Lab 3: User Environments

Introduction

在这个实验，你将会实现基础的内核特性用来使得一个 *保护的用户模式环境* 运行。

你将会加强 `JOS` 内核来设置一个数据结构来跟踪 *用户环境* ， 创建一个 *单一的用户环境* ， 加载一个 *程序镜像* ， 并且开始运行它。

你也将会使 `JOS` 内核能够处理任何 *用户环境* 给出的 `system calls` 以及导致的任何 `exceptions` 。

Note

这个实验中， `environment` 和 `process` 是可交换的，两个都指向一个允许你运行程序的抽象。介绍 `environment` 而不是传统的 `process` 是因为想 *强调* **`JOS environments` 和 `UNIX process` 提供不同的接口**并且不提供相同的 `semantics` 。

Getting Started

```
cd ~/6.828/lab
add git
git commit -am 'changes to lab2 after handin'
git pull        # 如果这里出错，要导入环境变量 export GIT_SSL_NO_VERIFY=1
#Already up-to-date
git checkout -b lab3 origin/lab3
git merge lab2
```

Lab3 包含一些新 *源文件* ，你应该浏览一下

`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

Lab Requirements

实验被分为A，B两部分。

在lab2中，你将需要做lab中描述的所有常规练习和 *至少一个* `challenge problem`（对于整个lab，而不是每个部分）。

写下简短的答案，对于lab中的问题。在 `answer-lab3.txt` 中写下一段或两段你做了什么去解决你选择的 `challenge problem` 。不要忘记在提交的时候包含答案文件， `git add answers-lab3.txt` 。

Inline Assembly

在这个lab，你可能会发现 `GCC` 的**inline assembly language**特性很有用，尽管不使用它也可能完成lab。至少你需要能够理解 `asm` 片段，这些片段已经存在于我们给你的源文件中。

你可以找到一些 `GCC` `inline assembly language` 在课程的引用材料页面。

Part A: User Environments and Exception Handling

新的头文件 `inc/env.h` 包含了 `JOS` 用户环境的基础定义。

`kernel` 使用 `Env` 这个数据结构来跟踪每个用户环境。

这个lab你将初步创建仅仅一个环境，但是你将需要设计 `JOS kernel` 来支持 *多环境* ； `lab4` 将通过允许一个用户环境 `fork` 其它环境来利用这个特性。

正如你在 `kern/env.c` 中看到的， `kernel` 维持三个主要的和环境相关的全局变量：

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

一旦 `JOS` 觉醒并运行， `envs` 指针指向一个 `Env` 结构的数组来表示系统中的 *所有环境* 。

在我们的设计中， `JOS kernel` 将会支持最大 `NENV` 个同时活动的环境，尽管在任何给定的时间正在运行的环境会远少于 `NENV` 。

一旦被分配， `envs` 数组将会包含每个 `NENV` 可能环境 `Env` 数据结构的单个实例。

`JOS kernel` 用 `env_free_list` 来维持所有不活动的 `Env` 结构。这个设计允许简单的环境分配和回收，因为仅仅需要在空闲链表中添加或移除。

`kenel` 使用 `curenv` 符号在任何给定的时间跟踪 *当前正在执行* 的环境。在启动过程中，第一个环境运行之前， `curenv` 被初始化为 `NULL` 。

Environment State

`Env` 结构定义于 `inc/env.h`

```
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
};
```

以下是 `Env` 各个域代表什么：

- **env_tf**

  该结构定义于 `inc/trap.h` ，记录环境中保存的寄存器值当该环境没有运行时：例如 `kernel` 或者不同的环境正在运行。 `kernel` 保存这些当从 *用户模式* 转为 *内核模式* 时，所以环境可以稍后从它离开的地方继续运行。

- **env_link**

  这是一个指向 `env_free_list` 中下一个 `Env` 的链接。 `env_free_list` 指向链表中第一个空闲环境。

- **env_id**

  `kernel` 在这里存放一个值，这个值可以独一无二地识别当前使用 `Env` 结构的环境。当一个用户环境结束之后， `kernel` 可能重新分配相同的 `Env` 结构给一个不同的环境，但是新环境将会有一个**不同的** `env_id` ，尽管新环境使用的是 `envs` 数组中相同的 `slot` 。

- **env_parent_id**

  `kernel` 存储创建该环境的 `env_id` 。这样，环境可以形成一个 `family tree` ，这对做出哪个环境被允许对谁做什么的安全决策很有用。

- **env_type**

  别用于区分特殊的环境。对于大多数的环境，这个值可能是 `ENV_TYPE_USER` 。在之后的lab中，我们会介绍更多特殊系统服务环境的类型。

- **env_status**

  这个变量保存以下值的其中之一：

  ENV_FREE

  表示 `Env` 结构未激活，因此在 `env_free_list` 中。

  ENV_RUNNABLE

  表示 `Env` 结构表示一个环境正在等待在处理器上运行。

  ENV_RUNNING

  表示 `Env` 结构表示当前正在运行的环境。

  ENV_NOT_RUNNABLE

  表示 `Env` 结构表示一个当前激活的环境，但是当前没有准备好去运行：例如，因为它正在等待来自另一个环境的进程间的通信。

  ENV_DYING

  表示 `Env` 结构表示一个僵尸环境。僵尸环境在下次陷入 `kernel` 时会被释放，这个flag将不会使用直到Lab4。

- **env_pgdir**

  这个变量保存 `kernel` 的 *虚拟地址* ，该地址是这个环境的 *页目录* 地址。

就像一个 `Unix` 进程，一个 `JOS` 环境耦合 `thread` 和 `address space` 概念。*线程* 主要定义于保存的寄存器（ `env_tf` 域）， *地址空间* 主要定义于 *页目录* 和 *页表* （由 `env_pgdir` 指向）。为了运行一个环境， `kernel` 必须设置CPU中保存的 *寄存器* 和合适的 *地址空间* 。

我们的 `struct Env` 和 *xv6* 中的 `struct proc` 是类似的。两个结构都在 `Trapframe` 结构中保持环境的 *用户模式寄存器状态* 。在 `JOS` 中，单独的环境没有它们自己的 `kernel stack` 。在一个时间 `kernel` 中只有一个 `JOS` 环境，所以 `JOS` 只需要单个的 `kernel stack` 。

Allocating the Environments Array

lab2中，在 `mem_init()` 中对 `pages[]` 分配了内存， `pages[]` 是 `kernel` 用来跟踪pages是否空闲的一个表。你现在将需要修改 `mem_init()` 来分配一个相似的 `Env` 结构的数组，叫做 `envs` 。

> **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.
>
> **answer**

Creating and Running Environments

现在你将要在 `kern/env.c` 文件中写必要的代码来运行一个用户环境。因为现在还没有一个文件系统，我们将设置 `kernel` 来加载一个 *静态的二进制镜像* ，这个镜像嵌入在 `kernel` 本身。

`JOS` 在 `kernel` 嵌入这个二进制文件作为一个ELF可执行镜像。

Lab3的 `GNUmakefile` 生成了许多 *二进制镜像* 在目录 `obj/user/` 中。如果你看 `kern/Makefrag` ，你会注意到一些魔法—直接“链接”这些二进制文件进入到 `kernel` 的可执行文件就好像它们是 `.o` 文件。链接器命令行的选项 `-b binary` 使得这些文件作为“原生的”未被解释的二进制文件而不是作为常规由编译器生成的 `.o` 文件被 *链接* 。（就链接器而言，这些文件根本不必是ELF镜像—它们可以是任何东西，比如文本文件或者图片！）

如果你看 `obj/kern/kernel.sym` 在构建完 `kernel` 之后，你将注意到链接器“魔法般地”生成了一些有趣的带有隐晦名字的标志像是 `_binary_obj_user_hello_start`，`_binary_obj_user_hello_end`， and `_binary_obj_user_hello_size`. 链接器通过破坏二进制的文件名来生成这些符号；这些符号为常规的 `kernel` 代码一种引用嵌入式二进制文件的方法。

在文件 `kern/init.c` 中的 `i386_init()`，你可以看到这些 *二进制镜像* 的其中之一在一个环境中运行。但是设置 *用户环境* 的关键函数没有完成，你需要去完成它。

> **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".

env.c

```
// 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.
//
// Application segment type bits
#define STA_X       0x8     // Executable segment
#define STA_E       0x4     // Expand down (non-executable
segments)
#define STA_C       0x4     // Conforming code segment (executable
only)
#define STA_W       0x2     // Writeable (non-executable segments)
#define STA_R       0x2     // Readable (executable segments)
#define STA_A       0x1     // Accessed
```

```
// Normal segment
  `#define SEG(type, base, lim, dpl)                    \
{ ((lim) >> 12) & 0xffff, (base) & 0xffff, ((base) >> 16) & 0xff,
  \
type, 1, dpl, 1, (unsigned) (lim) >> 28, 0, 0, 1, 1,         \
(unsigned) (base) >> 24 }
// 段各个字段的含义
struct Segdesc {
unsigned sd_lim_15_0 : 16;  // Low bits of segment limit
unsigned sd_base_15_0 : 16; // Low bits of segment base address
unsigned sd_base_23_16 : 8; // Middle bits of segment base address
unsigned sd_type : 4;       // Segment type (see STS_ constants)
unsigned sd_s : 1;          // 0 = system, 1 = application
unsigned sd_dpl : 2;        // Descriptor Privilege Level
unsigned sd_p : 1;          // Present
unsigned sd_lim_19_16 : 4;  // High bits of segment limit
unsigned sd_avl : 1;        // Unused (available for software use)
unsigned sd_rsv1 : 1;       // Reserved
unsigned sd_db : 1;         // 0 = 16-bit segment, 1 = 32-bit
segment
unsigned sd_g : 1;          // Granularity: limit scaled by 4K
when set
unsigned sd_base_31_24 : 8; // High bits of segment base address
};
// 定义一个全局描述符数组
struct Segdesc gdt[] =
{
    // inc/mmu.h 可以查看SEG定义
// 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),         //
内核代码段 可执行|可读

// 0x10 - kernel data segment
[GD_KD >> 3] = SEG(STA_W, 0x0, 0xffffffff, 0),                 //
内核数据段 不可执行|可写
```

```
// 0x18 - user code segment
[GD_UT >> 3] = SEG(STA_X | STA_R, 0x0, 0xffffffff, 3),          //
用户代码段 可执行|可读

// 0x20 - user data segment
[GD_UD >> 3] = SEG(STA_W, 0x0, 0xffffffff, 3),                  //
用户数据段 不可执行|可写

// 0x28 - tss, initialized in trap_init_percpu()
[GD_TSS0 >> 3] = SEG_NULL
};
```

