林子杰_20373980_lab3

一、思考题

Thinking 3.1

此处应当是加强了进程块拿取的判断条件。由于进程控制块数组中的进程控制块可能被替换分配，故而要确保所给`envid`拿取到id对应的正确的进程控制块。

Thinking 3.2

UTOP是用户进程可读写的最高地址；ULIM是kuseg的最高地址。UTOP到ULIM间的区域用于存放用户进程控制块和页表，不能被用户随意读写。UTOP以下的空间可以被用户读写。

`env_cr3`存储的是页目录的物理地址，`UVPT`的位置是自映射条件下，应存放页目录基地址的地址，此处将这一位置（即`pgdir[PDX(UVPT)]`）赋为`env_cr3`，建立了自映射。

在lab2中，我们已经建立好了二级页表映射机制，完善了物理内存和虚拟内存管理机制。通过这一机制，我们可以在进程中使用虚拟地址，进程在使用虚拟地址时，根本上操纵的也是物理地址。

Thinking 3.3

`user_data`这一参数是`load_icode()`函数中的`Env *e`。这个参数必须存在，否则`load_elf()`函数无法向其函数参数`load_icode_mapper()`传递该参数。C语言的库函数中，快速排序函数qsort有着类似的行为：

```
void qsort(void *base, size_t n, size_t size, *int (*cmp)(const void *, const void *));
```

`size`给`cmp()`函数传递了信息。至于具体是什么信息，我想这可能和C++中的STL有一些类似的行为，比如告知`cmp()`比较目标的大小；换句话说，也就是起到了自定义模版的作用。

Thinking 3.4

复制情况的不同主要是由`va`、`va+bin_size`和`va+sg_size`是否能对齐决定的。下面将情况列表展示。

va	va+bin_size	va+sg_size	备注
对齐	对齐	对齐	最理想的情况
对齐	对齐	未对齐	va+sg_size所在页需要特殊处理
对齐	未对齐	对齐	va+bin_size所在页需要特殊处理
对齐	未对齐	未对齐	va+sg_size和va+bin_size所在页都需要特殊处理
未对齐	对齐	对齐	va所在页需要特殊处理
未对齐	对齐	未对齐	va和va+sg_size所在页都需要特殊处理
未对齐	未对齐	对齐	va和va+bin_size所在页都需要特殊处理
未对齐	未对齐	未对齐	这三者所在的页都需要特殊处理

最直接且暴力的方法就是处理最复杂的情况。我们使用`MIN()`函数，对相关的值进行比较来处理这些情况。具体方法已经写在笔记中了。注意，可能存在多种需要处理的情景在同一页面产生。

Thinking 3.5

`env_tf.pc`存储的是虚拟地址。CPU依靠`env_tf.pc`来寻找中断位置，那么这个域存储的自然是虚拟地址。

`entry_point`在每个进程中都是一样的，这种统一是因为每个进程创建的方式都相同。新进程的创建，都是按照我们写好的进程创建函数逐步执行。不过需要注意，`entry_point`只是一个虚拟地址，并非物理地址，故而每个进程中`entry_point`可能映射到不同的物理地址。

Thinking 3.6

`epc`是进程发生中断时的指令的地址，这个值存储在`env_tf.cp0_epc`中。在恢复该进程时，应当从发生中断的指令继续向下执行，而不应该从头开始执行。因此将`pc`值设为中断的指令地址，那么下一次进入这个进程时，程序便会从此开始执行。

Thinking 3.7

在执行`sched_yield()`函数前，`KERNEL_SP`的一个`Trapframe`大小的内容被存到了`TIMESTACK`区域。这应该是一个用于保存现场的栈区，至于`sched_yield()`函数有什么作用，为什么要这样存储，刚做完第一部分的我可能还无法回答。

`TIMESTAKC`，顾名思义，显然发生时钟中断时的栈指针。`KERNEL_SP`是发生其他中断时的栈指针。

Thinking 3.8

- `handle_int`定义在`lib/genex.S`中，直接以mips中函数的形式给出。
- `handle_mod`、`handle_tlb`、`handle_reserved`这三个函数也是在`lib/genex.S`中实现的，不过它们的实现方法比较特殊。 咋一看，好像定义了很多函数，但和这三个函数没有任何的联系，不过我们来到文件末尾，会找到这样一段代码：

```
BUILD_HANDLER reserved do_reserved cli
BUILD_HANDLER tlb   do_refill   cli
BUILD_HANDLER mod   page_fault_handler cli
```

出现了熟悉的单词，但还仅仅是局部相似，不过确实可以找到`reserve`对应的`do_reserve`函数，`tlb`对应的`do_refill`函数，证明我们的寻找方向是正确的。我们可以猜测，这些函数的定义可能和`BUILD_HANDLER`有关。于是我们试着追溯一下它的源头，发现它正是定义在文件开头的一个宏：

```
.macro  BUILD_HANDLER exception handler clear
    .align  5
    NESTED(handle_\exception, TF_SIZE, sp)
    .set    noat
```

虽然不能完全明白这里的MIPS语法，但从`handle_\exception`可以看出，这里略有拼接的味道了，再结合之前的分析，我们可以肯定这三个函数也是定义在`lib/genex.S`下的。

- 前面几个函数都是定义在`.S`文件中的汇编函数，那么`handle_sys`函数也应当是如此。在`lib`目录下，可以找到一个`syscall.S`文件，`handle_sys`函数正是在其中实现。

Thinking 3.9

首先来解释`set_timer`函数。

```
.macro  setup_c0_status set clr
    //.set push是一条为指令，貌似是指示汇编器保存当前的汇编状态。在Mars中，其好像已经表为无用
    .set    push
    //将SR寄存器的内容存入t0寄存器
    mfc0    t0, CP0_STATUS
    //将t0寄存器的内容与STATUS_CU0|0x1001取或并写回，其内容应当是0x10001001
    or  t0, \set|\clr
    //将t0寄存器的内容与0取或并写回。此时t0寄存器的28位、12位、0位都置1
    xor t0, \clr
    //写回SR寄存器，写回值的意义参考SR寄存器的描述。
    mtc0    t0, CP0_STATUS
    //恢复原有的汇编状态
    .set    pop
.endm

LEAF(set_timer)

    //向t0寄存器中写入0xc8
    li t0, 0xc8
    //向0xb5000100中写入0xc8。0xb5000000是gxemul映射实时钟的位置，后者代表一秒钟触发中断200次
    sb t0, 0xb5000100
    //将栈指针寄存器的内容存入内存中内核栈的位置
    sw  sp, KERNEL_SP
    //调用宏setup_c0_status，传入set参数为STATUS_CU0|0x1001，clr参数为0
    setup_c0_status STATUS_CU0|0x1001 0
```

```
    //跳转到ra寄存器的内容，即回到调用其的函数处继续执行程序
    jr ra
    nop
END(set_timer)
```

