JOS 实验三实验记录
作者：卓达城
指导老师：邵志远
单位：华中科技大学集群网络与服务计算实验室

首先为 envs 分配内存空间，然后映射到物理地址上。
具体实现如下：

```
        // Make 'envs' point to an array of size 'NENV' of 'st
ruct Env'.
        // LAB 3: Your code here.
        envs = boot_alloc(NENV * sizeof( struct Env), PGSIZE);
```

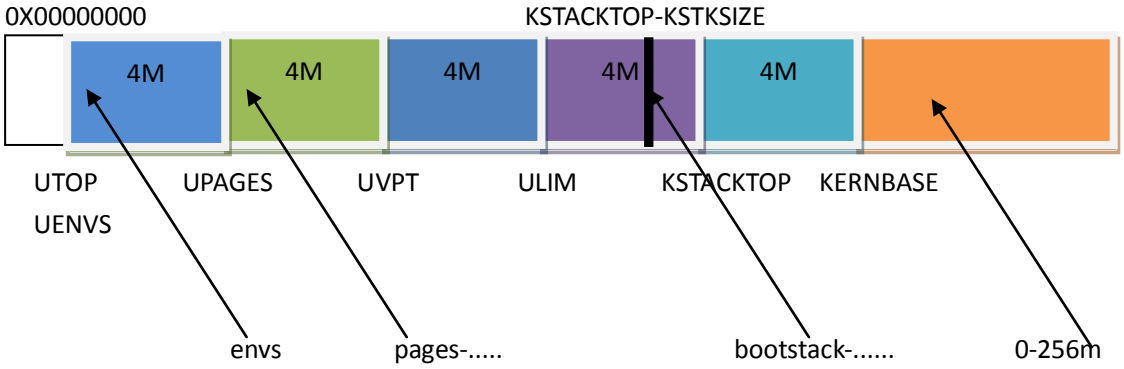
kmap.c vm_init() 分配空间

```
r address UENVS
        // (ie. perm = PTE_U | PTE_P).
        // Permissions:
        //    - envs itself -- kernel RW, user NONE
        //    - the image of envs mapped at UENVS  -- kernel R
, user R
        boot_map_segment(pgdir, UENVS, ROUNDUP( NENV * sizeof(
struct Env), PGSIZE), PADDR(envs),PTE_U | PTE_P);
```

kmap.c vm_init() 映射地址

这两个函数以后，内存的布局如下图所示。

线性地址：



然后现在初始化环境空闲链表，我们想要修改 env_init 函数，这里有一点要注意，应该把 envs[0]放在链表的头部，方便下面调用。具体代码如下：

```
void
env_init(void)
{
        // LAB 3: Your code here.
        struct Env *env;
        int i;
        for(i = NENV -1; i >= 0; i--){
                envs[i].env_id = i;
                LIST_INSERT_HEAD(&env_free_list, &envs[i], env
_link);
//              env =(struct Env *) LIST_FIRST(&env_free_list)
;
//              cprintf("%d\n",env->env_id);
//              LIST_REMOVE( &envs[i], env_link);
        }
//      LIST_REMOVE( &envs[0], env_link);
        return;
}
```

按照实验的要求，现在开始补充 env_setup_vm 函数：
这个函数的主要作用是为新的环境设置页目录，请注意，这里每个新的环境都有一个属于自己的页目录，当设置好页目录之后，把内核的页目录（UTOP 上面的线性地址）映射到新环境的页目录，以便新环境可以通过某些形式访问内核。

```
        // LAB 3: Your code here.
        memset(page2kva(p), 0, PGSIZE);
        e->env_pgdir = page2kva(p);
//      cprintf("%x---%x",(unsigned int)p,(unsigned int)page2k
va(p));
//      memset(p,0,PGSIZE);
//      e->env_pgdir = (pde_t *)p;
        e->env_cr3 = page2pa(p);
        p->pp_ref++;

        for( i = PDX(UTOP); i < NPDENTRIES; i++){
                e->env_pgdir[i] = boot_pgdir[i];
        }
        // VPT and UVPT map the env's own page table, with
        // different permissions.
        e->env_pgdir[PDX(VPT)]  = e->env_cr3 | PTE_P | PTE_W;
        e->env_pgdir[PDX(UVPT)] = e->env_cr3 | PTE_P | PTE_U;
        cprintf("env_setup_vm end!!\n");
        return 0;
}
```