SEG定义

```
// Normal segment
`#define SEG(type, base, lim, dpl)                    \
{ ((lim) >> 12) & 0xffff, (base) & 0xffff, ((base) >> 16) & 0xff,
  \
type, 1, dpl, 1, (unsigned) (lim) >> 28, 0, 0, 1, 1,        \
(unsigned) (base) >> 24
```

`envid2env()`

```
// Converts an envid to an env pointer.
// If checkperm is set, the specified environment must be either
the
// current environment or an immediate child of the current
environment.
//
// 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)
{
struct Env *e;
```

```
// If envid is zero, return the current environment.
if (envid == 0) {
*env_store = curenv;
return 0;
}

// Look up the Env structure via the index part of the envid,
// then check the env_id field in that struct Env
// to ensure that the envid is not stale
// (i.e., does not refer to a _previous_ environment
// that used the same slot in the envs[] array).
e = &envs[ENVX(envid)];
if (e->env_status == ENV_FREE || e->env_id != envid) {
*env_store = 0;
return -E_BAD_ENV;
}

// Check that the calling environment has legitimate permission
// to manipulate the specified environment.
// If checkperm is set, the specified environment
// must be either the current environment
// or an immediate child of the current environment.
if (checkperm && e != curenv && e->env_parent_id != curenv-
>env_id) {
*env_store = 0;
return -E_BAD_ENV;
}

*env_store = e;
return 0;
}
```

`env_init()`

```
// Mark all environments in 'envs' as free, set their env_ids to
0,
// and insert them into the env_free_list.
// Make sure the environments are in the free list in the same
order
// they are in the envs array (i.e., so that the first call to
```

```
// env_alloc() returns envs[0]).
//
void
env_init(void)
{
// Set up envs array
// LAB 3: Your code here.
envs[0].env_id = 0;
env_free_list = &envs[0];
envs[0].env_status = ENV_FREE;
struct Env* temp = env_free_list;
for(int i = 1; i < NENV; ++i){
    envs[i].env_id = 0;
    envs[i].env_status = ENV_FREE;
    temp->env_link = &envs[i];
    // cprintf("temp is: %x\n", temp);
    temp = temp->env_link;
}
envs[NENV - 1].env_link = NULL;
cprintf("env_free_list is: %x\n", env_free_list);
cprintf("env_free_list->env_link is: %x\n", env_free_list-
>env_link);
cprintf("env_free_list->env_link->env_link is: %x\n",
env_free_list->env_link->env_link);

// Per-CPU part of the initialization
env_init_percpu();
}
```

`env_setup_vm()`

```
//
// Initialize the kernel virtual memory layout for environment e.
// Allocate a page directory, set e->env_pgdir accordingly,
// and initialize the kernel portion of the new environment's
address space.
// Do NOT (yet) map anything into the user portion
// of the environment's virtual address space.
//
```

```
// Returns 0 on success, < 0 on error.  Errors include:
//  -E_NO_MEM if page directory or table could not be allocated.
//
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_pgdir 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_pgdir 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_pgdir
//  is an exception -- you need to increment env_pgdir's
//  pp_ref for env_free to work correctly.
//    - The functions in kern/pmap.h are handy.

// LAB 3: Your code here.
// convert the p into virtual address
e->env_pgdir = (pte_t*)page2kva(p);
p->pp_ref++;
memcpy(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;
}
```

`env_alloc()`

```
//
// Allocates and initializes a new environment.
// On success, the new environment is stored in *newenv_store.
//
// Returns 0 on success, < 0 on failure.  Errors include:
//  -E_NO_FREE_ENV if all NENV environments are allocated
//  -E_NO_MEM on memory exhaustion
//
int
env_alloc(struct Env **newenv_store, envid_t parent_id)
{
int32_t generation;
int r;
struct Env *e;

if (!(e = env_free_list))            // free list中没有可用环境
return -E_NO_FREE_ENV;

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

// Generate an env_id for this environment.
// 将后十位取0
generation = (e->env_id + (1 << ENVGENSHIFT)) & ~(NENV - 1);
// ENVGENSHIFT = 12 NENV = 1 << 10
if (generation <= 0)    // Don't create a negative env_id.
generation = 1 << ENVGENSHIFT;
e->env_id = generation | (e - envs);

// Set the basic status variables.
e->env_parent_id = parent_id;
e->env_type = ENV_TYPE_USER;
```

```
	e->env_status = ENV_RUNNABLE;
	e->env_runs = 0;

	// Clear out all the saved register state,
	// to prevent the register values
	// of a prior environment inhabiting this Env structure
	// from "leaking" into our new environment.
	/*  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 de field tf_esp

	        uint32_t tf_err;

	        uintptr_t tf_eip Type: int

	        uint16_t tf_cs;  below here only when crossing rings, such
	as from user to kernel
	        uint16_t tf_padd

	        uint32_t tf_efla // In Trapframe

	// below here on public: int tf_esp

	        uintptr_t tf_esp;
	  uint16_t tf_ss;
	  uint16_t tf_padding4;
	} __attribute__((packed));*/
	memset(&e->env_tf, 0, sizeof(e->env_tf));

	// 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.

// commit the allocation
env_free_list = e->env_link;
*newenv_store = e;

cprintf("[%08x] new env %08x\n", curenv ? curenv->env_id : 0, e-
>env_id);
return 0;
}
```

`region_alloc()`

```
// 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.)
//
// 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!)i
```

```
va = ROUNDDOWN(va, PGSIZE);
void* start = ROUNDDOWN(va, PGSIZE);
void* end = ROUNDUP((va + len), PGSIZE);
if(UTOP < (uint32_t)end)
{
    panic("the end address exceeds the 'UTOP'");
}
while((uint32_t)start < (uint32_t)end)
{
    struct PageInfo* p = page_alloc(ALLOC_ZERO);
    if(NULL == p)
    {
        panic("page_alloc failed.");
    }
int result = page_insert(e->env_pgdir, p, start, PTE_P | PTE_W |
PTE_U);
    if(0 > result)
    {
        panic("page insert failed.");
    }
    // boot_map_region(e->env_pgdir, va, PGSIZE,
PADDR(page2kva(p)));
    start += PGSIZE;
}
cprintf("region_alloc succeed\n");
}
```

`load_icode()`

Only called during kernel initialization.

```
// Set up the initial program binary, stack, and processor flags
// for a user process.
// This function is ONLY called during kernel initialization,
// before running the first user-mode environment.
//
// This function loads all loadable segments from the ELF binary
image
// into the environment's user memory, starting at the appropriate
// virtual addresses indicated in the ELF program header.
```

```
// At the same time it clears to zero any portions of these
segments
// that are marked in the program header as being mapped
// but not actually present in the ELF file - i.e., the program's
bss section.
//
// All this is very similar to what our boot loader does, except
the boot
// loader also needs to read the code from disk.  Take a look at
// boot/main.c to get ideas.
//
// Finally, this function maps one page for the program's initial
stack.
//
// load_icode panics if it encounters problems.
//  - How might load_icode fail?  What might be wrong with the
given input?
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.

// Now map one page for the program's initial stack
// at virtual address USTACKTOP - PGSIZE.

// LAB 3: Your code here.
struct Proghdr* ph;
struct Proghdr* eph;        // end of programe head
struct Elf* ELFhdr = (struct Elf*)binary;
if(ELF_MAGIC ≠ ELFhdr→e_magic){
    panic("the elfhdr→e_magic ≠ ELF_MAGIC");
}
lcr3(PADDR(e→env_pgdir));
ph = (struct Proghdr*) ((uint8_t*)ELFhdr + ELFhdr→e_phoff);
eph = ph + ELFhdr→e_phnum;
struct PageInfo* p = NULL;
// switch to the environment
// lcr3(PADDR(e→env_pgdir));
for(; ph < eph; ++ph)
{
    if(ELF_PROG_LOAD == ph→p_type)
    {
        assert(ph→p_filesz ⩽ ph→p_memsz);
        // allocate ph→p_memsz bytes to the program segment
        region_alloc(e, (void*)ph→p_va, ph→p_memsz);
        // copy 'binary + ph→p_offset' to 'ph→p_va'
        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);
        cprintf("ph->p_va is: %x\n", ph->p_va);
        cprintf("ph->p_va + ph->p_filesz is: %x\n", ph->p_va + ph-
>p_filesz);
        cprintf("binary + ph->p_offset + ph->p_filesz is: %x\n",
binary + ph->p_offset + ph->p_filesz);
        // memset(binary + ph->p_offset + ph->p_filesz, 0, ph-
>p_memsz - ph->p_filesz);
        // all page protection bits should be user read/write
        e->env_pgdir[PDX(ph->p_va)] = PTE_ADDR(e-
>env_pgdir[PDX(ph->p_va)]) | PTE_P | PTE_U | PTE_W;

    }
}
// set the 'tf_eip'
e->env_tf.tf_eip = ELFhdr->e_entry;


// Now map one page for the program's initial stack
// at virtual address USTACKTOP - PGSIZE.
if(!(p = page_alloc(ALLOC_ZERO)))
{
    panic("page_alloc in load_icode failed.");
}
page_insert(e->env_pgdir, p, (void*)(USTACKTOP - PGSIZE), PTE_P |
PTE_U | PTE_W);
  // or use `region_alloc()`
  /* region_alloc(e, (void*)(USTACKTOP - PGSIZE), PGSIZE); */
// switch back to the kernel address space
lcr3(PADDR(kern_pgdir));

}
```

`env_create()`

```
//
// 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, enum EnvType type)
{
// LAB 3: Your code here.
// struct Env** env_store = 0;
struct Env* e;
int r;
if(0 != (r = env_alloc(&e, 0)))
{
    panic("env_create: %e", r);
}
load_icode(e, binary);
e->env_type = type;
}
```

