lab3：User Environment 用户进程

lab3：User Environment 用户进程
- 实验简介
- 实验概括
- 练习部分
  - Exercise 1
  - Exercise 2
    - env_init()
    - env_setup_vm()
    - region_alloc()
    - load_icode()
    - env_create()
    - env_run()
  - Exercise 3
  - Exercise 4
  - Exercise 5
  - Exercise 6
  - Exercise 7
  - Exercise 8
  - Exercise 9
  - Exercise 10
- 遇到的问题

> 组员：李瑞峰 1711347 李汶蔚 1711352 常欢 1711308

完成情况:

- 全部练习

实验简介

1. Enc结构体

```
struct Env {

  struct Trapframe env_tf;  // Saved registers

  struct Env *env_link;     // Next free Env

  envid_t env_id;      // Unique environment identifier

  envid_t env_parent_id;   // env_id of this env's parent

  enum EnvType env_type;   // Indicates special system environments
```

```
  unsigned env_status;    // Status of the environment

  uint32_t env_runs;   // Number of times environment has run

  // Address space

  pde_t *env_pgdir;    // Kernel virtual address of page dir

};
```

2. 下面是用户代码执行过程
    - start (kern/entry.S)
    - i386_init (kern/init.c)
        - cons_init
        - mem_init
        - env_init
        - trap_init (这时候还不完整)
        - env_create
        - env_run
            - env_pop_tf
3. Trapframe结构体

```
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;
} __attribute__((packed));
```

4. 除零异常示例：

除零异常：

```
            +--------------------+ KSTACKTOP    //stack向下增长，留心“-”号

                      | 0x00000 | old SS    |      " - 4
                      |      old ESP        |      " - 8
                      |     old EFLAGS      |      " - 12
                      | 0x00000 | old CS    |      " - 16
                      |      old EIP        |      " - 20 <---- ESP
                      +---------------------+
```

1.处理器切换到由TSS中的SS0(包含GD_KD)与ESP0(包含KSTACKTOP)指向的stack。

2.处理器将old ss、old ESP、异常数据EFLAGS等推入堆栈

3.除零异常的中断向量是0，所以处理器读取IDT条目0并设置'CS:EIP'指向条目0描述的the handler function(处理函数)。

4.处理函数控制和处理这个exception，如结束这个用户环境

对于某些x86异常，除零这种five words "standard",处理器还会把"error code"推入堆栈，如The page fault exception, number 14

```
            +--------------------+ KSTACKTOP
                      | 0x00000 | old SS    |      " - 4
                      |      old ESP        |      " - 8
                      |     old EFLAGS      |      " - 12
                      | 0x00000 | old CS    |      " - 16
                      |      old EIP        |      " - 20
                      |     error code      |      " - 24 <---- ESP
                      +---------------------+
```

5. [Error code Summary][https://pdos.csail.mit.edu/6.828/2016/readings/i386/s09_10.htm]

9.10 Error Code Summary

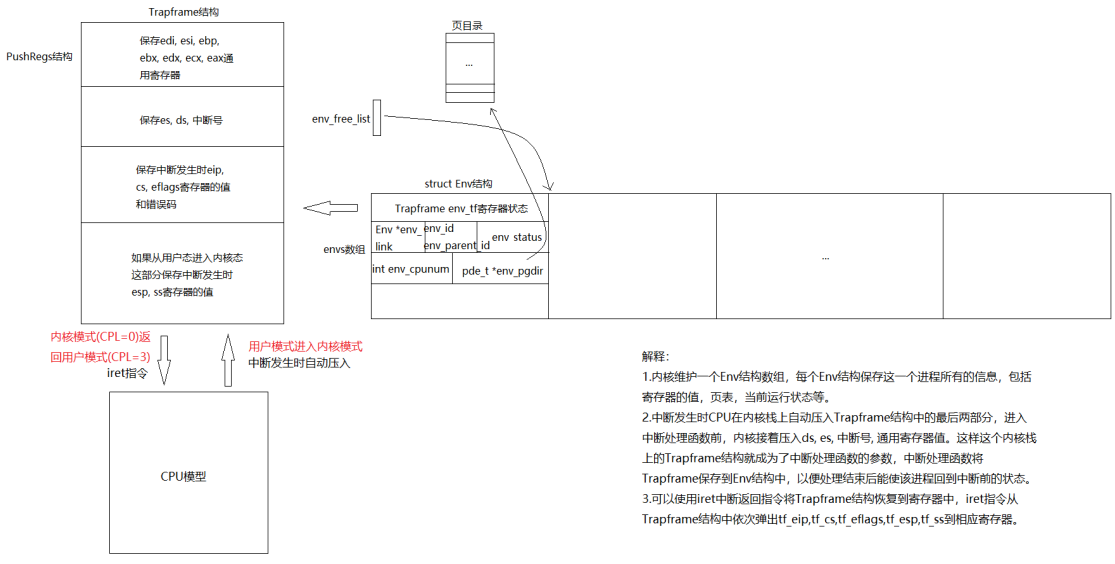
Table 9–7 summarizes the error information that is available with each exception.