然后是 segment_alloc()函数
这个函数的主要作用是从地址 va 开始分配 len 字节的空间
具体实现代码如下：

```
static void
segment_alloc(struct Env *e, void *va, size_t len)
{
	// LAB 3: Your code here.
	// (But only if you need it for load_icode.)
	//
	// Hint: It is easier to use segment_alloc if the caller can pass
	//   'va' and 'len' values that are not page-aligned.
	//   You should round va down, and round len up.
	int i;
	struct Page *pg;
	va = ROUNDDOWN(va,PGSIZE);
	for(i = 0; i< ROUNDUP(len, PGSIZE)/PGSIZE; i++){
		if( page_alloc(&pg) != 0){
			cprintf("segment_alloc page_alloc fail!!\n");
			return ;
		}
		if( page_insert(e->env_pgdir, pg, va + i * PGSIZE, PTE_U | PTE_W) != 0){
			cprintf("segment_alloc page_insert fail!!!\n");
			return ;
		}
	}
	return;
}
```

load_icode()函数

这个函数是第一部分最难的一个函数，它的主要任务是把程序加载到新环境的虚拟地址上，参数提供了要加载的程序的首地址。由于要在新环境（进程）中加载所以必须把 cr3 换成新环境的 cr3，加载完之后回到原来的 cr3。

然后要明白函数里面循环部分的意义必须明白 elf 的结构，每个应用程序都有自己的 elf 结构，包括内核，所有应用程序都为操作系统提供 elf 结构，然后操作系统根据这个结构把程序加载到合适的地方，elf 的结构如下图：

ELF 文件头的数据结构如下所示：

```
struct Elf {
    uint32_t e_magic;   // 标识文件是否是 ELF 文件
    uint8_t e_elf[12];   // 魔数和相关信息
    uint16_t e_type;     // 文件类型
    uint16_t e_machine;// 针对体系结构
    uint32_t e_version; // 版本信息
    uint32_t e_entry;    // Entry point 程序入口点
    uint32_t e_phoff;    // 程序头表偏移量
    uint32_t e_shoff;    // 节头表偏移量
    uint32_t e_flags;    // 处理器特定标志
    uint16_t e_ehsize;    // 文件头长度
    uint16_t e_phentsize;// 程序头部长度
    uint16_t e_phnum;    // 程序头部个数
    uint16_t e_shentsize;// 节头部长度
    uint16_t e_shnum;    // 节头部个数
    uint16_t e_shstrndx; // 节头部字符索引
};

struct Proghdr {
    uint32_t p_type;     // 段类型
    uint32_t p_offset;   // 段位置相对于文件开始处的偏移量
    uint32_t p_va;       // 段在内存中地址(虚拟地址)
    uint32_t p_pa;       // 段的物理地址
    uint32_t p_filesz;   // 段在文件中的长度
    uint32_t p_memsz; // 段在内存中的长度
    uint32_t p_flags;    // 段标志
    uint32_t p_align;    // 段在内存中的对齐标志
};
```

程序头表把程序分成好几个段，然后段的信息放在 proghdr 中，通过 proghdr 就可以把程序加载到指定的虚拟内存地址上。
具体信息可参照 第三章的书稿 《系统的启动和初始化》
具体实现代码如下：

```
        //  to make sure that the environment starts executing there.
        //  What?  (See env_run() and env_pop_tf() below.)

        // LAB 3: Your code here.
        struct Elf *env_elf;
        struct Proghdr *ph;
        struct Page *pg;
        int i;
        unsigned int old_cr3;
        env_elf = (struct Elf *)binary;
        old_cr3 = rcr3();
        lcr3(PADDR(e->env_pgdir));
        if( env_elf->e_magic != ELF_MAGIC){
                return;
        }
        ph = (struct Proghdr*)((unsigned int)env_elf + env_elf->e_phoff );
        for(i = 0;i < env_elf->e_phnum; i++){
                if( ph->p_type == ELF_PROG_LOAD){
//
                        segment_alloc(e,(void *)ph->p_va, ph->p_memsz);
                        memset((void *)ph-> p_va, 0, ph->p_memsz - ph->p_filesz);
                        cprintf("segment_alloc success !!!\n");

                        memmove((void *)ph->p_va,(void *)((unsigned int )env_elf + ph->p_offset),p
h->p_filesz);

                }
                ph++;
        }
        cprintf("for success!!\n");
        e->env_tf.tf_eip = env_elf->e_entry;
        cprintf("env_elf %x\n",env_elf->e_entry);
        // Now map one page for the program's initial stack
        // at virtual address USTACKTOP - PGSIZE.
```

```
        // LAB 3: Your code here.
        i = page_alloc(&pg);
        if( i != 0){
                cprintf("load_icode page_alloc fail!!\n");
                return;
        }
        page_insert(e->env_pgdir, pg,(void *)(USTACKTOP - PGSIZE),PTE_U | PTE_W);
        lcr3(old_cr3);
}
```

