Lab3实验报告

19376273 陈厚伦

思考题

Thinking3.1

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)

按照头插法，是逆序。保证链表中的顺序和`envs`数组中的顺序相同。加载空闲进程时会使用LIST_FIRST()函数，会优先加载`envs`数组前面的块。进程调度时，最近使用过的放回到空闲链表时也是放到头部，会被优先调度。

Thinking3.2

思考 env.c/mkenvid 函数和 envid2env 函数:

• 请谈谈对 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?

```
// 通过static变量标记mkenvid的调用次数
static u_long next_env_id = 0;
// 计算e在envs的索引
u_int idx = e - envs;
return (++next_env_id << (1 + LOG2NENV)) | idx;
```

id生成逻辑综合了进程控制块在数组中的索引和id生成的顺序。确保了每个进程的进程号是独有的。进程是动态的，所以id考虑了调用mkenvid的次数。id中有索引，可以通过id在envs中找到对应进程块获取进程信息。

如果envs数组中发生了替换，即e所指的进程块被替换，会导致envid与e->env_id不同的情况，导致找到的进程块错误。

Thinking3.3

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 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)

• 谈谈自己对进程中物理地址和虚拟地址的理解。

我们的实验采取一种混合的2G/2G模式，用户态占2G，内核态占2G。用户的2G地址空间是独享的，内核的部分还是同样的地址空间，所以要复制boot_pgdir后半部分。

UTOP以下是用户进程能够自由读写的部分，ULIM是kseg0与kuseg的分界线，是系统给用户进程分配的最高地址。UTOP到ULIM的区域只可以读不可以写，存放用户的进程信息与页表信息。

`env_cr3` 保存着进程的页目录的物理地址，`pgdir[PDX(UVPT)] = env_cr3` 建立了自映射，页目录的第 `PDX(UVPT)` 项需要映射到进程页目录自身。

进程能够使用的都是虚拟地址，通过查询进程页表对应到相应的物理地址。虚拟地址可以根据需要设定，物理地址位数由硬件决定。我们的实验中，不同进程对于内核的2G的地址空间是相同的，对于用户的2G有各自独立的地址空间，不同的进程使用同样的虚拟地址也可以访问到不同的物理地址，不会发生冲突。

