

Lab3 实验过程	2
Introduction.....	2
Inline Assembly	3
Part A : User Environments and Exception Handling	3
分析 inc/env.h 文件 :	4
Environment State.....	5
env_tf:	6
env_link:	6
env_id:.....	6
env_parent_id:.....	7
env_type:.....	7
env_status:.....	7
env_pgdir:	8
Allocating the Environments Array	8
Exercise 1.....	8
解答 :	9
Creating and Running Environments.....	9
Exercise 2.....	11
完成 env_init()函数.....	13
完成 env_setup_vm () 函数	14
完成 region_alloc()函数.....	15
完成 load_icode () 函数	16
完成 env_create()函数.....	17
完成 env_run () 函数	18
总结 :	19
Handling Interrupts and Exceptions.....	21
Exercise 3.....	21
Basics of Protected Control Transfer.....	22
Types of Exceptions and Interrupts	23
An Example	24
Nested Exceptions and Interrupts 嵌套异常和中断	24
Setting Up the IDT	25
Exercise 4:.....	26

Part B: Page Faults, Breakpoints Exceptions, and System Calls.....	30
Handling Page Faults	30
Exercise 5.....	30
The Breakpoint Exception	32
Exercise 6.....	32
System calls	33
Exercise 7.....	33
User-mode startup.....	36
Exercise 8.....	36
Page faults and memory protection.....	37
Exercise 9.....	38
Exercise 10.....	40
总结：	41

Lab3 实验过程

Introduction

本实验中，将会实现运行在保护模式下（进程）所需的基本内核功能。丰富 JOS 内核，以设置相关数据结构来跟踪用户环境，创建单个用户环境，将程序 image 加载到其中并开始运行。还需要使 JOS 内核能够处理用户环境发出的任何系统调用，并处理它引起的任何其他异常。

Note：在本实验中，术语“environment”和“process”是可互换的-两者均指允许您运行程序的抽象。为了强调 JOS environment 和 UNIX process 进程提供不同的接口，并且不提供相同的语义，我们引入术语“环境”而不是传统的术语“进程”。

Lab3 新包括的文件：

inc/ env.h	Public definitions for user-mode environments
trap.h	Public definitions for trap handling
syscall.h	Public definitions for system calls from user environments to the kernel
lib.h	Public definitions for the user-mode support library
kern/ env.h	Kernel-private definitions for user-mode environments
env.c	Kernel code implementing user-mode environments
trap.h	Kernel-private trap handling definitions
trap.c	Trap handling code
trapentry.s	Assembly-language trap handler entry-points
syscall.h	Kernel-private definitions for system call handling
syscall.c	System call implementation code
lib/ Makefrag	Makefile fragment to build user-mode library, obj/lib/libjos.a
entry.S	Assembly-language entry-point for user environments
libmain.c	User-mode library setup code called from entry.s
syscall.c	User-mode system call stub functions
console.c	User-mode implementations of putchar and getchar, providing console I/O
exit.c	User-mode implementation of exit
panic.c	User-mode implementation of panic
user/ *	Various test programs to check kernel lab 3 code

Inline Assembly

在本实验中，您可能会发现 GCC 的内联汇编语言功能很有用，尽管也可以不使用它来完成实验。至少，您将需要能够理解我们提供给您的源代码中已经存在的内联汇编语言（“asm”语句）的片段。您可以在课程参考资料页面上找到有关 GCC 内联汇编语言的多种信息来源。

Part A : User Environments and Exception Handling

新的包含文件 inc / env.h 包含 JOS 中用户环境的基本定义。立即阅读。内核使用 Env 数据结构来跟踪每个用户环境（process/environment）。在本实验中，您最初将仅创建一个环境，但是您将需要设计 JOS 内核以支持多个环境。实验 4 将通过允许用户环境派生其他环境来利用此功能。

分析 inc/env.h 文件：

```

typedef int32_t envid_t;

// An environment ID 'envid_t' has three parts:
//
// +-----+-----+-----+
// |0| Uniqueifier | Environment |
// | |
// +-----+-----+-----+
// | | |
// \--- ENVX(eid) ---/
//
// The environment index ENVX(eid) equals the environment's index in the
// 'envs[]' array. The uniqueifier distinguishes environments that were
// created at different times, but share the same environment index.
//
// All real environments are greater than 0 (so the sign bit is zero).
// envid_ts less than 0 signify errors. The envid_t == 0 is special, and
// stands for the current environment.

```

environment 使用一个 ID 号来唯一标识。

环境索引 ENVX (eid) 等于'envs []'数组中环境的索引。 uniqueifier 区分在不同时间创建但共享相同环境的索引。

所有实际环境都大于 0 (因此符号位为零)。

`envid_t` 小于 0 表示错误。`envid_t == 0` 是特殊的，代表当前环境。

```
#define LOG2NENV          10
#define NENV                (1 << LOG2NENV)
#define ENVX(envid)         ((envid) & (NENV - 1))
```

```
// Values of env_status in struct Env
enum {
    ENV_FREE = 0,
    ENV_DYING,
    ENV_RUNNABLE,
    ENV_RUNNING,
    ENV_NOT_RUNNABLE
};
```

这边是 Env 结构体中 env_status 的状态信息。

```
// Special environment types
enum EnvType {
    ENV_TYPE_USER = 0,
};
```

特殊的环境类型。

```

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
};

```

Env 结构体信息。

如您在 kern / env.c 中看到的那样，内核维护着与环境有关的三个主要全局变量：

```

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

```

JOS 启动并运行后，envs 指针指向代表系统中所有环境的 Env 结构数组。在我们的设计中，尽管在任何给定时间通常运行的环境通常要少得多，但是 JOS 内核将最多支持 NENV 同时活动的环境。（NENV 是在 inc / env.h 中 # 定义的常量。）分配它后，envs 数组将为 NENV 个可能的环境包含一个 Env 数据结构的单个实例。

JOS 内核将所有不活动的 Env 结构保留在 env_free_list 上。这种设计可以轻松分配和释放环境，因为只需将其添加到空闲列表中或从中删除。

内核使用 curenv 符号在任何给定时间跟踪当前正在执行的环境。在启动过程中，在运行第一个环境之前，curenv 最初设置为 NULL。

Environment State

```

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
};

```

分析各个字段的含义：

env_tf:

在 inc / trap.h 中定义的该结构在该环境未运行时（即，在内核或其他环境正在运行时）保存该环境的已保存寄存器值。从用户模式切换到内核模式时，内核会保存这些内容，以便以后可以从中断的地方恢复环境。

查看 inc/trap.h 文件：

```
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));
```

PushRegs 结构定义：

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

env_link:

这是指向 env_free_list 上的下一个 Env 的链接。env_free_list 指向列表中的第一个空闲环境。

env_id:

内核在此处存储一个值，该值唯一地标识当前使用此 Env 结构（即使用 envs 数组中的此特定插槽）的环境。用户环境终止后，内核可能会将相同的

Env 结构重新分配给不同的环境（可以看作进程！） - 但是新环境将具有与旧环境不同的 env_id，即使新环境正在重新使用 envs 数组中的相同插槽也是如此。

env_parent_id:

内核在此处存储创建此环境（进程）的环境（进程）的 env_id。这样，环境可以形成“家谱”，这对于制定允许哪些环境对谁执行操作的安全决策很有用。

env_type:

这用于区分特殊环境。对于大多数环境，它将是 ENV_TYPE_USER。在以后的实验中，我们将为特殊的系统服务环境引入更多类型。

```
// Special environment types
enum EnvType {
    ENV_TYPE_USER = 0,
};
```

env_status:

```
// Values of env_status in struct Env
enum {
    ENV_FREE = 0,
    ENV_DYING,
    ENV_RUNNABLE,
    ENV_RUNNING,
    ENV_NOT_RUNNABLE
};
```

ENV_FREE :

指示 Env 结构处于非活动状态，因此处于 env_free_list 上。

ENV_RUNNABLE :

指示 Env 结构表示正在处理器上等待运行的环境。

`ENV_RUNNING` :

表示 Env 结构代表当前正在运行的环境。

`ENV_NOT_RUNNABLE` :

表示 Env 结构代表当前处于活动状态的环境，但当前尚未准备好运行：例如，因为它正在等待来自另一个环境的进程间通信（IPC）。

`ENV_DYING` :

表示 Env 结构代表僵尸环境。 僵尸环境在下一次捕获到内核时将被释放。 在实验 4 之前，我们将不使用此标志。

`env_pgdir`:

此变量保存此环境的页面目录的内核虚拟地址。

像 Unix 进程一样，JOS 环境将“线程”和“地址空间”的概念结合在一起。 线程主要由保存的寄存器（`env_tf` 字段）定义，地址空间由 `env_pgdir` 指向的页面目录和页表定义。 要运行环境（进程），内核必须使用保存的寄存器和适当的地址空间来设置 CPU。

我们的 `struct Env` 类似于 xv6 中的 `proc`。 这两种结构都在 Trapframe 结构中保存环境（即进程）的用户模式寄存器状态。 在 JOS 中，各个环境（进程）不像 xv6 中的进程那样具有自己的内核堆栈。
一次在内核中只能有一个活动的 JOS 环境，因此 JOS 只需要一个内核堆栈。

Allocating the Environments Array

在实验 2 中，您在 `mem_init()` 中为 `pages[]` 数组分配了内存，这是内核用来跟踪哪些页面空闲和哪些页面不空闲的表。 现在，您将需要进一步修改 `mem_init()`，以分配一个类似的 Env 结构数组，称为 `envs`。

Exercise 1

Exercise 1. Modify `mem_init()` in `kern/pmap.c` to allocate and map the `envs` array. This array consists of exactly `NENV` instances of the `Env` structure allocated much like how you allocated the `pages` array. Also like the `pages` array, the memory backing `envs` should also be mapped user read-only at `UENVS` (defined in `inc/memlayout.h`) so user processes can read from this array.

You should run your code and make sure `check_kern_pgdir()` succeeds.

