实验报告

一、实验目的

> **创建一个进程并成功运行**
>
> **实现时钟中断，通过时钟中断内核可以再次获得执行权**
>
> **实现进程调度，创建两个进程，并且通过时钟中断切换进程执行**
>
> 在本次实验中你将运行一个用户模式的进程。你需要使用数据结构进程控制块 `Env` 来跟踪用户进程。通过建立一个简单的用户进程，加载一个程序镜像到进程控制块中，并 让它运行起来。同时，你的 `MIPS` 内核将拥有处理异常的能力。

二、实验步骤

找到lab3的文件

```
jovyan@74846d05e007:~/ouc21020007131-1ab$ git pull
remote: Counting objects: 110, done.
remote: Compressing objects: 100% (62/62), done.
remote: Total 63 (delta 26), reused 0 (delta 0)
Unpacking objects: 100% (63/63), done.
From 192.168.130.193:ouc21020007131-1ab
 * [new branch]      lab2-result -> origin/lab2-result
 * [new branch]      lab3        -> origin/lab3
Already up to date.
jovyan@74846d05e007:~/ouc21020007131-1ab$ git branch -a
  lab0
  lab1
* lab2
  remotes/origin/lab0
  remotes/origin/lab0-result
  remotes/origin/lab1
  remotes/origin/lab1-result
  remotes/origin/lab2
  remotes/origin/lab2-result
  remotes/origin/lab3
jovyan@74846d05e007:~/ouc21020007131-1ab$ ls
boot     include     lib       readelf
drivers  include.mk  Makefile  tags
gxemul   init        mm        tools
jovyan@74846d05e007:~/ouc21020007131-1ab$ git checkout lab3
Branch 'lab3' set up to track remote branch 'lab3' from 'origin'.
Switched to a new branch 'lab3'
jovyan@74846d05e007:~/ouc21020007131-1ab$ ls
boot     include     lib       readelf
drivers  include.mk  Makefile  tags
gxemul   init        mm        tools
jovyan@74846d05e007:~/ouc21020007131-1ab$
```

练习

Exercise 3.1

> **Exercise 3.1** • 修改 pmap.c/mips_vm_init 函数来为 envs 数组分配空间。
>
> - envs 数组包含 NENV 个 Env 结构体成员，你可以参考 pmap.c 中已经写过的 **pages** 数组空间的分配方式。
> - 除了要为数组 envs 分配空间外，你还需要使用 pmap.c 中你填写过的一个内核态函数为其进行段映射，envs 数组应该被 **UENVS** 区域映射，你可以参考**./include/mmu.h**。
>
> ■

理解下面三行代码：

```
envs = (struct Env *)alloc(NENV * sizeof(struct Env), BY2PG, 1);
//为envs数组开辟空间，且是页对齐的，共开辟了NENV个env
n = ROUND(NENV * sizeof(struct Env), BY2PG);
//计算开辟的空间在页对齐后的字节数
boot_map_segment(pgdir, UENVS, n, PADDR(envs), PTE_R);
//把开辟出来的物理地址映射到以UENVS为起始地址的虚拟地址上
```

Exercise 3.2

> Exercise 3.2 仔细阅读注释，填写 **env_init** 函数，注意链表插入的顺序 (函数位于 lib/env.c 中)。 ■

- 初始化 `env_free_list`
- 遍历 `envs` 中的元素，`init` 每个元素（主要是初始化它的状态，标记为 `free`），并将它们以相反的顺序插入 `env_free_list` 中

```
void
env_init(void)
{
    int i;
    /*Step 1: Initial env_free_list. */
    LIST_INIT(&env_free_list);

    /*Step 2: Travel the elements in 'envs', init every element(mainly initial its status, mark it as free)
     * and inserts them into the env_free_list as reverse order. */
    for ( i = NENV - 1; i >= 0; i--) {
        envs[i].env_status = ENV_FREE;
        LIST_INSERT_HEAD(&env_free_list, &envs[i], env_link);
    }
}
```

