操作系统第四次实验报告-Lab3

以下为学生闫思桥(19241091)的$Lab3$实验报告

$Part1$ 实验思考题

Thinking 3.1

思考envid2env 函数:

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?

MyAnswer

envid2env()函数的代码如下：

```
int envid2env(u_int envid, struct Env **penv, int checkperm)
{
    struct Env *e;
    /* Hint: If envid is zero, return curenv.*/
    if (envid == 0){
        *penv = curenv;
        return 0;
    }
    /* Step 1: Assign value to e using envid. */
    e = envs + ENVX(envid); // ENVX 就是取出 envid的 低10位  总共有1024个进程控制块 结合上面的envid结构图，这里取出来的就是idx部分
    // 此时的envs是 0x80432000
    if (e->env_status == ENV_FREE || e->env_id != envid) {
        *penv = 0;
        return -E_BAD_ENV;
    }
    /*  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;
}
```

可以看出：这里是因为envid2env()函数在判断 `if (e->env_status == ENV_FREE || e->env_id != envid)` 这一步之前，仅仅是根据envid的低10位，也就是idx来在envs中寻找进程。

判断 `e->env_status == ENV_FREE` ，如果结果为否，可以推断出此时idx对应的这个进程控制块不在 `env_free_list` 中， 但是 `e->env_id != envid` 也是需要的。因为不同的进程（ASID不同）可能调用同一个进程控制块。如果某一个进程已经释放了，还拿着原来的id想找这个进程，没有这一步判断，很可能找到的这个进程控制块 e 已经被重新分配给了新进程，那么其e->envid值就是新进程的envid，自然不等于我们拿着的之前的进程的envid。因此， `e->env_id != envid` 这一步判断是有必要的，并且返回报错。

这里可以举出一个测试函数env_check()中的例子：

```
assert(env_alloc(&pe2, 0) == 0);
pe2->env_status = ENV_FREE;
re = envid2env(pe2->env_id, &pe, 0);
```

```
assert(pe == 0 && re == -E_BAD_ENV);

pe2->env_status = ENV_RUNNABLE;
re = envid2env(pe2->env_id, &pe, 0);

assert(pe->env_id == pe2->env_id &&re == 0); // 此时pe和pe2指向的是同一个进程控制块

temp = curenv;
curenv = pe0;
re = envid2env(pe2->env_id, &pe, 1);     // 置1的时候，就会进行检查，pe2 指向的进程控制块代表的进程不是 pe0 指向的进程控制块代表的进程，也不是其子进程，所以会报错
assert(pe == 0 && re == -E_BAD_ENV);
curenv = temp;
```

Thinking 3.2

结合include/mmu.h 中的地址空间布局，思考env_setup_vm 函数：

- **UTOP 和ULIM 的含义分别是什么，UTOP 和ULIM 之间的区域与UTOP以下的区域相比有什么区别?**
- **请结合系统自映射机制解释Step4中pgdir[PDX(UVPT)]=e->env_cr3的含义。**
- **谈谈自己对进程中物理地址和虚拟地址的理解。**

MyAnswer

- **UTOP 和ULIM 的含义分别是什么，UTOP 和ULIM 之间的区域与UTOP以下的区域相比有什么区别?**

ULIM是用户内存和内核内存的分界线。虚拟地址ULIM以上的地方，kseg0和kseg1两部分内存的访问不经过TLB，它们不归属于某一个进程，而是由内核管理的。

用户环境将不具有ULIM之上的任何内存的权限，而内核将能够读取和写入该内存。对于地址范围[UTOP，ULIM），内核和用户环境都具有相同的权限：它们可以读取但不能写入该地址范围。该地址范围用于向用户环境公开某些内核数据结构。最后，UTOP下的地址空间供用户环境使用；用户环境将设置访问该内存的权限。

概括来说：UTOP是用户进程读写部分的最高地址，ULIM是用户进程的最高地址。UTOP 和ULIM 之间的区域为用户进程的进程块还有页表，不能被用户自己更改，只能读取。

- **请结合系统自映射机制解释Step4中pgdir[PDX(UVPT)]=e->env_cr3的含义。**

首先，需要注意到：为每一个进程申请的页表（当作页目录），是为了映射整个4G空间，并基于此做出如下回答。

用 “为了这个进程而申请的页表” 去映射0-4G，那么一个页表项映射4M空间（一个PDMAP大小）。

MOS 中，将整个4G空间映射到了 UVPT-ULIM 这一区域，而 0-4G-1 的 22位-31位恰好是 0-1023 的整数（也可以理解为是除以4M）。那么 UVPT-ULIM 这一区域在 “表示整个4G空间的页表” 中，就应该是第 PDX(UVPT)项。

而 “表示整个4G空间的页表” 的这一项放入了 自己的基地址 env_cr3 = pgdir（pgdir就是 “表示整个4G空间的页表” 它自己的基地址），相当于用一个页表项来映射整个页表，恰似整个4G空间中取出 UVPT-ULIM 这一区域来映射整个4G。

- **谈谈自己对进程中物理地址和虚拟地址的理解。**

进程中的物理地址对应着真实硬件上为了运行进程而分配的进程页目录空间、elf文件加载的空间……的地址。

进程中的虚拟地址则仅仅是进程自己的页目录中映射的4G空间中的虚拟地址，并不是内核虚拟地址4G。

这里摘录一段前辈学长的阐述：

> 每个进程看到的都是虚拟地址空间，进程需要的物理地址空间存储在进程的页表当中。物理地址空间是硬件有多大地方就会编址到多大，虚拟空间则是约定好的，也就是人为设置的。对于不同的进程来说。每个进程都有着各自独立的虚拟地址空间，这样进程切换时不同的进程对相同的虚拟地址空间进行访问时互不影响。

Thinking 3.3

找到 user_data 这一参数的来源，思考它的作用。没有这个参数可不可以？为什么？（可以尝试说明实际的应用场景，举一个实际的库中的例子）

MyAnswer

在load_icode()函数中的下面这一句，这里的(void*)e就是 userdata。

```
static void load_icode(struct Env *e, u_char *binary, u_int size)
//将开头地址为binary，大小为size的二进制文件（程序代码），加载到进程e的页目录映射的二级页表体系
r = load_elf(binary, size, &entry_point, (void *)e, load_icode_mapper);
// 这里的(void*)e就是 userdata
```

这个参数就是用于在加载二进制文件的镜像中说明，这个被加载的二进制镜像（开头指针是binary，文件大小是size）是属于进程e的。

并且在load_icode_mapper()函数中，userdata 赋值了函数内部的局部变量

```
struct Env *env = (struct Env *)user_data;
```

并通过env->env_pgdir，将进程自己的页目录和二进制文件要在页目录映射的4G框架中的地址建立起映射关系，即：将一级页表基地址 env->env_pgdirpgdir 对应的两级页表结构中，在 va 这一虚拟地址 所对应的二级页表项中，填入p这个物理内存控制块（这里的p是二进制文件被真正加载到的内核虚拟地址区域对应的物理内存控制块）对应的物理页面的页号，并设置页表权限为可写。

不能没有这个参数。这个参数的存在方便了向**更内层**函数传值。C语言stdlib中有

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

就是一个例子，第三个参数告诉了程序应当如何分割base起始的内存数据，方便调用传入的比较函数进行排序操作。

Thinking 3.4

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？

MyAnswer

基于**一个进程只有一个ELF文件、ELF文件中各个段segment中的要加载到的虚拟地址vaddr是按照从低到高升序排列**的前提，作答如下：

设前一个加载到的segment为lastSeg，起始地址为L1，终止地址为R1；当前即将被加载到的segment为Seg，起始地址为L2，终止地址为R2。

1. 当Seg的起始地址和lastSeg的终止地址在同一页上的时候，此时不能直接用page_insert()函数，而应该用page_lookup()函数去找到lastSeg的终止地址对应的物理内存控制块。在同一页虚拟地址上进行复制。
2. 当Seg的长度不足一页的时候，复制的长度应该考虑这一点，因此应该是 `MIN(BY2PG - offset, bin_size)` 。
3. 当一页一页地进行复制时，要时刻注意文件的剩余长度够不够一页，复制的长度是 `MIN(BY2PG, bin_size - i)` 。其中 i 是在复制完offset部分后，进行整页整页复制的索引。

Thinking 3.5

思考上面这一段话，并根据自己在lab2 中的理解，回答：

- **你认为这里的 env_tf.pc 存储的是物理地址还是虚拟地址?**
- **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?**

MyAnswer

- **env_tf.pc 存储的是物理地址还是虚拟地址?**

存储的是虚拟地址。在load_icode()函数中有如下的代码

```
r = load_elf(binary, size, &entry_point, (void *)e, load_icode_mapper);
// 这里的(void*)e就是 userdata
if (r < 0) {
    return;
}
e->env_tf.pc = entry_point;
```

load_elf()函数将二进制文件binary加载到对应的虚拟内存中，并将程序入口返回，存储在entry_point里。而entry_point又是由load_elf()函数中的赋值语句得来。

```
Elf32_Ehdr *ehdr = (Elf32_Ehdr *)binary;
*entry_point = ehdr->e_entry;
// Elf32_Addr      e_entry;         /* Entry point virtual address */
```

e_entry是二进制文件入口点的虚拟地址。

> 顺序执行时PC永远都是进行+4操作，但一个进程程序段可能不在一段连续的物理空间上，因此显然PC应当存放虚拟地址。

- **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?**

不一样。entry_point记录的是进程的elf文件入口的虚拟地址，但是这个虚拟地址是相对于进程而言的，是在进程的页目录所映射的二级页表体系的4G空间视野下的，并不是内核虚拟地址，只和进程本身有关系。所以，对于不同的进程来说，entry_point可能一样，可能不一样。这里记录一下一位学长的观点：

> 应当是一样的。存放虚拟地址的entry_point没有必要不一样，因为本身就可以从页表映射到不同的物理地址。如果设计成不一样会给操作系统带来更多麻烦事，一致则可以减少复杂度。

Thinking 3.6

请查阅相关资料解释，上面（env_run()函数）提到的epc是什么？为什么要将env_tf.pc设置为epc呢?

MyAnswer

epc是MIPS中，cp0协处理器的第14号寄存器的名字。它用于：当中断异常发生时, 该寄存器记录当前的PC寄存器值。也就是异常结束后程序恢复执行的位置 。

设置的原因要先看这个赋值语句在哪里。

```
if (curenv != NULL) { // 当前运行进程不为空，保存当前进程上下文
    struct Trapframe *old = (struct Trapframe*)( TIMESTACK-sizeof(struct
Trapframe) );  // 记录当前进程的环境
    bcopy((void *) old, (void *)(&(curenv->env_tf)),
          sizeof(struct Trapframe));
    curenv->env_tf.pc = curenv->env_tf.cp0_epc;
}
curenv = e;
```

这里，env_run(struct Env *e)函数为了运行传入的参数进程e，在当前运行的进程不为空的时候，是需要先中断当前运行的程序，再把代表当前运行进程的指针curenv指向参数进程e。

因此，需要把被中断的进程的epc存放在被终端进程的协处理器cp0的pc寄存器里。

Thinking 3.7

关于 TIMESTACK，请思考以下问题：

- **操作系统在何时将什么内容存到了 TIMESTACK 区域**
- **TIMESTACK 和 env_asm.S 中所定义的 KERNEL_SP 的含义有何不同**

MyAnswer

- 操作系统在何时将什么内容存到了 TIMESTACK 区域

在env_destroy()时，执行了

```
bcopy((void *)KERNEL_SP - sizeof(struct Trapframe),
              (void *)TIMESTACK - sizeof(struct Trapframe),
              sizeof(struct Trapframe));
```

在env_run()时，执行了

```
 if (curenv != NULL) { // 当前运行进程不为空，保存当前进程上下文
     struct Trapframe *old = (struct Trapframe*)( TIMESTACK-sizeof(struct
Trapframe) );
     bcopy((void *) old, (void *)(&(curenv->env_tf)),
           sizeof(struct Trapframe));
     curenv->env_tf.pc = curenv->env_tf.cp0_epc;
 }
```

- TIMESTACK 和 env_asm.S 中所定义的 KERNEL_SP 的含义有何不同

env_asm.S 中所定义的 `KERNEL_SP` 就是一个4字长的全局变量，意为内核栈顶。在目前的体系中，`KERNEL_SP` 在set_timer函数中被初始化为用户栈sp。

```
.data
        .global    KERNEL_SP;
KERNEL_SP:
        .word     0
```

`TIMESTACK` 则是一个定值，它是 kseg0 地址 `0x8200 0000`，在mmu.h中以宏的形式被定义。它是用于处理时钟中断的异常处理栈，是发生时钟中断时存放寄存器状态的栈顶地址。处理时钟中断的异常处理栈的具体范围是$[\mathrm{TIMESTACK}-\mathrm{BY2PG},\mathrm{TIMESTACK})$。

结合lab4中系统调用的内容，`TIMESTACK` 是发生时钟中断异常时用到固定的的栈指针。`KERNEL_SP` 是发生其他中断时栈指针的值。

Thinking 3.8

试找出上述 5 个异常处理函数的具体实现位置。

> **0 号异常**的处理函数为 `handle_int`，表示中断，由时钟中断、控制台中断等中断造成
>
> **1 号异常**的处理函数为 `handle_mod`，表示存储异常，进行存储操作时该页被标记为只读
>
> **2 号异常**的处理函数为 `handle_tlb`，TLB 异常，TLB 中没有和程序地址匹配的有效入口
>
> **3 号异常**的处理函数为 `handle_tlb`，TLB 异常，TLB 失效，且未处于异常模式（用于提高处理效率）
>
> **8 号异常**的处理函数为 `handle_sys`，系统调用，陷入内核，是执行了 syscall 指令造成的

MyAnswer

handle_int函数的实现在lib/genex.S，本身汇编代码就描述了函数的作用。

handle_mod和handle_tle函数的实现也在lib/genex.S，但是它们是通过另一个汇编宏函数 `BUILD_HANDLER` 来实现的，而且各自对应了其他处理函数。handle_mod对应了 `page_fault_handler` ；handle_tle对应了 `do_refill` 。

handle_sys函数的实现在lib/syscal.S，本身汇编代码就描述了函数的作用。

除了handle_sys函数，其他三个函数都会在后面的总结中详细分析。

Thinking 3.9

阅读 kclock_asm.S 和 genex.S 两个文件，并尝试说出 set_timer 和timer_irq 函数中每行汇编代码的作用。

MyAnswer

set_timer

```
.macro  setup_c0_status set clr
    .set    push
    mfc0    t0, CP0_STATUS          # 将CP0_STATUS的内容存入 t0
    or      t0, \set|\clr           # 将t0（SR寄存器）的第0、12、28位置1  IEc 4号中断 CU0
    # (0bit)允许中断 (12bit) 4 号中断可以被响应 (28bit)允许在用户模式下使用 CP0 寄存器
    xor     t0, \clr                # 将t0和0进行异或，
    mtc0    t0, CP0_STATUS
    .set    pop
.endm

    .text
```

```
LEAF(set_timer)

    li t0, 0xc8                 # 0xc8表示1秒钟中断200次
    # 0xb5000000 是模拟器(gxemul) 映射实时钟的位置。偏移量为 0x100 表示来设置实时钟中断的
频率，0xc8 则表示1 秒钟中断200次，如果写入0，表示关闭实时钟。
    sb t0, 0xb5000100           # 此处填入 0 就会关闭实时钟
    sw  sp, KERNEL_SP           # 初始化内核栈顶 KERNEL_SP 为 用户栈sp
setup_c0_status STATUS_CU0|0x1001 0     # 0x10000000|0x1001 = 0x10001001
    jr ra

    nop
END(set_timer)
```

timer_irq

首先写 0xb5000110 地址响应时钟中断，之后跳转到 sched_yield 中执行。而 sched_yield 函数会调用引起进程切换的函数来完成进程的切换。注意：这里是第一次进行进程切换，请务必保证：kclock_init 函数在 env_create 函数之后调用

```
timer_irq:
    sb zero, 0xb5000110             # 写 0xb5000110 地址响应时钟中断
1:  j   sched_yield                 # 跳转到 sched_yield 中执行，进程调度
    nop
    /*li t1, 0xff
    lw    t0, delay
    addu  t0, 1
    sw  t0, delay
    beq t0,t1,1f
    nop*/
    j   ret_from_exception          # 恢复现场和回滚
    nop
```

$Part2$ 实验难点图示

$Part2.1$ 进程

进程既是基本的分配单元，也是基本的执行单元。**每个进程都是一个实体，有其自己的地址空间，通常包括代码段、数据段和堆栈。程序是一个没有生命**的实体，只有被处理器赋予生命时，它才能成为一个活动的实体，而执行中的程序，就是我们所说的进程。

进程控制块

进程控制块 (PCB) 是系统专门设置用来管理进程的数据结构，它可以记录进程的外部特征，描述进程的运动变化过程。系统利用 PCB 来控制和管理进程，所以 **PCB是系统感知进程存在的唯一标志。进程与 PCB 是一一对应的**。通常 PCB 应包含如下一些信息：

```
struct Env {
     struct Trapframe env_tf;        // Saved registers
     LIST_ENTRY(Env) env_link;       // 类似于pplink，含有两个指针，用于构造空闲进程队列
     u_int env_id;                   // Unique environment identifier
     u_int env_parent_id;            // env_id of this env's parent
     u_int env_status;               // Status of the environment
     Pde *env_pgdir;                 // 进程页目录的内核虚拟地址
     u_int env_cr3;                  // 进程页目录的物理地址
     LIST_ENTRY(Env) env_sched_link;// 类似于pplink，含有两个指针，用于构造进程调度队列
     u_int env_pri;                  // 进程优先级
};
```

```
LIST_HEAD(Env_list, Env);       // 也就是说，可以用Env_list去定义一个struct Env链表的
头节点Head
// 即 struct Env_list{ struct Env* lh_first }
static struct Env_list env_free_list;    // Free list
struct Env_list env_sched_list[2];      // Runnable list
#define LIST_HEAD(name, type)                                                   \
        struct name {
\
                struct type *lh_first;   \  /* first element */

        }
#define LIST_ENTRY(type)                                                        \
        struct {
\
                struct type *le_next;   /* next element */
 \
                struct type **le_prev;  /* address of previous next element */
 \
        }
#define LIST_NEXT(elm, field)   ((elm)->field.le_next)
#define LIST_FIRST(head)        ((head)->lh_first)
#define LIST_FOREACH(var, head, field)                                          \
        for ((var) = LIST_FIRST((head));                                        \
                (var);                                                          \
                (var) = LIST_NEXT((var), field))
```

- env_tf : Trapframe 结构体的定义在include/trap.h 中，在发生进程调度，或当陷入内核时，会将当时的进程上下文环境保存在env_tf变量中。

  Trapframe 结构体和后面的TF_打头的宏是一一对应的，后面会具体介绍。

- env_link : env_link 的机制类似于lab2中的pp_link, 使用它和env_free_list来构造空闲进程链表。
- env_id : 每个进程的env_id 都不一样，它是进程独一无二的标识符。
- env_parent_id : 在之后的实验中，我们将了解到进程是可以被其他进程创建的，创建本进程的进程称为父进程。此变量记录父进程的进程 id，进程之间通过此关联可以形成一棵进程树。
- env_status : 该变量只能有以下三种取值：

