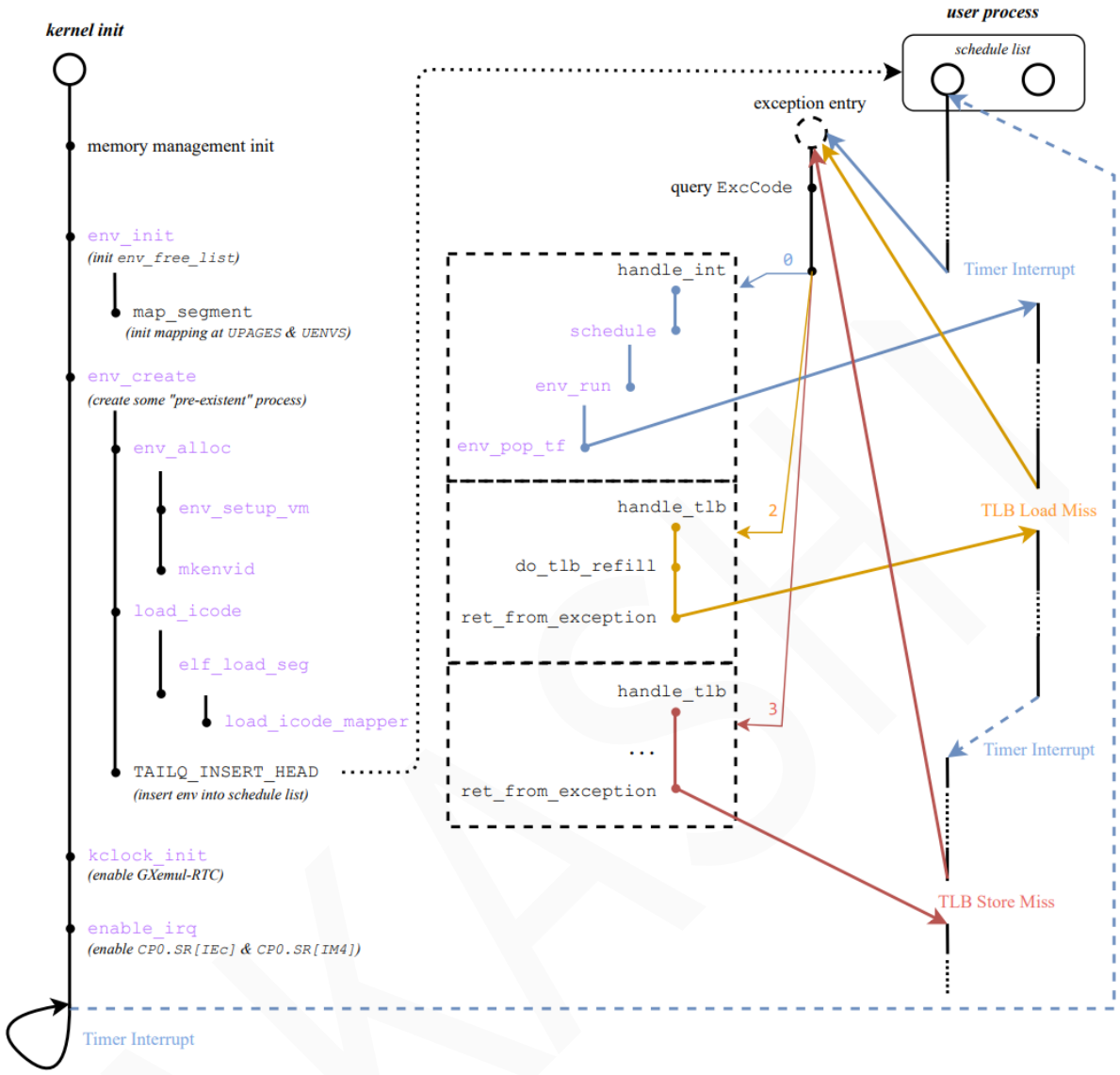
进程管理



进程控制块

定义

进程控制块PCB，也就是 Env 结构体定义：

```
// include/env.h
struct Env {
        struct Trapframe env_tf;  // Saved registers
        LIST_ENTRY(Env) env_link; // Free list
    /*
    低10位：在envs数组中的偏移
    中间1位：内核空间与用户空间区分标记
    高位：进程序号
    */
        u_int env_id;             // Unique environment identifier
```

```
        u_int env_asid;           // ASID
        u_int env_parent_id;      // env_id of this env's parent
    /*
    ENV_FREE : 空PCB
    ENV_NOT_RUNNABLE : 进程阻塞
    ENV_RUNNABLE : 进程就绪
    */
        u_int env_status;         // Status of the environment
        Pde *env_pgdir;           // Kernel virtual address of page dir
        TAILQ_ENTRY(Env) env_sched_link;
        u_int env_pri;
        // Lab 4 IPC
        u_int env_ipc_value;   // data value sent to us
        u_int env_ipc_from;    // envid of the sender
        u_int env_ipc_recving; // env is blocked receiving
        u_int env_ipc_dstva;   // va at which to map received page
        u_int env_ipc_perm;    // perm of page mapping received

        // Lab 4 fault handling
        u_int env_user_tlb_mod_entry; // user tlb mod handler

        // Lab 6 scheduler counts
        u_int env_runs; // number of times been env_run'ed
};


#define LOG2NENV 10
#define NENV (1 << LOG2NENV) // 最大进程数量

// 由env_id获取在envs数组中的偏移
#define ENVX(envid) ((envid) & (NENV - 1))
```

```
// kern/env.c
#define NASID 64 // 同时运行的进程数
```

Trapframe 结构体定义：

```
// include/trap.h

//保存寄存器的值
struct Trapframe {
        /* Saved main processor registers. */