`env_free()`

```
void
env_free(struct Env *e)
{
pte_t *pt;
uint32_t pdeno, pteno;
physaddr_t pa;

// If freeing the current environment, switch to kern_pgdir
// before freeing the page directory, just in case the page
// gets reused.
if (e == curenv)
    lcr3(PADDR(kern_pgdir));

// Note the environment's demise.
cprintf("[%08x] free env %08x\n", curenv ? curenv->env_id : 0, e-
>env_id);

// Flush all mapped pages in the user portion of the address space
static_assert(UTOP % PTSIZE == 0);
for (pdeno = 0; pdeno < PDX(UTOP); pdeno++) {
```

```
        // only look at mapped page tables
        if (!(e->env_pgdir[pdeno] & PTE_P))
            continue;

        // find the pa and va of the page table
        pa = PTE_ADDR(e->env_pgdir[pdeno]);
        pt = (pte_t*) KADDR(pa);

        // unmap all PTEs in this page table
        for (pteno = 0; pteno <= PTX(~0); pteno++) {
            if (pt[pteno] & PTE_P)
                page_remove(e->env_pgdir, PGADDR(pdeno, pteno, 0));
        }

        // free the page table itself
        e->env_pgdir[pdeno] = 0;
        page_decref(pa2page(pa));
    }

    // free the page directory
    pa = PADDR(e->env_pgdir);
    e->env_pgdir = 0;
    page_decref(pa2page(pa));

    // return the environment to the free list
    e->env_status = ENV_FREE;
    e->env_link = env_free_list;
    env_free_list = e;
    }
```

`env_destroy()`

```
// Frees environment e.
// If e was the current env, then runs a new environment (and does
not return
// to the caller).
//
void
env_destroy(struct Env *e)
```

```
{
  // If e is currently running on other CPUs, we change its state
to
  // ENV_DYING. A zombie environment will be freed the next time
  // it traps to the kernel.
  if (e->env_status == ENV_RUNNING && curenv != e) {
    e->env_status = ENV_DYING;
    return;
  }

  env_free(e);

  if (curenv == e) {
    curenv = NULL;
    sched_yield();
  }
}
```

`env_pop_tf()`

```
// 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"
    "\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 */
}
```

env_run()

```
// Context switch from curenv to env e.
// Note: if this is the first call to env_run, curenv is NULL.
//
// This function does not return.
//
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.
// 1.1
if(curenv != NULL && ENV_RUNNING == curenv->env_status){
    curenv->env_status = ENV_RUNNABLE;
}
// 1.2
curenv = e;
// 1.3
curenv->env_status = ENV_RUNNING;
// 1.4
++curenv->env_runs;
```

```
    // 1.5
    lcr3(PADDR(e->env_pgdir));
    // 2
    env_pop_tf(&(e->env_tf));
    // panic("env_run not yet implemented");
    }
```

下面是调用 *用户代码* 之前的代码调用图。确保你理解每一步的目的。

- 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

一旦你完成了你应该编译你的 `kernel` 并在QEMU下运行它。如果一切都没问题的话，你的系统应该进入用户空间，执行 `hello` 二进制文件直到它用 `int` 指令做了一个系统调用。

当此时这里会出现问题，因为 `JOS` 还没有设置硬件来允许任何种类从用户空间到内核的转换。当CPU发现它没有被设置来处理这个 *系统调用中断* ，它将生成一个普遍的保护exception，又发现不能处理，生成一个 `double fault exception` ，发现自己也不能处理，最终放弃，发出一个 **`triple fault`** 。

通常，你将看到CPU重置，系统重启。虽然这对合法应用程序很重要（详看这个博客），但对于 *内核开发* 很痛苦，所以在打完6.828补丁的QEMU你将会看到一个寄存器dump和一个“Triple fault”信息。

我们将很快解决这个问题，但现在可以使用 *调试器* 来检查是否进入 *用户模式* 。使用 `make qemu-nox-gdb` ，在 `env_pop_tf` 设置一个断点，这应该是在真正进入用户模式前最后一个进入的函数。

用 `si` 单步进入调试，处理应该**在 `iret` 指令之后进入用户模式**。你应该查看在用户环境可执行文件中的第一条指令，第一条指令应该是 `cmpl` 指令，位于 `lib/entry.S` 的 `start` 标志。

现在使用 `b *0x...` 在 `hello` （查看 `obj/user/hello.asm` 来获取 *用户空间地址* ）二进制文件中的 `sys_cputs()` 函数的 `int $0x30` 设置一个断点。这个 `int` 指令是一个系统调用在控制台显示一个字符。

如果你不能执行到 `int` ，你的 *地址空间设置* 或者 *程序加载代码* 中的某些出错了；返回并fix it。

Environment summary

系统对一个环境的创建和运行分为以下这几步：

1. 完成 `envs` 数组内存分配和映射

   在 `pmap.c` 文件中的 `mem_init()` 函数中对初始化 `envs` 数组，用于跟踪系统中的环境。

   使用 `boot_alloc()` 函数为数组分配 *物理空间* ， `boot_map_region()` 函数将分配的物理空间映射到 `kern_pgdir` 页表下的虚拟地址。

   注意： `boot_alloc()` 和 `boot_map_region()` 都只在初始化时使用，后面分别使用 `page_alloc()` 和 `page_insert()` ，或者 `region_alloc()` 。

2. 完成 `envs` 数组的初始化

   在 `kern/env.c` 文件中的 `env_init()` 函数，对 `envs` 数组中每个元素基本属性初始化并加入到 `env_free_list` 。

   另外，该函数中还调用了 `env_init_percpu()` 来加载 `Global Directory Table` 和 *段描述符* 。

3. 创建一个 `environment`

   - 调用 `env_alloc()`

     从刚才初始化的 `env_free_list` 链表中取一个空闲 `env` 。

     第一步对该空闲环境设置 *虚拟地址空间* ，可以将 `kern_pgdir` 的内容直接复制到 `e->env_pgdir` 。

     第二步为该环境生成一个 `env_id` ，将寄存器状态清零，为 `e→env_tf` 结构相关属性初始化合适的值。

   - 调用 **`load_icode()`**

     Only called during kernel initialization.

     该函数将每个程序段加载到 `ELF segment header` 指定的虚拟地址，步骤可以参考 `boot/main.c` 。

     使用 `page_alloc()` 为用户环境分配一个 *栈区* ，并使用 `page_insert()` 插入到用户环境的页表。

注意⚠️

该函数设置的是刚刚创建的环境，所以在加载操作之前，**要先使用 `lcr3()` 函数将 *页表切换为用户环境的页表*，以及在所有操作完成之后要 (再次) 使用 `lcr3()` 将页表切换为 `kern_pgdir()`**。

不要忘记将 `e->env_tf.tf_eip` 初始化为 `ELF->e_entry`，即修改环境运行的入口地址。

- 调用 `env_run()`

Handling Interrrupts and Exceptions

这里，用户空间中第一条系统调用 `int 0x30` 是一个死胡同：一旦处理器进入到 *用户模式*，就不可能返回出来。现在，你需要完成基本的 *异常* 和 *系统调用处理*，这样 `kernel` 才能从 *用户模式* 恢复对处理器的控制。

你需要做的第一件事就是彻底熟悉x86的 *中断* 和 *异常* 机制。

> 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.

Chapter 9 Exceptions and Interrupts

中断 和 *异常* 是控制转换的特殊类型。它们转换正常的程序流来处理 *外部事件* 或报告错误或异常条件。两者的区别在于 *中断* 用于处理处理器之外的 *异步事件*，但是 *异常* 处理由处理器自身检测到的条件。

外部中断和异常分别有两个来源：

1. 中断
   - *可屏蔽中断*，通过INTR引脚发出信号
   - *不可屏蔽中断*，通过NMI(Non-Maskable Interrupt)引脚发出信号
2. 异常
   - 处理器检测。又进一步被分为 *faults, traps, abort...*
   - 程序层面的。指令 `INTO`，`INT3`，`INT n` 和 `BOUND` 能触发异常。这些指令经常被叫做“软件中断”，但是处理器将它们当作异常处理。

9.1 Identifying Interrupts

处理器为每个不同类型的中断和异常关联一个数字。

NMI和异常被分配从0到31的 *预定标识符* ，没有被分配的标识符将被保留。**可屏蔽中断**的标识符由外部中断控制（如Intel's 8259A Programmable Interrupt Controller）决定，并在处理器的中断确认序列中和处理器进行通信。通过8259A PIC分配的数字可由软件指定——从32到255的任何数字都可以被使用。

Table 9-1. Interrupt and Exception ID Assignments

Identifier	Description
0	Divide error
1	Debug exceptions
2	Nonmaskable interrupt
3	Breakpoint (one-byte INT 3 instruction)
4	Overflow (INTO instruction)
5	Bounds check (BOUND instruction)
6	Invalid opcode
7	Coprocessor not available
8	Double fault
9	(reserved)
10	Invalid TSS
11	Segment not present
12	Stack exception
13	General protection
14	Page fault
15	(reserved)
16	Coprecessor error
17-31	(reserved)
32-255	Available for external interrupts via INTR pin

异常被分为 *faults, traps, abort...* ，取决于它们被报告的方式以及是否支持指令重新开始。

- Faults

Faults是在指令致使异常之前被报告。Faults不是在指令开始之前检测到就是在指令执行过程中被检测到。

如果在指令执行期间检测到故障，机器会恢复到允许重新启动指令的状态。

- Traps

Trap是在检测到异常的指令后立即在指令边界报告的异常。

- Aborts

Abort这种异常，既不告知指令的准确位置，也不重新开始导致异常的程序，用于报告严重错误，比如硬件错误和不一致或者系统表中的非法值。

9.2 Enabling and Disabling Interrupts

处理器只在一条指令的结束和下一条指令的开始提供中断和异常服务。某些条件和标志设置会导致处理器在指令边界处禁止某些中断和异常。

9.2.1 NMI Masks Further NMIs

当NMI处理程序正在执行时，处理器将忽略NMI引脚上的进一步中断信号，直到执行下一条IRET指令。

9.2.2 IF Masks INTR

IF(interrupt-enable flag)控制通过INTR引脚发出的外部信号的接收。0，表示不接收；1，表示接收。*RESET* 信号使处理器清空IF，`CLI` 和 `STI` 可以转换IF的设置。

CLI(Clear Interrupt-Enable Flag) 和 STI(Set Interrupt-Enable Flag) 转换IF(bit 9 in flag register)。这些指令可能只有在 `CPL ⩽ IOPL` 时才被执行，否则会产生一个 *保护异常* 。

IF也会被下面的操作所影响：

- `PUSHF` 存储所有在栈中的flags，包括IF
- *任务切换* 和指令 `POPF` 和 `IRET` 加载标志寄存器，因此它们能被用来修改IF
- 通过中断门的中断自动重置IF，禁用中断。（中断门会在这个章节的后面解释）。

