一．思考题

1．

采用逆序的方式构建空闲链表，这样第一次调用 env_alloc()就会返回下标最小的进程控制块，也就是 envs[0]

2．

1）有一个静态变量 next_env_id 初始化为 0，用于记录进程创建进程次数，在每次调用 mkenvid()时，next_env_id 都会先自增一，然后再左移 11 位，然后整体与临时变量 idx 做逻辑或操作，其中 idx 是进程控制块在进程控制块数组 envs 的下标，操作系统课程实验限定最多的进程数是 1024，所以需要用十位表示。这样既可保证进程的 id 唯一，又能保证 id 有意义。

2）在 envid2env()中，判断进程 id 是有效的当且仅当进程控制块中的 env_id 等于参数 envid 且进程的状态不为 ENV_FREE，所以如果不判断 e->env_id != envid 则可能出现进程的 id 不匹配的情况，造成错误

3．

1）按照提示将 PDX(UTOP)之前的页目录项都初始化为 0，除 PDX(UVPT)外之后的页目录项都照搬内核页目录项。其中 UTOP 以上的空间，从低地址到高地址分别为进程控制块、页表、内核空间的中断异常、代码段、页表等，所有进程都是相同的，这样每个进程在需要的时候都有成为内核进程的机会，所以照搬就可以

2)ULIM 对应地址 0x80000000,UTOP 对应地址 0x7f400000,在 UTOP 到 ULIM 之间的区域用户没有写权限

3）因为系统自映射机制，下标为（0x7f400000 右移 22 位）的页目录项对应该进程页目录的物理地址

4）进程的物理地址就是根据虚拟地址和页表得到的地址，只有相关代码被加载的内存的正确位置才能被运行

每个进程都有自己的 4G 虚拟空间，这样既可实现进程间的独立又可扩展程序的运行空间。

4．

不可以没有这个参数。user_data 在 load_icode_mapper()被强制转换成 Env 类型，而之后的代码需要根据被强制转换后的 user_data 才能将页面插入正确的位置，所以 user_data 是有意义的，而向上追溯，其一直以（struct Env *e）的形式在函数间传递，其源头是在 env_create_priority()中用 env_alloc()函数创建的进程。

5．

特殊情况：offset 不为 0，复制的内容大小不是 4096 的整数倍，存储大小大于复制文件大小等

当存储空间大小大于复制文件大小时，即参数 sgsize > bin_size 是需要自动填充.bss 段

6．

1）虚拟空间

2）entry_point 对每个进程是一样的，因为 elf 文件被加载到了固定位置，因此每个进程的入口是一致的，这种统一来自于 elf 文件格式的统一。

7．

env_tf.pc 的值应该设置为 cp0_epc。在计组课程中，一个之前被切换的进程还要戒指被中断的地方进行，而中断处的 PC 值就保存在 cp0_epc

8.

KERNEL_SP:0　　TIMESTACK: 0x82000000

其中 TIMESTACK 作为一个发生时钟中断时取出之前上下文的栈指针

在 include/stackframe.h 中出现了几处汇编代码其中最能说明的是 get_sp，如下：

```
.macor get_sp
    mfc0 k1, CP0_CAUSE
    andi k1, 0x107C
    xori k1, 0x1000
    bnez k1, 1f
    nop
    li sp, 0x82000000
    nop
1:
    bltz sp,2f
    nop
    lw sp, KERNEL_SP
    nop
2:
    nop
.endm
```

解释如下：将 CP0_CAUSE 中的值取至 k1 寄存器中，经过一系列逻辑运算，比较 k1 寄存器的值是否等于 0，若不等于 0 则将 sp 设置为 KERNEL_SP，若等于 0 则将 sp 设置为 0x82000000 也就是 TIMESTACK 的值；而 get_sp 在宏定义 SAVE_ALL 中被引用，SAVE_ALL 的作用是将进程现场保存至栈中。

所以当发生时钟中断时 CP0_CAUSE 中的值为 0，sp 被设置为 TIMESTACK，所以我认为 TIMESTACK 是一个发生时钟中断时指向栈顶的指针。而 KERNEL_SP 应该是发生非时钟中断指向栈顶的指针。