接下来看一看`timer_irq`函数。

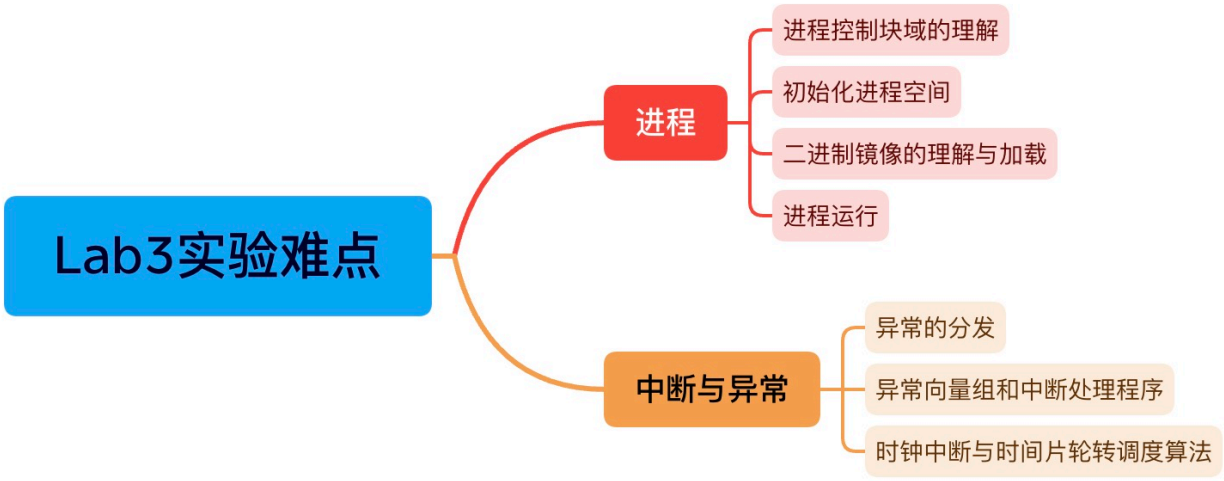
```
timer_irq:
    //向0xb5001100写入0x0。该位置应当与时钟相关
    sb zero, 0xb5000110
    //跳转到sched_yield函数执行
1:  j   sched_yield
    nop
    /*li t1, 0xff
    lw    t0, delay
    addu  t0, 1
    sw  t0, delay
    beq t0,t1,1f
    nop*/
    //跳转到ret_from_exceptio处执行
    j   ret_from_exception
    nop
```

Thinking 3.10

进程执行时，时钟会按照已有频率触发中断且提供属于自己的中断码。OS能够根据中断码跳转到特殊的中断处理程序——时钟中断处理程序对应的内容就是进程切换。我们在之前已经完成了保护现场、程序跳转等一系列工序，在此处只需按照轮转调度算法执行即可。

时钟每中断一次，都会进入一次`sched_yield()`，时间片`count`减一；当时间片用完，就会队列中选取满足条件的进程进行执行。

二、实验难点



lab3的内容主要分为**进程**和**中断与异常**两个部分。进程部分的难点之一在于二进制镜像加载部分的理解和进程控制块中有关与内存映射的问题；另外一个难点则在于其函数关系的复杂性，阅读时需要来回切换、不断梳理，才能建立完善的知识体系。

中断与异常部分相对简单，不过还是要对这一部分有高屋建瓴的视角，了解异常分发的原因、处理方法，以及进程切换和时钟中断之间的关系。另外，设计时间片轮转调度算法也有一定的难度。

三、心得体会

lab3中，进程的初始化部分和二进制镜像的部分和lab2中的内存管理有比较大的联系，对综合掌握知识能力的要求比较高；另外，合理使用ctags能够极大提升实验完成的速度。总体来讲，本次lab的课下部分完成相对平稳，但是感觉指导书出现了阅读吃力、讲解混乱矛盾的现象，希望能够改进；同时，我在lab3-1的上机中也栽倒在了指导书的歧义上，已经做足了充分的准备却因此失分，深感不甘与愤怒，希望课程组能从学生视角，加大审查指导书表述的力度。

仍和之前的报告一样，在下面附上自己的实验笔记。

Lab3实验笔记

O、前记

从lab0到lab2，感觉之前做笔记的方法还是有代提高。先在，对基础的知识有了初步的认识后，我希望我的笔记能够更多的关注于代码。故而本次实验笔记中的记录方法可能和以前有所不同，而这都是建立在对知识有一定的理解之后的。另外，我认为，只从线性的层面梳理实验中要填写的代码，其实并不自然。在本次笔记中，我会尝试在线性结构和树状结构两方面来理解代码。

一、进程

1.进程控制块

首先介绍一个和内存控制块非常相似的东西：进程控制块。他被定义在`incldue/env.h`中。下面介绍其中一些比较重要的代码（并未列出全部）。

```
struct Env {
    /* env_tf，是一个Trapframe结构体，里面保存了进程的上下问环境，用于进程调度或是
    陷入中断。这个结构体中有cp0_status、cp0_badvaddr、cp0_epc、pc等寄存器的信息 */
    struct Trapframe env_tf;
    /* 和pp_link类似，使用它和env_free_list构建空闲进程链表，用于链接 */
    LIST_ENTRY(Env) env_link;       // Free LIST_ENTRY
    /* 进程id，独一无二。是不是想到了之前见过的进程标识符？ */
    u_int env_id;                   // Unique environment identifier
    /* 父进程id，一个进程可以被一个进程创建，故而存在父进程这一说法 */
    u_int env_parent_id;            // env_id of this env's parent
    /* 进程状态，一共有三种：ENV_FREE，进程空闲，这个块处于空闲链表；ENV_NOT_RUNNABLE，
    进程阻塞，当前不可执行，等待信息唤醒；ENV_RUNNABLE，进程处于执行或就绪状态*/
    u_int env_status;               // Status of the environment
    /*  这个变量保存了该进程页目录的内核虚拟地址 */
    Pde *env_pgdir;                 // Kernel virtual address of page dir
    /* 这个变量保存了该进程页目录的物理地址 */
    u_int env_cr3;
    /* 这个变量用来构造调度队列 */
    LIST_ENTRY(Env) env_sched_link;
    /* 这个变量保存了该进程的优先级 */
    u_int env_pri;
};
```

在MOS系统中，放置内存进程块的物理内存在启动后就要安排好，即`envs`数组（和`pages`数组很相似），这一部分工作已经在`mips_vm_init()`实现了。代码如下：

```
/* 开辟了NENV个Env结构体大小的空间，起始地址为envs */
envs = (struct Env *)alloc(NENV * sizeof(struct Env), BY2PG, 1);
/* 将NENV个Env大小的空间和页的大小进行对齐，即求出envs数组占用了多少页 */
n = ROUND(NENV * sizeof(struct Env), BY2PG);
/* 将[UENVS,UENVS+n)的虚拟地址映射到[PADDR(envs),PADDR(envs)+n)的物理地址上 */
boot_map_segment(pgdir, UENVS, n, PADDR(envs), PTE_R);
```

阅读了这一部分代码后，我们能够体会到进程管理和内存管理的相似性，并且对进程相关的量有了初步的了解。下面我们按照提示来完成`exercise 3.2`。

```
/* 此函数的作用是，将envs中所有的进程控制块标记为空闲状态，并且将它们插入env_free_list用于管理。注
意，要将它们反过来插入链表，这样第一次使用函数env_alloc()分配进程时，调用的是envs[0] */
/*** exercise 3.2 ***/
env_init(void)
{
    int i;
    /* Step 1: Initialize env_free_list. */
```

```
    /* 初始化env_free_list，可以使用链表宏 */
    LIST_INIT(&env_free_list);
    /* Step 2: Traverse the elements of 'envs' array,set their status as free and
insert them into the env_free_list.Choose the correct loop order to finish the
insertion.Make sure, after the insertion, the order of envs in the list should be the
same as that in the envs array. */
    /* 遍历envs数组，将其中每个进程控制块的状态都设为空闲，并且插入env_free_list链表。注意，由于此处
使用的是LIST_INSERT_HEAD()宏，在循环的时候要选择正确的顺序，以保证插入后链表的顺序和数组的顺序相同 */
    for(i=NENV-1;i>=0;i--)
    {
        envs[i].env_status = ENV_FREE;
        LIST_INSERT_HEAD(&env_free_list,&envs[i],env_link);
    }
}
```

