Lab3

前言

之前我们研究了一下系统启动和堆栈，以及操作系统的内存分配，实现了物理分配和保护模式的跳转，这一个lab将会实现保护模式下的用户进程。

在本次实验中，你将实现使保护模式下的用户进程(英文原文是environment，下同)得以运行的基础内核功能。在你的努力下，JOS 内核将建立起用于追踪用户进程的数据结构，创建一个用户进程，读入程序映像并运行。你也会使 JOS 内核有能力响应用户进程的任何系统调用，并处理用户进程所造成的异常。

Lab3新增了一些代码，有点多，具体内容见下表。

目录	文件	说明
user/	*	检查Lab3内核代码的各种测试程序
inc/	env.h	用户模式进程的公用定义
	trap.h	陷阱处理的公用定义
	syscall.h	用户进程向内核发起系统调用的公用定义
	lib.h	用户模式支持库的公用定义
kern/	env.h	用户模式进程的内核私有定义
	env.c	用户模式进程的内核代码实现
	trap.h	陷阱处理的内核私有定义
	trap.c	与陷阱处理有关的代码
	trapentry.S	汇编语言的陷阱处理函数入口点
	syscall.h	系统调用处理的内核私有定义
	syscall.c	与系统调用实现有关的代码
lib/	Makefrag	构建用户模式调用库的Makefile fragment, obj/lib/libuser.a
	entry.S	汇编语言的用户进程入口点
	libmain.c	从entry.S进入用户模式的库调用
	syscall.c	用户模式下的系统调用桩(占位)函数
	console.c	用户模式下putchar()和getchar()的实现，提供控制台输入输出
	exit.c	用户模式下exit()的实现
	panic.c	用户模式下panic()的实现

Lab3的exercise1

问题

修改`kern/pmap.c`中的`mem_init()`函数来分配并映射`envs`数组。这个数组恰好包含`NENV`个`Env`结构体实例，这与你分配`pages`数组的方式非常相似。另一个相似之处是，支持`envs`的内存储应该被只读映射在页表中`UENVS`的位置（于`inc/memlayout.h`中定义），所以，用户进程可以从这一数组读取数据。

修改好后，`check_kern_pgdir()`应该能够成功执行。

前置知识

一个很严重的bug。如果我们直接进行`make qemu-nox`的话，会爆出一个神奇的错误。

```
kern_pgdir = 0
kernel panic at kern/pmap.c:158: PADDR called with invalid kva 00000000
Welcome to the JOS kernel monitor!
```

加入调试以后的代码是这样的。

```
kern_pgdir = (pde_t *) boot_alloc(PGSIZE);
cprintf("kern_pgdir = %x, PGSIZE = %x\n", kern_pgdir, PGSIZE);
memset(kern_pgdir, 0, PGSIZE);
cprintf("kern_pgdir = %x, PGSIZE = %x\n", kern_pgdir, PGSIZE);
```

对应的输出是这样的。从这里可以看出的是，`kern_pgdir`在`memset`过后自动清零了，这个是bug的产生表象。

```
kern_pgdir = f018f000, PGSIZE = 1000
kern_pgdir = 0, PGSIZE = 1000
```

对应的bug在知乎上已经有阐述，术语连接器的linker问题，需要对与bss段进行修改(`kern/kernel.ld`):

```
.bss : {
    PROVIDE(edata = .);
    *(.dynbss)
    *(.bss .bss.*)
    *(COMMON)
    PROVIDE(end = .);
}
```

至此运行qemu，运行正常。

解答

这里首先在`kern/pmap.c`中定位到`mem_init`，然后找一下Lab3要求新增加什么代码，这里有样学样就可以，而且样子就在前面不远的地方。

```
// Make 'envs' point to an array of size 'NENV' of 'struct Env'.
// LAB 3: Your code here.
// 给ENV分配内存
envs = (struct Env *) boot_alloc(NENV * sizeof(struct Env));
memset(envs, 0, NENV * sizeof(struct Env));
cprintf("envs = %x, NENV = %x, sizeof(Env) = %x\n", envs, NENV, sizeof(struct Env));
```

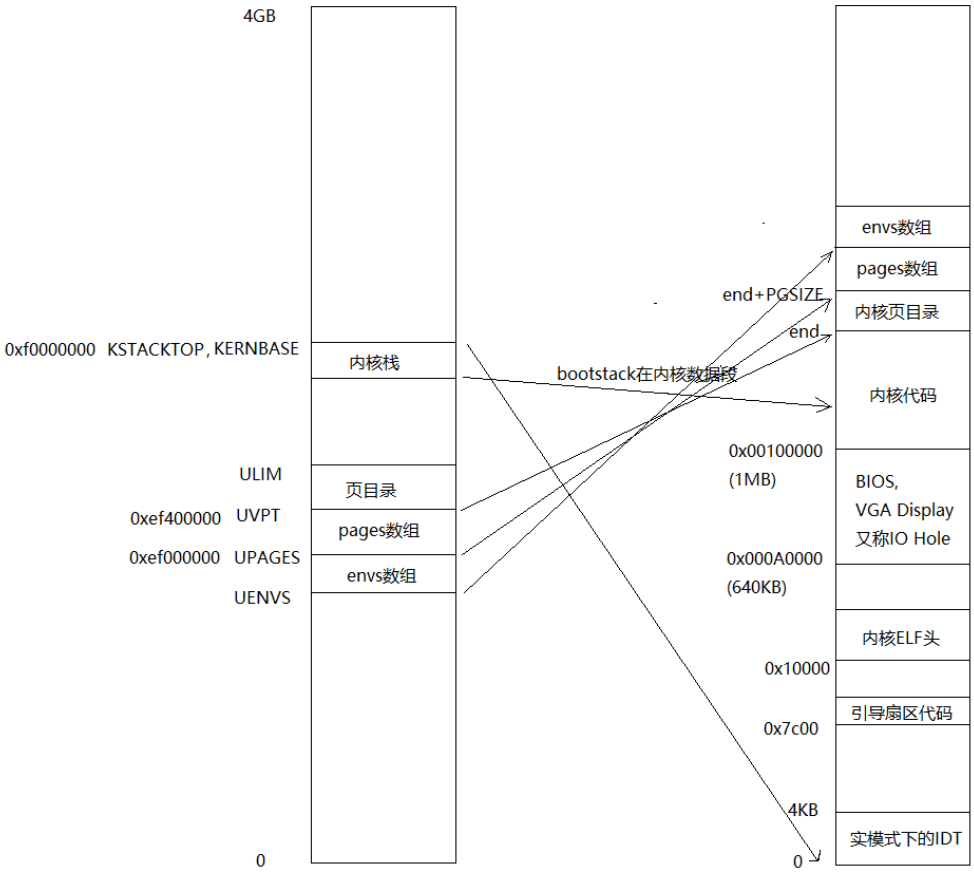
这里对应的输出是

```
envs = f01d0000, NENV = 400, sizeof(Env) = 60
```