在 `kern / pmap.c` 中修改 `mem_init ()` 以分配和映射 `envs` 数组。 该数组完全由分配的 `Env` 结构的 `NENV` 实例组成，就像分配页面数组的方式一样。 与页面数组一样，内存支持环境也应该在 `UENVS`（在 `inc / memlayout.h` 中定义）上以只读方式映射为用户，以便用户进程可以从该数组读取。

您应该运行代码，并确保 `check_kern_pgdir ()` 成功。

解答：

和分配页面数组一样的方式，设置 `Env` 数组。

```
//////////  
// 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,sizeof(struct Env)*NENV);|
```

设置虚拟地址和物理地址的映射关系：

```
//////////  
// 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);|
```

测试通过！

Creating and Running Environments

您现在将在 `kern / env.c` 中编写运行用户环境所需的代码。 因为我们还没有文件系统，所以我们将设置内核以加载嵌入在内核本身中的静态二进制映像。
`JOS` 将此二进制文件作为 `ELF` 可执行映像嵌入内核。

Lab 3 GNUmakefile 在 obj / user / 目录中生成许多二进制映像。 如果您查看 kern / Makefrag，您会注意到一些魔术，**可以将这些二进制文件直接链接到内核可执行文件，就像它们是.o 文件一样。** 链接器命令行上的**-b binary** 选项使这些文件链接为“原始”未解释的二进制文件，而不是作为编译器生成的常规.o 文件。（就链接程序而言，这些文件根本不必是 ELF 图像-它们可以是任何东西，例如文本文件或图片！）如果您在构建内核之后查看 obj / kern / kernel.sym，您会注意到，链接器“神奇地”产生了许多有趣的符号，它们的名称晦涩难懂，例如 _binary_obj_user_hello_start, _binary_obj_user_hello_end 和 _binary_obj_user_hello_size。链接器通过处理二进制文件的文件名来生成这些符号名。这些符号为常规内核代码提供了一种引用嵌入式二进制文件的方式。

在 obj/user 目录中生成的文件：

```
dingxiao@ubuntu:~/MIT6.828/lab/obj/user$ ls
badsegment      buggyhello.d    faultreadkernel.asm  hello.asm
badsegment.asm   buggyhello.o    faultreadkernel.d  hello.d
badsegment.d     buggyhello.sym  faultreadkernel.o  hello.o
badsegment.o     divzero        faultreadkernel.sym hello.sym
badsegment.sym   divzero.asm   faultread.o          softint
breakpoint       divzero.d     faultread.sym        softint.asm
breakpoint.asm   divzero.o     faultwrite           softint.d
breakpoint.d     divzero.sym   faultwrite.asm      softint.o
breakpoint.o     evilhello     faultwrite.d        softint.sym
breakpoint.sym   evilhello.asm  faultwritekernel   testbss
buggyhello       evilhello.d   faultwritekernel.asm testbss.asm
buggyhello2      evilhello.o   faultwritekernel.d  testbss.d
buggyhello2.asm  evilhello.sym  faultwritekernel.o  testbss.o
buggyhello2.d    faultread     faultwritekernel.sym testbss.sym
buggyhello2.o    faultread.asm  faultwrite.o        faultwrite.sym
buggyhello2.sym  faultread.d   faultwrite.kernel  hello
buggyhello.asm   faultreadkernel hello
```

在 kern/Makefrag 文件中：

```
# Binary program images to embed within the kernel.
# Binary files for LAB3
KERN_BINFILES :=      user/hello \
                      user/buggyhello \
                      user/buggyhello2 \
                      user/evilhello \
                      user/testbss \
                      user/divzero \
                      user/breakpoint \
                      user/softint \
                      user/badsegment \
                      user/faultread \
                      user/faultreadkernel \
                      user/faultwrite \
                      user/faultwritekernel
```

在 kern / init.c 中的 i386_init () 中，您将看到在环境（进程）中运行这些二进制映像之一的代码。但是，设置用户环境的关键功能还不完善。您将需要填写它们。

Exercise 2

Exercise 2. In the file env.c, finish coding the following functions:

```
env_init()
    Initialize all of the Env structures in the envs array and add them to the env_free_list. Also calls env_init_percpu, which
    configures the segmentation hardware with separate segments for privilege level 0 (kernel) and privilege level 3 (user).
env_setup_vm()
    Allocate a page directory for a new environment and initialize the kernel portion of the new environment's address space.
region_alloc()
    Allocates and maps physical memory for an environment
load_icode()
    You will need to parse an ELF binary image, much like the boot loader already does, and load its contents into the user
    address space of a new environment.
env_create()
    Allocate an environment with env_alloc and call load_icode to load an ELF binary into it.
env_run()
    Start a given environment running in user mode.
```

As you write these functions, you might find the new cprintf verb %e useful -- it prints a description corresponding to an error code. For example,

```
r = -E_NO_MEM;
panic("env_alloc: %e", r);
```

will panic with the message "env_alloc: out of memory".

题目：

在文件 env.c 中，完成对以下功能的编码：

`env_init ()`

初始化 envs 数组中的所有 Env 结构，并将它们添加到 env_free_list。还调用 env_init_percpu，它将为特权级别 0（内核）和特权级别 3（用户）配置具有单独段的分段硬件。

`env_setup_vm ()`

为新环境（进程）分配页目录，并初始化新环境的地址空间的内核部分。

`region_alloc ()`

分配和映射环境的物理内存

`load_icode ()`

您将需要像启动加载程序一样解析 ELF 二进制映像，并将其内容加载到新环境的用户地址空间中。

`env_create ()`

使用 env_alloc 分配环境，然后调用 load_icode 将 ELF 二进制文件加载到其中。

```
env_run ()
```

启动以用户模式运行的给定环境（进程）。

在编写这些函数时，您可能会发现新的 `cprintf` 动词 `%e` 很有用-它打印与错误代码相对应的描述。例如，

```
r = -E_NO_MEM;  
panic ("env_alloc : %e", r);
```

将对消息“`env_alloc`：内存不足”panic！

下面是代码的调用图，直至调用用户代码为止。确保您了解每个步骤的目的。

- `start (kern/entry.s)`
- `i386_init (kern/init.c)`
 - `cons_init`
 - `mem_init`
 - `env_init`
 - `trap_init (still incomplete at this point)`
 - `env_create`
 - `env_run`
 - `env_pop_tf`

完成后，您应该编译内核并在 QEMU 下运行它。如果一切顺利，您的系统应进入用户空间并执行 `hello` 二进制文件，直到使用 `int` 指令进行系统调用为止。那时会有麻烦，因为 JOS 尚未设置硬件以允许从用户空间到内核的任何类型的转换。**当 CPU 发现未将其设置为处理该系统调用中断时，它将生成一个常规保护异常**，发现它不能处理该异常，生成一个双重故障异常，发现它也不能处理该异常，最终放弃所谓的“三重故障”。**通常，然后您会看到 CPU 复位和系统重启。**尽管这对遗留应用程序很重要（有关原因的说明，请参阅此博客文章），但这对于内核开发是一件痛苦的事，因此，使用 6.828 修补的 QEMU，您将看到寄存器转储和“三重错误”信息。

我们将很快解决这个问题，但是现在我们可以使用调试器检查我们是否进入用户模式。使用 `make qemu-gdb` 并在 `env_pop_tf` 处设置 GDB 断点，该断点应该是您在实际进入用户模式之前命中的最后一个函数。使用 `si` 单步完成此功能；处理器应在 `iret` 指令之后进入用户模式。然后，您应该在用户环境的可执行文件中看到第一条指令，它是 `lib / entry.S` 中标签开头的 `cmpl` 指令。现

在使用 b * 0x ... 在 hello 的 sys_cputs () 中的 int \$ 0x30 处设置一个断点（有关用户空间地址， 请参见 obj / user / hello.asm）。此 int 是系统调用，用于向控制台显示字符。如果无法执行到最大的整数，则地址空间设置或程序加载代码有问题；返回并修复它，然后继续。

解答：

完成 env_init() 函数

首先，查看 env.c 文件中的各个字段的含义：

```
// Global descriptor table.  
//  
// Set up global descriptor table (GDT) with separate segments for  
// kernel mode and user mode. Segments serve many purposes on the x86.  
// We don't use any of their memory-mapping capabilities, but we need  
// them to switch privilege levels.  
//  
// The kernel and user segments are identical except for the DPL.  
// To load the SS register, the CPL must equal the DPL. Thus,  
// we must duplicate the segments for the user and the kernel.  
//  
// In particular, the last argument to the SEG macro used in the  
// definition of gdt specifies the Descriptor Privilege Level (DPL)  
// of that descriptor: 0 for kernel and 3 for user.  
//  
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 trap_init_percpu()  
    [GD_TSS0 >> 3] = SEG_NULL  
};  
  
struct Pseudodesc gdt_pd = {  
    sizeof(gdt) - 1, (unsigned long) gdt  
};
```

全局描述符表。

使用内核模式和用户模式的单独段设置全局描述符表（GDT）。段在 x86 上有多种用途。我们不使用它们的任何内存映射功能，但需要它们切换特权级别。除了 DPL，内核和用户段相同。要加载 SS 寄存器，CPL 必须等于 DPL。因此，我们必须为用户和内核复制这些段。特别是，在 gdt 定义中使用的 SEG 宏的最后一个参数指定了该描述符的描述符特权级别（DPL）：内核为 0，用户为 3。

```
// RETURNS
//   0 on success, -E_BAD_ENV on error.
//   On success, sets *env_store to the environment.
//   On error, sets *env_store to NULL.
//
int
envid2env(envid_t envid, struct Env **env_store, bool checkperm)
```

这是将 envid 转换为 env 指针的函数，成功返回 0，否则返回-E_BAD_ENV，如果成功则设置 env_store 指针指向这个环境。

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

完成 env_setup_vm () 函数

该函数的作用：初始化环境 e 的内核虚拟内存布局。分配页面目录，相应地设置 e->env_pgdir，并初始化新环境的地址空间的内核部分。请勿（尚未）将任何内容映射到环境的虚拟地址空间的用户部分。

```

static int
env_setup_vm(struct Env *e)
{
    int i;
    struct PageInfo *p = NULL;

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

    // Now, set e->env_pgdир and initialize the page directory.
    //
    // Hint:
    //   - The VA space of all envs is identical above UTOP
    //     (except at UVPT, which we've set below).
    //   - See inc/memlayout.h for permissions and layout.
    //   - Can you use kern_pgdир as a template? Hint: Yes.
    //   - (Make sure you got the permissions right in Lab 2.)
    //   - The initial VA below UTOP is empty.
    //   - You do not need to make any more calls to page_alloc.
    //   - Note: In general, pp_ref is not maintained for
    //     physical pages mapped only above UTOP, but env_pgdир
    //     is an exception -- you need to increment env_pgdир's
    //     pp_ref for env_free to work correctly.
    //   - The functions in kern/pmap.h are handy.

    // LAB 3: Your code here.
    e->env_pgdир=(pde_t*)page2kva(p);
    p->pp_ref++;
    memcpy(e->env_pgdир,kern_pgdир,PGSIZE);

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

    return 0;
}