Thinking3.4

结思考 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）

不可以没有这个参数。`load_elf` 需要传一个 `int*` 函数，这个函数需要 `void* user_data`，有这个参数是为了向内层函数传值。

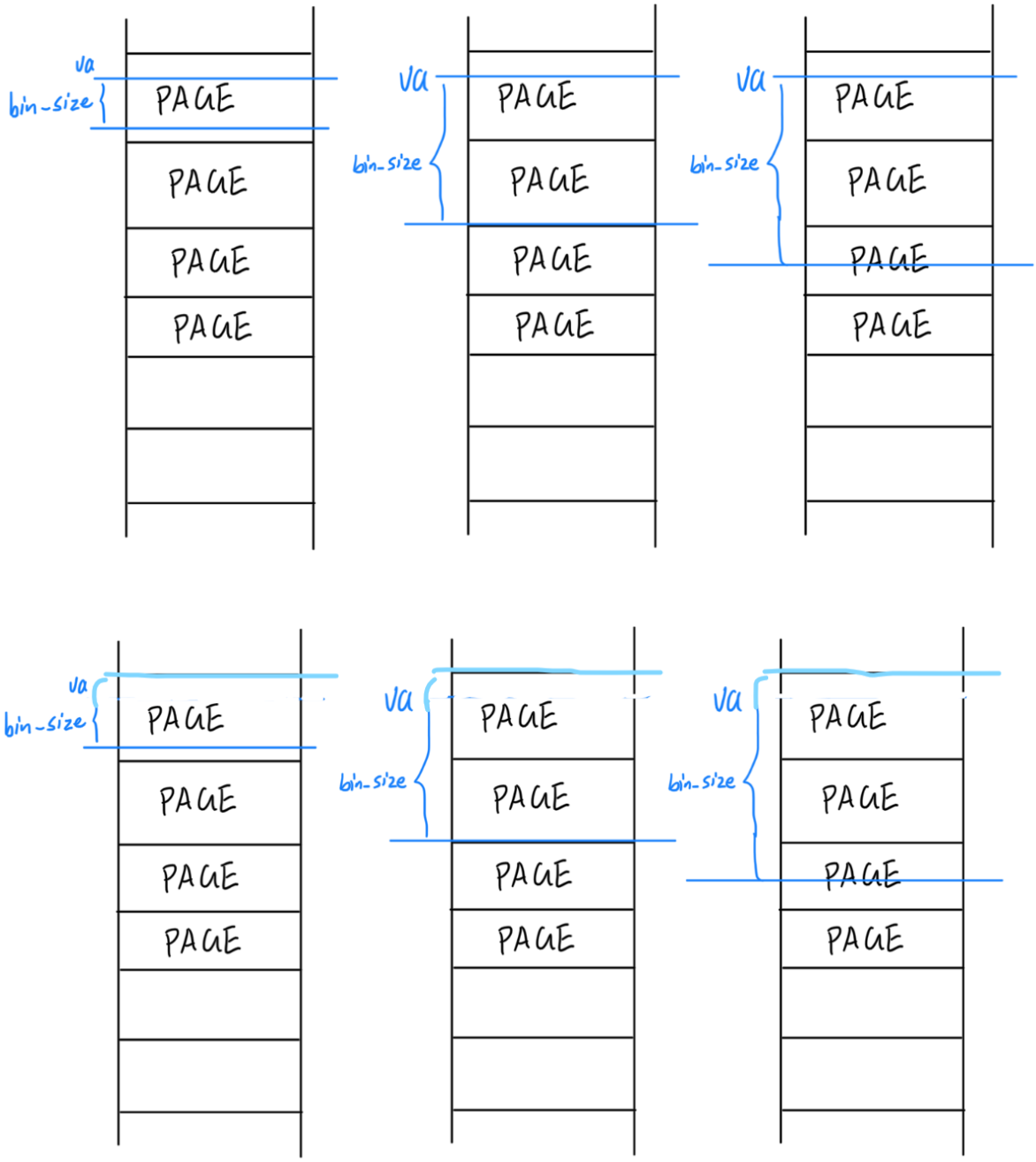
```
int load_elf(
    u_char *binary,
    int size, u_long *entry_point,
    void *user_data,
    int (*map)(
        u_long va,
        u_int32_t sgsize,
        u_char *bin,
        u_int32_t bin_size,
        void *user_data
    )
);
```

案例：`qsort()` 函数的width参数说明了数组每一个元素的大小，方便向比较函数传参。（比较函数的参数都是void*，需要一个元素的大小，确定是什么样的指针）

```
// qsort()函数
void qsort(
    void* base,
    size_t num,
    size_t width,
    int (*compare)(const void* e1,const void* e2)
);
```

Thinking3.5

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）



`sgsize`与`bin_size`同理。

Thinking3.6

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？

是虚拟空间，顺序执行时都是PC+4，对于不连续的物理内存来说，显然是不行的。

entry_point是一样的。每次执行时都是从一个固定的虚拟地址开始，对CPU是友好的，虚拟地址相同但是进程PCB不同，可以映射到各自不同的物理地址。

Thinking3.7

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?

设置为CP0的EPC寄存器的值，触发异常与中断时硬件会把处理完异常中断后重新执行时的指令地址写入EPC，所以PC要设置为EPC的值。

Thinking3.8

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

`mmu.h`中定义了TIMESTACK，

```
#define TIMESTACK 0x82000000
```

TIMESTACK是时钟中断时存放CPU寄存器状态的栈的栈顶地址。用于保存现场恢复时把这个区域的值写回寄存器，如果时间片用完触发中断要切换进程，要挂起的进程要从这个区域读出寄存器状态保存到自己的进程控制块中。

```
.macro get_sp
        mfc0    k1, CP0_CAUSE
        andi    k1, 0x107C
        xori    k1, 0x1000/*取出异常码 第12位取反*/
        bnez    k1, 1f
        nop
        /* k1==0 是中断异常 */
        li      sp, 0x82000000
        j       2f
        nop
1:
        bltz    sp, 2f
        /* sp>=0 */
        nop
        lw      sp, KERNEL_SP
        nop

2:      nop
        /* 返回 */
```