Table 9-7. Error-Code Summary

Description	Interrupt Number	Error Code
Divide error	0	No
Debug exceptions	1	No
Breakpoint	3	No
Overflow	4	No
Bounds check	5	No
Invalid opcode	6	No
Coprocessor not available	7	No
System error	8	Yes (always 0)
Coprocessor Segment Overrun	9	No
Invalid TSS	10	Yes
Segment not present	11	Yes
Stack exception	12	Yes
General protection fault	13	Yes
Page fault	14	Yes
Coprocessor error	16	No
Two-byte SW interrupt	0-255	No

实验概括

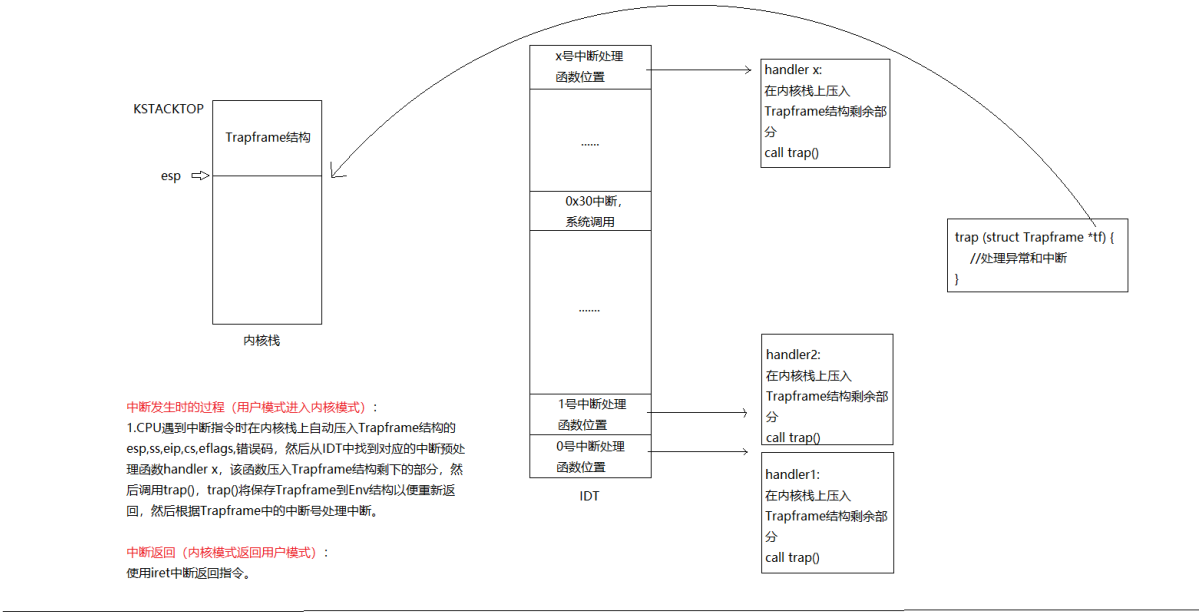
1. 进程建立，可以加载用户ELF文件并执行。
    1. 内核维护一个名叫envs的Env数组，每个Env结构对应一个进程，Env结构最重要的字段有Trapframe env_tf（该字段中断发生时可以保持寄存器的状态），pde_t *env_pgdir（该进程的页目录地址）。进程对应的内核数据结构可以用下图总结：

