操作系统 Lab3实验报告

实验思考题

Thinking 3.1

因为每个 `env_id` 只对应一个进程，而在通过代码 `e = &envs[ENVX(envid)];` 获取到的进程 `e` 有可能已经发生了替换，此时通过 `envid` 获取到的进程与此进程id不匹配。如果没有这步判断，则会通过进程id访问到某个错误的进程，导致程序错误。

Thinking 3.2

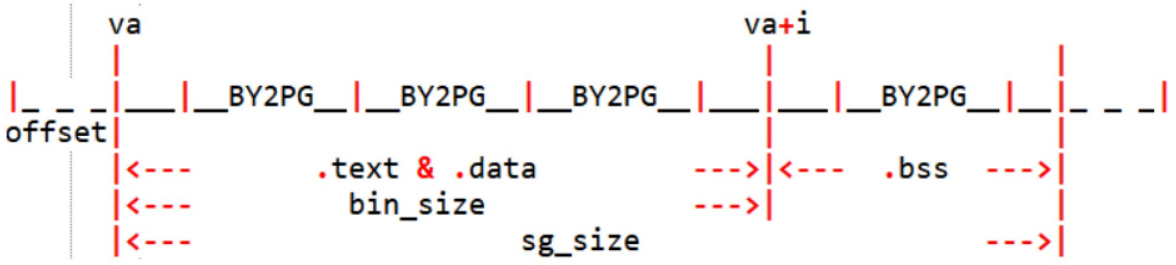
- `UTOP` 对应的地址为 `0x7f40 0000` ，是用户能够操纵的空间的地址最高值； `ULIM` 对应的地址为 `0x8000 0000` ，是操作系统分配给用户进程空间的地址最高值，同时也是内核进程空间的地址最低值。 `UTOP` 和 `ULIM` 之间的空间对于用户是只读的，存储进程信息，页表信息等内容。
- `env_cr3` 中保存了当前进程页目录的物理地址， `UVPT` 表示用户虚页表的起始地址，需要映射到进程页目录的物理地址，而 `pgdir[PDX(UVPT)]=env_cr3` 这样赋值正好找到了对应的页目录物理地址，完成了页目录的自映射。
- 每个进程读写的都是虚拟地址，操作系统将进程的虚拟地址与物理地址实现映射，这样操作能够保证不同进程使用不同的虚拟地址空间，在操作系统的调度下不会产生地址冲突。

Thinking 3.3

`user_data` 在 `load_icode_mapper` 函数中作为当前进程的指针用以获取页目录地址。 `load_icode` 函数在调用 `load_elf` 函数解析ELF文件时传入了 `load_icode_mapper` 函数和作为参数传入 `load_icode` 的进程块 `e` 。 `load_icode` 函数则由 `env_create_priority` 函数调用，传入的进程块 `e` 也是在 `env_create_priority` 中被创建并分配空间的。在创建新的进程块 `e` 时，结构体中存储了该进程页目录的内核虚拟地址，而这个地址在将ELF文件的各个段加载到内存时需要被用到，因此这个参数在整个函数调用的体系中是必要的，否则将无法加载ELF文件到正确的位置。

在C语言 `stdlib.h` 库中声明了快速排序的函数 `void qsort(void*base, size_t num, size_t width, int(__cdecl*compare)(const void*,const void*));` ，其中参数 `size_t width` 就是帮助函数分隔需要排序的数据元素，方便其进行比较排序。

Thinking 3.4



- 若 `offset != 0` 则需要从 `offset` 位置开始装载；
- 若 `BY2PG - offset >= bin_size` ，则在上一步中已完成 `bin_size` 部分的装载，否则需要将 `bin_size` 循环装载到若干页中。
- 由于 `.bss` 部分保存未初始化的全局变量和静态变量，因此不需要对这部分进行内容拷贝，而只为其分配页面空间即可。

最终代码结构如下：

```
/* Step 1: load all content of bin into memory. */
/*** load first page starting from `offset` ***/
if (offset) {
    r = page_alloc(&p);
    if (r) return r;
    bcopy(bin, page2kva(p) + offset, MIN(BY2PG - offset, bin_size));
    page_insert(env->env_pgdir, p, va, PTE_R);
}
/*** load the rest part of .text&.data ***/
for (i = MIN(BY2PG - offset, bin_size); i < bin_size; i += BY2PG) {
    /* Hint: You should alloc a new page. */
    r = page_alloc(&p);
    if (r) return r;
    bcopy(bin + i, page2kva(p), MIN(BY2PG, bin_size - i));
    page_insert(env->env_pgdir, p, va + i, PTE_R);
}
/* Step 2: alloc pages to reach `sgsize` when `bin_size` < `sgsize`.
     * hint: variable `i` has the value of `bin_size` now! */
/*** load .bss ***/
for ( ; i < sgsize; i += BY2PG) {
    /**/
    r= page_alloc(&p);
    if (r) return r;
    page_insert(env->env_pgdir, p, va + i, PTE_R);
}
```

