打印册子

链表操作

Page

```
#define LIST_HEAD(name, type)
    struct name {
        struct type *lh_first; /* first element */
    }
#define LIST_HEAD_INITIALIZER(head)
    { NULL }
#define LIST_ENTRY(type)
    struct {
        struct type *le_next;  /* next element */
        struct type **le_prev; /* address of previous next element */
    }
#define LIST_EMPTY(head) ((head)->lh_first == NULL)

#define LIST_FIRST(head) ((head)->lh_first)

#define LIST_FOREACH(var, head, field)
    for ((var) = LIST_FIRST((head)); (var); (var) = LIST_NEXT((var), field))

#define LIST_INIT(head)
    do {
        LIST_FIRST((head)) = NULL;
    } while (0)

#define LIST_INSERT_AFTER(listelm, elm, field)
        do{
                LIST_NEXT((elm),field) = LIST_NEXT(listelm,field);\
            if(LIST_NEXT(listelm,field)!=NULL)                \
            LIST_NEXT(listelm,field)->field.le_prev =&LIST_NEXT((elm),field);\
            LIST_NEXT(listelm,field) = (elm);\
            (elm)->field.le_prev = &(LIST_NEXT((listelm),field)); \
            }while(0)
```

创建一个名为 name 的链表头，lh_first 指向第一个类型为 type 的链表项

创建用于链接前后链表的指针（pp_link,env_link）

判断链表是否为空

得到链表的第一个元素

初始化链表，将链表头指向的第一个元素设为 NULL

将 elm 插入 listelm 的后面，field 是 pp_link(自己设定)

```
#define LIST_INSERT_BEFORE(listelm, elm, field)
    do {
        (elm)->field.le_prev = (listelm)->field.le_prev;
        LIST_NEXT((elm), field) = (listelm);
        *(listelm)->field.le_prev = (elm);
        (listelm)->field.le_prev = &LIST_NEXT((elm), field);
    } while (0)

#define LIST_INSERT_HEAD(head, elm, field)
    do {
        if ((LIST_NEXT((elm), field) = LIST_FIRST((head))) != NULL)
            LIST_FIRST((head))->field.le_prev = &LIST_NEXT((elm), field);
        LIST_FIRST((head)) = (elm);
        (elm)->field.le_prev = &LIST_FIRST((head));
    } while (0)

#define LIST_NEXT(elm, field) ((elm)->field.le_next)

#define LIST_REMOVE(elm, field)
    do {
        if (LIST_NEXT((elm), field) != NULL)
            LIST_NEXT((elm), field)->field.le_prev = (elm)->field.le_prev;
        *(elm)->field.le_prev = LIST_NEXT((elm), field);
    } while (0)
```

