

操作系统lab3实验报告

姓名：高远 学号：19374242 班级：202113

实验思考题

Thinking 3.1:

在该判断语句的之前有一句 $e = \text{envs} + \text{ENVX}(\text{envid})$ ，它通过 envid 来求出 env 。可以发现，在这个语句中判断对应关系只用了 envid 的低位位数，也就是进程块的偏移。但实际上， envid 还有高位部分，由于一个进程块同时只能对标一个正在执行的进程，所以若高位不同，代表所查询的 envid 所对应的进程一定不存在，因此返回 -E_BAD_ENV 。若没有这步判断，则会在查询一个不存在的进程 id 时却能够得到对应的进程，导致程序错误。

Thinking 3.2:

1. $\text{UTOP} = 0x7f400000$ ，其含义为用户所能操纵的地址空间的最大值； $\text{ULIM} = 0x80000000$ ，其含义为操作系统分配给用户地址空间的最大值。
2. UVPT 的含义为 User Virtual Page Table，因此这一段需要映射到他的进程在 pgdir 中的页目录地址。所以我们在将这一段空间的虚拟地址转化为物理地址时可以很快找到对应的页目录。
3. 进程只能操作虚拟地址，而物理地址是共同使用的真实地址，因此需要操作系统来完成实现虚拟地址和物理地址之间的映射。

Thinking 3.3:

user_data 在此处被使用，

```
static int load_icode_mapper(u_long va, u_int32_t sgsize,
                             u_char *bin, u_int32_t bin_size, void *user_data)
{
    struct Env *env = (struct Env *)user_data;
```

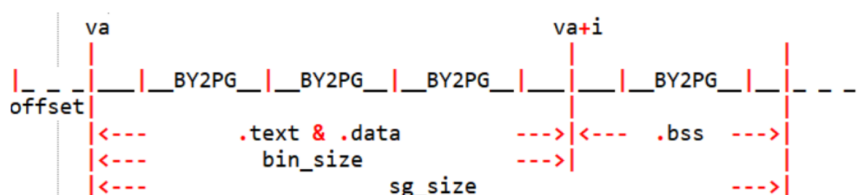
其真正含义是这个被操作的进程指针，它来源于下图，其中的 e 则为传入 load_icode 中的 struct Env $*e$

```
r = load_elf(binary, size, &entry_point, e, load_icode_mapper);
```

不能没有这个参数，没有进程指针，我们的加载镜像的步骤显然不能正常完成。

Thinking 3.4:

最差情况：



1. $\text{bin_size} < \text{BY2PG} - \text{offset}$ (填入的文件内容不能充满一页中的剩余位置)
2. $\text{bin_size} \geq \text{BY2PG} - \text{offset}$ 并且 $(\text{bin_size} - \text{BY2PG} + \text{offset}) \% \text{BY2PG} \neq 0$ (可以充满剩余位置，但是文件内容末尾不是 BY2PG 的整数倍)

3. $\text{bin_size} \geq \text{BY2PG} - \text{offset}$ 并且 $(\text{bin_size} - \text{BY2PG} + \text{offset}) \% \text{BY2PG} = 0$ (可以充满且是整数倍)
4. 均考虑到了

Thinking 3.5:

1. 针对的是虚拟空间

2. 该值是从 `load_elf` 中的 `*entry_point = ehdr->e_entry;` 语句中进行赋值, 所以该值对于每个进程是一样的。这种值来源于他们都是从ELF文件中的同一个部分进行取值的, elf文件结构的统一决定了该值的统一。

Thinking 3.6:

上面提到的epc值是 `curenv->env_tf.cp0_epc`。因为EPC寄存器就是用来存放异常中断发生时进程正在执行的指令的地址, 这样可以在处理完中断之后返回到之前的位置继续执行

Thinking 3.7:

1. 如图, TIMESTACK在 `env_destroy` 中被用到, 将 `(void *)KERNEL_SP - sizeof(struct Trapframe)` 存到了 TIMESTACK 区域

```
/* Hint: schedule to run a new environment. */
if (curenv == e) {
    curenv = NULL;
    /* Hint: Why this? */
    bcopy((void *)KERNEL_SP - sizeof(struct Trapframe),
          (void *)TIMESTACK - sizeof(struct Trapframe),
          sizeof(struct Trapframe));
    printf("I am killed ... \n");
    sched_yield();
}
```