Thinking 3.5

- 虚拟地址。在程序指令执行过程中，pc寄存器将会不断进行+4操作，而一个进程有可能被存储在不连续的物理空间中，因此这里 `env_tf.pc` 存储的是虚拟地址。
- `entry_point` 在 `load_elf` 函数中被赋值为 `ehdr->e_entry` ，即ELF文件入口的虚地址，这对于每个进程来说是相同的，均从ELF文件头中读出，但该虚拟地址在不同进程中映射到的物理地址可能是不同的。

Thinking 3.6

`epc` 是指异常返回地址寄存器，记录异常/中断结束后程序恢复执行的位置。因此将保存进程上下文的 `env_tf` 结构体中的 `pc` 值设置为 `epc` ，从而保证当这个进程被重新打开时，CPU能够回到相应指令处继续执行。

Thinking 3.7

- 操作系统在发生中断时将当前进程环境下CPU寄存器的状态保存在 `TIMESTACK` 中。
- `TIMESTACK` 是发生时钟中断异常时用到固定的的栈指针； `KERNEL_SP` 是非时钟中断异常用的栈指针。

Thinking 3.8

- `handle_reserved` ：在 `lib/genex.S` 中通过 `do_reserved` 实现

```
LEAF(do_reserved)
END(do_reserved)
```

- `handle_int` ：在 `lib/genex.S` 中通过 `handle_int` 实现

```
NESTED(handle_int, TF_SIZE, sp)
.set    noat

//1: j 1b
nop

SAVE_ALL
CLI
.set    at
mfc0    t0, CP0_CAUSE
mfc0    t2, CP0_STATUS
and t0, t2

andi    t1, t0, STATUSF_IP4
bnez    t1, timer_irq
nop
END(handle_int)
```

- `handle_mod`：在 `lib/traps.c` 中通过 `page_fault_handler` 实现

```
void
page_fault_handler(struct Trapframe *tf)
{
    u_int va;
    u_int *tos, d;
    struct Trapframe PgTrapFrame;
    extern struct Env * curenv;
    //printf("^^^^cp0_BadVAddress:%x\n",tf->cp0_badvaddr);

    bcopy(tf, &PgTrapFrame,sizeof(struct Trapframe));
    if(tf->regs[29] >= (curenv->env_xstacktop - BY2PG) && tf->regs[29]
<= (curenv->env_xstacktop - 1))
    {
        //panic("fork can't nest!!");
        tf->regs[29] = tf->regs[29] - sizeof(struct  Trapframe);
        bcopy(&PgTrapFrame, tf->regs[29], sizeof(struct Trapframe));
    }
    else
    {

        tf->regs[29] = curenv->env_xstacktop - sizeof(struct
 Trapframe);
        //printf("page_fault_handler(): bcopy(): src:%x\tdes:%x\n",
(int)&PgTrapFrame,(int)(curenv->env_xstacktop - sizeof(struct
Trapframe)));

        bcopy(&PgTrapFrame, curenv->env_xstacktop - sizeof(struct
 Trapframe), sizeof(struct Trapframe));
    }
    //printf("^^^^cp0_epc:%x\tcurenv->env_pgfault_handler:%x\n",tf-
>cp0_epc,curenv->env_pgfault_handler);

    tf->cp0_epc = curenv->env_pgfault_handler;
    return;
}
```