1. ENV_FREE : 表明该进程是不活动的，即该进程控制块处于进程空闲链表中。
2. ENV_NOT_RUNNABLE : 表明该进程处于阻塞状态，处于该状态的进程需要在一定条件下变成就绪状态从而被CPU调度。（比如因进程通信阻塞时变为 ENV_NOT_RUNNABLE，收到信息后变回 ENV_RUNNABLE）
3. ENV_RUNNABLE : 表明该进程处于**执行状态或就绪状态**，即其可能是正在运行的，也可能正在等待被调度。

- env_pgdir : 这个变量保存了该进程页目录的内核虚拟地址。

> **MyNote**：这里的页目录就是在创建进程的时候，在**page_free_list**中申请出来的一页内存，专门用于当作这个进程的页目录。

- env_cr3 : 这个变量保存了该进程页目录的物理地址。
- env_sched_link : 这个变量用来构造调度队列。

> **MyNote**：可以看到这个变量其实是**LIST_ENTRY(Env)**类型的，也就是说和**lab2**中的**pp_link**角色是一样的，都是包含两个指针的链表项。

- env_pri : 这个变量保存了该进程的优先级。

> 在我们的实验中，存放进程控制块的物理内存在系统启动后就要被分配好，也就是envs数组。

这里就是mm/pmap.c中的mips_vm_init函数。相应的代码已经解析过了。

进程初始化

先介绍和复习几个宏和定义

```
#define LOG2NENV    10
#define NENV        (1<<LOG2NENV)    // 这里可以看出，最多进程数是2^10=1024个
#define ENVX(envid) ((envid) & (NENV - 1)) // ENVX 就是取出 envid的 低10位
#define GET_ENV_ASID(envid) (((envid)>> 11)<<6) // 就是envid中的ASID部分左移6位，相当于拼出了EntryHi的0-5的NULL位和6-11的ASID位
```

当然，有了存储进程控制块信息的 envs 还不够，我们还需要像 lab2 一样将空闲的 env 控制块按照链表形式“串”起来，便于后续分配 ENV 结构体对象，形成 env_free_list。 一开始我们的所有进程控制块都是空闲的，所以我们要把它们都“串”到 env_free_list 上去。这里就用到了 `env_init()` 函数。

env_init()

这个函数首先初始化用于分配进程控制块的env_free_list以及表示进程调度的 `env_sched_list[0]` 和 `env_sched_list[1]` 。然后将 在mips_vm_init() 中分配的 NENV 个进程控制块 **按照 NENV-1, … , 0 的倒序** 插入env_free_list的头部，这样最先分配出来的是第0个进程控制块，也就是从低地址到高地址分配。并设置他们的进程状态 `env_status = ENV_FREE` 。

这样，envs数组下标正好对应链表中的由前到后的顺序，调用空闲进程时优先调用下标最小的。

```
void env_init(void)
{
    int i;
    /* Step 1: Initialize env_free_list. */
    LIST_INIT(&env_free_list);
    LIST_INIT(&env_sched_list[0]);
    LIST_INIT(&env_sched_list[1]);
    /* 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. */
    for (i = NENV-1; i >= 0; i--) {
        envs[i].env_status = ENV_FREE;
        LIST_INSERT_HEAD(&env_free_list, &envs[i], env_link);
    }
}
```

进程的标识——asid相关管理和envid详细解析

电脑中经常有很多进程同时存在，每个进程执行不同的任务，它们之间也经常需要相互协作、通信，那操作系统是如何识别每个进程呢？毫无疑问，在进程控制块PCB（即我们实验中的envs结构体）中的 `envid` 就是每个进程独一无二的标识符，**需要在进程创建的时候就被赋予。**

在 env.c 文件中可以找到一个叫做 mkenvid 的函数，它的作用就是生成一个新的进程 id。mkenvid 中调用的 asid_alloc 函数，这个函数的作用是为新创建的进程分配一个异于当前所有未被释放的进程的 ASID。

三个问题和解答

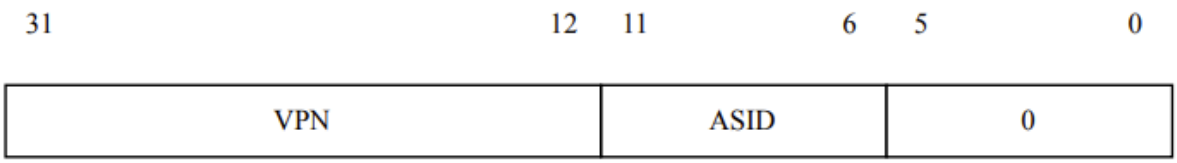
这个 ASID 是什么？为什么要与其他的进程不同呢？为什么不能简单的通过自增来避免冲突呢？

要解答这些问题，还需回到 TLB 的讨论：

根据 lab2 的学习我们得知进程是通过页表来访问内存的，**而不同的进程的同一个虚拟地址可能会映射到不同的物理地址。**

为了实现这个功能，TLB 中除了存储页表的映射信息之外，还会存储进程的标识编号，作为 Key 的一部分，用于保证查到的页面映射属于当前进程，而这个编号就是 ASID。显然，不同进程的虚拟地址是可以对应相同 VPN 的，而如果 ASID 也不具备唯一标识性，就与 TLB Field 的唯一性要求相矛盾了。因此，直到进程被销毁或 TLB 被 清空时，才可以把这个 ASID 分配给其他进程。

到此，前两个问题就得到了解答，而为了解答第三个问题，我们需要更加深入地了解 TLB 的结构。

> 我们采用的模拟器模拟的 CPU 型号是 MIPS R3000，其 TLB 结构的Key Fields（也就是 EntryHi 寄存器）中应用到了 ASID。
>
> 
>
>
> 图 3.1: TLB 对 ASID 的规定
>
> 可以发现，其中 ASID 部分只占据了 6-11 共 6 个 bit，所以如果单纯的通过自增的方式来分配 ASID 的话，很快就会发生溢出，导致 ASID 重复。
>
> 为了解决这个问题，我们采用限制同时运行的进程个数的方法来防止 ASID 重复。

具体实现是在asid_alloc()函数中，通过位图法管理可用的 64 个 ASID，如果当 ASID 耗尽时仍要创建进程，系统会 panic。这在下面的 `asid_alloc()` 函数中体现的淋漓尽致。

asid_alloc()

这个函数的作用是：**为新创建的进程分配一个异于当前所有未被释放的进程的 ASID。**

```
static u_int asid_bitmap[2] = {0};
static u_int asid_alloc() {
    int i, index, inner;
    for (i = 0; i < 64; ++i) {
        index = i >> 5;  // 相当于这个数除以32
        inner = i & 31;  // 01 1111 就相当于取到这个数的低5位 就相当于模运算32
        // 就相当于把64个数 搞成 除以32的index 和 余数 inner
        // 那么这也印证了下面的asid_bitmap数组其实也只有两个元素，
        // 就是两个u_int 类型的32位整数来代表 64 个ASID 是否被进程占用
        if ((asid_bitmap[index] & (1 << inner)) == 0) {
            asid_bitmap[index] |= 1 << inner;
            return i;
        }
    }
    panic("too many processes!");
}
```

那么整体函数的逻辑就很显然了，就是从0到63，遍历一遍由两个32位整数组成的64位，如果有代表空闲的0，就返回。

asid_free()

将位图管理模式下的$i(i \in [0,63])$对应的那个ASID对应的位 置零。

```
static void asid_free(u_int i) {
    int index, inner;
    index = i >> 5;
    inner = i & 31;
    asid_bitmap[index] &= ~(1 << inner);     // 对应位置置零，代表asid释放
}
```

mkenvid()

这个函数的作用就是为新创建的进程e分配一个异于当前所有未被释放的进程的 ASID。

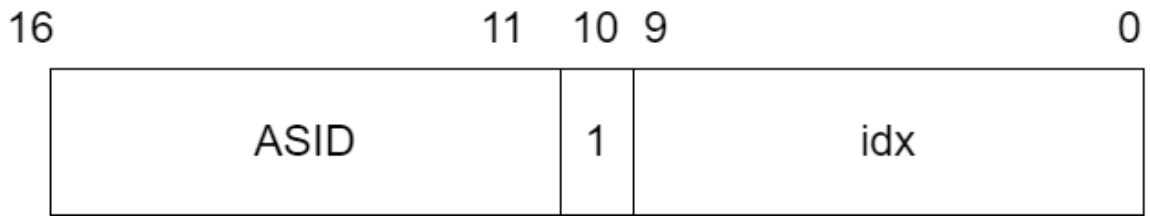
```
u_int mkenvid(struct Env *e) {
    u_int idx = e - envs;        // 算出这是第几个进程，总共不会超过NENV=1024
    u_int asid = asid_alloc();  // 用asid_alloc()函数申请一个asid
    return (asid << (1 + LOG2NENV)) | (1 << LOG2NENV) | idx;
} // LOG2NENV = 10
```

首先，算出这是第idx个进程；

然后开始拼接envid：

把申请到的asid右移11位 | 再把从左到右第10位置1 | idx。

那么我们可以画出下面的envid结构图，总共17位。



此外，非常值得注意的一点是，**mkenvid()函数不会返回0（在第10位上始终是1）。这是因为在lab4涉及到的进程间通信（IPC）中，会大量使用srcva为0的调用来表示只传value值，而不需要传递物理页面。换句话说，当srcva不为0时，我们才建立两个进程的页面映射关系。**

再联系envid2env函数中对于 `envid==0` 的特判（直接向参数中存入当前进程的物理内存控制块），就可以说明：**envid==0 是特意空出来的，用于使用envid2env()函数找到“当前进程 `curenv` ”：`envid2env(srcid,  &srcenv, 0), srcid == 0` 。事实上，在lab4中，envid为0是MOS修改了子进程的函数返回值，用于区分父子进程。MOS希望系统调用在内核态返回的 envid 只传递给父进程，对于子进程则需要对它的保存的现场Trapframe进行一个修改，从而在恢复现场时用 0 覆盖系统调用原来的返回值。**

envid2env()

这个函数实现：**通过一个 env 的 id 获取该 id 对应的进程控制块的功能。**

当参数checkperm为1时，还要判断传入的envid对应的进程e，要么就是当前进程curenv本身，要么就是当前进程curenv的子进程。当然，这个函数的通用性并不非要是父进程创建子进程或者当前进程，所以这个参数可以设置为0。（可以看作是一个进程中调用这个函数，那么它拿着一个envid去找进程，要么是它自己，要么是它的子进程。不应该找到其他进程。在lab4中，父进程为子进程申请物理页面的时候，就应该将checkperm置为1）

```
int envid2env(u_int envid, struct Env **penv, int checkperm)
{
    struct Env *e;
```

```
    /* Hint: If envid is zero, return curenv.*/
    if (envid == 0){
        *penv = curenv;
        return 0;
    }
    /* Step 1: Assign value to e using envid. */
    e = envs + ENVX(envid); // ENVX 就是取出 envid的 低10位  总共有1024个进程控制块 结合上面的envid结构图，这里取出来的就是idx部分
    // 此时的envs是 0x80432000
    if (e->env_status == ENV_FREE || e->env_id != envid) {
        *penv = 0;
        return -E_BAD_ENV;
    }
    // 检查调用进程env是否有足够的权限来操作指定的进程env。如果设置了checkperm位是1，则指定的env必须是curenv或curenv的直接子级。如果不是，返回-E_BAD_ENV错误!
    /*  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;
}
```

这里举一个测试函数load_icode_check()中的例子

```
assert(envid2env(1024, &e, 0) == 0); // 这里的envid就是ASID为0且idx为0
```

总结

由上面几个函数可以看出：

- envs 数组
- env_free_list、env_sched_list[2]链表

是两种查找进程控制块的方式。

1. 无论哪一个进程控制块，总能通过ENVX(envid)来取到envid的低10位——该进程控制块在envs数组中的下标idx，从而用&envs[idx]拿到它的地址。
2. 当进程控制块的状态是ENV_FREE时，它一定在env_free_list中，并且由于初始化的时候是按下标从大到小插入链表，所以第一次调用空闲进程时优先调用下标最小的。当进程被释放时，对应的进程控制块会被插回链表的头部。当然，此时链表中从头到尾的下标就不一定是严格升序了。（链表法不可能始终维持有序。）
3. 当进程控制块的状态是ENV_NOT_RUNNABLE或者ENV_RUNNABLE时，它一定在env_sched_list[0]或者env_sched_list[1]中的一个中。

设置进程控制块

env_setup_vm()

这里复习一下：boot_pgdir的第PDX(UPAGES)=510项就是记录了物理内存控制块们所对应的一级页表项，第PDX(UNENS)=PDX(UTOP)=509项就是记录了进程控制块们对应的一级页表项。

这个函数为每一个进程e（以进程控制块的形式表示），都申请一页物理内存作为进程e的专属页目录（表现为从page_free_list里申请一个物理内存控制块p）。再把p通过page2kva函数转为物理页对应的虚拟地址pgdir，利用pgdir，

将$[0, PDX(UTOP))$的一级页表项置零，

将$[PDX(UTOP),PDX(ULIM)\ )$对应的一级页表项从boot_pgdir拷贝过来。

> 在我们的实验中，虚拟地址ULIM以上的地方，kseg0和kseg1两部分内存的访问不经过TLB，它们不归属于某一个进程，而是由内核管理的。
>
> 在这里，操作系统将一些内核的数据暴露到用户空间，使得进程不需要切换到内核态就能访问，这也是MOS的微内核设计。我们将在lab4和lab6中用到此机制。而这里我们要暴露的空间是UTOP以上ULIM以下的部分，也就是把这部分内存对应的内核页表拷贝到进程页表中。
>
> **MyNote**
>
> 由于用户态不能访问到高2G，那么这里其实不用复制到1024（对应着4G），只需要复制到PDX(ULIM)-1 = 511项即可。
>
> 又因为PDX(UVPT)需要单独设定以实现自映射，所以**复制的区域只有PDX(UTOP)<= i < PDX(UVPT)，只有第509，510两项——分别是物理内存控制块们对应的一级页表项和进程控制块们对应的一级页表项**。然后单独设置第PDX(VPT)=511项，以实现自映射机制。
>
> 每一个进程页目录的一个页表项映射一个PDMAP=4MB大小的空间。当然，具体来说，为了更精确的表示，每一个一级页表项中其实存放的是一个二级页表的物理地址，由这个二级页表去表示这个PDMAP。一个二级页表项表示4K大小的空间，正好是一个页的大小BY2PG。在进程页目录的PDX(UVPT)项放入进程页目录自己的物理地址 | PTE_V，也是把页目录作为其中一个二级页表放在了一级页表项中。以此来建立自映射机制。

```
static int env_setup_vm(struct Env *e)
{
    int i, r;
    struct Page *p = NULL;
    Pde *pgdir;

    if ( (r = page_alloc(&p)) < 0 ) {    // 每申请一个进程，都需要申请一页物理内存，p是对应的物理内存控制块地址。这一页物理内存映射的是整个4G
        panic("env_setup_vm - page alloc error\n");
        return r;
    }
    p->pp_ref++;                         // 对应的物理内存页 引用次数加 1
    pgdir = (Pde*) page2kva(p);          // 由物理内存控制块的指针得到其控制的物理页面对应的内核虚拟地址
    e->env_cr3 = PADDR(pgdir);           // 进程 e 的页目录的物理地址
                                         // (由内核虚拟地址转化，减去ULIM，最高位置零)
    e->env_pgdir = pgdir;                // 进程 e 的页目录的内核虚拟地址，管辖一页大小的内核虚拟空间

    // 0-4G-1 的高10位 (22位-31位)   就是 0 - 1023
    for (i = 0; i < PDX(UTOP); i++) { // i < 509 将这些一级页表项置零
        pgdir[i] = 0;
    }

    for(i = PDX(UTOP); i < PDX(UVPT); i++) {
        // 其实这里复制的只有两项 PDX(UTOP)=PDX(UENVS)=509,PDX(UPAGES)=510;
        // PDX(UVPT)=511下面会单独设置，然后PDX(ULIM)=512到4G对应的1024是高2G，高两G分配地址的时候从来不在页表上登记
        pgdir[i] = boot_pgdir[i];
    }

    // 不同进程的页目录，在PDX(UTOP)及以上的页表项 (除了PDX(UVPT)) 都相同
    /* UVPT maps the env's own page table, with read-only permission.*/
    e->env_pgdir[PDX(UVPT)]  = e->env_cr3 | PTE_V;
    // UVPT = 7fc0 0000, PDX(UVPT) = 111 1111 11 = 511
    // 这里相当于是用 "为了这个进程而申请的" 页表 映射0-4G，那么一个页表项映射4M空间 (一个PDMAP大小)
    // 而 0-4G-1 的 22位-31位恰好是 0-1023 的整数，
    //  MOS 中，将页表和页目录映射到了 UVPT-ULIM 这一区域，那么这一区域在 "表示整个4G空间的页表" 中，就应该是第 PDX(UVPT)项
```

```
    // 而“表示整个4G空间的页表”的第PDX(UVPT)项放入了 自己的基地址 env_cr3 = pgdir
 (pgdir就是“表示整个4G空间的页表”自己的基地址)，
    // 恰似整个4G空间中取出 UVPT-ULIM 这一区域来映射整个4G。
    return 0;
}
```

env_alloc()

为envid是`parent_id`的父进程新申请一个进程（当然`parent_id`可以是0，这样就表示没有父进程），并将这个子进程对应的进程控制块的地址存在`struct Env **new`中。

```
int env_alloc(struct Env **new, u_int parent_id)
{
    int r;
    struct Env *e;

    /* Step 1: Get a new Env from env_free_list*/
    e = LIST_FIRST(&env_free_list);// 从env_free_list拿到一个空闲的进程控制块
    if (e == NULL) return -E_NO_FREE_ENV;

    /* Step 2: Call a certain function (has been completed just now) to init
kernel memory layout for this new Env.
     *The function mainly maps the kernel address to this new Env address. */
    // 用env_setup_vm(e)为申请一页物理内存，当作e对应进程的页目录
    if ((r = env_setup_vm(e)) != 0) {
        return r;
    }

    /* Step 3: Initialize every field of new Env with appropriate values.*/
    e->env_id = mkenvid(e);
    e->env_parent_id = parent_id;
    e->env_status = ENV_RUNNABLE;    // 将新申请的进程状态设置为ENV_RUNNABLE
    e->env_runs = 0;                 // 新申请的进程已经运行的次数为0

    /* Step 4: Focus on initializing the sp register and cp0_status of env_tf
field, located at this new Env. */
    // e->env_tf.cp0_status = 0x10001004;
    e->env_tf.cp0_status = 0x1000100c;// 今年新改的，是仿真器 gxemul 的实现与 IDT
R30xx 手册存在的差异
    e->env_tf.regs[29] = USTACKTOP; // 29号指针是 sp 栈指针
    // 这里是设置该进程的栈寄存器， USRACKTOP 是用户栈顶
    /* Step 5: Remove the new Env from env_free_list. */
    LIST_REMOVE(e, env_link);        // 已经占用了进程控制块，就要从链表中移除

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

加载二进制镜像

下面的两个函数将为**新进程的程序**分配空间来容纳程序代码。

load_icode_mapper()

先来看看参数