        unsigned long regs[32];

        /* Saved special registers. */
        unsigned long cp0_status;
        unsigned long hi;
        unsigned long lo;
        unsigned long cp0_badvaddr;
        unsigned long cp0_cause;
        unsigned long cp0_epc;
```

```
};

//所有异常的堆栈布局
#define TF_REG0 0
#define TF_REG1 ((TF_REG0) + 4)
//...
#define TF_REG25 ((TF_REG24) + 4)
/*
 * $26 (k0) and $27 (k1) not saved
 */
#define TF_REG26 ((TF_REG25) + 4)
//...
#define TF_REG31 ((TF_REG30) + 4)

#define TF_STATUS ((TF_REG31) + 4)

#define TF_HI ((TF_STATUS) + 4)
#define TF_LO ((TF_HI) + 4)

#define TF_BADVADDR ((TF_LO) + 4)
#define TF_CAUSE ((TF_BADVADDR) + 4)
#define TF_EPC ((TF_CAUSE) + 4)
/*
 * Size of stack frame, word/double word alignment
 */
#define TF_SIZE ((TF_EPC) + 4)
```

寄存器

表 2.1: 通用寄存器的习惯命名和用法

寄存器编号	助记符	用法
0	zero	永远返回 0
1	at	(assembly temporary 汇编暂存)保留给汇编器使用
2-3	v0, v1	子程序返回值
4-7	a0-a3	(arguments) 子程序调用的前几个参数
8-15	t0-t7	(temporaries) 临时变量，子程序使用时无需保存
24-25	t8-t9	
16-23	s0-s7	子程序寄存器变量；子程序写入时必须保存其值并在返回前恢复原值，从而调用函数看到这些寄存器的值没有变化。
26,27	k0,k1	保留给中断或自陷处理程序使用；其值可能在你眼皮底下改变
28	gp	(global pointer)全局指针；一些运行系统维护这个指针以便于存取 `static` 和 `extern` 变量。
29	sp	(stack pointer)堆栈指针
30	s8/fp	第九个寄存器变量；需要的子程序可以用来做帧指针(frame pointer)
31	ra	子程序的返回地址

初始化

```
/* 将指定长度的连续物理地址映射到指定长度的虚地址上
va，pa，size对齐到页
在长度范围内循环调用 page_insert 建立相应映射
*/
static void map_segment(Pde *pgdir, u_int asid, u_long pa, u_long va, u_int size,
u_int perm);

/*
初始化空闲PCB链表env_free_list和调度队列env_sched_list
将envs数组中的PCB插入空闲PCB链表，保证在前的PCB先被使用
申请一个物理页，并关联到模板页目录 base_pgdir
将所有页控制块映射到UPAGES，将所有PCB映射到UENVS
*/
void env_init(void);
```

