1. **思考题**

**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)**

逆序，

insert in reverse order,so that the first call to env\_alloc() return envs[0]

这样每一次调用env\_alloc函数时都返回env[0]，即第一个进程块

而进程块按照在数组中的排列顺序，在内存中按照顺序由低地址到高地址排列，可以按顺序逐块调用内存块。

我们使用的是LIST\_INSERT\_HEAD这个宏定义，将一个节点插入变成链表的第一个可用节点，而我们在取用的时候使用LIST\_FIRST这个宏定义来取出env块。

**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？**

使用静态变量next\_env\_id确保每一个生成的id之间不相等；

左移(1+LOG2NENV)位，在后10位或上env在envs数组中的下标，是为了在envid2env中使用ENVX宏定义获得这个索引并找到对应的env控制块。

e = &envs[ENVX(envid)] 这句语句在从envs数组中获取进程控制块时仅根据输入envid的后9位，防止特定envid对应的进程控制块被其他后9位相同但是前面几位不同的非法envid

获得。

**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)**

根据mmu.h中虚拟空间布局中，我们的小操作系统是2G/2G模式。用户态占用2G,内核态占用2G，在我们的实验中，对于不同的进程而言，其虚拟地址ULIM以上的地方，映射关系相同，这为内核地址，这2G虚拟地址与物理地址的对应，不是由进程管理，而是由内核管理的，在初始化内存管理时已经将虚拟内存内核部分的空间分配完毕了。

**• 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为0x7f400000，用户有写权限以及读权限的地址界限，UTOP与ULIM之间是用户只读的内核数据结构。

ULIM为0x80000000，这以上的地址为内核地址，这是用户态程序可以访问地址的界限，更高的内存用户不可读。

UTOP到ULIM是属于用户并且只读的区域，但是内核这部分映射到了用户区，UENVS与envs均映射到了envs对应的物理地址。开启中断后进程访问内核会产生异常来陷入内核，内核开辟这4M为用户进程虚拟区，用户读这4M空间的内容不会产生异常。

**• 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)**

根据自映射机制，pgdir[PDX(UVPT)] 指的是页目录中对应UVPT的页表项

设置UVPT映射到4MB大小的页表的起始地址。

**• 谈谈自己对进程中物理地址和虚拟地址的理解**

对于进程来说，访问的地址都是虚拟地址，每一个进程都有一个属于该进程的页目录，进程访问到的数据，通过这个页目录以及之前建立的页式虚拟内存管理机制，来访问到储存在实际物理内存中的数据。不同进程的不同的虚拟地址可以映射到同一个物理页框，但是相同的虚拟地址不一样一定指向同一个物理地址。

**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）**

没有这个参数无法正常执行。在load\_icode中实际上赋给它的是一个env进程控制块，作为在load\_icode, load\_elf,与load\_icod\_mapper三个函数之间传递信息的媒介，在load\_icode\_mapper中使用它找到进程对应得页目录，完成page\_insert工作。