```
static int load_icode_mapper(u_long va, u_int32_t sgsize,
                             u_char *bin, u_int32_t bin_size, void *user_data)
```

这里要透彻地理解，进程的二进制文件要加载到的“虚拟地址va”不等于“内核虚拟地址”

一句话解读：**va是进程自己视野下的虚拟地址。**

进程的页目录是映射了整个4G虚拟空间，那么这个va就是在这个进程页目录的视野下的虚拟地址。也是0-4G，只是和内核虚拟地址一点必然的关系也没有。

对应到这个函数中，也就是说，va只是用于在 **进程页目录所映射的二级页表体系中，寻找到对应的二级页表项，而二进制文件被加载到的真正的物理地址（以及该物理地址对应的内核虚拟地址）就是被存在了这里。**这就是映射的真正含义。由此，我们可以总结出这个函数的真正作用：

函数作用

将开头地址为bin，大小为bin_size的二进制文件（程序代码），加载到内核虚拟地址的合适位置处。并且将这个内核虚拟地址所对应的物理地址，存放在 userdata 所代表的**进程** 的 **页目录映射的二级页表体系 下 va 地址对应的二级页表项中。**当然，这个加载就是复制，而且需要一页一页地分配内核中的虚拟页内存用于复制，对应着硬件中的物理内存用于复制。

也可以抽象地说：**将进程的二进制文件加载到进程的页目录视野下的va处。**

这里需要复制一段我在lab2中对于page_alloc()函数的解读，来理解加载到的内核虚拟地址的位置。

> **由于mips_vm_init()函数是在page_init()函数之前，在mips_vm_init()早已经分配好了物理内存控制块和进程控制块的虚拟内存地址。而在page_init()函数中建立起了freemem和page_free_list的对应关系，所以在page_free_list中插入的物理内存控制块所对应的物理地址转换为的虚拟地址一定在执行完mips_vm_init()函数后的freemem（具体来说，是0x8046 d000）和0x8400 0000-1之间。所以page_alloc()分配出的虚拟地址不会干扰物理内存控制块本身和进程控制块本身。并且始终在[0x8046 d000, 0x8400 0000-1]之间。**
>
> ```
> for (i = unavailble_npage; i < npage; i++) {
>  pages[i].pp_ref = 0;
>  LIST_INSERT_HEAD(&page_free_list, &pages[i], pp_link);
> }
> ```

> **包括在后面的进程相关中，也有如下的应用：**

> **每申请一个进程，都要为它调用一次page_alloc()函数，分配一页内存。这里的内存就是虚拟内存，只不过切切实实对应着一页物理内存。程序中在kseg0分配一页虚拟内存，对应到硬件层面就是分配一页物理内存。**

> **需要为新进程的程序分配空间来容纳程序代码。当需要加载一个ELF文件时，也是把binary文件中各部分的segment利用bcopy()函数，拷贝到page_alloc()函数分配的若干页内核虚拟内存中。这些被加载的程序代码二进制文件，自然也是在[0x8046 9000, 0x8400 0000-1]之间。**

已有页面的检查

这里要注意，当前一个段已经加载完毕时，可能后一个段是要加载到前一个段末尾的那一页。此时不能新申请一页，更不能调用page_insert函数，否则就会覆盖掉前一段末尾的映射关系。像下面的这种代码就是错的，虽然不影响后续实验。

▶ 错误代码

以及只需要在开头的offset不为0的时候，进行必要的检查，这是因为一个进程只会对应一个elf文件，而一个elf文件中，各个段之间都是按照低地址到高地址的顺序排列的，所以不需要进行尾部检查。

验证资料链接：https://man7.org/linux/man-pages/man5/elf.5.html

关键信息截图：

members values are undefined. This lets the program header have ignored entries.

PT_LOAD
The array element specifies a loadable segment, described by *p_filesz* and *p_memsz*. The bytes from the file are mapped to the beginning of the memory segment. If the segment's memory size *p_memsz* is larger than the file size *p_filesz*, the "extra" bytes are defined to hold the value 0 and to follow the segment's initialized area. The file size may not be larger than the memory size. Loadable segment entries in the program header table appear in ascending order, sorted on the *p_vaddr* member.

函数主体具体语句解读

必须指出的一点是，加载二进制镜像的时候的bin一定是字对齐的，不然mips下的编译器中bcopy会出问题。这点在lab4中新增的文件 user/user.lds中有所体现。

这里吐槽一下，内核态的bcopy和bzero都是要求起始地址和目标地址字对齐的，不然会出错。用户态的user_bcopy和user_bzero的写法解决了这个问题。

```
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;// user_data就是说明：当前要加载到虚拟内存中的二进制文件（程序代码）是哪个进程的
    struct Page *p = NULL;
    u_long i;
    int r;
    u_long offset = va - ROUNDDOWN(va, BY2PG); // 如果 va 没有和BY2PG对齐，则势必会产生不为 0 的offset

    /* Step 1: load all content of bin into memory. */
    r = 0;
    if (offset > 0){
        p = page_lookup(env->env_pgdir, va, NULL);  // 先确认起始地址va所在的页（进程空间的页）是不是已经有了对应的物理内存控制块（即已经申请过对应的物理页面）
        // 具体来说，因为我们前面分析过，进程的页目录每一个页表项存放一个二级页表的物理地址，而内核空间中为这些二级页表也分配了对应的虚拟空间。在内核虚拟空间中的二级页表的每一个二级页表项管理进程空间视角下的4G内存中的一页，那么va作为进程空间视角下的地址，它所在的一页进程虚拟空间对应的实际物理地址，就应该存放内核虚拟空间中对应二级页表的二级页表项中。
        if (p == 0){// 说明并不存在上面说的这种情况
            // 那就需要为 未能整页对齐的开始部分新申请一页物理内存
            if ( (r = page_alloc(&p))<0 ){
                return r;
            }
             page_insert(env->env_pgdir, p, va, PTE_R);
            // 在内核中虚拟空间复制完毕后，就需要建立起虚拟空间和物理空间的映射关系
            //将一级页表基地址 pgdir 对应的两级页表结构中 va 这一虚拟地址 所对应的二级页表项中
            //填入p这个物理内存控制块对应的物理页面的页号，并设置页表权限为可写
            //还是要注意，进程的一级页表和二级页表都在内核虚拟空间中有对应的内存分配
        }
        /* 如果说要加载到的位置前面有前一个段申请好的页面，就不需要申请页和建立映射 */
        r = MIN(BY2PG - offset, bin_size); // 这里需要比较 va占用的第一页的 整页 减去offset后的大小 和整个 ELF 文件 大小，防止 ELF 只能占据不到一页
        bcopy((void *)bin, (void *)(page2kva(p)+offset), r);// 将开头部分复制到申请的物理控制块所映射的 内核虚拟空间 。因为page_free_list本身就是从struct Page们和struct Env们本身所占据的空间对应的物虚拟内存后开始划分的，所以只会在[0x8046 9000, 0x8400 0000-1]之间容纳这些文件
    }
```

```
    // 这里的 r 考虑到了两种情况：offset为 0时，r就是0；offset不为 0时，r是 MIN(BY2PG - offset, bin_size)。总之是让开始遍历的地址页对齐。
    for (i = r; i < bin_size; i += BY2PG) {
        if ((r = page_alloc(&p)) < 0) {// 经过上面的调整，i=r以后，i就已经整页对齐了
            return r;
        }
        bcopy((void *)(bin + i), (void *)page2kva(p), MIN(BY2PG, bin_size - i));// 这里要时刻留意二进制文件剩下的部分够不够一页
        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! */
    // 注意 .bss是不需要复制到内存中的 所以，即使这里有可能 ELF 文件在最后一页的残余部分 加上 .bss部分也不能超过最后一页，此时 i >= sgsize，但是因为不需要复制
    // 并且要求文件占不够的直接置零就好了，所以这样跳过了下面的代码，也是没问题的。(在申请页面时已经置零了)
    // 而如果 ELF 文件在最后一页的残余部分 加上 .bss部分 后超过了最后一页，那么 i < sgsize，执行代码
    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;
}
```

load_elf()

将开头地址为binary，大小为size的二进制文件（程序代码），加载到userdata所代表的进程专属的页目录（映射的二级页表体系）中。二进制文件中，每一个segment应该被加载到的虚拟地址va，实际上是对应到了userdata代表的进程的页目录（映射的二级页表体系）中。并将二进制文件的入口点的虚拟地址存入*entry_point。

> 复习：ProgramHeader表的每一个表项记录着一个segment的信息
>
> ```
> /* Program segment header.  */
> typedef struct {
>     Elf32_Word      p_type;                 /* Segment type */
>     Elf32_Off       p_offset;               /* Segment file offset */
>     Elf32_Addr      p_vaddr;                /* Segment virtual address */
>     Elf32_Addr      p_paddr;                /* Segment physical address */
>     Elf32_Word      p_filesz;               /* Segment size in file */
>     Elf32_Word      p_memsz;                /* Segment size in memory */
>     Elf32_Word      p_flags;                /* Segment flags */
>     Elf32_Word      p_align;                /* Segment alignment */
> } Elf32_Phdr;
> ```

```
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))
            // 这里的函数指针是load_icode_mapper函数。
{
    Elf32_Ehdr *ehdr = (Elf32_Ehdr *)binary;
    Elf32_Phdr *phdr = NULL;
    /* As a loader, we just care about segment,
         * so we just parse program headers.
         */
    u_char *ptr_ph_table = NULL;
        Elf32_Half ph_entry_count;
```

```
        Elf32_Half ph_entry_size;
        int r;

    // check whether `binary` is a ELF file.
    if (size < 4 || !is_elf_format(binary)) {
                return -1;
        }

        ptr_ph_table = binary + ehdr->e_phoff;  // 二进制文件基地址+ProgramHeader表
在elf文件中的偏移量
        ph_entry_count = ehdr->e_phnum;         // ProgramHeader表的表项数目
        ph_entry_size = ehdr->e_phentsize;      // ProgramHeader表的每一个表项的大
小
        // ProgramHeader表的每一个表项记录着一个segment的信息
        while (ph_entry_count--) {
                phdr = (Elf32_Phdr *)ptr_ph_table;

                if (phdr->p_type == PT_LOAD) { // PT_LOAD 就是 1
        /* Real map all section at correct virtual address.Return < 0 if error.
*/
        /* Hint: Call the callback function you have achieved before. */
                    r = 0;
                    r = map(phdr->p_vaddr, phdr->p_memsz,
                            binary + (phdr->p_offset), phdr->p_filesz,
user_data);
                    // 利用load_icode_mapper将每一个segment加载到正确的虚拟地址
                    // phdr->p_vaddr 记录了segment应该被加载到的虚拟地址
                    // phdr->p_memsz 记录了segment在内存中的大小
                    // binary + (phdr->p_offset) 计算出了该segment项的开头地址
                    // phdr->p_filesz 记录了segment的文件大小
                    // user_data是load_elf传入的参数，是说明：当前要加载到虚拟内存中的二
进制文件（程序代码）是哪个进程的
                    if (r<0){
                        return r;
                    }
                }
                // Program Header Table基地址加上每一个Elf32_Phdr 的大小，即以表项为单
位，一项项地往后遍历
                ptr_ph_table += ph_entry_size;
        }
        *entry_point = ehdr->e_entry;// e_entry是二进制文件入口点的虚拟地址。
        return 0;
}
```

load_icode()

将开头地址为binary，大小为size的二进制文件（程序代码），加载到目标进程struct Env *e的页目录映射的二级页表体系中。（抽象来说就是将elf加载到进程空间中）

值得注意的是，加载到的进程空间的地址是 `USTACKTOP - BY2PG`， 而 `USTACKTOP` 就是 `UTOP`，也是 `UENVS = 0x7f40 0000`

```
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. */
    r = page_alloc(&p);
    if (r < 0) {
        return;
    }

    /* Step 2: Use appropriate perm to set initial stack for new Env. */
    /* Hint: Should the user-stack be writable? */
    perm = PTE_R;
    r = page_insert(e->env_pgdir, p, USTACKTOP - BY2PG, perm);
    // 在进程e的二级页表体系中 虚拟地址为USTACKTOP - BY2PG的二级页表项中 写入p对应的物理页号，并设置权限为可写
    // 这里再次强调，page_insert这里的va只是页目录视角下的4G中的地址，并不是内核4G中的地址
    if (r < 0) {
        return;
    }

    /* Step 3: load the binary using elf loader. */
    // 将开头地址为binary，大小为size的二进制文件（程序代码），加载到进程e的页目录映射的二级页表体系
    // 并将加载完毕的二进制文件入口虚拟地址存入entry_point
    r = load_elf(binary, size, &entry_point, (void *)e, load_icode_mapper);// 这里的(void*)e就是 userdata
    if (r < 0) {
        return;
    }

    /* Step 4: Set CPU's PC register as appropriate value. */
    e->env_tf.pc = entry_point;// 将加载完毕的二进制文件入口虚拟地址存入进程e的PC寄存器信息处
}
```

这里的e->env_tf.pc是什么呢？这个字段指示了进程要恢复运行时 pc 应恢复到的位置。冯诺依曼体系结构的一大特点就是：程序预存储，计算机自动执行。我们要运行的进程的代码段预先被载入到了 entry_ point 为起点的内存中，当我们运行进程时，CPU 将自动从 pc 所指的位置开始执行二进制码。

创建进程

创建进程的过程很简单，就是实现对上述函数的封装，具体步骤是：

分配一个新的Env 结构体，设置进程控制块，并将二进制代码载入到对应地址空间即可完成。

env_create_priority()

分配一个新的Env 结构体e，设置进程控制块的优先级为priority，然后将开头地址为binary，大小为size的二进制文件（程序代码），加载到内存的对应地址空间。最后将进程e插入env_sched_list[0]的头部。

```
void env_create_priority(u_char *binary, int size, int priority)
{
    struct Env *e;
    /* Step 1: Use env_alloc to alloc a new env. */
    if (env_alloc(&e, 0) < 0) {
        return;
    }
    /* Step 2: assign priority to the new env. */
    e->env_pri = priority;
    /* Step 3: Use load_icode() to load the named elf binary,
```

```
        and insert it into env_sched_list using LIST_INSERT_HEAD. */
    load_icode(e, binary, size);
    LIST_INSERT_HEAD(&env_sched_list[0], e, env_sched_link);
}
```

env_create()

调用优先级为1时的env_create_priority函数

```
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);
}
```

比起上面两个简单的函数，真正完成进程创建的其他工作的其实是两个封装好的宏命令

ENV_CREATE 和 ENV_CREATE_PRIORITY

```
#define ENV_CREATE_PRIORITY(x, y) \
{\
    extern u_char binary_##x##_start[]; \
    extern u_int binary_##x##_size;\
    env_create_priority(binary_##x##_start, \
            (u_int)binary_##x##_size, y);\
}
#define ENV_CREATE(x) \
{ \
    extern u_char binary_##x##_start[];\
    extern u_int binary_##x##_size; \
    env_create(binary_##x##_start, \
        (u_int)binary_##x##_size); \
}
```

这个宏涉及到的语法是：

##代表拼接，例如下面这段代码

```
#define CONS(a,b) int(a##e##b)
int main()
{
    printf("%d\n", CONS(2,3)); // 2e3 输出:2000
    return 0;
}
```

> 具体讲解参考博客：C语言宏中"#"和"##"的用法 - Leo Chin - 博客园 (cnblogs.com)

那么，我们在lab3实验中，需要在 `init/init.c` 中增加下面两句代码，来初始化创建两个进程。

```
ENV_CREATE_PRIORITY(user_A, 2);
ENV_CREATE_PRIORITY(user_B, 1);
```

这里的 `user_A` 和 `user_B` 用于变量命名，以 `user_A` 为例，经过 `ENV_CREATE` 宏的拼接后，得到 `binary_user_A_start` 数组和 `binary_user_A_size` 变量。我们可以在 `init/code_a.c` 文件中可以找到它们的定义。

```
unsigned char binary_user_A_start[] = {...} // 这里不表，有很多以二进制文件形式写成的内容
unsigned int binary_user_A_size = 5298;
```

包括在后来，我们每一个在本地编写的、用于测试后续lab我们完成的功能的进程，其实都需要在 `init/init.c` 中添加对应的创建进程的语句。只不过，那时候的进程不需要像lab3这样以很多个16进制数组成 `unsigned char` 数组的形式来模拟二进制文件，这样的工作其实就是编译、链接后，转为机器码，是每一个c文件在执行时的必经之路。

```
ENV_CREATE(user_fktest);
ENV_CREATE(user_testpipe);
ENV_CREATE(fs_serv);
...
```

进程运行与切换

在env.c中我们会发现下面的两行函数声明

```
extern void env_pop_tf(struct Trapframe *tf, int id);
extern void lcontext(u_int contxt);
```

在env_asm.S中我们可以看到这两个函数的实现，这里他们都是为了env_run()函数服务

env_run()

复习：

```
#define CP0_CONTEXT $4      // $a0 这里就是context
```

env_run，是进程运行使用的基本函数，它包括两部分：

• 保存当前进程上下文**(如果当前没有运行的进程就跳过这一步)**

• 恢复要启动的进程的上下文，然后运行该进程。

进程上下文说来就是一个环境，相对于进程而言，就是进程执行时的环境。具体来说就是各个变量和数据，包括所有的寄存器变量、内存信息等。

其实我们这里运行一个新进程往往意味着是进程切换，而不是单纯的进程运行。进程切换，就是当前进程停下工作，让出CPU 处理器来运行另外的进程。那么要理解进程切换，我们就要知道进程切换时系统需要做些什么。

进程切换的时候，为了保证 **下一次进入这个进程的时候我们不会再“从头来过”，而是有记忆地从离开的地方继续往后走** ，我们要保存一些信息，那么，需要保存什么信息呢？理所当然地想想，你可能会想到下面两种需要保存的信息：

进程本身的信息 和 **进程周围的环境信息**

事实上，进程本身的信息无非就是进程控制块中那些字段，包括

```
env_id,env_parent_id,env_pgdir,env_cr3...
```

这些在进程切换后还保留在原本的进程控制块中，并不会改变，因此不需要保存。而会变的实际上是进程周围的环境信息，这才是需要保存的内容。也就是 env_tf 中的进程上下文。

那么你可能会想，进程运行到某个时刻，它的上下文——所谓的 CPU 的寄存器在哪呢？我们又该如何保存？在lab3 中，**我们在本实验里的寄存器状态保存的地方是TIMESTACK区域。**