将 elm 插入 listelm 前面

将 elm 插入 head 链表的头部

得到 elm 的下一个元素

将 elm 从链表中删除

Env

```
/*
 * Tail queue definitions.
 */
#define _TAILQ_HEAD(name, type, qual)
        struct name {
                qual type *tqh_first;      /* first element */
                qual type *qual *tqh_last; /* addr of last next element */
        }
#define TAILQ_HEAD(name, type) _TAILQ_HEAD(name, struct type, )

#define TAILQ_HEAD_INITIALIZER(head)
        { NULL, &(head).tqh_first }

#define _TAILQ_ENTRY(type, qual)
        struct {
                qual type *tqe_next;       /* next element */
                qual type *qual *tqe_prev; /* address of previous next element */
        }
#define TAILQ_ENTRY(type) _TAILQ_ENTRY(struct type, )

/*
 * Tail queue functions.
 */
#define TAILQ_INIT(head)
        do {
                (head)->tqh_first = NULL;
                (head)->tqh_last = &(head)->tqh_first;
        } while (/*CONSTCOND*/ 0)
```

```
#define TAILQ_INSERT_HEAD(head, elm, field)
        do {
                if (((elm)->field.tqe_next = (head)->tqh_first) != NULL)
                        (head)->tqh_first->field.tqe_prev = &(elm)->field.tqe_next;
                else
                        (head)->tqh_last = &(elm)->field.tqe_next;
                (head)->tqh_first = (elm);
                (elm)->field.tqe_prev = &(head)->tqh_first;
        } while (/*CONSTCOND*/ 0)

#define TAILQ_INSERT_TAIL(head, elm, field)
        do {
                (elm)->field.tqe_next = NULL;
                (elm)->field.tqe_prev = (head)->tqh_last;
                *(head)->tqh_last = (elm);
                (head)->tqh_last = &(elm)->field.tqe_next;
        } while (/*CONSTCOND*/ 0)

#define TAILQ_INSERT_AFTER(head, listelm, elm, field)
        do {
                if (((elm)->field.tqe_next = (listelm)->field.tqe_next) != NULL)
                        (elm)->field.tqe_next->field.tqe_prev = &(elm)->field.tqe_next;
                else
                        (head)->tqh_last = &(elm)->field.tqe_next;
                (listelm)->field.tqe_next = (elm);
                (elm)->field.tqe_prev = &(listelm)->field.tqe_next;
        } while (/*CONSTCOND*/ 0)

#define TAILQ_INSERT_BEFORE(listelm, elm, field)
        do {
                (elm)->field.tqe_prev = (listelm)->field.tqe_prev;
                (elm)->field.tqe_next = (listelm);
                *(listelm)->field.tqe_prev = (elm);
                (listelm)->field.tqe_prev = &(elm)->field.tqe_next;
        } while (/*CONSTCOND*/ 0)
```

```
#define TAILQ_REMOVE(head, elm, field)                                          \
        do {                                                                    \
                if (((elm)->field.tqe_next) != NULL)                            \
                        (elm)->field.tqe_next->field.tqe_prev = (elm)->field.tqe_prev;  \
                else                                                            \
                        (head)->tqh_last = (elm)->field.tqe_prev;               \
                *(elm)->field.tqe_prev = (elm)->field.tqe_next;                 \
        } while (/*CONSTCOND*/ 0)

#define TAILQ_FOREACH(var, head, field)                                         \
        for ((var) = ((head)->tqh_first); (var); (var) = ((var)->field.tqe_next))

#define TAILQ_FOREACH_REVERSE(var, head, headname, field)                       \
        for ((var) = (*(((struct headname *)((head)->tqh_last))->tqh_last)); (var);     \
             (var) = (*(((struct headname *)((var)->field.tqe_prev))->tqh_last)))

#define TAILQ_CONCAT(head1, head2, field)                                       \
        do {                                                                    \
                if (!TAILQ_EMPTY(head2)) {                                      \
                        *(head1)->tqh_last = (head2)->tqh_first;                \
                        (head2)->tqh_first->field.tqe_prev = (head1)->tqh_last; \
                        (head1)->tqh_last = (head2)->tqh_last;                  \
                        TAILQ_INIT((head2));                                    \
                }                                                               \
        } while (/*CONSTCOND*/ 0)

/*
 * Tail queue access methods.
 */
#define TAILQ_EMPTY(head) ((head)->tqh_first == NULL)
#define TAILQ_FIRST(head) ((head)->tqh_first)
#define TAILQ_NEXT(elm, field) ((elm)->field.tqe_next)

#define TAILQ_LAST(head, headname) (*(((struct headname *)((head)->tqh_last))->tqh_last))
#define TAILQ_PREV(elm, headname, field) (*(((struct headname *)((elm)->field.tqe_prev))->tqh_last))
```

地址转换操作

```
static inline u_long page2ppn(struct Page *pp) {
        return pp - pages;
}

static inline u_long page2pa(struct Page *pp) {
        return page2ppn(pp) << PGSHIFT;
}

static inline struct Page *pa2page(u_long pa) {
        if (PPN(pa) >= npage) {
                panic("pa2page called with invalid pa: %x", pa);
        }
        return &pages[PPN(pa)];
}

static inline u_long page2kva(struct Page *pp) {
        return KADDR(page2pa(pp));
}

static inline u_long va2pa(Pde *pgdir, u_long va) {
        Pte *p;

        pgdir = &pgdir[PDX(va)];
        if (!(*pgdir & PTE_V)) {
                return ~0;
        }
        p = (Pte *)KADDR(PTE_ADDR(*pgdir));
        if (!(p[PTX(va)] & PTE_V)) {
                return ~0;
        }
        return PTE_ADDR(p[PTX(va)]);
}
```