9.2.3 RF Masks Debug Faults

EFLAGS中的RF位控制 `debug faults` 的识别。这允许一个给定指令的调试错误只被引发一次，无论指令重新启动多少次。

9.2.4 MOV or POP to SS Masks Some Interrupts and Exceptions

需要修改 *栈* 这个段的软件经常使用一对指令

```
MOV SS, AX
MOV ESP, StackTop
```

如果在 *SS被修改之后* ， *ESP接收到对应修改之前* ，一个中断或者异常被处理，栈指针SS:ESP的两部分在 *中断处理程序* 或 *异常处理程序* 的持续时间内是**不一致**的。

为了阻止这种情况， *80386* ，在对SS进行操作的MOV和POP指令之后，禁止NMI、INTR、调试异常和在指令边界的单步陷阱，这个单步陷阱紧跟着改变SS的指令。一些异常，比如 *page fault* ， *general protection fault* ，可能仍会发生。总是使用80386的LSS指令，问题将不会出现。

```
lss mem, %esp    // 将mem指向内容的前四个字节装入ESP寄存器，后两个字节装入SS段寄存器
```

9.3 Priority Among Simultaneous Interrupts and Exceptions

如果不止一个中断或者异常在指令边界正等待处理，处理器在一个时间只处理其中一个，处理的**优先级**在Table 9-2中会给出。处理器首先处理来自最高优先级的中断或异常，将 *控制* 转移给 *中断处理程序* 的第一个指令。

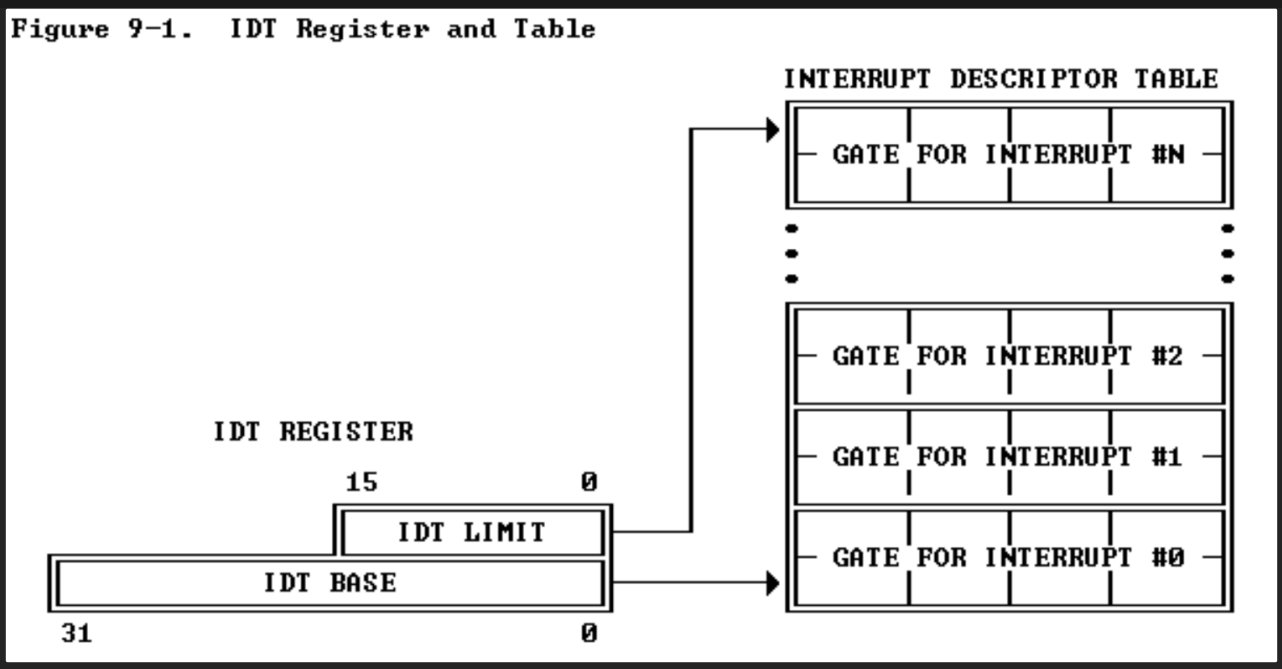
较低优先级的**异常**将被摒弃，较低优先级的**中断**将**持续等待处理**。当 *中断处理程序* 返回到中断点时，将重新发现被丢弃的异常。

9.4 Interrupt Descriptor Table

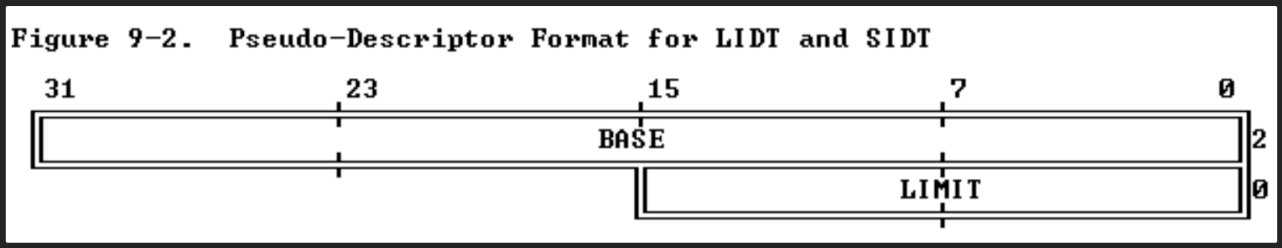
IDT(Interrupt descriptor table) 将每个中断或异常的 *标识符* 和一个服务于关联事件指令的 *描述符* 关联起来。和GDT和LDTs一样，IDT是一个8字节描述符的数组。和GDT，LDTs不一样的是，IDT的第一个入口可能包含一个描述符。

为了生成IDT的索引，处理器将中断或者异常的标识符乘以8。由于只有256个标识符，IDT不需要包含256个描述符。他可以包含少于256个条目；这些条目仅当中断标识符被真正使用时才需要。

IDT可能会在物理内存的任何地方。如图9-1所示，处理器通过IDT寄存器(IDTR)来定位IDT。



指令 `LIDT` 和 `SIDT` 对IDTR进行操作，两个命令都有一个显式操作数：内存中一个6字节区域的地址。图9-2展示了这块区域的格式。



接下来介绍 `LIDT` 和 `SIDT` 这两条指令：

- LIDT(Load IDT register)

加载IDT寄存器，操作数中包含线性基地址和limit value。

这个指令只有**当CPL是0**时才能被执行。通常在创建一个IDT时被一个操作系统的初始化逻辑所使用。一个操作系统可能也会使用它来改变一个IDT到另一个。

- SIDT(Store IDT register)

复制IDTR中的base和limit value到一个内存位置。该指令可以在任何权限等级下执行。

Table 9-2. Priority Among Simultaneous Interrupts and Exceptions

Priority	Class of Interrupt or Exception
HIGHEST	Faults except debug faults
	Trap instructions INTO, INT n, INT 3
	Debug traps for this instruction
	Debug faults for next instruction
	NMI interrupt
LOWEST	INTR interrupt

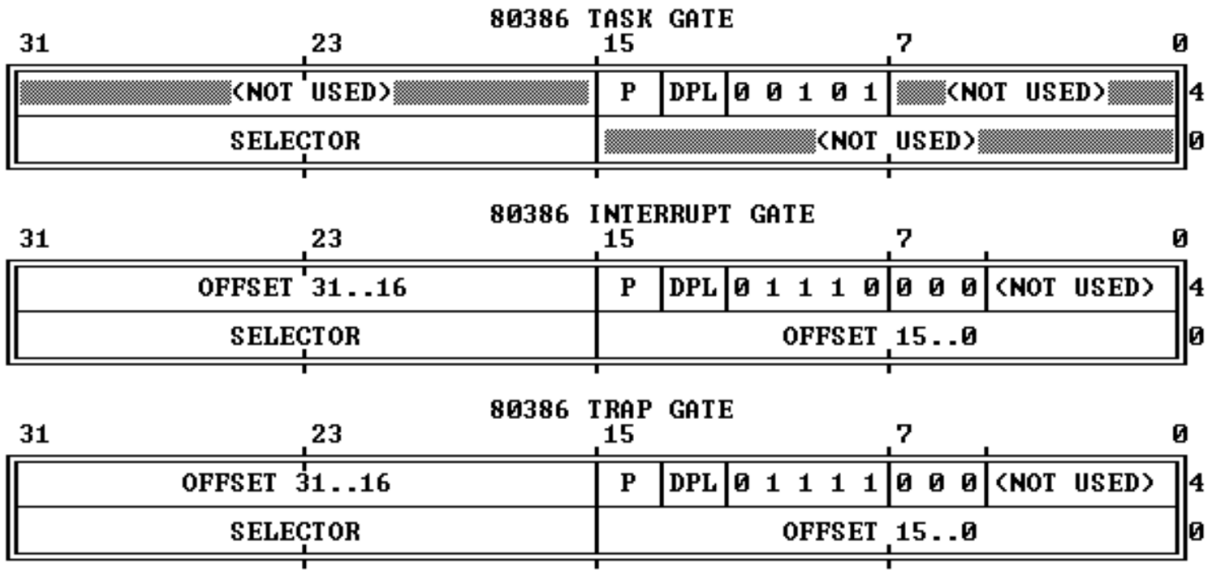
9.5 IDT Descriptor

IDT可能包含三种描述符:

- Task gates
- Interrupt gates
- Trap gates

图9-3展示了task gates, 80386 interrupt gates, trap gates的格式。

Figure 9-3. 80306 IDT Gate Descriptors



Interrupt Tasks and Interrupt Procedures

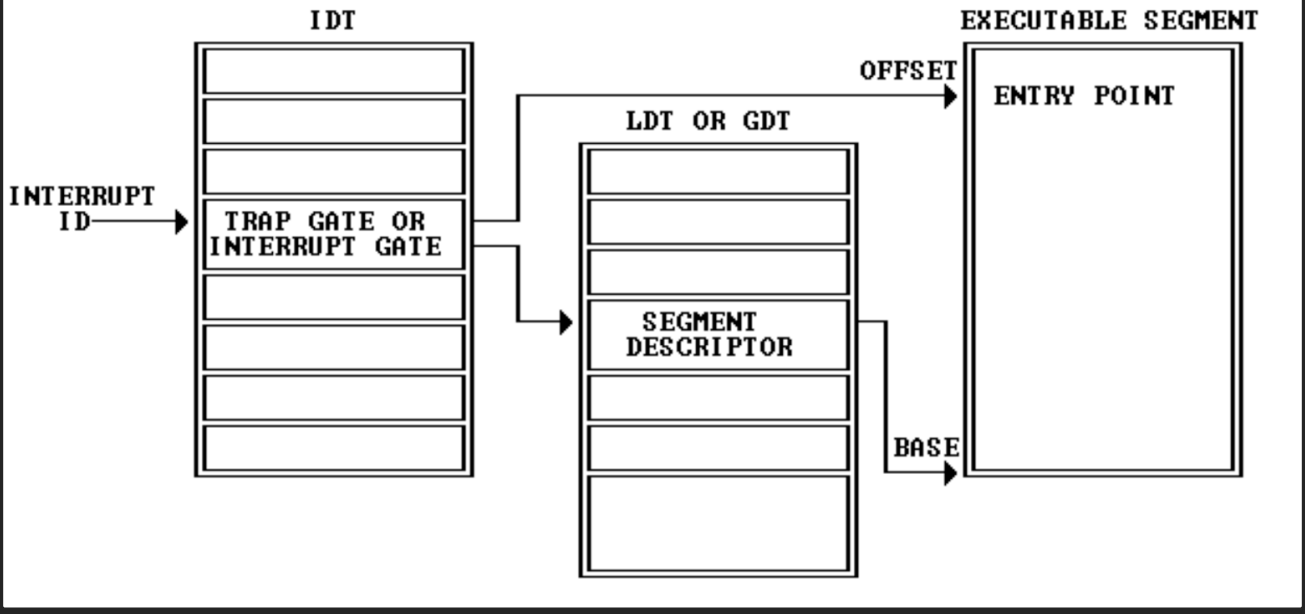
就像CALL指令能调用一个进程或一个任务，一个中断或者异常也能“调用”一个中断处理程序，这个中断处理程序也是一个进程或一个任务。

处理器使用一个中断或者异常 *标识符* 来索引IDT中的描述符，进而对中断或异常做出回应。如果处理器索引到一个 *中断门* 或 *trap gate* ，它唤醒处理程序的方式与CALL指令调用gate相似。如果处理器发现一个 *task gate* ，会致使一个任务切换，方式类似于CALL指令调用一个 *task gate* 。

9.6.1 Interrupt Procedures

如图9-4所示，一个 *interrupt gate* 或 *trap gate* 直接指向一个进程，这个进程将会在当前执行任务的上下文中执行。

Figure 9-4. Interrupt Vectoring for Procedures

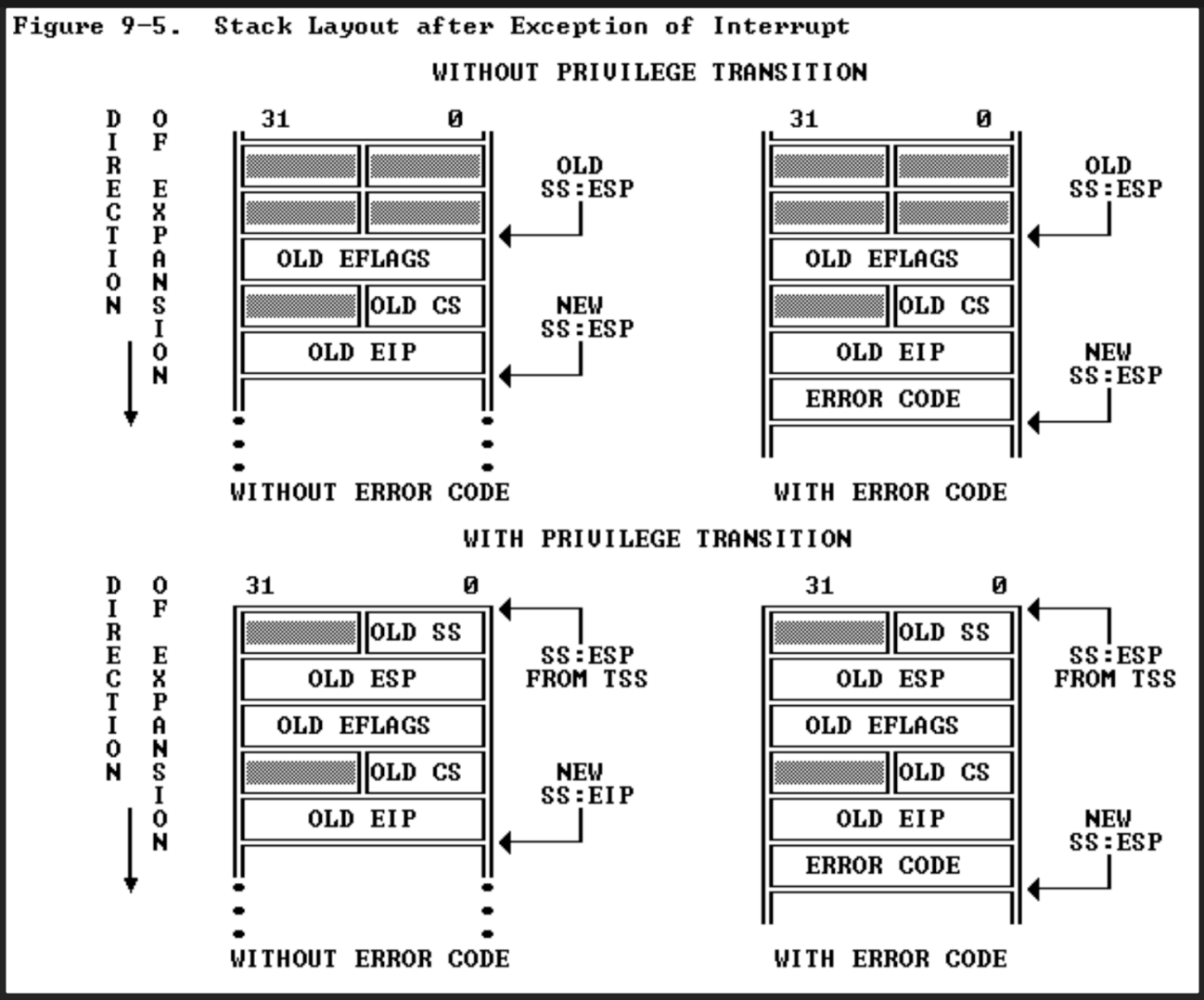


gate的选择器指向GDT或当前LDT中的 *可执行段描述符* 。gate的offset字段指向中断或异常处理程序的开始。

80386 唤醒一个中断或异常处理程序很大程度上相似于CALL指令调用进程；不同之处在之后的章节会解释。

9.6.1.1 Stack of Interrupt Procedure

和CALL指令一样，将控制转移到中断或异常处理程序时，需要使用栈来存储返回到原始进程的信息。如图9-5所示，在指针指向中断指令之前，中断将EFLAGS寄存器推向栈。



某些类型的异常也会导致错误代码被压入堆栈。异常处理程序可以用错误代码来帮助诊断异常。

9.6.1.2 Returning from an Interrupt Procedure

一个中断程序离开进程的方法和一个普通的程序也是不一样的。指令 `IRET` 用于从中断程序中退出。 `IRET` 和 `RET` 相似，不同的是 `IRET` 会将esp加上额外四个字节（因为栈中的flags）并且将保存的flags移入EFLAGS寄存器。EFLAGS的 `IOPL` 字段只在CPL是0的时候被改变。*IF标志只当CPL ≤ IOPL时才能被修改*

9.6.1.3 Flags Usage by Interrupt Procedure

通过*interrupt gates*和*trap gates*向量化的中断在将当前 `TF` 值存储到栈上作为EFLAGS的一部分之后，会把 `TF(the trap flag)` 重置。通过这个动作，处理器可防止使用单步执行的调试活动影响中断响应。随后的 `IRET` 指令将TF恢复为存储在栈上的EFLAGS中毒呢对应值。

interrupt gate 和 *trap gate* 的区别就是对 `IF(the interrupt-enable flag)` 的影响。通过 *interrupt gate* 向量化的中断**重置**IF，以此组织其它中断干扰当前 *中断处理程序* 。随后的 `IRET` 指令将TF恢复为存储在栈上的EFLAGS中毒呢对应值。通过 *trap gate* 向量化的中断**不改变**IF。

9.6.1.4 Protection in Interrupt Procedures

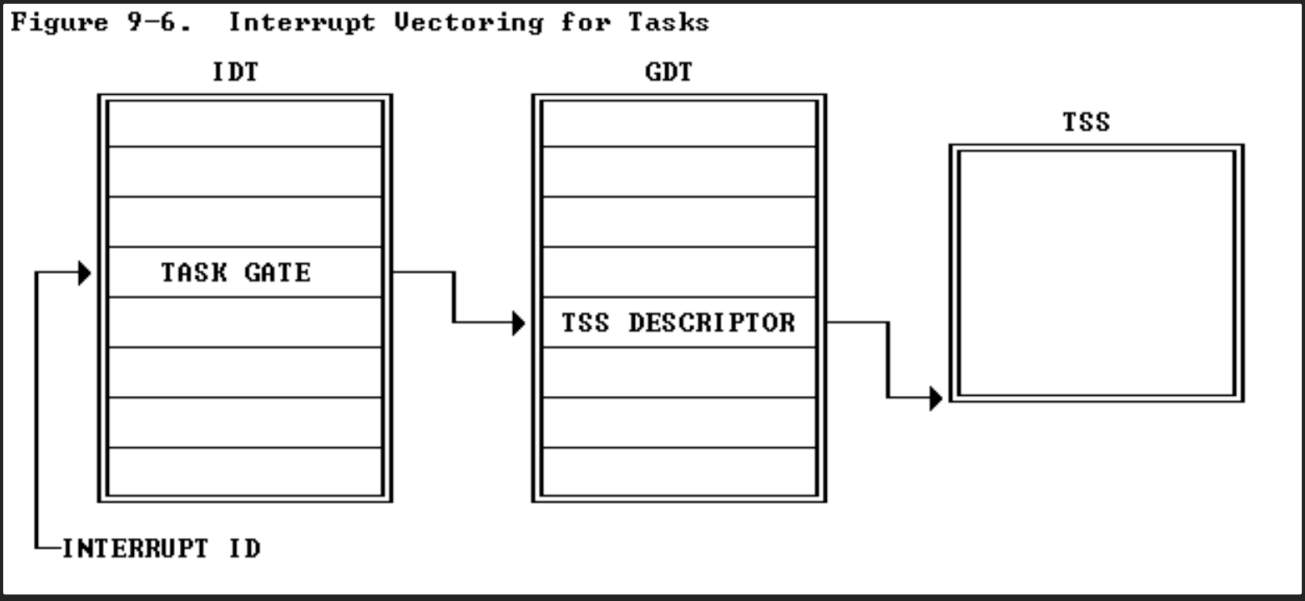
管理中断程序的privilege rule和进程调用的相似：CPU不允许中断将 *控制* 转移给一个比当前 *特权等级* 更低的进程中的段。如果尝试违反这个规则会导致普遍的保护异常。