2.进程的标识

OS在同一时间段内可能会运行很多不同的进程，操作系统需要依靠**标识**来识别这些不同的进程。介绍进程控制块的时候，我们知道有一个`env_id`域可以用来标识进程。为了避免标识符产生冲突，`env.c`中实现了一个`mkenvid()`函数，用于给一个新创建的进程分配和当前未释放的进程的标识符都不同的标识符。

还记得为什么需要标识符吗？当进程切换时，TLB根据虚拟地址判断重填情况。但是**不同进程**的**相同的虚拟地址**可能映射到**不同的物理页**，故而需要强化TLB充填的条件，通过增加标识符来解决这一问题。当然，我们的TLB只给ASID分配了6位，即最多可识别64个不同的进程，故而分配标识符不能用太过简单的方法，否则会导致进程溢出（这也是我们需要`mkenvid()`函数的原因）。当没有可分配标识符且需要创建新进程时，系统会panic，详见`asid_alloc()`函数。

`exercise 3.3`要求我们填写`lib/env.c`中的`envid2env()`函数，令其可通过`env`的id来获取对应的进程控制块。

```
/* 此函数的功能为，输入一个envid，可以将**penv指向该envid对应的进程控制块的指针。如果envid为0，那么
令**penv指向为当前正在执行进程的结构体指针；否则，令它指向envs[ENVX(envid)]这一进程控制块的结构体的
指针。另外，还有一个特殊的权限条件checkperm，需要函数根据实际情况做一些其他的判断。如果拿取控制块成功，
则返回0；否则返回-E_BAD_ENV */
/*** exercise 3.3 ***/
int envid2env(u_int envid, struct Env **penv, int checkperm)
{
    struct Env *e;
    /* 如果envid为0，则拿取当前正在执行的进程的控制块的结构体指针，否则通过宏来获得目标控制块的结构体
指针 */
  if(envid == 0) {
    *penv = curenv;
    return 0;
  } else {
    e = &envs[ENVX(envid)];
  }
    /* 如果当前进程控制块空闲（即这个控制块没有用来控制进程），或是拿取的进程控制块的id和当前所给的
envid不同时，令**penv的内容为0，并且返回错误代码-E_BAD_ENV */
    if (e->env_status == ENV_FREE || e->env_id != envid) {
```

```
        *penv = 0;
        return -E_BAD_ENV;
    }
    /* 检查checkperm是否需要判断，倘若其被置1，那么envid的进程必须是当前进程或是当前进程的直接子进程，否则返回错误码 */
  if(checkperm == 1) {
    if(e != curenv && e->env_parent_id != curenv->env_id) {
      *penv = 0;
      return -E_BAD_ENV;
    }
  }

    *penv = e;
    return 0;
}
```

Thinking 3.1

此处应当是加强了进程块拿取的判断条件。由于进程控制块数组中的进程控制块可能被替换分配，故而要确保所给`envid`拿取到id对应的正确的进程控制块。

3.设置进程控制块

配合空闲进程列表`env_free_list`，我们可以创建新的进程。创建进程的流程大致如下：

1. 申请一个空闲的进程控制块（即Env结构体），这需要从`env_free_list`中拿取。
2. 手动设置进程控制块的参数（现代操作系统中的进程一般都有模版支持，但MOS中没有）。
3. 为进程分配资源。这其中包括了程序、数据、用户栈的空间。
4. 从`env_free_list`中将进程块移除，进程创建完毕。

在创建过程中，我们需要用到`env_alloc()`和`env_setup_vm()`这两个函数。其中，第一个函数要调用后面这一函数，故而我们先分析第二个函数。

`env_setup_vm()`函数的作用是初始化新进程的地址空间，我们来接和注释进行分析。

```
/* 初始化进程控制块指针e所指向的进程控制块的内核虚拟地址。首先，分配一个页面目录pgdir，并设置e中的env_pgdir和env_cr3，初始化env地址空间的内核部分。不要将任何东西映射到env虚拟地址空间的用户部分。 */
/*** exercise 3.4 ***/
static int
env_setup_vm(struct Env *e)
{
    int i, r;
    struct Page *p = NULL;
    Pde *pgdir;
    /* 为页目录声明一个页面p，使用page_alloc()函数声明。倘若声明失败，则返回错误码。然后，将新声明的内存控制块p标记为使用过（使用pp_ref++）。pgdir是e的页目录 */
    if ((r = page_alloc(&p)) < 0) {
        panic("env_setup_vm - page alloc error\n");
        return r;
    }
```

```
    /* 标记p的页面已经被使用 */
    p->pp_ref++;
    /* 将pgdir设置为p所对应的内核虚拟地址 */
    pgdir = (Pde *)page2kva(p);
    /* 将pgdir在UTOP以下的区域清零 */
    for(i = 0;i < PDX(UTOP);i++) {
        pgdir[i] = 0;
    }
    /* 将内核目录boot_pgdir的内容拷贝到pgdir */
    for(i = PDX(UTOP);i <= PDX(~0);i++) {
        pgdir[i] = boot_pgdir[i];
    }
    /* 根据函数前的提示设置env_pgdir和env_cr3 */
    e->env_pgdir = pgdir;
    e->env_cr3 = PADDR(pgdir);
    /* UVPT maps the env's own page table, with read-only permission.*/
    e->env_pgdir[PDX(UVPT)]  = e->env_cr3 | PTE_V;
    return 0;
}
```

Thinking 3.2

UTOP是用户进程可读写的最高地址；ULIM是kuseg的最高地址。UTOP到ULIM间的区域用于存放用户进程控制块和页表，不能被用户随意读写。UTOP以下的空间可以被用户读写。

`env_cr3`存储的是页目录的物理地址，`UVPT`的位置是自映射条件下，应存放页目录基地址的地址，此处将这一位置（即`pgdir[PDX(UVPT)]`）赋为`env_cr3`，建立了自映射。

在lab2中，我们已经建立好了二级页表映射机制，完善了物理内存和虚拟内存管理机制。通过这一机制，我们可以在进程中使用虚拟地址，进程在使用虚拟地址时，根本上操纵的也是物理地址。

刚刚，我们完善好了`env_setup_vm()`这一函数。前面我们已经提到过，这一函数是用在`env_alloc()`函数中的，接下来，我们便开始讲解这一个函数。

```
/* 此函数的功能是创建并初始化一个新的进程。如果初始化成功，则将该进程的Env结构体指针填入*new并返回0，否则返回错误码。如果这个进程没有父进程，则其parent_id域应0，注意env_init()函数在此之前已经被调用过了。能够创建新进程的时候，要将Env结构体中的一些域设置为合适的值，它们是id、status、sp寄存器、CPU状态、parent_id */
/*** exercise 3.5 ***/
int env_alloc(struct Env **new, u_int parent_id)
{
    int r;
    struct Env *e;
    /* 首先，从env_free_list中得到一个Env结构体。此处首先判断env_free_list是否为空，为空则不能创建进程，返回错误码（此处可使用宏LIST_EMPTY）。如果不为空，则使用LIST_FIRST宏取出一个Env */
    if(LIST_EMPTY(&env_freelist)) {
        *new = NULL;
        return -E_NO_FREE_ENV;
    }
    e = LIST_FIRST(&env_free_list);
```

```
    /* 然后，调用一个特定的函数初始化这个Env结构体的内核布局。这个函数主要将内核地址映射到此Env的地址
*/
    env_setup_vm(e);
    /* 再之后，初始化Env结构体的某些域，赋予它们合适的值 */
    e->env_id = mkenvid(e);           //进程env_id，使用mkenvid()赋值
    e->env_status = ENV_RUNNABLE;    //进程状态env_status，赋值为ENV_RUNNABLE
    e->env_parent_id = parent_id;
    /* 初始化有关sp寄存器的域个env_tf的cp0_status域；同时，设置栈指针，即29号寄存器的值为
USTACKTOP，它是0x7f3fe000 */
    e->env_tf.cp0_status = 0x10001004;
    e->env_tf.regs[29]=USTACKTOP;
    /* 从空闲链表中摘除该进程控制块 */
    LIST_REMOVE(e,env_link);
    *new = e;
    return 0;
}
```