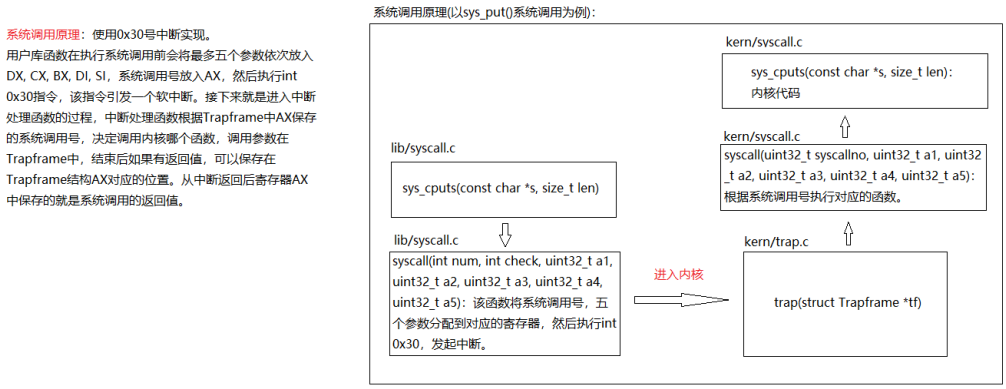


JOS进程抽象

    2. 定义了env_init()，env_create()等函数，初始化Env结构，将Env结构Trapframe env_tf中的寄存器值设置到寄存器中，从而执行该Env。

2. 创建异常处理函数，建立并加载IDT，使JOS能支持中断处理。要能说出中断发生时的详细步骤。需要搞清楚内核态和用户态转换方式：通过中断机制可以从用户环境进入内核态。使用iret指令从内核态回到用户环境。中断发生过程以及中断返回过程和系统调用原理可以总结为下图：





JOS中断以及系统调用过程

3. 利用中断机制，使JOS支持系统调用。要能说出遇到int 0x30这条系统调用指令时发生的详细步骤。见上图。

练习部分

Exercise 1

- 修改 `mem_init()` 来为 `Env` 结构体的数组`envs`，分配内存。
- 将`envs` 的物理内存设置为只读映射在页表中 `UENVS` 的位置，用户进程可以从这一数组读取数据。

```
// Make 'envs' point to an array of size 'NENV' of 'struct Env'.
  // LAB 3: Your code here.
  envs = (struct Env*)boot_alloc(NENV * sizeof(struct Env));
  memset(envs, 0, NENV * sizeof(struct Env));

//////////////////////////////////////////////////////////////////////
```

```
    // Map the 'envs' array read-only by the user at linear address UENVS
    // (ie. perm = PTE_U | PTE_P).
    // Permissions:
    //    - the new image at UENVS  -- kernel R, user R
    //    - envs itself -- kernel RW, user NONE
    // LAB 3: Your code here.
    boot_map_region(kern_pgdir, UENVS, PTSIZE, PADDR(envs), PTE_U);
```

在这里注意到一个细节，struct Env的大小为96个字节，NENV = 1024，算出实际分配的物理内存为98304Byte，即24个页，但是虚拟地址布局中 RO ENVS 区域大小是PTSIZE为4M（1024个页）。

Exercise 2

env_init()

前面已经为进程描述符表分配了内存空间，现在要初始化这些描述符:

- 将所有的描述符的进程id置位0
- 状态置位free
- 然后依次的放入到空闲列表中

注意：反向初始化，到最后就保证env_free_list指向第一个env，而且比正向初始化操作简便

```
void
env_init(void)
{
    // Set up envs array
    // LAB 3: Your code here.
    int i = 0;
    for (i = NENV-1; i >= 0; i--) {
        struct Env *e = &envs[i];
        e->env_id = 0;
        e->env_status = ENV_FREE;
        e->env_link = env_free_list;
        env_free_list = e;
    }
    // Per-CPU part of the initialization
    env_init_percpu();
}
```

env_setup_vm()

作用是为当前的进程分配一个页，用来存放页表目录，同时将内核部分的内存的映射完成

- Hint中提到，所有的进程，不论是内核还是用户，在虚地址UTOP之上的内容都是一样的，所以直接copy即可。
- 因为UVTP是一个特例，所以单独对UVTP进行设置。

```
static int
```

```
env_setup_vm(struct Env *e)
{
    int i;
    struct Page *p = NULL;

    // Allocate a page for the page directory
    if (!(p = page_alloc(ALLOC_ZERO)))
        return -E_NO_MEM;

    // Now, set e->env_pgdir and initialize the page directory.
    // LAB 3: Your code here.
    p->pp_ref++;
    e->env_pgdir = (pde_t *)page2kva(p);
    memmove(e->env_pgdir, kern_pgdir, PGSIZE);

    // UVPT maps the env's own page table read-only.
    // Permissions: kernel R, user R
    e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_P | PTE_U;

    return 0;
}
```

region_alloc()