Exercise 3.3

> Exercise 3.3 仔细阅读注释，完成 env.c/envid2env 函数，实现通过一个 env 的 id 获取该 id 对应的进程控制块的功能。 ■

- 使用 `envid` 给 e 赋值
- 根据 `checkperm` 进行检查

```
int envid2env(u_int envid, struct Env **penv, int checkperm)
{
        struct Env *e;
    /* Hint:
 *      *  If envid is zero, return the current environment.*/
    /*Step 1: Assign value to e using envid. */
        e = envid ? &envs[ENVX(envid)] : curenv;


```

```
        if (e->env_status == ENV_FREE || e->env_id != envid) {
                *penv = 0;
                return -E_BAD_ENV;
        }
    /* Hint:
 *      *  Check that the calling environment has legitimate permissions
 *           *  to manipulate the specified environment.
 *                *  If checkperm is set, the specified environment
 *                     *  must be either the current environment.
 *                          *  or an immediate child of the current
environment.If not, error! */
    /*Step 2: Make a check according to checkperm. */
        if (checkperm && e != curenv && e->env_parent_id != curenv->env_id)
{
                *penv = 0;
                return -E_BAD_ENV;
        }
        *penv = e;
        return 0;
}
```

Exercise 3.4

> **Exercise 3.4** 仔细阅读注释，填写 env_setup_vm 函数

- 为页目录分配一个页面，并添加它的引用。`pgdir` 是 `Env e` 的页目录，给它分配一个值。
- 将 `UTOP` 之前的 `pgdir` 字段清零
- 复制内核的 `boot_pgdir` 到 `pgdir`

```
static int
env_setup_vm(struct Env *e)
{

    int r;
    struct Page *p;
    Pde *pgdir;

    /*Step 1: Allocate a page for the page directory using a function you
completed in the lab2.
       * and add its reference.
       *pgdir is the page directory of Env e, assign value for it. */
    if ( (r = page_alloc(&p)) < 0 ) {/* Todo here*/
                panic("env_setup_vm - page alloc error\n");
                return r;
        }
    ++p->pp_ref;
    e->env_pgdir = pgdir = (Pde *) page2kva(p);
    e->env_cr3 = PADDR(pgdir);



    /*Step 2: Zero pgdir's field before UTOP. */
    bzero(pgdir, PDX(UTOP) * sizeof *pgdir);
```

```
24
25
26      /*Step 3: Copy kernel's boot_pgdir to pgdir. */
27
28      /* Hint:
29       *  The VA space of all envs is identical above UTOP
30       *  (except at VPT and UVPT, which we've set below).
31       *  See ./include/mmu.h for layout.
32       *  Can you use boot_pgdir as a template?
33       */
34      bcopy(boot_pgdir + PDX(UTOP), pgdir + PDX(UTOP), PDX(ULIM) - PDX(UTOP));
35
36      /*Step 4: Set e->env_pgdir and e->env_cr3 accordingly. */
37
38      /*VPT and UVPT map the env's own page table, with
39   *      *different permissions. */
40          e->env_pgdir[PDX(UVPT)]  = e->env_cr3 | PTE_V | PTE_R;
41      return 0;
42  }
```