讲解一句比较重要的代码:`e->env_tf.cp0_status = 0x10001004`。它和SR寄存器有关。

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

第28位置1，表示允许在用户模式下使用CP0寄存器；第12位置为1，表示4号中断可以被响应（计组的知识）。"另外，SR 寄存器的低六位是一个二重栈的结构。KUo和IEo是一组，每当中断发生的时候，硬件自动会将 KUp 和 IEp 的数值拷贝到这里；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。其中KU表示是否位于内核模式下，为1表示位于内核模式下；IE表示中断是否开启，为1表示开启，否则不开启。"

"而每当rfe指令调用的时候，就会进行上面操作的**逆操作**。我们现在先不管为何,但是已经知道, 下面这一段代码 (位于`lib/env_asm.S` 中) 是**每个进程在每一次被调度时都会执行**的，所以就一定会执行rfe这条指令。"

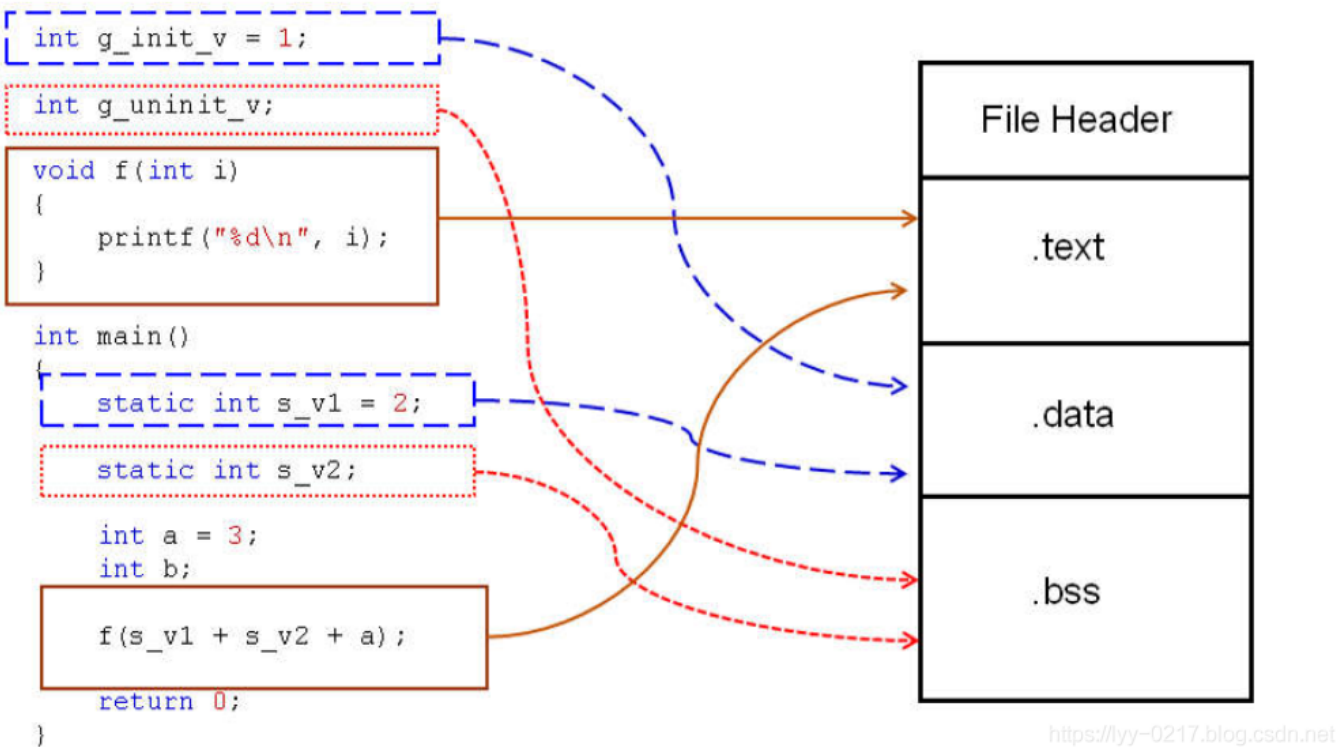
```
lw k0,TF_STATUS(k0) # 恢复CP0_STATUS 寄存器
mtc0 k0,CP0_STATUS
j k1
rfe
```

先在，我们大致明白为什么要将低六位设置为000100了。创建进程后，上面这段代码被调度，低六位就变成了000010，允许中断。这就是为什么要将`e->env_tf.cp0_status`设置为0x10001004了。

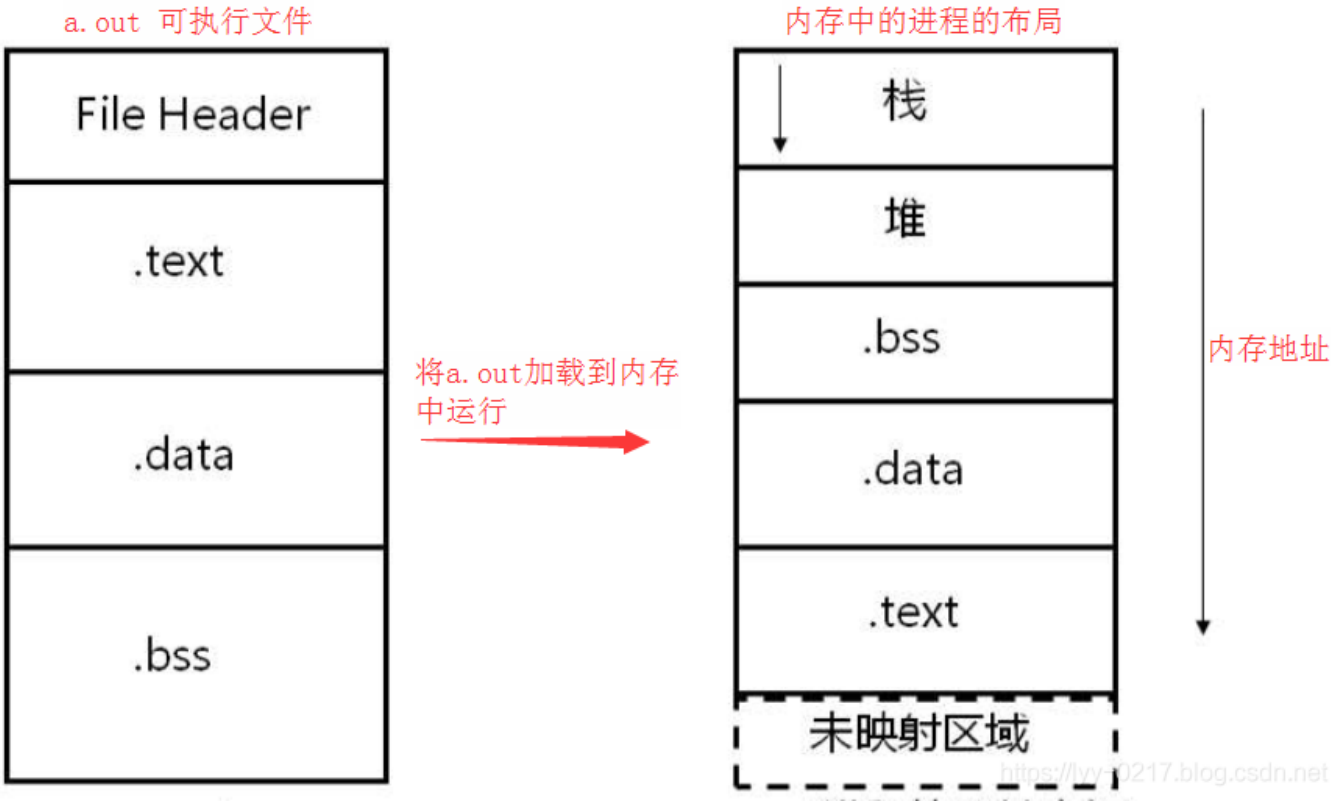
4.加载二进制镜像

2022.4.26，是我第一次写这一部分的笔记。说实在的，我现在并不是很清楚这个题目是什么意思、这里面函数之间有什么联系，只能先写一些零散的理解，日后还会回来补充。