2. 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 3.8:

具体位置在 `lib/genex.S` 中

Thinking 3.9:

```
.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, 0xc8
sb t0, 0xb5000100
sw sp, KERNEL_SP
setup_c0_status STATUS_CU0|0x101 0
jr ra

nop
END(set_timer)
```

set_timer: 如图, 首先先将 `0xc8` 写入地址 `0xb5000100` 中, 其中基地址 `0xb5000000` 为 `gxemu1` 用于映射实时钟的地址, 偏移量 `0x100` 代表时钟的频率。将栈指针设置为 `KERNEL_SP` 中能够正确产生时钟中断, 二、再调用宏函数 `setup_c0_status` 来设置 `CP0_STATUS` 的值, 最后通过 `jr ra` 来返回。

```
timer_irq:
sb zero, 0xb5000100
i: j sched_yield
nop
/*li t1, 0xc8f
lw t0, delay
addu t0, 1
sw t0, delay
beq t0, t1, if
nop*/
j ret_from_exception
nop
```

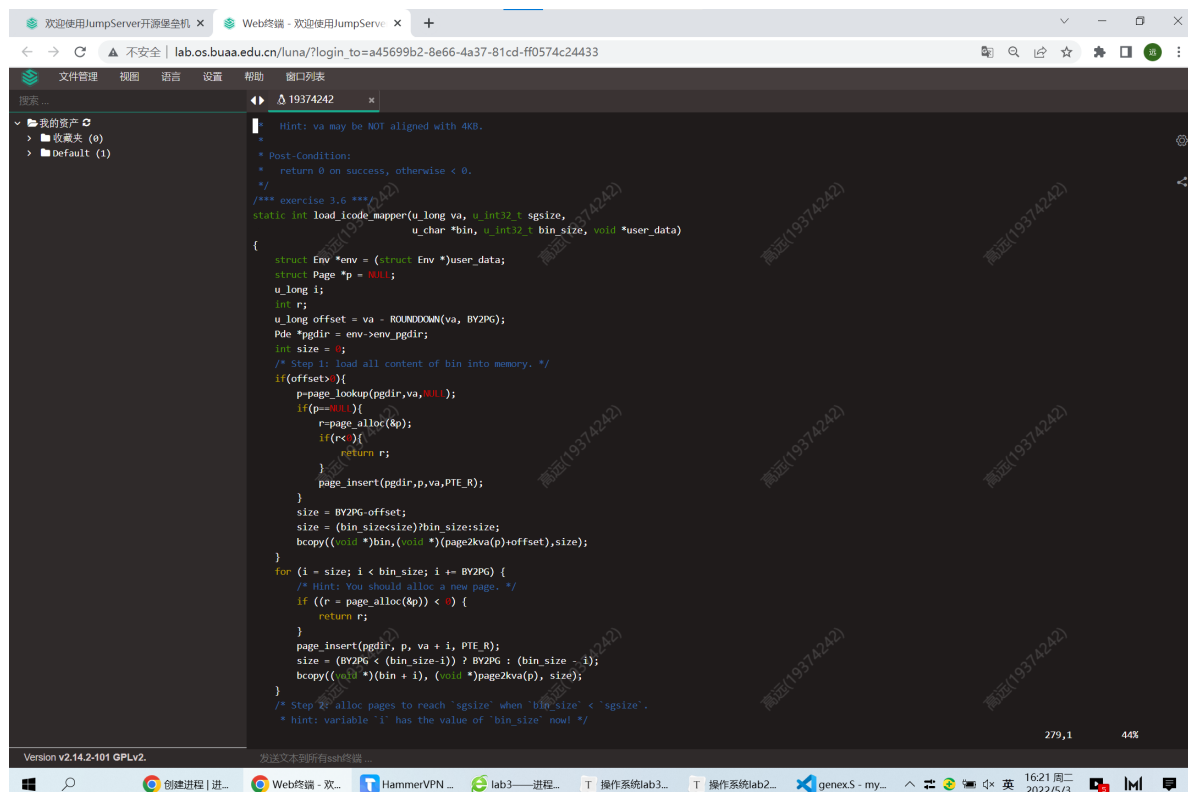
timer_irq: 如图, 先将0xb5000110置零, 然后直接跳转到sched_yield 中执行。