除此之外还有一个地方，就是在`boot_map_region`之间。这里也是有样学样。

```
// 但是对于ENV而言，和进程一样的存在还是不要随便瞎写比较好。
boot_map_region(kern_pgdir, (uintptr_t)UENVS,
                ROUNDUP(NENV * sizeof(struct Env), PGSIZE),
PADDR(envs),PTE_U);
```

然后运行`make qemu-nox`，可以看到`check_kern_pgdir()`通过即可。此时我们完成这样的映射（见下图）。



Lab3的exercise2

问题

在`env.c`中，完成接下来的这些函数：

1. `env_init()` 初始化全部 `envs` 数组中的 `Env` 结构体，并将它们加入到 `env_free_list` 中。还要调用 `env_init_percpu`，这个函数会通过配置段硬件，将其分隔为特权等级0(内核)和特权等级3(用户)两个不同的段。
2. `env_setup_vm()` 为新的进程分配一个页目录，并初始化新进程的地址空间对应的内核部分。
3. `region_alloc()` 为进程分配和映射物理内存。
4. `load_icode()` 你需要处理ELF二进制映像，就像是引导加载程序(bootloader)已经做好的那样，并将映像内容读入新进程的用户地址空间。
5. `env_create()` 通过调用 `env_alloc` 分配一个新进程，并调用 `load_icode` 读入ELF二进制映像。
6. `env_run()` 启动给定的在用户模式运行的进程。

当你在完成这些函数时，你也许会发现 `cprintf` 的新的%e很好用，它会打印出与错误代码相对应的描述，例如：`r = -E_NO_MEM; panic("env_alloc: %e", r);` 会panic并打印出 `env_alloc: out of memory`。

前置知识

1. JOS中的 `Env`，在JOS中，有这样的声明：

```
struct Env *envs = NULL;        // All environments
struct Env *curenv = NULL;      // The current env
static struct Env *env_free_list;   // Free environment list
```

   其中 `env_free_list` 维护所有不会动的 `Env` 结构体，类似空闲链表，内核使用 `curenv` 代表当前运行的环境，内核启动之前是 `NULL`，`envs` 维护操作系统各种环境中的Env结构体数组。

2. `Env` 的结构体和相关的赋值在这里给了出来。这里对应的是代码的 `inc/env.h`。

```
enum {
    ENV_FREE = 0,
    ENV_DYING,
    ENV_RUNNABLE,
    ENV_RUNNING,
    ENV_NOT_RUNNABLE
};
// Special environment types
enum EnvType {
    ENV_TYPE_USER = 0,
};
struct Env {
    struct Trapframe env_tf;    // 陷入trap时的寄存器状态
    struct Env *env_link;       // 构建链表
    envid_t env_id;             // 唯一id
    envid_t env_parent_id;      // 父环境id
    enum EnvType env_type;      // Env的所处大环境，比如kernel态和user态
    unsigned env_status;        // 这个Env的状态
    uint32_t env_runs;          // 这个环境跑了几次
    // Address space
    pde_t *env_pgdir;           // 指向页目录和页表定义
};
```

解答

这个EX是希望把 `ENV` 设置成一个自洽的数据结构，给之前的映射充分的不可写依据。

函数env_init()

首先定位到`kern/env.c`中，利用`Env`结构体的知识可以进行`env_init()`的定义。从lab2的实现中可以看出来这里和那个链表的定义如出一辙，所以直接可以把实现依葫芦画瓢照搬上去。选择初始化`env_link`，`env_id`，`env_status`。

```
void env_init(void)
{
    // 和page_init不同的是，这里需要进行倒序初始化
    for (int i = NENV - 1; i >= 0; i--)
    {
        envs[i].env_id = 0;
        envs[i].env_link = env_free_list;
        env_free_list = &envs[i];
    }
    // Per-CPU part of the initialization
    // 这里就不在报告中贴上这个函数的代码了，因为和本实验其实关系不大。
    env_init_percpu();
}
```

函数env_setup_vm()

为当前的进程分配一个页，用来存放页表目录，同时讲内核部分的内存映射完成。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。这个函数中完成了虚地址`UPAGES`和`UVPT`和内核栈的映射。其中的`UVPT`都是存放页表目录的地方，对于内核而言，是存放内核页表目录的地方，而对于用户是存放用户页表目录的地方。而UPAGES是存放内核页表的地方。

```
#define KADDR(pa) _kaddr(__FILE__, __LINE__, pa)
// 真实有用的是第三个参数
static inline void *_kaddr(const char *file, int line, physaddr_t pa)
{
    if (PGNUM(pa) >= npages)
        _panic(file, line, "KADDR called with invalid pa %08lx", pa);
    return (void *)(pa + KERNBASE);
}
static inline void *page2kva(struct PageInfo *pp)
{
    return KADDR(page2pa(pp));
}
static int env_setup_vm(struct Env *e)
{
    int i;
    struct PageInfo *p = NULL;
    // 分配页失败
    if (!(p = page_alloc(ALLOC_ZERO)))
        return -E_NO_MEM;
    // 页目录项拿出来
    e->env_pgdir = page2kva(p);
    // 存储的是自己的内核空间，不过继承kernel的不会有问题
    memcpy(e->env_pgdir, kern_pgdir, PGSIZE);
    p->pp_ref++; // 引用+1，因为在page_alloc并不会初始化为1
    // 和mem_init一样，只不过设置的是不同的进程
    e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_P | PTE_U;
    return 0;
}
```