因为中断的发生不可预测，这个特权规则有效地对 *特权等级* 施加限制，并可以被中断和异常处理程序执行。以下策略的其中之一可以确保 *特权规则* 不被违反：

- 将处理程序放在一致的段中。该策略适合一些异常的处理，如 *divide error* 。这样的处理程序只能使用栈上可获取的数据。如果需要使用来自data segment的数据，data segment将必须有 *特权等级3* ，从而使其不受保护。
- 将处理程序放入一个 *特权等级* 为0的段中。

9.6.2 Interrupt Tasks

如图9-6所示，一个task gate直接指向一个任务。selector指向GDT中的TSS描述符。



当中断或异常向量指向IDT中的 *task gate* ，会导致任务切换。用分离的task处理中断，有以下两点优势：

- 整个上下文被自动保存
- *中断处理程序* 可以从其它task分隔开，通过给它一个独立的地址空间，或使用它的LDT或使用它的页目录

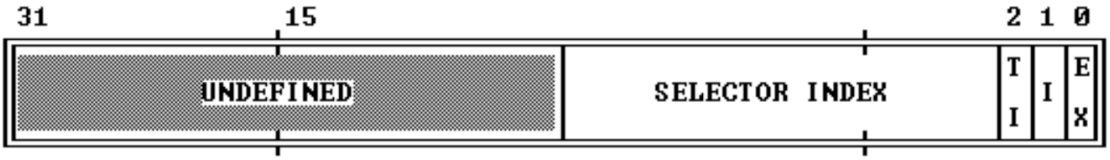
处理器执行任务切换的动作已经在Chapter 7中讨论过。中断任务返回被中断的任务通过执行 `IRET` 指令。

当 *中断任务* 在80386的操作系统中被使用，实际上有两种调度器： *软件调度器* （操作系统的一部分）和 *硬件调度器* （处理器中断机制的一部分）。 *软件调度器* 的设计应该考虑到 *硬件调度器* 可能会分派一个 *中断任务* 在中断被启用的任何时候。

9.7 Error Code

有了和特定段关联的异常，处理器将error code推入异常处理程序(whether procedure or task)的栈中。Error code格式如图9-7所示。

Figure 9-7. Error Code Format



格式和selector的格式相似；然而，error code包含两个一位的项，而不是一个RPL字段。

- 处理器设置一个 *EXT* 位，如果是程序外部的事件导致异常。
- 处理器设置一个 *I-bit(IDT-bit)* ，如果error code的索引部分指向IDT中的gate描述符。

如果 *I-bit* 没有设置， *TI位* 表明error code指向GDT(value 0)或IDT(value 1)。剩下的14位是所涉及段选择符的高14位。某些情况下栈上的error code 为null，如 all bit in the low-order word are zero.

9.8 Exception Condition

接下来详细介绍了每个可能的异常条件。每个描述将异常分为 *fault, trap, abort* 。此分类提供信息，该信息是系统程序在发生异常的地方重启进程所需要的。

- Faults

当一个fault被报告，CS和EIP的值被保存指向导致这个fault的指令。

- Traps

当trap被报告，CS和EIP的值被保存动态指向导致trap的指令之后的指令。

如果trap在指令中被检测并转换了程序流，被报告的CS和EIP的值会反应出程序流的转换。比如，在 `jmp` 指令中检测到trap，CS和EIP的值被推入栈中，指向 `jmp` 的目标，而不是 `jmp` 指令的下一条指令。

- Aborts

abort异常既不会提供导致异常指令的精确位置，也不会重启导致异常的程序。Aborts用于报告严重错误，比如 *硬件错误* ， *不一致* 或 *系统表中的非法值* 。

- Interrupt 0 -- Divide Error
- Interrupt 1 -- Debug Exceptions
- Interrupt 3 -- Breakpoint
- Interrupt 4 -- Overflow
- Interrupt 5 -- Bounds Check
- Interrupt 6 -- Invalid Opcode
- Interrupt 7 -- Coprocessor Not Available
- Interrupt 8 -- Double Fault
- Interrupt 9 -- Coprocessor Segment Overrun
- Interrupt 10 -- Invalid TSS
- Interrupt 11 -- Segment Not Present
- Interrupt 12 -- Stack Exception
- Interrupt 13 -- General Protection Exception
- Interrupt 14 -- Page Fault
- Interrupt 16 -- Coprocessor Error

9.9 Exception Summary

Table 9-6 summarizes the exceptions recognized by the 386.

Table 9-6. Exception Summary

Description	Interrupt Number	Return Address Points to	Exception Type	Function That Can Generate the Exception
Divide error	0	YES	FAULT	DIV, IDIV
Debug exceptions	1			Faulting Instruction

```
Some debug exceptions are traps and some are faults.  The
exception
handler can determine which has occurred by examining DR6.  (Refer
to Chapter 12.)
Some debug exceptions are traps and some are faults.  The
exception
handler can determine which has occurred by examining DR6.  (Refer
to Chapter 12.) Any instruction
Breakpoint                3          NO             TRAP
 One-byte INT 3
Overflow                  4          NO             TRAP
 INTO
Bounds check              5          YES            FAULT
  BOUND
Invalid opcode            6          YES            FAULT
  Any illegal instruction
Coprocessor not available 7          YES            FAULT
  ESC, WAIT
Double fault              8          YES            ABORT
  Any instruction that can
generate an exception
Coprocessor Segment
Overrun                   9          NO             ABORT
  Any operand of an ESC
instruction that wraps around
the end of a segment.
Invalid TSS               10         YES            FAULT
An invalid-TSS fault is not restartable if it occurs during the
processing of an external interrupt.       JMP, CALL, IRET, any
interrupt
Segment not present       11         YES            FAULT
  Any segment-register modifier
Stack exception           12         YES            FAULT
  Any memory reference thru SS
General Protection        13         YES            FAULT/ABORT
All GP faults are restartable. If the fault occurs while
attempting to
vector to the handler for an external interrupt, the interrupted
program is
```

restartable, but the interrupt may be lost. Any memory reference or code
fetch
Page fault 14 YES FAULT
Any memory reference or code
fetch
Coprocessor error 16 YES FAULT
Coprocessor errors are reported as a fault on the first ESC or WAIT
instruction executed after the ESC instruction that caused the error. ESC, WAIT
Two-byte SW Interrupt 0-255 NO TRAP
INT n