9.

```
.macro setup_c0_status set clr
// 宏定义，其中 setup_c0_status 为函数名，set 和 clr 为两个参数
    .set push              // 入栈
    mfc0 t0,CP0_STATUS     // 将 CP0_STATUS 中的值取出至 t0 寄存器中
    or t0,\set|\clr        // t0 = set | clr
    xor t0, \crl           // t0 = t0 ^ clr
    mtc0 t0,CP0_STATUS     // 将 t0 寄存器中的值写回 CP0_STATUS 中
    .set pop               // 出栈
.endm
    .text
LEAF(set_timer)
    li t0,0x01             // t0 = 0x01
    sb t0,0xb5000100
```

```
                                    // 将 t0 寄存器中的值写入 0xb5000100（模拟
                                    //器(gxemul) 映射实时钟的位置)
        sw sp, KERNEL_SP            // 将 sp 存入 KERNEL_SP
        setup_c0_status STATUS_CU0|0x1001 0
        // 调用宏定义，其中 STATUS_CU0 = 0x10000000
        jr ra                       // 返回
        nop
END(set_timer)
```

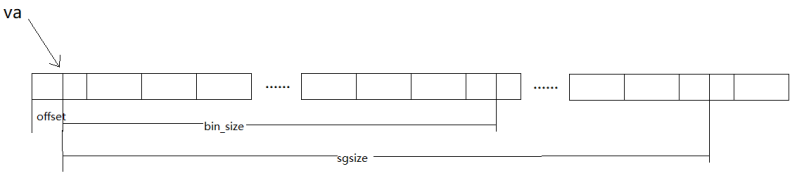
10.

利用一个静态变量 count，先令 count 等于一个进程的优先级数值，每当发生时钟中断的时候，count 自减 1，当 count 等于 0 的时候，切换进程。

二．实验难点图示

此次实验中的难点之一就是要将 elf 文件加载到正确位置上，与 elf 文件加载相关的函数主要有三个，分别是 load_icode()、load_elf() 和 load_icode_mapper()，调用关系是 load_icode() → load_elf() → load_icode_mapper()，其中最难填写的就是 load_icode_mapper()

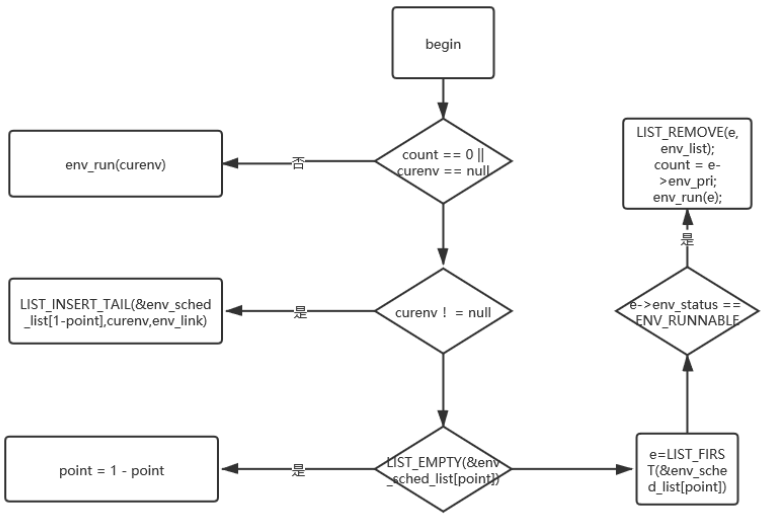
图示如下：



ATTENTION:

1. 文件内容不足一个页面大小也要分配一个页面
2. va可能不是4096的整数倍，此时需要用到偏移量offset
3. bin_size可能小于sgsize，此时应该按序将大小为(sgsize-bin_size)的部分填充为0

另外一个难点就是进程的调度，此时实验只用两个进程并且采用时间片轮转算法，流程如下：



三．体会和感想

通过此次的 lab，我大体上知道了操作系统进程运行和异常处理机制的过程，特别是进程控制块是如何记录进程信息以及 elf 文件是如何正确加载到正确位置上的。此外，在这次实验中我也对虚拟内存分布有了更深的理解，比如明白了 UTOP 和 ULIM 的位置以及相应部分的作用。

在这次的实验中，有较多的时间花费在了加载 elf 文件部分函数以及进程调度函数的填写，其中加载 elf 文件部分主要是没有考虑全面所可能出现的各种情况，导致 elf 文件始终不能正确加载，总是出现^^^^^TOO LOW^^^^^的 bug，最后我在纸上详细地整理了一遍所有可能出现的情况，最后成功修复；进程调度部分主要是对代码中的提示没有完全理解，最后和同学交流才成功修复 bug。

总的来说，这次的 lab3 难度较大，需要花费较多的时间才能完成。

四．Extra

此次的 lab3-extra 很简单，花费了两个小时，没有遇到什么障碍。