2022.4.27，今天思考了一下进程和程序的关系，好像对**加载二进制镜像**这个说法又有了一点理解。程序是一个静态的概念，一个`.c`文件就可以是一个程序；而进程是一个相对动态的概念，简单的理解，就是正在内存当中被执行的可执行文件。那么这和**加载二进制镜像**有什么关系呢？首先我们来看一下可执行文件的构造过程。以下图片来自CSDN。



可以看到，左边的一个C程序被编译连接成了右边的ELF文件，ELF文件在此处就是可执行文件。可执行文件的各个数据段存放了C文件的各段内容。可执行文件载入内存中运行就产生了进程。

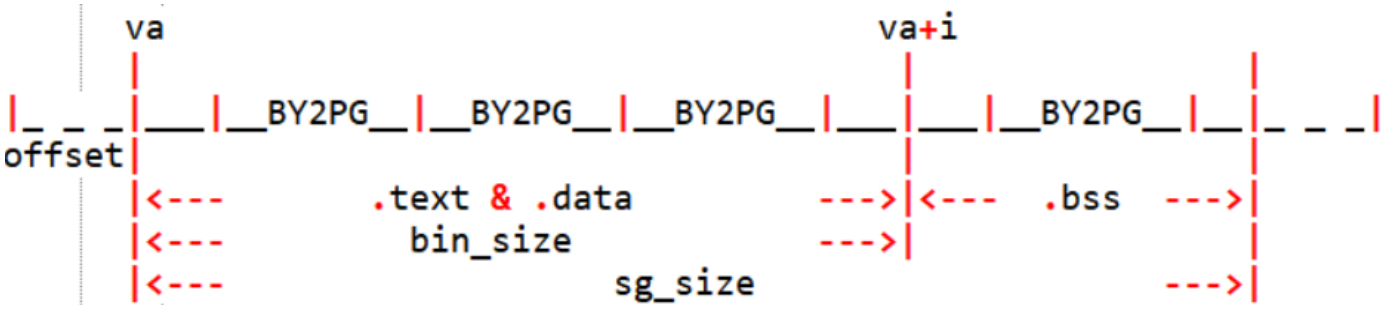


看到这个图，突然感觉明朗了一些，略微明白了什么是**加载二进制镜像**。我们需要将ELF文件的内容一一映射到内存当中，才能够生成一个进程，且`.text`、`.data`等段的位置确实有镜像的味道了。而且有了这么一张图，栈的初始化等操作也更加容易理解。

创建新的进程后，我们需要为该进程分配空间来容纳程序代码。我们使用两个函数`load_icode_mapper()`和`load_icode()`来实现这一功能。前者将在后者中被调用，故而先讲解前面这一个函数。

```
/* 此函数是的用途是为进程加载ELF的二进制文件。当然，这个函数并不是单独使用的，它需要配合另一个函
load_elf()使用，另一个函数帮助它解析了ELF文件的关键内容，并且传入这个函数。va是ELF文件的内核虚拟地
址，sgsize是ELF文件的段大小,bin是二进制区域的起始位置，bin_size是二进制区域的大小，user_data在面去
讲解 */
/*** exercise 3.6 ***/
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;
    struct Page *p = NULL;
    u_long i = 0;
    int r;
    u_long offset = va - ROUNDDOWN(va, BY2PG);
    /* 首先将二进制文件的所有内容加载到文件当中。Env结构需要通过页面来使用这些数据，故而我们在将ELF文
件的二进制内容复制到va对应的地址时，需要开辟页面并且传输给ENV结构体中的env_pgdir。这一部分的关键是按照
页对齐的方式向va拷贝数据 */
    /* Bad case:va is not round with BY2PG. */
    if(offset != 0) {
        if((r = page_alloc(&p)) < 0) {
            return r;
        }
        page_insert(env->env_pgdir, p, va, PTE_R);
```

```
        /* This is prepare for the worst case. */
        bcopy((void *)bin, (void *)(page2kva(p) + offset), MIN(bin_size, BY2PG -
offset));
        i = MIN(bin_size, BY2PG - offset);
    }
    for ( ; i < bin_size; i += BY2PG) {
        /* 声明新的页面，不断的拷贝直至i增长到二进制文件的大小。注意考虑页对齐的情况 */
        if((r = page_alloc(&p)) < 0) {
            return r;
        }
        page_insert(env->env_pgdir, p, va + i, PTE_R);
        /* MIN is prepare for the worst case. */
        bcopy((void *)bin + i, (void *)page2kva(p), MIN(bin_size - i, BY2PG));
    }
    /* 拷贝完二进制文件后，将剩余的ELF文件段加载到内存中 */
    while (i < sgsize) {
        if((r = page_alloc(&p)) < 0) {
            return r;
        }
        page_insert(env->env_pgdir, p, va + i, PTE_R);
        i += BY2PG;
    }
    return 0;
}
```



之前我们提到了页面对齐，我们的确要在保证将ELF文件关键内容拷贝到va地址的同时保证页面的完整且对齐。我们需要控制的几个关键部分是`va`、`bin_size`和`sg_size`我们思考一下我们需要处理什么情况。

最好的情况当然是`va`、`bin_size`和`sg_size`和BY2PG都对齐，这样我们只需要一直申请页面至`sg_size`即可。不过我们应该保证最坏的情况也能够被处理。最坏的情况是什么?那就是这三者都无法对齐，而我们需要在他们都无法对齐时照常分配页面，并且保证内容被拷贝到正确的地址上。下面我们来结合代码看看这一问题是如何解决的。

```
u_long i = 0;
int r;
/* offset对应了上面图中的偏移 */
u_long offset = va - ROUNDDOWN(va, BY2PG);
/* 如果偏移不为0，即va无法对齐，那么先声明一个页面，用于应对第一页无法对齐的情况。r是错误码，声明发生错
误时r得到的返回值小于0 */
if(offset != 0) {
    if((r = page_alloc(&p)) < 0) {
```

```
            return r;
        }
        /* 将新声明的页面插入到Env结构体的页目录中。这里使用page_insert()函数 */
        page_insert(env->env_pgdir, p, va, PTE_R);
        /* This is prepare for the worst case. */
        /* 将二进制文件的内容拷贝到目标位置。使用bcopy()函数进行拷贝。由于这是第一页，所以拷贝的起始位置是bin，拷贝的目标位置是page2kva(p)+offset，即va的地址。拷贝的长度为MIN(bin_size,BY2PG-offset)，这是考虑到了加载二进制文件后，可能还是只用到第一页的情况 */
        bcopy((void *)bin, (void *)(page2kva(p) + offset), MIN(bin_size, BY2PG - offset));
        i = MIN(bin_size, BY2PG - offset);
    }
    /* 解决了第一个页面的问题后，倘若i小于bin_size，我们开始处理剩下的页面 */
    for ( ; i < bin_size; i += BY2PG) {
        /* 声明新页面的操作和前面类似 */
        if((r = page_alloc(&p)) < 0) {
            return r;
        }
        page_insert(env->env_pgdir, p, va + i, PTE_R);
        /* MIN is prepare for the worst case. */
        /* 此处是考虑i+bin_size，即二进制内容的末尾和BY2PG无法对齐的情况。拷贝的长度变为MIN(bin_size-i,BY2PG)，这是为了处理最后一页无法对齐的情况 */
        bcopy((void *)bin + i, (void *)page2kva(p), MIN(bin_size - i, BY2PG));
    }
```

