Lab3 实验报告

19373135 田旗舰

一、实验思考题

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)

为了使链表中的 Env 块的顺序与 envs 数组中的 Env 块顺序一致，方便操作。

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？

低 10 位为 Env 块在 envs 数组中的下标，其余高位为调用 mkenvid 函数的次数，这样的 id 既包含了 Env 块在 envs 中的顺序信息，又使得每个 Env 块的 id 互不相同。

防止在没有分配该 envid 或 envid 错误时调用该函数，此时后 10 位为 0，错误地返回 envs[0]。

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？

• 在 env_setup_vm 函数的最后，我们为什么要让 pgdir[PDX(UVPT)]=env_cr3?(提示: 结合系统自映射机制)

• 谈谈自己对进程中物理地址和虚拟地址的理解

因为实验中的操作系统是 2G/2G 内存布局，没有真正的内核进程，用户进程可能陷入内核态访问内核，因此需要 boot_pgdir 访问内核空间。

UTOP 是用户可以使用的空间的最高地址，ULIM 是用户空间的最高地址。UTOP 到 ULIM 的区域用户没有权限访问。

env_cr3 储存着页目录的物理地址，通过该语句完成页目录的自映射。

物理地址是真实的地址，所有进程共享；虚拟地址是虚拟的，每个进程独占的。

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）

不可以，这个参数是为了向内层函数传递参数。

qsort 函数，需要传递 compare 函数。

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？ （提示：1、页面大小是多少；2、回顾 lab1 中的 ELF 文件解析，什么时候需要自动填充.bss 段）

bin_size<BY2PG-offset

bin_size>=BY2PG-offset &&(bin_size-BY2PG+offset)%BY2PG!=0

bin_size>=BY2PG-offset &&(bin_size-BY2PG+offset)%BY2PG=0

我都考虑到了:)

sg_size>bin_size 时需要自动填充.bss 段，即为其分配页面。

这里的 e->env_tf.pc 是什么呢？就是在我们计组中反复强调的甚为重要的 PC。它指示着进程当前指令所处的位置。你应该知道，冯诺依曼体系结构的一大特点就是：程序预存储，计算机自动执行。我们要运行的进程的代码段预先被载入到了 entry_ point 为起点的内存中，当我们运行进程时，CPU 将自动从 pc 所指的位置开始执行二进制码。

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？

• 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？

虚拟空间，因为虚拟空间是连续的。

是不是一样的取决于二进制文件首地址的定义，对于本实验中的 user_a 和 user_b 是一样的。都是从 elf 文件中读取得到的。

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？

env_tf.cp0_epc。cp0 会保存中断发生时的 pc，这样设置在处理完中断后可以回到中断发生前的位置继续执行。

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

• 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 是用来保存上下文信息的栈指针

stackframe.h 中

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

```
2: nop
.endm
```

get_sp 首先先判断是否发生的是 4 号中断引起的异常，然后选择相应的栈指针。

TIMESTACK 是产生时钟中断异常时用的栈指针，KERNEL_SP 是非时钟中断异常用的栈指针。

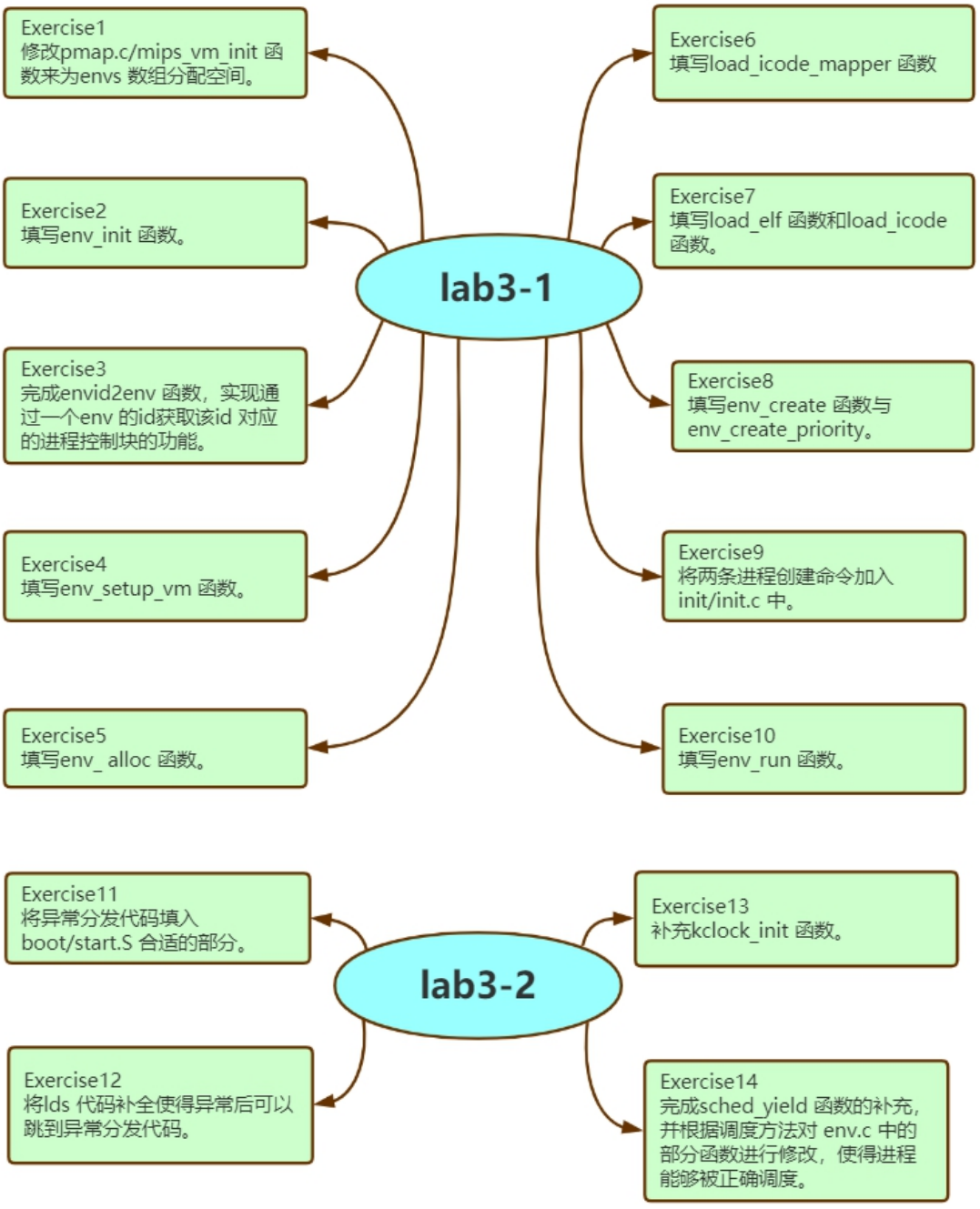
Thinking 3.9 阅读 kclock_asm.S 文件并说出每行汇编代码的作用

```
LEAF(set_timer)

   li t0, 0x01
   sb t0, 0xb5000100#将 1 存入 0xb5000100，开启实时钟
   sw sp, KERNEL_SP #将 sp 设为内核栈指针
setup_c0_status STATUS_CU0|0x1001 0 #设置 cp0 状态
   jr ra #返回上一级

   nop
END(set_timer)
```