```

完成 region_alloc() 函数

为环境 env 分配物理内存的 len 字节，并将其映射到环境地址空间中的虚拟地址 va。不为零或以任何方式初始化映射的页面。页面应可由用户和内核写入。

如果任何分配尝试失败均会出现 panic。

```

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.)
    //
    // 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!)
    void* begin=ROUNDDOWN((uint32_t)va,PGSIZE);
    void* end=ROUNDUP((uint32_t)(va+len),PGSIZE);
    while(begin<end){
        struct PageInfo *pp=page_alloc(0); //分配一个物理页
        if(!pp){
            panic("region_alloc failed!\n");
        }
        page_insert(e->env_pgdир,pp,begin,PTE_W|PTE_U); //这边调用lab2中的函数，设置页目录和页表,
    //建立线性地址和物理地址的映射关系
        begin+=PGSIZE; //更新线性地址
    }
}

```

完成 load_icode () 函数

该函数为用户进程设置初始程序二进制，堆栈和处理器标志。

在运行第一个用户模式环境之前，仅在内核初始化期间调用此函数。

此函数从 ELF 程序头中指示的适当虚拟地址开始，将所有可加载段从 ELF 二进制映像加载到环境的用户内存中。

同时，它会将这些段的任何部分清零，这些部分在程序头中被标记为已映射但实际上未出现在 ELF 文件中-即程序的 bss 部分。所有这些都与我们的引导加载程序非常相似，除了引导加载程序还需要从磁盘读取代码。看一下 boot / main.c 以获得想法。

最后，此函数为程序的初始堆栈映射一页。如果遇到问题，load_icode 会发生 panic。

```
void
bootmain(void)
{
    struct Proghdr *ph, *eph;
    // read 1st page off disk
    readseg((uint32_t) ELFHDR, SECTSIZE*8, 0);
    // is this a valid ELF?
    if (ELFHDR->e_magic != ELF_MAGIC)
        goto bad;
    // load each program segment (ignores ph flags)
    ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
    eph = ph + ELFHDR->e_phnum;
    for (; ph < eph; ph++)
        // p_pa is the load address of this segment (as well
        // as the physical address)
        readseg(ph->p_pa, ph->p_memsz, ph->p_offset);
    // call the entry point from the ELF header
    // note: does not return!
    ((void (*)(void)) (ELFHDR->e_entry))();
bad:
    outw(0x8A00, 0x8A00);
    outw(0x8A00, 0x8E00);
    while (1)
        /* do nothing */;
}
```

查看 boot loader 中加载内核 ELF 文件的方法。

```

static void
load_icode(struct Env *e, uint8_t *binary)
{
    // Hints:
    // Load each program segment into virtual memory
    // at the address specified in the ELF segment header.
    // You should only load segments with ph->p_type == ELF_PROG_LOAD.
    // Each segment's virtual address can be found in ph->p_va
    // and its size in memory can be found in ph->p_memsz.
    // The ph->p_filesz bytes from the ELF binary, starting at
    // 'binary + ph->p_offset', should be copied to virtual address
    // ph->p_va. Any remaining memory bytes should be cleared to zero.
    // (The ELF header should have ph->p_filesz <= ph->p_memsz.)
    // Use functions from the previous lab to allocate and map pages.
    //
    // All page protection bits should be user read/write for now.
    // ELF segments are not necessarily page-aligned, but you can
    // assume for this function that no two segments will touch
    // the same virtual page.
    //
    // You may find a function like region_alloc useful.
    //
    // Loading the segments is much simpler if you can move data
    // directly into the virtual addresses stored in the ELF binary.
    // So which page directory should be in force during
    // this function?
    //
    // You must also do something with the program's entry point,
    // 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 *ELFHDR=(struct Elf*)binary;
    struct Proghdr *ph;
    int ph_num;
    if(ELFHDR->e_magic != ELF_MAGIC){
        panic("binary is not ELF Format!\n");
    }
    ph = (struct Proghdr *)((uint8_t*)ELFHDR+ELFHDR->e_phoff);
    ph_num=ELFHDR->e_phnum;

    lcr3(PADDR(e->env_pgdir)); //e->env_pgdir与kern_pgdir除了PDX(UVPT)这一项不同，其余都一样，但是后面会给env_pgdir增加映射关系

    for(int i=0;i<ph_num;++i){
        if(ph[i].p_type==ELF_PROG_LOAD){ //只加载LOAD类型的段segment
            region_alloc(e,(void*)ph[i].p_va,ph[i].p_memsz); //分配物理地址并映射
            memset((void*)ph[i].p_va,0,ph[i].p_memsz);
            memcpy((void*)ph[i].p_va,binary+ph[i].p_offset,ph[i].p_filesz); //搜索BSS段
        }
    }

    e->env_tf.tf_eip=ELFHDR->e_entry; //加载完段后，需要设置程序的第一条指令的位置
    lcr3(PADDR(e->env_pgdir));
    // Now map one page for the program's initial stack
    // at virtual address USTACKTOP - PGSIZE.
    region_alloc(e,(void*)(USTACKTOP-PGSIZE),PGSIZE); //为用户环境分配栈空间。
    //

    // LAB 3: Your code here.
}

```

完成 env_create() 函数

该函数使用 env_alloc 分配新的 env，使用 load_icode 将指定的 elf 二进制文件加载到其中，并设置其 env_type。

仅在内核初始化期间，运行第一个用户模式环境之前调用此函数。

新环境的父 ID 设置为 0。

```

void
env_create(uint8_t *binary, enum EnvType type)
{
    // LAB 3: Your code here.
    struct Env *e;
    int res;
    if((res=env_alloc(&e,0))!=0){      //创建一个env
        panic("env create failed!\n");
    }

    load_icode(e,binary);
    e->env_type=type;
}

```

完成 env_run () 函数

上下文从 curenv 切换到 env e。如果这是对 env_run 的首次调用，则 curenv 为 NULL。

```

void
env_run(struct Env *e)
{
    // Step 1: If this is a context switch (a new environment is running):
    // 1. Set the current environment (if any) back to
    //     ENV_RUNNABLE if it is ENV_RUNNING (think about
    //     what other states it can be in),
    // 2. Set 'curenv' to the new environment,
    // 3. Set its status to ENV_RUNNING,
    // 4. Update its 'env_runs' counter,
    // 5. Use lcr3() to switch to its address space.
    // Step 2: Use env_pop_tf() to restore the environment's
    // registers and drop into user mode in the
    // environment.

    // Hint: This function loads the new environment's state from
    // e->env_tf. Go back through the code you wrote above
    // and make sure you have set the relevant parts of
    // e->env_tf to sensible values.

    // LAB 3: Your code here.
    if(curenv!=NULL && curenv->env_status == ENV_RUNNING){
        curenv->env_status=ENV_RUNNABLE;
    }
    curenv=e;
    curenv->status=ENV_RUNNING;
    curenv->env_runs++;
    lcr3(PADDR(curenv->env_pgdir));

    env_pop_tf(&curenv->env_tf);
    panic("env_run not yet implemented");
}

```

观察 env_pop_tf() 函数：

使用 “ iret” 指令在 Trapframe 中恢复寄存器值。

这将退出内核并开始执行某些环境的代码。

```
// Restores the register values in the Trapframe with the 'iret' instruction.  
// This exits the kernel and starts executing some environment's code.  
//  
// This function does not return.  
//  
void env_pop_tf(struct Trapframe *tf)  
{  
    asm volatile(  
        "\tmovl %0,%esp\n"      //将%esp指向tf地址处  
        "\tpopl\n"              //弹出Trapframe结构中的tf_regs值到通用寄存器  
        "\tpopl %es\n"          //弹出Trapframe结构中的tf_es值到es寄存器  
        "\tpopl %ds\n"          //弹出Trapframe结构中的tf_ds值到ds寄存器  
        "\taddl $0x8,%esp\n" /* skip tf_trapno and tf_errcode */  
        "\tiret\n"               //中断返回指令，具体动作如下：从Trapframe结构中依次弹出tf_eip,tf_cs,tf_eflags,tf_esp,tf_ss到相应  
    寄存器  
        : : "g" (tf) : "memory");  
    panic("iret failed"); /* mostly to placate the compiler */  
}
```

这个函数的弹出寄存器的动作，弹出顺序与 Trapframe 结构体中的顺序是一致的。

```
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));
```



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

总结：

这些函数的调用关系：

```
env_create()  
    -->env_alloc()  
        -->env_setup_vm()  
    -->load_icode()  
        -->region_alloc()
```

重新编译内核，在 env_pop_tf 函数中打上断点，单步调试。

```
+ symbol-file obj/kern/kernel
(gdb) b env_pop_tf
Breakpoint 1 at 0xf0102dc4: file kern/env.c, line 470.
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0xf0102dc4 <env_pop_tf>:      push    %ebp
```