这个函数将从va开始len字节的虚拟地址空间重新分配和映射到物理页中，更新页目录及二级页表。**注意给定的va不一定是4096对齐的，解决办法是按提示所说将`va` rounddown向下4096对齐，`len` roundup向上4096对齐。**

```
// Allocate len bytes of physical memory for environment env,
// and map it at virtual address va in the environment's address space.
// Does not zero or otherwise initialize the mapped pages in any way.
// Pages should be writable by user and kernel.
// Panic if any allocation attempt fails.
static void
region_alloc(struct Env *e, void *va, size_t len)
{
  // LAB 3: Your code here.
  // (But only if you need it for load_icode.)
  //
  void *begin = ROUNDDOWN(va, PGSIZE), *end = ROUNDUP(va + len, PGSIZE);
    for (; begin < end; begin += PGSIZE)
    {
        struct Page *p = page_alloc(0);
        if (!p)
          panic("env region_alloc failed");
        page_insert(e->env_pgdir, p, begin, PTE_W | PTE_U);
    }
  // Hint: It is easier to use region_alloc if the caller can pass
  //   'va' and 'len' values that are not page-aligned.
```

```
  //   You should round va down, and round (va + len) up.
  //   (Watch out for corner-cases!)
}
```

load_icode()

加载用户程序二进制代码。

- 设置进程的tf_eip值为 elf->e_entry，并分配映射用户栈内存。
- 类似于boot loader从磁盘中加载内核，首先需要读取ELF header，这里将binary做强制类型转换即可 接着将类型为ELF_PROG_LOAD的segment载入内存，其实最快的方法是直接利用memcpy的方法进行内存的拷贝，但是这里存在一个问题，因为此时的page directory依旧是kernel的kern_pgdir，而我们需要将数据拷贝到environment e自己的address space中.
  - 需要先执行指令"lcr3(PADDR(e->env_pgdir)); 进入e的address space，再进行memcpy。
  - 之后再lcr3(PADDR(kern_pgdir)),转换回来即可。
  - lcr3()函数进行space address的转换。
- 最后，我们需要制定environment e的执行入口，其实就是初始化e->env_tf.tf_eip。

```
static void
load_icode(struct Env *e, uint8_t *binary)
{
    struct Elf *env_elf;
    struct Proghdr *ph, *eph;
    env_elf = (struct Elf*)binary;
    ph = (struct Proghdr*)((uint8_t*)(env_elf) + env_elf->e_phoff);
    eph = ph + env_elf->e_phnum;

    lcr3(PADDR(e->env_pgdir));

    for (; ph < eph; ph++) {
        if(ph->p_type == ELF_PROG_LOAD) {
            region_alloc(e, (void *)ph->p_va, ph->p_memsz);
            memmove((void*)ph->p_va, (void *)(binary+ph->p_offset), ph-
>p_filesz);
            memset((void*)(ph->p_va + ph->p_filesz), 0, ph->p_memsz-ph-
>p_filesz);
        }
    }

    e->env_tf.tf_eip = env_elf->e_entry;
    lcr3(PADDR(kern_pgdir));

    // Now map one page for the program's initial stack
    // at virtual address USTACKTOP - PGSIZE.
    region_alloc(e, (void *)(USTACKTOP-PGSIZE), PGSIZE);
}
```

env_create()

创建并分配一个新的进程，设置进程的type，以及加载二进制文件到新创建的进程的地址空间。

```
// Allocates a new env with env_alloc, loads the named elf
// binary into it with load_icode, and sets its env_type.
// This function is ONLY called during kernel initialization,
// before running the first user-mode environment.
// The new env's parent ID is set to 0.
//
void
env_create(uint8_t *binary, size_t size, enum EnvType type)
{
    // LAB 3: Your code here.
    struct Env *e;
    env_alloc(&e, 0);//分配一个进程，包括分配envid，设置e->env_tf寄存器
    e->env_type = type;
    load_icode(e, binary);
}
```

env_run()