方框部分注意了，这样写比较安全，这个上课听说是 bss 段有可能不全部为零，所以有了这样的写法，具体原因我暂时还不是很清楚。
如果把方框里的前两句调换位置，这个程序就有问题了。+
写完这个函数之后，第一部分就差不多了。

然后进入 env_create 函数
这个函数里面调用了 env_alloc 这个函数我们不用改，但是最好理解。
现在先解释这个函数：

```
        if (!(e = LIST_FIRST(&env_free_list)))
                return -E_NO_FREE_ENV;

        // Allocate and set up the page directory for this environment.
        if ((r = env_setup_vm(e)) < 0)
                return r;
```

从空闲链表拿出一个新的 env，然后初始化。

```
        // Generate an env_id for this environment.
        generation = (e->env_id + (1 << ENVGENSHIFT)) & ~(NENV - 1);
        if (generation <= 0)    // Don't create a negative env_id.
                generation = 1 << ENVGENSHIFT;
        e->env_id = generation | (e - envs);
```

这几句是用来生成 env_id 的，低 10 位势根据 e - envs 定的，NENV 等于 1024，刚好是 10 位，这样就确保每一个 env_id 都不重复。如果前面把 env_id 初始化成零的话，那么(1)env 就会根据 e - envs 来定，具体就是 0-1024，(2)而且在 init.c 里面还有一个就是 env_run(&env[0]),这就是上面的 env_init 为什么要把第零项放第一的原因，其实更重要的是后一个原因，但是第一个原因也是有的（个人认为）。

```
  e->env_tf.tf_ds = GD_UD | 3;
  e->env_tf.tf_es = GD_UD | 3;
  e->env_tf.tf_ss = GD_UD | 3;
  e->env_tf.tf_esp = USTACKTOP;
  e->env_tf.tf_cs = GD_UT | 3;
```

这几句是设置好段寄存器，并把他们的特权级设置为 3，至于 gdt 表在上面时候设置，请看下面（i386_vm_init)：

```
        // Reload all segment registers.
        asm volatile("lgdt gdt_pd");
        asm volatile("movw %%ax,%%gs" :: "a" (GD_UD|3));
        asm volatile("movw %%ax,%%fs" :: "a" (GD_UD|3));
        asm volatile("movw %%ax,%%es" :: "a" (GD_KD));
        asm volatile("movw %%ax,%%ds" :: "a" (GD_KD));
        asm volatile("movw %%ax,%%ss" :: "a" (GD_KD));
        asm volatile("ljmp %0,$1f\n 1:\n" :: "i" (GD_KT));
reload cs
        asm volatile("lldt %%ax" :: "a" (0));
```

gdt 表项在：

```
struct Segdesc gdt[] =
{
        // 0x0 - unused (always faults -- for trapping NULL far pointers)
        SEG_NULL,

        // 0x8 - kernel code segment
        [GD_KT >> 3] = SEG(STA_X | STA_R, 0x0, 0xffffffff, 0),

        // 0x10 - kernel data segment
        [GD_KD >> 3] = SEG(STA_W, 0x0, 0xffffffff, 0),

        // 0x18 - user code segment
        [GD_UT >> 3] = SEG(STA_X | STA_R, 0x0, 0xffffffff, 3),

        // 0x20 - user data segment
        [GD_UD >> 3] = SEG(STA_W, 0x0, 0xffffffff, 3),

        // 0x28 - tss, initialized in idt_init()
        [GD_TSS >> 3] = SEG_NULL
};
```