在执行 iret 之前，查看寄存器存储的数据。执行 iret 之后，查看寄存器存储的数据。

```
dingxiao@ubuntu:~/MIT6.828/lab
0xf0102dcf 471         asm volatile(
(gdb)
=> 0xf0102dd0 <env_pop_tf+12>: add    $0x8,%esp
0xf0102dd0 471         asm volatile(
(gdb)
=> 0xf0102dd3 <env_pop_tf+15>: iret
0xf0102dd3 471         asm volatile(
(gdb) info registers
eax          0x0      0
ecx          0x0      0
edx          0x0      0
ebx          0x0      0
esp          0xf01be030 0xf01be030
ebp          0x0      0x0
esi          0x0      0
edi          0x0      0
eip          0xf0102dd3 0xf0102dd3 <env_pop_tf+15>
eflags        0x96     [ PF AF SF ]
cs           0x8      8
ss           0x10     16
ds           0x23     35
es           0x23     35
fs           0x23     35
gs           0x23     35
(gdb) si
=> 0x8000020: cmp    $0xeeebfe000,%esp
0x80000020 in ?? ()
(gdb) info registers
eax          0x0      0
ecx          0x0      0
edx          0x0      0
ebx          0x0      0
esp          0xeeebfe000 0xeeebfe000
ebp          0x0      0x0
esi          0x0      0
edi          0x0      0
eip          0x8000020 0x8000020
eflags        0x2     [ ]
cs           0x1b     27
ss           0x23     35
ds           0x23     35
es           0x23     35
fs           0x23     35
gs           0x23     35
(gdb)
```

发现 cs 从一开始的 0x8 变为 0x1b，在 memlayout.h 文件中堆 GD_UT 设置为 0x18，加上用户权限是 0x3，则选择子为 0x1b。

```
// Global descriptor numbers
#define GD_KT    0x08      // kernel text
#define GD_KD    0x10      // kernel data
#define GD_UT    0x18      // user text
#define GD_UD    0x20      // user data
#define GD_TSS0  0x28      // Task segment selector for CPU 0
```

上面的寄存器都是在 env_alloc 函数中设置好的。

```

// Set up appropriate initial values for the segment registers.
// GD_UD is the user data segment selector in the GDT, and
// GD_UT is the user text segment selector (see inc/memlayout.h).
// The low 2 bits of each segment register contains the
// Requestor Privilege Level (RPL); 3 means user mode. When
// we switch privilege levels, the hardware does various
// checks involving the RPL and the Descriptor Privilege Level
// (DPL) stored in the descriptors themselves.
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;
// You will set e->env_tf.tf_eip later.

```

这里就是设置 `e->env_tf` 结构的地方，设置完后再执行 `iret` 指令，寄存器就会加载这些设置了的值。

在 `obj/user/hello.asm` 找到断点位置。第一个系统调用是 `int $0x30`

```

(gdb) b *0x800a1c
Breakpoint 2 at 0x800a1c
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
=> 0x800a6b:    int      $0x30
0x00800a6b in ?? ()
(gdb) █

```

Handling Interrupts and Exceptions

这时，用户空间中的第一个 `int $0x30` 系统调用指令还不起作用：一旦处理器进入用户模式，就无法退出。现在，您将需要实现基本的异常和系统调用处理，以便内核可以从用户模式代码中恢复对处理器的控制。您应该做的第一件事是完全熟悉 x86 中断和异常机制。

Exercise 3

Exercise 3. Read [Chapter 9, Exceptions and Interrupts](#) in the 80386 Programmer's Manual (or Chapter 5 of the [IA-32 Developer's Manual](#)), if you haven't already.

这个练习是了解 X86 的中断和异常机制。

在本实验中，我们通常遵循 Intel 的术语来表示中断，异常等。但是，诸如异常，陷阱，中断，故障和中止之类的术语在整个体系结构或操作系统中没有标准含义，并且经常被使用而无视它们在诸如 x86 之类的特定体系结构上的细微差别。当您在本练习之外看到这些术语时，其含义可能会略有不同。

Basics of Protected Control Transfer

异常和中断都是“受保护的控制传递”，它们导致处理器从用户模式切换到内核模式 ($CPL = 0$)，而没有给用户模式代码任何干扰内核或其他环境功能的机会。用英特尔的术语来说，中断是受保护的控制传输，它是由通常在处理器外部的异步事件引起的，例如外部设备 I/O 活动的通知。相反，有一种异常是由当前运行的代码同步引起的受保护的控制传输，例如由于除 0 错误或无效的内存访问。

为了确保这些受保护的控制传递实际上是受到了保护，所以设计了处理器的中断/异常机制，以使在发生中断或异常时当前正在运行的代码不会随意选择进入内核的位置或方式。相反，处理器确保只能在精心控制的条件下才能输入内核。在 x86 上，两种机制可以共同提供这种保护：

1) 中断描述符表。处理器确保中断和异常只能使内核进入由内核本身确定的几个明确定义的入口点。（而不是采用中断或异常时运行的代码）。

x86 允许多达 256 个不同的中断或异常入口点进入内核，每个入口点都有不同的中断向量。向量是介于 0 到 255 之间的数字。中断的向量由中断源决定：不同的设备，错误条件以及对内核的应用程序请求会使用不同的向量生成中断。CPU 使用该向量作为处理器中断描述符表 (IDT) 的索引，**该中断描述符表由内核在内核专用内存中设置，就像 GDT 一样。**从此表中的适当条目中，处理器将加载：

- a) 要加载到指令指针 (EIP) 寄存器中的值，该值指向为处理这种异常而指定的内核代码。
- b) 要加载到代码段 (CS) 寄存器中的值，该值在 bits 0-1 中包含运行异常处理程序的特权级别。（在 JOS 中，所有异常都在内核模式下，特权级别 0 处理。）

2) 任务状态段 (TSS)。处理器需要在中断或异常发生之前保存旧处理器状态的位置，例如在处理器调用异常处理程序之前保存 EIP 和 CS 的原始值，以便异常处理程序以后可以恢复该旧状态并恢复被中断的状态。代码从它停止的地方开始。但是，必须反过来保护旧处理器状态的此保存区域不受非特权用户模式代码的影响；否则，错误或恶意的用户代码可能会损害内核。

因此，当 x86 处理器执行中断或陷阱，从而导致特权级别从用户模式更改为内核模式时，它还会切换到内核内存中的堆栈。称为任务状态段 (TSS) 的结构指定了段选择器和该堆栈所在的地址。处理器（在这个新堆栈上）推送 SS, ESP, EFLAGS, CS, EIP 和可选的错误代码。然后，它从中断描述符中加载 CS 和 EIP，并将 ESP 和 SS 设置为引用新堆栈。

尽管 TSS 很大，并且可能有多种用途，但 JOS 仅使用它来定义处理器从用户模式转换到内核模式时应切换到的内核堆栈。由于 JOS 中的“内核模式”在 x86 上的特权级别为 0，因此处理器在进入内核模式时使用 TSS 的 ESP0 和 SS0 字段来定义内核堆栈。JOS 不使用任何其他 TSS 字段。

Types of Exceptions and Interrupts

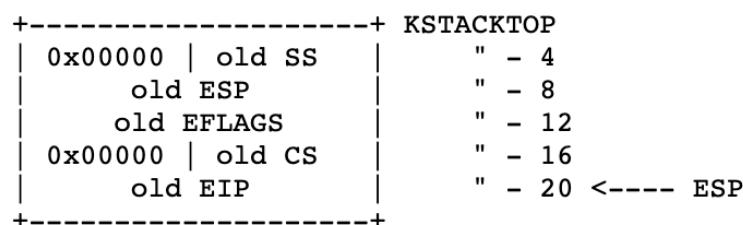
x86 处理器可以在内部生成的所有同步异常都使用 0 到 31 之间的中断向量，因此映射到 IDT 条目 0-31。例如，页面错误总是通过向量 14 引起异常。大于 31 的中断向量仅由软件中断使用，该中断可以由 int 指令生成，也可以由外部设备在需要注意时引起的异步硬件中断生成。

在本节中，我们将扩展 JOS 以处理向量 0-31 中内部生成的 x86 异常。在下一节中，我们将使 JOS 处理软件中断向量 48 (0x30)，JOS（相当任意）将其用作系统调用中断向量。在实验 4 中，我们将扩展 JOS 以处理外部生成的硬件中断，例如时钟中断。

An Example

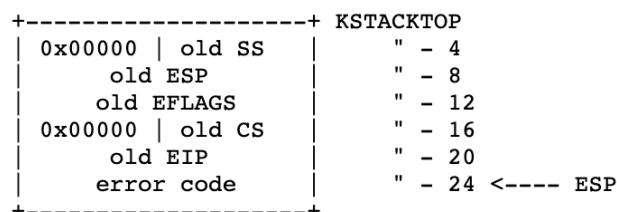
让我们将这些片段放在一起，并通过示例进行追溯。假设处理器正在用户环境中执行代码，并遇到试图除以零的除法指令。

- 1) 处理器切换到由 TSS 的 SS0 和 ESP0 字段定义的堆栈，该堆栈在 JOS 中将分别保存值 GD_KD 和 KSTACKTOP。
- 2) 处理器将异常参数从地址 KSTACKTOP 开始推入内核堆栈：



- 3) 因为我们正在处理除法错误，即 x86 上的中断向量 0，所以处理器读取 IDT 条目 0 并将 CS : EIP 设置为指向该条目描述的处理函数。
- 4) 处理程序函数控制并处理异常，例如通过终止用户环境。

对于某些类型的 x86 异常，除了上面的“标准”五个字以外，处理器还将另一个包含错误代码的字压入堆栈。重要的示例是 14 号页面错误异常。请参阅 80386 手册，以确定处理器将哪个错误号推送到错误代码，以及在这种情况下错误代码的含义。当处理器推送错误代码时，从用户模式进入时，堆栈在异常处理程序的开头看起来如下：



Nested Exceptions and Interrupts 嵌套异常和中断

处理器可以接受来自内核和用户模式的异常和中断。但是，只有从用户模式进入内核时，x86 处理器才会自动切换堆栈，然后再将其旧的寄存器状态推

入堆栈并通过 IDT 调用适当的异常处理程序。如果在发生中断或异常时处理器已经处于内核模式（CS 寄存器的低 2 位已经为零），则 CPU 会将更多的值压入同一内核堆栈。这样，内核可以优雅地处理由内核本身内的代码引起的嵌套异常。该功能是实现保护的重要工具，我们将在后面的系统调用部分中看到。

如果处理器已经处于内核模式并采用嵌套异常，则由于它不需要切换堆栈，因此不会保存旧的 SS 或 ESP 寄存器。因此，对于不推送错误代码的异常类型，内核堆栈在进入异常处理程序时如下所示：