在用户模式运行用户进程。

- 如果有正在运行的进程，就设置其属性为就绪态。
- 将该进程设置为运行态。
- 转换到该space address。
- 最后一句env_pop_tf函数，就是将当前进程的trapframe通过弹栈的形式，切换当前的运行环境。

```
void
env_run(struct Env *e)
{
    // panic("env_run not yet implemented");
    if (curenv && curenv->env_status == ENV_RUNNING) {
        curenv->env_status = ENV_RUNNABLE;
    }
    curenv = e;
    curenv->env_status = ENV_RUNNING;
    curenv->env_runs++;
    lcr3(PADDR(curenv->env_pgdir));
    env_pop_tf(&curenv->env_tf);
}
```

Exercise 3

学习异常和中断的理论知识。

> 如果还没有读过的话，读一读 80386 Programmer's Manual 中的 Chapter 9, Exceptions and Interrupts （或者 IA-32 Developer's Manual 的第五章）

Exercise 4

> 编辑一下trapentry.S 和 trap.c 文件，并且实现上面所说的功能。宏定义 TRAPHANDLER 和 TRAPHANDLER_NOEC 会对你有帮助。你将会在 trapentry.S文件中为在inc/trap.h文件中的每一个trap加入一个入口指， 你也将会提供_alttraps的值。
>
> 你需要修改trap_init()函数来初始化idt表，使表中每一项指向定义在trapentry.S中的入口指针，SETGATE宏定义在这里用得上。 你所实现的 _alltraps 应该：
>
> 1. 把值压入堆栈使堆栈看起来像一个结构体 Trapframe
> 2. 加载 GD_KD 的值到 %ds, %es寄存器中
> 3. 把%esp的值压入，并且传递一个指向Trapframe的指针到trap()函数中。
> 4. 调用trap 考虑使用pushal指令，他会很好的和结构体 Trapframe 的布局配合好。

先看一下`trapentry.S`文件，里面定义了两个宏定义，`TRAPHANDLER`，`TRAPHANDLER_NOEC`。他们的功能从汇编代码中可以看出：声明了一个全局符号name，并且这个符号是函数类型的，代表它是一个中断处理函数名。其实这里就是两个宏定义的函数。这两个函数就是当系统检测到一个中断/异常时，需要首先完成的一部分操作，包括：中断异常码，中断错误码(error code)。*正是因为有些中断有中断错误码，有些没有，所以我们采用利用两个宏定义函数*。可以参考实验简介中的error code相关信息。

alltraps函数其实就是为了能够让程序在之后调用trap.c中的trap函数时，能够正确的访问到输入的参数，即Trapframe指针类型的输入参数tf。

```
/*
 * Lab 3: Your code here for generating entry points for the different traps.
 */

TRAPHANDLER_NOEC(handler0, T_DIVIDE)
TRAPHANDLER_NOEC(handler1, T_DEBUG)
TRAPHANDLER_NOEC(handler2, T_NMI)
TRAPHANDLER_NOEC(handler3, T_BRKPT)
TRAPHANDLER_NOEC(handler4, T_OFLOW)
TRAPHANDLER_NOEC(handler5, T_BOUND)
TRAPHANDLER_NOEC(handler6, T_ILLOP)
TRAPHANDLER(handler7, T_DEVICE)
TRAPHANDLER_NOEC(handler8, T_DBLFLT)
TRAPHANDLER(handler10, T_TSS)
TRAPHANDLER(handler11, T_SEGNP)
TRAPHANDLER(handler12, T_STACK)
TRAPHANDLER(handler13, T_GPFLT)
TRAPHANDLER(handler14, T_PGFLT)
TRAPHANDLER_NOEC(handler16, T_FPERR)
TRAPHANDLER_NOEC(handler48, T_SYSCALL)

/*
 * Lab 3: Your code here for _alltraps
 */
_alltraps:
        pushl %ds
        pushl %es
```

```
        pushal
        movw $GD_KD, %ax
        movw %ax, %ds
        movw %ax, %es
        pushl %esp
        call trap /*never return*/

1:jmp 1b
```