SAVE_ALL把寄存器内容写到sp+偏移的空间中。

上面代码是判断中断类型并修改sp寄存器。如果分析得到时钟中断时会把寄存器的值放入TIMESTACK的区域。

TIMESTACK是发生时钟中断时固定的栈顶地址，KERNEL_SP是其他中断时的栈顶地址。

Thinking3.9

阅读 kclock_asm.S 文件并说出每行汇编代码的作用

```
#include <asm/regdef.h>
#include <asm/cp0regdef.h>
#include <asm/asm.h>
#include <kclock.h>

.macro  setup_c0_status set clr
        .set    push
        mfc0    t0, CP0_STATUS
        or      t0, \set|\clr
        xor     t0, \clr
        mtc0    t0, CP0_STATUS
        .set    pop
.endm

        .text
LEAF(set_timer)

        li t0, 0x01
        sb t0, 0xb5000100/* 向0xb5000100位置写入1 */
        sw      sp, KERNEL_SP/* 把sp寄存器的值写到KERNEL_SP */
setup_c0_status STATUS_CU0|0x1001 0 /* 把CP0_STATUS第0位第12位置1 */
        jr ra

        nop
END(set_timer)
```

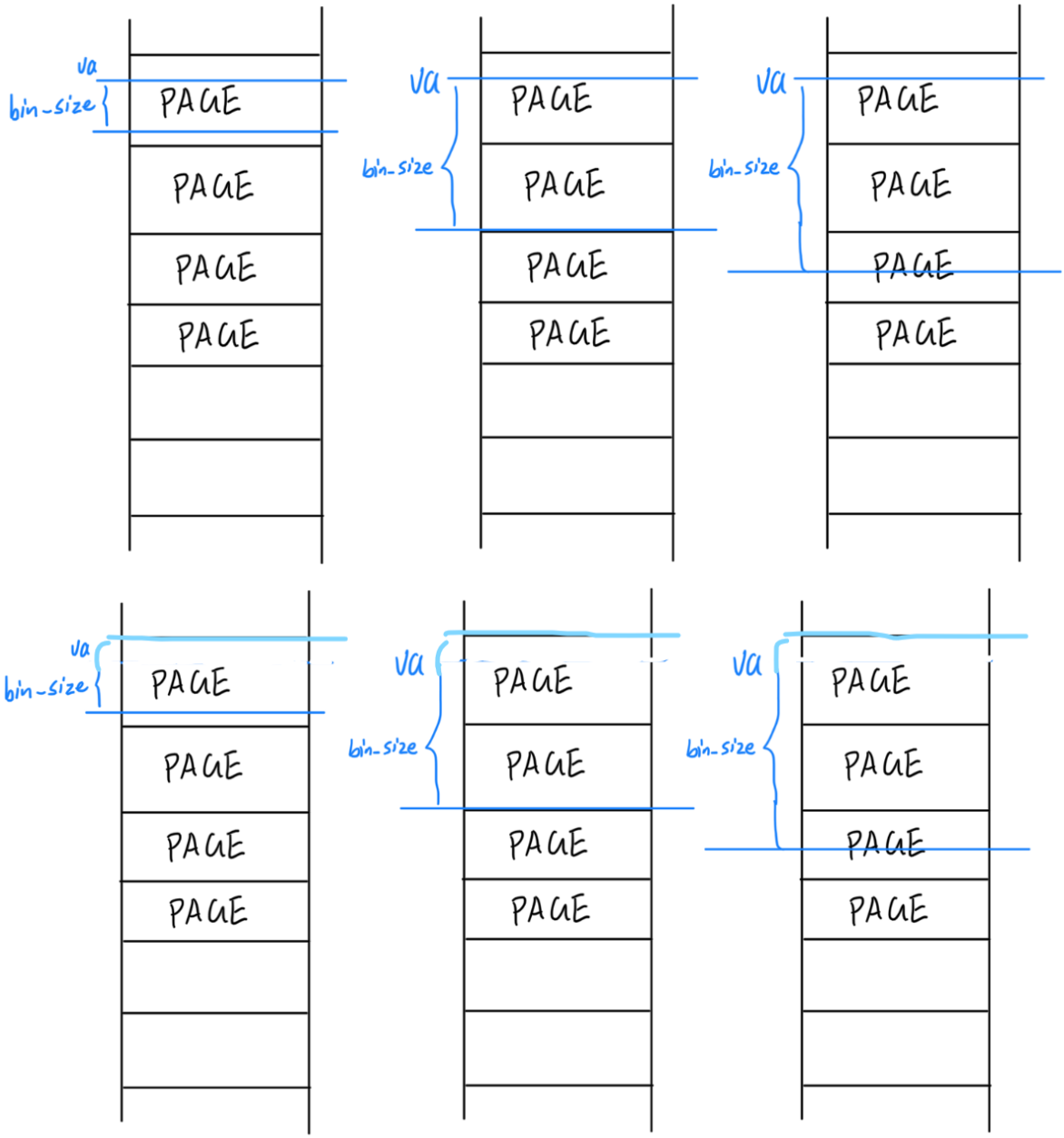
Thinking3.10

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。

env_sched_list有两个链表存放就绪进程。每当一个进程状态变成ENV_RUNNABLE，我们就把它插入第一个就绪状态进程链表。调用sched_yield 函数时, 先判断当前时间片是否用完。如果用完,将其插入另一个就绪状态进程链表结尾。之后判断当前就绪状态进程链表是否为空。如果为空, 将指针切换到另一个就绪状态进程链表。

实验难点图示

> load_icode_mapper的多情况讨论



以下是一个说明：

```
        va
        ↑                                                   |                 |
|_ _ _|___|__BY2PG__|__BY2PG__|__BY2PG__|_____|__|_________|___|_ _ _|