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。

当时钟中断产生时，PC 指向异常处理代码段，调用 handle_int 函数，读取 CPU_CASUE 和 CPU_SATUS 到 t0 和 t2 中，将 t0 and t2 存入 t0 得到具体的中断号，判断如果是 4 号中断位则执行中断服务函数 time_irq，跳转到 sched_yield，通过时间片轮转算法从而切换进程。

二、实验难点图示



本次实验主要涉及进程的创建及运行、时钟中断的实现以及通过时钟中断实现进程切换。本次实验代码量相比之前的实验明显增大，涉及的函数也较多，而且缺乏便捷有效的本地测试方法，导致出 bug 的可能较大，而且较难定位。在完成实验的过程中，我仅仅是在 load_icode 函数中在初始化新进程的栈时忘记将页

面设置为可写，导致在执行 env_run 时出现了 bug，以致于花费了巨大的精力调试。但这并不是不会或者写错而导致的，仅仅是因为粗心，可见，本次实验的一个难点就是将每一个简单的、单独的函数都写对。

除了需要细心外，我认为本次实验的难点如下：

1.load_icode_mapper 函数

由于 va、bin_size、sg_size 都可能不是 BY2PG 的整数倍因此要考虑 3 种不同的对齐情况，如同所示（图片来自讨论区）

因此，在编程的时候不能用一个 while 就解决，需要考虑 offset，在循环外部或内部加入 if 语句单独判断边界情况。

另外，还需理解好 load_icode、load_elf、load_icode_mapper 之间的关系。

2.sched_yeild 函数

函数本身的编写并不是太难，只需要切换进程 list 以及将进程的 priority 减 1 即可，但很容易忽略 curenv 为 NULL 的情况，而当运行第一个进程时，curenv 即为 NULL，因此需要在 if 中添加条件判断。

三、体会与感想

本次实验总体难度较高，其中，将各个函数按照注释填写完整难度并不是太高，但真正理解每一个函数，并且能够在出现 bug 时定位问题所在并解决难度较大，以至于我在本次实验上花费了大量的时间。完成 14 个 exercise 我总共只花费了一个下午加两个晚上的时间，但在本地运行时却出现了 bug，后面通过 printf

等方法debug花费了整整3天，让我第一次在这门课的学习中真正感受到了困难。但在发现 bug 改正后，回头想想其实也并没有那么难，其实我们的任务还是相对简单的，毕竟大量的代码已经写好，我们要做的只是阅读和理解，再进行补充。但是首先要建立一个整体的进程概念，清楚每一步在做什么，才能尽量在完成实验的过程中不出现 bug。

四、指导书反馈

在 sched_yeild 函数中，需要首先判断 curenv 为 NULL 的情况，即执行第一个进程的情况，但指导书上仅提到了当前进程，并没有强调当前进程为空的情况，感觉初次接触调度算法时很容易忽略，且不容易发现，最好能在指导书中加入一定的提示，考虑初始情况。

五、残留难点

虽然顺利完成了整个实验，但不看着实验代码还是无法构建出整个进程的布局，也不能详细地描述出进程创建和执行的流程，在整体布局的理解上还需要进一步熟悉。