最后，在`trap.c`中补全`trap_init()`函数，主要是使用SETGATE来初始化中断向量。关于SETGATE函数定义在`mmu.h`中。

设置 `IDT` ，需要先声明函数，需要注意，由于 `break_point` 普通用户也可以使用，所以 `DPL = 3`。

- gate：是idt表的index入口
- istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
- sel: 代码段选择子 for interrupt/trap handler
- off: 代码段偏移 for interrupt/trap handler
- dpl: 描述符特权级

```
void
trap_init(void)
{
    extern struct Segdesc gdt[];

    // LAB 3: Your code here.
    void handler0();
    void handler1();
    void handler2();
    void handler3();
    void handler4();
    void handler5();
    void handler6();
    void handler7();
    void handler8();
    void handler10();
    void handler11();
    void handler12();
    void handler13();
    void handler14();
    void handler15();
    void handler16();
    void handler48();

    SETGATE(idt[T_DIVIDE], 0, GD_KT, handler0, 0);
    SETGATE(idt[T_DEBUG], 0, GD_KT, handler1, 0);
    SETGATE(idt[T_NMI], 0, GD_KT, handler2, 0);

    // T_BRKPT DPL 3
```

```
    SETGATE(idt[T_BRKPT], 0, GD_KT, handler3, 3);

    SETGATE(idt[T_OFLOW], 0, GD_KT, handler4, 0);
    SETGATE(idt[T_BOUND], 0, GD_KT, handler5, 0);
    SETGATE(idt[T_ILLOP], 0, GD_KT, handler6, 0);
    SETGATE(idt[T_DEVICE], 0, GD_KT, handler7, 0);
    SETGATE(idt[T_DBLFLT], 0, GD_KT, handler8, 0);
    SETGATE(idt[T_TSS], 0, GD_KT, handler10, 0);
    SETGATE(idt[T_SEGNP], 0, GD_KT, handler11, 0);
    SETGATE(idt[T_STACK], 0, GD_KT, handler12, 0);
    SETGATE(idt[T_GPFLT], 0, GD_KT, handler13, 0);
    SETGATE(idt[T_PGFLT], 0, GD_KT, handler14, 0);
    SETGATE(idt[T_FPERR], 0, GD_KT, handler16, 0);

    // T_SYSCALL DPL 3
    SETGATE(idt[T_SYSCALL], 0, GD_KT, handler48, 3);

    // Per-CPU setup
    trap_init_percpu();
}
```

Question:

1. 为每个异常/中断设置单独的处理函数的目的是什么?

   解答：不同的中断需要不同的中断处理程序。因为对待不同的中断需要进行不同的处理方式，有些中断比如指令错误，就需要直接中断程序的运行。 而I/O中断只需要读取数据后，程序再继续运行。

2. 为什么`user/softint.c`程序调用的是`int $14`会报13异常(general protection fault)?

   解答：因为当前系统运行在用户态下，特权级为3，而INT 指令为系统指令，特权级为0。 会引发General Protection Exception，就是trap13。

Exercise 5

练习5，6是在trap_dispatch中对page fault异常和breakpoint异常进行处理。作业5实现对page_fault

```
static void
trap_dispatch(struct Trapframe *tf)
{
    int32_t ret_code;
    // Handle processor exceptions.
    switch(tf->tf_trapno) {
        case (T_PGFLT):
            page_fault_handler(tf);
            break;
         default:
            // 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 {
                env_destroy(curenv);
                return;
            }
    }
}
```

Exercise 6

只需要在练习五的基础上，加一下breakpoint异常处理即可。

```
static void
trap_dispatch(struct Trapframe *tf)
{
    int32_t ret_code;
    // Handle processor exceptions.
    switch(tf->tf_trapno) {
        case (T_PGFLT):
            page_fault_handler(tf);
            break;
        case (T_BRKPT):
            monitor(tf);
            break;
         default:
            // 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 {
                env_destroy(curenv);
                return;
            }
    }
}
```

Question：

> - 断点那个测试样例可能会生成一个断点异常，或者生成一个一般保护错，这取决你是怎样在 IDT 中初始化它的入口的（换句话说，你是怎样在 `trap_init` 中调用 `SETGATE` 方法的）。为什么？你应该做什么才能让断点异常像上面所说的那样工作？怎样的错误配置会导致一般保护错？
> - 你认为这样的机制意义是什么？尤其要想想测试程序 `user/softint` 的所作所为 / 尤其要考虑一下 `user/softint` 测试程序的行为。

如果设置 `break point` 的 `DPL = 0` 则会引发权限错误，由于这里设置的 `DPL = 3` ，所以会引发断点。

这个机制有效地防止了一些程序恶意任意调用指令，引发一些危险的错误，所以我认为这个粒度的权限机制时十分必要的。

Exercise 7

实现系统调用的支持，需要修改`trap_dispatch()`和`kern/syscall.c`。

1. 分别在trapentry.S和trap.c的trap_init()函数中添加如下代码：

```
TRAPHANDLER_NOEC(th_syscall, T_SYSCALL)
```

```
SETGATE(idt[T_SYSCALL], 0, GD_KT, th_syscall, 3);
```