当前物理页和第一个物理页间的偏移量（1..）

页控制块所控制的物理地址

物理地址转化为对应 Page 的地址

返回控制块对应的物理地址所对应的 kseg0 中的虚拟地址

返回页目录基地址 pgdir 所对应页表结构的虚拟地址所映射到的物理地址

PDX()得到一级页表项偏移量

PTX()得到二级页表项偏移量

```
Pde *pgdir_entryp;//一个指向一级页表项的指针
struct Page *pp;//指向Page的指针
/* Step 1: Get the corresponding page directory entry. */
pgdir_entryp = pgdir + PDX(va);//PDX拿出虚拟地址的高10位作为偏移量，pgfir_entryp此时指向对应的一级页表项
 Pte *pgtable;//指向一个二级页表项的指针
 pgtable = (Pte *)KADDR(PTE_ADDR(*pgdir_entryp));
 /* #define PTE_ADDR(pte) ((u_long)(pte) & ~0xFFF)得到二级页表的虚拟地址的基地址，存在pgtable中*/
```

```
#ifndef _MMU_H_
#define _MMU_H_

#include <error.h>

#define BY2PG 4096              // bytes to a page
#define PDMAP (4 * 1024 * 1024) // bytes mapped by a page directory entry
#define PGSHIFT 12
#define PDSHIFT 22 // log2(PDMAP)
#define PDX(va) ((((u_long)(va)) >> 22) & 0x03FF)
#define PTX(va) ((((u_long)(va)) >> 12) & 0x03FF)
#define PTE_ADDR(pte) ((u_long)(pte) & ~0xFFF)

// Page number field of an address
#define PPN(va) (((u_long)(va)) >> 12)
#define VPN(va) (((u_long)(va)) >> 12)
```

PTE_V 有效位，若某页表项的有效位为 1，则该页表项中高 20 位就是对应的物理页号。

PTE_D 可写位，若某页表项的可写位为 1，则可经由该页表项对物理页进行写操作。

PTE_G 全局位，若某页表项的全局位为 1，则 TLB 仅通过虚页号匹配表项，而不匹配 ASID，将在 Lab3 中用于映射 **pages** 和 **envs** 到用户空间。本 Lab 中可以忽略。