函数region_alloc()

分配`len`字节的物理内存给`env`环境，之后映射到环境中的虚拟地址`va`。要注意的是，`va`和`len`都要进行对齐。注意这里的`p`一直在改变。

```
static void region_alloc(struct Env *e, void *va, size_t len)
{
    // 上下界对齐
    uintptr_t va_start = ROUNDDOWN((uintptr_t)va, PGSIZE);
    uintptr_t va_end = ROUNDUP((uintptr_t)va + len, PGSIZE);
    // 先弄一个，省的在for里面浪费性能
    struct PageInfo *pginfo = NULL;
    for (int cur_va = va_start; cur_va < va_end; cur_va += PGSIZE)
    {
        pginfo = page_alloc(0); // 分配一页
        if (!pginfo)            // 分不出来，内核：我很慌
        {
            int r = -E_NO_MEM;
            panic("region_alloc: %e", r);
        }
        //插入e的页目录表
        page_insert(e->env_pgdir, pginfo, (void*)cur_va, PTE_U|PTE_W);
    }
}
```

函数load_icode()

本函数希望把之前直接连接进内核的可执行二进制文件拿出来执行，首先要做的事解析ELF的文件头和隔断的程序头，然后根据头文件中的信息，将指定字节的信息拷贝在指定的虚地址处。

这里的拷贝到指定的虚地址处，是指用户空间的虚地址，而不是内核空间的虚地址，所以还需要用`lcr3`函数加载用户空间的页表目录才能将地址转换为用户空间地址。载入elf文件并开始执行的程序段在`boot/main.c`中有，可以参考那部分代码。

```
static void load_icode(struct Env *e, uint8_t *binary)
{
    // LAB 3: Your code here.
    struct Proghdr *ph, *eph;
    struct Elf *elf = (struct Elf *)binary;
    if (elf->e_magic != ELF_MAGIC)
        panic("load_icode: not an ELF file");
    ph = (struct Proghdr *)(binary + elf->e_phoff);
    eph = ph + elf->e_phnum;
    lcr3(PADDR(e->env_pgdir));
    for (; ph < eph; ph++)
        if (ph->p_type == ELF_PROG_LOAD)
        {
            if (ph->p_filesz > ph->p_memsz)
                panic("load_icode: file size > memory size");
            region_alloc(e, (void *)ph->p_va, ph->p_memsz);
            memcpy((void *)ph->p_va, 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 = elf->e_entry;
    region_alloc(e, (void *)USTACKTOP - PGSIZE, PGSIZE);
    lcr3(PADDR(kern_pgdir));
```

```
}
```

函数env_create()

创建一个新的进程。首秀按申请一个进程描述符，调用`env_setup_vm`来完成页表目录的设置和内核区域的映射，然后将指定的二进制文件载入到内存中来。注意这里需要使用到`env_alloc`来分配各种东西。

```
void env_create(uint8_t *binary, enum EnvType type)
{
    // 输入二进制码和进程类型
    struct Env *e;
    // 获得一个进程的各种信息
    int r = env_alloc(&e, 0);
    // 分配失败
    if (r < 0)
        panic("env_create: %e", r);
    e->env_type = type;
    // 读取二进制
    load_icode(e, binary);
}
```

函数env_run()

真正启动的位置，这里涉及到与其他进程的交互，如果被占用了就要把它设置会`RUNNABLE`，而不是`RUNNING`，然后切换页表目录。

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

void env_run(struct Env *e)
{
    // 把进程停下来
    if (curenv && curenv->env_status == ENV_RUNNING)
        curenv->env_status = ENV_RUNNABLE;
    // 把进程换掉
    curenv = e;
    e->env_status = ENV_RUNNING;
    e->env_runs++; // 绝对是Lab4的伏笔
    // 更换页目录表
    lcr3(PADDR(e->env_pgdir));
    // 更新上一个寄存器的状态
    env_pop_tf(&e->env_tf);
    // panic("env_run not yet implemented");
}
```

运行结束以后会得到一个错误，这是因为没有建立任何允许从用空见到内核空间的方式，所以会产生一次保护异常，然后发现保护异常也没有，就重复三次，然后就重置了。下一部分有了中断以后会讲解这是怎么完成整个的跳转。

Lab3的exercise4

问题

编辑`trapentry.S`和`trap.c`，以实现上面描述的功能。`trapentry.S`中的宏定义`TRAPHANDLER`和`TRAPHANDLER_NOEC`，还有在`inc/trap.h`中的那些`T_`开头的宏定义应该能帮到你。你需要在`trapentry.S`中用那些宏定义为每一个`inc/trap.h`中的trap添加一个新的入口点，你也要提供`TRAPHANDLER`宏所指向的`_alltraps`的代码。你还要修改`trap_init()`来初始化IDT，使其指向每一个定义在`trapentry.S`中的入口点。`SETGATE`宏定义在这里会很有帮助。你的`_alltraps`应该：

1. 将一些值压栈，使栈帧看起来像是一个`struct Trapframe`。
2. 将`GD_KD`读入`%ds`和`%es`。
3. `push %esp`来传递一个指向这个`Trapframe`的指针，作为传给`trap()`的参数。
4. `call trap`（思考：trap这个函数会返回吗？）。

考虑使用`pushal`这条指令。它在形成`struct Trapframe`的层次结构时非常合适。用一些user目录下会造成异常的程序测试一下你的陷阱处理代码，比如`user/divzero`。现在，你应该能在`make grade`中通过`divzero`，`softint`和`badsegment`了。

前置知识

保护控制

在Intel的术语中，**中断**是一个由异步事件造成的保护控制转移，这一事件通常是在处理器外部发生的，例如外接设备的I/O活动通知。相反，**异常**是一个由当前正在运行的代码造成的同步保护控制转移，例如除零或者不合法的内存访问。

为了确保这些保护控制转移确实是受**保护**的，处理器的中断/异常处理机制被设计成当发生中断或异常时当前运行的代码没**有机会任意选择从何处陷入内核或如何陷入内核**，而是由处理器确保仅在小心控制的情况下才能进入内核。在x86架构中，两种机制协同工作来提供这一保护：

1. 中断描述符表，处理起确保中断和异常只能导致内核进入一些确定的，设计优良的，由内核自身确定的入口点，而不是在发生中断或异常时由正在运行的代码决定。x86最多允许256个不同的这样的入口，每个有不同的中断向量（0-255）。在这个表中对应的入口，处理器会读取：
    1. 一个读入指令寄存器的值，指向处理这一类型异常的内核代码。
    2. 一个读入代码段寄存器的值，用包含一些0-1位来表示异常处理代码应该运行在哪一个特权等级。在JOS中，所有的异常都是内核处理，特权0。
2. 任务状态段。处理器需要一处位置，用来在中断或异常发生前保存旧的处理器状态。但用于保存旧处理器状态的区域必须避免被非特权的用户模式代码访问到，否则有错误的或恶意的用户模式代码可能危及内核安全。因此，当x86处理器遇到使得特权等级从用户模式切换到内核模式的中断或陷阱时，它也会将栈切换到内核的内存中的栈。一个被称作任务状态段（TSS）来描述这个栈所处的段选择子和地址。
3. 处理器将`SS,ESP,EFLAGS,CS,EIP`和一个可能存在的错误代码压入新栈，接着它从中断向量表中读取`CS`和`EIP`，并使`ESP`和`SS`指向新栈。即使TSS很大，可以服务于多种不同目的，JOS只将它用于定义处理器从用户模式切换到内核模式时的内核栈。因为JOS的"内核模式"在x86中是特权等级0，当进入内核模式时，处理器用TSS结构体的`ESP0`和`SS0`字段来定义内核栈。JOS不使用TSS中的其他任何字段。

异常和中断的类型

处理起的全部同步异常使用0-31作为中断向量。比如缺页异常总是会引发异常14，大于31的中断向量只能被软件中断，这些终端可以用int生成，或者被用于异步硬件中断。

中断向量表

这在`inc/trap.h`中可以找到。