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
```

在这个实验我们会使用英特尔对 *interrupts, exceptions...* 的专业用语。

Basics of Protected Control Transfer

异常和中断都是“受保护的控制转移”，会使处理器从用户模式切换到 *核模式(CPL=0)* ，这样用户模式的代码不会有任何机会影响 `kernel` 或其它环境的功能。

在英特尔的用语中， *interrupt* 是受保护的控制转移，由一个 *异步事件* 导致（通常是处理器的外部事件），比如外部设备I/O活动的通知；而 *exception* 相反，是受保护的控制转移，由一个 *同步事件* （当前正在运行的代码）导致，比如 *除以零* 或无效的内存访问。

为了确保这些受保护的控制转移确实是受保护的，处理器的中断/异常机制被设计用来使得当中断/异常发生的时候当前正在运行的代码*不会随意选择进入 `kernel` 的那部分或是怎么进入 `kernel` 。相反，处理器确保 `kernel` 只能在严格控制的条件下被进入。

在x86，两个机制共同工作来提供保护：

1. **The Interrupt Descriptor Table**

   处理器确保中断和异常只能在**一些由 `kernel` 本身决定的特定的，定义好的进入点**进入 `kernel` ，而不是由中断异常发生的正在运行的代码决定。

   x86 允许256个不同的中断或异常进入点，每个都有不同的 *中断向量* （0-255的数字）。一个中断向量由中断源决定：不同的设备，错误条件和 应用需求生成有着不同向量的中断。CPU使用向量作为 *索引* 来访问处理器的 *interrupt descriptor table* (IDT)，处理器在 `kernel-private` 内存中设置该表，和GDT很相似。从表中合适的表项，处理器加载一下信息：

   - 加载进入 *instruction pointer register* (EIP)的值，指向处理该异常类型的 *内核代码*
   - 加载进入 *code segment register* (CS)的值，包含了异常处理程序运行的特权等级（0/1）。（在 `JOS` 中，所有的异常在 `kernel mode` (privilege 0)中处理

2. **The Task State Segment**

   在中断或异常发生之前，处理器需要一个地方存储“之前的”处理器状态，如在唤醒异常处理程序之前 *EIP* 和 *CS* 的原始值，所以异常处理程序可以之后恢复并继续。**但是**，保存的区域反过来**必须不受非特权用户模式代码影响**；否则，错误或恶意用户代码可能危害内核。

   因此，当x86处理器处理导致特权等级从用户模式切换到内核模式的interrupt或trap时，同时也会切换到内核内存的栈。 *task state segment* (TSS)结构指定 *段选择器* 和 *该栈的地址* 。处理器将 *SS, ESP, EFLAGS, CS, EIP 和有选择的错误代码* 推入这个新栈。然后它从 *中断描述符* 加载 *CS* 和 *EIP* ，并且设置 *ESP* 和 *SS* 指向新栈。

尽管TSS很大并可以为不同目的服务，`JOS`只用它来定义`kernel stack`，这个栈是当处理器从用户模式转向核模式时应该转入的。因为JOS中的`kernel model`在x86特权等级为0，处理器使用`ESP0`和`SS0`字段来定义`kernel stack`。`JOS`不使用TSS任何其它字段。

Types of Exceptions and Interrupts

所有x86处理器能内部生成的*同步异常*使用0-31的中断向量，因此映射到IDT条目0-31。例如，*page fault*总是通过向量14来导致异常。大于31的中断向量只由*软件中断*使用，可以被`int`指令生成，或者异步的*硬件中断*（由外部设备导致）。

这个部分，我们将扩展`JOS`来处理内部生成的x86异常（0-31向量）。下一章节，将使`JOS`处理中断向量为48(0x30)的软件中断，该中断是`JOS`用来作为其系统调用的中断向量（随意选取）。在lab 4中，将扩展`JOS`处理外部生成的*硬件中断*如*时钟中断*。

An Example

让我们将这些知识放在一起，跟踪一个例子。

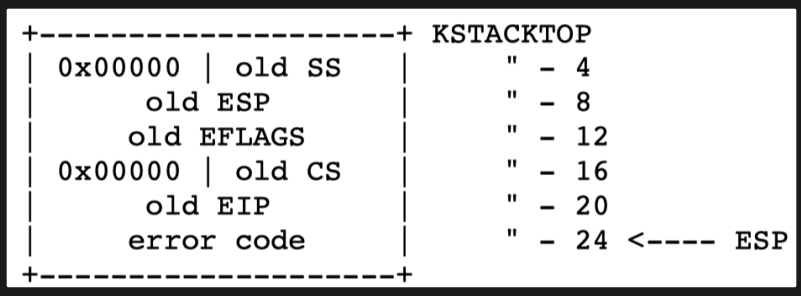
假设处理器正在一个用户环境执行代码，并且遇到尝试除以0的指令。

1. 处理器切换到TSS寄存器`SS0`和`ESP0`字段定义的栈，`SS0`和`ESP0`在`JOS`分别保存着值`GD_KD`和`KSTACKTOP`。
2. 处理器将异常参数推到`kernel stack`上，栈开始于地址`KSTACKTOP`

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

3. 因为我们正在处理的是*divide error*，在x86中它的中断向量是0。所以，处理器读取IDT条目0，设置`CS:EIP`指向条目所描述的处理函数。
4. 处理函数拿到控制并且处理异常，比如终止用户环境。

一些x86的异常，除了推入栈中“标准的”5个字以外，处理器还会推入另一个字包含一个`error code`。



参看 *80386 manual* 来决定哪个异常数字会推入 *错误代码* ，以及 *错误代码* 意味着什么。当处理器推入一个 *错误代码* ，栈在异常处理程序开始的地方看起来如上图。

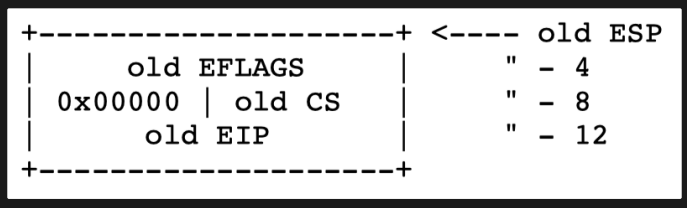
Nested Exception and Interrupts

处理器可以从kernel和user模式获取异常和中断。

然而，**只有从用户模式进入核模式**，x86处理器会自动地切换栈，时间在将 *old register state* 推入栈和从IDT唤醒合适的异常处理程序之前。

如果中断或异常发生时，处理器已经在核模式，CPU只会将更多的值推入相同的 `kernel stack` 。采用这种方式，kernel可以优雅地处理 *nested exceptions* （由kernel本身的代码所导致）。这种能力是实现保护很重要的工具，在后面 *system calls* 的章节中可以看到。

如果处理器已经在核模式并且收到了一个 *nested exception* ，由于不需要切换栈，就不用保存 `old SS/ESP` 。对于不需要推入 `error code` 的异常类型， `kernel stack` 如下图所示



若要推入 *错误代码* ，处理器在 *old EIP* 之后立即推入。

这里对于处理器处理 *nested exception* 能力有一个警告。如果已经在核模式的情况下，处理器遇到一个异常，并且因为某些原因（栈中没有空间），**无法将其 *old state* 推入 `kernel stack`** ，那么处理器就无法恢复，它就会简单地重置自己。不用去说，我们的设计应该避免这个问题。

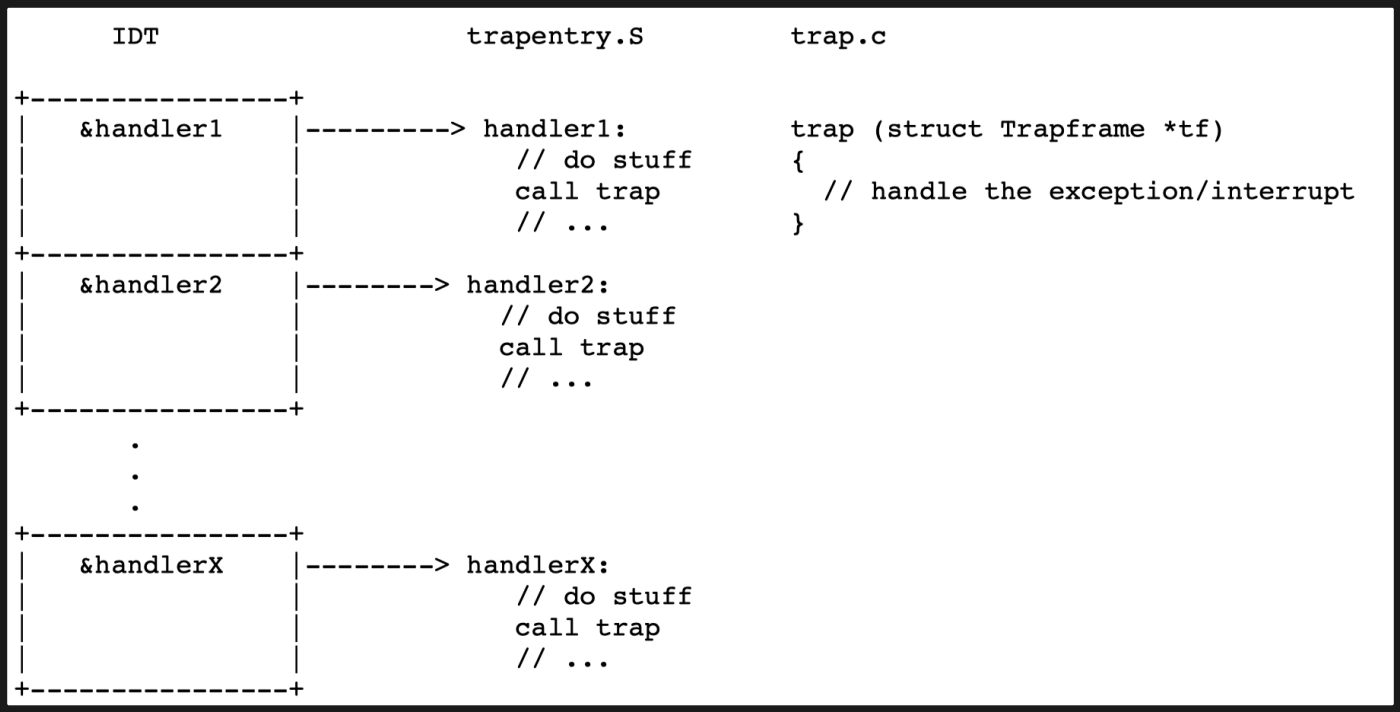
Setting Up the IDT

现在你将设置IDT来处理0-31的中断向量（处理器异常）。我们将在稍后处理 *系统调用* ，在之后的lab中会加入中断32-47（the device IRQs）。

头文件 `inc/trap.h` 和 `kern/trap.h` 包含了与中断异常相关的重要定义，你需要去熟悉。文件 `kern/trap.h` 包含了kernel严格私有的定义，而 `inc/trap.h` 包含的是对用户级程序或者库可能也有用的定义。

⚠️：0-31中的有些异常是被保留的。由于它们根本不会被处理器生成，你如何处理该中断不重要。

你应该达到的整个控制流描述如下：



每个中断或异常在 `trapentry.S` 中都应该有他自己的处理程序， `trap_init()` 应该初始化IDT中这些处理程序的地址。每个处理程序应该创建一个 `struct Trapframe` (see `inc/trap.h`)在栈上并且使用Trapframe指针作为参数调用 `trap()` (in `trap.c`)，然后 `trap()` 处理中断/异常，或分配给一个指定的处理函数。

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.

`kern/trapentry.S`

```
/* 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

/*
 * 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_coproc, T_COPROC);     reserved
TRAPHANDLER(t_tss, T_TSS);
TRAPHANDLER(t_segnp, T_SEGNP);
TRAPHANDLER(t_stack, T_STACK);
TRAPHANDLER(t_gpflt, T_GPFLT);
TRAPHANDLER(t_pgflt, T_PGFLT);

TRAPHANDLER_NOEC(t_fperr, T_FPERR);
```

```
TRAPHANDLER(t_align, T_ALIGN);

TRAPHANDLER_NOEC(t_mchk, T_MCHK);
TRAPHANDLER_NOEC(t_simderr, T_SIMDERR);



/*
 * 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        /* 这里的esp指向的是pushal压入栈的最后一个寄存器
    call trap
```

kern/trap.c

```
static struct Taskstate ts;

  /* For debugging, so print_trapframe can distinguish between
printing
   * a saved trapframe and printing the current trapframe and
print some
   * additional information in the latter case.
   */
  static struct Trapframe *last_tf;

  /* Interrupt descriptor table.  (Must be built at run time
because
   * shifted function addresses can't be represented in relocation
records.)
   */
struct Gatedesc idt[256] = { { 0 } };
struct Pseudodesc idt_pd = {
```

```
    sizeof(idt) - 1, (uint32_t) idt
  };
```

`struct Gatedesc` & `struct Pseudodesc`

```
// Gate descriptors for interrupts and traps
  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
  };

// Pseudo-descriptors used for LGDT, LLDT and LIDT instructions.
  struct Pseudodesc {
      uint16_t pd_lim;        // Limit
      uint32_t pd_base;       // Base address
  } __attribute__ ((packed));
```

`trap_init()`

在该函数中，应该对中断处理函数进行定义。并初始化每个 `Gatedesc` 。

```
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();

        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, 0);
        SETGATE(idt[T_OFLOW], 0, GD_KT, &t_oflow, 0);
        SETGATE(idt[T_BOUND], 0, GD_KT, &t_bound, 0);
        SETGATE(idt[T_ILLOP], 0, GD_KT, &t_illop, 0);
        SETGATE(idt[T_DEVICE], 0, GD_KT, &t_device, 0);
        SETGATE(idt[T_DBLFLT], 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_gpflt, 0);
        SETGATE(idt[T_PGFLT], 0, GD_KT, &t_pgflt, 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);