```
struct Trapframe *old;
old = (struct Trapframe *)(TIMESTACK - sizeof(struct Trapframe));
```

这个 old 就是当前进程的上下文所存放的区域。第一步注释还说到，让我们参考env_destroy ，其实就是把 old 区域的东西拷贝到当前进程的 env_tf 中，以达到保存进程上下文的效果。

还有一点很关键，**我们需要将env_tf.pc设置为env_tf.cp0_epc。**

> epc是MIPS中，cp0协处理器的第14号寄存器的名字。它用于：当中断异常发生时, 该寄存器记录当前的PC寄存器值。这里，env_run(struct Env *e)函数为了运行传入的参数进程e，在当前运行的进程不为空的时候，是需要先中断当前运行的程序，再把代表当前运行进程的指针curenv指向参数进程e。
>
> 因此，需要把被中断的当前进程curenv的cp0_epc寄存器里存放在它的pc里。以便于中断结束以后回来继续执行。

总结以上说明，我们不难看出 env_run 的执行流程：

1. 保存当前进程的上下文信息，设置当前进程上下文中的 pc 为epc。
2. 切换 curenv 为即将运行的进程。
3. 调用 lcontext 函数，设置全局变量mCONTEXT为当前进程页目录地址，这个值将在TLB重填时用到。
4. 调用 env_pop_tf 函数，恢复现场、异常返回。

这里用到的 env_pop_tf 是定义在 lib/env_asm.S 中的一个汇编函数。这个函数也呼应了我们前文提到的，进程每次被调度运行前一定会执行的 rfe汇编指令。

那么整个env_run函数的逻辑就是

Step1 保存当前进程的上下文信息

如果当前存在正在运行的进程（`curenv != NULL`），则需要将`TIMESTACK-sizeof(struct Trapframe)`以上的一整个`struct Trapframe`复制到正在运行的程序curenv自己的上下文env_tf（也是一个`struct Trapframe`）。并且要把正在运行的进程的env_tf中的pc值设置为cp0_epc。因为TIMESTACK里面的pc值始终都是0。

这里抽象的理解来说就是：当前的进程被打断时，自己本身的信息不需要额外保存（一直揣在身上），而为了中断结束以后还能“断点续传”而不是“重新开始”，需要额外记录此时的上下文环境，如CPU寄存器信息等。为了做到这一点，就需要从TIMESTACK为栈顶，压栈大小为一个`struct Trapframe`的区域处，将这些信息复制到当前被打断进程的 env_tf 中。现在，“时间冻结”的条件准备就绪，就剩下“何处解冻”的钥匙了，那么将当前进程上下文中的 pc 设置为 epc，这个“解冻”的地址就准备好了。

Step2 切换 curenv 为即将运行的进程

切换 curenv 为即将运行的进程e（`curenv = e`）

MOS 中切换地址空间, 不仅要让 `mCONTEXT` 改为进程的页目录基地址, 还要将进程的 ASID 存入 EntryHi。这两步分别由lcontext和env_pop_tf完成。

Step3 调用 lcontext 函数，设置全局变量mCONTEXT为当前进程页目录地址

调用 lcontext 函数，设置全局变量mCONTEXT为当前进程页目录地址，这个值将在TLB重填时用到。

传入参数的时候传入的是(u_int)(e->env_pgdir)，这个值进入a0寄存器，由`sw a0,mCONTEXT`送入mCONTEXT。

Step4 调用 env_pop_tf 函数，将即将运行的进程e的现场恢复到CPU、异常结束并返回用户态

调用 env_pop_tf 函数，恢复现场、异常处理结束，返回用户态。

参数：env_pop_tf 函数将&(e->env_tf)和GET_ENV_ASID(e->env_id)分别传入$a0, $a1寄存器。

将即将运行的进程e的上下文环境的头地址存入k0，将 ASID填入EntryHi中，将CPU的SR寄存器的低2位置零，然后利用k0和 TF_XX 宏将即将运行的进程 e 的 struct Trapframe 中的对应信息存入CPU的各个寄存器。

首先是LO、HI、EPC，然后是1-31的通用寄存器，接着将 struct Trapframe 中记录PC的值存入k1，将 struct Trapframe 中记录SR的值存入CPU的SR寄存器。然后利用 `j k1` 指令跳转到k1存放的地址，也就是即将运行的进程e的PC，开始运行进程。最后执行rfe使得 SR 寄存器低6位的二重栈出栈。

env_run()完整逻辑如下

```
void env_run(struct Env *e)
{
    /* Step 1: save register state of curenv. */
    /* Hint: if there is an environment running,
     *   you should switch the context and save the registers.
     *   You can imitate env_destroy() 's behaviors.*/
    if (curenv != NULL) { // 当前运行进程不为空，保存当前进程上下文
        struct Trapframe *old = (struct Trapframe*)( TIMESTACK-sizeof(struct
Trapframe) );
        bcopy((void *) old, (void *)(&(curenv->env_tf)), sizeof(struct
Trapframe));
        // 在发生进程调度，或当陷入内核时，会将当时的进程上下文环境保存在env_tf变量中。
        curenv->env_tf.pc = curenv->env_tf.cp0_epc;
    }

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

    /* Step 3: 设置全局变量mCONTEXT为当前进程页目录地址，这个值将在TLB重填时用到 */
    lcontext((u_int)(e->env_pgdir)); // 联系Bonus去理解

    /* Step 4: Use env_pop_tf() to restore the environment's
     *   environment   registers and return to user mode.
     *
     * Hint: You should use GET_ENV_ASID there. Think why?
     *   (read <see mips run linux>, page 135-144)
     */
    env_pop_tf(&(e->env_tf), GET_ENV_ASID(e->env_id));
}
```

进程释放与销毁

env_free()

将一个进程彻底释放。包括将它的页目录对应的整个二级页表体系涉及到的物理页面取消映射关系，释放页目录本身，释放该进程对应的ASID，设置该进程状态为ENV_FREE，并把进程控制块插入env_free_list，将进程控制块从它所在两个env_sched_list之一中移除。

本函数的二级页表体系的遍历也值得学习。

```
void env_free(struct Env *e)
```

```
{
    Pte *pt;
    u_int pdeno, pteno, pa;

    /* Hint: Note the environment's demise.*/
    printf("[%08x] free env %08x\n", curenv ? curenv->env_id : 0, e->env_id);

    /* Hint: Flush all mapped pages in the user portion of the address space */
    for (pdeno = 0; pdeno < PDX(UTOP); pdeno++) {
        /* Hint: only look at mapped page tables. */
        if (!(e->env_pgdir[pdeno] & PTE_V)) {
            continue;
        }
        /* Hint: find the pa and va of the page table. */
        pa = PTE_ADDR(e->env_pgdir[pdeno]);
        pt = (Pte *)KADDR(pa);
        /* Hint: Unmap all PTEs in this page table. */
        for (pteno = 0; pteno <= PTX(~0); pteno++)
            if (pt[pteno] & PTE_V) {
                page_remove(e->env_pgdir, (pdeno << PDSHIFT) | (pteno <<
PGSHIFT));
            }// 解除进程页目录e->env_pgdir 中虚拟地址为(pdeno << PDSHIFT) | (pteno
<< PGSHIFT)的到它原来对应的物理页面的映射关系
        /* Hint: free the page table itself. */
        e->env_pgdir[pdeno] = 0;
        page_decref(pa2page(pa)); // 减少一次一级页表项对应的物理页面的引用次数，在次数为
0的时候将其释放，插入page_free_list。
    }
    /* Hint: free the page directory. */
    pa = e->env_cr3;
    e->env_pgdir = 0;
    e->env_cr3 = 0;
    /* Hint: free the ASID */
    asid_free(e->env_id >> (1 + LOG2NENV));
    page_decref(pa2page(pa));
    /* Hint: return the environment to the free list. */
    e->env_status = ENV_FREE;
    LIST_INSERT_HEAD(&env_free_list, e, env_link);
    LIST_REMOVE(e, env_sched_link);
}
```

env_destory() 对env_free()函数的补全

```
/* Overview:
 *  Free env e, and schedule to run a new env if e is the current env.
 */
void env_destroy(struct Env *e)
{
    /* Hint: free e. */
    env_free(e);

    /* Hint: schedule to run a new environment. */
    if (curenv == e) {
        curenv = NULL;
        /* Hint: Why this? */
        // 这里是因为，env_run()是直接从TIMESTACK-sizeof(struct Trapframe)处读取要被
切换的进程的环境，并将其存放在进程自己的结构体里的，因此理论上讲，只要是进程切换，就应该执行这行
代码
        // 但是这里不一样，这里这个进程已经被销毁了，所以这个转移没有意义
        bcopy((void *)KERNEL_SP - sizeof(struct Trapframe),
              (void *)TIMESTACK - sizeof(struct Trapframe),
              sizeof(struct Trapframe));
        printf("i am killed ... \n");
```

```
        // 如果被销毁的是当前进程，那么就需要调度一个新进程来执行。
        sched_yield();
    }
}
```

两个汇编函数解析

lcontext

这个函数在env_run()函数中有如下的调用

```
lcontext((u_int)(e->env_pgdir));
```

就是把进程的页目录地址存入 mCONTEXT

```
LEAF(lcontext)
        .extern mCONTEXT        # 声明外部变量 mCONTEXT
        # 将参数(u_int)(e->env_pgdir)（即进程的页目录虚拟地址）存入mCONTEXT
        sw      a0, mCONTEXT
        jr      ra
        nop
END(lcontext)
```

env_pop_tf

env_pop_tf函数——弹出进程的环境到CPU中。

将ASID填入EntryHi中，以及将参数中的 `env_tf` 填充到CPU状态中。注意：在env_pop_tf结束以后，已经回到了用户态。此时用户进程正式开始运行，env_pop_tf已经将系统跳到了进程开始运行的地方。（类似异常发生的时候系统跳转到了 `0x8000080` ）

> MOS 中切换地址空间, 不仅要让 `mCONTEXT` 改为进程的页目录基地址, 还要将进程的 ASID 存入 EntryHi.
>
> - `mCONTEXT` 负责指导 TLB 重填例程 `do_refill` (软件)
> - EntryHI 的 ASID 则负责指导 TLB 的虚实地址转换 (硬件)
>
> > 用户空间虚地址访存, 需要将虚地址与 EntryHI 中的 ASID 同时作为关键字, 在 TLB 中查找

首先要注意：一系列TF打头的都是宏定义的4的倍数的数字，对应着前面提到的**struct Trapframe 的结构**。

```
struct Trapframe { //lr:need to be modified(reference to linux pt_regs) TODO
    /* Saved main processor registers. */
    unsigned long regs[32];     //
    /* Saved special registers. */
    unsigned long cp0_status;   // 记录cp0寄存器的状态
    unsigned long hi;           // 记录EntryHi
    unsigned long lo;           // 记录EntryLo
    unsigned long cp0_badvaddr; // 记录cp0 Badvaddr
    unsigned long cp0_cause;    // 记录cp0 的cause寄存器
    unsigned long cp0_epc;      // 记录中断发生时的pc值
    unsigned long pc;           // 记录pc值
};
```

因为struct Trapframe 中所有的记录寄存器的信息的成员都是unsigned int类型，都是4个字节，所以在宏定义中，TF_打头的都是以4为差距进行累加。

这里是具体的宏定义。

▶ 具体宏定义

可以清晰地看出，这些宏和struct Trapframe 的结构是一一对应的。只要拿到结构体的首地址，就可以通过这些宏记录的对应寄存器的偏移量，迅速访问到对应的寄存器。

在汇编函数中，如果让$k0寄存器存放 struct Trapframe 的基地址，那么上面**宏定义的记录各个寄存器的结构体成员可以通过的 `TF_XX(k0)` 方式来迅速获取**。这一点将在env_pop_tf函数中频繁用到。

CP0_XX形式的宏则是寄存器的名称，如 `#define CP0_STATUS $12`

下面是env_pop_tf这个汇编函数的整体逻辑：

每个进程在每一次被调度时都会执行 `env_run()` 函数， `env_run()` 函数会调用 `env_pop_tf` 这个汇编函数，并将 `&(e->env_tf)` 和 `GET_ENV_ASID(e->env_id)` 分别传入$a0, $a1寄存器

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

将即将运行的进程e的上下文环境的头地址存入k0，将 ASID填入EntryHi中，将CPU的SR寄存器的低2位置零，然后利用k0和TF_XX宏将即将运行的进程e的struct Trapframe中的对应信息存入CPU的各个寄存器。

首先是LO、HI、EPC，然后是1-31的通用寄存器，接着将 struct Trapframe 中记录PC的值存入k1，将 struct Trapframe 中记录SR的值存入CPU的SR寄存器。然后利用 `j  k1` 指令跳转到k1存放的地址，也就是即将运行的进程e的PC，开始运行进程。最后执行rfe使得 SR寄存器低6位的二重栈出栈。

env_tf.pc目前出现了两次，一次是在env_run()函数中，当前函数被中断，env_tf.pc被设置为env_tf.cp0_epc。

```
// 在发生进程调度，或当陷入内核时，会将当时的进程上下文环境保存在env_tf变量中。
curenv->env_tf.pc = curenv->env_tf.cp0_epc;
```

另一次是在load_icode中，env_tf.pc被设置为二进制文件的入口虚拟地址。

```
/* Step 4: Set CPU's PC register as appropriate value. */
e->env_tf.pc = entry_point;// 二进制文件的内核虚拟地址
```

```
LEAF(env_pop_tf)
.set    mips1
        //1:    j   1b
    nop
        move    k0, a0              # 将第一个参数&(e->env_tf)（即进程上下文环境的头地址）存入k0
        mtc0    a1, CP0_ENTRYHI     # 将第一个参数GET_ENV_ASID(e->env_id)存入CP0_ENTRYHI
                                    # 也就是cp0的EntryHi寄存器
# 这里GET_ENV_ASID(e->env_id)是取出envid的ASID部分后再右移6位，正好对应了entryHi中低6位是NULL
// 这里的四行代码其实感觉更像是冗余的代码，因为只要异常发生，系统就会自动进入用户态，那么SR寄存器的后两位就会自动置零（gxemul 中是KU为0是内核态，IE为0 禁用中断）
        mfc0    t0, CP0_STATUS      # 将cp0的SR寄存器的值存入t0
        ori     t0, 0x3             # 将t0的值低2位置1
        xori    t0, 0x3             # 将t0的值低2位置0     则此时cpu是内核态（因为gxemul 中是KU为0是内核态），且禁用一切中断
        mtc0    t0, CP0_STATUS      # 将t0的值存入cp0的SR寄存器
// 疑似冗余代码结束
        lw  v1,TF_LO(k0)            # 将 struct Trapframe 中记录EntryLo的值存入v1
        mtlo    v1                  # 将v1的值存入EntryLo寄存器
        lw  v0,TF_HI(k0)            # 将 struct Trapframe 中记录EntryHi的值存入v0
        lw  v1,TF_EPC(k0)           # 将 struct Trapframe 中记录EPC的值存入v1
        mthi    v0                  # 将v0的值存入EntryHi寄存器
        mtc0    v1,CP0_EPC          # 将v1的值存入EPC寄存器
        # 以下就是直接把 struct Trapframe 中记录31个通用寄存器的值放进cpu对应的通用寄存器
```

```
        lw  $31,TF_REG31(k0)
        lw  $30,TF_REG30(k0)
        lw  $29,TF_REG29(k0)
        lw  $28,TF_REG28(k0)
        lw  $25,TF_REG25(k0)
        lw  $24,TF_REG24(k0)
        lw  $23,TF_REG23(k0)
        lw  $22,TF_REG22(k0)
        lw  $21,TF_REG21(k0)
        lw  $20,TF_REG20(k0)
        lw  $19,TF_REG19(k0)
        lw  $18,TF_REG18(k0)
        lw  $17,TF_REG17(k0)
        lw  $16,TF_REG16(k0)
        lw  $15,TF_REG15(k0)
        lw  $14,TF_REG14(k0)
        lw  $13,TF_REG13(k0)
        lw  $12,TF_REG12(k0)
        lw  $11,TF_REG11(k0)
        lw  $10,TF_REG10(k0)
        lw  $9,TF_REG9(k0)
        lw  $8,TF_REG8(k0)
        lw  $7,TF_REG7(k0)
        lw  $6,TF_REG6(k0)
        lw  $5,TF_REG5(k0)
        lw  $4,TF_REG4(k0)
        lw  $3,TF_REG3(k0)
        lw  $2,TF_REG2(k0)
        lw  $1,TF_REG1(k0)

        lw  k1,TF_PC(k0)        # 将 struct Trapframe 中记录PC的值存入k1

        lw  k0,TF_STATUS(k0)    # 将 struct Trapframe 中记录SR的值存入k0
        nop
        mtc0    k0,CP0_STATUS   # 将k0的值存入cp0的SR寄存器，
                                # 也就是env_alloc()中设置的0x10001004
        j   k1                  # 跳转到k1存放的地址，即进程的PC，开始运行进程
        rfe                     # SR寄存器低6位的二重栈出栈
        # 0x 0001 0000 0000 0000 0001 0000 0000 0100  低6位出栈后，低6位便是00 0001
# lab4  # 0x 0001 0000 0000 0000 0001 0000 0000 1100  低6位出栈后，低6位便是00 0011
        # 说明回到了用户态，且允许响应中断
        nop
END(env_pop_tf)
```

此外，关于SR寄存器中的四行代码，其实感觉更像是冗余的代码，因为

只要异常发生，系统就会自动进入用户态，那么SR寄存器的后两位就会自动置零（gxemul 中是KU为0是内核态，IE为0 禁用中断），并不需要手动置零。

其实也很好理解：我们自己也很难找到一个地方来手动置零。用户态下不一定能 mtc0 ，内核空间不置零也访问不了。

$Part2.2$ 中断与异常

寄存器助记符	CP0寄存器编号	描述
SR	12	状态寄存器，包括中断引脚使能，其他 CPU 模式等位域
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(Status Register)寄存器

下图是 MIPS R3000 中 SR(Status Register)寄存器示意图，也就是我们在env_tf里的cp0_ status。

SR Register

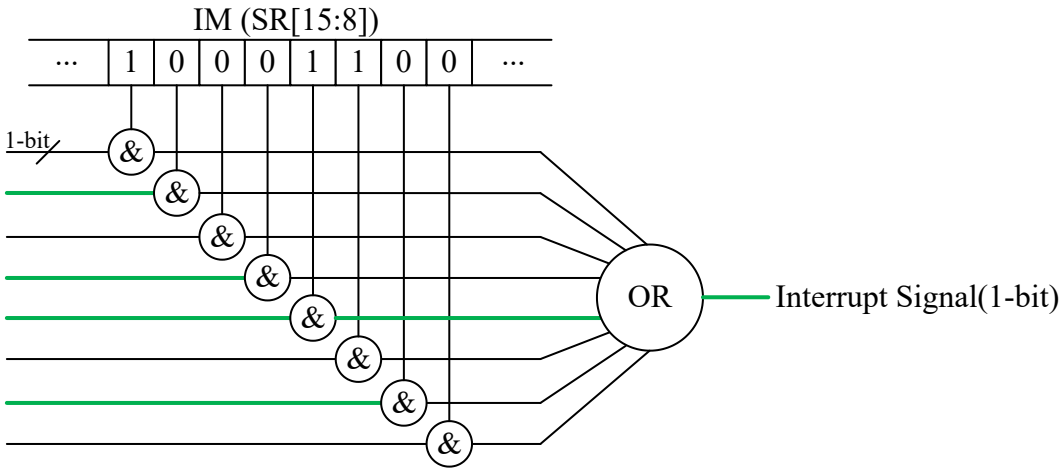
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CU3	CU2	CU1	CU0	0		RE	0		BEV	TS	PE	CM	PZ	SwC	IsC

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

15-8 位为中断屏蔽位，每一位代表一个不同的中断活动，其中 15-10 位使能外部中断源，**9-8 位是 Cause 寄存器软件可写的中断位**。

- `IM` : Interrupt Mask
- `KU` : Kernel-mode or User-mode
- `IE` : Interrupt Enable

`IM` 段共8位, 当第i位为1时, 意味着当第i中断信号置高时, CPU应响应中断



第28bit CU0设置为1，表示允许在用户模式下使用 CP0 寄存器。

第12bit 设置为1，表示 4 号中断可以被响应。

SR 寄存器的低六位：三组KU/IE的二重栈

R3000 的 SR 寄存器的低六位是一个二重栈的结构。下面摘录R3000手册对于二重栈的解释：

KUc、IEc是两个基本的CPU保护位。

当以内核权限运行时，KUc设置为1，用户模式设置为0。在内核模式下，软件可以获取整个程序地址空间，并使用特权（“协处理器0”）指令。用户模式将软件限制为0x0000 0000和0x7FFF FFFF之间的程序地址，并且可以拒绝运行特权指令的权限；试图违反规则会导致异常。
IEc设置为0以禁止CPU发生任何中断，1以启用。

> 简言之，按指导书上的话来说——**最低两位含义如下**：
>
> 1. `KUc` 为1, 意味CPU目前在内核态下运行; `KUc` 为0, 意味CPU目前在用户态下运行.
> 2. `IEc` 为1, 意味着CPU会响应中断(interrupt); `IEc` 为0, 意味着CPU不会响应中断.

但是，gxemul 实现的方式和R3000手册中有所偏差，在我们的实验中，KUc为0才是表示在内核态下。

KUp，IEp “KU previous，IE previous”：

在一个异常(exceptioin)情况下下，硬件采用KUc和IEc的值，并将其保存在此处；同时将KUc、IEc的值更改为[1,0]（内核模式，中断禁用）。rfe指令可用于将KUp、IEp复制回KUc、IEc。

KUo，IEo “KU old，IE old”：

在一个异常(exceptioin)情况下，KUp，IEp位保存在这里。

> Effectively, the six KU/IE bits are operated as a 3-deep, 2-bit wide stack which is pushed on an exception and popped by an rfe.

实际上，**这六个KU/IE位就是以一个“3层深、2位宽”的栈的方式运行，有异常的话就压栈，有ref指令的话就弹出。**这里的栈顶是指KUc，IEc 。

压栈

KUo 和 IEo 是一组，每当异常（MyNote：往往是实现时间片机制的时钟中断）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，当异常发生的时候，硬件会把 KUc 和 IEc 的数值拷贝到这里。

出栈

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
nop
mtc0    k0, CP0_STATUS      # 恢复 CP0_STATUS 寄存器 结束
j       k1
rfe
nop
```

