Lab3实验报告

思考题

Thinking 3.1

- `UVPT` 表示用户页表的起始虚拟地址
- `PDX(UVPT)` 表示取其页目录索引号
- `e -> env_pgdir[PDX(UVPT)]` 表示当前进程内存中 `UVPT` 地址的页目录项
- `PADDR(e -> env_pgdir)` 表示当前进程页目录的物理地址
- 整体来说就是当前进程中 `UVPT` 地址的页目录项赋值为自己的页目录的物理基地址，加上去权限

Thinking 3.2

`env.c`

```
314         /* Step 1: Use 'elf_from' to parse an ELF header from 'binary'. */
315         const Elf32_Ehdr *ehdr = elf_from(binary, size);
316         if (!ehdr) {
317                 panic("bad elf at %x", binary);
318         }
319
320         /* Step 2: Load the segments using 'ELF_FOREACH_PHDR_OFF' and 'elf_load_seg'.
321          * As a loader, we just care about loadable segments, so parse only program headers here.
322          */
323         size_t ph_off;
324         ELF_FOREACH_PHDR_OFF (ph_off, ehdr) {
325                 Elf32_Phdr *ph = (Elf32_Phdr *)(binary + ph_off);
326                 if (ph->p_type == PT_LOAD) {
327                         // 'elf_load_seg' is defined in lib/elfloader.c
328                         // 'load_icode_mapper' defines the way in which a page in this segment
329                         // should be mapped.
330                         panic_on(elf_load_seg(ph, binary + ph->p_offset, load_icode_mapper, e));
331                 }
332         }
333
334         /* Step 3: Set 'e->env_tf.cp0_epc' to 'ehdr->e_entry'. */
335         /* Exercise 3.6: Your code here. */
336         e -> env_tf.cp0_epc = ehdr -> e_entry;
337 }
```

```c
static int load_icode_mapper(void *data, u_long va, size_t offset, u_int perm, const void *src,
                             size_t len) {
        struct Env *env = (struct Env *)data;
        struct Page *p;
        int r;

        /* Step 1: Allocate a page with 'page_alloc'. */
        /* Exercise 3.5: Your code here. (1/2) */
        try(page_alloc(&p));
        /* Step 2: If 'src' is not NULL, copy the 'len' bytes started at 'src' into 'offset' at this
         * page. */
        // Hint: You may want to use 'memcpy'.
        if (src != NULL) {
                /* Exercise 3.5: Your code here. (2/2) */
                memcpy(page2kva(p) + offset, src, len);
        }

        /* Step 3: Insert 'p' into 'env->env_pgdir' at 'va' with 'perm'. */
        return page_insert(env->env_pgdir, env->env_asid, p, va, perm);
}
```

其中，`data` 来源于 `load_icode` 中通过函数传递的进程控制块指针 `e`

这个参数必须有，不然 `load_icode_mapper` 就无法通过 `page_insert` 将文件插入页表

Thinking 3.3

- 修改权限位
- 虚拟地址页面对齐
- 将段内页映射导物理空间
- 把多余的空间用 0 填充

```c
int elf_load_seg(Elf32_Phdr *ph, const void *bin, elf_mapper_t map_page, void *data) {
        u_long va = ph->p_vaddr;
        size_t bin_size = ph->p_filesz;
        size_t sgsize = ph->p_memsz;
        u_int perm = PTE_V;
        if (ph->p_flags & PF_W) {
                perm |= PTE_D;
        }

        int r;
        size_t i;
        u_long offset = va - ROUNDDOWN(va, PAGE_SIZE);
        if (offset != 0) {
                if ((r = map_page(data, va, offset, perm, bin,
                                  MIN(bin_size, PAGE_SIZE - offset))) != 0) {
                        return r;
                }
        }

        /* Step 1: load all content of bin into memory. */
        for (i = offset ? MIN(bin_size, PAGE_SIZE - offset) : 0; i < bin_size; i += PAGE_SIZE) {
                if ((r = map_page(data, va + i, 0, perm, bin + i, MIN(bin_size - i, PAGE_SIZE))) !=
                    0) {
                        return r;
                }
        }

        /* Step 2: alloc pages to reach `sgsize` when `bin_size` < `sgsize`. */
        while (i < sgsize) {
                if ((r = map_page(data, va + i, 0, perm, NULL, MIN(sgsize - i, PAGE_SIZE))) != 0) {
                        return r;
                }
                i += PAGE_SIZE;
        }
        return 0;
}
```