- `handle_tlb`：在 `lib/genex.S` 中通过 `do_refill` 实现

```
NESTED(do_refill,0 , sp)
            //li    k1, '?'
            //sb    k1, 0x90000000
            .extern mCONTEXT
//this "1" is important
1:          //j 1b
            nop
            lw      k1,mCONTEXT
            and     k1,0xfffff000
                mfc0        k0,CP0_BADVADDR
                srl     k0,20
                and     k0,0xfffffffc
            addu        k0,k1

            lw      k1,0(k0)
            nop
                    move        t0,k1
                    and     t0,0x0200
                    beqz        t0,NOPAGE
            nop
            and     k1,0xfffff000
                mfc0        k0,CP0_BADVADDR
                srl     k0,10
                and     k0,0xfffffffc
                and     k0,0x00000fff
            addu        k0,k1

            or      k0,0x80000000
            lw      k1,0(k0)
            nop
                    move        t0,k1
                    and     t0,0x0200
                    beqz        t0,NOPAGE
            nop
            move        k0,k1
            and     k0,0x1
            beqz        k0,NoCOW
            nop
            and     k1,0xfffffbff
NoCOW:
            mtc0        k1,CP0_ENTRYLO0
            nop
            tlbwr

            j       2f
            nop
NOPAGE:
//3: j 3b
nop
            mfc0        a0,CP0_BADVADDR
            lw      a1,mCONTEXT
            nop

            sw      ra,tlbra
            jal     pageout
            nop
//3: j 3b
```

```
nop
            lw      ra,tlbra
            nop

            j   1b
2:          nop

            jr      ra
            nop
END(do_refill)
```

- `handle_sys`：在 `syscall.S` 中通过 `handle_sys` 实现

```
NESTED(handle_sys,TF_SIZE, sp)

SAVE_ALL
CLI

//1: j 1b
nop
.set at
lw t1, TF_EPC(sp)
sw      t1, TF_EPC(sp)
la      t1, sys_call_table
lw      t2, (t1)
lw      t0,TF_REG29(sp)

lw      t1, (t0)
lw      t3, 4(t0)
lw      t4, 8(t0)
lw      t5, 12(t0)
lw      t6, 16(t0)
lw      t7, 20(t0)

subu    sp, 20

sw      t1, 0(sp)
sw      t3, 4(sp)
sw      t4, 8(sp)
sw      t5, 12(sp)
sw      t6, 16(sp)
sw      t7, 20(sp)

move    a0, t1
move    a1, t3
move    a2, t4
move    a3, t5

jalr    t2
nop

addu    sp, 20

sw      v0, TF_REG2(sp)

j       ret_from_exception//extern?
nop
```

```
45
46 illegal_syscall: j illegal_syscall
47             nop
48 END(handle_sys)
```

Thinking 3.9

- `set_timer`

```
1  LEAF(set_timer)
2
3      li t0, 0xc8 // 将 0xc8 存入 t0 寄存器
4      sb t0, 0xb5000100 // 将 t0 寄存器的值存入地址 0xb5000100
5      sw  sp, KERNEL_SP // 将栈指针 sp 的值存入KERNEL_SP
6  setup_c0_status STATUS_CU0|0x1001 0 // 把 CP0_STATUS 第0位和第12位置1
7      jr ra // 返回
8
9      nop
10 END(set_timer)
```

- `time_irq`

```
1  timer_irq:
2
3      sb zero, 0xb5000110 // 将 0 存入地址 0xb5000110
4  1:  j   sched_yield // 跳转到 sched_yield 函数
5      nop
6      /*li t1, 0xff
7      lw    t0, delay
8      addu  t0, 1
9      sw  t0, delay
10     beq t0,t1,1f
11     nop*/
12     j   ret_from_exception // 跳转到 ret_from_exeception 函数
13     nop
```

Thinking 3.10

- 系统中存在两个 `env_sched_list` 存储所有参与调度的进程。
- 当进程被创建时，将其插入第一个进程调度链表。
- 调用 `sched_yield` 函数实现对进程的调度，判断当前时间片是否用完，如果用完，则将其插入到另一个进程调度链表。
- 判断当前进程调度链表是否为空，如果为空，则切换到另一个进程调度链表。
- 由此循环，当两个进程调度链表都为空时，执行结束。

实验难点

SR 状态寄存器

SR Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CU3	CU2	CU1	CU0	0		RE	0		BEV	TS	PE	CM	PZ	SwC	IsC

15	8	7	6	5	4	3	2	1	0
IM		0		KUo	IEo	KUp	IEp	KUc	IEc

`SR` 寄存器的第六位是一个二重栈结构：

- `KU` 表示是否位于内核模式下， `IE` 表示中断是否开启。
- `o` 为 old ， 表示过期状态；
- `p` 为 previous ， 表示上一状态；
- `c` 为 current ， 表示当前状态 。

