Lab3

Thinking

Thinking 3.1

`env_setup_vm`函数第三部分：

```
e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_V;
```

- 解释：在自映射中，自映射页目录项相对于页目录的位置，等于页目录相对二级页表的位置，等于二级页表相对整个空间的位置。
  因此：`PDX(UVPT)`为二级页表相对整个进程空间的一级偏移量，因此：`e->env_pgdir[PDX(UVPT)]`为自映射页目录项；
  `PADDR(e->env_pgdir)`求出页目录的基地址。
  该语句含义为：自映射页目录项填入：物理页号（页目录的基地址）｜权限位

补充：
`MOS`中采用页目录自映射：

- 页表：一个页表项(`4B`)映射一个页(`4KB`)
- 两级页表中，对于一个进程的`4GB`地址空间，需要`4MB`存放页表(1024个页表)，`4KB`来存放页目录；
- 一个页表映射`1K*4KB=4MB`的空间；在1024个页表中，存在一个页表对应的`4MB`空间，就是1024个页表占用的`4MB`空间。该页表为页目录。

`include/mmu.h`：

```
//一个页目录对应的空间:4MB
#define PDMAP (4 * 1024 * 1024) // bytes mapped by a page directory entry
#define ULIM 0x80000000
#define UVPT (ULIM - PDMAP)

/*
o      ULIM     -----> +----------------------+------------0x8000 0000-------
o                      |         User VPT           |     PDMAP                /|\
o      UVPT     -----> +----------------------+------------0x7fc0 0000    |
*/
```

Thinking A.1

64位系统采用三级页表，页面大小4KB，字长8B，页目录有512项，则每级页表需要9位。
总共 `3*9+12=39` 位实现三级页表机制。
可映射至 `2^9*2^9*2^9*4KB=512GB` 的空间。
若三级页表的基地址为$PTbase$,有$PTbase$满足 `2M` 对齐，即低 `21` 位全为0，计算：

- 三级页表页目录的基地址：
  $PD_{base} = PT_{base} + (PT_{base} >> 12) * 8 = PT_{base} | (PT_{base} >> 9)$
- 自映射的页目录表项：
  $PDE_{selfmapping} = PD_{base} | (PT_{base} >> 9) | (PT_{base} >> 18)$

Thinking 3.2

- `elf_load_seg` 函数的 `data` 来源：
  调用者： `load_icode(struct Env *e, const void *binary, size_t size);` ,加载可执行文件binary至进程e内存中.
  `load_icode(...)` 调用 `elf_load_seg(...)` :
  `load_icode_mapper` 为回调函数，进程块 `e` 为传给回调函数的额外参数。
  因此， `data` 来源为进程块 `e` .

- 必要性：传入回调函数 `load_icode_mapper` 的参数 `e` 代表当前进程，需要将当前加载的页面插入该进程的页目录中.

Thinking 3.3

`elf_load_seg` 函数的实现：

```
int elf_load_seg(Elf32_Phdr *ph, const void *bin, elf_mapper_t map_page, void
*data) {
        //从程序段头部ph,提取:虚拟地址va,文件大小bin_size,内存大小sgsize
```

```
        u_long va = ph->p_vaddr;
        size_t bin_size = ph->p_filesz;
        size_t sgsize = ph->p_memsz;
        /*1.设置页面权限:
            设置为有效位PTE_V;如果程序可写(PF_W),添加脏位(PTE_D)*/
        u_int perm = PTE_V;
        if (ph->p_flags & PF_W) {
                perm |= PTE_D;
        }

        int r;
        size_t i;
        /*2.处理非页面对齐的虚拟地址:
            计算偏移量（offset），只映射至页面结束的位置，而非整个页面*/
        u_long offset = va - ROUNDDOWN(va, PAGE_SIZE);
        if (offset != 0) {
                if ((r = map_page(data, va, offset, perm, bin,
                                  MIN(bin_size, PAGE_SIZE - offset))) != 0) {
                        return r;
                }
        }

        /*3.程序段内容的完整加载:
            循环映射整个bin_size大小的二进制文件至内存,每次映射一个页面*/
        for (i = offset ? MIN(bin_size, PAGE_SIZE - offset) : 0; i < bin_size; i
+= PAGE_SIZE) {
                if ((r = map_page(data, va + i, 0, perm, bin + i, MIN(bin_size -
i, PAGE_SIZE))) !=0) return r;
        }

        /*4.程序段在内存中的扩展:
            若bin_size<sgsize,分配额外的页面(初始化为0),以达到sgsize*/
        while (i < sgsize) {
                if ((r = map_page(data, va + i, 0, perm, NULL, MIN(sgsize - i,
PAGE_SIZE))) != 0) return r;
                i += PAGE_SIZE;
        }
        /*5.错误处理:
            若map_page()返回错误码，本函数即返回错误码*/
        return 0;
}
```