		<---- old ESP
old EFLAGS		" - 4
0x00000	old CS	" - 8
old EIP		" - 12

对于推送错误代码的异常类型，处理器会像以前一样在旧 EIP 之后立即推送错误代码。

处理器的嵌套异常功能有一个重要警告。如果处理器已经在内核模式下时发生异常，并且由于缺少堆栈空间等任何原因而无法将其旧状态推送到内核堆栈，则处理器无法执行任何操作来恢复，因此只需重置自身即可。不用说，应该对内核进行设计，以免发生这种情况。

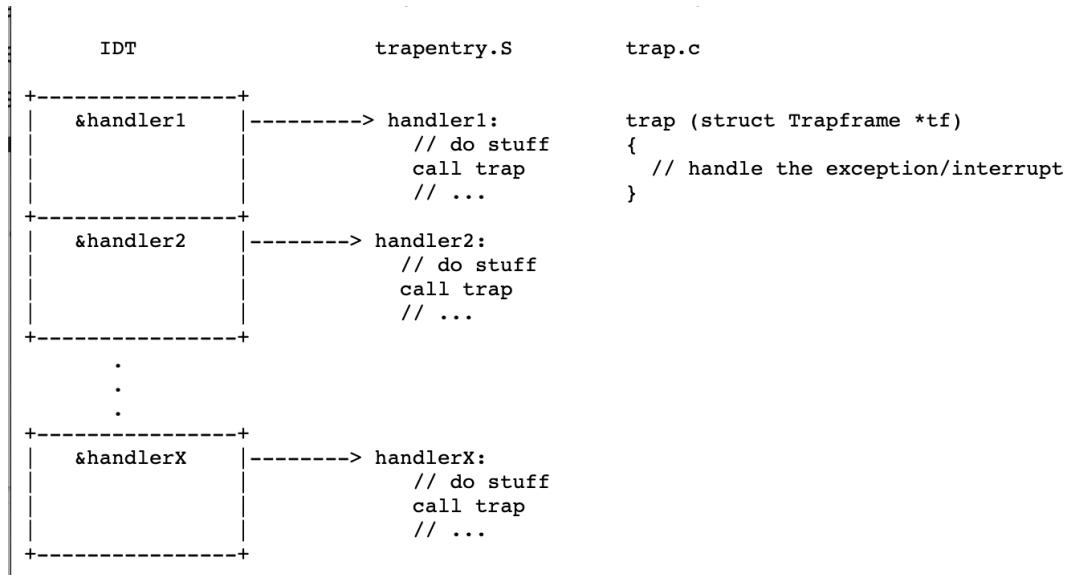
Setting Up the IDT

现在，您应该具有设置 IDT 和处理 JOS 中的异常所需的基本信息。现在，您将设置 IDT 以处理中断向量 0-31（处理器异常）。在本实验的后面，我们将处理系统调用中断，并在以后的实验中添加中断 32-47（设备 IRQ）。

头文件 inc / trap.h 和 kern / trap.h 包含与中断和异常相关的重要定义，您需要熟悉这些定义。文件 kern / trap.h 包含对内核严格专有的定义，而 inc / trap.h 包含对用户级程序和库也可能有用的定义。

注意：Intel 定义 0-31 范围内的某些例外情况为保留。由于它们永远不会由处理器生成，因此如何处理它们并不重要。

您应该实现的总体控制流程如下所示：



Each exception or interrupt should have its own handler in `trapentry.s` and `trap_init()` should initialize the IDT with the addresses of these handlers. Each of the handlers should build a `struct Trapframe` (see `inc/trap.h`) on the stack and call `trap()` (in `trap.c`) with a pointer to the `Trapframe`. `trap()` then handles the exception/interrupt or dispatches to a specific handler function.

所以整个操作系统的中断控制流程为：

1. `trap_init()` 先将所有中断处理函数的起始地址放到中断向量表 IDT 中。
2. 当中断发生时，不管是外部中断还是内部中断，处理器捕捉到该中断，进入内核态，根据中断号去查询中断描述符表，找到对应的表项
3. 保存被中断的程序的上下文到内核栈中，调用这个表项中指明的中断处理函数。
4. 执行中断处理函数。
5. 执行完成后，恢复被中断的进程的上下文，返回用户态，继续运行这个进程。

Exercise 4:

Exercise 4. Edit `trapentry.S` and `trap.c` and implement the features described above. The macros `TRAPHANDLER` and `TRAPHANDLER_NOEC` in `trapentry.s` should help you, as well as the `T_*` defines in `inc/trap.h`. You will need to add an entry point in `trapentry.s` (using those macros) for each trap defined in `inc/trap.h`, and you'll have to provide `_alltraps` which the `TRAPHANDLER` macros refer to. You will also need to modify `trap_init()` to initialize the `idt` to point to each of these entry points defined in `trapentry.s`; the `SETGATE` macro will be helpful here.

Your `_alltraps` should:

1. push values to make the stack look like a struct Trapframe
2. load `GD_KD` into `%ds` and `%es`
3. `pushl %esp` to pass a pointer to the Trapframe as an argument to `trap()`
4. `call trap` (can `trap` ever return?)

Consider using the `pushal` instruction; it fits nicely with the layout of the `struct Trapframe`.

Test your trap handling code using some of the test programs in the `user` directory that cause exceptions before making any system calls, such as `user/divzero`. You should be able to get `make grade` to succeed on the `divzero`, `softint`, and `badsegment` tests at this point.

问题：

编辑 `trapentry.S` 和 `trap.c` 并实现上述功能。

解答：

首先查看 `trapentry.S` 文件：

```
/* TRAPHANDLER defines a globally-visible function for handling a trap.
 * It pushes a trap number onto the stack, then jumps to _alltraps.
 * Use TRAPHANDLER for traps where the CPU automatically pushes an error code.
 *
 * You shouldn't call a TRAPHANDLER function from C, but you may
 * need to _declare_ one in C (for instance, to get a function pointer
 * during IDT setup). You can declare the function with
 * void NAME();
 * where NAME is the argument passed to TRAPHANDLER.
 */
#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

/* Use TRAPHANDLER_NOEC for traps where the CPU doesn't push an error code.
 * It pushes a 0 in place of the error code, so the trap frame has the same
 * format in either case.
 */
#define TRAPHANDLER_NOEC(name, num)
    .globl name;
    .type name, @function;
    .align 2;
    name:
    pushl $0;
    pushl $(num);
    jmp _alltraps

.text
```

这两个宏的作用是创建 0 ~ 16 号中断的中断处理函数。根据中断是否有中断错误码。最终函数都会调用 `_alltraps`。这个按照题目要求进行设定。

```
/*
 * Lab 3: Your code here for generating entry points for the different traps.
 */
TRAPHANDLER_NOEC(t_divide, T_DIVIDE)
TRAPHANDLER_NOEC(t_debug, T_DEBUG)
TRAPHANDLER_NOEC(t_nmi, T_NMI)
TRAPHANDLER_NOEC(t_brkpt, T_BRKPT)
TRAPHANDLER_NOEC(t_oflow, T_OFLOW)
TRAPHANDLER_NOEC(t_bound, T_BOUND)
TRAPHANDLER_NOEC(t_illop, T_ILLOP)
TRAPHANDLER_NOEC(t_device, T_DEVICE)
TRAPHANDLER(t_dblflt, T_DBLFLT)
TRAPHANDLER(t_tss, T_TSS)
TRAPHANDLER(t_segnp, T_SEGNP)
TRAPHANDLER(t_stack, T_STACK)
TRAPHANDLER(t_gpflt, T_GPFLT)
TRAPHANDLER(t_pgflt, T_PGFILT)
TRAPHANDLER_NOEC(t_fperr, T_FPERR)
TRAPHANDLER(t_align, T_ALIGN)
TRAPHANDLER_NOEC(t_mchk, T_MCHK)
TRAPHANDLER_NOEC(t_simderr, T_SIMDERR)

TRAPHANDLER_NOEC(t_syscall, T_SYSCALL)
```

_alltraps 的代码：

```
/*
 * Lab 3: Your code here for _alltraps
 */

_alltraps:
    pushl %ds
    pushl %es
    pushal

    movl $GD_KD,%eax
    movl %eax,%ds
    movl %eax,%es

    pushl %esp
    call trap
```

在 trap.c 文件中对 trap_init 函数的填充，主要的作用是对 IDT 中断描述表的初始化。

在 inc/mmu.h 文件中，定义了 SETGATE 函数，注意各参数的含义。

```

// Set up a normal interrupt/trap gate descriptor.
// - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
//   see section 9.6.1.3 of the i386 reference: "The difference between
//   an interrupt gate and a trap gate is in the effect on IF (the
//   interrupt-enable flag). An interrupt that vectors through an
//   interrupt gate resets IF, thereby preventing other interrupts from
//   interfering with the current interrupt handler. A subsequent IRET
//   instruction restores IF to the value in the EFLAGS image on the
//   stack. An interrupt through a trap gate does not change IF."
// - sel: Code segment selector for interrupt/trap handler
// - off: Offset in code segment for interrupt/trap handler
// - dpl: Descriptor Privilege Level -
//   the privilege level required for software to invoke
//   this interrupt/trap gate explicitly using an int instruction.
#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;
}

```

注意 sel 是内核的 cs 段。off 是偏移量。

```

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

    // LAB 3: Your code here.
    void t_divide();
    void t_debug();
    void t_nmi();
    void t_brkpt();
    void t_oflow();
    void t_bound();
    void t_illop();
    void t_device();
    void t_dblflt();
    void t_tss();
    void t_segnp();
    void t_stack();
    void t_gpflt();
    void t_pgflt();
    void t_fperr();
    void t_align();
    void t_mchk();
    void t_simderr();
    void t_syscall();
}