**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？**

由于ROUNDDOWN采用了向下对齐，

复制每一段时，要将对齐产生的offser偏移量考虑进去，

i==0,即复制第一段时，需要考虑：

(BY2PG-offset)<bin\_size

(BY2PG-offset)>=bin\_size

i>0，即复制完第一段还有剩余未复制的bin时，需要考虑：

0<i<bin\_size： BY2PG<bin\_size-i+offset

BY2PG>=bin\_size-i+offset

**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？**

针对物理内存，因为CPU只能访问物理内存

**• 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？**

对于每个进程一样,elf文件在模拟器中被加载到同一个位置，每个进程的入口点一样，这种统一来源于elf文件格式得高度一致性。

对于每个程序来说，起始地址是统一的，便于编程以及交互，进程访问的应该是虚拟地址。

对于CPU来说，每个进程的起始地址不同，CPU按照顺序从指令存储器中读出二进制码，并逐条执行。

**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？**

保存的是上一个进程的上下文中的cp0\_epc寄存器的内容

EPC寄存器保存的是系统发生中断或者异常得时候系统正在执行得指令对应得地址，

当下一次切换进程恢复上下文环境时，可以直接获得pc指令，继续执行。

**Thinking 3.8**

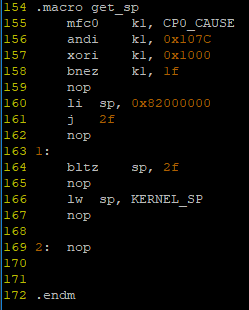
**思考TIMESTACK 的含义，并找出相关语句与证明来回答以下关于TIMESTACK 的问题：**

**• 请给出一个你认为合适的TIMESTACK 的定义**

发生时钟中断后保存现场使用得的存储区，是一个栈顶指针。

**• 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)**

./include/stackframe.h



**• 思考TIMESTACK 和第18 行的KERNEL\_SP 的含义有何不同**

TIMESTACK是时钟中断后的存储区

KERNEL\_SP是系统调用后的存储区

**Thinking 3.9**

**阅读 kclock\_asm.S  文件并说出每行汇编代码的作用**

macro宏定义定义了一个宏函数set\_c0\_status;

.set push 与 .set pop 用于保存和恢复由.set控制的所有选项的当前设置。

先取出CP0寄存器的值，与set和clr变量进行或运算，再与clr变量做异或运算，并将结果存入CP0寄存器。

20,21行向0xb5000100位置写入1，其中0xb5000000是模拟器(gxemul) 映射实时钟的位置。偏移量为0x100 表示来设置实时钟中断的频率。

22行将sp寄存器的值存入KERNEL\_SP栈中

23行使用宏定义修改CP0寄存器状态

24行执行return操作

**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。**

定时器提供了时间片的机制。首先通过kclock\_init对定时器初始化，在函数中调用set\_timer函数，对实时钟进行设置，再将栈指针压入栈中，修改CP0寄存器状态。

时间片运行结束，时钟中断被触发后，使用genex.s中的handle\_int函数，得到具体的中断号，若为实时钟触发的中断则跳到time\_irq。

time\_irq中先跳到我们自己设计的sched\_yield函数，使用时间片调度算法进行进程间的调度与切换。

1. **实验难点**

load\_icode\_mapper对处理出的elf文件的各个segment进行二进制镜像加载时，要考虑按BY2PG对齐后的offset对bcopy函数中开始与结束位置的影响，也要考虑每一次拷贝的大学奥是不是一页的大小，需要非常扎实的基础以及清晰的思路，很容易在某一个细节因为考虑疏忽而出错。

好几份汇编代码需要我们仔细阅读，包括env\_asm.S,genex.S,kclock\_ams.S等，需要将汇编代码的功能与C语言代码的功能结合起来思考，才能正确理解中断发生以及进程调度的时候对于硬件以及软件需要做出的处理。

sched\_yield需要我们思考进行进程调度时可能发生的场景，包括两个调度队列是否可能为空以及相应处理，在两个队列之间的切换方法等(pos 与 pos^1)。

需要联系很多前面lab2的知识，并且确保lab2的代码完全正确。

1. **体会与思考**

Lab3带领我们整体上带我们过了一遍进程以及调度、中断以及异常相关的知识，在lab2完成对内存的管理后，lab3带我们理解了进程运行时在操作系统以及硬件内部正在发生的事情。也让我对lab2中的部分内容有了更深入的理解，比如内核态与用户态、VPT与UVPT之间的关系等。

同样地，lab3也训练了我们阅读大量的，包含各种宏定义、汇编、C代码的工程代码的能力，在写lab3的过程中我不断地查阅前面的知识，甚翻开了上学期计组实验的内容，扎实了我们的基础。

1. **指导书建议**

对于一些在汇编代码中定义的函数，希望指导书可以给出更多的讲解，包括.set等代码的意义。

对于sched\_yield函数的要求，指导书中讲的不是很详细。