这里的段基地址都是一样，至于 JOS 是怎样使用段保护的，我暂时也不是很清楚。到后面应该会明白。

然后回到 env_create 函数：
它把应用程序放到指定的虚拟内存上去。然后就完了。具体代码如下：

```
void
env_create(uint8_t *binary, size_t size)
{
        // LAB 3: Your code here.
        struct Env *env;
        cprintf("env_create start!\n");
        if( env_alloc(&env,0) != 0){
                cprintf("env_create env_alloc fail!!\n");
                return;
        }
        cprintf("env_alloc success!!\n");
        load_icode(env,binary,size);
        cprintf("evn_create end\n");
        return ;
}
```

我们再来看这个宏 `ENV_CREATE(user_hello);`
我们一定想知道 user_hello 到底在哪里。这个也是暂时不知道，我这几个文件中都找不到它的信息。补充信息：后来我在 kernel.sym 中找到了 user_hello 的信息。

然后就是 env_run，这个函数很简单，但是它调用的函数 env_pop_tf 比较有意思，先看看 env_run 的具体实现，很简单：

```
        // LAB 3: Your code here.
        curenv = e;
        curenv -> env_runs ++;
        lcr3(curenv->env_cr3);
        cprintf("run the user programme!!\n");
        env_pop_tf(&(curenv -> env_tf));
        // panic("env_run not yet implemented");
}
```

现在进入 env_pop_tf 函数，在看代码之前要先看看 Trapframe 的结构：

```
struct PushRegs {
        /* registers as pushed by pusha */
        uint32_t reg_edi;
        uint32_t reg_esi;
        uint32_t reg_ebp;
        uint32_t reg_oesp;              /* Useless */
        uint32_t reg_ebx;
        uint32_t reg_edx;
        uint32_t reg_ecx;
        uint32_t reg_eax;
};

struct Trapframe {
        struct PushRegs tf_regs;
        uint16_t tf_es;
        uint16_t tf_padding1;
        uint16_t tf_ds;
        uint16_t tf_padding2;
        uint32_t tf_trapno;
        /* below here defined by x86 hardware */
        uint32_t tf_err;
        uintptr_t tf_eip;
        uint16_t tf_cs;
        uint16_t tf_padding3;
        uint32_t tf_eflags;
        /* below here only when crossing rings, such as from user to kernel */
        uintptr_t tf_esp;
        uint16_t tf_ss;
        uint16_t tf_padding4;
};
```

我们可以看到 Trapframe 的前面部分按照一定的顺序，就是 pushal 的顺把 32 为的寄存器排好，然后后面就是段寄存器，tf_padding1 是用来占位用的，因为段寄存器包含的隐藏信息是不能被程序员访问的。

然后我们看下面的代码：

```
env_pop_tf(struct Trapframe *tf)
{
        __asm __volatile("movl %0,%%esp\n"
                "\tpopal\n"
                "\tpopl %%es\n"
                "\tpopl %%ds\n"
                "\taddl $0x8,%%esp\n" /* skip tf_trapno and tf_errcode */
                "\tiret"
                : : "g" (tf) : "memory");
        panic("iret failed");  /* mostly to placate the compiler */
}
```

先把 esp 设置成 tf
然后我们看内存的结构

0x00 0xffffffff

regs	es	ds	trapno	tf_err	eip	cs	

esp

我们走到 esp 是读出数据，在 esp += x 的，然后 iret 是 popl eip 然后 popl cs 的，这样我们就明白了，这个函数是要 cpu 切换到新的环境中，还有就是 cpu 调用 iret 是会切换到用户态的，这个是 cpu 的硬件规定的，但是貌似在进入新