PTE_COW 写时复制位，将在 Lab4 中用到，通过该权限位实现了 **fork** 的写时复制机制。本 Lab 中可以忽略。

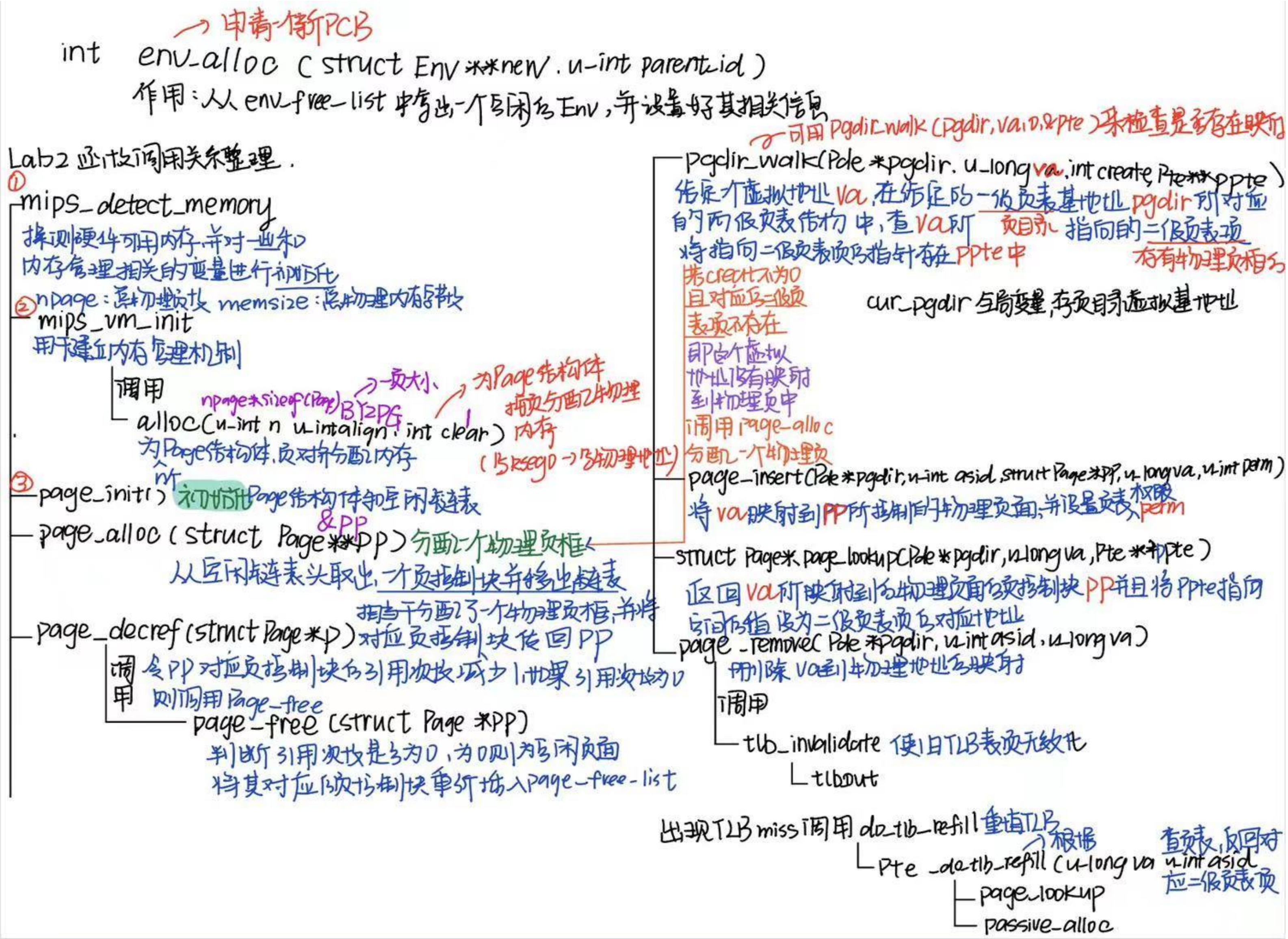
PTE_LIBRARY 共享页面位，将在 Lab6 中用到，用于实现管道机制，本 Lab 中可以忽略。

```
typedef struct {
    unsigned char   e_ident[EI_NIDENT];     /* Magic number and other info */
    // 存放魔数以及其他信息
    Elf32_Half      e_type;                 /* Object file type */
    // 文件类型
    Elf32_Half      e_machine;              /* Architecture */
    // 机器架构
    Elf32_Word      e_version;              /* Object file version */
    // 文件版本
    Elf32_Addr      e_entry;                /* Entry point virtual address */
    // 入口点的虚拟地址
    Elf32_Off       e_phoff;                /* Program header table file offset */
    // 程序头表所在处与此文件头的偏移
    Elf32_Off       e_shoff;                /* Section header table file offset */
    // 节头表所在处与此文件头的偏移
    Elf32_Word      e_flags;                /* Processor-specific flags */
    // 针对处理器的标记
    Elf32_Half      e_ehsize;               /* ELF header size in bytes */
    // ELF 文件头的大小（单位为字节）
    Elf32_Half      e_phentsize;            /* Program header table entry size */
    // 程序头表表项大小
    Elf32_Half      e_phnum;                /* Program header table entry count */
    // 程序头表表项数
    Elf32_Half      e_shentsize;            /* Section header table entry size */
    // 节头表表项大小
    Elf32_Half      e_shnum;                /* Section header table entry count */
    // 节头表表项数
    Elf32_Half      e_shstrndx;             /* Section header string table index */
    // 节头字符串编号
} Elf32_Ehdr;
```

```
typedef struct {
    // segment type
    Elf32_Word p_type;
    // offset from elf file head of this entry
    Elf32_Off  p_offset;
    // virtual addr of this segment
    Elf32_Addr p_vaddr;
    // physical addr, in linux, this value is meanless and has same value of p_vaddr
    Elf32_Addr p_paddr;
    // file size of this segment
    Elf32_Word p_filesz;
    // memory size of this segment
    Elf32_Word p_memsz;
    // segment flag
    Elf32_Word p_flags;
    // alignment
    Elf32_Word p_align;
} Elf32_Phdr;
```