```
// Trap numbers
// These are processor defined:
#define T_DIVIDE     0      // divide error
#define T_DEBUG      1      // debug exception
#define T_NMI        2      // non-maskable interrupt
#define T_BRKPT      3      // breakpoint
#define T_OFLOW      4      // overflow
#define T_BOUND      5      // bounds check
#define T_ILLOP      6      // illegal opcode
#define T_DEVICE     7      // device not available
#define T_DBLFLT     8      // double fault
/* #define T_COPROC  9 */   // reserved (not generated by recent processors)
#define T_TSS       10      // invalid task switch segment
#define T_SEGNP     11      // segment not present
#define T_STACK     12      // stack exception
#define T_GPFLT     13      // general protection fault
#define T_PGFLT     14      // page fault
/* #define T_RES    15 */   // reserved
#define T_FPERR     16      // floating point error
#define T_ALIGN     17      // aligment check
#define T_MCHK      18      // machine check
#define T_SIMDERR   19      // SIMD floating point error

// These are arbitrarily chosen, but with care not to overlap
// processor defined exceptions or interrupt vectors.
#define T_SYSCALL   48      // system call
#define T_DEFAULT   500     // catchall
```

解答

之前挖下的坑应该填上了，在这个比较自洽的结构之上，我们希望完成整个过程的梳理。

CPU如何控制中断-初始化

第一步还是进入`kern/init.c`的`i386_init()`，可以发现的是多了一些很神奇的东西，来看看代码。

```
    // Lab3我们要做的
    env_init();
    trap_init();
#if defined(TEST)
    // Don't touch -- used by grading script!
    ENV_CREATE(TEST, ENV_TYPE_USER);
#else
    // Touch all you want.
    ENV_CREATE(user_hello, ENV_TYPE_USER);
#endif // TEST*
    // 我们只允许一个用户在这个环境中，所以直接开始跑。
    env_run(&envs[0]);
```

之前的代码已经都获得讲述，所以都忽略掉了，当然在`mem_init`里面还是给进程分配了空间并且把东西映射到虚拟内存上。之后通过`env`的初始化来获得管理多个进程的能力，虽然现在还是用不上。

接下来是`trap_init()`，这里初始化中断控制的处理方式。特别要注意的是`trap_init()`在`kern/trap.c`中，这意味着这还是内核的代码，不过这个的理由显而易见，毕竟只有内核才能处理中断，而用户只能引发中断（甚至是部分的）。

接下来是这个`ENV_CREATE`，这个define定义在`kern/env.h`中找到。

```
#define ENV_PASTE3(x, y, z) x ## y ## z
#define ENV_CREATE(x, type)                                     \
    do                                                          \
    {                                                           \
        extern uint8_t ENV_PASTE3(_binary_obj_, x, _start)[]; \
        env_create(ENV_PASTE3(_binary_obj_, x, _start),       \
                   type);                                       \
    } while (0)
```

这个PASTE3用法比较奇特，[这个链接]给出了一个比较经典的用法。

```
#define t(x, y, z) x##y##z
int j[] = {t(1, 2, 3), t(, 4, 5), t(6, , 7), t(8, 9, ),
           t(10, , ), t(, 11, ), t(, , 12), t(, , )};
// int j[] = {123, 45, 67, 89, 10, 11, 12};
```

最后就是`env_run`来设置这个开始跑的线程。似乎比较简单，但是我们依旧没有完成设置-毕竟`trap_init`都还没进去看过。而且这里只是讲解初始化，真正的如何拦截还在后面。

不过在讲解这些之前，需要把空都填完了，让CPU具有基本的拦截能力才能比较好地理解。

文件trapentry.S

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

很显而易见，这两块对应压栈的处理，但是不是很完全，而且`jmp`对应的代码其实是不存在的，所以我们需要把这些补全，然后设置向量入口。除此以外，两者的区别在于给不同的中断使用，如果需要压错误码的话，由CPU完成这件事，如果没有错误码的话，压入一个0，这样就能保证结构体`trapframe`的结构保持一致，当然，CPU自身也会`push`一些进去。

添加下述代码即可。