环境之前没有保存内核的运行状态，至于为什么？我现在还不知道，不过我相信以后就会真相大白的。也许是根本就不用保存内核当前的运行状态。

现在进入 PART2
事先声明
要完美的完成 PART2 必须使用 gcc 3.4.3 或者以下的版本。这里包括 G++ 等也有要求。
第一步：
先理解中断的原理：
由于中断，异常什么什么的，谁都说不清，这里只能说是个人观点，不一定正确。
第一：中断时有软件或者硬件发出的。处理完中断之后，程序能继续运行。
例如：
程序一要在屏幕输出一个字符，那么它就必须使用中断，修改 0xB8000 处内存的值，才能实现显示功能，显示完毕之后我们回到我们的应用程序。这是软件中断的例子。硬件中断也是一样，例如键盘中断（一般键盘中断跟其它中断的处理有区别，为了更好的理解，可以想象成程序在等待键盘输入，输入完后，程序继续执行）。
但是为什么我们程序需要用中断，而不直接去改呢？原因很简单，因为 0xb8000 一般由操作系统管理，用户程序不能执行。通过使用中断，操作系统可以调节和管理不同程序的显示，例如两个程序都要在同一个地方显示，那操作系统就会处理这一问题。但是如果不通过中断，两个程序就会冲突了。这里先说明一下，我们现在看到的图形界面不是在 0xb8000 处，而是在 0xa0000 处。

第二：异常，异常出现之后，程序一般停在发生异常的第一，除非经过特殊处理，否则程序不能继续运行。有很多高级语言都可以写异常处理，例如 java。这里不探究。我们可以简单的认为，异常出现后程序停留在原来的位置，不执行。

第三：中断和异常都可以理解成调用一个函数，只不过是中断调用完后运行函数的下一句代码，异常就听到触发异常的代码里头。
如果大家对 call 指令比较熟悉的话就可以理解以下比喻（实模式），中断就是一个 call，异常也是一个 call，不同点在于他们压入的 ip 不同。中断是下一条指令，异常是当前指令。

第四：中断时怎么发生的，先要有一个程序或者硬件引发一个中断，然后 cpu 根据中断号在 IDT 搜索，可以这样理解，IDT 里面放的都是中断函数的符号，cpu 会进行 call，当然跟我们实际的 call 不一样，它的压栈方式不同。

第五：以上纯粹个人理解，不知道是否全对。
先完成函数定义：

```
/*
* Lab 3: Your code here for generating entry points for the different traps.
*/
//      TARPHANDLER_NOEC(divide_error, T_DIVIDE)
//
        TRAPHANDLER_NOEC(divide_error, T_DIVIDE)
        TRAPHANDLER_NOEC(debug, T_DEBUG)
        TRAPHANDLER_NOEC(nmi, T_NMI)
        TRAPHANDLER_NOEC(break_point, T_BRKPT)
        TRAPHANDLER_NOEC(overflow, T_OFLOW)
        TRAPHANDLER_NOEC(bounds, T_BOUND)
        TRAPHANDLER_NOEC(invalid_op, T_ILLOP)
        TRAPHANDLER_NOEC(device_not_available,T_DEVICE)
        TRAPHANDLER_NOEC(double_fault, T_DBLFLT)

        TRAPHANDLER_NOEC(float_point_error, T_FPERR)
        TRAPHANDLER_NOEC(system_call, T_SYSCALL)


        TRAPHANDLER(invalid_TSS, T_TSS)
        TRAPHANDLER(segment_not_present, T_SEGNP)
        TRAPHANDLER(stack_segment,T_STACK)
        TRAPHANDLER(general_protection, T_GPFLT)
        TRAPHANDLER(page_fault, T_PGFLT)
        TRAPHANDLER(alignment_check, T_ALIGN)
        TRAPHANDLER(machine_check, T_MCHK)
        TRAPHANDLER(SIMD_float_point_error, T_SIMDERR)
```

完成_alltraps

```
/*
 * Lab 3: Your code here for _alltraps
 */
_alltraps:
pushl %ds
pushl %es
pushal
movl $GD_KD,%eax
movw %ax,%ds
movw %ax,%es
pushl %esp
call trap
popl %esp
popal
popl %es
popl %ds
iret
```

这里我们来研究研究这个宏：

```
#define TRAPHANDLER_NOEC(name, num)
        .globl name;
        .type name, @function;
        .align 2;
        name:
        pushl $0;
        pushl $(num);
        jmp _alltraps
```