为新进程申请进程控制块

```
/* 申请env_asid
 从前向后遍历，寻找可用的asid，放置在*asid所指向的位置
 成功返回0，不成功返回-E_NO_FREE_ENV
 */
static int asid_alloc(u_int *asid);
```

```
/* 计算env_id
对于有效的e，返回e所对应的PCB对应的进程唯一id
无明确失败返回值
 */
u_int mkenvid(struct Env *e)

/* 初始化PCB的用户地址空间
申请一个物理页，引用+1，作为进程页目录env_pgdir（内核虚地址）
将base_pgdir中指向UENVS和UPAGES的页目录项拷贝到env_pgdir指向的页目录
将env_pgdir中属于UVPT（长度PDMAP）的页目录项赋值为env_pgdir的物理页框号，这是一个页目录自映射，
页表占UVPT这4M的连续虚拟空间
成功返回0，失败返回-E_NO_MEM
*/
static int env_setup_vm(struct Env *e);

/* 获取并创建一个新的进程PCB，设置其id，parent_id，asid，sp寄存器cp0_status，初始化用户地址空间
检查env_free_list是否为空并申请PCB，为空返回-E_NO_FREE_ENV
尝试调用env_setup_vm设置新PCB的用户地址空间，失败返回-E_NO_MEM
设置env_id，env_parent_id，尝试申请asid，失败返回-E_NO_FREE_ENV
设置cp0_status和sp寄存器（指向用户栈空间，并给argc和argv保留空间）
从空闲链表中删除刚刚申请的PCB
成功返回0
*/
int env_alloc(struct Env **new, u_int parent_id);
```

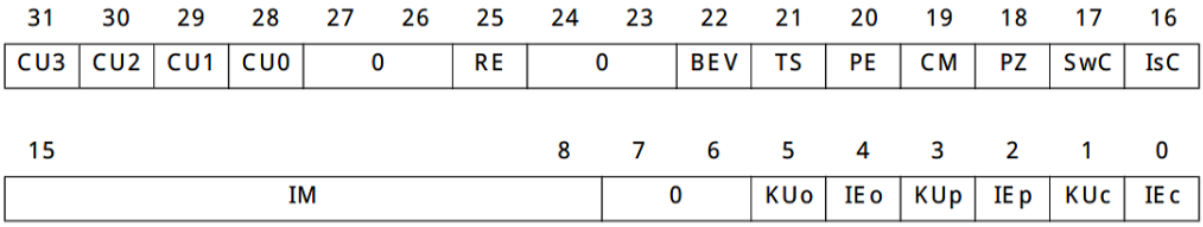
全局变量

```
struct Env envs[NENV] __attribute__((aligned(BY2PG))); // All environments，在.bss段
中

struct Env *curenv = NULL;        // the current env
static struct Env_list env_free_list; // Free list

// Invariant: 'env' in 'env_sched_list' iff. 'env->env_status' is 'RUNNABLE'.
struct Env_sched_list env_sched_list; // Runnable list

static Pde *base_pgdir; // 页目录模板，在 env_init 中申请到的页

static uint32_t asid_bitmap[NASID / 32] = {0}; // asid占用位图
```

SR寄存器

寄存器助记符	CP0 寄存器编号	描述
Status (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 Register



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。

SR 寄存器的低六位是一个二重栈的结构。KUo 和 IEo 是一组，每当异常发生的时候，CPU 自动会将 KUp 和 IEp 的数值拷贝到这里；KUp 和 IEp 是一组。

每当异常发生的时候，CPU 会把 KUc 和 IEc 的数值拷贝到这里。随后将 KUc 和 IEc 置为 0。 即陷入内核时**向高位移动（压栈）**，并将低两位（KUc，IEc）置（0，0），**进入内核态，关中断。**

一组数值表示一种 CPU 的运行状态，其中 KU 位表示是否位于用户模式下，为 1 表示位于 用户模式下；IE 位表示中断是否开启，为 1 表示开启，否则不开启，而 KUc 和 IEc 则为 CPU 当前实际的运行状态。 而每当执行 rfe 指令时，就会进行上述操作的 逆操作，即将 KUo 和 IEo 的值拷贝到 KUp 和 IEp，将 KUp 和 IEp 的值拷贝到 KUc 和 IEc。