Thinking 3.4

`e->env_tf.cp0_epc=ehdr->e_entry` 中，储存的是虚拟地址；在 `MMU` 的 MIPS 系统中，当进程恢复执行时，CPU 将此虚拟地址转换为物理地址，并从对应的物理地址处开始执行程序。

- `cp0_epc`:储存异常返回时的PC值。当CPU发生且处理完异常后，从`cp0_epc`寄存器恢复执行。
- `e_entry`:`ELF`文件制定的程序入口点。
  处理完异常后，恢复执行时，应该从 `ELF` 文件的入口点开始执行。

Thinking 3.5

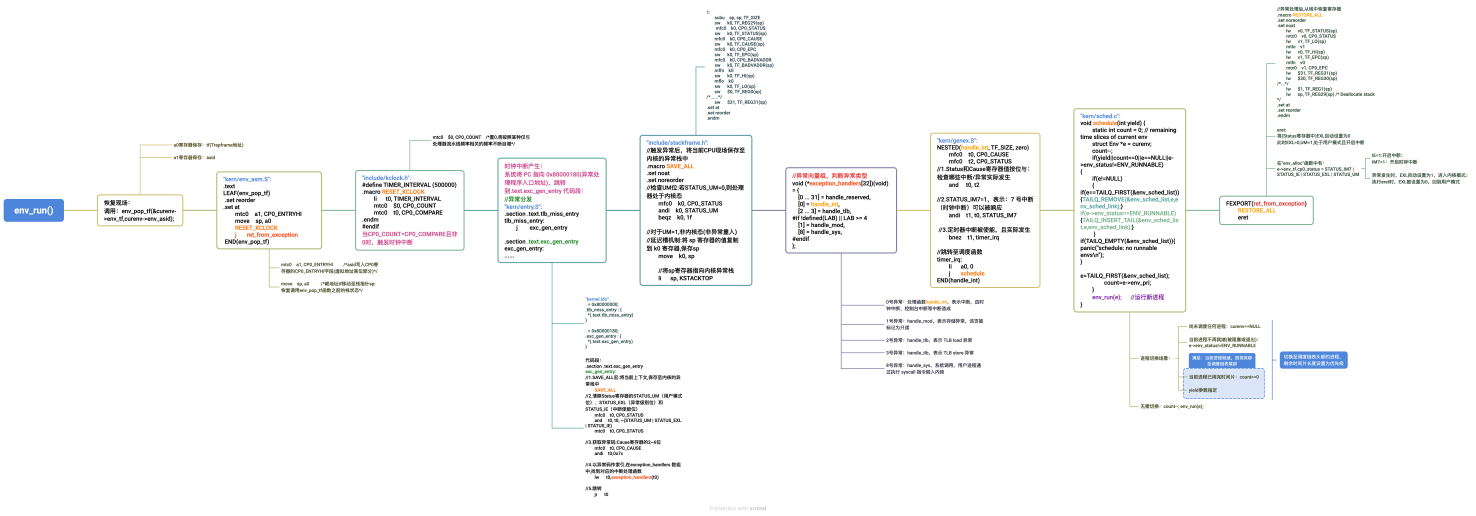
- `Status`寄存器：`15~8`位为中断屏蔽位，每一位代表一个不同的中断活动。
  `15~10`位：使能硬件中断源；`9~8`位：`Cause`寄存器软件可写的中断位。
- `Cause`寄存器：保存CPU中发生的中断/异常。
  `15~10`位：来自硬件；`9~8`位：可由软件写入。
  当 `Status` 寄存器中相同位允许中断（为 1）时，`Cause` 寄存器这一位活动就会导致中断。

