实验思考题

3.1

- 之所以按照逆序插入，是因为按照一般人的习惯，每次alloc出来的东西应该是有顺序的。按照逆序插入的话，将从envs[0]开始alloc

3.2

- 首先是必要性，这样生成的envid具有两两不同的性质，且envid可以反映他在envs中的位置
- 然后是充分性，这样生成的envid在寻找下标的时候，也就是调用ENVX的时候仅需做一些简单位运算，效率高

3.3

- 之所以可以复制一部分pgdir，是因为那一部分（UTOP以上）对于任何进程来说都是相同的。
- 总的来说，UTOP是用户写的边界，ULIM是用户读的边界。UTOP一下为用户自由使用，UTOP与ULIM之间的一段则用来给进程进入内核态查询使用
- 因为UVPT需要索引到用户pgdir的地址。保存之后可以很快的查找到
- 在2G2G布局下，从进程的角度来说，前一部分完全经由MMU达成，后一部分只有当成为内核态进程之后，才能用一个固定的方法访问得到。而物理内存就在那里，永远也不会动。怎么访问，要看是走流程还是直接开花。

3.4

- 比如是加载模式的可选择，比如对齐方式，比如pgdir的选定
- 具体的场景比如我现在为了效率，制定两个进程使用同一个pgdir或者同一个pgdir的一部分（size）来实现“内存共享”，这时便可以把这个参数利用起来。相当于是加载模式的参数化，类似的应用比如像sort函数中的自定义cmp

3.5

- 先ROUNDDOWN然后剩下的补齐，补零即可。具体分析请详见**难点分析**

3.6

- 是虚拟空间。
- 原因：在gxemul断点的时候，所有的指令位置均为0x8000....，因此是虚拟地址。理论上没有任何函数可以直接访问物理地址，就算不走MMU也要高位除0进行映射
- 一样。这种统一是一种标准，是ELF格式的标准。就像ASCII码那样。如果没有标准，那实际的应用兼容性将会更差

3.7

- 应为curenv的cp0
- 因为那是上一次异常发生时的现场

3.8

- TIMESTACK是0x82000000，是时钟中断来临后保存现场的栈。在中断保护中，有li sp,0x82000000

- TIME就是内部的时钟中断，STACK就是栈，这个栈本质上与kernel_sp没有特别大的不同。
- 只有在时钟中断来临进入exc_handler的时候，才会保存现场到TIMESTACK，其他的异常或中断都保存到kernel_sp

3.9

- 首尾定义函数，暂且不提。最后两句返回+延迟槽，暂且不提
- 前两句相当于时钟初始化，设置定时器的频率
- 然后将CP0的状态进行调整，以达到可以接受下一次中断的效果
- 宏里还有一些保存和恢复临时使用的寄存器的操作

3.10

- 初始化接受中断许可数为pri，每接受一次中断，许可数（cnt）减一，为0时切换进程，同时完成新进程的初始化

实验难点(包括残留难点)

本次实验中最难的，一个是env_run函数，一个是"还蛮难填的哦"的load_icode_mapper。当然，如果像我一样Lab2有bug，那无论哪个函数都很难（因为有bug）

- 首先，拿到55分还比较容易。~~唯一需要注意的是，由于评测体系的差异，page_init必须头插?~~
- 然后是蛮难填哦的函数（讲真，这个形如真的魔性）。这个函数本身没有多难，就是bcopy的使用。难点在于，各种对不齐，各种乱起八遭的情况。比如下面这个情况：
  - va=1；bin_size=BY2PG+2；sg_size=2*BY2PG+5
- 我认为，只要可以考虑出这个情况的代码该如何填写，就能够比较全面的想清楚所有的不对齐情况。

然后，是env_run

- 这个函数通过评测端比较容易，但是我仍有残留问题。在保存完Trapframe后，我在学长的代码中看到了一个env_runs++，我一开始没有写一条语句，顺利的出了现象，并通过了测试。但我后来将其补上，本地爆出了访问低地址（空指针）的异常。这个bug还没有被彻底解决

最后，是Lab2

- 我一开始的Lab2的虚存管理应该是写错了，导致异常分发代码的部分被奇妙的覆盖了。具体问题就出在boot_pgdir_walk中。将其改正后，又多次爆其他的bug，加上我之前在Lab2的代码中可能进行了奇怪的改动，我决定直接重写
- 然后，我删掉了Lab0之后的所有库，重新提交Lab0。迅速抄完了Lab1之后，仔细的抄完了Lab2。这一步完成之后，我利用lab2两次课上的方法，对自己的pgdir_walk等函数进行了一些测试，随后毅进入了Lab3。（这是我第四次进入Lab3）
- 这一次运气不错，没有发生异常代码被覆盖的诡异情况，然而，每当执行创建进程这一函数的时候，就会疯狂刷pageout。首先，我检查了page_alloc,发现可正常分页，然后我检查了pgdir_walk，发现可以正常寻找，之后我又检查了page_insert，发现也一切正常。在疯狂检查lab2的过程中，我消耗了大约5-6小时
- 之后，我终于决定进入进程内部一探究竟。我花了一些时间，把lab2-1和lab2-2的代码整套转移到了env_run中，同时检查全局页分配，进程页分配和物理页状态。同时为了更快的debug，我直接无视非法写入，在申请page时便直接或上了PTE_R。结果非常申必，我的进程申请了几百个**优质**（空白的，能走MMU访问的，PTE_V和PTE_R都有的）的页面，结果申一个pageout一个，这让我非常困惑

- 接下来，我注意到这次实验中，进程申请页面主要出现在加载二进制镜像中。于是，我检查了load_icode函数，发现他可以被执行，然后我将断点加在了**还蛮难填的哦**的函数上，我惊奇的发现他并没有被执行。我就很奇怪，我的env.c从头到尾都检查/对拍过好几次了，为什么会有这样的问题呢？
- 相信聪明的你已经发现了，我一直在怀疑pmap和env两个文件，完全没有注意到，lab3是**不自带load_elf函数的**，换句话说，这个函数我根本没有填写。想到这里，我豁然开朗。之前的错误大概是这样的：
    - 进程：OS，来一页内存
    - OS：好嘞，alloc，pgdir，insert
    - 进程：我要写我的镜像
    - 写镜像：函数都没填你写啥玩意儿？摸了摸了
    - 进程：写好了？诶不对啊，我还没加载完，OS，再来一页内存

    如此这样循环往复，最后就进入了疯狂pageout的地步