每个进程在每一次被调度时都会执行的 rfe（内核态返回用户态） 这条指令。其作用是上述操作的你操作，向地位移动，出栈。

加载镜像

使用的函数

```
使用 load_icode 在运行前一次性将所有数据加载进内存
load_icode ┬ elf_from
           └ elf_load_seg - map_page(load_icode_mapper)
```

```
// lib/elfloader.c
/* 获取ELF头
检测是否是ELF文件，如果是，返回ELF头；如果不是，返回NULL
*/
const Elf32_Ehdr *elf_from(const void *binary, size_t size);

/* 用于将一个 ELF 格式的二进制文件中的一个段加载到内存中，并将它的所有节映射到正确的虚拟地址上
    ph ：指向一个段头表项
    bin ：指向二进制数据
    elf_mapper_t map_page ：自定义回调函数，例如load_icode_mapper，用于加载单个页面
    void *data ：传给回调函数的参数
从段头项中读取虚地址 va ，二进制数据大小 bin_size ，占用内存大小 sgsize ，设置权限有效，如果段可写则加上可写 PET_D
循环调用 map_page 将段数据加载到内存上
成功返回0
```

```
*/
int elf_load_seg(Elf32_Phdr *ph, const void *bin, elf_mapper_t map_page, void
*data);
```

```
// kern/env.c
/* 将可执行文件binary加载到进程e的内存中
调用 elf_from 获取ELF头
调用 elf_load_seg 将ELF文件的每一个 PT_LOAD（需要加载到内存）的段加载到内存
*/
static void load_icode(struct Env *e, const void *binary, size_t size);

/* 自定义回调函数，用于加载单个页面到内存
    data ：回调函数参数，这里是指向PCB（Env结构体）的指针
    va ：目标虚拟地址，不需要对齐到页
    offset ：va 的页内偏移
    perm ：权限
    src ：数据源
    len ：需要加载的长度，到页边界或者数据末尾
申请一个空物理页，并引用加一
如果src不为空，则拷贝数据
将物理页与 va 在 e->env_pgdir 上建立映射
成功返回0
*/
static int load_icode_mapper(void *data, u_long va, size_t offset, u_int perm, const
void *src, size_t len);
```

创建第一个进程

在Lab3中，我们还没有做到创建并运行一个新的进程（通过fork），此处只是创建并运行第一个进程，或许应该是shell

宏定义

```
// include/env.h

// 创建一个 ELF 格式的二进制文件 x 对应的进程，将其优先级设置为 y。该宏的具体实现为：通过 extern
关键字获取二进制文件 x 的起始地址和大小，并调用 env_create 函数创建新进程，传递二进制文件地址、大
小和优先级 y 作为参数。
#define ENV_CREATE_PRIORITY(x, y)


// 创建一个 ELF 格式的二进制文件 x 对应的进程，将其优先级设置为 1。该宏的具体实现类似于
ENV_CREATE_PRIORITY 宏，但省略了 y 参数，使得新进程的优先级默认为 1。
#define ENV_CREATE(x)
```

使用的函数

```
// kern/env.c

/* 用于在初始化过程中从内核创建早期 env，在第一个创建的 env 被调度后弃用
调用 env_alloc 申请空闲PCB并初始化
设置其权限以及执行状态（就绪）
将镜像加载到内存，并将该PCB插入调度队列
*/
struct Env *env_create(const void *binary, size_t size, int priority);
```

进程销毁

```
// kern/env.c

/* 释放指定的进程结构体e以及其使用的所有内存
函数遍历进程的页目录表中的所有页表，找到并释放进程的所有映射过的页面，并释放相应的页表，同时在TLB中清除相应的映射，以此来清除进程使用的所有内存空间
函数释放进程使用的页目录表，并释放相应的ASID，同时在TLB中清除相应的映射
函数将该进程的状态置为ENV_FREE，并将该进程结构体插入空闲列表中，以便下一次使用
*/
void env_free(struct Env *e);

static void asid_free(u_int i) {
    int index = i >> 5;
    int inner = i & 31;
    asid_bitmap[index] &= ~(1 << inner);
}
```

进程调度

首先，MOS进行初始化，在初始化中设置是否接受时钟中断，并设置时钟频率。然后时钟开始运行，向CPU发送时钟中断。当CPU接收到该中断时，跳转到中断分发程序，陷入内核，在内核栈中保存当前寄存器的快照，然后依照CP0_CAUSE中的值，跳转到相应的异常处理程序，此处是通过handle_int调用schedule函数进行调度，选择下一个要运行的进程。再接着调用env_run，将原进程保存在内核栈**底**的寄存器快照，保存在进程控制块中的env_tf中，再调用env_pop_tf，从新进程的env_tf中恢复新进程的寄存器快照，设置sp指向用户栈顶，返回用户态，并将新进程的asid置于CP0_ENTRYHI的相应位置上，然后使用jr返回TF_EPC中的指令地址，回到原来的进程运行。