加载完二进制文件的内容，若`bin_szie`小于`sgsize`，还需要将页面申请直至增长到`sgsize`。后面的情况的处理思路和前面类似。

这个函数到此也就完善好了，不过我们提到，它还要结合另一个函数来发挥作用，那就是定义在`include/lib`下的`kernel_elfloader.c`中的`load_elf()`函数。

```
/* 此函数的作用是加载一个二进制的ELF，并将其所有的节都映射到正确的虚拟地址上。其二进制部分binary不能为空，且参数size是ELF文件二进制部分的大小。如果加载成功，binary的指针将被存储在start中 */
/*** exercise 3.7 ***/
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)) {
    Elf32_Ehdr *ehdr = (Elf32_Ehdr *)binary;
    Elf32_Phdr *phdr = NULL;
    u_char *ptr_ph_table = NULL;
    Elf32_Half ph_entry_count;
    Elf32_Half ph_entry_size;
    int r;
    /* 检查binary是否是二进制文件 */
    if (size < 4 || !is_elf_format(binary)) {
        return -1;
    }
    ptr_ph_table = binary + ehdr->e_phoff;
    ph_entry_count = ehdr->e_phnum;
    ph_entry_size = ehdr->e_phentsize;
```

```
    while (ph_entry_count--) {
        phdr = (Elf32_Phdr *)ptr_ph_table;
        if (phdr->p_type == PT_LOAD) {
            /* 调用函数进行映射 */
            if((r = map(phdr->p_vaddr, phdr->p_memsz, binary + phdr->p_offset, phdr->p_filesz, user_data)) < 0) {
                return r;
            }
        }
        ptr_ph_table += ph_entry_size;
    }
    *entry_point = ehdr->e_entry;
    return 0;
}
```

Thinking 3.3

`user_data`这一参数是`load_icode()`函数中的`Env *e`。这个参数必须存在，否则`load_elf()`函数无法向其函数参数`load_icode_mapper()`传递该参数。C语言的库函数中，快速排序函数qsort有着类似的行为：

```
void qsort(void *base, size_t n, size_t size, *int (*cmp)(const void *, const void *));
```

`size`给`cmp()`函数传递了信息。至于具体是什么信息，我想这可能和C++中的STL有一些类似的行为，比如告知`cmp()`比较目标的大小；换句话说，也就是起到了自定义模版的作用。

Thinking 3.4

复制情况的不同主要是由`va`、`va+bin_size`和`va+sg_size`是否能对齐决定的。下面将情况列表展示。

va	va+bin_size	va+sg_size	备注
对齐	对齐	对齐	最理想的情况
对齐	对齐	未对齐	va+sg_size所在页需要特殊处理
对齐	未对齐	对齐	va+bin_size所在页需要特殊处理
对齐	未对齐	未对齐	va+sg_size和va+bin_size所在页都需要特殊处理
未对齐	对齐	对齐	va所在页需要特殊处理
未对齐	对齐	未对齐	va和va+sg_size所在页都需要特殊处理
未对齐	未对齐	对齐	va和va+bin_size所在页都需要特殊处理
未对齐	未对齐	未对齐	这三者所在的页都需要特殊处理

最直接且暴力的方法就是处理最复杂的情况。我们使用`MIN()`函数，对相关的值进行比较来处理这些情况。具体方法已经写在笔记中了。注意，可能存在多种需要处理的情景在同一页面产生。

Thinking 3.5

`env_tf.pc`存储的是虚拟地址。CPU依靠`env_tf.pc`来寻找中断位置，那么这个域存储的自然是虚拟地址。

`entry_point`在每个进程中都是一样的，这种统一是因为每个进程创建的方式都相同。新进程的创建，都是按照我们写好的进程创建函数逐步执行。不过需要注意，`entry_point`只是一个虚拟地址，并非物理地址，故而每个进程中`entry_point`可能映射到不同的物理地址。

5.创建进程

由于我们在上面已经写好了一系列声明、初始化的函数，创建进程这一部分的工作量并不大。在这里，我们需要填写两个函数来创建进程，分别是`env_create()`和`env_create_proirity()`。

```
/* 使用env_create_priority()函数按照优先级创建进程 */
/*** exercise 3.8 ***/
void env_create(u_char *binary, int size) {
    /* 根据传入的参数调用函数env_create_priority()即可 */
    env_create_priority(binary ,size , 1);
}
```

另外一个函数稍微复杂一点。

```
/* 使用env_alloc()声明一个新的进程控制块，并且使用load_icode()将二进制文件binary加载到内存中。将新的进程控制块置于管理链表中 */
/*** exercise 3.8 ***/
void env_create_priority(u_char *binary, int size, int priority) {
    struct Env *e;
    int r;
    /* 使用env_alloc()声明一个新的进程控制块 */
    if((r = env_alloc(&e, 0)) < 0) {
        return;
    }
    /* 给该控制块赋予优先级 */
    e->env_pri = priority;
    /* 加载二进制文件并且将控制块插入管理链表 */
    load_icode(e, binary, size);
    LIST_INSERT_HEAD(&env_sched_list[0], e, env_sched_link);
}
```

使用我们之前写好的黑盒和一些宏就可以完成任务。这一部分的内容不难理解。

6.进程的运行与切换

还记得计组中的中断吗？在进入中断处理程序前，我们需要保存中断的位置和信息。同样，进程在发生切换的时候，我们也需要保存当前进程的周围环境（位置）和信息。这一部分的工作和驱动一个进程运行的工作结合了起来，在`env_run()`函数中实现。切换进程的流程为：

- 保存当前进程信息和上下文信息
- 切换`curenv`为即将执行的进程

- 调用`lcontext()`函数，设置全局变量`mContext`为当前进程的页目录地址（之后TLB的部分会用到）
- 调用`env_pop_tf()`汇编函数，恢复现场

下面根据代码来进行解读。

```
/* 进程运行函数 */
/*** exercise 3.10 ***/
void
env_run(struct Env *e) {
  /* 判断当前是否有进程运行，倘若有，使用Trapframe结构体保存当前进程的环境信息，并且拷贝到当前进程的env_tf域。然后，将env_tf.pc设置为env_tf.cp0.epc */
    if(curenv != NULL) {
        struct Trapframe *old;
        old = (struct Trapframe *)(TIMESTACK - sizeof(struct Trapframe));
        bcopy(old ,&(curenv->env_tf) ,sizeof(struct Trapframe));
        curenv->env_tf.pc = curenv->env_tf.cp0_epc;
    }
    /* 切换进程 */
    curenv = e;
    /* 调用函数。函数的具体信息目前还未了解，以后可能会补充 */
    lcontext(e->env_pgdir);
    env_pop_tf(&(e->env_tf) ,GET_ENV_ASID(e->env_id));
}
```

到这里，Lab3第一部分的填补代码的内容就差不多完成了。

Thinking 3.6

`epc`是进程发生中断时的指令的地址，这个值存储在`env_tf.cp0_epc`中。在恢复该进程时，应当从发生中断的指令继续向下执行，而不应该从头开始执行。因此将`pc`值设为中断的指令地址，那么下一次进入这个进程时，程序便会从此开始执行。