0号异常：中断

```
NESTED(handle_int, TF_SIZE, zero)
        mfc0    t0, CP0_CAUSE
        mfc0    t2, CP0_STATUS
        //1.Status和Cause寄存器值按位与：检查哪些中断/异常实际发生
        and     t0, t2
        //2.STATUS_IM7=1，表示：7 号中断（时钟中断）可以被响应
        andi    t1, t0, STATUS_IM7
        //3.定时器中断被使能，且实际发生：
        bnez    t1, timer_irq
timer_irq:
        li      a0, 0
        j       schedule
END(handle_int)
```

1号异常：存储异常

Thinking 3.6, 3.7



Notes

头文件：

`include/env.h`：

```
// Control block of an environment (process).
struct Env {
        //保存进程上下文环境
        struct Trapframe env_tf;          // saved context (registers) before
switching
        //用于构造空闲进程链表env_free_list
        LIST_ENTRY(Env) env_link;         // intrusive entry in 'env_free_list'
        u_int env_id;                     // unique environment identifier
        u_int env_asid;                   // ASID of this env
        //父进程id
        u_int env_parent_id;              // env_id of this env's parent
        //env_status三种取值:ENV_FREE,空闲;ENV_NOT_RUNNABLE,阻塞;ENV_RUNNABLE,运行/就
绪.
        u_int env_status;                 // status of this env
        //页目录基地址
        Pde *env_pgdir;                   // page directory
        //用于构造调度队列env_sched_list
        TAILQ_ENTRY(Env) env_sched_link; // intrusive entry in 'env_sched_list'
        u_int env_pri;                    // schedule priority
 /*...*/
 };

 LIST_HEAD(Env_list, Env);  //Env_list即:指向进程控制块Env的指针
 struct Env_list{
```

```
    struct Env *lh_first;
 }
u_int env_parent_id;            // env_id of this env's parent
```

`include/trap.h`:定义struct Trapframe

```
//保存进程的上下文环境
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;
};

/*
 * Stack layout for all exceptions
 */

#define TF_REG0 0
#define TF_REG1 ((TF_REG0) + 4)
#define TF_REG2 ((TF_REG1) + 4)
#define TF_REG3 ((TF_REG2) + 4)
#define TF_REG4 ((TF_REG3) + 4)
#define TF_REG5 ((TF_REG4) + 4)
#define TF_REG6 ((TF_REG5) + 4)
#define TF_REG7 ((TF_REG6) + 4)
#define TF_REG8 ((TF_REG7) + 4)
#define TF_REG9 ((TF_REG8) + 4)
#define TF_REG10 ((TF_REG9) + 4)
#define TF_REG11 ((TF_REG10) + 4)
#define TF_REG12 ((TF_REG11) + 4)
#define TF_REG13 ((TF_REG12) + 4)
#define TF_REG14 ((TF_REG13) + 4)
#define TF_REG15 ((TF_REG14) + 4)
#define TF_REG16 ((TF_REG15) + 4)
#define TF_REG17 ((TF_REG16) + 4)
#define TF_REG18 ((TF_REG17) + 4)
#define TF_REG19 ((TF_REG18) + 4)
#define TF_REG20 ((TF_REG19) + 4)
#define TF_REG21 ((TF_REG20) + 4)
```

```
#define TF_REG22 ((TF_REG21) + 4)
#define TF_REG23 ((TF_REG22) + 4)
#define TF_REG24 ((TF_REG23) + 4)
#define TF_REG25 ((TF_REG24) + 4)
/*
 * $26 (k0) and $27 (k1) not saved
 */
#define TF_REG26 ((TF_REG25) + 4)
#define TF_REG27 ((TF_REG26) + 4)
#define TF_REG28 ((TF_REG27) + 4)
#define TF_REG29 ((TF_REG28) + 4)
#define TF_REG30 ((TF_REG29) + 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)
#endif /* _TRAP_H_ */
```