```
.text
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_NOEC(handler7, T_DEVICE)
TRAPHANDLER(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(handler17, T_ALIGN)
TRAPHANDLER_NOEC(handler18, T_MCHK)
TRAPHANDLER_NOEC(handler19, T_SIMDERR)
TRAPHANDLER_NOEC(handler48, T_SYSCALL)
_alltraps:
    pushl %ds
    pushl %es
    pushal
    movw $GD_KD, %ax
    movw %ax, %ds
    movw %ax, %es
    pushl %esp
    call trap
```

函数trap_init()

在`kern/trap.c`中`trap_init()`中添加下述代码，这些都是链接在一起的，所以需要放在一起理解。在`inc/mmu.h`中涉及到`SETGATE`的定义。这里也一并放上来。

```
#define SETGATE(gate, istrap, sel, off, dpl)             \
    {                                                    \
        (gate).gd_off_15_0 = (uint32_t)(off)&0xffff;     \
        (gate).gd_sel = (sel);                           \
        (gate).gd_args = 0;                              \
        (gate).gd_rsv1 = 0;                              \
        (gate).gd_type = (istrap) ? STS_TG32 : STS_IG32; \
        (gate).gd_s = 0;                                 \
        (gate).gd_dpl = (dpl);                           \
        (gate).gd_p = 1;                                 \
        (gate).gd_off_31_16 = (uint32_t)(off) >> 16;     \
    }
struct Gatedesc
{
    unsigned gd_off_15_0 : 16;  // low 16 bits of offset in segment
    unsigned gd_sel : 16;       // segment selector
    unsigned gd_args : 5;       // # args, 0 for interrupt/trap gates
    unsigned gd_rsv1 : 3;       // reserved(should be zero I guess)
    unsigned gd_type : 4;       // type(STS_{TG,IG32,TG32})
    unsigned gd_s : 1;          // must be 0 (system)
    unsigned gd_dpl : 2;        // descriptor(meaning new) privilege level
    unsigned gd_p : 1;          // Present
```

```
    unsigned gd_off_31_16 : 16; // high bits of offset in segment
};

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();
    // 9号中断不可用，最近的处理器已经取消了这玩意
    void handler10();
    void handler11();
    void handler12();
    void handler13();
    void handler14();
    // 15号中断被保留，具体的可以在网上查找
    void handler16();
    void handler17();
    void handler18();
    void handler19();
    void handler48();
    SETGATE(idt[T_DIVIDE], 1, GD_KT, handler0, 0);
    SETGATE(idt[T_DEBUG], 1, GD_KT, handler1, 0);
    SETGATE(idt[T_NMI], 1, GD_KT, handler2, 0);
    // 断点调试可以被用户发起
    SETGATE(idt[T_BRKPT], 1, GD_KT, handler3, 3);
    SETGATE(idt[T_OFLOW], 1, GD_KT, handler4, 0);
    SETGATE(idt[T_BOUND], 1, GD_KT, handler5, 0);
    SETGATE(idt[T_ILLOP], 1, GD_KT, handler6, 0);
    SETGATE(idt[T_DEVICE], 1, GD_KT, handler7, 0);
    SETGATE(idt[T_DBLFLT], 1, GD_KT, handler8, 0);

    SETGATE(idt[T_TSS], 1, GD_KT, handler10, 0);
    SETGATE(idt[T_SEGNP], 1, GD_KT, handler11, 0);
    SETGATE(idt[T_STACK], 1, GD_KT, handler12, 0);
    SETGATE(idt[T_GPFLT], 1, GD_KT, handler13, 0);
    SETGATE(idt[T_PGFLT], 1, GD_KT, handler14, 0);

    SETGATE(idt[T_FPERR], 1, GD_KT, handler16, 0);
    SETGATE(idt[T_ALIGN], 1, GD_KT, handler17, 0);
    SETGATE(idt[T_MCHK], 1, GD_KT, handler18, 0);
    SETGATE(idt[T_SIMDERR], 1, GD_KT, handler19, 0);
    // interrupt
    // 系统调用也可以被用户发起
    SETGATE(idt[T_SYSCALL], 0, GD_KT, handler48, 3);
    // Per-CPU setup
    trap_init_percpu();
}
```

然后运行`make grade`，可以看到PartA的三个test都过去了。

在运行过程中如何拦截中断

以`divzero`来举例，这个文件在`user/divzero.c`，代码如下：

```
#include <inc/lib.h>
int zero;
void umain(int argc, char **argv)
{
    zero = 0;
    cprintf("1/0 is %08x!\n", 1/zero);
}
```

很显然会遇到一个问题，就是1/0是不合法的，根据之前的中断向量表，应当产生一个0号中断，这部分是CPU定死的，确定以后操作系统无法改变。

接下来打开`obj/user/divzero.asm`查看代码，使用gdb打上断点，为了更快确定问题，这里直接打到`printf`地前一条汇编，大约是这个位置。

```
  80004b: c7 00 00 00 00 00     movl   $0x0,(%eax)
  cprintf("1/0 is %08x!\n", 1/zero);
```

进入gdb，运行几次，可以发现在`idiv`这里发生中断。CPU捕获到了这个中断，完成了从用户态到内核的切换，并且根据中断号可以看出来的是捕获到0号中断，于是发生了对应的处理。

```
(gdb) b *0x80004b
Breakpoint 1 at 0x80004b
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x80004b:    movl   $0x0,(%eax)