Thinking 3.4

- 虚拟地址

因为程序计数器始终指向虚拟地址，由MMU来负责将其转换为物理地址进行访问，若 `cp0_epc` 存储物理地址，CPU需关闭MMU直接访问物理地址

Thinking 3.5

`kern/traps.c`

```
 6 extern void handle_int(void);
 7 extern void handle_tlb(void);
 8 extern void handle_sys(void);
 9 extern void handle_mod(void);
10 extern void handle_reserved(void);
11
12 void (*exception_handlers[32])(void) = {
13     [0 ... 31] = handle_reserved,
14     [0] = handle_int,
15     [2 ... 3] = handle_tlb,
16 #if !defined(LAB) || LAB >= 4
17     [1] = handle_mod,
18     [8] = handle_sys,
19 #endif
20 };
21
22 /* Overview:
23  *   The fallback handler when an unknown exception code is encountered.
24  *   'genex.S' wraps this function in 'handle_reserved'.
25  */
26 void do_reserved(struct Trapframe *tf) {
27         print_tf(tf);
28         panic("Unknown ExcCode %2d", (tf->cp0_cause >> 2) & 0x1f);
29 }
```

`kern/genex.S`

```
20 NESTED(handle_int, TF_SIZE, zero)
21         mfc0    t0, CP0_CAUSE
22         mfc0    t2, CP0_STATUS
23         and     t0, t2
24         andi    t1, t0, STATUS_IM7
25         bnez    t1, timer_irq
26 timer_irq:
27         li      a0, 0
28         j       schedule
29 END(handle_int)
30
31 BUILD_HANDLER tlb do_tlb_refill
32
33 #if !defined(LAB) || LAB >= 4
34 BUILD_HANDLER mod do_tlb_mod
35 BUILD_HANDLER sys do_syscall
36 #endif
```

Thinking 3.6

`kern/entry.S`进入异常处理

```
 9 exc_gen_entry:
10         SAVE_ALL
11         /*
12         * Note: When EXL is set or UM is unset, the processor is in kernel mode.
13         * When EXL is set, the value of EPC is not updated when a new exception occurs.
14         * To keep the processor in kernel mode and enable exception reentrancy,
15         * we unset UM and EXL, and unset IE to globally disable interrupts.
16         */
17         mfc0    t0, CP0_STATUS
18         and     t0, t0, ~(STATUS_UM | STATUS_EXL | STATUS_IE)
19         mtc0    t0, CP0_STATUS
20 /* Exercise 3.9: Your code here. */
21         mfc0    t0, CP0_CAUSE
22         andi    t0, 0x7c
23         lw      t0, exception_handlers(t0)
24         jr      t0
```

`kern/env_asm.S` 重置时钟

```
 7 LEAF(env_pop_tf)
 8 .set reorder
 9 .set at
10         mtc0    a1, CP0_ENTRYHI
11         move    sp, a0
12         RESET_KCLOCK
13         j       ret_from_exception
14 END(env_pop_tf)
```

`kern/genex.S` 打开中断

```
16 FEXPORT(ret_from_exception)
17         RESTORE_ALL
18         eret
```

Thinking 3.7

中断后，系统将PC指向 `0x80000180` ，跳转到 `exc_gen_entry` 代码段，之后进行异常分发 `handle_int`，之后进入 `schedule` 进行进程调度

在 `schedule` 函数中

先获取当前进程的进程控制块，将其可用全局静态变量时间片减一，判断是否达到切换进程的情况，如果当前有进程并话可以运行，则将其加入调度队列尾部，之后从队头取出一个进程，设置优先级等，然后进行切换进程，如果未达到调度情况，则不切换

实验难点

- 主要集中在对于中断的理解，中断流程以及中断前后进程以及内核以及寄存器发生了什么变化，需要多多参悟
- 对于寄存器恢复的理解，`Env` 块中的 `TrapFrame` 的恢复以及保存
- 另外就是连接内核空间与用户空间的桥梁打通了，需要将这两者好好区分

体会与感想

随着MOS操作系统越来越完善，系统的复杂度也是陡增，大量的函数调用以及宏的使用，让同学们对于代码的理解难度提升。这也是我希望能够多花些时间在Lab上进行更多思考的原因，在实验前后好好温习，上机的时候以及课后才能深入理解操作系统的核心!