offset|                                                     |                 |
        |<----------- .text & .data ------------>|<---- .bss ---->|
        |<-------------- bin_size -------------->|                 |
        |<----------------------- sg_size ----------------------->|

                        这段icode需要分配7个页面
```

几种情况考虑周全后难度就比较小了。

> sched_yield调度算法实现

sched_yield每执行一次，相应进程消耗一个时间片，如果当前有进程并且进程的时间片没有用完，则需要继续执行当前进程，并扣除一个时间片。如果当前没有进程（初始时刻）或者当前进程的时间片用完，则需要（若有进程）将当前进程放入另一个sched_list的末尾，如果当前的sched_list为空则需要切换到另一个sched_list，然后再取出一个进程并从sched_list移出，设置要执行的时间片，执行。

体会与感想

lab3的env管理和lab2的page管理比较相似，加上lab2中对“谜语”链表已经有了深刻的认识，所以前半部分实现起来并不是很困难，多看几遍熟悉函数的功能即可。env结构体中项目比较多，花了许多时间理解env的各个项目，并知道什么时候需要设置。

中断与异常在计组中已经从硬件层面有所接触，并也用汇编写过部分代码，所以理解代码功能不是很困难。异常识别与分发、保存恢复现场等内容的汇编代码看起来比较费事。汇编代码的语法很多和计组不同，让人读起来真的大费精力。

理解代码的每个细节很困难，但是从宏观上把握代码并且熟悉不同函数之间的调用关系更加困难，这是在一遍遍读代码的过程中逐步建立的，lab3写代码花费的时间并不是很多，大量的时间用在读代码与debug。

指导书反馈

补充代码的部分可以把预先给的几个定义好的变量说明一下。

函数load_icode_mapper中的只给了for循环，让人感觉比较迷惑，其实for循环前面也要填部分代码，感觉这一部分让我感到误导。其实这一部分的实现有比较大的灵活性，用while和for都可以，建议这一部分不再提供代码框架，而是注释里面对复制的多情况进行一下提示。

进程调度sched_yield函数注释里面直接出现了e这个变量，让人感觉无从下手。在实现这一部分时，用到了全局变量curenv，建议注释里面对curenv加一部分说明。

残留难点

异常分发与异常处理部分的汇编代码还需要继续加深理解