```

```

    |
    SETGATE(idt[T_DIVIDE], 0, GD_KT, t_divide, 0);
    SETGATE(idt[T_DEBUG], 0, GD_KT, t_debug, 0);
    SETGATE(idt[T_NMI], 0, GD_KT, t_nmi, 0);
    SETGATE(idt[T_BRKPT], 0, GD_KT, t_brkpt, 3);
    SETGATE(idt[T_OFLOW], 0, GD_KT, t_oflow, 0);
    SETGATE(idt[T_BOUND], 0, GD_KT, t_bound, 0);
    SETGATE(idt[T_ILOP], 0, GD_KT, t_illop, 0);
    SETGATE(idt[T_DEVICE], 0, GD_KT, t_device, 0);
    SETGATE(idt[T_DBLEFLT], 0, GD_KT, t_dblflt, 0);
    SETGATE(idt[T_TSS], 0, GD_KT, t_tss, 0);
    SETGATE(idt[T_SEGNP], 0, GD_KT, t_segnp, 0);
    SETGATE(idt[T_STACK], 0, GD_KT, t_stack, 0);
    SETGATE(idt[T_GPFLT], 0, GD_KT, t_gpfilt, 0);
    SETGATE(idt[T_PGFILT], 0, GD_KT, t_pgfilt, 0);
    SETGATE(idt[T_FPERR], 0, GD_KT, t_fperr, 0);
    SETGATE(idt[T_ALIGN], 0, GD_KT, t_align, 0);
    SETGATE(idt[T_MCHK], 0, GD_KT, t_mchk, 0);
    SETGATE(idt[T SIMDERR], 0, GD_KT, t_simderr, 0);
    SETGATE(idt[T_SYSCALL], 0, GD_KT, t_syscall, 3);

    // Per-CPU setup
    trap_init_percpu();
}

```

该函数会在进入内核时由 i386_init() 调用。我们添加的代码就是建立 IDT, trap_init_percpu() 中的 lidt(&idt_pd); 正式加载 IDT。

Part B: Page Faults, Breakpoints Exceptions, and System Calls

现在, JOS 内核具有基本的异常处理功能, 需要对其进行优化, 以提供依赖于异常处理的重要操作系统原语。

Handling Page Faults

页面错误异常, 中断向量 14 (T_PGFILT), 是一个特别重要的异常, 我们将在本实验以及下一个实验中大量使用它。当处理器发生页面故障时, 它将导致故障的线性 (即虚拟) 地址存储在特殊的处理器控制寄存器 CR2 中。在 trap.c 中, 我们提供了一个特殊的函数 page_fault_handler () 的开头, 用于处理页面错误异常。

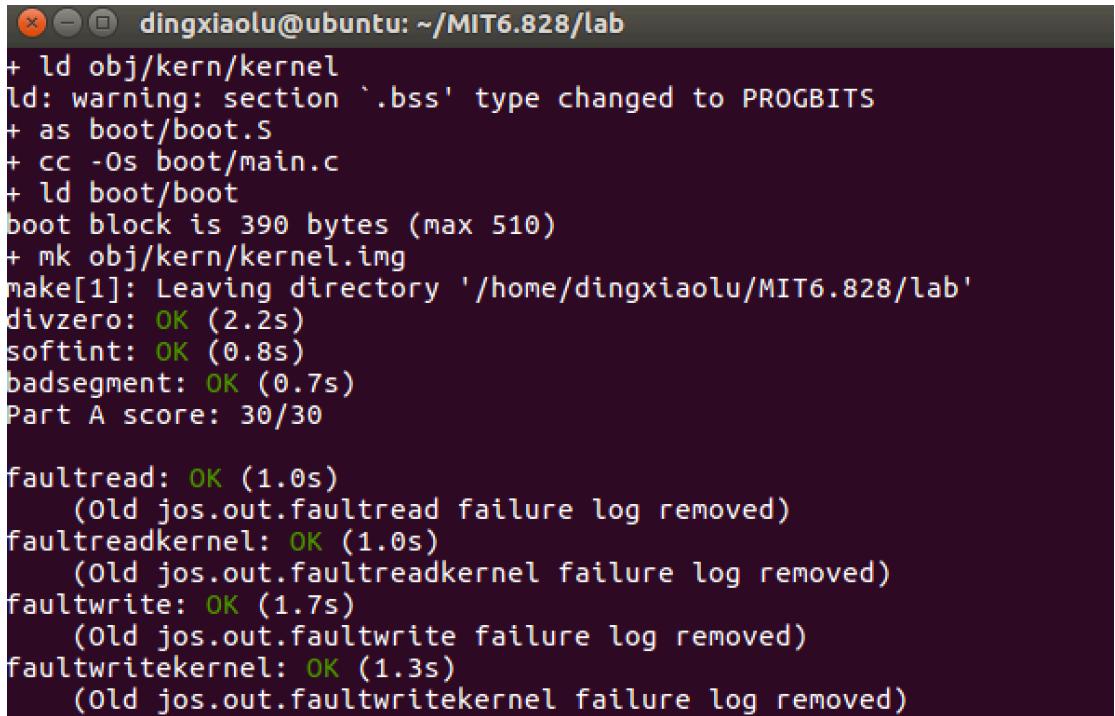
Exercise 5

Exercise 5. Modify `trap_dispatch()` to dispatch page fault exceptions to `page_fault_handler()`. You should now be able to get `make grade` to succeed on the `faultread`, `faultreadkernel`, `faultwrite`, and `faultwritekernel` tests. If any of them don't work, figure out why and fix them. Remember that you can boot JOS into a particular user program using `make run-x` or `make run-x-nox`. For instance, `make run-hello-nox` runs the `hello` user program.

解答：判断 tf 结构体中的 tf_trapno 字段。

```
static void
trap_dispatch(struct Trapframe *tf)
{
    // Handle processor exceptions.
    // LAB 3: Your code here.
    if(tf->tf_trapno == T_PGFLT){
        page_fault_handler(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 {
        env_destroy(curenv);
        return;
    }
}
```

测试成功：



A terminal window titled "dingxiaolu@ubuntu: ~/MIT6.828/lab". The window displays the following build logs:

```
+ ld obj/kern/kernel
ld: warning: section `\.bss' type changed to PROGBITS
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 390 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]: Leaving directory '/home/dingxiaolu/MIT6.828/lab'
divzero: OK (2.2s)
softint: OK (0.8s)
badsegment: OK (0.7s)
Part A score: 30/30

faultread: OK (1.0s)
    (Old jos.out.faultread failure log removed)
faultreadkernel: OK (1.0s)
    (Old jos.out.faultreadkernel failure log removed)
faultwrite: OK (1.7s)
    (Old jos.out.faultwrite failure log removed)
faultwritekernel: OK (1.3s)
    (Old jos.out.faultwritekernel failure log removed)
```

在操作系统调用时，您将在下面进一步完善内核的页面错误处理。

The Breakpoint Exception

断点异常中断向量 3 (T_BRKPT) 通常用于允许调试器通过用特殊的 1 字节 int3 软件中断指令临时替换相关的程序指令，从而在程序代码中插入断点。在 JOS 中，我们将通过将其转变为原始的伪系统调用（任何用户环境都可以用来调用 JOS 内核监视器）来稍微常用此异常。如果我们将 JOS 内核监视器视为原始调试器，则这种用法实际上是适当的。例如，lib / panic.c 中 panic () 的用户模式实现在显示其 panic 消息后执行 int3。

Exercise 6

Exercise 6. Modify trap_dispatch() to make breakpoint exceptions invoke the kernel monitor. You should now be able to get make grade to succeed on the breakpoint test.

解答：

继续完善 trap_dispatch () 函数。发生此异常调用 monitor。

```
static void
trap_dispatch(struct Trapframe *tf)
{
    // Handle processor exceptions.
    // LAB 3: Your code here.

    //page fault
    if(tf->tf_trapno == T_PGFLT){
        page_fault_handler(tf);
        return ;
    }
    //breakpoint fault
    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 {
        env_destroy(curenv);
        return;
    }
}
```

测试通过：

```
dingxiao@ubuntu: ~/MIT6.828/lab
sh: echo: I/O error
+ ld obj/kern/kernel
ld: warning: section `.bss' type changed to PROGBITS
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 390 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]: Leaving directory '/home/dingxiao/MIT6.828/lab'
divzero: OK (1.4s)
softint: OK (1.0s)
badsegment: OK (1.0s)
Part A score: 30/30

faultread: OK (1.0s)
faultreadkernel: OK (2.1s)
faultwrite: OK (1.0s)
faultwritekernel: OK (1.7s)
breakpoint: OK (1.3s)
    (Old jos.out.breakpoint failure log removed)
```

System calls

用户进程要求内核通过调用系统调用来为它们做事。当用户进程调用系统调用时，处理器进入内核模式，处理器和内核协作以保存用户进程的状态，内核执行适当的代码以执行系统调用，然后恢复用户进程。用户进程如何引起内核的注意以及如何指定要执行的确切细节因系统而异。

在 JOS 内核中，我们将使用 int 指令，这将导致处理器中断。特别是，我们将 int \$ 0x30 用作系统调用中断。我们为您定义了常数 T_SYSCALL 为 48 (0x30)。您将必须设置中断描述符以允许用户进程引起该中断。请注意，中断 0x30 不能由硬件生成，因此不会由于允许用户代码生成而引起歧义。

应用程序将在寄存器中传递系统调用号和系统调用参数。这样，内核就无需在用户环境的堆栈或指令流中乱搞。**系统调用号将以 %eax 开头，参数（最多五个）将分别以 %edx, %ecx, %ebx, %edi 和 %esi 开头。内核将返回值传回 %eax。使用寄存器传递系统调用号和参数。系统调用号保存在 %eax，五个参数依次保存在 %edx, %ecx, %ebx, %edi, %esi 中。返回值保存在 %eax 中。**在 lib / syscall.c 中的 syscall () 中已为您编写了用于调用系统调用的汇编代码。您应该通读它，并确保您了解正在发生的事情。

Exercise 7

Exercise 7. Add a handler in the kernel for interrupt vector T_SYSCALL. You will have to edit kern/trapentry.S and kern/trap.c's trap_init(). You also need to change trap_dispatch() to handle the system call interrupt by calling syscall() (defined in kern/syscall.c) with the appropriate arguments, and then arranging for the return value to be passed back to the user process in %eax. Finally, you need to implement syscall() in kern/syscall.c. Make sure syscall() returns -E_INVAL if the system call number is invalid. You should read and understand lib/syscall.c (especially the inline assembly routine) in order to confirm your understanding of the system call interface. Handle all the system calls listed in inc/syscall.h by invoking the corresponding kernel function for each call.