include/stackframe.h:

```
//异常处理后,从栈中恢复寄存器
.macro RESTORE_ALL
.set noreorder
.set noat
        lw      v0, TF_STATUS(sp)
        mtc0    v0, CP0_STATUS
        lw      v1, TF_LO(sp)
        mtlo    v1
        lw      v0, TF_HI(sp)
        lw      v1, TF_EPC(sp)
        mthi    v0
        mtc0    v1, CP0_EPC
        lw      $31, TF_REG31(sp)
        lw      $30, TF_REG30(sp)
        lw      $28, TF_REG28(sp)
        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_REG1(sp)
        lw      sp, TF_REG29(sp) /* Deallocate stack */
.set at
.set reorder
.endm
```

include/mmu.h:

```
//一个页目录对应的空间:4MB
//一个页目录包括1024个一级页表项;每个一级页表项对应一个二级页表,即4KB.
#define PDMAP (4 * 1024 * 1024) // bytes mapped by a page directory entry
#define ULIM 0x80000000
#define UVPT (ULIM - PDMAP)
#define UPAGES (UVPT - PDMAP)
#define UENVS (UPAGES - PDMAP)
/*
     4G ----------->  +----------------------+---------0x100000000
                      |       ...                |  kseg2
      KSEG2    -----> +----------------------+---------0xc000 0000
                      |          Devices         |  kseg1
      KSEG1    -----> +----------------------+---------0xa000 0000
                      |      Invalid Memory      |   /|\
                      +----------------------+----|----Physical Memory Max
```

```
                     |       ...                    |  kseg0
      KSTACKTOP-----> +----------------------+----|----0x8040 0000-------end
                     |       Kernel Stack           |    | KSTKSIZE /|\
                     +----------------------+----|---  |
                     |       Kernel Text            |    |   PDMAP
      KERNBASE -----> +----------------------+----|----0x8002 0000  |
                     |      Exception Entry         |   \|/          \|/
      ULIM     -----> +----------------------+---------0x8000 0000----
                     |         User VPT                    PDMAP     /|\
      UVPT     -----> +----------------------+---------0x7fc0 0000  |
                     |           pages              |     PDMAP      |
      UPAGES   -----> +----------------------+---------0x7f80 0000  |
                     |           envs               |     PDMAP      |
  UTOP,UENVS   -----> +----------------------+---------0x7f40 0000  |
  UXSTACKTOP -/      |     user exception stack     |     PTMAP      |
                     +----------------------+---------0x7f3f f000  |
*/
```

include/asm/cp0regdef.h:

Figure 6-12 Status Register Format

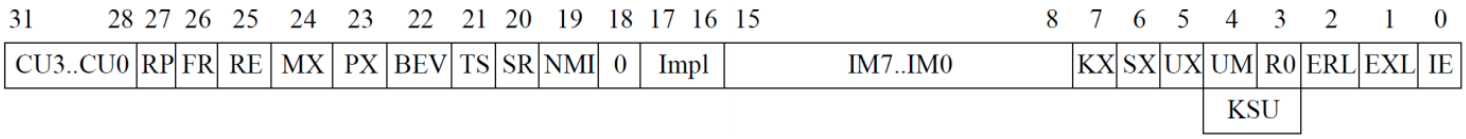


图 3.1: 4Kc 的 Status 寄存器示意图

```c
#define STATUS_IE 0x0001        //1:中断开启，0:未开启
//当且仅当：EXL=0，UM=1时，处理器处于用户模式；其他情况处于内核模式
#define STATUS_EXL 0x0002
#define STATUS_UM 0x0010
//IM7=1，表示：7 号中断（时钟中断）可以被响应
#define STATUS_IM7 0x8000
```