这个宏是什么意思呢？

.globl name 是定义一个函数名为全局都可以引用的符号（symbol 我只能这样翻译了）。

然后 name:表明函数从这里开始，用过汇编就知道，我们可以 call name 的，这里的道理是一样的。

这个文件的作用就是声明所有中断函数，然后发生中断以后都跳入_alltraps 函数，这个函数的作用是按照 Trapframe 结构压栈。

```
*/
#define TRAPHANDLER(name, num)                                          \
        .globl name;            /* define global symbol for 'name' */   \
        .type name, @function;  /* symbol type is function */           \
        .align 2;               /* align function definition */         \
        name:                   /* function starts here */              \
        pushl $(num);                                                   \
        jmp _alltraps
```

对于这个宏，cpu 会自动把 errorcode 压栈，但是为什么可以，cpu 根据什么定义压栈的 errorcode 呢？后面应该会有发现。后来发现 cpu 有些中断号是定死的，例如缺页中断，中断号必须是 14，往往这些中断会自动压入 errorcode。

完成中断映射：

```
extern void divide_error();
extern void debug();
extern void nmi();
extern void break_point();
extern void overflow();
extern void bounds();
extern void invalid_op();
extern void device_not_available();
extern void double_fault();

extern void invalid_TSS();
extern void segment_not_present();
extern void stack_segment();
extern void general_protection();
extern void page_fault();

extern void float_point_error();
extern void alignment_check();
extern void machine_check();
extern void SIMD_float_point_error();

extern void system_call();
```

用 extern 声明一下刚才在 trapency.S 中声明的函数

```
/       extern char coprocessor_segment_overrun[];
        SETGATE(idt[T_DIVIDE], 0, GD_KT, divide_error, 0);
        SETGATE(idt[T_DEBUG], 0, GD_KT, debug, 0);
        SETGATE(idt[T_NMI], 0, GD_KT, nmi, 0);
        SETGATE(idt[T_BRKPT], 0, GD_KT, break_point, 3);
        SETGATE(idt[T_OFLOW], 0, GD_KT, overflow, 0);

        SETGATE(idt[T_BOUND], 0, GD_KT, bounds, 0);
        SETGATE(idt[T_ILLOP], 0, GD_KT, invalid_op, 0);
       // SETGATE(idt[T_DIVICE], 0, GD_KT, device_not_available, 0);
        SETGATE(idt[T_DEVICE], 0, GD_KT, device_not_available, 0);
        SETGATE(idt[T_DBLFLT], 0, GD_KT, double_fault, 0);
/         SETGATE(idt[T_COPROC], 0, GD_KT, coprocessor_segment_overrun, 0);

        SETGATE(idt[T_TSS], 0, GD_KT, invalid_TSS, 0);
        SETGATE(idt[T_SEGNP], 0, GD_KT, segment_not_present, 0);
        SETGATE(idt[T_STACK], 0, GD_KT, stack_segment, 0);
        SETGATE(idt[T_GPFLT], 0, GD_KT, general_protection, 0);
        SETGATE(idt[T_PGFLT], 0, GD_KT, page_fault, 0);



/       extern char reserved[];

/       SETGATE(idt[T_RES], 0, GD_KT, reserved, 0);
        SETGATE(idt[T_FPERR], 0, GD_KT, float_point_error, 0);
        SETGATE(idt[T_ALIGN], 0, GD_KT, alignment_check, 0);
        SETGATE(idt[T_MCHK], 0, GD_KT, machine_check, 0);
        SETGATE(idt[T_SIMDERR], 0, GD_KT, SIMD_float_point_error, 0);
```

```
      SETGATE(idt[T_SYSCALL], 0, GD_KT, system_call, 3);
      // Setup a TSS so that we get the right stack
      // when we trap to the kernel.
      ts.ts_esp0 = KSTACKTOP;
      ts.ts_ss0 = GD_KD;

      // Initialize the TSS field of the gdt.
      gdt[GD_TSS >> 3] = SEG16(STS_T32A, (uint32_t) (&ts),
                               sizeof(struct Taskstate), 0);
      gdt[GD_TSS >> 3].sd_s = 0;

      // Load the TSS
      ltr(GD_TSS);

      // Load the IDT
      asm volatile("lidt idt_pd");
```