Breakpoint 1, 0x0080004b in ?? ()
(gdb) si
=> 0x800051:    mov    $0x1,%eax
0x00800051 in ?? ()
(gdb) si
=> 0x800056:    mov    $0x0,%ecx
0x00800056 in ?? ()
(gdb) si
=> 0x80005b:    cltd
0x0080005b in ?? ()
(gdb) si
=> 0x80005c:    idiv   %ecx
0x0080005c in ?? ()
(gdb) si
=> 0xf010424c <handler0+2>:     push   $0x0
0xf010424c in handler0 ()
    at kern/trapentry.S:50
50      TRAPHANDLER_NOEC(handler0, T_DIVIDE)
```

之后就进入了最开始提到的这些汇编，把保存现场，之后调用`trap`函数。在这个函数中，最重要的是打印这个TRAP frame，并且进行各自的错误转发，在当前状态下我们并没有动过这个函数，所以只能摧毁这个正在运行的程序，有可能这不是我们想要的结果（比如缺页异常只需要补上就行了）。

然后就是继续运行这个环境，此时`curenv`已经从用户态切换到内核态。

之前提到的handler是怎么回事？怎么确定的0号中断

这又是一个长长的故事。

引出的问题

当你只做到这里运行的时候，发现运行第一遍时没有问题的，第二遍运行的时候，`badsegment`会花费很长的时间，不过依旧能运行，但是运行到第三遍的时候，会发现直接遇到错误。这个时候运行一些windows本身的程序，会发现遇到了内部错误。重启对于这一类问题没有效果。不知道这类问题是否是普遍现象。

注意刚才的问题可能指在PartB没有做完，`make grade`在运行完PartA的之后就直接Ctrl+C的时候才会发生。

Lab3的question1

问题

1. 对每一个中断/异常都分别给出中断处理函数的目的是什么？换句话说，如果所有的中断都交给同一个中断处理函数处理，现在我们实现的哪些功能就没办法实现了？
2. 你有没有额外做什么事情让`user/softint`这个程序按预期运行？打分脚本希望它产生一个一般保护错(陷阱13)，可是softint的代码却发送的是`int $14`。为什么这个产生了中断向量13？如果内核允许softint的`int $14`指令去调用内核中断向量14所对应的的缺页处理函数，会发生什么？

解答

1. 每个异常和中断处理方式不同，用一个handler难以实现如此多不同的，奇奇怪怪的要求。比如说除零在处理完以后还是可以继续的，但是有很多严重问题有可能会导致系统出错（比如之前的问题）。
2. 既然这样就直接找代码：下面是`user/softint.c`的代码。

```
#include <inc/lib.h>
void umain(int *argc, char **argv)
{
    asm volatile("int $14");  // page fault
}
```

下述是对应的评分标准

```
@test(10)
def test_softint():
    r.user_test("softint")
    r.match('Welcome to the JOS kernel monitor!',
            'Incoming TRAP frame at 0xeffffffc',
            'TRAP frame at 0xf.......',
            '  trap 0x0000000d General Protection',
            '  eip  0x008.....',
            '  ss   0x----0023',
            '.00001000. free env 0000100')
```

可以看到的是产生了一个缺页异常，但是实际上输出的是通用保护异常。这是因为目前系统在用户态，权限级别3，但是`INT`是系统指令，权限级别为0，因此首先会引发异常13。

Lab3的exercise5

问题

修改`trap_dispatch()`，将缺页异常分发给`page_fault_handler()`。你现在应该能够让`make grade`通过`faultread`，`faultreadkernel`，`faultwrite`和`faultwritekernel`这些测试了。如果这些中的某一个不能正常工作，你应该找找为什么，并且解决它。记住，你可以用`make run-x`或者`make run-x-nox`来直接使JOS启动某个特定的用户程序。

解答

```
static void trap_dispatch(struct Trapframe *tf)
{
    // Handle processor exceptions.
    // LAB 3: Your code here.
    switch (tf->tf_trapno)
    {
    case T_PGFLT:
        page_fault_handler(tf);
        break;
    default:
        // 不知道发生了什么
        print_trapframe(tf);
        if (tf->tf_cs == GD_KT)
            panic("unhandled trap in kernel");
        else
        {
            env_destroy(curenv);
            return;
        }
    }
}
```

就是把`trapno`进行一个判定，加入一个特判，其他的基本不变。

至于测试部分就先不做了，等东西攒多了就一起做（做一次重启一次搞不起）。

引出的问题

之前提出的问题已经可以有一个比较好的解答了。如果说这类情况发生，说明只是wsl产生了问题，换句话说，重启wsl即可。关闭VSCode和其他链接到wsl的程序。进入Windows服务，找到LxssManager，重启即可。

Lab3的exercise6

问题

修改 trap_dispatch() 使断点异常唤起内核监视器。现在，你应该能够让 `make grade` 在 `breakpoint` 测试中成功了。

解答

和上题类似，新加一个case就行。

```
case T_BRKPT:
    monitor(tf);
    break;
```

Lab3的question2

问题

1. 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？
2. 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。

解答

1. 这个问题现在没有打算做，如果想看的话，请注意Lab3的EX4对应的内容是否更新。如果设置break point的`DPL=0`会引发权限错误，由于这里设置`DPL=3`，所以会引发断点。
2. 这个机制很有效的防止一些程序恶意调用指令，进而引发一些危险的错误。

Lab3的exercise7

问题

在内核中断描述符表中为中断向量`T_SYSCALL`添加一个处理函数。你需要编辑`kern/trapentry.S`和`kern/trap.c`的`trap_init()`方法。你也需要修改`trap_dispath()`来将系统调用中断分发给在`kern/syscall.c`中定义的`syscall()`。确保如果系统调用号不合法，`syscall()`返回`-E_INVAL`。你应该读一读并且理解`lib/syscall.c`（尤其是内联汇编例程）来确定你已经理解了系统调用接口。通过调用相应的内核函数，处理在`inc/syscall.h`中定义的所有系统调用。

通过`make run-hello`运行你的内核下的`user/hello`用户程序，它现在应该能在控制台中打印出`hello, world`了，接下来会在用户模式造成一个缺页。如果这些没有发生，也许意味着你的系统调用处理函数不太对。现在应该也能在`make grade`中通过`testbss`这个测试了。

前置知识

GCC内联汇编

```
static inline int32_t syscall
(int num, int check, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4,
uint32_t a5)
{
    int32_t ret;
    asm volatile("int %1\n"
             : "=a" (ret)       // 返回到eax
             : "i" (T_SYSCALL), // 直接操作数48
               "a" (num),       // eax
               "d" (a1),        // edx
               "c" (a2),        // ecx
               "b" (a3),        // ebx
               "D" (a4),        // edi
               "S" (a5)         // esi
             : "cc", "memory");
```