include/elf.h:

```c
//ELF文件头
typedef struct {
// 存放魔数以及其他信息，用于验证ELF文件的有效性
unsigned char e_ident[EI_NIDENT]; /* Magic number and other info */
//程序入口处的虚拟地址
Elf32_Addr e_entry;               /* Entry point virtual address */
```

```
...
// 程序头表所在处与此文件头的偏移
Elf32_Off e_phoff; /* Program header table file offset */
...
// 程序头表表项大小
Elf32_Half e_phentsize; /* Program header table entry size */
// 程序头表表项数
Elf32_Half e_phnum; /* Program header table entry count */
...
} Elf32_Ehdr;

/* Program segment header.
程序段,描述可执行文件和共享库，在加载至内存时的映射关系。用于加载和执行 */

typedef struct {
        Elf32_Word p_type;   /* Segment type */
        Elf32_Off p_offset;  /* Segment file offset:偏移，用于在文件中找到该程序段的位置
*/
        Elf32_Addr p_vaddr;  /* Segment virtual address 虚拟地址，程序加载至内存时，表示
程序段应该被映射>    到的内存地址 */
        Elf32_Addr p_paddr;  /* Segment physical address */
        Elf32_Word p_filesz; /* Segment size in file 程序段在文件中的字节数，读取的数据
量*/
        Elf32_Word p_memsz;  /* Segment size in memory 程序段在内存中的字节数*/
        Elf32_Word p_flags;  /* Segment flags */
        Elf32_Word p_align;  /* Segment alignment */
} Elf32_Phdr;

//遍历ehdr指向的ELF文件，所有程序头表项
#define ELF_FOREACH_PHDR_OFF(ph_off, ehdr)                                  \
        (ph_off) = (ehdr)->e_phoff;                                         \
        for (int _ph_idx = 0; _ph_idx < (ehdr)->e_phnum; ++_ph_idx, (ph_off) +=
(ehdr)->e_phentsize)                                                         \
```

常量:

`kern/env.c`:

```
//env数组:存放进程控制块
struct Env envs[NENV] __attribute__((aligned(PAGE_SIZE))); // All environments
struct Env *curenv = NULL;            //指向当前控制块
static struct Env_list env_free_list; //空闲队列
// Invariant: 'env' in 'env_sched_list' iff. 'env->env_status' is 'RUNNABLE'.
struct Env_sched_list env_sched_list; //调度队列
```

函数:

`kern/env.c`:

u_int mkenvid(struct Env *e);

```
//生成进程标识符env_id
#define LOG2NENV 10
u_int mkenvid(struct Env *e) {
        static u_int i = 0;
        return ((++i) << (1 + LOG2NENV)) | (e - envs);
}
```

static int asid_alloc(u_int *asid);

```
/* Overview:
 *  Allocate an unused ASID.
 *
 * Post-Condition:
 *   return 0 and set '*asid' to the allocated ASID on success.
 *   return -E_NO_FREE_ENV if no ASID is available.
 */
 //分配ASID:(共256个可用的ASID,0~7位表示)
static int asid_alloc(u_int *asid) {
        for (u_int i = 0; i < NASID; ++i) {
                int index = i >> 5;
                int inner = i & 31;     //inner为i的低5位
                //定义:static uint32_t asid_bitmap[NASID / 32] = {0};
                //asid_bitmap每个元素32位,对应32个ASID的分配状态
                if ((asid_bitmap[index] & (1 << inner)) == 0) { //未分配
                        asid_bitmap[index] |= 1 << inner;       //标为已分配
                        *asid = i;
                        return 0;
                }
        }
        return -E_NO_FREE_ENV;
}
```

static void map_segment(Pde *pgdir, u_int asid, u_long pa, u_long va, u_int size, u_int perm);

```
//段映射函数: 将虚拟地址段[va,va+size)映射至物理地址段[pa,pa+size).[按页映射]
static void map_segment(Pde *pgdir, u_int asid, u_long pa, u_long va, u_int size,
u_int perm) {
```

```
        assert(pa % PAGE_SIZE == 0);
        assert(va % PAGE_SIZE == 0);
        assert(size % PAGE_SIZE == 0);

        /* Step 1: Map virtual address space to physical address space. */
        for (int i = 0; i < size; i += PAGE_SIZE) {
        //将pgdir指向的页目录中,虚拟地址va映射至页控制块pp对应的物理页面
        //int page_insert(Pde *pgdir, u_int asid, struct Page *pp, u_long va,
u_int perm);
            struct Page *pp=pa2page(pa+i);
            page_insert(pgdir,asid,pp,va+i,perm | PTE_V);
        }
}
```