Exercise 3.5

> Exercise 3.5 根据上面的提示与代码注释，填写 **env_ alloc** 函数。

- 从`env_free_list`中获取一个新的`Env`。`env_free_list`是一个存放空闲进程（`Env`）的链表，用于管理未被使用的进程
- 调用一个已实现的函数来为新的`Env`初始化内核内存布局。这个函数主要是将内核地址映射到新的`Env`地址上，建立起内核与新进程之间的地址映射关系，以便新进程能够访问内核提供的服务和资源
- 对新的`Env`的每个字段进行适当的初始化，为其赋予合适的值。这些字段包括进程的唯一标识符、父进程标识符、进程状态等等
- 将新的`Env`从`Env free list`中移除，表示该进程已经被使用，不再是空闲进程

```
1   int
2   env_alloc(struct Env **new, u_int parent_id)
3   {
4       int r;
5       struct Env *e;
6
7       /*Step 1: Get a new Env from env_free_list*/
8       e = *new = LIST_FIRST(&env_free_list);
9       if( e == NULL) return -E_NO_FREE_ENV;
10
11      /*Step 2: Call certain function(has been implemented) to init kernel
    memory layout for this new Env.
12       *The function mainly maps the kernel address to this new Env address.
    */
13      r = env_setup_vm(e);
14      if (r < 0) {
15          *new =NULL;
16          return r;
```

```
    }

    /*Step 3: Initialize every field of new Env with appropriate values*/
    e->env_id =mkenvid(e);
    e->env_status = ENV_RUNNABLE;
    e->env_parent_id = parent_id;

    /*Step 4: focus on initializing env_tf structure, located at this new
Env.
     * especially the sp register,CPU status. */
    e->env_tf.cp0_status = 0x10001004;
    e->env_tf.regs[29] = USTACKTOP;

    /*Step 5: Remove the new Env from Env free list*/
    LIST_REMOVE(e, env_link);
    return 0;

}
```

注释掉`init/init.c`中`mips_init()`函数中

```
//trap_init();
//kclock_init();
```

重新`make`，然后运行命令`gxemul -E testmips -C R3000 -M 64 gxemul/vmlinux`

```
---------------------------------------------------------------------------

main.c: main is start ...

init.c: mips_init() is called

Physical memory: 65536K available, base = 65536K, extended = 0K

to memory 80401000 for struct page directory.

to memory 80431000 for struct Pages.

pmap.c:  mips vm init success

pe0->env_id 2048

pe1->env_id 4097

pe2->env_id 6146

env_init() work well!

envid2env() work well!

pe1->env_pgdir 83ffe000

pe1->env_cr3 3ffe000

env_setup_vm passed!

pe2`s sp register 7f3fe000

env_check() succeeded!

panic at init.c:28: ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

发现成功运行了`env_check()`

Exercise 3.6

> Exercise 3.6 通过上面补充的知识与注释，填充 **load_icode_mapper** 函数。 ■

- 需要将`bin`文件的所有内容加载到内存中
- 当`bin`文件的大小（`bin_size`）小于指定的`sgsize`（目标大小）时，需要进一步分配内存页面以达到`sgsize`

```
1  static int load_icode_mapper(u_long va, u_int32_t sgsize,
2                               u_char *bin, u_int32_t bin_size, void
   *user_data)
3  {
4      struct Env *env = (struct Env *)user_data;
5      struct Page *p = NULL;
6      u_long i;
7      int r;
8
9      u_long vp = ROUNDDOWN(va, BY2PG);
10     u_long offset = va - vp;
```

```
11
12  #define ALLOC_AND_MAP(pp, e, addr) \
13      if((r = page_alloc(pp)) < 0 || \
14          (r = page_insert(e->env_pgdir, *pp, addr,PTE_R)) <0 \
15          )\
16          return r;
17
18      if(offset) {
19          ALLOC_AND_MAP(&p, env, vp);
20          i = BY2PG - offset;
21          bcopy(bin, page2kva(p) + offset, MIN(bin_size, i));
22      }else i=0;
23      /*Step 1: load all content of bin into memory. */
24      for (; i < bin_size; i += BY2PG) {
25          /* Hint: You should alloc a page and increase the reference count of
    it. */
26          ALLOC_AND_MAP(&p,env, va + i);
27          if( i + BY2PG >= bin_size) {
28              bcopy(bin + i, page2kva(p), bin_size - i);
29          } else {
30              bcopy(bin + i, page2kva(p), BY2PG);
31          }
32      }
33      /*Step 2: alloc pages to reach `sgsize` when `bin_size` < `sgsize`.
34      * i has the value of `bin_size` now. */
35      while (i < sgsize) {
36          ALLOC_AND_MAP(&p, env, va + i);
37          i += BY2PG;
38      }
39      return 0;
40  }
```

Exercise 3.7

> Exercise 3.7 通过补充的 ELF 知识与注释，填充 **load_elf** 函数和 **load_icode** 函数。

`lib\kernel_elfloader.c`文件中`load_elf`函数：将一个 ELF 格式的二进制文件加载到内存中，并映射到正确的虚拟地址

我需要实现将所有节(section)映射到正确的虚拟地址上。如果发生错误，返回一个小于零的值。