在 ELF 头中，提供了节头表的入口偏移，假设 binary 为 ELF 的文件头地址，shoff 为入口偏移那么 binary + shoff 即为节头表第一项的地址。

Lab2 总结及函数



int env_alloc (struct Env **new, u_int parent_id)
作用：从env_free_list中拿出一个空闲的Env，并设置好其相关信息
Lab2函数调用关系整理
① mips_detect_memory
探测硬件可用内存，并对一些和内存管理相关的变量进行初始化
② npage：总物理页数 memsize：总物理内存字节数
mips_vm_init
用于建立内存管理机制
调用
alloc(u_int n, u_int align, int clear)
为Page结构体、页表等分配内存
③ page_init()
初始化Page结构体和空闲链表
page_alloc (struct Page **pp)
分配一个物理页框
从空闲链表头取出一个页控制块并移出链表
page_decref (struct Page *p)
调用
page_free (struct Page *pp)
判断引用次数是否为0，为0则为空闲页面
将其对应的页控制块重新插入page_free_list
pgdir_walk(Pde *pgdir, u_long va, int create, Pte **ppte)
cur_pgdir 全局变量，存页目录虚拟基地址
page_insert(Pde *pgdir, u_int asid, struct Page *pp, u_long va, u_int perm)
struct Page* page_lookup(Pde *pgdir, u_long va, Pte **ppte)
page_remove(Pde *pgdir, u_int asid, u_long va)
删除va到其物理地址的映射
调用
tlb_invalidate 使旧TLB表项无效化
tlbout
出现TLB miss调用do_tlb_refill重填TLB
Pte _do_tlb_refill (u_long va, u_int asid)
page_lookup
passive_alloc

Lab3 总结及函数

```
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;
};
```

```
struct Env {
    struct Trapframe env_tf; // 保存进程上下文
    LIST_ENTRY(Env) env_link; // 帮助控制空闲链表
    u_int env_id; // envid
    // 状态 ， ENV_FREE,ENV_NOT_RUNNABLE,ENV_RUNNABLE
    u_int env_parent_id; // 父进程 id
    u_int env_status;
    Pde *env_pgdir; // 页目录的在内核中的虚拟地址
    TAILQ_ENTRY(Env) env_sched_link; }// 帮助控制调度队列
    Envs[NENV]数组为存放进程控制块的物理地址
    , env_free_list,env_sched_list
```

env_init()
创建了模板页表 base_pgdir,
将内核数组 pages 和 envs 映射到了用户空间的 UPAGES、UENVS 处,以便用户态可以读取

void map_segment(Pde *pgdir, u_long pa, u_long va, u_long size, u_int perm)
一级页表基地址 pgdir 对应的两级页表结构中做段地址映射，将[va,va+size) 映射到[pa,pa+size)
size 必须是页面大小的整数倍,相关页表项的权限为设置为 perm。

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

env_alloc(struct Env **new, u_int parent_id)
申请一个空闲进程控制块,由 new 传出去

env_setup_vm(struct Env *e)
初始化新进程的地址空间，
PCB 中写入分配的页目录基地址，将模板页表中的 UTOP 和 UVPT 映射到当前页目录 UTOP 往上到 UVPT 之间所有进程共享的只读空间，也就是把这部分内存对应的内核页表 base_pgdir 拷贝到进程页表中。从 UVPT 往上到 ULIM 之间则是进程自己的页表。

load_icode(struct Env *e, const void *binary, size_t size);
将程序 binary 加载到对应的虚拟地址，一边加载一边和物理页面映射

const Elf32_Ehdr *elf_from(const void *binary, size_t size);
解析 elf 文件头,从 elf 文件解析出每个段头 ph,以及其数据在内存中的起始位置 bin

int elf_load_seg(Elf32_Phdr *ph, const void *bin, elf_mapper_t map_page, void *data);
将 ELF 文件的一个 segment 加载到内存 map_page 是我们自定义的回调函数，data 是传给回调函数的，此处传的是 struct env，从 ph 中获取 va（该段需要被加载到的虚地址）、sgsize（该段在内 存中的大小）、bin_size（该段在文件中的大小）和 perm（该段被加载时的页面权限）。

