1. 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)

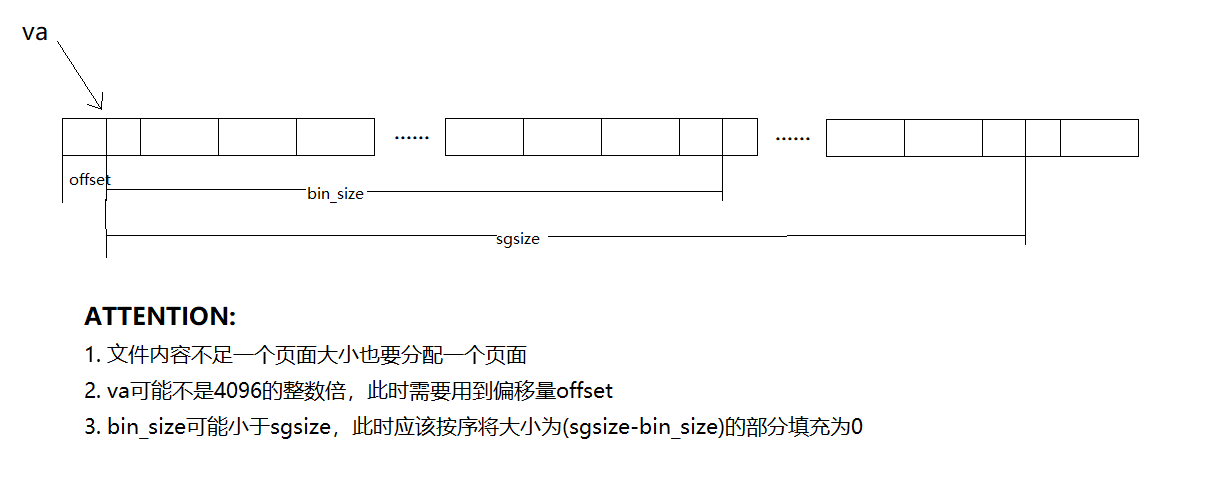
10.

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。

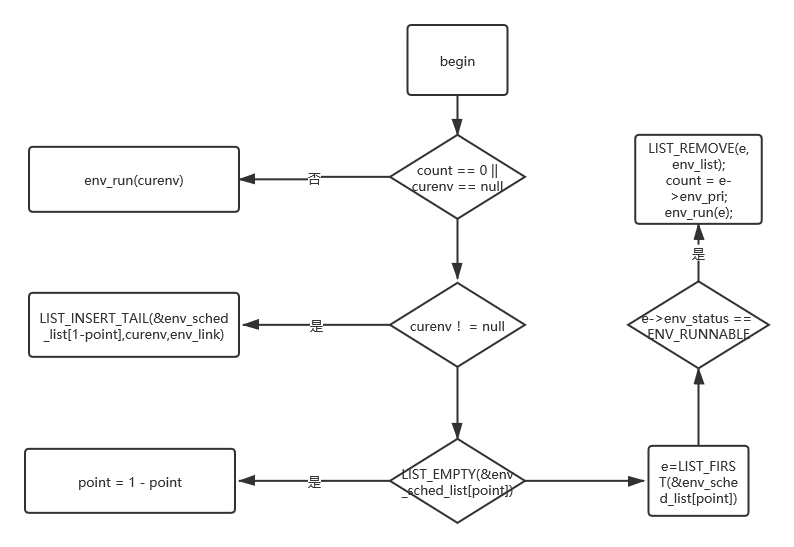
1. 实验难点图示

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：



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：



1. 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。

1. Extra

此次的lab3-extra很简单，花费了两个小时，没有遇到什么障碍。