void env_init(void);

```
void env_init(void) {
        int i;

        //1.初始化空闲队列env_free_list,调度队列env_sched_list.
        LIST_INIT(&env_free_list);
        TAILQ_INIT(&env_sched_list);

        //2.倒序将envs中进程块,插入env_free_list头部
        for(i=NENV-1;i>=0;i--){
                LIST_INSERT_HEAD(&env_free_list,&envs[i],env_link);
                envs[i].env_status=ENV_FREE;
        }

        //3.
        struct Page *p;
        panic_on(page_alloc(&p));
        p->pp_ref++;
        //为模版页表分配一页物理内存,转为内核虚拟地址base_pgdir
        base_pgdir = (Pde *)page2kva(p);
        //将内核数组pages,envs映射至用户空间的UPAGES,UENVS处
        map_segment(base_pgdir, 0, PADDR(pages), UPAGES,ROUND(npage *
sizeof(struct Page), PAGE_SIZE), PTE_G);
        map_segment(base_pgdir, 0, PADDR(envs), UENVS, ROUND(NENV * sizeof(struct
Env), PAGE_SIZE),PTE_G);
}
```

static int env_setup_vm(struct Env *e);

```
static int env_setup_vm(struct Env *e) {
        //1.为新进程e分配一页作为页目录(将其对应内核虚拟地址写入)
        struct Page *p;
        try(page_alloc(&p));
        p->pp_ref++;
        e->env_pgdir=page2kva(p);

        //2.[UTOP,UVPT)为所有进程共享的只读空间，将该部分对应的内核页表base_pgdir拷贝至进程
页表，使得进程可在用户态访问.
        memcpy(e->env_pgdir + PDX(UTOP), base_pgdir + PDX(UTOP),sizeof(Pde) *
(PDX(UVPT) - PDX(UTOP)));

        //3.自映射页目录项，内容为：物理页号（页目录的基地址）|权限位
        e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_V;
        return 0;
}
```

int env_alloc(struct Env **new, u_int parent_id);

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

        //1.从 env_free_list 中申请一个空PCB 块
        if(LIST_EMPTY(&env_free_list)){
                return -E_NO_FREE_ENV;
        }
        e=LIST_FIRST(&env_free_list);

        //2.初始化新进程的页目录
        try(env_setup_vm(e));

        //3.手工初始化进程块
        /*'env_user_tlb_mod_entry' (lab4), 'env_runs' (lab6), 'env_id' (lab3),
'env_asid' (lab3), 'env_parent_id' (lab3)*/
        e->env_user_tlb_mod_entry = 0; // for lab4
        e->env_runs = 0;               // for lab6
        e->env_id=mkenvid(e);
        if(asid_alloc(&e->env_asid)==-E_NO_FREE_ENV) return -E_NO_FREE_ENV;
        e->env_parent_id=parent_id;

        //异常发生时,EXL自动设置为1;IE为1表示中断开启;
        //当且仅当 EXL 被设置为 0 且 UM 被设置为 1 时，处理器处于用户模式
        e->env_tf.cp0_status = STATUS_IM7 | STATUS_IE | STATUS_EXL | STATUS_UM;
        // Reserve space for 'argc' and 'argv'.
```

```
        e->env_tf.regs[29] = USTACKTOP - sizeof(int) - sizeof(char **);

        //4.从空闲链表摘除
        LIST_REMOVE(e,env_link);

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

static int load_icode_mapper(void *data, u_long va, size_t offset, u_int perm, const void *src, size_t len);

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

        //1.分配空闲页p
        try(page_alloc(&p));
        p->pp_ref++;