Thinking 3.10:

在我们的操作系统中, 设置了一个进程就绪队列, 并且给每一个进程添加了一个时间片, 这个时间片起到计时的作用, 一旦时间片的时间走完, 则代表该进程需要执行时钟中断操作, 则再将这个进程移动到就绪队列的尾端, 并复原其时间片, 再让就绪队列最首端的进程执行相应的时间片段, 按照这种规律实现循环往复, 从而做到根据时钟周期切换进程。

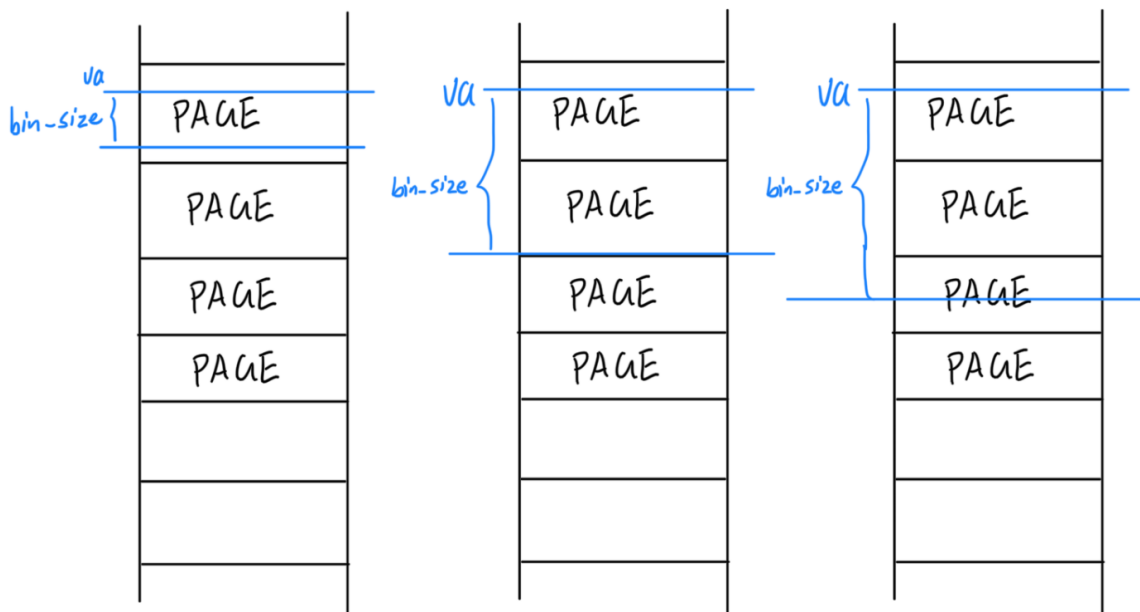
实验难点图示

难点1: load_icode_mapper



```
/* Hint: va may be NOT aligned with 4KB. */
/* Post-Condition:
 * return 0 on success, otherwise < 0.
 */
/** exercise 3.6 */
static int load_icode_mapper(u_long va, u_int32_t sgsz,
                           u_char *bin, u_int32_t bin_size, void *user_data)
{
    struct Env *env = (struct Env *)user_data;
    struct Page *p = NULL;
    u_long i;
    int r;
    u_long offset = va - ROUNDOWN(va, BY2PG);
    Pde *pgdir = env->env_pgdir;
    int size = 0;
    /* Step 1: load all content of bin into memory. */
    if (offset > 0) {
        p = page_lookup(pgdir, va, 0);
        if (p == NULL) {
            r = page_alloc(&p);
            if (r < 0) {
                return r;
            }
            page_insert(pgdir, p, va, PTE_R);
        }
        size = BY2PG - offset;
        size = (bin_size < size) ? bin_size : size;
        bcopy((void *)bin, (void *)page2kva(p) + offset, size);
    }
    for (i = size; i < bin_size; i += BY2PG) {
        /* Hint: You should alloc a new page. */
        if ((r = page_alloc(&p)) < 0) {
            return r;
        }
        page_insert(pgdir, p, va + i, PTE_R);
        size = (BY2PG < (bin_size - i)) ? BY2PG : (bin_size - i);
        bcopy((void *)bin + i, (void *)page2kva(p), size);
    }
    /* Step 2: alloc pages to reach 'sgsz' when 'bin_size' < 'sgsz'.
     * hint: variable 'i' has the value of 'bin_size' now! */
}
```

load_icode_mapper 加载该段在ELF 文件中的所有内容到内存, 如果该段在文件中的内容的大小达不到为填入这段内容新分配的页面大小, 那么余下的部分用0来填充。该函数有三种情况