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

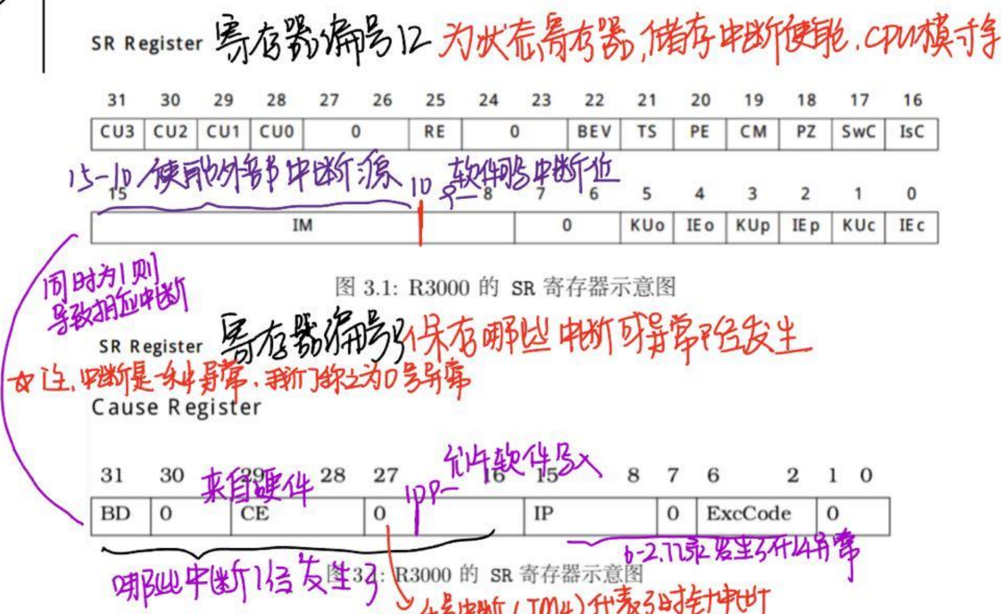
Schedule
时钟中断时调用

env_run(struct Env *e)保存当前进程上下文，恢复要启动的进程上下文，运行该进程 curven=e
cur_pgdir = curenv->env_pgdir

中断与异常详解

发生异常时，进入分发异常程序 entry.S — SAVE_All 保存上下文到异常栈

检测发生了哪个异常，并调用相应的异常处理程序



每当发生时钟中断后：PC跳到 0x80000080，开始执行 .text.exc_gen_entry，进行异常分发 → 调用 handle_int 处理时钟中断，若为IM4，调用 timer_irq，跳转到 schedule 进行进程调度

设置 EPC 指向从异常返回的地址。
设置 SR 位，强制 CPU 进入内核态（行使更高级的特权）并禁止中断。 设置 Cause 寄存器，用于记录异常发生的原因。
CPU 开始从异常入口位置取指，此后一切交给软件处理。

kern/entry.S
异常分发代码
使用 SAVE_ALL 宏将当前上下文保存到内核的异常栈中。
将 Cause 寄存器的内容拷贝到 t0 寄存器中。
取得 Cause 寄存器中的 2~6 位，也就是对应的异常码，这是区别不同异常的重要标志。
以得到的异常码作为索引在 exception_handlers 数组中找到对应的中断处理函数，后文中会有涉及。
跳转到对应的中断处理函数中，从而响应了异常，并将异常交给了对应的异常处理函数去处理。

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，表示 TLB load 异常
3 号异常 的处理函数为 handle_tlb，表示 TLB store 异常
8 号异常 的处理函数为 handle_sys，表示系统调用，用户进程通过执行 syscall 指令陷入内核
0 号异常，在中段处理函数中，进一步判断 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