2. 在trap_dispatch()中加入如下代码

```
 if (tf->tf_trapno == T_SYSCALL) {
        tf->tf_regs.reg_eax = syscall(
            tf->tf_regs.reg_eax,
            tf->tf_regs.reg_edx,
            tf->tf_regs.reg_ecx,
            tf->tf_regs.reg_ebx,
            tf->tf_regs.reg_edi,
            tf->tf_regs.reg_esi
        );
        return;
 }
```

3. 接着在`kern/syscall.c`中对不同类型的系统调用处理。

```
// Dispatches to the correct kernel function, passing the arguments.
int32_t
syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t
a4, uint32_t a5)
{
    switch (syscallno) {
    case SYS_cputs:
        sys_cputs((char *)a1, a2);
        return 0;
    case SYS_cgetc:
        return sys_cgetc();
    case SYS_getenvid:
        return sys_getenvid();
    case SYS_env_destroy:
        return sys_env_destroy(a1);
    default:
        return -E_INVAL;
    }
}
```

Exercise 8

完成作业7之后，在执行`user/hello.c`的第二句cprintf报 page fault，因为还没有设置它用到的thisenv的值。在`lib/libmain.c`的libmain()如下设置即可完成作业8：

```
thisenv = &envs[ENVX(sys_getenvid())];
```

Exercise 9

1. 首先如果页错误发生在内核态时应该直接panic。

   按照给定提示：要判断缺页是发生在用户模式还是内核模式下，只需检查`tf_cs`的低位。

```
void
page_fault_handler(struct Trapframe *tf)
{
    ...
    // Handle kernel-mode page faults.

    // LAB 3: Your code here.
    if ((tf->tf_cs & 3) == 0) {
        panic("kernel page fault at:%x\n", fault_va);
    }
    ...
}
```

2. 实现`kern/pmap.c`中的`user_mem_check()`工具函数。

   该函数检测用户环境是否有权限访问线性地址区域[va, va+len)。然后对在kern/syscall.c中的系统调用函数使用user_mem_check()工具函数进行内存访问权限检查。

```
int
user_mem_check(struct Env *env, const void *va, size_t len, int perm)
{
    uint32_t begin = (uint32_t)ROUNDDOWN(va, PGSIZE), end =
(uint32_t)ROUNDUP(va + len, PGSIZE);
    int check_perm = (perm | PTE_P);
    uint32_t check_va = (uint32_t)va;

    for (; begin < end; begin += PGSIZE) {
        pte_t *pte = pgdir_walk(env->env_pgdir, (void *)begin, 0);
        if ((begin >= ULIM) || !pte || (*pte & check_perm) != check_perm) {
            user_mem_check_addr = (begin >= check_va ? begin : check_va);
            return -E_FAULT;
        }
    }

    return 0;
```

```
}
```

提示：调整 `kern/syscall.c` 来验证系统调用的参数。

有了工具函数，在`kern/syscall.c`中的系统调用函数只有`sys_cputs()`参数中有指针，所以需要对其进行检测：

```
// Print a string to the system console.
// The string is exactly 'len' characters long.
// Destroys the environment on memory errors.
static void
sys_cputs(const char *s, size_t len)
{
    // Check that the user has permission to read memory [s, s+len).
    // Destroy the environment if not.

    // LAB 3: Your code here.
    user_mem_assert(curenv, s, len, 0);
    // Print the string supplied by the user.
    cprintf("%.*s", len, s);
}
```

此外，在`kern/kdebug.c`的debuginfo_eip()中加入检查。

```
// Make sure this memory is valid.
// Return -1 if it is not.  Hint: Call user_mem_check.
// LAB 3: Your code here.
if (user_mem_check(curenv, usd, sizeof(struct UserStabData), PTE_U))
    return -1;

// Make sure the STABS and string table memory is valid.
// LAB 3: Your code here.
if (user_mem_check(curenv, stabs, stab_end - stabs, PTE_U))
    return -1;

if (user_mem_check(curenv, stabstr, stabstr_end - stabstr, PTE_U))
    return -1;
```

Exercise 10

进行测试即可

遇到的问题

1. make grade 不给分，出现no jos.out

   按照网上的解决办法，修改了makefile的部分内容。

查看之前的git记录发现，在minor GUNmakefile时，删除了这一段话，至于为什么还不知道。