```
if (phdr->p_type == PT_LOAD) {

     /* Your task here!  */
        /* Real map all section at correct virtual address.Return < 0 if
error. */
        /* Hint: Call the callback function you have achieved before. */

                    r = map(phdr->p_vaddr, phdr->p_memsz, binary + phdr-
>p_offset, phdr->p_filesz, user_data);
                    if(r < 0) return r;
                }

                ptr_ph_table += ph_entry_size;
}
```

`load_icode` 函数

- 分配一个页面
- 使用适当的权限为新的 `env` 设置初始栈
- 使用 ELF 加载器加载二进制文件
- 将 CPU 的 PC 寄存器设置为适当的值

```
static void
load_icode(struct Env *e, u_char *binary, u_int size)
{
    /* Hint:
     *  You must figure out which permissions you'll need
     *  for the different mappings you create.
     *  Remember that the binary image is an a.out format image,
     *  which contains both text and data.
     */
    struct Page *p = NULL;
    u_long entry_point;
    u_long r;
    u_long perm;

    /*Step 1: alloc a page. */
    page_alloc(&p);

    /*Step 2: Use appropriate perm to set initial stack for new Env. */
    /*Hint: The user-stack should be writable? */
    page_insert(e->env_pgdir, p, USTACKTOP - BY2PG, PTE_R);

    /*Step 3:load the binary by using elf loader. */
    load_elf(binary, size, &entry_point, e, load_icode_mapper);

    /***Your Question Here***/
    /*Step 4:Set CPU's PC register as appropriate value. */
    e->env_tf.pc = entry_point;
}
```

Exercise 3.8

> Exercise 3.8 根据提示，完成 **env_create** 函数与 **env_create_priority** 的填写。
> ■

`env_create_priority` 函数:

- 使用 `env_alloc` 分配一个新 `env`
- 为新 `env` 分配优先级
- 使用 `load_icode()` 加载命名的 `elf` 二进制文件

```
void
env_create_priority(u_char *binary, int size, int priority)
{
    struct Env *e;
    /*Step 1: Use env_alloc to alloc a new env. */
    env_alloc(&e, 0);

    /*Step 2: assign priority to the new env. */
    e->env_pri = priority;

    /*Step 3: Use load_icode() to load the named elf binary. */
    load_icode(e, binary, size);
    LIST_INSERT_HEAD(env_sched_list, e, env_sched_link);
}
```

`env_create` 函数:

- 使用 `env_create_priority` 函数分配一个优先级为1的新 `env`

```
void
env_create(u_char *binary, int size)
{
     /*Step 1: Use env_create_priority to alloc a new env with priority 1 */
    env_create_priority(binary, size, 1);
}

```

Exercise 3.9

> Exercise 3.9 根据注释与理解，将上述两条进程创建命令加入 **init/init.c** 中。 ■

在 `mips_init` 函数中增加代码

```
void mips_init()
{
    printf("init.c:\tmips_init() is called\n");
    mips_detect_memory();

    mips_vm_init();
```

```
    page_init();

    env_init();
    env_check();

    /*you can create some processes(env) here. in terms of binary code, please refer current directory/code_a.c
     * code_b.c*/
    /*you may want to create process by MACRO, please read env.h file, in which you will find it. this MACRO is very
     * interesting, have fun please*/
    ENV_CREATE_PRIORITY(user_A, 2);
    ENV_CREATE_PRIORITY(user_B, 1);

    trap_init();
    kclock_init();
    panic("^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^");
    while(1);
    panic("init.c:\tend of mips_init() reached!");
}
```

Exercise 3.10

> Exercise 3.10 根据补充说明，填充完成 **env_run** 函数。 ■

- 保存当前环境的寄存器状态
- 将 `curenv` 设置为新 `env`
- 使用 `lcontext()` 切换到新 `env` 的地址空间
- 使用 `env_pop_tf()` 恢复环境的寄存器，并在环境中切换到用户模式