        // Per-CPU setup
```

```
      trap_init_percpu();
  }
```

SETGATE

```
// 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;       \
  }
```

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

现在你的kernel已经有了基本的处理异常能力，你将改进它来提供依赖于异常处理的重要 *操作系统原语*

Handling Page Faults

页中断异常，中断向量14，是一个尤其重要的异常，将贯穿这个lab和之后的lab。

当处理器遇到一个 *page fault*，首先将导致fault的 *线性地址* 存储到一个特殊的处理器控制寄存器，`CR2`。在 `trap.c`，我们已经提供了特殊函数的开始，`page_fault_handler()`，来处理 *page fault* 异常。

> **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.

在你实现 *系统调用* 时，你将进一步改进kernel的 *page fault* 处理程序。

The Breakpoint Exception

breakpoint exception，中断向量3(T_BRKPT)，一般用于允许调试器在程序代码中插入一个 *breakpoint*，通过临时用特殊一字节的软件中断指令 `int3` 来替换相关的程序指令。
在 `JOS` 中，我们会稍微滥用这个异常，通过将其变成一个原始的 *系统伪调用*，这个伪调用可以被任何用户环境用于唤醒 `JOS kernel monitor`。如果将 `JOS kernel monitor` 视为一个原始的调试器，这个用法在某种程度上是合适的。

举个例子：`lib/panic.c` 中 `panic()` 的用户模式实现，就是在显示了错误信息后执行了一个 `int3` 命令。

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.

`trap_dispatch()`

```
sta static void trap_dispatch(struct Trapframe *tf)
trap_dispatch(struct Trapframe *tf)
{
    // Handle processor exceptions.
    // LAB 3: Your code here.
    switch(tf->tf_trapno)
    {
        case T_PGFLT:
            page_fault_handler(tf);
            return;
        case T_BRKPT:
            monitor(tf);
            return;
        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;
            }
    }
}
```

System calls

用户进程通过 *系统调用* 让内核做事。当用户进程唤醒一个 *系统调用* ，进程进入核模式。 *进程* 和 *内核* 协作保存用户进程的状态，内核执行合适的代码来执行系统调用，然后继续用户进程。更多细节关于用户进程如何引起kernel的注意，如何指定它想要执行的调用，这些细节都是因系统而异。

在 `JOS` 核中，我们将使用 `int` 指令，该指令导致一个 *处理器中断* 。特别的，我们将使用 `int $0x30` 来作为系统调用中断。我们已经为你定义了常数 *T_SYSCALL* 为48(0x30)。你将设置中断描述符来允许用户进程引发该中断。注意，中断 `0x30` 不能被硬件生成，所以没有由用户代码生成的歧义。

应用将会传递 *系统调用数字* 和 *系统调用参数* 在寄存器中。这样一来，kernel将不需要在用户环境的栈中或者指令流中四处寻找。系统调用数字将存储于 `%eax` ，而系统调用参数（最多五个）将存储于 `%edx` ， `%ecx` ， `%ebx` ， `%edi` ， `%esi` 。kernel将返回值存放在 `%eax` 。唤醒系统调用的汇编代码已经写好了，位于 `lib/syscall.c` 中的 `syscall()` 。你应该仔细阅读，确保你理解发生了什么。

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.

`syscall()` in `lib/syscall.c`

```
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;

   // Generic system call: pass system call number in AX,
   // up to five parameters in DX, CX, BX, DI, SI.
   // Interrupt kernel with T_SYSCALL.
   //
   // The "volatile" tells the assembler not to optimize
   // this instruction away just because we don't use the
   // return value.
   //
   // The last clause tells the assembler that this can
   // potentially change the condition codes and arbitrary
   // memory locations.

   asm volatile("int %1\n"
                : "=a" (ret)
              : "i" (T_SYSCALL),
                "a" (num),
                "d" (a1),
                "c" (a2),
                "b" (a3),
                "D" (a4),
                "S" (a5)
              : "cc", "memory");

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

      return ret;
}
```

`inc/syscall.h`

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

`trap_dispatch()` in `kern/trap.c`

```
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);
            return;
        case T_BRKPT:
            monitor(tf);
            return;
        case T_SYSCALL:
        // 注意这里 `syscall()` 的返回值要赋给 tf->tf_regs.reg_eax
            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;
        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;
            }
    }
}
```

`syscall()` in `kern/syscall.c`

```
// 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");

    switch (syscallno) {
    case SYS_cputs:
        sys_cputs((const 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;
    }
}
```

User-mode startup

一个用户程序在 `lib/entry.S` 的顶端开始运行。在一些设置完成之后，代码会调用 `lib/libmain.c` 中的 `libmian()` 。你应该修改 `libmian()` 函数来初始化全局指针 `thisenv` 指向 `envs[]` 数组中该环境的 `struct Env` 。（注意到， `lib/entry.S` 已经定义了 `envs` 指向 `UENVS` ，这个 `UENVS` 映射到你在Part A中的设置。）

Hint: 查看 `inc/env.h` 并使用 `sys_getenvid` 。

在hello程序的例子中， `libmain()` 然后调用 `umain` ， `umain` 位于 `user/hello.c` 中。注意到，在打印完“hello, world"之后，它尝试去访问 `thisenv->env_id` 。这就是为什么它更早就fault的原因。注意到，你已经初始化完 `thisenv` ，现在它不应该fault。如果它仍然fault，你应该没用正确的将 `UNEVS` 映射到*用户可读* 的区域（检查Part A中的 `pmap.c` ）。

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.

`lib/libmain.c`

```
// Called from entry.S to get us going.
// entry.S already took care of defining envs, pages, uvpd, and
uvpt.

  #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.
  envid_t envid = sys_getenvid();
      thisenv = &envs[ENVX(envid)];

   // 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();
}
```

Page faults and memory protection

内存保护是一个操作系统很重要的特性，该特性能确保一个程序中的bug不会影响到别的程序或操作系统本身。

操作系统一般依赖硬件支持来实现内存保护。操作系统保存着硬件信息，这些信息包括哪个虚拟地址是有效的，哪个虚拟地址是无效的。

当一个程序尝试去访问无效的地址或者其没有访问权限的地址，处理器就会阻止该程序，是程序停在导致错误的指令然后带着有关操作的信息 *陷入内核* 。如果fault是固定的，kernel会修复它并让程序继续运行；若fault不是固定的，程序将不会继续，因为其无法越过导致的fault的指令。

将一个 *自动扩展的栈* 视为 *固定fault* 的例子。在许多系统中，kernel刚开始会分配单个stack page，如果一个程序因为访问该栈之后的page而fault，kernel就会自动分配pages，让程序继续运行。这样一来，kernel只会分配程序所需要的栈内存大小，但是程序可以认为它拥有任意大小的栈。

系统调用 时出现了一个有趣的内存保护问题。大多数的系统调用接口让用户程序给kernel传递指针。这些指针指向用户被读写的缓冲区。kernel在执行 *系统调用* 的时候解引用这些指针，这里会有两个问题：

1. kernel中的*page fault* 比用户程序中的*page fault* 要严重得多。如果kernel在操作它

自己的数据结构时发生*page faults*，那就是一个*kernel bug*，fault处理程序应该panic该kernel（也就是整个系统）。但是，当kernel正在解引用用户程序传递指针时，它需要记住任何解引用导致的*page fault*都用户程序导致的。

2. 一般来说，kernel拥有比用户程序更多的*内存权限*。用户程序可能传递一个*系统调用*的指针指向一个内存，kernel可以读写该内存，当用户程序却不可以。kernel必须谨慎不能被诱导处理那样的指针，因为那样可能会暴露*隐私信息*或破坏kernel的*完整性*。

这两个原因都使得kernel在处理用户程序传递的指针时要非常小心。

你现在将用一个机制来解决这两个问题，该机制会*仔细观察从用户空间传入kernel的所有指针*。当一个程序向内核传入一个指针，kernel将会检查该地址是否在地址空间的用户部分以及*页表*是否允许该内存操作。

因此，kernel将从不会因为解引用一个用户提供的指针而page fault。如果kernel确实page fault，应该panic并且terminate。

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.

`user_mem_check()` in `kern/pmap.c`

```
// Check that an environment is allowed to access the range of
memory
// [va, va+len) with permissions 'perm | PTE_P'.
// Normally 'perm' will contain PTE_U at least, but this is not
required.
// 'va' and 'len' need not be page-aligned; you must test every
page that
// contains any of that range.  You will test either 'len/PGSIZE',
// 'len/PGSIZE + 1', or 'len/PGSIZE + 2' pages.
//
// A user program can access a virtual address if (1) the address
is below
// ULIM, and (2) the page table gives it permission.  These are
exactly
// the tests you should implement here.
//
// If there is an error, set the 'user_mem_check_addr' variable to
the first
// erroneous virtual address.
//
// Returns 0 if the user program can access this range of
addresses,
// and -E_FAULT otherwise.
//
int
user_mem_check(struct Env *env, const void *va, size_t len, int
perm)
{
// LAB 3: Your code here.
uint32_t va_uint = (uint32_t)va;
uint32_t start = (uint32_t)ROUNDDOWN(va, PGSIZE);
uint32_t end = (uint32_t)ROUNDUP((va + len), PGSIZE);
pte_t* addr;
while(start < end){
    // ⚠️ 这里要调用 `pgdir_walk()` 和直接使用 env→env_pgdir[] 可能还
是不一样
```

```
    addr = pgdir_walk(env->env_pgdir, (void*)start, 0);
    // ⚠️ 这里要对判断条件的顺序进行修改
    // 如果addr=0，使用 *addr 就会发生页错误，所以 NULL==addr 应该放在
*addr 之前
   // if(ULIM <= start || !(*addr & PTE_U) || (*addr & perm) !=
perm || NULL == addr){
     if(ULIM <= start || NULL == addr || !(*addr & PTE_U) ||
(*addr & perm) != perm){
        user_mem_check_addr = (start < va_uint) ? va_uint : start;
        return -E_FAULT;
    }
    // addr = PTE_ADDR(env->env_pgdir[PDX(start)]) + PTX(start);
    // if(ULIM <= start || !(addr & PTE_U) || (addr & perm) !=
perm){
    //     user_mem_check_addr = (start < va_uint) ? va_uint :
start;
    //     return -E_FAULT;
    // }
    // if(ULIM <= start){
    //     user_mem_check_addr = (start < va_uint) ? va_uint :
start;
    //     return -E_FAULT;
    // }
    // addr = PTE_ADDR(env->env_pgdir[PDX(start)]) + PTX(start);
    // if(!(addr & PTE_U) || (addr & perm) != perm){
    //     user_mem_check_addr = (start < va_uint) ? va_uint :
start;
    //     return -E_FAULT;
    // }
    // if(!(env->env_pgdir[PDX(start)] & (perm | PTE_P))){
        // user_mem_check_addr = start;
        // return -E_FAULT;
    // }
    start += PGSIZE;
}
return 0;
}
```