使用的函数

```
// kern/sched.c

/* 用于进程调度
该函数不返回
```

```
剩余时间片自减
如果 yield 非零，或者当前时间片归零，或者当前无已被调度的进程，或者当前进程不再就绪，则切换进程，否则，仍使用原进程
调用 env_run() 保存并展开寄存器值，转到新进程运行
*/
void schedule(int yield) {
    static int count = 0; // remaining time slices of current env
    struct Env *e = curenv;

    count--;

    if(yield || count <= 0 || e == NULL || e->env_status != ENV_RUNNABLE) {
        if(e != NULL) {
            /*
            此时不确定 env_sched_list 是否需要由 schedule 函数来维护
            即不知道当进程被阻塞是是否需要由该函数将其移出调度队列
            因此使用更加保险的做法
            即判断当前 PCB 是否还在调度队头，如果在，将其移出
            */
            if(e == TAILQ_FIRST(&env_sched_list)) {
                TAILQ_REMOVE(&env_sched_list, e, env_sched_link);
            }
            if(e->env_status == ENV_RUNNABLE) {
                TAILQ_INSERT_TAIL(&env_sched_list, e, env_sched_link);
            }
        }
        if(TAILQ_EMPTY(&env_sched_list)) {
            panic("schedule: no runnable envs\n");
        }
        e = TAILQ_FIRST(&env_sched_list);
        count = e->env_pri;
    }
    env_run(e);
}
```

```
// kern/env.c

/* 将CPU控制权交给e所代表的进程
调用 pre_env_run() 对e进行运行前处理
如果当前 curenv 非空，则从*((struct Trapframe *)KSTACKTOP - 1)保存寄存器的值
将 curenv 更新为 e，并增加其被调度运行次数
调用不会返回的 env_pop_tf(&curenv->env_tf, curenv->env_asid) 展开新进程的寄存器值并跳转运行该函数不会返回
*/
void env_run(struct Env *e);

/* 用于实现特殊的调度机制
如果宏 MOS_SCHED_MAX_TICKS 被定义，则对进程执行的总时钟周期数进行限制。如果进程执行的时钟周期数超过了预设的最大值，则通过 halt() 系统调用使整个操作系统停机。
```

```
如果宏 MOS_SCHED_END_PC 被定义，则当进程的程序计数器（PC）指向了这个特定的地址时，就会执行一些特殊的操作。这个机制可以用于实现某些特殊的调度机制。例如，如果一个进程完成了某个任务，就可以将 PC 设为 MOS_SCHED_END_PC，然后调用 env_destroy() 销毁进程并重新进行调度。
*/
static inline void pre_env_run(struct Env *e);

// 见下汇编定义
extern void env_pop_tf(struct Trapframe *tf, u_int asid) __attribute__((noreturn));
```

```
; kern/env_asm.S
/* 恢复寄存器快照，设置 asid
将进程的地址空间标识符（ASID）写入CP0的EntryHi寄存器中，以便进行进程切换后能够查TLB，并调用 ret_from_exception 将指定进程的 Trapframe 结构体中的信息恢复到处理器寄存器中。当这个函数被调用时，它不会返回，因为它的目的是将控制权转移到新进程中。由于这个函数不会返回，它被标记为 attribute((noreturn))，这个标记告诉编译器这个函数不会返回。
*/
LEAF(env_pop_tf) ; 叶函数，不会再调用其他函数
.set reorder ; 允许指令重排
.set at ;
    sll     a1, a1, 6 ; asid 左移6位
    mtc0    a1, CP0_ENTRYHI ; 将a1的值写入到CP0_ENTRYHI寄存器，以供查TLB时使用
    move    sp, a0 ; 标记为叶函数，sp指针不会被移动，由于a0要被恢复，所以使用sp传参
    j       ret_from_exception ; 调用ret_from_exception函数恢复寄存器现场
END(env_pop_tf)
```

```
; kern/genex.S
; 恢复寄存器现场
FEXPORT(ret_from_exception)
    RESTORE_SOME ; 从内核栈中恢复寄存器快照，定义在include/stackframe.h 中的宏，不恢复 26、27（k0、k1）、29（sp）
    lw      k0, TF_EPC(sp) ;
    lw      sp, TF_REG29(sp) /* Deallocate stack 恢复 sp 寄存器，标志退出内核态*/
.set noreorder
    jr      k0 ; 使用寄存器 k0 中的值作为跳转地址，跳转到之前异常处理过程中被保存的异常返回地址
    rfe ; 内核态返回用户态，同时恢复 CP0 的状态寄存器
.set reorder
```

```
; include/stackframe.h

/* 保存寄存器现场
保存原来的 sp 寄存器，将 sp 至于内核栈顶（如果已经在内核态则跳过这一步），陷入内核
保存CP0_STATUS，CP0_CAUSE，CP0_EPC，CP0_BADVADDR，Hi，Lo
将寄存器快照保存在内核栈顶的一个 struct Trapframe 空间内
*/
.macro SAVE_ALL
.set noreorder
.set noat
    move    k0, sp
.set reorder
```

```
    bltz    sp, 1f ; 如果 sp 值小于零（表示最高位为1，已经在内核态中）则跳过将 sp 放置在内核栈顶，直接分配新的栈空间
    li      sp, KSTACKTOP
.set noreorder
1:
    subu    sp, sp, TF_SIZE
    sw      k0, TF_REG29(sp)
    mfc0    k0, CP0_STATUS
    sw      k0, TF_STATUS(sp)
    ; ... 保存除了sp的各种寄存器
    sw      $31, TF_REG31(sp)
.set at
.set reorder
.endm

/*
恢复 CP0_STATUS，Hi，Lo，CP0_EPC
恢复除了26，27，29（sp）以外的其它寄存器
*/
.macro RESTORE_SOME
.set noreorder
.set noat
    lw      v0, TF_STATUS(sp)
    mtc0    v0, CP0_STATUS
    ; 恢复寄存器
    lw      $1, TF_REG1(sp)
.set at
.set reorder
.endm
```