MyNote

这里其实是因为每个进程在每一次被调度时都会执行env_run()函数，env_run()函数会调用env_pop_tf这个汇编函数，并将&(e->env_tf)和GET_ENV_ASID(e->env_id)分别传入$a0, $a1寄存器。具体参见env_pop_tf。

> 现在你可能就懂了为何我们status 后六位是设置为000100了。当运行进程前，运行上述代码到rfe的时候(rfe 处于延迟槽中，因为它前面是一个分支指令)，就会将 KUp 和 IEp 拷贝回 KUc 和 IEc，令status 为 000001，最后两位 KUc，IEc 为 [0,1]，表示CPU目前在用户态下运行，并且开启了中断。之后第一个进程成功运行，这时操作系统也可以正常响应中断。
>
> **MyNote**
>
> env_alloc()函数中，新申请的进程的cp0_status被写入0001 0000 0000 0000 0001 0000 0000 0100
>
> 由于Gxemel的实现和R3000手册标准存在差异，Gxemel中KU为0时表示内核态，
>
> 因此新申请的进程的cp0_status被写入0001 0000 0000 0000 0001 0000 0000 1100
>
> ```
> e->env_tf.cp0_status = 0x10001004;// lab3
> e->env_tf.cp0_status = 0x1000100c;// lab4修改
> ```
>
> 而在env_pop_tf 汇编函数中真正写到了寄存器里。
>
> 而在rfe指令结束后，最后两位 KUc，IEc 为 [1,1]，表示CPU目前**在用户态下运行**，并且**开启了中断**。这也就是为什么要将第28位 CU0 设置为1，表示允许在用户模式下使用 CP0 寄存器。

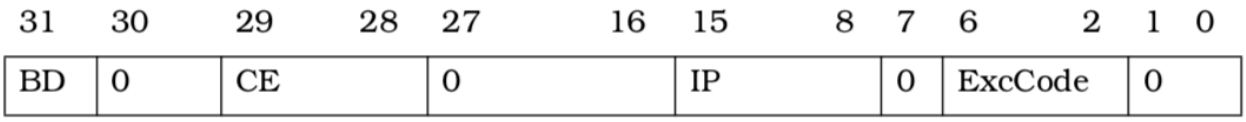
二重栈提供了一个机会，可以从异常处理例程早期发生的异常中干净地恢复，第一个异常尚未保存SR。可以这样做的情况是有限的，并且可能只在允许用户TLB重新填充代码变得更短时才真正有用，如内存管理一章中所述。

Cause(Cause Register)寄存器

下图是 MIPS R3000 中 Cause 寄存器。其中保存着 CPU 中哪一些中断或者异常已经发生。15-8 位保存着哪一些中断发生了，其中 15-10 位来自硬件，9-8 位可以由软件写入，当 SR 寄存器中相同位（即 IM）允许中断（为 1）时，Cause 寄存器这一位活动（为1）就会导致中断。6-2 位（ExcCode），记录发生了什么异常。

这里可以凸显出中断和异常的区别。IP记录的就是中断，ExcCode记录的就是异常。

Cause Register



- `BD` : Branch Delay
- `IP` : Interrupt Pending
- `ExcCode` : Exception Code

Cause(Cause Register)寄存器最高位：BD位和EPC寄存器的关系

这里 `BD` 的引入，是为了处理MIPS流水线延迟槽下发生异常中断的情形。

> 延迟槽：MIPS 流水线中，**跳转指令的下一指令执行完毕后才发生跳转**。 如：
>
> ```
> jal       tag                 # <~~ 1
> addiu     $31, $31, 4         # <~~ 2 delay-slot
> sw        $31, 0($29)
> addiu     $29, $29, -4
> tag:
> lw        $31, 0($29)         # <~~ 3
> addiu     $29, $29, 4         # <~~ 4
> ```
>
> 引入延迟槽，是为了解决流水线的控制冒险，属于 MIPS 流水线的特性。
>
> **考虑执行延迟槽指令时的异常与中断行为**。若 EPC 保存发生异常中断时的指令地址，即延迟槽指令的地址，那么当 CPU 回滚用户态时，将从延迟槽指令开始往后顺序执行，而没有执行跳转行为。**针对这一特殊情况，EPC 会保存延迟槽的上一条指令的地址**，即跳转指令的地址，使得回滚用户态时，CPU 重新执行跳转指令，保持了执行顺序的正确性。**为了提示软件此时 EPC 中存储的并不是真正的中断异常地址，就引入了 `BD` 位**。

再联系R3000手册上关于BD这一位的描述：

> BD(“branch delay”): 如果设置为1，该位表示EPC**不指向实际上产生异常的指令地址**，而是指向它前面的分支指令（the branch instruction which immediately precedes it）。
>
> When the exception restart point is an instruction which is in the “delay slot” following a branch, EPC has to point to the branch instruction（即发生异常的上一条分支指令的地址）; it is harmless to re-execute the branch, but if the CPU returned from the exception to the branch delay instruction itself the branch would not be taken（如果EPC是直接返回到发生异常的，且位于延迟槽的命令本身的地址，那么在它前面的分支指令就不会被执行） and the exception would have broken the interrupted program.

所以，我们得出结论：当 `BD` 位被置高时，就意味着 EPC 中存储的是**发生中断异常的指令的上一指令的地址**。

Cause(Cause Register)寄存器的IP位(15-8)：记录中断号

`IP` 用于记录当前发生的中断信息，它并不指示异常发生时发生了什么，而是指示当前正在发生什么。MOS中，就是通过对 `IP` 的判定来识别时钟中断的。

也就是说，这里的 `IP` (8-15)记录的是中断号，而下面的 `ExcCode` (2-6)记录的是异常号。在实验中，我们只涉及到了时钟中断。

Cause(Cause Register)寄存器的ExcCode位(6-2)：记录异常号

`ExcCode` 用于记录异常信息，MOS中需要重点关注下列异常码：

异常码	异常名缩写	描述
0	Int	外部中断
1	Mod	TLB项权限位中D为 `0` 时(只读), 向该虚地址写入数据
2	TLBL	TLB load异常, 即从虚地址取值时, 在TLB中找不到对应的表项
3	TLBS	TLB store异常, 即向虚地址存值时, 在TLB中找不到对应的表项
8	Syscall	系统调用, 仅由 `syscall` 指令引发

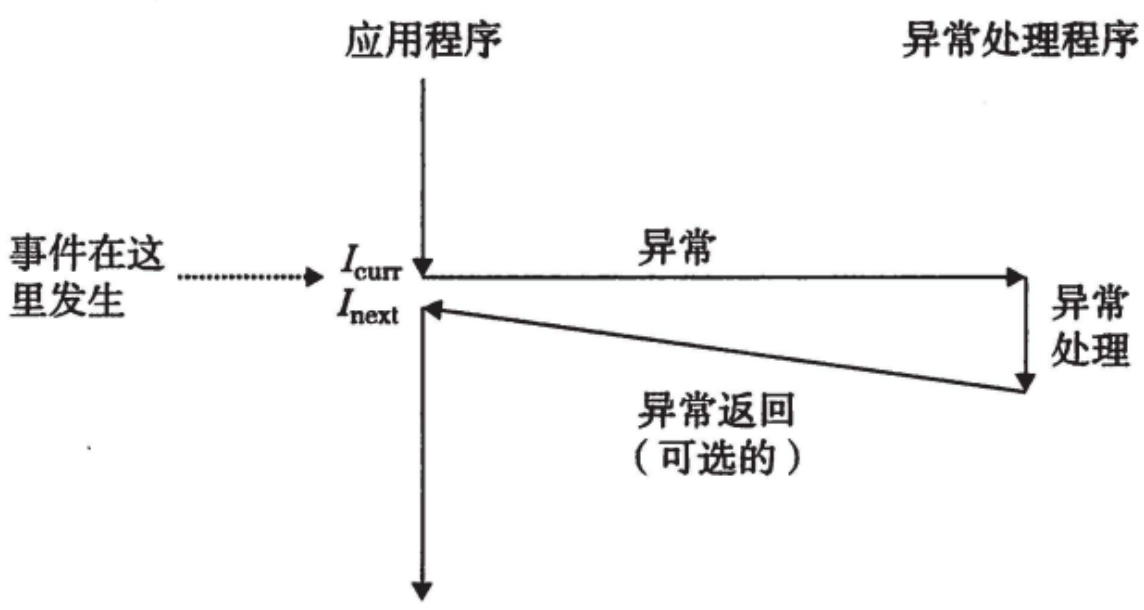
> TLBL 与 TLBS 统称 **TLB 缺失**。

在《See MIPS Run Linux》的第五章中中介绍到，MIPS CPU 处理一个异常时大致要完成四项工作：

1. 设置 EPC 指向异常结束时重新返回的地址。
2. 设置 SR 位，强制 CPU 进入内核态（行驶更高级的特权）并禁止中断。
3. 设置 Cause 寄存器，用于记录异常发生的原因。
4. CPU 开始从异常入口位置取指，此后一切交给软件处理。

而这句“一切交给软件处理”，就是我们当前任务的开始。

我们可以通过一个简单的图来认识一下异常的产生与返回（见下图）。



异常分发程序 except_vec3

异常分发代码位于start.S的开头。

概括来说，这个函数的作用是：

1. 将Cause寄存器中的值取出，存到 k1 寄存器
2. 将 exception_handlers 这个数组的首地址存到了 k0 寄存器
3. 此处和0x7c做按位与，使得 k1 中仅保留Cause寄存器中的ExcCode段 乘4的值
4. addu k0,k1　这条指令做偏移, 使得 k0 的值为 exception_handlers 数组的第ExcCode项
5. jr k0 跳转到该异常处理子函数

```
.section .text.exc_vec3
NESTED(except_vec3, 0, sp)
    .set noat
    .set noreorder
1:
    mfc0 k1,CP0_CAUSE           # 将Cause寄存器中的值取出，存到 k1 寄存器
    la k0,exception_handlers    # la 为扩展指令，意为load address。
                                # 该指令将 exception_handlers 这个数组的首地址存到了
k0 寄存器。
    andi k1,0x7c                # 此处和0x7c做按位与，使得 k1 中仅保留Cause寄存器中的
ExcCode段                                   # 0x7c=01111100，说明 k1 中存的是
ExcCode(2-6)乘4的值
    addu k0,k1
    # 在前面的语句中，k0 寄存器保存了 exception_handlers 的首地址，k1 保存了ExcCode乘4
的值，
    # 此处做偏移，使得 k0 的值为 exception_handlers 数组的第ExcCode项
    lw k0,(k0)
    # 将此时 k0 指向的数组项取出，仍存到 k0 中去。这里 lw k0 (k0) 就是 lw k0 (k0)
    # 此时，k0 的值就是数组第ExcCode项的值了，也就是异常码ExcCode对应的异常处理子函数的入口
地址。
    nop
    jr k0                       # 跳转到该异常处理子函数
    nop
END(except_vec3)
.set at
```

值得注意的是, 这里完全使用 `k0` ，`k1` 两个内核保留的寄存器，**可以保证用户态下其他通用寄存器的值不被改变**，也就保持了现场不变。

这里再回想起TF开头的宏中（即TrapFrame相关），有这样的注释

```
/*
 * $26 (k0) and $27 (k1) not saved
 */
```

联系这里，就不难理解上面这句注释的含义了。

> 此外，这里涉及到了两个 `.set` 指令：
>
> `noat` ：意味着接下来的代码中**不允许汇编器使用 `at` 寄存器(即 1 号寄存器)**。这是因为此时刚刚陷入内核，还未保存现场，用户态下除了 `k0` ，`k1` 之外都不能够被改变。
>
> `noreorder` ：意味着接下来的代码中不允许汇编器重排指令顺序。

.text.exc_vec3 段需要被链接器放到特定的位置，在 R3000 中这一段是要求放到地址 0x80000080 处，这个地址处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。因此，我们在lab3中需要在 tools/scse0_3.lds 中的开头位置增加如下代码，即将.text.exc_vec3 放到 0x80000080 处, 来为操作系统增加异常分发功能。

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

R3000的异常入口

根据R3000手册, R3000的异常入口有5个, 此处列出其中在MOS中最重要的两个入口:

入口地址	所在内存区	描述
`0x80000000`	kseg0	TLB缺失时, PC转至此处
`0x80000080`	kseg0	对于除了TLB缺失之外的其他异常, PC转至此处

注意，在MOS中，仅仅实现了 `0x80000080` 处的异常处理程序。这样也能工作的原因是：当TLB缺失时，PC转至 `0x80000000` 后，**空转32周期(执行了32条 `nop` 指令)**，到达 `0x80000080` ，随后也就与其他的异常一同处理了。也就是说我们只需要实现 `0x80000080` 处的程序即可。

异常向量组

异常分发程序通过 exception_handlers 数组定位异常处理子程序，而 exception_handlers 就称作异常向量组。

lib/traps.c 中的 trap_init() 函数说明了异常向量组里存放了什么。

trap_init() 和 set_except_vector()

```
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);
    set_except_vector(1, handle_mod);
    set_except_vector(2, handle_tlb);
    set_except_vector(3, handle_tlb);
    set_except_vector(8, handle_sys);
}
// 向异常向量组 exception_handlers 中数组下标为 n 的地方存入异常处理函数地址 addr
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;
}
```

实际上，trap_init()函数实现了对全局变量 exception_handlers[32] 数组初始化的工作，即通过把相应处理函数的地址填到对应数组项中，初始化了如下异常：

0 号异常的处理函数为 `handle_int`，表示中断，由时钟中断、控制台中断等中断造成

1 号异常的处理函数为 `handle_mod`，表示存储异常，进行存储操作时该页被标记为只读

2 号异常的处理函数为 `handle_tlb`，TLB 异常，TLB 中没有和程序地址匹配的有效入口

3 号异常的处理函数为 `handle_tlb`，TLB 异常，TLB 失效，且未处于异常模式（用于提高处理效率）

8 号异常的处理函数为 `handle_sys`，系统调用，陷入内核，执行了 syscall 指令

注意：异常不等于中断，x号中断和x号异常不一样。

异常处理子函数

handle_int 处理中断（interrupt）（其实在我们的实验中只有4号中断）

handle_int函数的实现在lib/genex.S

主要效果为：查Cause寄存器，通过掩码操作，获知当前的中断线。正如前面对Cause寄存器介绍的那样，在已经确定了 `excEode` 位为0，代表中断的前提下，接下来在hand_int函数中要做的就是查Cause寄存器的IP部分（8-15位）若为4号中断（时钟中断），则直接调用 `sched_yield`。

由于我们的实验中只有STATUSF_IP4这一个 **8-15位上的某一位为1，其余为0** 的宏，所以hand_int函数也只会处理一种中断——4号中断/时钟中断。

```
    .set noreorder
    .align 5
NESTED(handle_int, TF_SIZE, sp)
    SAVE_ALL    # 现在异常处理栈中已经有了异常发生时的上下文环境
    .set at
    mfc0    t0, CP0_CAUSE           # 将CP0_CAUSE存入t0
    mfc0    t2, CP0_STATUS          # 将CP0_STATUS存入t2
    and     t0, t2                  # 将t0 t2与运算，结果存入t0
    # 这里记录一下：SR寄存器和Cause寄存器的值进行与运算，主要是为了看第8-15位，
    # 两个寄存器的这些位必须同时为1，这些位对应的中断才会触发
    andi    t1, t0, STATUSF_IP4     # t0和0x1000与运算，即取t0的第12位。存入t1
    # 也就是说，这里第三个参数换成其他的宏，只要是8-15位上的某一位为1，其余为0，就是另一种中断
    bnez    t1, timer_irq           # t1不为0，即为1，说明4号中断触发，进入timer_irq:
    nop
END(handle_int)

timer_irq:
    sb zero, 0xb5000110             # 写 0xb5000110 地址响应时钟中断
1:  j   sched_yield
    nop
    /*li t1, 0xff
    lw    t0, delay
    addu  t0, 1
    sw  t0, delay
    beq t0,t1,1f
    nop*/
    j   ret_from_exception
    nop
```

- 由于当前 MOS 仅支持 4 号中断，因此此处仅对 4 号中断进行了识别。这里对应的宏便是 `STATUSF_IP4 0x1000`
- 若需要增加中断类型（如键盘中断），则需要在此处进行额外的掩码判定。

handle_mod、handle_tlb

handle_int函数的实现在lib/genex.S

但是它们是通过另一个汇编宏函数 `BUILD_HANDLER` 来实现的

BUILD_HANDLER

> .align 是用于地址对齐的伪指令，用来指定符号的对齐方式。.align的作用范围只限于紧跟它的那条指令或者数据，而接下来的指令或者数据的地址由上一条指令的地址和其长度决定。
>
> .align 4 //按 4 个字节的倍数对齐下一个符号，空隙默认用0 来填充

```
.macro  __build_clear_sti
    STI
.endm

.macro  __build_clear_cli
    CLI
.endm

.macro  BUILD_HANDLER exception handler clear
    .align  5
    NESTED(handle_\exception, TF_SIZE, sp)
    .set    noat
nop
    SAVE_ALL                    # 保存现场到异常处理栈
    __build_clear_\clear
    .set    at
    move    a0, sp      # 将压栈以后的异常处理栈存入a0
    # (4号中断是TIMESTACK - struct Trapframe处，其他异常则是KERNEL_SP - struct
Trapframe处)
                        # 此时 a0 作为下面的 \handle 对应的函数 的 参数
                        # do_refill 不需要参数，所以是 page_fault_handler
                        # 那么这里的参数就是
    jal \handler        # 在ra处存入这里下一部分的地址，跳转到 \handle 参数对应的函数的位
置
    nop                 # tlb异常的时候执行 do_refill，存储异常的时候调用
page_fault_handler
    j   ret_from_exception      # 调转到 ret_from_exception 恢复现场与回滚
    nop
    END(handle_\exception)
.endm


BUILD_HANDLER tlb   do_refill   cli
BUILD_HANDLER mod   page_fault_handler cli
```

补档：TLB重填流程

TLB 的重填过程由 do_refill 函数（lib/genex.S）完成，相关流程我们在介绍两级页表时已经有所了解，但当时并没有涉及 TLB 部分，加入 TLB 后，整个流程大致如下：