Run the user/hello program under your kernel (`make run-hello`). It should print "hello, world" on the console and then cause a page fault in user mode. If this does not happen, it probably means your system call handler isn't quite right. You should also now be able to get `make grade` to succeed on the testbss test.

解答：

在内核中为中断向量 T_SYSCALL 添加一个处理程序。您将必须编辑 kern / trapentry.S 和 kern / trap.c 的 trap_init ()。

添加：

在 trapentry.S 中添加：

```
TRAPHANDLER_NOEC(t_syscall, T_SYSCALL)
```

在 trap.c 的 trap_init 函数中添加：

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

您还需要更改 trap_dispatch () 以处理系统调用中断，方法是使用适当的参数调用 syscall ()（在 kern / syscall.c 中定义），然后安排将返回值传递回%eax 中的用户进程。

按照 lib / syscall.c 中的 syscall () 中对每个参数放入什么寄存器的规则。

```
static void
trap_dispatch(struct Trapframe *tf)
{
    // Handle processor exceptions.
    // LAB 3: Your code here.

    //page fault
    if(tf->tf_trapno == T_PGFLT){
        page_fault_handler(tf);
        return ;
    }
    //breakpoint fault
    if(tf->tf_trapno == T_BRKPT){
        monitor(tf);
        return ;
    }

    //system Call
    if(tf->tf_trapno == T_SYSCALL){
        tf->tf_regs.reg_eax=syscall(tf->tf_regs.res_eax, tf->tf_regs.res_edx, tf->tf_regs.res_ecx, tf->tf_regs.res_ebx, tf->tf_regs.res_edi, tf->tf_regs.res_esi);
        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 {
        env_destroy(curenv);
        return;
    }
}
```

最后，您需要在 kern / syscall.c 中实现 syscall ()。

如果系统调用号码无效，请确保 syscall () 返回-EINVAL。您应该阅读并理解 lib / syscall.c（尤其是内联汇编例程），以确认您对系统调用接口的理解。

通过为每个调用调用相应的内核函数来处理 inc / syscall.h 中列出的所有系统调用。

在文件 inc/syscall.h 文件中：

```
/* system call numbers */
enum {
    SYS_cputs = 0,
    SYS_cgetc,
    SYS_getenvid,
    SYS_env_destroy,
    NSYSCALLS
};

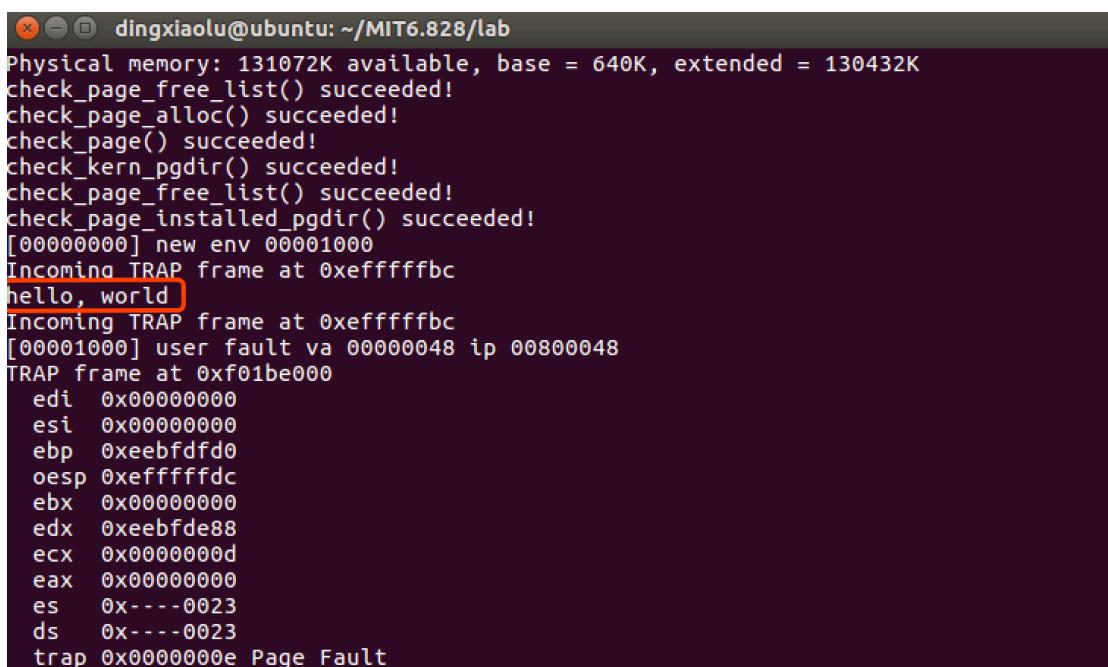
// Dispatches to the correct kernel function, passing the arguments.
int32_t
syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4, uint32_t a5)
{
    // Call the function corresponding to the 'syscallno' parameter.
    // Return any appropriate return value.
    // LAB 3: Your code here.

    //panic("syscall not implemented");
    int32_t ret;

    switch (syscallno) {
        case (SYS_cputs):
            sys_cputs((char*)a1,(size_t)a2);
            ret=0;
            break;
        case (SYS_cgetc):
            ret=sys_cgetc();
            break;
        case (SYS_getenvid):
            ret=sys_getenvid();
            break;
        case (SYS_env_destroy):
            ret=sys_env_destroy((envid_t)a1);
            break;

        default:
            return -E_INVAL;
    }
    return ret;
}
```

测试通过：使用 make run-hello



```
dingxiaolu@ubuntu: ~/MIT6.828/lab
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
[00000000] new env 00001000
Incoming TRAP frame at 0xfffffb
hello, world
Incoming TRAP frame at 0xfffffb
[00001000] user fault va 00000048 ip 00800048
TRAP frame at 0xf01be000
    edi 0x00000000
    esi 0x00000000
    ebp 0xeebfdf0
    oesp 0xfffffd0
    ebx 0x00000000
    edx 0xeebfde88
    ecx 0x0000000d
    eax 0x00000000
    es 0x----0023
    ds 0x----0023
trap 0x0000000e Page Fault
```

User-mode startup

用户程序开始在 lib / entry.S 的顶部运行。 进行一些设置后，此代码在 lib / libmain.c 中调用 libmain () 。 您应该修改 libmain () ， 以初始化全局指针 thisenv 指向 envs [] 数组中此环境的结构 Env。（请注意， lib / entry.S 已经定义了 env， 以指向您在 A 部分中设置的 UENVS 映射。） 提示：查看 inc / env.h 并使用 sys_getenvid。

然后， libmain () 调用 umain，在 hello 程序的情况下，它位于 user / hello.c 中。 请注意，在打印 “hello, world” 之后，它将尝试访问 thisenv-> env_id。 这就是为什么它较早出现故障。 既然您已经正确初始化了 thisenv， 它应该不会出错。 如果仍然存在问题，则您可能尚未将 UENVS 区域映射为用户可读（返回 pmap.c 中的 A 部分；这是我们第一次实际使用 UENVS 区域）。

Exercise 8

Exercise 8. Add the required code to the user library, then boot your kernel. You should see user/hello print "hello, world" and then print "i am environment 00001000". user/hello then attempts to "exit" by calling sys_env_destroy() (see lib/libmain.c and lib/exit.c). Since the kernel currently only supports one user environment, it should report that it has destroyed the only environment and then drop into the kernel monitor. You should be able to get make grade to succeed on the hello test.

解答：

在 lib/libmain.c 文件中：

```
#include <inc/lib.h>

extern void umain(int argc, char **argv);

const volatile struct Env *thisenv;
const char *binaryname = "<unknown>";

void
libmain(int argc, char **argv)
{
    // set thisenv to point at our Env structure in envs[].
    // LAB 3: Your code here.
    thisenv = &envs[ENVX(sys_getenvid())];//获得当前环境的envid，通过sys_getenvid得到；ENVX()获取到envid的索引；

    // save the name of the program so that panic() can use it
    if (argc > 0)
        binaryname = argv[0];

    // call user main routine
    umain(argc, argv);
    // exit gracefully
    exit();
}
```

```
dingxiaolu@ubuntu: ~/MIT6.828/lab
+ ld obj/user/faultwritekernel
sh: echo: I/O error
+ ld obj/kern/kernel
ld: warning: section `.bss' type changed to PROGBITS
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 390 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]: Leaving directory '/home/dingxiaolu/MIT6.828/lab'
divzero: OK (2.7s)
softint: OK (1.1s)
badsegment: OK (0.8s)
Part A score: 30/30

faultread: OK (0.7s)
faultreadkernel: OK (2.0s)
faultwrite: OK (1.0s)
faultwritekernel: OK (1.7s)
breakpoint: OK (2.3s)
testbss: OK (1.0s)
hello: OK (1.7s)
    (Old jos.out.hello failure log removed)
```

Page faults and memory protection

内存保护是操作系统的一项重要功能，可确保一个程序中的错误不会破坏其他程序或破坏操作系统本身。

操作系统通常依靠硬件支持来实现内存保护。操作系统使硬件知道哪些虚拟地址有效，哪些无效。当程序尝试访问无效地址或没有权限的地址时，处理器会在导致故障的指令处停止程序，然后使用有关尝试操作的信息捕获内核。如果该错误是可修复的，则内核可以对其进行修复，并让程序继续运行。如果故障无法修复，则程序将无法继续，因为它将永远不会超过导致故障的指令。

作为可修复故障的示例，请考虑自动扩展栈。在许多系统中，内核最初分配一个栈页面，然后，如果程序无法访问堆栈中更远的页面，内核将自动分配这些页面并使程序继续。这样，内核仅分配程序需要的栈内存，但是程序可以在其拥有任意大栈的错觉下工作。

系统调用提出了一个有趣的内存保护问题。大多数系统调用接口都允许用户程序将指针传递给内核。这些指针指向要读取或写入的用户缓冲区。然后，内核在执行系统调用时取消引用这些指针。这有两个问题：

- 1) 内核中的页面错误可能比用户程序中的页面错误严重得多。如果内核在处理自己的数据结构时发生页面错误，那就是内核错误，并且错误处理程序应该使内核（进而整个系统）感到恐慌。但是，当内核取消引用用户程序给它

的指针时，它需要一种方法来记住这些取消引用引起的任何页面错误实际上都代表用户程序。

2) 内核通常具有比用户程序更多的内存权限。 用户程序可能会传递一个指向系统调用的指针，该指针指向内核可以读取或写入但程序无法读取的内存。 内核必须小心，不要被欺骗去解引用这样的指针，因为这可能会泄露私有信息或破坏内核的完整性。

由于这两个原因，内核在处理用户程序提供的指针时必须格外小心。

现在，您将通过一种机制来仔细检查这两个问题，该机制将仔细检查从用户空间传递到内核的所有指针。 当程序将指针传递给内核时，内核将检查地址是否在地址空间的用户部分中，以及页表是否允许进行内存操作。

因此，内核将永远不会由于取消引用用户提供的指针而遭受页面错误。 如果内核确实发生页面错误，则它应该崩溃并终止。

Exercise 9

Exercise 9. Change `kern/trap.c` to panic if a page fault happens in kernel mode.