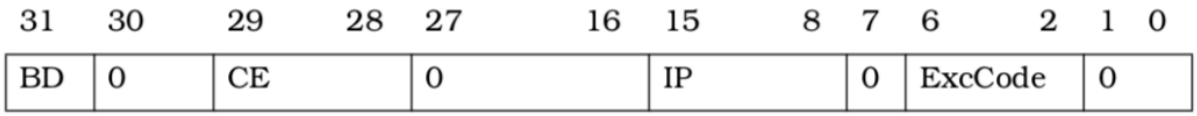
中断与异常

这个过程大概是。中断的同时修改SR寄存器，标记进入内核态并关闭中断，跳转到 exc_gen_entry 进行中断分发，该函数在内核栈上分配空间保存寄存器快照，读取Cause中的中断原因并作为参数传递给处理函数（handle_*）。后者是相应处理函数（do_*）的包装，在该函数中，将指向寄存器快照的指针（sp的值）作为参数保存在 $a0 中，并在栈上保留两个4字节的临时存储空间，然后跳转到相应的处理函数中，待处理函数返回重新将 $sp 指向寄存器快照，然后调用 ret_from_exception 从中断返回。

并不是全部的中断都走这条路，例如时钟中断的处理，在 handle_int 中调用 schedule -> env_run -> env_pop_tf -> ret_from_exception 走这条路回到用户态。通过 env_pop_tf 的主要原因是要设置 asid 以供TLB使用。

Cause寄存器

Cause Register



31	30	29 28	27 16	15 8	7	6 2	1 0
BD	0	CE	0	IP	0	ExcCode	0

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 寄存器中相同位允许中断（为 1）时，Cause 寄存器这一位活动就会导致中断。6-2 位（ExcCode），记录发生了什么异常。

宏定义

```
// include/asm/cp0regdef.h

#define STATUS_CU0 0x10000000
#define STATUS_IM4 0x1000
#define STATUS_KUp 0x8
#define STATUS_IEp 0x4
#define STATUS_KUc 0x2
#define STATUS_IEc 0x1
```

函数

```
; kern/entry.S
; 异常分发函数，在0x80000080
exc_gen_entry:
    SAVE_ALL ; 移动sp，在内核栈上分配空间保存寄存器快照
    mfc0 t0, CP0_CAUSE
    andi t0, 0x7c
    lw t0, exception_handlers(t0)
    jr t0
```