1. **确定此时的一级页表基地址：**mCONTEXT 中存储了当前进程一级页表基地址位于 kseg0 的虚拟地址；

   通过自映射相关知识，可以计算出对于每个进程而言，0x7fdff000 这一虚拟地址也同样映射到该进程的一级页表基地址，但是重填时处于内核 态，如果使用 0x7fdff000 则还需要额外确定当前属于哪一个进程，使用位于 kseg0 的虚拟地址可以通过映射规则直接确定物理地址。

2. 从 BadVaddr 中取出引发 TLB 缺失的虚拟地址, 并确定其对应的一级页表偏移量（高 10 位）；

3. 根据一级页表偏移量, 从一级页表中取出对应的表项：此时取出的表项由**二级页表基地址的物理地址与权限位组成；**
4. 判定权限位: 若权限位显示该表项无效（无 PTE_V ）, 则调用 page_out ，随后回到第一步；
5. 确定引发 TLB 缺失的虚拟地址对应的二级页表偏移量（中间 10 位），与先前取得的二级页表基地址的物理地址共同确认二级页表项的物理地址；
6. 将**二级页表项物理地址转为位于 kseg0 的虚拟地址**（高位补 1），随后页表中取出对应的表项：**此时取出的表项由物理地址与权限位组成**；
7. 判定权限位: 若权限位显示该表项无效（无 PTE_V），则调用 page_out ，随后回到第一步；（PTE_COW 为写时复制权限位，将在 lab4 中用到，此时将所有页表项该位视为 0 即可）
8. 将物理地址存入 EntryLo , 并**调用 tlbwr 将此时的 EntryHi 与 EntryLo 写入到 TLB 中**（EntryHi 中保留了虚拟地址相关信息）。

其中第 4 步与第 7 步均可能调用 **page_out 函数**（mm/pmap.c）来**处理页表中找不到对应表项的异常**

do_refill

如果物理页面在页表中存在，则会将其填入 TLB 并 返回异常地址再次执行内存存取的指令。如果物理页面不存在，则会触发一个一般意义的缺页错误，并跳转到 mm/pmap.c 中的 `pageout` 函数：如果存取地址是合法的用户空间地址，内核会为对应地址分配并映射一个物理页面（被动地分配页面）来解决缺页的问题。

```
NESTED(do_refill, 0 , sp)               # 定义do_refill函数
            //li    k1, '?'
            //sb    k1, 0x90000000
##1. 确定此时的一级页表基地址：mCONTEXT 中存储了当前进程一级页表基地址位于 kseg0 的虚拟地址;
            .extern mCONTEXT            # 外部引入mCONTENT变量
//this "1" is important
1:          //j 1b
            nop
            lw      k1,mCONTEXT         # 将 mCONTENT 的值（当前进程一级页表基地址）存入 k1
            and     k1,0xfffff000       # 将 k1 的值与运算0xfffff000的结果存入k1(低12位置零)
##2. 从 BadVaddr 中取出引发 TLB 缺失的虚拟地址，并确定其对应的一级页表偏移量（高 10 位）；
            mfc0    k0,CP0_BADVADDR     # 将 BADVADDR 寄存器中的值（即引发TLB异常的地址）存入 k0
            srl     k0,20               # 将 k0 中的值右移20位
            and     k0,0xfffffffc       # 将 k0 的值低2位置零
# 此时，k0中存放的就是一级页表偏移量*4的结果，正好是页表项个数*页表项大小4字节，得到了实际上的地址偏移量
            addu        k0,k1           # 将 k0 + k1 的值存入k0
##3. 根据一级页表偏移量，从一级页表中取出对应的表项：此时取出的表项由二级页表基地址的物理地址与权限位组成;
            lw      k1,0(k0)            # 将此时k0中地址 存放的内容存入k1
# 此时 k1中就是一级页表项中存放的内容
##4. 判定权限位：若权限位显示该表项无效（无 PTE_V ），则调用 page_out ，随后回到第一步；
            nop
                    move        t0,k1   # k1 -> t0
                    and     t0,0x0200   # PTE_V 0x0200，这里就是检查一级页表项的权限位（有效位）
                    beqz        t0,NOPAGE   # 如果为0，说明一级页表项无效，那么跳转到NOPAGE
                                        # 在NOPAGE中跳回 pageout
            nop
##5. 确定引发 TLB 缺失的虚拟地址对应的二级页表偏移量（中间 10 位），与先前取得的二级页表基地址的物理地址共同确认二级页表项的物理地址;
            and     k1,0xfffff000       # k1 低12位置零
```

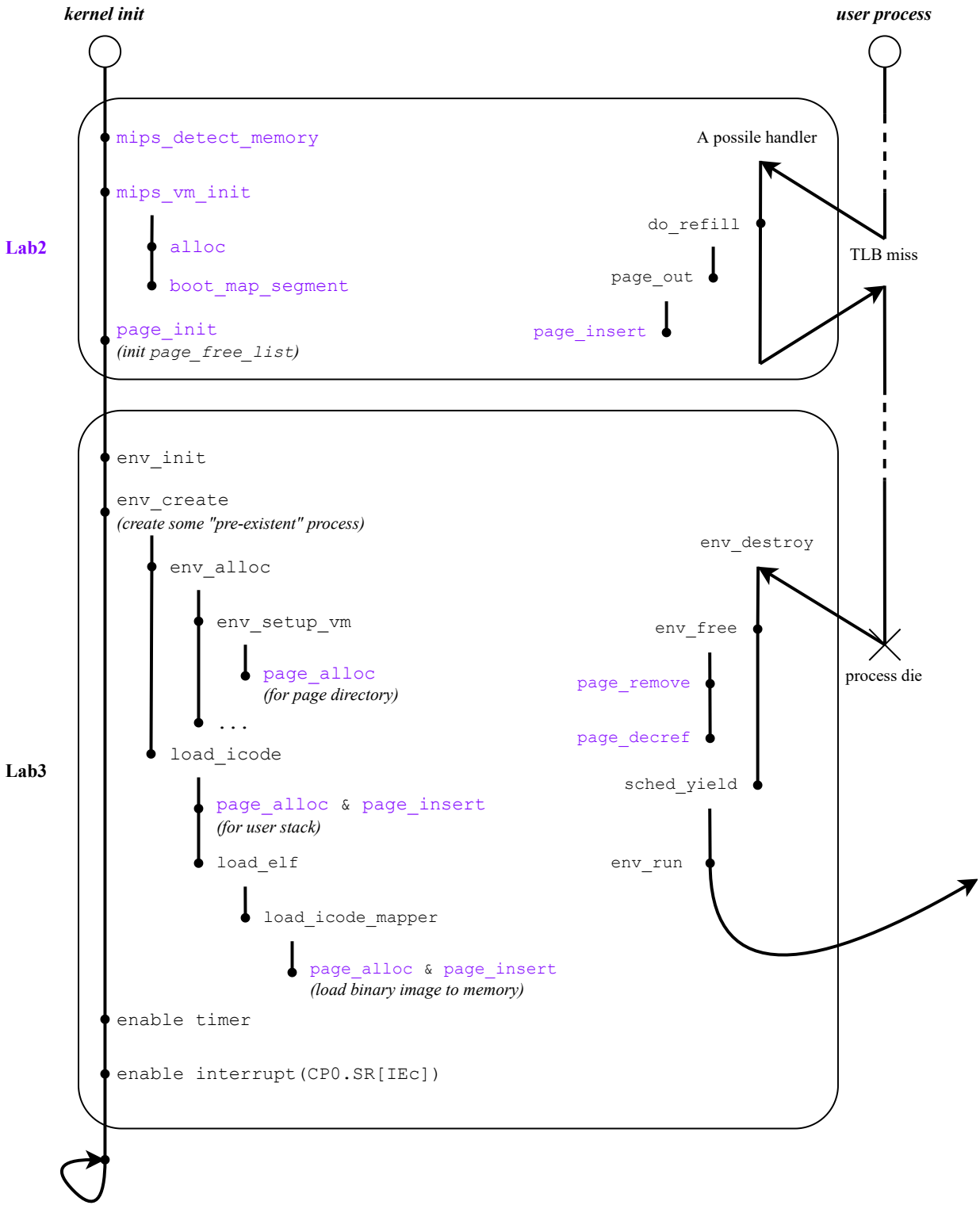
```
# 此时，k1中存放的就是对应的二级页表基地址的物理地址
            mfc0    k0,CP0_BADVADDR     # 将 BADVADDR 寄存器中的值存入 k0
            srl     k0,10               # 将 k0 中的值右移10位
            and     k0,0xfffffffc       # 将 k0 的值低2位置零
            and     k0,0x00000fff       # 将 k0 的值高20位置零
# 此时，k0中存放的就是二级页表偏移量*4的结果，正好是页表项个数*页表项大小4字节，得到了实际上的
地址偏移量

            addu        k0,k1           # 将 k0 + k1 的值存入k0
                                # 此时 k0 存放的就是二级页表项的物理地址
##6. 将二级页表项物理地址转为位于 kseg0 的虚拟地址（高位补 1），随后页表中取出对应的表项：此
时取出的表项由物理地址与权限位组成；
            or      k0,0x80000000       # 将 k0 的最高位 置 1
                                # 此时，k0中存放的就是二级页表项的虚拟地址
            lw      k1,0(k0)            # 将此时k0中地址 存放的内容存入k1
##7. 判定权限位：若权限位显示该表项无效（无 PTE_V），则调用 page_out ，随后回到第一步；
（PTE_COW 为写时复制权限位，将在 lab4 中用到，此时将所有页表项该位视为 0 即可）
            nop
                    move        t0,k1   # k1 -> t0
                    and     t0,0x0200   # PTE_V 0x0200，这里就是检查二级页表项的权限
位（有效位）
                    beqz        t0,NOPAGE   # 如果为0，那么跳转到NOPAGE
                                            # 在NOPAGE中跳回 pageout
            nop
            move        k0,k1   # k1 -> k0
            and     k0,0x1      # PTE_COW 0x0001，这里就是检查二级页表项的权限位（写时
复制位）
            beqz        k0,NoCOW    # 如果为0，那么跳转到NOCOW
            nop
##8. 将物理地址存入 EntryLo ，并调用 tlbwr 将此时的 EntryHi 与 EntryLo 写入到 TLB 中
（EntryHi 中保留了虚拟地址相关信息）
            and     k1,0xfffffbff   # 将k1（二级页表项）的第10位（即可写位PTE_R
0x0400）置零
                                    # 即现在的二级页表项是 只读
NoCOW:
            mtc0    k1,CP0_ENTRYLO0 # 将k1寄存器的值（即物理页号+权限位）存入 EntryLo
            nop
            tlbwr                   # 调用 tlbwr 将此时的 EntryHi 与 EntryLo 写入到
TLB 中

            j       2f              # 调转到下面的 2 标签处 f 即 forward
            nop
NOPAGE:
//3: j 3b
nop
            mfc0    a0,CP0_BADVADDR     # 将 BADVADDR 寄存器中的值存入 a0
            lw      a1,mCONTEXT         # 将 mCONTEXT 存入 a1
            nop

            sw      ra,tlbra            # 将 ra 寄存器中的值存入 tlbra 这个全局变量
            # 这里存放的 ra 是 do_refill 函数执行的下一步
            jal     pageout             # 调用 page_out函数，这里是C函数，
                                        # ra被写入返回地址，即 pageout 函数执行的下一
步
                                        # C函数转为汇编后，会在结尾加一个jr ra 跳回来
            nop
//3: j 3b
nop
            lw      ra,tlbra            # 将 tlbra 这个全局变量的值存入 ra 寄存器中
            # 其实也就是将 do_refill 函数执行完以后应该回去的地方保留下来
            nop
            j   1b                      # 跳转回到 do_refill 函数的开始部分 1 标签处
b 即 back
2:          nop
```

```
            jr      ra          # 跳转回到 ra，注意这里的 ra 是 do_refill 函数执行的下
一步
            nop
END(do_refill)
```

page_out()

1. 若 TLB 缺失时，mCONTEXT 不位于 kseg0, 则说明系统出现了故障。
2. 若引发 TLB 缺失的虚拟地址不合法（如访问了过低或过高的地址），则说明系统故障。
3. 否则可以为此虚拟地址申请一个物理页面（page_alloc），并将虚拟地址映射到该物理页面（page_insert）。(被动扩充内存)

```
void pageout(int va, int context)
{
    u_long r;
    struct Page *p = NULL;
    if (context < 0x80000000) {
        panic("tlb refill and alloc error!");
    }
    if ((va > 0x7f400000) && (va < 0x7f800000)) {
        panic(">>>>>>>>>>>>>>>>>>>>>>>it's env's zone");
    }
    if (va < 0x10000) {
        panic("^^^^^^TOO LOW^^^^^^^^^");
    }
    if ((r = page_alloc(&p)) < 0) {
        panic ("page alloc error!");
    }
    p->pp_ref++;
    page_insert((Pde *)context, p, VA2PFN(va), PTE_R);
    printf("pageout:\t@@@___0x%x___@@@  ins a page \n", va);
}
```



page_fault_handler()

根据 `BUILD_HANDLER` 中的分析，page_fault_handler()函数的参数 `struct Trapframe *tf` 一定是 `KERNEL_SP - struct Trapframe`

```
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);
    // 将异常处理栈中KERNEL_SP - struct Trapframe的现场信息拷贝到struct Trapframe
PgTrapFrame里
```

```
    bcopy(tf, &PgTrapFrame,sizeof(struct Trapframe));
    // 如果当前环境中用户栈指针在 [curenv->env_xstacktop - BY2PG, env_xstacktop - 1]
之间
    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_sys（这个放到lab4里面再解释）

handle_sys函数的实现在lib/syscall.S。

虽然 lab3 中没有用到系统调用，但我们在这里还是简单介绍一下，相关机制在 lab4 中会用到。系统调用是通过在用户态执行 syscall 指令来触发的，一旦触发， 根据上面的介绍，会调用.text.exc_vec3 代码段的代码进行异常分发，最终调用 handle_sys 函数。

这个函数实质上也是一个分发函数，首先这个函数需要从用户态拷贝参数到内核 中，然后根据第一个参数（是一个系统调用号）来作为一个数组的索引，取得该索 引项所对应的那一项的值，其中这个数组就是sys_call_table(系统调用表)，数组 里面存放的每一项都是位于内核中的系统调用服务函数的入口地址。一旦找到对 应的入口地址，则跳转到该入口处进行代码的执行。

保存现场

MIPS-R3000的现场，是指其寄存器的值 - PC, GRF, CP0, HI, LO的值。在MOS中，采用 `struct Trapframe` 结构来存储现场，也就是 `struct Env` 中的 `env_tf` 段。

当前进程被中断异常打断时，需要先使用汇编宏 `SAVE_ALL` (位于include/stackframe.h)**保存现场**，然后再**进行中断异常处理**。

> 由于处理中断异常的过程仍然需要使用CPU中的各种寄存器，因此需要先将现场暂存，处理完毕后再恢复现场。

获知异常处理栈 get_sp

在保存现场之前，需要解决一个重大问题：保存到什么地方(异常处理栈)？MOS代码中include/stackframe.h中的汇编宏 `get_sp` 给出了答案。

该宏的简要解释:

1. 首先获取Cause寄存器的值，存入 `k1` 。

2. 若发生了4号中断(4号中断线接时钟, 即时钟中断)，则设置 `sp` 寄存器为 `0x82000000` (即 `TIMESTACK`)；否则设置 `sp` 寄存器为 `KERNEL_SP` (此为一个全局变量，用于保存此时内核栈的栈顶，初始化于lib/kclock_asm.S的 `set_timer` 汇编函数中)。

> 按照 `KERNEL_SP` 的语义, 其初始化时应当指示内核栈顶, 因此在 `mips_init` 中最后一个函数处初始化.

```
.macro get_sp
    mfc0    k1, CP0_CAUSE       # 首先获取Cause寄存器的值，存入 k1
    andi    k1, 0x107c          # 0x107c = 0001 0000 0111 1100 k1取对应位
    xori    k1, 0x1000          # 0x1000 = 0001 0000 0000 0000
                                # 000() 0000 0()()() ()()00
    bnez    k1, 1f
    # 如果k1取对应位以后是 0001 0000 0000 0000 则异或后就是 0 ，等于0说明发生了4号中断
    # SR的第12bit设置为1，表示4号中断可以被响应。而在异或后为0---就表示Cause的第12位是1，
    # 硬件中断是15-10位，那么第12位为1，即100 (12 11 10)，就是4号中断
    nop
    li      sp, 0x82000000      # 发生了4号中断，设置 `sp` 寄存器为 TIMESTACK 的值
0x82000000
    j       2f
    nop
1:                              # 没发生4号中断
    bltz    sp, 2f              # sp小于0，结束
    nop
    lw      sp, KERNEL_SP       # 设置 sp 寄存器为 KERNEL_SP
    nop
2:
    nop
.endm
```

这里使用 `1f/2f` 简易标签来指示跳转地址： `1f` 中 `f` 表示 forward, 意为后面的 `1` ； `2f` 同理。

接下来就可以将现场信息保存到 `sp` 指示的异常处理栈中了。

> **`TIMESTACK` 仅在时钟中断时被使用，这其实是与进程切换达成了"协议"**：因为时钟中断的处理函数就是 `sched_yield` ，最终在 `env_run` 时将 `TIMESTACK` 中保存的现场存入当前进程的 `env_tf` 中。

保存现场到异常处理栈 SAVE_ALL

保存现场的主要逻辑处于汇编宏 `SAVE_ALL` 中，主要逻辑如下：

前四行其首先取出了 SR 寄存器的值, 然后利用移位等操作判断第 28 位的值, 根据前面的讲述我们可以知道, 也即判断当前是否处于用户模式下。

接下来将当前运行进程的用户栈的地址保存到 k0 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，获取处理栈指针，再将处理栈顶下移一个 struct Trapframe 的大小；将之前的运行栈地址与 2 号寄存器 $v0 先保存起来，便于后面可以放心的使用 sp 寄存器与 v0 寄存器。剩下的部分便是用 `xxx(sp)` 的寻址方式复制整个 struct Trapframe 需要的信息，将各个cpu寄存器中的值复制到sp以上的对应位置。

这个宏函数相当于只改变了CPU中 k0 k1 v0三个寄存器的值，现在它们三个的值分别是：

- k0，用户栈指针
- k1，处理栈顶
- v0，EntryLo寄存器的值

此时的sp指针是处理栈下移一个 struct Trapframe 的大小以后的地址。

```
.macro SAVE_ALL
```

```
        mfc0    k0,CP0_STATUS   # 将 SR 寄存器中的值存放到 k0 寄存器中
        sll     k0,3      /* extract cu0 bit */ # 现在cu0被移到了最高位
        # 第28bit CU0设置为1，表示允许在用户模式下使用 CP0 寄存器。
        bltz    k0,1f       # 如果cu0是1，则在移到最高位的情况下，就会小于0
        nop
        /*
         * Called from user mode, new stack
         */
        //lui   k1,%hi(kernelsp)
        //lw    k1,%lo(kernelsp)(k1)  //not clear right now