Hint: to determine whether a fault happened in user mode or in kernel mode, check the low bits of the `tf_cs`.

Read `user_mem_assert` in `kern/pmap.c` and implement `user_mem_check` in that same file.

Change `kern/syscall.c` to sanity check arguments to system calls.

Boot your kernel, running `user/buggyhello`. The environment should be destroyed, and the kernel should *not* panic. You should see:

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

Finally, change `debuginfo_eip` in `kern/kdebug.c` to call `user_mem_check` on `usd`, `stabs`, and `stabstr`. If you now run `user/breakpoint`, you should be able to run `backtrace` from the kernel monitor and see the backtrace traverse into `lib/libmain.c` before the kernel panics with a page fault. What causes this page fault? You don't need to fix it, but you should understand why it happens.

第一个任务：在 `kern/trap.c` 中，当出现页错误时，判断是否发生在内核模式，如果发生在内核模式就 panic。

利用 `tf_cs` 段中的低位去判断是否属于内核模式，在该寄存器的低 2 位，叫做 CPL 位，是用来表示当前运行代码的访问权限级别，0 表示内核态，3 表示用户态。

```

void
page_fault_handler(struct Trapframe *tf)
{
    uint32_t fault_va;

    // Read processor's CR2 register to find the faulting address
    fault_va = rcr2();

    // Handle kernel-mode page faults.

    // LAB 3: Your code here.
    // 检查权限位是内核模式和用户模式
    if((tf->tf_cs && 0x01)==0){
        panic("page_fault happen in kernel mode,faultAddress %d \n",fault_va);
    }
    // We've already handled kernel-mode exceptions, so if we get here,
    // the page fault happened in user mode.

    // Destroy the environment that caused the fault.
    sprintf("[%08x] user fault va %08x ip %08x\n",
            curenv->env_id, fault_va, tf->tf_eip);
    print_trapframe(tf);
    env_destroy(curenv);
}

```

第二个任务：修改 kern/pmap.c 文件，使其能够检查某一虚拟内存地址范围拥有 perm|PTE_P 访问权限。根据虚拟地址范围，在页表中找到相应的页表项，观察页表项相应的访问权限进行判断。

```

int
user_mem_check(struct Env *env, const void *va, size_t len, int perm)
{
    // LAB 3: Your code here.
    uint32_t start=(uint32_t)ROUNDDOWN(va,PGSIZE);
    uint32_t end=(uint32_t)ROUNDUP(va+len,PGSIZE);
    uint32_t i;
    for(i=start;i<end;i+=PGSIZE){
        pte_t *pte=pgdir_walk(env->env_pgdir, va, perm);
        if(i>ULIM || !pte || !(pte&PTE_P) || ((*pte&perm)!=perm)){
            user_mem_check_addr= i<(uint32_t)va ? (uint32_t)va : i;
            return -E_FAULT;
        }
    }
    return 0;
}

```

第三个任务：丰富 kern/syscall.c 文件，修改 sys_cputs 函数，使其检查用户程序对虚拟地址的访问权限。

```

// 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);
}

```

第四个任务：修改 kern/kdebug.c 文件，完善 debuginfo_eip 函数。

```

// The user-application linker script, user/user.ld,
// puts information about the application's stabs (equivalent
// to __STAB_BEGIN__, __STAB_END__, __STABSTR_BEGIN__, and
// __STABSTR_END__) in a structure located at virtual address
// USTABDATA.
const struct UserStabData *usd = (const struct UserStabData *) USTABDATA;

// 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(*usd),PTE_U)<0){
    return -1;
}
stabs = usd->stabs;
stab_end = usd->stab_end;
stabstr = usd->stabstr;
stabstr_end = usd->stabstr_end;

// 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)<0) || (user_mem_check(curenv, stabstr, stabstr_end-stabstr_end,PTE_U)<0)){
    return -1;
}
}

```

Exercise 10

Exercise 10. Boot your kernel, running `user/evilhello`. The environment should be destroyed, and the kernel should not panic.
You should see:

```
[00000000] new env 00001000
...
[00001000] user_mem_check assertion failure for va f010000c
[00001000] free env 00001000
```

重启内核，观察。

```

dingxiao@ubuntu:~/MIT6.828/lab$ cd ..
dingxiao@ubuntu:~/MIT6.828/lab$ make run-evilhello
make[1]: Entering directory '/home/dingxiao/MIT6.828/lab'
make[1]: Leaving directory '/home/dingxiao/MIT6.828/lab'
qemu-system-i386 -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -
serial mon:stdio -gdb tcp::26000 -D qemu.log
VNC server running on '127.0.0.1:5900'
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
[00000000] new env 00001000
Incoming TRAP frame at 0xfffffb0
Incoming TRAP frame at 0xfffffb0
[00001000] user_mem_check assertion failure for va f010000c
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>

```

最终测试通过：

```

dingxiaolu@ubuntu:~/MIT6.828/lab$ ./grade-lab3
divzero: OK (1.9s)
softint: OK (1.2s)
badsegment: OK (0.7s)
Part A score: 30/30

faultread: OK (0.8s)
faultreadkernel: OK (2.0s)
faultwrite: OK (1.0s)
faultwritekernel: OK (1.8s)
breakpoint: OK (1.2s)
testbss: OK (1.8s)
hello: OK (1.1s)
buggyhello: OK (1.8s)
buggyhello2: OK (2.2s)
evilhello: OK (1.0s)
Part B score: 50/50

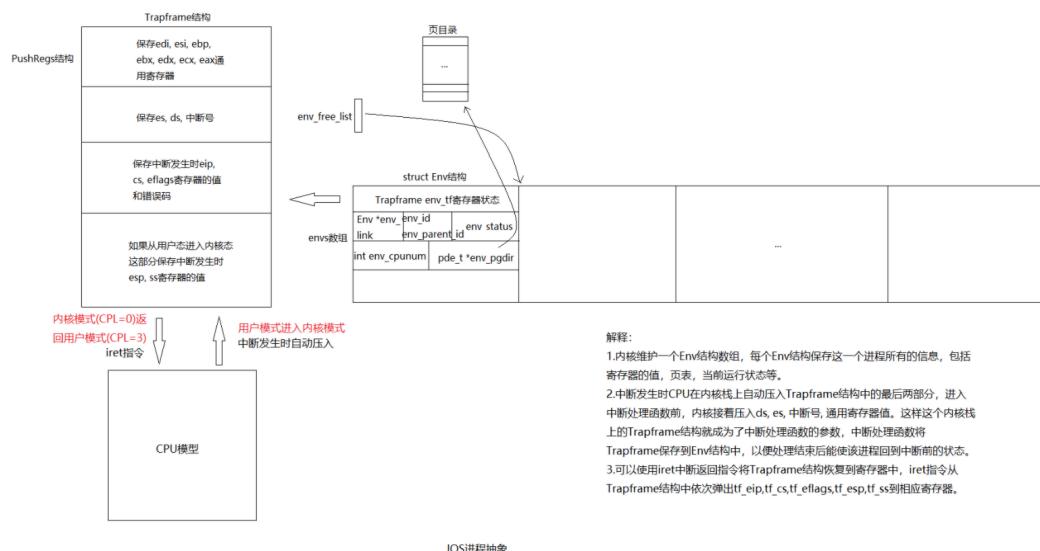
Score: 80/80

```

总结：

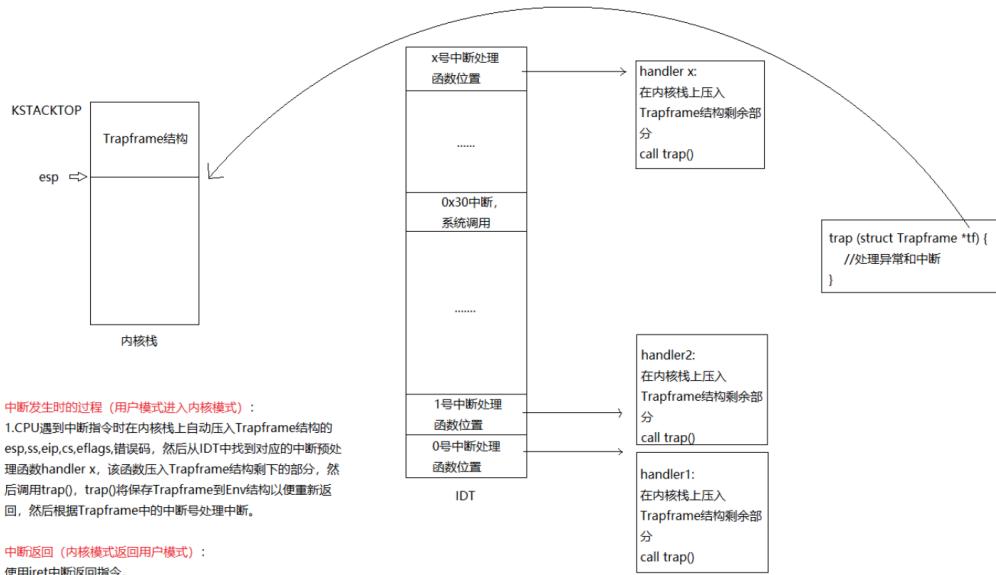
1, 进程建立，可以加载用户 ELF 文件并执行。

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：



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：



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。见下图：