        //2.将src开始的len个字节，拷贝至页p的offset处
        if (src != NULL) {
                memcpy((void *)page2kva(p)+offset,src,len);
        }

        //3.将页p插入进程的页目录中：
        return page_insert(env->env_pgdir, env->env_asid, p, va, perm);
}
```

static void load_icode(struct Env *e, const void *binary, size_t size);

```
//将可执行文件binary(size大小)加载至进程e的内存中
static void load_icode(struct Env *e, const void *binary, size_t size) {
        /* Step 1: Use 'elf_from' to parse an ELF header from 'binary'. */
        const Elf32_Ehdr *ehdr = elf_from(binary, size);
        if (!ehdr) {
                panic("bad elf at %x", binary);
        }

        /* Step 2: Load the segments using 'ELF_FOREACH_PHDR_OFF' and
'elf_load_seg'.
         * As a loader, we just care about loadable segments, so parse only
program headers here.*/
        size_t ph_off;
```

```
        //遍历ehdr指向的ELF文件中，每个程序头表项ph.
        ELF_FOREACH_PHDR_OFF (ph_off, ehdr) {
                Elf32_Phdr *ph = (Elf32_Phdr *)(binary + ph_off);
                if (ph->p_type == PT_LOAD) {
                        panic_on(elf_load_seg(ph, binary + ph->p_offset,
load_icode_mapper, e));
                }
        }

        /* Step 3: Set 'e->env_tf.cp0_epc' to 'ehdr->e_entry'. */
        e->env_tf.cp0_epc=ehdr->e_entry;
}
```

创建进程：struct Env *env_create(const void *binary, size_t size, int priority);

```
struct Env *env_create(const void *binary, size_t size, int priority) {
        struct Env *e;
        /* Step 1: Use 'env_alloc' to alloc a new env, with 0 as 'parent_id'. */
        panic_on(env_alloc(&e,0));

        /* Step 2: Assign the 'priority' to 'e' and mark its 'env_status' as
runnable. */
        e->env_pri=priority;
        e->env_status=ENV_RUNNABLE;

        /* Step 3: Use 'load_icode' to load the image from 'binary', and insert
'e' into 'env_sched_list' using 'TAILQ_INSERT_HEAD'. */
        load_icode(e,binary,size);
        //#define TAILQ_INSERT_HEAD(head, elm, field)
        TAILQ_INSERT_HEAD(&env_sched_list,e,env_link);

        return e;
}
```

`void env_run(struct Env *e);`

```
void env_run(struct Env *e){
    assert(e->env_status == ENV_RUNNABLE);
#ifdef MOS_PRE_ENV_RUN
    MOS_PRE_ENV_RUN_STMT
#endif

    //1.当前进程的上下文保存至env_tf中
    if (curenv) {
                curenv->env_tf = *((struct Trapframe *)KSTACKTOP - 1);
```

```
        }
    //2.切换curenv至即将运行的进程e
    curenv = e;
    curenv->env_runs++;

    //3.设置全局变量cur_pgdir为当前进程页目录地址(TLB重填时使用)
    cur_pgdir=curenv->env_pgdir;

    //4.恢复要启动进程的上下文
    env_pop_tf(&curenv->env_tf,curenv->env_asid);
}
```

`lib/elfloader.c`:

const Elf32_Ehdr *elf_from(const void *binary, size_t size);

```
//从给定的二进制数据binary,提取ELF头
const Elf32_Ehdr *elf_from(const void *binary, size_t size) {
        const Elf32_Ehdr *ehdr = (const Elf32_Ehdr *)binary;
        if (size >= sizeof(Elf32_Ehdr) && ehdr->e_ident[EI_MAG0] == ELFMAG0
&&ehdr->e_ident[EI_MAG1] == ELFMAG1 && ehdr->e_ident[EI_MAG2] == ELFMAG2 && ehdr-
>e_ident[EI_MAG3] == ELFMAG3 && ehdr->e_type == 2) {
                return ehdr;
        }
        return NULL;
}
```

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