```
void
env_run(struct Env *e)
{
    /*Step 1: save register state of curenv. */
    /* Hint: if there is a environment running,you should do
    *  context switch.You can imitate env_destroy() 's behaviors.*/
    if(curenv) {
        curenv->env_tf = *((struct Trapframe *) TIMESTACK - 1);
        curenv->env_tf.pc = curenv->env_tf.cp0_epc;
    }

    /*Step 2: Set 'curenv' to the new environment. */
    curenv = e;

    /*Step 3: Use lcontext() to switch to its address space. */
    lcontext(e->env_pgdir);

    /*Step 4: Use env_pop_tf() to restore the environment's
     * environment   registers and drop into user mode in the
     * the   environment.
     */
    /* Hint: You should use GET_ENV_ASID there.Think why? */
```

```
    env_pop_tf(&e->env_tf, GET_ENV_ASID(e->env_id));
}
```

Exercise 3.11

> Exercise 3.11 将异常分发代码填入 **boot/start.S** 合适的部分。

```
.section .text.exc_vec3
NESTED(except_vec3, 0, sp)
        .set    noat
        .set    noreorder
        /*
         * Register saving is delayed as long as we don't know
         * which registers really need to be saved.
         */
1:  //j 1b
    nop

        mfc0    k1,CP0_CAUSE
        la  k0,exception_handlers
        /*
         * Next lines assumes that the used CPU type has max.
         * 32 different types of exceptions. We might use this
         * to implement software exceptions in the future.
         */

        andi    k1,0x7c
        addu    k0,k1
        lw  k0,(k0)
        NOP
        jr  k0
        nop
        END(except_vec3)
        .set    at
```

Exercise 3.12

> Exercise 3.12 将 lds 代码补全使得异常后可以跳到异常分发代码。

把这段代码放到 `tool/scse0_3.lds` 文件

Exercise 3.13

> Exercise 3.13 通过上面的描述，补充 **kclock_init** 函数。

理解代码：

```
void
kclock_init(void)
{
    /* initialize 8253 clock to interrupt 100 times/sec */
    //outb(TIMER_MODE, TIMER_SEL0|TIMER_RATEGEN|TIMER_16BIT);
    //outb(IO_TIMER1, TIMER_DIV(100) % 256);
    //outb(IO_TIMER1, TIMER_DIV(100) / 256);
    //printf("  Setup timer interrupts via 8259A\n");
    set_timer();
    //irq_setmask_8259A (irq_mask_8259A & ~(1<<0));
    //printf("  unmasked timer interrupt\n");

}
```

可以看到，这个函数只是调用了`set_timer`函数

`set_timer`函数，其定义在`lib/kclock_asm.S`文件里：

```
LEAF(set_timer)   //定义set_timer函数

    li t0, 0x01   //t0设为1
    sb t0, 0xb5000100  //设置时钟频率为1秒钟1次
    sw  sp, KERNEL_SP  //保存当前栈指针
setup_c0_status STATUS_CU0|0x1001 0   //把CP0_STATUS第12位和第0位设置1，允许4号中断，并表示开启了中断，禁止再次响应中断
    jr ra  //函数返回

    nop
END(set_timer)  //结束
```

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

图 3.2: R3000 的 SR 寄存器示意图

Exercise 3.14

> Exercise 3.14 根据注释，完成 **sched_yield** 函数的补充，并根据调度方法对 env.c 中的部分函数进行修改，使得进程能够被正确调度。 ■

带优先级的双队列调度

```
void sched_yield(void)
{
    static int count = 0;
    static int point = 0;

```