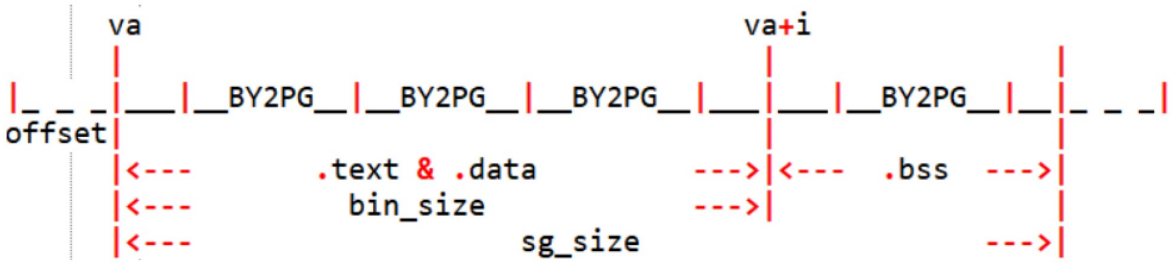
每当中断发生的时候，硬件自动会将 `KUp` 和 `IEp` 的数值拷贝到 `KUo` 和 `IEo` ； `KUp` 和 `IEp` 是一组，当中断发生的时候，硬件会把 `KUc` 和 `IEc` 的数值拷贝到这里。