因此，该函数需要考虑三种对其的情况，因为一开始没有想明白，花费了我大量的时间

难点2: sched_yield

```
void sched_yield(void)
{
    static int count = 0; // remaining time slices of current env
    static int point = 0; // current env_sched_list index
    static struct Env *e = NULL;
    /* hint:
     * 1. if (count==0), insert 'e' into 'env_sched_list[1-point]'
     *    using LIST_REMOVE and LIST_INSERT_TAIL.
     * 2. if 'env_sched_list[point]' is empty, point = 1 - point;
     *    then search through 'env_sched_list[point]' for a runnable env 'e',
     *    and set count = e->env_pri.
     * 3. count--
     * 4. env_run()
     */
    /* functions or macros below may be used (not all):
     * LIST_INSERT_TAIL, LIST_REMOVE, LIST_FIRST, LIST_EMPTY
     */
    if (count == 0 || e == NULL || e->env_status != ENV_RUNNABLE)
    {
        if (e != NULL)
        {
            LIST_REMOVE(e, env_sched_link);
            if (e->env_status != ENV_FREE)
            {
                LIST_INSERT_TAIL(&env_sched_list[1 - point], e, env_sched_link);
            }
        }
        while (!)
        {
            while (LIST_EMPTY(&env_sched_list[point]))
            {
                point = 1 - point;
                e = LIST_FIRST(&env_sched_list[point]);
                if (e->env_status == ENV_FREE)
                {
                    LIST_REMOVE(e, env_sched_link);
                }
                else if (e->env_status == ENV_NOT_RUNNABLE)
                {
                    LIST_REMOVE(e, env_sched_link);
                    LIST_INSERT_TAIL(&env_sched_list[1 - point], e, env_sched_link);
                }
            }
            else
            {
                count = e->env_pri;
                e->env_status = ENV_RUNNABLE;
                env_run(e);
            }
        }
    }
}
```

sched_yield就是时间片轮转的算法。env 中的优先级在这里起到了作用，我们规定其数值表示进程每次运行的时间片数量。不过寻找就绪状态进程不是简单遍历进程链表，而是用两个链表存储所有参与调度进程。当进程被创建时，我们要将其插入第一个进程调度链表的头部。调用 sched_yield函数时，先判断当前时间片是否用完。如果用完，将其插入另一个进程调度链表的尾部。之后判断当前进程调度链表是否为空。如果为空，切换到另一个进程调度链表。

根据调度队列的用途，我们需要在创建进程时将该 env 插入到第一个队列中。相应的，我们需要在其被销毁 (env_destroy) 时，将其从队列中移除。

我一开始对count和point两个参数百思不得其解，再请教了助教后才明白过来，这个函数花费了我大量的时间。

体会与感想

就我而言，本次实验给我的挑战远远大于lab2和lab1，尤其是进程的创建和时间片轮转算法，给我带来了极大的困惑，我也花费了数倍于前几个实验的时间进行学习。因此，做出lab3也给我带来了极大的收获。因为看不懂题目中的函数，我大量阅读源码，对进程初始化、分配以及时间片轮转算法的使用有了更为深刻的了解。同时，在填写一些复杂度很高的函数，如load_icode_mapper时，不能拿到代码，看到hint就往上一股脑的填，而应该去尝试复盘整个过程，并考虑到各种情况，对每个情况进行处理。与此同时，我也复习巩固了指针的使用方法，可以说，lab3实验使我受益匪浅。

指导书反馈

本次指导书个人认为可读性没有前几次实验那么好，很多问题都没有实实在在进行解答，而让我们自己思考，这样花费了我很多无用的精力，个人认为指导书可以写的更加详细一些。

残留难点

对中断时的汇编代码还没有非常了解，希望能够通过后续的学习进行巩固。