```
    struct Env *e = curenv;
    if(count == 0 || e == NULL || e->env_status != ENV_RUNNABLE) {
        if(e) {
            LIST_REMOVE(e, env_sched_link);
            LIST_INSERT_TAIL(&env_sched_list[1-point], e, env_sched_link);
        }
        while (1) {
            while (LIST_EMPTY(&env_sched_list[point]))
                point = 1-point;
            e = LIST_FIRST(&env_sched_list[point]);
            if(e->env_status != ENV_RUNNABLE) {
                LIST_REMOVE(e, env_sched_link);
                LIST_INSERT_TAIL(&env_sched_list[1-point], e,
env_sched_link);
            } else {
                count = e->env_pri;
                break;
            }
        }
    }
    count--;
    env_run(e);
}

```

重新`make`，然后运行命令`gxemul -E testmips -C R3000 -M 64 gxemul/vmlinux`

```
--------------------------------------------------------------------------------

main.c: main is start ...

init.c: mips_init() is called

Physical memory: 65536K available, base = 65536K, extended = 0K

to memory 80401000 for struct page directory.

to memory 80431000 for struct Pages.

pmap.c:  mips vm init success

pe0->env_id 2048

pe1->env_id 4097

pe2->env_id 6146

env_init() work well!

envid2env() work well!

pe1->env_pgdir 83ffe000

pe1->env_cr3 3ffe000

env_setup_vm passed!

pe2`s sp register 7f3fe000

env_check() succeeded!

panic at init.c:28: ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 21 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

思考

Thinking 3.1

> **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)

完成空闲链表的插入后，`envs`数组下标正好对应链表中的由前到后的顺序，调用空闲进程数组时优先调用下标最小的

Thinking 3.2

> 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？
>
> ■

- 高位使用静态变量以生成不同的 id，低位通过位置偏移计算，便于使用进程 id 查询到进程控制块
- 如果没有进行`e->env_id != envid`的判断，即使`envid`在`envs`数组的范围内，也会将`envs`数组中的任意一个值赋值给e，而不关心`envid`是否匹配，查询到不正确或者不存在的进程

Thinking 3.3

> 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 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
>
> ■

- 因为每个进程都需要拥有内核的页表信息，这样才能够在陷入内核时正确执行，因此需要把内核拥有的虚拟地址对应的页表进行拷贝
- `UTOP = 0x7f400000`，是为用户所能操纵的地址空间的最大值；`ULIM = 0x80000000`，是操作系统分配给用户地址空间的最大值。两者之间的区域对用户进程而言是一个只读片段，保存着页表、`struct Page`、`struct Env`，显然这些都是用户程序不允许动的。`UTOP`以下的区域就可以被用户程序所读写
- 由自映射机制可得，`pgdir[PDX(UVPT)]`就代表页表起始地址所对应的页目录项。`env_cr3`是该进程的页目录物理地址，这样即可通过页表项的虚拟地址准确找到物理地址
- 物理地址是实际的内存存储单元的地址，用于处理器直接读取和写入数据。虚拟地址是程序生成的逻辑地址，表示进程认为自己能访问到的内存范围。每个进程使用自己独立的虚拟地址，而不同进程的虚拟地址可以映射到相同的物理地址，实现内存共享

Thinking 3.4

> 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)

不能，因为`user_data`是向内层函数传递的参数，即向最后一个参数——函数指针对应的函数传递。如果没有`user_data`，那么传递的函数指针对应的函数无法获得所需要的外部参数，与C语言中标准库`stdlib.h`中对应的快排`qsort`类似。

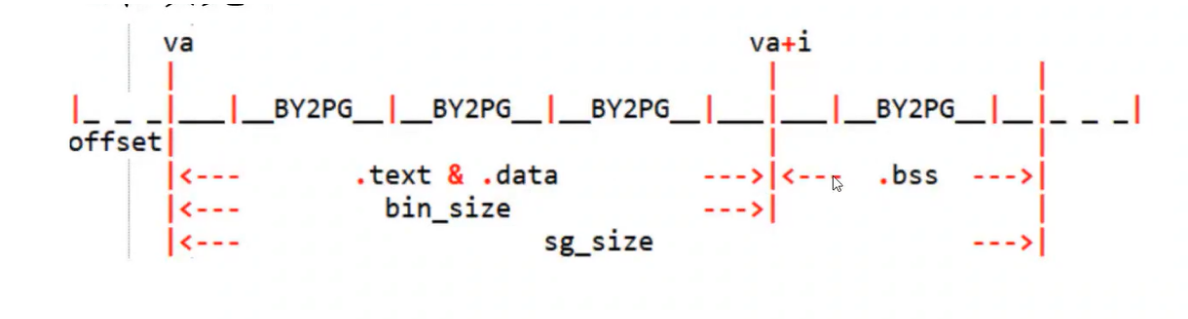
```
1 void qsort(void*base, size_t num, size_t width, int(__cdecl *compare)(const
  void*,const void*));
```