1:
        move    k0,sp       # 首先将用户栈指针存入 k0
        get_sp              # 然后使用 get_sp 获取处理栈指针（两种结果都是内核栈指针），
存入sp
        move    k1,sp       # 再将找到的处理栈指针存入k1
        subu    sp,k1,TF_SIZE   # 将处理栈顶下移一个 struct Trapframe 的大小，压栈后的
指针存入sp
                                # (TF_SIZE 为 sizeof(struct Trapframe))
        sw  k0,TF_REG29(sp) # 将用户栈指针 k0 存入 Trapframe 中的 TF_REG29 处
        sw  $2,TF_REG2(sp)  # v0 寄存器即 $2。由于借助 v0 这个用户态下会用到的寄存器，因
此必须先将用户态下 v0 的值存入 Trapframe 中
    # 借助 v0 寄存器将CP0寄存器以及HI，LO寄存器的值存入 Trapframe 中的对应位置
        mfc0    v0,CP0_STATUS
        sw  v0,TF_STATUS(sp)
        mfc0    v0,CP0_CAUSE
        sw  v0,TF_CAUSE(sp)
        mfc0    v0,CP0_EPC
        sw  v0,TF_EPC(sp)
        mfc0    v0, CP0_BADVADDR
        sw  v0, TF_BADVADDR(sp)
        mfhi    v0
        sw  v0,TF_HI(sp)
        mflo    v0
        sw  v0,TF_LO(sp)
        # 下面的GRF复制中没有$2  $29
        # 由于 $2 （v0 寄存器），$29 （sp 寄存器）都已经保存了，所以此处跳过它们.
        sw  $0,TF_REG0(sp)
        sw  $1,TF_REG1(sp)      # 跳过 $2
        sw  $3,TF_REG3(sp)
        sw  $4,TF_REG4(sp)
        sw  $5,TF_REG5(sp)
        sw  $6,TF_REG6(sp)
        sw  $7,TF_REG7(sp)
        sw  $8,TF_REG8(sp)
        sw  $9,TF_REG9(sp)
        sw  $10,TF_REG10(sp)
        sw  $11,TF_REG11(sp)
        sw  $12,TF_REG12(sp)
        sw  $13,TF_REG13(sp)
        sw  $14,TF_REG14(sp)
        sw  $15,TF_REG15(sp)
        sw  $16,TF_REG16(sp)
        sw  $17,TF_REG17(sp)
        sw  $18,TF_REG18(sp)
        sw  $19,TF_REG19(sp)
        sw  $20,TF_REG20(sp)
        sw  $21,TF_REG21(sp)
        sw  $22,TF_REG22(sp)
        sw  $23,TF_REG23(sp)
        sw  $24,TF_REG24(sp)
        sw  $25,TF_REG25(sp)
        sw  $26,TF_REG26(sp)
        sw  $27,TF_REG27(sp)
        sw  $28,TF_REG28(sp)        # 跳过 $29
        sw  $30,TF_REG30(sp)
```

```
        sw  $31,TF_REG31(sp)
.endm
```

恢复现场与回滚 RESTORE_SOME

恢复现场的主要逻辑由汇编宏 `RESTORE_SOME` (include/stackframe.h)支持。此宏将现场中除了 `sp` `k1` `k0` 寄存器之外的所有寄存器都恢复到CPU中：

```
.macro RESTORE_SOME
    # sp在此时依然是内核异常处理栈指针
    .set mips1
    lw      v0, TF_STATUS(sp)    # 从现场中取出 SR 寄存器的值放入 v0
    mtc0    v0, CP0_STATUS       # 以 v0 为中介，将 SR 寄存器的 值恢复
    lw      v1, TF_LO(sp)        # 恢复 EntryLo 寄存器的值
    mtlo    v1
    lw      v0, TF_HI(sp)        # 恢复 EntryHi 寄存器的值
    mtlo    v0
    lw      v1, TF_EPC(sp)       # 恢复 EPC 寄存器的值
    mtc0    v1, CP0_EPC
    # 接下来恢复 除了 29 27 26（即sp k1 k0）以外的通用寄存器的值
    lw      $31, TF_REG31(sp)
    lw      $30, TF_REG30(sp)
    //lw      $29, TF_REG29(sp) <~~ $29 <=> sp
    lw      $28, TF_REG28(sp)
    //lw      $27, TF_REG27(sp) <~~ $27 <=> k1
    //lw      $26, TF_REG26(sp) <~~ $26 <=> k0
    lw      $25, TF_REG25(sp)
    lw      $24, TF_REG24(sp)
    lw      $23,TF_REG23(sp)
    lw      $22,TF_REG22(sp)
    lw      $21,TF_REG21(sp)
    lw      $20,TF_REG20(sp)
    lw      $19,TF_REG19(sp)
    lw      $18,TF_REG18(sp)
    lw      $17,TF_REG17(sp)
    lw      $16,TF_REG16(sp)
    lw      $15,TF_REG15(sp)
    lw      $14,TF_REG14(sp)
    lw      $13,TF_REG13(sp)
    lw      $12,TF_REG12(sp)
    lw      $11,TF_REG11(sp)
    lw      $10,TF_REG10(sp)
    lw      $9,TF_REG9(sp)
    lw      $8,TF_REG8(sp)
    lw      $7,TF_REG7(sp)
    lw      $6,TF_REG6(sp)
    lw      $5,TF_REG5(sp)
    lw      $4,TF_REG4(sp)
    lw      $3,TF_REG3(sp)
    lw      $2,TF_REG2(sp)
    lw      $1, TF_REF1(sp)
.endm
```

此处并未恢复 `k0/k1` 这两个内核态寄存器, 这是 **由于在 MIPS 规范下, 用户程序不会使用这两个寄存器. 而用户程序的编写者也应该遵循这个规范。**

- 若需要直接编写汇编代码, 应当时刻遵循此规范.
- 若通过编译器得到汇编代码, 则应当选择符合规范的编译器, 如: mips-4KC.

恢复现场与回滚 ret_from_exception

通过 `RESTORE_SOME` ，可以使得用户现场中，除了 `sp` 栈指针寄存器外的信息都被恢l复，而 `sp` 寄存器将在回滚现场前的最后关头才恢复。**MOS中定义了汇编函数 `ret_from_exception` (lib/genex.S)用于恢复现场并回滚用户态**。

```
FEXPORT(ret_from_exception)
    .set noat
    .set noreorder
    RESTORE_SOME
    .set at
    lw  k0,TF_EPC(sp)           # 将异常处理栈中EPC对应的位置存入k0，
                        # 对于写入页异常来说，此时EPC中存放着真正处理写入页异常的函数
pgfault，
    # 此时sp的值依然是0x82000000-TF_SIZE (TF_SIZE 为 sizeof(struct Trapframe)
    lw  sp,TF_REG29(sp) /* Deallocate stack */
    # 在SAVE_ALL中，将用户栈指针由sp 存入 k0 再存入 Trapframe 中的 TF_REG29 处，这里就是
将sp重置为用户栈指针
//1:    j   1b
    nop
    jr  k0                      # 异常处理完毕，跳转到中断时的位置继续执行
    rfe
```

此汇编函数首先使用 `RESTORE_SOME` 将除了 `sp` 外的寄存器得到恢复. 随后将栈指针恢复到用户进程栈, 再**通过R3000标准回滚指令序列 `jr k0 + rfe` 恢复用户态**.

> *R3000 中没有提供 `eret` 指令, 而仅仅提供了 `rfe` 指令. 详情参看 R3000 文档.*

rfe指令解释如下：

> rfe --- restore from exception
>
> Note that this is not “return from exception”. This instruction restores the status register to go back to the state prior to the trap. To understand what it does, refer to the status register SR defined later in this chapter. The only secure way of returning to user mode from an exception is to return with a jr instruction which has the rfe in its delay slot.

> 从异常返回：控件最终必须在输入时返回到存储在EPC中的值。
>
> 无论是哪种异常，软件都必须在异常返回后重新调整SR。特殊指令rfe就会完成该工作；
>
> 但请注意，它不会转移控制权。为了实现跳回，软件必须将原始EPC值加载回通用寄存器，并使用jr操作。

时钟中断

在前面的介绍中我们已经知道 Cause 寄存器中有 8 个独立的中断位。其中 6 位来自外部，另外 2 位是由软件进行读写，且不同中断处理起来也会有差异。所以在完成这一部分内容之前，我们首先来介绍一下从 CPU 到操作系统中关于中断处理的普遍性流程。

1. 将当前 PC 地址存入 CP0 中的 EPC 寄存器。
2. 将 IEc,KUc 拷贝至 KUp 和IEp 中，同时将 IEc 置为 0，表示关闭全局中断使能，将 KUc 置 1，表示处于内核态。
3. 在 Cause 寄存器中，保存 ExcCode 段。由于此处是中断异常，对应的异常码即为 0。
4. PC 转入异常分发程序入口。
5. 通过异常分发，判断出当前异常为中断异常，随后进入相应的中断处理程序。在MOS 中即对应 handle_int 函数。
6. 在中断处理程序中进一步判断 CP0_CAUSE 寄存器中是由几号中断位引发的中断，然后进入不同中断对应的中断服务函数。
7. 中断处理完成，将 EPC 的值取出到 PC 中，恢复 SR 中相应的中断使能，继续执行。

以上流程中 1-4 以及第 7 步是由 CPU 完成的，真正需要我们完成的只有 5-6 步，而且在这一部分我们只需要完成外设中断中的时钟中断。

下面我们来简单介绍一下时钟中断的概念。

时钟中断和操作系统的时间片轮转算法是紧密相关的。时间片轮转调度是一种很公平的算法。每个进程被分配一个时间段，称作它的时间片，即该进程允许运行的时间。如果在时间片结束时进程还在运行，则该进程将挂起，切换到另一个进程运行。那么 CPU是如何知晓一个进程的时间片结束的呢？就是通过定时器产生的时钟中断。当时钟中断产生时，当前运行的进程被挂起，我们需要在调度队列中选取一个合适的进程运行。如何“选取”，就要涉及到进程的调度了。

---

set_timer

这个函数是init/init.c中的函数mips_init()中调用的最后一个函数，但是它本身又调用了很多函数，需要详细解析

复习前面提到的

第28bit CU0设置为1，表示允许在用户模式下使用 CP0 寄存器。

第12bit 设置为1，表示 4 号中断可以被响应。

第0bit `IEc` 为1, 意味着CPU会响应中断; `IEc` 为0, 意味着CPU不会响应中断.

```
.macro  setup_c0_status set clr
    .set    push
    mfc0    t0, CP0_STATUS      # 将CP0_STATUS的内容存入 t0
    or      t0, \set|\clr       # 将t0（SR寄存器）的第0、12、28位置1  IEc 4号中断 CU0
    # (0bit)允许中断 (12bit) 4 号中断可以被响应 (28bit)允许在用户模式下使用 CP0 寄存器
    xor     t0, \clr            # 将t0和0进行异或，
    mtc0    t0, CP0_STATUS
    .set    pop
.endm

    .text
LEAF(set_timer)

    li t0, 0xc8                 # 0xc8表示1秒钟中断200次
    # 0xb5000000 是模拟器(gxemul) 映射实时钟的位置。偏移量为0x100 表示来设置实时钟中断的频率，0xc8 则表示1 秒钟中断200次，如果写入0，表示关闭实时钟。
    sb t0, 0xb5000100
    sw  sp, KERNEL_SP           # 初始化内核栈顶 KERNEL_SP 为 内核栈sp
setup_c0_status STATUS_CU0|0x1001 0     # 0x10000000|0x1001 = 0x10001001
    jr ra

    nop
END(set_timer)
```

下面逐步解读set_timer整个调用的逻辑

Step1 set_timer本身的逻辑——设置时钟并触发4号中断

```
li t0, 0xc8             # 0xc8表示1秒钟中断200次
sb t0, 0xb5000100       # 此处填入 0 就会关闭实时钟
```

首先向0xb5000100 位置写入0xc8，其中0xb5000000 是模拟器(gxemul) 映射实时钟的位置。偏移量为0x100 表示来设置实时钟中断的频率，0xc8 则表示1 秒钟中断200次，如果写入0，表示关闭实时钟。实时钟对于R3000 来说绑定到了4 号中断上，故这段代码其实主要用来触发了4 号中断。注意这里的中断号和异常号是不一样的概念，我们实验的异常包括中断。

以及注意， `KERNEL_SP` 就是在set_timer函数中初始化为用户栈sp。

Step2 触发中断后，PC指向0x80000080 --> 跳转到异常分发程序exc_vec3 --> 异常向量组

一旦实时钟中断产生，就会触发 MIPS 中断，从而 MIPS 将PC 指向0x80000080，从而跳转到 .text.exc_vec3 代码段执行。即执行异常分发程序。对于实时钟引起的中断，通过 .text.exc_vec3 代码段的分发，在异常向量组 exception_handlers 数组中选择正确的下标。根据下标索引，最终会调用 handle_ int 函数来处理实时钟中断。

```
.section .text.exc_vec3
NESTED(except_vec3, 0, sp)
    .set noat
    .set noreorder
1:
    mfc0 k1,CP0_CAUSE           # 将Cause寄存器中的值取出，存到 k1 寄存器
    la k0,exception_handlers    # la 为扩展指令，意为load address。
                                # 该指令将 exception_handlers 这个数组的首地址存到了
k0 寄存器。
    andi k1,0x7c                # 此处和0x7c做按位与，使得 k1 中仅保留Cause寄存器中的
ExcCode段                                   # 0x7c=01111100，说明 k1 中存的是ExcCode
乘4的值
    addu k0,k1
    # 在前面的语句中，k0 寄存器保存了 exception_handlers 的首地址，k1 保存了ExcCode乘4
的值，
    # 此处做偏移，使得 k0 的值为 exception_handlers 数组的第ExcCode项
    lw k0,(k0)
    # 将此时 k0 指向的数组项取出，仍存到 k0 中去。这里 lw k0 (k0) 就是 lw k0 0(k0)
    # 此时，k0 的值就是数组第ExcCode项的值了，也就是异常码ExcCode对应的中断异常处理子函数的
入口地址.
    nop
    jr k0                       # 跳转到该中断处理子函数
    nop
END(except_vec3)
.set at
```

为了实现这一步，exception_handlers 数组必须早已经初始化完毕，所以这也是为什么在 `mips_init()` 函数中， `trap_init()` 函数在 `set_timer()` 函数之前。

Step3 执行handle_init函数

在 handle_ int 函数中，判断CP0_CAUSE寄存器是不是对应的 4 号中断位引发的中断，如果是，则执行中断服务函数timer_ irq。

但是，在handle_init函数中调用了SAVE_ALL的汇编宏函数，以及SAVE_ALL中也调用了get_sp汇编宏函数，所以这里需要进一步展开讨论。

Step3.1 调用SAVE_ALL和get_sp，获取异常处理栈指针，并保存现场到异常处理栈

handle_init在最开始就调用了SAVE_ALL汇编宏函数，所以我们需要解读这个东西。

SAVE_ALL汇编宏函数：前四行其首先取出了 SR 寄存器的值, 然后利用移位等操作判断第 28 位的值, 根据前面的讲述我们可以知道, 也即判断当前是否处于用户模式下。

接下来将当前运行栈的地址保存到 k0 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，获取处理栈指针，再将处理栈顶下移一个 struct Trapframe 的大小；将之前的运行栈地址与 2 号寄存器 $v0 先保存起来，便于后面可以放心的使用 sp 寄存器与 v0 寄存器。剩下的部分便是用 `xxx(sp)` 的寻址方式复制整个 struct Trapframe 需要的信息，将各个cpu寄存器中的值复制到sp以上的对应位置。

```
.macro get_sp
    mfc0    k1, CP0_CAUSE       # 首先获取Cause寄存器的值，存入 k1
    andi    k1, 0x107c          # 0x107c = 0001 0000 0111 1100 k1取对应位
    xori    k1, 0x1000          # 0x1000 = 0001 0000 0000 0000
                                # 000() 0000 0()()() ()()00
    bnez    k1, 1f
    # 如果k1取对应位以后是 0001 0000 0000 0000 则异或后就是 0 ，等于0说明发生了4号中断
    # SR的第12bit设置为1，表示4号中断可以被响应。而在异或后为0---就表示Cause的第12位是1，
    # 硬件中断是15-10位，那么第12位为1，即100 (12 11 10)，就是4号中断
    nop
    li      sp, 0x82000000      # 发生了4号中断，设置 `sp` 寄存器为 TIMESTACK 的值
0x82000000
    j       2f
    nop
1:                              # 没发生4号中断
    bltz    sp, 2f              # sp小于0，结束
    nop
    lw      sp, KERNEL_SP       # 设置 sp 寄存器为 KERNEL_SP
    nop
2:
    nop
.endm

.macro SAVE_ALL
        mfc0    k0,CP0_STATUS
        sll     k0,3       /* extract cu0 bit */ # 现在cu0被移到了最高位
        # 第28bit CU0设置为1，表示允许在用户模式下使用 CP0 寄存器。
        bltz    k0,1f       # 如果cu0是1，则在移到最高位的情况下，就会小于0，说明此时是用
户态
        nop
        /*
         * Called from user mode, new stack
         */
        //lui   k1,%hi(kernelsp)
        //lw    k1,%lo(kernelsp)(k1)  //not clear right now

1:
        move    k0,sp       # 首先将用户栈指针存入 k0
        get_sp              # 然后使用 get_sp 获取处理栈指针，存入 sp
                            # 两种处理栈都是内核栈指针，4号中断是  TIMESTACK，其他都是
KERNEL_SP
        move    k1,sp       # 再将 get_sp 获取的处理栈指针 存入k1
        subu    sp,k1,TF_SIZE   # 将处理栈顶下移一个 struct Trapframe 的大小
                                # (TF_SIZE 为 sizeof(struct Trapframe))
        sw  k0,TF_REG29(sp) # 将用户栈指针 k0 存入 Trapframe 中的 TF_REG29 处（29就是
sp号数）
        sw  $2,TF_REG2(sp)  # v0 寄存器即 $2。由于借助 v0 这个用户态下会用到的寄存器，因
此必须先将用户态下 v0 的值存入 Trapframe 中
        # 以 v0 寄存器为中介，将CP0寄存器以及HI，LO寄存器的值存入 Trapframe 中的对应位置
        # 前面cu0是1，允许在用户态下使用协处理器
        mfc0    v0,CP0_STATUS
        sw  v0,TF_STATUS(sp)
        mfc0    v0,CP0_CAUSE
        sw  v0,TF_CAUSE(sp)
        mfc0    v0,CP0_EPC
        sw  v0,TF_EPC(sp)
        mfc0    v0, CP0_BADVADDR
        sw  v0, TF_BADVADDR(sp)
        mfhi    v0
        sw  v0,TF_HI(sp)
        mflo    v0
        sw  v0,TF_LO(sp)
        # 下面的GRF复制中没有$2  $29
        # 由于 $2 (v0 寄存器) , $29 (sp 寄存器) 都已经保存了，所以此处跳过它们.
        sw  $0,TF_REG0(sp)
```

```
        sw  $1,TF_REG1(sp)      # 跳过 $2
        sw  $3,TF_REG3(sp)
        // 4~27 原函数都有，这里就不写了
        sw  $28,TF_REG28(sp)        # 跳过 $29
        sw  $30,TF_REG30(sp)
        sw  $31,TF_REG31(sp)
.endm
```

Step3.2 判断是不是4号中断位引发中断，是则调用timer_ irq汇编宏函数

判断CP0_CAUSE寄存器是不是对应的 4 号中断位引发的中断，如果是，则执行中断服务函数timer_ irq。

```
#define CP0_STATUS $12
#define CP0_CAUSE $13
#define CP0_EPC $14
// -----事实上，每一个cp0寄存器都有这样的宏定义，用名称代替寄存器号-----------
#define STATUSF_IP4 0x1000
#define STATUS_CU0 0x10000000
#define STATUS_KUC 0x2
```