```
    if(check && ret > 0)
        panic("syscall %d returned %d (> 0)", num, ret);
    return ret;
}
```

其中调用的asm代码解释如下：

限定符	意义
“m”, “v”, “o”	内存单元
“r”	任何寄存器
“q”	寄存器eax, ebx, ecx, edx之一
“i”, “h”	直接操作数
“E”, “F”	浮点数
“g”	任意
“a”, “b”, “c”, “d”	分别表示寄存器eax, ebx, ecx, edx
“S”, “D”	寄存器esi, edi
“I”	常数0-31

其中输出值还有一个约束修饰符：

输出修饰符	意义
+	可以读取和写入操作数
=	只能写入操作数
%	如果有必要操作数可以和下一个操作数切换
&	在内联函数完成之前，可以删除和重新使用操作数

根据表格内容，可以看出该内联汇编就是引发一个int中断，中断向量为立即数`T_SYSCALL`，同时对寄存器进行操作。

JOS系统调用产生中断和之前的除零中断的异同。

1. 都在用户态执行代码。
2. 产生错误的时候，`SYSCALL`中断会在`lib/syscall`中产生一些调用，手动产生中断来跳转到内核，但是除零中断是直接被CPU侦测。
3. 之后的处理几乎如出一辙。
4. 最后`SYSCALL`会返回到用户态继续处理，但是除零中断就是kernel接管并强制终止用户态程序。

解答

首先注意把`kern/trap.c`是否修改，使得其支持用户来系统调用。

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

然后很自然的就要在`trap_dispatch`，参考`lib/syscall.c`的参数位置和之前提供的内敛参数表，添加如下代码。系统调用就要调用`syscall`，很显然的吧。

```
// 系统调用，跳转到kern/syscall.c的syscall
// 和其他不一样的是，这里是程序手动产生中断
case 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);
break;
```

回去看`syscall`的实现，发现什么都没有，这个函数在`kern/syscall.c`里面就有

```
int32_t syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3,
                uint32_t a4, uint32_t a5)
{
    panic("syscall not implemented");
    switch (syscallno) {
    default:
        return -E_INVAL;
    }
}
```

补全case就行。记得注释掉`panic`。

```
case SYS_cputs:
    sys_cputs((const char *)a1, a2);
    break;
case SYS_cgetc:
    retVal = sys_cgetc();
    break;
case SYS_env_destroy:
    retVal = sys_env_destroy(a1);
    break;
case SYS_getenvid:
    retVal = sys_getenvid();
    break;
```

然后`make grade`，可以发现`testbss`可以通过。这个exercise已经完成。

Lab3的exercise8

问题

在用户库文件中补全所需要的代码，并启动你的内核。你应该能看到`user/hello`打出了`hello, world`和`i am environment 00001000`。接下来，`user/hello`尝试通过调用`sys_env_destory()`方法退出（在`lib/libmain.c`和`lib/exit.c`）。因为内核目前只支持单用户进程，它应该会报告它已经销毁了这个唯一的进程并进入内核监视器。在这时，你应该能够在`make grade`中通过`hello`这个测试了。

解答

实际上这里(`lib/libmain.c`)只需要修改一行代码。

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

那么问题来了，为什么只需要这一行代码呢？来追踪一下。首先全局搜索`libmain`这个函数，看看从哪里出现的，可以发现在`lib/entry.S`里面的最后一行存在这个调用。

```
args_exist:
    call libmain
1:  jmp 1b
```

再往前面追，可以结合`i386_init`来理解哪里开始调用。首先进入的是kernel环境，之后创建用户环境，通过`env.c`中的函数来装载这个二进制程序，最后通过`env_run`来运行。注意下面代码中`eip`的设定，对于寄存器比较了解的人就知道，一旦这里的`eip`被设定到寄存器中，那么就会自动开始执行程序。

```
lcr3(PADDR(e->env_pgdir));
for (; ph < eph; ph++)
    if (ph->p_type == ELF_PROG_LOAD)
    {
        if (ph->p_filesz > ph->p_memsz)
            panic("load_icode: file size > memory size");
        region_alloc(e, (void *)ph->p_va, ph->p_memsz);
        memcpy((void *)ph->p_va, 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 = elf->e_entry;

```

在`libmain`调用之后，就是`umain`。如果能够正常运行，那么会进行`exit`来销毁环境；如果不能正常运行（包括`SYSCALL`的中断），那么会跳转到kernel帮忙解决或者直接就地销毁，回到父进程。

引出的问题

1. 为什么都有`printf`，但是之前的不需要补全这里的代码，但是这里需要?

   在理解了程序如何被读入之后，实际上就变得非常简单，因为`printf`的参数已经产生了问题，直接被CPU拦下来了。所以前面的代码是没有`SYSCALL`的。

Lab3的exercise9

问题

修改`kern/trap.c`，如果缺页发生在内核模式，应该恐慌。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。读一读`kern/pmap.c`中的`user_mem_assert`并实现同一文件下的`user_mem_check`。

调整`kern/syscall.c`来验证系统调用的参数。启动你的内核，运行`user/buggyhello`(`make run-buggyhello`)。

最后，修改在`kern/kdebug.c`的`debuginfo_eip`，对`usd`，`stabs`，`stabstr`都要调用`user_mem_check`。修改之后，如果你运行`user/breakpoint`，你应该能在内核监视器下输入`backtrace`并且看到调用堆栈遍历到`lib/libmain.c`，接下来内核会缺页并恐慌。是什么造成的内核缺页？你不需要解决这个问题，但是你应该知道为什么会发生缺页。（注：如果整个过程都没发生缺页，说明上面的实现可能有问题。如果在能够看到`lib/libmain.c`前就发生了缺页，可能说明之前某次实验的代码存在问题，也可能是由于GCC的优化，它没有遵守使我们这个功能得以正常工作的函数调用传统，如果你能合理解释它，即使不能看到预期的结果也没有关系。）

解答

由上下文可以知道的是，这讲解的是内存缺页保护。首先根据文体要求，在`page_fault_handler`(`kern/trap.c`)中加入下述代码。

```
// Handle kernel-mode page faults.
// 内核缺页，那是真的慌
// LAB 3: Your code here.
if ((tf->tf_cs & 3) == 0)
    panic("Page fault in kernel-mode");
```

有人可能会问了：这里为什么cs最后两位为3就行了？注意这里是存储的虚拟地址，至于什么时候赋值上去的嘛，使用vscode的C++插件可以很方便的寻找出现在哪里了。