其中width参数是为了给`qsort`函数内部调用的`compare`函数提供信息。

Thinking 3.5

> 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？

最坏的情况：



即`va`以及`va+bin_size`均不是BY2PG对齐的，分为四种复制。第一种，首尾均不对齐，第二种，首对齐尾不对齐，第三种，首不对齐尾对齐，第四种，首尾均对齐。

都考虑到了

Thinking 3.6

> 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？

“指令的位置”是虚拟空间，即“第几条”指令，而“第几条”是一种相对偏移，即一块“连续”区域，只有虚拟空间是连续的

`entry_point` 是虚拟地址，对每个进程都一样。`*entry_point = ehdr->e_entry;`，尽管不同进程其实虚拟地址一样，但加载的二进制文件、页表肯定是不一样的。每个进程起始地址统一会降低操作系统的复杂度，但在部分相同虚拟地址中的内容不同也区分了不同的进程

Thinking 3.7

> 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？ ■

应该设置为 `env_tf.cp0_epc` 的值，`cp0_epc` 中保存的是进程发生中断的指令地址，当切换上下文，恢复该进程时，应当从发生中断的指令继续向下执行，因此将 `pc` 值设为中断的指令地址。

Thinking 3.8

> 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` 的定义：发生时钟中断的栈顶地址
- 由 `mmu.h` 中我们可以看到，`TIMESTACK` 的值刚好是0x82000000，`SAVE_ALL` 函数为保存当前进程的相关寄存器值，其中调用了 `get_sp` 函数来获取栈顶的值

```
99
00  #define UTOP UENVS
01  #define UXSTACKTOP (UTOP)
02  #define TIMESTACK 0x82000000
03
```

- `KERNEL_SP` 为发生其他中断的栈顶地址

Thinking 3.9

> 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)   //定义set_timer函数

    li t0, 0x01   //t0设为1
    sb t0, 0xb5000100  //设置时钟频率为1秒钟1次
    sw  sp, KERNEL_SP  //保存当前栈指针
setup_c0_status STATUS_CU0|0x1001 0   //把CP0_STATUS第12位和第0位设置1，允许4号中断，并表示开启了中断，禁止再次响应中断
    jr ra  //函数返回

    nop
END(set_timer)  //结束
```

Thinking 3.10

> **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。 ■

RUNNABLE状态下的进程有两个链表队列，初次装进程装在两个队列中，一次运行一个进程。定时器周期性产生中断，使得当前进程被迫停止，通过执行`sched_yield`函数，来进行进程的调度，若该进程时间片还未用完，则可用时间片数量 - 1，否则会切换到下一个进程，保存上下文。并将原来的进程送到另一个队列的末尾，若进程不处于RUNNABLE状态，则会进行其他处理。

提交 lab3

```
jovyan@74846d05e007:~/ouc21020007131-1ab$ git add .
jovyan@74846d05e007:~/ouc21020007131-1ab$ git config --global user.email "zym800
4@stu.ouc.edu.cn"
jovyan@74846d05e007:~/ouc21020007131-1ab$ git config --global user.name "munume"
                jovyan@74846d05e007:~/ouc21020007131-1ab$ git commit -m "lab3"
[lab3 079430a] lab3
 39 files changed, 1631 insertions(+), 143 deletions(-)
 rewrite drivers/gxconsole/console.o (100%)
 create mode 100644 include/.ipynb_checkpoints/pmap-checkpoint.h
 create mode 100644 include/.ipynb_checkpoints/trap-checkpoint.h
 create mode 100644 init/code_a.o
 create mode 100644 init/code_b.o
 create mode 100644 lib/.ipynb_checkpoints/env-checkpoint.c
 create mode 100644 lib/.ipynb_checkpoints/env_asm-checkpoint.S
 create mode 100644 lib/.ipynb_checkpoints/genex-checkpoint.S
```

