 EPC（异常返回地址）寄存器中储存的那条指令。 这个知识实际在上学期计算机组成原理 P7 中已经学习过，这里没有什么难度。

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

• 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**

  - TIMESTACK 是一个保存上一个现场的栈结构。每个新的进程在 run 的时候，都需要将其拷贝一份到自己的 trap frame 里，以便记录该进程中断返回时回到的状态。这也符合栈的直观印象，是一个时间栈，越新的进程将在顶部的位置，每次只需要取顶部的拷贝进新进程的相关结构中即可。

Thinking 3.9 阅读 kclock_asm.S 文件并说出每行汇编代码的作用

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。