Thinking 3.7

在执行`sched_yield()`函数前，`KERNEL_SP`的一个`Trapframe`大小的内容被存到了`TIMESTACK`区域。这应该是一个用于保存现场的栈区，至于`sched_yield()`函数有什么作用，为什么要这样存储，刚做完第一部分的我可能还无法回答。

`TIMESTAKC`，顾名思义，显然发生时钟中断时的栈指针。`KERNEL_SP`是发生其他中断时的栈指针。

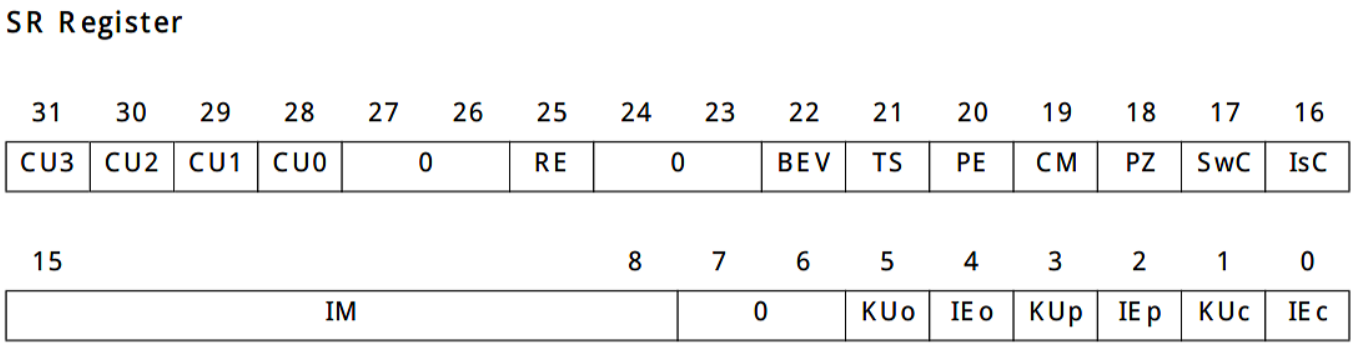
二、异常与中断

1.引言

首先搬运指导书的部分内容，来回顾计组P7有关协处理器CP0的部分知识。CP0中有许多寄存器，我们回顾以下三个寄存器。

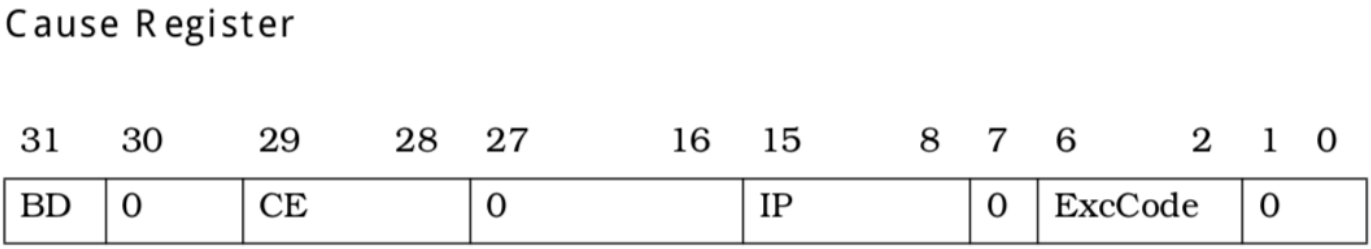
寄存器助记符	CP0寄存器编号	描述
SR	12	状态寄存器，包括中断使能位和CPU模式位等
Cause	13	记录导致异常的原因
EPC	14	异常结束后程序恢复执行的位置

SR寄存器的区域划分如下：



低六位的功能在前面已经介绍过了；IM区域是我们在计组课程中主要关注的区域。其中，8-9位是软件可写的中断位，10-15位对应了六种外部中断的使能位。

Cause寄存器的区域划分如下：



8-15位和SR寄存器的响应位相对应，记录了哪些中断和异常是已经发生的。ExcCode是异常标识码，记录异常发生的原因。在计组课程中，异常发生的原因有ALU溢出、取址不对齐等。

MIPS处理异常时要执行以下几个步骤：

1. 设置EPC，记录异常结束时需要返回的地址
2. 设置SR的相应位，使CPU进入内核态，关闭外部中断
3. 设置Cause寄存器，记录异常发生的原因
4. CPU进入异常处理程序

最后一个部分就是本部分我们要完成的内容。

2.异常的分发

在计组课程中，每当CPU触发异常，我们都会跳到一个0x4180的地址，进入异常处理程序。在MOS中，我们也要实现相应的功能，即当出现异常时，处理器能够根据异常码的种类跳转到相应的异常处理代码处。

`exercise3.12`要求我们将下一部分代码添加到`strat.S`中。

```
NESTED(except_vec3, 0, sp)
```

```
    .set noat
    .set noreorder
1:
    mfc0 k1,CP0_CAUSE          //将CP0_CAUSE寄存器的内容写入k1寄存器
    la k0,exception_handlers   //将exception_handlers（异常处理程序的地址）这一地址写入k0寄存器
    andi k1,0x7c               //取出2-6位的异常码，写回k1
    addu k0,k1                 //使用k0的地址加上k1的内容
    lw k0,(k0)                 //将k0的值存到其对应的地址处，相当于堆栈
    nop
    jr k0                      //跳转到k0寄存器的值所对应的地址处，即中断处理函数的入口地址
    nop
END(except_vec3)
.set at
```

`exc_vec3`段这一地址存放的是异常处理程序的入口地址。一旦 CPU 发生异常，就会自动跳转到地址 0x80000080 处，开始执行。将下面的代码加入`scse0_3.lds`中即是`exercise3.13`的内容。

```
. = 0x80000080;
.except_vec3 : {
    *(.text.exc_vec3)
}
```

3.异常向量组

前面提到的异常处理程序`ecxeption_handlers`还有一个名字， 就是异常向量组。它是定义在`lib/traps.c`中的一个`unsigned long`类型的数组，有32位。在这个文件中，我们做了对这一数组的初始化操作，即把相应异常码对应的异常处理函数的地址，填入到这个数组当中，以便操作系统能够根据异常码调用异常处理函数。

```
extern void handle_int();
extern void handle_reserved();
extern void handle_tlb();
extern void handle_sys();
extern void handle_mod();
unsigned long exception_handlers[32];

void trap_init()
{
    int i;
    for (i = 0; i < 32; i++) {
        set_except_vector(i, handle_reserved);
    }

    set_except_vector(0, handle_int);    //0号异常，实际上是中断，包括时钟中断、控制台中断等
    set_except_vector(1, handle_mod);    //1号异常，标识存储异常，程序对被标记为只读页面进行了写
操作
    set_except_vector(2, handle_tlb);    //2号异常，TLB异常，TLB中没有和程序地址匹配的有效入口
    set_except_vector(3, handle_tlb);    //3号异常，TLB异常，TLB失效切不在异常中（异常处理较快）
```

```
    set_except_vector(8, handle_sys);    //8号异常，系统调用，即执行syscall导致系统陷入内核态
}

void *set_except_vector(int n, void *addr)
{
    unsigned long handler = (unsigned long)addr;
    unsigned long old_handler = exception_handlers[n];
    exception_handlers[n] = handler;
    return (void *)old_handler;
}
```