```
jovyan@74846d05e007:~/ouc21020007131-1ab$ git push
Counting objects: 39, done.
Delta compression using up to 8 threads.
Compressing objects:   2% (1/38)Compressing objects:   5% (2/38)Compressing obje
cts:   7% (3/38)Compressing objects:  10% (4/38)Compressing objects:  13% (5/38)
Compressing objects:  15% (6/38)Compressing objects:  18% (7/38)Compressing obje
cts:  21% (8/38)Compressing objects:  23% (9/38)Compressing objects:  26% (10/38
Compressing objects:  28% (11/38Compressing objects:  31% (12/38Compressing obje
cts:  34% (13/38Compressing objects:  36% (14/38Compressing objects:  39% (15/38
Compressing objects:  42% (16/38Compressing objects:  44% (17/38Compressing obje
cts:  47% (18/38Compressing objects:  50% (19/38Compressing objects:  52% (20/38
Compressing objects:  55% (21/38Compressing objects:  57% (22/38Compressing obje
cts:  60% (23/38Compressing objects:  63% (24/38Compressing objects:  65% (25/38
Compressing objects:  68% (26/38Compressing objects:  71% (27/38Compressing obje
cts:  73% (28/38Compressing objects:  76% (29/38Compressing objects:  78% (30/38
Compressing objects:  81% (31/38Compressing objects:  84% (32/38Compressing obje
cts:  86% (33/38Compressing objects:  89% (34/38Compressing objects:  92% (35/38
Compressing objects:  94% (36/38Compressing objects:  97% (37/38Compressing obje
cts: 100% (38/38Compressing objects: 100% (38/38), done.
Writing objects: 100% (39/39), 39.81 KiB | 2.49 MiB/s, done.
Total 39 (delta 18), reused 1 (delta 0)
remote: ************************************************************************
remote:
remote:                       BUAA OSLAB AUTOTEST SYSTEM
remote:                     Copyright (c) BUAA 2015-2019
remote:
remote: ************************************************************************
remote:
remote: [ You are changing the branch: refs/heads/lab3. ]
remote:
remote: Autotest: Begin at Mon Jan  8 04:45:24 CST 2024
remote:
remote: warning: remote HEAD refers to nonexistent ref, unable to checkout.
remote:
remote: Switched to a new branch 'lab3'
remote: Branch lab3 set up to track remote branch lab3 from origin.
```

```
remote: /OSLAB/compiler/usr/bin/mips_4KC-gcc -O -G 0 -mno-abicalls -fno-builtin
-Wa,-xgot -Wall -fPIC -I./ -I../ -I../include/ -c tlb_asm.S
remote: make[1]: Leaving directory `/usr/src/workdir/mm'
remote: /OSLAB/compiler/usr/bin/mips_4KC-ld -o gxemul/vmlinux -N -T tools/scse0_
3.lds boot/start.o init/*.o drivers/gxconsole/console.o lib/*.o mm/*.o
remote:
remote: End build at Mon Jan  8 04:45:39 CST 2024
remote: [ You got 100 (of 100) this time. Mon Jan  8 04:45:49 CST 2024 ]
remote:
remote:
remote: >>>>>> Collecting autotest results >>>>>>
```

三、实验总结

收获与体会：

对进程的创建，使用过程更加了解，这里很多需要阅读汇编代码的地方，需要好好体会，而且解决问题不能仅仅看一个文件，需要对不同文件中的多个函数或定义理解清楚，比较费时费力，也很锻炼能力

总时长：大概六七个小时