链接

.text.exc_gen_entry 段和 .text.tlb_miss_entry 段需要被链接器放到特定的位置。在 R3000 中，这两个段分别要求放到地址 0x80000080 和 0x80000000 处，它们是异常处理程序的入口地址。在我们的系统中，CPU 发生异常（除了用户态地址的 TLB Miss 异常）后，就会自动跳转到地址 0x80000080 处；发生用户态地址的 TLB Miss 异常时，会自动跳转到地址 0x80000000 处。开始执行。

异常向量组

```
// kern/traps.c

// 异常处理函数指针数组，是对相应 do_* 函数的包装
void (*exception_handlers[32])(void) = {
    [0 ... 31] = handle_reserved, // 保留
    [0] = handle_int, // 时钟中断、控制台中断，定义在 kern/genex.S 中
    [2 ... 3] = handle_tlb, // TLB load 和 store 异常
#if !defined(LAB) || LAB >= 4
    [1] = handle_mod, // 存储异常，存储时该页被标记为只读（写时复制）
    [8] = handle_sys, // 系统调用，用户使用 syscall 陷入内核
#endif
};
```

中断相关函数

```
; kern/genex.S

; 启用中断
LEAF(enable_irq)
    li      t0, (STATUS_CU0 | STATUS_IM4 | STATUS_IEc) ; 将t0寄存器设置为一个特定的值，这个值表示要开启协处理器0中断、定时器中断、以及全局中断（内核模式下是否开启中断）。
    mtc0    t0, CP0_STATUS ; 将t0寄存器的值写入协处理器0的状态寄存器中，以使中断得到启用
    jr      ra ; 跳转回函数调用的地方
END(enable_irq)


/*
这是一个中断处理函数的定义。函数名为 handle_int，接收一个 struct Trapframe 类型的指针作为参数。
NESTED 宏是一个宏，它将当前的汇编语句作为函数的前导程序，生成必要的栈帧和恢复信息。
该函数首先从协处理器 0 中的 CP0_CAUSE 寄存器中获取中断原因码，并将其与当前的状态寄存器中的状态位进行按位与运算。接着将获取的结果与一个被屏蔽的中断位 STATUS_IM4 进行按位与操作，若结果非零，则表示此时的中断是时钟中断。
在时钟中断处理时，该函数会在设备 RTC（Real-Time Clock）的地址上写入一个值，以此来确认中断，并将 a0 赋值为 0。最后，函数调用 schedule 函数进行进程调度。

时钟中断是0号异常，但是是4号中断
*/
NESTED(handle_int, TF_SIZE, zero)
    mfc0    t0, CP0_CAUSE
    mfc0    t2, CP0_STATUS
    and     t0, t2 ; 排除相应中断被屏蔽的情况
    andi    t1, t0, STATUS_IM4 ; 与四号中断位，判断是否是时钟中断
    bnez    t1, timer_irq ; 转到中断服务函数
    // TODO: handle other irqs
timer_irq:
    sw      zero, (KSEG1 | DEV_RTC_ADDRESS | DEV_RTC_INTERRUPT_ACK) ; 清除RTC（Real-time clock）中断信号的挂起，以便下一个中断能够正确触发
    li      a0, 0 ; schedule 的参数 yield，非零时表明强制发生切换
    j       schedule ; 调用 schedule 函数，进行进程的调度
END(handle_int)
```

异常处理宏定义

```
; kern/genex.S

/* 通过异常名以及其处理函数定义异常处理函数调用器
将 sp 的值保存到 a0
调用相应的处理函数
将 sp 的值作为参数传递给 \handler，也许能留到 ret_from_exception 函数
*/
.macro BUILD_HANDLER exception handler
NESTED(handle_\exception, TF_SIZE, zero)
    move    a0, sp
    jal     \handler
    j       ret_from_exception ; 调用从中断返回的函数
END(handle_\exception)
.endm

# 该宏定义在Lab4之中做了修改
.macro BUILD_HANDLER exception handler
NESTED(handle_\exception, TF_SIZE + 8, zero)
    move    a0, sp
    addiu   sp, sp, -8 # 在栈上保留两个4字节的临时存储空间
    jal     \handler
    addiu   sp, sp, 8
    j       ret_from_exception
END(handle_\exception)
.endm
```