```
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;
```

为什么只有这一处？因为在分配内存的时候初始化都是0，那么这个时候只需要拿出来就是0了，也就不会有初始化的步骤，自然也看不到kernel的`cs`应该在哪里了。不过仅仅通过这些代码就可以弄明白这是虚拟地址还是物理地址（就算不讲这些，也不太可能觉得是物理地址吧）。

接下来查看`kern/pmap.c`的`user_mem_assert`。可以看出的是这一部分进行了内存的检查，如果不合要求的话就会自动销毁`Env`，反过来如果符合要求的话就会啥都不干。

```
void user_mem_assert(struct Env *env, const void *va, size_t len, int perm)
{
    if (user_mem_check(env, va, len, perm | PTE_U) < 0) {
        cprintf("[%08x] user_mem_check assertion failure for "
            "va %08x\n", env->env_id, user_mem_check_addr);
        env_destroy(env);   // may not return
    }
}
```

对着参考和解释，很容易可以给出这样的答案：

```
int user_mem_check(struct Env *env,const void *va,size_t len, int perm)
{
    // 字节对齐
    uintptr_t start_va = ROUNDDOWN((uintptr_t)va, PGSIZE);
    uintptr_t end_va = ROUNDUP((uintptr_t)va + len, PGSIZE);
    for (uintptr_t cur_va = start_va; cur_va < end_va;cur_va += PGSIZE)
    {
        pte_t *cur_pte = pgdir_walk(env->env_pgdir, (void *)cur_va, 0);
```

```
        // 三种情况：页面不存在，页面不满足要求或者进入了kernel
        // 注意：这个是用户的内存检查
        if (cur_pte == NULL || cur_va >= ULIM ||
             (*cur_pte & (perm | PTE_P)) != (perm | PTE_P))
        {
            // 只是起始地址的特判而已
            if (cur_va == start_va)
                user_mem_check_addr = (uintptr_t)va;
            else
                user_mem_check_addr = cur_va;
            return -E_FAULT; // 页面寻找错误
        }
    }
    return 0;
}
```

那写了这个函数是干嘛用的呢？接下来的要求就是`kern/syscall.c`调用这玩意，因此进入看看，发现只有一个地方需要补充，那么就可以给出实现：

```
static void sys_cputs(const char *s, size_t len)
{
    // LAB 3: Your code here.
    // 检查用户是否有读[s, s+len)的权限，如果不能就销毁Env
    user_mem_assert(curenv, s, len, PTE_U);
    // Print the string supplied by the user.
    cprintf("%.*s", len, s);
}
```

接下来要求在`kdebug.c`中添加对于`memcheck`的检查。

```
// 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, (void *)usd, sizeof(struct UserStabData),
    PTE_U) < 0)
    return -1;
// Make sure the STABS and string table memory is valid.
// LAB 3: Your code here.
if (user_mem_check(curenv, (void *)stabs, stab_end - stabs, PTE_U) < 0)
    return -1;
if (user_mem_check(curenv, (void *)stabstr,
    stabstr_end - stabstr, PTE_U) < 0)
    return -1;
```

添加完代码以后运行`user/buggyhello`，`user/breakpoint`分别检查一下效果。这里为了好看点节约了一点内容，自己发现#滑稽。

```
[00001000] user_mem_check assertion failure for va 00000001
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!

Stack backtrace:
  ebp efffff00  eip f0100a63  args 00000001 efffff28 f01d2000 f0106963
             kern/monitor.c:145: monitor+353
  ebp efffff80  eip f0104397  args f01d2000 effffffbc f01066f1 00000092
```

```
             kern/trap.c:248: trap+300
  ebp efffffb0  eip f0104457  args efffffbc 00000000 00000000 eebfdfc0
             kern/syscall.c:83: syscall+0
  ebp eebfdfc0  eip 00800087  args 00000000 00000000 eebfdff0 00800058
             lib/libmain.c:29: libmain+78
Incoming TRAP frame at 0xefffffe7c
kernel panic at kern/trap.c:259: Page fault in kernel-mode
```

虽然说内核确实应当恐慌（实验要求是这么说的）但是为什么恐慌确实应当是一个值得思考的问题（为啥MIT总是要提那种要思考一个星期的"显然"的问题呢？）。

通过观察可以发现的是，`eip`和`ebp`的位置存在一种突变，对照`memlayout`查看得到的是ebp中前三个都是CPU0的栈，第四个是Normal User Stack中，`eip`中前三个是kernel的地址，第四个是用户程序段，说明这之间发生了状态的跳转，那么这个时候查看入口的代码可以发现，如果两者不相同则压入两个0以后开始运行用户态程序。

```
_start:
    // See if we were started with arguments on the stack
    cmpl $USTACKTOP, %esp
    jne args_exist

    // If not, push dummy argc/argv arguments.
    // This happens when we are loaded by the kernel,
    // because the kernel does not know about passing arguments.
    pushl $0
    pushl $0
```

实际上问题就出在这里，如果把`mon_backtrace`中的程序修改成只显示两个参数，那么输出就是正常的。

Lab3的exercise10

问题

启动你的内核，运行`user/evilhello`。进程应该被销毁，内核不应该恐慌，你应该能看到类似下面的输出：

解答

如果你能够通过EX9的话，那么你也应该有大概率通过EX10，如果没有通过，请检查代码的问题在哪里。

下面是我的代码的`make grade`的输出（已经去除编译部分）。系统wsl不一定能一次跑完所有的内容，比如发生之前的问题（暂时没发现过其他的wsl问题），多跑几遍就行了。

```
make[1]: Leaving directory '/home/ivy233/lab'
divzero: OK (4.4s)
softint: OK (2.9s)
badsegment: OK (2.6s)
Part A score: 30/30

faultread: OK (2.9s)
faultreadkernel: OK (2.8s)
faultwrite: OK (2.7s)
faultwritekernel: OK (2.7s)
```

```
breakpoint: OK (2.7s)
testbss: OK (2.6s)
hello: OK (15.5s)
buggyhello: OK (10.4s)
buggyhello2: OK (4.6s)
evilhello: OK (14.0s)
Part B score: 50/50

Score: 80/80
```