中断号映射完之后，我们就可以改 trapdispatch 函数

```
static void
trap_dispatch(struct Trapframe *tf)
{
        // Handle processor exceptions.
        // LAB 3: Your code here.
        cprintf("%d\n",tf->tf_trapno);
        if( tf->tf_trapno == T_SYSCALL){
                tf->tf_regs.reg_eax = syscall( tf->tf_regs.reg_eax, tf->tf_regs.reg_edx, tf->tf_re
gs.reg_ecx, tf->tf_regs.reg_ebx, tf->tf_regs.reg_edi, tf->tf_regs.reg_esi);
                return;
        }
        if( tf->tf_trapno == T_PGFLT){
                page_fault_handler(tf);
                return;
        }
        if(tf->tf_trapno == T_BRKPT){
                monitor(tf);
                return;
        }
        // Unexpected trap: The user process or the kernel has a bug.
        print_trapframe(tf);
        if (tf->tf_cs == GD_KT)
                panic("unhandled trap in kernel");
        else {
                cprintf("mapping interup finished!!\n");
                env_destroy(curenv);
                return;
        }
}
```

现在我们来看看这个函数，trap 函数

```
void
trap(struct Trapframe *tf)
{
        cprintf("Incoming TRAP frame at %p\n", tf);
//      cprintf("tf------\n");
//      print_trapframe(tf);
//      cprintf("env_tf-------\n");
//      print_trapframe(&curenv->env_tf);
        if ((tf->tf_cs & 3) == 3) {
                // Trapped from user mode.
                // Copy trap frame (which is currently on the stack)
                // into 'curenv->env_tf', so that running the environment
                // will restart at the trap point.
                assert(curenv);
                curenv->env_tf = *tf;
//              cprintf("fuck!!\n");
//              print_trapframe(tf);
//              cprintf("kao!!\n");
//              print_trapframe(&curenv->env_tf);
                // The trapframe on the stack should be ignored from here on.
                tf = &curenv->env_tf;
        }

        // Dispatch based on what type of trap occurred
        trap_dispatch(tf);

        // Return to the current environment, which should be runnable.
//        print_trapframe(&curenv->env_tf);
        assert(curenv && curenv->env_status == ENV_RUNNABLE);
        env_run(curenv);
}
```

先检查中断的类型是不是 user 中断
然后分配中断
然后恢复程序当前的运行状态。`env_run(curenv);`

因为

```
// will restart at the trap
assert(curenv);
curenv->env_tf = *tf;
```

这里已经把当前的状态付给了 curenv 环境了。所以可以恢复。

现在特别声明一下：如果用的 GCC 在 4.3.3 以上，则以上代码在 0 地址不会出现缺页中断，必须得换到 4.3.3 或者以下，建议用 Ubuntu9.04 操作系统，其它版本可能不行，例如 10.04。我自己的就不行。

做到这里我们可以修改 init.c 里面的 CREATE_ENV 宏的参数，来运行各种应用程序验证效果。

完成 user_mem_check 函数：

```
int
user_mem_check(struct Env *env, const void *va, size_t len, int perm)
{
	// LAB 3: Your code here.
	int i;
	unsigned int begin;
	begin = (unsigned int)va / PGSIZE * PGSIZE + PGSIZE;
	if( ((unsigned int)*pgdir_walk(env->env_pgdir, va, 0) & (perm | PTE_P)) == (perm | PTE_P)){
		for(;begin < ((unsigned int)va + len) / PGSIZE * PGSIZE; begin += PGSIZE){
			if( ((unsigned int)*pgdir_walk(env->env_pgdir,(void *) begin, 0) & (perm | PTE_P)) == (perm | PTE_P)){
				//NOTHING TO DO
			} else {
				user_mem_check_addr = begin;
				cprintf("fuck 2!!\n");
				return -E_FAULT;
			}
		}
	}else {
		user_mem_check_addr = (unsigned int)va;
		cprintf("%x\n",*pgdir_walk(env->env_pgdir,va,0));
		cprintf("%x\n", PTE_P | PTE_U);
		cprintf("fuck 1!!\n");
		return -E_FAULT;
	}
	return 0;
}
```

这个函数不复杂，就是检查指定内存可不可以用而已。
我写的代码有点难看，敬请原谅。

修改 CREATE_ENV 宏的参数，运行指定测试程序。
到此为止，实验完成。