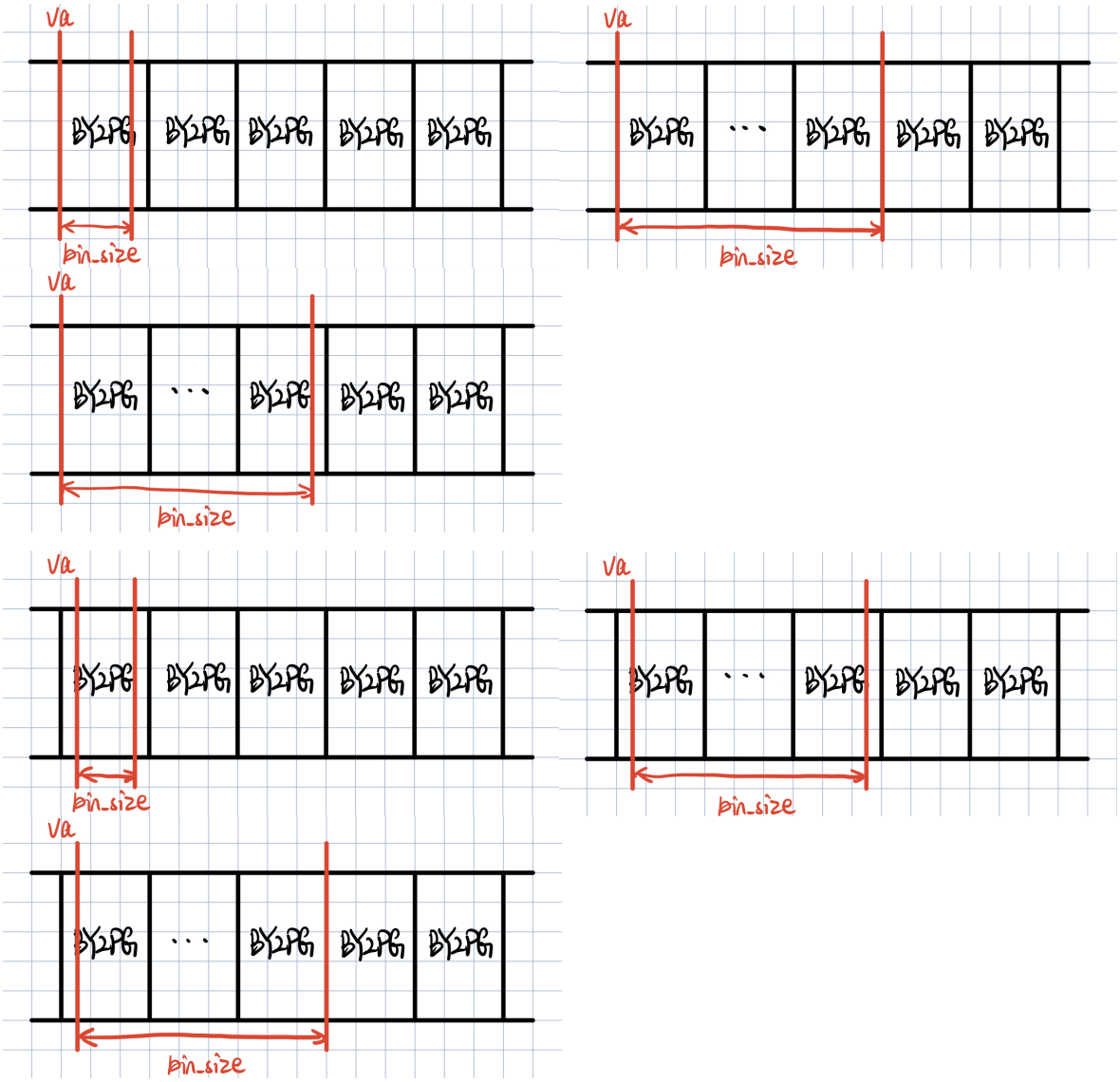
而每当 `rfe` 指令调用时，就会进行上面操作的逆操作。

加载二进制镜像

加载二进制镜像共涉及到 `load_icode` 、 `loac_elf` 、 `load_icode_mapper` 三个函数，它们之间的调用关系为 `load_icode -> loac_elf -> load_icode_mapper` 。

- `load_icode`
  - 申请一个物理页 `p` ，通过 `pagr_insert` 函数将其对应到 `USTACKTOP - BY2PG` 这 $4KB$ 的空间，从而初始化一个用户进程栈。
  - 调用 `load_elf` 函数将完整的二进制镜像加载到对应的位置。
  - 将进程块中对应 `pc` 寄存器的值设置为代码入口地址。
- `load_elf`
  - 判断二进制镜像是否为 ELF 文件
  - 借助 ELF 文件头解析出文件的相关信息。
  - 通过 `load_icode_mapper` 函数将 ELF 文件的每个 `segment` 加载到相应的虚拟地址。
- `load_icode_mapper`
  - 考虑每个 `segment` 加载位置的不同情况，将其正确加载到对应的虚页中。





进程调度

进程调度过程中使用到了 `sched_yield` 函数，实现了基于时间片的进程调度。

```
// 当时间片为0，或进程为空，或进程不处于执行或就绪状态时，切换新的进程
if (count == 0 || e == NULL || e->env_status != ENV_RUNNABLE) {
    if (e != NULL) { // 进程为空时
        LIST_REMOVE(e, env_sched_link); // 将进程从当前调度链表中移除
        LIST_INSERT_TAIL(&env_sched_list[1 - point], e, env_sched_link); // 将进程插入到另一个调度链表尾部
    }
    while(1) {
        if (LIST_EMPTY(&env_sched_list[point])) { // 当前调度链表为空时，切换到另一个调度链表
            point = 1 - point;
        }
        e = LIST_FIRST(&env_sched_list[point]); // 取出调度链表的首个元素
        if (e->env_status == ENV_FREE) { // 若进程不活动，则将其从调度链表中移出
            LIST_REMOVE(e, env_sched_link);
        } else if (e->env_status == ENV_NOT_RUNNABLE) { // 若进程阻塞，则将其从本调度链表移至另一链表尾
            LIST_REMOVE(e, env_sched_link);
            LIST_INSERT_TAIL(&env_sched_list[1 - point], e, env_sched_link);
        } else { // 若进程处于执行或就绪状态，则将时间片设置为其优先级，并退出循环
            count = e->env_pri;
```

```
            break;
        }
    }
}
count--; // 时间片递减
env_run(e); // 启动进程
```

体会感想

与lab2情况相似，在完成lab3课下部分时绝大部分时间都是花在了阅读指导书和理解代码上，随着MOS操作系统的功能越来越完善，系统的复杂度也大幅增加，函数间调用越来越复杂，同时应用了大量的宏定义和外部变量，无疑提高了代码的阅读和理解难度。在完成实验时，常常需要在gitlab平台打开数个窗口分别显示不同的文件内容，一边阅读代码一边翻阅有关函数和定义，深刻考验着我对系统的整体把握能力。

在本次实验中的中断与异常部分涉及到了计组课程中的知识，需要将操作系统的知识与前置计组知识进行联系，才能彻底理解操作系统处理中断的流程。同时，在本次需要完成的代码片段中也涉及到了一部分汇编代码，虽然并不需要我们完全自主编写，但想要完全理解其实现的功能仍然有一定困难。这需要我之后继续温习汇编语言的相关知识，理解相关指令，才能更好地认识、理解、完成课程设计。

残留难点

- 尚未完全理解 `handle_reserved` ， `handle_int` ， `handle_mod` ， `handle_tlb` ， `handle_sys` 这几个异常处理函数的具体实现方式。