```
//将一个ELF文件的程序段，加载至内存中：
/*用给定的程序段头部（Elf32_Phdr *ph）和二进制数据（const void *bin），通过map_page函数，和
一个用户数据指针data来执行实际的内存映射。*/
```

异常处理：

`kern/entry.S`:

```
#include <asm/asm.h>
#include <stackframe.h>

.section .text.tlb_miss_entry
tlb_miss_entry:
```

```
        j       exc_gen_entry

.section .text.exc_gen_entry
exc_gen_entry:
        //1.SAVE_ALL宏:将当前上下文,保存至内核的异常栈中
        SAVE_ALL

        //EXL=1或UM=0时,处理器处于内核态.
        //EXL=1时,新异常发生时,EPC的值不更新.
        //我们设置EXL=0,UM=0,IE=0,使得:处于内核态;关闭中断;支持嵌套异常

        //2.清除Status寄存器的STATUS_UM（用户模式位）、STATUS_EXL（异常级别位）和
STATUS_IE（中断使能位）
        mfc0    t0, CP0_STATUS
        and     t0, t0, ~(STATUS_UM | STATUS_EXL | STATUS_IE)
        mtc0    t0, CP0_STATUS

        //3.获取异常码:Cause寄存器的2~6位
        mfc0    t0, CP0_CAUSE
        andi    t0,0x7c

        //4.以异常码作索引,在exception_handlers 数组中,找到对应的中断处理函数
        lw      t0,exception_handlers(t0)

         //5.跳转
         jr      t0
```

`kern/genex.S`:

`handle_int`为异常处理函数，此时时钟中断已开启；`ret_from_exception`完成异常处理并返回。

```
#include <asm/asm.h>
#include <stackframe.h>

.macro BUILD_HANDLER exception handler
NESTED(handle_\exception, TF_SIZE + 8, zero)
        move    a0, sp
        addiu   sp, sp, -8
        jal     \handler
        addiu   sp, sp, 8
        j       ret_from_exception
END(handle_\exception)
.endm

.text
```

```
FEXPORT(ret_from_exception)
        RESTORE_ALL
        eret

NESTED(handle_int, TF_SIZE, zero)
        mfc0    t0, CP0_CAUSE
        mfc0    t2, CP0_STATUS
//1.Status和Cause寄存器值按位与：检查哪些中断/异常实际发生
        and     t0, t2
//2.STATUS_IM7=1，表示：7 号中断（时钟中断）可以被响应
        andi    t1, t0, STATUS_IM7
//3.定时器中断被使能，且实际发生：
        bnez    t1, timer_irq
timer_irq:
        li      a0, 0
        j       schedule
END(handle_int)

BUILD_HANDLER tlb do_tlb_refill

#if !defined(LAB) || LAB >= 4
BUILD_HANDLER mod do_tlb_mod
BUILD_HANDLER sys do_syscall
#endif

BUILD_HANDLER reserved do_reserved
```

`kernel.lds`：

- `.text.exc_gen_entry` 段和 `.text.tlb_miss_entry` 段需要被链接器放到特定的位置

```
/*
 * Set the architecture to mips.
 */
OUTPUT_ARCH(mips)

/*
* Set the ENTRY point of the program to _start.
*/
ENTRY(_start)

SECTIONS {
        /*异常处理的入口地址
            CPU异常:跳转至0x80000180处;用户态地址的TLB Miss异常,跳转至0x80000000处
        */
```

```
        . = 0x80000000;
        .tlb_miss_entry : {*(.text.tlb_miss_entry)}

        . = 0x80000180;
        .exc_gen_entry : {*(.text.exc_gen_entry)}

        /* fill in the correct address of the key sections: text, data, bss. */
        /* Step 1: Set the loading address of the text section to the location
counter ".". */
        . = 0x80020000;

        /* Step 2: Define the text section. */
        .text : { *(.text) }

        /* Step 3: Define the data section. */
        .data : { *(.data) }

        bss_start = .;
        /* Step 4: Define the bss section. */
        .bss : { *(.bss) }
        bss_end = .;
        . = 0x80400000;
        end = . ;
}
```