我们在MOS实验中主要使用的是0号异常。

Thinking 3.8

- `handle_int`定义在`lib/genex.S`中，直接以mips中函数的形式给出。
- `handle_mod`、`handle_tlb`、`handle_reserved`这三个函数也是在`lib/genex.S`中实现的，不过它们的实现方法比较特殊。 咋一看，好像定义了很多函数，但和这三个函数没有任何的联系，不过我们来到文件末尾，会找到这样一段代码：

  ```
  BUILD_HANDLER reserved do_reserved cli
  BUILD_HANDLER tlb   do_refill   cli
  BUILD_HANDLER mod   page_fault_handler cli
  ```

  出现了熟悉的单词，但还仅仅是局部相似，不过确实可以找到`reserve`对应的`do_reserve`函数，`tlb`对应的`do_refill`函数，证明我们的寻找方向是正确的。我们可以猜测，这些函数的定义可能和`BUILD_HANDLER`有关。于是我们试着追溯一下它的源头，发现它正是定义在文件开头的一个宏：

  ```
  .macro  BUILD_HANDLER exception handler clear
      .align  5
      NESTED(handle_\exception, TF_SIZE, sp)
      .set    noat
  ```

  虽然不能完全明白这里的MIPS语法，但从`handle_\exception`可以看出，这里略有拼接的味道了，再结合之前的分析，我们可以肯定这三个函数也是定义在`lib/genex.S`下的。
- 前面几个函数都是定义在`.S`文件中的汇编函数，那么`handle_sys`函数也应当是如此。在`lib`目录下，可以找到一个`syscall.S`文件，`handle_sys`函数正是在其中实现。

4.时钟中断

本部分只有一个`exercise3.14`，且只用写一个简单的函数，就不再介绍这一部分的内容了。主要讲解时钟中断有关的知识。

首先来了解一下gxemul是如何模仿时钟中断的。首先，我们在`kclock_init()`函数中对时钟的信息进行了初始化。此函数调用了`set_timer()`汇编函数（这也是我们`exercise3.14`的内容），`set_timer()`定义在`kclock_asm.S`中。其内容大致如下：

```
LEAF(set_timer)

    //向t0寄存器中写入0xc8
    li t0, 0xc8
    //向0xb5000100中写入0xc8。0xb5000000是gxemul映射实时钟的位置，后者代表一秒钟触发中断200次
    sb t0, 0xb5000100
    //其它的初始化操作
    sw  sp, KERNEL_SP
setup_c0_status STATUS_CU0|0x1001 0
    jr ra

    nop
END(set_timer)
```

我们继续往前看。前面提到过，在发生时钟中断的时候，我们的程序将进入异常处理函数`handle_int`。

```
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)
```

首先，函数执行了一个`SAVE_ALL`宏，这个宏也是一个汇编函数，它将寄存器堆的内容和协处理器CP0的部分内容保存到了栈中。其函数的部分非常长，就不在此列出，只描述它的执行流程。

- 取出CP0中SR寄存器的内容，根据28位的内容，判断当前CPU是否处于用户模式下
- 将运行栈的地址保存到k0寄存器中
- 然后调用 get_sp 宏，根据中断异常的种类判断需要保存的位置
- 将运行栈地址即k0寄存器的内容推栈
- 将寄存器堆的内容和CP0部分寄存器的内容推栈

接下来，函数执行另外一个宏`CLI`，它设置了SR寄存器的`CU0`位和`IEc`位，修改操作系统的状态并标记中断已开启。然后，它对CP0_CAUSE的第四位进行判断，判断是否是时钟引起的中。如果是，则转而执行函数`timer_irq`；在后者中，还会跳转到另一函数`shced_yield`处执行程序。这一函数的内容大致如此，一些细节日后还可能会补充。

Thinking 3.9

首先来解释`set_timer`函数。

```
.macro  setup_c0_status set clr
    //.set push是一条为指令，貌似是指示汇编器保存当前的汇编状态。在Mars中，其好像已经表为无用
    .set    push
    //将SR寄存器的内容存入t0寄存器
    mfc0    t0, CP0_STATUS
    //将t0寄存器的内容与STATUS_CU0|0x1001取或并写回，其内容应当是0x10001001
    or  t0, \set|\clr
    //将t0寄存器的内容与0取或并写回。此时t0寄存器的28位、12位、0位都置1
    xor t0, \clr
    //写回SR寄存器，写回值的意义参考SR寄存器的描述。
    mtc0    t0, CP0_STATUS
    //恢复原有的汇编状态
    .set    pop
.endm

LEAF(set_timer)

    //向t0寄存器中写入0xc8
    li t0, 0xc8
    //向0xb5000100中写入0xc8。0xb5000000是gxemul映射实时钟的位置，后者代表一秒钟触发中断200次
    sb t0, 0xb5000100
    //将栈指针寄存器的内容存入内存中内核栈的位置
    sw  sp, KERNEL_SP
    //调用宏setup_c0_status，传入set参数为STATUS_CU0|0x1001，clr参数为0
    setup_c0_status STATUS_CU0|0x1001 0
    //跳转到ra寄存器的内容，即回到调用其的函数处继续执行程序
    jr ra
    nop
END(set_timer)
```

接下来看一看`timer_irq`函数。

```
timer_irq:
    //向0xb5001100写入0x0。该位置应当与时钟相关
    sb zero, 0xb5000110
    //跳转到sched_yield函数执行
1:  j   sched_yield
    nop
    /*li t1, 0xff
    lw    t0, delay
    addu  t0, 1
    sw  t0, delay
    beq t0,t1,1f
    nop*/
```

```
    //跳转到ret_from_exceptio处执行
    j   ret_from_exception
    nop
```

5.进程调度

时钟中断在此处是一种特殊的中断。之所以产生时钟中断，是为了在我们的操作系统中能够实现时间片轮转调度算法；而进程的切换需要系统进入中断。

我们在上面已经完成了"让系统能够正确进入中断服务函数"的工作。为了使时钟中断与轮转调度结合，我们需要在中断处理函数中完善`sched_yield()`函数，用于进程的分发调度。

```
void sched_yield(void)
{
  /* static变量只会在第一次生命的时候被赋值，以后的声明语句无效 */
  static int count = 0;
  static int point = 0;
  static struct Env *e = NULL;
  /* 倘若时间片count用完了，或者是进程不再处于可执行态，则需要切换进程 */
  if(count <= 0 || e->env_status != ENV_RUNNABLE) {
    do {
      /* 倘若当前point的队列为空，则切换到另外一个队列 */
      if(LIST_EMPTY(&env_sched_list[point])) {
        point = 1 - point;
      }

      /* 取出当前point所指队列的第一个进程控制块 */
      e = LIST_FIRST(&env_sched_list[point]);

      /* 如果该控制块不满足条件且不为NULL，那么将其插入队列尾部 */
      if(e != NULL) {
        LIST_REMOVE(e, env_sched_link);
        LIST_INSERT_TAIL(&env_sched_list[1 - point], e, env_sched_link);
        /* 为进程分配时间片，其值为之前分配给进程的pri */
        count = e->env_pri;
      }
    /* 控制块满足条件，跳出循环 */
    } while (e == NULL || e->env_status != ENV_RUNNABLE);
  }
  /* 进程执行，执行前时间片自减1 */
  count--;
  env_run(e);
}
```

Thinking 3.10

进程执行时，时钟会按照已有频率触发中断且提供属于自己的中断码。OS能够根据中断码跳转到特殊的中断处理程序——时钟中断处理程序对应的内容就是进程切换。我们在之前已经完成了保护现场、程序跳转等一系列工序，在此处只需按照轮转调度算法执行即可。

时钟每中断一次，都会进入一次`sched_yield()`，时间片`count`减一；当时间片用完，就会队列中选取满足条件的进程进行执行。

Lab3的实验笔记大致如此。