```
    .set noreorder
    .align 5

NESTED(handle_int, TF_SIZE, sp)
    SAVE_ALL    # 现在异常处理栈中已经有了异常发生时的上下文环境
    .set at
    mfc0    t0, CP0_CAUSE           # 将CP0_CAUSE存入t0
    mfc0    t2, CP0_STATUS          # 将CP0_STATUS存入t2
    and     t0, t2                  # 将t0 t2与运算，结果存入t0
    # 这里记录一下：SR寄存器和Cause寄存器的值进行与运算，主要是为了看第8-15位，
    # 两个寄存器的这些位必须同时为1，这些位对应的中断才会触发
    andi    t1, t0, STATUSF_IP4     # t0和0x1000与运算，即取t0的第12位。存入t1
    # 也就是说，这里第三个参数换成其他的宏，只要是8-15位上的某一位为1，其余为0，就是另一种中断
    bnez    t1, timer_irq           # t1不为0，即为1，说明4号中断触发，进入timer_irq:
    nop
END(handle_int)

timer_irq:
    sb zero, 0xb5000110             # 写 0xb5000110 地址响应时钟中断
1:  j   sched_yield                 # 跳转到 sched_yield 中执行，进程调度
    nop
    /*li t1, 0xff
    lw    t0, delay
    addu  t0, 1
    sw  t0, delay
    beq t0,t1,1f
    nop*/
    j   ret_from_exception          # 恢复现场和回滚
    nop
```

Step4 timer_ irq函数完成进程调度和恢复现场与回滚

首先写 0xb5000110 地址响应时钟中断，之后跳转到 sched_yield 中执行。而 sched_yield 函数会调用引起进程切换的函数来完成进程的切换。注意：这里是第一次进行进程切换，请务必保证：kclock_init 函数在 env_create 函数之后调用。

Step4.1 调用sched_ yield()函数

在timer_ irq 里直接跳转到 sched_ yield 中执行

sched_yield()函数解析

在解析函数之前，先来通过下面的测试程序复习一下C语言中静态局部变量的特性：只会在它所在的函数第一次被调用的时候初始化，并且在每一次函数执行完毕后，它并不会释放，也就是说会保留自己的值。

```
#include <stdio.h>
void f(){
    static int n = 0;
    if(n==0){
        printf("AAA\n");
        n = 5;
    }
    printf("n is %d\n", n);
    n--;
}
int main()
{
    for (int i = 0; i < 15; i++){
        f();
    }
    return 0;
}
```

执行程序以后的输出如下

```
AAA
5
4
3
2
1
AAA
5
4
3
2
1
AAA
5
4
3
2
1
```

了解了这一点以后，我们就可以着眼于这个函数的作用了：

当两个调度队列至少有一个不为空时，从不为空的那个调度队列中不断抽取首元素，插入另一个队列的尾端，寻找状态为 `ENV_RUNNABLE` 的进程，并将这个进程存放在静态变量e中，并且将静态变量count赋值为e的env_pri。

如果抽取的首元素进程e状态是 `ENV_FREE` 和 `ENV_NOT_RUNNABLE` ，则还得继续寻找。

如果找到了状态是 `ENV_RUNNABLE` 的进程e，终止寻找。

然后将count--，并用env_run()运行进程e。相当于进程e运行了一个时间片。

当下一个时钟中断到来时，如果进程e本身的时间片长度大于1，那么count减一后大于0，所以不需要进行进程寻找流程，直接继续count--，注意存放 正在消耗时间片的进程 的e本身也是静态变量，因此运行的还是同一进程。直到这一进程运行完自己的时间片。

结束上述流程后，此时count==0，第一个找到的状态是 `ENV_RUNNABLE` 的进程已经运行完毕，现在重新进入循环，寻找下一个状态是 `ENV_RUNNABLE` 的进程e。

当两个调度队列均为空时，此时函数陷入死循环。

```
void sched_yield(void)
{
    // 由于这三个变量都是static静态变量，因此只会进行一次初始化，而且会保留上一次执行完以后的值
    static int count = 0; // remaining time slices of current env
    static int point = 0; // current env_sched_list index
    static struct Env *e = NULL;
    // 利用静态变量的特性，当count不为0的时候，是不会进入下面的流程中，去找下一个状态是ENV_RUNNABLE的进程e的。但是，这里需要注意的是，在当前进程时间片不为0，而进程状态改变的时候，此时也需要寻找一个新的状态是ENV_RUNNABLE的进程e
    if (count <= 0 || e == NULL || (e->env_status != ENV_RUNNABLE)) {
        do {// 不断寻找调度队列中状态为ENV_RUNNABLE的进程e
            // 注意这里，当两个调度队列为零的时候，能够不断切换，陷入死循环
            if (LIST_EMPTY(&env_sched_list[point])) {
                point = 1 - point;
            }
            // 从调度队列中抽取队列的首元素
            e = LIST_FIRST(&env_sched_list[point]);
            // 如果当前进程不为空
            if (e != NULL) {
                // 无论它是什么状态，都要将他从当前调度队列中移除，插入另一个调度队列的尾端
                LIST_REMOVE(e, env_sched_link);
                LIST_INSERT_TAIL(&env_sched_list[1 - point], e, env_sched_link);
                count = e->env_pri;
                // 将count赋值为e的env_pri，终止寻找
            }
        } while (e == NULL || e->env_status != ENV_RUNNABLE);
        // 找到了状态是ENV_RUNNABLE的进程e，终止寻找，此时count已经被赋值为e的env_pri
    }
    count --;
    env_run(e);// 这里env_run()函数中写了e->env_runs++;
}
```

这里再来复习一下env_run。

(1) 将正在执行的进程（如果有）的现场保存到对应的进程控制块中。（开头的复制）

(2) 将sched_yield为它选好的一个可以运行的进程e，恢复该进程e上次被挂起时候的现场。（调用env_pop_tf函数，这里终于明白这个函数为什么叫这个名字了——弹出进程的环境）

~~Step4.2 ret_from_exception完成恢复现场与回滚~~

然后再跳转到 ret_from_exception 中执行。

这里打上删除线的原因是：在timer_irq中实际上并没有执行 `j ret_from_exception` 。因为在执行完 `sched_yield()` 函数中的 `env_run()` 中的 `env_tf_pop` 以后，就已经回到了用户态。系统跳转至进程的入口，开始在用户态中执行进程。 这里加上ret_from_exception的原因很可能是为了整齐，因为其他的异常处理函数是需要ret_from_exception的。

不过在这里还是解析一下。

通过 `RESTORE_SOME` ，可以使得用户现场中，除了 `sp` 栈指针寄存器外的信息都被恢复，而 `sp` 寄存器将在回滚现场前的最后关头才恢复。**MOS中定义了汇编函数 `ret_from_exception` (lib/genex.S)用于恢复现场并回滚用户态**。

```
FEXPORT(ret_from_exception)
    .set noat
    .set noreorder
    RESTORE_SOME
    .set at
    lw  k0,TF_EPC(sp)            # 将异常处理栈中EPC对应的位置存入k0，
    # 此时sp的值依然是0x82000000-TF_SIZE (TF_SIZE 为 sizeof(struct Trapframe)
    lw  sp,TF_REG29(sp) /* Deallocate stack */
    # 在SAVE_ALL中，将用户栈指针由sp 存入 k0 再存入 Trapframe 中的 TF_REG29 处，这里就是将sp重置为用户栈指针
//1:    j   1b
    nop
    jr  k0                       # 异常处理完毕，跳转到中断时的位置继续执行
    rfe
```

Bonus TLB 中断何时被调用?

从上面的分析看，操作系统在实时钟的驱动下，通过时间片轮转算法实现进程的并发执行，不过如果没有 TLB 中断，真的可以正确的运行吗，当然不行。

因为每当硬件在取数据或者取指令的时候，都会发出一个所需数据所在的虚拟地址，TLB 就是将这个虚拟地址转换为对应的物理地址，进而才能够驱动内存取得正确的所期望的数据。但是当 TLB 在转换的时候发现 **没有 对应于该虚拟地址 的项**，那么此时就会**产生一个 TLB 异常**。

> MyNote: 这里对于每一个进程来说，转化过程如下：在进程页目录映射的二级页表体系所代表的4G空间（也就是进程空间）里的某一个地址env_vaddr，对应的二级页表是额外申请的一页物理内存，二级页表的地址存放在进程的页目录的一级页表项，而二级页表本身的二级页表项中，存放了env_vaddr所代表的进程需要的数据、资源等在内存中实际存放在的物理地址。

TLB 对应的中断处理函数是 handle_tlb，通过宏映射到了 do_refill 函数上： 这个函数完成 TLB 的填充，在 lab2 中我们已经学习了 TLB 的基本结构，简单 来说就是对于不同进程的同一个虚拟地址，结合 ASID 和虚拟地址可以定位到不 同的物理地址（虚拟页号12-31共20位 + ASID 6-11共6位 + 0-5是NULL），下面重点介绍 TLB 缺失处理的过程。

> PTE_ADDR(va) | GET_ENV_ASID(curenv->env_id) 刚好拼成一个entryHi的域。

硬件为我们做了什么?

在发生 TLB 缺失的时候，会把引发 TLB 缺失的虚拟 地址填入到 BadVAddr Register 中，这个寄存器具体的含义请参看 MIPS 手册。接 着触发一个 TLB 缺失异常。

BadVAddr Register

该寄存器保存导致异常的地址。它被设置的情形如下：

- 任何MMU相关的问题；
- 如果用户程序试图访问kuseg之外的地址
- 或者地址对齐错误

在发生任何其他异常之后，BadVAddr Register 都是未定义的。特别要注意的是，它不是在总线错误后设置的。如果CPU为64位是，它是64位。

关于地址对齐错误，这里要强调一点：BadVaddr存放的是导致异常的地址，而不是引发异常的指令所存放的地址。

比如说，我有一条lw指令，存在了0x40008624，它想取0x80008526位置的内容，这时候BAD_ADDR存的是0x80008526，EPC存的是0x40008624

我们需要做什么?

从 BadVAddr 寄存器中获取使 TLB 缺失的虚拟地址，接着 拿着这个虚拟地址通过手动查询页表（mContext 所指向的，在一开始它是boot_pgdir，后来又指向进程的页目录），找到这个虚拟地址所对应的二级页表项中的物理页面号，并将这个页面号填入到 PFN 中，也就是 EntryLo 寄存器，填写好之后，tlbwr 指令就会将EntryHi和EntryLo的内容随机填入到具体的某一项 TLB 表项中。

Bonus mCONTEXT轨迹查询

定义

mCONTEXT定义在start.S里

```
.data
            .globl mCONTEXT
mCONTEXT:
            .word 0

            .globl delay
delay:
            .word 0

            .globl tlbra
tlbra:
            .word 0

            .section .data.stk
KERNEL_STACK:
            .space 0x8000


            .text
LEAF(_start)

    .set    mips2
    .set    reorder

    /* Disable interrupts */
    mtc0    zero, CP0_STATUS

    /* Disable watch exception. */
    mtc0    zero, CP0_WATCHLO
    mtc0    zero, CP0_WATCHHI

    /* disable kernel mode cache */
    mfc0    t0, CP0_CONFIG
    and t0, ~0x7
    ori t0, 0x2
    mtc0    t0, CP0_CONFIG

    /* set up stack */
    li  sp, 0x80400000

    li  t0,0x80400000
    sw  t0,mCONTEXT

    /* jump to main */
    jal main
```

```
loop:
    j   loop
    nop
END(_start)
```

初次赋值为boot_pgdir基地址

并在pmap.s里的mips_vm_init()出现

```
extern char end[];
extern int mCONTEXT;

Pde *pgdir;
u_int n;

pgdir = alloc(BY2PG, BY2PG, 1);
printf("to memory %x for struct page directory.\n", freemem);
mCONTEXT = (int)pgdir;  // 这里赋值也是boot_pgdir基地址 0x8040 1000
boot_pgdir = pgdir;
```

赋值为当前进程页目录基地址

最为常用的是在env_asm.s的lcontext宏函数里面出现，这是在env.c的env_run里设置全局变量mCONTEXT为当前进程页目录地址。

```
LEAF(lcontext)
        .extern mCONTEXT
        sw      a0,mCONTEXT
        jr  ra
        nop
END(lcontext)
```

```
/* Step 3: 设置全局变量mCONTEXT为当前进程页目录地址，这个值将在TLB重填时用到 */
lcontext((u_int)(e->env_pgdir));
```

在TLB重填时发挥作用

然后是具体体现 **在TLB重填时发挥作用**——genex.S中的 do_refill 宏函数。

```
NESTED(do_refill,0 , sp)                # 定义do_refill函数
            //li    k1, '?'
            //sb    k1, 0x90000000
##1. 确定此时的一级页表基地址：mCONTEXT 中存储了当前进程一级页表基地址位于 kseg0 的虚拟地
址；
            .extern mCONTEXT            # 外部引入mCONTENT变量
//this "1" is important
1:          //j 1b
            nop
            lw      k1,mCONTEXT         # 将 mCONTENT 的值（当前进程一级页表基地址）存
入 k1
            and     k1,0xfffff000       # 将 k1 的值与运算0xfffff000的结果存入k1(低
12位置零)
##2. 从 BadVaddr 中取出引发 TLB 缺失的虚拟地址，并确定其对应的一级页表偏移量（高 10 位）；
            mfc0    k0,CP0_BADVADDR     # 将 BADVADDR 寄存器中的值（即引发TLB异常的地
址）存入 k0
            srl     k0,20               # 将 k0 中的值右移20位
            and     k0,0xfffffffc       # 将 k0 的值低2位置零
# 此时，k0中存放的就是一级页表偏移量*4的结果，正好是页表项个数*页表项大小4字节，得到了实际上的
地址偏移量
            addu        k0,k1           # 将 k0 + k1 的值存入k0
```

```
##3. 根据一级页表偏移量，从一级页表中取出对应的表项：此时取出的表项由二级页表基地址的物理地址与权限位组成；
            lw      k1,0(k0)            # 将此时k0中地址 存放的内容存入k1
# 此时 k1中就是一级页表项中存放的内容
##4. 判定权限位：若权限位显示该表项无效（无 PTE_V ），则调用 page_out ，随后回到第一步；
            nop
                    move        t0,k1   # k1 -> t0
                    and     t0,0x0200   # PTE_V 0x0200，这里就是检查一级页表项的权限位（有效位）
                    beqz        t0,NOPAGE   # 如果为0，说明一级页表项无效，那么跳转到NOPAGE
                                            # 在NOPAGE中跳回 pageout
            nop
##5. 确定引发 TLB 缺失的虚拟地址对应的二级页表偏移量（中间 10 位），与先前取得的二级页表基地址的物理地址共同确认二级页表项的物理地址；
            and     k1,0xfffff000       # k1 低12位置零
# 此时，k1中存放的就是对应的二级页表基地址的物理地址
            mfc0    k0,CP0_BADVADDR     # 将 BADVADDR 寄存器中的值存入 k0
            srl     k0,10               # 将 k0 中的值右移10位
            and     k0,0xfffffffc       # 将 k0 的值低2位置零
            and     k0,0x00000fff       # 将 k0 的值高20位置零
# 此时，k0中存放的就是二级页表偏移量*4的结果，正好是页表项个数*页表项大小4字节，得到了实际上的地址偏移量

            addu        k0,k1           # 将 k0 + k1 的值存入k0
                                # 此时 k0 存放的就是二级页表项的物理地址
##6. 将二级页表项物理地址转为位于 kseg0 的虚拟地址（高位补 1），随后页表中取出对应的表项：此时取出的表项由物理地址与权限位组成；
            or      k0,0x80000000       # 将 k0 的最高位 置 1
                                # 此时，k0中存放的就是二级页表项的虚拟地址
            lw      k1,0(k0)            # 将此时k0中地址 存放的内容存入k1
##7. 判定权限位：若权限位显示该表项无效（无 PTE_V），则调用 page_out ，随后回到第一步；（PTE_COW 为写时复制权限位，将在 lab4 中用到，此时将所有页表项该位视为 0 即可）
            nop
                    move        t0,k1   # k1 -> t0
                    and     t0,0x0200   # PTE_V 0x0200，这里就是检查二级页表项的权限位（有效位）
                    beqz        t0,NOPAGE   # 如果为0，那么跳转到NOPAGE
                                            # 在NOPAGE中跳回 pageout
            nop
            move        k0,k1   # k1 -> k0
            and     k0,0x1      # PTE_COW 0x0001，这里就是检查二级页表项的权限位（写时复制位）
            beqz        k0,NoCOW    # 如果为0，那么跳转到NOCOW
            nop
##8. 将物理地址存入 EntryLo ，并调用 tlbwr 将此时的 EntryHi 与 EntryLo 写入到 TLB 中（EntryHi 中保留了虚拟地址相关信息）
            and     k1,0xfffffbff   # 将k1（二级页表项）的第10位（即可写位PTE_R 0x0400）置零
                                    # 即现在的二级页表项是 只读
NoCOW:
            mtc0    k1,CP0_ENTRYLO0 # 将k1寄存器的值（即物理页号+权限位）存入 EntryLo
            nop
            tlbwr                   # 调用 tlbwr 将此时的 EntryHi 与 EntryLo 写入到 TLB 中

            j       2f              # 调转到下面的 2 标签处 f 即 forward
            nop
NOPAGE:
//3: j 3b
nop
            mfc0    a0,CP0_BADVADDR     # 将 BADVADDR 寄存器中的值存入 a0
            lw      a1,mCONTEXT         # 将 mCONTEXT 存入 a1
            nop

            sw      ra,tlbra            # 将 ra 寄存器中的值存入 tlbra 这个全局变量
```

```
            # 这里存放的 ra 是 do_refill 函数执行的下一步
            jal     pageout             # 调用 page_out函数，这里是C函数，
                                        # ra被写入返回地址，即 pageout 函数执行的下一
步
                                        # C函数转为汇编后，会在结尾加一个jr ra 跳回来
            nop
//3: j 3b
nop
            lw      ra,tlbra            # 将 tlbra 这个全局变量的值存入 ra 寄存器中
            # 其实也就是将 do_refill 函数执行完以后应该回去的地方保留下来
            nop
            j   1b                      # 跳转回到 do_refill 函数的开始部分 1 标签处
b 即 back
2:          nop

            jr      ra          # 跳转回到 ra，注意这里的 ra 是 do_refill 函数执行的下
一步
            nop
END(do_refill)
```

$Part3$ 体会与感想&指导书反馈

感谢叶助教的博客『MOS』 Introduction - Coekjan

这次主要是上机的题目表意不清导致很遗憾没有做出lab3-1-extra。另外关于课下评测我也指出了存在的问题，包括lab3-1的课下测试数据，以及lab3-1-extra-offline的测评姬异常问题。

总的来说，指导书的质量是有了较为明显的提高，最后的代码解读部分在回看的时候也帮助我理清了TLB缺失异常的流程。但是有些地方还是存在表意模糊的问题。

另外值得肯定的是，思考题和实验结合的比较紧密，lab4更是如此。没有必要像lab2那样要求同学们去看一本又一本的参考资料。这个只有在同学们有余裕，为了搞清楚知识，主动去寻找的时候，才会容易接受。这个是一个附加品，而不是必须要做的任务。如果说课程不依赖手册就不能说清楚，为什么不把相关内容直接放上来呢？像这次的lab3就是很好的例子。把两个重要的寄存器放上来一讲，同学们就能明白代码到底在干嘛了。而且为了了解指导书里没有说明白的一些位数的功能，同学们自然而然就会去找资料去看。

思考题的定位，在lab4里面得到了最好的诠释——有用而不繁重。需要的是和课程理论+知识紧密结合的思考，而不是刻意增加的学习成本。